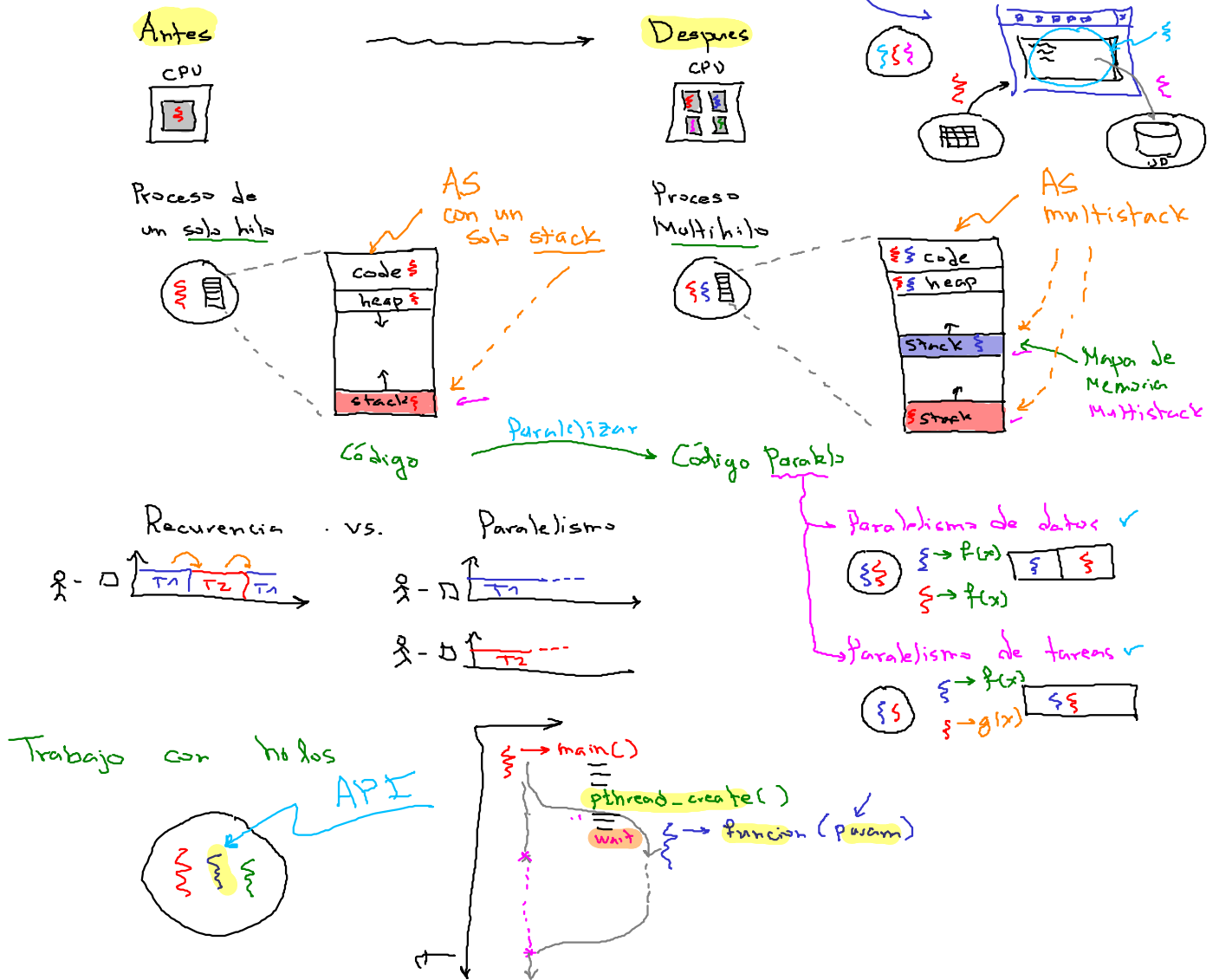


23/10/2025 - Sistemas Operativos (Ude@)

1. Repaso clase anterior: Aplicaciones concurrentes { -web server
- word
- reproductor de musica



¿Como crear hilos?

```
#include <pthread.h>

int pthread_create(pthread_t* thread,
    const pthread_attr_t* attr,
    void* (*start_routine)(void*),
    void* arg);
```

Donde:

- **thread**: permite manejar del hilo (*handler*).
- **attr**: permite crear atributos al hilo.
 - Tamaño del stack, prioridad en el planificador, ...
- **start_routine**: función que inicia ejecutando este hilo.
- **arg**: argumento pasado a la función (**start_routine**)
 - Un **apuntador** a **void** permite pasar argumentos de **cualquier tipo**.

- Si **start_routine** requiere argumentos de diferente tipo, la declaración sería como se muestra a continuación:

- Un **argumento** tipo entero (**int**)

```
int pthread_create(..., // first two args are the same
    void* (*start_routine)(int),
    int arg);
```

- Un **retorno** tipo entero (**int**)

```
int pthread_create(..., // first two args are the same
    int (*start_routine)(void*),
    void* arg);
```

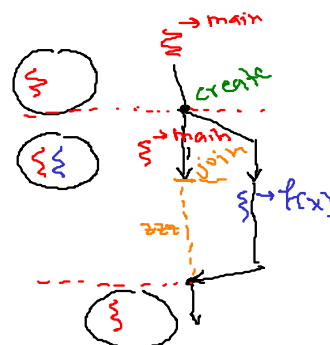
Para esperar hacer que un hilo espere la terminación de otro para acabar se emplea **pthread_join**:

sincronización

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Donde:

- **thread**: Especifica el hilo por el cual se va a esperar.
- **value_ptr**: un apuntador al valor de retorno.
 - Como **pthread_join** cambia (internamente) el valor de retorno, se debe **pasar un apuntador** a ese valor.



Problemas:

1. Comportamiento no determinista (Impredecible)

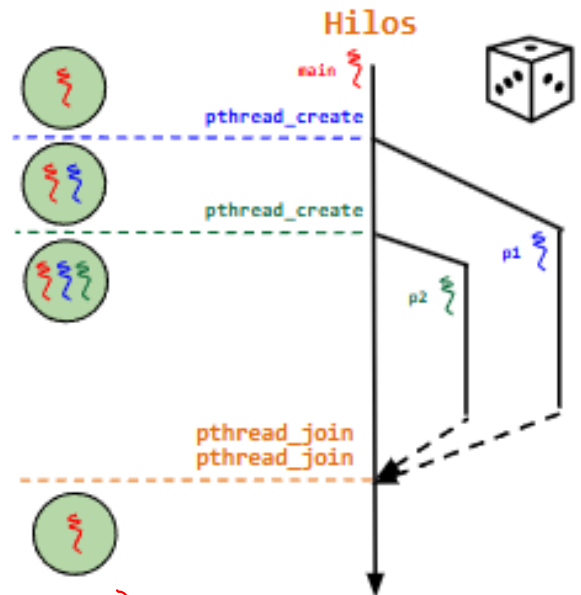
Solución: Implementar mecanismos de sincronización

t0.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  #include "common.h"
6  #include "common_threads.h"
7
8  void *mythread(void *arg) {
9      printf("%s\n", (char *) arg);
10     return NULL;
11 }
12
13 int main(int argc, char *argv[]) {
14     if (argc != 1) {
15         fprintf(stderr, "usage: main\n");
16         exit(1);
17     }
18
19     pthread_t p1, p2;
20     printf("main: begin\n");
21     Pthread_create(&p1, NULL, mythread, "A");
22     Pthread_create(&p2, NULL, mythread, "B");
23     // join waits for the threads to finish
24     Pthread_join(p1, NULL);
25     Pthread_join(p2, NULL);
26     printf("main: end\n");
27     return 0;
28 }

```



scheduler (político) SO

? ¿cuál hilo usa la CPU?

```

$ ./t0
main: begin
A
B
main: end

```

```

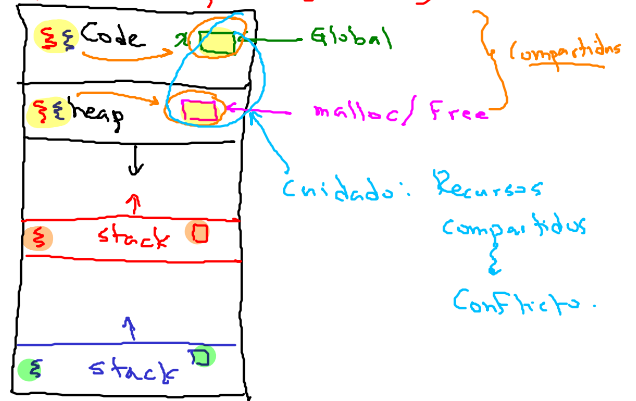
$ ./t0
main: begin
B
A
main: end

```

2. Datos compartidos

Por que sucede?

Address Space (Memory map)



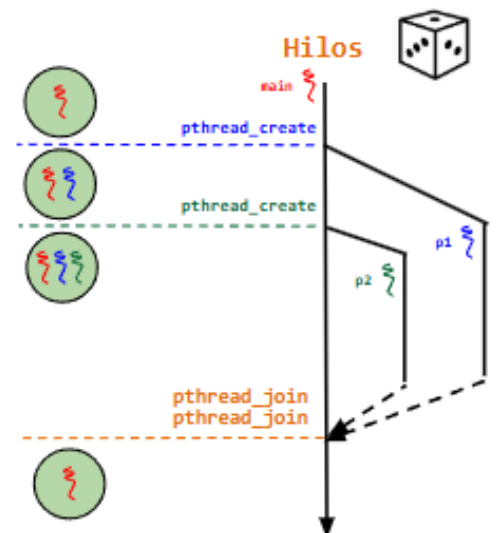
```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 #include "common.h"
6 #include "common_threads.h"
7
8 int max;
9 volatile int counter = 0; // shared global variable
10
11 void *mythread(void *arg) {
12     char *letter = arg;
13     int i; // stack (private per thread)
14     printf("%s: begin [addr of i: %p]\n", letter, &i);
15     for (i = 0; i < max; i++) {
16         counter = counter + 1; // shared: only one
17     }
18     printf("%s: done\n", letter);
19     return NULL;
20 }
21
22 int main(int argc, char *argv[]) {
23     if (argc != 2) {
24         fprintf(stderr, "usage: main-first <loopcount>\n");
25         exit(1);
26     }
27     max = atoi(argv[1]);
28
29     pthread_t p1, p2;
30     printf("main: begin [counter = %d] [%x]\n", counter,
31           (unsigned int) &counter);
32     pthread_create(&p1, NULL, mythread, "A");
33     pthread_create(&p2, NULL, mythread, "B");
34     // join waits for the threads to finish
35     pthread_join(p1, NULL);
36     pthread_join(p2, NULL);
37     printf("main: done\n [counter: %d]\n [should: %d]\n",
38           counter, max*2);
39     return 0;
40 }

```

counter

Hilos



counter = counter + 1;

```

100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c

```

```

$ ./t1
usage: main-first <loopcount>

```

```

$ ./t1 2000
main: begin [counter = 0] [7F7FE1FFD02C]
A: begin [addr of i: 0x7f7fe1d7fedc]
A: done
B: begin [addr of i: 0x7f7fe156fedc]
B: done
main: done
[counter: 4000] ✓
[should: 4000] ✓

```

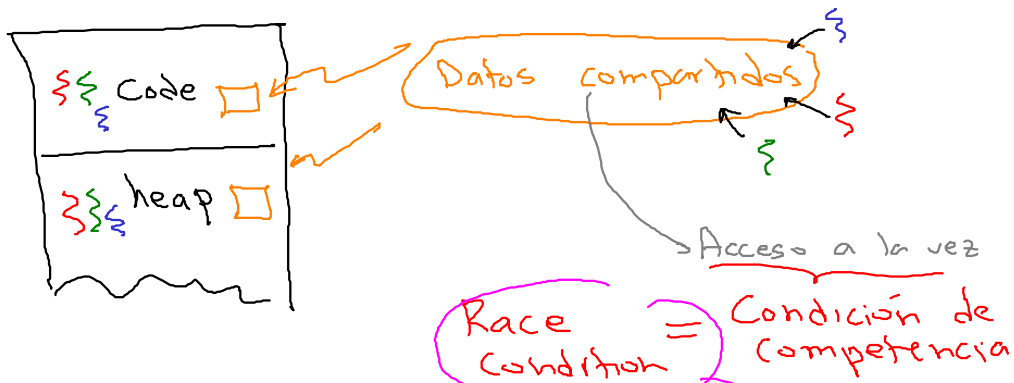
```

$ ./t1 100000000
main: begin [counter = 0] [7F996F89B02C]
A: begin [addr of i: 0x7f996f61fedc]
B: begin [addr of i: 0x7f996ee0fedc]
B: done
A: done
main: done
[counter: 99919881] ✗ (Race condition)
[should: 200000000] ✓

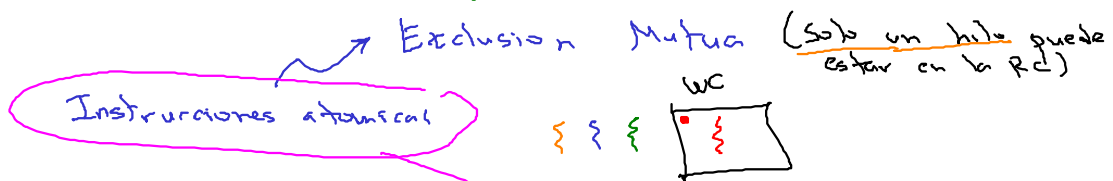
```

Wrong fitting room





Como evito la Race condition



No hay atomicidad

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Instrucción atómica

```
memory-add 0x8049a1c, $0x1
```

3. Locks

① Region critica (Recurso compartido)



```
void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // sección critica
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```
counter = counter + 1;
```

```
100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

② Problema: Race condition (Access simultáneo al recurso critico)



Improbables

③

¿Cuál es la solución?

EXCLUSIÓN MUTUA (Solo un hilo puede estar en Rcrit)

Se implementa con lock

28/10/2025 - Sistemas Operativos (Ude@)

Locks

Un lock es un objeto (en memoria) que proporciona exclusión mutua.

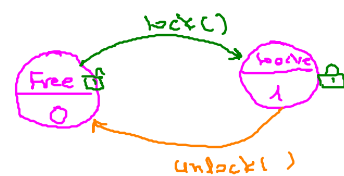
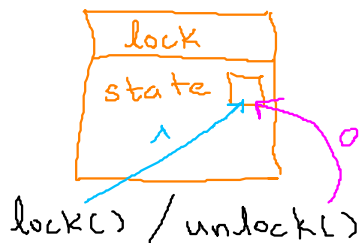
Estados



acquired (locked, held):
Indica que un hilo ha **adquirido** el lock y que este, se está ejecutando en la sección crítica



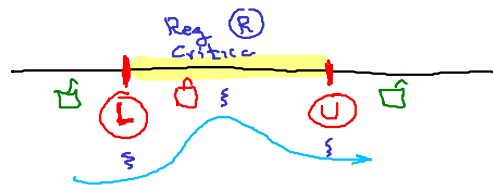
available (unlocked, free): Indica que ningún hilo ha obtenido el lock.



- Estados:
 - locked (1) ✓
 - Free (0) ✓

- Funciones:
 lock ()
 release ()

④ Como usamos los locks (condados) → Protocolo de Usa.



L Lock ()
 R Region critica
 U Unlock ()

```
acquire(lock);
Región critica
release(lock);
```

Proporciona **exclusión mutua** a una **región crítica**

Interfaz

```
int pthread_mutex_lock(pthread_mutex_t *mutex); → Lock = L
int pthread_mutex_unlock(pthread_mutex_t *mutex); → Unlock = U
```

Uso (sin inicialización, ni chequeo de errores)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

- Si **no hay hilos bloqueando al lock** → **acceda a la región crítica**
- Si **hay un hilo bloqueando el lock** → el **hilo no retorna del llamado** hasta que pueda bloquear (adquirir) el lock

Todos los **locks** debe ser **adecuadamente inicializados**:

Una manera: Usando PTHREAD_MUTEX_INITIALIZER

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Forma dinámica: usando la función pthread_mutex_init()

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Chequeo de errores cuando se llame lock y unlock

Ejemplo usando una función wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

```
pthread_mutex_t lock; // lock variable (Mutex)
```

```
void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        pthread_mutex_lock(&lock); // ----- Lock
        counter = counter + 1;      // critical section: only one [RC]
        pthread_mutex_unlock(&lock); // ----- Unlock
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);
}
```

```
pthread_t p1, p2;
```

```
int rc = pthread_mutex_init(&lock, NULL); // Initialize the lock
```

```
if (rc != 0) {
    fprintf(stderr, "main: mutex init failed\n");
    exit(1);
}
```

```
printf("main: begin [counter = %d] [%lx]\n", counter, (long unsigned int) &counter);
```

```
pthread_create(&p1, NULL, mythread, "A");
```

```
pthread_create(&p2, NULL, mythread, "B");
```

```
// join waits for the threads to finish
```

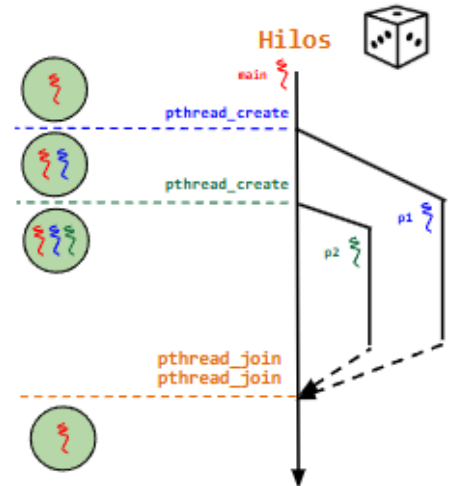
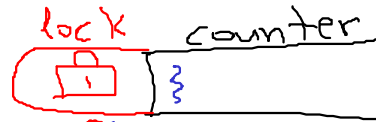
```
pthread_join(p1, NULL);
```

```
pthread_join(p2, NULL);
```

```
printf("main: done\n [counter: %d]\n [should: %d]\n",
       counter, max*2);
```

```
return 0;
```

```
}
```



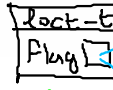
o/counter 1'000.000
counter
2'000.000

4. Como implementar locks?

- Metricas
1. Exclusion mutua ✓
 2. Equidad ✓
 3. Overhead ✓

Implementación:

Determinación de cómo implementar las funciones lock() y unlock().



0: Libre
1: Ocupado

```
void lock() {
    // Instrucciones lock
    // ...
}

void unlock() {
    // Instrucciones lock
    // ...
}
```

Que debe ir aca

Implementación en code

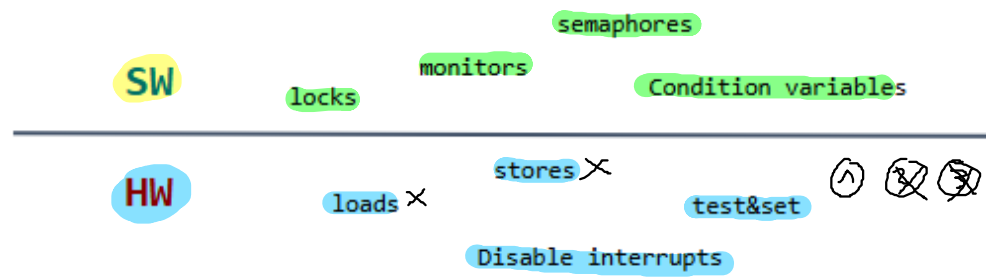
- Evaluación
1. ✓
 2. ✓
 3. ✓

Implementación de locks

¿Cómo construir un lock?

- ¿Cómo construir un lock que sea eficiente? ✓
- ¿Qué hardware es necesario (Primitivas del ISA, su uso, etc)? ✓
- ¿Cuál es el soporte del SO (Qué necesitamos usar de este)? ✓

Objetivo: Obtener exclusión mutua con el menor costo posible.



Implementación:

Determinación de cómo implementar las funciones lock() y unlock().

```
void lock() {
    // Instrucciones lock
    // ...
}

void unlock() {
    // Instrucciones lock
    // ...
}
```

La siguiente tabla muestra algunas formas de implementar locks:

Algoritmos de solo software	Instrucciones atómicas
Algoritmo de Dekker (1962)	✓ Test & Set
Algoritmo de Petterson (1981)	✓ Compare & Swap
Algoritmo de Lamport para más de dos procesos (1974)	Load-Linked (LL) & Store-Conditional
	Fetch & add

i. Implementación mediante interrupciones

Implementación mediante el control de interrupciones:

- Deshabilitar interrupciones al entrar a las regiones críticas.
 - Cuando se deshabilitan las interrupciones se bloquean eventos externos que podrían generar un cambio de contexto.
 - El código dentro de la sección crítica no será interrumpido.
 - No hay un estado asociado al lock.

Fácil →

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

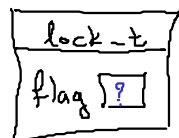
Métricas

① ✓ ② ✗ ③ ✗

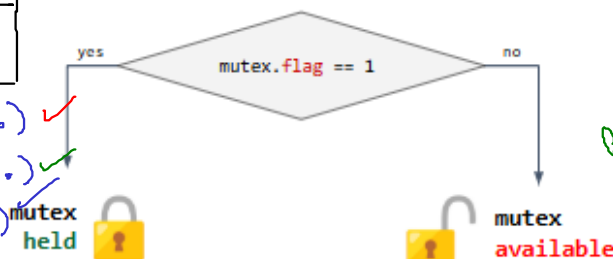
ii. Implementación mediante load & store

Implementación mediante el uso de instrucciones Load/Store

En esta solución por software, se usa una **flag** para determinar el estado.



init(...)
lock(...)
unlock(...)



```
1 typedef struct lock_t {  
2     int flag;  
3 } lock_t;  
4  
5 void init(lock_t *mutex) {  
6     // 0 -> lock is available, 1 -> held  
7     mutex->flag = 0;  
8 }  
9  
10 void lock(lock_t *mutex) {  
11     while (mutex->flag == 1) // TEST the flag  
12         ; // spin-wait (do nothing)  
13     mutex->flag = 1; // now SET it!  
14 }  
15  
16 void unlock(lock_t *mutex) {  
17     mutex->flag = 0;  
18 }
```

status → Flag = { 0, 1 }

Métricas

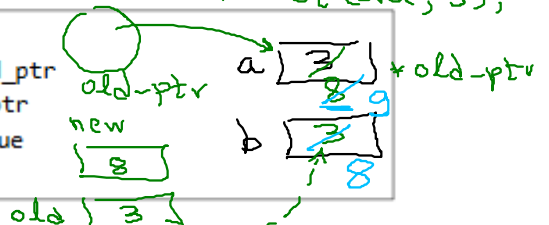
① ✗ ② ③

iii. Implementación mediante la instrucción atómica test & set

- **Instrucciones atómicas:** Garantizan que las operaciones de lectura, modificación y escritura (**read-modify-write**) sean ejecutadas atómicamente (**sin interrupciones**).
- **Instrucción test-and-set (Intercambio atómico):**
 - x86: xchg
 - RISC-V: amoswap

int a = 3; ✓
int b = testAndSet(&a, 8); ✓
b = testAndSet(&a, 9); ✓

Atómico {
int TestAndSet(int *old_ptr, int new) {
 int old = *old_ptr; // fetch old value at old_ptr
 *old_ptr = new; // store 'new' into old_ptr
 return old; // return the old value
}



- La instrucción **test-and-set** devuelve el **valor anterior** y **actualiza** (de manera **simultánea**) el valor de una posición de memoria.

Implementación de un lock usando la instrucción test-and-set

- Implementación de un spinlock
la instrucción **test-and-set**
(xv6: [spinlock.h](#); [spinlock.c](#))

```

1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0: lock is available, 1: lock is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (testAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     //lock->flag = 0;
17     TestAndSet(&lock->flag, 0);
18 }

```

Sera que sirve?

TestAndSet (Atomica)

iv. Implementacion mediante la instruccion compare & swap

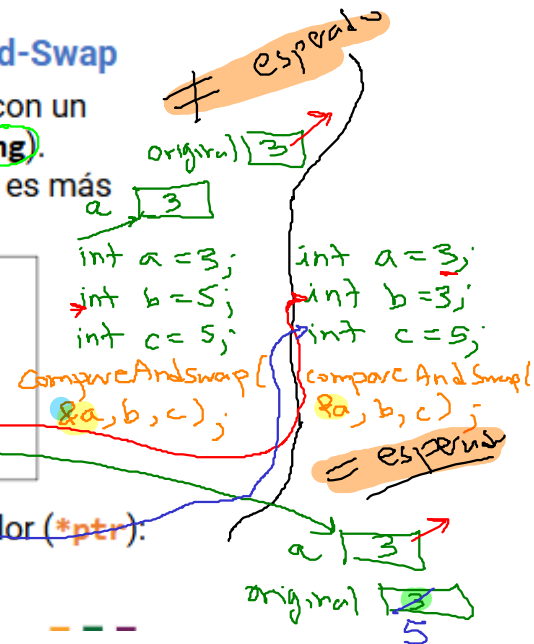
Implementación de un lock usando la instrucción Compare-And-Swap

- Esta instrucción atómica actualiza una localización de memoria con un valor anterior sólo cuando este es igual al esperado. (x86: **cmpxchg**).
- Esta instrucción trabaja de manera similar al **test-and-set** pero es más poderosa.

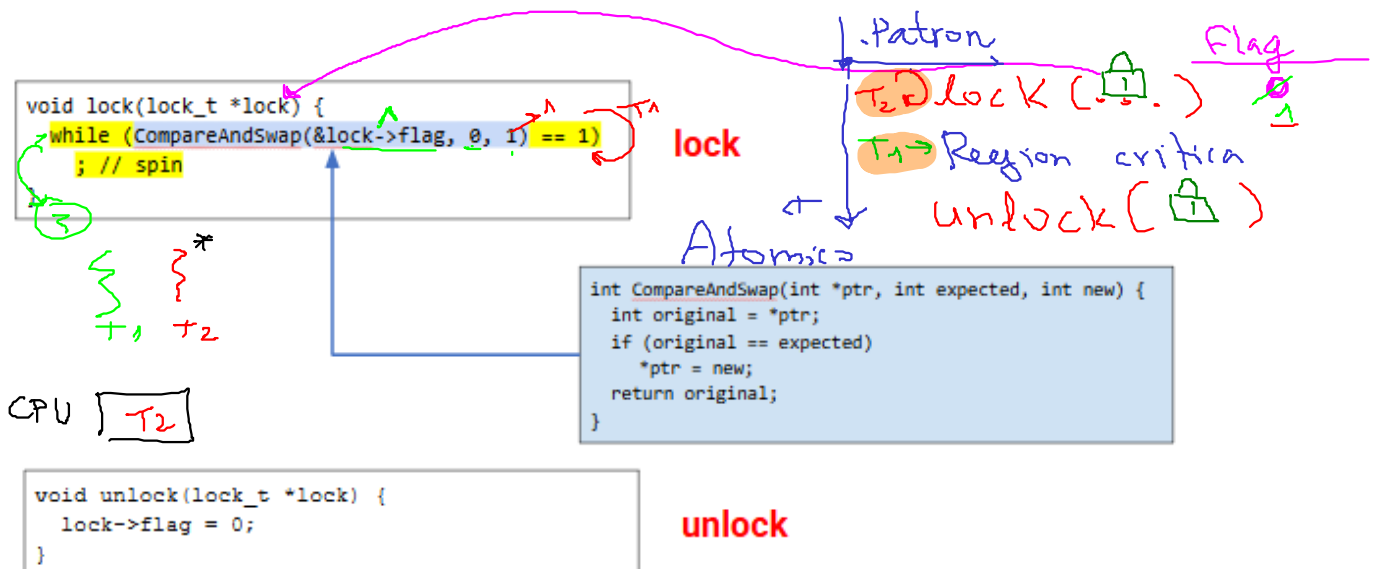
```

int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}

```



- Análisis del pseudocódigo implementado en C:** Chequea si el valor (***ptr**):
 - Si ***ptr == esperado** → actualiza la memoria al valor new.
 - Si ***ptr != esperado** → retorna el valor actual. ✓



v. Implementación mediante el uso de LL&SC

Implementación con instrucciones LL&SC (load linked – store conditional)

LL (Load linked): Opera como una instrucción load típica trayendo simplemente un valor de memoria y poniéndolo en un registro.

```
int LoadLinked(int *ptr) {  
    return *ptr;  
}
```

SC (Store conditional): El almacenamiento solo funciona si no se ha realizado otro almacenamiento previo en el mismo lugar:

- **Éxito:** Devuelve 1 y actualiza el valor de *ptr a new.
- **Falla:** Retorna 0 sin actualizar la memoria.

```
int StoreConditional(int *ptr, int value) {  
    if (no update to *ptr since LoadLinked to this address) {  
        *ptr = value;  
        return 1; // success!  
    }  
    else {  
        return 0; // failed to update  
    }  
}
```

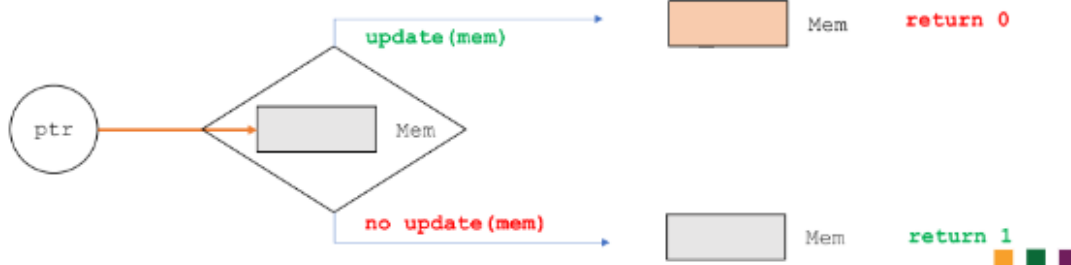
Implementación con instrucciones LL&SC (load linked – store conditional)

no update to *ptr since LoadLinked to this address – Que es?

```
int LoadLinked(int *ptr)
```



```
int StoreConditional(int *ptr, int value)
```



Implementación con instrucciones LL&SC (load linked – store conditional)

Implementación del lock

```
void lock(lock_t *lock) {  
    while (1) {  
        while (LoadLinked(&lock->flag) == 1)  
            ; // spin until it's zero  
        if (StoreConditional(&lock->flag, 1) == 1)  
            return; // if set-it-to-1 was a success: all done  
                    // otherwise: try it all over again  
    }  
}
```

```
void unlock(lock_t *lock) {  
    lock->flag = 0;  
}
```



Implementación con instrucciones LL&SC (load linked – store conditional)

¿Por qué funciona? - Análisis

```
void lock(lock_t *lock) {  
    while (1) {  
        while (LoadLinked(&lock->flag) == 1);  
        if (StoreConditional(&lock->flag, 1) == 1)  
            return;  
    }  
}
```

```
int LoadLinked(int *ptr) {  
    return *ptr;  
}
```

```
int StoreConditional(int *ptr, int value) {  
    if (no update to *ptr since LoadLinked to this address) {  
        *ptr = value;  
        return 1; // success!  
    }  
    else {  
        return 0; // failed to update  
    }  
}
```

```
void unlock(lock_t *lock) {  
    lock->flag = 0;  
}
```



vi. Implementación mediante Fetch & add

Implementación usando la instrucción fetch-and-add:

el contenido

Incrementa posición de memoria, retornando (de manera simultánea) el valor antiguo:

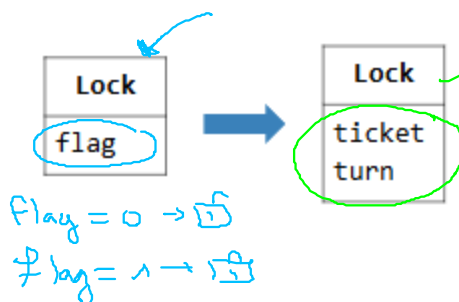
```
int FetchAndAdd(int *ptr) {  
    int old = *ptr;  
    *ptr = old + 1;  
    return old;  
}
```

(3) a [3] FetchAndAdd(&a);

Mellon-Crummey y Scott usaron esta instrucción para construir un mecanismo de acceso a regiones críticas conocido como ticket lock.

Ticket-lock usando Fetch-and-Add

En vez de un solo valor (**flag**) esta instrucción usa un **ticket** y un **turn** para construir el lock.



```
typedef struct __lock_t {  
    int ticket;  
    int turn;  
} lock_t;  
  
void lock_init(lock_t *lock) {  
    lock->ticket = 0;  
    lock->turn = 0;  
}  
  
void lock(lock_t *lock) {  
    int myturn = FetchAndAdd(&lock->ticket);  
    while (lock->turn != myturn) {  
        ; // spin  
    }  
}  
  
void unlock(lock_t *lock) {  
    lock->turn = lock->turn + 1;  
}
```



Ticket-lock - Análisis

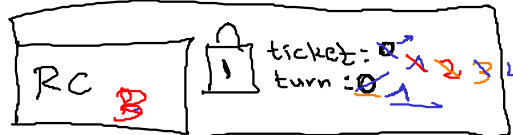
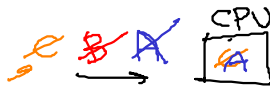
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn) {
        ; // spin
    }
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

	ticket	turn	R - Run S - Spin State (R/S)
lock_init	0	0	---
A lock()			
B lock()			
C lock()			
A unlock()			
B			
A lock()			
B unlock()			
C			
C unlock()			
A			
A unlock()			
C lock()			



Ticket-lock - Análisis

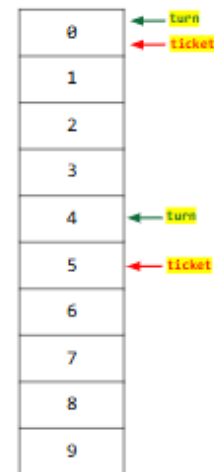
```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn) {
        ; // spin
    }
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

	ticket	turn	R - Run S - Spin State (R/S)
lock_init	0	0	---
A lock()	1	0	R
B lock()	2	0	S(1)
C lock()	3	0	S(2)
A unlock()	3	1	---
B			R
A lock()	4	1	S(3)
B unlock()	4	2	---
C			R
C unlock()	4	3	---
A			R
A unlock()	4	4	---
C lock()	5	4	R



Resumen

- Aspectos claves de los spinlocks:
 - Simples de implementar
 - El hilo que está en espera activa **desperdicia todo el quantum de tiempo** haciendo **nada** (solo chequea la variable).
 - Gran gasto de CPU, esto los hace costosos.
- Los **spinlocks** (y deshabilitar interrupciones en sistemas con un única CPU) son mecanismos de sincronización primitivos.
 - Son empleados para crear construcciones de sincronización de alto nivel
- Para evitar el gran gasto de CPU se requiere **asistencia del sistema operativo**.

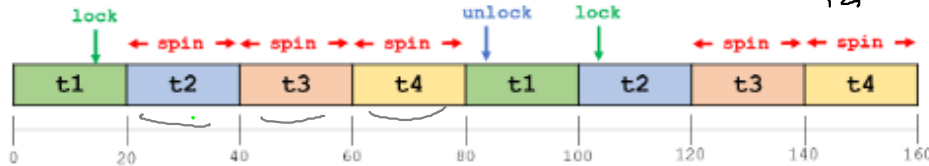
syscall).

5. Datos compartidos.

¿Cuál es el problema con los spinlocks?

- Mientras un hilo espera el lock gasta tiempo de uso del procesador chequeando un valor que no sabe cuándo cambiará (**gasto de CPU**).
- Entre más hilos (**N hilos**) más ciclos de CPU son gastados (**N - 1**), de modo que el problema empeora.

```
void lock(lock_t *lock) {  
    while (TestAndSet(&lock->flag, 1) == 1);  
}
```



T_2
 T_3
 T_4 \rightarrow $L(\text{lock})$
 $RC: T_1$

Idea clave ¿Cómo evitar el spinning?

¿Cómo desarrollar un lock que no gaste innecesariamente tiempo de la CPU iterando?

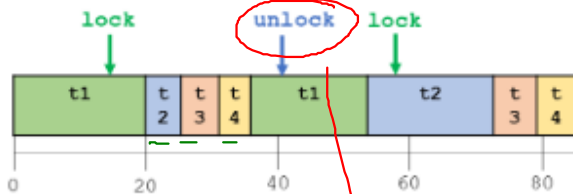


Yield en vez de spin

Cuando se requiera realizar espera activa, simplemente libera voluntariamente la CPU (llamado al sistema **yield**).

```
void lock(lock_t *lock) {  
    while (TestAndSet(&lock->flag, 1) == 1);  
}
```

```
void lock(lock_t *lock) {  
    while (TestAndSet(&lock->flag, 1) == 1)  
        yield(); // give up the CPU  
}
```

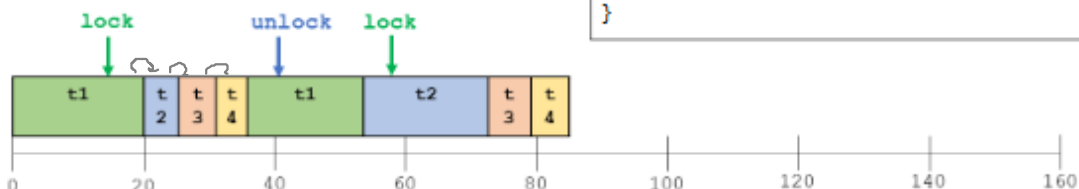


T_2 (222)
 T_3 (222)
 T_4 (222) \rightarrow $L(\text{lock})$
 $RC: T_1$

Yield en vez de spin

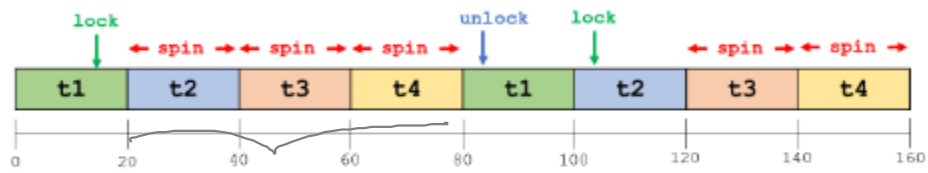
- El sistema operativo mueve el hilo de estado running a ready.
- Aún se gasta tiempo en los cambios de contexto.
- Puede existir inanición.

```
typedef struct __lock_t {  
    int flag;  
} lock_t;  
void init(lock_t *lock) {  
    lock->flag = 0;  
}  
void lock(lock_t *lock) {  
    while (TestAndSet(&lock->flag, 1) == 1)  
        yield();  
}  
void unlock(lock_t *lock) {  
    lock->flag = 0;  
}
```



Comparación

Sin `yield()`: $O(\text{threads} * \text{time_slice})$



Con `yield()`: $O(\text{threads} * \text{context_switch})$

