6.1. Llamadas al sistema básicas de control de procesos

En esta actividad se comprueba el funcionamiento de diferentes llamadas al sistema que sobre la ejecución de nuevos programas en un proceso y la creación de procesos.

fork, wait

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
void main () {
  int a, x, idh, ret;
  a = 1;
  printf("\nAntes del fork\n");
  x=fork();
  printf("Después del fork!!!!!!!!\n");
  if (x==0) { // hijo
    a = 2;
    printf("Soy hijo %d \t a %d\n",getpid(), a);
    exit(a);
  }
  else { // padre
    printf("    Yo soy tu padre, a %d\n",a);
    idh=wait(&ret);
    printf("    Hijo terminado, a=%d\t idh %d \t ret %d\n",a, idh, ret/256);
  }
}
```

Fichero 6.1: `a.c`

Observa, analiza y prueba este programa y los siguientes.

Usa la llamada `sleep(50)` junto con `exit(3)` (código de salida 3) para que hijo o padre tarde más y comprobar cómo queda el hijo zombi o huérfano. Usa `ps aux` o `htop` y `pstree` para observar los procesos.

fork+execlp+perror

`exec` es una familia de llamadas al sistema que sustituyen el binario que se ejecuta en un proceso con el de un fichero ejecutable. Además se suministra al proceso un vector de cadenas `argv` con diferentes formatos.

Podemos observar que sólo se ejecuta la línea `printf` si no existe el ejecutable `prg` en el mismo directorio. Se debe a que si falla el `exec` se sigue ejecutando este código por lo que se llega a la línea siguiente al `exec`

```c
#include<unistd.h>
#include<stdio.h>
void main(){
  execlp("./prg","prog","drg","aesrgeg",NULL);
  printf("No existe el ejecutable prg");
}
```

Fichero 6.2: `lanza.c`

En el siguiente programa se combina el uso de `fork` con el de `exec` para que el proceso hijo pase a ejecutar el código del programa `ls`

`perror` combina el mensaje de error que escribimos en el programa con el que ha devuelto la anterior instrucción.

Se recomienda sustituir `execlp` por `execl` para comprobar la diferencia al no encontrar el ejecutable `ls` porque no usa el `PATH`.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void main (int argc, char * argv[]) {
  int x, idh, ret;
  x=fork();
  if (x==0) { // hijo
    execlp("ls","ls","-l", argv[1], NULL);
    perror("no encuentro");
  }
  else { // padre
    idh=wait(&ret);
    printf("\nidh %d \t ret %d",idh,ret);
  }
}
```

Fichero 6.3: `f.c`

`fork`, `wait` y códigos de salida

Este programa es conceptualmente las mismas operaciones en cuanto a llamadas al sistema que el anterior.

Es un código más ordenado, que usa `switch` y funciones, comprueba valores por si hay errores como que no sea posible una cierta operación, y procesa los valores que da el hijo como resultado con macros (en mayúsculas al final del programa).

El hijo puede informar al padre de que ha terminado con `exit` o por una señal. En este caso el programa del hijo siempre debería terminar con la llamada al sistema `exit`

Más adelante veremos el caso de un proceso que termine por recibir una señal.

```c
/* wait: parent waits for a child prcess to exit. */
/*  The child runs "ls -aF /". The parent sleeps until the child is done  */
/* Paul Krzyzanowski */

#include <stdlib.h> /* needed to define exit() */
#include <unistd.h> /* needed for fork() and getpid() */
#include <errno.h>  /* needed for errno */
#include <stdio.h>  /* needed for printf() */

int main(int argc, char **argv) {
  void child(void); /* the child calls this */
  void parent(int pid); /* the parent calls this */
  int pid;  /* process ID */

  switch (pid = fork()) {
  case 0:   /* a fork returns 0 to the child */
    child();
    break;

  default:  /* a fork returns a pid to the parent */
    parent(pid);
    break;

  case -1:  /* something went wrong */
    perror("fork");
    exit(1);
  }
  exit(0);
}

void child(void) {
  printf("About to run ls\n");
  execlp("ls", "ls", "-aF", "/", (char*)0);
  perror("execlp"); /* if we get here, execlp failed */
  exit(1);
}

void parent(int pid) {
  int got_pid, status;
```

```
40
  while (got_pid = wait(&status)) {   /* go to sleep until something happens */
    /* wait woke up */
43    if (got_pid == pid)
      break;  /* this is what we were looking for */
    if ((got_pid == -1) && (errno != EINTR)) {
46      /* an error other than an interrupted system call */
      perror("waitpid");
      return;
49    }
  }
  if (WIFEXITED(status))  /* process exited normally */
52    printf("child process exited with value %d\n", WEXITSTATUS(status));
  else if (WIFSIGNALED(status)) /* child exited on a signal */
    printf("child process exited due to signal %d\n", WTERMSIG(status));
55  else if (WIFSTOPPED(status))  /* child was stopped */
    printf("child process was stopped by signal %d\n", WIFSTOPPED(status));
}
```

Fichero 6.4: `wait.c`

fork+for

¿Cuántos hijos tiene el programa principal? ¿Qué hace cada hijo? ¿Qué hace el padre?

```
#include <stdio.h>
#include <string.h>
3 #include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
6 #include <sys/wait.h>
void main () {
  int i, x, idh, ret;
9  for (i=1; i<=5;i++) {
    x=fork();
    if (x==0) { // hijo
12    printf("\nSoy el hijo %d\n",i);
      sleep(i);
      exit(i) ;
15    }
  }
  // padre
18 printf("\nSoy el padre\n");
  while ((idh=wait(&ret)) != -1)
    printf("\nidh %d \t ret %d",idh,ret/256);
21 printf("\nFin\n");
}
```

Fichero 6.5: `s.c`

argv,v

Este programa ejecuta un comando de `bash` que le pasemos detrás del nombre del programa. Si lo compilamos llamando *e* al ejecutable y arrancamos el programa `./e` pasándole las cadenas `haz, ls, -l` que componen su `argv` podremos crear otro vector de cadenas para usarlo en `execve` de la siguiente forma:

```
gcc -o e ejecuta.c
2 ./e haz ls -l
```

Como resultado el hijo ejecutará el programa `ls` con la opción `-l`

El programa `ejecuta.c` será el siguiente:

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

void main (int argc, char * argv[]) {
  int pidhijo,p,r;

  // Un puntero a punteros a char es un puntero al primer elemento de un vector de cadenas
  // Una cadena es un puntero a char
  char ** v;
  v = argv;
  v+=2;

  printf("\nargv contiene:\n%s, %s, %s, %s\n", argv[0],argv[1],argv[2],argv[3]);
  printf("\nv contiene:\n%s, %s\n", v[0],v[1]);
  printf("\n--------------------\n");

  pidhijo=fork();
  if (pidhijo==0) {
    execvp(v[0], v);
  }
  else {
    p=wait(&r);
    printf("\nA. Pid del hijo=%d\n",pidhijo);
    printf("\nB. Pid del hijo=%d\n",p);
    printf("\nC. Res=%d\n",r/256);
  }
}
```

Fichero 6.6: `ejecuta.c`

`argv` es un *array* de cadenas. El nombre de un vector o *array* es un puntero al primer elemento del vector, y dado que cada elemento (cada cadena) es en realidad un puntero a carácter, el nombre del vector es un puntero a punteros a carácter (`**char`).

Ejecuta diferentes comandos internos y externos

Como `busybox`, este programa `caja` muestra `argv` y es ejemplo de diferentes formas de ejecutar.

Si no tiene suficientes argumentos muestra una ayuda.

Si el primer argumento no es un comando, muestra *"comando incorrecto"*

Si es el comando `eco` imprime la siguiente cadena a `eco`

Si es el comando `info` ejecuta el comando `uname -a` a través de la instrucción de C `system`

Si es el comando `ll` ejecuta `ls -l` con un `exec`

Si es el comando `rm` borra un fichero con la llamada al sistema `unlink`

Si es el comando `ejecuta` recorta `argv` para ejecutar el comando que sigue, con sus argumentos.

```
./caja
./caja a a
./caja eco a
./caja info
./caja ll
./caja rm hola.txt
./caja ejecuta
./caja ejecuta ls -i
```

El programa `caja.c` es el siguiente:

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
void muestraargs(int argc, char * argv[]) {
    int i;
    char * c;
    c=*argv;
        for (i=0; i< argc; i++) {
            while(*c!=0) {
                printf("argv[%d] \t%3d   \t %c \n", i, *c, *c);
                c++;
            }
            printf("argv[%d] \t%3d   \t %c \n",i , *c, *c);
            *c++;
        }
}

void ejecuta(int argc, char * argv[]) {
    int i;
    char ** v = argv;
    v+=2;
    for (i=0; i< argc; i++)
        printf("\nargv[%d]: %s\t", i,argv[i]);
    for (i=0; i< argc-2; i++)
        printf("\nv[%d]: %s\t", i,v[i]);
    printf("\nResultado: \n");
    if (argv[2]==NULL)
        printf("Uso: %s ejecuta instruccion\n", argv[0]);
    else
        execvp(argv[2], v); // execvp
}

void procesa(int argc, char * argv[]) {
    if (strcmp(argv[1],"eco")==0)
        printf("%s\n", argv[2]);
    else if (strcmp(argv[1],"info")==0)
        system("uname -a");           // varios parÃ¡metros de system (nivel C)
    else if (strcmp(argv[1],"ll")==0) {
        execlp("ls","ls","-l",NULL);  // llamada al sistema execlp
        printf("\n ############# \n");
        perror("no encuentro");
    }
    else if (strcmp(argv[1],"rm")==0) { // borrar enlaces duros
        if (unlink(argv[2]) != 0)
            perror("Problema");
    }
    else if (strcmp(argv[1],"ejecuta")==0)
        ejecuta(argc,argv);
    else
        fprintf(stderr,"Comando incorrecto\n");

}

int main(int argc, char * argv[]) {
    muestraargs(argc, argv);
    if (argc<2) {
        printf("Uso: %s instruccion [nombre_de_fichero]\n", argv[0]);
        exit(1);
    }
    else
        procesa(argc,argv);
    return 0;
}
```

Fichero 6.7: `caja.c`

6.2. Creación de un intérprete de comandos básico.

Vamos a crear nuestro propio intérprete de comandos. Con él mejoraremos nuestras capacidades de programación y veremos cómo controla la ejecución de programas una línea de comandos como `bash`

`lanzador1`

La primera versión que llamaremos `lanzador1` queremos que tenga únicamente la funcionalidad de ejecutar un programa en modo `RUN`. Ejecutar un `programaX` en modo `RUN` quiere decir que una vez lanzado el `programaX` el lanzador1 debe esperar a que el `programaX` finalice antes de continuar con la tarea de lanzar otro programa. El `lanzador1` termina cuando se detecte fin de fichero en la entrada estándar.

El `lanzador1` debe proporcionar las siguientes características propias de los interpretes de comandos: cada vez que el lanzador1 esté en condiciones de ejecutar un nuevo comando, debe mostrar un *prompt* al usuario. En este caso el prompt será "`Lanzador1>`"

La persona usuaria, una vez que observa el *prompt* puede introducir un comando. Para ello escribirá en una sola línea el nombre de programa a ejecutar seguido de sus argumentos ,todos ellos separados por al menos un blanco o tabulador.

Para realizar el análisis de la línea de comandos dispones de la función ya programada `ExtraeArgs()`. Esta función se encuentra en el fichero `extrae.h`

Mejora del lanzador añadiendo el modo de ejecución RUN o SPAWN.

Vamos a mejorar nuestro lanzador1 proporcionándole la posibilidad de ejecutar un comando en modo `RUN` o en modo `SPAWN` de la misma forma que hacemos en cualquier linux con `sleep 10 &` o `sleep 10`. Para ello vamos a modificar la sintaxis de la línea de comandos:

El usuario, una vez que observa el *prompt* puede introducir un comando. Para ello escribirá en una sola línea el modo de ejecución `R` o `S` , el nombre de programa a ejecutar seguido de sus argumentos, todos ellos separados por al menos un blanco o tabulador

Ejecutar un `programaX` en modo `RUN` quiere decir que una vez lanzado el `programaX` el lanzador2 debe esperar a que el `programaX` finalice antes de continuar con la tarea de lanzar otro programa. Ejecutar un `programaX` en modo `SPAWN` quiere decir que el lanzador2 puede pasar a realizar la siguiente tarea sin esperar a que finalice el `programaX`.

El lanzador2 termina cuando se detecte fin de fichero en la entrada estándar o cuando el usuario introduzca el comando `exit`.

En este caso el prompt será "`Lanzador2>`"

```
1 Lanzador2> R cp f1 f2
  Lanzador2> S ls
```

Para implementar `lanzador2.c` te debes utilizar como base el programa `lanzador1.c`.

Cambio del lanzador añadiendo que se muestre el pid de cada proceso hijo al término de su ejecución.

Vamos a cambiar nuestro lanzador2 haciendo que se muestre el pid de cada proceso hijo al término de su ejecución.

En este caso el prompt será "`Lanzador3>`"

Para implementar `lanzador3.c` te debes utilizar como base el programa `lanzador2.c`.

Cambio del lanzador proporcionando un tiempo máximo de ejecución a cada comando.

Vamos a cambiar nuestro lanzador3 proporcionando un tiempo máximo de ejecución a cada comando. Si el programa rebasa el tiempo máximo, se abortara su ejecución y se escribirá el mensaje correspondiente. El formato de la línea de comandos en este caso será : R o S para indicar el modo de ejecución `RUN` o `SPAWN`, el tiempo máximo de ejecución del programa `T= xx` y a continuación el programa con todos los argumentos que necesite.

Por ejemplo:

```
Lanzador4> R T= 3 cp f1 f2
Lanzador4> S T= 4 ls
```

En este caso el *prompt* será "`Lanzador4>`"

Para implementar `lanzador4.c` te DEBES utilizar como base el programa `lanzador3.c`.

Debes probar este lanzador al menos con el programa `pesado` (un bucle infinito).

Creación de una función para ejecutar programas.

En la realización de la tarea previa habrás notado que linux no ofrece una función directa para ejecutar un programa. Sin embargo nos ayudaría de forma notoria el disponer de un función que realice la ejecución de programas. Otra característica que nos interesaría en esa función sería que nos permita realizar la "redirección" los canales estándar a otros ficheros o dispositivos.

a) Busca en la bibliografía o en internet ejemplos de cómo se realiza la redirección de los canales estándar.

b) Implementa la función `ejecutar_programa`

```
int ejecutar_programa (char *nom, char *c0, char *c1, char *c2, char *argv[]);
```

c) Pruébala en una nueva versión del lanzador1. (modo sin redirección)

d) Pruébala en una nueva versión del lanzador2. (modo con redirección)

```
LanzadorX> R ls -al > listado .dat
LanzadorX> R cat < listado .dat
LanzadorX> R cat < listado .dat > f2.dat
```