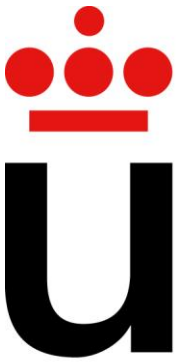

Universidad
Rey Juan Carlos

Sistemas Operativos

PRÁCTICA 2: MINISHELL

STEFANO TOMASINI HOEFNER

TABLA DE CONTENIDO

Autor

Stefano Tomasini Hoefner
Correo: s.tomasini.2019@alumnos.urjc.es

Descripción del código

Diseño del código

Primeramente, en *main*, hay un bucle de pedido de input al usuario de la shell. Cuando se lee una línea, se escribe en un buffer, el cual es pasado por referencia a *tokenize* como argumento, que devuelve un puntero a *tline*, que a su vez es procesado en mi función *run_line*.

En esta función, primero se detecta si hay algún comando nativo incluido y se maneja tal caso, teniendo en cuenta las restricciones (no se pueden redireccionar su entrada y/o salidas; y no pueden ser concatenados con otros comandos). Si no hay ningún comando nativo dentro de la *line*, entonces se continúa y se aloca suficiente espacio en el *heap* para el "array" de tuberías. Siguiente, se aloca una tubería (un "array" de dos enteros) por cada posición del array, y se "pipea".

Una vez terminada esa preparación, se empieza a crear hijos para los comandos. Por cada comando contenido en *line*, se realiza un *fork*, y si se es el proceso hijo se configura la señal *SIGINT* a que sea ignorada. Siguiente, se realizan las sustituciones de los descriptores de fichero de *stdout*, y/o *stdin*, y se cierran todas las tuberías no utilizadas por ese hijo concreto. Todos los hijos llaman a la función *close_non_adjacent_pipes*, que cierra todas las tuberías que no estén relacionadas con el hijo actual (las que fueron creadas para ser utilizadas por otros mandatos-hijos no adyacentes). El padre cierra todas las tuberías que tiene abiertas en su proceso una vez se realicen todos los *forks*.

Antes de empezar a esperar a esperar a los hijos, se comprueba si la línea se envió al "background", o no.

En el caso de que sea para el "foreground", se espera bloqueantemente que termine cada hijo. Si algún hijo falla, o se cancela la ejecución, la función *stop_foreground_execution* se encarga de enviar la señal *SIGTERM* a los hijos restantes por ejecutar, y posteriormente matarlos asíncronamente con *SIGKILL*, debido a la posibilidad de que no respondan adecuadamente al *SIGTERM* y no se mueran (este *SIGKILL se envía* con un cierto retraso para darles tiempo de liberar sus recursos correctamente tras recibir el *SIGTERM*). Volviendo a la función que estábamos describiendo previamente, *run_line*, tras esperar a todos los hijos, se liberan los dos "arrays" alocados en el *heap*: el de todas las tuberías y el de *pids* de los hijos "forkeados" (este último "array" solo se liberará cuando no lo haga *stop_foreground_execution*).

En el caso de que la línea se mandó al *background*, se instancia un hilo nuevo el cual tendrá la responsabilidad de crear el *job* correspondiente, esperar a la terminación de cada mandato-hijo, manejar posibles fallos, y liberar la memoria utilizada una vez se termine la ejecución de la línea completa. Hay que hacer una copia profunda de los *strings* apuntados por la *tline* ya que la siguiente llamada a *tokenize* va a invalidar los bloques de memoria originales para dichos *strings*, al liberarlos. Los argumentos para crear el *pthread* mencionado anteriormente se guardan en un *struct* de tipo *AddJobArgs*, que se guarda en el *heap*, no en la pila. Esto es debido a que inmediatamente se va a retornar de la función actual, entonces el *thread* apuntaría a memoria inválida si se le pasara un puntero a variables de la pila. El *thread* se encargará de liberar el *struct* de argumentos una vez copie en otra variable sus valores contenidos, la cual vivirá en la pila propia del nuevo hilo.

Consiguientemente, se crea un *Job*, en el cual se vuelcan los datos recibidos por el argumento, y el puntero a la *tline* correspondiente. Se añade el trabajo al array global compartido *bg_jobs*, pero dentro de un *mutex_lock*, para no realizar modificaciones mientras un hilo ajeno esté leyéndolo o modificándolo. A continuación, se realiza la espera de cada hijo, de forma bloqueante, pero ya que es un hilo distinto, no bloquea al foreground. Si falla la ejecución de algún mandato-hijo, se actualiza el estado del *job* a *FAILED* (si este hilo no se definió como el *foreground*; esto será explicado posteriormente), y se matan a todos los hijos posteriores adecuadamente (primero enviando *SIGTERM*). Si termina la ejecución correctamente, se actualiza el estado del trabajo a *DONE* en el array global.

Para implementar el mandato *jobs*, hace falta el array global *bg_jobs*, ya que en mi implementación este es usado por múltiples hilos para agregar su propio *job*, o actualizar el estado para que este se vea al ejecutar *jobs*. Este comando está implementado en la función *execute_jobs*. Realiza un *mutex_lock* antes de imprimir cada trabajo existente en *bg_jobs*. Si el trabajo está terminado, entonces se inserta su *uid* (identificador único) en el array de trabajos a borrar. Tras finalizar la impresión de todos los trabajos junto a su estado, se borran todos los trabajos identificados como terminados, se suelta el *mutex*, y se libera este "array" (ya que vive en el *heap*).

Para realizar la implementación del mandato *fg*, se hizo uso de varias variables globales necesarias debido a la programación multihilo: *foreground_thread*, *fg_forks_pids_arr*, *fg_forks_pids_arr*, *fg_n_commands*, *fg_forks_pids_arr*, *fg_awaited_child_cmd_i*, *fg_execution_cancelled.* En la propia función *execute_fg*, lo que se hace es buscar al trabajo de *bg_jobs* que tenga la *uid* especificada. Si se encuentra y no terminó, se les asignan a las variables globales recién mencionadas los valores y direcciones contenidas en el *job* encontrado, y se asigna como *foreground_thread* la id del hilo que se estaba ocupando del trabajo. Ese hilo se encargará de la cancelación de la ejecución cuando *fg_execution_cancelled* se vuelva *true* (también comprueba que su id sea igual a la de *foreground_thread*, para solo parar si se designó como tal, sino sería un hilo de *background* y no debería parar ante un [ctrl + c]). Volviendo a la función que estaba describiendo (*execute_fg*), esta espera de forma bloqueante con *pthread_join* a que termine el hilo que se designó como *foreground_thread*. De esta forma, es bloqueante hasta que termine la ejecución del hilo del *job*. Al terminar, se reasigna *foreground_thread* de vuelta a la id del hilo original.

La función que se encarga de manejar la señal *SIGINT* es *stop_foreground_execution*. Esta se encarga de señalizar la terminación con *SIGTERM* de todos los comandos-hijos vivos restantes, desde *fg_awaited_child_cmd_i* hasta *fg_n_commands*. También, vuelve verdadera la variable global *fg_execution_cancelled* para que el hilo que se identifique como el hilo de *foreground* deje de esperar bloqueantemente, y libere la memoria que tenga que liberar. Antes de terminar la función, se crea un nuevo hilo que ejecute la función *async_delayed_force_kill*. En este hilo asíncrono, tras esperar cinco segundos, se envía la señal *SIGKILL* (9) a todos los procesos hijos del mandato. Esto es para cubrir el caso en el que algún proceso hijo tenga la señal *SIGTERM* mal configurada respecto a la respuesta esperada (terminación ordenada), y la ignore. Con esto se asegura de que no queden vivos procesos hijos que consuman recursos innecesariamente.

No se utilizó ninguna estructura de datos especial, solo se usaron bloques de memoria contiguos ("arrays" en el *heap*) a los que se le iba asignando/cambiando el tamaño con *malloc*, *realloc* y *free*, para poder albergar el número de elementos necesarios en cada momento. Tanto los trabajos, como las tuberías, como los *pids* de los hijos resultantes de cada *fork* de cada *tline* ejecutada se guardaron así, lo cual es muy rudimentario, pero es útil para adquirir destreza en el manejo manual de la memoria de *heap*.

Principales funciones

	run_line	Nombre	Tipo	Descripción
Argumentos	Argumento 1	line	puntero a tline	Contiene el puntero que apunta a una instancia estática del struct *tline* que devuelve parser.a al llamar a *tokenize*.
Variables locales	Variable 1	N_PIPES	entero sin signo const	Define el número de tuberías a crear, así que se inicializa como *ncommands* - 1
	Variable 2	pipes_arr	puntero a puntero a entero	Es un "array" de tuberías, cada elemento es un puntero a un "array" de dos enteros que reside en el *heap*.
	Variable 3	builtin_command_present	booleano	Se vuelve verdadero si entre todos los comandos en line se encuentra alguno que sea nativo a la *shell*. Sino es falso.
	Variable 4	input_from_file	booleano	Es verdadero si hay que tomar un fichero como *stdin*.

	Variable 5	output_to_file	booleano	Es verdadero si hay que redireccionar *stdout* a un fichero
	Variable 6	output_stderr_to_file	booleano	Es verdadero si hay que redireccionar *stderr* a un fichero
	Variable 7	exec_exit_status	entero	Contiene el resultado de la ejecución en el *foreground*. Si falla algún comando, se devuelve el código de error de este.
	Variable 8	current_pid	pid_t: alias para entero	Guarda temporalmente la id de proceso devuelta por el *fork* realizado en la iteración actual del bucle for.
	Variable 9	file	puntero a FILE	Se usa para guardar el valor de retorno devuelto al llamar a *freopen*, con el fin de comprobar si no se pudo abrir como *stdin*; o crear el fichero para recibir *stdout* o *stderr*.
	Variable 10	placeholder	pthread_t	Solo se usa para que el *pthread_create* guarde la id del hilo nuevo en una dirección válida, en vez de que suceda un *segfault* por intentar escribir en la dirección 0.
	Variable 11	ch_status	entero	Almacena el *status* codificado escrito por *wait_pid* al pasarselo por referencia.
Valor devuelto	Entero: *exec_exit_status* si se ejecuta la línea en el *foreground*. Si se ejecuta en el *background*, se devuelve 1 si se pudo crear el *pthread*, y 0 si no.			
Descripción de la función	Se encarga de procesar y ejecutar la línea previamente devuelta por la llamada a *tokenize*. Este es el punto en donde se hacen todos los *forks* y *pipes* necesarios para ejecutar los comandos, y se cierran los pipes no utilizados. Después de eso, o se esperan a los hijos en la misma función bloqueantemente (*foreground*), y después se libera el "array" de pipes y el "array" de pids de hijos, o se crea otro hilo para esperarlos ahí y liberar los recursos en ese lugar cuando se termine, mientras se sigue con el flujo del hilo principal.			

	async_add_bg_job	Nombre	Tipo	Descripción
Argumentos	Argumento 1	uncasted_args	puntero a void	Este parámetro guarda la dirección al struct que reside en el *heap* y que contiene en sí los argumentos pasados al thread. Después de que se copien sus valores contenidos en la variable *args*, se libera.
Variables locales	Variable 1	args	AddJobArgs	Contiene los argumentos, pero bajo el tipo correcto. Guarda el puntero al "array" alocado en el *heap* de pipes utilizadas, el puntero al "array" de las pids de los hijos, y *tline*. Todos estos punteros se usan para manejar la ejecución del *job* dentro de este hilo asíncrono.
	Variable 2	cmd_i	entero sin signo	La "i" del comando actual dentro del *for* que va desde 0 hasta *ncommands.*
	Variable 3	ch_status	entero	Guarda el estado codificado resultante de la ejecución del hijo i. Es escrito por *wait_pid* al pasárselo por referencia.
	Variable 4	this_job	Job	Es un *struct* local de tipo Job que va guardando los datos del *job* actual, para después insertarlo en el "array" de *jobs* compartido entre todos los hilos creados: *bg_jobs*.
Valor devuelto	-	this_job.exec_exit_status	entero	Es el código de salida resultante de la ejecución del *job* actual. Vale 0 si no hubo ningún fallo, o el valor de error del primer comando que falle.
Descripción de la función	Esta función se encarga de añadir el job al array estático global *bg_jobs*; de esperar a cada comando-hijo, y de liberar el "array" de tuberías que usó. También, maneja el posible caso en el que este *thread* se convierta en el *foreground* mediante el uso del comando *fg*, mediante varias comprobaciones.			

	execute_fg	Nombre	Tipo	Descripción
Argumentos	Argumento 1	command_data	puntero a tcommand	El *tcommand* apuntado guarda los datos del comando (argumentos).
Variables locales	Variable 1	uid	entero long	Se usa para guardar el resultado del *parseo* a long de argv[1], y con este número, posteriormente se busca al trabajo especificado que tenga la misma uid.
	Variable 2	job	puntero a Job	Guarda el resultado de la búsqueda del trabajo, que puede ser *NULL*. Si no es *NULL*, se traspasan sus datos a las variables del *foreground*.
	Variable 3	thread_return	entero	El código de retorno resultante de la ejecución del trabajo.
Valor devuelto	-	-	entero	Se devuelve *thread_return* si se encontró el trabajo y no está terminado. O 1 si: no se pudo parsear argv[1], o no se encontró el trabajo especificado por su uid, o este ya está terminado.
Descripción de la función	En esta función, se "envía" un *job* al *foreground*. Esto se hace buscando al trabajo con la job-uid especificada, sacando la id del hilo que se está ocupando de su ejecución, y usándola para designarlo como hilo principal (asignándole a las variables *fg* los datos del *job*), y llamando a *pthread_join* con argumento la id del hilo del *job*. Así, se bloquea la ejecución hasta que el hilo de ejecución de este trabajo termine, o se pulse [ctrl + c].			

	main	Nombre	Tipo	Descripción
Variables locales	Variable 1	buf	array de 2048 chars	Es solo un buffer para almacenar cada línea que escriba el usuario.
	Variable 2	cwd	array de 2048 chars	Es un buffer que se usa para la llamada a *getcwd*
Valor devuelto	-	-	entero	Resultado de la ejecución.
Descripción de la función	Esta función comprende el bucle de entrada → ejecución, propio de una shell. Pero, antes de entrar en el bucle, se inicializa el mutex que es usado para escribir y leer en *bg_jobs* sin que se pisen los distintos hilos. También, se asigna el manejador de la señal *SIGINT*: *stop_foreground_execution*, y se asigna (inicialmente) al hilo de ejecución actual como el *foreground_thread*. Después de eso, se adentra en el bucle, en donde se ejecuta la línea que se lee en cada iteración del bucle *while,* habiendo antes impreso "msh [directorio_actual]>" con *printf*. Después de ejecutar la línea, se reinicia el estado de algunas variables globales para que funcionen correctamente varios condicionales encontrados en otras funciones.			

Casos de prueba

Todos los siguientes ejemplos producen los mismos resultados y comportamiento que en la shell Bash.

Ejemplo 1:

>sleep 2 | echo "terminzdo_bzckground" | tr z a &

Ejemplo 2:

>echo eeee | tr e a | tr a z | tr z f | tr f l | tr l k

Ejemplo 3:

>echo "contenido" > fichero_STDOUT.txt (aparece el fichero con el contenido)

Ejemplo 4:

>head -n 5 < myshell.c (aparecen las 5 primeras líneas)

Ejemplo 5:

>cat < fichero_inexistente

>mv no_existo >& SOY_UN_FICHERO_QUE_CONTIENE_STDERR

Ejemplo 6:

>sleep 10

[ctrl+c] (Para porque es un *sleep* hecho en el foreground)

Ejemplo 7:

>sleep 2 | sleep 3 | sleep 2 | sleep 3 | sleep 2 | sleep 3 | sleep 2 | sleep 3 &

>[ctrl+c] (No se muere porque está en el *background*)

>jobs (Se ve el job ahí)

>fg 1 (Lo envía al *foreground*)

[ctrl+c] (Ahora sí para)

Ejemplo 8:

>sleep 1 &

>jobs (Aparece el trabajo “sleep 1” como terminado)

>jobs (Desaparece el trabajo terminado)

Comentarios personales

Al principio intenté hacer lo de ejecutar N mandatos usando solo dos tuberías, pero fue bastante complicado: tanto de programar, como de debuggear. Al final no lo hice de esa forma ya que no es algo que otorgue puntuación extra, y ya había desperdiciado varias horas intentándolo.

El *multithreading* con estado global compartido con memoria en el *heap* que hay que controlar manualmente, junto al manejo de una señal, que también usa ese estado global compartido, me causó una gran cantidad de bugs, pero al final creo que solucioné la gran mayoría. También, es mucho más difícil de debuggear programas multihilo porque no se pueden ejecutar por pasos los otros hilos, al menos de una forma fácil.

Sigue estando en desacuerdo con lo de declarar todas las variables arriba de todo, si la función es grande no se pueden ver a simple vista los tipos de ninguna variable si estás por abajo. Y, se puede acceder a cualquier variable de la función desde cualquier lugar en donde estés, se pueden leer variables sin inicializar, y no hay nada de separación en ámbitos; para funciones medianas o grandes es un claro detrimento que solo hace producir bugs más probable. Pero igualmente cumplí con eso porque es un requisito y es parte del estándar de ANSI C.

No veo nada malo del diseño de la práctica, así que no se me ocurre que proponer, solo me molesta lo de declarar todo arriba y rellenar la plantilla.

Realicé el trabajo en un transcurso de tiempo de ocho días.