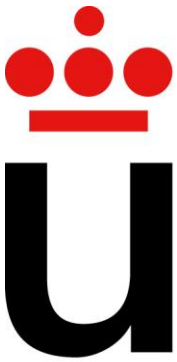

Universidad
Rey Juan Carlos

Sistemas Operativos

[PRÁCTICA 2 - MINISHELL]

AMANDA CASTRO LÓPEZ Y CARLOS RODRÍGUEZ GÓMEZ

TABLA DE CONTENIDO

Autores

Amanda Castro López
DNI: 21155084Y
Correo de la administración: c.rodriguezgo.2018@alumnos.urjc.es
Carlos Rodríguez Gómez
DNI: 21155084Y
Correo de la administración: a.castrol.2020@alumnos.urjc.es

Descripción del Código

Diseño del Código

Explicación e Introducción del Diseño del Código:

Para comenzar a podernos plantear los requisitos y objetivos de esta práctica, decidimos realizar un desglosamiento de cada requisito y de que significaba a nivel de código cada uno de ellos para de esta forma ir dando, al menos en nuestra cabeza forma a la estructura de la práctica. Como esta vez era una practica mas densa que la anterior, aunque pensamos en hacer pseudocódigo tuvimos que descartarlo porque necesitábamos optimizar todo el tiempo posible, ya que no íbamos con demasiado tiempo debido a la gran carga de trabajo que hemos tenido.

Estrategia de Ejecución de Mandatos y Gestión de Tuberías

A la hora de ejecutar mandatos y gestionar las tuberías, lo hicimos centrándonos en conseguir primero tener una versión en primer plano funcional y ejecutable, y una vez viésemos y comprendiésemos el segundo plano, trataríamos de implementarlo para el segundo plano.

De esta manera comenzamos, para cumplir con la ejecución en primer plano tenemos un código capaz de ejecutar uno o más mandatos. Cada mandato será un nuevo proceso y a su vez todos estos procesos serán hijos del mismo padre, que será la propia MiniShell. El código se ha separado en dos partes principales, si hay un único mandato o si hay más de un mandato.

En caso de que haya un solo mandato, el proceso hijo simplemente contralara cualquier posible redirección y sencillamente ejecutará el mandato en cuestión. Su padre, por otro lado, tendrá que esperarlo mediante un waitpid antes de poder volver a mostrar el prompt.

Si hay más de un mandato entonces, será necesario la utilización de tuberías que conecten los distintos hijos para que de esta forma puedan pasarse la información. Para cumplir con esto, primero creamos una tubería para el número total de comandos menos uno, es decir si hay tres comandos, habrá dos tuberías. En función de si estamos en el padre o el hijo, y dependiendo de si el proceso hijo es el primero, intermedio o último se irán abriendo o cerrando las tuberías de diferentes formas

En caso de que nos encontremos en el primer mandato conectaremos mediante el dup2 la salida del primer proceso hijo y la entrada del segundo proceso padre, y después se cerraran todas las tuberías, además podremos controlar si hay redirección de salida.

Si el proceso hijo es el intermedio tendremos como entrada del hijo la salida de la tubería del anterior proceso hijo, y conectaremos nuestra salida, a la entrada de la siguiente tubería, para que de esta forma el siguiente proceso hijo reciba por entrada la salida de esta tubería, de nuevo cerraremos todas las tuberías.

Por último, si nos encontramos en el proceso hijo del último mandato, recibiremos por entrada la salida de la última tubería y cerraremos el resto de las tuberías, además tendremos en cuenta las redirecciones de salida y de salida de error.

Ahora tendremos que controlar las tuberías del padre, debido a que cada vez que un hijo nuevo es creado, heredará las mismas tuberías que el padre tenga abiertas. Y por eso el proceso padre según vaya avanzando tendrá que ir cerrando las tuberías ya usadas, para que de esta forma los procesos hijos no hereden tuberías inútiles y se puedan producir fallos a la hora te transmitir los datos, de esta forma si el padre ahora lo siguiente dependiendo del mandato:

Cuando se encuentre en el primer mandato cerrara la escritura de la primera tubería.

En caso de que el padre se encuentre en un mandato intermedio, se cerrara la lectura de la tubería anterior y la de escritura de la actual.

Por último, cuando se encuentre valga la redundancia en el último mandato se cerrará la lectura de la tubería actual.

De esta manera conseguimos gestionar las tuberías, pudiendo ejecutar las instrucciones con múltiples mandatos y haciendo que de la misma manera que en caso de un solo comando el padre espere.

Implementación de Background o Segundo Plano

El funcionamiento del segundo plano consiste en utilizar la misma estructura que hemos usado para el primer plano, pero sin permitir que el padre espere por su hijo antes de empezar su ejecución. De esta manera, el hijo se quedaría ejecutándose en segundo plano.

Una vez que se ha terminado la ejecución del padre, se repite otra vez el bucle principal del programa y se vuelve a ejecutar el prompt, y una vez introducido el comando, se hace una revisión del segundo plano. Esto, quiere decir que se comprueba si el proceso hijo ha acabado su ejecución. En el caso de que se haya terminado, es el momento de hacer que el padre espere por el hijo para que pueda morir, y así no convertirse en zombi.

De una forma más técnica, en el caso de que el bit de line->background esté activado, realiza una ejecución muy similar a cuando está en primer plano. Esta se diferencia en que cuando el padre se está ejecutando, no hay ningún waitpid a la espera del hijo.

Para que este hijo no quede desprendido del programa, lo que se hace es guardar su pid en un array de structs. Este struct está compuesto por un array de pids (por si se da el caso de que es un conjunto de mandatos enlazados por tuberías) y por un string donde se guarda el prompt de esa ejecución. Montando este array, tenemos un struct por cada ejecución que se ha realizado.

Un detalle que destacar es que, a la hora de ejecutar comandos en segundo plano unidos por tuberías, hemos tenido en cuenta el caso de que uno de los comandos intermedios o el final fuese erróneo. En este caso no se guardaría en el array del segundo plano.

Una vez que se ha terminado la ejecución dentro del mandato en segundo plano, al acabar el bucle while que engloba el programa y volver a empezar, tenemos un bucle for que decrece. Esto se debe a que, si tenemos 0 mandatos en segundo plano, no va a conseguir meterse en el bucle, ya que, si la variable i que recorre el for es igual a 0 y su límite es 0, no puede hacer ninguna iteración. Funcionaría como una especie de if con contador.

Una vez dentro de este bucle, cada i va a representar un struct del array. Así que empezaría por la ultima ejecución que se ha introducido por el prompt. Durante el bucle, nos encontramos un while; este será el encargado de comprobar que todos los pids de ese struct han acabado. En el caso de que, aunque solo sea uno el que no lo haya conseguido, significará que la ejecución sigue en marcha y por tanto no se realiza el waitpid correspondiente a cada pid del array de pids de ese struct. En caso contrario, se ignorará la existencia de ese struct (ya que la ejecución en su totalidad ha acabado) y el array de struct se someterá a otro bucle while para recolocar los procesos del segundo plano en orden.

Para la comprobación del correcto funcionamiento de lo explicado anteriormente, se realizó el código de jobs y fg, que explicaremos más adelante.

Implementación de Señales

Respecto a las señales, se nos pedía que cualquier comando lanzado en segundo plano y la propia minishell no reaccionasen a la señal de control SIGINT o Ctrl+C, esto lo conseguimos con el comando signal y añadiendo SIG_IGN como segundo argumento, ya que el primero tiene que ser la señal que queremos modificar, en este caso SIGINT.

De esta forma conseguimos ignorar la señal de control SIGINT en la MiniShell colocando el comando signal y los argumentos SIGINT y SIG_IGN antes de llegar al bucle del while, donde leerá hasta que se cierre la MiniShell en este caso con exit. Esta línea de comando es para que si aun no hemos escrito nada y no este dentro del while si hacemos Ctrl+C la MiniShell ignore la señal, pero también tenemos que colocar de la misma manera signal dentro del bucle while para que cuando empiece a leer por teclado no reaccione a Ctrl+C.

Además de esto para ingorar la señal en los procesos de segundo plano tanto para un mandato como para varios mandatos lo que hacemos es que tanto al hijo como al padre le asignamos con signal, que la señal SIGINT se ignora. Mientras que en los procesos en primer plano tanto para un mandato como para varios haremos que el padre los ignore para que no se salga de la MiniShell, pero los procesos hijos si reaccionaran a Ctrl+C, para esto necesitaremos utilizar el comando signal indicando la señal que se quiere cambiar (SIGINT) y la restablecemos a por defecto ya que por defecto reaccionaria a esa señal (SIG_DFL).

Implementación De Jobs+Fg

Por un lado, jobs lo único que hace es recorrer el array de struct del segundo plano mostrando el número de su ejecución y el prompt al que está relacionado. Además de mostrar por pantalla un mensaje de aviso en el caso de que no haya trabajos ejecutándose.

Por otro lado, la función de fg es más compleja. Esto se debe a que hay que tener en cuenta cuatro cosas: que haya procesos en el segundo plano, si el comando es únicamente la palabra fg, si el comando tiene un argumento o si tiene más de un argumento.

Si está escrito fg sin argumento, se comprueba primero si hay procesos en el segundo plano, en el caso de que sí, lo único que se hace es un waitpid a los pids de del último comando del array de ejecuciones. Esto se puede gracias a una variable que llevaba la contabilidad de las ejecuciones que entran en el segundo plano y que terminan. Después de este waitpid, se decrementa dicha variable. Y, en caso de que no haya comandos en segundo plano se muestra un aviso por pantalla.

Si en vez de no tener argumentos, tiene, comprobamos si el argumento es válido y si además es un número comprendido entre los procesos del segundo plano. Es caso afirmativo, se hace un while a los pid de este struct que nos han indicado con el argumento de fg. En caso negativo, se muestra un aviso por pantalla del error.

Estructuras de Datos Específicas

Para la resolución del segundo plano hemos necesitado crear un struct nuevo llamado tBG, donde en una variable llamada pids de tipo puntero a strings hemos almacenado el pid de los procesos que iban entrando en segundo plano y otra variable donde lo que hace es guardarse el prompt de esa ejecución en formato string.

Principales Funciones

	Main	Nombre	Tipo	Descripción
Argumentos		**void**		**La función main no necesita argumentos**
Variables Locales	Variable 1	**buf[BUF]**	**char**	**String donde se almacena la línea que se está leyendo actualmente**

	Variable 2	**ruta**	**char ***	**Puntero de string que se utiliza para donde se almacena el string de la dirección para que funcione cd**
	Variable 3	**cwd[]**	**char**	**String donde se almacena la dirección a la que se va a acceder**
	Variable 4	**line**	**tline ***	**Es el struct de la librería parse.h que almacena todo lo relacionado con los mandatos.**
	Variable 5	**i**	**int**	**Variable auxiliar para recorrer estructuras de control for**
	Variable 6	**j**	**int**	**Variable auxiliar para recorrer estructuras de control for**
	Variable 7	**l**	**int**	**Variable auxiliar para recorrer estructuras de control for**
	Variable 8	**fdin**	**int**	**Almacena el valor para abrir un fichero**
	Variable 9	**fderr**	**int**	**Almacena el valor para abrir un fichero**
	Variable 10	**fdout**	**int**	**Almacena el valor para abrir un fichero**
	Variable 11	**nump**	**int**	**Es el numero de procesos en segundo plano**
	Variable 12	**estado**	**int**	**Indica el estado en el que se encuentra la espera de un hijo para compararlo y hacer cosas en función de su valor**
	Variable 13	**largo**	**int**	**Variable auxiliar para ayudar a una mejor visibilidad del prompt**
	Variable 14	**comandos**	**int ***	**Array para guardar el pid de cada hijo**
	Variable 15	**background**	**tBG ***	**Creamos un array de estructuras para ir guardando nuestros datos**
	Variable 16	**acabado**	**bool**	**Indica si un proceso a acabado o no**
	Variable 17	**correcto**	**bool**	**Variable auxiliar para enviar un mandato a segundo plano hasta que se demuestre lo contrario**

Valor Devuelto		**void**		**No devuelve nada al ser la función main**
Descripción de la Función				**Programa entero**

Casos de Prueba

Para el correcto funcionamiento de la MiniShell hemos utilizado los siguientes casos de uso:

Un solo mandato:

```
carlitosrogo@carlitosrogo-VirtualBox:~/Escritorio/Practica2$ ./test
msh> ls
fichero.txt   hola.txt  libparser32.a  ncommands  prub.txt  test.c
ficherto.txt  holi.txt  libparser.a    parser.h   test
msh>
```

Redirecciones con solo un mandato:

```
msh> ls > Nuevo.txt
redirección de salida: Nuevo.txt
```

Abrir — Nuevo.txt [Solo lect — ~/Escritorio/Practica2

```
1 fichero.txt
2 ficherto.txt
3 hola.txt
4 holi.txt
5 libparser32.a
6 libparser.a
7 ncommands
8 Nuevo.txt
9 parser.h
10 prub.txt
11 test
12 test.c
```

```
msh> grep a < Nuevo.txt
redirección de entrada: Nuevo.txt
hola.txt
libparser32.a
libparser.a
ncommands
parser.h
msh>
```

Mandatos con Pipes:

```
msh> ls | grep a | wc
      5       5      54
msh>
```

Redirecciones con pipes:

```
msh> grep h < holi.txt | wc > Nuevo2.txt
redirección de salida: Nuevo2.txt
msh>
```

Abrir — Nuevo2.txt [Solo lectu... — ~/Escritorio/Practica2

```
1       3       6       52
```

Jobs:

```
msh> sleep 10000 &
msh> sleep 12312312 &
msh> sleep 23423 &
msh> jobs
[1]+ Running    sleep 10000 &
[2]+ Running    sleep 12312312 &
[3]+ Running    sleep 23423 &
msh>
```

CD:

```
carlitosrogo@carlitosrogo-VirtualBox:~/Escritorio/Practica2$ ./test
msh> cd
Ruta actual: /home/carlitosrogo
msh> cd ..
Ruta actual: /home
msh> cd ./carlitosrogo/Escritorio
Ruta actual: /home/carlitosrogo/Escritorio
msh>
```

FG:

```
msh> sleep 10000 &
msh> fg
El trabajo sleep 10000 &  esta en foregraund.
```

Señales:

```
msh> sleep 1000
^Cmsh> ls
fichero.txt   hola.txt  libparser32.a  ncommands   Nuevo.txt  prub.txt  test.c
ficherto.txt  holi.txt  libparser.a    Nuevo2.txt  parser.h   test
msh>
msh> ls
fichero.txt   hola.txt  libparser32.a  ncommands   Nuevo.txt  prub.txt  test.c
ficherto.txt  holi.txt  libparser.a    Nuevo2.txt  parser.h   test
msh> ^C^C
```

Exit:

```
msh> ^C^Cls
fichero.txt   hola.txt  libparser32.a  ncommands   Nuevo.txt  prub.txt  test.c
ficherto.txt  holi.txt  libparser.a    Nuevo2.txt  parser.h   test
msh> exit
carlitosrogo@carlitosrogo-VirtualBox:~/Escritorio/Practica2$
```

Estos son los principales casos de uso que hemos capturado en imágenes con la función de intentar expresarlo de una forma más visual, evidentemente existe muchas más variaciones y combinaciones que no hemos llegado a capturar o

errores como en la primera imagen de señales donde se observa que después de hacer Ctrl+C a un proceso en primer plano se tabula mal la siguiente línea.

Comentarios Personales

Somos conscientes de que nuestro código tiene un gran problema que es que no hemos utilizado funciones para evitar duplicar código y lo hace mucho menos intuible además de muy denso y cansado. Es por eso por lo que tenemos muchas líneas de código, que se podrían a cortar con subprogramas como para las redirecciones y los propios mandatos, además muchas mas que seguro que no hemos pensado, pero debido a errores que nos surgían al intentar separarlo desde el principio de la practica y la gran carga de trabajo nos hemos visto obligados a mantener esa estructura por no disponer del tiempo necesario o bien los conocimientos necesario o tiempo para aprenderlos.

Olvidando el gran problema de código duplicado, llegamos a la conclusión de que nos ha aportado un gran conocimiento esta práctica ya que tenemos todo hecho menos el unmask que no comprendíamos y hemos decidido no hacerlo por falta de tiempo. Creemos que es una práctica muy autodidacta, quizás demasiado, sobre todo en la parte de segundo plano, que es donde mas chicha hay.

Respecto al tiempo hemos dedicado entorno a 1 semana entera la (hablando de días de trabajo) y habremos trabajado entorno a 4-5 horas al día juntos cuando podíamos y cuando no individualmente, más las vueltas de cabeza que le dábamos indirectamente fuera de esas horas para intentar hallar las soluciones a nuestros problemas de ejecución.

Como posible mejora, nosotros creemos que si se explicase alguna herramienta para Debugear sería bastante mas sencillo alcanzar algunas soluciones.