Trabajo de Diseño y Administración de Sistemas Operativos

CARIDE SANTEIRO, CARLOS (FIRMA)

Firmado digitalmente por CARIDE SANTEIRO, CARLOS (FIRMA)
Fecha: 2018.12.15 15:31:56 +01'00'

Alumno: Carlos Caride Santeiro
DNI: 44446239-G
Centro Asociado: Ourense
Teléfono de contacto: 690155343
Email: ccaride5@alumno.uned.es

# Segunda PED

## *Introducción*

Los procesos deben comunicarse e interactuar entre ellos compartiendo datos y recursos. Para ello UNIX pone mecanismos IPCs universales (señales y tuberías). Asimismo, System V ofrece mecanismos de cola de mensajes, memoria compartida y semáforos para realizar la intercomunicación entre procesos.

En este trabajo consiste en un combate entre varios procesos hijos que será arbitrada por el proceso padre.

El proceso inicial es el padre, que se encarga de crear n hijos dado como argumento. Usando métodos de intercomunicación de procesos (IPCs), se encargará de iniciar rondas de combate, eliminar los hijos que han sido abatidos y proclamar al campeón o empate.

El proceso hijo es iniciado por el proceso padre. Al igual que padre, este realizara la comunicación entre otros hijos y con el padre mediante el uso de mecanismos IPC. El hijo establecerá una postura de ataque, y atacará a otro hijo, o defensa. Al acabar la ronda, informará al proceso padre en qué estado se encuentra.

## *Implementación*

El código implementado se puede observar en el apartado ***Código fuente***.

Para el desarrollo de este, se ha utilizado el editor de texto Sublime Text, así como la máquina virtual de Ubuntu 16.04 LTS aportada por el equipo docente.

El encabezado *common.h* declara la estructura del mensaje que se usara entre el proceso padre y los hijos. También declara los métodos de inicializar, solicitar y liberar el semáforo para el recurso compartido de la lista de PID que se usará.

El archivo de código fuente *common.c* implementa los métodos inicializar, solicitar y liberar el semáforo.

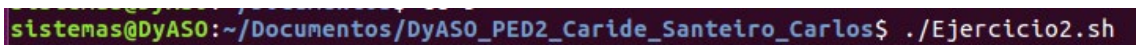
El archivo Ejercicio2.sh es el punto de entrada del ejercicio, ya que este compila tanto *padre.c* como *hijo.c*, crea un archivo FIFO, ejecuta un *cat* en segundo plano al archivo FIFO y ejecuta *PADRE* con 10 hijos como argumento. Finalmente, elimina todos los archivos creados.

El archivo *padre.c* implementa el comportamiento de *PADRE*. Este inicia la llave para los mecanismos IPC. Crea una nueva cola de mensajes, memoria compartida (para la lista de PID de los hijos), un semáforo y una tubería sin nombre. Tras ello, crea n hijos (siendo n pasado como argumento) e inicia una ronda escribiendo por la tubería sin nombre n bytes (siendo n el número de hijos “vivos”). Los hijos le envían el resultado de la ronda y el padre eliminar los que tengan como estado “KO”. Repetirá el proceso hasta que quede uno o ningún hijo, imprimiendo el resultado en el archivo FIFO. Finalmente, elimina todos los IPC que ha creado.

El archivo *hijo.c* implementa el comportamiento de *HIJO*. Este obtiene los IPC creados por el proceso padre y espera a que este último mande la señal de inicio de ronda. Tras ello, se pone aleatoriamente en modo ataque o defensa. Si ataca, elige un hijo aleatorio (no a si mismo), y lo ataca enviándole la señal de usuario 1. Tras ello, envía el mensaje de estado al proceso padre y espera una nueva ronda.

## *Ejecución de ejemplo*

El proceso se inicia con la llamada al script *Ejercicio2.sh*.




En la siguiente imagen se pude comprobar el estado de la primera ronda, donde los hijos 2,8 y 9 has sido emboscados. Asimismo, el hijo 1 y 10 repelen un ataque.

```
sistemas@DyASO:~/Documentos/DyASO_PED2_Caride_Santeiro_Carlos$ ./Ejercicio2.sh
Hijos creados. Preparando hijos para ataques...

Iniciando ronda de ataques
El hijo 6 decide defender
El hijo 10 decide defender
El hijo 5 decide defender
El hijo 9 decide atacar
El hijo 7 decide defender
El hijo 8 decide atacar
El hijo 3 decide atacar
El hijo 4 decide atacar
El hijo 2 decide atacar
El hijo 1 decide defender
El hijo 9 ataca al hijo 1
El hijo 8 ataca al hijo 9
El hijo 9 ha sido emboscado mientras realizaba un ataque
El hijo 3 ataca al hijo 2
El hijo 4 ataca al hijo 8
El hijo 8 ha sido emboscado mientras realizaba un ataque
El hijo 2 ha sido emboscado mientras realizaba un ataque
El hijo 2 ataca al hijo 10
El hijo 10 ha repelido un ataque
El hijo 1 ha repelido un ataque
```

En la ronda número dos, los hijos 2,8 y 9 ya no existen, y cae en esta ronda el hijo 1. En la tercera ronda, caen los hijos 3 y 4. En la cuarta, el hijo 6 es el emboscado. En la siguiente imagen se pueden ver los resultados de las rondas 2 a 4.

```
Iniciando ronda de ataques
El hijo 10 decide defender
El hijo 5 decide atacar
El hijo 6 decide atacar
El hijo 7 decide defender
El hijo 3 decide atacar
El hijo 4 decide atacar
El hijo 1 decide atacar
El hijo 5 ataca al hijo 10
El hijo 10 ha repelido un ataque
El hijo 6 ataca al hijo 1
El hijo 3 ataca al hijo 1
El hijo 4 ataca al hijo 1
El hijo 1 ha sido emboscado mientras realizaba un ataque
El hijo 1 ataca al hijo 7
El hijo 7 ha repelido un ataque

Iniciando ronda de ataques
El hijo 10 decide defender
El hijo 5 decide defender
El hijo 6 decide defender
El hijo 7 decide atacar
El hijo 3 decide atacar
El hijo 4 decide atacar
El hijo 7 ataca al hijo 3
El hijo 3 ha sido emboscado mientras realizaba un ataque
El hijo 3 ataca al hijo 4
El hijo 4 ha sido emboscado mientras realizaba un ataque
El hijo 4 ataca al hijo 3
El hijo 3 ha sido emboscado mientras realizaba un ataque

Iniciando ronda de ataques
El hijo 6 decide atacar
El hijo 5 decide atacar
El hijo 10 decide defender
El hijo 7 decide defender
El hijo 5 ataca al hijo 6
El hijo 6 ha sido emboscado mientras realizaba un ataque
El hijo 6 ataca al hijo 10
El hijo 10 ha repelido un ataque
```

Finalmente, en la siguiente imagen se muestra las últimas tres rondas de este combate, donde el hijo 7 ha sido el ganado de la contienda. Asimismo, se puede ver los *IPCs* del sistema, donde se puede comprobar que los *IPCs* creados por *PADRE* han sido eliminados.

```
Iniciando ronda de ataques
El hijo 5 decide defender
El hijo 10 decide defender
El hijo 7 decide atacar
El hijo 7 ataca al hijo 10
El hijo 10 ha repelido un ataque

Iniciando ronda de ataques
El hijo 10 decide defender
El hijo 5 decide defender
El hijo 7 decide atacar
El hijo 7 ataca al hijo 10
El hijo 10 ha repelido un ataque

Iniciando ronda de ataques
El hijo 10 decide atacar
El hijo 5 decide atacar
El hijo 7 decide defender
El hijo 10 ataca al hijo 5
El hijo 5 ha sido emboscado mientras realizaba un ataque
El hijo 5 ataca al hijo 10
El hijo 10 ha sido emboscado mientras realizaba un ataque
El hijo 7 ha ganado

------ Colas de mensajes -----
key        msqid      propietario perms      bytes utilizados mensajes

---- Segmentos memoria compartida ----
key        shmid      propietario perms      bytes      nattch     estado
0x00000000 327680     sistemas   600        524288     2          dest
0x00000000 1376257    sistemas   600        524288     2          dest
0x00000000 458754     sistemas   600        524288     2          dest
0x00000000 819203     sistemas   600        524288     2          dest
0x00000000 851972     sistemas   600        16777216   2
0x00000000 1081349    sistemas   600        524288     2          dest
0x00000000 1277958    sistemas   600        33554432   2          dest
0x00000000 1212423    sistemas   600        524288     2          dest
0x00000000 1245192    sistemas   600        67108864   2          dest
0x00000000 1474569    sistemas   600        524288     2          dest

------ Matrices semáforo -------
key        semid      propietario perms      nsems
```

## *Código fuente*

Archivo padre.c:

```c
/*****************************************************************
 * Alumno: Carlos Caride Santeiro
 * DNI: 44446239-G
 * Centro Asociado: OURENSE
 * Teléfono de contacto: 690.155.343
 * Email: ccaride5@alumno.uned.es
 * Curso: 2018/2019
 * Fecha: 15/12/2018
 *
 * Proceso padre del Trabajo II: Combate de procesos
 *****************************************************************/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>
#include <sys/wait.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <string.h>

#include "common.h"

#define TRUE 1

int main(int argc, char *argv[]) {
    //Estructura para el contenido del mensaje recibido y longitud del mismo
    struct MensajeEstado mensaje;
    int longitudMensaje = sizeof(mensaje) - sizeof(mensaje.tipo);

    //Nombre del archivo FIFO que se usará para escribir el resultado.
    char * resultado = "resultado";

    //Enteros auxiliares para el desarrollo del programa
    int k, hijosIniciales, msgReceived, deletedPID, retornoWait, resultadoFIFO, pid;

    //Llave usada para la cola de mensajes, semáforo y memoria compartida
    key_t llave;
    //Identificador de la cola de mensajes
    int mensajes;
    //Identificador de la memoria compartida
    int listaSHM;
    //Puntero al primer elemento de la memoria compartida
    int *lista;
    //Identidcicador del semáforo
    int sem;
    //Identificadores de la tuberia sin nombre
    int barrera[2];

    //Mensaje que se envía antes de cada ronda de ataques
    char msgContienda[] = "\nIniciando ronda de ataques";

    //Mensaje de resultado
    char msgResultado[50];

    //Comprobamos que se nos pasara como arguemento el número de hijos
    if (argc < 2) {
        printf("PADRE--Error en el numero de argumentos\n");
        exit(-1);
    }

    //Convertimos el segundo argumento en el número de hijos que realizarán el combate.
    k = atoi(argv[1]);
    hijosIniciales = atoi(argv[1]);

    //Creamos la llave para la cola de mensajes, semaforo y memoria compartida
    if ((llave = ftok(argv[0], 'C')) == (key_t)-1) {
        perror("PADRE--ftok");
        exit(-1);
    }

    //Creamos la cola de mensajes
    if((mensajes = msgget(llave, IPC_CREAT | 0600)) == -1) {
        perror("PADRE--msgget");
        exit(-1);
    }

    //Creamos la memoria compartida y la obtenemos
    if ((listaSHM = shmget(llave, hijosIniciales * sizeof(int), IPC_CREAT | 0600)) == -
1) {
        perror("PADRE--shmget");
        exit(-1);
    }

    //Se obtiene el puntero al array de la memoria compartida
    lista = shmat(listaSHM, 0, 0);

```

```
91.     //Se crea el semaforo
92.     if((sem = semget(llave, 1, IPC_CREAT | 0600)) == -1) {
93.         perror("PADRE--semget");
94.         exit(-1);
95.     }
96.
97.     //Se inicia el semaforo
98.     init_sem(sem, 1);
99.
100.    //Se crear la tubería sin nombre
101.    if(pipe(barrera) == -1) {
102.        perror("PADRE--pipe");
103.        exit(-1);
104.    }
105.
106.    //Iniciamos los hijos indicados como argumento
107.    for (int i = 0; i < hijosIniciales; i++)
108.    {
109.        char iStr[30];
110.        char barreraStr[30];
111.        sprintf(iStr, "%d", i);
112.        sprintf(barreraStr, "%d", barrera[0]);
113.        if ((pid=fork()) == -1) {
114.            perror("fork");
115.            exit(7);
116.        } else if (pid == 0) {
117.            int ret = execl("HIJO", "HIJO", argv[0], iStr, argv[1], barreraStr, NULL);
118.            exit(0);
119.        }
120.    }
121.
122.    //Establecemos el numero de hijos "caidos en combate" a cero
123.    deletedPID = 0;
124.
125.    //Esperamos 0.1 segundos para que esten todos los hijos esperando a la señal
126.    printf("Hijos creados. Preparando hijos para ataques...\n");
127.    usleep(100000);
128.
129.    //Iniciamos la batalla hasta que solo quede uno o ninguno
130.    while (TRUE) {
131.
132.        printf("%s\n", msgContienda);
133.
134.        //Damos la señal de ataque por la tuberia con k bytes = numero de hijos vivos
135.        write(barrera[1], msgContienda, k);
136.        //Esperamos 0.3 segundos a que acaben la ronda
137.        usleep(300000);
138.
139.        //Leemos los mensajes que nos envian los hijos
140.        msgReceived = 0;
141.        while(msgReceived < k) {
142.            if(msgrcv(mensajes, &mensaje, longitudMensaje, 1, 0) == -1) {
143.                perror("HIJO--msgrcv");
144.                exit(-1);
145.            }
146.
147.            //Si el estado del mensajes es "KO" terminamos el proceso.
148.            if (strcmp(mensaje.estado, "KO") == 0) {
149.                kill(mensaje.PID, SIGTERM);
150.                wait(&retornoWait);
151.
152.                //Asignamos al valor de hijo en la lista como 0 para indicar que ha falle-
    cido el hijo
153.                wait_sem(sem);
154.                for (int i = 0; i < hijosIniciales; ++i)
155.                {
156.                    if (mensaje.PID == lista[i]) {
157.                        lista[i] = 0;
158.                    }
```

```
159.                }
160.                signal_sem(sem);
161.                deletedPID++;
162.            }
163.            msgReceived++;
164.        }
165.
166.        //Comprobamos los hijos que continuan vivos. Si es uno, es el gana-
    dor. Si cero, empate. En otro caso,
167.        //se realiza otra ronda de ataques.
168.        k = hijosIniciales - deletedPID;
169.        if(k == 1){
170.            //Comprobamos que hijo es el ganador e imprimimos el resultado por el fi-
    chero FIFO
171.            wait_sem(sem);
172.            for (int i = 0; i < hijosIniciales; ++i)
173.            {
174.                if (0 != lista[i]) {
175.                    //Terminamos el hijo ganador.
176.                    kill(lista[i], SIGTERM);
177.                    wait(&retornoWait);
178.
179.                    //Imprimimos el resultado
180.                    resultadoFIFO = open("resultado", O_WRONLY);
181.                    sprintf(msgResultado, "El hijo %d ha ganado\n", i+1);
182.                    write(resultadoFIFO, msgResultado, strlen(msgResultado));
183.                    close(resultadoFIFO);
184.
185.                    break;
186.                }
187.            }
188.            signal_sem(sem);
189.            break;
190.        }
191.        if (k == 0) {
192.            //Imprimimos por el fichero FIFO que hubo un empate
193.            resultadoFIFO = open("resultado", O_WRONLY);
194.            sprintf(msgResultado, "Empate\n");
195.            write(resultadoFIFO, msgResultado, strlen(msgResultado));
196.            close(resultadoFIFO);
197.            break;
198.        }
199.    }
200.
201.    //Eliminamos la cola de mensajes
202.    if (msgctl(mensajes, IPC_RMID, 0) == -1) {
203.        perror("PADRE--msgctl--remove");
204.        exit(-1);
205.    }
206.
207.    //Desasociamos la lista y eliminamos la memoria compartida
208.    shmdt(lista);
209.    if (shmctl(listaSHM, IPC_RMID, 0) == -1) {
210.        perror("PADRE--shmctl--remove");
211.        exit(-1);
212.    }
213.
214.    //Eliminamos el semaforo
215.    if (semctl(sem, 0, IPC_RMID, 0) == -1) {
216.        perror("PADRE--semctl--remove");
217.        exit(-1);
218.    }
219.
220.    //Cerramos la tuberia sin nombre
221.    close(barrera[0]);
222.    close(barrera[1]);
223.
224.    //Mostramos los IPCs del sistema
225.    system("ipcs");
```

```
226.
227.     return 0;
228.}
```

Archivo hijo.c:

```
1.  /*****************************************************************
2.  * Alumno: Carlos Caride Santeiro
3.  * DNI: 44446239-G
4.  * Centro Asociado: OURENSE
5.  * Teléfono de contacto: 690.155.343
6.  * Email: ccaride5@alumno.uned.es
7.  * Curso: 2018/2019
8.  * Fecha: 15/12/2018
9.  *
10. * Proceso hijo del Trabajo II: Combate de procesos
11. *****************************************************************/
12.
13. #include <sys/types.h>
14. #include <sys/ipc.h>
15. #include <sys/msg.h>
16. #include <sys/sem.h>
17. #include <sys/shm.h>
18. #include <stdio.h>
19. #include <sys/wait.h>
20. #include <stdlib.h>
21. #include <unistd.h>
22. #include <string.h>
23. #include <time.h>
24.
25.
26. #include "common.h"
27.
28. #define TRUE 1
29.
30. //Estado del hijo "KO" o "OK"
31. char estado[2];
32.
33. //Manejador de señal SIGUSR1 cuando el hijo se defiende
34. void defensa();
35. //Manejador de señal SIGUSR1 cuando el hijo se defiende
36. void indefenso();
37.
38. //Numero de hijo
39. int hi;
40.
41. int main(int argc, char *argv[]) {
42.     //Estructura para el contenido del mensaje recibido y longitud del mismo
43.     struct MensajeEstado mensaje;
44.     int longitudMensaje = sizeof(mensaje) - sizeof(mensaje.tipo);
45.
46.     //buffer auxiliar para la lectura de 1 byte de la barrera
47.     char data[1];
48.
49.     //PID del hijo
50.     int pid;
51.     //PID del hijo que se va a atacar
52.     int pidAtaque;
53.     //Numero de hijos iniciales
54.     int hijosIniciales;
55.     //Auxiliar de la funcion rand
56.     int rndIndex;
57.
58.     //Llave usada para la cola de mensajes, semáforo y memoria compartida
59.     key_t llave;
60.     //Identificador de la cola de mensajes
61.     int mensajes;
62.     //Identificador de la memoria compartida
63.     int listaSHM;
```

```
64.     //Puntero al primer elemento de la memoria compartida
65.     int *lista;
66.     //Identidcicador del semáforo
67.     int sem;
68.     //Identificador de la tuberia sin nombre (lectura)
69.     int barrera;
70.
71.     //Obtenemos el PID del proceso hijo
72.     pid = getpid();
73.
74.     //El segundo argumento es el numero del hijo
75.     hi = atoi(argv[2]);
76.     //El tercer argumento es el numero total de hijos
77.     hijosIniciales = atoi(argv[3]);
78.     //El cuarto argumento es el descriptor de la barrera (lectura)
79.     barrera = atoi(argv[4]);
80.
81.     //Se establecen los datos invariables del mensaje: tipo y PID
82.     mensaje.tipo = 1;
83.     mensaje.PID = pid;
84.
85.     //Creamos la llave para la cola de mensajes, semaforo y memoria compartida
86.     if ((llave = ftok(argv[1], 'C')) == -1) {
87.         perror("HIJO-ftok");
88.         exit(-1);
89.     }
90.
91.     //Obtenemos la cola de mensajes (ya creada por PADRE)
92.     if((mensajes = msgget(llave, IPC_CREAT | 0600)) == -1) {
93.         perror("HIJO--msgget");
94.             exit(-1);
95.     }
96.
97.     //Obtenemos la memoria compartida (creado por PADRE)
98.     if ((listaSHM = shmget(llave, hijosIniciales * sizeof(int), IPC_CREAT | 0600)) == -
    1) {
99.         perror("HIJO--shmget");
100.            exit(-1);
101.    }
102.
103.    //Se obtiene el puntero al array de la memoria compartida
104.    lista = shmat(listaSHM, 0, 0);
105.
106.    //Obtenemos el semaforo (creado por PADRE)
107.    if((sem = semget(llave, 1, IPC_CREAT | 0600)) == -1) {
108.        perror("HIJO--semget");
109.            exit(-1);
110.    }
111.
112.    //Establecemos en la lista el PID del proceso
113.    wait_sem(sem);
114.    lista[hi] = pid;
115.    signal_sem(sem);
116.
117.    //Inicializamos rand con una semilla aleatoria.
118.    srand(time(NULL)*(hi+1));
119.
120.    //Bucle de ronda de ataques
121.    while (TRUE) {
122.
123.        //Establecemos el estado OK
124.        strcpy(estado, "OK");
125.
126.        //Esperamos a la señal de inicio de ataque leyendo un byte de la tuberia sin nom-
    bre
127.        read(barrera, data, 1);
128.
129.        //Se decide si se actaca o se defiende
130.        if(rand() % 2 == 0) {
```

```
            //Establecemos defensa y esperamos al final de la contienda
            signal(SIGUSR1, defensa);
            printf("El hijo %d decide defender\n", hi + 1);
            usleep(200000);
        } else {
            //Establecemos ataque
            signal(SIGUSR1, indefenso);
            printf("El hijo %d decide atacar\n", hi + 1);
            usleep(100000);

            //Decidimos a que hijo atacamos aleatoriamente
            pidAtaque = 0;

            wait_sem(sem);
            while(pidAtaque == 0) {
                rndIndex = rand() % hijosIniciales;
                if (lista[rndIndex] != pid && lista[rndIndex] != 0) {
                    pidAtaque = lista[rndIndex];
                }
            }
            signal_sem(sem);

            printf("El hijo %d ataca al hijo %d\n", hi + 1, rndIndex + 1);

            //Atacamos al hijo seleccionado
            kill(pidAtaque, SIGUSR1);
            usleep(100000);
        }

        //Mandamos un mensaje a PADRE indicando el estado actual del hijo
        strcpy(mensaje.estado, estado);
        if(msgsnd(mensajes, &mensaje, longitudMensaje, 0) == -1) {
            perror("HIJO--msgsend");
            exit(-1);
        }
    }
}

void defensa() {
    printf("El hijo %d ha repelido un ataque\n", hi + 1);
    strcpy(estado, "OK");
}

void indefenso() {
    printf("El hijo %d ha sido emboscado mientras realizaba un ataque\n", hi + 1);
    strcpy(estado, "KO");
}
```

Archivo common.h:

```
/*****************************************************************
* Alumno: Carlos Caride Santeiro
* DNI: 44446239-G
* Centro Asociado: OURENSE
* Teléfono de contacto: 690.155.343
* Email: ccaride5@alumno.uned.es
* Curso: 2018/2019
* Fecha: 15/12/2018
*
* Procedimientos init_sem, wait_sem y signal_sem comunes a padre e hijo
* Estructura del mensaje
*****************************************************************/

//Estructura del mensaje usado
struct MensajeEstado
{
  long tipo;
  int PID;
  char estado[2];
```

```
};

//Inicializacion del semaforo
int init_sem(int semid, int valor);

//Soliciud de uso exclusivo
int wait_sem(int semid);

//Libracion del uso exclusivo
int signal_sem(int semid);
```

Archivo common.c

```
/**************************************************************************
* Alumno: Carlos Caride Santeiro
* DNI: 44446239-G
* Centro Asociado: OURENSE
* Teléfono de contacto: 690.155.343
* Email: ccaride5@alumno.uned.es
* Curso: 2018/2019
* Fecha: 15/12/2018
*
* Procedimientos init_sem, wait_sem y signal_sem comunes a padre e hijo
**************************************************************************/

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

#include "common.h"

//Estructura del mensaje usado
int init_sem(int semid, int valor) {
    if (semctl(semid, 0, SETVAL, valor) == -1) {
        perror("COMMON--init_sem:");
        return -1;
    }

    return 0;
}

//Soliciud de uso exclusivo
int wait_sem(int semid) {

    struct sembuf op[1];

    op[0].sem_num = 0;
    op[0].sem_op = -1;
    op[0].sem_flg = 0;

    if (semop(semid, op, 1) == -1) {
        perror("COMMON--wait_sem:");
        return -1;
    }

    return 0;
}

//Libracion del uso exclusivo
int signal_sem(int semid) {

    struct sembuf op[1];

    op[0].sem_num = 0;
    op[0].sem_op = 1;
    op[0].sem_flg = 0;

    if (semop(semid, op, 1) == -1) {
```

```
57.         perror("COMMON--signal_sem\n");
58.         return -1;
59.     }
60.
61.     return 0;
62. }
```

Archivo Ejercicio2.sh:

```
1.  #!/bin/bash
2.  #/**************************************************************************
3.  #* Alumno: Carlos Caride Santeiro
4.  #* DNI: 44446239-G
5.  #* Centro Asociado: OURENSE
6.  #* Teléfono de contacto: 690.155.343
7.  #* Email: ccaride5@alumno.uned.es
8.  #* Curso: 2018/2019
9.  #* Fecha: 15/12/2018
10. #*
11. #* Script Ejercicio2.sh del Trabajo II: Combate de procesos
12. #**************************************************************************/
13.
14. #1 compila los fuentes padre.c e hijo.c con gcc
15. gcc Trabajo2/padre.c Trabajo2/common.c -o PADRE
16. gcc Trabajo2/hijo.c Trabajo2/common.c -o HIJO
17.
18. #2 crea el fihero fifo "resultado"
19. mkfifo resultado
20.
21. #3 lanza un cat en segundo plano para leer "resultado"
22. cat resultado &
23.
24. #4 lanza el proceso padre con 10 hijos
25. ./PADRE 10
26.
27. #5 al acabar limpia todos los ficheros que ha creado
28. rm resultado
29. rm PADRE
30. rm HIJO
```

## Bibliografía.

Para la realización del trabajo se han consultado las siguientes fuentes:

**[MANPAGES]** Manual del programador de Linux. *The Linux man-pages project.*

http://man7.org/linux/man-pages/man5/proc.5.html

**[FSOU2018]** Fundamentos del sistema operativo UNIX.

*José Manuel Díaz Martínez, Rocío Muñoz Mansilla, Dictino Chaos García*

*ISBN: 978-84-615-8772-8 – Segunda revisión*