

Sistemas Operativos 2024

Laboratorio 1: MyBash

Revisión 2024, Javier Mansilla, Pablo Ventura

Revisión 2023, Pablo Ventura

Revisión 2022, Marco Rocchietti

Revisión 2021, Facundo Bustos

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 **conurrencia** 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* (*Bourne Again SHell*) al que llamaremos **mybash**. El programa debe tener las siguientes funcionalidades generales:

- Ejecución de comandos en modo *foreground* y *background*
- Redirección de entrada y salida estándar.
- *Pipe* entre comandos.

Debería poder ejecutar correctamente los siguientes ejemplos:

```
ls -l mybash.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 `cd`, `help` y `exit`.
- Poder salir con `CTRL-D`, el caracter de fin de transmisión ([EOT](#)).
- Ser robusto ante entradas incompletas y/o inválidas.

Para la implementación se pide en general:

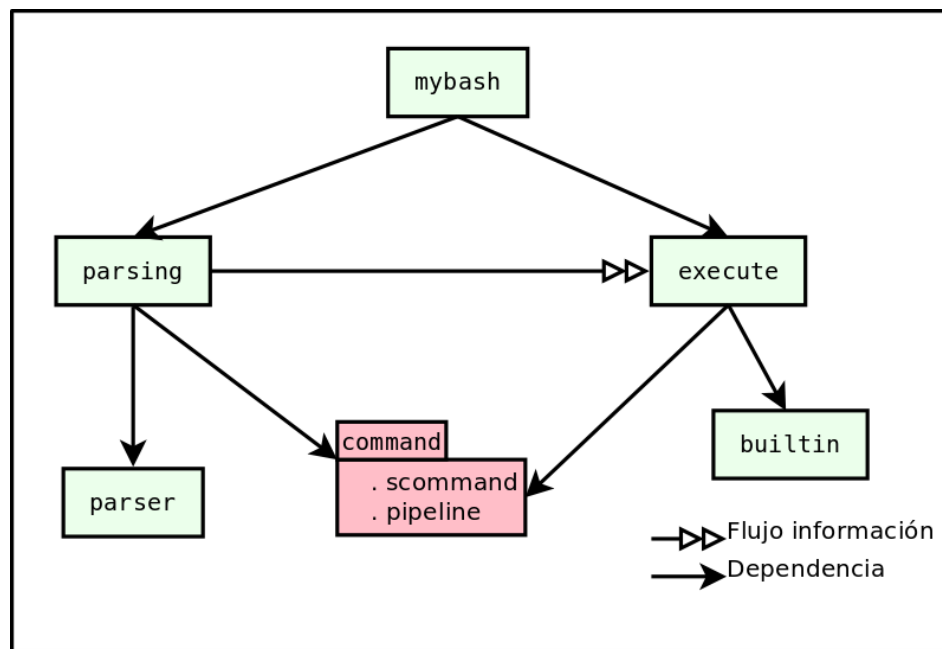
- Utilizar TADs opacos.
- No perder memoria.
- Seguir buenas prácticas de programación (*coding style*, modularización, buenos comentarios, buenos nombres de variables, uniformidad idiomática, etc).

Modularización

Para la implementación de este laboratorio se propone una división en 6 módulos:

- `mybash`: módulo principal
- `command`: módulo con las definiciones de los TADs para representar comandos
- `parsing`: módulo de procesamiento de la entrada del usuario usando un *parser*
- `parser`: módulo que implementa el TAD *parser*
- `execute`: módulo ejecutor de comandos, administra las llamadas al sistema operativo
- `builtin`: módulo que implementa los comandos internos del intérprete de comandos.

El diseño se puede ilustrar con el siguiente diagrama:



El módulo `command` define el TAD *scommand* y *pipeline* que proveen una representación abstracta de los comandos la cual es utilizada por los otros módulos. El módulo `parsing` se encarga de procesar la entrada del usuario (leída desde la *standard input*) y generar una instancia del TAD *pipeline* que contiene el comando interpretado. Para procesar la entrada debe utilizar accesoriamente el TAD *parser* que está definido en el módulo `parser`. El módulo principal `mybash`, invoca alternadamente en un ciclo al procesador de entrada y al ejecutor. Finalmente el módulo `execute` toma la instancia de *pipeline* y procede a realizar los `{fork, execvp,`

`wait, dup, pipe, etc`} necesarios, o bien, en caso de comandos internos, invoca a `builtin`.

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

Adicionalmente se incluye el módulo `strexta` donde se declara la función `strmerge()` que deberán implementar en `strexta.c`. Esta función será de utilidad para implementar las funciones `scommand_to_string()` y `pipeline_to_string()` de los TADs `scommand` y `pipeline` respectivamente.

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 el operador *pipe* (que se corresponde con el símbolo `|`) y con un terminador opcional que indica si el *shell* debe esperar la terminación del comando antes de permitir ejecutar uno nuevo.

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

Ejemplos de *pipelines*

Entrada	Estructura abstracta
<code>cd ../..</code>	<code>([(["cd", "../.."], NULL, NULL)], true)</code>

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"],NULL,NULL)], 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"],NULL,NULL),(["wc" , "-l"],NULL,NULL)], 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
sort -r < /proc/cpuinfo uniq wc -l > nlines.txt &	([sc0 , sc1 , sc2], false) where sc0 =(["sort" , "-r"],NULL, "/proc/cpuinfo") sc1 =(["uniq"],NULL,NULL) sc2 =(["wc" , "-l"], "nlines.txt" ,NULL)

La estructura usada al máximo.

Implementación

Ambas estructuras de datos requieren utilizar internamente una lista o secuencia de *objetos*. En el primero es una lista o secuencia de *strings* (que son `char*`) y en el segundo una lista o secuencia de `scommand`. Cualquier TAD que permita realizar las siguientes 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: [GSLList](#), [GList](#), [GQueue](#) y [GSequence](#). La diferencia radica en el tipo de operaciones que soportan, y la eficiencia en tiempo y en espacio.

Para instalar GLib hay que hacer:

```
$ sudo apt-get install libglib2.0-dev
```

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, 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. Para instalar *check* hay que hacer:

```
$ sudo apt-get install check
```

Para compilar e invocar el *unit testing* de `scommand` y `pipeline`, basta con hacer:

```
$ make test-command
```

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

Parsing y Parser

La tarea de estos módulos 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. El módulo `parser` ya viene implementado por la cátedra. Su interfaz está dada en el encabezado `parser.h`, y la implementación se encuentra en los binarios `parser.o` y `lexer.o`. El módulo `parsing` es el que hay que completar, donde se debe utilizar el TAD *parser* implementado en `parser` para realizar el procesamiento de la entrada. Este módulo `parsing` tiene que ser **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 `parse_pipeline()` debería indicar un error devolviendo un `pipeline` nulo.

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 último módulo tiene la tarea de invocar las *syscalls* `fork()`; `execvp()` necesarias para ejecutar los comandos en un entorno aislado del intérprete de línea de comandos. Además, debe redirigir la entrada y la salida antes de llevar a cabo el reemplazo de la imagen en memoria mediante `execvp()`.

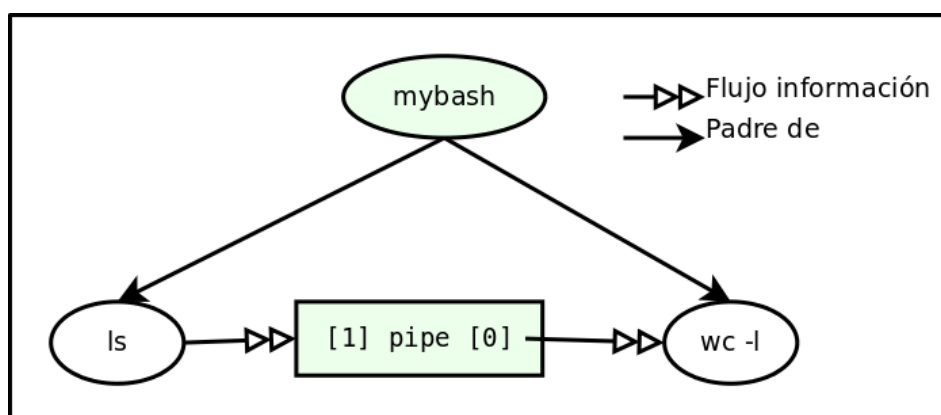
En este módulo también se arma la tubería para conectar los dos o más comandos dentro de un *pipeline*.

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 *mybash* y las *syscalls*:

Entrada	SysCalls relacionadas	Comentario
<code>cd ../..</code>	<code>chdir()</code>	El comando es interno, solo hay que llamar a la <i>syscall</i> de cambio de directorio.
<code>gzip Lab1G04.tar</code>	<code>fork()</code> ; <code>execvp()</code> ; <code>wait()</code>	Ejecutar el comando y el padre espera.
<code>xeyes &</code>	<code>fork()</code> ; <code>execvp()</code>	Un comando simple sin redirectores y sin espera.
<code>ls -l ej1.c > out < in</code>	<code>fork()</code> ; <code>open()</code> ; <code>close()</code> ; <code>dup()</code> ; <code>execvp()</code> ; <code>wait()</code>	Redirige tanto la entrada como la salida y el shell padre espera.
<code>ls wc -l</code>	<code>pipe()</code> ; <code>fork()</code> ; <code>open()</code> ; <code>close()</code> ; <code>dup()</code> ; <code>execvp()</code> ; <code>wait()</code>	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.

Builtin

El módulo *builtin* debería tener un par de funcionalidades básicas sobre un *scommand* y *pipeline*. La primera sería detectar si un *scommand* contiene un comando interno, mientras que la segunda es efectuar dicho comando. Además es de utilidad verificar si un *pipeline* contiene sólo un comando el cual además es un comando interno. Esto es porque pueden querer implementar un comportamiento distinto de los *builtins* cuando son combinados en un *pipeline* de varios comandos, al comportamiento que tendrán si son ejecutados de manera solitaria.

Se piden solo tres comandos internos:

- `cd`: Se implementa de manera directa con la *syscall* `chdir()`
- `help`: Debe mostrar un mensaje por la salida estándar indicando el nombre del *shell*, el nombre de sus autores y listar los comandos internos implementados con una breve descripción de lo que hace cada uno.
- `exit`: Es conceptualmente el 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 permitir agregar de manera sencilla nuevos comandos sin tener que modificar el diseño del código de forma significativa.

Kickstart

Para implementar todo lo de arriba les entregamos:

- *Headers* con una interfaz mínima a implementar. (`command.h`, `parsing.h`, `execute.h`, `builtin.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 de **parsing.c**:

```
$ make test-parsing
```

- 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. El contenido del kickstart se tiene que **descomprimir en el directorio raíz** del repositorio

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 ciclo 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*.

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](#).
4. Preservar las buenas prácticas de programación ya adquiridas durante los cursos anteriores (Algoritmos I y II).

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

Puntos Estrella

Se pueden hacer las siguientes mejoras:

- Hay un enunciado extra *con suspenso*, que recién será visible dentro de unos días. Para verlo, deben entrar a este [link](#) y seguir las instrucciones allí plasmadas.
 - Nota: la primera vez que entren a ese link deberán hacer click donde dice "Visit Site" en un botón azul como este 
- Generalizar el comando pipeline "`|`" a una cantidad arbitraria de comandos simples:
`scommand_1 | ... | scommand_n`
- Implementar el comando secuencial "`&&`" entre comandos simples:
`scommand_1 && ... && scommand_n`
- Imprimir un *prompt* con información relevante, por ejemplo, nombre del host, nombre de usuario y camino relativo.
- 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.
- Cualquier otra mejora que ustedes consideren relevante.

Material de lectura adicional

- [Preguntas y respuestas sobre Testing, UnitTesting y cómo funcionan los tests de este Lab.](#)
- [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*.