Concurrent Systems
8L for Part IB

Handout 2

Dr Robert Watson

Event Counts & Sequencers

- Alternative synchronization scheme (1979)
- **Event Counts**: a special type of variable
  - Essentially an increasing integer, initialized to zero
- Supports three operations:
  - int **advance**(ec) { ec.val++; return ec.val; }
  - int **read**(ec) { return ec.val; }
  - void **await**(ec, v) { sleep until ec.val >= v; return}
- Can be somewhat lazy
  - **read**() can provide a stale value
  - **await**() can be a little “late”, i.e. (ec.val-v) can be > 0

Event Counts: Producer-Consumer

```
int buffer[N]; int in = 0, out = 0;
CEV = new EventCount();  // counts no of "consumptions"
PEV = new EventCount();  // counts no of "productions"
```

```
// producer thread
while(true) {
  item = produce();
  await(CEV, (in-N)+1);
  buffer[in % N] = item;
  in = in + 1;
  advance(PEV);
}
```

```
// consumer thread
while(true) {
  await(PEV, out+1);
  item = buffer[out % N];
  out  = out + 1;
  advance(CEV);
  consume(item);
}
```

- Very similar to semaphore solution (although free running counters … problem?)
- Again, no **explicit** mutual exclusion

Sequencers

- To complete the picture, add **Sequencers**
  - Special type of variable: an integer initialized to 0
- Has just one operation:
  - int **ticket**(seq) { v = seq.val; seq.val++; return v; }
  - atomically produces a unique (increasing) value
- Can use an event count & a sequencer together to implement a mutual exclusion lock:

```
LOCK(L) {
  turn = ticket(L.SQ);
  await(L.EV, turn);
}
```

```
UNLOCK(L) {
  advance(L.EV);
}
```

Generalized Producer-Consumer

```
int buffer[N];
PEV = new EventCount(); CEV = new EventCount();
PSQ = new Sequencer();  CSQ = new Sequencer();
```

```
// producer threads
while(true) {
  item = produce();
  turn = ticket(PSQ);
  await(PEV, turn);
  await(CEV, (turn-N)+1);
  buffer[turn % N] = item;
  advance(PEV);
}
```

```
// consumer threads
while(true) {
  turn = ticket(CSQ);
  await(CEV, turn);
  await(PEV, turn+1);
  item = buffer[turn % N];
  advance(CEV);
  consume(item);
}
```

- Safe concurrent access by any { producer , consumer } pair
- A single **advance**() invocation provides both mutual exclusion & condition synchronization

Event Counts & Sequencers: MRSW

```
WEV = new EventCount();  // counts no of updates (writes)
WSQ = new Sequencer();   // for writer mutual exclusion
REV = new EventCount();  // 'version' of data
```

```
// a writer thread
advance(REV);
turn = ticket(WSQ);
await(WEV, turn);
.. perform update to data
advance(WEV);
```

```
// a reader thread
do {
  v1 = read(REV);
  await(WEV, v1);
  .. read data
  v2 = read(REV);
} while(v1 != v2);
```

- Core of writer is mutual exclusion (WSQ, WEV)
- Q: why does reader need to **await**()?

Event Counts & Sequencers: Summary

- A different scheme than semaphores
  - Basic primitives are synchronization & ordering
  - (tho can be used to build mutual exclusion)
- Lazy semantics allow efficient implementation
  - Originally designed for multiprocessors
- Can lead to simpler [well, shorter] code…
  - But still pretty low-level and hard to use
  - (convince yourself all the examples are correct;-)
- A higher-level paradigm would be nice!

Conditional Critical Regions

- One early (1970s) effort was CCRs
  - Variables can be explicitly declared as 'shared'
  - Code can be tagged as using those variables, e.g.

```
shared int A, B, C;
region A, B {
    await( /* arbitrary condition */);
    // critical code using A and B
}
```

- Compiler automatically declares and manages underlying primitives for mutual exclusion or synchronization
  - e.g. wait/signal, read/await/advance, …
- Easier for programmer (c/f previous implementations)

CCR Example: Producer-Consumer

```
shared int buffer[N];
shared int in = 0; shared int out = 0;
```

```
// producer thread
while(true) {
  item = produce();
  region in, out, buffer {
    await((in-out) < N);
    buffer[in % N] = item;
    in = in + 1;
  }
}
```

```
// consumer thread
while(true) {
  region in, out, buffer {
    await((in-out) > 0);
    item = buffer[out%N];
    out  = out + 1;
  }
  consume(item);
}
```

- Explicit (scoped) declaration of critical sections
  - automatically acquire mutual exclusion lock on region entry
- Powerful **await**(): any evaluable predicate

CCR Pros and Cons

- On the surface seems like a definite step up
  - Programmer focuses on **variables** to be protected, compiler generates appropriate semaphores (etc)
  - Compiler can also check that shared variables are never accessed outside a CCR
  - (still rely on programmer annotating correctly)
- But **await**(<expr>) is problematic…
  - What to do if the (arbitrary) <expr> is not true?
  - very difficult to work out when it becomes true?
  - Solution was to leave region & try to re-enter: this is busy waiting, which is very inefficient…

Monitors

- **Monitors** are similar to CCRs (implicit mutual exclusion), but modify them in two ways
  - Waiting is limited to explicit **condition variables**
  - All related routines 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 invokes a monitor method, it will block (queue) if there is another thread active inside
  - Hence all methods within the monitor can proceed on the basis that mutual exclusion has been ensured

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

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

Shared variables can be initialized here

Condition Variables

- Mutual exclusion not always sufficient
  - e.g. may need to wait for a condition to occur
- Monitors allow condition variables
  - Explicitly declared & managed by programmer
  - 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;
}
broadcast(cv) {
   wake all threads queued on cv;
}
```

Monitor Producer-Consumer Solution?

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

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

If buffer is full (in==out+N), must wait for consumer

If buffer was full (in==out), signal the consumer

If buffer is empty (in==out), must wait for producer

If buffer **was** full before, signal the producer

Does this work?

- Depends on implementation of **wait**() & **signal**()
- Imagine two threads, T1 and T2
  - 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 (so-called "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 a queue **E** to enter 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 difficult 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, example on p14 is broken:
  - we **signal**() before incrementing in/out

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 is true when resume!
  - Other threads may have executed after the signal, but before you continue

Signal-and-Continue Example

- 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(notfull);

Monitors: Summary

- Structured concurrency control
  - groups together shared data and methods
  - (today we'd call this object-oriented)
- Considerably simpler than semaphores (or event counts), 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 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:
  - Linux kernel
  - POSIX pthreads (C/C++ API)
  - Java
  - C#

Example: Linux Kernel

- Kernel provides spinlocks & semaphores
  - – Spinlocks busy wait so only hold for short time
  - – (dynamically optimized out on UP kernels)

```
DEFINE_SPINLOCK(mylock);
spin_lock_irqsave(&mylock, flags);
// do stuff (not much!)
spin_lock_irqrestore(&mylock, flags);
```

- Gradual migration to mutexes – we'll see why shortly
- Also get *reader-writer* spinlock variants
  - allows many readers or a single writer
  - (mostly deprecated now in favor of RCU)

Example: pthreads

- 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

- 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);
```

Example: pthreads

- Barriers: explicit synchronization mechanism
  - Wait until all threads reach some point

```
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: Java [original]

- Synchronization inspired by monitors
  - Objects already encapsulate data & methods!
- Mesa-style, but no explicit condition variables

```
public class MyClass {
   //
   public synchronized void myMethod() throws ...{
     while(!condition)
        wait();
     // do stuff
     if(condition)
        notifyAll();
   }
}
```

- Java 5 provides many additional options…

Example: C#

- Very similar to Java, tho explicit arguments

```
public class MyClass {
   //
   public void myMethod() {
     lock(this) {
        while(!condition)
            Monitor.Wait(this);
        // do stuff
        if(condition)
        Monitor.PulseAll(this);
     }
   }
}
```

- Also provides spinlocks, reader-writer locks, semaphores, barriers, event synchronization, …

Concurrency Primitives: Summary

- Concurrent systems require means to ensure:
  - **Safety** (mutual exclusion in critical sections), and
  - **Progress** (condition synchronization)
- Seen spinlocks (busy wait); semaphores; event counts / sequencers; CCRs and monitors
- Almost all of these are still used in practice
  - subtle minor differences can be dangerous
  - require care to avoid bugs

Safety and Liveness

- Desirable properties for concurrent systems
  - **Safety**: bad things don't happen
  - **Liveness**: good things (eventually) happen
- Mutual exclusion is primarily about safety
  - Want to ensure two threads don't "collide" in terms of accessing shared data
- …but may have consequences for liveness too!
  - i.e. must ensure our program doesn't get stuck

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);
```

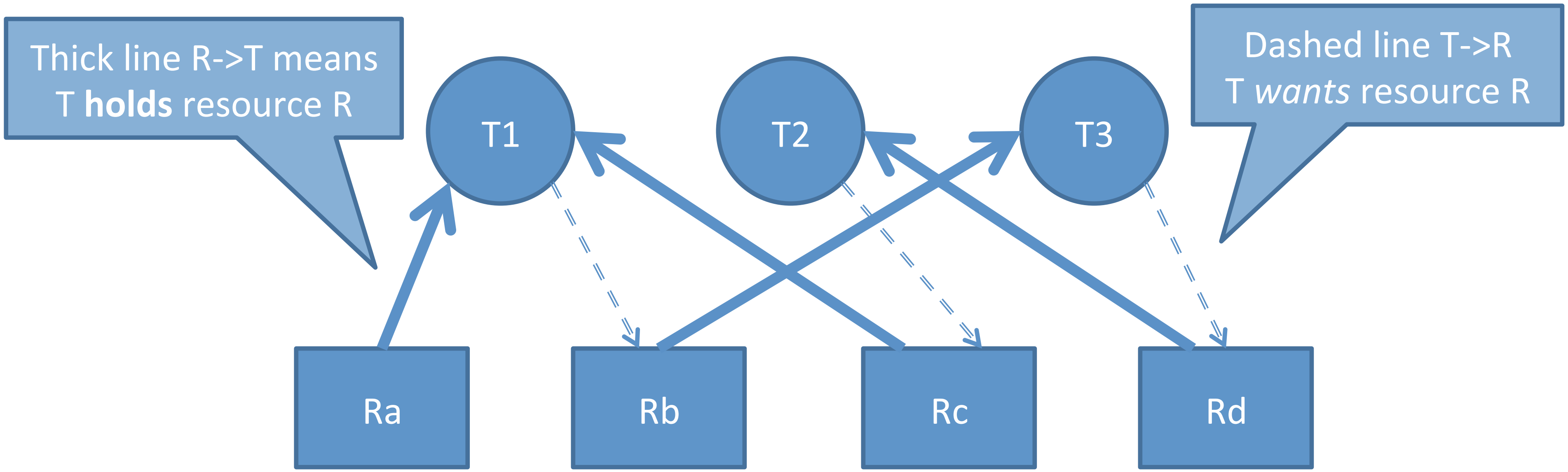
```
// thread 2
lock(Y);
 ...
  if(<cond>) {
    lock(X);
    ...
```

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

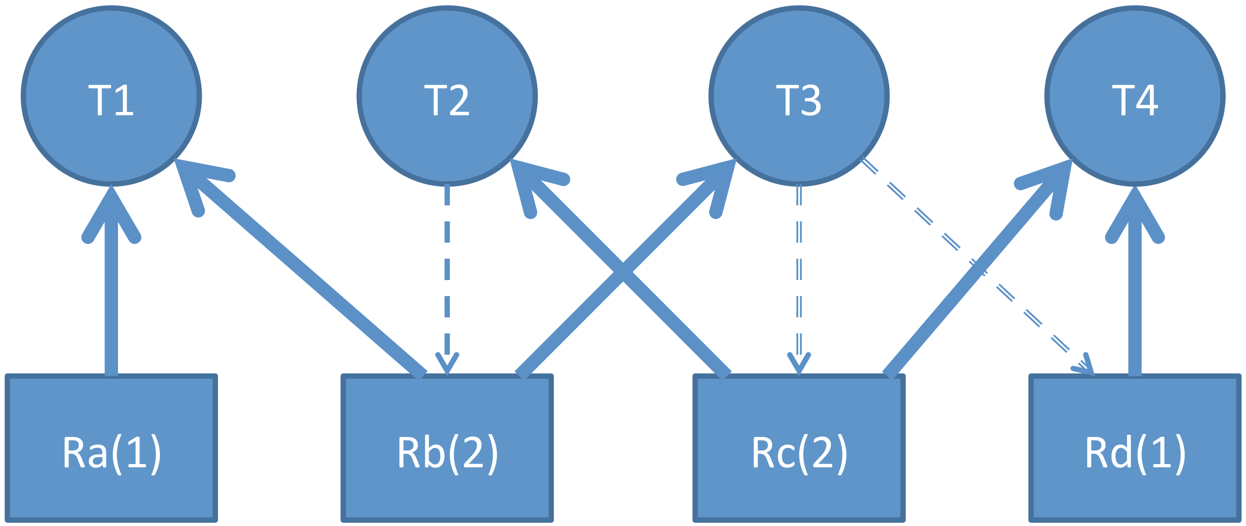
Resource Allocation Graphs

- 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



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

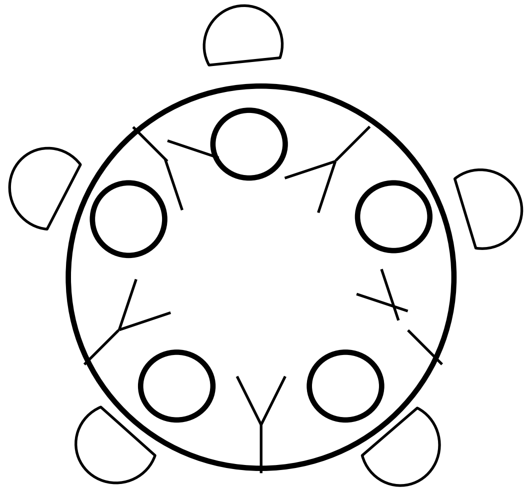
1. **Mutual Exclusion**: resources have bounded #owners
   - Could always allow access… but probably unsafe ;-(
   - However can help e.g. by using MRSW locks
2. **Hold-and-Wait**: can get **R**x and wait for **R**y
   - Require that we request all resources simultaneously; deny the request if *any*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



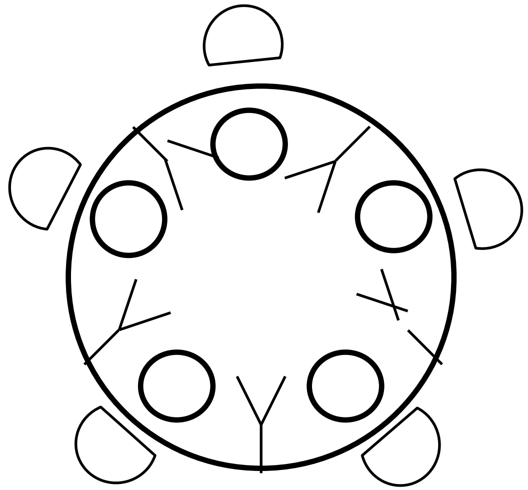
- Possible for everyone to acquire ‘left’ fork (i)
  - Q: what happens if we swap order of **signal**()s?

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

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

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		
	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	1			
T3	2	1	1	1	0	0			
T4	0	0	1	0	0	2			

W		
X	Y	Z
0	1	0

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)

Handling Priority Inversion

- Typical solution is **priority inheritance**:
  - Temporarily boost priority of lock holder to that of the highest waiting thread
  - Hard to reason about resulting behaviour
  - (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
  - ☺