Multi-Object Synchronization

Synchronization Performance

- A program with lots of concurrent threads can still have poor performance on a multiprocessor:
  - Overhead of creating threads, if not needed
  - Lock contention: only one thread at a time can hold a given lock
  - Shared data protected by a lock may ping back and forth between cores
  - False sharing: communication between cores even for data that is not shared

Atomic CompareAndSwap

- Operates on a memory word
- Check that the value of the memory word hasn't changed from what you expect
  - E.g., no other thread did compareAndSwap first
- If it has changed, return an error (and loop)
- If it has not changed, set the memory word to a new value

Multi-Object Automicity

- Combining into one object, e.g., using a function
  - checkforMilkAndSetNoteifNeeded()
- Acquire-All/Release All (Serializable)
  - AND SYNCHRONIZATION
  - Must know all required locks *a priori*
- Two-Phase Locking
  - Acquire locks as needed and do not release locks until all locks have been acquired
  - How to avoid deadlock?

Deadlock Definition

- Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  - Preemptable: can be taken away by OS
  - Non-preemptable: must leave with thread
- Starvation: thread waits indefinitely
- Deadlock: circular waiting for resources
  - Deadlock => starvation, but not vice versa

Example: two locks

Thread A	Thread B
lock1.acquire();	lock2.acquire();
lock2.acquire();	lock1.acquire();
lock2.release();	lock1.release();
lock1.release();	lock2.release();

Bidirectional Bounded Buffer

Thread A	Thread B
buffer1.put(data);	buffer2.put(data);
buffer1.put(data);	buffer2.put(data);
buffer2.get();	buffer1.get();
buffer2.get();	buffer1.get();

Suppose buffer1 and buffer2 both start almost full.

Two locks and a condition variable

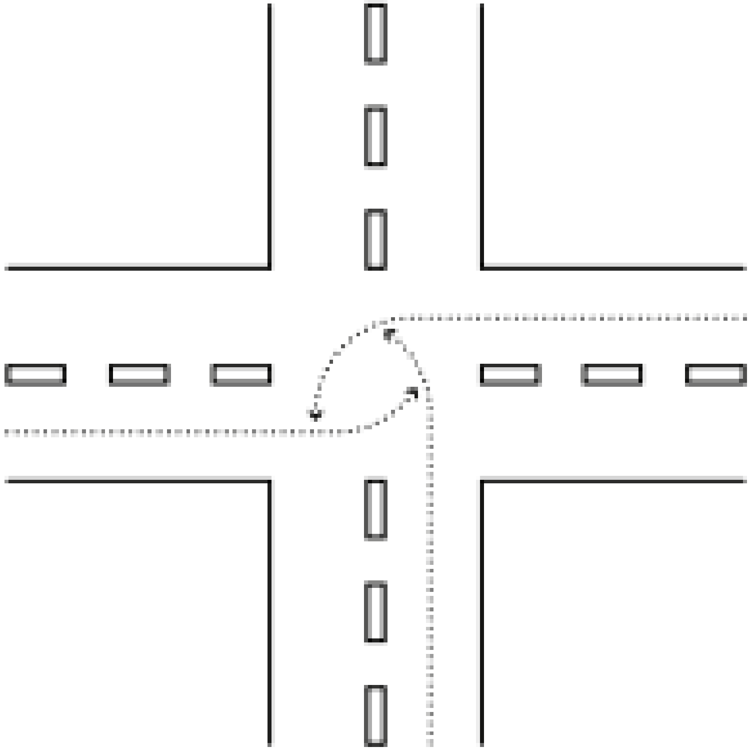
Thread A

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

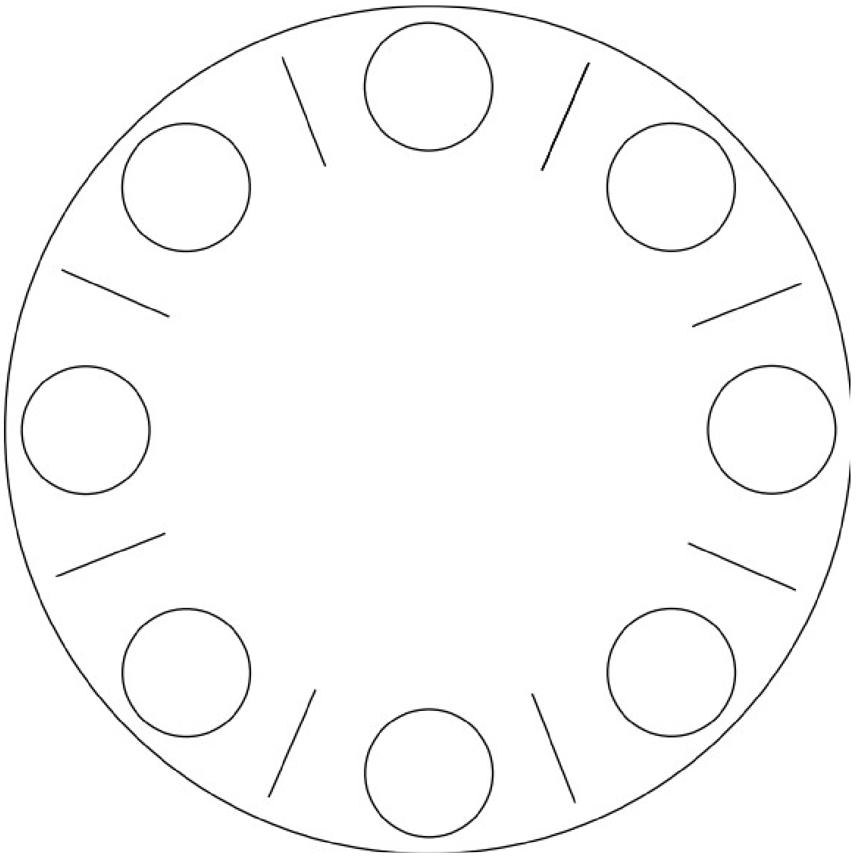
Thread B

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

Yet another Example



Dining Lawyers



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

Necessary Conditions for Deadlock

- Limited access to resources
  - If infinite resources, no deadlock!
- No preemption
  - If resources are virtual, can break deadlock
- Multiple independent requests
  - “wait while holding”
- Circular chain of requests

Question

- How does Dining Lawyers meet the necessary conditions for deadlock?
  - Limited access to resources
  - No preemption
  - Multiple independent requests (wait while holding)
  - Circular chain of requests

- How can we modify Dining Lawyers to prevent deadlock?

Preventing Deadlock

- Exploit or limit program behavior
  - Limit program from doing anything that might lead to deadlock
- Predict the future
  - If we know what program will do, we can tell if granting a resource might lead to deadlock
- Detect and recover
  - If we can rollback a thread, we can fix a deadlock once it occurs

Exploit or Limit Behavior

- Provide enough resources
  - How many chopsticks are enough?
- Eliminate wait while holding
  - Release lock when calling out of module
  - Telephone circuit setup
- Eliminate circular waiting
  - Lock ordering: always acquire locks in a fixed order
  - Example: move file from one directory to another

Example

Thread 1

1. Acquire A
2.
3. Acquire C
4.
5. If (maybe) Wait for B

Thread 2

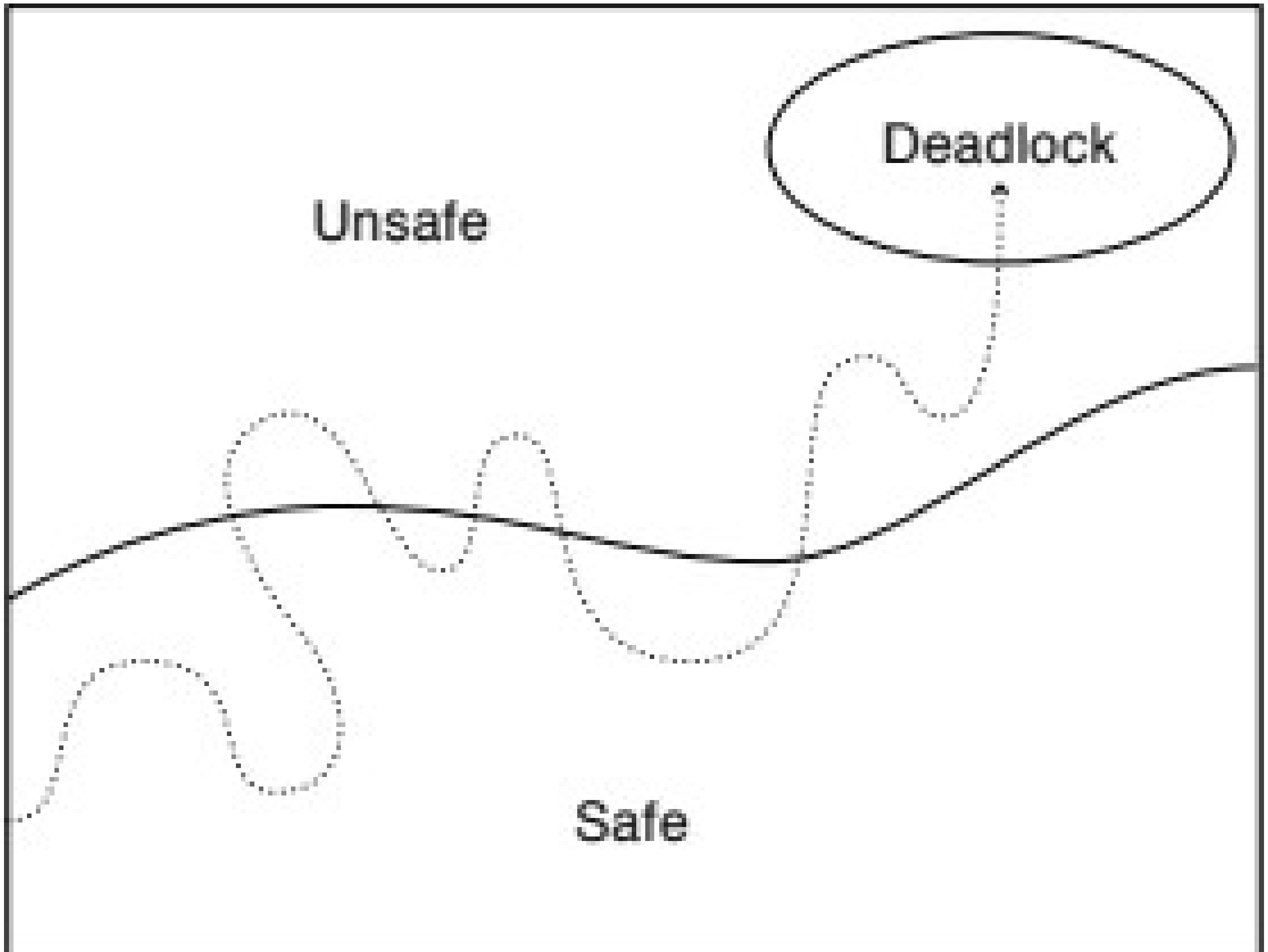
1.
2. Acquire B
3.
4. Wait for A

How can we make sure to avoid deadlock?

Deadlock Dynamics

- Safe state:
  - For any possible sequence of future resource requests, it is possible to eventually grant all requests
  - May require waiting even when resources are available!
- Unsafe state:
  - Some sequence of resource requests can result in deadlock
- Doomed state:
  - All possible computations lead to deadlock

Possible System States



Question

- What are the doomed states for Dining Lawyers?

- What are the unsafe states?

- What are the safe states?

Communal Dining Lawyers

- n chopsticks in middle of table
- n lawyers, each can take one chopstick at a time
- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Dining Lawyers

- N chopsticks in the middle of the table
- N lawyers, each takes one chopstick at a time
- Lawyers need k chopsticks to eat, $k > 1$

- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Communal Mutant Absent-Minded Dining Lawyers

- N chopsticks in the middle of the table
- N lawyers, each takes one chopstick at a time
- Lawyers need k chopsticks to eat, k > 1
  - k larger if lawyer is talking on his/her cellphone

- What are the safe states?
- What are the unsafe states?
- What are the doomed states?

Predict the Future

- Banker's algorithm
  - State maximum resource needs in advance
  - Allocate resources dynamically when resource is needed -- wait if granting request would lead to deadlock
  - Request can be granted if some sequential ordering of threads is deadlock free

Banker's Algorithm

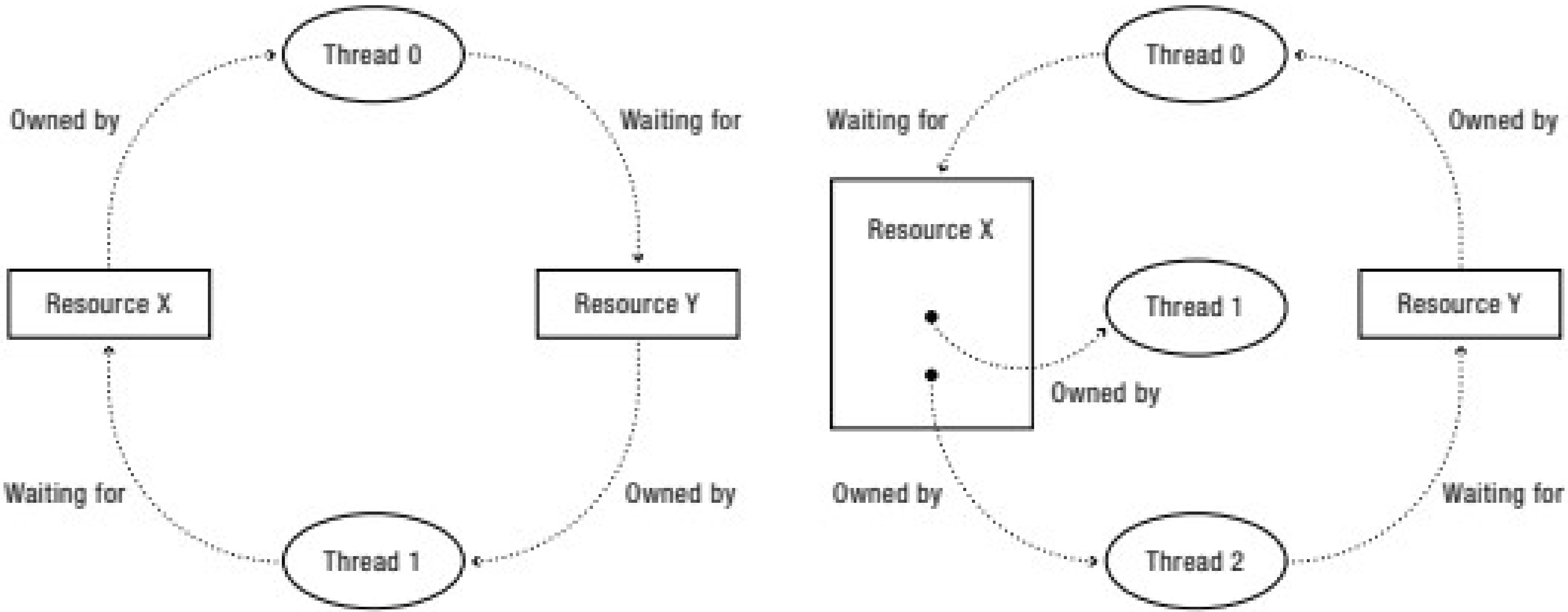
- Grant request iff result is a safe state
- Sum of maximum resource needs of current threads can be greater than the total resources
  - Provided there is some way for all the threads to finish without getting into deadlock
- Example: proceed iff
  - total available resources - # allocated >= max remaining that might be needed by this thread in order to finish
  - Guarantees this thread can finish

Check Figure 6.20 and 6.21 for detailed pseudo-code.

Detect and Repair

- Algorithm
  - Scan wait for graph
  - Detect cycles
  - Fix cycles
- Proceed without the resource
  - Requires robust exception handling code
- Roll back and retry
  - Transaction: all operations are provisional until have all required resources to complete operation

Detecting Deadlock



An example of a non-deadlocked scenario is mis-classified as being deadlocked

More precise: Figure 6.23

Non-Blocking Synchronization

- Goal: data structures that can be read/modified without acquiring a lock
  - No lock contention!
  - No deadlock!
- General method using compareAndSwap
  - Create copy of data structure
  - Modify copy
  - Swap in new version iff no one else has
  - Restart if pointer has changed

Lock-Free Bounded Buffer

```
tryget() {
    do {
        copy = ConsistentCopy(p);
        if (copy->front == copy->tail)
            return NULL;
        else {
            item = copy->buf[copy->front % MAX];
            copy->front++;
     } while (compareAndSwap(&p, p, copy));
    return item;
}
```