

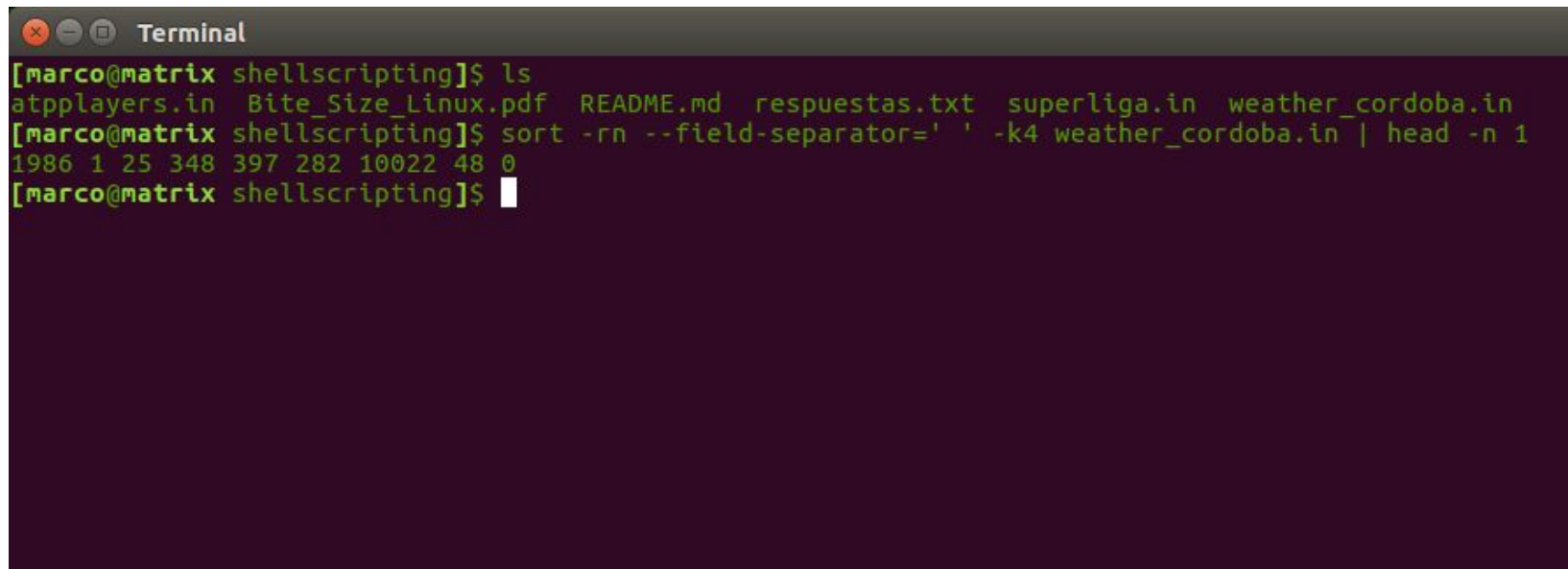
# Laboratorio 1 - Sistemas Operativos 2022 - Mybash

Programando nuestro propio *shell de linux*

Universidad Nacional de Córdoba - FAMAF

Agosto, 2022

# ¿Qué es un shell?

A terminal window titled "Terminal" with standard window controls (close, minimize, maximize). The prompt is [marco@matrix shells scripting]\$ and the background is dark purple. The user has entered two commands: 'ls' and 'sort -rn --field-separator=' ' -k4 weather\_cordoba.in | head -n 1'. The first command lists files in the current directory, and the second command sorts files by size in descending order and shows the top one.

```
[marco@matrix shells scripting]$ ls
atpplayers.in  Bite_Size_Linux.pdf  README.md  respuestas.txt  superliga.in  weather_cordoba.in
[marco@matrix shells scripting]$ sort -rn --field-separator=' ' -k4 weather_cordoba.in | head -n 1
1986 1 25 348 397 282 10022 48 0
[marco@matrix shells scripting]$
```

# ¿Qué es un shell?

- Es una interfaz entre el sistema operativo y el usuario, ya que, permite a este último acceder a los servicios del SO (ej: ejecutar comandos, redireccionar entradas y salidas, etc).
- En particular, en el laboratorio anterior utilizamos **Bash** (**B**ourne **A**gain **S**hell)
- Ahora programaremos nuestro propio shell llamado “**Mybash**”.
- Por supuesto, será un shell *hiper minimalista*.



# Mybash: Requerimientos mínimos

Se espera que cumpla al menos con las siguientes funcionalidades:

- Ejecutar comandos simples con sus respectivos parámetros en modo *foreground* y *background*.
- Soportar redirección de entrada y salida de los comandos.
- Permitir *pipes* | entre comandos (hasta 2).
- Ser robusto ante entradas incompletas y/o inválidas.
- Poder salir con CTRL-D, el caracter de fin de transmisión (EOT).



# Ejemplos de algunos comandos posibles

Se debería poder ejecutar correctamente los siguientes comandos:

- `evince -f file.pdf` *(un comando y sus argumentos en modo foreground)*
- `evince -f file.pdf &` *(igual pero en modo background)*
- `wc -l > out.txt < in.txt` *(redirección de entrada y/o salida)*
- `sleep 10 | echo "hola"` *(comandos en pipeline)*



# Mybash: Planteo

Podemos pensar nuestro bash como un programa que espera un *“string que representa un comando”* y *ejecuta efectivamente* dicho comando en el SO:

```
"wc -l > out.txt < in.txt"
```

En realidad el string se obtiene de la entrada estándar, por lo que viene línea por línea. Por lo tanto, podemos dividir el lab en dos grandes sub-funcionalidades:

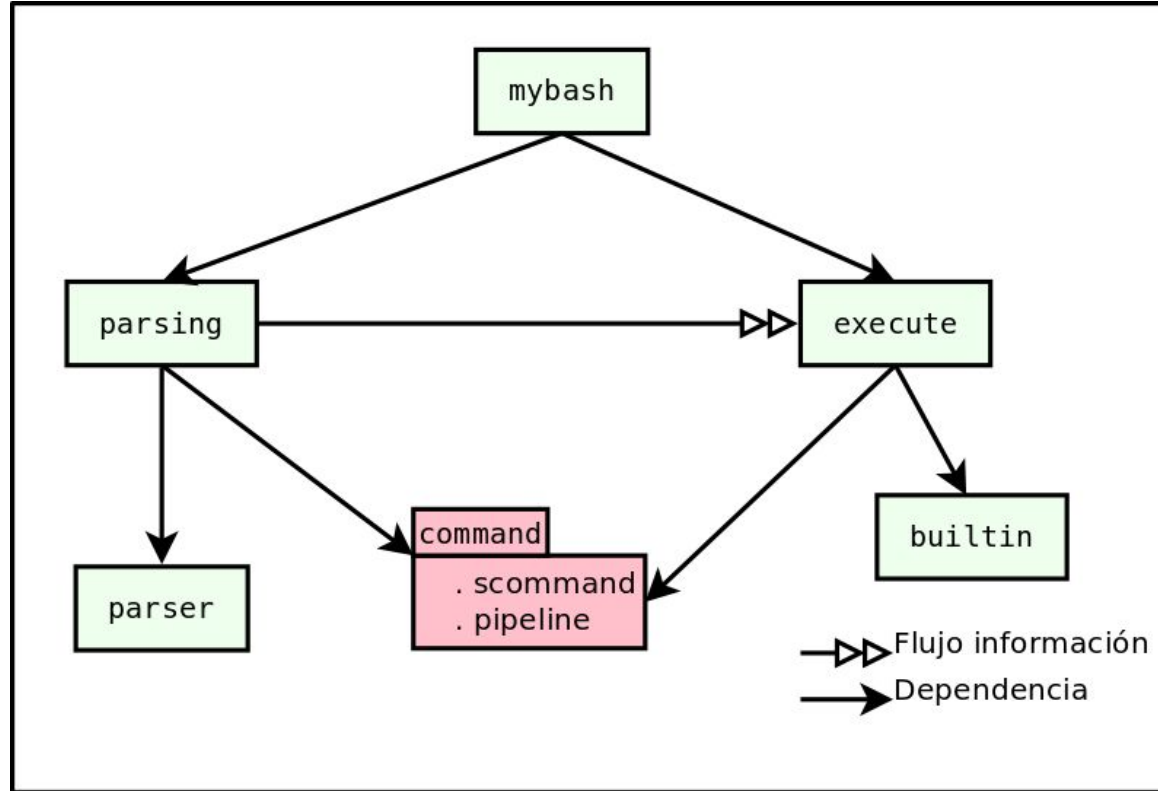
## 1) Dado un comando (un string), individualizar (parsear) cada parte del mismo:

- nombre del programa: "wc"
- sus argumentos: "-l"
- archivos de redirección de entrada y salida: "out.txt", "in.txt"

## 2) Una vez parseado el comando, ejecutar efectivamente dicho comando en el SO.

- (1) es muy parecido a lo que vinieron haciendo en Algoritmos2 (~ 1 semana)
- (2) involucra conceptos propios de SO, por lo tanto, es la parte más interesante y difícil del lab (~ 2 semanas)

# Mybash: Módulos Principales



# Mybash: Parseando comandos (1)

Nuestro bash soportará **comandos simples** y **pipelines**.

Un **comando simple** consiste de un nombre, sus argumentos y sus archivos de redirección de entrada y salida:

```
"wc -l > out.txt < in.txt"
```

TAD	Ejemplo	Representación (estilo Haskell)
scommand	wc -l > out.txt < in.txt	<code>([char*],char*,char*)</code>
		<code>(["wc", "-l"], "out.txt", "in.txt")</code>
	cd ../..	<code>(["cd", "../.."], NULL, NULL)</code>



# Mybash: Parseando comandos (1)

Un **pipeline** consiste de dos (o más) comandos simples conectados vía operador "|":

```
"ls -l | wc -l"
```

TAD	Ejemplo	Representación (estilo Haskell)
pipeline	ls   wc -l	( [scommand], bool )
		( [ (["ls"],NULL,NULL), (["wc", "-l"],NULL,NULL) ], true )
	xeyes &	( [ (["xeyes"],NULL,NULL) ], false )


# Mybash: Ejecutar efectivamente un comando (2)

Es la parte más interesante del lab porque involucra conceptos propios del área de SO.

Su implementación se espera en el módulo “execute.c” y este será el encargado de invocar a las “*llamadas al sistema (syscalls)*” necesarias para ejecutar los comandos en un ambiente aislado de nuestro bash.

Algunas syscalls que van a necesitar son:

```
fork()      /* para crear un nuevo proceso (hijo) */
pipe()      /* para crear una tubería */
dup()       /* para modificar un descriptor de archivo */
wait ()     /* para bloquear un proceso */
execvp()    /* para ejecutar un programa externo */
```



# Módulo 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.

# Manejo de strings en C

- Deberán 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 ya implementa la operación de merge entre cadenas.



# Listas de GLib

- Tanto el TAD `scommand` como `pipeline` necesitan usar algún tipo de listas (`[char*]` y `[scommand]` respectivamente).
  - 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](#) (`sudo apt-get install libglib2.0-dev`)
  - 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.
  - Usaremos GSLlist preferentemente.
- 

# Módulo Parser y Parsing

Consiste en recorrer 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 generosamente provee una implementación terminada en los módulos `parser.o` y `lexer.o`.

El módulo `parsing` es el que hay que completar, donde se debe utilizar el TAD *parser* para realizar el procesamiento de la entrada

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

Si necesitaran una compilación diferente deben avisar!



# Módulo Builtin

El módulo *builtin* encapsula todo lo referido a los comandos internos de nuestro bash.

Se encarga de detectar si se introdujo un comando interno y también sabe cómo ejecutar efectivamente cada uno de ellos:

- `cd`: Se implementa de manera directa con la syscall `chdir()`
- `help`: Debe mostrar un mensaje por la salida estandar 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

Debe ser sencillo agregar nuevos comandos internos al *shell*



# Testing

Se incluyen baterías de test para ayudarles a debuggear sus implementaciones

- 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`





# Puntos que suman

Vale la pena intentar que nuestro shell tenga las siguientes características, en orden de importancia:

1. Generalizar el operador pipeline “|” a una cantidad arbitraria de comandos simples:

```
scomand_1 | ... | scommand_n
```

2. Imprimir un *prompt* con información relevante, por ejemplo, nombre del host, nombre de usuario y camino relativo. Pueden agregar un módulo `prompt` para no sobrecargar el módulo principal si lo consideran necesario.



# Puntos Estrellas

Libre y voluntariamente pueden realizar las siguientes mejoras a su bash:

1. Implementar el operador “&&” entre comandos simples:

```
scomand_1 && ... && scomand_n
```

2. Prompt configurable desde la variable de entorno `PS1`
3. 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 repensar los TADs *scommand* y *pipeline*.
4. Cualquier otra mejora que ustedes consideren relevante.



# Sobre la entrega del lab

Se espera de su proyecto:

1. Manejar comandos *simples* con sus respectivos argumentos y sus redirecciones de entrada-salida.
2. Ejecutar comandos *pipelines* de hasta 2 comandos simples.
3. Correcta modularización del código y buenas prácticas de programación.
4. Escribir un pequeño informe que detalle cómo compilar su código y listar los comandos que ustedes probaron. Además, reportar sobre las decisiones de implementación que se tomaron durante el proceso de desarrollo de su bash y/o todo aquello que ustedes consideren relevante.

La entrega se hará directamente mediante un commit en el sistema de control de revisiones (bitbucket) que les asigna la cátedra.

Deadline: Martes 13/09/2022 hasta las 15:59 hs (ARG).

