MiniShell

Sistemas Operativos

David Ángel Leo Acedo y Silvia Moreno Uribarri

Índice de contenidos

Compilación

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.

La creación y testeo de esta praćtica ha sido en Ubuntu 14.04 LTS de 64bits.

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
```

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.

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 el comando o señal introducidos no es válido.

Por otro lado, main es capaz de ejecutar los comandos que no son programas del sistema sino internos de la shell. 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 la secuencia marcada como foreground si la hay, y no es hasta su terminación que retoma sus tareas.

Background y Foreground

Para conseguir el requisito de gestionar los procesos tanto en background como en foreground primero tuvimos que definir que significaban esos conceptos en el contexto en el que nos referimos. A partir de la experiencia de uso de Bash decidimos que el comportamiento debía ser el siguiente:

- Foreground: Bloquea la ejecución de la shell hasta la finalización del mandato (entendiéndose como tal un conjunto de órdenes separadas por n pipes), evitando la aparición del prompt. Además, responde ante las señales emitidas por el usuario (SIGINT, SIGTQUIT y SIGTST). En el caso concreto de SIGTST, el mandato se lanza a background.
- Background: No debe interactuar ante las señales del usuario ni interrumpir la ejecución de la shell. Devolverá su estado ante el mandato propio *Jobs* y pasará a foreground respondiendo a la orden *fg.* Desaparecerá de la lista de mandatos en background una vez hayan terminado todos su procesos y se haya informado al usuario a través de *jobs* o un intento fallido de *fg*.

Para implementar lo descrito hemos optado por crear una pequeña librería personalizada que implementa una lista básica de nuestro tipo *tSequence*, nuestra representación de una secuencia en ejecución (almacena el string del usuario y la lista de pids de sus procesos). Esta librería implementa las funciones más básicas necesarias (creación, adición y eliminación de nodos) además de una función de copia completa.

De este modo, los mandatos que el usuario vaya mandando a background irán siendo añadidos a la lista de procesos en background (con los pids de sus procesos) y la secuencia en foreground se almacenará en un nodo *tSequence* independiente. Así, al final del bucle principal, si el nodo foreground existe, la shell esperará por todos los pids asociados a él.

Para la limpieza de los procesos muertos se procede del siguiente modo: cada vez que el usuario pida información de los procesos en background (ya sea a través de jobs o de fg) la shell almacena las posiciones en la lista de procesos de aquellos que ya hayan finalizado para que la próxima vez que el usuario pida información se eliminen de la lista. De esta forma el número de mandato con el que el usuario invoca la función *fg* se mantiene consistente con respecto a lo nombrado. Además, la función getSingleTextStatus llama a waitpid si detecta que un proceso ha finalizado para que el sistema deje de almacenarlo.

Ejecutar una secuencia

Para la ejecución de una secuencia hemos divido el código en 3 funciones que se ejecutan secuencialmente para dividir a su vez tres tareas definidas: lectura del input (background/foreground y redirecciones), gestión de los pipes (existan o no) e invocación del mandato.

Execute Line

ExecuteLine se encarga de leer el input tokenizado y preparar las redirecciones y el modo para su ejecución. Si hay redirecciones abre los ficheros necesarios (y los cierra al

final de la ejecución) y decide si el proceso ha de ser incorporado a la lista de procesos en background o debe quedarse en foreground.

ForkPipes

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*, todos los procesos a invocar poseerán la misma redirección si la hay, sin ningún pipe involucrado. Forkpipes devolverá la lista de pids de los procesos invocados.

SpawnProc

Si *forkPipes* 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 salidas 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*.

Es importante destacar que para gestionar apropiadamente las señales, los procesos se crean en grupos distintos del padre para evitar que bash propague las señales a los procesos ya invocados sin nuestro control (ya que una vez hecho exec el control de las señales pasa a gestionarlo el programa concreto que se invoque).

Funciones de estado

Finalizan el código 2 funciones auxiliares a parte de los manejadores de señal. g*etTextStatus* es una función auxiliar capaz de devolver el estado de una secuencia utilizando para ello la función auxiliar *getSingleTextStatus*. Ésta última devuelve el estado de un solo proceso por dos vías complementarias: en forma de texto plano a través del parámetro status y devolviendo un entero que indica si los procesos han muerto o no. Dicha funcionalidad será utilizada por *fg* y *jobs*, no solamente para mostrar el estado de una secuencia, sino para decidir si las acciones que el usuario quiere hacer sobre procesos en segundo plano son viables.

Para obtener el estado de un proceso accedemos al fichero */proc/[pid]/stat*, leemos su estado y lo traducimos a texto plano según lo indicado en el manual de Linux. Así mismo, si el estado corresponde a un proceso muerto lo limpiamos de la lista de procesos con *waitpid(NOHANG)* y devolvemos 1 para indicarlo.

Gestión de señales

Los gestores de señales son bastante simples. Su única función es evitar que el bucle principal muera por su causa y propagar la señal recibida únicamente al proceso en primer plano. La señal de pausa tiene más enjundia porque manda el proceso pausado a background y, por tanto, tiene que modificar la lista de procesos.

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.

Por otro lado, la falta de funcionalidad de alto nivel del lenguaje (estructuras de datos, funciones) ha incomodado el proceso de aprendizaje de algunos fundamentos ya que aumentaba la posibilidad de fallos de implementación de las rutinas.

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.