Segundo encuentro cercano con un SO en serio

Manejo avanzado de consola - Señales - Job control - Makefiles

Federico Raimondo (Nicolás Rosner)

Sistemas Operativos · DC · FCEyN · UBA

Segundo cuatrimestre de 2013

Temario

Prerrequisitos

Supondremos que a esta altura no deberían tener problemas para:

- conectarse por `ssh`
- moverse por el filesystem
- operar con archivos y dirs.
- distinguir "allá" de "acá"
- editar un archivo de texto
- escribir un `hello.c`
- lograr compilarlo
- lograr ejecutarlo
- guardar salida a archivo
- salida normal vs. errores
- filtrar líneas de texto
- buscar comandos
- buscar syscalls
- buscar ayuda
- buscar stdlib
- RTFM

Comandos y conceptos elementales

Vamos a pasar por alto los detalles de (un conj. equivalente a):

- `ssh`, `sftp`, `scp`
- `who`, `uname`, `uptime`
- `ls`, `echo`, `cat`, `less`
- `cd`, `pwd`, `mkdir`, `ln`
- `cp`, `mv`, `rm`, `rmdir`

- `nano`, `vi` básico
- `gcc`/`g++` básico
- `chmod u+x`, `file`
- `hola` vs. `./hola`

- stdin, stdout, stderr
- Ctrl-`D`, EOF en streams
- redirecc. simple: <, > y >>
- pipe | como filtro sencillo
- `grep` básico, `head`, `tail`

- Ctrl-`C` vs. `q` de *quit*
- `apropos`, `whatis`
- `whereis`, `info`
- `man`

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.

Procesos, señales y control de tareas en `bash`

- Procesos, pids, `ps`, `top`
- Subprocesos, ppids, `init`
- Señales como IPC de bajo nivel
- `kill(1)`, `kill(2)`, `signal(3)`
- `HUP`, `INT`, `TERM`, `KILL`
- Errores frecuentes
- Segundo plano, `&`
- Ctrl-Z, `STOP`, `CONT`
- `fg`, `bg`, `jobs`
- Tareas y señales
- Pipelines y listas
- Ejemplos

Repaso de Orga I

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., espánglish: *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`).

Undefined reference y la . . .

```
$ nm /usr/bin/more
         U _PC
         U __DefaultRuneLocale
         U ___divdi3
         U ___error
         U ___maskrune
         U ___moddi3
         U ___sprintf_chk
         U ___stack_chk_fail
         U ___stack_chk_guard
         U ___stdinp
         U ___strcat_chk
         U ___strcpy_chk
         U ___tolower
         U __longjmp
00001000 A __mh_execute_header
         U __setjmp
         U _atoi
         U _calloc
         U _cfgetospeed
         U _close$UNIX2003
         U _compat_mode
         U _creat$UNIX2003
         U _dup
         U _exit
         U _fchmod$UNIX2003
         U _fclose
         U _fgetc
```

Observemos qué reporta el programa `nm` para un par de binarios distintos, como

`nm /usr/bin/more`

Notar que en muchos casos `nm` indica que existen referencias indefinidas (`U`).

Muy posiblemente sus implementaciones se encuentren en una *biblioteca compartida*.

GNU/Linux	`.so`	"shared `.o`"
Windows	`.dll`	dynamic link library
Mac OS	`.dylib`	idem.

$\Rightarrow$ Linkeo estático vs. dinámico.

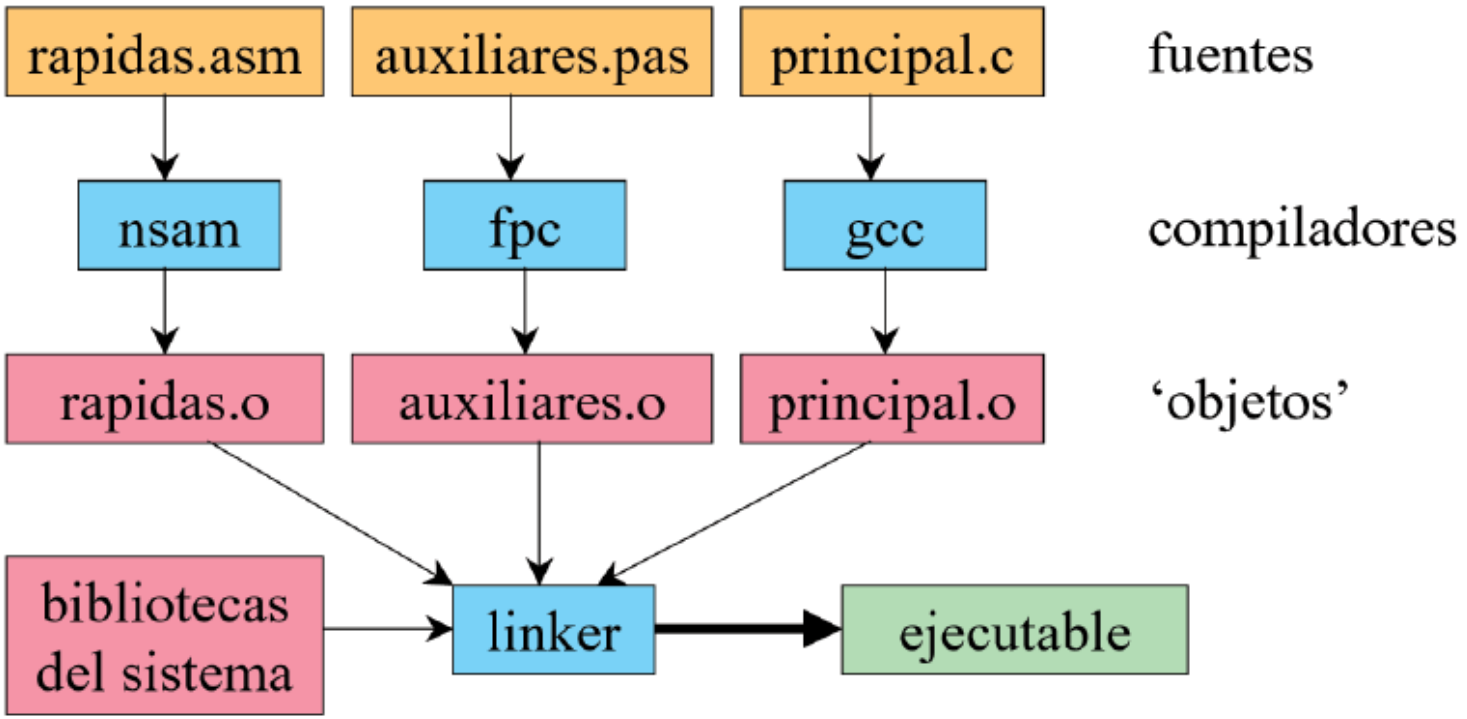
Enlazamiento estático vs. dinámico

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*.

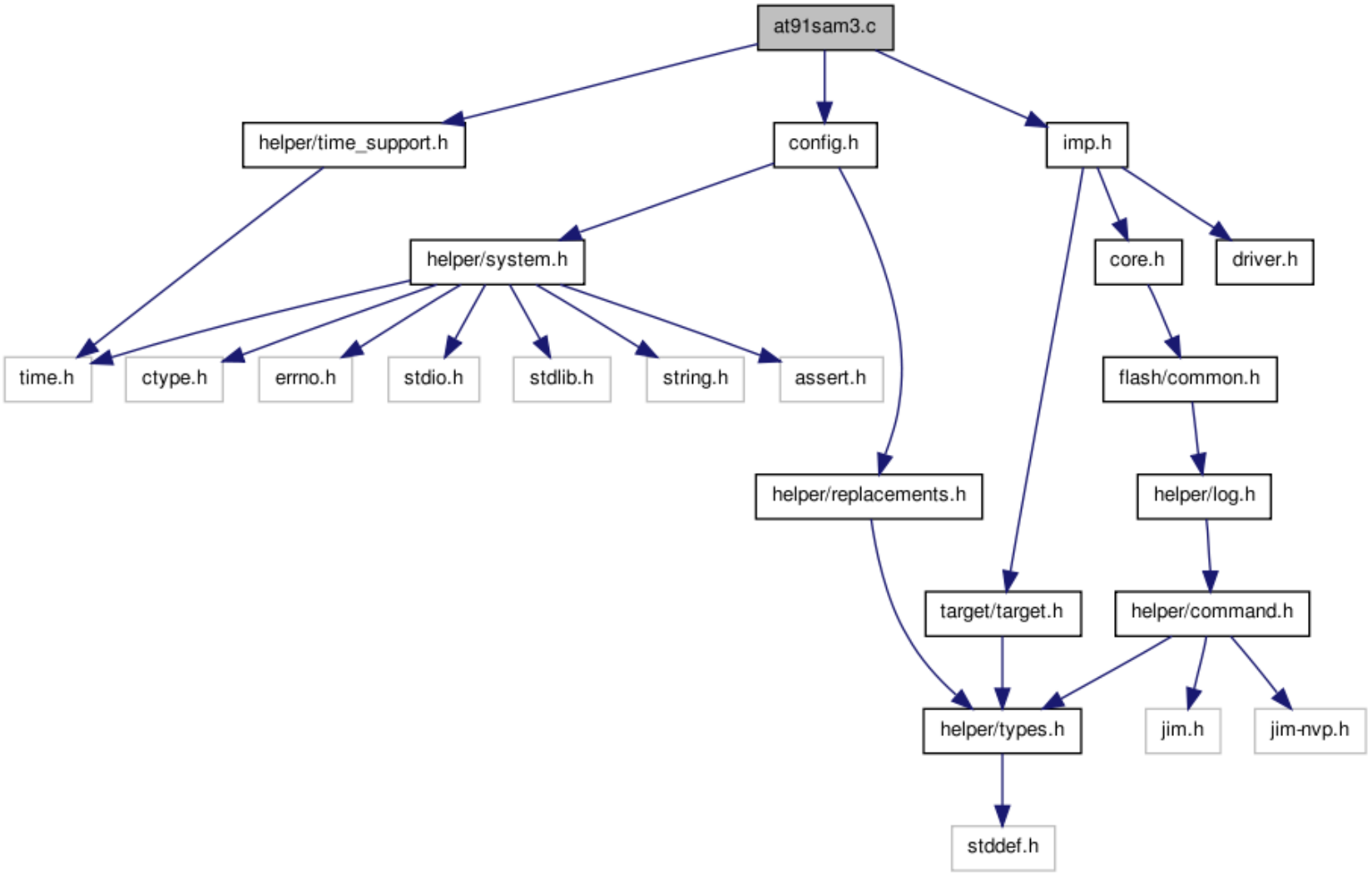
Notar que, en el 2do caso, las mismas bibliotecas deberán estar presentes en un sistema para que sea posible cargar el programa.

El proceso de *build* completo

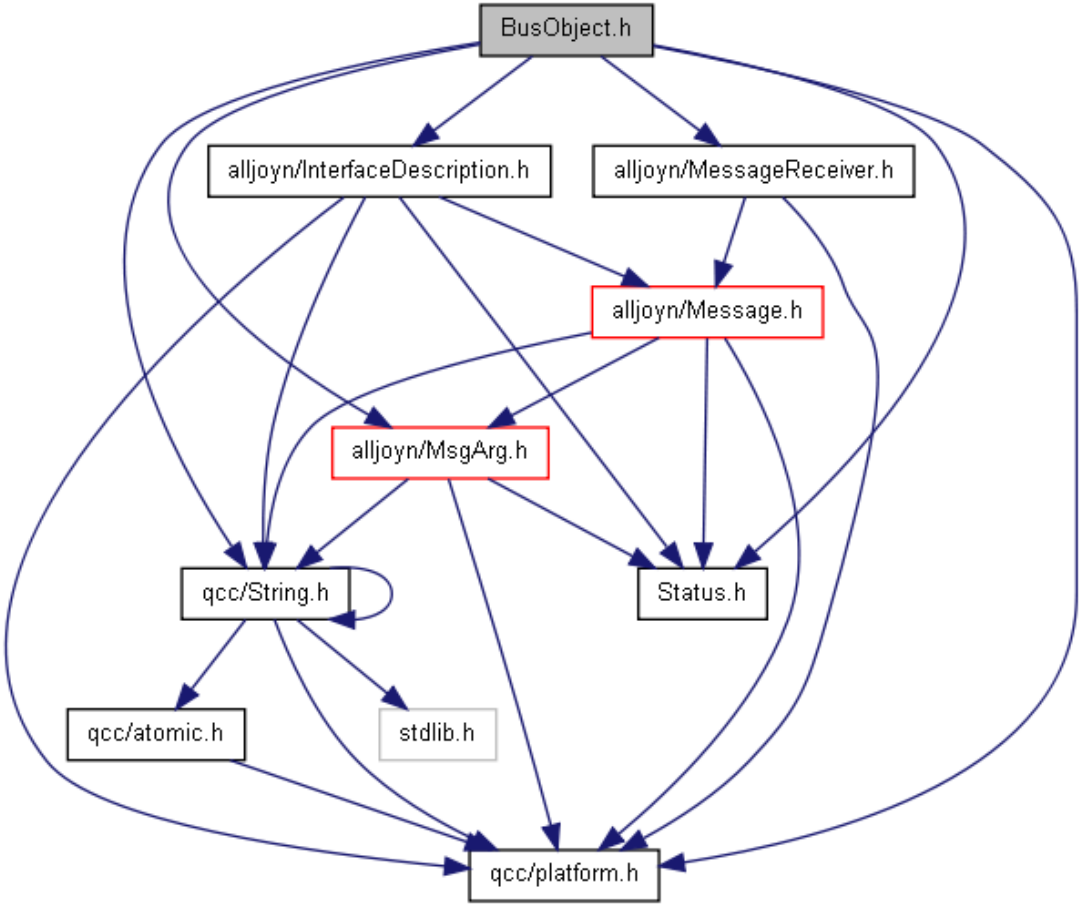


(Errata a corregir en el gráfico: donde dice *nsam* debería decir *nasm*.)

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 un archivo de salida que la regla se declara capaz de generar, o un target "trucho" (e.g. `clean`) que lleva a cabo una acción arbitraria;
- **requisito** es un archivo de entrada necesario para poder generar el target;
- **comando** es sintaxis que, al ser ejecutada en un shell con todos los requisitos satisfechos, invoca los programas necesarios y genera el target.

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

Un ejemplo bien sencillito

(Del manual de `make`, para un programa llamado `edit`.)

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Si siguiéramos avanzando . . .

- Reglas implícitas y otras convenciones
- Distintas maneras de agregar nuevas
- Variables con significados especiales
- Trucos y técnicas frecuentes
- Meta-makefiles, autotools, . . .

Nada de esto es estrictamente necesario para cumplir lo requerido por la normativa.

Referencias (Makefiles)

- `man make`
- Manual completo de GNU `make`
  `http://www.gnu.org/software/make/manual/`
- **Extreme Makefile Makeover** en 7 pasitos 7
  (disponible en la sección de descargas del sitio Web)
- Las opciones `-M`, `-MM` y similares de GCC
  (buscar dentro de `man gcc`)
- El script `cinclude2dot` (requiere GraphViz)
  `http://flourish.org/cinclude2dot/`

Shells y procesos

Repasemos brevemente:

- ¿Qué es un "shell" y por qué necesito uno?
- ¿Qué shells hay? ¿Cuál estoy usando ahora?
- ¿Qué es un proceso? ¿Y un subproceso?
- ¿Qué procesos está ejecutando el sistema?
- ¿Qué relación hay entre shells y procesos?

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

Ejecución en segundo plano

- En espánglish vernáculo: ejecución *en background*.
- 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 (y más).

Ejercicio

1. Péguele una breve mirada a `man ps` y a `man sleep`.
2. Compare el efecto de `sleep 10` con el de `sleep 10 &`.

Señales en Linux

- Los procesos pueden enviarse diversos tipos de "señales".
- Léase: *inter-process communication* de bajo nivel.
- En el próximo taller veremos en detalle el tema IPC.

Lo importante por ahora

- Un canal asincrónico "gratis" para programas (por lo demás) sincrónicos.
- Rudimentario –de expresividad limitada– pero eficiente y omnipresente.
- Fundamental 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.
- El programa `kill(1)` sirve para mucho más que enviar la señal `KILL`.
- Hay señales para matar procesos con mayor y menor violencia.
- La señal `KILL` debería ser el último recurso (¿por qué?).

¿Qué señales existen en Linux?

Según `man 3 signal` y/o `signal.h`:

Recibiendo señales

- El syscall `signal()` permite "instalar un *signal handler*".

  "A partir de ahora tal o cual señal será manejada por `miHandler()`."

- Será llamado incluso si el programa está haciendo otra cosa.

  "Mirá, mamá, ¡sin threads!" – "Ay, nene, ¡te podés caer!"

⇒ Podemos cambiar el comportamiento por defecto.

- ¿Por qué algunas señales no son interceptables?
- ¿Por qué no es trivial escribir un buen signal handler?

Tareas vs. procesos

proceso es un concepto primitivo del SO, implementado en el kernel, que asocia un `pid` único creciente (e.g. 29281) con una instancia en ejecución de un único programa (e.g. `cat`).

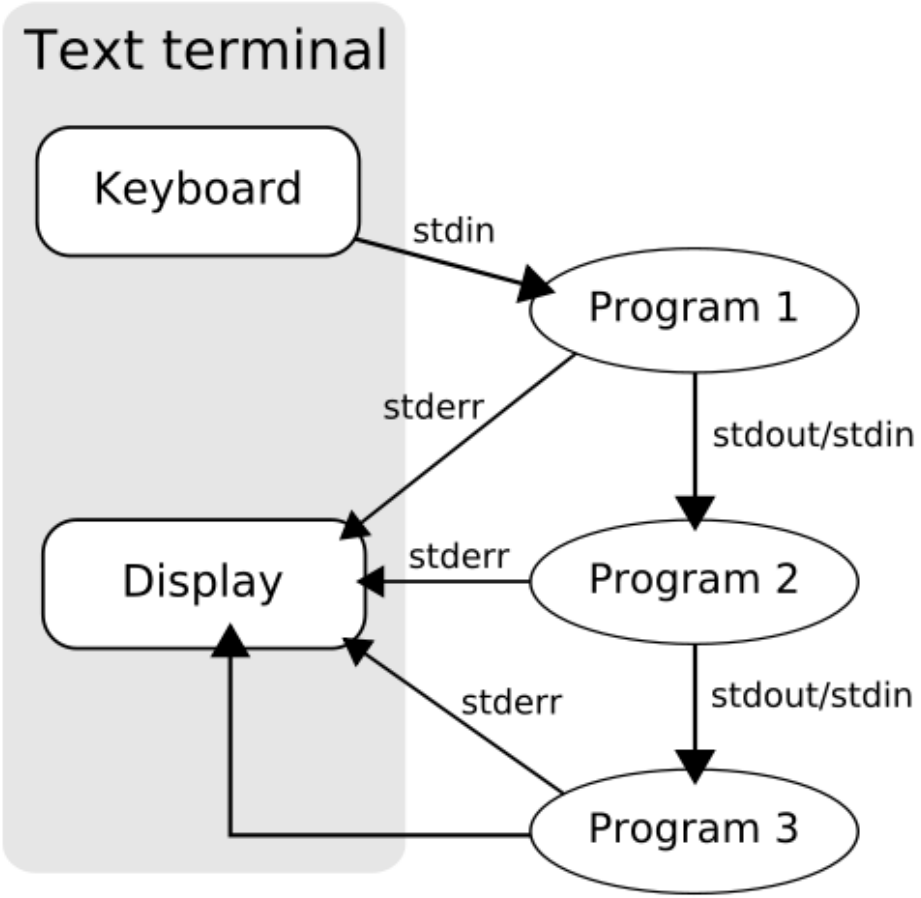
tarea o *job* es un concepto más general, implementado en el shell*, que asocia un Job ID más declarativo (e.g. `[1]`, `%1`, `%backup`) con uno o más procesos relacionados de cierta forma particular.

Permite al usuario referirse, mediante algo más fácil de recordar que *N* `pids`, a un grupo de *N* procesos vinculados de algún modo no trivial, como ser un *pipeline*.

Esto provee una abstracción un poco más flexible para referirse a "ese laburo que estoy corriendo en background mientras hago otras cosas en foreground".

(*) En realidad el kernel ofrece cierto nivel de soporte primitivo a través de *process groups* genéricos, que los shells pueden usar para su implementación de jobs y job control. Para más detalles ver "`JOB CONTROL`" en `man bash`.

Pipelines

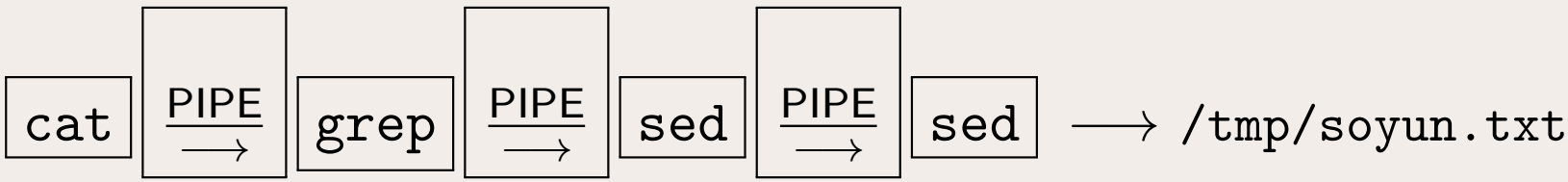

Text terminal
Keyboard
stdin
Program 1
stderr
stdout/stdin
Display
stderr
Program 2
stderr
stdout/stdin
Program 3

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 ..
```

Control de tareas en bash

Ejercicio

1. Ejecutar `sleep 10 ; echo diez &`.
2. Comparar con `( sleep 10 ; echo diez ) &`.
3. Repetir ibídem con otros valores como 20, 30, etc.
4. Usar `jobs` para listar las tareas en curso.
5. ¿Qué hace exactamente Ctrl-Z?
6. ¿Para qué sirven `fg` y `bg`?

Referencias (Shell y control de tareas)

- `man bash`
- Guía del usuario de GNU `bash`
  `http://www.gnu.org/software/bash/manual/`
  `http://www.gnu.org/software/bash/manual/bash.pdf`
- Apunte **"Primera vez con un SO en serio"** by Nacho
  (disponible en la sección de descargas del sitio Web)
- `man 1 kill`, `man 2 kill`
- `man 2 signal`, `man 3 signal`, `man 7 signal`

Eso es todo por hoy

Próximamente . . .

- Más sobre llamadas al sistema (syscalls)
- Más sobre comunicación entre procesos (IPC)
- Primeros ejercicios entregables

¿Dudas, preguntas . . . ?