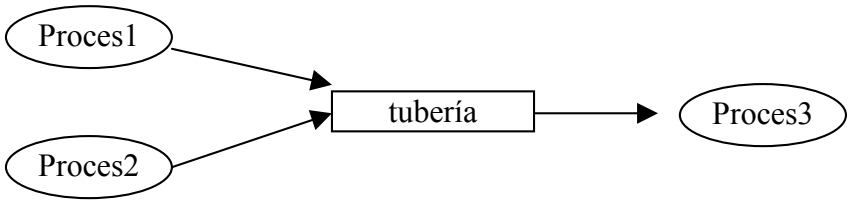
Sistemas Operativos UCM 2012/2013

Módulo 3.2 Comunicación y Sincronización

1. 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 sincronización (entre el proceso 1 y el proceso 2)

a. Usando tuberías
b. Usando semáforos y memoria compartida

2. Implemente las operaciones de semáforos POSIX (`sem_init`, `sem_destroy`, `sem_wait` y `sem_port`) utilizando tuberías.

3. El algoritmo de la panadería de Lamport es un algoritmo de computación creado por el científico en computación Dr Leslie Lamport, para implementar la exclusión mutua de N procesos o hilos de ejecución sin necesidad de ningún soporte HW específico. Se inspira en el funcionamiento normal de cualquier comercio con un tendero y múltiples clientes (ej. una panadería). En este escenario, un cliente que llega a la panadería coge un número con su turno y espera pacientemente a ser atendido. Sin embargo, como se ha ilustrado en otros ejemplos, obtener el número para el turno no es trivial en un computador debido a la posibilidad de que varios procesos reciban el mismo. El siguiente pseudo-código (wikipedia) ilustra la solución de Lamport a dicho problema:

```
// Variables globales
Num[N] = {0, 0, 0, ..., 0};
Eligiendo[N] = {falso, falso, falso, ..., falso};

//Código del hilo i-ésimo
Hilo(i) {
  loop {
    //Calcula el número de turno
    Eligiendo[i] = cierto;
    Num[i] = 1 + max(Num[1],..., Nun[N]);
    Eligiendo[i] = falso;

    //Compara con todos los hilos
    for j in 1..N {
      //Si el hilo j está calculando su número, espera a que termine
      while( Eligiendo[j] ) {yield()}

      while( Num[j]!=0 && (Num[j]<Num[i]||(Num[j]==Num[i] && i<j)) ) {yield()}
    }
```

```
    // Sección crítica
       ...
    // Fin de sección crítica

    Número[i] = 0;

    // Código restante
  }
}
```

Discutir la validez de la solución de Lamport (seguridad, interbloqueo, inanición).

4. Utilizando la instrucción **XCHG(a,b)** (exchange) 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 hilos
   c. ¿Qué ocurriría si **XCHG** no fuera indivisible?

```
int c; //Compartida entre hilos
main() {
  c=1;
  Hilo1(); Hilo2(); // Crea 2 nuevos hilos
}
```

```
Hilo1() {
  int d;

  d = 0;
  while (1) {
    Secc_no_crítica();
    do{
      XCHG(c,d)}
    while(d==0);
    Secc_crítica();
    XCHG(c,d);
  }
}
```

```
Hilo2() {
  int e;

  e = 0;
  while (1) {
    Secc_no_crítica();
    do{
      XCHG(c,e)}
    while(e==0);
    Secc_crítica();
    XCHG(c,e)
  }
}
```

5. Codificar el problema de los **lectores/escritores** de forma que sean los escritores los que tengan prioridad de acceso (conocido como "Segundo Problema de los Lectores/Escritores"). Emplear como método de comunicación/sincronización únicamente semáforos.

6. En el problema de los **lectores/escritores**, si se prioriza a un colectivo, ya sea lectores o escritores, es posible que los procesos priorizados nieguen el acceso al recurso compartido al otro colectivo y que, por lo tanto, se produzca inanición (*starvation*). Codificar una solución para el problema de los lectores/escritores en la cual se otorgue acceso en función del orden de solicitud (conocido como "Tercer Problema de los Lectores/Escritores"). Emplear para ello semáforos.

7. **El Barbero Dormilón**: una barbería está compuesta por una sala de espera, con *n* sillas, y la sala del barbero, que tiene un sillón para el cliente que está siendo atendido. Las condiciones de atención a los clientes son las siguientes:

   a. Si no hay ningún cliente, el barbero se va a dormir
   b. Si entra un cliente en la barbería y todas las sillas están ocupadas, el cliente abandona la barbería
   c. Si hay sitio y el barbero está ocupado, se sienta en una silla libre

d. Si el barbero estaba dormido, el cliente le despierta
e. Una vez que el cliente va a ser atendido, el barbero invocará `cortarPelo()` y el cliente `recibirCortePelo()`

Escriba un programa que coordine al barbero y a los clientes utilizando mutex y semáforos POSIX para la sincronización entre procesos.

8. Un **monitor** lo podemos entender como un objeto cuyos métodos son ejecutados con exclusión mutua, por lo que un hilo/proceso que invoque un método del monitor se puede bloquear a la espera de un evento generado por otro hilo/proceso a través del mismo monitor. Dicha exclusión mutua y espera selectiva, se suele implementar mediante cerrojos y variables condicionales.

Resolver el problema de la **cena de los filósofos** codificando un monitor (mutex y variables condicionales) basándose en el siguiente ejemplo de filósofo situado en la posición i-ésima de la mesa:

```
Filósofo(int i){
  while(1){
    pensar();
    //Solicitamos al monitor la necesidad de coger los palillos
    cogerPalillosMonitor(i);
    comer();
    //Solicitud al monitor que queremos dejar los palillos
    dejarPalillosMonitor(i);
  }
}
```

9. Considerar las siguientes primitivas de sincronización:
   a. **ADVANCE(A)**: incrementa el valor de la variable **A** en 1.
   b. **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 para un único productor y consumidor con tamaño del almacén circular $n > 1$

10. **El Problema de los Fumadores** [Suhas Patil]: 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 infinito papel, otro tabaco y el tercero cerillas. El agente tiene provisión infinita de los tres ingredientes.

El agente coloca dos ingredientes distintos y de forma aleatoria 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 en este instante repite la operación, pone otros dos de los tres ingredientes en la mesa. La operación se repite indefinidamente.

Escribir un programa que sincronice al agente y los fumadores mediante semáforos o mutexes y variables condicionales. Asúmase que el agente no tiene forma de consultar los ingredientes que posee cada fumador.

11. **Una tribu de salvajes** se sirven comida de un caldero con M raciones de estofado de misionero. Cuando un salvaje desea comer, se sirve una ración del caldero a menos que esté vacío. Si está vacío deberá avisar al cocinero para que reponga otras M raciones de estofado, y entonces se podrá servir su ración. Un número arbitrario de salvajes se comportan del siguiente modo:

```
while (True){
  getServingFromPot()
  eat()
}
```

Y el concinero se compora de la siguiente forma:

```
while (True){
  putServingsInPot(M)
}
```

Se deben cumplir las siguientes restricciones:

1. Los salvajes no pueden invocar `getServingFromPot()` si el caldero está vacío
2. El cocinero sólo puede invocar `putServingsInPot(M)` si el caldero está vacío

a. Añada el código necesario para garantizar la correcta sincronización, usando semáforos POSIX y variables de tipo entero, booleano...
b. Añada el código necesario para garantizar la correcta sincronización, usando cerrojos y variables condicionales

12. **El problema de la montaña rusa** [Andrews's Concurrent Programming]: suponga que hay un hilo por cada pasajero, haciendo un total de n, y un hilo para el coche. Los pasajeros están constantemente esperando y subiéndose a la montaña rusa, que puede llevar C pasajeros ($C<n$). El coche sólo puede comenzar un viaje cuando está lleno. Se deben cumplir las siguientes restricciones:

1. Los pasajeros deben invocar `board()` y `unboard()`
2. El coche debe invocar `load(), run()` y `unload()`
3. Los pasajeros no pueden hacer `board()` hasta que el coche haya invocado `load()`
4. El coche no puede partir (`run()`) hasta que no haya C pasajeros en él.
5. Los pasajeros no se pueden bajar hasta que el coche invoque `unload()`

a. Escriba el código necesario usando semáforos POSIX generales (y variables enteras, booleanas…)
b. Escriba el código necesairio usando cerrojos y variables condicionales (y variables enteras, booleanas…)

13. **El problema del sushi bar** [Kenneth Reek]: supóngase un restaurante japonés con 5 asientos, si un cliente llega cuando hay un asiento libre puede sentarse inmediatamente. Sin embargo, si un cliente llega y encuentra los 5 asientos ocupados, supondrá que todos los clientes están cenando juntos y esperará hasta que todos ellos se levanten antes de tomar asiento. Además, los clientes son atendidos por orden. Codifique un programa que se comporte como un cliente según las especificaciones anteriores usando semáforos generales.

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 (con semáforos generales) el código correspondiente a los adultos de manera que se cumpla esta restricción y suponiendo que el número de niños se mantiene constante.