DISEÑO Y ADMINISTRACIÓN DE SISTEMAS OPERATIVOS

SEGUNDA PED: COMUNICACIÓN Y SINCRONIZACIÓN DE PROCESOS EN UNIX

David Pastor Sanz

Índice

1. Introducción

Esta práctica consiste en la comunicación y sincronización de tres procesos mediante el paso de mensajes de distintas formas. Para la realización de la misma se han creado tres archivos con código fuente en lenguaje C, tal como lo pide el enunciado. Mediante el compilador de C **gcc** se generan los tres ejecutables y se hace correr el primero, una vez finalizado el programa se borran estos tres. Estas acciones de generar, ejecutar y borrar se realizan mediante un **script bash**, cuyo código se puede ver en la sección 4.1 de este documento. A este script se le han dado los permisos pertinentes para poder ser ejecutado.

Se han seguido al pie de la letra las indicaciones del enunciado de la práctica, siguiendo las pautas para poder realizar una correcta implementación a partir de los ejemplos y la teoría estudiados en el libro de texto base de la asignatura.

El funcionamiento principal del programa se basa en la introducción de un mensaje por parte del usuario, mediante la entrada estándar, y el paso de este a otros procesos (que se crean en la ejecución de distintos puntos de los ejecutables) mediante distintas formas, como son el uso de **tuberías sin nombre**, **ficheros FIFO**, **colas de mensajes** y el uso de **memoria compartida**; y sincronización de estos, con **semáforos** y las esperas de los mensajes que producen algunas de las formas de compartición de mensajes. Tras el envío y la recepción del mensaje de las diversas maneras expuestas se muestra el mensaje por la salida estándar.

Cada vez que se crea y o destruye un proceso, se envía o recibe un mensaje, se ejecuta un bucle para tomar tiempos acordes para las estadísticas o se ejecuta **Ej2** o **Ej3** mostrará un mensaje por pantalla indicando quien es el proceso que lo realiza y en qué ejecutable. También se muestra un mensaje cuando se realizan "tomas" de datos para las estadísticas (se explica en la sección 4) del tiempo de uso de la CPU del programa que se mostrarán al final.

Para poder ejecutar el programa completo se abrirá un terminal y desde el directorio en el que se encuentra este documento se introducirá el siguiente comando:

```
$ ./Ejercicio.sh
```

Y se podrá ver el resultado de la ejecución del programa programa tal como se ve en la Figura 3.

2. Implementación

Se va a explicar el funcionamiento del programa paso por paso. Se dividirá en la explicación de cada uno de los mecanismos que se utilizan, sus funcionamientos y la eliminación de los mismos al final. Cada uno de los mecanismos se explicarán con ejemplos de código.

2.1. Creación de procesos hijos

El primer paso que se pide en el ejecutable **Ej1** es crear un proceso hijo, **P2**, esto se realizará mediante el uso de `fork()` (se realizará la misma acción en el ejecutable**Ej2** para que **P2** cree a **P3**). Se realiza mediante la siguiente linea de código:

```
pid=fork();
```

Donde en `pid` almacenará el valor de -1 si no se ha creado el proceso con éxito. Si el proceso se ha creado correctamente almacenará el valor 0 cuando se esté ejecutando dicho proceso hijo (**P2**) y el pid de dicho proceso hijo cuando se esté ejecutando el proceso padre (**P1**).

2.2. Tubería sin nombre

En el ejecutable **Ej1**, después de la creación de **P2** y cuando se ejecuta **P1**, se crea una tubería sin nombre. Antes de crear la tubería se tiene que declarar un array de enteros de dos elementos (**[1]**), en la primera posición de dicho se almacenará el descriptor para leer de la tubería y en la segunda el de lectura. Después se crea la tubería mediante la sentencia **[2]**, si todo es correcto esta función devolverá el valor 0, en caso contrario devolverá -1.

```
[1] int tuberia[2];
[2] pipe(tuberia);
    ...
[3] write(tuberia[1], mensaje, tamanoMensaje);
[4] close(tuberia[1]);
    ...
[5] read(tuberia[0], mensajeLeer, tamanoALeer);
[6] close(tuberia[0]);
```

Después de que el usuario introduzca un mensaje por la entrada estándar, mediante la función `gets(mensaje)` (`mensaje` almacenará el texto introducido por el usuario) de la librería **stdio.h**, **P1** lo mandará por la tubería mediante la sentencia **[3]** (la función `write` de la librería `unistd.h` que toma como parámetros el descriptor de fichero, el mensaje a escribir y el tamaño del mismo). Seguidamente se cerrará el descriptor de fichero de escritura de la tubería mediante la sentencia **[4]**.

El proceso **P2** será el encargado de leer el mensaje de la tubería mediante la sentencia **[5]**, que contiene los mismos parámetros que `write`, solo que en este caso el segundo almacena el mensaje que contiene la tubería. Después de la lectura se cierra el descriptor de fichero de lectura de la tubería mediante **[6]**.

Cabe destacar, que al igual que los demás mecanismos de comunicación que se van a explicar, la operación de lectura es bloqueada hasta que no es escrito el mensaje.

2.3. Cola de mensajes

Después de crear la tubería sin nombre y enviar el mensaje por dicha, **P1** crea una cola de mensajes. Los mensajes que transportan estos mecanismos de IPC son una estructura que se componen de el tipo de mensaje a leer (para así poder identificar en caso de diferentes procesos emisores y receptores), que se asignará antes de mandarlo; y el mensaje (en este caso como el mensaje que **P1** debe recibir de **P3** es su pid, el tipo de mensaje será de `pid_t`). La declaración de esta estructura se ve en **[1]**.

Para poder crear la cola de mensajes se necesita una llave **[1]** (se usa para poder controlar el acceso a una instancia de un mecanismo comunicación de procesos, en este caso la cola de mensajes). Si la creación de la llave es correcta se procede a crear la cola de mensajes mediante la sentencia **[2]** que toma como parámetros la llave creada, la constante para la creación y los permisos que tendrá, si no hay ningún error `msgid` almacenará el identificador de la cola de mensajes, en caso contrario el valor de -1.

```
[1] struct
    {
        long tipoMensaje;
        pid_t pid;
    } mensaje;
    ...
[2] llave=ftok(ruta, letra);
[3] msqid=msgget(llave, IPC_CREAT | 0600);
    ...
[4] msgsnd(msqid, &mensaje, tamanoAEscribir, comportamiento);
    ...
[5] msgrcv(msqid, &mensaje, tamanoALeer, tipoMensaje, comportamiento);
    ...
[6] msgctl(msqid, IPC_RMID, 0);
```

P3 abrirá la cola de mensajes desde el ejecutable **Ej3** mediante la sentencia **[3]** pero sin la constante que indica la creación (solo con los permisos). Creará el mensaje y lo enviará mediante la sentencia **[4]**, la cual toma como parámetros el identificador de la cola, el mensaje, el tamaño del mismo y el comportamiento que tomará **P3** en caso de no poder enviar el mensaje (el cual en este caso será el valor de 0). Si el envío es correcto la función devolverá 0, en caso contrario -1.

El proceso **P1** recibirá el pid de **P3** por la cola de mensajes mediante la sentencia [5], es igual que el envío con la diferencia que esta acción toma un parámetro más: el tipo de mensaje a leer de la cola de mensajes.

Una vez **P1** ha leído el mensaje se puede proceder a borrar la cola de mensajes para que el núcleo pueda liberar el espacio que ocupa mediante la sentencia **[6]**.

2.4. Fichero FIFO

Una vez **P2** en el ejecutable **Ej1** lee el mensaje enviado por **P1** a través de la tubería sin nombre crea un fichero FIFO mediante la sentencia **[1]** que toma como parámetros la ruta en la que se creará el fichero, en este caso se creará en el mismo directorio donde se ejecuta el programa y tomará el nombre de `fichero1`.

Después se abrirá la tubería con nombre mediante la sentencia **[2]** que toma como parámetros la ruta

("fichero1") y los permisos (en este caso de lectura y escritura). Almacenará en la variable descriptor el descriptor del fichero FIFO, en caso de error tomará el valor de -1.

```
[1] mknod(ruta, modo, 0);
[2] descriptor=open(ruta, O_RDWR);
[3] write(descriptor, mensajeEscribir, tamanoMensaje);
    ...
[5] read(descriptor, mensajeLeer, tamanoALeer);
[6] close(descriptor);
    ...
[7] unlink(ruta);
```

Seguidamente **P2** escribe el mensaje mediante la sentencia 3, que es la misma que se utiliza para la tubería sin nombre. Cabe destacar que después de realizar esta acción **P2** ejecuta el ejecutable **Ej2** mediante el uso de la siguiente sentencia:

```
execv("./Trabajo2/Ej2", NULL);
```

Desde el ejecutable **Ej2** el proceso **P2** leerá el mensaje que había escrito en la tubería con nombre mediante la sentencia **[5]** y después se cierra el descriptor de fichero con la sentencia **[6]**, que se realizan de la misma como se realizaron para la tubería sin nombre.

Antes de finalizar la ejecución del programa el proceso **P1** desde el ejecutable **Ej1** borrará la tubería sin nombre con la sentencia **[7]**.

2.5. Memoria compartida

Una vez **P2** ha leído el mensaje de la tubería con nombre creará una zona de memoria compartida. Para ello primero se creará una llave (el fichero que se le asocia será el fichero FIFO fichero1) para poder asociar al mecanismo IPC. La zona de memoria compartida se asociará al espacio de direcciones virtuales del proceso mediante una variable puntero (en este caso como el mensaje es una cadena de texto su tipo será char) que indicará la dirección de inicio de la misma, **[1]**.

Se crea la zona de memoria compartida mediante la sentencia **[2]** que toma como parámetros la llave a la que se asocia , el tamaño de memoria a reservar y la constante de creación junto a los permisos y que devuelve el identificador de la zona de memoria compartida creada. Seguidamente se asigna un espacio de direcciones virtuales al segmento de memoria mediante la sentencia **[3]**, tomando como parámetros el identificador de la zona de memoria compartida, la dirección de inicio de dicha (la cual en este caso se toma desde el principio) y una mascara de bits que indica la forma de acceso a esta (en este caso el 0 indica que será tanto de escritura como lectura). El resultado de esta función será hacer que vc1 almacene la dirección de inicio de la zona de memoria compartida.

Para escribir en esta zona de memoria compartida simplemente se asigna el valor correspondiente a vc1, en este caso se ha copiado el mensaje mediante la función que se muestra en la sentencia **[4]** de la librería **string.h**. Seguidamente **P2** realizará la operación V() del semáforo (el cual se explicará en la siguiente sección) para que **P3** pueda acceder a dicha zona.

Desde el ejecutable **Ej3**, el proceso **P3** carga la zona de memoria compartida con la sentencia **[2]** (pero sin el parámetro que indica la constante de creación) y seguidamente se une esta, de igual manera

que como se ha hecho en el ejecutable **Ej2**, al espacio de direcciones virtuales del proceso mediante la sentencia **[3]**. De esta manera **P3** puede acceder al mensaje que había dejado **P2**.

```
[1] char *vc1;
    ...
[2] shmid=shmget(llave, tamanoMC, IPC_CREAT | 0600);
[3] vc1=shmat(shmid, 0, 0);
    ...
[4] strcpy(vc1, mensaje);
    ...
[5] shmdt((int *)vc1);
[6] shmctl(shmid, IPC_RMID, 0);
```

Para finalizar se desenlaza la zona de memoria compartida del proceso **P3** indicando la dirección memoria virtual de comienzo de esta (al ser un puntero de caracteres se hace un cast a entero para que no indique ningún aviso el compilador) mediante la sentencia **[5]**. Seguidamente se borra el segmento de memoria compartida del sistema mediante la sentencia **[6]**.

2.6. Semáforo

El proceso **P2** desde el ejecutable **Ej2** es el encargado de crear un semáforo que protegerá el acceso a la zona de memoria compartida. Para crear el semáforo se utilizará la misma llave que se ha usado para crear la zona de memoria compartida. En la sentencia **[1]** se puede observar como se puede crear un conjunto de semáforos, esta función tomará como parámetros la llave a la que se asociará, el número de semáforos a crear (en este caso solo se necesita uno) y la constante de creación junto a los permisos necesarios. Devolverá el identificador del conjunto de semáforos.

Se inicia el semáforo con el valor de 0 con la sentencia **[2]** mediante el identificador del conjunto de semáforos, el número del semáforo dentro de este conjunto (como solo hay uno y se empieza a contar desde 0 este valor será 0), la constante que indica que se va a asignar un valor al semáforo y el valor que tomará.

```
[1] sem1=semget(llave, numeroSemaforos, IPC_CREAT | 0600);
[2] semctl(sem1, numSem, SETVAL, 0);
    ...
[3] struct sembuf opV;
    opV[0].sem_num=0;
    opV[0].sem_op=1;
    opV[0].sem_flg=0;
[4] struct sembuf opP
    ops[1].sem_num=0;
    ops[1].sem_op=-1;
    ops[1].sem_flg=0;
    ...
[5] semop(sem1, opV, 1);
    ...
[6] semop(sem1, opP, 1);
```

A continuación se crean las operaciones que realizará el semáforo, es decir `V()` y `P()`. Para ello se

creará una variable de estructura del tipo `sembuf` (este contiene los campos de número del semáforo sobre el que se realizará la operación, el incremento o decremento que se realizará y los indicadores de la llamada que en este caso no los habrá). Se crearán dos de estas estructuras una para la operación `V()` (sentencia **[3]**) en el ejecutable **Ej2** para que **P2** pueda permitir la lectura de la zona de memoria compartida a **P3** y la otra para la operación `P()` (sentencia **[4]**) en el ejecutable **Ej3** con la que **P3** esperará a que **P2** termine de escribir en la zona de memoria compartida.

Para realizar una operación sobre el semáforo se usarán la sentencias **[4]** y **[5]**. Esta función toma como parámetros el indentificador del conjunto de semáforos, la estructura `sembuf` que contiene la operación (u operaciones en otro caso) y el número de operaciones que contiene (una para este lance).

2.7. Eliminación de procesos hijos

Una vez **P1** recibe mediante la cola de mensajes el pid de **P3** procede a eliminar el fichero FIFO `fichero1`, tal como se ha indicado antes, se dispone a eliminar los procesos **P2** y **P3** mediante la siguiente sentencia:

```
kill(pid, SIGKILL);
```

Esta función toma como parámetros el pide del proceso a matar y la señal que se le envía, en este caso la que indica que se mate al proceso.

2.8. Toma de estadísticas de uso de la CPU

Después de matar a **P2** y **P3** y eliminar la tubería con nombre se muestran las estadísticas de uso de la CPU. Estas estadísticas son los tiempos de uso en modo núcleo y usuario del proceso padre **P1** y de los proceso hijo **P2** y su descendiente **P3**.

Para realizar esto se tomarán medidas de tiempos al principio y al final del programa. Estas medidas serán tomadas mediante a la llamada al sistema `times` cuya sintaxis es la siguiente:

```
tiempo=times(&tms);
```

Dicha función da como resultado el tiempo real transcurrido (en tics) a partir de un instante pasado arbitrario. Este instante puede ser el momento de arranque del sistema y no cambia de una llamada a otra. Como parámetro toma un puntero a una estructura de tipo `tms` que contiene cuatro campos: el tiempo (en tics) de uso de la CPU del proceso en modo usuario y modo núcleo, así lo mismo para los procesos hijos y descendientes. En estos tiempos no se cuenta cuando los procesos están dormidos o el tiempo de cambios de contexto. Para poder tomar los tiempos de los procesos hijos el proceso padre debe ejecutar una llamada al sistema `wait()`.

Como el programa se ejecuta de manera rápida se han introducido dos bucles (uno lo realiza **P1** y otro **P2**) que abren y cierran un fichero, mediante sendas llamadas al sistema, (el fichero `fuente1.c` que contiene el código fuente del ejecutable **Ej1**) 5 millones de veces para así poder tomar tiempos acordes.

Al final del programa se muestra el tiempo total del programa, los tiempos de uso en modo usuario y núcleo de **P1** y de su hijo **P2** y descendiente **P3**. Restando la medida tomada al principio del programa a la del final del total y los diferentes campos de la estructura `tms`. Además se muestra el porcentaje de uso de CPU a partir de los tiempo mostrados.

3. Ejecución de ejemplo

Mediante la siguiente figura se puede ver el correcto funcionamiento del programa, observando los distintos mensajes que muestra el programa se comprueba que realiza las acciones tal y como pide el enunciado de la práctica y de la manera que se han explicado en este documento.

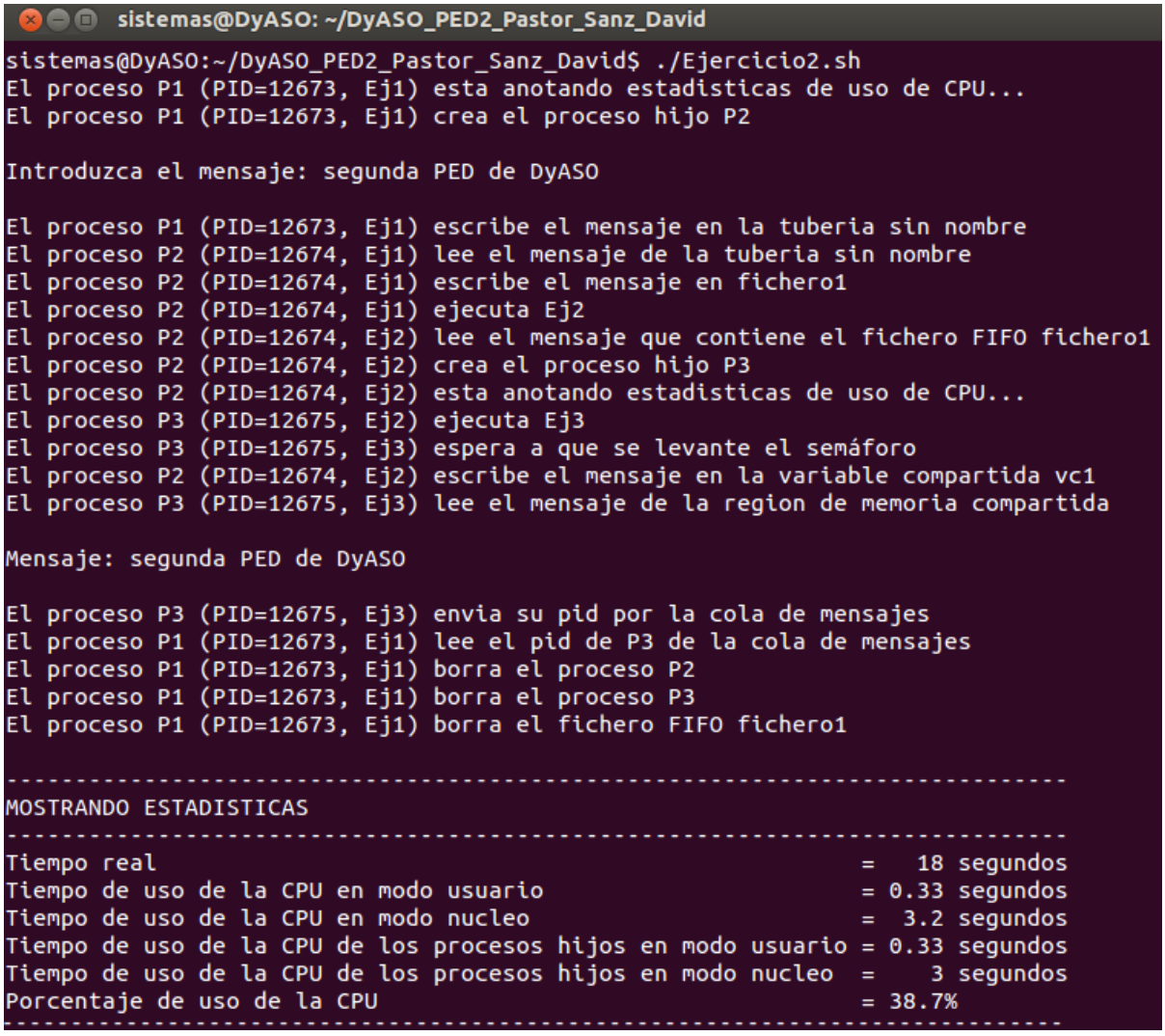

Figura 1: Ejemplo de ejecución del programa

4. Código fuente

4.1. Ejercicio2.sh

```
#!/bin/bash

#se compilan los tres archivos fuente
gcc ./Trabajo2/fuente1.c -o ./Trabajo2/Ej1
gcc ./Trabajo2/fuente2.c -o ./Trabajo2/Ej2
gcc ./Trabajo2/fuente3.c -o ./Trabajo2/Ej3

#se ejecuta Ej1
./Trabajo2/Ej1

#se borran los tres ejecutables
rm ./Trabajo2/Ej1
rm ./Trabajo2/Ej2
rm ./Trabajo2/Ej3
```

4.2. fuente1.c

```
//este archivo es el fichero fuente que al compilarse produce el ejecutable Ej1
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <time.h>
#include <sys/times.h>

#define MAX 256        //tamano maximo del mensaje

//imprime la tarea realizada por un proceso
void printTask(int process, char *task)
{
  printf("El proceso P%d (PID=%d, Ej1) %s\n", process, getpid(), task);
}

//imprime el error causado y termina el programa
void printError(char *error)
{
  perror(error);
  exit(-1);
}

int main()
{
  char mensaje[MAX];   //mensaje recibido por la entrada estandar
  char msgPipe[MAX];   //mensaje recibido por la tuberia sin nombre
  int pids[2];         //almacena los pids de P2 y P3
  int tuberia[2];      //descriptores de la tuberia sin nombre
  int f1;              //descriptor de fichero1
  int msqid;           //identificador de la cola de mensajes
  int CLK_TCK;         //tics por segundo
  int fd, h;           //para el calculo de estadisticas
  float tt, tu, tn, tcu, tcn;
```

```
41  key_t llave;          //llave para la cola de mensajes
42  clock_t t[2];         //contabilizan los ticks al inicio y al final
43  struct tms pb[2];     //tiempos de ejecucion al inicio y al final
44  struct
45  {
46    long tipo;
47    pid_t pid;
48  } msgQueue;           //almacena el mensaje y el tipo de una cola de mensajes
49  msgQueue.tipo=2;      //se asigna el tipo 2
50
51  //almacena los tiempos de ejecucion hasta el momento
52  t[0]=times(&pb[0]);
53
54  //se abre este mismo fichero (fuente1.c)
55  //repetidas veces para poder tomar tiempos para las estadisticas acordes
56  printTask(1, "esta anotando estadisticas de uso de CPU...");
57  for(h=1;h<=5000000;h++)
58  {
59    fd=open("fuente1.c",O_RDONLY);
60    close(fd);
61  }
62
63  //se crea la tuberia
64  if(pipe(tuberia)==-1) printError("error al crear la tuberia");
65
66  //se crea P2
67  if((pids[0]=fork())==-1) printError("error en la creacion de P2");
68  //se ejecuta P2
69  else if(pids[0]==0)
70  {
71    //se lee el mensaje de la tuberia sin nombre
72    if(read(tuberia[0], msgPipe, MAX)==-1)
73      printError("error en la lectura de la tuberia sin nombre");
74    printTask(2, "lee el mensaje de la tuberia sin nombre");
75    //se cierra el descriptor de escritura de la tuberia
76    if(close(tuberia[1])==-1) printError("error al cerrar el descriptor de escritura");
77
78    //se crea el fichero FIFO fichero1
79    if(mknod("fichero1", S_IFIFO | 0666, 0)==-1) printError("error en la creacion de
      fichero1");
80    //se abre el fichero1 con permisos de lectura/escritura
81    if((f1=open("fichero1", O_RDWR))==-1) printError("error al abrir fichero1");
82    //se escribe el mensaje en fichero1
83    if(write(f1, msgPipe, strlen(msgPipe)+1)==-1) printError("errir al escribir en
      fichero1");
84    printTask(2, "escribe el mensaje en fichero1");
85
86    //se ejecuta Ej2
87    printTask(2, "ejecuta Ej2");
88    execv("./Trabajo2/Ej2", NULL);
89  }
90  //se ejecuta P1
91  else
92  {
93    //se lee el mensaje desde la entrada estandar
94    printf("Introduzca el mensaje: ");
95    gets(mensaje);
96    //se escribe el mensaje en la tuberia
97    if(write(tuberia[1], mensaje, strlen(mensaje)+1)==-1)
98      printError("error en la escritura de la tuberia sin nombre");
99    printTask(1, "escribe el mensaje en la tuberia sin nombre");
100   //se cierra el descriptor de escritura de la tuberia
101
102   if(close(tuberia[1])==-1) printError("error al cerrar el descriptor de lectura");
```

```
103
104     //se crea la llave a la que se asociara la cola de mensajes
105     if((llave=ftok("./Trabajo2/Ej1", 'D'))==-1) printError("error al crear la llave");
106
107     //se crea la cola de mensajes
108     if((msqid=msgget(llave, IPC_CREAT | 0600))==-1) printError("error al crear la cola
        de mensajes");
109     //se lee el mensaje de la cola de mensajes
110     if((msgrcv(msqid, &msgQueue, MAX, 2, 0))==-1) printError("error al leer de la cola
        de mensajes");
111     printTask(1, "lee el pid de P3 de la cola de mensajes");
112     //se obtiene el pid de P3 del mensaje leido
113     pids[1]=msgQueue.pid;
114     //se borra la cola de mensajes
115     msgctl(msqid, IPC_RMID, 0);
116
117     //se espera a que P2 haya termnidado
118     wait();
119     //se mata a P2 y P3
120     if((kill((pids[0]), SIGKILL))==-1) printError("error al matar a P2");
121     printTask(1, "borra el proceso P2");
122     if((kill(pids[1], SIGKILL))==-1) printError("error al matar a P3");
123     printTask(1, "borra el proceso P3");
124     //se borra fichero1
125     if((unlink("fichero1"))==-1) printError("error al borrar fichero1");
126     printTask(1, "borra el fichero FIFO fichero1");
127
128     //se obtienen los ticks por segundo del sistema
129     CLK_TCK=sysconf(_SC_CLK_TCK);
130     //almacena los tiempos de ejecucion hasta el momento
131     t[1]=times(&pb[1]);
132     //se imprimen las estadisticas
133     tt=(float)(t[1]-t[0])/CLK_TCK;
134     tu=(float)(pb[1].tms_utime-pb[0].tms_utime)/CLK_TCK;
135     tn=(float)(pb[1].tms_stime-pb[0].tms_stime)/CLK_TCK;
136     tcu=(float)(pb[1].tms_cutime-pb[0].tms_cutime)/CLK_TCK;
137     tcn=(float)(pb[1].tms_cstime-pb[0].tms_cstime)/CLK_TCK;
138     printf("\n
        ------------------------------------------------------------------------------\n");
139     printf("MOSTRANDO ESTADISTICAS");
140     printf("\n
        ------------------------------------------------------------------------------\n");
141     printf("Tiempo real                                                  = %4.2g
        segundos\n", tt);
142     printf("Tiempo de uso de la CPU en modo usuario                      = %4.2g
        segundos\n", tu);
143     printf("Tiempo de uso de la CPU en modo nucleo                       = %4.2g
        segundos\n", tn);
144     printf("Tiempo de uso de la CPU de los procesos hijos en modo usuario = %4.2g
        segundos\n", tcu);
145     printf("Tiempo de uso de la CPU de los procesos hijos en modo nucleo  = %4.2g
        segundos\n", tcn);
146     printf("Porcentaje de uso de la CPU                                  = %4.1f%%\n",
         ((tu+tn+tcu+tcn)/tt)*100);
147     printf("
        ------------------------------------------------------------------------------\n");
148   }
149   return 0;
150 }
```

4.3. fuente2.c

```
1  //este archivo es el fichero fuente que al compilarse produce el ejecutable Ej1
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/shm.h>
8  #include <sys/sem.h>
9  #include <fcntl.h>
10
11 #define MAX 256        //tamano maximo del mensaje
12
13 //imprime la tarea realizada por un proceso
14 void printTask(int process, char *task)
15 {
16   printf("El proceso P%d (PID=%d, Ej2) %s\n", process, getpid(), task);
17 }
18
19 //imprime el error causado y termina el programa
20 void printError(char *error)
21 {
22   perror(error);
23   exit(-1);
24 }
25
26 int main()
27 {
28   char msgFIFO[MAX];   //mensaje de fichero1
29   char *vc1;           //mensaje a almacenar en memoria compartida
30   int pid3;            //proceso P3
31   int f1;              //descriptor del fichero1
32   int shmid;           //identificador de la zona de memoria compartida
33   int sem1;            //identificador del semaforo
34   int fd, h;           //para el calculo de estadisticas
35   key_t llave;         //llave que se asociara a la region de MC y al semaforo
36
37   //se abre el fichero FIFO fichero1
38   if((f1=open("fichero1", O_RDWR))==-1) printError("error al abrir fichero1");
39   //se lee en mensaje del fichero FIFO
40   if(read(f1, msgFIFO, MAX)==-1) printError("error al leer el mensaje de fichero1");
41   printTask(2, "lee el mensaje que contiene el fichero FIFO fichero1");
42   //se cierra el fichero FIFO fichero1
43   if(close(f1)==-1) printError("error al cerrar fichero1");
44
45   //se crea la llave a la que se asociara la MC y al semaforo
46   if((llave=ftok("fichero1", 'P'))==-1) printError("error al crear la llave");
47
48   //se crea la zona de memoria compartida
49   if((shmid=shmget(llave, MAX*sizeof(char), IPC_CREAT | 0600))==-1)
50     printError("error al crear la zona de memoria compartida");
51   //se une la zona de memoria compartida al espacio de direcciones virtuales de vc1
52   if((vc1=shmat(shmid, 0, 0))==(char *)-1)
53     printError("error al asignar el espacio de direcciones virtuales a la zona de
       memoria compartida");
54
55   //se crea el semaforo
56   if((sem1=semget(llave, 1, IPC_CREAT | 0600))==-1) printError("error en la creacion del
       semaforo");
57   //se inicia el semaforo a 0
58   if(semctl(sem1, 0, SETVAL, 0)==-1) printError("error al iniciar el semaforo");
59   //se crea la operacion V()
60   struct sembuf opV[1];
```

```
61    opV[0].sem_num=0;
62    opV[0].sem_op=1;
63    opV[0].sem_flg=0;
64
65    //se crea P3
66    if((pid3=fork())==-1) printError("error en la creacion de P2");
67    //se ejecuta P3
68    else if(pid3==0)
69    {
70      //se ejecuta Ej3
71      printTask(3, "ejecuta Ej3");
72      execv("./Trabajo2/Ej3", NULL);
73    }
74    //se ejecuta P2
75    else
76    {
77      //se abre el fichero fuente1.c que se encuentra en el directorio actual
78      //repetidas veces para poder tomar tiempos para las estadisticas acordes
79      printTask(2, "esta anotando estadisticas de uso de CPU...");
80      for(h=1;h<=5000000;h++)
81      {
82        fd=open("fuente1.c",O_RDONLY);
83        close(fd);
84      }
85      //se duerme a P2 durante 1 segundo
86      sleep(1);
87
88      //se escribe el mensaje en memoria compartida
89      strcpy(vc1, msgFIFO);
90      printTask(2, "escribe el mensaje en la variable compartida vc1");
91
92      //se ejecuta la operacion V() del semaforo
93      if(semop(sem1, opV, 1)==-1) printError("error al realizar operacion V del semaforo")
        ;
94      printTask(2, "levanta el semaforo y se suspende su ejecucion");
95
96      //se espera a que termine P3 y se suspende la ejecucion de P2
97      wait();
98      exit(1);
99    }
100 }
```

4.4. fuente2.c

```
1  //este archivo es el fichero fuente que al compilarse produce el ejecutable Ej3
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <sys/types.h>
7  #include <sys/ipc.h>
8  #include <sys/msg.h>
9  #include <sys/shm.h>
10 #include <sys/sem.h>
11 #include <fcntl.h>
12
13 #define MAX 256      //tamano maximo del mensaje
14
15 //imprime la tarea realizada por un proceso
16 void printTask(int process, char *task)
17 {
18   printf("El proceso P%d (PID=%d, Ej3) %s\n", process, getpid(), task);
19 }
```

```
20
21 //imprime el error causado y termina el programa
22 void printError(char *error)
23 {
24   perror(error);
25   exit(-1);
26 }
27
28 int main()
29 {
30   char *vc1;          //mensaje de la memoria compartida
31   int shmid;          //identificador de la zona de memoria compartida asociado a llave
32   int sem1;           //identificador del semaforo
33   int msqid;          //identificador de la cola de mensajes
34   key_t llaveCM;      //llave para la cola de mensajes
35   key_t llave;        //llave para abrir el semaforo y la memoria compartida
36   struct
37   {
38     long tipo;
39     pid_t pid;
40   } msgQueue;         //almacena el mensaje y el tipo de una cola de mensajes
41   msgQueue.tipo=2;  //se asigna el tipo 2 y el pid de P3
42   msgQueue.pid=getpid();
43
44   //se crrea la misma llave que se utilizo en Ej2 para crear el semaforo y la MC
45   if((llave=ftok("fichero1", 'P'))==-1) printError("error al crear la llave");
46
47   //se carga la region de memoria compartida creada en Ej2
48   if((shmid=shmget(llave, MAX*sizeof(char), 0600))==-1) printError("error al cargar la
       memoria compartida");
49
50   //se carga el semaforo creado en Ej2
51   if((sem1=semget(llave, 1, 0666))==-1) printError("error al cargar el semaforo");
52   //se crea la operacion P()
53   struct sembuf opP[1];
54   opP[0].sem_num=0;
55   opP[0].sem_op=-1;
56   opP[0].sem_flg=0;
57   printTask(3, "espera a que se levante el semÃ¡foro");
58   //se realiza la operacion P() del semaforo
59   if(semop(sem1, opP, 1)==-1) printError("error al realizar operacion V del semaforo");
60
61   //se une la zona de memoria compartida al espacio de direcciones virtuales de vc1 para
        leer el mensaje
62   if((vc1=shmat(shmid, 0, 0))==(char *)-1)
63     printError("error al asignar el espacio de direcciones virtuales a la zona de
       memoria compartida");
64   printTask(3, "lee el mensaje de la region de memoria compartida");
65   //se muestra el mensaje
66   printf("Mensaje: %s\n", vc1);
67   //se separa la MC del espacio de direcciones virtuales
68   if(shmdt((int *)vc1)==-1)
69     printError("error al separar el espacio de direcciones virtuales de la region de
       memoria compartida");
70   //se borra la zona de memoria compartida
71   if(shmctl(shmid, IPC_RMID, 0)==-1) printError("error al borrar la zona de memoria
       compartida");
72   //se libera el semaforo
73   if(semctl(sem1, 0, IPC_RMID)==-1) printError("error al liberar el semaforo");;
74
75   //se crea la misma llave que se utilizo en Ej1 para crear la cola de mensajes
76   if((llaveCM=ftok("./Trabajo2/Ej1", 'D'))==-1) printError("error al crear la llave");
77
78   //se abre la cola de mensajes creada en Ej1
```

```
79    if((msqid=msgget(llaveCM, 0600))==-1) printError("error al abrir la cola de mensajes")
        ;
80    //se envia el pid de P3 por la cola de mensajes
81    printTask(3, "envia su pid por la cola de mensajes");
82    if(msgsnd(msqid, &msgQueue, MAX, 0)==-1) printError("error al enviar por la cola de
        mensajes");
83
84    //se suspende la ejecucion de P3
85    exit(1);
86  }
```

5. Opinión personal y material utilizado

La dificultad de la práctica, comparada con la primera, ha sido más sencilla y amena. Prácticamente todo el código se puede ir obteniendo de los ejemplos que se muestran a lo largo del capítulo. Ha sido muy útil para repasar el temario estudiado, comprender y aprender a utilizar los distintos sistemas de comunicación entre procesos.

Como dificultades se han tenido algunos errores al escribir el código, los cuales la mayoría han sido fácilmente solventados gracias a la claridad de como los muestra el compilador **gcc**. Ha habido otros que no se han llegado a comprender, por ejemplo: al principio se habían creado dos variables en `fuente1.c` para almacenar los pids de **P2** y **P3**, las cuales almacenaban correctamente sus valores, pero a la hora de matar el proceso **P2** el valor de la variable había cambiado, lo mismo sucedió con las variable que almacena la segunda toma de tiempo de ejecución, **t2**, tomando el mismo valor que la que almacenaba la variable con el pid de **P2**. Estos errores se han solucionado creando, en vez de variables individuales, sendos arrays.

El material utilizado ha sido la máquina virtual ofrecida por el equipo docente. Para la edición de los tres programas se ha usado `gedit`. El texto base (en especial el **Capítulo 6** y el **Apéndice B**) ha sido la guía principal para la realización, no ha sido necesaria ninguna fuente más. Este documento ha sido creado con el editor de texto MiTEXescrito en LaTeX.