Concurrent systems

Lecture 4: Safety and liveness

Dr Robert N. M. Watson

Reminder from last time

- Alternatives to simple semaphores/locks:
  - Conditional critical regions (CCRs)
  - Monitors and 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)

Progress turns out to be quite difficult, in large part because of concurrency primitives themselves, and is the topic of this lecture

- Se t
  co
- A
  - subtle minor differences can be dangerous
  - require care to avoid bugs

This time

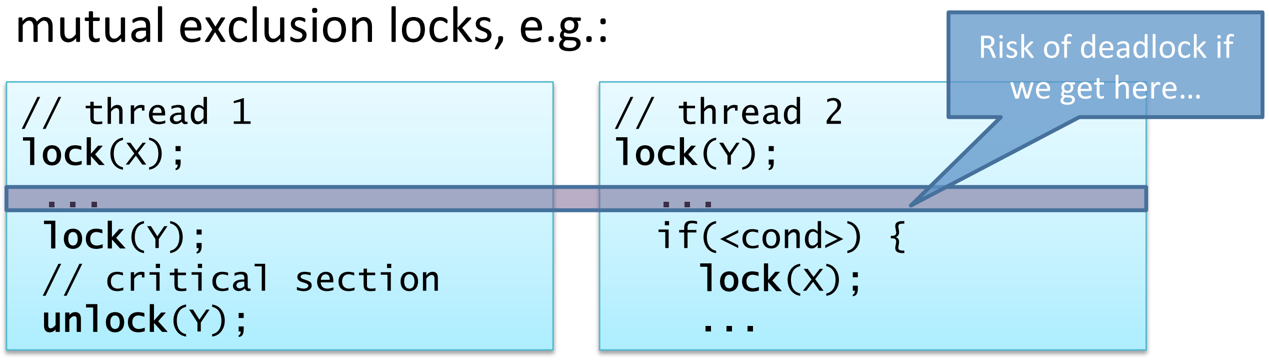
- Liveness properties
- Deadlock
  - Requirements
  - Resource allocation graphs
  - Detection
  - Prevention – the Dining Philosophers
  - Recovery
- Priority inversion
- Priority inheritance

Liveness properties

- From a theoretical viewpoint must ensure that we eventually make progress, i.e. want to avoid
  - **Deadlock** (threads sleep waiting for each other), 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 :-(

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.:



Risk of deadlock if we get here…

```
// thread 1
lock(X);
 ...
 lock(Y);
 // critical section
 unlock(Y);
```

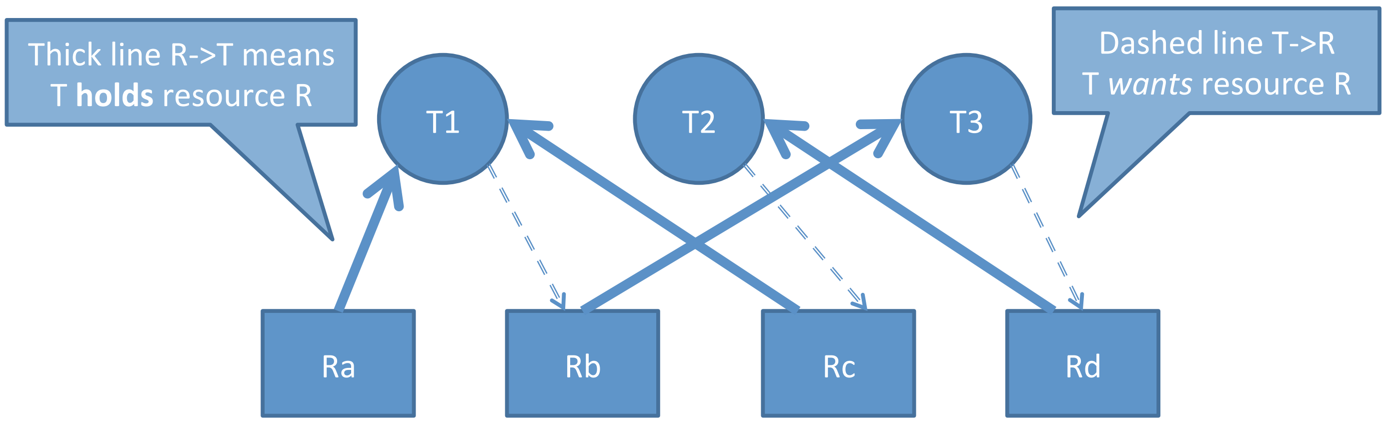
```
// thread 2
lock(Y);
 ...
 if(<cond>) {
   lock(X);
   ...
```

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 get **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 be true to get deadlock
  – But most modern systems always satisfy 1, 2, 3

Resource allocation graphs

- Graphical way of thinking about deadlock
- Circles are threads (or processes), boxes are single owner resources (e.g. mutual exclusion locks)
- A **cycle** means we (will) have deadlock

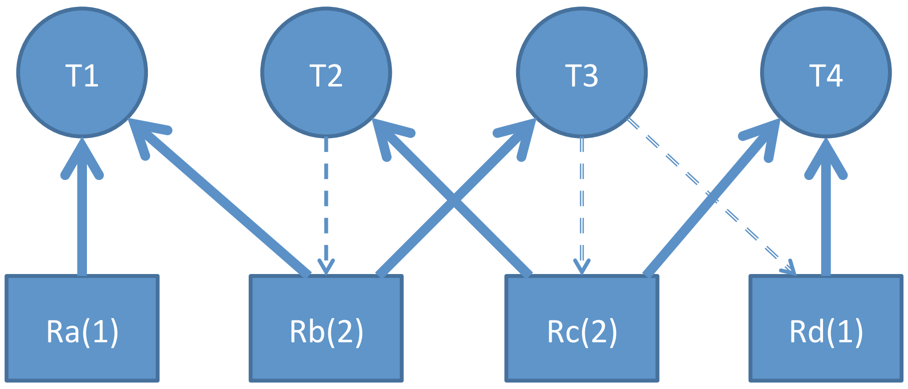


Thick line R->T means T **holds** resource R

Dashed line T->R T *wants* resource R

Resource allocation graphs

- 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 have* deadlock, e.g.



Dealing with deadlock

1. Ensure it never happens
   - Deadlock prevention
   - Deadlock avoidance (Banker's Algorithm)
2. Let it happen, but recover
   - Deadlock detection & recovery
3. Ignore it!
   - The so-called "Ostrich Algorithm" ;-)
   - i.e. let the programmer fix it
   - Very widely used in practice!

Deadlock prevention

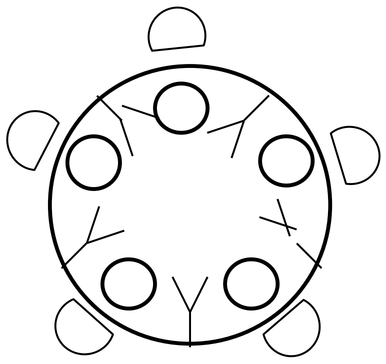
1. **Mutual Exclusion**: resources have bounded #owners
   - Could always allow access… but probably unsafe ;-(
   - However can help e.g. by using MRSW locks
2. **Hold-and-Wait**: can get **R**x and wait for **R**y
   - Require that we request all resources simultaneously; deny the request if *any* 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 (tho 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 eg 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];
}
```



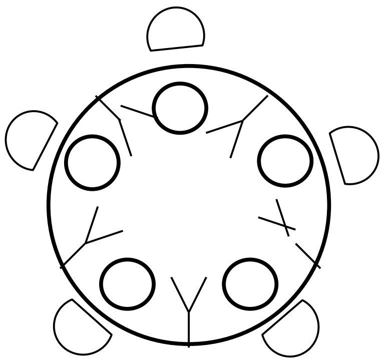
- Possible for everyone to acquire ‘left’ fork (i)
  - Q: what happens if we swap order of **signal**()s?

Example: Dining Philosophers

- (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 avoidance

- Prevention aims for deadlock-free “by design”
- **Deadlock avoidance** is a dynamic scheme:
  - Assume we know maximum possible resource allocation for every process / thread
  - 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 – see textbooks
  - Not really useful in general as need *a priori* knowledge of #processes/threads, and their max resource needs

Deadlock detection

- A dynamic scheme which attempts to determine if deadlock exists
- 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 $O(n^2)$
- More difficult if have multi-instance resources

Deadlock detection

- Have *m* distinct resources and *n* threads
- **V**[0:m-1], vector of available resources
- **A**, the *m* x *n* resource allocation matrix, and **R**, the *m* x *n* (outstanding) request matrix
  - $\mathbf{A}_{i,j}$ is the number of objects of type *j* owned by *i*
  - $\mathbf{R}_{i,j}$ is the number of objects of type *j* needed by *i*
- Proceed by 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 fulfill thread *i*'s request R*i*, then it will run to completion and release held resources for other threads to allocate.

Deadlock detection algorithm

- 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**
- Select an unmarked row *i* of **A** s.t. **R**[*i*] <= **W**
  - (i.e. find a thread who’s request can be satisfied)
  - Set **W** = **W** + **A**[i]; mark row *i*, and repeat
- Terminate when no such row can be found
  - Unmarked rows (if any) are in the deadlock set

W[] describes any free resources at start, **plus** any resources released by a hypothesized sequence of satisfied threads freeing and terminating

Deadlock detection example 1

- Five threads and three resources (none free)

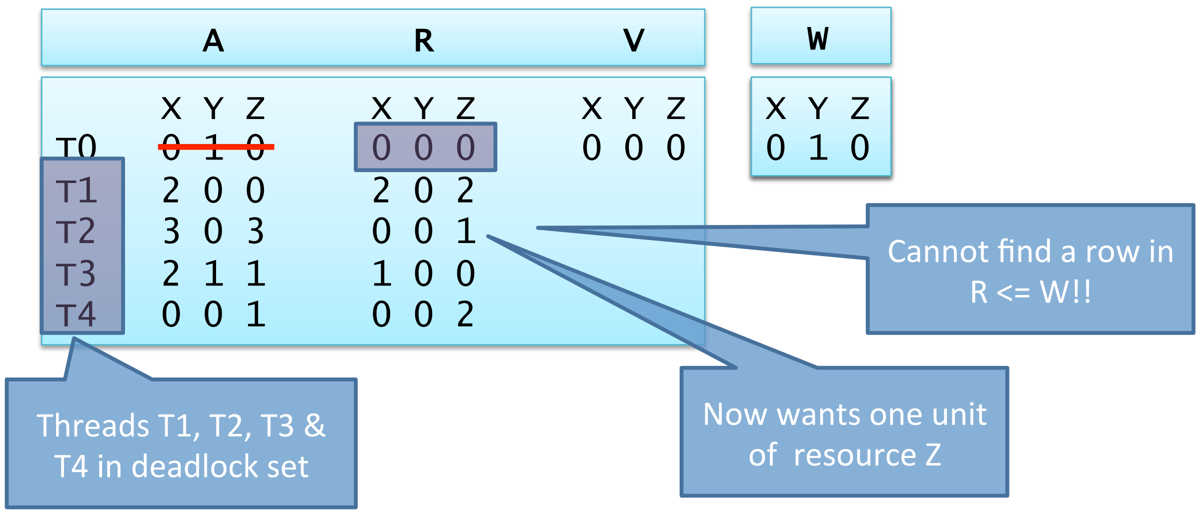
	A	R	V
	X Y Z	X Y Z	X Y Z
T0	~~0 1 0~~	0 0 0	0 0 0
T1	~~2 0 0~~	2 0 2	
T2	~~3 0 3~~	0 0 0	
T3	~~2 1 1~~	1 0 0	
T4	~~0 0 1~~	0 0 2	

W
X Y Z
7 2 5

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

Deadlock detection example 2

- 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	0	1	0
T1	2	0	0	2	0	2						
T2	3	0	3	0	0	1						
T3	2	1	1	1	0	0						
T4	0	0	1	0	0	2						

Cannot find a row in R <= W!!

Now wants one unit of resource Z

Threads T1, T2, T3 & T4 in deadlock set

- One minor tweak to T2’s request vector…

Deadlock recovery

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

Livelock

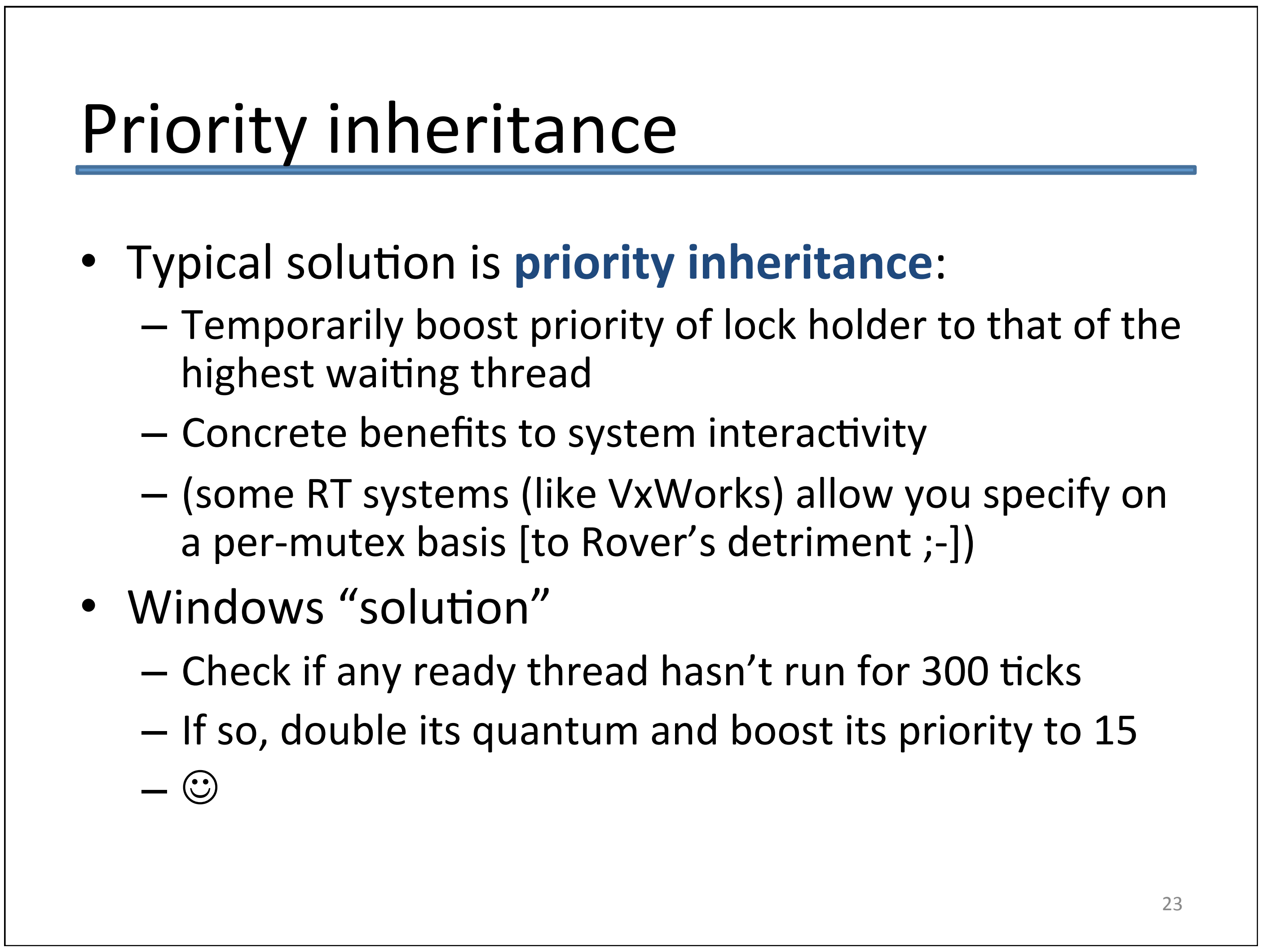
- Deadlock is at least ‘easy’ to detect by humans
  - System basically blocks & stops making any progress
- Livelock is less easy to detect as threads continue to run… but do nothing useful
- Often occurs from trying to be clever, e.g.:

```
// thread 1
lock(X);
 ...
 while (!trylock(Y)) {
   unlock(X);
   yield();
   lock(X);
 }
 ...
```

```
// thread 2
lock(Y);
  ...
  while(!trylock(X)) {
    unlock(Y);
    yield();
    lock(Y);
  }
  ...
```

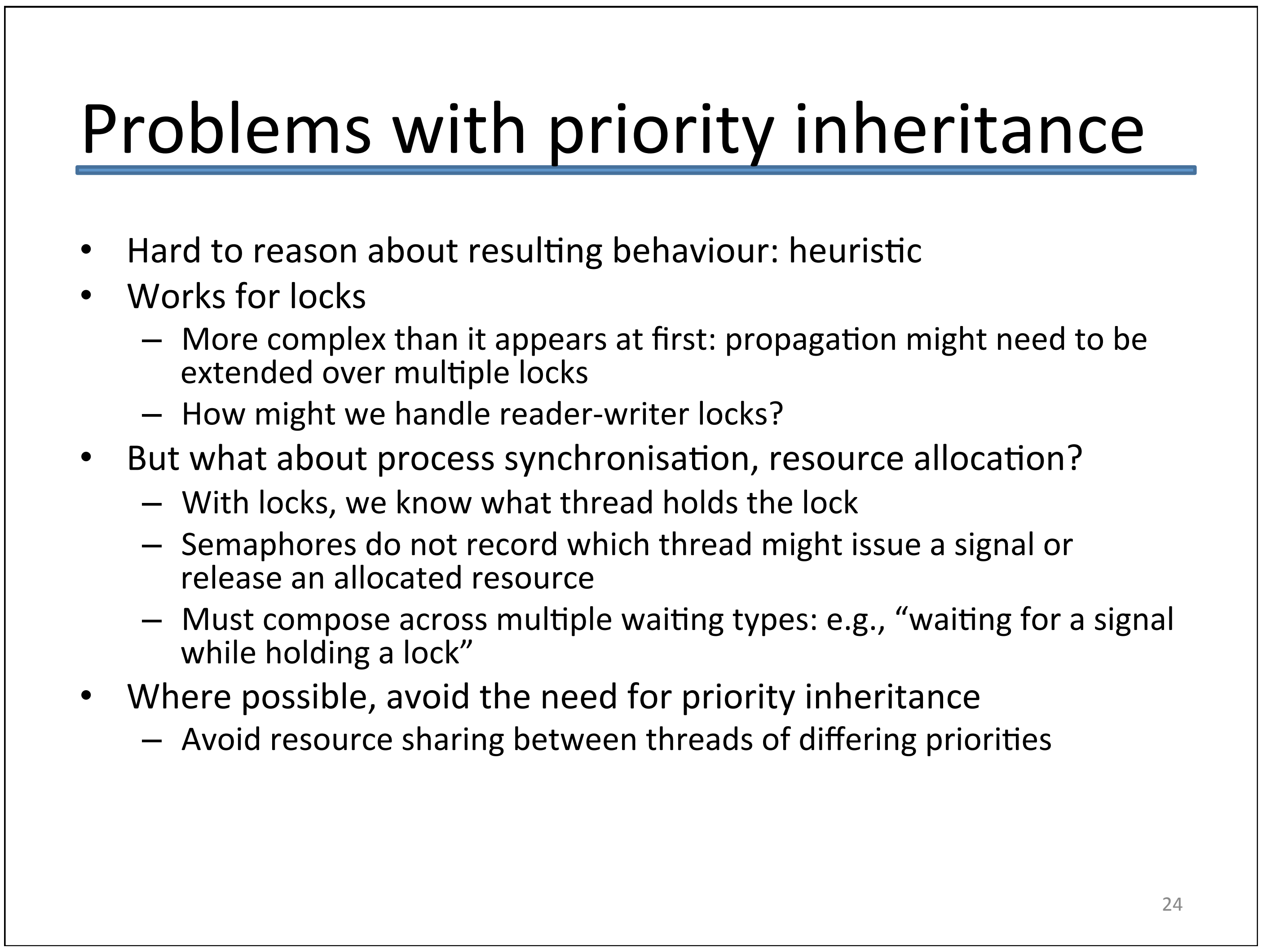
Priority inversion

- Another liveness problem…
  - Due to interaction between locking and scheduler
- Consider three threads: T1, T2, T3
  - T1 is high priority, T2 low priority, T3 is medium
  - T2 gets lucky and acquires lock L…
  - … T1 preempts him and sleeps waiting for L…
  - … then T3 runs, preventing T2 from releasing L!
- This is not deadlock or livelock
  - But not very desirable (particularly in RT systems)



Priority inheritance

- Typical solution is **priority inheritance**:
  - Temporarily boost priority of lock holder to that of the highest waiting thread
  - 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 at first: propagation might need to be extended over multiple locks
  - How might we handle reader-writer locks?
- But what about process synchronisation, resource 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 resource sharing between threads of differing priorities

Summary + next time

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

- Next time:
  - Concurrency without shared data
  - Active objects; message passing
  - Composite operations; transactions
  - ACID properties; isolation; serialisability