Operating Systems
COMS(3010A)
Locks

Branden Ingram

branden.ingram@wits.ac.za

Office Number : ???

Recap

- Concurrency
- Threads
- Thread API

Sharing Objects

- If a program has “independent threads” that operate on completely separate subsets of memory, we can reason about each thread separately

Sharing Objects

- If a program has “independent threads” that operate on completely separate subsets of memory, we can reason about each thread separately

- However, most multi-threaded programs have both:
    - per-thread state (stack, registers)
    - shared state (heap)

- Threads which can read and write the shared state are called “Cooperating threads”

Sharing Objects

- If a program has “independent threads” that operate on completely separate subsets of memory, we can reason about each thread separately

- However, most multi-threaded programs have both:
  - per-thread state (stack, registers)
  - shared state (heap)

- Threads which can read and write the shared state are called “Cooperating threads”

- Unfortunately then cooperating threads share state, writing correct multithreaded programs becomes much more difficult

Sharing Objects

- If a program has “independent threads” that operate on completely separate subsets of memory, we can reason about each thread separately

- However, most multi-threaded programs have both:
  - per-thread state (stack, registers)
  - shared state (heap)

- Threads which can read and write the shared state are called “Cooperating threads”

- Unfortunately then cooperating threads share state, writing correct multithreaded programs becomes much more difficult
  - Program execution depends on the possible interleaving of threads access to shared state
  - Program execution can be non deterministic
  - Compilers and processor hardware can reorder instructions

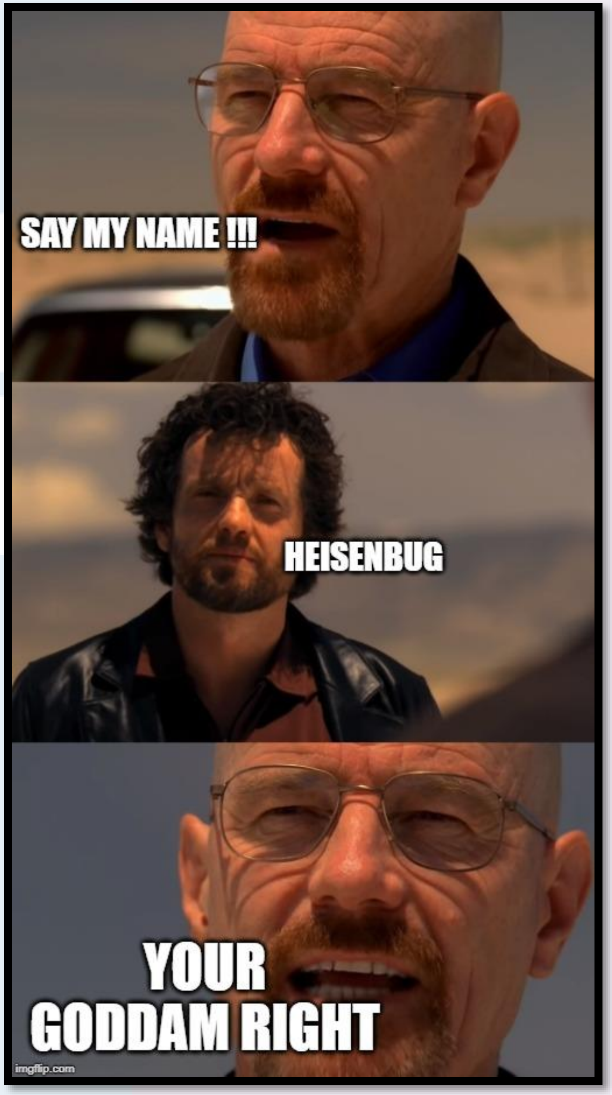
How can we reason about all possible interleavings of threads' actions?

- If two threads write a shared variable
  - Thread A writes with value 1
  - Thread B write with value 2
    - The final value will depend on the order in which it was written
- This problem explodes in complexity when programs grow
- Programmers cannot make any assumptions on the relative speed of threads

How can we debug programs with behaviours that change across runs?

- Different runs of the same program might produce different results
  - The scheduler might make different decisions
  - The processor might run at a different frequency
  - Another concurrent process may affect the operations

- Bugs might arise in one execution or behave in a different manner across executions
  - “Heisenbugs” are bugs that change behaviour or disappear when you try examine them
  - “Bohr bugs” are the opposite (deterministic bugs)



How can we reason about thread interleavings when compilers and hardware may reorder their operations?

- Modern compilers and hardware reorder instructions to improve performance

- bool checked1 = true;
- bool checked2 = true;

How can we reason about thread interleavings when compilers and hardware may reorder their operations?

- **Modern compilers and hardware reorder instructions to improve performance**

- Walk into a cafe and ask for a drink and a sandwich. The person behind the counter hands you the sandwich (which is right next to him), then walks to the fridge to get your drink. Do you care that he gave them to you in the "wrong" order? Would you rather he did the slow one first, simply because that's how you gave the order?

- https://stackoverflow.com/questions/37725497/how-does-memory-reordering-help-processors-and-compilers

How can we reason about thread interleavings when compilers and hardware may reorder their operations?

- **Modern compilers and hardware reorder instructions to improve performance**

- Walk into a cafe and ask for a drink and a sandwich. The person behind the counter hands you the sandwich (which is right next to him), then walks to the fridge to get your drink. Do you care that he gave them to you in the "wrong" order? Would you rather he did the slow one first, simply because that's how you gave the order?

- https://stackoverflow.com/questions/37725497/how-does-memory-reordering-help-processors-and-compilers

- **This reordering is generally invisible to single-threaded programs**
- **However, reordering can become visible when accessing shared state or as a result of process inteleaving**

Structured Synchronization

- Given these challenges, multithreaded code can introduce subtle, non deterministic non reproducible bugs

- A naive approach to these problems would be an ad hoc reasoning to the effect of process interleaving

- A better approach is structured synchronization
  - Structure the program to facilitate reasoning about concurrency
  - Use a set of standard synchronization techniques to control access to shared states

Challenges

- Race Conditions : Occurs when the behaviour of a program depends on the interleaving of operations of different threads

Thread A
x = x + 1;

Thread B
x = x + 2;

Interleaving 1

```
load r1,x
add r2,r1,1
store x,r2
            load,r1,x
            add,r2,r1,x
            store x,r2
```

Final x=3

Challenges

- **Race Conditions : Occurs when the behaviour of a program depends on the interleaving of operations of different threads**

Thread A	**Thread B**
x = x + 1;	**x = x + 2;**

Interleaving 1

load r1,x	
add r2,r1,1	
store x,r2	
	load,r1,x
	add,r2,r1,x
	store x,r2

Final x=3

Interleaving 2

load r1,x	
	load,r1,x
add r2,r1,1	
	add,r2,r1,x
store x,r2	
	store x,r2

Final x=2

Challenges

- **Race Conditions : Occurs when the behaviour of a program depends on the interleaving of operations of different threads**

Thread A	**Thread B**
x = x + 1;	**x = x + 2;**

r1, r2 = registers, initially set to 0

Interleaving 1

Thread A	Thread B
load r1,x	
add r2,r1,1	
store x,r2	
	load,r1,x
	add,r2,r1,2
	store x,r2

Final x=3

Interleaving 2

Thread A	Thread B
load r1,x	
	load,r1,x
add r2,r1,1	
	add,r2,r1,2
store x,r2	
	store x,r2

Final x=2

Interleaving 3

Thread A	Thread B
load r1,x	
	load,r1,x
add r2,r1,1	
	add,r2,r1,2
	store x,r2
store x,r2	

Final x=1

Challenges

- Atomic Operations : are program operations that run completely independently of any other processes.

- During an atomic operation, a processor can read and write a location during the same data transmission.

- In this way, another input/output mechanism or processor cannot perform memory reading or writing tasks until the atomic operation has finished.

- An “atomic” operation is one which executes as if were not interrupted in space or time

- Load, Store

Challenges

- Too Much Milk : models the problem of coordinating access to shared memory by multiple threads using only loads and stores

- Problem : two roommates share a fridge, these are good roommates and always make sure the fridge is stocked with milk

Challenges

- <u>Too Much Milk</u> : models the problem of coordinating access to shared memory by multiple threads using only loads and stores

- **Problem : two roommates share a fridge, these are good roommates and always make sure the fridge is stocked with milk**

- **Scenario :**

Time	Roommate 1's actions	Roommate 2's actions
3:00	Look in fridge out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home put milk away
3:31		Oof

Challenges

- If the only atomic operations on shared state are atomic loads and stores to memory
- Is there a solution to the Too Much Milk Problem that satisfies :
  - Safety – The program never enters a bad state
    - Never more than one person buys milk
  - Liveness – The program eventually enters a good state
    - If milk is needed, someone eventually buys it

Conditions to ensure for atomic operations ^^

Challenges

- **A basic idea could be to leave a note**

```
If(milk==0){                          // if no milk
        if(note==0){                  // if no note
                note=1;               // leave note
                milk++;               // buy milk
                note=0;               // remove note
        }
}
```

- **Unfortunately this solution can violate safety**

Challenges

- **We have created a Heisenbug which occasionally causes the program to fail**

Thread A

```
If(milk==0){
```

Scheduling interrupt occurs (interleaving)

Thread B

```
If(milk==0){
        if(note==0){
                note=1;
                milk++;
                note=0;
        }
}
```

```
        if(note==0){
                note=1;
                milk++;
                note=0;
        }
}
```

Challenges

- Another idea is two have notes per thread which can be used to check if another is contemplating buying milk

Thread 1

```
noteA=1;
if(noteB==0){
        if(milk==0){
                milk++;
        }
}
noteA=0;
```

Thread 2

```
noteB=1;
if(noteA==0){
        if(milk==0){
                milk++;
        }
}
noteB=0;
```

- Unfortunately this solution can violate liveness
  - It is possible for both threads to check the other threads note and to both not decide to buy milk

Challenges

- **Additionally to solve this new problem is by ensuring atleast one of the threads determines whether the other thread has bought milk**

Thread 1

```
noteA=1;
While(noteB==1){
        DoNothing;
}
If(milk==0){
        milk++;
}
noteA=0;
```

Thread 2

```
noteB=1;
if(noteA==0){
        if(milk==0){
                milk++;
        }
}
noteB=0;
```

- **Here we have a solution which is both safe and live**

Challenges

- Additionally to solve this new problem is by ensuring atleast one of the threads determines whether the other thread has bought milk

Thread 1

```
noteA=1;
While(noteB==1){
        DoNothing;
}
If(milk==0){
        milk++;
}
noteA=0;
```

Thread 2

```
noteB=1;
if(noteA==0){
        if(milk==0){
                milk++;
        }
}
noteB=0;
```

- Here we have a solution which is both safe and live
  - This solution is also inefficient – While thread 1 is in this loop "busy-waiting" and consuming CPU resources

Challenges

- A better solution is to utilise synchronization objects to coordinate different threads' access to shared state

```
Kitchen::buyIfNeeded(){
        lock.acquire();
        if(milk==0){
                milk++;
        }
        lock.release();
}
```

- We can use a primitive synchronisation object called a lock
    - It is designed to enforce a mutual exclusion control policy
        - Only one thread can own a lock at a given time

Locks

- A lock can be in one of two states
  - BUSY/FREE

- A lock is initially FREE
- Lock::acquire waits until a lock is FREE and then automatically makes it BUSY
- Lock::release makes the lock FREE

- Properties
  - Mutual Exclusion – At most one thread can hold a lock
  - Progress – At some point some threads succeeds at obtaining a lock
  - Bounded Waiting – waiting time is bounded by the number of threads

Locks

- **Ensure that any critical section executes as if it were a single atomic instruction.**

An example: the canonical update of a shared variable

```
balance = balance + 1;
```

Add some code around the critical section

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks

- **Lock variable holds the state of the lock.**
  - **available (or unlocked or free)**
    - No thread holds the lock.
  - **acquired (or locked or held)**
    - Exactly one thread holds the lock and presumably is in a critical section.

Locks

- **lock()**
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the critical section**.**
    - This thread is said to be the owner of the lock.

  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there

<u>Pthread Locks - mutex</u>

The name that the POSIX library uses for a <u>lock</u>.

Used to provide mutual exclusion between threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

We may be using *different locks* to protect *different variables* → Increase concurrency (a more **fine-grained** approach).

Building a Lock

- Efficient locks provided mutual exclusion at low cost.
- Building a lock need some help from the hardware and the OS

Evaluating a Lock

- Mutual exclusion
  - Does the lock work, preventing multiple threads from entering a critical section?

- Fairness
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)

- Performance
  - The time overheads added by using the lock

Controlling Interrupts

- **Disable Interrupts for critical sections**
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.

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

Controlling Interrupts

- **Disable Interrupts for critical sections**
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.

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

- **Problem:**
  - **Require too much trust in applications**
    - Greedy (or malicious) program could monopolize the processor.
  - **Do not work on multiprocessors**
  - **Code that masks or unmasks interrupts be executed slowly by modern CPUs**

Why hardware support is needed?

- **First attempt: Using a flag denoting whether the lock is held or not.**
  - The code below has problems.

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

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

void lock(lock_t *mutex) {
     while (mutex->flag == 1)  // TEST the flag
               ;  // spin-wait (do nothing)
     mutex->flag = 1;  // now SET it !
}

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

Why hardware support is needed?

- **Problem 1: No Mutual Exclusion (assume flag=0 to begin)**

Thread1	**Thread2**
call `lock()` `while (flag == 1)` interrupt: switch to Thread 2	

	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1

`flag = 1;` **// set flag to 1 (too!)**	

Why hardware support is needed?

- **Problem 1: No Mutual Exclusion (assume flag=0 to begin)**

Thread1	**Thread2**
call `lock()` `while (flag == 1)` interrupt: switch to Thread 2	

	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1

`flag = 1;` **// set flag to 1 (too!)**	

- **Problem 2: Spin-waiting wastes time waiting for another thread.**

Why hardware support is needed?

- **Problem 1: No Mutual Exclusion (assume flag=0 to begin)**

Thread1	**Thread2**
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` **// set flag to 1 (too!)**	

- **Problem 2: Spin-waiting wastes time waiting for another thread.**

- **So, we need an atomic instruction supported by Hardware!**
  - test-and-set instruction, also known as atomic exchange

Test And Set (Atomic Exchange)

- **An instruction to support the creation of simple locks**

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

- return(testing) old value pointed to by the ptr.
- Simultaneously update(setting) said value to new.
- This sequence of operations is performed atomically.

A Simple Spin Lock using test-and-set

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

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

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

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

- **Note**: To work correctly on *a single processor*, it requires a preemptive scheduler.

<u>Evaluating Spin Locks</u>

Correctness: yes

The spin lock only allows a single thread to entry the critical section.

Fairness: no

Spin locks <u>don't provide any fairness</u> guarantees.

Indeed, a thread spinning may spin *forever*.

Performance:

In the single CPU, performance overheads can be quite *painful*.

If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*.

Compare-And-Swap

- **Test whether the value at the address(ptr) is equal to expected.**
  - If so, update the memory location pointed to by ptr with the new value.
  - In either case, return the actual value at that memory location.

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

Compare-and-Swap hardware atomic instruction (C-style)

Compare-And-Swap

- **Test whether the value at the address(ptr) is equal to expected.**
  - If so, update the memory location pointed to by ptr with the new value.
  - In either case, return the actual value at that memory location.

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

Compare-and-Swap hardware atomic instruction (C-style)

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

Spin lock with compare-and-swap

Load-Linked and Store-Conditional

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

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

Load-linked And Store-conditional

- **The store-conditional only succeeds if no intermittent store to the address has taken place.**
    - success: return 1 and update the value at ptr to value.
    - fail: the value at ptr is not updated and 0 is returned.

Load-Linked and Store-Conditional

```
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;
}
```

Using LL/SC To Build A Lock

Load-Linked and Store-Conditional

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
            ; // spin
}
```

A more concise form of the `lock()` using LL/SC

Fetch-And-Add

- **Atomically increment a value while returning the old value at a particular address.**

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

Fetch-And-Add Hardware atomic instruction (C-style)

Ticket Lock

- Ticket lock can be built with fetch-and add.
  - Ensure progress for all threads. -> fairness

```
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) {
        FetchAndAdd(&lock->turn);
}
```

So Much Spinning

- **Hardware-based spin locks are simple and they work.**

- **In some cases, these solutions can be quite inefficient.**
  - Any time a thread gets caught spinning, it wastes an entire time slice doing nothing but checking a value.

How To Avoid *Spinning*?
We'll need OS Support too!

A Simple Approach: Just Yield

- When you are going to spin, give up the CPU to another thread.
  - OS system call moves the caller from the running state to the ready state.
  - The cost of a context switch can be substantial and the starvation problem still exists.

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

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

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

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- **Queue to keep track of which threads are waiting to enter the lock.**
- park()
    - **Put a calling thread to sleep**
- unpark(threadID)
    - **Wake a particular thread as designated by threadID.**

Using Queues: Sleeping Instead of Spinning

```
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
…
```

Lock With Queues, Test-and-set, Yield, And Wakeup

Using Queues: Sleeping Instead of Spinning

```c
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)

Wakeup/waiting race

- In case of releasing the lock (thread A) just before the call to park() (thread B) -> Thread B would sleep forever (potentially).

Wakeup/waiting race

- **In case of releasing the lock (thread A) just before the call to park() (thread B) -> Thread B would sleep forever (potentially).**

- **Solaris solves this problem by adding a third system call: setpark().**
  - By calling this routine, a thread can indicate it is about to park.
  - If it happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping.

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
park();
```

Code modification inside of `lock()`

Futex

- **Linux provides a futex (is similar to Solaris's park and unpark).**
    - **futex_wait(address, expected)**
        - Put the calling thread to sleep
        - If the value at address is not equal to expected, the call returns immediately.
    - **futex_wake(address)**
        - Wake one thread that is waiting on the queue.

Futex

- **Snippet from lowlevellock.h in the nptl library**
  - The high bit of the integer v: track whether the lock is held or not
  - All the other bits : the number of waiters

```
void mutex_lock(int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (this is the fastpath) */
    if (atomic_bit_test_set(mutex, 31) == 0)
        return;
    atomic_increment(mutex);
    while (1) {
        if (atomic_bit_test_set(mutex, 31) == 0) {
            atomic_decrement(mutex);
            return;
        }
        /* We have to wait now. First make sure the futex value
           we are monitoring is truly negative (i.e. locked). */
        v = *mutex;
        …
```

Linux-based Futex Locks

Futex

```
                if (v >= 0)
                        continue;
                futex_wait(mutex, v);
        }
}

void mutex_unlock(int *mutex) {
        /* Adding 0x80000000 to the counter results in 0 if and only if
           there are not other interested threads */
        if (atomic_add_zero(mutex, 0x80000000))
                return;
        /* There are other threads waiting for this mutex,
           wake one of them up */
        futex_wake(mutex);
}
```

Linux-based Futex Locks (Cont.)

Two-Phase Locks

A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.

Two-Phase Locks

A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.

First phase

The lock spins for a while, *hoping that* it can acquire the lock.

If the lock is not acquired during the first spin phase, a second phase is entered

<u>Two-Phase Locks</u>

A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.

- **First phase**
  - The lock spins for a while, *hoping that* it can acquire the lock.
  - If the lock is not acquired during the first spin phase, <u>a second phase</u> is entered,
- **Second phase**
  - The caller is put to sleep.
  - The caller is only woken up when the lock becomes free later.