MiniShell

Sistemas Operativos

David Ángel Leo Acedo y Silvia Moreno Uribarri

Sumario

[Compilación 2](#__RefHeading___Toc115_117951520)

[Descripción del código: funciones implementadas 2](#__RefHeading___Toc105_711374270)

[Main 2](#__RefHeading___Toc117_117951520)

[Comentarios personales 3](#__RefHeading___Toc119_117951520)

1. Compilación
2. El código de la práctica incluye una librería que implementa una lista dinámica de procesos, donde cada nodo tiene ciertos atributos beneficiosos para su manejo. Cada nodo o 'tsequence' representa un secuencia introducida por el usuario. Entre otras, posee un array de los pids de los procesos hijo de cada comando de la secuencia, o vacío si aún no se ha ejecutado. Esto permitirá entre otras aplicaciones ser capaz de reanudar la ejecución de una secuencia tras su pausa entre otras aplicaciones.
3. Para la correcta compilación y uso de la librería con la minishell, se requieren los siguientes pasos:

gcc -c processList.c -o processList.o

ar -rv libprocessList.a processList.o//crea la librería

gcc myshell.c -L. -lparser\_64 -lprocessList -o myshell//compila el programa

./myshell //ejecuta la minishell

1. Descripción del código: funciones implementadas

La elaboración del código de esta práctica ha estado marcada desde su inicio por un gran deseo de modularización. El código sería extenso, y debía ser leíble y ampliable por el compañero sin necesidad de preguntar sobre ninguna línea.

Por ello, el programa está dividido en varias funciones que pasamos a comentar en detalle:

* + 1. **Main**

La función main es el padre de toda la funcionalidad. Es la encargada de mostrar el prompt al usuario, así como de leer los comandos y decidir cómo deben ser interpretados. No se encarga, sin embargo, de la multiplicidad de comandos ni de las redirecciones, sino que ambos son obviados hasta que su funcionalidad entra en juego.

Main es por tanto el primer intérprete de la entrada del usuario, y detecta si algún comando o señal introducidos no son válidos.

Por otro lado, main es capaz de ejecutar los comandos que no son programas del sistema sino internos de la sell. Implementa como fue requerido **cd**, **fg**, **jobs** y **quit** (que finaliza la shell).

Con el fin de ser capaz de coordinar la aparición del prompt con la ejecución de los comandos en primer plano, el proceso padre ejecutando main espera a aquel hijo marcado como foreground si lo hay, y no es hasta su terminación que retoma sus tareas.

**ExecuteLine**

Si los comandos introducidos son propios del sistema, la linea completa pasa a esta función. En ella, en primer lugar se realizan las **redirecciones** pertinentes de stdin, stdout o stderr a ficheros especificados por el usuario. Tras esto, comienza un código generalista que es capaz de adaptarse al número de comandos de la entrada. Si el usuario desea ejecutar en segundo plano o **background**, se añaden los comandos a una lista de procesos donde se ejecutan. La ejecución en primer plano, omitiendo la lista, es similar: llega el momento de *forkPipes*.

**ForkPipes**

La manera de entender forkPipes es a través de un input de múltiples comandos. ForkPipes recibe las redirecciones creadas, y por cada par de procesos **crea un pipe** entre ellos e invoca su ejecución (en manos de spawnProc). Ya que el bucle depende del número de comandos, este método maneja igualmente bien un comando que n encadenados. La entrada del primer pipe es la redirección stdin o en su defecto la entrada estándar, y la salida del último es la redirección de stdout o salida estándar. En el caso de stderr, todo los procesos a invocar poseerán la misma redirección si la hay, sin ningún pipe involucrado.

**SpawnProc**

Si forkPipe crea los pipes necesarios, spawnProc es el encargado de redireccionar las salidas o entradas del proceso en sí. Es llamado con cada comando de la línea, y tras **crear un proceso hijo**, **redirecciona entradas y salida**s de éste antes de invocar su mandato con **exec**. Así el proceso hijo muere tras ejecutar mientras el proceso padre está concurrentemente asignando tareas y pipes a otros hijos desde forkPipes.

Finalizan el código 2 funciones auxiliares a parte de los manejadores de señal. Éstas son jobs y getTextStatus. Jobs se invoca desde el main para su ejecución, y fue oportuno separar su código de éste por su extensión. GetTextStatus es una función auxiliar capaz de devolver el **estado de un proceso**, lo que es una herramienta útil para fg y jobs, no solamente para mostrar el estado sino para decidir si las acciones que el usuario quiere hacer sobre procesos en segundo plano son viables.

Comentarios personales

En nuestra opinión la mayor dificultad de la práctica residía en la concatenación de procesos mediante pipes. El problema más reseñable encontrado fue el hecho de que implementaciones correctas conceptualmente fallaban inexplicablemente con un índice de aletoriedad; tres ejecuciones podrían ir bien y a la cuarta fallar. Creemos que uno de nuestros obstáculos principales para conseguir la implementación ha sido la gran dificultad para debuguear los procesos hijo. También la mala modularización, como todo mal código, hacía más difícil la detección de fallos. Una modularización adecuada así como un diseño minucioso consiguió que la implementación actual fuera exitosa en las funcionalidades.

Nos parece en general que la práctica ha sido muy beneficiosa para nuestro aprendizaje y en particular para superar la asignatura con buenos conocimientos. Sin embargo sí cabría destacar la estrechez de los tiempos de entrega en la convocatoria de diciembre, donde la época de exámenes y la gran carga lectiva hace que sea casi imposible realizar una entrega de calidad. Son numerosos los detalles y posibles pulidos que se pueden dar a la práctica, y el tiempo ajustado puede hacer que se pierdan ambos, perdiendo así el aprendizaje de éstos.

El tiempo dedicado a esta práctica, evaluado obviando la primera convocatoria, ha sido aproximadamente un mes con una carga media o moderada de ambos integrantes del grupo.