

Práctica 04

DOCENTE	CARRERA	CURSO
MSc. Maribel Molina Barriga	Escuela Profesional de Ingeniería de Software	Sistemas Operativos

GRUPO	TEMA	DURACIÓN
6	Planificación de Procesos	5 horas

Integrantes

- José Carlos Machaca Vera
- Jhosep Alonso Mollapaza Morocco
- Patrick Andres Ramirez Santos

Índice

1. Actividades	2
1.1. Actividad 1	2
1.2. Actividad 2	2
2. Ejercicios	4
2.1. Ejercicio 1	4
2.2. Ejercicio 2	4
2.3. Ejercicio 3	6
2.4. Ejercicio 4	7
2.5. Ejercicio 5	8
3. Cuestionario	10
Indice Source Code	12
Indice de Capturas de Pantalla	12

1. Actividades

1.1. Actividad 1

Este código imprime los identificadores del proceso en ejecución (PID) y su proceso padre (PPID). Se refleja la gestión de procesos, donde cada nuevo proceso se crea a partir de otro existente (proceso padre) y recibe un PID único. Al observar el PID y PPID, se puede entender la estructura jerárquica del sistema operativo, además de que cada nuevo proceso se crea a partir de otro existente y recibe un PID único, donde cada proceso tiene un lugar específico en el árbol de procesos, iniciando desde el proceso INIT.

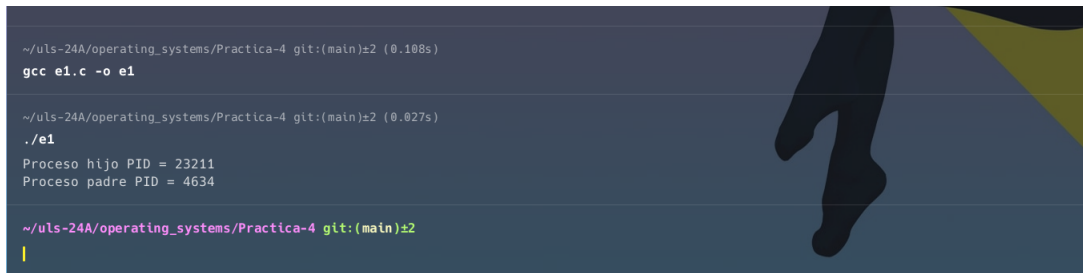
Source Code 1: Actividades/e1.c

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

int main(void) {
    printf("Proceso hijo PID = %d\n", getpid());
    printf("Proceso padre PID = %d\n", getppid());

    return EXIT_SUCCESS;
}
```

Figura 1: Compilación y ejecución de Actividad 1



```
~/uls-24A/operating_systems/Practica-4 glt:(main)±2 (0.108s)
gcc e1.c -o e1

~/uls-24A/operating_systems/Practica-4 glt:(main)±2 (0.027s)
./e1
Proceso hijo PID = 23211
Proceso padre PID = 4634

~/uls-24A/operating_systems/Practica-4 glt:(main)±2
```

1.2. Actividad 2

Analizamos el código y respondemos las siguientes preguntas ¿cuántos procesos se crean?, ¿qué realiza cada uno de estos procesos?, ¿existe algún tipo de relación entre estos procesos?.

- Creación de procesos: Se crean 2 procesos, el proceso principal (padre) y un proceso hijo. "pid_t pid_hijo;pid_hijo = fork();"
- Función de los procesos: El Proceso padre Invoca la función crear(), llama a fork() para crear un proceso hijo y luego espera 20 segundos antes de terminar. Y el proceso hijo ejecuta el comando ls -lh / para listar los archivos y directorios en el directorio raíz usando execv(). Si execv() falla, imprime un error y aborta.
- Relación entre los procesos: Sí, existe una relación padre-hijo entre los procesos. El proceso padre crea el proceso hijo mediante fork() y mantiene un breve control sobre él mediante la espera de 20 segundos antes de terminar su ejecución.

Source Code 2: Actividades/e2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int crear(char *programa, char **argumentos) {
    pid_t pid_hijo;
    pid_hijo = fork();

    if (pid_hijo != 0) {
        sleep(20);
        return pid_hijo;
    } else {
        execv(programa, argumentos);
        fprintf(stderr, "Se ha generado un error");
        abort();
    }
}

int main() {
    char *argumentos[] = {"ls", "-lh", "/", NULL};
    crear("/bin/ls", argumentos);

    return EXIT_SUCCESS;
}
```

Figura 2: Compilación y ejecución de Actividad 2



```
~/uls-24A/operating_systems/Practica-4 git:(main)±2 (0.112s)
gcc e2.c -o e2

~/uls-24A/operating_systems/Practica-4 git:(main)±2 (20.026s)
./e2
total 20K
dr-xr-xr-x.  1 root root    0 Jul 20  2023 afs
lrwxrwxrwx.  1 root root    7 Jul 20  2023 bin -> usr/bin
dr-xr-xr-x.  7 root root 4.0K Apr 23 09:58 boot
drwxr-xr-x. 20 root root 4.3K Apr 30 09:43 dev
drwxr-xr-x.  1 root root 5.2K Apr 25 15:59 etc
drwxr-xr-x.  1 root root    6 Dec 23 18:48 home
lrwxrwxrwx.  1 root root    7 Jul 20  2023 lib -> usr/lib
lrwxrwxrwx.  1 root root    9 Jul 20  2023 lib64 -> usr/lib64
drwx-----  1 root root    0 Apr 13  2023 lost+found
drwxr-xr-x.  1 root root    0 Jul 20  2023 media
drwxr-xr-x.  1 root root   24 Apr 11 22:46 mnt
drwxr-xr-x.  1 root root   50 Apr 21 14:45 opt
dr-xr-xr-x. 476 root root    0 Apr 30 04:43 proc
dr-xr-x----  1 root root 282 Mar 27 17:16 root
drwxr-xr-x. 56 root root 1.4K Apr 30 09:46 run
lrwxrwxrwx.  1 root root    8 Jul 20  2023/sbin -> usr/sbin
drwxr-xr-x.  1 root root    0 Jul 20  2023 srv
dr-xr-xr-x. 13 root root    0 Apr 30 09:43 sys
drwxrwxrwt. 25 root root 760 Apr 30 12:27 tmp
drwxr-xr-x.  1 root root 168 Nov 10 08:41 usr
drwxr-xr-x.  1 root root 206 Nov 10 08:45 var

~/uls-24A/operating_systems/Practica-4 git:(main)±2
```

2. Ejercicios

2.1. Ejercicio 1

El comportamiento del código es que el proceso hijo modifica una copia local de la variable value, y el proceso padre conserva el valor original de value porque fork() crea un espacio de memoria separado para el hijo. Por eso, el valor impreso por el padre sigue siendo 5.

Source Code 3: Ejercicios/e1.c

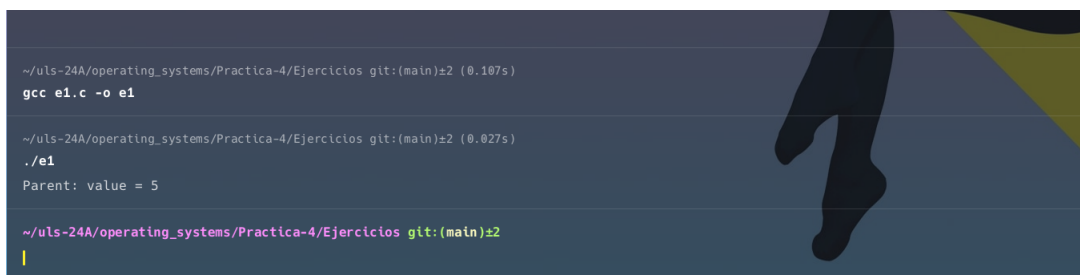
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int value = 5;

int main(void) {
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        value += 15;
        return 0;
    } else if (pid > 0) {
        wait(NULL);
        printf("Parent: value = %d", value);
        return 0;
    }
}
```

Figura 3: Compilación y ejecución de Ejercicio 1



```
~/uls-24A/operating_systems/Practica-4/Ejercicios git:(main)±2 (0.107s)
gcc e1.c -o e1

~/uls-24A/operating_systems/Practica-4/Ejercicios git:(main)±2 (0.027s)
./e1
Parent: value = 5

~/uls-24A/operating_systems/Practica-4/Ejercicios git:(main)±2
```

2.2. Ejercicio 2

Código ejecutado por el proceso padre: El proceso padre espera a que el proceso hijo termine su ejecución con wait(NULL). Una vez que el hijo ha completado su tarea y ha terminado, el padre procede a imprimir "Child completado". Esto indica que el padre ha detectado la finalización del hijo.

```
else {
    wait(NULL);
    printf("Child completado");
}
```

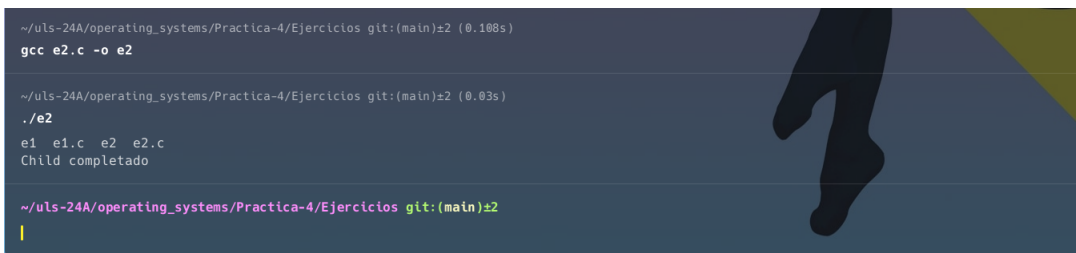
Código ejecutado por el proceso hijo: El proceso hijo ejecuta el comando `ls` para listar el contenido del directorio actual usando `execlp()`, una función que reemplaza la imagen del proceso actual con un nuevo programa. Aquí, el proceso hijo efectivamente se convierte en el proceso del comando `ls`.

```
    else if (pid == 0) {  
        execlp("/bin/ls", "ls", NULL);  
    }
```

Source Code 4: Ejercicios/e2.c

```
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <unistd.h>  
  
int main() {  
    pid_t pid;  
    pid = fork();  
  
    if (pid < 0) {  
        fprintf(stderr, "Fork Fallido");  
        return 1;  
    } else if (pid == 0) {  
        execlp("/bin/ls", "ls", NULL);  
    } else {  
        wait(NULL);  
        printf("Child completado\n");  
    }  
  
    return 0;  
}
```

Figura 4: Compilación y ejecución de Ejercicio 2



```
~/uls-24A/operating_systems/Practica-4/Ejercicios git:(main)±2 (0.108s)  
gcc e2.c -o e2  
  
~/uls-24A/operating_systems/Practica-4/Ejercicios git:(main)±2 (0.03s)  
./e2  
e1 e1.c e2 e2.c  
Child completado  
  
~/uls-24A/operating_systems/Practica-4/Ejercicios git:(main)±2
```

2.3. Ejercicio 3

Inicio del proceso padre

- Imprime que inicialmente hay un solo proceso.
- Realiza la primera llamada a `fork()` para crear el primer hijo.

Primer hijo (HIJO 1)

- Imprime un saludo y menciona que se detendrá por 20 segundos (aunque en el código dice 5s, en realidad son 20).
- Se suspende con `sleep(20)` y luego termina.

Proceso padre (continuación)

- Después de crear el primer hijo, imprime el PID del primer hijo.
- Realiza otra llamada a `fork()` para crear el segundo hijo.

Segundo hijo (HIJO 2)

- Imprime un saludo y anuncia que ejecutará el comando `ls`.
- Ejecuta `execlp("ls", "ls", NULL)` para listar los directorios. Si `execlp()` falla, imprime un mensaje de error.

Proceso padre (finalización)

- Después de crear ambos hijos, imprime el PID del segundo hijo.
- Anuncia que esperará a que ambos hijos terminen. Utiliza `wait(NULL)` dos veces para esperar que cada hijo termine, imprimiendo el PID de cada hijo que termina.

Source Code 5: Ejercicios/e3.c

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int pid;
    printf("Hasta aqui hay un solo proceso...\n");
    // Creamos un nuevo proceso
    pid = fork();
    if (pid == 0) {
        printf("HIJO 1: Holaaa soy el primer hijo...\n");
        printf("HIJO 1: Voy a pararme por 5s y luego termino\n");
        sleep(20);
    } else if (pid > 0) {
        printf("PADRE: Hola, soy el padre. El PID de mi hijo es: %d\n", pid);
        pid = fork();

        if (pid == 0) {
```

```
printf("HIJO 2: Holaaa, soy el segundo hijo...\n");
printf("HIJO 2: Y voy a ejecutar la orden 'ls'...\n");

execlp("ls", "ls", NULL);

printf("Si ve este mensaje el execlp no funciona :(\n");
} else if (pid > 0) {
    printf("PADRE: Hola otra vez. El PID de mi segundo hijo es: %d\n", pid);
    printf("PADRE: Voy a esperar a que terminen mis hijos...\n");
    printf("PADRE: Ha terminado mi hijo %d\n", wait(NULL));
    printf("PADRE: Ha terminado mi hijo %d\n", wait(NULL));
}
} else {
    printf("Hubo un error al llamar a fork()\n");
}
}
```

Figura 5: Compilación y ejecución de Ejercicio 3

```
~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±8 (0.108s)
gcc e3.c -o e3

~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±8 (0.029s)
./e3
Hasta aqui hay un solo proceso...
PADRE: Hola, soy el padre. El PID de mi hijo es: 14677
HIJO 1: Holaaa soy el primer hijo...
HIJO 1: Voy a pararme por 5s y luego termino
PADRE: Hola otra vez. El PID de mi segundo hijo es: 14678
PADRE: Voy a esperar a que terminen mis hijos...
HIJO 2: Holaaa, soy el segundo hijo...
HIJO 2: Y voy a ejecutar la orden 'ls'...
e1.c e2 e2.c e3 e3.c
PADRE: Ha terminado mi hijo 14678
PADRE: Ha terminado mi hijo 14677

~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±8
```

2.4. Ejercicio 4

Después de las tres llamadas a `fork()`, hay un total de 8 procesos generados. Este resultado es una consecuencia del crecimiento exponencial donde cada `fork()` duplica el número de procesos actuales.

Source Code 6: Ejercicios/e4.c

```
#include <unistd.h>

int main() {
    fork();
    fork();
    fork();

    return 0;
}
```

Figura 6: Compilación y ejecución de Ejercicio 4

```
nvim e4.c

~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±9 (0.1s)
gcc e4.c -o e4

~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±10 (0.026s)
./e4

~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±10
|
```

2.5. Ejercicio 5

Funciones que permiten compartir memoria

- `shm_open`: Esta función crea o abre un objeto de memoria compartida. En el código del productor, se usa con los flags `O_CREAT | O_RDWR` para crear el objeto si no existe y permitir lectura y escritura. En el código del consumidor, se abre con `O_RDONLY` para solo lectura.
- `ftruncate`: Usada por el productor para establecer el tamaño del objeto de memoria compartida. En este caso, se establece a `SIZE`, que es 4096 bytes.
- `mmap`: Mapea el objeto de memoria compartida en el espacio de direcciones del proceso. En el productor, se configura con `PROT_WRITE` para permitir escritura y en el consumidor con `PROT_READ` para solo lectura, ambos usando `MAP_SHARED` para permitir que los cambios sean visibles entre procesos.
- `shm_unlink`: Utilizada por el consumidor para desvincular el objeto de memoria compartida, lo que indica que el nombre puede ser reutilizado y el objeto será eliminado una vez que todos los procesos que lo tienen abierto lo cierren.

El código se compiló y ejecutó con los siguientes comandos:

```
$ gcc -lrt productor.c -o productor
$ gcc -lrt consumidor.c -o consumidor
$ ./productor && ./consumidor
```

Source Code 7: Ejercicios/producer.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/mman.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <uchar.h>
#include <unistd.h>

int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    const char *message_0 = "Hello";
```



```
const char *message_1 = "World!";

char *ptr;
int fd;

fd = shm_open(name, O_CREAT | O_RDWR, 0766);
ftruncate(fd, SIZE);
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);
sprintf(ptr, "%s", message_0);

ptr += strlen(message_0);
sprintf(ptr, "%s", message_1);
strlen(message_1);
return 0;
}
```

Source Code 8: Ejercicios/consumer.c

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#include <sys/mman.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    int fd;
    char *ptr;

    fd = shm_open(name, O_RDONLY, 0766);
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);

    printf("%s", (char *)ptr);
    shm_unlink(name);

    return 0;
}
```

Figura 7: Compilación y ejecución de Ejercicio 5

```
~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±13 (0.111s)
gcc -lrt consumer.c -o consumer

~/uls-24A/operating_systems/Practica-4/Actividades git:(main)±13 (0.028s)
./producer && ./consumer
HelloWorld!
```

3. Cuestionario

1. La principal característica de fork es que crea una copia exacta del proceso llamante (padre) en un nuevo proceso (hijo). El proceso hijo recibe una copia del espacio de memoria del padre pero con su propio espacio de direcciones. Esto significa que cualquier cambio realizado en el espacio de memoria del hijo no afecta al padre, y viceversa.
2. En sí, puedes crear tantos procesos hijos como lo permitan los recursos del sistema, pero en la práctica, hay límites establecidos por el sistema operativo. Estos límites pueden incluir el número máximo de procesos por usuario o límites globales de procesos, que se pueden consultar con comandos como 'ulimit -u' en sistemas tipo Unix/Linux.
3. 'execlp' es una función que ejecuta un programa, reemplazando la imagen del proceso actual con un nuevo programa. Utiliza el PATH del sistema para buscar el ejecutable especificado si no se da una ruta completa. Por ejemplo, execlp("ls", "ls", "l", NULL); ejecutaría el comando 'ls -l', listando archivos en el directorio actual con detalles
4. Este código crea procesos para simular el grafo de precedencia de la figura, asegurando que los mensajes se impriman en el orden correcto de acuerdo con las dependencias del grafo

Source Code 9: Implementación del grafo

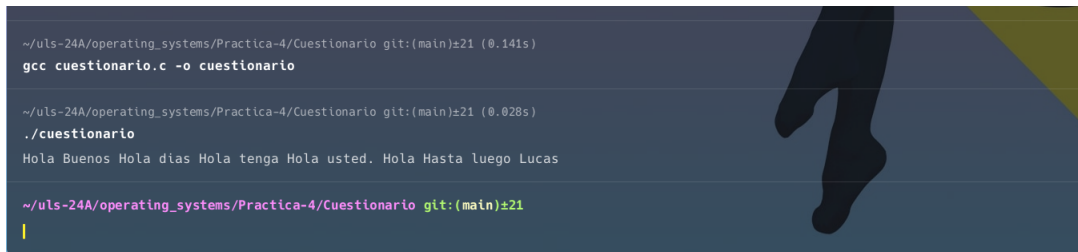
```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int pid;

    pid = fork();
    if (pid == 0) {
        printf("Hola ");
        pid = fork();
        if (pid == 0) {
            printf("Buenos ");
        } else {
            wait(NULL);
            pid = fork();
            if (pid == 0) {
                printf("dias ");
            } else {
                wait(NULL);
                pid = fork();
            }
        }
    }
}
```

```
    if (pid == 0) {  
        printf("tenga ");  
    } else {  
        wait(NULL);  
        pid = fork();  
        if (pid == 0) {  
            printf("usted. ");  
        } else {  
            wait(NULL);  
            printf("Hasta luego Lucas\n");  
        }  
    }  
}  
}  
} else {  
    wait(NULL);  
}  
return 0;  
}
```

Figura 8: Compilación y ejecución del grafo



```
~/uls-24A/operating_systems/Practica-4/Cuestionario git:(main)±21 (0.141s)  
gcc cuestionario.c -o cuestionario  
  
~/uls-24A/operating_systems/Practica-4/Cuestionario git:(main)±21 (0.028s)  
./cuestionario  
Hola Buenos Hola días Hola tenga Hola usted. Hola Hasta luego Lucas  
  
~/uls-24A/operating_systems/Practica-4/Cuestionario git:(main)±21
```

Indice Source Code

1.	Actividades/e1.c	2
2.	Actividades/e2.c	3
3.	Ejercicios/e1.c	4
4.	Ejercicios/e2.c	5
5.	Ejercicios/e3.c	6
6.	Ejercicios/e4.c	7
7.	Ejercicios/producer.c	8
8.	Ejercicios/consumer.c	9
9.	Implementación del grafo	10

Indice de Capturas de Pantalla

1.	Compilación y ejecución de Actividad 1	2
2.	Compilación y ejecución de Actividad 2	3
3.	Compilación y ejecución de Ejercicio 1	4
4.	Compilación y ejecución de Ejercicio 2	5
5.	Compilación y ejecución de Ejercicio 3	7
6.	Compilación y ejecución de Ejercicio 4	8
7.	Compilación y ejecución de Ejercicio 5	10
8.	Compilación y ejecución del grafo	11