Guía preparación control 3 (Diciembre 2023)

Ayudante: Nicolás Araya. Profesor: Rodrigo Verschae

I. Manejo de Archivos

En esta sección se trata el manejo de archivos y directorios. Las formas de manejo de archivos corresponden a la creación de archivos, abrirlos, lectura, escritura y cerrar. El manejo de archivos está muy ligado a lo que sería la comunicación entre procesos pesados, pero eso se verá más adelante en la sección correspondiente.

*1) **Lo importante**:* Para el control de los archivos en código es importante conocer los **File Descriptors**. Los FD corresponden a pequeños números enteros que se utilizan principalmente en código para realizar todas las operaciones existentes.

```
nicoa@LAPTOP-T1MGHDFA:~$ ls -l
total 24
drwxr-xr-x 5 nicoa nicoa 4096 Sep 10 01:05 Desktop
drwxr-xr-x 3 nicoa nicoa 4096 Sep  9 22:53 Dev
-rw-r--r-- 1 nicoa nicoa  258 Sep 10 14:07 compiler.sh
drwxr-xr-x 2 nicoa nicoa 4096 Nov  2 15:27 tareaRedes2
-rw-r--r-- 2 nicoa nicoa  599 Sep  6 15:28 test.txt
-rw-r--r-- 2 nicoa nicoa  599 Sep  6 15:28 wena.txt
```

Fig. 1. Implementación de ls -l, se puede apreciar en cada linea permisos del fichero, el número de enlace, nombre del propietario, nombre del grupo al que pertenece, tamaño en bytes, una marca de tiempo y nombre del fichero

- **ln [OPTION]... TARGET**: El comando ln corresponde al hard link que es comparable con los accesos directos. Ejemplo de uso: ln test.txt totest.txt
- **ls [OPTION]... [FILE]...**: ls es utilizado para ver el contenido de las carpetas. Ejemplo de uso : ls -l

II. Operaciones

Para las operaciones de lectura y escritura de archivos se utiliza la librería unistd.h.

A. Escritura

```c
#include <unistd.h>

size_t write(int filedescriptor, const void *buf,
    size_t nbytes);
```

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

int main()
{
    if((write(1, "Hola Mundo\n", 11) != 11))
        write(2, "ERROR EN WRITE\n", 15);
    exit(0);
}
```

En este caso, write solo retorna en la salida standar (stdout) el string.

```
nicoa@LAPTOP-T1MGHDFA:/codigos$ gcc -o writestdout
    writestdout.c
nicoa@LAPTOP-T1MGHDFA:/codigos$ ./writestdout
Hola Mundo
nicoa@LAPTOP-T1MGHDFA:/codigos$
```

B. Lectura

```c
#include <unistd.h>

size_t read(int filedescriptor, void *buf, size_t
    nbytes);
```

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

int main()
{
    char buffer[128];
    int nread;

    /*Recordar que size_t es un entero*/
    nread = read(0, buffer, 128);
    if (nread == -1)
        write(2, "ERROR EN WRITE\n", 15);
    /*Usamos nread como tamanno de buffer*/
    if ((write(1, buffer, nread)) != nread)
        write(2, "ERROR EN WRITE\n", 15);

    exit(0);
}
```

El proposito de este código es copiar los 128 bytes de la entrada estandar (stdin) a la salida estandar (stdout) .

```
nicoa@LAPTOP-T1MGHDFA:/codigos$ gcc -o read read.c
nicoa@LAPTOP-T1MGHDFA:/codigos$ echo wenas | ./read
wenas
nicoa@LAPTOP-T1MGHDFA:/codigos$
```

El método utilizado para la entrada estandar es llamado pipe, se realiza con — y se encarga de enviar el stdout hacia el proceso de la derecha.

C. Abrir Archivos

Open es una función atómica, lo que quiere decir que es realizada por una sola llamada a la funcion. Esto protege que varios programas intenten crear el mismo archivo al mismo tiempo.

Para abrir archivos se utiliza la librería fcntl. La sintaxis de open es la siguiente:

```c
#include <fcntl.h>
/*Dependiendo del sistema Unix, seran o no
*requeridos los siguientes headers
*/
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

Mode	Description
O_RDONLY	Open for read-only
O_WRONLY	Open for write-only
O_RDWR	Open for reading and writing

Fig. 2. Flags para especificar el tipo de operación.

La forma de uso de las flags (ver Fig. 2) va de la mano con las siguientes:

- O_APPEND: Agrega stdout de escritura al final del archivo.
- O_TRUNC: Setea el tamaño del archivo a 0, ignorando el contenido.
- O_CREAT: Crea el archivo tomando en consideración las flags de la Fig. 2.
- O_EXCL: Macro para asegurarse de que O_CREAT haya creado el archivo. En caso de que el archivo ya exista, open falla.

Dependiendo de la situación, las flags serán true o false. La función open puede recibir varias flags mediante la operación bitwise OR (|) y de la siguiente manera:

$$\text{O\_CREAT}|\text{O\_WRONLY}|\text{O\_TRUNC} \quad (1)$$

Esta sería para crear un archivo de solo escritura y complementamos con O_TRUNC para asegurarnos de ocupar toda la memoria designada. Implementado se ve:

```
open(''text.txt'', O_CREAT|O_WRONLY|O_TRUNC)
```

Los permisos (ver Fig. 3) se agregan según la misma lógica:

S_IRUSR: Read permission, owner
S_IWUSR: Write permission, owner
S_IXUSR: Execute permission, owner
S_IRGRP: Read permission, group
S_IWGRP: Write permission, group
S_IXGRP: Execute permission, group
S_IROTH: Read permission, others
S_IWOTH: Write permission, others
S_IXOTH: Execute permission, others

Fig. 3. Lista de permisos para open. **open("myfile", O_CREAT, S_IRUSR|S_IXOTH);**

D. Cerrar Archivos

```c
#include <unistd.h>

int close(int filedescriptor);
```

Como se mencionó anteriormente el File Descriptor es utilizado para todas las operaciones. La función open retorna un File Descriptor que podemos utilizar posteriormente para realizar las operaciones correspondientes al archivo que hemos abierto. Por ejemplo:

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main() {
    int filedescriptor = open("example.txt", O_CREAT
     | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);

    if (filedescriptor == -1) {
        perror("Error opening the file");
        return 1;
    }

    write(filedescriptor, "Hello, File Descriptor!\n
    ", 24);

    close(filedescriptor);

    return 0;
}
```

```
nicoa@LAPTOP-T1MGHDFA:/codigos$ gcc -o close close.c
nicoa@LAPTOP-T1MGHDFA:/codigos$ cat example.txt
Hello, File Descriptor!
nicoa@LAPTOP-T1MGHDFA:/codigos$
```

E. Ejercicios

Con lo aprendido, crear un código para la copia de archivos:

1) Paso 1: Librerías:

```c
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main()
{
```

2) Paso 2: Abrir archivos de lectura y escritura:

```c
int fd_read = open("in.txt", O_RDONLY, S_IRUSR);
int fd_write = open("out.txt", O_CREAT|O_WRONLY|
    O_TRUNC, S_IRUSR|S_IWUSR);
```

3) Paso 3: Lectura del archivo:

```c
char c;

while(read(fd_read, &c, 1) == 1){
```

4) Paso 4: Escritura del archivo:

```c
while(read(fd_read, &c, 1) == 1){
    write(fd_write, &c, 1);
}

exit(0);
}
```

5) Propuesto 1: Incrementa la cantidad de char que escribe desde el fd de lectura hacia el fd de lectura y haz que reciba el archivo desde la entrada estandar. Pista: Usa un buffer del tamaño de los char que quieres.

6) Propuesto 2: Según la cátedra de manejo de archivos y esta guía realizar lo siguiente:

- **a)** Crear código para cambio de nombre, donde el nombre se recibe mediante un pipe: echo NombreArchivo | ./codigo
- **b)** Cree un link duro para un archivo ya existente y realize a) con link duro para ver que pasa.
- **c)** (Precaución, realizar con las medidas adecuadas. Puede tomar tiempo) Si tiene una máquina virtual, **crear otra que no tenga nada importante** y elimine el sistema con chmod y unlink. Puede usar la obtención de inode para explorar los permisos de los archivos y encontrar el adecuado.

III. Directorios

A continuación se mencionarán funciones útiles para la manipulación de los directorios en un sistema Unix.

A. Opendir

```c
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
```

Según la dirección que se entrega abre el directorio correspondiente y retorna un puntero hacia la estructura DIR en caso de ser una llamada exitosa.

B. Readdir

```c
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *directory_pointer);
```

Este es llamado a través de una estructura que representará todos los atributos del directorio. Se recomienda ver la última ayudantía grabada.

C. Telldir

```c
#include <sys/types.h>
#include <dirent.h>

long int telldir(DIR *directory_pointer);
```

Se utiliza para obtener el valor de la posición actual del directorio.

D. Seekdir

```c
#include <sys/types.h>
#include <dirent.h>

void seekdir(DIR *directory_pointer, long int loc);
```

La función seekdir establece el puntero de entrada de directorio en el flujo de directorio dado por dirp. El valor de loc, utilizado para establecer la posición, debe haberse obtenido de una llamada previa a telldir explicada anteriormente.

E. Closedir

```c
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *directory_pointer);
```

F. Pasos para el manejo de directorios

- Declarar puntero a directorio DIR *dp;
- Llamar a la estructura dirent y al atributo *entry;
- Asignar un directorio a dp mediante opendir u otros.
- Realizar una lectura de los directorios mediante readdir u otras funciones del mismo fin.

```c
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>

int main() {
    // Declarar un puntero a directorio
    DIR *dp;

    // Estructura dirent para almacenar la
    informacion de cada entrada de directorio
    struct dirent *entry;

    // Asignar un directorio a dp mediante opendir
    dp = opendir(".");  // Abre el directorio actual

    if (dp == NULL) {
        perror("Error al abrir el directorio");
        return 1;
    }

    // Realizar una lectura de los directorios
    mediante readdir
    while ((entry = readdir(dp)) != NULL) {
        // Ignorar las entradas "." y ".." que
    representan el directorio actual y el directorio
     padre
        if (strcmp(entry->d_name, ".") != 0 &&
    strcmp(entry->d_name, "..") != 0) {
            printf("%s\n", entry->d_name);
        }
    }

    // Cerrar el directorio
    closedir(dp);

    return 0;
}
```

1) ***Propuestos manejo directorios****:* Realizar un programa de escaneo de directorios. Este debe entregar los subdirectorios y los archivos correspondientes. Puede guiarse de la ayudantía grabada.

IV. Procesos Pesados

Como hemos visto a lo largo del curso, existen 2 tipos de procesos, los procesos livianos o threads y los procesos pesados. En este caso nos enfocaremos en los procesos pesados que corresponden a un puntero a un espacio de memoria de uno o más hilos ejecutandose y que requiere de recursos para sus hilos.

A. Partes de un proceso pesado

Al momento de explorar en nuestra shell de unix utilizando por ejemplo los comandos **ps -af** o **htop** (siendo estos comandos para administrar y ver los procesos activos) podemos descubrir que cada proceso tiene códigos que lo identifican. Estos códigos son llamados PID o Process ID, es importante tener en cuenta esto al momento de trabajar con procesos pesados.

Recordando la estructura de un espacio de memoria destinado a la ejecución de procesos pesados/programas tenemos el siguiente ejemplo (ver Fig. 4):

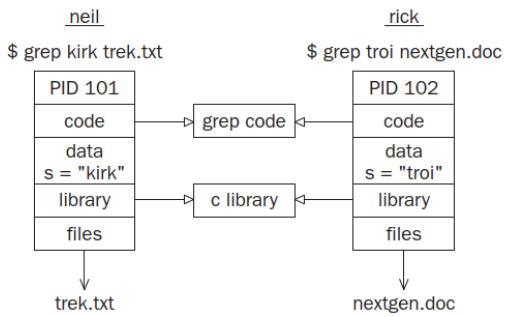

Fig. 4. Corresponde a 2 procesos creados por los usuarios rick y neil.

B. Comenzando un nuevo proceso

En la consola de unix, existe una forma de realizar ejecuciones de procesos a través del stdout llamado **sh -c** donde sh

corresponde al comando utilizado para la ejecución de código bash y al agregar la flag -c nos permite ejecutar comandos de consola.

```c
#include <stdlib.h>

int system(const char *string);
```

La función system, puede hacer que se ejecute un programa dentro de otro programa creando un nuevo proceso, esto porque system realiza un sh -c en el string que se le entregue.

El siguiente código es una implementación de system:

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

int main()
{
    printf("Running System\n");
    system("ps -ax");
    printf("DONE\n");

    exit(0);
}
```

Puede probar la función implementando más funciones que usted conozca de unix.

En general, utilizar system está lejos de ser la forma ideal de iniciar otros procesos, porque llama al programa deseado utilizando la consola. Esto es ineficaz, porque se inicia un intérprete de comandos antes de que se inicie el programa.

C. Reemplazar imagen de proceso

Las funciones *exec* reemplazan el proceso actual por uno nuevo especificado por una dirección de archivo o un archivo que recibe como argumento.

```c
#include <unistd.h>

char **environ;

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

Utilizando exec para inicializar el programa **ps** se puede escoger uno de los 6 exec y utilizarlos de la siguiente manera:

```c
#include <unistd.h>

/* Ejemplo de lista con argumentos */
/* Necesitamos nombre de programa para argv[0] */
char *const ps_argv[] =
{"ps", "-ax", 0};

/* Environment */
char *const ps_envp[] =
{"PATH=/bin:/usr/bin", "TERM=console", 0};

execl("/bin/ps", "ps", "-ax", 0);            /*
    asume ps esta en /bin */
execlp("ps", "ps", "-ax", 0);               /* asume
     /bin esta en PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /*
    passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Una buena práctica corresponde a utilizar el comando whereis “comando” para saber la ubicación del comando Unix y poder llamarlo desde exec.

1) ***Propuesto:*** Implemente con cada uno de los 6 exec el comando **ls**.

D. Duplicar imagen de proceso

Para duplicar la imagen de un proceso se utiliza la función fork. Esta crea una nueva entrada en la tabla del proceso con los mismos atributos originales. Este proceso es similar al original, pues contiene los mismos atributos, pero se encuentra en otro espacio de memoria (ver Fig. 5).

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

pid_t fork(void);
```

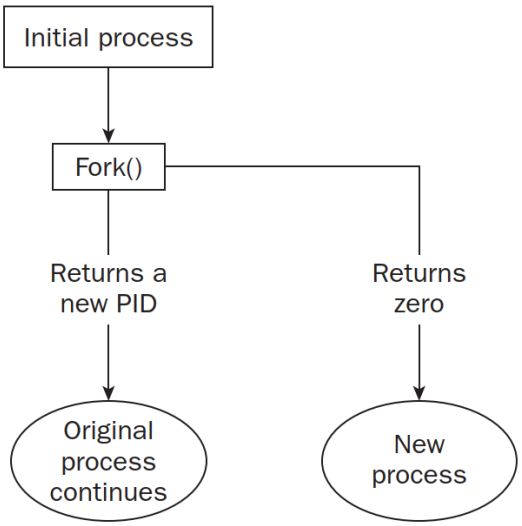

Fig. 5. Corresponde a una representación del funcionamiento de la duplicación de imagen.

Dependiendo del valor de retorno de la función fork existen los siguientes casos primordiales:

- Caso pid_t es -1: Si es valor de retorno es -1, es porque ocurrió un error en la creación de un nuevo proceso. Esto puede deberse a las propias limitaciones del sistema generalmente.
- Caso pid_t es 0: Si es valor es 0, es porque estamos en el proceso hijo correspondiente a la duplicación de la imagen.

1) ***Ejemplos de implementación de fork:*** Antes de pasar a los ejemplos es importante mencionar los pasos para la aplicación de fork en el código:

- El primer paso corresponde a la declaración de la variable tipo pid_t para asignar posteriormente el resultado de la llamada de fork a esta.
- Asignar el resultado de fork a la variable.

- Según el valor de pid_t detectar si nos encontramos en el padre, el hijo o en un error. En este paso existe una liberdad de implementación, pues se puede utilizar cualquier método que evalue según el estado de pid_t.

Mencionar también que es fundamental comprender que, después de la llamada a fork, se crea una duplicación exacta del proceso padre, incluido el código, los datos y el estado del proceso. La ejecución a partir de ese punto es independiente entre el padre y el hijo. Esto significa que cualquier modificación realizada en el proceso hijo no afecta al proceso padre y viceversa. Cualquier cambio en las variables, la memoria o el estado del proceso se realiza de forma independiente, por lo que estos pueden estar trabajando de forma paralela según la instrucción común que estos tengan. (ver Fig. 4)

2) ***Ejemplos****:* A continuación se muestran 2 ejemplos de implementación de fork con switch y con condicionales.

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

int main()
{
    pid_t child;

    child = fork();

    switch (child)
    {
    case -1:
        perror("ERROR");
        exit(0)
    case 0:
        execlp("ps", "ps", "-ax", NULL);
        break;
    default:
        break;
    }

    waitpid(child,(int *) 0, NULL);

    return 0;
}
```

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

int main()
{
    pid_t child;

    child = fork();

    if(child == -1) {
        perror("ERROR");
        exit(0)
    }
    else if(child == 0){
        execlp("ps", "ps", "-ax", NULL);
    }
    else{
        printf("Soy el padre\n");
    }

    waitpid(child,(int *) 0, NULL);

    return 0;
}
```

3) ***Propuesto 1****:* Utilice getpid() de unistd para ver los pid de cada proceso durante la ejecución imprimiendolos en consola.

4) ***Propuesto 2****:* Para ver la ejecución paralela de padre e hijos implemente la estructura de fork y realice lo que estime pertinente para ver en consola la ejecución de ambos procesos. Puede realizar algo similar a rick y neil en la Fig. 4 donde el proceso padre e hijo realizan un grep y observar su ejecución con ps -ax o imprimir en consola de forma simple.

E. zombie, wait, waitpid

1) Zombie: Los procesos zombie corresponden a procesos que el padre no pudo enterrar de forma adecuada. Estos procesos generalmente son controlados por el sistema operativo Unix, pero es posible verlos siendo lo suficientemente rápido mientras el código que genera procesos zombie se está ejecutando.

2) Wait: La función wait espera a que cualquier proceso termine su ejecución recuperando el estado de salida de ese proceso en particular y retornando el PID del proceso terminado.

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

pid_t wait(int *stat_loc);

/*----------------------------------------*/

int status;
wait(&status);
```

Existen formas de rescatar la información obtenida de wait y es mediante macros definidas por sys/wait.h:

Macro	Definition
`WIFEXITED(stat_val)`	Nonzero if the child is terminated normally.
`WEXITSTATUS(stat_val)`	If `WIFEXITED` is nonzero, this returns child exit code.
`WIFSIGNALED(stat_val)`	Nonzero if the child is terminated on an uncaught signal.
`WTERMSIG(stat_val)`	If `WIFSIGNALED` is nonzero, this returns a signal number.
`WIFSTOPPED(stat_val)`	Nonzero if the child has stopped.
`WSTOPSIG(stat_val)`	If `WIFSTOPPED` is nonzero, this returns a signal number.

Fig. 6. Se muestra una lista de las macros que sirven para el análisis de la variable status.

```c
int stat_val;
pid_t child_pid;

child_pid = wait(&stat_val);

printf("Child has finished: PID = %d\n", child_pid);
if(WIFEXITED(stat_val))
printf("Child exited with code %d\n", WEXITSTATUS(
    stat_val));
else
printf("Child terminated abnormally\n");
```

3) Waitpid: Al igual que wait, se encarga de terminar los procesos, pero en este caso, corresponde a un proceso en especifico.

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

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

Recibe el pid del proceso, una variable para asignarle el estado y las flags como WNOHANG que hace que la función retorne 0 en caso de que no termine o se detenga el proceso. waitpid devolverá -1 en caso de error y establecerá errno. Esto puede ocurrir si no hay procesos hijo (errno establecido a ECHILD), si la llamada es interrumpida por una señal (EINTR), o si el argumento de opción es inválido (EINVAL).

F. Pipes

Un pipe es un canal de comunicación entre dos procesos. Se comporta como una cola (FIFO) en donde lo que se escribe por un extremo se lee por el otro. Fue el primer mecanismo que tuvo Unix para comunicar procesos.

Los pipes en consola son implementados con el simbolo —. Este envía el stdout al proceso de la derecha.

Un ejemplo de uso es el siguiente:

```
nicoa@LAPTOP-T1MGHDFA:/codigos$ awk '{print}'
    execexe.c | less
```

```
#include <unistd.h>

/* Ejemplo de lista con argumentos */
/* Necesitamos nombre de programa para argv[0] */
char *const ps_argv[] =
{"ps", "-ax", 0};

/* Environment */
char *const ps_envp[] =
{"PATH=/bin:/usr/bin", "TERM=console", 0};

execl("/bin/ps", "ps", "-ax", 0);             /*
    asume ps esta en /bin */
execlp("ps", "ps", "-ax", 0);                /* asume
     /bin esta en PATH */
execle("/bin/ps", "ps", "-ax", 0, ps_envp); /*
    passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
(END)
```

En este código se implementa awk que corresponde a un lenguaje de programación de patrones y acciones diseñado para el procesamiento y análisis de texto en sistemas Unix. Al usar '{print}' execex.c está imprimiendo cada linea en la consola, pero al aplicar un pipe redirige la salida estandar al comando less que recibe estas lineas y las muestra en pantalla de forma interactiva.

Para trabajar con pipes en C es importante recordar lo que era los file descriptors. Estos son valores enteros mayores que cero utilizados para realizar operaciones a los archivos.

Tal como en una consola de Unix, se realizará la comunicación entre procesos mediante pipes respetando la naturaleza FIFO de este.

Se utilizarán los descriptores para lectura y escritura según correspoda. Se define entonces:

```
#include <unistd.h>

int file_descriptors[2];
pipe(file_descriptors);
```

Donde file_descriptor[0] será utilizado para lectura y file_descriptor[1] para escritura.

G. Ejemplo

Se implementará una función que realice lo mismo: awk '{print}' codigo.c — less.

Para esto se utilizarán dos fork con exec que ejecuten los comandos a petición para posteriormente conectarlos con pipe mediante lectura y escritura definida por los descriptores.

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

int main()
{
    /*Se declaran las variables que representan
     *el PID una vez realizado fork
     */
    pid_t awkpid, lesspid, pid;

    int status = 0;
    /* File descriptors */
    int file_descriptors[2];

    /*En caso de error*/
    if(pipe(file_descriptors) < 0){
        perror("ERROR PIPE\n");
        exit(1);
    }

    /* Se evaluan los casos de manera individual */
    awkpid = fork();
    if(awkpid < 0){
        perror("ERROR FORK\n");
        exit(1);
    }
    if(awkpid == 0){
        /* Para evitar una comunicacion paralela
    entre los
         * procesos se cierra el descriptor que no
    se utilizara
         */
        close(file_descriptors[0]);

        /* Se asigna la escritura a la salida
    estandar */
        close(1);
        dup(file_descriptors[1]);
        /* Una vez duplicado tenemos la informacion
    necesaria */

        close(file_descriptors[1]);

        chdir("/");
        execl("/bin/awk", "awk", "'{print}' read.c",
     0);

    }


    lesspid = fork();

    if(lesspid < 0) {
        fprintf(stderr, "fallo el fork\n");
        exit(1);
    }
    if(lesspid == 0) { /* more */
        /* Cerrar el extremo write del pipe que no
    voy a usar */
        close(fds[1]);
        /* Asigno: 0 (stdin) = fds[0] (lado de
    lectura del pipe) */
        close(0); dup(fds[0]);
        /* Cerrar la copia que me queda sobre el
    pipe o no tendre' EOF*/
```

```
        close(fds[0]);
        execl("/bin/less.txt", "less", 0);
        fprintf(stderr, "Fallo el exec\n");
        exit(-1);
    }
    /* Como padre comun, cierro el pipe, ambos
    extremos (yo no louso) */
    close(fds[0]); close(fds[1]);

    if(waitpid(morepid, &status, 0) != morepid) {
        fprintf(stderr, "fallo el wait2\n");
        perror("wait2");
        exit(-1);
    }
    if(waitpid(lspid, &status, 0) != lspid) {
        perror("wait1");
        exit(-1);
    }

}
```

1) ***Propuesto 1****:* Implementa pipes para que un proceso envíe mensajes y otro lo reciba.

2) ***Propuesto 2****:* Escribe un programa en C que cuente el número de palabras en un archivo de texto. Divide la tarea entre el proceso padre y el proceso hijo. El proceso padre deberá leer el archivo y enviar las palabras al proceso hijo a través de un pipe. El proceso hijo deberá contar las palabras y enviar el resultado al proceso padre.

3) ***Propuesto 3****:* Escribe un programa en C que ordene una lista de números utilizando el comando sort a través de un pipe y exec. El programa principal deberá crear un proceso hijo que ejecute sort utilizando execlp y comunicarse con el proceso hijo a través de un pipe para enviar la lista de números.

V. Señales

Una señal es un evento generado por los sistemas UNIX y Linux en respuesta a alguna condición, tras cuya recepción un proceso puede a su vez realizar alguna acción. Usamos el término ”raise” para indicar la generación de una señal, y el término ”catch” para indicar la recepción de una señal. Esta señal permite controlar la ejecución de los procesos, tanto para el sistema operativo como para el usuario.

Los siguientes correspondien a posibilidades de proceso de una señal:

- Terminar el proceso.
- Ignorar la señal.
- Detener el proceso.
- Reanudar el proceso.

Atrapar la señal mediante una función del programa

Existen una variedad de señales que se activan según una condición especifica (ver Fig. 7)

A. Signal Handling

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

Esta corresponde a una declaración un tanto compleja, pero básicamente dice que esta función recibe dos parámetros. El primero corresponde a la señal que será tomada o ignorada y una función que es llamada en caso de que se reciba la señal correspondiente.

Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating-point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal quit
SIGSEGV	*Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

Fig. 7. Se muestra una variedad de señales, el más interesante corresponde a SIGINT conocido como ctrl + C.

Se especifica si ignorar o no con:

- SIG_IGN para realizar la ignoración.
- SIG_DFL para que la señal actue de forma predeterminada.

Podemos detectar la señal entrante implementado la función signal de la siguiente manera:

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}

int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }

}
```

En el momento que se presione ctrl+C se activará y imprimirá en pantalla el stdout de la función ouch.

2) kill: Un proceso puede enviar una señal a otro proceso, incluido él mismo, llamando a kill. La llamada fallará si el programa no tiene permiso para enviar la señal, a menudo porque el proceso de destino es propiedad de otro usuario. Este es el programa equivalente al comando del shell del mismo nombre.

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

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

kill fallará, devolverá -1, y pondrá errno si la señal dada no es válida (errno puesto a EINVAL), si no tiene permiso (EPERM), o si el proceso especificado no existe (ESRCH).

3) alarm: Las señales nos proporcionan una útil función de despertador. La llamada a la función de alarma puede ser utilizada por un proceso para programar una señal SIGALRM en algún momento en el futuro.

La llamada de alarma programa la entrega de una señal SIGALRM en segundos. De hecho, la alarma se emitirá poco después, debido a los retrasos en el procesamiento y a las incertidumbres en la programación.

```c
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

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

static int alarm_fired = 0;

void ding(int sig)
{
    alarm_fired = 1;
}

int main()
{
    pid_t pid;

    printf("alarm application starting\n");

    pid = fork();
    switch (pid)
    {
    case -1:
        /* Error al hacer fork */
        perror("fork failed");
        exit(1);
    case 0:
        /* Proceso hijo */
        sleep(5); //Espera 5 segundos
        kill(getppid(), SIGALRM); //Envia la sennal SIGALRM al proceso padre
        exit(0);
    }

    /* El proceso padre configura la captura de la sennal SIGALRM con una llamada a signal y luego espera la sennal */
    printf("waiting for alarm to go off\n");
    signal(SIGALRM, ding);

    pause(); // proceso padre se bloquea hasta que reciba una sennal

    if (alarm_fired)
        printf("Ding!\n");

    printf("done\n");

    return 0;
}
```

```c
#include <signal.h>

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

La estructura sigaction, es utilizada para definir las acciones que se llevarán a cabo al recibir la señal especificada por **sig**. Se define en signal.h y tiene al menos los siguientes atributos:

```c
void (*) (int) sa_handler; /*  SIG_DFL o SIG_IGN */
sigset_t sa_mask;   /*  signals to block in
    sa_handler */
int sa_flags;     /*  signal action modifiers */
```

La siguiente es una ejemplificación de cómo utilizar sigaction:

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

int main()
{
    struct sigaction act;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGNAL, &act, 0);
}
```

6) ***Propuesto 1****:* Implemente el código de Signal Handling de más atrás con sigaction utilizando la estructura.

7) ***Propuesto 2****:* Desarrolle un programa que utilice sigaction para manejar la señal SIGTERM. Cuando el programa recibe la señal SIGTERM, deberá imprimir un mensaje y luego finalizar. En caso de no recordar que hace SIGTERM ver figura 7.

8) ***Propuesto 3****:* Escribe un programa que entre en un bucle infinito y maneje las señales SIGINT y SIGTERM. Cuando se reciba SIGINT, el programa debe imprimir un mensaje indicando que se recibió SIGINT. Cuando se reciba SIGTERM, el programa debe imprimir un mensaje indicando que se recibió SIGTERM y luego finalizar de manera controlada.

VI. Bibliografía

[1] Neil Matthew, Richard Stones. Beginning Linux Programming.