FACULTAD DE INGENIERÍA
UNIVERSIDAD DE CONCEPCIÓN

DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

SISTEMAS OPERATIVOS

Tarea 1: Shell

Integrantes:

Javier Castillo

Máximo Beltrán

Martín Henríquez

Profesora: Cecilia Hernández

1. Objetivos

Introducir a los estudiantes en el manejo de procesos concurrentes en Unix, creación, ejecución y terminación usando llamadas a sistemas fork(), exec() y wait(). Además el uso de otras llamadas a sistema como signals y comunicación entre procesos usando pipes.

2. Descripción

En este trabajo se realizó la implementación de una shell simplificada en C, la cual acepta comandos Linux sumado un comando personalizado *"miprof"*, el cual está orientado a medir tiempos de ejecución y recursos consumidos por programas externos.

3. Metodología

3.1. Funcionamiento general

La shell funciona como un intérprete de comandos que mantiene un ciclo de lectura, análisis de texto y ejecución de procesos. Al iniciar, la shell imprime en pantalla un *prompt* (*shellao* >) que indica al usuario que el sistema puede recibir instrucciones. Cada línea ingresada desde teclado se procesa para eliminar caracteres de control y dividirla en tokens correspondientes al comando principal y sus argumentos.

Una vez analizada la entrada, la *shell* diferencia entre comandos internos y externos. Los comandos internos se ejecutan directamente dentro de la misma *shell*, mientras que los comandos externos se ejecutan en procesos hijos creados con *fork()*, donde cada hijo reemplaza su contexto con el comando solicitado mediante *execvp()*. El proceso padre usa *waitpid()* para esperar el término del proceso para asegurar que el proceso hijo se encuentre en una ejecución en primer plano.

Cuando existen tuberías, se emplean llamadas al sistema como *pipe()* y *dup2()* para redirigir las salidas estándar de un proceso hacia las entradas del siguiente. De esta forma, la *shell* soporta *pipelines* anidados de forma generalizada, con un límite de 15 *pipes* en simultáneo. El ciclo continúa repitiéndose hasta que el usuario introduce *exit*, momento en que se imprime un mensaje de cierre y se termina la ejecución de la *shell*.

3.2. Prompt

Se implementó un prompt definido como *"shellao >",* que se muestra en cada iteración del bucle principal para indicar disponibilidad de entrada.

3.3. Lectura y parsing de comandos

La lectura se realiza con *fgets()*. El *parser* elimina saltos de línea, divide la instrucción por pipes y posteriormente separa argumentos con *strtok()*. El resultado de esto es una matriz de argumentos (*argv[i][j]*) que alimenta a cada proceso hijo.

3.4. Comando *exit*

La *shell* finaliza cuando se detecta el token *exit*. Esto se gestiona retornando -1 desde la función *manejarInput()*, lo que rompe el bucle principal.

3.5. Manejo de errores y entradas vacías

Si el comando ingresado es inexistente, la ejecución de *execvp()* falla y se imprime un mensaje de error. Si la entrada es vacía, la *shell* simplemente reimprime el *prompt* sin ejecutar nada.

3.6. Ejecución concurrente de comandos

Cada comando se ejecuta en un proceso hijo creado con *fork()*. El padre espera la terminación de todos los procesos usando *waitpid()*, lo que asegura que los procesos hijos se ejecuten en primer plano.

3.7. Soporte de *pipes*

Se crearon descriptores de archivos con *pipe()* para redirigir flujos entre procesos. Con *dup2()* se establecieron las conexiones entre entrada y salida estándar. Esto habilita la ejecución de comandos encadenados.

3.8. Comandos internos

Además de *exit*, se implementaron los siguientes comandos:

- **cd**: cambia de directorio mediante *chdir()*.
- **help**: despliega un programa auxiliar que lista los comandos internos disponibles.
- **miprof**: ejecuta un programa auxiliar que mide tiempos de ejecución.

4. Resultados

4.1. Ejecución de comandos básicos

- ***ls -l*** se ejecuta en primer plano y retorna al *prompt*.
- ***cd ..*** cambia de directorio correctamente.

4.2. Manejo de errores

- Ingresar un comando no existente **hola**, esto imprime "Comando no encontrado: hola

  Pruebe usando 'help'".

- Al apretar *Enter* sin ingresar texto, se imprime una nueva línea con el *prompt* inicial.

4.3. Soporte de *pipes*

- **ls -l | grep .c | wc -l** devuelve correctamente el número de archivos **.c**.

4.4. Uso de *miprof*

- **miprof ejec ls -l** muestra tiempos de ejecución.
- **miprof ejecsave salida.txt ls -l** guarda los resultados en un archivo "salida.txt".

5. Conclusión

Con este trabajo se logró aplicar de forma práctica los conceptos de procesos, concurrencia, señales y comunicación entre procesos en sistemas UNIX, integrando esto en la creación de una *shell* interactiva capaz de ejecutar comandos, manejar errores y soportar *pipelines*. La experiencia permitió comprender cómo el sistema operativo crea y controla procesos mediante *fork()*, *exec()* y *wait()*, así como la importancia de las tuberías para coordinar la comunicación entre programas. El comando personalizado *miprof* reforzó estos aprendizajes al medir tiempos y usar señales para limitar la ejecución, mostrando una forma simple de ampliar la *shell*. En general, el desarrollo consolidó los conocimientos teóricos y su aplicación práctica en sistemas operativos.

Haga click aquí para ingresar al repositorio GitHub con el código ejecutable.