Tema 5 - Concurrencia avanzada

Procesos

Felipe Ortega, Enrique Soriano

GSyC, ETSIT. URJC.

Sistemas Distribuidos (SD)

23 de octubre, 2019

Fundamentos y soporte hardware

Contenidos

Repaso: Condición de carrera

- El resultado depende de cual ejecute primero.
- No sabemos qué proceso/thread ejecutará primero.
- Es el peor bug posible (porque funciona con cierta probabilidad, que puede ser muy alta).

clone()

```
int clone(int (*fn)(void *), void *child_stack,
int flags,  void *arg, ... );
```

Tiene muchas flags para compartir distintos recursos en padre e hijo:

- `CLONE_VM`: padre e hijo comparten memoria. Se comparte toda la memoria. Los mmap/munmap afectan a los dos.
- `SIGCHLD` para poder esperar por el hijo.
- La pila nunca se comparte (evidentemente). A `clone` se le pasa la pila para el hijo.
- Las pilas de los procesos NO están en distintos espacios de memoria
- Usar `clone` directamente puede ser complejo.

pthreads

- En Linux hay una implementación del estándar POSIX de threads.
- POSIX threads es una interfaz para tratar con hilos. Tiene distintas implementaciones en distintos sistemas.
- La implementación actual en Linux, NPTL, utiliza la llamada `clone` para crear los hilos → modelo 1:1.

```
man 7 pthreads
```

Repaso: variable local

```c
void *
fn(void *p)
{
        int i = 0;

        i++;
        fprintf(stderr, "FN: i is %d\n", i);
        return NULL;
}

int
main(int argc, char *argv[])
{
        int i = 0;
        pthread_t  thread;

        if(pthread_create(&thread, NULL, fn, NULL)) {
                warn("error creating thread");
                return 1;
        }
        i++;
        fprintf(stderr, "MAIN: i is %d\n", i);

        if(pthread_join(thread, NULL)  != 0){
                warn("error joining thread");
                return 1;
        }
        return 0;
}
```

Repaso: variable global

```
int i;

void *
fn(void *p)
{
        i++;
        fprintf(stderr, "FN: i is %d\n", i);
        return NULL;
}

int
main(int argc, char *argv[])
{
pthread_t  thread;

if(pthread_create(&thread, NULL, fn, NULL)) {
        warn("error creating thread");
        return 1;
}
i++;
fprintf(stderr, "MAIN: i is %d\n", i);

if(pthread_join(thread, NULL)  != 0){
        warn("error joining thread");
        return 1;
}
return 0;
}
```

Repaso: Condición de carrera

- Depende de cual ejecute primero.
- Es incluso peor.
- Pueden ejecutar a la vez (varios procesadores, paralelismo).
- $i++$ pueden ser varias instrucciones (LOAD, INC, STORE).
- caches, pipelines... impredecible.
- **MANTRA: no se comparten variables/recursos sin protección**.
- Problema general: varios hilos de ejecución, recurso compartido.

Región crítica

- Sección del código que accede al recurso compartido.
- Se ejecuta en los diferentes hilos.
- Se debe evitar que ejecute concurrentemente.

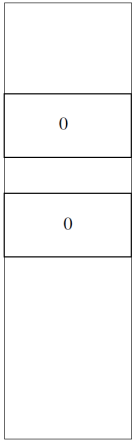
Intento protegerme con un `if`: MAL

```
int i;
int busy;

void *
fn(void *p)
{
        if(!busy){
                busy = 1;
                i++;
                busy = 0;
        }
        printf("FN: i is %d\n", i);
        return NULL;
}

int
main(int argc, char *argv[])
{
        ... (se crea el thread) ...
        if(!busy){
                busy = 1;
                i++;
                busy = 0;
        }
        printf("FN: i is %d\n", i);
        ... (se espera por el thread) ...
}
```

Intento protegerme con un `if`: MAL



PADRE

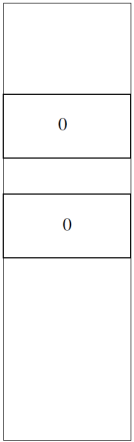
```
if(busy==0){
    busy = 1;
    i++;
    busy = 0;
}
```

Es busy 0?

HIJO

```
if(busy==0){
    busy = 1;
    i++;
    busy = 0;
}
```

Intento protegerme con un `if`: MAL



PADRE

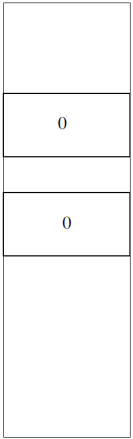
```
if(busy==0){
                    <-------- (busy es 0, entro)
    busy = 1;
    i++;
    busy = 0;
}
```

HIJO

```
if(busy==0){

    busy = 1;
    i++;
    busy = 0;
}
```

Intento protegerme con un `if`: MAL



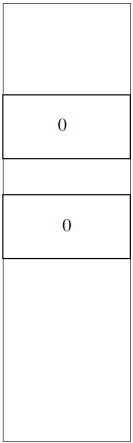
PADRE

```
if(busy==0){
    busy = 1;
    i++;
    busy = 0;
}
```

HIJO

```
if(busy==0){      <-- Es busy 0?
    busy = 1;
    i++;
    busy = 0;
}
```

Intento protegerme con un `if`: MAL



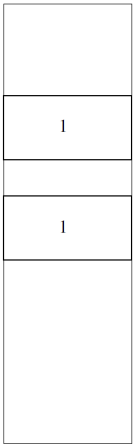
PADRE

```
if(busy==0){
    busy = 1;
    i++;
    busy = 0;
}
```

HIJO

```
if(busy==0){
                    <-------- (busy es 0, entro)
    busy = 1;
    i++;
    busy = 0;
}
```

Intento protegerme con un `if`: MAL



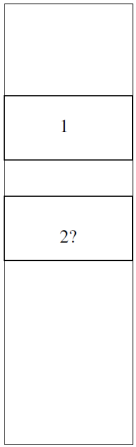
PADRE

```
if(busy==0){
    busy = 1;
    i++;
    busy = 0;
}
```

HIJO

```
if(busy==0){
    busy = 1;
    i++;        <-------- incremento i
    busy = 0;
}
```

Intento protegerme con un `if`: MAL



PADRE

```
if(busy==0){
    busy = 1;
    i++;          <-- incremento i
    busy = 0;
}
```

HIJO

```
if(busy==0){
    busy = 1;
    i++;
    busy = 0;
}
```

Intento protegerme con un `if`: MAL

- He compartido busy...
- ... condición de carrera!

Intento protegerme con un `if`: MAL

- Imaginemos que estoy en una biblioteca.
- Consulto si hay un libro.
- Mientras otro compañero se lo lleva.
- Necesito comprobar y coger el libro en una sóla acción.

¡Necesito soporte hardware!

test and set

- ¿Sería suficiente con inhibir interrupciones?
- ¿Y si tengo 2 procesadores?
- Necesitamos `if` + `asignación` en una sola instrucción (para que no me echen).
- Se llama **test and set**.

test and set

- Me protege de paralelismo y pseudoparalelismo.
- Exclusión mutua: sólo uno ejecuta el código a la vez (en un procesador o varios).
- Cerrar el bus/region de memoria.
- En este caso necesito también atomicidad (para que no me echen en mitad).

Test and set en Plan 9

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

test and set: Exclusión mutua

- Si está a *false*, lo pone a *true* y devuelve *false*.
- Si está a *true*, lo pone a *true* y devuelve *true*.
- ¡Recuerda! En C, 0 es *false* y distinto de 0 es *true*.
- Una instrucción: atomicidad

XCHG–Exchange Register/Memory with Register

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

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

- ¿Si queremos exclusión mutua en una región crítica? Para eso implementamos locks.

test and set: Exclusión mutua

Ejemplo de implementación de spin locks:

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

        /* once fast */
        if(!_tas(&lk->val))
                return;
        /* a thousand times pretty fast */
        for(i=0; i<1000; i++){
                if(!_tas(&lk->val))
                        return;
                sleep(0);
        }
```

test and set: Exclusión mutua

```
        /* now nice and slow */
        for(i=0; i<1000; i++){
                if(!_tas(&lk->val))
                        return;
                sleep(100);
        }

        /* take your time */
        while(_tas(&lk->val))
                sleep(1000);
}
```

test and set: Exclusión mutua

```
void
unlock(Lock *lk)
{
        lk->val = 0;
}
```

Locks/Cerrojos

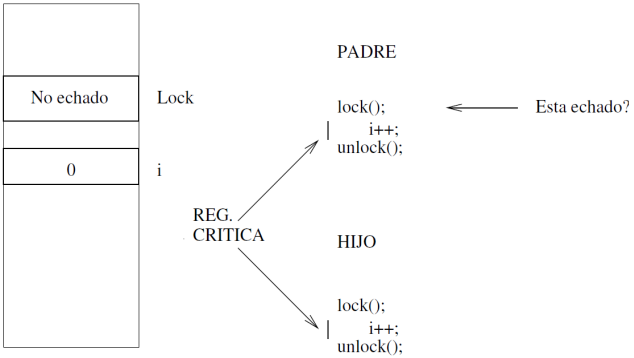
- Sabemos cómo funciona, pero ¿cómo se usa?
- Subimos el nivel de abstracción
- Coger/Soltar lock/cerrojo
- Ahora tenemos una variable especial con acceso exclusivo el lock (si usamos el interfaz) que protege el código dándonos exclusión mutua
- En `pthreads` hay locks: `man pthread_spin_lock`

De nuevo, ahora con un `lock` (BIEN)

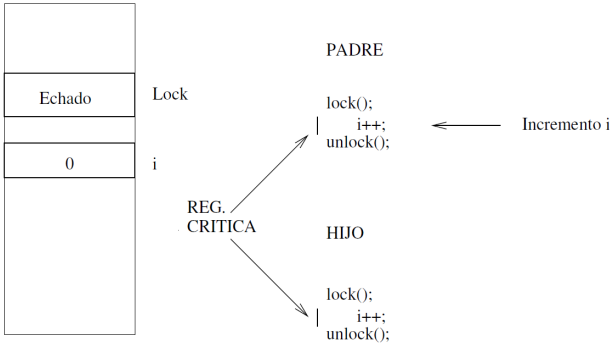
¡Ahora sí!

```
pthread_spin_lock(&l);
i++;
pthread_spin_unlock(&l);
```

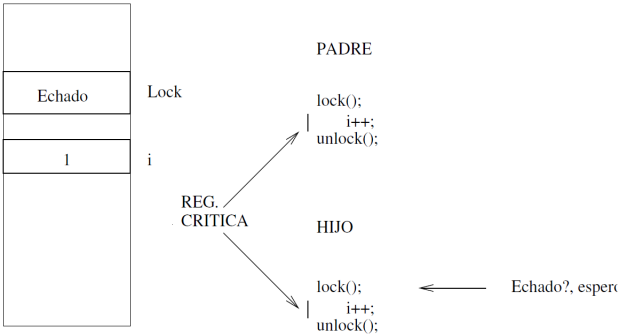
Me protejo con un `lock` (BIEN)



Me protejo con un `lock` (BIEN)

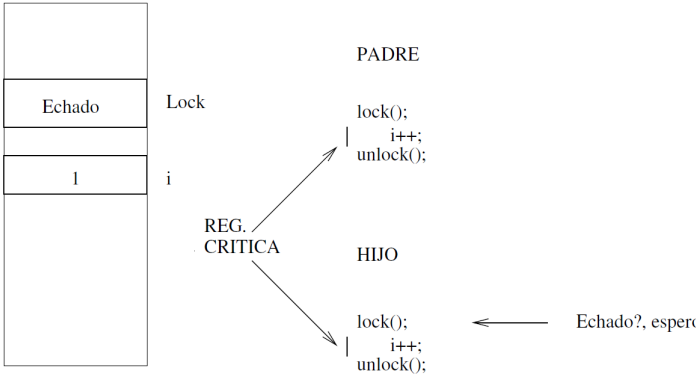


Me protejo con un `lock` (BIEN)

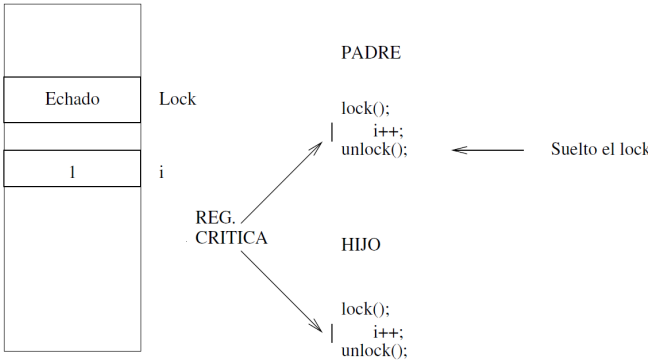


Me protejo con un `lock` (BIEN)

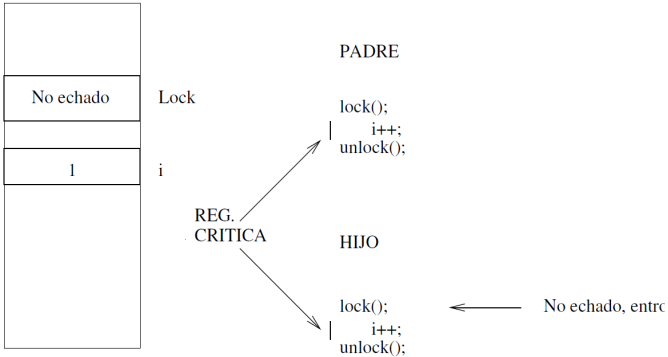
Es un spinlock, agoto el cuanto haciendo *polling*.



Me protejo con un `lock` (BIEN)



Me protejo con un `lock` (BIEN)



Locks: Espera activa

- Se llaman spin locks porque iteran.
- Hacen espera activa, desperdicio el procesador.
- Hace *polling* sobre la variable.

Locks: Exclusión mutua

- ¿Por qué el el lock es una variable global?
- ¿Y si quiero que sólo uno incremente `i`? Hay que usar otra variable booleana *incrementado*.
- ¿Y si quiero que ejecuten en un orden concreto? Espera activa.

Deadlock, interbloqueo, abrazo mortal

```
...
// en un hilo...
pthread_spin_lock(&l1);
pthread_spin_lock(&l2);
//blabla
pthread_spin_unlock(&l2);
pthread_spin_unlock(&l1);
...

// en otro hilo...
pthread_spin_lock(&l2);
pthread_spin_lock(&l1);
//blabla
pthread_spin_unlock(&l1);
pthread_spin_unlock(&l2);
```

Deadlock, interbloqueo, “abrazo mortal”

- Sucede con dos locks.
- A coge L1, espera a que B suelte L2.
- B coge L2, espera a que A suelte L1.
- A espera a B, pero B no acaba porque espera a A.
- Dependencia circular.

Deadlock, ejemplo vida real

- Dos estudiantes, Juan y Alberto.
- Práctica compartida (no en esta asignatura).
- Juan: haré mi parte de la práctica cuando Alberto acabe la suya.
- Alberto: haré mi parte de la práctica cuando Juan acabe la suya.
- ¿Cuando hacen la práctica?

Deadlock, síntomas

- Spin locks: el uso de cpu sube al máximo (espera activa).
- Varios procesos (los implicados), no progresan, siempre igual.

Evitar deadlocks

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

Deadlock, detección

- En ocasiones se pueden detectar cuando ya han sucedido.
- Se construye un WFG, wait for graph (grafo de quién espera a quién).

Buenas prácticas: cada return su `unlock`

Mal:

```
pthread_spin_lock(&l);
if (bla) {
        dosomething();
        doanotherthing();
}
else
        return;
pthread_spin_unlock(&l);
```

Buenas prácticas: cada return su `unlock`

Mejor:

```
pthread_spin_lock(&l);
if (bla) {
        dosomething();
        doanotherthing();
}
else {
        pthread_spin_unlock(&l);
        return;
}
pthread_spin_unlock(&l);
```

Buenas prácticas: cada return su `unlock`

Todavía mejor (no siempre se puede):

```
pthread_spin_lock(&l);
if (bla) {
        dosomething();
        doanotherthing();
}
pthread_spin_unlock(&l);
return;
```

Caso interesante de deadlock

- En vuestra biblioteca de threads, dos threads en un mismo proc usan un spin lock para proteger una región crítica
- ¿Donde está el deadlock?

Caso interesante de deadlock

Respuesta:

1. El thread A del proceso P coge el lock.
2. El thread A del proceso P hace `yield()`.
3. El thread B del proceso P intenta coger el lock.
4. El thread B del proceso P entra en un bucle infinito: A nunca podrá soltar el lock (2 threads, 1 proceso).

Contention/Contienda

- Cuando dos hilos de ejecución compiten por entrar a la vez en la reg. crítica, uno espera.
- Si muchos intentan entrar $\rightarrow$ mucha contienda $\rightarrow$ malo.
- Malo especialmente si hay espera activa.
- Malo en general (tenemos procesos que no hacen nada).
- Mantener las regiones críticas pequeñas.

Evitar la espera activa

- Seguimos sin tener forma eficiente de esperar.
- Hacer *polling* sobre una variable: convierto el PC en una parrilla.
- Desperdicia procesador.
- Latencia vs. carga procesador.
- No es justa (no es FIFO, sino casi aleatoria).
- Puede haber inanición (*starvation*). no en teoría, pero en la práctica sí (ejecuta al cabo de mucho).
- Evitar la contienda (*contention*) ayuda $\rightarrow$ mantener las regiones críticas pequeñas.
- Pero ¿y si podemos evitarla del todo...?

Bloquear un hilo de ejecución

- Para evitar espera activa: mecanismo para bloquear un proceso.
- Primitiva de sincronización: llamada al sistema (en el kernel) o de librería.
- Bloquear un proceso;
  - cambia el estado a bloqueado;
  - se le echa del procesador.
- Despertar un proceso;
  - cambia el estado a listo para ejecutar.

Bloquear un proceso

- Se puede hacer con un *pipe*.
- Para un proceso lo pongo a leer (ojo, no escribo: buffering).
- Para despertar a uno escribo en el *pipe*.

... tenemos muchas más primitivas de sincronización disponibles...

Operaciones atómicas en Go

En Go tenemos operaciones atómicas en `sync/atomic`:

- `Swap`: intercambia dos valores.
- `CompareAndSwap`: compara un valor con otro, y si es igual, lo reemplaza por un tercer valor y retorna True. En otro caso retorna False. Es parecido a Test-and-Set.
- `Add`: aplica un delta de forma atómica a un valor.

Elementos de sincronización

Contenidos

Primitivas del OS para bloquear un proceso

- Seguimos con el problema: hay que bloquear un proceso para evitar la espera activa.
- Cada sistema operativo tiene sus primitivas básicas para esto.
- Con esas primitivas básicas se pueden construir otras primitivas de sincronización.

Futex

Primitiva de sincronización de Linux. Es un valor entero con básicamente estas operaciones:

- `Wait(n)`: si el futex tiene dicho valor n, el hilo entra al kernel y se duerme.
- `Wake(x)`: despierta a un número x de hilos dormidos en ese futex.
- Es básicamente un *compare-and-wait*.
- "Futexes are tricky"[1]. **Son complicados de entender y usar**.

[1] Ulrich Drepper, Red Hat. 2005

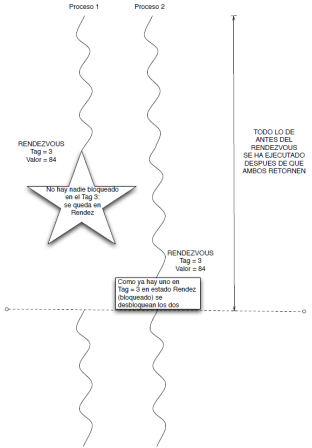
Rendezvous, cita

- Plan 9 tiene una llamada al sistema `rendezvous`:

```
void* rendezvous(void* tag, void* value)
```

- Entre dos (cita *romántica*).
- Es simétrica, el primero que llega se bloquea.
- Garantiza: que lo de antes se ha ejecutado en los dos procesos.
- Se usa un identificador, tag, (identifica una pareja).
- Intercambian dos valores.

Rendezvous, cita



Rendezvous, cita

- ¿Qué se usa en tag?
- Un valor único para esta pareja acordado.

Rendezvous, cita

- ¿Qué se usa en el valor?
- Un valor que quiero intercambiar
- Normalmente intercambio punteros (el valor del puntero, no su contenido).

Queue lock

- Interfaz igual que la de un spin lock
- Estos son justos (tienen una cola o algo análogo).
- No hacen espera activa: el hilo se bloquea si no puedes coger el cierre.
- En Go: Mutex.
- En pthreads: mutex.

Queue lock

```
var mutex sync.Mutex

mutex.Lock()
counter++
mutex.Unlock()
```

Queue RW lock

- N lectores pueden tener el lock simultáneamente.
- Si un escritor coge el lock, nadie más lo puede coger (ni lectores ni escritores).

Queue RW lock

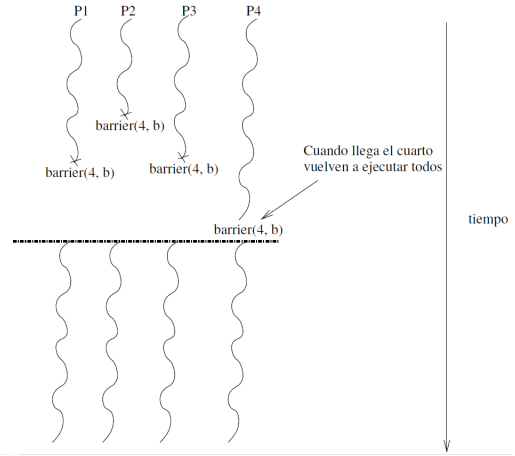
En Go exite RWMutex

- Lock: coge el mutex como escritor.
- Unlock: suelta el mutex como escritor. Si no lo tenemos cogido como escritor hay un error de ejecución.
- RLock: coge el mutex como lector.
- RUnlock: uelta el mutex como lector. Si no lo tenemos cogido como lector hay un error de ejecución.

Barrera

- Otra primitiva de sincronización.
- Como rendezvous pero para N procesos.
- Se van bloqueando hasta que hay N, cuando hay N se desbloquean.

Barrera de 4, ejemplo



WaitGroup

Primitiva de Go. Parecido a una barrera, pero no igual:

- Las Goroutines se pueden bloquear llamando a `Wait`.
- Tiene un contador asociado, cuando el contado llega a cero, todas las Goroutines bloqueadas se desbloquean.
- Se modifica el contador llamando a `Add`. Si el contador pasa a negativo, hay un error de ejecución. Se suele usar para inicializar.
- `Done` decrementa el contador en uno.
- No es una barrera: cualquier Goroutine puede cambiar el contador, no únicamente las que se duermen como en la barrera. Permite crear barreras.

Semáforos (semaphores, seinpalen, 1965 Dijkstra)

- Otra primitiva de sincronización
- Más general
- Como un cuenco de fichas con N fichas (puede empezar a 0)
- Coger ficha (si no hay fichas, espero), wait, down, acquire, P (*"probeer"*)
- Soltar ficha (si alguien esperando, le despierto), signal, up, release, V (*"verhoren"*, *"verlaag"*)
- No hay un número máximo de fichas en el cuenco

Semáforos

Los semáforos pueden ofrecer dos semánticas:

- Débil: no se garantiza que la *ficha* (wakeup) generada por un Up sea para un hilo que ya está bloqueado en el semáforo con un Down: se le puede colar otro hilo que haga Down entre medias. Estos tipos de semáforos tienen hambruna. Malo.
- Fuerte: si se garantiza eso. Puede ser FIFO (block-queue) o no (block-set), pero se garantiza.

Semáforos

Los semáforos pueden ofrecerse a nivel de biblioteca o a nivel del sistema operativo. Los del sistema operativo pueden ser *optimistas*:

- Parte en área de usuario y parte en área de kernel.
- Optimizados para el caso común: si hay ficha, no hace falta entrar al kernel.
- Más caros en el caso raro.
- Más complicados de implementar.

Semáforos: usos

Mutex:

- Para que sólo entre uno en la región crítica
- Se inicializa a 1
- Similar a un lock
- `down()` $\equiv$ `lock()`
- `up()` $\equiv$ `unlock()`

Ejemplo de uso de mutex

```
// paquete sems propio, no standard en Go
mutex := sems.NewSem(1)

...

// desde distintas Gs podemos hacer...
mutex.Down()
counter++
mutex.Up()
```

Semáforos: usos

- Para que sólo entren N en la región crítica
- Se inicializa a N
- down() antes de entrar
- up() al salir

Semáforos: usos

- Para esperar
- Se inicializa a 0
- down() para esperar
- up() para levantarlo

¿Cómo implemento semáforos?

Por ejemplo, con pipes:

- `up()`: escribir en el pipe.
- `down()`: leer del pipe.

El problema de los lectores escritores con semáforos

- Región crítica con dos tipos de hilos que acceden.
- Un tipo de hilos **lectores** pueden acceder varios a la vez.
- Otro **escritores** sólo puede acceder uno.
- Hay **exclusión mutua** entre ambos tipos (o lectores o escritores).
- Otra forma de verla es dos tipos de formas de coger un cierre.

Solución 0 (mala)

- Tener un mutex para la región crítica...
- ...sólo puede entrar un lector (y quiero que puedan entrar varios).

Solución 1 (injusta)

- Dos semáforos, un mutex para la región crítica.
- Un mutex para proteger el contador de lectores.
- Un contador del número de lectores.
- El primero echa el cierre y el último lo suelta.
- Ojo! con el orden y quién se queda bloqueado.

Solución 1 (injusta): escritor

```
//globales
mutexesc := sems.NewSem(1)
mutexnl := sems.NewSem(1)
nl int

//escritor
mutexesc.Down()

//region critica
writeData()

mutexesc.Up()
```

Solución 1 injusta: lector

```
//lector
mutexnl.Down()
nl++
if nl == 1 {
        mutexesc.Down()
}
mutexnl.Up()

//region critica
readData()

mutexnl.Down()
nl--
if nl == 0 {
        mutexesc.Up()
}
mutexnl.Up()
```

Solución 1 injusta

- Uso peculiar de mutex entrelazados, ojo!!
- Es injusta, los lectores se apropian del cierre y se lo pasan sólo entre ellos.
- Si hay un escritor esperando, no debería dejar entrar a mas lectores

Idea general, torniquetes (turnstile)

- Para evitar injusticias como las anteriores
- Uso peculiar de semáforos

Torniquetes (turnstile)

Patrón:

```
torn.Down()
torn.Up()
```

Torniquetes (turnstile)

- Si espera para el mutex, tiene parado el torniquete
- El torniquete deja pasar en orden justo (es un semáforo)

Solución 2 justa: escritor con torniquete

```
//globales
mutexesc := sems.NewSem(1)
mutexnl := sems.NewSem(1)
ltorn := sems.NewSem(1)
nl int

//local
torn.Down()
mutexesc.Down()
torn.Up()

//region critica
writeData()

mutexesc.Up()
```

Solución 2 justa: lector con torniquete

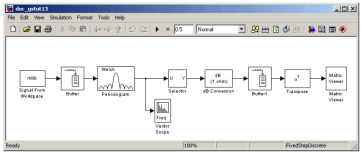
```
torn.Down()
torn.Up()
mutexnl.Down()
nl++
if nl == 1 {
        mutexesc.Down()
}
mutexnl.Up()

//region critica
readData()

mutexnl.Down()
nl--
if nl == 0 {
        mutexesc.Up()
}
mutexnl.Up()
```

Productor/consumidor

- Tenemos un hilo que `produce` tickets
- Tenemos otro hilo que las `consume`
- Problema muy muy común:
  - Hilo que lee del micro produce audio , programa lo consume (teléfono).
  - Hilo que lee de la red, hilo que lo consume (programa que usa la red).
  - Hilo que que produce datos, hilo que los manda a otra máquina (programa que usa la red).
  - Hilo que que produce datos, hilo que los procesa, se lo manda a otro hilo. . . (procesado de señal).



Productor/consumidor

- El productor deja las cosas en un buffer (con N huecos).
- El consumidor las recoge.
- El buffer tiene que ser circular (mod N).
- El buffer es compartido, mutex.
- ¿Y si no hay nada, espera activa?
- Puedo usar semáforos.

Productor

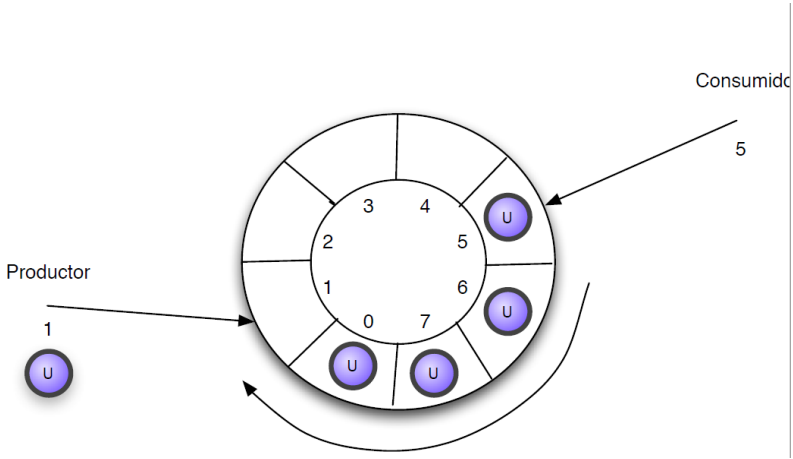
```
//global
ticketsem := sems.NewSem(0)
holesem := sems.NewSem(N)
buf  = make([]*string, N)

//local
i := 0
for  {
        ticket := produce()
        holesem.Down()
        buf[i] = ticket;
        i = (i + 1) % N
        ticketsem.Up()
}
```

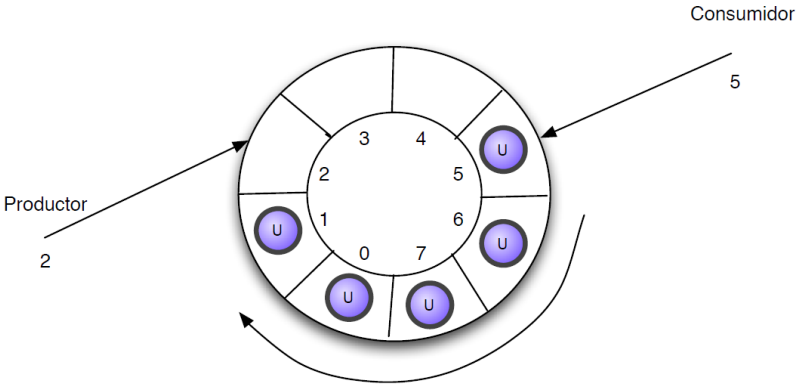
Consumidor

```
//local
j := 0
for {
        ticketsem.Down()
        ticket := buf[j];
        j = (j + 1) % N
        holesem.Up()
        consume(ticket)
}
```

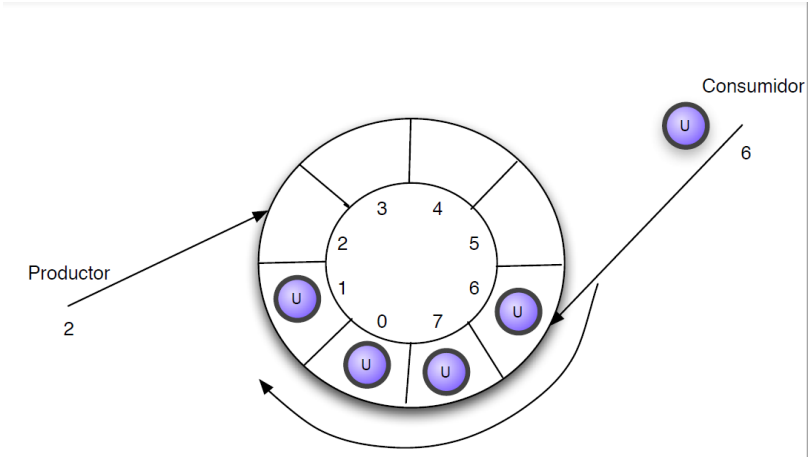
Productor/consumidor (N=8)



Productor/consumidor



Productor/consumidor



Productor/consumidor

- Solución mala, da miedo, el buffer no tiene mutex (¿lo necesita?), por si acaso, es una variable compartida, lo pongo.
- ¿Qué pasa si hay más de un productor y un consumidor? Necesito que compartan i y j.
- Solución: uso un mutex general para acceder al buffer y modificar i y j.

Productor/consumidor

```
//global
mutex := sems.NewSem(1)
ticketsem := sems.NewSem(0)
holesem := sems.NewSem(N)
i, j int
```

Productor

```
//local
for {
        ticket := produce()
        holesem.Down()
        mutex.Down()
        buffer[i] = ticket
        i = (i + 1) % N
        mutex.Up()
        ticketsem.Up()
}
```

Consumidor

```
//local
for {
        ticketsem.Down()
        mutex.Down()
        ticket := buffer[j]
        j = (j + 1) % N
        mutex.Up()
        holesem.Up()
        consume(ticket)
}
```

Productor/consumidor

Como es circular y sólo pasan los productores/consumidores que pueden pasar...

- Podríamos tener dos mutex distintos: uno para proteger i y otro para proteger j, porque el buffer en sí no necesita el mutex.

... pero...

- ¿Y si no queremos sacarlos en orden FIFO?
- ¿Y si el recurso compartido no es un buffer y es otro tipo de recurso?
- Mantra: un recurso compartido, ¡uso un mutex!

Problema de los filósofos cenando, Dijkstra 1965

- Tengo N filósofos (hilos de ejecución) en una mesa.
- Cada filósofo necesita dos tenedores para comer.
- Un filósofo duerme/come.
- Cada filósofo es un proceso independiente.
- Un filósofo se puede quedar dormido hasta que suceda una condición.

Problema de los filósofos: deadlock

- Si un filósofo coge primero el tenedor izquierdo y luego el derecho.
- Puede pasar que todos los filósofos cojan su tenedor izquierdo...
- ... y se queden durmiendo esperando cada uno al siguiente alrededor de la mesa.

Problema de los filósofos: livelock

- Si un filósofo coge primero el tenedor izquierdo, y si el derecho está ocupado, lo suelta y duerme otro rato. En otro caso, coge el derecho y come.
- Puede pasar que todos los filósofos se coordinen e intenten coger al mismo tiempo el de su derecha, que está ocupado, esperen otro rato y se vuelvan a sincronizar.
- Que cada filósofo espere un número aleatorio de tiempo para reintentar soluciona el problema, aunque no es lo más elegante y no es deseable en muchos casos.

Problema de los filósofos: infinite overtaking

- Si hay un filósofo muy rápido y su vecino es muy lento, puede que al rápido le de tiempo a soltar el tenedor y volverlo a coger antes de que el lento lo pueda coger: hambruna.

Alternativas (I)

- Cojo un mutex global para todos los palillos mientras como:
  - Correcto, pero sólo come un filósofo a la vez $\rightarrow$ ineficiente, no pueden comer dos filósofos no vecinos en paralelo.

Alternativas (II)

- Solución[2]:
  - Array con el estado de cada filósofo (Durmiendo, Comiendo, Hambriento).
  - Un filósofo puede pasar de estado Hambriento a Comiendo sólo si ninguno de sus vecinos no tiene estado Comiendo.
  - Un mutex para acceder al array de estados.
  - Un semáforo por cada filósofo, para bloquearlo hasta que pueda pasar a estado Comiendo.
  - Hay que tener cuidado: puede haber hambruna si los dos vecinos de un Hambriento se alternan.

[2] Modern Operating Systems, A. Tanenbaum, Cap. 2

Alternativas (III)

- Solución: Orden total. Un camarero establece una cola de comandas que imponen un orden total. Un filosofo no puede comer aunque pueda si un vecino suyo lleva más tiempo esperando para comer.

Monitores

- C. A. R. Hoare y Per Brinch Hansen.
- Asociado al lenguaje.
- Objeto, ADT o módulo especial que actúa como mecanismo de sincronización.
- Tengo el estado compartido dentro del monitor.
- Una serie de operaciones para acceder a él.
- Hay el equivalente a un cierre global escondido asociado al monitor.

Monitores y variables condición

- Estoy "fuera" o "dentro" del monitor, automaticamente se echa el cierre al entrar, se suelta al salir.
- Se entra al monitor al llamar a uno de los métodos o funciones del monitor.
- Si estoy dentro puedo llamar a otros métodos del monitor.

Monitores y variables condición

Es una condición para esperar, asociada a un monitor o a un lock (si el lenguaje no ofrece monitores). Sus operaciones son:

- `Wait (o sleep)` : el hilo se duerme y suelta el cierre de forma atómica.
- `Signal (o notify)` un hilo dormido se despierta y coge el cierre de forma atómica.
- `Broadcast (o notifyAll)` todos los hilos dormidos se despiertan y van cogiendo el cierre cuando pueden.

Monitores y variables condición

- Si no hay nadie esperando a la condición, se pierde el Signal.
- La variable no lleva asociada la condición, sólo es una cola y un *timbre*.
- El código tiene que comprobar la condición o pasar una función que lo haga.
- Hay varios tipos de monitores: en unos `Signal()` es lo último a lo que puedo llamar (o `condsignalreturn(cv)`), en unos se puede llamar a Signal desde fuera del monitor, en otros no se puede, algunos no nos garantizan orden al despertar, etc... **Moraleja: en cada lenguaje/sistema tenemos que leer su manual**.

Monitores y variables condición

- Pocos lenguajes tienen soporte de monitores.
- Se puede (y se debe) pensar en monitores al implementar un módulo, ADT u objeto.
- En Go hay variables condición asociadas a un `Locker`: `sync.Cond`.

Variables condición en Go

Operaciones de `sync.Cond`:

- `NewCond()`: crea una variable condición. No se debe copiar después de su primer uso.
- `Wait()`: la goroutine se queda dormida en la variable condición. Hay que tener cogido el Locker asociado a la variable condición, que se soltará al dormir. **Siempre se debe comprobar la condición al despertar**.
- `Signal()`: se despertará una de las Goroutines que están dormidas en la variable condicíón. Se puede llamar sin tener el Locker.
- `Broadcast()`: se despertarán todas las Goroutines que están dormidas en la variable condicíón. Se puede llamar sin tener el Locker.

Paso de mensajes

- CSP 1978 C. A. R. Hoare.
- Idea: convertir el productor/consumidor en una primitiva.
- Bloqueante (para permitir sincronización).
- Usar un proceso para que ejecute las regiones críticas (transferir la propiedad junto con la comunicación).
- Similar a pipes (salvo que aquí puedo mandar punteros), salvo por eso la idea es parecida.
- Go tiene canales!

Canal

- Tiene que verlo todos los implicados.
- Un canal es como un buzón.
- Se puede mandar o recibir a través de él
- Puede tener espacio (buffering) dentro o no.
- Si no hay espacio (no lo hay buffering o está lleno), el envío se bloquea hasta que alguien reciba.
- Si no hay nada en el buffer y no hay nadie bloqueado mandando, la recepción se bloquea hasta que alguien mande.

Canal

- Los canales se crean con `make`:

```
ch := make{chan int} // sin buffer
bch := make{chan int, 10} // con buffer de 10 ints
```

Canal

- Para mandar y recibir se usa el operador <-:

```
ch <- 123          // envio el int 123 por el canal ch
...
x := <- ch     // recibo un int de ch, lo guardo en x
```

Canal

- Un canal se puede cerrar para indicar al receptor de que no habrá más envíos.
- Sólo el que envía puede cerrar el canal.
- Enviar a un canal cerrado provoca un error en ejecución.
- No es necesario cerrar los canales si no necesitamos notificar que no se van a mandar más mensajes. El recolector de basura se encarga de liberarlos.

Canal

```
//emisor
..
close(ch)
...

//receptor
...
x, ok := <- ch
if !ok {
        // canal cerrado
        ...
}
```

Canal

Si queremos recibir hasta que se cierre el canal, podemos usar `range`:

```
for x := range ch {
        ...
}
```

Canal

- Con `select` podemos bloquearnos en múltiples operaciones de comunicación.
- Se bloquea hasta que una de las operaciones puede progresar, descartando el resto de operaciones.
- Es una operación de selección no determinista.
- Si más de una operación puede progresar al hacer el `select`, selecciona una de ellas de forma aleatoria.

Ejemplo: Select

```
select {
case x := <- ch1:          // recibir de ch1
...
case y := <- ch2:          // recibir de ch2
...
case ch3 <- counter:          // enviar a ch3
...
}
```

Ejemplo: Select

```
select {
case x := <- ch1:    // recibir de ch1
...
case y := <- ch2:    // recibir de ch2
...
case ch3 <- counter: // enviar a ch3
...
default:               // no puede hacer nada sin bloquearse
...
}
```

Memoria Transaccional

- Es otra aproximación distinta. Se puede implementar en hardware o software (STM, Software Transactional Memory).
- Las operaciones sobre los datos compartidos se agrupan en una transacción.
- Los hilos entran en la región crítica sin intentar coger ningún cierre (o similar).
- Si se detecta colisión mientras que se ejecuta la transacción, se reinicia la transacción.
- Ojo: con mucha contienda se puede entrar en **livelock**.

Memoria Transaccional

Ejemplo:

```
__atomic{
        references--;
        if(references == 0){
                busy = 0;
        }
}

__atomic{
        if(references == 0){
                busy = 1;
        }
        references++;
}
```

Memoria Transaccional

Se puede abortar la transacción de forma explícita para que se reinicie:

```
__atomic{
        ...
        retry;
        ...
}
```

Memoria Transaccional

Se pueden especificar una condición de guarda para que la transacción quede bloqueada hasta que se cumpla:

```
__atomic(queuesize > 0){
        ...
}
```

Bibliografía I

- *A. S. Tanenbaum.* Operating Systems, design and implementaiton. Pearson Prentice Hill.
- *A. S. Tanenbaum.* Distributed Systems. Pearson Prentice Hill.
- *A. S. Tanenbaum.* Modern Operating Systems. Pearson Prentice Hill.

[van Steen & Tanenbaum, 2017] van Steen, M., Tanenbaum, A. S.
Distributed Systems.
Third Edition, version 01. 2017.

[Colouris et al., 2011] Colouris, G., Dollimore, J., Kindberg, T., Blair, G.
Distributed Systems. Concepts and Design.
Pearson, May, 2011.