Concurrent systems
Lecture 5: Liveness and Priority Guarantees

Dr David J Greaves
(Thanks to Dr Robert N. M. Watson)
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**
- **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 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 :-(

(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.:

```c
// 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 \(<\text{cond}>\) is mostly false)

• In practice there are four necessary conditions
  1. **Mutual Exclusion**: resources have bounded \#owners
  2. **Hold-and-Wait**: can acquire Rx and wait for Ry
  3. **No Preemption**: keep Rx 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.

Thick line $R \rightarrow T$ means $T$ **holds** resource $R$

Dashed line $T \rightarrow R$ means $T$ **wants** resource $R$

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.

No deadlock: If $T_1$ releases $R_b$, then $T_2$'s acquire of $R_b$ can be satisfied
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 Design Approaches

1. Ensure it never happens
   – Deadlock (static) prevention \textit{(using code structure rules)}
   – Deadlock (dynamic) avoidance \textit{(cycle finding or Banker’s Alg)}

2. Let it happen, but recover
   – Deadlock \textit{(dynamic)} detection & recovery

3. Ignore it!
   – The so-called \textbf{“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 Rx and wait for Ry
   - 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 Rx 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...

```java
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

• (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 some moment 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 $\mathcal{O}(n^2)$
- 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 \( j \) 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 fulfill 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. $R[i] \leq W$
  - (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
Banker’s Algorithm: Example 1

• Five threads and three resources (none free)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Y Z</td>
<td>X Y Z</td>
<td>X Y Z</td>
<td>X Y Z</td>
</tr>
<tr>
<td>T0</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>T1</td>
<td>2 0 0</td>
<td>2 0 2</td>
<td>7 2 5</td>
</tr>
<tr>
<td>T2</td>
<td>3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>0 0 1</td>
<td>0 0 2</td>
<td></td>
</tr>
</tbody>
</table>

• 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)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>R</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>X Y Z</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>T1</td>
<td>2 0 0</td>
<td>2 0 2</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>3 0 3</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>0 0 1</td>
<td>0 0 2</td>
<td></td>
</tr>
</tbody>
</table>

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

Threads T1, T2, T3 & T4 in deadlock set

Now wants one unit of resource Z

Cannot find a row in $R \leq W$!!
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.:

```java
// 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; e.g.,
  – 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
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**

• **Priority inheritance**

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