

## Práctica 2.3. Procesos

### Objetivos

En esta práctica se revisan las funciones del sistema básicas para la gestión de procesos: políticas de planificación, creación de procesos, grupos de procesos, sesiones, recursos de un proceso y gestión de señales.

### Contenidos

- Preparación del entorno para la práctica
- Políticas de planificación
- Grupos de procesos y sesiones
- Ejecución de programas
- Señales

### Preparación del entorno para la práctica

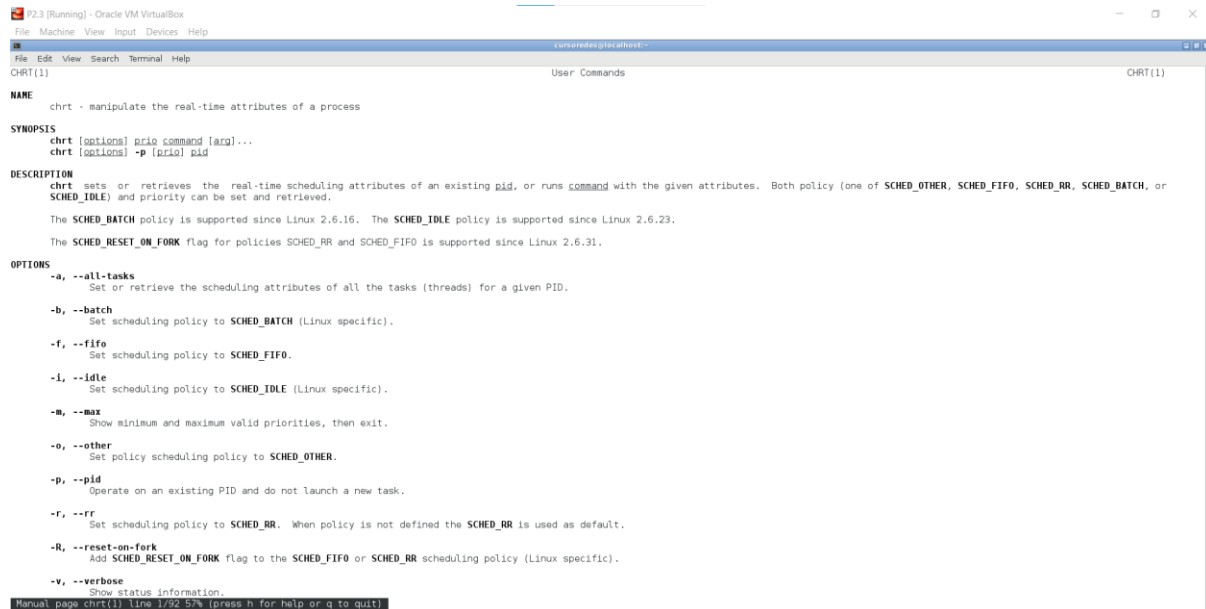
Algunos de los ejercicios de esta práctica requieren permisos de superusuario para poder fijar algunos atributos de un proceso, ej. políticas de tiempo real. Por este motivo, es recomendable realizarla en una **máquina virtual** en lugar de las máquinas físicas del laboratorio.

### Políticas de planificación

En esta sección estudiaremos los parámetros del planificador de Linux que permiten variar y consultar la prioridad de un proceso. Veremos tanto la interfaz del sistema como algunos comandos importantes.

**Ejercicio 1.** La **política de planificación y la prioridad** de un proceso puede consultarse y modificarse con el comando **chrt**. Adicionalmente, los comandos **nice** y **renice** permiten ajustar el valor de **nice** de un proceso. Consultar la página de manual de ambos comandos y comprobar su funcionamiento cambiando el valor de **nice** de la **shell** a -10 y después cambiando su política de planificación a **SCHED\_FIFO** con prioridad 12.

## man chrt



```
man chrt
NAME
  chrt - manipulate the real-time attributes of a process

SYNOPSIS
  chrt [options] prio command [arg]...
  chrt [options] -p (prio) pid

DESCRIPTION
  chrt sets or retrieves the real-time scheduling attributes of an existing pid, or runs command with the given attributes. Both policy (one of SCHED_OTHER, SCHED_FIFO, SCHED_RR, SCHED_BATCH, or SCHED_IDLE) and priority can be set and retrieved.

  The SCHED_BATCH policy is supported since Linux 2.6.16. The SCHED_IDLE policy is supported since Linux 2.6.23.

  The SCHED_RESET_ON_FORK flag for policies SCHED_RR and SCHED_FIFO is supported since Linux 2.6.31.

OPTIONS
  -a, --all-tasks
    Set or retrieve the scheduling attributes of all the tasks (threads) for a given PID.

  -b, --batch
    Set scheduling policy to SCHED_BATCH (Linux specific).

  -f, --fifo
    Set scheduling policy to SCHED_FIFO.

  -i, --idle
    Set scheduling policy to SCHED_IDLE (Linux specific).

  -m, --max
    Show minimum and maximum valid priorities, then exit.

  -o, --other
    Set policy scheduling policy to SCHED_OTHER.

  -p, --pid
    Operate on an existing PID and do not launch a new task.

  -r, --rr
    Set scheduling policy to SCHED_RR. When policy is not defined the SCHED_RR is used as default.

  -R, --reset-on-fork
    Add SCHED_RESET_ON_FORK flag to the SCHED_FIFO or SCHED_RR scheduling policy (Linux specific).

  -v, --verbose
    Show status information.

Manual page chrt(1) file /usr/share/man/man1/chrt.1.gz (press h for help or q to quit)
```

Podemos consultar el estado del proceso: **chrt -v -p**

Y cambiar la planificación del proceso: **chrt -o** (cambia la política de planificación a **SCHED\_OTHER**);  
**chrt -f** (a **fifo**); **chrt -r** (a **RR**)

El comando **nice** permite lanzar un programa con su política de planificación modificada (ajustamos el valor de **nice** del proceso).

El comando **renice** alterna la prioridad de los procesos que están ejecutándose

Cambiamos el valor de **nice** de la **shell** a -10:

```
[root@localhost ~]# nice -n-10 /bin/sh
sh-4.2#
```

Cambiamos su política de planificación a **fifo** con prioridad 12

```
sh-4.2# echo $$
2313
sh-4.2# chrt -f -p 12 2313
sh-4.2#
```

**Ejercicio 2.** Escribir un programa que muestre la política de planificación (como cadena) y la prioridad del proceso actual, además de mostrar los valores máximo y mínimo de la prioridad para la política de planificación.

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

int main(int argc, char **argv) {
    // consultamos la PID y la política de planificación del proceso actual
    int pid = atoi(argv[0]); // convertimos a int
    int politica = sched_getscheduler(pid); // consultamos la política de planificación (scheduler) del proceso actual (si pid=0, recupera la política del proceso de llamada)

    // mostramos como cadena la política de planificación consultada
    switch (politica) {
        case SCHED_OTHER:
            printf("SCHEDULER: OTHER\n");
            break;
        case SCHED_FIFO:
            printf("SCHEDULER: FIFO\n");
            break;
        case SCHED_RR:
            printf("SCHEDULER: RR\n");
            break;
        default:
            printf("ERROR\n");
            break;
    }

    // consultamos la prioridad del proceso
    struct sched_param parametros;
    sched_getparam(pid, &parametros);

    // mostramos la prioridad asignada al proceso
    printf("PRIORIDAD: %i\n", parametros.sched_priority);

    // tambien podriamos mostrar la prioridad actual del proceso:
    // printf("PRIORIDAD: %i\n", parametros.sched_curpriority);

    // consultamos los valores max y min de la prioridad para la política de planificación
    int max = sched_get_priority_max(politica);
    int min = sched_get_priority_min(politica);

    // mostramos los valores max y min de la prioridad para la política de planificación
    printf("MAX: %i - MIN: %i\n", max, min);
}
```

**Ejercicio 3.** Ejecutar el programa anterior en una *shell* con prioridad 12 y política de planificación SCHED\_FIFO como la del ejercicio 1. ¿Cuál es la prioridad en este caso del programa? **¿Se heredan los atributos de planificación?**

```
[cursoredes@localhost ~]$ sudo nice -n-10 /bin/sh
sh-4.2# echo $$
3396
sh-4.2# chrt -f -p 12 3396
sh-4.2# gcc -o outputej2 /home/cursoredes/p2.3/ej2.c
sh-4.2# ./outputej2
SCHEDULER: FIFO
PRIORIDAD: 12
MAX: 99 - MIN: 1
sh-4.2#
```

## Grupos de procesos y sesiones

Los grupos de procesos y las sesiones simplifican la gestión que realiza la *shell*, ya que permite enviar de forma efectiva señales a un grupo de procesos (suspender, reanudar, terminar...). En esta sección veremos esta relación y estudiaremos el interfaz del sistema para controlarla.

**Ejercicio 4.** El comando `ps` es de especial importancia para ver los procesos del sistema y su estado. Estudiar la página de manual y:

- Mostrar todos los procesos del usuario actual en formato extendido.
- 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? ¿Cuál es el identificador de grupo de procesos cuando se crea un nuevo proceso? [El PID y el SID de la shell son el SID del nuevo proceso. Comparten el GID, que es 1000](#)

**man ps**

**ps -u \$USER -f** muestra todos los procesos del usuario actual en formato extendido.

```
[cursoredes@localhost ~]$ ps -u $USER -f
UID          PID    PPID  C  TIME CMD
cursoredes+ 1328  1323  0  20:37 ?    00:00:00 /usr/bin/openbox --startup /usr/libexec/openbox-autostart OPENBOX
cursoredes+ 1337    1  0  20:37 ?    00:00:00 /bus-launch --sh-syntax --exit-with-session
cursoredes+ 1338    1  0  20:37 ?    00:00:00 /usr/bin/dbus-daemon --fork --print-pid 5 --print-address 7 --session
cursoredes+ 1406    1  0  20:37 ?    00:00:00 /usr/libexec/lssettings-daemon
cursoredes+ 1410    1  0  20:37 ?    00:00:00 /usr/libexec/gvfsd
cursoredes+ 1415    1  0  20:37 ?    00:00:00 /usr/libexec/gvfsd-fuse /run/user/1000/gvfs -f -o big_writes
cursoredes+ 1506    1  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --clipboard
cursoredes+ 1508  1506  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --clipboard
cursoredes+ 1517    1  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --display
cursoredes+ 1518  1517  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --display
cursoredes+ 1522    1  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --seamless
cursoredes+ 1524  1522  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --seamless
cursoredes+ 1527    1  0  20:37 ?    00:00:00 /usr/bin/VBoxClient --draganddrop
cursoredes+ 1530  1527  0  20:37 ?    00:00:13 /usr/bin/VBoxClient --draganddrop
cursoredes+ 1540  1328  0  20:37 ?    00:00:00 /usr/bin/ssh-agent /bin/sh -c "/usr/bin/openbox-session"
cursoredes+ 1563    1  0  20:37 ?    00:00:00 tint2
cursoredes+ 1565    1  0  20:37 ?    00:00:00 /usr/bin/python /usr/share/system-config-printer/applet.py
cursoredes+ 1566    1  0  20:37 ?    00:00:00 na-applet
cursoredes+ 1568    1  0  20:37 ?    00:00:00 abrt-applet
cursoredes+ 1614    1  0  20:37 ?    00:00:00 /usr/libexec/at-spi-bus-launcher
cursoredes+ 1631  1614  0  20:37 ?    00:00:00 /bin/dbus-daemon --config-file=/usr/share/defaults/at-spi2/accessibility.conf --nofork --print-address 3
cursoredes+ 1642    1  0  20:37 ?    00:00:00 /usr/libexec/at-spi2-registrd --use-gnome-session
cursoredes+ 1661    1  0  20:37 ?    00:00:00 /usr/bin/pulseaudio --start --log-target=syslog
cursoredes+ 1678    1  0  20:37 ?    00:00:00 /usr/libexec/dconf-service
cursoredes+ 1853    1  0  20:38 ?    00:00:00 /usr/libexec/xdg-desktop-portal
cursoredes+ 1858    1  0  20:38 ?    00:00:00 /usr/libexec/xdg-document-portal
cursoredes+ 1862    1  0  20:38 ?    00:00:00 /usr/libexec/xdg-permission-store
cursoredes+ 1874    1  0  20:38 ?    00:00:00 /usr/libexec/xdg-desktop-portal-gtk
cursoredes+ 2341  1563  1  20:57 ?    00:00:31 /usr/share/code/code --unity-launch
cursoredes+ 2343  2341  0  20:57 ?    00:00:00 /usr/share/code/code --type=zygote --no-sandbox
cursoredes+ 2451  2343  0  20:57 ?    00:00:00 /usr/share/code/code --type=renderer --js-flags=--no-lazy --disable-mojo-local-storage --no-sandbox --disable-features=ColorCorrectRendering --service-pipe-token=0B
cursoredes+ 2470    1  0  20:57 ?    00:00:00 /usr/libexec/gvfs-udisks2-volume-monitor
cursoredes+ 2475    1  0  20:57 ?    00:00:00 /usr/libexec/gvfs-afc-volume-monitor
cursoredes+ 2481    1  0  20:57 ?    00:00:00 /usr/libexec/gvfs-gphoto2-volume-monitor
cursoredes+ 2486    1  0  20:57 ?    00:00:00 /usr/libexec/gvfs-mtp-volume-monitor
cursoredes+ 2491    1  0  20:57 ?    00:00:00 /usr/libexec/gvfs-goa-volume-monitor
cursoredes+ 2495    1  0  20:57 ?    00:00:00 /usr/libexec/goa-daemon
cursoredes+ 2503    1  0  20:57 ?    00:00:00 /usr/libexec/goa-identity-service
cursoredes+ 2511    1  0  20:57 ?    00:00:00 /usr/libexec/mission-control-5
cursoredes+ 2513    1  0  20:57 ?    00:00:00 /usr/libexec/gvfsd-trash --spawner :1.3 /org/gtk/gvfs/exec_spaw/0
cursoredes+ 2532    1  0  20:57 ?    00:00:00 /usr/libexec/gvfsd-network --spawner :1.3 /org/gtk/gvfs/exec_spaw/1
cursoredes+ 2546    1  0  20:57 ?    00:00:00 /usr/libexec/gvfsd-dnssd --spawner :1.3 /org/gtk/gvfs/exec_spaw/3
cursoredes+ 2593  2343  2  20:58 ?    00:00:48 /usr/share/code/code --type=renderer --js-flags=--no-lazy --disable-mojo-local-storage --no-sandbox --disable-features=ColorCorrectRendering --service-pipe-token=0B
cursoredes+ 2611  2593  0  20:58 ?    00:00:01 /usr/share/code/code /usr/share/code/resources/app/out/bootstrap --type=extensionHost
cursoredes+ 2612  2593  0  20:58 ?    00:00:00 /usr/share/code/code /usr/share/code/resources/app/out/bootstrap --type=watcherService
cursoredes+ 3347    1  0  21:29 ?    00:00:00 /usr/libexec/gnome-terminal-server
cursoredes+ 3354  3347  0  21:29 ?    00:00:00 gnome-pty-helper
cursoredes+ 3355  3347  0  21:29 pts/0  00:00:00 bash
cursoredes+ 10046 2611  0  21:33 ?    00:00:00 /usr/share/code/code /usr/share/code/resources/app/extensions/json-language-features/server/dist/jsonServerMain --mode=ipc --clientProcessId=2611
cursoredes+ 10098 3347  0  21:37 pts/1  00:00:00 bash
cursoredes+ 10144 10098 0  21:37 pts/1  00:00:00 ps -u cursoredes -f
[cursoredes@localhost ~]$
```

**ps -eo pid,gid,sid,s,command** muestra los procesos del sistema, incluyendo PID, del grupo, la sesión, el estado y la línea de comandos.

```
[cursoredes@localhost ~]$ ps -eo pid,gid,sid,s
PID    GID    SID    S
1      0      1      S
2      0      0      S
3      0      0      S
5      0      0      S
6      0      0      S
7      0      0      S
8      0      0      S
9      0      0      R
10     0      0      S
11     0      0      S
13     0      0      S
14     0      0      S
~      ~      ~      ~
```

**Ejercicio 5.** Escribir un programa que muestre los identificadores del proceso (PID, PPID, PGID y SID), el número máximo de ficheros que puede abrir y su directorio de trabajo actual.

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

int main() {
    // mostramos identificador pid
    printf("PID: %i\n", getpid());

    // mostramos identificador ppid
    printf("PPID: %i\n", getppid());

    // mostramos identificador pgid
    printf("PGID: %i\n", getpgid(getpid()));

    // mostramos identificador sid
    printf("SID: %i\n", getsid(getpid()));

    // calculamos y mostramos el max de ficheros que puede abrir
    struct rlimit limit;
    if (getrlimit(RLIMIT_NOFILE, &limit) == -1) {
        perror("resource limits error");
        return -1;
    }
    printf("MAX LIMIT: %li\n", limit.rlim_max);

    // calculamos y mostramos el dir de trabajo actual (reservando)
    char *reserv_path = malloc(sizeof(char)*(4096 + 1));
    char *absolute_path = getcwd(reserv_path, 4096 + 1);
    printf("CWD: %s\n", reserv_path);
    free (reserv_path);

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej5 /home/cursoredes/p2.3/ej5.c
[cursoredes@localhost ~]$ ./outputej5
PID: 11138
PPID: 10098
PGID: 11138
SID: 10098
MAX LIMIT: 4096
CWD: /home/cursoredes
[cursoredes@localhost ~]$
```

**Ejercicio 6.** Un **demonio** es un proceso que se ejecuta en segundo plano para proporcionar un servicio. Normalmente, un demonio está en su propia sesión y grupo. Para garantizar que es posible crear la sesión y el grupo, el demonio crea un nuevo proceso para crear la nueva sesión y ejecutar la lógica del servicio. Escribir una plantilla de demonio (creación del nuevo proceso y de la sesión) en el que únicamente se muestren los atributos del proceso (como en el ejercicio anterior). Además, fijar el directorio de trabajo del demonio a /tmp.

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

void print_atributos(char *type){
    // mostramos identificador pid
    printf("PID: %i\n", getpid());

    // mostramos identificador ppid
    printf("PPID: %i\n", getppid());

    // mostramos identificador pgid
    printf("PGID: %i\n", getpgid(getpid()));

    // mostramos identificador sid
    printf("SID: %i\n", getsid(getpid()));

    // calculamos y mostramos el max de ficheros que puede abrir
    struct rlimit limit;
    if (getrlimit(RLIMIT_NOFILE, &limit) == -1) {
        perror("resource limits error");
    }
    else{
        printf("MAX LIMIT: %li\n", limit.rlim_max);

        // calculamos y mostramos el dir de trabajo actual (reservando)
        char *reserv_path = malloc(sizeof(char)*(4096 + 1));
        char *absolute_path = getcwd(reserv_path, 4096 + 1);
        printf("CWD: %s\n", reserv_path);
        free (reserv_path);
    }
}

int main() {
    // creamos 1 proceso hijo
    pid_t pid = fork();

    switch (pid) {
        // caso fallo
        case -1:
            perror("fork");
            exit(-1);
            break;

        // caso ejecutando el hijo
        case 0:
            // creamos sesion
            setsid();

            //cambiamos el dir de trabajo del demonio a tmp
            chdir("/tmp");

            //mostramos
            printf("[Hijo] Proceso %i (Padre: %i)\n",getpid(),getppid());
            print_atributos("Hijo");

            // dormimos para que termine antes el padre:
            //sleep (3);
            break;

        // >0 caso ejecutando el padre (el valor es el pid del hijo)
        default:
            // dormimos para que termine antes el hijo:
            //sleep (3);
            printf("[Padre] Proceso %i (Padre: %i)\n",getpid(),getppid());
            print_atributos("Padre");
            break;
    }
    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej6 /home/cursoredes/p2.3/ej6.c
[cursoredes@localhost ~]$ ./outputej6
[Padre] Proceso 11414 (Padre: 10098)
PID: 11414
PPID: 10098
PGID: 11414
SID: 10098
MAX LIMIT: 4096
CWD: /home/cursoredes
[Hijo] Proceso 11415 (Padre: 1)
PID: 11415
PPID: 1
PGID: 11415
SID: 11415
MAX LIMIT: 4096
CWD: /tmp
[cursoredes@localhost ~]$ sudo gcc -o outputej6 /home/cursoredes/p2.3/ej6.c
[cursoredes@localhost ~]$ ./outputej6
[Padre] Proceso 11473 (Padre: 10098)
PID: 11473
PPID: 10098
PGID: 11473
SID: 10098
MAX LIMIT: 4096
CWD: /home/cursoredes
[Hijo] Proceso 11474 (Padre: 1)
PID: 11474
PPID: 1
PGID: 11474
SID: 11474
MAX LIMIT: 4096
CWD: /tmp
[cursoredes@localhost ~]$ sudo gcc -o outputej6 /home/cursoredes/p2.3/ej6.c
[cursoredes@localhost ~]$ ./outputej6
[Hijo] Proceso 11507 (Padre: 11506)
PID: 11507
PPID: 11506
PGID: 11507
SID: 11507
MAX LIMIT: 4096
CWD: /tmp
[Padre] Proceso 11506 (Padre: 10098)
PID: 11506
PPID: 10098
PGID: 11506
SID: 10098
MAX LIMIT: 4096
CWD: /home/cursoredes
[cursoredes@localhost ~]$ █
```

A

B

C

- A: ejecución sin sleeps
- B: ejecución con sleep en el hijo para que termine antes el padre
- C: ejecución con sleep en el padre para que termine antes el hijo

¿Qué sucede si el proceso padre termina antes que el hijo (observar el PPID del proceso hijo)? **El hijo se queda huérfano y el ppid lo recoge el init o la shell**

¿Y si el proceso que termina antes es el hijo (observar el estado del proceso hijo con ps)? **El proceso se queda esperando**

**Nota:** Usar `sleep(3)` o `pause(3)` para forzar el orden de finalización deseado.

## Ejecución de programas

**Ejercicio 7.** Escribir dos versiones, una con `system(3)` y otra con `execvp(3)`, de un programa que ejecute otro programa que se pasará como argumento por línea de comandos. En cada caso, se debe imprimir la cadena “El comando terminó de ejecutarse” después de la ejecución. ¿En qué casos se imprime la cadena? ¿Por qué?

**Nota:** Considerar cómo deben pasarse los argumentos en cada caso para que sea sencilla la implementación. Por ejemplo: ¿qué diferencia hay entre `./ej7 ps -e1` y `./ej7 “ps -e1”`?

### Programa con `execvp`:

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

int main(int argc, char **argv){
    //ejecutamos con execvp el programa que nos pasan como argumento
    if (execvp(argv[1], argv + 1) == -1) {
        printf("ERROR\n");
    }

    printf("El comando terminó de ejecutarse.\n");

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej7_execvp /home/cursoredes/p2.3/ej7_execvp.c
[cursoredes@localhost ~]$ ./outputej7_execvp
Segmentation fault (core dumped)
[cursoredes@localhost ~]$ █
```

### Programa con `system`

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

int main(int argc, char **argv){
    //Concatenamos todo lo que nos pase por pantalla:

    // calculamos el tamaño del argumento que nos pasan
    int longitud = 1;
    int i;
    for (i = 1; i < argc; i++)
        longitud += strlen(argv[i]) + 1;

    // reservamos espacio en memoria para dicho tamaño
    char *cmd = malloc(sizeof(char)*longitud);
    strcpy(cmd, "");

    // concatenamos los argumentos
    for (i = 1; i < argc; i++) {
        strcat(cmd, argv[i]);
        strcat(cmd, " ");
    }

    // ejecutamos con system el programa que nos pasan como argumento
    if (system(cmd) == -1) {
        printf("ERROR\n");
    }

    printf("El comando terminó de ejecutarse.\n");

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej7_system /home/cursoredes/p2.3/ej7_system.c
[cursoredes@localhost ~]$ ./outputej7_system
El comando terminó de ejecutarse.
[cursoredes@localhost ~]$ █
```



La cadena de “el comando terminó de ejecutarse” solo se imprime cuando se usa system, ya que con exec estamos sustituyendo la imagen del programa con la imagen del programa pasado por args.

En cuanto a la diferencia entre ps -el y "ps -el", cabe destacar que cuando se pasa por parámetros ps -ef equivaldría a dos argumentos y si se quiere ejecutar como system sería necesario unirlos. Por ello, "ps -el" equivale a un único parámetro lo que nos permitiría ejecutar directamente el comando system sin necesidad de unirlos.

**Ejercicio 8.** Usando la versión con execvp(3) del ejercicio 7 y la plantilla de demonio del ejercicio 6, escribir un programa que ejecute cualquier programa como si fuera un demonio. Además, redirigir los flujos estándar asociados al terminal usando dup2(2):

- La salida estándar al fichero /tmp/daemon.out.
- La salida de error estándar al fichero /tmp/daemon.err.
- La entrada estándar a /dev/null.

Comprobar que el proceso sigue en ejecución tras cerrar la shell. Sí, porque no se imprime “el comando terminó de ejecutarse” que hemos puesto al final del código de ejecución del hijo.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv){

    // creamos 1 proceso hijo
    pid_t pid = fork();

    switch (pid) {
        // caso fallo
        case -1:
            perror("fork");
            exit(-1);
            break;

        // caso ejecutando el hijo
        case 0:
            // creamos sesion
            setsid();
            printf("[Hijo] Proceso %i (Padre: %i)\n", getpid(), getppid());
            int fd = open("/tmp/daemon.out", O_CREAT | O_RDWR, 00777);
            int fderr = open("/tmp/daemon.err", O_CREAT | O_RDWR, 00777);
            int null = open("/dev/null", O_CREAT | O_RDWR, 00777);
            int fd2 = dup2(fd, 2);
            int fd3 = dup2(fderr, 1);
            int fd4 = dup2(null, 0);

            //ejecutamos con execvp el programa que nos pasan como argumento (ej7_execvp)
            if (execvp(argv[1], argv + 1) == -1) {
                printf("ERROR\n");
            }

            printf("El comando terminó de ejecutarse.\n");
            break;

        // >0 caso ejecutando el padre (el valor es el pid del hijo)
        default:
            printf("[Padre] Proceso %i (Padre: %i)\n", getpid(), getppid());
            break;
    }

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej8 /home/cursoredes/p2.3/ej8.c
[cursoredes@localhost ~]$ ./outputej8
[Padre] Proceso 11865 (Padre: 10098)
[Hijo] Proceso 11866 (Padre: 1)
[cursoredes@localhost ~]$
```

## Señales

**Ejercicio 9.** El comando `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 del comando y las señales que se pueden enviar a un proceso.

man kill

man pkill

```
KILL(1) User Commands KILL(1)
NAME
    kill - terminate a process

SYNOPSIS
    kill [-s signal] [-p] [-q signal] [-a] [-l] pid...
    kill -t [signal]

DESCRIPTION
    The command kill sends the specified signal to the specified process or process group. If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

    Most modern shells have a builtin kill function, with a usage rather similar to that of the command described here. The '-a' and '-p' options, and the possibility to specify processes by command name are a local extension.

    If sig is 0, then no signal is sent, but error checking is still performed.

OPTIONS
    pid... Specify the list of processes that kill should signal. Each pid can be one of five things:
        0       where g is larger than 0. The process with pid g will be signaled.
        0       All processes in the current process group are signaled.
        -1      All processes with pid larger than 1 will be signaled.
        -g      where g is larger than 1. All processes in process group g are signaled. When an argument of the form '-n' is given, and it is meant to denote a process group, either the signal must be specified first, or the argument must be preceded by a '-' option, otherwise it will be taken as the signal to send.
    commandname
        All processes invoked using that name will be signaled.
    -s, --signal signal
        Specify the signal to send. The signal may be given as a signal name or number.
    -l, --list [signal]
        Print a list of signal names, or convert signal given as argument to a name. The signals are found in /usr/include/linux/signal.h
    -L, --table
        Similar to -l, but will print signal names and their corresponding numbers.
    -a, --all
        Do not restrict the commandname-to-pid conversion to processes with the same uid as the present process.
    -p, --pid
        Specify that kill should only print the process id (pid) of the named processes, and not send any signals.
    -q, --quote signal
        Use sigqueue(2) rather than kill(2) and the signal argument is used to specify an integer to be sent with the signal. If the receiving process has installed a handler for this signal using the SA_SIGINFO flag to sigaction(2), then it can obtain this data via the si_value field of the siginfo_t structure.

NOTES
    It is not possible to send a signal to explicitly selected thread in a multithreaded process by kill(2) syscall. If kill(2) is used to send a signal to a thread group, then kernel selects arbitrary member of the thread group that has not blocked the signal. For more details see clone(2) CLONE_THREAD description.

    The command kill(1) as well as syscall kill(2) accepts TID (thread ID, see gettid(2)) as argument. In this case the kill behavior is not changed and the signal is also delivered to the thread group rather than to the specified thread.
```

## Todas las señales de sistema :

```
[cursoredes@localhost ~]$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

**Ejercicio 10.** En un terminal, arrancar un proceso de larga duración (ej. `sleep 600`). En otra terminal, enviar diferentes señales al proceso, comprobar el comportamiento. Observar el código de salida del proceso. ¿Qué relación hay con la señal enviada?

Ejemplo:

Terminal 1: `sleep 600` (con echo \$\$ sé que es 12375)

Terminal 2: `kill -1 12375` y la cierra

### Resumen señales:

- **SIGHUP:**

Comportamiento: Se ha terminado el proceso (Se ha desconectado de la SHELL).

Salida: Colgar (hangup)

- **SIGINT:**

Comportamiento: Se ha interrumpido el sleep.

Salida: nada

- **SIGQUIT:**

Comportamiento: Se ha interrumpido el sleep.

Salida: Abandona (core generado)

- **SIGILL:**

Comportamiento: Se ha interrumpido el sleep.

Salida: Instrucción ilegal (core generado)

- **SIGTRAP:**

Comportamiento: Se ha interrumpido el sleep

Salida: «trap» para punto de parada/seguimiento (core generado)

- **SIGKILL:**

Comportamiento: Se ha interrumpido el sleep

Salida: Terminado (killed)

- **SIGBUS:**

Comportamiento: Se ha interrumpido el sleep

Salida: Error del bus (core generado)

- **SIGSEGV:**

Comportamiento: Se ha interrumpido el sleep

Salida: Violación de segmento (core generado)

- **SIGPIPE:**

Comportamiento: Se ha interrumpido el sleep

Salida: N/A (Ha salido una notificación en la pantalla).

- **SIGTERM:**

Comportamiento: Se ha interrumpido el sleep

Salida: Terminado

**Ejercicio 11.** Escribir un programa que bloquee las señales SIGINT y SIGTSTP. Después de bloquearlas el programa debe suspender su ejecución con `sleep(3)` un número de segundos que se obtendrán de la variable de entorno `SLEEP_SECS`. Al despertar, el proceso debe informar de si recibió la señal SIGINT y/o SIGTSTP. En este último caso, debe desbloquearla con lo que el proceso se detendrá y podrá ser reanudado en la *shell* (imprimir una cadena antes de finalizar el programa para comprobar este comportamiento).

## Creamos la variable de entorno

```
[cursoredes@localhost ~]$ SLEEP_SECS='5'
[cursoredes@localhost ~]$ export SLEEP_SECS
```

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

int main(int argc, char **argv) {
    sigset_t set;

    // inicializamos las señales
    sigemptyset(&set);

    // añadimos las señales de int y tstp
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTSTP);

    // protegemos la región de código contra la recepción de las señales
    sigprocmask(SIG_BLOCK, &set, NULL);

    // obtenemos la variable de entorno
    char *sleep_secs = getenv("SLEEP_SECS");
    int secs = atoi(sleep_secs);
    printf("El proceso se va a dormir durante %d s\n", secs);

    //Dormimos el proceso
    sleep(secs);

    // comprobamos las señales pendientes
    // señal int
    sigset_t pending;
    sigpending(&pending);
    if (sigismember(&pending, SIGINT) == 1) {
        printf("Se ha recibido la señal SIGINT\n");
        //Eliminamos la señal del conjunto anterior
        sigdelset(&set, SIGINT);
    }

    else {
        printf("No se ha recibido la señal SIGINT\n");
    }

    // señal tstp
    if (sigismember(&pending, SIGTSTP) == 1) {
        printf("Se ha recibido la señal SIGTSTP\n");
        //Eliminamos la señal del conjunto anterior
        sigdelset(&set, SIGTSTP);
    }
    else {
        printf("No se ha recibido la señal SIGTSTP\n");
    }

    sigprocmask(SIG_UNBLOCK, &set, NULL); //to fetch and/or change the signal mask of the calling thread.

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej11 /home/cursoredes/p2.3/ej11.c
[cursoredes@localhost ~]$ ./outputej11
El proceso se va a dormir durante 5 s
No se ha recibido la señal SIGINT
No se ha recibido la señal SIGTSTP
```

**Ejercicio 12.** Escribir un programa que instale un manejador para las señales SIGINT y SIGTSTP. El manejador debe contar las veces que ha recibido cada señal. El programa principal permanecerá en un bucle que se detendrá cuando se hayan recibido 10 señales. El número de señales de cada tipo se mostrará al finalizar el programa.

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

volatile int int_count = 0;
volatile int tstp_count = 0;

// configuramos la funcion del manejador
void manejador(int senial){
    if (senial == SIGINT) int_count++;
    if (senial == SIGTSTP) tstp_count++;
}

int main(){
    struct sigaction act;

    // señal int
    sigaction(SIGINT, NULL, &act); //get manejador
    act.sa_handler = manejador;
    sigaction(SIGINT, &act, NULL); //set sa_manejador

    // señal tstp
    sigaction(SIGTSTP, NULL, &act); //set manejador
    act.sa_handler = manejador;
    sigaction(SIGTSTP, &act, NULL); //set sa_manejador

    sigset_t set;
    sigemptyset(&set);

    // permanecemos en bucle hasta que se hayan recibido 10 señales
    while (int_count + tstp_count < 10)
        sigsuspend(&set);

    // mostramos el total de señales recibidas de cada tipo
    printf("SIGINT recibidas: %i\n", int_count);
    printf("SIGTSTP recibidas: %i\n", tstp_count);

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej12 /home/cursoredes/p2.3/ej12.c
[cursoredes@localhost ~]$ ./outputej12
^C^C^C^C^C^C^C^C^CSIGINT recibidas: 10
SIGTSTP recibidas: 0
[cursoredes@localhost ~]$
```

**Ejercicio 13.** Escribir un programa que realice el borrado programado del propio ejecutable. El programa tendrá como argumento el número de segundos que esperará antes de borrar el fichero. El borrado del fichero se podrá detener si se recibe la señal SIGUSR1.

**Nota:** Usar `sigsuspend(2)` para suspender el proceso y la llamada al sistema apropiada para borrar el fichero.

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

volatile int stop = 0;

void manejador(int senial){
    // si recibe la señal usr1 se podra detener
    if (senial == SIGUSR1) stop = 1;
}

int main(int argc, char **argv) {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR1);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    struct sigaction act;

    // señal int
    sigaction(SIGUSR1, NULL, &act); //get manejador
    act.sa_handler = manejador;
    sigaction(SIGUSR1, &act, NULL); //set sa_manejador

    // pasamos a int el num de segundos que nos pasan por args
    int secs = atoi(argv[1]);
    int i = 0;
    // esperamos dichos segundos antes de seguir
    while (i < secs && stop == 0) {
        i++;
        sleep(1);
    }

    if (stop == 0) {
        printf("Borramos fichero");
        unlink(argv[0]);
    } else {
        printf("No borramos fichero. Se recibio la señal usr1");
    }

    return 0;
}
```

```
[cursoredes@localhost ~]$ sudo gcc -o outputej13 /home/cursoredes/p2.3/ej13.c
[cursoredes@localhost ~]$ ./outputej13 6
Borramos fichero[cursoredes@localhost ~]$ █
```