# PROBLEMES TEMA 6: Compartició de recursos

NOTA: En aquests problemes utilitzarem per fer **mutex** les crides de pthreads (podeu utilitzar el man de linux) :

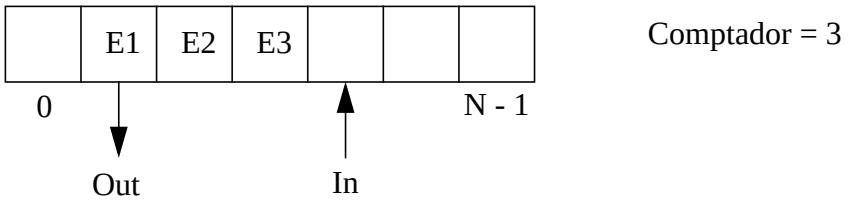
- int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr); /* Posarem attr=NULL */
- int pthread_mutex_lock(pthread_mutex_t *mutex);
- int pthread_mutex_unlock(pthread_mutex_t *mutex);

Com a crides de **semàfors** utilitzarem (podeu consultar el man):

- int sem_init(sem_t *sem, int pshared, unsigned int value); /* possarem pshared=0 */
- int sem_wait(sem_t *sem);
- int sem_post(sem_t *sem); /* Equivalent al sem_signal vist a classe*/

## Problema 6.1 Productors/Consumidors

Tenim un buffer circular de tamany N que conté elements d'un cert tipus i que permet comunicar diferents fluxes. Uns d'ells - productors - escriuen elements dins el buffer mentre no sigui ple. Altres - els consumidors - treuen elements del buffer mentre no sigui buit. És possible tenir un comptador que indiqui el nombre d'elements que hi ha dins el buffer.




A) Implementeu el programa amb un productor i un consumidor i detecteu possibles problemes d'exclusió mutua. Quins són els objectes a protegir?

B) Implementeu una solució fent servir el comptador.

C)Implementeu una solució amb semàfors.

D) Implementeu una situació general amb *m* productors i *n* consumidors. Quants semàfors són necessaris? Tenen tots la mateixa funció?

- Objectiu: Veure els problemes de concurréncia que hi ha en accedir a estructures de dades compartides

## Problema 6.2 Semàfors amb mutex (I)

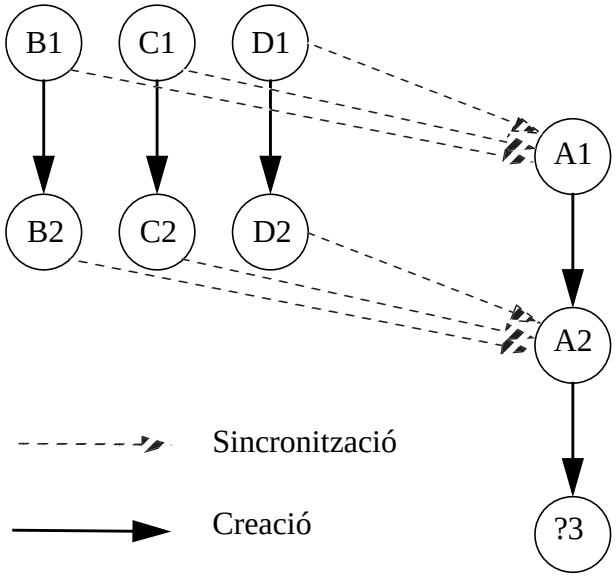
Implementeu un semàfor amb mutex. Recordeu que el codi de semàfors accedeix a la variable que utilitzem al semàfor i que és compartida, per tant hem d'accedir en exclusió mutua. Supposseu el codi de semàfors vist a classe.

## Problema 6.3 Rendez-vous

Una doble sincronització entre dos fluxes, és a dir, que cadascú espera per l'altre, es coneix com a rendez-vous. Proposeu una implementació mitjançant semàfors. Tenir en compte que qualsevol dels dos fluxes pot bloquejar-se en primer lloc.

## Problema 6.4 Planificació familiar

Tenim el següent graf de fluxes on les línies sòlides representen creació de fluxes i les línies discontínues indiquen sincronització.




El flux A1 esperarà que els fluxes B1, C1 i D1 hagin creat els fluxes B2, C2, i D2, respectivament. Després, el flux A1 crearà el flux A2. Aquest últim, crearà el flux B3, C3, o D3 si el que acaba primer és B2, C2, o D2, respectivament.

Es demana que resoleu els problemes de concurrència que puguin afectar a aquests fluxes fent servir semàfors.

## Problema 6.5 Crides Atómiques

Mostreu, amb un exemple, que si les operacions de `sem_wait()` i `sem_signal()(o sem_post)` no s'executen atòmicament no es garanteix l'exclusió mútua.

## Problema 6.6 sem_ask

En un sistema operatiu s'ha afegit la primitiva

```
int sem_ask (n_sem);
```

aquesta crida detecta el que passaria si es fes un `sem_wait` sobre el semàfor indicat; en cas de detectar que no es quedaria bloquejat fa el `sem_wait` i ens retorna `cert`, en cas contrari ens retorna `fals`.

1. Raoneu el bon o mal funcionament d'aquesta primitiva per a evitar *deadlocks.*
2. Doneu un esquema d'utilització d'aquesta primitiva a fi d'evitar el *deadlock* en el cas típic:

```
Flux A                Flux B
  sem_wait (sem_1);     sem_wait (sem_2);
  ...                   ...
  sem_wait(sem_2);      sem_wait(sem_1);
```

## Problema 6.7 Múltiple wait/Múltiple signal

Un sistema operatiu ofereix un mecanisme d'exclusió mútua sobre dos recursos a l'hora. Per a fer-ho es basa en les següents primitives:

```
multiple_wait (int recurs_1, int recurs_2);
```

que comprova que les exclusions mútues representades pels dos arguments estiguin lliures. Si ho estan, les dóna les dues a l'hora. Sinó, el flux espera fins que s'alliberin totes dues.

```
multiple_signal (int recurs_1, int recurs_2);
```

que allibera els dos recursos a l'hora.

Es demana que implementeu aquestes dues rutines amb semàfors.

## Problema 6.8 La barberia es reforma

Una barberia té una sala d'espera amb N cadires, i una sala amb la cadira de barber. Si no hi ha clients per atendre, el barber es fa la migdiada. Si entra un client a la barberia, i totes les cadires estan ocupades, marxa de la barberia. Si el barber està ocupat, però hi ha cadires lliures, llavors s'espera a una de les cadires. Si el barber està adormit, el client desperta al barber. Implementarem les rutines del barber i els clients tot fent servir semàfors.

Et donem l'estructura del programa. Cal definir els semàfors necessaris, fer la seva inicialització, i incloure les crides necessaries per a evitar problemes d'exclussió mutua. Cal seguir l'esquema proposat:

```
int seients_lliures;   /*Variable numero de seients lliures a la sala d'espera */
int barber_adormit;    /*Indica si el barber esta adormit o no */
sem_t...;          /* Aqui cal declarar els semafors que es considerin necessaris
*/

/*Rutina per inicialitzar els semafors i les variables necessaries */
void inicialitzacio ()
{...}

void barber ()
{
  while (1)
    {
      /* Mirar si hi ha clients a la sala d'espera */
      if (seients_lliures == N)
        {
          barber_adormit = TRUE;
          /* Aqui cal esperar que arribi un client */
        }
      /* Ara un client de la sala d'espera pasa a la cadira del barber */
      seients_lliures++;
      /*Ara ja tenim un client a la cadira del barber, cal treballar */
      afeitar;
    }
}

void client ()
{
  /* Arriba un nou client a la barberia */
  if (seients_lliures != 0)
    {
      if (barber_adormit)
        {
          /* Cal despertar-lo */
          barber_adormit = FALSE;
        }
      seients_lliures--;
      /* Esperem el nostre torn a la sala d'espera, fins que el barber ens avisi*/
      /* Ara passem a la cadira del barber */
      afeitat;
    }
  /* el client surt de la barberia */
}
```

## Problema 6.9 Instrucciones en orden

Queremos asegurar que la ejecución de una instrucción de un proceso (o flujo) se produce siempre después de la ejecución de otra en otro proceso (o flujo). Para ello, utilizaremos diferentes mecanismos, pero el objetivo será siempre que la instrucción INS2 en el proceso/flujo PF2 se ejecute después de la instrucción INS1 del proceso/flujo PF1. Los mecanismos a utilizar serán cuatro: a) Semáforos, b) Una pipe sin nombre, c) Una pipe con nombre y d) Signals (excepciones UNIX)

Apartado 1

Antes de resolver el problema, dinos, en general, para cada uno de los mecanismos anteriores, cuándo es lo más adecuado aplicarlo para el caso de dos procesos, para el caso de dos flujos, y si podría ser utilizado para ambos casos.

Apartado 2

Programa ahora los cuatro casos a)-d): muestra el código de inicialización y la parte anterior y/o posterior a las instrucciones INS1 e INS2 utilizando para cada caso únicamente el mecanismo pedido; presenta, para cada caso, la parte de inicialización y en dos columnas los procesos o flujos, de la forma:

| Inicializacion: | Procesos/flujos: | |
|---|---|---|
| ..... | PF1 | PF2 |
| ..... | ..... | ..... |
| | INS1 | INS2 |
| | ..... | ..... |