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 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.

```
sudo renice -10 9336
sudo renice -10 $$
sudo chrt -f -p 12 9336
sudo chrt -f -p 12 $$

$$: pid shell
```

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

```
#include <sys/resource.h>


int main(){

   pid_t pid = getpid();

   int policy = sched_getscheduler(pid);

   if(policy == -1){
           perror("Error ");
           exit(EXIT_FAILURE);
   }

   switch(policy){
           case SCHED_OTHER:
                   printf("Policy: SCHED_OTHER\n");
                   break;
           case SCHED_FIFO:
                   printf("Policy: SCHED_FIFO\n");
                   break;
           case SCHED_RR:
                   printf("Policy: SCHED_RR\n");
                   break;
   }

   int priority = getpriority(PRIO_PROCESS, 0);

   if(priority == -1){
           perror("Error ");
           exit(EXIT_FAILURE);
   }

   printf("Priority: %i\n", priority);

   int min = sched_get_priority_min(policy);

   if(min == -1){
           perror("Error ");
           exit(EXIT_FAILURE);
   }

   printf("Min priority: %i\n", min);

   int max = sched_get_priority_max(policy);

   if(max == -1){
           perror("Error ");
           exit(EXIT_FAILURE);
   }

   printf("Max priority: %i\n", max);

   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? ¿Se heredan los atributos de planificación?

Si heredan los atributos:
Policy: SCHED_FIFO
Priority: -10
Min priority: 1
Max priority: 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.

ps -aucursoredes -l
ps -u $USER -l
ps -u 1000 -l

- 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.

ps -eo pid,gid,session,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?

Todos comparten el identificador de grupo (GID), pero cada uno tiene su propio PID. El identificador de grupo depende de quien lo haya creado, si lo hace el usuario, tendrá el mismo GID.

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 ficheros que puede abrir el proceso y el directorio de trabajo actual.

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

```
#include <sys/time.h>
#include <sys/resource.h>

int main(){

   int pid = getpid();
   int ppid = getppid();
   int pgid = getpgid(0);

   if(pgid == -1) {
            perror("Error getpgid");
            exit(EXIT_FAILURE);
   }

   int sid = getsid(0);

   if(sid == -1) {
            perror("Error getsid");
            exit(EXIT_FAILURE);
   }

   printf("PID: %i\n", pid);
   printf("PPID: %i\n", ppid);
   printf("PGID: %i\n", pgid);
   printf("SID: %i\n", sid);

   struct rlimit rlimit;
   int limit = getrlimit(RLIMIT_NOFILE, &rlimit);

   if(limit == -1) {
            perror("Error getrlimit");
            exit(EXIT_FAILURE);
   }

   printf("Numero máximo ficheros: %i\n", rlimit.rlim_max);

   char directorio[512];
   getcwd(directorio, 512);

   if(directorio == NULL) {
            perror("Error getcwd");
            exit(EXIT_FAILURE);
   }

   printf("Directorio: %s\n", directorio);

   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`.

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

void imprime(char* proceso) {
   int pid = getpid();
   int ppid = getppid();
   int pgid = getpgid(0);

   if(pgid == -1) {
            perror("Error getpgid");
            exit(EXIT_FAILURE);
   }

   int sid = getsid(0);

   if(sid == -1) {
            perror("Error getsid");
            exit(EXIT_FAILURE);
   }

   printf("PID %s: %i\n", proceso, pid);
   printf("PPID %s: %i\n", proceso, ppid);
   printf("PGID %s: %i\n", proceso, pgid);
   printf("SID %s: %i\n", proceso, sid);

   struct rlimit rlimit;
   int limit = getrlimit(RLIMIT_NOFILE, &rlimit);

   if(limit == -1) {
            perror("Error getrlimit");
            exit(EXIT_FAILURE);
   }

   printf("Numero máximo ficheros %s: %i\n", proceso, rlimit.rlim_max);

   char directorio[512];
   getcwd(directorio, 512);

   if(directorio == NULL) {
            perror("Error getcwd");
            exit(EXIT_FAILURE);
   }

   printf("Directorio %s: %s\n", proceso, directorio);

}

int main(){
```

```
   pid_t pid = fork();

   if(pid == -1) {
            perror("Error en el fork");
            exit(EXIT_FAILURE);
   }

   //Proceso hijo
   else if(pid == 0) {
            setsid();
            chdir("/tmp");
            imprime("hijo");
   }

   else {
            imprime("padre");
   }

   return 0;
}
```

¿Qué sucede si el proceso padre termina antes que el hijo (observar el PPID del proceso hijo)?

El PPID del hijo pasa a ser el PID del proceso raíz.

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

El PPID del hijo es el PID del padre.

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?

system

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

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

   if(argc != 2) {
                  fprintf(stderr, "Usage: %s <comando con comillas>\n", argv[0]);
                  exit(EXIT_FAILURE);
```

```
        }

    int tam = strlen(argv[1]) + 1;

    char *command = malloc(sizeof(char)*tam);;

    command = argv[1];
    command[tam - 1] = '\0';


    if(system(command) == -1) {
        fprintf(stderr, "Error comando system\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

execvp

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

int main(int argc, char** argv) { //char* argv[] <- otra forma

    if(argc < 2) {
        fprintf(stderr, "Usage: %s <command>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if(execvp(argv[1], argv + 1) == -1) { // &argv[1] <- segundo arg otra forma
        perror("Error execvp");
        exit(EXIT_FAILURE);
    }

    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”`?

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

void cambio_descriptores(){
   int fdout = open("/tmp/daemon.out", O_CREAT | O_RDWR, 00777);
   if(fdout == -1) {
            perror("Error fichero daemon.out");
            exit(EXIT_FAILURE);
   }

   int fderr = open("/tmp/daemon.err", O_CREAT | O_RDWR, 00777);
   if(fderr == -1) {
            perror("Error fichero daemon.err");
            exit(EXIT_FAILURE);
   }

   int fdin = open("/dev/null", O_CREAT | O_RDWR, 00777);
   if(fdin == -1) {
            perror("Error fichero null");
            exit(EXIT_FAILURE);
   }

   if(dup2(fdout, 1) == -1) {
            perror("Error dup2 fdout");
            exit(EXIT_FAILURE);
   }

   if(dup2(fderr, 2) == -1) {
            perror("Error dup2 fdout");
            exit(EXIT_FAILURE);
   }

   if(dup2(fdin, 0) == -1) {
            perror("Error dup2 fdout");
            exit(EXIT_FAILURE);
   }
}

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

   if(argc < 2) {
            fprintf(stderr, "Usage: %s <command>\n", argv[0]);
            exit(EXIT_FAILURE);
   }

   pid_t pid = fork();

   if(pid == -1) {
            perror("Error en el fork");
```

```
            exit(EXIT_FAILURE);
            }

    //Proceso hijo
    else if(pid == 0) {
            setsid();
            cambio_descriptores();
            if(execvp(argv[1], &argv[1]) == -1) {
                    perror("Error execvp\n");
                    exit(EXIT_FAILURE);
            }
    }

    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?

kill -l: lista todas las señales

kill -9 PID: Termina con el proceso con pid PID. El proceso termina con el código de salida Killed.

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).

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.

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.