



# Hoja de Ejercicios 3. Procesos


## Ejercicios Prácticos

Los ejercicios marcados con el icono  son prácticos y deben realizarse en el laboratorio. Los ejercicios prácticos de esta hoja requieren el entorno de usuario básico (shell) y de desarrollo (compilador, editores y depurador). Además algunos ejercicios necesitan acceso de superusuario que está disponible en las máquinas virtuales de la asignatura.

## Bloque de Control de Procesos

 **Ejercicio 1.** `ps(1)` permite ver los procesos del sistema y su estado. Estudiar la página de manual y determinar las opciones necesarias para:


- Mostrar todos los procesos del usuario actual en formato extendido. (Nota: usar la variable de entorno `USER`).
- Mostrar los procesos del sistema, incluyendo el identificador del proceso, el identificador del grupo de procesos, el identificador de sesión, el estado y el comando con todos sus argumentos.
- Observar el identificador de proceso, grupo de procesos y sesión de los procesos. ¿Qué identificadores comparten la *shell* y los programas que se ejecutan en ella (usar la opción `-H` para identificar fácilmente la relación entre procesos)?
- ¿Cuál es el identificador de grupo de procesos cuando se crea un nuevo proceso? Usar el comando `sleep` para poder verlo fácilmente en la salida del comando `ps`.

 **Ejercicio 2.** Escribir un programa que muestre los identificadores del proceso (PID, PPID, PGID y SID), el identificador del usuario y grupo, y su directorio de trabajo actual.

**Ejercicio 3.** Considera la siguiente orden de la línea de comandos:

```
echo "12345" > /proc/$$/fd/1
```


Discuta razonadamente cuál es el resultado esperado de la ejecución.

 **Ejercicio 4.** Consulta el tipo de fichero y contenido de los siguientes ficheros del proceso de la shell actual (`/proc/$$`) completa la siguiente tabla:

Fichero	Tipo enlace, dir,...	Descripción contenido/propósito
<code>cmdline</code>	Regular empty file	Archivo vacío
<code>cwd</code>	Enlace simbolico	Current working directo
<code>environ</code>	Regular empty file	Archivo vacío
<code>exe</code>	Enlace simbolico	Enlace al bash

fd	directorio	
limits		
maps		
root		

## Creación y Gestión de Procesos

 **Ejercicio 5.** Escribir un programa que cree un proceso hijo con las siguientes características:

- El programa recibirá dos argumentos en la forma:

```
./ejercicio5 <segundos_padre> <segundos_hijo>
```

- El hijo creará su propia sesión, imprimirá sus identificadores (como en el Ejercicio 2), esperará <segundos\_hijo> (segundo argumento) con la llamada `sleep(3)`; y terminará.
- El padre imprimirá sus identificadores (Ejercicio 2), esperará <segundos\_padre> (primer argumento) con la llamada `sleep`; y terminará.

Ejemplo de salida:

```
$ ./eje5 2 1
[Padre] PID=1616, PPID=1362, PGID=1616, SID=1362. Durmiendo 2s
[Hijo] PID=1617, PPID=1616, PGID=1617, SID=1617. Durmiendo 1s
[Hijo] Terminado.
[Padre] Terminado.
```


Considera las siguientes ejecuciones, observa los procesos con la orden `ps -lHu $USER` y completa la siguiente tabla para los procesos relacionados con el ejercicio:


Orden	Proceso PID	PPID, CMD ( padre)	Estado
./eje5 600 1&			
./eje5 1 600&			

**NOTA:** no todas las filas son necesarias


Explica razonadamente el estado de los procesos en el primer caso (el hijo termina antes) y el PPID de los procesos en el segundo caso (el padre termina antes), identifica el proceso padre con la ayuda del comando `ps(1)`.

Ejecuta el programa con un tiempo de espera largo (ej. `./eje5 60 60`), antes de que terminen las llamadas a `sleep(3)` presiona Ctrl-C en el terminal. ¿Qué ocurre? ¿Mueren ambos procesos? ¿Por qué?

 **Ejercicio 6.** `kill(1)` permite enviar señales a un proceso o grupo de procesos por su identificador (`pkill(1)` permite hacerlo por nombre de proceso). Estudiar la página de manual y las señales que se pueden enviar a un proceso.

 **Ejercicio 7.** En un terminal, arrancar un proceso de larga duración (ej. `sleep 600`). En otra terminal, enviar diferentes señales al proceso (terminar, interrumpir, parar, continuar) y comprobar el comportamiento.

**Ejercicio 8.** Determina las opciones adecuadas para el comando `kill(1)` para terminar (`SIGINT`) los dos procesos que se crean en la ejecución del programa del ejercicio 5.

 **Ejercicio 9** Escribir un programa que ejecute otro programa (ejecutable y argumentos) que se pasará como argumento. El programa creará un proceso hijo que ejecutará el programa dado en el argumento con la función `execvp(3)`. El proceso padre esperará que termine el hijo e imprimirá su código de salida. Ejemplos de ejecución:

```
$ ./eje9 cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
...
El proceso hijo terminó con código de salida 0

$ ./eje9 cat /etc/shadow
cat: /etc/shadow: Permission denied
El proceso hijo terminó con código de salida 1


$ (./eje9 sleep 3600&) ; sleep 1 ; pkill -SIGKILL sleep
El proceso hijo terminó por señal 9
```

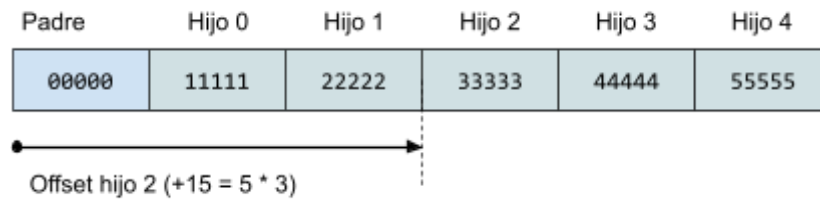
**Nota:** Considerar cómo deben pasarse los argumentos en cada caso para que sea sencilla la implementación. Por ejemplo: ¿qué diferencia hay entre: `./eje9 ps -el` y `./eje9 "ps -el"`?

**Ejercicio 10** Dibuja el esquema jerárquico de los procesos creados en la ejecución del siguiente programa con `argc=3`:

```
void main(int argc, char *argv[])
{
    int i;

    for(i=1; i<=argc; i++)
    {
        pid_t pid = fork();
        ...
    }
    return 0;
}
```

 **Ejercicio 11** Escribir un programa que genere un fichero con 5 procesos concurrentes con la siguiente estructura:



El proceso padre creará el fichero de salida `output.txt` y escribirá el primer segmento de 5 ceros (00000). A continuación creará 5 hijos y esperará a que termine cada uno mostrando el PID y número de hijo. Cada hijo realiza las siguientes acciones:

- Abrirá el archivo y desplazará el puntero de escritura el offset correspondiente.
- Escribirá la secuencia correspondiente ("11111", "22222"...). Esta parte puede implementarse de dos formas:
  - Creando una cadena con la secuencia correspondiente, con `sprintf(3)`; y escribiendo la cadena en el archivo.
  - Definiendo un array con cada posición inicializada al carácter correspondiente al hijo (`char c = '0' + index`); y escribiendo el array en el archivo

Ejemplo de ejecución:

```
$ ./eje10
El proceso hijo 5 con PID 1890 terminó correctamente
El proceso hijo 4 con PID 1889 terminó correctamente
El proceso hijo 3 con PID 1888 terminó correctamente
El proceso hijo 2 con PID 1887 terminó correctamente
El proceso hijo 1 con PID 1886 terminó correctamente
Escritura completada en output.txt

$ cat output.txt
000001111122222333334444455555
```

**Pista:** Usar el código de retorno en cada hijo igual al número de tarea (`exit(2)`) de forma que con la llamada `wait(2)` se puedan obtener ambos valores: PID e ID del hijo.

**Ejercicio 12** Considera el programa descrito en el Ejercicio 11. Dado que la tabla de descriptores se hereda ¿Es posible usar el descriptor de fichero abierto por el padre en los hijos?

**Ejercicio 13** Considere el código que se muestra a continuación:

```
int global;

void main() {
    int local=3;
    pid_t pid;

    global=10;

    pid = fork();

    if (pid == 0 ) {
        global = global + 5;
        local = local + 5;
```

```

    }
    else {
        wait(NULL);

        global += 10;
        local  += 10;
    }

    printf("global:%d local:%d\n", global, local);
}

```

Determine qué mensajes se imprimirán en la terminal. ¿Es posible que el resultado de la ejecución del programa cambie según el orden en que se ejecuten los procesos padre e hijo?. **Nota:** suponer que todas las llamadas al sistema se ejecutan exitosamente.

**Ejercicio 14** Considere el código que se muestra a continuación:

```

int a = 3;
void main() {
    int b=2;

    for (i=0;i<4;i++) {
        pid_t p=fork();


        if (p==0) {
            b++;
            execlp("/usr/bin/sleep", "/usr/bin/sleep", "2", NULL);
            a++;
        }
        else {
            wait(NULL);
            a++;
            b--;
        }
    }

    printf("variables - a:%d b:%d\n", a, b);
}

```

Determine qué mensajes se imprimirán en la terminal. ¿Cuántos procesos se crean en total? ¿Cuántos coexisten en el sistema como máximo?. **Nota:** suponer que todas las llamadas al sistema se ejecutan exitosamente.

## Planificación Procesos

 **Ejercicio 15.** La política de planificación y la prioridad de un proceso puede consultarse y modificarse con `chrt(1)`. Adicionalmente, `nice(1)` y `renice(1)` permiten ajustar el valor de *nice* de un proceso. Consultar la página de manual de ambos comandos:

- Ejecutar una shell (`/usr/bin/bash`) con `nice +10`.
- Ejecutar una shell (`/usr/bin/bash`) con `nice -10`.
- Ejecutar una shell (`/usr/bin/bash`) con política de planificación a `SCHED_FIFO` y prioridad 12.

En todos los casos comprobar la salida del comando `chrt -p <PID de la nueva shell>` y `ps -al`, y completar la siguiente tabla.

Proceso	Pólítica de planificación	Prioridad	nice	Require privilegios de root
Shell con nice +10				
Shell con nice -10				
Shell fifo y prio 12				

- Considerando que los tres procesos están listos para ejecutarse determina el orden en el que se ejecutarán.
- ¿Cuál es la prioridad y nice de los procesos que se crean en la nueva shell? (p.ej. ejecuta el comando `sleep 600` y determina sus prioridad, política de planificación y nice)

**Ejercicio 16.** En un sistema monoprocesador se ejecutan los procesos mostrados a continuación (todos los tiempos son en segundos):

Proceso	Llegada	CPU	E/S	CPU	E/S
P1	0	1	5	1	-
P2	1	3	1	1	1
P3	0	5	4	1	-
P4	3	3	2	1	1

Determina los tiempos de retorno (*turnaround*) y de espera para cada proceso y la productividad (*throughput*) del sistema para las siguientes políticas de planificación:

- *First Come First Serve* (FCFS)
- *Shortest Job First* (SJF)
- *Round Robin* (RR) con cuanto de 3s
- *Round Robin* (RR) con cuanto de 1s

**Ejercicio 17.** Considere un sistema monoprocesador con una política de planificación de procesos de 3 niveles con realimentación. Los procesos que agotan el cuanto bajan de nivel y los que ceden la CPU antes de agotarlo son promocionados. Además cuando un proceso acaba una operación de E/S se añade a la cola de espera de mayor prioridad. Cada nivel usa una política round robin cuyos cuantos de tiempo son 2, 4 y 8, respectivamente para cada nivel.

Al principio hay 3 procesos en la cola del nivel 1 (máxima prioridad) y el resto de colas están vacías. Los procesos tienen el siguiente patrón de ejecución que se repite indefinidamente:

Proceso	CPU	E/S
P1	3	5

P2	8	5
P3	5	5

---

Dibuja un diagrama de gantt para la ejecución para los primeros 30s que muestre además el estado de las colas de espera de cada nivel. Calcula los tiempos de espera de cada proceso.

**Ejercicio 18.** Repetir el problema anterior con las siguientes configuraciones:

- MLF de 10 niveles con el cuanto de cada nivel ( $Q_i$ )  $Q_i = 2 \cdot i$
- MLF de 10 niveles con el cuanto de cada nivel ( $Q_i$ )  $Q_i = 2$  (constante en todos los niveles).