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.

chrt - manipulate the real-time attributes of a process

chrt [options] prio command [arg]...
chrt [options] -p [prio] 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.

nice - run a program with modified scheduling priority

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

Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process).

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

renice - alter priority of running processes

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

-n, --priority priority

```
sudo renice -n -10 $$
sudo chrt -f -p 12 $$
sudo chrt -f -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.

```
sched_setscheduler, sched_getscheduler - set and get scheduling policy/parameters
sched_setparam, sched_getparam - set and get scheduling parameters
sched_get_priority_max, sched_get_priority_min  - get static priority range

#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
int sched_getscheduler(pid_t pid);
int sched_setparam(pid_t pid, const struct sched_param *param);
int sched_getparam(pid_t pid, struct sched_param *param);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

struct sched_param {
        ...
        int sched_priority;
        ...
};


#include <sched.h>
#include <stdio.h>
#include <errno.h>

int main() {
   int sched = sched_getscheduler(0);

   printf("Politica de planificacion: ");
   if (sched == SCHED_OTHER)
      printf("SCHED_OTHER\n");
   else if (sched == SCHED_FIFO)
      printf("SCHED_FIFO\n");
   else
      printf("SCHED_RR\n");
```

```
    struct sched_param params;
    int priority = sched_getparam(0, &params);
    if (priority == -1) {
        printf("Error: %d %s\n", errno, strerror(errno));
        return 1;
    }
    printf("Prioridad: %d\n", params.sched_priority);

    int max = sched_get_priority_max(sched);
    int min = sched_get_priority_min(sched);
    printf("Valor maximo: %d Valor minimo: %d\n", max, min);

    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?

```
[cursoredes@localhost ~]$ sudo nice -n 12 /bin/sh
sh-4.2# ps
  PID TTY          TIME CMD
 4634 pts/0    00:00:00 sudo
 4635 pts/0    00:00:00 sh
 4636 pts/0    00:00:00 ps
sh-4.2# sudo chrt -f -p 12 4635
sh-4.2# ./ej2
Politica de planificacion: SCHED_FIFO
Prioridad: 12
Valor maximo: 99 Valor minimo: 1

La prioridad es 12 al igual que la del shell y se heredan los atributos de planificación
```

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?

```
ps - report a snapshot of the current processes.

ps [options]

ps -f -u cursoredes
ps -e -o pid,pgid,sid,s,cmd
```

ps -o pid,pgid,sid,s,cmd: Comparten el SID y el PID y PGID también son el mismo valor. Cuando se crea un nuevo proceso su PGID es el mismo valor que su PID.

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.

```
getpid, getppid - get process identification
getgid, getegid - get group identity
getsid - get session ID
setpgid, getpgid, setpgrp, getpgrp - set/get process group
getcwd, getwd, get_current_dir_name - get current working directory

#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
pid_t getsid(pid_t pid);
char *getcwd(char *buf, size_t size);
char *getwd(char *buf);
char *get_current_dir_name(void);

getrlimit, setrlimit, prlimit - get/set resource limits

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

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
int prlimit(pid_t pid, int resource, const struct rlimit *new_limit, struct rlimit *old_limit);

struct rlimit {
   rlim_t rlim_cur;  /* Soft limit */
   rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};


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

int main() {
   pid_t pid = getpid();
   printf("PID: %d\n", pid);

   pid_t ppid = getppid();
   printf("PPID: %d\n", ppid);
```

```
    pid_t pgid = getpgid(pid);
    if (pgid == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }
    printf("PGID: %d\n", pgid);

    pid_t sid = getsid(pid);
    if (sid == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }
    printf("SID: %d\n", sid);

    struct rlimit rlim;
    int lim = getrlimit(RLIMIT_NOFILE, &rlim);
    if (lim == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }
    printf("Max files: %d\n", rlim.rlim_max);

    char *path = malloc(sizeof(char)*(4096 + 1)); //Max length of a path in the system + \o
    getcwd(path, 4096 + 1);
    printf("CWD: %s\n", path);
    free(path);

    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`)?

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

```
fork - create a child process
chdir, fchdir - change working directory
sleep - sleep for the specified number of seconds

#include <unistd.h>

pid_t fork(void);
int chdir(const char *path);
int fchdir(int fd);
unsigned int sleep(unsigned int seconds);

wait, waitpid, waitid - wait for process to change state
```

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);


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

int processAttribs() {
   pid_t pid = getpid();
   printf("PID: %d\n", pid);

   pid_t ppid = getppid();
   printf("PPID: %d\n", ppid);

   pid_t pgid = getpgid(pid);
   if (pgid == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   printf("PGID: %d\n", pgid);

   pid_t sid = getsid(pid);
   if (sid == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   printf("SID: %d\n", sid);

   struct rlimit rlim;
   int lim = getrlimit(RLIMIT_NOFILE, &rlim);
   if (lim == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   printf("Max files: %d\n", rlim.rlim_max);

   char *path = malloc(sizeof(char)*(4096 + 1));
   getcwd(path, 4096 + 1);
   printf("CWD: %s\n", path);
   free(path);
}

int main() {
   pid_t pid = fork();
   if (pid == 0) {
      printf("Soy el hijo\n");
```

```
    pid_t sid = setsid();
    if (sid == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
    }

    int dir = chdir("/tmp");
    if (dir == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
    }

    processAttribs();
    exit(0);
  }
  else if (pid > 0) {
    printf("Soy el padre\n");
    processAttribs();

    int status;
    wait(&status);
    if (status == -1) {
      printf("Error in child");
      return 1;
    }
  }
  else if (pid == -1) {
    printf("Error: %d %s", errno, strerror(errno));
    return 1;
  }

  return 0;
}
```

Lo que pasa es que el proceso hijo se queda huérfano y es adoptado por el proceso init (PPID 1). En caso contrario el proceso hijo se convierte en zombie ya que ha terminado su ejecución pero no se han liberado sus recursos.

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 -el` y `./ej7 "ps -el"`?

```
execl, execlp, execle, execv, execvp, execvpe - execute a file

#include <unistd.h>

extern char **environ;
```

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);

system - execute a shell command

#include <stdlib.h>

int system(const char *command);


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

int main(int argc, char *argv[]) {
   if (argc < 2) {
      printf("Incorrect number of arguments\n");
      return 1;
   }

   int length = 0, i = 0;
   for (i = 1; i < argc; i++)
      length += strlen(argv[i]) + 1;
   char *command = malloc(sizeof(char)*length + 1);

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

   int x = system(command);
   if (x == -1) {
      printf("Error executing system\n");
      return 1;
   }

   printf("El comando termino de ejecutarse\n");
   return 0;
}


#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
   if (argc < 2) {
      printf("Incorrect number of arguments\n");
      return 1;
   }

   int x = execvp(argv[1], argv + 1);
   if (x == -1) {
```

```
    printf("Error executing execvp\n");
    return 1;
  }

  printf("El comando termino de ejecutarse\n");
  return 0;
}
```

Solo se imprime la cadena en el caso de system ya que lo que provoca execvp es que el programa que estaba corriendo se sustituya por el nuevo comando ejecutado. En el caso de system es mejor usar las comillas para que se interprete el comando entero como un único char *. En el caso de execvp por los parámetros que recibe es mejor que no contenga las comillas y se interprete como distintos argumentos.

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

int main(int argc, char *argv[]) {
  if (argc < 2) {
                printf("Incorrect number of arguments\n");
                return 1;
        }

  pid_t pid = fork();
  if (pid == 0) {
    // Hijo
    pid_t sid = setsid();
    if (sid == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      exit(EXIT_FAILURE);
    }

    int fd_out = open("/tmp/daemon.out", O_CREAT | O_RDWR, 0777);
    if(fd_out == -1){
      printf("Error creating the file\n");
      close(fd_out);
      exit(EXIT_FAILURE);
    }
    dup2(fd_out, STDOUT_FILENO);

    int fd_err = open("/tmp/daemon.err", O_CREAT | O_RDWR, 0777);
    if(fd_err == -1){
```

```
            perror("Error creating the file\n");
            close(fd_out);
            close(fd_err);
            exit(EXIT_FAILURE);
        }
        dup2(fd_err, STDERR_FILENO);

        int fd_in = open("/dev/null", O_CREAT | O_RDWR, 0777);
        if(fd_in == -1){
            printf("Error creating the file\n");
            close(fd_out);
            close(fd_err);
            close(fd_in);
            exit(EXIT_FAILURE);
        }
        dup2(fd_in, STDIN_FILENO);

        int x = execvp(argv[1], argv + 1);
        if (x == -1) {
            printf("Error executing execvp\n");
            exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
    }
    /*else if (pid > 0) {
        // Padre
        int status;
        wait(&status);
        if (status == -1) {
            printf("Error in child");
            return 1;
        }
    }*/
    else if (pid == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }

    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.

```
kill - terminate a process

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

-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.

0 - ?

1 - SIGHUP - ?, controlling terminal closed,

2 - SIGINT - interupt process stream, ctrl-C

3 - SIGQUIT - like ctrl-C but with a core dump, interuption by error in code, ctl-/

4 - SIGILL

5 - SIGTRAP

6 - SIGABRT

7 - SIGBUS

8 - SIGFPE

9 - SIGKILL - terminate immediately/hard kill, use when 15 doesn't work or when something disasterous might happen if process is allowed to cont., kill -9

10 - SIGUSR1

11 - SIGEGV

12 - SIGUSR2

13 - SIGPIPE

14 - SIGALRM

15 - SIGTERM - terminate whenever/soft kill, typically sends SIGHUP as well?

16 - SIGSTKFLT

17 - SIGCHLD

18 - SIGCONT - Resume process, ctrl-Z (2nd)

19 - SIGSTOP - Pause the process / free command line, ctrl-Z (1st)

20 - SIGTSTP

21 - SIGTTIN

22 - SIGTTOU

23 - SIGURG

24 - SIGXCPU

25 - SIGXFSZ

26 - SIGVTALRM

27 - SIGPROF

28 - SIGWINCH

29 - SIGIO

29 - SIGPOLL

30 - SIGPWR - shutdown, typically from unusual hardware failure

31 - SIGSYS

pgrep, pkill - look up or signal processes based on name and other attributes

```
pgrep [options] pattern
pkill [options] pattern
```

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?

```
[cursoredes@localhost ~]$ kill 2404
[cursoredes@localhost ~]$ sleep 600
Terminated
[cursoredes@localhost ~]$ kill -s SIGINT 2758
[cursoredes@localhost ~]$ sleep 600

[cursoredes@localhost ~]$ kill -s 6 2616 (SIGABRT)
[cursoredes@localhost ~]$ sleep 600
Aborted (core dumped)
[cursoredes@localhost ~]$ kill -s SIGHUP 2730
[cursoredes@localhost ~]$ sleep 600
Hangup
[cursoredes@localhost ~]$ kill -s SIGTSTP 2794
[cursoredes@localhost ~]$ sleep 600

[1]+  Stopped                 sleep 600
```

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).

```
sigprocmask - examine and change blocked signals
sigemptyset,  sigfillset, sigaddset, sigdelset, sigismember

#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);

Requiere: SLEEP_SECS=”30” y export SLEEP_SECS en consola


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

int main() {
   sigset_t set;
   sigemptyset(&set);
   sigaddset(&set, SIGINT);
```

```
    sigaddset(&set, SIGTSTP);

    int v = sigprocmask(SIG_BLOCK, &set, NULL);
    if (v == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }

    printf("Sleeping\n");
    sleep(atoi(getenv("SLEEP_SECS")));
    printf("Waking up\n");

    sigset_t set_p;
    sigpending(&set_p);

    v = sigismember(&set_p, SIGINT);
    if (v == 1) {
        printf("SIGINT received\n");
        sigdelset(&set_p, SIGINT);
    }
    else {
        printf("SIGINT not received\n");
    }

    v = sigismember(&set_p, SIGTSTP);
    if (v == 1) {
        printf("SIGTSTP received\n");
        sigdelset(&set_p, SIGTSTP);
    }
    else {
        printf("SIGTSTP not received\n");
    }

    v = sigprocmask(SIG_UNBLOCK, &set, NULL);
    if (v == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }

    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.

```
sigaction - examine and change a signal action
sigsuspend - wait for a signal

#include <signal.h>

int sigsuspend(const sigset_t *mask);
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

struct sigaction {
```

```
        void    (*sa_handler)(int);
        void    (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int     sa_flags;
        void    (*sa_restorer)(void);
};


#include <signal.h>
#include <errno.h>
#include <stdio.h>

volatile int cont_int = 0;
volatile int cont_tstp = 0;

void handler(int signal) {
   if (signal == SIGINT)
      cont_int++;
   if (signal == SIGTSTP)
      cont_tstp++;
}

int main() {
   struct sigaction handle;

   int v = sigaction(SIGINT, NULL, &handle);
   if (v == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   handle.sa_handler = handler;
   v = sigaction(SIGINT, &handle, NULL);
   if (v == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }

   v = sigaction(SIGTSTP, NULL, &handle);
   if (v == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }
   handle.sa_handler = handler;
   v = sigaction(SIGTSTP, &handle, NULL);
   if (v == -1) {
      printf("Error: %d %s", errno, strerror(errno));
      return 1;
   }

   sigset_t set;
        sigemptyset(&set);

        while ((cont_int + cont_tstp) < 10)
                sigsuspend(&set);

        printf("Num SIGINT: %d\n", cont_int);
        printf("Num SIGTSTP: %d\n", cont_tstp);
```

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

volatile int stop = 0;

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

int main(int argc, char *argv[]) {
    if(argc != 2){
                        printf("Incorrect number of arguments\n");
                        return 1;
            }

    sigset_t set;
            sigemptyset(&set);
    sigaddset(&set, SIGUSR1);

    int v = sigprocmask(SIG_UNBLOCK, &set, NULL);
    if (v == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }

    struct sigaction handle;

    v = sigaction(SIGUSR1, NULL, &handle);
    if (v == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }
    handle.sa_handler = handler;
    v = sigaction(SIGUSR1, &handle, NULL);
    if (v == -1) {
        printf("Error: %d %s", errno, strerror(errno));
        return 1;
    }

    int secs = atoi(argv[1]);

    printf("Sleeping\n");
```

```
   sleep(secs);
   printf("Waking up\n");

   if (stop == 1) {
      printf("File saved\n");
   }
   else {
      printf("The file will be deleted\n");
      v = unlink(argv[0]);
      if (v == -1) {
         printf("Error: %d %s", errno, strerror(errno));
         return 1;
      }
      printf("I deleted myself! :(\n");
   }

   return 0;
}
```