PRÁCTICO 6

Objetivos

- Familiarizarse con el uso de primitivas de sincronización con pasaje de mensajes.
- Usar Ada en la solución de problemas de sincronización.
- Familiarizarse con los problemas de deadlock en la ejecución de procesos concurrentes.

Ejercicio 1 (medio)

Resolver el ejercicio 9 del práctico 5 usando mailboxes. Se debe indicar que semántica de mailbox de las vistas en el curso se utiliza en la solución.

Ejercicio 2 (medio)

En un ambiente concurrente se desea garantizar la mutua exclusión de una sección crítica y se dispone únicamente de las siguientes primitivas para la intercomunicación de procesos :

(a)

`send(tarea, mensaje)`	no bloqueante
`receive(mensaje)`	bloqueante

(b) Mejorar la solución anterior con las siguientes primitivas:

`send(tarea, mensaje)`	bloqueante
`receive(mensaje)`	bloqueante

Ejercicio 3 (medio)

Se desea modelar usando mailboxes la atención de un peaje de ***n*** cajas a vehículos.

Los autos deberán darle al cajero el número de tarjeta de crédito y este le devolverá un numero de ticket para lo cual contará con la función:

```
function pago(nro_de_tarjeta : in integer): integer
```

Los autos deberán elegir la caja con menor cantidad de autos en cola.

Solo se podrá implementar una tarea auxiliar (`Admin`).

El programa principal ejecutará el siguiente código:

```
begin
   cobegin
     Admin
     Auto(nro_tarjeta1)
       ...
     Auto(nro_tarjetaM)
     Caja(1)
       ...
     Caja(n)
   coend
end
```

Ejercicio 4 (en clase)

Se desea modelar utilizando mailboxes el siguiente problema:

El taller mecánico "*Boxes*" tiene capacidad para 20 vehículos en sus instalaciones. La entrada de los usuarios a las instalaciones debe ser estrictamente en orden de llegada (la solución no debe permitir bajo ningún concepto "colados").

El usuario debe ser atendido por el primer mecánico libre de los 5 con que cuenta el taller. El mecánico, una vez terminado el arreglo, le indicará a la caja el monto a cobrar por el arreglo.

El cliente recibirá de la caja el monto a pagar.

Se dispone de las siguientes funciones:

- **arreglar_auto() :integer**

  Invocada por un mecánico para arreglar el auto. Retorna el costo del arreglo.

- **pagar_arreglo(Monto)**

  Invocada por el usuario para pagar el monto que le fue indicado por la caja.

Notas:

- Se prohíbe expresamente el busy-waiting.
- Se debe explicitar la semántica de las primitivas de mailbox utilizadas.
- Tener cuidado de que el socio pague el importe correcto del arreglo a la caja.

Ejercicio 5 (básico)

Un proceso `sumador` recibe números enteros desde dos procesos independientes. Los números que obtiene así deben ser sumados y la actividad de `sumador` finaliza cuando transcurren 10 segundos sin recibir ningún numero o cuando se llega a un total de 100 números, cualquiera sea su origen. Escribir en Ada la tarea `sumador`.

Ejercicio 6 (básico)

Para la realidad planteada en el ejercicio 4 del práctico 5.

Se pide construir un programa Ada con tres tareas que permitan controlar adecuadamente estos procesos.

Ejercicio 7 (en clase)

Sea un consultorio médico, el cual posee una sala de espera en la cual pueden haber como máximo 10 pacientes. El médico atiende a una sola persona de la sala de espera por vez, teniendo prioridad los niños. Cuando el médico termina de atender al paciente deja que el próximo entre. Si no hay ninguno lee durante cinco minutos y vuelve a ver si hay alguien. Si no lo hay vuelve a leer y así sucesivamente.

Se dispone de las siguientes funciones:

- **atender()**

  Invocada por el doctor para atender un paciente.

- **leer_diario()**

  Invocada por el doctor para leer el diario.

Se pide:

Representar mediante tareas de ADA los pacientes, el doctor y la sala de espera.

Nota:

El paciente debe solicitar permiso al portero para entrar a la sala y al médico para entrar al consultorio.

Ejercicio 8 **(medio)**

Implementar semáforos no binarios con ADA

Ejercicio 9 **(avanzado)**

Un boliche bailable dispone de un baño con capacidad para 4 hombres, y otro baño para mujeres con igual capacidad, un limpiador y un reponedor de insumos (papel higiénico, jabón y toallas).

El limpiador descansará 15 minutos entre cada limpieza de un baño, mientras que el reponedor descansará 10 minutos luego de reponer un baño.

Por otro lado los materiales de limpieza y de reposición se encuentran en una despensa a la que solo puede entrar una persona por vez.

Ni el limpiador ni el reponedor pueden entrar al baño hasta que no salgan las personas que se encontraban dentro. Además, las personas no pueden entrar a un baño si se encuentra el limpiador y/o el reponedor en él.

Una vez que el limpiador y/o el reponedor indican su voluntad de entrar al baño, las personas que estén esperando para entrar deberán abstenerse de hacerlo hasta que el personal termine su tarea.

El limpiador y el reponedor pueden realizar sus tareas en el mismo baño a la vez.

Se desea modelar en ADA las tareas Persona, Limpiador, Reponedor, Despensa y Baño.

Notas:

No se podrán implementar tareas auxiliares. Considerar que no hay límite para la cantidad de personas dentro del boliche.

No debe haber más de una entrada para la comunicación entre el limpiador y la despensa. Ídem entre el reponedor y la despensa.

Se dispone de las siguientes funciones y procedimientos:

- que_soy():{0, 1} que debe ser ejecutado por la persona. Devuelve 0 si para personas de sexo masculino.
- utilizar_sanitario(), que debe ser ejecutado por una persona para utilizar el sanitario una vez entrado en el baño.
- elegir_proximo_baño():integer 1..2. Devuelve al azar el baño que le toca limpiar o reponer. Debe ser ejecutado por el limpiador y el reponedor.
- calcular_insumos():insumos. Debe ser ejecutado por el limpiador y el reponedor para saber que insumos necesita retirar de la despensa.
- obtener_insumos(insumos). Debe ser ejecutado por la despensa para retirar los insumos necesarios.
- limpiar(cantidad_de_personas:integer). Debe ser ejecutado para limpiar el baño, siendo cantidad_de_personas las que entraron al baño desde la última vez que se limpió, para

determinar el grado de suciedad del baño. El estudiante determinará donde ejecutar esta función (baño o limpiador).

- reponer(). Debe ser ejecutado para reponer los insumos del baño. El estudiante determinará donde ejecutar esta función (baño o reponedor).

Ejercicio 10 (avanzado)

La heladería “La Marsellesa” tiene 4 vendedores, y vende cucuruchos de un solo gusto de entre 20 disponibles.

Los clientes que ingresan se forman en una única fila por orden de llegada, hasta que un vendedor pueda atender al primero, y así sucesivamente. La cantidad máxima de clientes no es conocida.

La heladería tiene un recipiente por cada gusto. Los vendedores no pueden extraer helado a la vez del mismo recipiente.

Se pide: Modelar en Ada las tareas Cliente y Vendedor.

Se dispone de los siguientes procedimientos:

- **que_helado_quiero(out:gusto)** Llamada por los clientes y que devuelve el gusto de helado que va a pedir.
- **armar_helado(in:gusto)** Llamada por los vendedores para poner el helado en el cucurucho.
- **entregar_o_recibir_helado()** Deberá ser ejecutado por el cliente o el vendedor para que el cliente reciba el helado (solamente debe ser llamada por una de las dos tareas).
- **comer_helado()** Llamada por los clientes para comer el helado que le sirvieron

Nota: Se pueden utilizar tareas auxiliares

Ejercicio 11 (medio)

Sea una maquina de tejer, constituida por las siguientes unidades :

- **Tres expendedoras de hilo**: ofrecen hilo a la unidad tejedora mediante el ofrecimiento del encuentro `quiero_hilo` cuando lo hay. El procedimiento `hay_hilo` indicara la falta del mismo, en cuyo caso se deberá invocar a `cambiar_rollo` que culminara cuando el operador ponga uno.
- **Tejedora**: pide el hilo a las expendedoras invocando a los encuentro `quiero_hilo` y teje utilizándolos completamente invocando para ello a la rutina `tejer`. Considerar que la unidad tejedora comienza a tejer una vez que tiene el hilo de las 3 expendedoras.
- **Control**: es la encargada de verificar la producción debiendo invocar la alarma si transcurre más de un minuto sin que la tarea tejedor este tejiendo (proponer la interfaz con tejedora).

Se pide:

Implementar utilizando Ada el sistema descrito de forma de que cada unidad sea una tarea.

Ejercicio 12 (básico)

Una computadora tiene seis unidades de cinta, con *n* procesos compitiendo por ellas. Cada proceso puede necesitar dos unidades. Para que valores de *n* el sistema esta libre de deadlock.

Ejercicio 13 (medio)

Considere un sistema con 5 procesos ejecutando y con 4 tipos de recursos. La siguiente tabla indica cuantos recursos tiene asignado cada proceso y cual es la cantidad máxima que se necesita. Además se indica en la última columna la cantidad disponible de cada recurso.

	Asignados	Máximo	Disponible
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Conteste las siguientes preguntas utilizando el algoritmo del banquero:

(a) ¿El sistema está en un estado seguro?

(b) Si llega el pedido (0, 4, 2, 0) del proceso P_1, ¿puede ser satisfecho el pedido inmediatamente?

Solución parte (a)

Un **estado** es **seguro** si el sistema puede asignar recursos a cada proceso (hasta su máximo) en algún orden y aún así evitar los bloqueos mutuos.

Más formalmente, un sistema está en un estado seguro solo si existe una secuencia segura. Una secuencia de procesos [P_0, P_1, ..., P_n] es una secuencia segura para el estado de asignación actual si, para cada P_i, los recursos que P_i todavía puede solicitar se pueden satisfacer con los recursos que actualmente están disponibles más los recursos que tienen todos los P_j, donde j < i. En esta situación, si los recursos que P_i necesita todavía no están disponibles, P_i podrá esperar hasta que todos los P_j hayan terminado. En ese momento, P_i podrá obtener todos los recursos que necesita, llevar a cabo su tarea designada, liberar los recursos que adquirió y terminar.

Vamos a ver si el sistema está en estado seguro. Aplicando el ***Algoritmo de Seguridad*** (Silverschatz-Galvin)

trabajo$_0$ = (1 5 2 0)
fin$_0$ = (False False False False False)

	Necesidades[1]
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_4	0 6 4 2

Elijo P_0 (Necesidad(0) ≤ trabajo$_0$: (0 0 0 0) ≤ (1 5 2 0)):
trabajo$_1$ = trabajo$_0$ + Asignados(0) =
(1 5 2 0) + (0 0 1 2) = (1 5 3 2)
fin$_1$ = (True False False False False)

Elijo P_2 (Necesidad(2) ≤ trabajo$_1$: (1 0 0 2) ≤ (1 5 3 2)):

[1] Necesidades = (Máximo – Asignados)

trabajo$_2$ = trabajo$_1$ + Asignados(2) =
(1 5 3 2) + (1 3 5 4) = (2 8 8 6)
fin$_2$ = (True False True False False)

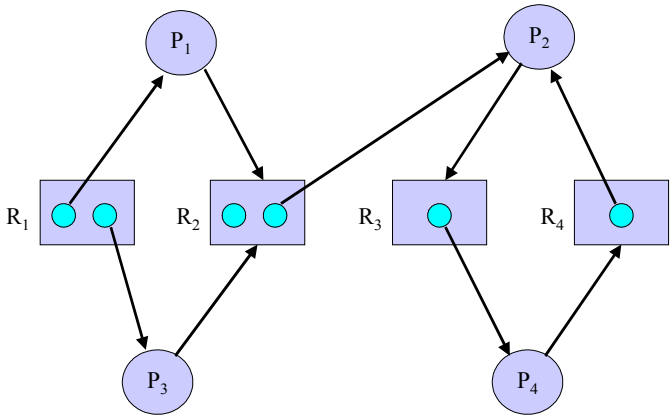
Elijo P_4 (Necesidad(4) ≤ trabajo$_2$: (0 6 4 2) ≤ (2 8 8 6)):
trabajo$_3$ = trabajo$_2$ + Asignados(4) =
(2 8 8 6) + (0 0 1 4) = (2 8 9 10)
fin$_3$ = (True False True False True)

Elijo P_1: (Necesidad(1) ≤ trabajo$_3$: (0 7 5 0) ≤ (2 8 9 10)):
trabajo$_4$ = trabajo$_3$ + Asignados(1) =
(2 8 9 10) + (1 0 0 0) = (3 8 9 10)
fin$_4$ = (True True True False True)

Elijo P_3 (Necesidad(3) ≤ trabajo$_4$: (0 0 2 0) ≤ (3 8 9 10)):
trabajo$_5$ = trabajo$_4$ + Asignados(3) =
(3 8 9 10) + (0 6 3 2) = (3 14 12 12)
fin$_5$ = (True True True True True)
Verificación:
Asignados$_{ini}$ + Disponibles$_{ini}$ = Trabajo$_{fin}$
(2 9 10 12) + (1 5 2 0) = (3 14 12 12)

Entonces existe una secuencia [P_0, P_2, P_4, P_1, P_3] que es segura.
Entonces es estado seguro y por lo tanto no tiene que haber deadlock.

Ejercicio 14 (básico)



Considere el grafo de recursos reusables de la figura.

(a) ¿Cuáles procesos están bloqueados?

(b) ¿Cuáles procesos están en deadlock?

(c) ¿El estado es un estado de deadlock?

Ejercicio 15 (básico)

Considere tres procesos P_1, P_2, y P_3 ejecutando concurrentemente con la siguiente secuencia de código:

P_1	P_2	P_3
M	M	M
P(x)	P(y)	P(z)
M	M ◀	M
P(z) ◀	V(y)	P(x) ◀
M		M
V(x)		V(z)
M		M
V(z)		V(x)

El símbolo "◀" en cada columna indica que instrucción de cada proceso se esta ejecutando actualmente. Todos los semáforos fueron inicializados en uno.

(a) Dibuje un grafo de recurso reusable describiendo esta situación donde cada semáforo esta representado como un recurso, y P y V representan pedidos y liberaciones de recursos.

(b) Reduzca el grafo hasta donde sea posible, muestre si representa un estado de deadlock.

(c) Si Ud. pudiera aumentar la cantidad de unidades de cualquiera de los tres recursos, ¿cuál aumentaría? ¿Resolvería el deadlock?