

## CAPÍTULO 1. PLANIFICACIÓN DE PROCESO

### INTRODUCCIÓN

Los sistemas operativos se encargan de la gestión de los elementos del computador: procesadores, memoria central, discos, terminales, conexiones de red, E/S, ...

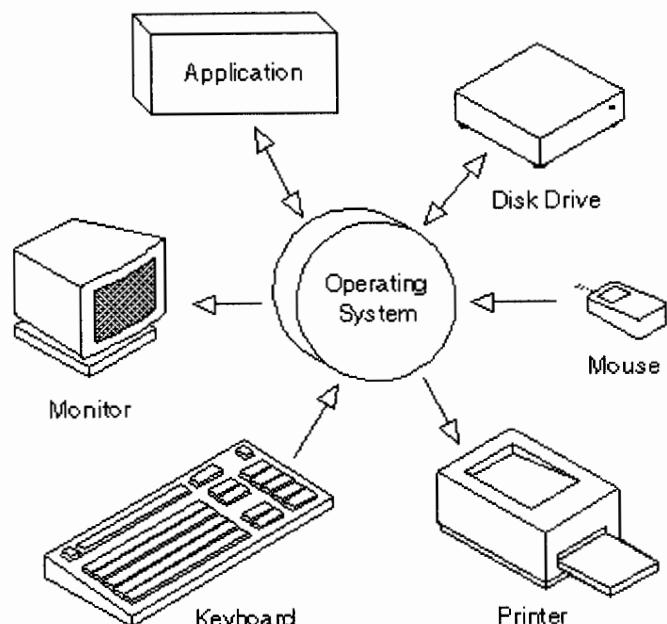


Figura 1-1: Sistema Operativo y sus comunicaciones HW y SW.

Programa que controla la ejecución del resto de programas y actúa como interfaz entre los usuarios del computador y el hardware del mismo.

Las funciones principales de un sistema operativo son:

- Manipulación de Interrupciones
- Gestión de procesos
- Gestor de memoria
- Gestión de E/S

- Mantenimiento de estructuras de datos básicas
- Y los módulos básicos de que dispone para el tratamiento de estas funciones son:

- Manejador de interrupciones
- Planificador y Cargador
- Primitivas de sincronización y comunicación

Por este motivo el sistema operativo es un programa que está en permanente ejecución en memoria principal.

En el presente libro se van a tratar en especial los temas de planificación y gestión de procesos y las primitivas de sincronización y comunicación, además de la programación de llamadas al sistema sobre el estándar POSIX.

La gestión de procesos se encarga de gestionar adecuadamente los procesos. La definición de un proceso es la de un programa en ejecución. Por ello, hay que diferenciar los términos proceso y programa.

Los servicios que ofrece el SO para una gestión óptima son:

- Ejecución concurrente
- Sincronización de procesos
- Comunicación entre procesos

Los estados de un proceso pueden ser:

- Suspendido
- Activado
  - Ejecución
  - Preparado
  - Espera

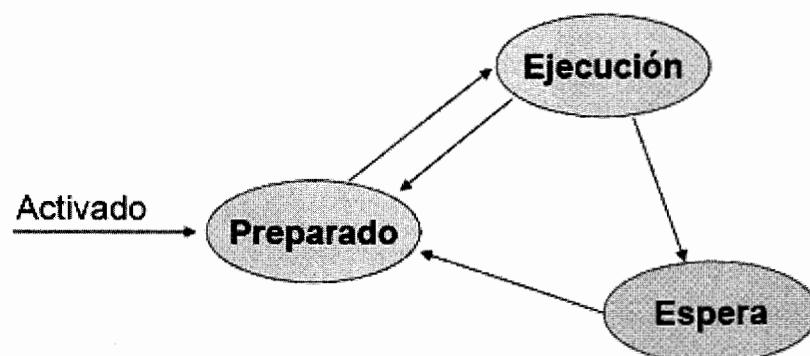


Figura 1-2: Estados de un proceso.

La justificación de la existencia de una planificación para los procesos se debe a la escasez de recursos, existencia de recursos de acceso exclusivo como son la CPU, impresoras, etc... Por este motivo debe de existir una política de asignación.

Las situaciones que se plantean en la planificación son:

- Cuando un proceso pasa del estado de ejecución a espera
- Cuando un proceso pasa del estado de ejecución a preparado
- Cuando un proceso pasa del estado de espera al estado de preparado
- Cuando un proceso termina

Tipos de planificación

- Expropiativa
- No expropiativa

El planificador de la CPU es el scheduler mientras que el cargador el dispatcher. Que se encargan de cambiar de contexto y saltar al punto apropiado del proceso para continuar desde ahí.

Los criterios de planificación buscan mejorar algunos de los siguientes parámetros:

- Utilización de la CPU
- Rendimiento
- Tiempo de retorno
- Tiempo de espera
- Tiempo de respuesta

#### Algoritmos de planificación no expropiativos:

- Servicio por orden de llegada (FCFS)

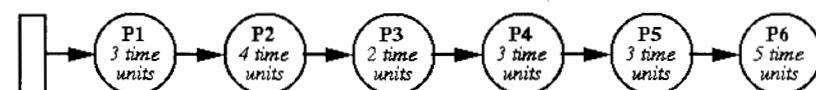


Figura 1-3: Algoritmo FCFS.

- Planificación con prioridad
  - Estática
  - Dinámica
- Primero la tarea más corta (SJF)

### Algoritmos de planificación expropiativos:

- Planificación por turno circular (RR)

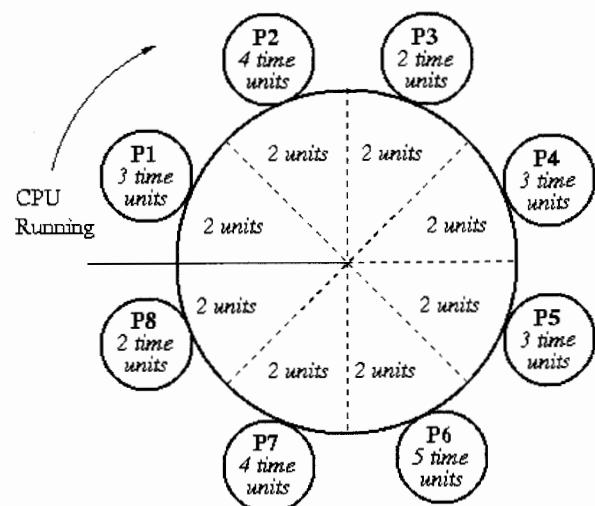


Figura 1-4: Algoritmo RR.

- Planificación con prioridad
- Tiempo que queda más corto (SRT)
- Planificación con colas multinivel (MLQ)
  - Número de colas
  - Algoritmo de planificación para cada cola
  - Prioridad de las colas
  - Criterio que indique la cola en la que entra una tarea cuando necesite un servicio
- Planificación con colas multinivel con realimentación (MLFQ)
  - Criterio que determine el movimiento de las tareas entre las colas

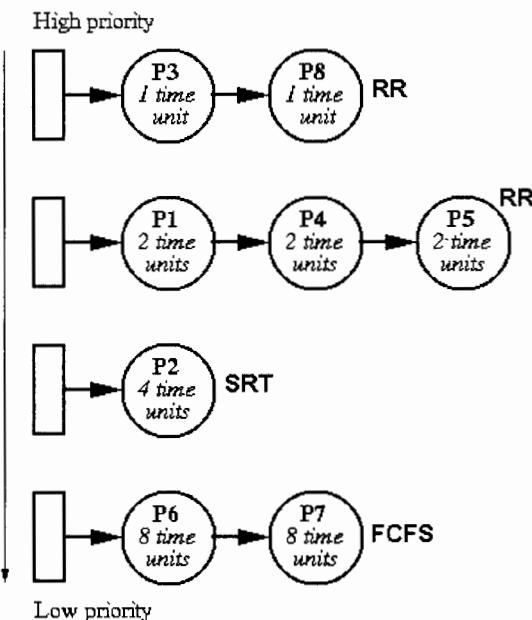


Figura 1-5: Planificación MLFQ.

- Planificación de múltiples procesadores
- Planificación en tiempo real

### EJERCICIO 1

Sea un sistema con los procesos que se muestran en la tabla 1-1.Calcular el tiempo de retorno, de espera y de respuesta de cada uno de los trabajos y representar su evolución temporal para los siguientes algoritmos de planificación.

- a) FCFS
- b) SJF
- c) SRT

Tabla 1-1: Esquema de ejecución de los procesos.

Proceso	Tiempo de llegada (ms)	Tiempo de ejecución (ms)
A	0	11
B	1	2
C	2	4
D	3	3
E	4	7
F	5	1

Solución

a) FCFS

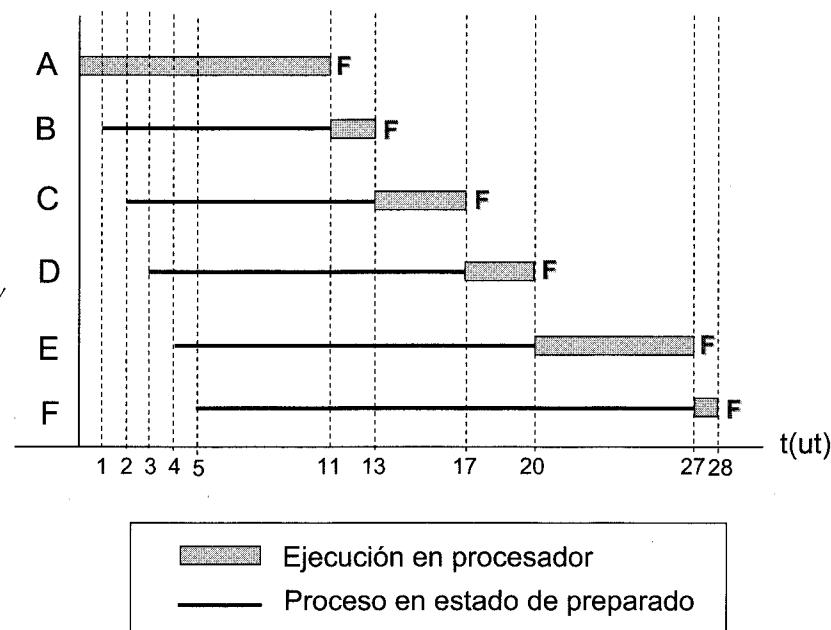


Figura 1-6: Evolución temporal de los procesos en el sistema con planificación FCFS.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-2: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t.espera
A	0	11	0
B	10	12	10
C	11	15	11
D	14	17	14
E	16	23	16
F	22	23	22
Promedio	12,16	16,83	12,16

b) SJF

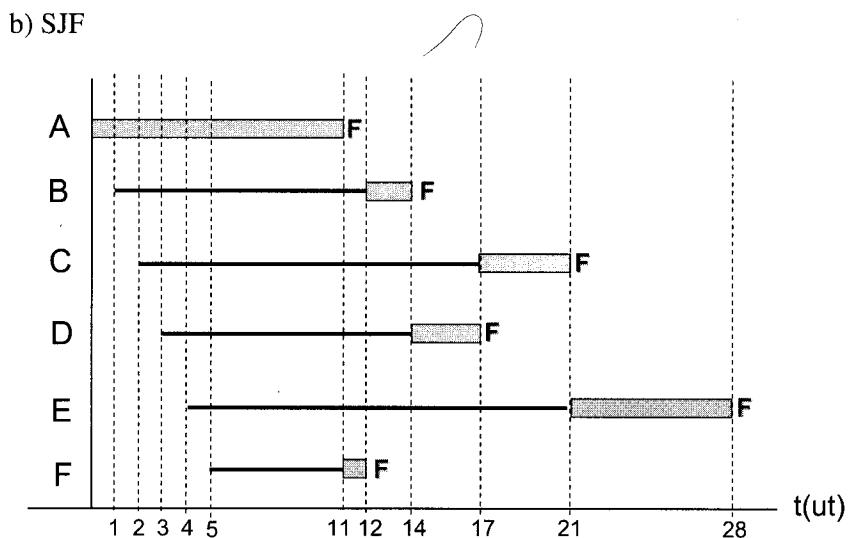


Figura 1-7: Evolución temporal de los procesos en el sistema con planificación SJF.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-3: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t.espera
A	0	11	0
B	11	13	11
C	15	19	15
D	11	14	11
E	17	24	17
F	6	7	6
Promedio	10,00	14,6	10

### c) SRT

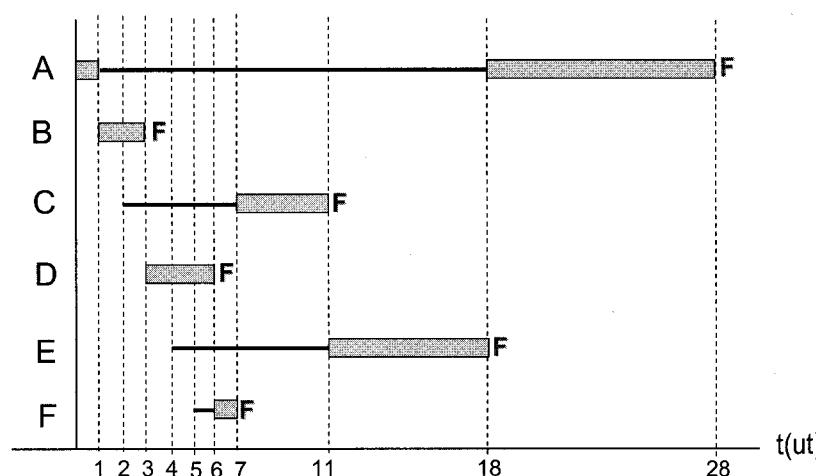


Figura 1-8: Evolución temporal de los procesos en el sistema con planificación SRT.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-4: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t. espera
A	0	28	17
B	0	2	0
C	5	9	5
D	0	3	0
E	7	14	7
F	1	2	1
Promedio	2,16	9,66	5,00

### EJERCICIO 2

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 3$  ut. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 1 ut., el proceso C en 2 ut., el proceso D en 3 ut. y el proceso E en 4 ut.

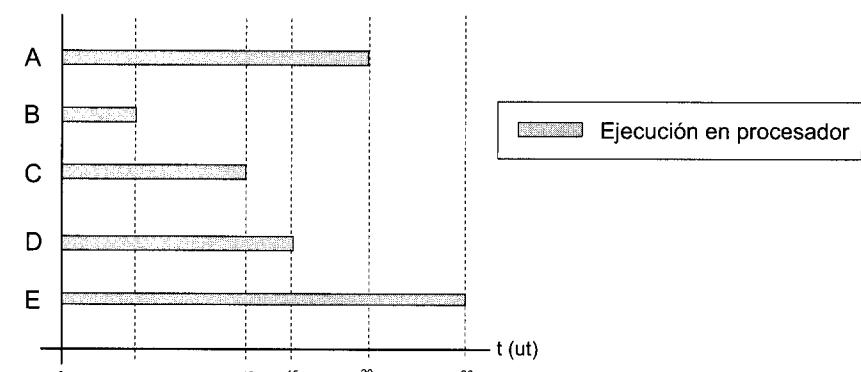


Figura 1-9: Esquema de ejecución de los procesos en el sistema.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

*Solución*

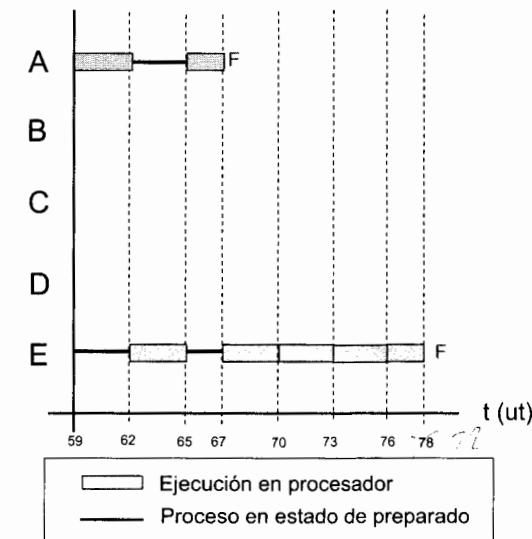
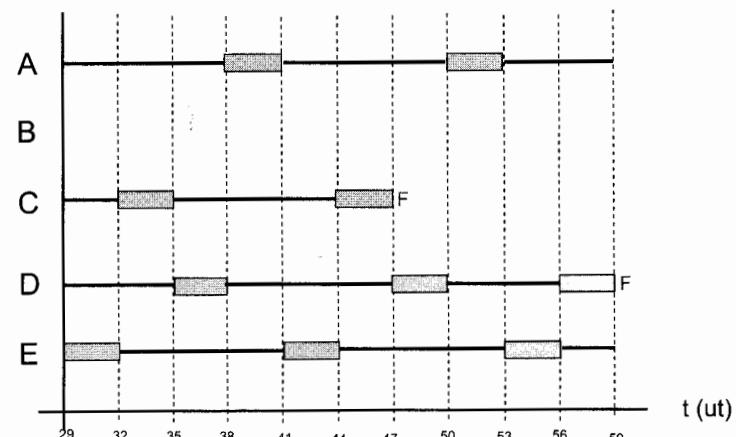
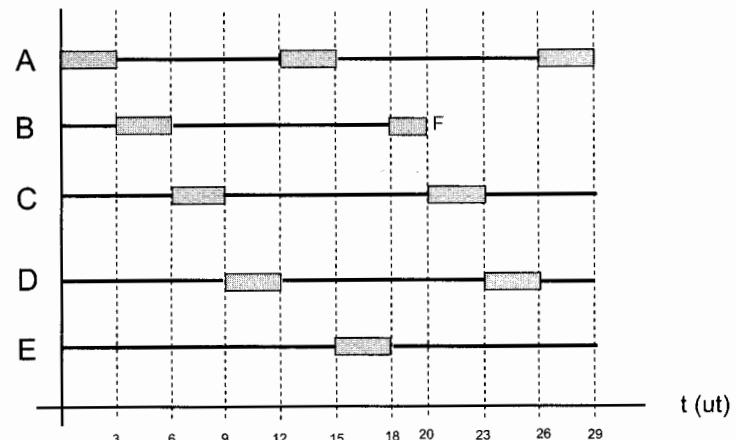


Figura 1-10: Evolución temporal de los procesos en el sistema.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-5: Tiempos de retorno, respuesta y espera de los procesos.

	t. retorno	t. respuesta	t. espera
A	67	0	47
B	19	2	14
C	45	4	33
D	56	6	41
E	74	11	48

## EJERCICIO 3

Se dispone de un sistema monoprocesador con política de planificación del procesador SRT. La ejecución de los procesos al sistema sigue el esquema descrito en la figura. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 1 ut., el proceso C en 2 ut., el proceso D en 3 ut. y el proceso E en 4 ut.

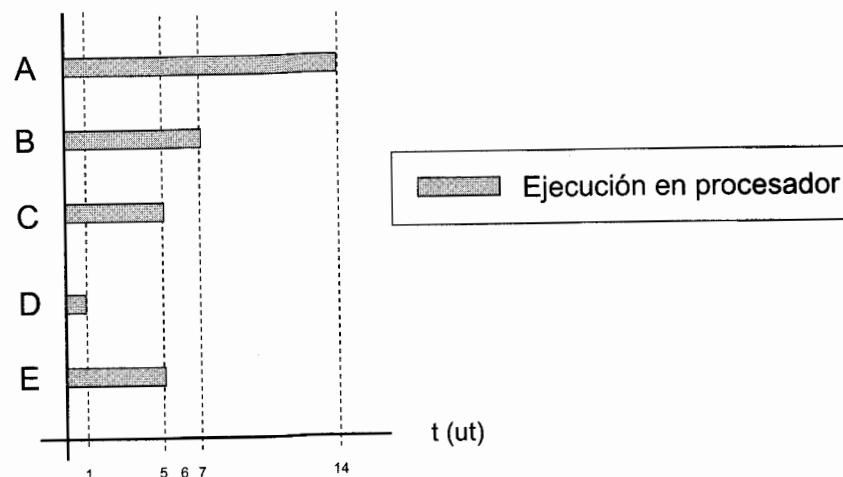


Figura 1-11: Esquema de ejecución de los procesos en el sistema.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

## Solución

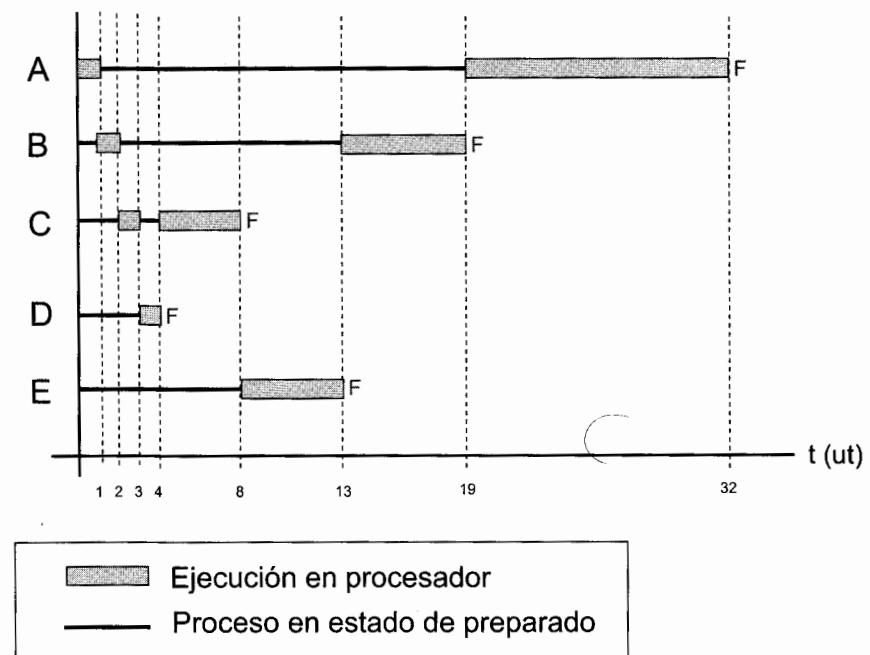


Figura 1-12: Evolución temporal de los procesos en el sistema.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-6: Tiempos de retorno, respuesta y espera de los procesos.

	t. retorno	t. respuesta	t. espera
A	32	0	18
B	18	0	11
C	6	0	1
D	1	0	0
E	9	4	4

## EJERCICIO 4

Se dispone de un sistema monoprocesador con política de planificación del procesador por prioridades. La ejecución de los procesos en el sistema sigue el esquema descrito en la figura. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 1 ut., el proceso C en 2 ut., el proceso D en 3 ut. y el proceso E en 4 ut.

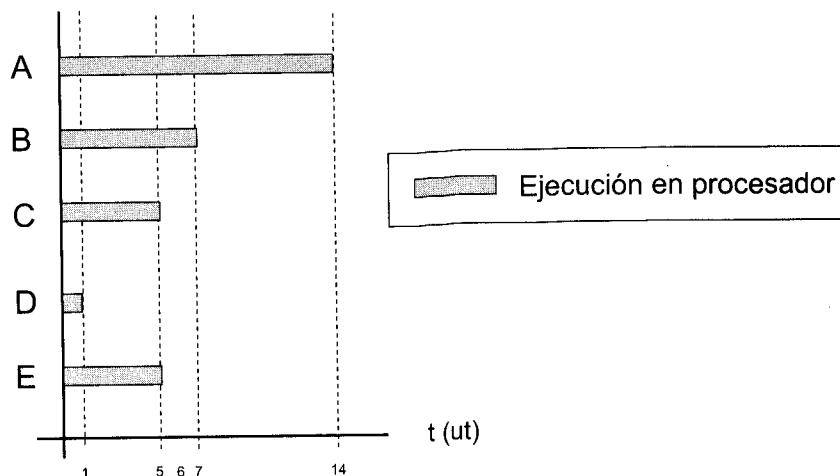


Figura 1-13: Esquema de ejecución de los procesos en el sistema.

Los procesos tienen las prioridades que se muestran en la tabla siguiente, donde un menor número indica mayor prioridad (1 tiene más prioridad que 14).

Tabla 1-7: Prioridades de los procesos.

A	14
B	7
C	5
D	1
E	5

Mostrar la evolución temporal de los procesos del sistema.

## Solución

Este es uno de los algoritmos de planificación más sencillos, pero se incluye para mostrar al lector que el algoritmo SRT no es más que un algoritmo de prioridades donde la prioridad es el tiempo, más concretamente a menor necesidad de tiempo mayor prioridad.

Observemos que las necesidades de ejecución de los procesos descritas en el enunciado coinciden con otro de los ejercicios de este tema en el que se ha aplicado el algoritmo de planificación SRT. Notemos también que según la tabla de prioridades, el valor de la prioridad de cada proceso coincide con su tiempo de ejecución. Además se ha definido la prioridad de forma que el menor valor tiene más prioridad.

Teniendo en cuenta el párrafo anterior que se extrae de la definición del problema, podemos comprobar que el cronograma resultante es exactamente igual que el del problema con planificación SRT mencionado. Hecho lógico porque el algoritmo SRT es un caso particular del algoritmo de prioridades, donde es más prioritario el proceso más corto.

Hasta el instante 3, cada proceso que va llegando es más prioritario que el anterior, por lo que obtiene la CPU. El proceso D que pasa a ejecución en este instante sólo necesita un instante, por lo que termina en el 4.

En el instante 4, llega el proceso E que tiene la misma prioridad que el C, y en este instante son los dos procesos más prioritarios. Como se muestra en el cronograma, pasa a ejecución el proceso C (y no el E) por que estaba antes que el E. Obviamente, cuando finalice C entrará E.

En el instante 13, el algoritmo tiene que decidir entre los dos procesos que quedan en el sistema, A y B. Pasa a ejecutarse B por tener el doble de prioridad que A. Finalmente se terminará de ejecutar A.

Podemos observar que el proceso A, pese a ser el primer proceso en llegar al sistema ha sido el último en terminar por ser el menos prioritario. Es probable que en una dinámica normal de un sistema, en el que van llegando procesos continuamente, si los procesos que van llegando son más prioritarios, los procesos menos prioritarios nunca se ejecutarían. A esto se le conoce como problema de la inanición, y como se puede observar, el algoritmo de prioridades presenta este problema.

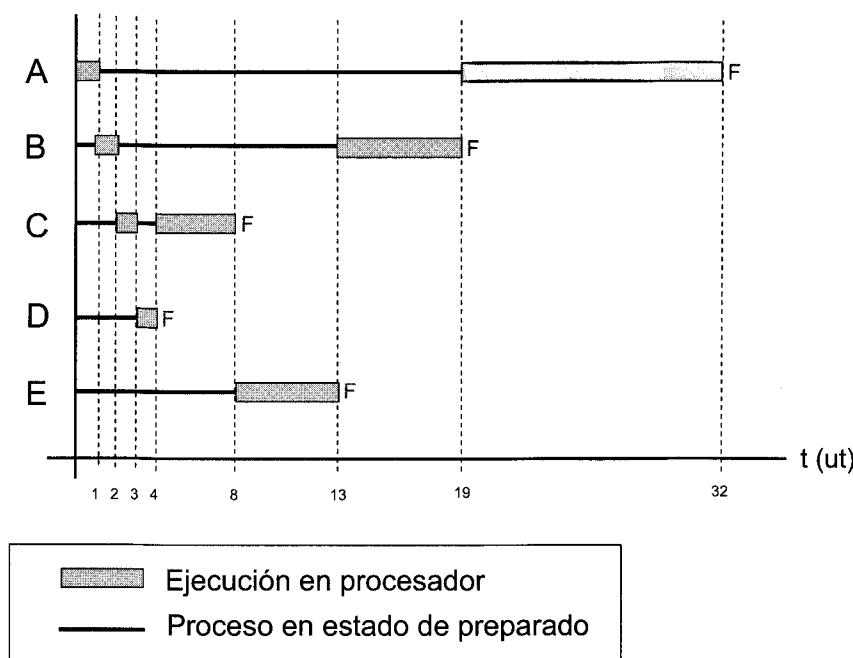


Figura 1-14: Evolución temporal de los procesos en el sistema.

#### EJERCICIO 5

Sea un sistema con los procesos que se muestran en la tabla 1. Calcular el tiempo de retorno, de espera y de respuesta de cada uno de los trabajos y representar su evolución temporal para los siguientes algoritmos de planificación.

- a) FCFS
- b) SJF y SRT
- c) Prioridades
- d) RR con  $q = 1$  ms.

Tabla 1-8: Esquema de ejecución de los procesos.

Proceso	Tiempo de ejecución (ms)	Prioridad
A	6	4
B	2	2
C	2	2
D	3	3
E	1	9

Todos los procesos llegan al sistema en el mismo instante.

#### Solución

##### a) FCFS

Como todos los procesos llegan en el mismo instante se escogen según su orden alfabético. Podría usarse cualquier otro criterio para decidir el orden en el que entran.

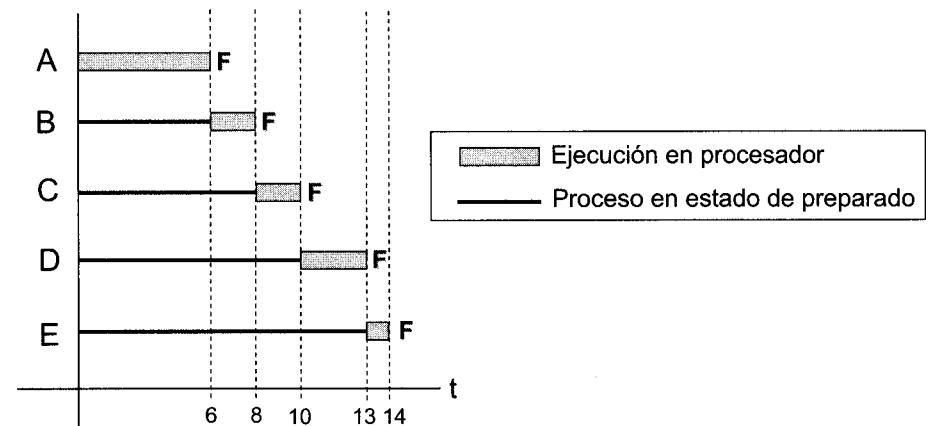


Figura 1-15: Evolución temporal de los procesos en el sistema con planificación FCFS.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-9: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t. espera
A	0	6	0
B	6	8	6
C	8	10	8
D	10	13	10
E	13	14	13
Promedio	7,4	10,2	7,4

## b) SJF y SRT

En este caso, ambos algoritmos de planificación producen los mismos resultados ya que todos los procesos llegan a la vez.

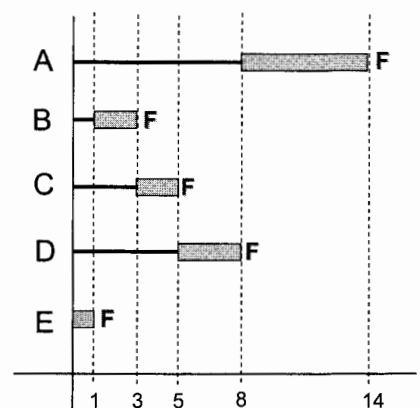


Figura 1-16: Evolución temporal de los procesos en el sistema con planificación SRT y SJF.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-10: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t. espera
A	8	14	8
B	1	3	1
C	3	5	3
D	5	8	5
E	0	1	0
Promedio	3,4	6,2	3,4

## c) Prioridades

En la política de planificación con prioridades se debe indicar si es expropiativa o no y si es creciente con el valor numérico o decreciente. En este caso el resultado es el mismo en los casos de expropiación y sin ella, ya que todos los procesos llegan a la vez. Como en el enunciado no se indica nada suponemos que a mayor valor menor prioridad. La figura siguiente muestra la evolución temporal en la ejecución de los procesos.

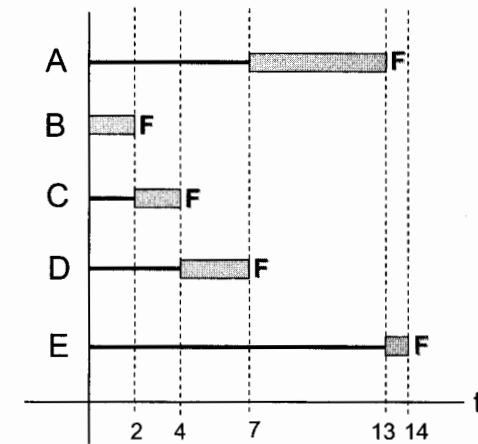


Figura 1-17: Evolución temporal de los procesos en el sistema con planificación con prioridades.

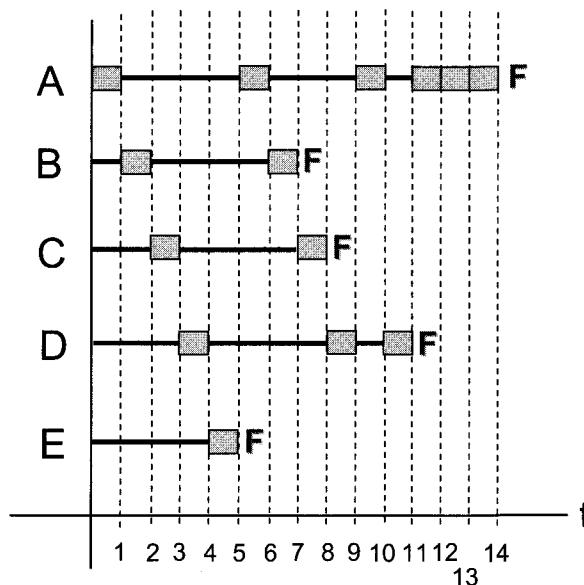
La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-11: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t. espera
A	7	13	7
B	0	2	0
C	2	4	2
D	4	7	4
E	13	14	13
Promedio	5,2	8	5,2

d) RR con  $q = 1$  ms.

La figura siguiente muestra la evolución temporal en la ejecución de los procesos.

Figura 1-18: Evolución temporal de los procesos en el sistema con planificación con RR ( $q=1$ ).

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-12: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t. espera
A	0	14	8
B	1	7	5
C	2	8	6
D	3	11	8
E	4	5	4
Promedio	2	9	6,2

#### EJERCICIO 6

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 10$  ut. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. El proceso A llega al sistema en el instante 0 ut., el proceso B y el proceso C en 2 ut (aunque se deben de atender en este orden), el proceso D en 12 ut. y el proceso E en 14 ut.

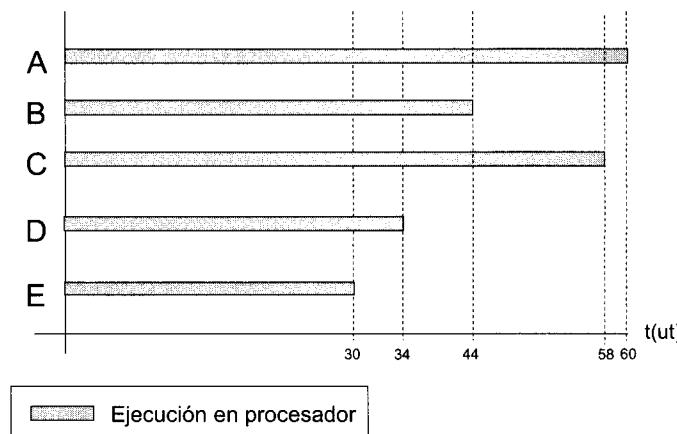


Figura 1-19. Ejecución de los procesos del sistema.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

*Solución*

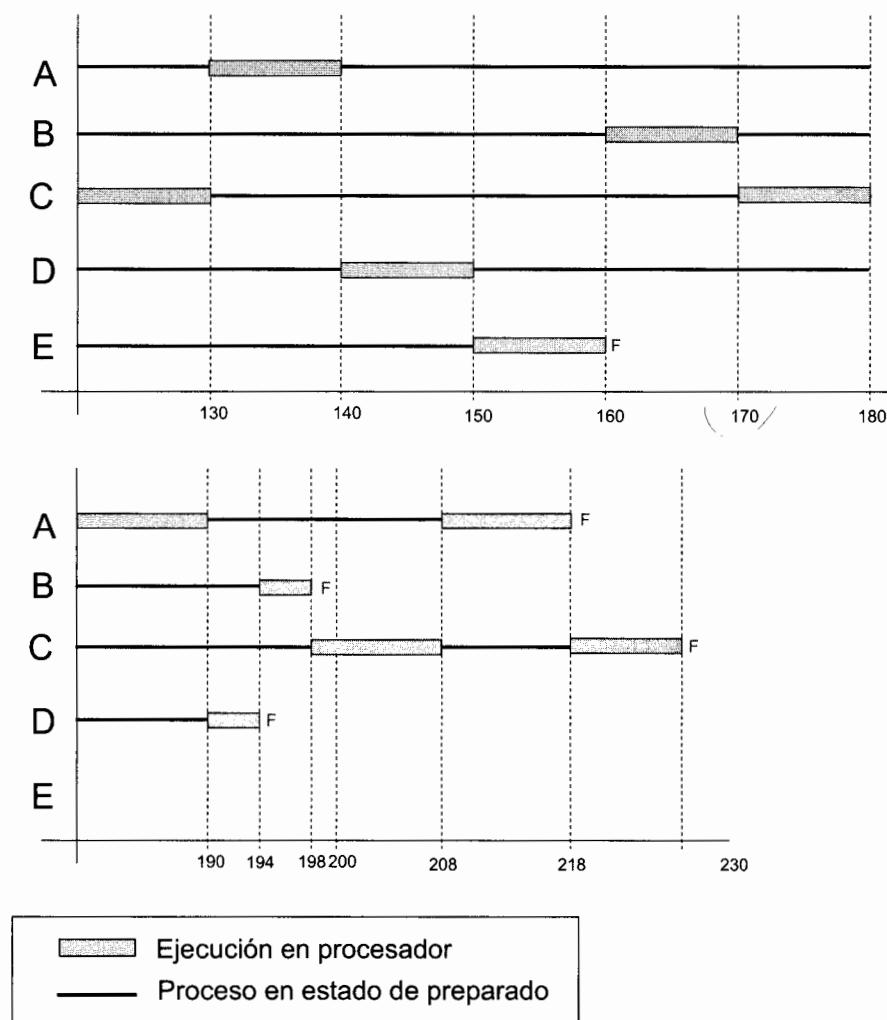
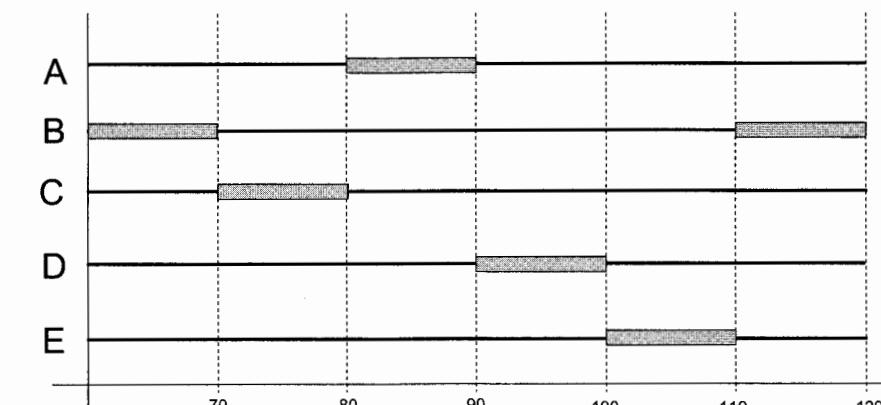
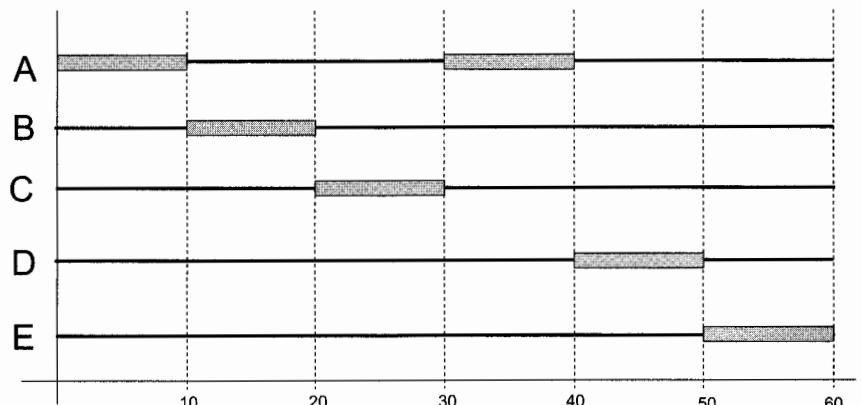


Figura 1-20: Evolución temporal de los procesos en el sistema.

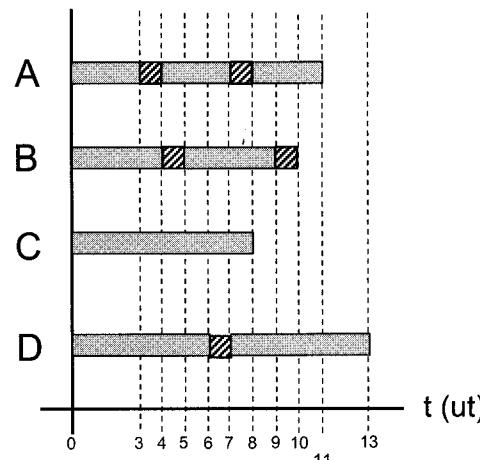
La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-13: Tiempos de retorno, respuesta y espera de los procesos.

	t. retorno	t. respuesta	t. espera
A	218	0	158
B	196	8	152
C	228	18	170
D	182	28	148
E	146	36	116

## EJERCICIO 7

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 1$  ut y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1-21. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 2 ut., el proceso C en 4 ut. y el proceso D en 6 ut.



■ Proceso x usando la CPU  
■ Operación de E/S

Figura 1-21. Esquema de ejecución de los procesos.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

## Solución

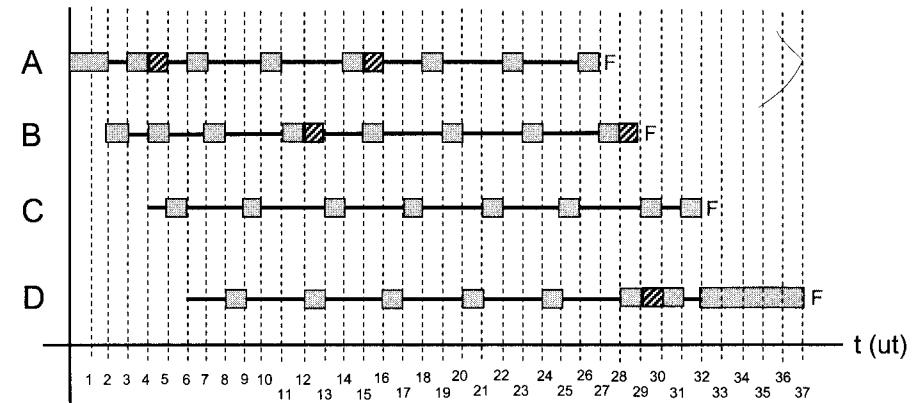


Figura 1-22: Evolución temporal de los procesos en el sistema.

En la gráfica anterior cabe destacar ciertos detalles en relación con los siguientes estados de tiempo:

- 3 ut. Sólo están los procesos A y B, por lo que al terminar el proceso B vuelve a entrar el A.
- 4 ut. B está en la cola antes de que llegase C, mientras que A aunque acaba en t=4 se coloca después del nuevo proceso C, tal como dice el enunciado del problema.
- 6 ut. A termina en el instante 4 y luego en la cola están B y C. B termina en el instante 5, así que se sitúa en la cola antes de D pero C que termina justo cuando llega el nuevo proceso D se sitúa después que éste.

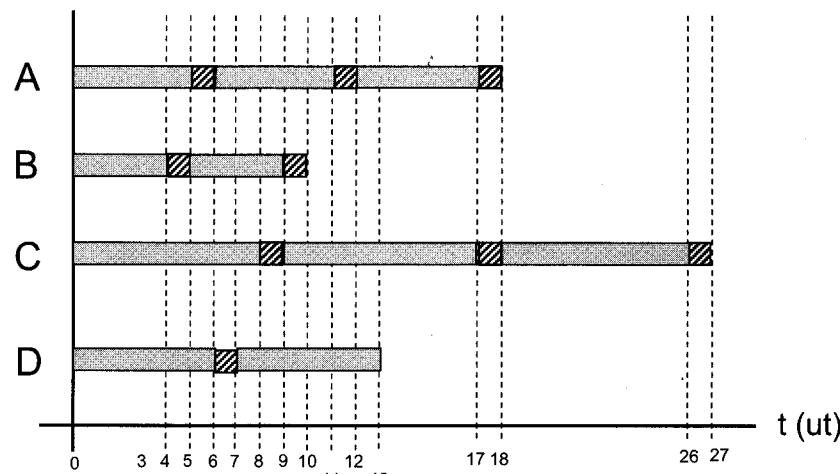
La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-14: Tiempos de retorno, respuesta y espera de los procesos.

	t. retorno	t. respuesta	t. espera
A	27	0	16
B	27	0	17
C	28	1	20
D	31	2	18

## EJERCICIO 8

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 2$  ut y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1-23. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 4 ut., el proceso C en 8 ut. y el proceso D en 10 ut.



■ Proceso x usando la CPU  
■ Operación de E/S

Figura 1-23. Esquema de ejecución de los procesos.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

## Solución

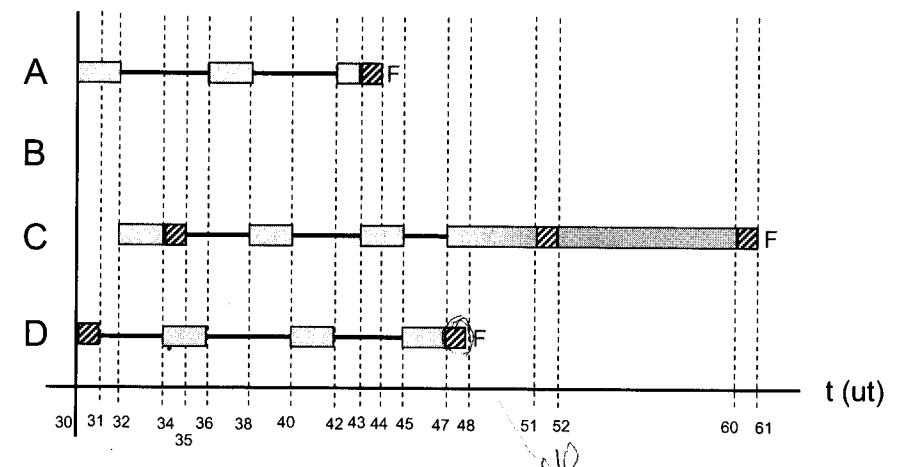
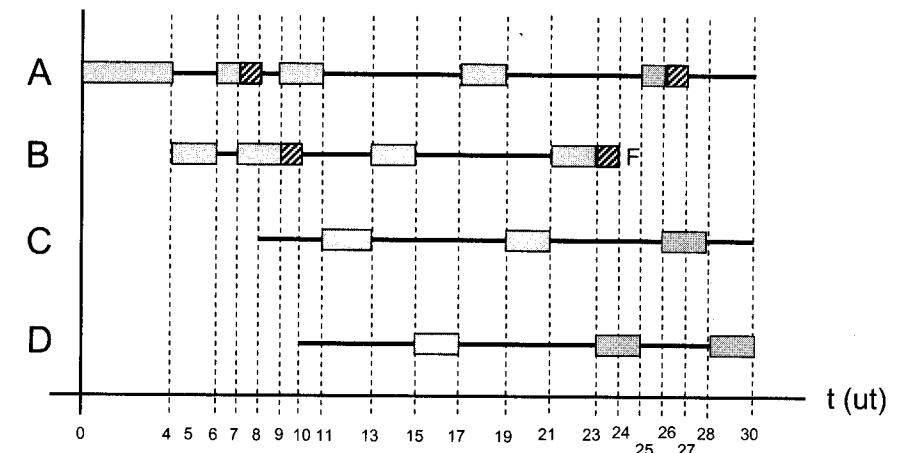


Figura 1-24: Evolución temporal de los procesos en el sistema.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-15: Tiempos de retorno, respuesta y espera de los procesos.

	t. retorno	t. respuesta	t. espera
A	44	0	26
B	20	0	10
C	53	3	26
D	38	5	35

En la gráfica cabe destacar ciertos detalles en relación con los siguientes estados de tiempo:

- 7 ut. Cuando el proceso A acaba su quantum y se vuelve a poner en la cola lo hace antes del proceso C que llega en el instante 8.
- 9 ut. B acaba su quantum antes de que llegue el nuevo proceso D que lo hace en el instante 20, así que se coloca en la cola antes.

#### EJERCICIO 9

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 2$  ut y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 1 ut., el proceso C en 2 ut. y el proceso D en 3 ut.

Además tenemos el proceso S que es un proceso del sistema con prioridad más alta que cualquiera de los otros procesos que llega al sistema en el instante 2 ut. y que se repite automáticamente cada 5 ut.

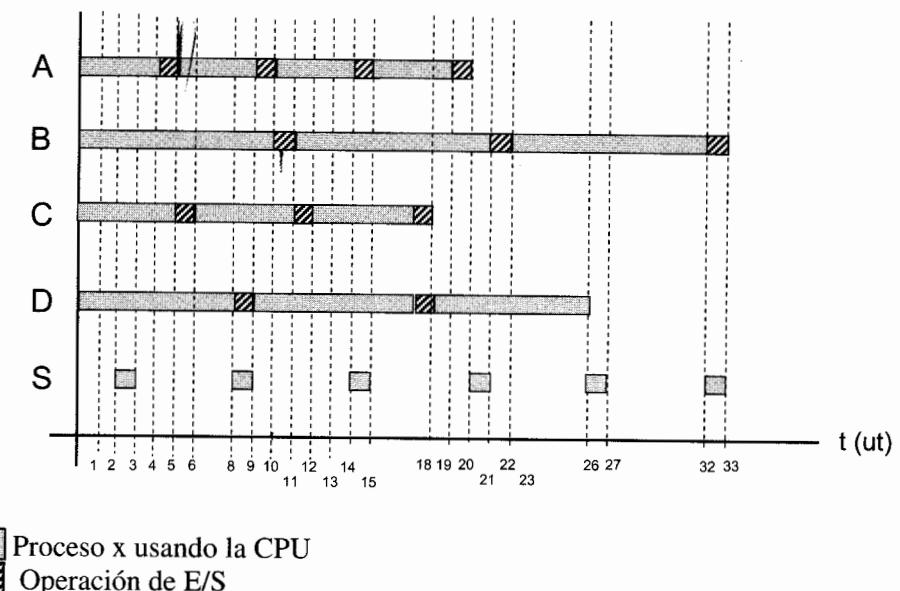
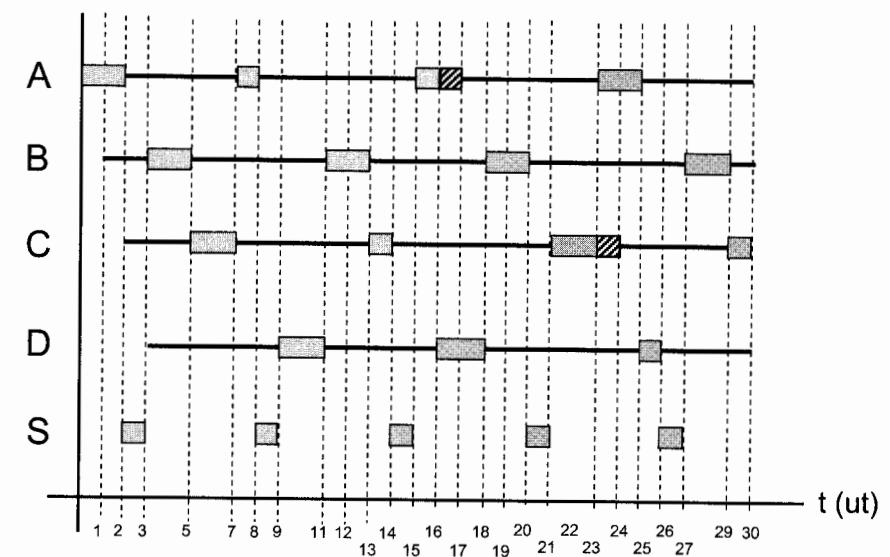


Figura 1-25. Esquema de ejecución de los procesos.

Solución:



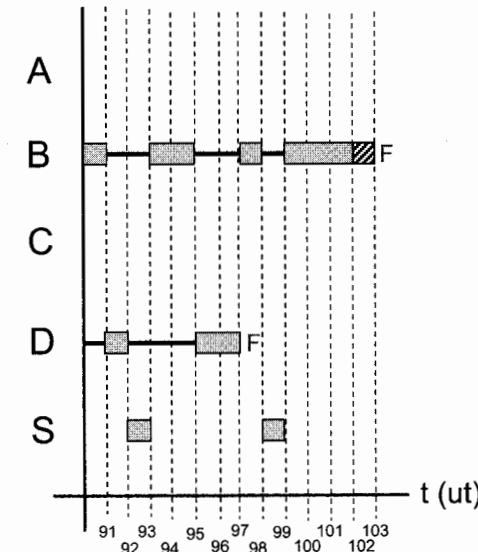
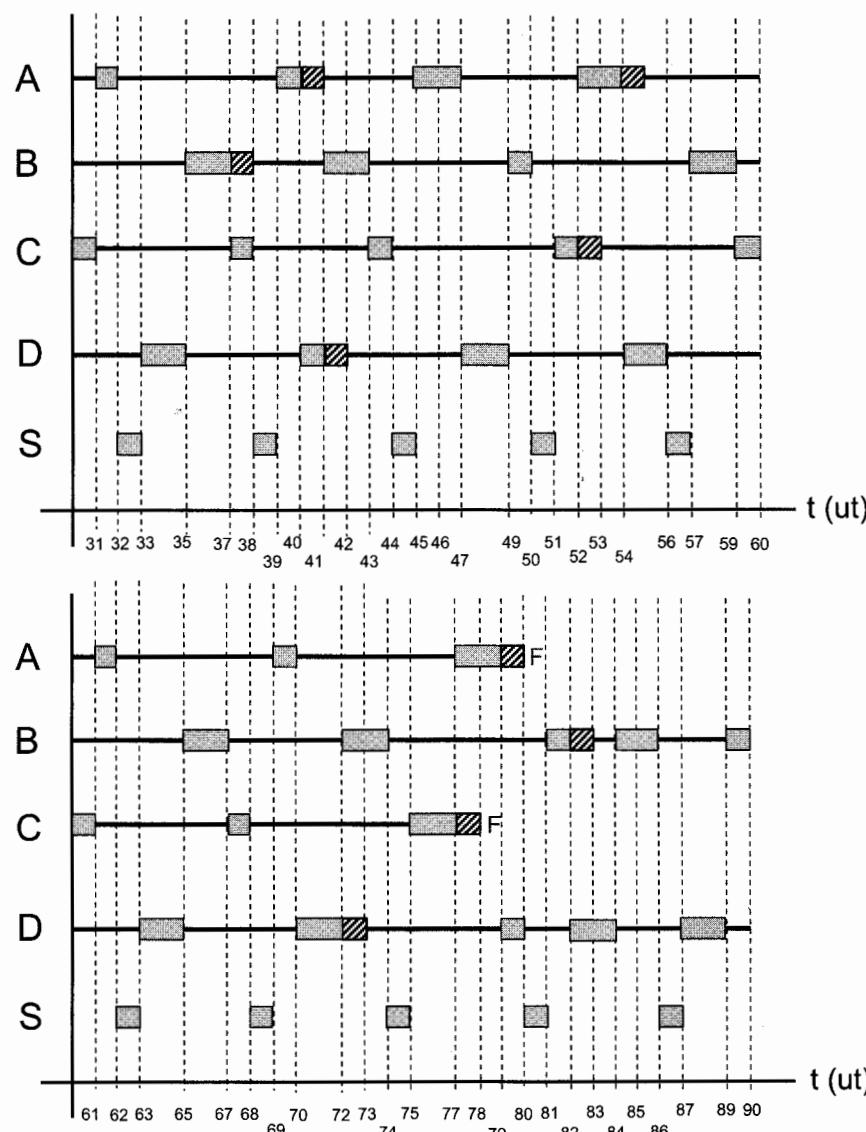


Figura 1-26: Evolución temporal de los procesos en el sistema.

#### EJERCICIO 10

Se dispone de un sistema monoprocesador con política de planificación del procesador SRT y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1. El proceso A llega al sistema en el instante 0 ut., el proceso B en el instante 1 ut., el proceso C en 2 ut. y el proceso D en 3 ut.

Además tenemos el proceso S que es un proceso del sistema con prioridad más alta que cualquiera de los otros procesos que llega al sistema en el instante 2 ut. y que se repite automáticamente cada 5 ut.

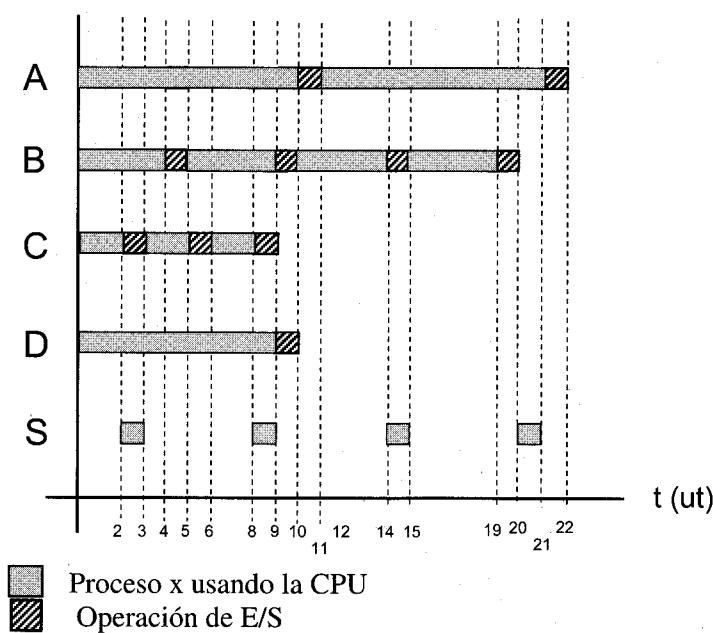


Figura 1-27. Esquema de ejecución de los procesos.

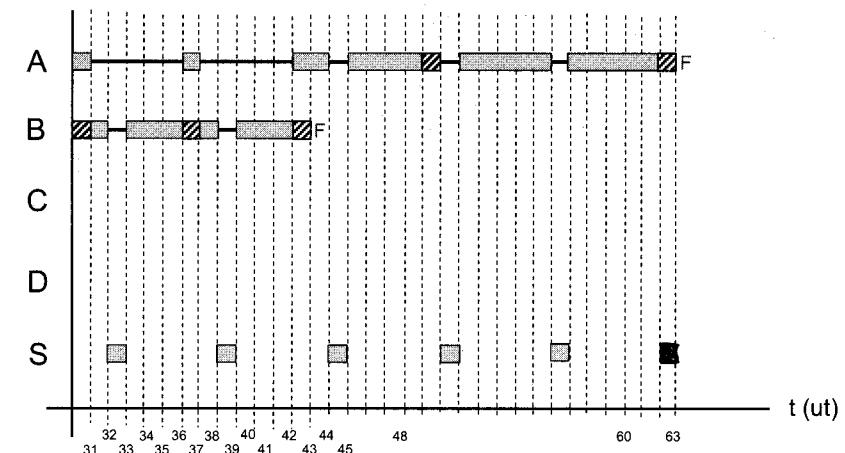
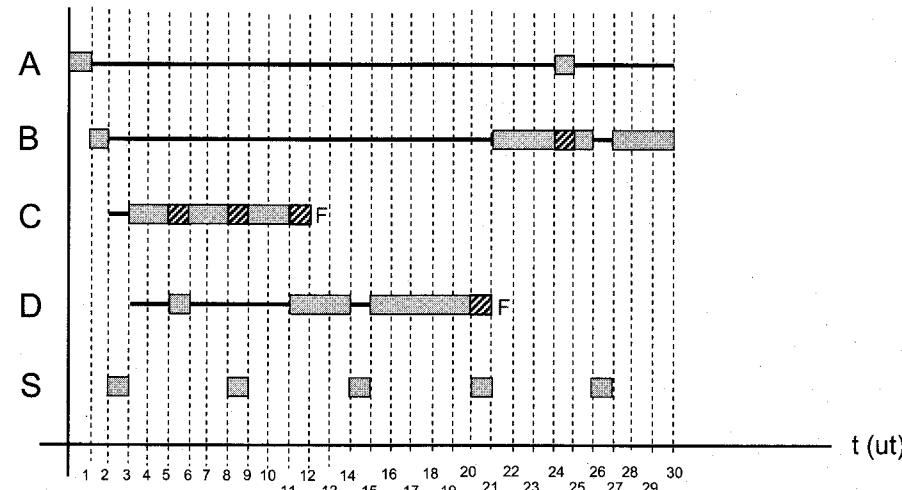
**Solución**

Figura 1-28: Evolución temporal de los procesos en el sistema.

**EJERCICIO 11**

En un SO de tiempo compartido se dispone de un planificador que emplea el algoritmo de planificación RR con quantum = 10 ms. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. Se ejecutan los procesos que se muestran en la figura siguiente:

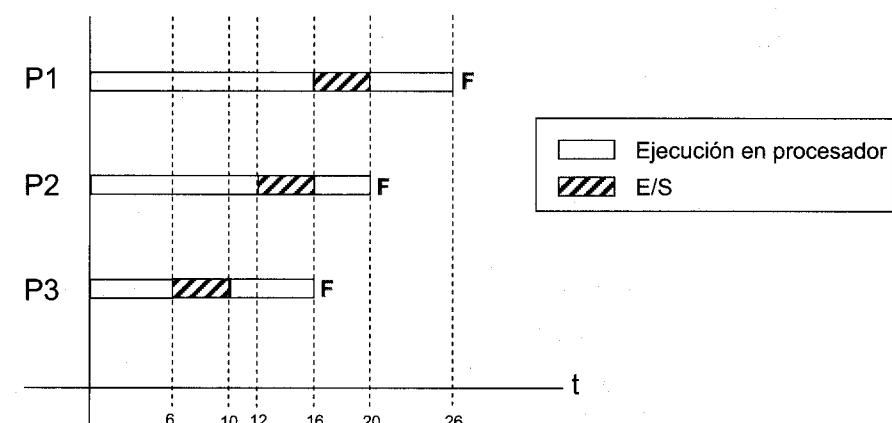


Figura 1-29: Esquema de ejecución de los procesos.

Las operaciones de E/S se realizan sobre un único dispositivo. En la siguiente tabla se indica el instante de llegada de cada proceso.

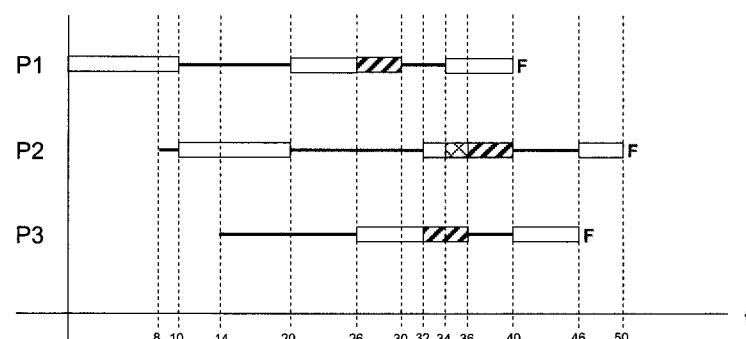
Tabla 1-16. Instante de llegada de los procesos.

Proceso	Tiempo de llegada
1	0
2	8
3	14

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y del dispositivo de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

Solución:

La siguiente figura muestra el cronograma de la ejecución de los procesos:



donde

- Ejecución en procesador
- ▨ E/S
- ▢▢▢ Proceso en estado de espera
- Proceso en estado de preparado

Figura 1-30: Evolución temporal de los procesos en el sistema.

La tabla siguiente muestra el cálculo de los tiempos de respuesta, espera y retorno de los procesos.

Tabla 1-17: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t. espera
1	0	40	14
2	2	42	18+2
3	12	34	16

## EJERCICIO 12

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 2$  ut y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. El instante de llegada de los procesos se indica en la tabla. Suponer que el acceso a los dispositivos es en exclusión mutua.

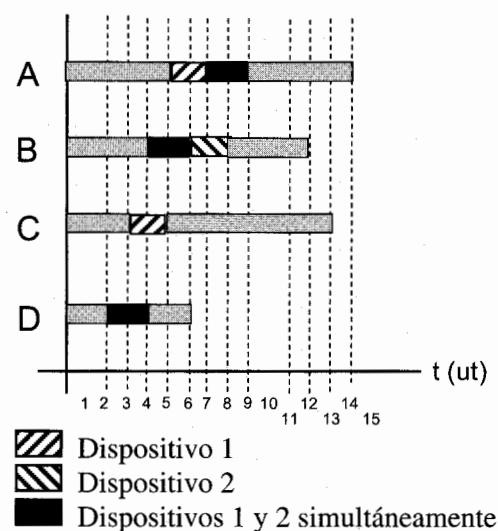


Figura 1-31: Esquema de ejecución de los procesos.

Tabla 1-18. Instante de llegada de los procesos.

A	0
B	4
C	8
D	10

Mostrar la evolución temporal de los procesos del sistema.

### Solución

Ejercicio similar a los que se han puesto con suficiente frecuencia en los exámenes, cuatro procesos con requerimientos tanto de ejecución como de varios dispositivos, e incluso en ocasiones necesidades de utilizar dos dispositivos simultáneamente. Además teniendo en cuenta que el acceso a los dispositivos se debe realizar en exclusión mutua.

Inicialmente sólo está en el sistema el proceso A (llega en el instante 0) que consume dos quants de tiempo. En el instante 4, llega el proceso B y finaliza el segundo quantum del proceso A, obtiene la CPU el proceso B porque en la definición del problema se dice que si el quantum de un proceso finaliza en el mismo instante que la llegada de un proceso nuevo, el nuevo proceso irá delante en la cola de preparados.

En el instante 7, el proceso A solicita el dispositivo 1 y lo obtiene directamente ya que no está siendo utilizado por nadie. Mientras el proceso B estará en ejecución.

En el instante 9, el proceso B (que está en ejecución) solicita utilizar simultáneamente los dispositivos 1 y 2. En ese mismo instante el proceso A, que termina de utilizar el dispositivo 1, también necesita los dos dispositivos al mismo tiempo. Será el proceso B el que obtendrá los dispositivos y los podrá utilizar, quedando el proceso A en la cola de los dispositivos esperando que sean liberados. Se le conceden a B por ser el proceso que está en ejecución, en la realidad para solicitar recursos el proceso tiene que estar en ejecución, por tanto, el proceso A en la realidad volvería a preparados, esperaría a entrar en ejecución y una vez en ejecución la primera instrucción que haría sería solicitar los dos dispositivos. En los ejercicios consideramos el proceso descrito para A despreciable, y por ello asumimos que después de liberar un dispositivo, solicita los dos directamente. De igual manera, el cambio de contexto en la realidad consume tiempo, y en los ejercicios (salvo que se diga lo contrario) se considera despreciable, por lo que no se representa.

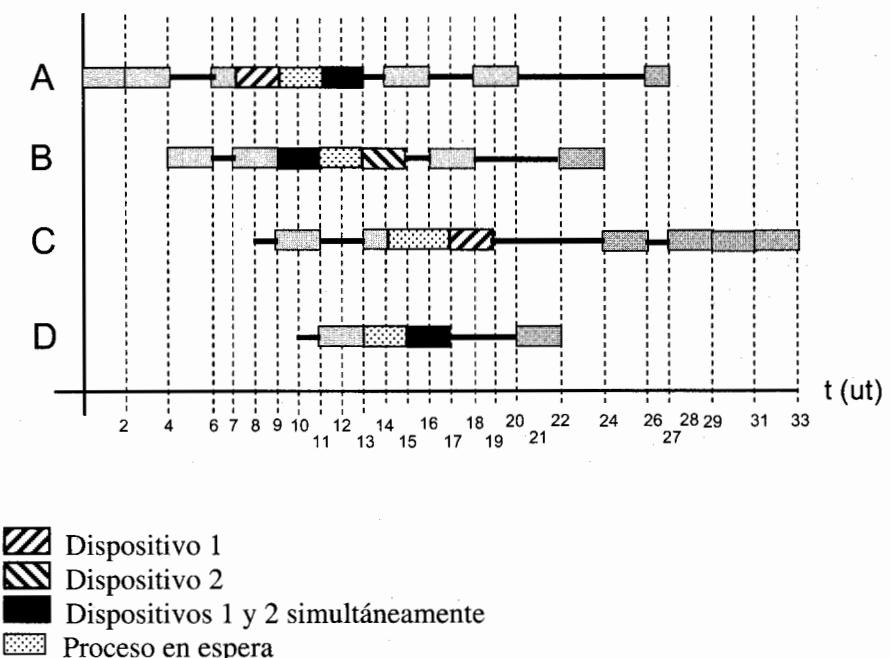


Figura 1-32: Evolución temporal de los procesos en el sistema.

### EJERCICIO 13

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 2$  ut y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos que llegan al sistema sigue el esquema descrito en la figura. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un proceso a la cola de preparados (independientemente de si el proceso es nuevo o estaba realizando una operación de E/S), entonces el proceso que llega se añade a la cola de procesos preparados antes que el proceso que termina su quantum.

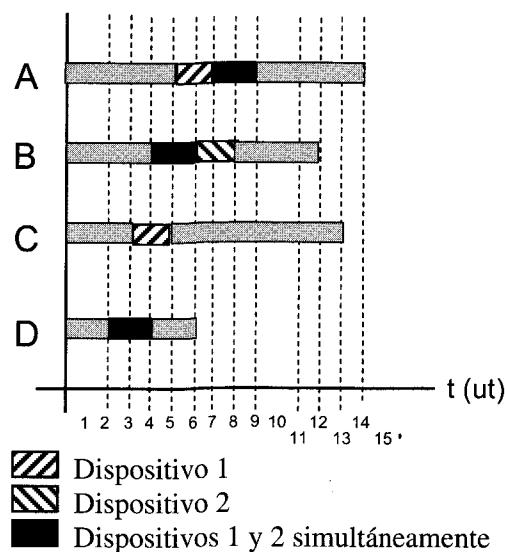


Figura 1-33: Esquema de ejecución de los procesos.

El instante de llegada de los procesos se indica en la tabla. Suponer que se puede acceder a los dispositivos de forma concurrente, sin necesidad de exclusión mutua.

Tabla 1-19. Instante de llegada de los procesos.

A	0
B	4
C	8
D	10

Mostrar la evolución temporal de los procesos del sistema.

### Solución

Como podemos observar, este problema es el mismo que el anterior pero con la diferencia de que a los dispositivos no es necesario que se acceda en exclusión mutua, y por tanto se puede acceder concurrentemente, podremos tener varios procesos accediendo simultáneamente a los mismos dispositivos.

Se puede observar que es idéntico al ejercicio anterior hasta el instante 9, simplemente recordar que en el instante 4, cuando llega el proceso B y finaliza el segundo quantum del proceso A, se le concede la CPU al proceso B porque en la definición del problema se dice que si el quantum de un proceso finaliza en el mismo instante que la llegada de un proceso nuevo, el nuevo proceso irá delante en la cola de preparados.

Analicemos el cronograma entre los instantes 9 y 11, los procesos A y B han solicitado realizar operaciones en los dispositivos 1 y 2 simultáneamente. Los dos procesos están utilizando los dos dispositivos al mismo tiempo, esto es posible porque según el enunciado no se necesita exclusión mutua para acceder a los dispositivos.

Finalmente, indicar que en los instantes 11, 15 y 18 se ha producido la siguiente situación: un proceso termina su quantum de CPU y otro proceso termina una operación en los dispositivos, en este caso el proceso que termina la operación en los dispositivos se ha añadido antes en la cola de preparados.

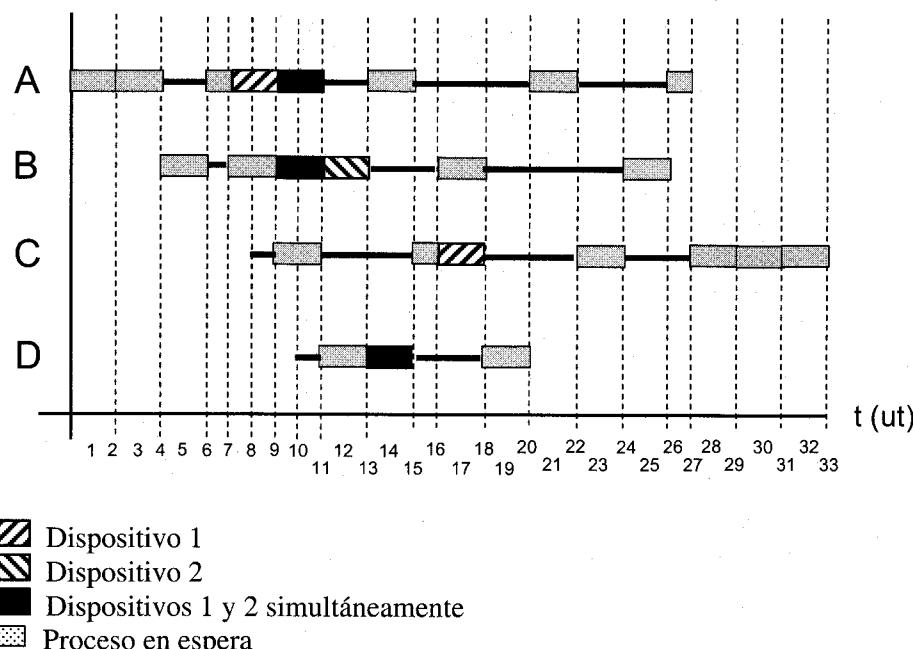


Figura 1-34: Evolución temporal de los procesos en el sistema.

#### EJERCICIO 14

En un sistema operativo de tiempo compartido se dispone de un planificador que emplea el algoritmo de planificación RR con quantum = 10 ms. Si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina. Se ejecutan los procesos que muestra la figura 1-35.

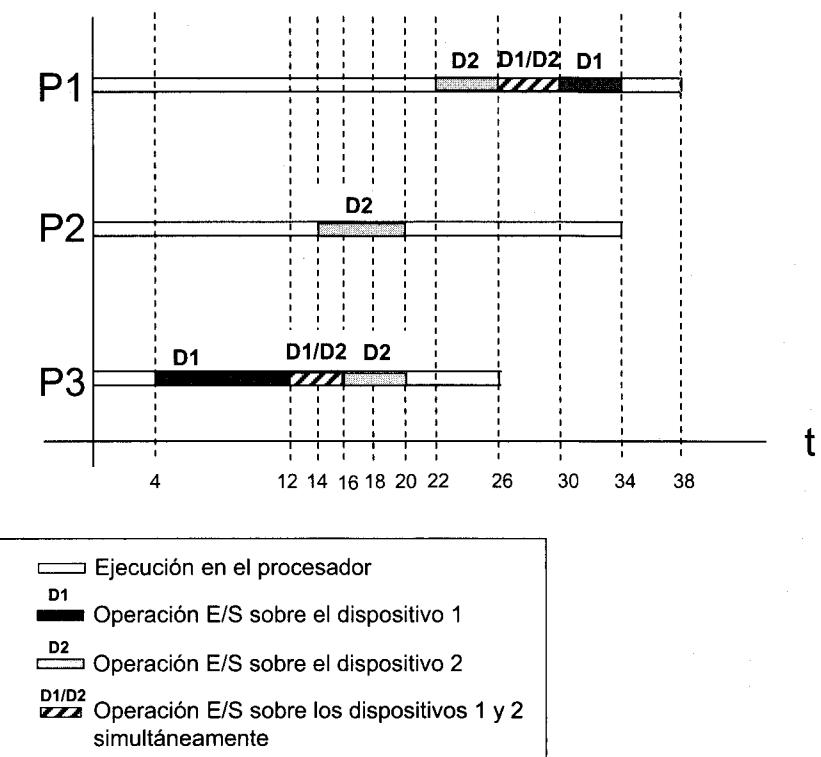


Figura 1-35: Evolución temporal de los procesos.

Las operaciones de E/S se realizan sobre los dispositivos D1 y D2 de acceso en exclusión mutua. La gestión de la cola de acceso a los dispositivos sigue una planificación según el orden de llegada. El Sistema Operativo no permite la expropiación de recursos a procesos.

El instante de llegada de cada proceso se muestra en la tabla siguiente:

Tabla 1-20: Instante de llegada de los procesos.

Proceso	Tiempo de llegada
1	0
2	8
3	18

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta,

de retorno y de espera para cada uno de los procesos. Resolver las situaciones de planificación en el acceso a los recursos y justificar las decisiones al respecto.

#### Solución 1:

Cuando un proceso tiene un recurso y necesita otro lo solicita sin liberar el que ya tiene.

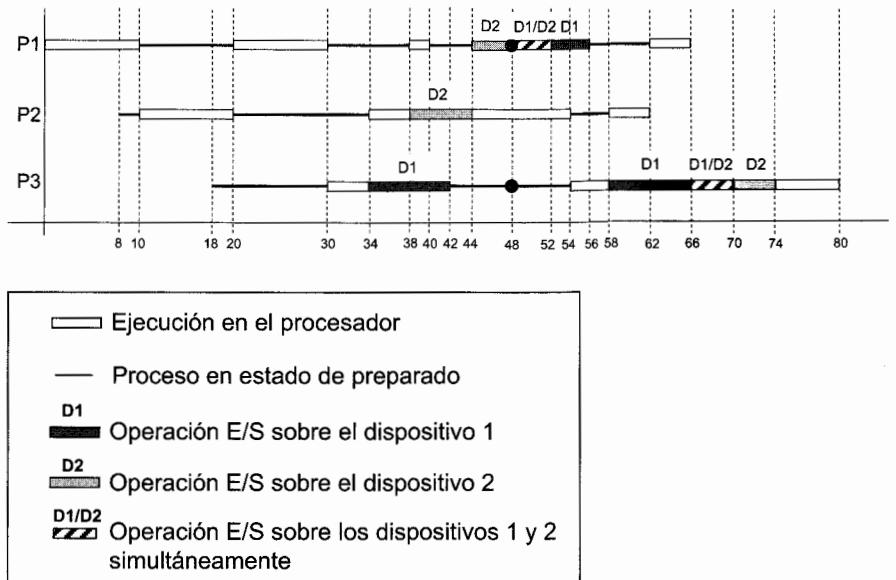


Figura 1-36: Evolución temporal de los procesos en el sistema según la solución 1.

En el instante 48 se produce un interbloqueo entre los procesos p1 y p3. La decisión que se toma consiste en reiniciar el proceso p3.

Los tiempos a calcular para esta solución se muestran en la tabla 2:

Tabla 1-21: Tiempos de retorno, respuesta y espera de los procesos según la solución 1.

	t. retorno	t. respuesta	t. espera
p1	66	0	28
p2	54	2	20
p3	62	12	24

#### Solución 2:

En este caso, se considera que cuando un proceso termina de utilizar un dispositivo, lo abandona antes de solicitar otros, aunque entre esos otros esté el que ya tenía.

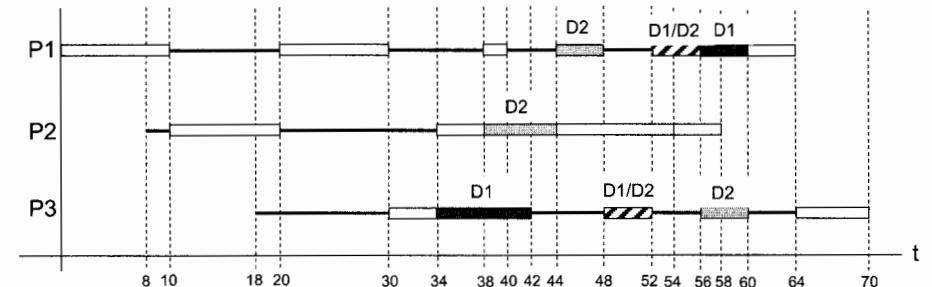


Figura 1-37: Evolución temporal de los procesos en el sistema según la solución 2.

Los tiempos a calcular para esta solución se muestran en la tabla 3:

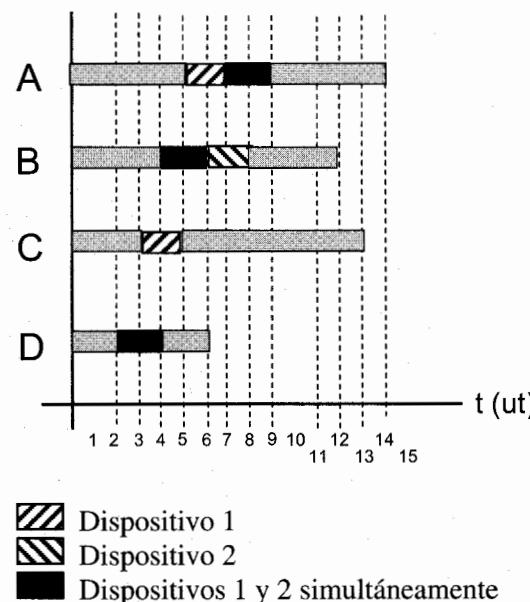
Tabla 1-22: Tiempos de retorno, respuesta y espera de los procesos según la solución 2.

	t. retorno	t. respuesta	t. espera
p1	64	0	26
p2	50	2	16
p3	52	12	26

#### EJERCICIO 15

Se dispone de un sistema monoprocesador con política de planificación del procesador MLQ con dos colas, la cola 1 es más prioritaria que la 2 y es expropiativo entre colas. La gestión de la cola 1 es RR con  $q = 2$  ut, mientras que la gestión de la cola 2 es FCFS. Además, existe gestión de los

dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura. Si el quantum de un proceso en ejecución expira a la vez que la llegada de otro a la cola de preparados (nuevo o desde operación de E/S), entonces el proceso que llega se añade antes que el proceso que termina.



El instante de llegada de los procesos y la cola a la que pertenecen se indica en la tabla. Suponer que el acceso a los dispositivos es en exclusión mutua.

Tabla 1-23. Instante de llegada de los procesos.

Proceso	T <sup>a</sup> ini	Cola
A	0	1
B	4	2
C	8	1
D	10	2

Mostrar la evolución temporal de los procesos del sistema.

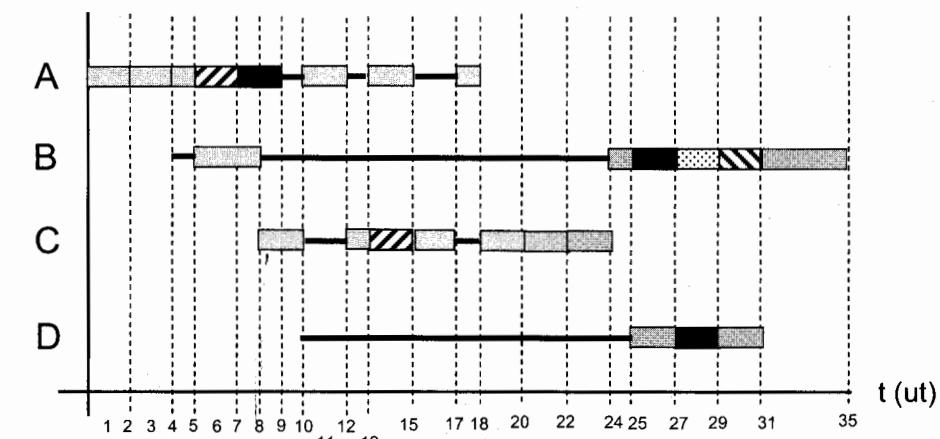
### Solución

Gestión multicolas sin realimentación en el que tenemos dos colas con distinta prioridad, y donde existe expropiatividad entre colas, los procesos tienen requerimientos de acceso a los dispositivos y el acceso se debe realizar en exclusión mutua.

En el instante 8, podemos observar que está en ejecución el proceso B (cola 2) y llega el proceso C (cola 1) que pertenece a la cola más prioritaria, y como el algoritmo es expropiativo, se le quita la CPU al proceso B y entra directamente a ejecución el proceso C.

En el instante 15, el proceso A termina un quantum de ejecución, mientras que el proceso C termina una operación en el dispositivo 1, obtiene la CPU el proceso C por la restricción de la definición del problema, el proceso que llega se pone antes en la cola de preparados, por lo que será el que se seleccione para ejecución.

Finalmente, indicar que en el instante 27, el proceso D está en ejecución y solicita una operación sobre los dispositivos 1 y 2, mientras el proceso B termina una operación sobre los dispositivos y solicita otra. Se le conceden los dispositivos al proceso D por ser el proceso que estaba en ejecución en ese momento (el lector pueden encontrar la explicación en un problema anterior).



### EJERCICIO 16

Se dispone de un sistema monoprocesador con política de planificación MLFQ.

La configuración de las colas es la siguiente:

Cola 0: Planificación FCFS.

Cola 1: Planificación RR con quantum = 100 ut.

Cola 2: Planificación RR con quantum = 200 ut.

Cola 3: Planificación RR con quantum = 300 ut.

La prioridad de los procesos es decreciente con el número de cola.

La ejecución de los procesos sigue el esquema que muestra la tabla siguiente:

Tabla 1-24: Esquema de ejecución de los procesos en el sistema.

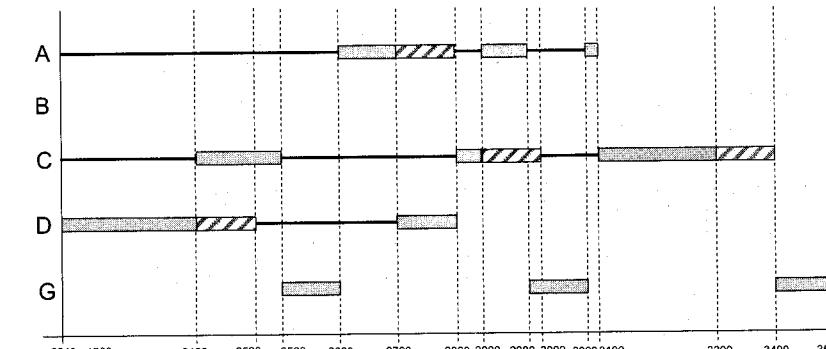
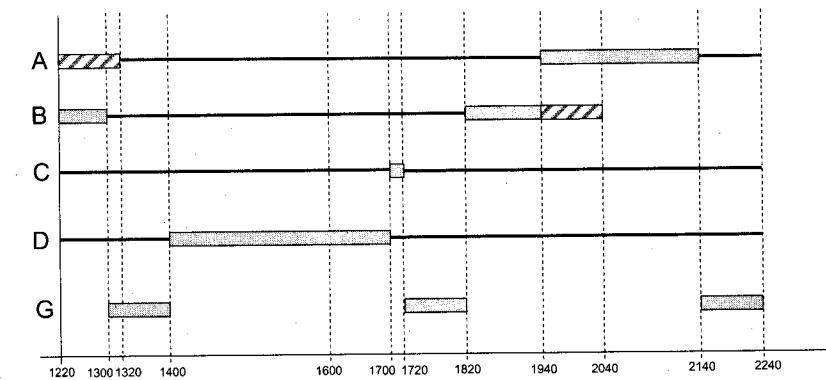
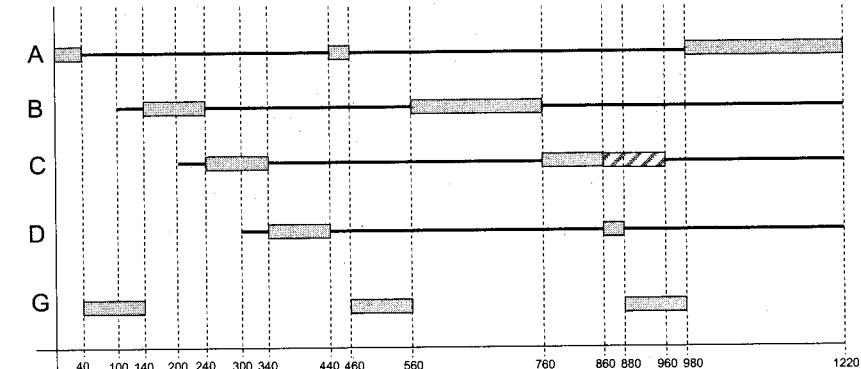
Instante de llegada	Tipo	Duración
0 ut.	A de usuario	700 ut. cada 300 ut. hace E/S de 100 ut.
100 ut.	B de usuario	500 ut. cada 500 ut. hace E/S de 100 ut.
200 ut.	C de usuario	600 ut. cada 200 ut. hace E/S de 100 ut.
300 ut.	D de usuario	700 ut. cada 600 ut. hace E/S de 100 ut.
40 ut.	G de sistema	100 ut..
<b>Este proceso se invoca automáticamente cada 320 ut. después de finalizar su ejecución.</b>		

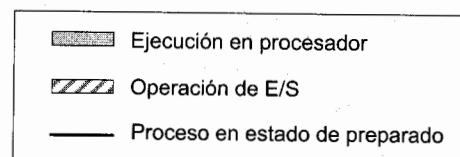
Los procesos de sistema entran directamente a la cola 0. Los procesos de usuario entran a la cola 1. Cuando abandonan el procesador pasan a la cola siguiente de menor prioridad.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

Solución:

La evolución temporal de los procesos se muestra en la figura 1 siguiente:



*Figura 1-40: Evolución temporal de los procesos.*

El cálculo de los tiempos de respuesta, retorno y espera se muestra en la tabla 2:

*Tabla 1-25: Tiempos de retorno, respuesta y espera de los procesos.*

Proceso	t.respuesta	t.retorno	t.espera
A	0	3100	2200
B	40	1940	1340
C	40	3200	2300
D	40	2560	1760
G	--	--	--

## EJERCICIO 17

Se dispone de un sistema monoprocesador con política de planificación MLFQ. La prioridad de las colas es expropiativa y decreciente con el número de cola. La configuración de las colas es la siguiente:

Cola 0: Planificación FCFS

Cola 1: Planificación RR con quantum = 100 ut.

Cola 2: Planificación SRT

Cola 3: Planificación RR con quantum = 200 ut.

La ejecución de los procesos sigue el esquema que muestra la tabla siguiente:

*Tabla 1-26: Esquema de ejecución de los procesos en el sistema.*

Instante de llegada	Tipo	Duración
0 ut.	A de usuario	700 ut. cada 300 ut. hace E/S de 100 ut.
100 ut.	B de usuario	500 ut. cada 500 ut. hace E/S de 100 ut.
200 ut.	C de usuario	600 ut. cada 200 ut. hace E/S de 100 ut.
300 ut.	D de usuario	800 ut. cada 200 ut. hace E/S de 100 ut.
40 ut.	G de sistema <b>Este proceso se invoca automáticamente cada 320 ut. después de finalizar su ejecución.</b>	100 ut..

Los procesos de sistema se ejecutan en la cola 0. Los procesos de usuario se envían a una de las colas según la siguiente expresión:

$$\text{cola} = (\text{tiempo\_ejecución\_proceso DIV } 300) + 1$$

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

**Solución**

La evolución temporal de los procesos se muestra en la figura siguiente:

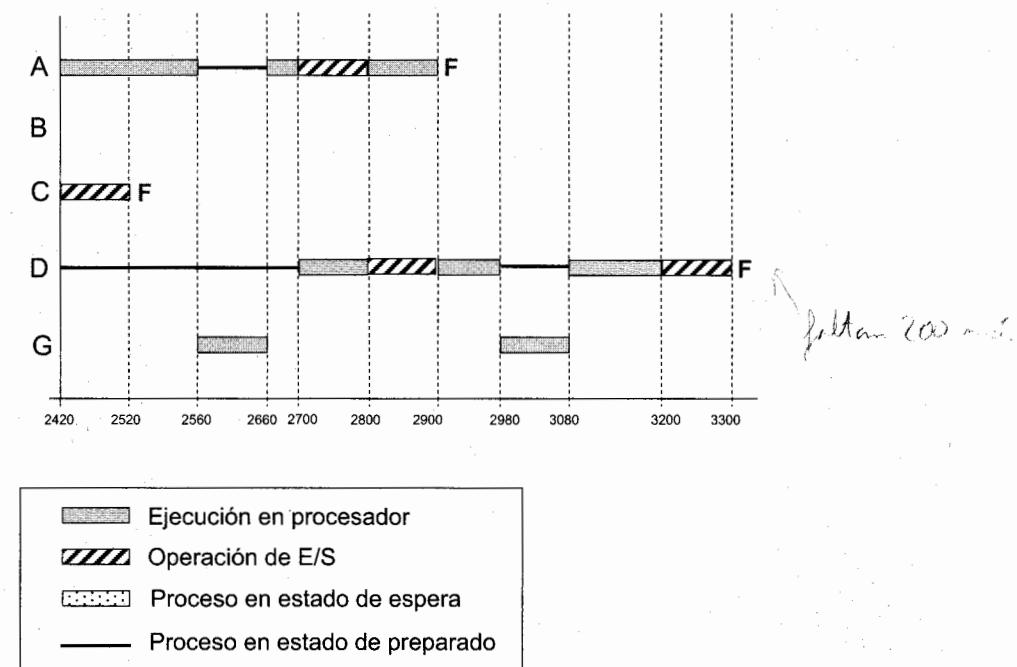
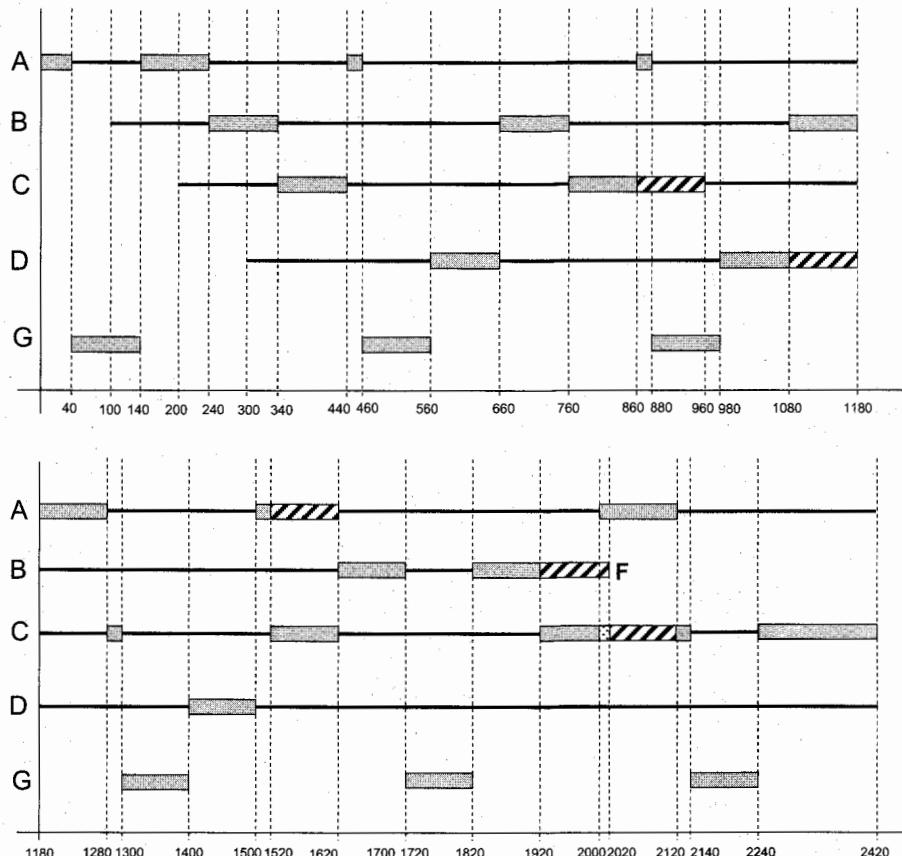


Figura 1-41: Evolución temporal de los procesos.

El cálculo de los tiempos de respuesta, retorno y espera se muestra en la tabla siguiente:

Tabla 1-27: Tiempos de retorno, respuesta y espera de los procesos.

Proceso	t.respuesta.	t.retorno	t.espera
A	0	2900	2000
B	140	1920	1320
C	140	2320	1400+20
D	160	3000	1800
G	0	--	0

### EJERCICIO 18

Se dispone de un sistema monoprocesador con política de planificación MLQ. La prioridad de las colas es expropiativa y decreciente con el número de cola.

La configuración de las colas es la siguiente:

Cola 0: Planificación FCFS

Cola 1: Planificación RR con quantum = 100 ut.

Cola 2: Planificación SRT

La llegada de los procesos al sistema sigue el siguiente esquema:

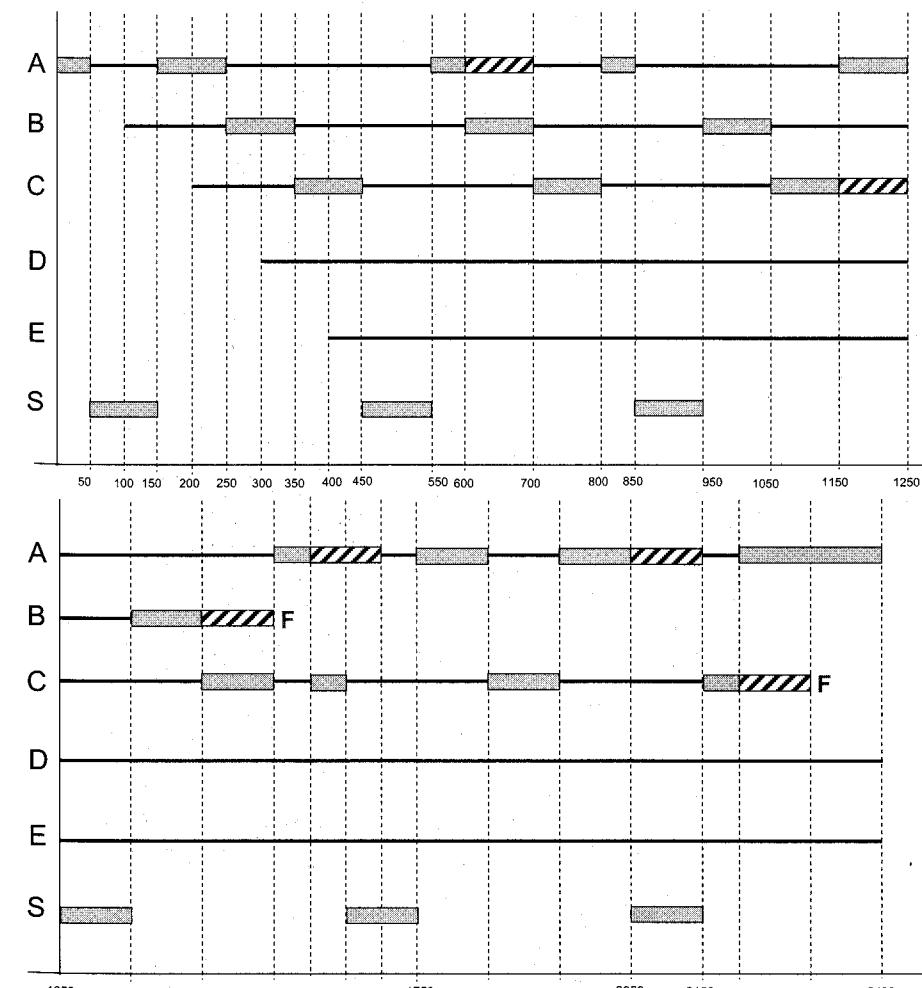
Tabla 1-28: Esquema de ejecución de los procesos en el sistema.

Instante de llegada	Tipo	Duración
0 ut.	A de usuario	800 ut. cada 200 ut. hace E/S de 100 ut.
100 ut.	B de usuario	400 ut. cada 400 ut. hace E/S de 100 ut.
200 ut.	C de usuario	600 ut. cada 300 ut. hace E/S de 100 ut.
300 ut.	D de usuario	1000 ut. cada 500 ut. hace E/S de 100 ut.
400 ut.	E de usuario	800 ut. cada 400 ut. hace E/S de 100 ut.
50 ut.	S de sistema	100 ut..
<b>Este proceso se invoca automáticamente cada 300 ut. después de finalizar su ejecución.</b>		

Los procesos de sistema se ejecutan en la cola 0. Los procesos de usuario A, B y C se ejecutan en la cola 1 (si el quantum de un proceso en ejecución expira a la vez que la llegada de un nuevo proceso, entonces el nuevo proceso se añade a la cola de procesos en espera de ejecutarse antes que el proceso que termina) mientras que los procesos D y E en la cola 2.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

### Solución



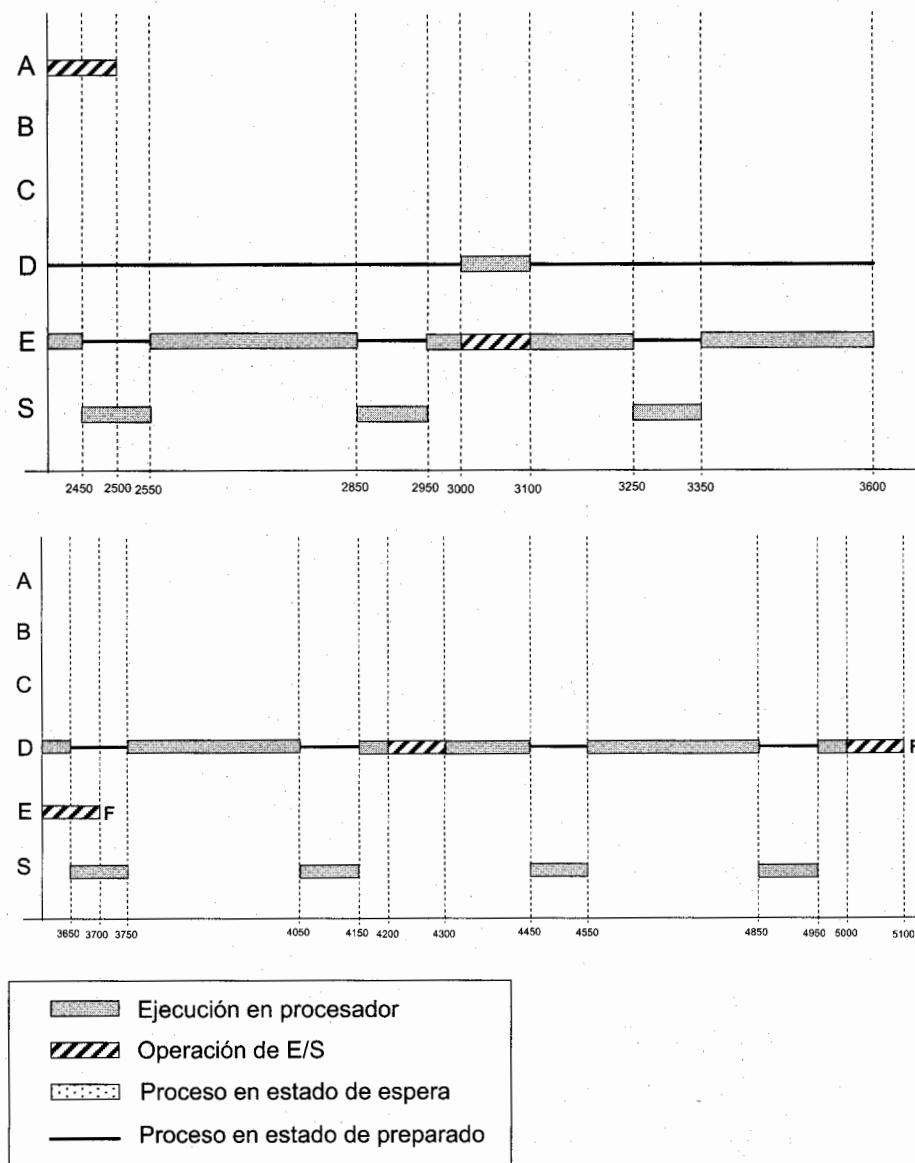


Figura 1-42: Evolución temporal de los procesos.

El cálculo de los tiempos de respuesta, retorno y espera se muestra en la tabla siguiente:

Tabla 1-29: Tiempos de retorno, respuesta y espera de los procesos.

Proceso	t.respuesta	t.retorno	t.espera
A	0	2500	1300
B	150	1450	950
C	150	2100	1300
D	2700	4800	3900
E	2000	3300	2300
S	0	--	0

## EJERCICIO 19

Se dispone de un sistema monoprocesador con política de planificación MLFQ. La prioridad de las colas es expropiativa y decreciente con el número de cola.

La configuración de las colas es la siguiente:

Cola 0: Planificación FCFS

Cola 1: Planificación RR con quantum = 100 ut.

Cola 2: Planificación RR con quantum = 200 ut.

Cola 3: Planificación SRT

La llegada de los procesos al sistema sigue el siguiente esquema:

Tabla 1-30: Esquema de ejecución de los procesos en el sistema.

Instante de llegada	Tipo	Duración
0 ut.	A de usuario	800 ut. cada 200 ut. hace E/S de 100 ut.
100 ut.	B de usuario	400 ut. cada 400 ut. hace E/S de 100 ut.
200 ut.	C de usuario	600 ut. cada 300 ut. hace E/S de 100 ut.
300 ut.	D de usuario	1000 ut. cada 500 ut. hace E/S de 100 ut.
50 ut.	S de sistema	100 ut..

**Este proceso se invoca automáticamente cada 300 ut. después de finalizar su ejecución.**

Los procesos de sistema se ejecutan en la cola 0. Los procesos de usuario entran a la cola 1. Cada vez que han ejecutado la tercera parte de su tiempo de ejecución total abandonan el procesador y pasan a la cola siguiente de menor prioridad.

Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

#### Solución

Se indica en la gráfica cuando los procesos abandonan cada una de las colas, indicando en rojo a la cola a la que se dirigen. Hay que tener en cuenta que los quantum de las colas 1 y 2 son diferentes y el proceso del sistema tiene prioridad y es expropiativo no dejando acabar el quantum del proceso en curso.

Al llegar a la cola 3 los procesos llegan en el orden A, C y D pero esto no se tiene en cuenta sino el tiempo que les queda por finalizar que es 200, 100 y 200 respectivamente por lo que se ejecutarán en el orden C, y a continuación puede ser cualquiera de los otros 2 procesos, vamos a seleccionar A por llevar más tiempo en la cola y por ser un proceso más corto que D.

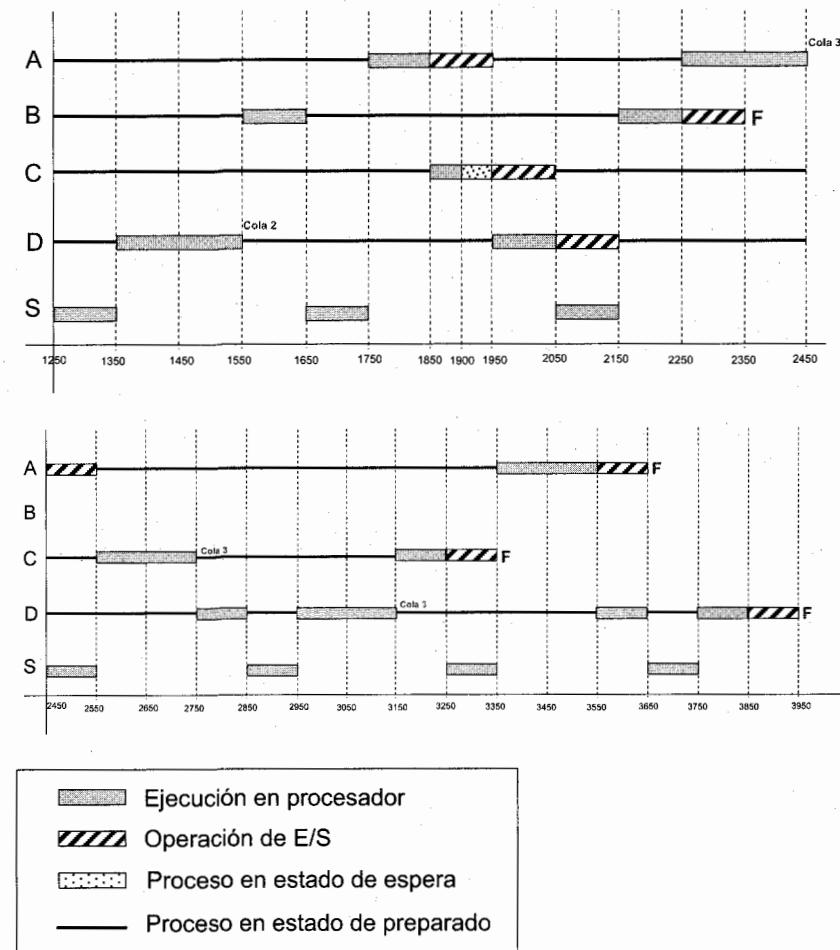
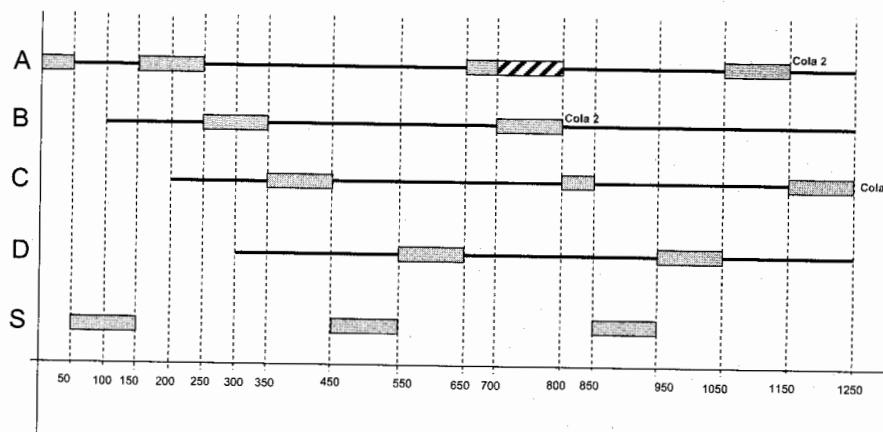


Figura 1-43: Evolución temporal de los procesos.

El cálculo de los tiempos de respuesta, retorno y espera se muestra en la tabla siguiente:

Tabla 1-31: Tiempos de retorno, respuesta y espera de los procesos.

Proceso	t.respuesta	t.retorno	t.espera
A	0	3650	2450
B	150	2250	1750
C	150	3150	2300+50
D	250	3650	2450

**EJERCICIO 20**

Se dispone de un sistema con dos procesadores: el primer procesador tiene política de planificación SRT y ejecuta cualquier tipo de proceso; el segundo procesador tiene política FCFS y sólo ejecuta procesos de sistema. Se dispone de un recurso de E/S con gestión FCFS. El sistema recibe para su ejecución procesos de usuario y de sistema. La ejecución de los procesos de usuario sigue el esquema descrito en la figura.

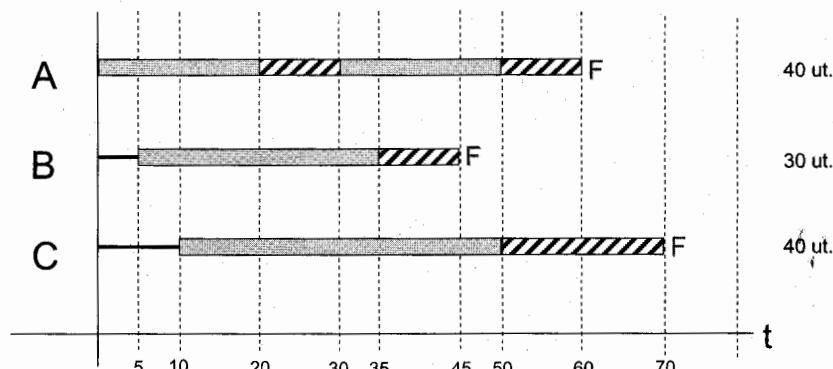


Figura 1-44: Esquema de ejecución de los procesos.

La ejecución de los procesos de sistema sigue el esquema que se muestra en la tabla siguiente. Los tiempos de ejecución periódica de estos procesos se miden en relación al reloj del sistema sin tener en consideración el instante en que terminaron.

Tabla 1-32: Esquema de ejecución de los procesos de sistema.

PS 1	Ejecución en CPU de duración 5 u.t.	Se ejecuta periódicamente cada 10 u.t.
PS 2	Ejecución en CPU de duración 5 u.t.	Se ejecuta periódicamente cada 20 u.t.
PS 3	Realiza E/S de duración 5 u.t.	Se ejecuta periódicamente cada 15 u.t.

El instante de llegada de los procesos de usuario se muestra en la tabla siguiente:

Tabla 1-33: Instantes de llegada de los procesos de usuario.

A	0 u.t.
B	5 u.t.
C	10 u.t.

Los procesos de sistema tienen prioridad expropiativa sobre los procesos de usuario, salvo que haya algún procesador libre. No es posible expropiar recursos a procesos. Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y del dispositivo de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos. Considera un tiempo de cambio de contexto despreciable.

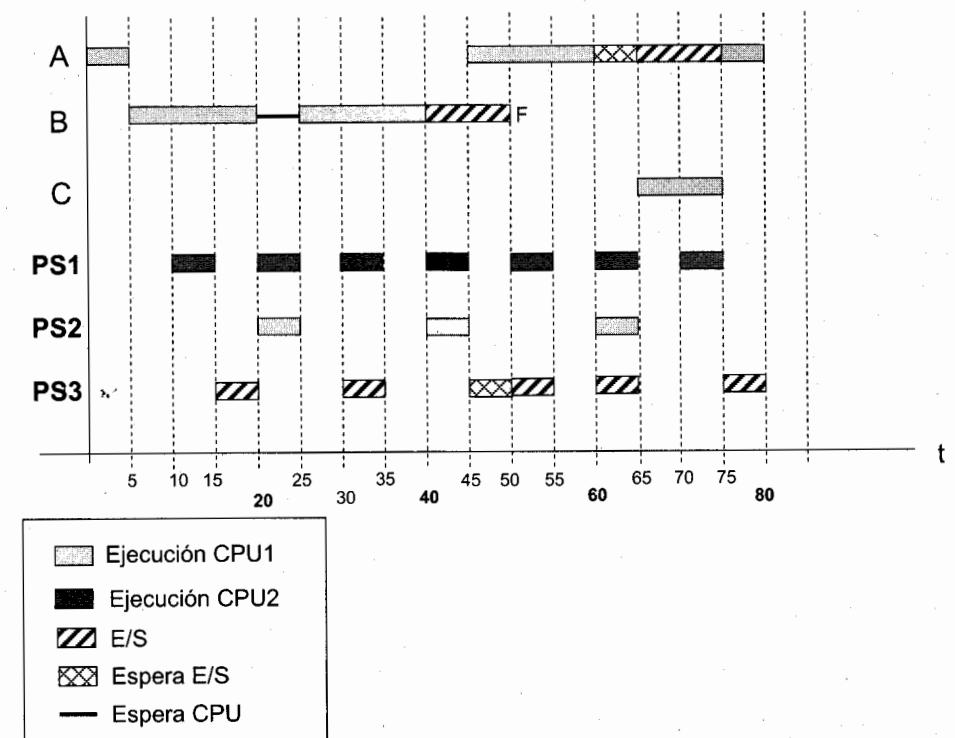


Figura 1-45: Evolución temporal de los procesos.

En el instante  $t=10$  los tiempos restantes de B expropia al proceso A de la CPU 1, y puede continuar con esta CPU hasta que llega el otro proceso del sistema que, con mayor prioridad, expropia al proceso A.

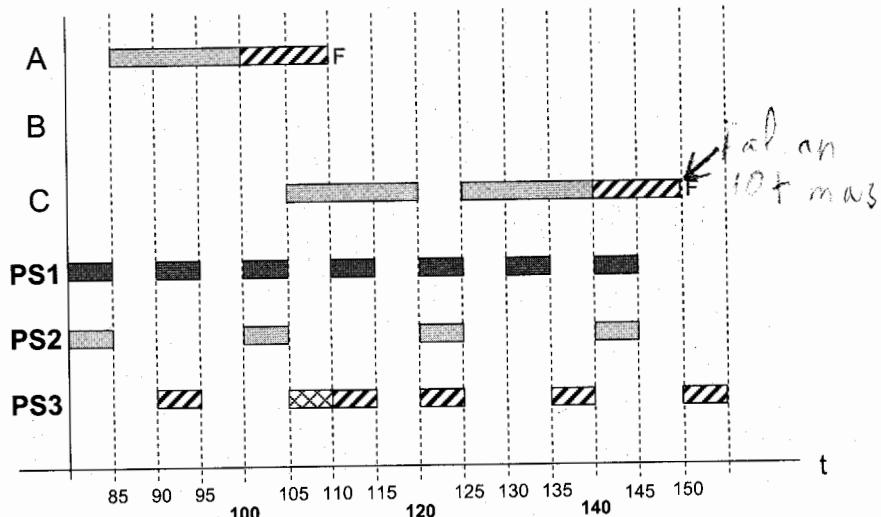


Figura 1-46: Evolución temporal de los procesos (continuación).

El cálculo de los tiempos de respuesta, retorno y espera se muestra en la tabla siguiente:

Tabla 1-31: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t.espera
A	0	110	45+5
B	0	50	5
C	55	150	90

Utilización del procesador:

Procesador 1: 100%

Procesador 2:  $45/100 = 45\%$

### EJERCICIO 21

Se dispone de un sistema monoprocesador cuyo algoritmo de planificación es RR con quantum=q u.t. Se lanzan simultáneamente al sistema n procesos, donde cada proceso  $p_i$  tiene una duración de  $T_i$  u.t. El sistema emplea un tiempo de cambio de contexto fijo de t u.t. Los procesos no realizan operaciones de E/S y sólo emplean el procesador durante su ejecución. Indica cual es la expresión del grado de utilización del procesador para este sistema. Extrae conclusiones acerca del efecto del valor de t y q en la utilización.

Solución:

$$U = \frac{\sum_{i=1}^n T_i}{\sum_{i=1}^n T_i + t \sum_{i=1}^n \left\lceil \frac{T_i}{q} \right\rceil}$$

### EJERCICIO 22

Se dispone de un sistema monoprocesador con política de planificación del procesador RR con  $q = 10$  ut. y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1-47. El proceso A llega al sistema en el instante 0 ut., el proceso B en 4 ut. y el proceso C en 8 ut..

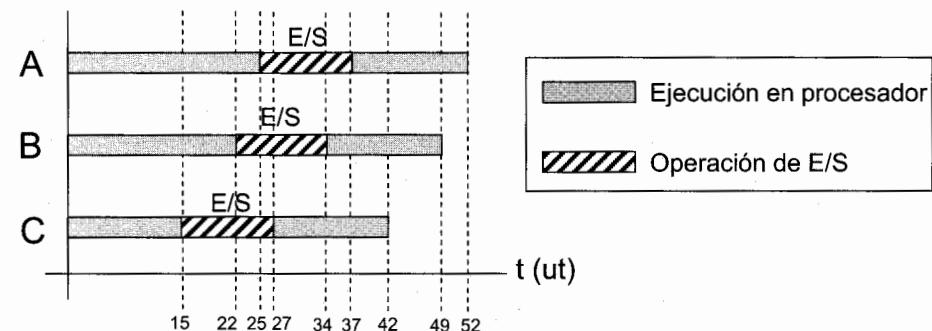


Figura 1-47. Esquema de ejecución de los procesos.

Mostrar la evolución temporal de los procesos en el sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y de los dispositivos de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos.

- Considera un tiempo de cambio de contexto de 0 ut.
- Considera un tiempo de cambio de contexto de 3 ut.
- Comenta los cambios en el algoritmo RR para mejorar la utilización de la CPU sin alterar el tiempo de cambio de contexto.

Nota: El tiempo en el que los procesos cambian de contexto no pertenece ni a ejecución ni a espera.

*Solución:*

- La evolución temporal de los procesos en el sistema se muestra en la figura 2:

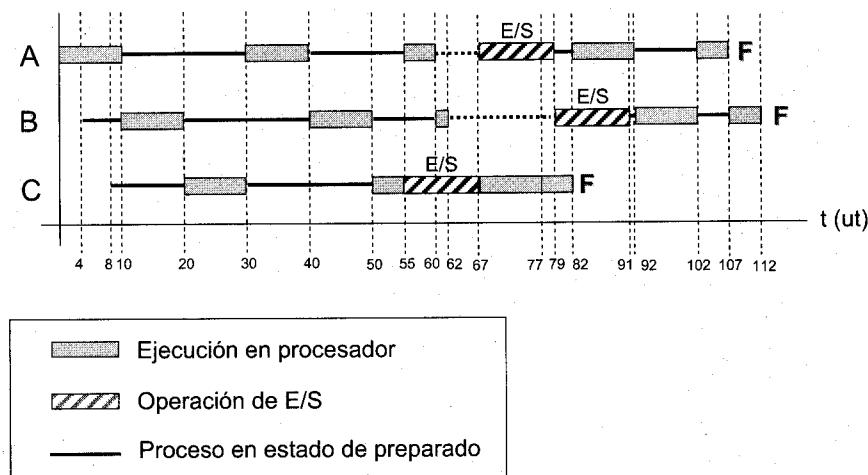


Figura 1-48: Evolución temporal de los procesos en el sistema.

Los tiempos se muestran en la tabla 1

Tabla 1-32: Tiempos de retorno, respuesta y espera de los procesos.

	T. Respuesta	T. Retorno	T. Espera
A	0	107	55
B	6	108	59
C	12	74	32

Cambios de contexto = 12

$$\text{Utilización del procesador} = (112-5)/112 = 95,53\%$$

- La evolución temporal de los procesos en el sistema se muestra en la figura 3:

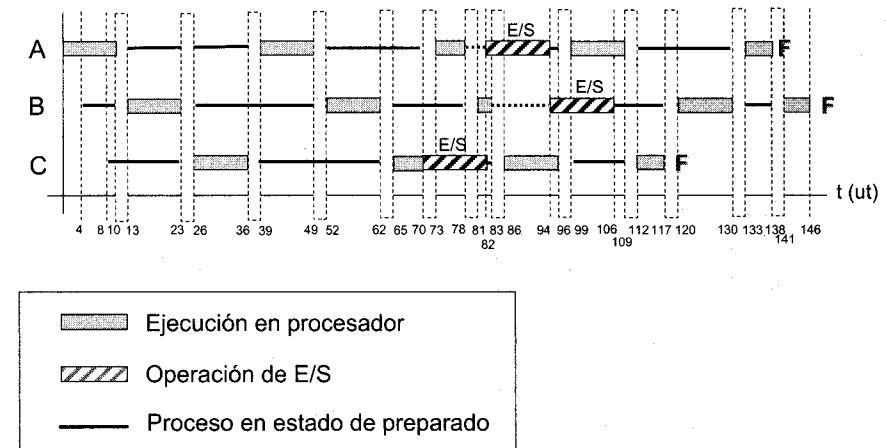


Figura 1-49: Evolución temporal de los procesos en el sistema.

Los tiempos se muestran en la tabla 2

Tabla 1-33: Tiempos de retorno, respuesta y espera de los procesos.

	T. Respuesta	T. Retorno	T. Espera
A	0	138	63
B	9	142	71
C	18	109	49

Cambios de contexto = 12

$$\text{Utilización del procesador} = (146-36)/146 = 75,34\%$$

c) Como se deduce de los resultados obtenidos en los apartados a) y b), los cambios de contexto alteran la utilización del procesador. A más cambios de contexto, peor grado de utilización se obtiene, por este motivo, una técnica para mejorar la utilización es la de incrementar el quantum del algoritmo RR.

### EJERCICIO 23

Se dispone de un sistema monoprocesador con política de planificación del procesador RR y con gestión de los dispositivos de E/S FCFS. La ejecución de los procesos al sistema sigue el esquema descrito en la figura 1. Los tres procesos llegan al sistema en el mismo instante de tiempo aunque se ejecutan en el orden A, B y C. Si el quantum de un proceso en ejecución expira a la vez que un proceso termina una operación de E/S, entonces el proceso que abandona el procesador se añade a la cola de procesos en preparado antes que el proceso que termina la E/S.

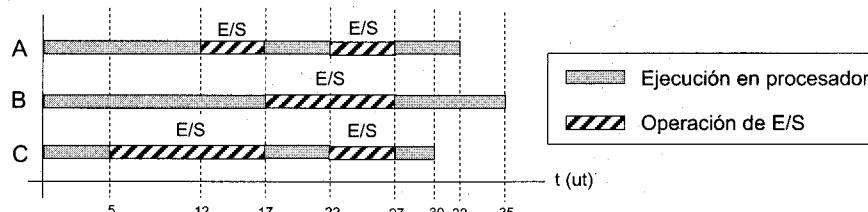


Figura 1-50. Ejecución de los procesos del sistema.

Determina el quantum del algoritmo de planificación RR para que el tiempo promedio de retorno de los procesos sea mínimo. Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y del dispositivo de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos. Considera un tiempo de cambio de contexto despreciable.

*Solución:*

El quantum del algoritmo RR puede ser muy pequeño para permitir que los procesos cambien frecuentemente ( $q=1$ ); muy alto ( $q=20$ ) para que el algoritmo RR funcione como FCFS; o bien un valor medio.

Probamos las tres alternativas y comprobamos en cual de ellas se obtiene un tiempo de retorno promedio menor.

En la figura 2 se muestra el cronograma para el caso 1 con  $q=1$

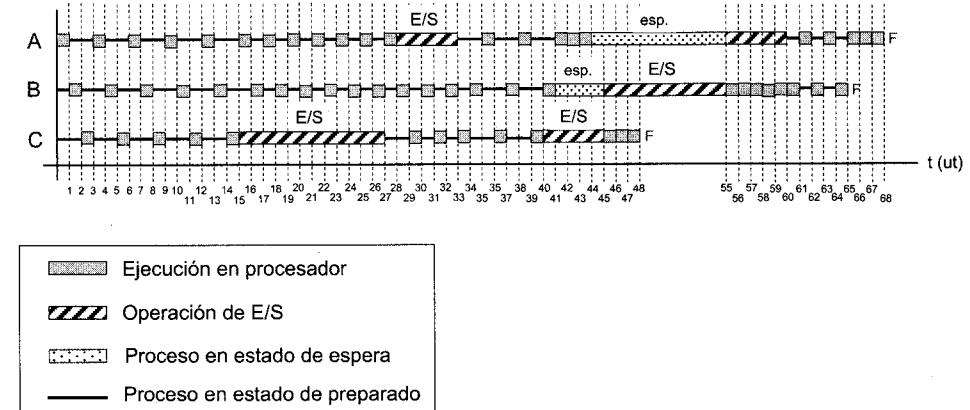


Figura 1-51: Evolución temporal de los procesos en el sistema con  $q = 1$ .

$$T \text{ ret. promedio} = (T \text{ ret. A} + T \text{ ret. B} + T \text{ retc. C}) / 3 = (68+65+48)/3 = 60,33 \text{ ut.}$$

En la figura 3 se muestra el cronograma para el caso 2 con  $q=20$

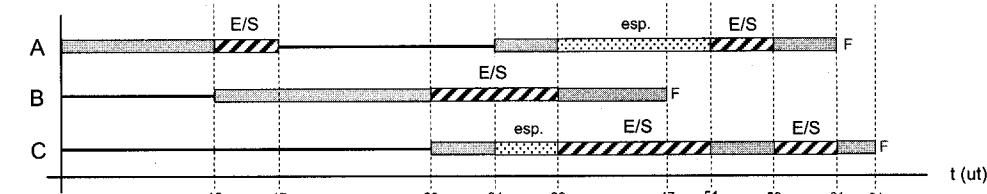


Figura 1-52: Evolución temporal de los procesos en el sistema con  $q = 20$ .

$$T \text{ ret. promedio} = (T \text{ ret. A} + T \text{ ret. B} + T \text{ retc. C}) / 3 = (61+47+64)/3 = 57,33 \text{ ut.}$$

En la figura 4 se muestra el cronograma para el caso 3 con  $q=5$

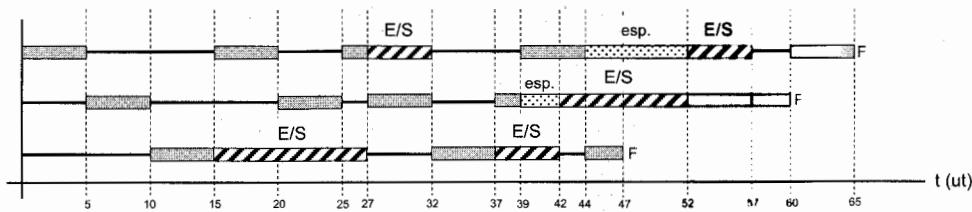


Figura 1-53: Evolución temporal de los procesos en el sistema con  $q = 5$ .

$$T \text{ ret. promedio} = (T \text{ ret. A} + T \text{ ret. B} + T \text{ ret. C}) / 3 = (65+60+47)/3 = 57,33 \text{ ut.}$$

Se observa que existen varios quantum distintos que alcanzan ese valor promedio mínimo.

#### EJERCICIO 24

Se dispone de un sistema con dos procesadores: el primer procesador tiene política de planificación RR con  $q = 10$  u.t. y ejecuta cualquier tipo de proceso; el segundo procesador tiene política FCFS y sólo ejecuta procesos de sistema. Se dispone de un recurso de E/S con gestión FCFS. El sistema recibe para su ejecución procesos de usuario y de sistema. La ejecución de los procesos de usuario sigue el esquema descrito en la figura siguiente.

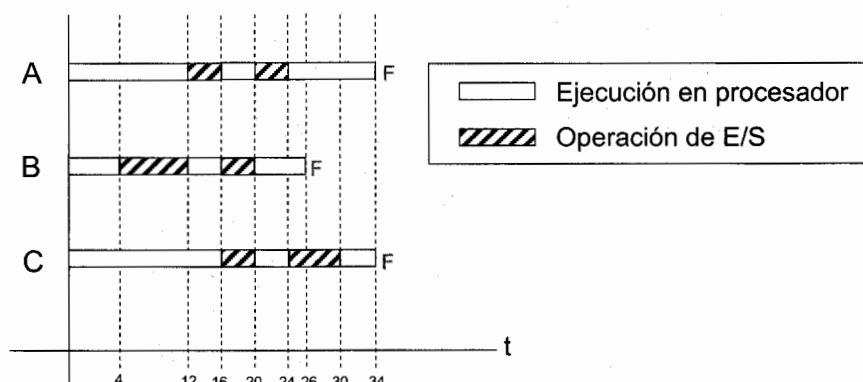


Figura 1-54. Ejecución de los procesos del sistema.

La ejecución de los procesos de sistema sigue el esquema que se muestra en la tabla 1-34. Los tiempos de ejecución periódica de estos procesos se

miden en relación al reloj del sistema. Su llegada coincide con el primer instante de su ejecución.

Tabla 1-34: Esquema de ejecución de los procesos de sistema.

<b>PS1</b>	Ejecución en CPU de duración 4 u.t.	Se ejecuta periódicamente cada 10 u.t
<b>PS2</b>	Ejecución en CPU de duración 4 u.t.	Se ejecuta periódicamente cada 20 u.t
<b>PS3</b>	Realiza E/S de duración 4 u.t.	Se ejecuta periódicamente cada 14 u.t

El instante de llegada de los procesos de usuario se muestra en la tabla siguiente:

Tabla 1-35: Instantes de llegada de los procesos de usuario.

<b>A</b>	0 u.t.
<b>B</b>	6 u.t.
<b>C</b>	10 u.t.

Los procesos de sistema tienen prioridad expropiativa sobre los procesos de usuario, salvo que haya algún procesador libre. No es posible expropiar recursos a procesos. Si el quantum de un proceso en ejecución expira a la vez que un proceso termina una operación de E/S o que la llegada de un nuevo proceso, entonces el proceso que abandona el procesador se añade a la cola de procesos en preparado antes que el proceso que termina la E/S o el que llega. Mostrar la evolución temporal de los procesos del sistema señalando el estado en el que se encuentra cada proceso, así como la ocupación temporal de la CPU y del dispositivo de E/S. Calcular los tiempos de respuesta, de retorno y de espera para cada uno de los procesos. Considera un tiempo de cambio de contexto despreciable.

#### Solución

La evolución temporal de la ejecución de los procesos se muestra en la figura 1-50.

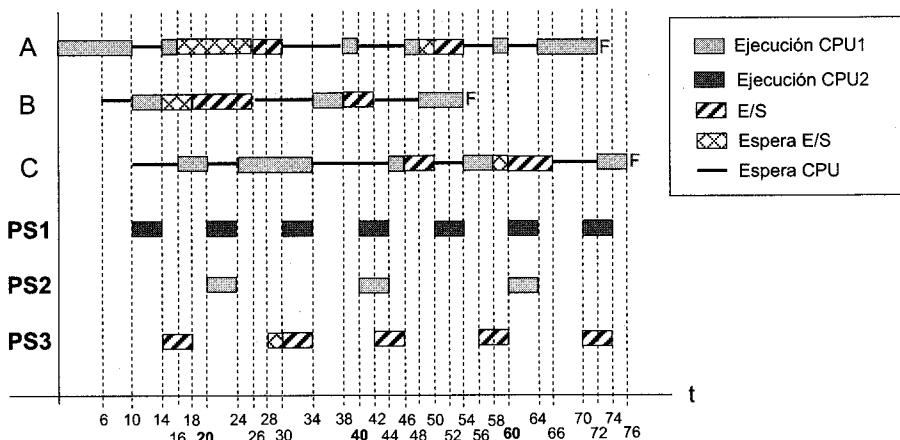


Figura 1-55: Evolución temporal de los procesos en el sistema.

Los tiempos de respuesta, retorno y espera se muestran en la tabla siguiente:

Tabla 1-36: Tiempos de retorno, respuesta y espera de los procesos.

	t. respuesta	t. retorno	t.espera
A	0	72	26+12
B	4	48	18+4
C	6	66	30+2

Utilización:

Procesador 1: 100%

Procesador 2:  $28/100 = 28\%$

## CAPÍTULO 2. CONCURRENCIA Y SINCRONIZACIÓN DE PROCESOS

### INTRODUCCIÓN

Hemos abordado en el capítulo anterior la situación, real por otra parte, de varios procesos ejecutándose simultáneamente, y la problemática asociada de la planificación de dichos procesos. Vamos a estudiar en el presente capítulo la necesidad de comunicación entre los procesos y cuales son los enfoques o herramientas utilizadas para llevar a cabo dicha comunicación.

Cuando los procesos solicitan servicios proporcionados por el sistema operativo, estos servicios son suministrados por procesos del propio sistema, es necesario comunicar el proceso de alto nivel con el del sistema operativo. Cuando ejecutamos varios comandos en el intérprete de órdenes y los conectamos con tuberías, el intérprete debe comunicar los datos de salida de un comando con la entrada del siguiente. Existen multitud de situaciones como las descritas donde es necesario comunicar procesos, para ello se deben proporcionar mecanismos de comunicación de procesos o IPC (InterProcess Communications).

Considerada la necesidad de comunicar distintos procesos, la pregunta es, ¿cómo podemos comunicar procesos? Existen dos mecanismos principales usados en la comunicación de procesos:

- Memoria compartida. Consiste en que los procesos comparten una zona de almacenamiento común, ésta zona puede ser la memoria principal o un archivo compartido en el disco. Para comunicarse, los procesos escriben y leen los datos en la zona compartida, y lo que escribe un proceso puede ser leído por el otro.
- Intercambio de mensajes. En este caso, los procesos no comparten nada, simplemente envían y reciben mensajes con los datos que desean transferir, de la misma forma que enviamos cartas o correos electrónicos.

Consideremos dos procesos A y B, que comparten una variable global x. En algún momento de la ejecución el proceso A incrementa la variable compartida, mientras que el proceso B la decrementa. Tanto el incremento como el decremento están formados por tres instrucciones a bajo nivel. La primera instrucción carga un registro de la CPU con el valor de la variable

x, la segunda realiza el incremento en el caso de A o el decremento en el caso de B, y la tercera devuelve el resultado a la variable compartida x.

Consideremos que la variable x tiene el valor 5 en un momento determinado y que los dos procesos ejecutan su instrucción en el mismo instante. Ahora, imaginemos la siguiente secuencia:

- El proceso A carga el registro de la CPU con el valor de 5.
- Se incrementa el valor a 6. En este momento, el planificador expripiá el procesador al proceso A y se lo otorga a B.
- El proceso B carga el registro de la CPU con el valor de 5.
- Se decrementa el valor a 4.
- Se devuelve el valor 4 a x. En este momento, el planificador expripiá el procesador al proceso B y se lo concede a A.
- Se devuelve el valor 6 a x.

Los dos procesos continuarán su ejecución con un valor de 6 en la variable x, el valor es erróneo, ya que el único valor correcto después de un incremento y un decremento es 5.

La situación anterior, y otros escenarios similares, se conoce como *condición de carrera*. Dicha condición se produce cuando varios procesos leen o escriben datos de tal forma que el resultado final depende del orden de ejecución de las instrucciones.

De nuevo surge una nueva pregunta, ¿cómo podemos evitar que se produzcan condiciones de carrera? La forma de evitar condiciones de carrera, es no permitir que dos o más procesos accedan a los datos compartidos simultáneamente, o lo que es lo mismo, en un momento dado, sólo un proceso puede estar accediendo a la zona de datos compartida. A esta condición se le conoce como *exclusión mutua*.

La parte de un programa que accede a datos compartidos con otros programas, y por lo tanto lo hace en exclusión mutua, se denomina *sección crítica*. La sección crítica debe cumplir las condiciones siguientes:

- Exclusión mutua. Dos procesos no pueden estar simultáneamente dentro de sus secciones críticas.
- Espera limitada. Ningún proceso podrá esperar indefinidamente para entrar en su sección crítica.
- Progresión. Un proceso que está fuera de su sección crítica no puede bloquear a otros procesos. Dicho de otra forma, si no hay ningún proceso en su sección crítica, cualquier proceso que llegue podrá entrar a su respectiva sección.

Para garantizar que las secciones críticas de los procesos cumplen las condiciones anteriores, en la mayoría de los programas concurrentes se sigue un protocolo a la hora de abordar la parte concerniente a la sección crítica. Antes de entrar en la sección crítica, implementaremos el protocolo

de entrada, cuya función principal será garantizar la exclusión mutua y la progresión. Después de salir de la sección o región crítica, implementaremos el protocolo de salida para habilitar que otros procesos puedan pasar a su sección y no queden esperando indefinidamente, estos procesos es probable que estén bloqueados en su protocolo de entrada debido a que el proceso saliente estaba dentro de la sección (exclusión mutua).

Para solucionar el problema de la sección crítica se han usado principalmente las siguientes herramientas:

- Soluciones software. En estas soluciones se utiliza un algoritmo software que implementa los protocolos de entrada y salida. Un exponente muy conocido de este tipo de soluciones es el algoritmo de Peterson.
- Soluciones hardware. Consisten en inhabilitar interrupciones antes de entrar en la sección crítica, sin interrupciones no se puede expripiar el procesador, y habilitar interrupciones después de salir de la región crítica.
- Semáforos.
- Monitores.

Un *semáforo* se define mediante un tipo abstracto de datos que está formado por dos componentes y tres operaciones. Los componentes serán un *contador* entero y una *cola* de procesos en espera. Las operaciones que se pueden realizar en un semáforo son las siguientes:

- Inicializar: Inicia el contador a un valor no negativo
- P(): Disminuye en una unidad el valor del contador. Si el contador se hace negativo, el proceso que ejecuta P se bloquea.
- V(): Aumenta en una unidad el valor del contador. Si el valor del contador no es positivo, se desbloquea un proceso bloqueado por una operación P.

Las operaciones son atómicas a nivel hardware, lo que significa que mientras se está realizando una operación de un semáforo no se puede interrumpir el procesador. De esta manera se garantiza que una vez iniciada una operación de un semáforo, ésta se llevará a cabo de forma continua y sin interrupción. Esta característica impide, por ejemplo, que dos procesos distintos ejecuten una operación P sobre un semáforo al mismo tiempo.

Se denomina *semáforo binario* a un caso particular de semáforos donde el valor del contador sólo puede ser 0 ó 1, y no cualquier número entero. Debemos observar que en un semáforo general, si el valor del contador es negativo, dicho valor nos está indicando el número de procesos que están bloqueados en el semáforo.

Una ventaja de los semáforos con respecto a las soluciones software es que, el proceso que espera entrar en la SC no usa el procesador, está bloqueado. Por el contrario, en los algoritmos software el proceso está en un bucle comprobando continuamente el estado de las condiciones y usando el

procesador, esto se conoce como *espera activa*. Pero las condiciones no pueden cambiar, a menos que las cambie otro proceso, por lo que es mejor no usar el procesador y dejarlo disponible para el resto de procesos, bloquearse y que se desbloquee cuando cambien las condiciones.

Toda programación concurrente suele tener dos tipos de problemas, la comunicación y la sincronización. Si el problema es complejo, es difícil implementar una solución completamente correcta utilizando semáforos. Es mejor utilizar una herramienta de más alto nivel como los monitores.

Un monitor está formado por tres componentes: variables, inicialización, y procedimientos del monitor. Variables y procedimientos puede haber tantos como se quiera. Los procesos pueden invocar a los procedimientos del monitor, pero no pueden acceder directamente a las variables del monitor. El monitor garantiza la exclusión mutua, ya que sólo permite un proceso activo en el monitor en un momento dado.

Existe un tipo espacial de variables que se llaman *variables de condición*, que son variables sobre las que se pueden realizar sólo dos operaciones: espera y señal. Espera sobre una variable de condición bloquea siempre al proceso que la invoca. La operación señal sobre una variable de condición desbloquea a un proceso, en el caso de haber procesos bloqueados en dicha variable. Estas variables son de gran utilidad para solucionar los aspectos de sincronización en un problema de concurrencia.

El intercambio de mensajes constituye la segunda alternativa a la hora de comunicar procesos. El mecanismos de transferencia de mensajes consiste en dos operaciones o primitivas: *enviar* y *recibir*. La operación de enviar, envía un mensaje a un destinatario que puede ser un proceso o un buzón. La primitiva recibir, permite recibir un mensaje de un proceso concreto o de un buzón. La comunicación se dice que es directa si involucra a procesos, en cambio, es indirecta si una parte de la comunicación es un buzón. Además, las operaciones enviar y recibir pueden ser bloqueantes o no bloqueantes.

La principal ventaja de los mensajes con respecto a los mecanismos analizados previamente, es que se pueden usar en entornos distribuidos y redes de comunicaciones, mientras que el resto al tener que compartir memoria están más pensados para sistemas locales.

Finalmente, indicar que los problemas de concurrencia se suelen abordar estudiando problemas clásicos de comunicación y sincronización entre procesos. Los problemas del productor y consumidor, lectores y escritores, y la comida de filósofos, son buenos ejemplos. El resto del capítulo analiza estos problemas clásicos, además de otros que presentan una problemática similar.

## SEMÁFOROS

### EJERCICIO 1

Uno o más productores generan cierto tipo de datos y los sitúan en una zona de memoria o buffer. Un único consumidor saca elementos del buffer de uno en uno. El sistema debe impedir la superposición de operaciones sobre el buffer.

Este es uno de los problemas clásicos de comunicación y sincronización entre procesos. Comunicación a través de una zona común de memoria que es el buffer, este buffer será la sección crítica que como sabemos se debe acceder en exclusión mutua. Sincronización en dos aspectos básicos; por una parte, si un consumidor intenta coger un elemento del buffer y no hay ningún elemento, debe bloquearse hasta que algún productor añada un elemento, por otra, si un productor intenta añadir un nuevo elemento y el buffer está lleno, debe bloquearse hasta que el consumidor retire algún elemento del buffer.

Este problema tiene dos variantes o se aborda desde dos puntos de vista distintos, una variante asume que el tamaño del buffer es ilimitado, mientras que la segunda asume un tamaño de buffer limitado. La diferencia entre ambas es que en la versión ilimitada sólo existe un problema de sincronización que afecta al consumidor, mientras que la versión limitada presenta los dos problemas de sincronización descritos en el párrafo anterior.

#### *Solución: Tamaño de buffer ilimitado*

En esta variante tenemos el problema de sección crítica en el acceso al buffer, no pueden existir dos procesos accediendo simultáneamente al buffer, y uno de los problemas de sincronización, el del consumidor.

TSemáforo s, n;

```

void productor()
{
    while (true)
    {
        producir();
        P(s);
        añadir_buffer();
        V(s);
        V(n);
    }
}

void consumidor()
{
    while (true)
    {
        P(n);
        P(s);
        coger_buffer();
        V(s);
        consumir();
    }
}

void main()
{
    inicializar(s, 1); inicializar(n, 0);
    cobegin
        productor();
        consumidor();
    coend;
}

```

Observemos con detenimiento la solución presentada. Primero, se han declarado dos semáforos: uno para garantizar el acceso a la sección crítica (*s*) y otro para abordar el problema de sincronización del consumidor (*n*).

El semáforo *s* nos garantiza el acceso en exclusión mutua de la siguiente manera:

- Lo inicializamos a 1 para que sólo un proceso esté en un momento dado accediendo al buffer.
- Todos los procesos (tanto productores como consumidores) ejecutan *P(s)* justo antes de entrar en la sección crítica. Por tanto, si existe un proceso en la sección crítica, el valor de contador del semáforo será 0, y cualquier otro proceso que realice la operación *P* sobre el semáforo *s* se bloqueará automáticamente.

El semáforo *n* nos permite sincronizar al consumidor, se inicializa a 0 indicando que inicialmente no existen elementos. El valor del contador nos indicará en todo momento el número de elementos que tiene el buffer, pero este valor no lo podemos saber, ya que no disponemos de ningún procedimiento del semáforo que nos devuelva dicho valor. Ahora bien, el valor del contador del semáforo *n* coincide con el número de operaciones *P(n)* seguidas que puede ejecutar el consumidor sin bloquearse.

Supongamos la situación inicial y supongamos que llega primero un consumidor, éste ejecutará la operación *P(n)* y se bloqueará ya que el semáforo se ha inicializado a 0, cosa lógica si tenemos en cuenta que inicialmente el buffer está vacío. Supongamos que a continuación llega un productor, ejecuta *P(s)* (no se bloquea debido a que *s* se ha inicializado a 1 y es el primer proceso que realiza una *P*) y añade un elemento. Sale de la sección crítica y al final ejecuta *V(n)*, operación que despertará al consumidor y le permitirá progresar.

#### *Solución: Tamaño de buffer limitado*

En la segunda solución propuesta tenemos el problema de sección crítica en el acceso al buffer, no pueden existir dos procesos accediendo simultáneamente al buffer, y los dos problemas de sincronización. El consumidor debe bloquearse si intenta coger un elemento y no existe ninguno, y el productor debe bloquearse si después de producir un elemento intenta dejarlo en el buffer y éste está lleno.

Esta solución es muy similar al caso anterior, obviamente la definición es prácticamente idéntica, por lo que no comentaremos las similitudes de código ya estudiadas en el caso del buffer ilimitado. Se añade un nuevo semáforo *e* para tratar el nuevo problema de sincronización, sincronizar al productor.

Observemos la inicialización del nuevo semáforo, éste semáforo se inicializa al tamaño del buffer. Recordemos que el valor de un semáforo nos indica el número de operaciones *P* seguidas que se pueden realizar sin bloquear al proceso que las invoca, por lo tanto, el productor podrá realizar *N* operaciones *P* seguidas sin bloquearse, lógico si pensamos que el buffer está vacío inicialmente y que tiene capacidad *N*.

Supongamos que el buffer está lleno y llega un productor, éste ejecutará la operación *P(e)* y se bloqueará, ya que el contador del semáforo será 0 en este momento. Supongamos que a continuación llega un consumidor, coge un elemento y sale de la sección crítica, al final ejecutará *V(e)*, operación que despertará al productor y le permitirá avanzar.

```

#define tamaño_buffer N
TSemáforo e, s, n;

void productor()
{
    while (true)
    {
        producir();
        P(e);
        P(s);
        añadir_buffer();
        V(s);
        V(n);
    }
}

void consumidor()
{
    while (true)
    {
        P(n);
        P(s);
        coger_buffer();
        V(s);
        V(e);
        consumir();
    }
}

void main()
{
    inicializar(s, 1); inicializar(n, 0);
    inicializar(e, tamaño_buffer);
    cobegin
        productor();
        consumidor();
    coend;
}

```

## EJERCICIO 2

Se dispone de una zona de memoria o fichero a la que acceden unos procesos (lectores) en modo lectura y otros procesos en modo escritura (escritores).

- Los lectores pueden acceder al fichero de forma concurrente.
- Los escritores deben acceder al fichero de manera exclusiva entre ellos y con los lectores.

El sistema debe coordinar el acceso a la memoria o al fichero para que se cumplan las restricciones.

Como ocurría en el problema del productor-consumidor, el problema de los lectores y escritores también tiene distintas variantes, concretamente tres. Las variantes tienen que ver con otro de los aspectos importantes estudiados generalmente en problemas de concurrencia: el concepto de prioridad. Se suelen analizar tres prioridades cuando tenemos dos tipos de procesos: prioridad de un tipo, del otro, y prioridad por orden de llegada (sin prioridad).

### Solución: Prioridad a los lectores

En esta solución se utiliza el semáforo  $w$  para el tratamiento de la sección crítica. En el caso de los escritores y por la definición del problema, sólo puede haber un escritor en un momento dado accediendo al fichero, por lo que el código expuesto obliga a todos los escritores a realizar la operación  $P$  sobre el semáforo  $w$ , semáforo que se ha inicializado a 1. La obligación de que todos ejecuten ésta operación y la inicialización a 1 nos garantiza que nunca existirán dos escritores accediendo al fichero.

Analicemos el caso de los lectores, por definición del problema podemos tener varios lectores leyendo el fichero simultáneamente. Esto se consigue eliminando la obligación de que todos los lectores ejecuten  $P(w)$ . La idea es que el primer lector realice la operación  $P(w)$ , y mientras el primer lector esté leyendo en el fichero (sección crítica), cualquier otro lector que llegue pueda pasar a leer sin ejecutar la citada operación  $P$  y, por tanto, sin posibilidad de bloquearse.

Para asegurar la progresión de otros procesos se suele ejecutar en el protocolo de salida la operación  $V$  sobre el semáforo asociado a la sección crítica ( $w$  en este caso). En el caso de los escritores, todos ejecutan la operación  $V(w)$ , por el contrario, no todos los lectores realizan esta operación, sino sólo el último en salir de la sección crítica.

Hemos comentado que sólo el primer lector ejecuta la operación  $P$  sobre el semáforo  $w$ , y que sólo el último lleva a cabo la operación  $V$ . Pero, ¿cómo sabemos cual es el primer proceso y cual el último?, añadiendo una variable que nos indicará en todo momento el número de lectores que están simultáneamente leyendo en el fichero.

El hecho de añadir una variable a la que pueden acceder varios lectores lleva asociado un nuevo problema de sección crítica en el acceso a ésta variable. Puede haber un lector incrementando la variable al entrar en el sistema y, simultáneamente, otro lector decrementando la variable al salir del sistema. Para garantizar el acceso en exclusión mutua a la variable se utiliza el semáforo *mutex*, de tal manera que, cuando un lector quiera acceder a la variable, deberá ejecutar primero la operación  $P$  sobre el semáforo *mutex*, realizando la operación  $V$  después del acceso para permitir que cualquier otro lector pueda acceder a la variable.

El último aspecto que debemos tener en cuenta en la solución propuesta es el concepto de prioridad. En la solución de prioridad a los lectores, la prioridad está implícita en el código, a diferencia de la prioridad escritores que, como veremos, asocia un semáforo al tratamiento de la prioridad.

Supongamos que llegan por éste orden, un lector, un escritor y un nuevo lector. El primer lector entraría a leer el fichero, al llegar el escritor se

quedaría bloqueado en la instrucción  $P(w)$ , el semáforo  $w$  lo ha cerrado el lector que está leyendo en estos momentos. Finalmente, llega el segundo lector, incrementa la variable *lectores* (*lectores*=2), y pasa a leer directamente. Como hemos comprobado, el segundo lector entra en la sección crítica antes que el escritor, pese a haber llegado después, por ello, consideramos que los lectores tienen prioridad.

```
TSemáforo mutex, w;
int lectores;
```

```
void escritor_i()
{
    . .
    P(mutex);
    escribir();
    V(w);
    .
}

void lector_i()
{
    .
    P(mutex);
    lectores++;
    if (lectores==1) P(w);
    V(mutex);
    leer();
    P(mutex);
    lectores--;
    if (lectores==0) V(w);
    V(mutex);
    .
}

void main()
{
    inicializar(mutex, 1); inicializar(w, 1);
    lectores=0;
    cobegin
        escritor_i(); ...; escritor_n(); lector_i(); ... ; lector_m();
    coend;
}
```

### Solución: Prioridad a los escritores

Podemos observar que la utilización del semáforo  $w$  es exactamente igual que el caso anterior, tanto para escritores como para lectores. No ocurre lo mismo con el tratamiento de la prioridad, en este caso los escritores tienen prioridad sobre los lectores y, a diferencia del caso anterior, la prioridad no está implícita en el algoritmo sino que utilizamos un semáforo adicional  $r$  para implementar la prioridad. Para dar prioridad a los escritores, la idea es que sólo el primer escritor realice la operación  $P(r)$ , y mientras el primer escritor esté accediendo al fichero (sección crítica), cualquier otro escritor que llegue pase sin ejecutar la citada operación  $P$  en el semáforo  $r$  y, por

tanto, sin posibilidad de bloquearse en el semáforo de tratamiento de la prioridad. Por el contrario, todos los lectores tendrán la obligación de ejecutar la operación  $P(r)$ , y no en cualquier lugar, sino que será la primera instrucción a ejecutar cuando llegan al sistema. Además, la última operación del protocolo de entrada que realizarán los lectores es  $V(r)$ , para habilitar que mientras unos lectores estén leyendo, otros puedan acceder y no se queden bloqueados en la primera instrucción.

Como en la solución anterior, sólo el primer escritor ejecuta la operación  $P$  sobre el semáforo  $r$ , y sólo el último lleva a cabo la operación  $V$ . Pero, ¿cómo sabemos cual es el primer escritor y cual el último?, añadiendo una variable que nos indicará en todo momento el número de escritores. La variable se convierte en sección crítica y se trata exactamente igual que la variable *lectores*.

```
TSemáforo mutex1, mutex2, w, r;
int lectores, escritores;
```

<pre>void lector_i() {     .     P(r);     P(mutex1);     lectores++;     if (lectores==1) P(w);     V(mutex1);     V(r);     leer();     P(mutex1);     lectores--;     if (lectores==0) V(w);     V(mutex1);     . }</pre>	<pre>void escritor_j() {     .     P(mutex2);     escritores++;     if (escritores==1) P(r);     V(mutex2);     P(w);     escribir();     V(w);     P(mutex2);     escritores--;     if (escritores==0) V(r);     V(mutex2);     . }</pre>
--	--

```
void main()
{
    inicializar(mutex1, 1); inicializar(mutex2, 1);
    inicializar(w, 1); inicializar(r, 1);
    lectores = 0; escritores = 0;
    cobegin
        escritor_i(); ...; escritor_n(); lector_i(); ... ; lector_m();
    coend;
}
```

Supongamos que llegan por éste orden, un lector, un escritor, un lector y un escritor. El primer lector entraría a leer el fichero, al llegar el escritor se queda

bloqueado en la instrucción  $P(w)$ , el semáforo  $w$  lo ha cerrado el lector que está leyendo en estos momentos. Pero, antes de quedarse bloqueado ha ejecutado la instrucción  $P(r)$ , ya que es el primer escritor. Ahora, llega el segundo lector, la primera instrucción que ejecuta es  $P(r)$  que lo bloquea automáticamente, el semáforo  $r$  lo ha cerrado el primer escritor. Finalmente, llega el último escritor que también se queda bloqueado en el semáforo  $w$ .

Cuando salga el lector ejecutará  $V(w)$ , que despertará al primer escritor, y cuando salga éste escritor ( $V(w)$ ) despertará al segundo lector, y no al segundo escritor. Sólo cuando salga el segundo y último escritor ( $escritores==0$ ) ejecutará la instrucción  $V(r)$ , permitiendo progresar al lector bloqueado en el semáforo  $r$  inicial.

#### Solución: Acceso según orden de llegada

La solución sin prioridad es muy parecida a la primera solución presentada al problema, la diferencia es que se añade un semáforo para la gestión de la prioridad (*fifo*) y el semáforo se emplea de la siguiente manera: tanto lectores como escritores ejecutarán la operación  $P$  sobre el nuevo semáforo como primera instrucción de su protocolo de entrada, y la operación  $V$  como última instrucción del protocolo de entrada.

Se debe notar que la operación  $V$  sobre el semáforo *fifo* se puede situar al final del código, pero en el caso de los lectores sólo permitiría pasar uno, con lo que nos cargaríamos una restricción del problema.

```
TSemáforo mutex, fifo, w;
int lectores;
```

<pre>void lector_i() {     . .     P(fifo);     P(mutex);     lectores++;     if (lectores==1) P(w);     V(mutex);     V(fifo);     leer();     P(mutex);     lectores--;     if (lectores==0) V(w);     V(mutex);     . }</pre>	<pre>void escritor_i() {     . .     P(fifo);     P(w);     V(fifo);     escribir();     V(w);     . }</pre>
--	--

```
void main()
{
    inicializar(mutex, 1); inicializar(fifo, 1);
    inicializar(w, 1);
    lectores = 0;
    cobegin
        escritor_1(); ...; escritor_n(); lector_1(); ... ; lector_m();
    coend;
}
```

#### EJERCICIO 3

Una barbería tiene una sala de espera con  $n$  sillas, y una habitación con un sillón donde se atiende a los clientes. Si no hay clientes el barbero se duerme. Si un cliente entra en la barbería y todas las sillas están ocupadas, entonces se va, sino, se sienta en una de las sillas disponibles. Si el barbero está dormido, el cliente lo despertará. El sistema debe coordinar el barbero y los clientes.

#### Solución

La solución utiliza tres semáforos, *clientes* para sincronizar (bloquearlo) al barbero cuando no hay clientes en la barbería, se inicializa a 0 y el barbero lo primero que ejecuta es una operación  $P$  sobre el semáforo, operación que lo bloqueará excepto en el caso de que hayan llegado antes clientes y hayan ejecutado la operación  $V$  correspondiente.

El segundo semáforo es *barbero*, en este semáforo se bloquean los clientes al ejecutar la operación  $P$ , los clientes se bloquean en espera de que el barbero vaya despertando uno a uno con la ejecución de la instrucción  $V(\text{barbero})$ .

Finalmente, podemos observar la utilización del semáforo *mutex*, semáforo que ya hemos usado en los ejercicios anteriores y que utilizaremos extensamente a lo largo de todo el capítulo.

Es un semáforo asociado a la variable *espera* que nos garantiza el acceso en exclusión mutua a la variable, la variable *espera* nos indica en todo momento el número de clientes que están en la sala de espera. Como en los ejemplos anteriores, debemos realizar una llamada  $P(\text{mutex})$  antes de acceder a la variable, para asegurarnos que sólo nosotros estaremos accediendo a la variable en ese momento, y una llamada  $V(\text{mutex})$  después del acceso, para permitir que otros procesos puedan acceder igualmente.

Supongamos que llegan  $n+1$  clientes y que no ha llegado el barbero, los primeros  $n$  clientes incrementarán la variable *espera* y se quedarán bloqueados en el semáforo *barbero* (hasta que llegue el barbero y los despierte), el último cliente comprobará que *espera* ya no es menor que el número de sillas y abandonará la barbería ejecutando  $V(mutex)$  para liberar el semáforo.

```
#define sillas n
TSemáforo mutex, clientes, barbero;
int espera;

void barbero()
{
    while (true)
    {
        P(clientes);
        P(mutex);
        espera=espera-1;
        V(barbero);
        V(mutex);
        cortar_pelo();
    }
}

void cliente_i()
{
    P(mutex);
    if (espera<sillas)
    {
        espera=espera+1;
        V(clientes);
        V(mutex);
        P(barbero);
        se_corta_pelo();
    }
    else V(mutex);
}

void main()
{
    inicializar(mutex, 1); inicializar(clientes, 0);
    inicializar(barbero, 0); espera=0;
    cobegin
        barbero();
        cliente_1(); cliente_2(); ... cliente_n();
    coend;
}
```

#### EJERCICIO 4

Cinco filósofos se dedican a pensar y a comer en una mesa circular. En el centro de la mesa hay un cuenco con arroz, y la mesa está puesta con cinco platos y cinco palillos, uno por cada filósofo. Cuando un filósofo tiene hambre se sienta en la mesa a comer en su sitio. El filósofo sólo puede coger un palillo cada vez y no le puede quitar un palillo a un compañero que lo tenga en la mano. Cuando un filósofo tiene los dos palillos come sin soltarlos hasta que termine y vuelve a pensar.

El sistema debe coordinar los filósofos para evitar la espera indefinida y no se mueran de hambre.

*Solución: mantiene la exclusión mutua*

Se produce interbloqueo cuando acuden a comer todos a la vez. En la solución, realizar la operación *P* sobre cualquiera de los semáforos palillo simula coger el palillo, mientras que la operación *V* simula dejar libre el palillo.

Supongamos que todos los filósofos intentan comer al mismo tiempo y llegan al sistema simultáneamente, todos los filósofos ejecutarán la operación *P* sobre el palillo de su mismo índice  $i$ , la operación será satisfactoria en todos ellos y todos obtendrán el palillo correspondiente a su índice. A continuación, los filósofos realizarán la operación *P* sobre el semáforo asociado al palillo  $i+1$ , pero en este caso todos los filósofos quedarán bloqueados debido a que el palillo siguiente lo tiene el siguiente filósofo, que como él lo cogió en la instrucción anterior. A esta situación el la que todos los procesos del sistema están bloqueados se le conoce con el nombre de interbloqueo, y se estudiará en los problemas del siguiente capítulo.

Es importante que observemos el hecho de que al resolver problemas de comunicación y sincronización, el protocolo de salida suele servir para que un proceso ejecute alguna instrucción que permita la progresión de algún otro proceso bloqueado. En una situación de interbloqueo, donde todos los procesos están bloqueados, ningún proceso podrá llegar al protocolo de salida, y por lo tanto ningún proceso podrá hacer que progresen los demás.

```
TSemáforo palillo[5];

void filósofo(int i)
{
    while (true)
    {
        pensar();
        P(palillo[i]);
        P(palillo[(i+1)%5]);
        comer();
        V(palillo[i]);
        V(palillo[(i+1)%5]);
    }
}

void main()
{ int i;
    for (i=0; i<5; i++) inicializar(palillo[i], 1);
```

```

cobegin
    filosofo(0); filosofo(1); ... filosofo(4);
coend;
}

```

*Solución: evita los interbloqueos y mantiene la exclusión mutua*

En esta solución, realizar las operaciones *P* y *V* sobre cualquiera de los semáforos *palillo* tiene el mismo significado que en la primera aproximación estudiada. Esta implementación mantiene la exclusión mutua (como en el caso anterior) y evita la situación de interbloqueo.

Para tratar el problema del interbloqueo introducimos un nuevo semáforo *silla*, que nos garantiza que sólo un único filósofo estará en un momento dado intentando coger los palillos. Para ello, y al igual que si de una sección crítica se tratase, todos los filósofos antes de intentar coger los palillos ejecutarán la operación *P* sobre *silla*, y después de coger los dos palillos realizarán la operación *V*, liberando el semáforo y permitiendo que el resto de filósofos puedan intentarlo.

En esta solución que evita el interbloqueo, el número de filósofos que pueden comer simultáneamente depende de la dinámica del sistema pero serán uno o dos (no pueden ser más de dos porque sólo tenemos 5 palillos). El primer filósofo que llegue podrá pasar a comer con sus dos palillos, en cambio, mientras el primer filósofo esté comiendo, el segundo que llegue dependerá de qué filósofo sea.

Si está comiendo el filósofo 1 y llega el 2, el segundo filósofo quedará bloqueado porque no podrá coger el palillo 2. En cambio, si está comiendo el 1 y llega el 3, éste último podrá coger tranquilamente los palillos 3 y 4, y por tanto comer.

```

TSemáforo palillo[5], silla;

void filosofo(int i)
{
    while (true)
    {
        pensar();
        P(silla);
        P(palillo[i]);
        P(palillo[(i+1)%5]);
        V(silla);
        comer();
        V(palillo[i]);
        V(palillo[(i+1)%5]);
    }
}

```

```

void main()
{
    int i;
    for (i=0; i<5; i++) inicializar(palillo[i], 1);
    inicializar(silla, 4);
    cobegin
        filosofo(0); filosofo(1); ... filosofo(4);
    coend;
}

```

#### EJERCICIO 5

En un comedor de un colegio hay *n* alumnos. Cuando un alumno, que normalmente está estudiando en su clase, tiene hambre va al comedor y se acerca a alguna de las tres barras para recoger el menú. En estos menús, que varían cada día, sólo se puede elegir o cambiar el postre o café. Para el postre hay dos mostradores mientras que para el café hay un único mostrador. Realizar un programa que, usando semáforos, coordine las peticiones de los alumnos en el comedor.

*Solución*

Este es un problema en el que el número de recursos es limitado y definido, tres barras, dos de postre y una de café.

Como podemos observar, se utilizan cuatro semáforos: el semáforo *comedor* asociado a la sala y que se inicializa a *n*, indicando que tiene una capacidad de *n* comensales; *menu* inicializado a 3, representando a las tres barras donde se sirven menús; *postre* inicializado a 2, indicando las dos barras donde se sirven postres; y el semáforo *cafe* inicializado a 1, simulando la única barra donde existe café.

```

#define alumnos n
TSemáforo comedor, menu, postre, cafe;
bool postre_cafe;

void alumno()
{
    while (true)
    {
        P(comedor);
        entrar_en_comedor();
        P(menu);
        coger_menu();
        V(menu);
        decidir_postre_cafe();
    }
}

```

```

if (postre_cafe == postre)
{
    P(postre);
    coger_postre();
    V(postre);
}
else
{
    P(cafe);
    coger_cafe();
    V(cafe);
}
comer();
V(comedor);
}

Void main()
{
    inicializar(comedor, n);  inicializar(menu, 3);
    inicializar(postre, 2);  inicializar(cafe, 1);
    cobegin
        alumno1(); alumno2(); ... alumnom();
    coend;
}

```

Observemos como en los semáforos *menú*, *postre* y *cafe* se ejecuta la operación *P* justa antes de llevar a cabo la acción correspondiente y la operación *V* inmediatamente después, para liberar la barra y que otro compañero pueda acceder. En cambio, en el semáforo *comedor* se ejecuta la operación *P* al entrar en el comedor y la operación *V* al salir del mismo, asegurando de este modo que no se superará la capacidad del comedor.

#### EJERCICIO 6

Un parque de atracciones decide usar semáforos en sus actividades de mayor afluencia de público. En concreto en las pistas blandas (con 5 pistas) y el kamikaze( con dos pistas). Además, por exigencias de seguridad, se obliga a que no haya más de 2000 personas en el parque, todos deben de ducharse antes de entrar en estas actividades que se realizarán en el orden que se han descrito. Realizar un programa con semáforos que cumpla las exigencias descritas del parque acuático.

#### Solución

Como es fácilmente observable, este problema guarda muchas similitudes con el anterior, también es un problema de recursos limitados y definidos, 2000 personas de capacidad máxima en el parque, cinco pistas blandas, dos kamikaze y número de duchas indefinido.

Como en el caso anterior, asociaremos un semáforo a cada uno de los recursos anteriores y trabajaremos con la inicialización de mismos. También como en el problema anterior, se puede ver como en la utilización de las atracciones el semáforo correspondiente se cierra y se libera (operaciones *P* y *V*) antes y después del uso de la atracción. En cambio, en el semáforo parque la operación *P* se ejecuta al entrar y la operación *V* al salir, garantizando en todo momento no exceder la capacidad del parque.

Observemos la estructura secuencial del código que garantiza que las actividades se realizan en el orden en que se han descrito, en cumplimiento de la definición del problema.

```

#define duchas n
TSemaforo blandas, kamikaze, parque, duchas;

void cliente_i()
{
    while (true)
    {
        P(parque);
        entrar_en_el_parque();
        P(duchas);
        ducharse();
        V(duchas);
        P(blandas);
        atraccion_pistasblandas();
        V(blandas);
        P(kamikaze);
        Atraccion_kamikaze();
        V(kamikaze);
        V(parque);
    }
}

Void main()
{
    inicializar(parque, 2000);
    inicializar(duchas, n);
    inicializar(blandas, 5);
    inicializar(kamikaze, 2);
    cobegin
        cliente_1(); cliente_2(); ... cliente_m();
    coend;
}

```



## EJERCICIO 7

En el almacén de una empresa de logística existen distintos toritos que pueden ser utilizados por personal del almacén que mueve los bultos internamente y por personal de los muelles que carga los camiones de transporte. Supongamos que la gestión de los toritos se implementa de la siguiente forma:

```
TSemáforo sc, mutex;
int numero;
```

<pre>void muelle_i() {     . .     P(sc);     coger_torito();     V(sc);     . . }</pre>	<pre>void almacen_j() {     . .     P(mutex);     numero++;     if (numero==1) P(sc);     V(mutex)     coger_toriro();     P(mutex);     numero--;     if (numero==0) V(sc);     V(mutex)     . . }</pre>
--	---

```
void main()
{
    inicializar(mutex, 1);
    inicializar(sc, 1);
    numero=0;
    cobegin
        muelle_i(); ...; muelle_n(); almacen_i(); ... ; almacen_m();
    coend;
}
```

- 1) ¿Pueden utilizar a la vez los toritos el personal del almacén y del muelle?
- 2) Si hay un señor del almacén utilizando un torito, ¿puede otro del almacén utilizar algún torito simultáneamente?
- 3) Si una persona de los muelles está cargando, ¿puede otra persona de los muelles cargar al mismo tiempo?
- 4) Indicar si existe algún tipo de prioridad. En caso afirmativo, realizar las modificaciones necesarias para que no exista prioridad.

## Solución

En el apartado 1 la respuesta es negativa, no pueden utilizar ambos tipos de personas los toritos simultáneamente. Simplemente tenemos que observar el código que ejecuta el personal del muelle, antes de coger un torito todos ejecutan una operación *P* sobre el semáforo asociado a la sección crítica *sc* y el semáforo se ha inicializado a 1.

En el apartado 2 la respuesta es afirmativa, varias personas del almacén pueden estar utilizando toritos al mismo tiempo. Si observamos el código correspondiente al almacén, podemos observar que tan sólo el primer señor del almacén que quiere coger un torito ejecuta la operación *P* sobre el semáforo de la sección crítica, mientras que el resto no realizarán la citada operación mientras haya alguna persona del almacén utilizando los toritos. Es la técnica utilizada para permitir que varios procesos del mismo tipo estén simultáneamente en la sección crítica.

No ocurre lo mismo en el apartado 3, no pueden haber dos personas de los muelles usando toritos al mismo tiempo. Todas las personas del muelle tienen que ejecutar obligatoriamente la operación *P* sobre el semáforo de la sección crítica inicializado a 1, por lo que sólo podrán coger un torito en un momento dado.

Finalmente, indicar que sí existe prioridad a favor en este caso del personal del almacén. Observemos que la prioridad está implícita en el código y no existe ningún semáforo asociado al tratamiento de ésta. La prioridad se puede eliminar simplemente añadiendo un nuevo semáforo para el tratamiento de la prioridad y utilizándolo de la misma manera que se usó en el problema de lectores y escritores sin prioridad.

```

TSemáforo sc, mutex, fifo;
int numero;

void muelle_i()
{
    . .
    P(fifo);
    P(mutex);
    numero++;
    if (numero==1) P(sc);
    V(mutex);
    V(fifo);
    coger_toriro();
    P(mutex);
    numero--;
    if (numero==0) V(sc);
    V(mutex)
}

void main()
{
    inicializar(mutex, 1);
    inicializar(sc, 1);
    inicializar(fifo, 1);
    numero=0;
    cobegin
        muelle_i(); ...; muelle_n(); almacen_i(); ... ; almacen_m();
    coend;
}

```

#### EJERCICIO 8

En la entrada de un hotel existe una enorme puerta giratoria para la entrada y salida de los clientes, pero la citada puerta tiene unas normas de utilización muy estrictas. Los clientes del hotel pueden ser señoritas o caballeros, pero debido al puritanismo de la dirección del hotel, no se permite el paso simultáneo de ambos sexos. La puerta permite el paso de varias señoritas simultáneamente, y lo mismo ocurre con los caballeros. Ahora bien, en caso de querer utilizar la puerta clientes de ambos sexos, los caballeros harán honor a su nombre y cederán el paso a las señoritas.

Implementar una solución que controle la puerta ajustándose a las restricciones del enunciado.

#### Solución

En este ejercicio tenemos dos tipos de procesos y tanto señoritas como caballeros pueden pasar varios la puerta simultáneamente, además existe prioridad de las señoritas sobre los caballeros. Analizando el enunciado, vemos que el problema es similar al clásico lectores-escritores con prioridad escritores, excepto por la diferencia de que en este caso, al tipo de procesos que tiene prioridad (señoritas) también se le permite que pasen muchos por la puerta al mismo tiempo (recordemos que escritores sólo podía haber uno en un momento dado).

Los cuatro semáforos utilizados en la solución propuesta tienen la siguiente función:

- *sc* es el semáforo utilizado para tratar el problema de la sección crítica
- *mutex1* es el semáforo asociado a la variable *señoritas* para su acceso en exclusión mutua
- *mutex2* tiene la misma funcionalidad que *mutex1*, pero en este caso asociado a la variable *caballeros*
- *r* nos permitirá implementar la prioridad de las señoritas sobre los caballeros

Analicemos el caso de las señoritas, por definición del problema podemos tener varias señoritas pasando por la puerta del hotel simultáneamente. Como se ha estudiado anteriormente en el problema de lectores y escritores, esta restricción se consigue eliminando la obligación de que todas las señoritas ejecuten *P(sc)*. La idea es que la primera señora (*señoritas==1*) realice la operación *P(sc)*, y mientras ésta señora esté pasando, cualquier otra que llegue pueda pasar por la puerta sin ejecutar la citada operación *P* y, por tanto, sin posibilidad de bloquearse.

Necesitamos saber cual es la primera y la última señora, y para ello, añadimos la variable *señoritas* que me indicará el número de señoritas que están pasando por la puerta en un momento dado. Debido a que esta variable se convierte en una nueva sección crítica, añadimos el semáforo *mutex1* para garantizar su acceso en exclusión mutua entre las señoritas. La implementación de los caballeros es exactamente la misma excepto por la diferencia de los nombres de variable y semáforo.

Llegados a este punto, simplemente nos falta garantizar que las señoritas tengan prioridad de paso por la puerta. Para tratar el aspecto de la prioridad añadimos un nuevo semáforo *r*, y seguiremos la misma técnica analizada en el clásico problema de lectores y escritores con prioridad escritores.

Para dar prioridad a las señoritas, sólo la primera señora llevará a cabo la operación *P(r)*, y mientras la primera señora esté pasando por la puerta

(sección crítica), cualquier otra que llegue pasará sin ejecutar la citada operación  $P$  en el semáforo  $r$  y, por tanto, sin posibilidad de bloquearse en el semáforo de tratamiento de la prioridad. Por el contrario, todos los caballeros tendrán la obligación de ejecutar la operación  $P(r)$ , y no en cualquier lugar, sino que será la primera instrucción a ejecutar cuando llegan al sistema. Además, la última operación del protocolo de entrada que realizarán los caballeros es  $V(r)$ , para habilitar que mientras unos caballeros estén pasando, otros puedan acceder y no se queden bloqueados en la primera instrucción.

```
TSemáforo sc, mutex1, mutex2, r;
int señoritas, caballeros;
```

<pre>void señora_1() {     . .     P(mutex1);     señoritas++;     if (señoritas==1)         {P(r), P(sc)};     V(mutex1);     pasar_puerta();     P(mutex1);     señoritas--;     if (señoritas==0)         {V(r), V(sc)};     V(mutex1)     . }</pre>	<pre>void caballero_() {     .     P(r);     P(mutex2);     caballeros++;     if (caballeros==1)         P(sc);     V(mutex2);     V(r);     pasar_puerta();     P(mutex2);     señoritas--;     if (señoritas==0)         V(sc);     V(mutex2)     . }</pre>
---	---

```
void main()
{
    inicializar(mutex1, 1);
    inicializar(mutex2, 1);
    inicializar(sc, 1);
    inicializar(r, 1);
    señoritas=0;
    caballeros=0;
    cobegin
        señora_1(); ...; señora_n(); caballero_();
        ; caballero_m();
    coend;
}
```

### EJERCICIO 9

Cuando se realiza el mantenimiento de las carreteras secundarias (aquellas que únicamente tienen un carril en cada sentido de circulación), los operarios suelen cortar el carril en el que están realizando el mantenimiento y gestionar el tráfico de ambos sentidos sólo por el carril que queda libre.

Para la gestión del tráfico se sitúan dos operarios, uno a cada lado de las obras de mantenimiento, que sólo permiten tráfico en un solo sentido en un momento dado. La empresa ha decidido ahorrarse dos puestos de trabajo y quiere implementar el sistema de control de este tipo de situaciones mediante semáforos. No existe prioridad en ninguno de los sentidos.

### Solución

Este es un ejercicio muy típico, que en la bibliografía se puede encontrar como dos caminos que se cruzan en un puente, dos sentidos de circulación que en algún momento tienen un túnel por el que sólo se puede pasar en un sentido, etc.

De nuevo necesitamos cuatro semáforos para solucionar el problema, semáforos que tienen las funciones siguientes:

Los cuatro semáforos utilizados en la solución propuesta tienen la siguiente función:

- $sc$  es el semáforo utilizado para tratar el problema de la sección crítica
- $mutex1$  es el semáforo asociado a la variable  $contadorA$  para su acceso en exclusión mutua
- $mutex2$  tiene la misma funcionalidad que  $mutex1$ , pero en este caso asociado a la variable  $contadorB$
- $r$  nos permitirá implementar el aspecto de la prioridad, aunque recordemos que en este caso no existe prioridad, o lo que es lo mismo, prioridad por orden de llegada

De nuevo, es una buena estrategia para solucionar este tipo de problemas es afrontar las restricciones una a una; primero que puedan pasar varios automóviles del sentido A, segundo que pasen varios automóviles del sentido B y, finalmente, el aspecto de la no existencia de prioridad.

Solucionar el problema de que cuando tengan su turno, puedan pasar varios automóviles del mismo sentido se ha solucionado en ejercicios anteriores, por lo que no se aborda aquí en profundidad. Añadimos variables para saber la cantidad de automóviles que pasan en cada sentido, añadimos semáforos para acceder en exclusión mutua a éstas variables y utilizamos la

técnica de que sólo el primer automóvil realice la operación *P* sobre el semáforo de sección crítica *sc*.

Al igual que en el ejercicio anterior, en este momento ya hemos implementado dos de las tres restricciones que tiene el problema, simplemente nos falta garantizar que ambos sentidos de circulación tengan la misma prioridad (sin prioridad o por orden de llegada). Para tratar el aspecto de la prioridad añadimos un nuevo semáforo *r*, y seguiremos la misma técnica analizada en el clásico problema de lectores y escritores con prioridad según el orden de llegada.

Para garantizar la misma prioridad, tanto los automóviles que circulan en un sentido como los que circulan en el contrario ejecutarán como primera instrucción de su protocolo de entrada la operación *P* sobre el semáforo que gestiona la prioridad *r* y, como última instrucción del protocolo de entrada la operación *V* sobre el mismo semáforo.

Observemos que la solución es muy estricta y poco eficiente, cuando existen varios automóviles en cada sentido, lo lógico sería que pasaran en cada sentido por grupos, pero el algoritmo sigue un riguroso orden de llegada. Los dos operarios habrían tenido más sentido común.

```
TSemáforo sc, mutex1, mutex2, r;
int contadorA, contadorB;
```

```
void sentidoA_i()
{
    .
    .
    P(r);
    P(mutex1);
    contadorA++;
    if (contadorA==1) P(sc)
    V(mutex1);
    V(r);
    circular();
    P(mutex1);
    contadorA--;
    if (contadorA==0)
    V(sc);
    V(mutex1)
    .
}
```

```
void sentidoB_i()
{
    .
    .
    P(r);
    P(mutex2);
    contadorB++;
    if (contadorB==1)
    P(sc);
    V(mutex2);
    V(r);
    circular();
    P(mutex2);
    contadorB--;
    if (contadorB==0)
    V(sc);
    V(mutex2)
    .
}
```

```
void main()
{
    inicializar(mutex1, 1); inicializar(mutex2, 1);
    inicializar(sc, 1); inicializar(r, 1);
    contadorA=0;
```

```
contadorB=0;
cobegin
    sentidoA1() ; ... ; sentidoAn() ; sentidoB1() ; ... ; sentidoBm();
    coend;
}
```

#### EJERCICIO 10

Tenemos un cruce de calles en una ciudad. En una de las calles circulan coches y la calle tiene varios carriles por lo que pueden pasar varios coches simultáneamente. La otra calle sólo tiene un carril por el que pasa un tranvía que tiene prioridad sobre los coches.

#### Solución

Este ejercicio es una muestra de que la problemática asociada a la concurrencia aparece en multitud de campos de aplicación. Si observamos con detenimiento el enunciado, tenemos dos tipos de procesos, de un tipo sólo puede pasar uno, del otro pueden pasar varios al mismo tiempo y tiene prioridad el que sólo puede pasar uno. Llevando a cabo una pequeña abstracción, e incluso más que una abstracción una traslación, este ejercicio es idéntico al clásico problema de lectores-escritores con prioridad escritores.

El lector puede encontrar la explicación del código en el ejercicio referenciado, simplemente, tenga en cuenta que es una buena práctica solucionar el problema abordando una restricción detrás de otra.

Primero abordar la restricción de que sólo pase un tranvía, en este sentido es el mismo caso que los escritores. Segundo, implementar la restricción de que puedan pasar varios coches (no importa el número de carriles), la traslación nos llevará a proponer un código como el de los lectores.

Finalmente, abordaremos el problema de la prioridad del tranvía, que implicará el uso de una nueva variable, semáforo para su acceso en exclusión mutua, y una implementación según la técnica desarrollada en el ejercicio de lectores-escritores con prioridad escritores.

```

TSemáforo mutex1, mutex2, sc, r;
int coches, numtran;

void coche_i()
{
    . .
    P(r);
    P(mutex1);
    coches++;
    if (coches==1) P(sc);
    V(mutex1);
    V(r);
    circular();
    P(mutex1);
    coches--;
    if (coches==0) V(sc);
    V(mutex1);
}

void tranvía_j()
{
    . .
    P(mutex2);
    numtran++;
    if (numtran==1) P(r);
    V(mutex2);
    P(sc);
    circular();
    V(sc);
    P(mutex2);
    numtran--;
    if (numtran==0) V(r);
    V(mutex2);
}

void main()
{
    inicializar(mutex1, 1); inicializar(mutex2, 1);
    inicializar(sc, 1); inicializar(r, 1);
    coches = 0; numtran = 0;
    cobegin
        coche_i(); ...; coche_n(); tranvía_i(); ... ; tranvía_m();
    coend;
}

```

### EJERCICIO 11

Supongamos un agricultor que tiene una parcela en la que cultiva pepinos y melones, cuando es la época de la recogida el agricultor va a la parcela, si tiene pepinos podrá coger varios pepinos simultáneamente, si tiene melones podrá coger de uno en uno, y si tiene que elegir entre coger melones o pepinos recogerá primero los pepinos. Implementar el código asociado a los tipos de procesos pepinos y melones.

### Solución

De nuevo otro ejemplo de que la problemática asociada a la concurrencia tiene multitud de campos de aplicación. Si observamos con detenimiento el enunciado, tenemos dos tipos de procesos, de un tipo sólo se puede recoger

uno, del otro pueden recogerse varios al mismo tiempo y tienen prioridad los que pueden cogerse varios.

Llevando a cabo una pequeña abstracción, este ejercicio es idéntico al clásico problema de lectores-escritores con prioridad lectores.

Observemos que en éste caso la prioridad está implícita en el algoritmo y no se añade un semáforo adicional para su tratamiento.

Supongamos la secuencia, pepino, melón, pepino. El primer pepino entraña en la sección crítica, el melón se quedaría bloqueado en la instrucción *P(sc)*, el semáforo *sc* lo ha cerrado el primer pepino.

Finalmente, el segundo pepino, incrementa la variable *pepinos* (*pepinos=2*), y pasa a la sección crítica directamente. Como hemos comprobado, el segundo pepino entra en la sección crítica antes que el melón, pese a haber llegado después, consideraremos que los pepinos tienen prioridad.

```

TSemáforo mutex, sc;
int pepinos;

```

```

void pepino_i()
{
    . .
    P(mutex);
    pepinos++;
    if (pepinos==1) P(sc);
    V(mutex);
    recoger();
    V(sc);
    . .
}

void melón_i()
{
    . .
    P(sc);
    recoger();
    V(sc);
    . .
}

```

```

void main()
{
    inicializar(mutex, 1);
    inicializar(sc, 1);
    pepinos=0;
    cobegin
        melón_i(); ...; melón_n();
        pepino_i(); ... ; pepino_m();
    coend;
}

```

## EJERCICIO 12

Supongamos que existe en la ciudad de Madrid un autobús turístico “Madrid visión”, pero que en vez de realizar una ruta pasando por todos los lugares importantes, realiza un desplazamiento punto a punto entre el Palacio Real y el Museo del Prado, y viceversa. El autobús únicamente se pone en marcha cuando tiene un número suficiente de turistas, concretamente 30. Los turistas del Palacio Real invocarán el procedimiento Ir\_al\_Prado, mientras que los turistas del Museo del Prado invocarán el procedimiento Ir\_al\_Palacio. Implementar los dos procedimientos, considerando que inicialmente el autobús se encuentra parado en el museo y está vacío.

### Solución

Este problema es sobretodo un problema de sincronización, y tiene varios aspectos muy importantes que vamos a abordar a continuación.

El primer aspecto importante es el tratamiento de la alternancia, es decir, el autobús o está en el palacio o en el museo, pero no puede estar en los dos sitios al mismo tiempo. Como en algunos de los problemas analizados anteriormente, el tratamiento de la alternancia también tiene su técnica.

Para implementar el concepto de alternancia se utilizan dos semáforos, *museo* y *palacio* en este caso, cada semáforo está asociado a un tipo de procesos. La técnica consiste en que cada tipo de proceso ejecuta, al inicio de su ejecución, una operación *P* sobre su semáforo asociado, por ejemplo *P(museo)* en el caso de los turistas que están en el museo y quieren ir al palacio. Luego, al final de su ejecución, cada tipo de proceso realizará una operación *V* sobre el semáforo asociado al otro tipo de procesos, por ejemplo *V(palacio)* en el caso de los turistas que están en el museo. Esta última operación *V* habilita que puedan progresar los procesos del otro tipo.

Como en cualquier problema de alternancia, se debe considerar una alternativa inicial, en este caso y según el enunciado, el autobús está inicialmente parado en el museo, por lo que sólo podrán subir turistas que se encuentren en dicho lugar. Para conseguir tal fin, tenemos que recurrir a la inicialización de los semáforos. El semáforo *museo* (situación inicial) se ha inicializado a 1, para permitir que si llegan turistas que están en el museo puedan subir al autobús. Por el contrario, el semáforo *palacio* se ha inicializado a 0, para no permitir que los turistas del palacio progresen, dado que el autobús no se encuentra allí.

En este ejercicio no todos los procesos ejecutan el procedimiento *arrancar*, sino que es sólo el turista número 30 el que activa el evento arrancar, simulando el traslado de un lugar turístico a otro, el resto de turistas se quedarán bloqueados en el semáforo *sc* que se ha inicializado a 0. Por supuesto, aunque sólo el turista número 30 ejecuta el procedimiento se asume que viajan todos.

```
TSemáforo mutex, sc, palacio, museo;
int turistas;
```

<pre>void Ir_al_Palacio_i() {     . .     P(museo);     P(mutex);     turistas++;     if (turistas&lt;30) {         V(museo);         V(mutex);         P(sc);     }     else {         V(mutex);         arrancar();     }     P(mutex);     turistas--;     if (turistas==0)         V(palacio);     else         V(sc);     V(mutex);     . }</pre>	<pre>void Ir_al_Prado_j() {     . .     P(palacio);     P(mutex);     turistas++;     if (turistas&lt;30) {         V(palacio);         V(mutex);         P(sc);     }     else {         V(mutex);         arrancar();     }     P(mutex);     turistas--;     if (turistas==0)         V(museo);     else         V(sc);     V(mutex);     . }</pre>
--	--

```
void main()
{
    inicializar(mutex, 1); inicializar(palacio, 0);
    inicializar(museo, 1); inicializar(sc, 0);
    turistas=0;
    cobegin
        Ir_al_Prado_1(); ...; Ir_al_Prado_n();
        Ir_al_Palacio_1(); ...; Ir_al_Palacio_m();
    coend;
}
```

### EJERCICIO 13

En medio de la jungla existe un puente en altura que cruza un río, el puente es muy estrecho y tan sólo puede pasar una persona en un instante dado. La orientación del puente es norte-sur, por lo que las personas pueden pasar en sentido norte-sur y en sentido sur-norte. Debido a que todos los que llegan al puente en medio de la selva están cansados y quieren cruzar el puente tan pronto como sea posible, no existe prioridad y se respeta un escrupuloso orden de llegada.

#### Solución

Este es un ejercicio sencillo en el que se presentan dos de las tres restricciones más estudiadas en asignaturas fundamentales de sistemas operativos. Primera, la restricción de que únicamente puede pasar una persona y, segunda, el tratamiento de la prioridad, que en este caso concreto es prioridad por orden de llegada (sin prioridad o idéntica prioridad). La restricción de que sólo pueda pasar una persona se cumple haciendo que todos ejecuten obligatoriamente la operación *P* sobre el semáforo *sc*.

Podemos observar la prioridad por orden de llegada con el estudio del semáforo *fifo*. En este caso la primera instrucción del protocolo de entrada es *P(fifo)* y la última del mismo protocolo es *V(fifo)*, tal como se ha visto en problemas anteriores.

*TSemáforo fifo, sc;*

<pre>void norte_sur_i() {     . . .     P(fifo);     P(sc);     V(fifo);     cruzar();     V(sc); } . . .</pre>	<pre>void sur_norte_j() {     . . .     P(fifo);     P(sc);     V(fifo);     cruzar();     V(sc);     . . . }</pre>
---	---

```
void main()
{
    inicializar(fifo, 1);
    inicializar(sc, 1);
```

```
cobegin
    norte_sur_1(); ...; norte_sur_n();
    sur_norte_1(); ...; sur_norte_m();
coend;
}
```

### EJERCICIO 14

En un avión de largo recorrido la cocina está en el habitáculo inferior del avión, mientras que los pasajeros viajan en el habitáculo superior. Existe un pequeño montacargas que permite subir bandejas llenas de comida y bajar las bandejas vacías. Para no estar siempre subiendo y bajando el montacargas, la política de la compañía es que el montacargas se subirá cuando tenga diez bandejas de comida y se bajará cuando tenga diez vacías. El montacargas se encuentra inicialmente en el habitáculo inferior para iniciar el servicio de comidas.

#### Solución

De nuevo estamos ante un ejercicio fundamentalmente de sincronización, y más concretamente de alternancia (el montacargas no puede estar simultáneamente en ambas plantas).

Como se ha estudiando anteriormente, para la alternancia se utilizan dos semáforos, *planta0* y *planta1*, cada semáforo está asociado a una de las plantas. Cada tipo de proceso ejecuta, al inicio de su ejecución, una operación *P* sobre su semáforo asociado, por ejemplo *P(planta0)* en el caso del habitáculo inferior. Luego, al final de su ejecución, cada tipo de proceso realizará una operación *V* sobre el semáforo asociado al otro tipo de procesos, por ejemplo *V(planta1)* en el caso del habitáculo inferior. Ésta última operación *V* habilita que puedan progresar los procesos del otro tipo.

Según la definición del problema, el pequeño montacargas está inicialmente en el habitáculo inferior, por lo que sólo se podrán poner bandejas llenas. De nuevo, la idea es trabajar con la inicialización de los semáforos. El semáforo *planta0* (situación inicial) se ha inicializado a 1, para permitir que se pongan bandejas llenas. En cambio, el semáforo *planta1* se ha inicializado a 0, para no permitir que las azafatas del habitáculo superior puedan poner bandejas vacías en el montacargas.

Podemos observar que las nueve primeras bandejas se quedarán bloqueadas en el semáforo de la sección crítica *sc*, dado que éste se ha inicializado a 0. Por el contrario, la bandeja número 10 no se quedará bloqueada y ejecuta-

rá la operación de *subir* o *bajar*, según el caso.

Después, al llegar a la planta contraria se irán quitando bandejas, cada una de las cuales ejecutará *V(sc)* y permitirá desbloquear a otra. Finalmente, la última bandeja realizará la operación *V* sobre el semáforo asociado a la planta contraria, lo que permitirá que empiece a ejecutarse el código asociado a la otra planta.

Como en un ejemplo anterior, las dos plantas utilizan la misma variable para llevar la cuenta, pero no ocurre nada porque en un momento dado sólo se estará ejecutando el código asociado a una de las plantas, en la otra planta los procesos se pararán en la primera operación *P*.

Finalmente, el lector debe observar que igual que sólo utilizamos una variable, sólo utilizamos un semáforo asociado a ésta, *mutex*. Éste semáforo nos garantiza el acceso en exclusión mutua a la variable, recordemos que el hecho de añadir una variable a la que acceden varios procesos la convierte en sección crítica.

```
TSemáforo mutex, sc, planta1, planta0;
int bandejas;
```

```
void Subir_i()
{
    .
    .
    P(planta0);
    P(mutex);
    bandejas++;
    if (bandejas<10) {
        V(planta0);
        V(mutex);
        P(sc);
    }
    else {
        V(mutex);
        subir();
    }
    P(mutex);
    bandejas--;
    if (bandejas==0)
        V(planta1);
    else
        V(sc);
    V(mutex);
    .
}
```

```
void Bajar_j()
{
    .
    .
    P(planta1);
    P(mutex);
    bandejas++;
    if (bandejas<10) {
        V(planta1);
        V(mutex);
        P(sc);
    }
    else {
        V(mutex);
        bajar();
    }
    P(mutex);
    bandejas--;
    if (bandejas==0)
        V(planta0);
    else
        V(sc);
    V(mutex);
    .
}
```

```
void main()
```

```
inicializar(mutex, 1);
inicializar(planta1, 0);
inicializar(planta0, 1);
inicializar(sc, 0);
bandejas=0;
cobegin
    Subir1()...; Subirn() ; Bajar1() ; ... ; Bajarm();
coend;
}
```

### EJERCICIO 15

Se está acercando la hora de cierre del bar, y el camarero Carlos va recogiendo las mesas y sillas que los clientes van dejando libres. Se pueden recoger varias sillas al mismo tiempo, ya que se pueden amontonar y pesan menos. En cambio, las mesas se tienen que recoger de una en una, es decir, en un momento dado sólo podrá llevar una mesa. Si en algún momento tiene mesas y sillas, optará por recoger primero las mesas. Implementar el código asociado a las mesas y a las sillas.

### Solución

De nuevo un ejemplo que aborda las tres restricciones que más se estudian en clase de teoría. La restricción de que sólo pase un proceso por la sección crítica (mesas), que pasen varios procesos al mismo tiempo por la sección crítica (sillas), y que un tipo de procesos tenga prioridad sobre el otro (mesas tienen prioridad sobre las sillas).

Todas las mesas ejecutan *P(sc)*, mientras que las sillas sólo la primera realiza la citada operación. Son las técnicas que hemos estudiado para las restricciones “sólo un proceso” y “varios” respectivamente.

Para la prioridad se añade un semáforo, el semáforo *r*, y tiene el mismo tratamiento que en ejercicios anteriores.

```

TSemáforo mutex1, mutex2, sc, r;
int sillitas, mesas;

void sillita_1()
{
    . .
    P(r);
    P(mutex1);
    sillitas++;
    if (sillitas==1) P(sc);
    V(mutex1);
    V(r);
    recoger();
    P(mutex1);
    sillitas--;
    if (sillitas==0) V(sc);
    V(mutex1);
    . .
}

void mesa_1()
{
    . .
    P(mutex2);
    mesas++;
    if (mesas==1) P(r);
    V(mutex2);
    P(sc);
    recoger();
    V(sc);
    P(mutex2);
    mesas--;
    if (mesas==0) V(r);
    V(mutex2);
    . .
}

void main()
{
    inicializar(mutex1, 1); inicializar(mutex2, 1);
    inicializar(sc, 1); inicializar(r, 1);
    sillitas = 0; mesas = 0;
    cobegin
        mesa_1(); ...; mesa_n(); sillita_1(); ... ; sillita_m();
    coend;
}

```

#### EJERCICIO 16

En la época de la recogida de la oliva para la elaboración del aceite, los agricultores llevan los sacos llenos de oliva a la cooperativa para su pesado y entrega. Dado que existen distintos tipos de oliva que luego generan aceite de distinta calidad, el pesado se realiza por tipos, y se suele distinguir entre "blanqueta" y "gordal" (en este último se suelen incluir otros tipos menos comunes como "negreta", etc). Es evidente que tanto en la blanqueta como en la gordal se pueden pesar varios sacos simultáneamente. Finalmente, debemos tener en cuenta que no existe ningún tipo de prioridad, por lo que utilizaremos prioridad por orden de llegada. Implementemos una solución que ayude al operario de la báscula.

#### Solución

Necesitamos cuatro semáforos y dos variables para solucionar el problema, semáforos y variables que tienen las funciones siguientes:

- *sc* es el semáforo utilizado para tratar el problema de la sección crítica
- *mutex1* es el semáforo asociado a la variable *numBlanq* para su acceso en exclusión mútua
- *mutex2* tiene la misma funcionalidad que *mutex1*, pero en este caso asociado a la variable *numGordal*
- *fifo* nos permitirá implementar el aspecto de la prioridad, aunque recordemos que en este caso no existe prioridad, o lo que es lo mismo, prioridad por orden de llegada
- *numBlanq* nos indicará en todo momento el número de sacos de blanqueta que tiene la báscula
- *numGordal* nos indicará el número de sacos de gordal que tiene la báscula

Como en la mayoría de ejercicios, la estrategia para solucionar este tipo de problemas es afrontar las restricciones una a una; primero que se puedan pesar varios sacos de blanqueta al mismo tiempo, segundo que se pesen varios sacos de gordal simultáneamente y, finalmente, que el pesado sea por orden de llegada sin existir prioridad.

Para permitir que se puedan pesar varios sacos del mismo tipo, añadimos variables para saber el número de sacos de cada tipo que subimos a la báscula, añadimos semáforos para acceder en exclusión mutua a éstas variables y utilizamos la técnica de que sólo el primer saco de cada tipo realice la operación *P* sobre el semáforo de sección crítica *sc*. Mientras el primer saco esté en la báscula, el resto de sacos que pongamos en la báscula no llevarán a cabo la citada operación *P*.

Llegados a este punto, ya hemos implementado dos de las tres restricciones que tiene el problema, simplemente nos falta garantizar que ambos tipos de oliva tengan la misma prioridad. Para tratar el aspecto de la prioridad añadimos un nuevo semáforo *fifo*, y seguiremos la misma técnica analizada en el clásico problema de lectores y escritores con prioridad según el orden de llegada.

Para garantizar la misma prioridad, tanto los sacos de blanqueta como los de gordal ejecutarán como primera instrucción de su protocolo de entrada la operación *P* sobre el semáforo que gestiona la prioridad *fifo* y, como última instrucción del protocolo de entrada la operación *V* sobre el mismo semáforo.

```
TSemáforo sc, mutex1, mutex2, fifo;
int numBlanq, numGordal;
```

<pre>void blanqueta_i() {     . .     P(fifo);     P(mutex1);     numBlanq++;     if (numBlanq==1) P(sc);     V(mutex1);     V(fifo);     pesar();     P(mutex1);     numBlanq--;     if (numBlanq==0) V(sc);     V(mutex1)     . }</pre>	<pre>void gordal_j() {     .     P(fifo);     P(mutex2);     numGordal++;     if (numGordal==1)         P(sc);     V(mutex2);     V(fifo);     pesar();     P(mutex2);     numGordal--;     if (numGordal==0)         V(sc);     V(mutex2)     . }</pre>
---	--

```
void main()
{
    inicializar(mutex1, 1); inicializar(mutex2, 1);
    inicializar(sc, 1); inicializar(fifo, 1);
    numBlanq=0; numGordal=0;
    cobegin
        blanqueta_i(); ...; blanqueta_n();
        gordal_j(); ...; gordal_m();
    coend;
}
```

#### EJERCICIO 17

El ferry de Estambul permite a los coches pasar de oriente a occidente pero no viceversa. Este ferry tiene una capacidad para 20 coches y espera a estar lleno para cruzar el estrecho de Constantinopla, dejando los coches y volviendo a oriente vacío. Realizar un programa con semáforos que cumpla las exigencias descritas.

#### Solución

```
#define max_ferry 20
TSemáforo ferry, coche;
int coches;

void coche_i()
{
    P(mutex);
    if (coches<max_ferry)
    {
        coches++;
        V(coche);
        V(mutex);
        P(ferry);
        Pasar_coche_a_ferry();
    }
    else V(mutex);

void ferry()
while (true)
{
    P(coche);
    P(mutex);
    if (coches==max_ferry)
    {
        Cruzar_estrecho()
        coches=0;
    }
    V(ferry);
    V(mutex);
    Subir_coche_a_ferry();
}
```

#### EJERCICIO 18

Un bar dispone únicamente de un servicio con lo que se debe de utilizar éste indistintamente para hombres y mujeres. Se ha adoptado por “caballerosidad” el criterio de que las mujeres tengan prioridad de acceso en caso de que quieran acceder al mismo tiempo dos personas de distinto sexo. Evidentemente, esta situación sólo se debe de adoptar en el caso de acceso a la vez, en otro caso siempre hay que esperar que termine la persona que hay dentro excepto que una mujer esté dentro permitiendo entonces que otra entre a la vez (es conocida la tendencia a acceder en pareja por ellas). Comentar

también la solución en el caso de que no se diera esta última circunstancia.

Gestionar el funcionamiento del servicio mediante el uso de semáforos.

#### Solución

```
#define MUJERES 2;
Tsemáforo h, s;
int mujeres;

void hombre_i()
{
    . .
    P(s);
    if (mujeres<MUJERES)
    {
        mujeres++;
        if (mujeres==1) P(h);
        V(s)
        Entrar_al_servicio();
        P(s);
        mujeres--;
        if (mujeres==0) V(w);
        V(s)
    }
    else V(s);
    .
}

void main()
{
    inicializar(s, 1);inicializar(h, 1);
    mujeres=0;
    cobegin
        hombre_i();...; hombre_i(); mujer_i(); ... ; mujer_i();
    coend;
}
```

La modificación necesaria para el caso en el que se quiera una solución con una sola persona a la vez en el servicio se puede efectuar cambiando la constante de MUJERES por 1 en lugar de 2

#### EJERCICIO 19

Se pretende utilizar semáforos para sincronizar cuatro procesos (P1, P2, P3 y P4) con el siguiente orden de ejecución P1, P3, P4 y P2. Así, primero se ejecuta P1 y cuando finaliza P1 se puede ejecutar P3, cuando finaliza P3 se puede ejecutar P4, cuando finaliza P4 se puede ejecutar P2 y cuando finaliza P2 se puede ejecutar P1.

#### Solución

<pre>void P1 () {     while (true)     {         P(P2);         Operaciones_P1;         V(P1);     } }</pre>	<pre>void P2 () {     while (true)     {         P(P4);         Operaciones_P2;         V(P2);     } }</pre>
<pre>void P3 () {     while (true)     {         P(P1);         Operaciones_P3;         V(P3);     } }</pre>	<pre>void P4 () {     while (true)     {         P(P3);         Operaciones_P4;         V(P4);     } }</pre>

#### EJERCICIO 20

En una pequeña empresa de dispone de 1GB de ancho de banda (BW) para la conexión a Internet. Se disponer de los servicios básicos de web y correo electrónico. El correo electrónico es prioritario sobre la navegación web dando a estos dos tipos de procesos 100KB y 50KB respectivamente según van siendo solicitados por los empleados.

Gestionar la conexión de Internet de esta empresa con las especificaciones dadas y mediante el uso de semáforos.

**Caso 1**

Se supone un BW estático de 1GB. Además debe de existir un servidor de internet con la capacidad de gestión para ofrecer los recursos de que dispone y liberarlos una vez que no se necesiten para el posterior uso por otro empleado.

**Solución**

Por una parte necesitamos uno para crear u ofrecer los servicios de internet. Este lo ofrece el servidor que ofrece los servicios que piden los empleados con los semáforos donde la l indica los espacios liberados o huecos que se van disponiendo en el BW general decrementandolos sucesivamente según va disponiendo de ellos. Así mismo, incrementa los semáforos donde la c indica la creación de esos recursos. Los procesos empleados hacen, evidentemente, las operaciones inversas según van demandando recursos y liberándolos.

```
#define BW 1048576 (Bytes);
Tsemáforo s, bw_w_c, bw_m_c, bw_w_l, bw_m_l;

void empleado_web()
{
    while (true)
    {
        P(bw_w_c);
        P(s);
        Coger_BW();
        V(s);
        V(bw_w_l);
        Navegar_en_internet();
    }
}

void empleado_mail()
{
    while (true)
    {
        P(bw_m_c);
        P(s);
        Coger_BW();
        V(s);
        V(bw_m_l);
        Enviar_recibir_mails();
    }
}

void servidor_internet()
{
    while (true)
    {
        Crear_BW();
        P(bw_w_l); /* Necesita huecos, los solicita
        */
        P(bw_m_l);
        P(s);
        Ofrecer_BW();
        V(s);
    }
}
```

```
V(bw_w_c); /* ya ha creado los recursos */
V(bw_m_c);
}

void main()
{
    inicializar(s, 1);
    inicializar(bw_w_c, 0);
    inicializar(bw_m_c, 0);
    inicializar(bw_w_l, 1048576);
    inicializar(bw_m_l, 1048576);
    cobegin
        servidor_internet();
        empleado_web();
        empleado_mail();
    coend;
}
```

**Caso 2**

En muchos casos no se dispone de un conocimiento exacto del BW que va a necesitar la empresa y se pretende disponer de una solución más dinámica para lo que se establece un determinado acuerdo y contrato con la empresa que proporciona la conexión a internet mediante el cual conforme van creciendo las necesidades de la empresa se van ofreciendo un incremento de los recursos de infraestructuras con los que dotar a la empresa (sería conexión más rápida e incremento de BW). La política sería disponer de un proceso que se encuentra en la empresa que proporciona los servicios de internet que incrementa el BW según lo van usando los empleados de la empresa.

**Solución**

Para esta solución la capacidad (en teoría ilimitada) que ofrece un nuevo proceso se incorpora para esta solución empresa\_servicios\_internet.

```
Tsemáforo s, bw_w_c, bw_m_c;
```

<pre>void empleado_web() {     while (true)     {         P(bw_w_c);         P(s);         Coger_BW();         V(s);         Navegar_en_internet();     } }</pre>	<pre>void empleado_mail() {     while (true)     {         P(bw_m_c);         P(s);         Coger_BW();         V(s);         Enviar_recibir_mails();     } }</pre>
---	---

```
void empresa_servicios_internet()
{
    while (true)
    {
        Contratar_nuevo_BW();
        P(s);
        Ampliar_BW();
        V(s);
        V(bw_w_c);
        V(bw_m_c);
    }
}

void main()
{
    inicializar(s, 1);
    inicializar(bw_w_c, 0);
    inicializar(bw_m_c, 0);
    cobegin
        empresa_servicios_internet();
        empleado_web();
        empleado_mail();
    coend;
}
```

Para cualquiera de estas dos soluciones hay que hacer una aclaración con las primitivas P y V. En principio y para nuestro caso deberían efectuar una operación algo distinta a la que es habitual, pues sus operaciones son de decrementar e incrementar una unidad respectivamente. Sin embargo, en nuestro caso necesitamos un incremento y decremento de 50 y 100 para los semáforos bw\_m\_c y bw\_w\_c. En cualquier caso habría que realizar las oportunas modificaciones o al menos aclararlo en la solución.

### EJERCICIO 21

Sean las siguientes operaciones atómicas:

- **dormir:** suspende al proceso que la ejecuta.
- **despertar:** despierta a todos los procesos suspendidos mediante sleep.

Implementa las operaciones anteriores mediante semáforos.

#### Solución

```
TSemaforo mutex, S;
```

##### Dormir:

```
P(mutex);
c=c+1;
V(mutex);
P(S);
P(mutex);
c=c-1;
if (c>0) V(S);
V(mutex)
```

##### Despertar:

```
P(mutex);
if (c>0) V(S);
V(mutex);
```

```
void main()
{
    c=0;
    Inicializar(mutex, 1); Inicializar(S,0);
}
```

### EJERCICIO 22

En un sistema multiprogramado tenemos un fichero compartido por varios procesos de dos tipos diferentes: lectores y escritores. El acceso se implementa como sigue:



TSemáforo mutex1, mutex2, mutex3, mutex4;  
int lectores, escritores;

<pre>void lector_i() {     . .     P(mutex4);     P(mutex1);     lectores++;     if (lectores==1)         P(mutex3);     V(mutex1);     leer();     V(mutex4);     P(mutex1);     lectores--;     if (lectores==0)         V(mutex3);     V(mutex1);     . }</pre>	<pre>void escritor_j() {     . .     P(mutex2);     escritores++;     if (escritores==0)         P(mutex4);     V(mutex2);     P(mutex3);     escribir();     V(mutex3);     P(mutex2);     escritores--;     if (escritores==1)         V(mutex4);     V(mutex2);     . }</pre>
--	--

```
void main()
{
    inicializar(mutex1, 1); inicializar(mutex2, 1);
    inicializar(mutex3, 1); inicializar(mutex4, 1);
    lectores = 0; escritores = 0;
    cobegin
        escritor_i(); escritor_n(); lector_i(); lector_m();
    coend;
}
```

Contesta las siguientes preguntas justificando la respuesta en cada caso:

- ¿Pueden utilizar a la vez el fichero lectores y escritores?
- Si hay un proceso lector utilizando el fichero, ¿puede otro lector utilizarlo al mismo tiempo en todos los casos?. En caso negativo indica qué modificaciones hay que realizar para que sea posible.
- Si hay un proceso escritor utilizando el fichero, ¿puede otro escritor utilizarlo al mismo tiempo en todos los casos?. En caso afirmativo indica qué modificaciones hay que realizar para que no sea posible.
- ¿Hay algún tipo de prioridad para algún proceso?
- ¿Qué diferencias habría en la implementación de una planificación con prioridad a los escritores entre un sistema monoprocesador y multiprocesador?

### Solución

- No pueden, en cualquiera de las trazas que se hagan se comprueba que no es posible. Entre las trazas de interés se encuentran las siguientes: LEL; ELE; LLEL; EELE. En todas ellas se considera que los procesos permanecen en la Sección crítica el tiempo suficiente para que coexistan varios de ellos en ejecución.
- No pueden. El semáforo mutex4 lo impide. La modificación a realizar consiste en eliminar el semáforo mutex4.
- No pueden. El semáforo mutex4 lo impide.
- No hay prioridad. En las trazas se demuestra. Como ejemplo se puede probar la siguiente traza: LLEL. En ésta, se comprueba que el primer escritor puede entrar antes o después del segundo y tercer lector.
- Ninguna, es una cuestión del sistema operativo.

### EJERCICIO 23

Considera la siguiente operación atómica y utilízala para resolver el problema de exclusión mutua.

```
atomic procedure Swap(var a, b: boolean);
var temp: boolean;
begin
    temp:=a;
    a:=b;
    b:=temp;
end;
```

### Solución

```
var
    bloqueo, flag1, flag2, ..., flagn: boolean

Procesoi
begin
    while true do
        begin
            flagi:=true;
repeat
```

```

        swap(bloqueo, flagi);
        until flagi=false;
        /*sección crítica*/
        bloqueo:=false;
    end;

begin
    bloqueo:=false;
    cobegin
        Proceso1, Proceso2, ..., ProcesoN;
    coend;
end.

```

La implementación de la exclusión mutua con esta instrucción puede provocar los problemas de espera activa y de cierre. La espera activa significa que durante la espera de un proceso a entrar en la sección crítica, el procesador permanece ocupado en un bucle. El cierre significa que un mismo proceso puede dejar su sección crítica y volver a entrar antes que otros procesos que estaban esperando.

#### EJERCICIO 24

Sea un sistema con dos procesos: P1 y P2 que se ejecutan de forma concurrente:

P1	P2
{ ... (1) ... }	{ ... (2) ... }

Se desea sincronizar ambos procesos para que P1 ejecute (1) al mismo tiempo que P2 ejecuta (2).

#### Solución

TSemáforo sem1, sem2;

void P1()	void P2()
{ ...; P(sem1); V(sem2); (1) ...; }	{ ...; V(sem1); P(sem2); (2) ...; }

```

void main()
{
    inicializar(sem1,0);
    inicializar(sem2,0);
    cobegin
        P1(); P2();
    coend;
}

```

#### EJERCICIO 25

Se dispone de un puente por el que pasan coches en un solo sentido, por razones de seguridad por el puente sólo se permite el paso de 20 coches simultáneamente.

Se pide programar la barrera de entrada y de salida del puente de modo que se satisfagan las condiciones impuestas. Se debe utilizar semáforos binarios.

#### Solución

Se pide programar únicamente las barreras de entrada y de salida, NO los coches.

Cuando llega un coche al puente se activa el proceso de la barrera de entrada y cuando cruza el puente se activa el de la barrera de salida. El problema es similar al del **Productores y Consumidores** donde hay un único proceso productor, *barrera de entrada*, y un solo proceso consumidor, *barrera de salida*.

```

#define max 20
TSemBin mutex, puente;
int coches=0;

void Barrera_Entrada()
{
    P(mutex);
    if (coches==max)
    {
        V(mutex);
        P(puente);
        P(mutex);
    }
    coches=coches+1;
    V(mutex);
    /*Subir barrera de entrada*/
}

void Barrera_Salida()
{
    P(mutex);
    coches=coches-1;
    V(mutex);
    /*Subir barrera de salida*/
    if (!puente.cola.esvacia())
        V(puente)
}

void main()
{
    inicializar(mutex, 1); inicializar(puente, 0);
    cobegin
        Barrera_Entrada(); Barrera_Salida();
    coend
}

```

## MONITORES

### EJERCICIO 26

Uno o más productores generan cierto tipo de datos y los sitúan en una zona de memoria o buffer. Un único consumidor saca elementos del buffer de uno en uno. El sistema debe impedir la superposición de operaciones sobre el buffer.

#### Solución

En la solución tenemos los dos problemas de sincronización. El consumidor debe bloquearse si intenta coger un elemento y no existe ninguno, y el productor debe bloquearse si después de producir un elemento intenta dejarlo en el buffer y éste está lleno.

En este caso no tenemos problema de sección crítica porque el monitor nos garantiza que nunca existirá más de un proceso en el monitor, en un momento dado sólo podrá haber un proceso ejecutando uno de los procedimientos del monitor.

Vamos a analizar con detenimiento el monitor implementado para la solución del problema, el monitor está formado por seis variables locales, la inicialización y dos procedimientos del monitor. La funcionalidad de cada uno de los elementos anteriores se muestra a continuación:

- *buffer* es el vector que simula el buffer de elementos
- *sigent* es un índice del vector que apuntará en todo momento al lugar donde el productor dejará el elemento producido
- *sigsal* es un índice del vector que apuntará en todo momento al lugar desde donde el consumidor cogerá el elemento
- *contador*, variable importantísima que indicará el número de elementos que tiene el buffer en un momento dado
- *no\_lleno* variable de tipo condition que permitirá bloquear al productor en el caso de que el buffer esté lleno, el productor se bloqueará hasta que el buffer deje de estar lleno o esté *no\_lleno*
- *no\_vacio* variable de tipo condition que permitirá bloquear al consumidor en el caso de que el buffer esté vacío, el consumidor se bloqueará hasta que el buffer deje de estar vacío o esté *no\_vacio*
- *añadir* es el procedimiento del monitor que invocará el productor cuando quiera añadir un elemento al buffer
- *coger* es el procedimiento del monitor que invocará el consumidor.

Finalmente, indicar que los procedimientos *productor* y *consumidor* no son procedimientos del monitor, es el código que ejecutarán los procesos productor y consumidor respectivamente.

Vamos a estudiar la implementación del procedimiento del monitor *añadir*. La primera instrucción del procedimiento lleva a cabo el tratamiento de la sincronización por parte del productor, comprueba si el buffer está lleno (*contador==N*), en cuyo caso ejecuta la operación *espera(no\_lleno)* que bloqueará al productor hasta que sea despertado por el consumidor. Las instrucciones centrales añaden el elemento y actualizan la variable *contador*. Finalmente, la última instrucción *señal(no\_vacio)* intentará despertar a algún consumidor bloqueado por estar el buffer vacío indicando que yo no estoy vacío, si no existe ningún consumidor esperando, la operación no tiene ningún efecto (a diferencia de una *V* sobre un semáforo).

Observemos el procedimiento del monitor *coger*. La primera instrucción del procedimiento lleva a cabo el tratamiento de la sincronización del consumidor, comprueba si el buffer está vacío (*contador==0*), en cuyo caso ejecuta la operación *espera(no\_vacio)* que bloqueará al consumidor. Las

instrucciones centrales quitan el elemento y actualizan la variable *contador*. Finalmente, la última instrucción *señal(no\_lleno)* intentará despertar al productor bloqueado por estar el buffer lleno, si no existe ningún productor esperando, la operación no tiene ningún efecto (a diferencia de una V sobre un semáforo).

```

struct TMonitor
{
    TElementos buffer[N];
    int sigent, sigsal;
    int contador;
    condition no_lleno, no_vacio;
}
sigent=0; sigsal=0; contador=0; }

void añadir(TElemento x)
{
    if (condador==N)
        espera(no_lleno);
    buffer[sigent]=x;
    sigent=(sigent + 1) % N;
    contador++;
    señal(no_vacio)
}

void productor()
{
    TElemento x;
    while (true)
    {
        x=producir();
        añadir(x);
    }
}

void coger(TElemento x)
{
    if (condador==0)
        espera(no_vacio);
    x=buffer[sigsal];
    sigsal=(sigsal + 1) % N;
    contador--;
    señal(no_lleno)
}

void consumidor()
{
    TElemento x;
    while (true)
    {
        coger(x);
        consumir(x);
    }
}
```

```

void main()
{
cobegin
    productor();
    consumidor();
coend;
}
```

### EJERCICIO 27

Se dispone de una zona de memoria o fichero a la que acceden unos procesos (lectores) en modo lectura y otros procesos en modo escritura (escritores).

- Los lectores pueden acceder al fichero de forma concurrente.
- Los escritores deben acceder al fichero de manera exclusiva entre ellos y con los lectores.
- Los escritores tienen prioridad sobre los lectores

El sistema debe coordinar el acceso a la memoria o al fichero para que se cumplan las restricciones.

### Solución

Analicemos detenidamente el monitor implementado y el cumplimiento de las tres restricciones: escritores sólo uno, lectores varios y prioridad de escritores sobre lectores.

La función de las variables utilizadas es la siguiente:

- *lectores* es la variable que nos indicará en todo momento el número de lectores que están leyendo en un momento dado
- *escritores* es la variable que nos permitirá saber si existe un escritor escribiendo en el fichero, observar que sólo puede valer 0 o 1
- *leer* es una variable de tipo condition que permitirá bloquear a los lectores
- *escribir* es una variable de tipo condition que permitirá bloquear a los escritores

Veamos la técnica que nos permite que varios lectores puedan estar simultáneamente accediendo al fichero. Podemos comprobar que los procesos lectores al ejecutar su protocolo de entrada (procedimiento *pre\_leer*) sólo comprueban si hay escritores, pero no si ya existen otros lectores, por lo que podremos tener varios lectores leyendo en el fichero al mismo tiempo. Observemos también el caso de que lleguen varios lectores mientras algún escritor esté escribiendo o en la cola, los lectores se quedarán bloqueados en *espera(leer)*, pero cuando el último escritor permita progresar a los lectores ejecutando *señal(leer)* en el procedimiento *post\_escribir* (protocolo de salida de los escritores), cada lector desbloqueado ejecutará de nuevo *señal(leer)* el final de *pre\_leer* para despertar a otro lector y permitir que varios puedan estar en la sección crítica.

Examinemos ahora la técnica contraria, que sólo pueda haber un escritor en la sección crítica. En este caso los escritores en su protocolo de entrada

(procedimiento *pre\_escribir*) sí que comprueban la existencia de otros escritores, por lo tanto, si ya existe un escritor accediendo al fichero ningún otro escritor podrá pasar y quedará bloqueado en *espera(escribir)*. Observemos que también se bloqueará si existen lectores accediendo al fichero, lógico si tenemos en cuenta que los escritores tienen que acceder en exclusión mutua entre ellos y con los lectores.

```
struct TMonitor
{
    int lectores, escritores;
    condition leer, escribir;
} lectores=0; escritores=0; }

void pre_leer()
{
    if ((escritores>0) || (escribir.n_cola>0))
        espera(leer);
    lectores++;
    señal(leer)
}

void post_leer()
{
    lectores--;
    if (lectores==0) señal(escribir)
}

void pre_escribir()
{
    if ((lectores>0) || (escritores>0))
        espera(escribir);
    escritores++;
}

void post_escribir()
{
    escritores--;
    if (escribir.n_cola>0) señal(escribir);
    else señal(leer);
}
```

<pre>void lector_i() {     ...     pre_leer();     leer();     post_leer();     ... }</pre>	<pre>void escritor_j() {     ...     pre_escribir();     escribir();     post_escribir();     ... }</pre>
---	---

```
void main()
{
    cobegin
        escritor_1(); ... escritor_n();
        lector_1(); ... lector_m();
    coend;
}
```

El aspecto de la prioridad se observa como sigue, los lectores comprueban si hay algún escritor accediendo al fichero (lógico por tener exclusión mutua con los escritores) o si existe algún escritor en la cola (prioridad, para no colarse a ningún escritor). Los escritores sólo comprueban si hay lectores leyendo, no en la cola. Finalmente y lo más importante, los escritores en su protocolo de salida despiertan primero a los escritores, y sólo cuando no hay escritores despiertan a los lectores, por lo tanto, todos los escritores que vayan llegando mientras tengamos algún escritor escribiendo o en la cola pasaran antes que lectores hayan llegado con anterioridad.

Observemos que tenemos cuatro procedimientos del monitor: *pre\_leer*, *post\_leer*, *pre\_escribir*, *post\_escribir*, y los dos procesos lector y escritor. Se puede observar en los procesos la distinción clara de la secuencia protocolo de entrada, sección crítica y protocolo de salida, que coinciden con los procedimientos del monitor.

#### EJERCICIO 28

Cinco filósofos se dedican a pensar y a comer en una mesa circular. En el centro de la mesa hay un cuenco con arroz, y la mesa está puesta con cinco platos y cinco palillos, uno por cada filósofo. Cuando un filósofo tiene hambre se sienta en la mesa a comer en su sitio. El filósofo sólo puede coger un palillo cada vez y no le puede quitar un palillo a un compañero que lo tenga en la mano. Cuando un filósofo tiene los dos palillos come sin soltarlos hasta que termine y vuelve a pensar. El sistema debe coordinar los filósofos para evitar la espera indefinida y que no se mueran de hambre.

*Solución*

```

struct TMonitor
{
    int estado[5]; //0=pensando, 1=hambriento, 2=comiendo
    condition silla[5];
}
for(i=0; i<5; i++) estado[i]=0;

void coge_palillos(int i)
{
    estado[i]=1;
    prueba(i);
    if (estado[i]==1) espera(silla[i]);
}

void deja_palillos(int i)
{
    estado[i]=0;
    prueba((i+1)%5);
    prueba((i+4)%5);
}

void prueba(int i)
{
    if ((estado[(i+1)%5]!=2) && (estado[(i+4)%5]!=2)
        && estado[i] ==1)
    {
        estado[i]=2;
        señal(silla[i]);
    }
}

void filosofo(int i)
{
    while (true)
    {
        pensar();
        coge_palillos(i);
        comer();
        deja_palillos(i);
    }
}

void main()
{
    int i;
    cobegin
        filosofo(0); filosofo(2); ... filosofo(4);
    coend;
}

```

El monitor tiene una variable de *estado* por cada filósofo y una variable *condition silla* por cada filósofo para bloquearlo en el caso de que no obtenga los dos palillos.

Como en el ejercicio anterior, podemos observar claramente, en el código que implementa los procesos de los filósofos, los protocolos de entrada y salida a ejecutar por cada filósofo: *coge\_palillos* y *deja\_palillos*. El monitor tiene tres procedimientos, los dos citados y el procedimiento de apoyo *prueba*.

El procedimiento de apoyo aunque no se puede invocar directamente por los procesos filósofos es el más importante.

Es invocado por el procedimiento *coge\_palillos* para comprobar que los dos filósofos adyacentes no están comiendo y por tanto puede comer el filósofo correspondiente. Si al volver de *prueba* el *estado* sigue siendo 1 (hambriento), significa que no ha podido coger los palillos, ejecutará entonces *espera(silla[i])* y se bloqueará.

Además, también es invocado por el procedimiento *deja\_palillos*, pero en este caso el lector debe observar que la llamada la realiza el filósofo *i* pasando los índices de los filósofos adyacentes, con la esperanza de despertarlos en el caso de que estuviesen bloqueados.

**EJERCICIO 29**

Implementar un monitor que simule un semáforo general

*Solución*

El problema que nos ocupa en estos momentos es un clásico entre los clásicos, pero de una sencillez extrema. Para solucionarlo, sólo debemos conocer la definición de semáforo general, la traslación de la definición a monitores es directa.

El procedimiento *inicializar* lo único que hace es asignar el valor de inicialización a la variable *contador* del semáforo.

El procedimiento *P* ejecuta exactamente la definición de la operación *P* en un semáforo, pero donde el hecho de bloquear al proceso que la invoca en la cola del semáforo se simula bloqueándolo en la cola de la variable *condition cola, espera(cola)*.

El procedimiento *V* ejecuta exactamente la definición de la operación *V* en un semáforo, pero donde el hecho de desbloquear a un proceso que esté en la cola del semáforo se simula desbloqueándolo de la cola de la variable *condition cola, señal(cola)*.

```

struct TMonitor
{
    int contador;
    condition cola;
}

void inicializar(int i)
{
    .
    .
    contador=1;
}

void P()
{
    contador--;
    if (contador<0) espera(cola);
}

void V()
{
    Contador++;
    if (contador<=0) señal(cola);
}

```

### EJERCICIO 30

Realizar un monitor que simule un buzón con las siguientes características. Podremos enviar y recibir mensajes, y la capacidad del buzón está limitada a  $N$  mensajes. Implementar los procedimientos enviar y recibir.

#### Solución

Este ejercicio es exactamente igual que el problema del productor y consumidor con buffer limitado visto anteriormente, como en el caso de los semáforos podemos observar que las mismas técnicas se pueden aplicar a campos muy diversos.

En este caso no es un problema concreto, sino que es la simulación de una estructura que utilizamos en otro de los mecanismos de comunicación y sincronización, los mensajes. Los mensajes pueden ser directos si el mensaje se envía directamente al destinatario, o indirectos si el mensaje se envía a una entidad intermedia que es el buzón.

Como en el caso similar, si el buzón está lleno de mensajes se bloqueará a cualquier proceso que intente enviar un mensaje en *espera(lleno)*. Si el

buzón está vacío se bloqueará a cualquier proceso que intente recibir un mensaje desde el buzón, el bloqueo se llevará a cabo en esta ocasión en *espera(vacio)*.

De igual manera, el procedimiento del monitor *enviar* tiene al final la instrucción *señal(vacio)*, para despertar a algún proceso que pueda estar bloqueado por haber encontrado el buzón vacío. Simétricamente, el procedimiento del monitor *recibir* tiene al final la operación *señal(lleno)*, para despertar a algún proceso que pueda estar bloqueado por haber encontrado el buzón lleno.

```

struct TMonitor
{
    mensaje buzon[N];
    int siggent, sigsal;
    int contador;
    condition lleno, vacio;
}
{ siggent=0; sigsal=0; contador=0; }

```

```

void enviar(mensaje x)
{
    if (condador==N)
        espera(lleno);
    buzon[siggent]=x;
    siggent=(siggent + 1) % N;
    contador++;
    señal(vacio)
}

```

```

void recibir(mensaje x)
{
    if (condador==0)
        espera(vacio);
    x=buzon[sigsal];
    sigsal=(sigsal + 1) % N;
    contador--;
    señal(lleno)
}

```

```

void main()
{
    cobegin
        proceso1(); .. proceson();
    coend;
}

```

### EJERCICIO 31

En la entrada de un hotel existe una enorme puerta giratoria para la entrada y salida de los clientes, pero la citada puerta tiene unas normas de utilización muy estrictas. Los clientes del hotel pueden ser señoritas o caballeros, pero debido al puritanismo de la dirección del hotel, no se permite el paso simultáneo de ambos sexos. La puerta permite el paso de varias señoritas

simultáneamente, y lo mismo ocurre con los caballeros. Ahora bien, en caso de querer utilizar la puerta clientes de ambos sexos, los caballeros harán honor a su nombre y cederán el paso a las señoritas.

### Solución

Examinemos el cumplimiento de las restricciones en la implementación del monitor. Para comprobar si las señoritas pueden pasar varias simultáneamente, observemos el protocolo de entrada de las señoritas (procedimiento *pre\_señora*), la única condición que examinan es si existen caballeros pasando por la puerta ( $numC > 0$ ). En ningún momento comprueban si ya existen señoritas pasando, por lo que podrán pasar todas las señoritas que lleguen al mismo tiempo si no hay caballeros pasando.

Para comprobar si los caballeros pueden pasar varios al mismo tiempo, observemos el protocolo de entrada de los caballeros (procedimiento *pre\_caballero*), los caballeros comprueban si existen caballeros pasando o en la cola ( $numS > 0 \parallel señoras.n\_cola > 0$ ). En ningún momento comprueban si ya existen otros caballeros pasando, por lo que podrán pasar todos los caballeros que lleguen.

La prioridad se las señoritas se demuestra si observamos las mismas condiciones anteriores, las señoritas sólo comprueban si ya existen caballeros pasando pero no si los hay esperando pasar en la cola, por lo que las señoritas se le colarán a estos últimos. Por el contrario, los caballeros comprueban tanto si hay señoritas pasando como si están esperando, en ambos casos se bloquearán. Mientras estos caballeros están bloqueados, pueden llegar nuevas señoritas con prioridad, pero como no comprueban si los caballeros están esperando, pasarán directamente, colándose a los caballeros que estaban esperando.

```
struct TMonitor
{
    int numS, numC;
    condition señoras, caballeros;
} numS=0; numC=0;
```

<pre>void pre_señora() {     if (numC&gt;0)         espera(señoras);     numS++;     señal(señoras) }  void post_señora() {     numS--;     if (numS==0)         señal(caballeros) }</pre>	<pre>void pre_caballero() {     if ((numS&gt;0)            (señoras.n_colas&gt;0))         espera(caballeros);     numC++;     señal(caballeros); }  void post_caballero() {     numC--;     if (numC==0)         señal(señoras); }</pre>
<pre>void señora_i() {     ...     pre_señora();     pasar();     post_señora();     ... }</pre>	<pre>void caballero_j() {     ...     pre_caballero();     pasar();     post_caballero();     ... }</pre>
<pre>void main() {     cobegin         señora_1(); .. señora_n(); caballero_1(); .. caballero_m();     coend; }</pre>	

### EJERCICIO 32

Cuando se realiza el mantenimiento de las carreteras secundarias (aquellas que únicamente tienen un carril en cada sentido de circulación), los operarios suelen cortar el carril en el que están realizando el mantenimiento y gestionar el tráfico de ambos sentidos sólo por el carril que queda libre. Para la gestión del tráfico se sitúan dos operarios, uno a cada lado de las obras de mantenimiento, que sólo permiten tráfico en un solo sentido en un momento dado. La empresa ha decidido ahorrarse dos puestos de trabajo y quiere implementar el sistema de control de este tipo de situaciones mediante monitores. No existe prioridad en ninguno de los sentidos.

### Solución

Observemos que dado que ambos sentidos de circulación tienen la misma prioridad, también tienen el mismo código, excepto por las variables que les afectan a cada tipo de proceso.

Se puede observar asimismo, la similitud existente entre la estructura del código desarrollado para la solución con semáforos y la estructura del código para monitores. Ambas soluciones tienen la misma estructura.

En semáforos la técnica cuando los dos tipos de procesos tenían la misma prioridad consistía en que la primera instrucción del protocolo de entrada fuera la operación P sobre el semáforo de prioridad y la última del protocolo de entrada fuera la V sobre dicho semáforo.

Observemos que en monitores lo primero que hacemos en los protocolos de entrada es bloquear a los procesos en la variable *pasar* si hay alguien esperando. Lo último que se ejecuta en el protocolo de entrada es *señal(pasar)*, similar a la operación V en semáforos.

```
struct TMonitor
{
    int numA, numB;
    condition libre, pasar;
} numA=0; numB=0;

void pre_sentidoA()
{
    if (libre.n_cola>0)
        espera(pasar);
    if (numB>0)
        espera(libre);
    numA++;
    señal(pasar)
}

void post_sentidoA()
{
    numA--;
    if (numA==0)
        señal(libre)
}

void pre_sentidoB()
{
    if (libre.n_cola>0)
        espera(pasar);
    if (numA>0)
        espera(libre);
    numB++;
    señal(pasar)
}

void post_sentidoB()
{
    numB--;
    if (numB==0)
        señal(libre)
}
```

<pre>void sentidoA<sub>i</sub>() {     ...     pre_sentidoA();     pasar();     post_sentidoA();     ... }</pre>	<pre>void sentidoB<sub>j</sub>() {     ...     pre_sentidoB();     pasar();     post_sentidoB();     ... }</pre>
--	--

```
void main()
{
    cobegin
        sentidoA1(); .. sentidoAn(); sentidoB1(); .. sentidoBm();
    coend;
}
```

### EJERCICIO 33

En una fábrica de juguetes hay dos operarios en cadena pintando de distinto color las diversas zonas de un mismo juguete, para ello deben intercambiar el juguete cuando uno termina y debe empezar el otro, pero sin dejarlo en ningún sitio ya que la pintura del primero está reciente. Si el primer operario termina antes, debe esperar que el segundo termine su parte del juguete anterior para darle el nuevo juguete. Si el segundo operario termina antes, debe esperar hasta que el primer operario termine su parte. Implementar un monitor que tenga los procedimientos *dar* y *coger*, procedimientos que invocarán el primer y segundo operario respectivamente.

### Solución

Observemos que de nuevo ambos operarios tienen el mismo código salvo por las variables que comprueban, la definición es la misma para ambos operarios, el primero que llega espera y cuando llega el segundo intercambian el juguete.

Cada operario llega y comprueba si el otro está bloqueado esperando, en cuyo caso lo desbloquean e intercambian el juguete. Si un operario llega y el otro no está esperando será él el que se bloquee hasta que el otro lo desperte cuando llegue.

```

struct TMonitor
{
    Tjuguete juguete;
    condition operario1, operario2;
}

void dar(Tjuguete x)
{
    if (operario2.n_cola>0)
        señal(operario2);
    else
        espera(operario1);
    intercambiar();
}

void operario1()
{
    Tjuguete x;
    while (true)
    {
        x=pintar();
        dar(x);
    }
}

void coger(Tjuguete x)
{
    if (operario1.n_cola>0)
        señal(operario1);
    else
        espera(operario2);
    intercambiar();
}

void operario2()
{
    Tjuguete x;
    while (true)
    {
        coger(x);
        x=pintar();
    }
}

void main()
{
cobegin
    operario1();
    operario2();
coend;
}

```

#### EJERCICIO 34

Supongamos que existe en la ciudad de Madrid un autobús turístico “Madrid visión”, pero que en vez de realizar una ruta pasando por todos los lugares importantes, realiza un desplazamiento punto a punto entre el Palacio Real y el Museo del Prado, y viceversa. El autobús únicamente se pone en marcha cuando tiene un número suficiente de turistas, concretamente 30. Los turistas del Palacio Real invocarán el procedimiento Ir\_al\_Prado, mientras que los turistas del Museo del Prado invocarán el procedimiento Ir\_al\_Palacio. Implementar los dos procedimientos para solucionar el problema, considerando que inicialmente el autobús se encuentra parado en el museo y está vacío.

#### Solución

Como se ha comentado en la solución mediante semáforos, este problema es un problema de sincronización, veamos como se han solucionado con monitores los distintos aspectos.

El primer aspecto importante es el tratamiento de la alternancia, es decir, el autobús o está en el palacio o en el museo, pero no puede estar en los dos sitios al mismo tiempo. En este caso, para la gestión de la alternancia tenemos una variable llamada *lugar* que nos indicará en qué sitio nos encontramos.

Además de la variable anterior, se utilizan dos variables *condition*, *museo* y *palacio*, cada una asociada a un tipo de procesos. La técnica consiste en que cada tipo de proceso comprueba, al inicio de su ejecución, el lugar en el que se encuentra el autobús, y si el autobús no está en el mismo lugar que está el turista, éste se bloqueará invocando la operación *espera* sobre la variable *condition* de su lugar, por ejemplo *espera(museo)* en el caso de los turistas que están en el museo y quieren ir al palacio. Luego, al final de su ejecución, cada tipo de proceso realizará una operación *señal* sobre la variable *condition* asociada al otro tipo de procesos, por ejemplo *señal(palacio)* en el caso de los turistas que están en el museo. Esta última operación habilita que puedan progresar los procesos del otro tipo.

Como en cualquier problema de alternancia, se debe considerar una alternativa inicial, en este caso y según el enunciado, el autobús está inicialmente parado en el museo, por lo que sólo podrán subir turistas que se encuentren en dicho lugar. En el caso de los monitores, indicar la alternancia inicial es tan sencillo como asignar inicialmente 0 a la variable *lugar*.

Se puede comprobar también, que los primeros 29 turistas se quedan bloqueados en la variable *condition sc*, hasta que llega el turista 30, el cual permite arrancar el autobús y cambia el valor de la variable *lugar*, indicando que ya han llegado al otro lugar.

Después de llegar al destino, cada turista irá decrementando la variable *turistas* (simulando que baja del autobús) y despertando a otro turista (*señal(sc)*). El último turista en bajar del autobús (*turistas==0*), ejecutará la operación *señal* sobre la variable *condition* asociada al otro lugar, permitiendo la alternancia que se ha comentado anteriormente.

Observemos que los dos tipos de procesos utilizan la misma variable para saber el número de turistas que han subido al autobús. Pero sólo podrá ser incrementada en un único lugar.

```

struct TMonitor
{
    int lugar; //0=museo 1=palacio
    int turistas;
    condition museo, palacio, sc;
}
lugar=0; turistas=0;

void Ir_al_Palacio()
{
    if (lugar==1)
        espera(museo);
    turistas++;
    if (turistas<30){
        señal(museo);
        if (lugar==0)
            espera(sc);
    }
    else {
        arrancar();
        lugar=1;
    }
    /* Al llegar al Prado */
    turistas--;
    señal(sc);
    if (turistas==0)
        señal(palacio);
    . .
}

void Ir_al_Prado()
{
    if (lugar==0)
        espera(palacio);
    turistas++;
    if (turistas<30){
        señal(palacio);
        if (lugar==1)
            espera(sc);
    }
    else {
        arrancar();
        lugar=0;
    }
    /* Ya en el Palacio */
    turistas--;
    señal(sc);
    if (turistas==0)
        señal(museo);
    . .
}

void main()
{
    cobegin
        turista1();...; turistan();
    coend;
}

```

### EJERCICIO 35

En la época de la recogida de la oliva para la elaboración del aceite, los agricultores llevan los sacos llenos de oliva a la cooperativa para su pesado y entrega. Dado que existen distintos tipos de oliva que luego generan aceite de distinta calidad, el pesado se realiza por tipos, y se suele distinguir entre “blanqueta” y “gordal” (en este último se suelen incluir otros tipos menos

comunes como “negreta”, etc). Es evidente que tanto en la blanqueta como en la gordal se pueden pesar varios sacos simultáneamente. Finalmente, debemos tener en cuenta que no existe ningún tipo de prioridad, por lo que utilizaremos prioridad por orden de llegada. Implementemos un monitor que ayude al operario de la báscula.

### Solución

De nuevo estamos ante dos tipos de procesos, dos variedades de oliva en este caso, que han sido definidos de forma similar e incluso con la misma prioridad, por tanto, también tienen el mismo código, excepto por las variables que les afectan a cada tipo de proceso.

Se puede observar asimismo, la similitud existente entre la estructura del código desarrollado para la solución con semáforos y la estructura del código para monitores. Ambas soluciones tienen la misma estructura.

En semáforos la técnica cuando los dos tipos de procesos tenían la misma prioridad consistía en que la primera instrucción del protocolo de entrada fuera la operación P sobre el semáforo de prioridad y la última del protocolo de entrada fuera la V sobre dicho semáforo.

Observemos que en monitores lo primero que hacemos en los protocolos de entrada es bloquear a los procesos en la variable condition *pesar* si hay alguien esperando. Lo último que se ejecuta en el protocolo de entrada es *señal(pesar)*, similar a la operación V en semáforos.

Además, en el caso de los monitores la última instrucción del protocolo de entrada nos permite también abordar la restricción de que se puedan pesar varios sacos de la misma oliva, con esta instrucción cada saco despertará a otro que esté bloqueado en la variable condition *pesar*, si los nuevos sacos son de la misma variante se podrán pesar todos al mismo tiempo. Si se desbloquea un saco de la otra variante, el nuevo saco quedará bloqueado en *espera(libre)* y no será desbloqueado hasta el final del protocolo de salida de la variante que está siendo pesada. Además, en el momento en que se despierte un saco de la otra variante, ya no se podrán añadir más sacos de la variante que está siendo pesada.

```

struct TMonitor
{
    int numBlanq, numGordal;
    condition libre, pesar;
}
{ numBlanq=0; numGordal=0; }

```

<pre>void pre_blanqueta() {     if (libre.n_cola&gt;0)         espera(pesar);     if (numGordal&gt;0)         espera(libre);     numBlang++;     señal(pesar) }  void post_blanqueta() {     numBlang--;     if (numBlang==0)         señal(libre) }</pre>	<pre>void pre_gordal() {     if (libre.n_cola&gt;0)         espera(pesar);     if (numBlang&gt;0)         espera(libre);     numGordal++;     señal(pesar) }  void post_gordal() {     numGordal--;     if (numGordal==0)         señal(libre) }</pre>
--	--

<pre>void blanqueta_i() {     ...     pre_blanqueta();     pesar();     post_blanqueta();     ... }</pre>	<pre>void gordal_i() {     ...     pre_gordal();     pesar();     post_gordal();     ... }</pre>
---	--

```
void main()
{
    cobegin
        blanqueta_1(); .. blanqueta_n(); gordal_1(); .. gordal_m();
    coend;
}
```

### EJERCICIO 36

En el almacén de una empresa sólo existen cinco toritos para carga y descarga de los distintos bultos, en cambio, el número de operarios que potencialmente pueden utilizar los toritos es mayor. La empresa tiene un operario encargado de gestionar la asignación de los toritos, pero como en ejemplos anteriores quiere ahorrarse el coste del operario (la pele es la pela), por lo que está pensando en desarrollar un monitor que le resuelva el problema de la asignación de los toritos y la espera de los operarios cuando todos están siendo utilizados.

### Solución

Observemos que lo único que tenemos que comprobar en el protocolo de entrada es que existe algún recurso libre, en cuyo caso lo cogemos e indicamos que hay un recurso más ocupado. En caso de no haber ninguno libre bloqueamos al proceso en la cola de la variable condition.

```
struct TMonitor
{
    int ocupados, total;
    condition cola;
}
total=5; ocupados=0 }
```

```
void coger_torito()
{
    if (ocupados == total)
        espera(cola);
    ocupados++;
}

void dejar_torito()
{
    ocupados--;
    señal(cola);
}

void operario_i()
{
    coger_totiro();
    utilizar();
    dejar_torito();
}
```

```
void main()
{
    cobegin
        operario_1(); .. operario_n();
    coend;
}
```

### EJERCICIO 37

En un centro de salud existe una sala donde curan a las personas que tienen infecciones pero donde también pueden entrar personas sanas para su diagnóstico. Como no queremos que las personas que entren sanas se

infecten en la propia sala, después de haber estado personas con alguna infección y antes de que entren personas sanas, la sala debe ser esterilizada.

### Solución

Ejercicio en el que intervienen tres tipos de procesos, pero en el que se debe seguir una secuencia en la ejecución de cada uno de los tipos, después de los infectados siempre se ejecutará el proceso esteriliza.

Observemos el código de los infectados, comprueban si existen sanos en la sala, en cuyo caso no pasan a la sala y se bloquean en la variable condition *infectados*. Si no hay sanos en la sala pasan, incrementan el número de infectados, asignan a *esteril* el valor 0 indicando que la sala está contaminada, y finalmente, intentan desbloquear a otros infectados ya que pueden pasar varios simultáneamente.

```
struct TMonitor
{
    int numI, numS;
    int esteril; //0=no esteril 1=esteril
    condition infectados, esteriliza, sanos;
} numI=0; numS=0; esteril=1 }
```

<b>void pre_infectado()</b>	<b>void post_infectado()</b>
<pre>{ if (numS&gt;0)     espera(infectados); numI++; esteril=0; señal(infectados); }</pre>	<pre>{ numI--; if (numI==0)     señal(esteriliza); }</pre>
<b>void esterilizar()</b>	<b>void pre_sano()</b>
<pre>{ if (numS&gt;0    numI&gt;0)     espera(esteriliza); esteril=1; señal(sanos); }</pre>	<pre>{ if (numI&gt;0    esteril==0)     espera(sanos); numS++; señal(sanos); }</pre>
	<b>void post_sano()</b>
	<pre>{ numS--; if (numS==0)     señal(infectados); }</pre>

<b>void infectado<sub>i</sub>()</b>	<b>void sano<sub>j</sub>()</b>
<pre>{ ... pre_infectado(); sala(); post_infectado(); ... }</pre>	<pre>{ ... pre_sano(); sala(); post_sano(); ... }</pre>
<b>void esteriliza()</b>	
<pre>{ esterilizar(); }</pre>	

```
void main()
{
cobegin
    infectado1(); .. infectadon();
    esteriliza();
    sano1(); .. sanom();
coend;
}
```

Al salir los infectados, el último despertará al proceso que debe esterilizar ejecutando la operación *señal* sobre la variable condition *esteriliza*, en la que posiblemente esté bloqueado el proceso que debe esterilizar.

El proceso *esteriliza* comprueba si existen cualquier otro proceso en la sala, en cuyo caso se bloquea ejecutando *espera(esteriliza)*. Si no hay nadie, pondrá la variable *esteril* a 1 indicando que la sala está esterilizada y despertará a los sanos.

Finalmente, los sanos comprobarán si hay infectados en la sala o si la sala no está esterilizada, en cuyo caso se bloquearán. Y al salir despertarán a los infectados.

### EJERCICIO 38

Implementar las siguientes operaciones atómicas mediante monitores.

- **dormir:** suspende al proceso que la ejecuta.
- **despertar:** despierta a todos los procesos suspendidos mediante sleep.

**Solución**

```
struct TMonitor
{
    TCondition dormir;
}
```

**Procedimientos del monitor:**

<b>Dormir</b>	<b>Despertar</b>
{ espera(dormir); señal(dormir); }	{ señal(dormir); }

**EJERCICIO 39**

Sea un sistema con dos procesos: P1 y P2 que se ejecutan de forma concurrente:

<b>P1</b>	<b>P2</b>
{ ... (1) ... }	{ ... (2) ... }

Se desea sincronizar ambos procesos para que P1 ejecute (1) al mismo tiempo que P2 ejecuta (2). Resolver este problema mediante monitores.

**Solución**

```
struct TMonitor
{
    condition sinc1, sinc2;
}
```

**Procedimientos del monitor:**

```
void Pre1()
{
    señal(sinc2);
```

```
espera(sinc1);
señal(sinc2);
}
```

```
void Pre2()
{
    señal(sinc1);
    espera(sinc2);
    señal(sinc1);
}
```

**Procesos:**

<b>void P1()</b>	<b>void P1()</b>
{     ...     Pre1();     (1)     ... }	{     ...     Pre2();     (2)     ... }

**Programa principal:**

```
Void main()
{
    cobegin
        P1(); P2();
    coend;
}
```

**PASO DE MENSAJES****EJERCICIO 40**

Uno o más productores generan cierto tipo de datos y los sitúan en una zona de memoria o buffer. Un único consumidor saca elementos del buffer de uno en uno. El sistema debe impedir la superposición de operaciones sobre el buffer. Recordemos que este clásico problema tiene dos enfoques, pero en este caso se abordará con el tamaño del buffer limitado.



### Solución

Empezamos el estudio de los *mensajes*, el último mecanismo de comunicación y sincronización de procesos, con el análisis de uno de los problemas clásicos de concurrencia, que ya ha sido estudiado anteriormente en varias ocasiones en este mismo capítulo.

Antes de analizar la solución propuesta el problema y aunque los mensajes no tienen el problema de la sección crítica, veamos en qué se basa el tratamiento de la sección crítica mediante la utilización de mensajes.

- Utilización de operaciones de *enviar* no bloqueantes y operaciones de *recibir* bloqueantes
- Direccionamiento indirecto a través de buzones
- Uso de mensajes con contenido nulo o testigos

La solución propuesta se basa en la utilización de dos buzones (se supone definidos los tipos TBuzon y mensaje), *puede\_producir* usado por el productor para obtener testigos y *puede\_consumir* utilizado por el consumidor para obtener los mensajes producidos por el productor.

Estudiemos el código del proceso productor, después de producir el mensaje (simula los elementos del buffer en este ejercicio) ejecuta la operación *recibir* en el buzón *puede\_producir*. Recordemos que ésta operación es bloqueante, lo que significa que si existe algún mensaje en el citado buzón, el productor cogerá el mensaje y continuará, en caso contrario, quedará bloqueado hasta que alguien envíe un mensaje al buzón. Finalmente, el productor envía el mensaje producido al buzón *puede\_consumir*, desde donde lo recogerá el consumidor.

Analicemos ahora el código del consumidor, lo primero que hace es intentar coger un mensaje para lo que ejecuta la operación *recibir* en al buzón *puede\_consumir*. De nuevo, esta operación es bloqueante, por lo que el consumidor sólo continuará si existe algún mensaje en el buzón, en caso contrario se bloqueará. Dado que el buzón simula el buffer, el hecho de que no existan mensajes significa que el buffer está vacío y por tanto el consumidor debe bloquearse (sincronización del consumidor). Después de coger el mensaje, el consumidor envía un mensaje nulo al buzón *puede\_producir*, este mensaje será un nuevo testigo que podrá coger el productor sin bloquearse, o bien, permitirá progresar al productor en caso de que estuviese bloqueado por estar el buffer lleno.

Finalmente, observemos el aspecto de la sincronización del productor cuando el buffer está lleno (en este caso cuando el buzón *puede\_consumir* esté lleno). Lo que se hace es llenar el buzón *puede\_producir* de N mensajes nulos o testigos, por lo que el productor podrá obtener N testigos y enviar N mensajes (buffer lleno) antes de bloquearse. Cuando intente coger el testigo

N+1 se bloqueará hasta que el consumidor consuma algún mensaje y le devuelva un testigo.

```
#define tamaño N
TBuzon puede_producir, puede_consumir;
```

```
void productor()
{
    mensaje msjp, aux;
    while(true)
    {
        msjp = producir();
        recibir(puede_producir, aux);
        enviar(puede_consumir, msjp);
    }
}

void consumidor()
{
    mensaje msjc;
    while(true)
    {
        recibir(puede_consumir, msjc);
        enviar(puede_producir, NULL);
        consumir(msjc);
    }
}
```

```
void main()
{
    int i;
    crear_buzón(puede_producir);
    crear_buzón(puede_consumir);
    for(i=0; i<tamaño; i++)
        enviar(puede_producir, NULL);
    cobegin
        productor();
        consumidor();
    coend;
}
```

### EJERCICIO 41

En la entrada de un hotel existe una enorme puerta giratoria para la entrada y salida de los clientes, pero la citada puerta tiene unas normas de utilización muy estrictas. Los clientes del hotel pueden ser señoras o caballeros, pero debido al puritanismo de la dirección del hotel, no se permite el paso

simultáneo de ambos sexos. La puerta permite el paso de varias señoritas simultáneamente, y lo mismo ocurre con los caballeros. Ahora bien, en caso de querer utilizar la puerta clientes de ambos sexos, los caballeros harán honor a su nombre y cederán el paso a las señoritas.

Implementar una solución que controle la puerta ajustándose a las restricciones del enunciado.

### Solución

Como podemos observar, en la solución tenemos tres tipos de procesos, los dos involucrados en la definición del problema y un tercero llamado *puerta*, que se encargará de la coordinación de la puerta y tratamiento de la prioridad. Asimismo, utilizaremos cinco buzones; *pasoS* estará asociado indirectamente con la prioridad, *entradaS* para que las señoritas indiquen que quieren pasar, *salidaS* utilizado por las señoritas para indicar que han salido de la puerta, *entradaC* para que los caballeros indiquen que quieren pasar y *salidaC* usado por los caballeros para informar que ya han salido de la puerta.

Observemos el código de las señoritas, primero enviarán un mensaje con su *pid* al buzón *pasoS* que será utilizado por el proceso *puerta* para incrementar la variable *espS* que se utilizará para la prioridad.

Segundo, enviarán de nuevo su *pid* al buzón *entradaS*, para indicarle al proceso *puerta* que quieren pasar por la puerta.

Tercero, realizarán la operación *recibir(puerta,testigo)* y esperarán en esta instrucción hasta que el proceso *puerta* les pase un testigo, el lector debe notar que en este caso la operación de mensajes es directa a un proceso concreto (no a un buzón).

Finalmente, después de pasar por la puerta enviarán su *pid* al buzón *salidaS* para comunicarle al proceso *puerta* que han salido.

Los caballeros ejecutan un código muy similar, primero, enviarán su *pid* al buzón *entradaC*, para indicarle al proceso *puerta* que quieren pasar por la puerta. Segundo, realizarán la operación *recibir(puerta,testigo)* y esperarán en esta instrucción hasta que el proceso *puerta* les pase un testigo, en este caso la operación de mensajes también es directa a un proceso concreto (no a un buzón). Finalmente, después de pasar por la puerta enviarán su *pid* al buzón *salidaC* para comunicarle al proceso *puerta* que han salido.

Analicemos detenidamente el código del proceso coordinador *puerta*. Éste proceso está constantemente recibiendo mensajes de los cinco buzones involucrados en todo el proceso, lógico si se tiene en cuenta que va a ser el moderador y necesita toda la información.

Si recibe un mensaje de *pasoS* incrementará la variable *espS* (número señoritas que esperan pasar), si recibe un mensaje de *salidaS* decrementará la variable *numS* (número señoritas que están pasando), y si recibe de *salidaC* decrementará la variable *numC* (número caballeros que están pasando).

Finalmente, si recibe mensajes de los buzones de las entradas, en el caso de las señoritas si no hay caballeros pasando (*numC==0*) pasarán ellas. En el caso de los caballeros, para pasar no tienen que haber señoritas ni pasando ni esperando (*numS==0 && espS==0*), lo que le da prioridad a las señoritas, ahora puede llegar una señora y enviar un mensaje a *pasoS* y *entradaS*, esta señora pasará antes que el caballero.

```
TBuzon pasoS, entradaS, entradaC;
TBuzon salidaS, salidaC;

void puerta();
{
    int numS, numC, espS;
    numS=0; numC=0; espS=0;
    while (true)
    {
        if (numC==0 && espS>0 && recibir(entradaS,pid)) {
            numS++;
            espS--;
            enviar(pid,testigo);
        }
        if (recibir(pasoS, pid))
            espS++;
        if (numS==0 && espS==0 && recibir(entradaC,pid)) {
            numC++;
            enviar(pid,testigo);
        }
        if (numS>0 && recibir(salidas,pid))
            numS--;
        if (numC>0 && recibir(salidaC,pid))
            numC--;
    }
}

void señora_()
{
    ...
    enviar(pasoS, pid);
    enviar(entradaS, pid);
    recibir(puerta,testigo);
    pasar();
    enviar(salidas, pid);
    ...
}

void caballero_()
{
    ...
    enviar(entradaC, pid);
    recibir(puerta,testigo);
    pasar();
    enviar(salidaC, pid);
    ...
}
```

```

void main()
{
    int testigo=1;
    crear_buzón(pasosS);
    crear_buzón(entradas);
    crear_buzón(salidas);
    crear_buzón(entradaC);
    crear_buzón(salidaC);

    cobegin
        puerta();
        señora1() .. señoran();
        caballero1() .. caballerom();
    coend;
}

```

#### EJERCICIO 42

Cuando se realiza el mantenimiento de las carreteras secundarias (aquellas que únicamente tienen un carril en cada sentido de circulación), los operarios suelen cortar el carril en el que están realizando el mantenimiento y gestionar el tráfico de ambos sentidos sólo por el carril que queda libre. Para la gestión del tráfico se sitúan dos operarios, uno a cada lado de las obras de mantenimiento, que sólo permiten tráfico en un solo sentido en un momento dado. La empresa ha decidido ahorrarse dos puestos de trabajo y quiere implementar el sistema de control de este tipo de situaciones mediante mensajería. No existe prioridad en ninguno de los sentidos.

#### Solución

Al igual que en el caso anterior, en la solución tenemos tres tipos de procesos, los dos involucrados en la definición del problema y un tercero llamado *carril*, que se encargará de la coordinación del carril habilitado y tratamiento de la prioridad (sin prioridad en este caso). Utilizaremos tres buzones; *entradaA* para que los vehículos del sentido A indiquen que quieren pasar, *entradaB* para que los del sentido contrario indiquen que quieren pasar y *salida* usado por los dos tipos de procesos para informar que ya han pasado en lugar en obras.

Entendemos el código del sentido A, primero, enviarán su *pid* al buzón *entradaA*, para indicarle al proceso *carril* que quieren pasar por el carril habilitado. Segundo, realizarán la operación *recibir(carril,testigo)* y esperarán en esta instrucción hasta que el proceso *carril* les pase un testigo, al

igual que ocurría en el ejercicio anterior, el lector debe notar que en este caso la operación de mensajes es directa a un proceso concreto (no a un buzón). Finalmente, después de pasar por la zona en mantenimiento enviarán un 0 al buzón *salida* para comunicarle al proceso *carril* que ha pasado un vehículo en el sentido A.

Observemos el código asociado al sentido B, primero, enviarán su *pid* al buzón *entradaB*, para indicarle al proceso *carril* que quieren pasar por el carril habilitado. Segundo, realizarán la operación *recibir(carril,testigo)* y esperarán en esta instrucción hasta que el proceso *carril* les pase un testigo, de nuevo la operación de mensajes es directa a un proceso concreto (no a un buzón). Finalmente, después de pasar por la zona en mantenimiento enviarán un 1 al buzón *salida* para comunicarle al proceso *carril* que ha pasado un vehículo en el sentido B.

Analicemos detenidamente el código del proceso coordinador *carril*. Éste proceso está constantemente recibiendo mensajes de los tres buzones involucrados en todo el proceso, lógico si se tiene en cuenta que va a ser el moderador y necesita toda la información.

Si recibe un mensaje de *entradaA* (indicando que quieren pasar en sentido A) y no hay vehículos en el otro sentido pasando (*sentidoB==0*), el coordinador incrementará el número de vehículos de A y le enviará un testigo, permitiéndole al proceso del sentido A circular. El caso de recibir mensajes de *entradaB* es idéntico.

Finalmente, si recibe mensajes del buzón de salida, decrementará la variable del sentido que corresponda, indicando que hay un vehículo menos de ese sentido pasando. El sentido del vehículo que sale se le indica en el mensaje.

```

TBuzon entradaA, entradaB, salida;

Void carril();
{
    int dir;                                //0=sentidoA 1=sentidoB
    int sentidoA, sentidoB;
    sentidoA=0; sentidoB=0;
    while (true)
    {
        if (sentidoB==0 && recibir(entradaA,pid)) {
            sentidoA++;
            enviar(pid,testigo);
        }
        if (sentidoA==0 && recibir(entradaB,pid)) {
            sentidoB++;
            enviar(pid,testigo);
        }
        if ((sentidoA>0 || sentidoB>0) && reci-

```

```

bir(salida,dir) {
    if (dir == 0)
        sentidoA--;
    else
        sentidoB--;
}
}

void sentidoA_i()
{
    ...
enviar(entradaA, pid);
recibir(carril,testigo);
circular();
enviar(salida, 0);
...
}

void sentidoB_i()
{
    ...
enviar(entradaB, pid);
recibir(carril,testigo);
circular();
enviar(salida, 1);
...

void main()
{
    int testigo=1;
    crear_buzón(entradaA);
    crear_buzón(entradaB);
    crear_buzón(salida);
    cobegin
        carril();
        sentidoA_i(); .. sentidoA_n();
        sentidoB_i(); .. sentidoB_n();
    coend;
}

```

#### EJERCICIO 43

Supongamos que existe en la ciudad de Madrid un autobús turístico “Madrid visión”, pero que en vez de realizar una ruta pasando por todos los lugares importantes, realiza un desplazamiento punto a punto entre el Palacio Real y el Museo del Prado, y viceversa. El autobús únicamente se pone en marcha cuando tiene un número suficiente de turistas, concretamente 30. Los turistas del Palacio Real invocarán el procedimiento Ir\_al\_Prado, mientras que los turistas del Museo del Prado invocarán el procedimiento Ir\_al\_Palacio. Implementar una solución mediante mensajes, considerando que inicialmente el autobús se encuentra parado en el museo y está vacío.

#### Solución

Ejercicio de alternancia solucionado mediante los mecanismos de semáforos y monitores y que abordamos ahora mediante la utilización de mensajes.

En este caso sólo tendremos dos tipos de procesos, los turistas y el proceso *autobús*, proceso que como en los casos anteriores se encargará de la coordinación de todos los turistas en la subida y bajada del autobús. Se debe tener en cuenta que los turistas, aunque sólo son un tipo de procesos, pueden invocar tanto al procedimiento *ir\_al\_palacio* como a *ir\_al\_prado*.

Los turistas que están en el museo y quieren ir al palacio enviarán su *pid* al buzón *museo*, para indicarle al proceso *autobus* que quieren subir. Segundo, realizarán la operación *recibir(autobus,testigo)* y esperarán en esta instrucción hasta que el proceso *autobus* les pase un testigo. Finalmente, después de llegar al palacio (y por supuesto, después de ser 30) enviarán un 0 al buzón *bajar* para comunicarle al proceso *autobus* que han bajado del autobús y que son los turistas que estaban en el museo.

Los turistas que están en el palacio y quieren ir al museo ejecutarán prácticamente el mismo código, excepto por el hecho de que indican que quieren subir al autobús en el buzón *palacio*, y comunican al proceso *autobús* que han bajado enviando un 1 al buzón *bajar*, indicandole que son los turistas que estaban en el palacio.

El estudio del proceso *autobus* muestra que las dos primeras decisiones son iguales, comprueban que el mensaje recibido coincide con el lugar desde el que viene, y si es así, incrementa el número de turistas y envía un testigo al proceso. Finalmente, si recibe mensajes en *bajar*, decrementa turistas y cambia de lugar.

```

TBuzon museo, palacio, bajar;

Void autobus()
{
    int lugar;                                //0=museo 1=palacio
    int turistas, i;
    int pids[30];
    turistas=0; lugar=0;
    while (true)
    {
        if (lugar==0 && turistas<30 && recibir(museo,pid)) {
            pids[turistas]=pid; turistas++;
        }
        if (lugar==1 && turistas<30 && recibir(palacio,pid)) {
            pids[turistas]=pid; turistas++;
        }
        if (turistas == 30){
            for(i=0;i<30;i++) enviar(pids[i],testigo);
        }
    }
}

```

```

if (turistas>0 && recibir(bajar,lugar)) {
    turistas--;
    if (turistas == 0) {
        if (lugar == 0)
            lugar=1;
        else
            lugar=0;
    }
}
}

void ir_al_Palacio()
{
    ...
    enviar(museo, pid);
    recibir(autobus,testigo);
    arrancar();
    enviar(bajar, 0);
    ...
}

void ir_al_Prado()
{
    ...
    enviar(palacio, pid);
    recibir(autobus,testigo);
    arrancar();
    enviar(bajar, 1);
    ...
}

void main()
{
    int testigo=1;
    crear_buzón(museo);
    crear_buzón(palacio);
    crear_buzón(bajar);
    cobegin
        autobus();
        turista_(); .. turista_n();
    coend;
}

```

#### EJERCICIO 44

En la época de la recogida de la oliva para la elaboración del aceite, los agricultores llevan los sacos llenos de oliva a la cooperativa para su pesado y entrega. Dado que existen distintos tipos de oliva que luego generan aceite de distinta calidad, el pesado se realiza por tipos, y se suele distinguir entre “blanqueta” y “gordal” (en este último se suelen incluir otros tipos menos comunes como “negreta”, etc). Es evidente que tanto en la blanqueta como en la gordal se pueden pesar varios sacos simultáneamente. Finalmente, debemos tener en cuenta que no existe ningún tipo de prioridad, por lo que utilizaremos prioridad por orden de llegada. Implementemos una solución mediante mensajes que ayude al operario de la báscula.

#### Solución

Problema en el que tenemos tres tipos de procesos, los dos involucrados en la definición, y el proceso *pesa* que actúa como coordinador y gestor de la prioridad.

Los procesos de las variedades de oliva, comunican que quieren pasar, intentan recibir el testigo, y después del pesado comunican que han sido vaciados en el silo y su tipo como argumento. El proceso *pesa* si recibe un mensaje en los buzones de entrada y no se están pesando sacos del otro tipo, permite pesar al saco que envía el mensaje enviándole un testigo, y cuando recibe un mensaje en el buzón de salida *silo*, decrementa el número de sacos del tipo que le pasan como parámetro del mensaje.

```

TBuzon blanqueta, gordal, silo;

void pesa();
{
    int tipo;                                //0=blanqueta 1=gordal
    int numBlanq, numgordal;
    numBlanq=0; numgordal=0;
    while (true)
    {
        if (numgordal==0 && recibir(blanqueta,pid)) {
            numBlanq++;
            enviar(pid,testigo);
        }
        if (numBlanq==0 && recibir(gordal,pid)) {
            numgordal++;
            enviar(pid,testigo);
        }
        if ((numBlanq>0||numgordal>0) && recibir(silo, tipo)){
            if (tipo == 0)
                numBlanq--;
            else
                numgordal--;
        }
    }
}

```

```

void blanqueta_i()
{
    ...
    enviar(blanqueta, pid);
    recibir(pesa,testigo);
    pesar();
    enviar(silo, 0);
    ...
}

void gordal_i()
{
    ...
    enviar(gordal, pid);
    recibir(pesa,testigo);
    pesar();
    enviar(silo, 1);
    ...
}

void main()
{
    int testigo=1;
    crear_buzón(blanqueta);
    crear_buzón(gordal);
    crear_buzón(silo);
    cobegin
        pesa();
        blanqueta_i(); .. blanqueta_n();
        gordal_i(); .. gordal_m();
    coend;
}

```

**EJERCICIO 45**

Implementa las siguientes operaciones mediante paso de mensajes

- **dormir**: suspende al proceso que la ejecuta.
- **despertar**: despierta a todos los procesos suspendidos mediante sleep.

*Solución*Condiciones de la comunicación:

- Enviar bloqueante.
- Recibir no bloqueante.
- Comunicación indirecta.

TBuzón B;

<b>Dormir</b> { TMensaje msj;  enviar(B, NULL); recibir(B, msj); }	<b>Despertar</b> { TMensaje msj;  recibir(B, msj); }
--	---

Programa principal:

```

Void main()
{
    CrearBuzón(B);
}

```

**EJERCICIO 46**

Sea un sistema con dos procesos: P1 y P2 que se ejecutan de forma concurrente:

<b>P1</b> { ... (1) ... }	<b>P2</b> { ... (2) ... }
--	--

Se desea sincronizar ambos procesos para que P1 ejecute (1) al mismo tiempo que P2 ejecuta (2). Resolver este problema mediante:

- Paso de mensajes
- Paso de mensajes con las siguientes condiciones de comunicación:
  - Operación **enviar** bloqueante
  - Operación **recibir** no bloqueante
  - Direccionamiento indirecto

*Solución*

## a) Solución mediante paso de mensajes

Como en el enunciado no se indica ninguna condición se suponen las siguientes características de la comunicación:

### Condiciones de la comunicación:

- Operación **enviar** bloqueante
- Operación **recibir** bloqueante
- Direccionamiento directo

<pre>void P1() {     ...     Enviar(P2, NULL);     (1)     ... }</pre>	<pre>void P2() {     TMensaje msj;     ...     Recibir(P1, msj);     (2)     ... }</pre>
--	--

b) Paso de mensajes con las condiciones indicadas en el enunciado

<pre>void P1() {     TMensaje msj1, msj2;     a;     Recibir(Box, msj1);     msj2="P1";     Enviar(Box, msj2);     while (msj1!="P2")         Recibir(Box, msj1);     (1)     b; }</pre>	<pre>void P2() {     TMensaje msj1, msj2;     c;     Recibir(Box, msj1);     msj2="P2";     Enviar(Box, msj2);     while (msj1!="P1")         Recibir(Box, msj1);     (2)     d; }</pre>
--	--

### Programa principal:

```
TBuzón Box;

void main()
{
    inicializarBuzon(Box);
    cobegin
        P1(); P2();
    coend;
}
```

## CAPÍTULO 3. INTERBLOQUEO

### INTRODUCCIÓN

Un interbloqueo se define como un bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o bien se comunican unos con otros.

Los procesos adquieren algún recurso y esperan a que otros recursos retenidos por otros procesos se liberen.

### Ejemplo:

<pre>void Proceso1() {     . . .     P(S1)     P(S2)     . . .     V(S2)     V(S1) }</pre>	<pre>void Proceso2() {     . . .     P(S2)     P(S1)     . . .     V(S1)     V(S2) }</pre>
--	--

### Caracterización del interbloqueo:

- Exclusión mutua
- Retención y espera
- No existencia de expropiación
- Espera circular

**Prevención:** Evitar cualquier posibilidad que pueda llevar a una situación de interbloqueo Evitar una de las condiciones del interbloqueo

- Retención y espera: Un proceso con un recurso no puede pedir otro.
- No existencia de expropiación: Permitir la expropiación de recursos no utilizados.
- Espera circular: Se solicitan los recursos según un cierto orden establecido.

**Predicción:** Se decide dinámicamente si la petición actual de asignación de un recurso a un proceso podría, de concederse, llevar potencialmente a un interbloqueo

- No iniciar un proceso si sus demandas pueden llevar a interbloqueo.
- No conceder una solicitud de recursos a un proceso si esta asignación puede llevar a interbloqueo: Algoritmo del banquero.

**Detección:** Comprobar periódicamente si se ha producido un interbloqueo

Grafos de asignación de recursos: Se identifican los procesos, recursos, peticiones y asignaciones.

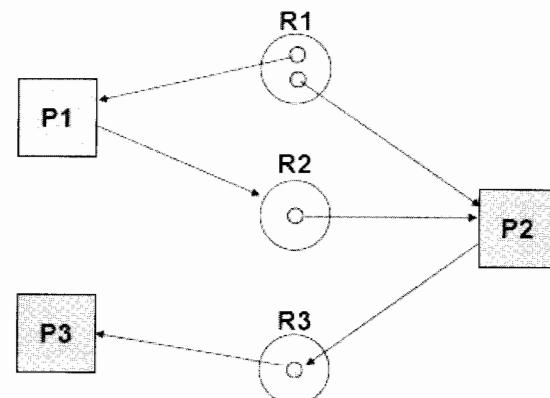


Figura 3-1. Grafo de asignación de recursos.

**Si no existen ciclos:** No hay interbloqueo

**Existen ciclos:** Puede existir interbloqueo.

- Si sólo hay un elemento por cada tipo de recurso, la existencia de un ciclo es condición necesaria y suficiente para el interbloqueo.
- Si hay algún camino que no sea ciclo, que sale de alguno de los nodos que forman el ciclo, entonces no hay interbloqueo.

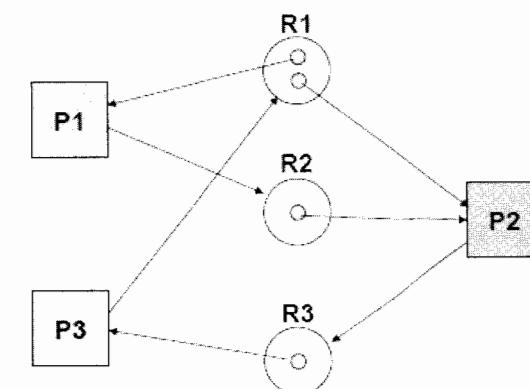


Figura 3-2. Grafo de asignación de recursos con interbloqueo.

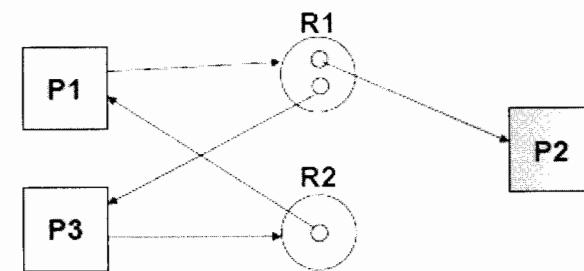


Figura 3-3. Grafo de asignación de recursos con ciclo pero sin interbloqueo.

**Recuperación:** Romper el interbloqueo para que los procesos puedan finalizar su ejecución y liberar los recursos.

Si no se consigue de esta forma se debe de reiniciar uno o más procesos bloqueados, considerando:

- La prioridad del proceso
- El Tiempo de procesamiento utilizado y el que resta
- Tipo y número de recursos que posee
- Número de recursos que necesita para finalizar
- Número de procesos involucrados en su reiniciación.

Otra opción a la de reiniciar los procesos bloqueados es la de expropiar los recursos de algunos de los procesos bloqueados

También se pueden buscar soluciones combinadas como:

- Agrupar los recursos en clases disjuntas
- Se evita el interbloqueo entre las clases

- Usar en cada clase el método más apropiado para evitar o prevenir en ella el interbloqueo

### EJERCICIO 1

Comentar si los siguientes estados son seguro o no y explicar porqué en cada uno de ellos.

a)

Tabla 3-1. Recursos totales disponibles : 4.

Proceso	Recursos asignados	Recursos máximos necesarios
P1	3	5
P2	1	4
P3	4	5

b)

Tabla 3-2. Recursos totales disponibles : 2.

Proceso	Recursos asignados	Recursos máximos necesarios
P1	3	6
P2	1	4
P3	4	8

### Solución

- El estado es seguro puesto que se disponen de un total de 4 recursos y esto permite terminar la ejecución a cualquiera de los tres procesos. Una vez hecho esto se liberarían y se procedería a la ejecución de otro proceso que evidentemente no tendría problemas en la realización puesto que va a disponer de aún más recursos una vez liberados los de su proceso predecesor.
- El estado no es seguro puesto que la cantidad de recursos necesarios para acabar cualquiera de los procesos es superior a los dos recursos de que disponemos.

### EJERCICIO 2

En esta ocasión y teniendo en cuenta que existen multiples recursos comentar si los siguientes estados son seguro o no y explicar porqué en cada uno de ellos. Utilizar el algoritmo del banquero como método de predicción de interbloqueo.

a)

Tabla 3-3. Recursos totales disponibles, asignados y máximos necesarios.

R1	R2	R3
1	2	1

Proceso	Recursos asignados			Recursos máximos necesarios		
	R1	R2	R3	R1	R2	R3
P1	2	3	1	4	5	1
P2	3	4	4	4	6	4
P3	0	0	1	2	2	1

b)

Tabla 3-4. Recursos totales disponibles, asignados y máximos necesarios.

R1	R2	R3
1	2	0

Proceso	Recursos asignados			Recursos máximos necesarios		
	R1	R2	R3	R1	R2	R3
P1	1	1	0	2	1	0
P2	2	0	1	4	4	1
P3	2	1	0	2	2	2

### Solución

- El estado es seguro si se coge la secuencia correcta de ejecución de los procesos. Según el algoritmo del banquero y con su estrategia de predicción de interbloqueo a través de la ejecución de procesos sólo en los estados seguros tendremos: P2, P3, P1 o también P2, P1, P3.

- b) El estado es inseguro. Aunque el proceso P1 se puede ejecutar hasta el final no ocurre lo mismo con los procesos P2 ni P3 puesto que, ni aún cuando los recursos usados por el proceso P1 se liberan después de su ejecución, se disponen de las suficientes unidades de los recursos R2 y R3 para los procesos P2 y P3 respectivamente.

### EJERCICIO 3

Dada la siguiente tabla con los recursos asignados y máximos necesarios, así como los recursos disponibles, comentar que solución se puede buscar para no llevar al sistema a casos inseguros.

Tabla 3-5. Recursos y procesos gestionado mediante el algoritmo del banquero.

Asignados						Máximos necesarios						Disp.	
	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5		
R1	0	5	2	3	2		R1	2	5	3	4	4	1
R2	0	1	1	2	0		R2	2	2	1	3	1	0
R3	1	3	3	1	2		R3	3	3	4	1	2	1
	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5		
R1	0	5	0	3	2		R1	2	5	0	4	4	3
R2	0	1	0	2	0		R2	2	2	0	3	1	1
R3	1	3	0	1	2		R3	3	3	0	1	2	4
	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5		
R1	0	0	0	3	2		R1	2	0	0	4	4	8
R2	0	0	0	2	0		R2	2	0	0	3	1	2
R3	1	0	0	1	2		R3	3	0	0	1	2	7
	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5		
R1	0	0	0	3	0		R1	2	0	0	4	4	1
R2	0	0	0	2	0		R2	2	0	0	3	1	2
R3	1	0	0	1	0		R3	3	0	0	1	2	9

	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5		
R1	0	0	0	3	0		R1	0	0	0	4	0	1
R2	0	0	0	2	0		R2	0	0	0	3	0	2
R3	0	0	0	1	0		R3	0	0	0	1	0	1
	P1	P2	P3	P4	P5		P1	P2	P3	P4	P5		
R1	0	0	0	0	0		R1	0	0	0	0	0	3
R2	0	0	0	0	0		R2	0	0	0	0	0	4
R3	0	0	0	0	0		R3	0	0	0	0	0	1

### Solución

Para la estrategia de predicción del interbloqueo hay que asegurar que el sistema de procesos y recursos está siempre en un estado seguro. Para ello, cuando un proceso realiza una solicitud de un conjunto de recursos, se supone que la solicitud se concede, se actualiza el estado del sistema y se determina si el resultado es un estado seguro. Si lo es, se concede la solicitud y, si no, se bloquea al proceso hasta que sea seguro conceder la solicitud. Así vamos a proceder con cada una de las peticiones de los cuatro procesos, comprobando que la única válida en el instante 1 es la de conceder la solicitud del proceso P3 pues es el único que lleva a un estado seguro. En definitiva, hay que comprobar si la concesión de cada unidad lleva a un estado seguro (que haga finalizar el proceso). De no ser así, no se concede la solicitud.

Por ejemplo, vamos a imaginar que fuera el proceso P1 el que hiciera la solicitud, necesita 2 unidades del recurso 1, 2 del recurso 2 y 2 del recurso 3. Como no se disponen de estos recursos (el algoritmo del banquero siempre dispone de menos recursos de los verdaderamente necesarios si se suman todos los procesos con el fin de evitarse problemas y buscando la liberación de los recursos una vez los procesos hayan acabado) se deniega la solicitud del proceso P1. Lo mismo ocurre con el proceso 2. A continuación se observa que el 3 no tiene problemas para cumplir sus solicitudes de los recursos necesarios. Así que se ejecuta y al terminar sus recursos pasan a ser recursos disponibles para peticiones de los restantes procesos.

Aún así hay varias secuencias que no se pueden dar y que serán rechazadas por las estrictas normas del "banquero". Una solución correcta conducida por estados seguros es la representada en la tabla 3-5 y que se consigue mediante la ejecución secuencial de los procesos P3, P2, P5, P1 y P4.

**EJERCICIO 4**

Si disponemos de 3 recursos R1,R2,R3 y 4 procesos P1,P2,P3,P4.

Y teniendo en cuenta que los recursos disponibles son 1,2,2 para cada uno de los 3 recursos respectivamente.

*Tabla 3-6. Tabla de máximos recursos necesarios.*

	R1	R2	R3
P1	1	1	1
P2	1	1	1
P3	0	1	1
P4	1	0	0

Dada la siguiente secuencia de peticiones:

P1 solicita (0,1,0)  
 P3 solicita (0,1,0)  
 P2 solicita (1,0,0)  
 P1 solicita (1,0,0)  
 P2 solicita (0,0,1)  
 P1 solicita (0,0,1)  
 P3 solicita (0,0,1)  
 P2 solicita (0,1,0)

¿Qué solicitud produce estado inseguro en el sistema?

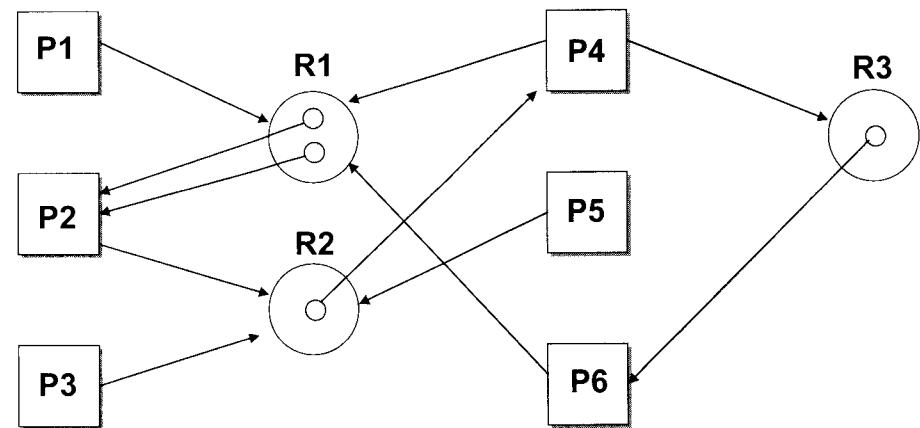
*Solución*

P1 solicita (0,1,0)	→	1,1,2
P3 solicita (0,1,0)	→	1,0,2
P2 solicita (1,0,0)	→	0,0,2
P2 solicita (0,0,1)	→	0,0,1
P1 solicita (0,0,1)	→	0,0,0
P1 solicita (1,0,0)		estado inseguro
P3 solicita (0,0,1)		
P2 solicita (0,1,0)		

**EJERCICIO 5**

Indica si en el grafo de procesos y recursos que representa la figura 3-4 existe interbloqueo.

- Si no existe, explica la secuencia de ejecución con la que terminen todos los procesos.
- Si existe, explica un mecanismo para romper el interbloqueo y aplicalo al grafo de la pregunta.



*Figura 3-4. Grafo de procesos y recursos.*

*Solución*

Sí que hay interbloqueo. Tenemos un ciclo:

R1 → P2 → R2 → P4 → R3 → P6 → R1

No hay ningún camino del ciclo que conduzca a un nodo fuera del ciclo.

**EJERCICIO 6**

El sistema informático de una empresa posee 6 impresoras en red y 4 discos duros compartidos. En el sistema se ejecutan 6 procesos concurrentemente. Representa mediante un grafo de procesos y recursos el estado del sistema en el que las 6 impresoras estén asignadas a los 6 procesos, los 4 discos asignados a 4 procesos y donde los otros procesos que no tienen disco realizan dos peticiones cada uno a dos de los discos duros.

Indica si existe interbloqueo en ese sistema:

- Si existe, explica un mecanismo para romper el interbloqueo y aplícalo al grafo de la pregunta.
- Si no existe, explica la secuencia de ejecución con la que terminen todos los procesos.

#### Solución

Se construye el diagrama de recursos y procesos que muestra la figura 3-5 siguiente:

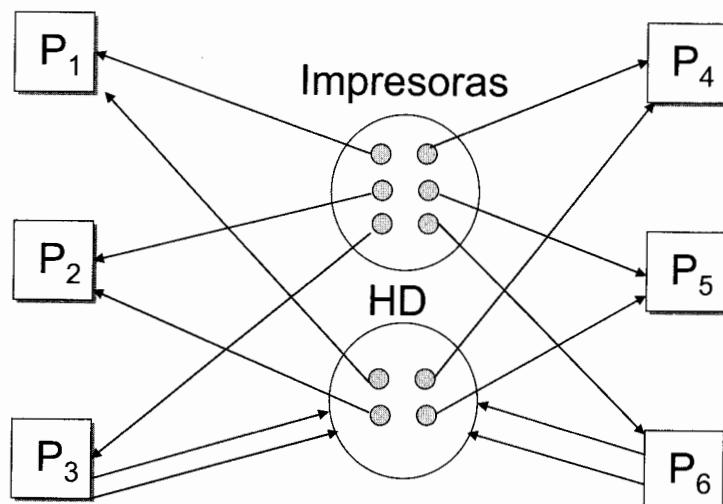


Figura 3-5. Grafo de procesos y recursos.

Como se observa en ella, no existe ciclo, por lo que no hay interbloqueo. Una secuencia válida en la que terminan todos los procesos es la siguiente:

Terminan dos de los procesos P1, P2, P3 ó P4 que tienen asignados todos los recursos que solicitaron. Al terminar liberan sus discos y se les pueden asignar a P3 y P4. Depende del orden de asignación de los recursos, habrá que esperar a que terminen los cuatro o solo dos de ellos para que P3 y P4 puedan continuar. Cuando estos últimos tengan todos sus recursos y los utilicen podrán terminar.

#### EJERCICIO 7

Sea un sistema monoprocesador con  $n$  recursos compartidos ( $R_i$ ) en el que se lanzan a ejecución simultáneamente  $n$  procesos ( $P_i$ ). Analiza si existe interbloqueo en algún instante del siguiente esquema de ejecución de los procesos:

Tabla 3-7. Tabla de solicitud y asignación de recursos.

$t=T_0$	Los procesos $i$ , con $i$ par, solicitan el recurso $(i+1) \% n$ .
$t=T_1$	Los procesos $i$ , con $i$ impar, solicitan el recurso $i$ .
$t=T_2$	Los procesos $i$ , con $i$ impar, solicitan el recurso $(i+1) \% n$ .
$t=T_3$	Los procesos $i$ , con $i$ par, solicitan el recurso $i$ .
$t=T_4$	Los recursos asignados en $T_0$ se liberan tras ser utilizados.
$t=T_5$	Los recursos asignados en $T_1$ se liberan tras ser utilizados.
$t=T_6$	Los recursos asignados en $T_2$ se liberan tras ser utilizados.
$t=T_7$	Los recursos asignados en $T_3$ se liberan tras ser utilizados.

Nota: Los procesos y los recursos se deben empezar a contar por 0.

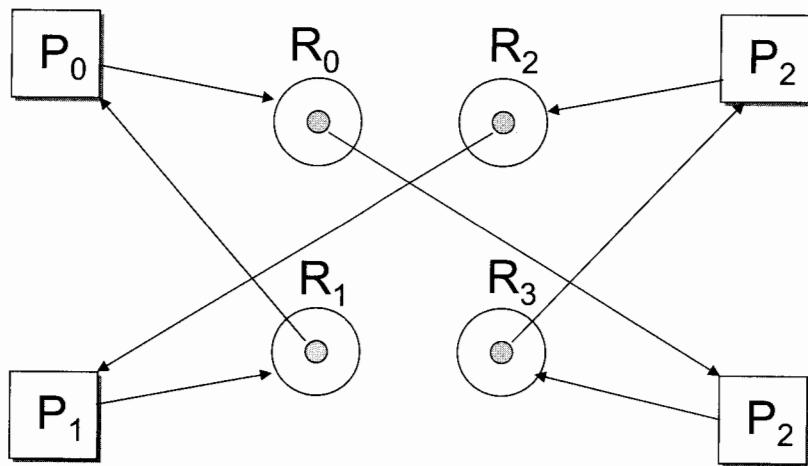
#### Solución

Se puede producir una situación de interbloqueo dependiendo de la cantidad de procesos y si de su número es par o impar, ya que esta característica condiciona su comportamiento.

Caso 1: Con  $n=1$  no hay interbloqueo. Un proceso no puede interbloquearse consigo mismo.

Caso 2: Con una cantidad par de procesos y recursos ( $n$  par).

Por ejemplo, con  $n=4$ . La situación del sistema en el instante  $T_3$  se muestra en la figura 3-6:

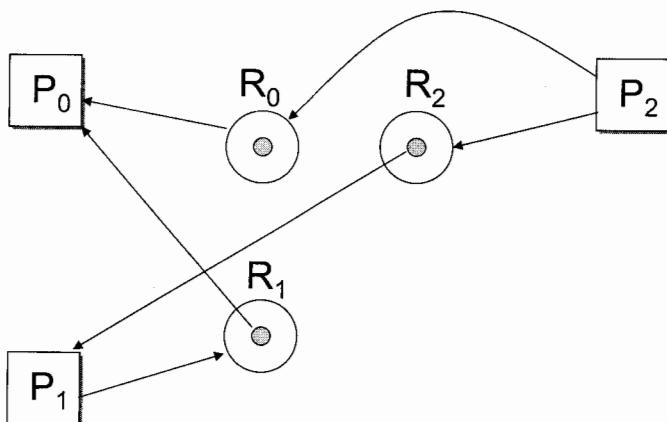
Figura 3-6. Grafo de procesos y recursos para  $n=4$ .

Existe un ciclo en el instante  $T_3$ :  $P_1 \rightarrow R_1 \rightarrow P_0 \rightarrow R_0 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$

Como solo hay una instancia de recurso por tipo de recurso la existencia de ciclo es condición suficiente para que se dé interbloqueo.

Caso 3: Con una cantidad impar de procesos y recursos ( $n$  impar).

Por ejemplo, con  $n=3$ . La situación del sistema en el mismo ( $T_3$ ) instante se muestra en la figura 3-7:

Figura 3-7. Grafo de procesos y recursos para  $n=3$ .

En este caso no existe ciclo y en consecuencia tampoco interbloqueo en ningún instante del proceso.

#### EJERCICIO 8

Comprueba si en el grafo de procesos y recursos que representa la figura siguiente existe interbloqueo.

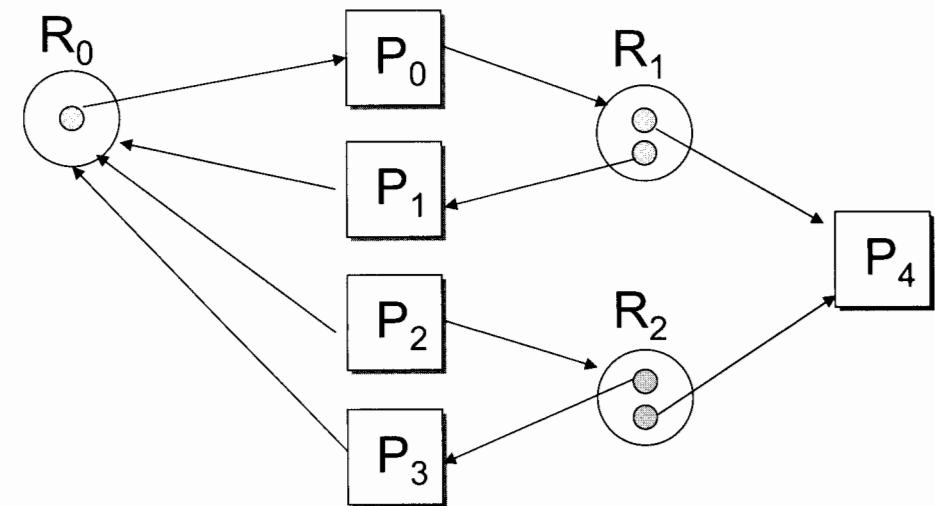


Figura 3-8. Grafo de procesos y recursos.

#### Solución

En el grafo que muestra la figura 3-8 contiene un ciclo:  $P_0 \rightarrow R_1 \rightarrow P_1 \rightarrow R_0 \rightarrow P_0$

Sin embargo existe un camino que sale del ciclo, es decir, que nos lleva a un nodo que no pertenece al ciclo: desde  $R_1$  podemos ir a  $P_4$  y  $P_4$  no pertenece al ciclo anterior. Por consiguiente en el sistema que refleja el grafo de la figura 3-8 no existe interbloqueo.

## EJERCICIO 9

Sea un sistema formado por tres procesos: P1, P2 y P3; y los recursos siguientes: una impresora, dos unidades de disco y una cinta. En un instante dado se tiene la siguiente situación:

- El proceso P1 posee uno de los discos R2 y solicita la cinta
- El proceso P2 posee uno de los discos R2, la cinta y solicita la impresora.
- El proceso P3 posee la impresora y solicita un disco.

Se pide:

- a) Ilustrar dicha situación mediante un grafo de asignación de recursos.
- b) Determinar si existe interbloqueo

*Solución*

a) En la figura 3-9 se muestra la situación del sistema indicada.

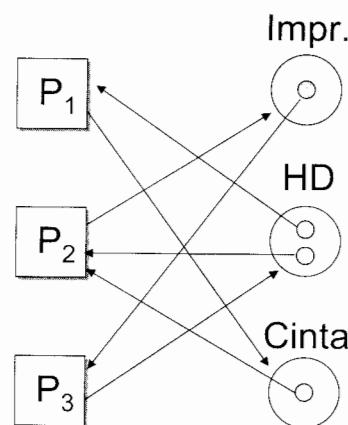


Figura 3-9. Grafo de procesos y recursos.

b) En la figura 3-9 se observa la existencia de un ciclo:

$$P_1 \rightarrow \text{Cinta} \rightarrow P_2 \rightarrow \text{Impr.} \rightarrow P_3 \rightarrow \text{HD} \rightarrow P_1$$

Además, todos los caminos que salen de nodos del ciclo nos llevan de nuevo al ciclo, por tanto es condición suficiente para la existencia de interbloqueo.

## EJERCICIO 10

Indica si en el grafo de procesos y recursos que representa la figura 3-10 existe interbloqueo.

- Si no existe, explica la secuencia de ejecución con la que terminen todos los procesos.
- Si existe, explica un mecanismo para romper el interbloqueo y aplícalo al grafo de la pregunta.

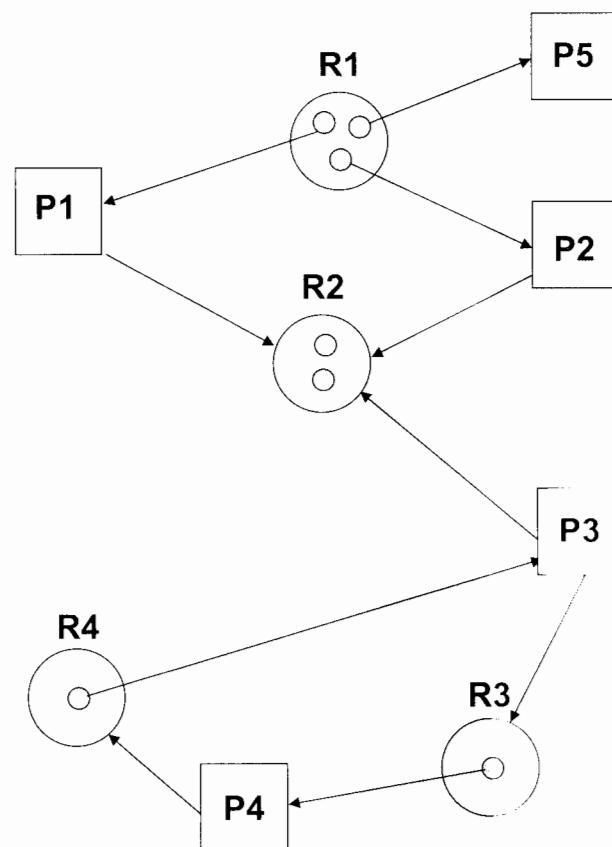


Figura 3-10. Grafo de procesos y recursos.

**Solución**

En el grafo que muestra la figura 3-10 contiene un ciclo:  $P_0 \rightarrow R_1 \rightarrow P_1 \rightarrow R_0 \rightarrow P_0$ . Sin embargo existe un camino que sale del ciclo, es decir, que nos lleva a un nodo que no pertenece al ciclo: desde  $P_3$  podemos ir a  $R_3$  y  $R_2$  no pertenece al ciclo anterior. Por consiguiente en el sistema que refleja el grafo de la figura 3-10 no existe interbloqueo.

**EJERCICIO 11**

Sea un sistema formado por 7 procesos y los siguientes recursos: 1 recurso  $R_1$ , 2 recursos  $R_2$ , 1 recurso  $R_3$ , 3 recursos  $R_4$  y 1 recurso  $R_5$ . En un instante dado se tiene la situación que refleja la tabla 3-8:

Tabla 3-8. Asignaciones y solicitudes de recursos a procesos.

	<b>R1</b>	<b>R2</b>	<b>R3</b>	<b>R4</b>	<b>R5</b>
<b>P1</b>	S	S	S	A	A
<b>P2</b>		A	S		
<b>P3</b>		A		A	
<b>P4</b>		S	S	A	S
<b>P5</b>					S
<b>P6</b>	A	S	A	S	S
<b>P7</b>		S	S	S	

A: Asignado  
S: Solicitado

Se pide:

Ilustrar dicha situación mediante un grafo de asignación de recursos.  
Determinar si existe interbloqueo

**Solución**

a) En la figura 3-11 se muestra la situación del sistema indicada.

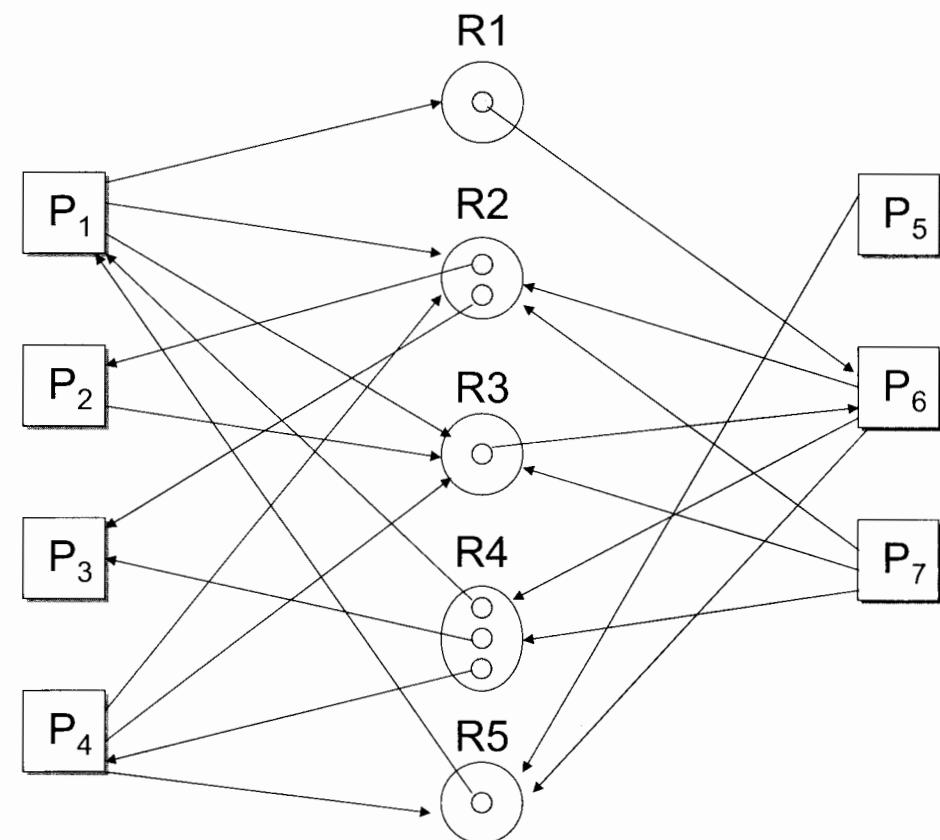


Figura 3-11. Grafo de procesos y recursos.

b) En la figura anterior se observan los siguientes ciclos:

Ciclo 1:  $P_6 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_6$

Ciclo 2:  $P_1 \rightarrow R_1 \rightarrow P_6 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_6 \rightarrow R_4 \rightarrow P_1$

Ciclo 3:  $P_4 \rightarrow R_3 \rightarrow P_6 \rightarrow R_4 \rightarrow P_4$

Ciclo 4:  $P_4 \rightarrow R_3 \rightarrow P_6 \rightarrow R_5 \rightarrow P_1 \rightarrow R_1 \rightarrow P_6 \rightarrow R_4 \rightarrow P_4$

En algunos de ellos se observa que no hay caminos que salen del ciclo, por tanto, hay interbloqueo. No es posible encontrar una secuencia lógica de ejecución que haga que todos los procesos terminen.

## EJERCICIO 12

Un sistema formado por 7 procesos y los siguientes recursos: 2 recursos R1, 2 recursos R2, 1 recurso R3, 3 recursos R4, 3 recursos R5, 1 recurso R6 y 2 recursos R7. En un instante dado se tiene la situación que refleja la tabla 3-9:

Tabla 3-9. Asignaciones y solicitudes de recursos a procesos.

	R1	R2	R3	R4	R5	R6	R7
P1	A	S	S	S		A	A
P2		A	A				
P3		S	S	A	S		
P4	A	S	S	A			
P5			S	A	A	S	S
P6	S	A		S	A	S	S
P7	S	S		S	A	S	A

A:Asignado  
S:Solicitado

Se pide:

- Ilustrar dicha situación mediante un grafo de asignación de recursos.
- Determinar si existe interbloqueo

## Solución

- a) En la figura siguiente se muestra la situación del sistema indicada por la tabla 3-9:

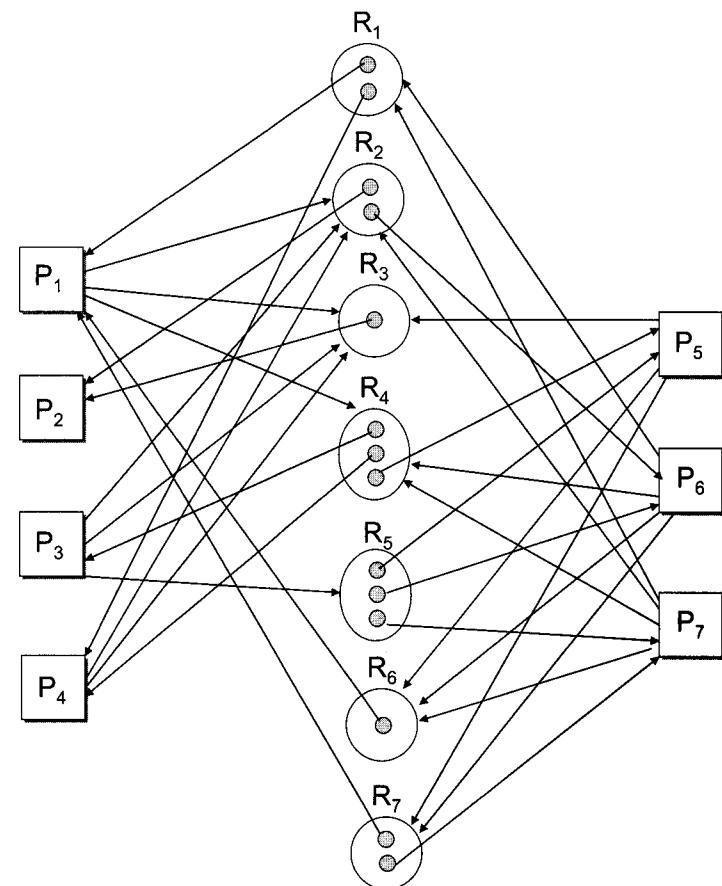


Figura 3-12. Grafo de procesos y recursos.

- b) En la figura anterior varios ciclos, algunos de ellos son los siguientes:  
 Ciclo 1: P1 → R2 → P6 → R6 → P1  
 Ciclo 2: P1 → R2 → P6 → R1 → P1  
 Ciclo 3: R2 → P6 → R1 → P4 → R2  
 Ciclo 4: P1 → R4 → P3 → R2 → P6 → R6 → P1

Sin embargo, en este caso, a pesar de los múltiples ciclos no hay interbloqueo porque hay caminos que salen de todos los ciclos a nodos que están fuera del ciclo.

Una secuencia lógica con la que terminan todos los procesos es la siguiente:

El proceso P2 tiene todos los recursos que necesita para terminar. Cuando termina libera los recursos que posee. Estos recursos se pueden asignar al proceso P4 que solicita ambos. En ese momento el proceso P4 tiene todos los recursos que necesita y también termina. A partir de ese momento P1 también puede obtener todos los recursos que necesita y termina. A continuación puede terminar P7, P6, P5 y finalmente P3.

#### EJERCICIO 13

Indica si en el grafo de procesos y recursos que representa la figura 3-13 existe interbloqueo.

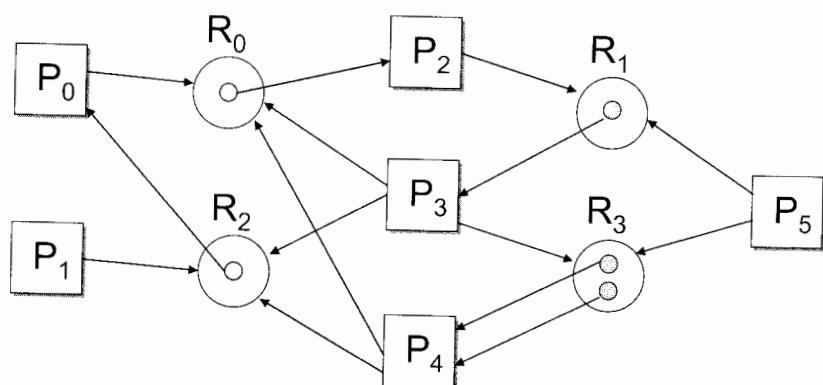


Figura 3-13. Grafo de procesos y recursos.

Si existe interbloqueo, indica qué proceso habría que reiniciar para romper el interbloqueo y cuál sería la secuencia de terminación de los procesos.

#### Solución

En la figura 3-13 existen varios ciclos:

Ciclo 1:  $R_0 \rightarrow P_2 \rightarrow R_1 \rightarrow P_3 \rightarrow R_0$

Ciclo 2:  $R_0 \rightarrow P_2 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_0 \rightarrow R_0$

Ciclo 3:  $R_0 \rightarrow P_2 \rightarrow R_1 \rightarrow P_3 \rightarrow R_3 \rightarrow P_4 \rightarrow R_0$

Ciclo 4:  $R_0 \rightarrow P_2 \rightarrow R_1 \rightarrow P_3 \rightarrow R_3 \rightarrow P_4 \rightarrow R_2 \rightarrow P_0 \rightarrow R_0$

Hay algún ciclo en el que no hay ningún camino que salga del mismo, por tanto, hay interbloqueo.

Para romper el interbloqueo hay que reiniciar P3. Se liberará el recurso que éste tenía y se podrá asignar a P2. En caso de que se asignara a P5 habría que reiniciar otro proceso. Tras P2 termina P0, P1, P4, P5 y finalmente P3.

#### EJERCICIO 14

Sea un sistema formado por 3 procesos y los siguientes recursos: 2 recursos R1, 2 recursos R2, 1 recurso R3. En un instante dado se tiene la situación de la tabla 3-10:

Tabla 3-10. Asignaciones y solicitudes de recursos a procesos.

	R1	R2	R3
P1	A	S	
P2	A	A	
P3	S	A	A

A:Asignado  
S:Solicitado

Se pide:

- Ilustrar dicha situación mediante un grafo de asignación de recursos.
- Determinar si existe interbloqueo

*Solución*

a) En la figura siguiente se muestra la situación del sistema indicada por la tabla 3-10:

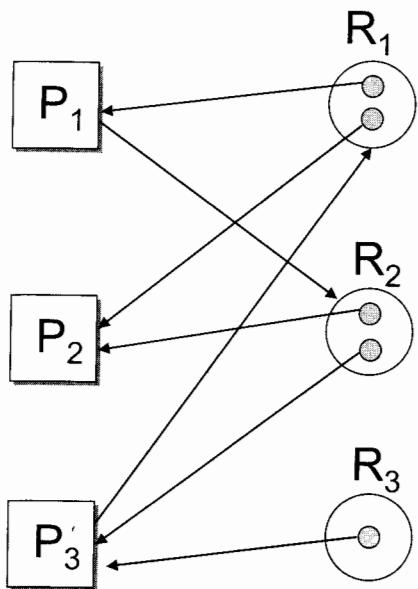


Figura 3-14. Grafo de procesos y recursos.

b) Se observa que existe un ciclo:  $P_1 \rightarrow R_2 \rightarrow P_3 \rightarrow R_1 \rightarrow P_1$

Sin embargo, existe un arco que sale del ciclo y conduce a un nodo ( $P_2$ ) que no pertenece al ciclo, por lo que no hay interbloqueo.

**EJERCICIO 15**

Sea un sistema en el que todos los recursos están asignados a un mismo proceso. ¿Se puede dar en ese sistema una situación de interbloqueo?

*Solución*

Una de las condiciones para la existencia de interbloqueo es la de espera circular. Esta condición significa que los procesos retienen recursos y están esperando por recursos retenidos por otros procesos. Sea cual sea el número

de procesos y recursos de un sistema, si todos los recursos están asignados al mismo proceso no se puede dar esa situación de espera circular, y por tanto no habrá interbloqueo.

La figura siguiente ilustra esta situación para un sistema con tres procesos y tres recursos:

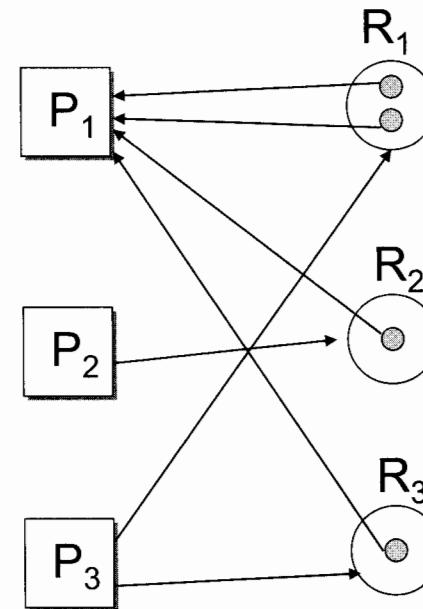


Figura 3-15. Grafo de procesos y recursos.

Como se aprecia en la figura anterior, independientemente de las peticiones de los otros procesos, no se podrá producir un ciclo en el grafo que representa el estado de ese sistema.

**PARTE II.**

**PRÁCTICAS**

## CAPÍTULO 4.

# CREACIÓN Y TERMINACIÓN DE PROCESOS

### 4.1 INTRODUCCIÓN

Uno de los objetivos de los sistemas operativos es hacer que el computador se más fácil de utilizar. Para ello el sistema operativo ofrece un conjunto de operaciones que actúan a diferentes niveles (procesos, archivos, comunicación, etc.). Estas operaciones se pueden clasificar en dos grupos:

- Llamadas al sistema que proporcionan la interfaz a nivel del programador. Son operaciones básicas que facilita el sistema operativo para realizar programas. Por ejemplo, crear un proceso, abrir un archivo, crear un tubo, etc.
- Órdenes o comandos de shell que proporcionan la interfaz a nivel de usuario. Normalmente son programas más elaborados construidos a partir de las llamadas al sistema anteriores. Por ejemplo, copiar un archivo, crear un directorio, imprimir un archivo, etc.

En este capítulo estudiaremos algunas de las llamadas al sistema relacionadas con la creación y terminación de procesos. Hemos incorporado al final del capítulo un anexo que recoge en una tabla las llamadas al sistema Unix.

### 4.2 CREACIÓN DE PROCESOS

#### 4.2.1 *Llamada fork*

En Unix, un proceso es creado mediante la llamada del sistema fork. El proceso que realiza la llamada se denomina proceso padre (parent process) y el proceso creado a partir de la llamada se denomina proceso hijo (child process). La sintaxis de la llamada efectuada desde el proceso padre es: pid=fork(); La llamada fork se realiza una sola vez, pero retorna dos veces (correspondientes a los procesos padre e hijo). Las acciones implicadas por la petición de un fork son realizadas por el núcleo (kernel) del sistema operativo. Tales acciones son las siguientes:

1. Asignación de un hueco en la tabla de procesos para el nuevo proceso (hijo).
2. Asignación de un identificador único (PID) al proceso hijo.
3. Copia de la imagen del proceso padre (con excepción de la memoria compartida).
4. Asignación al proceso hijo del estado "preparado para ejecución".
5. Dos valores de retorno de la función: al proceso padre se le entrega el PID del proceso hijo, y al proceso hijo se le entrega el valor cero.

A la vuelta de la llamada fork, los dos procesos poseen copias iguales de sus contextos a nivel usuario y sólo difieren en el valor del PID devuelto. La indicada relación de operaciones del fork se ejecuta en modo núcleo en el proceso padre. Al concluir, el planificador de procesos pondrá en ejecución un proceso que puede ser:

- el mismo proceso padre: vuelta al modo usuario y al punto en el que quedó al hacer la llamada fork.
- el proceso hijo: éste comenzará a ejecutarse en el mismo punto que el proceso padre, es decir, a la vuelta de la llamada fork.
- cualquier otro proceso que estuviese "preparado para ejecución" (en tal caso, los procesos padre e hijo permanecen en el estado "preparado para ejecución").

Las dos primeras situaciones se distinguen gracias al parámetro devuelto por fork (cero identifica al proceso hijo, no cero al proceso padre).

La ejecución del siguiente programa creaproc.c y la posterior petición de información sobre procesos en ejecución permitirá comprobar el funcionamiento de la llamada fork.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int pid;

    pid = fork();
    switch (pid)
    {
        case -1:
            printf("No he podido crear el proceso hijo \n");
            break;
        case 0:
            printf("Soy el hijo, mi PID es %d y mi PPID es %d \n", getpid(),
getppid());
            sleep (20);
            break;
        default:
            printf ("Soy el padre, mi PID es %d y el PID de mi hijo es %d
\n",getpid(), pid);
            sleep (30);
    }
    printf ("Final de ejecución de %d \n", getpid());
    exit (0);
}
```

Para ejecutar el programa en modo background, y así poder verificar su ejecución con la orden ps, se debe añadir al nombre del programa el carácter & (ampersand):

```
$ gcc -o creaprocreaprocc
$ creaprocc &
Soy el hijo, mi PID es 944 y mi PPID es 943
Final de ejecución de 944
Soy el padre, mi PID es 943 y el PID de mi hijo es 944
Final de ejecución de 943
$ ps
  PID TTY      TIME CMD
 893 pts/1    00:00:00 bash
 943 pts/1    00:00:00 creaprocc
 944 pts/1    00:00:00 creaprocc
 946 pts/1    00:00:00 ps
$
```

Dos procesos vinculados por una llamada fork poseen zonas de datos propias, de uso privado (no compartidas). La ejecución del siguiente programa, en el cual se asigna distinto valor a una misma variable según se trate de la ejecución del proceso padre o del hijo, permitirá comprobar tal característica de la llamada fork.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main ( )
{
    int i;
    int j;
    int pid;

    pid = fork( );
    switch (pid) {
    case -1:
        printf ("\nNo he podido crear el proceso hijo");
        break;
    case 0:
        i = 0;
        printf("\n");
        printf("Soy el hijo, mi PID es %d y mi variable i (inicialmente a %d) es
par",getpid(),i);
        for ( j = 0; j < 5; j ++ ) {
            i++;
            i++;
            printf ("\nSoy el hijo, mi variable i es %d", i);
        }
        break;
    default:
        i = 1;
        printf("\n Soy el ");
        printf("padre, mi PID es %d y mi variable i (inicialmente a %d) es
impar",getpid(),i);
        for ( j = 0; j < 5; j ++ ) {
            i++;
            i++;
            printf ("\nSoy el padre, mi variable i es %d", i);
        };
        printf ("\nFinal de ejecucion de %d \n", getpid());
        exit (0);
    }
}
```

En este programa, el proceso padre visualiza los sucesivos valores impares que toma su variable *i* privada, mientras el proceso hijo visualiza los sucesivos valores pares que toma su variable *i* privada y diferente a la del proceso padre.

```
$ gcc -o creaproci creaproci.c
$ creaproci &

Soy el padre, mi PID es 956 y mi variable i (inicialmente a 1) es imp
Soy el padre, mi variable i es 3
Soy el padre, mi variable i es 5
Soy el padre, mi variable i es 7
Soy el padre, mi variable i es 9
Soy el padre, mi variable i es 11
Final de ejecucion de 956

Soy el hijo, mi PID es 957 y mi variable i (inicialmente a 0) es par
Soy el hijo, mi variable i es 2
Soy el hijo, mi variable i es 4
Soy el hijo, mi variable i es 6
Soy el hijo, mi variable i es 8
Soy el hijo, mi variable i es 10
Final de ejecucion de 957
$
```

Las variables descriptores, asociadas a archivos abiertos en el momento de la llamada fork, son compartidas por los dos procesos existentes a la vuelta del fork. Es decir, los descriptores de archivos en uso por el proceso padre son heredados por el proceso hijo generado. El siguiente programa en el cual los dos procesos escriben distintos strings en los mismos archivos, permitirá comprobar tal característica de la llamada fork.

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main ( ) {
    int i;
    int fd1, fd2;
    const char string1[10] = "*****";
    const char string2[10] = "-----";
    int pid;

    fd1 = creat ("archivoA", 0666);
    fd2 = creat ("archivoB", 0666);
    pid = fork ();
    switch (pid) {
        case -1:
            printf ("\nNo he podido crear el proceso hijo");
            break;
        case 0:
            for ( i = 0; i < 10; i ++ ) {
                write (fd1, string2, sizeof(string2));
                write (fd2, string2, sizeof(string2));
                usleep(1); /* Abandonamos voluntariamente el procesador */
            };
            break;
        default:
            for ( i = 0; i < 10; i ++ ) {
                write (fd1, string1, sizeof(string1));
                write (fd2, string1, sizeof(string1));
                usleep(1); /* Abandonamos voluntariamente el procesador */
            };
    }
}

```

```
    printf ("\nFinal de ejecucion de %d \n", getpid( ));  
    exit (0);  
}
```

Para ejecutar este programa, no es preciso más que compilarlo. Después verificar los resultados del mismo usando el mandato cat:

Un proceso no sabe a priori su nombre y ni el de su padre (aunque lo tenga en sus estructuras internas). Para obtener el identificador de un proceso o el de su padre hay que acudir a las llamadas `getpid()` y `getppid()`.

#### 4.2.2 Llamada getpid

La llamada `getpid()` devuelve el número de identificación de proceso (PID) del proceso actual. Devuelve -1 en caso de error y el PID del proceso en caso de éxito.

### 4.2.3 Llamada `getppid`

La llamada `getppid()` devuelve el número de identificación de proceso (PID) del proceso padre al proceso actual (el PID del proceso que creó al actual). Devuelve -1 en caso de error y el PID del padre en caso de éxito.

### 4.3 TERMINACIÓN DE PROCESOS

### 4.3.1 Llamada exit

La llamada `exit()` hace finalizar un proceso con estado `estado`. Este estado es un valor entero que se retorna al padre del proceso finalizado. Para que el padre puede leer este valor, deberá realizar una llamada `wait()`.

Si un proceso finaliza sin ejecutar la llamada `exit()`, por ejemplo, al realizar un retorno de la función `main()`, su estado de finalización será indefinido.

El sistema tiene una variable de entorno, denominada `?`, donde puede recogerse el estado de finalización de la ejecución de la última orden. Para ver el estado de finalización del último hijo de shell (última orden) hay que introducir

```
$ echo $?
...
```

A lo que el sistema contestará con 0, 2, o cualquier estado posible.

#### 4.3.2 Llamada `wait`

La llamada `wait()` hace que el proceso actual quede detenido en espera hasta que muere alguno de sus hijos o recibe una señal. Devuelve `-1` en caso de error y el PID del hijo que muere en caso contrario.

Un proceso puede finalizar por dos causas: una terminación explícita mediante la ejecución de una llamada `exit()`, o bien porque otro proceso lo mató enviándole una señal.

El parámetro `estado` es un entero en el que `wait()` escribe información dividida en dos partes:

Si el hijo termina con una llamada `exit(estado)`, en los 7 bits menos significativos (`estado & 0x7F`) escribe 0 y en los 8 bits más significativos (`((estado>>8) & 0xFF)`) escribe el valor `estado`.

Si el hijo termina matado por una señal, en los 7 bits menos significativos (`estado & 0x7F`) escribe el número de señal.

(estado>>8) & 0xFF								estado & 0x7F																							
valor exit								núm. Señal (0 si exit)																							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main ( ) {
    int estado, numero;

    switch(fork()) {
        case -1 : /* ERROR */
            perror("Error en fork");
            exit(1);
        case 0 : /* HIJO */
            numero = 13;
            printf("Soy el hijo y muero con %d...\n", numero);
            sleep(20);
    }
}
```

```
        exit(numero);
    default : /* PADRE */
        wait(&estado);
        printf("Soy el padre. ");
        if ((estado & 0x7F) != 0) {
            printf("Mi hijo ha muerto con una señal.\n");
        } else {
            printf("Mi hijo ha muerto con exit(%d).\n",
                   (estado>>8) & 0xFF);
        }
    }
    exit(0);
}
```

Si compilamos y ejecutamos obtenemos lo siguiente:

```
$ gcc -o creaproc3 creaproc3.c
$ creaproc3
Soy el hijo y muero con 13...
Soy el padre. Mi hijo ha muerto con exit(13).
$ creaproc3 &
[1] 1006
Soy el hijo y muero con 13...
$ ps
UID      PID  PPID  C STIME TTY          TIME CMD
alu      893   891  0 18:06 pts/1    00:00:00 /bin/bash
alu     1006   893  0 18:34 pts/1    00:00:00 creaproc3
alu     1007  1006  0 18:34 pts/1    00:00:00 creaproc3
alu     1008   893  0 18:34 pts/1    00:00:00 ps -f
$ kill -9 1007
Soy el padre. Mi hijo ha muerto con una señal.
$
```

## 4.4 EJERCICIOS

### 4.4.1 Ejercicio 1

Realizar un programa llamado `hfork.c` que cree `n` hijos con una estructura horizontal donde cada hijo mostrará su número, su PID y su PPID y el padre mostrará su PID y el identificador del hijo que ha terminado.

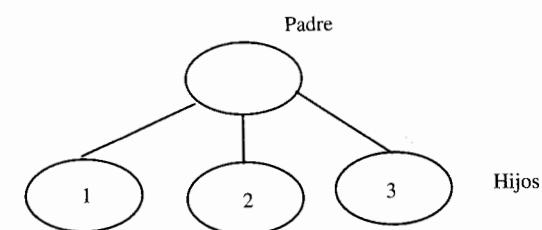


Figura 4-1. Relación padres-hijos. Ejercicio 1.

El número de hijos se pasa como argumento en la línea de órdenes.  
El hijo antes de ejecutar `exit` deberá esperar 20 segundos.

Un ejemplo de ejecución es el siguiente:

```
$ hfork 3
Soy el hijo 1. Mi pid es 850 y mi ppid es 849
Soy el hijo 2. Mi pid es 851 y mi ppid es 849
Soy el hijo 3. Mi pid es 852 y mi ppid es 849
Soy el padre con pid 849 y mi hijo 1 ha terminado
Soy el padre con pid 849 y mi hijo 2 ha terminado
Soy el padre con pid 849 y mi hijo 2 ha terminado
$
```

### Solución

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int pid;
    int numhijos, i, estado;

    if (argc != 2) {
        printf("USO: hfork num_hijos\n");
        exit(-1);
    }
    else {
        numhijos = atoi(argv[1]);

        for (i = 1; i <= numhijos; i++) {
            pid = fork();
            if (pid == 0) {
                printf("Soy el hijo %d. Mi pid es %d y mi ppid es %d\n", i, getpid(), getppid());
                sleep(20);
                exit(0);
            }
            for (i = 1; i <= numhijos; i++) {
                wait(&estado);
                printf("Soy el padre con pid %d y mi hijo %d ha terminado\n", getpid(), i);
            }
        }
    }
    return 0;
}
```

### 4.4.2 Ejercicio 2

Realizar un programa llamado vfork.c que cree n hijos con una estructura vertical donde cada proceso mostrará su número, su PID y su PPID.

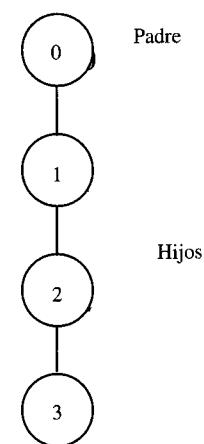


Figura 4-2. Relación padres-hijos. Ejercicio 2.

El número de hijos se pasa como argumento en la línea de órdenes. El hijo antes de ejecutar exit deberá esperar 20 segundos.

Un ejemplo de ejecución es el siguiente:

```
$ vfork 4
hijo: 0 pid: 850 ppid: 849
hijo: 1 pid: 851 ppid: 850
hijo: 2 pid: 852 ppid: 851
hijo: 3 pid: 853 ppid: 852
$
```

### Solución

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int pid;
    int i, numhijos;

    if (argc != 2) {
        printf("USO: vfork num_hijos\n");
        exit(-1);
    }
    else {
        numhijos = atoi(argv[1]);

        for (i = 0; i < numhijos; i++) {
            pid = fork();
            if (pid == 0) {
                printf("hijo: %d pid: %d ppid: %d\n", i, getpid(), getppid());
                sleep(20);
                exit(0);
            }
        }
    }
    return 0;
}
```

```

    }
}

return 0;
}

```

#### 4.4.3 Ejercicio 3

Realizar un programa llamado `mfork.c` que al ejecutarlo genere un hijo que haga un `sleep` de `n` segundos y produzca un `exit` con el segundo parámetro que pasamos al `main`.

Un ejemplo de ejecución es el siguiente:

```
$ mfork 10 2
Mi hijo con PID 850 ha muerto con estado 2
$
```

En el caso que se le envíe una señal al hijo, el padre deberá mostrar

```
Mi hijo con PID 851 ha muerto por la señal 9
```

#### Solución

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int pid;
    int seg, est, estado, pid_hijo;

    if (argc != 3) {
        printf("USO: mfork segundos estado\n");
        exit(-1);
    } else {
        seg = atoi(argv[1]);
        est = atoi(argv[2]);

        pid = fork();
        switch (pid) {
            case -1: /* ERROR */
                printf("No he podido crear el proceso hijo\n");
                exit(-1);
            case 0: /* PROCESO HIJO */
                sleep(seg);
                exit(est);
            default: /* PROCESO PADRE */
                wait(&estado);
                if ((estado & 0x7F) != 0)
                    printf("Mi hijo con PID %d ha muerto por la señal %d\n", pid, (estado
& 0x7F));
                else
                    printf("Mi hijo con PID %d ha muerto con estado %d\n", pid, (estado >
8) & 0xFF);
                break;
        }
    }
    return 0;
}

```

#### APÉNDICE 1: TABLA DE LLAMADAS AL SISTEMA

Llamada	Categoría	Descripción
alarm()	Comunicación	Programa una alarma con un plazo de un cierto número de segundos.
kill()	Comunicación	Envía una señal a un proceso.
pause()	Comunicación	Detiene la ejecución indefinidamente (hasta la llegada de una señal).
pipe()	Comunicación	Crea una tubería y proporciona sus descriptores.
signal()	Comunicación	Programa un manejador para una señal.
chdir()	Archivos	Cambia el directorio por defecto (directorio actual).
chmod()	Archivos	Cambia los permisos de un archivo.
chown()	Archivos	Cambia el propietario de un archivo.
close()	Archivos	Cierra un archivo.
creat()	Archivos	Crea un archivo y lo abre para escritura.
dup()	Archivos	Duplica un descriptor de archivo en la siguiente entrada libre de la tabla de archivos.
dup2()	Archivos	Duplica un descriptor de archivo en la entrada de la tabla de archivos que se indique.
fstat()	Archivos	Obtiene información sobre un archivo abierto: tamaño, fecha, permisos, etc.
link()	Archivos	Crea un enlace (duro).
lseek()	Archivos	Sitúa el puntero de lectura/escritura en cualquier posición de un archivo.
mkdir()	Archivos	Crea un directorio.
mknod()	Archivos	Crea un dispositivo especial.
mount()	Archivos	Monta un dispositivo en un directorio.
open()	Archivos	Abre un archivo y proporciona un descriptor.
read()	Archivos	Lee datos de un archivo.
rmdir()	Archivos	Elimina un directorio vacío.
stat()	Archivos	Obtiene información sobre un archivo: tamaño, fecha, permisos, etc.
symlink()	Archivos	Crea un enlace simbólico.
umount()	Archivos	Desmonta un dispositivo.

Llamada	Categoría	Descripción
unlink()	Archivos	Elimina un enlace duro de un archivo (lo borra del disco).
utime()	Archivos	Modifica la fecha y hora de acceso y modificación de un archivo.
write()	Archivos	Escribe datos de un archivo.
getgid()	Otras	Proporciona el identificador de grupo del usuario actual (GID).
getuid()	Otras	Proporciona el identificador del usuario actual (UID).
setgid()	Otras	Cambia el identificador de grupo del usuario actual (GID).
setuid()	Otras	Cambia el identificador del usuario actual (UID).
time()	Otras	Proporciona el número de segundos desde el 1 de enero de 1970 a las 0 horas.
times()	Otras	Proporciona el tiempo consumido en la ejecución del proceso actual.
execXX()	Procesos	Cambia el programa actual por el programa en disco indicado (ejecuta un programa).
exit()	Procesos	Termina el proceso actual indicando un estado de finalización.
fork()	Procesos	Crea un proceso hijo (desdobra el proceso actual).
getpid()	Procesos	Proporciona el PID del proceso actual.
getppid()	Procesos	Proporciona el PID del proceso padre.
nanosleep()	Procesos	Detiene la ejecución en espera pasiva durante un número de nanosegundos.
sleep()	Procesos	Detiene la ejecución en espera pasiva durante un número de segundos.
usleep()	Procesos	Detiene la ejecución en espera pasiva durante un número de microsegundos.
wait()	Procesos	Espera por la muerte de un hijo (cualquiera) y obtiene su estado de finalización.

## CAPÍTULO 5. EJECUCIÓN DE PROCESOS

### 5.1 INTRODUCCIÓN

Los sistemas operativos multiproceso permiten ejecutar varios procesos simultáneamente. Para ello deben proporcionar un conjunto de llamadas al sistema que no sólo puedan crear y eliminar procesos, sino también cambiar su imagen, esto es, cambiar su código, memoria, etc.

En este capítulo vamos a estudiar cómo se puede cambiar la imagen de un proceso para que se pueda ejecutar otro programa

### 5.2 LLAMADA EXEC

Los procesos Unix pueden modificarse usando la llamada al sistema exec. A diferencia de fork (donde a la vuelta de la llamada existen dos procesos: el invocador o padre y el generado o hijo), la llamada exec produce la sustitución del programa invocador por el nuevo programa invocado. La ejecución del proceso continúa siguiendo el nuevo programa.

fork crea nuevos procesos, exec sustituye la imagen de memoria del proceso por otra nueva (sustituye todos los elementos del proceso: código del programa, datos, pila, montículo).

Las opciones de las distintas versiones de exec se pueden entender a través de las letras añadidas a exec en el nombre de las llamadas, como podemos ver en la siguiente tabla.

Tabla 5-1: Opciones de la llamada exec.

Letra	Significado
l	Los argumentos pasados al programa se especifican como una lista parámetros de tipo cadena, terminada con un parámetro NULL (que indica que la lista ha terminado).
v	Los argumentos pasados al programa se especifican como un vector de punteros a carácter (cadenas), terminado con un puntero nulo (que indica que el vector ha terminado).

Letra	Significado
p	Indica que para encontrar el programa ejecutable hay que utilizar la variable PATH definida, siguiendo el algoritmo de búsqueda de UNIX. Por lo tanto no es necesario especificar una vía absoluta al archivo, sino únicamente el nombre del ejecutable.
e	Indica que las variables de entorno que utilizará el programa serán las que se indique en el parámetro correspondiente mediante un vector de punteros a carácter (cadenas) terminadas en un puntero nulo, y no las variables que utiliza el proceso actual.

El nuevo programa activado mantiene el mismo PID así como otras propiedades asociadas al proceso. Sin embargo, el tamaño de memoria de la imagen del proceso cambia dado que el programa en ejecución es diferente.

Por ejemplo, si el proceso padre quiere ejecutar el comando ls

```
main() {
    execvp("ls", "ls", "-a", "-l", NULL);
    perror("Error al ejecutar comando");
}
```

Si por el contrario el padre crea un hijo para que ejecute el comando ls

```
main() {
    printf("Este es el listado de archivos\n");
    switch(fork()) {
        case -1 : /* ERROR */
            perror("Error en fork");
            exit(1);
        case 0 : /* HIJO */
            execvp("ls", "ls", "-la", NULL);
            perror("Error en exec");
            exit(-1);
        default : /* PADRE */
            wait(&statusp);
            printf("Esto ha sido todo\n");
            exit(0);
    }
}
```

También podemos crear un programa y llamarlo desde otro. El código fuente del primer programa a ejecutar, prog1.c, es el siguiente:

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
    int i;
    printf ("\nEjecutando el programa invocador (prog1). Sus argumentos son:
\n");
    for ( i = 0; i < argc; i ++ )
        printf ("    argv[%d] : %s \n", i, argv[i]);
    sleep( 10 );
    strcpy (argv[0], "prog2");
}
```

```
if (execvp ("./prog2", argv) < 0) {
    printf ("Error en la invocación a prog2 \n");
    exit (1);
}
exit (0);
}
```

El código fuente del segundo programa a ejecutar, prog2.c, es el siguiente:

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
    int i;
    printf ("Ejecutando el programa invocado (prog2). Sus argumentos son: \n");
    for ( i = 0; i < argc; i ++ )
        printf ("    argv[%d] : %s \n", i, argv[i]);
    sleep(10);
    exit (0);
}
```

Será necesario compilar ambos programas, usando la orden gcc. Tras ello, se estará en condición de ejecutarlos:

```
$ gcc -o prog2 prog2.c
$ gcc -o prog1 prog1.c
```

Para ejecutar el programa en modo background y así poder verificar su ejecución con la orden ps, se debe añadir al nombre del programa el carácter & (ampersand):

```
$ prog1 p1 p2 param3 &
Ejecutando el programa invocador (prog1). Sus argumentos son:
    argv[0] : prog1
    argv[1] : p1
    argv[2] : p2
    argv[3] : param3
$ ps -f
   UID  PID  PPID  C   STIME   TTY  TIME CMD
  alu 8736 9176  0 10:23:32 pts/0  0:00 prog1 p1 p2 param3
  alu 9176 6328  1 10:21:21 pts/0  0:00 -ksh
  alu 9298 9176  7 10:23:36 pts/0  0:00 ps -f
$ Ejecutando el programa invocado (prog2). Sus argumentos son:
    argv[0] : prog2
    argv[1] : p1
    argv[2] : p2
    argv[3] : param3
$ ps -f
   UID  PID  PPID  C   STIME   TTY  TIME CMD
  alu 8736 9176  0 10:23:32 pts/0  0:00 prog2 p1 p2 param3
  alu 9176 6328  1 10:21:21 pts/0  0:00 -ksh
  alu 9298 9176  5 10:23:46 pts/0  0:00 ps -f
$
```

### 5.3 EJERCICIOS

#### 5.3.1 Ejercicio 1

Realizar un programa llamado ejecutals.c que cree un hijo y éste ejecute la orden "ls -a -l /" que permite visualizar el contenido del directorio raíz (/).

El tipo de llamada exec a ejecutar se especificará en la línea de órdenes:

- a. execl
- b. execlp
- c. execle

Si no se especifica ningún parámetro, se deberá mostrar un mensaje de error.

```
$ ejecutals a
...
$ ejecutals b
...
$ ejecutals c
...
```

El padre debe mostrar si el hijo ha terminado correctamente o no:

Si no hay error, "Mi hijo con PID YY ha terminado satisfactoriamente"

Si hay error, "Mi hijo con PID YY ha finalización con el estado XX"

#### Solución

```
#include <ctype.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main( int argc, char** argv, char** env )
{
    char OPCION;

    if( argc != 2 )
    {
        perror( "Uso: ejecutals [abc]\n a .. execl\n b .. execlp\n c .."
                "execle\n" );
        exit( -1 );
    }
    else
    {
        OPCION = tolower( argv[1][0] );
        if( ( OPCION != 'a' ) && ( OPCION != 'b' ) && ( OPCION != 'c' ) )
            perror( "Error: el tipo de llamada a exec no es válido ([abc])\n" );
        exit( -1 );
    }
}
```

```
switch( fork() )
{
    case -1: /* error */
    {
        perror( "Error en fork\n" );
        exit( -1 );
    }
    case 0: /* hijo */
    {
        switch( OPCION )
        {
            case 'a': /* execl */
            {
                execl( "/bin/ls", "ls", "-a", "-l", "/", NULL );
                break;
            }
            case 'b': /* execlp */
            {
                execlp( "ls", "ls", "-a", "-l", "/", NULL );
                break;
            }
            case 'c': /* execle */
            {
                execle( "/bin/ls", "ls", "-a", "-l", "/", NULL, env );
                break;
            }
            default:
            {
                ; // no hace nada
            }
        }
        perror( "Error en exec\n" );
        exit( -1 );
    }
    default: /* padre */
    {
        int estado, pid;

        pid = wait( &estado );
        if( ( estado & 0x7F ) != 0 )
        {
            printf( "Mi hijo con PID %d ha finalizado con el estado %d\n",
                    pid, ( estado >> 8 ) & 0xFF );
        }
        else
        {
            printf( "Mi hijo con PID %d ha terminado satisfactoriamente\n",
                    pid );
        }
    }
    return 0;
}
```

#### 5.3.2 Ejercicio 2

Realizar un programa llamado ejecuta.c que cree un hijo y ejecute el comando que se le pasa como parámetro.

```
$ ejecuta ls -l -a
...
$ ejecuta find / -name "ejecuta" -print
...
$
```

El padre debe mostrar si el hijo ha terminado correctamente o no:  
 Si no hay error, "Mi hijo con PID YY ha terminado satisfactoriamente"  
 Si hay error, "Mi hijo con PID YY ha finalización con el estado XX"

### Solución

```
#include <ctype.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main( int argc, char** argv )
{
    if( argc == 1 )
    {
        perror( "Uso: ejecuta COMANDO [PARAM1 [PARAM2 [...]]] \n" );
        exit( -1 );
    }
    else
    {
        ; // no hace nada
    }
    switch( fork() )
    {
        case -1: /* error */
        {
            perror( "Error en fork\n" );
            exit( -1 );
        }
        case 0: /* hijo */
        {
            execvp( argv[1], argv + 1 );
            perror( "Error en exec\n" );
            exit( -1 );
        }
        default: /* padre */
        {
            int estado, pid;

            pid = wait( &estado );
            if( ( estado & 0x7F ) != 0 )
            {
                printf( "Mi hijo con PID %d ha finalizado con el estado %d\n",
                        pid, ( estado >> 8 ) & 0xFF );
            }
            else
            {
                printf( "Mi hijo con PID %d ha terminado satisfactoriamente\n",
                        pid );
            }
        }
    }
    return 0;
}
```

## CAPÍTULO 6. SEÑALES

### 6.1 INTRODUCCIÓN

De la misma forma que el núcleo del sistema operativo trabaja mediante interrupciones para sincronizar-comunicar, a nivel de procesos, el sistema ofrece un mecanismo análogo basado en interrupciones software para trabajar con los procesos a nivel de usuario: las señales. A lo largo de este capítulo vamos a estudiar cómo se gestionan las señales (enviar, programar, esperar, etc.).

Cuando a un proceso le envían una señal y no ha activado el manejador de esa señal el proceso se terminará. La forma de capturar una señal es mediante la llamada al sistema signal(). Esta llamada tiene dos parámetros: uno de ellos es la señal que esperamos recibir y el otro el puntero a la rutina de servicio que vamos a ejecutar en el momento que la recibamos. Podemos ayudarnos, a veces, de la llamada pause() que suspende la ejecución del proceso hasta que recibe una señal.

Por ejemplo, el siguiente código programa al manejador de reloj mediante la llamada al sistema alarm() para que nos avise dentro de 5 segundos enviándonos la señal SIGALRM. Una vez recibida, se salta a la rutina de servicio alarma que imprime un mensaje y continúa la ejecución normal del proceso.

```
#include <signal.h>
void alarma()
{
    printf("acabo de recibir un SIGALRM\n");
}

main()
{
    signal(SIGALRM, alarma);
    printf("Acabo de programar la captura de un SIGALRM\n");
    alarm(3);
    printf("Ahora he programado la alarma en 3 seg.\n");
    pause();
    printf("Ahora continúo con la ejecución normal\n");
}
```

El sistema tiene por defecto un par de manejadores para el tratamiento de todas y cada una de las señales: SIG\_DFL y SIG\_IGN. El primero de ellos, llama al manejador por defecto asociado a la señal el cual realiza un exit con el número de señal que recibe, mientras que el segundo de ellos, ignora la señal que llega.

## 6.2 LLAMADA SIGNAL

La llamada prepara a un proceso para recibir una señal, asignando una función manejadora (*handler*) a un tipo de señal. *numero* es el tipo de señal y *funcion* es una función de tipo *void* y con un parámetro entero. Devuelve un puntero a la antigua función manejadora de esa señal en caso de éxito, y *SIG\_ERR* en caso de error.

Cuando un proceso recibe una señal, su ejecución se detiene y se pasa a ejecutar un manejador, que consiste en una función que se ejecuta cuando se recibe esa señal. Cuando esta función termina, el proceso continúa ejecutándose por el mismo punto en que se quedó.

Existen diversas señales, identificadas por un número entero. Cada una de ellas tiene un significado particular. Mediante el archivo *<signal.h>* o bien por medio del comando *kill -l* se pueden identificar las señales del sistema. Las más importantes son las siguientes:

Tabla 5-1: Señales más relevantes.

Nombre	Número	Significado
SIGINT	2	Se ha pulsado CRTL-C
SIGQUIT	3	Se ha pulsado CRTL-/
SIGILL	4	Se ha intentado ejecutar una instrucción no válida.
SIGTRAP	5	Se ha producido una interrupción de traza.
SIGFPE	8	Se ha producido un error en una operación de punto flotante.
SIGKILL	9	Matar proceso.
SIGUSR1	10	Señal de usuario. Libre para ser reprogramada.
SIGUSR2	12	Señal de usuario. Libre para ser reprogramada.
SIGPIPE	13	Utilización de una <i>PIPE</i> que no tiene ningún proceso en el otro extremo.
SIGALRM	14	Vencimiento de alarma programada.
SIGTERM	15	Señal de terminación enviada por el sistema
SIGCHLD	17	Señal que envía un hijo a su padre cuando finaliza

Si un proceso no ha definido explícitamente una función manejadora para una determinada señal, al recibirse esa señal el proceso muere. Por tanto, antes de esperar a recibir una señal debe definirse su manejador. Sin embar-

go, la señal *SIGKILL* no puede ser manejada. Así, si se recibe, matará al proceso incondicionalmente.

La función manejadora es una función de tipo *void* con un parámetro de tipo entero por el cual se indica el número de la señal recibida, de manera que el mismo manejador pueda valer para distintas señales. Existen dos funciones estándar que pueden programarse como manejadores de cualquier señal (excepto *SIGKILL*):

*SIG\_DFL*: es la rutina por defecto, que mata al proceso.

*SIG\_IGN*: que hace que se ignore la señal (no hace nada, pero no mata al proceso).

La programación de una función manejadora sólo sirve una vez. Después de ejecutarse el manejador, se vuelve automáticamente a programar la función por defecto *SIG\_DFL*, por lo que si queremos que sirva para más veces debemos reprogramar el manejador dentro del propio manejador. Esto ocurre en el estándar POSIX. El siguiente ejemplo lo ilustra:

```
#include <signal.h>
void alarma() {
    printf("acabo de recibir un SIGALRM\n");
}
main(){
    signal(SIGALRM, alarma);
    printf("acabo de programar la captura de un SIGALRM\n");
    alarm(3);
    printf("Ahora he programado la alarma en 3 seg.\n");
    pause();
    printf("vuelvo a programar la alarma\n");
    alarm(3);
    pause();
    printf("En POSIX esta línea nunca se ejecutaría porque me ha matado el
SIGALRM\n");
}
```

Al crearse procesos mediante *fork()* la programación de señales en el hijo se hereda de tal forma que las señales programadas con *SIG\_IGN* y *SIG\_DFL* se conservan y las programadas con funciones propias pasan a ser *SIG\_DFL*.

La recepción de señales no se hace necesariamente en el mismo orden en que se envían. Además, no existe una memoria de señales, de manera que si llegan varias señales (iguales) a la vez, sólo se ejecutará el manejador una vez.

## 6.3 LLAMADA KILL

La llamada *kill* permite a un proceso enviar una señal a otro proceso o así mismo a través del PID. PID es el PID del proceso señalado y *numero* es el tipo de señal enviada. Devuelve -1 en caso de error y 0 en caso contrario.

Existe una limitación sobre los procesos a los que pueden enviarse señales: deben tener alguna relación de parentesco del tipo: antecesor, hermano o hijo.

Por ejemplo, el siguiente código sirve para que un proceso se suicide:

```
#include <signal.h>
main() {
    printf("Voy a suicidarme\n");
    kill(getpid(), SIGKILL);
    perror("No he muerto??");
}
```

#### 6.4 LLAMADA ALARM

La llamada `alarm` programa una alarma, de manera que cuando hayan transcurrido `tiempo` segundos, se enviará al proceso una señal de tipo `SIGALRM`. Devuelve 0 si no había otra alarma previamente programada, o el número de segundos que faltaban para cumplirse el tiempo de otra alarma previamente programada.

Sólo es posible programar una alarma por proceso. Al programarse una alarma se desconecta cualquier otra que hubiera previamente.

Un ejemplo de uso de `alarm` es el siguiente: poder imprimir en el terminal durante 5 segundos.

```
#include <signal.h>
int seguir = 1; /* Variable global */
void fin(int n) {
    seguir = 0;
}
main() {
    int contador = 0;
    signal(SIGALRM, fin);
    alarm(5);
    do {
        printf("Esta es la línea %d\n", contador++);
    } while (seguir);
    printf("TOTAL: %d líneas\n", contador);
}
```

#### 6.5 LLAMADA PAUSE

La llamada `pause` detiene la ejecución de un proceso (mediante espera pasiva) hasta que se reciba alguna señal. Devuelve siempre -1 y establece el error *Interrupted system call*.

Este ejemplo crea un hijo y espera su señal.

```
#include <signal.h>
main() {
    switch(fork()) {
        case -1 : /* ERROR */
            perror("Error en fork");
            exit(1);
        case 0 : /* HIJO */
            printf("Hola, soy el hijo. Espero 2 segundos...\n");
            sleep(2);
            kill(getppid(), SIGUSR1);
            printf("Soy el hijo. He señalado a mi padre. Adios.\n");
            exit(0);
        default : /* PADRE */
            printf("Hola, soy el padre y voy a esperar.\n");
            signal(SIGUSR1, SIG_IGN); /* Ignoro señal para no morir */
            pause();
            printf("Soy el padre y ya he recibido la señal.\n");
            exit(0);
    }
}
```

Una señal muy utilizada es `SIGCHLD` que es enviada por todo proceso hijo a su padre en el mismo instante que realiza `exit`. De ésta manera, el padre sabe que su hijo ha pedido terminar. Por ejemplo, el siguiente programa, una vez convertido en proceso, tiene un hijo que realiza `exit(5)`. El padre captura el 5.

```
#include <signal.h>
int status,pid;
void finhijo(){
    pid=wait(&status);
}
main(){
    signal(SIGCHLD,finhijo);
    if (fork()==0) {sleep(3); exit(5);}
    pause();
    printf("mi hijo ha muerto con estado %d\n",status/256);
    printf("ahora continúo con la ejecución\n");
}
```

## 6.6 EJERCICIOS

### 6.6.1 Ejercicio 1

Realizar un programa (*gessignal.c*) que cree un proceso y que el padre envíe un `SIGUSR1` a su hijo. Cuando éste lo reciba que imprima en pantalla que ha capturado la señal (el `printf` debe estar incrustado en el manejador de esta señal). Hecho esto, el padre realizará un `pause` mientras que el hijo terminará su ejecución con un `exit` (El valor del `exit` será pasado como parámetro a `main`). Al realizar esto, el padre capturará su estado y lo imprimirá en pantalla como se ha visto en el primer ejemplo. A continuación, de nuevo el padre tendrá un hijo que ejecutará la orden “`ls -lr`” del directorio raíz. De nuevo se pide que se capture el estado de finalización y se imprima en pantalla.



### Solución

```
#include <stdio.h>
#include <signal.h>
int estadohijo;
void signalhijo()
{
    //cuando capturamos sigusr1, mostramos este mensaje
    printf("Yo, con pid %i, he capturado una señal\n",getpid());
}
void muerehijo()
{
    //Cada vez que muere un hijo decimos quien era y con que estado ha muerto
    int hijo;
    hijo=wait(&estadohijo);
    printf("Mi hijo %i ha muerto con estado %i\n",hijo,estadohijo/256);
}

int main(int narg, char* arg[])
{
    if(narg!=2)
        printf("Error en los parámetros.\n");
    else
    {
        int salida,pid;
        salida=atoi(arg[1]);
        pid=fork();
        if(pid!=0)
        {
            signal(SIGCHLD,muerehijo);
            sleep(1); //esperamos a que el hijo este
                       //preparado para captar la señal
            //Mandamos la señal sigusr1 al hijo
            kill(pid,SIGUSR1);
            pause();
            if(fork()!=0)
            {
                //Esperamos a que el hijo muera para
                //capturar su señal
                pause();
            }
            else
            {
                execlp("ls","ls","-lR","",0);
            }
        }
        else
        {
            signal(SIGUSR1,signalhijo);
            //esperamos a que el padre nos mande una señal
            pause();
            exit(salida);
        }
    }
}
```

### 6.6.2 Ejercicio 2

Crear un programa llamado timeout.c que acepte un número de segundos y una orden (sin redireccionamientos ni backgorund). Al cabo de esos segundos, si la orden no ha terminado, se deberá eliminar.

El formato es el siguiente:

```
timeout num_seg orden
$ timeout 5 find / -name "alu" -print
...
$
```

Hay que tener en cuenta lo siguiente:

Si no ha dado tiempo se debe mostrar: "El tiempo de ejecución de la orden ha vencido"

Si ha dado tiempo se debe mostrar: "Ejecución de la orden satisfactoria"

### Solución

```
#include <stdio.h>
#include <string.h>
#include <signal.h>

#define CIERTO 1
#define FALSO 0
#define SIG_ALARM 14
#define SIG_KILL 15
#define TAM 200

int pid,flag;

main(int argc, char *argv[])
{
    void matar(int p);
    char *argt[TAM];
    char cadena[TAM];
    char c;
    int i,status,tim,err,tiempo;
    flag=0;

    for(i=2;i<argc;i++)
        argt[i-2]=argv[i];

    if((pid=fork())==0)
    {
        err=execvp(argt[0],argt);
    }
    else
    {
        tiempo=atoi(argv[1]);
        signal(SIG_ALARM,matar);
        tim=alarm(tiempo);
        wait(&status);
        if(flag==0)
            printf("\nEjecución de la orden satisfactoria.\n");
    }
}
```

```

void matar(int p)
{
    kill(pid,SIG_KILL);
    flag=1;
    printf("El tiempo de ejecución de la orden ha vencido.\n");
}

```

## CAPÍTULO 7. ARCHIVOS Y TUBERÍAS

### 7.1 INTRODUCCIÓN

Una de las funciones más importantes de los sistemas operativos es gestionar los recursos disponibles en el computador, repartiéndolos de forma adecuada entre los diferentes procesos que puedan estar ejecutándose en cada instante. Entre los recursos que debe gestionar se tiene, por ejemplo, la memoria principal, la memoria secundaria, los dispositivos y la CPU. El sistema de archivos es una de las partes del sistema operativo que se encarga de estudiar la gestión de la memoria secundaria.

El sistema de archivos tiene una gran importancia en el computador. Como es conocido uno de los cuellos de botella en un computador es la gestión del almacenamiento secundario y por ello el cómo implementar los archivos y directorios influirá en el rendimiento del sistema.

Para la mayoría de los usuarios, el sistema de archivos es el aspecto más visible de un sistema operativo, pues proporciona el mecanismo para el almacenamiento tanto de datos como de programas del sistema operativo y de los propios usuarios del sistema informático.

En este capítulo vamos a estudiar las llamadas al sistema más representativas que proporciona el sistema operativo para gestionar el sistema de archivos. Asimismo, se describirá cómo se pueden comunicar procesos empleando tuberías.

### 7.2 LLAMADA OPEN

La llamada `open()` crea un archivo o abre un archivo ya creado para lectura y/o escritura. Antes de realizar la mayoría de operaciones sobre un archivo, éste debe ser previamente abierto.

El parámetro `nombre` es una cadena de caracteres que representa el nombre del archivo o dispositivo al que se quiere acceder, con las convenciones de nombres de archivo de UNIX.

El parámetro `operaciones` es un número entero que representa las operaciones que se van a realizar sobre el archivo: lectura, escritura, creación, etc. Lo habitual es confeccionar este número mediante la concatena-

ción con operadores OR de una serie de opciones definidas mediante constantes en el archivo `<fcntl.h>`. Las más importantes son las que aparecen en la siguiente tabla:

Tabla 7-1: Operaciones sobre un archivo de la llamada open.

Valor	Significado
<code>O_RDONLY</code>	Abre para lectura
<code>O_WRONLY</code>	Abre para escritura
<code>O_RDWR</code>	Abre para lectura y escritura (equivalente a <code>O_RDONLY   O_WRONLY</code> )
<code>O_CREAT</code>	Crea el archivo si no existe e ignora el parámetro si ya existe.
<code>O_APPEND</code>	Sitúa la posición al final antes de escribir

El parámetro opcional permisos sólo tiene sentido al utilizar `O_CREAT`, y sirve para asignar los permisos del archivo recién creado, en caso de que no existiera.

La llamada `open()` devuelve -1 en caso de error, o un entero positivo como *descriptor* o manejador de archivo, en caso de éxito. Este entero debe ser almacenado para acceder posteriormente a los datos del archivo, ya que el resto de operaciones sobre el mismo se realizarán a través del descriptor de archivo.

El descriptor de archivo es un índice que identifica una entrada en la *tabla de descriptores de archivos*. La tabla de descriptores de archivos es una tabla que posee cada proceso. En ella se almacenan toda la información concerniente a los archivos (y tuberías) que se están utilizando en un proceso (sus archivos abiertos). Cada vez que se abre un archivo se utiliza una entrada de esta tabla y se devuelve como descriptor el número de entrada utilizada.

Cuando se crea un proceso, su tabla de descriptores de archivos contiene las tres primeras entradas ocupadas con otros tantos archivos abiertos, que se utilizan como entradas y salidas estándar del proceso, como se muestra en la siguiente tabla. Hay que tener en cuenta que un proceso siempre hereda la tabla de descriptores de archivos del proceso que lo creó (proceso padre).

Tabla 7-2: Tabla de descriptores de archivos general.

Entrada	Archivo
0	Entrada estándar
1	Salida estándar
2	Salida de errores estándar
...	Archivos abiertos
...	Archivos abiertos

Cuando se emplean los caracteres de redireccionamiento sobre un proceso, realmente lo que se está haciendo es modificar su tabla de descriptores de archivos. Por ejemplo, `$ a.out > salida.txt` modifica la tabla como sigue

Tabla 7-3: Tabla de descriptores de archivos del ejemplo.

Entrada	Archivo
0	Entrada estándar
1	Salida estándar
2	Salida de errores estándar

La llamada `open()` ocupará la entrada de la tabla con el descriptor más bajo posible. Así, es posible cambiar los dispositivos E/S estándar simplemente liberando (llamada `close()`) las entradas 0, 1 o 2 (según convenga) y abriendo a continuación un nuevo archivo o dispositivo.

#### Proceso A:

```
fd1 = open ("/etc/passwd", O_RDONLY);
fd2 = open ("local", O_WRONLY);
fd3 = open ("/etc/passwd", O_RDONLY|O_WRONLY);
```

#### Proceso B:

```
fd1 = open ("/etc/passwd", O_RDONLY);
fd2 = open ("private", O_RDONLY);
```

## TABLA DE DESCRIPTORES

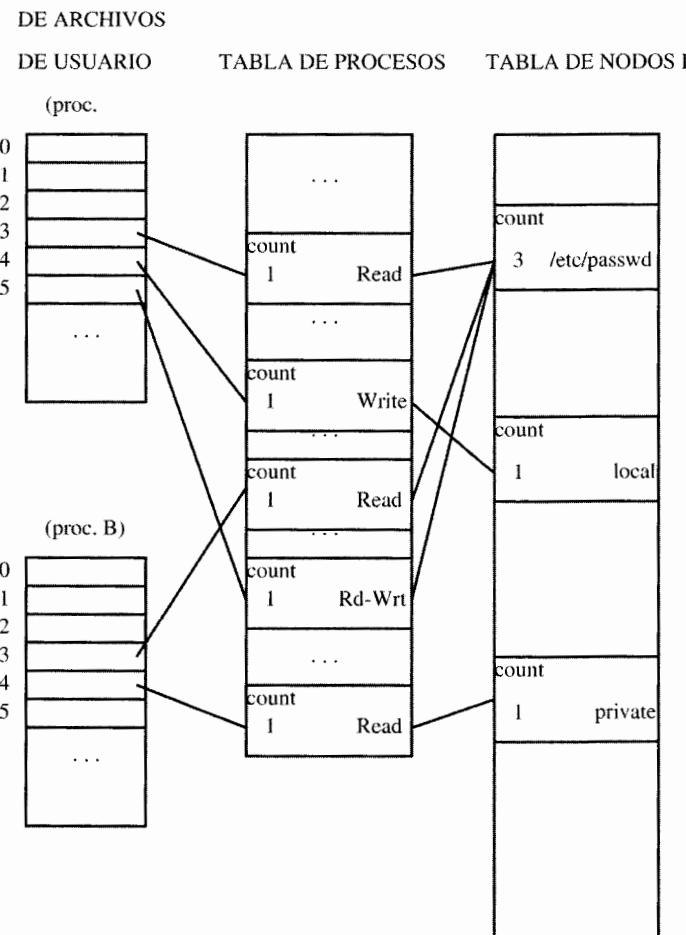


Figura 7-1. Tablas de archivo del proceso A y B.

## 7.3 LLAMADA CLOSE

La llamada `close()` cierra un archivo abierto, cuyo descriptor es `archivo` y libera la entrada correspondiente en la tabla de descriptores de archivos. Devuelve `-1` en caso de error y `0` en caso de éxito.

Ejemplo: programa que comprueba la existencia de un archivo utilizando `close()`.

```
#include <fcntl.h>
main(int argc, char *argv[])
{
    int df;
    if (argc != 2) {
        printf("Falta o sobra parámetro\n");
        exit(-1);
    }
    df = open(argv[1], O_RDONLY); /* SI HAY ERROR SE IGNORA */
    if (close(df) < 0) printf("NO "); /* close(-1) DEVUELVE -1 */
    printf("EXISTE\n");
}
```

## 7.4 LLAMADA CREAT

La llamada `creat()` crea un nuevo archivo con el nombre indicado en el parámetro `nombre` y con los permisos indicados en el parámetro `permisos`, y además lo abre para escritura aunque los permisos no lo permitan. Si el archivo ya existe, los trunca a cero y lo abre para escritura, ignorando el parámetro de permisos.

`creat()` devuelve `-1` en caso de error o el descriptor del archivo abierto en caso de éxito.

Ejemplo: creación de un archivo vacío con todos los derechos.

```
int df;
...
df = creat("nuevo.txt", 0777);
if (df < 0)
    perror ("Error al crear");
else close(df);
```

## 7.5 LLAMADA READ

La llamada `read()` lee una secuencia de `nbytes` octetos (bytes) del archivo o dispositivo abierto (para lectura) cuyo descriptor es `archivo` y los escribe a partir de una posición de memoria indicada por el parámetro `buffer`. Devuelve `-1` en caso de error o el número de bytes realmente leídos en caso de éxito.

Es posible que el número de bytes que realmente se lean no coincida con el pedido a través del parámetro `nbytes`. Esto puede suceder cuando, por ejemplo, se intentan leer más bytes de los que realmente hay en un archivo. El hecho de que la llamada `read()` devuelva `0` no es un error, y por tanto no actualiza el valor de `errno`.

El parámetro `buffer` es un puntero a carácter (vector de caracteres) y por tanto debe apuntar a una zona de memoria suficientemente grande como para contener todos los caracteres solicitados en el parámetro `nbytes`. De lo contrario, la llamada `read` podría sobrescribir otras variables del proceso, producir un error o incluso producir un error de protección y abortar.

Después de una llamada `read()` la posición del archivo (punto del archivo desde el cual empieza a leerse o escribirse) avanza tantos octetos como caracteres (bytes) hayan sido leídos.

Ejemplo: leer un número del terminal y multiplicarlo por 2.

```
#include <fcntl.h>

main() {
    int df,tam;
    long numero;
    char buffer[10];

    df=open("/dev/tty", O_RDONLY);
    if (df<0) {
        perror("Error al abrir tty");
        exit(-1);
    }
    tam=read(df, buffer, 9); /* COMO MUCHO DE HASTA 9 DIGITOS */

    if (tam == -1)
        perror("Error de lectura");
    else {
        buffer[tam]=0; /* PONE EL FINAL DE CADENA */
        numero=atoi(buffer);
        printf("Resultado: %ld\n",numero*2);
    }
    close(df);
}
```

## 7.6 LLAMADA WRITE

La llamada `write()` es semejante a `read()`, pero a la inversa. Escribe una secuencia de `nbytes` octetos (bytes) en el archivo o dispositivo abierto (para escritura) cuyo descriptor es `archivo`. La secuencia de datos a escribir se obtiene a partir de una posición de memoria indicada por el parámetro `buffer`. Devuelve `-1` en caso de error o el número de bytes realmente escritos en caso de éxito. Es posible que el número de bytes que realmente se escribe no coincida con el pedido a través del parámetro `nbytes`. Esto puede suceder cuando, por ejemplo, se intentan escribir más bytes de los que realmente caben en el disco.

Después de una llamada `write()` la posición del archivo avanza tantos bytes como caracteres hayan sido escritos.

Ejemplo: crear un archivo con las variables de entorno.

```
#include <string.h>

main(int argc, char *argv[], char *env[]) {
    int df, cont=0;
    char buffer[128];

    df=creat("variables.txt", 0755);
    if (df<0) {
        perror("Error al crear archivo");
        exit(-1);
    }
    while (env[cont] != NULL) {
        strcpy(buffer, env[cont]);
        strcat(buffer, "\n");
        if (write(df, buffer, strlen(buffer)) != strlen(buffer)) {
            perror("Error al escribir");
            break; /* NO SIGUE */
        }
        cont++;
    }
    close(df);
}
```

## 7.7 LLAMADA LSEEK

La llamada `lseek()` sirve para desplazar la posición del archivo cuyo descriptor es `archivo`. Hay tres posibilidades, seleccionadas mediante el parámetro `modo`, definidas en el archivo `<stdio.h>` y recogidas en la tabla siguiente:

Tabla 7-4: Opciones de la llamada `lseek`.

	Valor	Desplazamiento
Modo		
SEEK_SET	0	Absoluto: desplazamiento es la nueva posición
SEEK_CUR	1	Relativo: la nueva posición es desplazamiento bytes desde la posición actual
SEEK_END	2	Relativo al fin de archivo: la nueva posición es desplazamiento bytes desde el final del archivo

`lseek()` devuelve `-1` en caso de error y la nueva posición del archivo en caso de éxito. La nueva posición puede ser diferente de la que indica `desplazamiento` en el caso de que se excedan los límites del archivo.

## 7.8 LLAMADA UNLINK

La llamada `unlink()` elimina del disco el archivo cuyo nombre es `archivo`. Devuelve `-1` en caso de error y `0` en caso de éxito.

Ejemplo: borrar todos los archivos pasados como parámetro.

```
main(int argc, char *argv[]) {
    int cont, ok=0;
    char mensaje[256];

    for (cont=1; cont<argc; cont++) {
        if (unlink(argv[cont]) != 0)
            sprintf(mensaje, "Error al eliminar %s", argv[cont]);
        perror(mensaje);
    }
    else ok++;
}
printf("Borrados %d enlaces.\n", ok);
```

## 7.9 LLAMADA DUP

La llamada `dup()` duplica una entrada de la tabla de descriptores de archivos. La entrada duplicada es la que indica el parámetro manejador. La nueva entrada, se sitúa en la primera entrada libre de la tabla de descriptores de archivos.

Devuelve `-1` en caso de error y el número de entrada ocupada en caso de éxito (nuevo descriptor de archivo). Después de una llamada `dup()` el archivo puede ser accedido de la misma forma tanto desde el descriptor antiguo como desde el nuevo, ya que para ambos descriptores tienen las mismas características:

Están asociados al mismo archivo o dispositivo.

La posición del archivo es la misma para ambos.

Comparten el mismo modo de acceso (lectura, escritura...).

Los dos descriptores deben cerrarse independientemente para liberar las entradas correspondientes en la tabla de descriptores de archivos.

Ejemplo: rutina para hacer que la salida de errores sea la misma que la salida estándar.

```
void igualar_salidas() {
    int ocupada;

    close(2); /* CIERRA LA SALIDA DE ERRORES STD_ERR */
    ocupada=dup(1); /* INTENTA OCUPAR SU ENTRADA */

    if (ocupada == -1) {
        perror("Error al duplicar");
        return;
    }

    /* QUEREMOS QUE OCUPE LA 2, PERO SI LA 0 ESTABA LIBRE HABRÁ OCUPADO
     * LA 0 */
    if (ocupada == 0) { /* LA PRIMERA ENTRADA ESTABA LIBRE */
        ocupada=dup(1); /* VOLVEMOS A DUPLICAR */
        close(0); /* VOLVEMOS A LIBERAR LA PRIMERA */
    }

    /* COMPROBAMOS QUE HA OCUPADO LA ENTRADA 2 */
}
```

```
    if (ocupada == 2) printf("Ok.\n");
    else printf("No ha sido posible.\n");
}
```

## 7.10 LLAMADA STAT

La llamada `stat()` devuelve información sobre un archivo: tipo de dispositivo, propietario, grupo, hora de acceso, modificación, etc. Todos estos datos están contenidos en la estructura `stat`, definida en `<sys/stat.h>`.

```
struct stat {
    dev_t      st_dev;   /* DISPOSITIVO */
    ino_t      st_ino;   /* INODO */
    umode_t    st_mode;  /* PROTECCIÓN */
    nlink_t    st_nlink; /* NÚMERO DE ENLACES Duros */
    uid_t      st_uid;   /* PID DEL PROPIETARIO */
    gid_t      st_gid;   /* GID DEL PROPIETARIO */
    dev_t      st_rdev;  /* TIPO DE DISPOSITIVO (SI DISPOSITIVO) */
    off_t      st_size;  /* TAMAÑO TOTAL EN BYTES */
    unsigned long st_blksize; /* TAMAÑO DE BLOQUE */
    unsigned long st_blocks; /* NÚMERO DE BLOQUES OCUPADOS */
    time_t     st_atime; /* HORA DE ÚLTIMO ACCESO */
    time_t     st_mtime; /* HORA DE ÚLTIMA MODIFICACIÓN */
    time_t     st_ctime; /* HORA DE ÚLTIMO CAMBIO */
};
```

## 7.11 LLAMADAS MKDIR, RMDIR Y CHDIR

La llamada `mkdir()` crea un directorio: `nombre` es la ruta a crear y `modo` son los permisos asignados; `rmdir()` elimina un directorio (que debe estar vacío); y `chdir()` cambia el directorio actual.

En todos los casos se retorna `-1` si ocurre algún error y `0` de lo contrario. `mode_t` se define en `<sys/types.h>`.

## 7.12 TUBERÍAS. LLAMADA PIPE

Una tubería (`pipe`) se puede considerar como un canal de comunicación entre dos procesos. Los mecanismos que se utilizan para manipular tuberías son los mismos que para archivos, con la única diferencia de que la información de la tubería no se almacena en el disco duro, sino en la memoria principal del sistema.

Esta comunicación consiste en la introducción de información en una tubería por parte de un proceso (similar a la escritura en un archivo de disco). Posteriormente otro proceso extrae la información de la tubería

(similar a la lectura de información almacenada en un archivo de disco) de forma que los primeros datos que se introdujeron en ella son los primeros en salir. Este modo de funcionamiento se conoce como FIFO (First In, First Out; el primero en entrar es el primero en salir).

La comunicación mediante tuberías es de tipo half-duplex, es decir, en un instante dado, la comunicación solamente puede tener lugar en un sentido. Si se quiere que un proceso A pueda simultáneamente enviar y recibir información de otro B, en general se debe recurrir a crear dos tuberías, una para enviar información desde A hacia B y otra para enviar desde B hacia A.

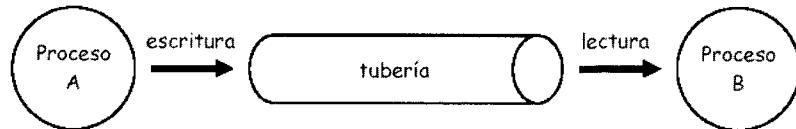


Figura 7-2. Comunicación mediante tuberías.

Un tubo se puede ver como un archivo intermedio o un pseudoarchivo. A nivel del shell su implementación consiste en variar las entradas estándar sobre las tablas de descriptores de archivos de cada proceso.

```
$ cat | wc genera
```

cat:

Entrada	Archivo
0	Entrada estándar
1	Salida estándar
2	Salida de errores estándar

wc:

Entrada	Archivo
0	Entrada estándar
1	Salida estándar
2	Salida de errores estándar

Se pueden distinguir dos tipos de tuberías dependiendo de las características de los procesos que pueden tener acceso a ellas:

Sin nombre. Solamente pueden ser utilizadas por los procesos que las crean y por los descendientes de éstos.

Con nombre o fifo. Se utilizan para comunicar procesos entre los que no existe ningún tipo de parentesco.

En este capítulo nos centraremos en la tubería sin nombre.

La llamada `pipe()` crea una *pipe* (tubería) sin nombre. Esta llamada admite como parámetro un vector `descriptores` con espacio para almacenar dos descriptores de archivo. El primero de ellos (`descriptores[0]`) sirve para que los procesos lean de la tubería, mientras que el segundo (`descriptores[1]`) se emplea para escribir los datos en la tubería.

Para que dos procesos puedan intercambiar información entre sí deben compartir las entradas en la tabla de descriptores de archivos, y la única forma de que esto sea posible es que mantengan algún tipo de parentesco. Al hacer `fork()` la tabla de descriptores de archivos se hereda, y por tanto, cualquier tubería creada con anterioridad también. Para que la tubería se conserve al hacer `exec()` es necesario mantener sus descriptores en algunas de las tres primeras entradas de la tabla de descriptores de archivos, ya que son las únicas que se conservan en el nuevo proceso creado con `exec()`.

Si un proceso se va a limitar a leer o a escribir sobre una tubería, puede cerrar el descriptor de la operación que no va a realizar, ya que los descriptores son un recurso limitado.

Las operaciones `read()` y `write()` tienen un comportamiento ligeramente distinto para las tuberías respecto a los archivos. Una tubería funciona como una estructura de tipo cola en la que los elementos escritos en ella la van llenando, y se va vaciando cuando se lee.

El funcionamiento de la tubería sigue las siguientes pautas:

Si un proceso intenta leer de una tubería vacía, quedará detenido hasta que haya datos que leer.

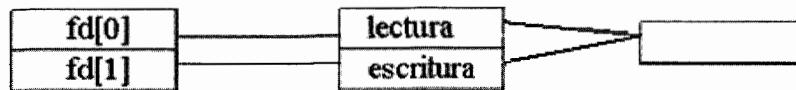
Si un proceso intenta leer de una tubería vacía, pero todos los posibles descriptores del otro extremo de la tubería (es decir, de la parte de escritura) han sido cerrados, el proceso no quedará detenido y se retornará un final de archivo (EOF).

Si un proceso intenta leer de una tubería no vacía, pero con menos datos de los pedidos en `read()`, se leerán los datos disponibles y no quedará detenido.

Si un proceso intenta escribir en una tubería que está llena, quedará detenido hasta que haya espacio suficiente para escribir todos los datos. La escritura se completará o no comenzará; en ningún caso se escribirán menos bytes de los indicados en `write()`.

Si un proceso intenta escribir en una tubería, pero todos los posibles descriptores del otro extremo de la tubería (es decir, de la parte de lectura) han sido cerrados, se producirá un error, el proceso recibirá una señal de tipo `SIGPIPE`, y si no está preparado para recibirla, morirá.

Con la llamada `pipe(fd)` creamos dos enlaces con el archivo `fd`, uno de lectura y otro de escritura



Ejemplo: Esqueleto de programa para disponer dos procesos en cadena.

```

#define STD_INPUT 0 // descriptor de archivo entrada estandar
#define STD_OUTPUT 1 // descriptor de archivo salida estandar
cadena(proceso1, proceso2)
char *proceso1, *proceso2; // punteros a nombres de programa
{
    int fd[2];
    pipe(&fd[0]);
    if (fork() != 0) {
        close(fd[0]); // el proceso 1 no ha de leer del tubo
        close(STD_OUTPUT); // preparación de la nueva salida estandar
        dup(fd[1]); // salida estandar vinculada a fd[1]
        close(fd[1]); // ya no se necesita más el tubo
        execl(proceso1, proceso1, 0);
    } else {
        close(fd[1]); // el proceso 2 no escribe en el tubo
        close(STD_INPUT); // preparación de la nueva entrada estandar
        dup(fd[0]); // entrada estandar vinculada a fd[0]
        close(fd[0]); // ya no se necesita más el tubo
        execl(proceso2, proceso2, 0)
    }
}
  
```

## 7.13 EJERCICIOS

### 7.13.1 Ejercicio 1

Realizar un programa llamado `mcopy.c` que copie un archivo origen varias veces en archivos diferentes.

El formato es el siguiente:

`mcopy archivo_origen archivo_destino1 archivo_destino2`

Un ejemplo de la línea de comandos del shell podría ser la siguiente:

```
$ mcopy datos.txt datos(seg).bak
...
$
```

Hay que tener en cuenta lo siguiente:

- Las entradas y salidas de dispositivos se realizarán con llamadas al sistema (no se puede utilizar `printf`, `scanf`, etc.)

- No se puede utilizar ninguna función de C ó C++ que sustituya a las llamadas al sistema de archivos (`fopen`, `fprintf`, etc.)
- El padre creará tantos hijos como copias del archivo a realizar.
- El padre lee el archivo origen y mediante tuberías le pasa la información del archivo a los hijos.
- Cada hijo crea el archivo con el nombre e información que le ha pasado el padre.
- Cuando el hijo haya grabado el archivo deberá mostrar: "Hijo con PID YY. Archivo XXX creado correctamente"
- Se debe controlar los errores de archivos como "Archivo origen no indicado", "Archivo XXX inexistente", "¿Desea sobrescribir el archivo XXX (S/N)?".

### Solución

```

#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

#define BUF_SIZE 1024 /* Tamaño del buffer de E/S , en bytes */

/* crea_hijo
 * Crea un hijo que leerá de fd
 * y lo copiará en el archivo "nombre"
 *
 * Devuelve el pid del proceso hijo en el lado del padre
 */
void crea_hijo(char *nombre,int fd)
{
    if ( fork() == 0 ) {
        char buffer[BUF_SIZE];
        int escribe;
        int bytes_read;
        escribe = open(nombre,O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
        if (escribe == -1) {
            perror("Imposible escribir");
            exit(EXIT_FAILURE);
        }
        do {
            bytes_read = read(fd,&buffer,BUF_SIZE);
            write(escribe,&buffer,bytes_read);
        } while ( bytes_read == BUF_SIZE );
        close(escribe);
        exit(EXIT_SUCCESS);
    }
    return;
}

/*
 * pregunta_sobreescribir
 *
 * Efectúa stat sobre el el path que le pasamos.
 * si da resultado positivo ( el archivo existe )
 * pedimos confirmacion para sobreescritura.
 *
 * Caso de que se vaya a sobreescribir, devolveremos
 * 1, caso contrario 0
 */
  
```

```

/*
int pregunta_sobreescibir(char *path)
{
    struct stat s;
    int ret = 0;
    char check = 0, junk;
    if (stat(path,&s) == -1)
        ret = 1;
    else {
        while (!check) {
            printf("¿ Desea sobreescibir el archivo %s ?\n", path);
            check = getchar();
            junk = getchar(); /* \n a la basura */
            switch(check) {
                case 'n':
                case 'N':
                    ret=0;
                    break;
                case 's':
                case 'S':
                    ret=1;
                    break;
                default:
                    check = 0;
            }
        }
    }
    return ret;
}

int main( int argc, char **argv)
{
    int origen; /* Descriptor a archivo que vamos a leer */
    int n_hijos; /* Numero de hijos que vamos a crear */
    int *pipes; /* Vector de enteros descriptoros de archivo
                  * para los pipes :
                  * [2i] = lectura
                  * [2i+1] = escritura
    */
    int i;
    int bytes_read = 0;
    char buffer[BUF_SIZE];
    struct stat orig_stat; /* Stat del archivo a copiar */

    if( argc < 3 ) {
        printf("Debes suministrar como minimo 2 parametros\n");
        exit(EXIT_FAILURE);
    }

    /* Antes de nada nos aseguraremos si podemos abrir el archivo */
    if ( stat(argv[1],&orig_stat) == -1) {
        perror("Imposible hacer stat");
        exit(EXIT_FAILURE);
    }

    origen = open(argv[1],O_RDONLY);
    if ( origen == -1 )
        perror("Imposible abrir el archivo origen");

    /* Determinamos el numero de hijos y reservamos
     * memoria para el vector de pipes
     */
    n_hijos = argc-2;
    pipes = malloc(sizeof(int) * n_hijos * 2);

    /* Creamos los pipes y lanzamos
     * los hijos , si el archivo existe, ponemos
     * el valor del pipe a -1 para que luego se pueda
     * evitar escribir sobre ese pipe
     */
}

```

```

for ( i = 0; i < n_hijos; i++) {
    if ( pregunta_sobreescibir(argv[i+2]) ) {
        pipe(&pipes[2*i]);
        crea_hijo(argv[i+2],pipes[2*i]);
        close(pipes[2*i]); /* El padre no necesita leer */
    } else
        pipes[(2*i)+1] = -1;
}

/*Escribimos el contenido del fichero leido*/
do {
    bytes_read = read(origen,&buffer,BUF_SIZE);
    for ( i = 0; i < n_hijos ; i++) {
        if ( pipes[(2*i)+1] != -1 )
            write(pipes[(2*i)+1],&buffer,bytes_read);
    }
} while ( bytes_read == BUF_SIZE );

/* Y ahora esperamos a que los hijos acaben */
while(n_hijos) {
    wait(NULL);
    n_hijos--;
}

return 0;
}

```

### 7.13.2 Ejercicio 2

Crear un programa llamado examen.c que acepte un número entre 1 y 4 (ambos inclusive) y un archivo.

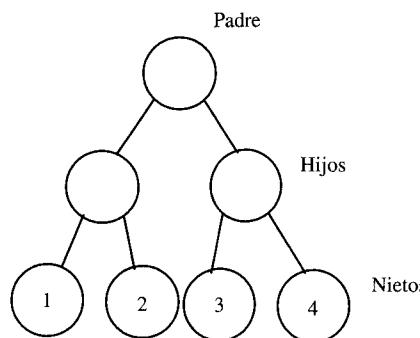
El formato es el siguiente:

examen nieto archivo

Un ejemplo de la línea de comandos del shell podría ser la siguiente:

```
$ ls -l > datos.txt
$ examen 2 datos.txt
...
$
```

El padre creará la siguiente estructura donde cada proceso debe mostrar "PID: XX PPID: YY"



Además:

1. El padre lee el archivo (datos.txt)
2. El padre envía por un tubo la información del archivo al nieto i
3. El nieto i lee del tubo y muestra en pantalla la información leída.

### Solución

```

#include <sys/types.h>
#include <utmp.h>
#include <unistd.h>
#include <fcntl.h>
#define TAMBUF 20 //TAM DEL BUFFER PARA COMUNICACION ENTRE NIETO-HIJO
void mostrarPid(){
    char mipid[40]; //CADENA TEMPORAL PARA EVITAR EL USO DE PRINTF
    sprintf(mipid,"PID: %d PPID: %d\n",getpid(),getppid()); //volcamos los datos
    en una cadena
    write(1,mipid,strlen(mipid)); //VOLCAMOS LA CADENA SOBRE LA SALIDA ESTANDAR
}

int parametrosOk(int argc,char *argv[]){
    int retorno=0;
    int parameter;
    int fic_temp;

    if(argc==3){ //VALIDACION DEL NUMERO DE PARAMETROS
        parameter=atoi(argv[1]); //ATOI CONVIERTE EL CARACTER A ENTERO
        switch(parameter){
            case 1:
            case 2:
            case 3:
            case 4:
                retorno=parameter;break;// SI EL PARAMETRO ESTA ENTRE 1 Y 4, ASIGNA EL
        VALOR
            default:
                retorno=0;break;
        }
        if(fic_temp=(open(argv[2],O_RDONLY))==-1){ //VALIDAMOS QUE EXISTA EL ARCHIVO
            write(1,"El archivo '",13);
            write(1,argv[2],strlen(argv[2])); //COMO INFO ADICIONAL SE MUESTRA EL
        PARAMETRO
            write(1," no existe\n",12);
            retorno=0;
        }else{
    
```

```

            close(fic_temp);// CERRAMOS EL ARCHIVO, EN CASO DE QUE EXISTIESE
        }
        return(retorno);
    }

    void errorParametros(){
        char sintaxis[]="Modo de empleo: examen NIETO{1,2,3,4} ARCHIVO\n\n";
        write(1,sintaxis,strlen(sintaxis));
    }

    main (int argc,char *argv[]){
        int nieto_elegido=parametrosOk(argc,argv);
        int mipipe[2];
        int fic_entrada;
        int r,t; //UTILIZADOS PARA LA ESC/LECT EN LA TUBERIA
        int i,j=0; //UTILIZADOS PARA EL PADRE/HIJOS
        int f,g=0; //UTILIZADOS PARA HIJOS/NIETOS
        int nieto;
        char buf[TAMBUF];

        pipe(mipipe); //DECLARACION DE LA TUBERIA
        if(nieto_elegido==0){
            errorParametros();
            exit(1); //HUBO UN ERROR EN LA INVOCACION
        }else{
            for(i=1;i<3;i++){
                if(fork()==0){ //CODIGO HIJO
                    j=i;
                    i=3; //PARA QUE NO VUELVA A ENTRAR EN EL 'FOR'
                }
                if(j==0){ //Codigo del Padre originario
                    //-----CODIGO DEL PADRE-----
                    mostrarPid();
                    fic_entrada=open(argv[2],O_RDONLY); //APERTURA EN MODO LECTURA
                    close(mipipe[0]); //CERRAMOS LA ENTRADA ESTANDAR DE LA TUBERIA
                    r=read(fic_entrada,buf,TAMBUF); //VAMOS LEYENDO DE LA TUBERIA TAMBUF
                }
                CHARS
                while(r>0){ //CUANDO FINALICE R=-1
                    write(mipipe[1],buf,r);
                    r=read(fic_entrada,buf,TAMBUF);
                }
                close(fic_entrada);
                while(wait()!=-1); //ESPERA QUE FINALICEN TODOS SU HIJOS
                exit(0); //SALIMOS DE FORMA CORRECTA
            }
            if(j!=0){
                mostrarPid();
                for(f=1;f<3;f++){
                    if(fork()==0){
                        g=f;
                        if(j==1){ //HIJO DEL PRIMER HIJO
                            nieto=j+f-1; //ALMACENAMOS EL NUMERO DE NIETO
                        }
                        if(j==2){ //HIJO DEL SEGUNDO HIJO
                            nieto=j+f; //ALMACENAMOS EL NUMERO DE NIETO
                        }
                        f=3; //PARA QUE NO TENGA MAS HIJOS
                    }
                }
            }
            if(g!=0){
                mostrarPid();
                if(nieto_elegido==nieto){
                    //-----CODIGO DEL NIETO ELEGIDO-----
                    close(mipipe[1]); //CERRAMOS LA SALIDA DEL PIPE
                    t=read(mipipe[0],buf,TAMBUF); //LEEMOS DE LA ENTRADA DEL PIPE
                    while(t>0){
                        write(1,buf,t);
                        t=read(mipipe[0],buf,TAMBUF);
                    }
                }
            }
        }
    }
}

```

```

    }
    exit(0);
}else{
exit(0);
}
while(wait() != -1); //ESPERAMOS QUE FINALICEN TODOS LOS NIETOS
}
sleep(1);
}

```

## CAPÍTULO 8. MEMORIA COMPARTIDA

### 8.1 INTRODUCCIÓN

Los sistemas operativos pueden proporcionar diferentes mecanismos para que varios procesos se comuniquen. Hasta el momento hemos estudiado 2 mecanismos de comunicación: mediante las llamadas `exit` y `wait` y a través de la llamada `pipe`. Otro mecanismo es mediante memoria compartida. En este sentido, el sistema operativo UNIX proporciona un paquete de comunicación y sincronización de procesos denominado IPC (InterProcess Communication). Este paquete contiene, entre otros componentes, la gestión de memoria compartida y la gestión de semáforos.

Tal y como hemos estudiados, cuando ejecutamos la llamada `fork` se crea un nuevo proceso, idéntico al padre, en el que se duplican las variables, funciones, etc. Si el hijo actualiza una de sus variables, el proceso padre no tiene tal actualización ya que son variables distintas (son copias). Para que el padre y el hijo tengan la misma variable y se puedan comunicar es necesario emplear las llamadas al sistema que permiten gestionar la memoria compartida.

Para utilizar la memoria compartida se siguen pasos:

- Obtención de un segmento de memoria compartida con su identificador en el sistema (un número entero). Se hace con la llamada `shmget`.
- Vincular el segmento de memoria compartida a través de un puntero a un determinado tipo de datos. Se usa la llamada `shmat`.
- Acceder a la memoria compartida por medio del puntero.
- Desvincular el puntero del segmento de la memoria compartida. Se usa la llamada `shmdt`.
- Eliminación del segmento de memoria compartida. Se usa la llamada `shmctl`.
- A continuación se describen brevemente estas llamadas.



## 8.2 LLAMADA SHMGET

La llamada `shmget` devuelve el identificador de memoria compartida asociado al segmento de memoria. Tiene como parámetros:

- La clave que identifica al segmento de memoria compartida. Puede ser `IPC_PRIVATE` (el proceso y sus descendientes pueden acceder al segmento) u otra clave obtenida mediante la función `ftok` (varios procesos sin parentesco pueden acceder al segmento).
- El tamaño del segmento.
- Indicadores de permisos para acceder al recurso compartido.
- Si la ejecución se realiza con éxito, entonces devolverá un valor no negativo denominado identificador de segmento compartido. En caso contrario, retornará -1 y la variable global `errno` tomará en código del error producido.

## 8.3 LLAMADA SHMAT

Esta llamada asocia o vincula el segmento de memoria compartida especificado por `shmid` (el identificador devuelto por `shmget`) al segmento de datos del proceso invocador. Presenta tres parámetros:

- El identificador obtenido mediante `shmget`,
- La dirección donde se desea que se incluya (en la práctica será `NULL` (0), lo cuál permite que se incluya en cualquier zona libre del espacio de direcciones del proceso).
- Una serie de identificadores de permisos (en la práctica también será `NULL`).

Si la llamada se ejecuta con éxito, entonces devolverá la dirección de comienzo del segmento compartido, si ocurre un error devolverá -1 y la variable global `errno` tomará el código del error producido.

## 8.4 LLAMADA SGMDT

Desasocia o desvincula del segmento de datos del proceso invocador el segmento de memoria compartida ubicado en la localización de memoria especificada por `shmaddr`. Si la función se ejecuta sin error, entonces devolverá 0, en caso contrario retornará -1 y `errno` tomará el código del error producido.

## 8.5 LLAMADA SHMCTL

La llamada `shmctl` permite realizar un conjunto de operaciones de control sobre una zona de memoria compartida identificada por `shmid`. El argumento `cmd` se usa para codificar la operación solicitada. Los valores más usados para este parámetro son:

- `IPC_STAT`: lee la estructura de control asociada a `shmid` y la deposita en la estructura apuntada por `buff`.
- `IPC_RMID`: elimina el identificador de memoria compartida especificado por `shmid` del sistema, destruyendo el segmento de memoria compartida y las estructuras de control asociadas.
- `SHM_LOCK`: bloquea la zona de memoria compartida especificada por `shmid`. Este comando sólo puede ejecutado por procesos con privilegios de acceso apropiados.
- `SHM_UNLOCK`: desbloquea la región de memoria compartida especificada por `shmid`. Esta operación, al igual que la anterior, sólo la podrán ejecutar aquellos procesos con privilegios de acceso apropiados.

La función `shmctl` retornará el valor 0 si se ejecuta con éxito, o -1 si se produce un error, tomando además la variable global `errno` el valor del código del error producido.

## 8.6 EJEMPLO DE USO

Como ejemplo vamos a ver cómo se comparte la variable entera `numero`. Para ello, tendremos como variables globales la variable `numero` y el identificador de la memoria compartida `shmid`.

```
int shmid;
int *numero=NULL;
```

Para las llamadas anteriores hay que incluir los archivos de cabecera:

```
#include <sys/types>
#include <sys/ipc.h>
#include <sys/shm.h>
```

La función `creaComp` crea el identificador de la memoria compartida y la variable que contiene.

```
int creaComp(void)
{
    if((shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666)) == -1)
```

```

{
    perror("Error al crear memoria compartida: ");
    return 1;
}

/* vinculamos el segmento de memoria compartida al proceso */
numero=(int *) shmat (shmid,0,0);

/*iniciamos las variables */
*numero=0;

return 0;
}

```

La función borraComp separa la variable numero del proceso y a continuación elimina la memoria compartida.

```

void borraComp(void)
{
    char error[100];
    if (numero!=NULL)
    {
        /* desvinculamos del proceso la memoria compartida */
        if (shmctl((char *)numero,<0)
        {
            sprintf(error,"Pid %d: Error al desligar la memoria compartida:
",getpid());
            perror(error);
            exit(3);
        }
        /* borramos la memoria compartida */
        if (shmctl(shmid,IPC_RMID,0)<0)
        {
            sprintf(error,"Pid %d: Error al borrar memoria compartida: ",getpid());
            perror(error);
            exit(4);
        }
        numero=NULL;
    }
}

```

## CAPÍTULO 9. SEMÁFOROS

### 9.1 INTRODUCCIÓN

Los sistemas operativos, además de proporcionar diferentes mecanismos para comunicar procesos, pueden ofrecer herramientas que permiten sincronizar varios procesos. El paquete denominado IPC (InterProcess Communication) del sistema operativo UNIX proporciona un conjunto de componentes para la comunicación y sincronización de procesos, entre los que destaca la gestión de semáforos y la gestión de memoria analizada en el capítulo anterior. A lo largo de este capítulo estudiaremos las llamadas al sistema que proporciona el sistema operativo para gestionar los semáforos.

Los semáforos que emplea el sistema operativo UNIX es muy similar a la definición clásica de Dijkstra, en el sentido de que es una variable entera con operaciones atómicas de inicialización, incremento y decremento con bloqueo.

UNIX define tres operaciones fundamentales sobre semáforos:  
**semget.** Crea o toma el control de un semáforo  
**semctl.** Operaciones de lectura y escritura del estado del semáforo. Destrucción del semáforo  
**semop.** Operaciones de incremento o decremento con bloqueo

UNIX no ofrece las operaciones clásicas P y V, sino que dispone de la llamada al sistema **semop** que permite realizar varias operaciones que incluyen las P y V. Previamente a utilizar estas operaciones será necesario crear el semáforo e inicializarlo.

A continuación se describen las tres llamadas al sistema que permiten gestionar los semáforos en UNIX.

### 9.2 LLAMADA SEMGET

La llamada **semget** se emplea para solicitar al sistema operativo que cree un conjunto de semáforos (un vector de semáforos). El valor devuelto (si no ha habido error) es el identificador del vector de semáforos y permite referirse a él en sucesivas llamadas al sistema.

**semget** tiene los siguientes argumentos:

- key es la clave con la que se crea el semáforo, de modo que siempre que se utilice la misma clave se podrá acceder al mismo semáforo. Puede ser IPC\_PRIVATE (obliga a semget a crear un nuevo y único identificador, nunca devuelto por anteriores llamadas a semget hasta que sea liberado con semctl) o bien obtenerse de la función ftok que devuelve una clave (un número) a partir de un nombre de archivo existente y accesible y un número que lo identifica (ver ejemplo de uso y consultar man).
- nsems es el número de semáforos que se van a crear (el tamaño del vector de semáforos).
- semflg es un flag que permite establecer los permisos del semáforo y en caso de que ya esté creado obtener su identificador.

Si la ejecución se realiza con éxito, entonces devolverá un valor no negativo identificando el conjunto de semáforos. En caso contrario, devolverá -1 y la variable global errno tomará en código del error producido.

### 9.3 LLAMADA SEMCTL

La llamada semctl se emplea para realizar operaciones de control sobre los semáforos. Las operaciones que más se emplean son:

- SETVAL permite establecer el valor de los elementos del vector de semáforos.
- IPC\_RMID elimina el vector de semáforos del sistema y debe ser llamada antes de que el proceso finalice, pues los semáforos no son eliminados del sistema cuando el proceso finaliza.
- GETVAL devuelve el valor actual del semáforo.
- Los argumentos de la llamada semctl son los siguientes:
  - semid es el identificador del vector de semáforos (obtenido de semget).
  - semnum es el índice del vector de semáforos, es decir, el semáforo a utilizar en esta operación.
  - cmd representa el tipo de operación de control a realizar (por ejemplo: SETVAL).
  - El cuarto parámetro está relacionado con el tercero, se pasará un tipo de parámetro u otro dependiendo de lo que se haya escogido en el tercer parámetro.

Si ocurre un error la llamada devolverá -1 y la variable global errno tomará el código del error producido.

### 9.4 LLAMADA SEMOP

Esta llamada permite realizar modificar el contador del semáforo. En UNIX es posible añadir cualquier valor al semáforo y no solamente 1 y -1 (funciones P() y V()). A semop se le pasa como argumento:

- semid que indica el identificador del semáforo
- struct sembuf representa el vector de operaciones sobre semáforos.
- nsops es el número de operaciones a realizar
- La estructura struct sembuf está formada por:
  - sem\_num. Índice del elemento del vector de semáforos sobre el que se desea realizar la operación.
  - sem\_op. Cantidad a añadir al valor del semáforo (puede ser negativa, en nuestro caso ±1)
  - sem\_flg. Modificadores para describir cómo se desea que se realice la operación (en nuestro caso no es necesario ningún modificador, por lo que el valor de este campo será 0 para todas las operaciones).

Si el campo sem\_op de una operación es positivo, se incrementa el valor del semáforo. Asimismo, si sem\_op es negativo, se decrementa el valor del semáforo si el resultado no es negativo. En caso contrario el proceso espera a que se dé esa circunstancia. Es decir, sem\_op==1 produce una operación V y sem\_op== -1, una operación P.

La llamada semop devolverá el valor 0 si se ejecuta con éxito, o -1 si se produce un error, tomando además la variable global errno el valor del código del error producido.

### 9.5 EJEMPLO DE USO

Como ejemplo vamos a crear una serie de funciones base para construir las operaciones P y V de los semáforos.

Lo primero hay que definir las variables y declarar los prototipos de las funciones a utilizar.

```
/* Definimos las constantes utilizadas para identificar las operaciones P y V */
#define OP_P ((short) -1)
#define OP_V ((short) 1)

/* Variables globales */
int semid;

/* Prototipos de las funciones que podemos utilizar sobre los semáforos */
int creaSEM(int cuantosSem, char *queClave);
void iniSEM(int semID, ushort queSemaforo, int queValor);
void P(int semID, ushort queSemaforo);
void V(int semID, ushort queSemaforo);
void destruyeSEM(int semID);
```

Para utilizar las llamadas de semáforos hay que incluir los archivos de cabecera:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Con la función `creaSEM` creamos el semáforo basándolo en la llave generada con la cadena `queClave` y con la letra '`k`'. Indicamos que lo cree con permisos de lectura y escritura para el usuario y que si ya existía, simplemente devuelva el identificador. El número de semáforos a crear lo indicamos con `cuantosSEM`.

```
int creaSEM(int cuantosSEM, char *queClave)
{
    int idSEM;
    key_t llave;

    llave = ftok(queClave, 'k');
    if ((idSEM=semget(llave, cuantosSEM, IPC_CREAT|0600))==-1)
    {
        perror("semget");
        exit(-1);
    }

    return idSEM;
}
```

Con la función `iniSEM` inicializamos el valor del contador de semáforo `queSemaforo` perteneciente al grupo de semáforos `semID`.

```
void iniSEM(int semID, ushort queSemaforo, int queValor)
{
    semctl(semID, queSemaforo, SETVAL, queValor);
}
```

Ya podemos realizar las operaciones P y V del semáforo `queSemaforo` perteneciente al grupo de semáforos `semID`.

```
void P(int semID, ushort queSemaforo)
{
    struct sembuf operacion;
    operacion.sem_num = queSemaforo; /* semáforo sobre el que trabajaremos */
    operacion.sem_op = OP_P; /* operación a realizar: p */
    operacion.sem_flg = 0; /* Siempre 0 */
    semop(semID, &operacion, 1); /* Finalmente, ejecutamos la operación */
}

void V(int semID, ushort queSemaforo)
{
    struct sembuf operacion;
    operacion.sem_num = queSemaforo; /* semáforo sobre el que trabajaremos */
    operacion.sem_op = OP_V; /* operación a realizar: v */
    operacion.sem_flg = 0; /* Siempre 0 */
    semop(semID, &operacion, 1); /* Finalmente, ejecutamos la operación */
}
```

Por último con la función `destruyeSEM` liberamos el semáforo de la memoria.

```
void destruyeSEM(int semID)
{
    semctl(semID, 0, IPC_RMID, 0);
}
```

## 9.6 EJERCICIOS

### 9.6.1 Ejercicio 1

Realizar un programa (`parimpar.c`) que cree un proceso y que ambos incrementen una variable alternativamente hasta llegar al valor indicado como argumento. Para que los dos procesos modifiquen la misma variable tendrán que compartir una zona de memoria y asociarla a la variable. Para que la incrementen alternativamente tendrán que utilizar los semáforos para sincronizarse.

Ejemplo:

```
$ parimpar 5
Padre (345): numero=1
Hijo (346): numero=2
Padre (345): numero=3
Hijo (346): numero=4
Padre (345): numero=5
```

Descripción:

El proceso `parimpar` actúa como proceso padre y crea un proceso hijo. La variable, que se debe llamar `numero`, será de tipo entero e inicialmente valdrá 0.

El proceso padre incrementa la variable e imprime `Padre (<pid>): numero=<valor>`

El proceso hijo incrementa la variable e imprime `Hijo(<pid>): numero=<valor>`

### Solución

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <unistd.h>
```

```

/* defines */
#define STDERR 2
#define STDOUT 1
/* identificadores de semáforos para alternarse */
#define SINC1 ((ushort) 0) /* PADRE */
#define SINC2 ((ushort) 1) /* HIJO */

/* Definimos las constantes utilizadas para identificar las operaciones P y V */
#define OP_P ((short) -1)
#define OP_V ((short) 1)

/* variables globales */
int shmid;
int *numero=NULL;
int semid;
int max;

/* Prototipos de las funciones que podemos utilizar sobre los semáforos */
int creaSEM(int cuantosSEM, char *queClave);
void iniSEM(int semID, ushort queSemaforo, int queValor);
void P(int semID, ushort queSemaforo);
void V(int semID, ushort queSemaforo);
void destruyeSEM(int semID);

/* funciones */
void salir(void);

int creaSEM(int cuantosSEM, char *queClave)
{
    int idSEM;
    key_t llave;

    /* Creamos el semáforo basandolo en la llave generada con la cadena
     * queClave y con la letra 'k'.
     * Indicamos que lo cree con permisos de lectura y escritura para
     * el usuario y que si ya existía, simplemente devuelva el id.
     * El número de semáforos a crear lo indicamos con 'cuantosSEM'
    */

    llave = ftok(queClave, 'k');
    if ((idSEM=semget(llave, cuantosSEM, IPC_CREAT|0600))==-1)
    {
        perror("semget");
        exit(-1);
    }

    /* Devolvemos el identificador general de los semáforos */
    return idSEM;
}

void iniSEM(int semID, ushort queSemaforo, int queValor)
{
    /* Inicializamos el valor del contador de semáforo queSemaforo
     * perteneciente al grupo de semáforos semID
    */
    union semun aux;
    aux.val=queValor;
    semctl(semID, queSemaforo, SETVAL, aux);
}

void P(int semID, ushort queSemaforo)
{
    struct sembuf operacion;
    operacion.sem_num = queSemaforo; /* semáforo sobre el que trabajaremos */
    operacion.sem_op = OP_P; /* operación a realizar: P */
    operacion.sem_flg = 0; /* Siempre 0 */
    semop(semID, &operacion, 1); /* Finalmente, ejecutamos la operación */
}

void V(int semID, ushort queSemaforo)
{
    struct sembuf operacion;
    operacion.sem_num = queSemaforo; /* semáforo sobre el que trabajaremos */
}

```

```

operacion.sem_op = OP_V; /* operación a realizar: V */
operacion.sem_flg = 0; /* Siempre 0 */
semop(semID, &operacion, 1); /* Finalmente, ejecutamos la operación */
}

void destruyeSEM(int semID)
{
    /* Liberamos el semáforo. Hay que tener en cuenta que se trata de una
     * estructura de memoria global que debe liberarse.
    */
    union semun aux;
    aux.val=0;
    semctl(semID, 0, IPC_RMID, aux);
}

/**
 * devuelve:
 * 0 ok!
 * >0 error
 */
int creaSem(void)
{
    semid=creaSEM(2,"clave que nosostros queremos\0");
    iniSEM(semid,SINC1,0);
    iniSEM(semid,SINC2,0);
    return 0;
}

/**
 * Crea memoria compartida
 * devuelve:
 * 0 ok!
 * >0 error
 */
int creaComp(void)
{
    /* creamos memoria compartida */
    if((shmid=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|0666))==-1)
    {
        perror("Error al crear memoria compartida: ");
        return 1;
    }

    /* unimos la MC al proceso */
    numero=(int *) shmat(shmid,0,0);

    /*iniciamos las variables */
    *numero=0;

    return 0;
}

/**
 * Trabajo que realiza el hijo
 */
void Hijo(void)
{
    int ok=0;
    char men[100];
    while (ok!=1)
    {
        P(semid,SINC2);
        if ((*numero)!=max)
        {
            (*numero)++;
            sprintf(men,"Hijo(%d) numero=%d\n\0",getpid(),*numero);
            write(STDOUT,men,strlen(men));
        }
        else
        {
            ok=1; /* para salir del bucle */
            if ((*numero) == 1) V(semid,SINC1);
        }
    }
}

```

```

        V(semid,SINC1);
    }
    /* destruyo semaforo */
    destruyeSEM(shmid);
    exit(0); /* salimos */
}
/***
 * Creacion de proceso hijo
 * devuelve:
 * 1 si error
 * 0 ok!
 */
int
creaHijo(void)
{
    int pid;
    pid=fork();
    if (pid<0)
    {
        perror("Imposible crear Hijo: ");
        return 1;
    }
    else if(pid==0)
    { /* hijo */
        Hijo();
    }
    else
    { /* padre */
        return 0;
    }
}
/***
 * Libera recursos ocupados
 */
void
salir(void)
{
    char error[100];
    if (numero!=NULL)
    {
        /* separamos del proceso la MC */
        if (shmctl((char *)numero)<0)
        {
            sprintf(error,"Pid %d: Error al desligar la memoria compartida:
",getpid());
            perror(error);
            exit(3);
        }
        /* borramos ls MC */
        if (shmctl(shmid,IPC_RMID,0)<0)
        {
            sprintf(error,"Pid %d: Error al borrar memoria compartida:
",getpid());
            perror(error);
            exit(4);
        }
        numero=NULL;
    }
    /* destruye semaforo */
    destruyeSEM(semid);
}

int main(int argc,char **argv)
{
    int ok=0;
    char error[3][100]={"Uso: parimpar numero\n\n",
                        "numero debe ser positivo\n\n","\0"};
    if (argc!=2)
    {
        write(STDERR,error[0],strlen(error[0]));
        exit(1);
    }
    max=atoi(argv[1]);
    if (max < 1)
    {
        write(STDERR,error[1],strlen(error[1]));
    }
}

```

```

        exit(5);
    }
    if (creaSem(>0) exit(1);
    if (creaComp(>0)
    {
        salir();
        exit(1);
    }
    if (creaHijo(>0)
    {
        salir();
        exit(2);
    }
    while(ok!=1)
    {
        if ((*numero)!=max)
        {
            (*numero)++;
            sprintf(error[2],"Padre(%d) numero=%d\n\0",getpid(),*numero);
            write(STDOUT,error[2],strlen(error[2]));
        }
        else ok=1;
        V(semid,SINC2);
        P(semid,SINC1);
    }
    /* matamos al hijo */
    wait(NULL);
    salir();
}

```

## CAPÍTULO 10. PRÁCTICAS GENERALES

Hasta ahora, los ejercicios prácticos se centraban en los aspectos específicos de la teoría de cada capítulo. En este capítulo se plantean prácticas más complejas que combinan lo estudiado hasta el momento.

### 10.1 PRÁCTICA 1

Crear un shell reducido llamado (minishell.c) que lea órdenes y las execute hasta que se introduzca la orden exit. El shell debe aceptar redirecciones estándar (>, <, 2>, >> y |) y permitir la ejecución de procesos tanto en foreground como en background. El prompt del shell debe contener miniShell:<ruta\_actual>, es decir, que si se cambia de directorio se tiene que actualizar.

Un ejemplo de ejecución es el siguiente:

```
/home/guest$minishell
miniShell:/home/guest$ls -l
total 16
-rw----- 1 guest dtic      513 04 mar 20:10 mbox
drwxr-sr-x 2 guest dtic      512 03 mar 11:40 FSO
-rwxr-xr-x 1 guest dtic     25163 04 mar 20:06 minishell
-rw----- 1 guest dtic    30607 04 mar 20:05 minishell.c
MiniShell:/home/guest$mkdir prueba
MiniShell:/home/guest$cd prueba
MiniShell:/home/guest/prueba$echo Hola, mundo > sal.txt
MiniShell:/home/guest/prueba$ls -l
total 8
-rw-r--r-- 1 guest dtic      12 04 mar 20:22 sal.txt
MiniShell:/home/guest/prueba$echo Hello, world >> sal.txt
MiniShell:/home/guest/prueba$wc sal.txt
      2      4      25 sal.txt
MiniShell:/home/guest/prueba$cd
MiniShell:/home/guest/prueba$more < minishell.c | grep fork | wc -l
      8
MiniShell:/home/guest$find / -name minishell -print > salida.txt 2> salerror.txt
&
[1]   13394
MiniShell:/home/guest$exit
/home/guest$
```

### 10.2 PRÁCTICA 2

Realizar un programa (encripta.c) que codifique (encripte) un fichero origen cuyo nombre se pasa como primer argumento y cree un nuevo fichero cuyo nombre se pasa como segundo argumento con la información del fichero origen encriptada.

Un ejemplo de ejecución es el siguiente:

```
$ encripta datos.org datos.enc
```

El programa crea el fichero datos.enc que contiene la información que hay en el fichero datos.org pero encriptada.

El método de encriptación es el siguiente:

- El proceso encripta actúa como proceso padre creando n procesos hijos, donde n es el número de líneas del fichero origen.
- El proceso padre lee todo el fichero origen y envía mediante tubos una línea del fichero origen a cada proceso hijo de tal forma que al proceso hijo 1 le envía la línea 1, al proceso hijo 2 la línea dos, etc. El proceso padre espera a que los hijos le envíen mediante tubos las líneas encriptadas y a continuación crea el nuevo fichero con la información recibida.

Los procesos hijos son los que realmente encriptan. La forma de hacerlo es la siguiente:

```
/* Algoritmo del proceso hijo i */
Recibir línea completa del padre
Para cada carácter car_org de línea_completa hacer
car_aux1=car_org+1
    Enviar car_aux1 al proceso hijo (i+1)
    Recibir car_aux2 del proceso hijo (i-1)
car_enc=car_aux1-car_aux2
fPara
Enviar -1 al proceso hijo (i+1) /* Le indica que ha
finalizado */
Enviar línea_encriptada al padre
```

El proceso hijo 1 no recibe car\_aux2 y el proceso hijo n no envía car\_aux1 ni -1

Para verificar que el programa de encriptación funciona correctamente se puede realizar otro programa llamado *desencripta* que realice el proceso inverso.

### 10.3 PRÁCTICA 3

La práctica consiste en simular el funcionamiento de un proceso gestor de tareas en un entorno multiusuario, existiendo dos procesos: el proceso cliente que planifica la tarea (comando) a ejecutarse en un tiempo determinado; y el proceso servidor que recoge las tareas (comandos) de los clientes y las ejecuta cuanto toque.

Tras la ejecución del gestor y comprobado que no existe una instancia suya, el gestor analizará cada cierto tiempo un fichero que contiene una lista

con las tareas que se han de ejecutar en un instante de tiempo determinado; una vez ejecutada una tarea, se almacenará los resultados (Tiempo actual, Usuario, Número identificador de la tarea y Comandos) en otro fichero; éste sólo será accesible por el usuario del gestor de tareas.

La periodicidad con la que el gestor debe analizar el fichero se indicará mediante la variable de entorno INTERVALO, mientras que el fichero de tareas y el fichero de resultados se indicarán mediante las variables de entorno FICHEROTMP y FICHEROLOG, respectivamente. Hay que tener en cuenta que la variable INTERVALO puede ser modificada una vez lanzado el proceso gestor de tareas.

La ejecución del proceso gestor de tareas se realizará mediante el comando:

```
$ gestortarea // Inicia gestor
$ gestortarea -t // Elimina gestor de la memoria
```

Además, del gestor, existirá otro proceso que se encargará de gestionar el fichero de tareas; este proceso se encargará de añadir tareas, eliminarlas o bien visualizar las tareas planificadas solo del usuario. Este proceso debe emitir un error en caso de que el gestor de tareas no exista.

La ejecución del proceso manejador del fichero de tareas se realizará mediante el siguiente comando y sus correspondientes parámetros:

```
$ activatarea -l
// lista las tareas planificadas del usuario que lo ha invocado.
```

```
$ activatarea -r <número de tarea>
// elimina la tarea indicada por su identificador. El número de tarea es único para todos los usuarios, es decir, de ámbito global.
```

```
$ activatarea -h <tiempo> <comandos>
// añade una nueva tarea con el tiempo de ejecución y los comandos indicados. El parámetro <tiempo> podrá expresar periodicidad si no se incluyen horas, minutos o segundos (ejemplo: ::05 @ ejecutar siempre y cuando sea el segundo 5; :30: @ ejecutar siempre y cuando sea media hora).
```

```
$ activatarea -p <tiempo> <comandos>
// añade una nueva tarea que se ejecutará periódicamente según los segundos indicados en el parámetro <tiempo>, y ejecutando el comando indicado.
```

Ejemplo:

```
$ export INTERVALO=5
$ export FICHEROTMP=tareas.tmp
$ export FICHEROLOG=tareas.log
$ gestortarea
Gestor de tareas activado
$ activatarea -h 20:05:30 ls -l | more > back.txt
Tarea 1 programada a las 20:05:30
$ activatarea -h 21:50:25 sort < lista.txt > listaor.txt
Tarea 2 programada a las 21:50:25
$ activatarea -h ::35 cat < 11.txt >> back.txt
Tarea 3 programada cada ::35
$ activatarea -l
Lista de tareas pendientes
1 20:05:30 ls -l | more > back.txt
2 21:50:25 sort < lista.txt > listaor.txt
3 ::35 cat < 11.txt >> back.txt
$ activatarea -r 2
$ activatarea -p 5 ls -l > back.txt
$ gestortarea -t
Gestor de tareas desactivado
$
```

#### 10.4 PRÁCTICA 4

Realizar un programa (tree.c) mediante llamadas al sistema que muestre el árbol de directorios de una ruta.

El formato es el siguiente:

```
tree [-a] [ruta]
```

donde

- a Parámetro opcional. Muestra también los archivos.
- Ruta Parámetro opcional. Indica la ruta.

En el caso de no indicar parámetros se partirá de la ruta actual.

#### 10.5 PRÁCTICA 5

Realizar un programa que calcule el producto escalar en paralelo (PEP) de dos vectores de tamaño  $n$  teniendo en cuenta que se tiene que llamar *pep.c*

El formato es el siguiente:

\$ pep [t] n

donde

*t* es un parámetro opcional. Indica que se tiene que imprimir una traza de lo que está haciendo cada proceso (recoger datos, operarlos, enviarlos, etc.). Si no se especifica se imprime el resultado.

*n* es un parámetro obligatorio. Indica el tamaño del vector.

Para implementar la multiplicación se considerará lo siguiente:

Se dispone únicamente de tres tipos de procesos:

Proceso de Entrada/Salida (pep). Solicita los datos por teclado, los distribuye, recoge el resultado final y los imprime por pantalla.

Proceso Multiplicador. Multiplica los dos datos de su entrada estándar y envía el resultado por su salida estándar.

Proceso Sumador. Suma los dos datos de su entrada estándar y envía el resultado por su salida estándar.

El paso de información se realizará mediante tubos

#### 10.6 PRÁCTICA 6

Un problema clásico en el estudio de los Sistemas Operativos es aquél que implica la existencia de procesos que han de acceder de forma concurrente a un recurso compartido, sobre el cual se realizan dos operaciones distintas. Se trata del problema de los Lectores y Escritores.

Por tanto, existirán en el sistema dos clases de procesos: un proceso Lector se caracteriza porque accede al recurso de modo que no altera el contenido del mismo; un proceso Escritor, por el contrario, accede al recurso crítico con la posibilidad de modificar su contenido.

De este modo, puede haber varios procesos Lectores accediendo concurrentemente al recurso sin importar que se superpongan, puesto que nunca comprometerán su consistencia. Sin embargo, cada Escritor ha de acceder de forma exclusiva al recurso, y no debe nunca entrelazarse con otro proceso.

Se pide realizar un programa que sincronice el acceso de varios procesos concurrentes a un recurso crítico.

El formato de ejecución es el siguiente:

```
$lecesc -n [-monitor] fich_ent fich_sal fich_sec_cri
```

-n: -1 Prioridad Escritores –2 Prioridad Lectores –3 FIFO

fich\_ent: Tendrá el siguiente formato:

Identificador Proceso	Inicio	Duración Crítica	Sección Crítica
1	1	3	
2	2	4	
3	2	3	
A	3	2	
B	4	1	

0..9 indica que es un lector y A..B indica que es un escritor.

El fichero estará ordenado por la columna Inicio.

No se pueden repetir identificadores de proceso.

fich\_sec\_cri: Fichero de texto que contiene un número y simula la sección crítica.

fich\_sal: Fichero donde se escribe la traza. Esta traza depende si se ha especificado el parámetro monitor.

### [SIN MONITOR]

Tiempo	Id	Recurso	Acción	Valor
1	1	SEC_CRI	ENTRA	5
2	1	SEC_CRI	SALE	5
3	A	SEC_CRI	ENTRA	5
5	A	SEC_CRI	SALE	6
.....				

Con la opción MONITOR, además de monitorizar la sección crítica, se debe monitorizar las operaciones de los semáforos utilizados, P y V.

En la sección crítica, los lectores al entrar leerán del fichero fich\_sec\_cri el valor y escribirán la traza; al salir, leerán el fichero de nuevo y escribirán la traza. Los escritores, al entrar leerán el valor del fichero y escribirán la traza; incrementarán el valor en uno y al salir escribirán la traza con el nuevo valor y actualizarán el fichero fic\_sec\_cri.

Cuando el recurso es un semáforo, valor valdrá el contenido del contador del semáforo.

### 10.7 PRÁCTICA 7

Realizar un programa llamado `barbero.c` que resuelva el problema del barbero durmiente mediante semáforos. El programa admitirá por la línea de comandos el número de sillas, el tiempo que está abierta la barbería (número máximo de segundos que funciona la barbería) y un archivo de traza que indica el tiempo de llegada de cada cliente y el tiempo que tarda en cortarse el pelo.

Un ejemplo de ejecución es el siguiente:

```
$ cat traza.txt
A 5 5
B 6 5
C 7 5
D 8 3
E 12 3
$ barbero 3 30 traza.txt
Tiempo          Acción
15:30:00        Barbero se duerme
15:30:05        A se sienta en silla
15:30:05        Barbero se despierta
15:30:05        A se corta el pelo
15:30:06        B se sienta en silla
15:30:07        C se sienta en silla
15:30:08        D no se sienta y se va
15:30:10        A termina y se va
15:30:11        B se corta el pelo
15:30:12        E se sienta en silla
15:30:16        B termina y se va
15:30:17        C se corta el pelo
15:30:22        C termina y se va
15:30:22        E se corta el pelo
15:30:25        E termina y se va
15:30:23        Barbero se duerme
$
```



## **REFERENCIAS**

1. A. Silberschatz y P. B. Galvin, 1999. *Sistemas Operativos*. Addison Wesley.
2. W. Stallings, 2001. *Sistemas Operativos*. Prentice Hall.
3. F. M. Márquez, 1996. *UNIX. Programación avanzada*. Rama.
4. K. A. Robbins y S. Robbins, 1997, *UNIX. Programación práctica*. Prentice Hall.
5. Syed Mansoor Sarwar, 2002. *El libro de UNIX*. Addison Wesley.
6. J. Tackett y D. Gunter, 2000. *Linux 4ª Edición*. Prentice Hall.
7. F. Maciá Pérez y A. Soriano Payá, 1999. *El Shell Korn. Manual de usuario y programador*. Servicio de Publicaciones de la Universidad de Alicante.
8. J. Carretero; F. García, 2001. *Sistemas Operativos. Una visión aplicada*. Mc Graw-Hill.
9. J. Carretero; F. García; F. Pérez, 2002. *Prácticas de Sistemas Operativos*. Mc Graw-Hill.
10. J. Aranda, M.a A. Canto, J. M. De la cruz, S. Dormido. *Sistemas Operativos: teoría y problemas*. 2002. Sanz y Torres.
11. Harvey M. Deitel, Paul J. Deitel, David R. Choffnes, 2003. *Operating Systems (3rd Edition)*. Prentice Hall.