

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Variables de condición y semáforos

Esperar a que algo pase, sin girar

Gerson Garrido

Semana 11

Hoy

Sesión A · Variables de condición

1. Esperar sin girar: `wait` y `signal`
2. El padre espera al hijo, y dos versiones rotas
3. Productor/consumidor, paso a paso
4. Semántica Mesa vs. Hoare; `broadcast`

Sesión B · Semáforos

5. `sem_wait` y `sem_post`: lock y orden
6. Productor/consumidor; lectores/escritores
7. Filósofos, barreras; inversión de prioridad
8. `sleep` y `wakeup` en xv6-riscv

Lectura

OSTEP cap. 30 *Condition Variables* y cap. 31 *Semaphores*. Opcional: el apéndice *Monitors*. Código: `code/week-11`.

SECCIÓN 01

Variables de condición

01



Esperar girando desperdicia la CPU

esperar con una bandera

```c
volatile int done = 0;

void *child(void *arg) {
    printf("hijo\n");
    done = 1;
    return NULL;
}

int main(void) {
  pthread_t c;
  pthread_create(&c, NULL, child, NULL)
      ;
  while (done == 0)
    ;                // girar
  printf("padre: fin\n");
}
```



- Con una CPU, el padre gasta todo su quantum: el hijo, que es quien puede cambiar `done`, ni siquiera corre.
- Con varias CPUs, una queda al 100 % sin hacer nada útil.
- Queremos: dormir hasta que la condición cambie, y que quien la cambia nos despierte.

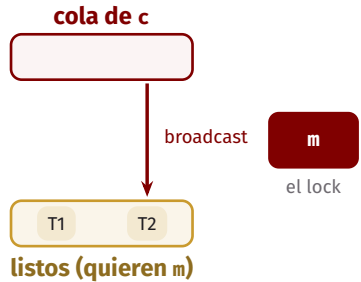
Sin `volatile`, con `-O2` el compilador puede leer `done` una sola vez y girar para siempre.

Variable de condición: `wait` y `signal`

Variable de condición *(condition variable)*

Una cola de hilos dormidos que esperan a que un estado cambie. No guarda el estado.

- `pthread_cond_wait(&c, &m)`: con `m` tomado, suelta `m` y duerme en `c`, en un solo paso. Antes de retornar, vuelve a tomar `m`.
- `pthread_cond_signal(&c)`: despierta a uno. Si nadie espera, no pasa nada: el aviso no se guarda.
- `pthread_cond_broadcast(&c)`: despierta a todos.



`signal` saca a uno de la cola y `broadcast` a todos. Al salir de la cola, cada hilo compite otra vez por `m`.

¿Por qué `wait` suelta el lock?



Si durmiera con el lock

El hijo no podría tomar `m` para cambiar `done` y nadie despertaría al padre: deadlock.

Y en un solo paso

Si soltara `m` y *luego* durmiera, el `signal` podría caer en medio: la carrera de `setpark` (semana 10).

El padre espera al hijo: la versión correcta

join.c (sin trazas)

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit(void) {  // hijo
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void thr_join(void) {  // padre
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

Tres piezas que trabajan juntas:

- **Estado**: `done`. Es lo que de verdad importa.
- **Lock**: `m` protege `done`.
- **CV**: `c`, solo para dormir y despertar.

- **Padre primero**: ve `done == 0` y duerme. El hijo pone `done = 1` y hace `signal`. El padre despierta, revisa otra vez y sale.
- **Hijo primero**: pone `done = 1`; su `signal` no despierta a nadie. El padre ve `done == 1` y ni duerme.

¿Hace falta cada pieza? Quitemos una a la vez.

Roto 1: sin variable de estado

sin done

```c
void thr_exit(void) {
    pthread_mutex_lock(&m);
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void thr_join(void) {
    pthread_mutex_lock(&m);
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

	Padre	Hijo
1	crea al hijo; lo desalojan	
2		`lock`, `signal`: nadie espera
3		`unlock`; termina
4	`lock`, `wait`	
5	duerme para siempre	

El aviso se perdió

`signal` no recuerda nada. Si el hijo avisa antes, el padre no tiene cómo saberlo: necesita un *estado* que revisar.

Roto 2: el estado sin lock

sin lock al revisar

```c
void thr_exit(void) {
    done = 1;
    pthread_cond_signal(&c);
}

void thr_join(void) {
    if (done == 0) {
        // desalojado aquí
        pthread_mutex_lock(&m);
        pthread_cond_wait(&c, &m);
        pthread_mutex_unlock(&m);
    }
}
```

	Padre	Hijo
1	`done == 0`: SÍ	
2	lo desalojan antes del `wait`	
3		`done = 1`
4		`signal`: nadie espera
5	`wait`: duerme para siempre	

Revisar y dormir no fueron atómicos

Con el lock tomado del `if` al `wait`, el hijo no puede cambiar `done` en medio. Por eso `wait` recibe el lock.

Lo comprobamos: `code/week-11/join.c`

join ok · join nostate

```
$ ./join ok
[   0 ms] padre: done == 0, wait
[   0 ms] hijo:  trabajo terminado
[   0 ms] hijo:  done = 1, signal
[   0 ms] padre: el hijo terminó
$ ./join nostate
[   0 ms] hijo:  trabajo terminado
[   0 ms] hijo:  signal
[  50 ms] padre: wait (no hay estado...)
[1057 ms] WATCHDOG: 1 s sin avance.
          El padre duerme: el aviso se perdió.
```

join nolock

```
$ ./join nolock
[   0 ms] padre: done == 0 (revisado sin lock)
[  54 ms] hijo:  trabajo terminado
[  54 ms] hijo:  done = 1, signal (sin lock)
[ 112 ms] padre: wait
[1055 ms] WATCHDOG: 1 s sin avance.
          El padre duerme: el aviso se perdió.
```

- Unos `msleep` fuerzan la intercalación mala; un hilo *watchdog* detecta que nadie avanza en 1 s y termina el programa.
- Sin el retraso, el error aparece de vez en cuando. Así son los bugs de concurrencia: las pruebas pasan casi siempre.

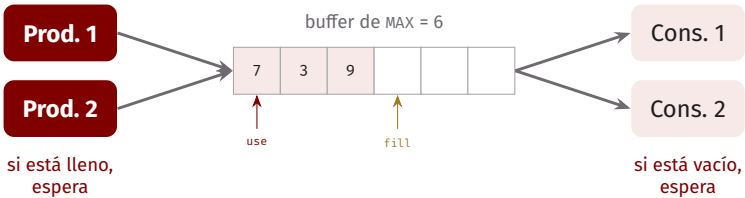
SECCIÓN 02

Productor/consumidor

02



El problema del bounded buffer



- `grep main *.c | wc -l`: `grep` produce, `wc` consume. El pipe del kernel es un bounded buffer (en xv6, de 512 bytes).
- Un servidor web: un hilo acepta conexiones y las pone en una cola; varios hilos trabajadores las sacan.
- Dos condiciones de espera: lleno para el productor y vacío para el consumidor. Empecemos con un buffer de *un* elemento.

Intento 1: `if` y una sola CV

productor

```
1  void *producer(void *arg) {
2    for (int i = 0; i < loops; i++) {
3      mutex_lock(&mutex);           // p1
4      if (count == 1)               // p2
5        cond_wait(&cond, &mutex);   // p3
6      put(i);                       // p4
7      cond_signal(&cond);           // p5
8      mutex_unlock(&mutex);         // p6
9    }
10 }
```

consumidor

```
1  void *consumer(void *arg) {
2    for (;;) {
3      mutex_lock(&mutex);           // c1
4      if (count == 0)               // c2
5        cond_wait(&cond, &mutex);   // c3
6      int tmp = get();              // c4
7      cond_signal(&cond);           // c5
8      mutex_unlock(&mutex);         // c6
9    }
10 }
```

Buffer de un elemento: `count` vale 0 o 1. `put()` exige `count == 0` y `get()` exige `count == 1` (con `assert`).

`mutex_lock`, `cond_wait`... abrevian las funciones `pthread_*`.

- Con un productor y un consumidor, funciona.
- ¿Con un productor y dos consumidores?

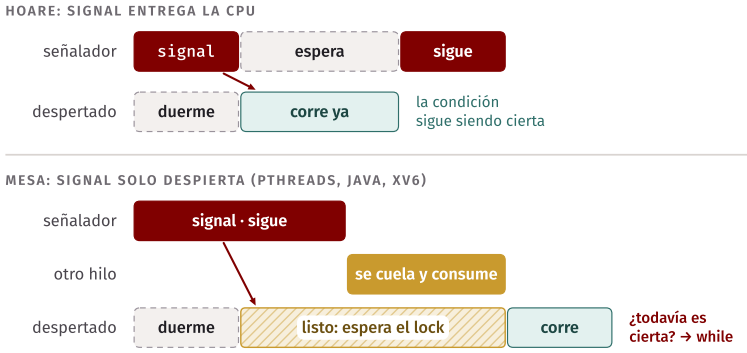
La intercalación que rompe el `if`

	Tc1	Tc2	Tp	count	Qué pasa
1	`c1 c2 c3` duerme			0	Vacío: Tc1 duerme.
2	duerme		`p1 p2 p4 p5` corre	1	Tp pone un dato y hace `signal`.
3	listo		`p6 p1 p2 p3` duerme	1	Tc1 está listo, pero no corre. Tp ve lleno y duerme.
4	listo	`c1 c2 c4` corre	duerme	0	Tc2 llega antes y se lleva el dato.
5	listo	`c5 c6` corre	listo	0	Tc2 despierta a Tp.
6	`c4` corre			0	Tc1 sigue tras `c3`: `get()` con el buffer vacío. ¡Falla el `assert`!

El problema

Entre el `signal` y el momento en que Tc1 corre, el estado cambió. Despertar es solo una *pista*: hay que volver a revisar. **Solución: `while`.**

Semántica Mesa vs. Hoare



- **Hoare**: `signal` entrega el lock y la CPU al despertado. Fácil de razonar, difícil de implementar bien.
- **Mesa**: `signal` solo pasa al despertado a la cola de listos. Lo usan pthreads, Java, Linux y xv6. POSIX permite además despertares espurios *(spurious wakeups)*: `wait` puede retornar sin ningún `signal`.
- Regla con Mesa: siempre `while`. Con `while` el código es correcto también bajo Hoare.

Intento 2: `while` con una sola CV

	Tc1	Tc2	Tp	count	Qué pasa
1	`c1 c2 c3` duerme			0	Tc1 duerme.
2	duerme	`c1 c2 c3` duerme		0	Tc2 también.
3	listo	duerme	`p1 p2 p4 p5` corre	1	Tp pone un dato y despierta a Tc1.
4	listo	duerme	`p6 p1 p2 p3` duerme	1	Lleno: Tp duerme.
5	`c2 c4 c5` corre	listo	duerme	0	Tc1 consume y hace `signal`… ¡despierta a Tc2, no a Tp!
6	`c6 c1 c2 c3` duerme	listo	duerme	0	Vacío: Tc1 vuelve a dormir.
7	duerme	`c2 c3` duerme	duerme	0	Tc2 revisa: vacío. Los tres duermen.

El problema

Con una sola CV, un consumidor puede despertar a otro consumidor. Los consumidores deben despertar solo a productores, y viceversa: **dos CVs**.

La solución: `while` y dos CVs

code/week-11/pc.c (reformateado)

```c
void produce(int value) {
  pthread_mutex_lock(&mutex);
  while (count == max)
    pthread_cond_wait(&empty, &mutex);
  put(value);
  pthread_cond_signal(&fill);
  pthread_mutex_unlock(&mutex);
}
```

code/week-11/pc.c (reformateado)

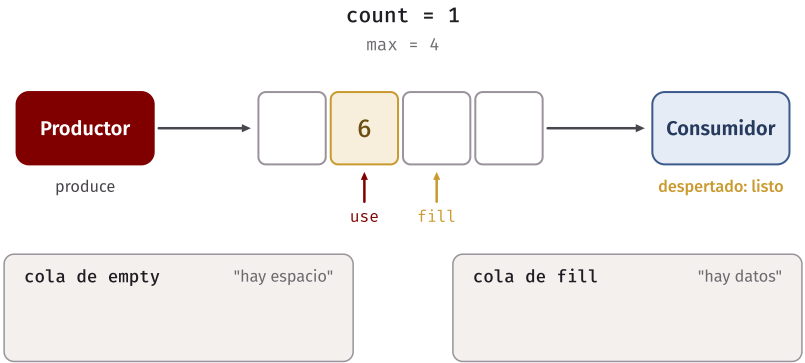
```c
int consume(void) {
  pthread_mutex_lock(&mutex);
  while (count == 0)
    pthread_cond_wait(&fill, &mutex);
  int tmp = get();
  pthread_cond_signal(&empty);
  pthread_mutex_unlock(&mutex);
  return tmp;
}
```

- Productores duermen en `empty` (“hay espacio”); consumidores, en `fill` (“hay datos”).
- Buffer circular de `max` celdas: se llenan varias sin esperar.

buffer 4, 2 prod., 3 cons.

```
$ ./pc 4 2 3 100000
esperado 99900000, obtenido 99900000: OK
```

Dos CVs en acción: buffer de 4 celdas



El productor pone el 6 y hace signal(&fill): el consumidor despierta.

Lo comprobamos: `code/week-11/pcbug.c`

if + una CV (recortada)

```
$ ./pcbug if
[   0 ms] Tc1: buffer vacío, wait
[  26 ms] Tc2: pide el lock
[  40 ms] Tp : put(1), signal
[  60 ms] Tc2: consume 1 (se coló)
[  60 ms] Tc1: despierta con el lock
[  60 ms] Tc1: get() con el buffer VACÍO
resultado: ERROR, un consumidor leyó...
```

Con `./pcbug while`, Tc1 revisa, vuelve a dormir y consume el siguiente dato: OK.

while + una CV (recortada)

```
$ ./pcbug onecv
[   0 ms] Tc1: buffer vacío, wait
[   6 ms] Tc2: buffer vacío, wait
[  22 ms] Tp : put(1), signal
[  22 ms] Tp : buffer lleno, wait
[  22 ms] Tc1: consume 1, signal
[  22 ms] Tc1: buffer vacío, wait
[  22 ms] Tc2: buffer vacío, wait
[1133 ms] WATCHDOG: 1 s sin avance.
```

Con `./pcbug twocv`, el `signal` de Tc1 va a `empty` y despierta a Tp: OK.

Los retrasos fuerzan el orden de llegada al mutex. En el contenedor, 10 de 10 corridas reproducen cada error.

Covering conditions: cuando no se sabe a quién despertar

un asignador de memoria

```
1  int bytes_left = MAX_HEAP;
2  void *allocate(int size) {
3    pthread_mutex_lock(&m);
4    while (bytes_left < size)
5      pthread_cond_wait(&c, &m);
6    void *ptr = ...;  // reservar
7    bytes_left -= size;
8    pthread_mutex_unlock(&m);
9    return ptr;
10 }
11 void release(void *ptr, int size) {
12   pthread_mutex_lock(&m);
13   bytes_left += size;
14   pthread_cond_signal(&c); // ¿a quién?
15   pthread_mutex_unlock(&m);
16 }
```

Con `bytes_left = 0`:

1. Ta pide 100 y duerme. Tb pide 10 y duerme.
2. Tc libera 50 y hace `signal`: despierta a Ta.
3. Ta necesita 100: vuelve a dormir. Tb, que sí cabía, sigue dormido.

Covering condition

Con `broadcast` despiertan *todos*; cada uno revisa su condición y el que no puede avanzar vuelve a dormir.

Costo: se despiertan hilos de más. Úsenlo cuando una sola CV representa condiciones distintas.

`signal` o `broadcast`

CON SIGNAL: DESPIERTA UNO, Y FUE EL EQUIVOCADO



CON BROADCAST: DESPIERTAN TODOS Y CADA UNO REVISA



Monitores y cuatro reglas

Un monitor *(monitor)* es un objeto cuyos métodos corren con un lock implícito, más variables de condición. Lo propusieron Brinch Hansen y Hoare; Java lo trae en el lenguaje:

BoundedBuffer.java

```
1  synchronized void put(int v)
2          throws InterruptedException {
3      while (count == MAX)
4          wait();          // suelta el lock
5      buf[fill] = v;
6      fill = (fill + 1) % MAX;
7      count++;
8      notifyAll();         // broadcast
9  }
```

Java tiene una sola CV por objeto: por eso se usa `notifyAll`.

Reglas para usar variables de condición

1. Toda espera tiene un **estado** explícito (`done`, `count`…).
2. Ese estado se lee y se cambia con el **lock** tomado.
3. `wait` siempre dentro de un **`while`**.
4. Una CV por condición; si no se puede, `broadcast`.

Además: hagan el `signal` con el lock tomado. No siempre es obligatorio, pero siempre es correcto.

SECCIÓN 03

Semáforos

03

Semáforo: un entero con dos operaciones

semáforos POSIX

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // 0: entre hilos
                    // 1: valor inicial
sem_wait(&s); // s--; si s < 0, duerme
sem_post(&s); // s++; despierta a uno
```

Semáforo *(semaphore)* (Dijkstra)

Si el valor es negativo, su magnitud es cuántos hilos esperan.

Valor inicial 1 (binario: un lock), tres hilos:

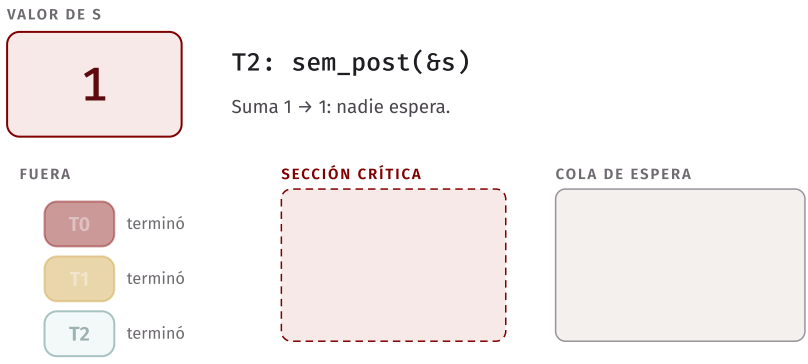
Operación	s	Efecto
	1	libre
T0: `sem_wait`	0	T0 entra
T1: `sem_wait`	−1	T1 duerme
T2: `sem_wait`	−2	T2 duerme: dos esperan
T0: `sem_post`	−1	despierta a T1, que entra
T1: `sem_post`	0	despierta a T2, que entra
T2: `sem_post`	1	nadie espera: libre



Edsger W. Dijkstra los propuso en los años 60, junto con el problema de los filósofos comensales.

E. W. Dijkstra, 2002. Hamilton Richards, CC BY-SA 3.0, Wikimedia Commons.

El semáforo, paso a paso



Valor negativo: su magnitud es cuántos hilos esperan (definición de Dijkstra).

Semáforo para ordenar: el padre espera al hijo

`sem_init(&s, 0, 0);`

Padre: `create(child);`
`sem_wait(&s);`

Hijo: `...; sem_post(&s);`

El valor inicial

Es cuántos pueden pasar *sin esperar*: lock, 1; orden, 0; *K* recursos, *K*.

Caso 1: el padre llega primero

padre: `sem_wait`	−1	duerme
hijo: `sem_post`	0	despierta al padre

Caso 2: el hijo llega primero

hijo: `sem_post`	1	nadie espera: queda anotado
padre: `sem_wait`	0	no duerme

A diferencia de `signal`, el semáforo recuerda el aviso: el valor es el estado. Es el “roto 1” de la sesión A, arreglado.

Productor/consumidor con semáforos

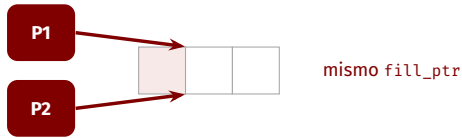
primer intento

```c
sem_t empty, full;
sem_init(&empty, 0, MAX); // libres
sem_init(&full, 0, 0);    // datos

void produce(int v) {
    sem_wait(&empty); // esperar espacio
    put(v);
    sem_post(&full);  // un dato más
}

int consume(void) {
    sem_wait(&full);  // esperar dato
    int v = get();
    sem_post(&empty); // un espacio más
    return v;
}
```

- `empty` cuenta los espacios libres y `full` los datos. No hace falta `count`: los semáforos son el estado.
- Con un productor y un consumidor, funciona con cualquier `MAX`.
- Con `MAX > 1` y dos productores: los dos pasan `sem_wait(&empty)` y entran a `put` a la vez.

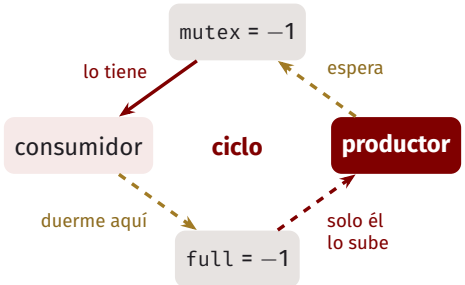


Los dos escriben en la misma celda: se pierde un dato. Falta exclusión mutua alrededor de `put` y `get`.

¿Dónde va el mutex? Afuera: deadlock

mutex afuera (mal)

```
1  void produce(int v) {
2      sem_wait(&mutex);  // primero
3      sem_wait(&empty);
4      put(v);
5      sem_post(&mutex);
6      sem_post(&full);
7  }
8
9  int consume(void) {
10     sem_wait(&mutex);  // primero
11     sem_wait(&full);
12     int v = get();
13     sem_post(&mutex);
14     sem_post(&empty);
15     return v;
16 }
```



El consumidor espera un dato que solo el productor puede poner, y el productor espera el mutex que tiene el consumidor: deadlock.

El mutex adentro: `code/week-11/sempc.c`

mutex adentro (bien)

```c
void produce(int v) {
    sem_wait(&empty); // esperar
    sem_wait(&mutex); // excluir
    put(v);
    sem_post(&mutex);
    sem_post(&full);
}

int consume(void) {
    sem_wait(&full);
    sem_wait(&mutex);
    int v = get();
    sem_post(&mutex);
    sem_post(&empty);
    return v;
}
```

- Primero se espera la condición; recién después se toma el mutex, solo para la sección crítica.
- Regla: no duerman en un semáforo con un lock tomado. Nadie más podrá entrar a despertarlos.

2 productores, 2 consumidores

```
$ ./sempc ok
consumidos 100000 de 100000: OK
$ ./sempc outside
[    0 ms] Co: tiene el mutex, sem_wait(&full)
[   60 ms] P1: sem_wait(&mutex)
[   60 ms] Po: sem_wait(&mutex)
[1049 ms] WATCHDOG: 1 s sin avance.
          Deadlock: C duerme en full con el mutex.
```

Lock de lectores/escritores

code/week-11/rwlock.c (reformateado)

```c
void acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // el primero
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // el último
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

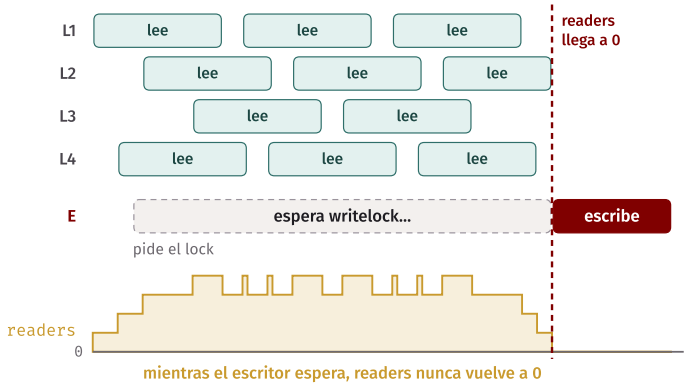
estado y escritores

```c
typedef struct {
  sem_t lock;      // para readers
  sem_t writelock; // empieza en 1
  int readers;
} rwlock_t;

void acquire_writelock(rwlock_t *rw)
    { sem_wait(&rw->writelock); }
void release_writelock(rwlock_t *rw)
    { sem_post(&rw->writelock); }
```

- Muchas lecturas a la vez; una escritura va sola.
- El primer lector toma `writelock` por el grupo; el último lo suelta.
- ¿Y si los lectores nunca dejan de llegar?

Inanición del escritor



Con lectores que se solapan, `readers` nunca vuelve a 0: el escritor espera indefinidamente. Es inanición *(starvation)*.

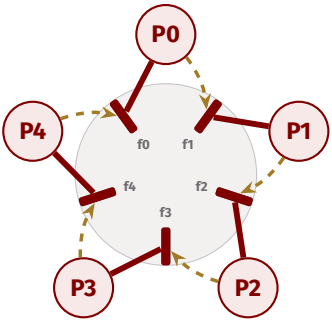
Inanición del escritor: `code/week-11/rwlock.c`

- Con un solo lector, el escritor espera poco. Con ocho que se solapan, espera hasta que los lectores se detienen.
- Arreglo: cuando un escritor espera, los lectores nuevos deben formarse detrás. Es parte del laboratorio.

lectores durante 1 s (recortada)

```
$ ./rwlock 1
[  54 ms] escritor: entra tras esperar 2 ms
$ ./rwlock 8
[  54 ms] escritor: pide el lock
[1001 ms] los lectores se detienen
[1013 ms] escritor: entra tras esperar 959 ms
```

Los filósofos comensales



Granate: lo tiene. Punteado: lo espera.

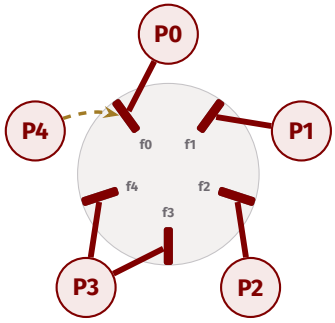
code/week-11/philo.c (naive)

```c
int left(int p)  { return p; }
int right(int p) { return (p + 1) % 5; }

void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}
void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

Ahora cada uno espera el derecho, que tiene su vecino. Nadie suelta nada: espera circular. Deadlock.

Romper el ciclo



come

philo.c, ordered (reformateado)

```
1  void get_forks(int p) {
2    if (p == 4) {  // al revés
3      sem_wait(&forks[right(p)]);
4      sem_wait(&forks[left(p)]);
5    } else {
6      sem_wait(&forks[left(p)]);
7      sem_wait(&forks[right(p)]);
8    }
9  }
```

f4 quedó libre: P3 lo toma y come. Al soltar, come P2, luego P1… No hay ciclo.

Idea general: tomar los locks en un orden global (el tenedor menor primero) impide la espera circular. Más en la semana 12.

Lo comprobamos: `code/week-11/philo.c`

5 filósofos, 1000 comidas cada uno

```
$ ./philo naive
[1059 ms] WATCHDOG: 1 s sin avance.
          Deadlock: espera circular.
  filósofo 0: tiene el tenedor 0, espera el 1
  filósofo 1: tiene el tenedor 1, espera el 2
  filósofo 2: tiene el tenedor 2, espera el 3
  filósofo 3: tiene el tenedor 3, espera el 4
  filósofo 4: tiene el tenedor 4, espera el 0
$ ./philo ordered
[  51 ms] todos comieron 1000 veces: sin deadlock
```

- Para que el deadlock aparezca siempre, la primera vez cada filósofo espera 20 ms entre los dos tenedores.
- Sin ese retraso, `naive` puede terminar o colgarse según el orden en que corran los hilos: el peor tipo de bug.
- El watchdog guarda qué tenedor tiene y cuál espera cada uno: el ciclo $0 \to 1 \to 2 \to 3 \to 4 \to 0$ se ve en la salida.

Zemaphore: un semáforo con un lock y una CV

throttle.c (reformateado)

```c
typedef struct {
  int value;
  pthread_cond_t cond;
  pthread_mutex_t lock;
} zem_t;

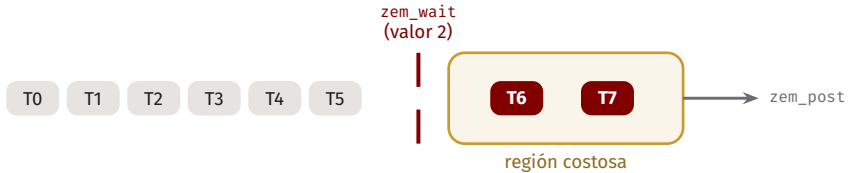
void zem_wait(zem_t *s) {
  pthread_mutex_lock(&s->lock);
  while (s->value <= 0)
    pthread_cond_wait(&s->cond,
                      &s->lock);
  s->value--;
  pthread_mutex_unlock(&s->lock);
}
```

throttle.c (reformateado)

```c
void zem_post(zem_t *s) {
  pthread_mutex_lock(&s->lock);
  s->value++;
  pthread_cond_signal(&s->cond);
  pthread_mutex_unlock(&s->lock);
}
```

- Estado (`value`), lock y CV: las reglas de la sesión A.
- A diferencia de Dijkstra, `value` nunca baja de 0, como en Linux.
- Al revés (CV con semáforos) hay trampas sutiles. Son igual de expresivos; las CVs con `while` son más fáciles de usar bien.

Throttling: limitar cuántos entran



- Un semáforo con valor inicial K deja pasar a K hilos; el $K + 1$ espera.
- Ejemplo: muchos hilos que, en una parte del programa, piden mucha memoria. Si entran todos a la vez, el sistema empieza a hacer thrashing.
- Se usa para conexiones a una base de datos, descargas simultáneas, etc.

8 hilos, límite 2

```
$ ./throttle 8 2
8 hilos, límite 2: máximo dentro a la vez
= 2 (150 ms)
```

Salida partida en dos líneas. Cada hilo entra 5 veces por 5 ms: al menos $8 \cdot 5 \cdot 5/2 = 100$ ms.

Una barrera con `broadcast`

barrier.c (reformateado)

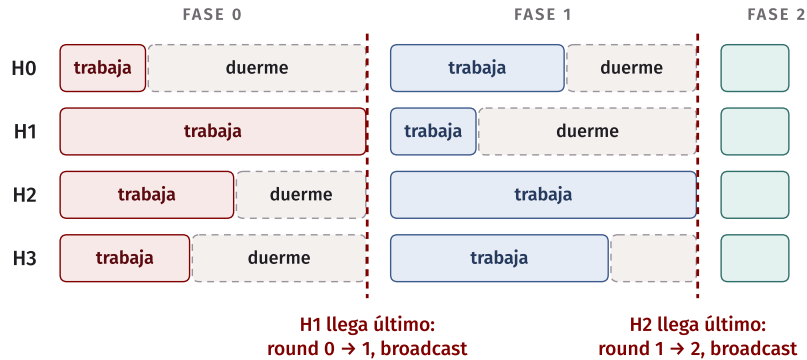
```c
void barrier_wait(barrier_t *b) {
  pthread_mutex_lock(&b->lock);
  int my_round = b->round;
  if (++b->arrived == b->n) { // el último
    b->arrived = 0;
    b->round++;
    pthread_cond_broadcast(&b->cond);
  } else {
    while (my_round == b->round)
      pthread_cond_wait(&b->cond,
                        &b->lock);
  }
  pthread_mutex_unlock(&b->lock);
}
```

- Barrera *(barrier)*: nadie pasa a la fase siguiente hasta que *todos* terminan la actual.
- El último en llegar despierta a todos: aquí `broadcast` es lo natural.
- ¿Por qué esperar a que cambie `round` y no mirar `arrived`? Porque la barrera se reutiliza: `arrived` ya volvió a 0.
- En el laboratorio la construirán con semáforos.

4 hilos, 3 fases

```
$ ./barrier 4 3
[  10 ms] fase 0: 4 de 4 hilos la terminaron
[  20 ms] fase 1: 4 de 4 hilos la terminaron
[  30 ms] fase 2: 4 de 4 hilos la terminaron
errores: 0
```

La barrera en el tiempo



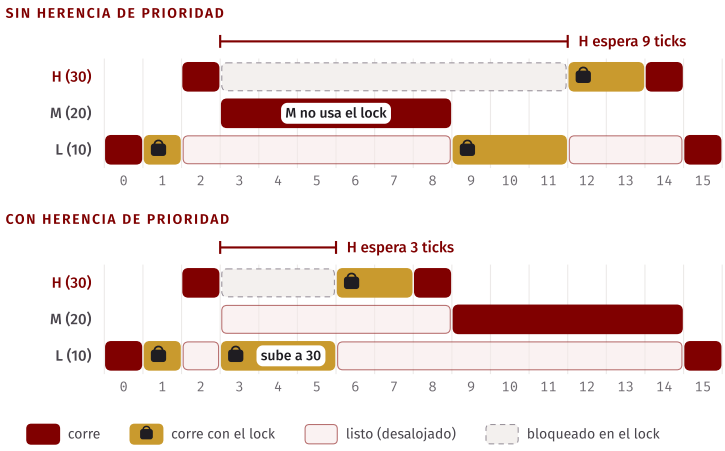
Nadie empieza la fase siguiente hasta que llega el último; los demás duermen en la CV.

SECCIÓN 04

Inversión de prioridad

04

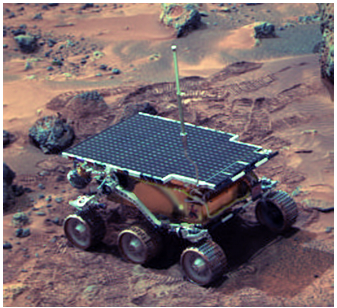
Inversión de prioridad: tres hilos y un lock



Una CPU, prioridades fijas (la regla 1 de MLFQ, semana 3). L y H comparten un lock; M no lo toca.

Sin herencia: H espera $9 = 6 + 3$ ticks: todo M y el resto de L. Con más hilos medios, sin cota.
Con herencia: al bloquearse H, L sube a 30 y M ya no lo desaloja. H espera solo el resto de la sección crítica: 3 ticks.

El caso del Mars Pathfinder (julio de 1997)



El rover Sojourner, visto desde la sonda. NASA, dominio público, Wikimedia Commons.

- A los pocos días de aterrizar, la computadora de la sonda (VxWorks) se reiniciaba sola y se perdía el trabajo del día.
- **H**: `bc_dist`, que administraba el bus de información y debía terminar en cada ciclo. **L**: la tarea meteorológica, desalojada con un semáforo tomado. **M**: tareas de prioridad media.
- Watchdog: si al empezar el ciclo siguiente `bc_dist` no había terminado, reinicio de todo el sistema.
- Arreglo: VxWorks ya tenía herencia, pero el semáforo se creó sin ella. Cambiaron una variable global y la enviaron a Marte como parche.

Fuentes: M. Jones, *What really happened on Mars?* (1997), y la respuesta de G. Reeves.

Tres soluciones

- **Herencia de prioridad** *(priority inheritance)*: el owner del lock corre con la prioridad del que más espera; al soltarlo, vuelve a la suya. La espera de H queda acotada por la sección crítica.
- **Techo de prioridad** *(priority ceiling)*: cada lock tiene un techo (la mayor prioridad de quienes lo usan) y quien lo toma sube al techo de inmediato. Hay que saber de antemano quién usa cada lock.
- **Deshabilitar las interrupciones** (semana 10): nadie desaloja al owner. Solo con una CPU y secciones críticas cortas.

```
X: con el lock, -: bloqueado

$ ./priosim inherit
        0123456789012345
H (30)    #---XX#
M (20)     ......######
L (10)  #X.XXX.........#
$ ./priosim ceiling
        0123456789012345
H (30)    ...#XX#
M (20)     ......######
L (10)  #XXXX..........#
```

Con techo, H nunca se bloquea en el lock: espera *listo* (`.`) hasta que L lo suelta.

En Linux: `PTHREAD_PRIO_INHERIT`

prioinv.c (recortado)

```c
pthread_mutexattr_t ma;
pthread_mutexattr_init(&ma);
pthread_mutexattr_setprotocol(&ma,
    PTHREAD_PRIO_INHERIT);
pthread_mutex_init(&m, &ma);
```

- Hilos `SCHED_FIFO` en la CPU 0 (`sched_setaffinity`).
- PI futex: la palabra guarda el TID del owner (el futex de la semana 10 solo dice 0, 1 o 2); así el kernel sabe a quién subir.
- `SCHED_FIFO` requiere `CAP_SYS_NICE`: `--cap-add=SYS_NICE` o `sudo`.

L usa el lock 50 ms; M, 200 ms de CPU

```
$ ./prioinv none
[  10 ms] H (30): pide el lock
[  12 ms] M (20): empieza (no usa el lock)
[ 212 ms] M (20): termina
[ 250 ms] H (30): obtiene el lock, esperó 240 ms
$ ./prioinv inherit
[  10 ms] H (30): pide el lock
[  51 ms] H (30): obtiene el lock, esperó 41 ms
[  56 ms] M (20): empieza (no usa el lock)
```

Salida recortada. Sin herencia, $240 = 200 + 40$ (todo M y el resto de L), como $9 = 6 + 3$ ticks.

SECCIÓN 05

sleep y wakeup en xv6

05



Las variables de condición de xv6: `kernel/proc.c`

kernel/proc.c (sin el panic)

```
1  void sleep_prepare(void *chan) {
2    struct proc *p = myproc();
3    acquire(&p->lock);
4    p->chan = chan;
5    release(&p->lock);
6  }
7
8  void sleep(void) {
9    struct proc *p = myproc();
10   acquire(&p->lock);
11   if (p->chan != 0) {
12     p->state = SLEEPING;
13     sched();
14   }
15   release(&p->lock);
16 }
```

kernel/proc.c (reformateado)

```
1  void wakeup(void *chan) {
2    struct proc *p;
3    for (p = proc; p < &proc[NPROC]; p++) {
4      acquire(&p->lock);
5      if (p->chan == chan) {
6        p->chan = 0;
7        if (p->state == SLEEPING)
8          p->state = RUNNABLE;
9      }
10     release(&p->lock);
11   }
12 }
```

- El canal es una dirección que acuerdan los dos lados. No hay objeto CV.
- `wakeup` despierta a todos: un `broadcast`, y Mesa.

Anotar el canal *antes* de soltar el lock

	Lector (`piperead`), orden **malo**	Escritor (`pipewrite`)
1	`acquire(&pi->lock)`; vacío	
2	`release(&pi->lock)`	
3		`acquire`; escribe un byte
4		`wakeup(&pi->nread)`: nadie anotado
5	`sleep_prepare(&pi->nread)`; `sleep()`	
6	duerme aunque hay datos	

El orden real de xv6

`sleep_prepare` con `pi->lock` tomado. Si el `wakeup` llega antes de `sleep()`, borra `p->chan` y `sleep()` no duerme.

¿Y el lock?

`pi->lock` hace atómicos "revisar y anotarse" frente a "cambiar y despertar": es el `m` de `pthread_cond_wait`.

El padre espera al hijo, en el kernel

kernel/proc.c: kwait (recortado)

```
1  acquire(&wait_lock);
2  for (;;) {
3    for (pp = proc; pp < &proc[NPROC]; pp++) {
4      if (pp->parent == p) {
5        acquire(&pp->lock);
6        if (pp->state == ZOMBIE) {
7          ...  // recogerlo y retornar
8        }
9        release(&pp->lock);
10     }
11   }
12   sleep_prepare(p);   // canal: el padre
13   release(&wait_lock);
14   sleep();
15   acquire(&wait_lock);
16 }
```

proc.c: kexit (recortado)

```
1  acquire(&wait_lock);
2  wakeup(p->parent);
3  acquire(&p->lock);
4  p->xstate = status;
5  p->state = ZOMBIE;
6  release(&wait_lock);
7  sched();
```

join.c	xv6
done	pp->state == ZOMBIE
m	wait_lock
c	canal p (el padre)
while	for (;;)

Avisa con `wait_lock` tomado: el padre revisa cuando ya es zombi.

El pipe: un bounded buffer con dos canales

kernel/pipe.c: piperead (recortado)

```
1  acquire(&pi->lock);
2  while (pi->nread == pi->nwrite
3         && pi->writeopen) {
4    sleep_prepare(&pi->nread);
5    release(&pi->lock);
6    sleep();
7    acquire(&pi->lock);
8  }
9  ...  // copiar bytes; pi->nread++
10 wakeup(&pi->nwrite);
11 release(&pi->lock);
```

pc.c	kernel/pipe.c
`count == 0`	`nread == nwrite`
`count == max`	`nwrite == nread + 512`
CV `fill`	canal `&pi->nread`
CV `empty`	canal `&pi->nwrite`

pipe.c: pipewrite (recortado)

```
1  while (i < n) {
2    if (pi->nwrite ==
3        pi->nread + PIPESIZE) {
4      wakeup(&pi->nread);
5      sleep_prepare(&pi->nwrite);
6      release(&pi->lock);
7      sleep();
8      acquire(&pi->lock);
9    } else { ... }  // copiar un byte
10 }
```

- Dos canales, como dos CVs. El mutex es `pi->lock`.
- En `pipewrite`, el `if` está dentro de `while (i < n)`: vuelve a revisar. Un `while` disfrazado.

Resumen

- Una **variable de condición** es una cola para dormir hasta que un estado cambie. `wait` suelta el lock y duerme en un solo paso.
- Se necesitan las tres piezas: **estado**, **lock** y **CV**. Sin estado, el aviso se pierde; sin lock, revisar y dormir no son atómicos.
- Semántica **Mesa**: despertar es solo una pista, así que siempre **while**. Productor/consumidor: `while` y **dos CVs**. `broadcast` cuando no se sabe a quién despertar.
- Un **semáforo** es un entero: `sem_wait` resta y quizá duerme; `sem_post` suma y quizá despierta. Valor inicial 1 para un lock, 0 para ordenar, *K* para limitar.
- Con semáforos: el mutex **adentro** de `empty`/`full`; lectores/escritores con inanición; filósofos sin deadlock si se **rompe el ciclo**.
- **Inversión de prioridad**: un hilo medio desaloja al owner del lock que espera el alto. Se evita con **herencia** (`PTHREAD_PRIO_INHERIT`) o **techo** de prioridad.
- xv6: `sleep_prepare`, `sleep` y `wakeup` sobre un **canal**; así esperan `kwait` y los pipes.

Para la próxima clase

- **Laboratorio**: las variantes `main-*.c` del productor/consumidor (cap. 30): ¿por qué falla o funciona cada una? Tarea del cap. 31: barrera, rendezvous, lectores/escritores y mutex sin inanición.
- Semana 12: OSTEP cap. 32 *Common Concurrency Problems* y 33 *Event-based Concurrency*; sesión B: cap. 35–37.

Entregas

Se publica el bono: `fork` con copy-on-write en xv6 (lab `cow` del MIT), para la semana 16. P4 `pzip` sigue en curso: entrega en la semana 13.

Preguntas para pensar

Muestre una intercalación en la que usar `if` en vez de `while` rompe el productor/consumidor. ¿A qué valor se inicializa un semáforo usado como lock? ¿Y uno para esperar a que un hilo termine? ¿Por qué los filósofos ingenuos llegan a deadlock y cómo se evita? ¿Por qué la herencia de prioridad acota la espera del hilo alto?