Concurrencia entre procesos.

Índice de contenidos

1.- Objetivos

- Utilizar mecanismos de gestión de la concurrencia (Semáforos).
- Gestionar la interacción entre diferentes procesos *LINUX* utilizando los mecanismos de sincronización que ofrece.

2.- Mecanismos de sincronización de procesos en *LINUX*

Linux ofrece tres mecanismos de sincronización y comunicación de procesos:

- Memoria compartida: Para lo que utiliza la librería *shm.h*.
- Semáforos: Para lo que utiliza la librería *sem.h.*
- Colas de mensajes: Para lo que utiliza la librería *msg.h.*

Memoria compartida (*shm.h*)

Es posible hacer que dos procesos distintos sean capaces de compartir una zona de memoria común y, de esta manera, compartir o comunicarse datos. La forma de conseguirlo en un programa *C* es la siguiente:

En primer lugar necesitamos construir una clave de acceso compartido, de tipo key_t, que sea común para todos los programas que quieran compartir la zona de memoria. Para ello existe la función ftok():

```
key_t ftok(char *path, int i);
```

Recibe los siguientes parámetros:

- `path`: Ruta de un fichero que exista y se accesible.
- `i`: Cualquier número entero.

Con esto se construye una clave que que sirve como llave de acceso a la zona de memoria compartida. Si todos los programas utilizan el mismo fichero y el mismo número entero, obtendrán la misma clave.

Habitualmente como primer parámetro pasaremos algún fichero del sistema que sepamos seguro de su existencia, como por ejemplo */bin/ls*.

Para el segundo argumento, bastaría con poner definir una constante común en el código de los

programas que vayan a acceder a la zona de memoria compartida.

Una vez obtenida la clave, se crea la zona de memoria. Para ello usamos la función *shmget()*:

```
int shmget(key_t key, int size, int flg);
```

Con dicha función creamos la zona de memoria compartida. Nos devuelve un identificador para dicha zona. Si la zona de memoria correspondiente a la Clave **key** ya estuviera creada, simplemente nos daría el identificador (siempre y cuando los parámetros no indiquen lo contrario).

Recibe los siguientes parámetros:

- **key**: El primer parámetro es la clave *key_t* obtenida anteriormente y que debería ser la misma para todos los programas que quieran acceder a la zona de memoria compartida.
- **size**: Tamaño en bytes que deseamos para la memoria.
- **flg**: Códigos bandera (*flags*) que nos permiten modificar algunas opciones:
  - Los 9 bits menos significativos, son permisos de lectura/escritura/ejecución para propietario/grupo/otros, al igual que en los ficheros. Para obtener una zona de memoria con todos los permisos para todo el mundo, debemos poner como parte de los *flags* el número *0777*. Es importante el cero delante, para que el número se interprete en octal y queden los bits en su sitio (En C, cualquier número que empiece por cero, se considera octal).
  - *IPC_CREAT*: Junto con los bits anteriores, este bit indica si se debe crear la memoria en caso de que no exista. Si está puesto, la zona de memoria memoria se creará, si no lo está ya, y se devolverá el identificador. Si no está puesto, se intentará obtener el identificador y se obtendrá un error si no está ya creada.

  En resumen, este parámetro debería tener un valor similar a *0777 | IPC_CREAT*.

El último paso poder usar la memoria consiste en obtener un puntero que apunte a ella para poder escribir o leer sobre ella. Declaramos en nuestro código un puntero al tipo que sepamos que va a haber en la zona de memoria (una estructura, un vector, tipos básicos, etc.) y utilizamos la función *shmat()*:

```
char *shmat(int id, char *addr, int flg);
```

Recibe los siguientes parámetros:

- **id**: Identificador de la memoria obtenido en el paso anterior.
- Para **addr** y **flg** bastará con que le pasemos el valor *0*.

El puntero devuelto es de tipo *char**. Debemos hacerle un *cast* al tipo que queramos. Por ejemplo: *(mi_estructura *)shmat(...);*

Esta función lo que en realidad hace, además de darnos el puntero, es asociar la memoria compartida a la zona de datos de nuestro programa, por lo que es necesario llamarla sólo una vez en cada proceso. Si queremos más punteros a la zona de memoria, bastará con igualarlos al que ya tenemos.

Ahora ya estamos en condiciones de utilizar la memoria. Cualquier cosa que escribamos en el contenido de nuestro puntero, se escribirá en la zona de memoria compartida y será accesible para

los demás procesos.

Una vez hemos terminado de usar la memoria, debemos liberarla. Para ello utilizamos las funciones *shmdt()* e *shmctl()*:

```
int shmdt(char *addr);
int shmctl(int id, int flg, struct shmid_ds *shm);
```

La primera función desasocia la memoria compartida de la zona de datos de nuestro programa. Basta pasarle el puntero que tenemos a la zona de memoria compartida y llamarla una vez por proceso.

La segunda función destruye realmente la zona de memoria compartida. Hay que pasarle el identificador de memoria obtenido con *shmget()*, un *flag* que indique que queremos destruirla *(IPC_RMID),* y un tercer parámetro al que bastará con pasarle *NULL*. A esta función sólo debe llamarla uno de los procesos.

Ejemplo: En los archivos *shm1.c* y *shm2.c* tenemos dos programas que comparten memoria. El primero crea una zona de memoria con capacidad para 100 números enteros. Empieza a escribir en la primera posición de memoria, con espera de un segundo, los números del 1 al 10. El segundo programa lee, a intervalos de un segundo, el número guardado en la primera posición de memoria de la zona compartida y lo muestra en pantalla.

Semáforos (*sem.h*)

El funcionamiento de los semáforos implementados en la librería *sem.h* es como el de una variable contador. Imaginemos que el semáforo controla un fichero y que inicialmente tiene el valor *1*. Cuando un proceso quiere acceder al fichero, primero debe decrementar el semáforo. El contador queda a *0* y como NO es negativo, deja que el proceso siga su ejecución y, por tanto, acceda al fichero.

Ahora un segundo proceso lo intenta y para ello también decrementa el contador. Esta vez el contador se pone a *-1* y como es negativo, el semáforo se encarga de que el proceso quede en una cola de espera. Este segundo proceso no continuará por tanto su ejecución y no accederá al fichero.

Supongamos ahora que el primer proceso termina de escribir el fichero. Al acabar con el fichero debe incrementar el contador del semáforo. Al hacerlo, este contador se pone a *0*. Como NO es negativo, el semáforo se encarga de mirar el la cola de procesos pendientes y desbloquear al primer proceso de dicha cola. Con ello, el segundo proceso que quería acceder al fichero continua su ejecución y accede al fichero.

Cuando este proceso también termine con el fichero, incrementa el contador y el semáforo vuelve a ponerse a *1*.

Es posible hacer que el valor inicial del semáforo sea, por ejemplo, *3*, con lo que pasarán los tres primeros procesos que lo intenten. Pueden a su vez quedar muchos procesos encolados simultáneamente, con lo que el contador quedará con un valor negativo grande. Cada vez que un

proceso incremente el contador (libere el recurso común), el primer proceso encolado despertará. Los demás seguirán dormidos.

Como vemos, el proceso de los semáforos requiere colaboración de los procesos. Un proceso debe decrementar el contador antes de acceder al fichero e incrementarlo cuando termine. Si los procesos no siguen este protocolo (y pueden no hacerlo), el semáforo no sirve de nada. Para utilizar los semáforos en un programa, deben seguirse los siguientes pasos:

Primero debemos obtener una clave de recurso compartido. Para ello utilizaremos la función *ftok()* descrita previamente.

Después hay que btener un array de semáforos con la función semget():

```
int semget(key_t key, int n, int flg):
```

Recibe como parámetros:

- **`key`**: Clave obtenida en el paso anterior.
- **`n`**: Número de semáforos.
- **`flg`**: Códigos bandera. Estos flags permiten poner los permisos de acceso a los semáforos, similares a los de los ficheros, de lectura y escritura para el usuario, grupo y otros. También lleva unos modificadores para la obtención del semáforo.
  Normalmente usaremos *0600 | IPC_CREATE* que indica permiso de lectura y escritura para el propietario y que los semáforos se creen si no lo están al llamar a *semget()*. Es importante el 0 delante del 600, así el compilador de *C* interpretará el número en octal y pondrá correctamente los permisos.

La función *semget()* nos devuelve un identificador del vector de semáforos.

Tras esto, uno de los procesos debe inicializar el semáforo con la función *semctl()*:

```
int semctl(int id, int index, int cmd, …);
```

Recibe los siguientes parámetros:

- **`id`**: Identificador del vector de semáforos obtenido en el paso anterior.
- **`index`**: Índice del semáforo que queremos inicializar dentro del vector de semáforos.
- **`cmd`**: Indica qué queremos hacer con el semáforo. En función de su valor, los siguientes parámetros serán una cosa u otra. En nuestro ejemplo, que queremos inicializar el semáforo, el valor del tercer parámetro es *SETVAL*.

Ahora podemos usar los semáforos. El proceso que quiera acceder a un recurso común debe primero decrementar el semáforo. Para ello se utiliza la función *semop()*:

```
int semop(int id, struct sembuf *ops, size_t nops);
```

Recibe los siguientes parámetros:

1. **`id`**: Identificador del vector de semáforos obtenido con *semget()*.
2. **`Ops`**: Vector de operaciones sobre el semáforo. Para decrementarlo basta con un vector de una posición.
3. **`Nops`**: Número de elementos del vector anterior.

La estructura del segundo parámetro contiene tres campos:

- *sem_num:* Índice del vector del semáforo sobre el que queremos actuar.
- sem_op: El valor en el que queremos decrementar el semáforo. En nuestro caso, *-1*.
- *sem_flg*: Códigos bandera que afectan a la operación. Utilizaremos *0*.

Al realizar esta operación, si el semáforo se vuelve negativo, nuestro proceso se quedará bloqueado hasta que alguien incremente el semáforo y lo haga, como mínimo, *0*.

Cuando el proceso termine de usar el recurso común, debe incrementar el semáforo. La función a utilizar es la misma, pero poniendo *1* en el campo *sem_op* de la estructura *sembuf*.

Ejemplo: Los ficheros *sem1.c* y *sem2.c* contienen el código de dos programas que nos permiten probar el uso de semáforos de la librería *sem.h*.

sem1 tiene un bucle infinito para entrar en el semáforo. Escribe en pantalla cuando entra y cuando sale. Como el semáforo está cerrado por defecto, *sem1* entra en él y no sale hasta que *sem2* lo abre.

sem2 tiene un bucle con 10 iteraciones en el que abre el semáforo y espera un segundo. El resultado es que *sem1* entra en el semáforo, queda bloqueado un segundo, sale del semáforo para volver a entrar en él y repetir el proceso 10 veces.

Colas de mensajes (*msg.h*)

Es posible hacer que dos procesos distintos sean capaces de enviarse mensajes (estructuras de datos) y de esta forma pueden intercambiar información. El mecanismo para conseguirlo es el de una cola de mensajes. Los procesos introducen mensajes en la cola y se van almacenando en ella. Cuando un proceso extrae un mensaje de la cola, extrae el primer mensaje que se introdujo y dicho mensaje se borra de la cola.

También es posible hacer tipos de mensajes distintos, de forma que cada tipo de mensaje contiene una información distinta y va identificado por un entero. Los procesos luego pueden retirar mensajes de la cola selectivamente por su tipo.

La forma de implementar una cola de mensajes en un programa es la siguiente:

Primero obtenemos una clave de recurso compartido. Para ello utilizaremos la función *ftok()* descrita previamente.

Una vez obtenida la clave, se crea la cola de mensajes. Para ello existe la función *msgget()*:

```
int msgget(key_t key, int flg);
```

Recibe los siguientes parámetros:

- **`key`**: Clave obtenida en el paso anterior.
- **`Flg`**: Códigos bandera (*flags*) que nos permiten modificar algunas opciones:

- Los 9 bits menos significativos, son permisos de lectura/escritura/ejecución para propietario/grupo/otros, al igual que en los ficheros. Para obtener una zona de memoria con todos los permisos para todo el mundo, debemos poner como parte de los *flags* el número *0777*. Es importante el cero delante, para que el número se interprete en octal y queden los bits en su sitio (En C, cualquier número que empiece por cero, se considera octal).
- *IPC_CREAT*: Junto con los bits anteriores, este bit indica si se debe crear la memoria en caso de que no exista. Si está puesto, la zona de memoria memoria se creará, si no lo está ya, y se devolverá el identificador. Si no está puesto, se intentará obtener el identificador y se obtendrá un error si no está ya creada.

En resumen, este parámetro debería tener un valor similar a *0777 | IPC_CREAT*.

Ahora podemos utilizar la cola de mensajes creada en el paso anterior. Para meter un mensaje en la cola, se utiliza la función *msgsnd()*:

```
int msgsnd(int id, struct msgbuf *msg, int size, int flg);
```

Recibe los siguientes parámetros:

- **`id`**: Identificador de la cola obtenido con *msgget()*.
- **`msg`**: El segundo parámetro es el mensaje en sí. El mensaje debe ser una estructura cuyo primer campo sea de tipo *long*. En dicho atributo se almacena el tipo de mensaje. El resto de los campos pueden ser de cualquier tipo que se desee enviar (otra estructura, campos sueltos, etc.). Al pasar el mensaje como parámetro, se pasa un puntero al mensaje y se hace un *cast* a *struct msgbuf**.
- **`size`**: Tamaño en *bytes* del mensaje exceptuando el primer atributo de tipo *long*, es decir, el tamaño en bytes de los campos con la información.
- **`flg`**: Códigos bandera (*flags*) que nos permiten modificar algunas opciones. Lo más habitual es poner *0* o bien *IPC_NOWAIT*. En el primer caso la llamada a la función queda bloqueada hasta que se pueda enviar el mensaje. En el segundo caso, si el mensaje no se puede enviar, se vuelve inmediatamente con un error. El motivo habitual para que el mensaje no se pueda enviar es que la cola de mensajes esté llena.

Para recoger un mensaje de la cola se utiliza la función *msgrcv()*:

```
ssize_t msgrcv(int id, struct msgbuf *msg, int size, int type, int flg);
```

Recibe los siguientes parámetros:

- **`id`**: Identificador de la cola obtenido con *msgget()*.
- **`msg`**: El segundo parámetro es un puntero a la estructura donde se desea recoger el mensaje. Puede ser, como en la función *msgsnd()*, cualquier estructura cuyo primer campo sea de tipo *long* para el tipo de mensaje.
- **`size`**: Tamaño en *bytes* del mensaje exceptuando el primer atributo de tipo *long*, es decir, el tamaño en bytes de los campos con la información.
- **`type`**: El cuarto parámetro entero es el tipo de mensaje que se quiere retirar. Se puede indicar un entero positivo para un tipo concreto o un 0 para cualquier tipo de mensaje.
- **`flg`**: Códigos bandera (*flags*) que nos permiten modificar algunas opciones. Lo más habitual es poner *0* o bien *IPC_NOWAIT*. En el primer caso, la llamada a la función se queda

bloqueada hasta que haya un mensaje del tipo indicado. En el segundo caso, se vuelve inmediatamente con un error si no hay mensaje de dicho tipo en la cola.

Una vez hemos terminado de usar la cola, se debe liberar. Para ello se utiliza la función *msgctl()*:

```
int msgctl (int id, int cmd, struct msqid_ds *buf);
```

Es una función genérica para control de la cola de mensajes con posibilidad de varios comandos. Aquí sólo se explica como utilizarla para destruir la cola. Recibe los siguientes parámetros:

- **`id`**: Identificador de la cola obtenido con *msgget()*.
- **`cmd`**: El segundo parámetro es el comando que se desea ejecutar sobre la cola, en este caso *IPC_RMID*.
- **`buf`**: El tercer parámetro son datos necesarios para el comando que se quiera ejecutar. En este caso no se necesitan datos y se pasará *NULL*.

Ejemplo: En los archivos *msg1.c* y *msg2.c* hay dos programas que ilustran el manejo de una cola de mensajes.

msg1 abre la cola de mensajes, envía un mensaje de tipo *1* para que lo lea *msg2* y espera un mensaje de tipo *2* . Cuando llega el mensaje de tipo *2*, *msg1* destruye la cola de mensajes.

Por su parte, *msg2* abre la cola mensajes y espera un mensaje tipo *1*. Cuando lo recibe, envía un mensaje de tipo *2* y se sale. Deja la destrucción de la cola para *msg1*.