Segundo Parcial Junio 2013

Lea detenidamente las siguientes instrucciones. No cumplir los requerimientos puede implicar la pérdida de los puntos del parcial.

Formato

- **Indique su nombre completo y número de cédula en cada hoja (No se corregirán las hojas sin nombre, sin excepciones).**
- **Numere todas las hojas e indique la cantidad total de hojas que entrega en la primera.**
- **Escriba las hojas de un solo lado y empiece cada problema en una hoja nueva.**
- **Si se entregan varias versiones de un problema solo se corregirá el primero de ellos según el orden de hojas.**

Dudas

- **Sólo se contestarán dudas de letra.**
- **No se aceptarán dudas en los últimos 20 minutos del parcial.**

Material

- **El examen es SIN material (no puede utilizarse ningún apunte, dispositivo móvil, libro ni calculadora). Sólo puede tenerse las hojas del examen, lápiz, goma y lapicera en su banco. Todas sus demás pertenencias debe colocarlas en el piso debajo de su asiento.**

Finalización

- **El parcial dura 3 horas 15 minutos.**
- **Al momento de finalizar el parcial no se podrá escribir absolutamente nada en las hojas, debiéndose parar e ir a la fila de entrega. Identificar cada una de las hojas con nombre, cédula y numeración forma parte de la duración del parcial.**

Problema 1 (10 puntos)

1. Explique el algoritmo de reemplazo de páginas de segunda chance mejorado (*NRU, Not Recently Used*) e indique el soporte de hardware necesario para una implementación eficiente.
2. Indique ventajas y desventajas de utilizar las estructuras de tabla de páginas jerárquica y diccionario, para una arquitectura de 64 bits que utilice paginación y tenga un tamaño de páginas de 4KB.
3. Describa dos razones para utilizar *"buffering".*
4. En el contexto de Virtualización, describa qué son las instrucciones *sensibles* y que condición debe cumplir el hardware para soportar hipervisores de tipo 1.
5. Explique el mecanismo en un sistema Unix para permitir que un usuario ejecute una aplicación con privilegios de administrador.

Problema 2 (20 puntos) (3,3,14)

En un sistema operativo experimental se determinó que dada las características de su utilización la mejor estrategia para implementar su sistema de archivos es una que combine la asignación de bloques en forma de lista (Linked Allocation) y la asignación indexada (Indexed Allocation).

Este sistema de archivos cuenta con la siguiente estructura de datos:

```
type sector = array [0..4095] of byte; // 4096 bytes


type dir_entry = Record
     name : Array [0..11] of char;    // 12 byte
     type : (file,dir);               // 1 bit
     used : Boolean;                  // 1 bit
     comienzo_archivo: Integer;       // 4 byte
     tamano_archivo: Integer;         // 4 byte
     id_dueno: Permiso;               // 3 byte
     id_grupo: Permiso;               // 3 byte
     i_node_num : Integer;            // 4 byte
     permisos : array [0..13] of bit; // 14 bits
     End;


type i_node = Record
     i_node_num : Integer;           // 16 bits
     used : Boolean;                 // 1 bit
     data: array [1..15] of Integer; // 60 byte
     tope: Integer;                  // 4 byte
     reserved : array[0..6] of bit;  // 7 bits
     End;


type inode_table = Array [0..max_i_node_on_disk-1] of i_node;
type fat = array [0..max_sectors_in_disk-1] of integer;
type disk = Array [0..max_sectors_in_disk-1] of sector;

Var
     TD : inode_table;
     FAT :fat;
     D : disk;
```

Las características de este sistema de archivos son la siguientes:

- los archivos son almacenados siguiendo una estrategia de asignación en forma de lista tipo FAT (File Allocation Table), donde el atributo `comienzo_archivo` del `dir_entry` indica el comienzo del archivo.

- La información de los directorios es almacenada utilizando una estrategia de asignación indexada directa, donde el atributo `i_node_num` indica el inodo que almacena la información del directorio.
- Las variables TD, FAT y D son globales
- En la FAT el valor 0 representa "fin de archivo", el -1 "sector libre" y el -2 indica que ese sector está siendo utilizado por un directorio.
- Los archivos de tamaño 0 no reservan ningún bloque de disco.
- El inodo 0 almacena la información del directorio raíz.

Por otra parte se dispone de los siguientes procedimientos:

- `Procedure readBlock(d: disk; block_num: 0..MAX_BLOCKS_ON_DISK-1; Var buff: block, Var ok: Boolean);` Lee de disco el bloque pasado como parámetro (carga el parámetro de salida buff). En el parámetro `ok` se retorna el éxito de la ejecución de la operación.
- `Procedure writeBlock(d : disk; block_num : 0..MAX_BLOCKS_ON_DISK-1; buff : block, ok : Boolean);` Escribe en el bloque pasado como parámetro la información que se encuentra en el parámetro buff. En el parámetro ok se retorna el éxito de la ejecución de la operación.
- `Procedure getInode(cam: array of char; Var nro_inodo: Integer; Var ok: Boolean);` Retorna en `nro_inodo` el número de inodo correspondiente al camino absoluto (`cam`). En el parámetro `ok` se retorna el éxito de la ejecución de la operación.
- `Procedure dirname(cam : array of char; Var dir: array of char);` Retorna en el parámetro `dir` el directorio que contiene al archivo referenciado en el parámetro `cam`. Ejemplo, `dirname('/home/sistoper/arch.txt') = '/home/sistoper'`.
- `Procedure basename(cam : array of char; Var base: array of char);` Retorna en el parámetro base el directorio que contiene al archivo referenciado en el parámetro `cam`. Ejemplo, `basename('/home/sistoper/arch.txt') = 'arch.txt'`.

Se pide:

1. ¿Cuántos archivos y/o directorios puede contener un directorio como máximo?. Justifique su respuesta.
2. Determine cuál es la cantidad máxima de elementos (archivos y/o directorios) que se pueden almacenar en este sistema de archivos. Se sabe que `max_sectors_in_disk` es mucho mayor que `max_i_node_on_disk`. Justifique su respuesta.
3. Implementar una función (createFile) que cree un archivo de n bytes con 0s en el sistema. El cabezal de la función es el siguiente:

   ```
   Procedure createFile(cam : array of char; id_dueno: Permiso;
   id_grupo: Permiso; size: Integer; p: array [0..13] of bit; Var ok :
   Boolean);
   ```

   El parámetro `cam` representa el camino absoluto al archivo (ej. '/home/sistoper/archivo.txt'). El parámetro `id_dueno` y `id_grupo` son los identificadores del dueño del archivo y el identificador del grupo del archivo respectivamente, el parámetro `size` indica la cantidad de bytes del archivo y el parámetro `p` indica los permisos. En el parámetro `ok` se retorna el éxito en la ejecución de la operación. Se considera que hay espacio suficiente en disco para realizar la operación.

Solución

Parte 1)

Cada dir_entry ocupa 32 bytes.
Cada sector tiene 4096 bytes.
Por lo tanto cada sector puede contener 4096/32 = 128 dir_entrys.
Un directorio tiene como máximo 15 sectores de disco.
Entonces un directorio puede tener como máximo 15*128 = 1920 archivos y/o directorios.

Parte 2)

Asumiendo que max_sectors_in_disk > 15* max_i_node_on_disk. Si consideramos que están ocupados todos los posibles inodos y que cada uno ocupa los 15 posibles sectores en disco tendríamos todas las posibles entradas ocupadas. La cantidad total de dir_entrys total (sumando la de todos los directorios) sería: 1920*(max_i_node_on_disk).

Parte 3)

Nota: La solución planteada comprueba que haya suficiente espacio en el disco para crear el archivo y, en caso de necesitarlo, para utilizar un nuevo bloque para el directorio.

```
procedure createFile(cam : array of char; id_dueno: Integer; id_grupo: Integer;
size: Integer; p: array [1..14] of bit; Var ok : Boolean)

  var base: array of char;
  var dir: array of char;
  var entry: integer;
  var entryLibre: integer;
  var bloqueConEntryLibre: integer;
  var sectorFAT: integer;
  var buffBloque: array [1..128] of dir_entry;
  var buff: array [1..128] of dir_entry;
  var buffCeros: array [1..4096] of byte;

  var isNuevoBloque: boolean;
  var auxSize: integer;
  var antFAT: integer;

begin
  dirname(cam, dir);
  dirname(cam, base);
  getInode(dir, num_inode_dir, ok);
  if(ok)
  {
    sectorFAT = 0;
    isNuevoBloque = false;

    entryLibre:= -1;
    bloqueData:= 0;
    entry = 0;
    while((bloqueData <= TD[num_inode_dir].tope) && ok)
    {
      if(entry == 0)
      {
        bloqueData:= bloqueData + 1;
```

```
      readBlock(D, TD[num_inode_dir].data[bloqueData], buff, ok);

      if(not ok)
        return;
    }
    if((buff[entry].used) && (buff[entry].name == base))
    {
      ok:= false;
      return;
    }

    if(!buff[entry].used && entryLibre == -1)
    {
      entryLibre := entry;
      bloqueConEntryLibre := TD[num_inode_dir].data[bloqueData];
      buffBloque:= buff;
    }

    entry = entry + 1;
    if(entry == 128)
      entry = 0;
  }

  if(ok)
  {
    if(entryLibre == -1)
    {
      if(TD[num_inode_dir].tope < 15)
      {
        while((sectorFAT < sectors_in_disk) && fat[sectorFAT] != -1)
        {
          sectorFAT++;
        }

        if(sectorFAT < sectors_in_disk)
        {
          buffBloque:= new dir_entry[128];
          for(i=0; i<128; i++)
          {
            buffBloque[i].used := false;
          }
          entryLibre:= 0;
          bloqueConEntryLibre:= sectorFAT;
          isNuevoBloque := true;

          TD[num_inode_dir].tope := TD[num_inode_dir].tope + 1;
          TD[num_inode_dir].data[TD[num_inode_dir].tope] := sectorFAT;
          fat[sectorFAT] := -2;
        }
        else
        {
          ok := false;
          return;
        }
      }
      else
      {
        ok := false;
        return;
      }
    }
```

```
buffBloque[entryLibre].used:= true;
buffBloque[entryLibre].name:= base;
buffBloque[entryLibre].type:= file;
buffBloque[entryLibre].tamano_archivo:= size;
buffBloque[entryLibre].id_dueno:= id_dueno;
buffBloque[entryLibre].id_grupo:= id_grupo;
buffBloque[entryLibre].i_node_num:= -1;
buffBloque[entryLibre].permisos:= p;
buffBloque[entryLibre].comienzo_archivo:= -1;

for(i = 0; 1 < 4096; i++)
{
  buffCeros[i] := 0;
}
auxSize := 0
while (auxSize < size && ok)
{
  while((sectorFAT < sectors_in_disk) && fat[sectorFAT] != -1)
  {
    sectorFAT++;
  }
  if((sectorFAT < sectors_in_disk))
  {
    if(auxSize == 0)
    {
      buffBloque[entryLibre].comienzo_archivo:= sectorFAT;
      fat[sectorFAT] := 0;
      antFAT := sectorFAT;
    }
    else
    {
      fat[antFAT] := sectorFAT;
      fat[sectorFAT] := 0;
      antFAT := sectorFAT;
    }

    writeBlock(D, sectorFAT, buffCeros, ok);
  }
  else
  {
    ok := false;
  }

  auxSize := auxSize + 4096;
}

if(ok && auxSize >= size)
{
  writeBlock(D, bloqueConEntryLibre, buffBloque, ok);
}
if(not ok || auxSize < size)
{
  ok := false;
  sectorFAT := buffBloque[entryLibre].comienzo_archivo;
  while(sectorFAT != 0)
  {
    antFAT := sectorFAT;
    sectorFAT := FAT[antFAT];
    FAT[antFAT] := -1;
  }
  if(isNuevoBloque)
  {
```

```
            fat[TD[num_inode_dir].data[TD[num_inode_dir].tope]] := -1;
            TD[num_inode_dir].tope := TD[num_inode_dir].tope - 1;
          }
        }
      }
    }
end
```

Problema 3 (32 puntos)

Se desea implementar la consulta y actualización de los registros de la base de datos de estudiantes de la Facultad de Ingeniería.

Hay un grupo de 5 funcionarios que consultan los registros de la base en busca de errores en lotes de 100 registros (compartido entre los 5 funcionarios). Si encuentran un error en un registro deben actualizar el mismo con la información corregida. Se debe permitir que varios funcionarios consulten a la vez el mismo registro. Un funcionario que quiere modificar un registro deberá tener prioridad sobre los que quieren leerlo.

Un funcionario A no podrá modificar un registro que haya sido modificado por otro funcionario B luego de la lectura de A; en ese caso deberá volver a leer y verificar nuevamente el registro.

Un supervisor cambiará el lote de 100 registros periódicamente, para lo cual ningún funcionario deberá estar usando el lote. El supervisor tendrá prioridad sobre los funcionarios para el acceso al lote.

Se dispone de las siguientes funciones:

- que_registro():{1..100}, que ejecutado por un funcionario retorna el número de registro a verificar.
- verificar(int: registro):boolean, que ejecutado por un funcionario retorna si el registro indicado debe ser modificado
- modificar(int: registro), que ejecutado por un funcionario actualiza el registro con la información corregida
- cambiar_lote(), que ejecutado por el supervisor cambia el lote de registros
- otras_tareas_supervisor(), que ejecutado por el supervisor realiza otras tareas luego de cambiar de lote de registros

Implementar utilizando monitores las tareas Funcionario y Supervisor. No se permiten tareas auxiliares.

Solución

```
procedure Supervisor()
begin
      while (true)
      {
            Lote.nuevo_lote();
            otras_tareas_supervisor();
      }
end Supervisor;

procedure Funcionario()
VAR idReg: Integer;
    pronto, modificar: boolean;
begin
      while (true)
      {
            idReg = que_registro();
            modificar := false;
            pronto := false;
            repeat
                  registros[idReg].inicio_consulta();
```

```
                  modificar = verificar(idReg);
                  if modificar
                        pronto := registros[idReg].inicio_modificacion();
                        if pronto
                              modificar(idReg);
                        endif
                  else
                        pronto := true;
                  endif
                  registros[idRed].fin_consulta(pronto && modificar);
                  Lote.sale_funcionario();
            until pronto;
      }
end Funcionario;

MONITOR Lote IS
VAR
      supervisor: Condition;
      funcionarios: Condition;
      quiere_cambiar: Boolean;
      cant_funcionarios: integer;

      procedure nuevo_lote()
      {
            if(cant_funcionarios == 0)
                  cambiar_lote();
            else
                  quiere_cambiar := true;
                  supervisor.wait();
                  cambiar_lote();
                  quiere_cambiar := false;
                  funcionarios.signal();
            endif
      }

      procedure entra_funcionario()
      {
            if(quiere_cambiar)
                  funcionarios.wait();
                  funcionarios.signal();
            endif;
            cant_funcionarios := cant_funcionarios + 1;
      }

      procedure sale_funcionario()
      {
            cant_funcionarios := cant_funcionarios - 1;
            if(cant_funcionarios == 0)
                  supervisor.signal();
            endif;
      }

begin
      quiere_cambiar := false;
      cant_funcionarios := 0;
end;
END MONITOR;

TYPE MONITOR REGISTRO

VAR:
      quiere_modificar : boolean;
```

```
      ok_consultar,ok_modificar : Condition;
      cant : integer;

      procedure inicio_consulta()
      {
            if(quiere_modificar)
                  ok_consultar.wait();
                  ok_consultar.signal();
            endif
            Lote.entra_funcionario();
            cant := cant + 1;
      }

      procedure inicio_modificacion():boolean
      {
            if(quiere_modificar)
                  return false;
            else
                  quiere_modificar := true;
                  if(cant > 1) // hay por lo menos un lector y un escritor
                        ok_modificar.wait();
                  endif;
                  return true;
            endif
      }

      procedure fin_consulta(boolean fin_modificacion)
      {
            cant := cant - 1;
            if(fin_modificacion)
                  quiere_modificar := false;
                  ok_consultar.signal();
            else
                  if(cant == 1)
                        ok_modificar.signal;
                  endif
            endif
      }

Begin
      quiere_modificar := false;;
      cant := 0;
End


END MONITOR;
VAR registros: array [1..100] of REGISTRO;

//main
begin
      cobegin
            Funcionario();
            Funcionario();
            Funcionario();
            Funcionario();
            Funcionario();
            Supervisor();
      coend
end
```