Llamadas al sistema

Sistemas Operativos (SO)
Grado en Ingeniería Informática

Universidad de León

Resumen

El objetivo principal de esta práctica es entender y aprender a manejar algunas de las llamadas al sistema más importantes. En concreto, trabajaremos con llamadas al sistema de las siguientes categorías: llamadas relacionadas con procesos, llamadas relacionadas con señales, llamadas relacionadas con ficheros y llamadas para comunicar procesos.

1. Llamadas relacionadas con procesos

En este apartado trabajaremos con llamadas relacionadas con procesos: *fork*, *getpid*, *getppid*, *sleep*, *wait*, *waitpid*, *exit* y *exec*. En la Sección 1.5 se proponen una serie de ejercicios para trabajar estas llamadas.

1.1. *fork*, *getpid* y *getppid*

- ***fork*** crea un nuevo proceso al duplicar el proceso que lo llama (proceso padre). El nuevo proceso se denomina proceso hijo. En el momento de fork() ambos espacios de memoria tienen el mismo contenido. El proceso hijo y el proceso padre se ejecutan en espacios de memoria separados y siguen caminos independientes. Si la llamada a fork() finaliza con éxito, el PID del proceso hijo se devuelve en el padre y se devolverá 0 en el hijo. En caso de fallo, no se crea ningún proceso hijo, y se devuelve -1 en el padre.
- ***getpid*** devuelve el identificador del proceso que realiza la llamada.
- ***getppid*** devuelve el identificador del proceso padre del proceso que realiza la llamada.

El programa `fork.c` muestra como utilizar las llamadas *fork*, *getpid* y *getppid*:

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

int main(void) {
  int pid = fork();
  if (pid==0)
    printf("[H] ppid = %5d, pid = %5d\n", getppid(), getpid());
  else
    printf("[P] ppid = %5d, pid = %5d, H = %5d\n", getppid(), getpid(), pid);

  return 0;
}
```

Para compilar el programa `fork.c` se utiliza el siguiente comando:

```
$ cc fork.c -o fork
```

Para ejecutar el programa `fork`, creado con el comando anterior, ejecuta el siguiente comando:

```
$ ./fork
```

Ejecuta el programa varias veces, observa como cambian los PIDs y fíjate en el orden de ejecución de los procesos. Su prototipo es el siguiente:

1.2. *wait* y *waitpid*

wait suspende la ejecución del proceso que la utiliza hasta que alguno de sus procesos hijo cambia de estado. Su prototipo es el siguiente:

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

pid_t wait(int *wstatus);
```

wait devuelve el pid del proceso que ha cambiado de estado. Recibe como argumento la dirección de una variable de tipo entero donde se almacenará el código de estado del proceso hijo que ha cambiado.

waitpid es similar a *wait*, pero permite esperar por un proceso concreto. Su prototipo es el siguiente:

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

`pid` puede tomar los siguientes valores:

`<-1` Se espera por cualquier proceso hijo cuyo ID de grupo sea igual al valor absoluto de `pid`.

`-1` Se espera por cualquier proceso hijo.

`0` Se espera por cualquier proceso hijo cuyo ID de grupo es igual al del proceso que realiza la llamada *waitpid*.

`>0` Se espera por el proceso hijo cuyo PID es igual al valor de `pid`.

1.3. *exit*

exit termina la ejecución del proceso que la invoque. su prototipo es el siguiente:

```
#include <stdlib.h>

void exit(int status);
```

`status` es el código que se devolverá al proceso padre del proceso que ejecute la llamada *exit*.

Cuando un proceso utiliza *wait* recibe el estado de terminación del hijo, que tiene que interpretarse como muestra la figura 1.

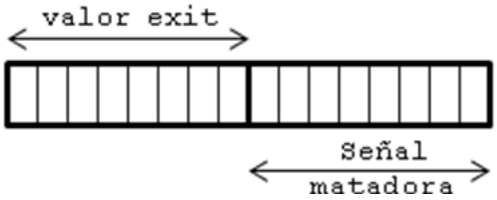

Figura 1: Interpretación del estado en la llamada *wait*.

La macro `WEXITSTATUS()`, definida en `/usr/include/sys/wait.h`, permite acceder al primer octeto. Para acceder al segundo octeto podemos utilizar la siguiente operación binaria: `status & 0xff`. Podemos saber cuándo un proceso ha terminado porque ha realizado una llamada *exit* utilizando la macro `WIFEXITED()`. `WIFEXITED()` devuelve 0, si el hijo ha terminado de una manera anormal (terminado por una señal SIGKILL, etc.), y distinto de 0 si ha terminado porque ha realizado una llamada *exit*.

1.4. *exec*

Las funciones de la familia *exec* sustituyen la imagen del proceso actual por una nueva imagen. La nueva imagen se construye a partir de un fichero regular ejecutable. No hay valor de retorno en caso de que *exec* tenga éxito, porque la nueva imagen sustituye a la anterior.

Los prototipos de las funciones de la familia *exec* son los siguientes:

```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

A continuación se muestran algunos ejemplos de como utilizar las funciones de la familia *exec*:

```
#include <unistd.h>

/* The following example executes the ls command, specifying the pathname of the executable (/bin/ls
    ) and using arguments supplied directly to the command to produce single-column output. */
int ret;
ret = execl ("/bin/ls", "ls", "-l", (char *)0);
```

```c
/* The following example is similar to Using execl(). In addition, it specifies the environment for
    the new process image using the env argument. */
int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);

/* The following example searches for the location of the ls command among the directories specified
     by the PATH environment variable. */
int ret;
ret = execlp ("ls", "ls", "-l", (char *)0);

/* The following example passes arguments to the ls command in the cmd array. */
char *cmd[] = { "ls", "-l", (char *)0 };
int ret;
ret = execv ("/bin/ls", cmd);

/* The following example passes arguments to the ls command in the cmd array, and specifies the
    environment for the new process image using the env argument. */
int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
ret = execve ("/bin/ls", cmd, env);

/* The following example searches for the location of the ls command among the directories specified
     by the PATH environment variable, and passes arguments to the ls command in the cmd array. */
int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
ret = execvp ("ls", cmd);
```

1.5. Ejercicios

Ejercicio 1. *Modifica el programa `fork.c` visto en la Sección 1.1 para que el proceso padre, después de escribir su traza, espere a que termine el proceso hijo y escriba una segunda traza similar a la siguiente:*

```
[P] el proceso pid=PID acaba de terminar con esado STATUS
```

Donde `PID` es el identificador del proceso que termina y `STATUS` si código de estado. Para hacerlo, utiliza la llamada wait.

Solución:

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

int main(void) {
  int pid = fork();
  if (pid==0){
    sleep(1);
    printf("[H] ppid = %5d, pid = %5d\n", getppid(), getpid());
  } else {
    int p, status;
    printf("[P] ppid = %5d, pid = %5d, H = %5d\n", getppid(), getpid(), pid);
    p = wait(&status);
    printf("[P] el proceso pid=%d acaba de terminar con estado %d\n", p, status);
  }

  return 0;
}
```

Ejercicio 2. *Modifica el programa del ejercicio anterior para que el proceso padre, después de escribir su traza, espere a que termine el proceso hijo creado después de la llamada fork, pero en este caso, utilizando la llamada waitpid.*

Solución:

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

int main(void) {
  int pid = fork();
  if (pid==0){
    sleep(1);
    printf("[H] ppid = %5d, pid = %5d\n", getppid(), getpid());
  } else {
    int p, status;
    printf("[P] ppid = %5d, pid = %5d, H = %5d\n", getppid(), getpid(), pid);
    p = waitpid(pid, &status, 0);
    printf("[P] el proceso pid=%d acaba de terminar con estado %d\n", p, status);
```

```
  }

  return 0;
}
```

Ejercicio 3. *Modifica el programa del anterior. Introduce una llamada exit en el proceso hijo después de escribir su traza con un valor de 33.*

Solución:

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

int main(void) {
  int pid = fork();
  if (pid==0) {
    sleep(1);
    printf("[H] ppid = %5d, pid = %5d\n", getppid(), getpid());
    exit(33);
  }
  else {
    int p, status;
    printf("[P] ppid = %5d, pid = %5d, H = %5d\n", getppid(), getpid(), pid);
    p = waitpid(pid, &status, 0);
    printf("[P] el proceso pid=%d acaba de terminar con esado %d\n", p, status);
  }

  return 0;
}
```

Ejercicio 4. *Modifica el programa del anterior utilizando la macro* `WEXITSTATUS()` *para interpretar el estado del proceso que termina.*

Solución:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void) {
  int pid = fork();
  if (pid==0) {
    sleep(1);
    printf("[H] ppid = %5d, pid = %5d\n", getppid(), getpid());
    exit(33);
  }
  else {
    int p, status;
    printf("[P] ppid = %5d, pid = %5d, H = %5d\n", getppid(), getpid(), pid);
    p = waitpid(pid, &status, 0);
    printf("[P] el proceso pid=%d acaba de terminar con estado %d\n", p, WEXITSTATUS(status));
  }

  return 0;
}
```

Ejercicio 5. *Crea un pequeño programa en C, que muestre por pantalla su PID y acto seguido utilizando la llamada a* `exec` *muestre un calendario por pantalla (usa el comando* `cal`*). Una vez realizado este proceso trata de volver a mostrar el PID que antes obtuviste. ¿Es posible? ¿Por qué?*

Solución:

```
#include <unistd.h>
#include <stdio.h>
int main(void) {
  printf("Mi pid es %5d\n", getpid());
  int ret;
  ret = execlp ("cal", "cal", (char *)0);
  printf("Mi pid es %5d\n", getpid());

  return 0;
}
```

2. Llamadas relacionadas con señales

En este apartado trabajaremos con llamadas relacionadas con señales: `sigaction`, `kill`, `alarm` y `pause`. En la Sección 2.4 se proponen una serie de ejercicios para trabajar estas llamadas. Las señales son interrupciones software que pueden llegarle a un proceso comunicando un evento asíncrono (por ejemplo, el usuario ha pulsado `Ctrl+C`). Las señales que gestiona el sistema operativo están definidas en el archivo `/usr/include/signal.h`, cuyo contenido es el siguiente:

```
#define SIGHUP       1 /* hangup */
#define SIGINT       2 /* interrupt (DEL) */
#define SIGQUIT      3 /* quit (ASCII FS) */
#define SIGILL       4 /* illegal instruction */
#define SIGTRAP      5 /* trace trap (not reset when caught) */
#define SIGABRT      6 /* IOT instruction */
#define SIGBUS       7 /* bus error */
#define SIGFPE       8 /* floating point exception */
#define SIGKILL      9 /* kill (cannot be caught or ignored) */
#define SIGUSR1     10 /* user defined signal # 1 */
#define SIGSEGV     11 /* segmentation violation */
#define SIGUSR2     12 /* user defined signal # 2 */
#define SIGPIPE     13 /* write on a pipe with no one to read it */
#define SIGALRM     14 /* alarm clock */
#define SIGTERM     15 /* software termination signal from kill */
#define SIGEMT      16 /* EMT instruction */
#define SIGCHLD     17 /* child process terminated or stopped */
#define SIGWINCH    21 /* window size has changed */
```

Una señal también puede enviarse por errores de ejecución, como SIGILL (intento de ejecutar una instrucción ilegal) o SIGSEGV (intento de acceder a una dirección inválida). La expiración de una temporización (llamada *alarm*), también provoca que se envíe una señal (SIGALRM).

Un proceso puede seleccionar qué hacer si le llega una señal concreta, optando entre:

1. Dejar el tratamiento por defecto.
2. Ignorar la señal (salvo para SIGKILL).
3. Capturar la señal y tratarla de forma específica.

2.1. *sigaction*

sigaction nos permite elegir qué hacer cuando un proceso recibe una señal. Su prototipo es el siguiente:

```
int sigaction (int sig, const struct sigaction *act, struct sigaction *oact)
```

Donde:

`sig` es la señal que queremos tratar.

`act` es un puntero a una estructura que define el comportamiento del proceso cuando llegue la señal `sig`.

`oact` es un puntero a una estructura donde el sistema dejará el comportamiento que tenía la señal `sig` por si más adelante queremos restaurarlo.

La estructura `sigaction` permite definir el comportamiento del proceso ante la llegada de una señal. Se define en el fichero `/usr/include/signal.h` como sigue:

```
struct sigaction {
  __sighandler_t sa_handler; /* SIG_DFL, SIG_IGN, or pointer to function */
  sigset_t       sa_mask;    /* signals to be blocked during handler */
  int            sa_flags;   /* special flags */
};
```

El programa `sigaction.c` ilustra el uso de la llamada *sigaction*.

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

struct sigaction sa;

void handler (int sig) {
  printf ("SIGINT received %d\n", sig);
}

int main(void) {
  sa.sa_handler = handler;

  sigaction (SIGINT, &sa, NULL);
```

```
  while(1) {}

  return 0;
}
```

2.2. *kill*

El prototipo de la llamada *kill* es el siguiente:

```
int kill(pid_t pid, int sig)
```

Esta llamada envía la señal `sig` al proceso `pid`.

`kill.c` ilustra el funcionamiento de *kill*. El programa utiliza la llamada *fork* para crear un proceso hijo que ejecuta un bucle relativamente grande. Mientras, el proceso padre está esperando una orden nuestra (basta con pulsar INTRO) para matar al hijo.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
  int pid;

  pid = fork();
  if (pid==0) {
    int i;
    for (i=1; i<10000; i++) {
      printf ("%c", 'H');
      if ((i%60)==0) printf ("\n");
    }
    exit(33);
  } else {
    int result, status;
    char c;
    scanf("%c", &c);
    result = kill(pid, SIGKILL);
    printf ("[P] SIGKILL sent to pid=%d with result=%d\n", pid, result);

    result = wait(&status);

    if (WIFEXITED(status))
      printf("[P] pid=%d finished with status=%d\n", result, WEXITSTATUS(status));
    else
      printf("[P] pid=%d killed by signal=%d\n", result, status & 0xf);
  }

  return 0;
}
```

2.3. *pause*

El prototipo de la llamada *pause* es el siguiente:

```
int pause (void)
```

Esta llamada suspende la ejecución del proceso que la ejecuta hasta que llegue una señal.

2.4. Ejercicios

Ejercicio 6. *Compila* `sigaction.c` *visto en la Sección 2.1, ejecútalo, intenta terminarlo pulsando* `Ctrl+C` *y observa que ocurre.*

Para terminar el proceso utiliza la señal `SIGKILL`*.*

Ejercicio 7. *Modifica* `sigaction.c` *para que el programa ignore la señal* `SIGINT` *en lugar de escribir un mensaje en la salida estándar.*

Solución:

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

struct sigaction sa;
```

```
int main(void) {
  sa.sa_handler = SIG_IGN;

  sigaction (SIGINT, &sa, NULL);

  while(1) {}

  return 0;
}
```

Ejercicio 8. *Compila* `kill.c` *visto en la Sección 2.2, ejecútalo 2 veces, primero espera a que termine el proceso hijo antes de pulsar INTRO, luego, pulsa INTRO antes de que termine el proceso hijo. Observa que ocurre.*

3. Llamadas relacionadas con ficheros

En este apartado trabajaremos con llamadas relacionadas con ficheros: `open`, `read`, `write` y `close`. En la Sección 3.4 se proponen una serie de ejercicios para trabajar estas llamadas.

3.1. *open*

Para acceder a un fichero utilizamos la llamada *open*, cuyo prototipo es el siguiente:

```
#include <unistd.h>

int open(const char *path, int flags [, mode_t mode])
```

open abre el fichero cuya ruta indica en el argumento `path`, devolviendo el menor descriptor disponible.

Los valores que pueden utilizarse como *flags* se muestran en el cuadro 1. El parámetro `mode`(el tamaño de esta variable son 4 bytes) indica el modo de protección del fichero si es de nueva creación, los valores se explican en el cuadro 2.

Flag	Significado
O_RDONLY	Sólo lectura.
O_WRONLY	Sólo escritura.
O_RDWR	Lectura y escritura.
O_APPEND	Se sitúa al final del fichero.
O_CREAT	Crea el fichero si no existe.
O_TRUNC	Trunca el tamaño del fichero a cero.
O_EXCL	Con O_CREAT, si existe el fichero, falla.

Cuadro 1: Valores del argumento `flags` en la llamada *open*.

Número	Binario	Lectura (r)	Escritura (w)	Ejecución (x)
0	000	No	No	No
1	001	No	No	Sí
2	010	No	Sí	No
3	011	No	Sí	Sí
4	100	Sí	No	No
5	101	Sí	No	Sí
6	110	Sí	Sí	No
7	111	Sí	Sí	Sí

Cuadro 2: Valores del argumento `mode` en la llamada *open*.

3.2. *close*

Para cerrar un fichero cuando hemos acabado de trabajar con el utilizamos la llamada *close*, cuyo prototipo es el siguiente:

```
int close(int fildes);
```

`fildes` es el descriptor del fichero que queremos cerrar.

3.3. *read* y *write*

Para leer y escribir un fichero utilizamos las llamadas *read* y *write* cuyos prototipos se muestran a continuación:

```
#include  <fcntl.h>
int read(int handle, void  *buffer, int nbyte);
int write(int handle, void  *buffer, int nbyte);
```

Donde:

`handle` es el descriptor del fichero que vamos a leer o escribir.

`*buffer` es un puntero al buffer en el que vamos a guardar los datos leidos (*read*) o del que sacamos los datos que vamos a escribir (*write*) en el fichero.

`nbyte` número de bytes que queremos leer o escribir.

La llamada *read* devuelve el número de bytes leidos como valor de retorno. En caso de llegar al final del fichero, devuelve el valor 0. En caso de error devuelve el valor -1.

La llamada *write* devuelve el número de bytes escritos o -1 en caso de error.

3.4. Ejercicios

Ejercicio 9. *Escribe un programa que reciba como argumento la ruta de un fichero. Si el fichero no existe, el programa debe crearlo y escribir "Hola mundo" dentro. Si el fichero existe, el programa debe sobreescribir su contenido por "Hola mundo".*

Ejercicio 10. *Escribe un programa que reciba como argumento la ruta de un fichero. Si el fichero no existe, el programa debe mostrar un error. Si el fichero existe, el programa debe leer y mostrar su contenido en pantalla.*

Utiliza ficheros de texto plano de diferentes longitudes para probarlo.

Ejercicio 11. *Escribe un programa que emule el comportamiento del comando cp. Que reciba dos argumentos: la ruta origen de un fichero y la ruta destino donde queramos copiar ese fichero.*

Para hacerlo, necesitarás las llamadas open, read, write y close. Las lecturas y escrituras las haremos sobre un buffer de 4096 B (`char buffer[4096]`).

4. Llamadas para comunicar procesos

En este apartado trabajaremos con llamadas relacionadas con procesos: *pipe* y *dup2*.

4.1. *pipe*

Una forma de comunicar procesos, es el uso de ficheros especiales denominados *pipes* que se crean con la llamada al sistema *pipe* cuyo prototipo es el siguiente:

```
int pipe(int fildes[2])
```

Esta llamada crea un mecanismo especial de entrada/salida de tal forma que se dispone de dos descriptores de fichero:

- fildes[0]: De solo lectura.
- fildes[1]: De solo escritura.

La escritura en un pipe (a través de fildes[1]) permite ir almacenando octetos en el pipe (hasta un máximo de 7.168 en MINIX 3) antes de que se bloque al proceso. Posteriores lecturas del pipe (a través de fildes[0]), leerán los caracteres previamente almacenados por las escrituras.

4.2. *dup2*

La llamada *dup2* permite generar un duplicado de un descriptor existente, tiene el siguiente prototipo:

```
int dup2(int oldd, int newd)
```

Esta llamada duplica el descriptor de fichero `oldd` devolviendo como nuevo descriptor de fichero `newd`. Si `newd` se corresponde con un fichero previamente abierto, lo cierra antes de hacer el duplicado. Tras ejecutarse *dup2*, es indistinto utilizar `oldd` o `newd` para acceder al fichero que inicialmente sólo manipulábamos a través de `oldd`. Con esta facilidad se puede redirigir la E/S de un proceso.

4.3. Ejercicios

Ejercicio 12. *Escribe un programa en el que el programa principal cree un proceso hijo. El proceso hijo debe escribir "Hola mundo" en el extremo de escritura de un Pipe. El padre debe leer el contenido del extremo de lectura del Pipe y mostrarlo por pantalla.*

Ejercicio 13. *Haciendo uso de la llamada pipe escribe un programa que utilice la llamada fork y que escriba unas trazas similares a estas:*

```
[P] Mi padre=PADRE, yo=YO, mi hijo=HIJO
[H] Mi padre=PADRE, yo=YO, mi abuelo=ABUELO
```

En el proceso padre `PADRE` *se corresponde con la salida de la llamada getppid,* `YO` *con la de getpid e* `HIJO` *con el PID del proceso hijo. En el proceso hijo* `PADRE` *es el PID del proceso padre (salida de la llamada getppid),* `YO` *es la salida de getpid y* `ABUELO` *es el PID del padre del proceso padre. Para hacerlo tendrás que pasar del padre al hijo el valor de* `ABUELO` *a través de un pipe.*

Ejercicio 14. *Escribe un programa que ejecute el comando* `ls | wc`*. El programa debe utilizar fork para crear un proceso hijo. El proceso padre debe ejecutar ls con la llamada exec. El proceso hijo debe ejecutar wc con la llamada exec. La salida del comando ejecutado por el proceso padre debe llegarle como entrada estándar al proceso hijo a través de un pipe.*

Además de exec, tendrás que utilizar las llamadas dup2 y pipe.

NOTA: Los descriptores de la entrada y salida estándar son 0 y 1 respectivamente.

5. Práctica entregable

En esta sección se especifíca la práctica entregable correspondiente al bloque de llamadas al sistema. La práctica consistirá en el desarrollo de dos programas **mycat.c** y **myshell-basic.c**. A continuación se especifican los requisitos de ambos programas.

5.1. Desarrollo del comando *mycat*

El objetivo de esta parte es desarrollar un programa en c que emule el comportamiento del comando cat. El programa deberá tomar cómo parámetro un único argumento que se corresponderá con la ruta de un fichero. El programa debe hacer lo siguiente:

1. Si el fichero no existe, indicar una traza del tipo: "No such file or directory" y terminar el programa.
2. Si el fichero existe, el programa debe leer su contenido e imprimirlo por la salida estándar.

5.2. Desarrollo de un intérprete de comandos

En este apartado se propone el desarrollo de un intérprete de comandos totalmente funcional. Para hacerlo tendrás leer el comando que introduzca el usuario y combinar las llamadas *fork*, *wait*, *exec* y *exit*.

5.3. Lectura de comandos

Para empezar, compila el siguiente programa:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>

#define MAX_TOKENS 100 // Maximum number of tokens to parse
#define MAX_TOKEN_LENGTH 100 // Maximum length of each token

void tokenize(char *str, char **tokens, int *num_tokens) {
    *num_tokens = 0;
    char *tok = strtok(str, " ");
    while (tok != NULL && *num_tokens < MAX_TOKENS) {
        int len = strlen(tok);
        if (len > MAX_TOKEN_LENGTH - 1) {
            printf("Error: token '%s' is too long.\n", tok);
            return;
        }
        for (int i = 0; i < len; i++) {
            tok[i] = tolower(tok[i]); // convert each token to lowercase
        }
        tokens[*num_tokens] = tok;
```

```
        (*num_tokens)++;
        tok = strtok(NULL, " ");
    }
    tokens[*num_tokens] = NULL;
}

int main() {
    char str[1024];
    char *tokens[MAX_TOKENS];
    int num_tokens;

    printf("Introduce un commando (p.e. ls -l -a): ");
    fflush(stdout);
    fgets(str, sizeof(str), stdin);
    str[strlen(str)-1] = '\0';

    tokenize(str, tokens, &num_tokens);
    printf("Number of tokens: %d\n", num_tokens);
    for (int i = 0; i < num_tokens; i++) {
        printf("Token %d: %s\n", i+1, tokens[i]);
    }

    if (execvp(tokens[0], tokens) == -1)
        printf("Error al ejecutar el comando '%s': %s\n", tokens[0], strerror(errno));

    return 0;
}
```

Intenta comprender su funcionamiento. Para probarlo introduce el comando `ls -l -a`, cuando el programa te lo solicite.

5.4. Funcionamiento básico de un intérprete

Para la funcionalidad básica de nuestro intérprete, usaremos las llamadas *fork*, *wait*, y *exec*. El intérprete debe seguir estos pasos:

1. Nuestro programa solicitará al usuario que introduzca un comando. Para facilitar el desarrollo de nuestro intérprete utilizaremos la función *tokenize* del programa anterior.
2. Una vez leído el comando usaremos *fork* para crear un proceso hijo.
3. El proceso padre debe ejecutar *wait* para esperar a que el proceso hijo termine.
4. El proceso hijo debe ejecutar el introducido por el usuario mediante la llamada *exec*.
5. El programa debe ejecutar un bucle infinito para permitir al usuario meter más de un comando. Para cerrar el bucle se debe capturar la señal **SIGINT**. Cuando la señal se recibe, el programa preguntará al usuario si desea salir (introduciendo s/n por la entrada estándar).

5.5. Funcionalidad avanzada

Ahora queremos que nuestro intérprete sea capaz de ejecutar procesos en segundo plano y de ejecutar pipes.
Para ello, tienes que procesar el comando de entrada y buscar los caracteres '`&`' y '`|`'.
Esta parte debe entregarse en un tercer archivo: **myshell-advanced.c**. Este nuevo programa debe ser una copia del anterior, pero con las modificaciones necesarias para incorporar la funcionalidad avanzada.