LECTURE 2: SUPPORT FOR CORRECTNESS IN CONCURRENCY

Intro to Concurrent Processing

- Models of correctness in concurrency.
- Deeper Dive on Race Conditions.
- Lock-based Solutions to Mutual Exclusion.
  - Benefits and challenges of Locks
  - Dekker's Algorithm.
  - Mutual Exclusion for n processes: The Bakery Algorithm.
- Higher level supports for Mutual Exclusion:
  - Semaphores & Monitors
  - Emulating Semaphores with Monitors & Vice Versa
- Solution of Classical Problems of Synchronization:
  - The Readers-Writers Problem
  - The Dining Philosophers problem in SR;
  - The Sleeping Barber Problem;

SECTION 2.0: RACE CONDITIONS & CONCURRENT CORRECTNESS

A Model of Concurrent Programming

- Concurrent code: *interleaving sets of sequential atomic instructions*.
  - i.e. interacting sequential processes run at same time, on same/different processor(s).
  - processes *interleaved* i.e. at any time each processor runs one of instructions of the sequential processes.
  - relative rate at which steps of each process execute is not important.

- Each sequential process consists of a series of *atomic instructions*.
  - *Atomic instruction* is one that, once it starts, proceeds to completion without interruption.
  - These only exist at the hardware/machine code/assembly level
  - Different processors have different atomic instructions, and this can have a big effect.
  - Compiler decides the allocation of lines of code to atomic instructions
  - O/S Scheduler decides when process runs
  - JVM Scheduler decides when java thread runs

Correctness

Consider the atomic instructions:

```
Program1:        load reg,      N
Program2:        load reg,      N
Program1 :       add reg,       #1
Program2 :       add reg,       #1
Program1 :       store reg,     N
Program2 :       store reg,     N
```

- If all math done in registers => results obtained depend on interleaving (indeterminate computation).
- This dependency on unforeseen circumstances is known as a *Race Condition*
- Can generalise to say a program is correct if: when its preconditions hold then its post conditions will hold
- For programs that are supposed to terminate we define
- **Partial Correctness**
- If the preconditions hold and the program terminates, then the postconditions will hold.
- **Total Correctness**
- If the preconditions hold, then the program will terminate and the postconditions will hold.
- A concurrent program *must be* correct under all possible *interleavings*.

Types of Correctness Properties

For programs that are not supposed to terminate, we have to write correctness criteria in terms of properties that must *always hold* (safety properties) and those that must *eventually hold* (liveness properties)

1. **Safety properties** These must *always* be true.

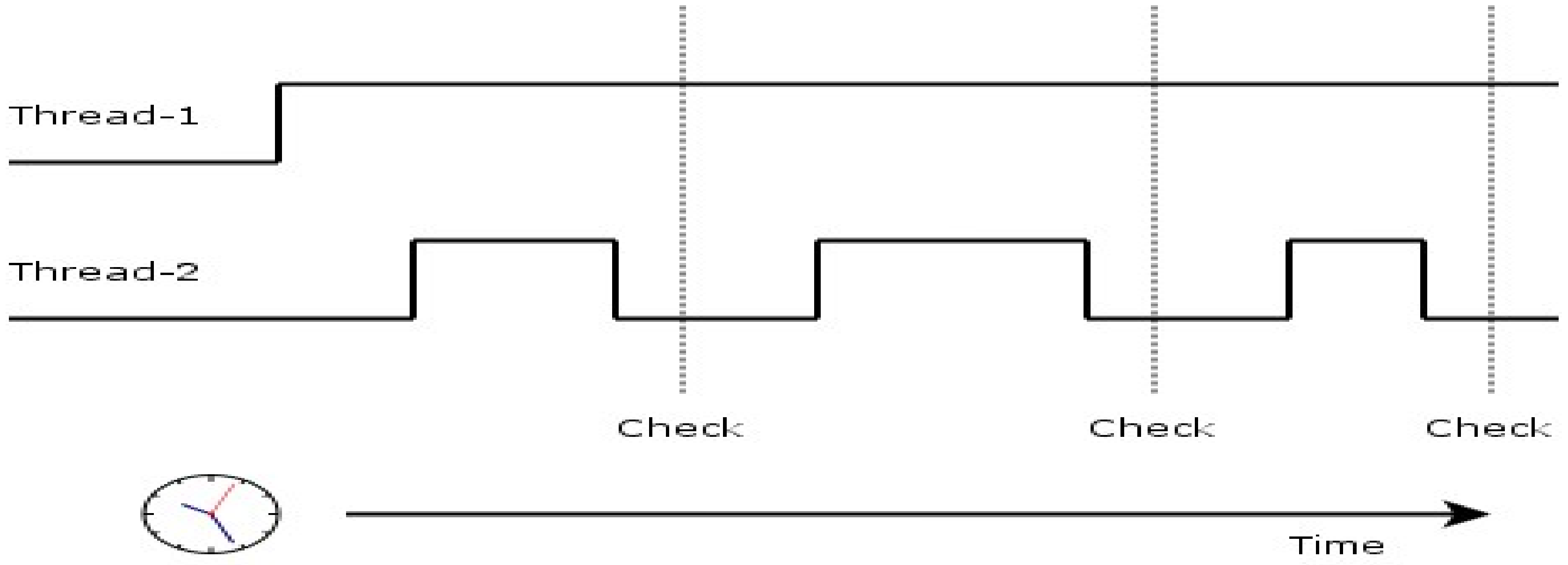
Mutual exclusion	Two processes must not interleave certain sequences of instructions.
Absence of deadlock	Deadlock is when a non-terminating system cannot respond to any signal.

2. **Liveness properties** These must *eventually* be true.

Absence of starvation	Information sent is delivered.
Fairness	That any contention must be resolved.

Correctness: Fairness

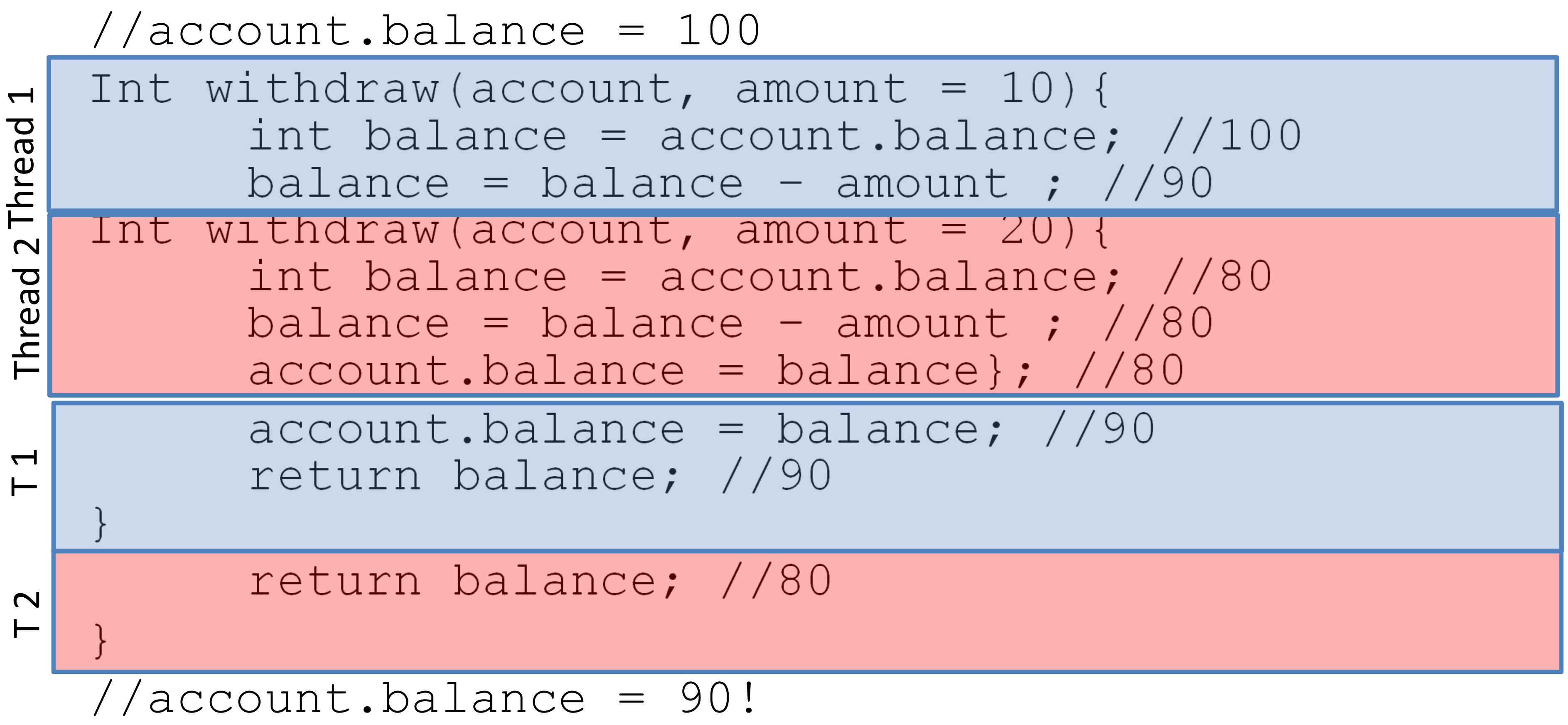
- There are 4 different way to specify *fairness*.
  - *Weak Fairness* — A process continuously requesting eventually has it granted.
  - *Strong Fairness* — A process requesting infinitely often, eventually will have it granted.



Lets Look at this in Practice: Race Conditions

- A **race condition** occurs when program output is dependent on the sequence or timing of code execution
- You must work to avoid this with concurrent code.
- E.g. if multiple threads of execution enter a critical section at about the same time; both attempt to update the shared data structure
  => leads to surprising results (undesirable)
- **critical section** = parts of the program where a shared resource is accessed need to be protected in ways that avoid the concurrent access

Eg Bank Transaction



```
//account.balance = 100
Int withdraw(account, amount = 10){
     int balance = account.balance; //100
     balance = balance - amount ; //90
Int withdraw(account, amount = 20){
     int balance = account.balance; //80
     balance = balance - amount ; //80
     account.balance = balance}; //80
     account.balance = balance; //90
     return balance; //90
}
     return balance; //80
}
//account.balance = 90!
```

Race Condition Consequences

- We can get different results every time we run the code
- => result is **indeterminate**
- Deterministic computation has the same result each time
  - Much easier to understand/debug
  - We want deterministic concurrent code

  => use synchronisation mechanisms

Handling Race Conditions

- Need mechanism to control access to shared resources for concurrent code

=> Synchronisation is necessary for any shared data structure

- Focus on critical sections of code i.e. bits that access shared resources
- Want critical sections to run with mutual exclusion

=> only one thread can execute that code at the same time

Example: Bank Transactions

What code should be within the critical section?

```
int withdraw(account, amount){
    int balance = account.balance;
    balance = balance - amount ;
    account.balance = balance};
    return balance;
}
```

Critical section

Q: Why is this not critical?

Recall: How is storage allocated

- Stack
  - One per thread
  - Contains
    - Function parameters
    - **Local variables**
- Heap
  - One per process
  - Contains
    - Dynamically allocated types, variables, objects
- For more see https://www.codeproject.com/Articles/76153/Six-important-NET-concepts-Stack-heap-value-types

Stack vs Heap Memory for Threads

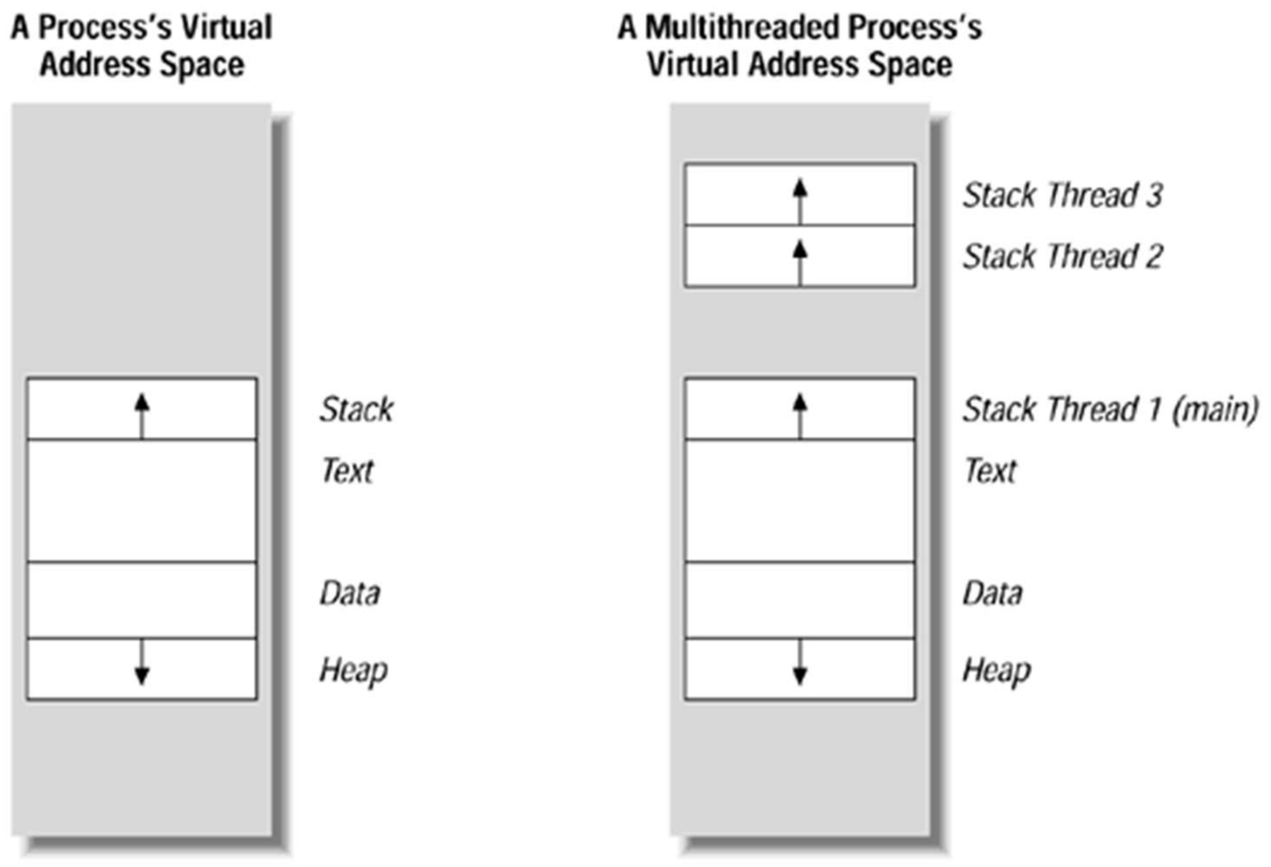

A Process's Virtual Address Space
Stack
Text
Data
Heap
A Multithreaded Process's Virtual Address Space
Stack Thread 3
Stack Thread 2
Stack Thread 1 (main)
Text
Data
Heap

Figure from *PThreads Programming* by Nichols et al. (1996)

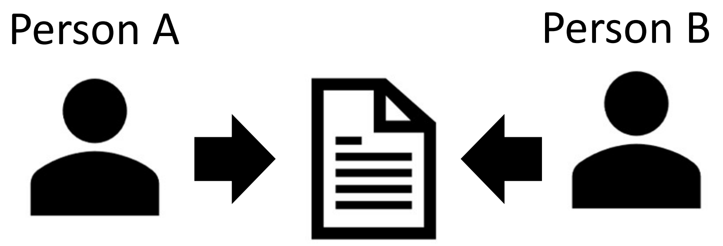
Critical Section Properties

- **Mutual exclusion**: only 1 thread at a time
- **Guarantee of progress**: threads outside the critical section cannot stop another from entering it
- **Bounded waiting**: a thread waiting to enter section will eventually enter
  - Threads in the critical section will eventually leave
- **Performance**: the overhead of entering/exiting should be small
  - Especially compared to amount of work done in there – why?
- **Fair**: don't make some threads wait much longer than others

Synchronisation Solutions

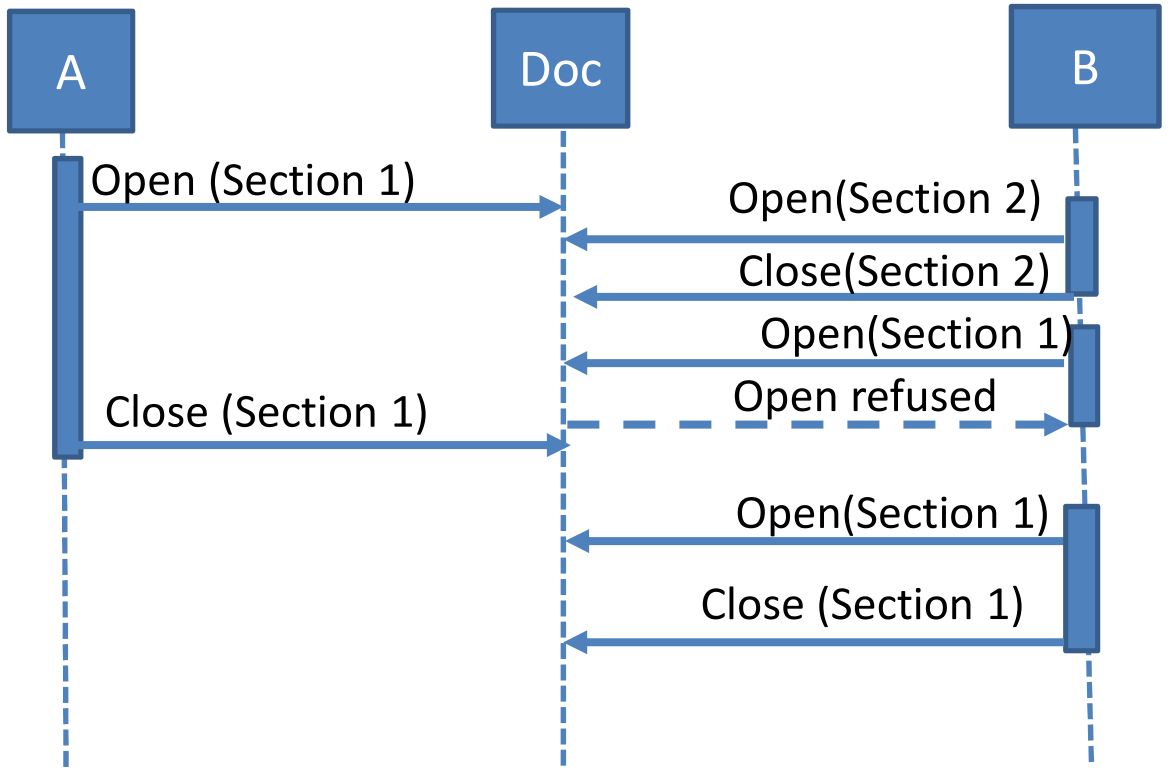
- Option 1: Atomicity
  - Making sure that we have mutual exclusion
- Option 2: Conditional synchronisation (ordering)
  - Making sure that one thread runs before another

Real World Concurrency Problem: Writing a Document Together



- 2 actors, 1 shared resource => concurrency
- **Solution 1**: Atomicity
  - Strategy: Ensure that they do not edit the same section at the same time
    - While editing you cannot be interrupted by the other person also editing that section
    - If you read a section and modify it, there will be no edits from the other person
    - i.e. Read and write the same version of the paper

What Might Atomicity Look Like?

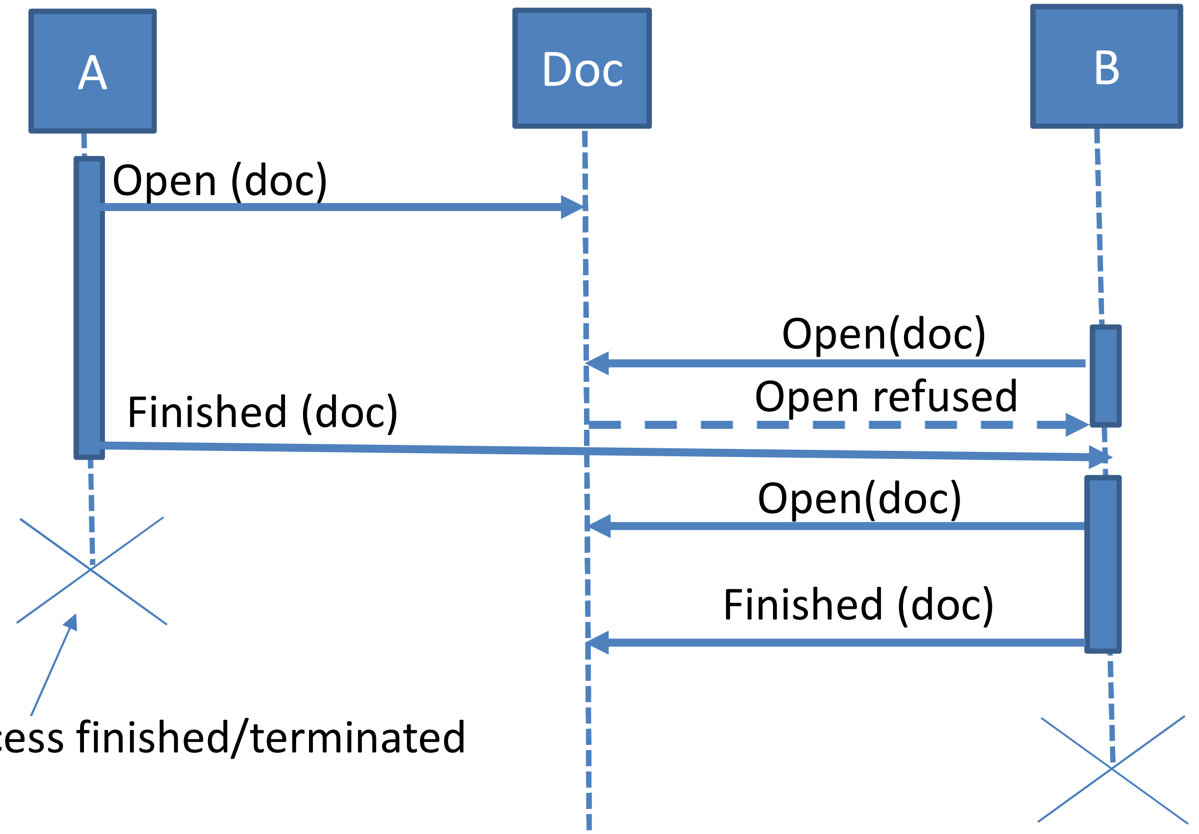


Exercise: Re-draw this with a scheduler

Writing a Document Together 2

- **Solution 2**: Conditional Synchronisation
  - Strategy: Person A writes a rough draft and then Person B edits it.
    - A and B cannot write at the same time (as they are working on different versions of the paper)
    - But must ensure that Person B cannot start until Person A is finished

What Might Conditional Synchronisation Look Like?



Exercise: Re-draw this with a scheduler

Code Constructs to Support Defining Critical Sections

- Locks
  - Very primitive, just provide mutual exclusion, minimal semantics, useful as a building block for other methods
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - Higher level abstraction, requires language support, implicit operations
  - Provide both atomicity and conditional synchronisation
  - Easy to program with: Java "synchronised()" as example
- Message Passing
  - Simple model of communication and synchronisation based on atomic transfer of data across a channel
  - Applied in distributed systems

SECTION 2.1: MUTUAL EXCLUSION SOLUTIONS: LOCKS

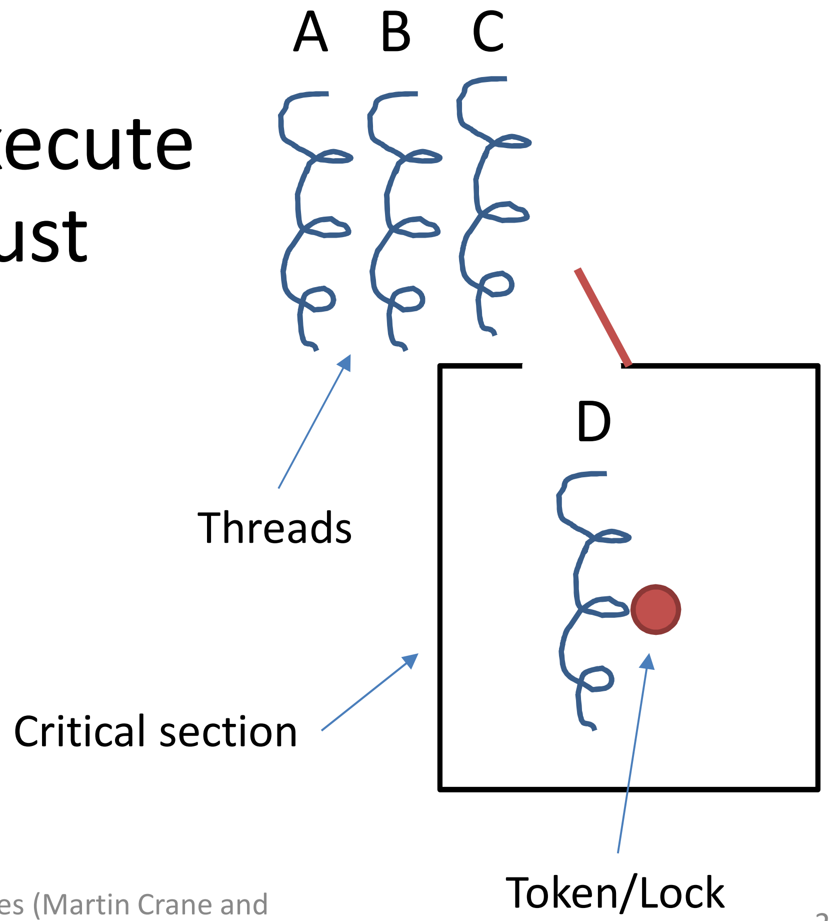
Mutual Exclusion (ME) with Locks

- From above, concurrent code must be correct in all allowable interleavings.
- So some (ME) parts of different processes *cannot* be interleaved as they lead to race conditions
- These are called *critical sections*.
- Try solving ME issue with locks before advanced solutions
- **A lock is designed to enforce a mutual exclusion concurrency control policy (i.e. a synchronization protocol).**

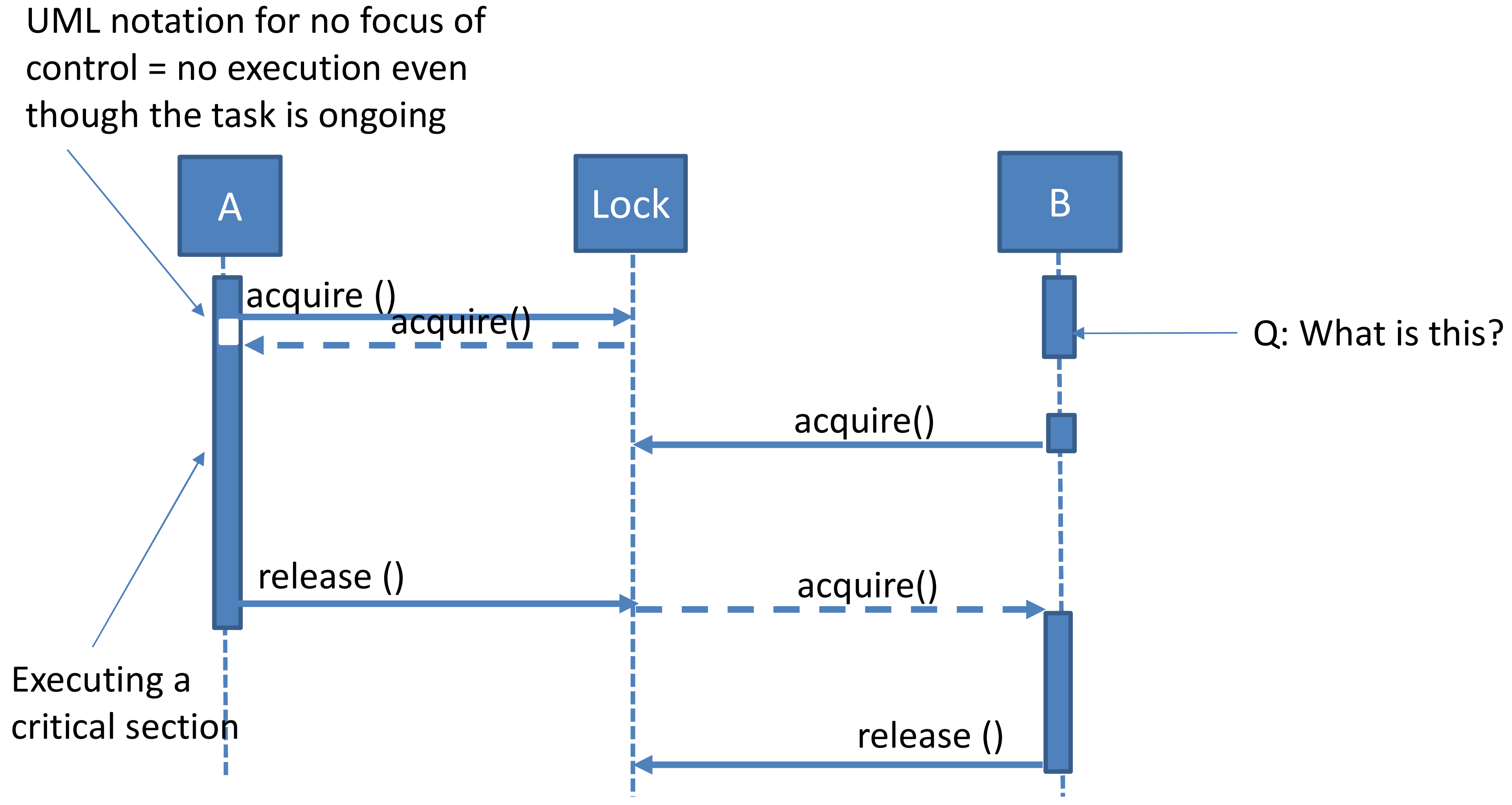
```
// Pseudo Code showing a critical section shared by
le (true)
          // different processes
   whi// Non_Critical_Section
          // Pre_protocol
          // Critical_Section
          // Post_protocol
   end while
```

Locks: Basic idea

- Lock = a token you need to enter a critical section of code
- If a thread wants to execute a critical section…it must have the lock:
  - Need to ask for lock
  - Need to release lock
- No restrictions on executing other code



Threads Executing with Locks


UML notation for no focus of control = no execution even though the task is ongoing
A
Lock
B
acquire ()
acquire()
Q: What is this?
acquire()
release ()
acquire()
Executing a critical section
release ()

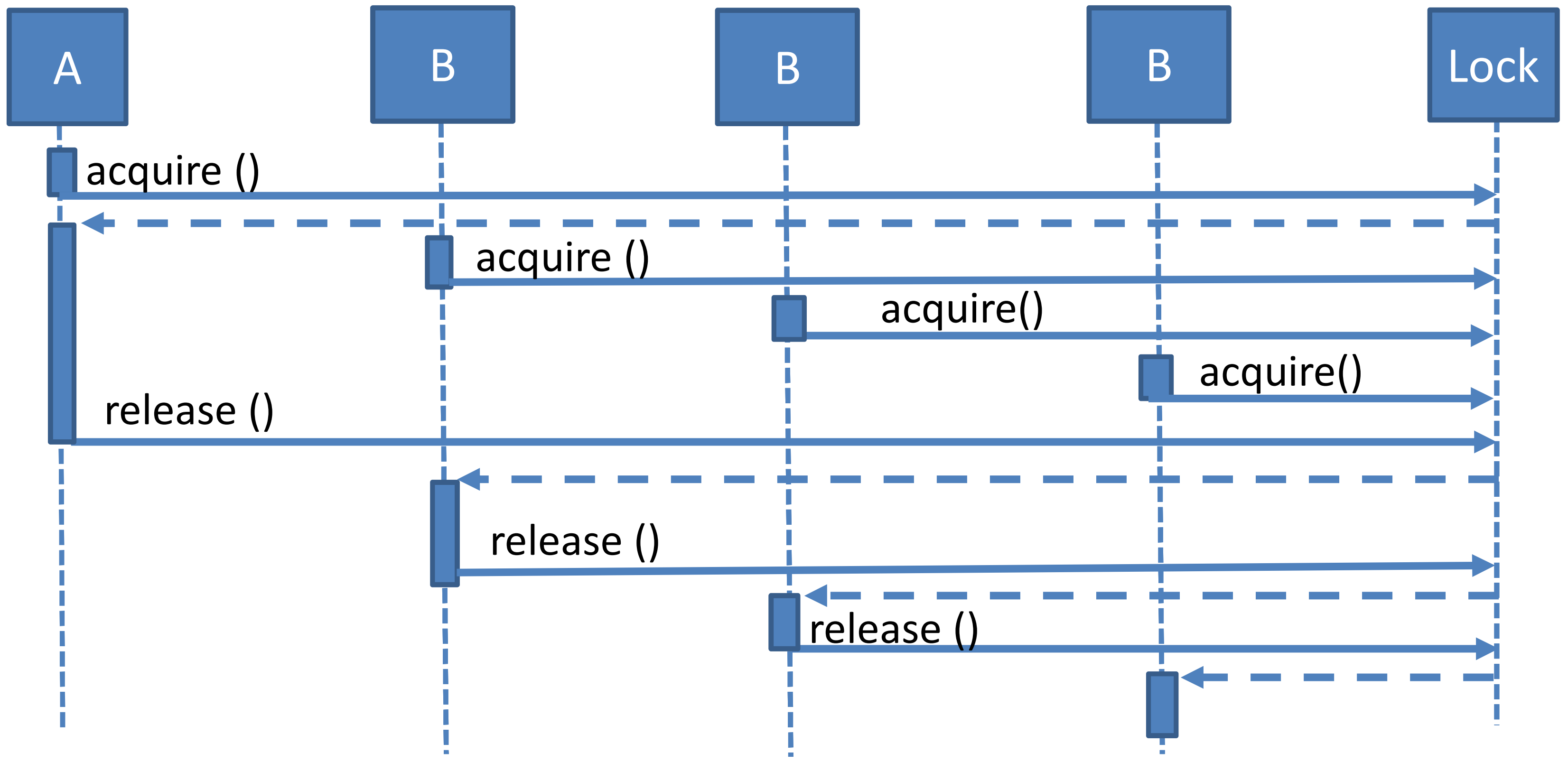
Q: How many CPUs are there on this machine?

Lock States and Operation

- Locks have 2 states:
  - Held: some thread is in the critical section
  - Not held: no thread is in the critical section
- Locks have 2 operations:
  - Acquire – mark lock as held or wait until released
    - If not held => execute immediately
  - Release – mark as not held

=> If many threads call acquire, only 1 thread can get the lock

Many Threads Want the Lock



Q: Any problems here? Is there anything we have not specified?

Using Lock

- Locks are declared like variables:
  `Lock myLock;`
- A program can use multiple locks – why?
  `Lock myDataLock, myIoLock;`
- To use a lock:
  - Surround critical section as follows:
    - Call **acquire**() at start of critical section
    - Call **release**() at end of critical section
- Remember our general pattern for mutex:

```
while (true)
   // Non_Critical_Section
        // Pre_protocol - for locks:
        myLock.acquire();
        // Critical_Section
        // Post_protocol - for locks
         myLock.release();
   end while
```

i.e. Surround critical section of code

Lock Benefits

- Only 1 thread can execute the critical section code at a time
- When a thread is done (and calls release) other threads can enter the critical section

=> Achieves requirements of **mutual exclusion** and **progress** for concurrent systems

Lock Limitations

- Acquiring a lock *only* blocks threads trying to acquire the *same* lock
  - i.e. threads can acquire other locks
  - $\Rightarrow$Can have different threads in different critical sections at the same time
  - $\Rightarrow$What if we have to stop all threads?
- **Must use same lock for all critical sections accessing the same data** (or resource)
  - E.g. withdraw() and deposit() for a bank account
- Q: What does this mean for code complexity?
  - Eg Add a new feature that accesses same data
  - Eg Critical sections start to overlap in the data accessed

Lock in Use Example: Bank Transactions

```
See our old code:
int withdraw(account, amount){
    acquire(myBalanceLock);
    int balance = account.balance;
    balance = balance - amount ;
    account.balance = balance};
    release(myBalanceLock);
    return balance;
}
```

Critical section

Local variable, does not need to be protected

Eg Bank Transaction with Locks

```
//account.balance = 100
```

Thread 1

```
Int withdraw(account, amount = 10){
       acquire(myBalanceLock);
       int balance = account.balance; //100
```

T 2

```
Int withdraw(account, amount = 20){
       acquire(myBalanceLock);      // THREAD STALLED
```

T 1

```
balance = balance - amount ; //90
       account.balance = balance; //90
       release(myBalanceLock);     // NOW T2 can start
```

T 2

```
       int balance = account.balance; //90
       balance = balance - amount ; //70
       account.balance = balance}; //70
       release(myBalanceLock);
       return balance; //70
}
```

T 1

```
       return balance; //90
}
```

```
//account.balance = 70
```

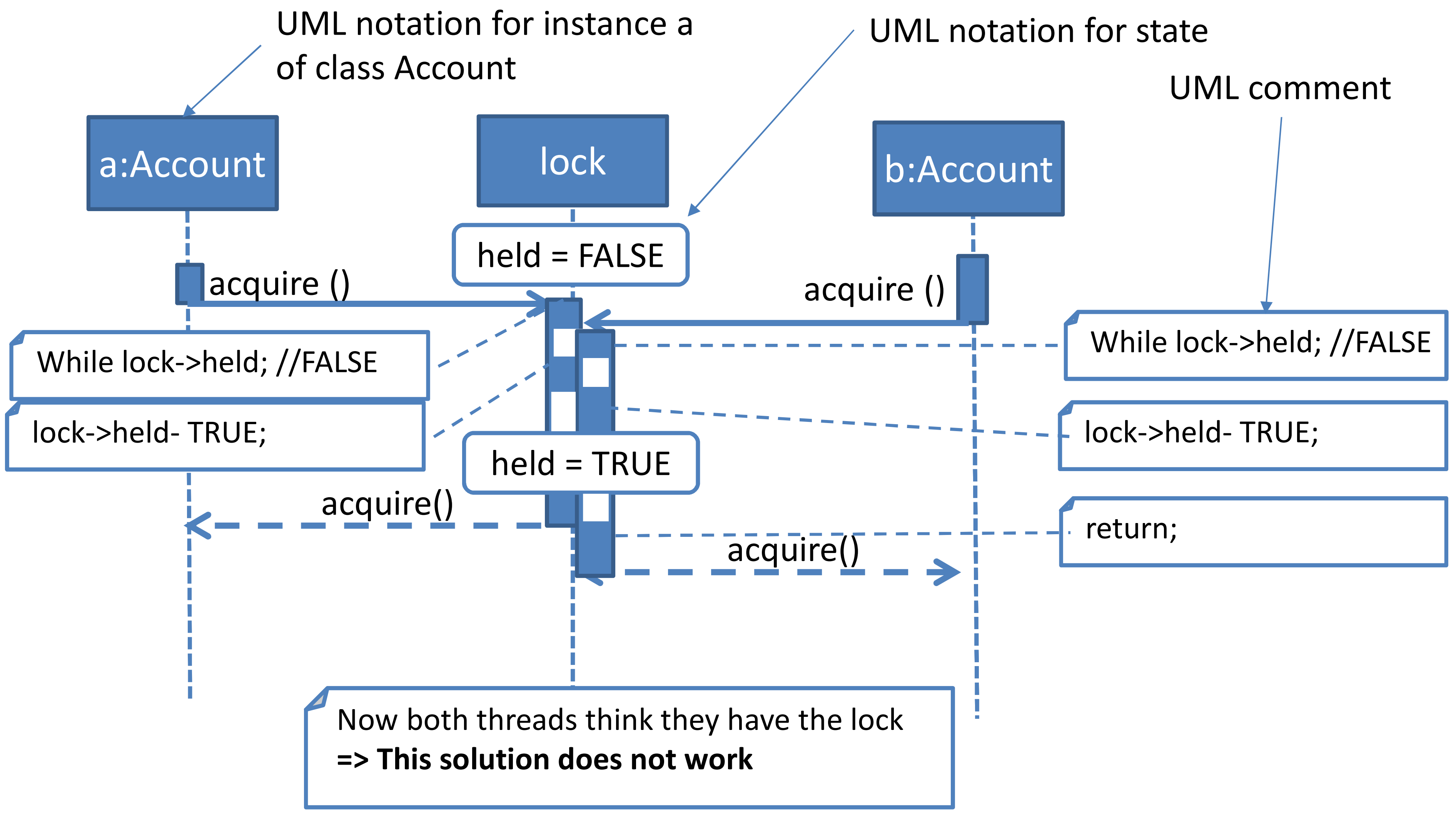
Impacts

- We can run the threads in any order and have the correct final balance
- We no longer have a race condition

Implementing Locks

```
Struct lock {
     bool held; //initially FALSE
}
void acquire(lock) {
     while(lock->held)
          ; //just wait
     lock->held = TRUE;
}
void release(lock) {
     lock->held = FALSE;
}
```

How does it run?


UML notation for instance a of class Account
UML notation for state
UML comment
a:Account
lock
b:Account
held = FALSE
acquire ()
acquire ()
While lock->held; //FALSE
While lock->held; //FALSE
lock->held- TRUE;
lock->held- TRUE;
held = TRUE
acquire()
return;
acquire()
Now both threads think they have the lock
=> This solution does not work

Solve via Hardware Support

```
//c code for test and set behaviour
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
}
```

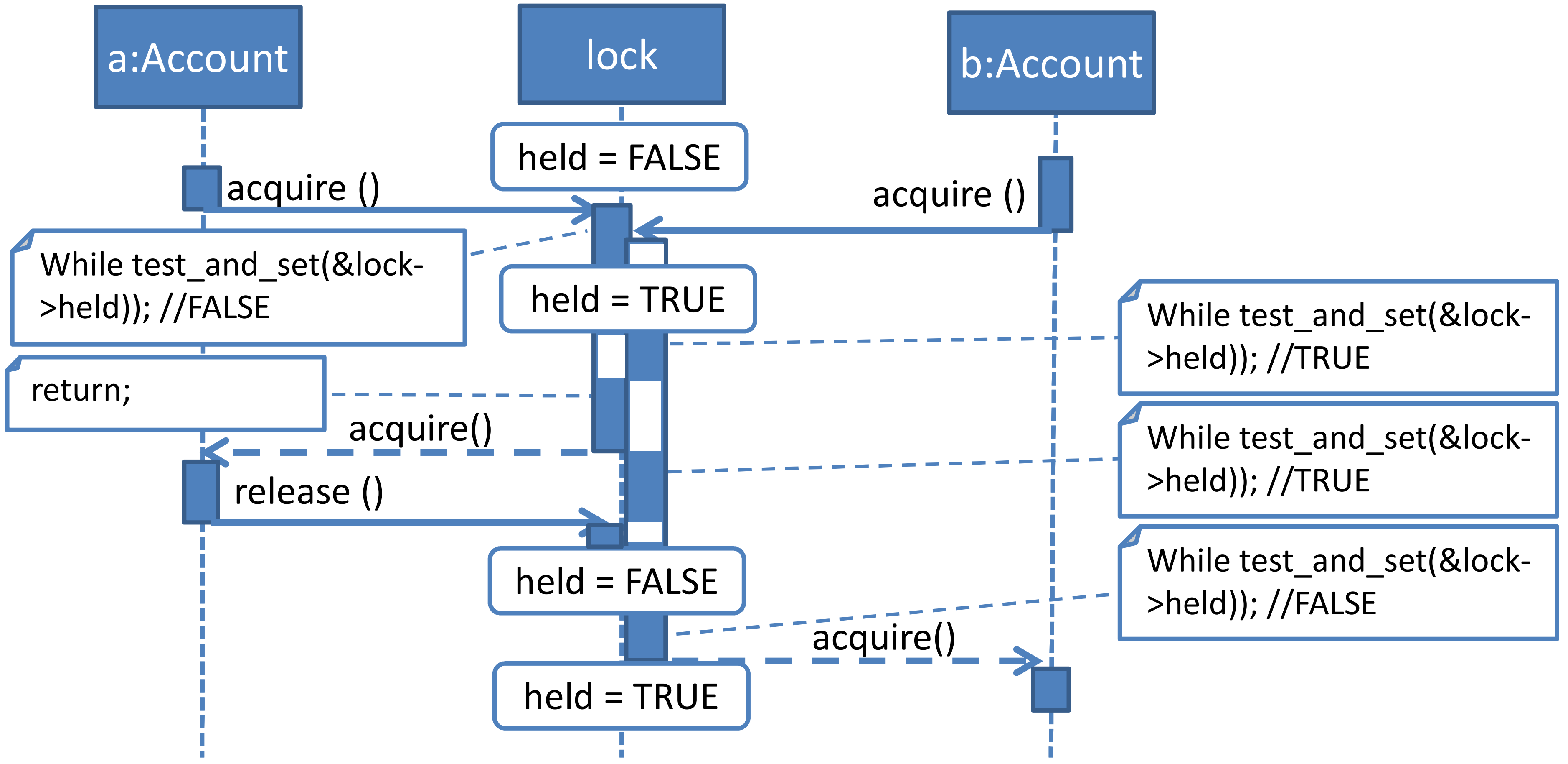
Processor has a special instruction called “test and set”

- Allows atomic read and update

Hardware-based Spinlock

```
struct lock {
     bool held; //initially FALSE
}
void acquire(lock) {
     while(test_and_set(&lock->held))
          ; //just wait
     return;
     }
void release(lock) {
     lock->held = FALSE;
}
```

How does it run?


a:Account
lock
b:Account
held = FALSE
acquire ()
acquire ()
While test_and_set(&lock->held)); //FALSE
held = TRUE
While test_and_set(&lock->held)); //TRUE
return;
acquire()
While test_and_set(&lock->held)); //TRUE
release ()
held = FALSE
While test_and_set(&lock->held)); //FALSE
acquire()
held = TRUE

Q: Why is this called a spin lock?

Do Locks give us sufficient safety?

1. **Recall Safety properties** (Correctness)
   These must *always* be true.

Mutual exclusion	Two processes must not interleave certain sequences of instructions.

Q: What do you think?

Absence of deadlock	A non-terminating system cannot respond to any signal.

Q: What do you think?

Do Locks give us sufficient Liveness?

1. **Recall Liveness properties** (Correctness)
   These must ***eventually*** be true.

Absence of starvation	Information sent is delivered.

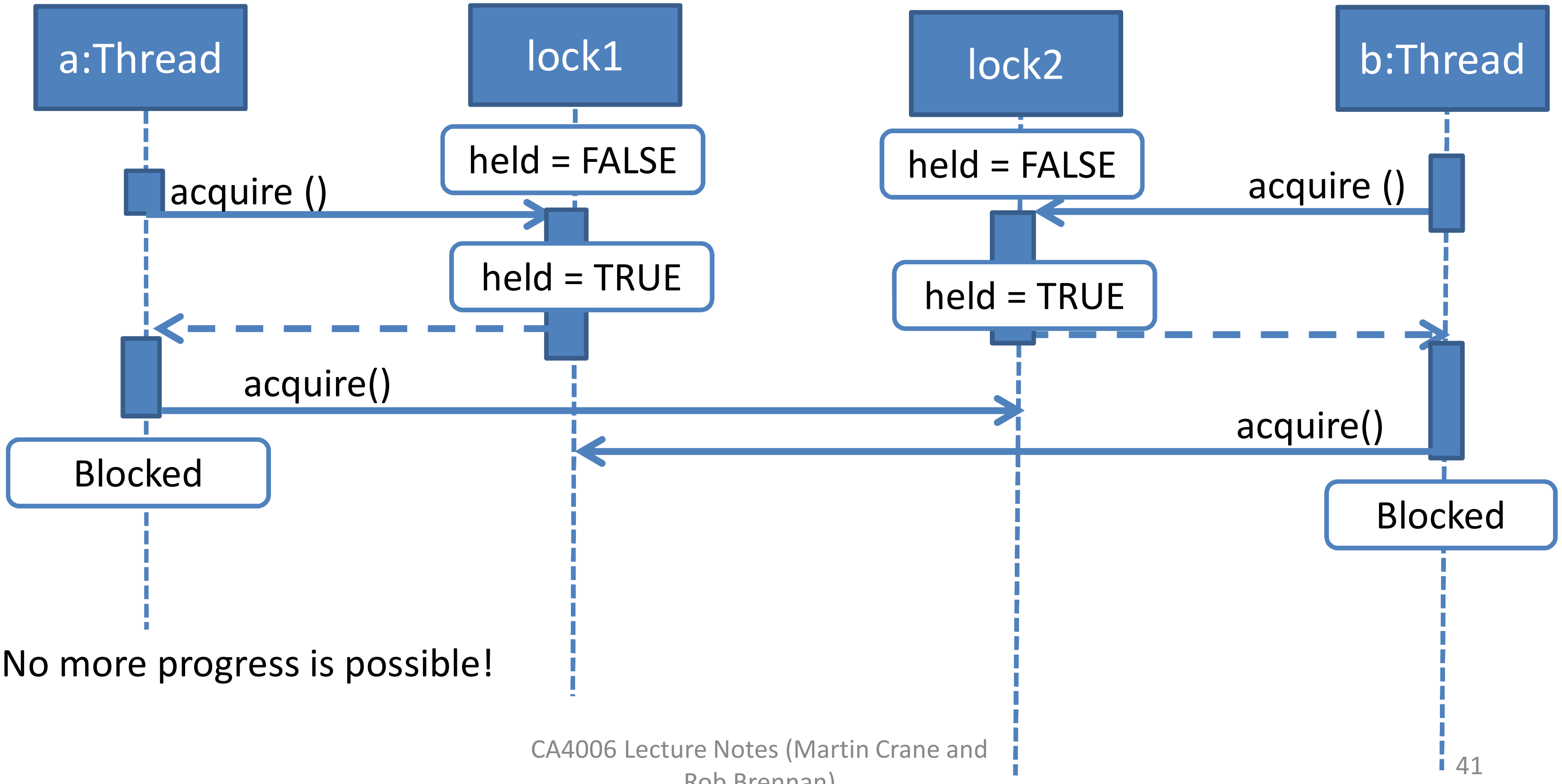
   Q: What do you think?

Fairness	That any contention must be resolved.

   Q: What do you think?

Lock Deadlock Scenario

- 2+ threads, 2 shared resources, 2 locks



No more progress is possible!

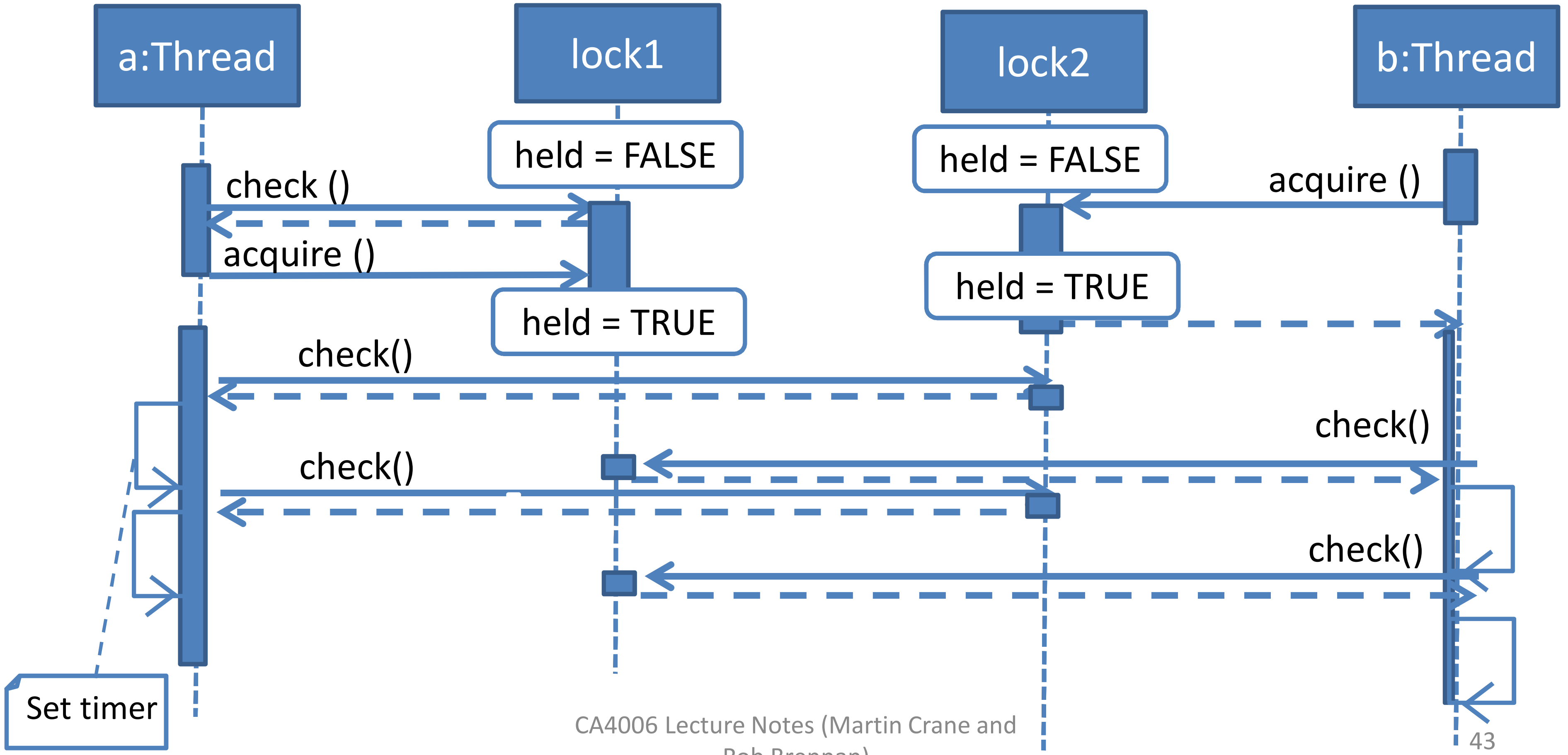
Protocols to avoid deadlock

- Add a timer to lock.request() method
- Add a new lock.check() method to see if a lock is already held before requesting it

  => you can wait or do something else and come back and check again
- Hold and wait protocol – never hold onto 1 resource when you need 2!
- **But these all lead to problems too!**

Livelock by trying to avoid deadlock

- 2 threads, 2 resources, locks with checking



Starvation

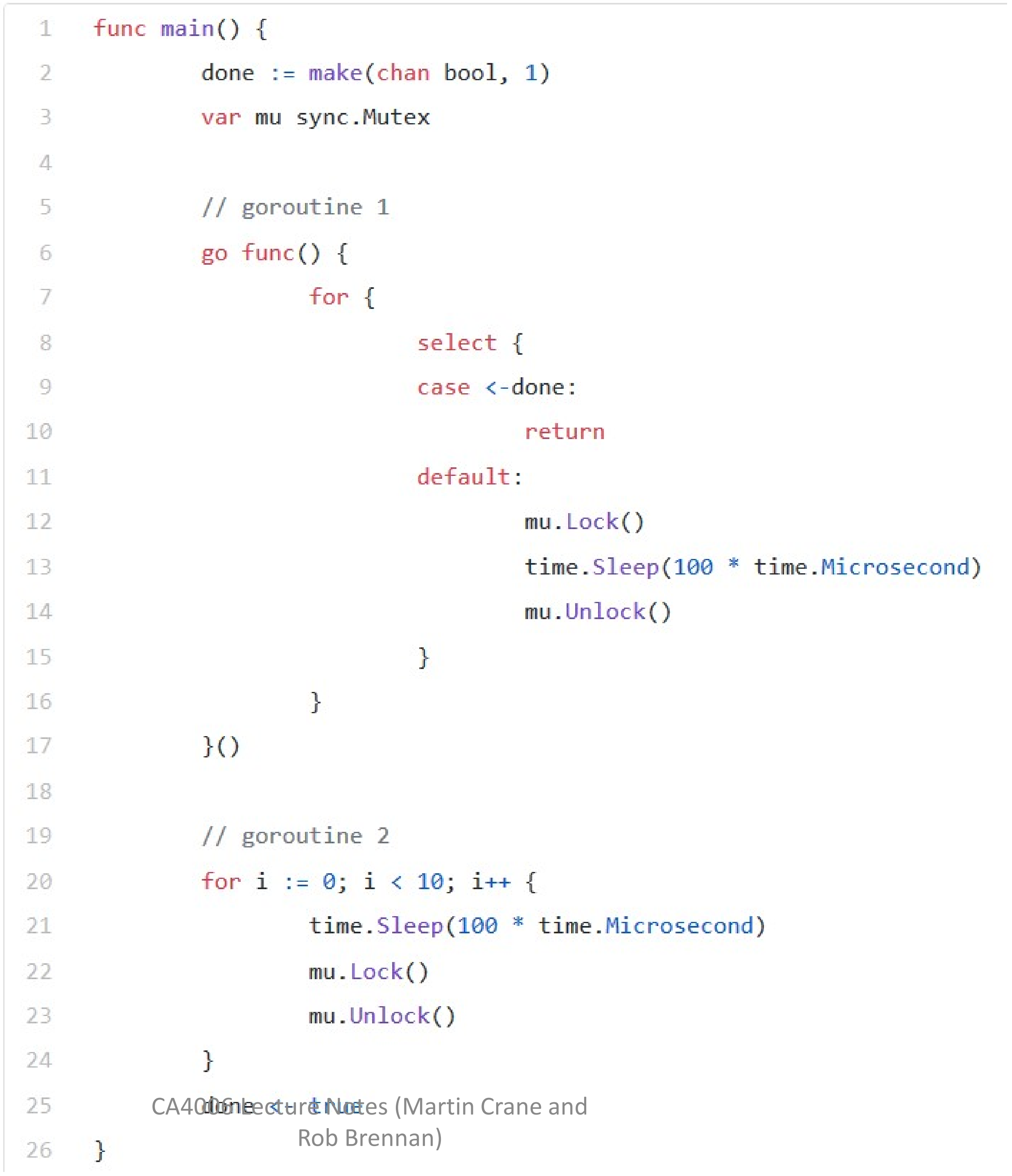
- More general case of livelock
- 1+ threads do not get to run as another thread is locking the resource
- Example:
  - 2 threads
    - **Thread A** runs for 99ms, releases lock for 1ms
    - **Thread B** runs for 1ms, releases lock for 90ms

  $\Rightarrow$A sends many more requests for resource

  $\Rightarrow$B hardly ever gets allocated the resource

Go Code Starvation Example

Example run:
Lock acquired per goroutine:
g1: 7200216
g2: 10



```go
func main() {
        done := make(chan bool, 1)
        var mu sync.Mutex

        // goroutine 1
        go func() {
                for {
                        select {
                        case <-done:
                                return
                        default:
                                mu.Lock()
                                time.Sleep(100 * time.Microsecond)
                                mu.Unlock()
                        }
                }
        }()

        // goroutine 2
        for i := 0; i < 10; i++ {
                time.Sleep(100 * time.Microsecond)
                mu.Lock()
                mu.Unlock()
        }
        done <- true
}
```

https://medium.com/a-journey-with-go/go-mutex-and-starvation-3f4f4e75ad50

Locks/Critical Sections and Reliability

- What if a thread is interrupted, is suspended, or crashes inside its critical section?
- In the middle of the critical section, the system may be in an inconsistent state
- Not only that, the thread is holding a lock and if it dies no other thread waiting on that lock can proceed!
- Critical sections have to be treated as transactions and must always be allowed to finish.

- **Developers must ensure critical regions are very short and always terminate.**

Fairness Solutions

- **Enforced fairness**: every thread gets a fixed time slice (naive)
- **Thread priority**: threads have priority allocated to enable variable execution schemes
- **Barging.** Designed to improve throughput. When the lock is released, it will wake up the first waiter and give the lock to either the first incoming request or this awoken waiter.
- **Handoff.** When released, the mutex will hold the lock until the first waiter is ready to get it. It reduces the throughput here since the lock is held even if another thread would be ready to acquire it

Drawbacks

- Spinlocks are a form of busy waiting

  => burn CPU time
- Once acquired they are held until explicitly released
  - What about other threads?
- Inefficient if lock is held for long periods
  - Short time OK as avoids O/S overhead of context switching
  - If O/S scheduler sleeps thread while lock is held

    $\Rightarrow$All other threads use their CPU time to spin while thread with the lock makes no progress
- Without hardware support, accessing the lock causes race conditions

Beyond Locks: Semaphores

- Locks only provide mutual exclusion
  - Ensure only 1 thread is in the critical section at a time
  - Good for protecting our shared resource eg bank balance to prevent race conditions and avoid nondeterministic execution
- We want more!
  - What about fairness, avoiding starvation, and livelock?

  => Need to be able to place an ordering on the scheduling of threads

Locks: Key Message

- Do not directly use Locks any more
- Use patterns with higher level of abstraction to ensure you can deal with all correctness concerns, not just mutex
- Race conditions, deadlock, livelock, fairness, reliability are all concerns when writing concurrent code

References

- UML Sequence Diagram notation, https://www.uml-diagrams.org/sequence-diagrams.html
- Mike Swift Concurrency videos https://www.youtube.com/channel/UCBRYU9uye8e-ZuWQMPBAoYA