SESIÓN N° 9 Y 10

SINCRONIZACIÓN DE PROCESOS EN C
SEMÁFOROS Y MEMORIA COMPARTIDA

I
OBJETIVOS

- Conocer y aplicar los conceptos de mecanismos de sincronización para procesos pesados y de compartición de memoria.
- Elaborar programas que sincronicen la ejecución de varios procesos y/o que compartan memoria.

II
TEMAS A TRATAR

- Funciones para la sincronización de procesos
- Funciones de memoria compartida

III
MARCO TEORICO

La concurrencia y la sincronización entre procesos es un problema de los Sistemas Operativos, en el que se debe de controlar el acceso a recursos compartidos que los procesos están utilizando (en este caso ya no hay variables como en los hilos), o cuando se requiere que los procesos se pongan de acuerdo en la ejecución de ciertas actividades las cuales ocurren en determinados momentos.

Los procesos pesados en Linux no comparten memoria, ni siquiera los padres con sus hijos. Por tanto, hay que establecer algún mecanismo en caso de que se quiera comunicar información entre proceso concurrentes. Linux define tres clases de herramientas de comunicación entre procesos (IPC): los semáforos, la memoria compartida y los mensajes. En esta sesión nos enfocaremos en los semáforos y la memoria compartida.

Semáforos

Los semáforos es un mecanismo de los sistemas operativos cuyo objetivo es evitar los interbloqueos y abrazos mortales de los procesos pesados. También son utilizados para resolver problemas de sincronización.

Para utilizar los semáforos en un programa, deben seguirse los siguientes pasos:

- **Obtener una clave de semáforo**
  Esta clave se trata de una clave para un recurso compartido, y se utiliza la función ***ftok*** para este propósito. Esta función también se utiliza para la memoria compartida y las colas de mensajes.

```
key_t ftok(char *path, int num)
```

El primer parámetro es el nombre y la ruta de un archivo cualquiera. Y el segundo parámetro es un número cualquiera. Todos los procesos que quieran compartir el semáforo deben suministrar el mismo archivo y el mismo número.

- **Obtener un conjunto de semáforos**

```
int semget(key_t key, int size, int flags)
```

El primer parámetro es la clave obtenida en el paso anterior o IPC_PRIVATE, el segundo parámetro es el número de semáforos que contendrá el conjunto y el tercer parámetro son algunos flags los cuales permiten poner permisos de acceso al semáforo (es similar a los ficheros) de lectura, escritura para el usuario, grupo y otros. Así mismo, lleva unos modificadores para la obtención de los semáforos. Por ejemplo, nosotros pondremos 0600 | IPC_CREAT que indica permiso de lectura y escritura para el propietario y que los semáforos se crearán sino existen al momento de invocar a *segmen()*. Es importante colocar el 0 delante del 600 para que el compilador del C interprete el número en un formato octal. Muchas veces se usa IPC_CREAT | SHM_W | SHM_R.
La función *semget()* nos devuelve un identificador al conjunto de semáforos.

- **Inicialización del semáforo**

Uno de los procesos que compartan el semáforo debe inicializarlo. Para esto utiliza la función.

```
int semctl(int idSem, int idx, int setVal, int uni)
```

donde el primer parámetro es el identificador del arreglo de semáforos obtenido en el paso anterior, el segundo es el índice del semáforo en el arreglo el cual queremos inicializar (si sólo hemos pedido uno, este parámetro es 0). El tercer parámetro indica qué queremos hacer con el semáforo. Por ejemplo, si ponemos SETVAL significa que queremos inicializarlo. El cuarto parámetro podría ser un entero, pero en realidad es un tipo de unión. Entonces, si este parámetro es 1 el semáforo estará en "verde" y si le ponemos 0 es que queremos un semáforo en "rojo".

Si vamos a trabajar con la unión, está se debe de definir de forma explícita en los programas que vayamos a hacer (ver el *man* para más detalle) de la siguiente forma:

```
union semun {
    int val;                    //valor para SETVAL
    struct semid_ds* buf;       //buffer para IPC_STAT, IPC_SET
    unsigned short* array;      //arreglo para GETALL, SETALL
    struct seminfo* __buf;      //buffer para IPC_INFO
};
```

- **Utilizar el semáforo**

Hasta este punto ya está todo preparado, solo falta que los procesos lo consuman. El proceso que quiera acceder al recurso común (puede ser una sección crítica) debe primero decrementar el semáforo. Para esto se utilizará la función:

```
int semop(int, struct sembuf *, size_t)
```

el primer parámetro es el identificador del arreglo de semáforos, el segundo índica las operaciones que se realizarán en el semáforo. El tercer parámetro es el número de elementos del arreglo, en nuestro ejemplo será 1.

La estructura del segundo parámetro ***struct sembuf*** contiene los siguientes campos:

- `short` **`sem_num`** que es el índice del arreglo de semáforos sobre el cual queremos hacer la operación. Si es un solo semáforo este valor será 0.
- `short` **`sem_op`** es el valor en el que queremos decrementar/incrementar el semáforo (-1/1).
- `short` **`sem_flg`** son los flags que afectan a la operación. Nosotros pondremos 0.

Si al realizar esta operación el semáforo se vuelve negativo, el proceso se quedará "bloqueado" hasta que algún otro proceso lo incremente y lo ponga como mínimo en 0.

Cuando el proceso termine de utilizar el recurso compartido, este debe de incrementar el valor del semáforo.

Para trabajar con semáforos es necesario que en nuestros programas incluyamos las siguientes librerías:

```
sys/ipc.h
sys/sem.h
```

Algunas funciones muy útiles

Estas funciones *wrapper* alrededor de los semáforos implementan semáforos binarios.

```
/*
**Crea un nuevo grupo de semáforos
**
**n, el número de semáforos
**vals, los valores por defecto para inicializar
**retorna el id del grupo de semáforos
*/
int CrearSemaforos(int n, short* vals) {
    union semun arg;
    int id;
    id=semget(IPC_PRIVATE, n, SHM_R | SHM_W);
    arg.array=vals;
    semctl(id, 0, SETALL, arg);
    return id;
}

/*
**Libera el grupo de semáforos indicado
**
**id, el id del grupo de semáforos
*/
void BorrarSemaforos(int id) {
    if(semctl(id, 0, IPC_RMID, NULL)==-1) {
    perror("Error liberando semáforo!");
    exit(EXIT_FAILURE);
    }
}
/*
```

```
**Bloquea a un semáforo dentro de un grupo
**
**id, id del grupo de semáforos al que pertenece
**i, semáforo a bloquear
**si ya está bloqueado, duermes
*/
void BloquearSemaforo(int id, int i) {
struct sembuf sb;
sb.sem_num = i;
sb.sem_op=-1;
sb.sem_flg=SEM_UNDO;
semop(id,&sb,1);
}

/*
**Desbloquea a un semáforo dentro de un grupo
**
**id, id del grupo de semáforos al que pertenece
**i, semáforo a desbloquear
*/
void DesbloquearSemaforo(int id, int i) {
    struct sembuf sb;
    sb.sem_num = i;
    sb.sem_op = 1;
    sb.sem_flg = SEM_UNDO;
    semop(id, &sb, 1);
}
```

Memoria compartida

Es un mecanismo de IPC nativo a UNIX. En resumen, dos procesos comparten un segmento de memoria sobre el cual pueden leer y escribir para comunicarse entre sí. Nada más ni nada menos, solo un pedazo de memoria. Una vez que sabemos dónde se encuentra, es como si fuera parte del espacio de memoria de nuestro proceso. Como es solo memoria, es el mecanismo de IPC más rápido que existe. Requiere de las siguientes bibliotecas: *#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>*. Un pequeño problema Existe un problema con la memoria compartida: condiciones de carrera, por lo que se requiere poder indicar a los procesos que la acceden si es seguro leer o escribir sobre sus datos. Una manera de resolver esto es mediante semáforos.

Algunas funciones usadas en memoria compartida.

shmget() también reserva un segmento de memoria compartida.

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

key es la llave asociada con el segmento de memoria compartida. Es mejor pasarle simplemente IPC_PRIVATE. *size* es el tamaño en bytes del segmento de memoria compartida a reservar. La memoria se reserva en páginas, así que es probable que recibamos un poco más de memoria de la que pedimos. *flags* indican como queremos que sea creado el segmento y sus permisos. La regla general es usar IPC_CREAT | SHM_W | SHM_R. El valor de retorno es el *id* de nuestro nuevo segmento reservado.

Ahora lo que falta es asociarlo a nuestro espacio de direcciones, usando *shmat*().

shmat() Mapea o asocia un segmento de memoria compartida al espacio de direcciones de nuestro proceso.

```
void* shmat(int id, const void* addr, int flags);
```

id es el id retornado por *shmget*() del segmento que queremos asociar.
addr es la dirección en donde queremos que sea mapeado el segmento. Es mejor pasarle NULL para que el sistema elija una dirección apropiada. La única vez que especificaremos esto es cuando tengamos un pedazo de datos específicos que deseemos compartir. *flags* especifica varios detalles misteriosos sobre cómo debería mapearse. Basta con pasarle 0. Lo que retorna es un puntero al segmento donde ha sido mapeado. Ahora podemos tratarlo como si fuera cualquier porción de memoria.

shmdt() Desasocia el segmento dado. Debemos hacer esto cuando ya no usemos la memoria compartida, p.ej. cuando el programa termina.

```
int shmdt(const void* addr);
```

addr es la dirección donde está mapeado el segmento de memoria a desasociar.

shmctl() Nos deja manipular información sobre el segmento compartido.

```
int shmctl(int id, int cmd, struct shmid_ds* buf);
```

id es el id del segmento retornado por shmget().
cmd es el comando que queremos realizar. Hay tres: IPC_STAT, IPC_SET, e IPC_RMID.
buf apunta a un buffer usado por IPC_STAT e IPC_SET. Cuando *cmd* está fijado a IPC_RMID, debería retornar NULL.

IPC_STAT & IPC_SET Estos nos permitan obtener y asignar información sobre el segmento compartido. Si quiere saber más, escriba *man shmctl*

IPC_RMID Esto sirve para marcar al segmento como destruido. En realidad será destruido una vez que el segmento haya sido desasociado, es decir, cuando todos los procesos que usan al segmento ya no existan. Esto debería utilizarse una sola vez, después de asociar el segmento con shmat().

Algunos puntos a recordar Después de un fork(),el hijo hereda todos los segmentos asociado. Después de un exec() o un exit() los segmentos son desasociados, pero no destruidos. Por eso usamos IPC_RMID.

Algunas funciones muy útiles
A continuación, veremos unas funciones *wrapper* que nos ayudan a facilitar el manejo de la memoria compartida. Las usaremos en el ejercicio del final.

```
/*
**Reserva un segmento de memoria compartida
**
**n es el tamaño (en bytes) del pedazo a reservar
**retorna el id del pedazo de memoria compartida.
*/
int ReservarMemoriaComp(int n) {
    return shmget(IPC_PRIVATE, n, IPC_CREAT | SHM_R | SHM_W);
```

```
}
/*
**Mapea un segmento de memoria compartida a nuestro
**espacio de memoria
**
**id bloque de memoria compartida a mapear
**retorna la direccion del bloque mapeado
*/
void* MapearMemoriaComp(int id) {
    void* addr;

    addr = shmat(id, NULL, 0); //adjuntamos el segmento
    shmctl(id, IPC_RMID, NULL); //y lo marcamos para que se destruya
    return addr;
}
```

IV ACTIVIDADES

01. Se tienen dos programas *semáforo_1.c* y *semáforo_2.c*. Cada uno de estos programas debe ser ejecutado en un terminal diferente para ver como sus ejecuciones son sincronizadas a partir del uso de un semáforo.

```
/*semaforo_1.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
        struct seminfo *_buf;
};

main() {
        key_t clave;
        int idSemaforo;
        struct sembuf operacion;
        union semun arg;
        int i=0;
        clave = ftok("/bin/ls",33);
        if (clave == (key_t)-1) {
            printf("No puedo conseguir la clave de semaforo\n");
            exit(0);
        }
        idSemaforo=semget(clave,10,0600|IPC_CREATE);
        if (idSemaforo == -1) {
            printf("No puedo crear el semaforo\n");
            exit(0);
        }
        arg.val = 0;
        semctl(idSemaforo,0,SETVAL,&arg);
        operacion.sem_num = 0;
        operacion.sem_op = -1;
        operacion.sem_flg = 0;
```

```
        while(1) {
            printf("Esperando semaforo %d\n",i);
            semop(idSemaforo,&operacion,1);
            printf("Salgo del semaforo %d\n",i);
            i++;
        }
}
```

```
/*semaforo_2.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
union semun {
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
        struct seminfo *_buf;
};

main() {
        key_t clave;
        int idSemaforo;
        struct sembuf operacion;
        union semun arg;
        int i=0;
        clave = ftok("/bin/ls",33);
        if (clave == (key_t)-1) {
            printf("No puedo conseguir la clave de semaforo\n");
            exit(0);
        }
        idSemaforo=semget(clave,10,0600|IPC_CREATE);
        if (idSemaforo == -1) {
            printf("No puedo crear el semaforo\n");
            exit(0);
        }
        operacion.sem_num = 0;
        operacion.sem_op = 1;
        operacion.sem_flg = 0;
        for(i=0; i<10; i++) {
            printf("Levanto semaforo\n");
            semop(idSemaforo,&operacion,1);
            sleep(1);
        }
}
```

02. Compile de forma separada los dos programas, y lance los procesos en dos terminales diferentes. Observe la ejecución.
03. En vista que la función *semop*() es bastante "engorrosa" debido a que se requiere asignar valores a la estructura, modifique los programas semáforo_1.c y semáforo_2.c, de forma que se adapte el uso de las funciones *wrapper* referente al uso de semáforos.
04. Un ejemplo: CrapChat

CrapChat es una aplicación de chat muy simple y no muy útil, ya que los usuarios que chatean tienen que estar en la misma computadora, además que deben tomarse turnos para conversar. Sin embargo, es muy útil para ilustrar IPC usando semáforos binarios y memoria compartida.
Copie el código de abajo (no se olvide incluir las funciones de arriba) y compile el programa. Para ejecutarlo:
`$./crapchat_v2`
Eso iniciará al proceso como el jugador 1 y le imprimirá el id de segmento, p.ej. '0'. Entonces para iniciar como el jugador 2 le pasamos dicho id como argumento al programa (en otra terminal):
`$./crapchat_v2 0`
Y con eso deberíamos poder chatear. Tipee `\quit` para salir.

```
/*
* crapchat_v2.c
* CrapChat original hecho por Keith Gaughan <kmgaughan@eircom.net>
* Adaptación y traducción al español por Gustavo Puma
*
* Software libre bajo la licencia DSL:
* http://www.dsl.org/copyleft/dsl.txt
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <syslog.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

//declaraciones de las funciones wrappers para simplificarnos la vida
int ReservarMemoriaComp(int n);
void* MapearMemoriaComp(int n);
int CrearSemaforos(int n, short* vals);
void BorrarSemaforos(int id);
void BloquearSemaforo(int id, int i);
void DesbloquearSemaforo(int id, int i);

enum {
   SEM_USER_1, //el turno de la primera persona
   SEM_USER_2 //el turno de la 2da persona
};

int main(int argc, char* argv[]) {
   int idShMem; //handle a la memoria compartida
   int idSem; //handle al grupo de semáforos
   char* buf; //dirección del buffer de memoria compartida

   short vals[2]; //valores para inicializar los semáforos
   int miSem; //semáforo para nuestro usuario
   int tuSem; //semáforo para el otro usuario

   puts("Bienvenido a CrapChat! Escribe '\\quit' para salir\n");

   //obtenemos el id del segmento de memoria de la línea de comandos
   if(argc<2) {
```

```
      //no nos pasaron ningún id, así que lo creamos
      idShMem = ReservarMemoriaComp(BUFSIZ);
      buf = (char*) MapearMemoriaComp(idShMem);

      //Queremos que los usuarios sean bloqueados cuando traten de
      //hacer lock a la memoria compartida. Cuando el segundo usuario
      //inicia el programa, desbloquearoan al primero para que pueda tipear.
      //Para eso son los ceros.

      vals[SEM_USER_1]=0;
      vals[SEM_USER_2]=0;

      idSem=CrearSemaforos(2,vals);
      //guardamos el id del semáforo en la memoria compartida para que
      //el otro usuario pueda obtenerlo
      *((int*) buf) =idSem;
      miSem = SEM_USER_1;
      tuSem = SEM_USER_2;

      //escribimos el id del segmento de memoria compartida para que el otro
      //usuario sepa y pueda chatear con nosotros.
      printf("Eres el usuario uno. El id de la memoria compartida es: %d\n",
            idShMem);
      printf("Esperando al usuario dos...\n");
   }
   else {
      //tenemos un valor! quiere decir que somos el usuario dos.
      idShMem = atoi(argv[1]);
      buf = (char*) MapearMemoriaComp(idShMem);

      //obtenemos el id del grupo de semaforos desde la memoria compartida
      idSem = *((int*) buf);

      //grabamos a qué semáforos tenemos que esperar
      //recordemos que aquí los roles cambian
      miSem=SEM_USER_2;
      tuSem=SEM_USER_1;

      //Ponemos una cadena vacía en la memoria compartida.
      sprintf(buf, "");

      //desbloqueamos al otro usuario para indicar que pueden hablar
      puts("Eres el usuario dos. Avisando al usuario uno...");
      DesbloquearSemaforo(idSem, tuSem);
   }

   for(;;) {
      //Esperar hasta que sea mi turno de hablar
      BloquearSemaforo(idSem, miSem);
      //salió el otro usuario?
      if(strcmp(buf, "\\quit\n")==0)
         break;

      //imprimir la respuesta, si hay alguna
      if(strlen(buf)>0)
         printf("Respuesta: %s", buf);

      //obtener tu respuesta

      printf("> ");
      fgets(buf, BUFSIZ, stdin);
```

```
      //Pasamos el control al otro usuario.
      DesbloquearSemaforo(idSem, tuSem);

      //quieres salir?
      if(strcmp(buf, "\\quit\n") == 0)
         break;
   }
   //el primer usuario tiene que liberar los semáforos.
   if(miSem == SEM_USER_1)
      BorrarSemaforos(idSem);
}
```

V
EJERCICIOS

01. Resuelva el ejercicio 1 de la sesión anterior, pero utilizando procesos pesado.
02. **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 terecro 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 con el uso de procesos pesados y semáforos.

    Muestre los mensajes que considere necesarios de forma que se evidencie la ejecución concurrente de los procesos de acuerdo a la especificación anterior.
03. Modifique el ejercicio 4 de las actividades (chat) para que sea un juego de tres en raya. Cada jugador escribe la posición en la cual va a dejar su marca, p.ej. ‘f1c1’ sería fila 1 columna 1, y el programa se encarga de mostrar a ambos jugadores como quedaría el tablero. El programa debe detectar cuando un jugador gana, indicar quién ganó y salir. El programa también debe detectar cuando existe un empate. 2.

    (OPCIONAL) Permita volver a jugar una vez que terminó una partida.

VI
INFORME A PRESENTAR

01. Se debe entregar un informe únicamente en formato PDF con la descripción del proceso realizado de manera detallada. Se recomienda realizar capturas de pantalla describiendo los resultados obtenidos. También se deben incluir conclusiones del aprendizaje de la práctica.

02. También se debe entregar los archivos con el código fuente de los programas elaborados en lenguaje C. Se recomienda realizar comentarios en el código para documentar el programa y colocar el nombre de los integrantes del grupo al principio del archivo de código fuente (como comentario).

VII
REFERENCIAS

- Gaughan, K. (2003). Shared Memory and Semaphores.
- Mitchell, M., Oldham, J., & Samuel, A. (2001). Processes. Advanced linux programming. (pp. 45-60) New Riders Publishing.