

Laboratorio 1 - Sistemas Operativos 2020 - Crash

Programando un *shell* al estilo de *bash*

Marco A. Rocchietti

Universidad Nacional de Córdoba

marco.rocchietti@unc.edu.ar

Septiembre 1, 2020

Intérprete de comandos

- Es una interfaz para el usuario que permite acceder a los servicios del sistema operativo (ejecutar procesos, redireccionar entradas y salidas, etc)
- Se lo denomina *shell* ya que es una especie de caparazón entre el sistema operativo y el usuario.
- En el laboratorio anterior usamos **Bash** (**B**ourne **A**gain **S**hell)
- Nosotros ahora programaremos **Crash**... (Crash Again SHell ¿?)



Crash - la misión

Codificar un **shell** al estilo de *bash* (**B**ourne **A**gain **S**hell) e implementar las siguientes funcionalidades generales:

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

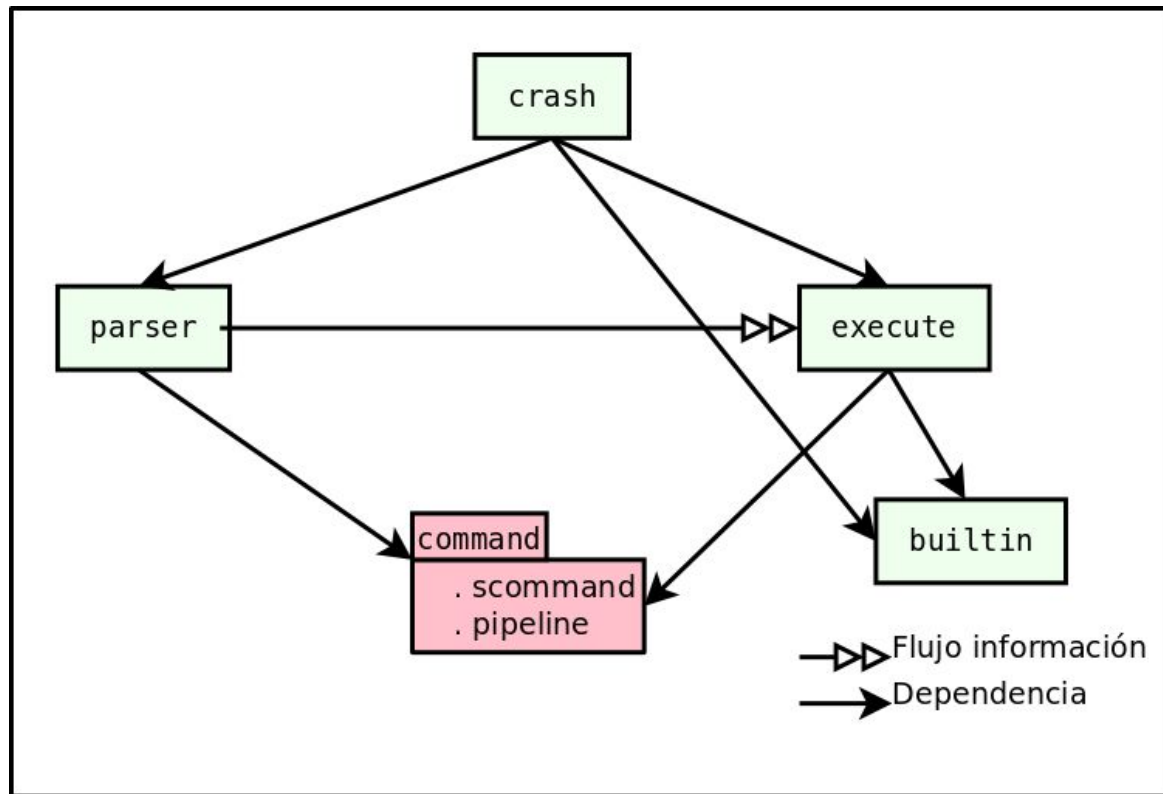
Ejemplos de ejecución

Luego de implementar lo anterior se 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`



Módulos



TADs - ejemplos y tipado

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

Manejo de strings en C

- Deberán aprender a trabajar con cadenas en C (`char *`)
- Usar las funciones definidas en la librería estándar `string.h` (`man string`)
- Adicionalmente se incluye `strextra.h` donde se declara una función `strmerge()` que deberán implementar.



Parser

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.

La interfaz del *parser* está dada en el encabezado `parser.h` y la cátedra (en su infinita generosidad) provee una implementación terminada en los módulos `parser.o` y `lexer.o`.

Cómo no todos usamos las mismas arquitecturas, se incluyen dos versiones de los módulos del parser en las carpetas `objects-i386` y `objects-x86_64`.

Si necesitaran una compilación diferente deben avisar!



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()`.



Execute - syscalls


Entrada	SysCalls relacionadas	Comentario
cd ../..	chdir()	El comando es interno, solo hay que llamar a la <i>syscall</i> 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.

Builtin

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.




Qué y cómo entregar (lo básico)

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



Adicionalmente

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 Development](#) (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.

Para el martes que viene (8/9)

Deben tener listo el módulo `command.c`

Testeen a medida que programan este módulo usando: `make test-command`

No se demoren en esa instancia ya que el módulo más interesante es *execute*



Suerte!!

