



UNIVERSIDAD
NEBRIJA

Grado en Ingeniería Informática

Sistemas Operativos

Práctica 3

Índice

Práctica 3: Comunicación y Sincronización de Procesos.....	3
Introducción	3
Tuberías (Pipes): mecanismo de comunicación y sincronización	3
Ejemplo integral de pipes.....	4
Semáforos: mecanismo de sincronización.	6
Ejemplo sección crítica con semáforos	7
Entregables:.....	8

Práctica 3: Comunicación y Sincronización de Procesos

Introducción

Una vez estudiada la concurrencia de los procesos, muchos de ellos necesitan en algún momento de su ejecución, comunicarse entre sí. Por ejemplo, para leer de un fichero, los procesos de usuario deben decir al proceso que gestiona los ficheros lo que desean. Este, a su vez, debe pedirle al proceso de disco que lea los bloques necesarios. Cuando el shell conecta dos procesos con un tubo o pipe, los datos de salida del primer proceso se transfieren al segundo. Entre los mecanismos de comunicación encontramos:

ficheros, pipes, variables de memoria compartida, paso de mensajes y sockets.

Además, la velocidad de un proceso con respecto a otro es impredecible ya que depende de la frecuencia de la interrupción asociada a cada uno de ellos y cuán a menudo y de por cuánto tiempo tiene asignado cada proceso un procesador (planificador/scheduler). Diremos, que un proceso se ejecuta asincrónicamente con respecto a otro, es decir, sus ejecuciones son independientes. Sin embargo, hay ciertos instantes en lo que los procesos deben sincronizar sus actividades. Son estos puntos a partir de los cuales un proceso no puede progresar hasta que otro haya completado algún tipo de actividad para lo que el sistema operativo nos ofrece una serie de servicios de sincronización como: *pipes, semáforos, mutex (procesos ligeros), variables condicionales y señales.*

Tuberías (Pipes): mecanismo de comunicación y sincronización

La utilización de tuberías mediante el uso de la shell es una tarea muy común que encadena comandos mediante tuberías de forma natural:

```
cat /etc/passwd | grep bash | wc -lines
```

Los comandos “cat”, “grep” y “wc” se lanzan en paralelo y el primero alimenta al segundo, que posteriormente alimenta al tercero. Al final tenemos una salida filtrada por esas dos tuberías. Las tuberías empleadas son destruidas al terminar los procesos que las estaban utilizando.

Utilizar tuberías en C es también bastante sencillo. Una tubería tiene dos descriptores de fichero: uno para el extremo de escritura y otro para el extremo de lectura. Como los descriptores de fichero de UNIX son simplemente enteros, un pipe o tubería no es más que un array de dos enteros:

```
int tuberia[2];
```

Para crear la tubería se emplea la función `pipe()`, que abre dos descriptores de fichero y almacena su valor en los dos enteros que contiene el array de descriptores de fichero. El primer descriptor de fichero es abierto como `O_RDONLY`, es decir, sólo puede ser empleado para lecturas. El segundo se abre como `O_WRONLY`, limitando su uso a la escritura. De esta manera se asegura que el pipe sea de un solo sentido: por un extremo se escribe y por el otro se lee, pero nunca al revés.

```
int tuberia[2];  
pipe(tuberia);
```

Una vez creado un pipe, se podrán hacer lecturas y escrituras de manera normal, como si se tratase de cualquier fichero. Sin embargo, no tiene demasiado sentido usar un pipe para uso propio, sino que se suelen utilizar para intercambiar datos con otros procesos. **Como ya sabemos, un proceso hijo hereda todos los descriptores de ficheros abiertos de su padre, por lo que la comunicación entre el proceso padre y el proceso hijo es bastante cómoda mediante una tubería.** Para asegurar la unidireccionalidad de la tubería, es necesario que tanto padre como hijo cierren los respectivos descriptores de ficheros.

Ejemplo integral de pipes

A continuación se muestra un ejemplo que integra la gestión de pipes donde un **proceso padre y su hijo comparten datos mediante una tubería.**

```
1  #include <sys/types.h>
2  #include <fcntl.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  #define SIZE 512
7
8  int main( int argc, char **argv )
9  {
10     pid_t pid;
11     int p[2], readbytes;
12     char buffer[SIZE];
13
14     pipe( p );
15
16     if ( (pid=fork()) == 0 )
17     { // hijo
18         close( p[1] ); /* cerramos el lado de escritura del pipe */
19
20         while( (readbytes=read( p[0], buffer, SIZE )) > 0 )
21             write( 1, buffer, readbytes );
22
23         close( p[0] );
24     }
25     else
26     { // padre
27         close( p[0] ); /* cerramos el lado de lectura del pipe */
28
29         strcpy( buffer, "Esto llega a traves de la tubería\n" );
30         write( p[1], buffer, strlen( buffer ) );
31
32         close( p[1] );
33     }
34     waitpid( pid, NULL, 0 );
35     exit( 0 );
36 }
```

Si se precisa una comunicación “full-duplex”, será necesario crear dos tuberías para ello. Dicha comunicación bidireccional entre dos procesos necesita (a[2] y b[2]), una para cada sentido de la comunicación. En cada proceso habrá que cerrar descriptores de ficheros diferentes. Vamos a emplear:

1. el pipe a[2] para la comunicación desde el padre al hijo
2. el pipe b[2] para comunicarnos desde el hijo al padre.

Por lo tanto, deberemos cerrar:

- En el padre:
 - el lado de lectura de a[2].
 - el lado de escritura de b[2].
- En el hijo:
 - el lado de escritura de a[2].
 - el lado de lectura de b[2].

A continuación se muestra un ejemplo para comunicación full-duplex.

Dos procesos se comunican bidireccionalmente con dos tuberías:

```
1  #include <sys/types.h>
2  #include <fcntl.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6
7  #define SIZE 512
8
9  int main( int argc, char **argv )
10 {
11     pid_t pid;
12     int a[2], b[2], readbytes;
13     char buffer[SIZE];
14
15     pipe( a );
16     pipe( b );
17
18     if ( (pid=fork()) == 0 )
19     { // hijo
20         close( a[1] ); /* cerramos el lado de escritura del pipe */
21         close( b[0] ); /* cerramos el lado de lectura del pipe */
22
23         while( (readbytes=read( a[0], buffer, SIZE ) ) > 0 )
24             write( 1, buffer, readbytes );
25         close( a[0] );
26
27         strcpy( buffer, "Soy tu hijo hablandote por
28                 la otra tuberia.\n" );
29         write( b[1], buffer, strlen( buffer ) );
30         close( b[1] );
31     }
32     else
33     { // padre
34         close( a[0] ); /* cerramos el lado de lectura del pipe */
35         close( b[1] ); /* cerramos el lado de escritura del pipe */
36
37         strcpy( buffer, "Soy tu padre hablandote
38                 por una tuberia.\n" );
39         write( a[1], buffer, strlen( buffer ) );
40         close( a[1] );
41
42         while( (readbytes=read( b[0], buffer, SIZE ) ) > 0 )
43             write( 1, buffer, readbytes );
44         close( b[0] );
45     }
46     waitpid( pid, NULL, 0 );
47     exit( 0 );
48 }
```

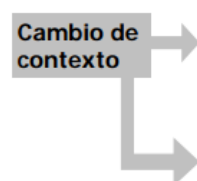
Semáforos: mecanismo de sincronización.

Los mecanismos de sincronización permiten forzar a un proceso/hilo a detener su ejecución hasta que ocurra un evento en otro proceso/hilo.

Si se supone que:

- Inicialmente **contador** vale 5
- Un productor ejecuta la instrucción **contador = contador + 1;**
- Un consumidor ejecuta la instrucción **contador = contador - 1;**
entonces, el resultado de **contador** debería ser otra vez 5

Pero, qué pasa si se produce un cambio de contexto en un momento "inoportuno":



t	Hilo	Operación	reg1	reg2	contador
0	productor	mov contador, reg1	5	?	5
1	productor	inc reg1	6	?	5
2	consumidor	mov contador, reg2	?	5	5
3	consumidor	dec reg2	?	4	5
4	consumidor	mov reg2, contador	?	4	4
5	productor	mov reg1, contador	6	?	6

¡Incorrecto!

Ejemplo de Condición de carrera.

- Posix define los semáforos a través de la variable de tipo **sem_t**.
- Los semáforos se pueden compartir entre procesos y pueden ser accedidos por parte de todos los hilos del proceso. Los semáforos se heredan de padre a hijo igual que otros recursos (como por ejemplo los descriptores de archivo).

```
int sem_init (sem_t *sem, int shared, int val);
```

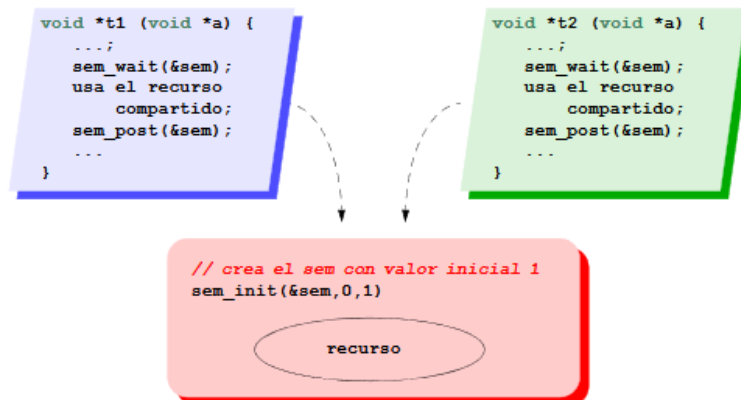
- Crea un semáforo identificado a través de sem y le asigna el valor inicial val.
- Si shared=0 → lo usarán hilos del proceso que lo inicializa. Un proceso hijo creado mediante fork() hereda el mapa del padre y por tanto también tendrá acceso al semáforo.
- Si shared≠0 → lo usarán procesos y por tanto deberá estar localizado en una región de memoria compartida (analizar shm_open(3), mmap(2), shmget(2)).

Las funciones que soporta un semáforo vienen definidas por:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);           /* Operación P(sem) */
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);           /* Operación V(sem) */
int sem_getvalue(sem_t *sem, int *sval);
```

Ejemplo sección crítica con semáforos



```
#include <pthread.h>  
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
#include <time.h>  
#include <semaphore.h>
```

```
int x=0;  
sem_t semaforo;
```

```
void *fhi1o1(void *arg)  
{ int i;  
  for (i=0; i<3; i++) {  
    sem_wait(&semaforo);  
    x=x+1;  
    sem_post(&semaforo);  
    printf ("Suma 1\n");  
    sleep (random()%3);  
  }  
  pthread exit (NULL);  
}
```

```
void *fhi1o2(void *arg)  
{ int i;  
  for (i=0; i<3; i++) {  
    sem_wait(&semaforo);  
    x=x-1;  
    sem_post(&semaforo);  
    printf ("Resta 1\n");  
    sleep (random()%3);  
  }  
  pthread exit (NULL);  
}
```

```
main()  
{ pthread t hilo1, hilo2;  
  time t;  
  srandom (time(&t));  
  printf ("Valor inicial de x: %d \n",x);  
  sem_init (&semaforo,0,1);  
  
  pthread create(&hilo1, NULL, fhi1o1, NULL);  
  pthread create(&hilo2, NULL, fhi1o2, NULL);  
  
  pthread join(hilo1,NULL);  
  pthread join(hilo2,NULL);  
  sem_destroy (&semaforo);  
  
  printf("Valor final de x: %d \n",x);  
  exit(0);  
}
```

Entregables:

1. Implementar un programa productor-consumidor mediante tuberías/pipes donde el proceso hijo genera datos y el proceso padre los consume.

2. El problema del productor-consumidor con buffer limitado (circular):
Planteamiento:

a. El proceso productor produce información y la almacena en un buffer.

b. El proceso consumidor accede al buffer y consume la información.

c. El productor y el consumidor comparten variables.

d. El productor no puede acceder al buffer si está lleno.

e. El consumidor no puede acceder al buffer si está vacío.

→ Acceso a SC

} → Sincronización

