UA

Universidad
de Alicante

ESCUELA POLITÉCNICA SUPERIOR

SISTEMAS OPERATIVOS Y DISTRIBUIDOS

Curso académico 2024 / 2025
Semestre 1
Grado en Ingeniería en Inteligencia Artificial
Grupo 1

Profesora: Iren Lorenzo Fonseca
Alumno: Santiago Álvarez Geanta

Universitat d'Alacant
Universidad de Alicante

Septiembre - Octubre 2024

ÍNDICE

Ejercicio 1

1.1 Filtrar Procesos Activos con Mayor Consumo de Memoria

Para realizar este primer ejercicio deberemos crear un script en **Bash (.sh)** que deberá listar todos los procesos activos en el sistema y deberá mostrar los que más memoria consumen.

Las instrucciones son las siguientes:
1. Creación del script en Bash ***consumo_proc.sh***
2. Abrir el archivo en un editor de texto
3. Utilizar el comando `***ps***` para listar los procesos.
4. Ordenar los procesos por **uso de memoria**.
5. Mostrar los N procesos que **más memoria consumen**, donde N es un número configurable.
6. Dar **permisos de ejecución** al script

Explicación
Comenzaremos con la creación del script en Bash, en mi caso utilicé VisualStudio Code para escribir el código (podemos utilizar un editor de texto corriente). En este script usaremos la instrucción **ps** para mostrar los procesos. Si queremos ver aquellos procesos activos usaremos seguidamente **aux** y para ordenar por memoria utilizaremos la instrucción **--sort=** seguida por el tipo de ordenamiento, **%mem** en nuestro caso. Si quisiéramos ordenar de forma descendente usamos **-%mem**. Por último, para mostrar los **N** procesos que más memoria consumen utilizaremos **head -n 10**, donde head es el principio de la lista ordenada y **-n 10** representa la cantidad de **N** procesos a mostrar.

```
consumo_proc.sh
1   ps aux --sort=%mem | head -n 10
```

Imagen 1.1

En la *Imagen 1.2* podemos ver el procedimiento seguido para ejecutar el script utilizando **chmod +x consumo_proc.sh** para otorgar permisos de ejecución.

```
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas Operativos$ chmod +x consumo_proc.sh

sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas Operativos$ ./consumo_proc.sh
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           2  0.0  0.0      0     0 ?        S    12:31   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    12:31   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   12:31   0:00 [kworker/R-rcu_g]
root           5  0.0  0.0      0     0 ?        I<   12:31   0:00 [kworker/R-rcu_p]
root           6  0.0  0.0      0     0 ?        I<   12:31   0:00 [kworker/R-slub_]
root           7  0.0  0.0      0     0 ?        I<   12:31   0:00 [kworker/R-netns]
root          10  0.0  0.0      0     0 ?        I<   12:31   0:00 [kworker/0:0H-events_highpri]
root          12  0.0  0.0      0     0 ?        I<   12:31   0:00 [kworker/R-mm_pe]
root          13  0.0  0.0      0     0 ?        I    12:31   0:00 [rcu_tasks_kthread]
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas Operativos$
```

Imagen 1.2

1.2 Contar archivos y directorios en un directorio

En este segundo ejercicio deberemos contar cuantos directorios y archivos contiene un directorio dado como parámetro. Para ello utilizaremos de nuevo un archivo en **Bash (.sh)**.

Las instrucciones son las siguientes:
1. Creación del script en Bash ***contar_archivos.sh***
2. Abrir el archivo en un editor de texto
3. Usa el comando **find** para encontrar todos los archivos y directorios en el directorio
4. Usar **pipe** para pasarlo al comando **wc** para contarlos
5. Dar **permisos de ejecución** al script

Explicación
Para encontrar todos los archivos y directorios en el directorio que el usuario introduzca usaremos el comando **find**. A continuación especificaremos la ruta del directorio escribiendo **$1** que representa el primer registro o campo que el usuario introduce luego de ejecutar el script en Bash. Además, tendremos que especificar que tipo de información queremos buscar dentro de esta ruta, para ello usaremos **-type f** (para buscar **files/ficheros)** o (**-o) -type d (**para buscar **directories/directorios).** De esta forma hemos obtenido todos los ficheros y carpetas que contiene la ruta, pero para contar cuántos hay, tendremos que utilizar un pipe **(|)** junto al comando **wc (word count)** para contar cuántas líneas (utilizando **-l**), o lo que vendrían siendo ficheros y carpetas, contiene el directorio. Finalmente, utilizando el comando **echo,** mostraremos por la terminal el número de directorios y ficheros que hemos guardado en una variable / registro **$** (observar *Imagen 1.3)*.

```
conteo=$(find "$1" -type f -o -type d | wc -l)

echo "El número total de archivos y directorios en '$1' es: $conteo"
```

Imagen 1.3

En la *Imagen 1.4* podemos ver el procedimiento seguido para ejecutar el script utilizando **chmod +x contar_archivos.sh** para otorgar permisos de ejecución.

```
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sis
temas Operativos/P1/e1$ chmod +x contar_archivos.sh
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sis
temas Operativos/P1/e1$ ./contar_archivos.sh /home/sant_vz6/
Escritorio/UA/Sistemas_Operativos/P1
El número total de archivos y directorios en '/home/sant_vz6
/Escritorio/UA/Sistemas_Operativos/P1' es: 15
```

Imagen 1.4

Ejercicio 2. Gestión Básica de Procesos

Enunciado

Realiza un programa llamado **ejec.c** que reciba dos argumentos. El programa tendrá que generar el árbol de procesos que se indica y llevar a cabo la funcionalidad que se describe a continuación. El proceso Z, transcurridos los segundos indicados por el segundo argumento, le ordenará mediante una señal al proceso identificado con el primer argumento (A, B, X, Y) que ejecute una orden. Si la señal la recibe el proceso A o B ejecutarán el comando "pstree", Si la señal la recibe el proceso X o Y ejecutarán el comando "ls". El proceso Z no puede utilizar la syscall "sleep" para realizar la espera. Se deberá controlar la correcta destrucción del árbol (los padres no pueden morir antes que los hijos).

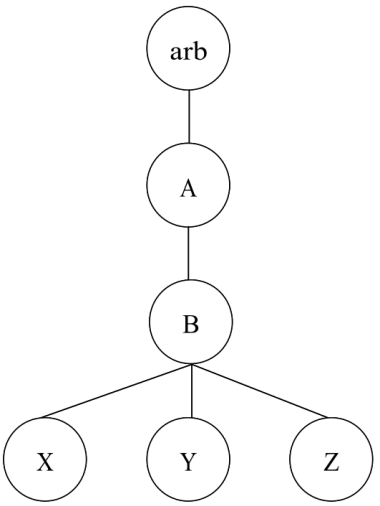

Imagen 2.1

Desarrollo

Para la elaboración de nuestro script, tendremos que utilizar principalmente el método **fork()**. Este método crea un subproceso (proceso hijo) en memoria a partir del proceso padre. Ambos procesos serán idénticos en cuánto a instrucciones, ya que fueron clonados y comparten el mismo **PC**. Normalmente identificaremos a nuestro subproceso con el nombre **pid** con tipo de dato **___pid_t,** observar *Imagen 2.2* Esta nueva variable podrá tomar 3 tipos de valores distintos, en caso de que su pid sea **-1**, no se habrá podido crear el subproceso; en caso de que su pid sea **0**, se habrá creado el subproceso y, además, estaríamos ejecutando al proceso hijo; en caso de que su valor sea **>0**, estaríamos ejecutando al proceso padre (observar *Imagen 2.3)*

```
// fork() > crea un proceso hijo que se ejecuta en paralelo con el proceso padre.
// El proceso padre es el hueco reservado en RAM para ejecutar las instrucciones del script.
// Al crar un proceso hijo se crea otro hueco en la RAM

__pid_t pid; // declaramos la variable pid con el tipo de dato: __pid_t
pid = fork();
```

Imagen 2.2

```
switch (pid){
    // Ha fallado el fork;
    case -1:
        printf ("No he podido crear el proceso hijo A\n");
        break;

    // Instrucciones para el hijo A;
    case 0:
        printf ("Soy el hijo A, mi PID es %d y mi PPID es %

        __pid_t pidA;
        pidA = fork(); // El hijo A se transforma en Padre

    // Instrucciones para el padre R;
    default:
        printf ("Soy el padre R, mi PID es %d y el PID de mi
        // sleep (30); Muy impreciso dado que no sé cuando
        wait(NULL); // usamos wait para esperar de forma ex
};
```

Imagen 2.3

Teniendo esto en cuenta ya podemos crear cualquier distribución de hijos, padres, abuelos, etc. De tal forma que estaremos creando un árbol de procesos.

Creación del Código

Para desarrollar este ejercicio utilizaremos el lenguaje de programación **C**. Por tanto, incluiremos las siguientes cuatro librerías (observar *Imagen 2.4*).

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#include <signal.h>
```

Imagen 2.4

Además, declararemos las siguientes variables globales, para posteriormente utilizarlas en los handler correspondientes (observar *Imagen 2.5*).

```c
__pid_t pidA; // declara
__pid_t pidB;
__pid_t pidX;
__pid_t pidY;
__pid_t pidZ;

char proceso;
unsigned int seconds;
```

Imagen 2.5

Respecto a nuestra función **main()**, esta recibirá dos argumentos válidos de nuestro puntero **argv.**

Si nos centramos en el interior de nuestra función , aquí realizaremos todas las llamadas a **fork()** necesarias, recordemos que nuestra variable pid podrá tomar 3 valores distintos por lo que utilizaremos un switch-case para facilitar el código. De momento nos centraremos en la parte dónde pid toma el **valor 0** al realizar la llamada a fork**()**. En esta parte tendremos la ejecución de nuestro subproceso. Por tanto, si queremos que este nuevo subproceso se convierta en padre, tendremos que utilizar nuestro segundo fork() con nombre **pidA** dentro del código reservado al subproceso **pid(case 0)** (observar *Imagen 2.6*).

```c
// Instrucciones para el hijo A;
case 0:
    printf ("Soy el hijo A, mi PID es %d y mi PPID es %d\n", getpid(), getppid());

    __pid_t pidA;
    pidA = fork(); // El hijo A se transforma en Padre de B
```

Imagen 2.6

Hasta el momento hemos hecho dos llamadas a fork(). La primera ,desde el proceso padre R para crear al hijo A; y la segunda, desde el proceso hijo A (ya que estamos en el case 0) para crear al hijo . Por lo tanto, ahora A pasará a ser padre del nuevo subproceso B.

En el último caso se nos pide crear tres subprocesos desde un mismo padre. Podremos crear 3 hijos desde B realizando 3 llamadas independientes al fork(), o podremos realizar la llamada utilizando la misma variable pid dentro de un bucle for. En mi caso, utilicé el bucle for para comprender mejor el funcionamiento del código (observar *Imagen 2.7*).

```c
case 0:
    printf ("Soy el hijo B, mi PID es %d y mi PPID es %d\n", getpid(), getppid());
    for (int i=0; i<3; i++){
        __pid_t pidB;
        pidB = fork(); // El hijo B se transforma en Padre de XYZ
```

Imagen 2.7

El funcionamiento a seguir será el siguiente: Nuestro iterador llamará en su primera iteración al método fork(), este creará un subproceso (llamémosle X). Al igual que antes tendremos los tres tipos de casos (-1, 0, >0), el añadido del problema vendrá dado por el valor de la **variable iteradora (i)** en ese momento. En caso de que i=0, estaremos ante el primer hijo de B (subproceso X); en caso de que i=1, nuestro iterador habrá realizado su segunda llamada a fork() (hijo Y); y por último cuando i=2, nuestro iterador habrá realizado la tercera llamada a fork().

En este punto habremos realizado todas las correspondientes creaciones de subprocesos para imitar el árbol de procesos (observar *Imagen 2.1*). Es importante mencionar que cuando un subproceso haya realizado sus instrucciones este será eliminado para que así no ejecute el código de su padre. A esta acción se le llama "muerte de procesos", para realizar la acción en **C** utilizaremos el método **exit(0)** (observar *Imagen 2.8*).

```
        case 1:
            printf ("Soy el hijo Y, mi PID es %d y mi PPID es %d\n", getpid(), getppid());
            exit (0); // El hijo no ejecuta la parte del padre
        case 2:
            printf ("Soy el hijo Z, mi PID es %d y mi PPID es %d\n", getpid(), getppid());
            exit (0); // El hijo no ejecuta la parte del padre
    };
```

Imagen 2.8

Métodos y señales utilizadas

En primer lugar, el ejercicio restringe utilizar la función **wait()** para esperar los segundos del segundo parámetro que introduce el usuario. Es por ello que utilizaremos una señal de alarma que tendrá que esperar la cantidad que el usuario introdujo como parámetro en **argv.** Una vez haya transcurrido ese tiempo se enviará la señal a **signal(SIGALARM, handler_z)**, que quedó en espera gracias a **pause()**. Esta línea de código espera una señal de tipo SIGALARM y cuando la recibe ejecuta la función correspondiente, en nuestro caso el handler de z (observar *Imagen 2.9*).

```
signal(SIGALRM, handler_z);
alarm(seconds);
pause();
```

Imagen 2.9

Además, el ejercicio nos pide que en base al primer parámetro se ejecute una acción en el sistema. Si este parámetro hace referencia al proceso "A" o "B", se ejecutará la instrucción "pstree"; en cambio si hace referencia a "X" o "Y", se ejecutará la instrucción "ls". De nuevo, utilizaremos una señal junto a un pause() que se quedará en espera hasta recibir una señal **SIGUSR1** (en caso de llamar al

```
void handler_z(){

    printf("Ejecutando handler_z\n");

    switch(toupper(proceso)){
        case 'A':
            printf("Señal a A");
            kill(pidA, SIGUSR1);
            break;
        case 'B':
        printf("Señal a B");
            kill(pidB, SIGUSR1);
            break;
        case 'X':
            printf("Señal a X");
            kill(pidX, SIGUSR2);
            break;
        case 'Y':
            printf("Señal a Y");
            kill(pidY, SIGUSR2);
            break;
        default:
            printf("No se ha seleccionado un proceso válido\n");

    };
};
```

Imagen 2.10

proceso “A” o “B”) o una señal **SIGUSR2** (en caso de esperar al proceso “X” o “Y”) (observar *Imagen 2.10*).

Una vez el handler_z haya envíado la correspondiente señal **SIGUSR**(que vendrá dada por los parámetros del usuario), se ejecutará el **handler_ab** o el **handler_xy** (observar *Imagen 2.11)*

```
signal(SIGUSR2, handler_xy);
pause(); //De esta forma se queda en la lista de bloqueados
 signal(SIGUSR1, handler_ab);
 pause();
```

Imagen 2.11

Nuestros handler de los procesos tendrán la siguiente forma (observar *Imagen 2.12*). Esto se debe a que cuando ejecutamos una instrucción **execlp,** estamos sustituyendo los datos de nuestro proceso por los datos de la instrucción execlp (**pstree** o **ls**, en nuestro caso). Por tanto, la solución será crear un nuevo proceso hijo en el que ejecutaremos la instrucción **execlp** y que posteriormente será eliminado.

```
void handler_ab(){
    printf("Ejecutando handler_ab\n");
    __pid_t pidExec;
    pidExec = fork();
    switch(pidExec){
        case -1:
            printf ("No he podido crear el proceso hijo A\n");
            break;
        case 0:
            printf("Soy execlp(%d) y muero", getpid());
            execlp("pstree", "pstree", NULL);
    }
};
```

Imagen 2.12

Por último, tendremos que eliminar todos aquellos procesos hijos y posteriormente sus procesos padre. Es muy importante seguir este orden para que ningún proceso hijo quede sin eliminar.

Comparación de resultados

Resultado Esperado

```
$ ejec A 15
Soy el proceso ejec: mi pid es 751
Soy el proceso A: mi pid es 752. Mi padre es 751
Soy el proceso B: mi pid es 753. Mi padre es 752. Mi abuelo es 751
Soy el proceso X: mi pid es 754. Mi padre es 753. Mi abuelo es 752. Mi bisabuelo es 751
Soy el proceso Y: mi pid es 755. Mi padre es 753. Mi abuelo es 752. Mi bisabuelo es 751
Soy el proceso Z: mi pid es 756. Mi padre es 753. Mi abuelo es 752. Mi bisabuelo es 751 /*Tras un intervalo de 15
segundos aparecerá*/ Soy el proceso A con pid 752, he recibido la señal.
/*Resultado del comando*/
Soy Z (756) y muero
Soy Y (755) y muero
Soy X (754) y muero
Soy B (753) y muero
Soy A (752) y muero
Soy ejec (751) y muero
```

Resultado Obtenido

```
Soy el padre R, mi PID es 6918 y el PID de mi hijo A es 6919
Soy el hijo A, mi PID es 6919 y mi padre es 6918
Soy el padre A, mi PID es 6919 y el PID de mi hijo B es 6920
Soy el hijo B, mi PID es 6920, mi padre es 6919 y mi abuelo es 6918
Soy el padre B, mi PID es 6920 y el PID de mi hijo X es 6921
Soy el hijo X, mi PID es 6921, mi padre es 6920, mi abuelo es 0 y mi bisabuelo e
s 6918
Soy el padre B, mi PID es 6920 y el PID de mi hijo Y es 6922
Soy el hijo Y, mi PID es 6922, mi padre es 6920, mi abuelo es 0 y mi bisabuelo e
s 6918
Soy el padre B, mi PID es 6920 y el PID de mi hijo Z es 6923
Soy el hijo Z, mi PID es 6923, mi padre es 6920, mi abuelo es 0 y mi bisabuelo e
s 6918
Ejecutando handler_z
Ejecutando handler_xy, (6921)
Ejecutando handler_xy, (6922)
Soy Y(6922) y muero
Soy X(6921) y muero
Señal definida por el usuario 2
Soy execlp(6925) y muero
Soy execlp(6924) y muero
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas_Operativos/P1/
e2$ C1.png      C2.png  C3.png  P  P1  Practica1.c  Practica1.pdf
```

Ejercicio 3. COMUNICACIÓN ENTRE PROCESOS: TUBERÍAS

Enunciado

Realizar un programa llamado copiar.c que permita copiar archivos. El programa recibirá dos argumentos:
archivo_origen y **archivo_destino**, el archivo origen debe existir y el archivo destino se creará. El proceso copiar (proceso padre) generará un proceso hijo y, a partir de ese momento, el proceso padre será el encargado de leer el archivo origen y enviarle la información al proceso hijo que la almacenará en el archivo de destino. La comunicación entre los dos procesos se realizará mediante **tuberías** (pipe).

Desarrollo

Para implementar el siguiente programa será necesario conocer el funcionamiento que nuestra tubería utilizará.

En primer lugar, el proceso padre tendrá que leer el archivo Origen.txt, en caso de que no exista no se podrá continuar la ejecución. Una vez el padre haya leído el contenido de Origen.txt, escribirá el contenido en el extremo de escritura de nuestra tubería **pipe[1].**

En segundo lugar, ahora que el padre ha escrito lo necesario en la tubería, el hijo será el encargado de abrir el archivo Destino.txt (en caso de que no exista se creará uno). Una vez abierto el archivo Destino.txt, se procederá a leer el contenido de la tubería para posteriormente escribirlo en el archivo Destino.txt.

Código

Las librerías específicas que utilizaremos para ese ejercicio son las siguientes (observar *Imagen 3.1*).

```c
#include <fcntl.h> // Read
#include <string.h> // Write
```

Imagen 3.1

También declararemos las variables globales de:

- la tubería (pipe(**pipe_v)**)
- el pid (pid)
- el tamaño del BUFFER (BUFFER_SIZE)
- el puntero a cadena del buffer (BUFFER[BUFFER_SIZE])
- el almacenamiento de los elemento leídos (bytesLeidos).

Nuestra función **main**() se encargará de la ejecución principal y recibirá dos argumentos, para ello utilizaremos **argc** y **argv**. Para la ejecución principal nos encargaremos de crear un proceso hijo mediante la llamada a **fork()**. Utilizaremos la misma sintaxis del **switch-case** utilizada en el ejercicio anterior y en la parte del padre abriremos el archivo Origen, usando **open()**; leeremos el contenido de este, usando **read()**; almacenaremos los resultados en el extremos de escritura (**pipe_v[1]**), usando **write()**; y por último cerraremos aquellos archivos que hayamos abierto, usando **close()**. Observar *Imagen 3.2.*

```c
// El hijo escribe
case 0:
    close(pipe_v[1]);  // Cerramos el extremo de escritura de la tubería

    // argv[2] > parámetro archivo destino
    int archivoDest = open(argv[2], O_WRONLY | O_TRUNC, 0644);
    if (archivoDest == -1){
        printf("No se ha podido abrir el archivo destino\n");
        printf("Creando nuevo archivo destino: %s\n", argv[2]);
        archivoDest = creat(argv[2], 0644);
        if (archivoDest < 0) {
            perror("Error al crear el archivo destino");
            exit(1);
        }
    }


    // El buffer tiene que ser mayor o igual al tamaño leído
    while ((bytesLeidos = read(pipe_v[0], buffer, BUFFER_SIZE)) > 0)
        write(archivoDest, buffer, bytesLeidos);

    close(pipe_v[0]);
    close(archivoDest);
    exit(1);
```

Imagen 3.2

Para la parte del hijo utilizaremos una estructura similar, sólo que en este caso al intentar abrir el archivo Destino, en caso de que no exista, se creará un nuevo archivo utilizando **creat()**. Nuestro hijo leerá el contenido de la tubería y lo escribirá en el archivo Destino utilizando **read()** y **write()** respectivamente. Observar *Imagen 3.3*.

```
// El padre lee
default:
    close(pipe_v[0]);  // Cerramos el extremo de lectura de la tubería

    int archivoOrigen = open(argv[1], O_RDONLY);
    if (archivoOrigen == -1) {
        perror("No se ha podido abrir el archivo origen\n");
    }

    while ((bytesLeidos = read(archivoOrigen, buffer, BUFFER_SIZE)) > 0)
        write(pipe_v[1], buffer, bytesLeidos);


    close(pipe_v[1]);
    close(archivoOrigen);

    // Esperar a que el proceso hijo termine
    wait(NULL);
```

Imagen 3.3

Comparación de resultados

Resultado Esperado

```
$ ls
origen.txt
$ copiar origen.txt destino.txt
$ ls
origen.txt destino.txt
$ diff origen.txt destino.txt
$
```

Resultado Obtenido

```
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas_Operativos/
P1/e3$ ./copiar Origen.txt Destino.txt
No se ha podido abrir el archivo destino
Creando nuevo archivo destino: Destino.txt
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas_Operativos/
P1/e3$ ls
copiar  copiar.c  Destino.txt  Origen.txt
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas_Operativsss
sant_vz6@santvz6-IdeaPad-Gaming-3-15ACH6:~/Escritorio/UA/Sistemas_Operativos/
P1/e3$
```

Ejercicio 4. COMUNICACIÓN ENTRE PROCESOS: MEMORIA COMPARTIDA

Enunciado
Los estudiantes deben crear un programa en C que simule un sistema de procesamiento de datos en paralelo. El programa debe:
· Crear dos procesos: un proceso padre y un proceso hijo utilizando la llamada al sistema fork().
· Utilizar memoria compartida para permitir que el proceso hijo escriba datos y que el proceso padre los lea.
· El proceso hijo generará una lista de diez números aleatorios y los almacenará en un segmento de memoria compartida.
· El proceso padre accederá a la memoria compartida para leer estos números, calcular su media y mostrarla en pantalla.
· Asegurarse de que la memoria compartida se libere adecuadamente después de su uso.

Código
Las librerías específicas que utilizaremos para ese ejercicio son las siguientes (observar *Imagen 4.1*).

```c
// Uso de Share Memory
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include <time.h> // usasdo para generar números aleatorios
```

Imagen 4.1

También declararemos las siguientes variables:

- el identificador de memoria (shmid = **shmget()**)
- el pid (pid)
- el tamaño de la memoria (SHM_TAM)
- el segmento de memoria donde guardaremos los números (*numeros).

En nuestra función **main**() nos encargaremos de crear un proceso hijo mediante la llamada a **fork()**, utilizaremos de nuevo la sintaxis del **switch-case**.

En la parte del hijo ligaremos el segmento de memoria compartida al proceso utilizando **shmat(shmid, direccion=NULL "donde haya hueco", identificadores de permisos = NULL)**. Una vez ligada la memoria al proceso podremos generar los números aleatorios utilizando el tiempo (*para esta parte me ayudé de Internet ya que no controlo mucho de C en cuánto a librerías*). A medida que se vayan generando los números vamos guardandolos en la variable que ligó el proceso con la memoria compartida (**numeros** en este caso**).** Finalmente desligamos nuestra memoria compartida usando **shmdt((char *) numeros)** y haremos un **exit(0)** para que el padre pueda finalizar el **wait(NULL)**. Observar *Imagen 4.2.*

```c
case 0:
    // ligamos el segmento de memoria compartida al proceso (usando shm
    // NULL → La dirección donde se desea que se incluya. NULL → lo cuá
    // se incluya en cualquier zona libre del espacio de direcciones de
    // NULL → identificadores de permisos (en la práctica también será
    numeros = (int *)shmat(shmid, NULL, NULL);

    // Generamos 10 números aleatorios y los almacenamos en la memoria
    // Como explicó el profesor de Métodos de Inferencia (los números a
    printf("Soy el hijo (%d): los números generados son: ",getpid());
    srand(time(NULL));
    for (int i = 0; i < 10; i++){
        numeros[i] = rand() % 100; // Números aleatorios entre 0 y 99
        printf("%d, ", numeros[i]);
    }
    printf("\n");

    // Desligamos el segmento de memoria compartida
    if (shmdt((char *)numeros)<0){
        sprintf(error,"Pid %d: Error al desligar la memoria compartida:
        perror(error);
        exit(1);
    }
    exit(0);
```

Imagen 4.2

Para la parte del padre seguiremos el mismo procedimiento que en la parte del hijo. La diferencia radica que en vez de generar números aleatorios, utilizaremos el bucle for para acceder a cada elemento guardado en la memoria compartida y los sumaremos a la variable **sum**. Una vez hayamos sumado todos los elementos podremos realizar la media y podremos imprimirla por la terminal. Finalmente desligaremos el proceso de la memoria de nuevo, pero en este caso utilizaremos **shmctl** para ejecutar una operación de control de tipo "destrucción del segmento de memoria compartida". La sintaxis será la siguiente **shmctl(shmid, tipo_IPC_control, 0)**. Observar *Imagen 4.3*

```
wait(NULL); // Una vez generados los números y guardados en la memoria compartida
// Importante desligar la memoria compartida del proceso hijo

// ligamos el segmento de memoria compartida al proceso padre
numeros = (int *)shmat(shmid, NULL, NULL);

// Leemos los números de la memoria compartida y calculamos la media
printf("Soy el hijo (%d): los números generados fueron: ",getpid());
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += numeros[i];
    printf("%d, ", numeros[i]);
}
printf("\n");
float media = sum / 10.0;
printf("La media es de: %f\n", media);

// Desadjuntmos el segmento de memoria compartida
if (shmdt((char *)numeros)<0){
    sprintf(error,"Pid %d: Error al desligar la memoria compartida: ",getpid());
    perror(error);
    exit(1);
}

// shmctl: para realizar un conjunto de operaciones de control sobre una zona de memoria compartida
// Liberamos el segmento de memoria compartida usando IPC_RMID
// IPC_RMID: elimina el identificador de memoria compartida especificado por shmid del sistema,
// destruyendo el segmento de memoria compartida y las estructuras de control asociadas

if (shmctl(shmid, IPC_RMID, 0)<0){
    sprintf(error,"Pid %d: Error al liberar la memoria compartida: ",getpid());
    perror(error);
    exit(1);
}
```

Imagen 4.3

Comparación de resultados

Resultado esperado

El proceso hijo debe mostrar el siguiente mensaje:

"Soy el hijo (pid) : los números generados son: x, x1,...x9"

El proceso padre debe mostrar:

"Soy el padre (pid). Los números generados fueron: x, x1,...x9. La media es de y"

Resultado Obtenido

```
oy el hijo (6069): los números generados son: 70, 24, 16, 65,
10, 31, 71, 26, 36, 50,
oy el hijo (6068): los números generados fueron: 70, 24, 16,
5, 10, 31, 71, 26, 36, 50,
a media es de: 39.900002
```