Comunicació i sincronització

José Ramón Herrero Zaragoza
Enric Morancho Llena
Dolors Royo Vallés

PID_00169373

UOC
Universitat Oberta de Catalunya

Índex

Introducció..... 5

Objectius..... 6

1. Comunicació i sincronització..... 7

2. Primer escenari: compartició de recursos..... 9
2.1. La sincronització de processos 9
2.1.1. Per què és necessària la sincronització? 9
2.1.2. La secció crítica 13
2.2. Semàfors 15
2.2.1. Definició de semàfor 15
2.2.2. Utilització dels semàfors 16
2.2.3. Consideracions sobre la implementació dels semàfors 21
2.3. Semàfors en Linux 21

3. Segon escenari: memòria no compartida..... 24
3.1. Pas de missatges 24
3.1.1. Característiques generals 25
3.1.2. Exemple: l'exclusió mútua mitjançant missatges 27
3.1.3. Pas de missatges en UNIX (cues de bytes) 28
3.2. Els senyals de programari (senyals *software, signals*) 33
3.2.1. Descripció 33
3.2.2. Linux: senyals POSIX 35

4. *Deadlocks* (abraçades mortals o interbloquejos)..... 44

Resum..... 47

Activitats..... 49

Exercicis d'autoavaluació..... 49

Solucionari..... 53

Glossari..... 57

Bibliografia..... 58

Annex..... 59

Introducció

En aquest mòdul didàctic estudiem les possibilitats que ofereix el sistema operatiu perquè dos o més processos o fluxos d'execució puguin cooperar. Per fer-ho, el SO facilitarà mecanismes de comunicació i de sincronització. Aquests mecanismes acostumen a rebre el nom d'*IPC* (*inter-process communication*).

Estudiarem dos escenaris: el cas que els processos que cooperen comparteixin recursos (una part de l'espai lògic o dispositius) i el cas que no comparteixin (o no vulguin compartir) l'espai lògic. A cada escenari, veurem el repertori típic de crides al sistema que el SO ha d'oferir i també les problemàtiques que poden aparèixer. En especial, introduirem la problemàtica dels *deadlocks* (abraçades mortals o interbloquejos), que poden aparèixer especialment entre processos cooperatius que comparteixen memòria o dispositius.

Objectius

En els continguts didàctics d'aquest mòdul trobareu les eines bàsiques per a assolir els objectius següents:

1. Ser conscients de la necessitat de sincronitzar i comunicar els processos que cooperen concurrentment per acomplir una tasca comuna.

2. Saber que la compartició de recursos lògics del sistema (els dispositius, el codi, les variables compartides...) dóna lloc a la necessitat de sincronitzar els processos que els comparteixen per garantir que l'accés dels processos a aquests recursos s'efectua correctament (en exclusió mútua). Els processos també comparteixen els recursos del sistema computador a escala física, però aquesta compartició és gestionada pel sistema operatiu i és transparent als processos. En aquest mòdul sempre ens referirem a la compartició de recursos lògics.

3. Conèixer els avantatges i desavantatges dels diversos mecanismes que ofereix un sistema operatiu per garantir l'accés en exclusió mútua als recursos lògics compartits.

4. Saber que totes les eines que serveixen per a assegurar l'accés exclusiu als recursos del sistema són difícils d'utilitzar i que s'ha de tenir molta cura de no produir situacions inconsistents o bloquejadores quan s'hi treballa.

5. Conèixer les característiques generals d'un mecanisme de comunicació basat en pas de missatges.

6. Saber utilitzar les eines proporcionades per algun sistema operatiu concret per a comunicar i sincronitzar processos.

1. Comunicació i sincronització

En sistemes concurrents els processos cooperen en el càlcul i la realització de tasques diverses i comparteixen recursos lògics[1]. En aquests sistemes els processos han de sincronitzar el seu accés als objectes compartits i han de poder intercanviar informació entre ells.

(1)Els dispositius, el codi i les variables són alguns dels recursos que comparteixen els processos.

En general, es considera que hi ha dues maneres d'establir una cooperació entre els processos:

1) La **sincronització**: aquest procediment permet l'accés concurrent dels processos a objectes del sistema que requereixen un accés seqüencial i assegura la integritat i la consistència del sistema.

La sincronització entre processos en sistemes concurrents és imprescindible perquè el sistema es mantingui coherent i, a més, és necessària a l'hora d'accedir als recursos compartits, tant a les estructures de dades com als dispositius. Com que el sistema operatiu no sap quines són les intencions dels processos, tots aquells processos que vulguin compartir algun objecte dintre del sistema s'han de posar d'acord per tal que l'accés que s'ha de portar a terme sigui coherent i no tingui cap tipus d'error. Per a poder establir els protocols d'accés, el sistema i alguns llenguatges de programació ofereixen eines de sincronització entre els processos.

Exemples

Exemples clars d'objectes del sistema als quals accedeixen de manera concurrent els processos són els següents:

- Les variables compartides per tots els processos dintre del sistema.
- Els dispositius de naturalesa no compartida, com la impressora.

La utilització de protocols de sincronització adequats als processos que utilitzen recursos comuns pot ser complicada i pot tenir conseqüències bastant negatives per al sistema.

En aquest mòdul didàctic presentem amb detall els principals problemes que planteja la sincronització i descrivim els semàfors com a mecanisme de sincronització que la majoria de sistemes operatius sol oferir.

2) La **comunicació**: en processos concurrents normalment cal intercanviar resultats parcials, s'ha d'enviar informació de l'estat dels processos o intercanviar informació en general. Els diferents sistemes operatius han proposat diversos mecanismes que permeten comunicar certes quantitats d'informació entre els processos.

Alguns d'aquests mecanismes són els següents:

- La **memòria compartida**, que permet comunicar processos mitjançant l'ús de variables compartides.

- El **pas de missatges**, un mecanisme relativament senzill que integra tasques de sincronització i comunicació entre processos en sistemes centralitzats i sistemes distribuïts.

- Els **senyals**, interrupcions de programari que poden rebre els processos per indicar que s'ha produït un determinat succés; per exemple, que ha expirat un temporitzador, que s'ha produït una baixada de tensió, que s'ha generat una sortida de rang en operacions aritmètiques (*overflow*), etc.

A continuació veurem com podem fer aquestes operacions en dos escenaris: quan els processos comparteixen algun recurs lògic (per exemple, un rang d'adreces de l'espai lògic) i quan els processos no comparteixen l'espai lògic.

2. Primer escenari: compartició de recursos

En aquest escenari considerarem diferents processos (o fluxos d'execució) que comparteixen algun recurs lògic. Per exemple, un rang d'adreces lògiques de memòria (on s'emmagatzema alguna estructura de dades compartida) o un dispositiu d'entrada/sortida. La utilització de variables compartides és una manera bastant habitual de comunicació que els processos fan servir dintre del sistema. En aquest apartat veurem els problemes que sorgeixen quan diversos processos comparteixen un objecte lògic del sistema. Introduirem els semàfors, una eina que ens pot ajudar a solucionar aquest problema, i veurem la interfície de semàfors oferta per un sistema operatiu.

2.1. La sincronització de processos

2.1.1. Per què és necessària la sincronització?

Vegem per què és necessària la sincronització dels processos amb un cas concret. Un exemple clar de variable compartida[2] és el cas d'una variable comptador, que anomenarem `treball_pendent`, que és modificada per diferents processos (usuari i gestor); tenim el següent:

(2)Suposem que els processos poden compartir memòria.

a) Els diferents processos usuari que s'estan executant concurrentment en el sistema l'actualitzen (la incrementen) cada cop que s'envia un treball[3] a la impressora. Així, aquesta variable compartida indica quants treballs pendents d'imprimir hi ha en el sistema.

(3)Els treballs que els processos usuari envien a la impressora són fitxers.

b) El procés gestor de la impressora envia els treballs per imprimir i actualitza el valor del comptador decrementant-lo cada cop que finalitza la impressió d'un treball.

Si l'actualització d'aquesta variable comptador s'efectua sense cap tipus de restricció es poden generar inconsistències, és a dir, resultats erronis. En el cas del nostre exemple aquests resultats erronis podrien repercutir en el procés d'impressió, de manera que es podria deixar d'imprimir algun treball, o bé intentar imprimir algun treball que no existeix.

Vegem com es poden generar aquestes inconsistències analitzant el nostre exemple. Com ja hem indicat abans, un **procés usuari** a l'hora d'enviar un treball per imprimir incrementa la variable `treball_pendent` (figura 1).

El **procés gestor de la impressora** cada cop que s'acaba d'imprimir un treball decrementa la variable `treball_pendent` (figura 2).

```
...
treball_pendent = treball_pendent + 1
...
```

Figura 1. Fragment del codi del procés usuari

```
...
treball_pendent = treball_pendent - 1
...
```

Figura 2. Fragment del codi del procés gestor

Així doncs, la variable compartida `treball_pendent` indica quants treballs estan pendents de ser enviats a la impressora.

El principal problema que es planteja amb el codi que acabem de presentar, i en general amb el codi programat en qualsevol llenguatge d'alt nivell, és que durant la compilació del programa una ordre o instrucció d'alt nivell es tradueix en una o més ordres de llenguatge màquina.

Per exemple, en una màquina RISC (*reduced instruction set computer*) el codi generat un cop s'ha compilat el codi del nostre exemple podria ser el que indiquem a continuació: en el cas del **codi del procés usuari** (figura 3) i en el cas del **codi del procés gestor de la impressora** (figura 4).

```
LOAD R0,treball_pendent  ;Copiar el contingut de treball_pendent al registre R0
INC R0                   ;Incrementar R0
STORE treball_pendent,R0 ;Emmagatzemar R0 a treball_pendent
```

Figura 3. Codi assemblador corresponent al fragment de codi del procés usuari

```
LOAD R1,treball_pendent  ;Copiar el contingut de treball_pendent al registre R1
DEC R1                   ;Decrementar R1
STORE treball_pendent,R1 ;Emmagatzemar R1 a treball_pendent
```

Figura 4. Codi assemblador corresponent al fragment de codi del procés gestor

Els detalls de la codificació no són importants, i el que ens interessa notar aquí és que durant l'execució de l'ordre d'alt nivell per a incrementar o decrementar la variable `treball_pendent` es fan còpies temporals del seu contingut en registres del sistema[4]. Aquestes còpies locals, com les que es fan als registres `R0` i `R1`, poden ser inconsistents amb el valor de la variable de la memòria en un determinat instant de l'execució del procés. Aquesta inconsistència és la que pot portar a les dues situacions errònies esmentades:

(4)Els registres del sistema són registres de maquinari i no han de ser necessàriament diferents per a cada còpia temporal.

1) No-impressió d'un treball

Considerem, per exemple, que la variable `treball_pendent` té un valor inicial igual a 3 quan el gestor de la impressora executa les ordres en llenguatge màquina que s'han especificat anteriorment de la manera següent (figura 5):

a) En primer lloc fa una còpia del valor de la variable al registre `R1`. En aquest moment el valor de `R1` coincideix amb el valor de la variable `treball_pendent`, 3.

b) A continuació decrementa el valor del registre, que passa a valer 2. Ara el valor de la còpia és diferent del valor de la variable `treball_pendent` a la memòria, 3, i, per tant, hi ha una inconsistència.

Línies de codi		Variable comptador	Registres	
Procés en espera	Procés actiu	treball_pendent	R0	R1
...	...	3	...	...
LOAD R0, treball_pendent	**LOAD R1, treball_pendent**	3	...	3
INC R0	**DEC R1**	3	...	2
STORE treball_pendent, R0	STORE treball_pendent, R1			
...	...			

Figura 5. El procés gestor comença a decrementar la variable `treball_pendent`

Si en aquest punt de l'execució del codi màquina passa alguna cosa en el sistema[5] que fa que el procés gestor es deixi d'executar, el sistema queda inconsistent.

(5)Un canvi de context perquè expira el quàntum del procés gestor, una interrupció d'un dispositiu, etc.

En aquest estat d'inconsistència, el pitjor que pot passar és que un procés usuari intenti enviar un treball per imprimir. Llavors, quan el procés vol incrementar el valor de la variable `treball_pendent`, passa el següent (figura 6):

a) Es fa una còpia de la variable al registre `R0`, i `R0` val 3, perquè la variable `treball_pendent` continua tenint el valor 3, ja que el gestor no l'havia actualitzada.

b) S'incrementa el valor del registre `R0`, que passa a valer 4.

c) S'actualitza la variable `treball_pendent`, que passa a valer 4.

Línies de codi		Variable comptador	Registres	
Procés en espera	Procés actiu	treball_pendent	R0	R1
...	...	3	...	2
STORE treball_pendent, R1	**LOAD R0, treball_pendent**	3	3	2
	INC R0	3	4	2
	STORE treball_pendent, R0	4	4	2
	...			

Figura 6. El procés usuari incrementa la variable `treball_pendent`

Al cap d'un cert temps, quan el procés gestor es torna a executar (figura 7), com que s'havia interromput just quan havia d'executar l'ordre `STORE`, el primer que executa és `STORE treball_pendent, R1`. El valor de `R1` en el procés gestor era 2, i per tant el valor de la variable `treball_pendent` a la memòria s'actualitza i passa a ser 2.

Línies de codi		Variable comptador	Registres	
Procés en espera	Procés actiu	treball_pendent	R0	R1
...	...	4	4	2
...	**STORE treball_pendent, R1**	2	4	2
	...			

Figura 7. El procés gestor finalitza el decrement de la variable `treball_pendent`

Després d'aquesta seqüència d'operacions, el valor de `treball_pendent` no és correcte. Havíem partit d'un valor de 3, el procés gestor l'ha decrementat un cop i un procés usuari l'ha incrementat un cop. Per tant, el valor final de la variable hauria de ser 3 i, en canvi, el que hem aconseguit és assignar-li un valor igual a 2.

En la pràctica vol dir que en el sistema hi ha un treball que no s'imprimirà.

2) Intent d'impressió d'un treball no existent

El cas simètric a l'anterior és aquell en què l'usuari inicia l'operació d'actualització de la variable i no la finalitza perquè és interromput (figura 8 i figura 9). Aquest procés implica que el valor final de la variable considerada, `treball_pendent`, sigui erroni i igual a 4.

Línies de codi		Variable comptador	Registres	
Procés en espera	Procés actiu	treball_pendent	R0	R1
...	...	3	...	...
LOAD R1, treball_pendent	**LOAD R0, treball_pendent**	3	3	...
DEC R1	**INC R0**	3	4	...
STORE treball_pendent, R1	STORE treball_pendent, R0			
...	...			

Línies de codi		Variable comptador	Registres	
Procés en espera	Procés actiu	treball_pendent	R0	R1
...	...	3	4	...
STORE treball_pendent, R0	**LOAD R1, treball_pendent**	3	4	3
...	**DEC R1**	3	4	2
	STORE treball_pendent, R1	2	4	2
	...			

Figura 8. Procés usuari és interromput mentre incrementa la variable `treball_pendent`, i el procés gestor decrementa la mateixa variable

Línies de codi		Variable comptador	Registres	
Procés en espera	Procés actiu	`treball_pendent`	R0	R1
...	...	2	4	2
...	`STORE treball_pendent, R0`	4	4	2
	...			

Figura 9. El procés usuari finalitza l'increment de la variable `treball_pendent`

En aquest cas, el gestor de la impressora intentaria enviar a imprimir un treball que no existeix.

Tots els processos s'haurien executat correctament si les operacions d'increment i decrement que el programador escriu en el llenguatge d'alt nivell s'haguessin executat de manera indivisible o atòmica. La indivisibilitat a l'hora d'executar aquestes operacions ens assegura en la pràctica que quan un procés incrementa o decrementa la variable no deixa el processador fins que no ha finalitzat l'actualització o, dit d'una altra manera, un cop iniciada una operació d'actualització d'una variable compartida cap procés no pot iniciar una altra operació d'actualització d'aquella variable fins que no ha finalitzat la primera.

Amb aquest exemple es demostra que l'execució concurrent no sincronitzada de processos que utilitzen variables compartides pot donar lloc a errors irrecuperables. Aquest tipus de situacions reben el nom de *race conditions* (situacions de competició), perquè poden generar un resultat erroni en funció de com hagin estat planificats els processos, i cal eradicar-les dels programes.

2.1.2. La secció crítica

Davant la possibilitat de generar resultats no coherents i incorrectes durant l'execució concurrent de processos amb variables compartides és absolutament necessari trobar alguna solució. Abans de fer cap proposta vegem quin és el problema real en el funcionament descrit en l'exemple anterior. Com hem vist, l'aparició de còpies temporals i la possibilitat que un procés accedeixi al contingut d'una variable compartida en qualsevol instant abans d'assegurar-se que totes les peticions de modificació prèvies han finalitzat són la causa principal de tots els mals.

L'actualització d'una variable compartida pot ser considerada una **secció crítica**, és a dir, una seqüència ben delimitada (amb inici i fi) d'ordres que modifiquen una o més variables compartides.

Quan un procés entra en una secció crítica, ha de completar totes les ordres de dintre la secció abans que qualsevol altre procés pugui accedir a la regió crítica de la mateixa variable compartida. D'aquesta manera es garanteix que les variables compartides que es modifiquen dintre la secció tan sols són modificades per un únic procés a la vegada. Aquesta manera de procedir s'anomena **accés**

(6)És a dir, l'accés s'efectua estrictament de manera seqüencial.

a la secció crítica en exclusió mútua, i garanteix que fins que un procés no finalitza la modificació de la variable compartida, cap altre procés no la pot començar a modificar[6].

La solució al problema de l'accés en exclusió mútua a variables i dispositius compartits en el sistema ha de complir els requisits següents:

a) Assegurar l'exclusió mútua entre els processos a l'hora d'accedir al recurs compartit.

b) No fer cap tipus de suposició de la velocitat dels processos ni de l'ordre en què s'executaran.

c) Garantir que si un procés finalitza l'accés a la zona d'exclusió mútua, no afectarà la resta de processos que estiguin interessats a accedir a la secció crítica.

d) Permetre que tots els processos que estan esperant per a entrar a la secció crítica ho puguin fer en un temps finit (evitar la inanició, *starvation*).

La manera més senzilla d'assegurar l'exclusió mútua és no permetre la concurrència de processos. Però aquesta solució és inacceptable. El que ens interessa és determinar protocols i eines d'accés a les seccions crítiques que compleixin els requisits que hem enumerat.

En la majoria d'estratègies proposades per a garantir l'exclusió mútua, els processos segueixen el protocol següent (figura 10):

```
1. Petició d'accés a la secció
2. Accés a la secció crítica
3. Alliberar la secció crítica
```

Figura 10. Protocol d'entrada i sortida d'una regió crítica

Abans d'accedir a la secció crítica tots els processos formulen una petició d'accés; llavors:

1) Si la secció crítica està ocupada per un altre procés, esperen que s'alliberin o bé bloquejant-se[7] o bé duent a terme una espera activa[8]. En aquesta assignatura considerarem que els processos en espera es bloquegen i no fan una espera activa perquè, en general, això comporta una utilització millor dels recursos del sistema operatiu.

(7)Adormint-se, posant-se en alguna cua d'espera.

(8)Consultant contínuament l'estat del recurs fins que es produeixi un canvi d'estat.

2) Quan la secció crítica estigui lliure, hi entrarà un dels processos en espera i tancarà l'accés a qualsevol altre procés que estigui interessat a entrar-hi. Un cop dins, utilitzarà el recurs compartit, i quan l'ha utilitzat, alliberarà l'exclusió mútua perquè altres processos puguin utilitzar el recurs.

En l'apartat següent descrivim els semàfors, una estratègia eficient i fàcil d'utilitzar que permet l'exclusió mútua i que en general s'utilitza per a implementar altres eines de programari de nivell més alt.

2.2. Semàfors

Els semàfors són una eina de sincronització que permet garantir l'exclusió mútua d'una secció crítica entre un nombre arbitrari de processos de manera neta i senzilla.

2.2.1. Definició de semàfor

Un semàfor és una variable (assumim que és de tipus *semaphore*) que té associat un comptador. El semàfor es manipula utilitzant les tres operacions següents:

- `sem_init(semaphore s, unsigned int valor)`: inicialitza el comptador del semàfor assignant el valor indicat. El valor inicial del comptador ha de ser més gran o igual que zero.

- `sem_wait(semaphore s)`: de manera atòmica, espera que el comptador sigui més gran que zero; quan ho és, decrementa el valor del comptador (figura 11).

- `sem_signal(semaphore s)`: de manera atòmica, incrementa el valor del comptador associat al semàfor (figura 12).

```
sem_wait (semaphore s) {
   while (s <= 0) /* atòmicament */
      espera();
   s = s - 1;
}
```

Figura 11. Pseudocodi de `sem_wait`

```
sem_signal (semaphore s) {
   s = s + 1; /* atòmicament */
}
```

Figura 12. Pseudocodi de `sem_signal`

Observacions:

- Hi ha dos tipus de semàfors en funció del rang de valors que pugui assolir el comptador associat al semàfor: **binaris** (el comptador només pot valdre 0 o 1) i ***n*-aris** (el comptador pot prendre qualsevol valor més gran o igual

que zero). Els semàfors binaris són els que implementen de manera més natural l'accés en exclusió mútua a una regió crítica.

- Cal notar que les operacions fan les seves accions de manera atòmica. Suposem que el comptador associat a un semàfor val 0 i dos processos estan esperant, fent un `sem_wait`, que el comptador prengui un valor més gran que zero. Si ara algun procés executa un `sem_signal` sobre aquest semàfor i incrementa el comptador del semàfor, únicament un dels processos que executa el `sem_wait` veurà que el comptador val 1 i el decrementarà; l'altre procés haurà de continuar esperant fins a que es faci un altre `sem_signal`.

- Si diversos processos estan esperant, fent un `sem_wait`, que el comptador del semàfor prengui un valor més gran que zero i algun procés executa un `sem_signal` sobre aquest semàfor, es planteja un problema d'elecció. Quin dels processos en espera ha de veure aquest increment del semàfor? Inicialment, considerarem que aquesta elecció serà aleatòria, però més endavant tornarem sobre aquest tema.

- Les operacions bàsiques no retornen quin és el valor actual del comptador associat al semàfor. Algunes implementacions de semàfors ofereixen una operació per obtenir aquest valor però en aquest document no les considerarem.

- Els noms de les operacions `sem_wait` i `sem_signal` no són estàndard. A la literatura o en implementacions concretes podeu trobar que l'operació `sem_wait` es denomini `P`, `down`, `signal`, `acquire` o `pend`; l'operació `sem_signal` es pot denominar `V`, `up`, `wait`, `release` o `post`. És important no confondre les crides al sistema UNIX `wait` i `signal` amb les operacions sobre semàfors `wait` i `signal`.

2.2.2. Utilització dels semàfors

A continuació presentem tres usos possibles dels semàfors: exclusió mútua, comptador de recursos i sincronització.

Accés a la secció crítica en exclusió mútua

Per a assegurar l'accés a la secció crítica en exclusió mútua cal utilitzar els semàfors binaris o *n*-aris de la manera següent:

1) S'inicialitza el semàfor. El valor inicial del semàfor ha de ser 1 per a indicar que no hi ha cap procés dintre de la secció crítica i deixar pas al primer procés que demani permís per entrar-hi mitjançant l'operació `sem_wait`.

2) S'executa l'operació `sem_wait`. El procés que vol accedir a una zona compartida en exclusió mútua executa la funció `sem_wait` sobre el semàfor associat a la secció.

3) S'executa l'operació `sem_signal`. El procés que ha finalitzat l'execució de la zona crítica allibera l'exclusió i permet el pas a d'altres processos.

Com a exemple, vegem quin seria el codi dels processos usuaris i del procés gestor de la impressora en el cas que utilitzin semàfors per a actualitzar la variable compartida `treball_pendent`.

En un principi el sistema inicialitza la variable semàfor `exclusio`, fent servir el codi de la figura 13. Considerem el **codi d'un procés usuari** (figura 14) i el **codi del procés gestor de la impressora** (figura 15).

```
semaphore exclusio;
...
sem_init(exclusio, 1);
```

Figura 13. Definició de la variable de tipus semàfor i inicialització

```
...
/*Generar fitxer*/
...
sem_wait(exclusio);
treball_pendent = treball_pendent + 1;
sem_signal(exclusio);
...
```

Figura 14. Codi dels processos usuari

```
...
while (cert){
   /* Espera activa */
   while (treball_pendent == 0);
   sem_wait(exclusio);
   treball_pendent = treball_pendent - 1;
   sem_signal(exclusio);
   /*Enviar treball a la impressora*/
   ...
}
```

Figura 15. Codi del procés gestor

Per il·lustrar el comportament i el funcionament dels semàfors mostrem a continuació un possible escenari d'execució de tres processos usuari (`P1`, `P2`, `P3`) i el procés gestor (`G1`). Les diferents columnes indiquen quines accions porta a

terme cadascun dels processos en cada unitat de temps, quins processos estan esperant el semàfor, quin procés és dins la secció crítica i quin és el valor del semàfor a cada instant concret (figura 16).

Suposarem que el semàfor, `exclusio`, s'ha inicialitzat a 1 i que els processos usuari i el procés gestor són dins un bucle infinit. Els processos usuari generen contínuament treballs, els quals s'envien a la impressora. El procés gestor imprimeix contínuament els treballs que han generat els processos usuari.

A la taula es veu que en el temps `T0` s'inicialitza el semàfor i encara no s'ha creat cap procés. No hi ha cap treball pendent de ser imprès. A `T1`, el semàfor `exclusio` ha decrementat el seu valor fins a 0. Això indica que un procés ha rebut el permís per entrar a la secció crítica. En aquest cas ha estat el procés usuari `P1`. La resta de processos, el gestor i els usuaris, estan intentant entrar a la secció, però el mecanisme del semàfor assegura que cap altre procés no hi pugui entrar fins que `P1` no l'alliberi, mitjançant l'execució de l'operació `sem_signal(exclusio)`.

Temps	Estats dels processos				Semàfor 1-Lliure 0-Ocupat	Dintre: fora (espera)
	P1	P2	P3	G1		
T0	...	...	...	...	1	... : ...
T1	sem_wait	sem_wait	sem_wait	sem_wait	0	... : P1, P2, P3, G1
T2	Secció crítica	Esperant	Esperant	Esperant	0	P1 : P2, P3, G1
T3	sem_signal	Esperant	Esperant	Esperant	1	... : P2, P3, G1
T4	Altres operacions	Esperant	Secció crítica	Esperant	0	P3 : P2, G1
T5	sem_wait	Esperant	Secció crítica	Esperant	0	P3 : P2, G1, P1
T6	Esperant	Esperant	sem_signal	Esperant	1	... : P2, G1, P1
T7	Secció crítica	Esperant	Altres operacions	Esperant	0	P1 : P2, G1
T8	sem_signal	Esperant	Altres operacions	Esperant	1	... : P2, G1

Figura 16. Evolució de l'estat dels tres processos usuari, del procés gestor i del semàfor

Suposem que tots tres processos usuari generen els fitxers corresponents i, tal com s'ha indicat en el codi del procés usuari, a continuació examinen el valor del semàfor `exclusio` amb la intenció de decrementar-ne el valor i entrar a la secció crítica (modificar la variable). Un dels tres té èxit, encara que no podem assegurar quin acabarà entrant[9] a la secció crítica. En l'exemple, l'operació `sem_wait` que executa `P1`, després d'haver llegit el valor del semàfor a 1, s'ha d'apropiar la variable *semàfor* i evitar que la resta de processos concurrents (`P2` i `P3`) hi pugui accedir abans que sigui modificada (decrementada a 0). Aquest és el motiu pel qual el codi de l'operació `sem_wait` ha de ser indivisible. Més endavant veurem com podem aconseguir-ho.

[9] Amb aquesta utilització dels semàfors l'ordre d'accés dels processos a la secció crítica és totalment aleatori.

Un cop dins la secció crítica, el procés `P1` actualitza la variable. A continuació allibera la secció crítica executant l'operació `sem_signal(exclusio)`. En aquest instant, `P2`, `P3` i el procés gestor de la impressora tenen la mateixa probabilitat d'entrar a la secció crítica. En l'exemple és el procés `P3` el que ho aconsegueix. Es tornen a repetir les mateixes operacions i quan `P3` allibera la secció entra de nou `P1`, mentre que els processos `P2` i el procés gestor de

la impressora continuen esperant. Aquesta situació es coneix com a *inanició* i no és desitjable; més endavant, en parlar de la implementació dels semàfors, veurem com es pot evitar.

Observació: la **granularitat d'un semàfor** indica el nombre de zones crítiques a les quals està associat. Distingim dos graus de granularitat:

- **Granularitat baixa**: comporta tenir semàfors diferents per a zones crítiques diferents. Això augmenta el grau de concurrència entre els diferents processos concurrents. En general interessa tenir aquest grau de granularitat.
- **Granularitat alta**: és quan s'utilitza un mateix semàfor per a assegurar l'exclusió mútua de diferents zones crítiques (independents entre elles).

Comptador de recursos disponibles

Moltes aplicacions han de controlar la quantitat de recursos disponibles d'un determinat tipus però no necessiten saber exactament quants recursos disponibles hi ha, i només necessiten saber si n'hi ha algun; si no n'hi ha cap, han d'esperar que n'hi hagi algun de lliure.

És possible utilitzar semàfors *n*-aris en aquest entorn. El comptador associat al semàfor reflectirà el nombre de recursos disponibles.

1) S'inicialitza el semàfor. El valor inicial del semàfor ha de ser el nombre inicial de recursos disponibles (serà un nombre enter més gran o igual que zero).
2) S'executa l'operació `sem_wait`. Quan un procés necessiti un recurs, invocarà aquesta operació. Si hi ha algun recurs disponible, es decrementarà el comptador associat al semàfor; altrament, s'esperarà fins que hi hagi algun recurs disponible.
3) S'executa l'operació `sem_signal`. El procés que ha finalitzat d'utilitzar un recurs incrementa el comptador i permet que altres processos l'utilitzin.

L'exemple anterior dels processos usuari i el procés gestor de la impressora es podria haver resolt utilitzant un semàfor *n*-ari i aprofitant el comptador del semàfor per a acumular el nombre de treballs pendents (figura 17). El procés usuari incrementarà aquest comptador fent `sem_signal` (figura 18) i el procés gestor esperarà l'aparició de treballs fent `sem_wait` (figura 19). El comptador associat al semàfor indicarà el nombre de treballs pendents.

```
semaphore treballs_pendents;
...
sem_init(treballs_pendents, 0);
```

Figura 17. Definició de la variable de tipus semàfor i inicialització

```
...
while (cert){
   /*Generar fitxer*/
   ...
   sem_signal(treballs_pendents);
}
```

Figura 18. Codi dels processos usuari

```
...
while (cert){
   sem_wait(treballs_pendents);
   /*Enviar treball a la impressora*/
   ...
}
```

Figura 19. Codi del procés gestor de la impressora

Fixeu-vos que en la figura 19 el gestor es bloqueja si no hi ha treballs pendents. En canvi, el codi de la figura 15 feia una espera activa. En l'apartat 2 de l'annex teniu un exemple més complet d'utilització d'aquest tipus de semàfors.

Sincronització entre processos

Un semàfor binari o *n*-ari pot implementar una sincronització entre dos processos en què un procés espera una autorització per part de l'altre per poder continuar l'execució.

1) S'inicialitza el semàfor. El valor inicial del semàfor ha de ser 0.

2) S'executa l'operació `sem_wait`. El procés que espera l'autorització per part de l'altre procés ha d'invocar aquesta operació.

3) S'executa l'operació `sem_signal`. El procés que concedeix l'autorització ha d'invocar aquesta operació.

El procés que espera l'autorització (figura 21) executarà el codi posterior al `sem_wait` únicament quan el procés que autoritza (figura 22) hagi executat el `sem_signal`.

```
semaphore sync;
...
sem_init(sync, 0);
```

Figura 20. Definició de la variable de tipus semàfor i inicialització

```
...
sem_wait(sync);
```

```
...
```

Figura 21. Codi del procés que espera autorització

```
...
sem_signal(sync);
...
```

Figura 22. Codi del procés que autoritza

2.2.3. Consideracions sobre la implementació dels semàfors

A continuació presentem algunes consideracions sobre la implementació dels semàfors:

- L'operació `sem_wait` ha de fer una espera fins que el comptador associat al semàfor assoleixi un valor determinat. Aquesta espera es pot implementar bàsicament de dues maneres: espera activa i bloqueig. L'espera activa és l'opció més senzilla, però provoca un cert desaprofitament del processador. En funció de la quantitat de processos en execució, aquest desaprofitament pot ser inacceptable.

- Quan diversos processos estan esperant fent un `sem_wait` sobre un mateix semàfor i algun procés invoca `sem_signal`, la implementació de semàfors ha d'escollir un dels processos en espera. Es poden considerar diversos criteris, com ara aleatori, FIFO (primer d'entrar, primer de sortir o en anglès, *first-in, first-out*), prioritats... La implementació hauria de triar un criteri que no provoqui inanició (*starvation*, com a la figura 16), és a dir, evitar que un procés esperi indefinidament mentre altres processos completen el `sem_wait` sobre el mateix semàfor. El criteri FIFO evita la inanició i és el que acostumen a utilitzar les solucions basades en bloquejos.

- Alguns sistemes operatius ofereixen els semàfors dins del seu repertori de crides al sistema. Aquestes implementacions acostumen a estar basades en bloquejos. Els processos en espera passen a l'estat *Blocked* i deixen de competir per consumir temps de processador fins que el sistema operatiu detecta que s'ha fet un `sem_signal` sobre el semàfor i els passa a l'estat *Ready*.

- En l'apartat 1 de l'annex descrivim el suport que ha d'oferir el maquinari per poder implementar l'exclusió mútua.

2.3. Semàfors en Linux

Linux ofereix una implementació de semàfors POSIX dins del seu repertori de crides al sistema. En aquest subapartat presentem una descripció bàsica de la interfície de crides al sistema i de com cal utilitzar-los dins dels programes.

Tot i que la implementació ofereix semàfors amb nom (accessibles per mitjà del sistema de fitxers) i semàfors sense nom (accessibles per mitjà de memòria compartida), en aquest subapartat ens centrarem en els semàfors sense nom, que seran utilitzats per fluxos d'execució que pertanyin a un mateix procés (per tant, comparteixen l'espai lògic de memòria).

POSIX ofereix el tipus de dades `sem_t` per declarar variables de tipus semàfor. La interfície de crides al sistema és la següent (observeu que el paràmetre de tipus semàfor es passa per referència):

- `int sem_init (sem_t * sem, int pshared, unsigned int value)`: inicialitza el semàfor al valor indicat. En el paràmetre `pshared` especificarem el valor 0.

- `int sem_wait(sem_t *sem)`: anàloga a l'explicada al subapartat anterior.

- `int sem_post(sem_t *sem)`: és l'equivalent a l'operació `sem_signal` explicada al subapartat anterior.

Totes tres crides retornen 0 en cas de poder-se fer correctament i –1 en cas d'error. Un possible error està relacionat amb la recepció de senyals programari (apartat 3.2).

Com a exemple, la figura 23 mostra un programa[10] que emula el funcionament d'un motor d'explosió de quatre temps. Per fer-ho, crea 4 fluxos d'execució que se sincronitzen utilitzant semàfors per a garantir que l'ordenació dels temps sigui la correcta (admissió, compressió, explosió i escapament). Noteu que un dels semàfors ha estat inicialitzat a 1 per a permetre que el flux *Admissió* pugui "arrencar" el motor.

[10] Per a generar l'executable corresponent cal utilitzar la biblioteca de *pthreads* (paràmetre de compilació `-pthread`).

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

sem_t sync_temps[4];

char *name[4]={"Admissió", "Compressió", "Explosió", "Escapament"};

/* Codi comú per als quatre temps.
  En funció del paràmetre "par", cadascú imprimirà un missatge
  i determinarà quins semàfors utilitza
*/

void *temps(void *par)
```

```
{
 int t = (int) par;

 while (1) {
  if (sem_wait(&sync_temps[t]) < 0 ) error();

  printf("%d %s\n", t, name[t]);
  sleep(rand() % 4); /* Retard aleatori */

  if (sem_post(&sync_temps[(t+1)%4]) < 0 ) error();
 }
}

int main()
{
 int i;
 pthread_t th[4];

 /* Sincronisme Escapament -> Admissió. Inicialitzat a 1 */
 if (sem_init(&sync_temps[0], 0, 1) < 0 ) error();
 /* Sincronisme Admissió -> Compressió */
 if (sem_init(&sync_temps[1], 0, 0) < 0 ) error();
 /* Sincronisme Compressió -> Explosió */
 if (sem_init(&sync_temps[2], 0, 0) < 0 ) error();
 /* Sincronisme Explosió -> Escapament */
 if (sem_init(&sync_temps[3], 0, 0) < 0 ) error();

 /* Creació de fils */
 for (i=0; i<4; i++)
  if (pthread_create(&th[i], NULL, temps, (void*)i) < 0) error();

 /* Espera la mort dels fils */
 for (i=0; i<4; i++)
  if (pthread_join(th[i], NULL) < 0) error();

 return(0);
}
```

Figura 23. Programa d'exemple que utilitza semàfors POSIX en Linux

3. Segon escenari: memòria no compartida

En aquest escenari els processos no comparteixen l'espai lògic (perquè el SO no ho permet o perquè els processos no volen). Perquè aquests processos es puguin comunicar, el SO ha d'oferir un mecanisme de pas de missatges. Presentarem les característiques abstractes del mecanisme i a continuació descriurem les característiques concretes dels mecanismes de pas de missatges disponibles en UNIX. També presentarem els senyals de programari, que permeten apropar a l'usuari un mecanisme similar a les interrupcions de maquinari. Aquests senyals permetran que els processos pugin rebre notificacions d'esdeveniments asíncrons per part de el SO o d'altres processos, i tractar-los convenientment.

3.1. Pas de missatges

Hem vist que repartint la realització d'una activitat entre un conjunt de processos que es poden executar concurrentment és possible millorar considerablement el rendiment del sistema. El fet de dur a terme les activitats conjuntament planteja la necessitat de sincronitzar i de comunicar les dades, ja siguin dades parcials, resultats finals, etc. Com que sovint les dues accions són necessàries, s'han desenvolupat mecanismes que permeten la sincronització i alhora l'intercanvi d'informació.

El **pas de missatges** és un mecanisme bastant senzill i intuïtiu que permet la sincronització i la comunicació entre processos que s'estan executant en sistemes centralitzats o en sistemes distribuïts.

En general, un **missatge** és un conjunt d'informació que pot ser intercanviada entre un emissor i un receptor. Un missatge pot contenir qualsevol tipus d'informació. El format d'un missatge és flexible, però, per norma general, inclou camps per indicar el tipus, la longitud, els identificadors de l'emissor i del receptor, i un camp de dades. De fet, tota aquesta informació i la que es pugui afegir es classifica en els dos grups següents:

- La **capçalera**, amb un format predefinit en un sistema, conté la informació necessària perquè el missatge pugui arribar a la seva destinació correctament. Sol tenir una longitud fixa.

- El **cos**, constituït pel missatge pròpiament dit. Sol ser de longitud variable.

Els sistemes operatius actuals

Són molts els sistemes operatius actuals que proporcionen algun mecanisme que permet la comunicació dels processos mitjançant el pas de missatges. Per exemple, UNIX ofereix una implementació específica d'aquest mecanisme.

3.1.1. Característiques generals

Un sistema que suporta missatges, generalment ofereix dos tipus d'operacions, que són *enviar_missatge* (*send*) i *rebre_missatge* (*receive*). La implementació específica del pas de missatges en un sistema afecta directament els paràmetres que s'han de passar a aquestes operacions. Així doncs, passem a descriure amb més detall algunes qüestions que s'han de tenir en compte a l'hora de definir un mecanisme de pas de missatges, que són:

- Els missatges directes i els missatges indirectes.
- La comunicació síncrona i asíncrona.
- La longitud dels missatges.

Els missatges directes i els missatges indirectes

Si el tipus de missatge és **directe**, el procés emissor ha d'indicar quin és el procés receptor, i a l'inrevés, el procés receptor ha d'indicar quin és el procés de qui vol rebre el missatge.

Per exemple, si tenim dos processos `A` i `B`, i `A` vol enviar un missatge a `B`, es podria executar en els dos processos el codi següent (figura 24):

```
Procés A
... send(B, missatge)


Procés B
... receive(A, missatge)
```

Figura 24. Pas de missatges directe

en què `A` i `B` han de ser els identificadors dels processos dintre el sistema i la variable *missatge* conté la informació que s'estan intercanviant. Aquest tipus de comunicació s'anomena **comunicació simètrica**, ja que cada emissor ha de conèixer tots els possibles receptors, i a l'inrevés.

En cas que el receptor no conegui *a priori* qui és l'emissor que envia el missatge, es parla de **comunicació asimètrica**.

Quan el tipus de missatge és **indirecte**, el procés emissor no ha d'indicar quin és el procés receptor i el procés receptor no ha d'indicar quin és el procés emissor. Els missatges s'envien i es reben mitjançant unes estructures que fan d'intermediàries. Aquestes estructures s'anomenen **bústies (*mailboxes*)**, a causa del seu funcionament.

Per exemple, si el procés `A` vol enviar un missatge al procés `B` mitjançant una bústia, els processos podrien executar les ordres següents (figura 25):

```
Procés A
... send(bustia1, missatge)

Procés B
... receive(bustia1, missatge)
```

Figura 25. Pas de missatges indirecte

El procés `A` deixa el missatge a la bústia designada, `bustia1`, mitjançant l'operació `send(bustia1, missatge)` i el procés `B` quan el vol llegir el treu de la bústia mitjançant l'operació `receive(bustia1, missatge)`. Quan es disposa d'un mecanisme d'aquest tipus també són necessàries altres operacions per al manteniment de la bústia, com ara les de crear i destruir una bústia.

Considerarem que intentar llegir (*receive*) d'una bústia buida bloqueja l'execució del procés i intentar escriure (*send*) en una bústia plena també ho fa. La comunicació mitjançant bústies permet la comunicació d'un a un, d'un a molts, de molts a un i de molts a molts (segurament amb certes restriccions).

La comunicació síncrona i asíncrona

La **comunicació síncrona** es basa en l'intercanvi síncron de missatges. Aquest intercanvi implica que l'emissor i el receptor han d'acabar la transferència d'informació en el mateix moment, s'han de sincronitzar[11].

[11] La telefonia clàssica seria un exemple convencional d'aquest tipus de comunicació.

En sistemes síncrons, l'operació `send` és bloquejadora; per tant, si un emissor vol enviar un missatge i el receptor encara no ha efectuat l'operació complementària, `receive`, l'emissor queda bloquejat. Per tant, només hi pot haver un únic missatge pendent per a cada parella d'emissor/receptor.

Els avantatges de la comunicació síncrona són la implementació fàcil i un cost baix de recursos. A més a més, aquest mode de comunicació permet establir un punt de sincronisme entre l'emissor i el receptor: l'emissor pot estar segur que el missatge ha estat rebut.

La **comunicació asíncrona** no bloqueja el procés, i el sistema operatiu s'encarrega de desar el missatge temporalment a la memòria fins que es porta a terme l'operació `receive` complementària. En aquesta manera de procedir, l'emissor, un cop ha enviat el missatge, pot continuar l'execució independentment del que facin els receptors[12].

(12)El correu postal seria un exemple convencional d'aquest tipus de comunicació.

Amb la comunicació asíncrona augmenta el grau de concurrència del sistema. Com a inconvenient cal esmentar la gestió dels vectors de memòria intermèdia per a emmagatzemar els missatges pendents, que és l'aspecte més complicat d'aquest mètode de comunicació.

La longitud dels missatges

Finalment, hem de considerar la longitud dels missatges, que pot ser fixa o variable. Hem de tenir en compte, a l'hora de decidir la longitud dels missatges, que cal arribar a un compromís entre la flexibilitat dels missatges de longitud variable i la senzillesa i la facilitat de gestió dels missatges de longitud fixa.

3.1.2. Exemple: l'exclusió mútua mitjançant missatges

Un mètode molt senzill d'implementar l'accés en exclusió mútua en una regió crítica és mitjançant missatges indirectes. En l'exemple que es mostra a continuació es crea una bústia que anomenem `bustia`. Per demanar accés a la secció crítica, el procés llegeix de la bústia amb l'operació `receive` i llavors:

- Si la bústia és buida el procés es queda bloquejat esperant que algú li escrigui el missatge.
- Si la bústia és plena el procés llegeix el missatge, deixa la bústia buida, i accedeix a la secció.

Ens interessa que el primer procés que demani accés a la secció crítica pugui passar. Per tal d'aconseguir-ho, la bústia s'inicialitza introduint-hi un missatge de qualsevol longitud. En el codi que es presenta a la figura 26, la bústia desenvolupa un paper equivalent al d'un semàfor binari.

```
receive(bustia, missatge);  /* entrada exclusió mútua */
/* secció crítica */
...
send(bustia, missatge);     /* sortida exclusió mútua */
```

Figura 26. Implementació de l'exclusió mútua utilitzant missatges indirectes

També seria possible implementar semàfors *n*-aris amb bústies, així es permetria que a la bústia hi pogués haver fins a *n* missatges.

3.1.3. Pas de missatges en UNIX (cues de bytes)

UNIX ofereix diversos mecanismes de pas de missatges de tipus indirecte, asíncron i de mida variable. En aquesta secció considerem tres mecanismes: les *pipes*, les *named pipes* i els *sockets*.

Conceptualment, els tres mecanismes són molt similars. Ofereixen una cua de bytes que es gestionarà seguint la política FIFO. Alguns processos (productors) afegiran informació a un extrem de la cua i altres processos (consumidors) en podran extreure de l'altre extrem. Per tant, seran mecanismes propicis per a implementar models de comunicació productor-consumidor.

Els tres mecanismes difereixen en els requisits que imposen als processos que els pretenen utilitzar i en el seguit de crides al sistema que han d'invocar per utilitzar-los.

- **Les *pipes*** permeten comunicar dos processos que tinguin algun tipus de parentiu (pare i fill, dos germans...).

- **Les *named pipes*** permeten comunicar dos processos del sistema que tinguin accés al directori del sistema de fitxers on s'hagi creat un fitxer de tipus *named pipe*.

- **Els *sockets*** permeten comunicar processos que s'executen en màquines diferents.

La taula 1 mostra les crides del sistema UNIX que cal invocar per a crear, obrir, llegir i escriure sobre aquests mecanismes. Un cop que el mecanisme està disponible per al procés (el procés d'usuari disposa d'un *file descriptor* per a operar sobre el mecanisme de comunicació), tots tres mecanismes s'accedeixen amb les mateixes crides al sistema `read` i `write` i des del punt de vista de l'usuari programador són idèntics.

		Pipes	***Named pipes***	***Sockets***
Crides al sistema	Crear	pipe	mknod	socket
	Obrir		open	bind, listen, accept (servidors), connect (clients)
	Llegir	read		
	Escriure	write		
	Tancar	close		

Taula 1. Crides del sistema UNIX que permeten accedir a les *pipes*, *named pipes* i *sockets*

Tot i que les lectures i escriptures sobre aquests mecanismes es fan amb les mateixes crides al sistema que permeten accedir a fitxers ordinaris (`read` i `write`), hi ha tres casos particulars que es tracten de manera especial en el cas de treballar sobre cues de bytes:

- **Lectura sobre cua buida**. Quan un procés consumidor intenta llegir d'una cua que en aquest moment es troba buida, la cua pot ser buida perquè a) el procés consumidor "és més ràpid" que el procés productor, o b) el procés consumidor ja ha consumit tota la informació que el productor havia de produir. Normalment, al procés consumidor li interessarà tractar de manera diferent tots dos casos; en el cas a), el més raonable seria esperar que el productor produeixi alguna cosa; en el cas *b*), el més raonable seria rebre la notificació de final de fitxer. Perquè el SO pugui saber en quin cas ens trobem, utilitzarà una dada que coneix amb exactitud: el nombre de *file descriptors* d'escriptura oberts actualment sobre aquesta cua. Si aquest nombre és més gran que 0, considerarà que estem en el cas *a*); si el nombre és 0, considerarà que estem en el cas *b*). Per tant, en el cas *a*) la crida `read` es bloquejarà fins que algun productor escrigui alguna cosa a la cua; en el cas *b*) la crida `read` retornarà immediatament i indicarà que s'han llegit 0 caràcters.

  Els *file descriptors* d'escriptura

  El fet que el SO utilitzi el nombre de *file descriptors* d'escriptura oberts fa que deixar obert innecessàriament algun *file descriptor* d'escriptura sobre una cua de bytes pugui provocar comportaments anòmals en els nostres programes. És aconsellable tancar els *file descriptors* tan aviat com deixin de ser necessaris.

- **Escriptura sobre cua plena**. Internament, el SO assigna una mida a aquestes cues. Si es produeix informació sobre una cua a un ritme superior al que la informació és consumida, arribarà un moment en què la cua s'omplirà. Si en aquest moment s'intenta afegir més informació, el comportament per defecte és que la crida `write` es bloquegi[13] fins que hi hagi prou espai disponible per a escriure totes les dades.

[13] No considerem el cas que una crida `write` intenti escriure un nombre de bytes superior a la mida interna de la cua de bytes.

- **Escriptura sobre cua sense procés consumidor**. UNIX considera que aquesta situació és anòmala i probablement provocada per un error en algun dels processos involucrats. Per tant, avisa al procés productor utilitzant un mecanisme semblant a les excepcions vistes al mòdul 2, el mecanisme dels senyals programari (els veurem en l'apartat 3.2). Si el procés consumidor no està preparat per a tractar aquest senyal, el SO el farà avortar.

Altres diferències entre *pipes* i fitxers ordinaris són que no és possible utilitzar la crida `lseek` per a reposicionar el punter de lectura-escriptura sobre una `pipe` (la gestió de les dades és estrictament FIFO) i que les *pipes* són volàtils. Si aturem el sistema operatiu (*shutdown*), es perd tota la informació que no hagi estat llegida d'una *pipe*.

Exemple: *pipes*

Una *pipe* és un dispositiu que no es pot obrir de manera explícita mitjançant la crida al sistema `open`; es crea en el moment en què s'invoca la crida al sistema `pipe`, i es destrueix quan l'últim procés que el té obert el tanca. La crida `pipe` crea el dispositiu i li associa dos *file descriptors*, un de lectura i un d'escriptura. Si ara el procés crea processos fills (mitjançant la crida al sistema `fork`), aquests heretaran els *file descriptors* del seu procés pare i podran accedir a aquesta mateixa *pipe*.

La figura 27 mostra un exemple d'utilització de les *pipes* (assumim que cap crida al sistema retornarà error).

```
int descFitxer[2], n, estat;
char buf[512];

estat = pipe(descFitxer);  /* Creació de la pipe */
switch(fork()){
    case 0:
        /* El procés fill llegeix del canal 0 i escriu a la pipe */
        close(descFitxer[0]);
        while ((n = read(0, buf, sizeof(buf))) > 0) {
            write(descFitxer[1], buf, n);
        }
        close(descFitxer[1]); /* Permet que el pare detecti fi de fitxer */
        break;

    default:
        /* El procés pare llegeix de la pipe i escriu al canal 1 */
        close(descFitxer[1]); /* Permet detectar el final de transmissió quan el
                                 fill acabi */

        while ((n = read(descFitxer[0], buf, sizeof(buf))) > 0) {
            write(1, buf, n);
        }
        close(descFitxer[0]);
}
...
```

Figura 27. Fragment de programa que utilitza una *pipe* per a comunicar un procés fill amb un procés pare.

El procés demana crear una *pipe*. El SO li retornarà dos *file descriptors* (un de lectura i un d'escriptura) per accedir a la *pipe*, com podeu veure a la figura 28, on es mostra esquemàticament l'estat del procés (espai lògic, PCB[14], taula de *file descriptors)* i del SO (taula de punters de lectura/escriptura).

(14)*Process control block*, estructura de dades gestionada pel sistema operatiu en què emmagatzema la informació necessària per a gestionar cada procés.

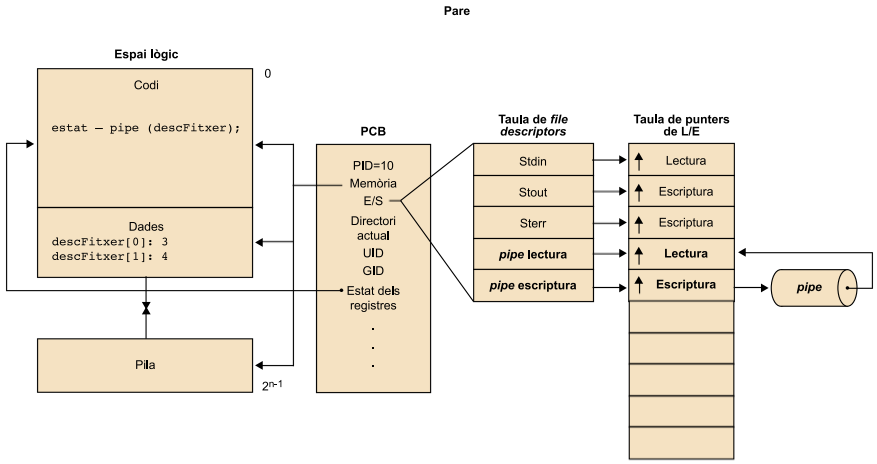

Figura 28. Estat del procés just després d'haver creat la *pipe*

Perquè aquesta *pipe* pugui comunicar dos processos, cal invocar la crida al sistema `fork`. D'aquesta manera, el procés fill hereta del pare els *file descriptors*, entre els quals trobem els creats amb la crida `pipe: descFitxer[0]` és el de lectura, i `descFitxer[1]` el d'escriptura.

Després de la creació del fill, per establir un canal de comunicació unidireccional entre el fill i el pare, tots dos processos tanquen el canal del sentit de la comunicació que no utilitzaran.

Finalment, el procés fill llegeix dades de la seva entrada estàndard (canal 0, *stdin*) i les escriu a la *pipe*. El procés pare llegeix les dades de la *pipe* i les escriu a la seva sortida estàndard (canal 1, *stdout*).

La figura 29 mostra l'estat dels dos processos mentre estan executant codi corresponent a les sentències `while`.

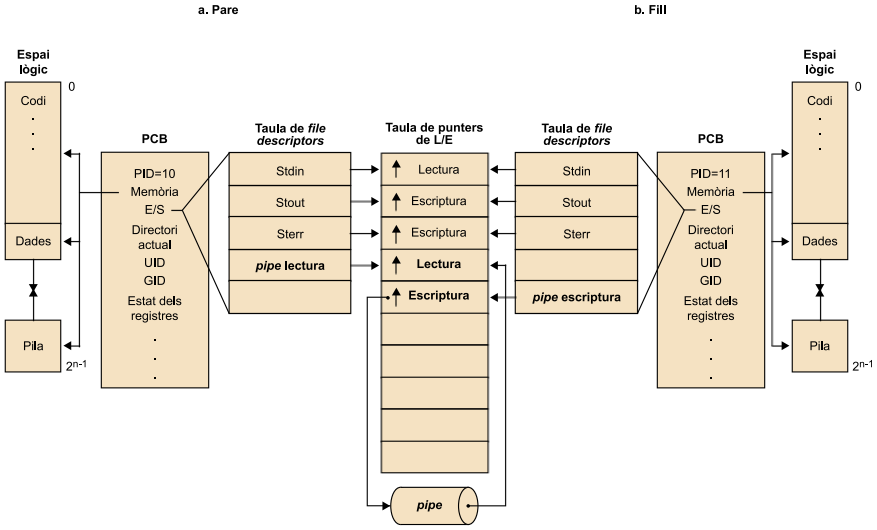

Figura 29. Estat del procés pare i del procés fill mentre els processos es comuniquen

Finalment, la figura 30 mostra un exemple complet d'un programa que usa *pipes*. Es tracta d'un programa que emula les crides al sistema que ha d'executar l'intèrpret d'ordres quan l'usuari introdueix la línia d'ordres `ps aux | grep`

`getty` (el metacaràcter | dels intèrprets d'ordres UNIX permet comunicar la sortida estàndard d'un procés amb l'entrada estàndard d'un altre seguint el paradigma de comunicació productor-consumidor). El programa crea una *pipe* i dos processos fills; un executarà l'ordre `ps` i l'altre l'ordre `grep`; la sortida estàndard del primer fill i l'entrada estàndard del segon fill estaran comunicades gràcies a la *pipe* i als reencaminaments efectuats (crida al sistema `dup2`). Noteu que els executables `ps` i `grep` són totalment aliens al fet d'estar escrivint/llegint sobre una *pipe*; `ps` escriu sobre la seva sortida estàndard (que en aquest cas està reencaminada al canal d'escriptura sobre la *pipe*) i `grep` llegeix de la seva entrada estàndard (que en aquest cas està reencaminada al canal de lectura sobre la *pipe*).

```
#include <unistd.h>
#include <sys/wait.h>

int
main (int argc, char *argv[])
{
 int p[2], st;

 if (pipe (p) < 0)
  error ();

 switch (fork ())
  {
  case -1:
   error ();

  case 0:
   if (dup2 (p[0], 0) < 0)
    error ();
   if (close (p[0]) < 0)
    error ();
   if (close (p[1]) < 0)
    error ();
   execlp ("grep", "grep", "getty", NULL);
   error ();

 default:
        switch (fork ())
         {
        case -1:
         error ();
        case 0:
         if (dup2 (p[1], 1) < 0)
          error ();
         if (close (p[0]) < 0)
```

```
            error ();
           if (close (p[1]) < 0)
            error ();
           execlp ("ps", "ps", "aux", NULL);
           error ();
          }

}
 if (close (p[0]) < 0)
  error ();
 if (close (p[1]) < 0)
  error ();
 if (wait (&st) < 0)
  error ();
 if (wait (&st) < 0)
  error ();
 return (0);
}
```

Figura 30. Programa que emula les crides al sistema que ha d'executar l'intèrpret d'ordres quan l'usuari introdueix l'ordre `ps aux | grep getty`

Vegeu també

Consulteu la documentació addicional de l'assignatura i el manual de crides al sistema per entendre tots els paràmetres de les crides al sistema utilitzades.

3.2. Els senyals de programari (senyals *software*, *signals*)

3.2.1. Descripció

Com hem vist, un SO defineix una nova màquina amb una semàntica més elaborada que la de la màquina física sobre la qual s'executa. Malgrat aquest augment del nivell semàntic, el SO conserva moltes de les funcions i els mecanismes de què disposa el maquinari. Un d'aquests mecanismes que reprodueix és el de les interrupcions. Els processos poden necessitar ser informats d'esdeveniments que succeeixen de manera imprevista o asíncrona en qualsevol instant de l'execució d'un programa, per tal de poder tractar-los i poder continuar l'execució del programa.

Per exemple, un procés pot necessitar saber si s'ha produït un error en l'accés a la memòria, a fi de demanar que s'augmenti l'àrea assignada a una certa variable. O pot necessitar saber si un cert terminal sobre el qual treballa ha estat apagat, per a acabar l'aplicació correctament, etc.

Els **senyals de programari** són l'eina que proporciona el sistema operatiu per traslladar el mecanisme de les interrupcions a l'àmbit del procés. Igual que el mecanisme de maquinari de les interrupcions, els senyals donen suport a un conjunt ampli de situacions diverses que els processos han de conèixer i atendre amb una certa urgència.

La **procedència dels senyals de programari** ens permet classificar-los de la manera següent:

1) Els **dispositius d'entrada/sortida**, que necessiten informar el procés de situacions que poden donar lloc a un error si el procés no en té coneixement. Són situacions com, per exemple, la desconnexió d'un terminal, la pulsació d'una tecla de control per part de l'usuari, etc.

2) Les **excepcions**, que són provocades pel procés de manera involuntària durant l'execució d'una instrucció del llenguatge màquina o a causa de la saturació d'un element de maquinari (*overflow*), d'un accés a memòria invàlid o d'una ordre anòmala[15].

[15]Són instruccions del llenguatge màquina anòmales la divisió per zero, l'arrel quadrada d'un nombre negatiu, etc.

3) Les **crides explícites al sistema**, que es poden fer per a enviar senyals a un procés determinat. Per a poder-ho fer el procés que envia el senyal ha de tenir permís. En general, només es permeten enviar senyals entre processos del mateix domini de protecció (l'administrador pot enviar senyals a qui vulgui). Es tracta de senyals com ara l'ordre per a eliminar un procés, o senyals per a sincronitzar l'execució de processos que ho necessitin. Per exemple, un procés que té com a funció inicialitzar una estructura de dades determinada podria utilitzar un senyal per a informar els processos usuaris d'aquesta estructura que ja ha estat inicialitzada.

4) El **rellotge del sistema**, que és utilitzat pels processos quan necessiten dur a terme certes accions a intervals concrets de temps. Per exemple, un procés que estigui esperant una certa entrada des d'un mòdem pot demanar ser avisat dintre d'un cert lapse de temps (*timeout*) a fi de detectar si la línia s'ha tallat o no.

5) Finalment, un procés pot rebre un senyal com l'**efecte lateral d'una crida al sistema**. Per exemple, un procés pare pot rebre un senyal que li indica que un dels seus processos fills ha estat destruït.

El tractament que dóna un procés a un senyal que li arriba pot ser d'un dels tres tipus següents:

1) Un tractament definit per l'usuari mateix. El procés indica mitjançant una crida al SO quin procediment s'ha d'executar quan arribi un cert senyal.

2) No donar cap tractament (ignorar l'esdeveniment). El procés pot decidir no fer cas del senyal i, per tant, quan arribi el senyal continuarà l'execució normalment com si no hagués passat res.

3) Un tractament donat pel SO. Algunes circumstàncies que provoquen senyals són degudes a un mal funcionament del procés si no s'utilitzen les mesures necessàries. En aquests, casos si el procés no ha definit un tractament, llavors cal que el sistema en proporcioni un. Normalment aquest tractament

per defecte porta associada la destrucció del procés al qual anava destinat el senyal. Per exemple, si un procés efectua un accés incorrecte a la memòria i el procés no ho soluciona, el SO es veu en l'obligació de destruir el procés i informar-ne el seu procés pare.

Així doncs, un procés serà destruït pel SO si rep un senyal no esperat a causa d'un error en la seva execució, a causa d'un esdeveniment imprevist en un dels dispositius als quals accedeix, o per l'actuació deliberada d'un altre procés. En aquest cas, el SO és l'encarregat d'enviar l'estat de finalització al procés pare a fi d'indicar-li el motiu de la destrucció.

Finalment, hem de fer notar que la programació dels senyals és una informació pròpia de cada procés que configura l'entorn d'execució i, com a tal, ha de formar part del PCB i s'ha de tenir en compte en el moment de la creació d'un nou procés.

3.2.2. Linux: senyals POSIX

El sistema operatiu UNIX ofereix als processos el mecanisme dels senyals. Malauradament, hi ha diverses interfícies de senyals (System V, BSD, POSIX...) que presenten diferències de funcionalitat molt significatives. Com la interfície de senyals més completa i que permet fer programes més robustos és la interfície POSIX (la utilitzada per Linux), en aquest subapartat en presentem una breu descripció. La interfície POSIX també rep el nom de *reliable signals*, mentre que la System V també rep el nom de *traditional signals*.

Terminologia

A continuació descrivim alguns termes utilitzats al llarg d'aquesta secció:

- **Generació del senyal**: moment en què es produeix un senyal.
- **Tractament del senyal**: rutina d'atenció per a un determinat senyal (pot ser el tractament per defecte proporcionat pel SO o una rutina proporcionada per l'usuari).
- **Dipòsit del senyal**: moment en què es comença a executar la rutina de tractament d'un senyal.
- **Senyal pendent**: senyal que ha estat generat però que encara no ha estat dipositat.
- **Programació d'un senyal**: fet d'associar un tractament a un senyal.
- **Captura d'un senyal**: execució d'una rutina proporcionada per l'usuari per a atendre un determinat senyal.

- **Senyal bloquejat**: si un procés bloqueja un senyal, el SO no dipositarà temporalment els senyals d'aquest tipus que es generin sobre el procés. Quan el procés desbloquegi aquest senyal, el SO dipositarà els senyals pendents. Seria equiparable a inhibir temporalment les interrupcions maquinari.

El terme *bloquejat* i els senyals pendents

El terme *bloquejat* té significats diferents en funció del context, perquè es pot aplicar a processos i a senyals. Recordem que un procés bloquejat és aquell que temporalment no pot competir per utilitzar el processador perquè està esperant algun esdeveniment.

El nombre de senyals pendents d'un determinat senyal està limitat. En algunes versions POSIX, només hi pot haver un senyal pendent de cada tipus, mentre que altres versions POSIX permeten acumular diversos senyals pendents d'un mateix tipus (*queued signals*). En aquest document assumirem que no serà possible acumular senyals pendents.

- **Senyal ignorat**: si un procés ignora un senyal, el SO descartarà els senyals d'aquest tipus que es generin sobre el procés i no els dipositarà.

- **Conjunt de senyals bloquejats**: atribut de tot procés que indica quins senyals té bloquejats el procés. S'emmagatzema al PCB del procés i es manipula utilitzant una crida al sistema.

La figura 31 mostra un diagrama de temps en què apareixen aquests termes. Es representa l'execució d'un procés que genera, bloqueja, desbloqueja i ignora senyals.

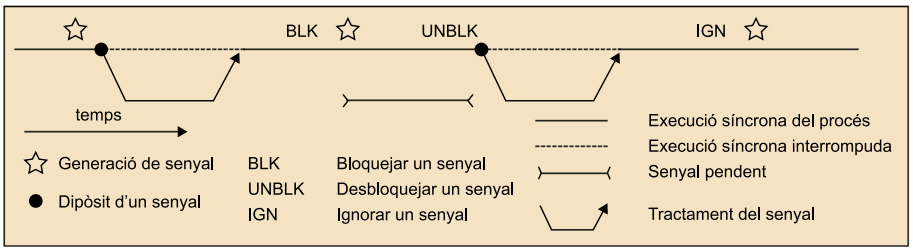

Figura 31. Terminologia utilitzada pels senyals POSIX

Llista de senyals

POSIX defineix un seguit de senyals amb un significat concret; altres implementacions de senyals presenten variacions respecte al tractament i al nombre de senyals possibles. A la taula 2 mostrem alguns dels senyals definits per POSIX amb el tractament per defecte que assigna el SO. El significat d'aquests tractaments és:

- **EXIT**: destrucció del procés,

- **EXIT+CORE**: destrucció del procés i generació d'un fitxer *core* que conté l'estat del procés en el moment del dipòsit del senyal,

- **IGNORE**: ignorar el senyal.

Vegeu també

En el mòdul 3 hem parlat d'aquests fitxers `core` quan un procés intenta accedir a una adreça de memòria invàlida.

- **STOP**: aturar l'execució del procés.

- **CONT**: reprendre l'execució d'un procés aturat.

Nom del senyal	Significat del senyal	Tractament per defecte
SIGHUP	S'ha produït un tall de la línia de comunicació, generalment amb un terminal o un mòdem.	EXIT
SIGINT	S'ha premut la seqüència de tecles de control associada a aquest senyal (típicament Ctrl-C).	EXIT
SIGABRT	Avorta l'execució del procés.	EXIT + CORE
SIGFPE	S'ha produït una excepció de coma flotant.	EXIT + CORE
SIGKILL	Destruir el procés. El senyal no es pot ignorar, bloquejar o programar.	EXIT
SIGSEGV	S'ha produït un accés indegut a memòria.	EXIT + CORE
SIGPIPE	S'ha escrit en una *pipe* sense cap lector.	EXIT
SIGALRM	S'ha produït una expiració del temporitzador.	EXIT
SIGTERM	Se sol·licita que el procés acabi ("Prepare to die").	EXIT
SIGUSR1	Disponible per a usos propis de les aplicacions.	EXIT
SIGUSR2	Disponible per a usos propis de les aplicacions.	EXIT
SIGCHLD	Ha mort algun procés fill.	IGNORE
SIGSTOP	S'ha premut la seqüència de tecles de control associada a aquest senyal (típicament Ctrl-Z). El senyal no es pot ignorar, bloquejar o programar.	STOP
SIGCONT	Reprèn l'execució d'un procés que hagi estat aturat amb SIGSTOP.	CONT

Taula 2. Llista d'alguns senyals POSIX: nom del senyal, significat associat i acció per defecte duta pel SO

Cada senyal (SIGHUP i SIGKILL...) té associat un codi numèric enter; per exemple, el SIGKILL té associat el valor 9[16]. Per a facilitar la llegibilitat del codi, en els fragments de codi d'exemple utilitzarem els noms simbòlics dels senyals i no el seu codi numèric.

[16]Els habituats a treballar amb algun intèrpret d'ordres UNIX, probablement han fet servir l'ordre `kill -9 pid` per a matar un procés. Aquesta ordre genera el senyal número 9 (SIGKILL) sobre el procés amb identificador `pid`.

Conjunts de senyals (*signal sets*)

Diverses crides al sistema relacionades amb la interfície de senyals POSIX estan parametritzades amb una variable que representa un conjunt de senyals. Per exemple, ens pot interessar bloquejar els senyals SIGUSR1, SIGUSR2 i SIGTERM. Per a fer-ho, necessitem definir una variable de tipus "conjunt de senyals" que contingui aquests tres tipus de senyals. POSIX ens ofereix un tipus de dades (sigset_t) i una sèrie de funcions per a manipular aquestes variables. La figura 32 enumera aquestes funcions i en descriu la funcionalitat; també mostra un parell d'exemples: crear un conjunt amb els senyals SIGUSR1 i SIGUSR2 i crear un conjunt amb tots els senyals llevat del SIGTERM.

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set); /* Inicialitza "set" al conjunt buit */
int sigfillset(sigset_t *set); /* Inicialitza "set" amb tots els signals */
int sigaddset(sigset_t *set, int signum); /* Afegeix "signum" a "set" */
int sigdelset(sigset_t *set, int signum); /* Elimina "signum" de "set" */
int sigismember(const sigset_t *set, int signum); /* Indica si "signum" pertany a "set" */

/* Exemples d'ús */
sigset_t u12, /* Contindrà els signals SIGUSR1 i SIGUSR2 */
      noterm; /* Contindrà tots els signals llevat del SIGTERM */

sigemptyset(&u12); sigaddset(&u12, SIGUSR1); sigaddset(&u12, SIGUSR2);
sigfillset(&noterm); sigdelset(&noterm, SIGTERM);
```

Figura 32. Funcions que permeten manipular variables de tipus `sigset_t` i dos exemples d'utilització

Programació d'un senyal

La crida al sistema que permet definir la rutina d'atenció a un senyal és la crida `sigaction` (figura 33). El seu primer paràmetre és l'identificador de senyal del qual volem modificar la rutina de tractament. El segon paràmetre és un punter a una estructura (`struct sigaction`) que indica quin serà el nou tractament:

- En el camp `sa_handler` indicarem quina rutina volem que atengui aquest senyal (en cas que vulguem ignorar el senyal, cal indicar el valor `SIG_IGN`, i en cas de voler recuperar el tractament per defecte cal indicar el valor `SIG_DFL`). Quan aquesta rutina s'executi, rebrà com a paràmetre el codi numèric corresponent al senyal que n'ha provocat l'execució (és possible associar la mateixa rutina a senyals diferents).

- En el camp `sa_mask`, de tipus "conjunt de senyals", indicarem quins senyals volem que estiguin bloquejats mentre s'executi la rutina d'atenció.

- En el camp `sa_flags` podrem modificar el comportament per defecte dels senyals POSIX. El valor 0 fa que el tractament d'aquest senyal segueixi l'estàndard POSIX.

```
#include <signal.h>

struct sigaction {
  void (*sa_handler)(int);
  sigset_t sa_mask;
  int sa_flags;
};

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Figura 33. Interfície de la crida al sistema `sigaction`

La crida retorna el valor 0 si s'ha pogut reprogramar el senyal i –1 en cas d'error. Un error típic és intentar reprogramar un senyal que el SO no permet reprogramar (per exemple, el `SIGKILL`). A més a més, al tercer paràmetre ens retorna el punter a una estructura `struct sigaction` en què podem recuperar quina era fins al moment la programació d'aquest senyal.

En l'estàndard POSIX, la programació del senyal és vàlida fins al moment que es torni a programar aquest mateix senyal (en l'estàndard System V, només era vàlida fins al primer dipòsit d'un senyal d'aquest tipus).

En cas d'invocar la crida al sistema `fork`, el procés fill hereta la programació dels senyals del procés pare. En cas d'invocar alguna crida al sistema `exec`, tots els senyals recuperen la programació per defecte llevat d'aquells que hagin estat programats com a `SIG_IGN`.

Generació de senyals

POSIX proporciona diverses crides al sistema per generar senyals. En comentarem dues: `kill`[17] i `alarm` (figura 34).

(17)El nom `kill` (matar) és així perquè les primeres implementacions de senyals es feien servir únicament per a provocar la mort de processos.

```
int kill(pid_t pid, int sig);
unsigned int alarm(unsigned int seconds);
```

Figura 34. Crides al sistema que permeten generar senyals

La crida `kill` genera un senyal "immediatament" sobre un determinat procés. Té com a paràmetre l'identificador de procés sobre el qual volem generar el senyal i el tipus de senyal per generar. La crida retornarà error si intentem generar un senyal sobre un procés que pertany a un altre usuari o sobre un procés inexistent.

La crida `alarm` permet programar el temporitzador associat al procés. Té com a paràmetre el nombre de segons[18] (de temps real) que han de transcórrer abans que expiri el temporitzador. Quan el temporitzador expiri, el SO generarà un senyal `SIGALRM` sobre el procés. Per a generar un nou `SIGALRM`, caldrà programar novament el temporitzador. En cas d'invocar la crida `alarm` amb el paràmetre 0, el SO ignorarà la darrera programació del temporitzador.

(18)POSIX ofereix altres temporitzadors amb una resolució més fina, però no són objecte d'aquest document.

Conjunt de senyals bloquejats

Tot procés té un atribut al seu PCB que indica quins senyals té bloquejats en aquest moment. La gestió d'aquest atribut s'ha de fer utilitzant la crida al sistema `sigprocmask`. El primer paràmetre de la crida (`how`) indica el tipus de modificació que volem fer (`SIG_BLOCK`, `SIG_UNBLOCK` o `SIG_SETMASK`); la crida permet modificar el valor d'aquest atribut afegint-hi (`SIG_BLOCK`), eli-

minant (SIG_UNBLOCK) o assignant directament (SIG_SETMASK) el conjunt de senyals indicat en el segon paràmetre (set). En el tercer paràmetre, la crida ens pot retornar el valor de l'atribut abans de fer aquest canvi.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

/* l'atribut es modifica en funció de valor de "how" (SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK):

   SIG_BLOCK:    atribut = atribut UNIÓ set;
   SIG_UNBLOCK:    atribut = atribut INTERSECCIÓ COMPLEMENTARI(set)
   SIG_SETMASK:    atribut = set;
*/
```

Figura 35. Crida al sistema sigprocmask

El valor d'aquest atribut s'hereta en invocar la crida fork, i es manté en invocar alguna de les crides exec. Si el valor d'aquest atribut es modifica dins de la rutina d'atenció a algun senyal (o utilitzant el camp sa_mask de l'estructura struct sigaction de la crida al sistema sigaction), en acabar l'execució de la rutina d'atenció, el SO restaurarà el valor previ d'aquest atribut. És a dir, l'execució d'una rutina de tractament d'un senyal no pot modificar permanentment el valor d'aquest atribut.

La figura 36 mostra un fragment d'un codi d'exemple que utilitza aquesta crida. Es pretén bloquejar temporalment el senyal SIGTERM mentre el procés està escrivint dades a un fitxer (s'entén que cal fer-ho sense modificar la resta de senyals bloquejats pel procés). Es vol evitar que es pugui dipositar aquest senyal mentre el procés escriu dades en el fitxer, perquè això provocaria la mort del procés i podria deixar el fitxer en un estat incoherent. El programa bloqueja temporalment el senyal, amb la qual cosa, si el senyal es genera, es dipositarà únicament quan totes les dades hagin estat escrites en el fitxer. Es presenten tres opcions per a fer-ho; la primera i la segona no són del tot correctes perquè no tenen en consideració quins senyals poden estar bloquejats actualment; la tercera manera té en compte totes les possibilitats.

```
sigset_t term, old;

sigemptyset(&term); sigaddset(&term, SIGTERM);

/* Opció 1: Errònia si ja tenim algun signal bloquejat */
sigprocmask(SIG_SETMASK, &term, NULL);
escriptura_fitxer();
sigprocmask(SIG_UNBLOCK, &term, NULL);

/* Opció 2: Errònia si ja tenim el SIGTERM bloquejat */
sigprocmask(SIG_BLOCK, &term, NULL);
```

```
escriptura_fitxer();
sigprocmask(SIG_UNBLOCK, &term, NULL);

/* Opció 3 : Correcta */
sigprocmask(SIG_BLOCK, &term, &old);
escriptura_fitxer();
sigprocmask(SIG_SETMASK, &old, NULL);
```

Figura 36. Exemples d'utilització de la crida `sigprocmask`

Una altre ús possible del bloqueig de senyals és garantir l'accés en exclusió mútua a una estructura de dades a la qual s'accedeixi tant des del programa principal com des d'una rutina d'atenció a un senyal. Cal bloquejar aquest senyal mentre el programa principal estigui modificant l'estructura de dades.

Espera del dipòsit d'un senyal

Una de les operacions més habituals que es fan amb senyals és la sincronització, en què un procés ha d'esperar fins que es dipositi un determinat tipus de senyal. Assumim que el procés està esperant un senyal de tipus `SIGUSR1` i que la rutina de tractament d'aquest senyal posa a TRUE la variable global `usr1`.

Una primera aproximació consistiria en una espera activa (figura 37), és a dir, que el procés estigui consultant contínuament el valor d'aquesta variable fins que canviï de valor. Això penalitza la resta de processos en execució en el sistema i, en general, no seria una solució acceptable.

```
int usr1 = FALSE;

void ras_usr1(int signum)
{
 usr1 = TRUE;
}

int main()
{
 struct sigaction act;

 /* Programació de la rutina d'atenció al SIGUSR1 */
 act.sa_handler = ras_usr1;
 sigemptyset(&act.sa_mask);
 act.sa_flags = 0;
 sigaction(SIGUSR1, &act, NULL);
 while (usr1 == FALSE); /* Espera activa */
 ...
}
```

Figura 37. Espera del dipòsit d'un senyal amb una espera activa

Les versions tradicionals de senyals oferien la crida al sistema `pause` però, en general, la utilització d'aquesta crida fa que els programes no siguin fiables (*reliable*). La crida `pause` fa esperar el procés fins que es dipositi un senyal sobre el procés. Si utilitzem aquesta crida (figura 38) el codi resultant es pot comportar de manera errònia en cas que el `SIGUSR1` es dipositi un cop que s'ha verificat que `usr1` té el valor `FALSE` però abans d'invocar la crida al sistema `pause`; s'executarà la rutina d'atenció al senyal i el procés esperarà el dipòsit d'un senyal que ja ha estat dipositat.

```
if (usr1 == FALSE) pause();
```

Figura 38. Espera del dipòsit d'un senyal amb la crida al sistema `pause`

Per solucionar aquest problema, la interfície de senyals POSIX proporciona la crida al sistema `sigsuspend` (figura 39). Aquesta crida fa dues operacions de manera atòmica[19]: fer que `mask` sigui el nou valor de l'atribut del conjunt de senyals bloquejats pel procés i fer que el procés s'esperi[20] fins al dipòsit d'un senyal no bloquejat. Quan es dipositi el senyal, s'executarà el tractament associat i, en acabar, es restaurarà el valor previ de l'atribut i el procés continuarà l'execució.

[19] Entre altres coses, el SO garanteix que no es dipositarà cap senyal sobre el procés mentre es fan aquestes dues operacions.

[20] En les crides `pause` i `sigsuspend`, el SO no implementa aquesta espera fent servir una espera activa. El SO fa que el procés passi a l'estat *Blocked* fins que es dipositi un senyal sobre el procés.

```
int sigsuspend(const sigset_t *mask);
```

Figura 39. Crida al sistema `sigsuspend`

Utilitzant aquesta crida podrem donar una nova solució al problema anterior. La figura 40 mostra el codi resultant assumint que volem permetre tractar qualsevol altre senyal mentre esperem el dipòsit del `SIGUSR1`.

```
sigset_t m, old;

sigemptyset(&m); sigaddset(&m, SIGUSR1);

sigprocmask(SIG_BLOCK, &m, &old);
sigemptyset(&m);

while (usr1 == FALSE)
 sigsuspend(&m); /* Esperant qualsevol senyal */

sigprocmask(SIG_SETMASK, &old, NULL);
```

Figura 40. Crida al sistema `sigsuspend`

Si mentre executem el `SIGUSR1` volem permetre el tractament únicament dels senyals que no estiguessin bloquejats, caldria introduir algunes modificacions en el codi (figura 41).

```
sigset_t m, old;
```

```
sigprocmask(SIG_BLOCK, NULL, &m); /* Obtenim signals bloquejats */
sigaddset(&m, SIGUSR1);

sigprocmask(SIG_BLOCK, &m, &old);
sigdelset(&m, SIGUSR1);

while (usr1 == FALSE)
 sigsuspend(&m);/* Esperant SIGUSR1 o qualsevol senyal no bloquejat */

sigprocmask(SIG_SETMASK, &old, NULL);
```

Figura 41. Espera del dipòsit d'un senyal amb la crida al sistema `sigsuspend`

En els exercicis d'autoavaluació es proposen diversos exercicis relacionats amb el conjunt de senyals bloquejats que té el procés i amb la utilització de la crida al sistema `sigsuspend`.

Dipòsit de senyals sobre processos bloquejats

Mentre un procés està bloquejat (per exemple, llegint del teclat o d'una *pipe* buida, esperant la mort d'un procés fill, fent un `sem_wait` sobre un semàfor), és possible que el SO dipositi un senyal sobre aquest procés. Si el senyal no està bloquejat, s'executarà el tractament associat. Ara bé, si aquest tractament no provoca la mort del procés, què passarà amb la crida al sistema invocada pel procés?

- En alguns casos, el SO avortarà la crida al sistema (la crida retornarà un –1 com a resultat i a la variable `errno` –codi d'error– trobarem el valor `EINTR`) i el procés continuarà l'execució.

- En altres casos, el SO reiniciarà la crida al sistema, amb la qual cosa el procés es tornarà a bloquejar.

En funció del tipus de crida al sistema i de com hagi estat definit el tractament del senyal (camp `sa_flags` de l'estructura `struct sigaction` de la crida al sistema `sigaction`) estarem en un cas o en un altre. Per trobar més informació pel que fa a això, podeu consultar el manual del sistema operatiu tot executant l'ordre `man 7 signal` sobre algun sistema Linux.

En tot cas, aconsellem fer un control d'errors exhaustiu a les crides al sistema per tal de detectar crides al sistema que el SO no hagi pogut servir.

4. *Deadlocks* (abraçades mortals o interbloquejos)

En els sistemes operatius actuals, que contenen un nombre de dispositius i un nombre de processos considerablement gran, la possibilitat de l'execució concurrent/paral·lela d'aquests processos i la possibilitat d'accedir concurrentment als dispositius augmenta el rendiment del sistema de manera significativa.

Tot aquest maremàgnum de processos i recursos als quals s'accedeix a la vegada de manera compartida o exclusiva pot generar fàcilment situacions d'interbloqueig (*deadlock*), que fan que el sistema deixi de fer una feina útil.

Un **interbloqueig** (*deadlock*) és una situació en què un grup de processos estan indefinidament bloquejats sense cap possibilitat que continuï l'execució. En termes generals, la causa d'aquest bloqueig indefinit és que cada procés del grup ha adquirit un conjunt de recursos necessaris per operar i està esperant altres recursos que han estat assignats a altres processos del grup. Això crea una situació en què cap procés no pot continuar l'execució.

Els interbloquejos habitualment apareixen en sistemes concurrents com a conseqüència de l'assignació de recursos del sistema de manera incontrolada, és a dir, sense mirar o establir cap protocol que intenti evitar la situació. Molts cops, és el programador mateix dels processos el qui crea aquesta situació de manera involuntària.

Suposem que un sistema té un dispositiu impressora i un dispositiu disc, als quals s'ha d'accedir en exclusió mútua. Es creen dos semàfors binaris, `disc` i `impressora`, inicialitzats a 1 per indicar que els recursos estan lliures. En el sistema creem dos processos que s'executen concurrentment: `P0` (figura 42) i `P1` (figura 43).

```
...
sem_wait(disc)
sem_wait(impressora)
/*Utilitzar el disc i la impressora*/
...
sem_signal(impressora)
sem_signal(disc)
...
```

Figura 42. Codi del procés `P0`

```
...
sem_wait(impressora)
sem_wait(disc)
/*Utilitzar el disc i la impressora*/
...
sem_signal(disc)
sem_signal(impressora)
...
```

Figura 43. Codi del procés `P1`

Podem observar que l'execució seqüencial d'aquests dos processos és totalment correcta i no comporta cap error. En canvi, a l'hora d'executar-se concurrentment, i segons com s'entrellacen les operacions dels dos processos, es pot generar una situació d'interbloqueig. Per exemple, si es produeix la seqüència següent:

1. El procés `P0` demana el recurs `disc` i se li concedeix, `sem_wait` (`disc`).

2. El procés `P1` demana el recurs `impressora` i se li concedeix, `sem_wait` (`impressora`).

3. El procés `P1` demana el recurs `disc` i es queda bloquejat, `sem_wait` (`disc`), ja que el disc ha estat assignat.

4. El procés `P0` demana el recurs impressora i es queda bloquejat, `sem_wait` (`impressora`), ja que aquest recurs ha estat assignat al procés `P1`.

A partir d'aquest moment, el procés `P1` s'espera que s'alliberi el recurs `disc`, però aquest està assignat al procés `P0`, que no el pot alliberar perquè està bloquejat esperant el recurs `impressora`. Aquest tampoc no serà alliberat, ja que el procés que el té assignat, el procés `P1`, també està bloquejat. En definitiva, s'ha creat un bucle d'espera del qual no hi ha sortida.

Perquè es produeixi un interbloqueig s'han de donar les quatre condicions següents simultàniament:

- **Exclusió mútua**: s'ha d'accedir als recursos en exclusió mútua.

- **Retenció i espera**: cada procés té assignats certs recursos en exclusió mútua i espera que se n'alliberin d'altres.

- **No-expropiació**: els recursos que un procés té assignats tan sols s'alliberen a petició explícita del procés mateix; el sistema operatiu no els els pot prendre.

- **Espera circular**: els processos bloquejats formen una cadena circular on cada procés està bloquejat esperant un recurs que ja ha estat assignat a un altre procés de la cadena.

S'han proposat diversos mecanismes per a tractar de solucionar aquest problema. Les diferents propostes es poden classificar en tres categories segons el seu objectiu:

1) Prevenir l'interbloqueig: aquestes propostes pretenen assegurar que una de les quatre condicions necessàries per a l'interbloqueig no es produirà mai. El més senzill és evitar l'espera circular tot ordenant de manera lineal els recursos del sistema, a fi que els processos estiguin obligats a demanar els recursos que necessitin en un ordre creixent dintre de l'ordenació especificada. A l'exemple anterior, es podria haver previngut l'interbloqueig si tots els processos haguessin demanat els recursos en el mateix ordre (per exemple, primer el disc i després la impressora).

2) Evitar l'interbloqueig: aquestes propostes es basen en el fet d'assignar només els recursos disponibles que no puguin generar cap tipus d'interbloqueig. Aquesta tècnica necessita dur un control detallat de quins recursos s'han assignat, a quins processos s'han assignat i si aquests processos estan esperant algun altre recurs.

3) Detectar i recuperar l'interbloqueig: les propostes que fan servir aquesta tècnica deixen que apareguin interbloquejos, ja que assignen lliurement els recursos segons els van demanant els processos. De tant en tant es comprova si s'ha produït un interbloqueig mirant si hi ha algun cicle en el graf de recursos assignats. Així es determina quins processos estan bloquejats. Per trencar l'interbloqueig, el sistema operatiu reinicia alguns dels processos que estaven bloquejats.

Resum

En aquest mòdul didàctic hem estudiat en profunditat el problema de la sincronització de diferents processos i hem donat una idea de com se solucionen els problemes de comunicació de processos i els interbloquejos (*deadlocks*).

Hem vist que en l'execució concurrent de processos la **sincronització** és necessària quan aquests accedeixen a recursos compartits (principalment, dispositius i variables compartides). Sense una sincronització adequada, l'execució concurrent, en entrellaçar l'execució de diversos processos, pot produir errors de temporització i pot deixar el sistema inconsistent. Les causes d'aquestes irregularitats depenen del recurs compartit. Així, tenim les causes següents:

- En el cas de variables compartides, la principal causa d'aquest problema és l'existència de còpies temporals per part dels processos concurrents.

- En el cas dels dispositius compartits, una sincronització incorrecta pot provocar problemes d'interbloqueig.

Per a solucionar els problemes que sorgeixen en el moment de sincronitzar els processos s'han estudiat amb detall els **semàfors**, un mecanisme que permet la sincronització de manera senzilla. El principal problema que presenten és que el programador és el responsable d'utilitzar-los correctament. Els programadors n'han de fer un bon ús per evitar problemes de *deadlock* i d'inanició.

També hem vist el mecanisme de pas de missatges que, a més de permetre la sincronització, també fa possible la comunicació d'informació entre processos cooperatius i el mecanisme dels senyals de programari. Finalment hem descrit el problema de l'interbloqueig de processos (*deadlock*).

Activitats

1. Per què interessa que les zones d'exclusió siguin com més petites millor? Proposeu un exemple en què es vegin clars els possibles avantatges.

2. És necessari accedir en exclusió mútua a una variable compartida si tan sols es vol consultar el valor? Per què?

3. Demostreu que si les operacions `sem_wait` i `sem_signal` no s'executen atòmicament es pot violar la condició d'exclusió mútua.

4. Implementeu un semàfor *n*-ari utilitzant missatges indirectes amb bústies.

5. Considereu la comunicació entre processos utilitzant l'esquema de les bústies. Considereu bústies infinites (*send* no bloqueja mai).

a) Suposeu que un procés vol esperar un missatge de la *bustia-A* i un missatge de la *bustia-B* (un missatge de cada bústia). Quina seqüència de *sends* i *receives* hauria d'executar?

b) Quina seqüència de *sends* i *receives* s'hauria d'executar si el mateix procés vol esperar un missatge d'una bústia o un missatge de l'altra, o de *bustia-A* o de *bustia-B*?

6. Escriviu el seguit de crides al sistema que ha d'invocar l'intèrpret d'ordres per donar servei a la línia d'ordres `ps | grep sh | wc -l`.

7. Quan es fa el *shutdown* d'una màquina amb el sistema operatiu Linux, el SO genera el senyal `SIGTERM` a tots els processos que resten en execució, espera uns segons, i a continuació genera el senyal `SIGKILL` sobre tots els processos que resten en execució. Per què creieu que procedeix d'aquesta manera i no genera directament el senyal `SIGKILL`?

8. Treballant sobre un terminal en Linux, prémer les tecles Ctrl-C acostuma a provocar la mort del procés en execució. Podeu explicar els esdeveniments que es produeixen des que l'usuari prem Ctrl-C fins que el procés mor?

9. Suposeu que un procés UNIX mor a causa del dipòsit d'un senyal. El seu procés pare pot saber quin senyal ha provocat la mort del fill? Per respondre podeu consultar la pàgina del manual de la crida al sistema `wait`.

10. Busqueu informació sobre quines característiques de l'estàndard de senyals POSIX es poden modificar utilitzant el camp `sa_flags` de l'estructura de dades `struct sigaction` de la crida al sistema `sigaction`.

11. Escriviu un programa que calculi quina és la mida interna de les *pipes*. Podeu utilitzar senyals.

Exercicis d'autoavaluació

1. Us proposem que feu un generador de nombres aleatoris. Els nombres s'aniran obtenint d'un vector de memòria intermèdia (*buffer*) circular de mida `maxbuff` en la qual diversos fluxos aniran inserint i extraient nombres seguint l'esquema del productor/consumidor. El sistema es compon dels tres tipus de fluxos següents:

- El flux productor, que s'encarrega de crear nombres i els va introduint en un vector de memòria intermèdia amb una política d'assignació FIFO compartida per tots els fluxos. Com que aquest vector té una mida finita, els fluxos productors es bloquejaran quan sigui ple.
- El flux aleatoritzador, que s'encarrega de consumir els nombres. Ho fa agafant els 5 nombres més antics del vector de memòria intermèdia (si no n'hi ha 5, s'espera que hi siguin), aplica la funció `int processa(X1,...,X5)`, i obté un altre nombre com a resultat. Aquest nombre és introduït una altra vegada al vector de memòria intermèdia com si aquest procés fos un productor.
- El flux consumidor, que s'encarrega d'obtenir els valors aleatoris i de donar-los a qui els demani. Per fer-ho, en l'instant en què necessita un nombre, mira el valor d'un nombre qualsevol dels que hi ha al vector de memòria intermèdia sense extreure'l.

Demanem el codi esquemàtic del generador de nombres aleatoris, posant èmfasi en les parts de sincronització i exclusió mútua d'aquests tres tipus de fluxos. La solució ha de tenir en compte que hi podria haver diversos fluxos de cada tipus.

2. Tenim un vector de memòria intermèdia circular de mida *N* que conté elements d'un cert tipus i que permet comunicar diferents processos entre ells. Uns dels processos –els productors– escriuen elements mentre el vector de memòria intermèdia no sigui ple, i els altres –els consumidors– llegeixen elements d'aquest vector de memòria mentre no sigui buit.

Volem introduir el concepte de *prioritat* entre els productors i els consumidors. Cada procés productor i cada procés consumidor tindrà una prioritat associada entre `0` (la menys prioritària) i `P - 1` (la més prioritària), en què `P` és el nombre de processos que hi ha en execució. Voldrem també que un procés consumidor no accedeixi al vector de memòria intermèdia mentre hi hagi algun altre procés consumidor amb més prioritat que també vulgui accedir-hi. Anàlogament, un procés productor no ha d'accedir al vector de memòria intermèdia mentre hi hagi algun altre procés productor amb més prioritat que també hi vulgui accedir.

Implementeu la solució de manera que si donem permís a un procés per accedir al vector de memòria intermèdia, la resta de processos del mateix tipus que hi vulguin accedir hagin d'esperar que aquest l'alliberi, independentment de la prioritat dels processos que demanin permís per accedir-hi.

Heu de resoldre aquest problema fent servir tants semàfors com necessiteu i també memòria compartida entre els processos (variables compartides). Disposeu d'una funció que retorna la prioritat del procés que l'executa anomenada *int prioritat*(), la qual torna un valor entre `0` i `P - 1`.

3. Simuleu una orxateria on treballen dues persones: un orxater i un cambrer. El primer s'encarrega de fer les orxates i passar-les al cambrer perquè les pugui vendre. Els clients, que poden ser molt nombrosos, quan arriben a l'orxateria demanen un nombre variable d'orxates al cambrer. Aquest, una vegada les ha servides, avisa el client perquè les agafi. Mentre no hi ha clients, l'orxater continua preparant orxates i el cambrer espera clients nous.

```
/*Definicions i inicialització de variables compartides*/
#define true 1
int n_orxates = 0;
semaphore sem1, sem2, sem3, sem4, sem5, sem6;

void cambrer() {
int tmp, i;
while (true) {
  sem_wait(sem2);
  sem_wait(sem5);
  tmp = n_orxates;
  n_orxates = 0;
  sem_signal(sem5);
  for (i=0; i<tmp; i++) {
   sem_wait(sem1);
   servir_orxata();
}
  sem_signal(sem3);
}
}
```

Figura 44

```
void client() {
  anar_a_la_orxateria();
  sem_wait(sem4);
  sem_wait(sem5);
  n_orxates =
   quantes_orxates();
  sem_signal(sem5);
  sem_signal(sem2);
  sem_wait(sem3);
  sem_signal(sem4);
  prendre_orxates();
}
```

Figura 45

```
void orxater() {
int preparades = 0;

while (true) {
  preparar_orxata();
```

```
  sem_signal(sem1);
  sem_wait(sem6);
  preparades++;
  sem_signal(sem6);
 }
}
```

Figura 46

Ajudes: abans de començar llegiu bé totes les preguntes; us pot ajudar substituir els noms dels semàfors per altres més significatius. Les rutines *servir_orxata*, *anar_a_la_orxateria*, *quantes_orxates*, *prendre_orxates i preparar_orxata* no modifiquen cap variable ni cap semàfor compartit.

a) Amb quins valor inicialitzaríeu cadascun dels semàfors?

b) Per a què se suposa que serveixen els semàfors `sem5` i `sem6`? Són realment necessaris aquests semàfors? Per què?

c) Quina informació es desa de manera implícita al comptador del semàfor `sem1`?

d) Per a què serveix el semàfor `sem4`?

4. En el programa de la figura 30, quin efecte tindria eliminar les dues invocacions a la crida al sistema `close` fetes pel procés pare just abans d'invocar les crides al sistema `wait`?

5. Indiqueu quins senyals hi ha bloquejats als punts d'execució A, B, C, D, E, F, G, H i I.

```
void sigusr1(int signum)
{ sigset_t mascara;
   /* C */
   sigemptyset(&mascara);
   sigaddset(&mascara, SIGINT); sigaddset(&mascara, SIGusr1);
   sigprocmask(SIG_BLOCK, &mascara, NULL);
   /* D*/
}
void sigusr2(int signum)
{ /* B */
   kill(getpid(), SIGUSR1);
}
void sigalrm(int signum)
{ /* H*/
}
main()
{ sigset_t mascara;
   struct sigaction new;
   new.sa_handler = sigusr1; new.sa_flags = 0;
   sigemptyset(&new.sa_mask);sigaddset(&new.sa_mask, SIGALRM);
   sigaction(SIGUSR1, &new, NULL);
   new.sa_handler = sigalrm; sigemptyset(&new.sa_mask);
   sigaction(SIGALRM, &new, NULL);
   new.sa_handler = sigusr2;
   sigemptyset(&new.sa_mask);sigaddset(&new.sa_mask, SIGPIPE);
   sigaction(SIGUSR2, &new, NULL);
   /* A */
   kill(getpid(), SIGUSR2);
   /* E */
   sigemptyset(&mascara); sigaddset(&mascara, SIGALRM);
   sigprocmask(SIG_BLOCK, &mascara, NULL);
   /* F */
   sigfillset(&mascara); sigdelset(&mascara, SIGALRM);
   alarm(2);
   /* G */
   sigsuspend(&mascara);
   /* I */
}
```

6. Escriviu un fragment de codi que esperi l'arribada d'un senyal de tipus SIGUSR1 o SIGUSR2, però que atengui qualsevol altre tipus de senyal que no estigui bloquejat.

7. Escriviu un fragment de codi que esperi l'arribada d'un senyal de tipus `SIGUSR1` i un altre de tipus `SIGUSR2` (en qualsevol ordre), però que atengui qualsevol altre tipus de senyal que no estigui bloquejat.

Solucionari

Exercicis d'autoavaluació

1. El codi proposat per al generador de nombres aleatoris és el següent:

```
void Productor()
{ for (;;)
 { n = crearNombre ();
  sem_wait(full); /*Esperar que pugui posar un element*/
  sem_wait(mutexProd); /*Exclusió mútua entre productors*/
  /*(i aleatoritzadors)*/
   buffer[in] = n;
  in = (in + 1) % maxbuff;
  sem_signal(mutexProd); /*Fi d'exclusió*/
  sem_signal(empty); /*Indica nou element al vector*/
  /*de memòria intermedia*/
 }
}

int Consumidor () /*Agafa un nombre qualsevol del vector*/
/*de memòria intermèdia*/
{ return buffer[out]; }

void Aleatoritzador () /*SOLUCIÓ 1.0*/
{ /*Variables locals*/
 int i,aux;
 int n[5];
 for (;;){ /*FALLA*/
  for (i = 0; i < 5; i++)
   sem_wait(empty); /*Espera que hi hagi 5 o més elements*/
   sem_wait(mutexAleat); /*Exclusió entre aleatoritzadors*/
  for (i = 0; i < 5; i++){
   n[i] = buffer [out];
   out = (out + 1) % maxbuff;
   sem_signal(full); /*FALLA*/
  } /*Fi de l'exclusió. Ha extret 5*/
  sem_signal(mutexAleat); /*Elements consecutius*/
  aux = processa (n[0], n[1], n[2], n[3], n[4]);
  sem_wait(full); /*Espera que hi hagi lloc*/
  /*Exclusió mútua entre productors (i aleatoritzadors)*/
  sem_wait(mutexProd); buffer[in] = aux;in = (in + 1) % maxbuff;
  sem_signal(mutexProd); /*Fi d'exclusió*/
  sem_signal(empty); /*Element nou disponible*/
 }
}
```

Cal observar que l'aleatoritzador pot portar problemes. Per exemple, si només hi ha un aleatoritzador, mentre està processant els últims 5 nombres que ha extret, els productors poden omplir el vector de memòria intermèdia –es quedaran tots bloquejats–, l'aleatoritzador farà un `sem_wait(full)` –també es bloquejarà– i ningú no consumirà elements.

Per a arreglar això només cal aconseguir que l'aleatoritzador desi el lloc de l'últim element que extreu: es pot substituir la línia de codi

```
sem_signal(full);/*FALLA*/
```

per

```
if (i < 4) sem_signal(full);/*NO FALLA*/
```

i eliminar el

```
sem_wait(full);/*Espera que hi hagi lloc*/
```

D'altra banda, si hi ha més d'un aleatoritzador, aquests es poden repartir els elements del vector de memòria intermèdia, sense que cap n'agafi 5 –faci 5 `sem_wait(empty)`–, el vector de memòria intermèdia es pot omplir i es quedaria tot bloquejat perquè cap aleatoritzador no trauria elements.

Per corregir aquests dos problemes proposem la solució següent per a l'aleatoritzador:

```
void Aleatoritzador () /*SOLUCIÓ 1.1*/
{ int i,aux; /*Declaració de variables locals*/
 int n[5];
 for (;;)
 { sem_wait(mutexAleat); /*Exclusió mútua entre aleatoritzadors.*/
  /*Evita que algun altre aleatoritzador pugui*/
  /*prendre-li elements*/
  for (i = 0; i < 5; i++)
  { sem_wait(empty); /*Espera un element*/
   n[i] = buffer[out];
   out = (out + 1) % maxbuff;
   if (i < 4) sem_signal(full); /*NO FALLA*/
  } /*Fi de l'exclusió. Ha extret 5*/
  sem_signal(mutexAleat); /*Elements consecutius*/
  aux = processa (n[0], n[1], n[2], n[3], n[4]);
  /*En aquest punt, SEGUR QUE TENIM LLOC PER A UN ELEMENT*/
  sem_wait(mutexProd); /*Exclusió mútua entre*/
  /*Productors (i Aleatoritzadors)*/
  buffer[in] = aux;
  in = (in + 1) % maxbuff;
  sem_signal(mutexProd); /*Fi d'exclusió*/
  sem_signal(empty); /*Element nou disponible*/
 }
}
```

2. Tindrem les variables i les estructures de dades a la memòria compartida següents (a més de les necessàries per a implementar el vector de memòria intermèdia circular):

```
/*Variables que indiquen si hi ha algun productor o algun*/
/*consumidor que actualment accedeix al vector de memòria*/
/*intermèdia. Inicialment no n'hi ha cap.*/
int cap_prod_accedint = 1;
int cap_cons_accedint = 1;

/*Vectors que indicaran el nombre de productors i de consumidors*/
/*que estan bloquejats esperant accedir al vector de memòria*/
/*intermèdia. Totes les posicions han d'estar inicialitzades a 0.*/
int prod_esperant[P];
int cons_esperant[P];

/*Vectors de semàfors on estaran bloquejats per prioritats*/
/*els consumidors i els productors. Els semàfors han d'estar*/
/*inicialitzats a 0. Tindrem 2 semàfors per a cada prioritat.*/
semaphore prod_semafors[P];
semaphore cons_semafors[P];

/*Semàfors d'exclusió mútua productors/consumidors,*/
/*inicialitzats a 1.*/
semaphore prod_mutex, cons_mutex;

/*Semàfors de bloqueig en cas de vector de memòria*/
/*intermèdia ple o buit*/
semaphore full; /*Inicialitzat a N*/
semaphore empty; /*Inicialitzat a 0*/

/*Productors*/
elem = produir();
sem_wait(prod_mutex);
if (cap_prod_accedint)
{ cap_prod_accedint = 0;
  sem_signal(prod_mutex);
}
else
{ prod_esperant[prioritat()]++;
  sem_signal(prod_mutex);
  sem_wait(prod_semafors[prioritat()]);
}
sem_wait(full);
buffer[in] = elem; /*No cal exclusió perquè només*/
in = (in + 1)% N; /*Pot accedir 1 productor*/
sem_signal(empty);
sem_wait(prod_mutex);
```

```
for (i = P - 1;( i >= 0) && (prods_esperant[i] == 0);i--);
if (i < 0)
  cap_prod_accedint = 1;
else
{ prod_esperant[i]--;
  sem_signal(prod_semafors[i]);
}
sem_signal(prod_mutex);

/*Consumidors*/
sem_wait(cons_mutex);
if (cap_cons_accedint)
{ cap_cons_accedint = 0;
  sem_signal(cons_mutex);
}
else
{ cons_esperant[prioritat()]++;
  sem_signal(cons_mutex);
  sem_wait(cons_semafors[prioritat()]);
}
sem_wait(empty);
elem = buffer[out];
out = (out + 1) % N;
sem_signal(full);
sem_wait(cons_mutex);
for (i = P - 1;(i >= 0) && (cons_esperant[i] == 0);i--);
if (i<0)
  cap_cons_accedint = 1;
else
{ cons_esperant[i]--;
  sem_signal(cons_semafors[i]);
}
sem_signal(cons_mutex);
consumir(elem);
```

3.a) Vegem el cas de cada semàfor:

- El `sem1` s'ha d'inicialitzar a 0. Aquest semàfor serveix per a sincronitzar l'orxater i el cambrer. Cada cop que l'orxater hagi acabat de preparar una orxata farà un `sem_signal` sobre `sem1`, i el cambrer la consumirà fent un `sem_wait`.
- El `sem2` s'ha d'inicialitzar a 0. Aquest semàfor sincronitza el cambrer amb el client; quan el client hagi escrit la seva petició, farà un `sem_signal` sobre aquest semàfor per indicar al cambrer que la serveixi.
- El `sem3` s'ha d'inicialitzar a 0. Aquest semàfor sincronitza el client amb el cambrer. Quan totes les orxates que ha demanat el client estiguin servides, el cambrer farà un `sem_signal` sobre `sem3` i el client es podrà continuar executant.
- El `sem4` s'ha d'inicialitzar a 1. Aquest semàfor regula un accés en exclusió mútua (garanteix que no hi pugui haver diversos clients fent peticions al cambrer simultàniament).
- El `sem5` s'ha d'inicialitzar a 1. Garanteix l'accés a la variable *quantes_orxates* en exclusió mútua.
- El `sem6` s'ha d'inicialitzar a 1 perquè també és un semàfor d'exclusió mútua. Regula l'accés a la variable *preparades*.

b) Els semàfors `sem5` i `sem6` serveixen per a implementar exclusions mútues.

El `sem5` no és necessari. Gràcies al `sem2` i al `sem4` no es pot donar el cas que en el mateix moment diversos processos accedeixin a *n*_orxates.

El semàfor `sem6` tampoc no és necessari, ja que controla una exclusió mútua sobre una variable (preparades) que no és compartida.

c) El comptador de `sem1` ens indica el nombre d'orxates preparades i pendents de ser servides.

L'orxater l'incrementa cada cop que en té una de feta, i el cambrer el decrementa quan en consumeix una.

d) El `sem4` serveix per a tenir exclusió mútua entre els clients. Fins que un no és totalment servit, els altres no poden indicar al cambrer que s'esperen.

4. Provocarà que el procés pare es bloquegi indefinidament al segon `wait` perquè estarà esperant la mort d'un dels seus fills (el que executa l'ordre *grep*), però aquest tampoc no morirà perquè estarà bloquejat llegint d'una *pipe* buida en què hi hagi un procés escriptor, el procés

pare mateix. Per tant, tenim un interbloqueig en què el pare està esperant que el fill mori i el fill està esperant que el pare tanqui el canal d'escriptura sobre la *pipe*.

5. Els senyals bloquejats a cada punt són:

- A: cap
- B: usr2, pipe
- C: usr2, pipe, usr1, alrm
- D: usr2, pipe, usr1, alrm, int
- E: buit
- F: alrm
- G: alrm
- H: tots
- I: alrm

6.
```
sigprocmask(SIG_BLOCK, NULL, &m);
sigaddset(&m, SIGUSR1); sigaddset(&m, SIGUSR2);
sigprocmask(SIG_BLOCK, &m, &old);
sigdelset(&m, SIGUSR1); sigdelset(&m, SIGUSR2);
while ((usr1rebut || usr2rebut) == FALSE)
  sigsuspend(&m);
sigprocmask(SIG_SETMASK, &old, NULL);
```

7.
```
sigprocmask(SIG_BLOCK, NULL, &m);
sigaddset(&m, SIGUSR1); sigaddset(&m, SIGUSR2);
sigprocmask(SIG_BLOCK, &m, &old);
sigdelset(&m, SIGUSR1); sigdelset(&m, SIGUSR2);
while ((usr1rebut&&usr2rebut) == FALSE)
  sigsuspend(&m);
sigprocmask(SIG_SETMASK, &old, NULL);
```

Glossari

canvi de context *m* Manera d'implementar la concurrència de processos en un sistema multiprogramat que comporta deixar d'executar el procés que estava ocupant el processador per a passar a executar un altre procés.

espera activa *f* Consulta continuada per part d'un procés de l'estat d'un recurs, generalment per a determinar si està lliure. Consumeix cicles del processador sense fer cap feina útil.

exclusió mútua *f* Accés individualitzat a un conjunt de recursos compartits, de manera que si una operació s'inicia no se'n pot iniciar una altra fins que no hagi finalitzat la primera.

execució concurrent *f* Execució entrellaçada de diversos processos.

execució paral·lela *f* Execució simultània de diversos processos. Només és possible en sistemes que disposin de més d'un processador.

inhibir interrupcions *v tr* No permetre interrupcions de cap tipus.

interbloqueig *m* Bloqueig indefinit de dos o més processos que esperen l'alliberament d'algun recurs compartit del sistema, el qual ja ha estat assignat.
en *deadlock*

pas de missatges *m* Eina que ofereixen alguns sistemes que permet sincronitzar i comunicar processos.

secció crítica *f* Conjunt d'ordres en llenguatge màquina que s'han d'executar en exclusió mútua perquè modifiquen l'estat dels recursos compartits del sistema.

semàfor *m* Mecanisme de sincronització.

senyal programari *m* Eina que proporciona el sistema operatiu amb l'objectiu de traslladar el mecanisme de les interrupcions a escala de procés. Igual que en el mecanisme de maquinari de les interrupcions, els senyals donen suport a un conjunt ampli de situacions diferents que han de ser conegudes i ateses amb una certa urgència per part dels processos.

Bibliografia

Silberschatz, A.; Galvin, P. B.; Gagne, G. (2008). *Operating Systems Concepts* (8a. ed.). John Wiley & Sons.

Tanembaum, A. (2009). *Modern Operating Systems*. Prentice Hall.

Documentació disponible sobre crides al sistema UNIX en els recursos de l'aula.

Annex

1. El suport de maquinari (*hardware*) per a l'exclusió mútua

Hem vist que la sincronització és necessària per a aconseguir que l'accés a la secció crítica en exclusió mútua es faci de manera correcta. També hem vist que els semàfors són una eina bastant fàcil d'utilitzar que soluciona el problema de la sincronització de manera elegant per a qualsevol nombre de processos. Ara bé, per a implementar els semàfors és necessari assegurar la indivisibilitat en l'execució del codi de les operacions `sem_wait` i `sem_signal`.

A continuació presentem solucions de maquinari per executar el codi de manera indivisible. Indirectament això ens permet la implementació dels semàfors.

1.1. Mode d'execució privilegiat: la inhibició de les interrupcions

En la majoria de computadors hi ha instruccions del llenguatge màquina que donen la possibilitat d'inhibir (deshabilitar) i desinhibir (habilitar) les interrupcions. Amb aquestes instruccions la implementació de l'exclusió mútua és trivial.

Hem vist que el problema de l'exclusió mútua en general és la desassignació del processador a un procés en execució quan aquest està modificant una variable compartida, ja que aleshores l'actualització d'aquesta variable queda a mig fer. El processador necessita el mecanisme de les interrupcions per a implementar la multiplexació de processos. Si un procés desactiva les interrupcions, aquest sempre tindrà assignat el processador fins que no les torni a activar.

Amb aquest funcionament es pot garantir l'exclusió mútua d'una secció crítica mitjançant l'esquema següent (figura 47).

```
inhibir interrupcions          /* desactivar les interrupcions */
secció crítica
desinhibir interrupcions       /* activar de nou les interrupcions */
```

Figura 47. Implementació de l'entrada i sortida de l'exclusió mútua inhibint i desinhibint interrupcions

Aquestes instruccions són molt perilloses si se'n fa un ús indegut o malintencionat. Podrien portar el sistema fàcilment al caos en no permetre l'execució de cap procés. Per aquest motiu interessa que aquestes instruccions no estiguin a

l'abast dels programadors d'aplicacions. El més habitual és que siguin instruccions privilegiades del llenguatge màquina i puguin ser utilitzades únicament pel sistema operatiu.

En el cas dels semàfors, el sistema operatiu pot utilitzar aquestes instruccions per a assegurar la indivisibilitat en l'execució de les crides al sistema `sem_wait` i `sem_signal`.

1.2. Mode d'execució no privilegiat

A continuació es presenten dues intruccions de llenguatge màquina no privilegiades que poden ser utilitzades per a implementar l'accés en exclusió mútua a una regió crítica. Per tant, aquestes intruccions poden ser utilitzades directament pels programes d'usuari.

1.2.1. *Test and set*

La instrucció *test and set* està pensada per a donar un suport de maquinari (*hardware*) al problema de l'accés a la zona crítica en exclusió mútua. Està dissenyada expressament per a resoldre conflictes d'accés a zones crítiques i assegurar que tan sols un procés pugui accedir a la vegada a la secció crítica.

La idea bàsica és definir una variable de control (global per a tots els processos) associada a la secció crítica[21] que determini si l'accés és possible o no. La variable s'inicialitza a `lliure`, i indica que el recurs compartit està disponible. Cada procés que vol accedir al recurs executa la instrucció *test and set* (`TS`) i li passa per paràmetre la variable de control associada al recurs.

(21) Recordeu que la secció crítica és un tros de codi associat a la utilització d'un recurs compartit.

Aquesta operació s'escriu `TS variable` i funciona comparant el valor de la variable amb `ocupat`. Si és a `lliure` el modifica a `ocupat`. Els bits de condició (*flags* d'estat del processador) es modifiquen per a indicar l'estat de la variable just abans d'executar l'operació `TS`.

Un semàfor binari que utilitzi espera activa podria implementar l'operació utilitzant `TS` de la manera següent (figura 48 i figura 49):

```
sem_wait: TS S
   BNF sem_wait
   return
```

Figura 48. Implementació de `sem_wait` amb *test_and_set*

```
sem_signal: mov S, lliure
```

Figura 49. Implementació de `sem_signal` amb *test_and_set*

Suposem que inicialment `S` té el valor `lliure`. Quan el primer procés executa sem_wait passa el següent:

1) La instrucció `TS S` mira l'estat de `S`, i com que `S` és a `lliure` el passa a `ocupat`. La instrucció `TS` modifica els indicadors per indicar que `S` estava `lliure`.

2) La instrucció `BNF` (*branch if not free*) salta a l'etiqueta `sem_wait` si el valor de `S` era a `ocupat`. En el nostre cas, com que és a `lliure` no hi salta, el procés retorna de la crida `sem_wait` i, per tant, se li assigna el recurs compartit.

Si altres processos volen accedir al recurs, quan executin l'operació `sem_wait` passarà el següent:

1) La instrucció `TS S` mira l'estat de `S`, i com que `S` és a `ocupat` no es modifica la variable. La instrucció `TS` modifica els indicadors per indicar que `S` estava `ocupat`.

2) La instrucció `BNF` salta a l'etiqueta `sem_wait`, ja que el valor de `S` era a `ocupat`. D'aquesta manera tots els processos es queden executant el bucle `sem_wait`, esperant que algú alliberi el recurs compartit. Per a alliberar el recurs compartit l'únic que s'ha de fer és posar a `lliure` el valor de `S`. El codi de l'operació `sem_signal` que ho pot fer és el que mostrem a la figura 49.

És important assenyalar que el codi de la instrucció `sem_wait` funciona perquè la instrucció `TS` és indivisible, o perquè mentre el maquinari executa el microcodi associat a aquesta instrucció les interrupcions estan inhibides.

1.2.2. L'intercanvi (*swap*)

La instrucció *swap* intercanvia el contingut de dues paraules de memòria atòmicament, i es defineix tal com s'indica a la figura 50.

```
swap (A,B){
   temp = A;
   A = B;
   B = temp;
}
```

Figura 50. Definició de la instrucció de llenguatge màquina *swap*

Si el llenguatge màquina ofereix una instrucció d'intercanvi (*swap*), llavors l'exclusió mútua es pot aconseguir de la manera següent: definim una variable global que anomenem `lock`, la qual pot prendre els valors `cert` o `fals`. La variable s'inicialitza a `fals`. A més a més, cada procés té una variable local, anomenada `clau`, que pot prendre els valors `cert` o `fals`.

Tot procés que vol accedir a una zona crítica en exclusió mútua ha d'executar el codi de la figura 51.

```
/* entrada secció crítica */
clau = cert;
do {
   swap (lock, clau);
} until (clau == fals);
/* secció crítica */
...
/* sortida secció crítica */
lock = fals;
```

Figura 51. Implementació de l'entrada i la sortida de la secció crítica amb la instrucció

2. Exemple: processos productors i consumidors

En un sistema és habitual trobar processos amb relacions de productor-consumidor. En general un **procés productor** genera informació que serà consumida (tractada/manipulada) pel **procés consumidor**.

El problema que es vol resoldre es podria formular en els termes següents: considerem un grup de processos consumidors i productors que s'estan executant concurrentment i que poden produir i consumir a diferents velocitats. Volem proposar un protocol de sincronització que permeti executar els processos productors i consumidors concurrentment a les seves velocitats respectives i que les dades es consumeixin en el mateix ordre en què s'han generat.

Totes les versions proposades per a aconseguir-ho estan basades en semàfors. En concret, es presenten les tres versions que s'indiquen a continuació:

1) Un productor i un consumidor amb vector de memòria intermèdia (*buffer*) il·limitat.
2) Diversos productors i diversos consumidors amb vectors de memòria intermèdia il·limitats.
3) Diversos productors i diversos consumidors amb vectors de memòria intermèdia limitats.

Per permetre als processos productors i consumidors treballar concurrentment, suposarem que disposem de vectors de memòria intermèdia, espais de memòria que els productors omplen amb les dades generades i d'on els consumidors obtenen les dades que han de ser tractades.

2.1. Un productor i un consumidor amb un vector de memòria intermèdia il·limitat

En una primera versió considerarem que la mida del vector de memòria intermèdia és il·limitada i que tenim un únic productor i un únic consumidor. Un cop inicialitzat el sistema, el primer procés que s'ha de poder executar és el productor.

Quan ja s'han produït dades el consumidor pot començar a tractar la informació. En el codi que es proposa a continuació s'utilitza un únic semàfor, `produit`, inicialitzat a 0. Això vol dir que no és possible entrar a la secció crítica o, dit d'una altra manera, si un procés executa l'operació `sem_wait` sobre el semàfor produït es bloquejarà sempre que cap altre procés no hagi executat l'operació `sem_signal` sobre el semàfor produït abans.

En aquest primer cas el **codi proposat per al procés productor** és el de la figura 52 i el **codi proposat per al procés consumidor** és el de la figura 53.

```
while (cert){
   /* Produir */
   ...
   /* Deixar al buffer */
   ...
   sem_signal(produit);
   /* Altres operacions */
   ...
}
```

Figura 52. Codi dels productors (versió 1)

```
while (cert){
   sem_wait(produit);
   /* Llegir del buffer */
   ...
   /* Consumir */
   ...
   /* Altres operacions */
   ...
}
```

Figura 53. Codi dels consumidors (versió 1)

El semàfor `produit` porta el compte del nombre de dades produïdes i encara no consumides. De fet, tal com s'ha especificat, el procés productor no necessita demanar permís per a accedir a la variable compartida (vector de memòria intermèdia) i l'únic que ha de fer és assenyalar que ha deixat un element nou al vector de memòria intermèdia. El productor ho senyalitza amb l'operació `sem_signal`.

El procés consumidor, d' altra banda, a fi que el funcionament sigui correcte s'ha d'assegurar que hi ha dades al vector de memòria intermèdia abans d'intentar accedir-hi. La manera que té de saber-ho és executar l'operació `sem_wait`.

Suposem que el consumidor no té assignat el processador i, per tant, no s'executa durant un cert temps. Mentre el productor va generant dades i ficant-les al vector de memòria intermèdia, perquè el consumidor sàpiga quantes dades ha de processar un cop es torni a executar cada `sem_signal`, ha de quedar reflectit en el semàfor `produit`. És, doncs, absolutament necessari utilitzar un semàfor *n*-ari[22] per a tenir constància del nombre de senyals i, per tant, del nombre de dades pendents de processar que hi ha al vector de memòria intermèdia.

[22] El fet de tenir més d'una dada fa inviable l'ús d'un semàfor binari.

El procés consumidor, un cop té assignat el processador, podrà entrar tants cops a la secció crítica com dades hi hagi al vector de memòria intermèdia. És a dir, l'operació `sem_wait` no el bloquejarà fins que el vector de memòria intermèdia es buidi. Aquesta primera solució és molt senzilla; de fet, no hi apareix cap secció crítica.

El vector de memòria intermèdia es pot considerar un vector infinit amb els dos punters següents:

- Un punter anomenat *ent* que indica quina és la posició lliure següent del vector de memòria intermèdia.

- Un altre punter, anomenat *sort*, que indica quina és la primera posició del vector de memòria intermèdia que té dades vàlides pendents de ser tractades.

Inicialment tots dos punters són inicialitzats a 0. El productor tan sols modifica el punter `ent` en executar la funció *deixar_al_buffer*, i el consumidor modifica el punter `sort` en executar la funció *llegir_del_buffer*, de manera que no hi ha cap tipus de conflicte.

2.2. Diversos productors i diversos consumidors amb un vector de memòria intermèdia il·limitat

En el cas de considerar la possibilitat de tenir més d'un productor i més d'un consumidor la cosa canvia una mica i no podem utilitzar el codi que s'ha mostrat abans. Tots els productors en executar la funció *deixar_al_buffer* modifiquen una variable compartida: el punter `ent` i tots els consumidors en executar la funció *llegir_del_buffer* modifiquen una variable compartida, el punter `sort`.

A continuació proposem el codi necessari per a garantir l'execució concurrent en el cas de tenir *N* processos productors i *M* processos consumidors.

En aquest segon cas el **codi proposat per als processos productors** és el de la figura 54 i el **codi proposat per als processos consumidors** és el de la figura 55.

```
while (cert){
   /* Produir */
   ...
   sem_wait(exclusio);
   /* Deixar al buffer */
   ...
   sem_signal(exclusio);
   sem_signal(produit);
   /* Altres operacions */
   ...
}
```

Figura 54. Codi dels productors (versió 2)

```
while (cert){
   sem_wait(produit);
   sem_wait(exclusio);
   /* Llegir del buffer */
   ...
   sem_signal(exclusio);
   /* Consumir */
   ...
   /* Altres operacions */
   ...
}
```

Figura 55. Codi dels consumidors (versió 2)

El problema que planteja la gestió de tants processos se soluciona amb un nou semàfor, anomenat `exclusio`[23] i inicialitzat a 1, perquè el primer procés que intenti accedir a la secció crítica tingui la possibilitat de fer-ho. Com en el cas d'abans, un consumidor no intentarà entrar a la zona crítica fins que algun productor no hagi generat dades. En aquesta solució també seria possible utilitzar dos semàfors diferents per a assegurar l'exclusió mútua en l'accés dels processos al vector de memòria intermèdia, un per als consumidors i un altre per als productors. Amb dos semàfors s'aconseguiria més concurrència entre productors i consumidors.

[23] El semàfor *exclusio* és binari. El semàfor produit és *n*-ari.

2.3. Diversos productors i diversos consumidors amb un vector de memòria intermèdia limitat

En la tercera versió considerem que tenim un nombre de productors i consumidors il·limitat i, a més a més, que la mida del vector de memòria intermèdia és limitada, té una capacitat màxima. En aquest cas el vector de memòria intermèdia és un recurs compartit per tots els processos, i el definim com un vector, `buffer[0..capacitat - 1]`, de capacitat finita i igual a `capacitat`, més dos punters, `ent` i `sort`, inicialitzats a 0. Els punters indiquen quina és la primera posició lliure del vector de memòria intermèdia (`ent`) i quina és la primera posició d'aquest vector que conté informació útil (`sort`). En aquest tercer cas el **codi proposat per als processos productors** és el de la figura 56 i el **codi proposat per als processos consumidors** és el de la figura 57.

```
while (cert){
   sem_wait(potproduir);
   /* Produir */
   pdata = produir();
   sem_wait(p_exclusio);
   /* Deixar al buffer */
   buffer[ent] = pdata;
   ent = (ent + 1) mod capacitat;
   sem_signal(p_exclusio);
   sem_signal(potconsumir);
   /*Altres operacions*/
   ...
}
```

Figura 56. Codi dels productors (versió 3)

```
while(cert){
   sem_wait(potconsumir);
   sem_wait(c_exclusio);
   /*Llegir del buffer */
   cdata = buffer[sort];
   sort = (sort + 1) mod capacitat;
   sem_signal(c_exclusio);
   sem_signal(potproduir);
   /*Consumir*/
   ...
   /*Altres operacions*/
   ...
}
```

Figura 57. Codi dels consumidors (versió 3)

En la solució proposada, s'han definit quatre semàfors:

- `c_exclusio` i `p_exclusio`, que són semàfors binaris inicialitzats a 1. Garanteixen l'exclusió mútua entre els productors que accedeixen a la variable `ent` i entre els consumidors que accedeixen a la variable `sort`.

- `potproduir`, que és un semàfor *n*-ari i s'inicialitza a la capacitat del vector de memòria intermèdia, de manera que estem permetent omplir el vector de memòria intermèdia. Si aquesta s'omple, el productor següent que intenti ficar-hi alguna cosa més es quedarà bloquejat fins que algun consumidor consumeixi dades, deixi espai lliure al vector de memòria intermèdia i executi `sem_signal(potproduir)`.

- `potconsumir`, que és un semàfor *n*-ari inicialitzat a 0, ja que en un principi el vector de memòria intermèdia és buit i no hi ha res per a consumir. Els processos productors cada cop que deixen alguna cosa al vector de memòria intermèdia ho indiquen amb `sem_signal(potconsumir)`. El valor del semàfor `potconsumir` serà com a màxim el valor de capacitat, el nombre de senyals (*signals*) a aquest semàfor que es poden executar abans que el vector de memòria intermèdia s'ompli.