Multi-Object Synchronization: Deadlock

Main Points

- Problems with synchronizing multiple objects
- Definition of deadlock
  - *Circular waiting for resources*
- Conditions for its occurrence
- Solutions for avoiding and breaking deadlock

Large Programs

- What happens when we try to synchronize across multiple objects in a large program?
  - Each object with its own lock, condition variables
  - Is concurrency modular?
- Deadlock
- Performance
- Semantics/correctness

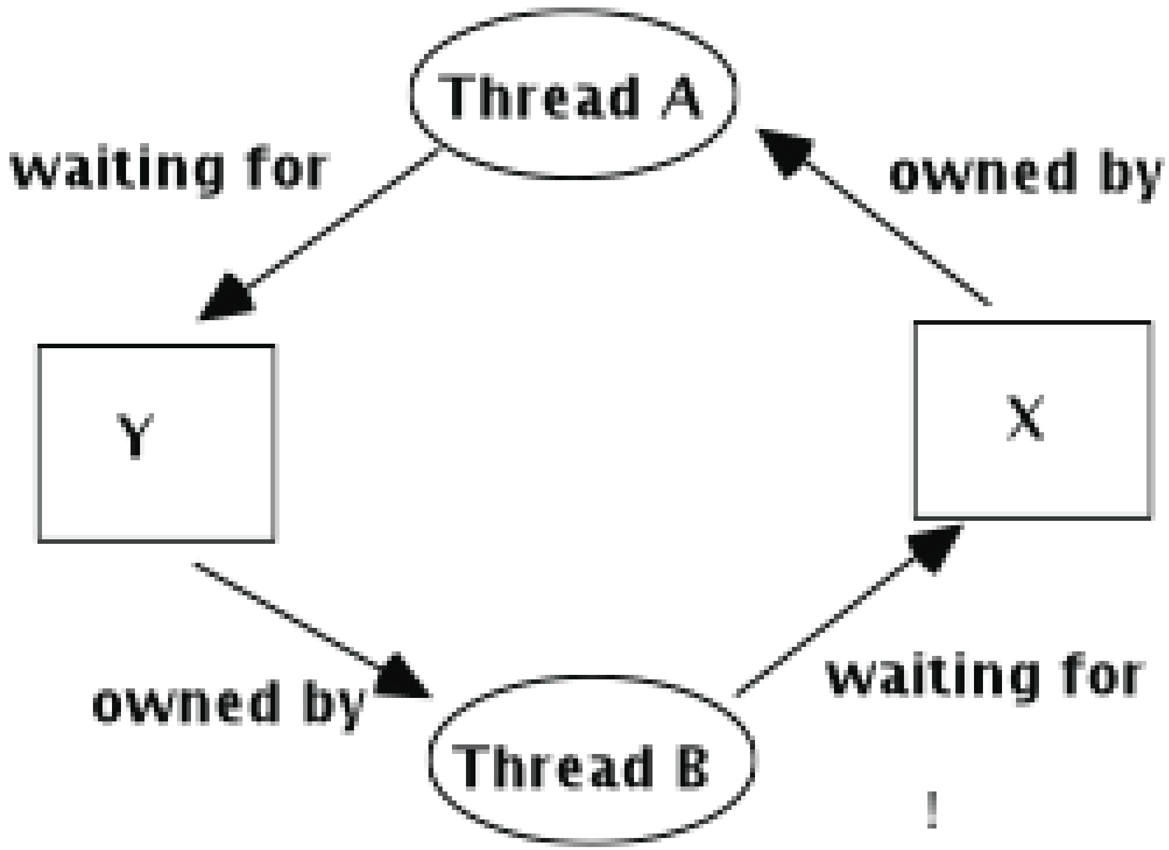
Deadlock: Preliminary Definitions

- *Resource*: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
- *Preemptable resource*: can be taken away by OS
  - *Non-preemptable*: must leave with thread
- *Starvation*: thread waits indefinitely

Deadlock: Definitions

- *Deadlock*: One or more threads are not making progress, and never will, due to circular waiting for resources
  - Thread 0 holds lock A and is trying to acquire lock B which is held by thread 1 which is trying to acquire lock 0
- Deadlock => starvation
  - but not vice versa
- *(Livelock:* threads change state, but don't make progress
  - *cell call drops, and each of you starts calling the other back as fast as you can)*

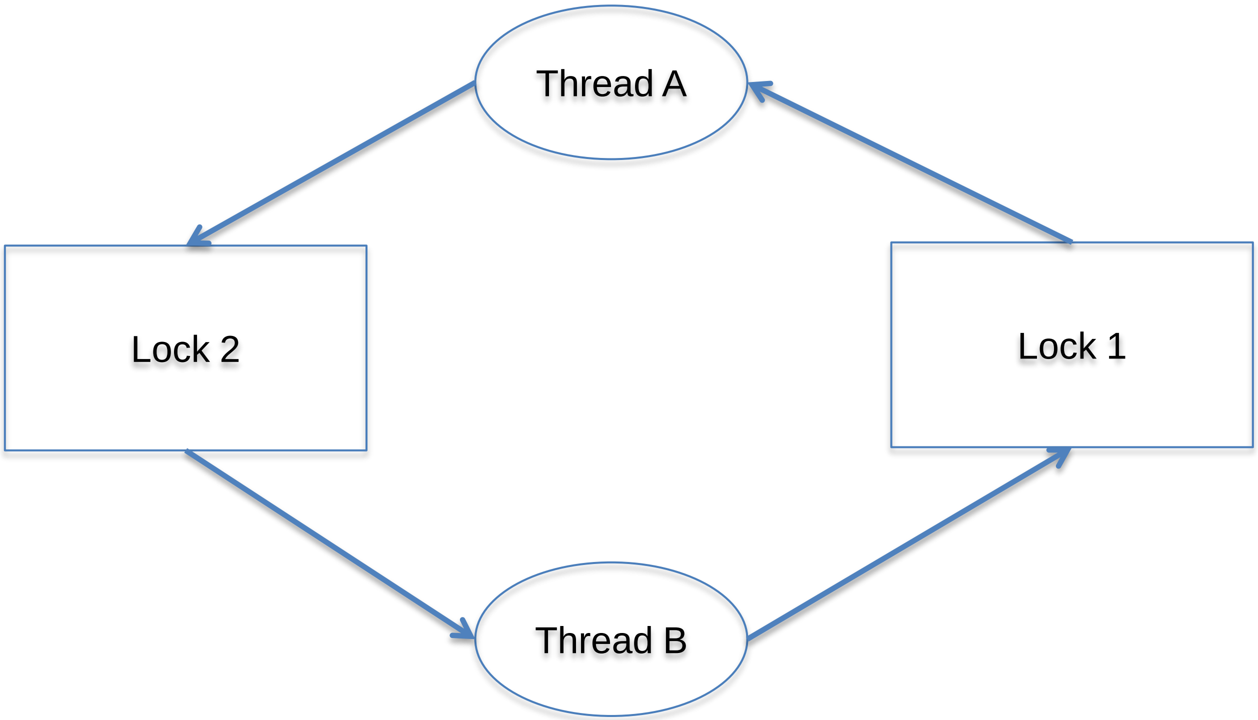
Circular Waiting



Deadlock Example 1: Two Locks

Thread A	Thread B
lock1.acquire();	lock2.acquire();
lock2.acquire();	lock1.acquire();
// update objs 1 and 2	// update objs 1 and 2
lock2.release();	lock1.release();
lock1.release();	lock2.release();

Example 1 Waiting-for Graph



Deadlock Example 2: Two Bounded Buffers

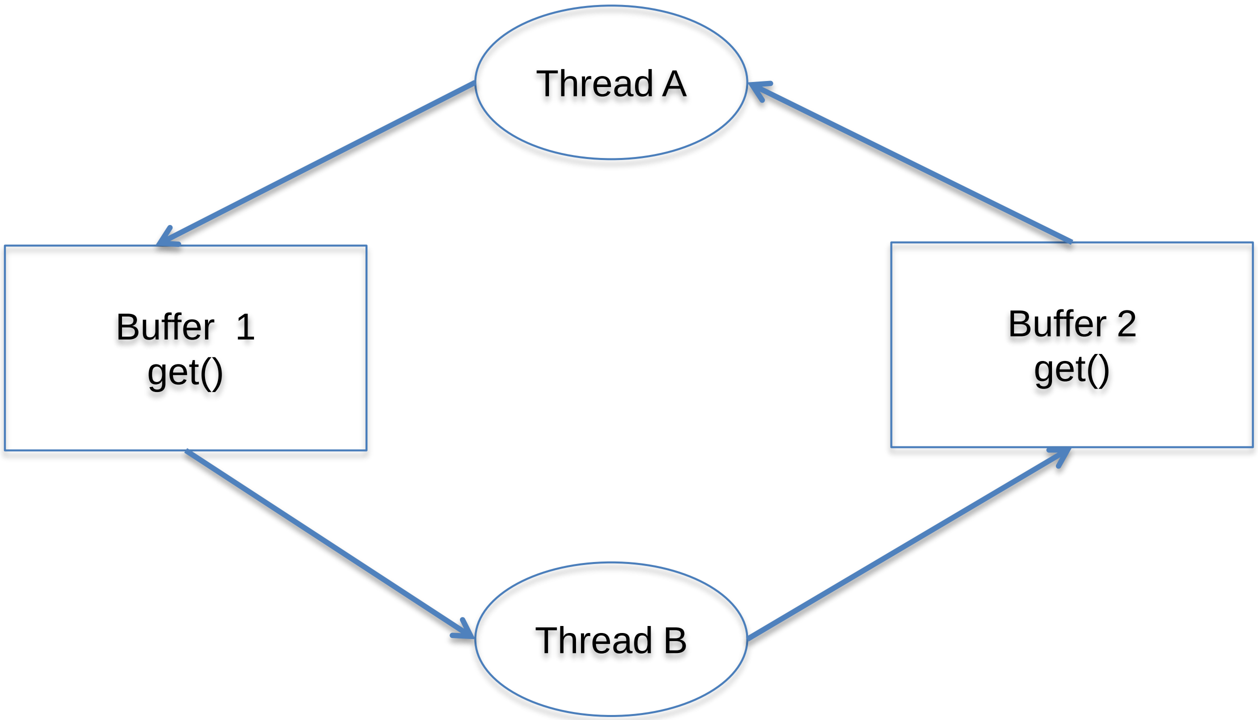
Thread A

```
buffer1.put(data);
buffer1.put(data);
…
buffer2.get();
buffer2.get();
```

Thread B

```
buffer2.put(data);
buffer2.put(data);
…
Buffer1.get();
Buffer1.get();
```

Example 2 Waiting-for Graph



Deadlock Example 3: Two locks and a condition variable

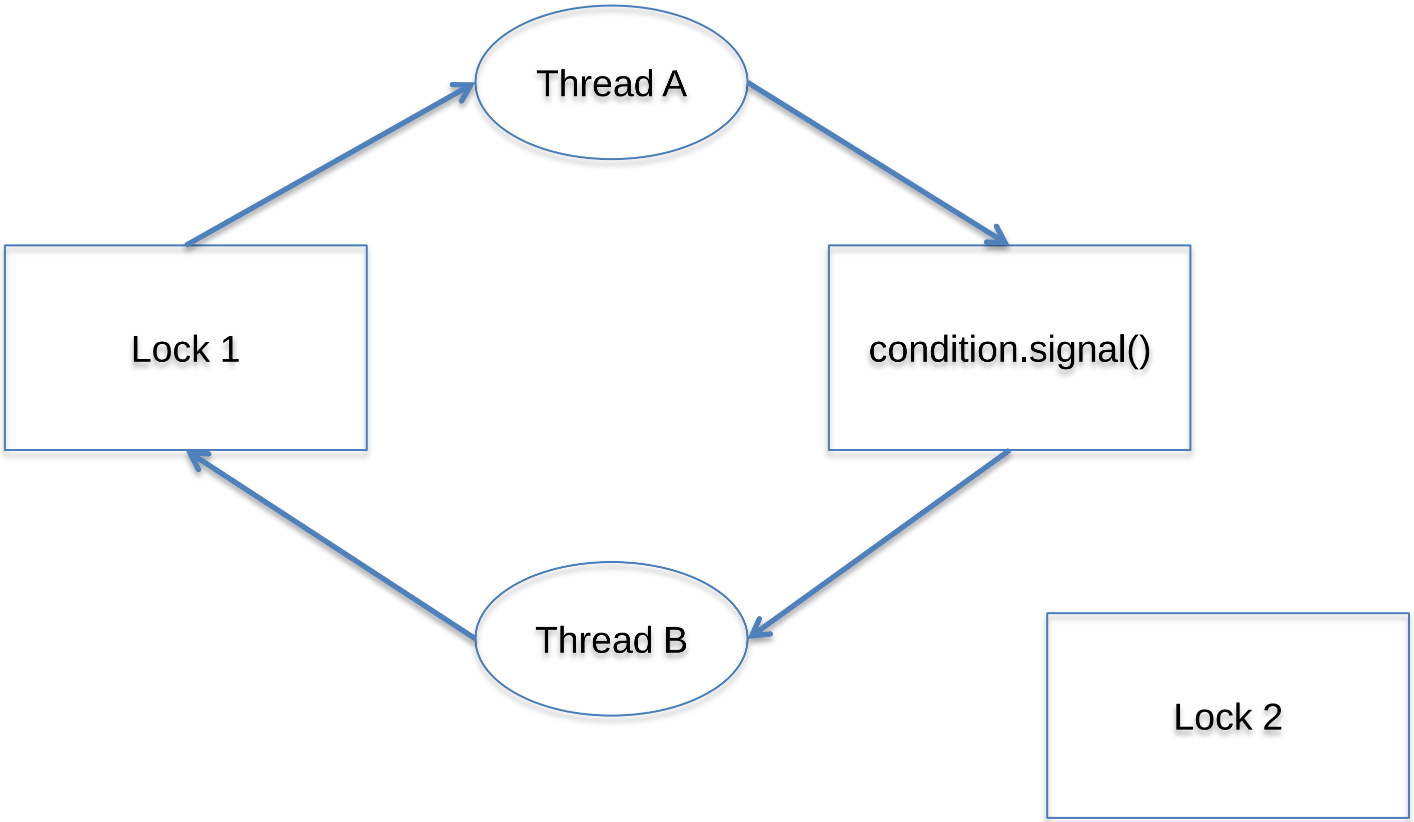
Thread A

```
lock1.acquire();
…
lock2.acquire();
while (need to wait)
   condition.wait(lock2);
lock2.release();
…
lock1.release();
```

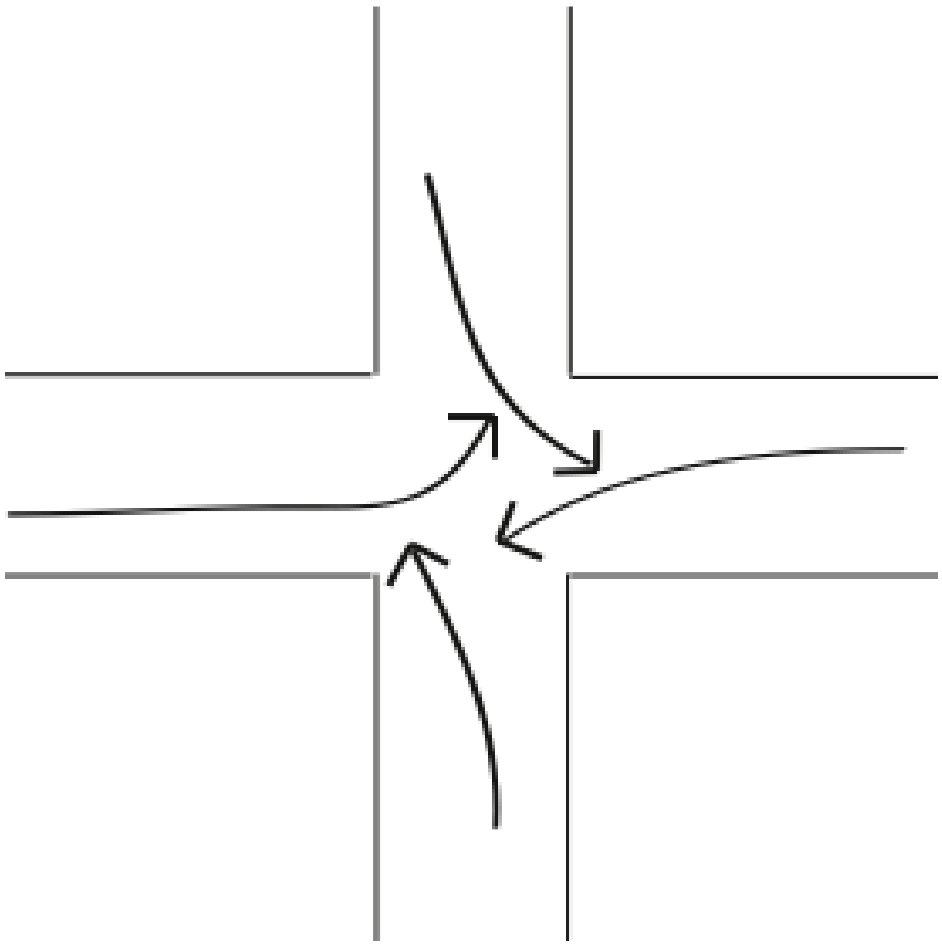
Thread B

```
lock1.acquire();
…
lock2.acquire();
….
condition.signal(lock2);
lock2.release();
…
lock1.release();
```

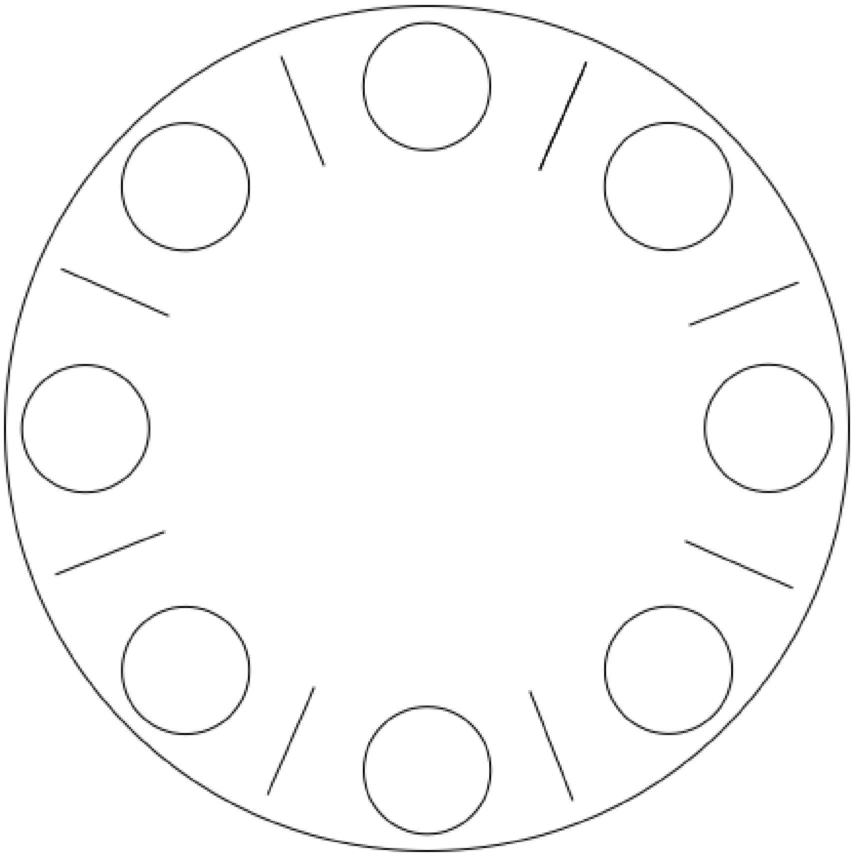
Example 3 Waiting-for Graph



Classic Deadlock Example (Multiple resources)



Famous Abstract Example: Dining Philosophers



Each philospher needs two chopsticks to eat.
Each grabs chopstick on the right first.

Conditions for Deadlock

- Bounded resources
  - If infinite resources, no deadlock!
- No preemption
  - Once acquired, resource cannot be taken away
- Hold while waiting
  - Don’t (voluntarily) relinquish resource when have to wait
- Circular “waiting-for” relationships

What to do about deadlock?

- Ensure that one of the four conditions doesn't hold
  - *Detection:* build waits-for graph and look for cycles. If you find one, do something extraordinary.
  - *Pseudo-detection:* if you make no progress for a long time, guess there's deadlock and do something extraordinary
  - *Prevention:* write code whose structure prevents at least one of the four conditions from holding

Deadlock Example 1: Two Locks

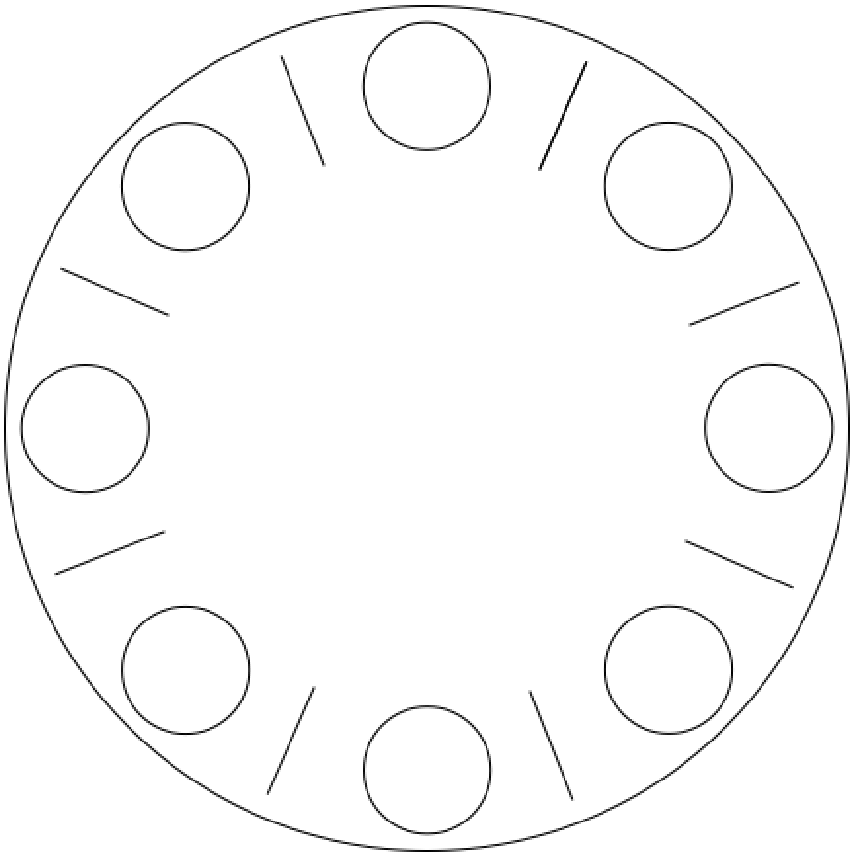
Thread A

```
lock1.acquire();
lock2.acquire();
// update objs 1 and 2
lock2.release();
lock1.release();
```

Thread B

```
lock2.acquire();
lock1.acquire();
// update objs 1 and 2
lock1.release();
lock2.release();
```

Famous Abstract Example: Dining Philosophers



Each philospher needs two chopsticks to eat.
Each grabs chopstick on the right first.

Deadlock Example 3: Two locks and a condition variable

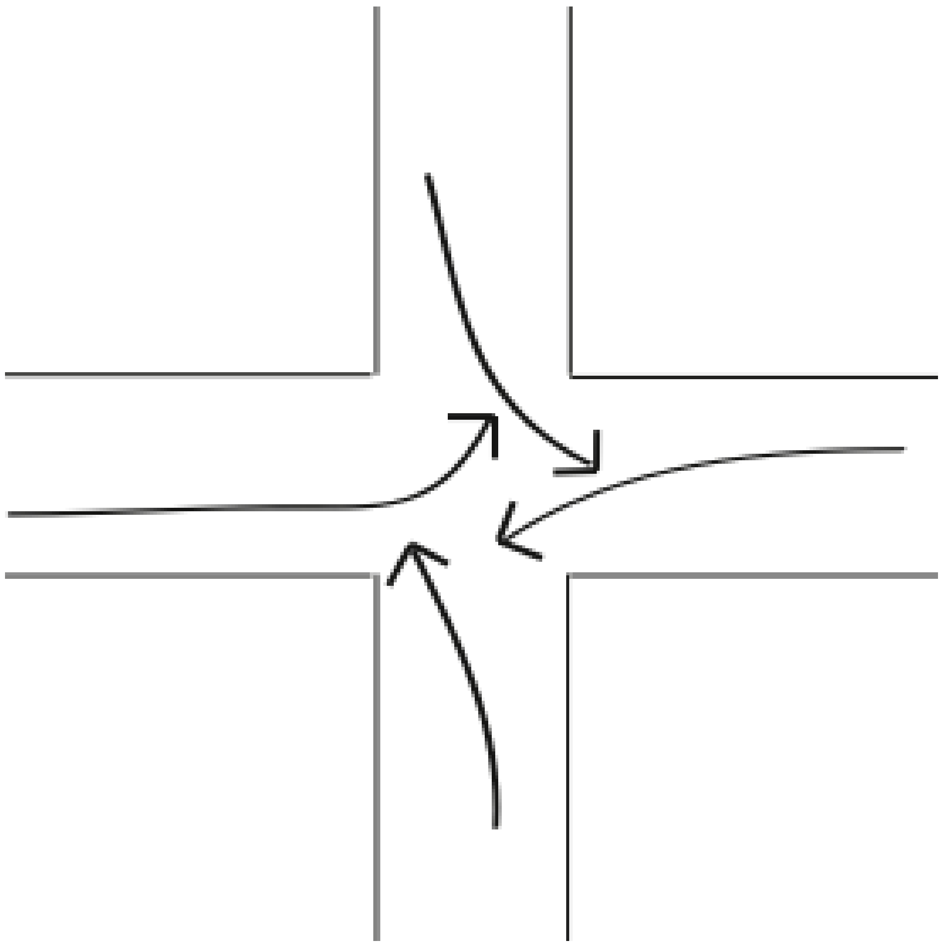
Thread A

```
lock1.acquire();
…
lock2.acquire();
while (need to wait)
   condition.wait(lock2);
lock2.release();
…
lock1.release();
```

Thread B

```
lock1.acquire();
…
lock2.acquire();
….
condition.signal(lock2);
lock2.release();
…
lock1.release();
```

Classic Deadlock Example (Multiple resources)



1: Deadlock Detection (and Breaking)

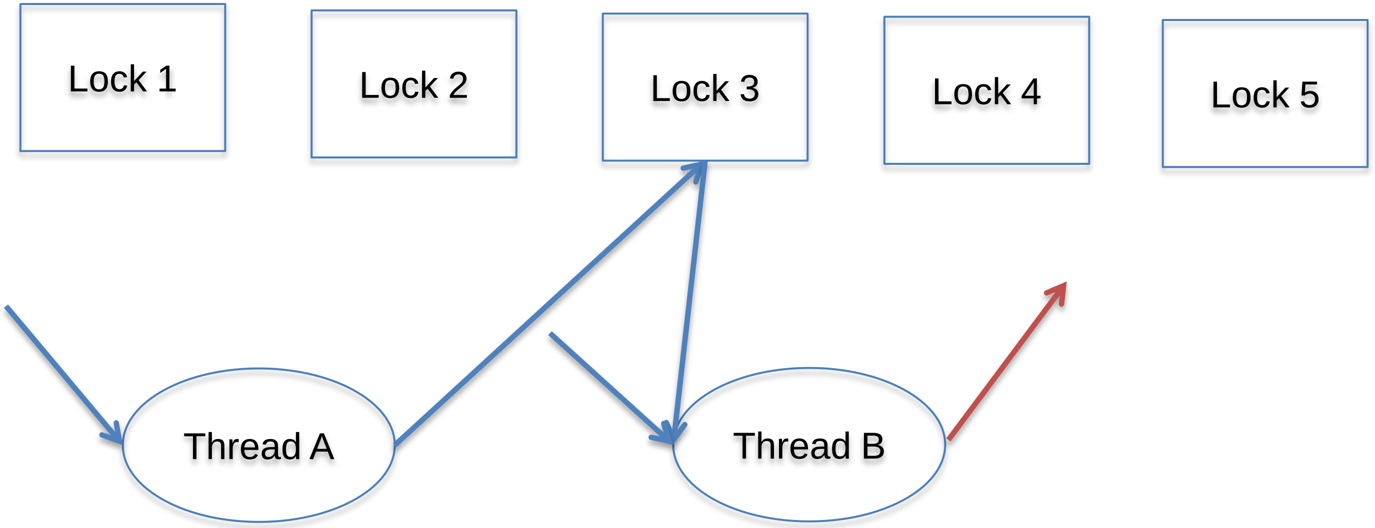
- Algorithm
  - Scan wait-for graph
  - Detect cycles
  - Fix cycles
- Fix cycles how?
  - Remove one thread, reassign its resources
    - Requires exception handling code to be very robust
  - Roll back actions of one thread
    - Databases: all actions are provisional until committed

2: Deadlock Prevention: Lock Ordering

Eliminate one of the four conditions for deadlock

- Lock ordering
  - Always acquire locks in the same order
    - Example: move file from one directory to another
      - Danger: concurrent moves in opposite directions
  - Widely used in OS kernels (and concurrent apps!)
- Infinite resources?
  - Ex: UNIX reserves a process for the sysadmin to run "kill"

Waits-for with Lock Ordering



2: Deadlock Prevention: Infinite Resources

- Infinite resources?
  - Example: UNIX reserves a process for the sysadmin to run “kill”

1.5: Pseudo-detection (or maybe prevention)

- Design system to release resources and retry if need to wait
  - No “wait while holding”
  - Could be done by the application itself or by some supporting layer (e.g., the OS) or by some mix of layers

- Example: (system) timeout and (app) roll-back
  - provide an “acquire with timeout” interface for synch objects
    - Either you get the resource by the timeout or you stop waiting without getting it
  - application includes recovery code for timeout events
    - Can be complicated to do if application has already updated some state when timeout occurs
      - Rollback

1.5: Pseudo-detection (or maybe prevention)

- Bright idea:
  Try to acquire all needed resources in advance
  - First acquire all resources
  - If a timeout occurs, you haven't modified any state, and so rollback is easy!
  - On the other hand, it's impossible to implement unless you can figure out all the resources you'll need before you've computed anything
  - (and, what about livelock?)

Prevention: Banker's Algorithm

- Acquiring in advance all resources you *might* use is wasteful
- Instead, allow thread to acquire them dynamically, with no discipline at all
- Costs:
  - must declare maximum resources you might require
  - system may delay fulfilling a resource request even though the resource is available

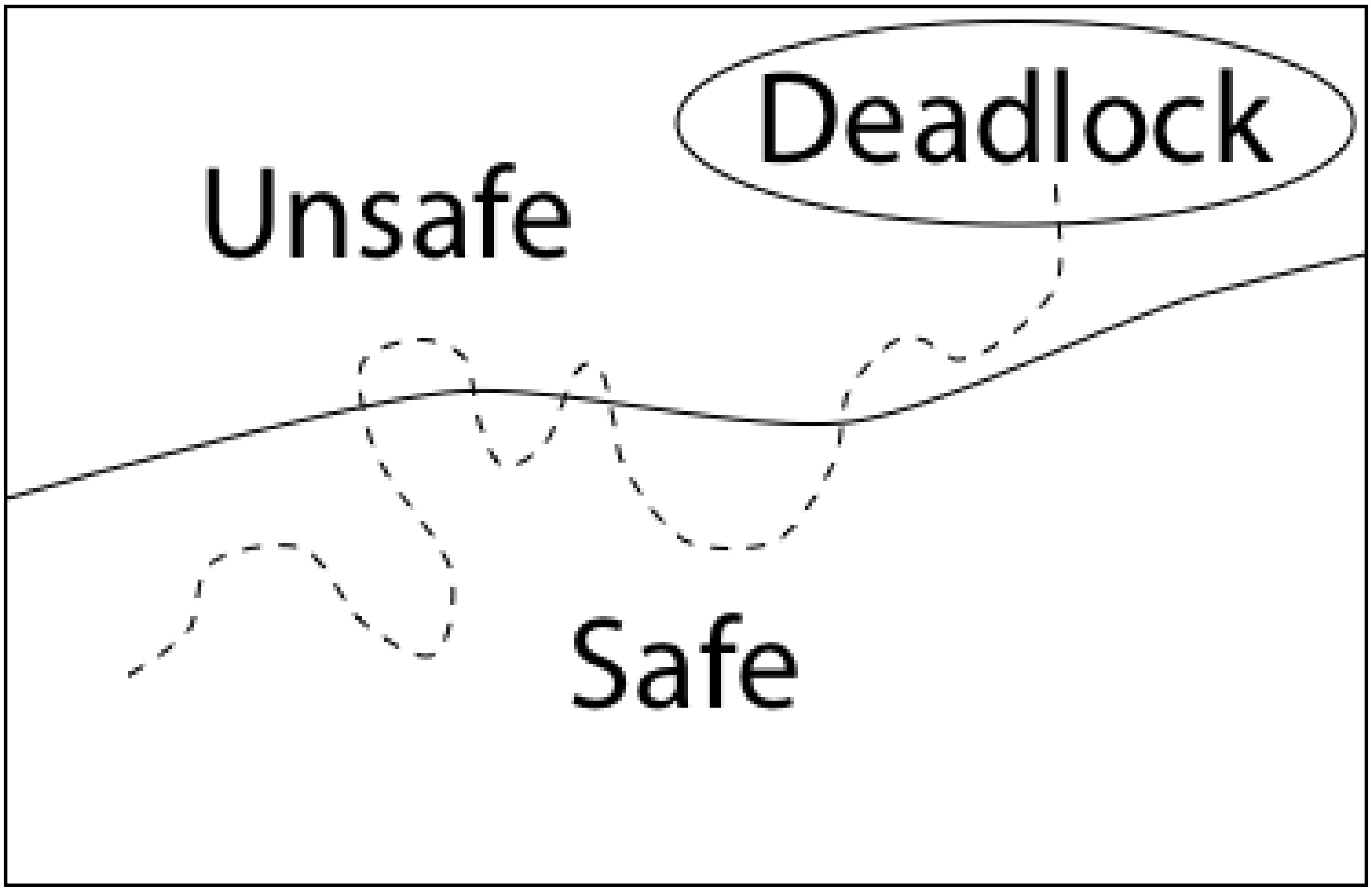
Prevention: Banker’s Algorithm

- Banker’s algorithm
  - Declare maximum resource needs in advance
  - Allocate resources dynamically when resource is needed
    - wait if granting request could possibly lead to deadlock
    - implies you allocate a requested resource only if you’re sure you can find a thread schedule that allows all threads to complete, even if they all request their maximums

Definitions

- Safe state:
  – For every possible sequence of future resource requests (that respect the declared maximums), it is possible to *eventually* grant all requests
- Unsafe state:
  – Some sequence of resource requests can result in deadlock
- Doomed state:
  – You're in deadlock

Possible System States



Bankers' Algorithm

- Grant request iff result is a safe state
  - If a thread makes a request that, if fulfilled, would cause system to move to an unsafe state, suspend execution of that thread
  - Otherwise, allocate resource to thread now

Banker's Algorithm Example

- Example:
  - 9 units of resource available total
  -

	Current Allocation	Maximum Need
Thread 0	0	3
Thread 1	3	5
Thread 2	4	7

  - This is a safe state, because we can certainly finish thread 1 (by pausing other two), then thread 2 then thread 0

Banker’s Algorithm Example

- Thread 1 requests an additional unit
- Is it granted?

(9 units total)

	Current Allocation	Maximum Need
Thread 0	0	3
Thread 1	3	5
Thread 2	4	7

Banker's Algorithm Example

- Thread 0 requests an additional unit
- Is it granted?

(9 units total)

	Current Allocation	Maximum Need
Thread 0	0	3
Thread 1	3	5
Thread 2	4	7

Banker's Algorithm: Dining Philosophers

- n chopsticks in middle of table
- n philosophers, each can take one chopstick at a time, and up to two total

- When is it ok for lawyer to take a chopstick?
- What if each lawyer could need up to n chopsticks?