

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Bugs de concurrencia, eventos y dispositivos de E/S

Cuando la concurrencia falla, y cómo el sistema operativo habla con el hardware

Gerson Garrido

Semana 12

Hoy

Sesión A · Cuando la concurrencia falla

1. Bugs reales: violación de atomicidad y de orden
2. Deadlock: prevenir, evitar (banquero), detectar, ignorar
3. Concurrencia por eventos: `poll`, `epoll`, `io_uring`

Sesión B · Persistencia: el hardware

4. Dispositivos: polling, interrupciones, DMA, drivers
5. xv6: el driver de la UART y el del disco virtio
6. Discos duros: geometría, tiempo de E/S, planificación

Lectura

OSTEP cap. 32 *Common Concurrency Problems*, cap. 33 *Event-based Concurrency*, cap. 34 (resumen), cap. 35 (diálogo), cap. 36 *I/O Devices* y cap. 37 *Hard Disk Drives*. Código: `code/week-12`.

SECCIÓN 01

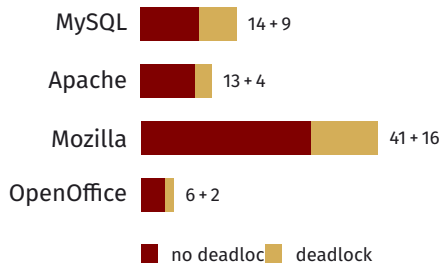
Bugs que no son deadlock

01



¿Qué bugs de concurrencia aparecen en la práctica?

- Lu y colegas (2008) estudiaron 105 bugs de concurrencia ya corregidos en cuatro programas grandes y maduros.
- **74** no eran deadlock; **31** sí.
- De los que no son deadlock, el 97 % cae en dos tipos:
  - violación de **atomicidad**
  - violación de **orden**
- Conocer los patrones ayuda a buscarlos, y a no escribirlos.



S. Lu, S. Park, E. Seo, Y. Zhou. *Learning from Mistakes*. ASPLOS 2008.

Violación de atomicidad

Se supone que un grupo de accesos es atómico, pero nada lo garantiza.

caché de un servidor web

```
1  // Hilo 1: responder desde la caché
2  if (c->page != NULL)
3      send(sock, c->page->data,
4           c->page->len, 0);
5
6  // Hilo 2: invalidar la caché
7  free(c->page);
8  c->page = NULL;
```

	Hilo 1	Hilo 2
1	`c->page != NULL`: sí	
2	*desalojado*	
3		`free(c->page)`
4		`c->page = NULL`
5	lee `c->page->data`	

Resultado

`NULL->data`: `segfault`. Si el hilo 1 ya había leído el puntero, usa memoria liberada: peor, porque no siempre se nota.

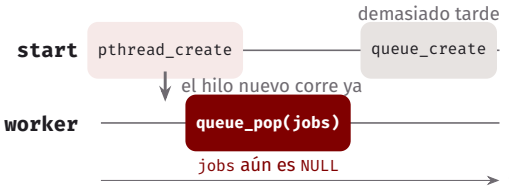
Violación de orden

Se espera que A ocurra antes que B, pero el orden no se impone.

arranque de un servidor

```c
struct queue *jobs;   // global

void start(void) {
    pthread_create(&t, NULL,
                   worker, NULL);
    jobs = queue_create();
}
void *worker(void *arg) {
    run(queue_pop(jobs));
    return NULL;
}
```



Resultado

Casi siempre funciona: `start` suele seguir corriendo. Basta que el planificador elija primero al hilo nuevo para que falle.

Cómo se arreglan

Atomicidad ⇒ un lock alrededor de *todo* el grupo

con un lock

```
1  // Hilo 1
2  pthread_mutex_lock(&c->m);
3  if (c->page != NULL)
4      send(sock, c->page->data,
5           c->page->len, 0);
6  pthread_mutex_unlock(&c->m);
7  // Hilo 2
8  pthread_mutex_lock(&c->m);
9  free(c->page);
10 c->page = NULL;
11 pthread_mutex_unlock(&c->m);
```

Orden ⇒ una variable de condición

con una variable de condición

```
1  // start(), después de crear la cola
2  pthread_mutex_lock(&m);
3  jobs = queue_create();
4  ready = 1;
5  pthread_cond_signal(&cv);
6  pthread_mutex_unlock(&m);
7  // worker(), antes de usarla
8  pthread_mutex_lock(&m);
9  while (ready == 0)
10     pthread_cond_wait(&cv, &m);
11 pthread_mutex_unlock(&m);
```

Herramientas de las semanas 10 y 11. Lo difícil es notar que hacían falta. (Aquí, lo más simple es crear la cola *antes* que el hilo.)

SECCIÓN 02

Deadlock

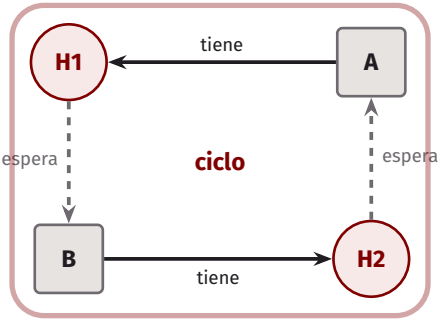
02

Deadlock: el ejemplo mínimo

dos hilos, dos locks

```
1  // Hilo 1          // Hilo 2
2  lock(&A);          lock(&B);
3  lock(&B);          lock(&A);
```

Hilo 2 pide `A`: se bloquea. El grafo tiene un ciclo: cada uno espera algo que tiene el otro, y nadie lo va a soltar. Esto es un deadlock.



grafo de asignación de recursos *(resource-allocation graph)*

En código real el ciclo se esconde: un sistema de archivos que pide memoria mientras la memoria virtual pide el sistema de archivos, o `vector_add_all(v1, v2)` en un hilo y `vector_add_all(v2, v1)` en otro. Ya lo vimos con los filósofos (`code/week-11/philo.c`).

Las cuatro condiciones de Coffman (1971)

Exclusión mutua

Cada recurso lo usa un hilo a la vez.

Retener y esperar

(hold-and-wait)

Un hilo retiene recursos mientras espera otros.

Sin expropiación

(no preemption)

Nadie puede quitarle a un hilo lo que tiene.

Espera circular

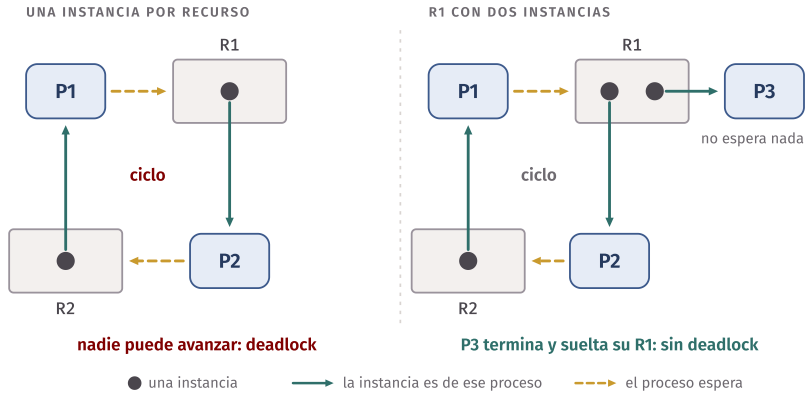
(circular wait)

Hay un ciclo: cada hilo espera algo que tiene el siguiente.

Las cuatro son necesarias

Si falta una sola, no hay deadlock. Toda técnica de **prevención** rompe una de ellas a propósito. Las otras opciones: **evitarlo** con planificación, **detectarlo** y recuperarse, o **ignorarlo**.

Recursos con varias instancias



Sin ciclo, nunca hay deadlock. Ciclo + una instancia por recurso $\Rightarrow$ deadlock. Ciclo + varias instancias: puede que no; hay que simular.

Prevenir 1: romper la espera circular con un orden

- Regla: todos toman los locks en el mismo orden global. Si siempre se toma `A` antes que `B`, no puede haber ciclo.
- **Orden total**: numerar todos los locks. **Orden parcial**: reglas entre grupos; Linux documenta el suyo en comentarios (por ejemplo, en `mm/filemap.c`).
- Truco cuando la función recibe los locks: ordenar por dirección.
- Es una *convención*: el compilador no la verifica. Herramientas como `lockdep` (Linux) la revisan en tiempo de ejecución.

deadlock.c (modo ordered)

```c
// el de menor dirección, primero
if (first > second) {
    pthread_mutex_t *tmp = first;
    first = second;
    second = tmp;
}
pthread_mutex_lock(first);
pthread_mutex_lock(second);
```

¿Qué condición rompe?

La **espera circular**. Las otras tres siguen ahí, pero solas no alcanzan.

Prevenir 2 y 3: retener y esperar, sin expropiación

Retener y esperar ⇒ pedir todo de una vez

modo global

```
1  pthread_mutex_lock(&prevention);
2  pthread_mutex_lock(first);
3  pthread_mutex_lock(second);
4  pthread_mutex_unlock(&prevention);
```

- Mientras uno adquiere, nadie más adquiere.
- Hay que saber de antemano qué locks se usarán, y se pierde concurrencia.

Sin expropiación ⇒ soltar lo que se tiene

modo trylock

```
1  for (;;) {
2      pthread_mutex_lock(first);
3      if (!pthread_mutex_trylock(second))
4          break;              // tengo los dos
5      pthread_mutex_unlock(first);
6      sched_yield();      // y reintentar
7  }
```

- El hilo se *expropia* a sí mismo: devuelve `first`.
- Riesgo de livelock: los dos sueltan y reintentan a la vez, para siempre. Arreglo: esperar un tiempo *aleatorio*.
- Si entre los dos locks se reservó memoria u otra cosa, también hay que devolverla.

Prevenir 4: sin exclusión mutua (lock-free)

lista enlazada sin lock

```
1  void insert(struct node **head, int key) {
2      struct node *n = malloc(sizeof *n);
3      n->key = key;
4      do {
5          n->next = *head;  // leer la cabeza
6      } while (!CAS(head, n->next, n));
7  }
```

`CAS(p, old, new)`, compare-and-swap: si `*p == old`, escribe `new` y devuelve 1; si no, devuelve 0. Todo en una sola operación atómica (semana 10).

- Sin locks no hay nada que retener: no puede haber deadlock.
- Si otro hilo cambió `*head` entre la lectura y el CAS, el CAS falla y se reintenta con la cabeza nueva.
- Un hilo puede reintentar varias veces, pero en cada vuelta *alguno* avanza.
- Insertar es fácil. Borrar, liberar memoria y estructuras completas son difíciles: el problema ABA, cuándo hacer `free`… Es un campo de investigación.

Evitar: la intuición del banco

Un banco tiene **12** mil soles. Quien llega a su línea de crédito termina y devuelve *todo*. Las líneas suman 23: no todos llegarán al máximo a la vez.

Cliente	Línea	Prestado	Le falta
Ana	8	4	4
Beto	5	2	3
Carla	10	3	7

En caja: $12 - 9 = \mathbf{3}$

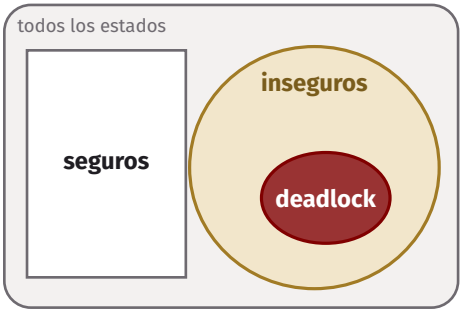
- **Peor caso**: todos piden ya lo que les falta. Con 3, Beto termina y devuelve 5 $\rightarrow$ caja 8 $\rightarrow$ Ana termina $\rightarrow$ 12 $\rightarrow$ Carla. Todos terminan: seguro.
- **Ana pide 1**: caja 2; faltan 3, 3 y 7. Si todos piden su máximo, nadie termina: inseguro. Ana espera, aunque hay plata.
- **Beto pide 1**: caja 2; a Beto le faltan 2: termina, y todo sigue como antes. Se concede.

Mismo monto, distinta respuesta

Solo presta si sigue habiendo un orden en que todos terminan: el **algoritmo del banquero** (Dijkstra, 1965).

Seguro, inseguro y deadlock

- Un estado es seguro *(safe state)* si existe una **secuencia segura** *(safe sequence)*: un orden en que cada proceso puede recibir lo que le falta con lo disponible más lo que devolvieron los anteriores.
- Desde un estado seguro, el sistema *puede* evitar el deadlock con certeza: solo concede pedidos que lleven a otro estado seguro.
- Inseguro no es deadlock: nadie está bloqueado todavía, y si Ana no usa toda su línea, todo sale bien. Pero ya no hay *garantía*: depende de lo que pidan los procesos.
- El precio: saber de antemano el **máximo** de cada proceso.



El algoritmo del banquero *(banker's algorithm)*

n procesos, m tipos de recurso; las cantidades son vectores:

Disponible	m	instancias libres *(Available)*
Máximo	$n \times m$	lo más que pedirá P_i *(Max)*
Asignado	$n \times m$	lo que P_i tiene *(Allocation)*
Necesidad	$n \times m$	Máximo $-$ Asignado *(Need)*

Solicitud de P_i:

1. $>$ Necesidad$_i$: error (pide más de lo declarado).
2. $>$ Disponible: espera.
3. Si no, simular la concesión y correr el algoritmo de seguridad. Seguro: se concede. Inseguro: se deshace y P_i espera.

Algoritmo de seguridad

1. Trabajo *(Work)* $=$ Disponible; nadie terminó.
2. Buscar P_i sin terminar con Necesidad$_i \leq$ Trabajo (componente a componente).
3. Si existe: Trabajo $+=$ Asignado$_i$ (termina y devuelve todo). Volver a 2.
4. Seguro $\iff$ todos terminaron. Ese orden es una secuencia segura.

¿Por qué basta ser voraz?

Terminar solo hace crecer a Trabajo: los otros candidatos seguirán cabiendo. No hay que probar todos los órdenes. Costo: $O(m\,n^2)$.

Un ejemplo: tres tipos, cuatro procesos

	Máximo	Asignado	Necesidad
	A B C	A B C	A B C
P1	5 2 2	1 1 0	4 1 2
P2	3 2 2	2 0 1	1 2 1
P3	2 2 3	1 1 2	1 1 1
P4	1 4 2	0 2 1	1 2 1

Disponible $= (2\ 1\ 2)$

Total $= (2\ 1\ 2) + (4\ 4\ 4) = (6\ 5\ 6)$

- Necesidad: Máximo − Asignado, fila por fila.
- El total es lo libre más lo asignado.
- ¿Es seguro? Con Trabajo $= (2\ 1\ 2)$, ¿de quién cabe la Necesidad? Solo de P3.

El estado está en `code/week-12/bank.txt`; lo analiza `banker.c`.

Buscando una secuencia segura



Con 6 3 5 cabe P4. Todos terminan: el estado es seguro.

Hay otras secuencias seguras (como ⟨P3, P4, P2, P1⟩): basta encontrar una.

Dos solicitudes: una se concede, otra se niega

code/week-12, en el contenedor

```
$ ./banker bank.txt
...
pide P1 1 0 1
  cabe en necesidad y disponible; si se concede, disponible = 1 1 1
  ...
  seguro (P3 P2 P1 P4): se concede
pide P2 0 1 0
  cabe en necesidad y disponible; si se concede, disponible = 1 0 1
  trabajo 1 0 1: nadie cabe (P1 3 1 1, P2 1 1 1, P3 1 1 1, P4 1 2 1)
  inseguro: se niega; P2 espera y el estado vuelve atrás
```

- P1 pide `(1 0 1)`: queda Disponible `(1 1 1)`, justo lo que necesita P3. Sigue seguro: se concede.
- Luego P2 pide `(0 1 0)`, la última B. Después, *todos* necesitan otra B: inseguro, se niega.
- El pedido negado era *más chico* que el concedido.
- Riesgo: un pedido grande puede quedar postergado para siempre (**inanición**). Se atiende a los que esperan en orden de llegada.

Detectar con una instancia: el grafo de espera

- Dejar que ocurra (si es raro) y tener un detector: cada cierto tiempo arma el grafo de espera *(wait-for graph)* y busca ciclos.
- Grafo de espera: $P_i \to P_j$ si P_i espera algo que tiene P_j. Con una instancia por recurso, ciclo $=$ deadlock.
- En `deadlock.c`, un hilo *watchdog* ve si el avance se detuvo 2 s y muestra quién tiene y quién espera cada lock.
- Los otros tres modos son las técnicas de prevención: terminan siempre.

code/week-12, en el contenedor

```
$ ./deadlock naive
watchdog: 2 s sin avance (862 de 200000 vueltas)
  hilo 0: tiene A, espera B
  hilo 1: tiene B, espera A
  ciclo: H0 -> B -> H1 -> A -> H0: deadlock
$ ./deadlock ordered
ordered: counter = 200000 de 200000 en 0.025 s
$ ./deadlock global
global: counter = 200000 de 200000 en 0.049 s
$ ./deadlock trylock
trylock: counter = 200000 de 200000 en 0.027 s
  27727 reintentos
```

Detectar con varias instancias

- El grafo de espera ya no sirve: un ciclo puede no ser deadlock.
- El mismo algoritmo de seguridad, con **Solicitud** (lo que cada uno pide *ahora*) en lugar de Necesidad. No hace falta conocer el futuro.
- Optimista: quien consigue lo que pide, termina; si luego pide más, la próxima revisión lo verá.
- Los que no terminan están en deadlock: P1, P2 y P3.
- ¿Cada cuánto? PostgreSQL espera `deadlock_timeout` (1 s) antes de buscar ciclos.

code/week-12, en el contenedor

```
$ ./banker detect.txt
recursos: A B C   total: 3 3 2   disponible: 0 1 0
proceso  solicitud  asignado
P1       0 0 1      1 0 1
P2       0 0 1      0 1 0
P3       2 0 0      1 1 1
P4       0 1 0      1 0 0

detección (solicitud en lugar de necesidad)
  trabajo 0 1 0: P4 pide 0 1 0, cabe; termina ...
  trabajo 1 1 0: nadie más cabe (P1 0 0 1, ...)
deadlock: P1 P2 P3

recuperación: abortar a uno solo
  abortar P1 (libera 1 0 1): los demás terminan ...
  abortar P2 (libera 0 1 0): no basta, siguen ...
  abortar P3 (libera 1 1 1): los demás terminan ...
```

Recuperarse: ninguna opción es elegante

Abortar procesos

Todos los del deadlock, o de uno en uno volviendo a detectar (abortar P2 no bastaba). Víctima: la que menos se pierde.

Expropiar recursos

Quitarle un recurso a uno y dárselo a otro. Solo si el recurso lo permite: un marco de memoria va a swap; una impresora a mitad de página, no.

Retroceder *(rollback)*

Volver a un *checkpoint* anterior a tomar el recurso. Las bases de datos abortan la transacción, la deshacen con el log y la repiten.

Problemas conocidos

Si siempre se elige a la misma víctima, nunca termina: **inanición** (se cuenta cuántas veces fue víctima). Si las víctimas reintentan a la vez, vuelven a chocar: el **livelock** de `trylock`, con el mismo arreglo, una espera aleatoria.

La estrategia del avestruz

Linux, Windows y macOS, en general, no evitan ni detectan deadlocks entre procesos de usuario: dos `pthread_mutex` en orden opuesto se quedan así para siempre. ¿Por qué?

- **Frecuencia contra costo**: es raro, y detectarlo exigiría registrar *cada* lock.
- **El banquero pide lo imposible**: nadie declara su máximo de memoria, archivos o locks.
- **La recuperación sería la misma**: matar un proceso, lo que el usuario hace con `kill` o Ctrl-C.
- **Muchos recursos no se esperan**: tabla llena ⇒ `fork` falla con `EAGAIN`, `open` con `EMFILE`. Sin espera, no hay ciclo.

Lo que sí hay

- `lockdep` (depuración del kernel): avisa de órdenes de locks opuestos aunque el deadlock no haya ocurrido.
- Detector de tareas colgadas: avisa si una tarea pasa 120 s en estado D. Detecta, no recupera.
- `fcntl(F_SETLKW)` falla con `EDEADLK` si cerraría un ciclo (hasta 10 pasos; puede equivocarse).
- Bases de datos: detectan, abortan una transacción y la repiten.

Cuatro estrategias

Prevenir (en el código), **evitar** (banquero), **detectar** (bases de datos), **ignorar** (el sistema operativo).

SECCIÓN 03

Concurrencia basada en eventos

03

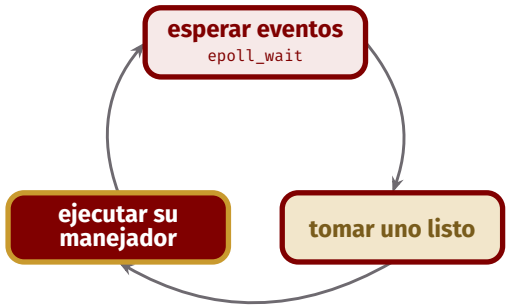


Otra forma de hacer muchas cosas a la vez

el ciclo de eventos (event loop)

```
1  while (1) {
2      events = wait_for_events();
3      for (e in events)
4          handle(e);   // no debe bloquear
5  }
```

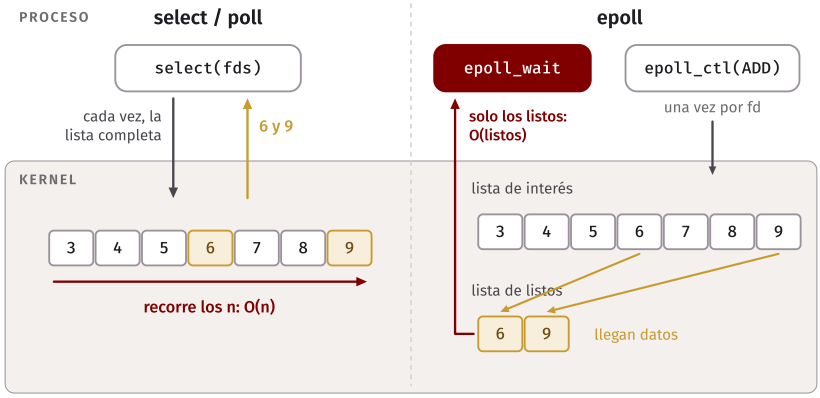
- Un solo hilo espera a que *algo* pase (llega una conexión, un socket tiene datos) y atiende ese evento.
- Nadie lo interrumpe en medio de un manejador: no hacen falta locks, y el programa decide qué atender primero.
- Lo usan nginx, Node.js, Redis, los navegadores…



De `select`/`poll` a `epoll`

- `select` (1983) y `poll`: “de estos *n* descriptores, ¿cuáles están listos para leer o escribir?”
- En *cada* llamada se pasa la lista completa y el kernel la recorre: costo $O(n)$ aunque solo uno tenga datos. `select` además se limita a `FD_SETSIZE` (1024).
- `epoll` (Linux 2.6): el kernel recuerda la lista de interés (`epoll_ctl`) y arma la de listos a medida que llegan datos. `epoll_wait` cuesta O(listos).
- Por omisión `epoll` avisa por *nivel*: mientras queden datos, el fd sigue listo. Con `EPOLLET` avisa por *flanco*: solo cuando cambia, y hay que leer hasta `EAGAIN`.
- En BSD y macOS: `kqueue`. En Windows: IOCP.

`select` frente a `epoll`



Con `select` el trabajo crece con los fd vigilados; con `epoll`, solo con los que tienen datos.

Un servidor de eco con `epoll`

epollecho.c (resumido)

```c
int ep = epoll_create1(0);  // + registrar ls
while (closed < limit) {
  int n = epoll_wait(ep, ready, MAXEV, -1);
  for (int i = 0; i < n; i++) {
    struct conn *c = ready[i].data.ptr;
    if (c == NULL) {          // conexión nueva
      int fd = accept(ls, NULL, NULL);
      set_nonblocking(fd);
      epoll_ctl(ep, EPOLL_CTL_ADD, fd, &e);
      continue;
    }
    ssize_t got = read(c->fd, c->buf, BUFSZ);
    if (got <= 0) { close(c->fd); continue; }
    c->len = got;
    if (flush(c) == 0)        // envío parcial
      watch(ep, c, EPOLLOUT);
  }
}
```

en el contenedor

```
$ ./epollecho test 50 20
servidor: puerto 33853, 1 cliente
  lento + 50 clientes x 20 mensajes
servidor: 50 conexiones atendidas
  en 0.012 s; 1 sigue abierta sin
  enviar nada
servidor: 51 conexiones, 22331
  bytes, 41 llamadas a epoll_wait
  (hasta 50 fd listos a la vez)
clientes: 51 de 51 recibieron su
  eco completo
```

- Un hilo, 51 conexiones. El cliente lento no frena a nadie.
- Un servidor de un hilo con `read` bloqueante haría esperar a los 50.

E/S no bloqueante y el estado a mano

con un hilo por cliente

```c
char buf[4096];  // en la pila
n = read(sock, buf, sizeof buf);
write(sock, buf, n);  // puede esperar
```

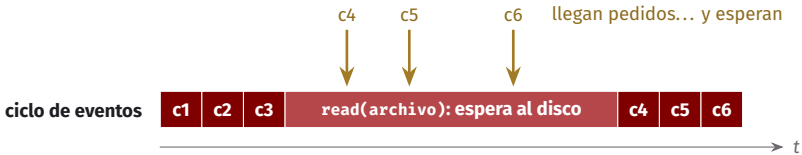
- Si `write` no puede enviar todo, el hilo se bloquea y sigue después. El estado (`buf`, cuánto falta) queda en su pila.
- Con `O_NONBLOCK`, `read` y `write` nunca esperan: si no pueden avanzar devuelven -1 con `errno = EAGAIN`, o escriben solo una parte.

con eventos (epollecho.c)

```c
struct conn {      // lo que antes estaba
    int fd;        // en la pila del hilo
    char buf[BUFSZ];
    size_t len;    // bytes en buf
    size_t off;    // bytes ya enviados
};
```

- Si `write` devuelve `EAGAIN`, el manejador guarda dónde quedó y pide a `epoll` que avise con `EPOLLOUT`. Cuando llegue ese evento, otro llamado retoma el trabajo.
- Es el **manejo manual de la pila** *(manual stack management)*: la estructura que guarda "cómo continuar" es una continuación *(continuation)*.

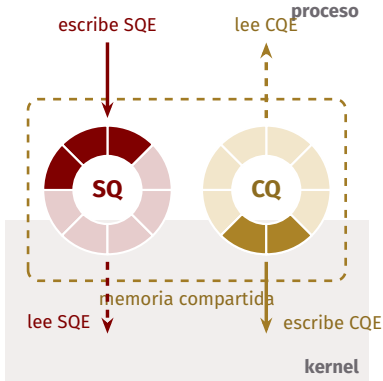
El problema: lo que bloquea sin avisar



- Una sola llamada que bloquea (leer un archivo del disco, resolver un nombre con DNS) detiene todo el servidor: es el único hilo.
- Peor aún, los fallos de página: si el manejador toca una página que está en swap, el hilo se bloquea sin haber llamado a nada. No se puede evitar con `O_NONBLOCK`.
- Soluciones: **E/S asíncrona** (pedir la operación y enterarse después), o un **grupo de hilos** auxiliar para lo que bloquea (así hacen libuv y nginx). Y en multinúcleo, un ciclo de eventos por núcleo… con locks otra vez.

E/S asíncrona en Linux: `io_uring`

- POSIX AIO (`aio_read`, `aio_error`): en Linux, glibc la implementa con hilos; la AIO nativa del kernel solo es asíncrona con `O_DIRECT`.
- `io_uring` (Linux 5.1, 2019): dos colas circulares en memoria compartida entre el proceso y el kernel (vía `mmap`).
- El proceso escribe pedidos (SQE) en la cola de envío; el kernel deja resultados (CQE) en la de completados.
- Una llamada `io_uring_enter` envía muchos pedidos y espera resultados. Sirve para archivos y sockets.



Lo que no resuelve: un fallo de página en el código del propio servidor sigue bloqueando a ese hilo.

¿Hilos o eventos? Lo que aprendimos de concurrencia

	Hilos	Eventos
Estado de cada tarea	en la pila, automático	a mano (continuaciones)
Datos compartidos	locks, variables de condición	sin locks (con un hilo)
Bugs típicos	atomicidad, orden, deadlock	un manejador que bloquea
Varios núcleos	natural	un ciclo por núcleo: vuelven los locks
E/S que bloquea	bloquea solo a ese hilo	bloquea a todos: E/S asíncrona

Cap. 34: el consejo de OSTEP

Simplicidad. Evitar la concurrencia cuando se pueda; si hace falta, usar patrones probados (locks, productor/consumidor, orden de locks) o modelos más simples (eventos, map-reduce). Y leer mucho código ajeno.

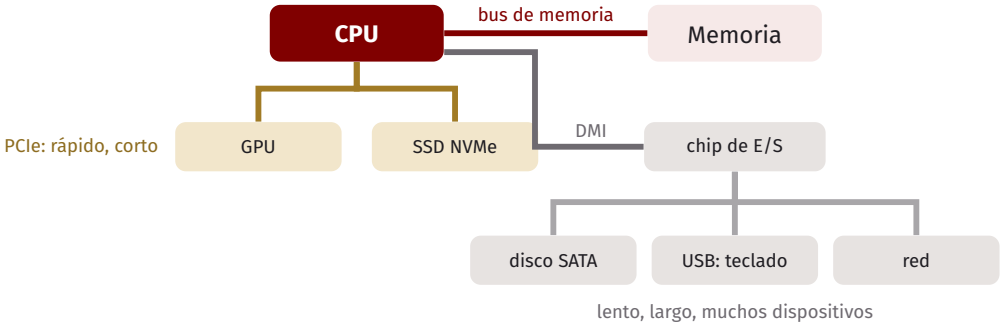
SECCIÓN 04

Dispositivos de E/S

04



Cómo se conectan los dispositivos



Jerarquía: lo más rápido, cerca de la CPU y en buses cortos (caros); lo más lento, lejos, en buses que admiten muchos dispositivos. Mandan la física y el costo.

El dispositivo canónico y su protocolo

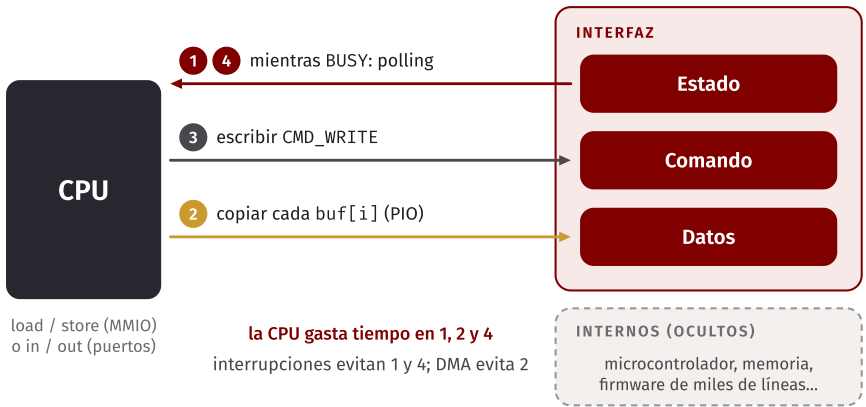
El sistema operativo solo ve la interfaz: tres registros (**estado**, **comando** y **datos**). Lo de adentro (un microcontrolador, memoria, en un disco moderno firmware de miles de líneas) queda oculto.

- Pasos 1 y 4: polling, preguntar una y otra vez. Gasta CPU.
- Paso 2: la CPU copia cada palabra: E/S programada *(PIO)*. Más CPU.

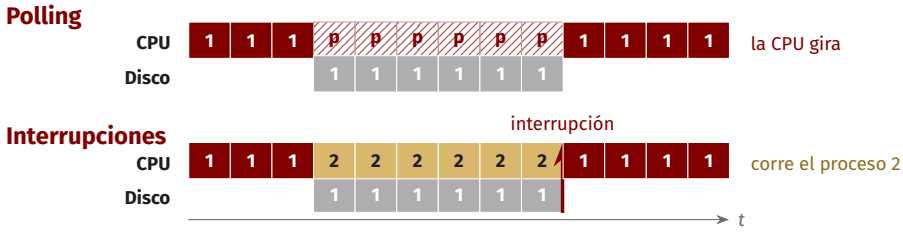
protocolo con polling y PIO

```
1  // 1. esperar a que esté libre
2  while (*status & BUSY) ;
3  // 2. pasarle los datos, palabra por palabra
4  for (int i = 0; i < n; i++)
5      *data = buf[i];
6  // 3. darle la orden
7  *command = CMD_WRITE;
8  // 4. esperar a que termine
9  while (*status & BUSY) ;
```

El protocolo, en los registros



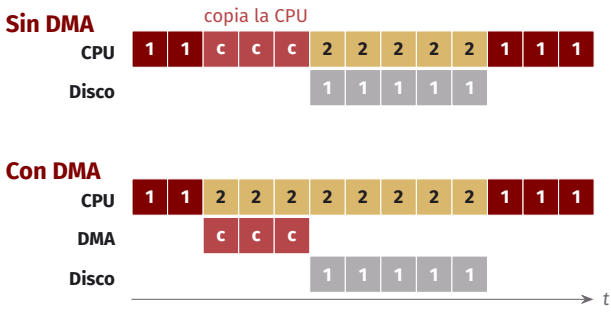
Polling o interrupciones



¿Siempre interrupciones? No

Si el dispositivo responde en microsegundos, atender una interrupción cuesta más que esperar: mejor **polling**. Con una avalancha de interrupciones (red), el sistema solo atiende interrupciones: livelock. Arreglos: híbridos (girar un rato y luego dormir), agrupar interrupciones *(coalescing)*, o pasar a polling bajo carga (NAPI en Linux).

DMA: que la CPU no copie



- El sistema operativo le dice al motor de DMA *(Direct Memory Access)*: "copia *n* bytes entre esta dirección de memoria y el dispositivo". Al terminar, interrumpe.
- Mientras tanto, la CPU corre el proceso 2.
- Hoy cada dispositivo rápido (disco, red, GPU) hace su propio DMA. Por eso el kernel le pasa direcciones físicas.

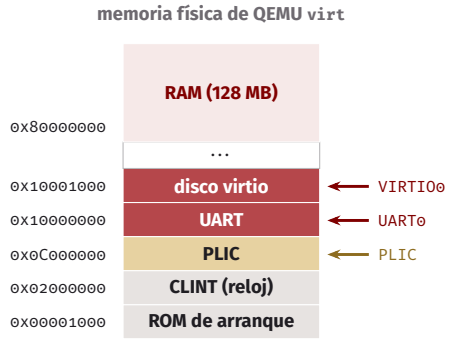
Cómo habla la CPU con los registros del dispositivo

E/S por puertos *(port-mapped I/O)*

Instrucciones especiales y privilegiadas: en x86, `in` y `out`, sobre un espacio de "puertos" aparte de la memoria.

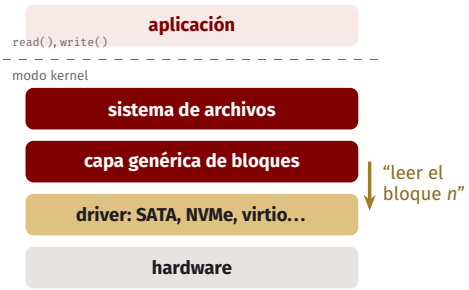
E/S mapeada en memoria *(memory-mapped I/O)*

Los registros tienen direcciones físicas: se usan con `load` y `store` comunes; el hardware desvía esos accesos al dispositivo.



RISC-V no tiene instrucciones de puertos: todo es MMIO (direcciones de `kernel/memlayout.h`).

Drivers: encapsular lo específico



- El **driver** *(device driver)* es la parte del kernel que conoce el protocolo de *un* dispositivo. Arriba, todo es genérico.
- Un sistema de archivos funciona igual sobre un disco SATA, un SSD o un USB.
- Precio: la interfaz común esconde funciones especiales del dispositivo.
- Según OSTEP, más del 70 % del código de Linux son drivers, y ahí se encuentran más bugs: mucho código, de muchos autores, poco probado.

xv6: la UART con polling

kernel/uart.c (extracto)

```
1  // los registros están mapeados en memoria
2  #define Reg(reg) ((volatile unsigned char *) \
3                    (UART0 + (reg)))
4  #define ReadReg(reg)     (*(Reg(reg)))
5  #define WriteReg(reg, v) (*(Reg(reg)) = (v))
6
7  #define THR 0  // transmit holding register
8  #define LSR 5  // line status register
9  #define LSR_TX_IDLE (1 << 5)
10
11 void uartputc_sync(int c) {
12   // ...
13   while ((ReadReg(LSR) & LSR_TX_IDLE) == 0)
14     ;
15   WriteReg(THR, c);
16   // ...
17 }
```

- La UART 16550 es el puerto serie: la consola de xv6 en QEMU.
- `Reg(LSR)` es la dirección `0x10000005`: un `load` común lee el registro de estado.
- `uartputc_sync` es el protocolo canónico con polling: girar hasta que el registro de envío esté libre, y escribir el byte.
- Se usa para `printk` del kernel y para el eco de lo que se teclea: tiene que funcionar incluso con las interrupciones apagadas o en un `panic`.

xv6: la UART con interrupciones

`kernel/uart.c` (reformateado): mitad superior *(top half)* e inferior *(bottom half)*

uartwrite: mitad superior, duerme

```
1  void uartwrite(char buf[], int n) {
2    acquire(&tx_lock);
3    int i = 0;
4    while (i < n) {
5      while (tx_busy != 0) {
6        sleep_prepare(&tx_chan);
7        release(&tx_lock);
8        sleep();
9        acquire(&tx_lock);
10     }
11     WriteReg(THR, buf[i]);
12     i += 1; tx_busy = 1;
13   }
14   release(&tx_lock);
15 }
```

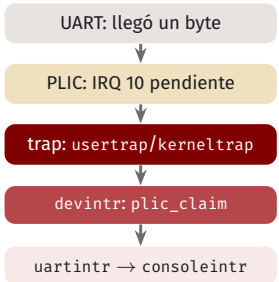
uartintr: mitad inferior, despierta

```
1  void uartintr(void) {
2    ReadReg(ISR);    // acusar recibo
3    acquire(&tx_lock);
4    if (ReadReg(LSR) & LSR_TX_IDLE) {
5      tx_busy = 0;
6      wakeup(&tx_chan);
7    }
8    release(&tx_lock);
9    while (1) {      // bytes recibidos
10     int c = uartgetc();
11     if (c == -1)
12       break;
13     consoleintr(c);
14   }
15 }
```

¿Quién llama a `uartintr`? `devintr` y el PLIC

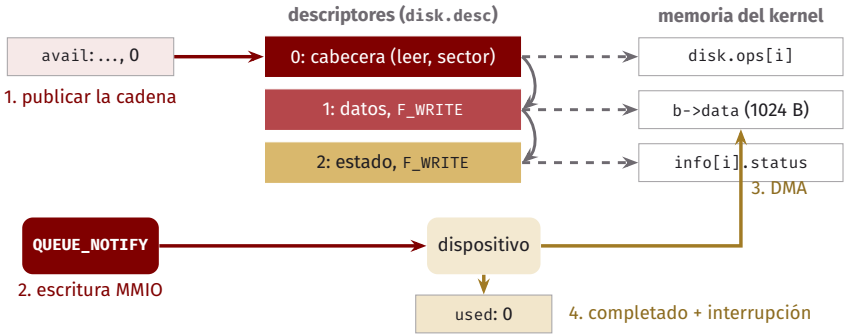
kernel/trap.c (extracto)

```
1  int devintr() {
2    uint64 scause = r_scause();
3    if (scause == 0x8000000000000009L) {
4      int irq = plic_claim();
5      if (irq == UART0_IRQ) {
6        uartintr();
7      } else if (irq == VIRTIO0_IRQ) {
8        virtio_disk_intr();
9      } // ...
10     if (irq)
11       plic_complete(irq);
12     return 1;
13   } else if (scause == 0x8000000000000005L) {
14     clockintr();     // reloj
15     return 2;
16   } // ...
17 }
```



- El PLIC (`kernel/plic.c`) junta las interrupciones de los dispositivos. `plic_claim` lee un registro MMIO: "¿cuál fue?"
- `plic_complete`: ya se atendió; puede volver a interrumpir.

xv6: el disco virtio, un pedido con DMA



Al terminar, el dispositivo anota la cadena en el anillo `used` e interrumpe. Mientras tanto, el proceso que pidió el bloque está dormido.

xv6: dormir hasta que el disco termine

virtio_disk_rw (final, reformateado)

```
1   b->disk = 1;   // el disco es owner
2   disk.info[idx[0]].b = b;
3   int k = disk.avail->idx % NUM;
4   disk.avail->ring[k] = idx[0];
5   io_fence();
6   disk.avail->idx += 1;
7   io_fence();
8   *R(VIRTIO_MMIO_QUEUE_NOTIFY) = 0;
9   while (b->disk == 1) {  // esperar
10    sleep_prepare(b);
11    release(&disk.vdisk_lock);
12    sleep();
13    acquire(&disk.vdisk_lock);
14  }
```

virtio_disk_intr (reformateado)

```
1   acquire(&disk.vdisk_lock);
2   *R(VIRTIO_MMIO_INTERRUPT_ACK) =
3     *R(VIRTIO_MMIO_INTERRUPT_STATUS)
4     & 0x3;
5   io_fence();
6   while (disk.used_idx != disk.used->idx) {
7     io_fence();
8     int k = disk.used_idx % NUM;
9     int id = disk.used->ring[k].id;
10    struct buf *b = disk.info[id].b;
11    b->disk = 0;   // terminó
12    wakeup(b);
13    disk.used_idx += 1;
14  }
15  release(&disk.vdisk_lock);
```

El canal de `sleep`/`wakeup` es el propio `struct buf`. Juntos: MMIO, DMA e interrupciones.

SECCIÓN 05

Discos duros

05

El disco duro por dentro



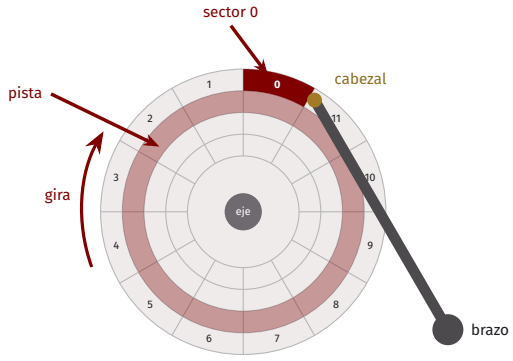
Disco de laptop abierto. Evan-Amos, CC BY-SA 3.0, Wikimedia Commons.

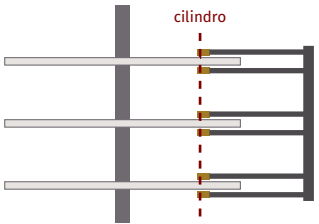


Cabezal y platos. Eric Gaba, CC BY-SA 3.0, Wikimedia Commons.

- **Interfaz**: un arreglo de sectores de 512 bytes, numerados de 0 a $n - 1$.
- Escribir *un* sector es atómico. Varios sectores no: si se va la luz queda una escritura rota *(torn write)*.
- Contrato implícito: sectores cercanos se leen más rápido que lejanos, y en orden es lo más rápido de todo.

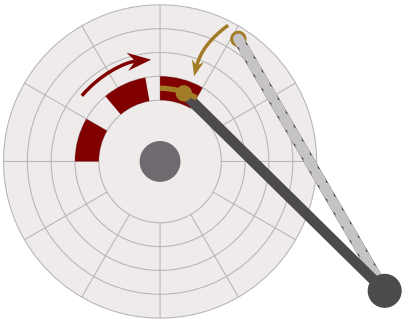
Geometría: platos, pistas, sectores, cabezales





Todos los cabezales se mueven juntos en un brazo; uno lee a la vez. Las pistas que están bajo los cabezales a la vez forman un **cilindro**.

Las tres partes de una lectura



1. Búsqueda *(seek)*: mover el brazo a la pista. Acelerar, avanzar, frenar y *asentarse*. Promedio: 4 a 10 ms.
2. Rotación *(rotational delay)*: esperar a que el sector pase bajo el cabezal. En promedio, *media vuelta*.
3. Transferencia *(transfer)*: leer mientras el sector pasa. Microsegundos para 4 KB.

Tiempo de E/S y tasa

$$T_{E/S} = T_{\text{búsqueda}} + T_{\text{rotación}} + T_{\text{transferencia}}$$

$$R_{E/S} = \frac{\text{tamaño}}{T_{E/S}}$$

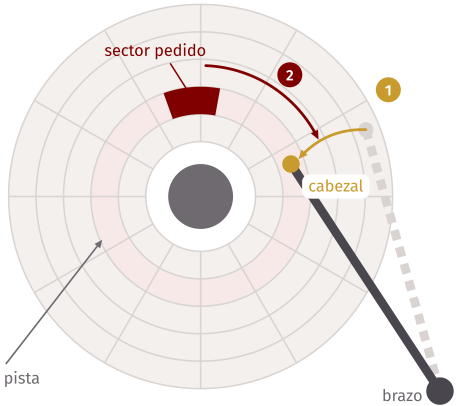
Ejemplo resuelto: aleatorio vs. secuencial

	Servidor	**Escritorio**
Velocidad · búsqueda prom. · transferencia	10 000 RPM · 4,5 ms · 150 MB/s	7 200 RPM · 9 ms · 105 MB/s
Una vuelta $= 60\,000/\text{RPM}$ ms	6,00 ms	8,33 ms
$T_{\text{rotación}}$ = media vuelta	3,00 ms	4,17 ms
$T_{\text{transf.}}$ de 4 KB $=$ 4 KB/tasa	0,027 ms	0,038 ms
Aleatorio 4 KB: $T_{E/S}$	$4{,}5 + 3{,}00 + 0{,}027 =$ **7,53** ms	$9 + 4{,}17 + 0{,}038 =$ **13,20** ms
E/S por segundo · $R_{E/S}$	133 · 0,53 MB/s	76 · 0,30 MB/s
Secuencial 100 MB: $T_{E/S}$	$4{,}5 + 3{,}00 + 666{,}7 = 674{,}2$ ms	$9 + 4{,}17 + 952{,}4 = 965{,}5$ ms
$R_{E/S}$	148,3 MB/s	103,6 MB/s
Secuencial / aleatorio	**279 veces**	**342 veces**

La regla más importante del disco duro

En aleatorio casi todo el tiempo se va en *mover* cosas, no en leer. Hay que leer y escribir en bloques grandes y secuenciales. Calculado con `code/week-12/disktime` (1 MB = 1000 KB).

Aleatorio: casi todo es posicionarse



LECTURA ALEATORIA DE 4 KB

7 200 RPM · 9 ms · 105 MB/s

1. **Búsqueda** — mover el brazo — 9 ms
2. **Rotación** — media vuelta — 4,17 ms
3. **Transferencia** — leer 4 KB — 0,038 ms

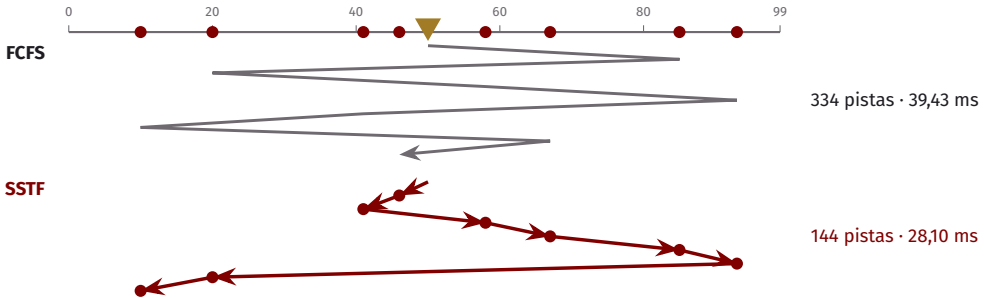


$T = 9 + 4{,}17 + 0{,}038 = 13{,}20 \text{ ms}$

casi todo es posicionarse

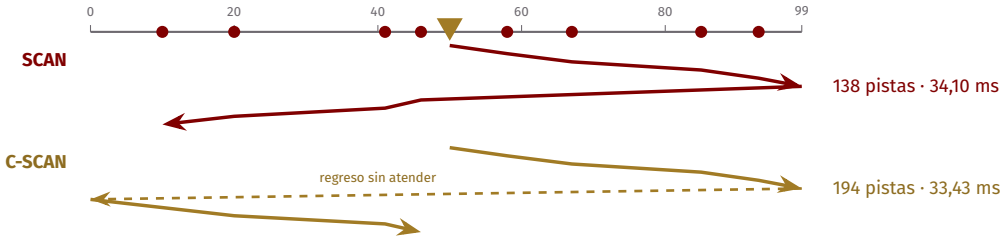
Planificación de disco: FCFS y SSTF

Cola en orden de llegada: **85, 20, 58, 93, 41, 10, 67, 46**. Cabezal en la pista 50.



SSTF *(shortest seek time first)*: siempre la pista más cercana. Mucho mejor, pero un pedido lejano puede esperar para siempre: inanición. Y el sistema operativo no conoce la geometría real; usa el número de bloque.

SCAN y C-SCAN: el ascensor



- **SCAN**: barrer en una dirección hasta el borde atendiendo en orden; luego volver. Nadie espera más de dos barridas: sin inanición. Si gira en el último pedido en vez del borde, se llama LOOK.
- **C-SCAN**: solo atiende de subida y regresa sin atender (el regreso se cuenta). Más justo con los extremos.
- SCAN recorre *menos* pistas que SSTF, pero tarda más: la rotación no aparece en la cuenta de pistas.

SPTF: lo que importa es el tiempo, no la distancia

Primera decisión desde la pista 50. Modelo: 10 000 RPM, búsqueda $0{,}5 + 0{,}05\,d$ ms, transferencia 0,1 ms.

Pista	d	Búsq.	Rotación	Total
46	4	0,70	2,63	3,43
58	8	0,90	4,60	5,60
41	9	0,95	5,55	6,60
67	17	1,35	0,65	**2,10**
10	40	2,50	1,50	4,10

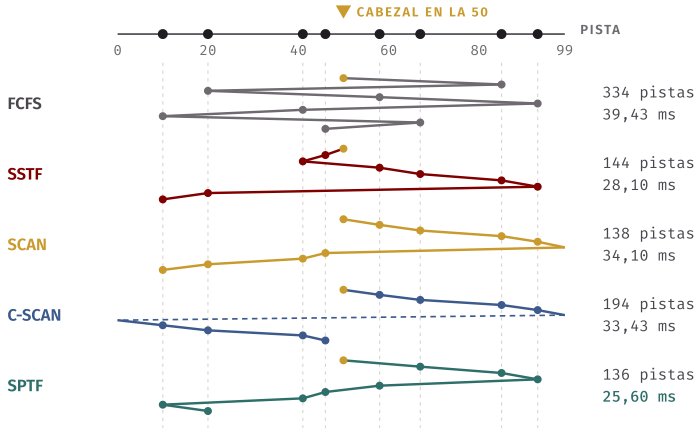
- SSTF elige la 46: la más cercana, pero el sector acaba de pasar.
- **SPTF** *(shortest positioning time first)* elige la 67: más lejos, pero el sector llega justo cuando el cabezal se asienta.

code/week-12/disksched

```
$ ./disksched
FCFS   : 85 20 58 93 41 10 67 46
         | 334 pistas, 39.43 ms
SSTF   : 46 41 58 67 85 93 20 10
         | 144 pistas, 28.10 ms
SCAN   : 58 67 85 93 46 41 20 10
         | 138 pistas, 34.10 ms
C-SCAN : 58 67 85 93 (99->0) 10 20 41 46
         | 194 pistas, 33.43 ms
SPTF   : 67 85 93 58 46 41 10 20
         | 136 pistas, 25.60 ms
```

Solo el disco puede hacer SPTF bien: conoce la geometría y la posición del plato. El sistema operativo le pasa varios pedidos (NCQ) y él elige.

Las cinco políticas, lado a lado



Detalles: la caché del disco y quién planifica

- **Desfase entre pistas** *(track skew)*: el sector 0 de cada pista está corrido; así, al cambiar de pista en una lectura secuencial, no se pierde una vuelta.
- **Zonas** *(multi-zoned)*: las pistas de afuera son más largas, tienen más sectores y se leen más rápido.
- **Caché del disco** *(track buffer)*: al leer un sector, guarda el resto de la pista.

Write-back vs. write-through

Con *write-back* el disco confirma apenas el dato está en su caché: rápido, pero si se va la luz se pierde, o llega *en otro orden*. Importa para la consistencia ante fallos (semana 14).

- El **sistema operativo** ordena y junta pedidos vecinos: los bloques 33 y 34 se vuelven un solo pedido *(I/O merging)*. Luego pasa varios al disco.
- ¿Enviar apenas llega un pedido *(work-conserving)*, o esperar un poco por si llega uno mejor *(anticipatory)*? A veces esperar gana.
- Linux hoy: `mq-deadline`, `bfq`, `kyber`, o `none` para SSD NVMe, donde no hay brazo que mover.

Resumen

- El 97 % de los bugs de concurrencia que no son deadlock son violaciones de **atomicidad** (arreglo: un lock) o de **orden** (arreglo: una variable de condición).
- **Deadlock** requiere exclusión mutua, retener y esperar, sin expropiación y espera circular. Prevenir rompe una: orden de locks, lock global, `trylock`, lock-free. **Evitar**: el banquero solo concede si el estado sigue *seguro*. **Detectar**: el mismo algoritmo con Solicitud. Linux: el *avestruz*.
- **Eventos**: un hilo, `epoll_wait` y manejadores que no bloquean. El estado se guarda a mano; las llamadas bloqueantes y los fallos de página lo frenan todo.
- **Dispositivos**: registros (MMIO o puertos), polling vs. interrupciones, DMA, y drivers. En xv6: `uart.c`, `plic.c`, `devintr` y `virtio_disk.c`.
- **Disco duro**: $T_{E/S}$ = búsqueda + rotación + transferencia. Secuencial es cientos de veces más rápido. Planificación: SSTF, SCAN/C-SCAN y SPTF.

Para la próxima clase

- **Laboratorio**: los `vector-*.c` del cap. 32 (`deadlock`, `global-order`, `try-wait`, `avoid-hold-and-wait`, `nolock`): reproducir el deadlock y medir las soluciones. `disk.py`: preguntas 1–6 del cap. 37.
- Semana 13: leer OSTEP cap. 38 *RAID* y cap. 39 *Files and Directories*.

Entregas

P4 (`pzip`) se entrega la próxima semana. Se publica P5: archivos grandes y enlaces simbólicos en xv6. El bono `cow` sigue abierto hasta la semana 16.

Preguntas para pensar

¿Qué condición del deadlock elimina ordenar los locks? ¿Y `trylock`? ¿Por qué un servidor basado en eventos sufre con los fallos de página?