Lab Shell: parte 3, challenges promocionales y código fuente final.

Dylan Alvarez, 98225
1º cuatrimestre, 04/05/2018

Parte 3: Redirecciones

Flujo estándar ☆☆

- **Entrada y Salida estándares a archivos** *(`<in.txt >out.txt`)*

Dentro del caso `REDIR` (función `exec_cmd` en `exec.c`) se redirigieron `stdin`, `stdout` y `stderr` hacia `cmd->in_file`, `cmd->out_file` y `cmd->err_file`, respectivamente. Luego se cambió el tipo de `cmd` a `EXEC`, y se volvió a llamar a `exec_cmd`.

Ahora bien, este reemplazo se realizó dentro de la función `replace` en `exec.c`, y consistió en: abrir el archivo con `open`, con flags acorde a si se va a escribir o leer del archivo, y luego llamar a `replace_fd` (wrapper de `dup2`) con el file descriptor correspondiente a (por ejemplo) `stdin`, y el file descriptor retornado por `open` para (por ejemplo) `cmd->in_file`.

- **Error estándar a archivo** *(`2>err.txt`)*

Esto fue resuelto con la implementación del paso anterior, con el uso de `cmd->err_file`.

- **Combinar salida y errores** *(`2>&1`)*

```
$ ls -C /home /noexiste >out.txt 2>&1

$ cat out.txt
---????---
```

- Investigar el significado de este tipo de redirección y explicar qué sucede con la salida de `cat out.txt`. Comparar con los resultados obtenidos anteriormente.

Esencialmente, lo que hace es redireccionar a un file descriptor en lugar de a un nombre de archivo. Lo que se implementó fue en la función `replace` chequear si el primer caracter del "nombre de archivo" dado es `'&'`, y si es ese el caso, se pasa directamente el entero que sigue a ese caracter a la función `replace_fd`. `out.txt` tendrá tanto lo escrito en `stdout` como en `stderr`.

- *Challenge:* investigar, describir y agregar la funcionalidad del operador de redirección **`>>`** y **`&>`**

El operador `>>` escribe al archivo en modo `append`. El parseo dado por la cátedra ante este caso da un nombre de archivo cuyo primer caracter es `'>'`. Se aprovecha esto, chequeando si es éste el primer caracter del archivo, en cuyo caso se agrega el flag `O_APPEND` a la llamada a `open`.

El operador `&>out` equivale a `>out 2>out`. En el parseo (función `parse_cmd` en `parsing.c`), transformé todo `&>` en `>&` (aprovechando que ambas formas son equivalentes). Entonces, este operador siempre va a entrar en el caso de que el "nombre de archivo" comienze con `'&'`, y que `strtol` retorne 0, ya que no se dio un file descriptor sino un nombre de archivo. Entonces, dentro de este caso se reemplaza `stdout` por el archivo dado y `stderr` por `stdout`. Nótese que `0` es el file descriptor de `stdin`, al que no se puede escribir, por lo que es válido asumir que si `strtol` retorna 0, se trata de un nombre de archivo y no un file descriptor.

Tuberías simples *(pipes)* ☆☆☆

- Investigar y describir brevemente el mecanismo que proporciona la syscall `pipe(2)`, en particular la sincronización subyacente y los errores relacionados.

La syscall `pipe` toma un área de memoria y lo trata como un "archivo virtual". Da un file descriptor de lectura y uno de escritura. Entonces, un proceso puede leer de este archivo virtual y otro escribir a él. Cuando uno quiera leer y el otro no haya todavía escrito, va a quedar bloqueado hasta que esto suceda. Un `close` del file descriptor de escritura de este archivo virtual (o la finalización del proceso que lo tenía abierto) va a dar un end of file a quien esta leyendo este 'archivo'. La llamada a `pipe` falla si el buffer que se le da para que escriba los dos file descriptor está en un área inválida del espacio de direcciones del proceso, si hay muchos file descriptors activos o la tabla de archivos está llena.

La implementación de tuberías consistió en: llamar a `pipe`, hacer un fork, y reemplazar (según si soy el proceso padre o el hijo) stdout por el file descriptor de escritura del pipe, o stdin por el file descriptor de lectura del pipe, en ambos casos cerrar el file descriptor no usado del pipe, y llamar, respectivamente, a `exec_cmd` para la instrucción izquierda o la derecha.

Challenges promocionales

Pseudo-variables ★

- Implementar **`?`** como única variable *mágica* (describir, también, su próposito).

La variable mágica `'?'` contiene el último código de estado retornado por un proceso ejecutado en primer plano. Esencialmente me permite ver si el proceso ejecutado falló o no. Este código de estado estaba ya siendo almacenado en una variable global `status` (definida en `runcmd.c`), y modificado por la llamada a `waitpid` en dicho archivo. Esencialmente se chequeó en `expand_environ_var` (en `parsing.c`) si la variable es `"?"`, y se expandió al valor de `status`.

- Investigar al menos otras dos variables mágicas estándar, y describir su propósito.

La variable mágica `'$'` es el process id de la shell, `'!'` es el process id del comando ejecutado en segundo plano más reciente, `'0'` es el nombre de la shell o el script de shell (esencialmente es argv[0] de la shell).

Tuberías múltiples ★★

Dado el diseño para tubería de 2 procesos, solo fue necesario, al parsear el lado derecho del comando de tipo PIPE, volver a parsearlo con la misma función (de modo tal que si también debería ser de tipo PIPE, no pase a parsearse como tipo EXEC). Es decir, termina siendo una secuencia de llamadas recursivas a `parse_line` (en `parsing.c`).

Segundo plano avanzado ★★★

Preguntas:

- Explicar detalladamente cómo se manejó la terminación del mismo.

Al ejecutar un comando en segundo plano (`run_cmd`, en `runcmd.c`), se almacenó en dos variables globales `last_back` y `last_back_pid`, respectivamente, el comando ejecutado y su process id. Además, en `main.c` se agregó como primera instrucción un llamado a la función `pay_attention_to_your_children`, donde se asocia mediante `sigaction`, el handler `listen_to_your_dying_child` a la señal `SIGCHLD`. Se usaron los flags: `SA_SIGINFO` para recibir como parámetro del handler el process id de quien acaba de morir, y `SA_RESTART` para evitar que se rompa la funcionalidad deseada si la señal llega durante una syscall. Dentro del handler `listen_to_your_dying_child` se chequea que el id de proceso del hijo muerto sea `last_back_pid`, en cuyo caso se escribe en un string global `preprompt` el mensaje de que finalizó un proceso en segundo plano. Ese string es chequeado antes de escribir cada prompt, y si contiene algo se lo escribe en `stdout` (tras lo cual se lo vacía de nuevo).

- ¿Por qué es necesario el uso de señales?

Es necesario porque no se puede bloquear el proceso padre hasta que muera el hijo (ya que de ese modo no se

correría en segundo plano). Eso nos deja dos posibilidades: o periódicamente se chequea el estado del proceso cuyo pid está almacenado en `last_back_pid`*, o se asocia el evento de su muerte a un handler mediante el uso de señales.*