Práctica 1 - Minishell

Héctor Rodrigo Iglesias Goldaracena Y Juan Montes Cano

1 Índice de contenidos

2 Descripción del código

2.1 Funcionalidad implementada

El programa es capaz de cumplir los objetivos que se proponen en el enunciado de la práctica. En efecto, es capaz de:

- reconocer y ejecutar tanto en *foreground* como en *background* líneas con un mandato con sus respectivos argumentos,
- reconocer y ejecutar tanto en *foreground* como en *background* líneas con dos o más mandatos con sus respectivos argumentos, enlazados por medio de «|»,
- reconocer y aplicar redirección de entrada estándar desde archivo, y redirección de salida estándar y de salida de error a un archivo,
- ejecutar los mandatos internos `cd`, `fg` y `jobs`,
- y evita que tanto los comandos en *background*, así como el Minishell, finalicen al enviar por teclado las señales `SIGINT` y `SIGQUIT`, mientras que permite que los procesos en *foreground* respondan ante ambas señales.

2.2 Pseudocódigo y planteamiento del programa

2.2.1 Planteamiento del programa

En primer lugar, el programa consta de un proceso principal, `PShell`, que cuenta con las siguientes características:

- nunca muere, a no ser que se lance `EOF` (`Ctrl+D`) en el caso de que nos encontremos en el `fgets` que está a la cabeza del bucle principal,
- y nunca ejecuta ninguna instrucción, salvo aquellas implementadas por nuestro propio código (`cd`, `fg`, `jobs`).

Como consecuecuencia de estas dos propiedades, `PShell` puede gestionar una lista de procesos en *background*, lo que nos permite marcarlos como acabados, detenidos o en ejecución en función de en qué estado se encuentren.

En caso de que se pida un mandato distinto a los tres de los que ofrecemos la implementación, se crea un hijo, `PMandato` (al que `PShell` esperará, si el mandato no se ejecuta en *background*), que será el encargado de supervisar la ejecución del mandato que se haya pedido, independientemente de que se esté ejecutando o no en *background*. Para ello, aplicará las redirecciones oportunas, que irán heredándose a cada uno de los procesos hijo que este supervisor vaya originando.

`PMandato` creará, entonces, tantos hijos como mandatos haya en la línea cuya ejecución tenga que supervisar (de uno en uno, solo existirá un hijo en un instante de tiempo dado).

Si `PMandato` recibe una línea que se tenga que ejecutar en *background*, `PShell` registra el `pid` de `PMandato` en una lista que constará de procesos en *background*, y continuará su ejecución sin esperar a la finalización de `PMandato`, quien supervisará con normalidad el mandato que le hayan asignado. Además, `PMandato` y sus hijos ignorarán las señales de `SIGINT` y de `SIGQUIT` de haberse lanzado en *background*, pero pasarán a atenderlas si dicho mandato pasa a *foreground*.

En todo momento, `PShell` estará atendiendo a los estados por los que atraviesa cada uno de sus `PMandato`s, actualizando con esta información su lista de procesos.

En la sección de pseudocódigo se procede a explicar más concretamente cómo implementamos la comunicación de `PMandato` con sus hijos, así como el funcionamiento del proceso `PShell` y de su lista de procesos, a la que nos referiremos como `ListaPID`.

2.2.2 Pseudocódigo

Vamos a explicar desde diferentes puntos de vista (el de `PShell`, el de `PMandato` y el de sus hijos) el desarrollo del algoritmo en pseudocódigo.

- `PShell`
    - Inicialización

      Declaramos los `sigaction`, su correspondiente `ListaPID` y otras variables auxiliares. Inmediatamente después, entramos en un bucle `while`.

    - Bucle `while` de lectura

      Una vez nos hayan pasado por pantalla texto (o bien un salto de línea), actualizamos preventivamente `ListaPID` y distinguimos casos:

      1. Si nos han pasado un salto de línea o la línea no se ha podido leer correctamente, actualizamos de nuevo `ListaPID` y volvemos el bucle.
      2. Si nos han pedido ejecutar `jobs`:
         - actualizamos `ListaPID`,
         - y mostramos `ListaPID` en orden, y acto seguido la «limpiamos» de procesos acabados.
      3. Si nos han pedido ejecutar `cd`:
         - cambiamos el directorio de trabajo,
         - actualizamos `ListaPID`,
         - y limpiamos `ListaPID` de procesos terminados.
      4. Si nos han pedido ejecutar `fg`, distinguimos casos en función de en qué estado se encuentre el proceso que pasará a ejecutar en primer plano:
         i. Si `ListaPID` está vacía, indicarlo al usuario.
         ii. Si `ListaPID` cuenta con un trabajo recién terminado como último elemento de su lista, indicarlo al usuario.
         iii. Si `ListaPID` tiene como último elemento de su lista a un elemento en ejecución, «indicar» al correspondiente `PMandato` que debe atender a `SIGINT` y a `SIGQUIT`, pasarle el control del minishell, y esperar por su finalización.
         iv. Si `ListaPID` tiene como último elemento de su lista a un elemento en ejecución, lanzar una señal de continuación al correspondiente `PMandato`, «indicar» al correspondiente `PMandato` que debe atender a `SIGINT` y a `SIGQUIT`, pasarle el control del minishell, y esperar por su finalización.

      En cualquier caso, se actualiza y se limpia posteriormente `ListaPID`.

5. Si no es ninguno de los anteriores, entonces `PShell` creará un `PMandato`, que supervisará la ejecución del mandato no implementado por nuestra shell.

   – Si el mandato pedido se ejecuta en *background*, `PShell` continúa su ejecución habitual e irá «esperando» dicho `PMandato` por medio de su correspondiente actualización de `ListaPID`.

   – Si el mandato pedido se ejecuta en *foreground*, `PShell` esperará a que finalice el correspondiente `PMandato`, y le indicará que escuche a `SIGINT` y a `SIGQUIT`.

- `PMandato`

  ◦ Inicialización de descriptores de fichero: si hay redirecciones de entrada, salida o error las aplica correspondientemente.

  ◦ Distinguimos casos en función del número de mandatos:

    1. si se ha de ejecutar un único mandato se crea un solo hijo, cuyo `pid` guardará `PMandato` de forma global. Este hijo será el encargado de ejecutar el mandato en cuestión.

    2. Si se ha de ejecutar más de un mandato, se creará un hijo para cada mandato. Tales hijos se irán intercomunicando por medio `pipe`s.

    En cualquier caso, informar correspondientemente al padre del estado en el que se encuentra el hijo (si ha acabado, o si se ha bloqueado debido al intentar leer estando en *background*).

  ◦ Si el proceso está en *foreground* o está en *background* pero ha sido traído a *foreground*, se debe devolver el control de la shell a `PShell`.

2.3 Descripción de las principales funciones implementadas

2.3.1 `int escribirPrompt()`

Esta función se encarga de escribir un *prompt* personalizado teniendo en cuenta el directorio de trabajo actual. Devuelve 1, a no ser que no exista la variable de entorno `HOME`, en cuyo caso devuelve 0, provocando la finalización del minishell.

2.3.2 `static void devolverControl(int sig, siginfo_t* siginfo, void* context)`

Previamente, se ha definido de forma global un `struct sigaction`, con flags `SA_SIGINFO` y `SA_RESTART`, lo que nos permite obtener, respectivamente, mayor información y evitar la muerte del proceso receptor de la señal una vez ejecutado su manejador. Esta función está destinada a que sea ejecutada por algún `PMandato`.

Dentro de la función `devolverControl`, lanzamos una señal de terminación al correspondiente proceso hijo de `PMandato`, para que dicho proceso hijo acabe de forma ordenada y no haga un uso estéril de los recursos del ordenador. Seguidamente, le entrega el «control» a `PShell`, que continúa ejecutando nuestro programa.

2.3.3 `void esperarHijos()`

Esta función, destinada a ser ejecutada por `PShell`, se encarga de realizar un «barrido» general de los estados de sus `PMandato`s con fin de actualizar `ListaPID` en consecuencia, marcando dichos procesos de su tabla como detenidos o terminados.

2.3.4 redirecciones (`redirStdin`, `redirStdout`, `redirStderr`)

Son funciones que se encargan de gestionar los descriptores de fichero de aquellos procesos destinados a realizar mandatos. Esto es, el proceso raíz no aplica redirección alguna.

2.3.5 `ejecutarComando(int i, tline* line)`

Esta función se encarga de lanzar el mandato especificado por `tline`.

2.3.6 Funciones de control de lista de procesos

Para la correcta gestión de procesos en *background*, hemos implementado la ya mencionada `ListaPID` como una lista con puntero al inicio y al final para lograr la mayor eficiencia posible de inserción de procesos, así como una rápida extracción de los mismos por medio del mandato implementado `fg`.

Las más relevantes, que escapan de la habitual implementación de una lista de estas características, son las siguientes:

- `int limpiarLista(listaPIDInsercionFinal_t* L, int clasificar)`

  Esta función se encarga de eliminar mandatos en *background* que estuvieran marcados como hechos. Cuenta con un flag, `clasificar`, cuya función nos permite diferenciar entre lo que se mostraría una vez acabado un mandato «habitual» (una línea que muestra el mandato como realizado) y lo que se mostraría tras ejecutar `jobs` (toda la lista de mandatos en *background*, estén realizados o no, en el orden en que fueron insertados en la lista).

- `void borrarElementoPID(pid_t pid, listaPIDInsercionFinal_t* L)`

  Como los `pid` de los procesos son únicos, nos podemos permitir, a la hora de eliminar un nodo de nuestra lista, comparar únicamente los `pid`s, y eliminar el nodo correspondiente.

- `void terminarElem(elem_t* elem)`

  Marcar el elemento dado como hecho (es decir, poner su flag de estado a cero).

- `void ejecutarElem(elem_t* elem)`

  Marcar el elemento dado como hecho (es decir, poner su flag de estado uno).

- `void detenerElem(elem_t* elem)`

  Marcar el elemento dado como detenido (es decir, poner su flag de estado a dos).

3 Comentarios personales

3.1 Problemas encontrados

- En el segundo algoritmo que probamos antes de realizar este programa, intentamos utilizar exclusivamente dos `pipe`s para intercomunicar un proceso hijo y un proceso padre. Llegados a cierto número de `pipe`s, estas quedaban bloqueadas para lectura e impedían la ejecución del resto de mandatos, que se quedaban esperando a recibir entrada con la que trabajar.

- Otro problema importante que tuvimos en una antigua implementación que probamos fue el llamar, por medio de `fg`, a un proceso en *background*. En dicho planteamiento, tuvimos problemas para acceder al `pid` de un proceso que se encontraba en ejecución, dado que en nuestra lista de `pid`s contábamos exclusivamente con aquellos de los procesos que se derivaban de forma directa del proceso principal, los `PMandato`s.

- Dentro de nuestros planteamientos en papel del minishell, necesitábamos conocer la señal del proceso que enviaba la señal al receptor, lo cual nos obligó a utilizar `sigaction` en detrimento de `signal`.

- Modificar, desde el proceso raíz, la forma de actuar ante una señal por parte de un proceso que se encuentra ejecutando un mandato. Aunque, en papel, de nuevo, éramos capaces de solucionar el problema, se sucedían situaciones inesperadas y ante las que no encontramos respuesta; por ejemplo, dicho proceso moría inmediatamente tras mandarle una señal —por ejemplo, `SIGUSR1`—, con un manejador distinto al por defecto. Esto lo solventaríamos, posteriormente, alzando el flag de `SA_RESTART` en un correspondiente `sigaction`.

- Otro problema que tendríamos sería el modificar, desde `PShell`, el comportamiento ante las señales por parte de procesos derivados de los `PMandatos`. Debido a esto, tuvimos que replantear de nuevo nuestra solución e integrarla con arreglo a las herramientas que nos ofrecían los mandatos `tcsetpgrp(3)` y `setpgid(3)` de control de grupos de procesos.

  Además, dichas funciones nos permitirían dar una solución simple a otro problema que presentaba nuestra implementación antigua. En efecto, el ejecutar en *background* comandos que solicitasen entrada de teclado debían ser detenidos, cuando nosotros los dejábamos «escuchando» por la entrada estándar, cuando no la cerrábamos para acabarlos inmediatamente.

3.2 Críticas constructivas y propuestas de mejora

- Estaría bien que, con el fin de aligerar la cantidad de texto en el código, y con fines puramente organizativos, pudiésemos implementar por nuestra cuenta archivos auxiliares (`.h`, `.c`), con los que, entre otras cosas, poder gestionar los tipos de datos más cómodamente, así como poder realizar pruebas aparte sin tener que comprometer con ello todo el código que había previamente.

- Un «juez electrónico» (como los de los concursos de programación) que evalúe la práctica para saber antes de entregar qué problemas tiene, y así poder asegurarnos la máxima nota sabiendo qué errores corregir antes de la corrección por parte de los profesores.

3.3 Evaluación del tiempo dedicado

Pensamos que hemos invertido más tiempo del que nos gustaría, causado por problemas más de implementación que de diseño, lo que nos llevaría a replantear nuestro algoritmo en varias ocasiones.

Nuevamente, también parte de este tiempo lo invertimos en investigar soluciones alternativas a los problemas que nos causaban las herramientas con las que contábamos de entrada. No obstante, esto nos ha permitido crear un mejor intérprete de mandatos.