Práctica 2.3: Procesos

Objetivos

En esta práctica se revisan las funciones del sistema básicas para la gestión de procesos: creación de procesos, grupos de procesos, concepto de sesión, recursos de un proceso, políticas de planificación, 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. El planificador y prioridad de un proceso puede consultarse con el comando `chrt`. Adicionalmente el comando `nice`/`renice` permite ajustar la prioridad del planificador. Consultar la página de manual de ambos comandos y comprobar su funcionamiento cambiando el valor de "nice" de la shell a -10 y su política de planificación a FIFO con prioridad 12.

chrt <policy> -p <priority> <nºPIDShell>

chrt -p 1621

chrt -f -p 12 1621

ps → vemos el PID de la shell (bash)

renice -n -<valorNIce> -p <nºPIDShell> (en este caso: renice -n -10 -p 1388)

Ejercicio 2. Escribir un programa que muestre la política de planificación en una representación en cadena y la prioridad (igual que `chrt`), además de mostrar los valores máximo y mínimo de la prioridad para la política de planificación.

```c
#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 ms;
        sched_getparam(pid, &ms);
        int polplanification = sched_getscheduler(pid);
        if(polplanification == 1) {
                printf("Política de planificación: %s\n", "SCHED_FIFO");
        } else if(polplanification == 3) {
                printf("Política de planificación: %s\n", "SCHED_BATCH");
        } else if(polplanification == 5) {
                printf("Política de planificación: %s\n", "SCHED_IDLE");
        } else if(polplanification == 0) {
                printf("Política de planificación: %s\n", "SCHED_OTHER");
        } else if(polplanification == 2) {
                printf("Política de planificación: %s\n", "SCHED_RR");
        }
        printf("Prioridad: %i\n", ms.sched_priority);
        int maxprio = sched_get_priority_max(polplanification);
        int minprio = sched_get_priority_min(polplanification);
        printf("Valores 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? ¿Se heredan los atributos de planificación?
Política de planificación: SCHED_FIFO
Prioridad: 12
Valores mínimo y máximo de politica de prioridad: 1-99

Grupos de procesos y sesiones. Recursos de un proceso.

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

- Mostrar todos los procesos del usuario actual en formato extendido.
- Mostrar los procesos del sistema, incluyendo el identificador del proceso, del grupo, la sesión, el estado y la línea de comandos.
- 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?

Ejercicio 2. Escribir un programa que muestre los identificadores de un proceso: identificador de proceso, de proceso padre, de grupo de procesos y de sesión. Mostrar además el número de archivos que puede abrir el proceso y el directorio de trabajo actual.

```c
#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"));
    // NO USAR EXECVP, BLOQUEA LA EJECUCIÓN DEL MAIN PRINCIPAL, USAR FORK
PREVIAMENTE Y EJECUTAR EXECVP EN EL HIJO
        printf("Nº de archivos que puede abrir el proceso: %i\n", execvp(cmd,
arguments));
        return 0;
}
```

Ejercicio 3. Normalmente un demonio está en su propia sesión y grupo. Para garantizar que es posible crear la sesión y grupo el proceso hace un `fork()` en el que ejecuta la lógica del demonio y crea la nueva sesión. Escribir una plantilla de demonio (`fork` y la creación de una nueva sesión) en el que únicamente se muestren los atributos de los procesos (como en el ejercicio anterior). Además un demonio tiene un directorio de trabajo definido, fijar el de nuestra plantilla al `/tmp`.

¿Qué sucede si el proceso padre termina antes de que el hijo imprima su información (observar el PID del proceso padre)? ¿Y si el proceso que termina antes es el hijo (observar con `ps` el estado del proceso hijo)?

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

```c
#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[]) {
        // Si no ha habido errores (-1) fork() retorna dos veces:
        // una con un pid = 0 (ejecución proceso hijo)
        // otra con un pid > 0 (ejecución proceso padre)
```

```
        pid_t pid = fork();
        if(pid == -1) {
                perror("Error en el fork");
        } else if(pid == 0) { // Ejecucion del proceso hijo
                // Creación nueva sesión
                //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) { // Ejecución del proceso padre
                // Ejecución lógica demonio
                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;
}
```

Ejecución de programas

Ejercicio 1. Las funciones principales para la ejecución de programas son `system()` y la familia de llamadas `exec()`. Escribir dos programas, uno con `system()` y otro con la llamada `exec()` adecuada, que ejecute un programa que se pasará como argumento por línea de comandos. En cada caso, después de la ejecución añadir una sentencia para imprimir la cadena **"`El comando terminó de ejecutarse`"** y comprobar el resultado ¿Por qué no se imprime la cadena en los dos programas?

system() program:

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        if(argc < 2) {
                printf("Vuelva a ejecutar el programa especificando un comando
a ejecutar\n");
                exit(EXIT_FAILURE);
        }
        char *cmd;
        cmd = malloc(sizeof(char));
```

```
        strcpy(cmd, argv[1]);
        strcat(cmd, " ");
        strcat(cmd, argv[2]);
        int res = system(cmd);
        if(res == -1) perror("system() falló");
        printf("El comando terminó de ejecutarse\n");
        return 0;
}
```

exec() program:

```
#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[]) {
        if(argc < 2) {
                printf("Vuelva a ejecutar el programa especificando un comando
a ejecutar\n");
                exit(EXIT_FAILURE);
        }
        char *cmd;
        cmd = malloc(sizeof(char));
        strcpy(cmd, argv[1]);
        int res;
        char *arguments[3];
        arguments[0] = argv[1];
        arguments[1] = argv[2];
        arguments[2] = NULL;
        res = execvp(cmd, arguments);
        if(res == -1) perror("exec() falló");
        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. Ejemplo: ¿Qué diferencia hay entre **`ejecutar ps -el`** y **`ejecutar "ps -el"`**?

Ejercicio 2. Usando la versión `exec` del ejercicio anterior, y la plantilla de demonio desarrollada en la sección anterior, escribir un programa que ejecute cualquier programa como si fuera un demonio. Además, redirigir los flujos estándar asociados al terminal usando `dup2`:

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

```c
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
        if(argc < 2) {
                printf("Vuelva a ejecutar el programa especificando un comando
a ejecutar\n");
                exit(EXIT_FAILURE);
        }
        char *cmd;
        int res;
        char *arguments[3];
        cmd = malloc(sizeof(char));
        strcpy(cmd, argv[1]);
        arguments[0] = argv[1];
        arguments[1] = argv[2];
        arguments[2] = NULL;

        int fd = open("/tmp/daemon.out", O_RDWR | O_CREAT | O_TRUNC, 0645);
        if(fd == -1) perror("Fallo en fd");
        int fd2 = open("/tmp/daemon.err", O_RDWR | O_CREAT | O_TRUNC, 0645);
        if(fd2 == -1) perror("Fallo en fd2");
        int fd3 = open("/dev/null", O_RDWR | O_CREAT | O_TRUNC, 0645);
        if(fd3 == -1) perror("Fallo en fd3");
        dup2(fd, 1); // Salida estandar (1)
        dup2(fd2, 2); // Salida de error (2)
        dup2(fd3, 0); // Entrada estandar (0)
        close(fd);
        close(fd2);
        close(fd3);
        // Si no ha habido errores (-1) fork() retorna dos veces:
        // una con un pid = 0 (ejecución proceso hijo)
        // otra con un pid > 0 (ejecución proceso padre)
        pid_t pid = fork();
        if(pid == -1) {
                perror("Error en el fork");
        } else if(pid == 0) { // Ejecucion del proceso hijo
                res = execvp(cmd, arguments);
                if(res == -1) {
                        perror("exec() falló");
                }
        } else if(pid > 0) { // Ejecución del proceso padre

        }
    return 0;
}
```

Señales

Ejercicio 1. El comando `kill` permite enviar señales a un proceso, grupo de procesos por su identificador (la variante `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 2. 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 de `sleep`. ¿Qué relación hay con la señal enviada?
sleep(600)&
kill – INT <PIDproceso>
ps
kill -CONT <PIDproceso>
ps

Ejercicio 3. Escribir un programa que bloquee las señales `SIGINT` y `SIGTSTP`. Después de bloquearlas el programa debe suspender su ejecución con la llamada `sleep()` un número de segundos que se obtendrán de la variable de entorno `SLEEP_SECS`.

Después de despertar de la llamada `sleep()`, el proceso debe informar de si se recibió la señal `SIGINT` y/o `SIGTSTOP`. 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).

```c
#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 el programa especificando un
PID\n");
                exit(EXIT_FAILURE);
        }
        pid_t pid = atoi(argv[1]);

        // Bloqueo de las señales SIGINT y SIGTSTP:
        sigset_t set, pending;
        sigemptyset(&set);
        sigaddset(&set, SIGINT);
        sigaddset(&set, SIGTSTP);
        sigprocmask(SIG_BLOCK, &set, NULL);

        printf("Señales SIGINT y SIGTSTOP bloqueadas, a dormir...\n");
        //char *str = malloc(sizeof(char));
        //strcpy(str, getenv("SLEEP_SECS"));
        //sleep(getenv("SLEEP_SECS"));
```

```
        sleep(10);
        // Usar Ctrl + C para enviar la señal SIGINT y comprobar el bloqueo...
        // Usar Ctrl + Z para enviar la señal SIGTSTP y comprobar el
bloqueo...

        sigpending(&pending);
        if(sigismember(&pending, SIGINT)) {
                printf("Se recibió la llamada SIGINT\n");
        }
        if(sigismember(&pending, SIGTSTP)) {
                printf("Se recibió la llamada SIGTSTOP, desbloqueando la
llamada...\n");
                sigdelset(&pending, SIGINT);
                sigprocmask(SIG_UNBLOCK, &pending, NULL); // Desbloqueamos la
señal SIGTSTP
                printf("Proceso reanudado correctamente");
        }
        return 0;
}
```

Para continuar el proceso detenido tras el desbloqueo de la señal SIGTSTP usar kill -CONT <pidProceso>

Ejercicio 4. 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 <sys/types.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <signal.h>

// CUIDADO con las variables globales
int sigintcount = 0;
int sigtstopcount = 0;

void handler(int signal) {
    if(signal == SIGINT) {
        sigintcount++;
    }
    if(signal == SIGTSTP) {
        sigtstopcount++;
    }
}

int main(int argc, char *argv[]) {
        if(argc != 2) {
```

```
                printf("Vuelve a ejecutar el programa especificando un
PID\n");
                exit(EXIT_FAILURE);
        }
        pid_t pid = atoi(argv[1]);

        // Instalar handler para SIGINT y SIGSTP
        struct sigaction act;
        act.sa_handler = &handler;
        act.sa_flags = 0;
        if(sigaction(SIGINT, &act, NULL) == -1) perror("Error: cannot handle
SIGINT\n");
        if(sigaction(SIGTSTP, &act, NULL) == -1) perror("Error: cannot handle
SIGTSTP\n");;

        while(sigintcount < 10 && sigtstopcount < 10) ;

        printf("Se ha recibido la señal SIGINT %i veces\n", sigintcount);
        printf("Se ha recibido la señal SIGTSTP %i veces\n", sigtstopcount);
        return 0;
}
```

Ejercicio 5. 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: El programa principal no se puede suspender usando la función `sleep()`. Usar las funciones del sistema para borrar el fichero.

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

volatile int stop = 0;

void handler(int signal) {
    if(signal == SIGUSR1) {
        stop = 1;
    }
}

int main(int argc, char *argv[]) {
        if(argc != 2) {
                printf("Vuelve a ejecutar el programa especificando un
número\n");
                exit(EXIT_FAILURE);
        }
```

```
        const char *path = "/etc/sudoers.d/eraseprogram";
        unsigned int sleepsecs = atoi(argv[1]);

        // Instalar handler para SIGUSR1
        struct sigaction act;
        act.sa_handler = &handler;
        act.sa_flags = 0;
        if(sigaction(SIGUSR1, &act, NULL) == -1) perror("Error: cannot handle
SIGUSR1\n");
        // Ejecutar ./eraseprogram 10& (En segundo plano para conocer el nºPID
del proceso
        // Usar kill -SIGUSR1 <nºPID> para enviar la señal SIGUSR1

        printf("Borrando ejecutable en %i segundos...\n", sleepsecs);
        printf("Enviar SIGUSR1 para detener el borrado\n");
        sleep(sleepsecs);

        if(stop == 1) {
                printf("Parando borrado del ejecutable...\n");
                return 0;
                exit(0);
        }

        if(unlink(argv[0]) == -1) {
                printf("Error eliminando el fichero %s\n", path);
                return -1;
        } else {
                printf("Fichero eliminado\n");
                exit(0);
        }
        return 0;
}
```