Facultad de Ingeniería

Programa Ingeniería de Sistemas

3004610-1

Práctica de Laboratorio
Sistemas Operativos

---

1. Modelo Fork-wait

El uso de la llamada al sistema `fork()` crea un nuevo proceso dentro el sistema. Éste suele denominarse proceso `hijo` y se ejecuta simultáneamente con el proceso que realiza la llamada `fork`, denominado proceso `padre`. Después de crear un nuevo proceso hijo, ambos procesos ejecutarán la siguiente instrucción después de la llamada al sistema fork(). El proceso creado copia, del padre, el mismo valor del registro *PC* del procesador (*Program counter*), los valores de los registros de estado de la CPU, los archivos abiertos, entre otros aspectos que se heredan.

El prototipo de la función Fork() es:

```
pid_t fork()
```

No recibe ningún argumento y devuelve un valor entero. Este valor indica la correcta ejecución o una condición de error, así:

1. Un valor positivo: indica la correcta ejecución devolviendo el pid (*process identifier*) del proceso hijo, este valor es devuelto al proceso padre.
2. Un valor de cero: es devuelto al proceso hijo
3. Un valor negativo: indicando una condición de error en la creación del proceso hijo. Este valor es devuelto al proceso que la realiza la llamada.

El valor de retorno corresponde a un entero con signo de 32-bits. Por razones de portabilidad el tipo de dato `pid_t` es una redefinición de tipo del tipo reservado `__pid_t` que al final es un entero signado de 32 bits, así:

```
//Tomado de types.h [£INSTALLDIR/include/sys/types.h]
typedef __pid_t pid_t;
#define __PID_T_TYPE __S32_TYPE
#define __S32_TYPE      int
```

La Figura 1 ilustra el proceso de creación del proceso padre. Los segmentos de datos y de pila `DS: Data segment, SS: Stack Segment` son copiados en nuevo espacio de memoria para el proceso hijo. El segmento de código `CD: Code segment` al tener propiedades de sólo lectura, generalmente, es compartido por los procesos padre e hijo.

por diferentes razones la ejecución de la función puede fallar, por lo que es necesario evaluar el valor de retorno y evitar comportamientos indeterminados en la ejecución de los programas.

El siguiente programa corresponde al funcionamiento básico para la creación de un proceso hijo:

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
```

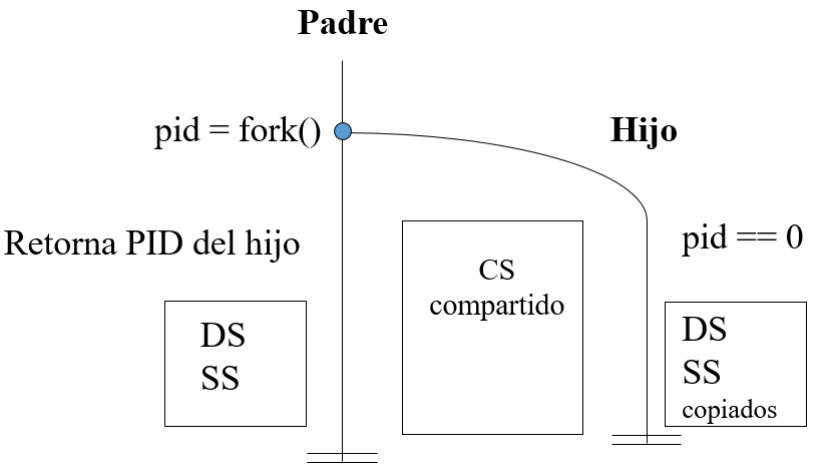

Figura 1: Esquema de ejecución de la llamada al sistema fork()

```
4  int main()
5  {
6      pid_t pidHijo;
7      pidHijo=fork();
8      if(pidHijo < 0){ perror("Error en la creación\n"); exit(-1); }
9      printf("Hola Mundo Fork() [%d] [%d]!\n", getpid(), getppid());
10     wait(NULL);
11     return 0;
12 }
```

La salida sería algo similar a:

```
Hola Mundo Fork() [520] [200]
Hola Mundo Fork() [521] [520]
```

Los enteros en las lineas de salida corresponden a los pid de cada proceso. El orden y el valor de esta salida puede variar en cada ejecución.

La ejecución de la instrucción de la linea 7 crea un nuevo proceso dentro del sistema. La ejecución del proceso padre y el proceso hijo es asíncrona y no se debe asumir ningún orden en particular.

El proceso padre y el proceso hijo completaran la instrucción de asignación como instrucción siguiente a la creación del proceso. En particular, la sentencia en lenguaje de alto nivel *pidHijo=fork()* una vez traducida a un representación en lenguaje intermedio implica la ejecución de más de una instrucción. Así, la siguiente instrucción a la ejecución fork() es

la asignación del valor a la variable pidHijo. El valor almacenado dependerá del proceso que la ejecute; para el proceso padre el valor será un entero positivo correspondiente al Pid de proceso hijo, en este caso 521. Para el proceso hijo la variable tomará un valor de 0.

La linea 8 ilustra la forma de validar el código de terminación de fork cuando su ejecución es fallida. Si el valor devuelto es un número menor que 0, la creación del nuevo proceso no finalizo correctamente por lo que se procede a informar del error y terminar con indicando una condición de error *exit(-1)*

La función getpid() de la linea 9 permite obtener el pid del proceso que la llama. su prototipo es *pid_t getpid()*. De igual forma la función *pid_t getppid()* es utilizada para obtener el identificador del proceso padre.

Finalmente, la llamada a la función *wait(NULL)* de la linea 10 evita la terminación anticipada del padre. Lo que podría causar que el proceso hijo al ejecutarse no reportara correctamente el valor del identificador del proceso padre. Debido a que la terminación del proceso padre antes de la terminación del proceso hijo convertiría este ultimo en un proceso Huérfano, que terminara siendo adoptado por el proceso *init* de pid igual a 1. El parámetro *NULL* indica que no se almacenara el valor de terminación de la ejecución del proceso hijo. La función *wait(int status)* es por naturaleza una función bloqueante. Si el proceso llamante tiene procesos hijos activos en el sistema, éste se bloquea hasta que alguno de ellos termine, permitiendo capturar el valor de terminación del primer proceso hijo que finalice. Si el proceso llamante no tiene hijo o al menos uno de ellos ha finalizado, la llamada no bloqueara al proceso y retornará su código de terminación.

2. Laboratorio

Realizar un programa que permita sumar un conjunto grande de enteros leídos desde un archivo.

Dado un archivo con un conjunto de enteros almacenados en una sola columna, la idea es estimar la suma total de los enteros dentro del archivo. Sin embargo, el proceso será realizado por dos procesos hijos. Éstos sumarán y escribirán en un archivo de texto el resultado parcial de cada parte del conjunto de enteros asignada. El primer hijo sumará la primera mitad del conjunto y el segundo proceso hijo sumará la mitad restante. Al final, el proceso padre leerá los resultados parciales y escribirá el total del proceso. La Figura 2 ilustra el esquema solicitado.

El proceso padre deberá:

1. Leer en un vector los datos del archivo de entrada *input.txt*
2. Estimar los índices (principio y fin) del vector sobre el cual cada proceso hijo trabajará

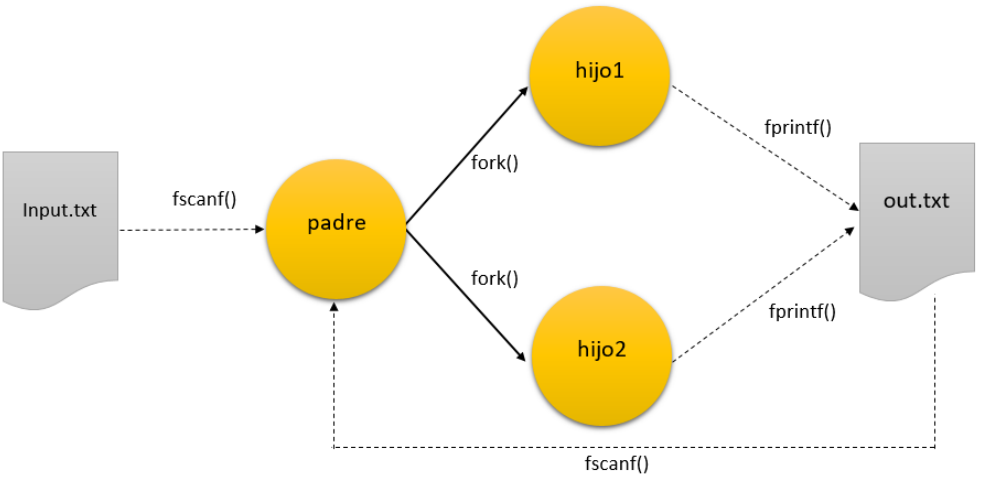

Figura 2: Esquema general del programa

3. Crear los procesos hijos
4. Esperar la terminación de los hijos
5. Imprimir el resultado final

Los procesos hijos deberán:

1. Recorrer el vector en las posiciones indicadas por el padre
2. Estimar la suma de las posiciones recorridas
3. Escribir el resultado en el archivo de salida *out.txt*

3. Funciones de apoyo

Lectura de datos: Esta función lee enteros desde un archivo de texto cuya estructura consiste en un numero inicial n indicando la cantidad de enteros en el archivo, seguido de n enteros, uno por cada fila. retorna el número de enteros leídos.

```
1  int leerNumeros(char *filename, int **vec){
2    int c, numero, totalNumeros;
3    FILE *infile;
4    infile = fopen(filename, "r");
5    if(! infile ){ error("Error fopen\n");}
6    fscanf(infile, "%d", &totalNumeros);
```

```
7     *vec = (int *)calloc(totalNumeros, sizeof(int));
8     if(!*vec){error("error calloc");}
9     for(c=0; c<totalNumeros; c++){
10        fscanf(infile, "%d", &numero);
11        (*vec)[c]=numero;
12        printf("%d\n", numero);
13    }
14    fclose(infile);
15    return c;
16  }
```

Recibe como argumento una arreglo de caracteres el nombre del archivo de entrada y un doble apuntador *int **vec* para que los datos leídos puedan ser accedidos por fuera de la función. Esta función debe llamarse así:

```
int *vector;
cantidadNumeros = leerNumeros("input.txt", &vector);
```

La función error es un utilitario definido así:

```
1  void error(char *msg){
2     perror(msg);
3     exit(-1);
4  }
```

Al finalizar los procesos hijos, el padre deberá leer el resultado del archivo *out.txt*, en el cual cada hijo escribió una linea con el resultado de su suma parcial.

```
1  int leerTotal(){
2    FILE *infile;
3    int sumap1=0,sumap2=0,total=0;
4    infile = fopen("out.txt","r");
5    if(!infile) error("Error padre archivo resultados");
6    fscanf(infile,"%d", &sumap1);
7    fscanf(infile,"%d", &sumap2);
8    total = sumap1 + sumap2;
9    return total;
10 }
```

Debido a que el esquema de comunicación entre los procesos está basado en archivos, es necesario que antes de cada ejecución se elimine el archivo *out.txt*, para evitar leer valores de ejecuciones anteriores. Para ésto es útil la llamada al sistema *remove*, así:

```
remove("out.txt");
```

Finalmente, una forma para estimar los índices de inicio y fin sobre los cuales operará cada proceso hijo puede ser:

```
delta = cantidadNumeros/2;
limites[0][0] = 0;              limites[0][1] = delta;
limites[1][0] = limites[0][1]; limites[1][1] = cantidadNumeros;
```