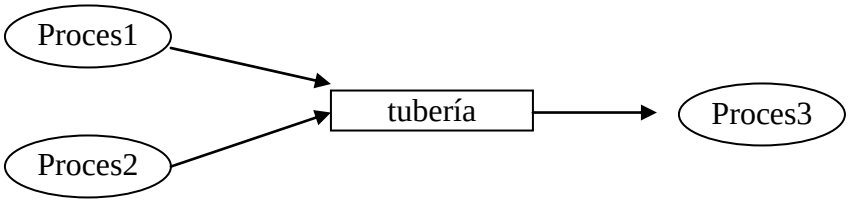
Sistemas Operativos UCM 2011/2012

Soluciones Módulo 3.2 Comunicación y Sincronización

1. Escriba un programa que cree tres procesos que se conecten entre ellos …



a. Usando tuberías

```
#define PROD 1000
int fds[2] , fdPar, fdImpar;

main(){
  int i, dato;
  char c;

  pipe(fdPar);  pipe(fdImpar);
  //Damos el turno a Par()
  write(fdPar[1], &c, sizeof(char));

  pipe(fds)
  if(!fork()){
    //Cerrar fds[0],fdImpar[0],fdPar[1]
    Pares();
  }
  if(!fork()){
    //Cerrar fds[0],fdImpar[1],fdPar[0]
    Impares();
  }

  //Cerrar fdPar[0-1],fdImpar[0-1],fds[1]

  for(i=0; i<PROD*2; i++){
    read(fds[0], &dato, sizeof(ind));
    print("Dato: %d", dato);

  //Esperamos a los hijos
  wait(NULL);wait(NULL);
}
```

```
Pares(){
  char c;
  int i;

  for(i=0; i<PROD*2; i+=2){
    read(fdPar[0], &c, sizeof(char));

    write(fds[1], &i, sizeof(ind));

    write(fdImpar[1], &c, sizeof(char));
  }
  exit(0);  //Cierra el resto de FD's
}


Impares(){
  char c;
  int i;

  for(i=0; i<PROD*2; i+=2){
    read(fdImpar[0], &c, sizeof(char));

    write(fds[1], &i, sizeof(ind));

    write(fdPar[1], &c, sizeof(char));
  }
  exit(0);  //Cierra el resto de FD's
}
```

b. Usando semáforos y memoria compartida

```
#define PROD 1000

main(){
  int fds[2];
  sem_t * sems;
  sem_t * semPar;
  sem_t * semImpar;

  sems = mmap(NULL, 2*sizeof(sem_t),
              PROT_READ|PROT_WRITE,
              MAP_SHARED|MAP_ANON,-1,0);
  semPar=sems [0];  semImpar=sems [1];
  //Damos el turno a Par()
  sem_init(semPar,   1, 1);
  sem_init(semImpar, 1, 0);

  pipe(fds)
  if(!fork()){
    //Cerrar fds[0];
    Pares(fdIn, semPar, semImpar);
  }
  else if(!fork()){
    //Cerrar fds[0];
    Impares(fdIn, semPar, semImpar);
  }
  else{
    //Cerrar fds[1];
    Consumidor(fds[0]);
    sem_destroy(semPar);
    sem_destroy(semImpar);
  }

  exit(0);  //Cierra FDs y hace unmap
}
```

```
Pares(fdIn, semPar, semImpar){
  int i;

  for(i=0; i<PROD*2; i+=2){
    sem_wait(&semPar);

    write(fdIn, &i, sizeof(int));

    sem_post(&semImpar);
  }
  exit(0); //Cierra FDs y hace unmap
}

Impares(fdIn, semPar, semImpar){
  int i;

  for(i=0; i<PROD*2; i+=2){
    sem_wait(&semImpar);

    write(fdIn, &i, sizeof(int));

    sem_post(&semPar);
  }
  exit(0);  //Cierra FDs y hace unmap
}

Consumidor(fdOut){
  int i, dato;

  for(i=0; i<PROD*2; i++){
    read(fdOut, &dato, sizeof(int));
    print("Dato: %d", dato);
  }
}
```

2. Implemente las operaciones de semáforos POSIX (`init`, `wait` y `signal`) utilizando tuberías.

```
semaphore CreateSem(int count){
   int *fd, i;

   /*Space for two file-descriptors*/
   fd = malloc(2*sizeof(int));
   pipe(fd);

   for(i=0; i<count; i++)
      write(fd[1], "x", sizeof(char));
   return(fd);
}
```

```
void Post(semaphore s){
   char c;
   read(s[0], &c, sizeof(char));
}
void Wait(semaphore s){
   char c;
   write(s[1], &c, sizeof(char));
}
int DelSem(semaphore s){
   close(s[0]);
   close(s[1]);
   free(s);
}
```

3. El algoritmo de la panadería de Lamport es un algoritmo …

Supongamos los tickets de la siguiente manera:

Id	1	2	3	4	5	6	7	8	9
Número	1	5	0	3	2	4	5	3	0
Eligiendo	false	false	false	false	false	false	true	false	true

- Si elegimos ejecutar Id1, hay que esperar (1er bucle while) a que elijan Id7 e Id9. Una vez que esto haya ocurrido, tendrán un número entero asignado (ojo, este valor puede ser 1-6). El segundo bucle esperará si encuentra algún hilo con ticket y que sea menor de 1 o igual y con Id menor. Esto no ocurre.
- Id2 esperará a Id7 e Id9 en el primer while y a todos los que tienen ticket menor de 5 o igual a 5 y con Id menor. Eespera a Id1, (no a si mismo, no a Id3), a Id4 (Id5-6 ya habrán terminado, si Id7 recibe un 5 no por que 2<7), a Id8 y dependerá a Id9.
- Id3 no tiene ticket
- Id4 …

4. Utilizando la instrucción **XCHG(a,b)** de intercambio de dos variables en …

a) Es segura, sin interbloqueo, y con posible aplazamiento indefinido.
Basta para ello advertir que inicialmente los valores de las variables **c** (global), **d** de p1 (local) y **e** de p2 (local) que intervienen en la negociación del acceso a la sección crítica forman la tripleta (1,0,0) y que las únicas operaciones que se efectúan son intercambios indivisibles sin modificación de valores. Por tanto los valores que se pueden alcanzar son:

(1,0,0) ==> p1 y p2 están fuera de sus Secciones Críticas
(0,1,0) ==> p1 está en su Sección Crítica, y p2 está fuera de la suya
(0,0,1) ==> p1 está fuera de su Sección Crítica y p2 dentro de la suya

Luego la exclusión mutua es segura.
Para producirse interbloqueo deberían alcanzarse los valores (0,0,0): ambos procesos fuera y sin posibilidad de entrar, lo cual no es posible.
El aplazamiento indefinido es posible puesto que no hay ningún mecanismo que impida que los valores estén oscilando entre (1,0,0) y (0,1,0), por ejemplo, sin dar opción a p2 a pasar a su sección crítica.
b) La generalización a n procesos es inmediata. Sólo hay que repetir el esquema de cualquiera de los procesos p1 o p2 el número de veces que haga falta, para negociar la exclusión mutua con un (n+1)-tuple de valores de los cuales uno vale 1 y el resto 0.
c) Si **XCHG** no fuera indivisible ya no se aseguraría la existencia invariante de uno y sólo un 1, y ello podría ocasionar fallos de exclusión mutua (ej. (0,1,1)) e interbloqueos (ej.(0,0,0)).

5. Codificar el problema de los lectores/escritores de forma que sean los escritores …

```
semáforo mtxLect, mtxEscr;        // inicializados a 1
semáforo lectores, escritores;    // inicializados a 1
integer n_lectores, n_escritores; // #lect/escr en SC=0
```

```
//Múltiples instancias
Lector() {
  while(1)
    wait(lectores);
    wait(mtxLect);
    n_lectores++;
    if(n_lectores==1)
      wait(escritores)
    post(mtxLect)
    post(lectores);

    ----- Leer -----

    wait(mtxLect);
    n_lectores--;
    if(n_lectores==0)
      post(escritores);
    post(mtxLect);
```

```
//Múltiples instancias
Escritor() {
  while(1){
    wait(mtxEscr);
    n_escritores++;
      if(n_escritores==1)
        wait(lectores);
    post(mtxEscr);
    wait(escritores);

    ----- Escribir -----

    post(escritores);
    wait(mtxEscr);
    n_escritores--;
    if (n_escritores==0)
      post(lectores);
    post(mtxEscr);
```

```
   }
}
```

```
   }
}
```

mtxLect, mtxEscr se utilizan únicamente para asegurar acceso exclusivo a **n_lectores** y **n_escritores**.
El semáforo **lectores** lo usan los escritores para denegar el acceso a los lectores. Cuando un escritor quiere acceder a la sección crítica, si es el primero en intentarlo, deniega futuros accesos a los lectores haciendo un **wait(lectores)**. Hasta que no haya ningún escritor intentando acceder a la sección crítica (**n_escritores=0**) no se hace un **post(lectores).**
Si un lector intenta acceder a la sección crítica y es capaz de hacerse con el semáforo **lectores**, es porque no hay escritores. Si hay un número arbitrario de lectores accediendo a la región compartida y un escritor solicita acceso, éste bloquea a futuros lectores (como se ha indicado anteriormente) y sólo tiene que esperar a que todos los lectores que están accediendo a la sección crítica terminen.

6. En el problema de los lectores/escritores, si se prioriza a un colectivo, …

```
semáforo mtxOrder, mtxAccess;     // inicializados a 1
semáforo mtxCount;                // inicializado  a 1
integer n_lectores;               // #lect en SC=0
```

```
//Múltiples instancias
Lector(){
  while(1)
    wait(mtxOrder);
    wait(mtxCount);
    contador++;
    if(n_lectores==1)
      wait(mtxAccess);
    post(mtxCount)
    post(mtxOrder);

    ----- Leer -----

    wait(mtxCount);
    n_lectores--;
    if(n_lectores==0)
      post(mtxAccess);
    post(mtxCount);
  }
}
```

```
//Múltiples instancias
Escritor(){
  while(1){
    wait(mtxOrder);
    wait(mtxAccess);
    post(mtxOrder);

    ----- Escribir -----

    post(mtxAccess);
  }
}
```

NOTA: Esta solución sólo asegura la ausencia de inanición sí y sólo sí los semáforos emplean la política FIFO cuando se producen bloqueos y desbloqueos

7. Codificar el problema del Barbero Dormilón usando mutex y semáforos.

```
sem_t  sillon, afeitado;   //Inicializados a 0
mutex_t mutex;
int sillas=0;
boolean esperando=FALSE;
const int  MAX = 5;
```

```
//Una instancia
Barbero (){
  while (TRUE) {
    lock(mutex);
    if (sillas == 0) {
      esperando = TRUE;
      unlock(mutex);
      wait(sillon);
      lock(mutex);
      esperando = FALSE
    }
    sillas = sillas-1;
    unlock(mutex);
    post(afeitado);
    cortarPelo();
  }
}
```

```
//Múltiples instancias
Cliente(){
  lock(mutex);
  if (sillas < MAX) {
    sillas = sillas+1;
    if (esperando)
      post(sillon);
    unlock(mutex);
    wait(afeitado);
    recibirCortePelo();
  } else
      unlock(mutex);
}
```

Con esta solución, ¿es posible que un cliente que solicite el corte de pelo a través de la cola `Clientes` después que otro reciba su corte de pelo antes? Sí, el primero que haga `wait(afeitado)` es el que primero se pone en cola de verdad si los semáforos se comportan de manera FIFO, primero en bloquearse, primero en desbloquearse.

8. Un monitor lo podemos entender como un objeto cuyos métodos son ejecutados …

Solución Silberschatz:

```
//Interfaz pública del monitor:
cogerPalillosMonitor(int i);
dejarPalillosMonitor(i);

//Variables privadas
int estado[N] {PENSANDO, PENSANDO, ..., PENSANDO};
condición espera[N];
```

```
cogerPalillosMonitor(int i){
  lock(mutex);
  estado[i]=HAMBRIENTO;
  test(i);
  if(estado[i]!= EATING)
    cond_wait(espera[i], mutex);
  unlock(mutex);
}
```

```
dejarPalillosMonitor(i){
  lock(mutex);
  estado[i]=PENSANDO;
  test((i-1)%N);
  test((i+1)%N);
  unlock(mutex);
}
```

```
test(int i){
```

```
        if( estado[(i-1)%N] != COMIENDO   &&
            estado[i]       == HAMBRIENTO &&
            estado[(i+1)%N] != COMIENDO   ){
          estado[i]=COMIENDO;
          cond_signal(espera[i]);
        }
      }
```

9. Considerar las siguientes primitivas de sincronización …

```
int prod=0, cons=0;
elementos deposito[N];
```

```
//Una instancia
void Productor(){
  while(TRUE) {
    AWAIT(cons, prod - N);
    deposito[prod % N] = producir();
    ADVANCE(prod);
  }
}
```

```
//Una instancia
void Consumidor(){
  while(TRUE) {
    AWAIT(prod, cons);
    consumir(deposito[cons % N]);
    ADVANCE(cons);
  }
}
```

10. El Problema de los Fumadores [Suhas Patil]: …

```
// Program Fumadores;
int ingr[3]= {FALSE,FALSE,FALSE};  // tabaco, cerillas, papel
mutex_t ingr_mutex;
cond_t ingr_cond;
```

```
Agente() {
  int i, j;

  while (TRUE) {
    i = random()%3;
    do{j=random()%3;} while (j == i);
    lock (ingr_mutex)
    while (tabaco || cerillas || papel)
      wait(ingr_cond, ingr_mutex)

    // tabaco?
    ingr[0] = (i==0) || (j==0);
    // cerillas?
    ingr[1] = (i==1) || (j==1);
    // papel?
    ingr[2] = (i==2) || (j==2);

    broadcast(ingr_cond1);
    unlock (ingr_mutex)
  }
}
```

```
void FumadorTabacoCerillas() {

  while (TRUE) {
    lock(ingr_mutex);
    //tabaco && cerillas
    while(!ingr[0] || !ingr[1])
      wait(ingr_cond, ingr_mutex);
    // tabaco
    ingr[0] = FALSE;
    // cerillas
    ingr[1] = FALSE;
    signal(ingr_cond);
    unlock(ingr_mutex);
    fumar();
  }
}
```

11. Una tribu de salvajes se sirven comida de un caldero …

a) Semáforos

```
servings = 0
mutex    = Semaphore(1)

emptyPot = Semaphore(0)
fullPot = Semaphore(0)
```

```
Savaje{
  while (True){
    mutex.wait()
    if (servings == 0) {
      emptyPot.signal()
      fullPot.wait()
      servings = M
    }
    servings -= 1
    getServingFromPot()
    mutex.signal()

    eat()
  }
}
```

```
Cocinero(){
  while True{
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()
  }
}
```

b) Cerrojos

```
servings = 0

lock_t mutex;
var_cond  emptyPot;
var_cond  fullPot;
```

```
Savaje{
  while (True){
    lock(mutex);
    while (servings == 0) {
      cond_signal(emptyPot);
      cond_wait(fullPot,mutex);
    }
    servings -= 1
    getServingFromPot()
    mutex.signal()

    eat()
  }
}
```

```
Cocinero(){
  while True{
    lock(mutex);
    while (servings>0) {
      cond_wait(emptyPot,mutex);
      servings=M;
    }
    cond_broadcast(fullPot);
    unlock(mutex);
  }
}
```

Esta codificación no es del todo estilosa porque:

- Cuando esté vacía la olla, llegará un salvaje y hará un "signal" al cocinero. Luego, se duerme dejando el mutex abierto. Luego PUEDE llegar otro(s) salvaje (antes de que el cocinero despierte) y volverá a hacer otro signal y quedará en el wait. Finalmente, el cocinero se despertará, servirá M raciones, despertará a todos (eso está bien) y se volverá a dormir….. salvo que "cond_signal" tenga un efecto "acumulativo", pero no lo tiene: si se hace un signal a una variable cond. en la que nadie espera, no hay efecto
- La diferencia con semáforos es que aquí no se garantiza el orden de los salvajes: podría ocurrir (en función de la planificación que se haga tras el broadcast) que el salvaje que avisó al cocinero no sea el primero en coger su ración (o cualquier variación en el orden entre los salvajes que hayan

llegado a la cola tras ese aviso).

12. El problema de la montaña rusa [Andrews's Concurrent Programming]: …

```
mutex1 = Semaphore(1)        //Protección para boarders
mutex2 = Semaphore(1)        //Protección para unboarders
boarders = 0                 //Pasajeros en el coche
unboarders = 0               //Pasajeros que han bajado del coche

boardQueue = Semaphore(0)    //Los pasajeros esperan aquí antes se subir
unboardQueue = Semaphore(0)  //Los pasajeros esperan aquí para bajar
allAboard = Semaphore(0)     //Coche lleno
allAshore = Semaphore(0)     //Coche vacío
```

```
//Coche
while(1){
  load()
  boardQueue.signal(C) // NO POSIX
  allAboard.wait()

  run()

  unload()
  unboardQueue.signal(C)
  allAshore.wait()
}
```

```
//Viajero
while True{
  boardQueue.wait()
  board()

  mutex1.wait()
  boarders += 1
  if boarders == C{
    allAboard.signal()
    boarders = 0
  }
  mutex1.signal()

  unboardQueue.wait()
  unboard()

  mutex2.wait()
  unboarders += 1
  if unboarders == C {
    allAshore.signal()
    unboarders = 0
  }
  mutex2.signal()
}
```

13. El problema del sushi bar [Kenneth Reek]: …

```
SOLUCION ERRONEA:
eating = waiting = 0    //# número de hilos sentados y esperando
mutex = Semaphore(1)    //Exclusión mutual par alas variables anteriores
must_wait = False       //Bar lleno o vaciándose después de lleno
block = Semaphore(0)    //Cola de clients en espera

mutex.wait()
if (must_wait) {
  waiting += 1
  mutex.signal()
  block.wait()

  mutex.wait() # reacquire mutex POSIBLE ERROR
  waiting -= 1
}

eating += 1
must_wait = (eating == 5)
mutex.signal()

eatSushi();

mutex.wait()
eating -= 1
if (eating == 0) {
  n = min(5, waiting)
  block.signal(n) // ESTO NO EXISTE EN POSIX. Sería como un bucle
  must_wait = False //NACHO CUIDADO: y si n==5???
}
mutex.signal()
```

If a customer arrives while the bar is full, he has to give up the mutex while he waits so that other customers can leave. When the last customer leaves, she signals block, which wakes up at least some of the waiting customers, and clears must wait.

But when the customers wake up, they have to get the mutex back, and that means they have to compete with incoming new threads. If new threads arrive and get the mutex first, they could take all the seats before the waiting threads.

This is not just a question of injustice; it is possible for more than 5 threads to be in the critical section concurrently, which violates the synchronization constraints.

The only reason a waiting customer has to reacquire the mutex is to update the state of eating and waiting, so one way to solve the problem is to make the departing customer, who already has the mutex, do the updating.

```
eating = waiting = 0    //# número de hilos sentados y esperando
mutex = Semaphore(1)    //Exclusión mutual par alas variables anteriores
must_wait = False       //Bar lleno o vaciándose después de lleno
block = Semaphore(0)    //Cola de clients en espera

mutex.wait()
if (must_wait){
  waiting += 1
  mutex.signal()
  block.wait()
}
else{
  eating += 1
  must_wait = (eating == 5)
  mutex.signal()
}

eatSushi();

mutex.wait()
eating -= 1
if eating == 0:
  n = min(5, waiting)
  waiting -= n
  eating += n
  must_wait = (eating == 5)
  block.signal(n)
}
mutex.signal()
```

When the last departing customer releases the mutex, eating has already been updated, so newly arriving customers see the right state and block if necessary. Reek calls this pattern “I’ll do it for you,” because the departing thread is doing work that seems, logically, to belong to the waiting threads.

SOLUCION POCO ELEGANTE (Y POCO DIDACTICA…)

Reek's alternative solution is based on the counterintuitive notion that we can transfer a mutex from one thread to another! In other words, one thread can acquire a lock and then another thread can release it. As long as both threads understand that the lock has been transferred, there is nothing wrong with this.

```
eating = waiting = 0    //# número de hilos sentados y esperando
mutex = Semaphore(1)    //Exclusión mutual par alas variables anteriores
must_wait = False       //Bar lleno o vaciándose después de lleno
block = Semaphore(0)    //Cola de clients en espera

mutex.wait()
if must_wait{
  waiting += 1
  mutex.signal()
  block.wait()                        //when we resume, we have the mutex
  waiting -= 1
}

eating += 1
must_wait = (eating == 5)
if (waiting and not must_wait)
  block.signal()                      // and pass the mutex
else
  mutex.signal()

eatSushi();

mutex.wait()
eating -= 1
if eating == 0
  must_wait = False

if (waiting and not must_wait)
  block.signal()                      //and pass the mutex
else
  mutex.signal()
```

If there are fewer than 5 customers at the bar and no one waiting, an entering customer just increments eating and releases the mutex. The fifth customer sets must wait.

If must wait is set, entering customers block until the last customer at the bar clears must_wait and signals block. It is understood that the signaling thread gives up the mutex and the waiting thread receives it. Keep in mind, though, that this is an invariant understood by the programmer, and documented in the comments, but not enforced by the semantics of semaphores. It is up to us to get it right.

When the waiting thread resumes, we understand that it has the mutex. If there are other threads waiting, it signals block which, again, passes the mutex to a waiting thread. This process continues, with each thread passing the mutex to the next until there are no more chairs or no more waiting threads. In either case, the last thread releases the mutex and goes to sit down.

14. El problema del cuidado de niños [Max Hailperin]: en cierta guardería se debe de cumplir que por cada tres niños debe de estar presente al menos un adulto. Codificar el código correspondiente a los adultos de manera que se cumpla esta restricción y suponiendo que el número de niños se mantiene constante.

Hailperin suggests that you can almost solve this problem with one semaphore.
Listing 7.5: Child care hint

```
multiplex = Semaphore(0)

Adulto{
  mutiplex.signal(3)

  # Atender a los niños

  multiplex.wait()
  multiplex.wait()
  multiplex.wait()

  # Salir de la sala
}
```

Multiplex counts the number of tokens available, where each token allows a child thread to enter. As adults enter, they signal multiplex three times; as they leave, they wait three times. But there is a problem with this solution.
Puzzle: what is the problem?

```
Adulto{
  mutiplex.signal(3)

  # Atender a los niños

  mutex.wait()
    multiplex.wait()
    multiplex.wait()
    multiplex.wait()
  mutex.signal()

  # Salir de la sala

}
```

Amplíe la solución anterior para que tenga en cuenta la posible variación de la cantidad de niños. Codifique el hilo correspondiente a los niños:

Solución página 199 de Allen B. Downey “The Little Book of Semaphores” Version 2.1.5