

Segundo encuentro cercano con un SO en serio

Manejo avanzado de consola - Señales - Job control - Makefiles

Pablo De Cristóforis

Sistemas Operativos · DC · FCEyN · UBA

Segundo cuatrimestre de 2014

Temario

- 1 Qué (no) veremos hoy
- 2 Control de procesos y tareas
- 3 Entregables y Makefiles

- 1 Qué (no) veremos hoy
- 2 Control de procesos y tareas
- 3 Entregables y Makefiles

Prerrequisitos

Supondremos que a esta altura no deberían tener problemas para:

- manejar un intérprete de comandos (*shell*): `sh`, `csch`, `ksh`, `bash`.
- conectarse a una máquina por `ssh`, `sftp`, `scp`.
- saber quiénes están conectados al sistema (`who`), en qué máquina estoy (`uname`), desde cuándo está corriendo el sistema (`uptime`)
- moverse por el filesystem (`ls`, `cd`, `pwd`)
- operar con archivos y dirs (`cp`, `mv`, `rm`, `mkdir`, `rmdir`, `ln`).
- editar un archivo de texto (`nano`, `vi`, `vim`)
- escribir un `helloWorld.c`
- lograr compilarlo (`gcc`, `g++`)
- lograr ejecutarlo (`chmod u+x, ./helloWorld`)
- diferenciar entre entrada y salida normal vs. errores (`stdin`, `stdout`, `stderr`).
- redireccionar y canalizar salidas y/o entradas (`<`, `>`, `>>` y `pipe` |).
- filtrar líneas de texto (`grep`, `head`, `tail`).
- diferenciar entre `Ctrl-C` vs `Ctrl-Z`.
- buscar comandos (`whereis`, `whatis`)
- buscar ayuda: `man` (o como decimos en SO RTFM).

- 1 Qué (no) veremos hoy
- 2 Control de procesos y tareas
- 3 Entregables y Makefiles

Shells y procesos

Repasemos brevemente:

- ¿Qué es un *shell* y por qué necesito uno?
- ¿Qué shells hay? (`cat /etc/shells`), ¿Cuál estoy usando ahora? (`echo $SHELL`)
- ¿Qué es un proceso? ¿Y un subprocesso (o hilo de ejecución o hebra o *thread*)?
- ¿Qué procesos está ejecutando el sistema? (`ps -ax` vs `top`)
- ¿Qué es un proceso hijo? (`ps -f`, `ps f`).
- ¿Qué relación hay entre shells y procesos? (proceso vs. tarea, `ps` vs `jobs`)

Ante la duda: `man bash`, `man ps`, `man top`, `man jobs`.

Ejecución en segundo plano (*background*)

- Cualquier proceso puede estar en primer o segundo plano.
- Sólo un proceso puede estar en primer plano en un momento dado y es con el que estemos interactuando en ese momento.
- Un proceso que está en segundo plano no interactúa con el usuario.
- Motivación: existen tareas que no requieren de nuestro control para que se ejecuten.
- Idea: en lugar de “correrlo”, lo “ponemos a correr”.
- Pero solicitando la devolución inmediata del *prompt*.
- Eso nos permite seguir trabajando en la misma consola.
- Todos los shells ofrecen primitivas para esto.

Ejecución en segundo plano (*background*)

Ejercicio

- 1 Compare el resultado de `sleep 10` con el de `sleep 10 &`.
- 2 Compare el resultado de `yes > /dev/null &` con el de `yes > /dev/null` y luego Ctrl-Z.

Ejecución en segundo plano (*background*)

¿Cómo hacemos para traer al primer plano (*foreground*) a un proceso (esté corriendo o esté detenido)?

Con el comando `fg`. Por ejemplo, ejecutar `yes > /dev/null &` y luego `fg`.

¿Cómo hacemos para volver a poner a ejecutar un proceso detenido en segundo plano?

Con el comando `bg`. Por ejemplo, ejecutar `yes > /dev/null &`, luego `Ctrl-Z` y finalmente `bg`.

Señales en Linux

- Los procesos pueden enviarse diversos tipos de “señales”.
- No son más que un mecanismo que permite informar a los procesos de eventos que han sido provocados, por ellos mismos o por otros procesos.
- Rudimentario –de expresividad limitada– pero eficiente y omnipresente.
- Cuando le llega una señal a un proceso el sistema interrumpe la ejecución normal del proceso (o de cualquier función o llamada que este hubiera realizado) para ejecutar la función de desviación.

Señales en Linux

- Las señales se pueden general de varias maneras: excepción hardware, llamada al sistema, evento gestionado por el kernel (alarmas), interacción del usuario y el terminal (Ctrl-Z), etc.
- Las señales se representan mediante unas constantes definidas en el archivo `signal.h` y tienen el formato: `SIGXXX`.
- Pueden pensarse como una funcionalidad de *Inter-Process communication* de bajo nivel.
- En el segundo taller de la materia veremos en detalle IPC.
- Por ahora, sepamos que las señales son fundamentales para que kernel y procesos puedan inter-operar.
- Por ejemplo, necesitamos algo para matar procesos ...

Enviando señales

- Un programa puede enviar señales a otros procesos activos.
- El usuario puede enviarlas desde el shell usando `kill`.
- ¿En qué se diferencian `man 1 kill` y `man 2 kill`?

Para evitar malentendidos frecuentes, recordar que

- Las señales sirven para mucho más que matar procesos (`kill -SIGTSTP 123`).
- El comando `kill` sirve para mucho más que enviar una señal para terminar un proceso.
- Hay señales para matar procesos con mayor y menor violencia (`kill -SIGTERM 123` vs `kill -SIGKILL 123`).
- La señal `SIGKILL` debería ser el último recurso, ¿Por qué?

¿Qué señales existen en Linux?

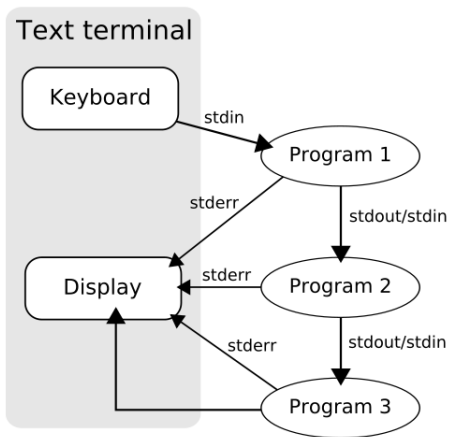
Para saber todas las señales que existen puedo ejecutar `kill -l`

```
1 SIGHUP El modem ha detectado línea telefónica colgada
2 SIGINT Interrupción de teclado, la tecla DEL ha sido pulsada
3 SIGQUIT Señal Quit (desde el teclado CTRL \)
4 SIGILL Instrucción ilegal
5 SIGTRAP Traza de los traps (excepciones)
6 SIGABRT Abortar un programa
7 SIGBUS Error en el Bus
8 SIGFPE Excepción por rebosamiento de coma flotante
9 SIGKILL Matar un proceso
10 SIGUSR1 El usuario define la señal # 1
11 SIGSEGV Violación de segmentación
12 SIGUSR2 El usuario define la señal # 2
13 SIGPIPE Escritura en pipe sin lectores
14 SIGALRM Alarma
15 SIGTERM Software genera una señal de terminación
16 SIGSTKFLT Fallo en el stack
17 SIGCHLD Cambia el estado del hijo, terminado o bloqueado
18 SIGCONT Si está bloqueado continuar
19 SIGSTOP Señal de paro
20 SIGTSTP Paro del teclado
21 SIGTTIN Proceso en segundo plano quiere leer, lectura terminal
22 SIGTTOU Proceso en segundo plano quiere escribir, escritura terminal
23 SIGURG Condición urgente
24 SIGXCPU Excedido el límite de la CPU
25 SIGXFSZ Excedido el límite del tamaño de un fichero
26 SIGVTALRM Alarma del reloj virtual
27 SIGPROF Historial del reloj
28 SIGWINCH Cambia el tamaño de la ventana
```

Tareas vs. procesos

- Proceso** o *process* es un concepto primitivo del SO, implementado a nivel del kernel, que asocia un pid único creciente (e.g. 29281) con una instancia en ejecución de un único programa (e.g. cat). Es un programa en ejecución.
- Tarea** o *job* es un concepto más general, implementado a nivel del shell, que asocia un Job ID más declarativo (e.g. [1], %1, %backup) con uno o varios procesos relacionados de cierta forma particular (por ejemplo a través de un *pipeline*). Es uno o varios *processos* que fueron ejecutados desde un shell.

Pipelines

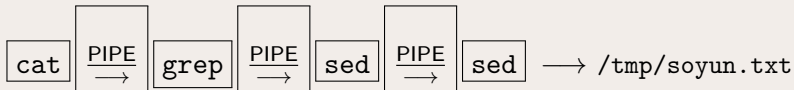


Ejemplo: un pipeline con redirección y &.

```
cat /usr/share/dict/words | grep 'otic$' \  
    | sed 's/tic$/filico/' | sed 's/^/soy /' >/tmp/soyun.txt &
```

(La contrabarra hace que el shell ignore el salto de línea y vea 1 único renglón largo.)

Abstracción de la plomería digital:



(Pipe significa *caño*. Un *pipeline* es un *ducto*, e.g. *oil pipeline* = *oleoducto*.)

Si ingresáramos el comando

```
cat /usr/share/dict/words | grep 'otic$' \  
| sed 's/tic$/filico/' | sed 's/^/soy /' >/tmp/soyun.txt &
```

una posible respuesta del shell sería

```
[1] 69704
```

y unos momentos más tarde ...

```
[1]+  Done          cat /usr/share/dict/words | grep 'otic$' | sed ..
```

Referencias del shell, de las señales y control de tareas

- `man bash`, `man kill`, `man signal`
- Guía del usuario de GNU bash
<http://www.gnu.org/software/bash/manual/>
<http://www.gnu.org/software/bash/manual/bash.pdf>

- 1 Qué (no) veremos hoy
- 2 Control de procesos y tareas
- 3 Entregables y Makefiles

Entregables y Makefiles

Normativa (extracto):

- Código legible
- C/C++ standard
- No adjuntar binarios
- Paquete autocontenido
- Recompilable en Linux con
`make clean ; make`
- Esto *no* es opcional.

Cómo usar Makefiles:

- Motivación
- Breve repaso
- Dependencias
- Sintaxis y reglas
- Convenciones
- Complicaciones
- Ejemplos y refs.

Repaso de Orga I y Orga II

compilar es el proceso por el cual a partir de uno o más archivos fuente (código en algún lenguaje humano) se genera un archivo de código objeto (código en lenguaje de máquina y tablas de símbolos).

enlazar (v., espáñglis: *linkear*) es la acción de generar un único ejecutable final a partir de uno o más archivos de código objeto; *linker* es el programa que realiza esta tarea (véase por ejemplo `man ld`).

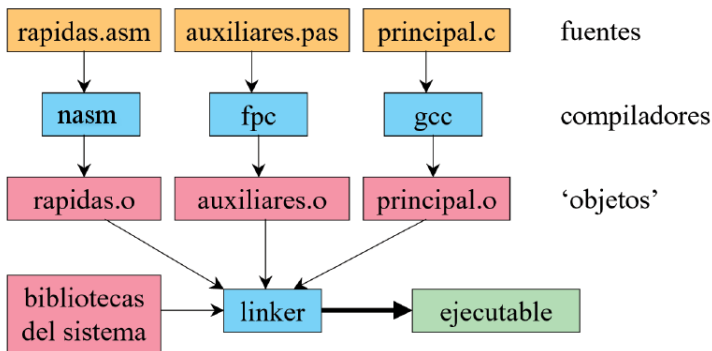
Enlazamiento estático vs. dinámico

estático El linker coloca **todo** el código de máquina dentro del ejecutable. Cuando linkeamos de esta manera, todas las referencias se resuelven en *link time*.

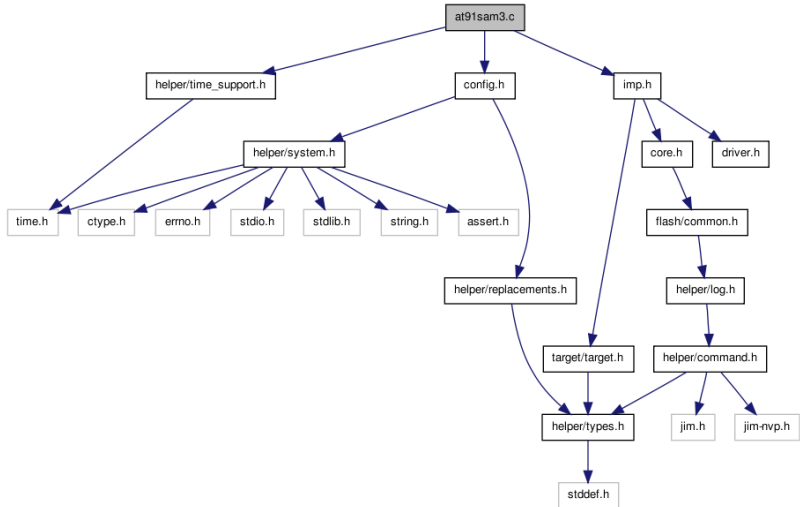
dinámico Algunas bibliotecas son “compartidas” –su código objeto no está en el ejecutable– con lo que algunas referencias recién podrán resolverse en *load time*.

Notar que, en el 2do caso, las mismas bibliotecas deberán estar presentes en un sistema para que sea posible cargar el programa.

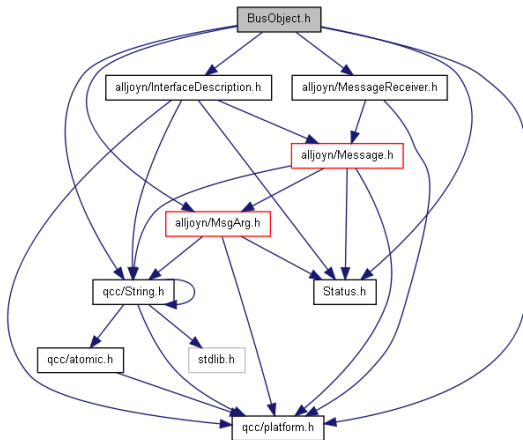
El proceso de *build* completo



Problema: rebuilds y dependencias



Dependencias en la práctica



Dependencias en el mundo real



¿Cómo incorporar make a mi TP?

- 1 Crear un archivo de texto llamado `Makefile` que *describa*
 - los targets deseados
(all, cliente, servidor, clean, entrega, ...)
 - los archivos involucrados
(archivos de código fuente, objeto, libs ...)
 - y **las dependencias** entre estas entidades.
- 2 Listo. Bastará situarse en el directorio del Makefile y decir

```
make <target>
```

para que make haga su magia.

¿Qué pinta tiene una regla genérica?

Léase: “Esto se puede generar así una vez generados tal y tal.”

```
targets ... : requisitos ...  
    [Tab]      comandos  
    [Tab]      ...
```

donde

target es el objetivo, puede ser un archivo de salida que la regla que se declare sea capaz de generar, o una acción determinada (e.g. `clean`) que se lleva a cabo cuando se invoca (e.g. `make clean`);

dependencias es uno o varios archivos de entrada necesarios para poder generar el target;

comando es una regla que, al ser ejecutada en un shell con todos las dependencias satisfechas, invoca los programas necesarios y genera el target.

Importante: todo renglón “comando” **debe** comenzar con exactamente 1 caracter Tab.

¿Se acuerdan?

suma.asm	nasm -f elf -o suma.o suma.asm
resta.asm	nasm -f elf -o resta.o resta.asm
producto.asm	nasm -f elf -o producto.o producto.asm
division.asm	nasm -f elf -o division.o division.asm
potencia.c	gcc -c -o potencia.o potencia.c
main.c	gcc -o main main.c suma.o resta.o \
	producto.o division.o potencia.o

Ejemplo

- La suma:

```
suma.o: suma.asm
```

```
    nasm -f elf -o suma.o suma.asm
```

- La resta:

```
resta.o: resta.asm
```

```
    nasm -f elf -o resta.o resta.asm
```

- El producto:

```
producto.o: producto.asm
```

```
    nasm -f elf -o producto.o producto.asm
```

Ejemplo

- La división:

```
division.o: division.asm
```

```
    nasm -f elf -o division.o division.asm
```

- La potencia:

```
potencia.o: potencia.c
```

```
    gcc -c -o potencia.o potencia.c
```

Ejemplo

- El ejecutable:
main: main.c suma.o resta.o producto.o division.o
potencia.o
gcc -o main main.c suma.o resta.o \
producto.o division.o potencia.o

El primer Makefile

```
suma.o: suma.asm  
    nasm -f elf -o suma.o suma.asm
```

```
resta.o: resta.asm  
    nasm -f elf -o resta.o resta.asm
```

```
producto.o: producto.asm  
    nasm -f elf -o producto.o producto.asm
```

```
division.o: division.asm  
    nasm -f elf -o division.o division.asm
```

```
potencia.o: potencia.c  
    gcc -c -o potencia.o potencia.c
```

```
main: main.c suma.o resta.o producto.o division.o potencia.o  
    gcc -o main main.c suma.o resta.o producto.o division.o po
```

¿Y ahora que hacemos? usuario@pc:/orga2/\$ make main

Más sobre Makefile: comodines y variables

```
suma.o: suma.asm
    nasm -f elf -o suma.o suma.asm

resta.o: resta.asm
    nasm -f elf -o resta.o resta.asm

producto.o: producto.asm
    nasm -f elf -o producto.o producto.asm

division.o: division.asm
    nasm -f elf -o division.o division.asm
```

```
%o: %.asm
    nasm -f elf -o $@ $<
```

- %: Es un *comodín*.
- \$@: El *target*
- \$<: La primer dependencia.
- \$^: Todas las dependencias.

Generalizando un poco

```
potencia.o: potencia.c
    gcc -c -o potencia.o potencia.c
```

⇒

```
%o: %.c
    gcc -c -o $@ $<
```

```
main: main.c suma.o resta.o producto.o division.o potencia.o
    gcc -o main main.c suma.o resta.o producto.o division.o potencia.o
```



```
main: main.c suma.o resta.o producto.o division.o potencia.o
    gcc -o $@ $< suma.o resta.o producto.o division.o potencia.o
```

El segundo Makefile

```
%o: %.asm  
    nasm -f elf -o $@ $<
```

```
%o: %.c  
    gcc -c -o $@ $<
```

```
main: main.c suma.o resta.o producto.o division.o potencia.o  
    gcc -o $@ $< suma.o resta.o producto.o division.o potencia.o
```

¿Y con que seguimos?

Targets especiales

Existen una serie de targets que se utilizan normalmente:

- ❶ `make`: Sin especificar target, intenta con el primero.
- ❷ `make clean`: Elimina los archivos binarios
- ❸ `make all`: Compila todo.
- ❹ `make dist`: Genera un archivo comprimido con todo el contenido compilado.
- ❺ `make install`: Instala lo compilado

Sólo haremos el 1, 2 y 3

All Clean

```
.PHONY: all clean

main: main.c suma.o resta.o producto.o division.o potencia.o
    gcc -o $@ $< suma.o resta.o producto.o division.o potencia.o

%.o: %.asm
    nasm -f elf -o $@ $<

%.o: %.c
    gcc -c -o $@ $<

all: main

clean:
    rm -f *.o
    rm -f main
```

.PHONY nos indica que all y clean NO son archivos

El Makefile posta!

```
CC=gcc
CFLAGS=-c -m32
NASM=nasm
NASMFLAGS=-f elf32
OBJ=suma.o resta.o producto.o division.o potencia.o

.PHONY: all clean

main: main.c $(OBJ)
    $(CC) -o $@ -m32 $< $(OBJ)

%.o: %.asm
    $(NASM) $(NASMFLAGS) -o $@ $<

%.o: %.c
```

CC,CFLAGS, NASM, NASMFLAGS, OBJ son variables que se remplazan por sus valores en los tarjets, las dependencias o los comandos.

Referencias (Makefiles)

- `man make`
- Manual completo de GNU make
<http://www.gnu.org/software/make/manual/>