Lab0: (Práctica)

Comandos básicos:

`grep [opciones] patrón [archivo...] :` Permite buscar cadenas de texto o patrones dentro de archivos.

Ejemplos:
`-grep "palabra" archivo.txt :` Muestra las líneas que contienen esa palabra en archivo.txt
`-grep -r "palabra" /ruta/del/directorio` : Busca palabra en todos los archivos dentro del directorio
`-grep -i "palabra" archivo.txt :` Ignora minúscula y mayúsculas
`-grep -n "palabra" archivo.txt` : Lineas con la palabra “palabra” y el número de la línea
`-grep -v "palabra" archivo.txt` : Lineas que no contienen la palabra “palabra”
`-grep -e "patrón1" -e "patrón2" archivo.txt` : Esto buscará líneas que contengan cualquiera de los patrones especificados.

`cat` : Concatenate FILE(s), or standard input, to standard output.

Ejemplos:

`-cat archivo.txt` Esto muestra el contenido del archivo `archivo.txt`.
`-cat archivo1.txt archivo2.txt` Esto mostrará el contenido de `archivo1.txt` seguido del contenido de `archivo2.txt`.
`-cat > nuevo_archivo.txt` Después de ejecutar este comando, puedes escribir texto que se guardará en `nuevo_archivo.txt`. Para finalizar, presiona `Ctrl+D`.
`-cat >> archivo_existente.txt` Similar al comando anterior, escribirás el texto que deseas agregar al final de `archivo_existente.txt` y luego presionas `Ctrl+D` para finalizar.
`-cat -n archivo.txt` Puedes mostrar el contenido de un archivo con los números de línea precediendo cada línea.
`-cat -A archivo1.txt archivo2.txt` : Para mostrar el contenido de varios archivos con nombres de archivo precediendo cada sección, puedes usar la opción `-A` o `--show-all`.

`-cat archivo.txt | grep -v '^$'` : Aunque `cat` no tiene una opción directa para eliminar líneas vacías, puedes combinarlo con otras herramientas como `grep` para lograr esto.

`sort` se utiliza para ordenar líneas de texto en archivos o en la entrada estándar.

Ejemplos:

-sort archivo.txt : Puedes ordenar las líneas de un archivo y mostrar el resultado en la salida estándar.

-sort -r archivo.txt : Por defecto, `sort` organiza las líneas en orden ascendente. Para ordenarlas en orden descendente, usa la opción `-r`.

-sort -k 2 archivo.txt : Puedes ordenar por una columna o campo específico usando la opción `-k` seguida del número de campo.

-sort -k 2 archivo.txt : Esto ordenará las líneas de `archivo.txt` basándose en el segundo campo.

-sort -n archivo.txt : Si necesitas ordenar líneas numéricamente en lugar de alfabéticamente, usa la opción `-n`.

-sort -M archivo.txt : Para ordenar fechas y otros datos con formato específico, usa la opción `-M` para meses o `-t` para especificar delimitadores.

-sort -t ',' -k 3 archivo.txt : Aquí, `-t` especifica la coma como delimitador y `-k 3` indica que se debe ordenar por el tercer campo.

-sort -u archivo.txt : Puedes eliminar líneas duplicadas usando la opción `-u` (único).

-sort archivo.txt | less : Para manejar grandes volúmenes de datos, puedes combinar `sort` con `less` para visualizar el resultado en páginas.

-sort archivo.txt > archivo_ordenado.txt : Puedes redirigir el resultado de la ordenación a un nuevo archivo.

El comando `head` se utiliza para mostrar las primeras líneas de uno o más archivos.

Ejemplos:

-head archivo.txt : Por defecto, `head` muestra las primeras 10 líneas de un archivo.

-head -n 20 archivo.txt : Puedes ajustar la cantidad de líneas que deseas ver usando la opción `-n` seguida del número de líneas.

-head archivo1.txt archivo2.txt : Cuando se proporcionan varios archivos, `head` muestra las primeras líneas de cada archivo, precedidas por el nombre del archivo.

-ls -l | head : `head` puede leer desde la entrada estándar, lo que permite usarlo en combinación con otros comandos mediante tuberías. Esto muestra las primeras 10 líneas de la salida del comando `ls -l`.

-head -c 50 archivo.txt : En lugar de mostrar un número específico de líneas, también puedes mostrar un número específico de bytes usando la opción `-c`.

El comando `wc` se utiliza para contar líneas, palabras y caracteres en uno o más archivos.

Ejemplos:

wc archivo.txt : Este comando muestra el número de líneas, palabras y caracteres en el archivo `archivo.txt`.

wc -l archivo.txt : `-l` Muestra solo el número de líneas :

`wc -w archivo.txt : -w` Muestra solo el número de palabras.

`wc -c archivo.txt : -c` Muestra solo el número de caracteres.

wc -l -w -c archivo.txt

Uso en combinación con otros comandos: `wc` se puede usar en tuberías para contar el contenido de la salida de otros comandos. Por ejemplo, para contar el número de líneas en la salida del comando `ls`:

ls | wc -l

En resumen, el comando `wc` es una herramienta versátil para obtener estadísticas básicas sobre el contenido de archivos o de la salida de otros comandos.

El comando `awk` es una herramienta poderosa en sistemas Unix/Linux utilizada para el procesamiento y análisis de texto. Es especialmente útil para manipular datos organizados en columnas y líneas, y se usa ampliamente en scripts de shell para tareas como la extracción, filtrado y formateo de datos.

awk 'patrón { acción }' archivo

`patrón`: Expresión que determina qué registros deben ser procesados. Si se omite, la acción se aplica a todas las líneas.

`Acción`: Comandos que se ejecutan para cada línea que cumple con el patrón. Si se omite el patrón, se aplican a todas las líneas.

Variables Internas

- **`$0`**: La línea completa.
- **`$1`, `$2`, ..., `$n`**: Campos individuales en una línea (el número del campo empieza en 1).
- **`NR`**: Número de registros (líneas) procesados.
- **`NF`**: Número de campos en la línea actual.
- **`FS`**: Separador de campos (por defecto, espacio o tabulación).
- **`OFS`**: Separador de campos de salida (por defecto, espacio).

Conector `&`

El símbolo `&` se utiliza para ejecutar comandos en segundo plano. Cuando pones un `&` al final de un comando, el sistema lo ejecuta en segundo plano, permitiéndote seguir usando la terminal mientras el comando se sigue ejecutando.

sleep 60 &

En este ejemplo, el comando `sleep 60` se ejecutará en segundo plano, lo que significa que puedes seguir usando la terminal para otros comandos mientras `sleep` sigue contando el tiempo en el fondo.

Características:

- **No bloqueante**: La terminal no espera a que el comando termine para que puedas ejecutar otros comandos.
- **Job ID**: Cuando ejecutas un comando con `&`, se te asigna un ID de trabajo (job ID) que puedes usar para controlar el proceso (por ejemplo, para detenerlo o traerlo al primer plano).

Conector `;`

El símbolo `;` se utiliza para ejecutar múltiples comandos en secuencia, uno después del otro, sin importar si el comando anterior tuvo éxito o falló. Cada comando se ejecuta de manera independiente del éxito o fracaso del comando anterior.

comando1 ; comando2 ; comando3

echo "Hola" ; ls -l ; echo "Adiós"

En este ejemplo, primero se ejecuta `echo "Hola"`, luego `ls -l` y, finalmente, `echo "Adiós"`. Cada comando se ejecuta en el orden en que se escriben, y todos se ejecutan independientemente del resultado de los comandos anteriores.

Características:

- **Secuencial**: Los comandos se ejecutan uno tras otro en el orden en que aparecen.
- **Independiente**: No depende del éxito o fracaso de los comandos anteriores.

Diferencias y Usos

- **`&`**: Usa este conector cuando quieras ejecutar un comando en segundo plano y continuar trabajando en la terminal sin esperar a que termine.
- **`;`**: Usa este conector cuando quieras ejecutar múltiples comandos secuencialmente, sin importar si los comandos anteriores tienen éxito o no.

Ejemplo de Uso Combinado

Puedes combinar ambos conectores para ejecutar un comando en segundo plano y luego ejecutar otros comandos en la secuencia:

```
comando1 & comando2 ; comando3
```

Aquí, `comando1` se ejecuta en segundo plano, y luego `comando2` se ejecuta en primer plano, seguido por `comando3` después de que `comando2` ha terminado.

En resumen, `&` es útil para tareas en segundo plano que no necesitan tu atención constante, mientras que `;` es útil para ejecutar una serie de comandos de manera secuencial.

El símbolo de **pipe** (`|`) en sistemas Unix y Linux se utiliza para redirigir la salida de un comando como entrada para otro comando. Este mecanismo permite encadenar varios comandos y crear tuberías de procesamiento de datos de manera eficiente.

¿Qué Hace el Pipe (`|`)?

1. **Redirige la Salida a la Entrada**:
    - La salida estándar (stdout) de un comando se pasa como entrada estándar (stdin) a otro comando. Esto permite realizar operaciones complejas mediante la combinación de comandos simples.
2. **Encadenar Comandos**:
    - Puedes usar pipes para combinar varios comandos y procesar datos de manera secuencial. Esto facilita la creación de comandos más poderosos y flexibles.

comando1 | comando2

En este caso, la salida de `comando1` se convierte en la entrada de `comando2`.

Ejemplos Comunes

1. **Buscar Texto en la Salida de Otro Comando**:
    - Puedes usar `grep` para buscar texto específico en la salida de otro comando.

ls -l | grep "archivo"

Aquí, `ls -l` lista los archivos en el directorio, y `grep "archivo"` filtra las líneas que contienen la palabra "archivo".

Contar el Número de Líneas en la Salida de un Comando:

- Usa `wc -l` para contar el número de líneas en la salida de otro comando.

ps aux | wc -l

Esto cuenta el número de líneas en la salida del comando `ps aux`, que muestra información sobre los procesos en ejecución.

Ordenar y Filtrar Resultados:

- Puedes combinar `sort` y `uniq` para ordenar y eliminar duplicados en los resultados.

cat archivo.txt | sort | uniq

Aquí, `cat` muestra el contenido de `archivo.txt`, `sort` ordena las líneas y `uniq` elimina las duplicadas.

ver el Contenido de un Archivo de Log en Tiempo Real:

- Usa `tail` con `-f` para ver el contenido en tiempo real y `grep` para filtrar resultados específicos.

tail -f /var/log/syslog | grep "error"

Esto muestra en tiempo real las líneas que contienen "error" en el archivo de log `/var/log/syslog`.

Redirección de Salida `>`

El operador `>` se usa para redirigir la salida estándar (stdout) de un comando a un archivo. Si el archivo ya existe, se sobrescribirá con la nueva salida.

comando > archivo

Ejemplos

- **Redirigir la salida de un comando a un archivo**:

echo "Hola Mundo" > archivo.txt

Esto crea el archivo `archivo.txt` (o lo sobrescribe si ya existe) con el texto "Hola Mundo".

Guardar la lista de archivos en un directorio en un archivo:

ls -l > listado.txt

Esto guarda la salida del comando `ls -l` en el archivo `listado.txt`.

Redirección de Entrada `<`

El operador `<` se usa para redirigir la entrada estándar (stdin) desde un archivo. En lugar de leer desde la entrada estándar (como el teclado), el comando lee los datos del archivo especificado.

comando < archivo

Ejemplos

- **Leer el contenido de un archivo como entrada para un comando**:

sort < archivo.txt

Esto toma el contenido de `archivo.txt` y lo pasa como entrada al comando `sort`.

- **Usar un archivo de datos como entrada para un script**:

python script.py < datos.txt

Esto pasa el contenido de `datos.txt` como entrada estándar al script de Python `script.py`.

Redirección de Salida y Entrada Simultáneamente

Puedes usar ambos operadores juntos para redirigir la entrada desde un archivo y la salida a otro archivo.

comando < archivo_entrada > archivo_salida

Ejemplos

- **Ordenar un archivo y guardar el resultado en otro archivo**:

sort < archivo_entrada.txt > archivo_salida.txt

Esto toma el contenido de `archivo_entrada.txt`, lo ordena y guarda el resultado en `archivo_salida.txt`.

Redirección Adicional `>>` y `2>`

- `>>`: Redirige la salida y la agrega al final de un archivo en lugar de sobrescribirlo.

echo "Nueva línea" >> archivo.txt

Esto agrega "Nueva línea" al final del archivo `archivo.txt`.

- `2>`: Redirige la salida de errores estándar (stderr) a un archivo.

comando 2> errores.txt

Esto guarda cualquier mensaje de error producido por `comando` en `errores.txt`.

Resumen

- **`>`**: Redirige la salida estándar a un archivo, sobrescribiendo el archivo si ya existe.
- **`<`**: Redirige la entrada estándar desde un archivo.
- **`>>`**: Redirige la salida estándar a un archivo, agregando al final del archivo en lugar de sobrescribir.
- **`2>`**: Redirige la salida de errores estándar a un archivo.

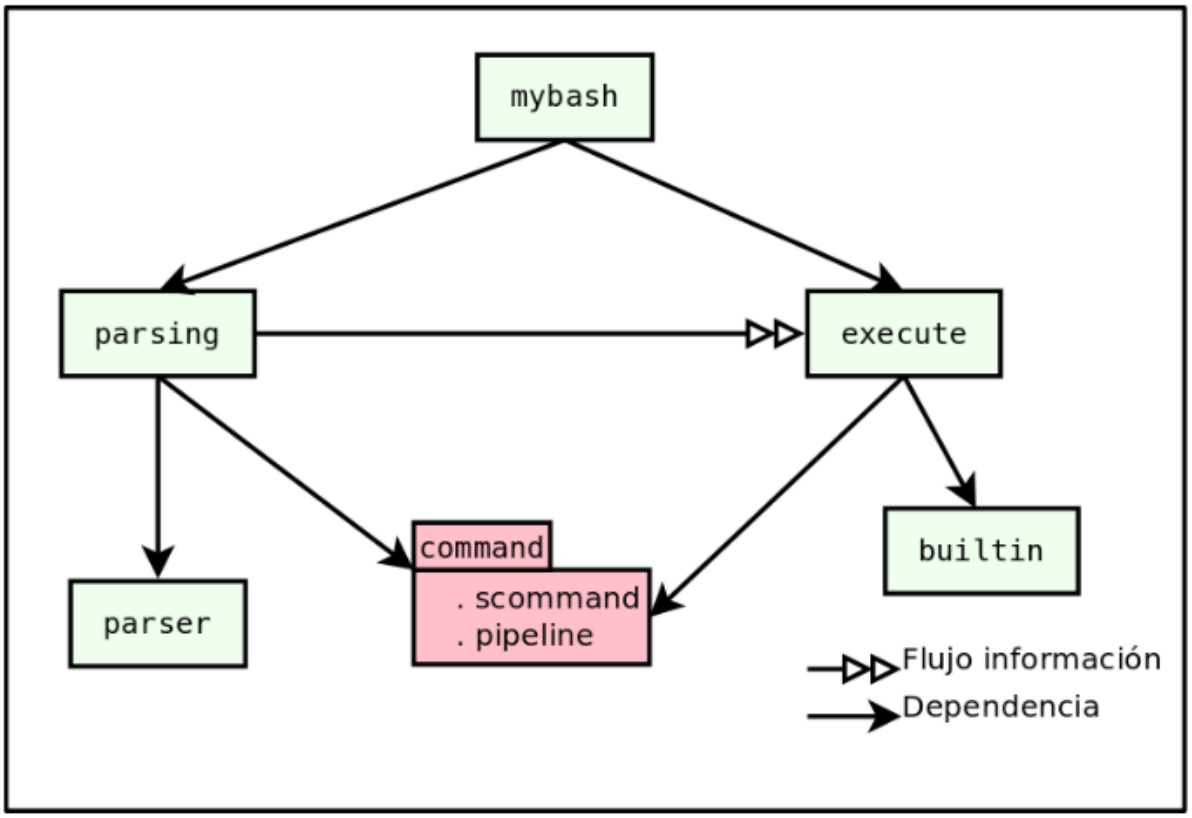
Lab1: (MyBash)

En este laboratorio se logró implementar una simple versión de la línea de comandos GNU Bash (MyBash).

La cual es capaz:

- Procesar y ejecutar comandos simples.
- Procesar y ejecutar múltiples comandos conectados con pipes.
- Ejecutar comandos en modo foreground y background.
- Redirigir entrada y salida a archivos.
- Entre otras cosas…

El proyecto grupal se realizó mediante la implementación de 4 módulos diferentes, a continuación se detalla el funcionamiento de cada uno de una manera resumida, denotando quizás, partes clave de su funcionamiento:



Parsing:

¿Qué hace?

- La función parse_pipeline() dentro de parsing.c toma un TAD parser (generalmente el input del usuario) y lo que hace es leer este input parte por parte para armar y devolver un pipeline.

¿Cómo lo hace?

- Se crea el pipeline a retornar y un scommand que se irá pisando con todos los scommands que se lean del parser. Un bucle se ejecuta mientras no haya ningún error y mientras haya otro pipe para leer del parser, es decir hasta el final del parser. En este bucle se asigna reiteradamente al scommand con la función parse_scommand(parser).
    - La función parse_scommand(parser) lee argumento por argumento hasta el primer símbolo “ | ”, es decir, se retorna una cadena, se especifica de qué tipo es (argumento, argumento de salida o de entrada) y dependiendo de esto, se va pusheando y armando el scommand a retornar. Cuando llegamos al primer “ | ” se corta el bucle habiendo consumido todo lo que estaba por detrás.

- Se quitan los espacios del parser
- Se pushea al pipeline el scommand construido en parse_scommand(parser)

- Luego another_pipe, la variable que nos indica la presencia de otro pipe, se actualiza. Si esta es false se corta el bucle.

Luego (fuera del bucle):

- Si se encuentra un & en el parser, se hace que el pipeline espere.
- Se indica si se encontraron simbolos raros en el parser (garbage).
- Si el pipeline es de length 1 con un scommand vacío se destruye el pipeline.
- Si hay basura se destruye el pipeline.

De esta manera se arma el pipeline a retornar, la magia está en parse_scommand(p) que se encarga de leer argumento por argumento mientras que consume el parser (no entendi bien si se consume o no).

Command:

La gramática que maneja shell se divide en 3 capas: comando simple, tubería y lista, en orden creciente de complejidad. Limitaremos la implementación a los 2 primeros niveles.

- Un comando simple (scommand) es una secuencia de palabras donde la primera es el comando
  y las siguientes sus argumentos. Resultan opcionales dos redirectores, uno de entrada y otro de
  salida.
- Una tubería (pipeline) es una secuencia de comandos simples conectados por el operador pipe
  (que se corresponde con el símbolo '|') y con un terminador opcional que indica si el shell debe
  esperar la terminación del comando antes de permitir ejecutar uno nuevo.

Para los comandos simples se usa una estructura del estilo `([char*],char*,char*)`. Es decir, una lista de punteros a cadenas, la primera corresponde al nombre del comando y el resto a sus argumentos. Luego las dos cadenas restantes hacen referencia al nombre del archivo al cual hay que redirigir la entrada o salida, NULL si no es necesario.
Ejemplo: ls -l ej1.c > out < in ≡ `( [ "ls", "-l" ], "in", "out").`

Luego, para implementar una tubería se utilizará la misma idea. La estructura es del estilo `([scommand], bool)`, es decir, una lista de scommand y un booleano que indica si tenemos que esperar o no. (inicialmente es true, si llega a detectar un & este seria false).

Para la lista de cada uno, se utiliza una implementación de la librería glib.h llamada GSList (SimpleList). No se si es de mucho uso pero su implementación está dadá por:

```
struct GSList {
    gpointer data;
    GSList* next;
    }
```

Cabe aclarar que no se utilizó para nada la implementación interna de GSLIst, se logró total encapsulamiento.

Algunos detalles que me parece importante destacar:

En scommand:

- `void scommand_push_back(scommand self, char * argument);`
    - Esta función agrega por detrás una cadena a la secuencia de cadenas.

- `void scommand_set_redir_in(scommand self, char * filename);`
    - Define la redirección de entrada.

- `void scommand_set_redir_out(scommand self, char * filename);`
    - Define la redirección de salida.

Lo importante de estas 3 funciones es que el TAD asume la responsabilidad de gestionar la memoria que le pasas. Esto incluye la liberación de la memoria dinámica cuando ya no se necesita.

Por último en, `char * scommand_front(const scommand self);` la cual toma la cadena de adelante de la secuencia de cadenas. La cadena retornada sigue siendo propiedad del TAD, y debería considerarse inválida si luego se llaman a modificadores del TAD. Hacer una copia si se necesita una cadena propia.

En pipeline:

- `void pipeline_push_back(pipeline self, scommand sc);`
    - Se apropia del comando.

- `scommand pipeline_front(const pipeline self);`
    - El comando devuelto sigue siendo propiedad del pipeline.

Builtin:

¿Qué hace?

- Detecta si un scommand contiene un comando interno y si es así, lo efectúa.
- Además se verifica si un pipeline contiene sólo un comando el cual además es un comando interno.
- Se contemplan solo tres comandos internos: cd, help y exit

¿Cómo lo hace?

Este módulo se divide en 3:

- `bool builtin_alone(pipeline p)` indica si es un comando interno solo.
    - Si el length del pipeline p es 1 y además su único elemento es un comando interno, retorna true.

- `bool builtin_is_internal(scommand cmd)` indica si es un comando interno.
    - compara cmd con los 3 comandos posibles, si se cumple para alguno retorna true.

- `void builtin_run(scommand cmd)` ejecuta el comando interno.
    - Se divide en 3 casos
        - caso 0: cmd = “cd”
            - Si la longitud de cmd es 1, esto implica que no hay argumentos para cambiar de directorio, se arma el path “/home/user” utilizando getlogin(), asignando memoria para el path y con strcpy y strcat se consigue llamar a chdir() con el path.
            - Si la longitud es mayor a 1 simplemente se llama a chdir con el cmd
            - Luego se manejan los posibles errores si chdir() devuelve un entero distinto de 0.
        - caso 1: cmd = “help”
            - Es un printf
        - caso 2: cmd = “exit”
            - exit(EXIT_SUCCESS);

Execute:

Este módulo se encarga de ejecutar un pipeline mediante el uso de fork(), execvp(), dup() entre otras funciones.

¿Qué hace?

- El algoritmo toma un pipeline de longitud n, se recorre el pipeline de izquierda a derecha y en cada iteración se extrae un scommand. Cada scommand será ejecutado redirigiendo o no la salida al siguiente scommand y así sucesivamente.

¿Cómo lo hace?

- Vamos a definir una “traza” del código para no perdernos en la explicación:
    - Si es un comando interno solo: Se ejecuta y termina.
    - Caso contrario: Se ejecuta un bucle en el siguiente orden.
        - Se guarda la información del pipe anterior o no (depende el caso).
        - Se ejecuta un nuevo pipe o no (depende el caso).
        - fork().

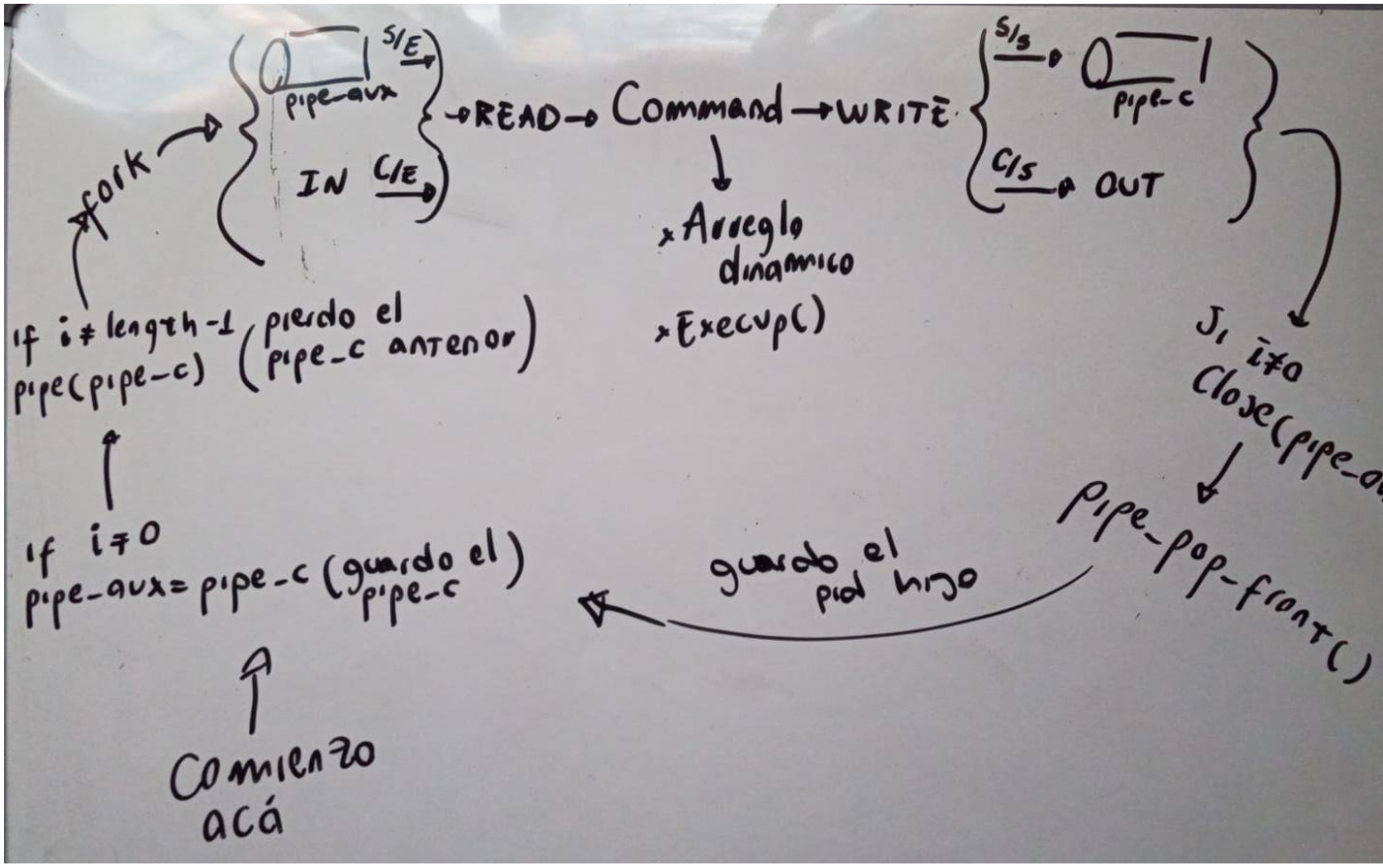
        - Consulto si redirijo la entrada y salida a los diferentes pipes o a archivos externos.
    - Ejecuto el comando.
    - Cierro el pipe auxiliar.
    - Guardo el pid del hijo.
    - Paso al siguiente comando.

En primer lugar, vamos a hacer uso de dos pipes, uno que guarde la información que nos tira un comando, y otro que apunte a esa información para luego leerla ya que el primer pipe será reutilizado para volcar nueva información. (Perdemos referencia al pipe_actual)

pipe_auxiliar -(leo)-> COMANDO -(escupo)-> pipe_actual

pipe_auxiliar = pipe_actual

reutilizamos pipe_actual



Luego de establecer si vamos a utilizar un pipe o no, se crea un hijo con fork().
El mismo se encarga de consultar dos cosas:

- Si no está en el primer comando, redirige la entrada cerrando el stdin y duplicando el pipe_auxiliar.
- Si no está en el último comando, redirige la salida cerrando el stdout y duplicando el pipe_actual.

También se encarga de ver si la salida o entrada se toma de < o > de la misma manera, pero utilizando open() para los archivos.

Ahora se ejecuta el comando, como la función execvp() necesita el nombre del comando para buscar el ejecutable y además un arreglo con los argumentos, se define una función auxiliar. En esta misma se copia cada cadena dentro de scommand y se guarda en un arreglo dinámico, si o si tiene que ser un arreglo dinámico porque si hiciéramos front y pop_front de command sin duplicar en definitiva estamos eliminando la memoria que, primero no nos pertenece y segundo perdemos lo que asignamos.

Una vez obtenido el arreglo con el nombre y los argumentos llamamos a execvp() y la memoria del arreglo se libera cuando termina el proceso.

Ahora estamos en el padre, si no estamos en el primer comando, cerramos el pipe_auxiliar para evitar errores. Guardamos el pid del hijo en un arreglo dinámico y hacemos pop_front del pipeline repitiendo este proceso para el siguiente.

Al salir del bucle el padre consultó si había que esperar o no al pipeline, si es así, se ejecuta un bucle que busca mandar la señal de n waitpid() para indicarle al padre que tiene que esperar a esos hijos. Caso contrario no se hace un waitpid() de nada. En este caso quedan procesos zombies dando vueltas los cuales se eliminan mandando una señal previamente.