Synchronization

Operating Systems

Department of Computer Science
Rutgers University

Synchronization

Problem

Threads may share data

Data consistency must be maintained

Example

Suppose a thread wants to withdraw $5 from a bank account and another thread wants to deposit $10 to the same account

What should the balance be after the two transactions have been completed?

What might happen instead if the two transactions were executed concurrently?

Synchronization (cont)

The balance might be CB – 5

Thread 1 reads CB (current balance)

Thread 2 reads CB

Thread 2 computes CB + 10 and saves new balance

Thread 1 computes CB – 5 and saves new balance

The balance might be CB + 10

How?

Ensure the orderly execution of cooperating threads

Terminology

Critical section: (CS) a section of code which reads or writes shared data

Race condition: potential for interleaved execution of a critical section by multiple threads

Results are non-deterministic

Mutual exclusion: synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical sections

Deadlock: permanent blocking of threads

Livelock: execution but no progress

Starvation: one or more threads denied resources

Synchronization Primitives

Most common primitives

- Locks (mutual exclusion)
- Condition variables
- Semaphores
- Monitors
- Barriers

Need

- Semaphores
- Locks and condition variables

Locks: Requirements for Mutual Exclusion

- No assumptions on hardware: speed, # of processors
- Mutual exclusion is maintained – that is, only one thread at a time can be executing inside a CS
- Execution of the CS takes a finite time
- A thread not in the CS cannot prevent other threads to enter the CS
- Entering the CS cannot be delayed indefinitely: no deadlock or starvation

Locks

Mutual exclusion ≡ want to be the only thread modifying a set of data items

Can look at it as exclusive access to data items or to a piece of code

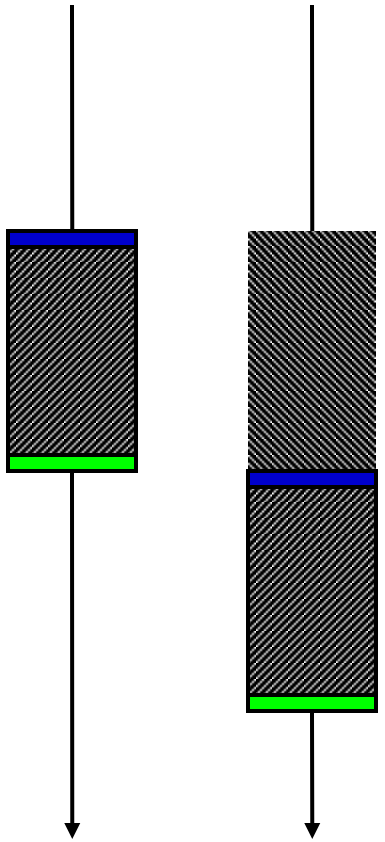
Have three components:

Acquire, Release, Waiting

Examples:

```
Acquire(A)
Acquire(B)
A ← A + 10
B ← B - 10
Release(B)
Release(A)
```

```
Function Transfer (Amount, A, B)
    Acquire(Transfer_Lock)
    A ← A + 10
    B ← B - 10
    Release(Transfer_Lock)
```



Example

```
public class BankAccount
{
    Lock aLock = new Lock;
    int balance = 0;

    ...

    public void deposit(int amount)
    {
            aLock.acquire();
            balance = balance + amount;
            aLock.release();
    }
```

```
public void withdrawal(int amount)
{
            aLock.acquire();
            balance = balance - amount;
            aLock.release();
}
```

Implementing (Blocking) Locks Inside OS Kernel

From Nachos (with some simplifications)

```
public class Lock {
    private KThread lockHolder = null;
    private ThreadQueue waitQueue =
              ThreadedKernel.scheduler.newThreadQueue(true);

    public void acquire() {
          KThread thread = KThread.currentThread();  // Get thread object (TCB)
          if (lockHolder != null) {                  // Gotta wait
              waitQueue.waitForAccess(thread);       // Put thread on wait queue
              KThread.sleep();                       // Context switch
          }
          else
              lockHolder = thread;                   // Got the lock
    }
```

Implementing (Blocking) Locks Inside OS Kernel

```
    public void release() {
            if ((lockHolder = waitQueue.nextThread()) != null)
                lockHolder.ready();        // Wake up a waiting thread
    }
}
```

This implementation is not quite right … what's missing?

Implementing (Blocking) Locks Inside OS Kernel

```
    public void release() {
            if ((lockHolder = waitQueue.nextThread()) != null)
                lockHolder.ready();        // Wake up a waiting thread
    }
}
```

This implementation is not quite right … what's missing?
Mutual exclusion when accessing lockHolder and waitQueue.
An approach is to protect against interrupts.

Implementing (Blocking) Locks Inside OS Kernel

```
public void release() {
        boolean intStatus = Machine.interrupt().disable();

        if ((lockHolder = waitQueue.nextThread()) != null)
            lockHolder.ready();

        Machine.interrupt().restore(intStatus);
}
```

acquire() also needs to block interrupts

Unfortunately, disabling interrupts only works for uniprocessors. We'll get back to this implementation later…

Implementing Locks At User-Level

Why?

Expensive to enter the kernel

What's the problem?

Can't disable interrupts …

Many software algorithms for mutual exclusion

See any OS book

Disadvantages: very difficult to get correct

So what do we do?

Implementing Locks At User-Level

Simple with a "little bit" of help from the hardware

Atomic read-modify-write instructions

Test-and-set

Atomically read a variable and, if the value of the variable is currently 0, set it to 1

Fetch-and-increment

Atomically return the current value of a memory location and increment the value in memory by 1

Compare-and-swap

Atomically compare the value of a memory location with an old value, and if the same, replace with a new value

Modern architectures perform atomic operations in the cache

Implementing **Spin** Locks Using Test-and-Set

```
#define UNLOCKED 0
#define LOCKED 1

Spin_acquire(lock)
{
        while (test-and-set(lock) == LOCKED);
}

Spin_release(lock)
{
        lock = UNLOCKED;
}
```

Problems?

Implementing **Spin** Locks Using Test-and-Set

```
#define UNLOCKED 0
#define LOCKED 1

Spin_acquire(lock)
{
        while (test-and-set(lock) == LOCKED);
}

Spin_release(lock)
{
        lock = UNLOCKED;
}
```

Problems? Lots of memory traffic if TAS always sets; lots of traffic when lock is released; no ordering guarantees. Solutions?

Spin Locks Using Test and Test-and-Set

```
Spin_acquire(lock)
{
                    while (1)  {
                                while (lock == LOCKED);
                                if (test-and-set(lock) ==
UNLOCKED) break;
                    }
}

Spin_release(lock)
{
                    lock = UNLOCKED;
}
```

Better, since TAS is guaranteed not to generate traffic unnecessarily. But there is still lots of traffic after a release. Still no ordering guarantees.

Spin Locks Using Fetch-and-Increment

- Ticket lock using fetch-and-increment:
  - Each thread gets a ticket from variable *next-ticket*
  - *Now-serving* variable holds ticket of current lock holder
- Think about how to implement acquire and release!
- Many other spin lock implementations exist

Implementing (Spin) Barriers

- Centralized barrier:
  - Each thread increments a shared counter upon arriving
  - Each thread polls the shared counter until all have arrived

```
Barrier (num_threads)
{
        if (fetch-and-increment (counter) ==
num_threads)
                counter = 0;
        else
                while (counter != 0);
}
```

Problems?

Implementing (Spin) Barriers

- Centralized barrier:
  - Each thread increments a shared counter upon arriving
  - Each thread polls the shared counter until all have arrived

```
Barrier (num_threads)
{
        if (fetch-and-increment (counter) ==
num_threads)
                counter = 0;
        else
                while (counter != 0);
}
```

Problems? Consecutive barriers may mess shared counter.
Contention for shared counter. Solutions?

Implementing (Spin) Barriers

- Centralized barrier with sense reversal:
  - Odd barriers wait for sense flag to go from true to false
  - Even barriers wait for sense flag to go from false to true

```
Barrier (num_threads)
{
        local_sense = ! local_sense;
        if (fetch-and-increment (counter) ==
num_threads) {
                counter = 0;
                sense = local_sense;
        }  else
                while (sense != local_sense);
}
```

Implementing (Spin) Barriers

- Previous implementation still suffers from contention
- Possible solution: combining tree barrier with sense reversal
  - Writes done in a tree; only *tree-degree* threads write to same counter
  - Last arrival at each level goes further up
  - Thread that arrives at the root wakes up the others by changing the sense variables on which they are spinning
- Think about how to implement this combining tree barrier!
- Many other spin barrier implementations exist

Implementing Blocking Locks Using Test-and-Set

Getting back to implementing blocking locks in the kernel…

```
public class Lock
{
    private int val = UNLOCKED;
    private ThreadQueue waitQueue = new ThreadQueue();

    public void acquire() {
            Thread me = Thread.currentThread();
            while (TestAndSet(val) == LOCKED) {
                waitQueue.waitForAccess(me);  // Put self in queue
                Thread.sleep();                       // Put self to sleep
            }
            // Got the lock
    }
```

Implementing Blocking Locks Using Test-and-Set

```
    public void release() {
            Thread next = waitQueue.nextThread();
            val = UNLOCKED;
            if (next != null)
                next.ready();     // Wake up a waiting thread
    }
}
```

Does this implementation work as is? No, for two reasons: (1) we need mutual exclusion in the access to the wait queue; and (2) we need an extra check of the lock in the acquire (the releaser may release the lock after a thread finds the lock busy, but before it enqueues itself).

Blocking Locks Using Test-and-Set

```
public void acquire() {
    Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
            Machine.interrupt().disable();
            if (val == LOCKED) {
                waitQueue.waitForAccess(me);  // Put self in queue
                Machine.interrupt().enable();
                Thread.sleep();                    // Put self to sleep
            }
            else {
                Machine.interrupt().enable();
            }
        }
        // Got the lock
}
```

Blocking Locks Using Test-and-Set

```
public void release() {
        Machine.interrupt().disable();
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        Machine.interrupt().enable();
        if (next != null)
            next.ready();     // Wake up a waiting thread
}
```

Disabling interrupts doesn't do the job in multiprocessors and is generally undesirable. So, what should be do instead?

Blocking Locks Using Test-and-Set: Correct

```
public void acquire() {
    Thread me = Thread.currentThread();
        while (TestAndSet(val) == LOCKED) {
            spin_lock(another_lock);
            if (val == LOCKED) {
                waitQueue.waitForAccess(me);  // Put self in queue
                spin_unlock(another_lock);
                Thread.sleep();                    // Put self to sleep
            }
            else {
                spin_unlock(another_lock);
            }
        }
        // Got the lock
}
```

Blocking Locks Using Test-and-Set: Correct

```
public void release() {
        spin_lock(another_lock);
        Thread next = waitQueue.nextThread();
        val = UNLOCKED;
        spin_unlock(another_lock);
        if (next != null)
            next.ready();     // Wake up a waiting thread
}
```

What happens if a thread is killed when holding a lock?

What happens when a thread is woken up as "next" above but there are more threads waiting for the lock?

What To Do While Waiting?

We have considered two types of primitives:

Blocking

- OS or RT system de-schedules waiting threads

Spinning

- Waiting threads keep testing location until it changes value
- Hmm … doesn't quite work in single-threaded uniprocessor, does it?

Spinning vs. blocking becomes an issue in multithreaded processors and multiprocessors

What is the main tradeoff?

What To Do While Waiting?

We have considered two types of primitives:

Blocking

- OS or RT system de-schedules waiting threads

Spinning

- Waiting threads keep testing location until it changes value
- Hmm … doesn't quite work in single-threaded uniprocessor, does it?

Spinning vs. blocking becomes an issue in multithreaded processors and multiprocessors

What is the main tradeoff? Overhead of blocking vs. expected waiting time

Condition Variables

A condition variable is always associated with:

A condition

A lock

Typically used to wait for the condition to take on a given value

Three operations:

```
public class CondVar
{
    public Wait(Lock lock);
    public Signal();
    public Broadcast();
    // ... other stuff
}
```

Condition Variables

Wait(Lock lock)

Release the lock

Put thread object on wait queue of this CondVar object

Yield the CPU to another thread

When waken by the system, reacquire the lock and return

Signal()

If at least 1 thread is sleeping on cond_var, wake 1 up. Otherwise, no effect

Waking up a thread means changing its state to Ready and moving the thread object to the run queue

Broadcast()

If 1 or more threads are sleeping on cond_var, wake everyone up

Otherwise, no effect

Producer/Consumer Example

Imagine a web server with the following architecture:

One "producer" thread listens for client http requests

When a request is received, the producer enqueues it on a circular request queue with finite capacity (if there is room)

A number of "consumer" threads service the queue as follows

Remove the 1st request from the queue (if there is a request)

Read data from disk to service the request

How can the producer and consumers synchronize?

Producer/Consumer (cont)

```
public class SyncQueue
{
    public boolean IsEmpty();
    public boolean IsFull();
    public boolean Enqueue(Request r);
    public Request Dequeue();

    public LockVar lock = new Lock;
    public CondVar waitForNotEmpty = new CondVar(LockVar lock);
    public CondVar waitForNotFull = new CondVar(LockVar lock);

    ...
}
```

Producer

```
public class Producer extends Thread
{
    private SyncQueue requestQ;
    public Producer(SyncQueue q) {requestQ = q;}
    public void run()
    {
            // Accept a request from some client
            // The request is stored in the object newRequest
            requestQ.lock.Acquire();
            while (requestQ.IsFull()) {
                waitForNotFull.Wait(requestQ.lock);
            }
            requestQ.Enqueue(newRequest);
            waitForNotEmpty.Signal();
            requestQ.lock.Release();
    }
}
```

Consumer

```
public class Consumer extends Thread
{
    private SyncQueue requestQ;
    public Consumer(SyncQueue q) {requestQ = q;}
    public void run()
    {
            requestQ.lock.Acquire();
            while (requestQ.IsEmpty()) {
                waitForNotEmpty.Wait(requestQ.lock);
            }
            Request r = requestQ.Dequeue();
            waitForNotFull.Signal()
            requestQ.lock.Release();

            // Process the request
    }
}
```

Implementing Condition Variables

Condition variables are implemented using locks

Implementation is tricky because it involves multiple locks and scheduling queue

Implemented in the OS or run-time thread systems because they involve scheduling operations

Sleep/Wakeup

Can you see how to do this from our discussion of how to implement locks?

You may be asked to implement condition variables!

Semaphores

Synchronized counting variables

Formally, a semaphore comprises:

An integer value

Two operations: P() and V()

P()

While value == 0, sleep
Decrement value

V()

Increment value
If there are any threads sleeping waiting for value to become non-zero, wakeup at least 1 thread

Using Semaphores

Binary semaphores can be used to implement mutual exclusion

Initialize counter to 1

P == lock acquire

V == lock release

General semaphores (with the help of a binary semaphore) can be used in producer-consumer types of synchronization problems

Can you figure out how? Hint: You should learn how to do it.

Implementing Semaphores

Can you see how to implement semaphores given locks and condition variables?

Can you see how to implement locks and condition variables given semaphores?

Hint: if not, learn how

Monitors

Semaphores have a few limitations: unstructured, difficult to program correctly. Monitors eliminate these limitations and are as powerful as semaphores

A monitor consists of a software module with one or more procedures, an initialization sequence, and local data (can only be accessed by procedures)

Only one process can execute within the monitor at any one time (mutual exclusion) => entry queue

Synchronization within the monitor implemented with condition variables (wait/signal) => one queue per condition variable

Monitors: Syntax

```
Monitor monitor-name
{
        shared variable declarations

        procedure body P1 (…) {
                …
        }
        procedure body Pn (…) {
                …
        }

        {
                initialization code
        }
}
```

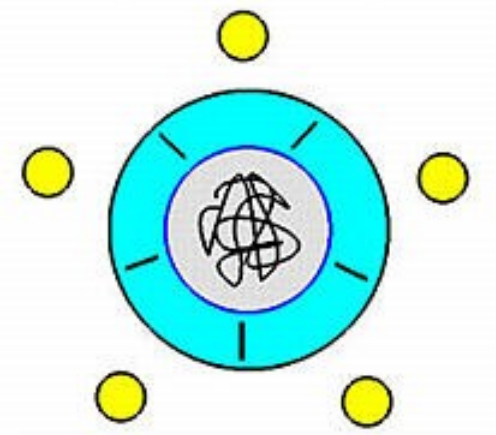
The Dining-Philosophers Problem

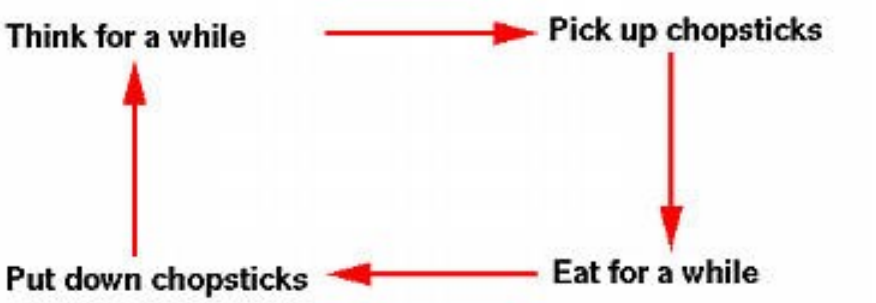
Let's solve the following problem together:

Consider 5 philosophers who spend their lives thinking and eating. The philosophers share a common circular table. Each philosopher has a bowl of rice (that magically replenishes itself). There are 5 chopsticks, each between a pair of philosophers. Occasionally, a philosopher gets hungry. He needs to get both the right and left chopsticks before he can eat. He eats for a while until he's full, at which point he puts the chopsticks back down on the table and resumes thinking.

How can we help the philosophers to synchronize their use of the chopsticks?

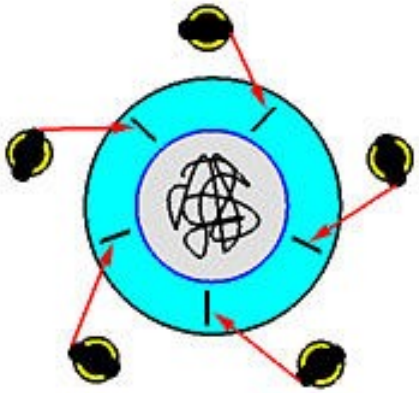
The Dining-Philosophers Problem

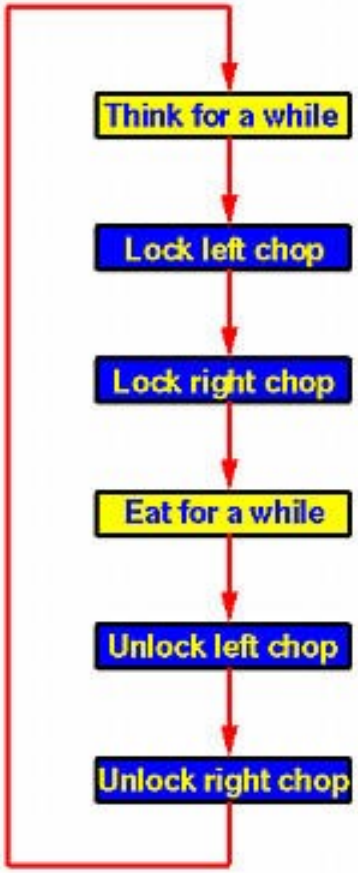


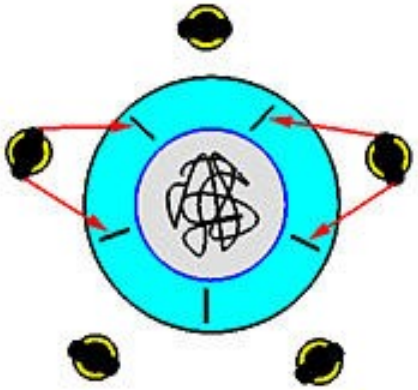


The Dining-Philosophers Problem

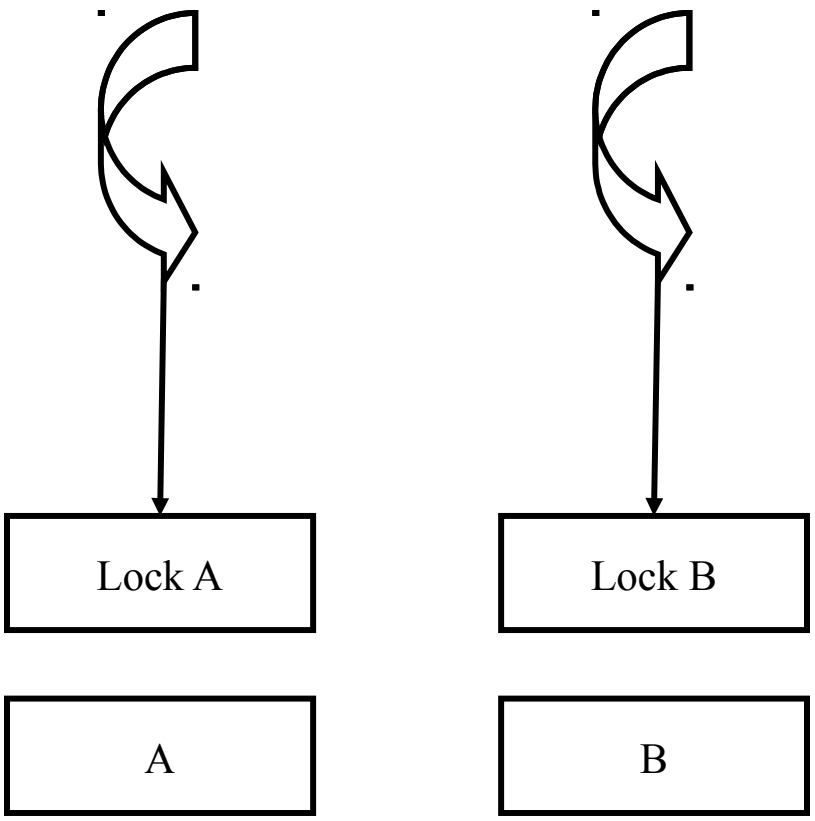
Any problems with alg?



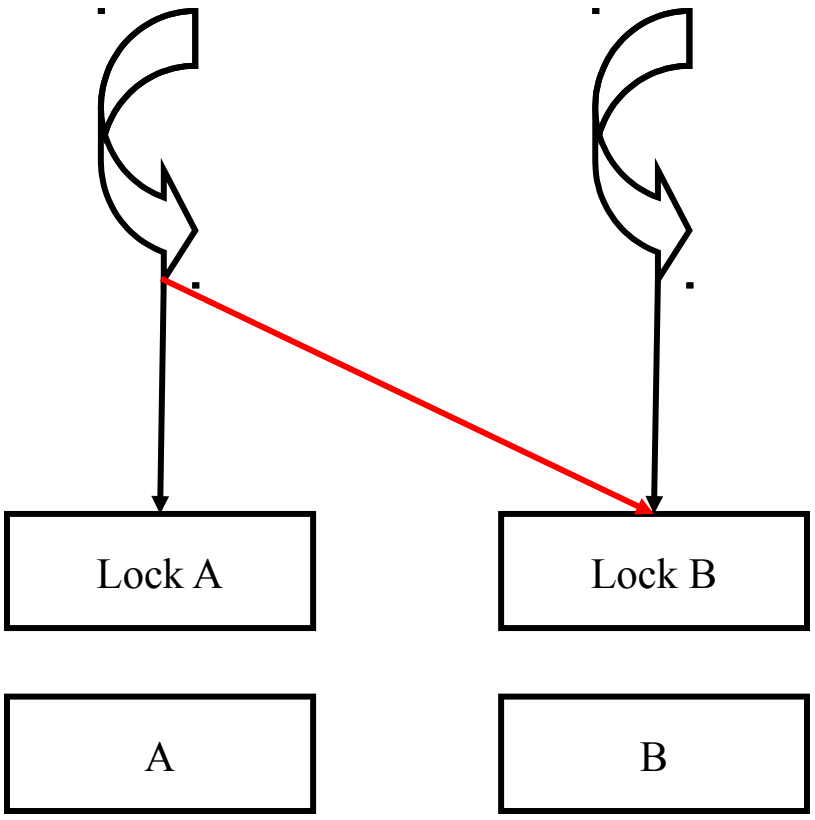




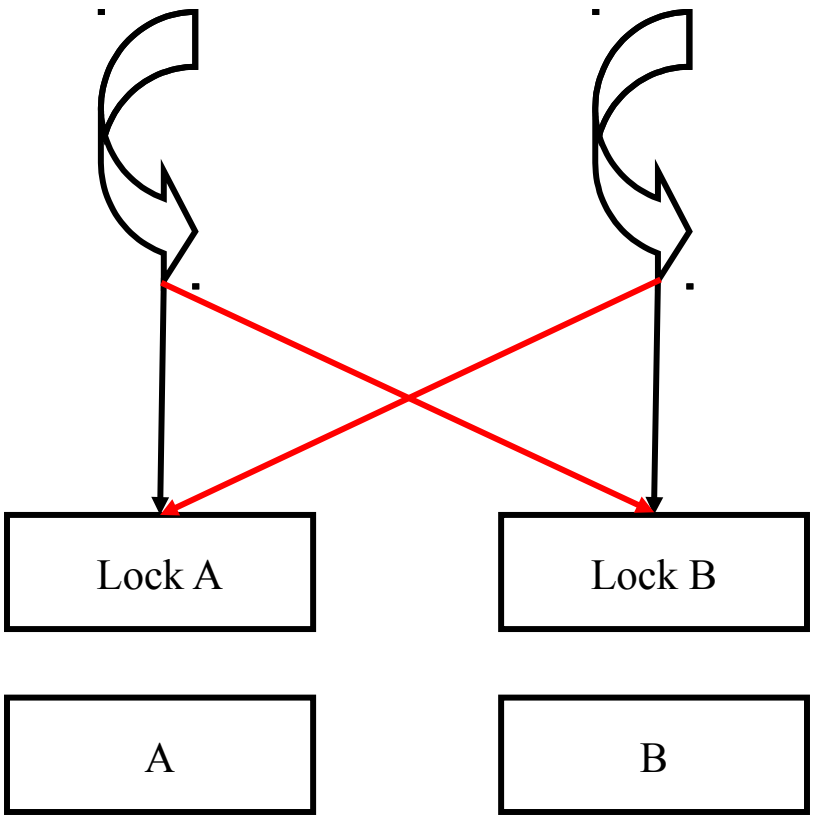
Deadlock



Deadlock



Deadlock



Deadlock (Cont'd)

Deadlock can occur whenever multiple parties are competing for exclusive access to multiple resources

How can we avoid deadlocks? Prevention, Avoidance, and Detection + Recovery

Necessary conditions for deadlock: mutual exclusion, hold and wait, no preemption, circular wait

- Hold and wait – a thread/process that needs a resource that is currently taken holds on to the resources it already has and waits for the resource
- No preemption – a thread/process never has a resource that it is holding taken away

What We Can Do About Deadlocks

Deadlock prevention

- Design a system without one of mutual exclusion, hold and wait, no preemption or circular wait (four necessary conditions)
- To prevent circular wait, impose a strict ordering on resources. For instance, if need to lock A and B, always lock A first, then lock B

Deadlock avoidance

- Deny requests that may lead to unsafe states (Banker's algorithm)
- Running the algorithm on all resource requests is expensive

Deadlock detection and recovery

- Check for circular wait periodically. If circular wait is found, abort all deadlocked processes (extreme solution but very common)
- Checking for circular wait is expensive

Banker's Algorithm

Idea: reject resource allocation requests that might leave the system in an "unsafe state".

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. Note that not all unsafe states are deadlock states.

This algorithm is conservative and simply avoids unsafe states altogether.

Banker's Algorithm (Cont'd)

Details:

- A new process must declare its maximum resource requirements (this number should not exceed the total number of resources in the system, of course)
- When a process requests a set of resources, the system must check whether the allocation of these resources would leave the system in an unsafe state
- If so, the process must wait until some other process releases enough resources

Banker's Algorithm (Cont)

Example: System has 12 tape drives

Processes	Maximum needs	Current allocation
P0	10	5
P1	4	2
P2	9	2

Is the system in a safe state?

What if we allocated another tape drive to P2?

Banker's Algorithm (Cont)

Example: System has 12 tape drives

Processes	Maximum needs	Current allocation
P0	10	5
P1	4	2
P2	9	2

Is system in a safe state? Yes. 3 tape drives are available and <P1, P0, P2> is a safe sequence.

What if we allocated another tape drive to P2? No. Only P1 could be allocated all its required resources. P2 would still require 6 drives and P0 would require 5, but only 4 drives would be available => potential for deadlock.