LAB 0 - Resumen

Shell: programa informatico se provee una interfaz de usuario para acceder a los servicios del sissema operativo

La shell de Linux usa lenguaje bash

Comandos bash útiles:

- Cat: concatena archivos e imprime por consola (stdout)
  cat copia stdin a stdout
  cat > archivo.txt redirige stdin a stdout

(*) los redirecsores en bash son
> para redirigir la salida del comando, sobreescribiendo el archivo si esse exisse,
>> idem, pero agrega la salida al final del archivo si esse exisse, y
< para redirigir la entrada del comando.
ejemplo
cat > archivo.txt
cat < archivo2.txt

- echo: muestra una linea de texto.
  No puede redirigirse su entrada, pero sí su salida
  echo "hola" imprime hola por pantalla
  echo "hola" > arch.txt enviará hola a arch.txt

- grep:
  sinpaxis:
  grep [OPTION] patterns [FILE]
  busca patrones en cada archivo

Algunas opciones:

+ -i: ignora mayúsculas y minúsculas
+ -v: muestra las líneas que NO contienen el patrón
+ -r: busca en directorios
+ -l: muestra solo los archivos que contienen

- sort: Ordena las líneas de un archivo de texto de manera alfabética y las imprime por consola (stdout)

sintaxis: sort [OPTION] [FILE]

algunas opciones:

+ -r: ordena al revés
+ -n: ordena numéricamente
+ -k Columna: ordena por una columna específica con -t delimitador se especifica el delimitador del campo

- head: imprime por consola las primeras 10 líneas de un archivo

algunas opciones:

+ -n: permite especificar el número de líneas a imprimir

- tail: imprime por consola las últimas 10 líneas de un archivo

algunas opciones:

+ -n: permite especificar el número de líneas a imprimir

- awk:

Sintaxis:

awk 'patrón {acción}' archivo

patrón determina qué líneas serán procesadas

acción qué se ejecutará para las líneas qe cumplen el patrón

ejemplos:

awk '{print $1}' archivo.txt : imprime la 1° columna de caden línea del archivo

awk '{print $0, $7-$8}' archivo.txt: imprime la línea completa y luego la resta entre columna 7 y 8.

- wc: imprime el número de líneas, palabras y bytes del archivo

algunas opciones:

+ -w : imprime el conteo de palabras
+ -l : " " " líneas
+ -m : " " " caracteres
+ -b : " " " bytes

- touch: Su función principal es crear archivos o modificar sus timestamps, pero también puede usarse para crear varios archivos a la vez.

ejemplo:

for i in {1..10}; do touch ./dir/arch${i}.txt

esto crea en la carpeta dir 10 archivos vacíos.

Todos estos comandos pueden tener redirecciones

Conectores de comandos

El conector principal es el pipe (|), si entre dos comandos hay un pipe, la salida del primero será tratada como la entrada del segundo.

Podemos conectar cuantos comandos queramos.

ejemplos:

ls | wc -l: cuenta el número de archivos y subdirectorios del directorio

cat archivo.txt | head -n 3: imprime las primeras 3 líneas de archivo.txt

sort -r -n archivo.txt | tail -n 5: imprime las 5 líneas con menor valor del archivo.

Otro "conector" sería &&: dados dos comandos conectados por &&, si la ejecución del primero finaliza con éxito, se ejecutará el siguiente.

ejemplo:

mkdir new-dir && cd new-dir: Si se crea el directorio con éxito moverse a él.

(*) Si al final del comando escribimos & (uno solo) ese proceso se ejecutará en segundo plano.

jobs despliega los procesos en segundo plano, estos tienen un número y su respectivo PID.

Para terminar el proceso:

kill %n°proceso o

kill PID

Un comando simple es el comando y sus argumentos.

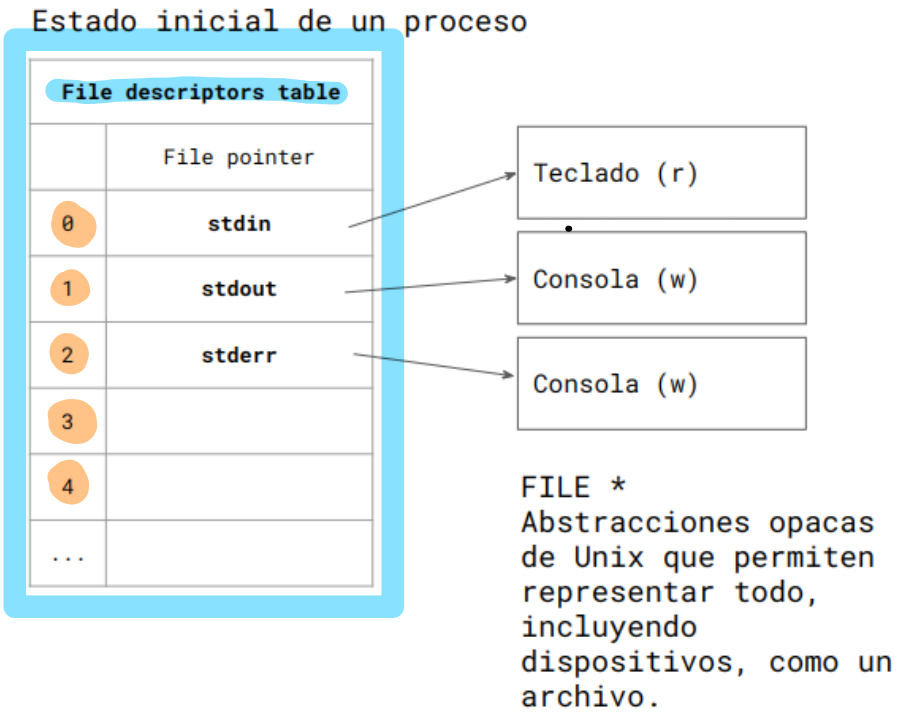
Dos o más comandos simples conectados por un pipe forman un pipeline.

(Lab 1)

LAB 1

Syscalls: forma en que un programa solicita servicios o recursos al sistema operativo (leer y escribir archivos, crear procesos, gestionar memoria, etc.)
Permite que los programas en espacio de usuario soliciten cosas al espacio kernel.

File descriptor: índice (n° entero) a una estructura de datos que contiene detalles de todos los archivos abiertos por un proceso
↳ cada uno tiene su propia tabla de file descriptors



FILE *
Abstracciones opacas de Unix que permiten representar todo, incluyendo dispositivos, como un archivo.

Algunas syscalls...

leer manual

- open(): `int open(const char *pathname, int flags, ...);`

Crea un nuevo file descriptor de archivo abierto (nueva entrada en la tabla de archivos abiertos del sistema)

devuelve un **file pointer** al primer índice no negativo desocupado de la tabla de file descriptors abiertos del proceso.

→ puede haber dos file descriptors apuntando al mismo archivo, hay que tener cuidado cuando estos procesos están emparentados

Resultado de llamar a open("lala.txt", O_RDONLY)

File descriptors table	
	File pointer
0	**stdin**
1	**stdout**
2	**stderr**
3	
4	
...	

Open files table	
	File info
0	"/dev/input"
1	
...	...
197	"lala.txt"
...	

close() va a cerrar los file descrip. creados con open()

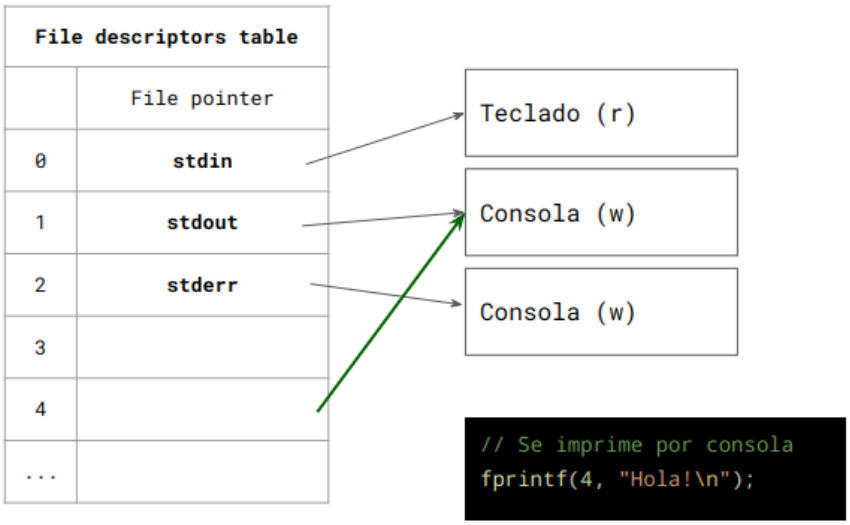
- dup2(): `int dup2(int oldfd, int newfd);`

asigna un nuevo file descriptor que apunta al mismo file descriptor abierto que oldfd.

newfd se ajusta para que ahora apunte a la misma descripción que oldfd

Si estaba abierto, se cierra

Resultado de llamar a dup2(1, 4)



- fork(): `pid_t fork(void);`

Crea un nuevo proceso duplicando el proceso original.
El proceso hijo hereda copias del conjunto de file descriptors del padre. Cada file descriptor del hijo apunta a lo mismo que los file descriptors del padre.

```
pid_t fork_result = fork();
```

Una vez ejecutado el fork, padre e hijo se ejecutan en paralelo

```c
pid_t fork_result = fork();
int pid = getpid();
if (fork_result == -1) {
    printf("fork failed\n");
} else if (fork_result == 0) {
    printf("I'm the child with pid=%d\n", pid);
} else {
    printf(
        "I'm the parent with pid=%d and my child's pid is=%d\n",
        pid,
        fork_result
    );
}
}
```

en caso de éxito, fork(), en el proceso hijo, devuelve un 0

→ aquí pid tiene el PID del hijo, pues estamos en su contexto

Cómo se reparten los PID's después de fork()

- Proceso padre: → mantiene su PID original.
  → fork() devuelve el PID del proceso hijo.

- Proceso hijo: → recibe un nuevo PID.
  → fork() devuelve 0 de forma que queda identificarse que estamos en el contexto del hijo.

- execvp(): `int execvp(const char *filename, char *const argv[]);`

es una función de la familia de funciones exec(), que sustituye la imagen del proceso actual por una nueva imagen de proceso.

Ejecuta el programa apuntado por filename (debe ser un ejecutable binario o un script)

Se modifica el programm counter, el stack, todo MENOS los file descriptors

(*) los ejecutables de los comandos están en /usr/bin

En caso de éxito (la imagen del proceso actual fue sustituida exitosamente), la función no retorna

```c
void  main(void)
{
   char *cmd = "ls";
   char *argv[3];
   argv[0] = "ls";
   argv[1] = "-la";
   argv[2] = NULL;

   // Esto corre "ls -la" como si lo ejecutaramos
   // desde el bash
   execvp(cmd, argv);
   printf("There has been an error\n");
   return 1;
}
```

ejemplito

- wait(): `pid_t wait(int *wstatus);`

Familia de llamadas, otra muy utilizada es waitpid()

El proceso padre se queda esperando a que alguno de sus hijos cambie de estado (termine)

```c
void  main(void) {
   pid_t fork_result = fork();
   int pid = getpid();
   if (fork_result == -1) {
       printf("fork failed\n");
   } else if (fork_result == 0) {
       printf("I'm the child with pid=%d\n", pid);
       printf("Bitch, i'm stopping my father from finishing his proc :*\n");
       sleep(5);
   } else {
       int status;
       wait(&status);
       printf(
           "I'm the parent with pid=%d and my child's pid is=%d\n",
           pid,
           fork_result
       );
   }
}
```

estas dos se ejecutan primero (2)

debido a estas líneas (1)

y después esta (3)

- pipe(): `int pipe(int pipefd[2]);`

Crea un canal de datos unidireccional que sirve para la comunicación entre procesos

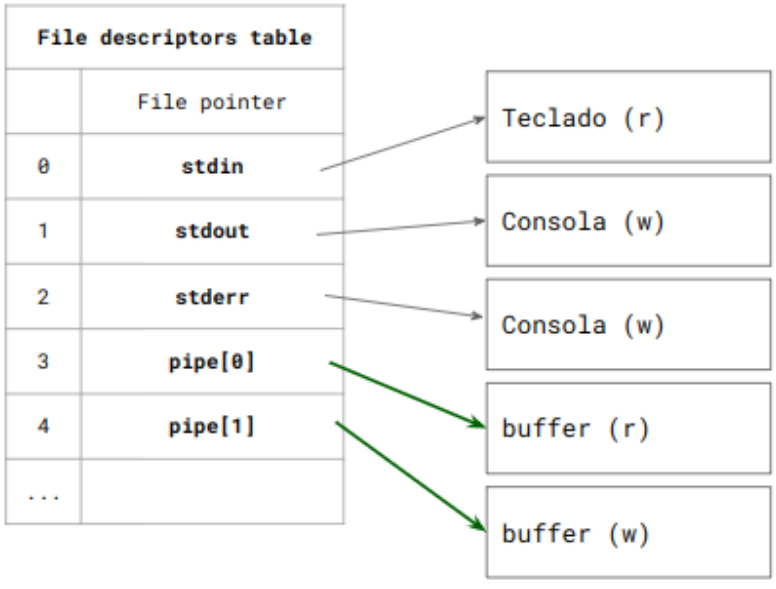
Abre un buffer en el espacio kernel y devuelve en pipefd[2] los file descriptors de lectura y escritura de ese buffer.

Ese buffer es un archivo (todo en UNIX es un archivo) que vamos a usar para que un proceso escriba por un lado y lea por el otro.

pipefd[0]: extremo de lectura

pipefd[1]: extremo de escritura

Resultado de llamar a pipe(pipefd)



Mismo archivo apuntado por dos punteros, uno con permisos de escritura y otro c/ permisos de lectura

¿Para qué lo usamos nosotros?

Si hago un fork luego de hacer un pipe, tanto padre como hijo tienen ese buffer referenciado, entonces podría escribir en ese buffer desde el padre y podría ver esa escritura desde el hijo. ¡Listo! Conexión hecha. :)

Veamos un código de cómo ejecutar un comando simple y un pipeline para que quede claro cómo funcionan todas estas syscalls.

```
void execute_command(scommand exc) {
```

la función tomará un comando simple, con sus argumentos y redirecciones (si tiene)

Obviaremos validaciones, definiciones de variables y otras cosas para ir directo al grano

```
  while (!scommand_is_empty(exc)) {
    strcpy(argv[i], scommand_front(exc));
    scommand_pop_front(exc);
    ++i;
  }
```

(a medida que voy cargando, los voy eliminando del scommand og)

iremos cargando en un arreglo dicho comando con sus argumentos, todavía no vemos las redirects.

```
char* filename_in = scommand_get_redir_in(exc);
if (filename_in) {
    int in_redir = open(filename_in, O_RDONLY, S_IRWXU);
    close(STDIN_FILENO);   // open input redir instead o
    dup(in_redir);
    close(in_redir);
}
```

Si el comando tiene redirecciones de entrada, usamos open() para crear un fd señalando la nueva entrada

Modificamos el fd de la entrada estándar para que apunte a esta nueva entrada.

```
char* filename_out = scommand_get_redir_out(exc);
if (filename_out){
    int out_redir = open(filename_out, O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
    close(STDOUT_FILENO);
    dup(out_redir);
    close(out_redir);
}
```

El proceso para las redirecciones de salida es muy parecido.

```
execvp(argv[0], argv);
```

por último ejecutamos el comando con todos sus argumentos (debe incluirse el comando tmb)

Una vez implementada la función para ejecutar un comando simple, podemos pasar a la de ejecutar un pipeline.

```
void execute_pipeline(pipeline apipe)
```

→ la función toma un pipeline

```
if (pipeline_is_empty(apipe))
    return;
```

Si el pipeline es vacío, retorna sin hacer nada

```
if (builtin_alone(apipe)){
    builtin_run(pipeline_front(apipe));
    return;
}
```

Si el pipeline es un solo builtin (comando interno, lo veremos más adelante) se ejecuta.

```
if (!pipeline_get_wait(apipe))
    signal(SIGCHLD, SIG_IGN);
```

Si el pipeline no requiere que el padre espere a sus hijos, se ignora la señal de estos, evitando que queden "zombies" (el SO los limpia una vez terminados)

```
for (unsigned int i = 0; i < apipe_length; ++i)
```

Las operaciones realizadas a continuación van a realizarse para todos los scommand unidos por un pipe que formen el pipeline

```
if (i != 0) {
    tmp[0] = fildes[0];
    tmp[1] = fildes[1];
}

if (i != apipe_length - 1) {
    pipe(fildes);
}
```

Si estamos sobre algún scommand intermedio, primero debemos guardar los fd del scommand anterior para poder conectar su salida con la entrada del scommand actual.

Si no es el último scommand, creamos un nuevo pipe para conectar su salida con la entrada del siguiente

```
int rc = fork();

if (rc < 0) {
    fprintf(stderr, "FORK FAILED.\n");
    return;
} else if (rc == 0) {

    if (i != apipe_length - 1) {
        close(fildes[0]);
        close(STDOUT_FILENO);
        dup(fildes[1]);
        close(fildes[1]);
    }

    if (i != 0) {
        close(tmp[1]);
        close(STDIN_FILENO);
        dup(tmp[0]);
        close(tmp[0]);
    }

    char* command_str = scommand_to_string(pipeline_front(apipe));
    execute_command(pipeline_front(apipe));
    fprintf(stderr, "Error executing: %s\n", command_str);
    exit(EXIT_FAILURE);
```

chequeamos que no falle

Hacemos un fork() para copiar los fd que creamos recién.

En el proceso hijo, si el scommand no es el último, debemos redirigir su salida (escritura) a la entrada (lectura) del siguiente. Cerramos su extremo de lectura (no vamos a leer nada) y redirigimos su salida estándar al extremo de escritura.

A su vez, si el scommand no es el primero (es intermedio) también debo hacer que lea desde la salida del scommand anterior (sus fd están en tmp[]). Cierro el extremo de escritura y redirijo su entrada estándar al extremo de lectura.

Por último, ejecuto el comando

```
} else {
    if (i != 0) {
        close(tmp[0]);
        close(tmp[1]);
    }
    children_pids[i] = rc;
    pipeline_pop_front(apipe);
}
```

En el proceso padre, si estoy en un comando intermedio, cierro los fd's del comando anterior, guardo el pid de los hijos en un arreglo y voy vaciando el pipeline a medida que ejecuto los comandos.

```
if (pipeline_get_wait(apipe)) {
    for (unsigned int i = 0; i < apipe_length; ++i)
        waitpid(children_pids[i], NULL, 0);
}
```

Si el pipeline requiere esperar a los hijos, hacemos que el padre espere usando el arreglo de pid's de los hijos que armamos a medida que se ejecutaba el pipeline.

Mas arriba se mencionaron los comandos internos, estos son

- cd: se mueve al directorio indicado en sus argumentos. Si no se le pasa ninguno se mueve al home.
  Se implementa de manera directa con la syscall chdir()
  ⟹ chdir(directorio-destino)
- help: muestra un mensaje por la salida estándar indicando nombre del shell (MyBash) y un manual de comandos internos disponibles.
- exit: el shell termina.