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.

NICE(1) User Commands NICE(1)

NAME
nice - run a program with modified scheduling priority

SYNOPSIS
nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
Run COMMAND with an adjusted niceness, which affects process scheduling. With no COMMAND, print the current niceness. Niceness
values range from -20 (most favorable to the process) to 19 (least favorable to the process).

Mandatory arguments to long options are mandatory for short options too.

-n, --adjustment=N
add integer N to the niceness (default 10)

--help display this help and exit

--version
output version information and exit

NOTE: your shell may have its own version of nice, which usually supersedes the version described here. Please refer to your
shell's documentation for details about the options it supports.

GNU coreutils online help: <http://www.gnu.org/software/coreutils/> Report nice translation bugs to <http://translationpro-
ject.org/team/>

RENICE(1) User Commands RENICE(1)

NAME
renice - alter priority of running processes

SYNOPSIS
renice [-n] priority [-gpu] identifier...

DESCRIPTION
renice alters the scheduling priority of one or more running processes. The first argument is the priority value to be used. The
other arguments are interpreted as process IDs (by default), process group IDs, user IDs, or user names. renice'ing a process
group causes all processes in the process group to have their scheduling priority altered. renice'ing a user causes all processes
owned by the user to have their scheduling priority altered.

OPTIONS
-n, --priority priority
Specify the scheduling priority to be used for the process, process group, or user. Use of the option -n or --priority is
optional, but when used it must be the first argument.

-g, --pgrp pgid...
Force the succeeding arguments to be interpreted as process group IDs.

-u, --user name_or_uid...
Force the succeeding arguments to be interpreted as usernames or UIDs.

-p, --pid pid...
Force the succeeding arguments to be interpreted as p

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 <sys/types.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <sched.h>

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

   pid_t pid = atoi(argv[1]);
   struct sched_param p;
   sched_getparam(pid, &p);
    int polplanification = sched_getscheduler(pid);
   if(polplanification == 1) {
         printf("Planificación: %s\n", "SCHED_FIFO");
   }else if(polplanification == 3) {
            printf("Planificación: %s\n", "SCHED_BATCH");
   } else if(polplanification == 5) {
         printf("Planificación: %s\n", "SCHED_IDLE");
   } else if(polplanification == 0) {
         printf("Planificación: %s\n", "SCHED_OTHER");
   } else if(polplanification == 2) {
         printf("Planificación: %s\n", "SCHED_RR");
   }
   printf("Prioridad: %i\n", p.sched_priority);
   int maxprio = sched_get_priority_max(polplanification);
   int minprio = sched_get_priority_min(polplanification);
   printf("Mínimo y máximo de politica de prioridad: %i-%i\n", minprio,maxprio);
   return 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? 12 ¿Se heredan los atributos de planificación? FIFO

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.

```
ps -u $USER -f
```

- 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.

```
ps -eo pid,gid,sid,s,command
```

- 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 es el SID del nuevo proceso. Comparten el GID (1000). Cuando se crea un nuevo proceso su identificador es 1000.

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

int main(int argc, char *argv[]) {
	pid_t pid = atoi(argv[1]);
	char *cmd = "cat";
	char *arguments[3];
	arguments[0] = "cat";
	arguments[1] = "/proc/sys/fs/file-max";
	arguments[2] = NULL;

	printf("Id. de proceso: %i\n", pid);
	printf("Id. de proceso padre: %i\n", getppid());
	printf("Id. de grupo de procesos: %i\n", getpgrp());
	printf("Id. de sesión: %i\n", getsid(pid));
	printf("Directorio de trabajo: %s\n", getenv("HOME"));
	printf("Nº de archivos que puede abrir el proceso: %i\n", execvp(cmd, arguments);
	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 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`.

¿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`)?

Si el padre termina antes que el hijo se queda huérfano y el ppid lo recoge la shell o init. Si ocurre al revés se queda esperando a que termine el padre.

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

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
	pid_t pid = fork();
	if(pid == -1) {
		perror("Error en el fork");
	} else if(pid == 0) {
		//sleep(10);
		int res = chdir("/tmp");
		if(res == -1) perror("error");
		pid_t pid2 = setsid();
		printf("Id. de proceso: %i\n", pid);
		printf("Id. de proceso padre: %i\n", getppid());
		printf("Id. de grupo de procesos: %i\n", getpgrp());
		printf("Id. de sesión: %i\n", getsid(pid));
		printf("Directorio de trabajo: %s\n", get_current_dir_name());
	} else if(pid > 0) {
		sleep(10);
		printf("Id. de proceso: %i\n", pid);
		printf("Id. de proceso padre: %i\n", getppid());
		printf("Id. de grupo de procesos: %i\n", getpgrp());
		printf("Id. de sesión: %i\n", getsid(pid));
		printf("Directorio de trabajo: %s\n", get_current_dir_name());
	}
	return 0;
}
```

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?

La cadena "El comando terminó de ejecutarse" sólo se ejecuta cuando se usa system. Esto se debe a que al ejecutar el comando exec sustituye la imagen del programa a la imagen del programa que hemos pasado por argumentos.

Ej7exec.c

```
#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;
 }
 if (execvp(argv[1], argv + 1) == -1) {
  printf("ERROR: No se ha ejecutado correctamente.\n");
 }

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


 return 0;
}
```

Ej7sys.c

```
#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;
 }

 int cmdLen = 1;
 int i;
 for (i = 1; i < argc; i++)
  cmdLen += strlen(argv[i]) + 1;
 char *cmd = malloc(sizeof(char)*cmdLen);
 strcpy(cmd, "");

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

 if (system(cmd) == -1) {
  printf("ERROR: No se ha ejecutado correctamente.\n");
 }
 printf("El comando terminó de ejecutarse.\n");


 return 0;
}
```

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

Cuando se pasa por parámetros de un programa ps -ef equivaldría a dos argumentos y si se quiere ejecutar como system sería necesario unirlos. Mientras que "ps -el" equivale a un único string 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*.

```
#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;
  case 0:;
   pid_t mi_sid = setsid(); //Creamos una nueva sesión
   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);

   if (execvp(argv[1], argv + 1) == -1) {
    printf("ERROR: No se ha ejecutado correctamente.\n");
    exit(-1);
   }
   //sleep(4);
   break;
  default:
    printf("[Padre] Proceso %i (Padre: %i)\n",getpid(),getppid());
     //sleep(4);
```

```
      break;
  }

  return 0;
}
```

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.

```
KILL(1)                         User Commands                         KILL(1)

NAME
     kill - terminate a process

SYNOPSIS
     kill [-s signal|-p] [-q sigval] [-a] [--] pid...
     kill -l [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:

          n     where n is larger than 0.  The process with pid n will be signaled.

          0     All processes in the current process group are signaled.

          -1    All processes with pid larger than 1 will be signaled.
  -n     where  n is larger than 1.  All processes in process group n 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, --queue sigval
Use sigqueue(2) rather than kill(2) and the sigval 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.

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?

sleep(600)&

-kill -INT<pid>

ps

-kill -CONT<pid>

ps

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).

```
#include <sys/types.h>
#include <string.h>
```

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

int main(int argc, char *argv[]) {
  if(argc != 2) {
       printf("Vuelve a ejecutar PID\n");
       exit(EXIT_FAILURE);
  }
  pid_t pid = atoi(argv[1]);
  sigset_t set, pendiente;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigaddset(&set, SIGTSTP);
  sigprocmask(SIG_BLOCK, &set, NULL);
  printf("Señales SIGINT y SIGTSTOP bloqueadas\n");
  sleep(10);
  sigpending(&pendiente);
    if(sigismember(&pendiente, SIGINT)) {
         printf("Se recibió la llamada SIGINT\n");
    }
    if(sigismember(&pendiente, SIGTSTP)) {
         printf("Se recibió la llamada SIGTSTOP, desbloqueando la llamada...\n");
         sigdelset(&pendiente, SIGINT);
         sigprocmask(SIG_UNBLOCK, &pendiente, NULL);
         printf("Proceso reanudado");
    }
    return 0;
}
```

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 <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <signal.h>

int sigint = 0;
int sigtstop = 0;

void handler(int signal) {
  if(signal == SIGINT) {
    sigint++;
  }
  if(signal == SIGTSTP) {
    sigtstop++;
  }
```

```
}
int main(int argc, char *argv[]) {
      if(argc != 2) {
            printf("Vuelve a ejecutar un PID\n");
            exit(EXIT_FAILURE);
      }
      pid_t pid = atoi(argv[1]);
      struct sigaction signal;
      signal.sa_handler = &handler;
      signal.sa_flags = 0;
      if(sigaction(SIGINT, &signal, NULL) == -1) perror("Error: SIGINT\n");
      if(sigaction(SIGTSTP, &signal, NULL) == -1) perror("Error:SIGTSTP\n");
      while(sigint < 10 && sigtstop < 10);
      printf("Se ha recibido la señal SIGINT %i veces\n", sigint);
      printf("Se ha recibido la señal SIGTSTP %i veces\n", sigtstop);
 return 0;
}
```

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

int main(int argc, char **argv) {
            if (argc != 2) {
               printf("ERROR: Introduce los segundos!\n");
               return -1;
             }

  sigset_t mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGUSR1);
  sigprocmask(SIG_UNBLOCK, &mask, NULL);

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

```
  int secs = atoi(argv[1]);

  int i = 0;
while (i < secs && stop == 0) {
   i++;
   sleep(1);
  }

  if (stop == 0) {
   printf("Se va a borrar");
   unlink(argv[0]);
  } else {
   printf("Has tenido suerte!");
  }

  return 0;
}
```