Deadlock Solutions: Avoidance, Detection, and Recovery

CS 241

March 30, 2012

University of Illinois
Deadlock: definition

There exists a cycle of processes such that each process cannot proceed until the next process takes some specific action.

Result: all processes in the cycle are stuck!
Deadlock solutions

Prevention
  • Design system so that deadlock is impossible

Avoidance
  • Steer around deadlock with smart scheduling

Detection & recovery
  • Check for deadlock periodically
  • Recover by killing a deadlocked processes and releasing its resources

Do nothing
  • Prevention, avoidance and detection/recovery are expensive
  • If deadlock is rare, is it worth the overhead?
  • Manual intervention (kill processes, reboot) if needed
Last time: Deadlock Prevention

#1: No mutual exclusion
   • Thank you, Captain Obvious

#2: Allow preemption
   • OS can revoke resources from current owner

#3: No hold and wait
   • When waiting for a resource, must not currently hold any resource

#4: Request resources in order
   • When waiting for resource $i$, must not currently hold any resource $j > i$
   • As you can see: If your program satisfies #3 then it satisfies #4
“Request In Order” is more permissive

All programs

Will not deadlock

Request in order

No hold and wait

Might deadlock (depending on scheduler, inputs, etc.)

Definitely deadlock

All programs
Are we always in trouble without ordering resources?

No, not always:

Ordered resource requests are **sufficient** to avoid deadlock, but **not necessary**

Convenient, but may be conservative
Q: What’s the rule of the road?

What’s the law? Does it resemble one of the rules we saw?
Deadlock Avoidance
Deadlock Avoidance

Idea: Steer around deadlock with smart scheduling

Assume OS knows:

- Number of available instances of each resource
  - Each individual mutex lock is a resource with one instance available
  - Each individual semaphore is a resource with possibly multiple “instances” available
- For each process, current amount of each resource it owns
- For each process, maximum amount of each resource it might ever need
  - For a mutex this means: Will the process ever lock the mutex?

Assume processes are independent

- If one blocks, others can finish if they have enough resources
How to guide the system down a safe path of execution

Helper function: is a given state safe?

- **Safe** = there’s definitely a way to finish the processes without deadlock

When a resource allocation request arrives

- Pretend that we approve the request
- Call function: Would we then be safe?
  - If safe,
    - Approve request
  - Otherwise,
    - Block process until its request can be safely approved
    - Some other process is scheduled in the meantime

This is called the Banker’s Algorithm

- Dijkstra, 1965
What is a state?

For each resource,
- Current amount *available*
- Current amount *allocated* to each process
- Future amount *needed* by each process (maximum)

<table>
<thead>
<tr>
<th>Buffer space</th>
<th>A mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td></td>
</tr>
<tr>
<td>P1 alloc</td>
<td></td>
</tr>
<tr>
<td>P2 alloc</td>
<td></td>
</tr>
<tr>
<td>P1 need</td>
<td></td>
</tr>
<tr>
<td>P2 need</td>
<td></td>
</tr>
</tbody>
</table>
When is a state safe?

There is an execution order that can finish

In general, that’s hard to predict
  • So, we’re conservative: find sufficient conditions for safety
  • i.e., make some pessimistic assumptions

Pessimistic assumptions:
  • A process might request its maximum resources at any time
  • A process will never release its resources until it’s done
Computing safety

“There is an execution order that can finish”

Search for an order P1, P2, P3, ... such that:

- P1 can finish using what it has plus what’s free
- P2 can finish using what it has + what’s free + what P1 releases when it finishes
- P3 can finish using what it has + what’s free + what P1 and P2 will release when they finish
- ...

...
“There is an execution order that can finish”

More specifically... Search for an order P1, P2, P3, ... such that:

- P1’s max resource needs \( \leq \) what it has + what’s free
- P2’s max resource needs \( \leq \) what it has + what’s free + what P1 will release when it finishes
- P3’s max resource needs \( \leq \) what it has + what’s free + what P1 and P2 will release when they finish
- ...

But how do we find that order?
Inspiration
Playing Pickup Sticks with Processes

Pick up a stick on top
  • = Find a process that can finish with what it has plus what’s free

Remove stick
  • = Process finishes & releases its resources

Repeat until...
  • ...all processes have finished
    ▪ Answer: safe
  • ...or we get stuck
    ▪ Answer: unsafe
Try it: is this state safe?

- Buffer space
  - Free
  - P1 alloc
  - P2 alloc
  - P1 need
  - P2 need
- A mutex

Which process can go first?
Try it: is this state safe?

Buffer space

<table>
<thead>
<tr>
<th>Free</th>
<th>P1 alloc</th>
<th>P2 alloc</th>
<th>P1 need</th>
<th>P2 need</th>
</tr>
</thead>
</table>

A mutex

Start with P2
Try it: is this state safe?

Buffer space

Free
P1 alloc
P2 alloc
P1 need
P2 need

A mutex

Release P2’s resources
Try it: is this state safe?

- **Free**: Buffer space
- **P1 alloc**: Buffer space
- **P2 alloc**: Buffer space
- **P1 need**: Buffer space
- **P2 need**: Buffer space

- **A mutex**: Release P2’s resources
Try it: is this state safe?

Buffer space

Free
P1 alloc
P2 alloc
P1 need
P2 need

A mutex

Continue with P1
Try it: is this state safe?

Buffer space

Free
P1 alloc
P2 alloc
P1 need
P2 need

A mutex

Continue with P1
Try it: is this state safe?

Yes, it’s safe: Order is P2, P1
Example 2: Is this state safe?

Can P1 go first?
Can P2 go first?
Can P3 go first?

Buffer space

Free
P1 alloc
P2 alloc
P3 alloc
P1 need
P2 need
P3 need
Example 2: Is this state safe?

Buffer space

Free
P1 alloc
P2 alloc
P3 alloc
P1 need
P2 need
P3 need
Example 2: Is this state safe?

Buffer space

Free
P1 alloc
P2 alloc
P3 alloc
P1 need
P2 need
P3 need
Example 2: Is this state safe?

Can P1 go next?
Can P2 go next?

Buffer space
Free
P1 alloc
P2 alloc
P3 alloc
P1 need
P2 need
P3 need

Unsafe!
Deadlock Detection & Recovery
Deadlock Detection

Check to see if a deadlock has occurred!

Special case: Single resource per type
- E.g., mutex locks (value is zero or one)
- Check for cycles in the resource allocation graph

General case
- E.g., semaphores, memory pages, ...
- See book, p. 355 – 358
Dependencies between processes

Resource allocation graph

Corresponding process dependency graph
Deadlock Recovery

Recovery idea: get rid of the cycles in the process dependency graph

Options:
- Kill all deadlocked processes
- Kill one deadlocked process at a time and release its resources
- Steal one resource at a time
- Roll back all or one of the processes to a checkpoint that occurred before they requested any resources, then continue
  - Difficult to prevent indefinite postponement
Deadlock Recovery

Resource allocation graph

Corresponding process dependency graph

Have to kill one more
Deadlock Recovery

Resource allocation graph

Corresponding process dependency graph

Only have to kill one
Deadlock Recovery

How should we pick a process to kill?

We might consider...

• process priority
• current computation time and time to completion
• amount of resources used by the process
• amount of resources needed by the process to complete
• the minimal set of processes we need to eliminate to break deadlock
• is process interactive or batch?
Rollback instead of killing processes

Selecting a victim
  • Minimize cost of rollback (e.g., size of process’s memory)

Rollback
  • Return to some safe state
  • Restart process for that state
  • Note: Large, long computations are sometimes checkpointed for other reasons (reliability) anyway

Challenge: Starvation
  • Same process may always be picked as victim
  • Fix: Include number of rollbacks in cost factor
Deadlock Summary

Deadlock: cycle of processes/threads each waiting for the next
- Nasty timing-dependent bugs!

Detection & Recovery
- Typically very expensive to kill / checkpoint processes

Avoidance: steer around deadlock
- Requires knowledge of everything an application will request
- Expensive to perform on each scheduling event

Prevention (ordered resources)
- Imposes conservative rules on application that preclude deadlock
- Application can do it; no special OS support
Deadlock Summary

Typical solution:

- OS (Unix/Windows) do nothing (Ostrich Algorithm)
- Application uses general-purpose deadlock prevention

Transaction systems (e.g., credit card processing) may use detection/recovery/avoidance