Sistemas Operativos 2020

Laboratorio 1: Crash

Revisión 2020, Marco Rocchietti, Facundo Bustos
Revisión 2019, Milagro Teruel, Marco Rocchietti, Ignacio Moretti
Revisión 2011, 2012, 2013, Carlos S. Bederián
Revisión 2009, 2010, Daniel F. Moisset
Original 2008, Nicolás Wolovick

Objetivos

- Utilizar los mecanismos de **concurrencia** y **comunicación** de *gruesa granularidad* que brinda UNIX.
- Comprender que un intérprete de línea de comandos refleja la arquitectura y estructura interna de las primitivas de comunicación y concurrencia.
- Implementar de manera *sencilla* un intérprete de línea de comandos (*shell*).
- Utilizar **buenas prácticas de programación**: estilo de código, tipos abstractos de datos (TAD), prueba unitaria (*unit testing*), prueba de caja cerrada (*black box testing*), programación defensiva; así como herramientas de *debugging* de programas y memoria.

Objetivos de implementación

Codificar un ***shell*** al estilo de *bash* (***B**ourne **A**gain **SH**ell*) al que llamaremos ***crash.*** El programa debe tener las siguientes funcionalidades generales:

- Ejecución de comandos en espera y concurrente pasando todos los parámetros correspondientes.
- Redirección de entrada y salida.
- *Pipe* entre comandos.

Debería poder ejecutar correctamente los siguientes ejemplos:

```
ls -l crash.c
ls 1 2 3 4 5 6 7 8 9 10 11 12 ... 194
wc -l > out < in
/usr/bin/xeyes &
ls | wc -l
```

En particular deberán:

- Implementar los comandos internos `exit` y `cd`.
- Poder salir con `CTRL-D`, el caracter de fin de transmisión (EOT).
- Aceptar entrada redirigida, es decir:

  ```
  echo -en "ls\nexit\n" | ./crash
  ```

  tiene que listar el directorio actual y salir.
- Ser robusto ante entradas incompletas y/o inválidas.

Para la implementación se pide en general:

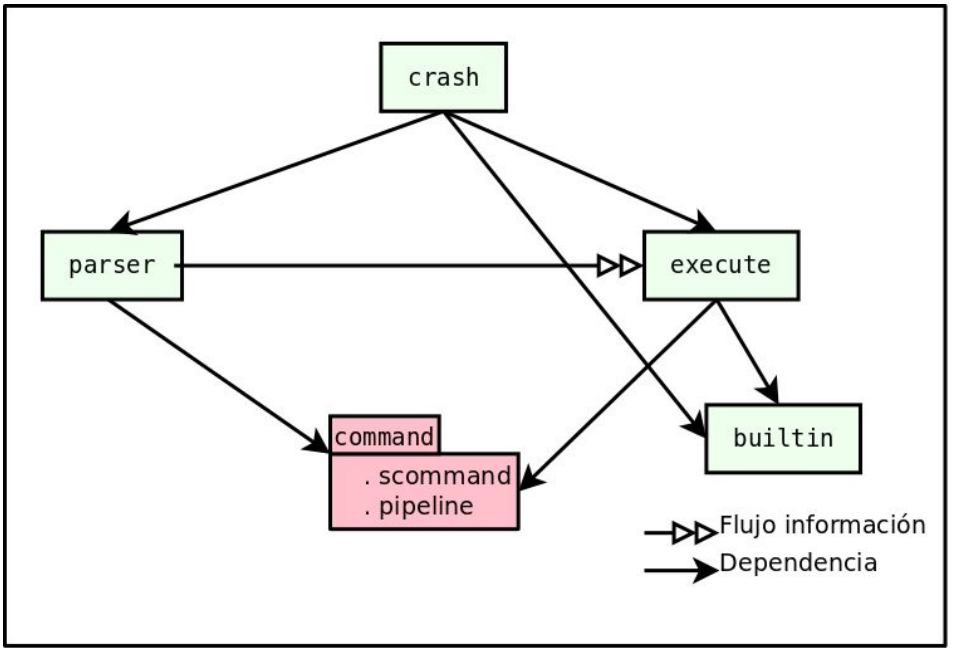
- Utilizar TAD opacos.
- No perder memoria, salvo casos debidamente reportados en el informe (de bibliotecas externas, por supuesto).
- Seguir los estándares de calidad de código de la materia (ver código fuente de *ksamp*).

Modularización

Se propone una división en 3 módulos. Uno estructural, y dos funcionales:

- Pipeline (estructural)
- Parser (funcional)
- Ejecutor (funcional)

El primero define un TAD *pipeline* a través del cual se pasan datos a los otros módulos. El módulo principal *crash* utiliza estos tres módulos. Además un módulo auxiliar *builtins* que servirá para facilitar la ejecución de comandos internos.



El módulo principal *crash*, invoca alternadamente en un ciclo al *parser* y al ejecutor.
El parser toma un *string* del *standard input* y devuelve una instancia de `pipeline` con toda la información interpretada de la secuencia de caracteres.
Finalmente el módulo `execute` toma la instancia de `pipeline` y procede a realizar los `{fork, execvp, wait, dup, pipe}` necesarios. O bien, en caso de comandos internos, invoca a `builtin`.

Notar que esta modularización funcional depende fuertemente del TAD `pipeline`, por lo tanto este módulo debe ser implementado rápida y correctamente.

Adicionalmente se incluye el módulo `strextra.h` donde se declara la función `strmerge()` que deberán implementar en `strextra.c`. Esta función será de utilidad para implementar las funciones `*_to_string` de los TADs `scommand` y `pipeline`.

TADs `pipeline` y `scommand`

A partir de la lectura de `man bash` en su sección `SHELL GRAMMAR`, se extrae la gramática que maneja el *shell*. Esta se divide en 3 capas: comando simple, tubería y lista, en orden creciente de complejidad. Limitaremos la implementación a los 2 primeros niveles.

- Un **comando simple** (`scommand`) es una secuencia de palabras donde la primera es el comando y las siguientes sus argumentos. Resultan opcionales dos redirectores, uno de entrada y otro de salida. Ejemplos:

```
ls -l Makefile
wc archivo.c > estadisticas.txt
```

- Una **tubería** (`pipeline`) es una secuencia de comandos simples conectados por *pipe* con un terminador que indica si el shell debe esperar la terminación del comando.

TAD	**Ejemplo**	**Tipo (estilo Haskell)**
`scommand`	`ls -l ej1.c > out < in`	`([char*],char*,char*)`
`pipeline`	`ls -l *.c > out < in	wc	grep -i glibc &`	`([scommand], bool)`

Ejemplos de *pipelines*

Entrada	**Estructura abstracta**
`cd ../..`	`( [(["cd", "../.."], "", "")], true )`

El *shell* padre espera la terminación del hijo. Un elemento de pipeline. Este único elemento tiene 2 cadenas y sin redirectores de entrada y salida.

Entrada	Estructura abstracta
`xeyes &`	`( [(["xeyes"],"","")], false )`

Un elemento en el pipeline con ejecución en 2do plano. Ese elemento no tiene redirectores ni argumentos.

Entrada	Estructura abstracta
`ls | wc -l`	`( [(["ls"],"","""),(["wc", "-l"],"","")], true )`

Sin ejecución en 2do plano, dos comandos simples conectados por un *pipeline*.

Entrada	Estructura abstracta
`ls -l ej1.c > out < in`	`( [(["ls","-l","ej1.c"],"out","in")], true )`

Vemos como el único comando simple del *pipeline* tiene redirectores.

Entrada	Estructura abstracta

`ls -l ej1.c > out < in | wc | grep -i glibc &`	`( [scomm0, scomm1, scomm2], false )`
	where
	`scomm0=(["ls","-l","ej1.c"],"out","in")`
	`scomm1=(["wc"],"","")`
	`scomm2=(["grep","-i","glibc"],"","")`

La estructura usada al máximo.

Implementación

Ambas estructuras de datos requieren utilizar internamente una lista o secuencia de *objetos*. En el primero es una secuencia de *strings* (que son `char*`) y en el segundo de `scommand`. Cualquier TAD que permita realizar estas operaciones alcanzaría para nuestros propósitos:

- Meter un elemento por detrás.
- Sacar un elemento de adelante.
- Consultar qué elemento está adelante.
- Consultar la longitud de la lista.

Dado que resulta una *mala práctica de la programación reinventar la rueda*, sugerimos el uso de alguna biblioteca de manejo de secuencias de objetos generales.

Un ejemplo de estas bibliotecas es GLib, sobre la cual se monta todo el *stack* de código de GNOME. Dentro de GLib tenemos varias implementaciones de listas que pueden ser útiles: GSList, GList, GQueue y GSequence. La diferencia radica en el tipo de operaciones que soportan, y la eficiencia en tiempo y en espacio.

Utilizando alguna implementación probada de secuencia, el resto es más o menos directo ya que el comando simple y el *pipeline* son tuplas que además de las listas contienen *booleanos* y cadenas de caracteres.

Resultan importantes las dos funciones `*_to_string` porque nos permitirán *debuggear* el resto de la implementación. Cuando tengamos dudas de lo que recibe o devuelve un módulo funcional, recurrimos a estas funciones para imprimir a la manera del shell los TAD.

Se da una *test-suite* implementada con *check* a fin de comprobar que la implementación dada tiene alguna parte de la funcionalidad esperada. Basta con hacer:

```
make test-command
```

para compilar e invocar el *unit testing* de `scommand` y `pipeline`.

¡ADVERTENCIA! Los tests nunca tienen cobertura total. Es decir, pasar los test al 100% no garantiza que el código esté libre de errores.

Parser

El primer módulo funcional consiste en ir recorriendo el `stdin` de manera lineal e ir tomando los comandos, sus argumentos, los redirectores, los *pipes* y el operador de segundo plano e ir armando una instancia del tipo `pipeline` con la interpretación de los datos de entrada.

Esta tarea se denomina *parsing* o análisis sintáctico y consiste en traducir la columna 1 a la columna 2 del ejemplo dado.

Este módulo tiene que ser especialmente **flexible** y **robusto** con respecto a sus entradas.

- **Flexible** en el sentido que no debe importar si agregamos espacios/tabs de más en cantidad arbitraria, o si usamos signos de puntuación en nombres de comando y argumentos.
- **Robusto** se refiere a que todo lo que no sea un comando válido respecto al TAD `pipeline` debe ser ignorado/informado. Por ejemplo si inyectamos ruido en la línea de comandos:

  ```
  ñsaj {}dfhwiuoyrtrjb23 b2 998374 2h231 #$L!,
  ```

  o si olvidamos alguna parte

  ```
  ls -l *.c >
  ```

  la función principal del módulo debería indicar un error y devolver un `pipeline` nulo.

La funcionalidad del *parser* es en algún sentido la inversa de `*_to_string` y que en general se debería cumplir que `parse_pipeline (pipeline_to_string (pipe)) = pipe`, más allá de los espacios.

El *parser* a utilizar, cuya interfaz está dada en el encabezado `parser.h`, será entregado por la cátedra y consiste de los módulos `parser.o` y `lexer.o`.

Notar que el TAD `Parser` toma un `FILE *` como entrada y como en *NIX, todo es un archivo, la entrada estándar también tiene un `FILE *` asociado.

Execute

El módulo final es el encargado de invocar a las *syscalls* `fork()`; `execvp()` necesaria para ejecutar los comandos en un ambiente aislado del intérprete de línea de comandos. Además tiene que redirigir la entrada y la salida antes de realizar el reemplazo de la imagen en memoria `execvp()`.

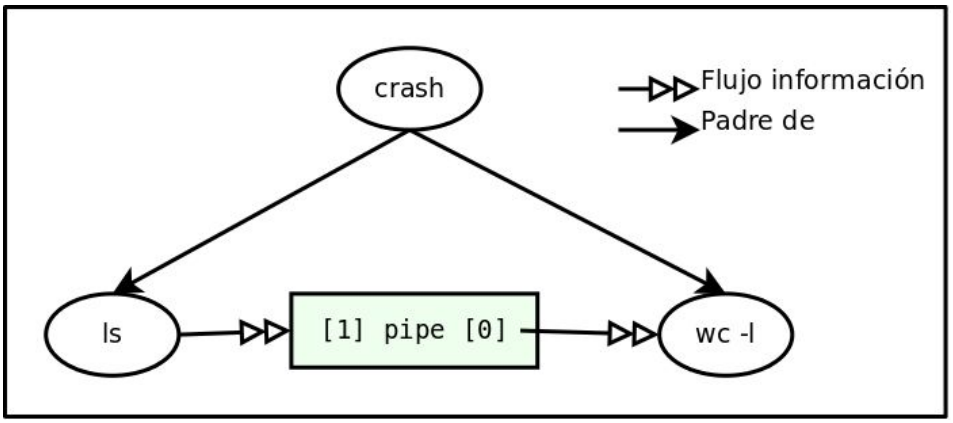
En este módulo también se arma la tubería para conectar los dos o más comandos dentro de un *pipeline*.

La primera tarea del módulo de ejecución es reconocer entre comandos internos y externos, y decidir si invocar a una función interna o a la ejecución de procesos de manera externa.

Podemos dar ejemplos de la relación entre las entradas *crash* y las *syscalls*:

Entrada	**SysCalls relacionadas**	**Comentario**
`cd ../..`	`chdir()`	El comando es interno, solo hay que llamar a la *syscall* de cambio de directorio.
`gzip Lab1G04.tar`	`fork(); execvp(); wait()`	Ejecutar el comando y el padre espera.
`xeyes &`	`fork(); execvp()`	Un comando simple sin redirectores y sin espera.
`ls -l ej1.c > out < in`	`fork(); open(); close(); dup(); execvp(); wait()`	Redirige tanto la entrada como la salida y el shell padre espera.
`ls | wc -l`	`pipe(); fork(); open(); close(); dup(); execvp(); wait()`	Sin ejecución en 2do plano, dos comandos simples conectados por un pipeline.

En el caso de los *pipes* pedimos respetar la siguiente estructura filiatoria:



Implementación

Como aquí se concentran la mayoría de las llamadas a sistema (*syscalls*), se deberá tener especial cuidado en los códigos de error que ellas devuelven e intentar manejar esta información de la mejor manera posible. Todas las *syscalls* (y también llamados a bibliotecas) pueden fallar, algunos con mayor probabilidad que otros (es de esperar que un `fork()` sea exitoso, pero no hay tanta seguridad para un `open()`).

Otro punto importante es la correcta interacción entre `fork`, `pipe`, `close`, a fin de **cerrar todas las puntas innecesarias**. Si esto es así, solo el proceso hijo estará apuntando a la entrada del *pipeline* `pipe[1]` y cuando el comando termine se produce automáticamente el cierre de todos los file descriptors. Cuando el sistema operativo recibe el último `close(pipe[1])`, induce la lectura de un `EOF` desde el proceso que está colgado a `pipe[0]` y éste puede terminar.
El síntoma más común de un `close` olvidado es un *pipeline* que queda bloqueado para siempre, esperando un `EOF` que jamás llegará.

Otros detalles a tener en cuenta:

- La adaptación del `scommand` a la estructura `char **argv` que necesita `execvp`.
- Los permisos con los cuales se abren los archivos de redirección, especialmente el de salida.

Finalmente en este módulo hay muchos detalles no especificados que deberán ser resueltos en la medida de lo posible, recurriendo a la experimentación.

¿Y los comandos internos?(módulo BUILTIN)

Hasta ahora no hablamos de los comandos internos en todo esta propuesta de modularización.

El módulo *builtin* debería tener un par de funcionalidades básicas sobre un `scommand`. La primera sería detectar si es un comando interno, mientras que la segunda es efectuar dicho comando.

Se piden solo dos comandos internos: `cd` y `exit`. El primero se implementa de manera directa con la *syscall* `chdir()`, mientras que el segundo es conceptualmente más sencillo pero requiere un poco de planificación para que el *shell* termine de manera limpia.

Aunque se piden pocos comandos, una buena implementación del módulo *builtin* debería poder soportar una cantidad arbitraria de comandos internos sin modificaciones mayores.

Kickstart

Para implementar todo lo de arriba les entregamos:

- *Headers* con una interfaz mínima a implementar. (`command.h`, `builtin.h`, `execute.h` y `strextra.h`)
- Un parser. (`parser.h`, `lexer.o`, `parser.o`)
- Un conjunto de pruebas unitarias (*unit testing*) para los dos módulos a implementar, que se puede llamar desde el `Makefile`:
  - Pruebas de command.c (`scommand` y `pipeline`): `make test-command`
  - Pruebas para todos los módulos juntos: `make test`
  - Pruebas de manejo de memoria en los módulos: `make memtest`

El código se puede bajar del sitio de la materia.

Algunos consejos

- Antes que nada completar y probar la implementación del TAD que será la interfaz de comunicación entre los módulos propuestos.
- **Dividir** el trabajo entre los integrantes del grupo. Esto servirá para trabajar en paralelo y focalizar a cada uno en una tarea particular. Luego si cada uno hizo bien su trabajo, se requerirá una etapa final de integración que no hay que subestimar en el tiempo que puede tomar.
- Hay muchísimas cosas de comportamiento no especificado, como por ejemplo qué hacer con `ls > out | wc < in`, o definir si el padre espera a todos los hijos de un pipe o solo al último. Aunque no es necesario definir todos estos detalles y hacer que nuestro *shell* se comporte de esta manera, podemos deducir el comportamiento del programa haciendo experimentos en la línea de comandos de nuestro *NIX favorito.
- Codificar rápidamente el lazo principal de entrada; *parse*; *execute* con stubs en todas las rutinas, para tener una base sobre la cual ir codificando y probando de manera interactiva.
- Tratar en lo posible de ir haciendo la integración de los módulos de manera incremental, a fin de no encontrar sorpresas en la etapa final de integración.
- Testear la implementación de manera exhaustiva, sobre todo en cuanto a su robustez. Pensar siempre que el usuario puede ser además de un enemigo, un gran conocedor de los *bugs* típicos que puede tener un *shell*.
- Recordar que el *shell* refleja tan bien las 5 *syscalls* que el módulo de ejecución deberá ser muy compacto. En caso contrario reorganicen sus ideas para lograr este objetivo.

Qué y cómo entregar

El proyecto deberá:

1. Pasar el 100% del *unit-testing* (`make test`) dado para todo el proyecto.
2. Manejar *pipelines* de dos comandos.
3. Manejar de manera adecuada la terminación de procesos lanzados en segundo plano con &, sin dejar procesos *zombies*. Pueden consultar la sección 3.4.3 de "Advanced Linux Programming", que está en la página 57, o bien en el artículo del Wikipedia acerca de Zombie process o el de SIGCHLD.

La entrega se hará directamente ingresando una revisión en el sistema de control de revisiones (*bitbucket*) que les asigna la cátedra.

Documentación

La documentación que se entregue deberá reflejar dos cosas principales: el proceso de desarrollo y el resultado final. La documentación entregada deberá estar en formato: Markdown (.md),

Tareas adicionales

Se pueden hacer las siguientes mejoras:

- Imprimir un *prompt* con información relevante, como por ejemplo nombre del host, nombre de usuario y camino relativo.
- Prompt configurable desde la variable de entorno PS1.
- Implementar todo usando la metodología Test Driven Developent (TDD).
- Implementar un parser propio.
- Implementar toda la generalidad para aceptar la gramática de list según la sección SHELL GRAMMAR de man bash. Por ejemplo se podrá ejecutar ls -l | wc ; ls & ps. Para hacer esto habrá que pensar mejor las estructuras porque pipeline incorpora el indicador de 2do plano que debería estar en list.
- Rediseñar completamente la arquitectura para tener solamente TAD y no módulos funcionales.

Material de lectura adicional

- Introducción a Unit Testing.

- Pipes, Redirection, and Filters del libro *The Art of Unix Programming*.
- Capítulo 3: Process y Capítulo 5: Interprocess Communication del libro *Advanced Linux Programming*.
- Capítulo 2: Escribiendo buenos programas GNU/Linux del libro *Advanced Linux Programming* muestra ejemplos de programación defensiva.
- Capítulo Implementing a Job Control Shell, de *The GNU C Library Reference Manual*.