Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Proyecto de Grado

Herramientas del sistema operativo para combatir el Software Aging

Daniel G. Pedraja, Fabricio S. González, Agustin Van Rompaey

Tesis

14 de Diciembre de 2009
Montevideo - Uruguay

Tribunal:

Gerardo Ares
Héctor Cancela
Jorge Merlino

Tutores:

Andrés Aguirre
Ariel Sabiguero

Resumen. El hardware falla, no podemos hacer nada al respecto. Diferentes causas hacen que los programas y datos que nosotros almacenamos en medios digitales no sean estáticos y se degraden con el paso del tiempo. Esto ocurre tanto en los medios de almacenamiento, como en los componentes electrónicos del sistema. Estas modificaciones pueden ocasionarse por fallos en el hardware (corrupción de memoria), fallos en el software (un buffer overflow que escribe en un área de memoria que no se espera) u otros motivos. El resultado es que, la imagen que tenemos del software en memoria y la imagen original en disco difieren. A este efecto se lo conoce como software aging.

En este trabajo se presenta una herramienta a nivel del sistema operativo que permite mitigar los efectos del software aging causado por errores de hardware en la memoria principal del sistema y su prototipación en el sistema operativo Linux. Se presenta además, un conjunto de experimentos y análisis de resultados, que justifican el grado de cubrimiento de errores que logra la herramienta, su aceptable efecto en la performance general del sistema, la efectividad comparativa de algunas de las estrategias que utiliza y el efecto de la incorporación de las mismas en un ambiente con requerimientos de alta disponibilidad.

Palabras clave:

Envejecimiento de Software, Errores Temporales, Rejuvenecimiento, Sistemas Operativos, kernel Linux, Administración de memoria

Keywords:

Software Aging, Soft Errors, Rejuvenation, Operative Systems, Linux kernel, Memory Management

Índice general

Índice de figuras 7

Capítulo 1. Introducción 9
1.1. Motivación 10
1.2. Objetivos 12
1.3. Público objetivo 16

Capítulo 2. Estado del arte 17
2.1. Relevamiento objetivo 17
2.2. Técnicas de rejuvenecimiento 23
2.3. Soluciones al envejecimiento 24
2.4. Investigación del Manejo de memoria en Linux 25

Capítulo 3. Prototipo 45
3.1. Motivación 46
3.2. Objetivos Iniciales 47
3.3. Selección de plataforma y herramientas 47
3.4. Revisión de Objetivos 50
3.5. Especificación Funcional 50
3.6. Arquitectura 53
3.7. Diseño 56
3.8. Implementación 84
3.9. Verificación 97
3.10. Pruebas de performance 103
3.11. Proceso de Desarrollo 109

Capítulo 4. Resultados 117
4.1. Frames de sólo lectura 117
4.2. Búsqueda de soft errors 120
4.3. Retraso de detección 122
4.4. Efecto en la disponibilidad 123

Capítulo 5. Conclusiones y Trabajo a futuro 125
5.1. Conclusiones 125
5.2. Trabajo a futuro 127

Bibliografía 129

Apéndice A. Glosario 131

Apéndice B. Manual de usuario 135

B.1. Modos de Funcionamiento 135
B.2. Registro de Procesos 136
B.3. Archivos de Estadísticas 138

Apéndice C. Porcentaje de frames de solo lectura 141

Apéndice D. Agente Inyector 153

Apéndice E. CosmicRayn 155

Apéndice F. Código para pruebas de inyección 157
F.1. Inyector 157
F.2. Hola Mundo 157
F.3. Increment 158
F.4. Function 158

Apéndice G. Salidas de las pruebas de performance 161
G.1. Pruebas de performance AB 161
G.2. Pruebas de performance POV-Ray Benchmark 177
G.3. Pruebas de performance POV-Ray Office 218

Índice de figuras

1. Clasificación de errores y acciones correctivas 19
2. Modelo de Paginación de x86 27
3. Modelo de Paginación de Linux 28
4. Mapa de la memoria física en el Linux kernel 29
5. Zoned Page Frame Allocator 32
6. Espacio de memoria virtual de un proceso 36
7. Representación del Espacio de Memoria 38
8. Diagrama de Arquitectura 54
9. Complemento de struct page 58
10. Maquina de estados de un frame 59
11. Diseño de métodos de búsqueda 72
12. Diseño de procesamiento de Errores 80
13. Inyección en una constante 100
14. Inyección en una función 100
15. Inyección en una llamada 101
16. Tiempo total de los tests 105
17. Tiempo por pedido 106
18. Pedidos por segundo 107
19. Tiempo para estar en el 80 % más rapido 108
20. Tiempo de Parseo 109
21. Tiempo de etapa Photon 110
22. Tiempo de Render 111
23. Tiempo total 112
24. Tiempo de parseo 113
25. Tiempo de render 114
26. Tiempo total 115
27. Cronograma Efectivo 116
28. Porcentaje de frames read-only por proceso antes de la prueba 119
29. Porcentaje de frames read-only durante la pruba 120

Capítulo 1

Introducción

El envejecimiento de software, o "software aging" es un fenómeno que últimamente ha empezado a cobrar importancia, siendo objetivo de varios estudios e investigaciones. Consiste en la degradación con el tiempo del estado de un proceso o de su ambiente. Esta degradación se debe a la acumulación de errores durante la ejecución de un programa lo que eventualmente afecta a su performance o provoca que falle. Como se verá en la sección 2.1 estos errores pueden tener diversos orígenes dentro de las distintas capas de software y hardware que componen un sistema de cómputo y llevan eventualmente a la degradación en la performance del software, o incluso a una falla total. En la sección 2.1.1 se ubica dichos errores en una taxonomía de acuerdo al tipo de fallas que causan, identificándose como causa de mayor interés para el proyecto la ocurrencia de errores de hardware conocidos como "soft errors". Para evitar estas fallas, se han buscado soluciones al problema aquí tratado. Como se verá en las sección 2.1 las más utilizadas son variaciones de una técnica pro-activa, llamada rejuvenecimiento de software, o "software rejuvenation", la cual consiste en detener la ejecución del software, limpiar su estado interno o su ambiente de ejecución, y finalmente volver a iniciar la ejecución. Existen distintos enfoques sobre cuándo y cómo se debe aplicar el rejuvenecimiento, ver sección 2.2. Mientras algunos se basan en predicciones de acuerdo a estadísticas del sistema como puede ser el consumo de recursos, otros apuestan a la redundancia de nodos para poder ejecutar rejuvenecimiento individual y transparente de los mismos, ante la evidencia de fallas. Se puede encontrar en el mercado algunas soluciones de software que toman en cuenta el problema de software aging, e intentan atacarlo. En la sección 2.3 se presenta un relevamiento de las mismas. Como se verá incluyen servidores web, suites de herramientas, e incluso sistemas operativos. La mayoría de estas herramientas solucionan el envejecimiento causado por el agotamiento de recursos. Existen casos en que se intenta mitigar la ocurrencia de errores en áreas particulares de la memoria, pero esto se hace por razones de seguridad, para solucionar vulnerabilidades del kernel ante ataques y no errores del hardware.

El presente trabajo se enfoca en el estudio de técnicas nuevas o existentes para mitigar los efectos del software aging. Dentro de un área tan extensa, se centrará la atención en el envejecimiento provocado por la ocurrencia de errores en el hardware, en particular, en el que constituye la memoria principal de un sistema de cómputo moderno. Existen diversos factores que pueden provocar cambios en los bits almacenados en las placas que componen dicha memoria, como pueden ser la emisión de partículas alfa, los rayos cósmicos o la interferencia electromagnética. Se verá estos y otros factores en detalle en la sección 2.1.1, pero por ahora basta con mencionar que la frecuencia con que ocurren estos errores bajo ciertas condiciones como la altura sobre el nivel del mar o la proximidad con dispositivos electrónicos, no es para nada despreciable, llegando a provocar efectos observables en la performance de un

sistema de computo. Se cuantificará más claramente la variación de la frecuencia de aparición de errores en la memoria en la sección 2.1.1 y en contraparte se mostrara en la sección 4.2 en base a experimentos, estudios y datos de fabricantes, la baja tasa de los mismos cuando no se presenta ninguna de las condiciones mencionadas.

El principal objetivo planteado para el proyecto fue llevar las técnicas para combatir el envejecimiento y en particular los soft errors, al contexto de los servicios brindados por el sistema operativo, definiendo y prototipando herramientas estándar para detectar la ocurrencia de los errores mencionados y tomar acciones para contrarrestar sus efectos en los procesos del espacio de usuarios. La sección 1.2 detallará los objetivos y el alcance planteados, trazando esto en los productos o aportes del proyecto que determinan su cumplimiento. Allí se hará también una introducción a dichos productos, los cuales se verán con mayor profundidad a lo largo de los capítulos 2, 3 y 4.

Una de las motivaciones más grandes del proyecto fue adentrarse en un área, la del envejecimiento de software, la cual es relativamente joven y cuenta todavía con mucho campo para explorar. Particularmente, fueron muy pocos los antecedentes a nivel de sistemas operativos que se pudieron encontrar sobre el tratamiento del tema. Se dedicará la primer sección de este capítulo a explicar la motivación de este proyecto y de la elección del prototipo realizado. Desde un principio se pensó en Linux como el sistema operativo ideal para el desarrollo de las herramientas planteadas. Esto es debido a cuatro razones fundamentales. La primera es que el mismo es open source, lo que facilita la investigación y modificación de su código, necesidades fundamentales para el proyecto. La segunda es la gran comunidad que hay detrás del mismo, la cual es un importante soporte en un desarrollo de este tipo. La tercera es su calidad de sistema multiplataforma, lo cual amplía el espectro de aplicación del producto desarrollado, en especial en el universo de los sistemas embebidos donde se encuentran distintas arquitecturas y Linux es cada vez más utilizado. Finalmente, la cuarta razón es la experiencia previa de los autores como usuarios del sistema e incluso haciendo pequeñas modificaciones a su núcleo con fines académicos. La posibilidad de trabajar con el kernel de Linux implicó una motivación extra, pero también un reto muy importante que se centró en comprender su funcionamiento y en particular el de su subsistema de administración de la memoria, el Memory Manager (MM), a tal nivel que permitiera realizar importantes modificaciones al núcleo del sistema operativo, sin perder la estabilidad original. Se dedicará la sección 2.4 a dar un resumen introductorio al funcionamiento del MM que sirva como contexto para el prototipo desarrollado. La tarea de investigación de Linux fue muy exigente y demandó más tiempo del esperado, manteniendo continuidad hasta las últimas etapas del proyecto. Afortunadamente, al cabo de un año y medio de haber comenzado, se pudo completar el alcance que se deseaba para redondear el mismo. No obstante todavía queda mucho trabajo a futuro para continuar lo hecho y seguir adentrándose en esta área. Se dedicará el capítulo 5 de este documento a presentar las conclusiones finales del proyecto y algunas propuestas para la continuación del mismo.

1.1. Motivación

Como se dijo, tal vez la mayor motivación para la selección del proyecto fue la posibilidad de hacer una contribución a un área relativamente joven cómo la del software aging. La problemática que la misma trata resulta de gran interés ya que

cada vez se puede encontrar más ejemplos de la misma en los problemas cotidianos que tratan la Ingeniería en Computación y la Ingeniería Electrónica. El hecho de que se trate un campo con mucho terreno inexplorado brinda la posibilidad de hacer aportes originales al mismo lo cual siempre implica una mayor motivación para el equipo de trabajo.

La diversidad de orígenes del software aging, la cual abarca un espectro demasiado amplio para un proyecto de este porte, hizo necesario seleccionar una causa en particular como son los errores en la memoria principal, específicamente originados en el hardware que constituye la misma. Mientras los errores de software que provocan envejecimiento, son causados básicamente por bugs en el mismo, los errores en el hardware no necesariamente tienen tal origen. En este contexto, el área de testing no provee técnicas que permitan encontrar la causa de los errores y eliminarlos, basta considerar el caso de los errores en la memoria causados por los rayos cósmicos (ver sección 2.1). Aún teniendo conciencia de la ocurrencia de los mismos, no ha sido posible modificar la tecnología utilizada para hacerla invulnerable a los fenómenos mencionados. Las acciones tomadas al respecto han sido a nivel del hardware, asumiendo la ocurrencia de errores y utilizando técnicas para corrección de los mismos, como las implementadas en las memorias ECC (ver sección 2.1.1). La falta de soluciones de diseño al problema y los costos de las memorias ECC, los cuales provocan que no sean las más ampliamente usadas, expresan una motivación extra para crear soluciones por software que intenten emular las técnicas de corrección implementadas en el hardware. Como se mencionó, las soluciones por software para combatir el envejecimiento que ofrece el mercado no brindan mecanismos para mitigar esta clase particular de errores y por tanto esto abre la posibilidad de hacer un aporte original al área.

Tomando en cuenta el rol del sistema operativo en un sistema de cómputo como capa de software que abstrae a las aplicaciones de usuario de las complejidades del hardware y considerando que los sistemas operativos modernos se conciben con un esquema de protección de la memoria de los procesos de usuario, lo cual implica la aplicación de una política de permisos de acceso. Es posible imaginar soluciones por software que detecten un porcentaje no despreciable de los errores en la memoria originados en hardware y tomen acciones para prevenir el envejecimiento provocado por los mismos. Incorporando dichas soluciones como servicios del sistema operativo y brindando una interfaz con los mismos como herramientas al espacio de usuario, se logra una alternativa a la utilización de hardware específico, la cual implica un ahorro de costos en el mismo y no agrega complejidad en el espacio de usuarios. Si estas ideas son combinadas con la existencia de un sistema operativo open source como Linux, se tiene el contexto perfecto para ponerlas en práctica.

La otra gran motivación para la herramienta y prototipo que se seleccionó implementar, está sin dudas en las posibles áreas de aplicación del mismo. Como se verá en la sección 2.1, basta con ser usuario de un PC en una ciudad medianamente elevada sobre el nivel del mar, para que los beneficios de estas herramientas comiencen a ser tangibles, ya que la altura es una de las condiciones que más favorece la ocurrencia de soft errors a causa de rayos cósmicos. Existen otros contextos a considerar como los satélites u observatorios ubicados a gran altura donde actualmente se invierte en hardware para mitigar los efectos de estos errores. A diferencia de esos casos, las PC no suelen contar con dicho hardware por razones de costo.

Otra área que podría verse muy beneficiada por una solución desarrollada en software es el universo de los sistemas embebidos, ya que los mismos son cada vez más comunes para el control de actividades industriales, tecnología aplicada a la medicina, aplicaciones para el hogar y muchos otros aspectos de la vida cotidiana. Como se verá en la sección 2.1, la interferencia electromagnética es una causa frecuente de la ocurrencia de soft errors en este tipo de sistemas y las probabilidades de su presencia aumentan considerablemente en ambientes como los industriales. A medida que avanzó la tecnología estos sistemas fueron adquiriendo cada vez mayor poder de cálculo y complejidad, por lo que los mismos empezaron a ser equipados con sistemas operativos para el control de los distintos módulos que los componen. Esta tendencia se volcó en los últimos años a que el sistema operativo seleccionado sea basado en el Linux kernel debido, entre otras cosas, a la evolución en soporte para tiempo real que este ha mostrado, por lo que la herramienta propuesta también otorgaría soluciones de gran utilidad y rápida aplicación en la práctica para esta área. No solo implicaría reducción en costos por fallos en operaciones industriales o electrodomésticos dañados, sino que podría llegar disminuir riesgos de vida, si se toma en cuenta la utilización de sistemas embebidos en la medicina.

1.2. Objetivos

El primero objetivo planteado para atacar el problema fue realizar un relevamiento de información, publicaciones y resultados de investigaciones que estudien el área de software aging a efectos de lograr una visión de estado del arte. En base a este relevamiento, se plateo además, proponer una taxonomía de errores nueva o existente que permitiera identificar y clasificar aquellos que originan el software aging, así como las causas y soluciones conocidas para los mismos. Como objetivo en paralelo para las etapas tempranas, se propuso además, investigar el funcionamiento del `MM`, componente encargado de la administración de la memoria en el núcleo de Linux, como paso previo a la prototipación de herramientas en dicho sistema operativo.

El cumplimiento de estos objetivos se ve reflejado en el capítulo 2 de este documento, donde se presenta en primer lugar un estudio sobre el estado del arte relativo al concepto de envejecimiento de software, el cual resulta de un trabajo investigativo que pretende introducir al lector en su definición, orígenes, métodos para evitarlo y algunos antecedentes de su tratamiento a nivel de software y en particular de sistemas operativos, entre otros. También se presenta allí la taxonomía de errores seleccionada en la sección 2.1.1. En segundo lugar, el capítulo 2 resume con espíritu introductorio, la extensa investigación realizada al funcionamiento del manejador de memoria de Linux, la cual tuvo continuidad durante todo el proyecto.

El segundo gran objetivo se centro en estudiar formas de mitigar, siempre que sea posible desde el lado del software y en particular a nivel del sistema operativo, los efectos de los errores causantes del software aging, al menos a nivel del hardware que constituye la memoria principal. En base a este estudio, se planteo proponer un conjunto de herramientas estándar a nivel de S.O. para detectar, notificar y en lo posible solucionar este tipo de errores, evitando así que los mismos afecten la ejecución de los procesos de usuario. A efectos de poder estudiar los resultados que podría tener la puesta en producción de la o las herramientas definidas, se propuso además prototipar las mismas en el sistema operativo Linux, adaptando estas a las características particulares de la administración de memoria y manejo de procesos

de su núcleo. Lo que se esperaba también con este prototipo, era determinar la viabilidad de implementación de las herramientas en un sistema operativo estándar como Linux y estudiar el efecto que esto tiene en la performance computacional del mismo.

El cumplimiento de estos objetivos se documenta en el capítulo 3 de este trabajo. Allí se describe el producto más importante de este proyecto, el cual consta de una herramienta a nivel del sistema operativo para combatir el software aging causado por errores originados en el hardware de la memoria y su prototipación en el sistema operativo Linux. La herramienta brinda funcionalidades para la detección de la ocurrencia de soft errors a nivel del sistema operativo, valiéndose del esquema de protección de la memoria utilizado por este. En un esquema de protección de acceso a la memoria en un sistema operativo moderno típico, existen porciones de la memoria física que el S.O., valiéndose normalmente de la unidad de memoria del hardware, garantiza que solo pueden ser accedidas para la lectura por los procesos del espacio de usuarios. Bajo esta hipótesis se puede asumir que cualquier modificación en un bit dentro de esas porciones de memoria, se debe a la ocurrencia de un error en el hardware. Valiéndose de este principio, la herramienta propuesta es capaz de detectar la ocurrencia de soft errors en las porciones de memoria física accedidas exclusivamente para lectura en un sistema de cómputo. Las funcionalidades planteadas incluyen la identificación durante la ejecución del sistema operativo del subconjunto actual de marcos de la memoria física de solo lectura y la utilización de códigos de detección de errores para determinar cambios en los datos contenidos en dichos marcos. Se definió además un conjunto de mecanismos para la búsqueda de errores mediante el método de detección. Se cuenta con un proceso que ejecuta permanentemente como hilo del sistema operativo, el cual utiliza dos posibles estrategias para recorrer los marcos del conjunto de solo lectura. La más simple recorre todo el conjunto de forma ordenada sin prioridad alguna. Se cuenta con una segunda estrategia la cual se vale del concepto de proceso registrado que se verá más adelante, para hacer seguimiento exclusivo a los marcos de memoria asignados a un conjunto de procesos dado. Existe un tercer mecanismo para la búsqueda de errores, el cual ejecuta desde planificador de tareas del sistema. La estrategia de este mecanismo es buscar errores en los marcos asignados al próximo proceso a obtener la CPU, justo antes de entregarle la misma.

Además de la detección de errores en la memoria física de solo lectura, se brindan funcionalidades para mitigar los efectos de los mismos sobre los procesos del espacio de usuarios. Cada marco de memoria física dentro del conjunto puede estar asignado a uno o más de estos procesos. Igual que antes, se cuenta con varios mecanismos para combatir estos errores. El primero es la utilización de códigos de corrección para identificar el o los bits que cambiaron y volverlos a su valor original. Se cuenta además con otros pasos a seguir cuando el error supera las capacidades correctivas del código. En el caso de que los datos contenidos en el marco de memoria correspondan a un archivo en disco, se cuenta con una funcionalidad que intenta refrescar los mismos desde la página original en el archivo de forma automática, ante la detección de un error. Estos métodos de corrección son llevados a cabo desde el sistema operativo, pero además la herramienta cuenta con otros mecanismos para asistir la aplicación de técnicas como el rejuvenecimiento desde el espacio de usuarios.

Cuando se mencionó los procesos registrados se refería a que el administrador de la herramienta tiene la posibilidad de seleccionar un subconjunto de las tareas (incluyendo sus subtareas o procesos livianos) que se ejecutan en el espacio de usuarios, para que solo se monitoree dicho subconjunto. Durante el registro, se les debe asignar a cada grupo de hilos un proceso agente que recibirá notificaciones de los errores detectados en sus espacios de memoria. Además de la notificación de los errores detectados, se publica al espacio de usuarios información detallada sobre los errores detectados por cada hilo. La combinación de las notificaciones y la publicación de detalles permiten la aplicación de diversas técnicas de rejuvenecimiento como las vistas en la sección 2.1 desde el espacio de usuarios. Además de las funcionalidades mencionadas, la herramienta brinda una interfaz con el espacio de usuarios que permite acceder a la información publicada por la misma, registrar procesos y configurar su comportamiento seleccionando los mecanismos de búsqueda y corrección y notificación a utilizar.

Aquí se hizo simplemente una introducción a las funcionalidades de la herramienta. Las mismas se especificaran de forma clara y completa en la sección 3.5. Teniendo bien definida la solución, esta fue adaptada e implementada como un módulo del kernel de Linux. En la sección 3.6 se dará una visión general de la arquitectura del mismo y se verá como este se integra con el sistema operativo. El diseño, implementación y verificación de dicho prototipo se documentan en las secciones 7,8 y 9 del capítulo 3, respectivamente.

Para medir la performance computacional del prototipo se seleccionó un conjunto de tests basados en pruebas de benchmarking para distintas aplicaciones, los cuales permitieron estimar los efectos del overhead y la reducción de recursos producidos en un sistema que ejecuta el modulo desarrollado, en comparación con uno que ejecuta una distribución sin modificaciones (vainilla) de la misma versión del kernel. Se repitió las pruebas con distintas configuraciones del modulo para tomar en cuenta las posibles estrategias de búsqueda de errores. Se seleccionaron tanto tests para mostrar el efecto en la performance del módulo en un escenario compuesto de aplicaciones intensivas en entrada-salida (IO-bound), como con aplicaciones intensivas en el uso del procesador (CPU-bound). Los resultados obtenidos fueron en general positivos. Se detalla la selección, ejecución, resultados y conclusiones de la medición de la performance del prototipo en la sección 3.10. Con esto se termina de describir el cumplimiento de la última meta planteada hasta el momento.

Finalmente, como último gran objetivo, se planteo medir y estimar las consecuencias de llevar a producción la herramienta desarrollada, por medio de experimentos y simulaciones, utilizando mecanismos como la inyección de fallos. Esto último, agrega el objetivo de desarrollar o reutilizar un mecanismo para inyectar o simular errores en el hardware de la memoria principal del sistema.

Una vez completo el prototipo, fue posible llevar a cabo algunos experimentos con éste, los cuales se documentaron en el capítulo 4. En dicho capítulo se presenta las pruebas realizadas y se analizan los resultados obtenidos a partir de las mismas, llegando en algunos casos a datos muy interesantes que fundamentan la utilidad práctica del producto. Para empezar, se estudió el porcentaje de la memoria física cubierto por el conjunto de marcos de sólo lectura y el grado de incidencia de los mismos en el funcionamiento del espacio de usuarios. Para determinar esto último, se midió la cantidad de marcos de solo lectura utilizados por un conjunto de procesos representativo durante la simulación de carga a los mismos. Los detalles de estos

experimentos, sus resultados y el análisis de los mismos, se presentan en la sección 4.1.

Se realizo además, un análisis del grado de éxito obtenido en la puesta en producción del sistema en un equipo de hardware al nivel del mar durante varios meses. En el mismo se compararon las tazas de errores observadas, con las esperables de acuerdo a un conjunto de datos de carácter teórico y práctico recolectados de diferentes fuentes como artículos, notas técnicas, foros y otros resultados obtenidos en experimentos realizados para el proyecto. Este análisis y sus conclusiones se presentan en la sección 4.2 del documento.

En otro experimento, se estudió y comparó la eficacia de las distintas estrategias de búsqueda de errores propuestas entre las funcionalidades del prototipo. Para esto se utilizó el concepto de retraso de detección o RD, el cual se definirá en la sección 3.7.6 como el tiempo transcurrido entre la ocurrencia de un error y su detección por parte del prototipo. El experimento apunto a estudiar los valores de RD que presentan las estrategias de búsqueda ante la simulación de errores en distintos escenarios. La descripción, resultados y análisis comparativo de las pruebas realizadas, se presentan en la sección 4.3 de este trabajo.

Para culminar, se analizó un caso de estudio de simulación de fallos en sistemas de alta disponibilidad ejecutando en un kernel compilado con el módulo desarrollado. Para esto simuló una secuencia de errores aleatorios en los marcos de solo lectura asignados a un conjunto representativo de procesos con requerimientos de alta disponibilidad. Se repitió esta prueba con y sin las funcionalidades de corrección de errores habilitadas y se analizó lo observado en búsqueda de cambios en la disponibilidad de los procesos entre uno y otro escenario. El desarrollo y análisis de este experimento se documentan en la sección 4.4. Con esto se termina de mostrar el cumplimiento el objetivo de testear y analizar los efectos de poner en producción la herramienta, aunque como se verá en el capítulo 5, aun queda mucho trabajo a futuro en esta área.

La principal funcionalidad de la herramienta desarrollada es la detección de cambios en bits en marcos de memoria los cuales sólo se pueden acceder para la lectura. Para verificar que se está brindando dicha funcionalidad correctamente se debe comprobar que se detectan errores ocurridos de forma externa al sistema operativo, en condiciones normales, en el hardware. Dado que provocar errores en el mismo puede llegar a ser una tarea compleja, costosa y la misma escapa al alcance de este proyecto, surgió la motivación de buscar una forma de poder generar estos errores por software. Para poder generar estos errores en memoria a demanda se decidió utilizar Fault-Injection o inyección de fallos. Esta técnica implica la inserción deliberada de fallas en un sistema de cómputo con el fin de determinar la respuesta del mismo. Ha probado ser un método muy efectivo para la validación de sistemas tolerantes a fallas como el que se desarrolló, permitiendo el cubrimiento de caminos en el código dedicados al manejo de errores, que de otra forma no sería posible testear. La aplicación de técnicas de inyección en el prototipo se hizo, como se adelantó, por sofware y apuntó a la memoria física del sistema. El mecanismo utilizado consistió en provocar de forma deliberada un cambio en un byte de una página de memoria seleccionada, la cual se encuentra almacenada en un frame de la memoria física. Se dedica la sección 3.9.1 de este trabajo al área de inyección de fallos, su aplicación a este prototipo y el mecanismo de inyección desarrollado en el Linux kernel para este proyecto. Mediante el mismo fue posible probar las

funcionalidades de detección, corrección y notificación de errores en todos los casos que se consideró necesario para garantizar el correcto funcionamiento de las mismas. Además, fue utilizado para la simulación de errores en varios de los experimentos documentados en el capítulo 4. Con este punto se traza el último objetivo planteado para el prototipo y se muestra el cubrimiento completo del alcance definido para el mismo.

1.3. Público objetivo

El público objetivo de esta tesis son profesionales de TI con conocimientos teóricos básicos de arquitectura de computadores, sistemas operativos y testing. Un conocimiento más profundo de estas áreas, así como conocimientos sobre el sistema de operativo Linux, sin dudas permiten un avance más rápido por el documento y una mayor comprensión de los temas aquí tratados al corto plazo. No obstante, se buscó introducir y dar buenas referencias a estos temas, especialmente software aging y Linux, de manera que el lector pueda comprender lo investigado, desarrollado y analizado en las distintas secciones.

Capítulo 2

Estado del arte

En este capítulo se presenta en primer lugar un estudio sobre el estado del arte relativo al concepto de envejecimiento de software. Es el resultado de un trabajo investigativo que pretende introducir al lector en el tema del envejecimiento de software, su definición, orígenes, métodos para evitarlo, entre otros. En segundo lugar, se resumirá a modo introductorio el producto de una extensa investigación realizada al funcionamiento del manejador de memoria de Linux , como paso previo a definir el desarrollo de herramientas para combatir el software aging a nivel de la memoria en dicho sistema operativo.

2.1. Relevamiento objetivo

El envejecimiento de software, o "software aging" es un fenómeno que últimamente ha empezado a cobrar importancia, siendo objetivo de varios estudios e investigaciones. Consiste en la degradación con el tiempo del estado de un proceso o de su ambiente. Esta degradación se debe a la acumulación de errores durante la ejecución de un programa, lo que eventualmente afecta a su performance o provoca que falle. Como se verá más adelante estos errores pueden ser errores de diseño del sistema así como también errores provocados por mal manejo de los recursos, agotamiento de los recursos del sistema operativo, corrupción de los datos, la acumulación de errores de cálculo o provocados por errores que se generen en el hardware. Esto lleva eventualmente a la degradación en la performance del software, o incluso a una falla total.

Un ejemplo de esto, es el caso de los lanza misiles Patriot durante la primera guerra del Golfo[**35**]. El software tenía un error en el mecanismo de seguimiento e identificación de objetivos. El mismo calculaba el siguiente lugar a explorar para identificar si el objetivo era del tipo requerido, y debido a un error de conversión entre enteros y reales, cuyos efectos se agravaban a medida que el tiempo de uso del sistema aumentaba, la ventana a inspeccionar cambiaba de lugar, provocando que el sistema no detectara como enemigos elementos que sí lo eran. Este es un claro ejemplo de un error de software que se agrava con el paso del tiempo, eventualmente llevando a una falla total del sistema.

Dado que el envejecimiento de software lleva a fallas, se ha tratado de buscar una solución a este problema. Huang propone una técnica pro-activa, llamada rejuvenecimiento de software, o "software rejuvenation"[**1**], que consiste en detener la ejecución del software, limpiar su estado interno o su ambiente de ejecución, y finalmente volver a iniciar la ejecución. Si bien los términos envejecimiento de software y rejuvenecimiento de software implican directamente al software, esto en realidad no es así, sino que lo que se ve afectado es el ambiente de ejecución. Técnicas de rejuvenecimiento de software se emplean en sistemas críticos como ser el software de switching de empresas de telecomunicaciones, aplicaciones encargadas del billing,

sistemas de control de vehículos espaciales, entre otros. Más recientemente, algunos servidores web como Microsoft IIS 5.0 [**19**] y Apache [**22**] también implementan técnicas de rejuvenecimiento. En particular, Apache mata y recrea procesos luego de que se atiende un cierto número de pedidos.

Cabe notar que existe otra definición de envejecimiento de software que refiere a la tolerancia del diseño y la construcción de dicho software al paso del tiempo. A medida que pasa el tiempo, a un software se le realizan varios mantenimientos (correctivo, evolutivo, etc.). Estos mantenimientos en general no son realizados por las personas que construyeron el software original, y por lo tanto es probable que no sigan los mismos lineamientos. Esto hace que el diseño e implementación de nuevas funcionalidades, o la corrección de defectos no estén alineados con el diseño del programa original. Si este efecto perdura durante varios años, el software va perdiendo calidad. Su mantenibilidad y desempeño disminuyen considerablemente. A este efecto también se lo conoce como envejecimiento de software, pero se puede solucionar con una reingeniería del software, no con la aplicación de técnicas de rejuvenecimiento. Esta definición no es la del problema que se desea tratar y se referirá a la degradación con el tiempo del estado de un proceso o ambiente, siempre que en este documento se mencione el término “envejecimiento de software”.

2.1.1. Taxonomía de errores. Jim Gray [**2**] distingue dos tipos distintos de errores en el software, los Bohrbugs y los Heisenbugs. Esta clasificación se basa en la facilidad para reproducir la falla causada por el error. La primera categoría, los Bohrbugs, cuyo nombre proviene del modelo atómico de Bohr, son esencialmente errores permanentes en el diseño y son por naturaleza, casi deterministas. Son fácilmente identificables y corregibles durante la fase de pruebas del sistema dentro del ciclo de vida del software. Por el contrario, los Heisenbugs, nombrados por el principio de incertidumbre de Heisenberg, incluyen aquellas fallas internas que son intermitentes. Son aquellas fallas cuyas condiciones de activación ocurren raramente o son difícilmente reproducibles. Fronteras entre distintos componentes de software, incorrecto o incompleto manejo de excepciones, o la dependencia de tiempo entre varios eventos, son algunas de las situaciones comunes en las cuales se pueden detectar Heisenbugs. Esta clase de falla es extremadamente difícil de identificar mediante las pruebas comunes del sistema. K. Vaidyanathan y Kishor S. Trivedi[**3**] agregan a esta clasificación una tercer categoría, que incluye las fallas ocasionadas por el envejecimiento de software. Las fallas ocasionadas por el envejecimiento del software son similares a los Heisenbugs, dado que se activan bajo ciertas condiciones, por ejemplo la falta de recursos del sistema operativo, algo que no es fácil de reproducir. Sin embargo, sus modos y métodos de recuperación difieren significativamente. A continuación se presenta un diagrama que muestra las distintas alternativas a seguir luego de detectar cada tipo de error. Esta clasificación extendida de fallas de software se consideró apropiada no sólo por su simplicidad, sino también porque incluye una categoría de errores (los relacionados con el envejecimiento de software) la cual interesa estudiar.

2.1.1.1. Bohrbugs. Esta clase de fallas de software son fácilmente reproducibles y pueden ser eliminadas con relativa facilidad en la etapa de depuración de un sistema, ya que su comportamiento en la mayoría de las veces es, determinista. Si este tipo de fallas permanece en la fase operativa del software, la única alternativa para salir de la situación es utilizar el concepto conocido como diversidad de diseño, o “Design Diversity”[**4**], en el cual se utilizan varias aplicaciones que proveen la

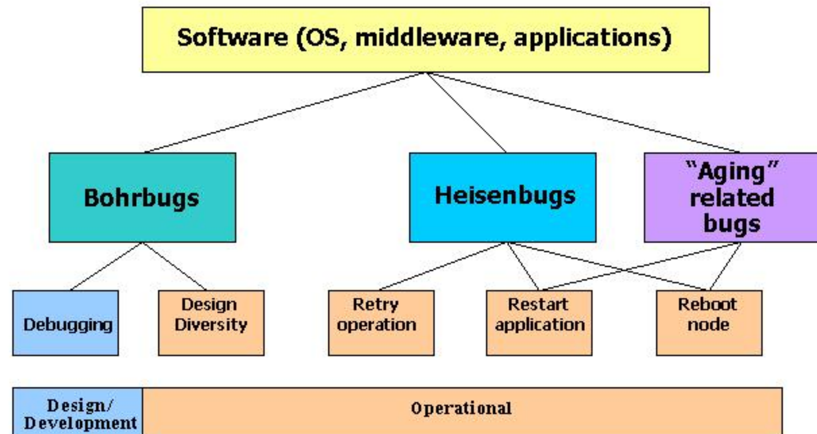


FIGURA 1. Clasificación de errores y acciones correctivas

misma funcionalidad, pero con distintos diseños o implementaciones. Esto permite enmascarar las fallas en una implementación particular.

2.1.1.2. Heisenbugs. Aún el software más maduro puede presentar fallas de este tipo, las cuales ocurren luego de una colisión de eventos determinada. No son fácilmente identificables mediante la depuración del software, y tampoco son fácilmente reproducibles. Problemas de sincronización en aplicaciones multi-hiladas son un claro ejemplo. En caso de encontrar una falla de este tipo, reintentar la operación fallida, reiniciar la aplicación o el nodo puede resolver el problema.

2.1.1.3. Aging related bugs. Este tipo de fallas aparece principalmente por el agotamiento de los recursos del sistema. Recursos tales como el espacio de swap, la memoria disponible, etc. son progresivamente consumidos por fallas en el software como el manejo inadecuado de la memoria, o la liberación incompleta de los recursos luego del uso. Estas fallas se pueden encontrar en cualquier software, desde sistemas operativos hasta aplicaciones de usuario. El tiempo estimado de agotamiento de los recursos es un tema bastante estudiado por aquellos que investigan las técnicas de rejuvenecimiento del software. Manteniendo cierta similitud con los Heisenbugs, los fallos por envejecimiento de software son en general difíciles de detectar y corregir, ya que es prácticamente imposible reproducirlos. La principal diferencia entre ellos radica en que mientras que las técnicas aplicadas a los primeros son puramente reactivas, las fallas causadas por el envejecimiento de software pueden ser prevenidas utilizando técnicas proactivas. La motivación original de este trabajo son los errores del hardware, específicamente la memoria. Esta clase de errores, conocidos como "soft errors" se podrían incluir en la categoría de fallos causados por el envejecimiento de software, si bien no se producen por el envejecimiento en sí, sino por factores externos como pueden ser los rayos cósmicos, temperatura, humedad, entre otros.

2.1.1.4. Soft Errors. Si bien los Soft Errors no son propiamente fallos del software, es importante mencionarlos aquí, ya que son una posible causa de problemas en la ejecución de un software. En particular, se desea enfocarse en este tipo de errores ya que normalmente no se tienen en cuenta durante la etapa de construcción de software, y sería interesante contar con un método de corregirlos que no esté

implementado directamente sobre el hardware. Los "Soft Errors" son errores que se originan en el hardware, posiblemente debido a fallos que se producen en los semiconductores que conforman un sistema[**5**]. Estos fallos son aleatorios, usualmente no catastróficos y no suelen causar daños permanentes en dichos semiconductores. Son causados por factores externos que no están bajo el control del desarrollador del programa. Sin embargo, es tarea de los desarrolladores construir el software de forma que pueda sobrevivir a estas fallas. Un ejemplo de estos errores en las aplicaciones de reproducción de vídeo puede manifestarse como pixeles con información incorrecta de color. Algunas de las posibles causas de aparición de este tipo de errores se explican a continuación.

Emisión de partículas alfa. Los soft errors se volvieron ampliamente conocidos con la introducción de la memoria RAM dinámica (DRAM) en los años 70. En los primeros chips construidos, se utilizaban pequeñas cantidades de material radiactivo. Al decaer el material radiactivo utilizado en el empaquetamiento del chip, se emitían partículas alfa, las cuales podían llegar a alterar la distribución de los electrones en el semiconductor, pudiendo llegar a cambiar la señal digital de un 0 a un 1, o viceversa.

Rayos cósmicos. Luego que la industria encontró la manera de controlar la emisión de partículas alfa, y observando que los errores seguían manifestándose, quedó claro que existían otras causas que podían llegar a generar dichos errores. James F. Ziegler, un investigador de IBM, publicó un gran número de papers demostrando que los rayos cósmicos podían llegar a causar errores en los semiconductores de los chips de memoria[**6**]. En estas investigaciones, IBM descubrió que los errores de memoria se incrementan con la altura, duplicándose la frecuencia de aparición a casi 800 metros por encima del nivel del mar. Por encima de esta altitud, la frecuencia era cinco veces mayor, y en la ciudad de Denver, Estados Unidos, que se encuentra a una altura de aproximadamente 1600 metros por encima del nivel del mar, era casi diez veces mayor. También se realizaron experimentos con distintos módulos de memoria operando en la segunda planta de un edificio, y luego, esos mismos módulos se trasladaron a una bóveda subterránea cercana, protegida por 20 metros de roca. En la primera parte del experimento, se probaron 864 módulos de memoria durante 4671 horas. En esas 4 millones de horas-dispositivo de pruebas, se encontraron 24 fallas de un único bit. En la segunda parte del experimento, con esos mismos 864 módulos operando durante 5863 horas, no se encontró ninguna falla. Dado que los 20 metros de roca bloquean prácticamente la totalidad de los rayos cósmicos, pero no tienen incidencia en la emisión de partículas alfa, quedó finalmente demostrado que los rayos cósmicos son la principal causante de la aparición de soft errors en dispositivos de memoria.

Interferencia Electromagnética. Otra fuente de Soft Errors es la Interferencia Electromagnética (EMI), la misma se produce cuando algún objeto natural o artificial genera un campo magnético y este interactúa con las cargas eléctricas de los circuitos integrados. Esta interacción puede provocar alteraciones en las cargas de transistores o buses, lo que a su vez puede provocar que el valor lógico de los mismos cambie. Este tipo de interferencias son especialmente comunes donde hay equipos que trabajen con motores eléctricos que tengan una alta frecuencia[**43**]. En ambientes industriales donde se utiliza motores de ese estilo, las probabilidades de ocurrencia de problemas aumentan considerablemente. Ya que las cadenas industriales modernas están altamente tecnificadas es común que se utilicen sistemas

embebidos para los sensores y demás elementos de control de la maquinaria. Al generarse un fallo en alguno de ellos es posible generar daños en los equipos, así como accidentes que generen lesiones a los operarios.

Formas de mitigar los soft errors. Hay dos formas principales de mitigar la aparición de soft errors. Una forma es diseñar los circuitos tomando en cuenta la existencia de dichos errores, y la otra es simplemente asumir que los soft errors van a ocurrir, pero intentar corregirlos. A continuación se explican estas dos formas.

- Diseño del circuito

Una forma de mitigar los efectos de los errores en la memoria, es diseñar los circuitos teniendo en cuenta que existen, por ejemplo eligiendo los semiconductores adecuados, los materiales utilizados en el empaquetamiento del chip, etc. Esto usualmente entra en conflicto con la necesidad de reducir los tamaños y voltajes de los chips, para lograr incrementos en la velocidad de funcionamiento. Ziegler encontró en 1998[**7**] que distintas tecnologías de celda para almacenar bits utilizadas en la construcción de chips de memoria DRAM de 16Mb, poseían distintas frecuencias de aparición de soft errors. En particular, celdas con carga interna (TIC cells), mostraban una frecuencia anual de 0.002 fallas por cada 32MB. Celdas con carga externa (TEC cells), poseían una frecuencia anual de error de 3 fallas por cada 32MB, y las celdas de capacitores apilados (stacked capacitor cells), mostraban una frecuencia anual de entre 0.2 y 1.1 fallas por cada 32MB. Esto demuestra claramente que el diseño del circuito utilizado en la construcción de los chips, incide directamente en la frecuencia de aparición de fallas de la memoria.

- Corrección de errores

Otra opción es asumir que los errores van a ocurrir, en lugar de intentar prevenirlos. Asumiendo que los errores ocurren, una solución es diseñar circuitos con técnicas de detección y corrección de errores, que permitan al sistema recuperarse de la aparición de un error. Los circuitos pueden ser diseñados utilizando dos enfoques distintos. "Forward error correction" refiere al uso de datos redundantes en cada palabra de la memoria, para permitir la implementación de un código de corrección de errores. "Roll-back error correction" implica la utilización de una técnica de detección de errores, tal como paridad, para luego de detectado el error, reescribir los datos corruptos desde otra fuente. Una técnica para corregir errores en cálculos de procesadores o en las memorias SRAM de los caches es utilizar sistemas con votación, en los cuales los cálculos se realizan en varios sistemas a la vez y luego se selecciona el resultado que fue obtenido en la mayor cantidad de sistemas. Debido a que la memoria (DRAM) es una parte importante de los sistemas informáticos, tanto de escritorio como servidores, y además tiene altas probabilidades de ser afectada por los soft errors, nos concentraremos particularmente en la aparición de dichos errores en este tipo de memoria. Un sistema puede utilizar una DRAM conocida como ECC para protegerse de los soft errors. Este tipo de memoria tiene construido en hardware un código de corrección de errores, tradicionalmente el llamado Código de Hamming. Este código permite la detección de errores de hasta 2 bits simultáneos y la corrección de errores de hasta 1 bit. Aunque un único rayo cósmico puede alterar simultáneamente varios bits en una DRAM, estos sistemas con corrección de errores están diseñados para que los bits vecinos pertenezcan a palabras distintas. En este caso, una perturbación puede provocar a lo sumo un solo error en una palabra dada, el cual puede ser corregido por el código de Hamming implementado. Mientras la cantidad de bits alterados no supere uno por

palabra, este tipo de memoria crea la ilusión de una memoria completamente libre de errores. Debido a la complejidad adicional que implica implementar los códigos de corrección de errores directamente sobre el hardware de la DRAM, sus costos son superiores.

2.1.2. Rejuvenecimiento de Software. Rejuvenecimiento de Software se denomina a una serie de métodos que se utilizan para solucionar el problema del software aging. A continuación se explican dos estrategias distintas de rejuvenecimiento.

Open-loop approach. Este método[**3**] se basa en aplicar el rejuvenecimiento sin utilizar ninguna clase de información sobre el sistema. En este escenario, el momento de ejecutar el rejuvenecimiento puede estar determinado simplemente por el tiempo transcurrido desde la última aplicación del algoritmo, o por el número de trabajos concurrentes que el sistema está ejecutando en el momento.

Closed-loop approach. El método de closed-loop[**3**] utiliza información sobre la "salud" del sistema para determinar si es necesario aplicar el rejuvenecimiento. En este escenario, el sistema es continuamente monitoreado, y se recolectan datos sobre los recursos disponibles y la carga de trabajo. Estos datos recolectados se utilizan luego para estimar el tiempo que le tomaría al sistema agotar un recurso que pueda causar la degradación del rendimiento o la falla de un componente o del sistema en su totalidad. Este estimado puede estar basado puramente en el tiempo, o puede estar basado en el tiempo y en la carga de trabajo del sistema. Otra opción podría ser basar la estimación en datos de fallas del sistema. Dentro del enfoque closed-loop, se puede hacer otra clasificación, que separa la forma de analizar los datos recolectados. En una primera sub categoría, el análisis se hace off-line, y en otra sub categoría, dicho análisis se hace on-line. El análisis off-line se basa en datos recolectados durante un período determinado de tiempo, usualmente semanas o meses. Lo que arroja el análisis es el período de aplicación del rejuvenecimiento. Este tipo de análisis generalmente es más adecuado para sistemas cuyo comportamiento es básicamente determinista. El análisis on-line se basa en datos recolectados cada cierto período de tiempo, mientras el sistema está en funcionamiento. Cuando se termina de recolectar un nuevo juego de datos, se utilizan en conjunto con las mediciones anteriores para determinar el tiempo adecuado para aplicar el rejuvenecimiento. Esta estrategia es muy general y sirve para sistemas cuyo comportamiento es difícil de predecir. En este caso, el comportamiento futuro del sistema está determinado por los valores actuales de las mediciones, y el histórico de valores medidos anteriormente.

2.1.3. Granularidad del rejuvenecimiento. La estrategia de rejuvenecimiento de software es muy general, y básicamente puede aplicarse en diferentes niveles - a nivel de sistema, o a nivel de aplicación [**3**]. Un ejemplo de rejuvenecimiento a nivel de sistema es un "hardware reboot". A nivel de aplicación, el rejuvenecimiento se implementa deteniendo y reiniciando la aplicación o los procesos impactados. A esto se le llama también rejuvenecimiento parcial. La aplicación de las estrategias de rejuvenecimiento puede traer consecuencias negativas en la disponibilidad del sistema o de la aplicación. Por este motivo, el rejuvenecimiento se comenzó a aplicar recientemente en arquitecturas de cluster, en las cuales más de un nodo trabajan conjuntamente como un sólo sistema, y donde la no disponibilidad de un nodo durante un corto período de tiempo tiene muy poco o ningún

impacto sobre la disponibilidad o la performance del sistema. Como se menciona en la sección anterior, la técnica para contrarrestar las fallas causadas por el envejecimiento de software consiste en reiniciar una aplicación o sistema, luego de limpiar su estado interno. Esto previene de forma proactiva, las caídas imprevistas y potencialmente costosas del sistema. Dado que el proceso de rejuvenecimiento puede ejecutarse a una hora óptima, por ejemplo cuando la carga del sistema es baja, se reduce drásticamente el costo frente a la recuperación reactiva del sistema luego de la aparición de una falla. Rejuvenecimiento de software, por lo tanto, es una forma muy efectiva en costo-beneficio de prevenir no solamente errores en el software sino también la degradación en la performance de una aplicación o sistema.

2.2. Técnicas de rejuvenecimiento

El software es extremadamente complejo y es casi imposible asegurarse que esté libre de errores. Esta situación se empeora por el hecho que el desarrollo de software es una actividad que depende fuertemente del mercado, y en la cual se busca que el producto salga lo más rápidamente posible a producción. Esto lleva a que las fallas introducidas durante el desarrollo, deban solucionarse en la fase operativa. Estas fallas pueden manifestarse de distintas formas, en particular resultan de interés las que causan el agotamiento a largo plazo de los recursos del sistema. Las técnicas que se encargan de mitigar el impacto de estas fallas se conocen como rejuvenecimiento de software. Un conocido pero extremo caso de rejuvenecimiento es reiniciar completamente el sistema. Como vimos anteriormente, aplicar estas técnicas tiene un costo de procesamiento para el sistema. Por esta razón, es importante definir correctamente la frecuencia de ejecución del rejuvenecimiento. Para determinar la frecuencia óptima de ejecución de las estrategias de rejuvenecimiento, se han seguido dos enfoques distintos[**8**].

2.2.1. Modelado analítico. El objetivo del modelado analítico es determinar el tiempo óptimo para ejecutar el rejuvenecimiento, teniendo en cuenta las consecuencias de intentar maximizar la disponibilidad del sistema, y al mismo tiempo minimizar la probabilidad de pérdida de información o el tiempo de respuesta de la aplicación. Esto es particularmente importante en aplicaciones críticas para el negocio, donde el tiempo de respuesta puede ser tanto o más importante que la disponibilidad. El cálculo analítico de la frecuencia de ejecución óptima es un tema complejo y existen actualmente una gran cantidad de modelos que intentan obtener este valor[**9**].

2.2.2. Enfoque basado en mediciones. Este enfoque se basa principalmente en la idea de detectar la presencia del envejecimiento de software utilizando mediciones del rendimiento del sistema analizado. Periódicamente se toman mediciones sobre los atributos que definen la salud del software ejecutado. Basándonos en estos datos, es posible en principio detectar la presencia de envejecimiento y poder visualizar el efecto que dicho envejecimiento tiene sobre los distintos recursos del sistema, para luego poder hacer un estimativo del tiempo restante antes de que ocurra una falla, posiblemente catastrófica. Utilizando estos mismos datos, es posible ordenar los distintos recursos del sistema según la susceptibilidad al envejecimiento, pudiendo monitorear con más detalle aquellos recursos que son más susceptibles. Si es posible obtener el tiempo antes de que ocurra una falla, es posible ejecutar el rejuvenecimiento de software con una frecuencia óptima, o relativamente

cercana a la óptima. Al utilizar enfoques basados en mediciones, en general se simplifica el modelo al suponer que el uso acumulado de un recurso del sistema en un período de tiempo sólo depende del tiempo transcurrido, pero es lógico pensar que también depende de la carga actual del sistema. Existen modelos más complejos que toman en cuenta esta variable en el cálculo de la frecuencia óptima de ejecución del rejuvenecimiento[**9**].

2.2.3. Rejuvenecimiento en cluster. Un entorno de cluster es ideal para la aplicación de estos métodos. Un cluster es una colección de sistemas independientes, que trabajan en conjunto para proveer un sistema más confiable y potente que el alcanzable por un único nodo[**10**]. Está probado que utilizar clustering es un método efectivo de escalar los sistemas para obtener mejor desempeño, soporte para una mayor cantidad de usuarios, entre otros atributos[**11**]. También permite obtener mayor disponibilidad del sistema, lo que implica ventajas al usar rejuvenecimiento en un sistema de estas características. Generalmente, los clusters cuentan con un mecanismo de recuperación frente a fallas, que transfiere la carga del nodo que falló, al resto de los nodos disponibles. El objetivo de este mecanismo es recuperarse de la forma más transparente posible para el usuario. En la práctica, esto depende en gran medida de la aplicación. Los sistemas de cluster y la aplicación del rejuvenecimiento de software forman una combinación natural, ya que es posible rejuvenecer un nodo cuando este lo requiere, sin afectar la disponibilidad del sistema en general[**36**]. Simplemente basta con invocar el mecanismo de recuperación frente a fallas del cluster para que la carga del nodo en cuestión se redistribuya, y luego reiniciar dicho nodo.

2.2.4. Soft errors y rejuvenecimiento. Si bien el rejuvenecimiento de software aplica principalmente al agotamiento de los recursos del sistema, se puede extender también a las fallas causadas por los soft errors. En este caso, el rejuvenecimiento no se puede aplicar como una técnica proactiva, ya que no hay forma de predecir la ocurrencia de un soft error. De cualquier modo, es interesante contar con técnicas que permitan detectar esta clase de errores, y de ser posible permitir al sistema recuperarse con gracia ante su aparición. Aún más interesante sería contar con técnicas de estas características implementadas en software, sin requerimientos especiales de hardware, como sería el uso de memorias ECC. En particular, podría ser de gran utilidad el desarrollo de una técnica de rejuvenecimiento sobre la memoria del sistema, basada en los servicios brindados por el sistema operativo.

2.3. Soluciones al envejecimiento

Como se mencionó anteriormente, algunos sistemas de software traen mecanismos de rejuvenecimiento incorporados. Un buen ejemplo de esto, es el reciclaje de procesos de IIS[**19**]. Sin embargo, también existen soluciones para sistemas preexistentes o que no es posible modificar, que incorporan las ventajas del rejuvenecimiento. Existen varias soluciones incluidas en este segundo caso, en especial para sistemas en los que se debe asegurar una alta disponibilidad. Algunas de estas soluciones se basan en la partición del sistema en varias máquinas virtuales y cuando se detectan signos de envejecimiento, fácilmente se pueden o bien reiniciar o en algún caso extremo destruir y volver a generar a partir de algún snapshot previo, que no sufra de envejecimiento. Otra solución posible, es la carga de un sistema auxiliar en el mismo ambiente donde está el sistema que se desea controlar, el cual se encarga

de realizar mediciones para determinar el envejecimiento [**18**] y que puede tomar acciones de rejuvenecimiento según lo que detecta. Este tipo de sistemas se conocen como Software Rejuvenation Agents (SRA). Un ejemplo de los mismos, es el IBM Director [**20**].

IBM Director permite monitorear un sistema para detectar signos de envejecimiento y aplicar rejuvenecimiento en el momento adecuado. Las primeras versiones de Director simplemente aplicaban rejuvenecimiento cada cierto tiempo fijo configurable, pero actualmente cuentan con algoritmos que realizan un análisis predictivo, tomando en cuenta no sólo el tiempo transcurrido sino también el desempeño actual del sistema para determinar el momento óptimo de aplicar rejuvenecimiento. También permite reiniciar aplicaciones individualmente, en lugar de reiniciar el sistema completo. Se puede ejecutar IBM Director en un ambiente de cluster si se requiere una alta disponibilidad del sistema. En este escenario, la herramienta reinicia el nodo afectado, y el entorno de cluster se encarga de levantar esos mismos recursos en otro nodo disponible. Hay que notar que este tipo de sistemas soluciona el problema de envejecimiento causado por el agotamiento de recursos, pero no presenta beneficios en los casos de envejecimiento causado por errores de hardware o soft errors.

La versión 10 del sistema operativo Solaris trae incluido un sistema predictivo de self-healing [**37**]. Solaris es capaz que diagnosticar un componente defectuoso utilizando un análisis predictivo, y aislarlo para evitar influencias negativas en el desempeño de otros componentes o del sistema en general. Es capaz de reiniciar automáticamente componentes de hardware o de software en caso de haberse detectado signos de envejecimiento. Si bien la disponibilidad del sistema se incrementa al contar con este tipo de mecanismos, tampoco se asegura la recuperación en caso de ocurrencia de un soft error.

La versión 10.6 del S.O. OS X de Apple [**32**] incluye como un componente nuevo de seguridad, un sistema de protección de la memoria heap del kernel basado en firmas por medio de checksums. Según Apple utilizan esas firmas para evitar la corrupción del mismo y de esta forma evitar ataques que se realizan por medio de sobreescrituras de buffers que residen en ese heap. Si bien no es mencionado directamente en la descripción del software, es de esperar que estos checksums sirvan para detectar la aparición de soft errors.

Luego de haber analizado alternativas al problema de envejecimiento de software, sigue siendo interesante considerar el caso de aparición de errores en la memoria por causas externas, conocidos como soft errors. Siguiendo este razonamiento, es interesante contar con mecanismos de protección contra este tipo de fallas, que no esté implementado directamente sobre el hardware. Ésta es la principal motivación para la creación del módulo de detección y corrección de errores de memoria, que se explica en las siguientes secciones.

2.4. Investigación del Manejo de memoria en Linux

Se resume en esta sección el funcionamiento del Memory Manager(MM), manejador de memoria de Linux. Linux es un sistema operativo multiplataforma y está diseñado de forma genérica para adaptarse a distintas arquitecturas. Si bien los prototipos desarrollados siguen este espíritu y son también independientes de la plataforma, sólo fue testeados en la arquitectura x86 y para ser concretos se prestará más atención a como se adapta Linux a ese caso particular.

Nota: Los fuentes del kernel se distribuyen en una estructura de directorios donde se agrupan de forma lógica, se tomará de aquí en más como convención, utilizar `{KERNEL}` para referirse al directorio raíz de dicha estructura.

2.4.1. Direccionamiento. Se verá aquí el esquema de direccionamiento utilizado por Linux, es decir, como se resuelven las direcciones de memoria utilizadas por los procesos en direcciones de las celdas de la memoria principal. La mayoría de las arquitecturas soportan esquemas de direccionamiento basados en segmentación y/o paginación. Por ejemplo, la familia de procesadores Intel x86 soporta un mecanismo de segmentación para separar el código, datos y el stack, el cual está siempre disponible. Además, provee un mecanismo para implementar un sistema de memoria virtual de paginación bajo demanda, que puede ser activado por el sistema operativo. En un contexto con segmentación y paginación se distinguen 3 espacios de direcciones: lógico, lineal (o virtual) y físico. El direccionamiento en el código de los procesos se hace a través de direcciones lógicas las cuales se traducen a lineales (o virtuales) por medio de la unidad de segmentación del hardware. Luego las direcciones lineales se mapean a físicas por medio de la unidad de paginación. Por más detalles referirse a[**12**].

Linux prefiere en particular el uso de un esquema basado en paginación a uno basado en segmentación porque es más portable entre arquitecturas. En las que soportan segmentación como x86, hace un uso mínimo de la misma mapeando los segmentos de código y datos, tanto de los procesos de usuario como del kernel, a todo el espacio de memoria virtual. De esta forma las direcciones lógicas se corresponden siempre con las lineales y es indistinto el uso o no de segmentación.

2.4.1.1. Paginación. Como dijimos se utiliza como mecanismo principal de resolución de direcciones la paginación, para explicar cómo Linux hace esto primero se hará un resumen de cómo funciona la unidad de paginación del hardware, por más detalles sobre esto, ver capítulo 2 de [**12**]. La unidad de paginación se encarga de traducir las direcciones virtuales en direcciones físicas. Otra tarea clave que lleva a cabo es chequear el tipo de acceso a la memoria contra los permisos de acceso estipulados para dicha dirección. Si el acceso a memoria es inválido se genera una excepción de fallo de página, para la cual el sistema operativo designa un manejador. Por eficiencia las direcciones virtuales se agrupan en intervalos de tamaño fijo llamados páginas. Direcciones virtuales contiguas se mapean a direcciones físicas contiguas. De esta forma el núcleo del sistema operativo puede especificar la dirección física y los permisos de acceso de una página completa, en vez de hacerlo por dirección virtual. La unidad de paginación particiona la memoria RAM en frames (también llamados marcos o páginas físicas) de tamaño fijo que coincide con el tamaño de página. Cada frame puede entonces contener una página, pero una página virtual es un bloque de datos que puede estar almacenado en un frame o en disco. Las estructuras de datos que mapean las direcciones virtuales con las físicas se llaman tablas de páginas. Estas se almacenan en memoria y deben ser inicializadas por el sistema operativo antes de habilitar la unidad de paginación. Las tablas de páginas funcionan como diccionarios donde el hardware busca la dirección física correspondiente a una virtual.

En una arquitectura de 32 bits como las de la familia x86 se manejan páginas de 4KB y los 32 bits de una dirección se dividen en tres campos. La traducción de direcciones se logra en dos pasos, cada uno de los cuales consta de la búsqueda dentro de una tabla. Los 10 bits más significativos de la dirección virtual se llaman

directorio y se utilizan en el primer paso de la búsqueda como índice dentro de un primer nivel de tablas llamado Directorio de Páginas. Los 10 del medio se llaman tabla y son utilizados en el segundo paso como índice dentro de un segundo nivel de tablas simplemente llamado Tabla de páginas. Los 12 bits restantes se utilizan como offset dentro de la página física y coinciden en ambos tipos de dirección. La figura 2 ilustra el mecanismo.

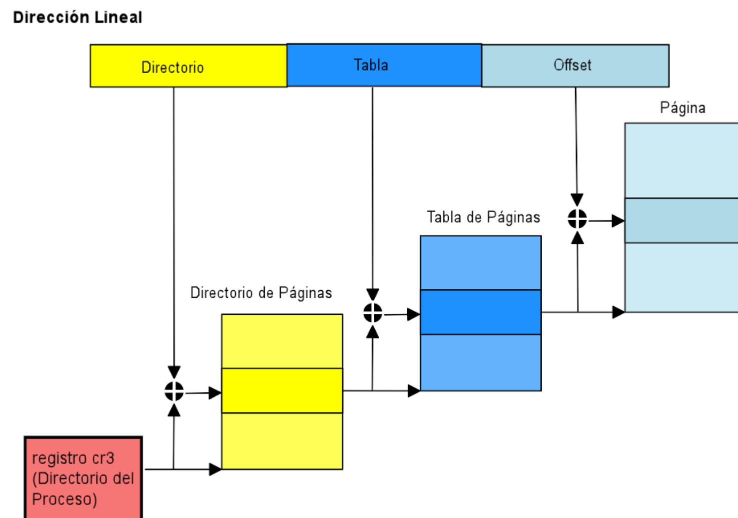


FIGURA 2. Modelo de Paginación de x86

La meta de utilizar un esquema de dos niveles es reducir la cantidad de memoria requerida para almacenar las tablas de páginas para cada proceso. Cada proceso cuenta con su propio Directorio de Páginas, pero no es necesario alojar de forma fija en memoria tablas de páginas para todo el espacio virtual, las mismas se alojan a demanda.

En una arquitectura de 64 bits, sin embargo, no es suficiente con paginación en dos niveles como la vista, pues implicaría tablas de páginas demasiado grandes. Para solucionar esto, las arquitecturas de 64 bits utilizan más niveles de paginación, normalmente entre 3 y 4. Por ejemplo, para la arquitectura ia64 se utilizan 3 niveles.

Para lograr independencia de la plataforma, Linux maneja un modelo común de paginación que se ajusta a arquitecturas de 32 y 64 bits. Este modelo fue adoptado a partir del kernel 2.6.11 y trabaja con paginación de 4 niveles como muestra la figura 3.

En el caso de 32 bits se trabaja realmente con solo 2 de estos niveles, eliminando las tablas intermedias. Para 64 bits se utilizan 3 o 4 dependiendo del hardware. Sin embargo para que el código sea independiente de la arquitectura se hace un

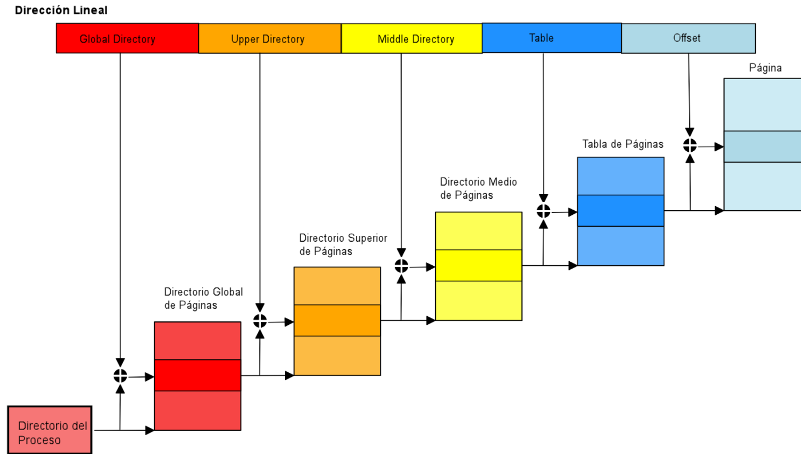


FIGURA 3. Modelo de Paginación de Linux

uso "tonto" de las tablas intermedias innecesarias, haciendo que tengan una única entrada. Linux utiliza por defecto páginas de 4k, aunque para algunas arquitecturas de 64 bits que lo soportan puede trabajar con páginas de 8k.

2.4.1.2. Mapa de la memoria física. Durante su inicialización, el kernel construye un mapa de la memoria física, el cual especifica los rangos de direcciones que son utilizables para el mismo y aquellos que no están disponibles debido a que mapean memoria de E/S de hardware o contienen datos de la BIOS. Se denomina frames reservados a los no disponibles mencionados más los que contienen el código y estructuras de datos inicializadas del kernel. Una página contenida en un frame reservado nunca puede ser reservada dinámicamente o intercambiada a disco por el mecanismo de swap, el cual se verá más adelante. El kernel se instala en RAM a partir del segundo megabyte (debido a que algunas arquitecturas hacen usos peculiares del primer megabyte), ocupando con una configuración típica 3MB.

Las arquitecturas de 32 bits pueden referenciar 4 GB de memoria física, esto significa que cada proceso puede acceder a un espacio virtual de 4 GB de memoria mediante sus tablas de páginas, obteniendo la idea de ser el único que utiliza el sistema. El kernel divide ese espacio virtual de 4 GB en dos partes de 3 GB y 1 GB, respectivamente. Los 3GB más bajos son accesibles como el espacio virtual de usuario del proceso y pueden ser accedidos tanto en modo usuario como en modo kernel, mientras que el GB más alto es el espacio virtual del kernel y sólo puede ser accedido por el proceso cuando ejecuta en modo kernel. Más adelante se hablará sobre el espacio de memoria virtual de los procesos de usuario, pero primero es importante entender el del kernel.

El kernel mantiene un conjunto de tablas de páginas para su propio uso en su Page Global Directory maestro. La parte alta de las mismas es la utilizada como referencia para las tablas de páginas modo kernel de los procesos. En ese último GB del espacio virtual se mapea directamente el primer GB de la memoria física. En caso de que el sistema cuente con más de 1GB de memoria física, esta no estará

mapeada de forma fija al espacio virtual del kernel, se denomina este rango de la memoria física high memory o memoria alta. En la próxima sección se verá como se administran y acceden los frames de memoria en este y otros casos particulares.

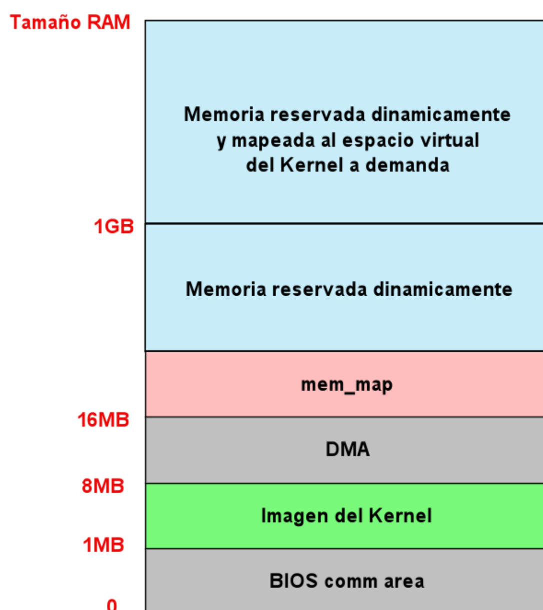


Figura 4. Mapa de la memoria física en el Linux kernel

Se definen un conjunto de variables que reflejan el mapa de la memoria física construido:

- num_physpages: número de frame del más alto utilizable.
- totalram_pages: total de frames utilizables
- min_low_pfn: número de frame del primero utilizable luego de la imagen del kernel
- max_pfn: número de frame del último utilizable
- max_low_pfn: número de frame del último mapeado directamente por el kernel (low memory)
- totalhigh_pages: total de frames no mapeados directamente por el kernel (high memory)
- highstart_pfn Page: número de frame del primero no directamente mapeado por el kernel
- highend_pfn Page: número de frame del último no directamente mapeado por el kernel

2.4.2. Administración de Frames de Memoria. Como se vio en la sección anterior, el kernel utiliza los frames como unidad básica de administración de

memoria física y se trabaja con frames de 4k y 8k dependiendo de la arquitectura, por ejemplo en intel 4k para 32 bits. El tamaño seleccionado para las páginas se maneja dentro del kernel mediante la constante `PAGE_SIZE`. De esta forma la memoria física se divide en frames de tamaño `PAGE_SIZE` los cuales se enumeran correlativamente desde la dirección más baja, asignando un índice a cada uno el cual se conoce como page frame number o `pfn`.

2.4.2.1. Estructura page. Para la administración de cada uno de esos frames y el mantenimiento de su estado actual se cuenta con la estructura *page*, definida en `{KERNEL}/include/linux/mm.h`. Es importante resaltar que se refieren exclusivamente a páginas físicas, no virtuales, ni los datos contenidos en estas. Estas estructuras se agrupan en el array global al kernel llamado `mem_map`. Dado que se encuentras alojadas en memoria de forma permanente, ocupan aproximadamente el 1 % de la memoria física para el caso en que se utilicen páginas de 4K.

Mediante los campos de la estructura `page`, el kernel puede determinar el uso que se le esta dando al frame. Por ejemplo, si contiene una página que pertenece a un proceso de usuario, si contiene código o datos del kernel. Además, debe determinar si un frame utilizado como memoria dinámica esta libre o asignado a algún proceso o cache, etc. Se mencionan a continuación algunos de los campos más importantes, pero se seguirá profundizando en el uso de la `page` en ésta y las próximas secciones.

- flags : Array de flags o banderas de estado
- _count : contador de uso del frame, si vale -1 está libre
- _mapcount : número de entradas de tabla de páginas que referencian el frame (-1 cuando no hay ninguna)

2.4.2.2. Soporte NUMA (Non Uniform Memory Access). NUMA implica que en un contexto con múltiples procesadores el tiempo de acceso de cada uno de estos a distintas regiones de memoria varía. Para cada CPU, Linux divide la memoria en una serie de nodos donde el tiempo de acceso de dicha CPU a cada dirección del nodo es el mismo. A partir de esto se trata de minimizar el acceso a nodos costosos poniendo las estructuras más usadas por cada CPU en la memoria más cercana a ella. La memoria física dentro de cada nodo se puede dividir a su vez en varias zonas como se verá a continuación. En los contextos más comunes, como por ejemplo las arquitecturas x86 de Intel, el soporte de NUMA no se necesita y se define un solo nodo con toda la memoria.

2.4.2.3. Zonas. Algunas arquitecturas tienen restricciones de hardware que limitan la forma en que los frames pueden ser utilizados. Para adaptarse a estas limitaciones, Linux divide la memoria en zonas. Algunos ejemplos de las restricciones que se pueden encontrar, aparecen en la arquitectura x86 de Intel. Por una parte, los procesadores de DMA (Direct Memory Access) de los buses ISA sólo pueden direccionar en los primeros 16MB de RAM, restricción que no tienen buses más modernos como pci o pci express ya que utilizan direcciones de 32 o hasta 64bits. Por otro lado, como ya vimos en secciones anteriores, cuando se cuenta con mucha memoria RAM en una arquitectura de 32 bits, la misma excede el espacio lineal de direcciones del kernel y la CPU no puede direccionar sobre esta. Para lidiar con limitaciones de esta clase, Linux define 3 zonas en `{KERNEL}/include/linux/mmzone.h`:

- *ZONE_DMA* : frames del rango que es posible utilizar para DMA
- *ZONE_HIGHMEM* : frames que no tienen dirección virtual fija en el espacio del kernel

- *ZONE_NORMAL* : frames normales, mapeadas de forma regular al espacio virtual del kernel

En Intel 32 bits *ZONE_DMA* esta compuesta por los frames incluidos en el rango de los primeros 16 MB de memoria física, *ZONE_NORMAL* va desde la marca de los 16 MB hasta los 896 MB y finalmente *ZONE_HIGHMEM* agrupa los frames de toda la memoria del rango superior a la marca de los 896 MB.

Como se mencionó, el espacio de memoria virtual del kernel es de 1GB (el último de los 4). El kernel reserva 128MB para sus estructuras por lo que dispone de 896MB para mapear el resto de la memoria y por esto *ZONE_NORMAL* se extiende sólo hasta esa marca. Por definición de la memoria alta, un frame incluido en la misma no tiene asociada una dirección virtual, si una estructura *page*. Parte de los últimos 128MB reservados del espacio del kernel, se usan por medio de primitivas, para mapear temporalmente al espacio virtual, frames de memoria alta. En arquitecturas de 64 bits *ZONE_HIGHMEM* está vacío, pues todos los frames tienen dirección virtual fija en el espacio del kernel.

2.4.2.4. Pool de frames reservados. Como se verá en la próxima sección, para satisfacer los pedidos de memoria dentro del kernel puede suceder que haya memoria disponible y se asigne inmediatamente o que por el contrario sea necesario recurrir a mecanismos para reclamar y liberar memoria, que se verá más adelante, los cuales pueden bloquear al hilo que hizo el pedido. En algunos contextos de ejecución del kernel como pueden ser los manejadores de interrupciones o en general la ejecución de zonas críticas sosteniendo locks, los pedidos de memoria deben ser satisfechos sin la posibilidad de bloquearse. En estos casos si no se cuenta con la memoria inmediatamente, el pedido falla. El kernel reserva una serie de frames para que sea más factible poder satisfacer estos pedidos cuando la memoria escasea. La cantidad de frames reservados varía según la cantidad de páginas con mapeo fijo (ZONE_DMA + ZONE_NORMAL), el valor se puede ver y cambiar en `/proc/sys/vm/min_free_kbytes` y se calcula como: ceil(sqrt(16 * (ZONE_DMA + ZONE_NORMAL))). Cada zona contribuye a la reserva de forma proporcional a su tamaño.

2.4.3. Reserva de memoria. Hasta ahora se habló de cómo el kernel organiza los frames de memoria. Se verá a continuación el mecanismo que se utiliza para satisfacer los pedidos de reserva frames dentro del mismo. El kernel cuenta con un subsistema dedicado a esta tarea, llamado Zoned Page Frame Allocator. El mismo maneja los pedidos de reserva de páginas contiguas, en la figura 5 se puede ver cómo está compuesto. Más adelante se verá las interfaces básicas que se brindan para reservar frames y los distintos niveles de abstracción que se construyen sobre ellas, pero primero se verá cómo funcionan cada uno de los componentes del diagrama para entender cómo se logran satisfacer los pedidos.

2.4.3.1. Buddy System. Un problema común de los algoritmos de asignación de memoria contigua es la fragmentación externa, esto es, la acumulación de pequeños segmentos de memoria libre (siempre tomando como unidad básica el frame) entre segmentos más grandes de memoria contigua reservada. Este fenómeno se produce debido a la asignación y liberación continua de grupos de páginas contiguas, de diferentes tamaños y provoca que no se puedan satisfacer pedidos aún cuando se cuenta con suficiente memoria libre. Una de las formas de combatir la fragmentación externa es la utilización de un algoritmo para administrar los segmentos de memoria contigua libre, que permita evitar, en lo posible, la partición

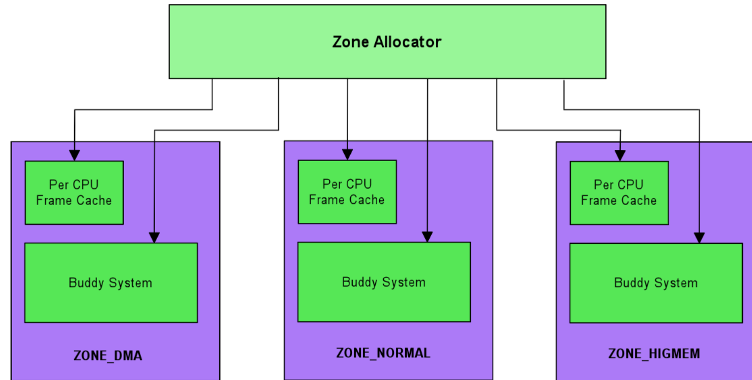


FIGURA 5. Zoned Page Frame Allocator

de un bloque grande para satisfacer un pedido de menor tamaño. Linux adopta el conocido algoritmo "buddy system" para resolver este problema. El mecanismo es bastante simple, los segmentos de frames libres se agrupan por tamaño en 11 pools. Los tamaños posibles son 1, 2,4,8,..., 1024 frames, lo que implica que con páginas de 4k el más grande tiene segmentos de 4MB. La dirección física de la primera página de un trozo es múltiplo del tamaño del mismo, de esta forma se puede ver como que cada segmento tiene otro contiguo de igual tamaño, un buddy. Como veremos, los pedidos de memoria se realizan indicando la cantidad de frames por medio de potencias de 2. Para satisfacer un pedido de memoria de 2^k frames se saca del pool correspondiente a dicho tamaño y si no hay disponibilidad, se busca en el pool inmediato más grande. Si se encuentra un segmento disponible ahí, se guarda el/los trozos sobrantes en el pool correspondiente. En caso contrario, se busca en el siguiente pool y así sucesivamente. Si se llega al pool más grande (el de 1024) y no se logra satisfacer el pedido, se retorna error. La parte más interesante se da en la liberación de un trozo de memoria y es lo que da nombre al algoritmo. Para retornar la memoria a los pools, se busca a su buddy de igual tamaño en el pool correspondiente para unirlo e intentar formar un trozo del siguiente pool. Si se logra, se repite hasta formar el mayor trozo posible, de lo contrario se retorna al pool del tamaño actual. Linux cuenta con un buddy system para cada zona de memoria.

2.4.3.2. Per-CPU Frame Cache. Dado que obtener y liberar un único frame es una operación muy común dentro del kernel, como forma de optimizar la performance, se cuenta con un cache de frames previamente reservados en cada zona, para satisfacer estos pedidos. Los frames reservados en el cache se obtienen del buddy system. Este cache está formado en realidad por 2 caches. Por una parte se tiene un hot cache con frames cuyos contenidos serán seguramente incluidos en las líneas del cache de hardware de la CPU. Se tiene además un cold cache, del cual conviene sacar páginas para operaciones DMA que no utilizan la CPU y por tanto su cache de hardware, de forma de reservar los hot para la ejecución de procesos de usuario y kernel.

2.4.3.3. Zone Allocator. El Zone Allocator es el front end del mecanismo de reserva de frames. Su tarea es encontrar una zona con suficiente memoria libre para satisfacer un pedido. Esto último no es trivial dado que tiene que cumplir los siguientes objetivos:

- Preservar el pool de frames reservados
- Accionar el algoritmo que realiza swap a disco cuando la memoria es escasa y es posible bloquearse, se verá más adelante
- Preservar la ZONE_DMA, o sea, evitar hasta donde sea posible, su asignación para otros usos

El mecanismo de obtención de páginas del Zone allocator se centraliza en la función `__alloc_pages( )`. Las interfaces para obtener memoria de más alto nivel que veremos, se basan en esta función. La misma recibe como parámetros un conjunto de flags que determinan cómo se obtiene la memoria, la cantidad de páginas a retornar expresada como el orden de una potencia de 2 y por último una lista de zonas de donde es posible obtener el segmento, ordenadas por prioridad. Lo que hace es básicamente recorrer la lista de zonas en busca de memoria y si no hay suficiente se activa el mecanismo de reclamación de páginas que se verá más adelante. Luego de esto se vuelve a recorrer la lista de zonas. Si sigue sin encontrar se puede recurrir a reserva o dar error, dependiendo del contexto desde el cual se pide y las flags.

2.4.4. Interfaces para reserva de memoria. Primero revisaremos la interfaz del mecanismo básico de asignación orientada a frames visto. Dado que dicho mecanismo brinda poca flexibilidad en el uso práctico, se crearon un conjunto de abstracciones sobre el mismo que revisaremos a continuación.

Una nota importante a tener en cuenta antes de recorrer las interfaces, es que a nivel de kernel no se utiliza protección de memoria "el kernel confía en si mismo" y por esto no se habla en ningún caso de permisos de acceso. El esquema de protección de memoria se aplica al espacio virtual de los procesos de usuario, el cual será visto con detalle en la próxima sección.

2.4.4.1. Obtener páginas. El único mecanismo de obtención de memoria de bajo nivel que ofrece el kernel funciona a nivel de páginas. Las funciones que se brindan para obtener memoria con granularidad de páginas están en `{KERNEL}/include/linux/gfp.h`, donde la operación central es:

```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int
order)
```

La misma reserva 2^{order} frames contiguos en memoria y retorna un puntero a la estructura *page* correspondiente al primero, o NULL en caso de error. Existen también algunas variaciones útiles basadas en *alloc_pages* que se listan a continuación.

- `alloc_page(gfp_mask)` : análoga a la anterior, pero reserva una sola página.
- `get_free_pages (gfp_mask, order)` y `get_free_page (gfp_mask)`: Análogas a *alloc_pages()* y *alloc_page()*, pero retornan la dirección virtual del frame en el espacio del kernel, en vez del descriptor.

De la misma forma existen diferentes variaciones para liberar la memoria reservada:

- `void __free_pages(struct page *page, unsigned int order)`

- `void free_pages(unsigned long addr, unsigned int order)`
- `void free_page(unsigned long addr)`

El parámetro *gfp_mask* que reciben *alloc_pages()* y sus variantes, se utiliza para indicar flags que determinan cómo se intenta obtener la memoria. Existen tres categorías de flags. Por una parte se tiene los llamados Modificadores de acción, los cuales determinan cómo se obtiene la memoria, si se puede dormir, bloquearse, etc. A continuación se listan los valores posibles:

- `__GFP_WAIT` : es posible dormir
- `__GFP_HIGH` : se puede acceder a los pools de emergencias
- `__GFP_IO` : es posible realizar E/S a disco
- `__GFP_FS` : es posible realizar E/S a filesystem
- `__GFP_COLD` : se debe usar páginas de cold cache
- `__GFP_NOWARN` : no se deben imprimir warnings de falla
- `__GFP_REPEAT` : se repetirá la reserva si falla, pero potencialmente la misma puede fallar
- `__GFP_NOFAIL` : se repetirá indefinidamente la reserva y la misma no puede fallar
- `__GFP_NORETRY` : no se hacen reintentos en caso de falla
- `__GFP_NO_GROW` : uso interno de la capa de Slab
- `__GFP_COMP` : uso interno del código de hugetlb

Por otra parte se cuenta con una categoría de flags llamada Modificadores de zona, los cuales especifican de qué zonas es posible obtener la memoria. Se tiene por un lado *__GFP_DMA* que indica que sólo se puede buscar en la *ZONE_DMA* y *__GFP_HIGHMEM* que indica que se puede buscar en *ZONE_HIGHMEM* o *ZONE_NORMAL*.

En la práctica no se utilizan directamente estas flags, sino que existe un tercer conjunto de flags llamado Modificadores de tipo, que definen patrones de reserva compuestos de varios modificadores de acción y zona. Cada tipo significa un uso específico de memoria, por ejemplo para usar en contexto de proceso dentro del kernel se tiene el tipo *GFP_KERNEL*, mientras que en un contexto de interrupción o zona crítica es necesario utilizar el tipo *GFP_ATOMIC* para evitar bloqueos. Se listan a continuación todos los tipos, con la composición de flags que implican:

- `GFP_ATOMIC = __GFP_HIGH`
- `GFP_NOIO = __GFP_WAIT`
- `GFP_NOFS = __GFP_WAIT | __GFP_IO`
- `GFP_KERNEL = __GFP_WAIT | __GFP_IO | __GFP_FS`
- `GFP_USER = __GFP_WAIT | __GFP_IO | __GFP_FS`
- `GFP_HIGHUSER = __GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM`

2.4.4.2. kmalloc(). En la práctica la memoria se suele reservar con granularidad de bytes o estructuras y no páginas. Por este motivo, se brinda una interfaz similar a la conocida `malloc()` de espacio de usuario, construida sobre la interfaz con granularidad de páginas mencionada. La diferencia con `malloc()` es que la versión del kernel recibe un parámetro extra para especificar las ya estudiadas flags de reserva. En caso de éxito, `kmalloc()` retorna una región de memoria físicamente contigua de al menos el tamaño solicitado en bytes o NULL en caso de error. Existe también la función `kfree()` para liberar la memoria obtenida por este método, ambas se definen en `{KERNEL}/include/linux/slab.h`.

2.4.4.3. vmalloc(). Esta interfaz es similar a la anterior. La diferencia es que la memoria retornada asegura ser virtualmente contigua, pero no tiene por qué ser físicamente contigua. Lo que hace es obtener trozos de memoria física no necesariamente contiguos y modificar las tablas de páginas del kernel para que sean virtualmente contiguas. La ventaja frente a `kmalloc()` es que se logra satisfacer pedidos en condiciones de mediana escasez y gran fragmentación, donde hay suficiente memoria, pero no contigua. La clara desventaja es la disminución en la performance provocada por modificar las tablas de páginas del espacio virtual del kernel. Esta interfaz se define en `{KERNEL}/include/linux/vmalloc.h` y se utiliza de la misma forma que la interfaz *malloc()/free()* de espacio de usuario:

2.4.4.4. Slab Allocator. Como se dijo, en la práctica no se maneja memoria a nivel de páginas sino a nivel de estructuras u objetos. Por esta razón conviene tener mecanismos de manejo de memoria que trabajen sobre el Zone allocator y ofrezcan una interfaz más apropiada. Uno de estos mecanismos se brinda mediante el slab allocator, el cual además, cumple con los objetivos de minimizar la fragmentación interna dentro de los frames que maneja y hacer eficiente la reserva de objetos comunes, que se crean y destruyen continuamente dentro del kernel.

La idea detrás de este mecanismo es agrupar los objetos en caches, los cuales funcionan como almacenes de objetos del mismo tipo. A su vez cada cache se divide en slabs que son conjuntos de uno o más frames de memoria contiguos. Cada slab se particiona en objetos, pudiendo contener libres y reservados al mismo tiempo. Cada cache se representa en el kernel por medio de la estructura `kmem_cache_t` y son creados y eliminados por medio de los métodos `kmem_cache_create()` y `kmem_cache_destroy()`, respectivamente. Por medio de la operación de creación se especifica el tamaño de cada objeto que se reservará por medio de ese cache. Para la reserva y liberación voluntaria de objetos se cuenta con la interfaz `kmem_cache_alloc()` / `kmem_cache_free()`. Dado que esta interfaz es una abstracción, internamente se reserva y libera memoria con granularidad de páginas para poder satisfacer los pedidos de objetos realizados.

2.4.4.5. Memory Pools. Son nuevos en el kernel 2.6. Es una forma que tiene un subsistema del kernel de reservar memoria dinámica para cuando la memoria escasea. Esta memoria puede ser usada sólo por dicho subsistema, que toma el rol de propietario de la misma. Este mecanismo está pensado para ser un último recurso cuando las formas habituales de reserva fallan por condiciones de baja memoria. Están definidos por la estructura `mempool_t` y se utilizan como una capa sobre otra forma de reserva de memoria. En su creación por medio de la operación `mempool_create()` se puede especificar un objeto que funcione como pool de datos (por ejemplo un slab cache) y se especifican las operaciones que debe usar internamente para liberar y reservar, junto con la cantidad mínima de elementos de memoria que debe garantizar el pool cuando la obtención de memoria de la capa inferior falla. Para obtener memoria se cuenta con la operación `mempool_alloc()` que intenta obtener de la capa inferior con la función de reserva especificada y retorna del pool cuando falla. Para liberar memoria se tiene la operación `mempool_free()` que libera memoria a la capa inferior cuando el pool está lleno. Se cuenta también con un método `mempool_destroy()` que libera toda la memoria contenida en el pool y la ocupada por las estructuras del mismo.

2.4.5. Espacio de memoria de procesos de usuario. El espacio de memoria de un proceso es el conjunto de direcciones virtuales a las que puede

acceder[**12**], capítulo 9. Los espacios de diferentes procesos son completamente independientes a excepción de los hilos. Es posible que hilos de un mismo grupo compartan el mismo espacio de memoria. Por más referencias sobre hilos y el manejo de procesos de Linux en general, referirse a [**12**], capítulo 3. Es importante aclarar que se refiere a la memoria virtual y no física. Como se dijo en la sección 2.4.1, cada proceso cuenta con su propio directorio maestro de tablas de páginas, mediante las cuales las direcciones de su espacio virtual se traducen a celdas de la memoria física y es posible que un mismo frame esté mapeado a páginas virtuales de distintos procesos. También se dijo que existe una porción de dichas tablas de páginas que es una réplica de las del kernel, de forma que el espacio virtual de este sea accesible en el contexto de cada proceso cuando ejecuta en modo kernel. Dependiendo de la arquitectura, el espacio lineal de direcciones puede ser de 32 o 64 bits. El espacio se divide en una marca denominada *TASK_SIZE* (3GB para 32 bits), por encima de la cual se sale del espacio virtual del proceso y se entra en el del kernel. Por debajo de *TASK_SIZE*, el espacio de memoria virtual de cada proceso queda determinado por los rangos de direcciones virtuales que tiene permitido utilizar, los cuales se denominan áreas o regiones de memoria y son definidos por una dirección de comienzo, un largo y un conjunto de permisos de acceso. Por razones de eficiencia, el largo del área debe ser un múltiplo de *PAGE_SIZE* de forma que los datos contenidos en la misma completen un conjunto de páginas virtuales y por tanto llenen los frames físicos a los cuales estén mapeadas. La figura 6 ilustra el espacio virtual de un proceso.

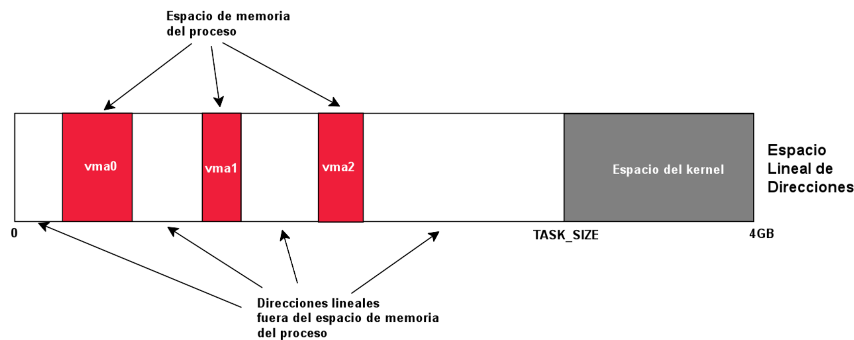


FIGURA 6. Espacio de memoria virtual de un proceso

Para entender el uso práctico de las áreas virtuales, se verán a continuación los casos más comunes en los cuales se crean las mismas:

- Cuando se crea un nuevo proceso, se crea para el mismo un espacio virtual con un conjunto de regiones, típicamente una copia de las regiones de su padre. Cada proceso cuenta al menos con regiones de memoria para mantener su stack de modo usuario, secciones de código, bibliotecas linkeadas dinámicamente, variables inicializadas, heap para reserva de memoria dinámicamente y más.

- Cada vez que un proceso realiza un mapeo a memoria de un archivo en disco para acceder al mismo, esto provoca la creación de un área virtual para el mapeo.
- Cada vez que un proceso crea un recurso de memoria compartida para comunicación con otros procesos, esto resulta en la creación de una nueva región de memoria.

Se vio en los ejemplos anteriores que las áreas virtuales del espacio de cada proceso reflejan claramente, la utilización de memoria durante la ejecución del mismo.

Se detallará en los próximos párrafos, las estructuras que utiliza el kernel para la definición y administración de los espacios y áreas virtuales.

2.4.5.1. Espacio de memoria. El kernel representa el espacio de memoria de un proceso con un descriptor que en código es la estructura mm_struct definida en `{KERNEL}/include/linux/mm_types.h`. Cada descriptor de proceso en Linux tiene referencia a una instancia de una de estas estructuras, varios procesos pueden apuntar a la misma en caso de que sean hilos de un mismo grupo. Los campos más importantes de las mm_struct son:

- *mm_users*: cantidad de procesos que comparten el espacio (cuenta uno por hilo)
- *mm_count*: contador global de referencias, cuenta 1 por todos los hilos
- *mmap*: lista con todas las áreas de memoria del espacio, ordenadas por dirección virtual de comienzo de forma ascendente
- *start_brk* y *brk*: direcciones de inicio y fin del heap respectivamente
- *rb_map*: red-black tree de áreas, estructura más apropiada que mmap para la búsqueda
- *mmlist*: campo list_head que le permite pertenecer a la lista de todos los espacios del sistema, cuya cabeza es el espacio del proceso init

La creación del espacio de memoria se hace durante la creación del proceso mediante la función `copy_mm()` a partir del espacio de memoria de su proceso padre. En el caso de los procesos tradicionales, se crea un nuevo `mm_struct` y se replican las áreas de memoria. Sin embargo, se comparten las tablas de páginas y las propias páginas virtuales con el espacio del padre, mientras estas sean accedidas como sólo lectura. La duplicación efectiva de todo el espacio se hace a demanda con los accesos de escritura de ambos procesos. Esta técnica se conoce como copy-on-write y será vista en detalle en las próximas secciones. Un caso particular, se da con los procesos lightweight [**12**], capítulo 3, pues simplemente se referencia en el nuevo proceso el mm_struct de su proceso padre, de forma que éstos comparten todo el espacio de memoria y deben sincronizarse para su uso concurrente.

Otro caso especial de procesos, son los hilos de kernel [**12**], capítulo 3. Dado que los mismos son procesos que ejecutan código del kernel, nunca acceden a direcciones por debajo de *TASK_SIZE*. Como el kernel no utiliza un esquema de protección de la memoria de sí mismo, estos hilos no utilizan áreas de memoria y por tanto la mayoría de los campos del `mm_struct` no se aplican a ellos, de modo que no poseen uno. Dado que las tablas de páginas para las direcciones por encima de *TASK_SIZE* son idénticas para todos los procesos, por motivos de eficiencia, los hilos de kernel simplemente ejecutan con el directorio de tablas ya cargado, el del proceso anterior que obtuvo el procesador.

En la figura 7 se ilustra la representación utilizada por el kernel para el espacio de memoria de un proceso.

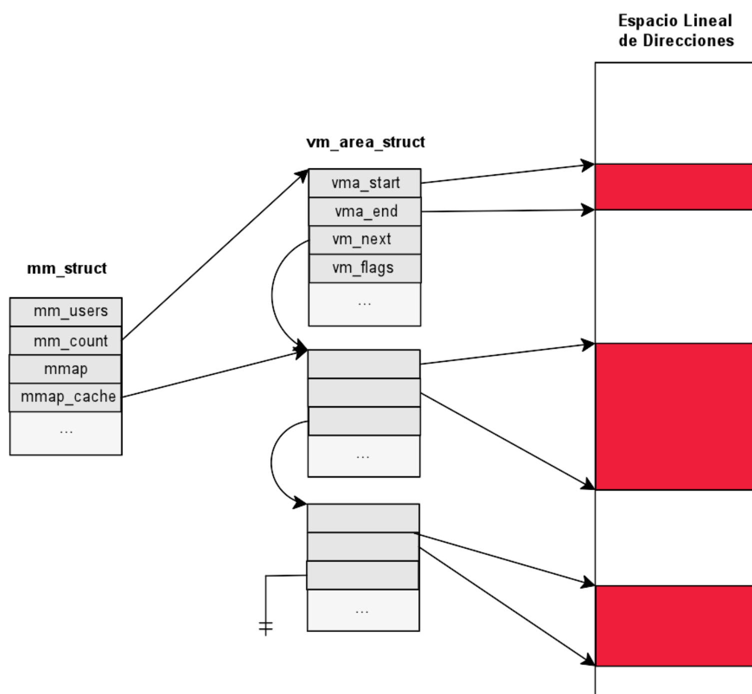


Figura 7. Representación del Espacio de Memoria

2.4.5.2. Áreas de memoria. Como dijimos son los intervalos del espacio lineal de direcciones de un proceso que este tiene permitido usar. Esto implica que sólo puede acceder a direcciones que caigan dentro de un área de memoria valida y respetando los permisos (del estilo read/write/execute) definidos para ésta, de lo contrario ocurrirá una excepción. Las áreas no se solapan, cada dirección válida del espacio existe en una y sólo una, además el kernel trata de unirlas si se tienen dos contiguas. El proceso puede agregar y remover áreas dinámicamente a través del kernel. Existen áreas individuales para cada porción de memoria del proceso entre las cuales se pueden identificar por ejemplo:

- Mapeo en memoria del ejecutable (text section)
- Mapeo de las variables globales inicializadas del ejecutable (data section)
- Mapeo de la página de ceros que contiene las variables globales sin inicializar del ejecutable (bss)
- Mapeo del stack de espacio de usuario del proceso
- Tripleta text, data, bss para cada biblioteca compartida (biblioteca de C, linker dinámico, etc.) que se carga en el espacio del proceso
- Cualquier archivo mapeado a memoria
- Segmentos de memoria compartida

- Mapeos anónimos (usados por ejemplo para malloc() implementado con mmap)

Las áreas se representan en el kernel por medio de la estructura . Los campos más importantes y que definen el área son una dirección inicial (*vm_start*), la primera dirección fuera del área (*vm_end*) la cual permite determinar su largo (*vm_end - vm_start*) y el conjunto de permisos de acceso (*vm_flags*) el cual se verá con más detalle luego. Tanto la dirección inicial como el largo deben estar alineados al tamaño de página.

2.4.5.3. Fallos de página. En la sección 2.4.1 se dijo que para garantizar el esquema de protección de la memoria la unidad de paginación genera una excepción de fallo de página cuando un acceso a memoria no respeta los permisos de acceso especificados en la entrada de tabla de páginas correspondiente. Este no es el único caso en que se genera dicha excepción, pues también sucede cuando la página de memoria virtual deseada no está mapeada a un frame de la memoria física en las tablas de páginas del proceso actual. El manejador de fallo de página de Linux puede distinguir entre los orígenes posibles de la falla, gracias los descriptores de las VMA del espacio del proceso. Si la falla se dio por un acceso no permitido, ya sea porque direccionó fuera de su espacio de memoria o violó los permisos asignados para éste, Linux envía una señal SIGSEV al mismo. Si el acceso fue legal, existen dos casos posibles. Si la página no está marcada como presente en las tablas de páginas del proceso, es decir, si no está asociada a un frame de la memoria física, se reserva un frame libre mediante los mecanismos vistos y se actualizan las tablas de páginas del proceso para mapear la página accedida al mismo. Este mecanismo se conoce como paginación bajo demanda y se verá con más detalle a continuación. Si la página está marcada en la tabla como presente y con permisos de sólo lectura, entonces se aplica otro mecanismo conocido como “copy on write” que también se verá en los próximos párrafos.

2.4.5.4. Paginación bajo demanda. Paginación bajo demanda es una técnica de reserva dinámica de la memoria que consiste en diferir la reserva de frames hasta el último momento posible, esto es, cuando el proceso accede a la misma, causando un fallo de página. De esta forma cada proceso comienza sin ningún frame de memoria reservado. Este mecanismo se basa en la hipótesis de que un proceso no accede desde el comienzo a todas las páginas de su espacio de memoria y tal vez haya algunas que nunca sean accedidas. Existe un principio de localidad que asegura que en cada etapa de la ejecución de un programa sólo accede un pequeño subconjunto de sus páginas y por tanto los frames que contengan páginas fuera de este subconjunto podrían ser utilizados por otro proceso. En ese contexto la utilización de esta técnica aumenta la cantidad promedio de frames libres y mejora el rendimiento del sistema con la misma cantidad de RAM. Si bien el procesamiento de los fallos de página significa un overhead en el sistema, el principio de localidad asegura que luego de que el proceso comienza a trabajar con un conjunto estable de páginas, los fallos se vuelven eventos poco frecuentes. Una página puede no estar presente en memoria si el proceso nunca la accedió o debido a que el frame al cual estaba mapeada fue liberado por el mecanismo de reclamación de páginas del kernel que se verá en la sección 2.4.7. En ambos casos el manejador de fallos de página debe asignar un nuevo frame a la página, pero la forma de inicializar dicho frame difiere según el caso y según el tipo de página. Se dan tres situaciones posibles:

1. La página nunca fue accedida y no mapea un archivo en disco (mapeo anónimo)
2. La página pertenece a un mapeo de un archivo en disco
3. La página ya fue accedida por el proceso, pero sus contenidos fueron salvados a disco por el mecanismo de reclamación

En el primer caso se debe asignar un frame relleno con ceros al proceso, pero se trata de forma diferente dependiendo de si el acceso fue de lectura o escritura. Si el acceso fue de escritura simplemente se reserva un nuevo frame, pero si fue de lectura, siguiendo el espíritu de la paginación bajo demanda, se difiere aún más la reserva de memoria mapeando la página con permisos de sólo lectura un frame especial relleno con ceros que mantiene el kernel. Como se verá en la próxima sección, el mecanismo de copy on write se encargará de reservar efectivamente un nuevo frame cuando se acceda la página para escritura, provocándose una excepción de fallo.

En el segundo caso la VMA a la cual corresponde la página, tiene definido el método `nopage()` el cual se encarga de retornar un frame que contenga la página de archivo deseada. Si la página corresponde al mapeo de un archivo, el método se encarga de obtener el frame desde el Cache de Páginas que es un concepto que se verá en la sección 2.4.6, pero básicamente se trata de un cache de archivos de disco. Dentro de este caso, también existe la posibilidad de que el mapeo correspondiera a un segmento de memoria compartida, en cuyo caso existe además la posibilidad de que el frame deba ser obtenido desde el mecanismo de Swap que se estudiará en la sección 2.4.7.

En el tercer caso el frame debe ser obtenido desde una área de Swap en disco por medio del Swap Cache, mecanismos que se verán con más detalle cuando se hable de liberación de frames, en la sección 2.4.7.

2.4.5.5. COW (Copy On Write). Durante la creación de un proceso, Linux crea su espacio de memoria como una réplica del espacio de memoria de su proceso padre. El mecanismo de Copy On Write, o COW, difiere la duplicación completa de frames al momento en que éstos sean accedidos, para ahorrar accesos a memoria y ciclos de CPU potencialmente innecesarios. Esto es debido a que el proceso podría incluso ejecutar un programa completamente distinto desechando todo el espacio de memoria heredado. En vez de duplicar los frames (reservando nuevos frames e inicializando su contenido), los mismos son compartidos entre el proceso y su proceso padre. Sin embargo, mientras éstos sean compartidos, no pueden ser modificados, es decir, se modifican las tablas de páginas para que sólo se permitan accesos de lectura. Cuando alguno de los dos procesos intenta escribir en un frame compartido, ocurre una excepción de fallo de página. En este caso el kernel identifica que el frame está presente y los permisos de acceso de la VMA no fueron violados y duplica la página en un nuevo frame, modificando las tablas de páginas para que sea éste el que se accede y se tengan permisos de escritura. El frame original sigue siendo read only y cuando el otro proceso intente escribir en él, el kernel verifica que es el único dueño del frame y simplemente otorga los permisos de escritura.

2.4.6. Cache de páginas. El cache de páginas es el principal cache de disco del kernel de Linux. Un cache de disco es un software que permite al sistema mantener en memoria, datos que normalmente se almacenan en disco, de manera que futuros accesos se hagan rápidamente desde memoria sin necesidad de operaciones de disco. La utilización de estos es crucial para mejorar la performance del sistema, pues accesos repetidos a los mismos datos, son algo muy común.

La particularidad de este cache es que funciona a nivel de páginas completas. Los pedidos de lectura de disco por parte de los procesos de usuario se hacen a través del mismo. En caso de que la página pedida no se encuentre en el cache, se agrega una nueva entrada y se rellena con los datos leídos desde disco. Si no hay escasez de memoria, la página permanece en el cache de forma indefinida, pudiendo ser reutilizada para satisfacer pedidos de lectura de otros procesos, sin acceder a disco. Un mecanismo similar se utiliza en las operaciones de escritura, donde primero se escribe en una entrada del cache de páginas y se difiere la escritura a disco para permitir que se hagan más modificaciones a la misma.

El código y estructuras del kernel no necesitan ser leídos o escritos a disco, por tanto las páginas más comúnmente encontradas en el cache pueden ser de los siguientes tipos:

- datos de archivos corrientes
- directorios, los cuales son tratados en Linux prácticamente como archivos
- datos leídos desde archivos de dispositivos de bloques
- páginas pertenecientes a archivos de sistemas de archivos especiales como el utilizado para memoria compartida de IPC

Como se ve, en todos los casos el cache contiene datos correspondientes a alguna clase de archivo. El mismo se conoce como el propietario de la página y prácticamente todas las operaciones de lectura y escritura a disco de estos, pasan por el cache de páginas.

2.4.7. Liberación de frames.

Hasta ahora se vio como el kernel administra la memoria física y los mecanismos que utiliza para la reserva de la misma y la asignación de ésta a los procesos de usuario. Queda un aspecto más por ver para terminar de comprender el funcionamiento de la memoria virtual, la reclamación de frames. En la sección siguiente se verá por qué necesita el kernel reclamar y liberar frames en algunas situaciones y los métodos que utiliza para lograrlo. Luego, en la sección 2.4.7.2 se verá el subsistema del kernel que se encarga de respaldar las páginas anónimas a disco.

2.4.7.1. PFRA (Page Frame Reclaiming Algorithm). El kernel no hace rigurosos chequeos para asignar memoria a procesos de usuario o caches de disco o memoria usados por el mismo. Esta falta de control es una decisión de diseño que le permite utilizar la RAM disponible de la mejor manera posible. Cuando la carga del sistema es baja, la RAM está utilizada en su mayoría por caches de disco y unos pocos procesos que se benefician de éstos. Por otra parte cuando la carga es alta, la mayoría de la RAM esta ocupada por páginas de procesos y los caches se reducen. Según lo que se vio hasta ahora tanto los caches de disco como los procesos mediante la paginación bajo demanda, piden continuamente frames pero nunca liberan. Esto es razonable porque no es posible determinar cuando un proceso dejará de acceder a estas páginas o forzarlo a liberarlas. Mediante estos mecanismos, eventualmente se agotará toda la memoria libre. Para evitar esto, el Page Frame Reclaiming Algorithm (PFRA) o algoritmo de reclamación de frames de Linux se encarga de rellenar las listas de frames libres del Buddy System, recuperando frames de los procesos y los caches del kernel y garantizando una cantidad mínima de frames libres que le permitan al kernel recuperarse de condiciones de escasez de memoria.

La función principal del algoritmo, es seleccionar los frames candidatos a ser reclamados y llevar a cabo la liberación de los mismos. No se entrará en detalles sobre las heurísticas utilizadas por el mecanismo para la selección de frames, ya que excede los objetivos de la investigación. Por más información sobre este tema, ver [**12**] capítulo 17. Para intentar la liberación efectiva del conjunto de frames seleccionados, se utiliza la función `shrink_page_list()` definida en `{KERNEL}/mm/vmscan.c`. La misma se encarga para cada frame de la lista, verificar si el mismo no cumple un conjunto de condiciones que podrían evitar su correcta liberación y en caso de pasar estos controles, se eliminan los mapeos existentes con procesos de usuario y se devuelve el frame al Buddy System. Algo interesante es el mecanismo utilizado por el kernel para poder llegar desde el frame a todos los mapeos con procesos de usuario. El mismo se denomina "reverse mapping" o mapeo hacia atrás y se basa en el campo `mapping` del descriptor `page` de los frames. Según el tipo de mapeos que se realizan al frame, este campo puede contener un tipo objeto diferente el cual le permite obtener todas las áreas de memoria virtual del sistema que mapean el frame. Para leer más sobre reverse mapping, referirse al capítulo 17 de [**12**].

2.4.7.2. Subsistema de Swapping. El concepto de Swapping fue introducido para brindar un respaldo en disco a las páginas que no están mapeadas a archivos, cuando el frame de memoria que las contiene es liberado por Linux. Existen tres clases de páginas que se encuentran en esta situación y son soportadas por el subsistema:

- páginas que pertenecen a regiones anónimas de memoria de un proceso
- páginas "sucias" (con cambios sin actualizar en disco) de mapeos privados de procesos
- páginas que pertenecen a una región de memoria compartida de IPC

El mecanismo de swapping permite la recuperación transparente y automática de las páginas respaldadas durante el fallo de página, gracias a que se aprovechan los bits no utilizados en una entrada de tabla de páginas cuando la página no está presente, para almacenar un identificador de la página dentro del swap. Las responsabilidades más grandes del mecanismo recaen en:

- preparar "áreas de swap" en el disco para almacenar las páginas que no tienen imagen propia en disco y manejar el espacio en las mismas para reservar y liberar ranuras para contener páginas a demanda. Las áreas de swap se implementan como particiones o simplemente archivos en disco.
- proveer funcionalidades tanto para respaldar páginas desde la RAM a las áreas de swap y para recuperarlas desde un área a la RAM
- incluir identificadores de página de swap en las entradas de tabla de páginas para mantener referencia a los datos en las áreas de swap. Los identificadores constan de dos índices, donde el primero identifica el área de swap donde se encuentra la página y el otro indica la ranura específica dentro de la misma.

El swapping es muy importante porque sin éste no sería posible la liberación de páginas en los casos que éstas no están asociadas a una imagen en disco como sucede con los cache de archivos. Sin dudas que sin esto no sería posible aprovechar el sistema de memoria virtual con paginación bajo demanda y sus beneficios para dar a los procesos la sensación de contar con recursos de memoria más amplios.

Un concepto crucial que permite el correcto funcionamiento de este subsistema es el Swap Cache. Transferir páginas desde y hacia un área de swap es una tarea que

puede provocar muchas race conditions. Por ejemplo, dos procesos podrían intentar recuperar la misma página compartida de forma concurrente, o un proceso podría intentar recuperar una página que está en proceso de respaldo por parte del PFRA. Para evitar estas situaciones, se introdujo el concepto de swap cache. Esta técnica se basa en que nadie puede comenzar una operación de recuperación o respaldo de una página, sin verificar si la misma ya pertenece al swap cache. De esta forma operaciones concurrentes sobre la misma página, se realizan siempre sobre el mismo frame y el kernel puede confiar en que el lock del mismo evita cualquier tipo de race condition.

Capítulo 3

Prototipo

En la sección 2.1 se introdujo el concepto de software aging, se presentaron los problemas que su existencia puede causar en la ejecución de instancias del software y se vieron algunas técnicas y herramientas existentes para combatirlo. Además, se nombro entre las posibles causas de dicho fenómeno la ocurrencia de errores en la memoria, provocados por distintas causas. Se verá en este capítulo el aporte más importante de este proyecto, el cual consta de un conjunto de herramientas a nivel del sistema operativo para combatir el software aging causado por errores en la memoria, desarrollado como una extensión del Linux kernel. En la sección 3.1 se habla de las razones que motivaron la elección de esa área particular de la materia para combatir el aging. En la sección 3.3 se detalla y justifica la elección de la plataforma utilizada.

La meta que se planteó para el prototipo fue brindar a las aplicaciones que ejecutan sobre el sistema operativo, una herramienta para la detección y tratamiento de los errores en la memoria principal que permitieran a las mismas contrarrestar los efectos del envejecimiento. En la sección 3.2, se delinea completamente los objetivos trazados.

Para determinar el grado de aplicación de los objetivos plateados inicialmente en el contexto del núcleo del sistema operativo Linux, se debió hacer una extensa investigación del mismo. En particular, fue necesario tener una importante comprensión del mecanismo de manejo de la memoria que este utiliza. Se resumen los resultados de dicha investigación en la sección 2.4y se encontrará en la sección 3.4 una redefinición más detallada de los objetivos, adaptada a la realidad particular de Linux. El lector encontrará en la sección 2.4, conceptos fundamentales que le permitirán entender mejor las funcionalidades del prototipo. Se dedica a la especificación de estas últimas la sección 3.5.

Teniendo una clara definición del problema y el contexto, se diseñó e implementó una solución al mismo. En la sección 3.6 se dará una visión general de la arquitectura de dicha solución y se verá como esta se integra con el Linux kernel. Luego, en la sección 3.7 se verá en más detalle el diseño y las decisiones que dieron lugar al mismo, las cuales fueron guiadas por una continua investigación del funcionamiento de Linux, llevada a cabo durante todo el proyecto. Finalmente, en la sección 3.8 se verá algunos detalles importantes sobre la implementación del prototipo, los cuales conciernen sobre todo a algoritmia e interacción con las APIs del kernel.

Dado que el prototipo fue realizado como parte del núcleo del sistema operativo, la verificación de las funcionalidades del mismo debieron hacerse también a ese nivel y sorteando las restricciones que esto conlleva. Se dedicará la sección 3.9 al testing de lo desarrollado.

El desarrollo de este prototipo fue realizado por etapas. Las primeras sirvieron para verificar lo investigado y mitigación de riesgos, mientras que las siguientes significaron iteraciones en el desarrollo del prototipo. Se explicará el proceso de desarrollo seguido en la sección 3.11

El prototipo cuenta con una interfaz con el espacio de usuarios. Para explicar cómo se interactúa con esta, se decidió hacer un pequeño manual de usuario el cual se incluyó en el apéndice B.

3.1. Motivación

Como se vio en la sección 2.4, existen múltiples causas del envejecimiento, asociadas a la ocurrencia de errores en el software y hardware. Mientras los errores de software que provocan envejecimiento, son causados básicamente por bugs en el mismo, los errores en el hardware no necesariamente tienen tal origen. En este contexto, el área de testing no provee técnicas que permitan encontrar la causa de los errores y eliminarlos, basta considerar el caso de los errores en la memoria causados por los rayos cósmicos. Aún teniendo conciencia de la ocurrencia de los mismos, no ha sido posible modificar la tecnología utilizada para hacerla invulnerable a dicho fenómeno. Las acciones tomadas al respecto han sido a nivel del hardware, asumiendo la ocurrencia de errores y utilizando técnicas para corrección de los mismos. Dichas soluciones se pueden encontrar, como vimos antes, en las memorias llamadas ECC, las cuales en la actualidad, por razones de costo, no son el tipo más ampliamente usado.

Tomando en cuenta el rol del sistema operativo en un sistema de cómputo como capa de software que abstrae a las aplicaciones de usuario de las complejidades del hardware. Considerando que se concibe al sistema operativo moderno con un esquema de protección de la memoria, al menos para los procesos de usuario, lo cual implica la aplicación de una política de permisos de accesos sobre los espacios de memoria virtual de los mismos. Es posible imaginar soluciones por software que tomando como base esta política, detecten un porcentaje no despreciable de los errores en la memoria física y tomen acciones para prevenir el envejecimiento provocado por los mismos. Incorporando dichas soluciones como servicios del sistema operativo y brindando una interfaz con los mismos en forma de herramientas al espacio de usuarios, se logra una alternativa a la utilización de hardware específico, la cual implica un ahorro de costos y no agrega complejidad a la visión del sistema por parte del usuario. Si estas ideas son combinadas con la existencia de un sistema operativo open source como Linux, se tiene el contexto perfecto para ponerlas en práctica.

La pregunta a responder ahora es: ¿Cuáles serían las áreas de aplicación? Basta con ser usuario de un PC en una ciudad con una mediana elevación sobre el nivel del mar para que los beneficios de estas herramientas comiencen a ser tangibles. No es necesario ir demasiado lejos, si los estudios realizados en Denver con 1600 metros de altura arrojaron valores significativos[**6**], imaginemos cuales pueden ser los resultados en ciudades de países de Sudamerica como Bolivia, Colombia o Ecuador. Existen otros contextos a considerar como los satélites u observatorios ubicados a gran altura donde también se multiplica el efecto de los rayos cósmicos y actualmente se invierte en hardware para solucionar estos problemas. A diferencia de estos casos las PC no suelen contar con dicho hardware por razones de costo y siguiendo

el espíritu de Linux y el software libre en general, ofrecer una solución gratuita al usuario de estos equipos resulta en una meta bastante atractiva.

Además de la aplicación en PCs, otra área que podría verse beneficiada por una solución desarrollada en software es el universo de los sistemas embebidos. Como se mencionó en 2.1.1.4, los mismos son cada vez más comunes para el control de actividades industriales, tecnología aplicada a la medicina, aplicaciones para el hogar y muchos otros contextos de la vida cotidiana. Como se vio en la sección 2.1 la interferencia electromagnética es una causa frecuente de la ocurrencia de soft errors en este tipo de sistemas y las probabilidades de su presencia aumentan considerablemente en ambientes como los industriales. A medida que avanzó la tecnología estos sistemas fueron adquiriendo cada vez mayor poder de cálculo y complejidad, por lo que los mismos empezaron a ser equipados con sistemas operativos para el control de los distintos módulos que los componen. Esta tendencia se volcó en los últimos años a que el sistema operativo seleccionado sea basado en el Linux kernel, por lo que la herramienta planteada también otorgaría soluciones de gran utilidad y rápida aplicación en la práctica para esta área. No solo implicaría reducción en costos por fallos en operaciones industriales o electrodomésticos dañados, sino que podría llegar disminuir riesgos de vida, si se toma en cuenta la utilización de sistemas embebidos en la medicina.

3.2. Objetivos Iniciales

Se plantearon los siguientes objetivos para el prototipo. Los mismos fueron revisados luego de investigar el manejo de memoria de Linux y se definirán con más detalle en la sección 3.4.

- Establecer subconjunto de celdas de la memoria principal para las cuales el sistema operativo garantiza que sólo pueden ser accedidas para lectura exclusivamente.
- Detectar cambios en los bits contenidos dentro de dicho conjunto, los cuales implican soft errors
- Contar con mecanismos que permitan un aceptable poder de corrección de los soft errors detectados
- Brindar mecanismos que asistan la utilización de técnicas anti-envejecimiento tales como el rejuvenecimiento en el espacio de usuarios
- Brindar una interfaz con el espacio de usuarios que permita a los mismos una completa interacción con las herramientas desarrolladas
- Implementar las herramientas como un módulo dentro del kernel, desacoplándolas en lo posible del mismo

3.3. Selección de plataforma y herramientas

Antes de seleccionar la plataforma a utilizar se plantearon una serie de requerimientos que la misma debía de cumplir, los cuales se detallan a continuación:

- Open Source[**15**]: La plataforma debía ser de código abierto de forma de poder trabajar directamente sobre la misma y además poder estudiar completamente el manejo de memoria del sistema.
- Popular: La plataforma a seleccionar debía de ser “popular”. Es decir, que la misma tenga una gran comunidad formada detrás de ella, de forma de poder conseguir documentación detallada sobre la misma y ayuda o soporte en caso de ser necesario.

- Conocimiento Previo: De ser posible se debe seleccionar una plataforma sobre la que se tenga conocimiento previo de su kernel y/o se esté familiarizado con las herramientas de desarrollo disponibles para la misma.

Luego de evaluar estas condiciones se decidió seleccionar como plataforma para este proyecto el Linux kernel, en particular la distribución OpenSuse 11.0 [**25**] con versión del kernel 2.6.25.9.

3.3.1. ¿Por qué de la selección? Se seleccionó un sistema operativo basado en el Linux kernel debido a que el mismo cumple con los tres requerimientos planteados, este kernel se distribuye bajo la licencia GPL, la cual es una licencia Open Source que permite trabajar libremente con el código del mismo teniendo acceso a todas sus secciones sin restricción alguna. La misma licencia que exige que el código de Linux sea distribuido, hace lo propio con todas las herramientas para poder compilar su código con las modificaciones realizadas sobre este, lo cual simplifica el trabajo con el kernel.

Linux es además el más popular entre los kernels de código abierto, teniendo una gran comunidad detrás de él desde la cual se puede conseguir soporte para posibles problemas que surjan. También cuenta con el respaldo de varias empresas y corporaciones que se encargan de apoyar su desarrollo y mantenimiento. Además, por su popularidad, Linux es utilizado en ámbitos educativos para materias vinculadas a los conceptos de sistemas operativos por lo que existe mucha documentación sobre su funcionamiento, tanto de carácter introductoria como avanzada. Al ser utilizado con fines educativos, el mismo es conocido por todos los integrantes del proyecto.

Otra ventaja que posee es que al estar diseñado para que su kernel sea reemplazado cuando se actualiza las distribuciones de Linux incluyen herramientas para facilitar esta tarea. Un beneficio extra, es la portabilidad de Linux. Esto implica que se puede utilizar en varias arquitecturas de hardware distintas. Como para el desarrollo de este prototipo no fue necesario modificar secciones del código dependientes de la plataforma, todo lo desarrollado puede utilizarse igualmente tanto en una PC basada en x64, un server Sparc o un sistema embebido ARM.

Se seleccionó la versión 2.6.25.9 del kernel porque la misma era la más actual al momento de iniciar el desarrollo del proyecto. Además se decidió utilizar la distribución OpenSuse 11.0 [**25**] ya que la misma al momento de iniciar el desarrollo era distribuida con el kernel 2.6.25.1.1 lo que facilitaba la sustitución de ese kernel con el seleccionado ya que se sabía que el sistema era compatible con los kernels basados en la versión 2.6.25.

Luego de seleccionar la versión del kernel a utilizar se debía seleccionar la plataforma de hardware a utilizar para el desarrollo del proyecto. Debido a que al momento de comenzar, se empezó utilizando máquinas virtuales para el desarrollo, se decidió utilizar como plataforma de hardware la tecnología x86, la cual es una de las soportadas por la mayoría de las máquinas virtuales a disposición y que además se utiliza en la gran mayoría de las PCs del mundo, por lo que permitía que todo los integrantes tengan acceso a hardware compatible y que además el producto final pueda ser utilizado en la mayoría de los equipos disponibles. El proyecto fue realizado para sistemas x86, pero la mayoría de los puntos en los que se interconectó el kernel original con el código del proyecto es el mismo para todas las versiones del kernel, lo único que no se puede asegurar para las distintas arquitecturas es

que todo funcione correctamente ya que las pruebas se realizaron solamente para la versión utilizada durante el desarrollo.

3.3.2. Herramientas. La selección de la plataforma fue determinante para la selección de las herramientas a utilizar para el desarrollo del prototipo. Se hará aquí mención de la mayoría de las herramientas utilizadas, justificando especialmente la necesidad de algunas que se consideraron más importantes.

3.3.2.1. Programación. Tomando en cuenta que el Linux kernel está desarrollado en C, la selección del lenguaje de programación no era una variable. Dada la elección de OpenSuse 11.0 como sistema operativo y tomando en cuenta que éste trae en su instalación el gcc versión 4.3.1, compatible con los fuentes utilizados, también resulto trivial la elección del compilador a utilizar. De esta forma quedó por definir simplemente un IDE o editor, para lo que no se fijó un estándar debido a que la dinámica de trabajo implicaba la edición de unos pocos fuentes individuales dentro del kernel y no se entendió necesario el manejo del proyecto como un todo. En la práctica en ambiente Linux se trabajó con vi y cuando fue necesario editar desde un equipo Windows se utilizó el editor gratuito de código fuente Notepad++ [**39**].

3.3.2.2. Virtualización. Uno de los aspectos clave de la forma de trabajo seleccionada fue la utilización de máquinas virtuales durante el desarrollo y buena parte de las pruebas del prototipo. Para esto se creó una imagen de máquina virtual básica para las tecnologías VMware [**38**] con OpenSuse 11.0 para 32bits, la cual se ejecutaba por medio de la herramienta VMware Player versión 2.0.2. Las distintas máquinas virtuales que se utilizaron se ejecutaban normalmente desde un sistema host Windows y se usó ssh como forma de comunicación por medio de los clientes Putty[**40**] y WinScp[**41**].

El uso de esta tecnología fue clave, ya que permitió trabajar de forma segura y contenida con las versiones modificadas del kernel. No se debe olvidar que se hicieron cambios en la programación del núcleo del sistema operativo y que en muchas ocasiones, como se comprobó en la práctica, errores de programación pueden causar graves bugs en el sistema o incluso no permitir el correcto arranque del mismo. De esta forma se hizo notoriamente más simple y seguro recuperarse de esos problemas en una máquina virtual, que brinda mayores flexibilidades que la opción de ejecutar directamente sobre el hardware. Además, mediante la máquina virtual es posible simular cambios en la configuración del hardware sin necesidad de tocar el mismo. Un buen ejemplo de esto es que, debido a estar trabajando con la memoria física, en muchos casos fue necesario modificar la cantidad disponible para evaluar las respuestas del sistema ante distintos valores.

3.3.2.3. Distribución y respaldo. Otro aspecto importante fue el mecanismo utilizado para distribuir cambios y control de la configuración. Una vez más el hecho de trabajar con el Linux kernel implicó que ya estuvieran definidos los mecanismos estándar para realizar estas tareas. Los cambios al Linux kernel suelen distribuirse en forma de parches, que son la salida del programa *GNU diff* en un formato que es legible por el programa *patch*. De esta forma se utilizó el formato de parches para distribuir y controlar las distintas versiones del prototipo y para propagar cambios hechos al mismo en los distintos ambientes.

Para respaldar, distribuir y compartir los fuentes se utilizó además un repositorio cvs remoto, accedido desde el intérprete de comandos en Linux.

3.3.2.4. Comunidad Linux. Es necesario hacer una mención especial a la comunidad Linux, la cual fue de gran ayuda en distintos aspectos. Se accedió a varios sitios con información sobre tecnologías del kernel, discusión de cambios entre versiones y otra documentación. También se accedió a varios foros oficiales y no oficiales donde se encontraron soluciones a algunos problemas. Otra herramienta de gran utilidad fueron los llamados "kernel identifier search" que brindan una forma práctica y rápida de recorrer las distintas versiones de los fuentes de Linux, extremadamente útil a la hora de investigar en profundidad el código. En general, todas estas herramientas y otras existentes comparten el concepto de colaboración, fundamental en el mundo open source.

3.4. Revisión de Objetivos

Entre los objetivos planteados, la determinación del subconjunto de celdas de memoria de sólo lectura, es el que se encontraba más abierto y más dependía de los resultados de la investigación del manejo de memoria en Linux para su formulación específica. Además, tiene la característica particular de que todos los otros objetivos dependen del él. Se resumen a continuación las conclusiones más importantes sacadas en base a los conocimientos adquiridos:

- Linux administra la memoria con granularidad de frames y por tanto son éstos la unidad de los elementos del subconjunto de la misma con el que se desea trabajar
- El kernel confía en sí mismo y por tanto no se mantiene un esquema de protección en su espacio de memoria. Sí se cuenta con uno para los frames asignados al espacio de memoria de los procesos de usuario y el mismo se implementa a nivel de la memoria virtual, por medio de las áreas de memoria y las tablas de páginas. De esta forma concluimos que el conjunto de frames de sólo lectura es un subconjunto de los asignados a los espacios de memoria de procesos de usuario.
- Un frame puede estar asignado a los espacios de memoria de más de un proceso de usuario y los mismos pueden tener distintos permisos de acceso sobre éste. De esta forma se determina que un frame pertenece al subconjunto deseado, si y sólo si, esta mapeado al espacio de uno o más procesos de usuario con permisos de sólo lectura en todos los casos.

Como resultado, se podría redefinir los dos primeros objetivos como:

- Determinar el subconjunto de frames que se encuentran mapeados a uno o más espacios de memoria de procesos de usuario, con permisos de lectura exclusivamente en todos los casos.
- Detectar cambios en los bits de la página contenida dentro de cada frame del conjunto, los cuales implican la ocurrencia de soft errors

3.5. Especificación Funcional

Se dedica esta sección a especificar de forma clara y concisa desde el punto de vista funcional, las herramientas diseñadas para combatir los errores provocados por el software aging a nivel de sistemas operativos y que fueron implementadas en la versión final del prototipo en Linux.

3.5.1. Frames de sólo lectura. Se mantiene identificado a lo largo de la ejecución del sistema operativo un subconjunto particular de los frames utilizables del sistema. El mismo está compuesto por aquellos que sólo pueden ser accedidos para lectura, por parte de los procesos del espacio de usuarios que los utilizan. Se actualiza el contenido del conjunto en cada uno de los eventos en los cuales existe un cambio en el conjunto de procesos que puede acceder a un frame, o los permisos con los cuales éstos lo hacen.

3.5.2. Procesos Registrados. Se mantiene un conjunto de grupos de procesos registrados. El mismo está compuesto por grupos de procesos los cuales deben contar con al menos un hilo vivo. Estos grupos pueden ser agregados y removidos explícitamente del conjunto por parte de cualquier proceso del espacio de usuarios, incluyendo el intérprete de comandos de usuario. Se creó una interfaz con dicho espacio para este fin. No se cuenta con ninguna política para agregar grupos al conjunto de forma automática. Los mismos pueden ser removidos automáticamente del conjunto cuando muere el último hilo del grupo o el proceso que se registró como agente del mismo. Se denomina agente al proceso que se indica como destinatario de notificaciones referentes a eventos detectados sobre grupo. El agente puede o no ser parte del grupo que monitorea.

3.5.3. Detección de errores. Se cuenta con un algoritmo para la detección de errores en los frames del conjunto mantenido. El mismo se basa en la utilización de un código de detección de errores. En cada evento que origina el ingreso de un frame al conjunto de sólo lectura, se calcula y almacena el código para dicho frame. El código será válido para ser verificado en busca de error mediante el algoritmo mientras el frame permanezca dentro del conjunto. En cada evento de salida de un frame del conjunto de sólo lectura se deja de almacenar el código para el mismo.

3.5.4. Corrección de errores. Se cuenta también con un algoritmo para la corrección de errores en frames. De forma similar a la detección, se utiliza un código criptográfico. El mismo tiene poder de corrección de error en 1 bit. En cada evento que origina el ingreso de un frame al conjunto de sólo lectura, se calcula y almacena el código para dicho frame. El código será válido para ser utilizado para la corrección de un error mediante el algoritmo mientras el frame permanezca dentro del conjunto. En cada evento de salida de un frame del conjunto de sólo lectura se deja de almacenar el código para el mismo.

3.5.5. Búsqueda de errores. Se cuenta con un conjunto de mecanismos para la búsqueda de errores en frames, mediante el algoritmo de detección definido, los cuales se detallan a continuación.

Se creó un demonio el cual ejecuta de forma continua ciclos de búsqueda de errores en el sistema. En cada ciclo se utiliza una de dos estrategias posibles para recorrer el conjunto de frames de sólo lectura y se verifica la correctitud de cada frame mediante el algoritmo de detección de errores. La primer estrategia recorre todos los frames del conjunto en el orden que se encuentran estos en el espacio de direcciones físicas (en Linux este orden esta dado por el page frame number o pfn). La segunda estrategia recorre el conjunto de grupos de procesos registrados por orden de llegada al registro. Para cada hilo dentro de alguno de estos grupos se verifican los frames a los que se tiene permiso de acceso por orden de dirección virtual dentro del espacio de memoria del hilo en cuestión.

Se cuenta con un segundo mecanismo de búsqueda de errores el cual verifica los frames del próximo hilo a obtener el procesador, según el algoritmo de planificación de procesos. Sólo se chequea el hilo si el mismo pertenece a un grupo de procesos registrado. Este mecanismo verifica los frames accedidos por dicho hilo por orden de dirección virtual en su espacio de memoria, mediante el algoritmo de detección de errores.

3.5.6. Rejuvenecimiento de archivos. Se cuenta con un mecanismo de rejuvenecimiento de frames desde disco en caso de que el frame afectado corresponda a la imagen o cache en memoria de un archivo en disco. El mecanismo intenta obtener una copia sin errores del contenido completo del frame desde la página correspondiente en el archivo de disco.

3.5.7. Secuencia de acciones frente a errores. En todos los casos en que se detecte un error, sin importar el mecanismo de búsqueda, se ejecuta una única secuencia de acciones para el tratamiento del mismo. Primero se intenta corregir el error mediante el algoritmo de corrección de errores. Si no se tuvo éxito, lo cual implica que el error es de más de un bit, en caso de que el frame afectado corresponda a la imagen o cache en memoria de un archivo en disco, se intenta rejuvenecer el mismo. Finalmente, sin importar el resultado de los pasos anteriores, se publica mediante una interfaz con el espacio de usuarios, para cada hilo que accede al frame, información detallada sobre la ubicación del error detectado y el resultado de las acciones tomadas para resolverlo. Los datos publicados para cada hilo son los siguientes:

- Número de frame donde ocurrió el error
- Dirección virtual del frame en el espacio de memoria del hilo actual.
- Comienzo, Fin y permisos de acceso (lectura, escritura, ejecución) del segmento de memoria virtual al cual pertenece el frame en el espacio virtual del hilo en cuestión.
- Si el frame corresponde al mapeo o cacheo de un archivo en disco, se muestra el nombre del mismo y número de página que mapeaba el frame dentro de éste. Si no corresponde a archivo simplemente se indica.
- Estado final del error luego de las acciones: solucionado o no.

Además, se envía una señal especifica (a efectos de notificar el evento) a los agentes de aquellos hilos que pertenezcan a algún grupo de procesos registrados y tengan permitido acceder al frame donde se detectó el error. El conjunto de las notificaciones y publicación de detalles de error por hilo, permiten la aplicación de múltiples técnicas para combatir este tipo de envejecimiento mediante rejuvenecimiento.

3.5.8. Configuración de la herramienta. Se mantiene un modo global que permite determinar la configuración de ésta. El modo está compuesto por un conjunto de banderas cada una de las cuales tiene dos valores posibles. Pueden estar activadas o desactivadas y su significado puede ser la habilitación o no de una funcionalidad, o la elección entre dos estrategias. El modo puede ser modificado en caliente, para cambiar el comportamiento del sistema. Se crearon las siguientes banderas para configurar el mecanismo de búsqueda de errores:

- *PGSA_CHECK_DAEMON*: si está activada se utiliza el mecanismo de búsqueda de errores del demonio, si no se deshabilita el mismo.

- *PGSA_CHECK_CURRENT*: si está activada se utiliza el mecanismo de búsqueda de errores que verifica las páginas de sólo lectura que están mapeadas al proceso seleccionado por el planificador de Linux para ser el próximo a obtener el procesador. En caso de que la bandera esté desactivada no se habilita el mecanismo.
- *PGSA_CHECK_ALL*: si está desactivada los mecanismos de búsqueda de errores existentes se aplican sólo a los frames mapeadas a procesos registrados y si está activada se aplican a todos los frames del sistema.

Además, se cuenta con las siguientes banderas para configurar las acciones a tomar en caso de detectar un error, mediante los mecanismos de búsqueda habilitados:

- *PGSA_CORRECTION_CODE*: si está activada se habilita el mantenimiento a nivel de frames de la redundancia del algoritmo de corrección y se habilita la aplicación del mismo.
- *PGSA_FILE_REJUVENATION*: si está activada / desactivada se habilita / deshabilita respectivamente, la acción de rejuvenecimiento automático de páginas mapeadas a archivos.
- *PGSA_NOTIFICATIONS*: si está activada / desactivada se habilita / deshabilita respectivamente, la acción de envío de notificaciones a el/los procesos que mapean un frame sobre el cual se detectó un error.
- *PGSA_SIGNAL*: si esta activada / desactivada se habilita / deshabilita respectivamente, la acción de envío de notificaciones sincrónicas o señales a el/los procesos agentes de procesos registrados que mapean un frame sobre la cual se detecto un error.
- Finalmente, a nivel general del módulo, se cuenta con la flag *PGSA_ON* que en caso de estar desactivada deshabilita completamente las modificaciones realizadas al kernel.

3.5.9. Interfaz con espacio de usuarios. Se cuenta con una interfaz con el espacio de usuarios que agrupa un conjunto de funcionalidades que permiten a los mismos interactuar con el modulo. Las mismas se describen a continuación.

Se permite consultar y modificar las banderas de modo por medio de operaciones de lectura y escritura, validando la coherencia del modo seleccionado en cada caso y modificando de forma inmediata el comportamiento del modulo para que cumpla con lo definido para ese modo.

Se permite realizar el registro de un grupo de procesos y el agente correspondiente, indicando los identificadores de proceso de ambos.

Se permite consultar desde el espacio de usuarios algunas estadísticas globales sobre la ejecución del módulo. Las estadísticas publicadas son las siguientes:

- cantidad de ciclos de búsqueda de errores realizados por el módulo
- cantidad de errores detectados por el módulo
- cantidad de frames verificados en el último ciclo, es decir, tamaño del conjunto de frames de sólo lectura

Se permite consultar para cada hilo en cuyo espacio de memoria se haya detectado un error, la lista de detalles que describen el error.

3.6. Arquitectura

Como primera aproximación a la solución desarrollada y con el propósito de trazar en la misma el cumplimiento de los objetivos planteados y la implementación

de las funcionalidades descriptas, se dedicará esta sección a mostrar y explicar la arquitectura seleccionada. Otro aspecto importante que se muestra, es la interacción del módulo desarrollado con el resto del kernel.

Se empezará diagramando los componentes y su interacción en la próxima sección, para luego dividir la discusión en partes. En la sección 3.6.2 se dirigirá la atención a las interfaces externas del módulo con el sistema operativo. Luego, en la sección 3.6.3 se verá la comunicación del mismo con el espacio de usuarios. Finalmente, se dedicará la sección 3.6.4 a dar una visión general de los componentes internos, la cual servirá como introducción a la sección de diseño.

3.6.1. Diagrama. El diagrama presentado en la figura 8 no sigue formatos clásicos de un diagrama de arquitectura, sino que se utiliza una representación más informal similar a la idea utilizada en http://www.makelinux.net/kernel_map para hacer un mapa del Linux kernel. Se decidió utilizar la misma, dado que gráfica muy bien la integración del módulo con el kernel, sin hacer una recorrida exhaustiva de todas las APIs del mismo utilizadas.

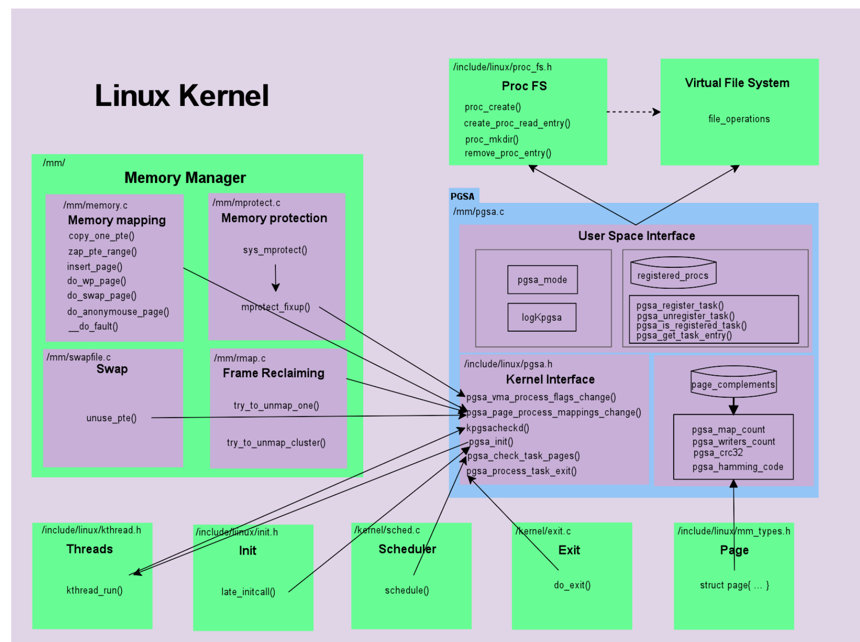


Figura 8. Diagrama de Arquitectura

3.6.2. Interfaz con el Linux kernel. Parece contradictorio hablar de comunicación con el kernel cuando se están modificando algunos de los mecanismos de más bajo nivel del mismo para brindar un conjunto de nuevas funcionalidades. Sin embargo, basándose en el objetivo de desacoplar lo más posible la implementación de dichas funcionalidades del resto del kernel, se entendió imprescindible reducir las modificaciones necesarias, a la invocación dentro del kernel de un conjunto reducido de handlers definidos como parte de una interfaz para este fin. Por medio de

los mismos se logra encapsular todas las acciones que debe tomar el módulo ante eventos internos del kernel, evitando realizar grandes modificaciones a los fuentes de la distribución del mismo e incluyendo la mayor parte de la implementación en fuentes propios. Se discuten a continuación las operaciones que forman parte de esta interfaz.

- `pgsa_init()`*:* Es la encargada de la inicialización de estructuras, variables, interfaces de usuario y procesos del módulo. Es invocada durante las últimas etapas de inicialización del kernel.
- `pgsa_page_process_mappings_change()`: Para el mantenimiento del conjunto de frames que son accedidos por procesos con permisos de sólo lectura, es necesario procesar cada evento dentro del kernel en el cual se produce un cambio en el mapeo entre el espacio de memoria de un proceso y la memoria física. En particular este manejador se encarga de los casos en que se crea o elimina un mapeo a un frame, actualizando el conjunto de frames y ejecutando las acciones correspondientes para aquellos elementos que entren o salgan del conjunto.
- `pgsa_vma_process_flags_change()`*:* Este manejador corresponde a eventos de la misma clase que el anterior. No sólo es posible que un frame entre o salga del conjunto debido a que se creó o eliminó un mapeo entre el mismo y un espacio de memoria. También es posible que esto se origine por cambios en los permisos con los que un proceso puede acceder a la página contenida en el frame. Este handler se ocupa de procesar dichos eventos.
- `pgsa_check_task_pages()`: El chequeo en búsqueda de errores en el espacio de memoria del próximo proceso a ejecutar, es accionado por medio de este handler desde el planificador de procesos de Linux, en el momento en que se selecciona la tarea candidata por medio del algoritmo de planificación.
- `pgsa_process_task_exit()`: Dado que se mantiene un conjunto de grupos de tareas registradas y agentes que reciben notificaciones de eventos sobre las mismas, es imprescindible manejar la actualización automática de dicho conjunto para garantizar su integridad, durante la finalización de un proceso. Este handler toma las acciones necesarias sobre el registro de grupos de procesos ante la terminación del último proceso de un grupo o el agente que monitorea al mismo.

Las operaciones anteriores cubren las notificaciones desde el kernel hacia el módulo PGSA, pero la comunicación también se da en el otro sentido. Existen dos formas en que el módulo utiliza interfaces del kernel y ambas están implícitas en el diagrama de la arquitectura presentado. Por una parte se tiene que el kernel es en sí un sistema de gran porte compuesto por muchos módulos los cuales tienen sus propias interfaces. Sin ir más lejos, el manejador de memoria provee una gran cantidad de métodos para que el resto del kernel interactúe con él. El módulo PGSA no es la excepción y hace uso de distintas interfaces del manejador de memoria y sus componentes, entre otros módulos, para la implementación de las funcionalidades brindadas. Por otra parte, dado que ese sistema no es nada menos que el núcleo de un sistema operativo, no cuenta con la posibilidad de delegar rutinas estándar como el manejo de cadenas y colecciones, a bibliotecas como la de C. Este tipo de bibliotecas están disponibles sólo en el espacio de usuarios. Dado que el kernel necesita hacer uso de una gran cantidad de rutinas estándar, el mismo incluye una biblioteca propia

que brinda un poder similar a las de espacio de usuario. El módulo PGSA también utiliza la API del kernel para acceder a dichas funcionalidades, implicando esto otra forma de acoplamiento con el mismo.

3.6.3. Interfaz con el espacio de usuarios. Para la interfaz con el espacio de usuario se decidió también modificar lo menos posible el código ya existente del kernel, y de ser posible realizar toda la comunicación desde dentro del módulo a desarrollar en este proyecto. Para esto se decidió que convenía utilizar algún sistema de los que provee el mismo kernel para la comunicación entre el espacio de kernel y el espacio de usuario. Este sistema no sólo tenia que permitir la publicación de información desde el kernel hacia el espacio de usuario sino que también debía soportar que el usuario enviara información hacia el kernel.

Por medio de este sistema se va a realizar la publicación de estadísticas de funcionamiento, así como el reporte de los errores detectados y permitir que el usuario cambie el modo de funcionamiento del módulo y registre procesos a verificar, junto con sus agentes.

3.6.4. Módulo PGSA. Se decidió desarrollar las herramientas y funcionalidades propuestas como parte de un único módulo llamado PGSA. El mismo brinda una interfaz para recibir eventos desde el kernel y un conjunto de callbacks para manejar la comunicación con espacio de usuario por medio de mecanismos bien conocidos, provistos por el kernel, para este fin. Se entendió que por las tareas realizadas y su fuerte relación con el manejo de memoria, que el módulo debería ser un subsistema del manejador de memoria de Linux (MM). Esto implica que en una instancia del kernel con las modificaciones hechas para el proyecto, el módulo se encuentra linkeado como parte de dicho componente y de hecho sus fuentes se incluyen bajo `{KERNEL}/mm/` donde se encuentra el resto del MM.

Cuando se habla de módulo, no se refiere al concepto de módulo dinámico que maneja el kernel. Un módulo dinámico es un componente que puede agregar funcionalidad a una instancia activa del kernel, ejecutando en un equipo. Este concepto es usado principalmente para el desarrollo de drivers de dispositivos. En el caso del módulo PGSA, sólo se puede integrar sus funcionalidades a un kernel en tiempo de compilación. Esto es debido a los puntos dentro del kernel donde se invoca la interfaz del módulo y a algunas APIs de Linux que son utilizadas, las cuales sólo pueden ser accedidas en este contexto.

El diseño e implementación de los handlers y callbacks en las interfaces, así como de los componentes internos del módulo se verán en secciones dedicadas a esto.

3.7. Diseño

En esta sección se define el diseño de la solución implementada. Al finalizar la misma se habrá justificado como se logra brindar cada una de las funcionalidades especificadas y las operaciones que componen las interfaces presentadas. La organización de esta sección del documento se basa en las funcionalidades y por tanto se encontrarán secciones asociadas a cada una de ellas. Se comienza sin embargo, por una pequeña sección que explica algunos componentes del módulo generales a varias de éstas. Otros aspectos generales a todo el prototipo se encuentran en el modo y el registro de los procesos, a los cuales por su importancia se les dedica secciones separadas, a saber la 3.7.4 y 3.7.3 respectivamente.

Una de las funciones del Sistema Operativo es brindar a los procesos de usuario un esquema de protección de la memoria. Como vimos en la sección 2.4, Linux utiliza la paginación como principal mecanismo de direccionamiento, adaptándose a los mecanismos de paginación por hardware provistos por las distintas arquitecturas. Además, vimos que para los procesos de usuario se utiliza un esquema de protección de la memoria virtual en su espacio de memoria, descrito en las Virtual Memory Areas (VMAs) que representan regiones de dicho espacio. A su vez, es a nivel del mecanismo de paginación, mapeando el esquema de protección virtual al hardware es que se garantiza el cumplimiento del mismo en cada acceso de memoria del proceso.

La solución planteada al problema de detección por software de errores en la memoria física se basa en la hipótesis de restringir el área de trabajo a aquellos frames de memoria que están mapeados con permisos de sólo lectura a procesos de usuario. Si se puede asegurar que se trabaja sobre regiones de memoria donde no habrá cambios originados por el software ejecutado en el equipo, se está en condiciones asumir que cualquier cambio detectado dentro de dichas regiones tiene su origen en el hardware. Se verá como se determinó el conjunto de dichas regiones en la sección 3.7.2. El segundo problema a resolver para la detección de errores es determinar un mecanismo para detectar cambios ocurridos en los frames del subconjunto mencionado durante su inclusión en el mismo. Se verá esto en la sección 3.7.5. Fue necesario además, contar con distintos mecanismos para la búsqueda de errores por medio del sistema de detección utilizado. Se estudiará el diseño de dichos mecanismos en la sección 3.7.6. También fue necesario proveer una variedad de acciones a tomar para solucionar los errores detectados o contrarrestar los efectos adversos que pudieran causar los mismos en el correcto funcionamiento del sistema. Se atacó este último problema utilizando dos enfoques, el primero más sencillo y directo, se basa en utilizar métodos de corrección de errores y lo se verá en la sección 3.7.7. El segundo enfoque se basa en los conceptos de rejuvenecimiento de software y su aplicación a nivel de procesos y sistema operativo. Se derivaron dos mecanismos de estos conceptos, el primero de los cuales consiste en el rejuvenecimiento automático y transparente de las páginas por parte del sistema operativo y será estudiado en la sección 3.7.8. El segundo mecanismo, es tal vez el que más se apega a los conceptos vistos en el estado del arte, pues se enfoca en el principio de que el rejuvenecimiento ocurre en el espacio de usuario. Dicho mecanismo se verá con más detalle en la sección 3.7.9.

Finalmente, luego de comprender todas las funcionalidades de la lógica implementada por el módulo, se dedica la última sección de este punto a la capa de presentación de las mismas hacia el espacio de procesos de usuario.

3.7.1. General. En este punto se cubrirá un aspecto del diseño del componente que no es particular a ninguna de las funcionalidades o interfaces del mismo. La implementación de dichas funcionalidades, requirió el mantenimiento de alguna información de estado sobre cada frame. En la sección 2.4 se explicó que para la administración de los mismos, Linux cuenta con un descriptor llamado *page*. Este descriptor, no cuenta en la distribución de Linux, con la información necesaria para soportar el diseño del módulo PGSA, por lo que se decidió definir un complemento del mismo. En la figura 9 se grafica dicho complemento, reservando la discusión de los campos particulares, para las secciones correspondientes donde se justifica su

definición.

FIGURA 9. Complemento de struct page

3.7.2. Conjunto de frames de sólo lectura. El conjunto de frames de sólo lectura se mantiene como una colección virtual de frames, que cumplen una condición particular. Se habla de colección virtual porque no se agrupan los mismos mediante una estructura de datos, sino que se decidió mantener para cada frame de memoria un estado que permitiera determinar su pertenencia al conjunto. Dado que lo que se busca son aquellos frames de memoria que estén mapeados al espacio de memoria de uno o más procesos de usuario, se decidió mantener para cada frame un contador de mapeos de dicha clase. A su vez interesa saber si todos los mapeos son con permisos de sólo lectura, es decir que no hay mapeos que permitan la escritura sobre el frame. Para esto se decidió mantener para cada frame un contador de mapeos de procesos con permisos de escritura. Los contadores mencionados determinan para cada frame tres estados posibles y conjunto de transiciones entre los mismos. En la figura 10 se grafica la máquina de estados generada, mostrando las transiciones más importantes.

El primer estado es el caso inicial cuando frame no tiene ningún mapeo y por tanto ambos contadores valen cero y no pertenece al conjunto de trabajo. El segundo es cuando existen mapeos y por tanto el primer contador es mayor a cero, pero ninguno es de escritura por lo cual el segundo contador vale cero. Este estado es el que determina los frames que pertenecen al conjunto. Finalmente, tenemos un tercer estado que se da cuando existen mapeos y al menos uno de ellos tiene permisos de escritura, por lo cual ambos contadores son mayores que cero y el frame no pertenece al conjunto.

Como se vio en la figura existe un conjunto de transiciones posibles entre estos estados, de acuerdo a cómo se mueven los contadores. Por ejemplo, se puede pasar al estado que indica pertenencia al conjunto cuando llega el primer mapeo y es de lectura o cuando se elimina el último mapeo de escritura y todavía existe uno o

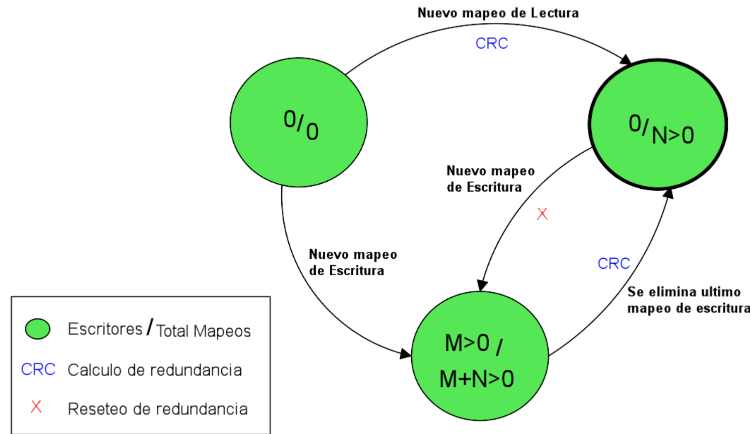


FIGURA 10. Maquina de estados de un frame

más de lectura. A su vez, también se puede encontrar transiciones hacia estados que indican no pertenencia al conjunto de sólo lectura, cuando llega un mapeo con permisos de escritura o cuando se elimina el último mapeo existente. Estas transiciones son disparadas en los eventos de cambios en los mapeos que se estudiará a continuación. Dentro de las transiciones posibles, tienen mayor importancia aquellas que pasan desde un estado que determina pertenencia al conjunto de trabajo a uno que determina no pertenencia y viceversa. Se verá cuando se explique el diseño de los manejadores de cambios en los mapeos y los permisos definidos en la interfaz con el kernel, que estas transiciones son las que desencadenan acciones referentes a la detección de cambios, mientras las restantes simplemente actualizan el estado.

3.7.2.1. Eventos de cambios en los mapeos. Para mantener actualizado el estado de cada frame y determinar su pertenencia al conjunto de sólo lectura, fue necesario determinar aquellos eventos en el funcionamiento del kernel en los cuales se producía alguna modificación en el mapeo entre los frames de la memoria física y el espacio de memoria de algún proceso de usuario. Para la determinación de dichos eventos, se decidió investigar desde el punto de vista de un proceso cuantos frames tenia mapeados en cada momento y cuántos de estos con permisos de sólo lectura (el proceso en particular y no todos los que lo mapean). Se encontró que el kernel ya mantenía esa información a nivel de los espacios de memoria de procesos de usuario por medio del número `rss` (resident set size), el cual se calcula como la suma de los valores `anon_rss` (cantidad de mapeos de frames a páginas contenidas en áreas de memoria anónimas del proceso) y `file_rss` (cantidad de mapeos de frames a páginas contenidas en áreas de memoria mapeadas a archivo del proceso). Recorriendo en los fuentes del kernel los puntos donde se modifican estos contadores e investigando cada uno de ellos, fue posible determinar los eventos en donde se reservan y liberan frames para un espacio de memoria. De esta forma, se determinó la mayoría de los eventos deseados, sin embargo, quedaba aún un aspecto por considerar. Se contempló la posibilidad de que hubiera cambios en los permisos

durante el ciclo de vida del mapeo entre la página virtual del proceso y el frame. Investigando el kernel se encontró que esto era posible, obteniéndose los eventos de cambios en los permisos de un mapeo. A continuación se estudiará cada uno de los casos encontrados.

Áreas de memoria virtual. Como se vio en la sección 2.4.5, dedicada al espacio de memoria de los procesos de usuario en Linux, los mismos están representados por un conjunto de áreas o regiones de memoria. Como se explica en la sección 9.3 de [**12**] se cuenta con un conjunto de operaciones para el manejo de las mismas. En particular se cuenta con la operación `do_mmap()` que se encarga de crear e inicializar una nueva área de memoria para el proceso actual y `do_unmap()` que borra un intervalo lineal de direcciones del espacio liberando los mapeos a frame existentes para el mismo. En el caso de `do_mmap()` es posible especificar mediante la bandera `VM_LOCKED` que las páginas de la nueva área deben estar fijas en la memoria física. Para esto, se invoca la primitiva `make_pages_present()` para mapear el área a frames y se fijan las páginas a la RAM. El trabajo efectivo de mapeo origina invocaciones a `handle_mm_fault()` la cual, como veremos más adelante, forma parte de la implementación de la paginación bajo demanda, en el manejador de fallos de página. Por su parte `do_unmap()` busca y elimina del espacio de memoria todas las áreas que forman parte del intervalo a borrar. Luego de esto, actualiza las tablas de páginas mediante la función `unmap_region()`, la cual hace uso de `unmap_vmas()` definida en `{KERNEL}/mm/memory.c` para la eliminación efectiva de los mapeos a frames, el cual es realizado con una sucesión de invocaciones a `unmap_page_range()`. La primitiva `unmap_page_range()` al igual que las anteriores mencionadas, forma parte de las APIs del manejador de memoria de Linux. La misma se encarga de modificar los distintos niveles de tablas de páginas para eliminar un intervalo del espacio virtual del proceso. Esta tarea se hace de forma recursiva en cada nivel de tablas por medio de una jerarquía de funciones de la forma `zap_XXX_range()`, donde XXX es la denominación de cada nivel de tabla de páginas. Estas trabajan invocando cada una a la correspondiente al siguiente nivel para todas las entradas implicadas. La última en la cadena es `zap_pte_range()` que trabaja a nivel de las entradas de tabla de páginas (o page table entry) que cabe recordar, referencian directamente a las páginas. Se identificó en el código de esta función el primer punto efectivo donde se originan eventos de cambio en los mapeos, eliminación en este caso, uno por cada página y frame del rango efectivamente desmapeados.

Linux cuenta además con la operación `vm_insert_page()` definida también en `{KERNEL}/mm/memory.c` creada para permitir a drivers de dispositivos insertar en un espacio de memoria virtual de usuario frames individuales que ellos reservan. En realidad es una operación que sólo se mantiene por compatibilidad con drivers antiguos y sus funciones son ahora llevadas a cabo por `remap_pfn_range()` definida en `{KERNEL}/mm/memory.c`. Se investigó `vm_insert_page()` y se encontró que el trabajo efectivo de inserción del frame en el espacio de memoria lo realiza la función `insert_page()`. De esta forma, se identificó aquí un nuevo evento, en este caso de creación de mapeos.

Paginación bajo demanda. Como se investigó, la carga de páginas en frames y la inclusión de referencias a las mismas en las tablas de páginas de los procesos son tareas llevadas a cabo durante el manejo de excepciones de fallo de página. Esto es debido a que Linux utiliza la estrategia de paginación bajo demanda, la cual implica que el mapeo a memoria física de las páginas se delega hasta que

alguno de los procesos que las comparten acceda a las mismas. El punto de entrada del mecanismo es la rutina de atención de fallo de página de cada arquitectura. Sin embargo, dentro de cada uno la rutina que se invoca para manejar el fallo cuando se detecta que fue debido a que la página no residía en memoria física (podría ser por otras razones, como falta de permisos en el acceso) es la misma para todas las arquitecturas. Esta rutina es `handle_mm_fault()` quien a su vez invoca a `handle_pte_fault()`, ambas definidas en `{KERNEL}/mm/memory.c`. Dentro de `handle_pte_fault()` se hace el intento de mapear la página mediante una función auxiliar distinta dependiendo del tipo de uso que se le da a la página en cuestión y si la misma fue o no accedida desde la creación del proceso:

- Si la página nunca fue cargada y no mapea un archivo en disco se invoca a `do_anonymous_page()` donde en caso de éxito se crea un nuevo mapeo a un frame. Tenemos aquí un nuevo punto donde se originan eventos de la clase buscada.
- Si la página pertenece al mapeo lineal de un archivo en disco se invoca a `do_linear_fault()` y si el mapeo es no lineal se llama a `do_nonlinear_fault()`. En ambos casos el nuevo frame es asignado por `__do_fault()`, donde se identificaron más eventos de creación de mapeos.
- Si la página deseada está respaldada en un área de swap, se invoca a `do_swap_page()`, donde también en caso de éxito se crea un nuevo mapeo y por tanto se originan eventos para el prototipo.

Si no se entra en ninguno de estos casos y fue un acceso de escritura a una página protegida contra la misma en las tablas de páginas, se invoca `do_wp_page()` que es parte de la implementación del mecanismo de copy-on-write visto. Se discute este caso y otros relacionados a continuación. Cabe aclarar que todas las primitivas aquí mencionadas se encuentran definidas bajo `{KERNEL}/mm/memory.c`.

Copy-on-write. Como se explicó en la sección 2.4, en Linux los procesos heredan el espacio de memoria de sus padres utilizando una técnica que se conoce como copy-on-write. Este mecanismo permite a padre e hijo compartir los frames mapeados a sus espacios de memoria mientras sean accedidos exclusivamente para lectura. De esta forma se difiere la duplicación de la página en un nuevo frame al momento de la primera escritura, evitando copias posiblemente innecesarias.

La tarea de creación del nuevo espacio de memoria la lleva a cabo la función `copy_mm()` definida en `{KERNEL}/kernel/fork.c`. Para el caso de que el proceso no sea un “proceso liviano” `copy_mm()` crea e inicializa un nuevo espacio de memoria. Luego de esto se invoca la función `dup_mmap()` que se encarga de duplicar las tablas de páginas y áreas de memoria del espacio del padre. Para la duplicación de las tablas de páginas se invoca a la función `copy_page_range()` definida en `{KERNEL}/mm/memory.c`. Se revisó la implementación de `copy_page_range()` y se encontró que de forma similar a lo visto con las `zap_XXX_range()`, se utiliza una cadena de funciones auxiliares para cubrir los distintos niveles de tablas de páginas. En particular el último nivel, el de las propias entradas de tabla de páginas, lo procesa la función `copy_pte_range()` definida en el mismo fuente. Esta última realiza sucesivas invocaciones a `copy_one_pte()` la cual realiza los mapeos de los frames heredados al nuevo espacio de memoria. Se identificó entonces, que por cada mapeo exitoso se produce un evento de modificación en los mapeos.

Como se dijo antes, la duplicación de la página a un nuevo frame se difiere al primer acceso de escritura y como se mencionó en la sección anterior el manejador

de fallo de páginas encarga esta tarea a la función `do_wp_page()`, de forma independiente a la arquitectura. De esta forma, se tiene allí un nuevo origen posible de eventos.

Mapeos no lineales de archivos. Además del método normal de mapear un archivo a memoria de forma lineal utilizando la system call `mmap()` (ver [**12**]), Linux provee una forma no lineal de mapear un archivo a el espacio de memoria de un proceso. Básicamente, un mapeo no lineal es como uno lineal, pero las páginas virtuales consecutivas de memoria no están mapeadas a páginas secuenciales del archivo, sino que aleatorias o arbitrarias. Para crear un mapeo de este tipo la aplicación de modo usuario primero crea un mapeo lineal compartido con la llamada al sistema `mmap()` y luego vuelve a mapear algunas de las páginas en el área de memoria creada para el mapeo anterior por medio de la llamada al sistema `remap_file_pages()`. Esta crea el nuevo mapeo e inicializa las flags de las entradas de tabla de páginas de forma que el mecanismo de paginación bajo demanda descrito reconozca que se trata del caso de mapeos no lineales de archivo. El manejo de las tablas de páginas lo hace por medio de `populate_range()`, quien a su vez utiliza la función auxiliar `install_file_pte()`. Esta última antes de insertar la entrada en la tabla de páginas, verifica que no existan mapeos previos a esa dirección virtual. Si existe alguno, simplemente lo elimina mediante la función `zap_pte()`. Dado que `zap_pte()` es utilizada al menos en este caso para eliminar posibles mapeos existentes a frames, se da aquí un nuevo punto de origen de eventos de la clase buscada.

Liberación de frames. Como se investigó en la sección 2.4, Linux maneja un algoritmo de selección de frames candidatos a ser reclamados. Este se basa en una serie de reglas para seleccionar el orden más conveniente de liberación de frames asignados a diferentes recursos del sistema o procesos de usuario. Sin entrar en detalles sobre dichas reglas, basta saber que todos los puntos de acción del mismo dan como salida una lista de frames candidatas a ser reclamadas y el intento de liberación efectiva de dichos frames está centralizado en un único lugar. La función `shrink_page_list()` definida en `{KERNEL}/mm/vmscan.c` recibe una lista de frames candidatos y realiza intentos de liberación de los mismos. En caso de que el frame este mapeado a uno o más espacios de memoria de procesos, se debe intentar eliminar dicho mapeo, tarea que es llevada a cabo por la función `try_to_unmap()` definida en `{KERNEL}/mm/rmap.c`. Esta última delega su trabajo dependiendo si el frame corresponde a un mapeo anónimo o de archivo. Para esto cuenta con dos funciones auxiliares `try_to_unmap_anon()` y `try_to_unmap_file()`. Estas se valen del mecanismo de "reverse mapping" (ver capítulo de 17 de [**12**]) implementado en el MM, el cual permite obtener para cada frame mapeado, las VMAs a las cuales está asociado y es a partir de las mismas que se logra desmapear el frame de los espacios de memoria de procesos. La función `try_to_unmap_one()` se encarga de eliminar el mapeo entre un frame y una VMA y es llamada repetidamente desde `try_to_unmap_anon()` y `try_to_unmap_file()`. Investigando la lógica de la misma se identificaron dos casos en los cuales se desmapea el frame de un proceso y por tanto se está ante nuevos eventos, esta vez, de eliminación de mapeos.

Los casos anteriores sólo cubren puntos en los que se trata con mapeos anónimos o de archivos mapeados linealmente, en los cuales se desmapea el frame de los espacios de memoria mediante el mecanismo de "object reverse mapping" (ver capítulo de 17 de [**12**]) que permite llegar a las VMAs donde está mapeado. Sin embargo, el object reverse mapping usado en estos casos, no funciona con VMAs

que realizan mapeos no lineales de archivo. En este último caso es necesario realizar una búsqueda lineal entre todas las VMAs que mapean el archivo para intentar desmapear el frame. Sin entrar en detalles de cómo funciona dicha búsqueda, la misma es implementada en `try_to_unmap_file()` y basta mencionar que ésta utiliza la rutina `try_to_unmap_cluster()` que es quien desmapea efectivamente el frame de los espacios de memoria de procesos. Es por esto que aquí también se producen eventos análogos a los anteriores.

Áreas de Swap. Linux provee las llamadas al sistema `swapon()` y `swapoff()` para activar y desactivar un área de swap respectivamente. En el caso de `swapoff()` al desactivar un área de swap es posible que la misma todavía contenga respaldos de páginas de procesos, en cuyo caso se debe hacer un "swap-in", es decir, devolver las mismas a la memoria física. De esta forma, se identificó otro lugar además del manejador de fallo de páginas, donde es necesario cargar una página en un frame y actualizar las tablas de páginas de los procesos que la usan. La llamada `swapoff()` esta implementada en `{KERNEL}/mm/swapfile.c` y según lo explicado en la sección 17.4 de [**12**] el trabajo de hacer el swap-in de las páginas existentes en el área lo lleva a cabo la función `try_to_unuse()`, definida en el mismo fuente. Investigando `try_to_unuse()` se entendió que para cada entrada del área a procesar se recorrían todos los espacios de memoria de procesos e hilos invocando para cada uno la rutina `unuse_mm()`. Eventualmente a partir de esta última se desencadenan invocaciones a la rutina auxiliar `unuse_pte()` que es donde efectivamente se actualizan las entradas de tabla de páginas del proceso, provocando un nuevo evento de creación de mapeos de frames al espacio de memoria de un proceso.

Execute in place (XIP). Execute in place es una técnica que permite la ejecución de programas desde almacenamiento secundario sin necesidad de copiarlos a la RAM [**16**]. Linux provee una implementación de dicha técnica en `{KERNEL}/mm/filemap_xip.c`, por medio de un conjunto de operaciones de dispositivo por bloques. La implementación de XIP utiliza su propio frame fijo relleno con ceros (recordar lo visto en la sección 2.4) vacía para evitar colisionar con otros usuarios del usado por el MM (ZERO_PAGE). Este frame se define en la variable global `__xip_sparse_page` y se cuenta con una función auxiliar `__xip_unmap()` que se encarga de desmapear el mismo de un espacio de memoria, la cual es utilizada por varias de las operaciones de la implementación de XIP. Por cada entrada de tabla de páginas donde se encuentra y desmapea la `__xip_sparse_page` se produce un evento de desmapeo entre un frame y un proceso del espacio de usuarios.

Cambios en los permisos. Hasta ahora vimos como mediante los puntos de actualización del valor *rss* se identificó el origen de todos los eventos dentro del kernel donde se crean y eliminan mapeos entre frames y procesos del espacio de usuarios. Como se mencionó, se contempló la posibilidad de que hubiera cambios en los permisos de un mapeo existente durante su ciclo de vida. Investigando el kernel se encontró que esto era posible, debido a que existe una llamada al sistema o syscall que permite modificar los permisos de un área de memoria virtual La syscall `sys_mprotect()` definida en `{KERNEL}/mm/mprotect.c`, modifica los permisos de un segmento de memoria virtual de un proceso, descrito por la dirección virtual de comienzo y largo del mismo. El procedimiento de la misma básicamente recorre todas las áreas virtuales cuya región de memoria esté contenida dentro del segmento objetivo y modifica los permisos de las mismas por medio de la función

`mprotect_fixup()`. Esta función se encarga de hacer las modificaciones necesarias en los permisos de la VMA(representados por flags), pudiendo además dividir la misma o combinar varias de ellas. En caso de éxito, se está ante eventos de modificación de los permisos de un área virtual y por tanto de los mapeos entre un proceso y los frames asignados a la misma. Dicho cambio no necesariamente modifica el estado de los frames afectados, pero se debe manejar el evento para considerar la posibilidad y tomar acciones.

3.7.2.2. Manejo de los eventos. Se pueden resumir los eventos identificados en tres categorías:

- Eventos en los que se crea un nuevo mapeo entre un frame y la memoria virtual de un proceso de usuario
- Eventos en los que se elimina un mapeo de este tipo
- Eventos en los que se modifican los permisos con los cuales fue creado un mapeo preexistente de la clase vista

Para la determinación y actualización del estado definido para un frame en la sección 3.7.2, fue necesario crear dos manejadores de eventos, los cuales forman parte de la interfaz del módulo con el kernel. El primero de ellos es `pgsa_page_process_mappings_change()` y se encarga de manejar las dos primeras clases de eventos vistos, es decir, todos los detectados menos el correspondiente a las llamada al sistema `sys_mprotect()`. El restante tipo de evento es manejado por `pgsa_vma_process_flags_change()`.

`pgsa_page_process_mappings_change()`. Se muestra a continuación la firma completa del método:

```
void pgsa_page_process_mappings_change(struct page * page,
                                       boolean writer,
                                       boolean unmap)
```

El parámetro *page* es una referencia al descriptor del frame sobre el cual se produce el evento. El parámetro *unmap* indica si el evento corresponde a la creación (false) o eliminación (true) de un mapeo. Finalmente, el parámetro *writer*, es un booleano que vale true en caso de que los permisos del mapeo permitan la escritura sobre el frame.

En todos los puntos donde se originan los eventos manejados por este handler, se conoce exactamente el área de memoria virtual y/o la entrada de tabla de páginas, además del frame que participa del mapeo que se acaba de crear o eliminar. De esta forma, se cuenta con los elementos necesarios para indicar al manejador, los parámetros que describen el evento. En particular, el parámetro *writer* se determina a partir de las flags del descriptor del área de memoria, las cuales indican los permisos con que el proceso puede acceder a las páginas contenidas en la misma.

La lógica del handler es bastante simple y consta de modificar los contadores definidos, incrementando y decrementando el contador de mapeos de acuerdo a si se agrega o quita un mapeo y haciendo lo propio con el contador de escritores en caso de que se trate de un mapeo de escritura. Luego de esto se determina si el frame pertenece al conjunto de sólo lectura y se compara dicho estado contra el anterior al evento. Si el frame estaba fuera del conjunto y ahora pertenece al mismo o vice versa, se toman las acciones correspondientes para permitir el correcto funcionamiento de los mecanismos de detección y corrección de errores como se verá más adelante.

`pgsa_vma_process_flags_change()`. Nuevamente se comienza mostrando la firma completa del manejador:

```
void pgsa_vma_process_flags_change(struct vm_área_struct *vma,
                                   unsigned long oldFlags)
```

Aquí el parámetro *vma* es la región virtual de memoria cuyos permisos cambiaron y oldFlags contiene el valor de las flags de la misma (recordemos que son estas las que indican los permisos de acceso de las páginas de la VMA) previo al cambio. La operación debe ser invocada una vez cambiados los permisos, para que la *vma* contenga las nuevas flags. Lo primero que se hace es comparar las flags previas con las contenidas en el campo `vm_flags` de vma. Si no hubo cambios en el bit *VM_ WRITE,* el cual determina los permisos de escritura, no se hace nada. Si hubo cambios en el mismo, se deben recorrer todas las páginas virtuales contenidas dentro de la región de memoria virtual delimitada por la VMA y para las que están mapeadas a un frame de memoria física, obtener la estructura *page* que lo describe y realizar las modificaciones correspondientes a los campos definidos para el prototipo. A continuación veremos en más detalle cómo se resolvió esto.

Como se vio en la sección 2.4 las VMAs tienen un campo `vm_start` y uno `vm_end` con las direcciones virtuales del comienzo y fin de la región de memoria virtual que describen. Si a esto se suma que el tamaño de la página es conocido y se accede en el kernel como la constante *PAGE_ SIZE*, es posible iterar sobre las páginas virtuales incrementando `vm_start` con *PAGE_ SIZE* hasta alcanzar `vm_end`. Para cada página virtual se necesita verificar si está mapeada a un frame de memoria y si lo está obtener el descriptor del mismo. Para esto se reutilizó el mecanismo que usa `get_user_pages()` función que mapea a frames físicos y bloquea en memoria todas las páginas dentro de un rango deseado de un espacio de memoria de usuario y permite obtener todas las estructuras *page* a las cuales se mapeo el rango, cargando tablas de páginas a demanda, verificando la presencia de la página y otras tareas. Este mecanismo se basa fundamentalmente en la utilización de la función `follow_page()` la cual dada una dirección virtual y una VMA retorna la *page* en la cual esta mapeada dirección o null cuando no existen mapeos. Luego, para cada estructura *page* encontrada se debió actualizar el estado, lo cual se simplificó representando el cambio de permisos como la ocurrencia de dos eventos de cambio en los mappings manejados por `pgsa_page_process_mappings_change()`. El primero es el evento de eliminación del mapeo previo al cambio (con los permisos de *oldFlags*) y el segundo es la creación de un nuevo mapeo con los permisos actualizados, como se muestra a continuación:

```
pgsa_page_process_mappings_change(page,
                                  oldFlags & VM_WRITE,
                                  1 /*unmap*/);

pgsa_page_process_mappings_change(page,
                                  vma->vm_flags & VM_WRITE,
                                  0 /*map*/);
```

3.7.3. Procesos Registrados.

Un concepto importante que se maneja es el de registro de procesos. Se llama procesos registrados a aquellos que notificaron explícitamente al módulo de PGSA, que desean ser incluidos como clientes de las funcionalidades que este brinda. La interfaz mediante la cual los mismos realizan alta y baja de su registro se discutirá en la sección 3.7.10, mientras que aquí se hará hincapié en el manejo interno de los procesos registrados en el módulo y las prestaciones que se le brindan a los mismos.

El concepto surge a partir de la decisión de agregar notificaciones de los errores detectados a los procesos de usuario afectados. Para poder brindar un mecanismo de notificaciones sincrónicas las cuales el proceso estuviera preparado para recibir, se vio la necesidad de poder restringir dichas notificaciones a un grupo de procesos seleccionados. Dado que algunos de los mecanismos de búsqueda de errores o bien provocan una disminución de la performance en los procesos verificados o bien aumentan su efectividad al restringir su acción a un grupo selecto de procesos, también se encontró otras aplicaciones útiles para el registro de procesos.

Investigando el manejo de procesos de Linux en el capítulo 3 de [**12**], se encontró que se soporta el concepto de procesos multihilos, donde los hilos de un mismo proceso comparten recursos pero cada uno puede ser tratado por el planificador de tareas para ejecutar como un proceso independiente. A su vez, se tiene el concepto de grupo de tareas, el cual permite agrupar todos los hilos que son subprocesos de un proceso original, el primer hilo del grupo. Cada hilo cuenta con un identificador único llamado pid, nombre estándar en el área de sistemas operativos para el identificador de proceso, y el pid del proceso original se utiliza como identificador del grupo, denominado tgid. Linux utiliza como descriptor de procesos (process control block en teoría de S.O.) la estructura *task_struct*, definida bajo `{KERNEL}/include/linux/sched.h`. Cada hilo tiene asociada su propia *task_struct* la cual indica entre otros datos el valor de su pid y su tgid. En este contexto, se decidió que el registro de procesos debía realizarse a nivel de grupo de tareas y no de hilo individual, dado que la mayoría de las aplicaciones de usuario de mediano o incluso pequeño porte suelen utilizar más de un hilo y las mismas deberían poder registrarse por medio de un único paso y no necesitar de tantos como hilos con los que cuenta la aplicación.

Para mantener los procesos registrados se cuenta con una colección de estructuras *task* global al módulo. Dado que por medio del descriptor de un hilo se conoce su identificador de grupo de tareas, basta con almacenar el descriptor del hilo para el cual se hace el registro, que puede ser cualquiera de los que componen un proceso de forma indistinta. De esta forma, todos los miembros del mismo se interpretan como registrados por medio de la referencia a un único hilo del grupo. Además de la tarea que identifica el grupo cuyas páginas serán monitoreadas, se guarda una segunda tarea a la cual se denomina agente y es la que recibirá por ejemplo las notificaciones por parte del módulo. El agente puede ser un proceso externo al grupo, por ejemplo, si se desea monitorear una aplicación existente por medio de un agente externo. También es posible que ambas tareas coincidan, en caso que se desee dedicar un hilo de la propia aplicación como agente para monitorear la misma. Para la modificación y consulta de esta colección se definieron en el módulo las siguientes operaciones:

- `pgsa_register_task(task, agent)`: agrega un nuevo elemento a la colección para el grupo y agente indicados.
- `pgsa_unregister_task(task)`: remueve el descriptor de proceso indicado de la colección, si es que éste pertenece a la misma. En caso contrario no tiene ningún efecto.
- `boolean pgsa_is_registered_task(task)`: retorna true si el descriptor de algún proceso del grupo de tareas identificado por el tgid del task_struct indicado, pertenece a la colección.

Las dos primeras operaciones son utilizadas por la interfaz con espacio de usuario para brindar mecanismos que permitan a los procesos ser registrados, ver sección 3.7.10. El procedimiento `pgsa_unregister_task()` debe ser además invocado en el manejador de terminación del proceso (función *exit()*) como técnica defensiva para evitar que figuren como registrados procesos que ya terminaron su ejecución. Esto podría traer además de errores lógicos en el funcionamiento del módulo, errores de ejecución graves en el kernel. La tercer operación definida es utilizada desde la lógica de las distintas funcionalidades del módulo para definir comportamientos específicos para esta clase de procesos, de acuerdo a los parámetros de configuración del módulo. Veremos más sobre esto en las próximas subsecciones.

Fue necesario definir en la interfaz del módulo con el resto del kernel, un procedimiento `pgsa_process_task_exit()` para manejar el evento de fin de un hilo. El mismo básicamente remueve el proceso del registro mediante `pgsa_unregister_task()` como se dijo. Se investigó el manejo de procesos en el kernel y se encontró que cuando un proceso termina su ejecución, uno de los primeros pasos es ejecutar la función `do_exit()` definida en `{KERNEL}/kernel/exit.c`, donde se maneja la terminación para los diferentes subsistemas del kernel. En el caso del MM, el trabajo se delega a la función `exit_mm()` y fue allí donde se entendió más coherente invocar el manejador definido. La firma del manejador es muy intuitiva, ya que recibe un único parámetro indicando la tarea que está terminando por medio de su descriptor *task_struct*.

```
void pgsa_process_task_exit(struct task_struct * task)
```

3.7.4. Banderas de Modo.

Las banderas definidas tienen dos valores posibles, pueden estar activadas o desactivadas y su significado puede ser la habilitación o no de una funcionalidad, o la elección entre dos estrategias. Las funcionalidades del prototipo se pueden agrupar en dos categorías bien marcadas y por tanto lo mismo sucede con las banderas definidas. Por un lado están las propiedades que configuran el mecanismo de búsqueda de errores, es decir aquel que decide cuando se verifican las redundancias mantenidas para los frames de sólo lectura para detectar cambios. Las banderas que pertenecen a este grupo son las siguientes:

- *PGSA_CHECK_DAEMON*: si está activada se utiliza el mecanismo de búsqueda de errores que consta de un demonio a nivel de kernel que está continuamente iterando sobre el conjunto de páginas a verificar, realizando justamente esa tarea sobre las mismas, e informando los resultados al final de cada iteración. En caso de que la bandera esté desactivada no se habilita este mecanismo.
- *PGSA_CHECK_CURRENT*: si está activada se utiliza el mecanismo de búsqueda de errores que verifica las páginas de sólo lectura que están mapeadas al proceso seleccionado por el planificador de Linux para ser el próximo a obtener el procesador, justo antes de que éste obtenga dicho recurso. En caso de que la bandera esté desactivada no se habilita el mecanismo.
- *PGSA_CHECK_ALL*: si está desactivada los mecanismos de búsqueda de errores existentes se aplican sólo a las páginas mapeadas a procesos registrados y si está activada se aplican a todo el sistema. En particular para el caso del demonio éste itera sobre todos los frames utilizables del sistema si está activada o se itera sobre los espacios de memoria de los procesos registrados en caso negativo. Para el segundo mecanismo mencionado, en

caso de que esté desactivada se verifican las páginas de sólo lectura del próximo proceso a ejecutar sólo si éste se encuentra registrado. Si está activada no se hace este filtro entre los procesos, aunque cabe destacar que en la práctica esta opción ha probado tener un efecto inaceptable en el rendimiento del sistema.

La otra categoría de banderas de modo definida, está conformada por las propiedades que configuran las acciones a tomar en caso de detectar un error, mediante los mecanismos de búsqueda habilitados. Todas estas acciones corresponden a funcionalidades nuevas desarrolladas completamente para esta etapa del prototipo. Se detalla a continuación las mismas:

- *PGSA_CORRECTION_CODE*: si está activada se habilita el mantenimiento a nivel de páginas de redundancia que permite implementar un código corrector. Para este caso cuando alguno de los mecanismos de búsqueda encuentra un error, se utiliza el código para intentar corregirlo. En caso de no estar activada no se mantiene la redundancia ni se intenta la corrección. Se verá el uso del código corrector en detalle en la sección 3.7.7.
- *PGSA_FILE_REJUVENATION*: si está activada / desactivada se habilita / deshabilita respectivamente la acción de rejuvenecimiento automático de páginas mapeadas a archivos, en caso de encontrar error en una mediante alguno de los mecanismos de búsqueda. Por más detalles ver la sección 3.7.8
- *PGSA_NOTIFICATIONS*: si está activada / desactivada se habilita / deshabilita respectivamente, la acción de publicación de detalles de los errores detectados al espacio de usuarios, ver sección 3.7.9.
- *PGSA_SIGNAL*: si está activada / desactivada se habilita / deshabilita respectivamente la acción de envío de notificaciones a el/los procesos que mapean una página sobre la cual se detectó un error mediante alguno de los mecanismos de búsqueda. Dichas notificaciones se envían sólo a los agentes de procesos que estén registrados en el módulo, por más detalles referirse a la sección 3.7.9.

Además de las categorías mencionadas, se definió la flag *PGSA_ON* de uso general en todo el módulo, la cual permite habilitar y deshabilitar todas las funcionalidades incluidas en el mismo. De esta forma cuando la flag está desactivada, el kernel ejecuta igual que si no estuviera instalado el módulo.

La definición de este conjunto de propiedades está pensada para que el módulo pueda ser configurado en caliente desde el modo usuario por un administrador del sistema. En la sección 3.7.10 se verá como se logró hacer esto posible a nivel de interfaz con espacio de usuario.

3.7.5. Método de detección de cambios. Una vez determinado claramente el conjunto de frames que cumplen las condiciones de nuestras hipótesis, se debe resolver el próximo problema planteado, la detección de cambios. La solución general consta de mantener alguna clase de redundancia sobre los bytes de la página contenida para cada frame del conjunto de trabajo, que garantice detección de errores en varios bits y sea eficiente para el volumen de datos requerido (cada frame almacena una página de 4kb). Dicha redundancia debe ser actualizada en los eventos de cambio de estado mencionados anteriormente, de forma que sea la correspondiente a la página contenida en el frame siempre que el frame pertenezca

al subconjunto de sólo lectura. La estrategia es recalcular y almacenar el código cada vez que el frame pasa a un estado dentro del conjunto y eliminarla (o ignorarla) cuando vuelve a uno fuera del mismo. Luego, es simple determinar si hubo cambios en la página contenida dentro de un frame de sólo lectura, pues basta solo con repetir el cálculo del código y compararlo con el almacenado. Se verá ahora como se seleccionó el mecanismo de detección de errores a utilizar.

El mantenimiento de la integridad de los datos es una materia muy estudiada. A la hora de buscar un método simple y efectivo de detección de cambios surgen como referencia métodos como paridad, checksum (o suma de comprobación) y CRC (Cyclic Redundancy Check), etc. Lo importante en este caso es seleccionar el que mejor se adapte a nuestros requerimientos. Dado que el mismo se utilizará en eventos del manejo de bajo nivel de la memoria de un sistema operativo, en contextos en los cuales no se admiten demoras mayores al grado de milisegundos o incluso menores, el método debe ser eficiente para calcular y verificar el código de redundancia con respecto a estos tiempos. Además, considerando que se maneja como bloque de datos las páginas de memoria las cuales tienen una dimensión mínima de 4Kb y el sistema cuenta normalmente con cientos de miles de estas, se tienen limitaciones también en las dimensiones de la redundancia utilizada. Para establecer un criterio se decidió acotar las mismas a un máximo de 64 bits por página. Dado que el prototipo está implementado en lenguaje C sobre el Linux kernel otra característica deseable del método seleccionado es que sea fácil de implementar o reutilizar en este contexto. Para terminar, como contra parte a estas restricciones, se debe buscar un método que las cumpla con el mayor poder de detección posible para la dimensión y características de los datos objetivo.

Se decidió utilizar un método de CRC ya que los mismos permiten detectar errores en grandes espacios de datos y se calculan de forma eficiente. El CRC calculado a un conjunto de datos permite controlar si los datos fueron modificados si los errores se generan en ráfagas contiguas. En el caso de este prototipo los errores ocurren de a bits en el espacio de memoria, por lo que una cobertura de ráfaga permite detectarlos. Además, el resultado del CRC de un stream de datos está acotado a un número fijo de bits, por lo que se puede seleccionar que cantidad de bits quiero tener como máximo de detección y para almacenar y entonces se puede seleccionar un algoritmo de CRC compatible con estas características.

En particular se decidió utilizar CRC 32 ya que sólo se necesita almacenar un valor de 32 bits para su resultado y el mismo es un algoritmo eficiente y probado. Se decidió utilizar un algoritmo que tenga una cobertura mucho mayor a la necesaria de forma de tener asegurada la cobertura en caso que el error no sea simplemente generado por partículas de alta energía, sino también por defectos en la memoria misma que provocaría errores más grandes a un bit en una región conjunta, o sea una ráfaga, también de esta forma se pueden detectar pequeños buffer overflow o underflow que modifiquen hasta 4 Bytes del espacio de código.

Una vez seleccionado el método de detección, se investigó la posibilidad de reutilización dentro del kernel para simplificar esta tarea. El Linux kernel cuenta desde su versión 2.4 con un framework de criptografía llamado Crypto API. Como se explica en el archivo `api-intro.txt` parte de la documentación de kernel bajo `{KERNEL}/Documentation/crypto`, la API está diseñada para recibir vectores de páginas y trabajar sobre éstas directamente. Sin embargo, allí mismo se encontró una restricción sobre los posibles contextos de aplicación de la API. La misma sólo

puede ser utilizada en contexto de usuario o softirq (mecanismo para diferir trabajo de interrupciones, ver [**12**] sección 2.7). Dado que buena parte del trabajo del prototipo se llevará a cabo en contexto de interrupciones, es un obstáculo importante. Conociendo esto se intentó comprender dichas restricciones para considerar la posibilidad de reutilización de la algoritmia implementada en la API, sin pasar por las interfaces del framework. Para esto se contactó a uno de los creadores de la API Herbert Xu, quien confirmó la restricción, explicó que se debía a una clase de mapeos de memoria utilizados y comentó que estaba desarrollando una interfaz para la API que no tenía dicha restricción. Debido a esta información se decidió implementar o reutilizar de una biblioteca externa un CRC32, dejando el mismo encapsulado de forma que fuera fácil de sustituir.

Como CRC32 se utiliza para la detección de errores en varios componentes de Linux, como ser el stack Ethernet, en algunos de los filesystems, en el stack Bluetooth, etc. se resolvió buscar en el kernel como había sido implementado en lugar de buscar una librería externa al mismo. Luego de realizada esta búsqueda se descubrió que todos los distintos cálculos de CRC han sido encapsulados en una serie de librerías según el polinomio que se utiliza para el mismo, en particular se utilizó `linux/crc32.h`, que es donde se definen las operaciones para el cálculo del CRC32.

3.7.6. Búsqueda de errores. Ya se definió una solución que permite mantener el estado de cada frame y detectar posibles errores de bits en el mismo, sin embargo, todavía resta ver cuándo, dónde y con qué estrategia se realizarán las verificaciones. Se consideraron distintas soluciones para el chequeo del conjunto de frames de sólo lectura. Antes que nada se debe tener en cuenta que se está realizando detección por software de errores que ocurren en el hardware sin ninguna asistencia del mismo. Esto implica que la detección será asincrónica al error y sin importar la estrategia seleccionada existirá un intervalo variable de tiempo entre ambos que llamaremos RD (retraso de detección). Dado que si bien la planificación de la ejecución de los procesos es una tarea del sistema operativo, la ejecución efectiva de los mismos cuando tienen el procesador asignado la lleva a cabo el hardware, tampoco es posible realizar verificación por software a demanda de las secciones de memoria accedidas en el código del proceso, de forma de garantizar que los errores sean detectados de forma previa a un posible acceso sobre el área afectada. Esto implica que la probabilidad de que un proceso acceda memoria adulterada por un error de hardware depende directamente de ese intervalo variable entre un error y su detección. De esta forma el objetivo es buscar la estrategia que minimice el RD. El enfoque más simple en este caso es realizar las verificaciones en un ciclo con la mayor frecuencia posible acotando así el RD a la duración de cada iteración del ciclo. Dado que no podemos asumir nada sobre donde ocurrirá el posible error, se debe recorrer todo el conjunto de forma ordenada en cada iteración. Para lograr esto, se decidió contar con un demonio a nivel de kernel dedicado exclusivamente a esta tarea. Dado que ese demonio es un proceso limitado por CPU y su ejecución puede reducir la performance general del sistema, se debió tomar precauciones para que tuviera la menor prioridad posible sobre los recursos de procesamiento, de acuerdo a las estrategias del planificador de procesos de Linux, ver [**12**]. De esta forma, el demonio definido itera por número de frame sobre todos los frames de la memoria física del sistema que se definen como utilizables para los procesos de usuario y realiza chequeos sobre la redundancia calculada para aquellos que se

identifica que pertenecen al conjunto de sólo lectura. Todo el trabajo de detección y manejo de errores para un frame se encapsuló en una invocación a al procedimiento `pgsa_page_check()`. El mismo es utilizado en todos los métodos de búsqueda como muestra la figura 11.

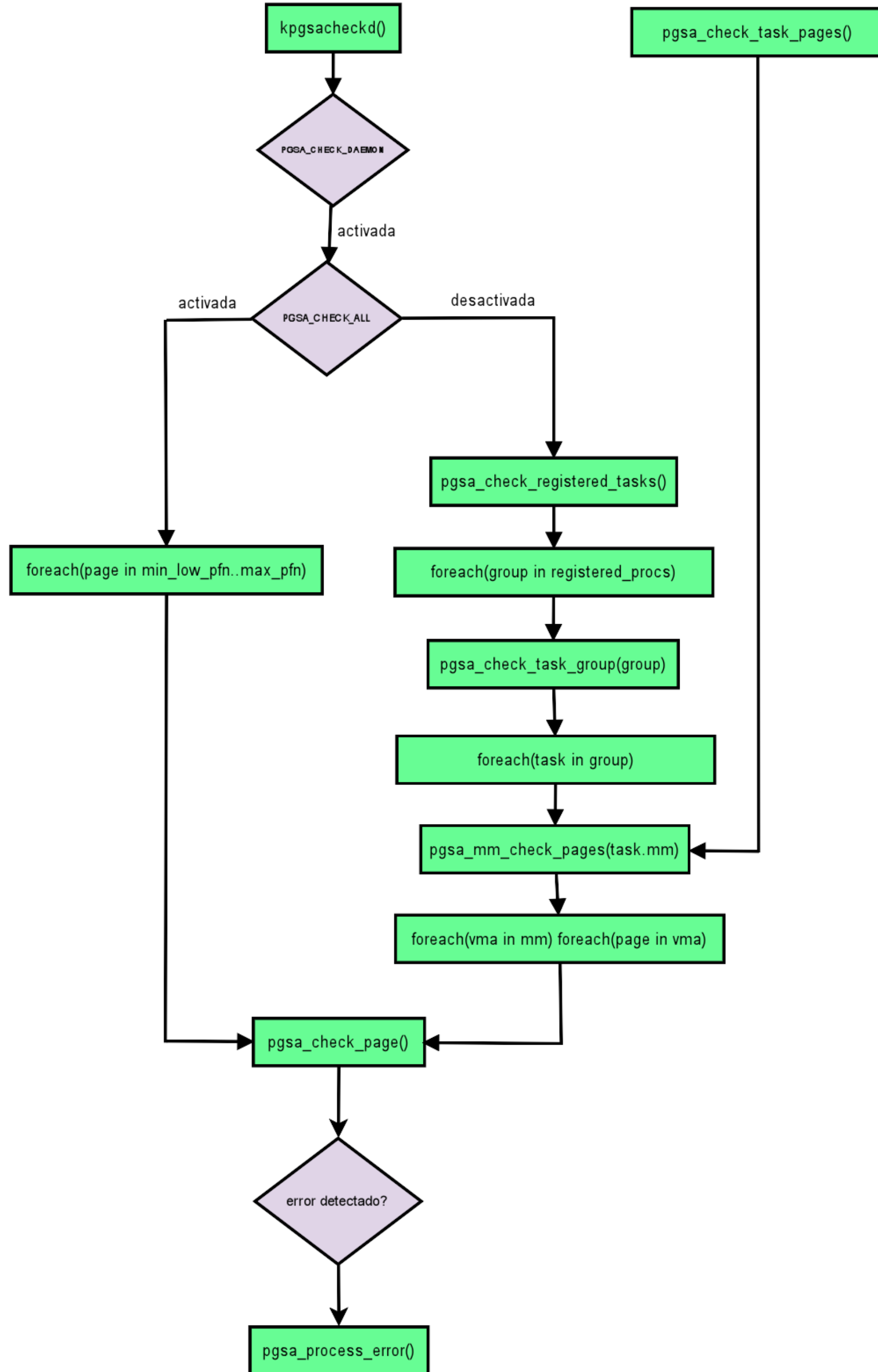


FIGURA 11. Diseño de métodos de búsqueda

Gracias a la definición del registro de procesos, fue posible agregar una estrategia alternativa para la búsqueda de errores en el demonio. La misma se basa en chequear únicamente los frames de sólo lectura que se encuentran mapeados a procesos registrados y como se mencionó en la sección 3.7.4, se habilita desactivando la bandera *PGSA_CHECK_ALL*. La gran ventaja de esta estrategia es que el usuario puede acotar el conjunto de frames a chequear a los utilizados por un conjunto de procesos que considera críticos. De esta forma el conjunto de frames chequeados será probablemente mucho menor al conjunto completo, lo cual permitirá mayor frecuencia de chequeo para cada frame dentro del mismo y reduciría el RD promedio. Otra ventaja importante es que al reducir la cantidad de frames, se reduciría el ruido que provoca el demonio en el cache de la memoria con el que cuenta el hardware.

Cuando está habilitada esta estrategia el demonio invoca el procedimiento `pgsa_check_registered_tasks()` que encapsula la lógica de la misma. Este itera sobre los elementos de la colección de procesos registrados invocando para cada elemento el procedimiento `pgsa_check_task_group()` . El mismo identifica para el descriptor recibido el grupo de tareas al que pertenece e itera sobre todos los hilos que componen el mismo invocando la función `pgsa_mm_check_pages()` para el espacio de memoria de cada hilo. Esta función se encarga de recorrer todos frames de memoria que tienen mapeada alguna de las páginas virtuales de dicho espacio de memoria y se realiza el chequeo de cada uno de ellos por medio del procedimiento `pgsa_page_check()`utilizado para la otra estrategia del demonio, de forma que el tratamiento de los frames siga encapsulado en un único lugar y sea independiente de la estrategia de búsqueda utilizada. Un problema que surge con la iteración realizada, es que a diferencia de la original utilizada por el demonio, en este caso es posible y muy probable que se pase por el mismo frame más de una vez. Para solucionar esto, se mantiene una colección durante toda la iteración del demonio, la cual contiene los pfn (page frame number) de los frames ya chequeados. La misma comienza vacía y en `pgsa_mm_check_pages()` para cada frame se consulta si su pfn pertenece a esta colección, saltándolo en caso positivo y agregándolo luego de ser chequeado en caso negativo.

Se cuenta además, con otro mecanismo para la búsqueda de errores en frames. El mismo implica realizar el chequeo de los frames mapeados al próximo proceso/hilo a obtener el procesador justo antes de que obtenga dicho recurso. El fuerte de este mecanismo es que se busca errores en el último momento antes de que los ocurridos puedan tener algún efecto en la ejecución de un proceso, aumentando las probabilidades de evitar esto en la mayoría de los casos. Al igual que el chequeo realizado por el demonio, éste es asincrónico a la ocurrencia de un error y por tanto no garantiza evitar el acceso a memoria corrupta en todas las oportunidades. Un ejemplo simple donde falla sería la ocurrencia de un error mientras el proceso se encuentra ejecutando. Se decidió implementar el mecanismo debido a las características de la planificación de procesos en Linux [**12**]. Linux utiliza la técnica de tiempo compartido, la cual divide el tiempo del recurso CPU en intervalos y asigna uno para cada proceso listo para ejecutar. A cada momento hay un sólo proceso ejecutando, y puede hacerlo sólo por un intervalo que ronda en promedio valores del orden de los 100 milisegundos y luego se planificará nuevamente. Con una carga de varios procesos compartiendo el procesador, un proceso dado está ejecutando sólo un pequeño porcentaje del tiempo. Si se suma a esto que los procesos no sólo

utilizan el procesador, sino que también deben esperar para realizar operaciones de E/S, el porcentaje del tiempo que se encuentran en ejecución es aún menor. Debido a esto, si asumimos que las probabilidades de ocurrencia de un error son uniformes en el tiempo, vemos que la probabilidad de ocurrencia de uno no detectable, es decir, mientras el proceso ejecuta es muy pequeña. Dado que si el error ocurre en cualquier otro momento, será detectado por este mecanismo antes de que el frame con error sea utilizado por algún proceso, esta verificación promete ser muy efectiva.

Para implementar el mecanismo se debió investigar el funcionamiento interno del scheduler, buscar el punto exacto donde se selecciona la próxima tarea a ejecutar y realizar el chequeo de los frames mapeados a la misma antes de que se realice el cambio de contexto en el procesador. Se definió la función `pgsa_check_task_pages()` en la interfaz del módulo para que sea invocada desde la lógica del scheduler. Ya se cuenta con un procedimiento que encapsula la verificación para el espacio de memoria de un único hilo, es la función `pgsa_mm_check_pages()` definida para llevar a cabo la nueva estrategia del demonio.

Como se mencionó más arriba existe una bandera de modo para configurar desde espacio de usuario la habilitación o no de este mecanismo y otra para restringir su aplicación solo a los procesos registrados. En la práctica se comprobó que la verificación de todos los procesos del sistema no sólo disminuye la efectividad de este mecanismo sino que afecta considerablemente el rendimiento del sistema, por lo cual se aconseja su uso sólo para un conjunto de procesos registrados.

3.7.7. Corrección de errores. Hasta ahora se describió un producto que utiliza códigos para detección de errores en bits, para buscar cambios en el contenido de la memoria. El mecanismo más intuitivo y sencillo para solucionar dichos errores fue apelar nuevamente a la criptografía, esta vez con códigos correctores. Existen varios tipos de código ampliamente utilizados para el almacenamiento y transmisión de datos, de los cuales se seleccionó un Código de Hamming [**42**].

Para seleccionar el método de corrección adecuado entre los existentes se debió tener en cuenta los pros y contras de los mismos con respecto a los requerimientos del problema. Recordando lo visto sobre rayos cósmicos y el efecto de éstos sobre la memoria como causa de la ocurrencia de soft errors, vimos que un rayo cósmico puede causar errores en varios bits vecinos. En las memorias ECC, las cuales corrigen dichos errores por hardware, es suficiente con métodos de corrección de un solo bit (como Códigos de Hamming). Esto es debido a que se puede tomar hipótesis sobre la distribución física en el chip de los bits contenidos en la palabra del código y garantizar que errores de más de un bit tengan altas probabilidades de afectar a lo sumo un bit dentro de cada palabra. Dado que en el caso de este prototipo se hace corrección de errores por software y la unidad de trabajo es una página de 4KB, no es posible asumir ninguna hipótesis sobre el chip de memoria y por tanto de la distribución física de nuestros bits en el mismo. Si asumimos el peor caso, los errores en bits contiguos en el hardware se corresponden a bits contiguos dentro de una página y por tanto es deseable soportar corrección de errores de varios bits, para garantizar mayor efectividad. Otro requerimiento importante es, igual que con la redundancia utilizada para la detección de errores, acotar el overhead agregado a una página por el código. Para el código de detección se tomo como límite para la redundancia los 64 bits, de los cuales se utilizaron sólo 32. En este caso dado que los códigos de corrección con poder de detección de varios bits suelen utilizar

redundancias mucho mayores, se va a fijar el límite en el doble 16 bytes. Esto sin dudas significa un overhead mayor en el tamaño de las estructuras *page*, las cuales como vimos en la sección 2.4 se encuentran alojadas de forma permanente en la memoria física reservada para el kernel, pero es un costo aceptable si el método brinda corrección de ráfagas de bits. Como se ha repetido de forma constante durante el desarrollo de los prototipos, el factor eficiencia computacional juega un rol muy importante a la hora de seleccionar la algoritmia a utilizar, debido a los contextos donde ejecuta la misma y los efectos que esto tiene sobre el rendimiento total del sistema. Finalmente, debido a los costos de implementación, optimización y verificación que puede tener incluir el manejo de uno de estos métodos en el Linux kernel se decidió seleccionar un código de fácil implementación o reutilización y dejar como trabajo a futuro la implementación de uno más complejo que cumpla de forma óptima con los requerimientos.

Entre los códigos de corrección más utilizados para almacenamiento y transmisión de datos, se encuentran los códigos de Golay [**42**], Reed-Solomon [**42**] y Hamming[**42**] los cuales fueron investigados y considerados antes de tomar una decisión y se discuten a continuación.

Primero veamos el Código Binario de Golay. Este código tiene la ventaja de corregir errores de hasta 3 bits, pero codifica 12 bits de datos en palabras de código de 23 o 24 bits, por lo cual supera ampliamente los límites plateados para la redundancia. Además, este código esta definido en base a conceptos de espacios vectoriales, por lo cual sus cálculos implican operaciones de matrices, lo que lo hace relativamente ineficiente. Por estas desventajas el mismo fue descartado.

Veamos ahora el código de Reed-Solomon[**42**] el cual es muy utilizado en medios de almacenamiento de datos (especialmente CD, DVD, BLUE-Ray, RAID, etc.). Este código utiliza palabras de código de n símbolos (con n máximo 255) para codificar palabras de datos de r símbolos, donde cada símbolo es un byte y se detectan ráfagas de errores de hasta (n-r)/2 símbolos. Aplicado al prototipo, se podría dividir la página en 16 palabras de 255 bytes y utilizar 1 byte de redundancia por palabra, con lo cual se respeta el límite de 16 bytes y se logra un poder de corrección de ráfagas de 4 bits. No se utilizó Reed-Solomon porque se estimó que sería muy costosa su implementación y verificación y no se encontró una fuente de reutilización con un grado de verificación confiable. Dado que teóricamente parece un código adecuado a los requerimientos establecidos, queda como trabajo a futuro la adaptación e implementación del mismo como parte del prototipo, para determinar su grado de aplicación práctica y eficiencia.

Finalmente veremos el Código de Hamming[**42**] que es el seleccionado para el prototipo y como vimos es también el más utilizado por las memorias ECC. Este código tiene la desventaja de sólo detectar errores de un bit, lo cual reduce su efectividad en caso de que en la práctica los errores ocurran en ráfagas, de lo cual tampoco se cuenta con pruebas. Sin embargo, es un código eficiente, simple de implementar y verificar. Además, genera una redundancia de apenas 16 bits para toda una página, lo cual resulta muy atractivo a la hora de minimizar el overhead. Por estas razones se decidió que era el que mejor se aplicaba a los objetivos del prototipo, permitiendo incluir funcionalidades mínimas de corrección de errores, sin un gran costo. En la sección 3.8.7 de implementación se dedicarán algunos párrafos a explicar cómo se adaptó este código para ser utilizado en un bloque de datos de

4KB y se separó la redundancia de los datos, lo cual no sucede en la definición teórica del mismo.

3.7.8. Rejuvenecimiento en el Sistema Operativo. La corrección de errores es un mecanismo directo y básico a aplicar en estos casos, pero en los conceptos de software aging vistos en el capítulo 2 se habla de otras técnicas a tener en cuenta en este contexto. El concepto más importante a repasar es el de rejuvenecimiento. El mismo básicamente implica reiniciar los recursos de un sistema bajo diferentes criterios para decidir cuándo y con qué granularidad. Existen enfoques que realizan el rejuvenecimiento de forma periódica sin importar el grado de envejecimiento. Hay otros que se basan en un diagnóstico estadístico del sistema para estimar cuando puede ser necesario el mismo. Con respecto a la granularidad hay enfoques que implican un reinicio completo del sistema y otros que actúan por componentes.

Toda la teoría encontrada se aplica principalmente al rejuvenecimiento desde el espacio de usuario, sin embargo, como se está realizando modificaciones a nivel del núcleo del sistema operativo se cuenta con importantes ventajas para tomar acciones más efectivas. Además, dichas medidas pueden ser tomadas de forma automática por el sistema operativo sin ninguna asistentencia de usuarios, lo cual aumenta la rapidez de acción de las mismas. En lo que respecta específicamente a soft errors en la memoria, área del envejecimiento de software atacada por el prototipo, contamos con que éste último fue pensado para tener altas probabilidades de detectar la ocurrencia de dichos errores. De esta forma, no es necesario apelar a rejuvenecimiento periódico o estadísticas de diagnóstico para aplicarlo, pues contamos con eventos precisos donde se procesa la detección de un error y se toman acciones al respecto. En lo que concierne a la granularidad, es claro que la unidad de trabajo siempre fueron los frames de memoria física tratados de forma independiente y por tanto es a nivel de los mismos que resulta lógico aplicar el rejuvenecimiento. La posibilidad de trabajar a este nivel de granularidad y con una distancia acotada en el tiempo entre la ocurrencia soft error y el manejo del mismo, disminuye mucho la probabilidad de que el error detectado o el proceso de rejuvenecimiento tengan un efecto negativo visible en la ejecución de el o los procesos afectados.

Resumiendo la letra del problema definido, se quiere rejuvenecer los datos contenidos en un frame de memoria física, cuando se detecta que ha ocurrido un error en el mismo. Para llevar a cabo el rejuvenecimiento se debe simplemente copiar en el frame los datos de la página virtual mapeada al mismo, para lo cual es necesario contar con un respaldo de ésta. Como vimos en la sección 2.4, existen dos clases de página a la que puede estar mapeado un frame. Por un lado están las páginas que mapean archivos en disco que funcionan como cache en memoria de los mismos o se utilizan para la implementación de memoria compartida y por otro existen las llamadas anónimas que contienen básicamente las secciones de datos de los procesos (constantes, variables, etc.). Dado que técnicamente las páginas de sólo lectura podrían pertenecer a cualquiera de estas categorías, se debe contemplar ambos casos.

Para manejar la necesidad de un almacén de respaldo de las páginas y un procedimiento de recuperación desde el mismo se intentó reutilizar mecanismos existentes en el kernel para fines similares. Para las páginas mapeadas a archivo el almacén de respaldo es el propio archivo en disco y el procedimiento de recuperación es la operación de lectura del archivo al frame que se utilizó para cargarlo inicialmente.

Investigando se encontró y evaluó más de un camino para iniciar dicha operación, los cuales se discuten en los párrafos siguientes. Ahora verá el caso de las páginas anónimas, para las cuales no existe por defecto un respaldo en disco o la memoria de su contenido. Para poder generar dicho respaldo sería necesario realizar una copia de la página de datos contenida en un frame cuando éste ingresa en el conjunto de páginas de sólo lectura y destruirla cuando sale del mismo, hacer la copia en cualquier otro momento pondría en riesgo la fidelidad de los datos. Dado que estos eventos se dan en su mayoría en manejadores de interrupción lo que implica un contexto delicado y que podrían existir problemas de sincronización durante la copia, el mecanismo para realizar la copia a disco podría ser bastante complejo. Sin ir más lejos esto se encuentra implementado para las áreas de swap del sistema de memoria virtual, para cuyo manejo y sincronización el mecanismo de swapping utiliza el concepto de swap cache. La opción más factible para poder implementar dicho almacén en disco sería reutilizar parte de la lógica del mecanismo de swap y los archivos de swap pero creando una instancia independiente de los mismos para no interferir con su tarea, la cual semánticamente no tiene mucha relación con la planteada. Se evaluó dicha opción y se entendió que su complejidad excedía el alcance del proyecto. La otra posibilidad era hacer el almacén en memoria, opción que se descartó porque no se cuenta con una cota sobre la cantidad de páginas anónimas de sólo lectura que puede haber en el sistema en un momento dado y alojar de forma permanente copias de páginas completas en memoria puede llegar a provocar un estado de escasez de memoria en el sistema. Para evitar entrar en dicho estado, sería necesario modificar el Mecanismo de Reclamación de páginas del kernel visto en la sección 2.4 para que incluya estas copias y sean swapeadas antes de convertirse en un problema, una vez más se considero que esta tarea implicaba una complejidad que excedía el alcance. La investigación más profunda de estas posibilidades y la selección e implementación de la que se considere más apropiada queda como trabajo a futuro, por lo que en esta versión no se hace rejuvenecimiento de páginas anónimas.

Veamos ahora el caso de frames mapeados a archivos para el cual se logró implementar el rejuvenecimiento de forma exitosa. Se consideraron dos estrategias para reutilizar la operación de carga de una página de un archivo a un frame. La primera opción fue utilizar la primitiva `try_to_free_pages()` sobre el frame deseado. Como vimos en la sección 2.4 la primitiva `shrink_page_list()` se encarga de llevar a cabo la liberación efectiva de los frames seleccionados por el algoritmo de reclamación de páginas, eliminando previamente los mapeos de los mismos con procesos de usuario. Lo que motivó esta idea fue que para el caso de frames mapeados a archivo no se envía al swap el contenido del frame, justamente porque el mismo ya se encuentra en el disco en el archivo. Dado que el cambio en los bits del frame ocurrió por un fallo del hardware, la página no fue marcada como “sucia” y no se propagará al archivo el error en los bits antes de liberar el frame. Luego, cuando algún proceso intente acceder dicha página, se provocará un fallo de página como vimos en la sección 2.4 y se leerá una copia limpia de errores desde el archivo. El problema con esta opción es que la función `shrink_page_list()` verifica algunas condiciones sobre el estado actual del frame y puede decidir no liberarlo, por tanto no sería efectiva en todos los casos.

La otra opción considerada constaba de investigar el mecanismo de carga de los datos desde el archivo hacia el frame utilizado como parte de la paginación bajo

demanda y extraer sólo la parte del manejo de archivos sin incluir otros mecanismos involucrados, en particular el Cache de Páginas. Como se vio, la paginación bajo demanda actúa para mapear a un frame una página virtual faltante desde el handler de la excepción de fallo de página. Se investigó el handler para el caso de los mapeos a archivos y se llegó como se esperaba al Cache de Páginas, ver [**12**] capítulo 15. En general es común que el sistema operativo use caches por software en memoria de los archivos en disco accedidos por los procesos, para poder satisfacer más rápido futuros pedidos. Linux implementa este concepto en el Cache de Páginas, por el cual pasan todos los pedidos de páginas de archivos y es él quien se encarga de alojarlas en frames físicos de memoria cuando es necesario o retornar un frame preexistente cuando es posible compartirlo entre varios procesos. Investigando internamente el funcionamiento del Cache de Páginas, se encontró la forma de invocar las operaciones de sistema de archivos para leer una página de datos desde el archivo correspondiente a la región de memoria virtual del proceso (VMA) del proceso hacia un frame. Reutilizando dichos mecanismos se logró refrescar una página como se deseaba y por tanto esta opción fue la utilizada para agregar la nueva funcionalidad al prototipo. En la sección 3.8.10 se dan más detalles sobre las operaciones utilizadas.

3.7.9. Rejuvenecimiento en el espacio de usuarios. En la sección anterior se refrescó el concepto de rejuvenecimiento y se mencionó que la teoría existente sobre el mismo aplica principalmente al espacio de usuarios. Ya se aplicaron técnicas para combatir el envejecimiento desde el sistema operativo, en especial el provocado por la ocurrencia de soft errors en la memoria. Dado que el rejuvenecimiento en espacio de usuario es una técnica muy utilizada, se entendió que en este caso la responsabilidad del sistema operativo era asistir dicha clase de rejuvenecimiento, brindando información y facilidades a los procesos que lo implementan. Se detectó aquí, una nueva fuente de acciones a tomar ante la detección de un error en la memoria, que permitan a un proceso desde el espacio de usuario contar con toda la información útil sobre la ocurrencia de errores con la que se cuenta internamente en el módulo. Se consideró necesario diseñar acciones, registros de datos y mecanismos de interacción específicos para este objetivo.

Se estudiaron los requerimientos desde el punto de vista de una aplicación de usuario que desea utilizar técnicas de rejuvenecimiento ante la ocurrencia de errores en su espacio de memoria. Primero, se decidió proveer a la aplicación de la posibilidad de recibir notificaciones sincrónicas de los eventos de ocurrencia de un error en su espacio de memoria. La recepción del evento en principio no necesita incluir información específica, basta con saber de su existencia para tomar acciones de rejuvenecimiento de alta granularidad. Aquí se encontró una de las justificaciones para el registro de procesos, dado que el módulo debe saber si el proceso está esperando el evento y lo que es más importante si es capaz de manejar el mismo. Un ejemplo claro de esto se verá en la próxima sección cuando se hable del envío de señales para realizar las notificaciones y los efectos de las mismas en procesos que no las manejan correctamente. Con el registro de procesos el módulo de software aging sabe si la aplicación está interesada en recibir el evento y en qué hilo de la misma, o incluso si se desea recibir en un proceso independiente al grupo de tareas donde se produce el fallo.

Para asistir un rejuvenecimiento más específico con nivel de granularidad a nivel de componentes o hilos de la aplicación, es necesario brindar información más

detallada sobre el error, en particular dónde ocurrió el mismo. Para esto se decidió publicar mediante algún mecanismo asincrónico, hacia el espacio de usuarios, una serie datos que se consideraron útiles. La publicación se organizó por grupo de tareas para que cada uno pudiera identificar rápidamente la información de errores que lo afectan. Se decidió publicar esta información para todos los grupos que fueran afectados por un error, independientemente si están registrados o no, dado que a diferencia de los eventos en este caso no hay ningún efecto negativo sobre el proceso. Se decidió incluir un registro para el grupo, por cada error detectado. Cada registro incluye información que permite al proceso determinar donde exactamente ocurrió el error. Los datos específicos que se decidió incluir se muestran en la sección 3.7.10.

La publicación estará disponible desde que ocurre el error hasta que termina el grupo de tareas afectado. Se verá con más detalle cómo se realiza el envío de eventos y la publicación, en la próxima sección.

La figura 12 ilustra la lógica diseñada en `pgsa_process_error()` para la aplicación de todos las acciones ante errores vistas en las últimas secciones.

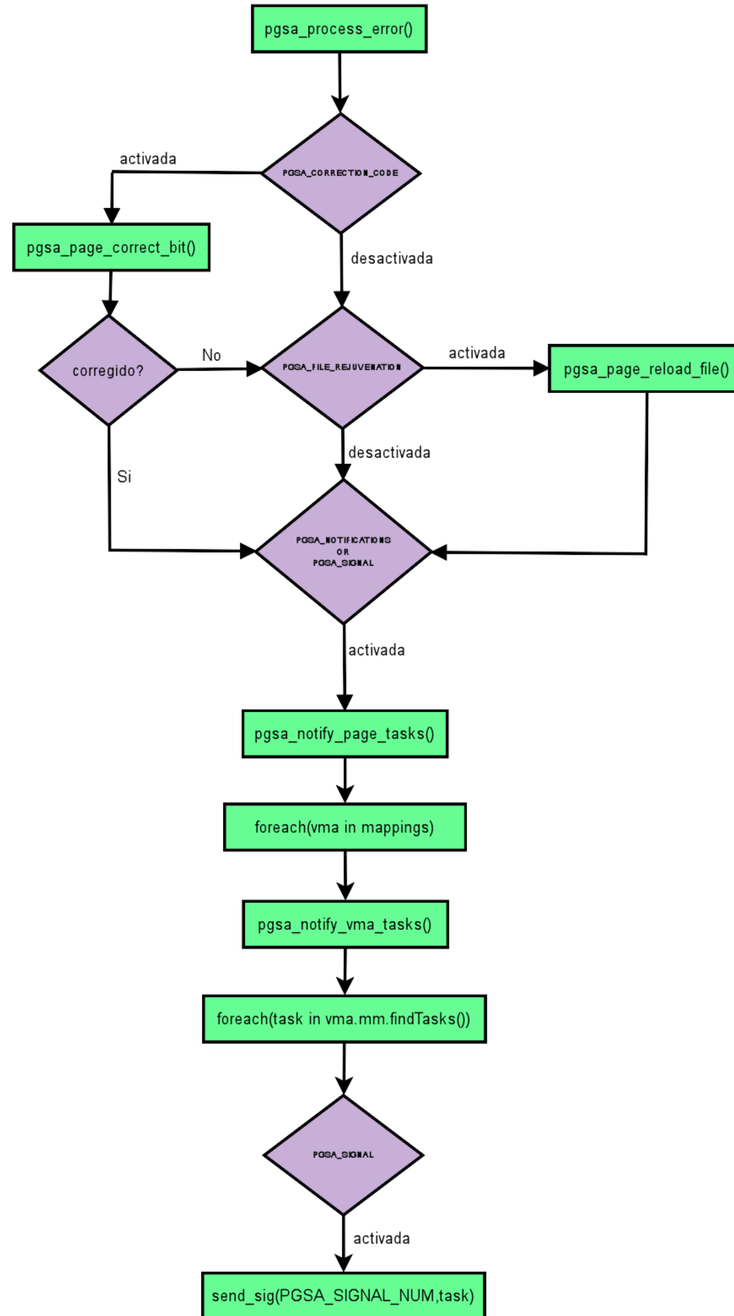


FIGURA 12. Diseño de procesamiento de Errores

3.7.10. Interfaz con espacio de usuario. Una vez diseñada la lógica del prototipo, se deben definir formas de proveer información al usuario para que éste sepa qué está pasando en el demonio y pueda modificar el funcionamiento del mismo. En esta sección se presentarán las opciones que se estudiaron para realizar esta comunicación, además se presentará la interfaz de la comunicación diseñada.

3.7.10.1. Servicios estudiados. Se identificaron varias formas mediante las cuales era posible implementar esta parte del proyecto, como ser *DebugFS*[**13**], *ProcFS*[**17**], *syscalls* y *NetLink sockets*[**14**].

Las *syscalls* son servicios que proporciona el SO que son invocables desde el espacio de usuario, como ser sockets, manejo de semáforos o memoria compartida. Las mismas se definen a nivel de cada arquitectura y llevan a la modificación de varios fuentes del kernel source original por lo que rápidamente fueron descartadas como forma de interactuar con el módulo ya que, provocaban un cambio muy grande al kernel. Además, con las mismas no se podía interactuar pasivamente con los procesos del espacio de usuario, es decir, no había forma de dejar una señal de que algo que se consideraba podría interesarle al usuario había ocurrido, como ser la detección de un error.

La siguiente opción evaluada fueron los *NetLink sockets*, que son sockets entre procesos del espacio de usuario con procesos del espacio de sistema. Un problema principal de los mismos era que se precisaba que hubiera un thread esperando los mensajes, esto o bien llevaría a la creación de un segundo demonio o que el demonio ya definido realizara dos funciones distintas lo que rompía la modularidad del mismo por lo que fue rechazada. Los mismos tenían como ventaja frente a las *syscalls* que no se debía modificar fuentes extra módulo para su implementación, pero tenían la misma desventaja de que no se podía interactuar pasivamente con el espacio de usuario.

Después se estudió *DebugFS*, el mismo es un *Sistemas de Archivos Virtuales* [1] del kernel. Los VFS son servicios del SO que permiten generar archivos que pueden ser vistos desde el espacio de usuario, pero que en vez de residir en los discos físicos existen solamente como una serie de callbacks definidas por medio de una API para cada archivo. Esto provoca que cuando se lea o escriba sobre alguno de esos archivos en lugar de invocar a las rutinas del kernel para trabajar sobre archivos físicos se invoca las callbacks provistas, brindando al módulo el acceso a los datos que el usuario está ingresando y/o teniendo acceso a un buffer desde el cual es posible enviar información que va a llegar al espacio de usuario. Estos VFS se montan en lugares particulares del sistema de archivos de forma de que se sepa que todo lo que está bajo cierta raíz no es realmente un archivo. Además, al montarse en ciertos lugares en particular se asegura que todo lo que está bajo esa raíz tiene el mismo significado, es decir, todos los archivos tratan sobre un tema particular del sistema. La API del servicio *DebugFS* permite de forma muy sencilla compartir variables desde el interior del kernel hacia el espacio de usuario. Esto provoca que el usuario pueda leer en cualquier momento el valor que esa variable tiene dentro del SO o que el usuario pueda modificar ese valor y que el mismo se propague inmediatamente hacia el kernel y que en el funcionamiento de un módulo que interactúe con ese valor luego del cambio se vea solamente el valor ingresado. Este servicio también permite un método más "complejo" de interacción, en el cual para cada archivo definido además de indicarle el espacio de memoria sobre el cual actúa el mismo,

[1] VFS por sus nombre en inglés

también se definen una serie de funciones que van a ser invocadas para las distintas operaciones que se provoquen sobre ese archivo desde el espacio de usuario. Estas son las callbacks de las que se habló anteriormente. Uno de los motivos por el cual se decidió no utilizar *DebugFS* es que como su nombre indica, este sistema de archivos fue creado para depurar el sistema y el uso que se le quiere dar no implica solamente tareas de debug, sino que incluye publicación de datos de operación del sistema y esto rompería la semántica del ambiente donde se monta el VFS. Además, el servicio de *DebugFS* puede no estar presente en todos los ambientes y por tanto no es aconsejable que la comunicación del módulo dependa del mismo.

Por último se evaluó la utilización de otro VFS, en este caso *ProcFS*. Este sistema de archivos es el que se utiliza para proveer a los usuarios con información sobre el comportamiento de los procesos del sistema. El mismo contiene entre otras cosas la información sobre el comportamiento de cada proceso, como ser, un listado de los archivos que el mismo tiene abiertos, el comando que se ejecutó para iniciar el proceso y los detalles de sus áreas de memoria, como direcciones virtuales, permisos, etc. Dado que el objetivo era publicar información sobre el funcionamiento de un proceso del sistema y reportes sobre errores en memoria de otros procesos, que éste era el lugar correcto para publicar esta información. Por esto, sumado a que este VFS permite también el método de interacción mediante callbacks con el espacio de usuario, se decidió utilizar este servicio para implementar la comunicación entre el módulo y el espacio de usuarios.

3.7.10.2. Información a publicar. Luego que se decidió el servicio del kernel a utilizar para publicar la información, había que decidir qué información se iba a publicar. Para informar sobre el funcionamiento del demonio se decidió definir los siguientes archivos *checks*, *cycles*, *errors*, *log_kpgsa* y *pgsa_mode*. Esta información es en orden, cantidad de chequeos ejecutados en el último ciclo, cantidad de ciclos ejecutados, cantidad de errores detectados durante todo el tiempo que ha estado activo el módulo, el nivel de log en que se encuentra el demonio y el modo en que se encuentra ejecutando el módulo. Todo esto se publica bajo el directorio `/proc/kpgsa`.

3.7.10.3. Interacción usuario sistema. En esta sección se presentan las formas mediante las cuales el módulo permite a los usuarios interactuar con sus funcionalidades. El diseño de la interfaz de usuario apunta a lograr un producto más cerrado y con menos modificaciones de gran impacto al kernel. Se verán más detalles sobre esta interacción en la sección 3.8.11.

Cambio de modo. El módulo implementa varios modos de operación distintos por lo que se necesitaba tener la posibilidad de cambiar de modo de forma dinámica. Para esto se creó un archivo `/proc/kpgsa/pgsa_mode` que mapea al modo de operación. De esta forma se puede saber en qué modo está operando el sistema y modificarlo dinámicamente desde la línea de comandos o por medio de operaciones de archivos. Al crear un archivo modificable en el `/proc` el mismo permite la implementación de funciones de callback para cuando se lee o se escribe al mismo, esto permite que en las mismas se agregue la lógica, por ejemplo, de levantar el demonio si el mismo estaba desactivado.

Registro de procesos. El módulo permite el registro de procesos que van a ser chequeados y de agentes que se van a encargar de la recuperación del mismo si ocurre un error que no se pueda solucionar. Para esto, se creó el archivo

`/proc/kpgsa/register`. En la función de escritura hacia el archivo se realiza el registro de la task a controlar y la task a notificar en caso de error. El archivo espera que se ingrese un texto en la forma de PID1-PID2, donde PID1 es el *pid* de la tarea a notificar y PID2 es el *pid* de la tarea cuyo *Task Group* va a ser chequeado.

Registro de Errores. Se decidió utilizar el *procFS* para publicar datos sobre los errores ocurridos, de forma que una aplicación que sea notificada sobre la ocurrencia de uno, tenga la posibilidad de solucionarlo. Para esto se va a crear bajo `kpgsa` un directorio que tiene como nombre el pid del proceso para el cual falló la memoria y dentro de esos directorios se van a generar archivos nombrados *Error_N* donde N es un número que va creciendo según la cantidad de errores que tenga la misma aplicación. Luego de que la aplicación donde se detectaron los errores termina se debe limpiar del *procFS* los archivos generados, por lo que es necesario tener una lista con todos los archivos generados para reportar los errores.

Notificación de Errores. Existe también la funcionalidad de notificación de errores a procesos agentes. Para enviar estas notificaciones es necesario buscar una forma de enviar mensajes a los procesos indicando que se produjo un error. Se decidió utilizar señales para esto debido a que es la forma estándar de notificar eventos del kernel a procesos del espacio de usuario. Esta forma no permite identificar el proceso que tuvo el error de memoria, pero es la más sencilla de implementar por parte de los procesos agentes. Otra forma podría haber sido que los procesos agentes abrieran puertos de escucha en sockets internos, de esta forma se podría notificar claramente cuál fue el proceso que tuvo el error, pero es más complicado para mantener de parte de dichos procesos, ya que deben tener un hilo especial esperando por los mensajes y no solamente esperar que les lleguen señales.

3.8. Implementación

En la sección 3.7 quedó definida la solución a nivel de diseño, pero dado que su implementación no es trivial, se considera necesario incluir este apartado dedicado a explicar los detalles de la misma. Esto es en parte porque la tarea de implementar e integrar el diseño planteado con el kernel no fue mecánica y requirió de una constante investigación y resolución de problemas no previstos en la etapa de diseño. Encontramos buenos ejemplos de esto en la secciones 3.8.8, donde se explica cómo se lograron efectivamente el chequeo del próximo proceso a ejecutar o en la sección 3.8.11 donde se explica cómo se trabajo con *procFS* para la implementación de la interfaz de usuarios. También se documenta aquí, la reutilización de código y APIs existentes en el kernel, como el mecanismo de lectura de una página de archivo explicado en la sección 3.8.10, la reutilización del algoritmo de CRC 32 bits explicada en la sección 3.8.6 o la implementación del registro de procesos por medio de colecciones de biblioteca del kernel, que se verá en la próxima sección. Finalmente, también se incluyen en esta sección detalles sobre la adaptación de algoritmos conocidos como el Código de Hamming que se discute en la sección 3.8.9.

3.8.1. Modularización. Se decidió crear un módulo PGSA dentro del kernel que contenga la lógica de implementación de la solución diseñada. En el resto de los módulos del kernel se incluyen solamente invocaciones a operaciones definidas en la interfaz de PGSA para este fin. Una excepción a esto es la definición de los campos que forman el complemento para cada frame, los cuales como se verá se incluyen directamente como parte de una de las estructuras principales del kernel, la *page*.

Para definir la interfaz del módulo se creó el fuente `pgsa.h`. Dado que el código del mismo es independiente de la arquitectura en la cual se compile el kernel, dicho fuente está incluido bajo el directorio `{KERNEL}/include/linux`. La definición concreta de las operaciones contenidas en esta interfaz se incluye en el fuente `pgsa.c` el cual se decidió incluir en el directorio `{KERNEL}/mm`, subárbol correspondiente a los fuentes del Memory Manager, debido a que la interfaz es invocada exclusivamente desde módulos del MM. Además, fue necesario modificar el *Makefile* del MM para que incluya el nuevo módulo y el mismo sea linkeado como parte del Manejador de memoria.

3.8.2. Complemento de page. Se diseñó un conjunto de campos que forman un complemento a la estructura *page*, descriptor de los frames del sistema. Ahora veamos cómo se implementó efectivamente, la colección de los mismos. La solución seleccionada debe permitir rápido acceso a esta información para un frame cualquiera y debe hacer un uso eficiente de la memoria dado que el conjunto de frames puede llegar a incluir casi toda la memoria física del equipo. Se consideraron dos posibles soluciones tomando en cuenta los requisitos. La primera consta de definir una colección propia con un registro para cada frame del conjunto que provea operaciones con orden constante. La segunda es simplemente incluir los nuevos campos en la estructura *page*, utilizada por el kernel para describir y gestionar los frames de memoria física y a las cuales se puede acceder con una macro con orden constante a partir del número de frame deseado. La segunda opción se entendió era la más lógica y era más coherente con el diseño del sistema operativo. Además, dado que no se encontró en el kernel una colección que cumpliera con las características

necesarias para implementar la primera opción, ésta se volvió la más costosa. Dado que el kernel mantiene siempre alojadas en memoria física las estructuras *page* correspondientes, se verificó que los bytes extra que se agregan a las mismas no causaran aumentos peligrosos en dichas áreas de memoria, haciendo rápidamente un prototipo. Al final de esta sección se hará una discusión del overhead resultante.

Como se explicó en la sección 3.7, para mantener el estado de los frames de memoria física se va a mantener para cada uno dos contadores, un código de redundancia CRC 32bits y un código de corrección de Hamming. Se dijo además que los valores de dichos campos para cada frame serían almacenados el descriptor del frame, su *struct page* asociada. El primer contador se llamó `pgsa_page_count` y mantiene el total de mapeos de procesos de espacio de usuario a la página física, es decir, el total de áreas de memoria virtual (VMA) que incluyen el frame. El segundo contador se llamo `pgsa_writers_count` mantiene el total de mapeos del frame en espacios de proceso de usuario con permisos de escritura, los cuales están determinados por aquellas VMAs que mapean al frame y cuyas flags indican permisos de escritura al tener seteado el bit VM_WRITE. El tercer campo que se creó fue el campo crc32, que contiene el valor del cálculo de un CRC 32bits como fue visto para la página contenida en el frame. Finalmente, el último campo corresponde a los bits de redundancia del código de corrección de Hamming. De esta forma, se modificó la definición de la *struct page* en el fuente `mm_types.h` bajo `{KERNEL}/include/linux`, agregando la definición de los campos.

```
...
/*PGSOFT*/
atomic_t crc32;

atomic_t pgsa_mapcount;
/* Cuenta las vmareas que estan mapeando esta página */

atomic_t pgsa_writers_count;
/* Cuenta los writers que estan mapeando esta página */

atomic_t pgsa_hamming_code;
/*PGSOFT*/
...
```

Como vemos todos los campos se definieron de tipo *atomic_t*. El tipo *atomic_t* es un tipo de datos definido en el Linux kernel el cual garantiza el acceso atómico siempre y cuando se utilicen para ello las macros definidas para consulta y modificación del mismo. Utilizando correctamente este tipo de datos, es posible hacer modificaciones concurrentes a estos contadores, sin generar race conditions.

Tomando en cuenta que para la arquitectura x86 el tipo de datos seleccionado utiliza enteros de 32 bits, el overhead total generado a la estructura *page* asciende a 16 bytes. En el caso de los contadores, la precisición de esta representación puede parecer excesiva, pero tomando en cuenta las operaciones atómicas que brinda, el overhead extra vale la pena. El caso del código de corrección es similar, dado que el mismo solo utiliza 16 bits para el tamaño de página por defecto de la arquitectura, pero cuenta además con la justificación de que en caso de seleccionarse el uso de páginas de 8KB, sería necesario un byte extra. De esta manera el overhead neto

introducido por los códigos de detección y corrección representa un 0.2 % para páginas de 4KB y el overhead neto del total de los campos asciende aproximadamente a un 0.4 % en el mismo caso.

3.8.2.1. Inicialización. Cuando inicia el sistema operativo, el kernel crea e inicializa una estructura *page* para cada frame de memoria física. Dado que se agregaron nuevos campos en la definición de la *page*, será necesario también inicializarlos en caso de que corresponda. En particular ambos contadores de mapeos deben ser inicializados en cero antes de que el frame sea liberado para su uso durante la carga. En el fuente `mm.h` bajo el directorio `{KERNEL}/include/linux`, se define la función `init_page_count()` donde se inicializa el campo de la estructura *page* que se utiliza como contador de usuarios. Dado que ese contador se debe inicializar antes de liberar la página igual que los definidos, se considero este un buen lugar para agregar nuestras inicializaciones

3.8.3. Eventos de cambios en los mapeos. En la sección 3.8.2 se definieron los campos que forman parte del complemento a la page. El tercero de ellos es el *crc32*, que como se explicó en la sección 3.7 debe contener el código CRC 32bits de la página de 4K contenida en el frame que describe la *page*, siempre que esta pertenezca al subconjunto de páginas de sólo lectura. Para esto es necesario cargar el nuevo cálculo del código en el campo cada vez que el frame entra al conjunto. Como se vio en la sección 3.7, esto se da cuando el cambio en los contadores indica una transición entre un estado que indica no pertenencia al conjunto y uno que indica pertenencia al mismo. Para la carga del código durante estas transiciones, se definió la función `u32 pgsa_page_calc_crc32(struct page * page)`, que encapsula el cálculo y carga de la redundancia para el frame de memoria descrito por el parámetro *page*. El algoritmo utilizado se verá con detalle más adelante. Lo más interesante de esta función, sin embargo, está en el tratamiento de los frames, el cual se detallará a continuación.

Como se vio en la sección 2.4, en su inicialización el kernel clasifica ciertos rangos de frames como reservados, pues estos contienen la imagen del propio kernel, la BIOS o son utilizados por dispositivos de hardware. Que esos frames estén reservados implica que los mismos no puedan ser asignados dinámicamente, ni swapeados a disco por el MM. El número del primer frame de memoria utilizable del kernel puede ser accedido a través de la variable `min_low_pfn` y el del último a través de `max_pfn`. A su vez, vimos que debido a limitaciones del hardware en arquitecturas de 32 bits existen entre los utilizables, frames para los cuales el kernel no tiene mapeada una dirección virtual fija dentro de su espacio de memoria. Ese rango de frames es denominado high memory o memoria alta y para direccionar dentro de los mismos deben ser mapeados a una dirección virtual dentro del espacio de memoria del kernel. Se puede acceder al número del primer frame de esta zona mediante la variable `highstart_pfn` o tomando en cuenta que se puede calcular como `max_low_pfn` + 1 y la zona se extiende hasta el final de la memoria usable (`max_pfn`). Para el manejo de dichos frames el kernel cuenta con el concepto de mapeos temporales, mediante los cuales se puede direccionar temporalmente con una dirección virtual tomada de un conjunto reservado. Se proveen dos clases de mapeos de este tipo. La primera son los mapeos denominados permanentes, los cuales están pensados para ser de alta duración y no se corresponden con las necesidades de este prototipo, dado que los eventos generados implican intervenciones cortas sobre una gran cantidad de frames y lo más importante, ocurren en contextos en los cuales

no se puede dormir. La segunda clase son los denominados temporales los cuales se realizan de forma atómica y pueden ser usados en contextos donde no se puede dormir. Estos fueron los utilizados para acceder la high memory en todos los casos. Para realizar estos mapeos el kernel provee la función `kmap_atomic()` la cual recibe una estructura *page* y la mapea en la memoria virtual del kernel a la dirección reservada solicitada. Existe un conjunto de direcciones reservadas identificadas por constantes las cuales corresponden a diferentes tareas del kernel. Entre estas existen dos para propósitos generales llamadas *KM_ USER0* y *KM_ USER1* que fueron las utilizadas en este caso, para no colisionar con otros componentes del kernel. El mapeo temporal puede ser liberado por medio de la primitiva `kunmap_atomic()` o sobrescrito en el próximo mapeo. Este procedimiento fue utilizado tanto para el cálculo del código CRC como el código de corrección.

3.8.4. Registro de Procesos. Se verá aquí los detalles de la implementación del registro de procesos diseñado en la sección 3.7.3. Para implementar la colección de procesos registrados, se utilizó la implementación de lista doblemente encadenada que forma parte de la API del kernel, ver [**12**] capítulo 3. Este tipo de datos permite definir una lista cuyos nodos sean una estructura creada a conveniencia, con la única restricción de que la misma debe contener un miembro de tipo *list_ head*. Para implementar la lista de procesos registrados se definió la estructura `pgsa_reg_proc` la cual cuenta con los siguientes campos:

- `struct task_struct *task` : descriptor de proceso que indica el grupo de tareas a registrar.
- `struct task_struct *notify_task` : descriptor de proceso que indica el proceso o hilo que cumplirá el rol de agente del grupo registrado.

El registro de procesos se creó como una instancia global a nivel de módulo de la lista definida bajo el nombre de `registered_procs`. Se definió además para el acceso sincronizado a la misma un read-write lock (mecanismo de sincronización provisto por las APIs del kernel llamado `rwlock`, ver [**12**]) llamado `registered_procs_lock`.

La implementación de las operaciones diseñadas para el registro de procesos se hizo en base a las primitivas que provee la API para manejar las listas, sin mayores complicaciones. Dichas operaciones son invocadas por la interfaz con el espacio de usuarios al manejar las registraciones.

3.8.5. Modo. Las banderas definidas en la sección 3.7.4 se implementaron como una variable global llamada `pgsa_mode` de tipo entero *atomic_ t* el cual ya se ha usado en el prototipo y provee operaciones para su manipulación que garantizan la correcta sincronización. Cada bandera corresponde a un bit de la variable, se definió una serie de constantes para determinar cuál en cada caso. Hasta ahora sólo se utiliza el byte más bajo, por lo cual hay posibilidad de agregar nuevas banderas en el futuro. Se tomó como convención que la parte baja del byte corresponde a las banderas del grupo de detección y la alta a las del grupo de acciones.

3.8.6. CRC 32 bits. Como se mencionó en la sección 3.7.5 se reutilizó una biblioteca definida en el kernel para realizar el cálculo del CRC 32 bits. La misma fue incorporada al código del módulo PGSA por medio de la directiva `#include <linux/crc32.h>`. Para calcular el CRC de una página se utilizó la función `u32 crc32(u32 seed, void* data, u32 length)` que recibe como parámetros una semilla para el CRC, un puntero al buffer de memoria al cual se desea calcular el CRC

y el largo de dicho buffer. En este caso, se decidió utilizar como semilla el 0, la dirección virtual del frame que contiene la página obtenida por medio de la primitiva `page_address()` como puntero al buffer y el tamaño de página dado por la constante *PAGE_SIZE* del kernel para el largo. Esto retorna un *u32* con el valor del CRC calculado para la página.

Algo que puede no resultar trivial o intuitivo, es entender como direccionar dentro de un frame físico de memoria. Cuando se está en el contexto de los procesos de usuario es claro que la página se accederá por medio del espacio virtual del proceso y esas direcciones virtuales se resolverán a partir de las tablas de páginas del mismo como se vio en la sección 2.4. Lo que se debió tener en cuenta en este caso, es que el uso del CRC que se hace es en modo kernel (mayormente en contexto de interrupciones) y por tanto para indicar la dirección virtual que resolviera correctamente al frame deseado se debía utilizar la dirección virtual asignada a dicho frame en las tablas de páginas del kernel. Como se vio en la sección 2.4 Linux asigna inicialmente una dirección virtual fija a cada frame, lo cual se refleja en sus tablas de páginas. De esta forma se debía obtener la dirección virtual asignada por el kernel a la estructura *page* correspondiente al frame deseado. La primitiva `page_address()` que se mencionó en el párrafo anterior hace exactamente esto. Una excepción a esto se da con los frames pertenecientes a la memoria alta, caso para el cual ya se explicó en la sección 3.8.3 que es necesario utilizar mapeos temporales.

3.8.7. Demonio. En la sección 3.7.6 se dijo que se decidió utilizar un demonio a nivel de kernel para realizar las verificaciones en busca de cambios en los frames del conjunto de trabajo. La decisión de implementar dicho demonio por medio de un Kernel Thread, se justifica a continuación.

Linux necesita delegar algunas tareas a procesos interminables o demonios, debido a que sería ineficiente para todo el sistema ejecutarlas de otra forma. Dado que muchos de ellos ejecutan exclusivamente en modo kernel, existen para esto los Kernel Threads, que son procesos más livianos sin el contexto innecesario correspondiente a modo Usuario, ver [**12**] sección 3.4.2. Se decidió implementar el demonio como un Kernel Thread porque la tarea continua que realizara el mismo requiere planificación para no afectar al sistema y es exactamente el tipo para el cual fue creada y optimizada esta clase de procesos.

Se creó el Kernel Thread “kpgsacheckd” por medio de la función `kthread_run()`. A ésta se le indica el nombre del thread y el procedimiento que ejecutará el mismo. Para este fin se creó en la interfaz del módulo PGSA la operación `kpgsacheckd()` que encapsula la lógica ejecutada por el demonio. La creación del thread se realiza como parte de las tareas de inicialización en el manejador `pgsa_init()` de la interfaz con el kernel. El mismo puede ser terminado e iniciado nuevamente además en los manejadores de interfaz de usuario que veremos más adelante.

Como se dijo en la sección 3.7.6 cuando la bandera *PGSA_CHECK_ALL* está activada, el demonio debe iterar sobre los frames de memoria física de forma ordenada sin utilizar ninguna clase de prioridades. El rango de frames a revisar está determinado por los frames utilizables, que como se dijo son aquellos cuyo número de frame está entre `min_low_pfn` y `max_pfn`. Para encapsular el procesamiento de cada frame del rango se definió la función `pgsa_page_check()`. Esta se encarga verificar si el frame pertenece al conjunto de trabajo mediante los valores de los contadores como ya se detalló y en caso positivo se realiza la verificación de la

redundancia repitiendo el cálculo y comparando contra el valor almacenado. El cálculo se realiza con la función `pgsa_page_calc_crc32()` ya comentada en la sección 3.8.3. Finalmente, cuando se detecta un error en un frame se invoca a la función `pgsa_process_error()` la cual se definió con el propósito de encapsular las acciones a tomar ante este evento y se discutirá en próximas secciones.

3.8.8. Chequeo por procesos. En las sección 3.7.6 se definieron también otras formas de realizar la verificación de frames en búsqueda de errores, en las cuales la verificación se hace por tarea y no directamente por frame como en el demonio. Como se vio en el diagrama de invocaciones presentado, se encapsuló la verificación de un espacio de memoria virtual de proceso en el procedimiento `pgsa_mm_check_pages()` . Comenzaremos detallando éste y luego se verá como se implementó cada mecanismo a partir de él.

El procedimiento `pgsa_mm_check_pages()` recibe un descriptor de espacio de memoria (*mm_struct*) e itera sobre la lista de sus regiones de memoria virtual (VMAs). Para cada VMA, se itera sobre las páginas virtuales que la componen verificando si están mapeadas a un frame por medio de la primitiva `follow_page()` ya utilizada para el manejo de cambios en los permisos, la cual en caso positivo retorna el descriptor del frame, una estructura *page*. Para evitar chequear múltiples veces el mismo frame, la función recibe una lista con los números de frame (pfn) de aquellos que ya fueron verificados por el mecanismo que la invoca. Se verifica la *page* obtenida con la función `pgsa_page_check()` si su pfn no pertenece a la lista y se lo agrega a la misma. El tipo de datos utilizado para implementar la lista de pfn se creó utilizando la lista doblemente encadenada de la API del kernel igual que para el registro de procesos. Se definió la estructura `pgsa_pfn_list` con un único miembro numérico representando el pfn.

Se comentará ahora la estrategia alternativa de verificación en el demonio. Al comienzo de cada iteración se consulta la bandera *PGSA_CHECK_ALL* y si está activada se usa la estrategia mencionada anteriormente. Si no está activada se invoca la función `pgsa_check_registered_tasks()`. En esta función se itera sobre la colección de procesos registrados por medio de una macro provista por la API para este fin y se procesa cada elemento mediante la función `pgsa_check_task_group()`, pasándole el nodo más una instancia de lista de pfn, creada vacía antes de comenzar la iteración.

El descriptor de tareas *task_struct* cuenta en su definición con punteros que permiten iterar sobre el árbol de tareas. El kernel cuenta con una macro `next_thread()` que permite iterar de forma circular sobre todos los descriptores de los hilos de un grupo a partir de uno arbitrario. Mediante esta macro y el descriptor del hilo del grupo usado para realizar el registro, se logra iterar sobre todos los miembros. Para cada tarea se verifica su espacio de memoria en búsqueda de errores por medio de una invocación a `pgsa_mm_check_pages()` pasándole el descriptor *mm_struct* contenido en su *task_struct* y la lista de pfn recibida. De esta forma la lista se mantiene de forma global a todos los procesos que se verifican durante la iteración y cada frame se chequea una sola vez, implementando de forma óptima la estrategia diseñada.

Se verá ahora al mecanismo de chequeo del próximo proceso a ejecutar. En la sección 3.7 se agregó la función `pgsa_check_task_pages()` para verificar el próximo proceso a ejecutar. Se investigó el código del scheduler de Linux y se encontró que la tarea de seleccionar el próximo proceso a ejecutar y hacer el cambio

de contexto en el procesador, la lleva a cabo la función `schedule()` definida en `{KERNEL}/kernel/sched.c`. Se incluyó la interfaz del módulo en `sched.c` y se agregó una invocación a `pgsa_check_task_pages()` pasándole como parámetro el *task_struct* correspondiente al proceso seleccionado para ejecutar, antes de que se realice el cambio de contexto. La función internamente se encarga de consultar si el mecanismo está habilitado por medio de la bandera *PGSA_CHECK_CURRENT* y si el proceso a ejecutar está registrado. La versión final de la implementación no contempla el caso de que la bandera *PGSA_CHECK_ALL* esté activada y se chequean únicamente tareas registradas en todos los casos, porque procesar cada hilo que va a utilizar el procesador resultó tener un efecto negativo en la performance del sistema. Si la tarea supera todos los chequeos se invoca una vez más la función `pgsa_mm_check_pages()` para el descriptor *mm_struct* de su espacio de memoria y una lista de pfn vacía.

3.8.9. Código de corrección. Para la corrección de errores se decidió utilizar Códigos de Hamming[**42**]. La justificación de la elección y la discusión de sus consecuencias en el prototipo y el kernel fueron vistas en la sección 3.7.7. Se verá ahora algunos detalles de la adaptación e implementación de la técnica en el módulo PGSA. El método está compuesto por un algoritmo de codificación para generar una palabra de código que incluye mezclados datos y un conjunto de bits de control, y un algoritmo de corrección que permite calcular la posición del error ocurrido dentro de dicha palabra y cambiar el bit correspondiente. Dado que para el prototipo los datos de una palabra son el contenido completo de un frame de memoria y no es posible almacenar ahí la redundancia, se hizo una pequeña abstracción del buffer de datos enumerando sus bits de acuerdo a su posición en la palabra de código y haciendo lo propio con los bits de control, los cuales se codifican en los bits de un número entero. El algoritmo original de codificación es el siguiente:

1. Todos los bits cuya posición es potencia de dos se utilizan como bits de control mediante el cálculo de paridad (posiciones 1, 2, 4, 8, 16, 32, 64, etc.).
2. Los bits del resto de posiciones son utilizados como bits de datos (posiciones 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.).
3. Cada bit de control se calcula como bit de paridad de todos los bits de datos cuya posición en la palabra tenga, en su representación binaria, en 1 el bit correspondiente a la potencia de dos de la posición del bit de control. Es decir que el bit de control 2. Así, por ejemplo, para los primeros bits de control se tiene:
   - Posición 1 (2^0 = 1), se comprueba los bits: 3, 5, 7, 9, 11, 13...
   - Posición 2 (2^1 = 2), los bits: 3, 6, 7, 10, 11, 14, 15...
   - Posición 4 (2^2 = 4), los bits: 5, 6, 7, 12, 13, 14, 15, 20, 21, 22, 23...
   - En la Posición 8 (2^3 = 8) tendríamos: 9, 10, 11, 12, 13, 14, 15, 24-31...

En el caso del prototipo el buffer de datos es una página de tamaño 4096 bytes, o sea 32769 bits y la palabra de código resultante tiene 32784(2^15) posiciones y por tanto se tienen 16 bits de control (2^0...2^15) los cuales se codifican en los dos bytes más bajos de un entero para ser almacenados. Codificarlos de esta manera permite que su cálculo se realice de forma eficiente utilizando operadores a nivel de bit. Para mantener la nueva redundancia a nivel de los frames se utilizó el mismo

criterio que para la detección y se incluyó un nuevo campo de tipo atomic_t en la estructura *page* llamado `pgsa_hamming_code`.

Para implementar el algoritmo de codificación se creó en el módulo la función `hamming_parity_code()` la cual retorna el valor de los bits de control para el contenido de un frame, indicado por su dirección virtual en el kernel. Igual que para el cálculo del CRC, fue necesario contemplar el caso de las páginas de memoria alta al obtener dicha dirección virtual, por eso se hizo un wrapper de la misma utilizando el mismo mecanismo de mapeos temporales que para el CRC. Dicho wrapper se implementó en la función `pgsa_page_calc_hamming()` y en caso de que la bandera *PGSA_ CORRECTION_ CODE* esté activada es invocada en todos los eventos de cambio en los mapeos en un frame al igual que el cálculo del CRC para actualizar la redundancia en la *page* cuando corresponde. Recordemos que el manejo de dichos eventos está encapsulado en la función `pgsa_page_process_mappings_change()`.

Según el algoritmo original de Hamming para la corrección de errores de un bit, basta recalcular el código y comparar los bits de control del mismo contra los almacenados para detectar la posición del error. Esto es porque la representación binaria del número que indica la posición de un bit de datos, determina los bits de control que lo comprueban, siendo estos las potencias de 2 con coeficiente 1 en dicha representación. Esto quiere decir que la suma de las posiciones de los bits de control que comprueban un bit de datos es igual a la posición del bit de datos en el código. De esta manera, si ocurre un cambio en un sólo bit de datos, basta comparar los nuevos bits de control contra los almacenados y la suma de las posiciones de aquellos que cambiaron indicará la posición del bit con error en el código. En el caso de la adaptación del algoritmo realizada, el concepto es el mismo, pero con una diferencia en el cálculo de la posición. Dado que se utiliza una abstracción para enumerar los bits de datos, la posición obtenida no es la posición del bit en la página, sino que se le debe restar uno por cada bit de redundancia cuya posición es menor a ésta. Este mecanismo de corrección se implemento en la función `pgsa_page_correct_bit()` la cual recalcula el código, determina la posición en la página del bit con error, cambia su valor y re-testea el CRC de la misma para ver si volvió a coincidir con el almacenado. En caso de que la corrección haya fallado se vuelve el bit a su valor original para evitar causar más errores en la página. La verificación por medio de CRC es necesaria porque Hamming por sí mismo no es capaz de determinar si ocurrió más de un error en cuyo caso la posición calculada es incorrecta.

Como se dijo en la sección 3.8.7, se encapsuló el manejo de los errores detectados por `pgsa_page_check()` en la función `pgsa_process_error()`. Se desencadenan ahí todas las acciones ante errores implementadas. Para el caso del código de corrección si la bandera *PGSA_ CORRECTION_ CODE* está activada se invoca `pgsa_page_correct_bit()` sobre el frame afectado, para intentar corregir el error. En `pgsa_page_check()` se intenta corregir el error por medio de todos los mecanismos habilitados y en caso de no tener éxito se recalcula y almacena en la *page* el nuevo valor de las redundancias. Esto se hace para evitar la detección repetida del mismo error y posibilitar detección y corrección de futuros errores. Luego de esto, si están habilitadas las notificaciones, éstas se envían.

3.8.10. Rejuvenecimiento de archivos.

En la sección 3.7.8 se diseñó un mecanismo de rejuvenecimiento automático para frames mapeados a archivos, el cual forma parte del grupo de acciones posibles ante la detección de un error. La

estrategia seleccionada para lograr el rejuvenecimiento, fue leer nuevamente el contenido de la página de disco sin hacer modificaciones en los mapeos y sin pasar por el Cache de Páginas [**12**], componente de Linux que funciona como cache de frames con mapeos a archivos y centraliza la creación y distribución de los mismos. Para investigar cómo lograr esto se revisó el handler de fallo de páginas, buscando seguir el camino de la carga inicial del frame cuando fue requerido por un proceso. En la sección 3.7.2 vimos que en particular para el caso de los mapeos de archivos tanto lineales, como no lineales el manejo de la excepción se hace en la operación `__do_fault()` definida en `{KERNEL}/mm/memory.c` en el árbol de distribución del kernel. Las VMA cuentan con un conjunto de punteros a operaciones entre las cuales se encuentra `fault()`, la cual es invocada desde `__do_fault()` con la información del fallo para atender el mismo. Para el caso de las áreas mapeadas a archivos normalmente `fault()` apunta a la función `filemap_fault()` definida en `/mm/filemap.c`. El manejador de falla utiliza el Cache de Páginas para obtener el frame, consultando primero si ya existe en el mismo y solicitándole su lectura en caso negativo por medio de la operación `page_cache_read()`. Esta última es la que inicia directamente la operación de lectura sobre el archivo por medio de la operación `readpage()` del objeto *address_space* asociado al archivo. No se entrará en detalles sobre la interacción entre archivos y el Cache de Páginas, porque esto implicaría un desvío de los objetivos. Basta con saber que se entendió posible reutilizar la invocación a la primitiva `readpage()`, accediendo a la misma mediante el objeto *address_space* asociado al frame a rejuvenecer y el archivo asociado a alguna de las VMA que mapean dicho frame. Un cuidado a tener que se detectó es que esta operación requiere que la *page* esté protegida con un lock, e intenta liberar dicho lock al terminar.

Los resultados de esta investigación se pusieron en práctica en la definición del método `pgsa_page_reload_file()` el cual como las demás acciones es invocado en `pgsa_process_error()` en caso de que la bandera correspondiente (a saber *PGSA_FILE_REJUVENATION*) esté activada.

3.8.11. Interfaz con espacio de usuario. En esta sección trataremos sobre la implementación de la interfaz diseñada en la sección 3.7.10. Se verá cómo se utilizaron los servicios provistos por el componente *procFS* del SO para generar los archivos necesarios para esta interfaz, también se verá cómo se realizaron las callbacks desde el módulo para proveer sus servicios.

3.8.11.1. Valores de estadísticas. Llamamos valores estadísticos a los campos *cycles*, *checks* y *errors*, los cuales son sólo de lectura desde el espacio de usuario y por tanto para implementar su publicación en `/proc/kpgsa` se utilizó la función `create_proc_read_entry()` de la API de *procFS*. Dicha función recibe como parámetros, el nombre del archivo que se desea crear, el modo a asignar en el espacio de usuario (para el prototipo se usa *0444* para que pueda ser leído por todos), luego recibe el directorio donde se quiere colocar el archivo, para el modulo PGSA es `/proc/kpgsa`, también recibe un puntero a la operación que se invoca al leer el archivo y por último un puntero a los datos que se van a acceder en la función de lectura. En el caso del prototipo se pasan punteros a los datos de tipo *atomic_t* que se están compartiendo.

La función de callback utilizada es `pgsa_read_atomic()` que recibe como parámetros un array de caracteres donde se colocarán los datos que se desean leer, otro puntero que indica a partir de qué ubicación se deben colocar los datos, un offset

que señala a partir de qué byte de la información publicada se desea leer. Un entero que señala cuantos bytes se esperan leer, un puntero a otro entero que se utiliza para notificar si quedaron más datos para proveerle al usuario o no y el puntero a la información que se pasa al invocar la operación `create_proc_read_entry()`. Esta operación espera que se pase como resultado la cantidad de bytes que se escribieron en el buffer que recibido.

El contenido de esta función de callback es muy sencillo en el modulo desarrollado y consta solamente de castear el puntero al tipo de datos ingresado y luego escribir el valor del mismo en el buffer mediante la operación `sprintf`.

3.8.11.2. Flag de logueo. Este flag es similar a los anteriores, pero a diferencia de ellos, este sí se debe poder modificar por parte del usuario. Por ese motivo, no se podía compartir de la misma forma que los otros campos y se debió utilizar la función `proc_create` también de la API de *procFS* y que tiene como primeros 3 parámetros los mismos que la función anterior y que tiene un sólo parámetro más, que es un puntero a una estructura del tipo `file_operations` donde registramos una función para el callback que se genera en el `open()`, el `read()` y el `write()`.

En el caso de la función de `open()`, la misma recibe como parámetro un inodo que contiene un puntero a la información que se asoció al archivo, que no es el caso del `proc_create`, el cual no permite asociar datos al archivo. Como segundo parámetro recibe una estructura de tipo `file` donde se carga la información que le va a llegar a las funciones de `read()` y `write()`. Lo que se decidió es realizar funciones de `open()` independientes para cada archivo que se cree con `proc_create` y en las mismas cargar el campo `file->private_data` con un puntero a la variable que se va leer o modificar.

Estas funciones de callback tienen una serie de parámetros que difieren de las operaciones anteriores, la función de `read()` recibe un puntero a `struct file` que es donde en la función de `open()` se carga el puntero a los datos que queremos manejar dentro de la función de `read()` y `write()`. Como segundo parámetros se recibe un puntero a un buffer en espacio de usuario, el tercer parámetro es un contador que indica la cantidad de bytes que se desean leer y por último se tiene un puntero que se utiliza para retornar el offset dentro del buffer de usuario donde se está copiando la información solicitada. Como valor de retorno se espera lo mismo que en la operación anterior, la cantidad de bytes que se leyeron.

Se implementó una función llamada `pgsa_read_atomic_t` que se utiliza para todos los archivos que están asociados a campos *atomic_t*, esta es una función genérica que carga la variable desde `file->private_data` y luego se carga en un array de caracteres el valor en formato hexadecimal de ese campo, por último se utiliza la operación `simple_read_from_buffer` para escribir desde ese array a un buffer en espacio de usuario.

El callback para las operaciones de escritura sobre el archivo tiene los mismos parámetros que la operación de escritura. Lo que se realizó en este caso es leer el primer byte que el usuario ingresa y descartar el resto, para copiar ese byte desde espacio de usuario al espacio de kernel se utiliza la operación `copy_from_user`. Luego que se copió el byte se utilizó la función `sscanf` para pasarlo de carácter a un entero y se asignó a la variable `logKpgsa` para cambiar el modo de logueo del módulo.

3.8.11.3. Cambio de Modo. Como se mencionó en la sección 3.7.10 se mapea la variable del modo al archivo `pgsa_mode` bajo `/proc/kpgsa` en el espacio de usuario

y de la misma forma que en el sección anterior se debió implementar operaciones de callback para este archivo.

Para el cambio de modo se implementó una operación distinta para el `open()` del archivo y para la escritura del mismo, pero la función de lectura es la misma que en el caso anterior. La única diferencia en el `open()` es que se carga la variable `pgsa_mode` en los datos del `file` en lugar de `logKpgsa`. Para la escritura sobre el archivo lo que se realizó en la función de `write()` fue tomar los dos primeros bytes que ingresa el usuario y cargarlos en la variable `pgsa_mode` del módulo. Antes de esto se almacena en una variable local el valor que tenía `pgsa_mode` de forma de poder verificar luego que se realizó el cambio, si es necesario levantar el demonio o no. Esta sección está protegida por un spinlock (mecanismo para lograr exclusión mutua) llamado `pgsa_mode_lock`, el cual se implementó para evitar que si la función de callback se ejecuta en hilos concurrentes, esto resulte en la ejecución de dos demonios lo cual sería incorrecto y probablemente inestable.

3.8.11.4. Registro de Errores. Como se detalló en la sección 3.7, para todos los errores que ocurren se registra información sobre el mismo para todos los procesos que tienen mapeada la página donde se produjo el error. Para realizar esto se implementaron dos funciones auxiliares, la primera `pgsa_notify_page_tasks` que recibe la página donde se produjo el error y un *char* indicando si se pudo solucionar o no. Esta función busca todas las *vmas* que tienen mapeada la página, luego cuando se obtiene la VMA se invoca la función `pgsa_notify_vma_tasks` que recibe la *vma* y la *page* donde se produjo el error, y además recibe el *char* mencionado. Para ese par busca todas las tareas que tengan mapeada la página por medio de esa *vma* y crea para esa task, si no está ya creado, un directorio en `/proc/kpgsa` con el *pid* de la task que falló y dentro de ese directorio crea un archivo que contiene la información del error. Para poder borrar estos archivos cuando la task termine se crea una lista de archivos creados que se almacenan en la siguiente estructura.

```
struct pgsa_tgid_files {
        struct list_head list;
        //list_head necesario para la lista

        struct proc_dir_entry *entry;
        //Archivo que se quiere almacenar

        struct proc_dir_entry *parent;
        //Directorio donde se encuentra el archivo

        struct pgsa_error_data *error_data;
        //Estructura donde se almacena un puntero a
        //los datos que se utilizan para crear el archivo

        int pid;
        //pid de la tarea para la cual se reporta el error
};
```

Esta estructura se almacena en la lista `proc_error_files` la cual es protegida con el `spinlock_t proc_error_files_lock`. Cuando se ejecuta el exit para un proceso y se llama a `pgsa_process_task_exit` dentro de ese proceso se recorre la lista de archivos de error generados y se buscan todos los archivos asociados al *pid* que termina para borrar todos sus archivos.

Para crear el archivo con el registro del error se utilizó el modo de crear archivos de sólo lectura como se vio en la sección 2.6.1 pero con una distinta operación para la lectura. A esta operación le pasamos como data sobre la cual va a operar una estructura del tipo `pgsa_error_data` donde tenemos un puntero al *struct page* donde ocurrió el error y un *char* indicando si se solucionó el error o no. Luego al ser invocada esta operación escribimos en el archivo los siguientes datos:

- Frame: value. Donde value es el número de frame asociado a la página donde se generó el error.
- Virtual Address: address. Donde address es la dirección en hexadecimal de la página donde se produjo el error
- VMA: start-end permisos. Donde start y end son la dirección de comienzo y fin de la VMA donde está la página para la cual se produjo el error y permisos son cuatro caracteres con el siguiente significado:
  - 1° 'r' o '-' para indicar si la VMA tiene permisos de lectura o no.
  - 2° 'w' o '-' para indicar si la VMA tiene permisos de escritura o no. Se dejó este campo, aunque vaya siempre a tener el mismo valor('-'), para que esta salida tenga un formato similar a la que se tiene en `/proc/<PID>/maps`
  - 3° 'x' o '-' para indicar si la VMA tiene permisos de ejecución o no
  - 4° 's' o 'p' para indicar si la VMA está compartida o es privada al proceso.
- Si la VMA tiene un file asociado se muestra
  - File: path. Donde path es la ruta al archivo
  - File Offset: valor. Donde valor es la cantidad de *PAGE_SHIFTS* dentro del file que está la VMA que tuvo el error.
- Anon: valor. Donde valor vale *true* o *false* según si el VMA es un mapeo *anon* o no.
- Fixed: valor. Donde valor vale *true* o *false* dependiendo si el error fue solucionado o no.
- Crc Value: valor. Donde valor es el valor en hexadecimal de la página en este momento.

Mientras se está escribiendo todo esto se van calculando la cantidad de bytes que se llevan escritos y ese es el valor que se retorna. Además, al igual que el caso anterior, marcamos que se llegó al fin de los datos a retornar. Fue necesario crear la estructura `pgsa_error_data` ya que se decidió que el dato que indica si se corrigió o no el error no es significativo para agregarla al *struct page*.

3.8.11.5. Notificación de Errores. Luego de que se genera el archivo con el registro del error, si el task que tiene la página con error tiene el mismo *tgid* que algún proceso registrado entonces se envía una señal utilizando la función `send_sig` la cual recibe como parámetros el número de señal a enviar, el proceso a notificar y si se quiere enviar información del proceso que envía la señal o no. Para el prototipo se decidió enviar la señal *SIG_UNUSED/SIG_BADSYS* que no es común que los procesos la utilicen para funciones propias de los mismos.

3.8.11.6. Registro de Procesos. Como se explicó en la sección 3.7.10 para registrar un proceso se utiliza el archivo `register` en `/proc/kpgsa`. Para implementar este archivo se utilizo el método `proc_create` como en los casos anteriores, por lo que también hubo que crear funciones para el callback de la lectura y la escritura. En particular para este caso importa la función de escritura sobre el mismo, ya que

la función de lectura es un dummy que lo único que hace es indicar cómo funciona el archivo de registro por línea de comando.

Para la escritura se desarrolló la función `pgsa_write_register`. Dentro de esta función se carga desde el espacio de usuario lo que se escribió en el archivo. Luego de cargar esos datos se buscan las 2 task identificadas por los pids y se llama a la función `pgsa_register_task` con las *task_ structs* obtenidas de forma que queden registradas las mismas.

3.9. Verificación

En esta sección se trata la verificación del prototipo desarrollado. Se comentarán las pruebas realizadas y las metodologías que se utilizaron para el diseño y realización de las mismas. Dado que éste es un proyecto que trata un tema tan particular como el manejador de memoria del Linux kernel, seleccionar e implementar las pruebas a realizar no es una tarea trivial, ya que los juegos de datos serían aplicaciones y/o modificaciones a la memoria del sistema.

La sección 3.9.1 se dedicará a introducir el concepto de inyección de fallos, justificar la utilización de esa técnica para la verificación del prototipo y describir cómo se implementó la misma en el kernel. Luego, en la sección 3.9.2 se definirán y justificarán los criterios utilizados para la elección de los casos de prueba ejecutados para testear el correcto funcionamiento de la aplicación. Finalmente, en la sección 3.9.3 se reportará el estado de verificación final del producto.

3.9.1. Inyección de fallos. La principal funcionalidad del prototipo desarrollado, es la detección de cambios en bits que ocurrieron en frames de memoria asociados a páginas virtuales de procesos de usuario, con permisos de sólo lectura en todos los casos. Para verificar que se está brindando dicha funcionalidad correctamente se debe comprobar que se detectan errores ocurridos de forma externa al sistema operativo, en condiciones normales, en el hardware. Por una parte en la práctica no es posible garantizar la ocurrencia de dicha clase de errores o acotar los tiempos de espera por ellos. Por otro lado tampoco se puede asegurar que un error no es producto de un bug en la implementación del módulo, pues no se cuenta con información confiable a nivel del hardware sobre la ocurrencia del mismo. Además de las demoras, otro problema con esperar la ocurrencia natural de un error, es que el mismo no sería reproducible, lo cual es una característica deseable. Por el contrario, los errores serían aleatorios y por tanto no se puede determinar claramente qué caso se está probando al detectar el mismo. Sumado a esto, el no ser reproducibles hace muy difícil verificar que un error detectado en el código del prototipo haya sido corregido. Dado que provocar errores en el hardware puede llegar a ser una tarea compleja, costosa y la misma escapa al alcance de este proyecto, surgió la motivación de buscar una forma de poder generar estos errores por software. Para poder generar estos errores en memoria a demanda se decidió utilizar Fault-Injection o inyección de fallos.

Fault-Injection implica la inserción deliberada de fallas en un sistema de cómputo con el fin de determinar la respuesta del mismo. Ha probado ser un método muy efectivo para la validación de sistemas tolerantes a fallas como el que se desarrolló, permitiendo el cubrimiento de caminos en el código dedicados al manejo de errores, que de otra forma no sería posible testear. Esta técnica puede ser implementada a nivel de hardware o software. El contraste entre ambos recae principalmente en los puntos donde se puede hacer la inyección, el costo y el nivel de perturbación. Los métodos por hardware pueden inyectar fallos directamente en los chips, mientras los orientados a software se basan en modificar el directamente el estado del mismo. Por otra parte, los métodos por software son menos caros que los otros, pero incluyen un mayor overhead de perturbación del sistema, al ejecutar software en el mismo.

La aplicación de técnicas de inyección en el prototipo se hizo, como se adelantó, por sofware y apuntó a la memoria física del sistema. El mecanismo utilizado

consistió en provocar de forma deliberada un cambio en un byte de una página de memoria seleccionada, la cual se encuentra almacenada en un frame de la memoria física. Mediante esta estrategia fue posible probar las funcionalidades para la detección, corrección y notificación de errores en todos los casos que se consideró necesario para garantizar el correcto funcionamiento de las mismas. Se verá más adelante los criterios que se seleccionaron para verificar el módulo mediante la utilización de inyección, pero primero es importante explicar cómo se logró aplicar esta técnica en el Linux kernel.

3.9.1.1. Diseño e Implementación. Se consideraron distintas posibilidades para lograr la inyección de fallos. La primera fue utilizar FIK (Fault Injection Kernel) [**21**], un componente de software liviano y poco intrusivo con la capacidad de inyectar fallos en registros de CPU, memoria y llamadas a funciones. Este componente está implementado para Linux y cumple con las funcionalidades necesarias, sin embargo al realizar pruebas con éste se detecto una incompatibilidad, dado que el módulo PGSA no detectaba los errores inyectados a frames del conjunto de sólo lectura. Dicha incompatibilidad se atribuyó a que FIK cambia los permisos con los que se accede al frame para poder modificar el contenido del mismo, haciendo que éste no pertenezca al conjunto de sólo lectura y por tanto siendo ignorado por el mecanismo de detección de errores.

De esta forma, fue necesario encontrar otro mecanismo de más bajo nivel, que actuara en modo kernel, sin modificar los permisos con los que se accede al frame. Se optó entonces por desarrollar en código del kernel, el cual ejecuta en el espacio de memoria virtual del mismo y como ya se ha visto no tiene un esquema de protección de la memoria (el kernel confía en sí mismo) las funcionalidades necesarias y hacerlas accesibles al espacio de usuarios. Para realizar esto se probaron dos opciones, la primera fue con *DebugFS*[**13**] y la segunda con syscalls o llamadas al sistema [**12**], en ambos casos se aprovechó el conocimiento obtenido al momento de implementar las interfaces hacia el espacio de usuario en los primeros prototipos. Esto ya fue explicado en la sección 3.7.10 y por eso no se repetirá aquí. Simplemente hay que aclarar que en este caso es correcto utilizar alguna de estas opciones, dado que el mecanismo de inyección no forma parte del producto final, sino que es una herramienta para verificación del mismo.

La primera implementación realizada fue con *DebugFS*, en la cual un byte de la memoria era compartido con la función `debugfs_create_u8` desde el kernel hacia el espacio de usuario, pero fue posteriormente desechada, ya que solo se logró compartir en todos los casos un byte fijo de un mismo proceso y esto no permitía completar las pruebas deseadas. Como la primera implementación no era parametrizable y para lograr hacerlo se debía desarrollar una forma de pasarle esta información al sistema operativo, se decidió crear una syscall llamada `sys_kpgsa_inyect()` la cual se encarga de inyectar un nuevo valor en la dirección virtual indicada del espacio de memoria de un proceso de usuario. La misma recibe el pid del proceso a inyectar, la dirección de memoria virtual dentro del espacio de éste, y el nuevo valor que se quiere colocar en el byte indicado. Con esta función se logró una interfaz para inyectar errores en el contenido de la memoria física a través del kernel.

Veamos ahora cómo se implementó el trabajo efectivo de realizar el cambio en un byte en el cuerpo de la syscall. La función busca la estructura *task* (Process Control Block de Linux, ver [**12**]) asociada al pid indicado. Luego, examinando el

espacio de memoria virtual del *task* por medio de la primitiva `find_vm()` se encuentra el área de memoria virtual que contiene la dirección seleccionada. Después, se utiliza la primitiva `follow_page()` para buscar la estructura *page* a la que está mapeada la página que contiene la dirección virtual. Si la página no se encuentra mapeada a la memoria física, no es posible realizar la inyección y por tanto simplemente se retorna error. En caso de que lo esté, por medio de la estructura `page` se determina la dirección virtual correspondiente al byte en el espacio de memoria del kernel y con una simple asignación se logra cambiar el valor almacenado por el nuevo, sin necesidad de modificar los permisos de acceso al frame.

Para realizar de forma simple inyecciones de error utilizando la syscall desarrollada, se creó además, el programa `inyector`. El mismo es una utilidad de línea de comandos que recibe los mismos parámetros que la llamada al sistema y hace con éstos una invocación a la misma, terminando luego su ejecución.

3.9.1.2. Verificación. Para verificar el inyector de errores desarrollado fue necesario corroborar que el mismo era capaz de modificar efectivamente el contenido de la memoria física mapeada con permisos de sólo lectura, de forma transparente para el sistema y el módulo PGSA. Para testear esto se decidió crear algunos programas de prueba y hacer inyecciones en la memoria a la cual se encuentra mapeado un proceso que los ejecutara. El código fuente de estos programas se incluye en el apéndice F. Durante estas pruebas se hizo uso de la utilidad *objdump* con el parámetro -S sobre los binarios ejecutables de los programas de prueba para seleccionar los bytes de memoria en los que se realizaron las inyecciones. La misma muestra el programa en código de máquina junto con las direcciones virtuales sobre las que se realizan las operaciones, lo que permite seleccionar convenientemente el byte a modificar. Por medio de *objdump* solamente es posible ubicar bytes en las páginas de código, si se quiere modificar otras páginas (como las que contienen variables) es necesario usar un mecanismo distinto para ubicar su dirección en la memoria.

Se verán ahora en detalle los programas creados y las distintas pruebas realizadas con los mismos. Se decidió realizar inyección en distintas partes de los programas para comprobar el correcto funcionamiento de la inyección. En las pruebas realizadas se inyectó en constantes, llamadas a funciones y operaciones de incremento. Para estas pruebas, se decidió que el demonio imprimiera en el log del sistema cuando detectaba algún error de forma de facilitar el chequeo del mismo. El primer programa es una versión del clásico "Hola Mundo" que imprime dentro de un loop infinito una constante estática de tipo string conteniendo dicho texto. La prueba realizada constó de modificar el primer caracter del string por medio del inyector y verificar que en la salida estándar del proceso cambiaba el mensaje desplegado para mostrar el valor inyectado. Se puede observar la salida de la prueba en la Figura 13. En la misma, se puede comprobar que en el log del sistema se imprime el texto indicando que se detectó el error inyectado. Además, se puede observar que el texto que imprime el programa varia y se empieza a imprimir "Iola Mundo" (se decidió inyectar el carácter I ya que el mismo es el siguiente al carácter H en la tabla de códigos ascii y los mismos tienen un solo byte de diferencia).

El segundo programa de prueba es otro loop infinito, que en este caso se encarga de incrementar en uno e imprimir una variable en cada iteración. El mismo se creó para poder probar la inyección de error en una operación. El test realizado consistió en inyectar el byte que determinaba la operación a realizar, intercambiando la suma por una resta y de esta forma se verifica la inyección, corroborando que el valor

```
10.33.23.164 - PuTTY
Last login: Thu Apr 30 14:52:13 2009 from 10.33.47.132
Have a lot of fun...
linux-2lnt:~ # cd log/
linux-2lnt:~/log # tail -f messages | grep "error crc"
Apr 30 14:57:26 linux-2lnt kernel: PGSOFTAG error crc pfn 139204 crcviejo 2040490021 crcnuevo 3526330196
Apr 30 14:58:04 linux-2lnt kernel: PGSOFTAG error crc pfn 139204 crcviejo 2040490021 crcnuevo 3526330196

linux-2lnt:~ # ./inyector 3242 0804852c 73
3242 804852c I
Return value 0
Hola Mundo
Hola Mundo
Hola Mundo
Iola Mundo
Iola Mundo
```

Figura 13. Inyección en una constante

impreso pasa de aumentar a descender. La figura 14 muestra una extracción de la salida estándar durante la prueba. En esta también se comprueba que el error fue detectado y logeado, y se ve en la salida del programa que la variable se iba incrementando y luego empieza a decrementar.

```
10.33.23.164 - PuTTY
linux-2lnt:~/log # tail -f messages | grep "error crc"
Apr 30 15:12:57 linux-2lnt kernel: PGSOFTAG error crc pfn 114878 crcviejo 2017484391 crcnuevo 1394835702
Apr 30 15:12:58 linux-2lnt kernel: PGSOFTAG error crc pfn 114878 crcviejo 2017484391 crcnuevo 1394835702

linux-2lnt:~ # ./inyector 4004 804843b 109
4004 804843b m
Return value 0
value 35
value 36
value 37
value 38
value 39
value 40
value 41
value 40
value 39
```

Figura 14. Inyección en una función

Finalmente, se creó un tercer programa para verificar la inyección en el byte del programa que contiene la dirección a la que apunta la llamada a una función y provocar que éste pase a llamar a otra función distinta de la original. Para que esto se pueda verificar de forma gráfica se creó un programa con dos funciones que se encargan de imprimir los string “Hola Mundo” y “Adios Mundo” respectivamente. En su versión original el programa invoca ambas funciones dentro de un loop infinito. Esto se pensó de esta forma para que la salida del *objdump* permitiera determinar la dirección de invocación de cada función. La prueba realizada constó de inyectar en una de las llamadas la dirección utilizada en la otra, de forma que en la salida estándar se observara el mismo mensaje dos veces, luego de la inyección. La figura 15 muestra algunas salidas de la prueba. Como en los casos anteriores, se ve en la figura que se detecta el cambio y que en programa empieza a llamar a la función que no estaba programado para invocar.

Se concluye por lo tanto que todas las pruebas realizadas indicaron el correcto funcionamiento del mecanismo de inyección utilizado en el alcance de su aplicación a este proyecto.

```
10.33.23.164 - PuTTY
linux-2lnt:~/log # tail -f messages | grep "error crc"
Apr 30 15:47:39 linux-2lnt kernel: PGSOFTAG error crc pfn 117759 crcviejo 2434357137 crcnuevo 3620880946
Apr 30 15:47:39 linux-2lnt kernel: PGSOFTAG error crc pfn 117759 crcviejo 2434357137 crcnuevo 3620880946

linux-2lnt:~ # ./inyector 7995 804845c 176
7995 804845c °
Return value 0

Adios Mundo
Hola Mundo
Adios Mundo
Hola Mundo
Adios Mundo
Hola Mundo
Adios Mundo
Adios Mundo
Adios Mundo
```

FIGURA 15. Inyección en una llamada

3.9.2. Criterios de Verificación. Se describen aquí los criterios utilizados para seleccionar los casos de prueba, mediante los cuales, se entendió era posible garantizar un buen grado de verificación funcional del prototipo. Los resultados de la ejecución de las pruebas realizadas utilizando estos criterios se presentan en la sección 3.9.3.

3.9.2.1. Conjunto de frames de sólo lectura. Para determinar si es correcta la conformación del conjunto en todo momento, es necesario verificar dos condiciones. Primero, que todos los frames que se encuentran dentro del mismo son efectivamente de sólo lectura, esto es, todas las áreas de memoria virtual que los mapean tienen la bandera de escritura deshabilitada. Segundo, que no hay frames que cumplan las condiciones y no estén dentro del conjunto. Una forma directa de verificar ambas condiciones, es chequear en cada evento de cambios la consistencia del estado mantenido para el frame contra el conjunto de áreas virtuales derivado a partir del mecanismo de reverse mapping visto en la sección 2.4. Gracias a que durante el desarrollo del prototipo se implementó una estrategia alternativa para mantener el estado, basada en el mecanismo de reverse mapping, la cual finalmente fue descartada debido a sus desventajas de performance, la verificación propuesta se puede lograr simplemente insertando algunas comparaciones en el código. El criterio de aceptación de la prueba es la no detección de bugs en los chequeos agregados durante varias horas de ejecución y simulación de carga en el sistema.

3.9.2.2. Métodos de búsqueda de errores. La verificación de cada uno de estos métodos y las distintas estrategias con que cuentan, se puede resumir también en dos condiciones triviales. Las mismas se pueden enunciar como: todos los errores reportados corresponden a ocurrencia de fallas (reales o inyectadas) y todas las fallas (reales o inyectadas) ocurridas son reportadas. Para el caso de fallas reales, como ya se mencionó, no es posible inducir las mismas y la posibilidad de su ocurrencia es muy baja. De esta forma el criterio tomado fue cubrir todas las opciones de búsqueda disponibles y para cada una verificar mediante inyección que se reportan todos los fallos simulados y solamente estos. Dentro del testeo por inyección de cada opción se definieron los siguientes casos a cubrir:

1. Distintos tipos de mapeos: archivos, anónimos
2. Distintos tipos de frames: Memoria normal y alta
3. Distintos sectores del frame: casos límite

Según la estrategia, existen otros casos a probar. Para las que buscan sólo en procesos registrados, se decidió comprobar que se cumplían las condiciones planteadas haciendo inyecciones sobre el espacio de memoria de distintos grupos de procesos que se encontraban registrados y al mismo tiempo comprobar que no se hacían reportes de error para frames mapeados exclusivamente a procesos que no estaban registrados.

3.9.2.3. Acciones ante errores. En el caso de las distintas acciones ante errores no hay un único criterio de verificación. Sin embargo, hay algunas consideraciones generales a todas. Las mismas tienen en común la característica de ser independientes a la estrategia de búsqueda que haya detectado el error, pero no del método. Esto es debido a que el demonio y el chequeo del próximo proceso a obtener la CPU, ejecutan en distintos contextos. En particular en este último no se permite dormir, porque esto ocasionaría un bloqueo de todo el sistema. De esta forma, para cada acción se deberá contar con casos que la verifiquen con las distintas estrategias. Además, existen una serie de consideraciones particulares a cada funcionalidad que se analizan a continuación.

Rejuvenecimiento de archivos. En este caso la condición a verificar es la corrección todos los fallos ocurridos en los frames inyectados. Es decir, que los errores se reporten como solucionados. Esto se debe cumplir en todos los casos en que se inyectan fallos single-bit o multi-bit en frames con mapeos de archivos. Dentro de éstos se pueden diferenciar dos clases de mapeos que generan distintos casos de prueba: privados y compartidos.

Código de corrección. Aquí se debe verificar de la misma forma que en el caso anterior que cada error inyectado se reporte como solucionado, pero en este caso el mecanismo solo corrige errores de un bit y por tanto se debe verificar que sólo se corrigen los de este tipo. Para esto es importante que no esté activado al mismo tiempo el mecanismo de rejuvenecimiento de archivos. Dado que el código trata la página de 4K definida en el frame como una única palabra de bits, se debe verificar destinos offsets dentro del frame y dentro de los bytes contenidos en éste, para ver que se corrigen los errores en todos los casos.

Envío de señales. En este caso se debe verificar que para cada error inyectado, todos los procesos registrados que mapean el frame afectado reciban la señal definida para notificaciones del módulo PGSA y que aquellos que no estén registrados no la reciban.

Archivos de Error. Para la verificación de esta funcionalidad es necesario sumar un criterio general para todos los casos de prueba que se definan en base a los criterios definidos hasta el momento. El mismo implica que en todos los casos se verifique la correctitud de todos los datos reportados sobre el error y no sólo la existencia del archivo. Además, en todos los casos es necesario acceder al archivo con todos los comandos y rutas posibles para testear en busca de problemas de actualización del árbol de archivos creado. Para esto se debe incluir en los casos los siguientes chequeos:

- el nuevo archivo aparece listado en el directorio con comandos como ls
- es posible acceder al mismo entrando al directorio con cd y listándolo o editándolo con cat, more, vi, etc.
- es posible leer el archivo con rutas relativas o absolutas desde una aplicación

3.9.3. Estado final. En base a los criterios descritos se definió un conjunto de casos de pruebas. Se resume a continuación las consecuencias más importantes de

la ejecución de estos casos, pero podemos adelantar que el grado final de verificación del módulo es bueno. Se logró un buen cubrimiento de las funcionalidades y se mantienen algunos bugs sin corregir los cuales también serán detallados en los próximos párrafos.

3.9.3.1. Pruebas. Las pruebas fueron de gran utilidad, se detectaron varios defectos, la mayoría de los cuales se logró corregir con éxito. Los tests más interesantes, fueron los de comparación de los valores de los contadores mantenidos con los inferidos por medio de los reverse mappings. Estas validaron por una parte que se estaban teniendo en cuenta todos los eventos de alta y baja de mapeos existentes, pero indicaron también la existencia de un evento de cambio en los permisos de los mapeos que no se estaba manejando.

Con respecto al comportamiento de la inyección sobre frames que mapean archivos de código objeto, se pudo ver que si bien los cambios inyectados persisten y aparecen en nuevas instancias del programa que contiene dicho código objeto, esto no se debe a que los errores inyectados sean propagados al archivo. En alguna etapa de pruebas de inyección sobre el prototipo, se tuvo la teoría de que esto sucedía así, considerándose como una característica no deseable del sistema de inyección y que no permitía la verificación confiable del rejuvenecimiento de archivos. Esto es porque el archivo es la fuente de corrección utilizada por ese mecanismo y si los errores se propagaban al mismo no sería efectivo el rejuvenecimiento hecho y no se podría observar y verificar el mismo. Mediante las pruebas realizadas, se determino que el fenómeno de persistencia del error se debe a que en condiciones de abundancia de memoria libre, Linux no libera frames de los caches de disco y futuros pedidos del archivo se satisfacen con el frame existente, desde el cache, provocando la persistencia del error. Bajo estas condiciones, el rejuvenecimiento desde disco es completamente efectivo y soluciona un fallo persistente causado por el error de hardware.

3.9.3.2. Bugs remanentes. Existe un importante bug remanente en el sistema. El mismo indica la no detección de un cambio en los permisos de algunas áreas virtuales de memoria. A pesar de los esfuerzos realizados por encontrar el sitio donde sucede el evento, no se logró corregir el mismo. Este error provoca la esporádica inclusión dentro del conjunto de frames de sólo lectura de algún frame con mapeos anónimos de escritura, provocando la detección de falsos errores, los cuales pueden ser identificados e ignorados, revisando los permisos de mapeo del área virtual donde se produjo el error.

3.10. Pruebas de performance

Una vez verificado el prototipo desde el punto de vista funcional, se entendió necesario testear la performance del sistema. Para esto se seleccionó un conjunto de tests basados en pruebas de benchmarking para distintas aplicaciones, los cuales permitieron estimar los efectos del overhead y la reducción de recursos producidos en un sistema que ejecuta el modulo PGSA, en comparación con uno que ejecuta una distribución sin modificaciones (vainilla) de la misma versión del kernel. El objetivo de estas pruebas es comprobar si existe, y medir en caso de que exista, una posible reducción en la performance global del sistema y determinar si la misma se encuentra dentro de niveles aceptables. En la próxima sección se explicarán los criterios utilizados para seleccionar las pruebas y se introducirán las mismas. Luego, se dedicará a cada prueba una sección con el fin de describir los procedimientos

con que se aplicó la misma y los resultados obtenidos a partir de ésta. Las salidas generadas en estas pruebas se incluyen en el anexo G. Finalmente, se sacarán conclusiones generales de los resultados obtenidos.

3.10.1. Selección de las pruebas. Para testear la performance se decidió hacer uso de suites de software que se utilizan comúnmente en Linux. Para simular carga de usuarios en las mismas y determinar la eficiencia con la que ejecutan se reutilizaron herramientas para benchmarking ya existentes para dichas suites. El criterio para estimar la reducción de la performance observada en un kernel que ejecuta el módulo PGSA, se basó en repetir las pruebas en la misma configuración de hardware y sistema operativo, pero con un kernel vainilla de la misma versión que el utilizado para el desarrollo del módulo. Todas las pruebas aquí descritas se ejecutaron en un equipo con arquitectura Intel 32bits, procesador Pentium D 3.2GHz, con 1GB de RAM y sistema operativo OpenSuse 11.0 [**25**]. A no ser que se especifique explícitamente lo contrario, todas las pruebas fueron realizadas con una configuración del módulo en la cual están activadas las notificaciones, pero sin envío de señales. Los modos de búsqueda de errores varían para poder testear las distintas estrategias implementadas.

La primer prueba realizada apuntó a mostrar el efecto en la performance del módulo en un escenario compuesto de aplicaciones intensivas en entrada-salida (IO-bound). Para ello se consideró un sistema ejecutando un servidor web y un servidor de bases datos, típico escenario donde la alta disponibilidad es un requerimiento importante y por tanto el módulo PGSA encuentra un lugar de aplicación. Se seleccionó para esto el servidor web Apache [**22**], trabajando en conjunto con PHP [**24**] y el motor de base de datos MySQL [**23**], típica suite de software open source utilizada en Linux y conocida como LAMP[2]. Para simular carga en el servidor, se instaló además la aplicación web Wordpress[**26**], conocido sistema de gestión de contenido muy utilizado para la creación de blogs. Para simular carga y obtener estadísticas de performance se utilizó la utilidad `ab` de Apache. Se verá el procedimiento y los resultados con detalles en la próxima sección.

La segunda prueba realizada apuntó a trabajar con aplicaciones intensivas en el uso del procesador (CPU-bound). Para ello se seleccionó la utilidad de render de imágenes POV-Ray[**33**]. La misma se alimenta de un modelo de la imagen a generar, más las texturas a utilizar y genera representaciones 3D de las mismas. Esta utilidad contiene una opción de ejecución en la que genera una imagen predefinida que está diseñada para testear el rendimiento del sistema. La generación de las imágenes es un proceso muy intensivo a nivel de uso de ciclos del procesador. Esto provoca una interesante interacción entre una herramienta típicamente utilizada en un equipo de escritorio con el prototipo.

3.10.2. Apache y MySQL. La primer prueba de performance realizada se basa en la utilidad *ab* de Apache, la misma es capaz de realizar n consultas concurrentes a un servidor web solicitando siempre la misma url y sirve para medir la velocidad de despacho de éste. Se entendió que solicitar una página estática de html plano era una prueba demasiado simple, por lo que además del servidor web Apache 2.2.8 se instaló el módulo *mod_php* para brindar soporte PHP al mismo, junto con el motor de base de datos MySQL Ver 5.0.51a y Wordpress 2.7. La prueba de performance consistió en solicitar 10000 veces con 5 hilos concurrentes

[2]Linux Apache MySQL PHP

la página de inicio de Wordpress que realiza consultas a la base MySQL y utiliza PHP para realizar todo el trabajo. Al finalizar la prueba *ab* brinda un conjunto de estadísticas sobre el comportamiento de respuesta del servidor. Entre estas se seleccionaron como indicadores de performance la duración total de la prueba en segundos, la cantidad de pedidos completados por segundo, el tiempo de atención promedio de los pedidos y la cota superior de los tiempos de respuesta, válida para el 80 % de los pedidos.

Para estas pruebas el kernel incluyendo el módulo se va a correr en 2 modos, primero con el demonio revisando la totalidad de las páginas y luego con el demonio revisando solamente las páginas de los procesos registrados. Vamos a llamar a esto modo 146 y modo 142 respectivamente. Estos números son las flags en hexadecimal que están habilitadas para el modo del módulo como se ve en B.1.

Cada prueba se ejecutó cinco veces para cada kernel. En las figuras 16, 17, 18 y 19 se grafican los valores observados para los indicadores mencionados.

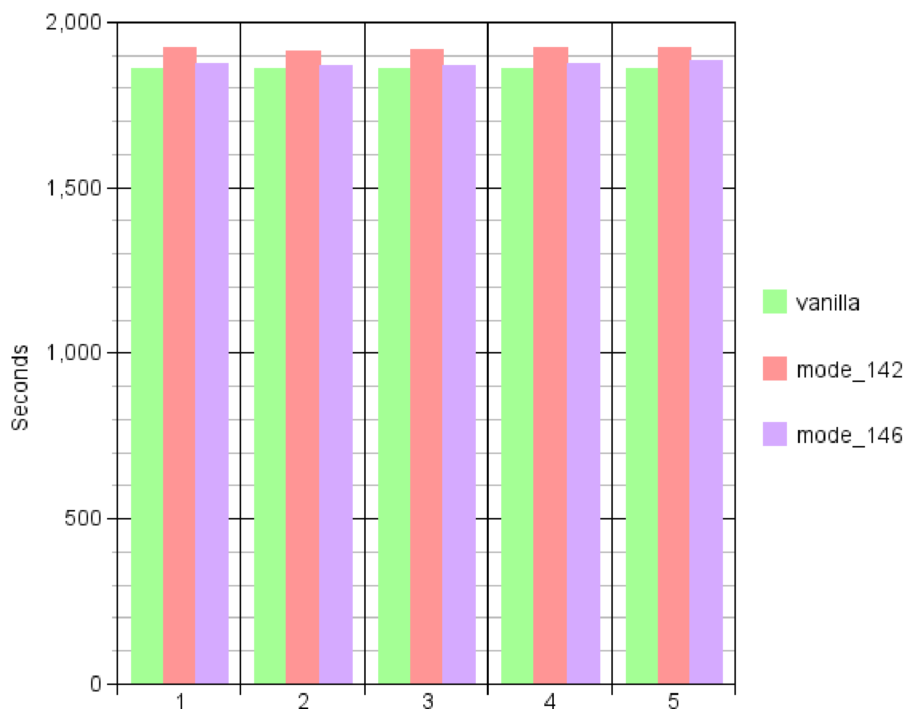


FIGURA 16. Tiempo total de los tests

Como se ve en la Figura 16 las diferencias no están en los dígitos más significativos, usando una escala basada en el 0 en el eje vertical, para las siguientes graficas se moverá el comienzo de dicho eje para que se pueda apreciar mejor la diferencia entre los distintos modos de ejecución.

De los promedios de los datos obtenidos en las pruebas se observa que para el indicador de duración total de la prueba, la cantidad de pedidos completados por segundo y el tiempo de atención promedio de los pedidos, la diferencia en performance

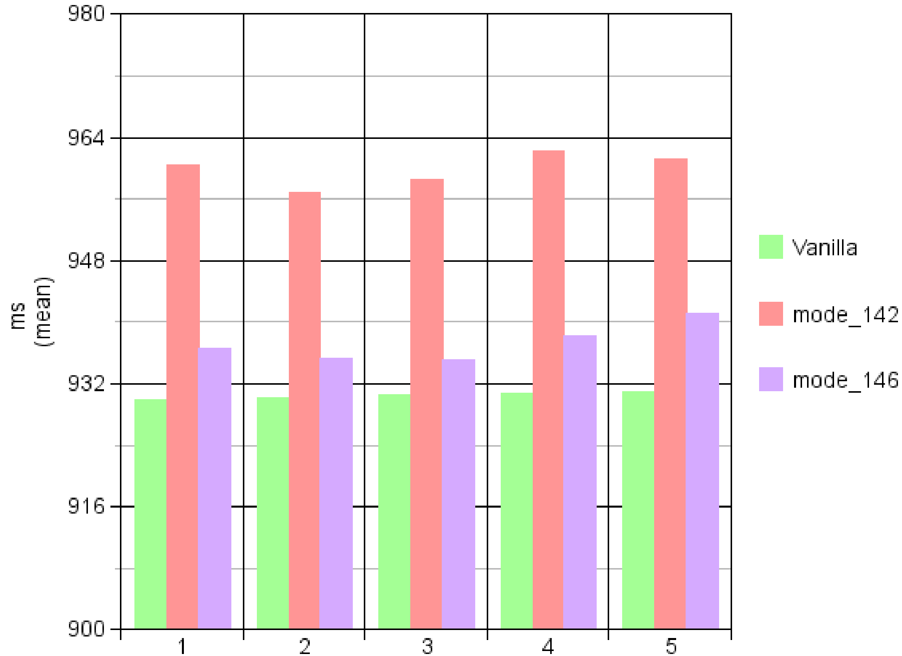


FIGURA 17. Tiempo por pedido

entre el kernel vainilla y el kernel que incluye el módulo es de aproximadamente 0.7 % para ejecuciones en el modo 146 y 3.1 % para el modo 142. Para los datos de la cota superior se ve una diferencia de performance de 0.6 % para el modo 146 y 11 % para el modo 142.

3.10.3. POV-Ray. Para esta prueba se utilizó el POV-Ray versión 3.6, último release estable al momento de comenzar las pruebas. Las pruebas fueron realizadas ejecutando el mismo render 5 veces para cada configuración del sistema. Las configuraciones utilizadas fueron las mismas que en el caso anterior. Para ejecutar las pruebas se corrió un script que invocaba al POV-Ray adjudicándole al mismo un ajuste en su valor de *nice* de -10. Esto se realizó para que el renderer tuviera más prioridad que el demonio a la hora de asignar tasks a ejecutar. Cada ejecución del aplicativo deja como salida el tiempo que tardó ejecutando distintas partes del render. Estas son, el tiempo que tardó en parsear el archivo, el tiempo que se demoró armando un mapa con los haces de luz y sus reflexiones y refracciones (Photon Map Time). Luego el tiempo que lleva generar la imagen en sí (Render Time) y por último el tiempo total de ejecución. Se verá que este último dato pierde mucho valor ya que el Render Time es en general mucho mayor que los anteriores, por lo que es él quien define el tiempo total.

En las figuras 20, 21, 22 y 23 se presentan los datos de estas pruebas en forma gráfica.

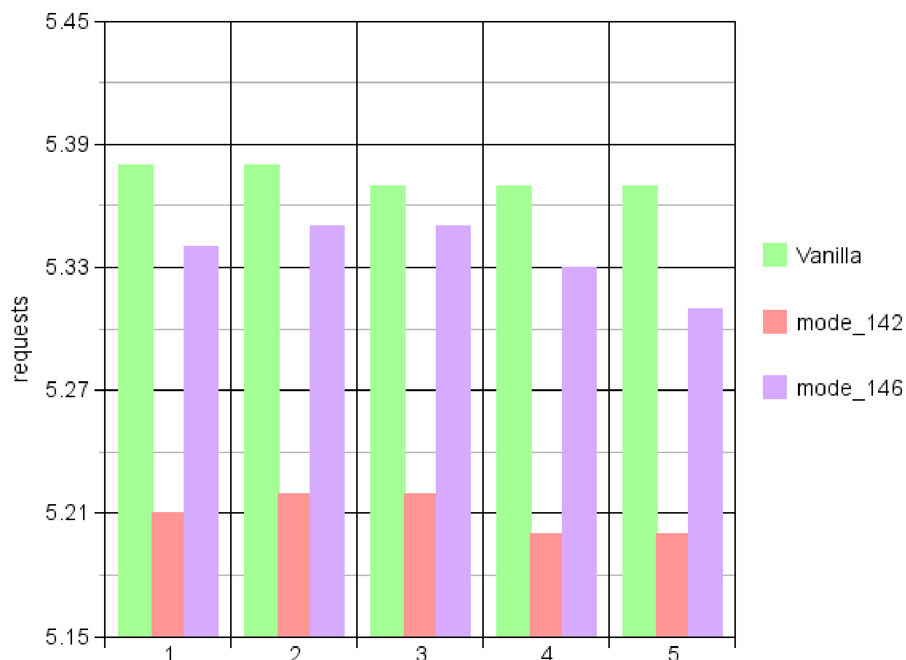


FIGURA 18. Pedidos por segundo

De estas pruebas podemos observar que en una aplicación que es intensiva en el uso de CPU como POV-Ray, el efecto del demonio es bastante mayor que en otras aplicaciones, en este caso el sistema en modo 146 fue aproximadamente un 43 % más lento que el kernel vainilla y en modo 142 un 51 %.

Al finalizar estas pruebas se vio que el consumo de memoria durante las mismas fue mínimo, solamente unos 5MB. La opción de benchmark de esta utilidad no está diseñada para testear el rendimiento del sistema, la misma sólo está pensada para testear el comportamiento de la CPU, por esto se da esa baja utilización de memoria, lo que es inconveniente para el proyecto. Por este motivo, se decidió realizar las pruebas generando una imagen del Hall of Fame de POV-Ray, la misma es la representación de una oficina, por Jaime Vives Piqueres[**34**]. Al generar esta imagen el consumo no fue mucho mayor, unos 55MB, pero sí se vio un cambio en el comportamiento del test. En las figuras 24, 25 y 26 se presentan los resultados de estas pruebas. Este modelo no generó mapa de haces de luz, por lo que los tiempos siempre fueron 0 y no se presentan.

A partir de los datos de estas pruebas se puede apreciar un comportamiento distinto. En las mismas, el tiempo de parseo para la versión del kernel modificada y ejecutando el módulo en modo 142 aumenta considerablemente, y ahora sí se nota una diferencia entre el tiempo de Render y el Total. En particular tenemos que

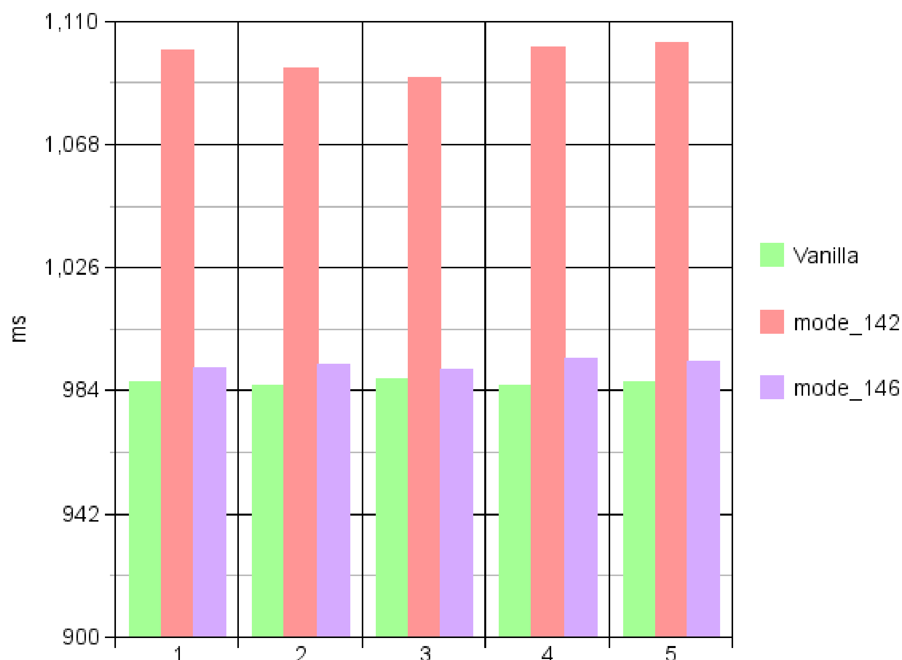


FIGURA 19. Tiempo para estar en el 80 % más rapido

para el demonio en modo 146 el render se comporta como en el caso anterior, pero el Render y el Total son aproximadamente un 31 % más lentos que el kernel sin modificar, y el mismo en modo 142 indica que el parser es 9 veces más lento. El Render un 35 % más lento y el Total un 38 %.

3.10.4. Conclusiones. Una de las primeras conclusiones a la que se llegó luego de estas pruebas, fue que es necesario realizar algunas optimizaciones al método de búsqueda en los procesos registrados. Intuitivamente no se esperaban diferencias en la performance con respecto al modo en el cual se cubren todos los frames del sistema, pero se comprobó lo contrario. Al estudiar más detalladamente el código del demonio, se observo que en el modo que cubre toda la memoria, se revisa un frame y se entrega el procesador. En el modo que se cubren sólo los procesos registrados se exploran todos los frames de todos los procesos de esa lista y solo después de esto se entrega el procesador. Dado que el demonio mantiene un `read_lock` sobre la lista de tareas registradas mientras revisa la misma, el thread no puede ser replanificado para que ejecute otro proceso. Esto implica que cada vez que el planificador le da el control al demonio, el mismo realiza una considerable cantidad de trabajo y demora las ejecuciones de otros procesos. Dado que por las restricciones mencionadas, la optimización del método no es trivial, la misma quedo como trabajo a futuro.

La segunda conclusión es que en ambientes con procesos que son intensivos en el uso de CPU, la presencia del demonio se vuelve más notoria y tiene mayor

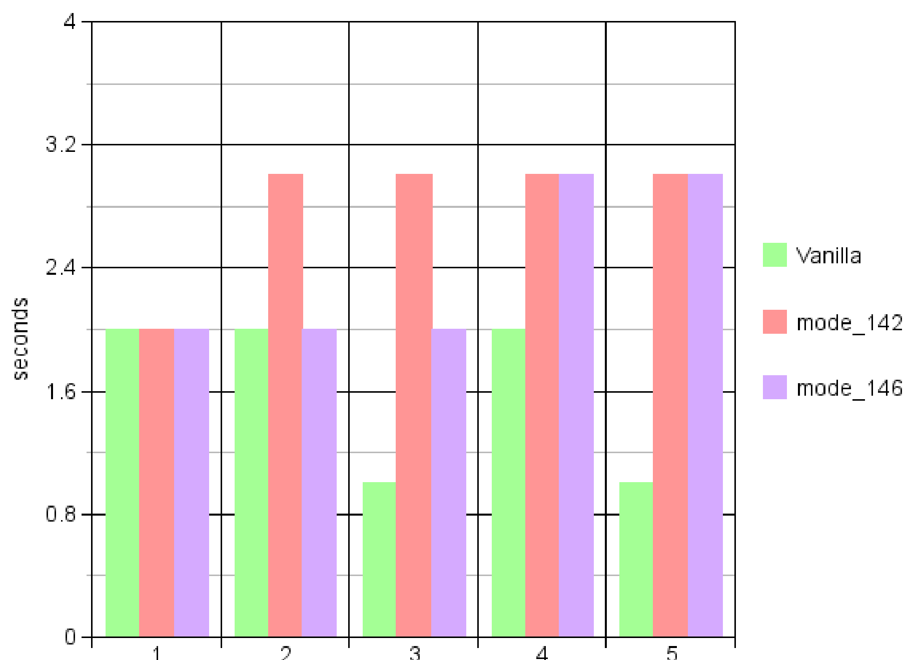


Figura 20. Tiempo de Parseo

impacto en la performance global. Esto se debe a que el mismo también consume muchos ciclos del CPU. Además, se puede concluir que cuanto más orientado a ser limitado por el acceso a IO sea el sistema, mejor comportamiento va a mostrar, en comparación a aquellos que realizan mayoritariamente operaciones de CPU.

3.11. Proceso de Desarrollo

Existen una serie de factores que agregaron dificultad al desarrollo del prototipo. Los mismos están relacionados principalmente a que se debió modificar un sistema existente de gran porte y gran complejidad, el cual fue necesario investigar y comprender previamente. Sumado a esto, el mismo se trata de un sistema operativo, por lo que el contexto en el que se trabajó no fue desde la perspectiva habitual de una aplicación de usuarios. En ese contexto se delegan muchas tareas al sistema operativo a través de bibliotecas de usuario de alto nivel. Sin embargo, en el contexto del kernel, es con las APIs de bajo nivel del mismo, con las que se debe interactuar. Las dificultades mencionadas y el conocimiento previo de las mismas, hizo que la definición del alcance final del producto estuviera sujeta a la evolución del proyecto.

Dadas las dificultades mencionadas y el grado de incertidumbre sobre el cubrimiento del alcance total que se podría lograr, se decidió seguir un proceso de desarrollo iterativo e incremental, fuertemente apoyado en la prototipación como mecanismo de validación de lo investigado y mitigación de riesgos. En todo proceso de estas características se busca dividir el trabajo en etapas, para transformar

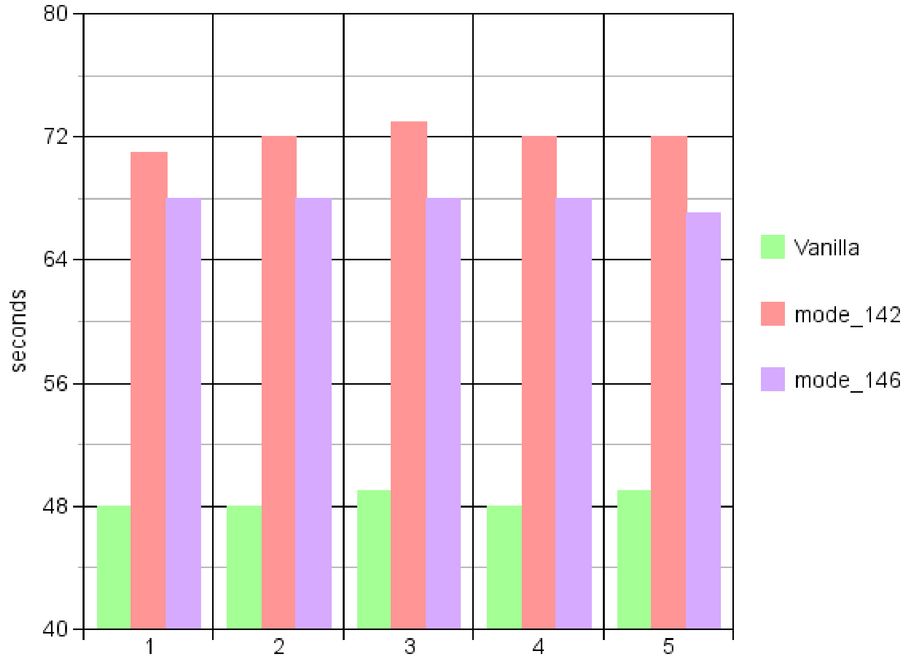


FIGURA 21. Tiempo de etapa Photon

cada una en un subproyecto de menor porte y/o complejidad. Cada iteración es un paso dentro del cronograma del proyecto y realiza un incremento en el cubrimiento del alcance del mismo. Para aumentar la efectividad de las iteraciones, las mismas fueron controladas, es decir, se fueron fijando objetivos claros para cada una. Como contraparte, nunca se logró una buena estimación de los plazos para éstas. Los incrementos fueron en un principio solución de subproblemas aislados, implicando que los primeros prototipos no fueran versiones iniciales del producto final, sino que se pueden ver más como prototipos de validación de lo investigado sobre el funcionamiento de algunos mecanismos de Linux y mitigación de algunos riesgos identificados. Las últimas etapas sirvieron para lograr primero una versión del componente con las funcionalidades básicas que se fijaron como mínimas para el prototipo y finalmente agregar algunas funcionalidades complementarias que permitieron redondear el producto y lograr un alcance de acuerdo a los esperado inicialmente.

Se realizaron cuatro iteraciones o etapas bien marcadas. Se dedicará un apartado a la descripción clara de los objetivos y resultados de cada una de ellas. Luego de esto, en la sección 3.11.5 se discutirá el cronograma efectivo de todo el proyecto, lo que permitirá ubicar las iteraciones descriptas en el tiempo.

3.11.1. Primera Iteración. Esta iteración se denominó Prototipo Etapa 1. Se entendió que como primer paso era necesario determinar de forma exacta y verificable los puntos claves de los mecanismos de manejo de memoria de Linux

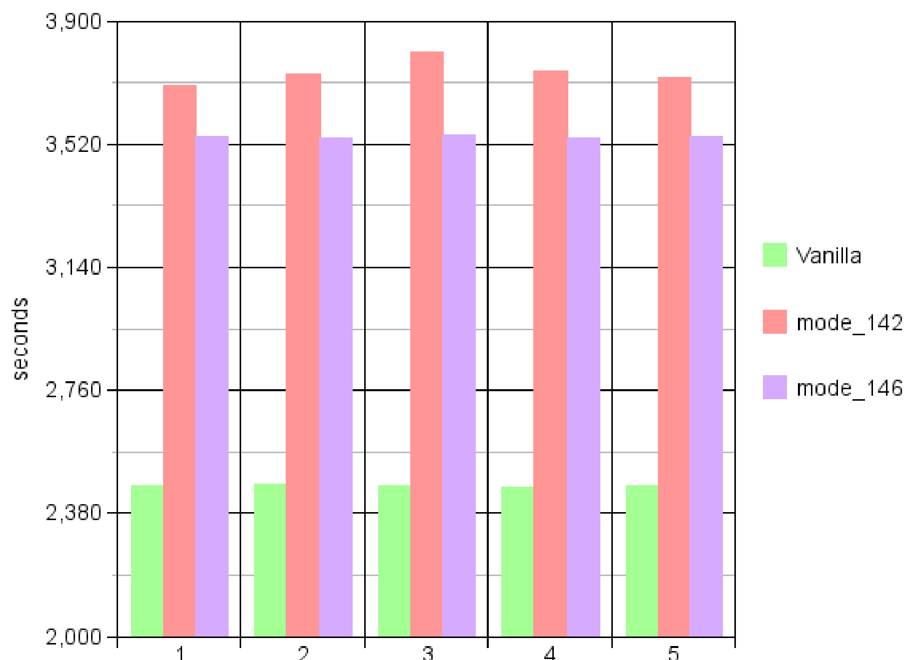


Figura 22. Tiempo de Render

investigados, mediante los cuales se asignan y reclaman frames físicos a un espacio de memoria de proceso de usuario. Una primera aproximación de esto sería mantener un contador durante el ciclo de vida de un espacio de memoria de proceso de usuario que almacene la cantidad exacta de frames que tiene asignados a cada momento. Se plantearon los siguientes objetivos para esta iteración:

- Mantener internamente un contador de frames asignados a un espacio de memoria.
- Hacer accesible dicha información al contexto de usuario mediante algún mecanismo básico de Linux.
- Hacer un prototipo simple y poco costoso que cumpla con los requerimientos anteriores.

Se logró un prototipo que cumpliera con lo planteado y el producto más importante del mismo fueron todos los puntos del mecanismo de manejo de memoria del Linux kernel donde ocurren eventos de altas y bajas de mapeos entre procesos de usuario y frames de memoria vistos en la sección 3.7.2. También se obtuvieron algunas estadísticas interesantes sobre la cantidad de páginas mapeadas a frames y porcentaje sobre el total de las áreas de memoria que tiene un proceso en ejecución. Se estudiará estos resultados en la sección 4.1.

3.11.2. Segunda Iteración. Esta iteración se denominó Prototipo Etapa 1.5, ya que se trata de un incremento sobre el prototipo de la Etapa 1. En esta etapa se propone hacer una iteración más para completar dicho prototipo agregando al

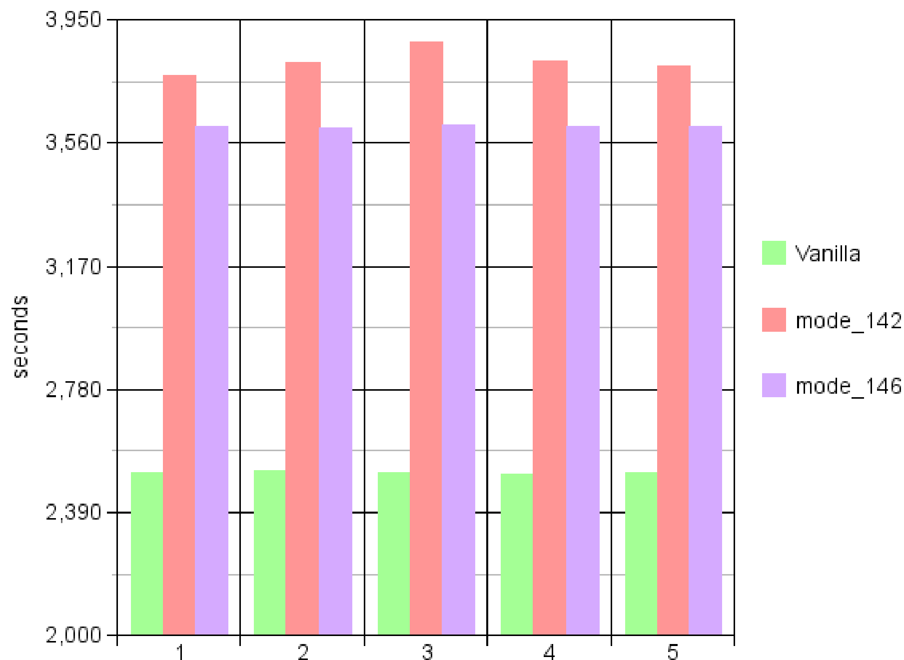


FIGURA 23. Tiempo total

mismo la capacidad de diferenciar los permisos con los que se realizan los mapeos contabilizados por el mismo. Como objetivo complementario se propuso prototipar la aplicación de mecanismos de detección de errores en bits a las páginas contenidas en los frames de memoria física del sistema. Así, los objetivos se plantearon de la siguiente manera:

- Contabilizar la cantidad de frames de memoria física asignados con permisos de sólo lectura a los procesos de usuario.
- Detectar cambios en bits de una página contenida en un frame de memoria física asignado a un proceso de usuario.

Se cumplió lo planteado y como principal salida se logró determinar en qué casos los procesos mapean los frames con permisos de sólo lectura. Además, ese prototipo señaló los puntos donde pueden cambiar los permisos del mapeo y dio como salida independiente un método verificado de detección de cambios en bits sobre frames de memoria. También en este caso se obtuvieron estadísticas, en particular sobre la cantidad de frames de sólo lectura que mapea cada proceso. Se estudiarán las mismas en la sección 4.1.

3.11.3. Tercera Iteración. La tercera iteración se denominó Prototipo Etapa 2. Tomando como entrada los resultados anteriores se entendió que estaban dadas las condiciones para hacer una primer versión del prototipo final, que agregue al Linux kernel mecanismos de detección de cambios de bits en los frames de memoria

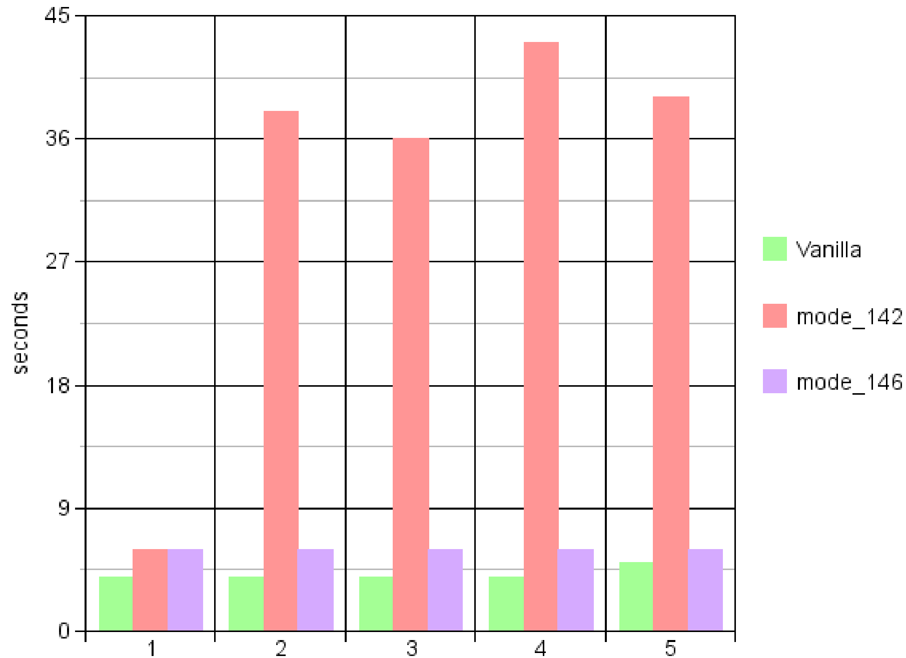


FIGURA 24. Tiempo de parseo

física para las páginas de sólo lectura de los procesos del espacio de usuarios. Los objetivos se detallan a continuación:

- Detectar soft errors en el espacio de memoria de los procesos de usuario.
- Desplegar detalles sobre la detección de dichos errores mediante alguna interfaz con el espacio de usuario.

El prototipo resultante de esta iteración cumplía con los objetivos planteados mediante un único mecanismo de búsqueda de errores implementado en el demonio ya descrito. Un resultado importante que dejó el mismo, fueron las primeras pruebas de performance del estilo visto en la sección 3.10, las cuales demostraron que las herramientas desarrolladas no implicaban una gran reducción en la performance del sistema.

3.11.4. Cuarta iteración. Esta cuarta y última iteración, se identificó como Prototipo Etapa 3 y tiene como salida la versión completa del módulo que se planteó desarrollar en el Linux kernel, el cual se describe a lo largo del capítulo 3 de este documento. En esta etapa se agregaron a las funcionalidades básicas de detección de errores sobre el conjunto de páginas de sólo lectura, nuevos mecanismos de detección y corrección de errores, así como también funcionalidades de rejuvenecimiento a nivel del sistema operativo y del espacio de usuario. Se completó además, una interfaz con el espacio de usuarios que permita la interacción de los distintos usuarios con el módulo y permite sacar provecho a las funcionalidades que el mismo brinda. Los objetivos planteados fueron los siguientes:

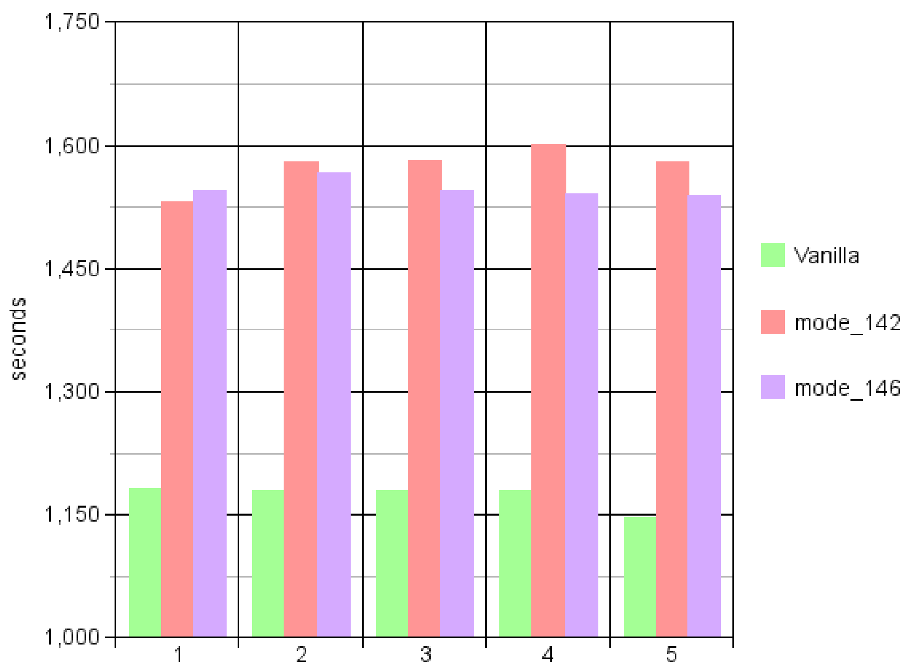


FIGURA 25. Tiempo de render

- Revisar el mecanismo de búsqueda de errores y agregar mecanismos y/o estrategias alternativas.
- Aplicar técnicas de rejuvenecimiento a nivel de Sistema Operativo para resolver los errores de memoria detectados por el prototipo.
- Adecuar la interfaz del módulo con el espacio de usuarios, para brindar interacción en ambos sentidos. Configuración en caliente del módulo desde espacio de usuario y métodos sincrónicos y asincrónicos de envío de eventos e información desde el módulo a los procesos.

Los resultados de esta etapa son tratados a lo largo de todo el documento.

3.11.5. Cronograma Efectivo. Se hará aquí una breve discusión del cronograma efectivo del proyecto. La figura 27 ilustra el mismo.

Al comienzo del proyecto se propuso un cronograma tentativo bastante optimista, con los siguientes puntos:

- **Abril-Junio**
  Relevamiento del estado del arte y elaboración de taxonomía de errores.
  Comienzo de prototipado/desarrollo de técnicas identificadas de forma temprana.
  Documentación de las tareas realizadas.
- **Julio-Setiembre**
  Prototipado/desarrollo de técnicas seleccionadas.

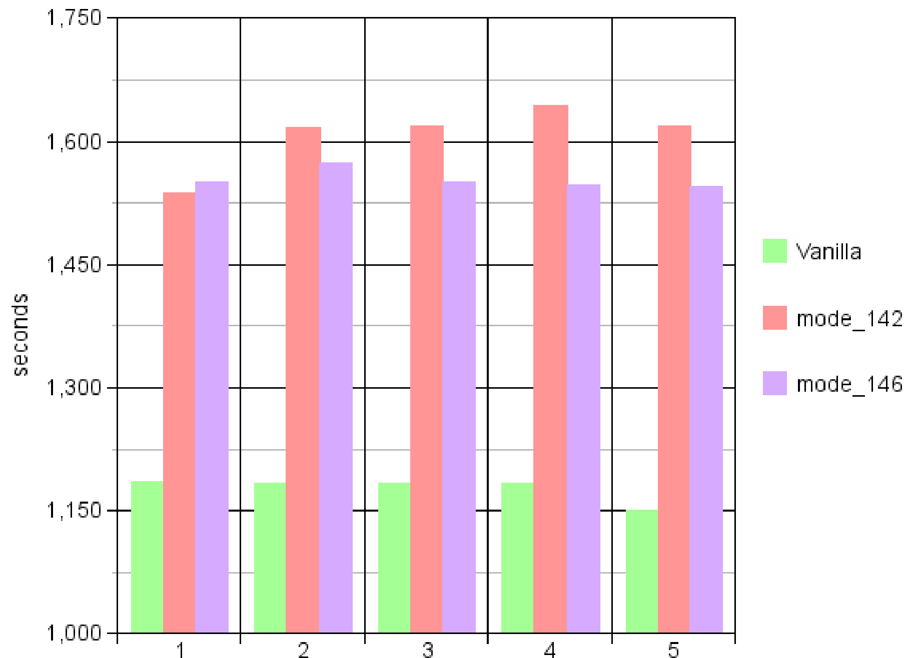


FIGURA 26. Tiempo total

Mediciones de su impacto en performance en el sistema.
Documentación de las tareas realizadas.

- **Octubre-Diciembre**
  Correcciones finales en los modelos/desarrollos.
  Documentación final del proyecto.

Luego de los primeros meses dedicados a investigar el estado del arte de software aging y el sistema de administración de memoria de Linux, tareas que tuvieron una duración de acuerdo a la propuesta, se comenzó a trabajar más fuerte en las primeras etapas del prototipo. Al inicio de cada iteración se fijo sus objetivos y se intentó estimar una fecha de fin de la misma. En general, durante el desarrollo del prototipo, la mayoría de las estimaciones realizadas por el equipo fueron demasiado optimistas en comparación con las fechas efectivas. Se atribuye esto a la falta de experiencia previa de los autores realizando un desarrollo de este porte en el contexto del kernel de Linux.

Se estimó que las dos primeras iteraciones se lograrían en aproximadamente un mes y medio, pero como se puede observar las mismas requirieron un total de tres meses, debido a que estos prototipos implicaban un alto grado de investigación del funcionamiento de Linux y revisión de su código fuente, profundizando los conceptos generales adquiridos durante el estado del arte. Al comienzo del tercer prototipo, el cronograma tentativo ya había perdido validez y se estimó que dicho prototipo se llevaría a cabo en un máximo de dos meses, tomando en cuenta los riesgos mitigados por las primeras etapas. Como se puede observar en la figura 27 el prototipo 2,

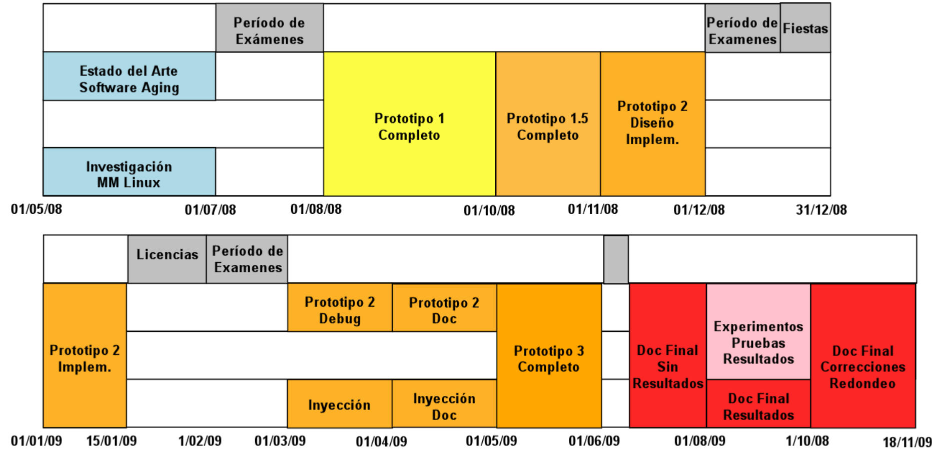


FIGURA 27. Cronograma Efectivo

correspondiente a la tercer iteración y a la primer versión del producto final, requirió una dedicación efectiva de aproximadamente cuatro meses, sin contar varias pausas realizadas por distintas razones. El retraso de este prototipo se debió a errores que provocaban su inestabilidad y llevó mucho tiempo de investigación y pruebas para comprender su naturaleza y encontrarles solución. Luego de superados estos problemas y lograda una versión estable, la última etapa del prototipo logró en un período menor, completar toda la funcionalidad planeada originalmente. Esta etapa se decidió realizar porque se confiaba en poder cumplir con el tiempo estimado para la misma en base a la experiencia adquirida, lo cual se logró, y se entendió que era indispensable para completar una versión de un producto con real utilidad práctica y no sólo las bases para la mismo. De todas formas, esta versión fue sometida a pruebas de carga, performance y algunos experimentos durante etapas siguientes, las cuales llevaron a varias correcciones, para lograr su mayor estabilidad al finalizar el proyecto.

Si bien se elaboraron documentos intermedios para el estado del arte y para cada prototipo, los cuales sirvieron como base para la elaboración de este trabajo, esta última tarea también fue muy demandante y requirió mucho tiempo. En particular fueron necesarias dos etapas de revisiones al mismo que permitieron llegar a la versión final que aquí se presenta.

Capítulo 4

Resultados

Hasta ahora este documento se dedicó a introducir el área de software aging y técnicas relacionadas, para luego hacer una extensa descripción del prototipo desarrollado. Una vez completo el mismo, fue posible llevar a cabo algunos experimentos con éste. Se documentan en este capítulo las pruebas realizadas y se analizan los resultados obtenidos a partir de las mismas, llegando en algunos casos a datos muy interesantes que fundamentan la utilidad práctica del producto. Un ejemplo de esto se verá en la sección 4.1, donde se estudió el porcentaje de la memoria física cubierto y el grado de incidencia de la misma en el funcionamiento del espacio de usuarios. La sección 4.2 por su parte, se dedica a analizar el grado de éxito obtenido en la puesta en producción del sistema en diferentes equipos de hardware. Luego, en la sección 4.3, se estudiará y comparará la eficacia de las distintas estrategias de búsqueda. Finalmente en la sección 4.4 se analizará un caso de estudio de simulación de fallos en sistemas de alta disponibilidad ejecutando en un kernel compilado con el módulo PGSA.

4.1. Frames de sólo lectura

Un producto importante de la segunda iteración del prototipo fue la capacidad de determinar la cantidad de frames que tiene asignados una tarea, con permisos de sólo lectura, además del número total de frames mapeado a la misma, ya conocido en la primera iteración. Por otra parte, la versión final del producto permite conocer en todo momento el tamaño en frames, del conjunto de sólo lectura de todo el sistema. Con estos números, fue posible tomar algunas mediciones que permitieran cuantificar de alguna forma el grado de cubrimiento de la memoria física realizado por el prototipo y la incidencia de la porción cubierta en el funcionamiento del espacio de usuarios.

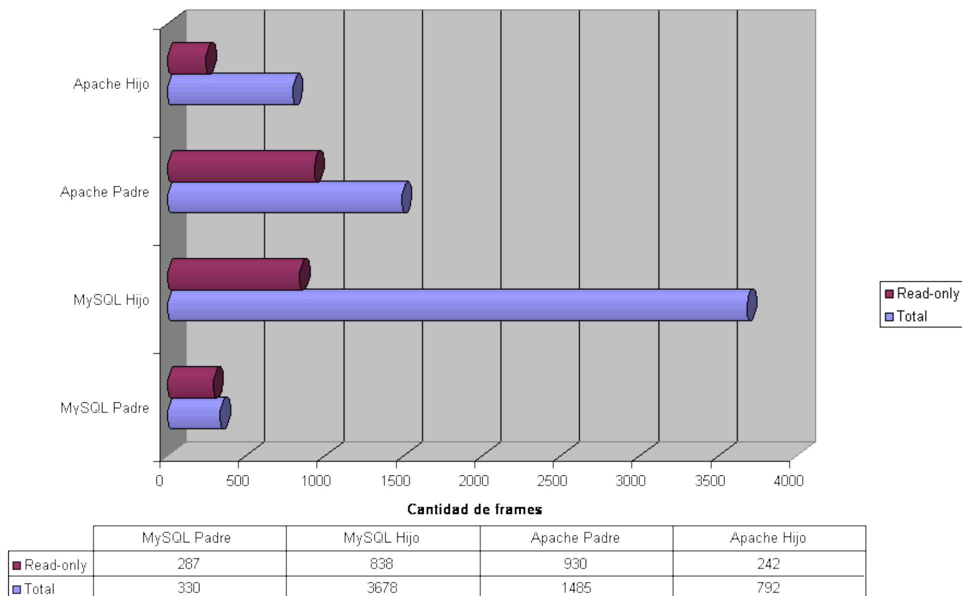
El primer experimento es bastante simple y toma en cuenta el indicador de la cantidad de páginas de sólo lectura verificadas en cada iteración de búsqueda de errores con el método basado en el demonio de kernel y la estrategia que recorre todos los frames utilizables del sistema. Para este experimento se utilizó un equipo con arquitectura Intel 32bits, con 1GB de RAM y sistema operativo OpenSuse 11.0 [**25**] ejecutando un kernel modificado para utilizar la versión final del módulo PGSA. Para simular carga en el sistema se ejecutó el test que utiliza Apache[**22**], MySQL [**23**] y Wordpress[**26**] creado para las pruebas de performance en la sección 3.10. Se tomó esta decisión, pues se entendió que era un buen ejemplo de software de alta disponibilidad candidato a ser usuario del módulo desarrollado. Revisando la cantidad de verificaciones por iteración mientras se ejecutaba el test, se obtuvo un valor cercano a los 17000 frames chequeados (con páginas de 4KB serían unos 67MB). Tomando en cuenta que el rango de frames utilizables para reserva dinámica de memoria contiene para esta configuración alrededor de 225.234 frames (880MB),

una primer observación podría ser que se cubre un rango pequeño del total de memoria. Sin embargo, hay una serie de consideraciones a tener en cuenta. Primero, dentro del total de frames de memoria para reserva dinámica hay un subconjunto que se encuentran libres, revisando en `/proc/vmstat` podemos ver que ese número desciende a 2621frames(10,2MB). Por otra parte, están los frames que son utilizados por el kernel para estructuras dinámicas y tablas de páginas, los cuales se encuentran fuera del alcance del prototipo. Como se ha mencionado, el kernel tiene una política de consumir memoria sin restricciones hasta que la misma se agota y por tanto ese número puede llegar a ser bastante grande. De esta forma, la fracción del total de frames verificada indica un bajo cubrimiento de la memoria física del sistema, pero se considera que no es un número útil para determinar el cubrimiento del espacio de procesos de usuario.

Los indicadores tomados en cuenta para el segundo experimento, fueron el porcentaje de páginas virtuales del espacio del proceso de usuario que se encuentran alojadas en frames de la memoria física y el porcentaje de este conjunto que sólo son accedidas con permisos de lectura. La versión original del prototipo 1.5 mostraba estos valores justo antes de que cada proceso terminara, en la llamada a `exit()` del mismo. Como parte de la documentación del prototipo, se incluyeron algunos muestreos tomados durante la ejecución de OpenSuse 11.0 [**25**] con el kernel modificado y se observó que en promedio el porcentaje de frames de sólo lectura mapeados por cada proceso se elevaba al 80 % del total. Para el nuevo experimento, sin embargo, se consideró que el momento en que se mostraban los valores no era un buen representante de todo el ciclo de vida de ejecución del proceso y se decidió hacer algunas modificaciones a ese prototipo para obtener mejores estadísticas. Para esto, gracias a la experiencia adquirida durante el desarrollo del producto final, se creó un demonio de kernel que imprime los porcentajes periódicamente para un grupo de tareas determinado, indicado por su *tgid*. Se hizo para un único grupo para evitar que el volumen de la información de todo el sistema hiciera compleja la interpretación de los resultados. Para indicar el grupo de tareas a monitorear se creó una llamada al sistema que lo configura. Se incluyó en el apéndice C un parche con las modificaciones al kernel mencionadas.

Se instaló en el mismo equipo de las pruebas anteriores el kernel con la variante del prototipo 1.5 y se ejecutó una vez más las pruebas de Apache [**22**] con MySQL [**23**] y Wordpress, asignando alternadamente como grupos de tareas a monitorear los del Apache y la base de datos para tomar muestras. Estos procesos son una vez más un buen representante del tipo de software de alta disponibilidad que es candidato a utilizar las funcionalidades del módulo PGSA. A continuación se presentan dos gráficas (figuras 28 y 29) mostrando las cantidades de frames de sólo lectura observadas en los procesos de MySQL y Apache antes y durante la prueba.

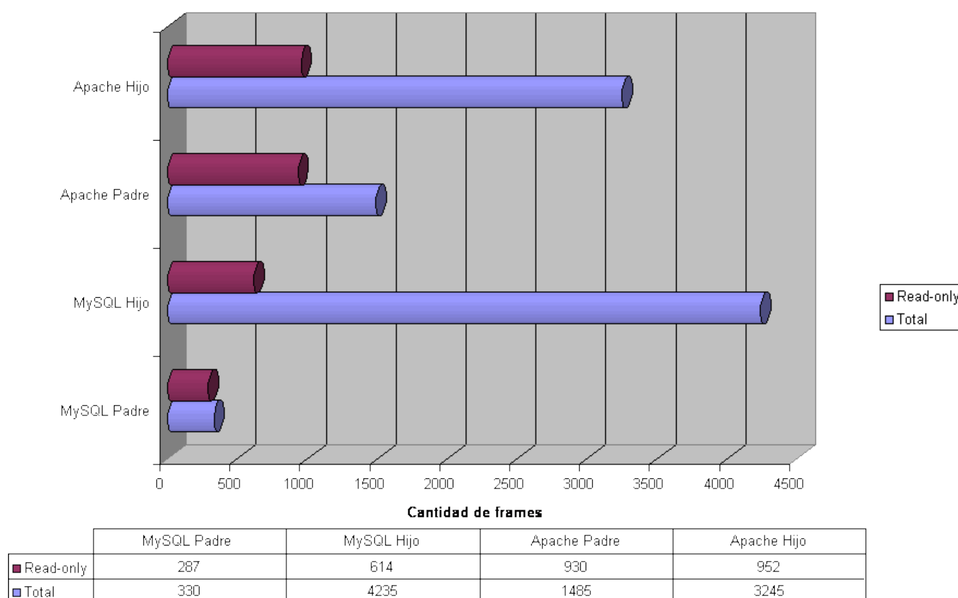
El procedimiento seguido durante la prueba, constó de levantar los servidores de MySQL y Apache e identificar los procesos creados. Se observó que MySQL crea un proceso padre y un hijo, el cual a su vez consta de varios hilos, los cuales comparten un único espacio de memoria. Para el proceso padre se observó inicialmente un total de 330 frames de memoria física de los cuales un 86 % se mapean con permisos de sólo lectura. El hijo por su parte cuenta con 3678 frames de los cuales el 22 % son de sólo lectura. En el caso de Apache, se crea un proceso



	MySQL Padre	MySQL Hijo	Apache Padre	Apache Hijo
Read-only	287	838	930	242
Total	330	3678	1485	792

FIGURA 28. Porcentaje de frames read-only por proceso antes de la prueba

padre y 5 procesos hijos con espacios de memoria independientes pero que se comportan exactamente de la misma forma, por lo que sólo se seguirá uno como representante. Para el proceso padre se observó un total de 1486 frames, con el 62 % de sólo lectura y para el hijo 792 con el 30 % de sólo lectura. Observando la composición del espacio virtual de los procesos en `/proc/<pid>/maps` se notó que estaban compuestos mayormente por bibliotecas y las áreas con permisos de escritura correspondían a mapeos de variables de éstas y el propio proceso. Se observa que en ambos casos el proceso principal tiene un alto porcentaje de frames mapeados con permisos de sólo lectura mientras que los hijos, seguramente más ligados a mantener datos temporales de la ejecución de pedidos particulares, tienen porcentajes menores, aunque no despreciables. En el global de lo observado, el cubrimiento realizado de los procesos es bastante bueno, lográndose mayores porcentajes en aquellos procesos que tienen secciones de variables de menor volumen. En el caso de los procesos hijos que muestran porcentajes menores de frames de sólo lectura, es más probable por otra parte que los datos no cubiertos correspondan a datos temporales de pedidos particulares y la ocurrencia de un error sobre los mismos no tenga efectos a largo plazo en la ejecución del servidor. De todas formas, esto son sólo suposiciones que se intentarán probar mediante otros experimentos en las próximas secciones. Por ahora sólo se puede concluir que el cubrimiento logrado disminuye en tanto más grande sea la sección de variables de los procesos. La figura 29 muestra los porcentajes observados para los mismos procesos durante la simulación de carga sobre los servidores.



	MySQL Padre	MySQL Hijo	Apache Padre	Apache Hijo
Read-only	287	614	930	952
Total	330	4235	1485	3245

FIGURA 29. Porcentaje de frames read-only durante la pruba

Continuando con el procedimiento de prueba, se ejecutó el proceso que simula carga en el servidor web, provocando además accesos a la base de datos. Luego de esto se repitieron las mediciones anteriores, obteniéndose los valores presentados. Lo primero que se observa aquí es un claro aumento en el tamaño de los conjuntos de páginas residentes en memoria para los procesos hijos, confirmando las suposiciones acerca de la naturaleza de éstos, dado que los procesos padre, sin embargo, mantuvieron la misma composición que antes. Los aumentos afectaron también a los subconjuntos de frames de sólo lectura, manteniendo porcentajes similares en el promedio del 29 % para el proceso hijo de apache, pero teniendo una modificación en el proceso hijo de MySQL donde el porcentaje bajó del 22 al 14 %. Se puede afirmar aquí que lo observado reafirma la conclusión de la disminución del porcentaje ante el aumento de datos variables temporales asociados a los pedidos.

4.2. Búsqueda de soft errors

Tal vez el resultado más interesante que se esperaba encontrar a partir de este proyecto, es la tasa de errores detectados en un ambiente de producción del prototipo. Hasta el momento no se han realizado experimentos formales y controlados, con buena planificación del ambiente, la versión del producto, las condiciones y la duración de la prueba. Las causas de esto son la falta de hardware para dedicar exclusivamente a esta tarea y los constantes cambios hechos al prototipo hasta llegar a su versión final. Los resultados que aquí se presentan se basan en "horas de vuelo" acumuladas en un equipo de pruebas utilizado en un área de desarrollo de software a la cual tienen acceso los autores. En este equipo se instaló durante un período de más de dos meses el kernel modificado en versiones estables de las funcionalidades de detección y entre las cuales los cambios hechos se situaban exclusivamente en la interfaz con el espacio de usuarios. Durante el período de producción del módulo en

dicho equipo, no se detectaron soft errors y por tanto la tasa de errores y otros datos inferibles se vuelven triviales. Se dedicará la sección 4.2.1 para buscar explicación a lo observado.

4.2.1. Probabilidad de ocurrencia de errores. Dado que se asume que el hardware utilizado durante las pruebas funciona correctamente y por tanto no ocurren hard errors en el mismo, se busco información teórica y práctica sobre la frecuencia con que ocurren soft errors a nivel del mar, para determinar si la tasa de errores observada era coherente y tener una idea aproximada de cuánto tiempo sería necesario tener el sistema en producción para que fuera probable la ocurrencia de errores de este tipo.

Una primer fuente de información al respecto, son las tasas informadas por los propios fabricantes de hardware, quienes cada vez dedican más esfuerzos en el diseño de los chips para contrarrestar los fenómenos que causan los errores aquí tratados. Estos datos no son fáciles de obtener, dado que en muchos casos los estudios y estimaciones realizados ni siquiera son publicados, pero se logró encontrar algunas notas técnicas al respecto. Los resultados encontrados muestran que las probabilidades en condiciones normales son realmente muy bajas, pero antes de dar valores específicos, es necesario hacer una pequeña introducción a las unidades con que se suele expresar la tasa de soft errors denominada SER (Soft Error Rate). A continuación se lista las más utilizadas:

- Fallas en el tiempo o failures-in-time (FIT): es equivalente a un error por billón de horas de operación del chip.
- Tiempo medio entre fallos o mean-time-between-failures (MTBF): se mide en años de operación del chip.

Remitiéndonos ahora sí a los números, se encontró una nota técnica [**29**] del fabricante Micron, uno de los líderes mundiales en provisión de soluciones avanzadas basadas en semiconductores, en particular DRAM y Flash. La misma data de Diciembre de 1999 y muestra una tabla con valores de SER expresados en MTBF para distintos módulos de DRAM de 16 y 64MB. En el peor de los casos el tiempo entre fallos reportado es de 7 años. Se obtuvieron más valores en el white paper "Soft Errors in Electronic Memory"[**30**] hallado en el sitio de Tezzaron® Semiconductor, que se especializa en módulos de memoria de alta velocidad. El mismo fue publicado en enero de 2004 y presenta valores recolectados de distintos artículos (incluyendo la nota de Micron mencionada). Allí se habla de SER de 1,000 FIT en el caso típico y de unos pocos cientos a unos pocos miles FIT en memorias de alta velocidad en pruebas a nivel del mar. En el mismo también se menciona que el SER aumenta por 5 a 800mts de altura, por 10 a 1600mts y por 14 a 3000mts. También menciona 200 FIT como meta en nuevos productos para el fabricante Cypress, aunque no se encontró información actual para verificar si tuvo éxito. Resumiendo, todas las tasas encontradas hablan de la posibilidad de que transcurran años antes de obtener un fallo, por lo que son coherentes con la experiencia del proyecto.

Existen otras fuentes de datos de carácter más práctico que las anteriores. Una de ellas son los logs de memorias ECC. En ese sentido, se obtuvo en Beowulf.org [**28**] algunos datos interesantes. Beowulf.org es una lista de correo para usuarios y diseñadores de clusters de computadoras de clase Beowulf, los cuales se caracterizan entre otras cosas por utilizar infraestructura de software open source, normalmente Linux. Algunos de los usuarios de la lista, cuentan con clusters de gran porte lo que

les permitió hacer buenos aportes al proyecto. Las experiencias prácticas hablan de varios meses sin detección de errores en clusters con miles de nodos al nivel del mar en un caso y tasas de error nulas en 5 terabytes en las mismas condiciones en otro. Se observó además un acuerdo general en que tasas de errores mayores se debían a defectos del hardware y el reemplazo de las piezas afectadas volvía los valores a la norma mencionada. Un usuario particular aportó datos actuales del fabricante Micron que no se habían podido encontrar, los cuales indicaban alrededor de 100 FIT. Siguiendo su razonamiento, en un equipo con 4GB de RAM se tienen 32 Gbits y con la tasa de Micron de 100 errores cada billón de horas de operación del bit, se tienen 5000 errores cada billón de horas, es decir 5 errores cada millón horas. Esto implica un error cada 200,000 horas o aproximadamente 22 años. Visto desde otro punto de vista, también se podría tener 1 error por hora cada 200,000 equipos. Además, dicho usuario referencia un artículo reciente (Abril de 2008), en el que se comentan experimentos realizados en instalaciones de testing donde se simularon los efectos de rayos cósmicos y se midieron y escalaron tasas en chips modernos, arrojando salidas entre 100 y 200 FIT, lo cual concuerda con los valores de Micron.

Se puede concluir entonces, que observando un único equipo, es probable pasar hasta varios años sin detectar errores al nivel del mar. El primer experimento realizado en la sección 4.1, mostró que en condiciones de carga en un equipo con un total de 1GB de memoria, el prototipo cubre aproximadamente 67MB de la misma. Esto implica que las probabilidades de ocurrencia de un error dentro del área cubierta, son poco más de un 6.5 % de las esperadas para el módulo completo. De acuerdo a los datos presentados, los resultados obtenidos en la puesta en producción del prototipo en un único equipo y durante un período de tan sólo meses, no solo son factibles, sino los más probables.

4.3. Retraso de detección

En la sección 3.7.6 se introdujo el concepto de retraso de detección o *RD* para explicar el diseño de las estrategias de búsqueda de errores utilizadas. La teoría presentada, sostenía que la estrategia del demonio que verifica sólo procesos registrados tendría un menor *RD* promedio, que la estrategia que itera sobre todo el conjunto de sólo lectura. En el siguiente experimento se buscará verificar esto con datos prácticos.

Para determinar el retraso de detección, es necesario conocer los instantes exactos de ocurrencia y detección del error. Para conocer el primer dato se deberá apelar una vez más, al mecanismo de inyección de errores desarrollado, mientras que la opción de notificación sincrónica de errores mediante señales dará a conocer el valor restante. Para implementar la prueba se desarrolló un programa que se registra como agente del proceso que se le indique e inyecta un error en el mismo, para luego imprimir el milisegundo exacto de la inyección y de la recepción de la señal. Para que esto funcione como se espera, se debe habilitar la flag *PGSA_SIGNAL* en todos los casos. Se incluyeron los fuentes del programa utilizado en el apéndice D.

Se tomaron varias muestras del *RD* con las dos estrategias mencionadas. La primera prueba se realizó con un simple programa "Hola Mundo", como el utilizado para testear el mecanismo de inyección. Se ejecutó el agente varias veces para una instancia de dicho programa con la configuración *PGSA_CHECK_DAEMON + PGSA_CHECK_ALL*, obteniéndose un *RD* promedio de 275 milisegundos. Se

repitió la misma prueba con la flag *PGSA_CHECK_ALL* desactivada y en este caso el *RD* promedio fue de apenas 1 ms, confirmando lo esperado.

Dado que en la prueba anterior el único proceso registrado era el “Hola Mundo” y esto probablemente influyó en las diferencias observadas, se decidió simular mayor carga de procesos registrados en el sistema, en un intento por lograr un contexto más real. Para lograr esto se recurrió una vez más al test utilizado para las pruebas de performance realizadas con Apache [**22**] y MySQL [**23**]. Se ejecutaron los procesos necesarios, se registraron los mismos para que fueran verificados en todos los casos y se ejecutó el test de performance para simular carga. En este ambiente, se tomaron nuevas muestras para el proceso helloWorld con las mismas configuraciones que antes. En este caso, el RD promedio observado con la flag *PGSA_CHECK_ALL* habilitada fue de 280 ms, mientras que con la otra estrategia el mismo fue una vez más de apenas 1 ms.

En resumen, todas las muestras tomadas confirmaron el comportamiento intuido del RD para las distintas estrategias de búsqueda analizadas.

4.4. Efecto en la disponibilidad

El último experimento que aquí se presentará, intenta determinar los beneficios en sistemas de alta disponibilidad de las herramientas para combatir el software aging diseñadas y prototipadas. Lo que es posible preguntarse al respecto, es en qué grado pueden mejorar la disponibilidad de un sistema de ese tipo, las funcionalidades de detección y corrección de errores en 1 bit para los frames de sólo lectura. En la sección 4.1 se estudió ejemplos de servidores de alta disponibilidad como Apache [**22**] y MySQL [**23**] y se observó que los porcentajes de frames de memoria física de sólo lectura que se asignan a sus procesos durante la simulación de carga son bastante interesantes. Por otra parte, observando la salida del archivo *maps* en las estadísticas por proceso presentadas en el sistema de archivos `/proc` se vio que la composición de áreas virtuales de dichos procesos muestra que más de un 80 % de las mismas tienen permisos de acceso de sólo lectura. Estos datos llevan a pensar, que la ocurrencia de errores en la memoria física que mapea las áreas de sólo lectura de estos procesos, podría tener graves efectos en la disponibilidad de los mismos y por tanto la oportuna corrección de estos errores, podría ser de gran importancia. Con la motivación de estos datos y teorías, se decidió realizar el experimento que aquí se describe.

El experimento busca básicamente determinar qué efectos tiene en la disponibilidad de un servidor la ocurrencia de errores en sus frames de sólo lectura con y sin la utilización de las herramientas brindadas por el módulo PGSA. Para simular este comportamiento se recurrió una vez más al mecanismo de inyección de errores desarrollado y al test de carga con Apache[**22**], PHP[**24**], MySQL[**23**] y Wordpress[**26**] ya utilizado en experimentos y pruebas anteriores. Para hacer más reales las condiciones, se decidió que los errores debían ocurrir en direcciones aleatorias de la porción de sólo lectura del espacio de alguno de estos procesos. Previendo que la ocurrencia de un único error de estas características podría no ser suficiente para llegar a observar efectos visibles en el proceso, se decidió simular una “lluvia de rayos cósmicos” sobre los frames de sólo lectura de su espacio de memoria. Dado que con el utilitario para inyección utilizado hasta ahora, ésta simulación sería difícil de llevar a cabo, se desarrolló un nuevo utilitario denominado `cosmicrayn`, al cual se le indica un proceso y un conjunto de áreas virtuales del mismo y éste es capaz de

inyectar un error en un bit aleatorio de una dirección aleatoria dentro de una de las áreas indicadas, seleccionada también de forma aleatoria. La utilidad repite este proceso cada un periodo de tiempo seleccionado, en este caso, cada un segundo. Se incluye el código fuente de la utilidad en el apéndice E. Es importante que el error inyectado sea de tan sólo 1 bit, ya que esto permite que se encuentre dentro de la distancia del código corrector implementado en el módulo. Para garantizar que se cambia un único bit en el byte a inyectar seleccionado de forma aleatoria y sin conocer su valor previo, se debió hacer una pequeña modificación a la system call de inyección de errores para que aceptara una máscara indicando el bit a dar vuelta, en lugar de un nuevo valor para todo el byte como recibía en su versión anterior.

Se realizó el experimento con el modo 146 en el cual se habilita la detección y reporte de errores por medio del demonio, pero no se hace corrección de los mismos. Se puso en ejecución el test de carga y se simuló una sucesión de errores para el espacio de memoria de sólo lectura de uno de los subprocesos del servidor Apache, por medio de `cosmicrayn`. No fue necesario más de una decena de inyecciones para observar un freno en el test y un comportamiento anormal en el Apache. Sus procesos hijos comenzaron a morir de forma continua, a pesar de los esfuerzos de éste por crear nuevos procesos para sustituirlos. Los errores inyectados fueron reportados por varios hijos a la vez antes de comenzar este comportamiento. Evidentemente se debió a la inyección de errores en código de bibliotecas que son compartidas por todos estos procesos. Como la imagen en memoria física de las páginas de archivos no es sincronizada al disco, a menos que las mismas sean modificadas por las aplicaciones, el frame inyectado seguía siendo utilizado por los nuevos procesos, que continuaban muriendo sin satisfacer los pedidos del test. Se pudo comprobar esto al observar el log del apache, donde se mostraba un mensaje de error originado en la biblioteca de extensiones de PHP, el cual indicaba que se intentó acceder una dirección de memoria fuera del espacio del proceso.

Se encontró aquí un buen ejemplo de la vulnerabilidad de un sistema con requerimientos de alta disponibilidad, ante la ocurrencia de un simple error de un bit en una de sus páginas de código de sólo lectura. La prueba se repitió con el modo 156 del módulo PGSA seleccionado, el cual es análogo al anterior, pero con la corrección por Hamming habilitada. Como era de esperarse, los errores inyectados por `cosmicrayn` se reportaron como corregidos y los mismos no tuvieron efecto aparente en la ejecución del test o los procesos. Incluso si el retraso de detección hubiera permitido la muerte de algún subproceso, el error hubiera estado solucionado para cuando Apache creara un nuevo hijo para sustituirlo y no se entraría en el estado observado anteriormente.

Como conclusión se puede ver aquí que las herramientas desarrolladas pueden llegar a hacer una diferencia importante en la disponibilidad de sistemas que ejecutan en hardware vulnerable a la ocurrencia de soft errors.

Capítulo 5

Conclusiones y Trabajo a futuro

En este capítulo se resumirán las conclusiones obtenidas del proyecto llevado a cabo y los productos y resultados derivados del mismo. Esto se verá en la sección 5.1.

Además, en la sección 5.2 se detallará el trabajo a futuro que se entiende hay para hacer, tanto para completar y mejorar lo hecho, como para continuar avanzando con el estudio de soluciones a los problemas vistos.

5.1. Conclusiones

Analizando los resultados del proyecto, se puede concluir antes que nada que se cumplieron los objetivos planteados originalmente y se logró el alcance deseado, abarcando incluso algunas herramientas de corrección de errores que formaban una parte opcional del mismo. Lo costoso de la investigación y aprendizaje constante que se requirió, especialmente sobre Linux, fue un factor que extendió la duración de algunas tareas y dilató en general el cronograma, pero brindó un importante crecimiento, formación y experiencia a los miembros del grupo.

Un factor clave para el exitoso desarrollo del prototipo, fue la división del mismo en etapas incrementales. Esta estrategia permitió realizar prototipos que validaran la investigación en etapas tempranas. Además, más allá del tiempo que insumió, gracias a que se logró estabilizar el Prototipo 2 con las funcionalidades básicas de detección y a los conocimientos y experiencia adquiridos en esta etapa, el desarrollo de las restantes funcionalidades incluidas en la versión final se pudo llevar a una versión bastante estable y en un período de tiempo mucho menor.

Uno de los factores que agregan mayor valor al prototipo logrado, es haber hecho un aporte a un área particular del problema de envejecimiento de software, la que como se mencionó en la introducción de este trabajo, tiene muchos contextos de aplicación práctica en la actualidad y para la cual el mercado no ofrece soluciones por software.

5.1.1. Prototipo. El prototipo desarrollado implementa completamente las herramientas definidas y el grado de verificación del mismo y su funcionamiento en general son buenos. Se destaca a continuación algunas de las características más importantes de la solución.

- **Modularización:** Por su naturaleza, las extensiones al kernel desarrolladas, necesitan ser compiladas como parte del mismo y no pueden ser cargadas como un módulo individual. Sin embargo en lo que respecta a la organización del código fuente, el prototipo presenta un buen grado de desacoplamiento del resto del kernel y puede verse como un módulo individual con una interfaz bien definida.

- **Portabilidad:** El mecanismo de distribución en forma de parches de Linux y la independencia de la arquitectura que presenta dicho kernel y que fue respetada por el módulo desarrollado, hacen el prototipo muy fácil de instalar en cualquier distribución de Linux, sobre cualquiera de las plataformas de hardware soportadas, aunque el módulo sólo fue testeado para x86.
- **Performance:** El prototipo ha mostrado además, que la ejecución del módulo desarrollado no tiene en general un efecto significativo en la performance del sistema operativo y la ejecución de procesos de usuario. Aunque cabe destacar que se observa mayor degradación en sistemas con aplicaciones intensivas en el uso del procesador.
- **Documentación:** si se revisa el capítulo de este trabajo sobre el prototipo, más los informes realizados en las distintas etapas y los comentarios incluidos en los fuentes, se encuentra una completa documentación y justificación de decisiones de diseño e implementación, más detalles de algoritmos y procedimientos. Se entiende que uno de los puntos fuertes de un prototipo cuya implementación puede ser bastante difícil de comprender a simple vista, es un alto grado de documentación del mismo.

5.1.2. Inyección de errores. Un punto aparte se merece el mecanismo de inyección de errores desarrollado. Es verdad que el mismo tiene la importante desventaja de ser un parche al kernel y requerir la recompilación de éste para su instalación. Además, la interfaz actual mediante la cual se accede utiliza system calls lo cual no resulta muy apropiado. Sin embargo, su simplicidad, el bajo nivel en el que trabaja y el hecho de que ejecuta en modo kernel, le permiten brindar una flexibilidad y transparencia en la inyección de errores, que son difíciles de igualar ante los requerimientos de simulación de errores como los que tuvo este proyecto.

5.1.3. Resultados. Las conclusiones y análisis más importantes que se pueden derivar del proyecto, salen sin dudas del capítulo 4, dedicado a documentar los experimentos realizados y la interpretación de sus resultados. Allí se llegó a varias afirmaciones importantes que cabe resumir:

- Las pruebas realizadas en la sección 4.1 permiten concluir primero que el conjunto de frames de sólo lectura de un proceso particular puede variar mucho dependiendo de las características del mismo. Puede llegar a valores cercanos al 90 % de su total de frames residentes, para procesos que manejan menor volumen de datos o bajar hasta un 20 % para procesos que manejan gran flujo de datos. En general se puede concluir además, que todos los porcentajes observados indican un buen grado de cubrimiento del espacio de memoria de estos procesos, considerando las limitaciones según los permisos.
- En contraparte a lo anterior, no se pudo detectar la ocurrencia de soft errors en condiciones naturales a nivel del mar, pero se encontraron estadísticas que evidencian la baja probabilidad de ocurrencia de dicho suceso. Sin embargo, como se dijo en varias oportunidades, el nivel del mar y las condiciones normales no son el principal escenario objetivo de las herramientas aquí planteadas.
- En un esfuerzo por demostrar la importancia del efecto en la disponibilidad de los sistemas que puede tener la protección de errores sobre el conjunto de

frames de sólo lectura, se pudo mostrar casos en los que esta funcionalidad es la diferencia entre una correcta atención de los pedidos a un servidor y la falta continua de disponibilidad del mismo.

- Finalmente en una comparación de efectividad de las estrategias de búsqueda de errores del demonio, se demostró que la verificación por procesos registrados brinda una ventana mucho menor en todos los casos, entre la ocurrencia y detección del error.

5.2. Trabajo a futuro

Las líneas de trabajo futuras que se proponen para la continuación de este proyecto se dividen en dos categorías principales. La primera incluye algunas carencias del trabajo aquí presentado, las cuales dan posibilidad de mejoras a futuro. La segunda clase se trata de ideas sobre rumbos a tomar en la búsqueda de nuevas herramientas o estrategias para extender el trabajo realizado.

5.2.1. Mejoras. El prototipo desarrollado tiene algunos defectos remanentes como se explicó en la sección 3.9.3, como por ejemplo la posible falta de manejo a un tipo de evento de cambios en los permisos de las áreas virtuales correspondientes a mapeos anónimos. También, en la sección 3.10.4 se señalaron optimizaciones de performance a realizar a la estrategia del demonio que recorre los procesos registrados. La solución de estos defectos que no se lograron corregir es sin dudas una importante tarea de mejora.

En la sección 3.7.5 donde se habla del mecanismo de detección de errores utilizado, se menciona que en futuras versiones de la API de criptografía de Linux las interfaces de la misma se modificaran y podrían ser reutilizadas por el prototipo. Dado que en las distribuciones del kernel liberadas hasta el momento, no se han incluido dichos cambios, queda como trabajo a futuro evaluar las ventajas de reutilizar dicha API para sustituir los algoritmos criptográficos utilizados hasta el momento. Además, en la sección 3.7.7 donde se justifica la elección del código de corrección utilizado, se menciona que los códigos de reed-solomon[**42**] podrían ser apropiados para mejorar las capacidades de corrección del prototipo sin exceder los niveles de overhead aceptables para la redundancia. Se descartó el uso de éstos códigos por lo costoso de su implementación, pero se deja planteado como una posible mejora a futuro.

En la sección 3.7.8 se trató el rejuvenecimiento automático de frames desde disco por parte del sistema operativo. Allí se mencionó que en caso de los frames con mapeos anónimos no existe por defecto un respaldo en disco o la memoria de su contenido y la complejidad de crear uno excedía el alcance del proyecto. Se considera que contar con un almacén en disco que brinde dicho respaldo, mejoraría considerablemente las capacidades de corrección del prototipo sobre esa clase de mapeos y que la opción más factible para lograrlo, sería reutilizar parte de la lógica del mecanismo de swap y los archivos de swap, pero creando una instancia independiente de los mismos para no interferir con su tarea.

Otro punto de mejora del prototipo se encuentra en lo que refiere a la plataforma. Si bien el mismo fue desarrollado sobre código del kernel que no depende de la plataforma, sólo fue testeado con la arquitectura x86 y por tanto no se puede afirmar nada sobre su comportamiento en otras arquitecturas. El testeo del prototipo en distintas arquitecturas y distribuciones de Linux, es otro importante punto a mejorar.

Con respecto al mecanismo de inyección de errores presentado en la sección 3.9.1, se entiende que sería mejor migrar su interfaz a una forma de comunicación con el espacio de usuarios, de menor impacto para el sistema operativo que la modificación del conjunto de llamadas al sistema. De todas formas, como se explicó en esa sección, dado que la reutilización del mismo requiere recompilar el kernel, puede no ser de mucha utilidad en proyectos donde esto no sea parte de los requerimientos originales.

En lo que refiere al capítulo de resultados y en general al análisis de la efectividad de las herramientas presentadas, se considera que éste no fue suficiente. En particular la puesta en producción de una versión estable del prototipo en varios equipos de forma controlada y continua durante un período de tiempo de varios meses o hasta años, aumentaría las probabilidades de detección de soft errors ocurridos de forma natural, aunque no es posible garantizar el éxito de esta prueba. Un experimento más efectivo constaría de realizar inyección de errores por hardware provocando interferencias electromagnéticas por medio de alguna clase de bobina. En conclusión, con los recursos dedicados y tiempo necesarios, se podrían llevar a cabo otros experimentos o repetir algunos de los realizados en simulaciones más complejas e interesantes.

5.2.2. Propuestas. Gracias a la experiencia adquirida durante el desarrollo del proyecto, surgieron algunas propuestas de técnicas o herramientas a aplicar en éste o posteriores prototipos para mejorar su efectividad o su grado de cubrimiento de la memoria de los procesos.

Una propuesta que se entendió bastante posible de llevar a cabo, es desde el sistema operativo e intenta aumentar el porcentaje de frames que se acceden sólo con permisos de lectura. La estrategia propuesta busca ampliar la detección de errores, difiriendo la exclusión del conjunto de sólo lectura de un frame mapeado con permisos de escritura a la primer escritura sobre el mismo. Esto implicaría un cambio en la política del módulo PGSA para la conformación del conjunto, pero además se debería usar una técnica similar a la presentada por el mecanismo de copy-on-write para detectar la primer escritura, por medio de la asignación de permisos de sólo lectura en la entrada de tablas de páginas.

Otra idea que podría contribuir a un incremento en la memoria física cubierta por el prototipo, seria la optimización del código objeto de los programas que ejecutan en Linux para que las áreas que se reservan para su espacio de memoria se creen con permisos de escritura solo cuando esto es estrictamente necesario, en particular para el caso de los mapeos de archivos. Esta idea necesita todavía alguna maduración y parece más difícil de llevar a cabo porque implica requerimientos de optimización en los compiladores para este sistema operativo e incluso políticas a tomar en cuenta en el desarrollo de software para esta plataforma.

Bibliografía

[1] Y. Huang, C. Kintala, N. Kolettis, and F. N. D. - Software rejuvenation: Analysis, module and applications. En: Proc. Of th Int. Symposium on Fault-Tolerance Computing (FTCS-25), Pasadena, CA, USA, June 1995. - ISBN: 0-8186-7079-7

[2] J. Gray. - Why do Computers Stop and What Can Be Done About it? Tandem Computers, Technical Report 85.7, PN 87614, June 1985.

[3] K. Vaidyanathan, K. S. Trivedi. - Extended Classification of Software Faults Based on Aging. Dept. of ECE, Duke University, Durham, USA, 2001. En: 12th International Symposium on Software Reliability Engineering, Hong Kong, November. pp. 99.

[4] B. Littlewood, P. Popov, L. Strigini. - Design Diversity: an Update from Research on Reliability Modelling. Centre for Software Reliability, City University London, U.K. En: Proc of the 9th Safety-critical Systems Symposium, Bristol 2001, pp. 139-154

[5] Actel. - Understanding Soft and Firm Errors in Semiconductor Devices - 51700002-1/12.02.

[6] J.F. Ziegler. - Terrestrial cosmic rays. En: IBM Journal of Research and Development, Vol. 40, no. 1, pp. 19-40, Jan 1996. - ISSN:0018-8646

[7] J. F. Ziegler, M. E. Nelson, J. D. Shell, R. J. Peterson, C. J. Gelderloos, H. P. Muhlfeld, C. J. Montrose. - Cosmic ray soft error rates of 16-Mb DRAM memory chips. En: IEEE Journal of Solid-State Circuits, Vol. 33, no. 2, pp. 246-252, Feb 1998. - ISSN: 0018-9200

[8] K. S. Trivedi. - Proactive Management of Software Systems. Center for Advanced Computing and Communication. Duke University. En: Simulation Symposium, 33rd Annual, pp. 3-3, 2000.

[9] K. S. Trivedi, K. Vaidyanathan, K. Goseva-Popstojanova. - Modeling and analysis of software aging and rejuvenation. En: 33rd Annual Simulation Symposium, pp. 270-279, 2000. - ISBN: 0-7695-0598-8

[10] G. Pfister. - In Search of Clusters: The Coming Battle in Lowly Parallel Computing, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1998. - ISBN: 0-13-437625-0

[11] K. Hwang. - Advanced Computer Architecture: Parallelism, Scalability and Programmability, McGraw-Hill Book Co., Inc., New York, 1993. - ISBN: 0-0703-1622-8

[12] Daniel P. Bovet, Marco Cesati. - Understanding the Linux Kernel, 3rd Edition, Noviembre 2005. - ISBN: 0-5960-0565-2

[13] corbet. - DebugFS (http://lwn.net/Articles/115405/). Posteado el 13 de Diciembre del 2004. Al día 17 de Noviembre del 2009

[14] Kernel Korner. - Why and How to Use Netlink Socket (http://www.linuxjournal.com/article/7356) - Kevin Kaichuan, Posteado el 5 de Enero del 2005. Al día 17 de Noviembre del 2009

[15] Open Source Definition (http://www.opensource.org/docs/osd). Al día 17 de Noviembre del 2009

[16] corbet. - Execute-in-place (http://lwn.net/Articles/135472/). Posteado el 11 de Mayo del 2005. Al día 17 de Noviembre del 2009

[17] E. Mouw. - Linux Kernel Procfs Guide (http://buffer.antifork.org/linux/procfs-guide.pdf). Delft University of Technology. Faculty of Information Technology and Systems. Al día 17 de Noviembre del 2009

[18] S. Garg, A. van Moorsel, K. Vaidyanathan, K. Trivedi. - A Methodology for Detection and Estimation of Software Aging. En: Proceedings of the The Ninth International Symposium on Software Reliability Engineering, p 283, 1998. - ISBN:0-8186-8991-9

[19] MSDN. - IIS Process Recycling. (http://msdn.microsoft.com/en-us/library/ms525803.aspx). Al día 17 de Noviembre del 2009

[20] IBM, IBM Director Software Rejuvenation (http://srejuv.ee.duke.edu/swrejuv.pdf). Al día 17 de Noviembre del 2009
[21] A. Sabiguero and A. Aguirre - Inyección de errores para evaluación de aspectos no funcionales en sistemas. Instituto de Computación, Facultad de Ingeniería. En: X Jornadas de Informática e Investigación Operativa 2008 - 24-28/11, Montevideo, Uruguay, 2008
[22] Apache HTTP Server (http://httpd.apache.org/). Al día 17 de Noviembre del 2009
[23] MySQL (http://www.mysql.com/). Al día 17 de Noviembre del 2009
[24] PHP (http://www.php.net/). Al día 17 de Noviembre del 2009
[25] openSUSE (http://es.opensuse.org/Bienvenidos_a_openSUSE.org). Al día 17 de Noviembre del 2009
[26] WordPress (http://es.wordpress.com/). Al día 17 de Noviembre del 2009
[27] ab - Apache HTTP server benchmarking tool (http://httpd.apache.org/docs/2.0/programs/ab.html). Al día 17 de Noviembre del 2009
[28] Beowulf (http://www.beowulf.org/). Al día 17 de Noviembre del 2009
[29] MICRON. 1997. Module mean time between failures (MTBF). Tech. Note TN-04-45. (http://download.micron.com/pdf/technotes/DT45.pdf). Al día 13 de Setiembre del 2009
[30] Tezzaron Semiconductor. - Soft Errors in Electronic Memory. (http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf). Al día 17 de Noviembre del 2009
[31] J. F. Ziegler. - Trends in Electronic Reliability - Effects of Terrestrial Cosmic Rays. United States Naval Academy: (http://www.srim.org/SER/SERTrends.htm). Al día 17 de Noviembre del 2009
[32] Apple - Mac OS X - New technologies in Snow Leopard. (http://www.apple.com/macosx/technology/). Al día 17 de Noviembre del 2009
[33] POV-Ray - The Persistence of Vision Raytracer (http://www.povray.org/). Al día 17 de Noviembre del 2009
[34] The Persistence of Ignorance. The average office (http://www.ignorancia.org/en/index.php?page=The_office). Al día 17 de Noviembre del 2009
[35] M. Blair, S. Obenski & P. Bridickas. - GAO/IMTEC-92-26 Patriot Missile Software Problem, Information Management and Technology Division, February 4,1992 . - B-247094
[36] K. Vaidyanathan, Richard E. Harper, W. Hunter, Kishor S. Trivedi. - Analysis and Implementation of Software Rejuvenation in Cluster Systems. En: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. p 62-71, 2001 - ISBN:1-58113-334-0
[37] Solaris Operating System - Features Availability. (http://www.sun.com/software/solaris/availability.jsp). Al día 17 de Noviembre del 2009
[38] VMware. (www.vmware.com). Al día 17 de Noviembre del 2009
[39] Notepad++. (http://notepad-plus.sourceforge.net/es/site.htm). Al día 17 de Noviembre del 2009
[40] PuTTY. (http://www.putty.org/). Al día 17 de Noviembre del 2009
[41] WinSCP. (http://winscp.net/eng/index.php). Al día 17 de Noviembre del 2009
[42] R. Roth. - Introduction to Coding Theory, Cambridge University Press, 2006. - ISBN: 0-5218-4504-1
[43] G. L. Skibinski, R. J. Kerkman, and D. Schlegel. - EMI emissions of modern PWM AC drives. Industry Applications Magazine, IEEE. Volume 5, Issue 6, p 47-80. - ISSN: 1077-2618

Apéndice A

Glosario

Benchmarking: Se refiere a definir un proceso sistemático para medir y comparar el comportamiento de distintos sistemas para ciertas tareas particulares.

Billing: Proceso de acumular los gastos de distintos clientes para luego generar las facturas correspondientes para los mismos.

Bobina: Es un componente de un circuito eléctrico que, debido al fenómeno de la autoinducción, almacena energía en forma de campo magnético.

Bus: Subsistema que trasmite información entre distintos sistemas dentro de un sistema de computación.

Callback o Retrollamada: Código ejecutable que es pasado cómo un argumento a otro código. Esto permite al software invocado llamar a una subrutina (función) definida por el software invocante.

Checksum: Es un valor de tamaño fijo calculado sobre un bloque de datos.

Colaboración: En desarrollo de software, se refiere a el trabajo entre varios desarrolladores en la creación de la misma pieza de software, compartiendo entre ellos los cambios realizados a la misma.

Dirección física: Tipo de dirección de memoria utilizada por el hardware para acceder a la DRAM.

Dirección virtual: Tipo de dirección de memoria utilizada por los procesos.

Directorio de Paginas: Tabla utilizada por el mecanismo de paginación en su nivel superior.

DRAM: Es un tipo de RAM, en la cual los valores almacenados, son dinámicos, en este caso dinámicos significa que si a la memoria es desconectada del sistema, no se asegura que los valores almacenados en la misma puedan volver a ser recuperables.

Espacio de usuario: Se refiere al modo de operación en que ejecutan los procesos de usuario, en este modo no se tiene acceso directo al hardware, sino que se accede al mismo mediante el kernel.

Kernel: se puede definir kernel o núcleo de un sistema operativo como el software del mismo dónde se realizan las funcionalidades básicas como la gestión de procesos, la gestión de memoria y de entrada salida.

GPL: Es una licencia de software que se utiliza para la creación de software open source. La misma se utiliza para garantizar que las personas que modifican el software licenciado por la misma deban distribuir los fuentes con las modificaciones si distribuyen el software modificado.

Logs: Se refiere en general a un archivo de texto en el que se detallan eventos que sucedieron en el sistema de forma que los operarios del mismo puedan enterarse que está pasando dentro del sistema.

Mapeo de Memoria: Se define como la asociación de una página de memoria virtual con un frame de memoria física.

Marco o frame: Intervalo de direcciones físicas de tamaño fijo.

Memoria Anónima: Se refiere a los mapeos de memoria que no están relacionados con ningún objetó del sistema, como ser, archivos, dispositivos o memoria compartida.

Memoria Mapeada a archivo: Es una sección de memoria virtual para la cual se ha asignado una relación byte-a-byte con algún archivo, o estructura similar a un archivo.

Memoria Virtual: Los sistemas operativos modernos no permiten el acceso directo a la memoria física a los procesos de usuario, sino que les muestran una indirección a la misma llamada memoria virtual. Esta memoria puede ser mayor que la memoria física y es individual para cada grupo de procesos. Además los procesos pueden ver a esta memoria como un espacio contiguo de memoria, mientras que en la realidad estar distribuida en frames disjuntos.

Modificado en caliente: Es el cambio de un componente de software o de su comportamiento sin necesidad de detener la ejecución del mismo.

Open source: En software, se refiere a que es posible tener acceso al código fuente de un programa, modificarlo y distribuir esas modificaciones sin tener que pedirle autorización a nadie en particular.

Página: Intervalo de direcciones virtuales de tamaño fijo.

Paginación: En sistemas operativos, la paginación se refiere a almacenar páginas de la memoria principal en memoria secundaria, de forma de liberar espacio en la primera.

Partículas alfa: Son el resultado de reacciones nucleares o la descomposición radiactiva de los elementos, las mismas están formadas por 2 protones y 2 neutrones, por lo que su carga eléctrica es positiva.

PFN (**Page Frame Number**): Índice de un marco de memoria dentro de la memoria física, si pensamos esta como un gran array de marcos.

Proceso liviano: Es un proceso que comparte su espacio de memoria y recursos con otros procesos, pero que tiene su propio identificador de proceso.

Race condition: Se refiere a que el resultado de la operación de un sistema depende del orden de la ejecución de distintos componentes del mismo, y que en caso de ejecutarse en distinto orden o con distintos delays, el resultado obtenido es distinto que el resultado esperado.

RAM (**Random Access Memory**): Es una tecnología de memoria en la cual se puede acceder a cualquier byte de la misma, sin tener que antes leer ningún otro byte en particular.

Rayos cósmicos: Son partículas subatómicas que proceden del espacio exterior, las mismas son partículas que debido a su alta velocidad, cercana

Salud del sistema o del software: Se refiere al estado interno del sistema o software, pero no el definido por el propio sistema, sino el que se va generando de acuerdo a la asignación y desechado de recursos que se hayan ido ocurriendo y como el mismo afecta al comportamiento actual del sistema. Un ejemplo de este estado interno seria la fragmentación de la memoria virtual.

Segmentación: En el manejo de memoria segmentación es el dividir la memoria en segmentos y utilizar cada uno de esos segmentos para distintos tipos de datos definidos, por ejemplo un segmento para el stack, otro para el código y otro para datos.

Self-healing: Es el concepto de un sistema que se repara a sí mismo cuando detecta algún comportamiento incorrecto.

Semiconductores: Es una sustancia que se comporta como conductor o aislante dependiendo de factores externos, y que se la puede hacer cambiar de un modo al otro.

Sistemas embebidos: Son sistemas informáticos de uso específico, generalmente incluidos en sistemas mayores. Los mismos a diferencia de los equipos de escritorio suelen tener todos los módulos a utilizar incorporados en la placa base del mismo.

Software Aging: Degradación con el tiempo del estado de un proceso o de su ambiente.

Software de switching: Se refiere al software encargado en las empresas de telecomunicaciones de dirigir una comunicación desde el emisor al destinatario.

SRAM: Este tipo de RAM, al contrario de la DRAM, si asegura que si la memoria es desconectada, cuando se vuelva a conectar, va a seguir conteniendo la misma información almacenada. La S es por Static.

Swap: Se le llama Swap al espacio reservado en memoria secundaria para almacenar los datos que están siendo paginados.

Syscall: Es el nombre que se les da a las operaciones que expone el kernel para que sean invocadas desde otros espacios de ejecución.

Tabla de Paginas: Tabla utilizada por el mecanismo de paginación en su nivel inferior.

TI: Tecnologías de la Información.

Transistores: Son dispositivos electrónicos semiconductores que cumplen el rol de amplificar o conmutar señales eléctricas. Los mismos son los bloques fundamentales de los circuitos integrados.

VMA (**Virtual Memory Area**): Es una sección de direcciones de espacio de memoria virtual contigua.

Apéndice B

Manual de usuario

Se dedica esta sección a describir los procedimientos que pueden seguir los usuarios del módulo, mediante la interfaz desarrollada, para configurar las funcionalidades que brinda el mismo.

B.1. Modos de Funcionamiento

El módulo de detección y corrección de errores posee distintos modos de funcionamiento. Estos modos se controlan mediante el uso de banderas. Cada bandera tiene un efecto distinto sobre el comportamiento del módulo, y cada una puede estar activada o desactivada independientemente del resto. Existen dos grupos de banderas, que afectan distintas características del funcionamiento, y cada una tiene asociada una constante numérica única que la identifica. Para configurar el módulo, se deberá escribir en el archivo `/proc/kpgsa/pgsa_mode` el valor numérico de la suma de las constantes de aquellas banderas que se desee habilitar.

Ejemplo::

Para activar las banderas PGSA_CHECK_DAEMON y PGSA_CHECK_ALL se deberá escribir en el archivo el valor 6 utilizando un comando como el siguiente:

```
# echo 6 > /proc/kpgsa/pgsa_mode
```

A continuación se presentan los distintos grupos de banderas, y dentro de cada uno, se explica el funcionamiento de cada bandera particular.

B.1.1. General. Este grupo controla el funcionamiento global del módulo y cuenta actualmente con una única bandera.

B.1.1.1. PGSA_ON (0x100). Funciona como interruptor tipo on/off de todo el módulo.

B.1.2. Detección de Errores. Este primer grupo de banderas controla las estrategias utilizadas para la detección de los errores. A continuación se detallan las banderas comprendidas en este grupo, y su significado.

B.1.2.1. PGSA_CHECK_CURRENT (0x01). Esta bandera controla la utilización del mecanismo de búsqueda de errores en el espacio de memoria del próximo proceso a ejecutar. Esto permite incrementar la probabilidad de detectar un error antes de que éste se pueda manifestar y posiblemente afectar negativamente los resultados del proceso, dado que la verificación se realiza antes que se obtenga el control del procesador. Si esta bandera se encuentra desactivada, no se chequean las páginas del siguiente proceso a ejecutar.

B.1.2.2. PGSA_CHECK_DAEMON (0x02). Esta bandera controla el mecanismo de detección de errores de memoria. Si se encuentra activada, se utiliza

un demonio a nivel de kernel que continuamente itera sobre el conjunto de páginas de sólo lectura a verificar en busca de errores en las mismas. Si esta bandera se encuentra desactivada, este mecanismo no se utiliza. Si las banderas PGSA_CHECK_CURRENT y PGSA_CHECK_DAEMON están ambas desactivadas se deshabilita completamente la funcionalidad del módulo de detección de errores. Este caso es similar a desactivar la bandera PGSA_ON, con la diferencia que se siguen ejecutando las otras secciones del código, como ser el mantenimiento de los CRCs y/o los códigos de Hamming.

B.1.2.3. PGSA_CHECK_ALL (0x04). Al activar esta bandera, los mecanismos de detección de errores se aplican a las páginas de todos los procesos presentes en el sistema. Si por el contrario se encuentra desactivada, solamente se chequean las páginas de aquellos procesos que se hayan registrado para tal fin. Combinando esta bandera con la anterior, si PGSA_CHECK_CURRENT está activada, solamente se chequea el proceso a ejecutar si PGSA_CHECK_ALL también está activada, o el proceso a ejecutarse está registrado para ser verificado. En caso que ambas banderas estén activadas hay que tener en cuenta que se agrega una carga significativa al sistema, ya que se chequean las páginas de todos los procesos. Es recomendable utilizar el mecanismo de registro de procesos para optimizar la carga del sistema.

B.1.3. Acciones al Detectar Errores. El otro grupo de banderas definido determinan el comportamiento del módulo una vez detectado un error en alguna página de memoria. Las posibles configuraciones se detallan en las siguientes secciones.

B.1.3.1. PGSA_CORRECTION_CODE (0x10). Esta bandera controla el funcionamiento de la corrección de errores. Si está activada, se intentará corregir un error una vez detectado, utilizando códigos de corrección de errores y redundancia. Si se encuentra desactivada, no se intentará corregir los errores, ni se llevará registro de la redundancia necesaria para la corrección.

B.1.3.2. PGSA_FILE_REJUVENATION (0x20). Esta bandera controla el funcionamiento del rejuvenecimiento automático de páginas mapeadas a archivos, cuando se detecta un error.

B.1.3.3. PGSA_NOTIFICATIONS (0x40). Al activar o desactivar esta bandera, se habilita o deshabilita respectivamente la acción de publicación al espacio de usuarios de información detallada sobre cada error ocurrido.

B.1.3.4. PGSA_SIGNAL (0x80). Al activar o desactivar esta bandera, se habilita o deshabilita respectivamente la acción de envío de señales a los procesos que tienen mapeada la página en la cual se detectó el error. Estas notificaciones solamente se envían a los agentes de procesos que estén registrados en el módulo como agentes de notificación.

B.2. Registro de Procesos

El registro de procesos permite por un lado limitar el alcance de la detección de errores, y por otro lado que un proceso actúe como agente, recibiendo las notificaciones de detección de errores, para tomar acciones correspondientes. Esta funcionalidad también es interesante para contar con un mecanismo que permita restringir la detección a un proceso o un conjunto de procesos de mayor importancia para el correcto funcionamiento del sistema en general.

B.2.1. Grupos de Tareas. Un proceso y sus hijos pertenecen a lo que se conoce como grupo de tareas. Para evitar tener que registrar cada uno de los hilos de un programa multi-hilado, el módulo de detección de errores permite registrar todos los hilos pertenecientes a una misma tarea simplemente registrando cualquiera de sus componentes.

B.2.2. Agente de Notificación. Otro concepto del registro de procesos es el llamado agente de notificación. El agente de notificación es el proceso encargado de recibir las notificaciones de detección de errores. En la sección anterior vimos que se registran grupos de tareas a ser monitoreados. El agente que recibe las notificaciones puede ser un hilo de éste mismo grupo de tareas (si se desea manejar los errores internamente por la aplicación), o puede ser un proceso totalmente independiente. Esto permite tener procesos externos cuya función es encargarse de lidiar con errores de memoria independientemente de la aplicación afectada.

B.2.3. Modo de Registro. Para registrar procesos a ser monitoreados, y sus agentes correspondientes se deberá escribir en el archivo `/proc/kpgsa/register`. El archivo recibe pares de pid (identificadores de proceso) de la siguiente manera:

PID1-PID2, donde PID1 es el identificador de proceso del agente de notificación, y PID2 es el identificador de proceso de un integrante del grupo de tareas a monitorear.

Ejemplo::

Supongamos que tengo los procesos con pid 2580 y 2581 pertenecientes a un mismo grupo de tareas, y que se desean monitorear. El agente de notificación será un proceso con identificador 2500. Se deberá ejecutar una sentencia como la que sigue para realizar el registro:

```
# echo 2500-2580 > /proc/kpgsa/register
```

Notar que se podría haber utilizado el pid 2581 en lugar de 2580 ya que ambos procesos pertenecen al mismo grupo.

A continuación tenemos un ejemplo de cómo se realizaría esto desde código, aquí tenemos un proceso que se registra como agente para un proceso en particular y se queda esperando la señal indicando el error.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "/usr/src/linux-2.6.25.9-fabricio/include/asm-x86/unistd.h"

char * get_time ( char time_str[] ) {
        struct tm * local_time;
        struct timeval start_time;
        memset(time_str,'\0',74);
        gettimeofday ( & start_time, NULL );
        local_time = localtime ( & start_time.tv_sec );
        strftime ( time_str, 64, "%C%y-%m-%d %H:%M:%S", local_time );
        char swap[6];
        sprintf(swap,".%03d",start_time.tv_usec/1000);
```

```
        strncat(time_str,swap,4);
        return ( time_str );
}

void handler(int sig) {
        char str_time[74];
        printf("Error detectado: %s", get_time (str_time));
}

int main(int argc, char *argv[]) {
        char *endptr;
        char str_time[74];
        if (argc <= 3) {
                printf("Usage agente pid address new_value\n");
                return -1;
        }
        pid_t pid = atoi(argv[1]);
        //primero registro este proceso como agente del proceso a
            inyectar
        FILE * f = fopen("/proc/kpgsa/register","w");
        if(f){
                fprintf (f, "%d-%d", getpid(), pid);
                fclose(f);
                printf("proceso registrado.");
        }else{
                printf("error al abrir archivo.");
        }

        //seteo handler para la señal
        signal(SIGUNUSED, handler);
        //duermo a esperar la señal
        pause();
        return ret;
}
```

B.2.4. Envío de Notificaciones. Una vez detectado un error, y si existe algún proceso registrado como agente de notificación, se le envía la señal SIGUNUSED a dicho agente, el cual deberá encargarse de implementar el comportamiento deseado al encontrar un error de memoria. Se utiliza la señal SIGUNUSED ya que es poco probable que los procesos la utilicen con algún otro propósito.

B.3. Archivos de Estadísticas

Mientras el módulo se encuentra en funcionamiento, se van recolectando ciertas estadísticas sobre su desempeño, las cuales se almacenan en archivos bajo el directorio `/proc/kpgsa`. Estos archivos son de sólo lectura y no es posible resetear sus valores. En general llevan contadores sobre las distintas actividades del módulo. Los archivos que se generan y el contenido de cada uno de ellos se explican en las siguientes secciones.

B.3.1. cycles. El archivo `/proc/kpgsa/cycles` contiene un conteo de todas las pasadas por la colección de páginas. Cuando el módulo termina de chequear todas las páginas que corresponden, se incrementa en uno el valor de este contador

antes de comenzar nuevamente el ciclo de chequeos. Para consultar el archivo se puede utilizar simplemente el comando cat de Unix de la siguiente manera:

```
# cat /proc/kpgsa/cycles
```

B.3.2. checks. Este archivo ubicado en `/proc/kpgsa/checks` contiene la cantidad de chequeos de errores que se realizaron en la última recorrida sobre las páginas. De la misma forma, el archivo checks se puede consultar utilizando el comando cat:

```
# cat /proc/kpgsa/checks
```

B.3.3. errors. El archivo `/proc/kpgsa/errors` contiene un conteo general de todos los errores que detecta el módulo. Al igual que el resto de los archivos de estadísticas, se consulta con un simple comando cat:

```
# cat /proc/kpgsa/errors
```

Apéndice C

Porcentaje de frames de solo lectura

En este anexo se presenta una versión modificada del prototipo 1.5, la cual imprime periódicamente para un proceso indicado, el porcentaje de frames mapeados con permisos de sólo lectura. Dicho proceso se selecciona por medio de una syscall, creada para este propósito.

La presentación del código será realizada por medio de un diff entre el código modificado y el kernel Vainilla.

```
Files linux-2.6.25.9//arch/x86/boot/compressed/relocs and
    linux//arch/x86/boot/compressed/relocs differ
Files linux-2.6.25.9//arch/x86/boot/compressed/vmlinux.bin.
    all and linux//arch/x86/boot/compressed/vmlinux.bin.all
    differ
Files linux-2.6.25.9//arch/x86/boot/compressed/vmlinux.
    relocs and linux//arch/x86/boot/compressed/vmlinux.
    relocs differ
diff -urN linux-2.6.25.9//arch/x86/kernel/syscall_table_32.S
     linux//arch/x86/kernel/syscall_table_32.S
--- linux-2.6.25.9//arch/x86/kernel/syscall_table_32.S
    2008-06-24 18:09:06.000000000 -0300
+++ linux//arch/x86/kernel/syscall_table_32.S    2009-08-17
    17:42:46.000000000 -0300
@@ -326,3 +326,5 @@
        .long sys_fallocate
        .long sys_timerfd_settime        /* 325 */
        .long sys_timerfd_gettime
+       .long sys_kpgsa_set_stats_pid
+
diff -urN linux-2.6.25.9//include/asm-x86/unistd_32.h linux
    //include/asm-x86/unistd_32.h
--- linux-2.6.25.9//include/asm-x86/unistd_32.h 2008-06-24
    18:09:06.000000000 -0300
+++ linux//include/asm-x86/unistd_32.h   2009-08-17
    17:44:24.000000000 -0300
@@ -332,6 +332,7 @@
 #define __NR_fallocate          324
 #define __NR_timerfd_settime    325
 #define __NR_timerfd_gettime    326
```

```
+#define __NR_kpgsa_set_stats_pid          327

 #ifdef __KERNEL__

 diff -urN linux-2.6.25.9//include/linux/mm_types.h linux//
    include/linux/mm_types.h
--- linux-2.6.25.9//include/linux/mm_types.h      2008-06-24
    18:09:06.000000000 -0300
+++ linux//include/linux/mm_types.h       2009-04-15
    12:24:43.000000000 -0300
@@ -225,6 +225,8 @@
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR
         struct mem_cgroup *mem_cgroup;
 #endif
+        mm_counter_t _page_count;
+        mm_counter_t _ro_page_count;
 };

 #endif /* _LINUX_MM_TYPES_H */
 diff -urN linux-2.6.25.9//include/linux/syscalls.h linux//
    include/linux/syscalls.h
--- linux-2.6.25.9//include/linux/syscalls.h      2008-06-24
    18:09:06.000000000 -0300
+++ linux//include/linux/syscalls.h       2009-08-17
    17:46:28.000000000 -0300
@@ -605,6 +605,12 @@
                                 size_t __user *len_ptr);
 asmlinkage long sys_set_robust_list(struct robust_list_head
      __user *head,
                                 size_t len);
+
+/* PGSOFTAG */
+asmlinkage long sys_kpgsa_set_stats_pid(pid_t pid);
+/* PGSOFTAG */
+
+
 asmlinkage long sys_getcpu(unsigned __user *cpu, unsigned
     __user *node, struct getcpu_cache __user *cache);
 asmlinkage long sys_signalfd(int ufd, sigset_t __user *
     user_mask, size_t sizemask);
 asmlinkage long sys_timerfd_create(int clockid, int flags);
 diff -urN linux-2.6.25.9//kernel/exit.c linux//kernel/exit.c
--- linux-2.6.25.9//kernel/exit.c         2008-06-24
    18:09:06.000000000 -0300
```

```
+++ linux//kernel/exit.c          2009-08-18
    15:00:11.000000000 -0300
@@ -581,7 +581,15 @@
 static void exit_mm(struct task_struct * tsk)
 {
         struct mm_struct *mm = tsk->mm;
-
+
+        // Imprimo los valores de rss y nuestro counter para
     comparar
+/*      if ( mm ) {
+          if(get_mm_counter(mm,page_count) < get_mm_counter(
    mm,ro_page_count))
+          printk("PGSOFTAG BUG exit_mm(%s) - pid %d - rss -
     %lu ++ pc - %lu - ro_pc - %lu\n", tsk->comm,tsk->pid
    , get_mm_counter(mm,file_rss) + get_mm_counter(mm,
    anon_rss) , get_mm_counter(mm,page_count),get_mm_counter
    (mm,ro_page_count));
+                 printk("PGSOFTAG exit_mm(%s) - pid %d -
    rss - %lu ++ pc - %lu - ro_pc - %lu\n", tsk->comm,tsk
    ->pid, get_mm_counter(mm,file_rss) + get_mm_counter(mm,
    anon_rss) , get_mm_counter(mm,page_count),get_mm_counter
    (mm,ro_page_count));
+        }else
+                 printk("PGSOFTAG exit_mm(%s) - pid %d - mm
     es NULL\n",tsk->comm,tsk->pid);
+*/
         mm_release(tsk, mm);
         if (!mm)
                 return;
 diff -urN linux-2.6.25.9//kernel/fork.c linux//kernel/fork.c
--- linux-2.6.25.9//kernel/fork.c          2008-06-24
    18:09:06.000000000 -0300
+++ linux//kernel/fork.c          2009-04-23
    16:44:48.000000000 -0300
@@ -353,6 +353,8 @@
         mm->nr_ptes = 0;
         set_mm_counter(mm, file_rss, 0);
         set_mm_counter(mm, anon_rss, 0);
+        set_mm_counter(mm, page_count, 0);
+        set_mm_counter(mm, ro_page_count, 0);
         spin_lock_init(&mm->page_table_lock);
         rwlock_init(&mm->ioctx_list_lock);
         mm->ioctx_list = NULL;
@@ -362,6 +364,7 @@
```

```
        if (likely(!mm_alloc_pgd(mm))) {
                mm->def_flags = 0;
+//              printk("PGSOFTAG -- mm_init(%s) -- pid %d
    -- rss - %lu ++ page_count -- %lu\n",p->comm,p->pid,
    get_mm_counter(mm,file_rss)+get_mm_counter(mm,anon_rss),
    get_mm_counter(mm,page_count));
                return mm;
        }

diff -urN linux-2.6.25.9//mm/filemap_xip.c linux//mm/
    filemap_xip.c
--- linux-2.6.25.9//mm/filemap_xip.c      2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/filemap_xip.c       2009-04-15
    14:33:28.000000000 -0300
@@ -197,6 +197,9 @@
                        pteval = ptep_clear_flush(vma,
                            address, pte);
                        page_remove_rmap(page, vma);
                        dec_mm_counter(mm, file_rss);
+                       dec_mm_counter(mm, page_count);
+               if(!(vma->vm_flags & VM_WRITE))
+                   dec_mm_counter(mm, ro_page_count);
                        BUG_ON(pte_dirty(pteval));
                        pte_unmap_unlock(pte, ptl);
                        page_cache_release(page);
diff -urN linux-2.6.25.9//mm/fremap.c linux//mm/fremap.c
--- linux-2.6.25.9//mm/fremap.c 2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/fremap.c  2009-04-15 14:34:05.000000000 -0300
@@ -38,6 +38,9 @@
                        page_cache_release(page);
                        update_hiwater_rss(mm);
                        dec_mm_counter(mm, file_rss);
+                       dec_mm_counter(mm, page_count);
+               if(!(vma->vm_flags & VM_WRITE))
+                   dec_mm_counter(mm, ro_page_count);
                }
        } else {
                if (!pte_file(pte))
diff -urN linux-2.6.25.9//mm/memory.c linux//mm/memory.c
--- linux-2.6.25.9//mm/memory.c 2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/memory.c  2009-05-06 13:46:07.000000000 -0300
```

```
@@ -52,6 +52,10 @@
 #include <linux/writeback.h>
 #include <linux/memcontrol.h>

+/* PGSOFTAG */
+#include <linux/crc32.h>
+/* PGSOFTAG */
+
 #include <asm/pgalloc.h>
 #include <asm/uaccess.h>
 #include <asm/tlb.h>
@@ -538,6 +542,9 @@
         spin_unlock(src_ptl);
         pte_unmap_nested(src_pte - 1);
         add_mm_rss(dst_mm, rss[0], rss[1]);
+     add_mm_counter(dst_mm, page_count, rss[0] + rss[1]);
+     if(!(vma->vm_flags & VM_WRITE))
+         add_mm_counter(dst_mm, ro_page_count, rss[0] + rss
    [1]);
         pte_unmap_unlock(dst_pte - 1, dst_ptl);
         cond_resched();
         if (addr != end)
@@ -634,6 +641,7 @@
         spinlock_t *ptl;
         int file_rss = 0;
         int anon_rss = 0;
+        int page_count = 0;

         pte = pte_offset_map_lock(mm, pmd, addr, &ptl);
         arch_enter_lazy_mmu_mode();
@@ -687,6 +695,7 @@
                                         SetPageReferenced(
                                             page);
                                 file_rss--;
                         }
+                        page_count--;
                         page_remove_rmap(page, vma);
                         tlb_remove_page(tlb, page);
                         continue;
@@ -703,6 +712,10 @@
         } while (pte++, addr += PAGE_SIZE, (addr != end && *
             zap_work > 0));

         add_mm_rss(mm, file_rss, anon_rss);
+        add_mm_counter(mm, page_count, page_count);
```

```
+     if(!(vma->vm_flags & VM_WRITE))
+         add_mm_counter(mm, ro_page_count, page_count);
+
          arch_leave_lazy_mmu_mode();
          pte_unmap_unlock(pte - 1, ptl);

@@ -1193,6 +1206,9 @@
          /* Ok, finally just insert the thing.. */
          get_page(page);
          inc_mm_counter(mm, file_rss);
+         inc_mm_counter(mm, page_count);
+     if(!pte_write(mk_pte(page, prot)))
+         inc_mm_counter(mm, ro_page_count);
          page_add_file_rmap(page);
          set_pte_at(mm, addr, pte, mk_pte(page, prot));

@@ -1691,8 +1707,12 @@
                                    dec_mm_counter(mm, file_rss)
                                        ;
                                    inc_mm_counter(mm, anon_rss)
                                        ;
                              }
-                   } else
-                             inc_mm_counter(mm, anon_rss);
+                   } else{
+                             inc_mm_counter(mm, anon_rss);
+                             inc_mm_counter(mm, page_count);
+                     if(!(vma->vm_flags & VM_WRITE))
+                         inc_mm_counter(mm, ro_page_count);
+                   }
                    flush_cache_page(vma, address, pte_pfn(
                        orig_pte));
                    entry = mk_pte(new_page, vma->vm_page_prot);
                    entry = maybe_mkwrite(pte_mkdirty(entry),
                        vma);
@@ -2104,6 +2124,9 @@
          /* The page isn't present yet, go ahead with the
             fault. */

          inc_mm_counter(mm, anon_rss);
+         inc_mm_counter(mm, page_count);
+     if(!(vma->vm_flags & VM_WRITE))
+         inc_mm_counter(mm, ro_page_count);
          pte = mk_pte(page, vma->vm_page_prot);
          if (write_access && can_share_swap_page(page)) {
```

```
                pte = maybe_mkwrite(pte_mkdirty(pte), vma);
@@ -2173,6 +2196,9 @@
        if (!pte_none(*page_table))
                goto release;
        inc_mm_counter(mm, anon_rss);
+       inc_mm_counter(mm, page_count);
+     if(!(vma->vm_flags & VM_WRITE))
+       inc_mm_counter(mm, ro_page_count);
        lru_cache_add_active(page);
        page_add_new_anon_rmap(page, vma, address);
        set_pte_at(mm, address, page_table, entry);
@@ -2218,6 +2244,8 @@
        struct vm_fault vmf;
        int ret;
        int page_mkwrite = 0;
+       u32 crc;
+       void *page_addr;

        vmf.virtual_address = (void __user *)(address &
           PAGE_MASK);
        vmf.pgoff = pgoff;
@@ -2336,7 +2364,15 @@
                                dirty_page = page;
                                get_page(dirty_page);
                        }
+/* PGSOFTAG */
+//                      page_addr = page_address(page);
+//                      crc = crc32(0,page_addr,PAGE_SIZE);
+//                      printk("PGSOFTAG CRC32 -- page %p
   crc32 %u -- pid %d\n",page_addr,crc,current->pid);
+/* PGSOFTAG */
                }
+               inc_mm_counter(mm, page_count);
+         if(!(vma->vm_flags & VM_WRITE))
+               inc_mm_counter(mm, ro_page_count);

                /* no need to invalidate: a not-present page
                     won't be cached */
                update_mmu_cache(vma, address, entry);
diff -urN linux-2.6.25.9//mm/mprotect.c linux//mm/mprotect.c
--- linux-2.6.25.9//mm/mprotect.c          2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/mprotect.c           2009-05-06
    11:22:08.000000000 -0300
@@ -204,6 +204,31 @@
```

```
                change_protection(vma, start, end, vma->
                    vm_page_prot, dirty_accountable);
         vm_stat_account(mm, oldflags, vma->vm_file, -nrpages
             );
         vm_stat_account(mm, newflags, vma->vm_file, nrpages)
             ;
+
+/*PGSOFTAG*/
+        if((vma->vm_flags & VM_WRITE) > (oldflags & VM_WRITE
    )){//si ahora es write y antes no
+            start = vma->vm_start;
+         long nrframes = 0;
+        do {
+                if(follow_page(vma, start, 0/*foll_flags*/))
    {
+                 nrframes++;
+                }
+                start += PAGE_SIZE;
+        } while (start < vma->vm_end);
+         add_mm_counter(mm, ro_page_count, -nrframes);
+    }else if((vma->vm_flags & VM_WRITE) < (oldflags &
    VM_WRITE)){//si anters era write y ahora no
+            start = vma->vm_start;
+         long nrframes = 0;
+        do {
+                if(follow_page(vma, start, 0/*foll_flags*/))
    {
+                 nrframes++;
+                }
+                start += PAGE_SIZE;
+        } while (start < vma->vm_end);
+         add_mm_counter(mm, ro_page_count, nrframes);
+    }
+/*PGSOFTAG*/
+
         return 0;

 fail:
diff -urN linux-2.6.25.9//mm/rmap.c linux//mm/rmap.c
--- linux-2.6.25.9//mm/rmap.c    2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/rmap.c    2009-04-15 14:31:25.000000000 -0300
@@ -749,6 +749,9 @@
                                 spin_unlock(&mmlist_lock);
                         }
```

```
                                dec_mm_counter(mm, anon_rss);
+                               dec_mm_counter(mm, page_count);
+                       if(!(vma->vm_flags & VM_WRITE))
+                           dec_mm_counter(mm, ro_page_count);
 #ifdef CONFIG_MIGRATION
                        } else {
                                /*
@@ -771,9 +774,12 @@
                        set_pte_at(mm, address, pte,
                            swp_entry_to_pte(entry));
                } else
 #endif
+               {
                        dec_mm_counter(mm, file_rss);
-
-
+                       dec_mm_counter(mm, page_count);
+                 if(!(vma->vm_flags & VM_WRITE))
+                       dec_mm_counter(mm, ro_page_count);
+               }
                page_remove_rmap(page, vma);
                page_cache_release(page);

@@ -867,6 +873,9 @@
                        page_remove_rmap(page, vma);
                        page_cache_release(page);
                        dec_mm_counter(mm, file_rss);
+                       dec_mm_counter(mm, page_count);
+                 if(!(vma->vm_flags & VM_WRITE))
+                       dec_mm_counter(mm, ro_page_count);
                        (*mapcount)--;
                }
                pte_unmap_unlock(pte - 1, ptl);
 diff -urN linux-2.6.25.9//mm/swapfile.c linux//mm/swapfile.c
--- linux-2.6.25.9//mm/swapfile.c        2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/swapfile.c         2009-04-15
    14:32:34.000000000 -0300
@@ -526,6 +526,10 @@
                }

                inc_mm_counter(vma->vm_mm, anon_rss);
+               inc_mm_counter(vma->vm_mm, page_count);
+       if(!(vma->vm_flags & VM_WRITE))
+               inc_mm_counter(vma->vm_mm, ro_page_count);
```

```
+
         get_page(page);
         set_pte_at(vma->vm_mm, addr, pte,
                  pte_mkold(mk_pte(page, vma->vm_page_prot)
                      ));
diff -urN linux-2.6.25.9//mm/vmscan.c linux//mm/vmscan.c
--- linux-2.6.25.9//mm/vmscan.c 2008-06-24
    18:09:06.000000000 -0300
+++ linux//mm/vmscan.c   2009-08-18 15:04:26.000000000 -0300
@@ -1918,6 +1918,40 @@
         return ret;
 }

+static pid_t stats_pid = -1;
+
+asmlinkage long sys_kpgsa_set_stats_pid(pid_t pid){
+        stats_pid = pid;
+        return 0;
+}
+
+int kpgsastatd(void *p){
+        struct task_struct * task, * first;
+        unsigned long last_jiffies = 0;
+
+        while(true){
+            if(jiffies - last_jiffies < 5000){
+                        yield();
+                        continue;
+                }else{
+                        last_jiffies = jiffies;
+                }
+                if(stats_pid != -1){
+                        printk("PGSA STATS FOR %d\n",
    stats_pid);
+                        //recorro todos los task del grupo
+                        task = first = find_task_by_pid(
    stats_pid);
+                        do{
+                                printk("PGSA pid %d mm %p
    pages %lu read-only %lu perc: %lu \n",task->pid,
+                                task->mm, get_mm_counter(
    task->mm,page_count),
+                                get_mm_counter(task->mm,
    ro_page_count), get_mm_counter(task->mm,ro_page_count)
    *100/get_mm_counter(task->mm,page_count) );
```

```
+                                       task = next_thread(task);
+                               }while(task != first);
+                       }
+               }
+               return 0;
+}
+
+
 static int __init kswapd_init(void)
 {
         int nid;
@@ -1925,10 +1959,15 @@
         swap_setup();
         for_each_node_state(nid, N_HIGH_MEMORY)
                 kswapd_run(nid);
+
         hotcpu_notifier(cpu_callback, 0);
+        kthread_run(kpgsastatd, NULL, "kpgsastatd");
+
         return 0;
 }

+
+
 module_init(kswapd_init)

 #ifdef CONFIG_NUMA
```

Apéndice D

Agente Inyector

En este anexo se presenta el código que se utilizó para las pruebas de cálculo de retraso de detección de errores. Este programa se encarga de registrarse a si mismo cómo agente de un proceso, inyectarlo y esperar la señal indicando que se detectó el error.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "/usr/src/linux-2.6.25.9-fabricio/include/asm-x86/
    unistd.h"

char * get_time ( char time_str[] ) {
    struct tm * local_time;
    struct timeval start_time;
    memset(time_str,'\0',74);
    gettimeofday ( & start_time, NULL );
    local_time = localtime ( & start_time.tv_sec );
    strftime ( time_str, 64, "%C%y-%m-%d %H:%M:%S",
        local_time );
    char swap[6];
    sprintf(swap,".%03d",start_time.tv_usec/1000);
    strncat(time_str,swap,4);
    return ( time_str );
}

void handler(int sig) {
        char str_time[74];
        printf("Error detectado: %s", get_time (str_time));
}

int main(int argc, char *argv[]) {
        char *endptr;
        char str_time[74];
```

```
        if (argc <= 3) {
                printf("Usage agente pid address new_value\n
                    ");
                return -1;
        }
        pid_t pid = atoi(argv[1]);

        //primero registro este proceso como agente del
            proceso a inyectar
        FILE * f = fopen("/proc/kpgsa/register","w");
        if(f){
                fprintf (f, "%d-%d", getpid(), pid);
                fclose(f);
                printf("proceso registrado.");
        }else{
                printf("error al abrir archivo.");
        }

        //seteo handler para la señal
        signal(SIGUNUSED, handler);

        //inyecto el error
        unsigned long address = strtoul(argv[2], &endptr,
            16);
        endptr = argv[3];
        unsigned char new_value = atoi(argv[3]);
        printf("%d %lx %c\n",pid,address,new_value);
        int ret = syscall(__NR_kpgsa_inyect, pid, address,
            new_value);

        printf("Error inyectado: %s", get_time (str_time));

        //duermo a esperar la senal
        sleep(10000);

        return ret;
}
```

Apéndice E

CosmicRayn

En este anexo se presenta el código utilizado para la inyección aleatoria de errores en un proceso mencionado en la sección 4.4. Este es el responsable de seleccionar los bytes a inyectar y que mascara se les va a aplicar a los mismos.

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include "/usr/src/linux-2.6.25.9-fabricio/include/asm-x86/
    unistd.h"

int main(int argc, char *argv[]) {
        unsigned char masks[] = {1,2,4,8,16,32,64,128};
        long starts[100];
        long ends[100];
        char *endptr;
        int count, i;
        struct timespec espera;
        espera.tv_sec = 1;
        espera.tv_nsec = 0;
        if (argc <= 2) {
                printf("Usage cosmicrayn pid count (start
                    end)*\n");
                return -1;
        }else if (argc <= 2 + atoi(argv[2])*2) {
                printf("Usage cosmicrayn pid count (start
                    end)*\n");
                return -1;
        }

        pid_t pid = atoi(argv[1]);
        count = atoi(argv[2]);
        for(i = 0;i < count;i++){
                starts[i] = strtoul(argv[2 + 2*i +1], &
                    endptr, 16);
                ends[i] = strtoul(argv[2 + 2*i +2], &endptr,
                     16);
        }
```

```
        srandom(time(NULL));
        int ret;
        while(1){
                unsigned char mask = masks[random() % 8];
                int index = random() % count;
                long diff = ends[index] - starts[index] + 1;
                long address = starts[index] + (random() %
                    diff);
                printf("address: %x\n",address);
                fflush(stdout);
                ret = syscall(__NR_kpgsa_inyect, pid,
                    address, mask);
                printf("Return value %d\n",ret);
                nanosleep(&espera,NULL);
        }
        return ret;
}
```

Apéndice F

Código para pruebas de inyección

En este anexo se presentan los fuentes de la aplicación `inyector` usada para invocar el mecanismo de inyección, así como el código que fue víctima de inyección de errores para las pruebas de la sección 3.9.1. Se presentarán 3 programas distintos, primero el simplista que realizaba solamente un loop e imprimía "Hola Mundo", luego el del programa que incrementaba una variable y por último que invocaba a 2 funciones distintas

F.1. Inyector

```
#include <stdio.h>
#include <unistd.h>
#include "/usr/src/linux-2.6.25.9/include/asm-x86/unistd.h"

int main(int argc, char *argv[]) {
        char *endptr;
        if (argc <= 3) {
                printf("Usage inyector pid address new_value
                    \n");
                return -1;
        }
        pid_t pid = atoi(argv[1]);
        unsigned long address = strtoul(argv[2], &endptr,
            16);
        endptr = argv[3];
        unsigned char new_value = atoi(argv[3]);
        printf("%d %lx %c\n",pid,address,new_value);
        int ret = syscall(__NR_kpgsa_inyect, pid, address,
            new_value);
        printf("Return value %d\n",ret);
        return ret;
}
```

F.2. Hola Mundo

```
#include <stdio.h>
#include <time.h>

const char holaWorld[] = "Hola Mundo\n";
```

```
int main() {
        while(1) {
                printf("%s",holaWorld);
                struct timespec espera;
                espera.tv_sec = 1;
                espera.tv_nsec = 0;
                nanosleep(&espera,NULL);
        }
        return 0;
}
```

F.3. Increment

```
#include <stdio.h>
#include <time.h>

int main() {
        unsigned int value = 0;
        struct timespec espera;
        espera.tv_sec = 1;
        espera.tv_nsec = 0;
        while(1) {
                value++;
                printf("value %u\n",value);
                nanosleep(&espera,NULL);
        }
        return 0;
}
```

F.4. Function

```
#include <stdio.h>
#include <time.h>

const char holaWorld[] = "Hola Mundo\n";

long func1() {
        printf("%s",holaWorld);
}

long func2() {
        printf("Adios Mundo\n");
}

int main() {
        struct timespec espera;
```

```
        espera.tv_sec = 1;
        espera.tv_nsec = 0;
        while(1) {
                func1();
                nanosleep(&espera,NULL);
                func2();
        }
        return 0;
}
```

Apéndice G

Salidas de las pruebas de performance

En este anexo se presentan las salidas de las distintas pruebas de performance explicadas en la sección 3.10. Primero se muestra la salida de las pruebas realizadas contra el Apache y MySQL, luego las salidas de ambas pruebas realizadas con POV-Ray.

Dentro de cada sección se listaran las salidas de cada una de las corridas realizadas separadas kernel utilizado o modo de ejecución del proyecto realizado.

G.1. Pruebas de performance AB

G.1.1. Kernel Vainilla.

G.1.1.1. Pasada 1. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
    zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
    apache.org/

Benchmarking 10.33.23.126 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.126
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1859.779878 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.38 [#/sec] (mean)
Time per request:       929.890 [ms] (mean)
```

```
Time per request:       185.978 [ms] (mean, across all
   concurrent requests)
Transfer rate:          29.02 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.7      0      35
Processing:   191  929 111.0    926    3979
Waiting:      191  923 111.6    920    3978
Total:        191  929 111.0    926    3979

Percentage of the requests served within a certain time (ms)
  50%    926
  66%    956
  75%    975
  80%    987
  90%   1022
  95%   1054
  98%   1091
  99%   1130
 100%   3979 (longest request)
```

G.1.1.2. Pasada 2. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.126 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.126
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1860.148648 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
```

```
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.38 [#/sec] (mean)
Time per request:       930.074 [ms] (mean)
Time per request:       186.015 [ms] (mean, across all
   concurrent requests)
Transfer rate:          29.02 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.9      0     129
Processing:   424  929 110.3    926    5929
Waiting:      424  922 111.2    920    5929
Total:        424  929 110.3    926    5929

Percentage of the requests served within a certain time (ms)
  50%    926
  66%    956
  75%    973
  80%    986
  90%   1017
  95%   1044
  98%   1076
  99%   1102
 100%   5929 (longest request)
```

G.1.1.3. Pasada 3. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.126 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.126
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
```

```
Time taken for tests:   1861.301903 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.37 [#/sec] (mean)
Time per request:       930.651 [ms] (mean)
Time per request:       186.130 [ms] (mean, across all
    concurrent requests)
Transfer rate:          29.00 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.5      0     116
Processing:   383  929  85.0    927    3964
Waiting:      382  923  86.0    921    3944
Total:        383  930  85.0    927    3964

Percentage of the requests served within a certain time (ms)
  50%    927
  66%    957
  75%    975
  80%    988
  90%   1020
  95%   1047
  98%   1082
  99%   1111
 100%   3964 (longest request)
```

G.1.1.4. Pasada 4. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
     apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
    zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
    apache.org/

Benchmarking 10.33.23.126 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.126
Server Port:            80
```

```
Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1861.432801 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.37 [#/sec] (mean)
Time per request:       930.716 [ms] (mean)
Time per request:       186.143 [ms] (mean, across all
   concurrent requests)
Transfer rate:          29.00 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.8      0     108
Processing:   373  930  76.0    929    3872
Waiting:      373  923  77.0    923    3871
Total:        373  930  76.0    929    3872

Percentage of the requests served within a certain time (ms)
  50%    929
  66%    957
  75%    974
  80%    986
  90%   1017
  95%   1045
  98%   1076
  99%   1102
 100%   3872 (longest request)
```

G.1.1.5. Pasada 5. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.126 (be patient)
```

```
Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.126
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1862.415 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.37 [#/sec] (mean)
Time per request:       931.000 [ms] (mean)
Time per request:       186.200 [ms] (mean, across all
    concurrent requests)
Transfer rate:          28.99 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.6      0     103
Processing:   529  930  69.3    929    1382
Waiting:      509  923  70.5    923    1373
Total:        529  930  69.4    929    1382

Percentage of the requests served within a certain time (ms)
  50%    929
  66%    958
  75%    975
  80%    987
  90%   1017
  95%   1044
  98%   1079
  99%   1100
 100%   1382 (longest request)
```

G.1.2. Kernel modificado, modo 142.

G.1.2.1. Pasada 1. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
    zeustech.net/
```

```
Copyright 2006 The Apache Software Foundation, http://www.
    apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1920.663174 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.21 [#/sec] (mean)
Time per request:       960.332 [ms] (mean)
Time per request:       192.066 [ms] (mean, across all
    concurrent requests)
Transfer rate:          28.10 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.3      0      83
Processing:   401  959 201.5    970    4074
Waiting:      361  949 201.8    960    3948
Total:        401  959 201.5    970    4075

Percentage of the requests served within a certain time (ms)
  50%    970
  66%   1031
  75%   1072
  80%   1100
  90%   1175
  95%   1242
  98%   1333
  99%   1395
 100%   4075 (longest request)
```

G.1.2.2. Pasada 2. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1913.988384 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.22 [#/sec] (mean)
Time per request:       956.994 [ms] (mean)
Time per request:       191.399 [ms] (mean, across all
   concurrent requests)
Transfer rate:          28.20 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.3      0      73
Processing:   389  956 181.8    968    1686
Waiting:      385  946 182.4    957    1686
Total:        389  956 181.8    968    1686

Percentage of the requests served within a certain time (ms)
  50%    968
  66%   1026
  75%   1068
  80%   1094
  90%   1168
  95%   1238
  98%   1329
```

```
 99%    1393
100%    1686 (longest request)
```

G.1.2.3. Pasada 3. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1917.275277 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.22 [#/sec] (mean)
Time per request:       958.638 [ms] (mean)
Time per request:       191.728 [ms] (mean, across all
   concurrent requests)
Transfer rate:          28.15 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.6      0      88
Processing:   395  957 189.4    969    3237
Waiting:      361  947 190.1    958    3237
Total:        395  957 189.4    969    3237

Percentage of the requests served within a certain time (ms)
  50%     969
  66%    1026
  75%    1065
```

```
  80%    1091
  90%    1170
  95%    1238
  98%    1323
  99%    1378
 100%    3237 (longest request)
```

G.1.2.4. Pasada 4. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
     apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
    zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
    apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1924.668198 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.20 [#/sec] (mean)
Time per request:       962.334 [ms] (mean)
Time per request:       192.467 [ms] (mean, across all
    concurrent requests)
Transfer rate:          28.04 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.8      0     101
Processing:   387  961 193.3    974    3398
Waiting:      368  950 194.1    962    3398
Total:        387  961 193.3    974    3398
```

```
Percentage of the requests served within a certain time (ms)
  50%    974
  66%   1035
  75%   1075
  80%   1101
  90%   1180
  95%   1247
  98%   1335
  99%   1409
 100%   3398 (longest request)
```

G.1.2.5. Pasada 5. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
     apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
    zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
    apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1922.500066 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.20 [#/sec] (mean)
Time per request:       961.250 [ms] (mean)
Time per request:       192.250 [ms] (mean, across all
    concurrent requests)
Transfer rate:          28.07 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.1      0      58
```

```
Processing:    372   960  183.0     974     1819
Waiting:       338   949  184.0     963     1819
Total:         372   960  183.1     975     1819

Percentage of the requests served within a certain time (ms)
  50%     975
  66%    1035
  75%    1076
  80%    1103
  90%    1176
  95%    1241
  98%    1318
  99%    1374
 100%    1819 (longest request)
```

G.1.3. Kernel modificado, modo 146.

G.1.3.1. Pasada 1. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
     apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
    zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
    apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1873.324378 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.34 [#/sec] (mean)
Time per request:       936.662 [ms] (mean)
Time per request:       187.332 [ms] (mean, across all
    concurrent requests)
```

```
Transfer rate:          28.81 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.2      0      72
Processing:   443  935 109.3    932    4032
Waiting:      442  929 109.3    925    3958
Total:        443  935 109.3    932    4037

Percentage of the requests served within a certain time (ms)
  50%    932
  66%    961
  75%    979
  80%    992
  90%   1025
  95%   1057
  98%   1093
  99%   1123
 100%   4037 (longest request)
```

G.1.3.2. Pasada 2. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1870.570424 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
```

```
Requests per second:    5.35 [#/sec] (mean)
Time per request:       935.285 [ms] (mean)
Time per request:       187.057 [ms] (mean, across all
   concurrent requests)
Transfer rate:          28.85 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.8      0      51
Processing:   449  934  91.8    931    3377
Waiting:      449  928  92.5    925    3375
Total:        449  934  91.8    931    3377

Percentage of the requests served within a certain time (ms)
  50%    931
  66%    961
  75%    980
  80%    993
  90%   1027
  95%   1058
  98%   1097
  99%   1129
 100%   3377 (longest request)
```

G.1.3.3. Pasada 3. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1870.223075 seconds
Complete requests:      10000
```

```
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.35 [#/sec] (mean)
Time per request:       935.112 [ms] (mean)
Time per request:       187.022 [ms] (mean, across all
   concurrent requests)
Transfer rate:          28.86 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.3      0      87
Processing:   562  934  71.1    933    1475
Waiting:      562  927  72.0    926    1452
Total:        562  934  71.1    933    1475

Percentage of the requests served within a certain time (ms)
  50%    933
  66%    962
  75%    979
  80%    991
  90%   1023
  95%   1051
  98%   1086
  99%   1111
 100%   1475 (longest request)
```

G.1.3.4. Pasada 4. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes
```

```
Concurrency Level:      5
Time taken for tests:   1876.551796 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.33 [#/sec] (mean)
Time per request:       938.276 [ms] (mean)
Time per request:       187.655 [ms] (mean, across all
   concurrent requests)
Transfer rate:          28.76 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.8      0     134
Processing:   569  937  83.9    935    3000
Waiting:      568  930  84.8    928    3000
Total:        569  937  84.0    935    3000

Percentage of the requests served within a certain time (ms)
  50%    935
  66%    963
  75%    982
  80%    995
  90%   1027
  95%   1056
  98%   1092
  99%   1121
 100%   3000 (longest request)
```

G.1.3.5. Pasada 5. .

```
This is ApacheBench, Version 2.0.40-dev <$Revision: 1.146 $>
    apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.
   zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.
   apache.org/

Benchmarking 10.33.23.141 (be patient)


Server Software:        Apache/2.2.8
Server Hostname:        10.33.23.141
```

```
Server Port:            80

Document Path:          /wordpress/
Document Length:        5276 bytes

Concurrency Level:      5
Time taken for tests:   1882.199058 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      55270000 bytes
HTML transferred:       52760000 bytes
Requests per second:    5.31 [#/sec] (mean)
Time per request:       941.100 [ms] (mean)
Time per request:       188.220 [ms] (mean, across all
   concurrent requests)
Transfer rate:          28.68 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   2.0      0     138
Processing:   562  940  97.9    937    3574
Waiting:      562  932  98.7    930    3573
Total:        562  940  97.9    937    3574

Percentage of the requests served within a certain time (ms)
  50%    937
  66%    964
  75%    983
  80%    994
  90%   1028
  95%   1056
  98%   1090
  99%   1117
 100%   3574 (longest request)
```

G.2. Pruebas de performance POV-Ray Benchmark

G.2.1. Kernel Vainilla.

G.2.1.1. Pasada 1. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
```

```
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 2000K tokens
  0:00:02 Creating bounding slabs
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:00:48 Sorting photons 0 of 63336
  0:00:48 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:41:04 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------
Pixels:          147840   Samples:          575936   Smpls/
    Pxl: 3.90
Rays:           1861058   Saved:             21855   Max
    Level: 12/12
----------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------
Box                           79994791         9417290
    11.77
Cone/Cylinder                 78720800         6627468
    8.42
CSG Intersection             170489517        58806730
    34.49
CSG Merge                       863397           34750
    4.02
Fractal                        1782085          105385
    5.91
```

```
Height Field                    3586026          102531
   2.86
Height Field Box                3586026          689676
   19.23
Height Field Triangle           3262054          105733
   3.24
Height Field Block              5712599         1682572
   29.45
Height Field Cell              22619393         1790282
   7.91
Isosurface                     11950852          733084
   6.13
Isosurface Container           12409949        11951336
   96.30
Isosurface Cache                 144545           43880
   30.36
Mesh                           15388221           64675
   0.42
Plane                          92234422         1295142
   1.40
Sphere                        281613813       175805479
   62.43
Superellipsoid                   639747           44455
   6.95
Torus                           2992390          423155
   14.14
Torus Bound                     2992390          489319
   16.35
True Type Font                   790747           80960
   10.24
Clipping Object                 2587261         1540019
   59.52
Bounding Box                  521371907       149086904
   28.60
Vista Buffer                   22502747        12948149
   57.54
----------------------------------------------------------------------------
Isosurface roots:           11945843
Function VM calls:         172379889
----------------------------------------------------------------------------
Roots tested:                 489319   eliminated:
                 278583
Calls to Noise:           4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
```

```
Media Intervals:           39692199   Media Samples:
           358081477 (9.02)
Shadow Ray Tests:         128818521   Succeeded:
                52507650
Reflected Rays:              224359   Total Internal:
               2374
Refracted Rays:              143178
Transmitted Rays:            621485
```

```
Number of photons shot:          74025
Surface photons stored:          63336
Priority queue insert:         1509036
Priority queue remove:          151992
Gather function called:         672370
```

```
Smallest Alloc:                    9 bytes
Largest  Alloc:              1440008 bytes
Peak memory used:            5509258 bytes
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:    0 hours  0 minutes 48 seconds (48 seconds)
  Render Time:    0 hours 41 minutes  4 seconds (2464 seconds
     )
  Total Time:     0 hours 41 minutes 54 seconds (2514 seconds
     )
```

G.2.1.2. Pasada 2. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1443K tokens
  0:00:02 Creating bounding slabs
```

```
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:00:48 Sorting photons 0 of 63336
  0:00:48 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:41:09 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:          147840   Samples:          575936   Smpls/
   Pxl: 3.90
Rays:           1861058   Saved:             21855   Max
   Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection           Tests        Succeeded
   Percentage
----------------------------------------------------------------------------
Box                            79994791          9417290
   11.77
Cone/Cylinder                  78720800          6627468
   8.42
CSG Intersection              170489517         58806730
   34.49
CSG Merge                        863397            34750
   4.02
Fractal                         1782085           105385
   5.91
Height Field                    3586026           102531
   2.86
Height Field Box                3586026           689676
   19.23
Height Field Triangle           3262054           105733
   3.24
Height Field Block              5712599          1682572
   29.45
Height Field Cell              22619393          1790282
   7.91
Isosurface                     11950852           733084
   6.13
Isosurface Container           12409949         11951336
   96.30
```

```
Isosurface Cache                  144545           43880
   30.36
Mesh                            15388221           64675
   0.42
Plane                           92234422         1295142
   1.40
Sphere                         281613813       175805479
   62.43
Superellipsoid                    639747           44455
   6.95
Torus                            2992390          423155
   14.14
Torus Bound                      2992390          489319
   16.35
True Type Font                    790747           80960
   10.24
Clipping Object                  2587261         1540019
   59.52
Bounding Box                   521371907       149086904
   28.60
Vista Buffer                    22502747        12948149
   57.54
----------------------------------------------------------------------
Isosurface roots:            11945843
Function VM calls:          172379889
----------------------------------------------------------------------
Roots tested:                  489319    eliminated:
               278583
Calls to Noise:            4832954757    Calls to DNoise:
   2620323810
----------------------------------------------------------------------
Media Intervals:             39692199    Media Samples:
         358081477 (9.02)
Shadow Ray Tests:           128818521    Succeeded:
               52507650
Reflected Rays:                224359    Total Internal:
              2374
Refracted Rays:                143178
Transmitted Rays:              621485
----------------------------------------------------------------------
Number of photons shot:              74025
Surface photons stored:              63336
Priority queue insert:             1509036
Priority queue remove:              151992
Gather function called:             672370
```

```
Smallest Alloc:                    9 bytes
Largest  Alloc:              1440008 bytes
Peak memory used:            5509258 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:   0 hours  0 minutes 48 seconds (48 seconds)
  Render Time:   0 hours 41 minutes  9 seconds (2469 seconds
     )
  Total Time:    0 hours 41 minutes 59 seconds (2519 seconds
     )
```

G.2.1.3. Pasada 3. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 290K tokens
  0:00:01 Creating bounding slabs
  0:00:01 Creating vista buffer
  0:00:01 Creating light buffers
  0:00:01 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:00:49 Sorting photons 0 of 63336
  0:00:49 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:41:07 Done Tracing
Render Statistics
Image Resolution 384 x 384
```

```
Pixels:          147840   Samples:          575936   Smpls/
   Pxl: 3.90
```

```
Rays:             1861058    Saved:            21855    Max
    Level: 12/12
----------------------------------------------------------------
Ray->Shape Intersection            Tests        Succeeded
    Percentage
----------------------------------------------------------------
Box                             79994791          9417290
    11.77
Cone/Cylinder                   78720800          6627468
    8.42
CSG Intersection               170489517         58806730
    34.49
CSG Merge                         863397            34750
    4.02
Fractal                          1782085           105385
    5.91
Height Field                     3586026           102531
    2.86
Height Field Box                 3586026           689676
    19.23
Height Field Triangle            3262054           105733
    3.24
Height Field Block               5712599          1682572
    29.45
Height Field Cell               22619393          1790282
    7.91
Isosurface                      11950852           733084
    6.13
Isosurface Container            12409949         11951336
    96.30
Isosurface Cache                  144545            43880
    30.36
Mesh                            15388221            64675
    0.42
Plane                           92234422          1295142
    1.40
Sphere                         281613813        175805479
    62.43
Superellipsoid                    639747            44455
    6.95
Torus                            2992390           423155
    14.14
Torus Bound                      2992390           489319
    16.35
```

```
True Type Font                  790747          80960
   10.24
Clipping Object                2587261        1540019
   59.52
Bounding Box                 521371907      149086904
   28.60
Vista Buffer                  22502747       12948149
   57.54
----------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------
Roots tested:                  489319   eliminated:
                  278583
Calls to Noise:            4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------
Media Intervals:             39692199   Media Samples:
           358081477 (9.02)
Shadow Ray Tests:           128818521   Succeeded:
                 52507650
Reflected Rays:                224359   Total Internal:
               2374
Refracted Rays:                143178
Transmitted Rays:              621485
----------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------
Smallest Alloc:                        9 bytes
Largest  Alloc:                  1440008 bytes
Peak memory used:                5509258 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  1 seconds (1 seconds)
  Photon Time:   0 hours  0 minutes 49 seconds (49 seconds)
  Render Time:   0 hours 41 minutes  7 seconds (2467 seconds
     )
  Total Time:    0 hours 41 minutes 57 seconds (2517 seconds
     )
```

G.2.1.4. Pasada 4. .

```
Running standard POV-Ray benchmark version 1.02
```

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1995K tokens
  0:00:02 Creating bounding slabs
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:00:48 Sorting photons 0 of 63336
  0:00:48 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:41:03 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:           147840   Samples:          575936   Smpls/
   Pxl: 3.90
Rays:            1861058   Saved:             21855   Max
   Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
   Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
   11.77
Cone/Cylinder                 78720800         6627468
   8.42
CSG Intersection             170489517        58806730
   34.49
```

```
CSG Merge                          863397          34750
    4.02
Fractal                           1782085         105385
    5.91
Height Field                      3586026         102531
    2.86
Height Field Box                  3586026         689676
    19.23
Height Field Triangle             3262054         105733
    3.24
Height Field Block                5712599        1682572
    29.45
Height Field Cell                22619393        1790282
    7.91
Isosurface                       11950852         733084
    6.13
Isosurface Container             12409949       11951336
    96.30
Isosurface Cache                   144545          43880
    30.36
Mesh                             15388221          64675
    0.42
Plane                            92234422        1295142
    1.40
Sphere                          281613813      175805479
    62.43
Superellipsoid                     639747          44455
    6.95
Torus                             2992390         423155
    14.14
Torus Bound                       2992390         489319
    16.35
True Type Font                     790747          80960
    10.24
Clipping Object                   2587261        1540019
    59.52
Bounding Box                    521371907      149086904
    28.60
Vista Buffer                     22502747       12948149
    57.54
----------------------------------------------------------------------------
Isosurface roots:           11945843
Function VM calls:         172379889
----------------------------------------------------------------------------
```

```
Roots tested:              489319   eliminated:
               278583
Calls to Noise:        4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:         39692199   Media Samples:
          358081477 (9.02)
Shadow Ray Tests:       128818521   Succeeded:
               52507650
Reflected Rays:            224359   Total Internal:
              2374
Refracted Rays:            143178
Transmitted Rays:          621485
----------------------------------------------------------------------------
Number of photons shot:          74025
Surface photons stored:          63336
Priority queue insert:         1509036
Priority queue remove:          151992
Gather function called:         672370
----------------------------------------------------------------------------
Smallest Alloc:                    9 bytes
Largest  Alloc:              1440008 bytes
Peak memory used:            5509258 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:   0 hours  0 minutes 48 seconds (48 seconds)
  Render Time:   0 hours 41 minutes  3 seconds (2463 seconds
     )
  Total Time:    0 hours 41 minutes 53 seconds (2513 seconds
     )
```

G.2.1.5. Pasada 5. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .
```

```
  0:00:00 Parsing
  0:00:01 Parsing 1232K tokens
  0:00:01 Creating bounding slabs
  0:00:01 Creating vista buffer
  0:00:01 Creating light buffers
  0:00:01 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:00:48 Sorting photons 0 of 63336
  0:00:48 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:41:06 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:          147840   Samples:           575936   Smpls/
   Pxl: 3.90
Rays:           1861058   Saved:              21855   Max
   Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
   Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
   11.77
Cone/Cylinder                 78720800         6627468
   8.42
CSG Intersection             170489517        58806730
   34.49
CSG Merge                       863397           34750
   4.02
Fractal                        1782085          105385
   5.91
Height Field                   3586026          102531
   2.86
Height Field Box               3586026          689676
   19.23
Height Field Triangle          3262054          105733
   3.24
Height Field Block             5712599         1682572
   29.45
Height Field Cell             22619393         1790282
   7.91
```

```
Isosurface                     11950852        733084
   6.13
Isosurface Container           12409949      11951336
   96.30
Isosurface Cache                 144545         43880
   30.36
Mesh                           15388221         64675
   0.42
Plane                          92234422       1295142
   1.40
Sphere                        281613813     175805479
   62.43
Superellipsoid                   639747         44455
   6.95
Torus                           2992390        423155
   14.14
Torus Bound                     2992390        489319
   16.35
True Type Font                   790747         80960
   10.24
Clipping Object                 2587261       1540019
   59.52
Bounding Box                  521371907     149086904
   28.60
Vista Buffer                   22502747      12948149
   57.54
----------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------
Roots tested:                489319   eliminated:
                 278583
Calls to Noise:          4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------
Media Intervals:           39692199   Media Samples:
           358081477 (9.02)
Shadow Ray Tests:         128818521   Succeeded:
                 52507650
Reflected Rays:              224359   Total Internal:
                2374
Refracted Rays:              143178
Transmitted Rays:            621485
----------------------------------------------------------
Number of photons shot:           74025
```

```
Surface photons stored:          63336
Priority queue insert:         1509036
Priority queue remove:          151992
Gather function called:         672370
```

```
Smallest Alloc:                      9 bytes
Largest  Alloc:                1440008 bytes
Peak memory used:              5509258 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  1 seconds (1 seconds)
  Photon Time:   0 hours  0 minutes 49 seconds (49 seconds)
  Render Time:   0 hours 41 minutes  6 seconds (2466 seconds
     )
  Total Time:    0 hours 41 minutes 56 seconds (2516 seconds
     )
```

G.2.2. Kernel modificado, modo 142.

G.2.2.1. Pasada 1. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1553K tokens
  0:00:02 Creating bounding slabs
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:10 Sorting photons 0 of 63336
  0:01:10 Sorting photons 63335 of 63336
  0:01:11 Sorting photons 15832 of 63336
```

```
  0:00:00 Rendering line 1 of 384, 0 supersamples
  1:01:40 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:          147840   Samples:          575936   Smpls/
   Pxl: 3.90
Rays:           1861058   Saved:             21855   Max
   Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
   Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
   11.77
Cone/Cylinder                 78720800         6627468
   8.42
CSG Intersection             170489517        58806730
   34.49
CSG Merge                       863397           34750
   4.02
Fractal                        1782085          105385
   5.91
Height Field                   3586026          102531
   2.86
Height Field Box               3586026          689676
   19.23
Height Field Triangle          3262054          105733
   3.24
Height Field Block             5712599         1682572
   29.45
Height Field Cell             22619393         1790282
   7.91
Isosurface                    11950852          733084
   6.13
Isosurface Container          12409949        11951336
   96.30
Isosurface Cache                144545           43880
   30.36
Mesh                          15388221           64675
   0.42
Plane                         92234422         1295142
   1.40
Sphere                       281613813       175805479
   62.43
```

```
Superellipsoid                  639747          44455
   6.95
Torus                          2992390         423155
   14.14
Torus Bound                    2992390         489319
   16.35
True Type Font                  790747          80960
   10.24
Clipping Object                2587261        1540019
   59.52
Bounding Box                 521371907      149086904
   28.60
Vista Buffer                  22502747       12948149
   57.54
----------------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------------
Roots tested:                489319   eliminated:
                  278583
Calls to Noise:          4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:           39692199   Media Samples:
            358081477 (9.02)
Shadow Ray Tests:         128818521   Succeeded:
                  52507650
Reflected Rays:              224359   Total Internal:
                 2374
Refracted Rays:              143178
Transmitted Rays:            621485
----------------------------------------------------------------------------
Number of photons shot:          74025
Surface photons stored:          63336
Priority queue insert:         1509036
Priority queue remove:          151992
Gather function called:         672370
----------------------------------------------------------------------------
Smallest Alloc:                    9 bytes
Largest  Alloc:              1440008 bytes
Peak memory used:            5509225 bytes
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:    0 hours  1 minutes 11 seconds (71 seconds)
```

```
  Render Time:    1 hours  1 minutes 40 seconds (3700 seconds
     )
  Total Time:     1 hours  2 minutes 53 seconds (3773 seconds
     )
```

G.2.2.2. Pasada 2. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 880K tokens
  0:00:03 Creating bounding slabs
  0:00:03 Creating vista buffer
  0:00:03 Creating light buffers
  0:00:03 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:12 Sorting photons 0 of 63336
  0:01:12 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  1:02:17 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:           147840    Samples:          575936    Smpls/
   Pxl: 3.90
Rays:            1861058    Saved:             21855    Max
   Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests        Succeeded
   Percentage
----------------------------------------------------------------------------
```

```
Box                          79994791       9417290
   11.77
Cone/Cylinder                78720800       6627468
   8.42
CSG Intersection            170489517      58806730
   34.49
CSG Merge                      863397         34750
   4.02
Fractal                       1782085        105385
   5.91
Height Field                  3586026        102531
   2.86
Height Field Box              3586026        689676
   19.23
Height Field Triangle         3262054        105733
   3.24
Height Field Block            5712599       1682572
   29.45
Height Field Cell            22619393       1790282
   7.91
Isosurface                   11950852        733084
   6.13
Isosurface Container         12409949      11951336
   96.30
Isosurface Cache               144545         43880
   30.36
Mesh                         15388221         64675
   0.42
Plane                        92234422       1295142
   1.40
Sphere                      281613813     175805479
   62.43
Superellipsoid                 639747         44455
   6.95
Torus                         2992390        423155
   14.14
Torus Bound                   2992390        489319
   16.35
True Type Font                 790747         80960
   10.24
Clipping Object               2587261       1540019
   59.52
Bounding Box                521371907     149086904
   28.60
```

```
Vista Buffer                   22502747        12948149
   57.54
----------------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------------
Roots tested:                489319   eliminated:
                 278583
Calls to Noise:          4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:           39692199   Media Samples:
            358081477 (9.02)
Shadow Ray Tests:         128818521   Succeeded:
                52507650
Reflected Rays:              224359   Total Internal:
               2374
Refracted Rays:              143178
Transmitted Rays:            621485
----------------------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------------------
Smallest Alloc:                     9 bytes
Largest  Alloc:               1440008 bytes
Peak memory used:             5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  3 seconds (3 seconds)
  Photon Time:   0 hours  1 minutes 12 seconds (72 seconds)
  Render Time:   1 hours  2 minutes 17 seconds (3737 seconds
     )
  Total Time:    1 hours  3 minutes 32 seconds (3812 seconds
     )
```

G.2.2.3. Pasada 3. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
```

```
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 10K tokens
  0:00:02 Parsing 300K tokens
  0:00:03 Parsing 1843K tokens
  0:00:03 Creating bounding slabs
  0:00:03 Creating vista buffer
  0:00:03 Creating light buffers
  0:00:03 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:13 Sorting photons 0 of 63336
  0:01:13 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  1:03:25 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:          147840   Samples:          575936   Smpls/
    Pxl: 3.90
Rays:           1861058   Saved:             21855   Max
    Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
    11.77
Cone/Cylinder                 78720800         6627468
    8.42
CSG Intersection             170489517        58806730
    34.49
CSG Merge                       863397           34750
    4.02
Fractal                        1782085          105385
    5.91
Height Field                   3586026          102531
    2.86
```

```
Height Field Box                 3586026          689676
   19.23
Height Field Triangle            3262054          105733
   3.24
Height Field Block               5712599         1682572
   29.45
Height Field Cell               22619393         1790282
   7.91
Isosurface                      11950852          733084
   6.13
Isosurface Container            12409949        11951336
   96.30
Isosurface Cache                  144545           43880
   30.36
Mesh                            15388221           64675
   0.42
Plane                           92234422         1295142
   1.40
Sphere                         281613813       175805479
   62.43
Superellipsoid                    639747           44455
   6.95
Torus                            2992390          423155
   14.14
Torus Bound                      2992390          489319
   16.35
True Type Font                    790747           80960
   10.24
Clipping Object                  2587261         1540019
   59.52
Bounding Box                   521371907       149086904
   28.60
Vista Buffer                    22502747        12948149
   57.54
----------------------------------------------------------------------------
Isosurface roots:            11945843
Function VM calls:          172379889
----------------------------------------------------------------------------
Roots tested:                  489319   eliminated:
                 278583
Calls to Noise:            4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:             39692199   Media Samples:
           358081477 (9.02)
```

```
Shadow Ray Tests:          128818521   Succeeded:
                52507650
Reflected Rays:               224359   Total Internal:
               2374
Refracted Rays:               143178
Transmitted Rays:             621485
----------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------
Smallest Alloc:                        9 bytes
Largest  Alloc:                  1440008 bytes
Peak memory used:                5509225 bytes
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes  3 seconds (3 seconds)
  Photon Time:    0 hours  1 minutes 13 seconds (73 seconds)
  Render Time:    1 hours  3 minutes 25 seconds (3805 seconds
     )
  Total Time:     1 hours  4 minutes 41 seconds (3881 seconds
     )
```

G.2.2.4. Pasada 4. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 795K tokens
  0:00:03 Creating bounding slabs
  0:00:03 Creating vista buffer
  0:00:03 Creating light buffers
```

```
  0:00:03 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:12 Sorting photons 0 of 63336
  0:01:12 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  1:02:25 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:           147840   Samples:          575936   Smpls/
    Pxl: 3.90
Rays:            1861058   Saved:             21855   Max
    Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
    11.77
Cone/Cylinder                 78720800         6627468
    8.42
CSG Intersection             170489517        58806730
    34.49
CSG Merge                       863397           34750
    4.02
Fractal                        1782085          105385
    5.91
Height Field                   3586026          102531
    2.86
Height Field Box               3586026          689676
    19.23
Height Field Triangle          3262054          105733
    3.24
Height Field Block             5712599         1682572
    29.45
Height Field Cell             22619393         1790282
    7.91
Isosurface                    11950852          733084
    6.13
Isosurface Container          12409949        11951336
    96.30
Isosurface Cache                144545           43880
    30.36
```

```
Mesh                            15388221            64675
   0.42
Plane                           92234422          1295142
   1.40
Sphere                         281613813        175805479
   62.43
Superellipsoid                    639747            44455
   6.95
Torus                            2992390           423155
   14.14
Torus Bound                      2992390           489319
   16.35
True Type Font                    790747            80960
   10.24
Clipping Object                  2587261          1540019
   59.52
Bounding Box                   521371907        149086904
   28.60
Vista Buffer                    22502747         12948149
   57.54
----------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------
Roots tested:                489319   eliminated:
              278583
Calls to Noise:          4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------
Media Intervals:           39692199   Media Samples:
          358081477 (9.02)
Shadow Ray Tests:         128818521   Succeeded:
             52507650
Reflected Rays:              224359   Total Internal:
             2374
Refracted Rays:              143178
Transmitted Rays:            621485
----------------------------------------------------------------------
Number of photons shot:           74025
Surface photons stored:           63336
Priority queue insert:          1509036
Priority queue remove:           151992
Gather function called:          672370
----------------------------------------------------------------------
Smallest Alloc:                       9 bytes
```

```
Largest  Alloc:             1440008 bytes
Peak memory used:           5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  3 seconds (3 seconds)
  Photon Time:   0 hours  1 minutes 12 seconds (72 seconds)
  Render Time:   1 hours  2 minutes 25 seconds (3745 seconds
    )
  Total Time:    1 hours  3 minutes 40 seconds (3820 seconds
    )
```

G.2.2.5. Pasada 5. .

```
Running standard POV–Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686–pc–linux–gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV–Ray Team(tm) is not responsible for supporting this
     version.
POV–Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991–2003 Persistence of Vision Team
Copyright 2003–2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1135K tokens
  0:00:03 Creating bounding slabs
  0:00:03 Creating vista buffer
  0:00:03 Creating light buffers
  0:00:03 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:12 Sorting photons 0 of 63336
  0:01:12 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  1:02:08 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------
Pixels:          147840    Samples:          575936    Smpls/
   Pxl: 3.90
```

```
Rays:            1861058    Saved:            21855     Max
    Level: 12/12
----------------------------------------------------------------
Ray->Shape Intersection          Tests        Succeeded
    Percentage
----------------------------------------------------------------
Box                           79994791          9417290
    11.77
Cone/Cylinder                 78720800          6627468
    8.42
CSG Intersection             170489517         58806730
    34.49
CSG Merge                       863397            34750
    4.02
Fractal                        1782085           105385
    5.91
Height Field                   3586026           102531
    2.86
Height Field Box               3586026           689676
    19.23
Height Field Triangle          3262054           105733
    3.24
Height Field Block             5712599          1682572
    29.45
Height Field Cell             22619393          1790282
    7.91
Isosurface                    11950852           733084
    6.13
Isosurface Container          12409949         11951336
    96.30
Isosurface Cache                144545            43880
    30.36
Mesh                          15388221            64675
    0.42
Plane                         92234422          1295142
    1.40
Sphere                       281613813        175805479
    62.43
Superellipsoid                  639747            44455
    6.95
Torus                          2992390           423155
    14.14
Torus Bound                    2992390           489319
    16.35
```

```
True Type Font                    790747            80960
   10.24
Clipping Object                  2587261          1540019
   59.52
Bounding Box                   521371907        149086904
   28.60
Vista Buffer                    22502747         12948149
   57.54
----------------------------------------------------------------------------
Isosurface roots:           11945843
Function VM calls:         172379889
----------------------------------------------------------------------------
Roots tested:                 489319   eliminated:
                278583
Calls to Noise:           4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:            39692199   Media Samples:
         358081477 (9.02)
Shadow Ray Tests:          128818521   Succeeded:
                52507650
Reflected Rays:               224359   Total Internal:
               2374
Refracted Rays:               143178
Transmitted Rays:             621485
----------------------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------------------
Smallest Alloc:                        9 bytes
Largest  Alloc:                  1440008 bytes
Peak memory used:                5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  3 seconds (3 seconds)
  Photon Time:   0 hours  1 minutes 12 seconds (72 seconds)
  Render Time:   1 hours  2 minutes  8 seconds (3728 seconds
     )
  Total Time:    1 hours  3 minutes 23 seconds (3803 seconds
     )
```

G.2.3. Kernel modificado, modo 146.

G.2.3.1. Pasada 1. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
    i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
      version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1253K tokens
  0:00:02 Creating bounding slabs
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:08 Sorting photons 0 of 63336
  0:01:08 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:59:04 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:            147840   Samples:           575936   Smpls/
    Pxl: 3.90
Rays:             1861058   Saved:              21855   Max
    Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
    11.77
Cone/Cylinder                 78720800         6627468
    8.42
CSG Intersection             170489517        58806730
    34.49
```

```
CSG Merge                      863397          34750
    4.02
Fractal                       1782085         105385
    5.91
Height Field                  3586026         102531
    2.86
Height Field Box              3586026         689676
    19.23
Height Field Triangle         3262054         105733
    3.24
Height Field Block            5712599        1682572
    29.45
Height Field Cell            22619393        1790282
    7.91
Isosurface                   11950852         733084
    6.13
Isosurface Container         12409949       11951336
    96.30
Isosurface Cache               144545          43880
    30.36
Mesh                         15388221          64675
    0.42
Plane                        92234422        1295142
    1.40
Sphere                      281613813      175805479
    62.43
Superellipsoid                 639747          44455
    6.95
Torus                         2992390         423155
    14.14
Torus Bound                   2992390         489319
    16.35
True Type Font                 790747          80960
    10.24
Clipping Object               2587261        1540019
    59.52
Bounding Box                521371907      149086904
    28.60
Vista Buffer                 22502747       12948149
    57.54
----------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------
```

```
Roots tested:              489319   eliminated:
               278583
Calls to Noise:        4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:         39692199   Media Samples:
           358081477 (9.02)
Shadow Ray Tests:       128818521   Succeeded:
              52507650
Reflected Rays:            224359   Total Internal:
             2374
Refracted Rays:            143178
Transmitted Rays:          621485
----------------------------------------------------------------------------
Number of photons shot:          74025
Surface photons stored:          63336
Priority queue insert:         1509036
Priority queue remove:          151992
Gather function called:         672370
----------------------------------------------------------------------------
Smallest Alloc:                  9 bytes
Largest  Alloc:            1440008 bytes
Peak memory used:          5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:   0 hours  1 minutes  8 seconds (68 seconds)
  Render Time:   0 hours 59 minutes  4 seconds (3544 seconds
     )
  Total Time:    1 hours  0 minutes 14 seconds (3614 seconds
     )
```

G.2.3.2. Pasada 2. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .
```

```
  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1558K tokens
  0:00:02 Creating bounding slabs
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:08 Sorting photons 0 of 63336
  0:01:08 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:58:58 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------
Pixels:          147840   Samples:          575936   Smpls/
    Pxl: 3.90
Rays:           1861058   Saved:             21855   Max
    Level: 12/12
----------------------------------------------------------------------
Ray->Shape Intersection            Tests         Succeeded
    Percentage
----------------------------------------------------------------------
Box                             79994791           9417290
    11.77
Cone/Cylinder                   78720800           6627468
    8.42
CSG Intersection               170489517          58806730
    34.49
CSG Merge                         863397             34750
    4.02
Fractal                          1782085            105385
    5.91
Height Field                     3586026            102531
    2.86
Height Field Box                 3586026            689676
    19.23
Height Field Triangle            3262054            105733
    3.24
Height Field Block               5712599           1682572
    29.45
```

```
Height Field Cell            22619393        1790282
    7.91
Isosurface                   11950852         733084
    6.13
Isosurface Container         12409949       11951336
    96.30
Isosurface Cache               144545          43880
    30.36
Mesh                         15388221          64675
    0.42
Plane                        92234422        1295142
    1.40
Sphere                      281613813      175805479
    62.43
Superellipsoid                 639747          44455
    6.95
Torus                         2992390         423155
    14.14
Torus Bound                   2992390         489319
    16.35
True Type Font                 790747          80960
    10.24
Clipping Object               2587261        1540019
    59.52
Bounding Box                521371907      149086904
    28.60
Vista Buffer                 22502747       12948149
    57.54
----------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------
Roots tested:                489319    eliminated:
              278583
Calls to Noise:          4832954757    Calls to DNoise:
    2620323810
----------------------------------------------------------------------
Media Intervals:           39692199    Media Samples:
          358081477 (9.02)
Shadow Ray Tests:         128818521    Succeeded:
              52507650
Reflected Rays:              224359    Total Internal:
             2374
Refracted Rays:              143178
Transmitted Rays:            621485
```

```
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
```

```
Smallest Alloc:                        9 bytes
Largest  Alloc:                  1440008 bytes
Peak memory used:                5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:   0 hours  1 minutes  8 seconds (68 seconds)
  Render Time:   0 hours 58 minutes 58 seconds (3538 seconds
     )
  Total Time:    1 hours  0 minutes  8 seconds (3608 seconds
     )
```

G.2.3.3. Pasada 3. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1365K tokens
  0:00:02 Creating bounding slabs
  0:00:02 Creating vista buffer
  0:00:02 Creating light buffers
  0:00:02 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:08 Sorting photons 0 of 63336
  0:01:08 Sorting photons 63335 of 63336
```

```
  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:59:07 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:          147840   Samples:          575936   Smpls/
    Pxl: 3.90
Rays:           1861058   Saved:             21855   Max
    Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Box                           79994791         9417290
    11.77
Cone/Cylinder                 78720800         6627468
    8.42
CSG Intersection             170489517        58806730
    34.49
CSG Merge                       863397           34750
    4.02
Fractal                        1782085          105385
    5.91
Height Field                   3586026          102531
    2.86
Height Field Box               3586026          689676
    19.23
Height Field Triangle          3262054          105733
    3.24
Height Field Block             5712599         1682572
    29.45
Height Field Cell             22619393         1790282
    7.91
Isosurface                    11950852          733084
    6.13
Isosurface Container          12409949        11951336
    96.30
Isosurface Cache                144545           43880
    30.36
Mesh                          15388221           64675
    0.42
Plane                         92234422         1295142
    1.40
Sphere                       281613813       175805479
    62.43
```

```
Superellipsoid                  639747          44455
   6.95
Torus                          2992390         423155
   14.14
Torus Bound                    2992390         489319
   16.35
True Type Font                  790747          80960
   10.24
Clipping Object                2587261        1540019
   59.52
Bounding Box                 521371907      149086904
   28.60
Vista Buffer                  22502747       12948149
   57.54
----------------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------------
Roots tested:                 489319   eliminated:
                  278583
Calls to Noise:           4832954757   Calls to DNoise:
   2620323810
----------------------------------------------------------------------------
Media Intervals:            39692199   Media Samples:
            358081477 (9.02)
Shadow Ray Tests:          128818521   Succeeded:
                 52507650
Reflected Rays:               224359   Total Internal:
                 2374
Refracted Rays:               143178
Transmitted Rays:             621485
----------------------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------------------
Smallest Alloc:                        9 bytes
Largest  Alloc:                  1440008 bytes
Peak memory used:                5509225 bytes
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes  2 seconds (2 seconds)
  Photon Time:    0 hours  1 minutes  8 seconds (68 seconds)
```

```
  Render Time:    0 hours 59 minutes  7 seconds (3547 seconds
      )
  Total Time:     1 hours  0 minutes 17 seconds (3617 seconds
      )
```

G.2.3.4. Pasada 4. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
    i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
      version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 852K tokens
  0:00:03 Creating bounding slabs
  0:00:03 Creating vista buffer
  0:00:03 Creating light buffers
  0:00:03 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:07 Sorting photons 0 of 63336
  0:01:07 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:59:01 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------------
Pixels:           147840   Samples:           575936   Smpls/
    Pxl: 3.90
Rays:            1861058   Saved:              21855   Max
    Level: 12/12
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
```

```
Box                            79994791        9417290
    11.77
Cone/Cylinder                  78720800        6627468
    8.42
CSG Intersection              170489517       58806730
    34.49
CSG Merge                        863397          34750
    4.02
Fractal                         1782085         105385
    5.91
Height Field                    3586026         102531
    2.86
Height Field Box                3586026         689676
    19.23
Height Field Triangle           3262054         105733
    3.24
Height Field Block              5712599        1682572
    29.45
Height Field Cell              22619393        1790282
    7.91
Isosurface                     11950852         733084
    6.13
Isosurface Container           12409949       11951336
    96.30
Isosurface Cache                 144545          43880
    30.36
Mesh                           15388221          64675
    0.42
Plane                          92234422        1295142
    1.40
Sphere                        281613813      175805479
    62.43
Superellipsoid                   639747          44455
    6.95
Torus                           2992390         423155
    14.14
Torus Bound                     2992390         489319
    16.35
True Type Font                   790747          80960
    10.24
Clipping Object                 2587261        1540019
    59.52
Bounding Box                  521371907      149086904
    28.60
```

```
Vista Buffer                    22502747         12948149
   57.54
----------------------------------------------------------------------
Isosurface roots:          11945843
Function VM calls:        172379889
----------------------------------------------------------------------
Roots tested:                489319    eliminated:
                 278583
Calls to Noise:          4832954757    Calls to DNoise:
   2620323810
----------------------------------------------------------------------
Media Intervals:           39692199    Media Samples:
            358081477 (9.02)
Shadow Ray Tests:         128818521    Succeeded:
                 52507650
Reflected Rays:              224359    Total Internal:
               2374
Refracted Rays:              143178
Transmitted Rays:            621485
----------------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------------
Smallest Alloc:                       9 bytes
Largest  Alloc:                 1440008 bytes
Peak memory used:               5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  3 seconds (3 seconds)
  Photon Time:   0 hours  1 minutes  8 seconds (68 seconds)
  Render Time:   0 hours 59 minutes  1 seconds (3541 seconds
     )
  Total Time:    1 hours  0 minutes 12 seconds (3612 seconds
     )
```

G.2.3.5. Pasada 5. .

```
Running standard POV-Ray benchmark version 1.02
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
```

```
POV–Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991−2003 Persistence of Vision Team
Copyright 2003−2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1045K tokens
  0:00:03 Creating bounding slabs
  0:00:03 Creating vista buffer
  0:00:03 Creating light buffers
  0:00:03 Creating light buffers 2299K tokens

  0:00:00 Building Photon Maps Photons 0 (sampling 0x0)
  0:01:07 Sorting photons 0 of 63336
  0:01:07 Sorting photons 63335 of 63336

  0:00:00 Rendering line 1 of 384, 0 supersamples
  0:59:04 Done Tracing
Render Statistics
Image Resolution 384 x 384
----------------------------------------------------------------------
Pixels:          147840    Samples:           575936    Smpls/
    Pxl: 3.90
Rays:           1861058    Saved:              21855    Max
    Level: 12/12
----------------------------------------------------------------------
Ray–>Shape Intersection            Tests         Succeeded
    Percentage
----------------------------------------------------------------------
Box                             79994791          9417290
    11.77
Cone/Cylinder                   78720800          6627468
    8.42
CSG Intersection               170489517         58806730
    34.49
CSG Merge                         863397            34750
    4.02
Fractal                          1782085           105385
    5.91
Height Field                     3586026           102531
    2.86
```

```
Height Field Box                 3586026          689676
   19.23
Height Field Triangle            3262054          105733
   3.24
Height Field Block               5712599         1682572
   29.45
Height Field Cell               22619393         1790282
   7.91
Isosurface                      11950852          733084
   6.13
Isosurface Container            12409949        11951336
   96.30
Isosurface Cache                  144545           43880
   30.36
Mesh                            15388221           64675
   0.42
Plane                           92234422         1295142
   1.40
Sphere                         281613813       175805479
   62.43
Superellipsoid                    639747           44455
   6.95
Torus                            2992390          423155
   14.14
Torus Bound                      2992390          489319
   16.35
True Type Font                    790747           80960
   10.24
Clipping Object                  2587261         1540019
   59.52
Bounding Box                   521371907       149086904
   28.60
Vista Buffer                    22502747        12948149
   57.54
----------------------------------------------------------------------
Isosurface roots:            11945843
Function VM calls:          172379889
----------------------------------------------------------------------
Roots tested:                  489319    eliminated:
                 278583
Calls to Noise:            4832954757    Calls to DNoise:
   2620323810
----------------------------------------------------------------------
Media Intervals:             39692199    Media Samples:
             358081477 (9.02)
```

```
Shadow Ray Tests:          128818521   Succeeded:
                52507650
Reflected Rays:               224359   Total Internal:
               2374
Refracted Rays:               143178
Transmitted Rays:             621485
----------------------------------------------------------------------
Number of photons shot:            74025
Surface photons stored:            63336
Priority queue insert:           1509036
Priority queue remove:            151992
Gather function called:           672370
----------------------------------------------------------------------
Smallest Alloc:                     9 bytes
Largest  Alloc:               1440008 bytes
Peak memory used:             5509225 bytes
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  3 seconds (3 seconds)
  Photon Time:   0 hours  1 minutes  7 seconds (67 seconds)
  Render Time:   0 hours 59 minutes  3 seconds (3543 seconds
     )
  Total Time:    1 hours  0 minutes 13 seconds (3613 seconds
     )
```

G.3. Pruebas de performance POV-Ray Office

G.3.1. Kernel Vainilla.

G.3.1.1. Pasada 1. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 350K tokens
  0:00:02 Parsing 1648K tokens
  0:00:03 Parsing 3191K tokens
```

```
  0:00:04 Parsing 4121K tokens
  0:00:04 Creating bounding slabs
  0:00:04 Creating vista buffer
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:19:42 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100    Samples:            83100    Smpls/
    Pxl: 1.00
Rays:            4775252    Saved:             107325    Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests        Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131                6
    4.58
Blob Component                     439              345
    78.59
Blob Bound                        1798              825
    45.88
Box                          266363203        121012543
    45.43
Cone/Cylinder                 23169118          1789386
    7.72
CSG Intersection             136989336         24887528
    18.17
CSG Merge                       138803            27599
    19.88
CSG Union                      6550385          1387411
    21.18
Height Field                   6226930          1016176
    16.32
Height Field Box               6226930          2731125
    43.86
Height Field Triangle          7038392          1043694
    14.83
Height Field Block             4556501          2979658
    65.39
```

```
Height Field Cell              24449821        3677475
   15.04
Mesh                           54934955       48890338
   89.00
Plane                         155259997       51322480
   33.06
Sphere                        100018753       40432864
   40.43
Sphere Sweep                    1609639         119680
   7.44
Superellipsoid                 29171499        8035390
   27.55
Torus                            933678         178534
   19.12
Torus Bound                      933678         184900
   19.80
Bounding Object                   63508          41980
   66.10
Bounding Box                 5266823890     1436450119
   27.27
Light Buffer                   57500763       20875597
   36.30
Vista Buffer                    6826541        4381728
   64.19
----------------------------------------------------------------------------
Function VM calls:            700031
----------------------------------------------------------------------------
Roots tested:               10203186   eliminated:
                  81623
Calls to Noise:              3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------------------
Media Intervals:               19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:           81952285   Succeeded:
                 19769016
Reflected Rays:                55714   Total Internal:
                 51
Refracted Rays:                 8652
Transmitted Rays:             491586
----------------------------------------------------------------------------
Radiosity samples calculated:           20681 (24.89 %)
Radiosity samples reused:               62393
----------------------------------------------------------------------------
Smallest Alloc:                       9 bytes
```

```
Largest  Alloc:           1179368 bytes
Peak memory used:         54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  4 seconds (4 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 19 minutes 42 seconds (1182 seconds
     )
  Total Time:    0 hours 19 minutes 46 seconds (1186 seconds
     )
```

G.3.1.2. Pasada 2. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .



  0:00:00 Parsing
  0:00:01 Parsing 27K tokens
  0:00:02 Parsing 1573K tokens
  0:00:03 Parsing 2868K tokens
  0:00:04 Parsing 4109K tokens
  0:00:04 Creating bounding slabs
  0:00:04 Creating vista buffer
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:19:39 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100    Samples:           83100    Smpls/
   Pxl: 1.00
Rays:            4775252    Saved:            107325    Max
   Level: 5/5
```

Ray->Shape Intersection	Tests	Succeeded	Percentage
Blob	131	6	4.58
Blob Component	439	345	78.59
Blob Bound	1798	825	45.88
Box	266363203	121012543	45.43
Cone/Cylinder	23169118	1789386	7.72
CSG Intersection	136989336	24887528	18.17
CSG Merge	138803	27599	19.88
CSG Union	6550385	1387411	21.18
Height Field	6226930	1016176	16.32
Height Field Box	6226930	2731125	43.86
Height Field Triangle	7038392	1043694	14.83
Height Field Block	4556501	2979658	65.39
Height Field Cell	24449821	3677475	15.04
Mesh	54934955	48890338	89.00
Plane	155259997	51322480	33.06
Sphere	100018753	40432864	40.43
Sphere Sweep	1609639	119680	7.44
Superellipsoid	29171499	8035390	27.55
Torus	933678	178534	19.12
Torus Bound	933678	184900	19.80

```
Bounding Object                 63508          41980
   66.10
Bounding Box               5266823890     1436450119
   27.27
Light Buffer                 57500763       20875597
   36.30
Vista Buffer                  6826541        4381728
   64.19
----------------------------------------------------------------------------
Function VM calls:           700031
----------------------------------------------------------------------------
Roots tested:              10203186    eliminated:
                 81623
Calls to Noise:             3145581    Calls to DNoise:
          16769112
----------------------------------------------------------------------------
Media Intervals:              19432    Media Samples:
              119624 (6.16)
Shadow Ray Tests:          81952285    Succeeded:
                19769016
Reflected Rays:               55714    Total Internal:
                 51
Refracted Rays:                8652
Transmitted Rays:            491586
----------------------------------------------------------------------------
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
----------------------------------------------------------------------------
Smallest Alloc:                   9 bytes
Largest  Alloc:             1179368 bytes
Peak memory used:          54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  4 seconds (4 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 19 minutes 39 seconds (1179 seconds
     )
  Total Time:    0 hours 19 minutes 43 seconds (1183 seconds
     )
```

G.3.1.3. Pasada 3. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
```

```
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .

  0:00:00 Parsing
  0:00:01 Parsing 387K tokens
  0:00:02 Parsing 1688K tokens
  0:00:03 Parsing 3191K tokens
  0:00:04 Parsing 4206K tokens
  0:00:04 Creating bounding slabs
  0:00:04 Creating vista buffer
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:19:39 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100   Samples:            83100   Smpls/
    Pxl: 1.00
Rays:            4775252   Saved:             107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131               6
    4.58
Blob Component                     439             345
    78.59
Blob Bound                        1798             825
    45.88
Box                          266363203       121012543
    45.43
Cone/Cylinder                 23169118         1789386
    7.72
CSG Intersection             136989336        24887528
    18.17
```

```
CSG Merge                     138803          27599
    19.88
CSG Union                    6550385        1387411
    21.18
Height Field                 6226930        1016176
    16.32
Height Field Box             6226930        2731125
    43.86
Height Field Triangle        7038392        1043694
    14.83
Height Field Block           4556501        2979658
    65.39
Height Field Cell           24449821        3677475
    15.04
Mesh                        54934955       48890338
    89.00
Plane                      155259997       51322480
    33.06
Sphere                     100018753       40432864
    40.43
Sphere Sweep                 1609639         119680
    7.44
Superellipsoid              29171499        8035390
    27.55
Torus                         933678         178534
    19.12
Torus Bound                   933678         184900
    19.80
Bounding Object                63508          41980
    66.10
Bounding Box              5266823890     1436450119
    27.27
Light Buffer                57500763       20875597
    36.30
Vista Buffer                 6826541        4381728
    64.19
----------------------------------------------------------------------
Function VM calls:          700031
----------------------------------------------------------------------
Roots tested:             10203186    eliminated:
                  81623
Calls to Noise:            3145581    Calls to DNoise:
          16769112
----------------------------------------------------------------------
```

```
Media Intervals:              19432   Media Samples:
               119624 (6.16)
Shadow Ray Tests:          81952285   Succeeded:
                 19769016
Reflected Rays:               55714   Total Internal:
                  51
Refracted Rays:                8652
Transmitted Rays:            491586
```

```
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
```

```
Smallest Alloc:                   9 bytes
Largest  Alloc:             1179368 bytes
Peak memory used:          54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  4 seconds (4 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 19 minutes 39 seconds (1179 seconds
     )
  Total Time:    0 hours 19 minutes 43 seconds (1183 seconds
     )
```

G.3.1.4. Pasada 4. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 250K tokens
  0:00:02 Parsing 1643K tokens
  0:00:03 Parsing 3171K tokens
  0:00:04 Parsing 4121K tokens
  0:00:04 Creating bounding slabs
  0:00:04 Creating vista buffer
```

```
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers
  0:00:04 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:19:39 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:           83100    Samples:           83100   Smpls/
    Pxl: 1.00
Rays:           4775252    Saved:            107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131               6
    4.58
Blob Component                     439             345
    78.59
Blob Bound                        1798             825
    45.88
Box                          266363203       121012543
    45.43
Cone/Cylinder                 23169118         1789386
    7.72
CSG Intersection             136989336        24887528
    18.17
CSG Merge                       138803           27599
    19.88
CSG Union                      6550385         1387411
    21.18
Height Field                   6226930         1016176
    16.32
Height Field Box               6226930         2731125
    43.86
Height Field Triangle          7038392         1043694
    14.83
Height Field Block             4556501         2979658
    65.39
Height Field Cell             24449821         3677475
    15.04
Mesh                          54934955        48890338
    89.00
```

```
Plane                          155259997          51322480
   33.06
Sphere                         100018753          40432864
   40.43
Sphere Sweep                     1609639            119680
   7.44
Superellipsoid                  29171499           8035390
   27.55
Torus                             933678            178534
   19.12
Torus Bound                       933678            184900
   19.80
Bounding Object                    63508             41980
   66.10
Bounding Box                  5266823890        1436450119
   27.27
Light Buffer                    57500763          20875597
   36.30
Vista Buffer                     6826541           4381728
   64.19
----------------------------------------------------------------------
Function VM calls:              700031
----------------------------------------------------------------------
Roots tested:                 10203186   eliminated:
                  81623
Calls to Noise:                3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------------
Media Intervals:                 19432   Media Samples:
               119624 (6.16)
Shadow Ray Tests:             81952285   Succeeded:
                 19769016
Reflected Rays:                  55714   Total Internal:
                 51
Refracted Rays:                   8652
Transmitted Rays:               491586
----------------------------------------------------------------------
Radiosity samples calculated:            20681 (24.89 %)
Radiosity samples reused:                62393
----------------------------------------------------------------------
Smallest Alloc:                      9 bytes
Largest  Alloc:                1179368 bytes
Peak memory used:             54630385 bytes
........
Total Scene Processing Times
```

```
  Parse Time:     0 hours  0 minutes  4 seconds (4 seconds)
  Photon Time:    0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:    0 hours 19 minutes 39 seconds (1179 seconds
     )
  Total Time:     0 hours 19 minutes 43 seconds (1183 seconds
     )
```

G.3.1.5. Pasada 5. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .

  0:00:00 Parsing
  0:00:01 Parsing 100K tokens
  0:00:02 Parsing 1593K tokens
  0:00:03 Parsing 2743K tokens
  0:00:04 Parsing 3909K tokens
  0:00:05 Parsing 4689K tokens
  0:00:05 Creating bounding slabs
  0:00:05 Creating vista buffer
  0:00:05 Creating light buffers
  0:00:05 Creating light buffers
  0:00:05 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:19:06 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100    Samples:            83100    Smpls/
   Pxl: 1.00
Rays:            4775252    Saved:             107325    Max
   Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection           Tests          Succeeded
   Percentage
----------------------------------------------------------------------------
```

Blob	131	6	4.58
Blob Component	439	345	78.59
Blob Bound	1798	825	45.88
Box	266363203	121012543	45.43
Cone/Cylinder	23169118	1789386	7.72
CSG Intersection	136989336	24887528	18.17
CSG Merge	138803	27599	19.88
CSG Union	6550385	1387411	21.18
Height Field	6226930	1016176	16.32
Height Field Box	6226930	2731125	43.86
Height Field Triangle	7038392	1043694	14.83
Height Field Block	4556501	2979658	65.39
Height Field Cell	24449821	3677475	15.04
Mesh	54934955	48890338	89.00
Plane	155259997	51322480	33.06
Sphere	100018753	40432864	40.43
Sphere Sweep	1609639	119680	7.44
Superellipsoid	29171499	8035390	27.55
Torus	933678	178534	19.12
Torus Bound	933678	184900	19.80
Bounding Object	63508	41980	66.10
Bounding Box	5266823890	1436450119	27.27

```
Light Buffer                   57500763        20875597
   36.30
Vista Buffer                    6826541         4381728
   64.19
----------------------------------------------------------------
Function VM calls:            700031
----------------------------------------------------------------
Roots tested:               10203186   eliminated:
                  81623
Calls to Noise:              3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------
Media Intervals:               19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:           81952285   Succeeded:
                19769016
Reflected Rays:                55714   Total Internal:
                 51
Refracted Rays:                 8652
Transmitted Rays:             491586
----------------------------------------------------------------
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
----------------------------------------------------------------
Smallest Alloc:                    9 bytes
Largest  Alloc:              1179368 bytes
Peak memory used:           54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  5 seconds (5 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 19 minutes  6 seconds (1146 seconds
     )
  Total Time:    0 hours 19 minutes 11 seconds (1151 seconds
     )
```

G.3.2. Kernel Modificado, modo 142.

G.3.2.1. Pasada 1. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
```

```
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .

  0:00:00 Parsing
  0:00:01 Parsing 187K tokens
  0:00:02 Parsing 1242K tokens
  0:00:03 Parsing 1863K tokens
  0:00:04 Parsing 2931K tokens
  0:00:05 Parsing 3886K tokens
  0:00:06 Parsing 4424K tokens
  0:00:06 Creating bounding slabs
  0:00:06 Creating vista buffer
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:25:32 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100   Samples:            83100   Smpls/
    Pxl: 1.00
Rays:            4775252   Saved:             107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131               6
    4.58
Blob Component                     439             345
    78.59
Blob Bound                        1798             825
    45.88
Box                          266363203       121012543
    45.43
Cone/Cylinder                 23169118         1789386
    7.72
CSG Intersection             136989336        24887528
    18.17
```

```
CSG Merge                          138803          27599
   19.88
CSG Union                         6550385        1387411
   21.18
Height Field                      6226930        1016176
   16.32
Height Field Box                  6226930        2731125
   43.86
Height Field Triangle             7038392        1043694
   14.83
Height Field Block                4556501        2979658
   65.39
Height Field Cell                24449821        3677475
   15.04
Mesh                             54934955       48890338
   89.00
Plane                           155259997       51322480
   33.06
Sphere                          100018753       40432864
   40.43
Sphere Sweep                      1609639         119680
   7.44
Superellipsoid                   29171499        8035390
   27.55
Torus                              933678         178534
   19.12
Torus Bound                        933678         184900
   19.80
Bounding Object                     63508          41980
   66.10
Bounding Box                   5266823890     1436450119
   27.27
Light Buffer                     57500763       20875597
   36.30
Vista Buffer                      6826541        4381728
   64.19
----------------------------------------------------------------------
Function VM calls:              700031
----------------------------------------------------------------------
Roots tested:                 10203186   eliminated:
                  81623
Calls to Noise:                3145541   Calls to DNoise:
             16769112
----------------------------------------------------------------------
```

```
Media Intervals:               19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:           81952285   Succeeded:
                19769016
Reflected Rays:                55714   Total Internal:
                 51
Refracted Rays:                 8652
Transmitted Rays:             491586
----------------------------------------------------------------
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
----------------------------------------------------------------
Smallest Alloc:                    9 bytes
Largest  Alloc:              1179368 bytes
Peak memory used:           54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  6 seconds (6 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 25 minutes 31 seconds (1531 seconds
     )
  Total Time:    0 hours 25 minutes 37 seconds (1537 seconds
     )
```

G.3.2.2. Pasada 2. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 497K tokens
  0:00:02 Parsing 645K tokens
  0:00:03 Parsing 657K tokens
  0:00:04 Parsing 670K tokens
  0:00:05 Parsing 682K tokens
  0:00:06 Parsing 700K tokens
```

```
  0:00:07 Parsing 862K tokens
  0:00:08 Parsing 1290K tokens
  0:00:09 Parsing 1573K tokens
  0:00:10 Parsing 1648K tokens
  0:00:11 Parsing 1848K tokens
  0:00:12 Parsing 2743K tokens
  0:00:13 Parsing 3243K tokens
  0:00:14 Parsing 3961K tokens
  0:00:15 Parsing 4046K tokens
  0:00:16 Parsing 4121K tokens
  0:00:17 Parsing 4126K tokens
  0:00:18 Parsing 4131K tokens
  0:00:19 Parsing 4136K tokens
  0:00:20 Parsing 4141K tokens
  0:00:21 Parsing 4151K tokens
  0:00:22 Parsing 4159K tokens
  0:00:23 Parsing 4214K tokens
  0:00:24 Parsing 4234K tokens
  0:00:25 Parsing 4276K tokens
  0:00:26 Parsing 4311K tokens
  0:00:27 Parsing 4351K tokens
  0:00:28 Parsing 4371K tokens
  0:00:29 Parsing 4406K tokens
  0:00:30 Parsing 4479K tokens
  0:00:31 Parsing 4499K tokens
  0:00:32 Parsing 4539K tokens
  0:00:33 Parsing 4601K tokens
  0:00:34 Parsing 4621K tokens
  0:00:35 Parsing 4669K tokens
  0:00:36 Parsing 4686K tokens
  0:00:37 Parsing 4689K tokens
  0:00:38 Parsing 4704K tokens
  0:00:38 Creating bounding slabs
  0:00:38 Creating vista buffer
  0:00:38 Creating light buffers
  0:00:38 Creating light buffers
  0:00:38 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:26:20 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:           83100   Samples:           83100   Smpls/
   Pxl: 1.00
```

```
Rays:            4775252    Saved:              107325    Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection               Tests          Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                                    131                  6
    4.58
Blob Component                          439                345
    78.59
Blob Bound                             1798                825
    45.88
Box                               266363203          121012543
    45.43
Cone/Cylinder                      23169118            1789386
    7.72
CSG Intersection                  136989336           24887528
    18.17
CSG Merge                            138803              27599
    19.88
CSG Union                           6550385            1387411
    21.18
Height Field                        6226930            1016176
    16.32
Height Field Box                    6226930            2731125
    43.86
Height Field Triangle               7038392            1043694
    14.83
Height Field Block                  4556501            2979658
    65.39
Height Field Cell                  24449821            3677475
    15.04
Mesh                               54934955           48890338
    89.00
Plane                             155259997           51322480
    33.06
Sphere                            100018753           40432864
    40.43
Sphere Sweep                        1609639             119680
    7.44
Superellipsoid                     29171499            8035390
    27.55
Torus                                933678             178534
    19.12
```

```
Torus Bound                        933678          184900
    19.80
Bounding Object                     63508           41980
    66.10
Bounding Box                   5266823890      1436450119
    27.27
Light Buffer                     57500763        20875597
    36.30
Vista Buffer                      6826541         4381728
    64.19
----------------------------------------------------------------------------
Function VM calls:              700031
----------------------------------------------------------------------------
Roots tested:                 10203186    eliminated:
                  81623
Calls to Noise:                3145581    Calls to DNoise:
          16769112
----------------------------------------------------------------------------
Media Intervals:                 19432    Media Samples:
              119624 (6.16)
Shadow Ray Tests:             81952285    Succeeded:
                19769016
Reflected Rays:                  55714    Total Internal:
                51
Refracted Rays:                   8652
Transmitted Rays:               491586
----------------------------------------------------------------------------
Radiosity samples calculated:              20681 (24.89 %)
Radiosity samples reused:                  62393
----------------------------------------------------------------------------
Smallest Alloc:                      9 bytes
Largest  Alloc:                1179368 bytes
Peak memory used:             54630385 bytes
........
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes 38 seconds (38 seconds)
  Photon Time:    0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:    0 hours 26 minutes 19 seconds (1579 seconds
    )
  Total Time:     0 hours 26 minutes 57 seconds (1617 seconds
    )
```

G.3.2.3. Pasada 3. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
    i686-pc-linux-gnu)
```

```
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 35K tokens
  0:00:02 Parsing 787K tokens
  0:00:03 Parsing 980K tokens
  0:00:04 Parsing 1548K tokens
  0:00:05 Parsing 1615K tokens
  0:00:06 Parsing 1693K tokens
  0:00:07 Parsing 2058K tokens
  0:00:08 Parsing 2846K tokens
  0:00:09 Parsing 3368K tokens
  0:00:10 Parsing 3961K tokens
  0:00:11 Parsing 4046K tokens
  0:00:12 Parsing 4121K tokens
  0:00:13 Parsing 4126K tokens
  0:00:14 Parsing 4129K tokens
  0:00:15 Parsing 4134K tokens
  0:00:16 Parsing 4139K tokens
  0:00:17 Parsing 4141K tokens
  0:00:18 Parsing 4146K tokens
  0:00:19 Parsing 4149K tokens
  0:00:20 Parsing 4154K tokens
  0:00:21 Parsing 4206K tokens
  0:00:22 Parsing 4226K tokens
  0:00:23 Parsing 4279K tokens
  0:00:24 Parsing 4311K tokens
  0:00:25 Parsing 4349K tokens
  0:00:26 Parsing 4371K tokens
  0:00:27 Parsing 4406K tokens
  0:00:28 Parsing 4476K tokens
  0:00:29 Parsing 4494K tokens
  0:00:30 Parsing 4534K tokens
  0:00:31 Parsing 4594K tokens
  0:00:32 Parsing 4614K tokens
  0:00:33 Parsing 4661K tokens
```

```
  0:00:34 Parsing 4679K tokens
  0:00:35 Parsing 4689K tokens
  0:00:36 Parsing 4696K tokens
  0:00:36 Creating bounding slabs
  0:00:36 Creating vista buffer
  0:00:36 Creating light buffers
  0:00:36 Creating light buffers
  0:00:36 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:26:24 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100   Samples:            83100   Smpls/
    Pxl: 1.00
Rays:            4775252   Saved:             107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131               6
    4.58
Blob Component                     439             345
    78.59
Blob Bound                        1798             825
    45.88
Box                          266363203       121012543
    45.43
Cone/Cylinder                 23169118         1789386
    7.72
CSG Intersection             136989336        24887528
    18.17
CSG Merge                       138803           27599
    19.88
CSG Union                      6550385         1387411
    21.18
Height Field                   6226930         1016176
    16.32
Height Field Box               6226930         2731125
    43.86
Height Field Triangle          7038392         1043694
    14.83
```

```
Height Field Block              4556501         2979658
   65.39
Height Field Cell              24449821         3677475
   15.04
Mesh                           54934955        48890338
   89.00
Plane                         155259997        51322480
   33.06
Sphere                        100018753        40432864
   40.43
Sphere Sweep                    1609639          119680
   7.44
Superellipsoid                 29171499         8035390
   27.55
Torus                            933678          178534
   19.12
Torus Bound                      933678          184900
   19.80
Bounding Object                   63508           41980
   66.10
Bounding Box                 5266823890      1436450119
   27.27
Light Buffer                   57500763        20875597
   36.30
Vista Buffer                    6826541         4381728
   64.19
----------------------------------------------------------------------
Function VM calls:            700031
----------------------------------------------------------------------
Roots tested:               10203186   eliminated:
                81623
Calls to Noise:              3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------------
Media Intervals:               19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:           81952285   Succeeded:
                19769016
Reflected Rays:                55714   Total Internal:
                51
Refracted Rays:                 8652
Transmitted Rays:             491586
----------------------------------------------------------------------
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
```

```
Smallest Alloc:                 9 bytes
Largest  Alloc:           1179368 bytes
Peak memory used:        54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes 36 seconds (36 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 26 minutes 22 seconds (1582 seconds
      )
  Total Time:    0 hours 26 minutes 58 seconds (1618 seconds
      )
```

G.3.2.4. Pasada 4. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
    i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
      version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 410K tokens
  0:00:02 Parsing 645K tokens
  0:00:03 Parsing 655K tokens
  0:00:04 Parsing 667K tokens
  0:00:05 Parsing 680K tokens
  0:00:06 Parsing 692K tokens
  0:00:07 Parsing 702K tokens
  0:00:08 Parsing 917K tokens
  0:00:09 Parsing 1405K tokens
  0:00:10 Parsing 1573K tokens
  0:00:11 Parsing 1635K tokens
  0:00:12 Parsing 1730K tokens
  0:00:13 Parsing 2298K tokens
  0:00:14 Parsing 3158K tokens
  0:00:15 Parsing 3401K tokens
  0:00:16 Parsing 3979K tokens
  0:00:17 Parsing 4046K tokens
```

```
  0:00:18 Parsing 4121K tokens
  0:00:19 Parsing 4126K tokens
  0:00:20 Parsing 4129K tokens
  0:00:21 Parsing 4134K tokens
  0:00:22 Parsing 4136K tokens
  0:00:23 Parsing 4141K tokens
  0:00:24 Parsing 4144K tokens
  0:00:25 Parsing 4149K tokens
  0:00:26 Parsing 4151K tokens
  0:00:27 Parsing 4166K tokens
  0:00:28 Parsing 4216K tokens
  0:00:29 Parsing 4259K tokens
  0:00:30 Parsing 4279K tokens
  0:00:31 Parsing 4311K tokens
  0:00:32 Parsing 4349K tokens
  0:00:33 Parsing 4369K tokens
  0:00:34 Parsing 4404K tokens
  0:00:35 Parsing 4479K tokens
  0:00:36 Parsing 4496K tokens
  0:00:37 Parsing 4536K tokens
  0:00:38 Parsing 4596K tokens
  0:00:39 Parsing 4616K tokens
  0:00:40 Parsing 4664K tokens
  0:00:41 Parsing 4681K tokens
  0:00:42 Parsing 4689K tokens
  0:00:43 Parsing 4699K tokens
  0:00:43 Creating bounding slabs
  0:00:43 Creating vista buffer
  0:00:43 Creating light buffers
  0:00:43 Creating light buffers
  0:00:43 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:26:42 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100   Samples:           83100   Smpls/
    Pxl: 1.00
Rays:            4775252   Saved:            107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests         Succeeded
    Percentage
----------------------------------------------------------------------------
```

```
Blob                                 131               6
   4.58
Blob Component                       439             345
   78.59
Blob Bound                          1798             825
   45.88
Box                            266363203       121012543
   45.43
Cone/Cylinder                   23169118         1789386
   7.72
CSG Intersection               136989336        24887528
   18.17
CSG Merge                         138803           27599
   19.88
CSG Union                        6550385         1387411
   21.18
Height Field                     6226930         1016176
   16.32
Height Field Box                 6226930         2731125
   43.86
Height Field Triangle            7038392         1043694
   14.83
Height Field Block               4556501         2979658
   65.39
Height Field Cell               24449821         3677475
   15.04
Mesh                            54934955        48890338
   89.00
Plane                          155259997        51322480
   33.06
Sphere                         100018753        40432864
   40.43
Sphere Sweep                     1609639          119680
   7.44
Superellipsoid                  29171499         8035390
   27.55
Torus                             933678          178534
   19.12
Torus Bound                       933678          184900
   19.80
Bounding Object                    63508           41980
   66.10
Bounding Box                  5266823890      1436450119
   27.27
```

```
Light Buffer                  57500763        20875597
   36.30
Vista Buffer                   6826541         4381728
   64.19
----------------------------------------------------------------------
Function VM calls:           700031
----------------------------------------------------------------------
Roots tested:              10203186   eliminated:
                 81623
Calls to Noise:             3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------------
Media Intervals:              19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:          81952285   Succeeded:
                19769016
Reflected Rays:               55714   Total Internal:
                 51
Refracted Rays:                8652
Transmitted Rays:            491586
----------------------------------------------------------------------
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
----------------------------------------------------------------------
Smallest Alloc:                   9 bytes
Largest  Alloc:             1179368 bytes
Peak memory used:          54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes 43 seconds (43 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 26 minutes 41 seconds (1601 seconds
     )
  Total Time:    0 hours 27 minutes 24 seconds (1644 seconds
     )
```

G.3.2.5. Pasada 5. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
      version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
```

```
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


 0:00:00 Parsing
 0:00:01 Parsing 135K tokens
 0:00:02 Parsing 642K tokens
 0:00:03 Parsing 655K tokens
 0:00:04 Parsing 665K tokens
 0:00:05 Parsing 677K tokens
 0:00:06 Parsing 692K tokens
 0:00:07 Parsing 702K tokens
 0:00:08 Parsing 932K tokens
 0:00:09 Parsing 1485K tokens
 0:00:10 Parsing 1578K tokens
 0:00:11 Parsing 1648K tokens
 0:00:12 Parsing 1820K tokens
 0:00:13 Parsing 2706K tokens
 0:00:14 Parsing 3191K tokens
 0:00:15 Parsing 3774K tokens
 0:00:16 Parsing 3986K tokens
 0:00:17 Parsing 4109K tokens
 0:00:18 Parsing 4124K tokens
 0:00:19 Parsing 4129K tokens
 0:00:20 Parsing 4134K tokens
 0:00:21 Parsing 4141K tokens
 0:00:22 Parsing 4146K tokens
 0:00:23 Parsing 4151K tokens
 0:00:24 Parsing 4164K tokens
 0:00:25 Parsing 4216K tokens
 0:00:26 Parsing 4261K tokens
 0:00:27 Parsing 4301K tokens
 0:00:28 Parsing 4324K tokens
 0:00:29 Parsing 4366K tokens
 0:00:30 Parsing 4404K tokens
 0:00:31 Parsing 4476K tokens
 0:00:32 Parsing 4499K tokens
 0:00:33 Parsing 4539K tokens
 0:00:34 Parsing 4601K tokens
 0:00:35 Parsing 4624K tokens
 0:00:36 Parsing 4674K tokens
 0:00:37 Parsing 4689K tokens
 0:00:38 Parsing 4691K tokens
 0:00:38 Creating bounding slabs
```

```
  0:00:38 Creating vista buffer
  0:00:38 Creating light buffers
  0:00:38 Creating light buffers
  0:00:39 Creating light buffers
  0:00:39 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:26:20 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100   Samples:           83100   Smpls/
    Pxl: 1.00
Rays:            4775252   Saved:            107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131               6
    4.58
Blob Component                     439             345
    78.59
Blob Bound                        1798             825
    45.88
Box                          266363203       121012543
    45.43
Cone/Cylinder                 23169118         1789386
    7.72
CSG Intersection             136989336        24887528
    18.17
CSG Merge                       138803           27599
    19.88
CSG Union                      6550385         1387411
    21.18
Height Field                   6226930         1016176
    16.32
Height Field Box               6226930         2731125
    43.86
Height Field Triangle          7038392         1043694
    14.83
Height Field Block             4556501         2979658
    65.39
Height Field Cell             24449821         3677475
    15.04
```

```
Mesh                          54934955         48890338
   89.00
Plane                        155259997         51322480
   33.06
Sphere                       100018753         40432864
   40.43
Sphere Sweep                   1609639           119680
   7.44
Superellipsoid                29171499          8035390
   27.55
Torus                           933678           178534
   19.12
Torus Bound                     933678           184900
   19.80
Bounding Object                  63508            41980
   66.10
Bounding Box                5266823890       1436450119
   27.27
Light Buffer                  57500763         20875597
   36.30
Vista Buffer                   6826541          4381728
   64.19
----------------------------------------------------------
Function VM calls:              700031
----------------------------------------------------------
Roots tested:                 10203186    eliminated:
                  81623
Calls to Noise:                3145581    Calls to DNoise:
          16769112
----------------------------------------------------------
Media Intervals:                 19432    Media Samples:
               119624 (6.16)
Shadow Ray Tests:             81952285    Succeeded:
                 19769016
Reflected Rays:                  55714    Total Internal:
                 51
Refracted Rays:                   8652
Transmitted Rays:               491586
----------------------------------------------------------
Radiosity samples calculated:             20681 (24.89 %)
Radiosity samples reused:                 62393
----------------------------------------------------------
Smallest Alloc:                      9 bytes
Largest  Alloc:                1179368 bytes
Peak memory used:             54630385 bytes
```

```
........
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes 39 seconds (39 seconds)
  Photon Time:    0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:    0 hours 26 minutes 19 seconds (1579 seconds
     )
  Total Time:     0 hours 26 minutes 58 seconds (1618 seconds
     )
```

G.3.3. Kernel Modificado, modo 146.

G.3.3.1. Pasada 1. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 342K tokens
  0:00:02 Parsing 1393K tokens
  0:00:03 Parsing 2015K tokens
  0:00:04 Parsing 3126K tokens
  0:00:05 Parsing 3979K tokens
  0:00:06 Parsing 4629K tokens
  0:00:06 Creating bounding slabs
  0:00:06 Creating vista buffer
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:25:44 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100   Samples:            83100   Smpls/
   Pxl: 1.00
```

```
Rays:              4775252    Saved:              107325    Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection           Tests          Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                                131                  6
    4.58
Blob Component                      439                345
    78.59
Blob Bound                         1798                825
    45.88
Box                           266363203          121012543
    45.43
Cone/Cylinder                  23169118            1789386
    7.72
CSG Intersection              136989336           24887528
    18.17
CSG Merge                        138803              27599
    19.88
CSG Union                       6550385            1387411
    21.18
Height Field                    6226930            1016176
    16.32
Height Field Box                6226930            2731125
    43.86
Height Field Triangle           7038392            1043694
    14.83
Height Field Block              4556501            2979658
    65.39
Height Field Cell              24449821            3677475
    15.04
Mesh                           54934955           48890338
    89.00
Plane                         155259997           51322480
    33.06
Sphere                        100018753           40432864
    40.43
Sphere Sweep                    1609639             119680
    7.44
Superellipsoid                 29171499            8035390
    27.55
Torus                            933678             178534
    19.12
```

```
Torus Bound                           933678          184900
    19.80
Bounding Object                        63508           41980
    66.10
Bounding Box                      5266823890      1436450119
    27.27
Light Buffer                        57500763        20875597
    36.30
Vista Buffer                         6826541         4381728
    64.19
----------------------------------------------------------------------
Function VM calls:              700031
----------------------------------------------------------------------
Roots tested:                 10203186   eliminated:
                 81623
Calls to Noise:                3145581   Calls to DNoise:
           16769112
----------------------------------------------------------------------
Media Intervals:                 19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:             81952285   Succeeded:
                19769016
Reflected Rays:                  55714   Total Internal:
                51
Refracted Rays:                   8652
Transmitted Rays:               491586
----------------------------------------------------------------------
Radiosity samples calculated:            20681 (24.89 %)
Radiosity samples reused:                62393
----------------------------------------------------------------------
Smallest Alloc:                      9 bytes
Largest  Alloc:                1179368 bytes
Peak memory used:             54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  6 seconds (6 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 25 minutes 44 seconds (1544 seconds
    )
  Total Time:    0 hours 25 minutes 50 seconds (1550 seconds
    )
```

G.3.3.2. Pasada 2. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
    i686-pc-linux-gnu)
```

```
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 830K tokens
  0:00:02 Parsing 1665K tokens
  0:00:03 Parsing 2743K tokens
  0:00:04 Parsing 3611K tokens
  0:00:05 Parsing 4159K tokens
  0:00:05 Creating bounding slabs
  0:00:05 Creating vista buffer
  0:00:05 Creating light buffers
  0:00:05 Creating light buffers
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:26:08 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------
Pixels:          83100   Samples:          83100   Smpls/
    Pxl: 1.00
Rays:          4775252   Saved:           107325   Max
    Level: 5/5
----------------------------------------------------------------
Ray->Shape Intersection          Tests        Succeeded
    Percentage
----------------------------------------------------------------
Blob                               131                6
    4.58
Blob Component                     439              345
    78.59
Blob Bound                        1798              825
    45.88
Box                          266363203        121012543
    45.43
```

```
Cone/Cylinder                    23169118          1789386
    7.72
CSG Intersection                136989336         24887528
    18.17
CSG Merge                          138803            27599
    19.88
CSG Union                         6550385          1387411
    21.18
Height Field                      6226930          1016176
    16.32
Height Field Box                  6226930          2731125
    43.86
Height Field Triangle             7038392          1043694
    14.83
Height Field Block                4556501          2979658
    65.39
Height Field Cell                24449821          3677475
    15.04
Mesh                             54934955         48890338
    89.00
Plane                           155259997         51322480
    33.06
Sphere                          100018753         40432864
    40.43
Sphere Sweep                      1609639           119680
    7.44
Superellipsoid                   29171499          8035390
    27.55
Torus                              933678           178534
    19.12
Torus Bound                        933678           184900
    19.80
Bounding Object                     63508            41980
    66.10
Bounding Box                   5266823890       1436450119
    27.27
Light Buffer                     57500763         20875597
    36.30
Vista Buffer                      6826541          4381728
    64.19
----------------------------------------------------------------
Function VM calls:              700031
----------------------------------------------------------------
Roots tested:                 10203186   eliminated:
                  81623
```

```
Calls to Noise:            3145581   Calls to DNoise:
         16769112
----------------------------------------------------------
Media Intervals:             19432   Media Samples:
             119624 (6.16)
Shadow Ray Tests:         81952285   Succeeded:
               19769016
Reflected Rays:              55714   Total Internal:
                51
Refracted Rays:               8652
Transmitted Rays:           491586
----------------------------------------------------------
Radiosity samples calculated:          20681 (24.89 %)
Radiosity samples reused:              62393
----------------------------------------------------------
Smallest Alloc:                  9 bytes
Largest  Alloc:            1179368 bytes
Peak memory used:         54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  6 seconds (6 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 26 minutes  7 seconds (1567 seconds
     )
  Total Time:    0 hours 26 minutes 13 seconds (1573 seconds
     )
```

G.3.3.3. Pasada 3. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 675K tokens
  0:00:02 Parsing 1608K tokens
  0:00:03 Parsing 2588K tokens
```

```
  0:00:04 Parsing 3421K tokens
  0:00:05 Parsing 4111K tokens
  0:00:05 Creating bounding slabs
  0:00:05 Creating vista buffer
  0:00:05 Creating light buffers
  0:00:05 Creating light buffers
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:25:45 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:            83100    Samples:           83100    Smpls/
   Pxl: 1.00
Rays:            4775252    Saved:            107325    Max
   Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection            Tests        Succeeded
   Percentage
----------------------------------------------------------------------------
Blob                                 131                6
   4.58
Blob Component                       439              345
   78.59
Blob Bound                          1798              825
   45.88
Box                            266363203        121012543
   45.43
Cone/Cylinder                   23169118          1789386
   7.72
CSG Intersection               136989336         24887528
   18.17
CSG Merge                         138803            27599
   19.88
CSG Union                        6550385          1387411
   21.18
Height Field                     6226930          1016176
   16.32
Height Field Box                 6226930          2731125
   43.86
Height Field Triangle            7038392          1043694
   14.83
```

```
Height Field Block              4556501         2979658
   65.39
Height Field Cell              24449821         3677475
   15.04
Mesh                           54934955        48890338
   89.00
Plane                         155259997        51322480
   33.06
Sphere                        100018753        40432864
   40.43
Sphere Sweep                    1609639          119680
   7.44
Superellipsoid                 29171499         8035390
   27.55
Torus                            933678          178534
   19.12
Torus Bound                      933678          184900
   19.80
Bounding Object                   63508           41980
   66.10
Bounding Box                 5266823890      1436450119
   27.27
Light Buffer                   57500763        20875597
   36.30
Vista Buffer                    6826541         4381728
   64.19
----------------------------------------------------------
Function VM calls:             700031
----------------------------------------------------------
Roots tested:                10203186   eliminated:
                 81623
Calls to Noise:               3145581   Calls to DNoise:
          16769112
----------------------------------------------------------
Media Intervals:                19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:            81952285   Succeeded:
                19769016
Reflected Rays:                 55714   Total Internal:
                 51
Refracted Rays:                  8652
Transmitted Rays:              491586
----------------------------------------------------------
Radiosity samples calculated:           20681 (24.89 %)
Radiosity samples reused:               62393
```

```
Smallest Alloc:                 9 bytes
Largest  Alloc:           1179368 bytes
Peak memory used:        54630385 bytes
........
Total Scene Processing Times
  Parse Time:     0 hours  0 minutes  6 seconds (6 seconds)
  Photon Time:    0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:    0 hours 25 minutes 45 seconds (1545 seconds
     )
  Total Time:     0 hours 25 minutes 51 seconds (1551 seconds
     )
```

G.3.3.4. Pasada 4. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
   i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
     version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
   . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
   .


  0:00:00 Parsing
  0:00:01 Parsing 650K tokens
  0:00:02 Parsing 1603K tokens
  0:00:03 Parsing 2561K tokens
  0:00:04 Parsing 3396K tokens
  0:00:05 Parsing 4109K tokens
  0:00:05 Creating bounding slabs
  0:00:05 Creating vista buffer
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:25:41 Done Tracing
Render Statistics
Image Resolution 320 x 240
```

```
Pixels:          83100   Samples:          83100   Smpls/
   Pxl: 1.00
Rays:          4775252   Saved:           107325   Max
   Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
   Percentage
----------------------------------------------------------------------------
Blob                               131               6
   4.58
Blob Component                     439             345
   78.59
Blob Bound                        1798             825
   45.88
Box                          266363203       121012543
   45.43
Cone/Cylinder                 23169118         1789386
   7.72
CSG Intersection             136989336        24887528
   18.17
CSG Merge                       138803           27599
   19.88
CSG Union                      6550385         1387411
   21.18
Height Field                   6226930         1016176
   16.32
Height Field Box               6226930         2731125
   43.86
Height Field Triangle          7038392         1043694
   14.83
Height Field Block             4556501         2979658
   65.39
Height Field Cell             24449821         3677475
   15.04
Mesh                          54934955        48890338
   89.00
Plane                        155259997        51322480
   33.06
Sphere                       100018753        40432864
   40.43
Sphere Sweep                   1609639          119680
   7.44
Superellipsoid                29171499         8035390
   27.55
```

```
Torus                                933678          178534
   19.12
Torus Bound                          933678          184900
   19.80
Bounding Object                       63508           41980
   66.10
Bounding Box                     5266823890      1436450119
   27.27
Light Buffer                       57500763        20875597
   36.30
Vista Buffer                        6826541         4381728
   64.19
----------------------------------------------------------------------
Function VM calls:             700031
----------------------------------------------------------------------
Roots tested:                10203186   eliminated:
                   81623
Calls to Noise:               3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------------
Media Intervals:                19432   Media Samples:
              119624 (6.16)
Shadow Ray Tests:            81952285   Succeeded:
                 19769016
Reflected Rays:                 55714   Total Internal:
                   51
Refracted Rays:                  8652
Transmitted Rays:              491586
----------------------------------------------------------------------
Radiosity samples calculated:              20681 (24.89 %)
Radiosity samples reused:                  62393
----------------------------------------------------------------------
Smallest Alloc:                     9 bytes
Largest  Alloc:               1179368 bytes
Peak memory used:            54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  6 seconds (6 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 25 minutes 41 seconds (1541 seconds
     )
  Total Time:    0 hours 25 minutes 47 seconds (1547 seconds
     )
```

G.3.3.5. Pasada 5. .

```
Persistence of Vision(tm) Ray Tracer Version 3.6.1 (g++ @
    i686-pc-linux-gnu)
This is an unofficial version compiled by:
 SUSE LINUX Products GmbH, Nuernberg, Germany
 The POV-Ray Team(tm) is not responsible for supporting this
      version.
POV-Ray is based on DKBTrace 2.12 by David K. Buck & Aaron A
    . Collins
Copyright 1991-2003 Persistence of Vision Team
Copyright 2003-2004 Persistence of Vision Raytracer Pty. Ltd
    .


  0:00:00 Parsing
  0:00:01 Parsing 37K tokens
  0:00:02 Parsing 1175K tokens
  0:00:03 Parsing 1850K tokens
  0:00:04 Parsing 2973K tokens
  0:00:05 Parsing 3961K tokens
  0:00:06 Parsing 4591K tokens
  0:00:06 Creating bounding slabs
  0:00:06 Creating vista buffer
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers
  0:00:06 Creating light buffers 4705K tokens

  0:00:00 Displaying
  0:25:39 Done Tracing
Render Statistics
Image Resolution 320 x 240
----------------------------------------------------------------------------
Pixels:           83100   Samples:          83100   Smpls/
    Pxl: 1.00
Rays:           4775252   Saved:           107325   Max
    Level: 5/5
----------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded
    Percentage
----------------------------------------------------------------------------
Blob                               131               6
    4.58
Blob Component                     439             345
    78.59
Blob Bound                        1798             825
    45.88
```

```
Box                          266363203      121012543
   45.43
Cone/Cylinder                 23169118        1789386
   7.72
CSG Intersection             136989336       24887528
   18.17
CSG Merge                       138803          27599
   19.88
CSG Union                      6550385        1387411
   21.18
Height Field                   6226930        1016176
   16.32
Height Field Box               6226930        2731125
   43.86
Height Field Triangle          7038392        1043694
   14.83
Height Field Block             4556501        2979658
   65.39
Height Field Cell             24449821        3677475
   15.04
Mesh                          54934955       48890338
   89.00
Plane                        155259997       51322480
   33.06
Sphere                       100018753       40432864
   40.43
Sphere Sweep                   1609639         119680
   7.44
Superellipsoid                29171499        8035390
   27.55
Torus                           933678         178534
   19.12
Torus Bound                     933678         184900
   19.80
Bounding Object                  63508          41980
   66.10
Bounding Box                5266823890     1436450119
   27.27
Light Buffer                  57500763       20875597
   36.30
Vista Buffer                   6826541        4381728
   64.19
----------------------------------------------------------
Function VM calls:           700031
----------------------------------------------------------
```

```
Roots tested:              10203186   eliminated:
                81623
Calls to Noise:             3145581   Calls to DNoise:
          16769112
----------------------------------------------------------------------------
Media Intervals:              19432   Media Samples:
               119624 (6.16)
Shadow Ray Tests:          81952285   Succeeded:
                19769016
Reflected Rays:               55714   Total Internal:
                51
Refracted Rays:                8652
Transmitted Rays:            491586
----------------------------------------------------------------------------
Radiosity samples calculated:           20681 (24.89 %)
Radiosity samples reused:               62393
----------------------------------------------------------------------------
Smallest Alloc:                   9 bytes
Largest  Alloc:             1179368 bytes
Peak memory used:          54630385 bytes
........
Total Scene Processing Times
  Parse Time:    0 hours  0 minutes  6 seconds (6 seconds)
  Photon Time:   0 hours  0 minutes  0 seconds (0 seconds)
  Render Time:   0 hours 25 minutes 39 seconds (1539 seconds
     )
  Total Time:    0 hours 25 minutes 45 seconds (1545 seconds
     )
```