UNIVERSIDAD DE GRANADA

Fundamentos del Software

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Fundamentos del Software

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

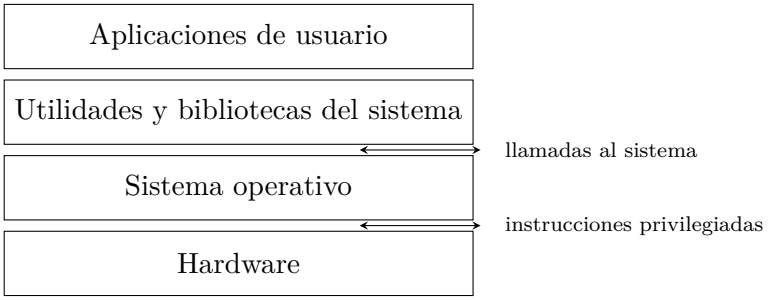
Capítulo 1

Sistema de cómputo

Tema 1 del programa. Qué componentes tiene un computador, qué hace el hardware por sí solo y dónde empieza a hacer falta el software de sistema.

1.1 Componentes

Un sistema de cómputo se organiza en capas, y cada una solo habla con la de al lado. Esa disciplina es lo que permite cambiar una sin reescribir las demás.



Capa	Qué aporta
Hardware	ejecuta instrucciones, guarda datos, conecta periféricos
Sistema operativo	gestiona los recursos y da una interfaz uniforme
Utilidades y bibliotecas	funciones y órdenes ya escritas
Aplicaciones	resuelven el problema del usuario

Los componentes físicos son cuatro, y la interconexión entre ellos importa tanto como cada uno:

Componente	Función
Procesador	ejecuta instrucciones y controla el sistema
Memoria principal	guarda datos e instrucciones en uso
Módulos de entrada y salida	conectan con el exterior
Bus del sistema	los comunica entre sí

1.2 El ciclo de instrucción

El procesador repite un ciclo elemental mientras esté encendido:

1. **Captación**: lee de memoria la instrucción cuya dirección marca el contador de programa.
2. **Decodificación**: interpreta qué operación es y qué operandos necesita.

3. **Ejecución**: la realiza.
4. **Escritura**: guarda el resultado, si lo hay.

Los registros que sostienen el ciclo:

Registro	Contiene
Contador de programa	dirección de la instrucción siguiente
Registro de instrucción	la instrucción en curso
Registro de estado	banderas de resultado y modo de ejecución
Puntero de pila	cima de la pila del programa
Registros generales	operandos y resultados

Que el ciclo sea una repetición ciega tiene una consecuencia inmediata: **el procesador no sabe esperar**. Si un periférico tarda milisegundos en responder, el procesador seguiría ejecutando instrucciones inútiles. Resolver eso es lo que justifica el resto del tema.

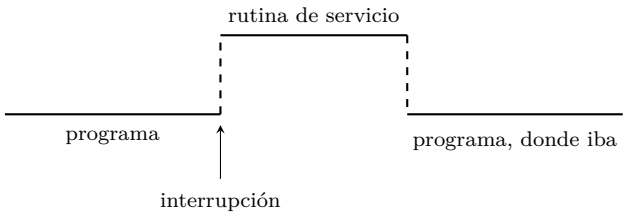
1.3 Interrupciones

Una **interrupción** es una señal que hace que el procesador suspenda lo que está haciendo, atienda un suceso y vuelva.

Tipo	Origen	Ejemplo
De hardware	un dispositivo externo	el disco termina una lectura
De reloj	el temporizador del sistema	fin de rodaja de tiempo
Excepción	la propia instrucción	división por cero, fallo de página
Llamada al sistema	una instrucción específica	el programa pide un servicio

El mecanismo, paso a paso:

1. El dispositivo activa la línea de interrupción.
2. El procesador termina la instrucción en curso, no la corta por la mitad.
3. Guarda el contador de programa y el registro de estado.
4. Consulta el **vector de interrupciones** y salta a la rutina de servicio.
5. La rutina atiende el suceso.
6. Se restaura el estado guardado y la ejecución continúa donde iba.



La diferencia entre interrupción y excepción conviene tenerla clara: la interrupción es asíncrona, la provoca algo externo y el programa interrumpido no tiene la culpa; la excepción es síncrona, la provoca la instrucción que se estaba ejecutando y por eso se puede reproducir.

Nota 1.1 . Las interrupciones se pueden inhibir mientras se ejecuta código que no debe ser interrumpido, pero las excepciones no: son consecuencia de la instrucción y no hay forma de

posponerlas. Una sección crítica del núcleo puede desactivar interrupciones, y aun así sufrir un fallo de página.

1.4 Protección

Si cualquier programa pudiera ejecutar cualquier instrucción, un error o un abuso bastaría para tumbar el sistema. El hardware ofrece tres mecanismos que el sistema operativo usa.

1.4.1 Modos de ejecución

Modo	Quién ejecuta	Qué puede hacer
Núcleo o supervisor	el sistema operativo	todo, incluidas las instrucciones privilegiadas
Usuario	las aplicaciones	solo el subconjunto no privilegiado

Un bit del registro de estado distingue los dos. Las instrucciones que manipulan la tabla de páginas, acceden a puertos de E/S o cambian el propio bit de modo son **privilegiadas**: intentarlas en modo usuario produce una excepción.

Cambiar de usuario a núcleo solo es posible por una vía controlada —una llamada al sistema, una interrupción o una excepción—, y siempre saltando a una dirección que fijó el sistema operativo. Esa es la garantía: el programa decide **cuándo** entra en el núcleo, nunca **dónde**.

1.4.2 Protección de memoria

Cada proceso solo debe acceder a su memoria. El hardware lo comprueba en cada acceso, y ese es el punto clave: es una comprobación en tiempo de ejecución, no una promesa del compilador.

Mecanismo	Cómo funciona
Registros base y límite	comprueba que la dirección cae en el intervalo asignado
Paginación	traduce direcciones y comprueba permisos por página
Segmentación	ídem por segmento, con tamaños variables

Un acceso fuera de lo permitido genera una excepción, y el sistema operativo suele terminar el proceso. Es lo que hay detrás del mensaje de violación de segmento.

1.4.3 Protección temporal

Un proceso podría no ceder nunca el procesador. El **temporizador** lo impide: genera una interrupción periódica que devuelve el control al sistema operativo, que decide si sigue ejecutando el mismo proceso u otro.

Sin temporizador no hay multiprogramación con reparto justo, solo cooperación voluntaria. Es la diferencia entre los sistemas de los años ochenta y los actuales.

1.5 Entrada y salida

Un módulo de E/S adapta la velocidad y el formato del periférico a los del bus. Hay tres formas de gobernarlo, y cada una nace de un problema de la anterior.

Técnica	Cómo	Coste para el procesador
E/S programada	consulta repetidamente el estado del módulo	lo consume entero
E/S por interrupciones	el módulo avisa al terminar	una interrupción por transferencia
Acceso directo a memoria	un controlador transfiere solo, y avisa al final	una interrupción por bloque

Las cifras explican por qué el DMA existe. Leer un bloque de disco de 4 KB con E/S por interrupciones supone una interrupción por palabra; con DMA, una por bloque. La diferencia son tres órdenes de magnitud de sobrecarga.

La jerarquía de memoria aparece por la misma razón que el DMA: el procesador es mucho más rápido que la memoria, y la memoria mucho más rápida que el disco.

Nivel	Tiempo de acceso aproximado	Capacidad
Registros	menos de 1 ns	cientos de bytes
Caché	1 a 20 ns	de kilobytes a megabytes
Memoria principal	50 a 100 ns	gigabytes
Disco de estado sólido	decenas de microsegundos	de cientos de gigabytes a terabytes
Disco magnético	milisegundos	terabytes

La jerarquía funciona por el **principio de localidad**: los programas acceden repetidamente a las mismas posiciones (localidad temporal) y a posiciones cercanas (localidad espacial). Si no fuera cierto, la caché no serviría de nada, y de hecho un recorrido aleatorio de un vector enorme la deja sin efecto.

1.6 El sistema operativo

Con lo anterior ya se puede decir qué es: el programa que gestiona los recursos del hardware y ofrece a las aplicaciones una interfaz uniforme para usarlos.

Como gestor de recursos	Como máquina extendida
reparte procesador, memoria, disco y dispositivos	esconde el hardware tras abstracciones
decide quién usa qué y cuándo	archivo, proceso, socket, memoria virtual
protege a unos usuarios de otros	la misma interfaz sobre hardware distinto

Las dos visiones son la misma cosa vista desde dos lados. La segunda es la que explica que un programa que lee un archivo funcione igual sobre un disco magnético, un SSD o un sistema de archivos remoto.

1.7 Utilidades del sistema

Alrededor del núcleo hay programas que no forman parte de él y sin los cuales el sistema no se puede usar:

Grupo	Ejemplos
Gestión de archivos	copiar, mover, listar, buscar
Intérprete de órdenes	la shell
Desarrollo	compilador, enlazador, depurador, `make`
Administración	usuarios, permisos, procesos, servicios
Red	transferencia, diagnóstico, acceso remoto

La distinción entre núcleo y utilidades no es cosmética: el núcleo se ejecuta en modo privilegiado y un fallo suyo tumba la máquina; una utilidad es un proceso de usuario más, y su fallo termina ese proceso y nada más.

1.8 Ejercicios

Ejercicio 1.8.1. ¿Por qué las instrucciones que acceden a puertos de entrada y salida son privilegiadas?

Solución 1.8.1. Porque un programa que escribiese directamente en el controlador del disco podría leer o sobrescribir datos de cualquier otro usuario, saltándose los permisos del sistema de archivos. Al reservarlas al modo núcleo, todo acceso pasa por el sistema operativo, que es quien comprueba los permisos.

Ejercicio 1.8.2. Un sistema sin temporizador ejecuta un proceso que entra en un bucle infinito sin llamadas al sistema. ¿Qué ocurre?

Solución 1.8.2. El sistema queda bloqueado. Sin interrupción de reloj, el sistema operativo solo recupera el control cuando el proceso hace una llamada al sistema o provoca una excepción, y ese bucle no hace ninguna de las dos cosas. Es exactamente el fallo de los sistemas con multitarea cooperativa.

Ejercicio 1.8.3. Un disco transfiere bloques de 4 KB y el procesador puede leer una palabra de 8 bytes por interrupción. ¿Cuántas interrupciones exige leer un bloque con E/S por interrupciones? ¿Y con DMA?

Solución 1.8.3. Con E/S por interrupciones, $4096/8 = 512$ interrupciones, cada una con su cambio de contexto y su rutina de servicio. Con DMA, una sola al terminar el bloque entero. Esa relación de 512 a 1 es la razón de que todo dispositivo de bloques moderno use DMA.

La descripción de los componentes y de los mecanismos de protección está en Stallings, 2018 y Prieto et al., 2006.

Capítulo 2

Introducción a los sistemas operativos

Tema 2 del programa. Qué componentes tiene un sistema operativo multiprogramado, cómo se le piden servicios y qué es exactamente un proceso.

2.1 Multiprogramación

En un sistema monoprogramado, mientras un programa espera al disco el procesador no hace nada. La **multiprogramación** mantiene varios programas cargados y conmuta a otro en cuanto uno se bloquea.

Modelo	Procesador ocioso	Cuándo cambia de programa
Monoprogramación	mucho	al terminar
Multiprogramación	poco	al bloquearse
Tiempo compartido	poco	al bloquearse o al agotar su rodaja

El **tiempo compartido** añade el temporizador del tema anterior: aunque un proceso no se bloquee nunca, pierde el procesador al agotar su cuanto. Eso es lo que hace que un sistema con muchos usuarios responda a todos.

La ganancia se calcula fácil. Si un proceso pasa una fracción p de su tiempo esperando E/S y hay n procesos independientes, el procesador está ocioso una fracción aproximada p^n:

$$\text{utilización} \approx 1 - p^n$$

Con $p = 0{,}8$, un solo proceso deja el procesador ocioso el 80 % del tiempo; cinco procesos lo bajan al 33 %, y diez al 11 %. El modelo es optimista —supone independencia— pero explica bien por qué la multiprogramación se impuso.

2.2 Componentes

Componente	De qué se ocupa
Gestión de procesos	crear, planificar, sincronizar y terminar
Gestión de memoria	asignar, proteger, memoria virtual
Sistema de archivos	organizar el almacenamiento persistente

Componente	De qué se ocupa
Gestión de E/S	controladores, planificación de disco, búferes
Red	protocolos y sockets
Seguridad	usuarios, permisos, autenticación
Intérprete de órdenes	interfaz con el usuario, fuera del núcleo

2.2.1 Cómo se organizan entre sí

Estructura	Idea	Coste y beneficio
Monolítica	todo el núcleo en un espacio de direcciones	rápida; un fallo tumba el sistema
Por capas	cada capa usa solo la inferior	ordenada; las capas cuestan rendimiento
Micronúcleo	el núcleo mínimo, el resto en modo usuario	robusta; el paso de mensajes cuesta
Híbrida	núcleo monolítico con módulos cargables	la opción práctica actual

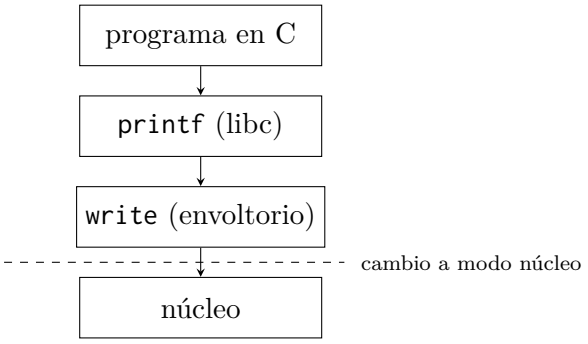
Linux es monolítico con módulos, y esa decisión explica dos cosas cotidianas: que un controlador se cargue sin reiniciar, y que un controlador con errores pueda bloquear la máquina entera, porque se ejecuta en modo núcleo.

2.3 Servicios: la API y la shell

Un programa pide servicios al sistema operativo de una única forma, la **llamada al sistema**, que es la instrucción especial que cambia a modo núcleo.

Categoría	Ejemplos en POSIX
Procesos	`fork`, `exec`, `wait`, `exit`
Archivos	`open`, `read`, `write`, `close`, `lseek`
Directorios	`mkdir`, `rmdir`, `link`, `unlink`
Información	`getpid`, `time`, `stat`
Comunicación	`pipe`, `socket`, `send`, `recv`
Protección	`chmod`, `chown`, `umask`

Lo que un programa en C escribe no es la llamada, sino la función de biblioteca que la envuelve:



La distinción importa al depurar: `printf` acumula en un búfer y puede no haber llegado al núcleo cuando el programa aborta, así que la última línea impresa no es necesariamente la última ejecutada. Con `write` no pasa, porque no hay búfer intermedio.

2.3.1 La shell

El intérprete de órdenes **no forma parte del núcleo**: es un proceso de usuario que lee una línea, la interpreta y pide al sistema que ejecute lo que corresponda.

Su ciclo, que es el mismo desde los años setenta:

1. Escribe el indicador y lee una línea.
2. La divide en palabras y expande comodines, variables y sustituciones.
3. Si la orden es interna —`cd`, `export`—, la ejecuta ella misma.
4. Si es externa, hace `fork`, y el hijo hace `exec` del programa.
5. Espera al hijo con `wait`, salvo que la orden terminase en `&`.

El paso 3 explica una duda recurrente: **`cd` tiene que ser interna**. Un proceso hijo que cambiase de directorio moriría al terminar y dejaría a la shell donde estaba, así que el cambio no serviría de nada.

2.4 Programas y procesos

Definición 2.1 (Proceso). Un programa en ejecución, junto con todo su estado: el contenido de su memoria, sus registros, sus archivos abiertos y la información que el sistema operativo mantiene sobre él.

Programa	Proceso
entidad pasiva, un archivo en disco	entidad activa, en ejecución
existe aunque nadie lo ejecute	existe mientras se ejecuta
uno	puede haber muchos del mismo programa

Tres ventanas de terminal abiertas son tres procesos y un solo programa. Cada una tiene su propio directorio actual, sus variables y su historial, porque el estado es del proceso y no del programa.

2.4.1 El bloque de control de proceso

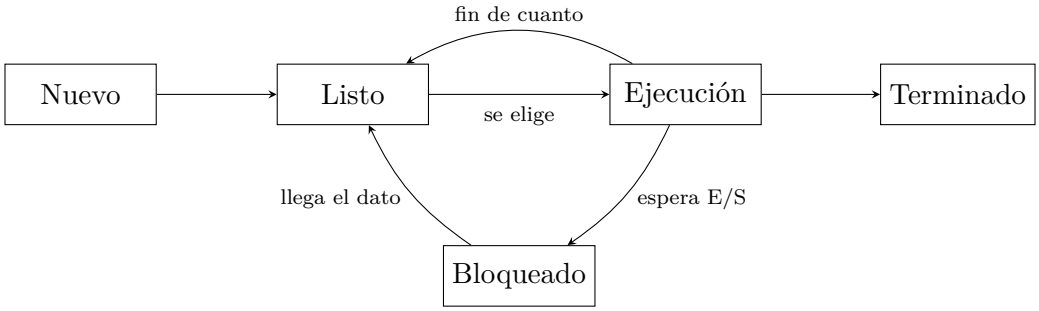
El sistema operativo mantiene por cada proceso una estructura con todo su estado:

Campo	Contenido
Identificador	PID, y PID del padre
Estado	listo, en ejecución, bloqueado, terminado
Contexto	contador de programa, registros, palabra de estado
Memoria	punteros a sus segmentos o a su tabla de páginas
Archivos	tabla de descriptores abiertos
Planificación	prioridad, tiempo consumido
Credenciales	usuario y grupo propietarios

El **cambio de contexto** consiste en guardar esa estructura para el proceso que sale y cargar la del

que entra. No hace trabajo útil, así que su coste es puro gasto, y por eso el cuanto de tiempo no puede ser demasiado pequeño.

2.4.2 Estados



Las dos transiciones que salen de **Ejecución** hacia atrás son las que distinguen los sistemas. La de «espera E/S» existe en cualquier sistema multiprogramado; la de «fin de cuanto» solo en los apropiativos, y es la que garantiza que ningún proceso monopolice el procesador.

2.4.3 Creación en POSIX

```c
pid_t pid = fork();          // duplica el proceso
if (pid == 0) {
  execlp("ls", "ls", "-l", NULL);   // el hijo se convierte en otro programa
  perror("exec");                   // solo se llega aquí si exec falla
  exit(1);
} else {
  int estado;
  waitpid(pid, &estado, 0);         // el padre espera
}
```

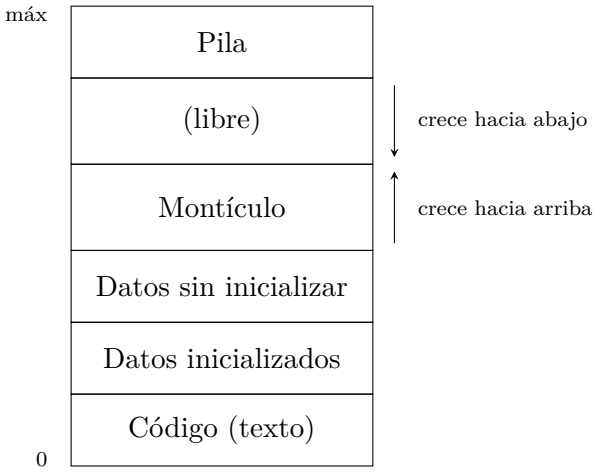
Tres cosas que este fragmento enseña y que se malentienden a menudo:

- **`fork` devuelve dos veces**, una en cada proceso: cero en el hijo y el PID del hijo en el padre. Es el único punto del lenguaje donde una función retorna dos veces, y de ahí que el `if` decida quién es quién.
- **`exec` no vuelve si tiene éxito**: sustituye la imagen del proceso. La línea siguiente solo se ejecuta si falló, y por eso ahí va el tratamiento del error.
- **El padre debe esperar al hijo.** Si no lo hace, el hijo terminado queda como **zombi**: ya no ejecuta nada, pero su entrada sigue en la tabla de procesos para que alguien lea su código de salida.

Nota 2.1 . Un proceso cuyo padre muere antes que él queda **huérfano** y lo adopta el proceso inicial del sistema, que sí llama a `wait`. Por eso los huérfanos no se acumulan y los zombis sí: el problema no es que el padre muera, es que viva sin recoger a sus hijos.

2.5 Modelos de memoria de un proceso

Un proceso ve un espacio de direcciones propio, dividido en zonas con propósitos distintos:



Zona	Qué contiene	Cuándo se reserva
Código	las instrucciones	al cargar; solo lectura
Datos inicializados	variables globales con valor	al cargar
Datos sin inicializar	globales sin valor, a cero	al cargar; no ocupan en el archivo
Montículo	lo que pide `malloc` o `new`	en ejecución, bajo demanda
Pila	marcos de llamada, locales, parámetros	en ejecución, por llamada

Que el código sea de solo lectura permite **compartirlo**: diez procesos del mismo programa comparten una sola copia en memoria física, y cada uno tiene sus datos.

Y que pila y montículo crezcan uno hacia el otro explica los dos desbordamientos clásicos: una recursión sin caso base agota la pila, y un `malloc` sin `free` repetido agota el montículo. En un sistema con memoria virtual no chocan entre sí, porque hay un hueco enorme sin asignar entre los dos, pero cada uno tiene su límite.

Ejemplo 2.1 . Una variable global sin inicializar no ocupa espacio en el archivo ejecutable, solo una anotación de cuántos bytes hay que poner a cero al cargar. Por eso declarar un vector global de un millón de enteros no engorda el binario, y declararlo con valores iniciales sí: cuatro megabytes de ceros escritos en el archivo.

2.6 Ejercicios

Ejercicio 2.6.1. ¿Cuántas líneas imprime este programa?

```
fork(); fork(); printf("hola\n");
```

Solución 2.6.1. Cuatro. El primer `fork` deja dos procesos y el segundo duplica cada uno, así que quedan $2^2 = 4$, y todos ejecutan el `printf`. Con n llamadas consecutivas serían 2^n.

Ejercicio 2.6.2. ¿Por qué `cd` no puede ser un programa externo?

Solución 2.6.2. Porque el directorio actual es un atributo del proceso. Un programa externo se

ejecuta en un proceso hijo, cambiaría su propio directorio y moriría al terminar, dejando a la shell donde estaba. Tiene que ser una orden interna que la propia shell ejecute sobre sí misma, con `chdir`.

Ejercicio 2.6.3. En un sistema donde los procesos pasan el 90 % del tiempo esperando E/S, ¿cuántos hacen falta para que el procesador esté ocupado más del 90 % del tiempo?

Solución 2.6.3. Hay que resolver $1 - 0{,}9^n > 0{,}9$, es decir $0{,}9^n < 0{,}1$, que da $n > \ln 0{,}1 / \ln 0{,}9 \approx 21{,}9$: veintidós procesos. El modelo supone independencia entre ellos, así que en la práctica hacen falta más, sobre todo si compiten por el mismo dispositivo.

El desarrollo de la multiprogramación y del modelo de procesos está en Stallings, 2018 y Carretero et al., 2021, y la interfaz POSIX en Johnson y Troan, 2005.

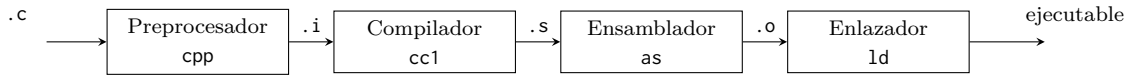
Capítulo 3

Compilación y enlazado de programas

Tema 3 del programa. Qué ocurre entre escribir un archivo `.c` y tener un proceso en ejecución, y cómo se automatiza.

3.1 Las cuatro etapas

`gcc programa.c -o programa` no es una orden, son cuatro encadenadas:



Etapa	Qué hace	Opción para pararse ahí
Preprocesado	expande `#include`, `#define` y condicionales	`gcc -E`
Compilación	traduce a ensamblador, optimiza	`gcc -S`
Ensamblado	traduce a código máquina en un objeto reubicable	`gcc -c`
Enlazado	resuelve símbolos y produce el ejecutable	ninguna: es la última

Poder pararse en cada etapa no es curiosidad académica: es la forma de localizar un error. Un mensaje raro sobre una macro se entiende mirando la salida de `gcc -E`, y un problema de enlazado no se arregla tocando el código fuente.

3.1.1 Preprocesado

Trabaja sobre texto y no entiende C. Esa es la causa de sus dos trampas clásicas:

```
#define CUADRADO(x) x * x
CUADRADO(2 + 3)      // se expande a 2 + 3 * 2 + 3, que es 11
```

Se corrige con paréntesis: `#define CUADRADO(x) ((x) * (x))`. Y aun así, `CUADRADO(i++)` incrementa dos veces, porque la macro duplica el argumento.

Las **guardas de inclusión** resuelven el otro problema, incluir dos veces la misma cabecera:

```
#ifndef PILA_H
#define PILA_H
```

```
/* ... */
#endif
```

3.1.2 Compilación

Traduce a ensamblador tras las fases habituales: análisis léxico, sintáctico, semántico, generación de código intermedio y optimización.

Nivel	Qué hace	Cuándo se usa
`-O0`	ninguna optimización	al depurar: el código se corresponde con las fuentes
`-O2`	el conjunto habitual	compilación de producción
`-O3`	añade las agresivas, como la vectorización	cuando se ha medido que compensa
`-Os`	optimiza el tamaño	sistemas empotrados

Y las opciones que más tiempo ahorran, que no son de optimización:

Opción	Qué aporta
`-Wall -Wextra`	avisos que delatan errores reales antes de ejecutar
`-g`	información de depuración para `gdb`
`-std=c99`	fija el estándar y evita depender del compilador
`-fsanitize=address`	detecta accesos fuera de rango y fugas en ejecución

Nota 3.1 . Compilar con `-O2` y depurar con `gdb` a la vez confunde: el optimizador reordena instrucciones y elimina variables, así que el depurador salta líneas y muestra valores «optimizados». Para depurar, `-O0 -g`.

3.1.3 Ensamblado y enlazado

El ensamblador produce un **objeto reubicable**, que ya es código máquina pero con las direcciones sin fijar y una tabla de símbolos con lo que define y lo que necesita.

El enlazador hace dos cosas:

- **Resolución de símbolos**: casa cada referencia con su definición.
- **Reubicación**: asigna direcciones definitivas y ajusta las referencias.

Los dos errores típicos son de esta etapa y se distinguen bien:

Mensaje	Causa
`undefined reference to 'f'`	se usa `f` y nadie la define, o falta la biblioteca
`multiple definition of 'v'`	dos objetos definen el mismo símbolo global

El segundo suele venir de definir una variable en una cabecera incluida dos veces. Se corrige declarándola `extern` en la cabecera y definiéndola en un único `.c`.

3.2 Ciclo de vida y modelo de memoria de un proceso

Cuando el ejecutable arranca, el sistema:

1. Crea el proceso con `fork` y ejecuta `exec`.
2. Lee la cabecera del ejecutable y descubre sus secciones.
3. Mapea el código y los datos en el espacio de direcciones.
4. Reserva la pila y coloca argumentos y variables de entorno.
5. Enlaza las bibliotecas dinámicas que falten.
6. Salta al punto de entrada, que llama a `main`.

Las secciones del ejecutable se corresponden con las zonas del tema anterior:

Sección	En memoria	Contiene
`.text`	código, solo lectura	instrucciones
`.rodata`	solo lectura	constantes y cadenas literales
`.data`	lectura y escritura	globales inicializadas
`.bss`	lectura y escritura	globales a cero; no ocupa en el archivo

Ejemplo 3.1 . Que `.rodata` sea de solo lectura explica un fallo frecuente:

```
char *s = "hola"; s[0] = 'H';
```

compila sin avisos y aborta al ejecutarse, porque el literal vive en `.rodata`. Con `char s[] = "hola";` el literal se copia a la pila y la escritura es válida.

3.2.1 Dónde vive cada variable

Declaración	Zona	Vida
Global	`.data` o `.bss`	todo el programa
`static` dentro de función	`.data` o `.bss`	todo el programa, visible solo ahí
Local	pila	la llamada
`malloc` / `new`	montículo	hasta `free` / `delete`

De aquí sale el error que más cuesta depurar: **devolver un puntero a una variable local**. El marco de pila desaparece al volver, y el puntero apunta a memoria que la llamada siguiente reutilizará. A veces funciona, que es lo peor que puede pasar.

3.3 Bibliotecas

Una biblioteca agrupa objetos ya compilados para reutilizarlos. Hay dos clases y la diferencia es visible en todo lo demás.

	Estática (.a)	Dinámica (.so)
Cuándo se une	al enlazar	al cargar o al usarse
Tamaño del ejecutable	mayor: lleva el código dentro	menor: solo la referencia
Memoria con n procesos	n copias	una compartida
Actualizar la biblioteca	hay que reenlazar	basta sustituir el archivo
Dependencias en ejecución	ninguna	tiene que estar instalada

```
# estática
gcc -c pila.c && ar rcs libpila.a pila.o
gcc main.c -L. -lpila -o programa

# dinámica
gcc -fPIC -c pila.c && gcc -shared -o libpila.so pila.o
gcc main.c -L. -lpila -o programa
export LD_LIBRARY_PATH=.        # si no, no la encuentra al ejecutar
```

El `-fPIC` no es opcional en la dinámica: genera **código independiente de la posición**, que funciona esté donde esté cargada. Sin él, la biblioteca solo valdría en una dirección fija, y dos bibliotecas querrían la misma.

Y la última línea explica el error más común con bibliotecas dinámicas: enlazan bien y al ejecutar dicen que no encuentran el `.so`. El enlazador y el cargador buscan en sitios distintos.

3.4 Automatización con `make`

Recompilar todo cada vez es lento, y recompilar a mano solo lo que cambió es propenso a error. `make` resuelve las dos cosas: **reconstruye lo que está desactualizado**, comparando fechas de modificación.

```
CC      = gcc
CFLAGS  = -Wall -Wextra -g -std=c99
OBJ     = main.o pila.o cola.o

programa: $(OBJ)
    $(CC) $(OBJ) -o $@

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJ) programa

.PHONY: clean
```

Elemento	Qué es
Objetivo	lo que se quiere construir
Dependencias	de qué depende
Receta	las órdenes, **con tabulador**, nunca espacios
`$@`	el objetivo

Elemento	Qué es
`$<`	la primera dependencia
`$^`	todas las dependencias
`.PHONY`	objetivos que no son archivos

El error de las dependencias incompletas merece atención porque no da ningún mensaje. Si `pila.o` depende de `pila.h` y la regla no lo dice, tocar la cabecera no provoca recompilación, y el objeto queda desfasado. El programa enlaza y se comporta de forma incoherente. Se resuelve generando las dependencias automáticamente:

```
-include $(OBJ:.o=.d)
CFLAGS += -MMD -MP
```

Nota 3.2 . `make` decide por **fecha de modificación**, no por contenido. Un archivo con fecha futura —por un reloj mal puesto o una copia que preserva marcas de tiempo— hace que su objetivo parezca siempre actualizado y no se reconstruya nunca. Ante un comportamiento inexplicable, `make clean`.

3.5 Ejercicios

Ejercicio 3.5.1. Un programa enlaza correctamente y al ejecutarlo da `error while loading shared libraries`. ¿Qué ocurre?

Solución 3.5.1. La biblioteca dinámica estaba disponible al enlazar, en la ruta indicada con `-L`, pero el cargador no la encuentra en tiempo de ejecución porque busca en otras rutas. Se resuelve instalándola en un directorio del sistema, añadiendo su ruta a `LD_LIBRARY_PATH`, o enlazando con `-Wl,-rpath`.

Ejercicio 3.5.2. ¿Por qué `#define DOBLE(x) x * 2` da un resultado inesperado en `DOBLE(3 + 1)`?

Solución 3.5.2. El preprocesador sustituye texto sin entender precedencia: la expansión es `3 + 1 * 2`, que vale 5 y no 8. Se corrige con `#define DOBLE(x) ((x) * 2)`. El paréntesis exterior hace falta además para que la macro se comporte bien dentro de una expresión mayor.

Ejercicio 3.5.3. Un Makefile no declara que `main.o` depende de `tipos.h`. Se modifica la cabecera cambiando el tamaño de una estructura y se ejecuta `make`. ¿Qué pasa?

Solución 3.5.3. `make` no ve motivo para recompilar `main.o`, que queda con la definición vieja mientras el resto usa la nueva. El programa enlaza y accede a campos en posiciones distintas según el objeto, con corrupción silenciosa. Es el peor tipo de error: no hay mensaje. Se evita generando las dependencias con `-MMD -MP`.

El proceso de compilación y enlazado está descrito en Gough, 2005, la automatización con `make` en Mecklenburg, 2004, y el desarrollo de aplicaciones sobre Linux en Johnson y Troan, 2005 y Matthew y Stones, 2008.

Capítulo 4

Sistemas de archivos. Introducción a las bases de datos

Tema 4 del programa. Cómo se organiza la información persistente, primero en archivos y después en bases de datos, y por qué hacen falta las dos cosas.

4.1 Archivos y directorios

Definición 4.1 (Archivo). Unidad lógica de almacenamiento persistente: una secuencia de bytes con nombre, independiente del soporte físico donde resida.

La definición es corta y esconde el trabajo entero del sistema operativo. Un archivo está repartido en bloques dispersos por el disco, y el sistema presenta esos bloques como una secuencia continua a la que se accede por nombre.

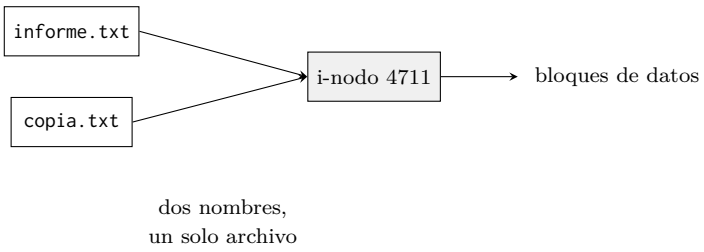
Atributo	Para qué
Nombre	identificarlo dentro de su directorio
Tipo	interpretar su contenido
Tamaño	saber cuánto ocupa
Fechas	creación, último acceso, última modificación
Propietario y permisos	control de acceso
Ubicación	dónde están sus bloques

Todos menos el nombre viven en la estructura de control del archivo, el **i-nodo** en los sistemas de tipo UNIX. El nombre está en el directorio, y esa separación tiene una consecuencia visible: un mismo archivo puede tener varios nombres.

4.1.1 Directorios

Un directorio es un archivo cuyo contenido es una tabla que asocia nombres con i-nodos. La estructura habitual es un **árbol**, con dos aditamentos que lo convierten en grafo:

Enlace	Qué es	Si se borra el original
Duro	otra entrada de directorio al mismo i-nodo	el archivo sigue: el i-nodo cuenta referencias
Simbólico	un archivo que contiene una ruta	el enlace queda roto



De ahí que borrar en UNIX se llame `unlink`: no borra el archivo, borra el nombre. El archivo desaparece cuando no queda ningún nombre **y** ningún proceso lo tiene abierto. Un programa puede abrir un archivo, borrarlo y seguir usándolo, y así se crean los temporales que se limpian solos.

4.1.2 Permisos

Tres clases de usuario y tres permisos:

	Leer	Escribir	Ejecutar
Propietario	`r`	`w`	`x`
Grupo	`r`	`w`	`x`
Otros	`r`	`w`	`x`

En notación octal, `rwxr-xr--` es 754. Sobre un **directorio** los tres permisos significan otra cosa, y es la fuente de la mitad de las confusiones:

Permiso sobre un directorio	Qué permite
`r`	listar los nombres que contiene
`w`	crear y borrar entradas dentro
`x`	atravesarlo para llegar a lo que hay debajo

Dos consecuencias que no son intuitivas: con `x` y sin `r` se puede acceder a un archivo cuyo nombre se conozca, pero no listar el directorio; y **borrar un archivo depende del permiso de escritura del directorio, no del archivo**, porque lo que se modifica es la tabla de nombres.

4.2 Organización de la información

4.2.1 Métodos de acceso

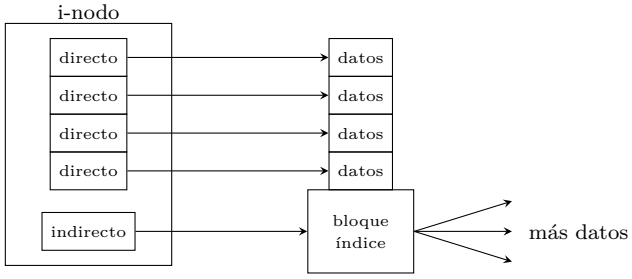
Método	Cómo se lee	Ejemplo
Secuencial	de principio a fin	registro de sucesos, cinta
Directo	saltando a una posición	archivo de registros de tamaño fijo
Indexado	por un índice que apunta a la posición	base de datos

El acceso directo exige registros de **tamaño fijo**: para saltar al registro i hay que poder calcular su desplazamiento como $i \times$ tamaño. Con registros variables no hay fórmula y hace falta un índice.

4.2.2 Asignación de espacio en disco

Estrategia	Cómo	Ventaja	Problema
Contigua	bloques consecutivos	acceso directo trivial, muy rápida	fragmentación externa; crecer es imposible
Enlazada	cada bloque apunta al siguiente	sin fragmentación, crece libre	no hay acceso directo
Indexada	un bloque índice con las direcciones	acceso directo y crecimiento	el índice ocupa y se puede quedar corto

Los sistemas reales usan la indexada con **indirección**: el i-nodo guarda unas pocas direcciones directas y luego una indirecta simple, una doble y una triple. Con eso, los archivos pequeños se leen con un solo acceso y los enormes siguen siendo representables.



4.2.3 Gestión del espacio libre

Método	Cómo	Coste
Mapa de bits	un bit por bloque	1/8 de byte por bloque; fácil buscar huecos contiguos
Lista enlazada	cada bloque libre apunta al siguiente	sin coste de espacio; buscar contiguos es caro
Agrupación	bloques con listas de direcciones libres	intermedio

Con bloques de 4 KB, el mapa de bits de un disco de 1 TB ocupa 32 MB. Es asumible, y por eso es lo habitual.

4.2.4 Fiabilidad

Un corte de corriente a mitad de una operación deja el sistema de archivos inconsistente: un bloque asignado a dos archivos, o marcado como ocupado sin pertenecer a ninguno. Las dos respuestas:

Técnica	Idea
Comprobación al arrancar	recorrer todo el sistema y reparar
Registro por diario (*journaling*)	anotar la operación antes de hacerla y borrar la anotación al acabar

La primera tarda en proporción al tamaño del disco, y con discos grandes eso son horas. La segunda solo tiene que revisar el diario, y por eso la usan todos los sistemas actuales.

4.3 Bases de datos

Un archivo basta mientras los datos sean pocos, los use un solo programa y no importe repetir información. Cuando eso deja de ser cierto aparecen cuatro problemas que el sistema de archivos no resuelve.

Problema con archivos	Qué produce
Redundancia	el mismo dato en varios archivos
Inconsistencia	se actualiza una copia y no las otras
Dependencia del formato	cambiar la estructura obliga a tocar todos los programas
Acceso concurrente	dos programas escribiendo a la vez corrompen los datos

Definición 4.2 (Base de datos). Colección estructurada de datos relacionados, con una descripción de su propia estructura, almacenada de forma que varios usuarios y aplicaciones puedan compartirla de manera controlada.

La parte que suele pasarse por alto es «con una descripción de su propia estructura»: la base de datos guarda su esquema dentro. Eso es lo que permite que una consulta que nadie previó funcione sin tocar ningún programa.

4.3.1 El gestor

El **sistema gestor de bases de datos** es el software que se interpone entre las aplicaciones y los datos.

Función	Qué aporta
Definición de datos	describir el esquema
Manipulación	consultar, insertar, modificar, borrar
Control de concurrencia	varios usuarios a la vez sin corromper nada
Recuperación	volver a un estado consistente tras un fallo
Seguridad	quién puede ver y modificar qué
Integridad	reglas que los datos deben cumplir siempre

4.3.2 El modelo relacional

Los datos se organizan en **tablas**. Cada fila es un registro y cada columna un atributo con su dominio.

Concepto	Qué es
Tabla	conjunto de filas con la misma estructura
Clave primaria	atributo que identifica cada fila de forma única

Concepto	Qué es
Clave ajena	atributo que referencia la clave primaria de otra tabla
Integridad referencial	toda clave ajena apunta a una fila que existe

```sql
CREATE TABLE alumno (
  dni      CHAR(9) PRIMARY KEY,
  nombre   VARCHAR(60) NOT NULL,
  curso    INTEGER CHECK (curso BETWEEN 1 AND 5)
);

CREATE TABLE matricula (
  dni       CHAR(9) REFERENCES alumno(dni),
  asignatura VARCHAR(10),
  PRIMARY KEY (dni, asignatura)
);

SELECT a.nombre, COUNT(*) AS asignaturas
FROM alumno a JOIN matricula m ON a.dni = m.dni
GROUP BY a.nombre
HAVING COUNT(*) > 3;
```

Lo que hace fuerte al modelo es que SQL es **declarativo**: la consulta dice qué se quiere, no cómo obtenerlo. El gestor decide el plan de ejecución, elige los índices y puede cambiar de estrategia si los datos cambian, sin que la consulta se toque.

4.3.3 Transacciones

Una transacción es una secuencia de operaciones que se ejecuta como una unidad. Sus cuatro garantías:

Propiedad	Qué asegura
Atomicidad	o se hacen todas las operaciones o ninguna
Consistencia	de un estado válido a otro estado válido
Aislamiento	el resultado es como si se ejecutaran una tras otra
Durabilidad	lo confirmado sobrevive a un fallo

Ejemplo 4.1 . Una transferencia bancaria resta de una cuenta y suma en otra. Si el sistema falla entre las dos operaciones, la atomicidad garantiza que se deshace la primera: el dinero no desaparece. Con archivos habría que programar esa garantía a mano, y acertar en todos los casos de fallo posibles.

La durabilidad se apoya en la misma técnica que el sistema de archivos con diario: **anotar antes de hacer**. La atomicidad, en poder deshacer con lo anotado.

4.3.4 Cuándo archivo y cuándo base de datos

Situación	Opción
Configuración de un programa	archivo de texto
Registro de sucesos que solo se añade	archivo
Datos con relaciones y consultas variadas	base de datos
Varios usuarios escribiendo a la vez	base de datos
Datos que deben sobrevivir a fallos a mitad de operación	base de datos
Un volumen enorme sin estructura, de solo lectura	archivo, a veces con índice aparte

4.4 Ejercicios

Ejercicio 4.4.1. Un directorio tiene permisos `d--x------` para el propietario. ¿Puede listar su contenido? ¿Puede leer un archivo de dentro?

Solución 4.4.1. Listar no, porque falta `r` sobre el directorio. Leer un archivo sí, siempre que se conozca su nombre exacto y el archivo tenga permiso de lectura: el `x` autoriza a atravesar el directorio. Es la configuración típica de los directorios personales compartidos, donde se quiere dar acceso sin permitir fisgonear.

Ejercicio 4.4.2. Un proceso abre un archivo, otro lo borra con `rm` y el primero sigue leyendo. ¿Qué ocurre?

Solución 4.4.2. Sigue leyendo sin problema. `rm` elimina la entrada del directorio y decrementa el contador de enlaces del i-nodo, pero el archivo solo se libera cuando ese contador llega a cero **y** ningún proceso lo tiene abierto. El espacio se recupera al cerrar. Por eso borrar un archivo de registro grande no libera espacio mientras el proceso que escribe en él siga vivo.

Ejercicio 4.4.3. Dos programas actualizan a la vez el saldo de la misma cuenta guardado en un archivo. ¿Qué puede salir mal y cómo lo evita una base de datos?

Solución 4.4.3. Los dos leen el mismo saldo inicial, cada uno le aplica su cambio y escribe: la segunda escritura pisa la primera y una de las dos operaciones se pierde sin dejar rastro. Un gestor lo evita con control de concurrencia —bloqueos o versiones— de modo que el resultado equivalga a ejecutarlas en algún orden, que es la propiedad de aislamiento.

La organización de los sistemas de archivos está en Stallings, 2018 y Carretero et al., 2021, y su manejo desde programa en Johnson y Troan, 2005.

Capítulo 5

Generación y depuración de aplicaciones

Tema 5 del programa. Qué es una plataforma, cómo se escribe software que no dependa de una, qué aportan los entornos de desarrollo y cómo se depura de verdad.

5.1 Plataforma

Definición 5.1 (Plataforma). Combinación de arquitectura de procesador, sistema operativo y bibliotecas del sistema sobre la que un programa se ejecuta.

Un ejecutable compilado para una plataforma no funciona en otra, y las razones son tres, independientes entre sí:

Nivel	Qué cambia
Arquitectura	el juego de instrucciones: x86-64, ARM, RISC-V
Sistema operativo	las llamadas al sistema y el formato del ejecutable
Bibliotecas	las funciones disponibles y sus versiones

Y dentro de la arquitectura, dos detalles que rompen programas aparentemente portables:

- **El orden de los bytes.** En *little-endian* el byte menos significativo va primero y en *big-endian* al revés. Un archivo binario escrito en una máquina y leído en la otra da valores absurdos.
- **El tamaño de los tipos.** `int` es de 32 bits casi siempre, pero `long` es de 32 en Windows de 64 bits y de 64 en Linux. Un programa que dé por hecho el tamaño falla al cambiar de sitio.

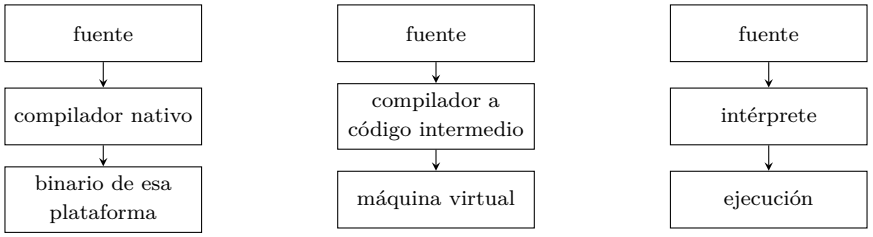
Se resuelven con `htonl` y compañía para el orden, y con `int32_t`, `uint64_t` y los demás tipos de `<stdint.h>` para los tamaños.

5.2 Software independiente de plataforma

Tres estrategias distintas, con precios distintos:

Estrategia	Cómo	Coste
Código portable recompilado	fuentes que compilan en varios sistemas	hay que compilar en cada uno

Estrategia	Cómo	Coste
Máquina virtual	se compila a un código intermedio que interpreta una VM	la VM tiene que estar instalada
Interpretación	el código fuente se ejecuta directamente	más lento; el intérprete debe estar



Los compiladores modernos de máquina virtual añaden una cuarta vía: la compilación **en tiempo de ejecución**, que traduce a código nativo los fragmentos que más se ejecutan. Da casi el rendimiento del nativo conservando la portabilidad, a cambio de un arranque más lento.

Para el software recompilable, las herramientas que absorben las diferencias:

Herramienta	Qué resuelve
`autoconf` / `automake`	detecta qué ofrece el sistema y configura la compilación
CMake	describe el proyecto una vez y genera el sistema de construcción de cada plataforma
Compilación condicional	`#ifdef _WIN32` para lo que no tiene equivalente

La última es la de peor calidad y a veces la única. Conviene concentrarla en unos pocos archivos de adaptación en vez de esparcirla por todo el código.

5.3 Entornos y marcos de desarrollo

Un **entorno de desarrollo integrado** reúne editor, compilador, depurador y sistema de construcción. No añade capacidades nuevas: hace más cómodo el ciclo de escribir, compilar, ejecutar y depurar. Conviene saber qué hay debajo, porque cuando algo falla lo que hay que leer es el error del compilador, no el del entorno.

Un **marco de trabajo** es distinto de una biblioteca, y la diferencia se resume en quién llama a quién:

	Biblioteca	Marco
Quién llama	el programa llama a la biblioteca	el marco llama al programa
Estructura	la decide el programador	la impone el marco
Coste de cambiar	bajo	alto

Por eso adoptar un marco es una decisión de arquitectura y usar una biblioteca no lo es. El programa se escribe **dentro** del marco, rellenando los huecos que deja.

Y una pieza que casi siempre acompaña: el **control de versiones**. Guarda la historia de los cambios, permite volver atrás y hace posible que varias personas trabajen a la vez. Es tan parte del entorno de desarrollo como el compilador.

5.4 Depuración

Depurar es localizar la causa de un comportamiento incorrecto. Que sea un método y no una intuición es lo que separa media hora de tres días.

5.4.1 El método

1. **Reproducir el fallo** de forma fiable. Un error que aparece una vez de cada veinte no se puede depurar; primero hay que encontrar la condición que lo dispara.
2. **Reducir el caso**: la entrada más pequeña que lo provoca.
3. **Formular una hipótesis** concreta y falsable sobre la causa.
4. **Diseñar la comprobación** que la confirma o la descarta.
5. **Corregir** y comprobar que el fallo desaparece y no aparece otro.

Nota 5.1 . El paso 3 es el que se salta casi siempre. Cambiar cosas hasta que deje de fallar no es depurar: si el fallo desaparece sin saber por qué, no hay ninguna garantía de que esté arreglado, y con frecuencia solo se ha ocultado.

5.4.2 Técnicas

Técnica	Cuándo sirve	Límite
Mensajes de traza	primera aproximación, errores reproducibles	ensucia el código y cambia el tiempo de ejecución
Depurador simbólico	inspeccionar estado y ejecutar paso a paso	requiere reproducir el fallo bajo el depurador
Aserciones	detectar el error donde se produce	hay que escribirlas antes
Análisis estático	encontrar errores sin ejecutar	avisa de cosas que no lo son
Analizadores dinámicos	fallos de memoria y de concurrencia	ralentizan mucho la ejecución
Volcado de memoria	fallos que no se pueden reproducir	solo da el estado final

Sobre los mensajes de traza hay un detalle práctico: **la salida estándar está almacenada en un búfer**, así que si el programa aborta, las últimas líneas pueden no haberse escrito. La traza engaña señalando un punto anterior al real. Se evita escribiendo por la salida de error, que no tiene búfer, o vaciando con `fflush`.

5.4.3 El depurador

```
gcc -g -O0 programa.c -o programa
gdb ./programa
```

Orden	Qué hace
`break f`	detiene al entrar en la función `f`
`break fichero.c:42 if i > 100`	punto de ruptura condicional
`run`	arranca
`next` / `step`	siguiente línea, sin entrar / entrando en la llamada
`print expr`	evalúa e imprime
`backtrace`	pila de llamadas
`watch v`	detiene cuando cambia `v`
`finish`	ejecuta hasta salir de la función

Las dos órdenes que más rinden son las menos usadas. **`backtrace` tras un fallo** dice la secuencia de llamadas que llevó ahí, que suele ser toda la información necesaria. Y **`watch`** resuelve el caso difícil: una variable que se corrompe sin saber quién la escribe. Poner un punto de observación y dejar correr señala la línea exacta.

5.4.4 Errores de memoria

Son los más frecuentes en C y C++ y los que peor se manifiestan, porque el síntoma aparece lejos de la causa.

Error	Qué produce
Acceso fuera de rango	corrupción de datos vecinos, o violación de segmento
Uso después de liberar	valores incoherentes
Doble liberación	corrupción de las estructuras del montículo
Fuga de memoria	consumo creciente hasta agotarla
Lectura de memoria sin inicializar	comportamiento distinto en cada ejecución

Dos herramientas los encuentran señalando la línea:

```
gcc -fsanitize=address -g programa.c -o programa && ./programa
valgrind --leak-check=full ./programa
```

El **sanitizador** instrumenta el programa al compilar y es rápido; **valgrind** ejecuta sobre una máquina virtual y no exige recompilar, pero ralentiza entre diez y cincuenta veces. Para desarrollo diario conviene el primero.

Ejemplo 5.1 . Un programa escribe una posición más allá del final de un vector reservado con `malloc` y funciona durante meses, porque esa posición cae en espacio que el gestor de memoria tenía de sobra. Al añadir una variable, el reparto cambia y el programa empieza a fallar en un sitio que no se ha tocado. El sanitizador detecta la escritura original desde la primera ejecución.

5.4.5 Aserciones

```c
#include <assert.h>
void quitar(Pila* p) {
  assert(p != NULL && p->n > 0);   // precondición
  --p->n;
}
```

Una aserción documenta y comprueba a la vez. Su valor es que **falla donde está el error**, no donde se manifiesta.

Dos reglas al usarlas:

- Se compilan fuera con `-DNDEBUG`, así que **no deben tener efectos laterales**. `assert(++i > 0)` deja de incrementar en la versión de producción.
- Comprueban errores de programación, no de entrada. Que un archivo no exista no es un fallo del programa y no se trata con una aserción, se trata con un error.

5.5 Del error al arreglo

Un resumen de qué usar según el síntoma:

Síntoma	Primera herramienta
Violación de segmento	`gdb` con `backtrace`, o el sanitizador
Resultado incorrecto pero sin fallo	puntos de ruptura y `print`
Consumo de memoria creciente	`valgrind --leak-check=full`
Falla en producción y no en desarrollo	comparar versiones, opciones y datos de entrada
Falla a veces, con los mismos datos	memoria sin inicializar, o concurrencia
Falla al optimizar y no sin optimizar	comportamiento indefinido en el código

Las dos últimas filas son las importantes. Un programa que solo falla con `-O2` casi siempre tiene **comportamiento indefinido**: el optimizador tiene derecho a suponer que eso no ocurre, y transforma el código en consecuencia. La conclusión correcta no es «el optimizador tiene un fallo», es «hay un error latente que ahora se ve».

5.6 Ejercicios

Ejercicio 5.6.1. Un programa funciona compilado con `-O0` y falla con `-O2`. ¿Qué conclusión es la razonable?

Solución 5.6.1. Que hay comportamiento indefinido en el código: acceso fuera de rango, lectura de una variable sin inicializar, desbordamiento de enteros con signo o un puntero mal usado. Con `-O0` el reparto de memoria y el orden de las operaciones lo esconden; el optimizador supone que no ocurre y reordena. La herramienta es el sanitizador, no bajar el nivel de optimización.

Ejercicio 5.6.2. ¿Por qué una aserción no debe llevar efectos laterales?

Solución 5.6.2. Porque `-DNDEBUG` las elimina por completo del código compilado. Si la condición modificaba algo, esa modificación desaparece y el programa se comporta de forma distinta en

producción y en desarrollo, que es el peor escenario posible: el fallo solo aparece donde no se puede depurar.

Ejercicio 5.6.3. Un servicio consume cada vez más memoria y acaba agotándola tras varios días. ¿Cómo se localiza la causa?

Solución 5.6.3. Con `valgrind --leak-check=full` sobre una ejecución que reproduzca la carga, que da la pila de llamadas de cada reserva no liberada. Si el servicio no se puede detener, sirve muestrear el consumo por zonas y correlacionarlo con la actividad para acotar el módulo. Conviene descartar antes que no sea crecimiento legítimo, como una caché sin límite: no toda memoria que crece es una fuga.

Las técnicas de depuración y las herramientas están descritas en Stallman et al., 2003 y Nethercote et al., 2008, y el desarrollo de aplicaciones sobre GNU/Linux en Matthew y Stones, 2008 y Gough, 2005.

Capítulo 6

Temario práctico

Las tres prácticas del programa, todas sobre un sistema de tipo UNIX.

6.1 Práctica 1. Órdenes básicas e intérprete de órdenes

6.1.1 Navegación y archivos

Orden	Qué hace
`pwd`, `cd`, `ls -la`	dónde se está, moverse, listar con detalle
`cp`, `mv`, `rm`, `mkdir`, `rmdir`	copiar, mover, borrar, crear directorios
`ln`, `ln -s`	enlace duro y enlace simbólico
`find . -name '*.c'`	buscar por nombre en el árbol
`du -sh`, `df -h`	espacio ocupado y espacio libre

6.1.2 Contenido

Orden	Qué hace
`cat`, `less`, `head`, `tail`	ver un archivo entero, paginado, el principio, el final
`grep -n patrón archivo`	líneas que casan, con su número
`wc -l`	contar líneas
`sort`, `uniq -c`	ordenar y contar repeticiones
`cut -d: -f1`	extraer campos
`sed`, `awk`	sustituir y procesar por campos

`uniq` solo agrupa líneas **consecutivas** iguales, así que casi siempre va detrás de `sort`. Es el error más frecuente de la práctica y no da ningún aviso: devuelve un resultado plausible y equivocado.

6.1.3 Permisos y procesos

Orden	Qué hace
`chmod 754 f`, `chmod u+x f`	permisos en octal o simbólico

Orden	Qué hace
`chown`, `chgrp`	propietario y grupo
`umask`	permisos que se quitan al crear
`ps aux`, `top`	procesos en el sistema
`kill -TERM pid`, `kill -9 pid`	pedir que termine, o forzarlo
`jobs`, `fg`, `bg`, `&`	control de trabajos de la shell

La diferencia entre `-TERM` y `-9` importa: el primero es una petición que el proceso puede atender para cerrar archivos y guardar estado; el segundo lo mata sin posibilidad de reaccionar. **Se prueba siempre el primero.**

6.1.4 Redirección y tuberías

Construcción	Qué hace
`> archivo`	redirige la salida estándar, truncando
`>> archivo`	redirige añadiendo
`2> archivo`	redirige la salida de error
`&> archivo`	redirige las dos
`< archivo`	toma la entrada estándar del archivo
`orden1 | orden2`	la salida de la primera es la entrada de la segunda

Las tuberías son la idea central del sistema: **programas pequeños que hacen una cosa y se combinan**.

```
# las diez palabras más frecuentes de un texto
tr -cs 'A-Za-z' '\n' < texto.txt | tr 'A-Z' 'a-z' | sort | uniq -c | sort -rn | head
```

Esa línea no necesita ningún programa nuevo, y resuelve en un segundo lo que en C serían cincuenta líneas. Es lo que la práctica pretende que se vea.

6.1.5 Expansión

Símbolo	Qué casa
`*`	cualquier cadena, incluida la vacía
`?`	un carácter
`[abc]`, `[a-z]`	un carácter del conjunto
`{a,b}`	expande a cada alternativa

La expansión la hace la shell, no el programa. `ls *.c` recibe ya la lista de nombres. Por eso `grep patrón *` falla si un nombre empieza por guion, y por eso hay que entrecomillar cuando se quiere pasar el asterisco literal.

6.2 Práctica 2. Construcción de guiones

Un guion automatiza lo que se haría a mano. Empieza por la línea que dice qué intérprete lo ejecuta:

```
#!/bin/bash
set -euo pipefail        # aborta al primer error, con variable sin definir y en tuberías
```

Esa segunda línea evita la mitad de los fallos de la práctica: sin ella, un guion sigue ejecutando después de que una orden falle, y borra o sobrescribe basándose en un resultado que no existe.

6.2.1 Variables

```
nombre="informe"
echo "Procesando ${nombre}.txt"
ruta=$(pwd)                        # sustitución de orden
```

Regla	Por qué
Sin espacios alrededor del `=`	`a = 1` intenta ejecutar el programa `a`
Comillas dobles al usar: `"$v"`	sin ellas, un valor con espacios se parte en varias palabras
`${v}` cuando sigue texto	`$nombreX` busca la variable `nombreX`
`"${@}"` para los argumentos	conserva los que llevan espacios

Las variables especiales: `$0` el nombre del guion, `$1` a `$9` los argumentos, `$#` cuántos hay, `$?` el código de salida de la última orden, `$$` el PID.

6.2.2 Condicionales y bucles

```
if [ -f "$archivo" ]; then
  echo "existe"
elif [ -d "$archivo" ]; then
  echo "es un directorio"
else
  echo "no existe" >&2
  exit 1
fi

for f in *.txt; do
  wc -l "$f"
done

while read -r linea; do
  echo "leída: $linea"
done < entrada.txt
```

Comprobación	Cierta si
`-f f`	existe y es archivo regular
`-d d`	existe y es directorio
`-r f`, `-w f`, `-x f`	hay permiso de lectura, escritura, ejecución
`-z "$s"`	la cadena está vacía
`"$a" = "$b"`	cadenas iguales
`$a -eq $b`, `-lt`, `-gt`	comparación numérica

Comparar cadenas con `-eq` es un error frecuente: `-eq` es numérico y con cadenas da un error o un resultado falso. Para cadenas, `=`.

6.2.3 Funciones y código de salida

```
uso() {
  echo "uso: $0 <directorio>" >&2
  exit 1
}

[ $# -eq 1 ] || uso
```

Un guion devuelve un código de salida: **cero es éxito** y cualquier otro es error. Es lo que permite encadenar con `&&` y `||`, y lo que hace que el guion sea utilizable desde otro.

6.2.4 Guion de ejemplo completo

```
#!/bin/bash
# Cuenta las lineas de codigo de los .c de un directorio, sin comentarios ni vacias.
set -euo pipefail

[ $# -eq 1 ] || { echo "uso: $0 <directorio>" >&2; exit 1; }
[ -d "$1" ]  || { echo "$1 no es un directorio" >&2; exit 1; }

total=0
while IFS= read -r -d '' f; do
  n=$(grep -vcE '^\s*(//.*)?$' "$f")
  printf '%6d %s\n' "$n" "$f"
  total=$((total + n))
done < <(find "$1" -name '*.c' -print0)

printf '%6d  TOTAL\n' "$total"
```

Dos detalles que la práctica pide entender:

- **`-print0` con `read -d ''`** trata bien los nombres con espacios y saltos de línea. Recorrer la salida de `find` con un `for` normal se rompe con el primer nombre que lleve un espacio.
- **La sustitución de proceso `< <(...)`** en vez de una tubería es necesaria para que el `while` se ejecute en la shell actual. Con `find ... | while ...` el bucle corre en un subproceso y `total` vuelve a cero al salir.

6.3 Práctica 3. Compilación de programas

6.3.1 Compilar por etapas

```
gcc -E prog.c -o prog.i        # preprocesado
gcc -S prog.i -o prog.s        # ensamblador
gcc -c prog.s -o prog.o        # objeto
gcc prog.o -o prog             # ejecutable
```

Se pide inspeccionar cada intermedio: cuánto crece el archivo tras el preprocesado —una sola `#include <stdio.h>` añade miles de líneas—, qué aspecto tiene el ensamblador generado, y qué

símbolos define el objeto con `nm`.

6.3.2 Varios módulos

```
gcc -Wall -Wextra -g -c main.c pila.c
gcc main.o pila.o -o programa
```

Los errores que aparecen y cómo se leen:

Mensaje	Dónde está el problema
`error: 'x' undeclared`	compilación: falta la declaración o el `#include`
`undefined reference to 'f'`	enlazado: falta el `.o` o la biblioteca
`multiple definition of 'v'`	enlazado: una variable definida en una cabecera
`warning: implicit declaration`	compilación: se usa una función sin declarar

El último es un aviso y no un error, y conviene tratarlo como error: significa que el compilador está adivinando los tipos, y si adivina mal el fallo es en ejecución. `-Werror` convierte los avisos en errores y es una buena costumbre.

6.3.3 Bibliotecas

```
# estatica
gcc -c pila.c cola.c
ar rcs libtda.a pila.o cola.o
gcc main.c -L. -ltda -o programa
nm libtda.a | head

# dinamica
gcc -fPIC -c pila.c cola.c
gcc -shared -o libtda.so pila.o cola.o
gcc main.c -L. -ltda -o programa
LD_LIBRARY_PATH=. ./programa
ldd programa
```

Lo que la práctica pide comprobar: el **tamaño del ejecutable** con una y con otra, y qué ocurre al modificar la biblioteca sin recompilar el programa. Con la estática no cambia nada, porque el código está dentro; con la dinámica, el programa usa la versión nueva. Esa es toda la diferencia, y explica por qué las actualizaciones de seguridad del sistema funcionan.

6.3.4 Makefile

Se pide escribir uno con compilación separada, dependencias correctas de cabeceras y un objetivo `clean`, y comprobar que al tocar un solo `.c` se recompila solo ese módulo.

La comprobación que enseña de verdad: **tocar una cabecera**. Si el Makefile no declara esa dependencia, `make` no recompila nada y el programa queda incoherente. Es el error del tema 3, visto en el propio proyecto.

6.3.5 Depuración

```
gcc -g -O0 prog.c -o prog
gdb ./prog
```

```
valgrind --leak-check=full ./prog
gcc -fsanitize=address -g prog.c -o prog && ./prog
```

Se propone introducir errores a propósito —salirse de un vector, liberar dos veces, usar memoria liberada, olvidar un `free`— y comprobar qué dice cada herramienta. Vale la pena ver que **el programa con el error no siempre falla**: eso es lo que hace peligrosos estos fallos y lo que justifica usar las herramientas.

6.4 Sobre la memoria de prácticas

Lo que se entrega por práctica:

1. Enunciado resuelto, con las órdenes o el código.
2. Salidas obtenidas, no descritas: pegadas.
3. Explicación de **qué hace cada orden y por qué esa y no otra**.
4. Los errores encontrados y cómo se diagnosticaron.
5. Conclusiones.

El punto 4 es el que distingue una memoria buena. Un guion que funciona a la primera enseña menos que uno que falló por comillas y se arregló entendiendo la expansión de palabras.

Los guiones de laboratorio y su material de apoyo siguen Newham y Rosenblatt, 2005 para la shell, Gough, 2005 y Mecklenburg, 2004 para la compilación, y Stallman et al., 2003 y Nethercote et al., 2008 para la depuración.

Bibliografía

Carretero, J., García Carballeira, F., de Miguel, P., & Pérez, F. (2021). *Sistemas Operativos* (3.ª ed.). McGraw-Hill.

Gough, B. J. (2005). *An Introduction to GCC* (2.ª ed.). Network Theory Limited.

Johnson, M. K., & Troan, E. W. (2005). *Linux Application Development* (2.ª ed.). Addison-Wesley Professional.

Matthew, N., & Stones, R. (2008). *Beginning Linux Programming* (4.ª ed.). Wiley.

Mecklenburg, R. (2004). *Managing Projects with GNU Make* (3.ª ed.). O'Reilly.

Nethercote, N., Weidendorfer, J., & Seward, J. (2008). *Valgrind 3.3: Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Limited.

Newham, C., & Rosenblatt, B. (2005). *Learning the bash Shell* (3.ª ed.). O'Reilly.

Prieto, A., Lloris, A., & Torres, J. C. (2006). *Introducción a la Informática* (4.ª ed.). McGraw-Hill.

Stallings, W. (2018). *Operating Systems: Internals and Design Principles* (9.ª ed.). Pearson.

Stallman, R. M., Pesch, R. H., & Shebs, S. (2003). *Debugging with GDB: The GNU Source-Level Debugger* (9.ª ed.). Free Software Foundation.