

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Locks y estructuras de datos concurrentes

Exclusión mutua con ayuda del hardware y del sistema operativo

Gerson Garrido

Semana 10

Hoy

Sesión A · Locks

1. Qué le pedimos a un lock
2. Intentos: interrupciones, bandera, Peterson
3. Hardware: test-and-set, CAS, LL/SC, fetch-and-add
4. Sin girar tanto: `yield`, colas, futex, dos fases

Sesión B · Estructuras con locks

5. Contador exacto y contador aproximado
6. Listas, colas y tablas hash
7. Locks en xv6-riscv: `spinlock.c` y `sleeplock.c`

> **Lectura**
>
> OSTEP cap. 28 *Locks* y cap. 29 *Lock-based Concurrent Data Structures*. Código: `code/week-10`.

SECCIÓN 01

Qué le pedimos a un lock

01



La semana pasada: `counter++` pierde sumas

con un lock

```
1  pthread_mutex_t m =
2      PTHREAD_MUTEX_INITIALIZER;
3
4  pthread_mutex_lock(&m);
5  counter++;  // sección crítica
6  pthread_mutex_unlock(&m);
```

- Un lock es una variable con dos estados: libre o tomado.
- `lock()`: si está libre, lo toma; si no, *espera*. Solo un hilo a la vez está dentro.
- `unlock()`: lo deja libre y, si alguien espera, le da paso.
- Hoy: ¿cómo se construye `lock()`? Con qué ayuda del hardware y del sistema operativo.

Recuerden: la sección crítica *(critical section)* es el código que toca datos compartidos; la meta es la exclusión mutua *(mutual exclusion)*.

Tres criterios para evaluar un lock

Corrección

¿Garantiza la exclusión mutua? ¿Nunca hay dos hilos dentro a la vez?

Equidad

¿Todo hilo que espera termina entrando? ¿O alguno sufre inanición *(starvation)*?

Rendimiento

¿Cuánto cuesta tomarlo y soltarlo? Sin competencia, con una CPU y con muchas

Vamos a pasar cada diseño por estas tres preguntas. Spoiler: ninguno gana en las tres a la vez.

SECCIÓN 02

Primeros intentos

02

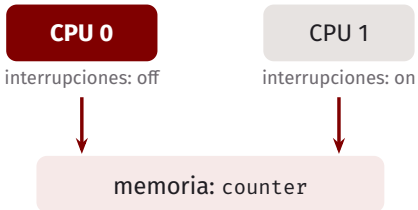
Intento 1: deshabilitar las interrupciones

primeros sistemas

```
1  void lock()   { DisableInterrupts(); }
2  void unlock() { EnableInterrupts(); }
```

Sin interrupciones no hay cambio de contexto.
Simple, pero:

- Es privilegiado: hay que confiar en que el programa las vuelva a habilitar.
- No sirve con varias CPUs: las otras siguen corriendo.
- Se pueden perder interrupciones, como el aviso de un disco.



Dónde sí se usa

En el kernel, *junto* a un lock: xv6 las apaga mientras tiene un spinlock.

Intento 2: una bandera

idea del modo flag de peterson.c

```c
int busy = 0;   // 1: tomado

void lock(void) {
    while (busy == 1)
        ;       // esperar girando
    busy = 1;   // tomarlo
}
void unlock(void) {
    busy = 0;
}
```

- Idea natural: *mirar* si está libre y *luego* tomarlo.
- Esperar en un lazo se llama girar *(spin-waiting)*: la CPU trabaja sin avanzar.
- Pero mirar y tomar son dos pasos. Entre uno y otro puede haber un cambio de contexto… o el otro hilo puede estar en otra CPU.

La bandera, intercalada

Paso	Hilo 1	Hilo 2	`busy`
1	`while (busy == 1)`: ve 0, sale		0
2	*interrupción: cambio de contexto*		0
3		`while (busy == 1)`: ve 0, sale	0
4		`busy = 1`	1
5		entra a la sección crítica	1
6	`busy = 1` (ya era 1)		1
7	entra a la sección crítica	… sigue dentro	1

Dos hilos dentro: el lock no es correcto

En el contenedor, con 2 hilos y 10^6 sumas cada uno, `./peterson flag` pierde **más de 800 000** de 2 000 000. Además, mientras espera, gira y gasta CPU.

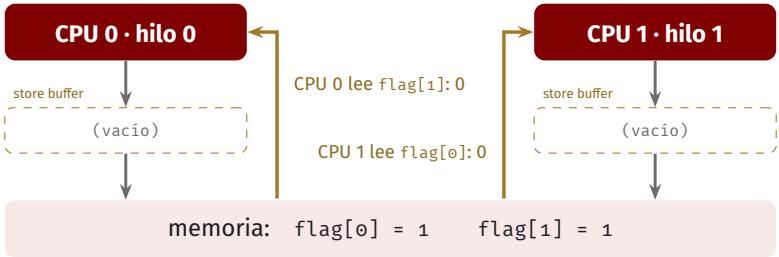
Intento 3: el algoritmo de Peterson (1981)

dos hilos: me = 0 o 1

```
1  int flag[2]; // 1: el hilo quiere
2  int turn;    // a quién se cede
3
4  void lock(int me) {
5      int other = 1 - me;
6      flag[me] = 1;  // quiero entrar
7      turn = other;  // pero pasa tú
8      while (flag[other] == 1 &&
9             turn == other)
10         ;          // esperar
11 }
12 void unlock(int me) { flag[me] = 0; }
```

- Solo usa loads y stores: ninguna instrucción especial.
- Si los dos quieren entrar, `turn` desempata: gana el que escribió `turn` primero (el otro lo sobrescribió cediéndole el paso).
- Exclusión mutua, sin inanición y para dos hilos. Se generaliza a *N*, pero es caro.
- Supone que la memoria es secuencialmente consistente: cada CPU ve las escrituras en el orden del programa. Hoy eso *no* es cierto.

Por qué Peterson falla en una CPU moderna



Los dos ven 0, salen del `while` y entran. Cuando los buffers se vacían ya es tarde. Solución: una barrera de memoria *(memory fence)* entre escribir y leer.

Peterson con y sin barrera, medido

code/week-10/peterson.c

```c
static void peterson_lock(int me) {
    int other = 1 - me;
    atomic_store_explicit(&flag[me], 1,
                          memory_order_relaxed);
    atomic_store_explicit(&turn, other,
                          memory_order_relaxed);
    if (mode == 2) // modo fence
        atomic_thread_fence(
            memory_order_seq_cst);
    while (atomic_load_explicit(&flag[other],
                    memory_order_acquire) == 1 &&
           atomic_load_explicit(&turn,
                    memory_order_relaxed) == other)
        ; // esperar
}
```

aarch64, 14 CPUs

```
$ ./peterson flag
flag: perdidas 812735 de 2000000
$ ./peterson peterson
peterson: perdidas 24 de 2000000
$ ./peterson peterson
peterson: perdidas 4 de 2000000
$ ./peterson fence
fence: perdidas 0 de 2000000
```

- Sin barrera falla pocas veces: el peor tipo de error.
- Con barrera (`dmb ish` en ARM, `fence rw,rw` en RISC-V), nunca.

SECCIÓN 03

Ayuda del hardware

03

Test-and-set: mirar y tomar en un solo paso

hardware, atómico

```
int TestAndSet(int *p, int new) {
    int old = *p; // leer
    *p = new;     // escribir
    return old;
}
```

Leer y escribir ocurren juntos: ninguna otra CPU puede meterse en medio. En RISC-V es `amoswap`; en x86, `xchg`.

Si devuelve 0, estaba libre y ahora es mío; si devuelve 1, sigo girando.

code/week-10/locks.h

```
typedef struct {
    atomic_int locked; // 0 libre, 1 tomado
} tas_lock;

void tas_acquire(tas_lock *l) {
    while (atomic_exchange_explicit(
                &l->locked, 1,
                memory_order_acquire) == 1)
        cpu_relax();
}
void tas_release(tas_lock *l) {
    atomic_store_explicit(&l->locked, 0,
        memory_order_release);
}
```

Evaluemos el spinlock

Criterio	Spinlock con test-and-set
Corrección	**sí**: el intercambio atómico deja entrar a uno solo.
Equidad	**no**: nada garantiza el orden; un hilo puede girar para siempre.
1 CPU	Malo: si desalojan al owner, los demás giran su quantum entero.
Varias CPUs	Razonable si la sección crítica es corta y hay pocos hilos.

Requisito

En una sola CPU, un spinlock solo funciona con un planificador apropiativo *(preemptive)*: sin interrupciones de reloj, el que gira no soltaría nunca la CPU.

Compare-and-swap

lo que hace el hardware, atómico

```
1  int CompareAndSwap(int *p,
2                     int expected,
3                     int new) {
4      int actual = *p;
5      if (actual == expected)
6          *p = new;  // solo si no cambi
       ó
7      return actual;
8  }
```

Un spinlock con CAS:

```
while (CompareAndSwap(&flag, 0, 1) == 1) ;
```

- "Escribe `new` solo si el valor sigue siendo `expected`".
- Para un spinlock, se comporta igual que test-and-set.
- Es más poderoso: permite actualizar estructuras sin locks *(lock-free)*: leer, calcular, y reintentar si alguien cambió el valor.
- En C11: `atomic_compare_exchange_strong`. En x86: `lock cmpxchg`.

RISC-V: load-reserved y store-conditional

rvatomics.s

```
test_and_set:
   li           a5,1
   amoswap.w.aq a5,a5,0(a0)
compare_and_swap:
1: lr.w.aq      a5,0(a0)
   bne          a5,a1,1f
   sc.w.rl      a4,a2,0(a0)
   bnez         a4,1b
1: ...
fetch_and_add:
   li           a4,1
   amoadd.w     a5,a4,0(a0)
```

`make rv` (gcc `-O2 -S`), sin directivas ni `ret`.

- `lr.w` lee y reserva la dirección.
- `sc.w` escribe *solo si* nadie escribió ahí desde el `lr.w`. Deja 0 si tuvo éxito y $\neq 0$ si falló.
- `bnez a4,1b`: si falló, reintentar todo. Así se arma CAS (y cualquier otra operación atómica).
- Los `amo*` (*atomic memory operations*) hacen el intercambio o la suma en una sola instrucción.
- `.aq`/`.rl`: semántica de *acquire* y *release*. ARM tiene el mismo par: `ldxr`/`stxr`.

Fetch-and-add y el ticket lock

code/week-10/locks.h (sin static inline)

```c
typedef struct {
    atomic_uint ticket; // siguiente número
    atomic_uint turn;   // número que se atiende
} ticket_lock;

void ticket_acquire(ticket_lock *l) {
    unsigned my = atomic_fetch_add_explicit(
        &l->ticket, 1, memory_order_relaxed);
    while (atomic_load_explicit(&l->turn,
                memory_order_acquire) != my)
        cpu_relax();
}
void ticket_release(ticket_lock *l) {
    atomic_fetch_add_explicit(&l->turn, 1,
        memory_order_release);
}
```

- `FetchAndAdd` suma 1 y devuelve el valor viejo, atómico (`amoadd.w` en RISC-V).
- Como en el banco: sacar un número y esperar a que lo llamen.
- Cada hilo que llega obtiene un número distinto. Se atiende en orden de llegada.
- Ventaja: **equidad**. Ningún hilo espera para siempre.

El ticket lock, paso a paso



Al final: A tomó el 0, B el 1, C el 2 y A volvió por el 3. B suelta (`turn = 2`) y entra C; después A. Con test-and-set, A podría haber ganado siempre.

SECCIÓN 04

Sin girar tanto

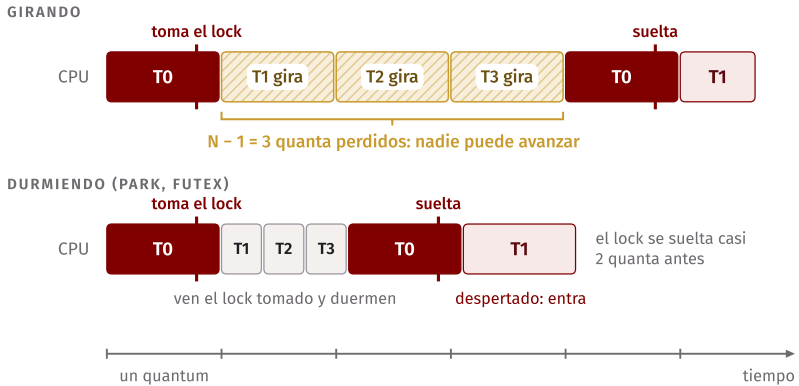
04

El costo de girar: una sola CPU



- Con el **ticket lock** es peor: solo puede entrar el hilo del número siguiente, así que *cada* entrega del lock puede costar un quantum.
- En el contenedor, con `taskset -c 0` (todo en una CPU) y 2 hilos: `tas` con 3×10^6 sumas por hilo tarda 0.013 s; `ticket` con la misma carga no terminó en 280 s.

Girar o dormir



Si los que esperan duermen, devuelven la CPU de inmediato. Eso buscan las soluciones que siguen: `yield`, `park` y futex.

Solución simple: ceder la CPU con `yield`

code/week-10/locks.h

```
1  void tas_acquire_yield(tas_lock *l) {
2      while (atomic_exchange_explicit(
3                  &l->locked, 1,
4                  memory_order_acquire) ==
       1)
5          sched_yield(); // ceder la CPU
6  }
```

Lo que falta

Elegir *quién* entra y *dormir* hasta entonces: una cola y ayuda del OS.

- Si el lock está tomado, no tiene sentido seguir girando: `yield` pasa el hilo de *ejecutando* a *listo*.
- Con 2 hilos en una CPU funciona bien: el owner vuelve a correr enseguida.
- Con 100 hilos: 99 cambios de contexto por cada vuelta. Más barato que 99 quanta, pero no gratis.
- Y sigue sin equidad: un hilo puede ceder una y otra vez para siempre.

Colas: park( ) y unpark( ) (Solaris)

lock con cola de espera

```
1  void lock(lock_t *m) {
2      while (TestAndSet(&m->guard, 1) == 1)
3          ;                  // giro corto
4      if (m->flag == 0) {
5          m->flag = 1;       // libre: es mío
6          m->guard = 0;
7      } else {
8          queue_add(m->q, gettid());
9          setpark();         // voy a dormir
10         m->guard = 0;
11         park();            // dormir
12     }
13 }
```

`guard`: spinlock que protege a `flag` y a la cola `q`.

lock con cola de espera

```
1  void unlock(lock_t *m) {
2      while (TestAndSet(&m->guard, 1) ==
         1)
3          ;
4      if (queue_empty(m->q))
5          m->flag = 0;       // nadie espera
6      else                   // pasarlo
         directo
7          unpark(queue_remove(m->q));
8      m->guard = 0;
9  }
```

- Solo se gira sobre `guard`, y la cola da equidad.
- Con alguien en la cola, `flag` sigue en 1: el lock pasa directo al que despierta.

El lock con cola, paso a paso



T2 suelta con la cola vacía: recién ahora flag vuelve a 0.

La carrera entre despertar y dormir

Paso	Hilo 1 (quiere el lock)	Hilo 2 (owner)
1	`queue_add(m->q, T1)`	
2	`m->guard = 0`	
3	*desalojado antes de* `park()`	
4		`unlock`: saca a T1 y hace `unpark(T1)`
5	`park()`: duerme… para siempre	(el aviso ya pasó)

Solaris: `setpark()`

Anuncia “estoy por dormir”. Si llega un `unpark` antes del `park`, el `park` retorna de inmediato.

xv6 hace lo mismo

`sleep_prepare(chan)` anota el canal *antes* de soltar el lock; si un `wakeup` llega en medio, borra el canal y `sleep()` no duerme.

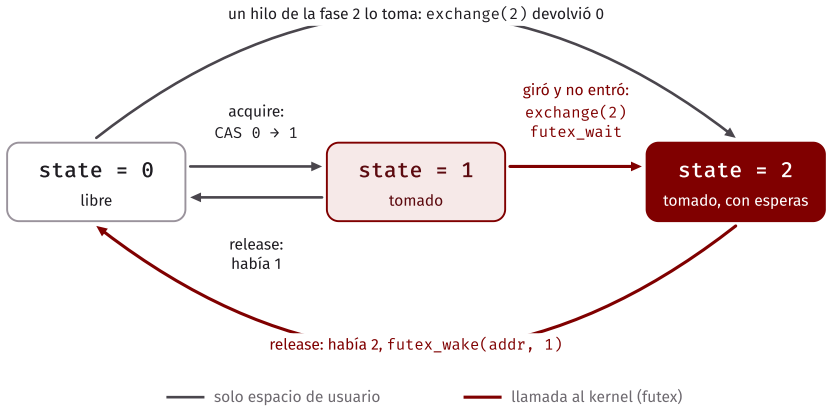
Linux: futex y locks de dos fases

code/week-10/locks.h

```c
void futex_acquire(futex_lock *l) {
    int c = 0;  // camino rápido: 0 -> 1
    if (atomic_compare_exchange_strong(
            &l->state, &c, 1))
        return;
    for (int i = 0; i < FUTEX_SPINS; i++) {
        c = 0;  // fase 1: girar un poco
        if (atomic_compare_exchange_strong(
                &l->state, &c, 1))
            return;
        cpu_relax();
    }   // fase 2: marcar 2 y dormir
    while (atomic_exchange(&l->state, 2) != 0)
        sys_futex(&l->state, FUTEX_WAIT_PRIVATE, 2);
}
```

- `futex_wait(addr, v)`: dormir solo si `*addr == v`. El kernel lo verifica atómicamente: no hay carrera.
- `futex_wake(addr, n)`: despertar a *n* hilos.
- Soltar: poner 0; si había 2, `futex_wake(addr, 1)`. Sin competencia, nunca se entra al kernel.
- **Dos fases**: girar y luego dormir. El `pthread_mutex` de glibc usa los mismos tres estados, pero gira solo una vez.

Los tres estados del futex



¿Cuál es más rápido? Medido en el contenedor

milisegundos, mediana de 3

```
$ ./bench tabla
...
tipo        1      2      4      8
atomico   2.5    9.9   34.4  211.9
mutex     6.8  148.0   89.8  355.7
tas       1.9   19.2  428.0 5512.6
yield     1.5    5.2  158.3 4202.8
ticket    2.6  211.3  509.2 1998.9
futex     2.4   50.8  428.0 1492.4
```

10^6 incrementos por hilo. Contenedor con 14 CPUs (Apple M4 Max): otra máquina da otros números. Con menos CPUs que hilos, los locks que giran salen como `--`.

- Con 1 hilo, todos cuestan unos pocos ns: sin competencia, un lock es barato.
- Con 8 hilos, `tas` es 15 veces más lento que `mutex`: todas las CPUs pelean por la misma línea de caché.
- `ticket` es justo, pero cada entrega va a un hilo concreto: si ese tarda, todos esperan.
- `pthread_mutex` (futex en glibc) es el más estable. Hasta la suma atómica sufre: el problema es compartir el contador.
- Todos dan el total exacto. La sección crítica es diminuta: el peor caso.

SECCIÓN 05

Estructuras de datos con locks

05



Un contador con un lock

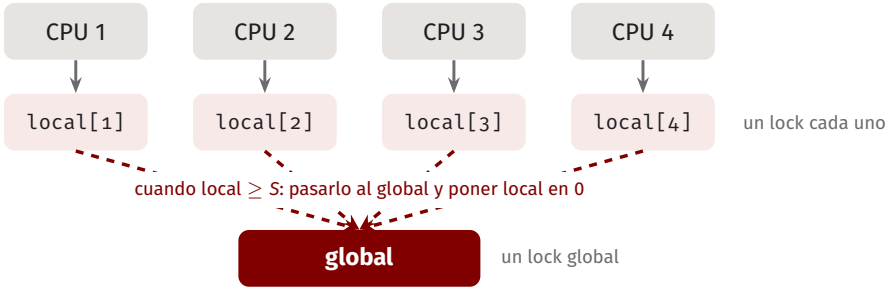
la versión obvia

```c
typedef struct {
    long value;
    pthread_mutex_t lock;
} counter_t;

void increment(counter_t *c) {
    pthread_mutex_lock(&c->lock);
    c->value++;
    pthread_mutex_unlock(&c->lock);
}
```

- Correcto y simple: empiecen siempre así.
- Pero no **escala**: con más hilos, cada incremento compite por el mismo lock y por la misma línea de caché.
- Escalar perfecto: con *N* hilos en *N* CPUs, hacer *N* veces el trabajo en el mismo tiempo.
- En el contenedor, 1 hilo: 6.8 ms. 8 hilos (8 veces el trabajo): 356 ms. Ideal: 6.8 ms.

El contador aproximado *(sloppy counter)*



- Cada CPU suma en su contador local: casi nunca compite con nadie.
- Leer el valor: leer `global`. Es aproximado: le pueden faltar hasta $\#\text{CPUs} \times (S - 1)$.
- El umbral S controla el compromiso: S chico, exacto y lento; S grande, rápido e impreciso.

Contador aproximado con $S = 5$ y 4 CPUs

Tiempo	L_1	L_2	L_3	L_4	global	valor real
0	0	0	0	0	0	0
1	1	0	1	0	0	2
2	2	0	2	1	0	5
3	3	1	3	1	0	8
4	4	1	4	2	0	11
5	$5 \to 0$	1	4	3	5	13
6	0	2	$5 \to 0$	3	10	15
7	1	3	0	4	10	18
8	2	4	1	$5 \to 0$	15	22

En el tiempo 8, `global` dice 15 pero el valor real es 22: le faltan 7 (el máximo posible es $4 \times 4 = 16$). Se tomó el lock global solo 3 veces en 22 sumas.

El contador aproximado en C

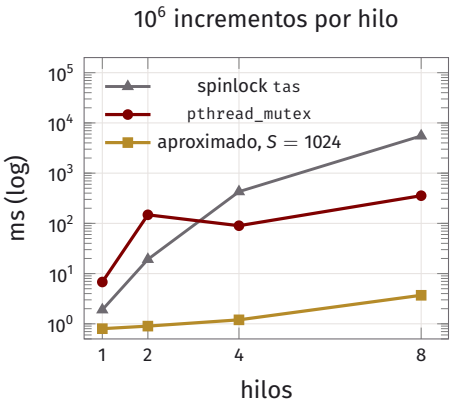
code/week-10/bench.c

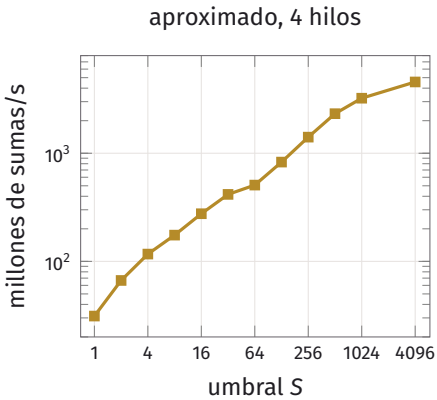
```c
struct local {
    alignas(128) long value; // su propia línea
};
static struct local local[MAXT];
static pthread_mutex_t glock; // protege counter

// en el lazo de cada hilo (id = 0, 1, ...):
if (++local[id].value >= threshold) {
    pthread_mutex_lock(&glock);
    counter += local[id].value; // al global
    pthread_mutex_unlock(&glock);
    local[id].value = 0;
}
```

- Un local por *hilo*: solo su owner lo toca, así que no necesita lock. OSTEP usa uno por CPU, con su lock.
- `alignas(128)`: cada local en su propia línea de caché (128 bytes en el Apple M4 del contenedor).
- Sin eso, varios locales comparten una línea y las CPUs se la pasan de mano en mano aunque no compartan datos: *false sharing*.

¿Escala? Medido en el contenedor





Con $S = 1$ es un contador con lock (31 millones de sumas/s); con $S = 4096$, más de 4500 millones.
El precio: un valor global menos exacto.

Una lista enlazada concurrente

code/week-10/hash.c

```
1  static int list_insert(list_t *l, long key,
2                         int use_lock) {
3      node_t *n = malloc(sizeof *n); // sin lock
4      if (n == NULL)
5          return -1;
6      n->key = key;
7      if (use_lock)
8          pthread_mutex_lock(&l->lock);
9      n->next = l->head;
10     l->head = n;
11     if (use_lock)
12         pthread_mutex_unlock(&l->lock);
13     return 0;
14 }
```

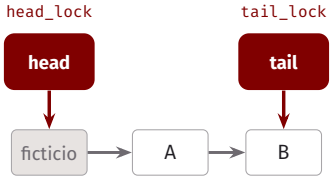
- Un lock para toda la lista: correcto y simple.
- `malloc` va fuera del lock: no toca la lista y es thread-safe. Sección crítica más corta.
- Y si `malloc` falla, salimos *sin* tener el lock tomado.
- En `list_lookup`, un solo camino de salida: olvidar el `unlock` en un `return` temprano es un error clásico.

Hand-over-hand: un lock por nodo



- Al avanzar, tomar el lock del siguiente nodo antes de soltar el del actual. Varios hilos recorren la lista a la vez, uno detrás de otro.
- Medido (`./list`, 1000 nodos, 4 hilos): un lock 0.108 s; hand-over-hand 0.714 s. Tomar y soltar un lock por nodo cuesta más de lo que se gana.

Una cola con dos locks (Michael y Scott)

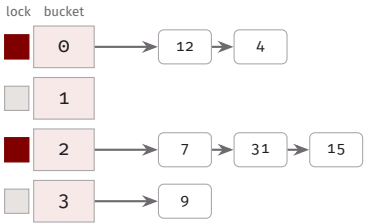


- Sacar toca `head`; meter toca `tail`.
- El nodo ficticio *(dummy)* separa los extremos: meter y sacar ocurren a la vez.
- Con la cola vacía, ambos tocan `next` del ficticio: se publica con release/acquire.
- ¿Cola vacía? Esperar algo: semana 11.

code/week-10/queue.c (resumido)

```
1  // meter: solo tail_lock
2  pthread_mutex_lock(&q->tail_lock);
3  __atomic_store_n(&q->tail->next, n,
4                   __ATOMIC_RELEASE);
5  q->tail = n;
6  pthread_mutex_unlock(&q->tail_lock);
7  // sacar: solo head_lock
8  pthread_mutex_lock(&q->head_lock);
9  node_t *old = q->head;
10 node_t *first = __atomic_load_n(
11     &old->next, __ATOMIC_ACQUIRE);
12 if (first != NULL) {
13     *value = first->value;
14     q->head = first; // nuevo ficticio
15 }
16 pthread_mutex_unlock(&q->head_lock);
```

Una tabla hash: un lock por bucket



dos hilos a la vez,
en buckets distintos

- Cada bucket es una lista con su propio lock.
- Dos hilos solo compiten si caen en el mismo bucket (1 en 1024).

hilos	1	2	4	8
lock global (ms)	1	8	52	391
lock por bucket (ms)	1	3	6	19

`./hash global|bucket HILOS`: cada hilo inserta y busca 20 000 claves.

- Con 8 hilos, 20 veces más rápida. La tabla hash es la estructura que mejor se reparte: el lock es pequeño y casi nunca se comparte.
- Por bucket aún crece algo: las listas se alargan (más claves en total) y `malloc` también tiene sus locks.

Consejos para estructuras concurrentes

- **Primero correcto, después rápido.** Un lock grande alrededor de todo es un buen comienzo; Linux y Solaris empezaron así (el *big kernel lock*).
- **Midan antes de optimizar.** Hand-over-hand parece más concurrente y resultó más lento.
- **Más concurrencia no siempre es más rápido**: más locks, más costo fijo por operación.
- **Cuidado con los caminos de salida**: cada `return` dentro de una sección crítica debe soltar el lock.
- **Ojo con el hardware**: false sharing, barreras de memoria. Un `volatile` no es un lock.

> **Knuth**
>
> "La optimización prematura es la raíz de todos los males."

SECCIÓN 06

Locks en xv6-riscv

06



kernel/spinlock.c: acquire y release

kernel/spinlock.c (reformateado)

```
1  void acquire(struct spinlock *lk) {
2    push_off(); // disable interrupts ...
3    if (holding(lk))
4      panic("acquire");
5    while (__atomic_exchange_n(&lk->locked,
6             1, __ATOMIC_ACQUIRE) != 0)
7      ;
8    lk->cpu = mycpu();
9  }
```

kernel/spinlock.c (reformateado)

```
1  void release(struct spinlock *lk) {
2    if (!holding(lk))
3      panic("release");
4    lk->cpu = 0;
5    __atomic_store_n(&lk->locked, 0,
6                     __ATOMIC_RELEASE);
7    pop_off();
8  }
```

- El lazo es un spinlock con test-and-set: `__atomic_exchange_n` escribe 1 y devuelve el valor viejo.
- `__ATOMIC_ACQUIRE`/`__ATOMIC_RELEASE`: ni el compilador ni la CPU sacan accesos de la sección crítica fuera de ella.
- `holding`: tomar dos veces el mismo lock en la misma CPU es un error; xv6 hace `panic` en vez de colgarse.
- Versiones anteriores usaban `__sync_lock_test_and_set` y `__sync_synchronize`; la idea es la misma.

Lo que compila: `amoswap.w.aq`

bash

```
$ riscv64-linux-gnu-objdump -d kernel/kernel
00000000800000bdc <acquire>:
  80000bdc: jal   80000b9a <push_off>
  80000be2: jal   80000b70 <holding>
  80000be6: li    a4,1
  80000be8: bnez  a0,80000c04 <acquire+0x34>
  80000bea: mv    a5,a4
  80000bec: amoswap.w.aq a5,a5,(s1)
  80000bf0: sext.w a5,a5
  80000bf2: bnez  a5,80000bea <acquire+0x1a>
  80000bf4: jal   800018c6 <mycpu>
00000000800000c5c <release>:
  80000c6e: sd    zero,16(s1)
  80000c72: fence rw,w
  80000c76: sw    zero,0(s1)
  80000c7a: jal   80000c10 <pop_off>
```

- El giro son cuatro instrucciones, de `80000bea` a `80000bf2`.
- `amoswap.w.aq a5,a5,(s1)`: guarda 1 en `lk->locked` y deja el valor viejo en `a5`. Si no era 0, `bnez` vuelve atrás.
- `release`: `sd zero,16(s1)` es `lk->cpu = 0`. Luego `fence rw,w` y un `sw` normal: todo lo escrito en la sección crítica se ve *antes* que el lock libre.
- Copia de xv6-riscv compilada en el contenedor (gcc 13.3). Se omiten bytes y algunas líneas.

¿Por qué apagar las interrupciones con un spinlock?



- `sys_sleep` (`kernel/sysproc.c`) toma `tickslock` para leer `ticks`.
- El manejador `clockintr` (`kernel/trap.c`) quiere el *mismo* lock para hacer `ticks++`.
- El owner está debajo, en la *misma CPU*, y no volverá a correr hasta que el manejador termine: deadlock.
- Regla de xv6: mientras una CPU tiene *cualquier* spinlock, sus interrupciones están apagadas. `push_off` y `pop_off` se anidan con un contador (`noff`) y recuerdan si estaban encendidas (`intena`).

`push_off` y `pop_off`

kernel/spinlock.c (reformateado)

```
1  void push_off(void) {
2    uint64 flags = rc_sstatus(SSTATUS_SIE);
3    int old = !!(flags & SSTATUS_SIE);
4    if (mycpu()->noff == 0)
5      mycpu()->intena = old;
6    mycpu()->noff += 1;
7  }
```

kernel/spinlock.c (reformateado)

```
1  void pop_off(void) {
2    struct cpu *c = mycpu();
3    if (intr_get())
4      panic("pop_off - interruptible");
5    if (c->noff < 1)
6      panic("pop_off");
7    c->noff -= 1;
8    if (c->noff == 0 && c->intena)
9      intr_on();
10 }
```

- `rc_sstatus` es un `csrrc`: lee `sstatus` y borra el bit `SIE` en una sola instrucción.
- Con dos locks tomados, `noff` vale 2. Las interrupciones vuelven al soltar el último, y solo si estaban encendidas antes (`intena`).
- Los spinlocks de xv6 son para secciones cortas: no se duerme con uno tomado. `sched()` hace `panic` si `noff != 1` (solo `p->lock`).

`kernel/sleeplock.c`: un lock que duerme

kernel/sleeplock.c (reformateado)

```
1  void acquiresleep(struct sleeplock *lk) {
2    acquire(&lk->lk);
3    while (lk->locked) {
4      sleep_prepare(lk);
5      release(&lk->lk);
6      sleep();
7      acquire(&lk->lk);
8    }
9    lk->locked = 1;
10   lk->pid = myproc()->pid;
11   release(&lk->lk);
12 }
```

kernel/sleeplock.c (reformateado)

```
1  void releasesleep(struct sleeplock *lk) {
2    acquire(&lk->lk);
3    lk->locked = 0;
4    lk->pid = 0;
5    wakeup(lk);
6    release(&lk->lk);
7  }
```

- `lk->lk` hace de `guard`, `sleep` de `park` y `wakeup` de `unpark`.
- `sleep_prepare` *antes* de soltar `lk->lk`: como `setpark`.
- Para esperas largas: buffers del disco (`kernel/bio.c`), inodos (`kernel/fs.c`).

Resumen

- Un lock se evalúa por **corrección**, **equidad** y **rendimiento**.
- Apagar interrupciones no sirve con varias CPUs. Una bandera no es atómica. Peterson funciona solo con **barreras de memoria**.
- El hardware da instrucciones atómicas: **test-and-set** (`amoswap`), **compare-and-swap** (con `lr`/`sc`), **fetch-and-add** (`amoadd`). El **ticket lock** agrega equidad.
- Girar desperdicia CPU: `yield`, colas con `park`/`unpark`, **futex** y locks de **dos fases**.
- Estructuras: el **contador aproximado** escala con un umbral S; listas, colas con dos locks y tablas hash con un lock por bucket. Primero correcto, después medir.
- xv6: `acquire` es un `amoswap.w.aq` en un lazo, con las interrupciones apagadas; `sleeplock` duerme.

Para la próxima clase

- **Laboratorio**: `x86.py` con `flag.s`, `test-and-set.s`, `peterson.s`, `ticket.s` y `yield.s` (cap. 28). Medir el contador con un lock y el aproximado con 1, 2, 4 y 8 hilos, y graficar (`code/week-10/bench`).
- Semana 11: leer OSTEP cap. 30 *Condition Variables* y cap. 31 *Semaphores*.

Entregas

P3 (lab `pgtbl`) se entrega esta semana. Se publica P4: `pzip`, compresión en paralelo con hilos. Entrega en la semana 13.

Preguntas para pensar

¿Por qué un spinlock no es justo y cómo lo arregla el ticket lock? ¿Qué pasa con un spinlock en una sola CPU con planificación apropiativa? ¿Qué parámetro del contador aproximado controla el compromiso entre exactitud y rendimiento?