CS 4414/5416: Systems Programming

Synchronization Issues

David Bindel

2026-11-03

Logistics

Upcoming

- P4 (netsim) revision
  - Shortened assignment, final part optional
  - Autograder posted (open to Thu; recommend Wed)
- P5 (Tetris) will be released by 11/4
- Lab 11 is Friday: multithreaded web service
- Next week: Start asynchronous programming

This Thursday

We have

(a) A CS colloquium speaker who is an old friend
(b) A distinguished lecture in Baker Lab 200 at 5

So expect us to end around 4

Recap

Where We Are

This module: Concurrent programming with threads

- Message-based concurrent programming
- Shared memory concurrent programming
- Synchronization issues
- Concurrent network programming

This Week

- Today: Synchronization issues
  - See: “Three Easy Parts” chapter
  - See also 4410 notes for much more detail
- Thu: Concurrent network programming

Why Synchronize?

Desiderata:

- Preserve global invariants (local violations OK)
- Manage ordering handshakes of various sorts

Basic Mechanisms

For shared memory (vs message-passing style):

- `Arc`, `Mutex`, `RwLock`, and `Condvar`
- Atomics (enabling lock-free shared data structures)

Basis for higher-level abstractions (work queues, parallel iteration, etc).

Poll Everywhere!

Which of these is statements about atomics is false?

(a) Under `SeqCst` order, ops act as if interleaved across threads.

(b) The happens-before relation within a thread is consistent with execution order constraints.

(c) In `Relaxed` order, the only inter-thread happens-before relations are matched loads and stores.

(d) All threads see the same total modification order when looking at a single variable.

Non-Deadlock Bugs

Where do Bugs Happen?

Most synchronization errors are *not* deadlock!

- Source: Lu et al, “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics,” ASPLOS 08 (found via OSTEP book)
- So what are the big issues?

Atomicity Violations

"Serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)."

- Atomicity here means "indivisible" (not "uses Rust atomics")
- Ex: Check validity, then use; another thread modifies between

NB: Harder to get this with actor model.

Rust to the Rescue!

Many issues ruled out in safe Rust!

- Borrow checker: no concurrent write + other access
- Have to work harder to screw things up:
  - Get lock for a sanity check, release, re-lock, and assume check?
  - Still *possible*, but Rust makes it harder
- Are atomics worse for thinking about atomicity?

Ordering Violations

"Desired order between two (groups of) memory accesses is flipped"

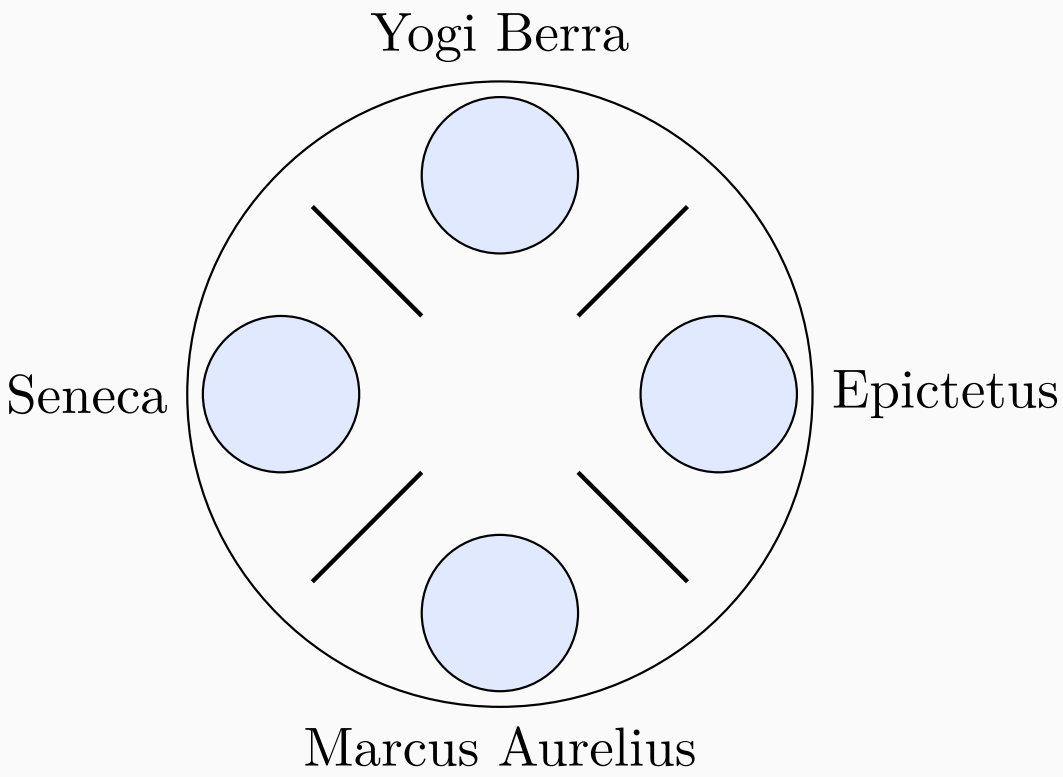
- Example: Read from a thread before write from another
- Easy to mess up with atomics (particularly in `Relaxed`)
- More likely to get right with locks + condition variables

Rust doesn't really help you with this.

Ask Me a Question!

A Philosophical Problem

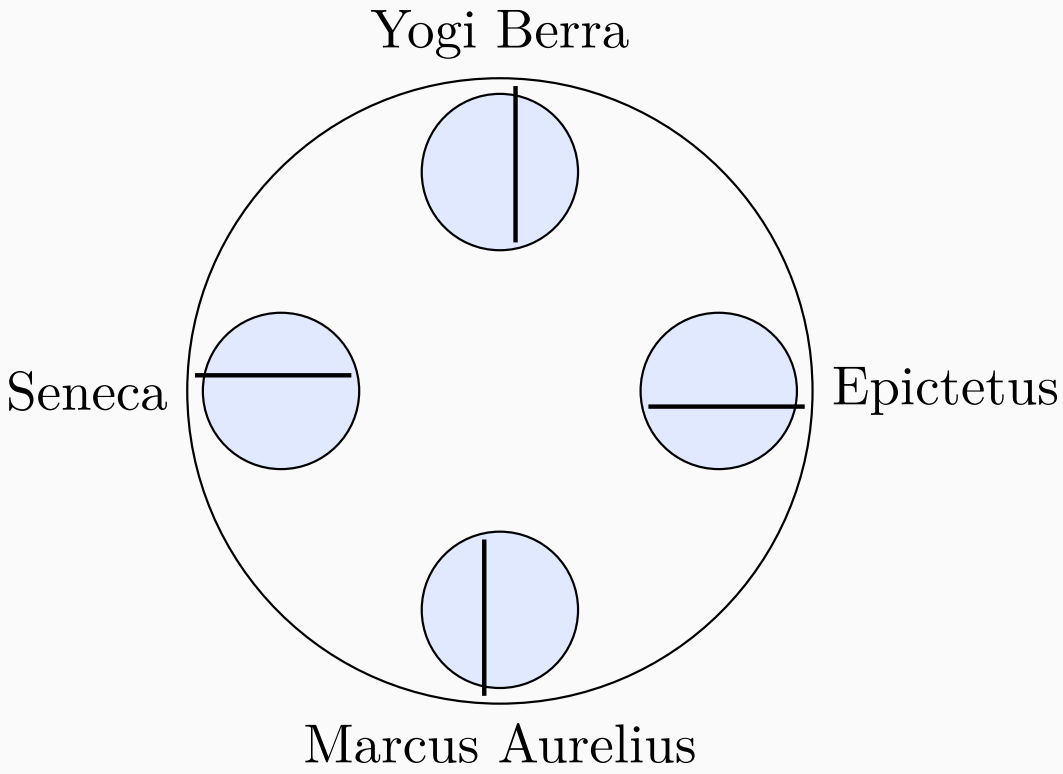
Dining Philosophers



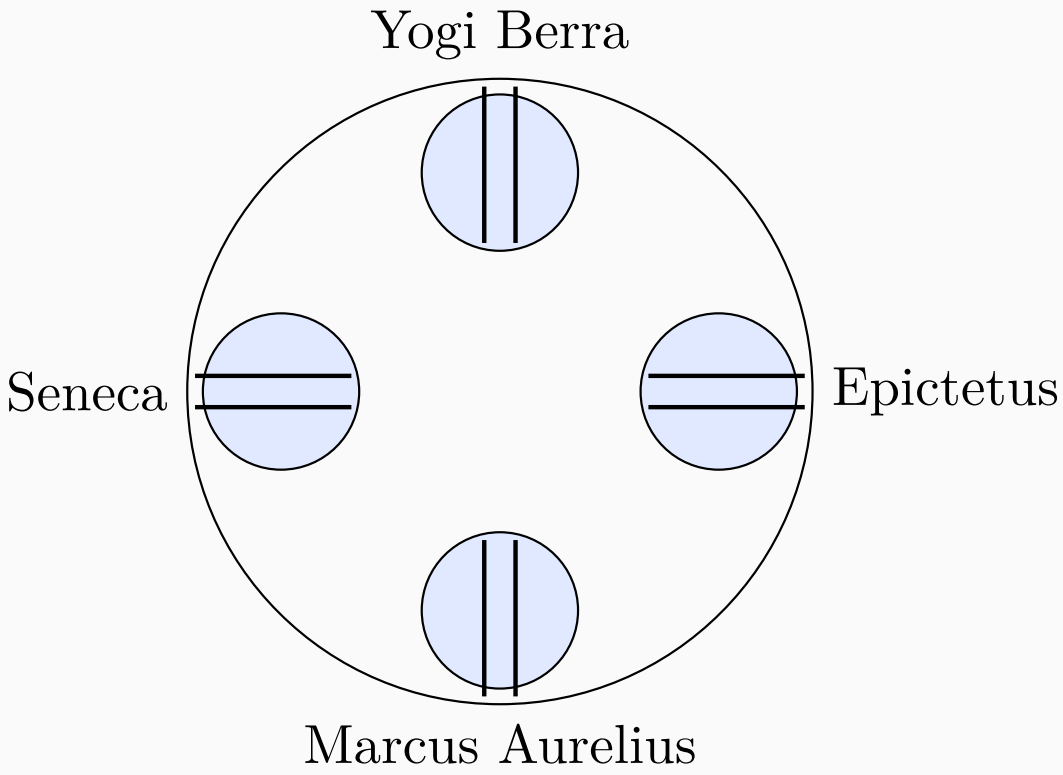
Dining Philosophers V1

```
fn dine(philosopher: Philosopher) {
    let chopsticks = philosopher.chopsticks;
    while philosopher.is_hungry() {
        let l = chopsticks.left.lock().unwrap();
        let r = chopsticks.right.lock().unwrap();
        philosopher.eat(l, r);
    }
}
```

Dining Philosophers Problem!

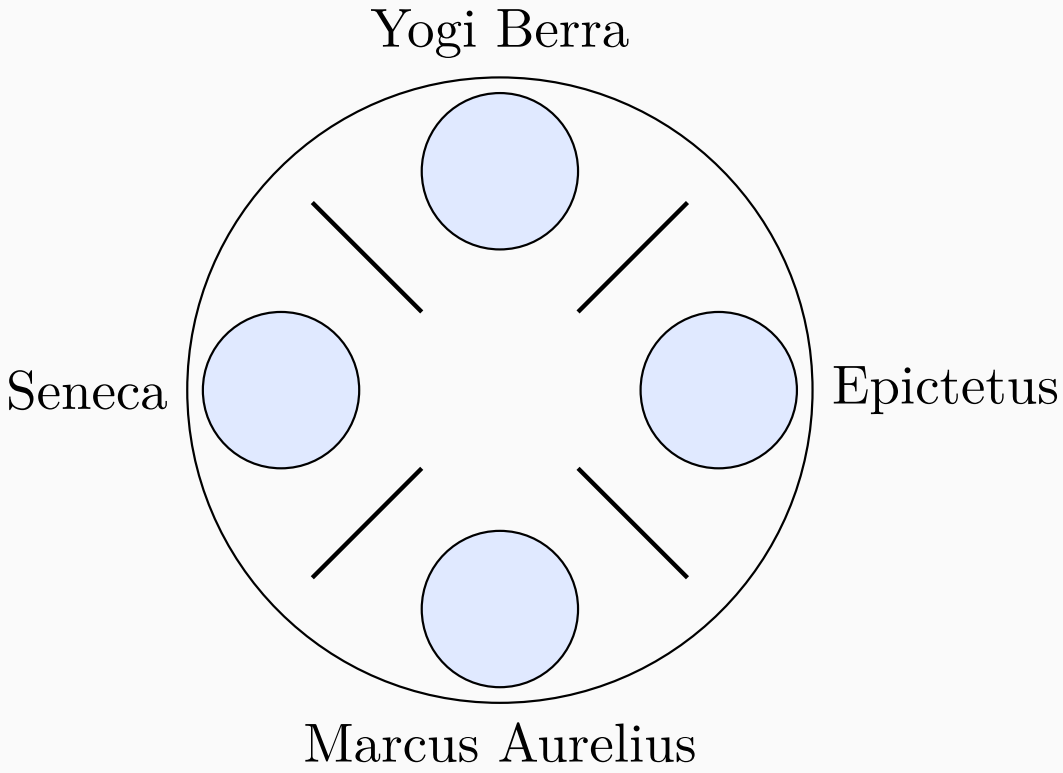


Dining Philosophers Fix 1



What if the philosophers have enough utensils?

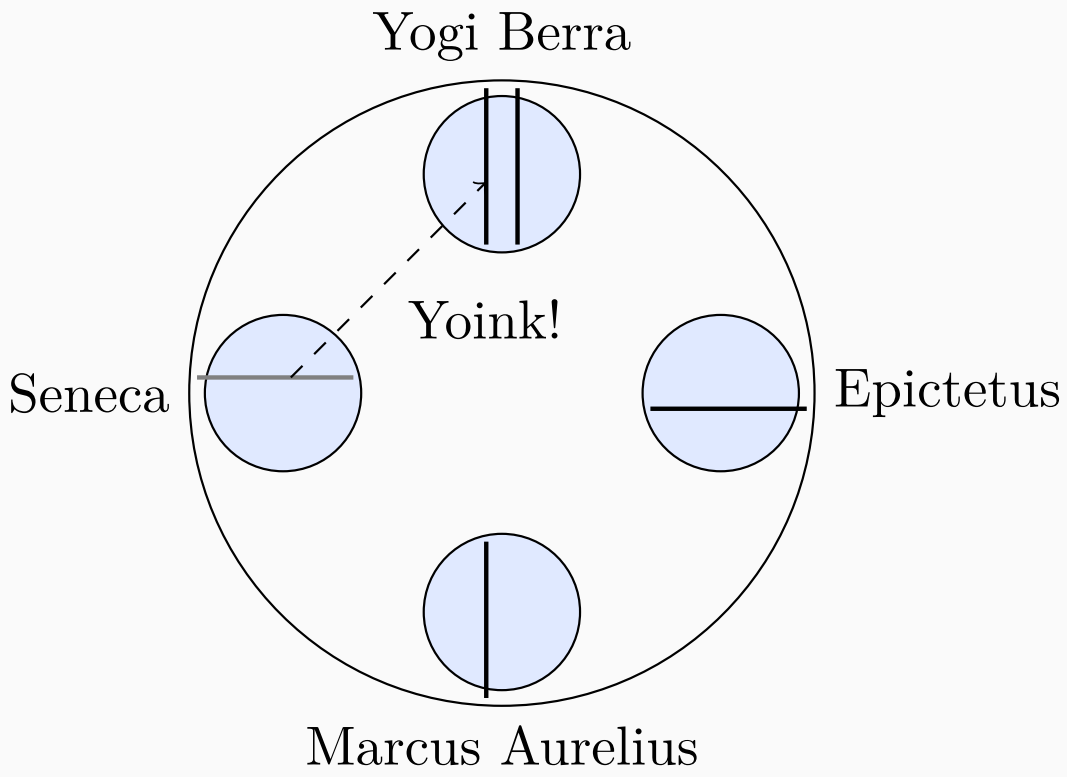
Dining Philosophers Fix 2



What if they put back a chopstick if they can't get both?

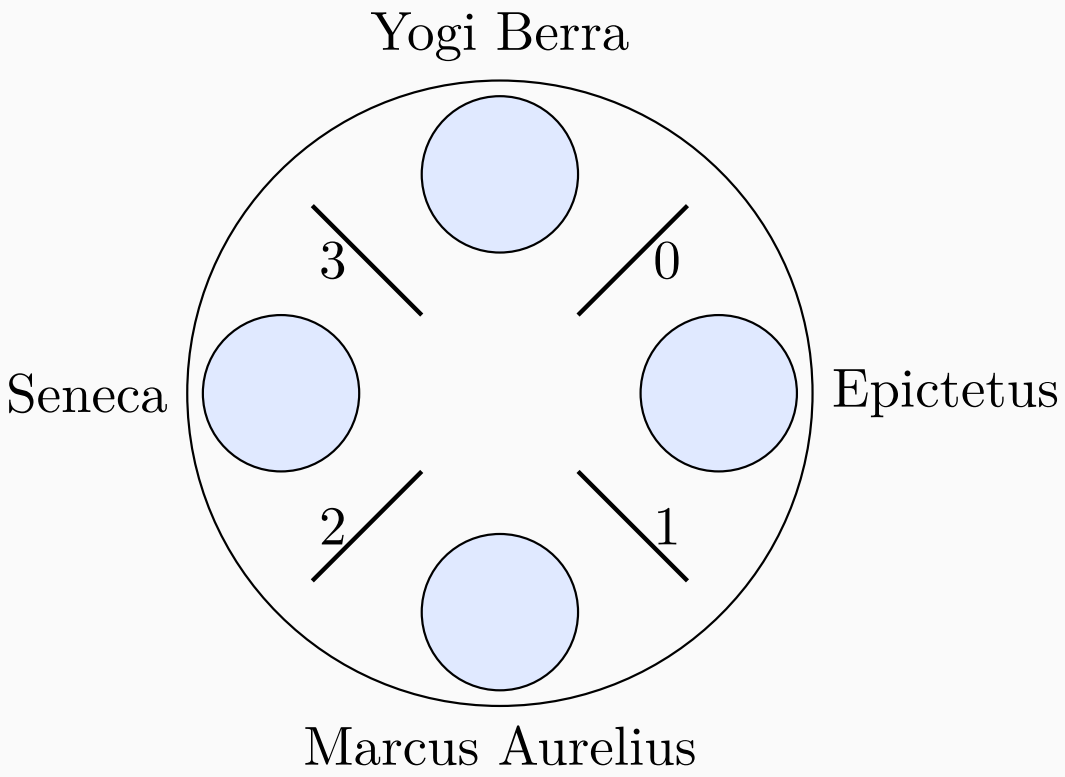
(Can implement with `try_lock` on `Mutex`)

Dining Philosophers Fix 3



What if they can steal chopsticks?

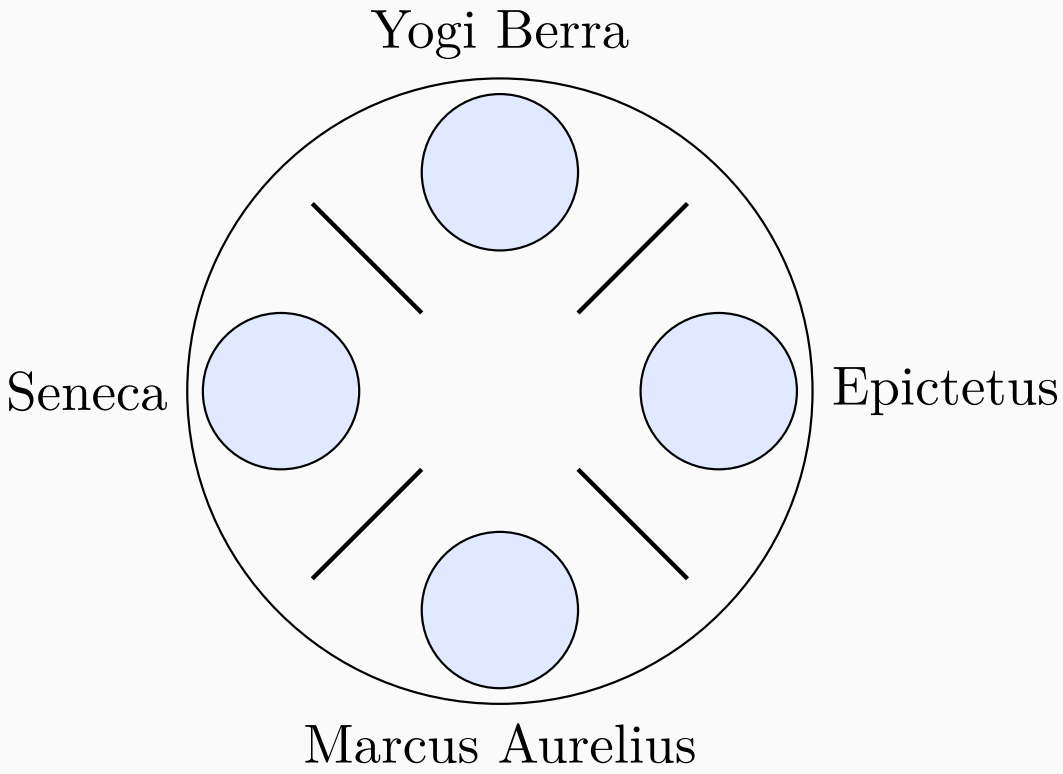
Dining Philosophers Fix 4



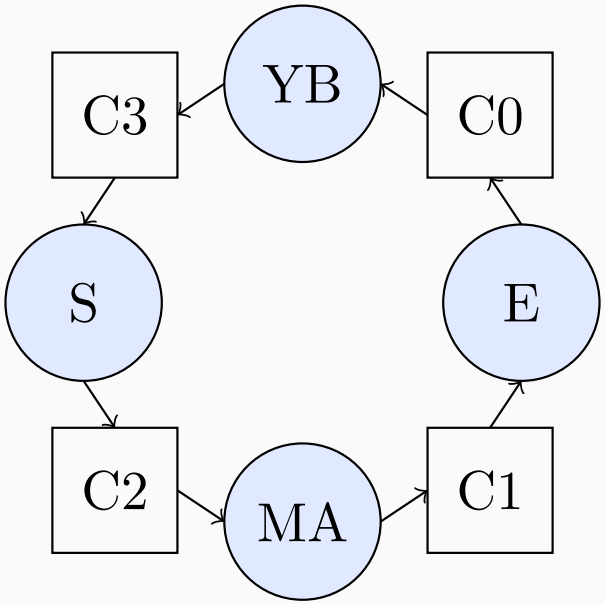
What if they acquire chopsticks via a global order?

Deadlock Conditions

Philosophers Revisited



Resource Allocation Graph



Resource Allocation Graph

Bipartite graph with:

- Thread-to-resource “waiting for” edges
- Resource-to-thread “held by” edges

Cycles mean trouble!

Necessary and Sufficient Conditions

Mutual exclusion Multiple threads cannot simultaneously hold the same resource

Hold and wait Threads can hold a resource and wait for another

No pre-emption A thread cannot be forced to relinquish a resource until it completes its task

Circular wait There is a cycle in the resource allocation graph

(Coffmann, 1971)

Prevention Strategy

Four necessary and sufficient conditions:

- Mutual exclusion
- Hold and wait
- No pre-emption
- Circular wait

Prevention: Make one of the four conditions impossible!

Break (or Reduce) Mutual Exclusion

- Replicate resources! More chopsticks...
- Lock-free data structures
- Reduce granularity of protection
    - `RwLock` vs `Mutex`

Breaking Hold-and-Wait

- `try_lock` for nonblocking attempt to lock
- Use condition variables to cede lock
- Acquire and release simultaneously

Allowing Preemption

Optimistic concurrency doesn't deadlock:

- Set a checkpoint
- Compute assuming no contention
- Validate assumption
- Commit if no contention, rollback/abort otherwise

Breaking Circular Wait

- Order resources (or locks around them)
- Must acquire resources in ascending order
- Then no cycles possible in resource allocation graph!

Ask Me a Question!

Detection and Recovery

What if We Deadlock?

- Kill the program?
- Or maybe kill a thread?
    - Maybe code will be robust to this?
- Or roll back a thread?
    - Requires some checkpointing

Simplest Protocol

- Add timeout to resource acquisition
- If timeout exceeded, kill a thread

Removes hold-and-wait.

Checking Resource Graphs

- Periodically compute resource graph
- Check for cycles (DFS or BFS)
- If cycle appears, kill a thread involved

Maybe more efficient than timeout?

Avoidance

Safe State

- Consider *state* of allocations (resources to threads)
- State is *safe* if deadlock is not guaranteed
- Scheduler blocks on resource acquisition that would create unsafe state

Banker's Algorithm (Dijkstra)

Check for safety involves simulating future:

- while processes exist (in simulation)
  - Choose p that can complete with available resources
    - If none, state is unsafe
  - Add resources held by p to available pool
  - Remove process p from list

If loop completes, state is safe

Ask Me a Question!

Livelock and Starvation

Livelock in Life

- The “no, after you” problem
- Action is happening, this is not deadlock!
- Deadlock prevention/avoidance can still cause livelock

Ordering

Fundamentally, need to break ties!

- Priority ordering: simple, but may cause *starvation*
- Often use randomized protocols

Wrap-Up

Poll Everywhere!

Which is true?

(a) We can always undo deadlock by killing a random thread.

(b) The four conditions for deadlock are sufficient, not necessary.

(c) The Banker's algorithm of Dijkstra is about deadlock avoidance.

(d) Deadlock-based errors are the most common synchronization bugs.

Summary

Various types of synchronization issues:

- Non-deadlock failures of atomicity, ordering
- Deadlock
- Livelock

Summary

- Four conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - No pre-emption
  - Circular wait
- Can prevent, detect, or avoid deadlock

Prevention, detection, or avoidance

- *Prevention*: Static conditions disallow deadlock
- *Avoidance*: Dynamic checks disallow deadlock
- *Detection*: Break deadlock state if it happens

Up Next

- Thu: Multi-threaded network programming
- Fri: Practicing multi-threaded network programming
- Next week: Start asynchronous programming