Deadlock (2)

Dave Eckhardt
Dave O’Hallaron
Roger Dannenberg
Travel Advisory

- Exam is upcoming (Tue 10/8, Wed 10/9, Thu 10/10)
  - If you haven’t already acted on Monday’s exam scheduling email, please do so today!
- That week is popular for mid-term exams in many classes
- If you provide a recruiter with a list of “blackout” dates, that person should schedule around that list
- Computing such a list is a good idea
Synchronization – P2

- Wednesday:
  - Thread creation working well enough to pass STARTLE test
    - [410-shell] misbehave_wrap 0 startle
- Friday:
  - Thread creation, mutexes, and cvars working well
- Tuesday:
  - Begun to debug CYCLONE and AGILITY_DRILL
- Friday:
  - P2 due at 23:59:59
  - P3 goes out afternoon/evening
Synchronization

- Project 2 advice:
  - Don't split the coding in a bad way
  - One popular bad way:
    - Person A codes list/queue, syscall stubs
    - Person B codes everything else
    - Person A will probably be in big trouble on the exam
Outline

- Review
  - Prevention/Avoidance/Detection
- Today
  - Avoidance
  - Detection/Recovery
Deadlock – Alternative Approaches

- Prevention
  - *Pass a law* against one of four ingredients
    - Note: static, absolute ban
  - Every legal application is continuously deadlock-free

- Avoidance
  - Processes *pre-declare usage patterns*
    - Note: more complicated for application, but more flexible
  - Request manager avoids “unsafe states”

- Detection/Recovery
  - Clean up only when trouble really happens
Deadlock Avoidance Assumptions

1. Processes pre-declare usage patterns
   - Could enumerate all paths through allocation space
     - Request R1, Request R2, Release R1, Request R3, ...
       - or else I will instead:
         - Request R1, Request R3, Release R3, Request R1, ...
   - Easier: declare maximal resource usage
     - I will never need more than 7 tape drives and 1 printer
Deadlock Avoidance Assumptions

2. Processes proceed to completion
   (a) Don't hold onto resources forever
       • Obvious how this helps!
   (b) Complete in “reasonable” time
       • So it is ok, if necessary, to stall P2 until P1 completes
       • We will try to avoid this
Safe Execution Sequence

- \((P_1, P_2, P_3, \ldots, P_n)\) is a **safe sequence** if
  - Every process \(P_i\) can be satisfied using
    - currently-free resources \(F\), plus
    - resources currently held by \(P_1, P_2, \ldots, P_i\)
  - Claim: \(P_i\)'s waiting is bounded by the sequence:
    - \(P_1\) will run to completion, release resources
    - \(P_2\) can complete with \(F + P_1's + P_2's\)
    - \(P_3\) can complete with \(F + P_1's + P_2's + P_3's\)
    - \(P_i\) won't wait forever, so no wait cycle, no deadlock □
Safe State

- System in a *safe state* iff...
  - there exists at least one safe sequence
- Worst-case situation
  - Every process asks for every resource at once
  - Solution: follow a safe sequence (run processes serially)
    - Slow, but not as slow as a deadlock!
- Serial execution is *worst-case*, not typical
  - Usually processes execute in parallel
Request Manager - Naïve

- Grant a resource request if
  - Enough resources are free now
- Otherwise, tell requesting process to *wait*
  - While *holding* resources
    - Which are *non-preemptible*, ...
- Easily leads to deadlock
Request Manager – Avoidance

- Grant a resource request if
  - Enough resources are free now, \textit{and}
  - Enough resources would \textit{still} be free
    - For some process to acquire the rest of its resources, complete, and release all held resources
    - And then another one
    - And then you
- Otherwise, tell requesting process to wait
  - While holding a smaller set of resources...
    - \textit{...which we previously proved it's ok to hold, because other processes don't need them to complete}
### Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

- Max = declared
- Has = allocated
- Room = Max - Has
Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Max = declared
Has = allocated
Room = Max - Has

The system has 12 items
Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

- Max = declared
- Has = allocated
- Room = Max - Has

9 items are allocated
Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Max = declared
Has = allocated
Room = Max - Has

3 items are free
Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Max = declared
Has = allocated
Room = Max - Has

“Is it safe?”

“Yes it’s safe; it’s very safe, so safe you wouldn’t believe it.”

(from “Marathon Man”)
Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Max = declared
Has = allocated
Room = Max - Has

How would we show that this state is safe?
**P1: 2 ⇒ 4**

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>
**P1: Complete**

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>5</td>
<td>-</td>
</tr>
</tbody>
</table>
**P0: 5 ⇒ 10**

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>5</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>
### P0: Complete

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>10</td>
<td>-</td>
</tr>
</tbody>
</table>

“Run P1, P0, P2” is a **safe sequence**.

So the system was in a **safe state**.
Example (from text)

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

Can P2 acquire more now?

“Is it safe?”

“No, it’s not safe; it’s very dangerous, be careful.”
**P2: 2 ⇒ 3?**

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>3</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Now, only P1 can be satisfied without waiting.
**P1: 2 ⇒ 4?**

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>
### P1: Complete

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P1</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>
Problem: P0 and P2 are each allowed to ask for >4.
If either does, it must wait, hoping the other frees some up.
If both ask for more than 4 total, both wait: **deadlock!**
**P1: Complete**

<table>
<thead>
<tr>
<th>Who</th>
<th>Max</th>
<th>Has</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>System</td>
<td>12</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

Q1: Is deadlock inevitable?

Q2: Did we miss some possible sequence other than (P1, ...)?
Avoidance - Key Ideas

- Safe state
  - Some safe sequence exists
  - Prove it by *finding one*
- Unsafe state: No safe sequence exists
- Unsafe *may not be fatal*
  - Processes might exit early
  - Processes might not use max resources today
Avoidance - Unique Resources

- Unique resources instead of multi-instance?
  - Graph algorithm
- Three edge types
  - Claim (future request)
  - Request
  - Assign
“Claim” (Future-Request) Edges

P1 → Tape 1 → P2
P2 → Tape 2 → P3
P3 → Tape 3 → P1
Claim $\Rightarrow$ Request

P1 → Tape 1 → P2 → Tape 2 → P3 → Tape 3
Request $\Rightarrow$ Assignment
Safe: No Cycle
Which Requests Are Safe?

- Pretend to satisfy request
- Look for cycles in resultant graph
A Dangerous Request
See Any Cycles?
Are “Pretend” Cycles Fatal?

- Must we worry about all cycles?
  - Nobody is waiting on a “pretend” cycle
    - Lots of the edges are only potential request edges
  - We don't have a deadlock
- “Is it safe?”
Are “Pretend” Cycles Fatal?

- No process can, without waiting
  - Acquire maximum-declared resource set
- So no process can acquire, complete, release
  - (for sure, without maybe waiting)
- Any new request could form a cycle
  - “No, it's not safe, it's very dangerous, be careful.”
- What to do?
  - Don't grant the request (block the process now, before it gets that tape drive, instead of blocking it later, while it holds it)
Avoidance - Multi-instance Resources

- Example
  - N interchangeable tape drives
  - Could represent by N tape-drive nodes
  - Needless computational expense

- Business credit-line model
  - Bank assigns each of N companies a maximum loan amount ("credit limit")
  - Bank maintains a current cash balance to make loans to companies
Avoiding “bank failure”

- Bank is “ok” when there is a safe sequence
- One company can
  - Borrow up to its credit limit
  - Do well
  - IPO
  - Pay back its full loan amount
- And then another company, etc.
No safe sequence?

- Company tries to borrow up to limit
  - Bank has no cash
  - Company C1 must wait for money C2 has
  - Maybe C2 must wait for money C1 has

- In real life
  - C1 cannot make payroll
  - C1 goes bankrupt
  - Loan never paid back in full
    - Can model as “infinite sleep”
Banker's Algorithm

int cash;        /* cash balance */
int limit[N];    /* credit limit */
int out[N];      /* borrowed */;
boolean done[N]; /* global temp! */
int future;      /* global temp! */

int progressor (int cash) {
    for (i = 0; i < N; ++i)
        if (!done[i])
            if (cash >= limit[i] - out[i])
                return (i);
    return(-1);
}
Banker's Algorithm

boolean is_safe(void) {
    future = cash;
    done[0..N] = false;

    while ((p = progressor(future)) > 0) {
        future += out[p];
        done[p] = true;
    }
    return (done[0..N] == true)
}
**Banker's Algorithm**

```c
boolean is_safe(void) {
    future = cash;
    done[0..N] = false;

    while ((p = progressor(future)) > 0) {
        future += out[p];
        done[p] = true;
    }
    return (done[0..N] == true)
}
```

What if progressor chooses processes in the wrong order?
Banker's Algorithm

- Can we loan more money to a company?
  - Pretend we did
    - update cash and out[i]
  - Is it safe?
    - Yes: lend more money
    - No: un-do to pre-pretending state, sleep

- Multi-resource version
  - Generalizes easily to N independent resource types
  - See text
Avoidance - Summary

- **Good news** - *No deadlock*
  + No *static* “laws” about resource requests
  + Allocations flexible according to system state
- **Bad news**
  - Processes must pre-declare maximum usage
  - Avoidance is *conservative*
    *Many* “unsafe” states are *almost* safe
    - System throughput reduced – extra sleeping
    - 3 processes, can allocate only 2 tape drives!?!?
Deadlock - What to do?

- **Prevention**
  - *Pass a law against one of four ingredients*

- **Avoidance**
  - *Processes pre-declare usage patterns*
  - *Request manager avoids “unsafe states”*

- **Detection/Recovery**
  - *Clean up only when trouble really happens*
Detection & Recovery - Approach

- Don't be paranoid
  - Don't refuse requests that might lead to trouble
    - (someday)
    - Most things work out ok in the end
- Even paranoids have enemies
  - Sometimes a deadlock will happen
  - Need a plan for noticing
  - Need a policy for reacting
  - Somebody must be told “try again later”
Detection - Key Ideas

- “Occasionally” scan for wait cycles
- Expensive
  - Must lock out all request/allocate/deallocate activity
  - Global mutex is the “global variable” of concurrency
  - Detecting cycles is an N-squared kind of thing
Scanning Policy

- **Throughput balance**
  - Scan too often - system becomes (very) slow
  - Scan too rarely - system becomes (extremely) slow

- **Policy candidates**
  - Scan every `<interval>`
  - Scan when CPU is “too idle”
Detection - Algorithms

- Detection: Unique Resources
  - Search for cycles in resource graph
    - (see above)

- Detection: Multi-instance Resources
  - Slight variation on Banker's Algorithm
    - (see text)

- Find a deadlock? Now what?
  - Abort
  - Preempt
Recovery - Abort

- Evict processes from the system
- All processes in the cycle?
  - Simple & blame-free policy
  - Lots of re-execution work later!
- *Just one* process in the cycle?
  - *Which* one?
    - Priority? Work remaining? Work to clean up?
    - Often immediately creates a smaller cycle – re-scan?
Recovery – Abort Just One?

P3's plan
\[ A(R3); A(R1); A(R2) \]
Recovery – Abort Just One?

P3's plan:

\[ A(R3); A(R1); A(R2) \]
Recovery – Abort Just One?

P3's plan
A(R3); A(R1); A(R2)
Recovery - Resource Preemption

- Tell some process(es): time to give, not take
  - `lock(R300) ⇒ “Ok”`
  - `lock(R346) ⇒ “EDEADLOCK”`

- What does “EDEADLOCK” mean?
  - *Can't* just retry the request (make sure you see this)
  - Must release *other* resources you hold, try later
  - Forced release may require “rollback” (yuck)

- Policy question: which process loses?
  - Lowest-numbered? ⇒ *starvation!*
Recovery – Reboot!

- You notice the system is hung (detection)
- Throw hands up and reboot (recovery)
- Hope for the best the next time around
- Go-to technique for IT help lines everywhere
Summary - Deadlock

- Deadlock is...
  - Set of processes
  - Each one waiting for something held by another
- Four “ingredients”
- Three approaches