Proyecto 1 - Sistemas Operativos

María Castillo Acuña, Francy Jelvez Jen, Pedro Palacios Rossi, Diego Venegas Anabalón

September 10, 2024

1 Introducción

A continuación se presentan detalles sobre nuestro proyecto, un intérprete de comandos simples de Linux, "Shellcito". Para compilarlo, se usa g++ -o Shellcito Shell.cpp. El repositorio de nuestro proyecto se encuentra en este enlace.

2 Implementación

2.1 Parte 1

1. La shell proporciona un prompt que identifica el modo de espera de comandos. En el código, esto se implementa en la función leercomando():

```
// Lee los comandos ingresados por el usuario.
void leer_comando() {
 while (condicion) {
  // prompt.
  cout << "\033[34m Shellcito\033[0m:\033[1;33m~\033[0m$ ";
```

   Donde muestra el prompt personalizado Shellcito con los colores azul y amarillo.

```
 Shellcito:~$
```

   Esto es un indicativo de que la shell está lista para recibir un comando.

2. Para parsear la entrada se usa la función:

   - Lectura del comando: Esto se realiza con la instrucción scanf en leer comando()
   - Parseo de comandos y argumentos: La función parsear comandos(char *comando) recibe el comando y lo divide en tokens (comando y sus argumentos). Esta función utiliza strtok() para dividir el comando por espacios, almacenando cada argumento en un arreglo de strings.

3. Las llamadas a fork() ocurren en set recordatorio y leer, donde cumplen unas funciones similares: crean un nuevo proceso según lo que el comando dicte. Las llamadas a exec() ocurren en la función de leer, en distintos casos:

   - En caso que no haya pipes y el fork() retorne 0, se usa execvp con los parámetros del comando, con los que remueve y reemplaza el proceso actual
   - En caso que hayan multiples pipes y fork() retorne 0, se usa para cerrar el proceso de reemplazo que inició fork() y reemplaza al proceso actual

4. Cuando se presiona enter sin ingresar un comando, la función `leer_comando()` detecta que el largo del comando ingresado es 0 y en respuesta, se ejecuta "continue", lo que ocasiona que el bucle while comience de nuevo y se imprima el prompt nuevamente. Este requirimiento está implementado en el siguiente fragmento del código:

```
 if (scanf("%m[^\n]%*c", &comando) == 0) {
scanf("%*c");
continue;
 }
```

5. Para soportar comandos con presencia de pipes se realizó el siguiente procedimiento:

   (a) Se hizo el conteo de la cantidad de pipes que tenía el comando con la función `contar_pipes()` y se guardó en la variable n

   (b) Después de realizar el parseado del comando utilizando el espacio como separador y almacenar los argumentos en una lista, con la función `parsear_pipes()` que recibe la lista de argumentos y cantidad de pipes que contiene el comando, se dividió la lista de argumentos en subcomandos basados en la ubicación de los pipes

   (c) Por último la creación de pipes para cada par de subcomandos que se comunican entre si, así como la creación de procesos y redirección de entradas y salidas para ejecutar cada subcomando, se realiza en la siguiente fracción del código

```
 // Se crea un espacio para todos los file descriptors.
int FDs[2 * n];
// Se crean los procesos hijos dependiendo de la cantidad de pipes.
for (int i = 0; i <= n; i++) {
 if (i < n) {
  pipe(FDs + 2 * i); // se abre una pipe en todas las iteraciones menos en la ultima.
 }
 if (fork() == 0) { // Proceso hijo.

  // redirección de la entrada (se omite el primer comando)
  if (i > 0) {
   dup2(FDs[2 * i - 2], STDIN_FILENO);
   close(FDs[2 * i - 2]);
   close(FDs[2 * i - 1]);
  }

  // redirección de la salida (se omite el último comando)
  if (i < n) {
   dup2(FDs[2 * i + 1], STDOUT_FILENO);
   close(FDs[2 * i]);
   close(FDs[2 * i + 1]);
  }

  // Ejecutar comando.
  execvp(comandos[i][0], comandos[i]);

  // En caso de que la ejecución falle.
  favs.remove_fav(comandos[i][0]);
  cout << "Comando no encontrado" << endl;
  return;
 }
 if (i > 0) {
  // cerrar los file descriptors que no se usan.
  close(FDs[2 * i - 2]);
  close(FDs[2 * i - 1]);
 }
 wait(NULL); // Esperar a que termine el proceso hijo.
}
```

6. Nuestra Shell soporta el comando exit, que permite salir del shell y terminar su ejecución. Cuando se ingresa 'exit' como comando, se activa este caso:

```
if (strcmp(comando, "exit") == 0) {
exit();
break;
}
```

Esto activa a la función exit()

```
void exit() {
 printf("\033[31m Saliendo de Shellcito\033[1;0m\n");
 sleep(1);
 system("clear");
}
```

Que imprime un mensaje de salida y limpia la pantalla, cerrando la shell.

7. Cuando un comando no existe, se imprime un mensaje de error. Esto se maneja en el proceso hijo donde se intenta ejecutar el comando:

```
execvp(comandos[i][0], comandos[i]);
cout << "Comando no encontrado" << endl;
```

Si execvp() no puede encontrar el comando, muestra un mensaje indicando que el comando no fue encontrado.

Es importante destacar que nuestra implementación solo contempla comandos que se encuentran en /usr/bin/, es decir, no funcionan ciertos comando "built in" de la shell como "cd" u otros.

2.2 Parte 2

Para los comandos de favs, se usa el archivo favs.hpp con las siguientes funciones:

- Para 'favs crear ruta/misfavoritos.txt': Se usa void crear(string ruta), donde se crea un archivo en la ruta, donde van a estar los favoritos del usuario
- Para 'favs mostrar': Se usa void mostrar(), donde se ocupa primariamente void `print_favs(int index)` en donde se imprime el índice y respectivo comando de cada elemento del archivo.
- Para 'favs eliminar num1, num2': Se usa `void remove_fav(int i1, int i2)` quelimina los comandos del vector favs en el rango dado.
- Para 'favs buscar cmd' : Se usa `int search_favs(const std::string &name)`, donde se busca un favorito que coincida con el input.
- Para 'favs borrar' : Se llama al método `favs.remove_favs()` que elimina los elementos almacenados en meemoria del vector favs usando fav.clear().
- Para 'favs num ejecutar' : Se usan dos funciones, `string get_fav(int index)` para recuperar el comando del archivo de favoritos usando el index, y `ejecutar_comando(char **args, Fav favs)` que maneja los subcomandos de favs relacionados con la creación, visualización, eliminación, búsqueda, carga y guardado de comandos favoritos.
- Para 'favs cargar' : Se usa `void load_favs()`, que primero verifica que se haya creado una ruta para el archiv y luego lo abre para alamacenar cada comando en el vector favs y así se mantenerlos en memoria.
- Para 'favs guardar' : Se usa `void save_favs()`, en donde guarda los favoritos en un archivo en la ruta asignada previamente con la función crear().

2.3 Parte 3

Se implementó el comando personalizado set recordatorio, que debe ser escrito de la siguiente forma: "set recordatorio tiempo mensaje", donde tiempo es un valor entero en segundos y mensaje es una cadena de caracteres. Cuando se ingresa este comando, la shell identifica que la segunda subcadena es "recordatorio" gracias a la funcion `ejecutar_comando()`. Esta función determina los valores del tiempo y del mensaje y se los pasa como parámetros a la función `set_recordatorio(segundos, mensaje)`. La funcion `set_recordatorio` crea un proceso hijo que utilizando la función sleep(), espera la cantidad de segundos especificada antes de mostrar el mensaje de recordatorio. Las porciones del código que permiten la ejecución de este comando son:

```
    void set_recordatorio(int segundos, char *mensaje) {
cout << "Recordatorio en " << segundos << " segundos" << endl;
pid_t pid = fork();
if (pid == 0) {
 sleep(segundos);
 cout << "\n \032[1;33mRecordatorio: \033[1;0m \n" << mensaje << endl;
 _exit(0);
}
```

Y en la función `ejecutar_comando()` es esta fracción:

```
   else if (strcmp(args[0], "set") == 0) {
 if (args[1] == NULL) {
  cout << "set: Faltan argumentos" << endl;
  return false;
 }
 if (strcmp(args[1], "recordatorio") == 0) {
  if (args[2] == NULL) {
   cout << "set: Faltan argumentos" << endl;
   return false;
  }
  int segundos = atoi(args[2]);
  char mensaje[100];
  for (int i = 3; args[i] != NULL; i++) {
   strcat(mensaje, args[i]);
   strcat(mensaje, " ");
  }
  set_recordatorio(segundos, mensaje);
  return true;
   }
```