Universidad Nacional ARTURO JAURETCHE

Ingeniería Informática

Sistemas Operativos II 2015

Práctica 4

# Sincronización entre procesos

1) En todas las computadoras actuales, al menos una parte de los manejadores de interrupción se escriben en lenguaje ensamblador ¿Por qué?

2) Se tiene el siguiente código que muestra una aproximación de Dekker y la solución de Peterson al problema de la sección critica para dos competidores. Se pretende experimentar con los valores de tiempo de "sección crítica y no crítica", para eliminar la aleatoriedad del mismo, a fin de tratar de obtener, para las aproximaciones de Dekker, estados indeseables de interbloqueo.

```
// Solucion de Peterson al problema de la sección crítica
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int mem_id_flags,mem_id_turn;
int *p_flags,*p_turn;
int id;

int main() {
	int hijo;
	// Crea el área de memoria compartida para las banderas y el turno
	if((mem_id_flags=shmget(IPC_PRIVATE,sizeof(int)*2,0774|IPC_CREAT))==-1)
	return -1;
	if(!(p_flags=(int *)shmat(mem_id_flags,(char *)0,0))) return -1;
	if((mem_id_turn=shmget(IPC_PRIVATE,sizeof(int),0774|IPC_CREAT))==-1)
	return -1;
	if(!(p_turn=(int *)shmat(mem_id_flags,(char *)0,0))) return -1;

	*p_turn=0;
	p_flags[0]=p_flags[1]=0;

	if(hijo=fork())
		id=0;
	else
		id=1;

	while(1) {
		int other;
		printf("Proceso %d desea entrar a la SC\n",id);
		other=1-id;
		p_flags[id]=1;
		*p_turn=other;
		while(p_flags[other]==1 && *p_turn==other); /* Se usa espera
		                                               activa */
		printf("Proceso %d entró a la SC\n",id);
		sleep(1);

		printf("Proceso %d salió de la SC\n",id);
		p_flags[other]=0;
		sleep(1);
	}
	if(hijo) kill(hijo,9);
	shmdt((char *)p_flags);
	shmctl(mem_id_flags,IPC_RMID,(struct shmid_ds *)NULL);
	shmdt((char *)p_turn);
	shmctl(mem_id_turn,IPC_RMID,(struct shmid_ds *)NULL);
```

```
        return 0;
}
```

3) Investigue y describa en que consiste el algoritmo de la panadería de Lamport.

4) Escriba el algoritmo “Productor-Consumidor” para:
   a. Semáforos
   b. Monitores

5) Demostrar que los monitores y los semáforos tienen una funcionalidad equivalente:
   a. Implemente un monitor por medio de semáforos.
   b. Implemente un semáforo por medio de monitores.

6) Si un sistema sólo tiene dos procesos ¿tiene sentido utilizar una barrera para sincronizarlo? ¿Por qué sí o por qué no?

7) Pruebe los siguientes dos programas escritos en C una maquina Linux:

   a. sem1: tiene un bucle infinito para entrar en el semáforo. Escribe en pantalla cuando entra y cuando sale. Como el semáforo está en "rojo" por defecto, **sem1** entra en él y no sale hasta que **sem2** lo indica.
   b. sem2: tiene un bucle de 1 a 10 en el que pone en verde el semáforo y espera un segundo. El resultado es que **sem1** entra en el semáforo, queda "bloqueado" un segundo y sale del semáforo para volver a entrar en él y repetir el proceso 10 veces.

```
/*--------------------------------CODIGO DE SEM1 ---------------------------------*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>

//
// Esta union hay que definirla o no según el valor de los DEFINE aqui
// indicados.
//
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
// La union ya está definida en sys/sem.h
#else
// Tenemos que definir la union
union semun
{
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
        struct seminfo *__buf;
};
#endif


main()
```

```
{
        key_t Clave;
        int Id_Semaforo;
        struct sembuf Operacion;
        union semun arg;
        int i=0;

    //
    // Igual que en cualquier recurso compartido (memoria compartida, semaforos
    // o colas) se obtien una clave a partir de un fichero existente cualquiera
    // y de un entero cualquiera. Todos los procesos que quieran compartir este
    // semaforo, deben usar el mismo fichero y el mismo entero.
    //
        Clave = ftok ("/bin/ls", 33);
        if (Clave == (key_t)-1)
        {
                printf("No puedo conseguir clave de semáforo");
                exit(0);
        }
        printf("Se consiguio la clave del semaforo\n");
        //
        // Se obtiene un array de semaforos (10 en este caso, aunque solo se usara
        // uno.
        // El IPC_CREAT indica que lo  cree si no lo está ya
        // el 0600 con permisos de lectura y escritura para el usuario que lance
        // los procesos. Es importante el 0 delante para que se interprete en
        // octal.
        //
        Id_Semaforo = semget (Clave, 10, 0600 | IPC_CREAT);
        if (Id_Semaforo == -1)
        {
                printf("No puedo crear semáforo");
                exit (0);
        }
        printf("Se creo el semaforo\n");
        //
        //      Se inicializa el semáforo con un valor conocido. Si lo ponemos a 0,
        //      es semáforo estará "rojo". Si lo ponemos a 1, estará "verde".
        //      El 0 de la función semctl es el índice del semáforo que queremos
        //      inicializar dentro del array de 10 que hemos pedido.
        //
        arg.val = 0;
        semctl (Id_Semaforo, 0, SETVAL, &arg);

        //
        //      Para "pasar" por el semáforo parándonos si está "rojo", debemos rellenar
        //      esta estructura.
        //      sem_num es el indice del semáforo en el array por el que queremos "pasar"
        //      sem_op es -1 para hacer que el proceso espere al semáforo.
        //      sem_flg son flags de operación. De momento nos vale un 0.
        //
        Operacion.sem_num = 0;
        Operacion.sem_op = -1;
        Operacion.sem_flg = 0;

        //
        //      Bucle infinito indicando cuando entramos al semáforo y cuándo salimos
        //      de él.
        //      i hace de contador del número de veces que hemos salido del semáforo.
        //
        while (1)
        {
                printf(" %d Esperando Semaforo\n",i);
                //
                //      Se hace la espera en el semáforo. Se le pasa un array de
operaciones
                //      y el número de elementos en dicho array. En nuestro caso solo 1.
                //
                semop (Id_Semaforo, &Operacion, 1);
                printf("%d Salgo de Semaforo\n ",i);
                i++;
        }
```

```
}


/*---------------------------------CODIGO DE SEM2---------------------------------*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
#include <unistd.h>


//
//      Esta union hay que definirla o no según el valor de los defines aqui
//      indicados.
//
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
// La union ya está definida en sys/sem.h
#else
// Tenemos que definir la union
union semun
{
        int val;
        struct semid_ds *buf;
        unsigned short int *array;
        struct seminfo *__buf;
};
#endif

main()
{
        key_t Clave;
        int Id_Semaforo;
        struct sembuf Operacion;
        union semun arg;
        int i;

        //
        //      Igual que en cualquier recurso compartido (memoria compartida, semaforos
        //      o colas) se obtien una clave a partir de un fichero existente cualquiera
        //      y de un entero cualquiera. Todos los procesos que quieran compartir este
        // semaforo, deben usar el mismo fichero y el mismo entero.
        //
        Clave = ftok ("/bin/ls", 33);
        if (Clave == (key_t)-1)
        {
                printf("No puedo conseguir clave de semáforo\n");
                exit(0);
        }

        //
        //      Se obtiene un array de semaforos (10 en este caso, aunque solo se usara
        //      uno.
        //      El IPC_CREAT indica que lo  cree si no lo está ya
        //      el 0600 con permisos de lectura y escritura para el usuario que lance
        //      los procesos. Es importante el 0 delante para que se interprete en
        //      octal.
        //
        Id_Semaforo = semget (Clave, 10, 0600 | IPC_CREAT);
        if (Id_Semaforo == -1)
        {
                printf("No puedo crear semáforo");
                exit (0);
        }

        //
        //      Se levanta el semáforo. Para ello se prepara una estructura en la que
        //      sem_num indica el indice del semaforo que queremos levantar en el array
        //      de semaforos obtenido.
        //      El 1 indica que se levanta el semaforo
        //      El sem_flg son banderas para operaciones raras. Con un 0 vale.
```

```
        //
        Operacion.sem_num = 0;
        Operacion.sem_op = 1;
        Operacion.sem_flg = 0;

        //
        //      Vamos a levantar el semáforo 10 veces esperando 1 segundo cada vez.
        //
        for (i = 0; i<10; i++)
        {
                printf("Levanto Semaforo");
                //
                // Se realiza la operación de levantar el semáforo. Se pasa un array
                // de Operacion y el número de elementos en el array de Operacion. En
                // nuestro caso solo 1.
                //
                semop (Id_Semaforo, &Operacion, 1);
                sleep (1);
        }
}
```

8) En función al ejercicio anterior escribir 3 procesos nuevos. Dos que ejecuten loops infinitos, que impriman en pantalla “proceso 1” y “proceso 2” respectivamente (cada 1 segundo). Previamente a entrar en el bucle infinito, los dos procesos se deben detener con semáforos. El tercer proceso, esperará hasta que lleguen los dos procesos anteriores para ejecutarlos juntos, liberando los semáforos de los procesos 1 y 2.
*Sugerencia:* Para ejecutarlo, escriba un script como el siguiente: semaforos.sh

```
#!/bin/bash
./proceso1 &
./proceso2 &
./proceso3 &
```