Deadlock Solutions

CS 241

March 28, 2012

University of Illinois
Announcements

Office hours today: 3-4 and 5-6

In between: Talk by Tom Wenisch
  • Energy efficiency in warehouse-scale computers
  • 4pm, in 3405 SC

Midterm exams: you may look at them through the end of this week
  • An extension of our one-week policy
  • Drop by office hours today (3-4 and 5-6) or schedule appointment
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 in the real world

Which way should I go?
Deadlock in the real world

Drat!

GRIDLOCK!
Deadlock: One-lane Bridge

Traffic only in one direction

Each section of a bridge can be viewed as a resource

What can happen?
Deadlock: One-lane Bridge

Traffic only in one direction

Each section of a bridge can be viewed as a resource

Deadlock
- Resolved if cars back up (preempt resources and rollback)
- Several cars may have to be backed up
Deadlock: One-lane Bridge

Traffic only in one direction

Each section of a bridge can be viewed as a resource

Deadlock
  ○ Resolved if cars back up (preempt resources and rollback)
  ○ Several cars may have to be backed up

But, starvation is possible
  ■ e.g., if the rule is that Westbound cars always go first when present
Deadlock: One-lane Bridge

Deadlock vs. Starvation

• Starvation = Indefinitely postponed
  ▪ Delayed repeatedly over a long period of time while the attention of the system is given to other processes
  ▪ Logically, the process may proceed but the system never gives it the CPU (unfortunate scheduling)

• Deadlock = no hope
  ▪ All processes blocked; scheduling change won’t help
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
Deadlock Prevention
Aside: Necessary Conditions for Deadlock

Mutual exclusion
- Processes claim exclusive control of the resources they require

Hold-and-wait (a.k.a. wait-for) condition
- Processes hold resources already allocated to them while waiting for additional resources

No preemption condition
- Resources cannot be removed from the processes holding them until used to completion

Circular wait condition
- A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain
Deadlock prevention

Goal 1: devise resource allocation rules which make circular wait impossible
  • Resources include mutex locks, semaphores, pages of memory, ...
  • ...but you can think about just mutex locks for now

Goal 2: make sure useful behavior is still possible!
  • The rules will necessarily be conservative
    ▪ Rule out some behavior that would not cause deadlock
  • But they shouldn’t be to be too conservative
    ▪ We still need to get useful work done
Rule #1: No Mutual Exclusion

For deadlock to happen: processes must claim exclusive control of the resources they require

How to break it?
Rule #1: No Mutual Exclusion

For deadlock to happen: processes must claim exclusive control of the resources they require.

How to break it?
- Non-exclusive access only
  - Read-only access
- Battle won!
  - War lost
  - Very bad at Goal #2
Rule #2: Allow preemption

A lock can be taken away from current owner
  • Let it go: If a process holding some resources is denied a further request, that process must release its original resources
  • Or take it all away: OS preempts current resource owner, gives resource to new process/thread requesting it

Breaks circular wait
  • ...because we don’t have to wait

Reasonable strategy sometimes
  • e.g. if resource is memory: “preempt” = page to disk

Not so convenient for synchronization resources
  • e.g., locks in multithreaded application
  • What if current owner is in the middle of a critical section updating pointers? Data structures might be left in inconsistent state!
Rule #3: No hold and wait

When waiting for a resource, must not hold others
- So, process can only have one resource locked
- Or, it must request all resources at the beginning
- Or, before asking for more: give up everything you have and request it all at one time

Breaks circular wait
- In resource allocation diagram: process with an outgoing link must have no incoming links
- Therefore, cannot have a loop!
Rule #3: No hold and wait

Breaks circular wait

- In resource allocation diagram: process with an outgoing link must have no incoming links
- Therefore, cannot have a loop!

Q: Which of these request links would be disallowed?
Rule #3: No hold and wait

Breaks circular wait

- In resource allocation diagram: process with an outgoing link must have no incoming links
- Therefore, cannot have a loop!

A: Legal links are...

- Process
- Resource
- request
- ownership
Rule #3: No hold and wait

Very constraining (bad job on Goal #2)

- Better than Rules #1 and #2, but...
- Often need more than one resource
- Hard to predict at the beginning what resources you’ll need
- Releasing and re-requesting is inefficient, complicates programming, might lead to starvation
Rule #4: request resources in order

Must request resources in increasing order
  • Impose ordering on resources (any ordering will do)
  • If holding resource $i$, can only request resources $> i$

Much less constraining (decent job on Goal #2)
  • Strictly easier to satisfy than “No hold and wait”: If we can request all resources at once, then we can request them in increasing order
  • But now, we don’t need to request them all at once
  • Can pick the arbitrary ordering for convenience to the application
  • Still might be inconvenient at times

But why is it guaranteed to preclude circular wait?
Dining Philosophers solution with unnumbered resources

Back to the trivial broken “solution”...

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```
Dining Philosophers solution with unnumbered resources

Back to the trivial broken “solution”...

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```
Dining Philosophers solution with unnumbered resources

Back to the trivial broken “solution”...

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork(((i+1)%N));
        eat(); /* yummy */
        put_fork(i);
        put_fork(((i+1)%N));
    }
}
```
Instead, number resources

First request lower numbered fork

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources...

One philosopher can eat!

```c
#include <unistd.h>

#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Ordered resource requests prevent deadlock

Without numbering

Cycle!
Ordered resource requests prevent deadlock

With numbering

Contradiction: Must have requested 3 first!
Proof by M.C. Escher
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?
Summary

Deadlock prevention

- Imposes rules on what system can do
- These rules are conservative
- Most useful technique: ordered resources
- Application can do it; no special OS support

Next: dealing with deadlocks other ways

- Avoidance
- Detection & recovery