Práctica 4: Sincronización de procesos e hilos

Índice

1 Objetivos

El objetivo de esta práctica es afianzar nuestro conocimiento de los mecanismos de sincronización de hilos y procesos que ofrece un sistema POSIX y sus esquemas de uso. En la práctica haremos uso de: mutex, variables de condición, semáforos y objetos de memoria compartida POSIX.

2 Ejercicios

La práctica está organizada en dos partes o ejercicios:

- Sincronización de hilos de un proceso
- Sincronización y comunicación de hilos de distintos procesos

En cada una de estos ejercicios el alumno deberá desarrollar uno o más programas que modelen un problema de comunicación y sincronización de la vida real. Los entes a comunicar serán modelados en cada caso como hilos o procesos y se deberán utilizar los mecanismos adecuados para que estos puedan compartir datos (comunicación) y coordinar su ejecución y acceso a los datos (sincronización).

El archivo ficheros_p4.tar.gz contiene una serie de ficheros que pueden ser usados como punto de partida para el desarrollo los ejercicios de esta práctica, así como unos makefiles que pueden ser usados para la compilación de los distintos proyectos.

Ejercicio1: Sincronización de hilos de un proceso

Se desea diseñar el sistema de control del aforo de una discoteca en la que el número máximo de clientes que pueden estar dentro en un momento dado (aforo) es N. Además, hay dos tipos de clientes: los vip y los normales. Las reglas que debe cumplir el sistema son las siguientes:

- Si el aforo está completo los nuevos clientes deberán eseperar a que salga algún cliente de la discoteca para poder entrar.
- Si hay esperando clientes vip y clientes normales, se les dará prioridad a los clientes vip. Los clientes normales deberán por tanto esperar a que entren los vip primero.

- Los clientes entrarán de uno en uno en orden estricto de llegada según su grupo (normales o vips), mientras el número de ocupantes de la discoteca sea menor que el aforo (N).

Se debe simular el sistema con un programa que cree M hilos que representan a los clientes de la discoteca. Estos hilos deberán ejecutar una función de entrada llamada **client**, con la siguiente estructura:

```
void *client(void *arg)
{
    ...

    if ( isvip )
        enter_vip_client( ... );
    else
        enter_normal_client( ... );
    dance( ... );
    exit_client( ... );

    ...
}
```

En su creación a cada hilo se le pasarán dos argumentos enteros:

- id: un identificador de la tarea que corresponde con el orden de creación del hilo.
- isvip: un valor que indique si el cliente es vip o no.

Diseñar las funciones **enter_vip_client**, **enter_normal_client** y **exit_client** de forma que garanticen las condiciones de funcionamiento del sistema descritas arriba. Añadir en estas funciones mensajes por consola con *printf* que permitan hacer el seguimiento del programa, indicando en cada mensaje el id del cliente y lo que está haciendo.

El programa principal recibirá por la línea de comandos el nombre de un fichero que contendrá el número de clientes a crear (M) y el carácter vip o normal de cada uno de estos clientes. El formato esperado de este fichero es el siguiente:

- Un fichero ascii organizado por líneas
- La primera línea indica el número de clientes a crear (M)
- Las M lineas siguientes (una por cliente) tomarán el valor 1 o el valor 0 para indicar el carácter vip de dicho cliente (nótese que el 1 y el 0 son caracteres ascii).

Por ejemplo, para 5 clientes con 3 no vip y 2 vip el contenido del fichero podría ser:

```
5
0
0
1
0
1
```

El fichero puede parsearse fácilmente utilizando la función de la biblioteca estándar de C *fscanf*, consultar su página de manual.

Ejercicio2: Sincronización y comunicación de hilos de distintos procesos

Este ejercico consiste en resolver el problema de la tribu de salvajes de la hoja de problemas de la asignatura, usando distintos procesos en lugar de hilos de un mismo proceso para simular cada uno de los salvajes y el cocinero. El enunciado del problema clásico es como sigue:

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 cocinero y número arbitrario de salvajes se comportan del siguiente modo:

```
//Cocinero:
while (!finish) {
  putServingsInPot()
}

//Salvajes:
for (i = 0; i < NUMITER; i++){
  getServingFromPot()
  eat()
}
```

Se deben cumplir las siguientes restricciones: - Los salvajes no pueden invocar getServingFromPot() si el caldero está vacío - El cocinero sólo puede invocar putServingsInPot() si el caldero está vacío

Para simular este problema vamos a crear dos programas:

- Un programa *cocinero.c* que cree los recursos compartidos necesarios y luego ejecute la función cocinero(). El usuario sólo debe crear un proceso que ejecute este programa. Este programa registrará un manejador para las señales SIGTERM y SIGINT. Cuando sean capturadas el programa debe terminar, limpiando todos los recursos compartidos creados en el sistema.
- Un programa *salvaje.c*, que intente abrir los recursos compartidos que haya creado el programa cocinero. Si no los encuentra, avisará con un mensaje de error advirtiendo al usuario que debe ejecutar primero el programa cocinero. Una vez abiertos los recursos compartidos ejecutará la función salvajes(), que intentará comer NUMITER veces del caldero y después terminará. El usuario puede crear tantos procesos que ejecuten este programa como quiera, en distintos terminales o en el mismo terminal lanzando los procesos en *background* (por ejemplo puede usar un bucle for de bash y lanzar varios procesos que ejecuten este programa).

Se aconseja a los estudiantes utilizar semáforos con nombre para la sincronización de los distintos procesos y una región de memoria compartida para alojar la variable que representa el contenido del caldero.

Las funciones *putServingsInPot()*, *getServingsFromPot()* y *eat()*, además de implementar la sincronización necesaria deben mostrar por el terminal menajes que permitan comprobar el funcionamiento del programa, indicando en cada mensaje el id del proceso correspondiente.