UNIVERSIDAD DEL BÍO-BÍO
DEPARTAMENTO DE SISTEMAS DE INFORMACIÓN
Sistemas Operativos
Laboratorio 2
MiniShell

PARTE A: Operadores de Control

Los operadores de control en basj son símbolos que realizan una función de control. La siguiente table ofrece un resumen de los operadores de control.

Operador	Descripción
`&`	Envía un proceso al fondo (background) para que se ejecute de forma asincrónica.
`&&`	Se utiliza como un operador lógico AND. El segundo comando solo se ejecuta si el primero retorna un valor de salida exitoso.
`( and )`	Agrupa comandos para que se ejecuten como una sola unidad o subshell.
`;`	Actúa como un delimitador de comandos. El comando siguiente se ejecutará después de que el anterior haya finalizado, sin importar el estado de salida del primero.
`;;`	Finaliza una cláusula en una instrucción case.
`|`	Redirige la salida estándar de un comando como entrada estándar para otro comando (pipe).
`||`	Se utiliza como un operador lógico OR. El segundo comando solo se ejecuta si el primero retorna un estado de salida distinto de 0 (falla).

1. & (5pts)

Veamos algunos de estos operadores de control en acción. El operador & envía un comando al fondo (background). Crea un archivo con el siguiente contenido, ejecútalo y explica por qué sucede eso y como lo puedes solucionar.

```
#!/bin/bash

# This script will send the sleep command to the background.
echo "Sleeping for 10 seconds..."
sleep 10 &

# Creates a file
```

```
echo "Creating the file test123"
touch test123

# Deletes a file
echo "Deleting the file test123"
rm test123
```

```
viasmussen@fdd484fa12b6:~$ sleep 10 &
[1] 24416
viasmussen@fdd484fa12b6:~$ touch test123
[1]+  Done                    sleep 10
viasmussen@fdd484fa12b6:~$ rm test123
viasmussen@fdd484fa12b6:~$
```

Lo que hace este código es detener todo por 10 segundos pero en background por lo que permite seguir ejecutando códigos sin esperar los 10 segundos, como en este caso crear un archivo y borrarlo luego, para solucionarlo solo habría que borrar el “&” y así no permitiría ejecutar nada durante el sleep.

2. &&, (Sequence ; ;;) || (5pts)

Los comandos que tardan mucho en ejecutarse suelen enviarse a segundo plano para evitar que los scripts se bloqueen. El operador && nos permite realizar una operación lógica AND entre dos comandos. En el siguiente ejemplo, el archivo test123 se creará solo si el primer comando se ejecuta con éxito:

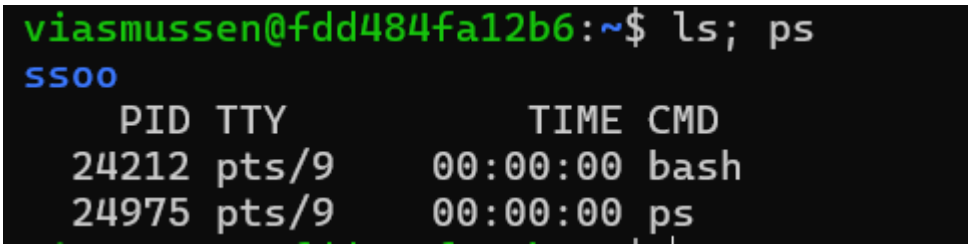
`touch test && touch test123`

El operador () nos permite agrupar comandos para que actúen como una sola unidad cuando necesitamos redirigirlos en conjunto:

```
viasmussen@fdd484fa12b6:~$ touch test && touch test123
viasmussen@fdd484fa12b6:~$ ls
ssoo  test  test123
viasmussen@fdd484fa12b6:~$ rm test && rm test123
viasmussen@fdd484fa12b6:~$ ls
ssoo
```

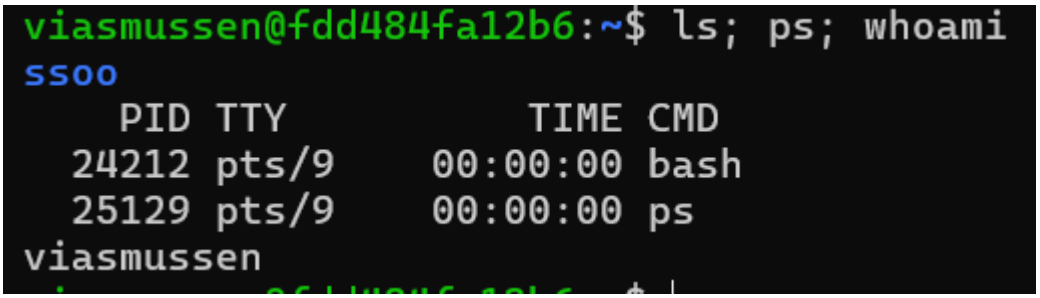
`(ls; ps)`

Esto generalmente es útil cuando necesitas redirigir los resultados de varios comandos a un flujo (stream). El operador ; nos permite ejecutar múltiples comandos sin importar su estado de salida.



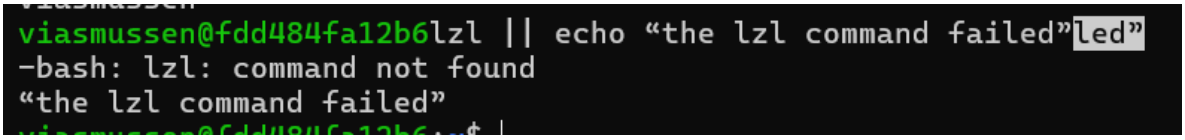
`ls; ps; whoami`

como resultado, cada comando se ejecuta uno tras otro tan pronto como el anterior termina. El operador || nos permite encadenar comandos utilizando una operación lógica OR:



`lzl || echo "the lzl command failed"`

En este ejemplo, el comando echo se ejecutará solo si el primer comando falla.



3. Pipe (5pts)

El comando pipe combina dos o más comandos para redirigir sus entradas y salidas.
Generalmente, la sintaxis para pipe es:

```
command 1 | command 2 | command 3 |...
```

Donde la salida del command 1 se utiliza como entrada para command 2. A su vez, la salida de command 2 se utiliza como entrada para el command 3, y así sucesivamente. Por ejemplo, supongamos que tenemos un archivo llamado name.txt que contiene nombres de estudiantes:

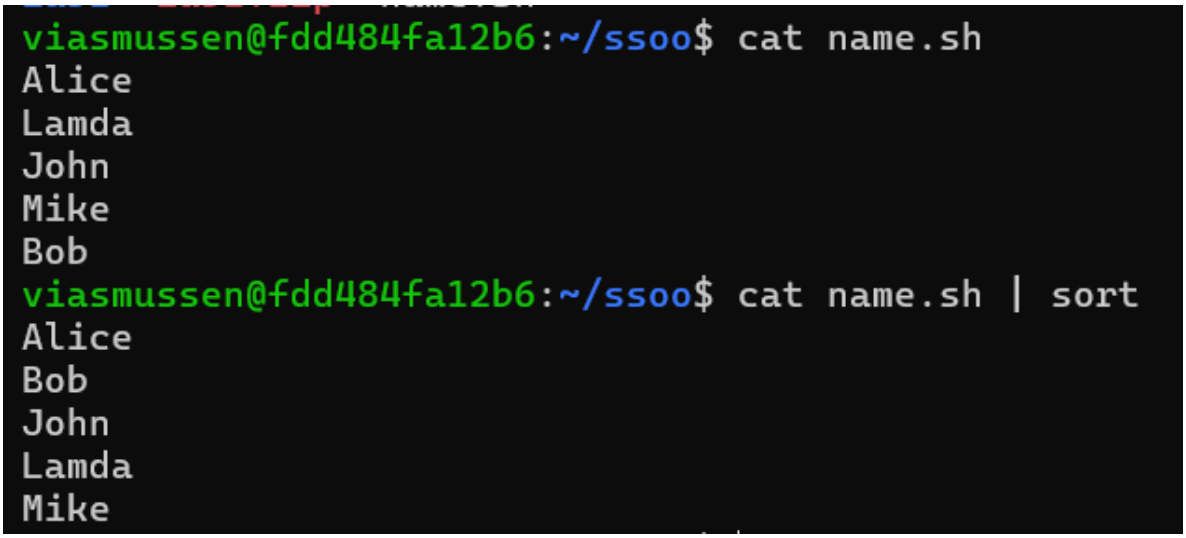
```
ubb-lab@user:~$ cat name.txt
Alice
Lamda
John
Mike
Bob
```

Entonces, para ordenar este archivo, usaremos `pipe`:

```
ubb-lab@user:~$ cat name.txt | sort
Alice
```

```
Bob
John
Lamda
Mike
```

Donde la salida del comando `cat` se alimenta del comando `sort`.



4. Redirection (5pts)

La redirección consiste en tomar la salida de un comando o script y usarla como entrada para otro script o archivo con fines de escritura. La siguiente tabla describe los operadores de redirección disponibles.

Operador	Descripción
`>`	Redirige stdout a un archivo.
`>>`	Redirige stdout a un archivo, añadiendo el contenido a lo existente.
`&> o >&`	Redirige stdout y stderr a un archivo.
`&>>`	Redirige stdout y stderr a un archivo, añadiéndolos al contenido existente.
`<`	Redirige la entrada a un comando.
`<<`	Llamado documento here (heredoc), redirige múltiples líneas de entrada a un comando.
`|`	Redirige la salida de un comando como entrada para otro comando (pipe).

Vamos a practicar el uso de los operadores de redirección para ver como funcionan los flujos estándar. El operador > redirige el flujo de salida estándar (stdout) a un archivo. Cualquier comando que preceda a este carácter enviará su salida a la ubicación especificada. Ejecuta el siguiente comando directamente en tu terminal:

`echo "Hello World!" > output.txt`

redirigimos el flujo de salida estándar (stdout) a un archivo llamado outpu.txt. Para ver el contenido de output.txt, simplemente ejecuta lo siguiente:

`cat output.txt`

```
viasmussen@fdd484fa12b6:~$ echo "Hello World!" > output.txt
viasmussen@fdd484fa12b6:~$ cat output.txt
Hello World!
```

A continuación, utilizaremos el operador >> para añadir contenido al final del mismo archivo.

`echo "Goodbye!" >> output.txt`

`cat output.txt`

```
viasmussen@fdd484fa12b6:~$ echo "Goodbye!" >> output.txt
viasmussen@fdd484fa12b6:~$ cat output.txt
Hello World!
Goodbye!
```

Si hubiéramos usado > en lugar de >>, el contenido de output.txt habría sido sobrescrito completamente con el texto “Goodbye!”.
Puedes redirigir tanto el flujo de salida estándar (stdout) como el flujo de error estándar (stderr) a un archivo utilizando &>. Esto es útil cuando no deseas enviar ninguna salida a la pantalla y prefieres guardar todo en un archivo de registro (logfile) para un análisis posterior:

`ls -l / &> stdout_and_stderr.txt`

```
viasmussen@fdd484fa12b6:~$ ls -l / &> stdout_and_stderr.txt
viasmussen@fdd484fa12b6:~$ ls
output.txt  ssoo  stdout_and_stderr.txt
```

```
viasmussen@fdd484fa12b6:~$ cat stdout_and_stderr.txt
total 64
lrwxrwxrwx    1 root root    7 Jul 30 02:11 bin -> usr/bin
drwxr-xr-x    2 root root 4096 Apr 18  2022 boot
drwxr-xr-x    5 root root  360 Aug 28 21:20 dev
drwxr-xr-x    1 root root 4096 Sep 23 13:29 etc
drwx--x--x    1 root root 4096 Sep 12 00:27 home
lrwxrwxrwx    1 root root    7 Jul 30 02:11 lib -> usr/lib
lrwxrwxrwx    1 root root    9 Jul 30 02:11 lib32 -> usr/lib32
lrwxrwxrwx    1 root root    9 Jul 30 02:11 lib64 -> usr/lib64
lrwxrwxrwx    1 root root   10 Jul 30 02:11 libx32 -> usr/libx32
drwxr-xr-x    2 root root 4096 Jul 30 02:11 media
drwxr-xr-x    2 root root 4096 Jul 30 02:11 mnt
drwxr-xr-x    2 root root 4096 Jul 30 02:11 opt
dr-xr-xr-x 1578 root root    0 Aug 28 21:20 proc
drwx------    1 root root 4096 Sep  9 11:30 root
drwxr-xr-x    1 root root 4096 Sep 23 13:25 run
lrwxrwxrwx    1 root root    8 Jul 30 02:11 sbin -> usr/sbin
drwxr-xr-x    2 root root 4096 Jul 30 02:11 srv
dr-xr-xr-x   13 root root    0 Aug 28 21:20 sys
drwxrwxrwt    1 root root 4096 Sep 23 13:34 tmp
drwxr-xr-x    1 root root 4096 Jul 30 02:11 usr
drwxr-xr-x    1 root root 4096 Jul 30 02:19 var
```

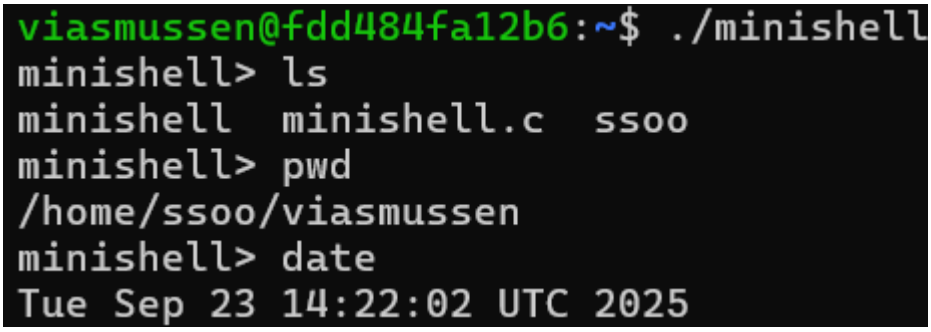
PARTE B: MINISHELL

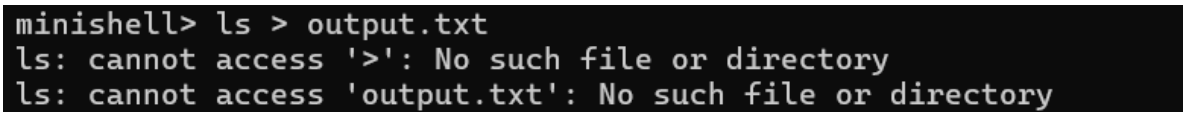
En esta parte implementarás un minishell el cual es una versión simplificada de un intérprete de comandos. Este Minishell aceptará comandos del usuario, los ejecutará y gestionará la redirección de entrada/salida y la ejecución de comandos en segundo plano.

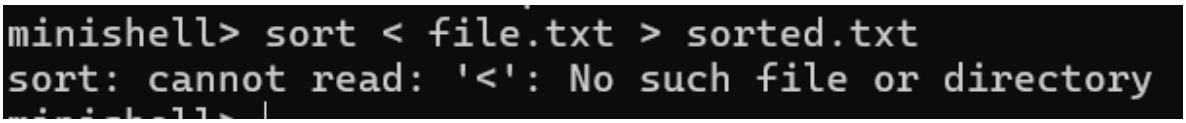
Crea un archivo 'minishell.c' en tu directorio de trabajo.

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

#define MAX_LINE 80 /* The maximum length command */

void parseCommand(char *input, char **args) {
    int i = 0;
    args[i] = strtok(input, " \n");
    while (args[i] != NULL) {
        i++;
        args[i] = strtok(NULL, " \n");
    }
}

int main(void) {
    char *args[MAX_LINE / 2 + 1]; /* command line arguments */
    char input[MAX_LINE];
    int should_run = 1; /* flag to determine when to exit program */

    while (should_run) {
        printf("minishell> ");
        fflush(stdout);

        if (fgets(input, MAX_LINE, stdin) == NULL) {
            perror("fgets failed");
            exit(1);
        }

        if (strncmp(input, "exit", 4) == 0) {
            should_run = 0;
            continue;
        }

        parseCommand(input, args);

        pid_t pid = fork();
        if (pid < 0) {
            perror("Fork failed");
            exit(1);
        } else if (pid == 0) {
            if (execvp(args[0], args) < 0) {
                perror("execvp failed");
            }
            exit(1);
        } else {
            wait(NULL);
        }
    }
    return 0;
}
```

1. **Ejecuta comandos básicos como** ‘`ls`’, ’`pwd`’, ’`date`’**. Luego prueba la redirección de entrada y salida** ‘`ls > output.txt`’, ‘`sort < file.txt > sorted.txt`’ **y ejecuta comandos en segundo plano** ‘`sleep 10 &`’**. Explica lo sucedido. (15pts)**





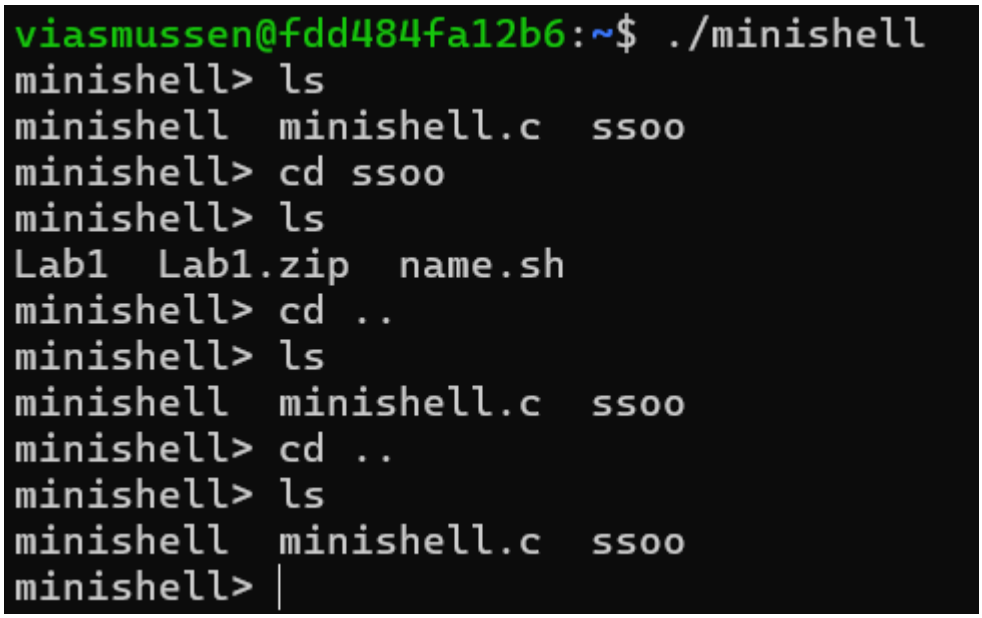


El minishell no permite o no soporta ejecuciones de segundo plano, pipes o redirecciones de archivos como lo son |, <, > o & por la forma que está hecho el código es una simulación de Shell por lo cual es básico dentro de sus posibilidades

2. **Investiga y añade soporte para el comando ‘**`cd`**’. Copia tu código completo aquí. (40pts)**

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

#define MAX_LINE 80 /* The maximum length command */

void parseCommand(char *input, char **args) {
    int i = 0;
    args[i] = strtok(input, " \n");
    while (args[i] != NULL) {
        i++;
        args[i] = strtok(NULL, " \n");
    }
}

int main(void) {
    char *args[MAX_LINE / 2 + 1]; /* command line arguments */
    char input[MAX_LINE];
    int should_run = 1; /* flag to determine when to exit program */

    while (should_run) {
        printf("minishell> ");
        fflush(stdout);

        if (fgets(input, MAX_LINE, stdin) == NULL) {
```

```
            perror("fgets failed");
            exit(1);
        }

        if (strncmp(input, "exit", 4) == 0) {
            should_run = 0;
            continue;
        }

        parseCommand(input, args);

        if (args[0] == NULL) {
            continue; // no ingresó nada va a home en cd
        }

        // Manejar comando "cd"
        if (strcmp(args[0], "cd") == 0) {
            char *dir = args[1];

            if (dir == NULL || strcmp(dir, "..") == 0) {
                dir = getenv("HOME"); // ir a HOME
            }

            if (chdir(dir) != 0) {
                perror("cd falló");
            }
            continue; // no crear proceso hijo
        }

        pid_t pid = fork();
        if (pid < 0) {
            perror("Fork failed");
            exit(1);
        } else if (pid == 0) {
            if (execvp(args[0], args) < 0) {
                perror("execvp failed");
            }
            exit(1);
        } else {
            wait(NULL);
        }
    }
    return 0;
```



Si bien el código permite volver a home no se vee el /home

ENTREGA (5pts)

Documente (screenshots) cada una de las acciones antes señaladas. Es de exclusiva responsabilidad del estudiante respetar el formato de entrega de informe de esta guía **(debajo de cada enunciado su screenshot en donde aparezca de forma clara las sentencias utilizadas)**. El formato de entrega debe ser en PDF, y el nombre del archivo debe contener su nombre y apellido **(Laboratorio_2_Nombre_Apellido).** Todas las actividades deben ser entregadas (subidas) a la plataforma digital Moodle en los plazos establecidos, por cada hora de atraso, se descontará 1 pto.