

Concurrent & Distributed Systems

Lecture 1: Introduction to concurrency, threads,
and mutual exclusion.

Michaelmas Term, 2026/27

Dr Martin Kleppmann

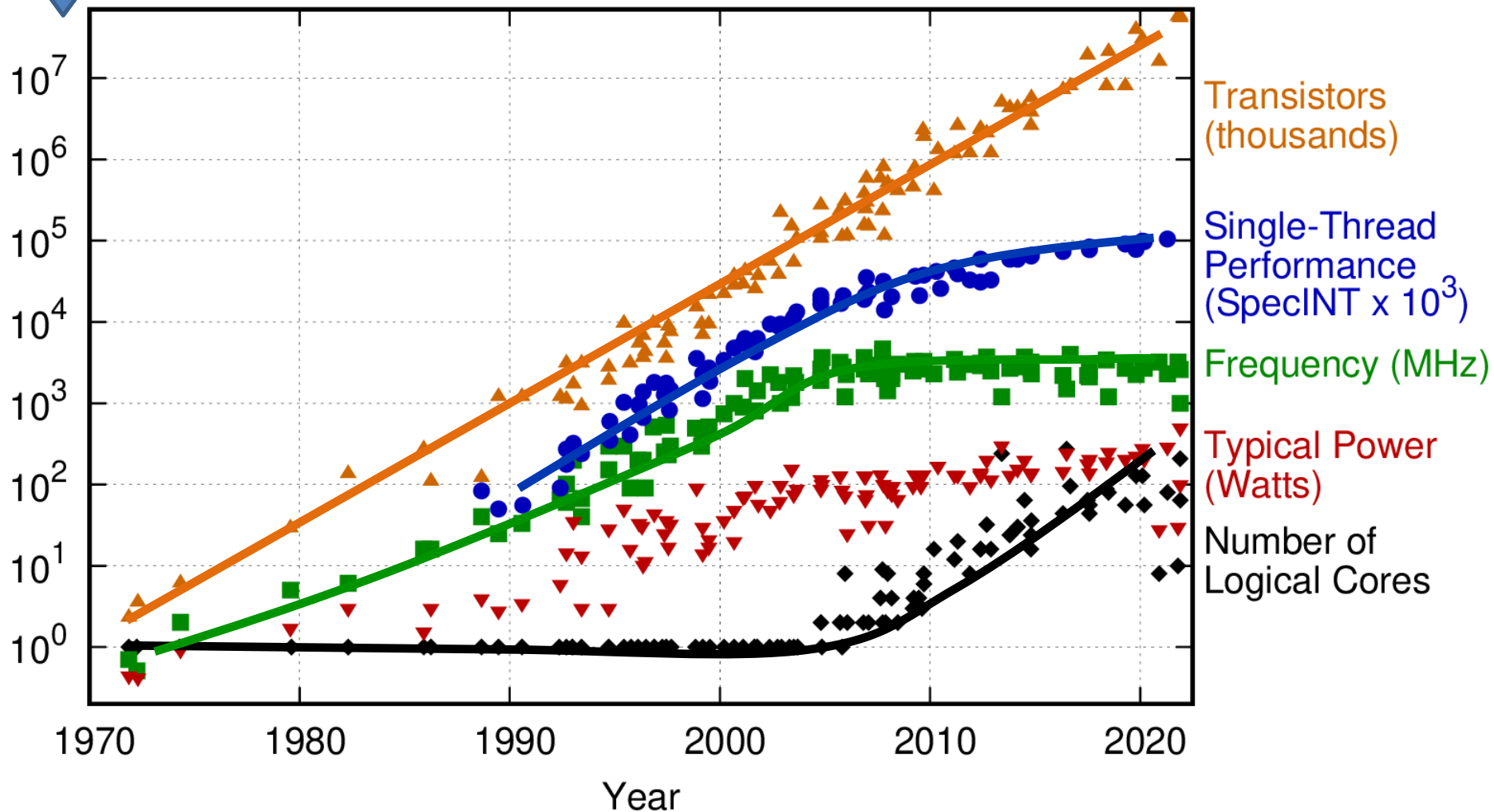
(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

Concurrent and Distributed Systems

- One course, two parts
 - 8 lectures on concurrent systems
 - 8 further lectures of distributed systems
- Similar interests and concerns:
 - **Scalability** given parallelism and distributed systems
 - Mask local or distributed **communications latency**
 - Importance in observing (or enforcing) **execution orders**
 - **Correctness** in the presence of concurrency (+debugging).
- Important differences
 - Underlying primitives: **shared memory** vs. **message passing**
 - Distributed systems experience **communications failure**
 - Distributed systems (may) experience **unbounded latency**
 - (Further) difficulty of **distributed time**.

log scale

50 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

<https://github.com/karlrupp/microprocessor-trend-data>

Concurrent systems outline

1. Introduction to concurrency, threads, and mutual exclusion.
2. Automata composition - safety and liveness.
3. Semaphores and associated design patterns.
4. CCR, monitors and concurrency in programming languages.
5. Deadlock, liveness and priority inversion and limits on parallelism.
6. Concurrency without shared data – message passing, composite operations (transactions).
7. Further transactions.
8. Crash recovery; lock-free programming; (Transactional memory).

Recommended reading

- ***“Operating Systems, Concurrent and Distributed Software Design”*, Jean Bacon and Tim Harris, Addison-Wesley 2003**
- ***“Designing Data-Intensive Applications”*, Martin Kleppmann O’Reilly Media 2017**
- ***“Modern Operating Systems”*, Andrew Tannenbaum, Prentice-Hall 2007 etc and free pdf online.**
- ***“Java Concurrency in Practice”*, Brian Goetz and others, Addison-Wesley 2006**

Look in books for more detailed explanations of algorithms; lectures only present sketches.

What is concurrency?

- Computers appear to do many things at once
 - E.g. running multiple programs on a laptop
 - E.g. writing back data buffered in memory to the hard disk while the program(s) continue to execute
- In the first case, this may actually be an illusion
 - E.g. processes **time sharing** a single-cored CPU
- In the second, there is **true parallelism**
 - E.g. Direct Memory Access (DMA) transfers data between memory and I/O devices (e.g., NIC, SATA) at the same time as the CPU executes code
 - E.g., several CPU cores execute code at the same time
- In both cases, we have a **concurrency**
 - Many things are occurring “at the same time”

In this course we will

- Investigate concurrency in computer systems
 - Processes, threads, interrupts, hardware
- Consider how to control concurrency
 - Mutual exclusion (locks, semaphores), condition synchronization, HLL primitives and lock-free programming
- Learn about deadlock, livelock, priority inversion
 - And prevention, avoidance, detection, recovery
- See how abstraction can provide support for correct & fault-tolerant concurrent execution
 - Transactions, serialisability, concurrency control
- Later, we will extend these ideas to distributed systems.

Recall: Processes and threads

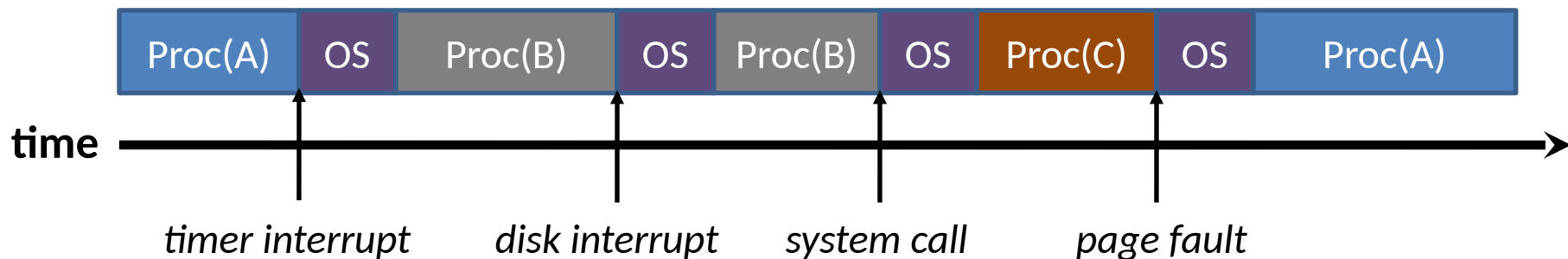
- **Processes** are instances of programs in execution
 - OS unit of protection & resource allocation
 - Has a virtual **address space**; and one or more threads
- **Threads** are entities managed by the **scheduler**
 - Represents an individual execution context
 - A thread control block (TCB) holds the saved context (registers, including stack pointer), scheduler info, etc
- Threads run in the **address spaces** of their process
 - (and also in the kernel address space on behalf of user code)
- **Context switches** occur when the OS saves the state of one thread and restores the state of another
 - If a switch is between threads in different processes, then process state is also switched – e.g., the address space.

Concurrency with a single CPU (1)

- **Process / OS** concurrency
 - Process X runs for a while (until **blocks** or **interrupted**)
 - OS runs for a while (e.g. does some TCP processing)
 - Process X resumes where it left off...
- **Inter-process** concurrency
 - Process X runs for a while; then OS; then Process Y; then OS; then Process Z; etc
- **Intra-process** concurrency
 - Process X has multiple threads X1, X2, X3, ...
 - X1 runs for a while; then X3; then X1; then X2; then ...

Concurrency with a single CPU (2)

- With just one CPU, can think of concurrency as **interleaving** of different executions, e.g.

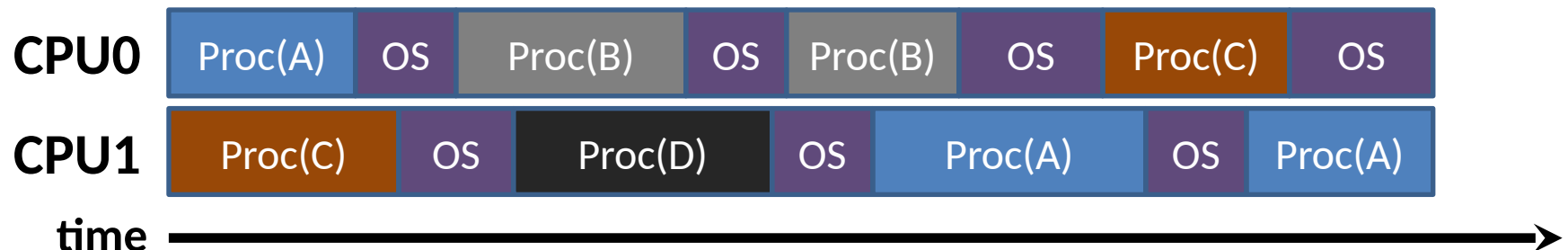


- Exactly where execution is interrupted and resumed is not usually known in advance...
 - this makes concurrency challenging!
- Generally should assume worst case behaviour

Non-deterministic or so complex as to be **unpredictable**

Concurrency with multiple CPUs (aka cores)

- Many modern systems have multiple CPUs
 - And even if don't, have other processing elements.
- Hence things occur in parallel, e.g.



- Notice that the OS runs on both CPUs: tricky!
- More generally, can have different threads of the same process executing on different CPUs too.

What might this code do?

```
#define NUMTHREADS 4
char *threadstr = "Thread";
```

Global variables are shared by all threads

```
void threadfn(int threadnum) {
    sleep(rand(2)); // Sleep 0 or 1 seconds
    printf("%s %d\n", threadstr, threadnum);
}
```

Each thread has its own local variables

```
void main(void) {
    threadid_t threads[NUMTHREADS]; // Thread IDs
    int i; // Counter

    for (i = 0; i < NUMTHREADS; i++)
        threads[i] = thread_create(threadfn, ...);

    for (i = 0; i < NUMTHREADS; i++)
        thread_join(threads[i]);
}
```

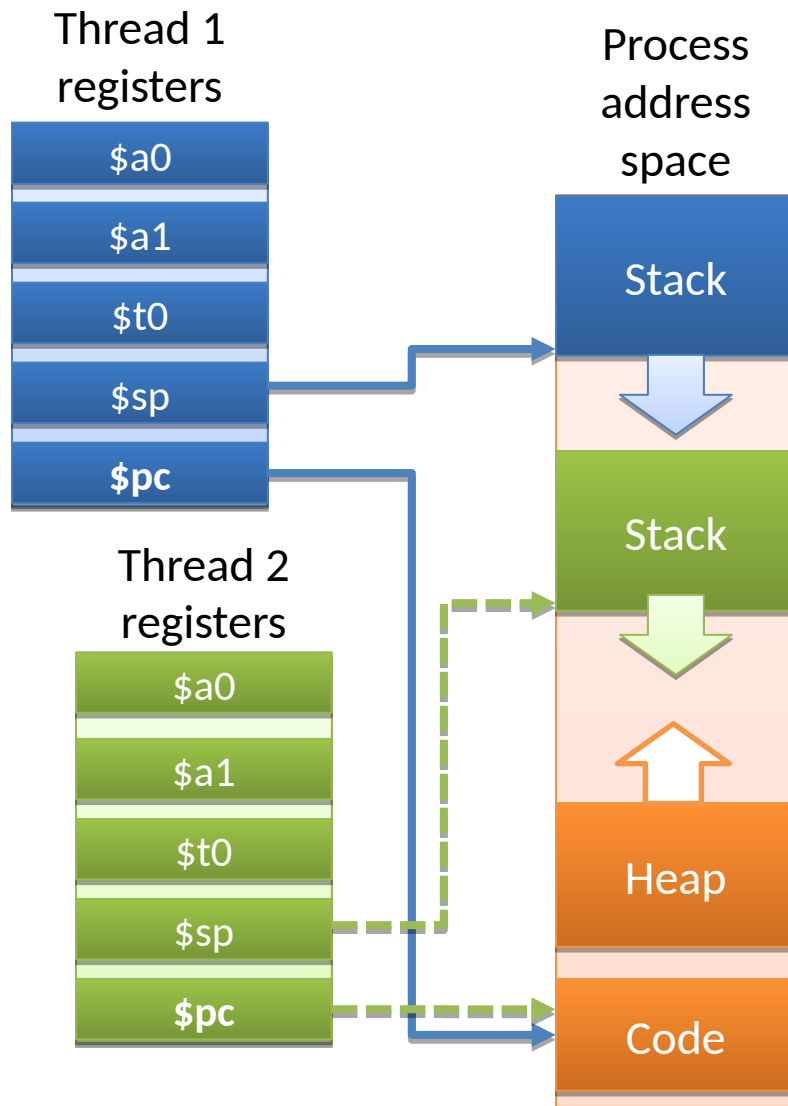
Additional threads are started explicitly

What **orders** could the `printf`s run in?

Possible orderings of this program

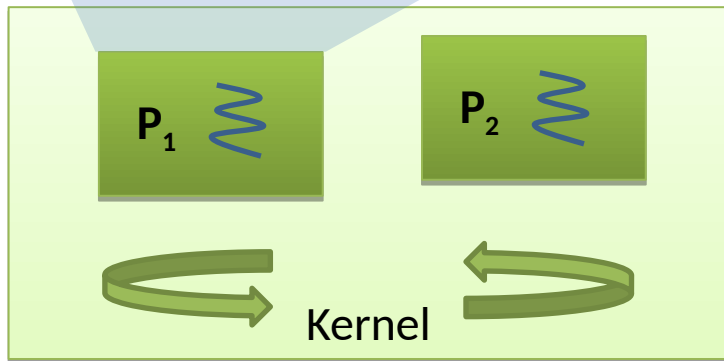
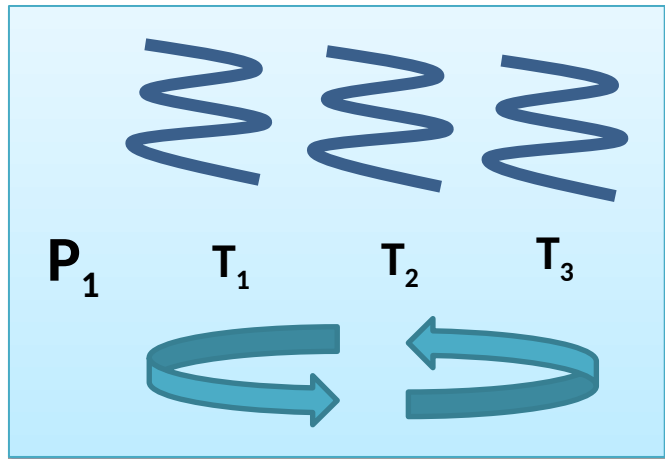
- What order could the `printf()`s occur in?
- Two sources of non-determinism in example:
 - **Program non-determinism**: Threads randomly sleep 0 or 1 seconds before printing
 - **Thread scheduling non-determinism**: Arbitrary order for unprioritised, concurrent wakeups, preemptions
- There are 4! (factorial) valid permutations
 - Assuming `printf()` is **indivisible**
 - Is `printf()` indivisible? Maybe.
- Even more potential **timings** of `printf()`s

Multiple threads within a process



- A single-threaded process has **code**, a **heap**, a **stack**, a **static global segment** and register set (including \$pc).
- **Additional threads** have their own registers and stacks
 - Per-thread **program counters** (\$pc) allow execution flows to differ
 - Per-thread **stack pointers** (\$sp) allow call stacks, local variables to differ
- Heap and code (+**global variables**) are shared between all threads
- Access to another thread's stack is possible in some languages – but deeply discouraged!

1:N - user-level threading

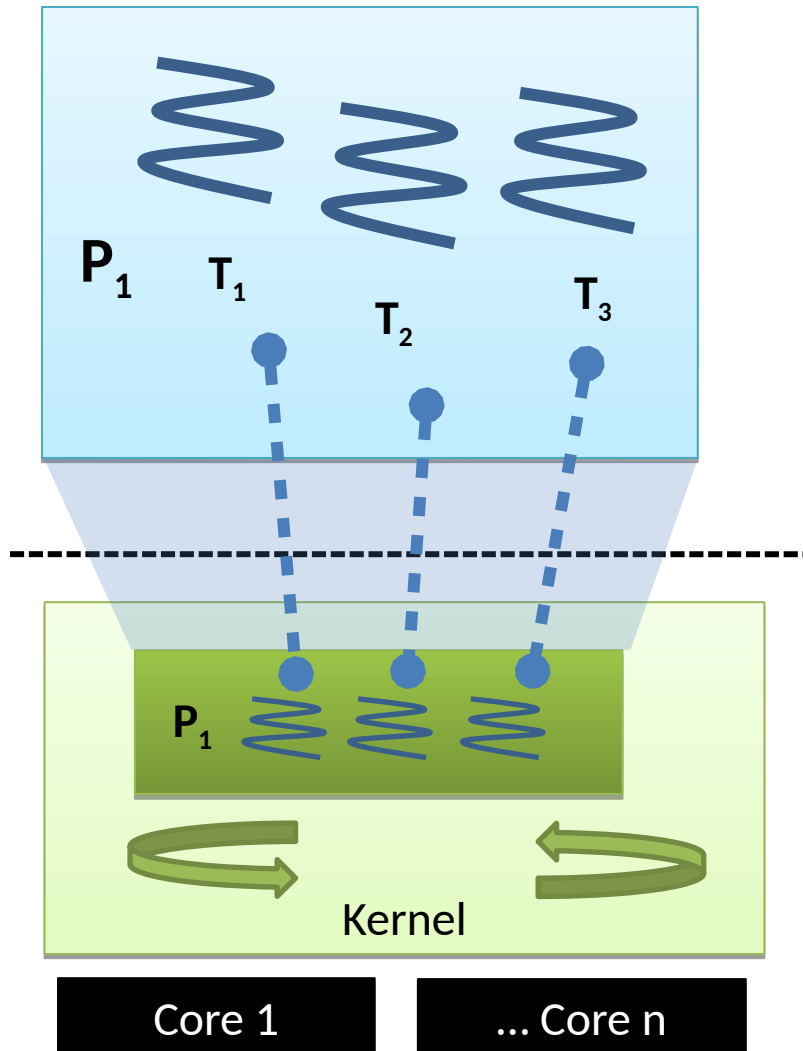


Core 1

... Core n

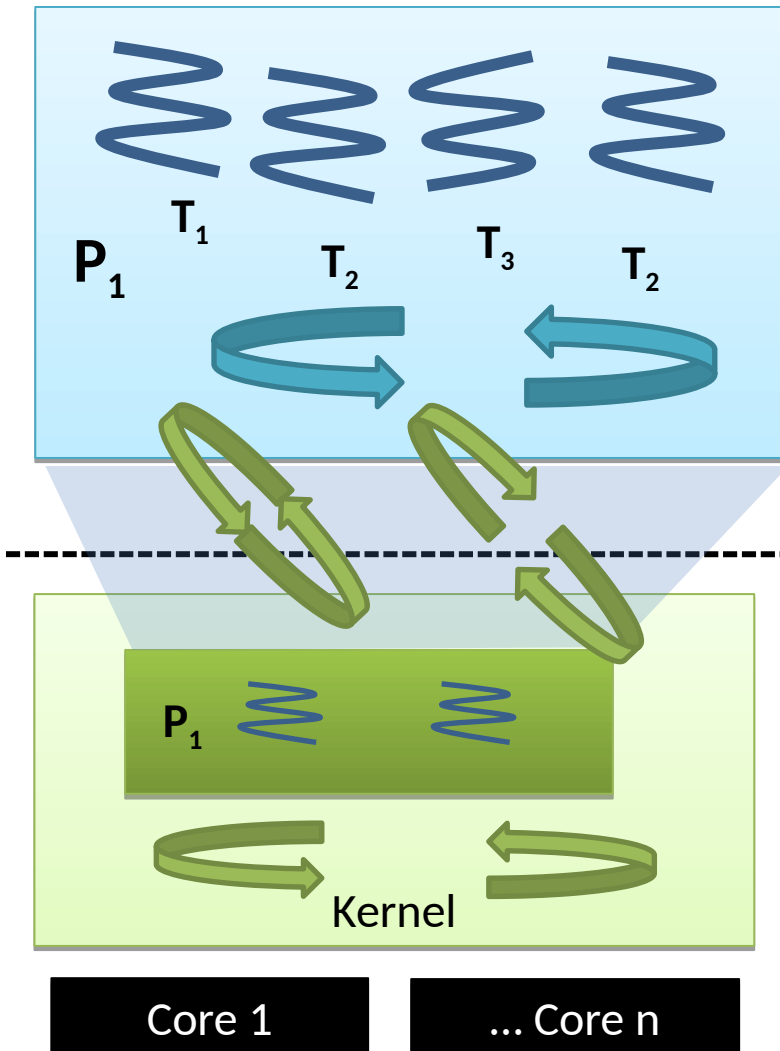
- **Kernel** only knows about (and schedules) processes.
- A **userspace library** implements threads, context switching, scheduling, synchronisation, ...
 - Eg. original JVM or a threading library
- Co-routine variant supports voluntary yield only.
- Advantages:
 - Lightweight creation/termination + context switch; application-specific scheduling; OS independence.
- Disadvantages:
 - Awkward to handle blocking system calls or page faults, preemption; cannot use multiple CPUs.
- Very early 1990s!

1:1 - kernel-level threading



- **Kernel** provides threads directly
 - By default, a process has one thread...
 - ...but can create further via system calls
- Kernel implements threads, thread context switching, scheduling, etc..
- **Userspace** thread 'library' **1:1** maps **user threads** into **kernel threads**
- Advantages:
 - Handles preemption, blocking syscalls,
 - Straightforward to use multiple CPUs.
- Disadvantages:
 - Higher overhead (trap to kernel); less flexible; less portable.
- Model of choice across major OSes
 - Windows, Linux, MacOS, FreeBSD, Solaris, ...

M:N - hybrid threading



- All sorts of other minor variations exist.
- Aim for best of all possible worlds.
- Advantages:
 - Lightweight thread switching entirely in user space is supported.
 - A custom scheduler can understand user-space inter-thread communication primitives (eg. message passing).
- Disadvantages:
 - Need a timer signal (user-space interrupt) to implement time sharing? Perhaps better to just use another kernel thread.
 - Kernel threads are the only ones that can block in a system call, so they are also needed for that, and so on.

Advantages of concurrency

- Allows us to overlap computation and I/O on a single machine.
- Can simplify code structuring and/or improve responsiveness
 - E.g. one thread redraws the GUI, another handles user input, and another computes game logic
 - E.g. one thread per HTTP request
 - E.g. background GC thread in JVM/CLR
- Enables the seamless (?!) use of multiple CPUs –greater performance through parallel processing.

Concurrent systems

- In general, have some number of **processes**...
 - ... each with some number of **threads**,
 - ... each with some number of **CPU cores**,
 - ... distributed over some number of **computers**.
- For this half of the course we'll mostly focus on a single computer running a multi-threaded process
 - most problems & solutions generalize to multiple processes, CPUs, and machines, but imperative programming for them becomes harder
 - (we'll look at distributed systems later in the term)
- Challenge of the thread model: threads will access shared resources concurrently via their common address space leading to races.
- Concurrent programming disciplines without shared memory are generally much 'cleaner' : easier to reason about and automatically map to available cores or other execution resources (GPU, FPGA, Cloud).

Example: Housemates Buying Beer

- Thread 1 (person 1)
 - 1.Look in fridge
 - 2.If no beer, go buy beer
 - 3.Put beer in fridge
- Thread 2 (person 2)
 - 1.Look in fridge
 - 2.If no beer, go buy beer
 - 3.Put beer in fridge
- In most cases, this works just fine...
 - But if both people look (step 1) before either refills the fridge (step 3)... we'll end up with too much beer!
 - Obviously more worrying if “look in fridge” is “check reactor”, and “buy beer” is “toggle safety system” ;-)

Solution #1: Leave a Note

- Thread 1 (person 1)
 - 1.Look in fridge
 - 2.If no beer & no note
 - 1.Leave note on fridge
 - 2.Go buy beer
 - 3.Put beer in fridge
 - 4.Remove note
- Thread 2 (person 2)
 - 1.Look in fridge
 - 2.If no beer & no note
 - 1.Leave note on fridge
 - 2.Go buy beer
 - 3.Put beer in fridge
 - 4.Remove note
- Probably works for human beings...
 - But computers are stooopid!
- Can you see the problem?

Non-Solution #1: Leave a Note

- Easier to see with pseudo-code...

```
// thread 1
beer = checkFridge();
if(!beer) {
    if(!note) {
        note = 1;
        buyBeer();
        note = 0;
    }
}
```

```
// thread 2
beer = checkFridge();
if(!beer) {
    if(!note) {
        note = 1;
        buyBeer();
        note = 0;
    }
}
```

Non-Solution #1: Leave a Note

```
// thread 1
beer = checkFridge();
if(!beer) {
    if(!note) {

        note = 1;
        buyBeer();
        note = 0;
    }
}
```

context switch

context switch

```
// thread 2
beer = checkFridge();
if(!beer) {
    if(!note) {
        note = 1;
        buyBeer();
        note = 0;
    }
}
```

- Easier to see with pseudo-code...

Non-Solution #1: Leave a Note

- Of course this won't happen all the time
 - Need threads to interleave in the just the right way (or just the wrong way ;-).
- Unfortunately code that is 'mostly correct' is much worse than code that is 'mostly wrong'!
 - Difficult to catch in testing, as occurs rarely.
 - May even go away when running under debugger
 - e.g. only context switches threads when they block
 - (such bugs are sometimes called **Heisenbugs**).

Critical Sections & Mutual Exclusion

- The high-level problem here is that we have two threads trying to solve the same problem
 - Both execute `buyBeer()` concurrently
 - Ideally want only one thread doing that at a time.
- We call this code a **critical section**
 - A piece of code which should never be concurrently executed by more than one thread.
- Ensuring this involves **mutual exclusion**
 - If one thread is executing within a critical section, all other threads are prohibited from entering it.

Achieving Mutual Exclusion

- One way is to **let only one thread ever execute a particular critical section** – e.g. a nominated beer buyer – but this restricts concurrency
- Alternatively our (broken) solution #1 was **trying** to provide mutual exclusion via the note
 - Leaving a note means “I’m in the critical section”;
 - Removing the note means “I’m done”
 - But, as we saw, it didn’t work ;-)
- This was because we could experience a context switch between reading ‘note’, and setting it.

Non-Solution #1: Leave a Note

```
// thread 1
beer = checkFridge();
if(!beer) {
    if(!note) {
```

We decide to enter the critical section here...

But only mark the fact here ...

```
        note = 1;
        buyBeer();
        note = 0;
```

```
    }
}
```

context switch

```
// thread 2
```

```
beer = checkFridge();
if(!beer) {
    if(!note) {
        note = 1;
        buyBeer();
        note = 0;
```

context switch

These problems are referred to as **race conditions** in which multiple threads “race” with one another during conflicting access to shared resources

Atomicity

- What we want is for the checking of note and the (conditional) setting of note to happen without any other thread being involved
 - We don't care if another thread reads it after we're done; or sets it before we start our check
 - But once we start our check, we want to continue without any interruption.
- If a sequence of operations (e.g. read-and-set) are made to occur as if one operation, we call them **atomic**
 - Since **indivisible** from the point of view of the program.
- An atomic **read-and-set** operation is sufficient for us to implement a correct beer program.

Solution #2: Atomic Note

```
// thread 1
beer = checkFridge();
if(!beer) {
    if(read-and-set(note)) {
        buyBeer();
        note = 0;
    }
}
```

```
// thread 2
beer = checkFridge();
if(!beer) {
    if(read-and-set(note)) {
        buyBeer();
        note = 0;
    }
}
```

- read-and-set(&address) **atomically** checks the value in memory and iff it is zero, sets it to one
 - returns 1 iff the value was changed from 0 -> 1
- This prevents the behavior we saw before, and is sufficient to implement a correct program...
 - although this is not that program :-)

Non-Solution #2: Atomic Note

```
// thread 1
beer = checkFridge();
if(!beer) {
```

context switch

```
    if(read-and-set(note)) {
        buyBeer();
        note = 0;
    }
}
```

```
// thread 2
```

```
beer = checkFridge();
if(!beer) {
    if(read-and-set(note)) {
        buyBeer();
        note = 0;
    }
}
```

- Our critical section doesn't cover enough!

General mutual exclusion

- We would like the ability to define a region of code as a critical section e.g.

```
// thread 1
ENTER_CS() ;
beer = checkFridge();
if(!beer)
    buyBeer();
LEAVE_CS() ;
```

```
// thread 2
ENTER_CS() ;
beer = checkFridge();
if(!beer)
    buyBeer();
LEAVE_CS() ;
```

- This should work ...
 - ... providing that our implementation of **ENTER_CS()** / **LEAVE_CS()** is correct

Implementing mutual exclusion

- One option is to prevent context switches
 - e.g. disable interrupts (for kernel threads), or set an in-memory flag (for user threads)
 - ENTER_CS() = “disable context switches”;
 - LEAVE_CS() = “re-enable context switches”
- Can work but:
 - Rather brute force (stops all other threads, not just those who want to enter the critical section)
 - Potentially unsafe (if disable interrupts and then sleep waiting for a timer interrupt ;-)
 - And doesn't work across multiple CPUs.

Implementing mutual exclusion

- Associate a **mutual exclusion lock** with each critical section, e.g. a variable **L**
 - (must ensure use correct lock variable!)
 - ENTER_CS() = “LOCK(L)”
 - LEAVE_CS() = “UNLOCK(L)”
- Can implement LOCK() using read-and-set():

```
LOCK(L) {  
    while(!read-and-set(L))  
        continue; // spin, doing nothing  
}
```

```
UNLOCK(L) {  
    L = 0;  
}
```

Solution #3: mutual exclusion locks

```
// thread 1
LOCK(fridgeLock);
beer = checkFridge();
if(!beer)
    buyBeer();
UNLOCK(fridgeLock);
```

```
// thread 2
LOCK(fridgeLock);
beer = checkFridge();
if(!beer)
    buyBeer();
UNLOCK(fridgeLock);
```

- This is – finally! – a correct program
- Still not perfect
 - Lock might be held for quite a long time (e.g. imagine another person wanting to get the milk!)
 - Waiting threads waste CPU time (or worse)
 - **Contention** occurs when consumers have to wait for locks.
- Mutual exclusion locks often known as **mutexes**
 - But we will prefer this term for **sleepable locks** – see Lecture 2
 - So think of the above as a **spin lock**.

Summary + next time

- Definition of a concurrent system
- Origins of concurrency within a computer
- Processes and threads
- Challenge: concurrent access to shared resources
- Critical sections, mutual exclusion, race conditions, atomicity
- Mutual exclusion locks (mutexes)
- Next time:
 - Operating System and hardware instructions and structures,
 - Interacting automata view of concurrency,
 - Introduction to formal modelling of concurrency.

Concurrent systems

Lecture 2: Hardware, OS and Automaton Views

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

From last time ...

- Concurrency exploits parallel and distributed computation.
- Concurrency is also a useful programming paradigm and a virtualisation means.
- Race conditions arise with imperative languages in shared memory (sadly(?) the predominant paradigm of last 15 years).
- Concurrency bugs are hard to anticipate.

This time

- Computer architecture and O/S summary
- Hardware support for **atomicity**
- Basic Automata Theory/Jargon and interactions.
- Simple model checking
- Dining Philosophers Taster
- Primitive-free atomicity (Lamport Bakery)

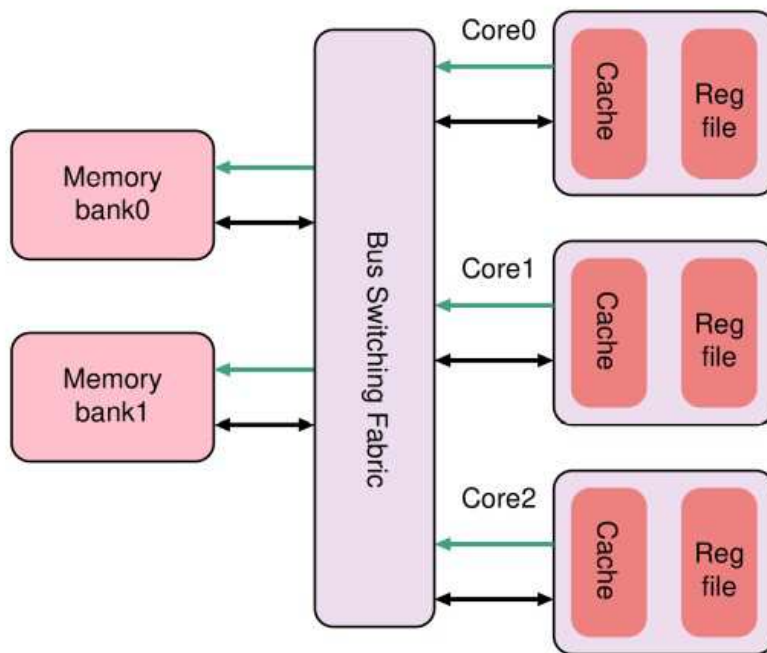
General comments

- Concurrency is essential in modern systems
 - overlapping I/O with computation
 - building distributed systems
 - But throws up a lot of challenges
- need to ensure safety, allow synchronization, and avoid issues of liveness (deadlock, livelock, ...)
- A major risk of over-engineering exists: putting in too many locks not really needed.
- Also its possible to get accidental, excessive serialisation, killing the expected parallel speedup.
- Generally worth building a sequential system first
 - and worth using existing libraries, tools and design patterns rather than rolling your own!

Computer Architecture Reference Models



Single-core, basic computer model (no hardware concurrency).



Multi-core, shared memory, flat-address space computer.

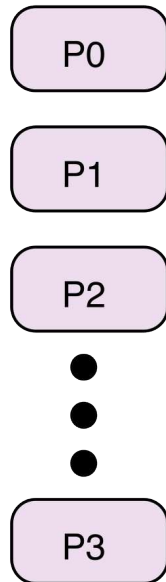
Even on a uniprocessor, interrupt routines will ‘magically’ change stored values in memory.

Stop-the-world atomic operations are undesirable on parallel hardware.

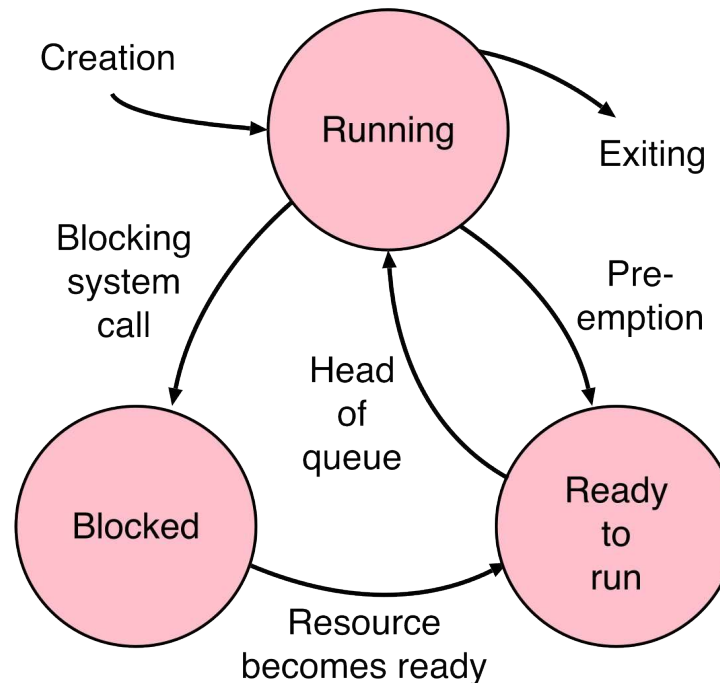
Some of the cores could equally well be DMA controllers.

Operating System Behaviour

Thread/Process control blocks.



Process state diagram



- TCB contains saved registers for non-running tasks.
- Ready-to-run tasks are in a nominal queue.
- Blocked TCBs reference a semaphore (or similar) they are awaiting.
- Most interrupt routines will invoke scheduler as they return.
- If nothing is ready-to-run, the core executes a 'halt' instruction, putting it in low power mode until the next hardware interrupt arrives.

Hardware foundations for atomicity 1

- On a simple uni-processor, without DMA devices, the crudest mechanism is to **disable interrupts**.
- We bracket critical section with `ints_off` and `ints_on` instructions. This guarantees no preemption.
- Can disrupt real-time response
- Not suitable when other CPUs and DMA exist
- Requires supervisor privilege.

Hardware foundations for atomicity 2

- How can we implement **atomic read-and-set**?
- Simple pair of load and store instructions fail the atomicity test (obviously divisible!)
- Need a new **ISA primitive** for protection against parallel access to memory from another CPU
- Two common flavours:
 - Atomic **Compare and Swap** (CAS)
 - **Load Linked, Store Conditional** (LL/SC)
 - (But we also find atomic increment, bitset etc..)

Atomic Compare and Swap (CAS)

- Instruction operands: memory address, prior + new values
 - If prior value **matches** in-memory value, **new value stored**
 - If prior value **does not match** in-memory value, **instruction fails**
 - Software checks return value, can loop on failure
- Found on CISC systems such as x86 (cmpxchg)?

```
    mov    %edx, 1                # New value -> register
spin:
    mov    %eax, [foo_lock] # Load prior value
    test   %eax, %eax          # If non-zero (owned),
    jnz    spin                # loop
lock cmpxchg [foo_lock], %edx # If *foo_lock == %eax,
    test   %eax, %eax          # swap in value from
    jnz    spin                # %edx; else loop
```

“It’s all done with the one instruction” - inner loop is an optimisation; outer is ‘acquire’.

- Atomic **Test and Set** (TAS) is another variation

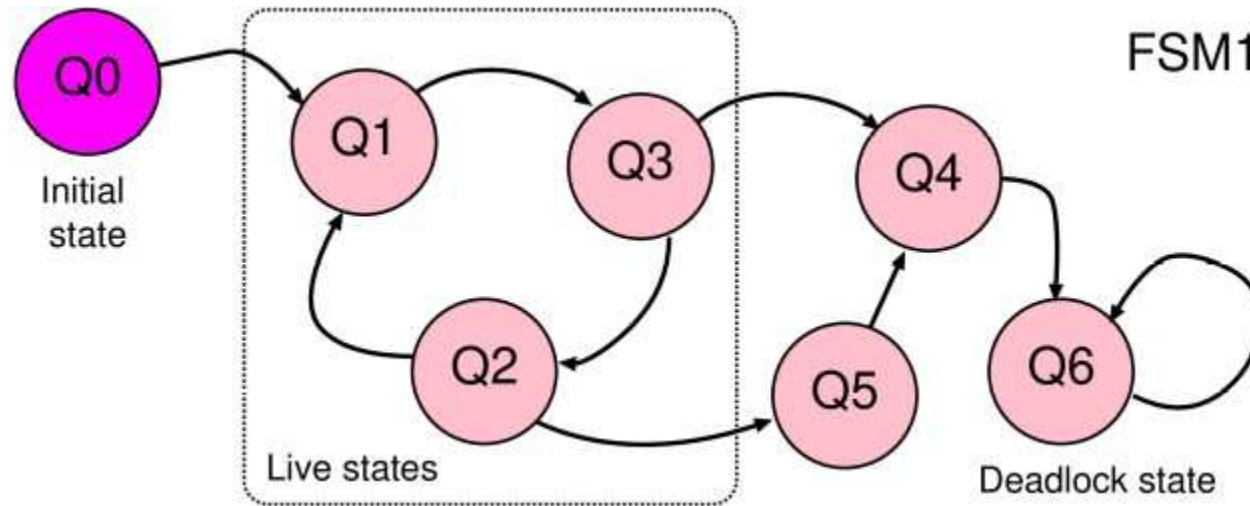
Load Linked-Store Conditional (LL/SC)

- Found on RISC systems (MIPS, RISC-V, ARM, ...)
 - Load value from memory location with **LL**
 - Manipulate value in register (e.g., compare, add, ...)
 - **SC** attempts to write back to same address and indicates success (or not)
 - **SC** fails if memory neighbourhood modified (or interrupt) since **LL**
 - Software checks SC return value and typically loops on failure
 - An example of optimistic concurrency (see later in course).
- Preferred since it does not lock up whole memory system while one core makes an atomic operation.

Code below requires a further outer loop to become an acquire.

```
test_and_set_bit:    ! RISC-V code
spin:
    movli.l          @mutex, %r_tmp1    ! Load linked
    mov              %r_tmp1, %r_tmp2    ! Copy to second register
    or               %r_bitno, %r_tmp1    ! Set the desired bit
    movco.l          %r_tmp1, @mutex     ! Store-conditional
    bf               spin                ! If store failed, try again
    and              %r_bitno, %r_tmp2    ! Return old value of the bit.
    ret
```

Finite State Machine Revision and Terminology



FSM is tuple: (Q, q_0, Σ, Δ) being states, start state, input alphabet, transition function.

A live state is one that can be returned to infinitely often in the future.

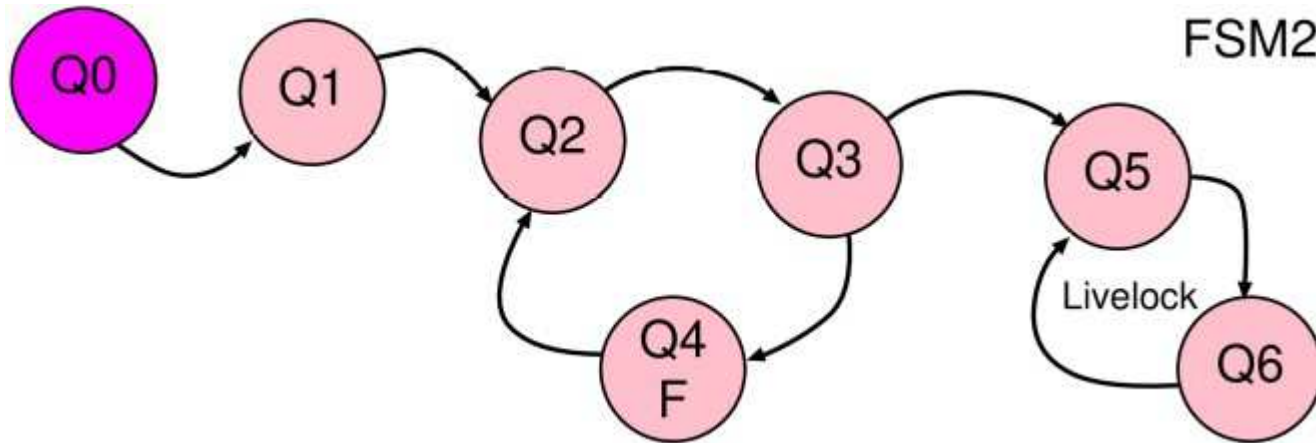
A dead(lock) state has no successors – machine stops if we enter it.

Start-up states are those before the main live behaviour.

‘Bad’ states are those that lead away from the main alive behaviour.

In this course, live states typically encompass/denote the normal/ongoing operation of our system.

Finite State Machine: Fairness and Livelock Syphons



Ignoring the 'F', the live states of this FSM include Q5 and Q6.

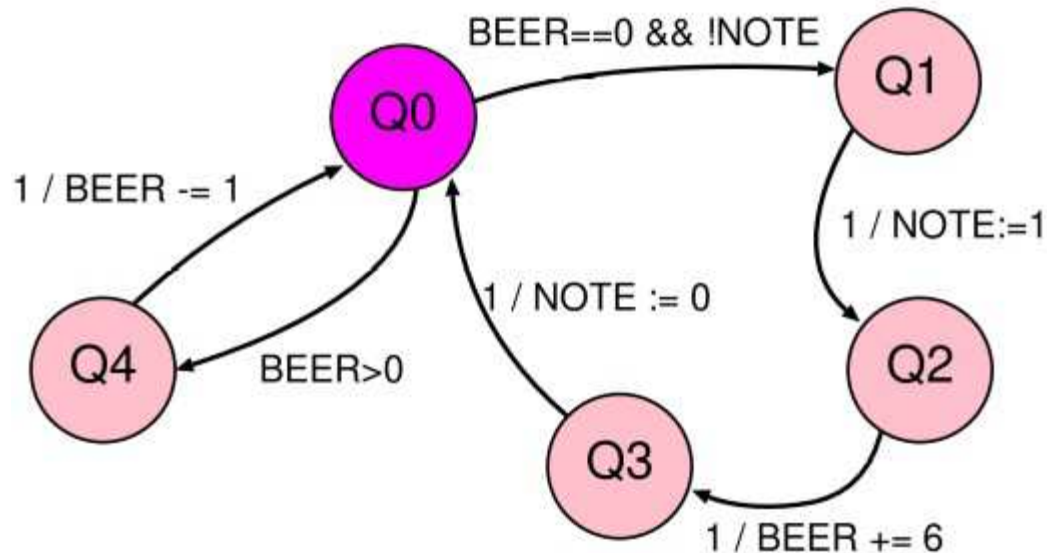
F has been labelled as a 'fair' state. If we also discard the start-up 'lasso stem', its existence changes the live states to just Q2, Q3, Q4. Manual labelling defines the intended system behaviour.

Any fair state is live and states from which any fair state cannot be reached are not live. [Hence if we also labelled Q5 as F, fairness cannot be achieved.]

Although more rigorous definitions exist, this is sufficient terminology for us to define livelock as: we have not deadlocked but cannot make 'useful' progress.

Finite State Machine: FSM view of thread control flow.

Per-thread FSM view of beer drinking and replenishing algorithm



System state vector:

Each person has a program counter: PC of 0..4

Global shared vars: NOTE of Boolean, Beer of 0..99

FSM expresses program control flow per thread.

FSM arcs have 'condition / action' annotations.

Conditions and actions range over shared global state.

Forward reference to semantics course notation

- The semantics course models a computer as a program (expression) e and a memory (store) s ;
- It uses the vertical bar to denote stuttering parallel composition.

$$\text{(parallel1)} \quad \frac{\langle e_1, s \rangle \longrightarrow \langle e'_1, s' \rangle}{\langle e_1 \mid e_2, s \rangle \longrightarrow \langle e'_1 \mid e_2, s' \rangle}$$

$$\text{(parallel2)} \quad \frac{\langle e_2, s \rangle \longrightarrow \langle e'_2, s' \rangle}{\langle e_1 \mid e_2, s \rangle \longrightarrow \langle e_1 \mid e'_2, s' \rangle}$$

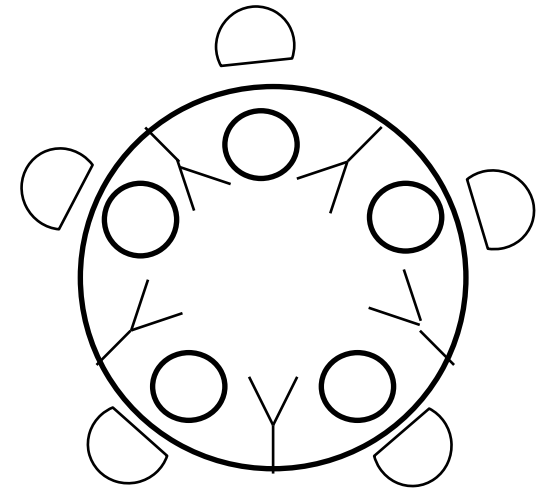
This slide says if either e_1 or e_2 is able to advance, one of them will go forward, updating its PC (e becomes e') and changing the shared memory (s becomes s').

Example: Dining Philosophers

- 5 philosophers, 5 forks, round table...

```
Semaphore forks[] = new Semaphore[5];
```

```
while(true) {           // philosopher i
    think();
    wait(fork[i]);
    wait(fork[(i+1) % 5]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1) % 5]);
}
```



- For now, read 'wait' as 'pick up' and 'signal' as 'put down'
- See next time for definitions.

Summary + next time

- We looked at underlying hardware structures (but this was for completeness rather than for examination purposes)
- We looked at finite-state models of programs and a model checker, but do note that today's tools can cope only with highly-abstracted models or small subsystems of real-world applications.
- Next time
 - Access to hardware primitives via O/S
 - Mutual exclusion using semaphores
 - Producer/consumer and one generalisation

Concurrent systems

Lecture 3: Mutual exclusion, semaphores, and producer-consumer relationships

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

Reminder from last time

- Automata models of concurrent systems
- Concurrency hardware mechanisms

- Challenge: concurrent access to shared resources
- Mutual exclusion, race conditions, and atomicity
- Mutual exclusion locks (mutexes)

From before: beer-buying example

- Thread 1 (person 1)
 1. Look in fridge
 2. If no beer, go buy beer
 3. Put beer in fridge
 - Thread 2 (person 2)
 1. Look in fridge
 2. If no beer, go buy beer
 3. Put beer in fridge
- In most cases, this works just fine...
 - But if both people look (step 1) before either refills the fridge (step 3)... we'll end up with too much beer!
 - Obviously more worrying if “look in fridge” is “check reactor”, and “buy beer” is “toggle safety system” ;-)

We spotted **race conditions** in obvious concurrent implementations.

Ad hoc solutions (e.g., leaving a note) failed.

Even naïve application of atomic operations failed.

Mutexes provide a general mechanism for mutual exclusion.

This time

- Implementing **mutual exclusion**
- Semaphores for mutual exclusion, condition synchronisation, and **resource allocation**
- Two-party and generalised **producer-consumer** relationships

Implementing mutual exclusion

- Associate a mutual exclusion lock with each critical section, e.g. a variable **L**
 - (must ensure use correct lock variable!)
- ENTER_CS() = “LOCK(L)”
LEAVE_CS() = “UNLOCK(L)”
- Can implement LOCK() using read-and-set():

```
LOCK(L) {  
    while(!read-and-set(L))  
        continue; // spin, doing nothing  
}
```

```
UNLOCK(L) {  
    L = 0;  
}
```

Semaphores

- Despite with atomic ops, busy waiting remains inefficient...
 - Lock contention with **spinning**-based solution wastes CPU cycles.
 - Better to sleep until resource available.
- Dijkstra (THE, 1968) proposed **semaphores**
 - New type of variable
 - Initialized once to an integer value (often 0).
- Supports two operations: **wait()** and **signal()**
 - Sometimes called **down()** and **up()**
 - (and originally called **P()** and **V()** ... blurk!).
- Can be used for **mutual exclusion** with sleeping
- Can also be used for **condition synchronisation**
 - Wake up another waiting thread on a **condition** or event
 - E.g. “There is an item available for processing in a queue.”

Semaphore implementation

- Implemented as an integer and a queue

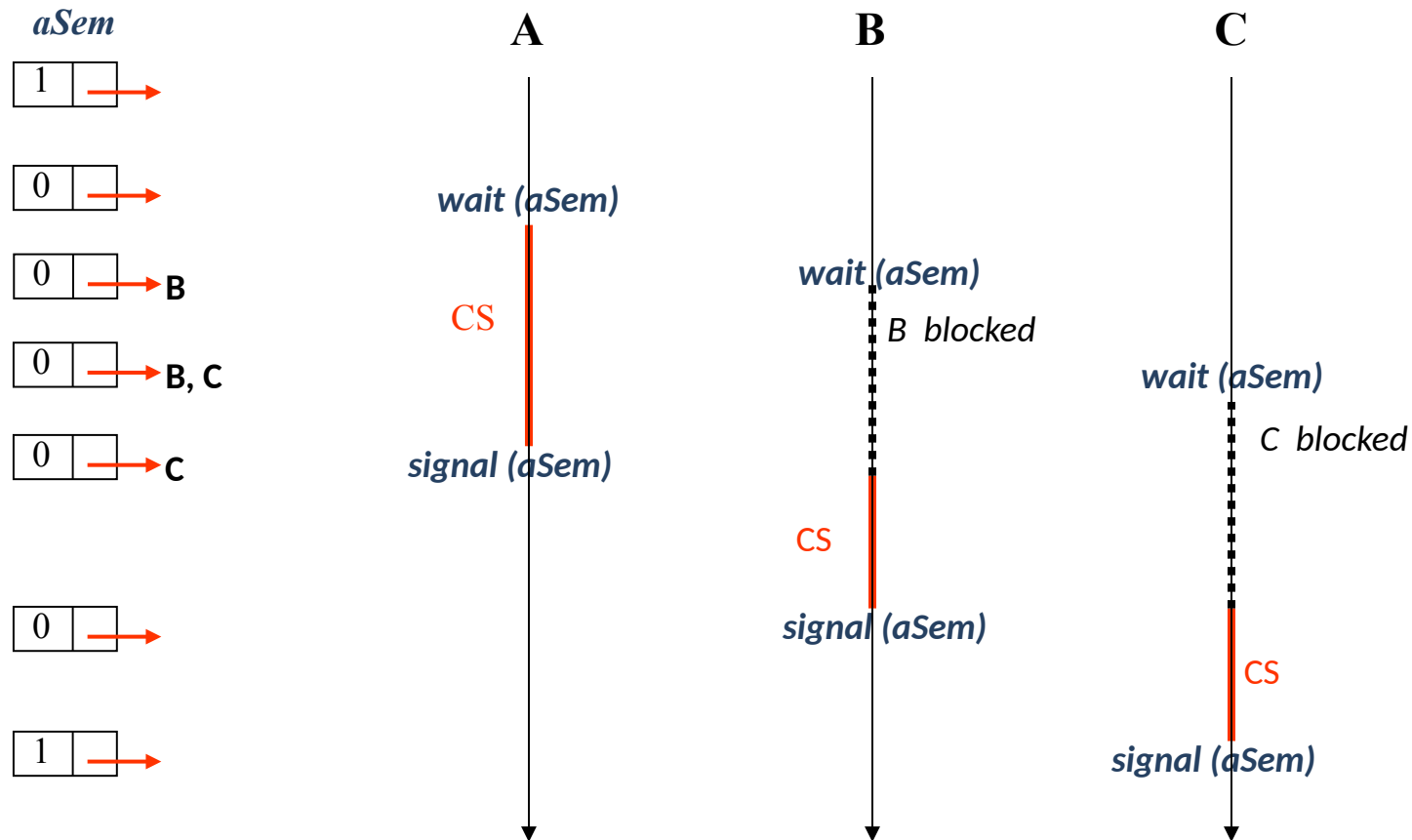
```
wait(sem) {  
    if(sem > 0) {  
        sem = sem - 1;  
    } else suspend caller & add thread to queue for sem  
}  
  
signal(sem) {  
    if no threads are waiting {  
        sem = sem + 1;  
    } else wake up some thread on queue  
}
```

- Method bodies are implemented **atomically**
- Think of “sem” as count of the number of available “items”
- “suspend” and “wake” invoke threading APIs

Hardware support for wakeups: IPIs

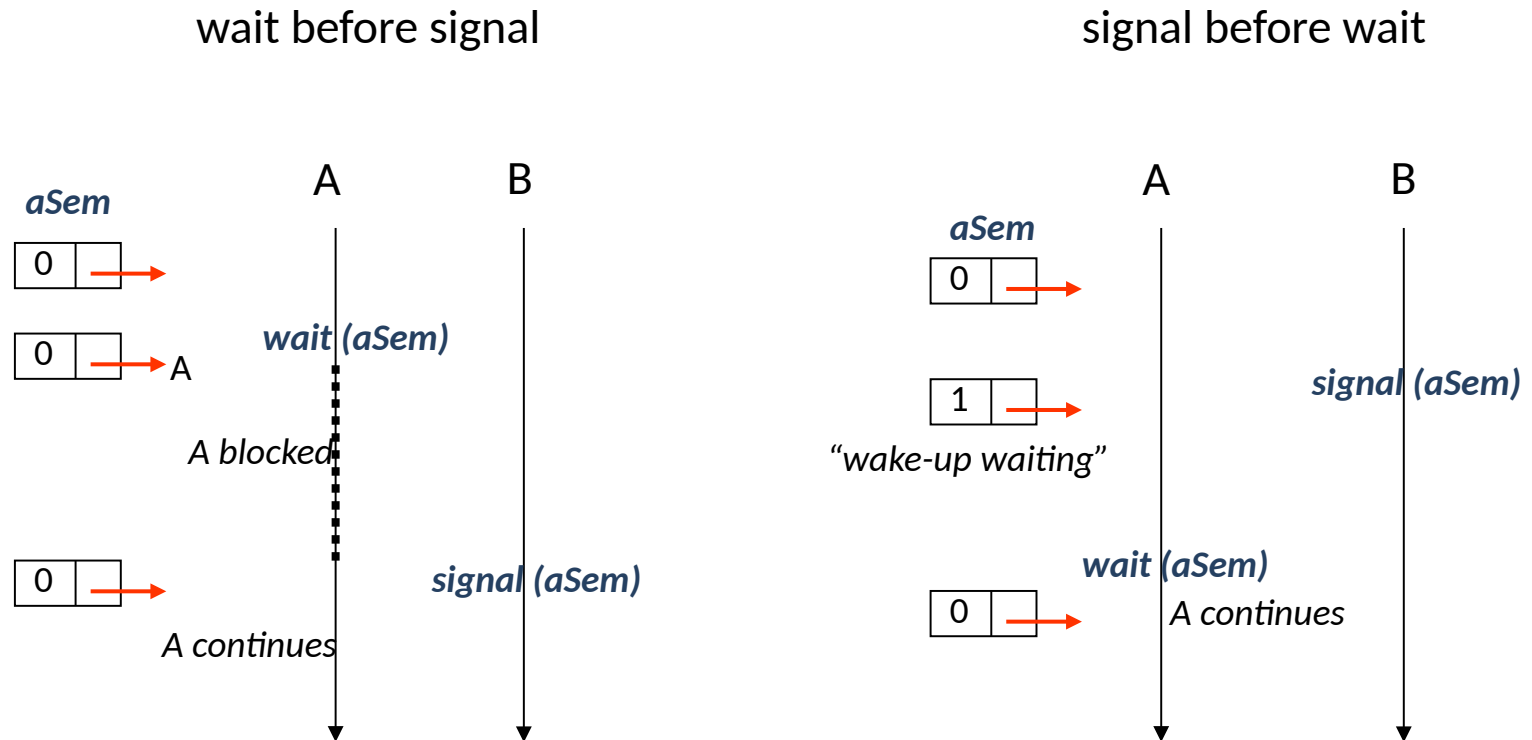
- CAS/LLSC/... support atomicity via shared memory
- But what about “**wake up thread**”?
 - E.g., notify waiter of resources now free, work now waiting, ...
 - Generally known as **condition synchronisation**
 - On a single CPU, wakeup triggers context switch
 - How to wake up a thread on another CPU that is already busy doing something else?
- **Inter-Processor Interrupts (IPIs)**
 - Mark thread as “runnable”
 - Send an interrupt to the target CPU
 - IPI handler runs thread scheduler, preempts running thread, triggers context switch.
- Together, shared memory and IPIs support **atomicity** and **condition synchronisation** between processors.

Mutual exclusion with a semaphore



- Initialize semaphore to 1; **wait()** is lock(), **signal()** is unlock()

Condition synchronisation



- Initialize semaphore to 0; A proceeds only after B signals

N-resource allocation

- Suppose there are **N** instances of a resource
 - e.g. **N** printers attached to a print server daemon.
- Can manage allocation with a semaphore **sem**, initialized to **N**
 - Any job wanting a printer does **wait**(sem)
 - After **N** jobs get a printer, next will sleep
 - To release resource after last page, **signal**(sem)
 - Will wake some job if any job is waiting.
- Will typically also require mutual exclusion
 - E.g. to decide which printers are free and hand them work.

Semaphore design patterns

- Semaphores are quite powerful
 - Can solve **mutual exclusion**...
 - Can also provide **condition synchronization**
 - Thread waits until some condition set by another thread
- Let's look at three common examples:
 - One producer thread, one consumer thread, with a **N**-slot shared memory buffer
 - Any number of producer and consumer threads, again using an **N**-slot shared memory buffer
 - Multiple reader, single writer synchronization (next time)

Producer-consumer problem

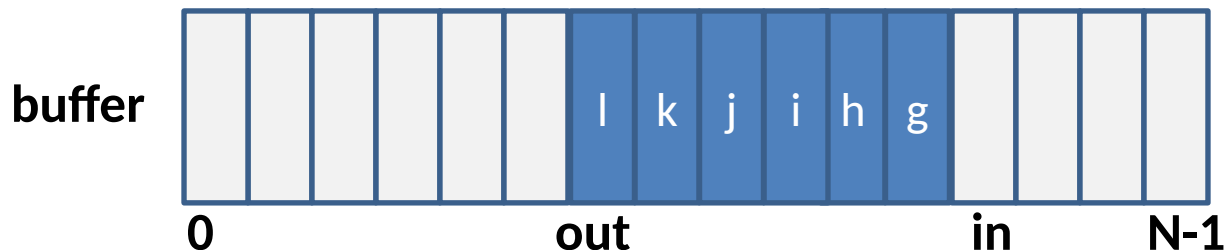
- General “pipe” concurrent programming paradigm
 - E.g. pipelines in Unix; staged servers; work stealing; download thread vs. rendering thread in web browser
- Shared buffer **B[]** with **N** slots, initially empty
- **Producer thread** wants to:
 - Produce an item
 - If there’s room, insert into next slot;
 - Otherwise, wait until there is room (this is called “backpressure”)
- **Consumer thread** wants to:
 - If there’s anything in buffer, remove an item (+consume it)
 - Otherwise, wait until there is something
- Maintain order, use parallelism, avoid context switches.

Producer-consumer pseudo solution

```
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items = new Semaphore(0);
```

```
// producer thread
while(true) {
    item = produce();
    if there is space {
        buffer[in] = item;
        in = (in + 1) % N;
    }
}
```

```
// consumer thread
while(true) {
    if there is an item {
        item = buffer[out];
        out = (out + 1) % N;
    }
    consume(item);
}
```



OO-style producer-consumer FIFO

```
class FIFO<DT> {  
    DT buffer[N]; int in = 0, out = 0;  
    spaces = new Semaphore(N);  
    items  = new Semaphore(0);  
  
    public void enqueue(DT item) { as before ... };  
    public DT dequeue()   { as before ... };  
}
```

```
// producer thread(s)  
while(true) {  
    item = produce();  
    the_fifo.enqueue(item);  
}
```

```
// consumer thread(s)  
while(true) {  
    item = the_fifo.dequeue();  
    consume(item);  
}
```

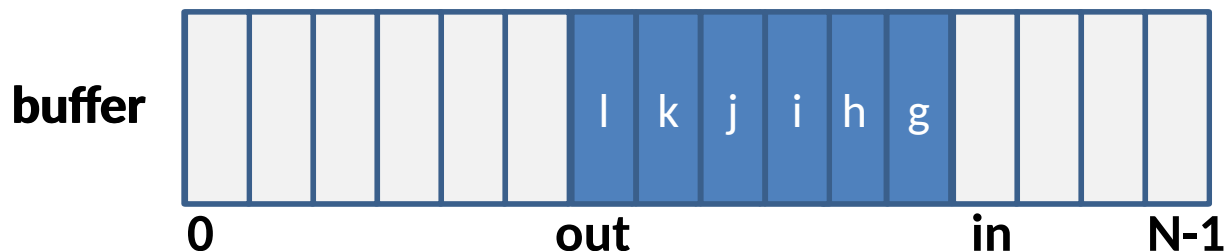
The buffer is often coded/refactored as a class like the above.
The code shape is different, but the executed code is identical.
Note: both exported methods are blocking.
Our method implementations will shortly be generalised to be re-entrant.

Producer-consumer solution

```
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items = new Semaphore(0);
```

```
// producer thread
while(true) {
    item = produce();
    wait(spaces);
    buffer[in] = item;
    in = (in + 1) % N;
    signal(items);
}
```

```
// consumer thread
while(true) {
    wait(items);
    item = buffer[out];
    out = (out + 1) % N;
    signal(spaces);
    consume(item);
}
```



Producer-consumer solution

- Use of semaphores for **N-resource allocation**
 - In this case, **resource** is a slot in the buffer
 - **spaces** allocates empty slots (for producer)
 - **items** allocates full slots (for consumer).
- No **explicit** mutual exclusion
 - Threads will never try to access the same slot at the same time; if “**in == out**” then either
 - **buffer** is empty (and consumer will sleep on **items**), or
 - **buffer** is full (and producer will sleep on **spaces**)
 - NB: **in** and **out** are each accessed solely in one of the producer (**in**) or consumer (**out**).

Generalized producer-consumer

- Previously had **exactly one** producer thread, and **exactly one** consumer thread.
- More generally might have **many threads** adding items, and many removing them
- If so, we **do** need explicit mutual exclusion
 - E.g. to prevent two consumers from trying to remove (and consume) the same item
 - (Race conditions due to concurrent use of **in** or **out** precluded when just one thread on each end)
- Can implement with one more semaphore...

Generalized P-C solution

```
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items = new Semaphore(0);
guard = new Semaphore(1);    // for mutual exclusion
```

```
// producer threads
while(true) {
    item = produce();
    wait(spaces);
    wait(guard);
    buffer[in] = item;
    in = (in + 1) % N;
    signal(guard);
    signal(items);
}
```

```
// consumer threads
while(true) {
    wait(items);
    wait(guard);
    item = buffer[out];
    out = (out + 1) % N;
    signal(guard);
    signal(spaces);
    consume(item);
}
```

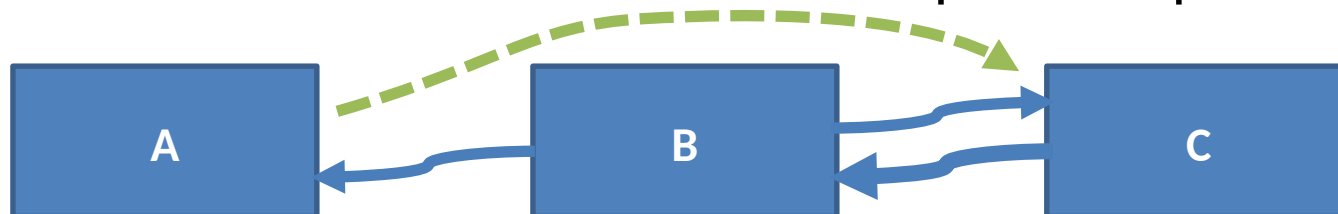
- Exercise: Can we modify this design to allow concurrent access by 1 producer and 1 consumer by adding one further semaphore?

Semaphores: summary

- Powerful abstraction for implementing concurrency control:
 - Mutual exclusion & condition synchronization
- Better than **read-and-set()**... **but** correct use requires considerable care
 - E.g. forget to **wait()**, can corrupt data
 - E.g. forget to **signal()**, can lead to infinite delay
 - Generally get more complex as add more semaphores
- Used internally in some OSes and libraries, but generally deprecated for other mechanisms...

Mutual exclusion and invariants

- One important goal of locking is to avoid exposing **inconsistent intermediate states** to other threads
- This suggests an **invariants**-based strategy:
 - Invariants **hold** as mutex is acquired
 - Invariants **may be violated** while mutex is held
 - Invariants **must be restored** before mutex is released.
- E.g. deletion from a doubly linked list:
 - Invariant: **an entry is in the list, or not in the list.**
 - Individually non-atomic updates of forward and backward pointers around a deleted object are fine as long as the lock isn't released in between the pointer updates



Summary + next time

- Implementing **mutual exclusion**: hardware support for **atomicity** and **inter-processor interrupts**
- Semaphores for mutual exclusion, **condition synchronisation**, and **resource allocation**
- Two-party and generalised **producer-consumer** relationships
- **Invariants** and locks
- Next time:
 - Multi-Reader Single-Writer (MRSW) locks
 - Starvation and fairness
 - Alternatives to semaphores/locks
 - Concurrent primitives in practice

Concurrent systems

Lecture 4: monitors and
concurrency in practice.

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

Reminder from last time

- Implementing **mutual exclusion**: hardware support for **atomicity** and **inter-processor interrupts**
- Semaphores for mutual exclusion, **condition synchronisation**, and **resource allocation**
- Two-party and generalised **producer-consumer** relationships
- **Invariants** and **locks**

From last time: Semaphores summary

- Powerful abstraction for implementing concurrency control:
 - mutual exclusion & condition synchronization
- Better than read-and-set()... **but** correct use requires considerable care
 - e.g. forget to wait(), can corrupt data
 - e.g. forget to signal(), can lead to infinite delay
 - generally get more complex as add more semaphores
- Used internally in some OSes and libraries, but generally deprecated for other mechanisms...

Semaphores are a low-level implementation primitive – they say **what to do**, rather than describing **programming goals**

This time

- **Multi-Reader Single-Writer (MRSW) locks**
 - **Starvation** and **fairness**
- Alternatives to semaphores/locks:
 - **Monitors**
 - **Condition variables**
 - **Signal-and-wait** vs. **signal-and-continue** semantics
- Concurrency primitives in practice
- Concurrency primitives wrap-up

Multiple-Readers Single-Writer (MRSW)

- Another common synchronisation paradigm is MRSW
 - Shared resource accessed by a set of threads
 - e.g. cached set of DNS results
 - Safe for many threads to read simultaneously, but a writer (updating) must have exclusive access
 - MRSW locks have **read lock** and **write lock** operations
 - Mutual exclusion vs. **data stability**
- Simple implementation uses two semaphores
- **First semaphore** is a mutual exclusion lock (**mutex**)
 - Any writer must wait to acquire this
- **Second semaphore** protects a **reader count**
 - Reader count incremented whenever a reader enters
 - Reader count decremented when a reader exits
 - First reader acquires mutex; last reader releases mutex.

Simplest MRSW solution

```
int nr = 0;           // number of readers
rSem    = new Semaphore(1); // protects access to nr
wSem    = new Semaphore(1); // protects writes to data
```

```
// a writer thread
wait(wSem);
.. perform update to data
signal(wSem);
```

Code for writer is simple...

.. but reader case more complex: must track number of readers, and acquire or release overall lock as appropriate

```
// a reader thread
wait(rSem);
nr = nr + 1;
if (nr == 1) // first in
    wait(wSem);
signal(rSem);
.. read data
wait(rSem);
nr = nr - 1;
if (nr == 0) // last out
    signal(wSem);
signal(rSem);
```

Simplest MRSW solution

- Solution on previous slide is “correct”
 - Only one writer will be able to access data structure, but – providing there is no writer – any number of readers can access it
- However writers can **starve**
 - If readers continue to arrive, a writer might wait forever (since readers will not release wSem)
 - Would be **fairer** if a writer only had to wait for all current readers to exit...
 - Can implement this with an additional semaphore.

A fairer MRSW solution

```
int nr = 0;           // number of readers
rSem   = new Semaphore(1); // protects access to nr
wSem   = new Semaphore(1); // protects writes to data
turn   = new Semaphore(1); // write is awaiting a turn
```

Once a writer tries to enter,
it will acquire turn...

... which prevents any further
readers from entering

```
// a writer thread
wait(turn);
wait(wSem);
.. perform update to data
signal(turn);
signal(wSem);
```

```
// a reader thread
wait(turn);
signal(turn);

wait(rSem);
nr = nr + 1;
if (nr == 1) // first in
    wait(wSem);
signal(rSem);
.. read data
wait(rSem);
nr = nr - 1;
if (nr == 0) // last out
    signal(wSem);
signal(rSem);
```

Monitors

- **Monitors** are a more powerful synchronisation primitive
 - Waiting for a condition to become true
 - All related methods are combined together, along with initialization code, in a single construct
- Idea is that only one thread can ever be executing 'within' the monitor
 - If a thread calls a monitor method, it will block (enqueue) if another thread is holding the monitor
 - Hence all methods within the monitor can proceed on the basis that mutual exclusion has been ensured
- Java's **synchronized** primitive implements monitors.

Example Monitor syntax

```
monitor <foo> {
```

```
    // declarations of shared variables
```

```
    // set of procedures (or methods)
```

```
    procedure P1(...) { ... }
```

```
    procedure P2(...) { ... }
```

```
    ...
```

```
    procedure PN(...) { ... }
```

```
    {
```

```
        /* monitor initialization code */
```

```
    }
```

```
}
```

All related data and
methods kept together

Shared variables only
accessible from within
monitor methods

Invoking any procedure
causes an [implicit] mutual
exclusion lock to be taken

Shared variables can be
initialized here

Condition Variables (Queues)

- Mutual exclusion not always sufficient
 - **Condition synchronization** -- e.g., wait for a condition to occur
- Monitors allow **condition variables**
 - Explicitly declared and managed by programmer
 - NB: No integrated counter – not a stateful semaphore!
 - Support three operations:

```
wait(cv) {  
    suspend thread and add it to the queue for CV,  
    release monitor lock;  
}  
signal(cv) {  
    if any threads queued on CV, wake one thread;  
}  
broadcast(cv) {  
    wake all threads queued on CV;  
}
```

Monitor Producer-Consumer solution?

```
monitor ProducerConsumer {  
  int in, out, buffer[N];  
  condition notfull = TRUE, notempty = FALSE;  
  
  procedure produce(item) {  
    if ((in-out) == N) wait(notfull);  
    buffer[in % N] = item;  
    if ((in-out) == 0) signal(notempty);  
    in = in + 1;  
  }  
  
  procedure int consume() {  
    if ((in-out) == 0) wait(notempty);  
    item = buffer[out % N];  
    if ((in-out) == N) signal(notfull);  
    out = out + 1;  
    return(item);  
  }  
  /* init */ { in = out = 0; }  
}
```

If buffer is full,
wait for consumer

If buffer was empty,
signal the consumer

If buffer is empty,
wait for producer

If buffer was full,
signal the producer

Does this work?

- Depends on implementation of **wait()** & **signal()**
- Imagine two threads, **T1** and **T2**, and a condition variable **C**
 - **T1** enters the monitor and calls **wait(C)** – this suspends **T1**, places it on the queue for **C**, and unlocks the monitor
 - Next **T2** enters the monitor, and invokes **signal(C)**
 - Now **T1** is unblocked (i.e. capable of running again)...
 - ... but can only have one thread active inside a monitor!
- If we let **T2** continue (**signal-and-continue**), **T1** must queue for re-entry to the monitor
 - And no guarantee it will be *next* to enter
- Otherwise **T2** must be suspended (**signal-and-wait**), allowing **T1** to continue...

Signal-and-Wait (“Hoare Monitors”)

- Consider the queue **E** to enter the monitor
 - If monitor is occupied, threads are added to **E**
 - May not be FIFO, but should be **fair**.
- If thread **T1** waits on **C**, added to queue **C**
- If **T2** enters monitor & signals, waking **T1**
 - **T2** is added to a new queue **S** “in front of” **E**
 - **T1** continues and eventually exits (or re-waits)
- Some thread on **S** chosen to resume
 - Only admit a thread from **E** when **S** is empty.

Signal-and-Wait pros and cons

- We call **signal()** exactly when condition is true, then directly transfer control to waking thread
 - Hence condition will still be true!
- But more difficult to implement...
- And can be complex to reason about (a call to signal *may or may not* result in a context switch)
 - Hence we must ensure that any invariants are maintained at time we invoke **signal()**
- With these semantics, our example is broken:
 - We **signal()** before incrementing in/out.

Monitor Producer-Consumer solution?

```

monitor ProducerConsumer {
  int in, out, buf[N];
  condition notfull, notempty;

  procedure produce(item) {
    if ((in-out) == N) wait(notfull);
    buffer[in % N] = item;
    if ((in-out) == 0) signal(notempty);
    in = in + 1; Race
  }

  procedure int consume() {
    if ((in-out) == 0) wait(notempty);
    item = buffer[out % N];
    if ((in-out) == N) signal(notfull);
    out = out + 1; Race
    return(item);
  }
  /* init */ { in = out = 0; }
}

```

If buffer is full,
wait for consumer

If buffer was empty,
signal the consumer

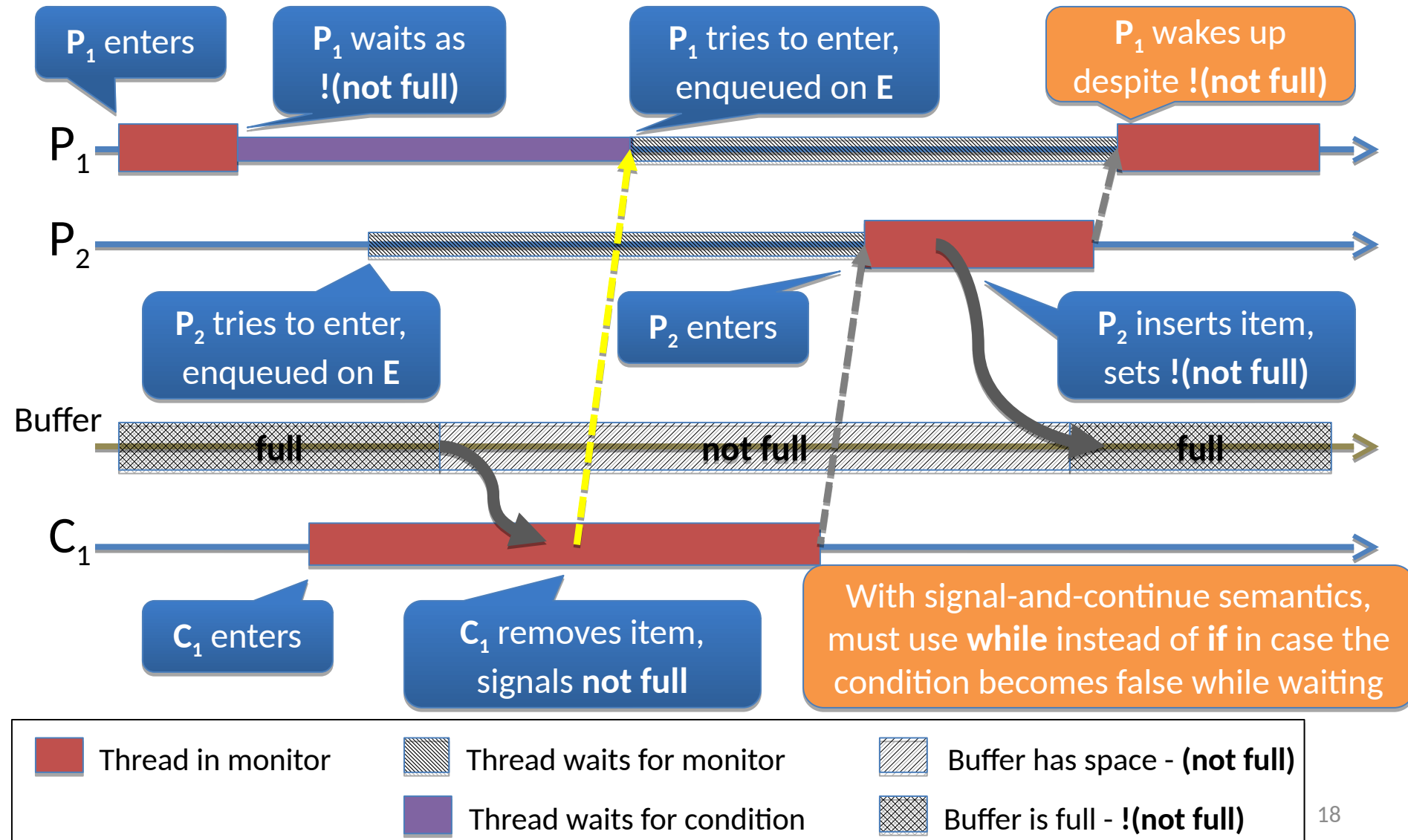
If buffer is empty,
wait for producer

If buffer was full,
signal the producer

Signal-and-Continue

- Alternative semantics introduced by Mesa programming language (Xerox PARC).
- An invocation of **signal()** moves a thread from the condition queue **C** to the entry queue **E**
 - Invoking threads continues until exits (or waits).
- Simpler to build... **but now not guaranteed that condition holds (is true) when resume!**
 - Other threads may have executed after the signal, but before you continue.

Signal-and-Continue example (1)



Signal-and-Continue example (2)

- Consider multiple producer-consumer threads
 1. **P1** enters. Buffer is full so blocks on queue for **C**
 2. **C1** enters.
 3. **P2** tries to enter; occupied, so queues on **E**
 4. **C1** continues, consumes, and signals **C** (“notfull”)
 5. **P1** unblocks; monitor occupied, so queues on **E**
 6. **C1** exits, allowing **P2** to enter
 7. **P2** fills buffer, and exits monitor
 8. **P1** resumes and tries to add item – BUG!
- Hence must *re-test condition*:
i.e. **while**((in - out) == N) wait(not full);

Monitor Producer-Consumer solution?

```
monitor ProducerConsumer {  
  int in, out, buf[N];  
  condition notfull, notempty;  
  
  procedure produce(item) {  
    while ((in-out) == N) wait(notfull);  
    buf[in % N] = item;  
    if ((in-out) == 0) signal(notempty);  
    in = in + 1;  
  }  
  
  procedure int consume() {  
    while ((in-out) == 0) wait(notempty);  
    item = buf[out % N];  
    if ((in-out) == N) signal(notfull);  
    out = out + 1;  
    return(item);  
  }  
  /* init */ { in = out = 0; }  
}
```

While buffer is full,
wait for consumer

If buffer was empty,
signal the consumer

While buffer is empty,
wait for producer

If buffer was full,
signal the producer

With signal-and-continue
semantics, increment after
signal does not race.

Monitors: summary

- Structured concurrency control
 - groups together shared data and methods
 - (today we'd call this object-oriented)
- Considerably simpler than semaphores, but still perilous in places
- May be overly conservative sometimes:
 - e.g. for MRSW cannot have >1 reader in monitor
 - Typically must work around with entry and exit methods (**BeginRead()**, **EndRead()**, **BeginWrite()**, etc)
- Exercise: **sketch a working MRSW monitor implementation.**

Concurrency in practice

- Seen a number of abstractions for concurrency control
 - Mutual exclusion and condition synchronization
- Next let's look at some concrete examples:
 - POSIX pthreads (C/C++ API)
 - FreeBSD kernels
 - Java.

Example: pthreads (1)

- Standard (POSIX) threading API for C, C++, etc
 - mutexes, condition variables, and barriers
- Mutexes are essentially binary semaphores:

```
int pthread_mutex_init(pthread_mutex_t *mutex, ...);  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- A thread calling **lock()** blocks if the mutex is held
 - **trylock()** is a non-blocking variant: returns immediately; returns 0 if lock acquired, or non-zero if not.

Example: pthreads (2)

- Condition variables are Mesa-style:

```
int pthread_cond_init(pthread_cond_t *cond, ...);  
int pthread_cond_wait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex);  
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- No proper monitors: must manually code e.g.

```
pthread_mutex_lock(&M);  
while (!condition)  
    pthread_cond_wait(&C, &M);  
// do stuff  
if (condition)  
    pthread_cond_broadcast(&C);  
pthread_mutex_unlock(&M);
```

Notice: **while()** and not **if()** due to
signal-and-continue semantics

Example: pthreads (3)

- **Barriers:** explicit synchronization mechanism
 - Wait until all threads reach some point
- E.g., in discrete event simulation, all parallel threads must complete one epoch before any begin on the next

```
int pthread_barrier_init(pthread_barrier_t *b, ..., N);  
int pthread_barrier_wait(pthread_barrier_t *b);
```

```
pthread_barrier_init(&B, ..., NTHREADS);  
for(i=0; i<NTHREADS; i++)  
    pthread_create(..., worker, ...);  
  
worker() {  
    while(!done) {  
        // do work for this round  
        pthread_barrier_wait(&B);  
    }  
}
```

Example: FreeBSD kernel

- Kernel provides spin locks, mutexes, conditional variables, reader-writer + read-mostly locks
 - Semantics (roughly) modelled on POSIX threads
- A variety of **deferred work primitives**
- “Fully preemptive” and highly threaded
 - (e.g., interrupt processing in threads)
 - Interesting debugging tools
 - such as DTrace, **lock**
 - **contention measurement**,
- **lock-order checking**
- Further details are in 2019’s lecture 8 ...



For modern C++ support, see <https://en.cppreference.com/w/cpp/thread>

Example: Java synchronization (1)

- Inspired by monitors – objects have **intrinsic locks**
- **Synchronized methods:**

```
public synchronized void myMethod() throws ...{  
    // This code runs with the intrinsic lock held.  
}
```

- Method return / statement exit release lock.
- **Synchronized statements:**

```
public void myMethod() throws ...{  
    synchronized(this) {  
        // This code runs with the intrinsic lock held.  
    }  
}
```

- Locks are **reentrant**: a single thread can re-enter **synchronized** statements/methods without waiting.
- **synchronized()** can accept other objects than **this**.

Example: Java synchronization (2)

- Objects have **condition variables** for **guarded blocks**
- **wait()** puts the thread to sleep:

```
public synchronized void waitDone() {  
    while (!done) {  
        wait();  
    }  
}
```

- **notify()** and **notifyAll()** wake threads up:

```
public synchronized void notifyDone() {  
    done = true;  
    notifyAll();  
}
```

- As with Mesa, **signal-and-continue semantics**
- As with locks, can name object (**thatObject.wait()**)

Example: Java synchronization (3)

- Java also specifies **memory consistency** and **atomicity properties** that make some **lock-free** concurrent access safe – if used **very** carefully
 - We will consider lock-free structures later in the term
- **java.util.concurrent** (especially as of Java 8) includes many higher-level primitives –for example, **thread pools**, **concurrent collections**, **semaphores**, **cyclic barriers**, ...
- Because Java is a type-safe, managed language, it is a much safer place to experiment with concurrent programming than (for example) C.

Concurrency Primitives: Summary

- Concurrent systems require means to ensure:
 - **Safety** (mutual exclusion in critical sections), and
 - **Progress** (condition synchronization)
- Spinlocks (busy wait); semaphores; MRSWs, CCRs, and monitors
 - Signal-and-Wait vs. Signal-and-Continue
- Many of these are used in practice
 - Subtle minor differences can be dangerous
 - Much care required to avoid bugs, **especially where concurrency is a bolt-on** to an existing imperative language.
 - E.g., failing to take out a lock or failing to release it,
 - E.g., “lost wakeups” – signal w/o waiter.

Summary + next time

- **Multi-Reader Single-Writer (MRSW)** locks
- Alternatives to semaphores/locks:
 - **Conditional critical regions (CCRs)**
 - **Monitors**
 - **Condition variables**
 - **Signal-and-wait** vs. **signal-and-continue** semantics
- Concurrency primitives in practice
- Concurrency primitives wrap-up
- Next time:
 - Problems with concurrency: deadlock, livelock, priorities
 - Resource allocation graphs; deadlock {prevention, detection, recovery}
 - Priority and scheduling; priority inversion; (auto) parallelism limits.

Concurrent systems

Lecture 5: Liveness and Priority Guarantees

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

Reminder from last time

- **Multi-Reader Single-Writer (MRSW)** locks
- Alternatives to semaphores/locks:
 - **Conditional critical regions (CCRs)**
 - **Monitors**
 - **Condition variables**
 - **Signal-and-wait** vs. **signal-and-continue** semantics
- Concurrency primitives in practice
- Concurrency primitives wrap-up

From last time: primitives summary

- Concurrent systems require means to ensure:
 - **Safety** (mutual exclusion in critical sections), and
 - **Progress** (condition synchronization)
- Spinlocks (busy wait); semaphores; CCRs and monitors
 - Hardware primitives for synchronisation
 - Signal-and-Wait vs. Signal-and-Continue
- Many of these are still used in practice
 - Subtle minor differences can be dangerous
 - Require care to avoid bugs – e.g., “lost wakeups”
- More detail on implementation in additional material on web page.

Progress is particularly difficult, in large part because of primitives themselves, which is the topic of this lecture

This time

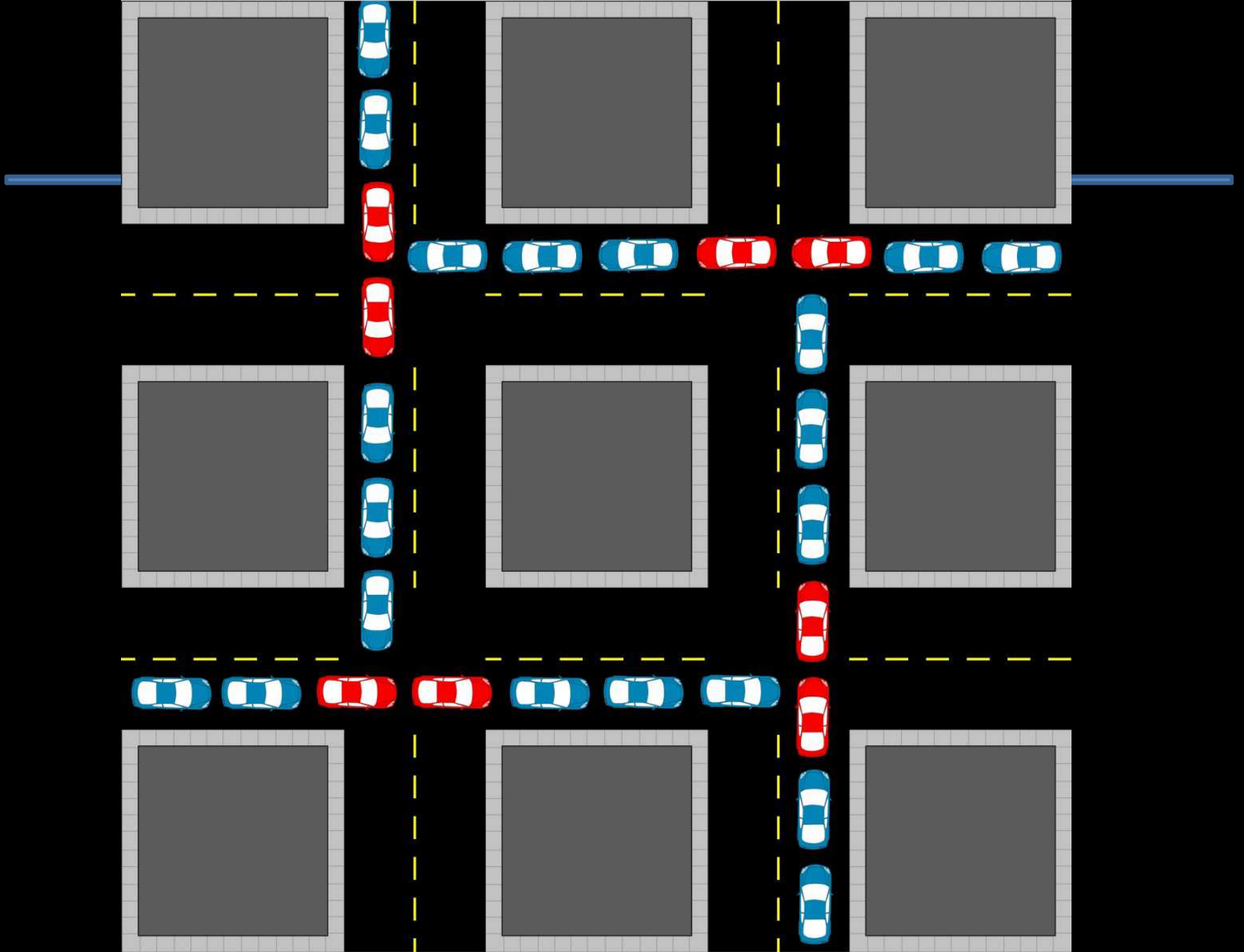
- **Liveness properties**
- **Deadlock**
 - Requirements
 - Resource allocation graphs and detection
 - Prevention – the **Dining Philosophers Problem** – and recovery
- **Thread priority** and the **scheduling problem**
- **Priority inversion** and **priority inheritance**
- **Limits to parallelisation and automation.**

Liveness properties

- From a theoretical viewpoint must ensure that we eventually make progress, i.e. want to avoid
 - **Deadlock** (threads sleep waiting for one another), and
 - **Livelock** (threads execute but make no progress)
- Practically speaking, also want good performance
 - **No starvation** (single thread must make progress)
 - (more generally may aim for **fairness**)
 - **Minimality** (no unnecessary waiting or signalling)
- The properties are often at odds with safety :-)



https://www.reddit.com/r/pics/comments/6qulze/traffic_deadlock/



(Compositional) Deadlock

- Set of **k** threads go asleep and cannot wake up
 - each can only be woken by another who's asleep!
- Real-life example (Kansas, 1920s):

“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”
- In concurrent programs, tends to involve the taking of mutual exclusion locks, e.g.:

```
// thread 1  
lock (X) ;
```

```
...
```

```
lock (Y) ;  
// critical section  
unlock (Y) ;
```

```
// thread 2  
lock (Y) ;
```

```
...
```

```
if (<cond>) {  
    lock (X) ;  
    ...  
}
```

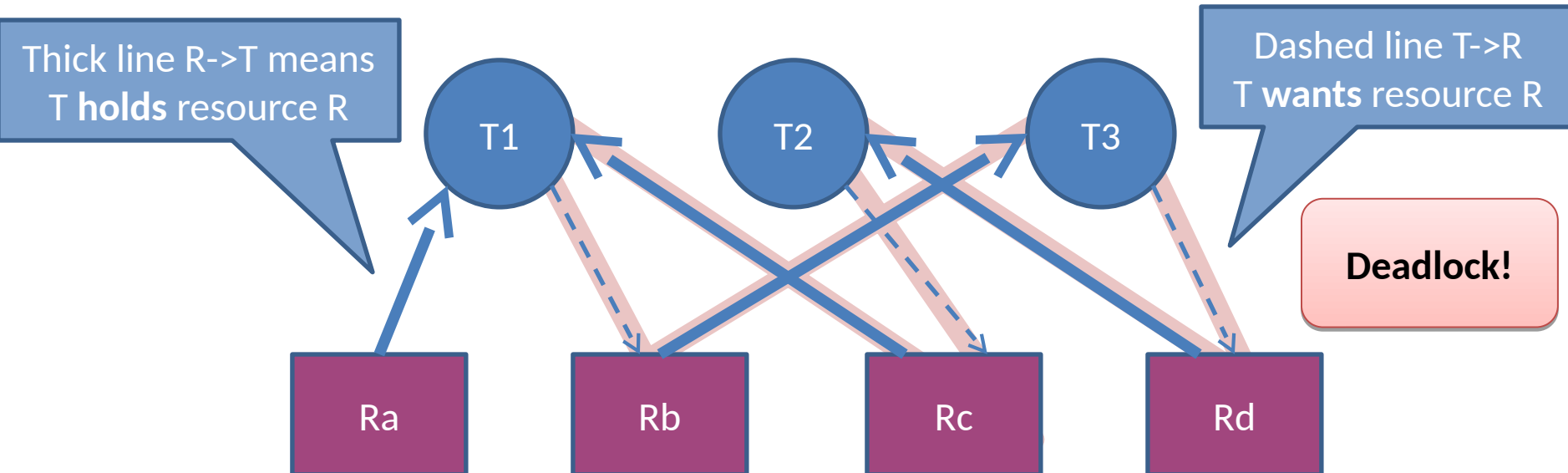
Risk of **deadlock** if
both threads get here
simultaneously

Requirements for deadlock

- Like all concurrency bugs, deadlock may be rare (e.g. imagine **<cond>** is mostly false)
- In practice there are four necessary conditions
 1. **Mutual Exclusion**: resources have bounded **#owners**
 2. **Hold-and-Wait**: can acquire **R_x** and wait for **R_y**
 3. **No Preemption**: keep **R_x** until you release it
 4. **Circular Wait**: cyclic dependency
- Require all four to hold for deadlock
 - But most modern systems always satisfy 1, 2, 3
- Tempting to think that this applies only to locks ...
 - But it also can occur for many other resource classes whose allocation meets conditions: memory, CPU time, ...

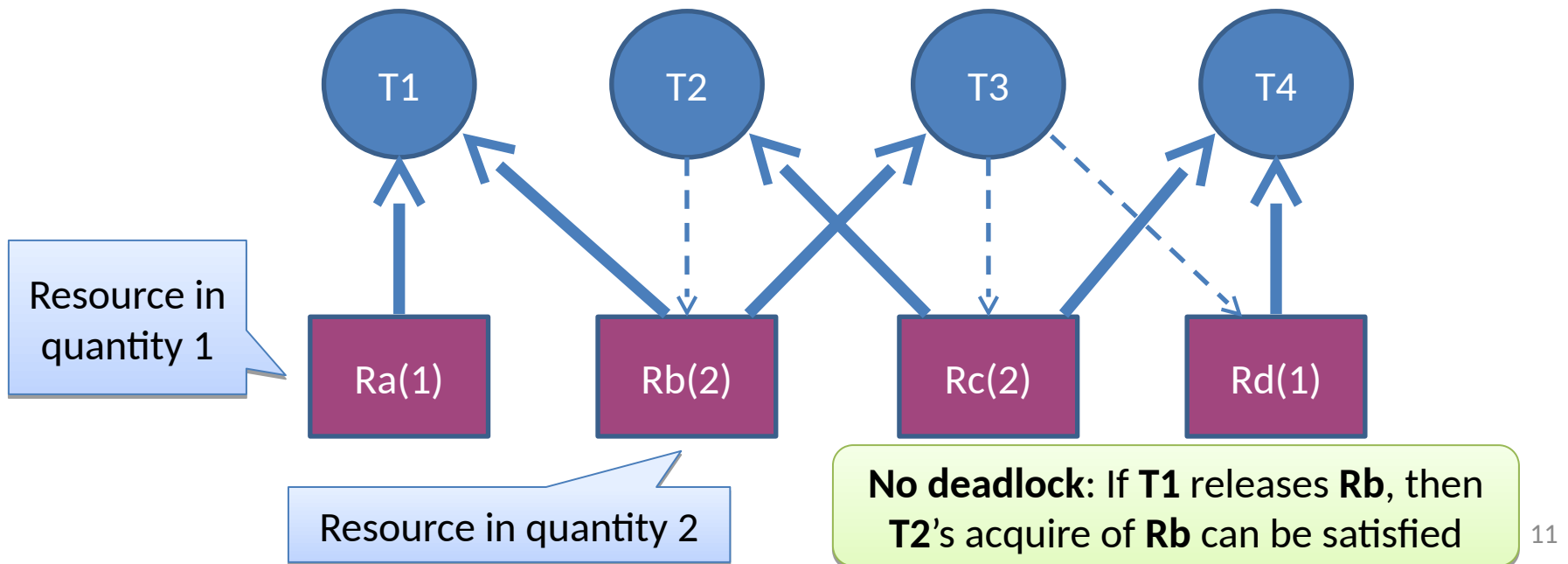
Resource allocation graphs

- Graphical way of thinking about deadlock:
 - **Circles** are threads (or processes)
 - **Boxes** are single-owner resources (e.g. mutexes)
 - Edges show **lock hold** and **wait** conditions
 - A **cycle** means we (will) have deadlock.



Resource allocation graphs (2)

- Can generalize to resources which can have **K** distinct users (c/f semaphores)
- Absence of a cycle means no deadlock...
 - but presence only means *may encounter* deadlock, e.g.



Resource allocation graphs (3)

- Another generalisation is for threads to have several possible ways forward and that are able to select according to which locks have already been taken.
- Read up on generalised AND-OR wait-for graphs for those interested (link will be on course web site).
- [This slide non-examinable].

Deadlock: 3 Design Approaches

1. Ensure it never happens
 - Deadlock (static) prevention (using code structure rules)
 - Deadlock (dynamic) avoidance (cycle finding or Banker's Alg)
2. Let it happen, but recover
 - Deadlock (dynamic) detection & recovery
3. Ignore it!
 - The so-called “**Ostrich Algorithm**” ;-)
 - “Have you tried turning it off and back on again?”
 - Very widely used in practice!

Deadlock Static Prevention

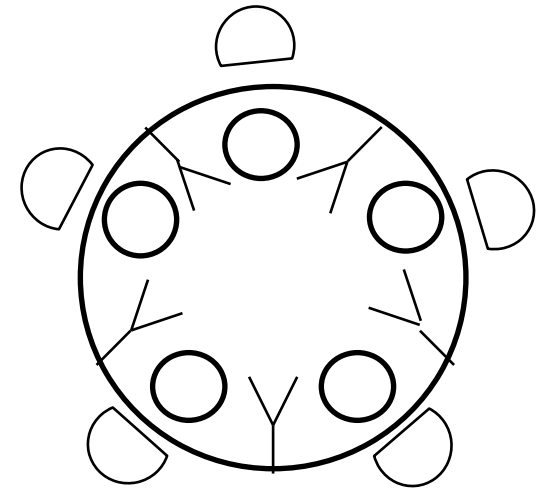
1. **Mutual Exclusion**: resources have bounded **#owners**
 - Could always allow access... but probably unsafe ;-(
 - However can help e.g. by using MRSW locks
2. **Hold-and-Wait**: can get **R_x** and wait for **R_y**
 - Require that we request all resources simultaneously; deny the request if *any* resource is not available now
 - But must know maximal resource set in advance = hard?
3. **No Preemption**: keep **R_x** until you release it
 - Stealing a resource generally unsafe (but see later)
4. **Circular Wait**: cyclic dependency
 - Impose a partial order on resource acquisition,
 - Can work: but requires programmer discipline.
 - Lock order enforcement rules used in many systems e.g., FreeBSD WITNESS – static and dynamic orders checked.

Example: Dining Philosophers

- 5 philosophers, 5 forks, round table...

```
Semaphore forks[] = new Semaphore[5];
```

```
while(true) {           // philosopher i
    think();
    wait(fork[i]);
    wait(fork[(i+1) % 5]);
    eat();
    signal(fork[i]);
    signal(fork[(i+1) % 5]);
}
```



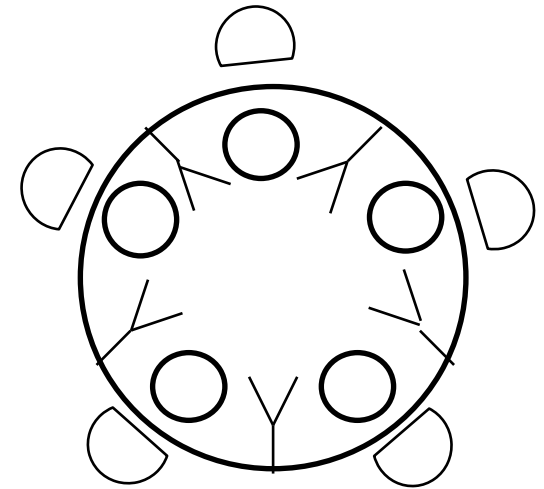
- Possible for everyone to acquire 'left' fork
 - Q: *what happens if we swap order of **wait()**s?*

Example: Dining Philosophers Sltn

- (one) Solution: always take lower fork first

```
Semaphore forks[] = new Semaphore[5];
```

```
while(true) {           // philosopher i
    think();
    first = MIN(i, (i+1) % 5);
    second = MAX(i, (i+1) % 5);
    wait(fork[first]);
    wait(fork[second]);
    eat();
    signal(fork[second]);
    signal(fork[first]);
}
```



- Now even if 0, 1, 2, 3 are held, 4 will not acquire final fork.

Deadlock Dynamic Avoidance

- Prevention aims for deadlock-free “by design”.
- **Deadlock avoidance** is a dynamic scheme:
 - Assumption: We know maximum possible resource allocation for every process / thread,
 - Assumption: A process granted all desired resources will **complete, terminate, and free its resources.**
 - Track actual allocations in real-time,
 - When a request is made, only grant if guaranteed no deadlock even if all others take max resources.
- E.g. **Banker's Algorithm**
 - Not really useful in general as need *a priori* knowledge of **#processes/threads**, and their max resource needs.

Deadlock detection (anticipation)

- **Deadlock detection** is a dynamic scheme that determines if deadlock exists (or would exist if we granted a request).
 - **Principle:** At specific moments in execution, examine resource allocations and graph.
 - Determine if there is **at least one** plausible sequence of events in which all threads could make progress.
 - I.e., check that we are not in an **unsafe state** in which no further sequences can complete without deadlock.
- When only a single instance of each resource, can explicitly check for a cycle:
 - Keep track which object each thread is waiting for,
 - From time to time, iterate over all threads and build the resource allocation graph,
 - Run a cycle detection algorithm on graph $O(n^2)$.
- Or use Banker's Alg if have multi-instance resources (more difficult)

Banker's Algorithm (1)

- Have **m** distinct resources and **n** threads
- $V[0:m-1]$, vector of **currently** available resources
- **A**, the $m \times n$ resource allocation matrix, and **R**, the $m \times n$ (outstanding) request matrix
 - $A_{i,j}$ is the number of objects of type j **owned** by i
 - $R_{i,j}$ is the number of objects of type j further **needed** by i
- Proceed by successively marking rows in **A** for threads that are not part of a deadlocked set.
 - If we cannot mark all rows of **A** we have deadlock.

Optimistic assumption: if we can fulfil thread i 's request R_i , then it will run to completion and release held resources for other threads to allocate.

Banker's Algorithm (2)

- Mark all zero rows of **A** (since a thread holding zero resources can't be part of deadlock set)
- Initialize a working vector **W**[0:m-1] to **V**
 - **W**[] describes any free resources at start, **plus** any resources released by a hypothesized sequence of satisfied threads freeing and terminating
- • Select an unmarked row *i* of **A** s.t. $\mathbf{R}[i] \leq \mathbf{W}$
 - (i.e. find a thread who's request can be satisfied)
 - Set $\mathbf{W} = \mathbf{W} + \mathbf{A}[i]$; mark row *i*, and repeat
- Terminate when no such row can be found
 - Unmarked rows (if any) are in the deadlock set

Banker's Algorithm: Example 1

- Five threads and three resources (none free)

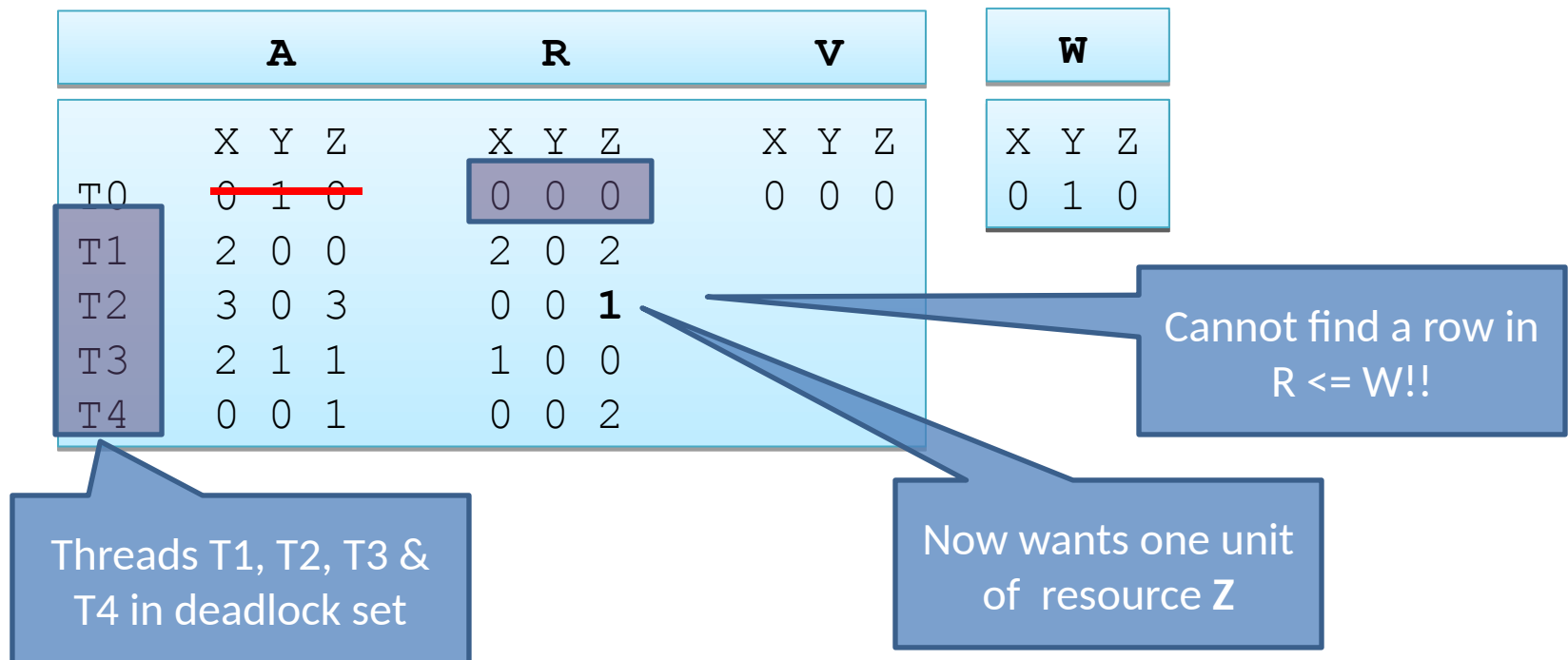
	A			R			V			W		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
T0	0	1	0	0	0	0	0	0	0	7	2	5
T1	2	0	0	2	0	2						
T2	3	0	3	0	0	0						
T3	2	1	1	1	0	0						
T4	0	0	1	0	0	2						

- Find an unmarked row, mark it, and update **W**
 - T0, T2, T3, T4, T1

At the end of the algorithm, all rows are marked:
the deadlock set is empty.

Banker's Algorithm: Example 2

- Five threads and three resources (none free)



- One minor tweak to T2's request vector...

Deadlock recovery

- What can we do when we detect deadlock?
- Simplest solution: kill something!
 - Ideally someone in the deadlock set ;-)
- Brutal, and not guaranteed to work
 - But sometimes the best (only) thing we can do
 - E.g. Linux OOM killer (better than system reboot?)
 - ... Or not – often kills the X server!
- Could also resume from checkpoint
 - Assuming we have one
- In practice computer systems seldom detect or recover from deadlock: rely on programmer.

Note: “kill someone” breaks the **no preemption** precondition for deadlock.

Livelock

- **Deadlock** is at least 'easy' to detect by humans
 - System basically blocks & stops making any progress
- **Livelock** is less easy to detect as threads continue to run... but do nothing useful
- Often occurs from trying to be clever, e.g.:

```
// thread 1
lock (X);
```

```
...
```

```
while (!trylock (Y)) {
    unlock (X);
    yield();
    lock (X);
}
```

```
...
```

```
// thread 2
lock (Y);
```

```
...
```

```
while (!trylock (X)) {
    unlock (Y);
    yield();
    lock (Y);
}
```

```
...
```

Livelock if both threads get here simultaneously

Scheduling and thread priorities

- Which thread should run when >1 runnable? E.g., if:
 - A thread releases a contended lock and continues to run,
 - CV broadcast wakes up several waiting threads.
- Many possible **scheduling policies**:
 - **Round robin** – rotate between threads to ensure progress,
 - **Fixed priorities** – assign priorities to threads, schedule highest– e.g., **real-time** $>$ **interactive** $>$ **bulk** $>$ **idle-time**
 - **Dynamic priorities** – adjust priorities to balance goals – e.g. boost priority after I/O to improve interactivity,
 - **Gang scheduling** – schedule for patterns such as P-C
 - **Affinity** – schedule for efficient resource use (e.g. caches).
- Goals: latency vs. throughput, energy, “fairness”, ...
 - NB: These competing goals cannot generally all be satisfied.

Priority inversion

- Another liveness problem...
 - Due to interaction between locking and scheduler.
- Consider three threads: **T1, T2, T3**
 - **T1** is high priority, **T2** medium priority, **T3** is low
 - **T3** gets lucky and acquires lock **L**...
 - ... **T1** preempts **T3** and sleeps waiting for **L**...
 - ... then **T2** runs, preventing **T3** from releasing **L**!
 - **Priority inversion**: despite having higher priority and no shared lock, **T1** waits for lower priority thread **T2**
- This is not deadlock or livelock
 - But not desirable (particularly in real-time systems)!
 - Disabled Mars Pathfinder robot for several months.



Priority inheritance

- Typical solution is **priority inheritance**:
 - Temporarily boost priority of lock holder to that of the highest waiting thread
 - **T3** would have run with **T1**'s priority while holding a lock **T1** was waiting for – preventing **T2** from preempting **T3**
 - Concrete benefits to system interactivity
 - (some RT systems (like VxWorks) allow you specify on a per-mutex basis [to Rover's detriment ;-])
- Windows “solution”
 - Check if any ready thread hasn't run for 300 ticks
 - If so, double its quantum and boost its priority to 15
 - 😊

Problems with priority inheritance

- Hard to reason about resulting behaviour: **heuristic**
- Works for locks
 - More complex than it appears: propagation might need to be **propagated** across chains containing multiple locks
 - (How might we handle reader-writer locks?)
- How about condition synchronisation, res. allocation?
 - With locks, we know what thread holds the lock
 - Semaphores do not record which thread might issue a signal or release an allocated resource
 - Must compose across multiple waiting types: e.g. “waiting for a signal while holding a lock.”
- Where possible, avoid the need for priority inheritance
 - Avoid sharing between threads of differing priorities.

Limits to Parallelisation

Depending on how it is coded, a program or task can exhibit various levels of dependency between its parts:

- **No dependencies** (embarrassingly parallel): No dependencies between work units, such as Mandelbrot pixel or JPEG tile.
- **Data dependencies**: where the result of one computation is needed for others.
- **Control dependencies**: where its not known if a result will be needed.
- One can **speculate** on both types of dependency, guessing the outcome, but some amount of work will be wasted and results must not be committed.

Summary + next time

- **Liveness** properties
- **Deadlock**
 - Requirements
 - Resource allocation graphs and detection
 - Prevention – the **Dining Philosophers Problem** – and recovery
- **Thread priority** and the **scheduling problem**
- **Priority inversion and inheritance**
- **Limits to parallelisation.**
- Next time:
 - Concurrency without shared data
 - Active objects; message passing
 - Composite operations; transactions
 - ACID properties; isolation; serialisability

Concurrent systems

Lecture 6: Concurrency without shared data, composite operations and transactions, and serialisability

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

Reminder from last time

- Liveness properties
- Deadlock (requirements; resource allocation graphs; detection; prevention; recovery)
- The Dining Philosophers
- Priority inversion
- Priority inheritance

Concurrency is so hard!

If only there were some way that programmers could accomplish useful concurrent computation without...

- (1) the hassles of shared memory concurrency
- (2) blocking synchronisation primitives

This time

- Concurrency without shared data
 - Use same hardware+OS primitives, but expose higher-level models via **software libraries** or **programming languages**
- **Active objects**
 - Ada
- **Message passing; the Actor model**
 - Occam, Erlang
- **Composite operations**
 - Transactions, **ACID** properties
 - **Isolation** and **serialisability**
- **History graphs; good (and bad) schedules**

This material has significant overlap with **databases** and **distributed systems** – but is presented here from a concurrency perspective

Concurrency without shared data

- The examples so far have involved threads which can arbitrarily read & write shared data
 - A key need for mutual exclusion has been to avoid race-conditions (i.e. 'collisions' on access to this data)
- An alternative approach is to have only one thread access any particular piece of data
 - Different threads can own distinct chunks of data
- Retain concurrency by allowing other threads to ask for operations to be done on their behalf
 - This 'asking' of course needs to be concurrency safe...

Fundamental design dimension: concurrent access via **shared data** vs. concurrent access via **explicit communication**

Example: Active objects

- A monitor with an associated **server** thread
 - Exports an **entry** for each operation it provides
 - Other (**client**) threads ‘call’ methods
 - Call returns when operation is done
- All complexity bundled up in an **active object**
 - Must manage mutual exclusion where needed
 - Must queue requests from multiple threads
 - May need to delay requests pending conditions
 - E.g. if a producer wants to insert but buffer is full.

Observation: the code of **exactly** one thread, and the data that only it accesses, effectively experience **mutual exclusion**

Producer-Consumer in Ada

```
task-body ProducerConsumer is
...
loop
  SELECT
    when count < buffer-size
      ACCEPT insert(item) do
        // insert item into buffer
      end;
    count++;
  or
    when count > 0
      ACCEPT consume(item) do
        // remove item from buffer
      end;
    count--;
  end SELECT
end loop
```

Clause is *active* only when condition is true

ACCEPT dequeues a client request and performs the operation

Single thread: no need for mutual exclusion

Non-deterministic choice between a set of *guarded* ACCEPT clauses

Reliable Message Passing

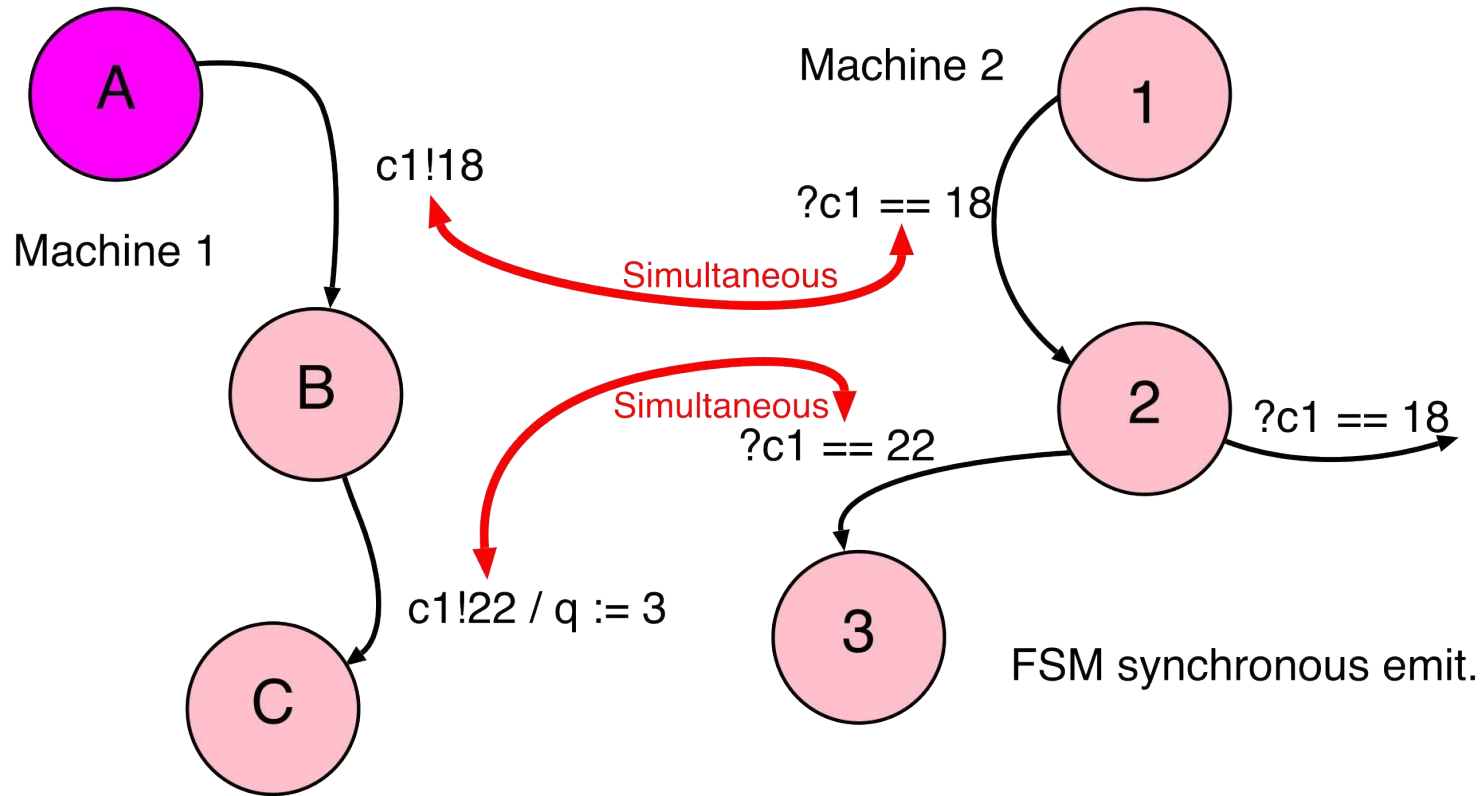
- Dynamic invocations between threads can be thought of as general **message passing**
 - Thread X can send a message to Thread Y
 - Contents of message can be arbitrary data values
- Can be used to build **Remote Procedure Call** (RPC)
 - Message includes name of operation to invoke along with as any parameters
 - Receiving thread checks operation name, and invokes the relevant code
 - Return value(s) sent back as another message
- (Called **Remote Method Invocation** (RMI) in Java)

We will discuss message passing and RPC in detail 2nd half; a taster now, as these ideas apply to local, not just distributed, systems.

Message passing semantics

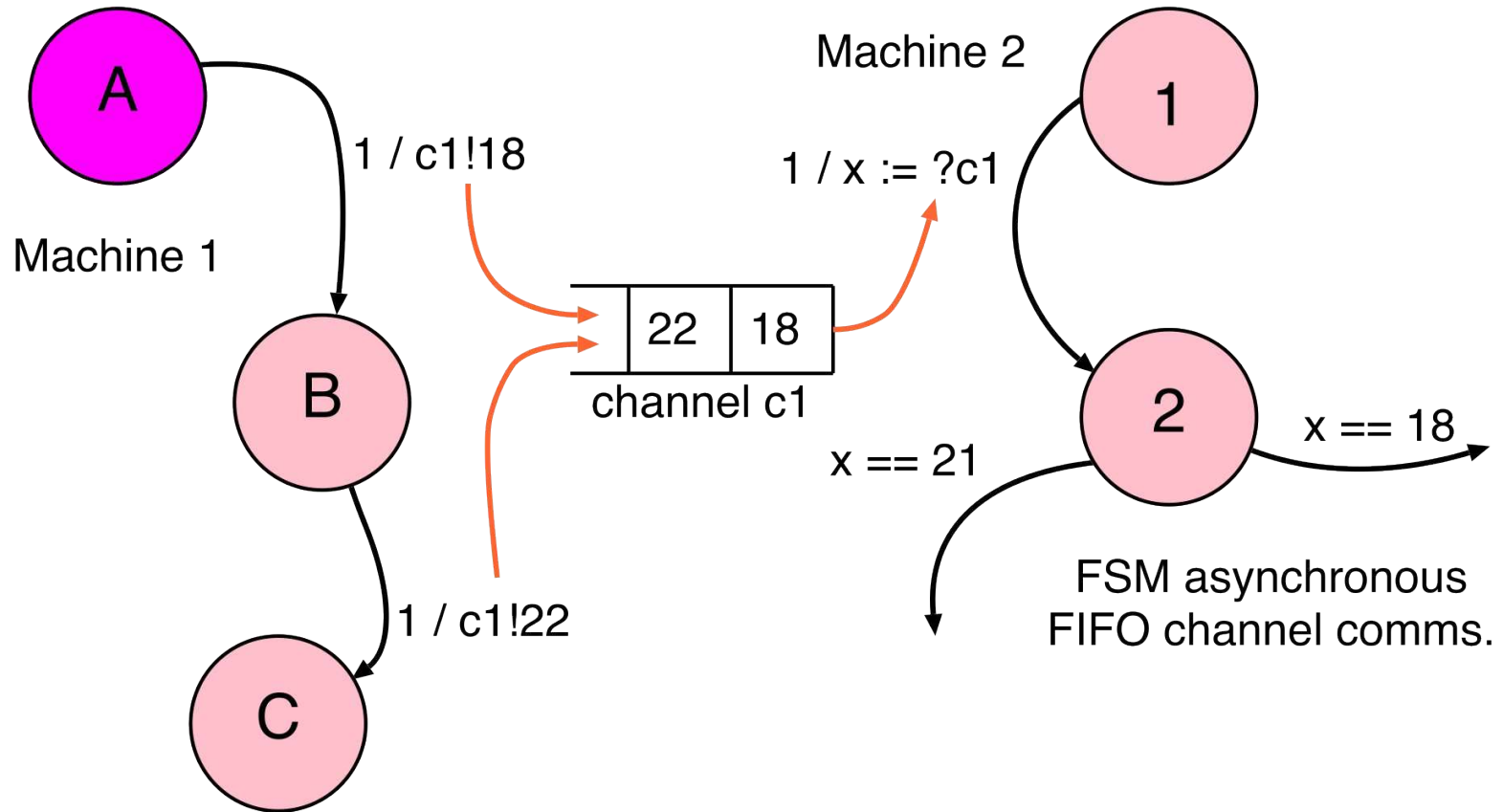
- Can conceptually view sending a message to be similar to sending an email *within a reliable system*:
 1. Sender prepares contents locally, and then sends
 2. System eventually delivers a **copy** to receiver
 3. Receiver checks for messages
- In this model, sending is **asynchronous**:
 - Sender doesn't need to wait for message delivery
 - (but they may, of course, choose to wait for a reply)
 - **Bounded FIFO may ultimately apply sender backpressure.**
- Receiving is also asynchronous:
 - messages first **delivered** to a mailbox, later **retrieved**
 - message is a **copy** of the data (ie. no actual sharing).

Synchronous Message Passing



- FSM view: both (all) participating FSMs execute the message passing primitive simultaneously.
- Send and receive operations must be part of edge guard (before the slash).

Asynchronous Message Passing



- We will normally assume asynchronous unless obviously or explicitly otherwise.
- Send and receive operations in action part (after slash).

Message passing advantages

- **Copy semantics** avoid race conditions
 - At least directly on the data
- Flexible API: e.g.
 - **Batching**: can send K messages before waiting; and can similarly batch a set of replies
 - **Scheduling**: can choose when to receive, whom to receive from, and which messages to prioritise.
 - **Broadcast/multicast**: can send messages to many recipients
- Works both within and between machines
 - ie. same design works for **distributed systems**.
- Explicitly used as basis of some languages ...

Example: Occam

- Language based on Hoare's **Communicating Sequential Processes** (CSP)
 - A projection of a process algebra into a real-world language.
- No shared variables.
- Processes **synchronously** communicate via **channels**

```
<channel> ? <variable>      // an input operation  
<channel> ! <expression>    // an output operation
```

- Build complex processes via SEQ, PAR and ALT, eg.

```
ALT  
  count1 < 100 & c1 ? Data  
    SEQ  
      count1 := count1 + 1  
      merged ! data  
  count2 < 100 & c2 ? Data  
    SEQ  
      count2 := count2 + 1  
      merged ! data
```

Example: Erlang

- Functional programming language designed in mid 80's, made popular more recently (especially in eternal systems such as telephone network).
- Implements the **actor model**
- **Actors**: lightweight language-level processes
 - Can spawn() new processes very cheaply
- **Single-assignment**: each variable is assigned only once, and thereafter is immutable
 - But values can be sent to other processes
- **Guarded receives** (as in Ada, occam)
 - Messages delivered in-order to local mailbox
- Message/actor-oriented model allows run-time restart or replacement of modules to limit downtime.

Proponents of Erlang argue that lack of synchronous message passing prevents deadlock. Why might this claim be misleading?

Producer-Consumer in Erlang

```
-module (producerconsumer) .  
-export ([start/0]) .
```

```
start() ->  
    spawn(fun() -> loop() end) .
```

```
loop() ->  
    receive  
        {produce, item } ->  
            enter_item(item),  
            loop();  
        {consume, Pid } ->  
            Pid ! remove_item(),  
            loop();  
    stop ->  
        ok
```

```
end .
```

Invoking start() will
spawn an actor...

receive matches
messages to patterns

explicit tail-recursion is
required to keep the
actor alive...

... so if send 'stop',
process will terminate.

Message passing: summary

- A way of sidestepping (at least some of) the issues with shared memory concurrency
 - No direct access to data => no **data** race conditions
 - Threads choose actions based on message.
- Explicit message passing can be awkward
 - Many weird and wonderful languages ;-)
- Can also use with traditional languages, eg.
 - Transparent messaging via RPC/RMI
 - Scala, Kilim (actors on Java), Bastion for Rust, ...
- May overcome cache-consistency scaling issues?

We have eliminated some of the issues associated with shared memory, but these are still concurrent programs subject to deadlock, livelock, etc.

Composite operations

- So far have seen various ways to ensure safe concurrent access to a single object
 - e.g. monitors, active objects, message passing
- More generally want to handle **composite operations**:
 - ie. build systems which act on multiple distinct objects
- As an example, imagine an internal bank system which allows account access via three method calls:

```
int amount = getBalance(account);  
bool credit(account, amount);  
bool debit(account, amount);
```

- If each is thread-safe, is this sufficient?
 - Or are we going to get into trouble???

Composite operations

- Consider two concurrently executing client threads:
 - One wishes to transfer 100 quid from the savings account to the current account,
 - The other wishes to learn the combined balance.

```
// thread 1: transfer 100
// from savings->current
debit(savings, 100);
credit(current, 100);
```

```
// thread 2: check balance
s = getBalance(savings);
c = getBalance(current);
tot = s + c;
```

- If we're unlucky then:
 - Thread 2 could see balance that's too small
 - Thread 1 could crash after doing debit() - ouch!
 - Server thread could crash at any point - ouch?

Problems with composite operations

Two separate kinds of problem here:

1. Insufficient Isolation

- Individual operations being atomic is not enough,
- Eg. want the credit & debit making up the transfer to happen as one operation.
- Could fix this particular example with a new transfer() method, but not very general ...

2. Fault Tolerance

- In the real-world, programs (or systems) can fail.
- Need to make sure we can recover safely.

Transactions

- Want programmer to be able to specify that a set of operations should happen **atomically**, eg.

```
// transfer amt from A -> B
transaction {
  if (getBalance(A) > amt) {
    debit(A, amt);
    credit(B, amt);
    return true;
  } else return false;
}
```

- A **transaction** either executes correctly (in which case we say it **commits**), or has no effect at all (i.e. it **aborts**).
 - regardless of other transactions, or system crashes!

ACID Properties

Want committed transactions to satisfy four properties:

- **Atomicity**: either all or none of the transaction's operations are performed
 - Programmer doesn't need to worry about clean up.
- **Consistency**: a transaction transforms the system from one consistent state to another – ie. preserves **invariants**.
 - Programmer must still ensure eg. conservation of money.
- **Isolation**: each transaction executes [as if] isolated from the concurrent effects of others
 - We can then ignore partial updates from concurrent transactions.
- **Durability**: the effects of committed transactions survive subsequent system failures
 - If system reports success, must ensure this is recorded on disk.

This is a different use of the word “atomic” from previously; we will just have to live with that, unfortunately.

ACID Properties

Can group these into two categories

1. **Atomicity** & **Durability** deal with making sure the system is safe even across failures:
 - (A) No partially complete txactions
 - (D) Transactions previously reported as committed don't disappear, even after a system crash.
2. **Consistency** & **Isolation** ensure correct behavior even in the face of concurrency:
 - (C) Can always code as if invariants in place,
 - (I) Concurrently executing transactions are indivisible.

Isolation

- To ensure a transaction executes in isolation, could just have a server-wide lock... simple!

```
// transfer amt from A -> B
transaction { // acquire server lock
  if (getBalance(A) > amt) {
    debit(A, amt);
    credit(B, amt);
    return true;
  } else return false;
} // release server lock
```

- But doesn't allow any concurrency...
- And doesn't handle mid-transaction failure (e.g. what if we are unable to credit the amount to **B**?).

Isolation – Serialisability

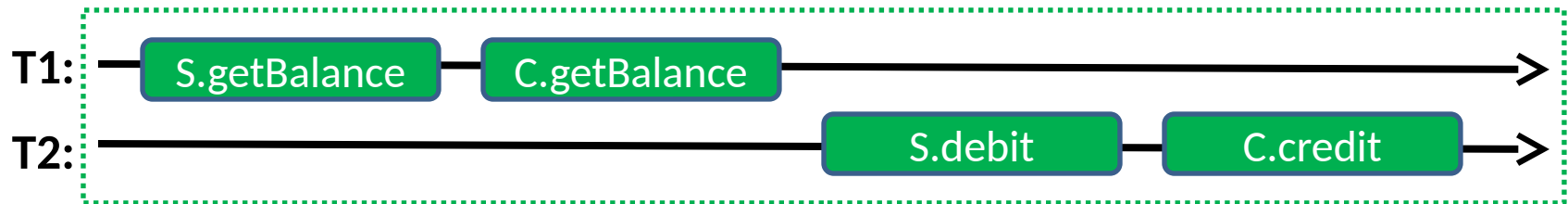
- The idea of executing transactions **serially** (one after the other) is a useful **model for the programmer**:
 - To improve performance, **transaction systems** execute many transactions concurrently,
 - But programmers must only observe behaviours consistent with a possible serial execution: **serialisability**.
- Consider two transactions, **T1** and **T2**

```
T1 transaction {  
    s = getBalance(S);  
    c = getBalance(C);  
    return (s + c);  
}
```

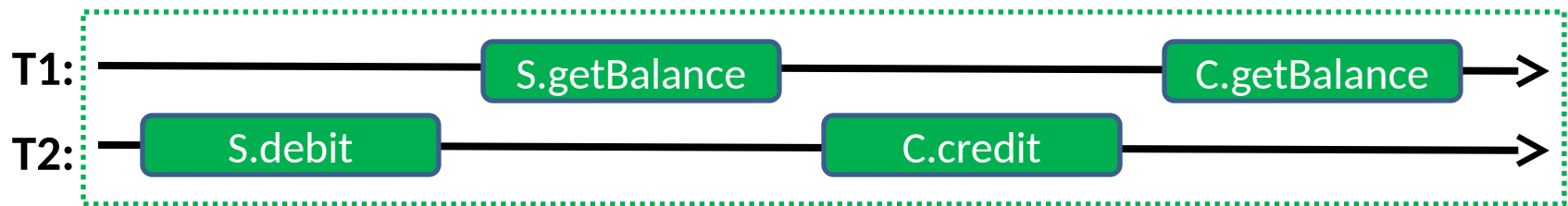
```
T2 transaction {  
    debit(S, 100);  
    credit(C, 100);  
    return true;  
}
```

- If assume individual operations are atomic, then there are six possible ways the operations can interleave...

Isolation – serialisability

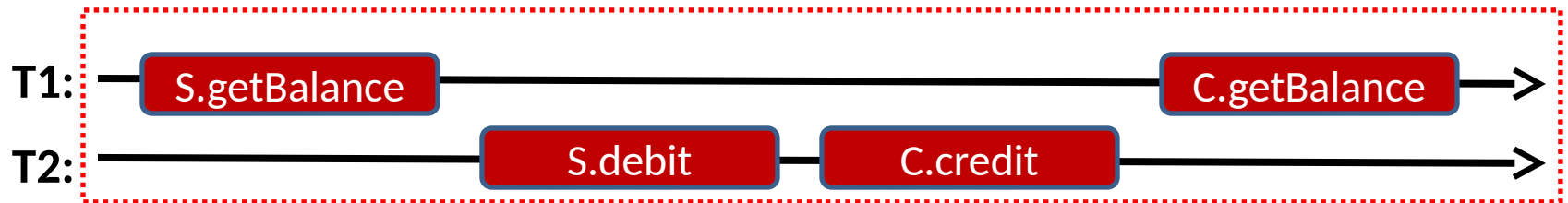


- First case is a **serial execution** and hence **serialisable**

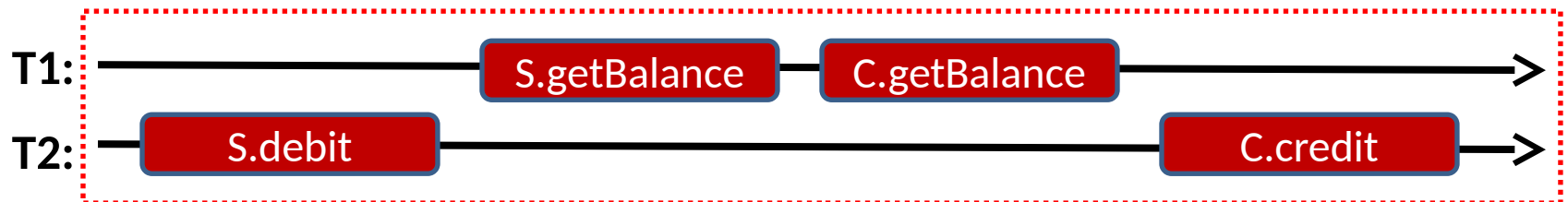


- Second case is **not serial** as transactions are interleaved
 - Its results are identical to serially executing T2 and then T1
 - The schedule is therefore **serialisable**.
- Informally: it is serialisable because we have only swapped the execution orders of **non-conflicting operations**.
 - All of T1's operations on any object happen after T2's updates.

Isolation – serialisability



- This execution is neither **serial** nor **serialisable**
 - T1 sees inconsistent values: old S and new C.



- This execution is also neither **serial** nor **serialisable**
 - T1 sees inconsistent values: new S, old C.
- Both orderings swap **conflicting operations** such that there is no matching serial execution.

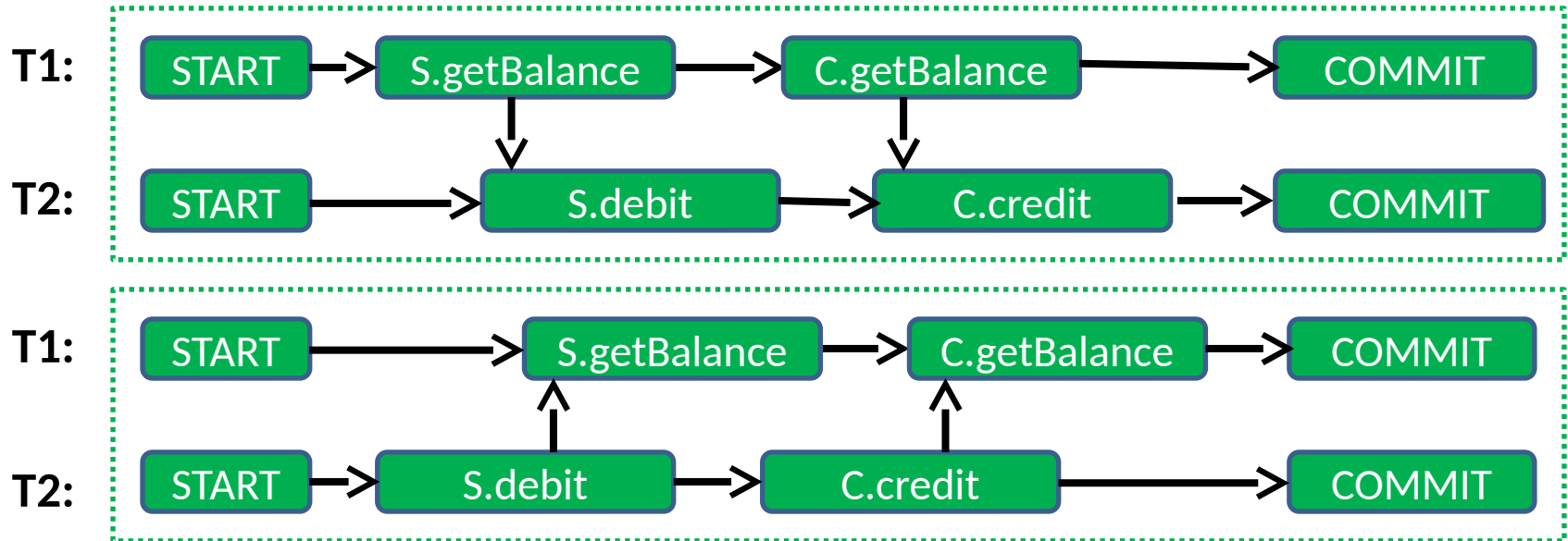
Conflict Serialisability

- There are many flavours of serialisability
- **Conflict serialisability** is satisfied for a schedule **S** if (and only if):
 - It contains the same set of operations as some serial schedule **T**; and
 - All **conflicting operations** are ordered the same way as in **T**.
- Define **conflicting** as **non-commutative**
 - IE. differences are permitted between the execution ordering and **T**, but they can't have a visible impact.

History graphs

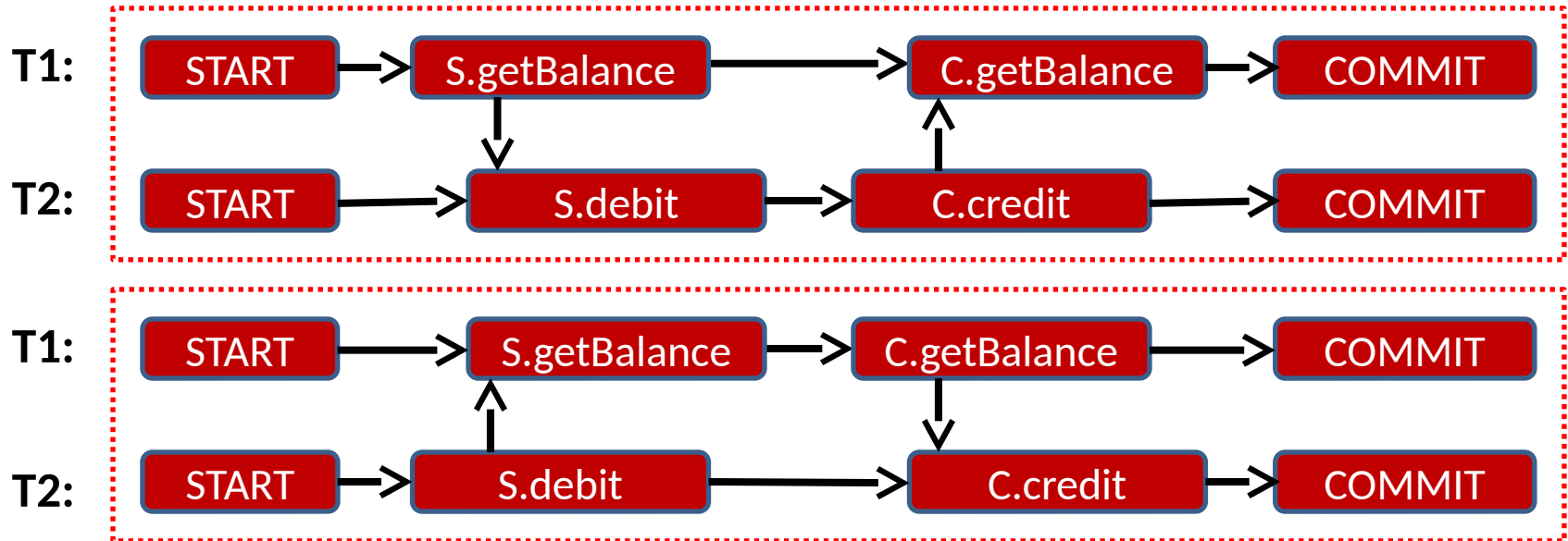
- Can construct a graph for any execution schedule:
 - Nodes represent individual operations, and
 - Arrows represent “**happens-before**” relations.
- Insert edges between operations within a given transaction in **program order** (ie. as written).
- Insert edges between **conflicting** operations operating on the same objects, ordered by execution schedule
 - e.g. A.credit(), A.debit() commute [don't conflict]
 - A.credit() and A.addInterest() **do** conflict
- NB: Graphs represent **particular execution schedules** not **sets of allowable schedules**.

History graphs: good schedules



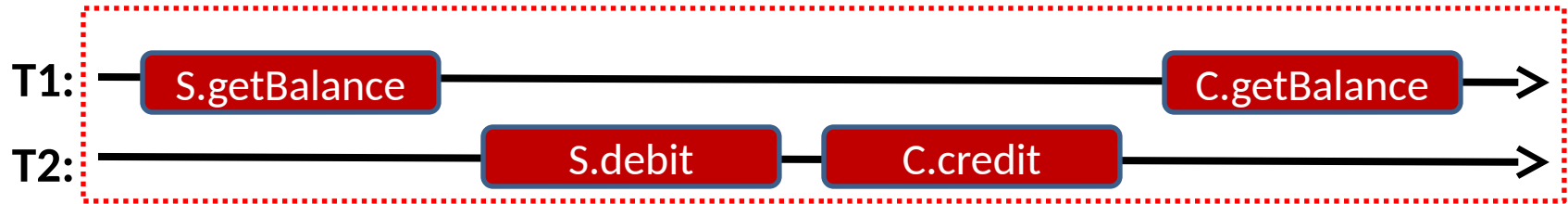
- Same schedules as before (both ok)
- Can easily see that everything in **T1** either happens before everything in **T2**, or vice versa
 - Hence schedule can be serialised.

History graphs: bad schedules

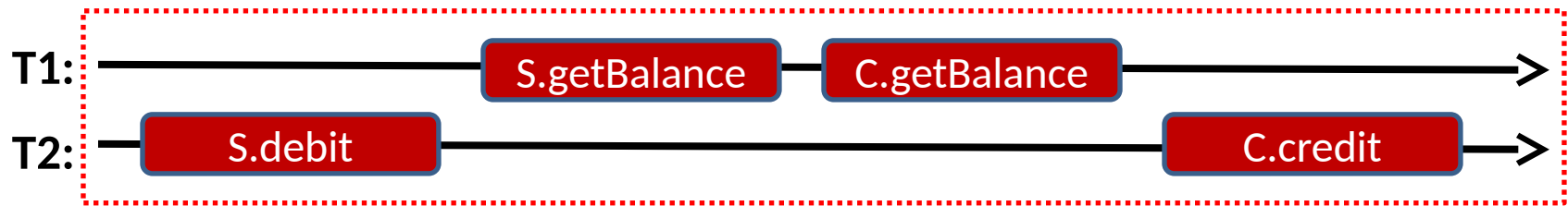


- Cycles between threads indicate that schedules are bad :-)
- Neither transaction strictly “happened before” the other:
 - Arrows from **T1** to **T2** mean “**T1** must happen before **T2**”
 - But arrows from **T2** to **T1** => “**T2** must happen before **T1**”
 - Notice the **cycle** in the graph (where each thread is considered one node)!
- Can’t both be true --- schedules are **non-serialisable**.

Isolation – serialisability



- This execution is neither **serial** nor **serialisable**
 - T1 sees inconsistent values: old S and new C



- This execution is also neither **serial** nor **serialisable**
 - T1 sees inconsistent values: new S, old C
- Both orderings swap **conflicting operations** such that there is no matching serial execution

The **transaction system** must ensure that, regardless of any actual concurrent execution used to improve performance, only results consistent with **serialisable orderings** are visible to the **transaction programmer**.

Summary + next time

- Concurrency without shared data (Active Objects)
- Message passing, actor model (Occam, Erlang)
- Composite operations; transactions; ACID properties
- Isolation and serialisability
- History graphs; good (and bad) schedules.

- Next time – more on transactions:
 - Isolation vs. strict isolation; enforcing isolation.
 - Two-phase locking; rollback
 - Timestamp ordering (TSO); optimistic concurrency control (OCC)
 - Isolation and concurrency summary.

Concurrent Systems

Lecture 7: Isolation vs. Strict Isolation,
2-Phase Locking (2PL), Time Stamp Ordering (TSO), and
Optimistic Concurrency Control (OCC)

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

Reminder from last time

- Concurrency without shared data
 - Active objects
- **Message passing**; the actor model
 - Occam, Erlang
- **Composite operations**
 - Transactions, **ACID** properties
 - Isolation and serialisability
- **History graphs**; good (and bad) schedules

Last time: isolation – serialisability

- The idea of executing transactions **serially** (one after the other) is a useful model
 - We want to run transactions concurrently
 - But the result should be **as if** they ran serially
- Consider two transactions, **T1** and **T2**

```
T1 transaction {  
  s = getBalance (S) ;  
  c = getBalance (C) ;  
  return (s + c) ;  
}
```

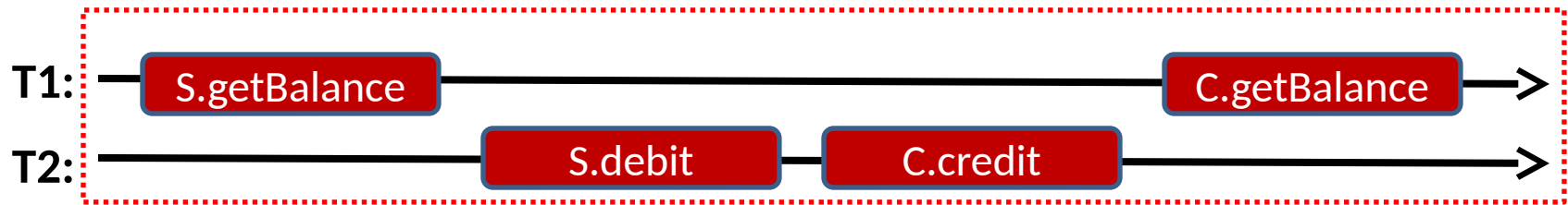
```
T2 transaction {  
  debit (S, 100) ;  
  credit (C, 100) ;  
  return true ;  
}
```

- If assume individual operations are atomic, then there are six possible ways the operations can interleave...

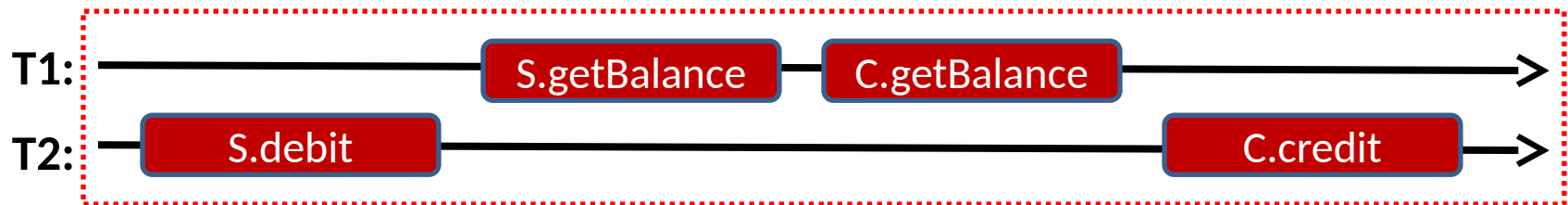
Isolation allow transaction programmers to reason about the interactions between transactions trivially: they appear to execute in **serial**.

Transaction systems execute transactions concurrently for performance and rely on the definition of **serialisability** to decide if an actual execution schedule is allowable.

Isolation – serialisability



- This execution is neither **serial** nor **serialisable**
 - T1 sees inconsistent values: old S and new C



- This execution is also neither **serial** nor **serialisable**
 - T1 sees inconsistent values: new S, old C
- Both orderings swap **conflicting operations** such that there is no matching serial execution

The **transaction system** must ensure that, regardless of any actual concurrent execution used to improve performance, only results consistent with **serialisable orderings** are visible to the **transaction programmer**.

This time

- Effects of bad schedules
- Isolation vs. strict isolation; enforcing isolation
- Two-phase locking; rollback
- Timestamp ordering (TSO)
- Optimistic concurrency control (OCC)
- Isolation and concurrency summary.

This lecture considers how the **transaction implementation** itself can provide transactional (**ACID**) guarantees

Effects of bad schedules

Lack of **atomicity**:
operation results “lost”

- **Lost Updates**

- **T1** updates (writes) an object, but this is then overwritten by concurrently executing **T2**
- (if T1 prevails when T2 logically later, write-write conflict, WaW)

- **Dirty Reads**

- **T1** reads an object which has been updated by an uncommitted transaction **T2**
- (can be a read-after-write conflict, RaW)

- **Unrepeatable Reads**

- **T1** reads an object which is then updated by **T2**
- Not possible for **T1** to read the same value again
- (also called a write-after-read conflict, WaR or *anti-dependance*. T2 needs to be held off)

Lack of **isolation**:
partial result seen

Lack of **isolation**:
read value
unstable

Atomicity: all or none of operations performed – **abort** must be “clean”
Isolation: transactions execute as if isolated from concurrent effects

Isolation and strict isolation

- Ideally want to avoid all three problems
- Two ways: **Strict Isolation** and **Non-Strict Isolation**
 - **Strict Isolation**: guarantee we never experience lost updates, dirty reads, or unrepeatable reads
 - **Non-Strict Isolation**: let transaction continue to execute despite potential problems (i.e., more **optimistic**)
- Non-strict isolation usually allows more concurrency but can lead to complications
 - E.g. if **T2** reads something written by **T1** (a “dirty read”) then **T2** cannot commit until **T1** commits
 - And **T2** must abort if **T1** aborts: **cascading aborts**
- Both approaches ensure that only serialisable schedules are visible to the transaction programmer.

Enforcing isolation

- In practice there are a number of techniques we can use to enforce isolation (of either kind)
- We will look at:
 - **Two-Phase Locking (2PL)**;
 - **Timestamp Ordering (TSO)**; and
 - **Optimistic Concurrency Control (OCC)**
- Essential difference: when is a serialisation decided?
- More complete descriptions and examples of these approaches can be found in:

Operating Systems, Concurrent and Distributed Software Design,
Jean Bacon and Tim Harris, Addison-Wesley 2003.
[Also, Chapter 12 of 1st year Databases book Lemahieu.]

Two-phase locking (2PL)

- Associate a lock with every object
 - Could be mutual exclusion, or MRSW
- Transactions proceed in two phases:
 - **Expanding Phase**: during which locks are acquired but none are released,
 - **Shrinking Phase**: during which locks are released, and no further are acquired.
- Operations on objects occur in either phase, providing appropriate locks are held
 - Guarantees serializable execution.

2PL example

Expanding
Phase

```
// transfer amt from A -> B
```

```
transaction {
```

```
  readLock(A);
```

```
  if (getBalance(A) > amt) {
```

```
    writeLock(A);
```

```
    debit(A, amt);
```

```
    writeLock(B);
```

```
    credit(B, amt);
```

```
    writeUnlock(B);
```

```
    addInterest(A);
```

```
    writeUnlock(A);
```

```
    tryCommit(return=true);
```

```
  } else {
```

```
    readUnlock(A);
```

```
    tryCommit(return=false);
```

```
}
```

Acquire a read lock
(shared) before 'read' A

Upgrade to a write lock
(exclusive) before write A

Acquire a write lock
(exclusive) before write B

Release locks when done
to allow concurrency

Shrinking
Phase

Problems with 2PL

- Requires knowledge of which locks required:
 - Complexity arises if complex control flow inside a transaction
 - Some transactions **look up objects dynamically**
 - But not really a problem and can be automated in many systems:
 - User may declare affected objects statically to assist checker tool or have built-in mechanisms in high-level language (HLL) compilers.
- Risk of deadlock:
 - Can attempt to impose a partial order,
 - Or can detect deadlock and **abort**, releasing locks
 - (this is safe for transactions due to **rollback**, which is nice)
- Non-Strict Isolation: releasing locks during execution means others can access those objects
 - e.g. **T1** updates **B**, then releases write lock; now **T2** can read or overwrite the uncommitted value
 - Hence **T2**'s fate is tied to **T1** (whether commit or abort).
 - Fixed using **strict 2PL**: hold all locks until transaction end.

Strict(er) 2PL example

Expanding
Phase

```
// transfer amt from A -> B
```

```
transaction {
```

```
  readLock(A);
```

```
  if (getBalance(A) > amt) {
```

```
    writeLock(A);
```

```
    debit(A, amt);
```

```
    writeLock(B);
```

```
    credit(B, amt);
```

```
    addInterest(A);
```

```
    tryCommit(return=true);
```

```
  } else {
```

```
    readUnlock(A);
```

```
    tryCommit(return=false);
```

```
  } on commit, abort {
```

```
    unlock(A);
```

```
    unlock(B);
```

```
}
```

Retain lock on B here to
ensure **strict isolation**

Unlock All
Phase

By holding locks longer, Strict
2PL risks greater **contention**

2PL: rollback

- Recall that transactions can **abort**
 - Could be due to run-time conflicts (non-strict 2PL), or could be programmed (e.g. on an exception).
- Using locking for isolation works, but means that updates are made ‘in place’
 - ie. once acquire write lock, can directly update.
 - If transaction aborts, need to ensure no visible effects.
- **Rollback** is the process of returning the world to the state it in was before the transaction started
 - IE. to implement **atomicity**: all happened, or none.

Why might a transaction abort?

- Some failures are internal to transaction systems:
 - Transaction **T2** depends on **T1**, and **T1** aborts,
 - Deadlock is detected between two transactions,
 - Memory is exhausted or a system error occurs.
- Some are programmer-triggered:
 - Transaction self-aborted – e.g., **debit()** was not possible owing to inadequate balance or account being locked ...
- Some failures **must be programmer visible**,
- Others may simply trigger **retry of the transaction**.

Implementing rollback: undo

- One strategy is to **undo** operations, e.g.
 - Keep a log of all operations, in order: $O_1, O_2, \dots O_n$
 - On abort, undo changes of $O_n, O_{(n-1)}, \dots O_1$
- Must know how to undo an operation:
 - Assume we log both operations and parameters
 - Programmer can provide an explicit counter action
 - **UNDO**(credit(A, x) $\Rightarrow$ debit(A, x));
- May not be sufficient (e.g. **setBalance**(A, x))
 - Would need to record previous balance, which we may not have explicitly read within transaction...

Implementing rollback: copy

- A more brute-force approach is to take a **copy** of an object before [first] modification
 - On abort, just revert to original copy.
- Has some advantages:
 - Doesn't require programmer effort
 - Undo is simple, and can be efficient (e.g. if there are many operations, and/or they are complex).
- However can lead to high overhead if objects are large ... and may not be needed if don't abort!
 - Can reduce overhead with partial copy on write, **shadow pages**, etc..

Timestamp ordering (TSO)

- 2PL and Strict 2PL are widely used in practice
 - But can limit concurrency (certainly the latter)
 - And must be able to deal with deadlock.
- **Time Stamp Ordering (TSO)** is an alternative approach:
 - As a transaction begins, it is assigned a **timestamp** – the **proposed eventual (total) commit order / serialisation**.
 - Timestamps are **comparable**, and **unique** (can think of as eg. current time – or a logical incrementing version number).
 - Every object **O** records the timestamp of the last transaction to successfully access (read? write?) it: **V(O)**.
 - **T** can access object **O** iff $V(T) \geq V(O)$, where **V(T)** is the timestamp of **T** (otherwise rejected as “**too late**”).
 - If **T** is non-serialisable with timestamp, abort with roll back.

Timestamps allow us to explicitly track new “happens-before” edges, detecting (and preventing) violations.

TSO Concrete Example 1

```
T1 transaction {  
  s = getBalance(S);  
  c = getBalance(C);  
  return = s + c;  
}
```

```
T2 transaction {  
  debit(S, 100);  
  credit(C, 100);  
  return true;  
}
```

Imagine that objects **S** and **C** start off with version **10**

1. **T1** and **T2** both start concurrently:
 - **T1** gets version **27**, **T2** gets version **29**
2. **T1** reads **S** => **ok!** ($27 \geq 10$); **S** gets version **27**
3. **T2** does **debit**(**S**, 100) => **ok!** ($29 \geq 27$); **S** gets version **29**
4. **T1** reads **C** => **ok!** ($27 \geq 10$); **C** gets version **27**
5. **T2** does **credit**(**C**, 100) => **ok!** ($29 \geq 27$); **C** gets version **29**
6. Both transactions commit.

Succeeded as all **conflicting operations** executed in **timestamp order**

Consider steps 4 and 5 now have different interleaving.

TSO Concrete Example 2

```
T1 transaction {  
  s = getBalance(S);  
  c = getBalance(C);  
  return = s + c;  
}
```

```
T2 transaction {  
  debit(S, 100);  
  credit(C, 100);  
  return true;  
}
```

As before, **S** and **C** start off with version **10**

1. **T1** and **T2** both start concurrently:
 - **T1** gets version **27**, **T2** gets version **29**
2. **T1** reads **S** => **ok!** ($27 \geq 10$); **S** gets version **27**
3. **T2** does **debit**(**S**, 100) => **ok!** ($29 \geq 27$); **S** gets version **29**
4. **T2** does **credit**(**C**, 100) => **ok!** ($29 \geq 10$); **C** gets version **29**
5. **T1** reads **C** => **FAIL!** ($27 < 29$); **T1** aborts
6. **T2** commits; **T1** restarts, gets version **30**...

Advantages of TSO

- Deadlock free.
- Can allow more concurrency than 2PL.
- Can be implemented in a decentralized fashion.
- Can be augmented to distinguish reads & writes
 - objects have read version **R** & write version **W**.

```
READ(O, T) {  
    if(V(T) < W(O)) abort;  
    // do actual read  
    R(O) := MAX(V(T), R(O));  
}
```

Only safe to read if no-one wrote "**after**" us

R(O) holds version of *latest* transaction to read

Unsafe to write if **later** txaction has read value

```
WRITE(O, T) {  
    if(V(T) < R(O)) abort;  
    if(V(T) < W(O)) return;  
    // do actual write  
    W(O) := V(T);  
}
```

But if **later** txaction wrote it, just skip write (he won!).

However...

- TSO needs a rollback mechanism (like 2PL)
- TSO does not provide strict isolation:
 - Hence subject to cascading aborts
 - (Can provide strict TSO by locking objects when access is granted – still remains deadlock free if can abort).
- TSO decides *a priori* on one serialisation
 - Even if others might have been possible.
- And TSO does not perform well under contention
 - Will repeatedly have transactions aborting & retrying & ...
- In general, TSO is a good choice for **distributed systems** [decentralized management] where conflicts are rare.

Optimistic concurrency control

- **OCC** is an alternative to 2PL or TSO
- **Optimistic** since assume conflicts are rare
 - Execute transaction on a **shadow** [copy] of the data
 - On commit, check if all “OK”; if so, apply updates; otherwise discard shadows & retry.
- “OK” means:
 - All shadows read were **mutually consistent**, and
 - No one else has committed “later” changes to any object that we are hoping to update.
- Advantages: no deadlock, no cascading aborts
 - And “rollback” comes pretty much for free!
- Key idea: when ready to commit, search for a **serialisable order** that accepts the transaction.

Implementing OCC (1)

- All objects are tagged with version/generation numbers
 - e.g. the **Validation timestamp** of the transaction which most recently wrote its updates to that object
 - Nominally stored with the object, but possibly held as a validator data structure.
- Many threads execute transactions
 - When wish to read any object, take a shadow copy, and take note of the version number
 - If wish to write: edit the shadows (perhaps as held as html data in hidden web forms while booking a multi-part holiday)
- When a thread/customer want to finally commit a transaction, it submits the edited shadows to a **validator**.
- **Validator** nominally single-threaded (but parallel and distributed exist ...).
- **Validator** could work on a batch of submissions at once, finding an optimal, non-conflicting subset to commit with retries requested for the remainder.

Implementing OCC (2)

- **NB: There are many approaches following this basic technique.**
- Various efficient schemes for shadowing
 - eg. write buffering, page-based copy-on-write.
- All complexity resides in the two-step validator that must reflect a serialisable commit order in its ultimate side effects.
- **Read validation:**
 - Must ensure that all versions of data read by **T** (all shadows) were valid at some particular time **t**
 - This becomes the tentative **start time** for **T**
- **Serialisability validation:**
 - Must ensure that there are **no conflicts** with any **committed transactions** which have a **later start time**.
- Optimality matching:
 - For a batch, must choose a serialisation that commits as many as possible, possibly weighted on other heuristic, such as success for those rejected last attempt.

OCC Example (1)

- A log of the most recent validated transactions, with their timestamps etc

Transaction	Validation Timestamp	Objects Updated	Writeback Done?
T5	10	A, B, C	Yes
T6	11	D	Yes
T7	12	A, E	No

- The versions of the objects are as follows:
 - T7** has started, but not finished; writingback
 - (**A** has been updated, but not **E**)

Object	Version
A	12
B	10
C	10
D	11
E	9

What will happen if we now start a new transaction **T8** on {**B**, **E**} before **T7** writes back **E**?

OCC example (2)

- Consider **T8**: { **updates(B)**, **updates(E)** };
- **T8** executes and makes shadows of **B** & **E**
 - Records timestamps: **B@10**, **E@9**
 - When done, **T8** submits for validation
- Phase 1: read validation
 - Check shadows are part of a consistent snapshot
 - Latest committed start time is 11 = OK (10, 9 < 11)
- Phase 2: serializability validation
 - Check **T8** against all later transactions (here, **T7**)
 - **Conflict detected!** (**T7** updates **E**, but **T8** read old **E**)

Looking at log: have other transactions interfered with **T8**'s inputs?

Looking at log: would committing **T8** invalidate other now-committed transactions?

Issues with OCC

- Preceding example uses a simple validator
 - Possible it will abort even when don't need to,
 - (e.g. can search for a 'better' start time).
- In general OCC can find more serializable schedules than TSO
 - Timestamps assigned after the fact, and taking the actual data read and written into account.
 - e.g. both stored 27, value-based conflict detection easy to deploy.
- However OCC is not suitable when high conflict rate
 - Can perform lots of work with 'stale' data => wasteful!
 - Starvation possible if conflicting set continually retries,
 - Will the transaction system always make progress?

Isolation & Concurrency: Summary

- **2PL** explicitly locks items as required, then releases
 - Guarantees a serializable schedule
 - Strict 2PL avoids cascading aborts
 - Can limit concurrency & prone to deadlock
- **TSO** assigns timestamps when transactions start
 - Cannot deadlock, but may miss serializable schedules
 - Suitable for distributed/decentralized systems.
- **OCC** executes with shadow copies, then validates
 - Validation assigns timestamps when transactions end
 - Lots of concurrency & admits many serializable schedules
 - No deadlock but potential livelock when contention is high.
- Differing tradeoffs between **optimism**, **concurrency**, but also potential **starvation**, **livelock**, and **deadlock**.
- Ideas like TSO/OCC will recur in Distributed Systems.

Summary & Next Time

- History graphs; good (and bad) schedules
- Isolation vs. strict isolation; enforcing isolation
- Two-phase locking; rollback
- Timestamp ordering (TSO)
- Optimistic concurrency control (OCC)
- Isolation and concurrency summary
- Next time:
 - Transactional durability: crash recovery and logging,
 - Lock-free programming,
 - Transactional memory (if time permits).

Concurrent systems

Lecture 8a: Durability & crash recovery.

Lecture 8b: lock-free programming & transactional memory.

Dr Martin Kleppmann

(Slides for lectures 1–8 by Dr Robert Watson,
Dr Steven Hand, Dr David Greaves, and others)

This time

- Transaction durability: crash recovery, logging
 - Write-ahead logging
 - Checkpoints
 - Recovery and Rollback
- Advanced topics (as time permits)
 - Lock-free programming
 - Transactional memory

Crash Recovery & Logging

- Transactions require **ACID** properties
 - So far have focused on **I** (and implicitly **C**).
- How can we ensure Atomicity & Durability?
 - Need to make sure that a transaction is always done entirely or not at all (*i.e. make sure rollback happens*).
 - Need to make sure that a transaction reported as committed remains so, even after a crash.
- Consider for now a **fail-stop** model:
 - If system crashes, all in-memory contents are lost
 - Data on disk, however, remains available after reboot.

The small print: we must keep in mind the limitations of **fail-stop**, even as we assume it. Failing hardware/software do weird stuff. Pay attention to hardware price differentiation.

Semantics of secondary store

- Most computers have volatile primary (DRAM) and non-volatile secondary storage (tape, SSD, disks, USB-sticks).
- Systems rely (perhaps falsely) on an idealised, erasure-channel, abstract semantics for secondary storage:

```
type blkaddress_t = integer 0 to 2^19-1 // say
type block_t = array [0..4095] of integer 0 to 255
method write : blkaddress_t * block_t -> unit
method read  : blkaddress_t -> block_t option
method trim  : blkaddress_t -> unit           // Forget a block (SSD)
method fsync : unit -> unit                 // Blocking flush
```

- It is critical that read returns an option: a failed write results in either exactly the previously written data or None
- a garbled mixture of new, old and random bits is never returned.

Using persistent (non-volatile) storage

- Simplest “solution”: write all updated objects to disk on commit, read back on reboot
 - Doesn’t work, since crash could occur *between* writes
 - Can fail to provide transaction Atomicity and/or Consistency.
- Instead split update into two stages:
 1. Write proposed updates to a **write-ahead log**
 2. Write actual updates.
- Crash during #1 => no actual updates done;
- Crash during #2 => use log to redo, or undo.
- Recall transactions can also abort (and cascading aborts), so log can help undo the changes made.

Write-ahead logging

- **Log**: an ordered, append-only file on disk (aka journal).
- Contains entries like **<txid, obj, op, old, new>**
 - ID of transaction, object modified, (optionally) the operation performed, the old value **and** the new value.
 - This means we can both “roll forward” (**redo operations**) and “rollback” (**undo operations**).
- When persisting a transaction to disk:
 - First log a special entry **<txid, START>**,
 - Next log a number of entries to describe operations,
 - Finally log another special entry **<txid, COMMIT>**.
- We build composite-operation atomicity from fundamental atomic operation: the **single-sector write**.
 - Much like building high-level primitives over **LL/SC** or **CAS**!

Using a write-ahead log (WAL)

- When executing transactions, perform updates to objects in memory with **lazy write back**
 - *I.e.* the OS will normally delay all disk writes to improve efficiency.
- Golden rule: write log records **before** corresponding data.
- But when wish to **commit** a transaction, must first **synchronously** flush a commit record to the log
 - Assume there is a `fsync()` or `fsyncdata()` operation or similar which allows us to force data out to disk.
 - Only report transaction committed after `fsync()` returns.
- Can improve performance by delaying flush until we have a number of transaction to commit – **batching**.
 - Hence at any point in time we have some prefix of the write-ahead log on disk, and the rest in memory.

The Big Picture

Log has saved x is now 2, but on-disk x is still 1.

RAM acts as a **cache** of disk
(e.g. no in-memory copy of z)

Log conceptually infinite,
and spans RAM & Disk

RAM

Object Values

x = 3
y = 27

Log Entries

T3, START
T2, ABORT
T2, y, 17, 27
T1, x, 2, 3

Disk

Object Values

x = 1
y = 17
z = 42

Log Entries

T2, z, 40, 42
T2, START
T1, START
T0, COMMIT
T0, x, 1, 2
T0, START

Log Write Order

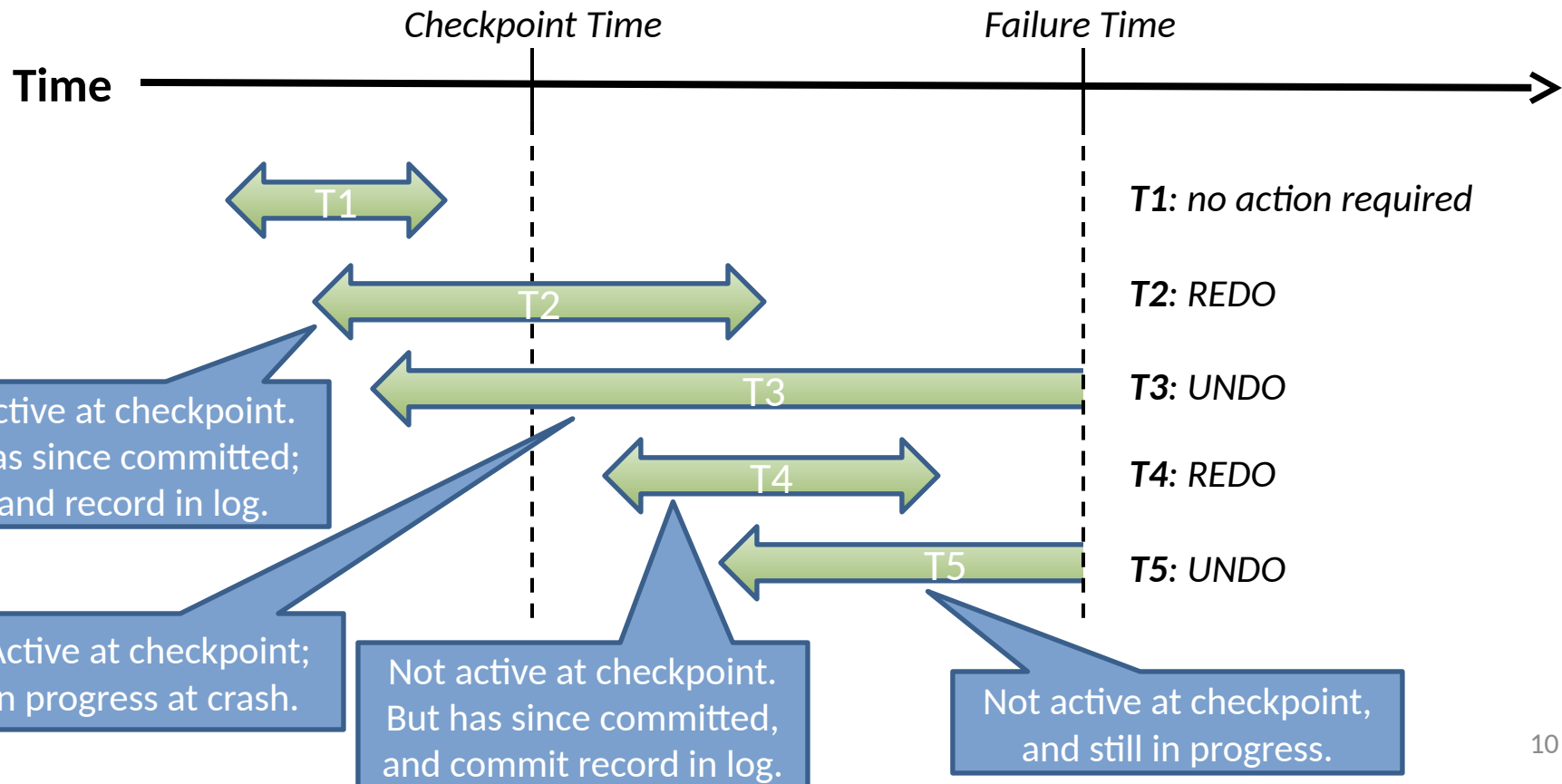
On-disk values may be **older versions** of objects
(e.g. x) – or **new uncommitted values** as long as
the on-disk log describes rollback (e.g. z)

Checkpoint Approach

- As described, log will get infeasibly/very long
 - And need to process every entry in log to recover.
- Better to periodically write a **checkpoint**
 1. Flush all current in-memory log records to disk.
 2. Write a special **checkpoint record** to log with a list of **active transactions**
(pointers to earliest undo/redo log entries that must be searched during recovery)
 3. Flush all 'dirty' objects (i.e. ensure object values on disk are up-to-date)
 4. Atomic (single sector) write of location of new checkpoint record to a special, well-known place in persistent store (disk). Truncate log, discarding no longer needed parts (perhaps by the same action).
- Atomic checkpoint location write supports crash during recovery.

Checkpoints and recovery

- Key benefit of a checkpoint is it lets us focus our attention on possibly-affected transactions



Recovery algorithm

- Initialize undo set **U** = { set of active txactions }
- Also have redo set **R**, initially empty.
- Walk log forward as indicated by checkpoint record:
 - If see a **START record**, add transaction to **U**
 - If see a **COMMIT record**, move transaction from **U**->**R**
- When hit end of log, perform undo:
 - Walk backward and undo all records for all **Tx** in **U**
- When reach checkpoint timestamp again, Redo:
 - Walk forward, and re-do all records for all **Tx** in **R**
- After recovery, we have effectively checkpointed
 - On-disk store is consistent, so can (generally) **truncate** the log.

The order in which we apply **undo/redo records** is important to properly handle cases where multiple transactions touch the same data.

Write-ahead logging: Assumptions

- What can go wrong writing commits to disk?
- Even if **sector writes are atomic**:
 - All affected objects may not fit in a single sector, large objects may span multiple sectors
 - Trend towards copy-on-write, rather than journalled, filesystems (btrfs etc).
 - Many of the problems seen with in-memory commit (ordering and atomicity) apply to disks as well!
- Contemporary disks may not be entirely honest about sector size and atomicity
 - E.g., unstable write caches to improve efficiency
 - E.g., larger or smaller sector sizes than advertised
 - E.g., non-atomicity when writing to mirrored disks (RAID).
- These assume **fail-stop** – not true for some media (SSD?)

Transactions: Summary

- Standard mutual exclusion techniques not programmer friendly when dealing with >1 object
 - intricate locking (& lock order) required, or
 - single coarse-grained lock, limiting concurrency
- Transactions allow us a better way:
 - potentially many operations (reads and updates) on many objects, but should execute as if **atomically**
 - underlying system deals with providing **isolation**, allowing safe concurrency, and even fault tolerance!
- Appropriate only if operations are “transactional”
 - E.g., **discrete events in time**, as must commit to be visible
- Transactions are used both in databases and filesystems.

Advanced Topics

- Will briefly look at two advanced topics
 - lock-free data structures, and
 - transactional memory
- Then, next time, Distributed Systems

Lock-free programming

- What's wrong with locks?
 - Difficult to get right (if locks are fine-grained)
 - Don't scale well (if locks too coarse-grained)
 - Don't compose well (deadlock!)
 - Poor cache behavior (and convoying <https://davekilian.com/lock-convoy.html>)
 - Priority inversion
 - And can be expensive
- **Lock-free programming** involves getting rid of locks ... but not at the cost of safety!
- Recall **TAS**, **CAS**, **LL/SC** from our early lecture: what if we used them to implement something other than locks?

Memory API Assumptions

- We have a cache-consistent shared-memory system (and we understand the sequential consistency model)

- Low-level (assembly instructions) include:

```
val  = read(addr);           // atomic read from memory
(void) write(addr, val);      // atomic write to memory
done = CAS(addr, old, new);   // atomic compare-and-swap
```

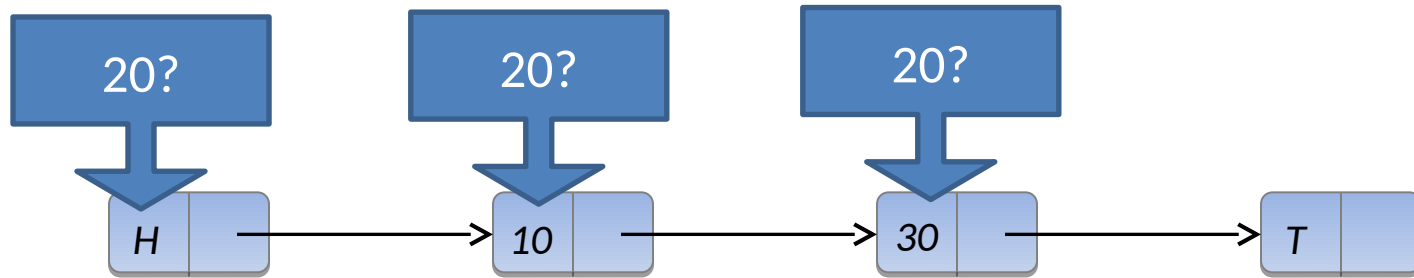
- Compare-and-Swap (CAS) is **atomic**
 - Reads value of addr ('**val**'), compares with '**old**', and updates memory to '**new**' iff **old==val** -- without interruption.
 - Something like this instruction common on most modern processors (e.g. **cmpxchg** on x86 – or **LL/SC** on RISC)
- Typically used to build spinlocks (or mutexes, or semaphores, or whatever...)

Lock-free approach

- Directly use **CAS** to update shared data
- For example, consider a lock-free linked list of integers
 - list is singly linked, and sorted
 - Use **CAS** to update pointers
 - Handle **CAS** failure cases (i.e., races)
- Represents the 'set' abstract data type, i.e.
 - **Find**: `int -> bool`
 - **Insert**: `int -> bool`
 - **Delete**: `int -> bool`
- Insert/delete return values indicate if operation failed, requiring retry (typically in a loop).
- Assumption: hardware supports atomic operations on pointer-size types.
- Assumption: Full sequential consistency (or fences used as needed).

Searching a sorted list

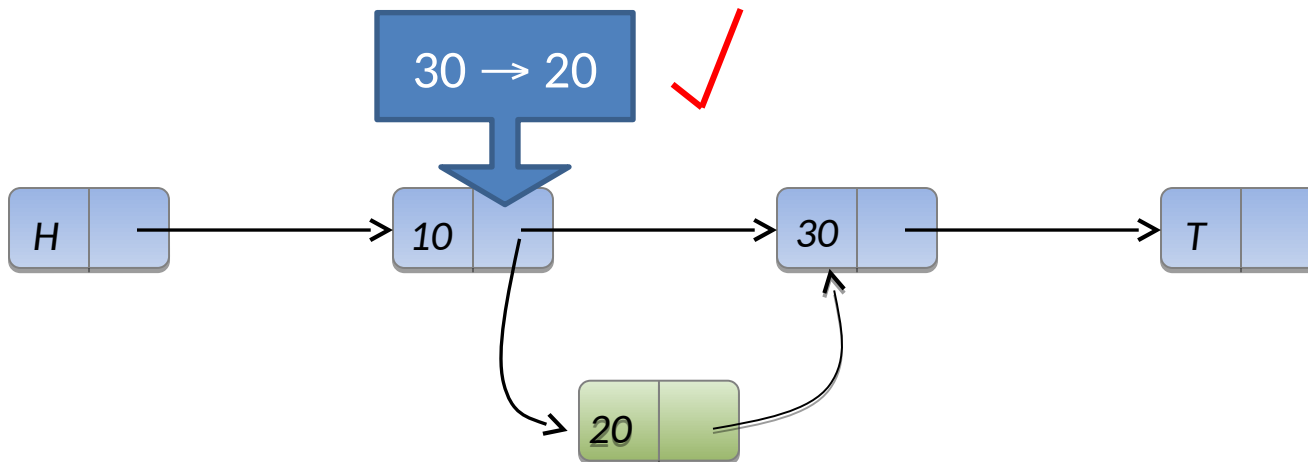
- find(20):



find(20) -> false

Inserting an item with a simple store

- `insert(20):`

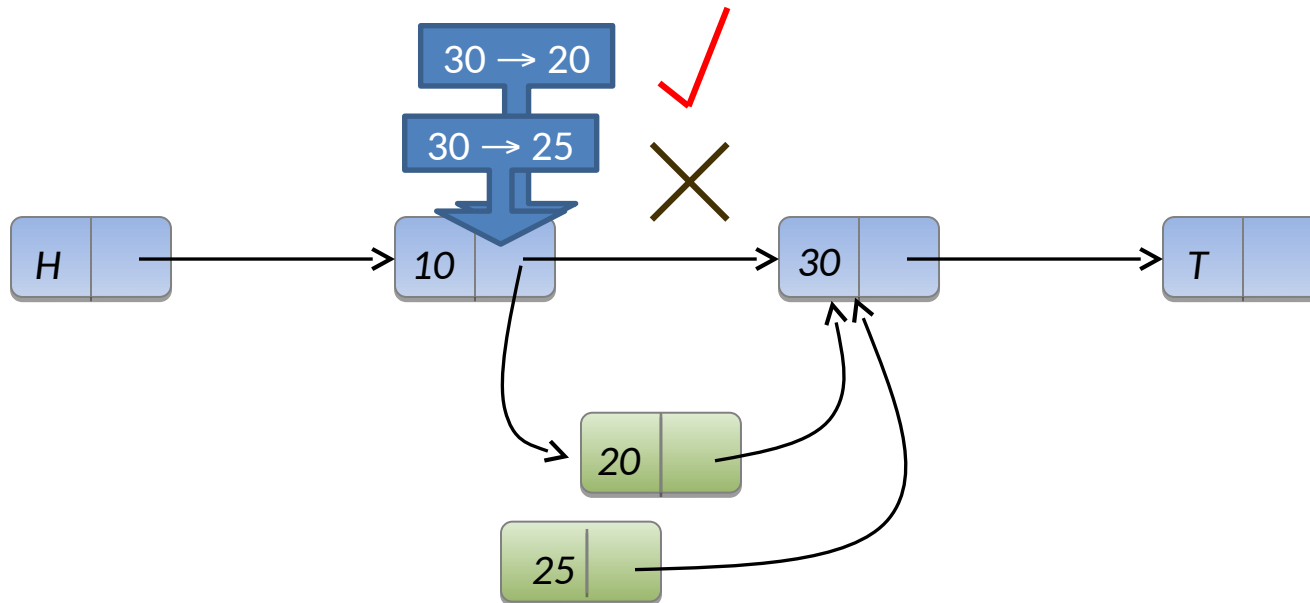


`insert(20) -> true`

Inserting an item with CAS

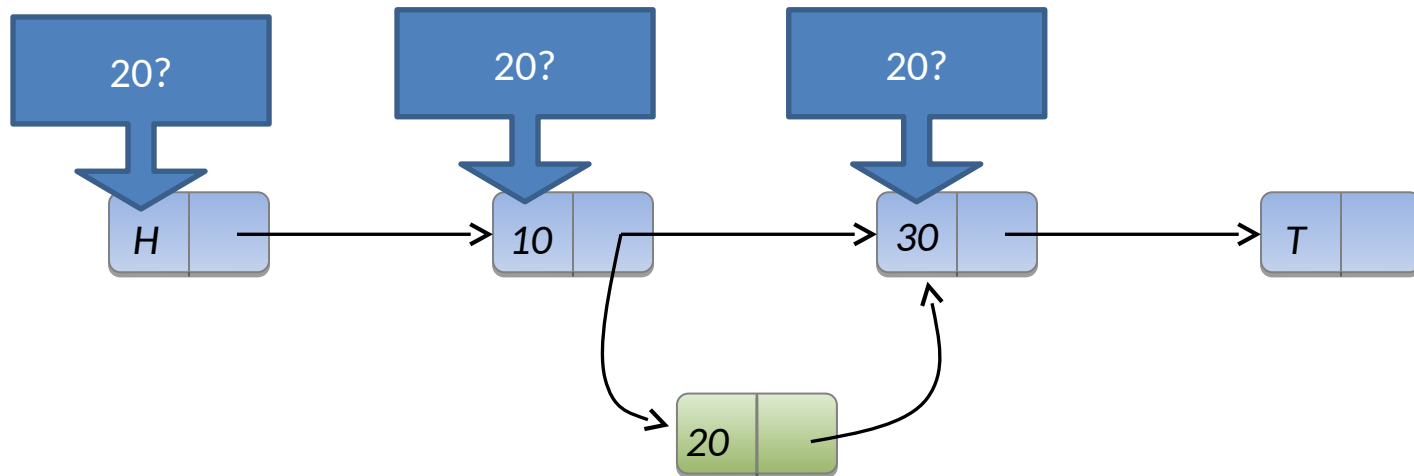
- insert(20):

- insert(25):



Concurrent find+insert

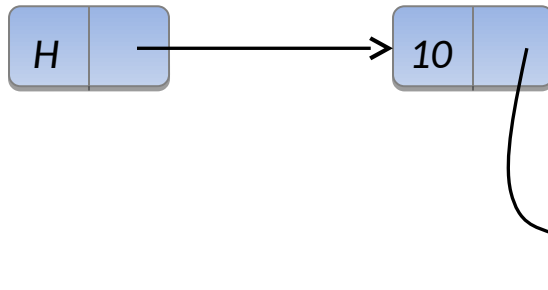
- find(20) -> false
- insert(20) -> true



Concurrent find+insert

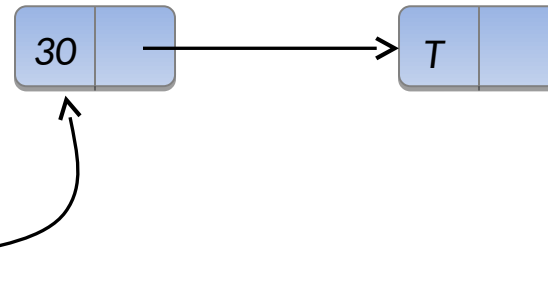
- `find(20) -> false`

This thread saw 20
was not in the set...



- `insert(20) -> true`

...but this thread
succeeded in putting
it in!



- Is this a correct implementation of a set?
- Should the programmer be surprised if this happens?
- What about more complicated mixes of operations?

Linearisability

- As with transactions, we return to a conceptual model to define correctness:
 - a lock-free data structure is ‘correct’ if all changes (and return values) are consistent with some **serial** view: we call this a **linearisable schedule**.
 - Lock-free structure and code must be designed to tolerate all possible thread interleaving patterns that may occur.
- Hence in the previous example, we are always ok:
 - Either the insert() or the find() can be deemed to have occurred first.
- Gets a lot more complicated for more complicated data structures & operations – (eg. money conservation in the credit/debit/xfer example)
- On some hardware, atomic primitives do more than just provide atomicity:
 - Eg. CAS may embody a memory fence for sequential consistency (observable memory ordering).
 - LL/SC may not and so explicit “happens-before” load and stores fences may be needed in the code.
 - Lock-free structures must take this into account as well.

(S/W) Transactional Memory (TM)

- Based on optimistic concurrency control.

- Instead of:

```
lock(&sharedx_mutex);  
sharedx[i] *= sharedx[j] + 17;  
unlock(&sharedx_mutex);
```

- ▶ Use:

```
atomic {  
    sharedx[i] *= sharedx[j] + 17;  
}
```

- ▶ Has “obvious” semantics, i.e. all operations within block occur as if atomically
- ▶ Transactional since under-the-bonnet it looks like:

```
do { txid = tx_begin(&thd, sharedx);  
    sharedx[i] *= sharedx[j] + 17;  
} while !(tx_commit(txid));
```

TM advantages

- Simplicity:
 - Programmer just puts **atomic** { } around anything they want to occur in isolation.
 - Fine-grain concurrency is possible without manual partition of variables or array locations into locking groups.
- Composability:
 - Unlike locks, **atomic** { } blocks nest, e.g.:

```
credit(a, x) = atomic {  
    setbal(a, readbal(a) + x);  
}  
debit(a, x) = atomic {  
    setbal(a, readbal(a) - x);  
}  
transfer(a, b, x) = atomic {  
    debit(a, x);  
    credit(b, x);  
}
```

TM advantages

- Cannot deadlock:
 - No locks, so don't have to worry about locking order
 - (Though may get live lock if not careful)
- No races (mostly):
 - Cannot forget to take a lock (although you can forget to put **atomic { }** around your critical section ;-))
- Scalability:
 - High performance possible via OCC
 - No need to worry about complex fine-grained locking
- There remains a simplicity vs. performance tradeoff
 - Too much **atomic { }** and implementation can't find concurrency. Too little, and errors arise from poor interleaving.

TM is very promising...

- Essentially does 'ACI' but no D
 - no need to worry about crash recovery
 - can work entirely in memory
 - can be implemented in HLL, VM or hardware (S/W v H/W TM)
 - x86 xbegin/xend instructions
- Last decade, both x86 and Arm offered direct support for transactions using augmented cache protocols
 - ... And promptly withdrawn in errata
 - Now back on the street again
 - Security vulnerabilities (timing attacks and the like)?
- But not a panacea
 - Contention management can get ugly (lack of parallel speedup)
 - Difficulties with irrevocable actions / side effects (e.g. I/O)
 - Still working out exact semantics (type of atomicity, handling exceptions, signalling, ...)

Concurrent systems: summary

- Concurrency is essential in modern systems
 - overlapping I/O with computation,
 - exploiting multi-core,
 - building distributed systems.
- But throws up a lot of challenges
 - need to ensure safety, allow synchronization, and avoid issues of liveness (deadlock, livelock, ...)
- Major risks of bugs and over-engineering
 - generally worth running as a sequential system first,
 - too much locking leads to too much serial execution,
 - and worth using existing libraries, tools and design patterns rather than rolling your own!

Summary + next time

- Transactional durability: crash recovery and logging
 - Write-ahead logging; checkpoints; recovery.
- Advanced topics
 - Lock-free programming
 - Transactional memory.
- Next time: Distributed Systems!