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

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 de 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.

```
chrt [options] -p <pid>
chrt [options] -p [priority] pid


Options:
-a | --all-tasks operate on all the tasks (threads) for a given pid
-h | --help display this help
-m | --max show min and max valid priorities
-p | --pid operate on existing given pid
-v | --verbose display status information
-V | --version output version information

Scheduling policies:
-b | --batch set policy to SCHED_BATCH
-f | --fifo set policy to SCHED_FIFO
-i | --idle set policy to SCHED_IDLE
-o | --other set policy to SCHED_OTHER
-r | --rr set policy to SCHED_RR (default)
```

Ejemplo de uso:

```
$ pidof bash
3284
$ chrt -a -p 3284
pid 3284's current scheduling policy: SCHED_OTHER
pid 3284's current scheduling priority: 0
```

El comando nice sirve para asignar prioridades a comandos (tareas) ejecutados desde el terminal. Si no hay comando, muestra el valor de 'nice' actual. El rango de valores de 'nice' abarca desde -20 (más favorable al proceso) hasta 19 (menos favorable al proceso).

```
nice [OPTION] [COMMAND [ARG]...]
-n, --adjustment=N add integer N to the niceness (default 10)
```

Ejemplo de uso:

```
nice -n10 /bin/bash/
```

Para ver la prioridad del proceso, hacer `ps -al`. NI es la prioridad de la tarea en el planificador. PRI es la prioridad actual del proceso.

El comando renice tiene el mismo funcionamiento que nice, pero cambia la prioridad de la tarea una vez ejecutada en el proceso de la shell.

```
renice [-n] <priority> [-p|--pid] <pid>...
renice [-n] <priority> -g|--pgrp <pgid>...
renice [-n] <priority> -u|--user <user>...
```

Cambiar el "nice" de la shell a -10 y su política de planificación a FIFO con prioridad 12.

Ejemplo de uso:

```
renice -n -10 -p 2724
chrt -f -p 12 2724
```

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.

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sys/sysmacros.h>
#include <string.h>
#include <sched.h>

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

	pid_t pid = getpid();
	int scheduler = sched_getscheduler(pid);
```

```c
    /* Consultar "man sched" para determinar el valor de retorno. */
      printf("* La politica de planificacion (pid = %i):\n  ", pid);
      switch(scheduler){
         case SCHED_RR:
              printf("Round-robin | SCHED_RR = %i\n", SCHED_RR); break;
         case SCHED_OTHER:
              printf("Default Linux time-sharing | SCHED_OTHER = %i\n", SCHED_OTHER); break;
         case SCHED_FIFO:
              printf("FIFO  | SCHED_FIFO = %i\n", SCHED_FIFO); break;
         default:
              printf("Error (No se ha encontrado)\n"); break;
      }

      /* Prioridad */
      struct sched_param p;
      sched_getparam(pid, &p);
      printf("\n* La prioridad es: %i\n\n", p.sched_priority);

      /* Valores máximo y mínimo de la prioridad para la política de planificación */
      int min = sched_get_priority_min(scheduler);
      int max = sched_get_priority_max(scheduler);
      printf("* El valor MINIMO de la prioridad es: %i\n", min);
      printf("* El valor MAXIMO de la prioridad es: %i\n", max);

   return 0;
}
```

OUTPUT:

```
* La politica de planificacion (pid = 99558):
  Default Linux time-sharing  | SCHED_OTHER = 0

* La prioridad es: 0

* El valor MINIMO de la prioridad es: 0
* El valor MAXIMO de la prioridad es: 0
```

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?

Si se heredan los atributos de planificación, ahora el programa tendrá una prioridad de 12.

```
$ps
      PID TTY      TIME CMD
   9299 pts/000:00:00 bash
   9555 pts/000:00:00 ps

$ sudo chrt -f -p 12 9299

$ ./ej2
* La politica de planificacion (pid = 9559): FIFO  | SCHED_FIFO = 1
* La prioridad es: 12
* El valor MINIMO de la prioridad es: 1
* El valor MAXIMO de la prioridad es: 99
```

Grupos de procesos y sesiones

Los grupos de procesos y 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 la línea de comandos.
- 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 comando ps hace un reporte de los procesos actuales.
`ps [options]`

Esta versión de ps acepta distintos tipos de opciones:

1. Opciones de UNIX, las cuales pueden ser agrupadas y deben estar precedidas por un guión.
2. Opciones BDS, las cuales deben estar agrupadas y no se suele usar con guiones.
3. Opciones largas GNU, las cuales están precedidas por dos guiones.

- **Mostrar todos los procesos del usuario actual en formato extendido**

```
$ ps -F (extra full)
$ ps -f (full-format, including command lines)
$ ps -l (long format)
```

- Mostrar los procesos del sistema, incluyendo el identificador del proceso, del grupo, la sesión, el estado y la línea de comandos.

```
$ ps -o pid,pgid,sess,state,cmd
```

- Observar el identificador de proceso, grupo 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?

```
$ ps -o pid,pgid,sess
```

```
  PID    PGID    SESS
14737   14737   14429
14740   14740   14429
14743   14743   14429
14744   14744   14429
15023   15023   14429
```

Comparten el identificador de sesión. El identificador de grupo de procesos cuando se crea un nuevo proceso es el mismo que el del proceso.

Ejercicio 5. Escribir un programa que muestre los identificadores del proceso: identificador de proceso, de proceso padre, de grupo de procesos y de sesión. Mostrar además el número máximo de archivos que puede abrir el proceso y el directorio de trabajo actual.

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

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

    pid_t pid = getpid();
    pid_t pgid = getpgid(pid);
    pid_t ses_id = getsid(pid);


    struct rlimit rlim;
    getrlimit(RLIMIT_NOFILE, &rlim);

    char dir[1024];
    getcwd(dir, sizeof(dir));

    printf("ID del proceso: %i \n", pid);
    printf("PGID del proceso: %i \n", pgid);
    printf("Identificador de sesión %i \n", ses_id);
    printf("Número máximo de ficheros que puede abrir: %lli\n",(long long int)rlim.rlim_max);
    printf("Directorio de trabajo: %s \n", dir);

    return 0;
}
```

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 ejecutar la lógica del servicio y crear la nueva sesión. 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`.

¿Qué sucede si el proceso padre termina antes que el hijo (observar el PPID del proceso hijo)? ¿Y si el proceso que termina antes es el hijo (observar el estado del proceso hijo con `ps`)?

El hijo se queda huérfano, entonces el proceso init hereda al hijo.
Si el hijo termina antes que el padre se convierte en zombie.

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

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

```c
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

int showProcessInfo(char *id) {
    if (id == NULL)
        return -1;

    pid_t pid = getpid();
    gid_t gid = getgid();
    pid_t sid = getsid(pid);

    struct rlimit lim;
    int rc = getrlimit(RLIMIT_NOFILE, &lim);

    char *path = malloc(sizeof(char)*(4096 + 1));
    getcwd(path, 4096 + 1);

    printf("[%s] PID: %i\n", id, pid);
    printf("[%s] GID: %i\n", id, gid);
    printf("[%s] SID: %i\n", id, sid);
    printf("[%s] Max num of files: %li\n", id, lim.rlim_max);
    printf("[%s] Directorio trabajo: %s\n", id, path);

    free (path);
    return 0;
}

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

       if(pid == -1){
          printf("Error en fork\n");
       }
       if(pid == 0){ //Proceso Hijo
          pid_t nsid = setsid();
          chdir("/tmp");
          printf("[Hijo] Proceso %i (Padre: %i)\n",getpid(),getppid());

          showProcessInfo("Hijo");
       }
       else{ //Proceso padre
          printf("[Padre] Proceso %i (Padre: %i)\n",getpid(),getppid());

          showProcessInfo("Padre");
          sleep(2);
       }
       return 0;
}
```

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?

PROGRAMA CON **"execvp"**

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

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

  if (argc < 2) {
       printf("ERROR: Introduce el comando.\n");
       return -1;
  }

  //Ejecutamos el comando correspondiente a la entrada por argumentos
  execvp(argv[1], argv + 1);

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

  return 0;
}
```

Sólo se ejecutará "El comando terminó de ejecutarse." si la llamada a la función execvp falla, ya que si se ejecuta correctamente se cambia de proceso y no vuelve a ejecutar el inicial.

PROGRAMA CON **"system"**

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

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

  if (argc < 2) {
       printf("ERROR: Introduce el comando.\n");
       return -1;
  }

  char cmd[1024] = "";
  for(int i = 1; i < argc; i++){
       strcat(cmd, argv[i]);
       strcat(cmd, " ");
  }

  system(cmd);

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

```
  return 0;
}
```

Cómo system hace un fork y ejecuta en el proceso hijo execl, siempre imprimirá "El comando terminó de ejecutarse."

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

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*.

```
#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){

  if (argc < 2) {
       printf("ERROR: Introduce el comando.\n");
       return -1;
  }

  pid_t pid = fork();

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

       //Proceso hijo
       case 0:;
       pid_t mi_sid = setsid(); //Creamos una nueva sesión
       printf("[Hijo] Proceso %i (Padre: %i)\n",getpid(),getppid());
```

```
        int fdout = open("/tmp/daemon.out",O_CREAT | O_RDWR, 0777);
        int fderr = open("/tmp/daemon.err", O_CREAT | O_RDWR, 0777);
        int fdin = open("/dev/null", O_CREAT | O_RDWR, 0777);

        int fd2 = dup2(fdout,2);
        int fd3 = dup2(fderr, 1);
        int fd4 = dup2(fdin, 0);

        execvp(argv[1], argv + 1);
        break;

        //Proceso padre
        default:
        printf("[Padre] Proceso %i (Padre: %i)\n",getpid(),getppid());
        break;
    }

    return 0;
}
```

Señales

Ejercicio 9. El comando `kill(1)` permite enviar señales a un proceso o grupo de procesos por su identificador (`pkill` permite hacerlo por nombre de proceso). Estudiar la página de manual del comando y las señales que se pueden enviar a un proceso.

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?

En una terminal ejecuto "sleep 600", desde otra terminal miro que pid tiene el proceso de dos maneras:

1. **$ ps -e** (muestra todos los procesos)
2. **$ pidof sleep** (muestra solo el pid del proceso sleep)

A continuación podemos enviar al proceso sleep señales de la siguiente manera:

1. **$ kill -9 4728** (Con el número de la señal obtenido con "$ kill -l"
2. **$ kill -SIGILL** 4728 (Con el nombre de la señal)

Salidas de la terminación del proceso sleep probando varias señales:

```
$ sleep 600
Killed

$ sleep 600

$ sleep 600
Hangup

$ sleep 600
Illegal instruction (core dumped)
```

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`.

Después de despertar de `sleep(3)`, 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).

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


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

    if(argc != 1){
        printf("ERROR en los parametros del programa.\n");
    }
    printf("Pid del proceso: %d.\n", getpid());

    char *sleep_c = getenv("SLEEP_SECS");
    int  sleep_i = 15;
    if(sleep_c != NULL)
        sleep_i = atoi(sleep_c);

    printf("Se va a dormir el proceso durante %d seg.\n", sleep_i);

    //Inicializamos un conjunto de señales y añadimos senales SIGINT y SIGTSTP
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTSTP);

    //Protegemos la region de codigo contra la recepcion de las senales.
    sigprocmask(SIG_BLOCK, &set, NULL);

    sleep(sleep_i);

    //Comprobamos las señales pendientes
    sigset_t set_pending;
    sigpending(&set_pending);

    if(sigismember(&set_pending, SIGINT) == 1){
        printf("Se ha recibido la señal SIGINT\n");
    }
    else{ printf("No se ha recibido la señal SIGINT\n");}

    if(sigismember(&set_pending, SIGTSTP) == 1){
        printf("Se ha recibido la señal SIGTSTP\n");
        sigprocmask(SIG_UNBLOCK, &set, NULL);
```

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

    return 0;
}
```

Para la ejecución, abro otra terminal y envio las señales que quiero de esta manera:
$ Kill -n 2 8816
$ Kill -n 20 8816 (Para este ejemplo de ejecución solo envio esta señal)

Desde la terminal principal de ejecución del programa la salida es la siguiente:

```
$ gcc ej11m.c -o ej11
$ ./ej11
Pid del proceso: 8816.
No se ha recibido la señal SIGINT
Se ha recibido la señal SIGTSTP

[2]+  Stopped                 ./ej11
```

Ejercicio 12. Escribir un programa que instale un manejador sencillo 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>
#include <unistd.h>

volatile int int_c = 0;
volatile int tstp_c = 0;

void hler(int senial){
  if (senial == SIGINT) int_c++;
  if (senial == SIGTSTP) tstp_c++;
}

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

  printf("Pid del proceso: %d.\n", getpid());
  struct sigaction act;

  //SIGINT
  sigaction(SIGINT, NULL, &act); //Get handler
  act.sa_handler = hler;
  sigaction(SIGINT, &act, NULL); //Set sa_handler

  //SIGTSTP
  sigaction(SIGTSTP, NULL, &act); //Get handler
  act.sa_handler = hler;
  sigaction(SIGTSTP, &act, NULL); //Set sa_handler
```

```
    while (int_c + tstp_c < 10)
        pause();

    printf("Numero de señales SIGINT recibidas: %i\n", int_c);
    printf("Numero de señales SIGTSTP recibidas: %i\n", tstp_c);

  return 0;
}
```

Desde otro terminal mando las señales **SIGINT** y **SIGTSTP** a este proceso.

```
$ ./ej12
Pid del proceso: 9011.
Numero de señales SIGINT recibidas: 6
Numero de señales SIGTSTP recibidas: 4
```

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>


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

    if(argc != 2){
        printf("ERROR en los parametros del programa.\n");
        return -1;
    }
    printf("Pid del proceso: %d.\n", getpid());


    int  sleep_i = atoi(argv[1]);
    printf("Se va a dormir el proceso durante %d seg.\n", sleep_i);

    //Inicializamos un conjunto de señales y añadimos senales SIGINT y SIGTSTP
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);

    //Protegemos la region de codigo contra la recepcion de las senales.
    sigprocmask(SIG_BLOCK, &set, NULL);

    sleep(sleep_i);
```

```
    //Comprobamos las señales pendientes
    sigset_t set_pending;
    sigpending(&set_pending);

    if(sigismember(&set_pending, SIGUSR1) == 1){
        printf("Has tenido suerte, hemos recibido la señal SIGUSR1.\n");
        return 0;
    }
    else{
        printf("No se ha recibido la señal SIGUSR1 -> Se elimina este ejecutable.\n");
        unlink(argv[0]);
    }
    return 0;
}
```

En este programa el proceso se duerme siempre los segundos indicados y luego comprueba las señales que se han recibido y realiza lo oportuno

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

volatile int stop = 0;

void hler(int senial){
  if (senial == SIGUSR1) stop = 1;
}

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

    if(argc != 2){
        printf("ERROR en los parametros del programa.\n");
        return -1;
    }
    printf("Pid del proceso: %d.\n", getpid());

    int  sec = atoi(argv[1]);
    printf("Se va a dormir el proceso durante %d seg.\n", sec);

    struct sigaction act;

    //SIGUSR1
    sigaction(SIGUSR1, NULL, &act); //Get handler
    act.sa_handler = hler;
    sigaction(SIGUSR1, &act, NULL); //Set sa_handler

    sleep(sec);

    if(stop == 0){
        printf("No se ha recibido la señal SIGUSR1 -> Se elimina este ejecutable.\n");
        unlink(argv[0]);
    }
    else{ printf("Has tenido suerte, hemos recibido la señal SIGUSR1.\n"); }
```

```
    return 0;
}
```

En este programa, a diferencia del anterior, el proceso se duerme los segundos indicados pero en cuanto llega la señal se despierta y realiza el tratamiento (indicar que el stop = 1) y continua el código sin tener que esperar al sleep entero.