



**UNIVERSIDAD NACIONAL  
SAN AGUSTIN DE AREQUIPA**



**Facultad de Ingeniería, Producción y Servicios**

**Escuela Profesional de Ciencia de la Computación**

---

Laboratorio 3

---

**Presentado por:**

Parizaca Mozo, Paul Antony

**CUI:**

20210686

**Curso:**

Sistemas Operativos - Grupo A

**Github:**

<https://github.com/PaulParizacaMozo/SistemasOperativos>

**Docente:**

Yessenia Yari Ramos

Arequipa, Perú

2023

Todos los códigos comentados y códigos que se usaron están en github:  
<https://github.com/PaulParizacaMozo/SistemasOperativos/tree/main/Laboratorio3%20Analisis%20Codigo>

## Ejercicio 1

### -Código comentado

```
21 //Ejercicio 1
22 #include <stdio.h>
23 #include <sys/types.h>
24 #include <unistd.h>
25
26 int main(void) {
27     pid_t pid; // Declaración de una variable para almacenar el ID del proceso
28     int var = 0; // Declaración e inicialización de una variable entera "var"
29
30     printf("PID antes de fork(): %d\n", (int) getpid()); // Imprimir el ID del proceso antes de crear un proceso hijo format:
31     if ((pid = fork()) > 0) { // Crear un proceso hijo con fork() y comprueba si se está ejecutando en el proceso padre
32         printf("PID del padre: %d\n", (int) getpid()); // Imprimir el ID del proceso padre format:
33         var++; // Incrementar la variable "var" en el proceso padre
34     } else {
35         if (pid == 0) // Comprueba si se está ejecutando en el proceso hijo
36             printf("PID del hijo: %d\n", (int) getpid()); // Imprimir el ID del proceso hijo además la variable var no es modificada format:
37         else
38             printf("Error al hacer fork()\n"); // Mensaje de error al crear el proceso hijo format:
39     }
40     printf("Proceso [%d] -> var = %d\n", (int) getpid(), var); // Imprime el ID del proceso actual y el valor de "var" format:
41     getchar();
42 }
```

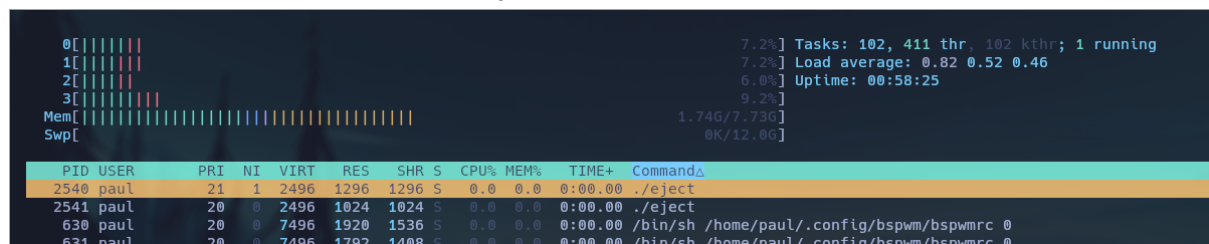
Lo que hace el programa es crear una variable `pid_t` para almacenar el ID de un proceso. Luego creamos un proceso hijo usando `fork()` y tenemos 3 casos si es mayor a 0 se está ejecutando el proceso padre, si es igual a 0 se está ejecutando el proceso hijo, caso contrario hay un error y no se pudo ejecutar de forma correcta el `fork()`. Y se irá imprimiendo cual caso sucede haciendo uso de la función `getpid()` que retorna el identificador(PID) del proceso actual.

### -Ejecución del código

```
> gcc -o eject main.c && ./eject
PID antes de fork(): 2540
PID del padre: 2540
Proceso [2540] -> var = 1
PID del hijo: 2541
Proceso [2541] -> var = 0
```

Solicitamos ingresar un enter para finalizar el programa y así poder observar los procesos creados.

Si vemos los procesos que están en ejecución en ese momento con `htop`



The screenshot shows the htop interface. At the top, system statistics are displayed: Tasks: 102, 411 thr, 102 kthr; 1 running; Load average: 0.82 0.52 0.46; Uptime: 00:58:25; Mem: 1.746/7.736; Swp: 0K/12.06. Below this, a table of running processes is shown.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
2540	paul	21	1	2496	1296	1296	S	0.0	0.0	0:00.00	./eject
2541	paul	20	0	2496	1024	1024	S	0.0	0.0	0:00.00	./eject
630	paul	20	0	7496	1920	1536	S	0.0	0.0	0:00.00	/bin/sh /home/paul/.config/bspwm/bspwmrc 0
631	paul	20	0	7496	1792	1408	S	0.0	0.0	0:00.00	/bin/sh /home/paul/.config/bspwm/bspwmrc 0

podemos observar que se encuentran los dos procesos creados por el `./eject` que es el ejecutable de nuestro programa.

## Ejercicio 2

### -Código Implementado

```
15 //Ejercicio2
14 #include <stdio.h>
13 #include <sys/types.h>
12 #include <unistd.h>
11
10 int main(void) {
9     pid_t pid; //Declaracion de una variable tipo pid_t que almacenara el Identificador del proceso
8     int i, n= 4; // Variable i sera el contador del bucle, y la variable n la cantidad maxima del bucle
7     for (i=0; i<n; i++){
6         if ( (pid = fork()) < 0 ){//Se llama al proceso fork() las n veces ejecutandose como proceso hijo o padre
5             break; } //Si el proceso es menor que 0 se detiene el bucle.
4     }
3     printf ("Proceso: %d / Padre: %d\n", (int) getpid(), (int) getppid()); // Imprime el Id del proceso actual y del padre format:
2     //El printf se ejecutara por cada proceso creado es decir 2 a la n veces.
1     getchar();
16 }
```

Se crea la variable de tipo pid\_t que almacena el identificador de los procesos.

Además de ejecutar un bucle 4 veces en el cual se llama a la función fork() para la creación de procesos hijo, el bucle se detendrá si fork() retorna -1.

Para finalizar se imprime el ID del proceso actual con getpid() y el ID del proceso padre getppid(). Esta línea se ejecuta tanto en el proceso padre original como en cada uno de los procesos hijos creados en el bucle.

### -Ejecución del código

```
> gcc -o eject main.c && ./eject
Proceso: 3138 / Padre: 1992
Proceso: 3140 / Padre: 3138
Proceso: 3139 / Padre: 3138
Proceso: 3147 / Padre: 3140
Proceso: 3143 / Padre: 3138
Proceso: 3141 / Padre: 3138
Proceso: 3145 / Padre: 3139
Proceso: 3148 / Padre: 3139
Proceso: 3144 / Padre: 3140
Proceso: 3149 / Padre: 3145
Proceso: 3146 / Padre: 3141
Proceso: 3150 / Padre: 3144
Proceso: 3142 / Padre: 3139
Proceso: 3152 / Padre: 3142
Proceso: 3151 / Padre: 3142
Proceso: 3153 / Padre: 3151
```

Solicitamos ingresar un enter para finalizar el programa y así poder observar los procesos creados.

Si vemos los procesos que están en ejecución en ese momento con htop

0[|||||]

1[|||||]

2[|||||]

3[|||||]

Mem[|||||]

Swp[|||||]

9.5%

9.2%

7.9%

7.8%

1.87G/7.73G

0K/12.0G

Tasks: 119, 439 thr, 101 kthr; 1 running

Load average: 0.09 0.35 0.47

Uptime: 01:12:02

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
3138	paul	20	0	2496	1228	1228	S	0.0	0.0	0:00.00	./eject
3139	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3140	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3141	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3142	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3143	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3144	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3145	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3146	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3147	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3148	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3149	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3150	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3151	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3152	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject
3153	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject

### Ejercicio 3

#### -Código Implementado

```
31 //Ejercicio 3
32 #include <stdio.h>
29 #include <sys/types.h>
28 #include <sys/wait.h>
27 #include <unistd.h>
26 #include <stdlib.h>
25
24 int main(void) {
23     //Declaracion de variables
22     pid_t pid_hijo; int estado, x; long i, j;
21
20     if ( (pid_hijo= fork()) == -1){ /* Código que ejecuta por el proceso PADRE: Error */
19         perror("Fallo al hacer fork()"); //Mensaje de error s:
18         exit(-1); // Salir del programa con un Código de error status:
17     } else if (pid_hijo == 0) { /* Código que ejecuta por el proceso HIJO */
16         fprintf(stdout, "PID hijo: %ld\n", (long) getpid()); // Imprime el id del proceso hijo stream: format:
15         fflush(stdout); // Limpia el buffer stream:
14         sleep(2); // Espera dos segundos seconds:
13     } else { /* Código que ejecuta el proceso PADRE */
12         //Espera a que termine el proceso hijo y almacena su estado. Si el valor que retorna wait() es distinto
11         //al del id del proceso hijo significa que el proceso padre fue interrumpido.
10         if ( (x=wait(&estado)) != pid_hijo) // stat_loc:
9             fprintf(stdout, "PADRE: interrumpido por señal\n"); // Imprime un mensaje a causa de una interrupcion stream: format:
8         else
7             // Imprime el ID del padre, el ID del hijo y el estado del hijo
6             fprintf(stdout, "PID padre: %ld / PID hijo: %ld / estado hijo: %d\n", (long) getpid(), (long) pid_hijo, estado); stream: format:
5             fflush(stdout); // Limpia el buffer stream:
4         }
3         //Sale del programa con codigo 0
2         getchar();
1         exit(0); /* Código PADRE e HIJO */ status:
32 }
```

Usando fork para crear un proceso hijo, si ocurre algún error este retornara -1 y el programa mostrará un mensaje de error y saldrá del programa.

Si se ejecuta el proceso hijo se imprime su identificador(PID) y con la función sleep(2) se espera dos segundos.

Si se ejecuta el proceso padre se espera a que el proceso hijo termine y se obtiene su estado. La función wait() retorna un número, si este es distinto al identificador del proceso hijo significa que el proceso padre fue interrumpido por una señal.

Caso contrario se imprime el identificador del proceso padre, el identificador del proceso hijo y el estado del hijo.

#### -Ejecución del Código

```
> gcc -o eject main.c && ./eject
PID hijo: 5660

PID padre: 5659 / PID hijo: 5660 / estado hijo: 0

A> ~/U/SistemasOperativos/Laboratorio3 AnalisisCodigo > on git main !2 ?3 > took 1m 9s
```

Agregando al código un getchar() para solicitar un enter para continuar el programa y así poder observar los procesos creados.

Si vemos los procesos que están en ejecución en ese momento con htop

5659	paul	20	0	2364	904	904	S	0.0	0.0	0:00.00	./eject
5660	paul	20	0	2496	896	896	S	0.0	0.0	0:00.00	./eject

## Ejercicio 4

### -Código Implementado

```
main.c
27 #include <sys/wait.h>
26 #include <unistd.h>
25 #include <stdlib.h>
24
23 int main(void) {
22     pid_t pid_hijo; int estado, x; long i, j;
21     /* Código PADRE: Error */
20     // Usando fork() se crea un nuevo proceso hijo y el valor lo almacenamos en la variable pid_hijo
19     if ( (pid_hijo= fork()) == -1){
18         perror("Fallo al hacer fork()"); //Si retorna -1 imprime un mensaje de error s:
17         exit(-1); // Sale del programa status:
16     }
15     /* Código HIJO */
14     else if (pid_hijo == 0) { //Si se cumple la condicion significa que se ejecuta el proceso hijo
13         if ( execl("/bin/ls", "ls", "-l", NULL) < 0) { // Intenta ejecutar el comando ls -l en el proceso hijo path: arg:
12             perror("Fallo al ejecutar: ls"); //Mensaje de error al ejecutar ls s:
11             exit(-1); //Sale del programa status:
10         }
9     }
8     /* Código PADRE */
7     //Caso contrario se ejecuta el proceso padre
6     else
5         if ( (x=wait(&estado)) != pid_hijo) { // Espera a que el proceso hijo termine stat_loc:
4             //Si el valor retornado es distinto al Id del proceso hijo significa que el padre fue interrumpido
3             fprintf(stdout, "PADRE: interrumpido por señal\n"); fflush(stdout); //Se imprime mensaje stream: format: stream:
2         }
1     exit(0); /* Código PADRE e HIJO, aunque el hijo nunca pasará por aquí */ status:
31 }
```

Usando fork para crear un proceso hijo, si ocurre algún error este retornara -1 y el programa mostrará un mensaje de error y saldrá del programa.

Si se ejecuta el proceso hijo haciendo uso de la función execl() se intenta ejecutar el comando "ls -l" en el proceso hijo. Si ocurre algún error se imprime un mensaje de error y se sale del programa.

Caso contrario se ejecuta el proceso padre el cual espera a que el proceso hijo termine y obtiene su estado. Luego compara el valor retornado por wait() con el identificador del proceso hijo y si son diferentes significa que el proceso padre fue interrumpido por una señal.

### -Ejecución del Código

```
> gcc -o eject main.c && ./eject
total 1084
-rwxr-xr-x 1 paul paul 15760 sep 29 09:48 eject
-rw-r--r-- 1 paul paul 1125 sep 29 08:00 ejercicio1.c
-rw-r--r-- 1 paul paul 722 sep 29 08:42 ejercicio2.c
-rw-r--r-- 1 paul paul 1460 sep 29 09:11 ejercicio3.c
-rw-r--r-- 1 paul paul 1327 sep 29 09:48 main.c
-rw-r--r-- 1 paul paul 1073193 sep 29 08:38 'SO -lab3.pdf'
```

```
^> ~/U/SistemasOperativos/Laboratorio3 AnalisisCodigo > on git P main !2 ?4 |
```

## Ejercicio 5

### -Código Implementado

#### ->defines.h

```
main.c defines.h
30 /* defines .h */
29 #ifndef DEFINES_H
28 #define DEFINES_H
27
26 #include <stdio.h>
25 #include <string.h>
24 #include <sys/types.h>
23 #include <fcntl.h>
22 #include <errno.h>
21 #include <signal.h>
20
19 #define max(a,b) (((a)>(b)) ? (a): (b)) // Define una funcion max(a,b) hace uso del operador ternario
18
17 /* Valores enteros asignados a verdadero, falso, ejecucion */
16 /* correcta y ejecucion con error. */
15
14 #define TRUE 1
13 #define FALSE 0
12 #define OK 1
11 #define ERROR 0
10
9 /* Longitud maxima de la linea de ordenes. */
8 #define MAXLINE 200
7 /* Numero maximo de argumentos para cada orden simple. */
6 #define MAXARG 20
5 /* LOGO que aparece al empezar e ejecutar el minishell */
4 #define LOGO "MINI SHELL (c)2004 Tu nombre\n"
3 /* Cadena que aparece como PROMPT en la linea de comandos */
2 #define PROMPT "msh_$ "
1
31 #endif
```

#### ->main.c

```
main.c defines.h
37 #include <stdio.h>
36 #include <stdlib.h>
35 #include <unistd.h>
34 #include <fcntl.h>
33 #include <sys/wait.h>
32 #include <stdlib.h>
31 #include "defines.h"
30
29
28 char *orden; // Puntero a cadena que sera el comando que se quiere ejecutar
27 char *argumentos[MAXARG]; // Array de punteros a cadenas que seran los argumentos del comando
26 int narg; // Contador para el numero de argumentos
25 int es_background;
24
23 void construye_orden(char *argv[]); // Se declara la funcion
22
21 int main(int argc, char *argv[])
20 {
19     int i;
18     int status;
17
16     if (argc != 2) { // Verifica que el numero de argumentos sea 2
15         fprintf(stderr, "Uso: %s fich_a_visualizar\n", argv[0]); // imprime mensaje stream: format:
14         exit(1); //sale del programa status:
13     }
12     // Si pasa el primer condicional significa que tiene los argumentos correctos llama a la funcion
11     construye_orden(argv);
10     /* CODIGO DEL HIJO */
9     if (fork() == 0) { // Crea un proceso hijo
8         execvp(orden, argumentos); // Ejecuta el comando con sus argumentos file: argv:
7         fprintf(stderr, "%s no encontrado o no ejecutable\n", orden); // Si hay algun error ,muestra un mensaje stream: format:
6         exit(1); // Sale del programa status:
5     }
4
3     /* CODIGO DEL PADRE*/
2     wait(&status); //Espera a que el proceso hijo termine stat_loc:
1     exit(1); // Sale del programa status:
38 }
```



```

40
1 void construye_orden(char * argv[]) // Se define lo que hara la funcion
2 {
3     int i,j;
4     //Inicializa las variables
5     narg=1;
6     es_background=FALSE;
7
8     //Inicializa el arreglo de argumentos
9     for(j=0; j<MAXARG; j++) argumentos[j]=NULL;
10
11     /* Atencion: La asignación de la cadena "cat" a 'orden' que está declarado como
12     un 'char *' se trata como estática, es decir, en tiempo de compilación, el compilador
13     reserva espacio de memoria suficiente para almacenar la cadena "cat" (4 bytes). */
14
15     orden="cat"; // El comando a ejecutar sera cat
16
17     /* Atención: Por contra, el array de cadenas 'argumentos' no se trata como una
18     variable estática, y por eso, es responsabilidad del programador reservar memoria
19     para las posiciones que se vayan a utilizar (en el ejemplo, 0 y 1) */
20
21     //Asigna memoria y copia la cadena para el primer argumento
22     argumentos[0]=(char *)malloc(strlen(orden)+1); size: s:
23     strcpy(argumentos[0],orden); dest: src:
24     //Asigna memoria y copia la cadena para el segundo argumento
25     argumentos[1]=(char *)malloc(strlen(argv[1])+1); size: s:
26     strcpy(argumentos[1],argv[1]); dest: src:
27 }

```

Primero se declaran variables las cuales almacenarán la cadena del comando a usar, las cadenas de los argumentos, un contador para el número de argumentos y un indicador para ver si el comando se ejecuta en segundo plano.

La función `construye_orden` es la encargada de construir el comando a ejecutar bajo el proceso hijo.

Separa memoria dinámica para los dos argumentos que requiere el comando usando la función `alloc()` y copia las cadenas en sus respectivas posiciones del arreglo de argumentos.

La función `main` verifica que el número de argumentos sea el correcto en este caso dos argumentos, si no cumple imprime un mensaje de error y termina el programa, si cumple la condición anterior llama a la función `construye_orden` y luego crea un proceso hijo y llama a la función `execvp()` si este se ejecuta de manera correcta ejecuta el comando, si da algún error imprime un mensaje y finaliza el programa.

El proceso padre espera a que el proceso hijo termine haciendo uso de la función `wait()` y sale del programa.

### -Ejecución del Código

```

> gcc -o eject main.c && ./eject "archivo.txt"
Lab 3
hola mundo
se ejecuto con exito
...

```

➤ ~ / U / SistemasOperativos / Laboratorio3 AnalisisCodigo > on git P main 12 ?7