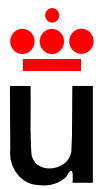
Transparencias de Sistemas Operativos

Grados en Ingeniería de Telecomunicación

Enrique Soriano y Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

Este trabajo se entrega bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" [1] (cc-by-sa).

http://hdl.handle.net/10115/20176

Usted es libre de:

- **Compartir**: *Copiar y redistribuir el material en cualquier medio o formato.*
- **Adaptar**: *Remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.*
- **Se pueden dispensar estas restricciones si se obtiene el permiso de los autores.**
- *Las imágenes de terceros mantienen sus derechos originales.*

[1] Algunos derechos reservados. "Atribución-CompartirIgual 4.0 Internacional" (cc-by-sa). Para obtener la licencia completa, véase *https://creativecommons.org/licenses/by-sa/4.0/deed.es*. También puede solicitarse a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Puedes conseguir la última versión de este documento en:

https://github.com/honecomp/honecomp.github.io/raw/main/slides/sot.pdf

1. Introducción y estructura del sistema
2. Conceptos básicos de GNU/Linux
3. Introducción a C sobre GNU/Linux
4. Procesos
5. Ficheros
6. Gestión de memoria
7. Comunicación entre procesos
8. Concurrencia básica
9. Shell scripting

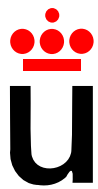
Introducción y estructura del sistema

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

Historia


Western Electric
A108A
Bell Laboratories
Western Electric
ALLENTOWN
Home of the Transistor

Historia

- 1945-1955: todo se hacía desde cero, se usan relevadores electromecánicos y después los tubos de vacío.
- 1955-1965: aparecen lenguajes de programación (FORTRAN) y sistemas operativos básicos (bibliotecas). Aparecen los circuitos integrados, sistemas por lotes.
- 1965-1980: aparecen lenguajes de programación de alto nivel (C, COBOL), minicomputadoras, multiprogramación y tiempo compartido. Los sistemas operativos evolucionan: MULTICS y después Unix.
- 1980: aparecen los ordenadores personales, redes de ordenadores, alto nivel de integración, distintos paradigmas de lenguajes de alto nivel, sistemas operativos modernos.

¿Qué es Unix?

Unix fue un sistema operativo creado por Ken Thompson y Dennis Ritchie, empezaron en 1969:

"...the number of Unix installations has grown to 10, with more expected..."
- Dennis Ritchie and Ken Thompson, June 1972

¿Qué es Unix?

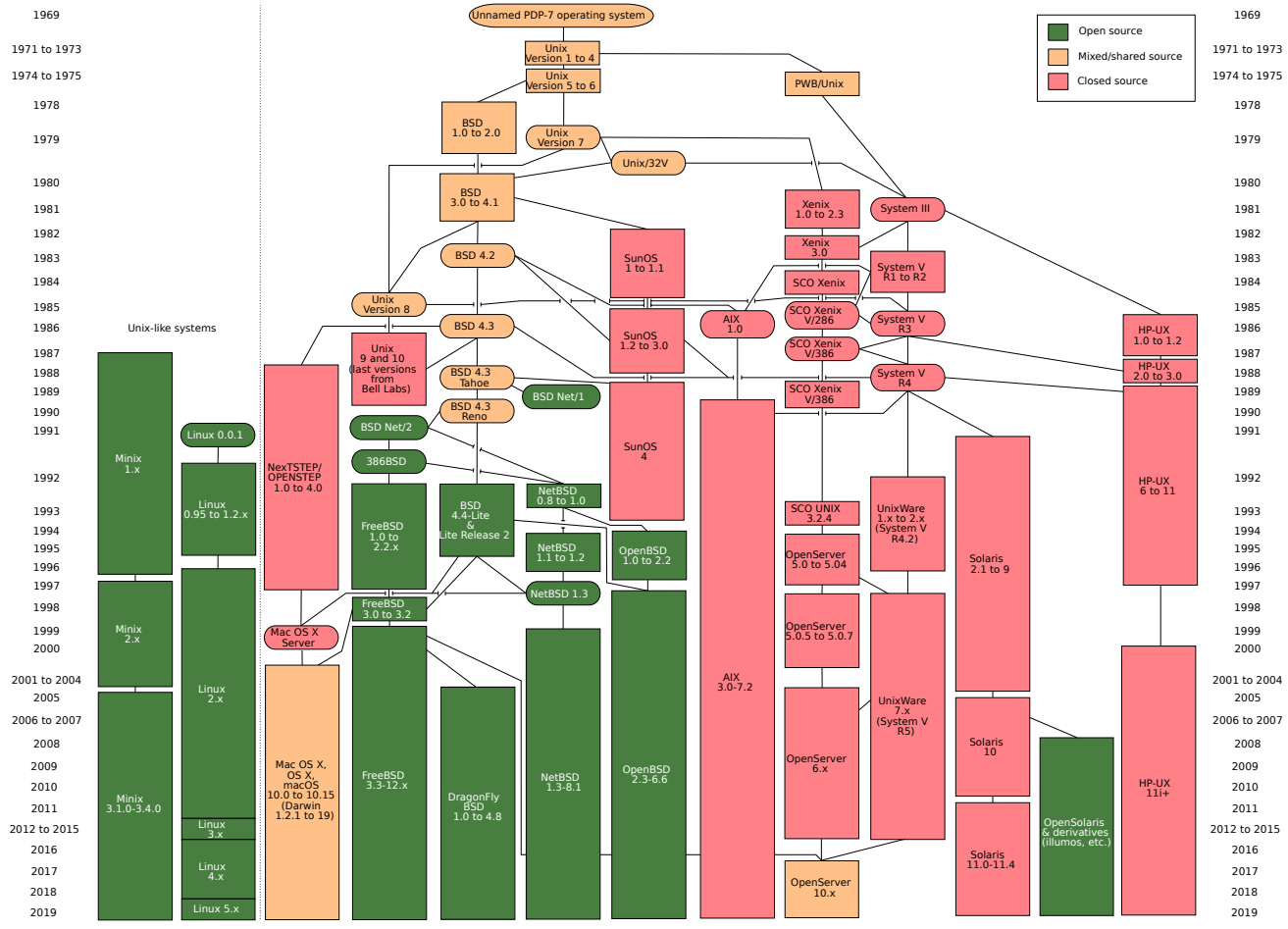
Hay muchos sistemas derivados y reimplementaciones, se llaman sistemas *Unix-like*[1]:

1BSD, 2BSD, 3BSD, 4BSD, 4.4BSD Lite 1, 4.4BSD Lite 2, 386 BSD, Acorn RISC iX, Acorn RISC Unix, AIX, AIX PS/2, AIX/370, AIX/6000, AIX/ESA, AIX/RT, AMiX, Android , AOS Lite, AOS Reno, AppleTV, ArchBSD, ASV, Atari Unix, A/UX, BBX, BOS, BRL Unix, BSD Net/1, BSD Net/2, BSD/386, BSD/OS, CB Unix, Chorus, Chorus/MiX, Coherent, CTIX, CXOs, Darwin, Debian GNU/Hurd, DEC OSF/1 ACP, Dell Unix, DesktopBSD, Digital Unix, DragonFly BSD, Dynix, Dynix/ptx, ekkoBSD, Eunice, FireFly BSD, FreeBSD, FreeDarwin, GNU, GNU-Darwin, Gnuppix GNU/Hurd-L4, HPBSD, HP-UX, HP-UX BLS, IBM AOS, IBM IX/370, Inferno, Interactive 386/ix, Interactive IS,iOS, iPhone OS, iPod OS, IRIS GL2, IRIX, Junos OS, Lites, LSX, macOS (Mac OS X), Mach, MERT, MicroBSD, MidnightBSD, Mini Unix, Minix, Minix-VMD, MIPS OS RISC/os, MirBSD, Mk Linux, Monterey, more/BSD, mt Xinu, MVS/ESA OpenEdition, NetBSD, NeXTSTEP, NonStop-UX, Open Desktop, Open Unix, OpenBSD, OpenDarwin, OpenIndiana, OpenServer, OpenSolaris, OPENSTEP, OS/390 OpenEdition, OS/390 Unix, OSF/1, OS X, PC-BSD, PC/IX,Plan 9, Plurix, PureDarwin, PWB, PWB/Unix, QNX, QNX RTOS, QNX/Neutrino, QUnix, ReliantUnix, Rhapsody, RISC iX, RT, SCO Unix, SCO UnixWare, SCO Xenix, SCO Xenix System V/386, Security-Enhanced Linux, Silver OS, Sinix, ReliantUnix,Solaris, SPIX, SunOS, Triance OS, Tru64 Unix, Trusted IRIX/B, Trusted Solaris, Trusted Xenix, TS, Tunis, UCLA Locus, UCLA Secure Unix, Ultrix, Ultrix 32M, Ultrix-11, Unicos, Unicos/mk, Unicos/mp, Unicox-max, UNICS, UniSoft UniPlus, Unix 32V, Unix Interactive, Unix System III, Unix System IV, Unix System V, Unix System V Release 2, Unix System V Release 3, Unix System V Release 4, Unix System V/286, Unix System V/386, Unix Time-Sharing System, UnixWare, UNSW, USG, Venix, Xenix OS, Xinu, xMach, z/OS Unix System Services, ...

... nosotros nos centraremos en uno llamado **GNU/Linux**

[1] Ver https://www.levenez.com/unix/

¿Qué es Unix?


Open source
Mixed/shared source
Closed source
Unix-like systems
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2015
2016
2017
2018
2019
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix Version 7
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
SunOS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
BSD 4.3
SunOS 1.2 to 3.0
AIX 1.0
SCO Xenix V/286
System V R3
SCO Xenix V/386
HP-UX 1.0 to 1.2
Unix 9 and 10 (last versions from Bell Labs)
BSD 4.3 Tahoe
BSD Net/1
System V R4
HP-UX 2.0 to 3.0
BSD 4.3 Reno
SCO Xenix V/386
BSD Net/2
SunOS 4
Minix 1.x
Linux 0.0.1
386BSD
NeXTSTEP/ OPENSTEP 1.0 to 4.0
NetBSD 0.8 to 1.0
AIX 3.0-7.2
SCO UNIX 3.2.4
UnixWare 1.x to 2.x (System V R4.2)
Solaris 2.1 to 9
HP-UX 6 to 11
Linux 0.95 to 1.2.x
FreeBSD 1.0 to 2.2.x
BSD 4.4-Lite & Lite Release 2
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
Minix 2.x
FreeBSD 3.0 to 3.2
NetBSD 1.3
OpenServer 5.0.5 to 5.0.7
UnixWare 7.x (System V R5)
Linux 2.x
Mac OS X Server
Solaris 10
OpenSolaris & derivatives (illumos, etc.)
HP-UX 11i+
Minix 3.1.0-3.4.0
Mac OS X, OS X, macOS 10.0 to 10.15 (Darwin 1.2.1 to 19)
FreeBSD 3.3-12.x
DragonFly BSD 1.0 to 4.8
NetBSD 1.3-8.1
OpenBSD 2.3-6.6
OpenServer 6.x
Linux 3.x
Linux 4.x
Linux 5.x
Solaris 11.0-11.4
OpenServer 10.x

¿Qué es un sistema operativo?

- Def.- Programas que te dejan usar la máquina, es subjetivo.
- Ventajas:
  - Similar a una biblioteca $\rightarrow$ reutilización.
  - Abstrae de la máquina: no necesitas conocer los detalles para usarla.
  - Gestiona y reparte la máquina: no necesitas preocuparte de gestionar el tiempo que ejecuta un programa, organizarle la memoria, etc.

¿Qué es un sistema operativo?

- Es una **máquina abstracta**: el sistema operativo proporciona una máquina que realmente no existe, es una máquina ficticia que nos ofrece dispositivos virtuales: ficheros, directorios, procesos, conexiones de red, ventanas...
- Hoy en día tenemos distintos tipos de software de sistemas: sistema operativo, hipervisores, contenedores, etc.

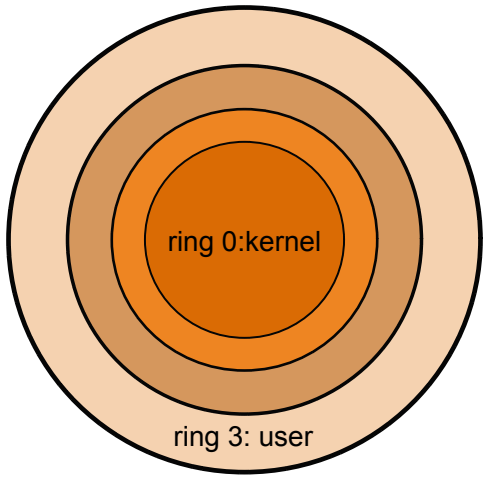
¿Qué es un sistema operativo?

Es un gestor de recursos:

- Multiplexa en tiempo: p. ej. procesador, red.
  - ¿Tienes que preocuparte de soltar el procesador en tu aplicación?
  - Ejemplo: abstracción llamada **proceso**.
- Multiplexa en espacio: p. ej. memoria, disco.
  - ¿Tienes que preocuparte de no pisar la memoria de otra aplicación?
  - Ejemplo: abstracción llamada **fichero**.

Niveles de privilegio de la CPU

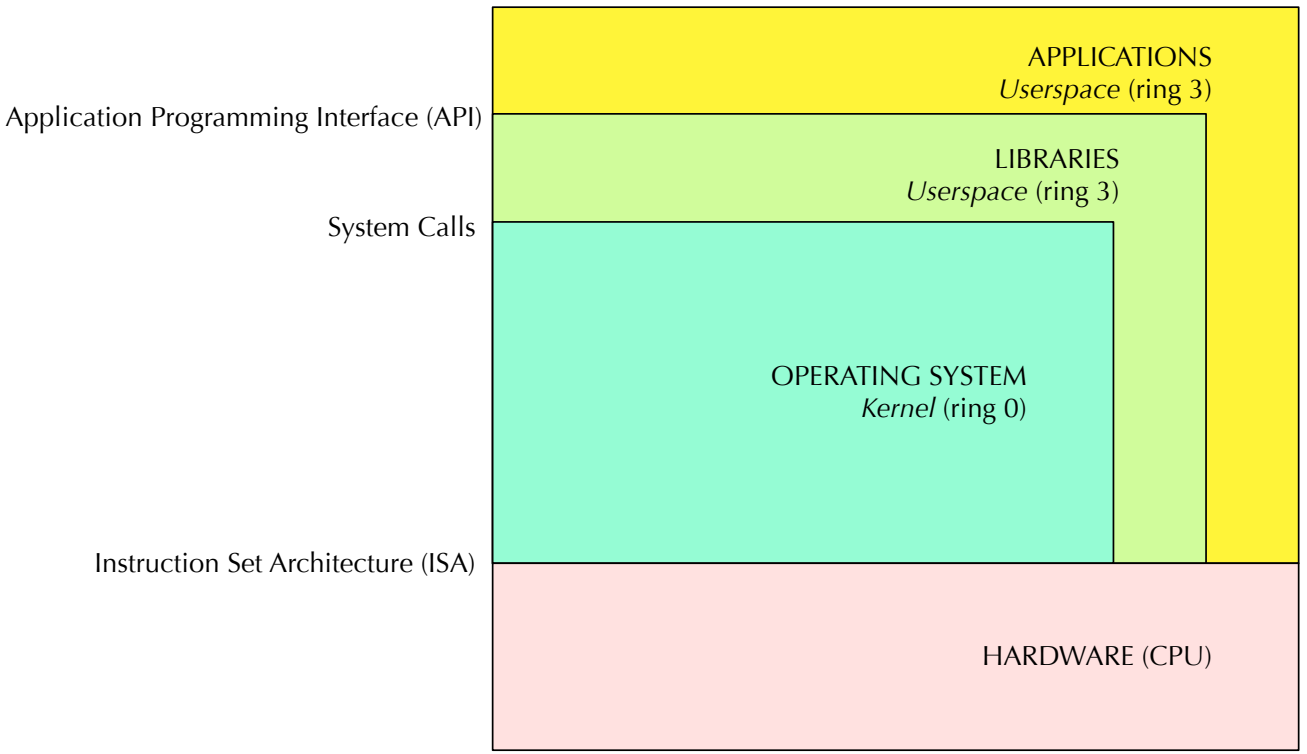
Visión clásica:



Las máquinas modernas tienen otros niveles *negativos*, el sistema operativo no es consciente de ellos:

- Ring -1: Intel VT-x, AMD-V, soporte para virtualización de sistema opertivo.
- Ring -2: System Management Mode (SMM), ejecuta el firmware de la máquina para gestionar la energía, manejar errores del hardware, etc.
- Ring -3: Intel ME, sistema que que ejecuta en una CPU separada y activo en todo momento, con acceso a toda la memoria física, a las interfaces de red, etc. Este sistema se activa antes de arrancar la CPU principal.

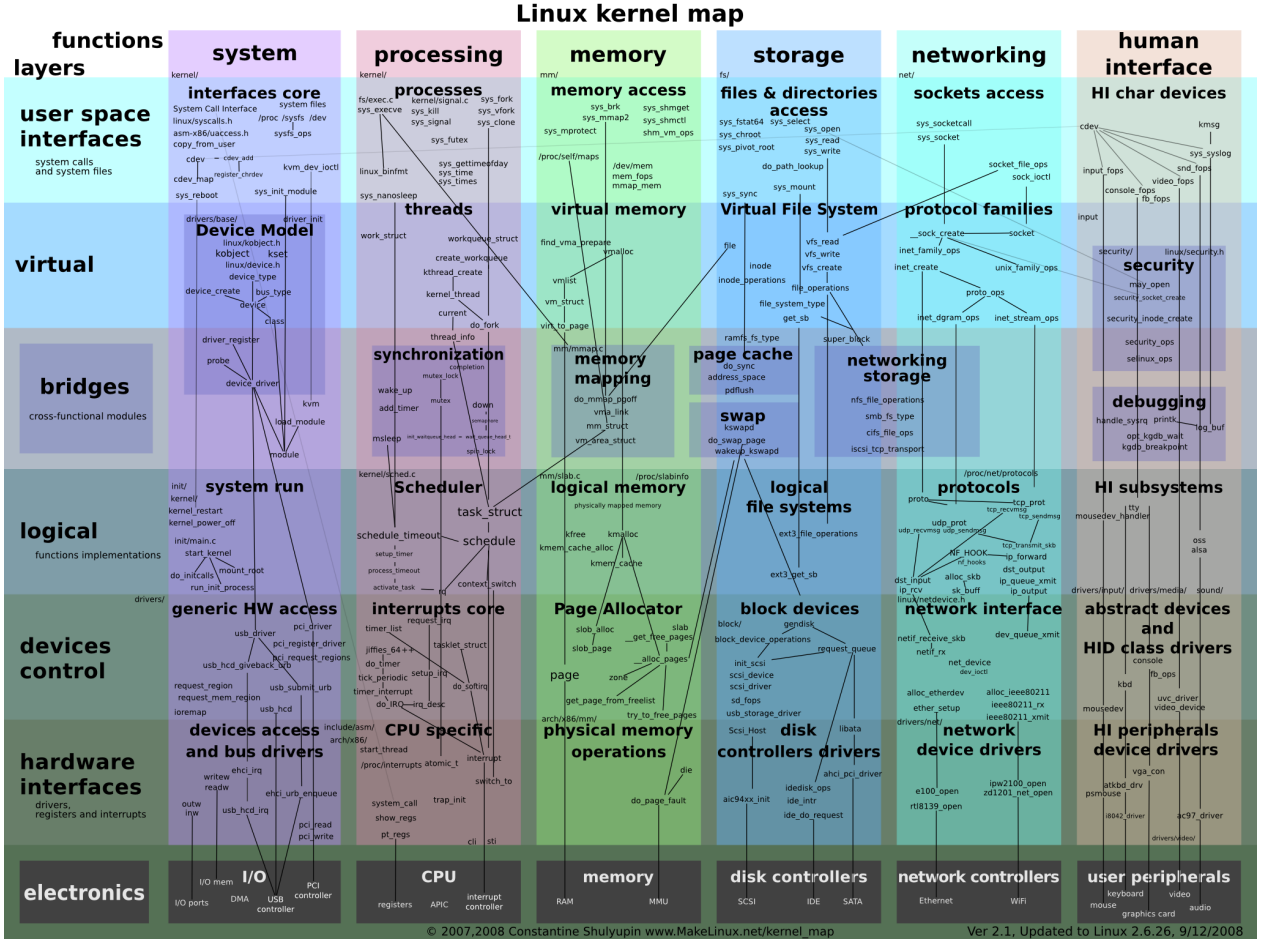
Estructura del sistema



Núcleo (*kernel*)

- Ejecución en **modo privilegiado** (ring 0): se pueden ejecutar instrucciones especiales (acceder a ciertos registros de la CPU, invalidar caches, etc.).
- Multiplexa la máquina (espacio y tiempo): implementa las **políticas y mecanismos** para repartir la CPU, memoria, disco, red,...
- Maneja el hardware: **drivers**.
- Proporciona **abstracciones**:
  - **Proceso**: programa en ejecución.
  - **Fichero**: datos agrupados bajo un nombre.
  - ...
- Da servicio al resto de programas en ejecución, que no ejecutan en modo privilegiado. Si el kernel es **reentrante**, puede dar servicio a múltiples simultáneamente. Todos los kernels de tipo Unix lo son.

Núcleo (*kernel*)



Área de usuario (*userspace, userland*)

- Así ejecutan los programas del usuario (aplicaciones, herramientas, GUI, etc.).
- Se ejecutan en **modo no privilegiado** (ring 3): no se pueden ejecutar instrucciones peligrosas.
- Piden servicio al kernel realizando **llamadas al sistema**.

Procesos y programas

- Programa: conjunto de datos e instrucciones que implementan un algoritmo.
- Proceso: programa que está en ejecución, un programa vivo que tiene su propio **flujo de control** y es independiente de los otros procesos.

Procesos

Elementos fundamentales para un flujo de control:

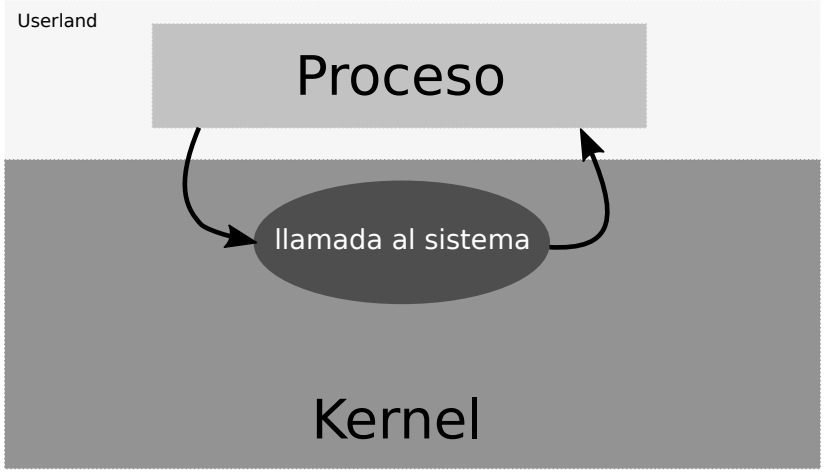
- **Contador de programa**: un registro de la CPU (PC) apunta a la instrucción por la que va ejecutando el programa.
- **Pila**: un registro de la CPU (SP) apunta a zona de la memoria donde se guardan los **registros de activación** (o marcos de pila) donde se guardan los datos necesarios para realizar llamadas a procedimiento (parámetros, variables locales, dirección del programa para retornar, etc.).

Procesos

- Procesos concurrentes: varios procesos que están ejecutando al mismo tiempo.
- El sistema operativo crea la ilusión de que cada uno tiene su propia CPU.
- Ejecución paralela vs. ejecución pseudo-paralela $\rightarrow$ para el programador es lo mismo.

Llamadas al sistema

- Es cuando el proceso *entra al kernel* por iniciativa propia, p. ej. porque necesita que el kernel realice operaciones privilegiadas para él.
- El proceso deja de ejecutar el código de la aplicación y pasa a ejecutar, en modo privilegiado, el código del kernel: se ejecuta el código del kernel *en el contexto del proceso*.
- El proceso tiene en realidad dos pilas: pila de usuario y pila kernel.



Llamadas al sistema

¿Cómo funciona? Ejemplo, Plan 9 para AMD64:

1. Coloca los argumentos en la pila del proceso.
2. Carga el número de llamada al sistema en un registro.
3. Ejecuta la instrucción `SYSCALL`, que pasa a ring 0 y salta al punto de entrada de las llamadas al sistema en el kernel (apuntado por el registro `Lstar`, configurado en tiempo de arranque) que:
   - 3.1 Cambia el Puntero de Pila (SP) para usar pila de kernel del proceso.
   - 3.2 Guarda el contexto del proceso en área de usuario en la pila de kernel.
4. Copia los argumentos de la llamada al sistema a la estructura que representa al proceso.
5. Indexa la tabla de llamadas al sistema con el número de llamada al sistema, y llama a la función que la implementa.

Arranque de la máquina

El arranque común es el siguiente:

1. Se ejecuta el firmware (p. ej. UEFI): realiza operaciones de comprobación, carga un cargador (p. ej.GRUB).
2. El cargador puede cargar otros cargadores (*stages*), hasta que uno carga el kernel y se salta a su punto de entrada.
3. El kernel ejecuta sus funciones de inicialización y configuración (hardware, estructuras, etc.).
4. El kernel crea los primeros procesos de usuario, entre ellos `init`[2].
5. El proceso `init` crea todos los procesos necesarios para arrancar servicios (demonios), shells, interfaz gráfica de usuario, etc. (dependiendo de la configuración del sistema).
6. Esos procesos van creando otros procesos, siguiendo una relación padre-hijo.

[2] Actualmente, en la mayoría de las distribuciones de GNU/Linux, se arranca `systemd`.

Estructura del OS: Tipos de kernel

Kernel monolítico:

- El kernel es un único programa.
- Pros: simplicidad, rendimiento.
- Contras: protección de los datos, puede haber falta de estructura (si se programa mal).
- Ejemplo: Linux, FreeBSD.

Estructura del OS: tipos de kernel

Microkernel:

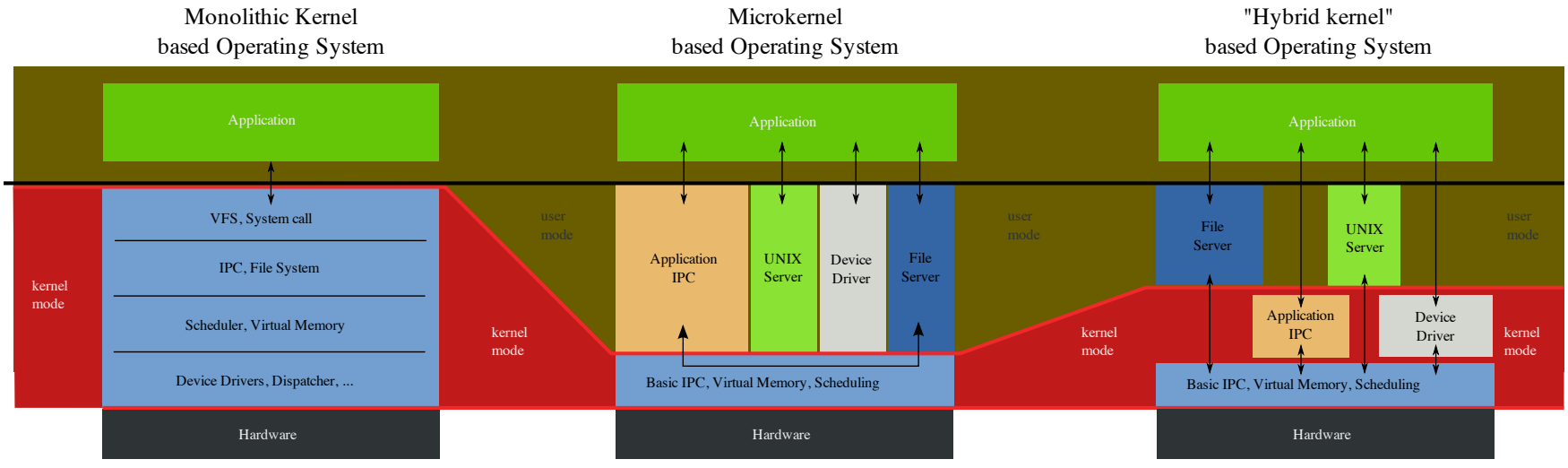
- El kernel queda reducido a lo mínimo: abstracción del HW (HAL), flujos de control, comunicación y gestión de memoria.
- El resto (*OS personality*: gestión de procesos, red, sistemas de ficheros...) se implementa en servidores independientes.
- Idea: **políticas** en espacio de usuario, **mecanismos** en espacio de kernel.
- Pros: modularidad, tolera fallos en los servidores, distribución.
- Contras: pero rendimiento en general, más complejo.
- Las nuevas generaciones tienen mejor rendimiento.
- Ejemplo: Mach, L4 y derivados.

Estructura del OS: tipos de kernel

Kernel Híbrido:

- Compromiso entre microkernel y monolítico.
- Algunos incluyen ciertos componentes en espacio de kernel: device drivers, gestión de procesos, etc. Ejemplos: Minix, QNX.
- Otros simplemente sólo siguen un diseño de microkernel, pero todos los servidores están en espacio de kernel. Ejemplos: XNU (OSX), Windows NT.

Estructura del OS: tipos de kernel



Kernel modular

La mayoría de los kernels actuales permiten la carga dinámica de módulos para ampliar/reducir su funcionalidad sin la necesidad de rearrancar el sistema.

- Ventaja: sólo se cargan los drivers necesarios $\rightarrow$ ahorro de memoria.
- Desventaja: seguridad.
- Ejemplos: Linux (.ko), Mac OSX (.kext), FreeBSD (.kld), Windows (.sys).

Módulos en GNU/Linux

Comandos:

- `lsmod` escribe en su salida la lista de módulos cargados.
- `modprobe` carga un módulo.
- `rmmod` descarga un módulo.

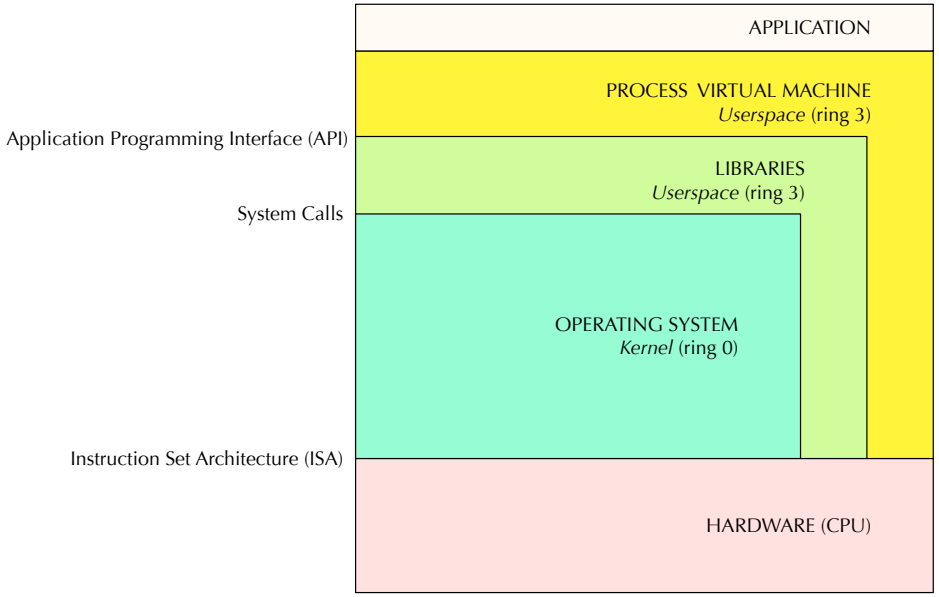
Los módulos están en `/lib/modules/`*`versión-de-kernel`*`/`

- `uname -r` escribe en su salida la versión del kernel que estamos ejecutando.

Virtualización

- La virtualización es muy común hoy en día (cloud computing, etc.).
- Existen distintos tipos de *máquinas virtuales*:
  - **Máquina virtual de proceso:** tiene como objetivo proporcionar una plataforma para ejecutar un único programa: emuladores de otra ISA (p. ej. OSX Rosetta), optimizadores, ISA virtuales (p. ej. Java VM, .NET).
  - **Máquina virtual de sistema:** un hipervisor (VMM o VM Monitor) proporciona un entorno completo y persistente para ejecutar un sistema operativo completo.

Máquinas Virtuales de Proceso

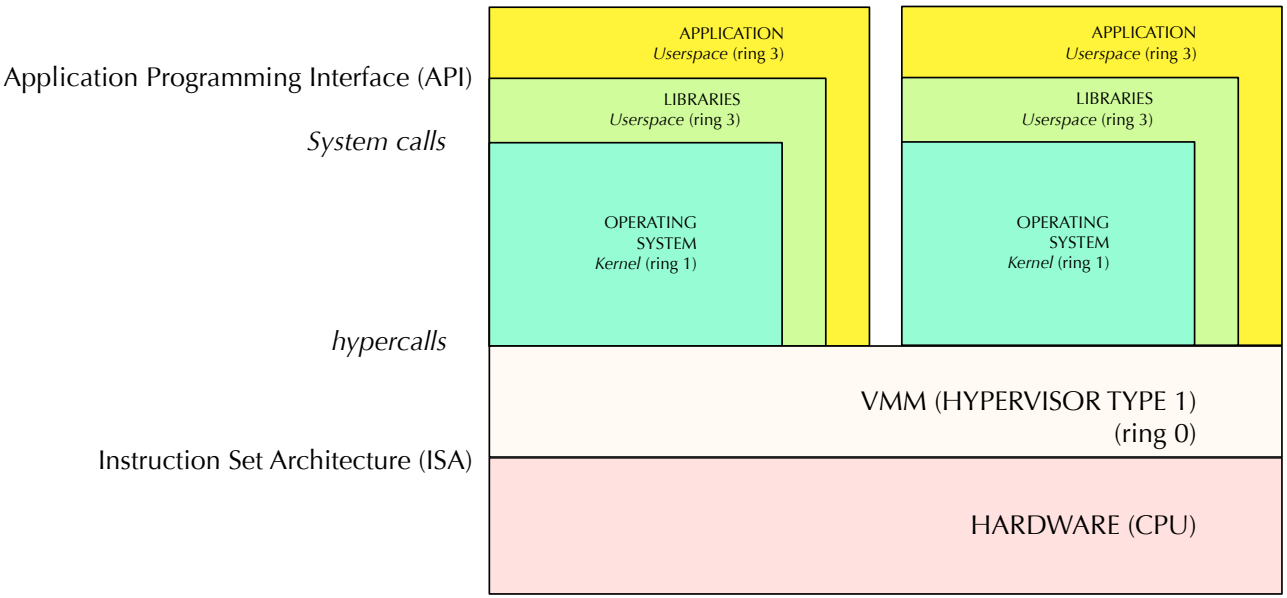


Máquinas Virtuales de Sistema

- VM clásica (hypervisor type 1 a.k.a. *bare metal* a.k.a. *unhosted*).
  - Paravirtualización.
  - Virtualización completa asistida por HW.
- VM alojada (hypervisor type 2).

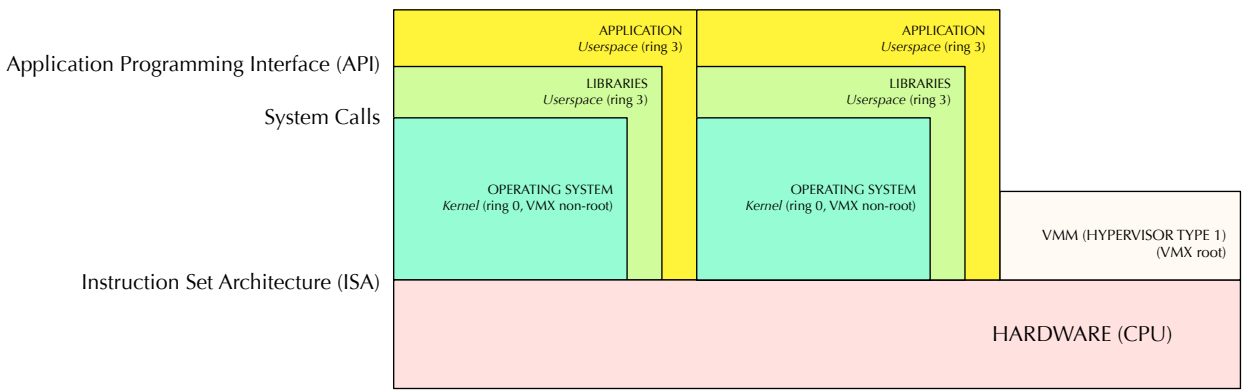
Máquinas Virtuales de Sistema: paravirtualización

- El OS huésped está modificado para ejecutar sobre el VMM.
- El OS huésped realiza *hypercalls* para gestionar la tabla de páginas, configurar el HW, etc.
- Ejemplos: Xen, KVM.



Máquinas Virtuales de Sistema: asistidas por HW

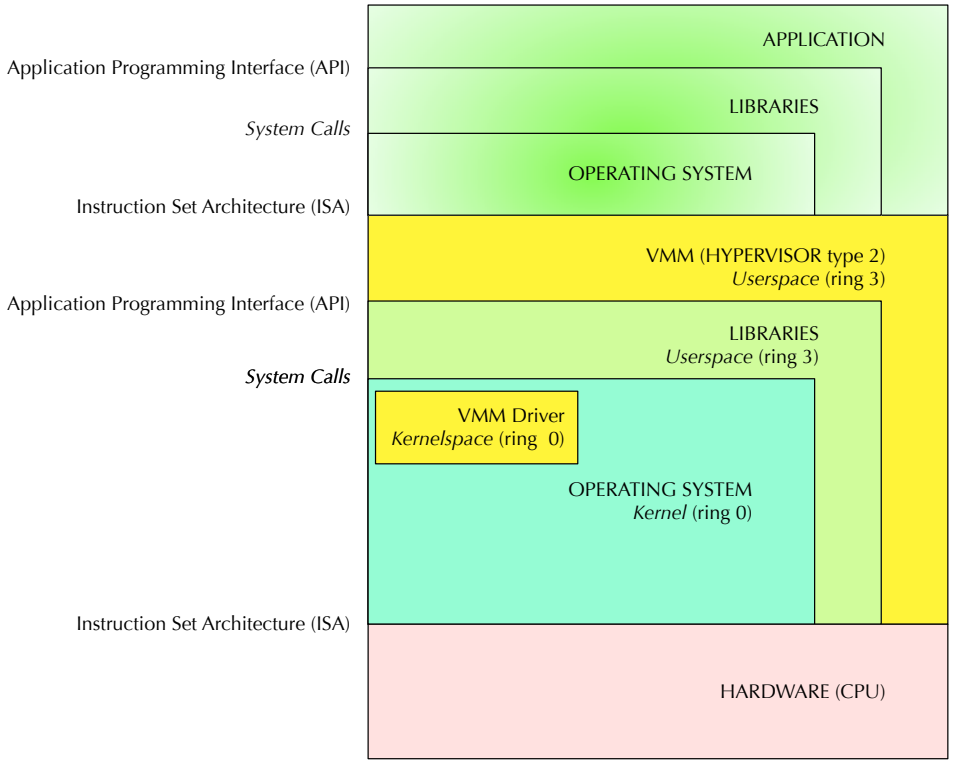
- Se basa instrucciones especiales de la CPU para virtualización. P. ej. Intel VT-x, AMD-V.
- Instrucciones VMX: activar el modo VMX root, lanzar una VM, pasar el control al VMM, retomar una VM, etc.
- Además de ring 0-3, hay un modo especial en el que ejecuta el VMM: VMX root.
- El OS huésped no necesita modificaciones.
- Ejemplo: VMware vSphere.



Máquinas Virtuales de Sistema: alojada

- La VM se aloja sobre otro OS.
- El VMM puede instalar drivers en el OS anfitrión para mejorar el rendimiento. P. ej. VMWare Fusion, Virtual Box.
- *Whole-system* VM: la ISA de la VM no es la misma que la del HW y necesita **emulación**. P. ej. Virtual PC.

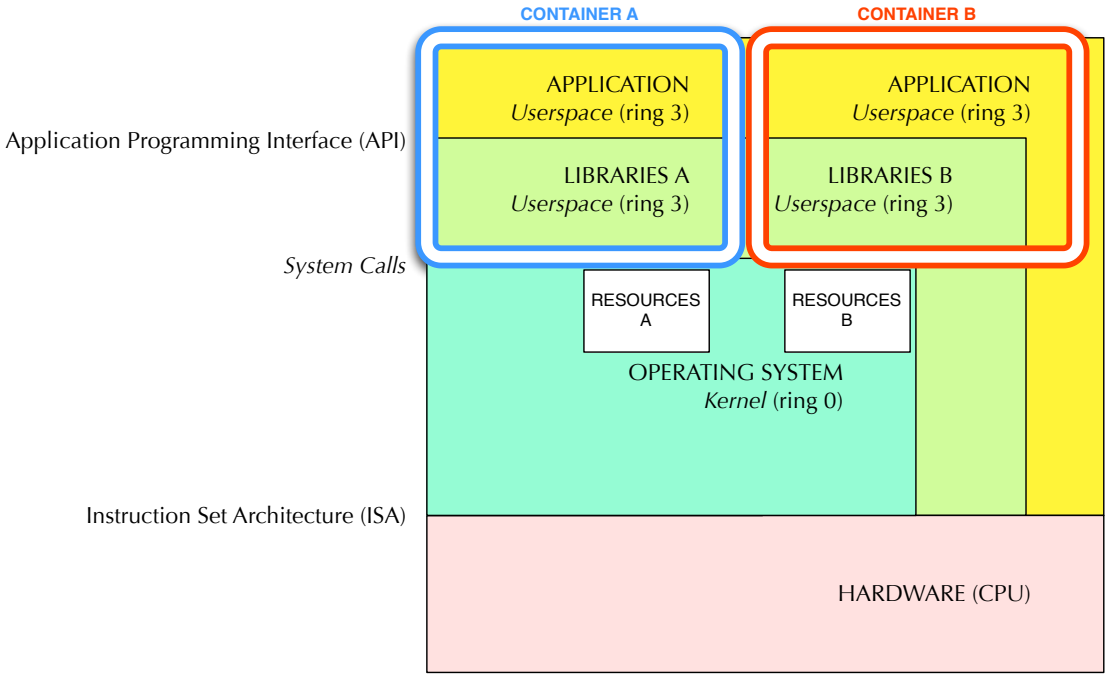
Máquinas Virtuales de Sistema: alojada



Virtualización a nivel del SO: contenedores

- Una VM aisla distintas **imágenes completas** de distintos sistemas operativos ejecutando. Si lo que queremos aislar es un servicio, pagamos cierto coste ejecutando un OS completo para él (**tiempo en arrancar y parar la VM**, rendimiento, etc.).
- Contenedor: dentro del mismo sistema operativo (kernel) se pueden crear distintos entornos aislados, cada uno con sus propias abstracciones y recursos (espacio de procesos, sistema de ficheros raíz, CPU, recursos de red, usuarios, etc.).
- Pros: arranque rápido, más ligeros en general, no necesitas una imagen entera del sistema alojado.
- Contras: menos aislamiento, menos seguridad.
- Ejemplos: Docker, Linux Containers (LXC), OpenVZ, FreeBSD Jails, Solaris Zones, etc.

Virtualización a nivel del SO: contenedores



En este curso nos centraremos en el el **sistema operativo**, sus partes fundamentales y su uso efectivo.

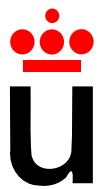
Conceptos básicos de GNU/Linux

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

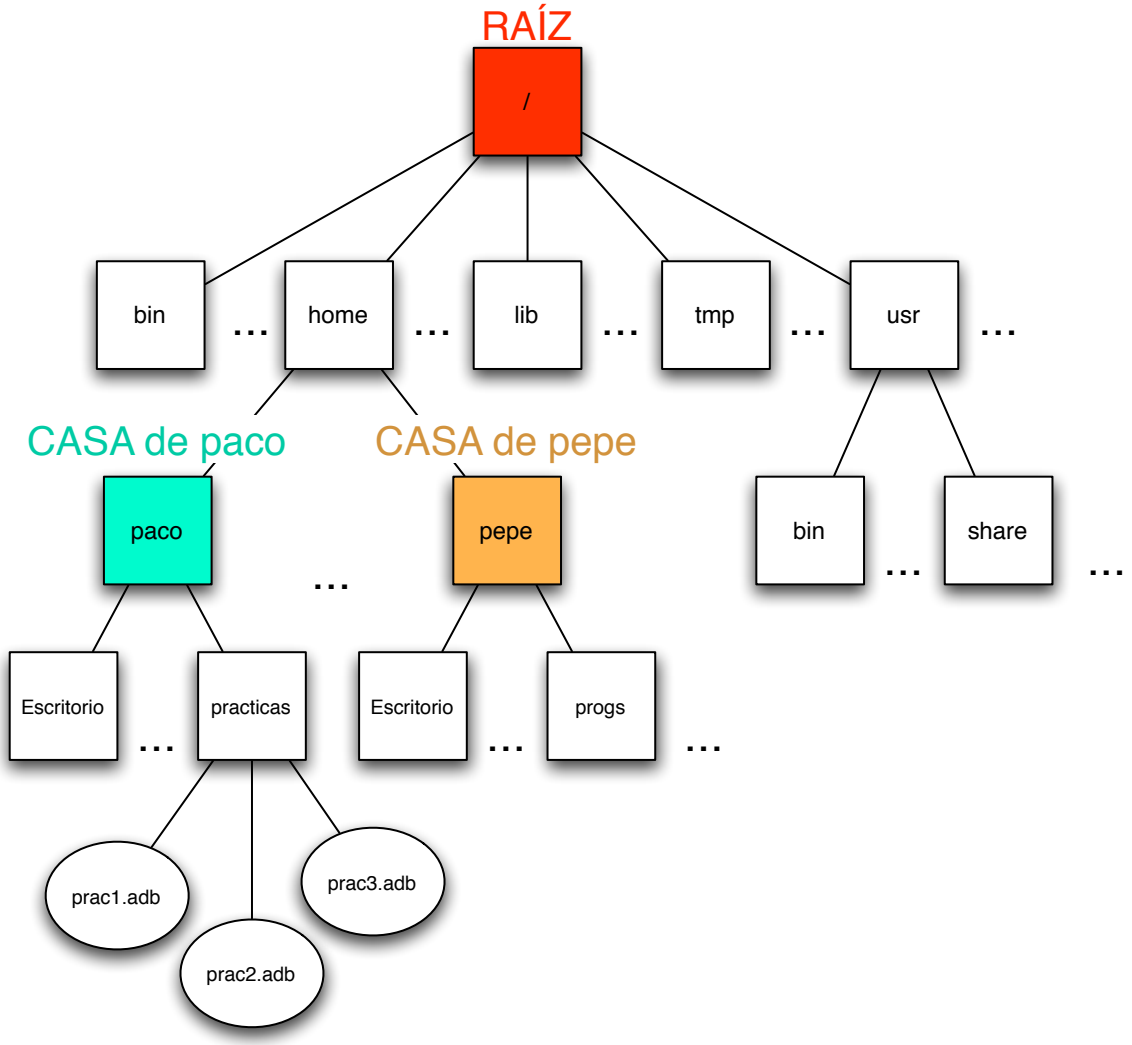
Definiciones

- **Comando o mandato** (command): cadena de texto que identifica a un programa u orden.
- **Shell**: programa que te deja ejecutar *comandos*. Por lo general, permite crear programas (scripts) en un lenguaje propio. Hay distintos tipos de shells, usaremos `bash`. Básicamente, un shell hace esto:
  1. leer un línea de comandos
  2. sustituir algunas cosas en esa línea
  3. crear los procesos para ejecutar los comandos descritos por la línea
- **Prompt**: texto que indica que el **shell** está esperando una orden.
- **Usuario** (login name): el nombre de usuario, todos los programas ejecutan en nombre de un usuario.
- **root**: superusuario o administrador del sistema.

Ficheros y directorios

- Organizados en **árbol** $\rightarrow$ directorio **raíz** (root).
- Dos ficheros que están en distintos directorios son dos ficheros diferentes.
- **Directorio de trabajo**, pwd.
- **Directorio casa**, $HOME

Árbol de ficheros



Directorios en GNU/Linux

- /bin tiene ejecutables.
- /dev tiene dispositivos.
- /etc tiene ficheros de configuración.
- /home tiene los datos personales de los usuarios.
- /lib tiene las bibliotecas (código) que usan los programas ejecutables.
- /proc y /sys ofrecen una interfaz para interaccionar con el núcleo del sistema.
- /sbin tiene los ejecutables del sistema.
- /tmp sirve para almacenar los ficheros temporales, se borra en cada reinicio.
- /usr existe por razones históricas (tamaño de almacenamiento) y contiene gran parte del sistema: contiene directorios similares a los anteriores (/usr/bin o /usr/lib), con los datos y recursos para los programas de usuario (no del sistema).
- /var tiene los datos que se generan en tiempo de ejecución (cache, logs y otros ficheros que generan los programas).
- /boot continen los ficheros de arranque del sistema.
- /media y /mnt puntos de montaje
- /opt contiene ficheros para programas *de terceros*.

Rutas (*path*)

- Ruta absoluta: serie de directorios desde el raíz separados por barras.
  - `/home/alumnos/pepe/fichero.txt`
- Ruta relativa: serie de directorios desde el directorio actual.
  - `alumnos/pepe/fichero.txt`
- .. : directorio padre.
  - `../pepe/fichero.txt`
- . : directorio actual.
  - `./fich1`
- Para indicar que queremos ejecutar un fichero del directorio actual:
  - `./miprograma`

Texto plano

- ASCII: usa 1 byte para representar 128 caracteres (7 bits + 1 bit).
- ISO-Latin 1 (8859-1): usa 1 byte para almacenar caracteres (8 bits).
- UTF-8: puede usar 1,2, o más bytes. Compatible hacia atrás.
- Hay muchas otras.

Ficheros de texto plano

Caracteres de control:

- El carácter `'\n'` indica nueva línea en el texto. Es una convención usada en todos los programas, bibliotecas, etc. en Unix.
- En otros sistemas operativos no tiene por qué ser así (Windows usa la secuencia `'\n\r'`).
- El carácter `'\t'` indica un tabulador.
- No hay carácter EOF: invención de los lenguajes.

Manual

- Las páginas de manual se pueden consultar con el comando `man`: `man sección asunto`
  Por ejemplo: `man 1 gcc`
- Secciones de interés: comandos (1), llamadas al sistema(2), llamadas a biblioteca(3).
- Para buscar sobre una palabra: `apropos`.
  Por ejemplo: `apropos gcc`.

Comandos básicos

- `cd`: cambia de directorio actual.
- `echo`: escribe sus argumentos por su salida.
- `touch`: cambia la fecha de modificación de un fichero. Si no existe el fichero, se crea.
- `ls`: lista el contenido de un directorio.
- `cp`: copia ficheros.
- `mv`: mueve ficheros.
- `rm`: borra ficheros.
- `mkdir`: crea directorios.
- `rmdir`: borra directorios vacíos.
- `date`: muestra la fecha.

Comandos básicos

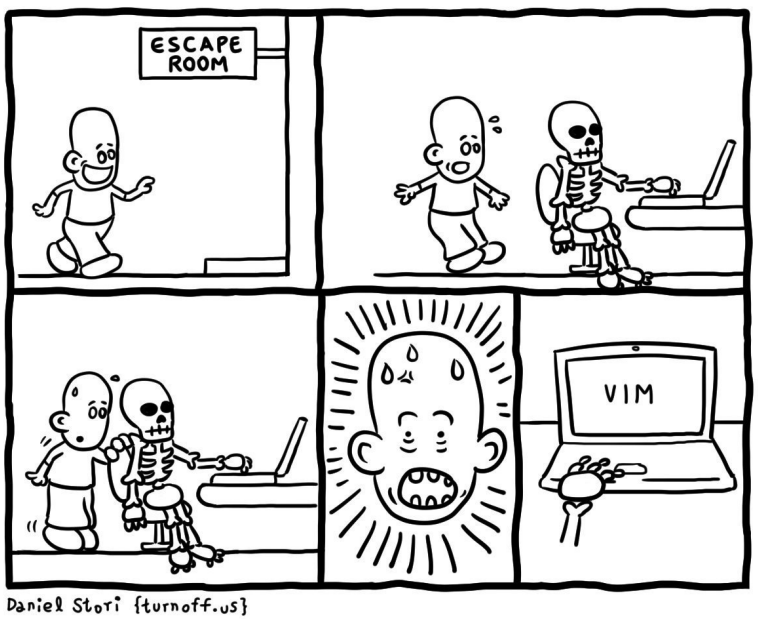
- `who`: muestra los usuarios que están en el sistema.
- `whoami`: muestra tu nombre de usuario.
- `sort`: ordena las líneas de un fichero.
- `wc`: cuenta caracteres, palabras y líneas de ficheros.
- `fgrep, grep`: buscan cadenas dentro de ficheros.
- `cmp, diff`: comparan ficheros.
- `cat`: escribe en su salida el contenido de uno o varios ficheros.
- `less`: permite leer un fichero de texto en el terminal usando *scroll*.

Comandos básicos

- `file`: da pistas sobre el contenido de un fichero.
- `od`: escribe en su salida el los datos de un fichero en distintos formatos.
- `head, tail`: escriben el las primeras/últimas líneas del fichero en su salida.
- `tar`: crea un fichero con múltiples ficheros dentro (comprimidos o no).
- `gzip/gunzip`: comprime/descomprime un fichero.
- `top`: muestra los procesos y el estado de sistema.
- `reset`: restablece el estado del terminal.
- `exit`: el shell termina su ejecución.

Editor en modo texto: vi

- Hay múltiples editores para usar en el terminal (nano, pico, vim, emacs, etc.). Es necesario saber cómo editar texto en un terminal.
- **vi** es el editor clásico.
- **vim** es un editor basado en **vi**. Implementa un superconjunto. En ciertas distribuciones, `vi` es en realidad `vim`.
- Tiene fama de ser complicado (sobre todo salir de él) :)



Editor en modo texto: vi

Hay múltiples editores para usar en el terminal (nano, pico, vim, emacs, etc.), pero **vi** es el editor clásico. Comandos para sobrevivir:

- Tiene dos modos: modo inserción (para escribir) y modo comando
  - **i** pasa a modo inserción
  - **ESC** pasa a modo comando
- En modo comando
  - **:q!** sale del editor sin guardar
  - **:x** salva el fichero y sale (también se puede con **:wq**)
  - **:w** salva el fichero
  - **:número** se mueve a esa línea del fichero
  - **i** inserta antes del cursor
  - **a** inserta después del cursor
  - **o** inserta en una línea nueva
  - **dd** corta una línea
  - **p** pega la línea cortada anteriormente
  - **h, j, k, l** mueve el cursor a izquierda, abajo, arriba, derecha

Variables

- Variable de shell: son locales a la shell, los programas ejecutados por el shell no tienen dichas variables.
- Variable de entorno: los programas ejecutados por el shell sí tienen su propia copia de la variable, con el mismo valor.
- `mivar=hola` define la variable de shell con nombre *mivar*, cuyo valor será *hola*.
- `$mivar` el shell sustituye eso por el valor de dicha variable (si no existe, lo sustituye por nada).
- `export mivar` exporta la variable (ahora es una variable de entorno).

Variables

- El comando `set` muestra todas las variables (de shell y de entorno).
- El comando `printenv` muestra las variables de entorno (también lo hace el comando `env`).
- El comado `unset` elimina una variable.
- Variables populares:
  - $PATH: la ruta de los programas.
  - $HOME: la ruta de tu directorio casa.
  - $USER: el nombre de usuario
  - $PWD: la ruta actual del shell
  - $LANG: configuración de localización (*locales*).
  - $LC_xxx: otras variables de localización (*locales*).

Usando el terminal

Globbing (wildcards): caracteres especiales para el shell que sirven para hacer referencia a nombres de ficheros:

- **?** cualquier carácter.
- * cualquier secuencia de caracteres.
- **[ab]** cualquiera de los caracteres que están dentro de los corchetes (letra a o la letra b en el ejemplo).
- **[b-z]** cualquier carácter que se encuentre entre esas dos (de la letra b a la z en el ejemplo).

Para escribir caracteres especiales sin que haya sustitución:

- ' ' las comillas simples *escapan* todo lo que tienen dentro (ya no tienen un significado especial).
- " " las comillas dobles *escapan* todo menos algunas sustituciones (p. ej. las variables de entorno).

Usando el terminal

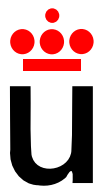
- ↑ repite los comandos ejecutados anteriormente en la shell.
- **Tab** El tabulador completa nombres de ficheros.
- Ctrl+r deja buscar comandos que ejecutamos hace tiempo.
- Ctrl+c mata el programa que se está ejecutando.
- Ctrl+z detiene el programa que se está ejecutando.
- Ctrl+d termina la entrada (o manda lo pendiente).
- Ctrl+a mueve el cursor al principio de la línea.
- Ctrl+e mueve el cursor al final de la línea.
- Ctrl+w borra la palabra anterior en la línea.
- Ctrl+u borra desde el cursor hasta el principio de la línea.
- Ctrl+k borra desde el cursor hasta el final de la línea.
- Ctrl+s congela el terminal. Ctrl+q lo descongela. En muchas configuraciones, Ctrl+s ya no hace esto, pero hay que tenerlo en mente por si sucede.
- Ctrl+l: limpia el terminal.

Introducción a C sobre GNU/Linux

Gorka Guardiola, Enrique Soriano

GSYC

16 de septiembre de 2019



Universidad
Rey Juan Carlos

Características

- Programación imperativa estructurada.
- Relativamente de “bajo nivel”.
- Lenguaje simple, la funcionalidad está en las bibliotecas.
- Básicamente maneja números, caracteres y direcciones de memoria.
- No tiene tipado fuerte.

Hola mundo: un vistazo rápido.

```
#include <stdlib.h>
#include <stdio.h>

/* Comentario */

int
main(int argc, char *argv[])
{
     printf("hola mundo");
     exit(EXIT_SUCCESS);
}
```

Antes de nada: cómo compilar

La compilación se compone de tres fases. El programa gcc se encarga de las tres, en realidad es un *front-end* que invoca a otros programas:

- Preprocesado: incluye ficheros de cabecera (#includes), quita comentarios, etc.
- Compilación (*compiling*): genera el código objeto a partir del código fuente, pasando por código ensamblador (aunque no nos demos cuenta).
- Enlazado (*linking*): a partir de uno o varios ficheros objeto y bibliotecas, genera un único binario ejecutable.

Antes de nada: cómo compilar

- Preprocesado y compilación:

```
gcc -c -Wall -Wshadow -g hello.c
```

OJO: los warnings hay que tratarlos como errores!

- Enlazado:

```
gcc -o hello hello.o
```

Antes de nada: cómo conseguir ayuda

- Las páginas de manual se pueden consultar con el comando `man`: `man sección asunto`
  Por ejemplo: `man 1 gcc`
- Secciones de interés: comandos (1), llamadas al sistema(2), llamadas a biblioteca(3).
- Para buscar sobre una palabra: `apropos`.
  Por ejemplo: `apropos gcc`.

Hola mundo: disección.

```
#include <stdlib.h>                /* Instrucciones para el preprocesador */
#include <stdio.h>

/* Comentario */

int
main(int argc, char *argv[])    /* Definición de función.
                                   Punto de entrada.*/
{                               /* Inicio de bloque */
     printf("hola mundo");      /* Sentencia, llamada a función */
     exit(EXIT_SUCCESS);        /* Sentencia, llamada a función */
}                               /* Fin de bloque */
```

Cosas importantes del hola mundo

- Los `#include` tienen que seguir un orden, especificado en la página de manual correspondiente.
- Los comentarios no pueden estar anidados.
- Todas las sentencias acaban con un ";".
- Un bloque o *sentencia compuesta* es un grupo de sentencias que se trata sintácticamente como una única sentencia. Los bloques se determinan mediante llaves ($\{\}$). Las sentencias de una función se engloban en un bloque.

Cosas importantes del hola mundo

```
int
main(int argc, char *argv[])    /* Definición de función.
                                   Punto de entrada.*/
```

- `main()` es la función por la que se comienza a ejecutar el programa, es lo que se denomina "punto de entrada".
- Recibe dos parámetros, retorna un valor, que es el status del programa (más adelante veremos las funciones).
- La llamada de biblioteca `exit()` indica si el programa ha acabado bien o no. La constante `EXIT_SUCCESS` significa que se ha acabado bien. Podemos usar `EXIT_FAILURE` para indicar lo contrario.

Tipos de datos fundamentales

Dejamos fuera los tipos reales, nos quedamos con los enteros:

- `char` : carácter con signo (1 byte), p.e. 'a' , 12
- `int` : entero con signo (4 bytes), p.e. 77 -11
- `unsigned char, uchar` : carácter sin signo (1 byte), usado para operar sobre bits.
- `unsigned int, uint` : entero sin signo (4 bytes), p.e. 77
- `long`: entero largo, p.e. 431414341L
- `long long`: entero más largo, p.e. 432423432423LL

Y otro:

- `void` : vacío (ya veremos para qué sirve).

Tamaño

- C tiene tipado débil y no se queja si intentas asignar una variable a otra de distinto tamaño.
- Si se asigna a una de menor tamaño, se trunca.
- Si se desborda una variable, nadie te lo va a decir.
- Puedes asignar una con signo a una sin signo (y viceversa).
- Que el tipo tenga signo sólo se tiene en cuenta en las comparaciones.

Declaración e inicialización de variables

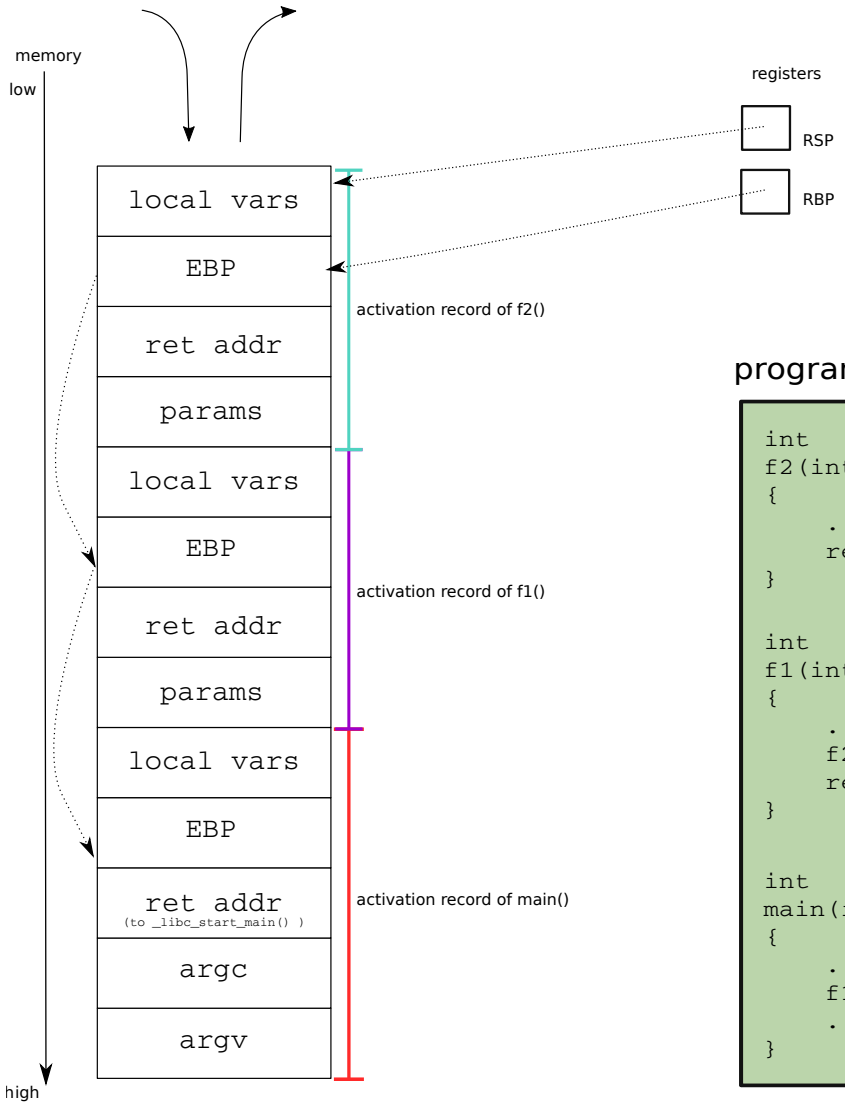
```
#include <stdlib.h>
#include <stdio.h>

int x = 1;                  /* variables globales */
int k;


int
main(int argc, char *argv[])
{
     int i, q=1, u=12;    /* variables locales */
     char c;
     char p = 'o';

     c = 'z';
     i = 13;
     exit(EXIT_SUCCESS);
}
```

Declaración e inicialización

Declaración de variables:

- Las variables globales o externas se declaran fuera de cualquier función. Se "ven" desde cualquier función del fichero, su ámbito es todo el fichero. Se localizan en el segmento de datos. Si no se inicializan explícitamente, se inicializan a 0.
- Las variables locales o automáticas se declaran dentro de una función y sólo se pueden "ver" dentro de su ámbito, que es el bloque en el que se han declarado y sus bloques anidados. Se localizan en la pila. Si no se inicializan explícitamente, tienen un valor indeterminado.

La pila



program

```
int
f2(int x)
{
    ...
    return y;
}

int
f1(int x)
{
    ...
    f2(3);
    return y;
}

int
main(int argc, char *argv[])
{
    ...
    f1(43);
    ...
}
```

Declaración e inicialización

- Una variable declarada dentro de una función como `static` conserva el valor entre distintas invocaciones. Es así porque no se localizan en la pila, sino en el segmento de datos.
- Unas variables pueden ocultar a otras. Para enterarnos, usamos el modificador -Wshadow del compilador.
- Para inicializar, usamos valores constantes o *literales*:
  - Entero Decimal: `777`
  - Entero Hexadecimal: `0x777`
  - Entero Octal: `0777`
  - Carácter: `'a'`, `'\92'`

Constantes

Declaración de constantes enteras mediante el uso de tipos enumerados con `enum`. Si no se da valor, se adjudican valores consecutivos desde el último definido:

```
enum{
        Lun,
        Mar,
        Mier,
        Jue,
        Vier,
        Ndias,
        SalarioBase = 2580,
};
```

Operadores aritméticos

+	Suma. Operandos enteros o reales
-	Resta. Operandos enteros o reales
*	Multiplicación. Operandos enteros o reales
/	División. Operandos enteros o reales
%	Módulo. Operandos enteros

Operadores lógicos

Las operaciones lógicas devuelven un entero. Cualquier valor distinto a 0 significa TRUE, 0 significa FALSE.

`a && b`	AND. 1 si "a" y "b" son distintos de 0
`a || b`	OR. 0 si "a" y "b" son iguales a 0
`!a`	NOT. 1 si "a" es 0, 0 si es distinto de 0

Operadores de relación

Recuerda: 0 si es falso, cualquier otro valor si es verdadero:

`a < b`	“a” menor que “b”
`a > b`	“a” mayor que “b”
`a <= b`	“a” menor o igual que “b”
`a >= b`	“a” mayor o igual que “b”
`a != b`	“a” distinto que “b”
`a == b`	“a” igual que “b”

Operadores de asignación

++	Incremento (pre o post)
−−	Decremento (pre o post)
=	Asignación simple
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
+=	Suma y asignación
−=	Resta y asignación

Operadores bit a bit y otros

OJO: estas operaciones sobre tipos con signo pueden sorprendernos!

`&`	(unario) Dirección-de. Da la dirección de su operando
`*`	(unario) Indirección. Acceso a un valor, teniendo su dirección
`~`	(unario) Complemento
`&`	AND de bits
`^`	XOR de bits
`|`	OR de bits
`<<`	Desplazamiento binario a la izquierda
`>>`	Desplazamiento binario a la derecha
`?:`	Operador ternario
`sizeof`	Operador de tamaño

Precedencia y asociatividad de operadores

`() [] -> .`	izquierda a derecha
`! ++ -- * & ~` sizeof (unarios)	derecha a izquierda
`* / %`	izquierda a derecha
`+ -`	izquierda a derecha
`<< >>`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`& ^ |`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`?:`	derecha a izquierda
`= += -= *= /= %= &= ...`	derecha a izquierda
`,`	izquierda a derecha

Definición y declaración de funciones

- Una función tiene que estar *declarada* antes de poder usarla en el código, pero puede estar *definida* después. Los tipos del *prototipo* y de la definición tienen que coincidir.
- Los argumentos siempre son por valor. Si queremos argumentos por referencia, tendremos que usar un puntero (lo veremos más adelante).
- Si una función no devuelve nada, es de tipo `void`.
- Si una función no tiene argumentos, entonces ponemos un argumento sin nombre de tipo `void`.

Librerías estándar

Las bibliotecas son colecciones de ficheros objeto pre-compilados que podemos usar. Las bibliotecas con funciones estándar:

`<stdio.h>`	Entrada y salida estándar `printf()`, `sprintf()`, `perror()`, ...
`<stdlib.h>`	Librería estándar de C `exit()`, `atoi()`, `getenv()`, ...
`<string.h>`	Operaciones con cadenas de caracteres `strlen()`, `strcat()`, `strcpy()`, ...
`<unistd.h>`	Llamadas al sistema de UNIX `fork()`, `read()`, `write()`, `close()`, ...
`<fcntl.h>`	Control de ficheros `open()`, `creat()`, ...

(Ahora sí) printf

```
int printf(const char * restrict format, ...);
```

- Tiene un número variable de parámetros.
- El primer parámetro indica en una cadena de caracteres el *formato* de lo que se quiere imprimir por pantalla.
- Cada % en el formato se sustituirá con el parámetro que ocupa ese lugar **después** del formato.
- Lo que viene después del % es la forma en la que se quiere imprimir el dato: %d es un entero, %c es un carácter, %x un entero hexadecimal sin signo, %o un entero octal sin signo, %s una cadena de caracteres o string (las veremos más tarde)... más info en `man 3 printf`.

If

```
if ( expresión ) {
     sentencias1...
} else  {
     sentencias2...
}
```

- Si la expresión evalúa a un entero distinto de 0, no se entra al if, se entra a `else` (si lo hay).
- Los paréntesis son obligatorios.
- Si sólo hay una sentencia, podemos prescindir de las llaves.

Switch

```
switch ( expresión ) {
case valor1:
                sentencias1...
case valor2:
                sentencias2...
default:
                sentencias3...
}
```

- El flujo pasa por el `case` que corresponde al valor de la expresión.
- La sentencia `break` rompe un bucle o un switch. Si no se rompe al final de un case, se entra a otros cases posteriores.
- Si no entra por ningún `case`, entrará en `default` (si lo hay).

While

```
while ( expresión ) {
    sentencias...
}
```

- Se itera hasta que la expresión evalúa a 0.
- La sentencia `break` rompe un bucle.

For

```
for ( inicialización ; condición ; actualización) {
    sentencias...
}
```

- La inicialización sólo se ejecuta una vez antes de la primera iteración y antes de evaluar la condición.
- Se itera en el bucle hasta que se deja de cumplir la condición.
- Al final de cada iteración se ejecuta la actualización.

Punteros

- Las variables son una o varias direcciones de memoria contiguas con los bytes correspondientes.
- Un puntero es una variable que contiene una dirección de memoria.

Punteros

- Para declarar una variable puntero a un tipo:
  `tipodedato * nombre;`
  Por ejemplo:
  `int * ptr; /* un puntero a entero */`
- En una sentencia, el operador * (*dereference*) delante de una variable de tipo puntero significa que queremos operar sobre el contenido de la dirección a la que apunta.
- Con el operador & (*address of*) delante de una variable obtenemos su dirección de memoria.
- No se puede usar un puntero que no apunta a ningún sitio (NULL)

Aritmética de punteros

- Los punteros se pueden sumar, restar, etc. P.e. para conseguir el tamaño de un buffer.
- Las operaciones se hacen en múltiplos del tamaño en bytes del tipo de datos al que apunta el puntero.

```
char * cptr; /* los char ocupan 1 byte */
int * iptr;  /* los int ocupan 4 bytes */

...

cptr = cptr + 4; /* la dirección de memoria + 4 posiciones */
iptr = iptr + 4; /* la dirección de memoria + (4*4) posiciones */
```

Los *Arrays* en C

- El índice para N elementos va de 0 a N-1.
- No se comprueban los límites.
- No son más que azúcar sintáctico para los punteros.
- `int lista[N];` $\rightarrow$ "reserva la memoria necesaria para tener N objetos de tipo `int` y guarda la dirección en la variable `lista`".
- `lista[NUM] = 3;` $\rightarrow$ "escribe el entero 3 en la posición de memoria (NUM * tamaño de `int`) a partir del puntero `lista`".

Los *Arrays* en C

- El operador `sizeof` sobre un array devuelve el tamaño de la memoria reservada (NO el número de elementos!).
- Inicialización de arrays:

```
int lista1[5] = { 1, 2, 3, 4, 5 };  /* damos el tamaño e
                                       inicializamos            */
int lista2[] = { 1, 2, 3, 4, 5 };   /* tamaño == numero elementos
                                       en la inicialización     */
int lista3[5] = { 1, 2, 3 };        /* da igual si sobra huecos */
int lista2[4] = { 1, 2, 3, 4, 5 };  /* error!!! ATENCION!!! esto compila
```

Pasando direcciones de memoria como argumento

```
#include <stdlib.h>
#include <stdio.h>

void
dameletra(char *c)
{
    *c = 'A';
}

int
main(int argc, char *argv[])
{
    char c = 'b';

    dameletra(&c);
    printf("c es %c\n", c);
    exit(EXIT_SUCCESS);
}
```

Cadenas de caracteres (string)

- Son *arrays* de caracteres acabados en un carácter `'\0'` (el carácter nulo).
- Si no se acaba en un nulo, no es una string.
- Inicializar una string:

```
char str[] = "hola";                          /* inicializando una string */

char str2[] = {'h', 'o', 'l', 'a', '\0'}; /* equivalente a lo anterior */
```

Cadenas de caracteres (strings)

Funciones para manejo de cadenas (ver prototipos en las páginas de manual):

- snprintf: similar a printf, pero imprime en una cadena. Escribe como mucho el número de bytes que se especifica en el segundo argumento, contando el carácter nulo. Devuelve el número de caracteres escritos en la cadena.
- strlen: devuelve el tamaño de una cadena, sin contar el carácter nulo.
- strcat: concatena dos cadenas, la segunda al final de la primera, dejando el resultado en la primera. Devuelve un puntero a la cadena resultante. La primera cadena tiene que tener espacio suficiente como para que quepa la concatenación.

Argumentos de main

- `argc`: variable entera que indica el número de argumentos que se le han pasado a `main`.
- `argv`: array de strings con cada uno de los argumentos. El primer argumento se corresponde con el nombre del programa que se invoca. Desde UNIX V7, siempre va terminado con un NULL.

Registros

```
struct Coordenada{
    int x;
    int y;
};

struct Coordenada c = {13, 33}; /* inicialización */
```

- El tamaño que ocupa en memoria no tiene porqué coincidir con la suma de los tipos de datos que contiene la estructura.
- Sólo se pueden hacer 3 cosas con ellas: copiarlas/asignarlas (esto incluye pasarlas como parámetro o retornarla), obtener su dirección (&), y acceder a sus campos.

Registros

```
struct Coordenada{
    int x;
    int y;
};

typedef struct Coordenada Coordenada; /* definición de tipo Coordenada */
Coordenada c = {13,31};               /* declaración e inicialización  */
```

- Se suele definir un tipo de datos nuevo con `typedef` para usarlas de forma más cómoda.
- Si tenemos un puntero a una estructura, el operador `->` sirve para acceder a sus campos:

$$\texttt{p->x} \equiv \texttt{(*p).x}$$

Memoria dinámica

Free y malloc (leer la página de manual):

- `malloc`: sirve para pedir memoria en tiempo de ejecución. La memoria devuelta se localiza en el *heap*.
- La memoria reservada con `malloc` puede tener cualquier contenido.
- Si no hay memoria en el sistema, devuelve NULL.
- `free`: sirve para liberar la memoria devuelta anteriormente por `malloc`. No se puede liberar memoria que no se ha solicitado con `malloc`.
- Hay que liberar la memoria cuando ya no nos hace falta.

Programas con varios ficheros fuente

- Las variables globales declara en un fichero externo tiene que definirse como `extern`.
- Una variable tiene que estar declarada en un fichero fuente.
- Una función o variable global declarada como `static` no es visible desde otros ficheros. Si no se especifica, sí son visibles.
- Las variables, tipos de datos, constantes, etc. compartidas por los ficheros fuente de un programa deberían estar en un fichero de cabeceras.

Programas con varios ficheros fuente

- Cada fichero fuente debe incluir los ficheros de cabeceras que necesite. No es buena idea incluir ficheros de cabecera en otros ficheros de cabecera.
- Para incluir un fichero de cabeceras que no está en los directorios del sistema (/usr/include,...):

  ```
  #include "rutadelfichero"
  ```

  (si no lo encuentra, lo busca entre los directorios del sistema)
- No se deben incluir dos veces un mismo fichero de cabecera.

gdb

El comando `gdb` es un depurador que nos permite:

- Inspeccionar un programa (p. ej. desensamblar).
- Inspeccionar un proceso (p. ej. ver los valores de la memoria).
- Inspeccionar un *core*: es la *foto* de un proceso en un fichero.
- En la mayoría de las ocasiones, **no es la forma más eficiente** de depurar un programa.

gdb

1. Arrancamos gdb con el ejecutable:
   `gdb ejecutable`
2. Ejecutamos dentro de gdb:
   `run argumento1 argumento2 ...`
3. ... fallo en ejecución ...
4. `bt` #vuelca la pila
5. `frame 3` #selecciono el registro de activación que deseo inspeccionar
6. `info locals` #veo el valor de las variables locales
7. `info args` #veo el valor de los argumentos
8. `whatis z` #veo el tipo de la variable z

gdb

Meter punto de ruptura y ejecutar paso a paso:

1. Arrancamos gdb con el ejecutable:
   `gdb ejecutable`
2. `break f1` #mete punto de ruptura en la función f1
3. Ejecutamos dentro de gdb:
   `run argumento1 argumento2 ...`
4. se para en f1, ahora podemos inspeccionar como en el ejemplo anterior.
5. `stepi` #ejecuta una instruccion
6. ...
7. `continue` # sigue ejecutando normalmente.

Introducción a C sobre GNU/Linux

Gorka Guardiola, Enrique Soriano

GSYC

21 de septiembre de 2022

Universidad
Rey Juan Carlos

Características

- Programación imperativa estructurada.
- Relativamente de “bajo nivel” (dentro de los lenguajes de *alto nivel*, es de los de menos nivel de abstracción).
- Lenguaje simple, la funcionalidad está en las bibliotecas.
- Básicamente maneja números, caracteres y direcciones de memoria.
- No tiene tipado fuerte.

Hola mundo: un vistazo rápido.

```
#include <stdlib.h>
#include <stdio.h>

/* Comentario */

int
main(int argc, char *argv[])
{
     printf("hola mundo");
     exit(EXIT_SUCCESS);
}
```

Antes de nada: cómo compilar

La compilación se compone de tres fases. El programa gcc se encarga de las tres, en realidad es un *front-end* que invoca a otros programas:

- Preprocesado: incluye ficheros de cabecera (#includes), quita comentarios, etc.
- Compilación (*compiling*): genera el código objeto a partir del código fuente, pasando por código ensamblador (aunque no nos demos cuenta).
- Enlazado (*linking*): a partir de uno o varios ficheros objeto y bibliotecas, genera un único binario ejecutable.

Antes de nada: cómo compilar

- Preprocesado y compilación:

  ```
  gcc -c -Wall -Wshadow -g hello.c
  ```

  OJO: ¡los warnings hay que tratarlos como errores!

- Enlazado:

  ```
  gcc -o hello hello.o
  ```

Antes de nada: cómo conseguir ayuda

- Las páginas de manual se pueden consultar con el comando `man`: `man sección asunto`
  Por ejemplo: `man 1 gcc`
- Secciones de interés: comandos (1), llamadas al sistema(2), llamadas a biblioteca(3).
- Para buscar sobre una palabra: `apropos`.
  Por ejemplo: `apropos gcc`.

Hola mundo: disección.

```
#include <stdlib.h>                /* Instrucciones para el preprocesador */
#include <stdio.h>

/* Comentario */

int
main(int argc, char *argv[])    /* Definición de función.
                                   Punto de entrada.*/
{                               /* Inicio de bloque */
     printf("hola mundo");      /* Sentencia, llamada a función */
     exit(EXIT_SUCCESS);        /* Sentencia, llamada a función */
}                               /* Fin de bloque */
```

Cosas importantes del hola mundo

- Los `#include` tienen que seguir un orden, especificado en la página de manual correspondiente.
- Los comentarios no pueden estar anidados.
- Todas las sentencias acaban con un ";".
- Un bloque o *sentencia compuesta* es un grupo de sentencias que se trata sintácticamente como una única sentencia. Los bloques se determinan mediante llaves ($\{\}$). Las sentencias de una función se engloban en un bloque.

Cosas importantes del hola mundo

```
int
main(int argc, char *argv[])    /* Definición de función.
                                   Punto de entrada.*/
```

- `main()` es la función por la que se comienza a ejecutar el programa, es lo que se denomina "punto de entrada".
- Recibe dos parámetros, retorna un valor, que es el status del programa (más adelante veremos las funciones).
- La llamada de biblioteca `exit()` indica si el programa ha acabado bien o no. La constante `EXIT_SUCCESS` significa que se ha acabado bien. Podemos usar `EXIT_FAILURE` para indicar lo contrario.

Tipos de datos fundamentales

Dejamos fuera los tipos reales, nos quedamos con los enteros:

- `char` : carácter con signo (1 byte), p.e. 'a' , 12
- `int` : entero con signo (4 bytes), p.e. 77 -11
- `unsigned char, uchar` : carácter sin signo (1 byte), usado para operar sobre bits.
- `unsigned int, uint` : entero sin signo (4 bytes), p.e. 77
- `long`: entero largo, p.e. 431414341L
- `long long`: entero más largo, p.e. 432423432423LL

Y otro:

- `void` : vacío (ya veremos para qué sirve).

Tamaño

- C tiene tipado débil y no se queja si intentas asignar una variable a otra de distinto tamaño.
- Si se asigna a una de menor tamaño, se trunca.
- Si se desborda una variable, nadie te lo va a decir.
- Puedes asignar una con signo a una sin signo (y viceversa).
- Que el tipo tenga signo sólo se tiene en cuenta en las comparaciones.

Declaración e inicialización de variables

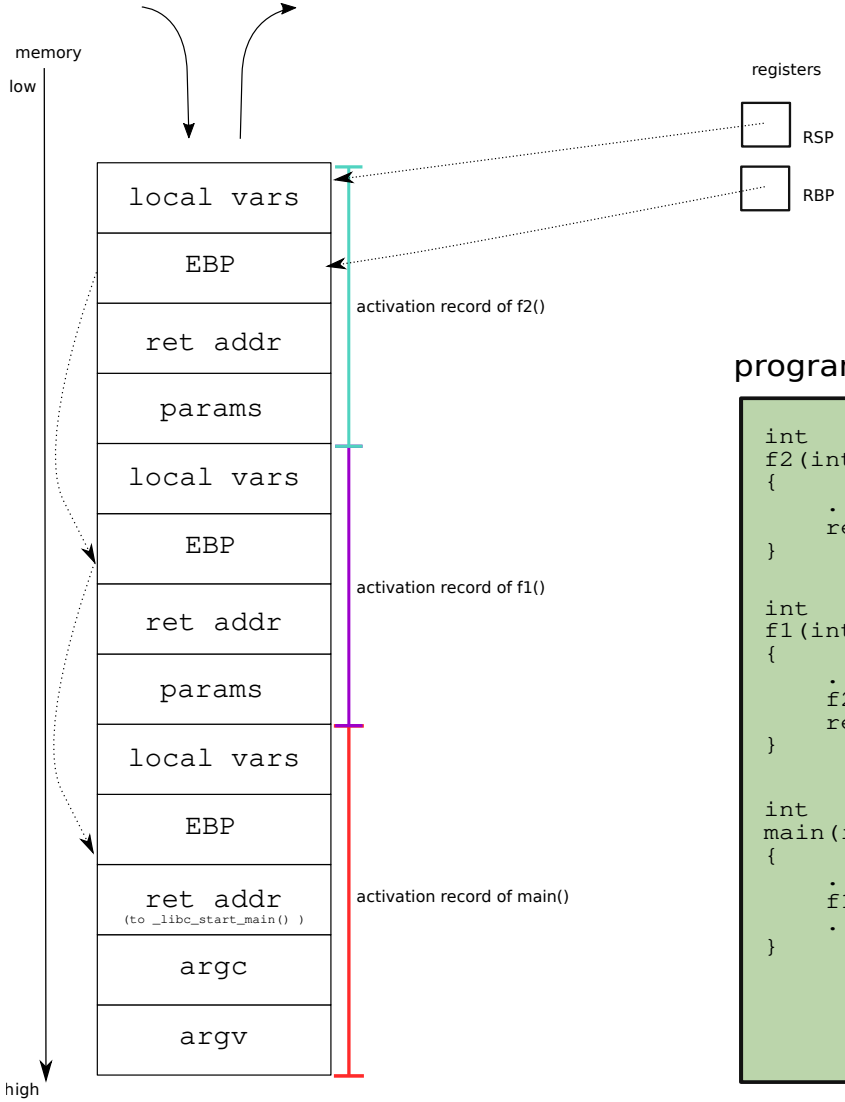
```
#include <stdlib.h>
#include <stdio.h>

int x = 1;                  /* variables globales */
int k;


int
main(int argc, char *argv[])
{
      int i, q=1, u=12;     /* variables locales */
      char c;
      char p = 'o';

      c = 'z';
      i = 13;
      exit(EXIT_SUCCESS);
}
```

Declaración e inicialización

Declaración de variables:

- Las variables globales o externas se declaran fuera de cualquier función. Se "ven" desde cualquier función del fichero, su ámbito es todo el fichero. Se localizan en el segmento de datos. Si no se inicializan explícitamente, se inicializan a 0.
- Las variables locales o automáticas se declaran dentro de una función y sólo se pueden "ver" dentro de su ámbito, que es el bloque en el que se han declarado y sus bloques anidados. Se localizan en la pila. Si no se inicializan explícitamente, tienen un valor indeterminado.

La pila



program

```
int
f2(int x)
{
    ...
    return y;
}

int
f1(int x)
{
    ...
    f2(3);
    return y;
}

int
main(int argc, char *argv[])
{
    ...
    f1(43);
    ...
}
```

Declaración e inicialización

- Una variable declarada dentro de una función como `static` conserva el valor entre distintas invocaciones. Es así porque no se localizan en la pila, sino en el segmento de datos.
- Unas variables pueden ocultar a otras. Para enterarnos, usamos el modificador -Wshadow del compilador.
- Para inicializar, usamos valores constantes o *literales*:
  - Entero Decimal: `777`
  - Entero Hexadecimal: `0x777`
  - Entero Octal: `0777`
  - Carácter: `'a'`, `'\92'`

Constantes

Declaración de constantes enteras mediante el uso de tipos enumerados con `enum`. Si no se da valor, se adjudican valores consecutivos desde el último definido:

```
enum{
        Lun,
        Mar,
        Mier,
        Jue,
        Vier,
        Ndias,
        SalarioBase = 2580,
};
```

Operadores aritméticos

+	Suma. Operandos enteros o reales
-	Resta. Operandos enteros o reales
*	Multiplicación. Operandos enteros o reales
/	División. Operandos enteros o reales
%	Módulo. Operandos enteros

Operadores lógicos

Las operaciones lógicas devuelven un entero. Cualquier valor distinto a 0 significa TRUE, 0 significa FALSE.

`a && b`	AND. 1 si "a" y "b" son distintos de 0
`a || b`	OR. 0 si "a" y "b" son iguales a 0
`!a`	NOT. 1 si "a" es 0, 0 si es distinto de 0

Operadores de relación

Recuerda: 0 si es falso, cualquier otro valor si es verdadero:

`a < b`	“a” menor que “b”
`a > b`	“a” mayor que “b”
`a <= b`	“a” menor o igual que “b”
`a >= b`	“a” mayor o igual que “b”
`a != b`	“a” distinto que “b”
`a == b`	“a” igual que “b”

Operadores de asignación

++	Incremento (pre o post)
--	Decremento (pre o post)
=	Asignación simple
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
+=	Suma y asignación
-=	Resta y asignación

Operadores bit a bit y otros

OJO: estas operaciones sobre tipos con signo pueden sorprendernos!

`&`	(unario) Dirección-de. Da la dirección de su operando
`*`	(unario) Indirección. Acceso a un valor, teniendo su dirección
`~`	(unario) Complemento
`&`	AND de bits
`^`	XOR de bits
`|`	OR de bits
`<<`	Desplazamiento binario a la izquierda
`>>`	Desplazamiento binario a la derecha
`?:`	Operador ternario
`sizeof`	Operador de tamaño

Precedencia y asociatividad de operadores

`() [] -> .`	izquierda a derecha
`! ++ -- * & ~` sizeof (unarios)	derecha a izquierda
`* / %`	izquierda a derecha
`+ -`	izquierda a derecha
`<< >>`	izquierda a derecha
`< <= > >=`	izquierda a derecha
`== !=`	izquierda a derecha
`& ^ |`	izquierda a derecha
`&&`	izquierda a derecha
`||`	izquierda a derecha
`?:`	derecha a izquierda
`= += -= *= /= %= &= ...`	derecha a izquierda
`,`	izquierda a derecha

Definición y declaración de funciones

- Una función tiene que estar *declarada* antes de poder usarla en el código, pero puede estar *definida* después. Los tipos del *prototipo* y de la definición tienen que coincidir.
- Los argumentos siempre son por valor. Si queremos argumentos por referencia, tendremos que usar un puntero (lo veremos más adelante).
- Si una función no devuelve nada, es de tipo `void`.
- Si una función no tiene argumentos, entonces ponemos un argumento sin nombre de tipo `void`.

Librerías estándar

Las bibliotecas son colecciones de ficheros objeto pre-compilados que podemos usar. Las bibliotecas con funciones estándar:

`<stdio.h>`	Entrada y salida estándar `printf()`, `sprintf()`, `perror()`, ...
`<stdlib.h>`	Librería estándar de C `exit()`, `atoi()`, `getenv()`, ...
`<string.h>`	Operaciones con cadenas de caracteres `strlen()`, `strcat()`, `strcpy()`, ...
`<unistd.h>`	Llamadas al sistema de UNIX `fork()`, `read()`, `write()`, `close()`, ...
`<fcntl.h>`	Control de ficheros `open()`, `creat()`, ...

(Ahora sí) printf

```
int printf(const char * restrict format, ...);
```

- Tiene un número variable de parámetros.
- El primer parámetro indica en una cadena de caracteres el *formato* de lo que se quiere imprimir por pantalla.
- Cada % en el formato se sustituirá con el parámetro que ocupa ese lugar **después** del formato.
- Lo que viene después del % es la forma en la que se quiere imprimir el dato: %d es un entero, %c es un carácter, %x un entero hexadecimal sin signo, %o un entero octal sin signo, %s una cadena de caracteres o string (las veremos más tarde)... más info en `man 3 printf`.

If

```
if ( expresión ) {
     sentencias1...
} else  {
     sentencias2...
}
```

- Si la expresión evalúa a un entero distinto de 0, no se entra al if, se entra a `else` (si lo hay).
- Los paréntesis son obligatorios.
- Si sólo hay una sentencia, podemos prescindir de las llaves.

Switch

```
switch ( expresión ) {
case valor1:
                sentencias1...
case valor2:
                sentencias2...
default:
                sentencias3...
}
```

- El flujo pasa por el `case` que corresponde al valor de la expresión.
- La sentencia `break` rompe un bucle o un switch. Si no se rompe al final de un case, se entra a otros cases posteriores.
- Si no entra por ningún `case`, entrará en `default` (si lo hay).

While

```
while ( expresión ) {
    sentencias...
}
```

- Se itera hasta que la expresión evalúa a 0.
- La sentencia `break` rompe un bucle.

For

```
for ( inicialización ; condición ; actualización) {
    sentencias...
}
```

- La inicialización sólo se ejecuta una vez antes de la primera iteración y antes de evaluar la condición.
- Se itera en el bucle hasta que se deja de cumplir la condición.
- Al final de cada iteración se ejecuta la actualización.

Punteros

- Las variables son una o varias direcciones de memoria contiguas con los bytes correspondientes.
- Un puntero es una variable que contiene una dirección de memoria.

Punteros

- Para declarar una variable puntero a un tipo:
  `tipodedato * nombre;`
  Por ejemplo:
  `int * ptr; /* un puntero a entero */`
- En una sentencia, el operador * (*dereference*) delante de una variable de tipo puntero significa que queremos operar sobre el contenido de la dirección a la que apunta.
- Con el operador & (*address of*) delante de una variable obtenemos su dirección de memoria.
- No se puede usar un puntero que no apunta a ningún sitio (NULL)

Aritmética de punteros

- Los punteros se pueden sumar, restar, etc. P.e. para conseguir el tamaño de un buffer.
- Las operaciones se hacen en múltiplos del tamaño en bytes del tipo de datos al que apunta el puntero.

```
char * cptr; /* los char ocupan 1 byte */
int * iptr;  /* los int ocupan 4 bytes */

...

cptr = cptr + 4; /* la dirección de memoria + 4 posiciones */
iptr = iptr + 4; /* la dirección de memoria + (4*4) posiciones */
```

Los *Arrays* en C

- El índice para N elementos va de 0 a N-1.
- No se comprueban los límites.
- No son más que azúcar sintáctico para los punteros.
- `int lista[N];` → "reserva la memoria necesaria para tener N objetos de tipo `int` y guarda la dirección en la variable `lista`".
- `lista[NUM] = 3;` → "escribe el entero 3 en la posición de memoria (NUM * tamaño de `int`) a partir del puntero `lista`".

Los *Arrays* en C

- El operador `sizeof` sobre un array devuelve el tamaño de la memoria reservada (NO el número de elementos!).
- Inicialización de arrays:

```
int lista1[5] = { 1, 2, 3, 4, 5 };  /* damos el tamaño e
                                       inicializamos            */
int lista2[] = { 1, 2, 3, 4, 5 };   /* tamaño == numero elementos
                                       en la inicialización     */
int lista3[5] = { 1, 2, 3 };        /* da igual si sobra huecos */
int lista2[4] = { 1, 2, 3, 4, 5 };  /* error!!! ATENCION!!! esto compila
```

Pasando direcciones de memoria como argumento

```
#include <stdlib.h>
#include <stdio.h>

void
dameletra(char *c)
{
    *c = 'A';
}

int
main(int argc, char *argv[])
{
    char c = 'b';

    dameletra(&c);
    printf("c es %c\n", c);
    exit(EXIT_SUCCESS);
}
```

Cadenas de caracteres (string)

- Son *arrays* de caracteres acabados en un carácter `'\0'` (el carácter nulo).
- Si no se acaba en un nulo, no es una string.
- Inicializar una string:

```
char str[] = "hola";                          /* inicializando una string */

char str2[] = {'h', 'o', 'l', 'a', '\0'}; /* equivalente a lo anterior */
```

Cadenas de caracteres (strings)

Funciones para manejo de cadenas (ver prototipos en las páginas de manual):

- snprintf: similar a printf, pero imprime en una cadena. Escribe como mucho el número de bytes que se especifica en el segundo argumento, contando el carácter nulo. Devuelve el número de caracteres escritos en la cadena.
- strlen: devuelve el tamaño de una cadena, sin contar el carácter nulo.
- strcat: concatena dos cadenas, la segunda al final de la primera, dejando el resultado en la primera. Devuelve un puntero a la cadena resultante. La primera cadena tiene que tener espacio suficiente como para que quepa la concatenación.

Argumentos de main

- `argc`: variable entera que indica el número de argumentos que se le han pasado a `main`.
- `argv`: array de strings con cada uno de los argumentos. El primer argumento se corresponde con el nombre del programa que se invoca. Desde UNIX V7, siempre va terminado con un NULL.

Registros

```
struct Coordenada{
    int x;
    int y;
};

struct Coordenada c = {13, 33}; /* inicialización */
```

- El tamaño que ocupa en memoria no tiene porqué coincidir con la suma de los tipos de datos que contiene la estructura.
- Sólo se pueden hacer 3 cosas con ellas: copiarlas/asignarlas (esto incluye pasarlas como parámetro o retornarla), obtener su dirección (&), y acceder a sus campos.

Registros

```
struct Coordenada{
    int x;
    int y;
};

typedef struct Coordenada Coordenada; /* definición de tipo Coordenada */
Coordenada c = {13,31};               /* declaración e inicialización  */
```

- Se suele definir un tipo de datos nuevo con `typedef` para usarlas de forma más cómoda.
- Si tenemos un puntero a una estructura, el operador `->` sirve para acceder a sus campos:

$$\texttt{p->x} \equiv \texttt{(*p).x}$$

Memoria dinámica

Free y malloc (leer la página de manual):

- `malloc`: sirve para pedir memoria en tiempo de ejecución. La memoria devuelta se localiza en el *heap*.
- La memoria reservada con `malloc` puede tener cualquier contenido.
- Si no hay memoria en el sistema, devuelve NULL.
- `free`: sirve para liberar la memoria devuelta anteriormente por `malloc`. No se puede liberar memoria que no se ha solicitado con `malloc`.
- Hay que liberar la memoria cuando ya no nos hace falta.

Programas con varios ficheros fuente

- Las variables globales declara en un fichero externo tiene que definirse como `extern`.
- Una variable tiene que estar declarada en un fichero fuente.
- Una función o variable global declarada como `static` no es visible desde otros ficheros. Si no se especifica, sí son visibles.
- Las variables, tipos de datos, constantes, etc. compartidas por los ficheros fuente de un programa deberían estar en un fichero de cabeceras.

Programas con varios ficheros fuente

- Cada fichero fuente debe incluir los ficheros de cabeceras que necesite. No es buena idea incluir ficheros de cabecera en otros ficheros de cabecera.
- Para incluir un fichero de cabeceras que no está en los directorios del sistema (/usr/include,...):

  ```
  #include "rutadelfichero"
  ```

  (si no lo encuentra, lo busca entre los directorios del sistema)
- No se deben incluir dos veces un mismo fichero de cabecera.

gdb

El comando `gdb` es un depurador que nos permite:

- Inspeccionar un programa (p. ej. desensamblar).
- Inspeccionar un proceso (p. ej. ver los valores de la memoria).
- Inspeccionar un *core*: es la *foto* de un proceso en un fichero.
- En la mayoría de las ocasiones, **no es la forma más eficiente** de depurar un programa.

gdb

1. Arrancamos gdb con el ejecutable:
   `gdb ejecutable`
2. Ejecutamos dentro de gdb:
   `run argumento1 argumento2 ...`
3. ... fallo en ejecución ...
4. `bt` #vuelca la pila
5. `frame 3` #selecciono el registro de activación que deseo inspeccionar
6. `info locals` #veo el valor de las variables locales
7. `info args` #veo el valor de los argumentos
8. `whatis z` #veo el tipo de la variable z

gdb

Meter punto de ruptura y ejecutar paso a paso:

1. Arrancamos gdb con el ejecutable:
   `gdb ejecutable`
2. `break f1` #mete punto de ruptura en la función f1
3. Ejecutamos dentro de gdb:
   `run argumento1 argumento2 ...`
4. se para en f1, ahora podemos inspeccionar como en el ejemplo anterior.
5. `stepi` #ejecuta una instruccion
6. ...
7. `continue` # sigue ejecutando normalmente.

Análisis dinámico: Valgrind

- Ayuda para encontrar errores y problemas automáticamente
- Hace análisis dinámico del código en ejecución (instrumentado o sin instrumentar)
- Leaks, corrupción de memoria, patrones incorrectos de uso de los recursos
- `http://valgrind.org`

Análisis dinámico: Valgrind

```
$ valgrind ./a.out lespaul sg telecaster stratocaster
==6491== Memcheck, a memory error detector
==6491== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6491== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6491== Command: ./a.out lespaul sg telecaster stratocaster
==6491==
[1] lespaul
[2] sg
[3] telecaster
[4] stratocaster
The largest argument is: stratocaster
==6491==
==6491== HEAP SUMMARY:
==6491==     in use at exit: 19 bytes in 2 blocks
==6491==   total heap usage: 4 allocs, 2 frees, 1,056 bytes allocated
==6491==
==6491== LEAK SUMMARY:
==6491==    definitely lost: 19 bytes in 2 blocks
==6491==    indirectly lost: 0 bytes in 0 blocks
==6491==      possibly lost: 0 bytes in 0 blocks
==6491==    still reachable: 0 bytes in 0 blocks
==6491==         suppressed: 0 bytes in 0 blocks
==6491== Rerun with --leak-check=full to see details of leaked memory
==6491==
==6491== For lists of detected and suppressed errors, rerun with: -s
==6491== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

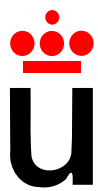
Procesos

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

Procesos

Recuerda:

- El proceso es un programa en ejecución, un flujo de control.
- El sistema operativo proporciona esa abstracción.
- Cada proceso piensa que está solo en la máquina, que tiene su propia CPU y su propia memoria.
- Los procesos tienen distintos datos asociados.

Llamadas al sistema y funciones de libc

- En este tema, comenzaremos a ver **llamadas al sistema** y funciones de la libc.
- Ojo: para algunas funciones de librería, existen versiones *thread safe* para cuando usemos hilos. Siempre debemos intentar usar esas funciones.

Llamadas al sistema y funciones de libc

- **Hay que comprobar los errores**
- Las llamadas al sistema suelen retornar un valor negativo en caso de error (`NULL` si retornan un puntero). Así se detecta si hay error.
- Si lo hay, puedes consultar una variable llamada `errno`, que contendrá el valor que describe el error que se ha producido (descrito en la página de manual correspondiente).

Errores

- `perror`: imprime la cadena asociada a `errno`. Si se pasa como argumento una cadena, escribe primero la cadena y después el error.
- `strerror`: devuelve la cadena asociada al error que se le pasa. Las función `strerror_r` es la versión *thread safe*.
- `warn`: escribe el error de la última llamada concatenado después de la cadena con formato que se le pasa.
- `warnx`: lo mismo pero sin el error de la última llamada.

```
void perror(const char *s);
char * strerror(int errnum);
void warn(const char *fmt, ...);
void warnx(const char *fmt, ...);
```

Trazando llamadas

- El comando `strace` nos permite trazar las llamadas al sistema que hace un proceso. P. ej.:

  ```
  strace -p 3234
  ```

- El comando `ltrace` nos permite trazar las llamadas a funciones de biblioteca. P. ej.:

  ```
  ltrace -p 3234
  ```

Carga de procesos

- Fuente $\rightarrow$ Compilación $\rightarrow$ Enlazado $\rightarrow$ Binario (ejecutable).
- El fichero binario contiene la información para crear un proceso, organizadas en secciones. Hay distintos formatos: ELF o Extensible Linking Format, PE (Windows), a.out (Unix antiguos), etc.
- El binario es simplemente una receta para crear un proceso para ejecutar el programa.
- En su cabecera se indica la arquitectura, el tamaño/**offset** de las tablas y secciones, y el punto de entrada (dirección para comenzar a ejecutar).

Enlazado

El enlazado puede ser:

- Estático: cuando se genera el binario ejecutable, se incluye todo el código necesario en el binario (el del programa y el código que usa de todas las bibliotecas).
- Dinámico: el binario no incluye el código de las bibliotecas. En *tiempo de ejecución*, el sistema operativo carga/enlaza las bibliotecas que intenta usar el binario.
  - Son bibliotecas compartidas (shared library) cuando los distintos procesos que usan una misma biblioteca la comparten (esto es, esa biblioteca sólo se carga una vez en memoria).

ELF: cabecera

```
#define EI_NIDENT 16
typedef struct{
  unsigned char  e_ident[EI_NIDENT]; //magic, 32/64bit, encoding...
  Elf32_Hal      e_type;      // executable, relocal, shared object...
  Elf32_Half     e_machine;   // intel, sparc, M68K, mips...
  Elf32_Word     e_version;   // version of elf
  Elf32_Addr     e_entry;     // entry point (virtual addr)
  Elf32_Off      e_phoff;     // offset for PH table (process segments)
  Elf32_Off      e_shoff;     // offset for SH table (elf sections)
  Elf32_Word     e_flags;     // flags for CPU
  Elf32_Half     e_ehsize;    // size of this header
  Elf32_Half     e_phentsize; // size of the PH table
  Elf32_Half     e_phnum;     // entries in the PH table
  Elf32_Half     e_shentsize; // size of the SH table
  Elf32_Half     e_shnum;     // entries in the SH table
  Elf32_Half     e_shstrndx;  // index of the string table in SH table
} Elf32_Ehdr;
```

- Section Header Table (SH): describe las **secciones** del fichero ELF.
- Program Header Table (PH): describe los **segmentos** del proceso que ejecute el binario.

ELF: secciones

Secciones: tienen la información necesaria para **enlazar un objeto con el fin de generar un binario ejecutable**. Tienen la información necesaria *en tiempo de enlazado*. Algunas importantes son:

- `.text`: código (instrucciones) del fichero.
- `.data`, `.rodata`: variables inicializadas.
- `.bss`: descripción de las variables sin inicializar.
- `.plt`, `.got`, `.got.plt`, `.dynamic`: información para el enlazado dinámico.
- `.symtab`, `.dynsym`: tabla de símbolos, que es información sobre las variables que es útil para depurar los programas.
- `.strtab`, `.dynstr`: tabla con las cadenas de texto de la tabla de símbolos y las secciones.

ELF: segmentos

- Un ELF contiene información para poder **cargar el binario en memoria** para ejecutarlo.
- Un segmento es en realidad una parte de la región de memoria de *un proceso* (es parte de la estructura de datos que describe al proceso en el kernel).
- ELF describe cómo serán los segmentos cuando se cargue el binario a memoria.
- Un segmento puede contener información descrita en 0 o más secciones.
- Un segmento tiene atributos (solo-lectura, lectura/escritura, ejecutable), indica dónde se tiene que cargar en memoria, alineación, etc.

ELF: símbolos

- El enlazador usa la tabla de símbolos para resolverlos: las referencias a los distintos objetos exportados por un fichero se traducen en direcciones relativas en el código generado.
- Los depuradores también usan la tabla de símbolos: facilita la depuración.
- La tabla de símbolos **no es necesaria** para ejecutar un programa. El sistema operativo no necesita esa información para cargar el ejecutable.
- El comando `strip` elimina la tabla de símbolos de un fichero.
- Para cada símbolo se guarda: nombre, tamaño, ámbito (local, global), tipo (función, variable,etc.), ...

ELF: símbolos

El comando `nm` muestra los símbolos de un fichero objeto / ejecutable y su dirección de **memoria virtual** cuando se ejecute el programa. Algunos de los tipos de símbolos son:

- T: indica que está en el segmento de texto, donde hay instrucciones (text). Son subprogramas, etiquetas, etc.
- D: indica que está en el segmento de datos (data). Normalmente variables globales inicializadas.
- R: indica que es una variable de solo-lectura (read-only). Por ejemplo, literales de tipo string.
- B: indica que está en el segmento de datos sin inicializar (bss). Normalmente variables globales sin inicializar.
- U: indica que es un símbolo que está pendiente de resolver (undefined). Ese símbolo es externo, y todavía no se sabe dónde estará. Se resolverá al ejecutar el programa.

Carga de un ejecutable

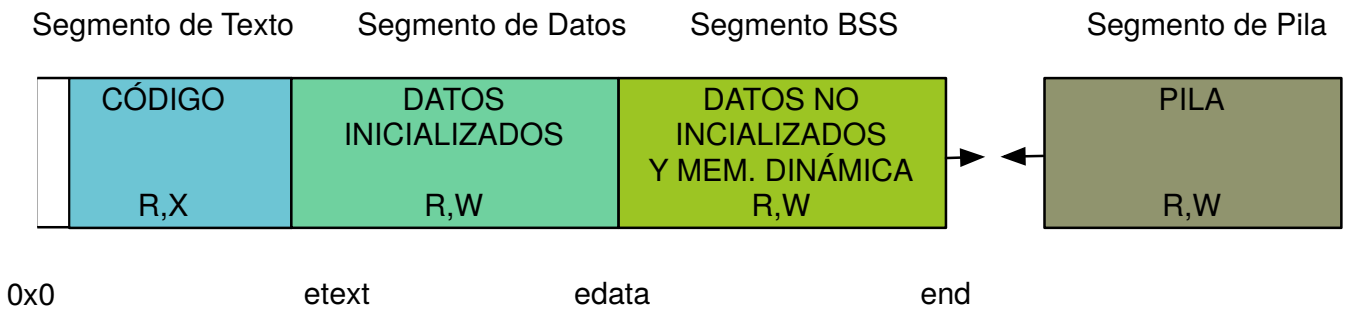
Cuando se ejecuta un fichero, el **cargador**[1] (que forma parte del kernel), hace lo siguiente:

1. Comprueba si se puede ejecutar (permisos, etc.).
2. Comprueba el tipo de ejecutable (interpretado o binario). Si el fichero empieza por los caracteres `#!`, es interpretado: se ejecutará el interprete indicado a continuación, con la ruta de este fichero como argumento.
3. Copia las secciones del ELF a memoria, que se organiza en *segmentos*.
4. Copia los argumentos del programa y variables de entorno (`argc`. `argv`, `envp`) a la pila del proceso.
5. Inicializa el contexto del proceso (registros, etc.) y sus atributos: se asigna **PID** (Process ID), etc.
6. Pone el proceso a ejecutar, comenzará por su *punto de entrada*.

[1] No confundir con el *boot loader* que realiza una tarea similar pero en el arranque.

Carga de un ejecutable

Esta es la representación general de la memoria de un proceso en arquitecturas x86/AMD64:



- Las bibliotecas dinámicas compartidas se proyectan en regiones de la memoria situadas entre el BSS y la pila.
- La memoria dinámica se obtiene una zona del BSS que se denomina *heap*.
- La `libc` hace crecer el segmento BBS cuando consumimos mucha memoria dinámica llamando a `malloc`. Para ello, usa la llamada al sistema `brk`.

Memoria virtual

- El proceso maneja únicamente direcciones de memoria virtual.
- El sistema operativo usa **paginación en demanda**: el programa se va cargando en memoria poco a poco según se demanda el acceso a sus direcciones de memoria.
- Si se borra o se sobreescribe un binario, los procesos pueden fallar.
- Hay segmentos que se pueden *compartir* entre procesos (texto, bibliotecas).

Enlazado dinámico

- Si el ejecutable fue enlazado estáticamente, tiene dentro todo lo necesario para ejecutar...
- ... pero cuando el enlazado es dinámico, el **enlazador dinámico**[2] debe resolver los símbolos que están sin definir: relocalizaciones.
- Básicamente, una relocalización es:
  "reemplaza el valor de X bytes que está en el offset Y por la dirección del símbolo externo S"
- **Lazy binding**: la resolución no se hace de golpe a la hora de comenzar, se hace a medida que se van usando los símbolos.
- No siempre es así: se puede forzar a que se resuelvan todos los símbolos al principio (RELRO o relocation read-only).

[2] En linux, `/lib/ld-linux.so`

Lazy binding

¿Cómo funciona?

- GOT (Global Offset Table): contiene las direcciones de los símbolos, esto es, la dirección a la que tiene que saltar un *trampolín*.
- PLT (Procedure Linkage Table): contiene el código (stubs) de los *trampolines*.
- Las relocalizaciones se realizan en la GOT, no en el texto (código) $\rightarrow$ se *parchea* la GOT, esto es, se escriben las direcciones allí según se van resolviendo.
- El texto siempre llama a la entrada en la PLT para la función a la que desea llamar (esto es, al *trampolín*).
- La primera vez que se llama a la entrada de la PLT para una función, se llama al enlazador dinámico para que resuelva el símbolo y *parchee* la entrada de la GOT. Después se llama a la función.
- En sucesivas llamadas, la GOT ya tiene *parcheada* la entrada y el trampolín salta a la función destino directamente.

Lazy binding

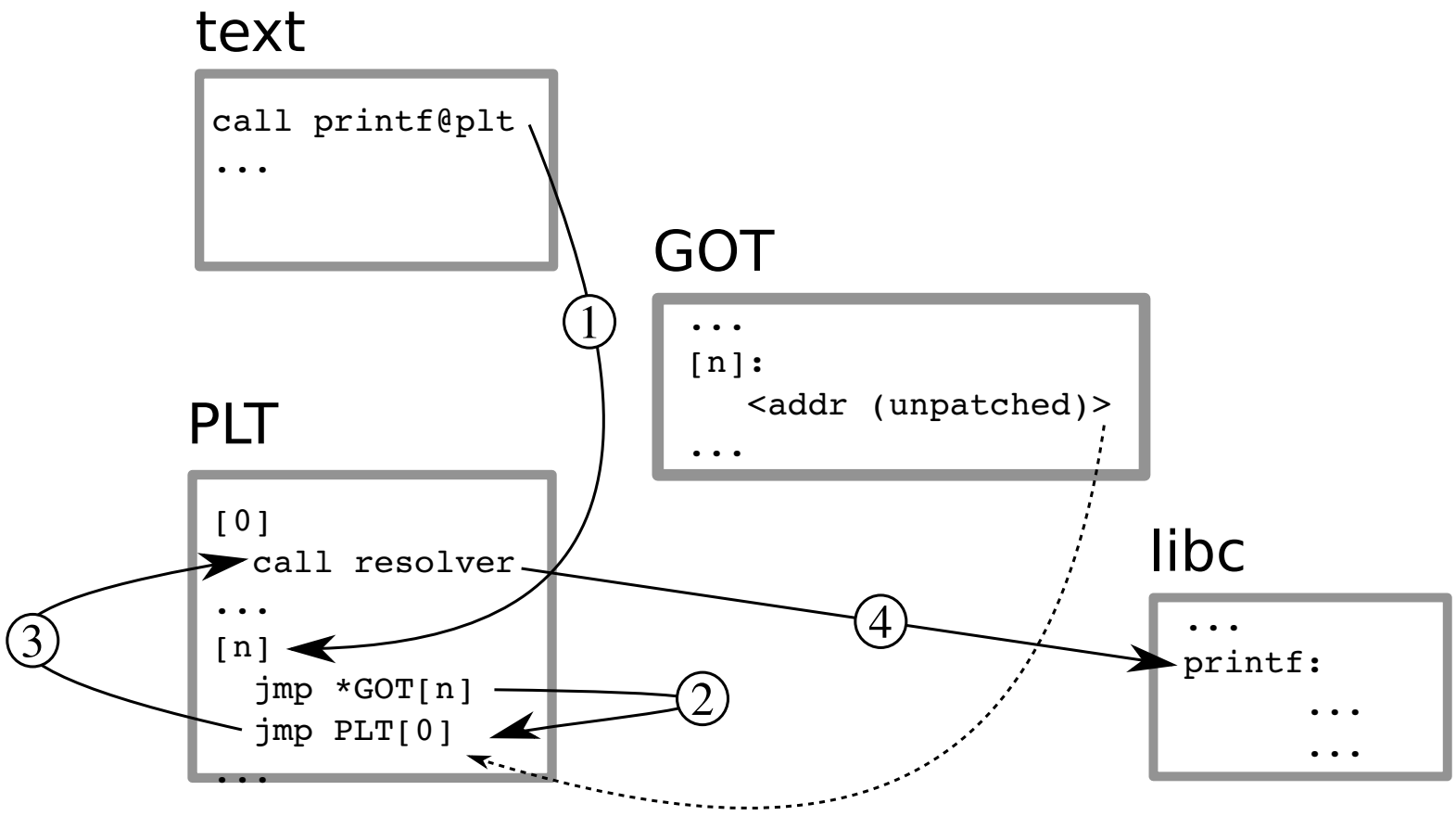
¿Cómo funciona? Supongamos que el proceso llama a `printf` **por primera vez**:

1. El texto en realidad llama a la función `printf@plt`.
2. La función `printf@plt` salta a la dirección de la entrada de `printf` en la GOT. Dicha entrada de la GOT está sin parchear, y está inicializada a la dirección siguiente instrucción (del propio código de `printf@plt`).
3. Dicha función llama al enlazador dinámico para resolver el símbolo y parchear la entrada de la GOT. Pasa como argumento el índice del símbolo en la tabla.
4. Después de resolver, se salta a la función `printf`.

Las llamadas **posteriores** saltarán, en el paso 2, a la función `printf` de la libc, porque la GOT ya está *parcheada*.

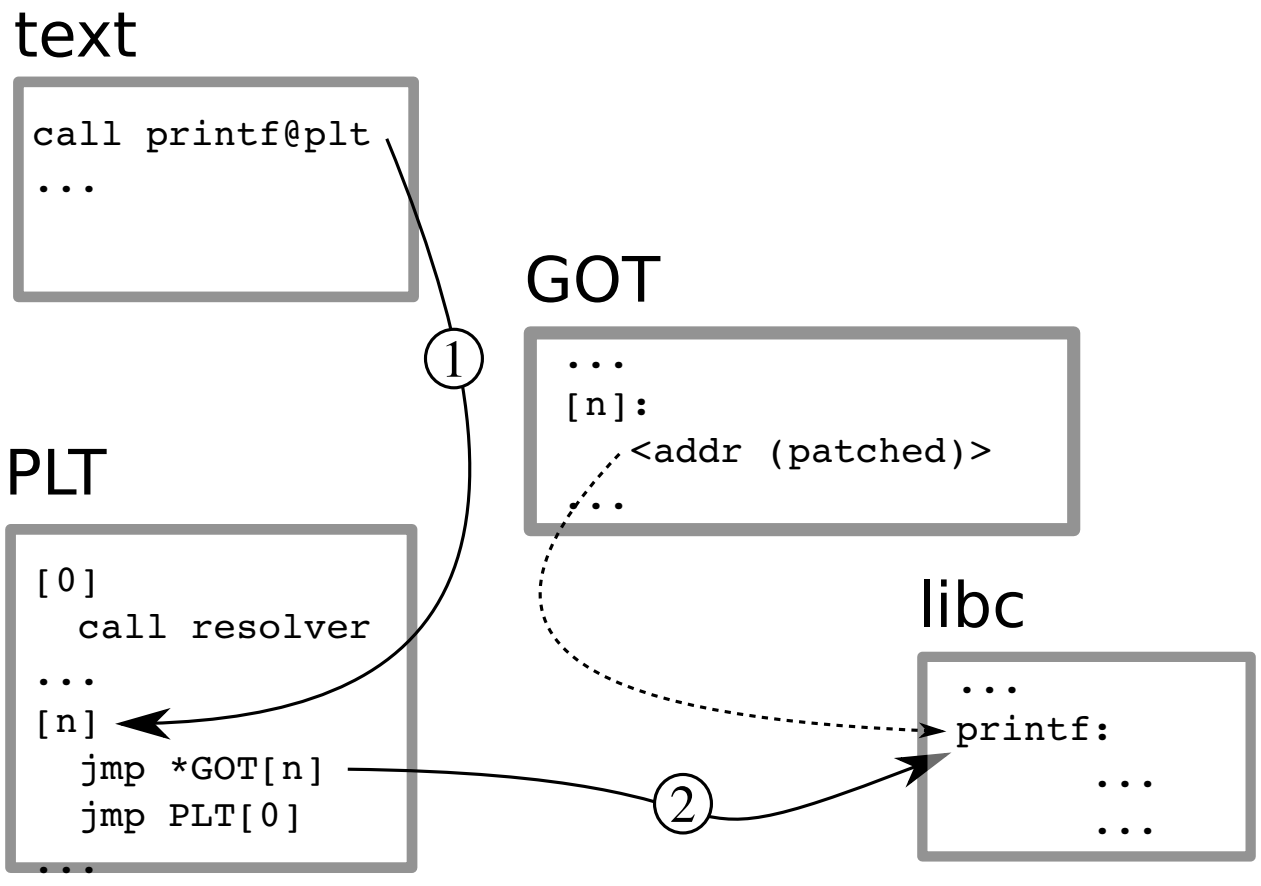
Lazy binding

Primera llamada:



Lazy binding

Siguientes llamadas:



Bibliotecas

- El comando `ldd` muestra las bibliotecas dinámicas que necesita un programa, junto con la dirección de memoria donde serán cargadas.
- Una biblioteca puede cargarse en la misma dirección de memoria para todos los procesos que la utilizan, pero no siempre es así. Si tenemos PIC (código independiente de posición), no ocurrirá de esa forma.

Bibliotecas

¿De dónde se saca la biblioteca? Depende del sistema y de su configuración. Por ejemplo:

1. Directorios de la variable de entorno `LD_PRELOAD`
2. Si el símbolo contiene barras ("/"), se carga de esa ruta (relativa o absoluta)
3. Directorios de la sección `DT_RPATH` del binario o del atributo `DT_RUNPATH` del binario
4. Directorios de la variable de entorno `LD_LIBRARY_PATH`
5. En el directorio `/lib`
6. En el directorio `/usr/lib`

Muerte de un proceso

- Cuando `main` retorna, la función que la llamó realiza una llamada al sistema `exit` para acabar. Se puede llamar desde cualquier parte del programa para terminar:
- El sistema terminará con el proceso y liberará los recursos asociados.
- El parámetro que se le pasa a `exit` determina el estado del proceso al acabar:
  - éxito (`EXIT_SUCCESS`, valor 0)
  - fallo (`EXIT_FAILURE`, valor distinto de 0).

```
void exit(int status);
```

Muerte de un proceso

- `err`: escribe el error de la última llamada concatenado después de la cadena con formato que se le pasa, y después termina el proceso con el estatus indicado.
- `errx`: escribe un error con la cadena con formato que se le pasa, y termina el proceso con el estatus indicado.

```
void err(int eval, const char *fmt, ...);
void errx(int eval, const char *fmt, ...);
```

Entorno

- Las variables de entorno son strings.
- En Linux y en general, se almacenan en área de usuario[3].
- El proceso hereda de su creador una **copia** de sus variables de entorno.
- El tercer parámetro de `main` es un array de cadenas con las variables de entorno, terminado en NULL.
- La libc mantiene una variable global llamada `environ` que apunta a las variables de entorno.
- Al crear el proceso, se meten en la pila. Si se definen nuevas variables de entorno durante su vida, se mueven al heap.

[3] En algunos sistemas se almacenan en área de kernel, p. ej. Plan 9.

Entorno

- `getenv`: devuelve el valor de una variable de entorno. No devuelve memoria dinámica.
- `setenv`: escribe el valor de una variable de entorno. Si existe, la puede sobrescribir o no (tercer parámetro). Internamente, hace una copia de la cadena.
- `unsetenv`: elimina una variable de entorno.

```
char * getenv(const char *name);
int setenv(const char *name, const char *val, int o);
int unsetenv(const char *name);
```

Propiedades

- El kernel mantiene la información de los procesos en tabla de procesos.
- Para cada proceso, mantiene una estructura de datos con:
  - PID: número que identifica el proceso.
  - PPID (PID de su creador)
  - Estado
  - Prioridad
  - Registros (para cambios de contexto).
  - Directorio de trabajo
  - UID, GID: credenciales del proceso (usuario y grupo).
  - Segmentos: text, data, bss, stack
  - Descriptores de fichero (ficheros abiertos)
  - ...
- El comando `ps` nos da información sobre los procesos.

Propiedades

- En linux podemos inspeccionar las propiedades de los procesos en `/proc`[4].
- Hay un directorio por PID. Dentro tenemos ficheros con información sobre cada proceso. Ejemplos:
  - stat:propiedades del proceso (PID, PPID, estado, ...).
  - cmline: línea que se usó para ejecutar.
  - exe: el programa que está ejecutando.
  - fd, fdinfo: información sobre ficheros abiertos.
  - maps: regiones de memoria del proceso.
  - mem: memoria del proceso.
  - ...

[4] man 5 proc

Credenciales

- El proceso ejecuta a nombre de un usuario y un grupo.
- El sistema aplica el **control de acceso** en base a las credenciales del proceso.
- La información sobre usuarios y grupos se almacena en los ficheros `/etc/passwd`, `/etc/shadow` y `/etc/group`.
- El UID es un número que identifica a un usuario. El GID identifica a un grupo. Un usuario puede pertenecer a múltiples grupos.
- Durante la vida del proceso las credenciales pueden cambiar.

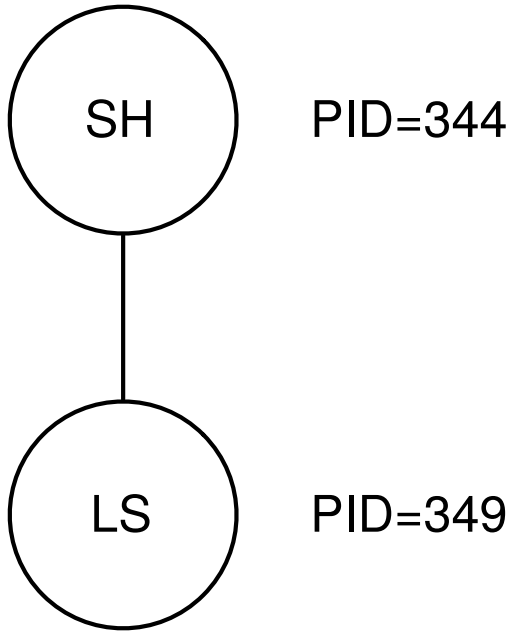
Llamadas al sistema

- `getpid`: devuelve el PID del proceso.
- `getppid`: devuelve el PID del creador.
- `getuid`: devuelve el UID.
- `getgid`: devuelve el GID.
- `getcwd`: devuelve el directorio de trabajo. Lo copia en el buffer que pasamos. Si buff es NULL, reserva memoria dinámica.
- `chdir`: cambia el directorio de trabajo. Retorna -1 en error.

```
pid_t getpid(void);
pid_t getppid(void);
gid_t getgid(void);
uid_t getuid(void);
char * getcwd(char *buf, size_t size);
int chdir(const char *path);
```

Hasta ahora...

- ... sólo hemos creado procesos con el shell.
  P. ej.
  `$> ls`



Llamadas al sistema

- Una para crear un proceso (fork) y otra para ejecutar un programa (exec).
- Podemos crear un nuevo flujo de control, para ejecutar el mismo programa o para ejecutar otro.
- Podemos configurar el nuevo proceso antes de que ejecute otro programa.
- Se podría hacer todo en una función, pero tendría demasiados parámetros.

Fork

- `fork` crea un nuevo proceso. Se crea un proceso con nuevo PID. En el padre, retorna el PID del hijo. En el hijo, la llamada retorna 0. En caso de error, retorna -1.
- En Linux, la función `fork` en realidad realiza la llamada al sistema `clone`.

```
int fork(void);
```

Fork

- El hijo es un clon exacto del padre.
- Padre e hijo ejecutan concurrentemente.
- El hijo es totalmente independiente del padre: abstracción de proceso.
- Se *"copia"* la memoria: TEXT, DATA, STACK. ¿Qué valor tienen las variables?

Condiciones de carrera

- No se sabe el orden en el que se van a ejecutar las instrucciones de los dos procesos.
- Cuando el resultado final depende de la carrera entre los dos procesos: condición de carrera.
- En este caso, los procesos no comparten memoria, pero sí comparten otros recursos del sistema (p. ej.: ficheros).
- Programación concurrente: evitar condiciones de carrera.

Execv

- `execv`: ejecuta un programa. `path` es la ruta del fichero ejecutable (binario o interpretado) que queremos ejecutar. `argv[]` es un array de strings con los argumentos para el programa, que tiene que terminar en un NULL. El primer elemento del array es el nombre del programa.

```
int execv(const char *path, char *const argv[]);
```

Execl

- `execl`: similar, pero pasando los argumentos de otra forma. Ideal para cuando se saben los argumentos de antemano. El último argumento tiene que ser siempre NULL.

```
int execl(const char *path, const char *arg, ...);
```

fork + exec

- Las dos funciones juntas nos permiten crear procesos para ejecutar cualquier programa.
- Desde `init`, todos los procesos se crean así.

Wait

- `wait`: retorna cada vez que un hijo cambia de estado. Es la forma de esperar a que terminen los hijos. No notifica el cambio de estado de los procesos nietos, etc.
- Retorna -1 si no hay hijos por los que esperar o en caso de ser interrumpida. En otro caso, retorna el PID del hijo.
- Hay que usar las macros descritas en la página de manual para discriminar entre los estados y conseguir el status de salida del hijo.

```
pid_t wait(int *wstatus);
```

Wait

Macros para ver qué ha pasado:

- `WIFEXITED(status)`
  Verdadero si el proceso ha terminado llamando a `exit()`.
- `WEXITSTATUS(status)`
  Evalúa a los 8 bits que describen el estado de salida del proceso (valor que el hijo pasó a `exit()`).
- `WIFSIGNALED(status)`
  Verdadero si el proceso ha terminado por una señal.
- `WIFSTOPPED(status)`
  Verdadero si el proceso no ha terminado pero ha sido parado.

Waitpid

- `waitpid`: espera hasta que el proceso con el PID indicado en el primer argumento acabe.

```
pid_t waitpid(pid_t pid, int *wstatus, int op);
```

Procesos muertos

- Cuando un proceso muere, no se puede eliminar su estructura de la tabla de procesos hasta que el padre no recoja su estatus.
- Un proceso muerto pendiente de que se recoja su status es un *zombie*.
- Si el padre muere antes que los hijos, estos pasan a ser huérfanos (*orphan*).
- Alguien se tiene que hacer cargo de esperar por los huérfanos. Por eso, los huérfanos pasan a ser hijos de `init`.
- `init` llama a `wait` por todos sus hijos para evitar que se queden *zombies* para siempre.

En el shell

- `comando &`
  el comando se ejecuta de forma asíncrona (segundo plano o *background*), el shell no espera a que termine ese comando para continuar.
- `$$`
  variable con el PID del shell.
- `$!`
  variable con el PID del último comando ejecutado en segundo plano.
- `$?`
  variable con el estatus del último comando.

En el shell

- `&&`
  se ejecuta el segundo comando si el primero termina con éxito. Ejemplo:
  `test -f /tmp/a && echo el fichero existe`
- `||`
  se ejecuta el segundo comando si el primero termina con fallo. Ejemplo:
  `test -f /tmp/a || echo el fichero no existe`

En el shell: Job Control

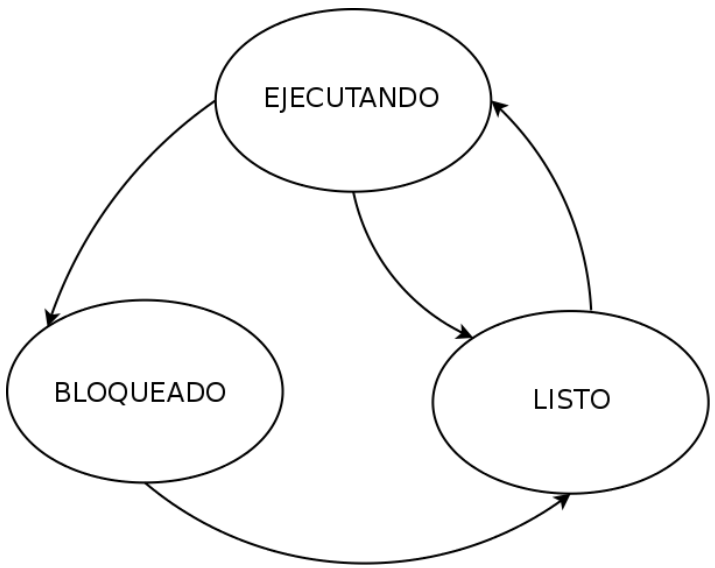
Cada comando que está ejecutando es un trabajo (*job*):

- `jobs`
  Lista los trabajos actuales.
- `Ctrl + c`
  Interrumpe el trabajo que está en primer plano.
- `Ctrl + z`
  Para el trabajo que está en primer plano.
- `bg`
  Reanuda, en segundo plano, un trabajo parado.
- `fg`
  Reanuda, en primer plano, un trabajo parado.

Planificación

Para la planificación, podemos resumir los posibles estados de un proceso en:

- Ejecutando
- Listo para ejecutar
- Bloqueado



Planificación

- El **planificador (scheduler)** del kernel se encarga de gestionar qué proceso ejecuta en la CPU (o CPUs).
- **Política** vs **Mecanismo**
- **Concurrencia** vs **paralelismo**.

Planificación: mecanismo

- **Cambio de contexto (simplificado)**
  1. Salvar el estado de los registros del procesador en la estructura de datos que representa al proceso en el kernel.
  2. Cargar el estado de los registros del proceso entrante en el procesador El contador de programa al final.

  Los **cambios de contexto** son costosos.

Planificación: políticas

- Ya sé sacar un proceso y meter otro... ¿Cómo reparto las CPUs?
- Los procesos se pueden ver como ráfagas de operaciones de CPUs y operaciones de entrada/salida.
- Algunos procesos están dominados por CPU y otros procesos están dominados por entrada/salida.

Criterios, ¡no se puede todo a la vez!

- **Justicia (fairness)**: todos los procesos tienen su tiempo de CPU.
- **Eficiencia (efficiency)**: sacar máximo partido a la CPU.
- **Respuesta interactiva**: es importante el tiempo de espera en la cola hasta que se empieza con él.
- **Respuesta (turnaround)**: tiempo de entrega del resultado de un cómputo.
- **Rendimiento (throughput)**: número de trabajos acabados por unidad de tiempo.

Planificación

La planificación puede ser:

- No expulsiva (colaborativa, non-preemptive): el proceso abandona la CPU porque ha terminado su ráfaga de CPU y se bloquea haciendo E/S, o por decisión propia.
- Expulsiva (preemptive): el planificador puede expulsar a los procesos cuando lo decida, aunque no hayan terminado su ráfaga de CPU.

Planificación no expulsiva: FCFS

FCFS: First Come First Serve

- Los procesos listos para ejecutar se ponen en un FIFO.
- Efecto *convoy*: llega un proceso dominado por CPU y la acapara. Deja esperando mucho tiempo a los procesos dominados por E/S que están listos para ejecutar ráfagas cortas de CPU $\rightarrow$ se malgasta capacidad de E/S.
- Tiempo promedio de espera alto.
- Sencillo y fácil de implementar.

Planificación no expulsiva: SJF

SJF: Shortest Job First.

- En la cola tienen prioridad los procesos con ráfaga más corta.
- Tenemos que saber el volumen del trabajo de antemano.
- Mejora el tiempo de espera promedio.
- Problema: hambruna. ¿Qué pasa con las ráfagas largas?

Planificación expulsiva: SRTF

SRTF: Shortest Remaining Time First.

- Es como SJF, pero expulsivo.
- Cuando entra en la cola de listos para ejecutar un proceso con una ráfaga de CPU más corta que lo que le queda al proceso actual, se expulsa al actual.
- Tiene el mismo problema: hambruna.

Planificación expulsiva: Round-Robin

Round Robin:

- Se asigna la CPU en *cuantos*: tiempo máximo que el proceso puede estar usando la CPU.
- Se rota por los procesos que están listos para ejecutar.
- Cuando se agota el *cuanto*, se expulsa al proceso.
- El proceso puede dejar la CPU antes de que acabe su *cuanto* (p. ej. si se queda bloqueado).
- Los procesos nuevos entran por el final de la cola.

Planificación de procesos: Round-Robin

Pros y contras:

- Aumenta la respuesta interactiva.
- Reduce el rendimiento (throughput) por los cambios de contexto.

Problema: ¿Cómo se elige el *cuanto*?

- Si es pequeño, se desperdicia mucha CPU en los cambios de contexto.
- Si es grande, las aplicaciones interactivas sufren.

Planificación de procesos: prioridades

- No todos los procesos tienen la misma importancia.
  - P. ej. un reproductor de video vs. un cliente de correo.
- En general, la prioridad pueden ser **estática** o **dinámica**.
- Problema: inanición (*starvation*). Solución: tener en cuenta la edad del proceso (*aging*).

Planificación de procesos: colas multinivel con retroalimentación

En general:

- Número de colas.
- Algoritmo para cada cola.
- Método para subir el proceso a una cola de mayor prioridad.
- Método para bajar el proceso a una cola de menor prioridad.
- Método para determinar la cola en la que empieza un proceso.

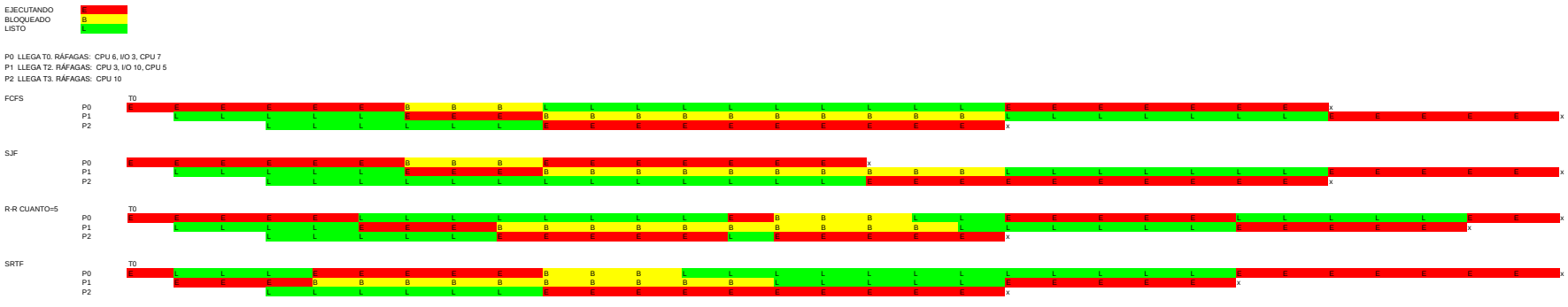
Planificación de procesos: colas multinivel con retroalimentación

Ejemplo particular: Round-Robin con prioridades dinámicas.

- Múltiples colas, según prioridad.
- R-R con los procesos de cada cola.
- Si hay procesos listos en una cola, no se atienden las colas de menor prioridad.
- Cuando un proceso agota su *cuanto*, se baja su prioridad.
- Cuando un proceso no agota su *cuanto*, se sube su prioridad.

Diagramas de planificación

Sirven para representar la planificación. Por ejemplo:.



Planificador para multiprocesasores

Cuando tenemos múltiples CPUs, el planificador debe especializarse para el tipo de arquitectura:

- SMP (Symmetric Multi-Processor): todas las CPUs son iguales y el acceso a las partes de la memoria física es el mismo para todas (UMA).
- NUMA (Non-Uniform Memory Access): algunas partes de la memoria y ciertos dispositivos están más cerca de unas CPUs que de otras.
- CPUs heterogéneas: algunas CPUs son menos potentes que otras o con distintas ISA que ofrecen instrucciones más apropiadas para ciertos trabajos (CPUs, GPUs, etc.).

Prioridades en Linux

- La prioridad (*niceness*) va del valor -20 (máxima prioridad) al 19 (mínima prioridad).
- La prioridad se hereda del proceso padre.
- El comando `nice` sirve para ejecutar un programa indicando el nivel de prioridad. P. ej.:

```
nice -n 5 $HOME/myprogram
```

- El comando `renice` sirve para cambiar la prioridad de un proceso. P. ej.:

```
renice -5 5232
```

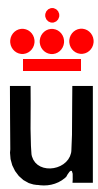
Ficheros

Sistemas Operativos

Enrique Soriano, Gorka Guardiola Múzquiz

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

Ficheros

- Tradicionalmente: datos que persisten tras la ejecución del proceso y/o del sistema, y que son compartidos por distintos procesos y/o sistemas.
- Requisitos:
  - Gran tamaño
  - Durabilidad
  - Acceso concurrente
  - Protección
- Más general: es una **interfaz**. P. ej. puede representar un dispositivo, puede ser almacenamiento volátil (ramfs), etc.

Ficheros

- Estructura: fichero simple (Unix), Resource Forks (Mac)...
- Nombre: ¿extensión?
- ¿Tipos?
- Ya hemos usado ficheros desde nuestros programas, p. ej. `printf()`.
- Operaciones básicas: open, read, write, close.
- El **sistema de ficheros** es el componente que proporciona los ficheros (i.e. el que implementa esas operaciones básicas). Suele estar dentro del kernel, pero también puede estar en área de usuario.

Discos: estructura interna

- **Bloque (o sector) físico** $\rightarrow$ determinado por el HW. Tradicionalmente de 512 bytes, algunos fabricantes están pasando a 4 Kb.
- **Bloque lógico** $\rightarrow$ algunos discos pueden manejar bloques de distintos tamaños (traduce de lógicos a físicos internamente).
- Los llamaremos **bloques de disco**.
- El **sistema de ficheros** también maneja *bloques*, a otro nivel de abstracción. A veces se llaman también *bloques lógicos*, *clusters*,... Los llamaremos **bloques del sistema de ficheros**.

Discos: estructura interna

- Si no coinciden el tamaño de bloque de disco y de bloque de sistema de ficheros, se empaqueta: un bloque de sistema de ficheros es un conjunto de bloques de disco.
- Siempre hay fragmentación interna en los bloques.
- ¿Tamaño? compromiso entre:

$$\uparrow \text{tamaño de bloque} \Rightarrow \uparrow \text{fragmentación}$$
$$\uparrow \text{tamaño de bloque} \Rightarrow \uparrow \text{tasa de transferencia}$$

Discos

Acceso al disco:

- Secuencial: se accede registro a registro (contiguos).
- Aleatorio: se accede a cualquier registro inmediatamente.

¿Y los discos duros actuales? Caches flash, discos SSD, híbridos, etc.

Discos

- Direccionamiento de bloques de disco:
  - CHS: Cylinder, Head, Sector. Esquema viejo.
  - LBA: Linear.
- Así *eran*:

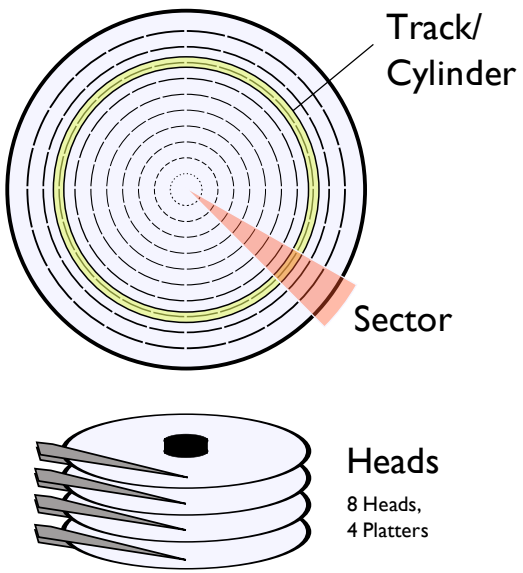


Imagen: Public Domain, Wikipedia

Discos

- Algoritmo de planificación de disco: ¿cómo organiza el SO las operaciones?
- El algoritmo depende totalmente del tipo de disco que usemos.
- ¿Cómo se relaciona el direccionamiento LBA con la geometría real del disco? Se sigue suponiendo que sí. ¿Se puede generalizar? No. ¿Hay caches internas en el disco? ¿Es un disco de verdad o es un disco virtual?
- Los clásicos:
  - FIFO: es justo, no altera el orden de las operaciones. Lento en discos mecánicos.
  - Shortest Seek First: no es justo, puede provocar hambruna, desordena operaciones.
  - Ascensor: compromiso justicia/eficiencia, no provoca hambruna, desordena operaciones.
  - ... (hay muchos otros)

Particiones

- Un disco se puede dividir en partes: particiones.
- En linux, el comando `fdisk` permite manipular las particiones de un disco.

Particiones PC (BIOS)

Esquema tradicional de los PC, ha sido reemplazado por UEFI:

- MBR: Primer bloque del disco (512 bytes).
  - Cargador primario: 440 bytes.
  - Tabla de particiones primarias: 4 entradas de 16 bytes.
    - Arrancable/no arrancable.
    - Dirección del primer bloque (CHS).
    - Tipo de partición (ntfs, linux, linux swap, plan9, ...).
    - Dirección del último bloque (CHS).
    - Dirección del primer bloque (LBA).
    - Número de bloques de la partición.
  - Direcciones de 32 bits: como mucho discos de 2TB.

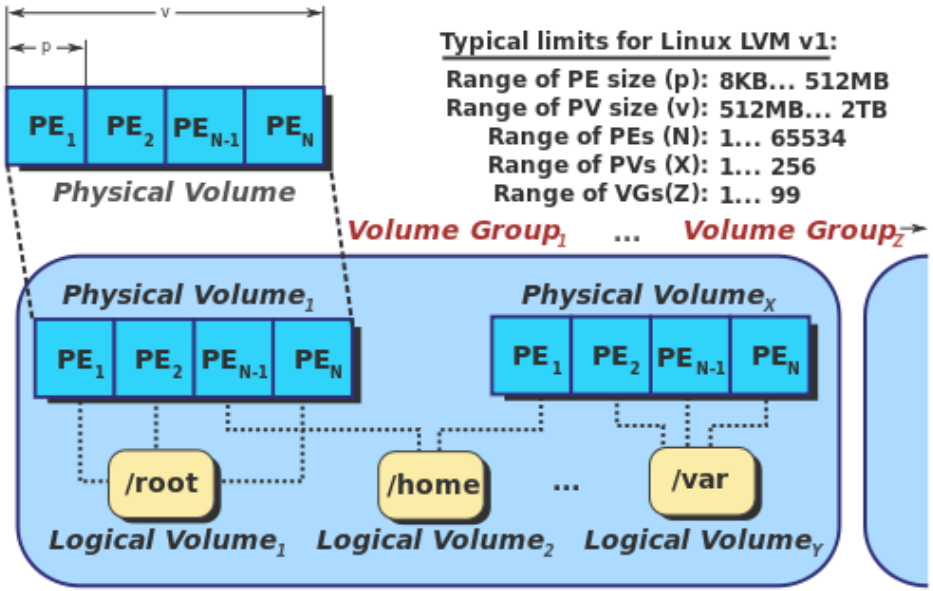
Particiones UEFI

GUID Partition Table (GPT) :

- Usa direcciones y tamaños de 64 bits.
- Primer bloque (LBA 0): legacy **MBR**.
- Segundo bloque (LBA 1): **GPT header**, con punteros para:
  - A partir de LBA 2 (2 al X): **GPT Table**: Array de bloques con la tabla de particiones. El array tiene un tamaño mínimo de 16 Kb independientemente del tamaño del sector físico. Las entradas en la tabla (128 bytes) contienen: tipo de particion, LBA del primer bloque, LBA del último bloque, atributos (*read-only*, oculta, etc.), nombre de particion...
- El primer bloque usable (aquí pueden empezar las particiones)
- El cargador está en una partición de tipo ESP (EFI System Partition, que es una FAT).
- ...
- Al final (LBA, de sz-2-X a sz-2) Copia de la tabla de particiones (**GPT Table**, copia del LBA 2 al X).
- Último bloque (LBA sz-1) Copia de la cabecera (**GPT HEADER**, copia del LBA 1).

Volúmenes Lógicos

- Los volúmenes lógicos (LV) pueden estar formados por distintos trozos (physical extends, PE) de distintos volúmenes físicos (PV). P. Ej. Linux LVM.



Imágenes de disco

- Un fichero que tiene dentro la estructura completa (bloque a bloque) de un disco duro, CD-ROM (ISO 9660), etc.
- Hay distintos formatos: iso, bin, dmg...
- Son útiles para máquinas virtuales, copias de seguridad, clonar sistemas, transmisión de discos por la red, etc.

Imágenes de disco: ejemplo

```
$> # creo un fichero de 1,44 MB con ceros
$> dd if=/dev/zero of=/tmp/f bs=512 count=2880
$> # formateo la imagen de floppy con VFAT
$> mkfs.vfat -v -c /tmp/f
$> # monto en el punto de montaje /mnt/floppy
$> mount -o loop /tmp/f -t vfat /mnt/floppy
$> # creo un fichero en el floppy
$> touch /mnt/floppy/afile
$> # ver el espacio de nombres
$> mount
$> # desmontar el volumen
$> umount /mnt/floppy
```

Sistemas de Ficheros

- El sistema de ficheros nos ofrece la abstracción de *fichero*: no pensamos en los bloques, para nosotros un fichero es una secuencia de bytes.
- ¿Cómo asigna los bloques a un fichero?
- Se puede implementar de distintas formas: asignación contigua, enlazada, enlazada con tabla, indexada

Asignación de espacio: contigua

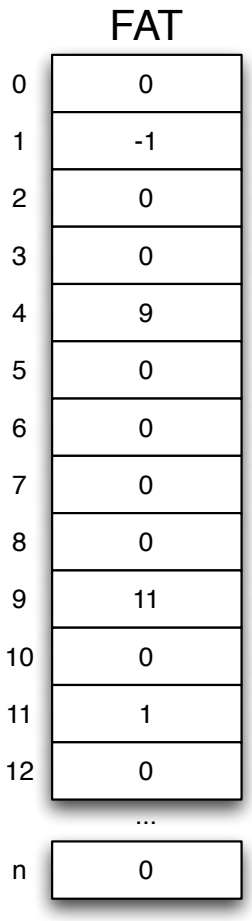
- Un fichero ocupa una serie de bloques contiguos en disco.
- Acceso rápido en discos de acceso secuencial.
- Rendimiento alto: un acceso para conseguir un dato cualquiera $\rightarrow$ la entrada de directorio contiene la dirección de inicio y la longitud del fichero.
- Problema: si hay asignación dinámica $\rightarrow$ fragmentación externa (huecos inusables).
- Problema: hay que saber el tamaño máximo del fichero cuando se crea.
- Tamaños limitados por los huecos $\rightarrow$ compactación.
- ¿Se usa? Sí: CD, DVD, ...

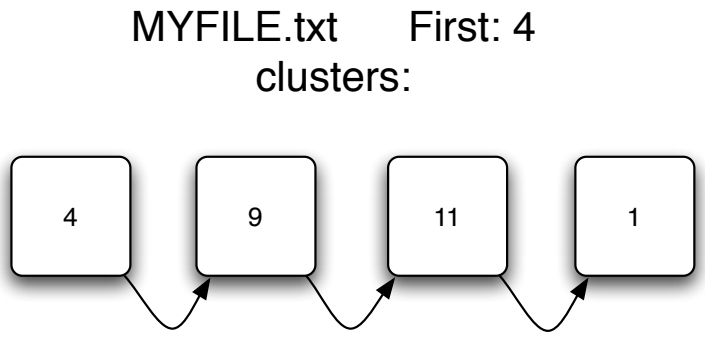
Asignación de espacio: enlazada con tabla

FAT es un caso de asignación de espacio usando una **lista enlazada** implementada con una tabla:

- Usa bloques del sistema de ficheros grandes: **clusters**.
- La tabla FAT tiene una entrada por cluster.
- El índice de la tabla es el número de cluster.
- La entrada de directorio tiene la referencia a la entrada primer cluster.
- La entrada del cluster actual referencia la entrada del siguiente cluster.
- Mejora del acceso aleatorio respecto a una lista sin tabla: no hay que leer los clusters del fichero para conseguir la dirección del cluster N.

FAT


FAT
0
0
1
-1
2
0
3
0
4
9
5
0
6
0
7
0
8
0
9
11
10
0
11
1
12
0
...
n
0


MYFILE.txt First: 4
clusters:
4
9
11
1

FAT

- Si queremos la tabla FAT en memoria principal $\rightarrow$ no puede ser muy grande $\rightarrow$ clusters grandes $\rightarrow$ fragmentación interna.
- El acceso aleatorio es más rápido que en lista enlazada normal.
- ¿Y si se estropea la tabla FAT?

Ejemplo: FAT32

- El sector 0 de la partición es el *boot sector* (también está duplicado en el sector 6). Contiene código del cargador secundario e información sobre el sistema de ficheros:
  - Nº de sectores, 4 bytes (Max. tam. de disco: 2 Tb)
  - Etiqueta del volumen.
  - Nº de copias de la tabla FAT.
  - Primer cluster del raíz.
  - ...

Ejemplo: FAT32

- Entrada de directorio (32 bytes):
  - Nombre del archivo, 8 Bytes.
  - Extensión del archivo, 3 Bytes.
  - Atributos del archivo, 1 Byte.
  - Reservado, 10 Bytes.
  - Hora de la última modificación, 2 bytes.
  - Fecha de la última modificación, 2 bytes.
  - Primer cluster del archivo, 4 bytes.
  - Tamaño del archivo, 4 bytes.
- Clusters de 32 Kb, 64 sectores.

Asignación de espacio: indexada

- Idea: se indexan los bloques de datos del fichero en un *bloque de indirección*.
- Bloque de indirección grande $\rightarrow$ desperdicio.
- Bloque de indirección pequeño $\rightarrow$ no soporta ficheros grandes.
- Para un acceso a datos, siempre son dos accesos a disco.

Asignación de espacio: indexada multinivel

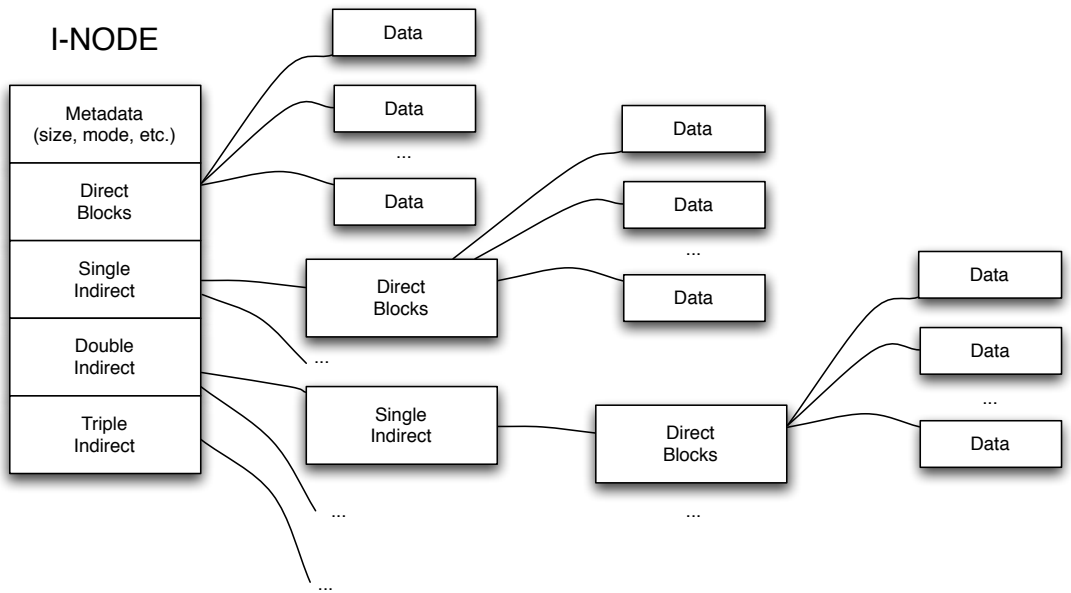
- Por niveles. P. ej. indirección doble:
  - Los bloques de índice de 1er nivel apuntan a bloques de índice de 2º nivel.
  - Los bloques de índice de 2º nivel apuntan a bloques de datos.
- Ejemplo: bloques de 4Kb con punteros de 4 bytes.
- Problema: Para los ficheros pequeños, desperdiciamos.
- Problema: Para un acceso a datos, n accesos a disco.

Asignación de espacio: indexada con esquema combinado

- Algunos bloques directos de datos.
- Algunos bloques de indirección simple.
- Algunos bloques de indirección doble.
- ...
- Nos quedamos con lo mejor de cada modelo anterior.

Ficheros en Unix: i-nodos

Es un caso de asignación indexada con esquema combinado:



Unix: Partición

- Bloque de arranque (boot block): código de arranque del sistema.
- Superbloque: tamaño del volumen, número de bloques libres, lista de bloques libres, tamaño del vector de i-nodos, siguiente i-nodo libre, cierres...
- Vector de i-nodos: representación de los ficheros.
- Bloques de datos.

Unix: i-nodos

- Cada fichero/directorio tiene un i-nodo asociado, definido por un número de i-nodo. El directorio raíz siempre tiene el número de i-nodo 2.
- La estructura se localiza en el vector indexando por el número de i-nodo.
- El OS mantiene una cache de i-nodos en memoria.

Unix: i-nodos

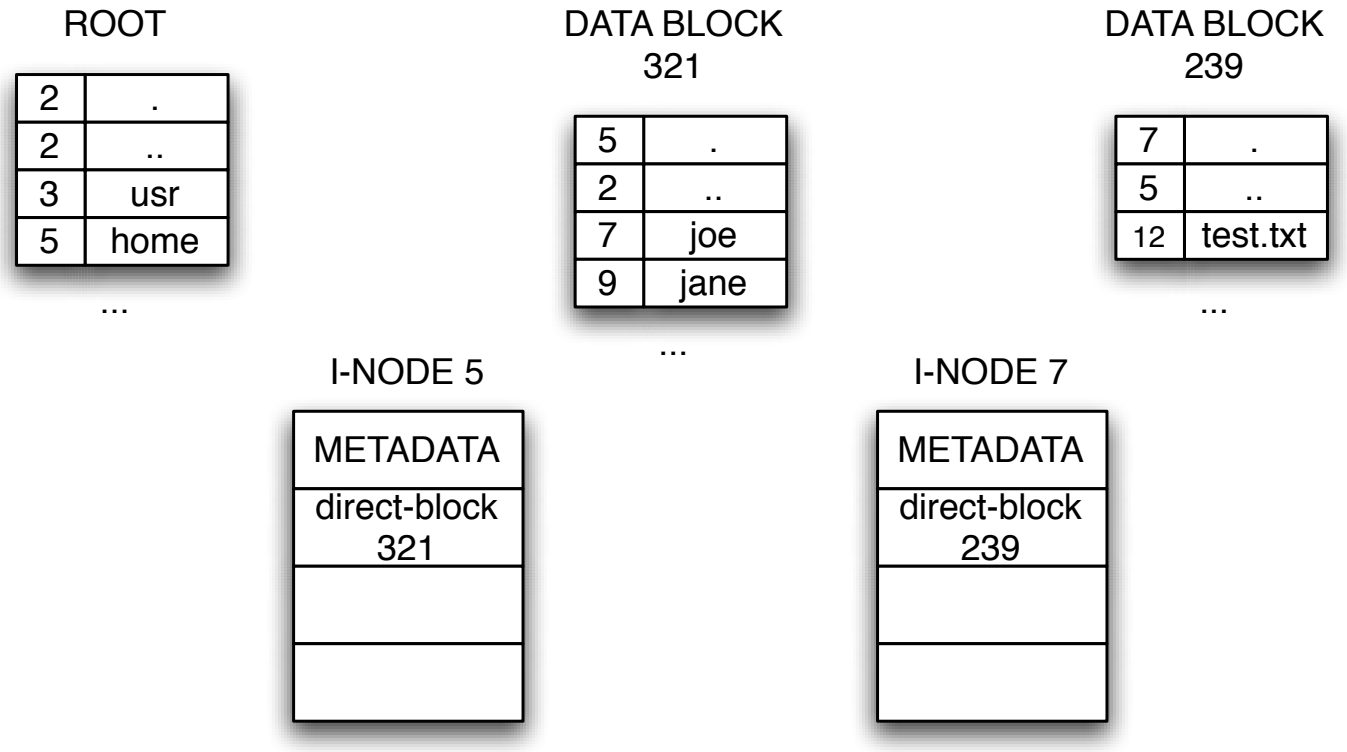
- La estructura contiene:
  - permisos
  - tiempos
    - acceso a datos (`atime`)
    - modificación de los datos (`mtime`)
    - modificación del i-nodo (`ctime`)
  - tamaño
  - dueño
  - tipo
  - número de bloques
  - contador de referencias (links)
- **No** contiene el nombre de fichero.

Unix: entradas de directorio

- Un directorio relaciona un nombre con un i-nodo: *entrada de directorio* (dentry).
- Un directorio tiene:
  - Su propio i-nodo
  - Bloques de datos con la lista de entradas de directorio.
- Entre las entradas, tiene la entrada de . (su i-nodo) y .. (i-nodo del padre).
- El kernel mantiene una cache de entradas de directorio en memoria.

Unix: ejemplo

/home/joe/test.txt



Unix: enlaces

- Enlaces duros (hard links): es otro nombre para el fichero, la entrada de directorio apunta al mismo i-nodo que el antiguo nombre. En general, no se permite crear enlaces duros para directorios: rompen la jerarquía, crean bucles y crea ambigüedad en dot-dot.
- Enlaces simbólicos (symbolic links): es un fichero cuyos datos contienen la ruta al fichero enlazado $\rightarrow$ pueden *romperse*.

Unix: permisos

Se establecen esos permisos para:

- dueño.
- grupo.
- resto de usuarios.

```
rwx rwx rwx
```

Tipo de acceso:

- **r**: permiso de lectura. En directorio: se pueden leer las entradas de directorio.
- **w**: permiso de escritura. En directorio: se pueden escribir las entradas del directorio (borrar, renombrar, añadir ficheros).
- **x**: permiso de ejecución. En directorios: se puede entrar o atravesar el directorio cuando se evalúa una ruta. Es necesario para acceder a un fichero del directorio (datos y metadatos).

Unix: permisos

- Los permisos se representan normalmente en octal:
  P. ej: 0664 es 110 para el dueño, 110 para el grupo, 100 para el resto.
- Hay otros permisos
  - **sticky bit** (+t): es para directorios: no puedes borrar una entrada si no eres el dueño del directorio, del fichero/directorio que representa la entra, o root.
  - **setuid/setgid bit** (+s): el proceso que ejecute el fichero adoptará el UID/GID del dueño/grupo del fichero.
- Podemos ver los permisos con el comando `ls -l`. Si los queremos ver del directorio y no de las entradas: `-d`.

Unix: permisos

- El comando `chmod` cambia los permisos de un fichero. Solo lo puede hacer el dueño del fichero y root[1].
- Inicialmente, el creador de un fichero es su dueño y grupo.
- El comando `chown` cambia el dueño de un fichero. Hay que tener privilegios especiales para hacer esto.
- El comando `chgrp` cambia el grupo de un fichero. El dueño puede cambiarlo a un grupo al que él pertenezca.

[1] En general, pero esto depende del sistema.

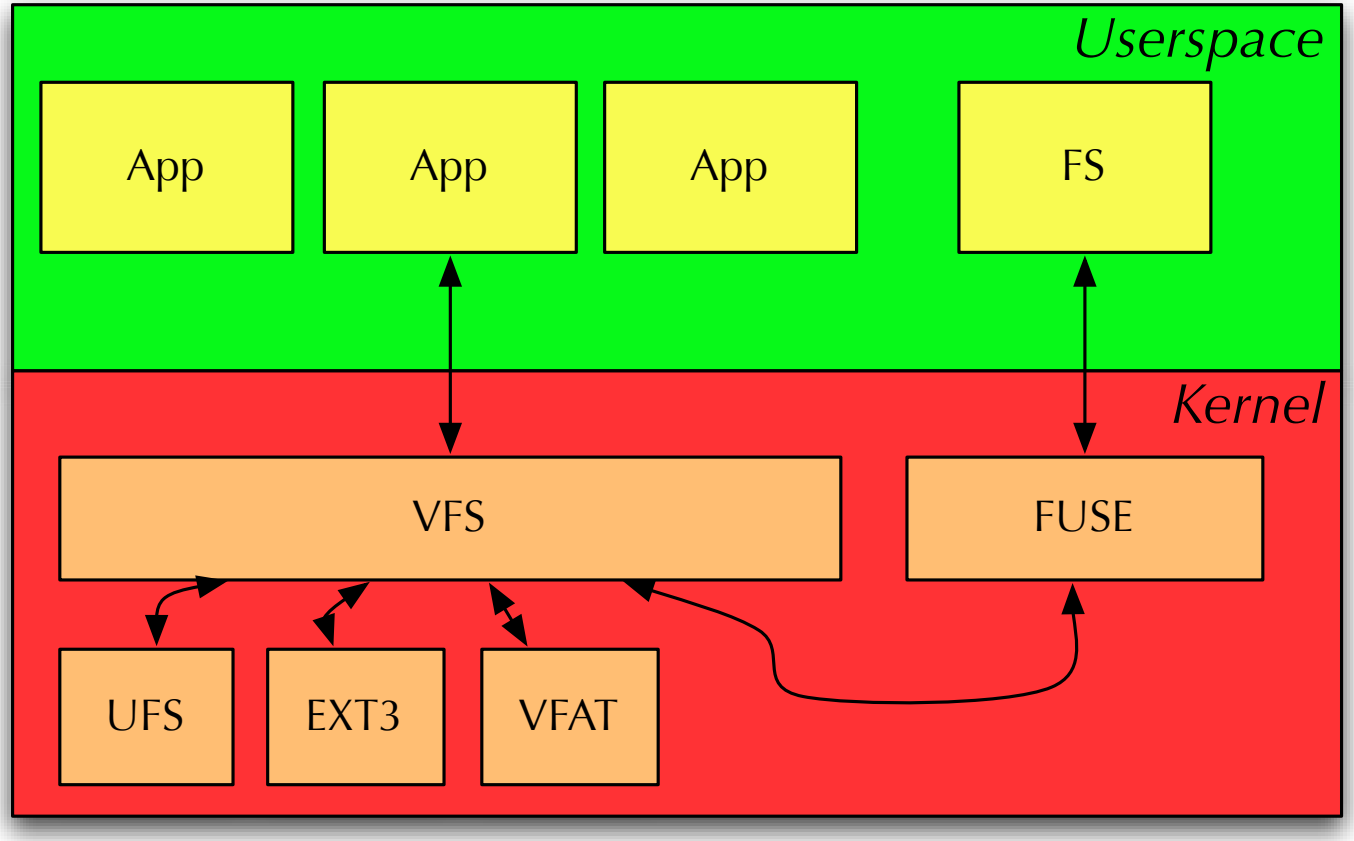
Fallos

- Una pérdida de coherencia en los metadatos (estructuras del sistema de ficheros) es peor que perder los datos.
- El comando `fsck` sirve para reparar el sistema de ficheros después de un fallo.
- Hay que recorrer los bloques de los ficheros para detectar errores, p. ej.:
  - Un bloque está asignado a un fichero y en la lista de bloques libres a la vez.
  - Un bloque en dos ficheros a la vez.
- Esto es caro.
- Los sistemas de ficheros modernos usan técnicas para no acabar con metadatos incoherentes: **journaling**.

Sistemas de Ficheros: implementación

- VFS (Virtual Filesystem Switch): interfaz común con el kernel para todos los sistemas de ficheros.
- Lo normal es que el sistema de ficheros esté implementado como un módulo del kernel.
- Pero también puede estar implementado como un programa de espacio de usuario.
- FUSE: módulo del kernel para implementar sistemas de ficheros en espacio de usuario que se puedan integrar con VFS.

Sistemas de Ficheros: implementación



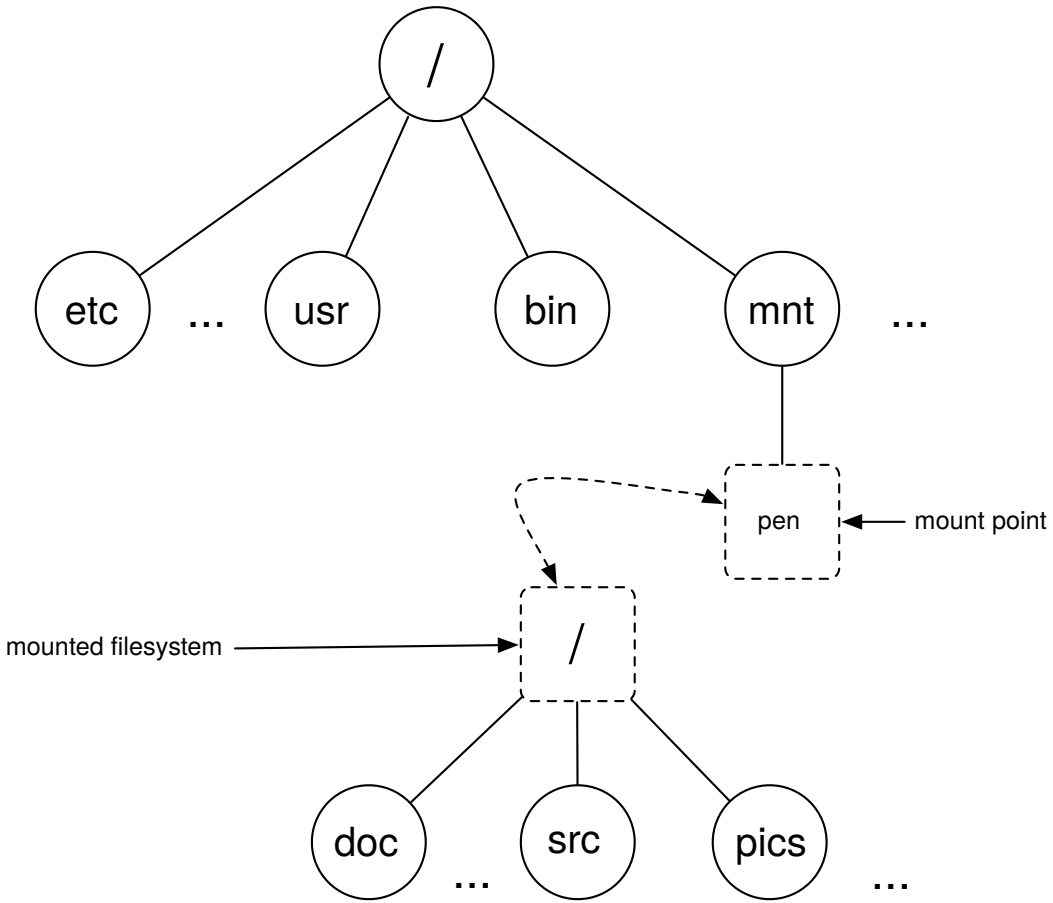
Espacio de nombres

- El árbol de ficheros del sistema.
- Se "camina" (walk) por la ruta, resolviendo cada parte.
- Se pueden *montar* nuevos árboles al espacio de nombres.
- El comando `mount` nos permite ver y modificar el espacio de nombres.
- Unix: un espacio de nombres común para todos los procesos del el sistema.
- Linux y otros sistemas modernos dejan tener diferentes espacios de nombres a grupos de procesos.

Espacio de nombres

Ejemplo:

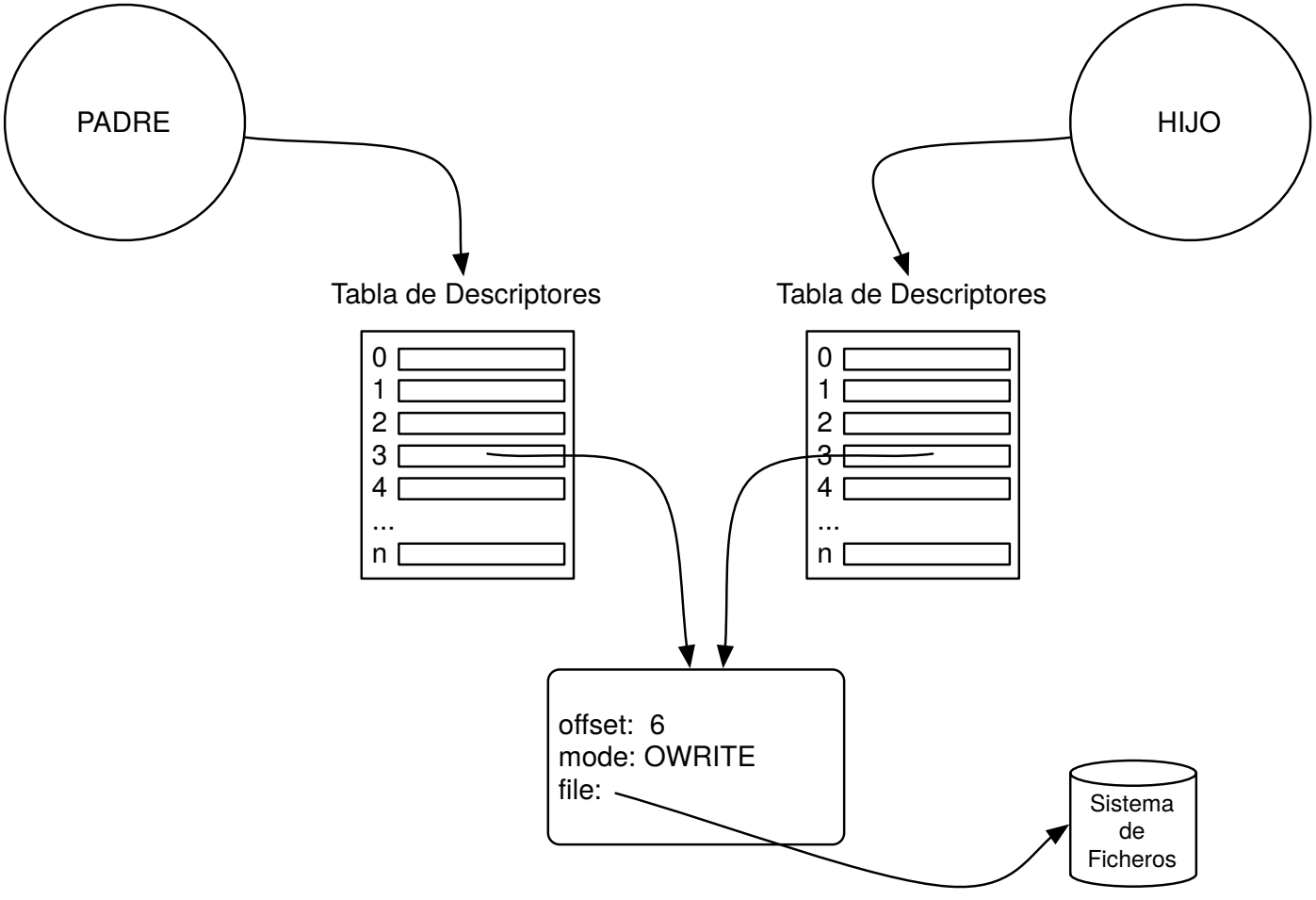
```
sshfs pepe@alpha.aulas.gsyc.urjc.es:/ /mnt/labs
```



Procesos y ficheros

- Por cada proceso, el kernel mantiene una tabla con los ficheros que tiene abiertos: **tabla de descriptores de fichero**.
- Un fichero abierto tiene un **descriptor de fichero (file descriptor)** que lo identifica.
- El número de descriptor de fichero es el índice en la tabla.
- Un fichero abierto tiene offset, modo de apertura, ...
- Los hijos heredan una copia de la tabla del padre.
- Hay tres posiciones especiales en la tabla: (0) entrada estándar, (1) salida estándar, y (2) salida de errores.

Descriptores de fichero



Terminal

- Los procesos que creamos en el shell normalmente tienen como entrada, salida y salida de errores al **terminal**.
- No siempre es así, y se puede cambiar.
- El terminal es un fichero como cualquier otro.
- Cuando se lee, se lee del teclado.
- Cuando se escribe, se escribe en la pantalla.
- `/dev/tty*` son los terminales, `/dev/pts/*` son pseudotermianles (terminales emulados).
- Para un proceso, su terminal siempre se llama:

  `/dev/tty`

Read

- `read`: lee como mucho el número de bytes indicado del fichero abierto en ese descriptor. Puede leer menos bytes sin ser un error: **lectura corta**. Si devuelve 0 bytes, se ha llegado al final del fichero.

```
ssize_t read(int fd, void *buf, size_t count);
```

Write

- `write`: escribe el número de bytes del buffer indicado en el fichero abierto en ese descriptor. Si devuelve un número distinto al número solicitado, se debe considerar un error.

```
ssize_t write(int fd, const void *buf, size_t count);
```

Pread/Pwrite

- Hay versiones de read y write a las que se le da el offset:

```
ssize_t pread(int d, void *buf,
              size_t nbyte, off_t offset);

ssize_t pwrite(int fildes, const void *buf,
              size_t nbyte, off_t offset);
```

Lseek

- `lseek`: cambia el offset del descriptor de fichero. Retorna el offset resultante, -1 en error. Tiene tres modos de operación (tercer parámetro):
  - SEEK_SET: el offset es absoluto.
  - SEEK_CUR: se suma al offset actual.
  - SEEK_END: se suma al offset del final del fichero.

```
off_t lseek(int fd, off_t offset, int whence);
```

Open

- `open`: abre un fichero en el modo indicado por el segundo parámetro. Se usará el primer descriptor de fichero libre.
  Retorna el número del descriptor, -1 en error.
- El segundo parámetro es una combinación de flags:
  - O_CREAT: el fichero se quiere crear si no existe. En este caso, hay que proporcionar el tercer parámetro.
  - O_RDONLY: se va a leer.
  - O_WRONLY: se va a escribir.
  - O_RDWR: se va a leer y escribir.
  - O_TRUNC: si el fichero existe y se abre para escribir (O_RDWR, O_WRONLY), entonces se trunca a longitud 0.
  - O_APPEND: se va a escribir siempre al final del fichero (lseek+write).
  - O_CLOEXEC: se debe cerrar automáticamente si se llama a `exec`.
  - ...
- El tercer parámetro son permisos en caso de creación (O_CREAT).

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Open

- Los permisos se comprueban en la apertura.
- El sistema mantiene el **offset** (posición actual) de un fichero que está abierto. El offset se actualiza cada vez que se lee o se escribe en él (empieza a 0).
- Si el proceso abre dos veces el mismo fichero, son dos ficheros abiertos (cada uno con su offset, modo de apertura, etc.).

Open

- Si se crea el fichero, el dueño del fichero será el UID del proceso. Dependiendo del sistema, el grupo puede ser el GID del proceso o el grupo del directorio en el que se crea [2].
- En creación, se abrirá el fichero en el modo indicado, independientemente de los permisos que se le asignen.
- Los permisos efectivos dependen de la máscara...

[2] En linux, depende de si el directorio en el que se crea tiene a 1 el bit `set-group-ID`.

Umask

- `umask` es una propiedad del proceso (como su PID o el directorio de trabajo actual). Se hereda del padre.
- Los bits a 1 en esa máscara no se pondrán en la creación (aunque los indiquemos). En realidad se ponen estos (siendo `mode` el modo que hemos pasado):

```
(mode & ~umask)
```

- P. ej. si `umask` es 077, aunque pongas los permisos al crear a 755, los permisos finales son 700.

Umask

- Podemos poner el `umask` del shell con el comando `umask`.
- `umask`: llamada al sistema que cambia la máscara de creación y devuelve la antigua. Nunca falla.

```
mode_t umask(mode_t mask);
```

Dup

- `dup`: duplica un descriptor de fichero en la primera posición disponible en la tabla. Retorna el número del descriptor en el que se ha duplicado, -1 en error.
- `dup2`: duplica el descriptor en la posición indicada en el segundo parámetro, cerrándolo antes si estaba abierto.

```
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

Close

- `close`: cierra un fichero abierto y libera los recursos. Retorna -1 si hay un error. Hay que cerrar los ficheros que ya no vayamos a usar. Las tres primeras posiciones en la tabla de descriptores no se deben quedar cerradas: el proceso siempre debe tener entrada, salida y salida de errores.

```
int close(int fd);
```

Access

- `access`: devuelve cero si el fichero puede accederse en el modo indicado, -1 en otro caso.
- Bits para `mode`:
  - F_OK: si existe.
  - R_OK: si se puede leer.
  - W_OK: si se puede escribir.
  - X_OK: si se puede ejecutar.

```
int access(const char *pathname, int mode);
```

Unlink

- `unlink`: elimina un nombre (enlace duro) de un fichero. Si es el último que le queda a un fichero, y ningún proceso lo tiene abierto, el fichero se elimina y se liberan sus recursos.

```
int unlink(const char *pathname);
```

Stat

- `stat`: lee los metadatos de un fichero, los deja en la estructura que se pasa por referencia. Si falla retorna -1.
- Cuidado: atraviesa los enlaces simbólicos. Si queremos los metadatos del enlace y no del fichero apuntado, hay que usar `lstat`.

```
int stat(const char *pathname, struct stat *statbuf);
```

Stat

Campos de `struct stat`:

- st_ino: i-nodo del fichero.
- st_mode: entero con el modo, que son permisos, etc.
- st_nlinks: número de nombres (enlaces duros).
- st_uid: dueño.
- st_gid: grupo.
- st_size: tamaño del fichero.
- st_atime: hora del último acceso al fichero.
- st_mtime: hora de la última modificacón de sus datos.
- st_ctime: hora del último cambio en sus datos o metadatos (excepto en el tiempo de acceso).
- ...

```
man 7 inode
```

Stat

Aplicando la máscara `S_IFMT` al campo `st_mode` podemos saber el tipo de fichero, comparando con estas constantes:

- S_IFDIR: es un directorio.
- S_IFREG: es un fichero normal.
- S_IFLNK: es un enlace simbólico.
- ...

Hay otras máscaras para acceder a los permisos, etc.
`man 7 inode`

Ejemplo:

```
if(st.st_mode & S_IFMT == S_IFREG)
     printf("it's a regular file!\n");
```

Mkdir

- `mkdir`: crea un directorio. Igual que en el caso anterior, los permisos están sujetos al valor de la máscara `umask`.

```
int mkdir(const char *path, mode_t mode);
```

Opendir

- `opendir`: abre un directorio para lectura. El directorio abierto se representa con el tipo `DIR`. En caso de error devuelve NULL.

```
DIR *opendir(const char *dirname);
```

Readdir

- `readdir`: retorna un puntero a la siguiente entrada de directorio, representada por la estructura `dirent`. Retorna NULL en caso de que no haya más o en error.
- En Linux, la estructura `dirent` tiene estos campos de interés:
  - d_ino: i-nodo.
  - d_name: string con el nombre del fichero.
  - d_type: tipo de entrada de directorio.

```
struct dirent *readdir(DIR *dirp);
```

Closedir

- `closedir`: cierra el directorio. Si no se cierra, tenemos un *leak* de memoria. En caso de error retorna -1.

```
int closedir(DIR *dirp);
```

Otras llamadas al sistema para ficheros

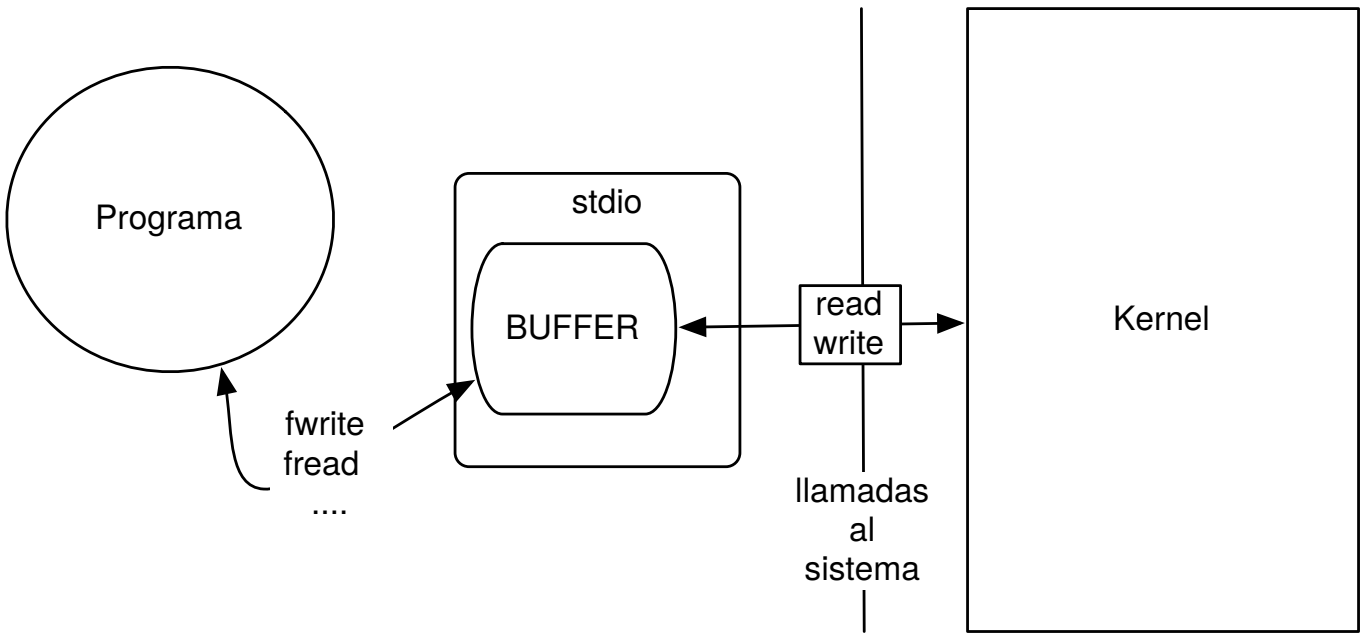
Para modificar los metadatos del fichero:

- `rename`: renombra un fichero.
- `chmod`: cambia los permisos.
- `chown`: cambia el dueño y grupo.
- `utime`: cambia la fecha de acceso y modificación.

Ver páginas de manual (sección 2).

E/S con buffering

- A veces no es fácil leer de un fichero: p. ej. leer líneas de un fichero.
- Realizar una llamada al sistema sale caro: p. ej. leer carácter a carácter.
- Las operaciones de `stdio` ofrecen *buffering*.



E/S con buffering

- Stdio usa *streams*, representados por la estructura `FILE`, para representar un fichero abierto.
- Aunque el estándar de C define distintos tipos de streams (binarios, de texto, etc.). En los sistemas de tipo Unix, no existe diferencia, ya que no hay tipos de ficheros.
- Nos ofrece streams para la entrada estándar (`stdin`), salida (`stdout`) y salida de errores (`stderr`).

E/S con buffering

En stdio se pueden tener un stream:

- Sin buffer (como `stderr`).
- De bloque: hasta que no se completa un bloque no se escriben los datos al fichero o se retorna de la lectura. A esto se le llama *fully buffered*. Ese el modo que se suele usar para ficheros convencionales.
- De línea: cuando se acaba una línea se escribe en el fichero o se retorna la de la lectura. Así es como se configura `stdin` cuando es un terminal.

El tipo se puede cambiar con la función *setvbuf(3)*, y el tamaño de buffer con *setbuffer(3)*.

Fopen

- `fopen`: abre un fichero para hacer E/S con buffering. El modo de apertura es una string, por ejemplo “r” para lectura, “rw” para lectura-escritura, etc.
- `fdopen`: configura un stream a partir de un fichero abierto (de su fd).

```
FILE * fopen(const char *restrict path,
                    const char *restrict mode);
FILE * fdopen(int fildes, const char *mode);
```

Fclose

- `fclose`: vacía el stream, lo cierra y cierra el descriptor de fichero subyacente.

```
int fclose(FILE *stream);
```

Fread

- `fread`: lee del fichero un número de elementos (`nitems`) de tamaño `size`, los guarda a partir de la dirección `ptr`. Retorna el número de elementos leídos. Retorna menos elementos leídos (o cero) que los que se querían leer cuando llega a fin de fichero, pero también cuando tiene un error. Para diferenciar entre esos dos casos, hay que usar las funciones `feof(3)` y/o `ferror(3)`. Tanto esas funciones como `fread` **no actualizan** la variable `errno`.

```
size_t fread(void *restrict ptr, size_t size,
             size_t nitems, FILE *restrict stream);
```

Fwrite

- `fwrite`: escribe en el fichero un número de elementos (`nitems`) de tamaño `size`, de la dirección `ptr`. Retorna el número de elementos escritos.

```
size_t fwrite(void *restrict ptr, size_t size,
              size_t nitems, FILE *restrict stream);
```

Fflush

- `fflush`: fuerza el vaciado del buffer y su escritura en el fichero.

```
int fflush(FILE *stream);
```

Fgets

- `fgets`: lee una línea de hasta `size` bytes. y la guarda en la string `str`. Siempre deja una string. Si no hay más líneas o hay error, retorna NULL.

```
char * fgets(char * restrict str, int size,
                    FILE * restrict stream);
```

Fprintf

- `fprintf`: igual que `printf`, pero escribe en el stream indicado (`printf` siempre lo hace en `stdout`).

```
int fprintf(FILE *stream, const char *format, ...);
```

Redirecciones en el Shell

- > y < son caracteres especiales para el shell.
- > para redirigir la salida estándar a otro fichero.
- < para redirigir la entrada estándar a otro fichero.
- P. ej.

```
$> ps > procesos.txt
$> wc -l < procesos.txt
$> echo hola que tal > hola.txt
$> cat < /NOTICE
```

Redirecciones en Shell

- La mayoría de los comandos que aceptan ficheros como argumentos también leen de su entrada estándar cuando se invocan sin argumentos:

```
$> cat < procesos.txt
$> cat procesos.txt # no es lo mismo!!
```

/dev/null

- `/dev/null` es “ninguna parte”.
  `$> echo hola > /dev/null`
- En algunas shells (p. ej. rc), los comandos que ejecutan en *background* (&) tienen su entrada estándar redirigida a `/dev/null`. P. ej:
  `$> cat &`

/dev/zero

- Fuente infinita de ceros.
- Útil para crear ficheros usando `dd`
- `dd if=/dev/zero of=/tmp/xxx count=1K bs=1`
- O mejor todavía con `tr` para cambiar los ceros por otras cosas.

Salida de errores

- Existe para que no se mezclen las salidas. P. ej.

```
$> ls /kokoko > /tmp/afile
ls: /kokoko does not exist
$> cat /tmp/afile # no hay nada!
```

Más redirecciones

- `>>`
  Redirige la salida sin truncar, añade al final del fichero.
- *número*`>`
  Redirige la posición especificada en la tabla de descriptores. P. ej.:
  `$> ls * 2> /tmp/errores`
- *número1*`>&`*número2*
  Duplica el descriptor número2 en el número1. Atención: se aplican de izquierda a derecha. P. ej.
  `$> ping 172.26.0.1 > /dev/null 2>&1`

Redirecciones en el Shell

- Se pueden conjugar:
  ```
  $> cat < /tmp/afile > /tmp/copia-afile
  $> wc -l /tmp/afile > /tmp/cuentas 2> /dev/null
  ```
- ```
  # en un script
  $> echo 'error!!!' 1>&2
  ```
- ```
  $> cat < afile > afile
  ```
  Error: afile se trunca.
  Solución: Usa un fichero temporal.

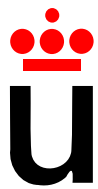
Gestión de memoria

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

Asignación dinámica de memoria

- Problema: tengo un área de memoria para repartir, me van reservando y liberando trozos de diferentes tamaños de forma dinámica.
- Una vez comprometida una reserva, no se puede mover.
- Gestión implícita: un *recolector de basura* (Garbage Collector) se encarga de liberar memoria no referenciada.
- Gestión explícita: lo haces manualmente. P. ej. conocemos:
  `malloc:` reserva memoria dinámica.
  `free:` libera memoria dinámica.

Asignación dinámica de memoria

- Esto se necesita implementar tanto en área de usuario como en el kernel.
- P. ej. el kernel de Linux tiene una función llamada `kmalloc`, que se apoya en un asignador que se llama *slab allocator*.
- En área de usuario: `malloc` es una función de la libc que reparte la memoria del *heap*. El heap puede crecer en demanda (con la llamada al sistema `brk`).
- Problemas de fragmentación: ya los conocemos, los hemos visto en sistemas de ficheros.

Problema: fragmentación externa

Fragmentación externa: vamos 4 personas al cine y quedan 12 butacas libres, pero no hay 4 contiguas.

- Después de reservar/liberar, quedan huecos inservibles.
- La suma de los fragmentos sí sería útil.
- Ley 50 %: dados N bloques, 0.5 * N se pierden por la fragmentación externa (en media).
- La compactación solucionaría el problema... ¿se puede si hablamos de memoria dinámica?

Problema: fragmentación interna

- Posible solución a la fragmentación externa: reservar en base a bloques fijos $\rightarrow$ un hueco libre siempre puede ser útil.
- Además, ahora no malgastamos recursos para apuntar huecos inservibles.
- Problema: dentro de la memoria reservada sobra espacio, y la suma del espacio sobrante en todas las reservas sería útil...

Fragmentación interna: vamos 2 personas a cenar, y nos dan una mesa de 4 comensales.

Estrategias

- Para minimizar fragmentación externa: tamaño mínimo, agrupación de reservas relacionadas (tiempo y tamaño), etc.
- Para minimizar fragmentación interna: buscar el mejor ajuste, etc.
- Para minimizar tiempo en las reservas: caches, segregación por tamaño, etc.

Políticas de asignación dinámica de memoria

- **First Fit:** el primero en el que cabe empezando por el principio. Se suele comportar bien. Es rápido y simple.
- **Next Fit:** el primero en el que cabe empezando por donde te quedaste. Empíricamente se comporta peor que First-Fit (más fragmentación).
- **Best Fit:** el que se ajuste mejor. Ayuda a tener los fragmentos pequeños. Más lento. Tiende a dejar huecos muy pequeños o muy grandes.
- **Worst Fit:** el que se ajuste peor. Peor resultado que los anteriores.
- **Quick Fit:** se mantienen listas de de trozos de tamaños populares para reservas rápidas. Segregación: tener distintas listas para trozos de distintos tamaños. Acelera la búsqueda, se comporta bien. Contra: más complejo, más estructuras.
- Hay otras: buddy system, bump allocator, etc.

Ejemplo real de implementación de malloc:

- Para peticiones grandes $>=$ 512 bytes, un asignador Best Fit puro.
- Para peticiones pequeñas $<=$ 64 bytes, un asignador Quick Fit con trozos de ese tamaño.
- Para peticiones intermedias usa una política mezcla de las anteriores.
- Para peticiones muy grandes $>=$ 128 Kb, no irá en el *heap*: se crea una región nueva de memoria[1].

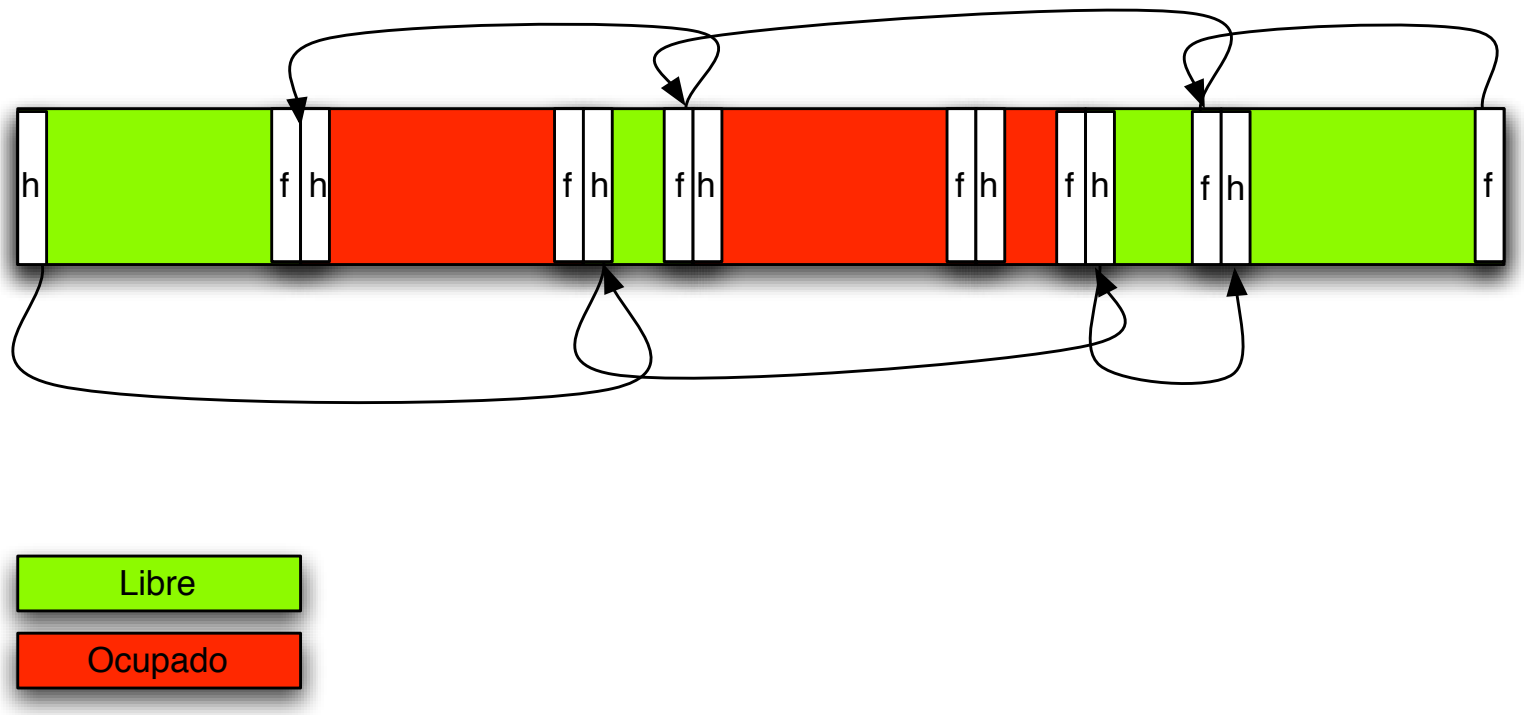
[1] mmap anónimo.

Mecanismos para asignación dinámica de memoria

- Implementación: lista enlazada de trozos libres, de trozos ocupados, de ambos, circular, doblemente enlazada, varias listas...
- Coalescing: fundir dos trozos libres contiguos en uno, ¿cuándo lo hago?
- Headers y Footers: para moverse rápido entre nodos adyacentes. Acelera el fundido.
- Envenenamiento: valores que indican zonas liberadas para detectar bugs.

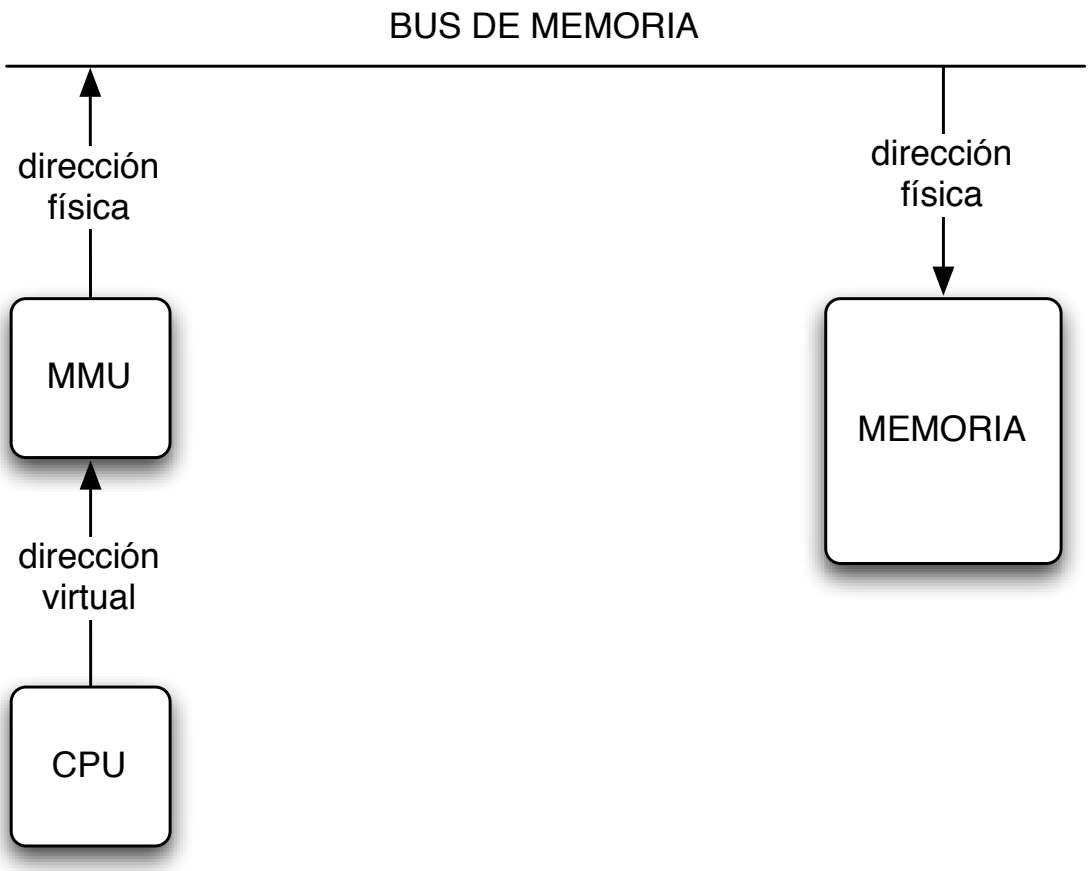
Ejemplo: doble lista de libres, con cabeceras y pies

¿Qué hay en el header (cabecera) y footer (pie)? el estado (libre, ocupado), el tamaño del trozo y (dependiendo del tipo) el puntero al siguiente/anterior libre.



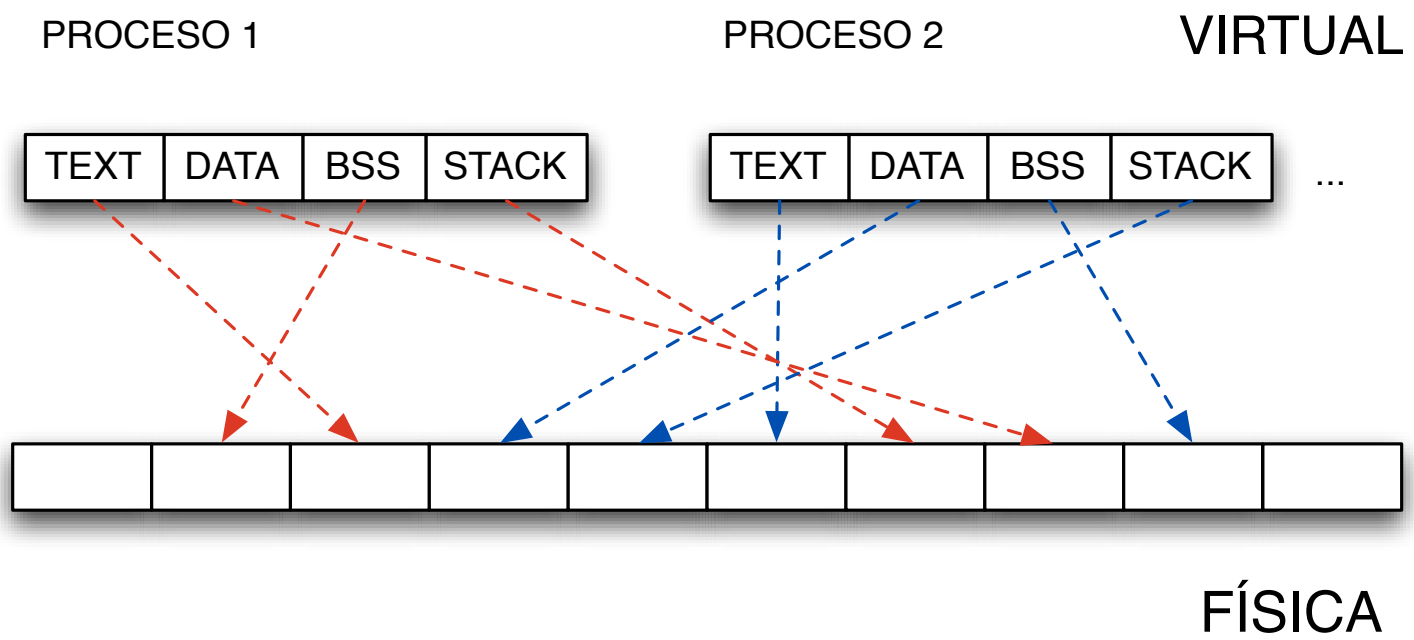
Memoria virtual

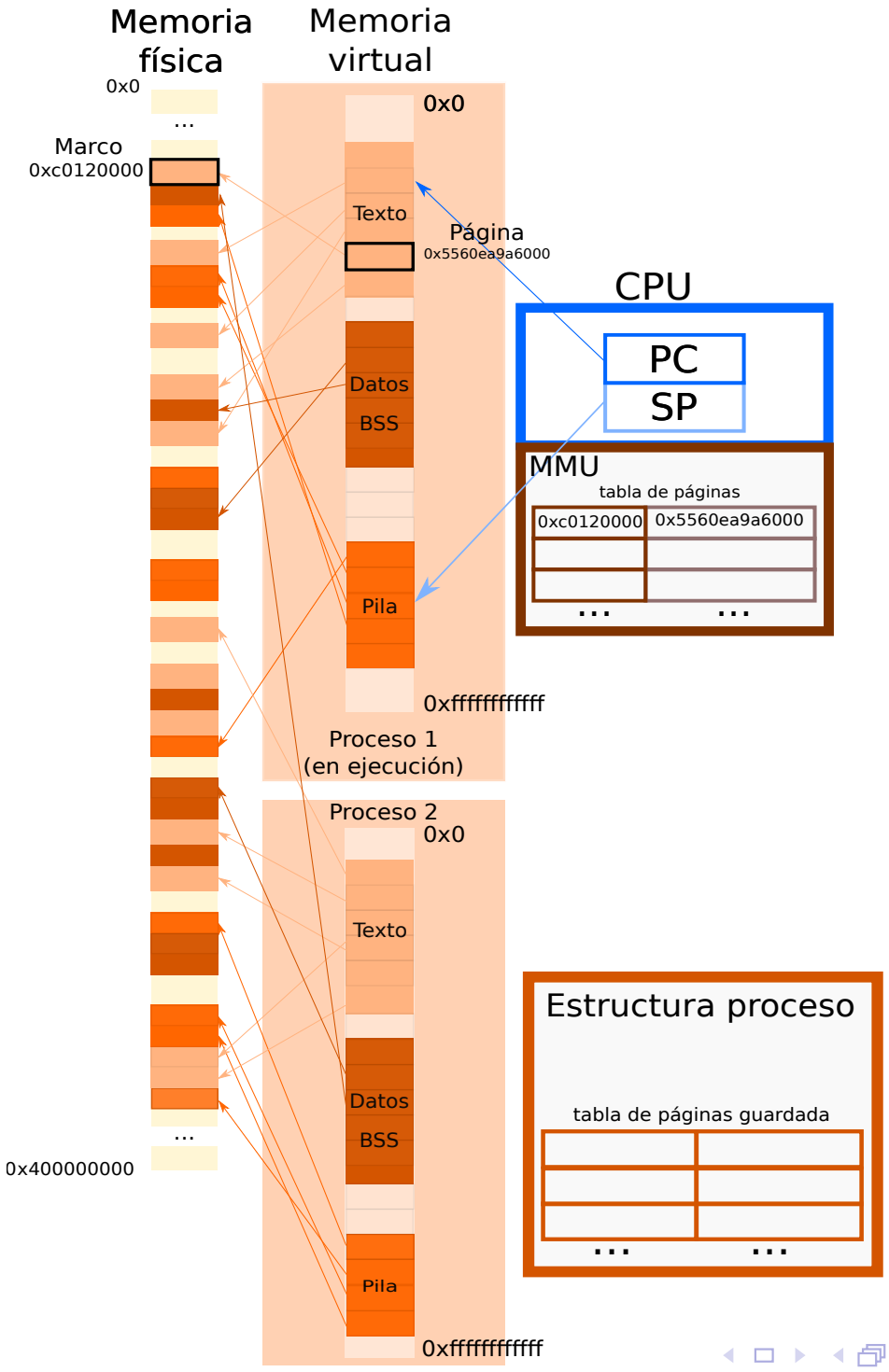
¡La CPU nunca ve direcciones físicas! La MMU se encarga de todo:



Memoria virtual

El kernel mantiene la correspondencia entre direcciones físicas y lógicas para cada proceso. Cuando se cambia el contexto a un proceso, se instala su mapa (tabla de páginas) en el hardware (simplificado, ver la siguiente transparencia).




Memoria física
Memoria virtual
0x0
...
Marco
0xc0120000
0x0
Texto
Página
0x5560ea9a6000
CPU
PC
SP
Datos
BSS
MMU
tabla de páginas
0xc0120000
0x5560ea9a6000
Pila
0xffffffffffff
Proceso 1
(en ejecución)
Proceso 2
0x0
Texto
Datos
BSS
Pila
0xffffffffffff
0x400000000
Estructura proceso
tabla de páginas guardada

Memoria virtual

- **Protección**: un proceso no puede acceder a la memoria de otro proceso ni a la del kernel (errores, ataques).
- **Simplicidad**: para el proceso, toda la memoria es contigua.
- **Abstracción**: un proceso cree que está en su propia máquina y que toda la memoria es suya.
- **Depuración**: la organización de la memoria es similar para todos los procesos y la misma para distintas ejecuciones de un programa.
- **Vinculación**: permite que sea en tiempo de ejecución (resolución de símbolos y relocalizaciones).
- **Reutilización/compartición**: una zona de memoria física se puede mapear en distintas zonas de la memoria virtual de distintos procesos, evitando copias.
- **Intercambio (swapping)**: se puede usar almacenamiento secundario para almacenar la memoria de un proceso si no tenemos suficiente memoria física. **Es un error pensar en que esta es la única utilidad de la memoria virtual**.

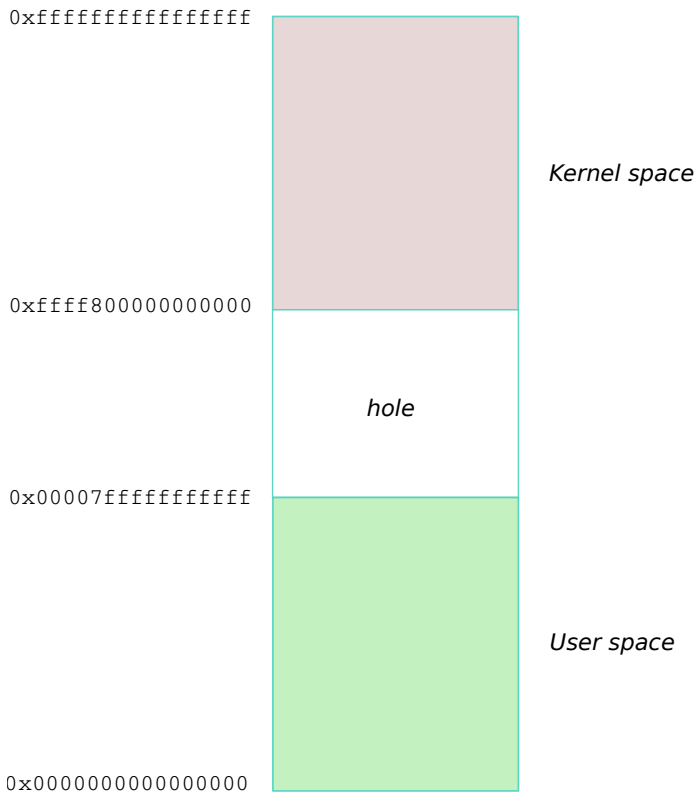
Memoria virtual

- Cuando se necesita tratar con direcciones *físicas*, el kernel puede instalar un *mapa identidad* (p. ej. para la inicialización del sistema).
- El kernel necesita comunicarse con el hardware a través de áreas mapeadas en memoria (p. ej. PCI):

```
$>  cat /proc/iomem
...
000a0000-000bffff : PCI Bus 0000:00
 000a0000-000bffff : Reserved
000c0000-000c3fff : PCI Bus 0000:00
...
```

Memoria virtual

- El espacio de direcciones virtuales se suele dividir para direcciones del kernel y direcciones de usuario. En Linux x86_64[2]:



[2] Direcciones de 48 bits.

Intercambio

- Consiste en mover la memoria (parte o todo) de algunos procesos de RAM a un dispositivo de almacenamiento (disco): swap.
- Cuando toca ejecutar, se trae de vuelta y se lleva la de otro proceso.
- Grano: toda la memoria del proceso, segmentos, **páginas** ...
- Es muy caro:
  I/O intercambio a disco + I/O intercambio desde el disco
  - Serial ATA (SATA-150) 1,200 Mbit/s
  - DDR3-SDRAM 136.4 Gbit/s
- Actualmente, no es tan útil.

MLock

- `mlock`: llamada que permite que las páginas correspondientes al rango dado nunca vayan a swap. Únicamente se permite hacer esto a procesos privilegiados.
- `mlockall`: lo hace con toda la memoria del proceso.

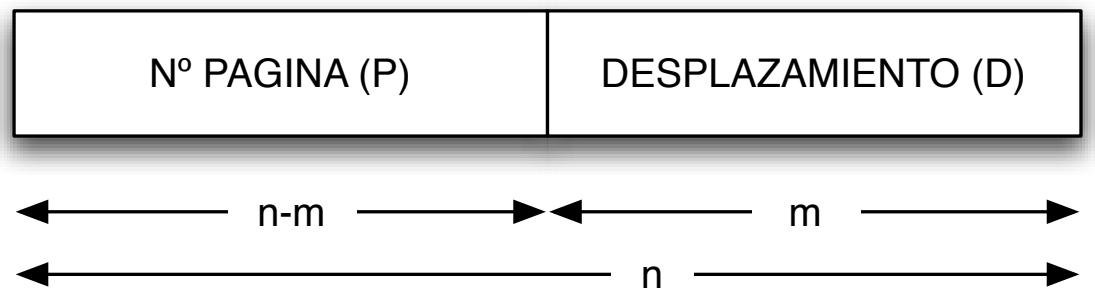
```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

Paginación

- Objetivo: poder tener la memoria de un proceso distribuida en zonas no contiguas de la memoria física.
- La memoria física se divide en **marcos**.
- La memoria lógica se divide en **páginas**.
- El hardware determina el tamaño de página.
- El medio de almacenamiento para el intercambio también se divide en porciones del tamaño de un marco.

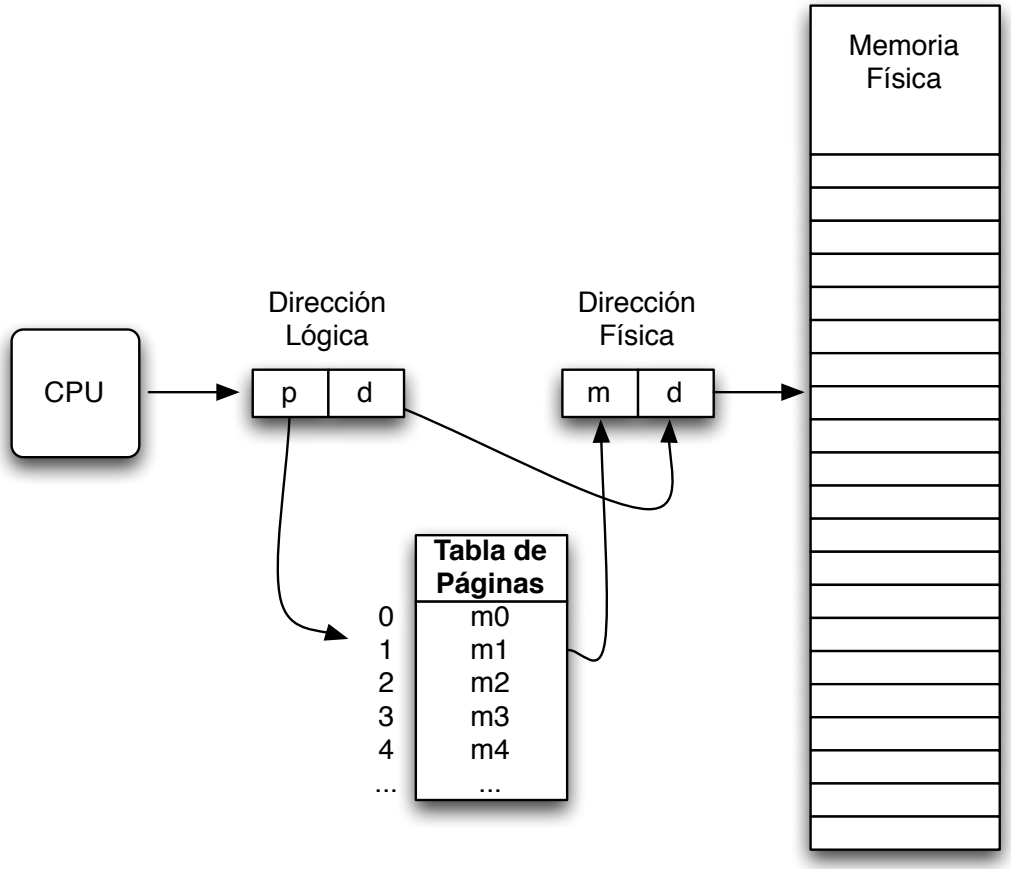
Paginación

- Tabla de páginas: dirección base para cada página. Cada proceso tiene su tabla de páginas.
- El sistema lleva la cuenta de los marcos de página.
- Si una dirección no tiene traducción: fallo de página.
- Dirección virtual:



2^m = tamaño de página

Paginación


Memoria
Física
CPU
Dirección
Lógica
p
d
Dirección
Física
m
d
Tabla de
Páginas
0
1
2
3
4
...
m0
m1
m2
m3
m4
...

Paginación

- De nuevo tenemos problemas de fragmentación interna.
- Si el tamaño de página es pequeño...
  - hay menos fragmentación.
  - necesitamos tablas de página con muchas entradas $\rightarrow$ búsqueda lenta.
- Si el tamaño de página es grande...
  - ganamos tiempo de I/O a la hora de traer páginas de almacenamiento.
- Hay que llegar a un compromiso: actualmente son de 4Kb - 8Kb.
- Algunas arquitecturas tienen superpáginas: hasta 2Gb.

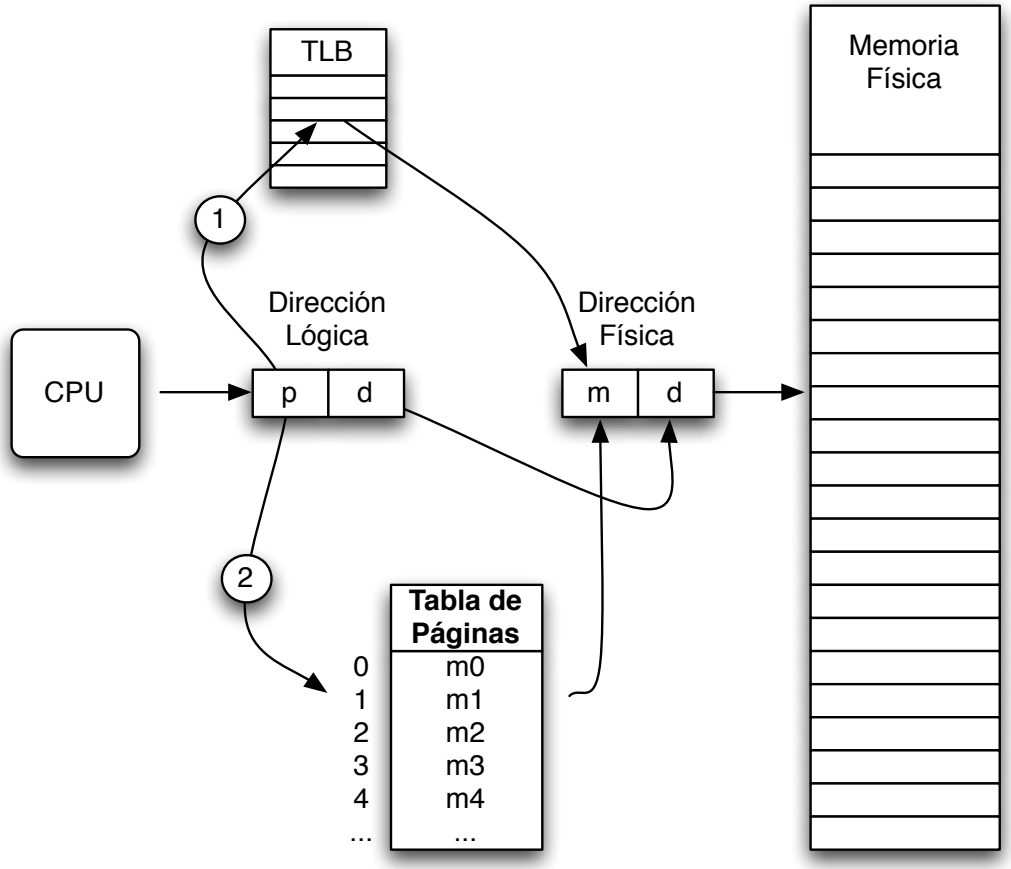
Paginación

- Cada proceso tiene su tabla de páginas, que forma parte de su contexto.
- Las tabla de páginas tienen que estar en memoria... y es grande.
- Se usan registros especiales de la CPU para apuntar a la tabla (los detalles dependen de la arquitectura).
- **Problema**: dos accesos a memoria principal para cada acceso real:
  1. acceso a la tabla de páginas
  2. acceso a la memoria

Paginación: TLB

- Solución hardware: TLB (*Translation Look-aside Buffer*).
- Es una pequeña memoria cache de la tabla de páginas ($\approx$ cientos de entradas).
- El acceso a TLB es muy rápido:
  1. TLB, SRAM fully-associative: 1 ciclo
  2. Cache L1, SRAM set-associative: 3 ciclos
  3. Cache L2, SRAM: 14 ciclos
  4. Memoria principal, DRAM: 240 ciclos
- Traducción
  1. Si está en la TLB, se traduce directamente.
  2. Si no está, se busca en la tabla de páginas.
- Cuanto mayor tamaño de página, mayor tasa de acierto en TLB.

Paginación: TLB



Paginación: TLB

- Cuando se cambia de contexto, se tiene que limpiar la TLB (*TLB flush*).
- Si se accede a una página que no estaba, se inserta.
- Si la TLB está llena, se debe desalojar una entrada.
- Se pueden bloquear entradas.
- Tasa de aciertos (*Hit Ratio*): porcentaje de veces que la página está en la TLB.
- Tiempo de acceso efectivo:

$$T_{ef} = P_{acierto} * T_{acierto} + P_{fallo} * T_{fallo}$$

e.g.

$$0{,}98 * (20ns + 100ns) + 0{,}02 * (20ns + 100ns + 100ns) = 122ns$$

penalización del 22 % en el acceso vs. 100 % sin TLB.

Page Table Entry (PTE)

Las entradas pueden tener ciertos bits:

- Bit de presente (o bit de válido): bit en cada entrada de la tabla de páginas que indica si la página tiene traducción o no. Si no la tiene, puede ser no válida válida o porque esté en *swap*.
- Bit de modo: permiso para escribirla o no.
- Otros bits: si puede ir a caché, permiso de ejecución, si se ha accedido, ...

Paginación: tabla de páginas

- Problema: espacios de direcciones demasiado grandes (p. ej. 2^{48}).
- La tabla de páginas excesivamente grande como para tener todas sus entradas contiguas.

Paginación multinivel

- Una solución: dividir la tabla de páginas en N niveles.
- Ejemplo: en Intel i7 la tabla es de 4 niveles.
- Dirección lógica para 2 niveles: **(p1, p2, d)**.
  - p1: índice en la tabla exterior.
  - p2: índice en la tabla interior.
  - d: desplazamiento en la página.

Paginación multinivel

- Problema: un acceso puede provocar hasta N+1 accesos reales a memoria.
- En la práctica, depende de la tasa de aciertos de la TLB.
- Ejemplo: con una tasa de aciertos del 98 %, tiempo de acceso a la TLB de 20 ns, y tiempo de acceso a memoria de 100 ns:

$$T_{acceso-efectivo} =$$
$$0{,}98 * (20ns + 100ns) + 0{,}02 * (20ns + 100ns + 100ns + 100ns) = 124ns$$

Paginación: compartiendo páginas

Los procesos pueden compartir marcos de página:

- Ejemplo: el código (*text*) de un binario se puede compartir si es reentrante $\rightarrow$ si no se modifica durante la ejecución, no hace malgastar más marcos para guardar exactamente lo mismo.
- Shared memory: los procesos que comparten su memoria (*data*, *bss*, *heap*) comparten sus marcos de páginas.
- **Copy-on-write**: se puede compartir mientras que no se modifique. Cuando se modifica, se hace una copia y cada uno tiene la suya. Es lo que se hace cuando llamamos a `fork`: padre e hijo comparten las páginas hasta que uno intenta modificar algo.

Paginación en demanda

Consiste en

- Consumir únicamente los marcos de página que se necesitan, no todos los necesarios para las páginas del fichero $\rightarrow$ se ahorra memoria si hay páginas que nunca se llegan a usar.
- Aproximación perezosa: las páginas se van llevando a memoria física según se van necesitando, a medida que avanza la ejecución.

Paginación en demanda

- ¿Cuándo se reserva el marco? El proceso intenta acceder a la dirección de memoria, Si la página no está presente, habrá un fallo de página.
- Contra: tenemos muchos más fallos de página que manejar $\rightarrow$ más lento que reservar todo al principio.
- ¿Se debe cambiar un binario mientras que hay procesos ejecutándolo?

Paginación en demanda

Cuando se ejecuta una instrucción que intenta acceder a una dirección de memoria sin mapeo $\rightarrow$ salta una *trap* por fallo de página $\rightarrow$ el control pasa al kernel y:

1. Se mira si la dirección pertenece al espacio de direcciones del proceso.
2. Si no pertenece, se manda un SIGSEGV al proceso (segmentation fault).
3. Si es correcta, se comprueban los permisos de la página y el tipo de acceso que se está haciendo (lectura, escritura, ejecución).
4. Si son correctos, se busca un marco para esa página.
5. Si es necesario, se copia el contenido del fichero binario al marco de página (p.ej. si es una página de TEXT o DATA).
6. Se modifica la tabla de páginas para poner su bit de presente.
7. Se ejecuta de nuevo la instrucción que ha generado el trap. Ya puede acceder a esa dirección de memoria virtual.

Overcommitment

- Cuando se necesita hacer crecer el heap (llamada al sistema `brk`) no se compromete memoria física hasta que se intenta usar (hasta que se genera un fallo de página). Con overcommitment, la llamada a `malloc` **no** va a fallar.
- Lo mismo pasa con las variables globales sin inicializar (BSS).
- Cuando se accede por primera vez a una dirección de memoria de esa página, se compromete su marco de página.
- ¿Cuándo falla tu programa si el sistema se queda sin memoria (OOM, out of memory)? Si no hay marcos en ese momento... ¡fallo!
- Algunos sistemas lo tienen y permiten configurar su comportamiento. En Linux, se controla con:

  `/proc/sys/vm/overcommit_memory`

Page cache

Los sistemas modernos usan una cache:

- Es una forma de aprovechar la memoria física que no se usa para los procesos.
- El kernel mantiene una cache de páginas para mantener los datos de los ficheros, directorios, dispositivos de bloques (buffers), etc. que se están usando y evitar operaciones de E/S.
- La mayoría de las llamadas `read` y `write` se satisfacen a través de esta cache.

Page cache

- Muy eficiente: en algunos sistemas se ahorra hasta el 85 % de operaciones de E/S.
- En ocasiones es necesario saltarse la cache (p. ej. la opción `O_DIRECT` de `open` en Linux).
- El comando y llamada al sistema `sync` sincroniza las páginas *sucias* de la cache, las baja a disco. Se ejecuta periódicamente.

Mmap

- `mmap`: esta llamada al sistema permite crear una región nueva para el proceso.
- Se llama *mmap anónimo* cuando simplemente queremos una nueva región de memoria inicializada a cero.
- También permite proyectar un fichero en memoria, para acceder al fichero sin usar las llamadas al sistema `read`, `write`, etc.
- **No sustituye a esas llamadas:** no es apto para ficheros sintéticos (p. ej. /proc), ficheros que crecen, ficheros pequeños, sistemas de ficheros en red, etc.

```
void *mmap(void *addr, size_t length, int prot,
                  int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

/proc/meminfo

- En Linux podemos ver el estado de la memoria en

  `/proc/meminfo`

- Campos:
  - MemTotal: memoria física usable (total menos el binario del kernel)
  - MemFree: memoria libre
  - Buffers: bloques de dispositivos en la page cache
  - Cached: la page cache
  - Dirty: memoria sucia que tiene que bajar a disco.
  - ...

Algoritmos de reemplazo

- Problema: cuando usamos respaldo (swap) y nos quedamos sin marcos de página, necesitamos expulsar marcos.
- Hay distintos algoritmos de reemplazo.
- La solución óptima (minimizar fallos) no es viable: consistiría en reemplazar la página que no se va a usar en el periodo de tiempo más largo. $\rightarrow$ ¡hay que saber la secuencia de accesos de antemano! $\rightarrow$ no nos sirve.

Algoritmos de reemplazo: FIFO

- Se reemplaza la página más vieja (la que lleva más tiempo).
- Problema: la antigüedad no tiene que ver con frecuencia de uso.
- Anomalía de Belady: siendo $N > M$, el número de fallos para N marcos puede ser mayor que para M marcos, *(ojo: no es exclusiva de FIFO)*.
- Secuencia: 1,2,3,4,1,2,5,1,2,3,4,5
  Con 4 marcos: 10 fallos.
  Con 3 marcos: 9 fallos.

Algoritmos de reemplazo: LRU

- Idea: reemplazar la página que no ha sido usada desde hace más tiempo (*Least-Recently Used*).
- Posible implementación: apuntar en la tabla de páginas el valor de un contador global cuando se accede a la página. Se reemplaza la página con el contador más bajo.
- Problema: necesita apoyo del hardware para marcar el tiempo porque no es admisible una interrupción por acceso para que el sistema operativo se encargue de esto $\rightarrow$ ninguna máquina lo ofrece.

Algoritmos de reemplazo: NRU

- Se reemplaza una página que no ha sido usada recientemente.
- Se usan dos bits: bit de referencia y bit de sucio.
- Cuatro clases en orden de prioridad ascendente:
  1. no referenciada / limpia: ideal para reemplazar.
  2. no referenciada / sucia: hay que escribirla de vuelta.
  3. referenciada / limpia: puede ser referenciada de nuevo.
  4. referenciada / sucia: la peor opción.
- El bit de referencia se pone a 0 periódicamente.
- Problema: el bit de referencia indica que no se ha usado recientemente, pero no indica orden.

Algoritmos de reemplazo: segunda oportunidad

- FIFO dando una nueva oportunidad a las páginas con el bit de referencia a 1:
  1. se pone el bit de referencia a 0.
  2. se coloca al final de la cola (esto es, se pone el tiempo de llegada al tiempo actual).
- Se puede implementar con un array circular (reloj).
- Degenera en FIFO si todas las páginas están referenciadas (con una vuelta adicional).

Otras tácticas del paginador

- Siempre mantiener un conjunto de marcos libres de reserva.
- Desalojar marcos de páginas de la Page Cache. P. ej. los ficheros binarios que no está ejecutando ningún proceso actualmente.
- En ratos ociosos se pueden escribir las páginas sucias a disco y marcarlas como limpias.
- Se pueden desalojar páginas dejando el contenido en el marco (aunque esté libre, se queda con el contenido). Si después hay que traer la misma página, no hace falta I/O.

Asignación de marcos a los procesos

Hay distintas políticas:

- **De forma equitativa**: a todos lo mismo
- **De forma proporcional** al tamaño del proceso en memoria. Siendo s_i el tamaño en memoria del proceso i,

  $$S = \sum s_i$$

  y M el número de marcos libres, al proceso i le corresponden:

  $$a_i = \frac{s_i M}{S}$$
- Teniendo en cuenta la prioridad del proceso.
- ...

Asignación de marcos a los procesos

- **Asignación local**: se desaloja una página del proceso que causa el fallo de página.
- **Asignación global**: se desaloja una página de cualquier proceso.

Thrashing

- Cuando el sistema gasta más en paginación que en procesamiento útil.
- Causas:
  - aumento del grado de multiprogramación
  - asignación global
- Efecto: los procesos se roban marcos entre ellos.

Conjunto de trabajo

- Solución al *thrashing*: tener marcos suficientes para el **conjunto de trabajo de cada proceso**.
- **Conjunto de trabajo:** conjunto de páginas en las Δ referencias más recientes (*working set window*).
- Se mantiene el número de marcos asignado a un proceso igual al número de páginas de su conjunto de trabajo.
- Se estima si la creación de otro proceso provocará *thrashing* mirando el tamaño del conjunto de trabajo de todos los procesos.

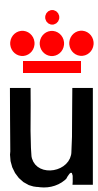
Comunicación entre procesos

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

IPC

- Los procesos necesitan comunicarse para sincronizarse, intercambiar información, etc.
- Existen distintos mecanismos de IPC para los procesos que no comparten memoria.

Pipe

- Estilo “Unix”: pequeños programas (filtros) que leen datos por su entrada estándar, los procesan, y los escriben por su salida estándar.
- Los *pipes* nos permiten concatenar programas conectando la entrada de uno a la salida de otro.

Pipe

- Cada extremo del pipe se comporta como un fichero.
- Todo lo que se escribe en un extremo se lee en el otro.

Llamada al sistema: pipe

- `pipe`: crea un nuevo pipe. Se pasa array contendrá los dos FDs del pipe (uno para cada extremo). En Unix son *simplex*, se escribe en fd[1] y se lee de fd[0].
- Se debe crear antes de llamar a fork() para que ambos procesos lo compartan.

```
int pipe(int fd[2]);
```

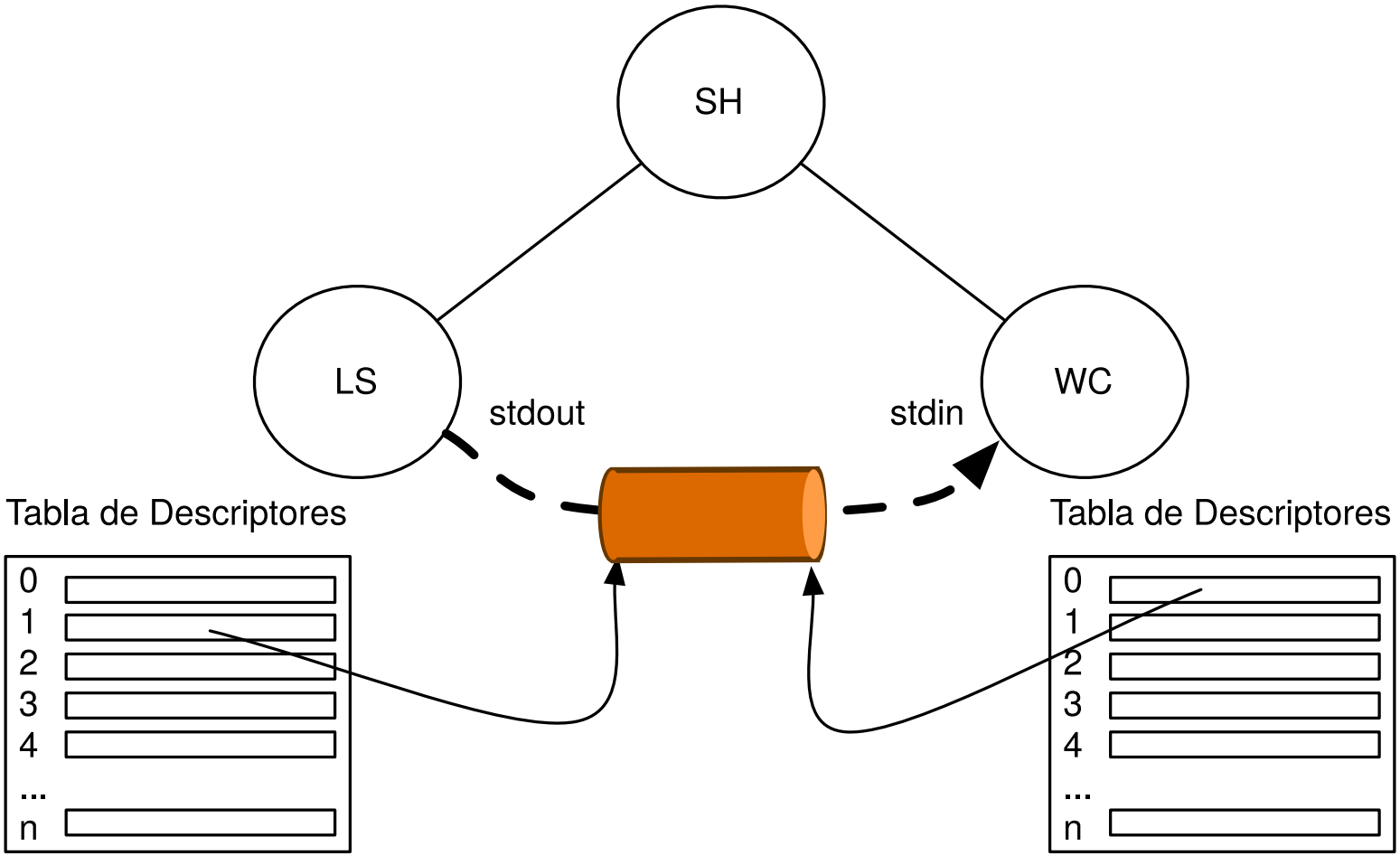
Llamada al sistema: pipe

- Debemos cerrar el extremo que no vamos a usar.
- En general, no se conservan los *límites de escritura*.
- Un pipe tiene un buffer limitado: se puede llenar.
- Leer de un pipe vacío te deja bloqueado.
- Escribir en un pipe lleno te deja bloqueado.
- Leer si no hay nadie al otro lado: retorna 0 bytes.
- Escribir en un pipe sin nadie al otro lado: error.
- Ambos procesos deben leer/escribir en paralelo, no secuencialmente. ¿Qué pasa si se llena el pipe? $\rightarrow$ **interbloqueo**.

En el shell

P. ej.

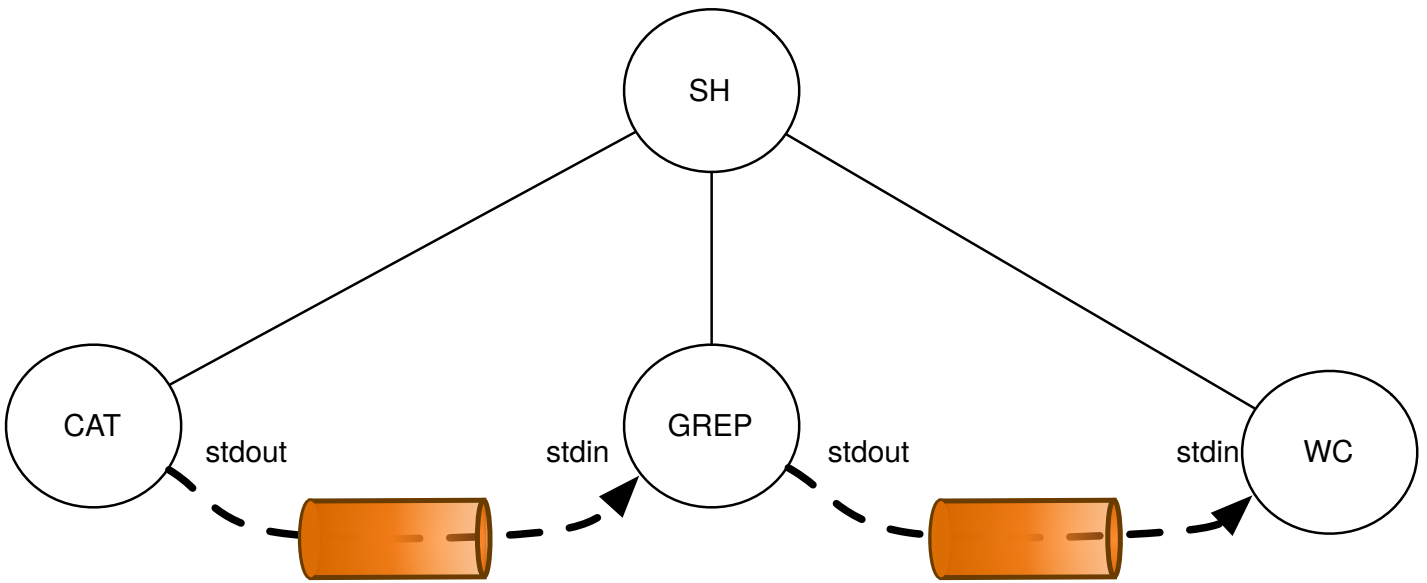
```
$> ls *.txt | wc -l
```



En el shell

P. ej.

```
$> cat *.txt | grep 'pepe' | wc -l
```



Fifo

- Son pipes con nombre: una ruta en el espacio de nombres. Se pueden crear en el shell con el comando `mkfifo`
- En el shell, se eliminan con `rm`. Si se borra, los que lo están usando lo pueden seguir usando (como cualquier fichero en este caso).

Fifo

- Lectura: un `open` de solo-lectura, si no hay ningún proceso con el *fifo* abierto para escribir en él, te deja bloqueado hasta que lo haya.
- Se puede forzar una apertura no bloqueante (`NON-BLOCKING`). Un `open` **no bloqueante** de solo-lectura falla si no hay ningún proceso que tenga abierto el *fifo* para escribir.
- Escritura: un `open` de solo-escritura, si no hay ningún proceso con el *fifo* abierto para leer de él, te deja bloqueado hasta que lo haya.
- Si la apertura es no bloqueante (`NON-BLOCKING`), fallará si el *fifo* no está actualmente abierto para lectura.
- Se puede abrir en modo lectura-escritura (aunque en general esto no tenga mucho sentido).
- Si el *fifo* se borra mientras el proceso está bloqueado en la apertura, se queda bloqueado de por vida.

Fifo

Igual que los pipes:

- Cuando no hay ningún proceso que pueda escribir, se leen 0 bytes.
- Cuando no hay ningún proceso que pueda leer, una escritura falla (se recibe la señal SIGPIPE).

Llamada al sistema: mkfifo

- `mkfifo`: crea un fifo con los permisos indicados. El dueño/grupo es el UID/GID del proceso. Si el fifo ya existe, retorna error (-1).

```
int mkfifo(const char *path, mode_t mode);
```

Señales

- Son un mecanismo para notificar cosas a un proceso, o interrumpir su ejecución.
- Son difíciles de entender y usar.
- Pueden ser:
  - Síncronas: se entregan inmediatamente porque son consecuencia de su ejecución, el proceso está ejecutando. Suelen ser enviadas por el sistema como reacción al algo que ha hecho el proceso (p. ej. SIGSEGV).
  - Asíncronas: la señal se entregará y manejará cuando el proceso sea planificado. Suelen ser enviadas por otros procesos (p. ej. SIGINT, SIGSTOP).

Señales

- Cuando recibe una señal, el proceso puede ignorarla o manejarla (con la acción por omisión o definir el manejador).
- Las señales pueden anidarse.
- Hay distintas acciones por omisión dependiendo del tipo de señal:
  - Terminar el proceso.
  - Provocar un core dump y terminar el proceso.
  - Ignorar la señal.
  - Suspender el proceso.
  - Reanudar su ejecución.

Señales

Las señales más populares son:

- 1 SIGHUP, default action: terminate process, description: terminal line hangup (terminal connection lost)
- 2 SIGINT, default action: terminate process, description: interrupt program (ctrl-c in terminal)
- 3 SIGQUIT, default action: create core image , description: quit program (ctrl-\ in terminal)
- 4 SIGILL, default action: create core image, description: illegal instruction
- 8 SIGFPE, default action: create core image, description: floating-point exception
- 9 SIGKILL, default action: terminate process, description: kill program (cannot be ignored)
- 11 SIGSEGV, default action: create core image, description: segmentation violation
- 13 SIGPIPE, default action: terminate process, description: write on a pipe with no reader
- 14 SIGALRM, default action: terminate process, description: real-time timer expired
- 15 SIGTERM, default action: terminate process, description: software termination signal (e.g. shutdown)
- 17 SIGSTOP, default action: stop process, description: stop (cannot be caught or ignored)
- 18 SIGTSTP, default action: stop process, description: stop signal generated from keyboard
- 19 SIGCONT, default action: discard signal, description: continue after stop
- 20 SIGCHLD, default action: discard signal, description: child status has changed
- 30 SIGUSR1, default action: terminate process, description: User defined signal 1
- 31 SIGUSR2, default action: terminate process, description: User defined signal 2

```
man 7 signal
```

En el shell

- El comando `kill` envía una señal. Por omisión envía un SIGTERM.
  `$> kill -9 2324`
- El comando `killall` envía una señal a los procesos fijándose en su nombre.
- El comando `pkill` hace lo mismo, pero aplicando un patrón (expresión regular).

En el shell

¿Recuerdas el *job control*?

- Toda sesión tiene un terminal controlador (tty).
- Una sesión tiene distintos grupos de procesos. La sesión está liderada por un proceso. El SID (id de la sesión) es el PID del *proceso líder de la sesión*.
- Un grupo de procesos está liderado por un proceso. Un grupo de procesos tiene un id: PGID, que es el PID del *proceso líder del grupo*. Un *job* es un grupo de procesos.
- Como mucho, un grupo de procesos de la sesión está en primer plano en el terminal.
- El terminal manda las señales a todos los procesos del grupo que está en primer plano. P. ej. Ctrl-c
- Si un proceso de un grupo que no está en foreground intenta leer del terminal, se le manda un SIGTTIN y se quedará parado.

En el shell

Ejemplo:

```
$>  sleep 1000 | cat | wc &
[1] 6308
$> jobs
[1]+  Running                 sleep 1000 | cat | wc &
$> ps  j
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 4381  4391  4391  4391 pts/0     6310 Ss    1000   0:00 /bin/bash
 4391  6306  6306  4391 pts/0     6310 S     1000   0:00 sleep 1000
 4391  6307  6306  4391 pts/0     6310 S     1000   0:00 cat
 4391  6308  6306  4391 pts/0     6310 S     1000   0:00 wc
 4391  6310  6310  4391 pts/0     6310 R+    1000   0:00 ps j
```

Sesión y grupos de procesos

- El grupo de procesos y la sesión se hereda del padre.
- Después de un `exec` se conserva el grupo de procesos.
- La llamada al sistema `setpgid` sirve para cambiar el grupo de procesos.
- La llamada al sistema `setsid` crea una nueva sesión, siendo el proceso llamador el líder.
- Para crear un *demonio* (un servicio), debemos crear una sesión y dejarlo sin terminal controlador. Eso lo hace la función `daemon` de la libc.

Llamada al sistema: kill

- `kill`: manda una señal a un proceso. Para poder hacerlo, el UID del proceso destino debe ser el mismo que el del proceso (o ser root), la única excepción es SIGCONT (pero el proceso deber ser descendiente).
- Si `pid` es 0, se pone la señal a todos los procesos del grupo de procesos al que pertences. Si es -1, manda la señal a todos los procesos que se pueda menos a los del sistema y él mismo.

```
int kill(pid_t pid, int sig);
```

Llamada al sistema: killpg

- `killpg`: como la anterior, pero manda la señal a un grupo de procesos.

```
int killpg(pid_t pgrp, int sig);
```

Manejo de señales

Un proceso puede tener una señal:

- No bloqueada. En ese caso puede tener distintas acciones definidas:
  - La acción por omisión para dicha señal (ver tabla mostrada anteriormente).
  - Una acción acción especial: manejador de señal.
  - Ignorarla.
- Bloqueada. Si te ponen una señal, se queda pendiente de entregar (se entregará cuando se desbloquee).

Llamada al sistema: signal

- `signal`: Asigna una acción para una señal. Registra un manejador para una señal, pone su acción por omisión (SIG_DFL) o la marca como ignorada (SIG_IGN).
- Sobrescribe el estado anterior.
- Es una versión simplificada de `sigaction`. Usaremos `signal` por simplicidad.
- No es portable, difiere en distintos sistemas, y ha variado con el tiempo.

```
typedef void (*sig_t) (int);
sig_t signal(int sig, sig_t func);
```

Si no quieres esperar por tus hijos...

- Si tu programa no quiere esperar (llamar a `wait`) por los procesos que crea, se debe avisar al sistema para que no se queden **zombies**.
- Se hace ignorando la señal SIGCHLD.

```
signal(SIGCHLD, SIG_IGN);
```

Manejadores

- Los manejadores no retornan nada (`void`).
- Se usará la pila del proceso si no se dice lo contrario (p. ej. con `signalstack` de glibc).
- Como se pueden recibir señales mientras que se maneja una señal $\rightarrow$ manejador debe ser **reentrante**: se tiene que poder llamar de nuevo antes de que haya retornado. Para ello: no usar variables globales/estáticas, no debe llamar a funciones no reentrantes (p. ej. malloc, free, etc.), no modificar `errno`, ...
- Debería ser lo más pequeño que se pueda.

Conclusión: si no necesitas de verdad manejar señales, no lo hagas. Es más complicado de lo que parece.

Llamada al sistema: alarm

- `alarm`: programa un temporizador para recibir un SIGALRM. Llamando con el valor 0, se anula el temporizador. Si se llama dos veces, se sobrescribe el temporizador.
- Se usa para poner un *timeout*. Hay que usarlo con precaución y mesura.

```
unsigned int alarm(unsigned int seconds);
```

Interrumpiendo llamadas

- Algunas llamadas al sistema son *lentas*: el proceso se puede bloquear mientras realiza la llamada: read, write, open, etc.
- Si se pone una señal mientras que se está realizando una llamada al sistema lenta, la llamada fallará (es interrumpida).
- Se puede pedir que las llamadas al sistema lentas que lo soporten, sean reiniciadas (si es posible) cuando se recibe una señal.
- Esto se hace con `siginterrupt`: activa (0) o desactiva (1) el reinicio para una señal.

```
int siginterrupt(int sig, int flag);
```

Bloquear señales

- Las señales se pueden bloquear temporalmente en partes de nuestro programa en las que no queremos ser interrumpidos.
- **No es lo mismo** que ignorar la señal.
- Si una señal está bloqueada y se recibe, se queda pendiente de entregar. No se pierden.
- En ese caso, cuando desbloqueamos la señal, se entregará la pendiente.

Bloquear señales

- La **máscara de señales** del proceso indica qué señales están bloqueadas.
- Podemos modificar o ver las señales bloqueadas con la llamada al sistema `sigprocmask`.
- `sigpending` nos da el conjunto de señales que están pendientes de entregar.

Señales

- El kernel apunta los manejadores de señales en la estructura del proceso.
- Todo el estado del proceso relacionado con las señales se hereda del padre.
- Después de un `exec` se restablecen las acciones por omisión, pero se respeta la configuración sobre señales bloqueadas.

Señales

¿Cómo funcionan las señales?

1. La señal se entrega al entrar al kernel por cualquier motivo (llamada al sistema, interrupción, etc.) o al salir del kernel.
2. Se añade un *signal frame* en la pila que contiene el contexto actual del proceso (registros, etc).
3. Se mete en la pila un PC de retorno que corresponde al stub de la libc de la llamada al sistema `sigreturn`.
4. Se pone a ejecutar al manejador de la señal.
5. Cuando el manejador retorna, se retorna al stub de la llamada al sistema `sigreturn()`: se entra al kernel.
6. La llamada al sistema saca el *signal frame* de la pila y restaura el contexto.
7. Cuando se vuelva a planificar el proceso, continuará donde fue interrumpido.

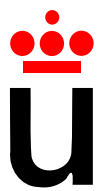
Concurrencia básica

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

Concurrencia

- Ya hemos visto varios problemas de concurrencia. P. ej.:
  - ¿Ejecuta antes el padre o el hijo?
  - ¿Qué pasa si dos procesos intentan escribir un fichero *a la vez*?
  - ¿Cómo se sincronizan dos procesos usando un pipe?
  - ...

ver fork0.c

Condición de carrera

- Recuerda: tenemos una **condición de carrera** cuando el resultado depende de qué flujo de ejecución llegue el primero y **no sabemos qué pasará**.
- El problema se complica si tenemos **memoria compartida**: varios flujos de ejecución leen/escriben variables compartidas concurrentemente.

Condición de carrera

- Una **condición de carrera** es el peor bug posible.
- En general, no es reproducible.
- Ocurre con cierta probabilidad (puede ser muy baja).
- Los resultados pueden ser sorprendentes y **muy difíciles de entender**.

Ejemplo 0

Ejemplo[1]: la variable x (inicializada a 0) está compartida entre dos flujos de control (A y B) que ejecutan este código, ¿cuáles son los posibles valores finales de la variable x? ¿Máximo y mínimo?

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

[1] Suponiendo que las líneas de este programa son atómicas.

Ejemplo 0

(1) A ejecuta hasta la línea 3:

```
1:          int i = 0, aux = 0;        <= B(x=0,aux=0,i=0)
2:          for(i=0; i<10; i++){
3:               aux = x;              <= A(x=0,aux=0,i=0)
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(2) B ejecuta 9 iteraciones y se queda en la línea 2:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){   <= B(x=9,aux=9,i=9)
3:               aux = x;          <= A(x=9,aux=0,i=0)
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(3) A ejecuta su primera vuelta, se queda en la línea 2 y deja `x` a 1:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){  <= B(x=1,aux=9,i=9), A(x=1,aux=1,i=1)
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(4) B avanza hasta la línea 3, su aux queda con valor 1:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){   <= A(x=1,aux=1,i=1)
3:               aux = x;          <= B(x=1,aux=1,i=9)
4:               aux = aux + 1;
5:               x = aux;
6:          }
```

Ejemplo 0

(5) A ejecuta todas sus vueltas:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++{
3:               aux = x;          <= B(x=9,aux=1,i=9)
4:               aux = aux + 1;
5:               x = aux;
6:          }
                                   <= A(x=9,aux=9,i=10)
```

Ejemplo 0

(6) B termina su última iteración:

```
1:          int i = 0, aux = 0;
2:          for(i=0; i<10; i++){
3:               aux = x;
4:               aux = aux + 1;
5:               x = aux;
6:          }
                                  <= A(x=2,aux=9,i=10),  B(x=2,aux=2,i=10)
```

$$x = 2$$

Ejemplo 1

Dos flujos de ejecución A y B que comparten la variable `x` inicializada a 0 ejecutan estas sentencias concurrentemente:

```
x++;
printf("%d\n", x);
```

Posibles resultados:

- A imprime 1 y B imprime 2.
- B imprime 1 y A imprime 2.
- A y B imprimen 2.
- Dependiendo en las instrucciones que requiera el incremento en la arquitectura, pueden imprimirse cualquier par de enteros.

Atomicidad

- Si una operación sobre el recurso compartido no es **atómica**, dos flujos pueden interferir entre ellos $\rightarrow$ resultado de las operaciones incorrecto.
- **Operación atómica**: operación indivisible, ninguna otra operación que realice otro flujo puede interferir con ella.
- En general, pocas operaciones son atómicas y debemos sincronizar los flujos.

Ejemplo 2

En el mismo escenario que el Ejemplo 1, con una variable compartida adicional llamada `busy` que está inicializada a 0:

```
if(! busy){
      busy = 1;
      x++;
      printf("%d\n", x);
      busy = 0;
}
```

¿Hemos eliminado la condición de carrera? **No**:

- A evalúa la condición del if: true.
- B evalúa *"a la vez"* la condición del if: true.
- A ejecuta el cuerpo del if mientras que B ejecuta el cuerpo del if.
- Mismos posibles resultados que en el Ejemplo 1.

Exclusión mutua

- Lo que estaba intentando hacer el Ejemplo 2 sin éxito es proporcionar **exclusión mutua**: cuando un flujo está dentro de una región, no puede entrar ningún otro.
- Se llama **región crítica** a esa región del programa donde se debe proporcionar exclusión mutua para evitar una colisión al usar un recurso compartido.

Exclusión mutua

- Para hacerlo bien, necesito una operación que compruebe y escriba la variable `busy` de forma atómica.
- Inhabilitar interrupciones **no** es suficiente:
  - Cada CPU tiene sus propias interrupciones $\rightarrow$ ¿funciona en un multiprocesador?
  - No es deseable que los procesos de usuario puedan desactivar todas las interrupciones de la CPU $\rightarrow$ ¿qué pasa si hay un bug?
- ¡Necesito soporte hardware! (cerrar bus de memoria, evitar interrupciones, etc.)

TAS

- Los procesadores incluyen operaciones atómicas como test-and-set (TAS):
  - Si está a *false*, lo pone a *true* y devuelve *false*.
  - Si está a *true*, lo pone a *true* y devuelve *true*.
- Algunos procesadores no tienen TAS, pero sí tienen otras operaciones atómicas que nos permiten implementar TAS.

Instrucciones equivalentes a TAS

Por ejemplo, en AMD64 tenemos XCHGL:

- Intercambia los dos valores de forma atómica.
- Del manual de intel:

[...] the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL [...]

Implementación de TAS en Plan 9/AMD64

```
TEXT        _tas(SB),$0
        MOVL        $0xdeadead,AX   // escribe literal en AX
        MOVL        1+0(FP),BX      // escribe puntero en BX
        XCHGL       AX,(BX)         // intecambia AX con *BX
        RET                         // retorna AX
```

Primitiva: lock/cerrojo

- Con TAS ya podemos tener exclusión mutua...
- ... pero es deseable tener una abstracción de más alto nivel, que sea más fácil de usar que TAS.
- Con TAS podemos implementar la primitiva **lock (cierre o cerrojo)**.

Primitiva: lock/cerrojo

- Un cerrojo tiene dos operaciones:
  - lock(): coge el cierre.
  - unlock(): suelta el cierre.
- Para entrar en la región crítica (i.e. usar el recurso compartido), hay que tener cogido el cierre.
- Si un cierre está libre, se puede coger.
- Si se intenta coger un cierre ya cogido, el flujo se queda bloqueado hasta que pueda cogerlo.
- Después de usar el recurso compartido, siempre se debe soltar el cierre.

Ejemplo 3

En el mismo escenario que el Ejemplo 1, con una variable compartida adicional `lk` de tipo cerrojo:

```
lock(&lk);
x++;
printf("%d\n", x);
unlock(&lk);
```

¿Hemos eliminado la condición de carrera? Ahora el contador es coherente. Ya sólo pueden pasar dos cosas:

- A imprime 1 y B imprime 2.
- A imprime 2 y B imprime 1.

Si queremos un orden específico, necesitamos más sincronización.

Primitiva: lock/cerrojo

- ¿Y si quiero proteger distintos recursos compartidos? Para optimizar el uso concurrente, puedo usar distintos cerrojos...
- Problema cuando hay varios cerrojos: **deadlocks** o interbloqueos. P. ej.:
  - A coge lk1, espera a que B suelte lk2
  - B coge lk2, espera a que A suelte lk1
  - A espera a B, pero B no acaba porque espera a A $\rightarrow$ dependencia circular, **ninguno progresa**.

```
// A ejecuta...

lock(&lk1);
lock(&lk2);
//blabla
unlock(&lk2);
unlock(&lk1);
```

```
// B ejecuta...

lock(&lk2);
lock(&lk1);
//blabla
unlock(&lk1);
unlock(&ll2);
```

Primitiva: lock/cerrojo

Hay que tener mucho cuidado con los locks. En general:

- Coger los locks siempre en el mismo orden.
- `lock()`/ `unlock()` en la misma función.
- `lock()`/ `unlock()` al mismo nivel de tabulación.
- Todos los locks al mismo nivel de abstracción.
- Tener claro qué funciones cogen el lock y cuales no.

Buenas prácticas: cada return con su unlock

Mal:

```
lock(&lk);
if (bla) {
                    dosomething();
                    doanotherthing();
}
else
                    return;
unlock(&lk);
```

Buenas prácticas: cada return con su unlock

Mejor:

```
lock(&lk);
if (bla) {
                    dosomething();
                    doanotherthing();
}
else {
                    unlock(&lk);
                    return;
}
unlock(&lk);
```

Buenas prácticas: cada return con su unlock

Todavía mejor (no siempre se puede):

```
lock(&lk);
if (bla) {
                    dosomething();
                    doanotherthing();
}
unlock(&lk);
return;
```

Contention/Contienda

- Cuando dos flujos de ejecución compiten por entrar a la vez en la región crítica, uno espera.
- Si muchos intentan entrar $\rightarrow$ **contienda** elevada $\rightarrow$ muchos esperan.
- Eso es malo en general.
- Siempre debemos mantener las regiones críticas lo más pequeñas que se pueda.

Primitiva: lock/cerrojo

En general, hay dos tipos de cerrojos:

- **Spin locks**. El flujo que se bloquea en el cerrojo hace **espera activa**: el flujo consume tiempo de CPU iterando hasta poder coger el cerrojo. Sólo se deben usar cuando hay **poca contienda** y la región crítica es pequeña. No son justos.
- **Queue locks** (también conocidos como **mutexes**). El flujo que se bloquea en el cerrojo suelta el procesador y pasa a no estar listo para ejecutar. Cuando pueda coger el cierre, volverá a estar listo para ejecutar y el planificador le dará CPU cuando toque. Suelen ser FIFO (depende de la implementación).

Ejemplo de implementación de spin locks con TAS

```
void
lock(Lock *lk)
{
        int i;


        if(!_tas(&lk->val))             /* once fast */
                return;
        for(i=0; i<1000; i++){          /* a thousand times pretty fast */
                if(!_tas(&lk->val))
                        return;
                sleep(0);
        }
        for(i=0; i<1000; i++){          /* now nice and slow */
                if(!_tas(&lk->val))
                        return;
                sleep(100);
        }
        while(_tas(&lk->val))           /* take your time */
                sleep(1000);
}
```

Ejemplo de implementación de spin locks con TAS

```
void
unlock(Lock *lk)
{
        /* on AMD64, this is atomic */
        lk->val = 0;
}
```

Primitiva: cierre de lectores/escritores

Hay un tipo de cerrojo especial llamado lock de lectores/escritores. Tiene dos tipos de operaciones para coger el cerrojo:

- Los que van a leer pueden hacerlo concurrentemente (N lectores). Cogen el cierre en modo lectura.
- El que viene a escribir necesita estar solo (1 escritor). Coge el cierre en modo escritura.
- El nivel de concurrencia es mayor cuando hay muchas lecturas y pocas escrituras (caso común).

Otras primitivas

Además de los cerrojos, existen muchos otros mecanismos de sincronización con distinta semántica:

- Futex
- WaitGroup
- Semáforos
- Barreras
- Rendezvous
- Variables condición y monitores
- Canales
- ...

Si vamos a hacer programas concurrentes, tenemos que estudiar detenidamente los mecanismos ofrece el sistema en el que trabajamos y programar con **mucho cuidado**.

Pthreads

Pthreads

- Un thread (hilo) es la unidad mínima de utilización de CPU.
- Hay distintos modelos de threads.
- Por ahora, los podemos ver como procesos que comparten su memoria: TEXT, DATA, BSS.
- ¿Qué pasa con la pila?
- El estándar POSIX tiene su interfaz: pthreads. Cada sistema puede implementarla de una forma distinta.
  `man 7 pthreads`

Pthreads

- Los threads comparten las variables globales.
- Dogma: cada vez que toquemos un recurso compartido (p. ej. una variable compartida), nos tenemos que proteger.
- Si no hacemos eso, nos metemos en problemas.
- Si usas una biblioteca, asegúrate de que es *thread_safe*.
- En caso de duda, protege.

Pthreads

- `pthread_create()`: crea un thread, que comenzará ejecutando la función que se le pasa en su tercer parámetro. El primer parámetro es un puntero a la variable de tipo `pthread_t` que identificará al thread creado.
- `pthread_join()`: espera a que muera el thread indicado en su primer parámetro.

Debemos enlazar el programa con la biblioteca de threads:

```
gcc -c -Wall -Wshadow -g miprograma.c
gcc -o miprograma miprograma.o -lpthread
```

ver pthreads.c
ver pthreads-race.c

Cerrojos en pthreads

Tenemos:

- Spin locks: tipo de datos `pthread_spin_lock_t`.
- Mutex: tipo de datos `pthread_mutex_t`.

pthread_spin_lock_t

Operaciones básicas de `pthread_spin_lock_t`:

- `pthread_spin_init()`: inicializa el cerrojo. El segundo parámetro indica si se comparte con otros procesos (usar `PTHREAD_PROCESS_SHARED`).
- `pthread_spin_lock()`: coge el cerrojo.
- `pthread_spin_unlock()`: suelta el cerrojo.
- `pthread_spin_destroy()`: libera los recursos asociados al cerrojo.

ver pthreads-spin.c

pthread_mutex_t

Operaciones básicas de `pthread_mutex_t`:

- `pthread_mutex_init()`: inicializa el cerrojo. Si el segundo parámetro se usa para establecer ciertos atributos, si es NULL se ponen los atributos por omisión.
- `pthread_mutex_lock()`: coge el cerrojo.
- `pthread_mutex_unlock()`: suelta el cerrojo.
- `pthread_mutex_destroy()`: libera los recursos asociados al cerrojo.

ver pthreads-mutex.c

Sincronización con ficheros

Sincronización con ficheros

Cuando desde varios procesos se opera con el mismo fichero, tenemos condiciones de carrera. Ejemplos:

- Cuando varios procesos quieren crear el mismo fichero, hay una condición de carrera.
- Cuando un proceso escribe en un fichero mientras que otros leen/escriben del mismo fichero, tenemos una condición de carrera. En general, no se garantiza atomicidad en las escrituras.

ver writerace.c

Open

- El modo `O_CREAT|O_EXCL` hace que la llamada `open()` falle si existe el fichero que se quiere crear. En ese caso `errno` será `EEXIST`.
- Muchas aplicaciones usan esto como un TAS y la existencia de un fichero como un cerrojo (p. ej. Firefox para mantener una única instancia de la aplicación ejecutando).
- Hay que tener cuidado con la posición (offset). Si siempre queremos escribir al final: `O_APPEND`. Si múltiples procesos añaden concurrentemente, todos los bytes escritos se escribirán al final, pero puede que los bytes del mismo `write` no terminen contiguos.

Flock

- `flock`: sirve para usar un **cierre de lectores/escritores** sobre el fichero.
  Tiene tres operaciones:
  - `LOCK_EX`: echa el cierre de escritores.
  - `LOCK_SH`: echa un cierre de lectores.
  - `LOCK_UN`: soltar el cierre que tienes.
- Se puede especificar que no sea bloqueante con `|LOCK_NB`. En ese caso, si no puedes coger el cierre la operación da error (no se bloquea).

```
int flock(int fd, int operation);
```

ver filerace.c

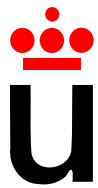
Shell scripting

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

21 de septiembre de 2022



Universidad
Rey Juan Carlos

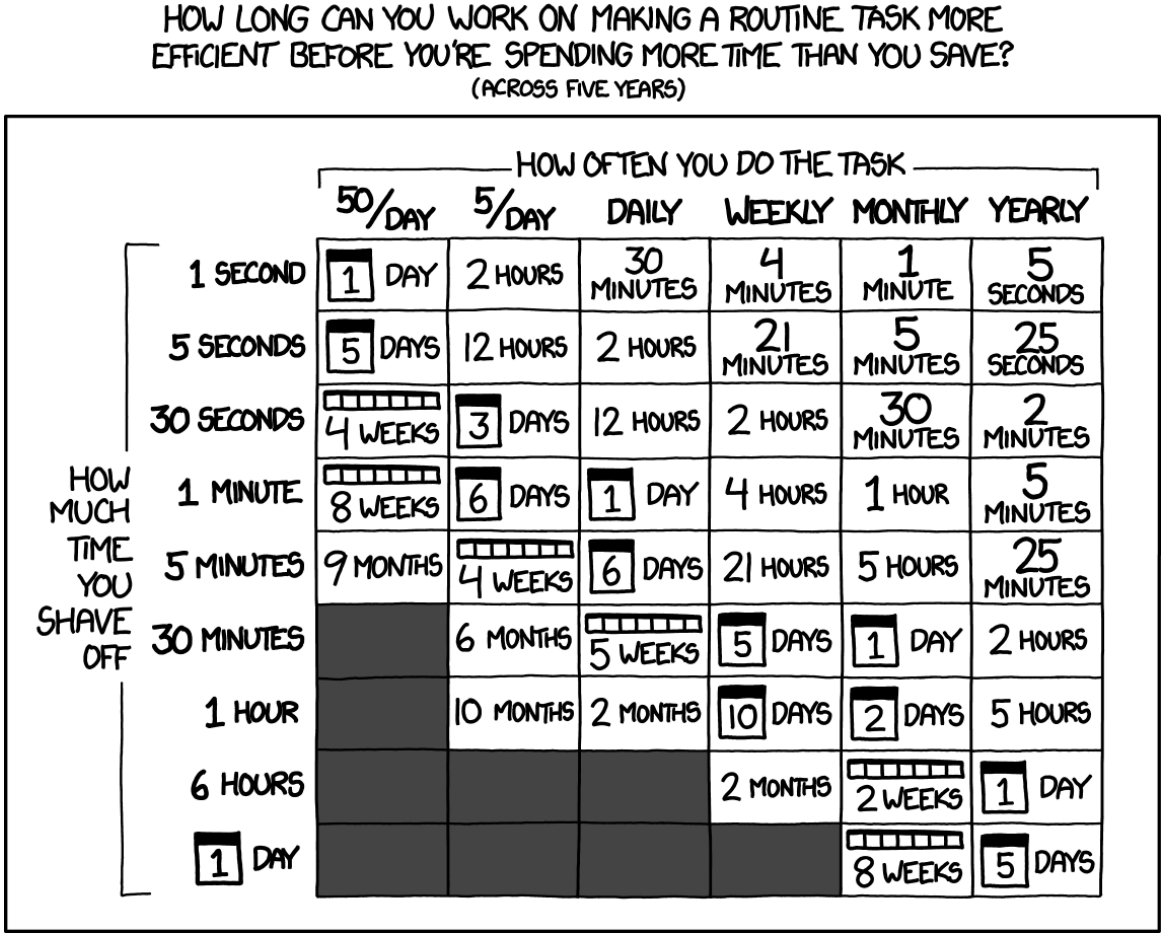
¿Cuándo hago un script de Shell?

- Pasos para realizar una tarea:
  1. Mirar si hay alguna herramienta que haga lo que queremos → buscar en el manual.
  2. Si no encontramos, intentar combinar distintas herramientas → **programar un script de Shell**. La primera aproximación debe ser pipelines de filtros, etc.
     IDEA: combinar herramientas que hacen bien una única tarea para llevar a cabo tareas más complejas.
  3. Si no podemos, hacer una herramienta → programada en C, Python, Java, Ada, Go, ...

¿Qué tipo de cosas ?

- La shell es especialmente buena
  - Para tareas que hago una vez
  - Para automatizar tareas (con un IDE, Makefile)
  - Para procesar texto

Automatizar



Shells

- `sh` es la shell original de Unix, escrita por Ken Thompson. Fue rescrito por Stephen Bourne en 1979 para Unix Version 7: *bourne shell*.
- Los sistemas derivados usan distintas shells: `sh`, `ash`, `bash`, `dash`, `ksh`, `csh`, `tcsh`, `zsh`, `rc`, etc.
- Cada una tiene sus características, pero también tienen mucho en común.
- En sistemas modernos, `/bin/sh` suele ser un enlace simbólico a su shell por omisión para ejecutar scripts. En Ubuntu y Debian es dash[1].
- Política: los scripts que tienen `#!/bin/sh` deben usar únicamente las características POSIX (IEEE Std 1003.1-2017): el subconjunto común que tienen la mayoría de las shells. Así, los scripts pueden ser portables entre distintos sistemas.

[1] No confundir con el shell por omisión para un terminal, que es bash.

Un script:

- Tiene que tener permisos de ejecución.
- Hay comandos que se implementan dentro del shell (no se ejecuta un fichero externo al shell, es una parte del propio shell). Se llaman *built-in*. Si se quiere ver si es builtin *type* o *whatis*.
- El comando *built-in* `exit` sale del script con el status indicado en su argumento.
- Si un script no sale con `exit`, deja el status que tiene `$?`.

```
#!/bin/sh

# este es un hola mundo en sh
# esto es un comentario

echo hello world
exit 0
```

Un script:

- Una ventaja de la shell, es que puedo probar de forma interactiva
- No escribo el script directamente, voy probando los comandos
- O ni siquiera escribo el script (escribo los comandos directamente)

Ya conocemos:

- | es un pipe
- & ejecuta un comando en background
- $ se usa para las variables. $var es lo mismo que ${var}
- " y ' se usan para escapar cadenas (las dobles expanden algunas cosas)
- $>$, $<$ y $>>$ son redirecciones
  - 5> para redir del fd 5
  - 5>&3 para hacer dup del 5 al 3
  - Cuando hay varios el orden importa ej: `2>&1 >/dev/null`
- \ se usa para escapar caracteres
- && , || para ejecución condicional
- Globbing (wildcards): ? * $[a-z]$ etc.

Parámetros posicionales

- Se pueden acceder a los parámetros que se han pasado al script con $1, $2, $3 ...
- $0 expande al nombre con el que se ha invocado el script.
- $# expande al número de parámetros (sin contar el 0).
- $* expande a los parámetros posicionales.
- "$*" expande a "$1 $2 ..."
- $@ expande a los parámetros posicionales (igual que $* pero separados)
- "$@" expande a "$1" "$2" ...
- `shift` desplaza los parámetros (p. ej. $4 pasará a ser $3). Se actualiza el valor de $#.
  - Útil para parámetros optativos (pongo lo que sea, o hago shift, el resto igual)

Agrupaciones

- Si queremos ejecutar comandos en un subshell:

(comando; comando; ...)

- Si queremos ejecutar una agrupación de comandos en el shell actual:

{ comando; comando; ... }

Ejemplo:

```
$> { echo uno; echo dos; } | tr o 0
un0
d0s
$> { echo los ficheros de /tmp son; ls /tmp; } > ficheros
```

Agrupaciones

- Ejecutar en un subshell útil
- Para no cambiar el entorno en el shell actual (`cd`, `export`)

Ejemplo:

```
#sigo en tmp al final:
$> pwd; (cd /etc; ls apt;); pwd;
/tmp
apt.conf.d      sources.list
preferences.d   sources.list.d   trusted.gpg    trusted.gpg.d
/tmp
#BLA no existe al final:
$> echo z$BLA; (export BLA=bla; echo $BLA;); echo z$BLA;
z
bla
z
$>
```

Sustitución de comando

- Se sustituye el comando por su salida.
- Se puede escribir de dos formas:

  `$(comando)`

  `‘comando‘`

```
$>  l=$(wc -l /tmp/a | cut -d' ' -f1)
$>  echo $l
31
$>
```

if

Las condiciones depende del status de salida del comando: éxito es verdadero, fallo es falso.

```
if comando
then
        comandos
elif comando
then
        comandos
else
        comandos
fi
```

case

Los casos pueden contener patrones de globbing.

```
case palabra  in
patrón1)
     comandos
     ;;
patrón2 | patrón3)
     comandos
     ;;
*)   # este es el default
     comandos
     ;;
esac
```

Bucles

```
while comando
do
        comandos
done
```

```
for variable in palabra1 palabra2 palabraN
do
        comandos
done
```

Read

- El comando `read` lee una línea de su entrada estándar y la guarda en la variable que se le pasa como argumento.
- Se puede usar para procesar la entrada línea a línea en un bucle.
- Solo debemos hacer eso cuando no tenemos ningún filtro o pipeline que nos sirva para hacer lo que queremos.

Read

- Por ejemplo, esto itera 2 veces:

```
echo 'a b
c d' > /tmp/e

while read line
do
  echo $line
done <  /tmp/e
```

  - Esto itera 4 veces:

```
for x in `cat /tmp/e`
do
  echo $x
done
```

Variable IFS

- Esta variable contiene los caracteres que se usan como separadores entre campos.
- Por omisión contiene el tabulador, espacio y el salto de línea.
- Hay que tener cuidado: cambiar el valor de esta variable rompe las cosas.
- Mejor en subshell.

```
$>  for i in $(echo uno dos tres) ; do echo $i ; done
uno
dos
tres
$>  export IFS=-
$>  for i in $(echo uno dos tres) ; do echo $i ; done
uno dos tres
$
```

Funciones

- Se pueden definir funciones. Sus parámetros se acceden como los parámetros posicionales. P. ej.:

```
hello () {
        echo hola $1
        shift
        echo adios $1
}

...
# ejecutamos la función
hello uno dos
```

Test

El comando `test` sirve para comprobar condiciones de distinto tipo.

Ficheros:

- -f fichero
  si existe el fichero
- -d dir
  si existe el directorio

Test

Cadenas:

- -n String1
  si la longitud de la string no es cero
- -z String1
  si la longitud de la string es cero
- String1 = String2
  si son iguales
- String1 != String2
  String1 and String2 variables no son idénticas
- String1
  si la string no es nula

Test

Enteros:

- Integer1 -eq Integer2
  si los enteros Integer1 e Integer2 son iguales.

Otros operadores:

- -ne: not equal
- -gt: greater than
- -ge: greater or equal
- -lt: less than
- -le: less or equal

Test

Test también se puede usar así:

- Esto:

```
[  $a -eq $b  ]
```

- es lo mismo que esto:

```
test $a -eq $b
```

Operaciones aritméticas

Para operaciones básicas con enteros podemos usar el propio shell. También podemos usar el comando `bc`.

- Esto:

```
$((5 + 7))
```

- se reemplaza por

```
12
```

Filtros útiles

- `sort`
  ordena las líneas de varias formas.
- `uniq`
  elimina líneas contiguas repetidas.
- `tail`
  muestra las últimas líneas.

P. ej:

```
$> ps | tail +3 # a partir de la 3ª
$> ps | tail -3 # las 3 ultimas
$> seq 1 1000 | sort
$> seq 1 1000 | sort -n
```

Sort

- Puede recibir una lista de columnas (empezando por la 1)
- Y un separador
- Y ordena por esos campos como clave (es un intervalo de campos)
- Ojo con estabilidad (`-s`)

```
$> cat x.txt
1-2-4
2-3-3
2-2-1
2-1-4
$> sort -k2 -t\- x.txt
2-1-4
2-2-1
1-2-4
2-3-3
$> sort -k1,2 -t\- x.txt
1-2-4
2-1-4
2-2-1
2-3-3
```

Comandos útiles

- `diff`
  compara ficheros de texto línea a línea
- `cmp`
  compara ficheros binarios byte a byte

P. ej:

```
$> diff -n fich1 fich2
$> cmp fich1 fich2
```

Tr

- Traduce caracteres. El primer argumento es el conjunto de caracteres a traducir. El segundo es el conjunto al que se traducen. El enésimo carácter del primer conjunto se traduce por el enésimo carácter del segundo.
- `-d`
  Borra los caracteres del único conjunto que se le pasa como argumento.
- Se le pueden dar rangos, p. ej.
  `$> cat fichero | tr a-z A-Z`

Expresiones regulares (*regexp*)

- Es un lenguaje formal para describir/buscar cadenas de caracteres.
- Parecidas a los patrones de la Shell o de globbing, pero más potentes.
- Veremos las que se llaman *extended regular expressions*. Es un estándar de POSIX.
- Una string encaja con sí misma, por ejemplo `'a'` con `'a'`.

Expresiones regulares (*regexp*)

- `.`
  encaja con cualquier carácter, por ejemplo 'a'.
- `[conjunto]`
  encaja con cualquier carácter en el conjunto, por ejemplo `[abc]` encaja con `'a'`. Se pueden especificar rangos, p. ej. `[a-zA-Z]`.
- `[^conjunto]`
  encaja con cualquier carácter que **no esté** en el conjunto, por ejemplo `[^abc]` NO encaja con `'a'`, sin embargo sí encaja con `'z'`.

Expresiones regulares (*regexp*)

- ^

  encaja con *principio de línea*.
- $

  encaja con *final de línea*.
- Una regexp e_1 concatenada a otra regexp e_2, $e_1 e_2$, encaja con una string si una parte p_1 de la string encaja con e_1 y otra parte contigua, p_2, encaja con e_2.

P. ej:

`'az'` encaja con la regexp `[a-d]z`

Expresiones regulares (*regexp*)

- `exp*`
  encaja si aparece **cero o más veces** la regexp que lo precede.
- `exp+`
  encaja si aparece **una o más veces** la regexp que lo precede.

P. ej:

`'aaa'` encaja con la regexp `a*`

`'baaa'` encaja con la regexp `ba+`

`'bb'` encaja con la regexp `ba*`

`'bb'` no encaja con la regexp `ba+`

Expresiones regulares (*regexp*)

- `exp?`
  encaja si aparece **cero o una vez** la regexp que lo precede. Se utiliza para partes opcionales.
- `(exp)`
  agrupa expresiones regulares.

P. ej:
`'az', 'av', 'a'` encajan con la regexp az?
`'abab'` encaja con la regexp `(ab)+`
`'abab', 'ababab', 'abababababa'` encajan con la regexp `(ab)+`

Expresiones regulares (*regexp*)

- `exp | exp`
  si encaja con alguna de las regexp que están separadas por la barras
- `\`
  carácter de escape: hace que el símbolo pierda su significado especial.

P. ej:

`'aass'` encaja con la regexp `(aass|booo)`

`'hola*'` encaja con la regexp `a\*`

Egrep

- Filtra líneas usando expresiones regulares.
- `-v`
  Realiza lo inverso: imprime las líneas que no encajan.
- `-n`
  Indica el número de línea.
- `-e`
  indica que el siguiente argumento es una expresión.
- `-q`
  silencioso, no saca nada por la salida (cuando solo nos interesa el status de salida).

Sed

- Stream Editor
- Editor de flujos de texto con comandos.
- Basado en Ed (editor con comandos, tatarabuelo de vi).
- Muchas de las cosas de sed, igual en ed.

Sed

- Es un editor: aplica el comando de sed a cada línea que lee y escribe el resultado por su salida. Sin el modificador `-n`, escribe todas las líneas después de procesarlas.
- Si queremos usar expresiones regulares extendidas, hay que usar la opción -E.
- Comandos:
  `q` $\rightarrow$ Sale del programa.
  `d` $\rightarrow$ Borra la línea.
  `p` $\rightarrow$ Imprime la línea. (correr con -n)
  `r` $\rightarrow$ Lee e inserta un fichero.
  `s` $\rightarrow$ Sustituye. $\leftarrow$ la que más se usa!!!

Sed

- Direcciones:
  `número` → actúa sobre esa línea.
  `/regexp/` → líneas que encajan con la regexp.
  `$` → la última línea.
- Se pueden usar intervalos:
  `número,número` → actúa en ese intervalo.
  `número,$` → desde la línea *número* hasta la última.
  `número,/regexp/` → desde la línea *número* hasta la primera que encaje con regexp.

Sed

Ejemplos:
`sed '3,6d'` $\rightarrow$ borra las líneas de la 3 a la 6
`sed -E -n '/BEGIN|begin/,/END|end/p'` $\rightarrow$ imprime las líneas entre esas regexp
`sed '3q'` $\rightarrow$ imprime las 3 primeras líneas.
`sed -n '13,$p'` $\rightarrow$ imprime desde la línea 13 hasta la última.
`sed -E '/[Hh]ola/d'` $\rightarrow$ borra las líneas que contienen 'Hola' u 'hola'.

Sed

Sustitución

- `sed 's/regexp/sustitución/'` $\rightarrow$ sustituye la primero subcadena que encaja con la exp. por la cadena *sustitución*.
- `sed 's/regexp/sustitución/g'` $\rightarrow$ sustituye todas las subcadenas de la línea que encajan con la exp. por la cadena *sustitución*.
- `sed 's/(regexp)regexp.../ \1 sustitución/g'` $\rightarrow$ usa las subcadenas que encajaron con las agrupaciones en la cadena de sustitución.

Sed

Ejemplos

`sed 's/[0-9]/X/'` $\rightarrow$ el primer dígito de la línea se sustituye por una X.

`sed 's/[0-9]/X/g'` $\rightarrow$ todos los dígitos de la línea se sustituyen por una X.

`sed 's/^([A-Za-z]+)[ ]+([A-Z]+)/NOMBRE:\1 NOTA:\2/g'`

hacer mykill.sh

AWK

Imprimir:

- Lenguaje completo de programación de texto.
- Útil, veremos sólo la superficie.

AWK

Imprimir:

- print
  Sentencia que imprime los operandos. Si se separan con comas, inserta un espacio. Al final imprime un salto de línea.
- printf()
  Función que imprime, ofrece control sobre el formato de forma similar a la función de libc para C:

```
$> ls -l | awk '{ printf("Size: %08d KBytes\n", $6) }'
```

AWK

Variables:

- $0
  La línea que está procesando.
- $1, $2 ...
  El primer, segundo... campo de la línea.
- NR
  Número del registro (línea) que se está procesando.
- Ejemplo
  para imprimir la tercera y segunda columna de un csv:

```
$> cat a.txt|awk -F, '{printf("%d\t%d", $3, $2)}'
```

AWK

Variables:

- NF
  Número del campos del registro que se está procesando.
- nombrevar=contenido
  Se pueden declarar variables dentro del programa. Con el modificador -v se pueden pasar variables al programa.

```
$> ls -l | awk '
{
size=$6 ; printf("Size: %08d KBytes\n", size)
}'
```

AWK

```
patrón { programa }
```

Actuando sólo en unas líneas, que se ajustan a un patrón, que puede ser:

- Expresión regular
  Se procesan las líneas que encajen con la regexp.

```
$> ls -l | awk '/[Dd]esktop/{ print $1 }'

$> ls -l | awk '$1 ~  /[Dd]esktop/ { print $1 }'
```

AWK

- Expresión de relación
  Se comparan valores y se evalúa la expresión.

```
$> ls -l | awk ' NR >= 5 && NR <= 10 { print $1 }'
```

AWK

Inicialización y finalización:

```
BEGIN{
...
}

patrón{
...
}

END{
...
}
```

AWK

Arrays asociativos:

- Cómodos, imprimir duplicadas (complétalo, falta un if):

```
$> awk '{dups[$1]++}END{for(line in dups){print line,dups[line]}}' data
```

Recorrer un árbol

- Para recorrese un árbol de ficheros
  - `du -a .`
  - `find .`

Join

- `join`
- Extremadamente útil
- Hace un *join* relacional de dos columnas (tienen que estar ordenadas)

```
$> echo '
> a bla
> b ble
> c blo' > a.txt
$> echo '
a ta
b te
c to' > b.txt
$> join a.txt b.txt
a bla ta
b ble te
c blo to
```

Join

- `join` quita las que no están en alguno de los dos (inner join)
- Tienen que estar ordenadas, usar sort antes
- Igual que sort puede usar diferentes campos

Peligro

- Ojo con las redirecciones
- Esto crea un fichero vacío
- ¿Por qué?

```
$> echo '
> a bla
> b ble
> c blo' > a.txt
$> cat a.txt | tail > a.txt
$> cat a.txt
```

Otros comandos

- Para ejecutar comandos sobre una lista
  - `echo a b c |xargs ls -l`

Comandos texto

- `cut` y `paste`
- Mejor dominar `awk` (se puede hacer todo)