Problemas de Sistemas Operativos

Facultad de Informática, UCM

Comunicación y Sincronización

Problemas Básicos

1.-Estudie el siguiente código que trata de resolver, únicamente por software y sin mediación del sistema operativo, el problema de la exclusión mutua entre dos hilos:

```
void thread1()
{
  while(TRUE){
   //Espera activa
   while(turn!=0);
   <<Sección crítica T1>>
   turn = 1;
   <<Más código T1>>
  }
}
```

```
void thread2()
{
  while(TRUE){
   //Espera activa
   while(turn!=1);
   <<Sección crítica T2>>
   turn = 0;
   <<Más código T2>>
  }
}
```

¿Resuelve correctamente el problema de la sección crítica garantizando exclusión, progreso y espera limitada?. Justifique su respuesta.

2.-Escriba un programa que cree tres hilos que se comunicarán entre ellos. El hilo 1 genera los 1000 primeros números pares y el hilo 2 los 1000 números impares. El hilo 3 irá leyendo esos números e imprimiéndolos en pantalla. Se debe garantizar que los números escritos por pantalla estén en orden: 1,2,3,4,5... Implementar dicha sincronización mediante:

a) Cerrojos y variables condicionales

b) Semáforos

3.-Implementar la funcionalidad de un semáforo general a partir de cerrojos y variables condicionales. Para ello, defina un tipo de datos llamado `sem_t` (un struct en C con los campos necesarios), y las operaciones *wait* y *signal* de acuerdo a la semántica estudiada en clase.

4.-Resolver el problema de los *lectores/escritores* utilizando un mutex, variables condicionales y otras variables compartidas (enteros y booleanos).

a) Codificar una solución que de prioridad a los lectores en el acceso a la sección crítica: *ningún lector debería esperar a acceder a la sección crítica si hay más lectores dentro de la sección crítica.*

b) Codificar una solución que de prioridad a los escritores en el acceso a la sección crítica: *si un escritor desea acceder a la sección crítica, debería poder acceder lo antes posible.*

5.-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 puede implementar mediante cerrojos y variables condicionales.

Resolver el problema de la **cena de los filósofos** codificando un monitor (usando cerrojos y variables condicionales) basándose en el siguiente ejemplo de filósofo situado en la posición i-ésima de la mesa. El monitor debe incluir la implementación de `cogerPalillosMonitor(i)` y `dejarPalillosMonitor(i)` así como la declaración de variables (privadas al monitor) que sean necesarias.

```
void 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);
  }
}
```

6.-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, notificando 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.

7.-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()
}
```

El cocinero se comporta como sigue:

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

Las restricciones de sincronización son las siguientes:

- Los salvajes no pueden invocar `getServingFromPot()` si el caldero está vacío.
- El cocinero solo puede invocar `putServingsInPot()` si el caldero está vacío.

Realize las modificaciones necesarias en el código de los salvajes y el cocinero para garantizar la correcta sincronización usando los siguientes recursos:

a) Cerrojos, variables condición y otras variables compartidas (enteros y booleanos).
b) Semáforos y otras variables compartidas.

8.-Simular con hilos el comportamiento de una gasolinera con 2 surtidores de pago directo con tarjeta. Cuando un cliente coge un surtidor puede servirse el combustible deseado (con una llamada a la función `void PumpFuel(int pump_id, int price)`). Una vez servido el combustible dejará libre el surtidor para que otro cliente pueda usarlo. Para que funcione correctamente el servicio deben satisfacerse las siguientes restricciones:

- Los clientes deben acceder al servicio en orden de llegada, pudiendo escoger cualquiera de los surtidores libres.
- No puede haber más de un cliente simultáneamente en un mismo surtidor.
- Mientras un cliente se está sirviendo el combustible en un surtidor, cualquier otro cliente debe poder utilizar el otro surtidor si está libre.

Cada cliente, modelado como un hilo del programa concurrente, se comporta del siguiente modo:

```
void cliente(int dinero) {
      int pump;
      pump=getUnusedPump();
      /* Uso del surtidor */
      PumpFuel(pump,dinero);
      /* Deja de usar el surtidor */
      releasePump(pump);
}
```

Implementar las funciones `getUnusedPump()` y `releasePump()` pare imponer las restricciones de sincronización arriba mencionadas. Notese que la función `getUnusedPump()` bloqueará a un cliente hasta que se le asigne un surtidor libre, cuyo identificador se devuelve como valor de retorno de esa función.

9.-Se desea simular el trabajo en un muelle de carga, usando hilos para simular los operarios que realizan la carga y los camiones que son cargados. El siguiente código describe la estructura de los hilos utilizados para los trabajadores y los camiones:

```
#define NP 10 //Capacidad máxima de camión

void *truck(void *arg)
{
  while (1) {

    // esperar a poder entrar

    enter_dock();

    // esperar a camión cargado

    delivery();
  }
}
```

```
void *worker(void *arg)
{
  while (1) {
    p = get_parcel();

    // esperar a poder cargar


    load_parcel(p);

    // notificar si cargado

  }
}
```

Los operarios (worker) repiten indefinidamente las acciones de coger un paquete (`get_parcel`) y cargarlo en el camión (`load_parcel`). Varios operarios pueden estar cogiendo un paquete al mismo tiempo, pero cargarán el camión de uno en uno. El hilo del operario sólo debe ejecutar la función `load_parcel` cuando haya algún camión en el muelle y quede sitio en dicho camión para el paquete.

Los camiones (truck) tiene capacidad para cargar `NP` paquetes, y repiten indefinidamente las acciones de entrar al muelle (`enter_dock`) para recoger la carga y salir al reparto (`delivery`). Varios hilos de camión pueden estar ejecutando la función `delivery` al mismo tiempo. Sin embargo, en el muelle sólo cabe un camión y por tanto el hilo que simula al camión debe esperar a que el muelle quede libre para ejecutar la función `enter_dock` en exclusiva. Asimismo, el hilo debe esperar a que el camión esté completamente cargado para salir del muelle y empezar el reparto.

Completa las funciones `truck` y `worker` para que los hilos se sincronicen correctamente, añadiendo las variables que consideres necesarias.

10.-Se desea simular con hilos un pequeño torneo de poker entre amigos. La función de entrada de los hilos que simulan los jugadores tiene la siguiente estructura:

```
void play(void);

void *player(void *arg)
{
  while (1) {
    seat_at_table();
    play();
    leave_table();
  }
}
```

La función `seat_at_table` debe garantizar que cada jugador accede a la mesa en estricto orden de llegada, mientras no haya empezado la siguiente partida. Asimismo, el hilo no debe ejecutar la función `play` mientras no se hayan sentado a la mesa los 4 jugadores que jugarán la siguiente partida. Los hilos de los 4 jugadores que juegan juntos deben poder ejecutar la función `play` concurrentemente.

Cuando la partida termine (retorno de la función `play`) el último jugador en dejar la mesa (`leave_table`) deberá indicar al resto que la partida actual ha finalizado y que puede empezar una nueva partida.

Implementar las funciones `seat_at_table` y `leave_table` para que los hilos se sincronicen correctamente, añadiendo las variables globales necesarias.

Problemas Adicionales

11.-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.

12.-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.

13.-El algoritmo de Peterson es una solución software correcta para el problema de la sección crítica. A continuación se muestra el algoritmo de Peterson para dos procesos/hilos. Justifique que, efectivamente, cumple los tres requisitos de cualquier solución correcta del problema de la sección crítica.

```
turno = 0
interesado = {false, false}
```

Hilo 0

```
interesado[0] = true
turno = 1
//no hace nada, espera ocupada
while(interesado[1]&&turno==1);
sección_crítica();
//fin de la sección crítica
interesado[0] = false
sección_no_critica();
```

Hilo 1

```
interesado[1] = true
turno = 0
//no hace nada, espera ocupada
while(interesado[0]&&turno==0);
sección_crítica();
//fin de la sección crítica
interesado[1] = false
sección_no_critica();
```

14.-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, 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 (fuente: 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).

15.-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.

16.-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 $n > 1$

17.-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:

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

A partir de las especificaciones realize los siguiente:

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...)

18.-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.

19.-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.
Amplíe la solución anterior para que tenga en cuenta la posible variación de la cantidad de niños. Codifique el hilo correspondiente a los niños.