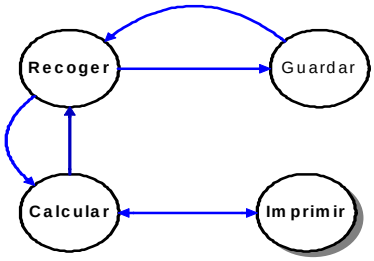
ESCUELA SUPERIOR DE INFORMATICA

SISTEMAS OPERATIVOS

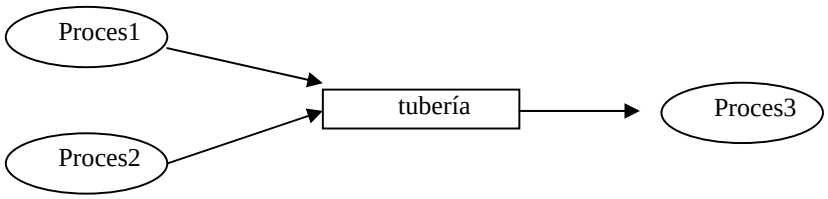
PROBLEMAS DE SINCRONIZACION Y COMUNICACION ENTRE PROCESOS

1. En lugar de utilizar señales, resolver el problema de la sincronización entre procesos que plantea un sistema de recogida de datos, con cuatro procesos: **recoger**, **guardar**, **calcular** e **imprimir** (versión de almacenamiento de un sólo buffer) utilizando las llamadas de gestión de procesos que sean aplicables. Determinar si la solución depende de ciertas presunciones de tiempos. En caso afirmativo, exponer esas presunciones y explicar problemas potenciales de tiempo que se evitan mediante su utilización.



Las llamadas al sistema para gestión de procesos que se emplearán son:
1 create, utilizada como una función a la que se llama con el nombre del proceso que se desea crear y que devuelve un valor entero correspondiente a su identificador de proceso (*pid*).
1 suspend, suspende el proceso que la ejecuta. Su misión será suplir la llamada de espera de señal.
1 resume, que reanuda al proceso cuyo *pid* se le pasa como parámetro. Su misión será la de suplir al mecanismo de envío de señales.

2. Escriba un programa que cree tres procesos que se conecten entre ellos utilizando una tubería tal y como se muestra en la Figura:



Modifique el programa anterior de forma que el proceso 1 genere 1000 números pares y el proceso 2 otros 1000 números impares. Tanto el proceso 1 como el proceso 2 introducen estos números en la tubería de forma que el proceso 3 los extrae e imprime por pantalla. El programa desarrollado debe asegurar que en la tubería nunca se insertan dos números pares seguidos o dos números impares seguidos. Implementar dicha sincoronización (entre el proceso 1 y el proceso 2)
a) usando tuberías
b) usando semáforos y memoria compartida

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

S1. Utilizando la instrucción **XCHG**(*a*,*b*) de intercambio de dos variables en una operación indivisible,
   a. Discutir la corrección (seguridad, interbloqueo, inanición) de la siguiente solución al problema de la exclusión mutua.
   b. Generalizar a *n* procesos
   c. ¿Qué ocurriría si **XCHG** no fuera indivisible?

```
int c;
 process p1() {
   int d;

   d = 0;
   while (1) {
      otro_proceso_1;
      do { XCHG(c,d)}
      while (d==0);
      SECCIÓN_CRÍTICA_1;
      XCHG(c,d)
   }
 };
```

```
process p2() {
  int e;

      e = 0;
      while (1) {
         otro_proceso_2;
         do { XCHG(c,e)}
         while (e==0);
        SECCIÓN_CRÍTICA_2;
         XCHG(c,e)
      }
};
```

```
main() {
  c=1;
  cobegin {p1(); p2(); }
}
```

S2. Programar el algoritmo de Lamport (Stalling 3ª Edición, capitulo 5, ejercicio 5.9) para coordinar por software la exclusión mutua entre N procesos. Explicar la lógica de este algoritmo y discutir si el programa es adecuado para sistemas multiprocesadores.

S3. En la definición de la operación de semáforo **wait** la variable de semáforo sólo se decrementa si su valor no va a resultar negativo. En una implementación alternativa, la variable se decrementa primero y luego se comprueba para ver si se pasa a la sección crítica o no. Dar definiciones revisadas con este criterio para las implementaciones de semáforos con espera activa y con cola de espera, indicar dónde se necesitan operaciones indivisibles y discutir los rangos de valores que las variables semáforos pueden tener. ¿Qué indica un valor negativo en los semáforos así definidos? ¿Afecta esta modificación a la operación **signal**?

S4. Simular con semáforos la operación de una máquina expendedora de un sólo producto. Llevar la cuenta de los productos que quedan en la máquina y la cantidad de dinero depositada por el cliente. Utilizar dos procesos: uno que recibe dinero y lleva el registro de la cantidad pagada, y otro que entrega el producto y el cambio una vez que se introduce la cantidad suficiente y se presiona el botón de la máquina. Si no se ha introducido suficiente dinero o ya no quedan productos, la máquina no entregará el producto y devolverá la cantidad depositada.

S5. El Problema ***del Barbero Dormilón***. Una barbería tiene una sala de espera con *n* sillas, y otra sala donde se encuentra el sillón de afeitado. Si no hay clientes a los que servir, el barbero se echa a dormir en el sillón. Si entra un cliente en la barbería y todas las sillas están ocupadas, el cliente se va. Si el barbero está ocupado afeitando, el cliente se sienta en una de las sillas disponibles. Si el barbero está dormido, el cliente despierta al barbero.
Escribir un programa en BACI que coordine al barbero y los clientes mediante semáforos.

S6. Codificar una solución al *problema de lectores y escritores* dando la misma prioridad a ambos tipos de procesos. Utilizar semáforos como mecanismo de sincronización .

S7. (a) Escribir un programa que lea tarjetas de 80 columnas y las escriba como líneas de 125 caracteres con las siguientes modificaciones: después de cada imagen de tarjeta se inserta un carácter blanco adicional. Cada par adyacente de asteriscos ****** se sustituye por /. Emplear tres procesos concurrentes: *LEER*, *MODIFICAR* e *IMPRIMIR* que compartan sendos buffers sin límite.
(b) Escribir un programa que resuelva el problema anterior con la exigencia adicional de que debe haber un buffer de 10 caracteres entre cada par de procesos (*LEER*,*MODIFICAR*) y (*MODIFICAR*,*IMPRIMIR*). Usar exclusión mutua sobre el buffer limitado.

S8. Considerar las siguientes primitivas de sincronización:
- **ADVANCE**(*A)*: incrementa el valor de la variables *A* en 1.
- **AWAIT**(*A*,*C*): bloquea el proceso que ejecuta esta instrucción hasta que $A > C$

Usando estas primitivas, desarrollar una solución para el *problema productor/consumidor* con buffer limitado si (a) el tamaño del buffer es 1, (b) el tamaño del buffer es *n* ($n > 1$)

MVC1. ***El Problema de los Fumadores***. Considerar un sistema con tres procesos fumadores y un proceso agente. Continuamente cada fumador se prepara un cigarrillo y lo fuma. Para hacer un cigarrillo se necesitan tres ingredientes: tabaco, papel y cerillas. Uno de los procesos tiene papel, otro tabaco, y el tercero cerillas. El agente tiene provisión infinita de los tres ingredientes. El agente coloca dos de los ingredientes en la mesa. El fumador que tiene el ingrediente que falta puede proceder a preparar y fumar un cigarrillo, haciendo una señal al agente cuando acaba. El agente pone otros dos de los tres ingredientes en la mesa y la operación se repite.
Escribir un programa que sincronice al agente y los fumadores mediante semáforos o mutexes y variables condicionales.

MVC2. Simular el tráfico en una intersección de dos calles de dirección única usando operaciones de semáforos. En particular, deben satisfacerse las siguientes reglas:
1. Sólo puede haber un coche cruzando cada vez.
2. Cuando un coche alcanza la intersección y no hay coches que se aproximen desde la otra calle, debe tener permiso para cruzar.
3. Cuando lleguen coches en ambas direcciones, deberán turnarse las preferencias para evitar un retraso indefinido en cualquier dirección.

Utilizar mutexes y variables condicionales como mecanismo de sincronización

MVC3. Referido al siguiente monitor (fichero con exportación de funciones sincronizadas):

Contenido de buffercircular.c

```
mutex_t mutexBC;
ELEMENTO buffer[postotal];
int posenuso = 0,   // 0..postotal
    sigposL = 0,          // 0..postotal-1
    sigposV = 0;          // 0..postotal-1
cond_t L, V;

void Llenar(ELEMENTO datos) {
       mutex_lock(mutexBC);
       while (posenuso == postotal)
           waitc(V, mutexBC);
       buffer [sigposL] = datos;
       posenuso = posenuso + 1;
       sigposL := (sigposL + 1) % postotal;
       signalc(L);
       mutex_unlock(mutexBC);
```

```
};

void Vaciar (ELEMENTO& datos) {
       mutex_lock(mutexBC);
       while (posenuso == 0)
           waitc(L,mutexBC);
       datos = buffer [sigposV];
       posenuso = posenuso - 1;
       sigposV = (sigposV + 1) % postotal;
       signalc(V)
       mutex_unlock(mutexBC);
};
```

responder a las siguientes preguntas:
a. ¿Qué función coloca datos en el buffer circular?
b. ¿Qué función retira datos del buffer circular?
c. ¿Qué disciplina de cola define mejor la operación del buffer circular?
d. ¿Es cierto que *sigposL* >= *sigposV* siempre?
e. ¿Qué sentencias forman la inicialización del monitor?
f. ¿Qué sentencias pueden "despertar" un proceso que espera en una variable condicional?
g. ¿Qué sentencias pueden suspender un proceso?
h. ¿Qué sentencias aseguran que el buffer se recorra circularmente?
i. ¿Qué sentencias modifican una variable crítica compartida para indicar que ya hay otra *pos*(ición) disponible en el buffer?

MVC4. Escribir un programa que implemente el sistema de control de un ascensor utilizando mutexes y variables condicionales. Un proceso servidor aceptará llamadas a los botones de los pisos y desplazará el ascensor al piso solicitado. El ascensor es muy pequeño y sólo puede llevar a una persona cada vez. En el ascensor hay botones que permiten al pasajero elegir el piso de destino. El programa deberá contener una serie de procesos pasajeros que hagan llamadas al ascensor.

MVC5. Un *semáforo acotado s* es un semáforo general que no puede exceder de un valor determinado *smax* > 0. Las operaciones **wait** y **signal** correspondientes se definen como:

wait(*s*): espera hasta que *s* > 0; luego decrementa *s* en 1
signal(*s*): espera hasta que *s* < *smax*; luego incrementa *s* en 1

Escribir un monitor (fichero con funciones sincronizadas mediante mutexes y variables condicionales) que implemente semáforos acotados. Programar con él un buffer finito.

MVC6. Codificar una solución al *problema de los lectores/escritores* dando la misma prioridad a ambos tipos de *procesos* utilizando para ello como mecanismo de sincronización un monitor con cuatro funciones públicas: *IniciarLectura*, *AcabarLectura*, *IniciarEscritura* y *AcabarEscritura*.

MVC7. *Problema **de los filósofos***. Un grupo de filósofos (supongamos 5) dedica todo su tiempo, alternativamente a dos actividades: *pensar* y *comer*. La actividad de *pensar* la llevan a cabo cada uno por separado, pero cuando desean *comer* deben dirigirse a un comedor comunal que dispone de una mesa redonda en donde hay un plato reservado a cada filósofo, y a ambos lados de cada plato sendos palillos, de modo que cada palillo es compartido por dos comensales adyacentes. Para que un filofoso pueda comer es preciso que esté en posesión de sus dos palillos correspondientes.
Programar una solución al *problema de los filósofos* mediante un monitor que disponga de dos procedimientos públicos: *Coger_Palillos* y *Soltar_Palillos*.
La solución propuesta no debe sufrir interbloqueos. Demostrarlo razonadamente. ¿Puede darse alguna situación de aplazamiento indefinido?

ME1. Resolver el *problema de los lectores/escritores* usando mensajes. Contrastar con otras soluciones (semáforos, monitores), y discutir las ventajas y desventajas relativas de la resolución con

mensajes dada la existencia de datos globales. Discutir la posibilidad y méritos de evitar la estructura de datos global en una solución basada en mensajes.

ME2. ¿Qué secuencia de operaciones **send** y **receive** (con bloqueo) debería ejecutar un proceso que desee recibir un mensaje del buzón *M1* o del buzón *M2*? Proporcionar una solución para cada uno de los siguientes casos:

a. El proceso receptor no debe quedar bloqueado en un buzón vacío si hay al menos un mensaje en el otro buzón. La solución sólo puede usar dos buzones y un único proceso receptor.

b. El proceso receptor puede ser suspendido sólo cuando no hay mensajes en cualquiera de los dos buzones y no se permite ninguna forma de espera activa.

ME3. Programar, utilizando mensajes como mecanismo de sincronización, el siguiente algoritmo solución del *problema de los filósofos*: un filósofo que desea comer coge el palillo de su izquierda. Si el palillo de la derecha está disponible, lo coge y comienza a comer; si no, deja el palillo que había cogido y repite el ciclo. Discutir la corrección de este algoritmo.