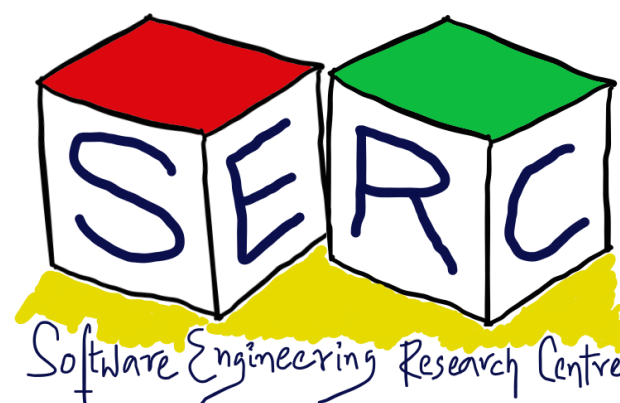


CS3.301 Operating Systems and Networks

Concurrency - Locks

Karthik Vaidhyanathan

<https://karthikvaidhyanathan.com>



Acknowledgement

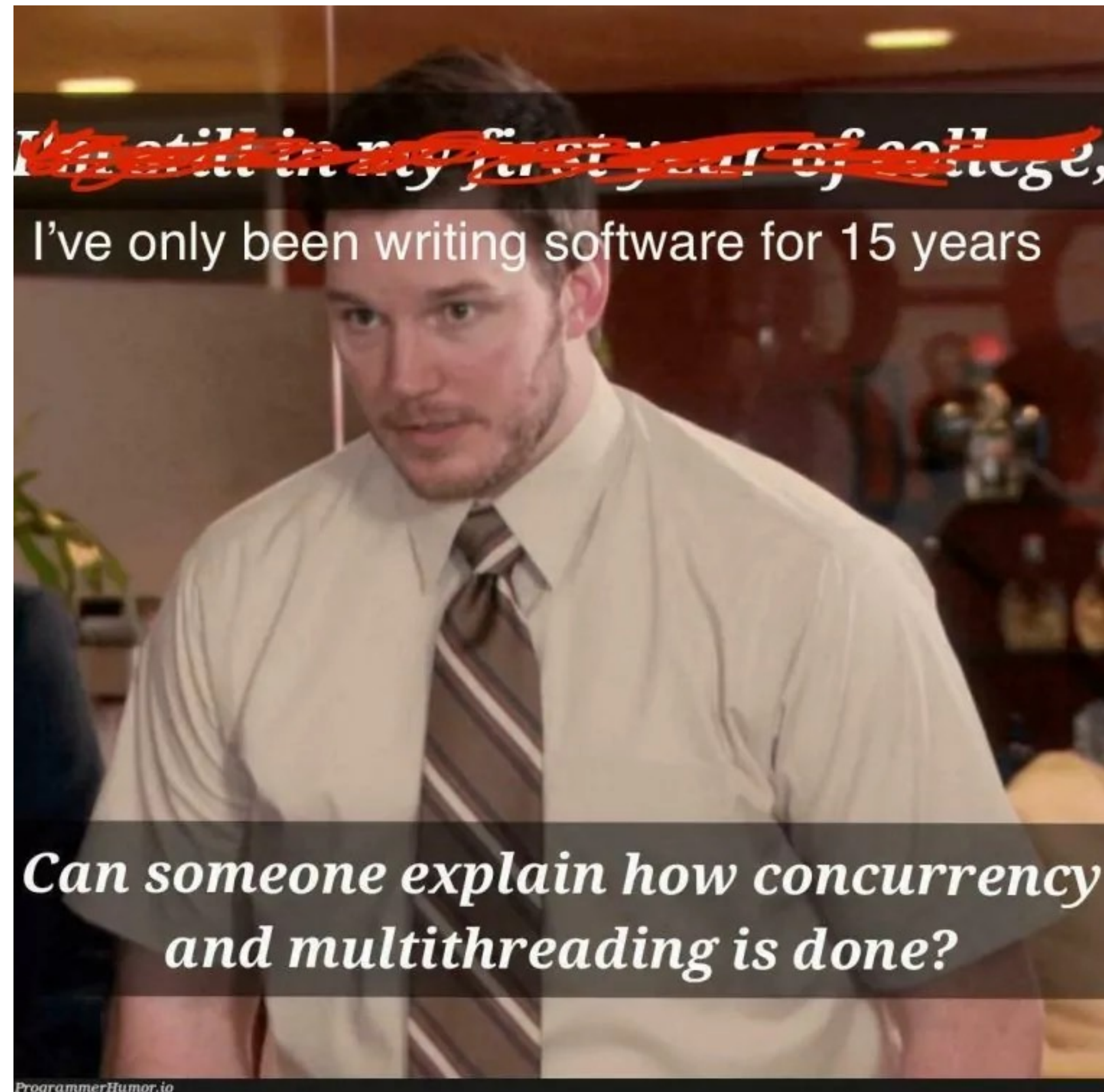
The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in three easy pieces by Remzi et al.



Concurrency can be tricky!



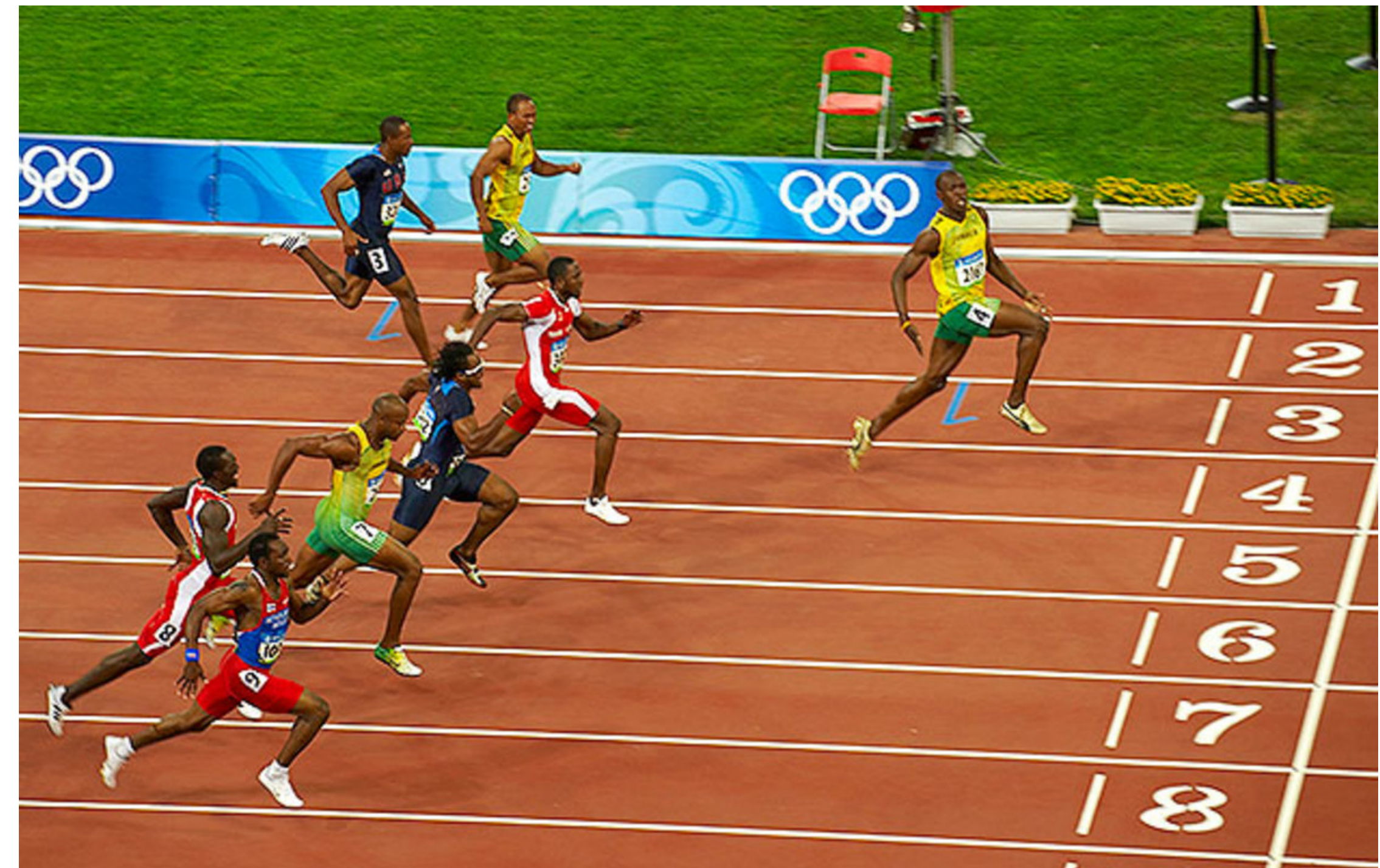
What can happen?

OS	Thread 1	Thread 2	Counter and Register (Initial value of counter = 50)
	<pre>mov 0x8049a1c, %eax add \$0x1, %eax</pre>		<p>eax = 51 counter = 50</p>
<p>interrupt Save T1' state Restore T2's state</p>			
		<pre>mov 0x8049a1c, %eax add \$0x1, %eax Mov %eax, 0x8049a1c</pre>	<p>eax = 51 counter = 51</p>
<p>interrupt Save T2' state Restore T1's state</p>			



Race Condition and Critical Section

- **Race Condition: Condition where**
 - Multiple threads executing concurrently and
 - results depend on order of execution (time)
 - Scheduler can swap threads, also interrupts
 - Non-deterministic results
- **Critical Section:**
 - The section of code that is shared between the threads (leads to race conditions)
 - Shared variables or data



Source: <https://www.si.com/olympics/2016/07/14/usain-bolt-2016-rio-olympics>



What can be done?

Bring in Atomicity

- What we want here is mutual exclusion!
 - When one thread is accessing critical section, others should wait
 - No two threads should access critical section at the same time
- In other words, **atomicity** needs to be provided
 - What if there was one instruction in assembly:
 - memory-add 0x8049a1c, &0x1 - Reality is not this!!

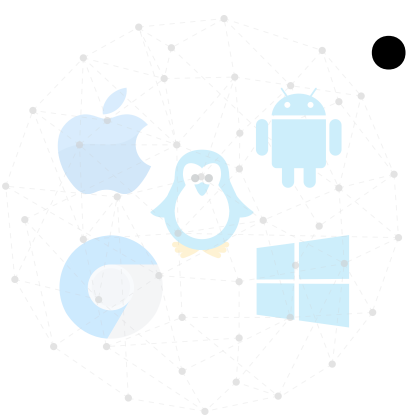
```
mov 0x8049a1c, %eax  
add &0x1, %eax  
mov %eax, 0x8049a1
```

This should execute atomically!!



What we need?

- Need to build synchronization primitives
 - Hardware + software support
 - Ensure that critical section is accessed in synchronised and controlled manner
- **One part:** Build some primitives for synchronisation
 - This will ensure atomicity (avoid race conditions)
- **Second part:** Ensure every thread gets access!
 - No one should starve



Some Issues needs to be addressed

- What support do we need from hardware?
- What support is needed from software?
- How to build primitives correctly and effectively?
- How can programs use these primitives?



Locks: A Basic Idea

- Lets go back to the shared variable code in **Critical Section (CS)**:

counter = counter + 1

- What if we can have a lock surrounding the statement

```
lock_t mutex; //a global lock
....
lock (&mutex); // lock the crticial section
counter = counter + 1;
unlock (&mutex); // release the lock
```



What are locks?

- Lock here is just a variable
- The lock variable holds the state of lock at any instant of time
- It is either available (free) or acquired:
 - **Available:** No threads hold the lock
 - **Acquired:** Lock not available, one thread is holding it and in CS
- We can also enrich with more information - which thread holds the lock, create a queue for threads to get locks, etc.



Lock and Unlock

- The thread that holds the lock - **Owner**
- Owner needs to call unlock to free the lock
 - The lock becomes free
 - There may be threads waiting for the lock, one of them will acquire it
 - The next thread with lock enters CS
 - When no thread is waiting for the lock, the lock stays as free

- **How to go about building a lock?**



How to go about building a lock?

- Think about classrooms and locks
- In the physical locks itself there are many options
- Many hardware primitives have been added to instruction set architecture to support locks
- The way in which the primitives are used + OS support forms the key



Criteria to evaluate

- **Mutual Exclusion**

- Does the lock prevent multiple threads from entering CS at same time?

- **Fairness**

- Does each thread get a fair chance to enter into the CS? - Avoid starvation!!

- **Performance**

- When there is only one thread what is the overhead?
- Multiple threads on single CPU - What about overhead?
- Multiple threads on multiple CPU - What's are overhead?



Why threading is challenging?

- Two threads running at the same time
 - Interrupts - Can we disable interrupt inside a lock?

```
void lock()
{
    DisableInterrupts();
    //disables interrupts
}

void unlock()
{
    EnableInterrupt();
}
```

Main positive approach is simplicity

Are there any issues?



There are Negatives

- Thread gets a **very high privilege**
 - Thread can switch on and off the interrupts
 - Arbitrary thread can monopolize the processor
 - Errant program could call lock() and get into endless loop
- The approach does not work on multiple processor systems
 - Even if interrupts are disabled, other threads can run on different processor
- **Inefficiency:** Code that masks or unmask interrupts are executed slowly
- Interrupt control is used in limited context as mutual exclusion primitive (**inside OS**)



Can we try with a software lock?

Software based lock

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

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

void lock(lock_t *mutex)
{
    while(mutex->flag==1)
        //do nothing
    mutex -> flag = 1;
}

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

- Use software based locking mechanism
- Create a lock which has a flag value
- Every time a thread wants to enter CS
 - Invoke lock function, if flag = 1, wait for the lock to be available
 - Else acquire the lock and enter CS
- **Do you foresee some problem here?**

Simple Trace

Thread 1	Thread 2
Call lock() while (flag== 1) Interrupt to Thread 2	
	Call lock() While (flag==1) flag = 1; Interrupt to thread 1
flag = 1	

- No mutual exclusion - Both threads have flag set to 1
- Performance overhead due to spin-waiting



Working Spin Lock with Test-And-Set

Test and Set Instruction

```
Test and Set Lock

int TestAndSet(int *ptr, int new)
{
    int old = *ptr;
    *ptr = new;
    return old;
}
```

C Pseudocode

- Simplest hardware primitive - test-and-set or atomic exchange instruction
- Sequence of instructions executed atomically
- Enables testing of old values while setting the main to a new value
- Think about implementing a CS code with this lock!



Implementing using Test and Set Lock

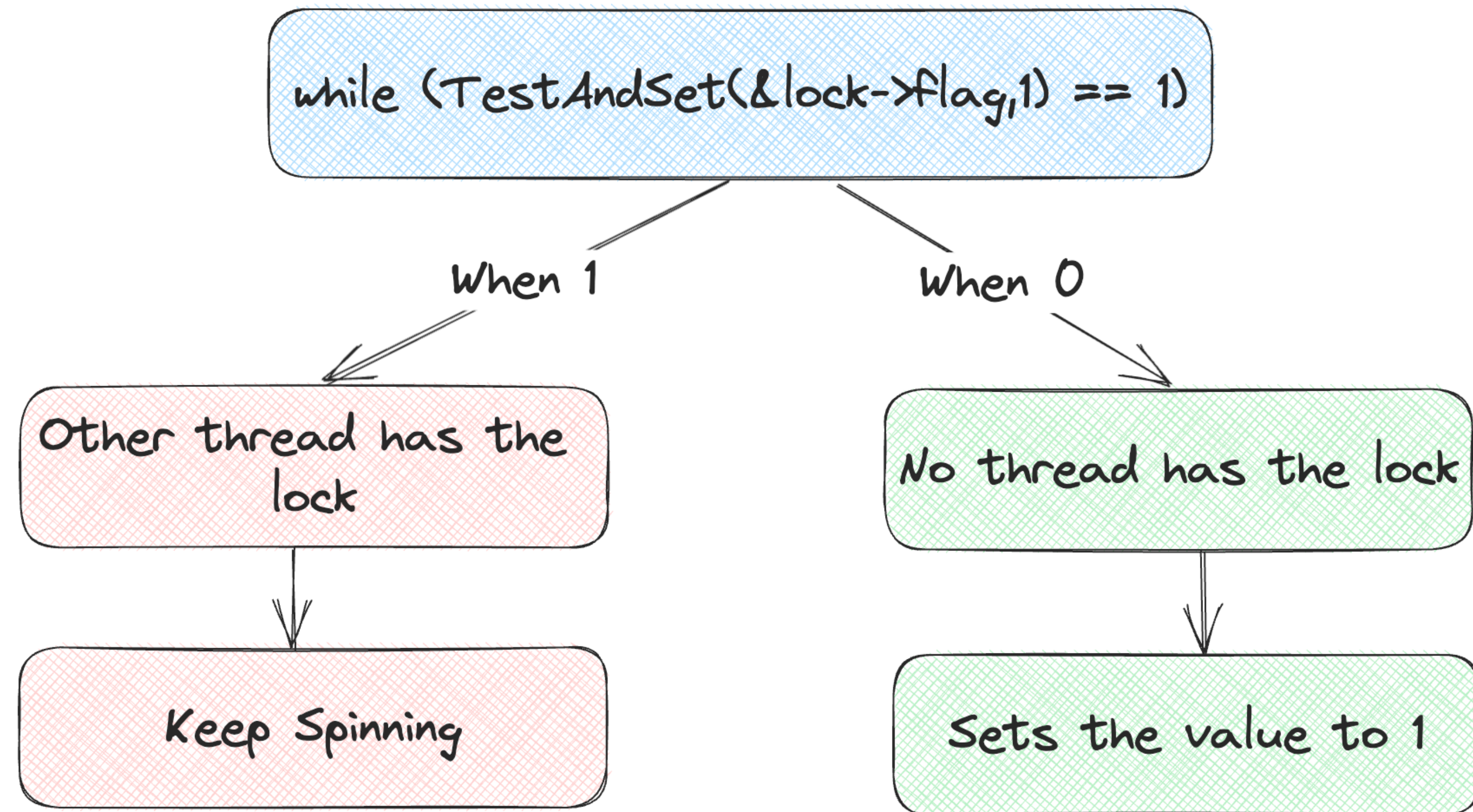
```
Using TestAndSet Lock

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)
        // Keep spinning
}

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



Only one thread can acquire the lock

Evaluating Spin Locks

- **Correctness perspective** Provides mutual exclusion, so correct!
- **Fairness perspective**
 - They don't provide fairness guarantee
 - Threads can keep spinning forever leading to starvation
- **Performance perspective**
 - In single CPU - significant overhead, What if thread holding the lock is preempted in critical section? - Run all N-1 threads to waste CPU
 - On multiple CPUs these locks work reasonably well (esp if n. Threads \sim n. CPUs)



Compare-And-Swap

Another Hardware Primitive

C Pseudocode

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

Inside the lock function the call will be:

```
while(CompareAndSwap(&lock->flag, 0,1) ==1)
```

- **Basic idea:** Test whether the value at address specified by ptr is equal to expected
 - If yes, update the memory location pointed to ptr by new value
 - If not, do nothing!
- In either case, return actual value at the memory location



Load-Linked and Store-Conditional (LL SC)

- In MIPS architecture, they can be used in tandem to build locks and concurrent structures

```
Load Linked

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

```
Store Conditional

int StoreConditional(int *ptr, int value)
{
    if (no one has updated ptr since load linked)
    {
        *ptr = value;
        return 1;
    }
    else
    {
        return 0;
    }
}
```

C Pseudocode



LL/SC for building locks

- Load linked is like a typical load operation
 - Simply fetches a value from memory and places it in a register
- Store conditional succeeds if no intermittent store to address has taken place
 - In case of success, it updates ptr to value and returns 1 else returns 0

When does failure condition of store conditional arise?

```
LL/SC for building locks

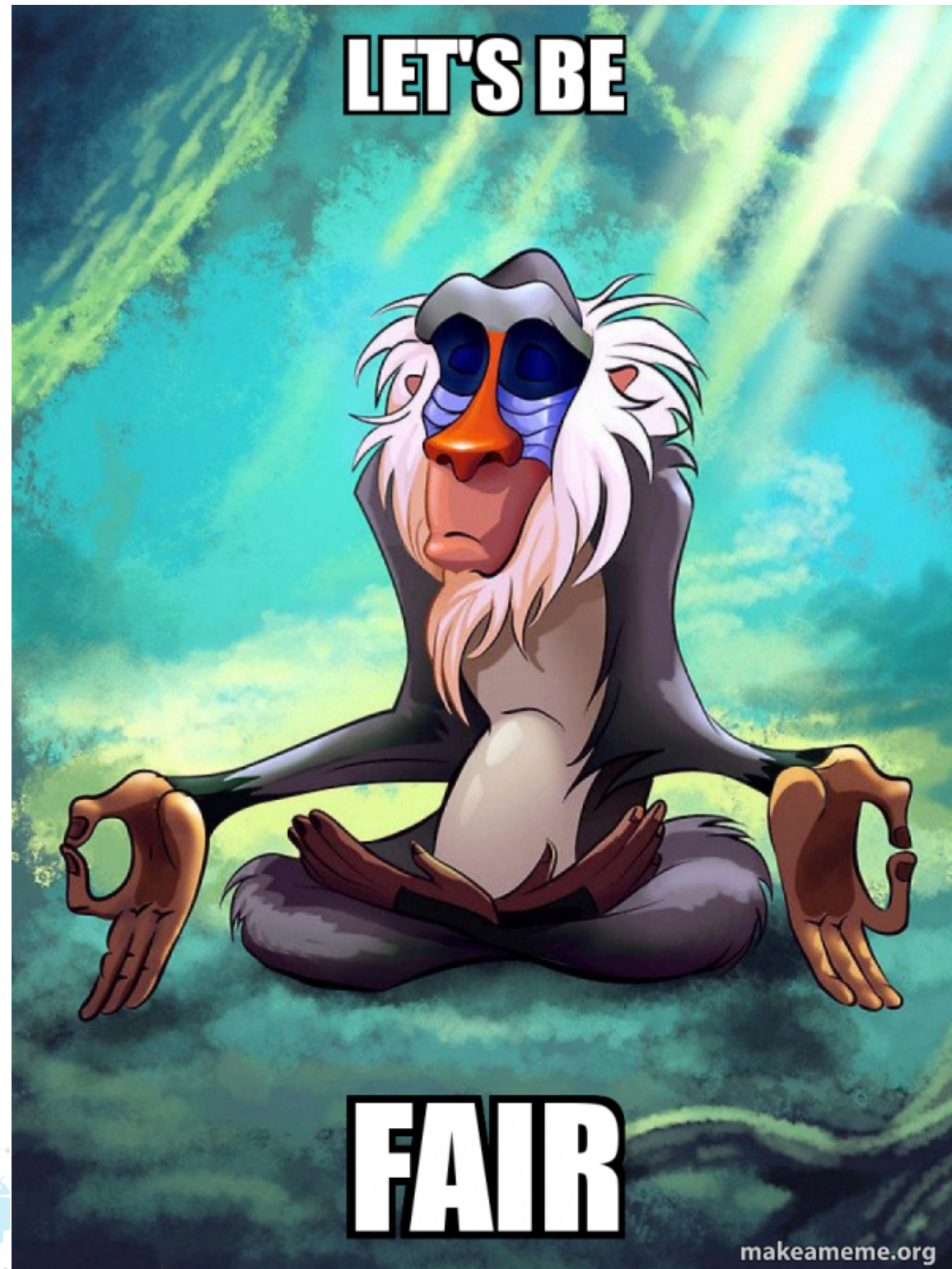
void lock(lock_t *lock)
{
    while (1)
    {
        while (LoadLinked(&lock->flag)==1)
            ; //Keep spinning

        if (StoreConditional(&lock->flag,1)==1)
        {
            //store is successful, retrun else repeat all
            again
            return;
        }
    }
}

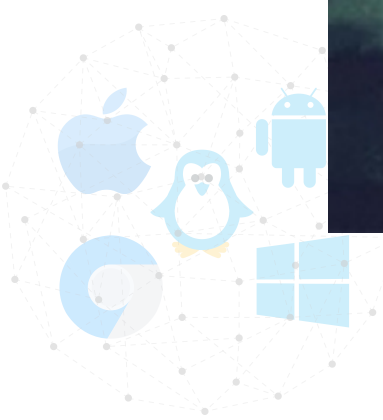
void unlock(lock_t *lock)
{
    lock->flag = 0;
}
```

What about Fairness?

Can we incorporate fairness through locks?



- Lock only has a flag variable
- Every time a thread acquires, it checks for flag
- However, which threads are checking is not recorded
- The threads that are looking for locks may have to be stored somewhere
- Can we store more information within each lock?



What if we leverage Turns and Tickets

Think about going to some crowded office space



Current Number and Window



Fetch and Add

Yet another hardware primitive but very powerful

```
Fetch And Add

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

- Atomically increment a value while returning the old value at a particular address
- Used to build interesting type of lock - **The ticket lock**
- Instead of single variable a combination of ticket and turn variable is used
- Not just flag: ticket and turn



Ticket Lock



Ticket Lock

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

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



Ticket Lock

```
void lock (lock_t *lock)
{
    int myturn = FetchAndAdd(&lock->ticket);
    //when ticket value = my turn, thread goes into CS
    while (lock->turn != myturn)
    {
        // keep spinning
    }
}

void unlock (lock_t *lock)
{
    // next waiting thread can enter CS
    FetchAndAdd(&lock->turn);
}
```



An Illustration of Ticket Lock

Four Processor Ticket Lock Example

Row	Action	next_ticket	now_serving	P1 my_ticket	P2 my_ticket	P3 my_ticket	P4 my_ticket
1	Initialized to 0	0	0	-	-	-	-
2	P1 tries to acquire lock (succeed)	1	0	0	-	-	-
3	P3 tries to acquire lock (fail + wait)	2	0	0	-	1	-
4	P2 tries to acquire lock (fail + wait)	3	0	0	2	1	-
5	P1 releases lock, P3 acquires lock	3	1	0	2	1	-
6	P3 releases lock, P2 acquires lock	3	2	0	2	1	-
7	P4 tries to acquire lock (fail + wait)	4	2	0	2	1	3
8	P2 releases lock, P4 acquires lock	4	3	0	2	1	3
9	P4 releases lock	4	4	0	2	1	3
10	...	4	4	0	2	1	3



How good is the spin based locks?

- Simple hardware based locks are simple to implement and powerful
- They are also quite inefficient especially when it comes to performance
 - Consider that there are two threads and one thread has the lock
 - When thread has lock, it may get interrupted, the other thread spins for a time slice, waste CPU cycle
 - Think about N threads, N-1 threads might waste CPU cycles in spinning (especially if round robin)
- **Can we come up with something better instead of wasting cycles with spinning?**





Thank you

Course site: karthikv1392.github.io/cs3301_osn

Email: karthik.vaidhyanathan@iiit.ac.in

Twitter: @karthi_ishere

