Explicacion de codigo

1. Estructura General

El código implementa un shell básico con soporte para **un solo pipe (`|`)**. Se mantiene en un ciclo infinito (`while(1)`) hasta que el usuario ingresa `exit`.

2. Captura del Comando

El usuario ingresa un comando en la terminal. Se almacena en la variable comando, eliminando el salto de línea con:

comando[strcspn(comando, "\n")] = '\0'; // Eliminar el salto de línea

Ejemplo de comando ingresado:

comando: ls | grep .c

Este comando usa un **pipe (`|`)** para conectar la salida de **`ls`** con la entrada de **`grep .c`.**

3. Cálculo de Tuberías (`|`)

Se utiliza la función **`totalTuberias(const char *comando)`** para contar cuántos pipes (`|`) existen en el comando. Esto es necesario para determinar cuántos segmentos debe procesar `execvp()`.

Ejemplo de representación del comando como array:
[0] = l
[1] = s
[2] = (espacio)
[3] = | → **posición de la tubería**
[4] = (espacio)
[5] = g
[6] = r
[7] = e
[8] = p
[9] = (espacio)
[10] = .
[11] = c
[12] = \0 → **fin de cadena**

El código recorre la cadena, aumentando `i` cuando encuentra `|`:

```
int totalTuberias(const char *comando) {
    int i = 0;
    for (; *comando; comando++)
        if (*comando == '|') i++;
    return i;
}
```

En este caso, `totalTuberias(comando)` retorna **1**, indicando un solo pipe.

4. División del Comando en Segmentos

Cada parte del comando antes y después del `|` se almacena en `segmentos[]`:

```
int size = totalTuberias(comando) + 1; // Sumamos 1 para tener dos segmentos

char *segmentos[size]; // Creamos los segmentos

int i = 1;
segmentos[0] = strtok(comando, "|");
while (i < size) {
    segmentos[i] = strtok(NULL, "|");
    i++;
}
```

El ciclo separa:

- **`segmentos[0] = "ls"`**
- **`segmentos[1] = "grep .c"`**

Cuando `i == size`, el ciclo se rompe.

5. Explicación de Pipes (`pipe()`)

Los **pipes** permiten la comunicación entre procesos en C. Se pueden usar de dos formas:

- **Lectura:** `STDIN_FILENO`
- **Escritura:** `STDOUT_FILENO`

Primero se crea la tubería con:

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

Donde:

- `pipefd[0]` → extremo de **lectura**.
- `pipefd[1]` → extremo de **escritura**.

Para que un proceso hijo pase información a otro:

Caso 1: Pasar información de hijo a padre

```
// Proceso hijo
close(fd[1]); // Cerramos extremo de escritura
dup2(fd[0], STDIN_FILENO); // Redirigir lectura de la tubería
close(fd[0]); // Cerramos extremo de lectura


// Proceso padre
close(fd[0]); // Cerramos extremo de lectura
dup2(fd[1], STDOUT_FILENO); // Redirigir escritura en la tubería
close(fd[1]); // Cerramos extremo de escritura
```

Caso 2: Escribir desde hijo y pasar al padre (como en nuestro código)

```
// Proceso hijo
close(fd[0]); // Cerramos extremo de lectura
dup2(fd[1], STDOUT_FILENO); // Escribir en la tubería
close(fd[1]); // Cerramos extremo de escritura


// Proceso padre
close(fd[1]); // Cerramos extremo de escritura
dup2(fd[0], STDIN_FILENO); // Leer desde la tubería
close(fd[0]); // Cerramos extremo de lectura
```

Este mecanismo se usa dentro de `ejecutarComando()` para gestionar la comunicación.

6. Creación de Procesos y Asignación de Entrada/Salida

Se utiliza un `for` para procesar los segmentos del comando. Las condiciones definen cómo se conectan los procesos:

```
for (i = 0; i < size; i++) {
    int entrada = (i == 0) ? STDIN_FILENO : pipefd[0]; // Si es el primer segmento, entrada es estándar
    int salida = (i == size - 1) ? STDOUT_FILENO : pipefd[1]; // Si es el último segmento, salida es estándar

    ejecutarComando(segmentos[i], entrada, salida);
```

```
    if (entrada != STDIN_FILENO) close(entrada);
    if (salida != STDOUT_FILENO) close(salida);
}
```

Aquí:

- **Primer segmento (`ls`)** escribe en la tubería.
- **Segundo segmento (`grep .c`)** lee desde la tubería.

Después de ejecutar los comandos, `wait(NULL)` espera la finalización de los procesos.

7. Implementación de `ejecutarComando()`

Cada proceso hijo:

1. **Redirige la entrada/salida con `dup2()`.**
2. **Tokeniza el comando (`strtok()`).**
3. **Ejecuta el comando con `execvp()`.**

```
void ejecutarComando(char *comando, int entrada, int salida) {
    pid_t pid = fork();
    if (pid == -1) {
        perror("Error al crear el proceso hijo");
    } else if (pid == 0) {
        if (entrada != STDIN_FILENO) {
            dup2(entrada, STDIN_FILENO);
            close(entrada);
        }

        if (salida != STDOUT_FILENO) {
            dup2(salida, STDOUT_FILENO);
            close(salida);
        }

        char *args[TAM];
        int i = 0;
        char *token = strtok(comando, " ");
        while (token != NULL) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }

        args[i] = NULL;
        execvp(args[0], args);
        perror("Error ejecutando el comando");
        exit(EXIT_FAILURE);
```

```
    }
}
```

8. Ejecución de los Segmentos

Para ejecutar los comandos con **pipes**, cada segmento debe procesarse en un **proceso hijo**. **La función `ejecutarComando()` maneja esto, configurando correctamente la entrada y salida de datos.**

1. Primera ejecución: `ls`

El primer segmento (`ls`) no necesita leer datos de otro proceso, solo debe **escribir** su salida en la tubería (`pipefd[1]`).

Configuración de entrada/salida en `ejecutarComando()`

```
ejecutarComando(segmentos[0], STDIN_FILENO, pipefd[1]);
```

Aquí:

- **Entrada: `STDIN_FILENO` (es la terminal).**
- **Salida: `pipefd[1]` (escritura en la tubería para que `grep .c` lo use).**

Paso 1: Verificar si se cambia la entrada

```
if (entrada != STDIN_FILENO) {
    dup2(entrada, STDIN_FILENO); // No se ejecuta en este caso
    close(entrada);
}
```

Como `entrada == STDIN_FILENO`, **este bloque no se ejecuta**.

Paso 2: Configurar la salida

```
if (salida != STDOUT_FILENO) {
    dup2(salida, STDOUT_FILENO);
    close(salida);
}
```

Aquí **sí se ejecuta**, porque `salida == pipefd[1]`.

- **`dup2(pipefd[1], STDOUT_FILENO);` redirige la salida de `ls` hacia la tubería.**
- **`close(pipefd[1]);` cierra el extremo de escritura para evitar bloqueos.**

Paso 3: Tokenización del comando (`ls`)

Se separa en argumentos para `execvp()`:

```
char *args[TAM];
int i = 0;
char *token = strtok(comando, " ");
while (token != NULL) {
    args[i++] = token;
    token = strtok(NULL, " ");
}
args[i] = NULL;
```

Ahora `args[0] = "ls"` y `execvp()` ejecuta el comando:

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

La salida de `ls` **se envía a la tubería**, esperando ser leída por `grep .c`.

2. Segunda ejecución: `grep .c`

El segundo segmento (`grep .c`) **lee** la salida de `ls` desde `pipefd[0]` y la filtra.

Configuración de entrada/salida en `ejecutarComando()`

```
ejecutarComando(segmentos[1], pipefd[0], STDOUT_FILENO);
```

Aquí:

- **Entrada:** `pipefd[0]` (lee datos de `ls`).
- **Salida:** `STDOUT_FILENO` (imprime el resultado filtrado en pantalla).

Paso 1: Verificar si se cambia la entrada

```
if (entrada != STDIN_FILENO) {
    dup2(entrada, STDIN_FILENO);
    close(entrada);
}
```

Aquí **sí se ejecuta**, porque `entrada == pipefd[0]`.

- **`dup2(pipefd[0], STDIN_FILENO);` redirige la entrada de `grep .c` a la tubería.**
- **`close(pipefd[0]);` cierra el extremo de lectura.**

Paso 2: Verificar si se cambia la salida

```
if (salida != STDOUT_FILENO) {
    dup2(salida, STDOUT_FILENO);
    close(salida);
}
```

No se ejecuta, porque `salida == STDOUT_FILENO`, por lo que la salida de `grep .c` se muestra en pantalla.

Paso 3: Tokenización del comando (`grep .c`)

Al igual que con `ls`, se separa en argumentos para `execvp()`:

```
char *args[TAM];
int i = 0;
char *token = strtok(comando, " ");
while (token != NULL) {
    args[i++] = token;
    token = strtok(NULL, " ");
}
args[i] = NULL;
```

Ahora `args[0] = "grep"` y `args[1] = ".c"`.

Paso 4: Ejecutar `grep .c`

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

Aquí:

- `grep .c` **lee la salida de** `ls` desde `pipefd[0]`.
- **Filtra los archivos terminados en** `.c`.
- **Imprime el resultado en pantalla**.

Explicación Completa del Flujo

1. **Primer proceso (`ls`):**
    - Lee desde la terminal.
    - Escribe la salida en `pipefd[1]`.
2. **Segundo proceso (`grep .c`):**
    - Lee la salida de `ls` desde `pipefd[0]`.
    - Filtra los archivos `.c`.
    - Imprime el resultado en pantalla.

Conclusión

Este código permite **encadenar dos comandos** con un **único pipe (`|`)** en C. Cada proceso hijo **redirige su entrada/salida** correctamente y ejecuta su segmento con `execvp()`.