

Agenda 11 October

Read Chapters 28 – 32 (Concurrency): Module 5

Midterm Exam Tuesday, October 25

Student Questions

Attendance

Test_and_set, compare_and_swap

Spin-lock

Blocking alternatives

Revisiting the consumer-producer problem

Condition variables

The Test & Set Lock Instruction

- Spin-Lock Implementation with a TSL instruction

Note:
REGISTER is
“local”

enter_region:

```
TSL REGISTER,LOCK
CMP REGISTER,#0
JNE enter_region
RET
```

SPIN-LOCK

| copy lock to register and set lock to 1
| was lock zero?
| if it was nonzero, lock was set, so loop
| return to caller; critical region entered

leave_region:

```
MOVE LOCK,#0
RET
```

| store a 0 in lock
| return to caller

TSL causes the CPU executing the instruction to lock the memory bus to prohibit other CPUs from accessing memory until complete. Ensures reads and writes are done atomically

This safe locking result would NOT be Accomplished by
Disabling Interrupts in a multiprocessor system

Simple lock functionality using test-and-set

- If TestAndSet(flag,1) returns 1, it means the lock is held by someone else, so wait busily
- This lock is called a spinlock – spins until lock is acquired

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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



Critical Section Entry Solution using TestAndSet

- Shared (global) boolean variable “lock”, initialized to false.
- Solution:

```
do {
    lock(lock_t *lock1)           Acquire Lock
    critical section
    unlock(lock_t *lock1);         Release Lock
    remainder section
} while (TRUE);
```

Swap Instruction

- Definition: Swap, an atomic function

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Critical Section Solution using Swap

- **Shared** (global) Boolean variable **lock** initialized to FALSE; Each process has a **local Boolean variable** **key**

do {

key = TRUE;

while (key == TRUE)

Swap (&lock, &key);

critical section

lock = FALSE;

remainder section

} while (TRUE);

} CS Entry

} CS Exit

The Test & Set Function

Spin-Lock Implementation with a **swap** instruction

<div style="border: 1px solid black; padding: 5px; display: inline-block;">SPIN- LOCK</div>	enter_region:	
	MOVE REGISTER,#1	put a 1 in the register
	XCHG REGISTER,LOCK	swap the contents of the register and lock variable
	CMP REGISTER,#0	was lock zero?
	JNE enter_region	if it was non zero, lock was set, so loop
	RET	return to caller; critical region entered
	leave_region:	
	MOVE LOCK,#0	store a 0 in lock
	RET	return to caller

Lock is global;
REGISTER is a machine register
(therefore local)

Spinlock using compare-and-swap

- Another atomic instruction: compare-and-swap

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

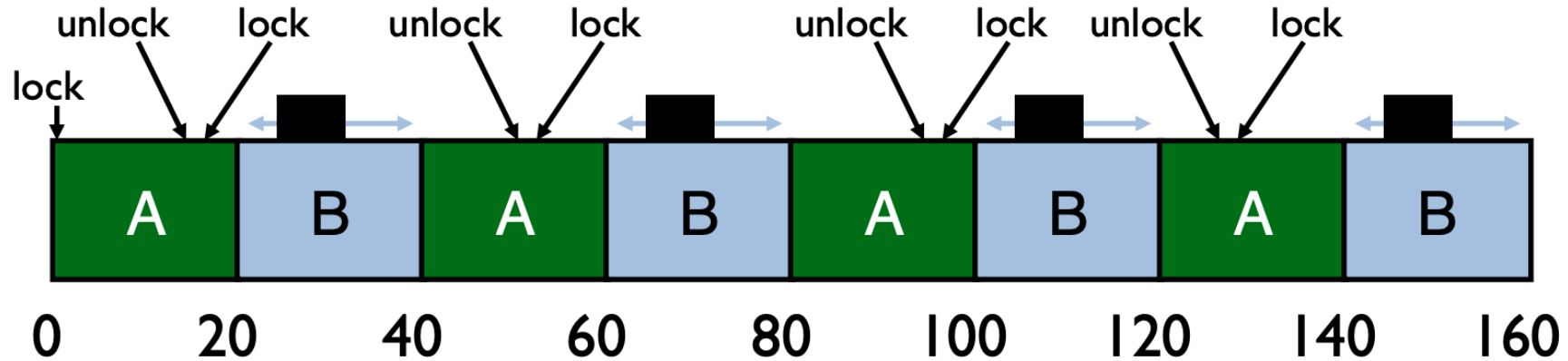
If unlocked, then lock; if locked,
save a memory write

- Spinlock using compare-and-swap

```
void lock(lock_t *lock) {  
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)  
        ; // spin  
}
```

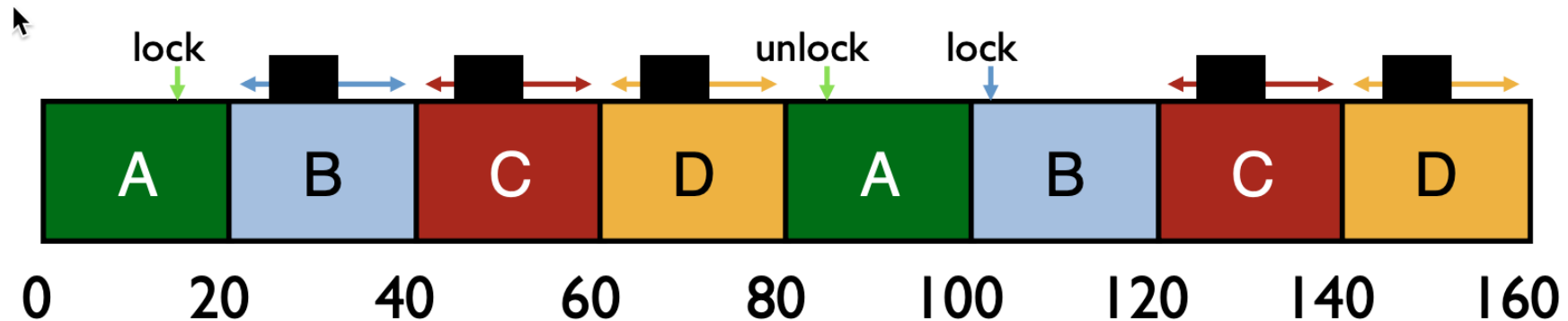
```
typedef struct __lock_t {  
    int flag; }  
lock_t;
```


Scheduler is Unaware of Locks/Unlocks



A is preempted and B is scheduled

B tries to access the CS and spins until preempted



CPU scheduler may run B, C, D instead of A, even though BCD are waiting for A

Alternative to spinning

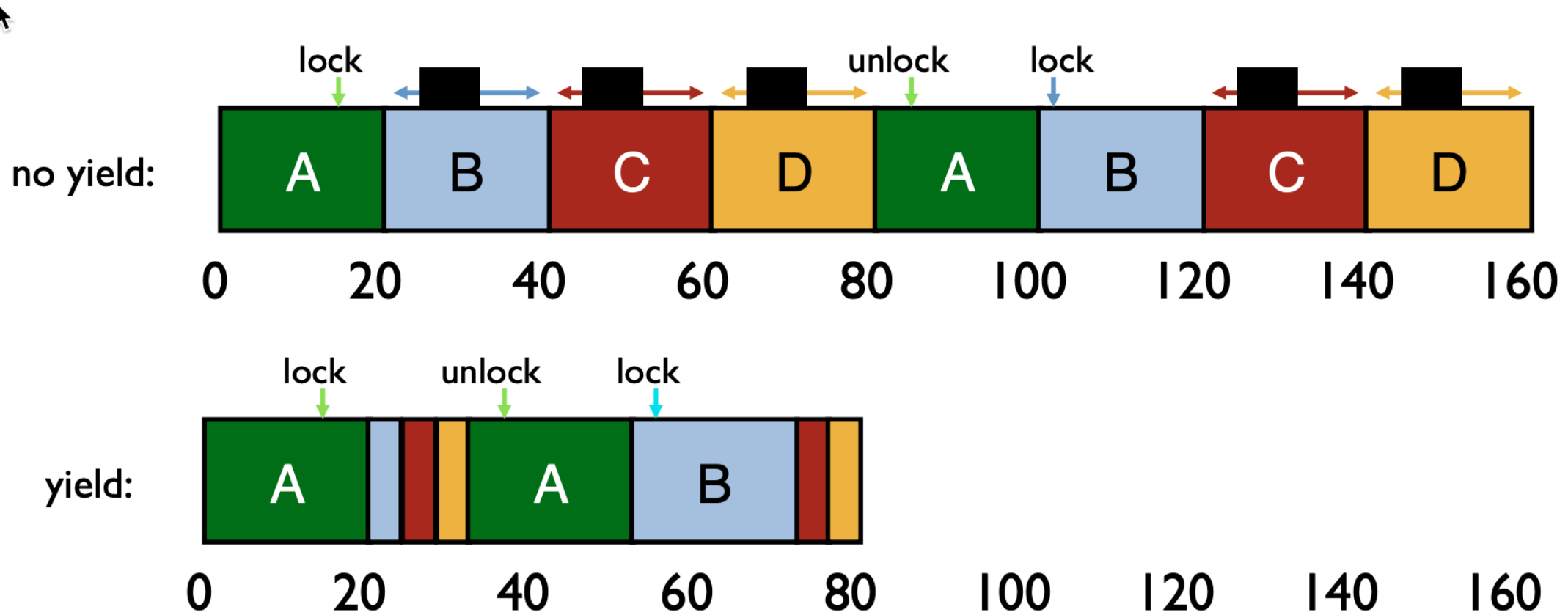
- Alternative to spinlock: a (sleeping) mutual exclusion mechanism
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later – `yield()` moves thread from running to ready state

```
void init() {
    flag = 0;
}

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

void unlock() {
    flag = 0;
}
```

Yield Instead of Spin



Spinlock vs. sleeping (blocking) mutex

- Most userspace lock implementations are of the sleeping or blocking mutex kind
 - CPU wasted by spinning contending threads
 - More so if a thread holds the spinlock and blocks for long time
- Locks inside the OS are always spinlocks
 - Why? Who will the OS yield to?
- When OS acquires a spinlock:
 - It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
 - It must not perform any blocking operation
 - never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

How should locks be used?

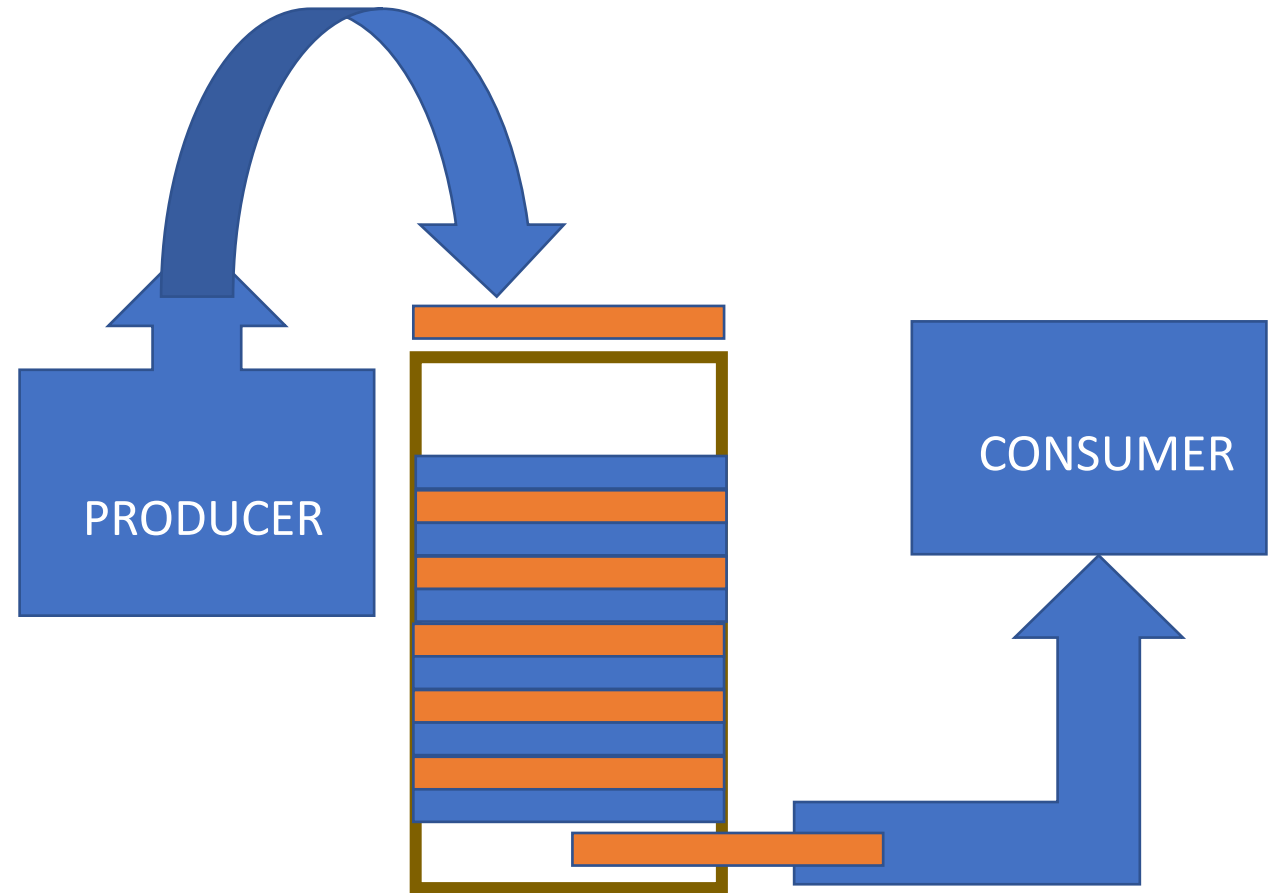
- A lock should be acquired before accessing any variable or data structure that is shared between multiple threads of a process
 - “Thread-safe” data structures
- All shared kernel data structures must also be accessed only after locking
- Coarse-grained vs. fine-grained locking: one big lock for all shared data vs. separate locks
 - Fine-grained allows more parallelism
 - Multiple fine-grained locks may be harder to manage
- OS only provides locks, correct locking discipline is left to the user

Example: Producer/Consumer problem

- A common pattern in multi-threaded programs
- Example: in a multi-threaded web server, one thread accepts requests from the network and puts them in a **buffer**. Worker threads get requests from this **buffer** and processes them. [See OS: Three Easy Pieces Chapter 30, 31]
- Setup:
 - one or more producer threads,
 - one or more consumer threads,
 - a shared buffer of bounded size

The Producer-Consumer Problem

- Bounded buffer: a concurrency problem
 - A fixed size buffer is shared among processes/threads
 - Producer 'creates' items and they are stored in the buffer
 - Consumer removes items from the buffer and consumes them
- Critical Conditions [points of synchronization]
 - Produce to a full buffer
 - Consume from an empty buffer
 - Need to sleep on critical events **AND** to be awakened when the condition is resolved



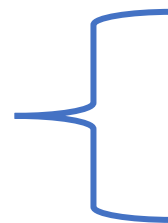
C/P Solution and Sleep / Wakeup

Solution A

- Spinlock solutions use Busy-Waiting for synchronization
 - Waste CPU resource
 - No Bounded Wait
 - **Deadlock or Priority Inversion [also 3 process case]**
 - High priority and Low priority processes share a CS

- BUSY WAITING – NO ORDERING OF ENTRY** controlled with a spinlock
- Hi awakens and needs to enter the same CS; busy-waits on the spinlock
 - Low can't execute and release CS since it is low priority
 - Will revisit again with process blocking (Priority Inversion Case)

Solution B

- 
- Sleep() blocks the caller until awakened
 - Wakeup(x) awakens a blocked process x

The **Producer**-Consumer Problem

- What is the critical section?

```
#define N 100
int count = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        if (count == N) sleep( );
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}
```

/* number of slots in the buffer */
/* number of items in the buffer */

/* repeat forever */
/* generate next item */
/* if buffer is full, go to sleep */
/* put item in buffer */
/* increment count of items in buffer */
/* was buffer empty? */

..The Producer-Consumer Problem

```
void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

/* repeat forever */
/* if buffer is empty, got to sleep */
/* take item out of buffer */
/* decrement count of items in buffer */
/* was buffer full? */
/* print item */

Is the solution good?

Race Condition in Solution

The Producer-Consumer Problem

- There is a problem in this Consumer Producer solution: access to **count** is **not constrained**
 - A wakeup can be issued for a process that is not really asleep (yet)
 - The wakeup will be 'missed'
- We need to be able to 'save wakeups' for use in the future

Now that we understand the problem, Let's look at different solutions

First: the Condition Variable Solution

Second: Semaphores Solution

Condition Variables: Another type of synchronization

- Locks allow one type of synchronization between threads – mutual exclusion
- Another common requirement in multi-threaded applications
 - **waiting and signaling**
 - e.g., Thread T1 wants to continue only after T2 has finished some task
- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: condition variables

Why Do we Need Condition Variables?

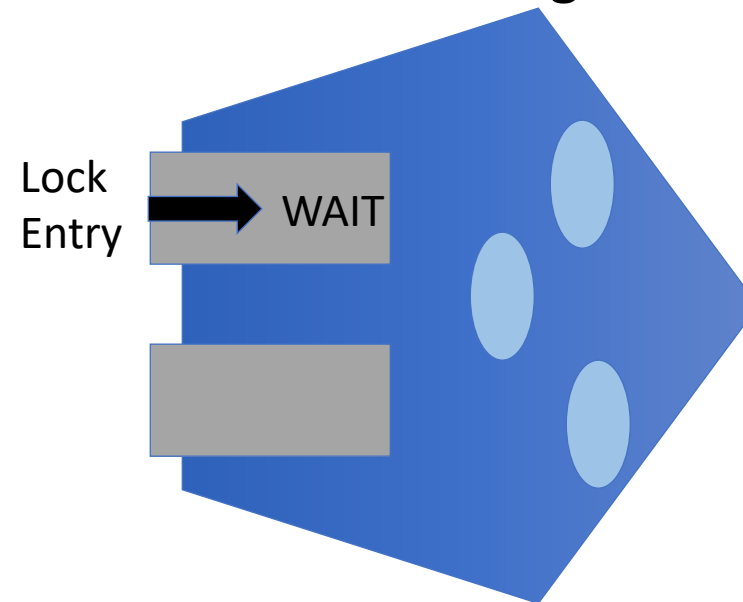
A Condition Variable is a Synchronization Object that lets a thread efficiently wait for a change to shared state protected by a lock.

A lock must always protect updates to shared state. Mutual Exclusion

We need a mechanism to safely block a thread while waiting for the shared resource to be available.

The Mechanism Must
ATOMICALLY
release the lock AND block the
thread

The Thread needs to reacquire
the lock as it is released from the
wait



Condition Variables

- A condition variable (CV) is a queue that a thread can be placed while waiting for some “condition” or state change
- Pthreads provides CV functions for user programs
 - OS has a similar functionality of wait/signal for kernel threads
- Called only when holding a lock

General Definitions of Operations:

- **Wait**: atomically release lock and relinquish processor until signaled
- **Signal**: wake up a waiter, if any
- **Broadcast**: wake up all waiters, if any

Condition Variable Operations

- a variable type that can be used for Block/Wakeup (and discussed later: “**within monitor**” event waits):
- **Condition Variable**
 - If **c** is a condition variable, then 2 operations can be applied to it:
 - wait(c) and signal(c) [broadcast is an optional operation]
 - These two operations are **not** the same as P and V, the operations defined on semaphores. [to be discussed later]
- **wait(c)** : The calling process is **blocked** and is entered on a queue (a ‘condition queue’) of processes blocked on this condition (c), i.e. have also executed wait(c) commands. We assume that the wait queues are FIFO.
- **signal(c)** : If the queue for c is not empty, then wake up the first process on the queue.

POSIX Condition Variable Functions

```
pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * lock);
```

Blocks on a condition variable. It must be called with *lock* locked

```
pthread_mutex_lock( lock )
```

```
pthread_cond_signal(pthread_cond_t * cond);
```

If there is a waiter on the condition queue, release the waiter

No waiter, then no effect