

Trabajo de Diseño y Administración de Sistemas Operativos

Alumno: Isabel Manzaneque Núñez

DNI: 53902577-F

Centro Asociado: Londres

Teléfono de contacto: +447762347351

Email: imanzaneq3@alumno.uned.es

Segunda PED

Introducción

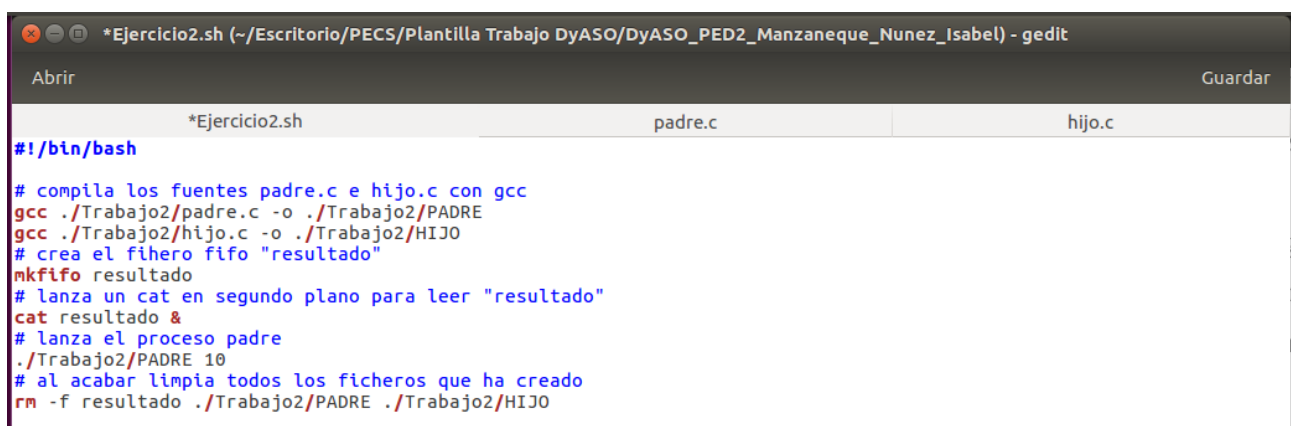
En el ejercicio propuesto, utilizamos dos programas escritos en C y un script en Shell para simular un combate entre procesos: los programas padre.c e hijo.c junto con el script Ejercicio2.sh, orquestarán un enfrentamiento entre procesos hijos arbitrado por el proceso padre. La estructura del combate consiste en un entramado de mecanismos IPC que involucra colas de mensajes, regiones de memoria compartida, semáforos y tuberías sin nombre.

La dinámica del enfrentamiento se basa en rondas, donde cada proceso hijo decide entre atacar o defenderse, siguiendo un protocolo de preparación y ataque. El proceso padre sincroniza las acciones de los contendientes. Tras cada ronda, los hijos comunicarán al padre los resultados de las mismas y el padre tomará decisiones en función de estos: eliminar los procesos derrotados, actualizar la lista de contendientes y, en última instancia, anunciar un ganador. Una vez se obtiene el ganador, se liberan los mecanismos IPC.

Implementación

El trabajo se divide en la implementación de un script de Shell y dos programas en C. A continuación se explica la implementación y las decisiones de diseño de cada uno.

Ejercicio2.sh



```
*Ejercicio2.sh (~/Escritorio/PECS/Plantilla Trabajo DyASO/DyASO_PED2_Manzaneque_Nunez_Isabel) - gedit
Abrir Guardar

*Ejercicio2.sh padre.c hijo.c

#!/bin/bash

# compila los fuentes padre.c e hijo.c con gcc
gcc ./Trabajo2/padre.c -o ./Trabajo2/PADRE
gcc ./Trabajo2/hijo.c -o ./Trabajo2/HIJO
# crea el fichero fifo "resultado"
mkfifo resultado
# lanza un cat en segundo plano para leer "resultado"
cat resultado &
# lanza el proceso padre
./Trabajo2/PADRE 10
# al acabar limpia todos los ficheros que ha creado
rm -f resultado ./Trabajo2/PADRE ./Trabajo2/HIJO
```

Ejercicio2.sh es un script de Shell que se encarga de arrancar la ejecución del ejercicio. En la captura de pantalla se puede observar los pasos que sigue:

1. Comienza compilando ambos programas fuente utilizando gcc. Los ejecutables PADRE e HIJO podrán encontrarse en el directorio ./Trabajo2.
2. Crea el fichero FIFO “resultado” en el directorio en el que se encuentra Ejercicio2.sh. En este fichero se escribirá el resultado de la ejecución del programa.
3. Lanza un proceso cat en segundo plano que se quedará a la espera de leer el resultado del programa, una vez este se haya escrito en el fichero FIFO.
4. Lanza el ejecutable PADRE y le pasa como parámetro el número 10. Este parámetro son los procesos hijos que deseamos que cree el proceso padre.
5. Por último, elimina todos los ficheros que se han creado a excepción del script Ejercicio2.sh, padre.c e hijo.c.

padre.c

El proceso padre es encargado de orquestar y supervisar el combate entre los procesos hijos, estableciendo y coordinando los mecanismos de comunicación entre ellos. A continuación se van a explorar en detalle sus funciones y responsabilidades, ilustrando como dirige el combate y evalúa sus resultados.

Antes de comenzar quiero comentar que, siguiendo las indicaciones del enunciado, he decidido imprimir mensajes adicionales que a mi parecer hace que se vea más claramente lo que está ocurriendo en la comunicación entre procesos. Además, tras cada escritura por salida estándar, he realizado un volcado inmediato del buffer para evitar condiciones de carrera.

Ejercicio2.sh	padre.c	hijo.c
<pre>#include <stdlib.h> #include <string.h> #include <sys/ipc.h> #include <sys/msg.h> #include <sys/types.h> #include <sys/shm.h> #include <sys/sem.h> #include <unistd.h> #include <sys/wait.h> #include <errno.h> #include <fcntl.h> #include <unistd.h> struct mensaje{ long tipo; pid_t pid; char state[3]; }; /** * Inicializa un semaforo a un valor */ void initSem(int sem, int value); /** * Decrementa en 1 el valor de un semaforo */ void waitSem(int sem); /** * Incrementa en 1 el valor de un semaforo */ void signalSem(int sem); /** * Crea N procesos hijo utilizando fork */ void crearNHijos(int N, char argv0[], pid_t *lista, int barrera[2], int sem); /** * Mata un proceso mandandole la senal SIGTERM */ void matarProceso(int *K, pid_t pid, pid_t *lista, int sem);</pre>		

El programa padre comienza con la definición del tipo estructurado “mensaje” y los prototipos de las funciones que se han escrito para que el padre realice su función.

Como se verá más adelante, habrá un intercambio de mensajes en el que los procesos hijo deberán transmitir al padre su PID y su estado. Es por esto que la estructura del mensaje consiste en estos dos campos, además del tipo del mensaje.

Las funciones se explicarán en más detalle cuando veamos sus definiciones, pero se menciona rápidamente que disponemos de cinco funciones: tres de ellas para la manipulación de los semáforos, una para crear los procesos hijo y otra para matar un proceso hijo.

La función main del programa se divide en tres secciones: inicialización, rondas y liberación de los recursos IPC.

```

int main(int argc, char *argv[]){
    // ----- INICIALIZACION -----
    int N, K;
    N = K = atoi(argv[1]);
    struct mensaje msgHijo;

    // crear clave
    key_t key = ftok(argv[0], 'X');

    // crear cola de mensajes
    int mensajes = msgget(key, IPC_CREAT | 0600);
    if (mensajes == -1) {
        perror("padre: msgget");
        exit(-1);
    }

    // crear región de memoria compartida de tamaño N pids
    // y enlazarla con un array con capacidad para N PIDs
    int shrdMemId = shmget(key, N*sizeof(pid_t), IPC_CREAT | 0600);
    if (shrdMemId == -1) {
        perror("padre: shmget");
        exit(-1);
    }
    pid_t *lista = (pid_t*)shmat(shrdMemId, NULL, 0);

    // crea semáforo e inicializar a 1 para exclusion mutua
    int sem = semget(key, 1, IPC_CREAT | 0600);
    if (sem == -1) {
        perror("padre: semget");
        exit(-1);
    }
    initSem(sem, 1);

    // crear tubería sin nombre
    int barrera[2];
    if(pipe(barrera) == -1){
        perror("padre: pipe");
        exit(-1);
    }
}

```

En la sección de inicialización, comenzamos declarando las variables N, K y msgHijo. N es el número de procesos que queremos crear y utilizaremos K como contador de procesos vivos, ambas inicializadas al argumento que se pasa al ejecutable PADRE desde el script Ejercicio2.sh. Por su parte, utilizaremos msgHijo para leer los mensajes que irán llegando al proceso padre desde los procesos hijo.

Hecho esto, se crean los mecanismos IPC que se utilizarán para gestionar la sincronización y comunicación entre procesos. A todos estos se les ha dado permisos 0600 de lectura y escritura para el propietario del archivo.

- Se crea una clave única asociada al fichero ejecutable y la letra X, a la cual se asociarán los mecanismos IPC. Esta clave se almacena en la variable 'key'
- Se utiliza 'msgget' para crear (si no existe) o acceder a una cola de mensajes asociada a la clave. Si hay un error en la creación, se informa al usuario de donde en qué programa está el error y qué llamada al sistema lo ha producido. Esto se realiza en la creación de todos los mecanismos IPC.
- Se utiliza 'shmget' para acceder a o crear una región de memoria de memoria compartida asociada a la clave. El tamaño de la región es N*sizeof(pid_t) para que haya espacio suficiente para N pids. Utilizando 'shmat' se enlaza esta a un array en el espacio de direcciones del proceso y *lista es un puntero al inicio de la región de memoria.
- Se utiliza 'semget' para crear o acceder a un conjunto de semáforos asociados a la clave. Se utiliza la función initSem (que se explicara más adelante) para inicializar el semáforo a 1 y así utilizarlo para realizar exclusión mutua.
- Por último, se utiliza 'pipe' para crear una tubería sin nombre como un array de 2 enteros. El descriptor de lectura de la tubería se almacena en barrera[0] y el de escritura en barrera[1].

Esto no se incluye en el código final, pero justo tras la inicialización de los mecanismos IPC podemos comprobar que estos se han creado correctamente de la siguiente manera:

```
printf("\nsemid : %d - msqid: %d - shmid: %d \n", sem, mensajes, shrdMemId);
fflush(stdout);

//Mostramos semáforos, memoria compartida y colas de mensajes activos
printf ("\nRecursos IPC activos:\n");
fflush(stdout);
system("ipcs -sqm");
```

```
sistemas@DyAS0:~/Escritorio/PECS/Plantilla Trabajo DyAS0/DyAS0_PED2_Manzaneque_Nunez_Isa
bel$ ./Ejercicio2.sh

semid : 6914205 - msqid: 7635104 - shmid: 16908451
```

Semáforo:

0x5801cc60	5046426	sistemas	600	1
0x5801cc39	5079195	sistemas	600	1
0x5801cc6f	5111964	sistemas	600	1
0x5801d43d	6914205	sistemas	600	1
0x5801cca9	5177502	sistemas	600	1
0x5801ccc7	5308575	sistemas	777	1

Cola de mensajes:

0x582703a9	6095006	sistemas	600	136	17
0x5801cca9	6160543	sistemas	600	328	41
0x5801d43d	7635104	sistemas	600	0	0
0x5801cd79	6389921	sistemas	600	0	0

Memoria compartida:

0x00000000	15827103	sistemas	600	40	1	dest
0x00000000	16285856	sistemas	600	40	1	dest
0x00000000	16482465	sistemas	600	40	1	dest
0x00000000	16875682	sistemas	600	40	1	dest
0x5801d43d	16908451	sistemas	600	40	1	

Al final del programa, nos aseguramos de que los mecanismos IPC se han cerrado correctamente comprobando que los IDs ya no se encuentran en estas listas.

Comienza la sección de rondas, con las rondas de ataques y defensas entre procesos:

Ejercicio2.sh	padre.c	hijo.c
<pre>// ----- RONDAS ----- // crea N procesos y espera a que se inicien crearNHijos(N, argv[0], lista, barrera, sem); usleep(20000); // mientras queden 2 o mas contendientes, se hara otra ronda while (K > 1){ printf("\n----- Iniciando ronda de ataques - quedan %d hijos vivos ----- \n", K); fflush(stdout); for(int i = 0; i < 10; i++){ waitSem(sem); printf("Hijo %d: %d\n", i, lista[i]); signalSem(sem); fflush(stdout); } printf("\n"); fflush(stdout); // manda un mensaje de 1 byte por cada hijo vivo char msg = 'P'; for (int i = 0; i < K; i++) { if(write(barrera[i], &msg, sizeof(msg)) < 0){ perror("padre: write"); } printf("padre envia mensaje\n"); fflush(stdout); } printf("\n"); fflush(stdout); // espera a que los hijos reciban mensaje y terminen sus rondas usleep(500000);</pre>		

Lo primero que se hace es crear N procesos hijo y esperar 20ms a que se inicien. Los N procesos (10 en este caso) se crean con la función crearNHijos() que se explicará más adelante en la sección de funciones y a la que se le pasa N, el nombre del programa, la lista de pids, la barrera y el semáforo.

Una vez hecho esto y mientras queden dos o más hijos vivos (K es mayor que 1), se permanecerá en el bucle principal del padre en el que se realizan las rondas de ataque y defensa. Al principio de cada ronda se imprime por pantalla el número de hijos que quedan vivos y el estado de la lista de pids. Aquí se puede apreciar un comportamiento que se repetirá todo el programa: El acceso a la lista de pids siempre se protege con el semáforo, realizando un waitSem(sem) antes de entrar y signalSem(sem) al salir. Ambas funciones se explican en la sección de definiciones de funciones.

Después, por cada hijo que queda vivo (variable K), el padre mandará un mensaje de 1 byte al extremo de escritura de la tubería que recibirán los hijos utilizando 'write(barrera[1])'. Se implementa gestión de errores en caso de que haya un problema al escribir en la barrera. Para asegurarse de que todos los hijos han recibido el mensaje y han terminado la ronda, antes de continuar espera 0.5s.

```
// Padre recibe los resultados de los hijos
int counter = K;
while(msgrcv(mensajes, &msgHijo, sizeof(struct mensaje) - sizeof(long), 2, 0) != -1) {

    printf("Padre recibe mensaje de %d: Tipo: %ld - Estado: %s\n", msgHijo.pid, msgHijo.tipo, msgHijo.state);
    fflush(stdout);

    if(strcmp("KO", msgHijo.state) == 0){
        matarProceso(&K, msgHijo.pid, lista, sem);
    }

    // Si ya ha recibido mensajes de todos los hijos sale del bucle
    counter--;
    if(counter == 0){
        break;
    }
}
```

Una vez el padre ha esperado a que todos los hijos hayan recibido su mensaje, comienza a leer los mensajes que le llegan de estos.

Entra en un bucle de lectura en el que utiliza 'msgrcv' para leer un mensaje de la cola de mensajes, que quedará referenciado en la variable 'msgHijo'. Mostrará por pantalla el contenido del mensaje. Si el campo de estado (state) del mensaje es "KO", entonces utilizará la función matarProceso() (que se explicará mas adelante) para matar al proceso que ha enviado ese mensaje

Se utiliza la variable 'counter' para salir del bucle de lectura: Esta variable se inicializa al número de hijos vivos y desciende tras cada mensaje leído. Al llegar a 0, el padre habrá leído un mensaje de cada hijo y saldrá del bucle con un break.

Cuando deja de cumplirse la condición del bucle principal ($K > 1$), se sale del bucle y llega el momento de anunciar al ganador de la ronda.

```
// Escribir en resultado el ganador de la ronda
char ganador[100];
if(K == 1){
    // 1 ganador
    for(int i = 0; i < 10; i++){

        waitSem(sem);
        if(lista[i] != 0){
            int resultado = open("resultado", O_WRONLY);
            snprintf(ganador, sizeof(ganador), "\n\nEl hijo %d ha ganado\n\n", lista[i]);
            write(resultado, ganador, strlen(ganador));
        }
        signalSem(sem);
    }
}else if(K == 0){
    // empate
    strcpy(ganador, "\n\nEmpate\n\n");
    int resultado = open("resultado", O_WRONLY);
    write(resultado, ganador, strlen(ganador));
}
```

Una vez terminan las rondas pueden darse dos situaciones: existe un solo ganador o ha habido un empate entre dos procesos.

En el primer caso, se recorre la lista de PIDs (protegida por el semáforo) para encontrar el único proceso cuyo PID no es 0. Se abre el fichero FIFO 'resultado' con 'open' y se utiliza 'write' para escribir en él el PID del proceso ganador.

En caso de empate todos los PIDs estarán a 0 en la lista de PIDs, por lo que tan solo se escribe en resultado que ha habido un empate.

En la última sección del main del padre se liberan los recursos IPC

Ejercicio2.sh	padre.c	hijo.c
	<pre>// ----- LIBERAR RECURSOS IPC ----- printf("\nsemid : %d - msqid: %d\n", sem, mensajes); fflush(stdout); // cerrar cola de mensajes msgctl(mensajes, IPC_RMID,0); // cerrar tubería close(barrera[0]); close(barrera[1]); // Desvincular memoria compartida shmdt(lista); shmctl(shrdMemId, IPC_RMID,0); // cerrar semáforo semctl(sem, IPC_RMID,0); //Mostramos semáforos y colas de mensajes activos printf ("\nRecursos IPC activos:\n"); fflush(stdout); system("ipcs -sq"); printf("Finalizando padre\n"); fflush(stdout); return 0; }</pre>	

Comienza mostrando por pantalla el id de la cola de mensajes y el semáforo, ya que nos será útil para buscarlos en las listas y comprobar si se han cerrado correctamente. Después realiza las siguientes acciones:

- Cierra la cola de mensajes usando 'msgctl'
- Cierra la tubería utilizando 'close' sobre cada descriptor
- Desvincula la memoria compartida utilizando 'shmdt' y 'shmctl'
- Cierra el semáforo utilizando 'semctl'
- Muestra los semáforos y colas de mensajes activos utilizando 'system("ipcs -sq")'.
- Indica que el padre ha finalizado

Terminada la función main del padre, se encuentran las definiciones de las funciones que este ha utilizado y cuyos prototipos se vieron al comienzo del programa:

```
void initSem(int sem, int value){  
    if (semctl(sem, 0, SETVAL, value) == -1) {  
        perror("padre: semctl");  
        exit(1);  
    }  
}
```

La función **initSem** inicializa un conjunto de semáforos 'sem' a un valor 'value'. Utiliza la llamada al sistema semctl con la operación SETVAL para establecer el valor del semáforo.

```

void waitSem(int sem){

    struct sembuf op;
    op.sem_num = 0;
    op.sem_op = -1;
    op.sem_flg = 0;
    if(semop(sem, &op, 1) == -1){
        perror("padre: semop");
        exit(1);
    }
}

```

La función **waitSem** realiza la operación de espera (wait) en un semáforo con identificador 'sem'. Utiliza la estructura 'sembuf' con 'sem_op' igual a -1 para realizar la operación de espera y la llamada al sistema 'semop' para realizar la operación en el semáforo.

```

void signalSem(int sem){

    struct sembuf op;
    op.sem_num = 0;
    op.sem_op = 1;
    op.sem_flg = 0;
    if(semop(sem, &op, 1) == -1){
        perror("padre: semop");
        exit(1);
    }
}

```

La función **signalSem** realiza la operación signal en un semáforo con identificador 'sem'. Utiliza la estructura 'sembuf' con 'sem_op' igual a 1 para realizar la operación de señalización y la llamada al sistema 'semop' para realizar la operación en el semáforo.

```

void crearNHijos(int N, char argv0[], pid_t *lista, int barrera[2], int sem){

    int pidCounter = 0;

    for(int i = 0; i < N; i++){

        pid_t resFork = fork();

        if(resFork == -1){
            perror("padre: fork");
            exit(-1);
        }else if(resFork == 0){
            // Proceso Hijo ejecuta HIJO
            char lectura[10];
            char escritura[10];
            sprintf(lectura, "%d", barrera[0]);
            sprintf(escritura, "%d", barrera[1]);
            execl("./Trabajo2/HIJO", "HIJO", argv0, lectura, escritura, NULL);
        }else{
            // Proceso Padre va guardando los pids de los hijos en lista
            waitSem(sem);
            lista[pidCounter] = resFork;
            signalSem(sem);
            pidCounter++;
        }
    }
}

```

La función **crearNHijos** crea N procesos hijos utilizando la llamada al sistema fork. Como parámetros recibe 'N' (la cantidad de hijos a crear), 'argv0' (el nombre del programa principal), 'lista' (array donde se almacenan los PIDs de los hijos), 'barrera' (la tubería sin nombre) y 'sem' (el identificador del conjunto de semáforos).

Si fork() devuelve 0, se está ejecutando el proceso hijo. Este ejecutará el ejecutable HIJO al que le pasará argv0 y los descriptores de la barrera. En el contexto del padre, fork() devuelve el PID del proceso hijo, el cual guardará en la lista de PIDs. Una vez más, el acceso a esta es protegido con el semáforo.


```

void matarProceso(int *K, pid_t pid, pid_t *lista, int sem) {
    if (kill(pid, SIGTERM) == 0) {
        // Esperar a que el proceso hijo termine
        waitpid(pid, 0, 0);
    } else {
        perror("padre: sigterm");
    }

    // actualizar lista
    for(int i = 0; i < 10; i++){
        waitSem(sem);
        if(lista[i] == pid){
            lista[i] = 0;
            (*K)--;
        }
        signalSem(sem);
    }
}

```

La función **matarProceso** mata un proceso hijo dado su PID y actualiza la lista de PIDs en el array lista. Recibe como argumentos ‘K’ (el contador de procesos vivos), ‘pid’ (el PID del proceso a matar), la lista de PIDs y el identificador del semáforo.

Primero envía la señal SIGTERM al PID del proceso que desea matar y utiliza ‘waitpid’ para esperar a que este termine. Una vez termina el proceso, recorre la lista de procesos y pone el PID del proceso correspondiente a 0. Por último, decrementa K.

hijo.c

El programa hijo, cuyo código está en el archivo hijo.c, es el protagonista de las rondas de ataques que se desatan durante el enfrentamiento de procesos. El propósito del programa hijo radica en la ejecución de acciones tácticas durante cada ronda del combate. Su responsabilidad es la toma de decisiones aleatorias entre atacar o defenderse y enviar su estado tras la ronda al padre.

Ejercicio2.sh	padre.c	hijo.c
<pre> #include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <string.h> #include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h> #include <sys/sem.h> #include <sys/shm.h> #include <time.h> #include <signal.h> #include <sys/wait.h> char estado[3]; struct mensaje{ long tipo; pid_t pid; char state[3]; }; /** * Inicializa un semaforo a un valor */ void initSem(int sem, int value); /** * Decrementa en 1 el valor de un semaforo */ void waitSem(int sem); /** * Incrementa en 1 el valor de un semaforo */ void signalSem(int sem); /** * Manejador de la senal SIGUSR1 cuando un proceso esta indefenso */ void indefenso(); </pre>		

```

/**
 * Manejador de la senal SIGUSR1 cuando un proceso se defiende
 */
void defensa();

/**
 * Ataca a un proceso aleatorio de la lista de procesos
 */
void ataque(pid_t *lista, int sem);

/**
 * Cuando el proceso recibe la SIGUSR1, realizara la accion del
 * manejador de defensa o indefenso aleatoriamente
 */
int preparacion();

/**
 * Prepara el mensaje con el resultado del ataque y
 * se lo manda al padre
 */
void enviarMensajeAPadre(int mensajes);

```

Al igual que en el caso de padre.c, hijo.c también comienza con la definición del tipo estructurado “mensaje” y los prototipos de las funciones de las que se sirve para realizar sus acciones. Todas estas se explicarán en la sección de definiciones de funciones, pero ya aquí podemos ver que tenemos las mismas funciones que el padre para el manejo de semáforos: ‘initSem’, ‘waitSem’ y ‘signalSem’. Tenemos además dos manejadores para la señal SIGUSR1: ‘defensa’ e ‘indefenso’, entre las que escogerá de manera aleatoria cuando le llegue esta señal. La función ‘ataque’ realiza las acciones necesarias para atacar a otro proceso. La función ‘preparación’ representa la fase de preparación en la que los procesos deciden si atacar o defender aleatoriamente. Por último, ‘enviarMensajeAPadre’ prepara y envía el mensaje resultado al proceso padre.

La función main del hijo se divide en inicialización y rondas.

Ejercicio2.sh	padre.c	hijo.c
<pre> int main(int argc, char const *argv[]) { // ----- INICIALIZACION ----- printf("Inicio hijo %d\n", getpid()); fflush(stdout); // Recuperar la clave key_t key = ftok(argv[1], 'X'); // Recuperar descriptores int descLectura = atoi(argv[2]); int descEscritura = atoi(argv[3]); // recuperar cola de mensajes int mensajes = msgget(key, 0); if (mensajes == -1) { perror("hijo: mssget"); exit(-1); } // recuperar semaforo int sem = semget(key, 0, 0); if (sem == -1) { perror("hijo: senget"); exit(-1); } // recuperar memoria compartida int shrdMemId = shmget(key, 0, 0); if (mensajes == -1) { perror("hijo: shmget"); exit(-1); } pid_t *lista = (pid_t*)shmat(shrdMemId, NULL, 0); </pre>		

En la fase de inicialización, el proceso hijo re-establece los mecanismos IPC para comunicarse con el padre. Comienza indicando que el hijo con PID getpid() se esta inicializando y pasa a realizar las siguientes acciones:

- Recupera la clave utilizando el nombre del programa principal (que se le había pasado como parámetro a execl) y la letra X.

- Recupera los descriptores de lectura y escritura de la barrera.
- Una vez recuperada la clave, la utiliza para recuperar la cola de mensajes con ‘msgget’. Como se está asumiendo que la cola de mensajes ya ha sido creada con los permisos adecuados, no hace falta especificar estos. Se realiza gestión del error en caso de que no se pueda recuperar la cola de mensajes indicando en qué programa se ha producido el error y qué llamada al sistema lo ha producido, al igual que en los siguientes casos.
- Recupera el semáforo con la clave y ‘semget’. Al igual que antes, al ser un grupo de semáforos ya creado no hace falta especificar nuevamente los permisos.
- Por último, se recupera la zona de memoria compartida utilizando la clave y ‘shmget’.

A continuación comienza la fase de rondas en la que los hijos se atacarán por rondas.

Ejercicio2.sh	padre.c	hijo.c
<pre> // ----- RONDAS ----- close(descEscritura); // bucle principal del hijo while(1){ char buffer[1]; if(read(descLectura, buffer, sizeof(buffer)) > 0){ printf("hijo %d recibe mensaje: %s\n", getpid(), buffer); fflush(stdout); // si la la ronda de preparacion devuelve 0, ataca if(preparacion() == 0){ ataque(lista, sem); } // envia resultado de la ronda a padre enviarMensajeAPadre(mensajes); } sleep(1); } close(descLectura); } </pre>		

Comenzamos cerrando el descriptor de escritura de la tubería porque no lo vamos a utilizar.

El proceso hijo entrará entonces en un bucle infinito en el que quedará esperando a que haya un mensaje del padre en el extremo de lectura de la tubería.

Si lo hay, muestra por consola que ha recibido un mensaje y entra en la fase de preparación (que será explicada más detalladamente en la sección de las definiciones de las funciones). Si la fase de preparación retorna un 0, esto significará que el proceso va a atacar, con lo que se llamará a la función ‘ataque’, también explicada más adelante.

Tras la fase de preparación y ataque/defensa, enviará un mensaje al padre utilizando la función ‘enviarMensajeAPadre’.

Por último, espera un segundo antes de volver a comprobar si hay mensajes en la cola para dar una oportunidad a que sus hermanos lean un mensaje también.

Terminada la función main, se encuentran las definiciones de las funciones que este ha utilizado y cuyos prototipos se vieron al comienzo del programa:

```

void initSem(int sem, int value){
    if (semctl(sem, 0, SETVAL, value) == -1) {
        perror("padre: semctl");
        exit(1);
    }
}

void waitSem(int sem){
    struct sembuf op;
    op.sem_num = 0;
    op.sem_op = -1;
    op.sem_flg = 0;
    if(semop(sem, &op, 1) == -1){
        perror("padre: semop");
        exit(1);
    }
}

void signalSem(int sem){
    struct sembuf op;
    op.sem_num = 0;
    op.sem_op = 1;
    op.sem_flg = 0;
    if(semop(sem, &op, 1) == -1){
        perror("padre: semop");
        exit(1);
    }
}

```

Las funciones que se utilizan para el control de los semáforos son exactamente iguales que en el padre, por lo que no se van a explicar otra vez.

```

void indefenso(){
    printf("\nEl hijo %d ha sido emboscado mientras realizaba un ataque\n", getpid());
    fflush(stdout);
    strcpy(estado, "KO");
}

void defensa(){
    printf("\nEl hijo %d ha repelido un ataque\n", getpid());
    fflush(stdout);
    strcpy(estado, "OK");
}

```

Las funciones ‘indefenso’ y ‘defensa’ implementan los manejadores de la señal SIGUSR1 por parte de los procesos. Esto es, si un proceso recibe esta señal, realizará las acciones del manejador correspondiente. En este caso, se muestran por pantalla unos mensajes y se establece la variable estado a “KO” en el caso de ‘indefenso’ o “OK” en el caso de ‘defensa’.

```

void ataque(pid_t *lista, int sem){
    srand((unsigned int)time(NULL));
    pid_t pidVictima = 0;

    // Escoge un proceso cuyo pid no sea 0 ni el propio
    while(pidVictima == 0 || pidVictima == getpid()){
        int ranIndx = rand() % 10;
        waitSem(sem);
        pidVictima = lista[ranIndx];
        signalSem(sem);
    }
    // envia SIGUSR1 al proceso victima
    if (kill(pidVictima, SIGUSR1) == 0) {
        printf("\n%d Atacando al proceso %d", getpid(), pidVictima);
        fflush(stdout);
    } else {
        perror("Hijo: ataque");
    }
    // Espera a que el proceso victima termine
    usleep(100000);
}

```

La función ‘ataque’ recibe como parámetro la lista de pids y el identificador del grupo de semáforos. Comienza creando una semilla que se utilizará para crear números aleatorios y evaluando una variable ‘pidVictima’ a 0.

En el bucle, procede a buscar el PID de un proceso que esté vivo (que no sea 0) y que no sea el propio. Esto lo hará accediendo a la posición de la lista de PIDs que es el resto de dividir un número aleatorio entre 10. A la lista de PIDs accede de manera segura realizando un waitSem antes de entrar y un signalSem al salir.

Una vez ha encontrado un candidato a atacar, le manda la señal SIGUSR1 y notifica que está atacando a ese proceso. Por último espera a que el proceso al que ha atacado termine.

```
int preparacion(){
    srand((unsigned int)time(NULL));
    strcpy(estado, "OK");

    if(rand() % 2 == 0){
        // proceso ataca
        if(signal(SIGUSR1, indefenso)==SIG_ERR){
            perror("Hijo: indefenso");
            exit(1);
        }
        usleep(100000);
        return 0;
    }else{
        // proceso defiende
        if(signal(SIGUSR1, defensa)==SIG_ERR){
            perror("Hijo: defensa");
            exit(1);
        }
        usleep(200000);
    }
    return 1;
}
```

La función ‘**preparación**’ comienza también creando una semilla aleatoria y estableciendo la variable ‘estado’ a “OK”. Realiza entonces una de las siguientes funciones de manera aleatoria:

- Si un número aleatorio es par, instala la función ‘indefenso’ en SIGUSR1 y espera 0.1 segundos. Devuelve 0
- Si el número aleatorio es impar, se instala la función ‘defensa’ y se espera 0.2 segundos. La función devuelve 1;

```
void enviarMensajeAPadre(int mensajes){
    // Configurar el tipo de mensaje, el PID y la cadena
    struct mensaje msg;
    msg.tipo = 2;
    msg.pid = getpid();
    strcpy(msg.state, estado);

    printf("Hijo %d envia mensaje: Tipo - %ld, Estado - %s\n", msg.pid, msg.tipo, msg.state);
    fflush(stdout);

    // Enviar el mensaje a la cola
    if (msgsnd(mensajes, &msg, sizeof(struct mensaje) - sizeof(long), 0) == -1) {
        perror("Hijo: msgsnd");
        exit(-1);
    }
}
```

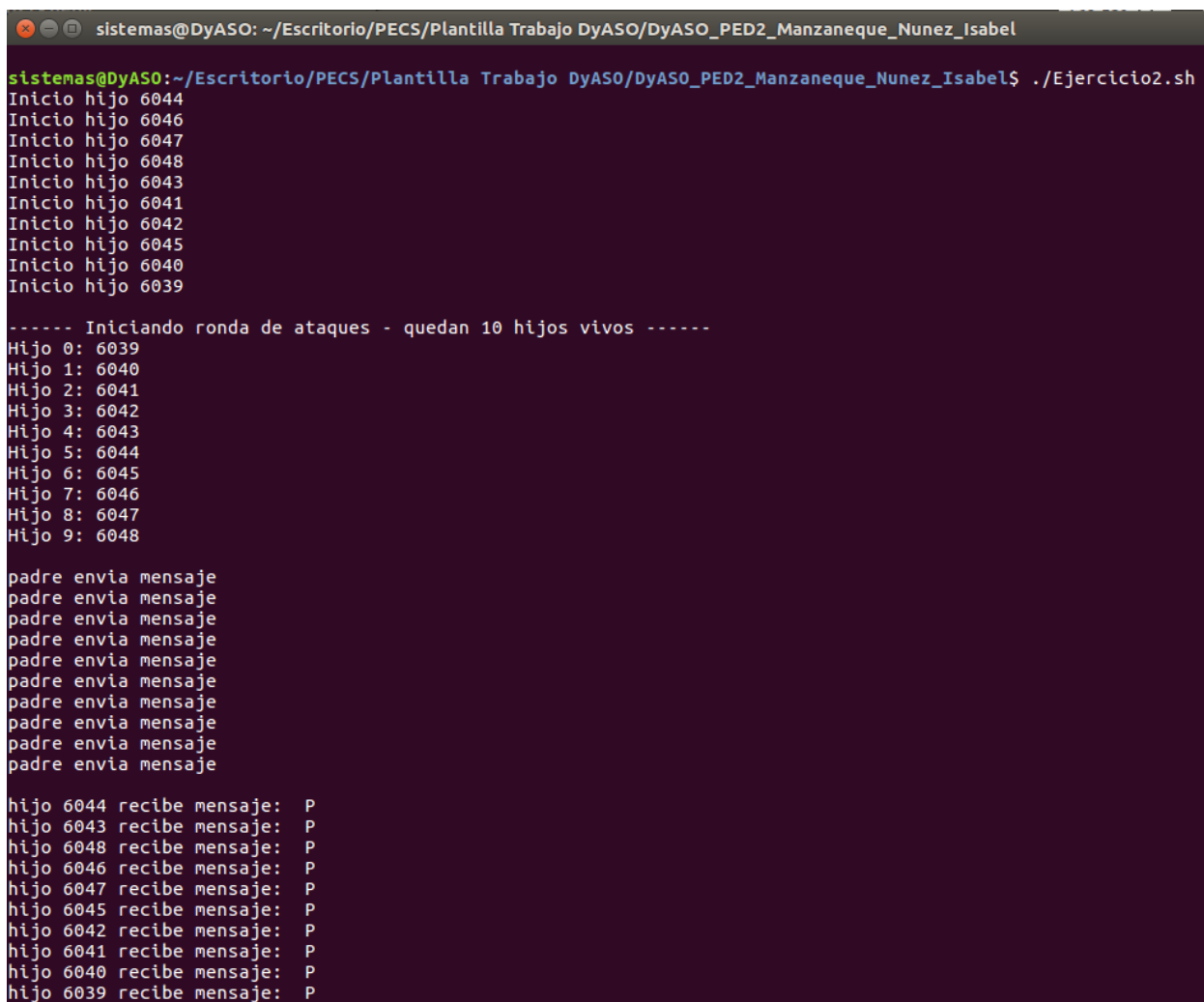
La función **enviarMensajeAPadre** prepara y envía un mensaje para el padre. Comienza configurando el mensaje con el tipo, el PID del proceso que envía el mensaje y el estado. Muestra por pantalla el mensaje que esta enviando y por último utiliza ‘msgsnd’ para enviarlo a la cola de mensajes que leerá el padre.

Ejecución de ejemplo

Comenzamos la ejecución. En la siguiente captura de pantalla podemos observar como el padre, tras realizar la inicialización de los mecanismos IPC, ha creado 10 procesos hijos utilizando la función ‘crearNHijos’ y estos han avisado de que han sido iniciados.

El padre entra en su bucle principal y comienza la primera ronda. Nos informa de cuántos procesos quedan vivos y nos muestra el estado de la lista de procesos, donde vemos el PID de cada uno de los hijos.

El padre nos informa de que envía un mensaje por cada hijo vivo y espera a que todos los hijos lo reciban. Los hijos, al recibir el mensaje del padre, nos confirman su recepción.



```
sistemas@DyASO: ~/Escritorio/PECS/Plantilla Trabajo DyASO/DyASO_PED2_Manzaneque_Nunez_Isabel
sistemas@DyASO:~/Escritorio/PECS/Plantilla Trabajo DyASO/DyASO_PED2_Manzaneque_Nunez_Isabel$ ./Ejercicio2.sh
Inicio hijo 6044
Inicio hijo 6046
Inicio hijo 6047
Inicio hijo 6048
Inicio hijo 6043
Inicio hijo 6041
Inicio hijo 6042
Inicio hijo 6045
Inicio hijo 6040
Inicio hijo 6039

----- Iniciando ronda de ataques - quedan 10 hijos vivos -----
Hijo 0: 6039
Hijo 1: 6040
Hijo 2: 6041
Hijo 3: 6042
Hijo 4: 6043
Hijo 5: 6044
Hijo 6: 6045
Hijo 7: 6046
Hijo 8: 6047
Hijo 9: 6048

padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje

hijo 6044 recibe mensaje: P
hijo 6043 recibe mensaje: P
hijo 6048 recibe mensaje: P
hijo 6046 recibe mensaje: P
hijo 6047 recibe mensaje: P
hijo 6045 recibe mensaje: P
hijo 6042 recibe mensaje: P
hijo 6041 recibe mensaje: P
hijo 6040 recibe mensaje: P
hijo 6039 recibe mensaje: P
```

En la siguiente captura de pantalla, podemos ver como los procesos hijo comienzan a atacarse. En esta ronda, podemos ver como el hijo 6047 es atacado por varios procesos y emboscado varias veces mientras realizaba un ataque (el proceso padre aún no había tenido oportunidad de eliminarlo). El hijo 6048 es también emboscado mientras realizaba un ataque.

Tanto el hijo 6047 como el 6048 mandan al proceso padre un mensaje con el estado a “KO”, mientras que, como se puede observar en la captura, todos los demás comunican al padre que su estado es “OK”.

```

6048 Atacando al proceso 6047
6043 Atacando al proceso 6047
6044 Atacando al proceso 6047
6046 Atacando al proceso 6047
El hijo 6047 ha sido emboscado mientras realizaba un ataque

6047 Atacando al proceso 6048
6045 Atacando al proceso 6047
El hijo 6047 ha sido emboscado mientras realizaba un ataque
Hijo 6047 envia mensaje: Tipo - 2, Estado - KO

El hijo 6048 ha sido emboscado mientras realizaba un ataque
Hijo 6048 envia mensaje: Tipo - 2, Estado - KO

6042 Atacando al proceso 6047
El hijo 6047 ha sido emboscado mientras realizaba un ataque

6041 Atacando al proceso 6047
El hijo 6047 ha sido emboscado mientras realizaba un ataque

6040 Atacando al proceso 6047
El hijo 6047 ha sido emboscado mientras realizaba un ataque

El hijo 6047 ha sido emboscado mientras realizaba un ataque

6039 Atacando al proceso 6047Hijo 6043 envia mensaje: Tipo - 2, Estado - OK
Hijo 6044 envia mensaje: Tipo - 2, Estado - OK
Hijo 6046 envia mensaje: Tipo - 2, Estado - OK
Hijo 6045 envia mensaje: Tipo - 2, Estado - OK
Hijo 6042 envia mensaje: Tipo - 2, Estado - OK
Hijo 6041 envia mensaje: Tipo - 2, Estado - OK
Hijo 6040 envia mensaje: Tipo - 2, Estado - OK
Hijo 6039 envia mensaje: Tipo - 2, Estado - OK

```

El padre nos comunica los mensajes los resultados que ha recibido por parte de los hijos. Dos de ellos están en estado “KO” y se encargará de eliminarlos, mientras que los ocho restantes continuarán a la siguiente ronda.

```

Padre recibe mensaje de 6047: Tipo: 2 - Estado: KO
Padre recibe mensaje de 6048: Tipo: 2 - Estado: KO
Padre recibe mensaje de 6043: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6044: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6046: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6045: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6042: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6041: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6040: Tipo: 2 - Estado: OK
Padre recibe mensaje de 6039: Tipo: 2 - Estado: OK

```

Comienza la segunda ronda. El padre comienza informándonos de que quedan 8 hijos vivos y nos muestra la lista de PIDs en la que puede apreciarse que los hijos 8 y 9 tienen ahora su PID a 0. Se nos muestra de nuevo como el padre envía un mensaje por cada hijo vivo y los hijos vivos confirman su recepción:

```

----- Iniciando ronda de ataques - quedan 8 hijos vivos -----
Hijo 0: 6039
Hijo 1: 6040
Hijo 2: 6041
Hijo 3: 6042
Hijo 4: 6043
Hijo 5: 6044
Hijo 6: 6045
Hijo 7: 6046
Hijo 8: 0
Hijo 9: 0

padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje
padre envia mensaje

hijo 6043 recibe mensaje: P
hijo 6044 recibe mensaje: P
hijo 6046 recibe mensaje: P
hijo 6045 recibe mensaje: P
hijo 6042 recibe mensaje: P
hijo 6041 recibe mensaje: P
hijo 6040 recibe mensaje: P
hijo 6039 recibe mensaje: P

```

En esta ronda de ataques tenemos bastantes emboscadas. Los hijos 6042, 6045, 6046, 6043 y 6041 son emboscados mientras realizaban ataques y mandan al padre un mensaje con su estado y este confirma la recepción. Los tres procesos restantes mandan al padre un mensaje informando de que están “OK” y el padre lo confirma.

```
6043 Atacando al proceso 6042
El hijo 6042 ha sido emboscado mientras realizaba un ataque

6042 Atacando al proceso 6045
El hijo 6045 ha sido emboscado mientras realizaba un ataque

6045 Atacando al proceso 6042
El hijo 6042 ha sido emboscado mientras realizaba un ataque
Hijo 6042 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6042: Tipo: 2 - Estado: KO

6046 Atacando al proceso 6045
6044 Atacando al proceso 6045
El hijo 6045 ha sido emboscado mientras realizaba un ataque
Hijo 6045 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6045: Tipo: 2 - Estado: KO

6041 Atacando al proceso 6046
El hijo 6046 ha sido emboscado mientras realizaba un ataque
Hijo 6046 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6046: Tipo: 2 - Estado: KO

6040 Atacando al proceso 6043
El hijo 6043 ha sido emboscado mientras realizaba un ataque
Hijo 6043 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6043: Tipo: 2 - Estado: KO

6039 Atacando al proceso 6041
El hijo 6041 ha sido emboscado mientras realizaba un ataque
Hijo 6041 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6041: Tipo: 2 - Estado: KO
Hijo 6044 envia mensaje: Tipo - 2, Estado - OK
Padre recibe mensaje de 6044: Tipo: 2 - Estado: OK
Hijo 6040 envia mensaje: Tipo - 2, Estado - OK
Padre recibe mensaje de 6040: Tipo: 2 - Estado: OK
Hijo 6039 envia mensaje: Tipo - 2, Estado - OK
Padre recibe mensaje de 6039: Tipo: 2 - Estado: OK
```

Tras eliminar los procesos que se encontraban “KO”, el padre comienza una tercera ronda. Ya tan solo quedan 3 procesos vivos como podemos ver en la captura de pantalla a continuación. En esta ronda sobreviven todos.

```
----- Iniciando ronda de ataques - quedan 3 hijos vivos -----
Hijo 0: 6039
Hijo 1: 6040
Hijo 2: 0
Hijo 3: 0
Hijo 4: 0
Hijo 5: 6044
Hijo 6: 0
Hijo 7: 0
Hijo 8: 0
Hijo 9: 0

padre envia mensaje
padre envia mensaje
padre envia mensaje

hijo 6044 recibe mensaje: P
hijo 6040 recibe mensaje: P
hijo 6039 recibe mensaje: P
Hijo 6044 envia mensaje: Tipo - 2, Estado - OK
Padre recibe mensaje de 6044: Tipo: 2 - Estado: OK
Hijo 6040 envia mensaje: Tipo - 2, Estado - OK
Padre recibe mensaje de 6040: Tipo: 2 - Estado: OK
Hijo 6039 envia mensaje: Tipo - 2, Estado - OK
Padre recibe mensaje de 6039: Tipo: 2 - Estado: OK
```


En esta última ronda volvemos a ver a nuestros 3 supervivientes que esta vez sí se atacan los unos a los otros. Los 3 atacan a sus hermanos, por lo que todos son emboscados y ninguno sobrevive a la ronda, habiendo ocurrido un empate.

Este resultado es leído por el proceso cat en segundo plano y se muestra al final de la captura de pantalla. Además, he decidido mostrar también el identificador de el grupo de semáforos (semid) y el de la cola de mensajes (msqid) porque esto nos va a facilitar el saber que identificadores no deberían estar en las listas de recursos IPC activos.

```
----- Iniciando ronda de ataques - quedan 3 hijos vivos -----
Hijo 0: 6039
Hijo 1: 6040
Hijo 2: 0
Hijo 3: 0
Hijo 4: 0
Hijo 5: 6044
Hijo 6: 0
Hijo 7: 0
Hijo 8: 0
Hijo 9: 0

padre envia mensaje
padre envia mensaje
padre envia mensaje

hijo 6044 recibe mensaje: P
hijo 6040 recibe mensaje: P
hijo 6039 recibe mensaje: P

6044 Atacando al proceso 6039
6040 Atacando al proceso 6044
El hijo 6044 ha sido emboscado mientras realizaba un ataque
Hijo 6044 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6044: Tipo: 2 - Estado: KO

El hijo 6039 ha sido emboscado mientras realizaba un ataque

6039 Atacando al proceso 6040
El hijo 6040 ha sido emboscado mientras realizaba un ataque
Hijo 6040 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6040: Tipo: 2 - Estado: KO
Hijo 6039 envia mensaje: Tipo - 2, Estado - KO
Padre recibe mensaje de 6039: Tipo: 2 - Estado: KO

semid : 6946973 - msqid: 7667872

Recursos IPC activos:

Empate
```

En cuanto a estas listas, son ambas muy largas y solamente voy a adjuntar capturas de pantalla de sus inicios para evitar redundancias, aunque puedo confirmar que tras buscar línea por línea, no he encontrado los identificadores de los recursos IPC que se estaban utilizando.

Por último, el proceso padre termina y nos informa de ello también.

semid : 6946973 - msqid: 7667872

Recursos IPC activos:

Empate

----- Colas de mensajes -----

key	msqid	propietario	perms	bytes utilizados	mensajes
0x580117ae	0	sistemas	600	0	0
0x580117b0	32769	sistemas	600	0	0
0x580117b3	65538	sistemas	600	0	0
0x580117b4	98307	sistemas	600	0	0
0x580117b6	131076	sistemas	600	0	0
0x580117b7	163845	sistemas	600	0	0
0x580117b8	196614	sistemas	600	0	0
0x580117b9	229383	sistemas	600	0	0
0x580117ba	262152	sistemas	600	0	0
0x580117bb	294921	sistemas	600	0	0
0x580117bf	327690	sistemas	600	0	0
0x580117c1	360459	sistemas	600	0	0
0x580117c2	393228	sistemas	600	0	0
0x580117c4	425997	sistemas	600	0	0
0x580117c5	458766	sistemas	600	0	0
0x580117c6	491535	sistemas	600	0	0
0x580117c7	524304	sistemas	600	0	0
0x580117c8	557073	sistemas	600	0	0
0x580117c9	589842	sistemas	600	0	0
0x580117cd	622611	sistemas	600	0	0
0x580117ce	655380	sistemas	600	0	0
0x580117cf	688149	sistemas	600	0	0
0x580117d0	720918	sistemas	600	0	0
0x580117d1	753687	sistemas	600	0	0
0x580117d2	786456	sistemas	600	0	0
0x580117d3	819225	sistemas	600	0	0
0x580117d4	851994	sistemas	600	0	0
0x580117d5	884763	sistemas	600	0	0
0x580117d6	917532	sistemas	600	0	0
0x580117d9	950301	sistemas	600	0	0
0x580114e7	983070	sistemas	600	0	0
0x580114e9	1015839	sistemas	600	0	0
0x580114eb	1048608	sistemas	600	0	0
0x5801150c	1081377	sistemas	600	0	0
0x58011548	1114146	sistemas	600	0	0
0x58011633	1146915	sistemas	600	0	0
0x580116b1	1179684	sistemas	600	0	0
0x580116b2	1212453	sistemas	600	0	0
0x580116ea	1245222	sistemas	600	0	0

----- Matrices semáforo -----

key	semid	propietario	perms	nsens
0x580117ae	0	sistemas	600	1
0x580117b0	32769	sistemas	600	1
0x580117b3	65538	sistemas	600	1
0x580117b4	98307	sistemas	600	1
0x580117b6	131076	sistemas	600	1
0x580117b7	163845	sistemas	600	1
0x580117b8	196614	sistemas	600	1
0x580117b9	229383	sistemas	600	1
0x580117ba	262152	sistemas	600	1
0x580117bb	294921	sistemas	600	1
0x580117bf	327690	sistemas	600	1
0x580117c1	360459	sistemas	600	1
0x580117c2	393228	sistemas	600	1
0x580117c4	425997	sistemas	600	1
0x580117c5	458766	sistemas	600	1
0x580117c6	491535	sistemas	600	1
0x580117c7	524304	sistemas	600	1
0x580117c8	557073	sistemas	600	1
0x580117c9	589842	sistemas	600	1
0x580117cd	622611	sistemas	600	1
0x580117ce	655380	sistemas	600	1
0x580117cf	688149	sistemas	600	1
0x580117d0	720918	sistemas	600	1
0x580117d1	753687	sistemas	600	1
0x580117d2	786456	sistemas	600	1
0x580117d3	819225	sistemas	600	1
0x580117d4	851994	sistemas	600	1
0x580117d5	884763	sistemas	600	1
0x580117d6	917532	sistemas	600	1
0x580117d9	950301	sistemas	600	1
0x580114e7	983070	sistemas	600	1
0x580114e9	1015839	sistemas	600	1
0x580114eb	1048608	sistemas	600	1
0x5801150c	1081377	sistemas	600	1
0x58011548	1114146	sistemas	600	1
0x58011633	1146915	sistemas	600	1
0x580116b1	1179684	sistemas	600	1
0x580116b2	1212453	sistemas	600	1
0x580116ea	1245222	sistemas	600	1
0x58011711	1277991	sistemas	600	1
0x58011712	1310760	sistemas	600	1
0x58011714	1343529	sistemas	600	1
0x5801171f	1376298	sistemas	600	1
0x58011720	1409067	sistemas	600	1
0x58011739	1441836	sistemas	600	1
0x5801176c	1474605	sistemas	600	1
0x5801176d	1507374	sistemas	600	1

Conclusiones

Este proyecto ha representado un desafío significativo, pero al mismo tiempo, una muy buena oportunidad para expandir mi comprensión de la programación en C y la programación multiproceso.

Mi experiencia en C previa a la práctica era prácticamente nula y esto ha hecho que tenga que hacer un esfuerzo extra en aprender la sintaxis y el funcionamiento de este lenguaje. Además, el tema en concreto de la práctica (mecanismos IPC, gestión de recursos compartidos, programación multiproceso etc) es también muy nuevo, pues son temas que había explorado de manera teórica en asignaturas como Sistemas Operativos y Sistemas en Tiempo Real, pero nunca había implementado de una manera práctica.

La combinación de mi inexperiencia del lenguaje con mi inexperiencia del tema ha resultado muchas horas atascada delante de la pantalla arreglando bugs e intentando adivinar por qué ciertos componentes se comportaban de manera extraña. Si bien esto me ha resultado en ocasiones frustrante, siento que realizar la práctica ha beneficiado enormemente mi comprensión sobre conceptos que me resultaban muy abstractos y desconocidos.

Por todo lo anterior, me ha resultado una práctica muy interesante y, aunque me ha supuesto más esfuerzo del que me gustaría reconocer, creo que ha valido mucho la pena.

Bibliografía.

Fundamentos del sistema operativo UNIX, José Manuel Díaz - Rocío Muñoz Mansilla - Dictino Chaos García

<https://stackoverflow.com/>

<https://devdocs.io/c/>

<https://developerinsider.co/compile-c-program-with-gcc-compiler-on-bash-on-ubuntu-on-windows-10/>

<https://www.cs.kent.edu/~ruttan/sysprog/lectures/shmem/semaphore.html>

<https://www.geeksforgeeks.org/ipc-using-message-queues/>

<https://tldp.org/LDP/lpg/node11.html>

<https://www.geeksforgeeks.org/ipcs-command-linux-examples/>

<https://www.cs.cmu.edu/~410/doc/doxygen.html>