Ollscoil na Gaillimhe
University of Galway

Dr Takfarinas Saber
takfarinas.saber@universityofgalway.ie

# CT213 Computing System & Organisation

## Lecture 6: Process Synchronisation

# Concurrent Programming

- Concurrent programs: *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

# Correctness

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.*

Generalisation**:** a program is correct when its preconditions hold then its post conditions will hold.

```
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*.

# Lets Look at this in Practice: Race Conditions

- A **race condition** occurs when a program output is dependent on the sequence or timing of code execution
  - if multiple processes of execution enter a **critical section** at about the same time; both attempt to update the shared data structure
  - ➢leads to surprising results (undesirable)
  - ❖You 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 processes:

- Process 1: withdraw 10 from account
- Process 2: withdraw 20 from account

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

Process 1

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

Process 2

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

Process 1

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

Process 2

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

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

Ollscoil na Gaillimhe
University of Galway

# 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

- We want deterministic concurrent code
- ➢ 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

<u>Idea:</u>

- Focus on critical sections of code
  - i.e., bits that access shared resources
- We want critical sections to run with mutual exclusion
  - ➢ only one process 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?

# Critical Section Properties

- **Mutual exclusion**: only 1 process can access at a time
- **Guarantee of progress**: processes outside the critical section cannot stop another from entering it
- **Bounded waiting**: a process waiting to enter a critical section will eventually enter
  - Processes 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 processes wait much longer than others

# Synchronisation Solutions

Ways to protect critical sections

- Option 1: Atomicity
  - Atomic operations cannot be interrupted, in order to avoid illogical outcomes
- Option 2: Conditional synchronisation (ordering)
  - Making sure that one process runs before another

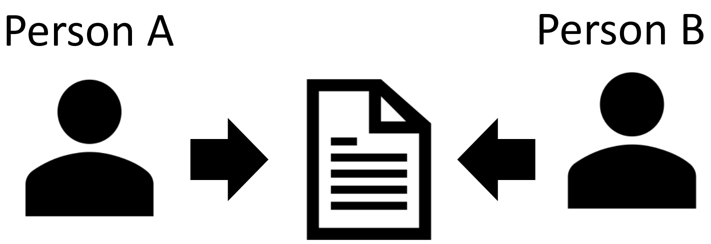
# Atomicity

- Basic atomicity is provided by the hardware
  - E.g., *References and assignments (i.e., read & write operations) are atomic in all CPUs*

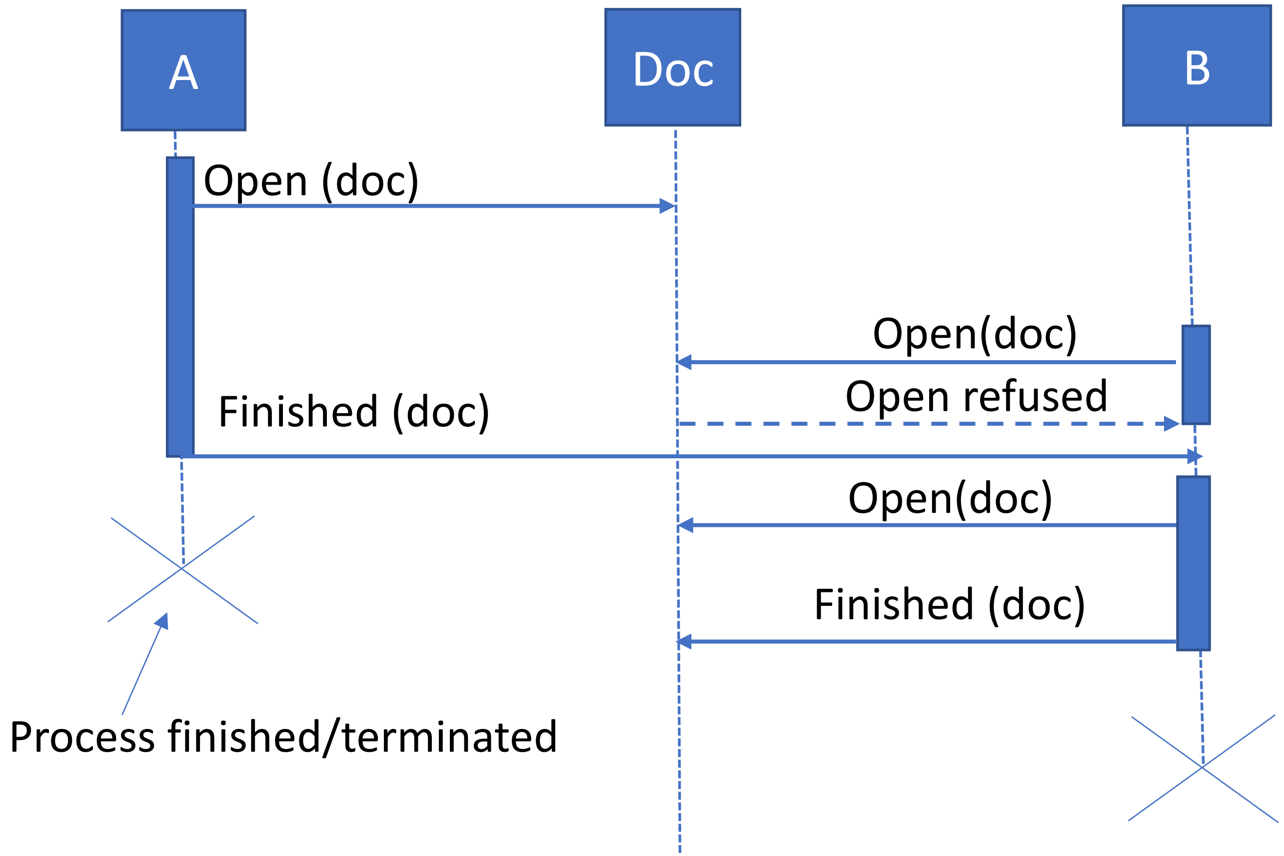
- However higher-level constructs (i.e., any sequence of two or more CPU instructions ) are not atomic in general
- Some languages (e.g., Java) have mechanisms to specify multiple instructions as atomic

# 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 Conditional Synchronisation Look Like?



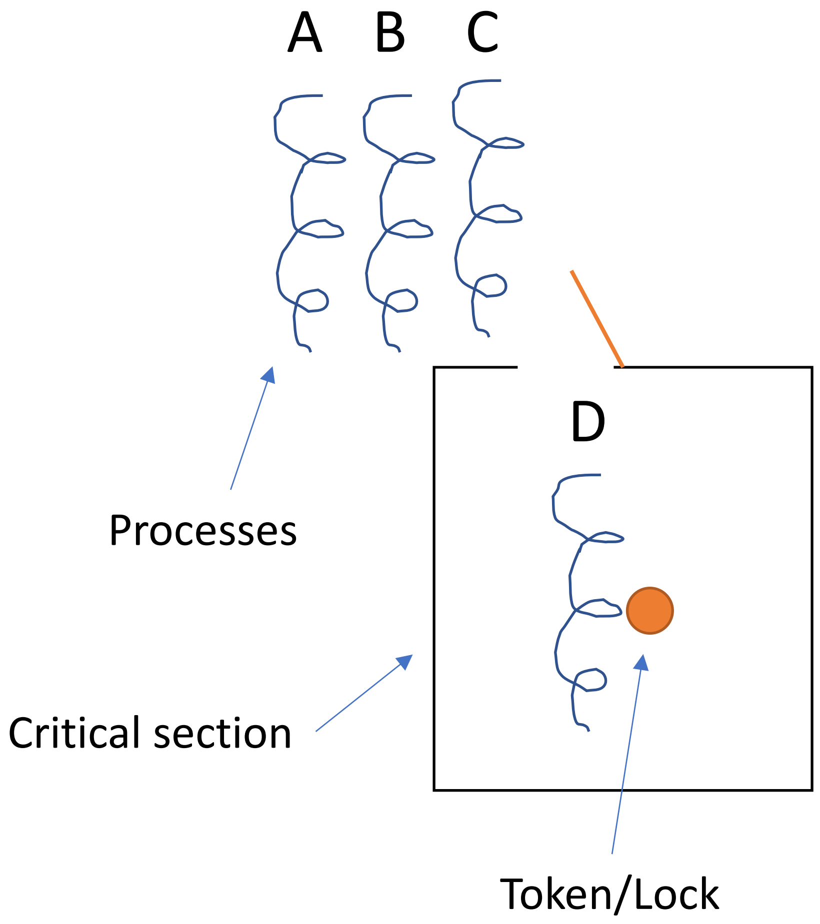
# Code Constructs to Support Defining Critical Sections

- Locks
  - Very primitive, just provide mutual exclusion, minimal semantics, useful as a building block for other methods
- Semaphores
  - Basic, easy to understand
- Monitors
  - Higher level abstraction, requires language support, implicit operations

# Mutual Exclusion solutions: Locks

# Locks: Basic idea

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



# Lock States and Operation

- Locks have 2 states:
  - Held: some process is in the critical section
  - Not held: no process 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 processes call acquire, only 1 process can get 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

    myLock.acquire();

    // Critical_Section

    myLock.release();

    // Non_Critical_Section
end while
```

Surround critical section of code

# Lock Benefits

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

➢Achieves requirements of **mutual exclusion** and **progress** for concurrent systems

# Lock Limitations

- Acquiring a lock *only* blocks processes trying to acquire the *same* lock
  - i.e., processes can acquire other locks
- **Must use the 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., Add a new process 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
```

P1
```
Int withdraw(account, amount = 10){
        acquire(myBalanceLock);
        int balance = account.balance; //100
```

P2
```
Int withdraw(account, amount = 20){
        acquire(myBalanceLock);      // Process STALLED
```

P1
```
balance = balance - amount ; //90
        account.balance = balance; //90
        release(myBalanceLock);     // NOW P2 can start
```

P2
```
        int balance = account.balance; //90
        balance = balance - amount ; //70
        account.balance = balance; //70
        release(myBalanceLock);
        return balance; //70
}
```

P1
```
        return balance; //90
}
```

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

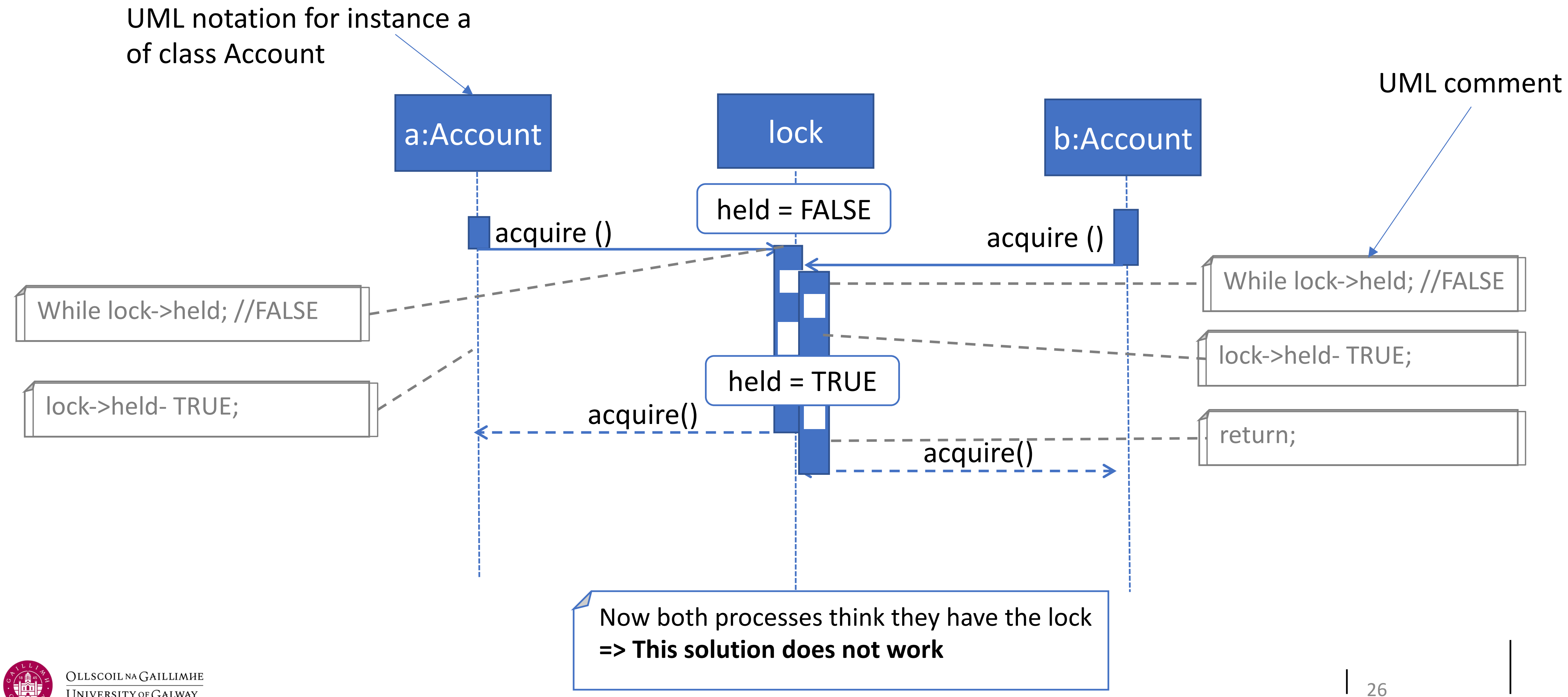
# Impacts

- We can run the processes in any order:
  - We will have the correct final balance
- ➢ We no longer have a race condition

# Software Implementation of Locks (v1)

```
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?



# 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;
}
```

Q: Why is this called a spin lock?

# Drawbacks of Spinlocks

- Spinlocks are a form of busy waiting

  => burn CPU time

- Once acquired they are held until explicitly released
  - What about other processes?
- Inefficient if lock is held for long periods
  - OS overhead of context switching
  - If Process Scheduler makes processes sleep while lock is held
    - ➢ All other processes use their CPU time to spin while the process with the lock makes no progress

# Do Locks give us sufficient safety?

1. **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

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

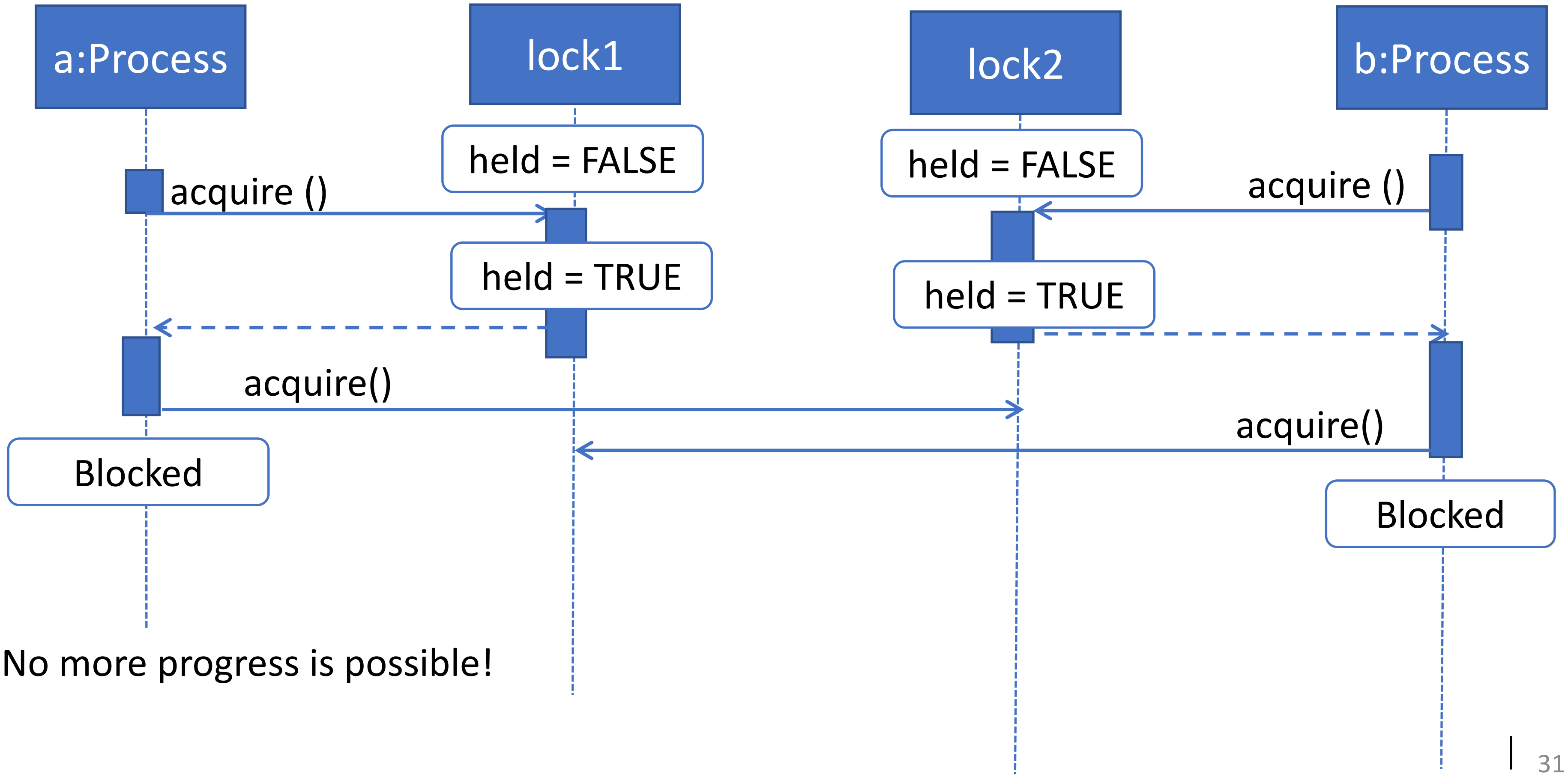
- If you can demonstrate **any** cases in which these properties do not hold

➢ then, **the system is not correct**

Q: What do you think?

# Lock Deadlock Scenario

- 2+ processes, 2 shared resources, 2 locks




No more progress is possible!

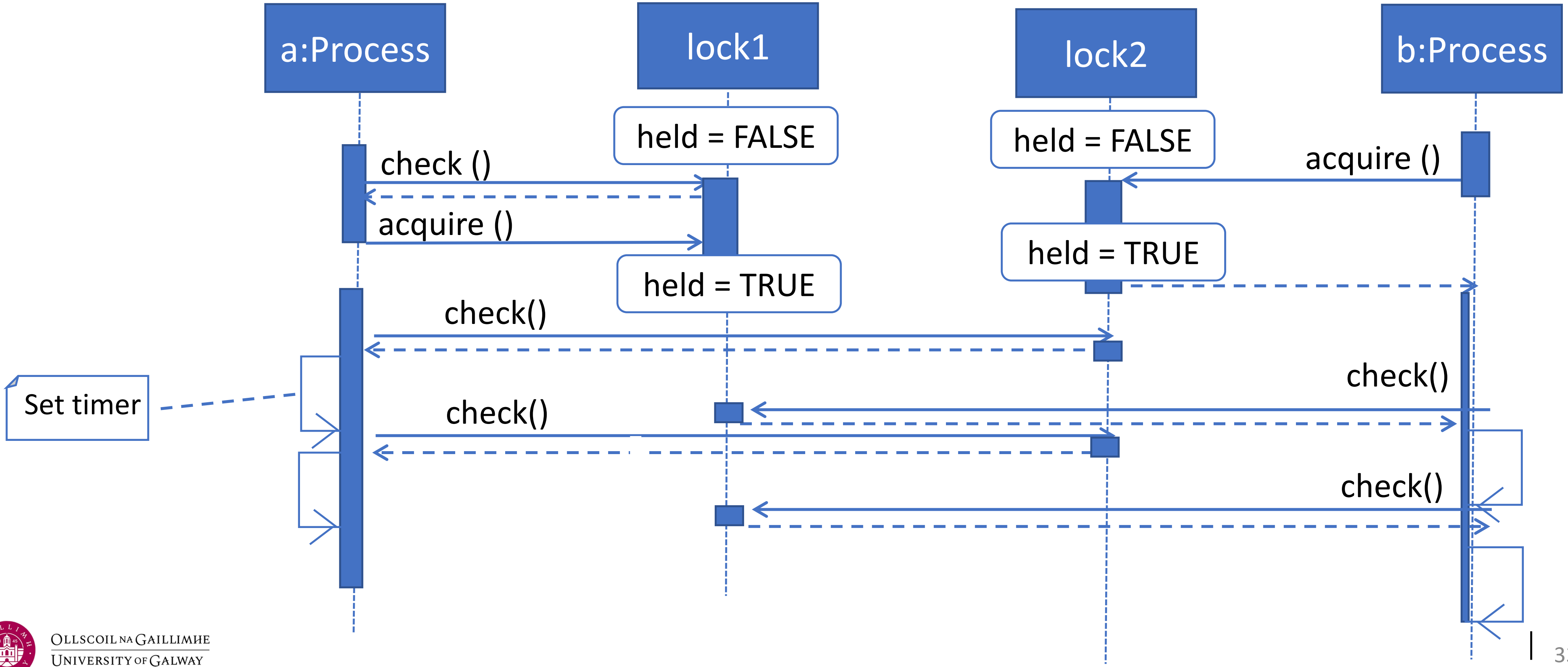
# Protocols to avoid deadlock

- 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

**But these all lead to problems too!**

# Livelock by trying to avoid deadlock

- 2 processes, 2 resources, locks with checking



# Starvation

- More general case of livelock
- 1 or more processes do not get to run as another process is locking the resource
- Example:
  - 2 processes
    - **Process A** runs for 99ms, releases lock for 1ms
    - **Process 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 process 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 process is holding a lock and if it dies no other process waiting on that lock can proceed!

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

# Beyond Locks

- Locks only provide mutual exclusion
  - Ensure only 1 process 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 We want more!
- What about fairness, avoiding starvation, and livelock?
  - ➢We need to be able to place an ordering on the scheduling of processes

# Take Home Message

- Race conditions, deadlock, livelock, fairness, and reliability are all concerns when writing concurrent code
- *Several mechanisms exist to ensure the orderly execution of cooperating processes*

# Higher Level Support for Mutual Exclusion: Semaphores

# Example Scenario: we want to place an order on when processes execute

- Producer- Consumer:
  - Producer: creates a resource (data)
  - Consumer: Uses a resource (data)
  - E.g. `ps | grep "gcc" | wc`
- Don't want producers and consumers to operate in lockstep (i.e., atomicity)
  - Each command must wait for the previous output
  - Implies lots of context switching (i.e., very expensive)
- **Solution:** place a fixed size buffer between producers and consumers
  - Synchronise access to buffer
  - Producer waits of buffer full; consumer waits if buffer empty

# Semaphores

- Semaphore = higher level synchronisation primitive
  - Invented by Dijkstra in 1965 as part of THE OS project
- Semaphores are a kind of generalized lock
  - Main synchronisation primitive used in original UNIX
- Implement with a **counter** that is manipulated atomically via 2 operations **signal** and **wait**

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

decrement counter

if counter is zero then block until semaphore is signalled

`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

`wait()` and `signal()` are critical sections!

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

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

Note: Blocking processes 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 P 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, process goes on queue
  - If first process calls signal()
    - Semaphore value stays at 0
    - Wakes up second process

$\Rightarrow$Acts like a mutex lock

$\Rightarrow$Can use semaphores to implement locks

This is called a **binary semaphore**

# What happens if we initialise to 2?

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

- Sem_init(sem, 2)
  - value=2, L =[]

Consider multiple processes:

- Process1: wait(sem)
  - `value=1,L=[],  P1 executes`
- Process2: wait(sem)
  - `value=0, L[], P2 executes`
- Process3: wait(sem)
  - `value=0, L[P3], P3 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 P from
S.L;
                wakeup(P);
        } else
                s.value = s.value + 1;
}
```

Ollscoil na Gaillimhe
University of Galway

# Uses of Semaphores

- Allocating a number of resources
  - Shared buffers: each time you want to access a buffer, call wait() => you are queued if there is no buffer available
- Counter is initialised to N = number of resources
- Called a **counting semaphore**
- Useful for conditional synchronisation
  - i.e., one process is waiting for another process to finish a piece of work before it continues

# Semaphores for Mutual Exclusion

With semaphores:

- guaranteeing mutual exclusion for $N$ processes is trivial

```
semaphore mutex = 1;

void Process(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 {
          Process(1);   Process(2);
    }
}
```

# Bounded Buffer 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 process, single consumer process
  - 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)

# Semaphores can be Hard to Use

- Complex patterns of resource usage
  - Cannot capture relationships with semaphores alone
  - Need extra state variables to record information

⇒ Produce buggy code that is hard to write

- 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

- *Monitors*: an extension of the monolithic monitor used in OS to allocate memory.
  - A programming language construct that supports controlled access to shared data
  - Synchronisation code added by compiler, enforced at runtime (Less work for programmer!)
- Monitors keep track of <u>who</u> is allowed to access the shared data and <u>when</u> they can do it
- Monitors Encapsulate
  - Shared data structures
  - Procedures that operate on shared data
  - Synchronisation between concurrent processes that invoke these procedures

# Detection and Protection of Deadlock

# Requirements for Deadlock

All 4 conditions must hold for deadlock to occur:

1. **Mutex**: at least one held resource must be non-shareable
2. **No pre-emption:** resources cannot be pre-empted (no way to break priority or take a resource away once allocated
   - 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 $P_1$

Make code more efficient, hence, we want them (conditions 1–3)

Need to avoid circular wait (condition 4)

If only 3 conditions hold then:

- you can get starvation
- but not deadlock

# Sample Deadlock

- Acquire locks in different orders
- Example:

| Process 1 | Process 2 |
| --- | --- |
| 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

- Example:

| <u>Process 1</u> | <u>Process 2</u> |
|---|---|
| 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); |

1. Do we have mutex?
2. Do we have hold and wait?
3. Do we have no pre-emption?
4. Do we have a circular wait?

# Deadlocks without Locks

- Deadlocks can occur for any resource or any time a process waits, e.g.
  - Messages: waiting to receive a message before sending a message
    - i.e., hold and wait
  - Allocation: waiting to allocate resources before freeing another resource
    - i.e., hold and wait

# Testing for Real World Deadlock

- How do cars do it?
  - We have rules to avoid it/recover from it
  - E.g.,
    - Never block an intersection
    - Must backup if you find yourself doing so (a form of pre-emption)
- Why does this work?
  - Breaks a “hold and wait”
  - Shows that refusing to hold a resource while waiting for something else is a key element of avoiding deadlock

# 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 of death and reboot computer
  - Pick a process to terminate, e.g., a low priority one
    - Only works with some types of applications
    - May corrupt data so process needs to do clean-up when terminated

# 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-process variables
  - Counters per process instead of global counter
- Not possible for all bits of code/applications

# Fixing our Sample Deadlock Code

**Original code:**

| <u>Process 1</u> | <u>Process 2</u> |
|---|---|
| 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); |

# Solution 1: Avoid Hold and Wait

Only request a resource when you have none

- I.e., release a resource before requesting another

| Process 1 | Process 2 |
|---|---|
| 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); |

**Original code:**

| Process 1 | Process 2 |
|---|---|
| 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); |

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

Acquire all resources at once

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

| Process 1 | Process 2 |
|---|---|
| **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);** |

**Original code:**

| Process 1 | Process 2 |
|---|---|
| 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); |

Problem: low concurrency

- All processes 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 process A is waiting for a resource held by process 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

- Processes must acquire the highest ranked resource first

| Process 1 | Process 2 |
|---|---|
| lock(x); | **lock(x);** |
| **lock(y);** | lock(y); |
| A=A+10; | B=B+10; |
| B=B+20; | A=A+20; |
| A = A+B; | A=A+B; |
| **unlock(y);** | **unlock(x);** |
| A=A+30; | B=B+30; |
| unlock (x); | unlock(y); |

**Original code:**

| Process 1 | Process 2 |
|---|---|
| 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); |

Locks are always acquired in the same order

- We have eliminated the circular dependency
- Means you will need to lock a resource for a longer period

# 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

- Recall our code for transferring money from 1 account to another

```
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);
}
```

How do we know the global order?

- Need extra code to find this out and then acquire them In the right order
- It could get worse

# Lock Hierarchy Problems

Solution 1.1:

- Order based on hash code of variable

Problem?

- What about same account with the same hash code?

```
transfer(acc1, acc2, amount){
    acc1Hash = hashCode(acc1);
    acc2Hash = hashCode(acc2);
    if (acc1Hash < acc2Hash) {
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
    }else{
        acquire(acc2.a_lock);
        acquire(acc1.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc2.a_lock);
        release(acc1.a_lock);
    }
```

# Lock Hierarchy Problems

Solution 1.2:

- Order based on hash code of the locked variable
- Deal with ties

```
lock tieLock; // a global lock

transfer(acc1, acc2, amount){
    acc1Hash = hashCode(acc1);
    acc2Hash = hashCode(acc2);
    if (acc1Hash < acc2Hash) {
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
    }else if (acc1Hash > acc2Hash) {
        acquire(acc2.a_lock);
        acquire(acc1.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc2.a_lock);
        release(acc1.a_lock);
    } else {
        acquire(tieLock);
        acquire(acc1.a_lock);
        acquire(acc2.a_lock);
        acc1.balance -= amount;
        acc2.balance += amount;
        release(acc1.a_lock);
        release(acc2.a_lock);
        release(tieLock);
    }
}
```

# Extra Resources:

Mike Swift Concurrency videos:

- https://www.youtube.com/channel/UCBRYU9uye8e-ZuWQMPBAoYA/videos