Concurrent and Distributed Programming (CA4006)

Concurrent Correctness

2022/2023

Graham Healy

These course slides are partly adapted from the original course slides prepared by:
Dr Martin Crane, Dr Rob Brennan and Dr Takfarinas Saber

Topics for Concurrent Processing

- Models of correctness in concurrency
- Deeper Look at Race Conditions
- Lock-based Solutions to Mutual Exclusion
- Higher level supports for Mutual Exclusion
  - Semaphores, Monitors, ...
- Solution of Classical Problems of Synchronization
  - The Readers-Writers Problem
  - The Dining Philosophers problem
  - The Sleeping Barber Problem

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/speed 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

Correctness

For example:

- If all the math is done in registers, then the results depend on interleaving (indeterminate computation).

- This dependency on unforeseen circumstances is known as a **Race Condition**.

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

A concurrent program must be correct under all possible interleavings.

Types of Correctness Properties

Programs 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.

Programs not supposed to terminate

We write correctness criteria in terms of:

- Safety properties: 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.
- Liveness properties: must eventually be true:
  - **Absence of starvation**: information sent is delivered, perpetually denied resources, ...
  - **Fairness**: any contention must be resolved.

Correctness: Fairness

There are different ways to specify fairness:

- **Strong Fairness**: every process that is enabled infinitely often, should be executed infinitely often in a state where it is enabled
- **Weak Fairness**: every process that is almost always enabled, should be executed infinitely often
- **FIFO**: A process requesting is granted it before another one making a later request
- **…**

Let's Look at this in Practice: Race Conditions

- A race condition occurs when program output is dependent on the sequence or timing of code execution
  - 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)
  - We must work to avoid this with concurrent code

- **Critical section** = parts of the program where a shared resource is accessed
  - It needs to be protected in ways that avoid the concurrent access

Example Bank Transaction

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

Example Bank Transaction

Two threads:

Thread 1: withdraw 10 from account

Thread 2: withdraw 20 from account

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

Thread 1

```
Int withdraw(account, amount = 10){
    int balance = account.balance; //100
    balance = balance - amount ; //90
```

Thread 2

```
Int withdraw(account, amount = 20){
    int balance = account.balance; //80
    balance = balance - amount ; //80
    account.balance = balance; //80
```

Thread 1

```
    account.balance = balance; //90
    return balance; //90
}
```

Thread 2

```
    return balance; //80
}
```

```
//account.balance = 90!
```

Race Condition Consequences

We can get different results every time we run the code

- result is **indeterminate**

Deterministic computations have the same result each time

- Much easier to understand/debug
- We want deterministic concurrent code
- To do this... we can use **synchronisation mechanisms**

Handling Race Conditions

- We need a mechanism to control access to shared resources in concurrent code
  - Synchronisation is necessary for any shared data structure

- Idea:
  - Focus on critical sections of code
    - i.e., bits that access shared resources
  - We want critical sections to run with mutual exclusion
    - only one thread can execute that code at the same time

Example: Bank Transactions

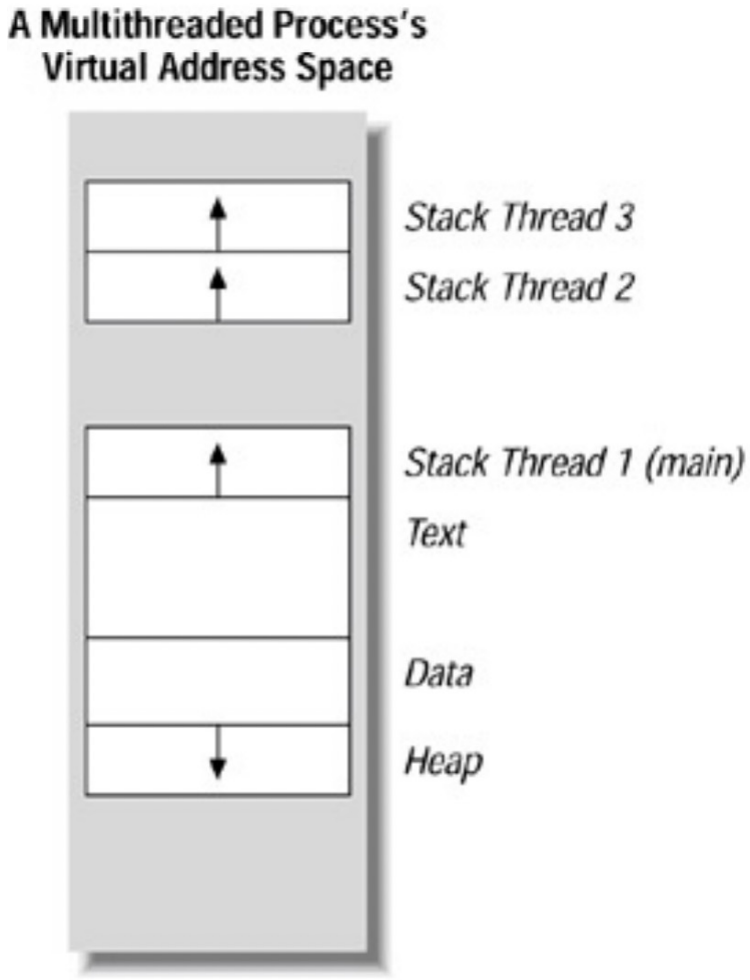
What code should be within the critical section?

```
1 int withdraw(account, amount){
2  int balance = account.balance;
3  balance = balance – amount ;
4  account.balance = balance};
5  return balance;
6}
```

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
  - Globally accessible
  - Contains Dynamically allocated types, variables, objects

Q: What types of resources are dangerous for concurrent code?

Critical Section Properties

- **Mutual exclusion**: only 1 thread can access 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
  - AND 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

Ways to protect critical sections:

Option 1: Atomicity

- Making sure that we have mutual exclusion

Option 2: Conditional synchronisation (ordering)

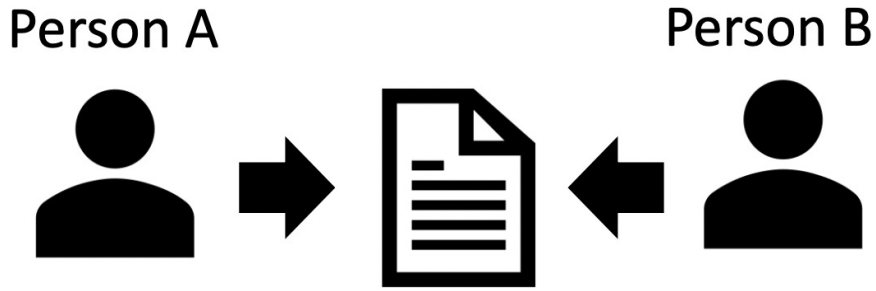
- E.g., making sure that one thread runs before another

Real World Concurrency Problem: Writing a Document Together

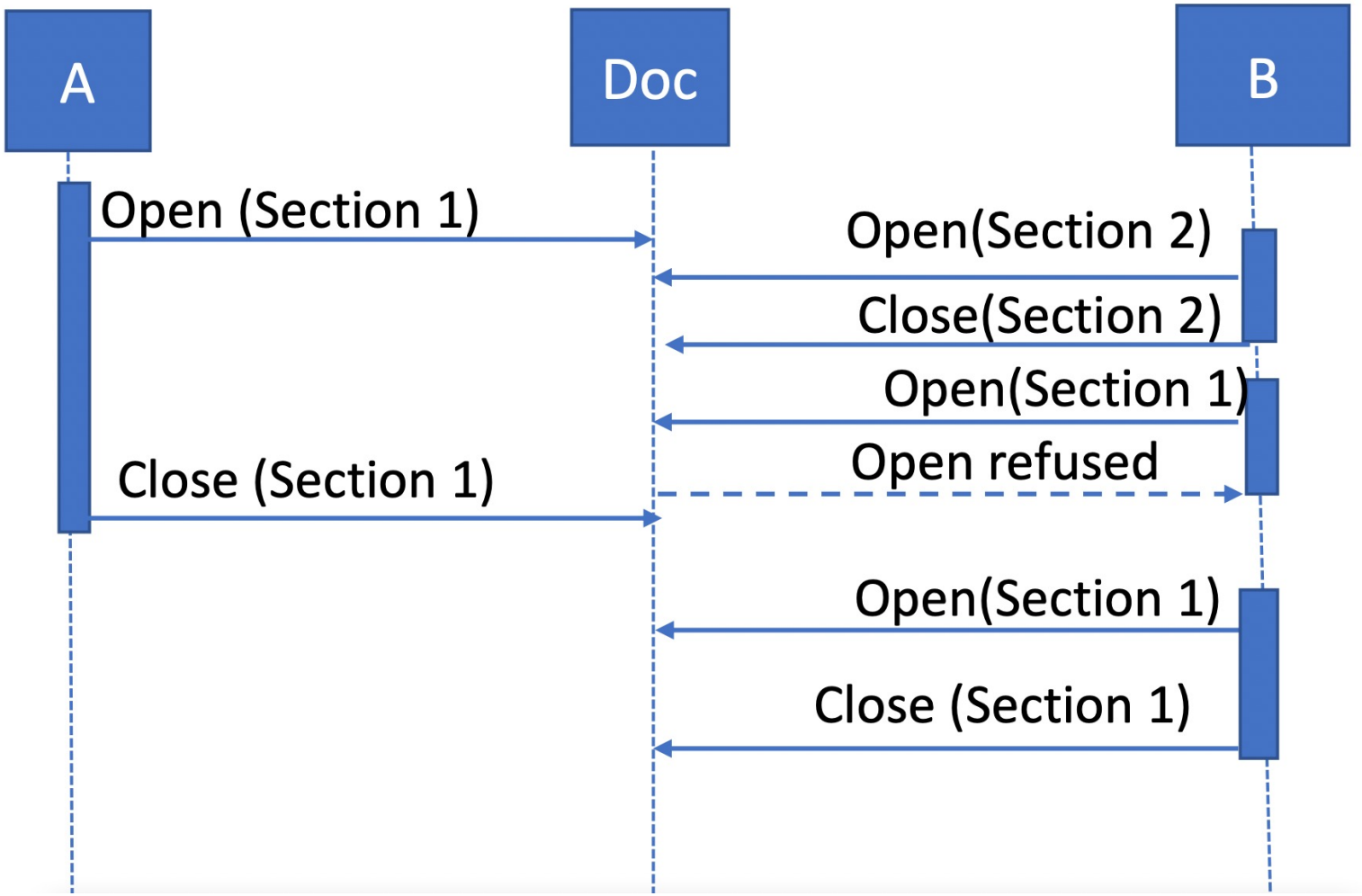
2 actors, 1 shared resource (e.g., document) => concurrency

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



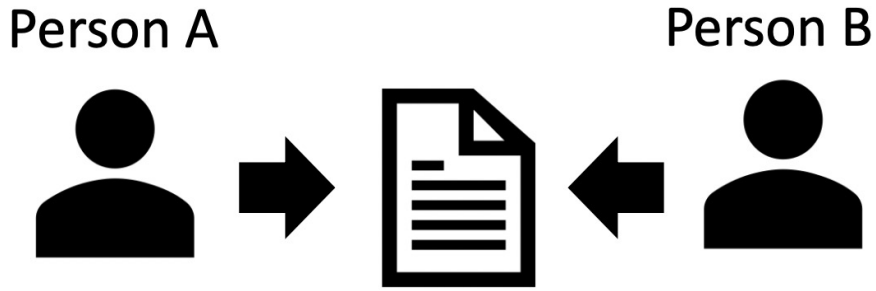
What Might Atomicity Look Like?



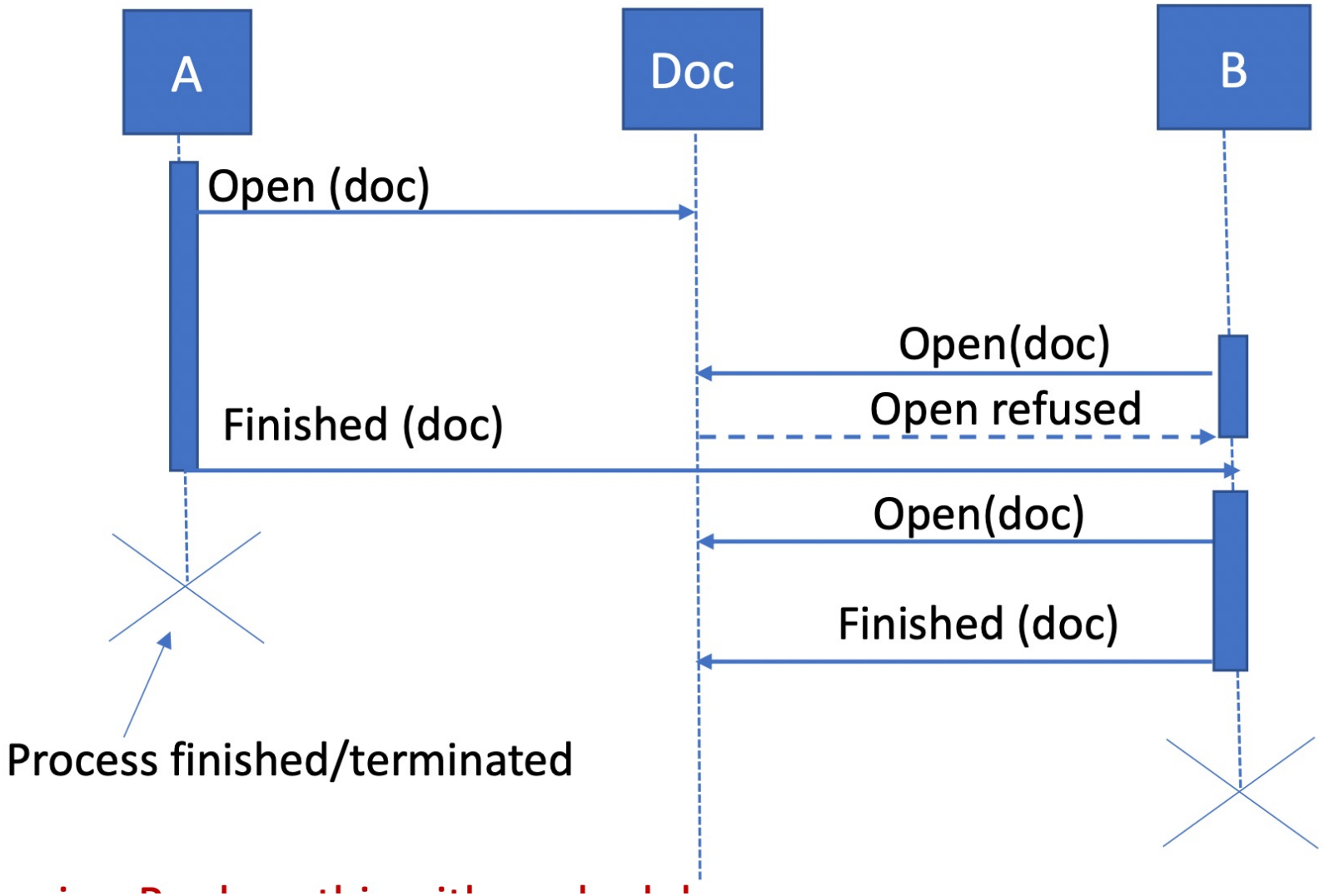
Real World Concurrency Problem: Writing a Document Together

2 actors, 1 shared resource (e.g., document) => concurrency

- 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)
  - Must ensure that Person B cannot start until Person A is finished



What Might Atomicity Look Like?



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

Mutual Exclusion solutions: Locks

Mutual Exclusion (ME) with Locks

- 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).

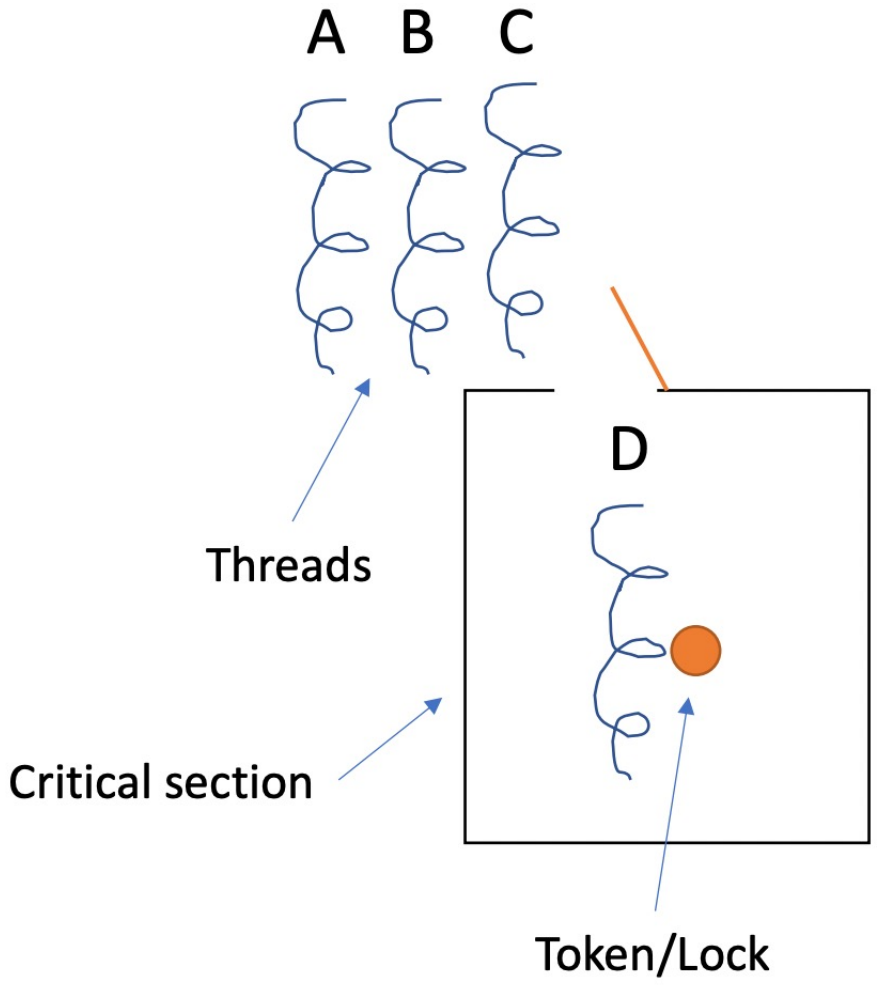
```
// Pseudo Code showing a critical section shared by
while (true)
        // different processes
        // Non_Critical_Section
        // Pre_protocol
        // Critical_Section
        // Post_protocol
end while
```

Locks: Basic Idea

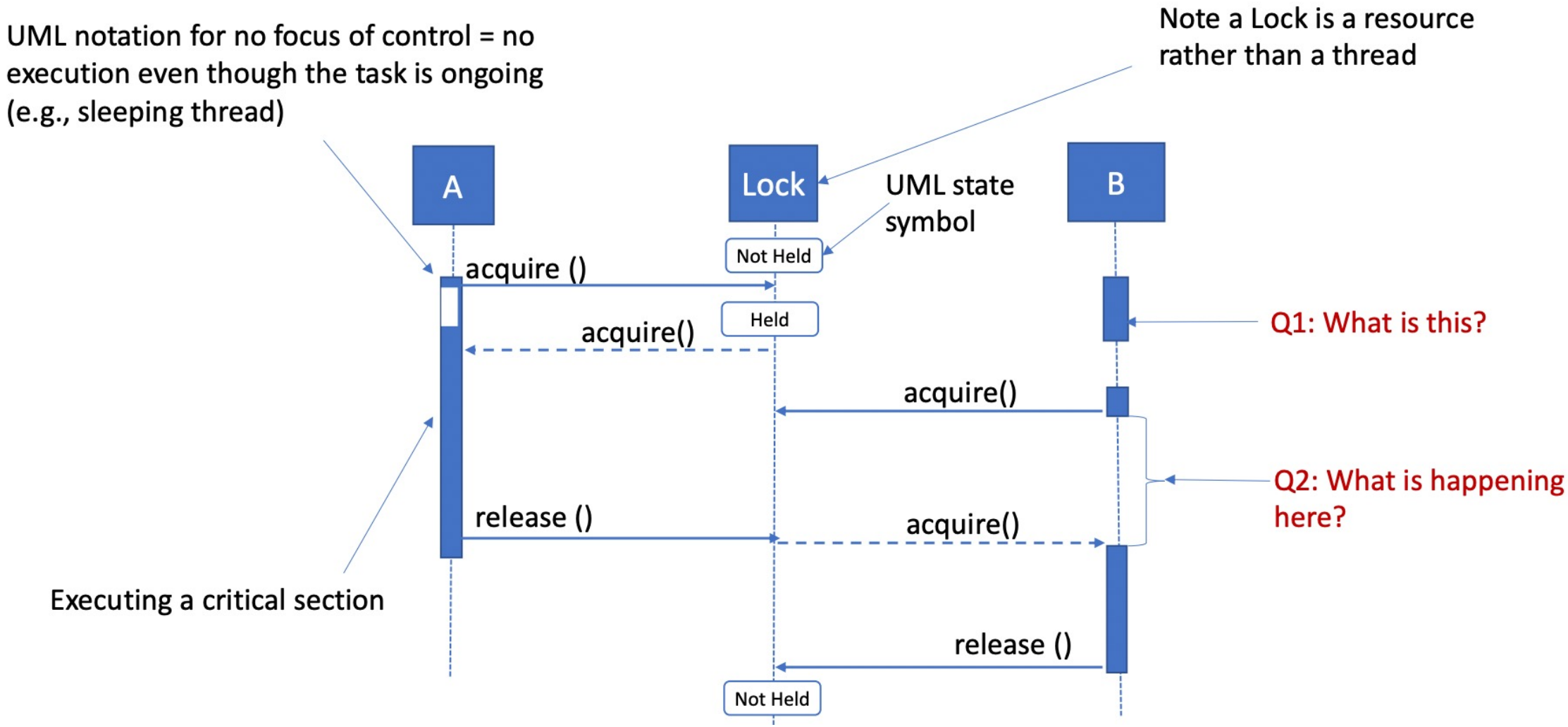
- Lock = a token you need to enter a critical section of code

- 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

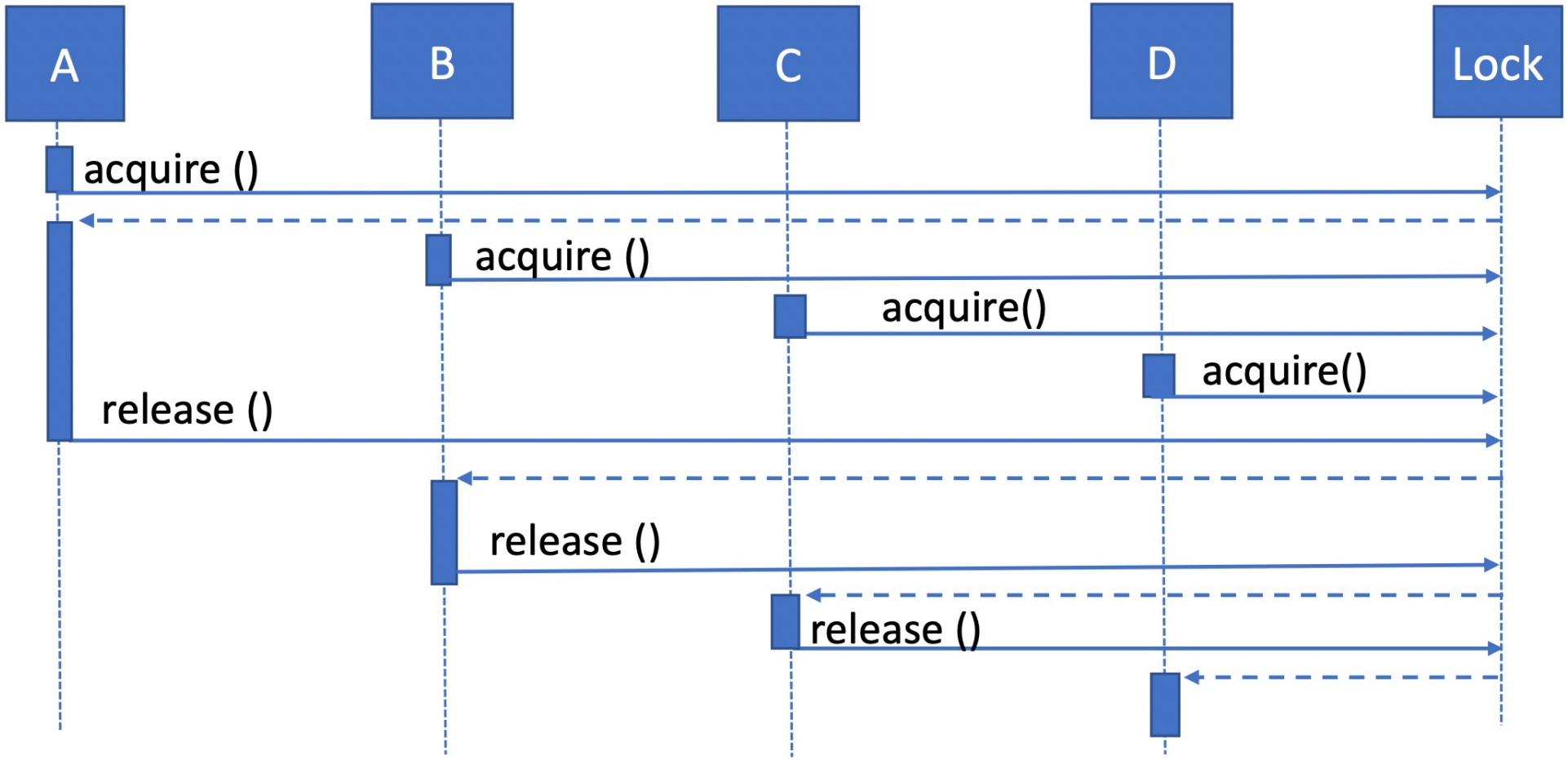


Lock States and Operation

- 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 lock as not held

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

Many Threads Want the Lock



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
```

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
    - Can have different threads in different critical sections at the same time
    - 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?

- E.g., managing critical sections that start to overlap in the data accessed
- E.g., we might want to add a new feature that accesses same data

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

The local variable, does not need to be protected

E.g., Bank Transaction with Locks

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

Thread 1

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

T2

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

T1

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

T2

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

T1

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

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

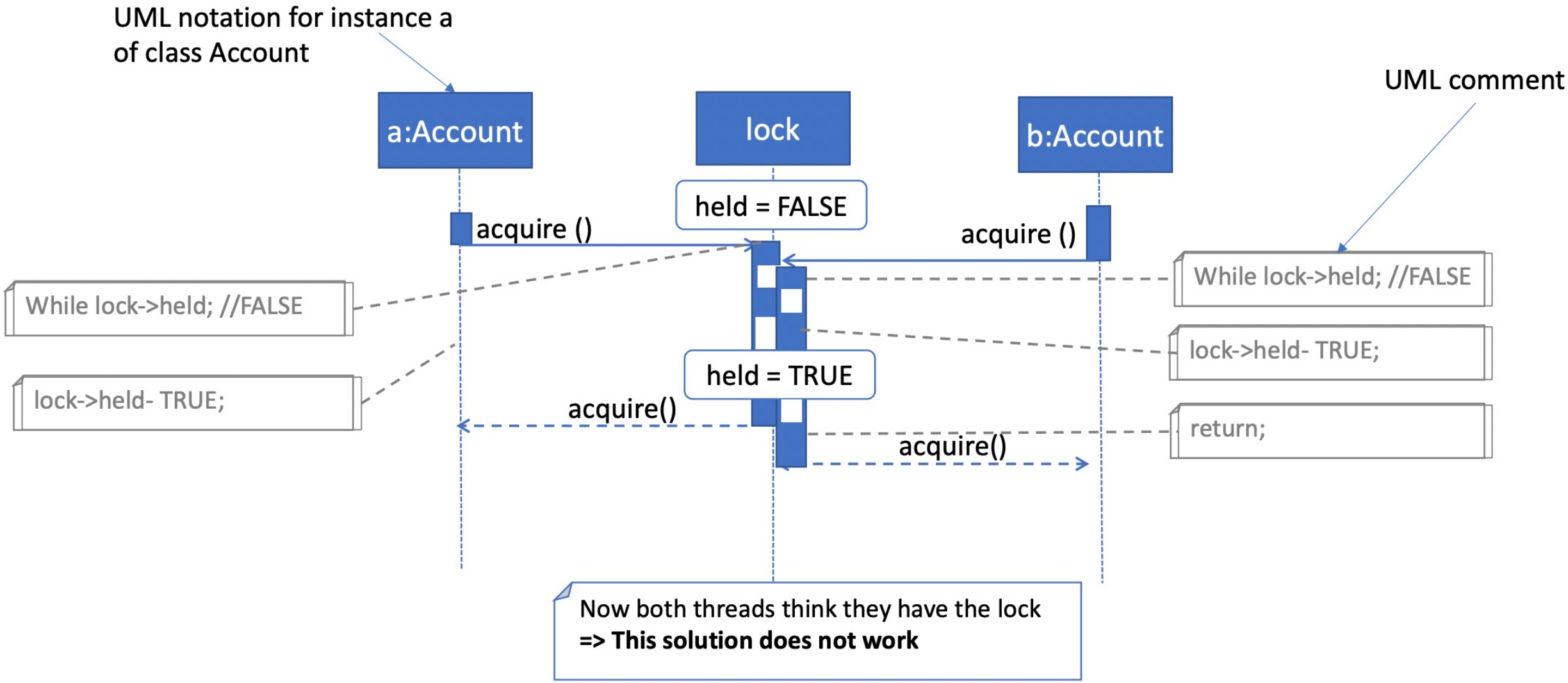
Impacts

- We can run the threads in any order:
  - We will 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? (and the problem)



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;
}
```

Solve via Hardware Support

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

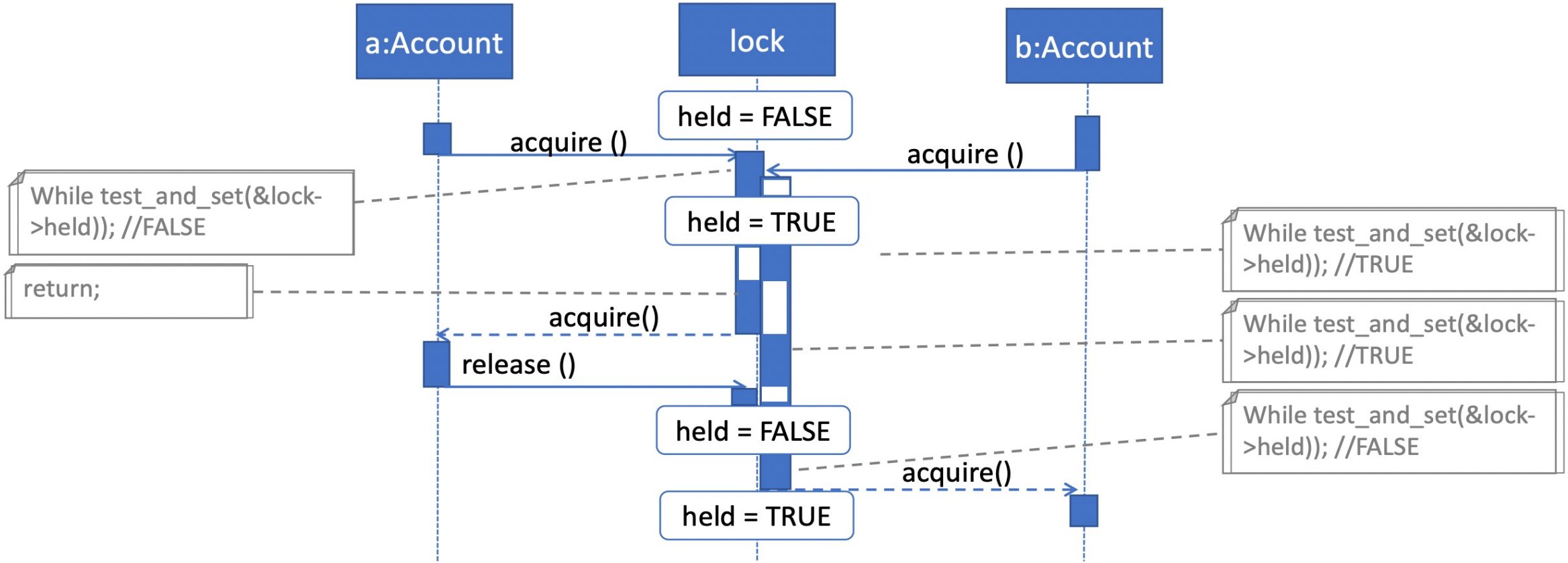
Processor has a special instruction called “test and set”

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;
}
```

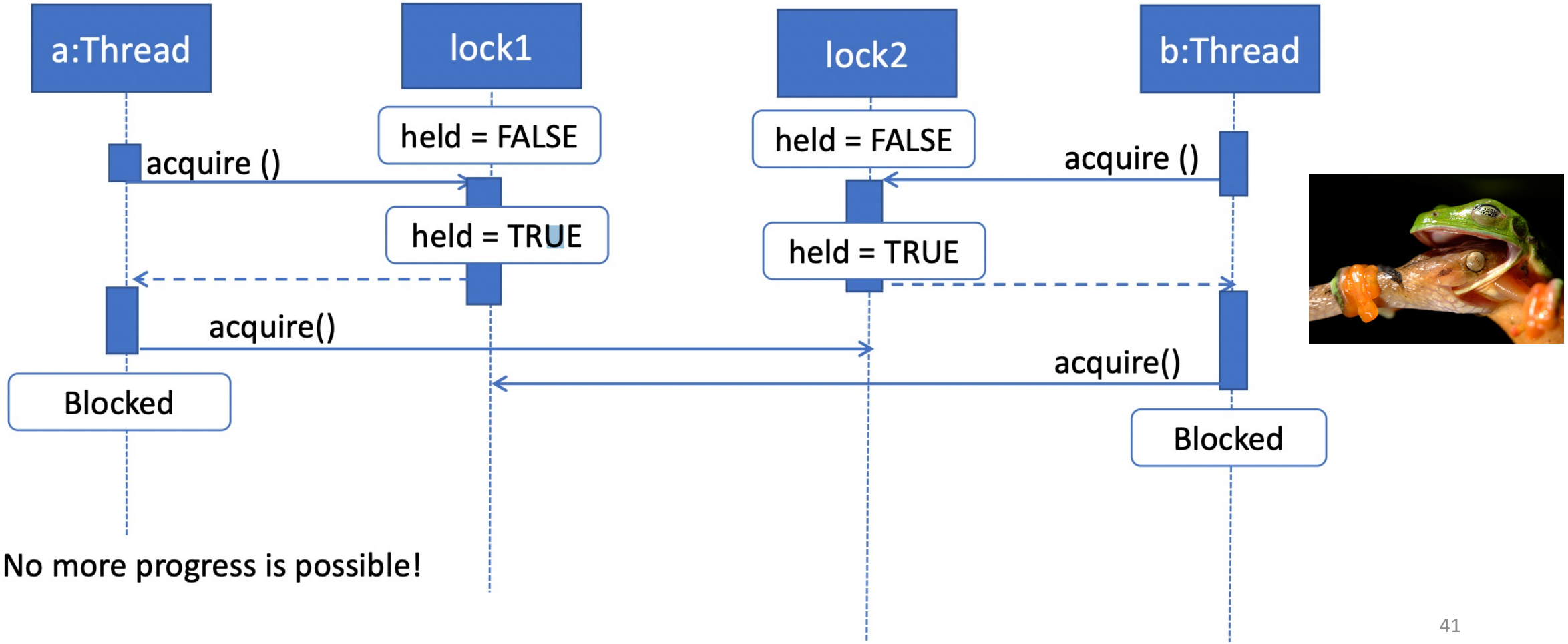
How does it run?



Q: Why is this called a spin lock?

Let's look at 2 locks: Lock Deadlock Scenario

- 2+ threads, 2 shared resources, 2 locks



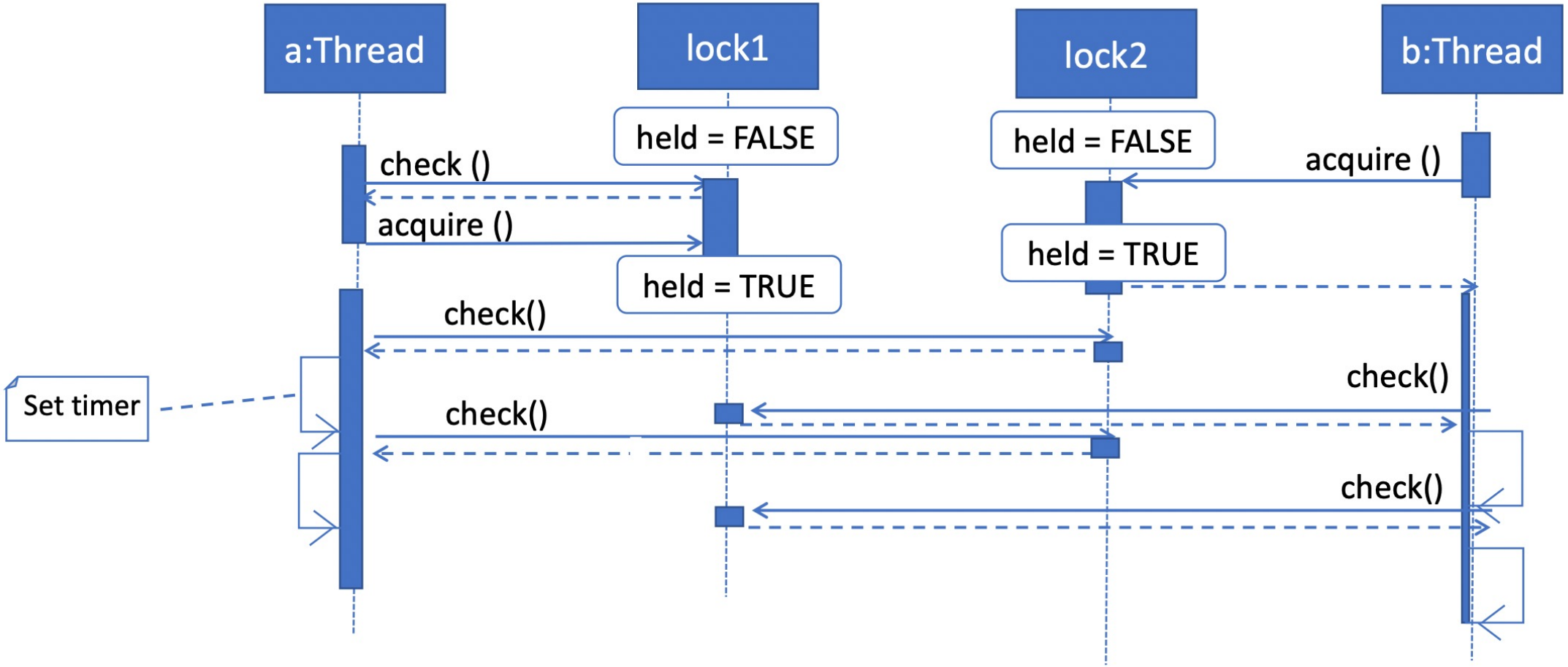
Protocols to avoid deadlock

- Add a timer to lock.request() method
  - Cancel job and attempt it another time
- Add a new lock.check() method to see if a lock is already held before requesting it
  - you can do something else and come back and check again
- Avoid hold and wait protocol
  - never hold onto 1 resource when you need 2
- Request locks in the same order each time

But these can 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
  - A sends many more requests for resource
  - B hardly ever gets allocated the resource

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 with Locks

- **Enforced fairness**: every thread gets a fixed time slice
  - naive
- **Thread priority**: threads have priority allocated to enable variable execution schemes
- **Barging**: 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
  - Designed to improve throughput

Fairness Solutions

Is there a queue in Linux for who get's the lock next?

- https://docs.kernel.org/locking/robust-futexes.html

".... The kernel creates a 'futex queue' internally, so that it can later on match up the waiter with the waker - without them having to know about each other. When the owner thread releases the futex, it notices (via the variable value) that there were waiter(s) pending, and does the sys_futex(FUTEX_WAKE) syscall to wake them up."

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
  - It avoids OS overhead of context switching
  - If OS Scheduler makes threads sleep while lock is held
    - 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

- Locks only provide mutual exclusion
  - Ensure only 1 thread is in the critical section at a time
  - Good for protecting our shared resource to prevent race conditions and avoid nondeterministic execution
  - E.g., bank balance but ..... We want more!

- What about fairness, avoiding starvation, and livelock?
  - We need to be able to do things like place an ordering on the scheduling of threads

Take Home Message

- Race conditions, deadlock, livelock, fairness, reliability are all concerns when writing concurrent code

- We can use **Locks** but ....

- Better to use patterns with higher level of abstraction to ensure you can deal with all correctness concerns
  - not just mutex

Demonstrating Failure of Locks (or any Concurrent System)

- **Check 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
- **Check Liveness properties**: These must eventually be true
  - **Absence of starvation**: Information sent is delivered
  - **Fairness**: That any contention must be resolved

If you can demonstrate **any** cases in which these properties do not hold .... then, **the system is not correct**

Mutual Exclusion WITHOUT Hardware Support

We need to run two similar threads with mutual exclusion on a shared critical section

But: we do not have any hardware support

We can only use the basic variables (bool, int, float, double) and operations (if/else, do/while)

i.e., we can use shared memory

```
Thread1()
{
    do {

        // entry section
        // critical section
        // exit section

        // remainder section

    } while (completed == false)
}
```

Attempt 1

```
Thread1()
{
    do {
        // entry section
        // wait until threadnumber is 1
        while (threadnumber == 2)
            {};

        // critical section

        // exit section
        // give access to the other thread
        threadnumber = 2;

        // remainder section
    } while (completed == false)
}
```

```
Thread2()
{
    do {
        // entry section
        // wait until threadnumber is 2
        while (threadnumber == 1)
            {};

        // critical section

        // exit section
        // give access to the other thread
        threadnumber = 1;

        // remainder section
    } while (completed == false)
}
```

Problem

- Each thread depends on the other for its execution.
  - both are forced to execute at the same rate: not acceptable.

- If one of the threads completes, then:
  - the second thread runs
  - then gives access to the completed one
  - then waits for its turn .... *forever*

- Why? the former thread is already completed
  - It would never run to return the access back to the latter one.
  - Hence, the second thread waits infinitely!

Attempt 2

- Idea: remove lockstep synchronization
  - use two flags (thread1 and thread2) to indicate the current status
  - update the two flags accordingly at the entry (true) and exit (false) section

```
Thread1()
{
    do {

        // entry section
        // wait until thread2 is in its critical section
        while (thread2 == true)
            {};

        // indicate thread1 entering its critical section
        thread1 = true;

        // critical section

        // exit section
        // indicate thread1 exiting its critical section
        thread1 = false;

        // remainder section
    } while (completed == false)
}
```

```
Thread2()
{
    do {

        // entry section
        // wait until thread1 is in its critical section
        while (thread1 == true)
            {};

        // indicate thread2 entering its critical section
        thread2 = true;

        // critical section

        // exit section
        // indicate thread2 exiting its critical section
        thread2 = false;

        // remainder section
    } while (completed == false)
}
```

Can we spot a way this will fail?

Attempt 2

- Idea: remove lockstep synchronization
  - use two flags (thread1 and thread2) to indicate the current status
  - update the two flags accordingly at the entry (true) and exit (false) section

```
Thread1()
{
    do {

        // entry section
        // wait until thread2 is in its critical section
        while (thread2 == true)
            {};

        // indicate thread1 entering its critical section
        thread1 = true;

        // critical section

        // exit section
        // indicate thread1 exiting its critical section
        thread1 = false;

        // remainder section
    } while (completed == false)
}
```

3 (i)

1 (i)

4

```
Thread2()
{
    do {

        // entry section
        // wait until thread1 is in its critical section
        while (thread1 == true)
            {};

        // indicate thread2 entering its critical section
        thread2 = true;

        // critical section

        // exit section
        // indicate thread2 exiting its critical section
        thread2 = false;

        // remainder section
    } while (completed == false)
}
```

2 (i)

4

Race condition....

Problems

- If **delay** during flag update (when changing flag to true)
  - both threads enter their critical section
- If both thread1 and thread2 are false
  - no mutual exclusion
- If one of the variables is true and the other is false
  - If the thread assigned true never starts, we have starvation
- ...

Attempt 3: Dekker's Algorithm

- Idea: remove lockstep synchronization
  - Two variables to announce threads want to enter critical section
  - use variable to determine favored thread to enter the critical section

```
int favouredthread = 1; // to denote which thread will enter next
// flags to indicate if each thread is in queue to enter its critical section
boolean thread1wantstoenter = false;
boolean thread2wantstoenter = false;
```

```
Thread1()
{
    do {
        thread1wantstoenter = true;
        // entry section
        // wait until thread2 wants to enter its critical sectior
        while (thread2wantstoenter == true) {
            // if 2nd thread is more favoured
            if (favaouredthread == 2) {
                // gives access to other thread
                thread1wantstoenter = false;
                // wait until this thread is favoured
                while (favouredthread == 2)
                    {};
                thread1wantstoenter = true;
            }
        }
        // critical section
        // favour the 2nd thread
        favouredthread = 2;
        // exit section
        // indicate thread1 has completed its critical section
        thread1wantstoenter = false;
        // remainder section
    } while (completed == false)
}
```

```
Thread2()
{
    do {
        thread2wantstoenter = true;
        // entry section
        // wait until thread1 wants to enter its critical section
        while (thread1wantstoenter == true) {
            // if 1st thread is more favoured
            if (favaouredthread == 1) {
                // gives access to other thread
                thread2wantstoenter = false;
                // wait until this thread is favoured
                while (favouredthread == 1)
                    {};
                thread2wantstoenter = true;
            }
        }
        // critical section
        // favour the 1st thread
        favouredthread = 1;
        // exit section
        // indicate thread2 has completed
        // its critical section
        thread2wantstoenter = false;
        // remainder section
    } while (completed == false)
}
```

Dekker's Algorithm

- An extension to last proposal:
  - Add a variable to explicitly pass right to enter Critical Sections between the processes (fairness, turn-taking),
- In Dekker's algorithm right to insist on entering a Critical Section is explicitly passed between processes
- Btw... Not safe on all hardware (optimisers can compromise this)

Dekker’s Algorithm

- Starvation = No
- Deadlock possible = No
- Mutual Exclusion = Yes

Higher Level Support for Mutual Exclusion: Semaphores & Monitors

Semaphores

Semaphore = higher level synchronisation primitive

- Invented by Dijkstra in 62 / 63 while working on ”THE OS” project

Implement with a counter that is manipulated **atomically** via 2 operations signal and wait

wait(semaphore): A.K.A., down() or P()

- if counter is zero then block until semaphore is signalled
- decrement counter

signal(semaphore): A.K.A., up() or V()

- increment counter
- wake up one waiter, if any

sem_init(semaphore, counter):

- set initial counter value

Semaphore Pseudocode

Note: wait() and signal() implementation are critical sections!

- Hence, they **must be executed atomically** with respect to each other

- Each semaphore has an associated queue of threads
  - When wait() is called by a thread
    - If semaphore is available => thread continues
    - If semaphore is unavailable => thread blocks, waits on queue
  - signal() opens the semaphore
    - If threads are waiting on a queue => one thread is unblocked
    - If no threads are on the queue => the signal is remembered for the next time wait() is called

Note: Blocking threads are not spinning, they release the CPU to do other work

```
struct semaphore {
        int value;
        queue L; // list of processes
}

wait (S) {
        if (s.value > 0)
                s.value = s.value - 1;
        else {
                add this process to s.L;
                block;
        }

signal (S) {
        if (S.L != EMPTY) {
                remove a process from S.L;
                wakeup(P);
         }
        else
                s.value = s.value + 1;
}
```

Semaphore Initialisation

- If semaphore initialised to 1
  - First call to wait goes through
    - Semaphore value goes from 1 to 0
  - Second call to wait() blocks
    - Semaphore value stays at zero, thread goes on queue
  - If first thread calls signal()
    - Semaphore value stays at 0
    - Wakes up second thread

Acts like a mutex lock

Can use semaphores to implement locks ...

This is called a **binary semaphore**

```
struct semaphore {
        int value;
        queue L; // list of processes
}

wait (S) {
        if (s.value > 0)
                s.value = s.value - 1;
        else {
                add this process to s.L;
                block;
        }

signal (S) {
        if (S.L != EMPTY) {
                remove a process from S.L;
                wakeup(P);
         }
        else
                s.value = s.value + 1;
}
```

Semaphore Pseudocode

Initial value of semaphore = number of threads that can be active at once:

Sem_init(sem, 2)

So, **value**=2, **L** =[]

Consider multiple threads:

- Thread1: wait(sem)
  - value=1,L=[], T1 **executes**
- Thread2: wait(sem)
  - value=0, L[], T2 **executes**
- Thread3: wait(sem)
  - value=0, L[T3], T3 **blocks**
- **…**

```
struct semaphore {
        int value;
        queue L; // list of processes
}

wait (S) {
        if (s.value > 0)
                s.value = s.value - 1;
        else {
                add this process to s.L;
                block;
        }

signal (S) {
        if (S.L != EMPTY) {
                remove a process from S.L;
                wakeup(P);
         }
        else
                s.value = s.value + 1;
}
```

Uses of Semaphores

Allocating a number of resources

- E.g., Shared buffers: each time you want to access a buffer, call wait() => you are queued if there is no buffer available
- **Pool of resources**: network connections, printers, ...

- Counter is initialised to N = number of resources
  - Called a counting semaphore
  - Useful for conditional synchronisation
    - i.e., one thread is waiting for another thread to finish a piece of work before it continues (*as we'll see in a little bit...*)

Semaphores for Mutual Exclusion

- With semaphores:
  - guaranteeing mutual exclusion for N processes is trivial

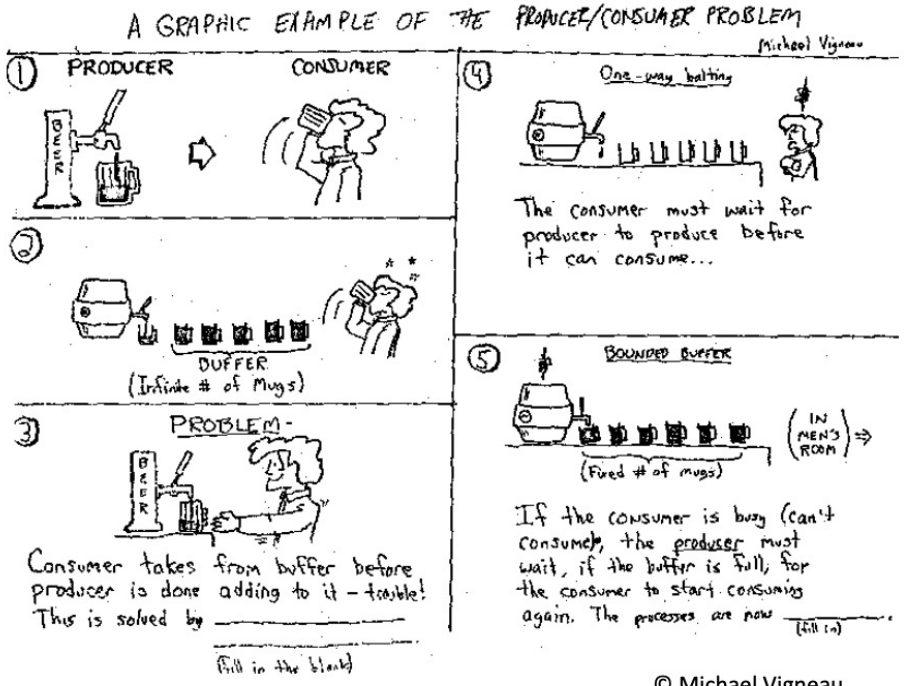
```
semaphore mutex = 1;
void P (int i) {
        while (1) {
                // Non Critical Section Bit
                wait(mutex) // grab the mutual exclusion semaphore
                // Do the Critical Section Bit
                signal(mutex)
                //grab the mutual exclusion semaphore
        }
}
int main ( ) {
        cobegin { P(1); P(2); }
}
```

Bounded Buffer (Producer-Consumer) Problem

Producer-consumer problem

- Buffer in memory
  - Finite size of N entries
- A producer process inserts an entry into it
- A consumer process removes an entry from it

- Processes are concurrent



- We must use a synchronisation mechanism to control access to shared variables describing buffer state

Producer-Consumer Single Buffer

- Simplest case
  - Single producer thread **&** single consumer thread
  - Single shared buffer between the Producer and the Consumer
  -
- Requirements
  - Consumer must wait for Producer to fill buffer
  - Producer must wait for Consumer to empty buffer (if filled)

Exercise

- How many shared data values do you have?
  - **lock**, **spaces**, **elements**
- How should we initialise **`lock`** ?
- How should we initialise **`spaces`** ?
- How should we initialise **`elements`** ?

Producer

```
while(1) {
        wait(&spaces);
        wait(&lock);
        fill(&buffer); // CS
        signal(&lock);
        signal(&elements);
}
```

Consumer

```
while(1) {
        wait(&elements);
        wait(&lock);
        use(&buffer); // CS
        signal(&lock);
        signal(&spaces);
}
```

Exercise

- How many shared data values do you have?
  - **lock**, **spaces**, **elements**
- How should we initialise **`lock`** ?
- How should we initialise **`spaces`** ?
- How should we initialise **`elements`** ?

```
struct semaphore {
        int value;
        queue L; // list of processes
}

...

signal (S) {
        if (S.L != EMPTY) {
                remove a process from S.L;
                wakeup(P);
         }
        else
                s.value = s.value + 1;
}
```

Producer

```
while(1) {
        wait(&spaces);
        wait(&lock);
        fill(&buffer); // CS
        signal(&lock);
        signal(&elements);
}
```

Consumer

```
while(1) {
        wait(&elements);
        wait(&lock);
        use(&buffer); // CS
        signal(&lock);
        signal(&spaces);
}
```

Exercise

- How many shared data values do you have?
  - **lock**, spaces, elements
- How should we initialise **`lock`** ?
- How should we initialise **`spaces`** ?
- How should we initialise **`elements`** ?

How does this compare to using locks? E.g., if we just locked the critical section before and after

Producer

```
while(1) {
        wait(&spaces);
        wait(&lock);
        fill(&buffer); // CS
        signal(&lock);
        signal(&elements);
}
```

Consumer

```
while(1) {
        wait(&elements);
        wait(&lock);
        use(&buffer); // CS
        signal(&lock);
        signal(&spaces);
}
```

Types of Semaphores

- Based on the values it could take:
  - **General semaphore**: can take the values from 0 and N (>=1) defined by the user
  - **Binary semaphore**: can only take the values 0 and 1.
- Based on which suspended process to wake:
  - **Blocked-set semaphore**: Wakes any one suspended process
  - **Blocked-queue semaphore**: Suspended processes are kept in FIFO
  - Processes are woken in order of suspension
- Busy-wait semaphore:
  - Semaphore value is continuously checked in a busy-wait loop
  - Busy-wait consumes the processor
  - A.K.A., Spinning or busy looping

Semaphores can be hard to use

- Enables complex patterns of resource usage (like our producer-consumer)
  - But cannot always capture relationships with semaphores alone
  - Need extra state variables to record information (see Sleeping Barber later)
  - *Often* use semaphores such that
    - One is for mutex around state variables
    - One for each class of waiting

Also, danger of producing buggy code that is hard to write

- e.g., If one coder forgets to do V() / signal() after critical section, the whole system can deadlock

Monitors

- Need a higher level construct:
  - Groups the responsibility for correctness
  - Supports controlled access to shared data
    - Synchronisation code added by compiler, enforced at runtime
  - Monitors: an extension of the monolithic monitor used in OS to allocate memory etc.
    - A programming language construct that supports controlled access to shared data
    - Synchronization code added by compiler, enforced at runtime i.e. less work for programmer!
      - i.e., Monitors keep track of who is allowed to access the shared data and when they can do it
  - Encapsulate
      - Shared data structures
      - Procedures that operate on shared data
      - Synchronization between concurrent processes that invoke these procedures
    - Ensure only one process execute a monitor procedure at once
      - Atomicity
    - Guarantees only way to access the shared data is through procedures
- Supported natively in Java 5+

Implementation with Monitors

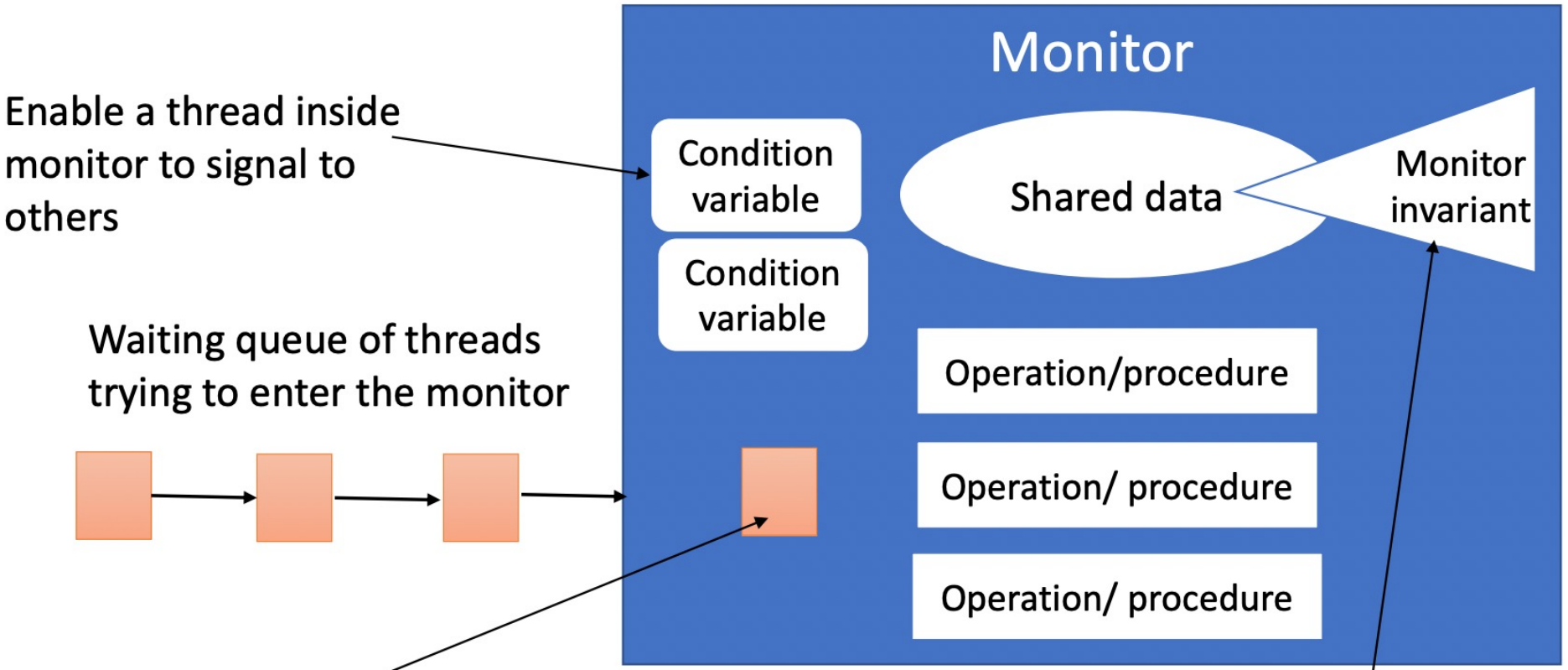
- Similar to a class in Java
  - Encapsulates a set of shared data and a set of operations to manipulate it
  - Similar to accessing a private member variable

- Using Monitors: When you identify a set of shared data being used by multiple threads:
  1. Create a monitor to contain the data
  2. Create a set of operations to work on the data
  3. Define a set of synchronisation rules between threads that invoke the operations

  You can control when a thread should + should not execute

- Protects you against synchronization issues

Monitor Concept



=> Provides **mutex** for the operations/procedures acting on the data

Monitor Invariants

- These are consistency rules for shared data
  - Bounded Buffer example:
    - 0 <= Count of items <= size of buffer
  - Linked List example
    - All list items have both forward and back link for a doubly linked list
  - Priority queue example
    - For all items I, priority(i) <= priority(successor(i))
- Monitor invariant must hold whenever monitor lock is free
    - While held, can violate to manipulate data structures
    - Nobody else can see intermediate states
  - With Monitors.. A thread manipulating the data must always ensure the invariant is true before it releases the lock
  - A thread entering the monitor can always assume the invariant is true

Monitors in Java

- Built-in language feature
  - Use synchronized keyword on a method
  - Compiler automatically creates a lock for increment() and generates code to acquire this lock on entry to increment()
  - You don’t have to
    - Write the code
    - Remember that this is shared state
    - Acquire or release the lock
  - Only a single condition available
    - Can be used with wait() and notify()

```
Class Counter {
        private int count = 0;
        public void synchronized
    increment() {
                int n = count;
                count = n + 1;
        }
}
```

Condition Variables

- Key feature of monitors

  Like wait/signal of **semaphores** but different **semantics**
- A place to wait, sometimes called a *rendezvous* point
  - Always used with a monitor lock
  - No value (history) associated with condition variable i.e. unlike a semaphore
- Operations on condition variables
  - Wait(c)
    - Release monitor lock, so another thread can get in

      Can check variables without holding the lock
    - Eliminates many hold and wait conditions
      - Wait for someone else to signal condition
    - Condition variables have wait queues
      - Must ensure invariant is true before you call wait

Condition Signalling

- Signal(c) (or notify(c)) means
  - Wake one thread waiting on this condition variable (if any)
    - Signaller can keep lock and CPU
- Waiter is made ready*, but the signaller continues
  - Waiter runs when signaller leaves monitor or waits
  - Condition is not necessarily true when waiter runs again
  - Signaller need not restore invariant until it leaves the monitor

- Broadcast(c) (or NotifyAll)
  - Wake all threads waiting on condition variable
  - Avoids need for multiple condition variables

* i.e., it joins the queue waiting for the lock outside the monitor

Waiting

- Waiting on a condition variable releases the lock and puts the thread on the condition variable's wait queue
  - Guarantees no other thread can enter the monitor before this thread is on the queue
  - i.e. no thread can call signal before it is put on queue
- Threads stay on the queue until signalled (and at head of queue) or broadcast
- After signalled, threads wait to acquire the monitor lock before running
  - There could be other threads ahead of it in the queue

General use of Signal

```
Class Counter {
        //Prints value every time count>0
        private int count = 0;

        public void synchronized increment() {
                int n = this.count;
                this.count = n + 1;
                If (count > 0)
                    notify(); //let print know
        }

        public void synchronized decrement(){
                int n = count;
                count = n - 1;
        }

        public void synchronized printVal() {
                if (count <= 0)
                    wait();

        System.out.println("Count=" + count);
        }
}
```

Problem: no guarantee notified thread runs right away!

E.,g.,

Count = -1

T1: increment //count = 0

T2 printVal // count=0, waits

T3: increment //count=1, signals

T1 decrement //count = 0

T2: printVal wakes up //count = 0 WRONG as signal() just marks another thread as eligible to run, scheduler decides which eligible thread runs

Solution: Treat Waking as a Hint

- Woken up => something has changed
- Another thread may have entered monitor between the signal() and the wake up
- Implication: must re-check conditional after waking
  - Test in a while loop
  - When thread wakes it will re-test

```
public void synchronized printVal() {
        while (count <= 0)
                wait();
        System.out.println(“Count=" + count);
}
```

Monitors (cont'd): Signal & Continue

- As a monitor guarantees mutual exclusion:
  - A process uses the signal (notify()) operation
  - And wakes up another process suspended in the monitor

- So 2 processes in same monitor at once????
  - Yes…

    but one is sleeping while waiting for a signal

Detection and Protection of Deadlock

Requirements for Deadlock

1. **Mutex**: at least one held resource must be non-shareable

2. **No pre-emption**: resources cannot be pre-empted (a resource can be released only voluntarily by the process holding it)

   .... Locks have this property

3. **Hold and wait**: there exists a process holding a resource and waiting for another resource

4. **Circular wait**: there exists a set of processes P_1, P_2,…,P_N such that P_1 is waiting for P_2, P_2 is waiting for P_3,… and P_N is waiting for P_1

Sample Deadlock

Acquire locks in different orders

Example:

Thread 1	**Thread2**
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x);	unlock(y);

Sample Deadlock – Check for Deadlock

Acquire locks in different orders

Example:

Thread 1	**Thread2**
lock(x);	**lock**(y);
A=A+10;	B=B+10;
lock(y);	**lock**(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x);	unlock(y);

Do we have mutex?

Do we have hold and wait?

Do we have no pre-emption?

Do we have a circular wait?

Dealing With Deadlocks: Ignore

- Strategy 1: Ignore the fact that deadlocks may occur
  - Write code, put nothing special in
  - Sometimes you have to re-boot the system
  - May work for some unimportant or simple applications where deadlock does not occur often

Quite a common approach ....

Dealing with Deadlock: Reactive

- Periodically check for evidence of deadlock
  - E.g. add timeouts to acquiring a lock, if you timeout then it implies deadlock has occurred and you must do something
  - Recovery actions:
    - Blue screen and re-boot computer
    - Pick a thread to terminate e.g., a low priority one
      - Only works with some types of applications
    - May corrupt data so thread needs to do cleanup when terminated
    - Then thread often re-tries from start
    - But this breaks the pre-emption condition
      - Databases often do this as state is generally well known

Better to have not ended up in the situation of deadlock in the in the first place!

Dealing with Deadlock: Proactive

- Prevent 1 of the 4 necessary conditions for deadlock

- No single approach is appropriate (or possible) for all circumstances
  - Need techniques for each of the four conditions

Solution 1: No Mutual Exclusion

Make resources shareable

- Example: read-only files
  - No need for locks
- Example: per-thread variables
  - Counters per thread instead of global counter

But not possible for all segments of code/applications ...

Solution 1: Avoid Hold and Wait

Only request a resource when you have none

I.e., release a resource before requesting another

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
lock(y);	lock(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	unlock(x);
unlock (x);	unlock(y);

Thread 1	Thread2
lock(x);	lock(y);
A=A+10;	B=B+10;
unlock(x);	unlock(y);
lock(y);	lock(x);
B=B+20;	A=A+20;
unlock(y);	unlock(x);
lock(x);	lock(y);
A=A+30;	B=B+30;
unlock (x);	unlock(y);

- Never hold x when want y:
  - Works in many cases
  - But you cannot maintain a relationship between x and y

Solution 2: Avoid Hold and Wait using Atomicity

Acquire all resources at once

- E.g., use a single lock to protect all data
- Having fewer locks is called **lock coarsening**

Thread 1	**Thread2**
lock(x);	**lock**(y);
A=A+10;	B=B+10;
lock(y);	**lock**(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	**unlock**(x);
unlock (x);	**unlock**(y);

Thread 1	**Thread2**
lock(z);	**lock**(z);
A=A+10;	B=B+10;
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock (z);	**unlock**(z);

- Problem: low concurrency
  - All threads accessing A or B cannot run at the same time
  - Even if they don't access both variables!

Prevention: Adding Pre-emption

- Locks cannot be pre-empted but other pre-emptive methods are possible
- **Strategy**: pre-empt resources
- Example:
  - If thread A is waiting for a resource held by thread B, then take the resource from B and give it to A
- Problems:
  - Only works for some resources
  - E.g., CPU and memory (using virtual memory)
  - Not possible if a resource cannot be saved and restored
  - Otherwise, taking away a lock causes issues
  - Also, there is an overhead cost for “pre-empt” and “restore”

Prevention: Eliminate Circular Waits

- Strategy: Impose an ordering on resources
  - Threads must acquire the highest ranked resource first

Thread 1	**Thread2**
lock(x);	**lock**(y);
A=A+10;	B=B+10;
lock(y);	**lock**(x);
B=B+20;	A=A+20;
A=A+30;	B=B+30;
unlock(y);	**unlock**(x);
unlock (x);	**unlock**(y);

Thread 1	**Thread2**
lock(x);	**lock**(x);
lock(y);	**lock**(y);
A=A+10;	B=B+10;
B=B+20;	A=A+20;
unlock(x);	**unlock**(x);
A=A+30;	B=B+30;
unlock (y);	**unlock**(y);

- Problem: low concurrency
  - All threads accessing A or B cannot run at the same time
  - Even if they don't access both variables!

Preventing Circular Wait: Lock Hierarchy

- Strategy: Define an ordering of all locks in your program
  - Always acquire locks in that order
- Problem: Sometimes you do not know the order that the events will be used

How do we know the global order?

- Need extra code to find this out and then acquire them in the right order

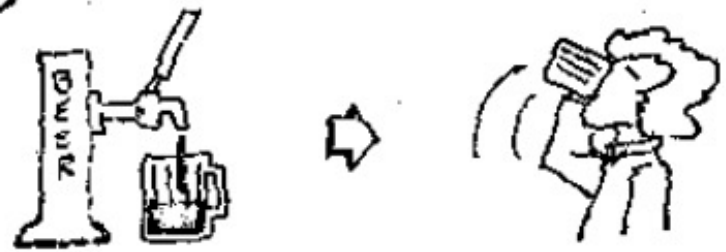
```
transfer(acc1, acc2, amount) {
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
}
```

Classical Problems of Synchronization

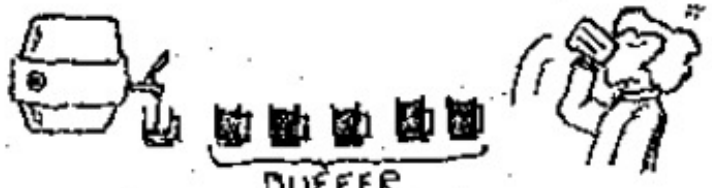
A GRAPHIC EXAMPLE OF THE PRODUCER/CONSUMER PROBLEM

Michael Vigneau

① PRODUCER CONSUMER



② DUFFER
(Infinite # of Mugs)

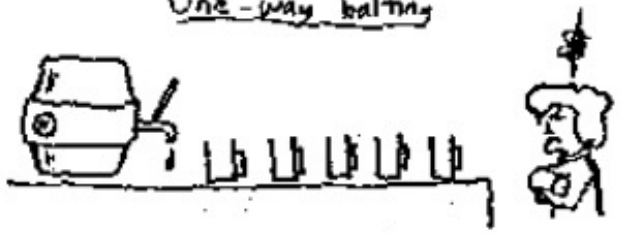


③ PROBLEM -

Consumer takes from buffer before producer is done adding to it - trouble! This is solved by _____

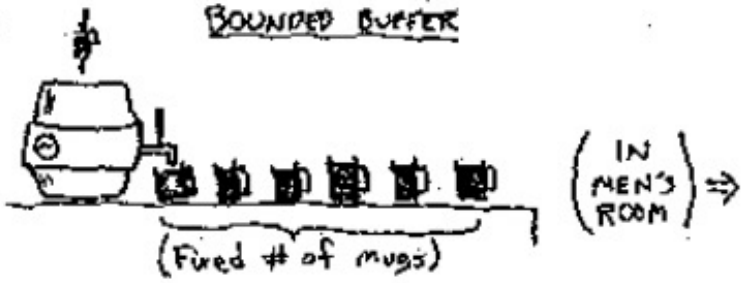
(Fill in the blank)

④ One-way halting



The consumer must wait for producer to produce before it can consume...

⑤ BOUNDED BUFFER



(Fixed # of mugs)

If the consumer is busy (can't consume), the producer must wait, if the buffer is full, for the consumer to start consuming again. The processes are now _____.

(fill in)

The Producer-Consumer Problem

- This type of problem has two types of processes:
  - **Producers**: processes that, from some inner activity, produce data to send to consumers
  - **Consumers**: processes that, on receipt of a data element, consume data in some internal computation

- We can join processes synchronously:
  - So, data is only sent when producer can send it and consumer can receive.
- Connect them by a buffer
- For an infinite buffer, the following invariants hold for the buffer:

$$\#elements \geq 0$$

$$\#elements = 0 + in\_pointer - out\_pointer$$

The Producer-Consumer Problem (Bounded Circular Buffer)

```
int in_pointer = 0, out_pointer = 0
semaphore elements = 0; // items produced
semaphore spaces = N; //spaces left

void producer( int i ) {
        while (1) {
                item = produceItem();
                wait( spaces );
                putItemIntoBuffer( item, in_pointer );
                in_pointer = ( in_pointer+1 ) mod N;
                signal( elements );
        }
}

void consumer( int j ) {
        while (1) {
                wait( elements );
                item = removeItemFromBuffer( out_pointer );
                out_pointer = ( out_pointer+1 ) mod N
                signal( spaces );
                consumeItem(item);
        }
}

int main ( ) {
        cobegin {
                producer(1); producer (2); consumer (1);
        consumer (2); consumer (3);
        }
}
```

//Spaces = 4

T1 p1: produce item, putItemIntoBuffer //spaces= 3 , elements = 1

T2 p2: produce item, putItemIntoBuffer //spaces= 2 , elements = 2

T1 p1: produce item, putItemIntoBuffer //spaces= 1 , elements = 3

T3 c1: removetemFromBuffer //spaces= 2 , elements = 2

T1 p1: produce item, putItemIntoBuffer //spaces= 1 , elements = 3

T2 p2: produce item, putItemIntoBuffer //spaces= 0 , elements = 4

T2 p2: produce item, wait //BLOCKED spaces= 0 , elements = 4

Semaphore Pseudocode

Note: wait() and signal() implementation are critical sections!

- Hence, they **must be executed atomically** with respect to each other

- Each semaphore has an associated queue of threads
  - When wait() is called by a thread
    - If semaphore is available => thread continues
    - If semaphore is unavailable => thread blocks, waits on queue
  - signal() opens the semaphore
    - If threads are waiting on a queue => one thread is unblocked
    - If no threads are on the queue => the signal is remembered for the next time wait() is called

Note: Blocking threads are not spinning, they release the CPU to do other work

```
struct semaphore {
        int value;
        queue L; // list of processes
}

wait (S) {
        if (s.value > 0)
                s.value = s.value - 1;
        else {
                add this process to s.L;
                block;
        }

signal (S) {
        if (S.L != EMPTY) {
                remove a process from S.L;
                wakeup(P);
         }
        else
                s.value = s.value + 1;
}
```

The Producer-Consumer Problem (cont'd)

- Code shows the case of a real, bounded circular buffer used to count empty places/spaces in the buffer

Can show the following:

i. No deadlock
ii. No starvation
iii. No data removal from an empty buffer
iv. No data appending to a full buffer

The Dining Philosophers Problem

- DCU hires 5 philosophers for hard problems
- Philosophers only **think** & **eat**
- Dining table has five plates & five forks
  ... or five bowls and five chopsticks
- Philosophers need 2 forks to eat
- Each plate is endlessly refilled
- Philosopher may only pick up the forks immediately to their left or right

Dining Philosophers (cont'd)

- For the system to operate correctly it is required that:
  - A philosopher eats only if he has two forks
  - No two philosophers can hold the same fork simultaneously
  - No deadlock, no starvation, efficient behaviour under the absence of contention
- Challenge: Develop an algorithm where no philosopher starves
- Question: What could go wrong?
- This problem is an analogy of multiple processes accessing a set of shared resources
  - e.g., a network of computers accessing a bank of printers

First attempt

```
void philosopher(int id) {
        while(TRUE) {
                think(); // not CS

                take_fork( right );
                take_fork( left );

                eat(); // CS

                put_fork( left );
                put_fork( right );
        }
}
```

Any issues?

Dining Philosophers: Solution #1

- Model each fork as a semaphore
- Then each philosopher must wait() on both the left and right forks before eating

```
semaphore fork [5] := ((5) 1)
/* pseudo-code */
/* fork is array of semaphores all initialised to have value 1 */
process philosopher (i := 0 to 4) {
        while (1) {
                think ( );
                wait( fork (i) );        // grab fork[i]
                wait( fork ((i+1) mod 5 );       // mod allows for circular

                eat ( ); // CS

                signal( fork (I ) );     // release fork[i]
                signal( fork ((i+1) mod 5 );      // release rh fork
        }
}
```

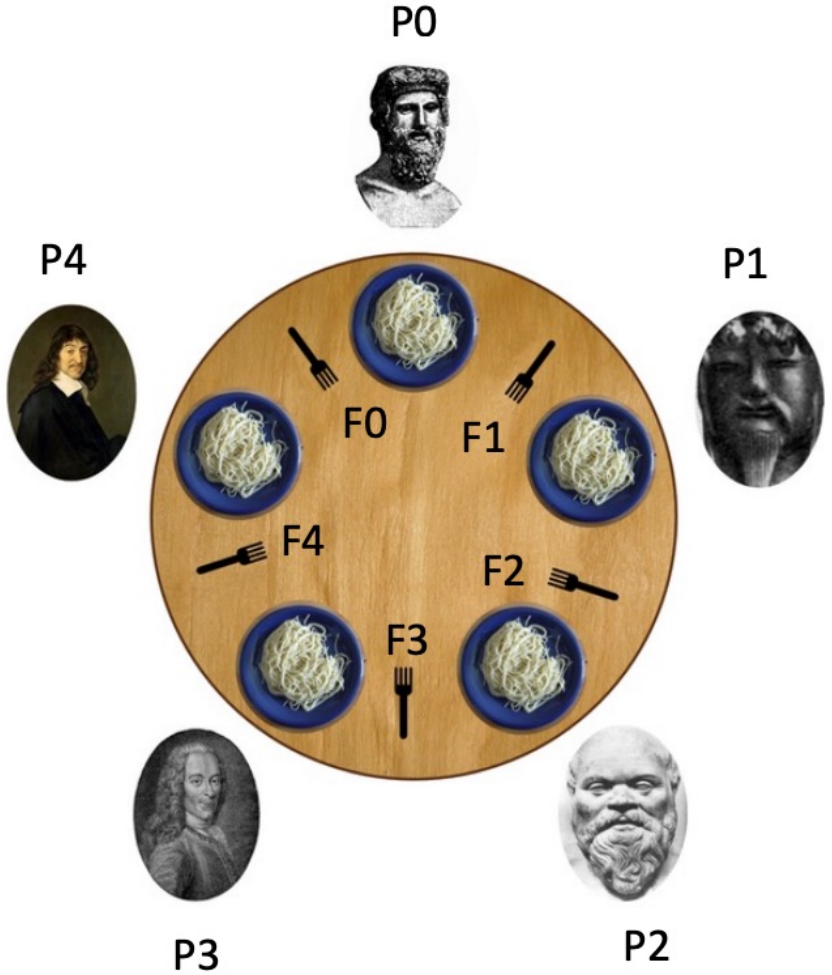
Look at this in Action

What happens when everyone picks up left fork?

```
semaphore fork [5] := ((5) 1)
/* pseudo-code */
/* fork is array of semaphores all initialised to have value 1 */
process philosopher (i := 0 to 4) {
        while (1) {
                think ( );
                wait( fork (i) );        // grab fork[i]
                wait( fork ((i+1) mod 5 );       // mod allows for circular

                eat ( ); // CS

                signal( fork (I ) );     // release fork[i]
                signal( fork ((i+1) mod 5 );      // release rh fork
        }
}
```



Dining Philosophers: Solution #1

- Called a *symmetric solution* as each task is identical.
- Symmetric solutions have advantages, e.g. for load-balancing.
- Can prove no 2 philosophers hold same fork - as `Eat()` is fork's CS.
  - If $\#P_i$ is number of philos with fork *i* then $\boldsymbol{Fork(i) + \#P_i = 1}$

    (ie either philo has the fork or sem is 1)
- But deadlock possible (i.e none can eat) when all philos pick up their left forks together;
  - i.e. all execute `P(fork[i])` before `P(fork[(i+1)mod 5]`

- Two solutions:
  - Make one philosopher take a right fork first (asymmetric solution);
  - Only allow four philosophers into the room at any one time.

Dining Philosophers #2: Symmetric Solution

```
semaphore Room := 4
semaphore fork [5] := ((5) 1)
/* pseudo-code */
/* fork is array of semaphores all initialised to have value 1 */
process philosopher (i := 0 to 4) {
        while (1) {
                think ( );
                wait( Room );
                wait( fork (i));
                wait( fork ((i+1) mod 5 );
                eat ( );  <------- Critical Section
                signal( fork (I ));
                signal( fork ((i+1) mod 5 );
                signal( Room );
        }
}
```

This solution solves the deadlock problem, added a semaphore hierarchy to **eliminate hold and wait AND** It is also **symmetric** (i.e., all processes execute same code)

Dining Philosophers#2: Asymmetric Solution

```
semaphore fork [5] := ((5) 1)
/* pseudo-code */
/* fork is array of semaphores all initialised to have value 1 */
process philosopher (i := 0 to 4) {
        while (1) {
                think ( );
                wait( min (fork (i), fork ((i+1) mod 5 );
                wait( max (fork (i), fork ((i+1) mod 5 );

                eat ( ); <------- Critical Section

                signal( max (fork (i), fork ((i+1) mod 5 );
                signal( min (fork (i), fork ((i+1) mod 5 );
        }
}
```

- This solution solves the deadlock problem
- It is also asymmetric as the last philosopher now picks up his right fork first to preserve the order of resources (forks)

The Readers-Writers Problem

- Two kinds of processes (i.e., readers and writers) share a DB
  - Readers run transactions that examine the DB
  - Writers can examine/update the DB
- To insure DB consistency:
  - writer process must have exclusive access
- Any number of readers may concurrently access the DB
- Obviously, for writers, writing is a Concurrent process
  - They cannot interleave with any other process

The Readers-Writers Problem (cont'd)

```
int M := 20; int N := 5;
int nr := 0; //numReaders
sem mutexR = 1; sem rw = 1

process reader (i:= 1 to M) {
        while (1) {
                wait ( mutexR );
                nr := nr + 1
                if (nr == 1)
                        wait( rw );
                signal (mutexR);
                Read_Database ( );
                wait ( mutexR );
                nr := nr – 1;

                if (nr = 0 )
                        signal( rw )
                signal ( mutexR );
        }
}
```

```
process writer(i:=1 to N) {
        while (1) {
                wait ( rw );
                Update_Database ( ); <--- CS
                signal ( rw );
        }
}
```

Called *readers' preference* solution:

If a reader accesses DB then reader & writer arrive at their entry protocol then readers always have preference over writers

Readers-Writers: Ballhausen's Solution

- Readers' Preference isn't fair.
- A continual flow of readers blocks writers from updating the database.
- Ballhausen's solution tackles this:
  - Idea: one reader takes up the same space as all readers reading together.
    - A semaphore `access` is used for readers to enter DB, with a value initially equalling the total number of readers.
    - Every time a reader accesses the DB, the value of `access` is decremented and when one leaves, it is incremented.
  - Writer wants to enter DB:
    - occupies all spaces step by step by waiting for all old readers to leave and blocking entry to new ones.
  - The writer uses a semaphore `mutex` to prevent deadlock between two writers trying to occupy half available space each.

Readers-Writers: Ballhausen's Solution (cont'd)

```
sem mutex = 1;
sem access = m;

void reader ( int i ) {
        while (1) {
                wait( access );

                // ... reading ...

                signal( access );
        }
}
```

```
void writer ( int i ) {
        while (1) {
                wait( mutex );
                for k = 1 to m {
                        wait( access );
                }
                //... writing ...
                for k = 1 to m {
                        signal( access );
                }
                // other operations
                signal( mutex );
        }
}

int main ( ) {
cobegin
        reader (1);reader (2);reader (3);
writer (1); writer (2);
}
```

Monitors: The Sleeping Barber Problem

A small barber shop has two doors, an entrance and an exit.

Inside, barber spends all his life serving customers, one at a time.

1. When there are no customers in the shop, he **sleeps** in his chair.
2. If a customer arrives and finds the barber asleep:
   - he **awakens** the barber,
   - sits in the chair and **sleeps** while hair is being cut.
3. If a customer arrives and the barber is **busy** cutting hair,
   - the customer goes **asleep** in one of the two waiting chairs.
4. When the barber finishes cutting a customer's hair,
   - he **awakens** the customer and holds the exit door open for them.
5. If there are waiting customers,
   - he **awakens** one and **waits** for the customer to sit in the barber's chair,
   - otherwise he sleeps.

Monitors: The Sleeping Barber Problem (cont'd)

- The barber and customers are interacting processes,
- The barber shop is the monitor in which they interact.

Sleeping Barber Using Monitors (cont'd)

- For the Barbershop, the monitor provides an environment for the customers and barber to *rendezvous*
- There are four synchronisation conditions:
  - Customers must wait for barber to be available to get a haircut
  - Customers have to wait for barber to open door for them
  - Barber needs to wait for customers to arrive
  - Barber needs to wait for customer to leave
- Processes
  - wait on conditions using wait()s in loops
  - signal() at points when conditions are true

Monitors: The Sleeping Barber Problem (cont'd)

- Use three counters to synchronize the participants:
  - barber, chair and open (all initialised to zero)
- Variables alternate between zero and one:
  - barber==1 the barber is ready to get another customer
  - chair==1 customer sitting on chair but no cutting yet
  - open==1 exit is open but customer not gone yet
- The following are the synchronization conditions:
  - Customer waits until barber is available
  - Customer remains in chair until barber opens it
  - Barber waits until customer occupies chair
  - Barber waits until customer leaves

Monitors: Sleeping Barbers (cont'd)

```
monitor( barber_shop )
        int barber:=0; int chair=0; int open=0;
        condition( barber_available ) ; // signalled when barber > 0
        condition( chair_occupied ) ; // signalled when chair > 0
        condition( door_open ); // signalled when open > 0
        condition( customer_left ) ; // signalled when open = 0
```

```
void get_haircut() {
        do
                wait( barber_available );
        while( barber==0 )

        barber = barber – 1;
        chair = chair + 1;

        signal( chair_occupied );

        do
                wait( door_open );
        while ( open==0 )

        open = open – 1;

        signal(customer_left);
} // called by customer
```

```
void get_next_customer() {
        barber = barber + 1;
        signal( barber_available );

        do
                wait( chair_occupied );
        while ( chair == 0 )

        chair = chair -1;
} // called by barber

void finished_cut() {
        open := open +1;
        signal( door_open );

        do
                wait( customer_left );
        while (open==0)
        } // called by barber
}
```

Sleeping Barber Using Monitors (cont'd)

```
void customer (i) {
        while (1) {
                get_haircut();
        }
}

void barber(i) {
        while(1) {
                get_next_customer(); // cut hair
                finished_cut();
        }
}

int main() {
        cobegin {
                barber (1);
                customer (1); customer (2);
        }
}
```