CS 422/522 Design & Implementation
of Operating Systems

Lecture 10: Multi-Object Synchronization

Zhong Shao
Dept. of Computer Science
Yale University

Acknowledgement: some slides are taken from previous versions of the CS422/522 lectures taught by Prof. Bryan Ford and Dr. David Wolinsky, and also from the official set of slides accompanying the OSPP textbook by Anderson and Dahlin.

Multi-object programs

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

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

Topics

- Multiprocessor cache coherence
- MCS locks (if locks are mostly busy)
- RCU locks (if locks are mostly busy, and data is mostly read-only)

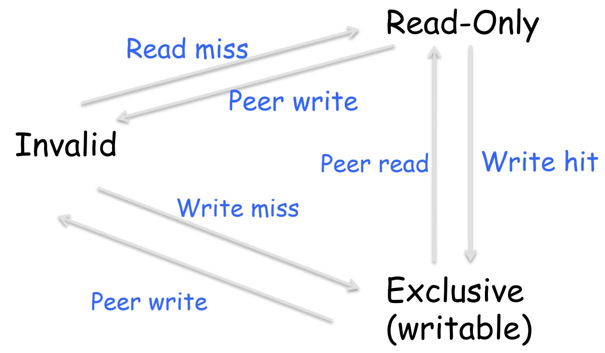
Multiprocessor cache coherence

- Scenario:
  - Thread A modifies data inside a critical section and releases lock
  - Thread B acquires lock and reads data
- Easy if all accesses go to main memory
  - Thread A changes main memory; thread B reads it
- What if new data is cached at processor A?
- What if old data is cached at processor B

Write-back cache coherence

- Cache coherence = system behaves as if there is one copy of the data
  - If data is only being read, any number of caches can have a copy
  - If data is being modified, at most one cached copy
- On write: (get ownership)
  - Invalidate all cached copies, before doing write
  - Modified data stays in cache ("write back")
- On read:
  - Fetch value from owner or from memory

Cache state machine



Directory-based cache coherence

- How do we know which cores have a location cached?
  - Hardware keeps track of all cached copies
  - On a read miss, if held exclusive, fetch latest copy and invalidate that copy
  - On a write miss, invalidate all copies
- Read-modify-write instructions
  - Fetch cache entry exclusive, prevent any other cache from reading the data until instruction completes

A simple critical section

```
// A counter protected by a spinlock
Counter::Increment() {
   while (test_and_set(&lock))
      ;
   value++;
   lock = FREE;
   memory_barrier();
}
```

A simple test of cache Behavior

Array of 1K counters, each protected by a separate spinlock

- Array small enough to fit in cache

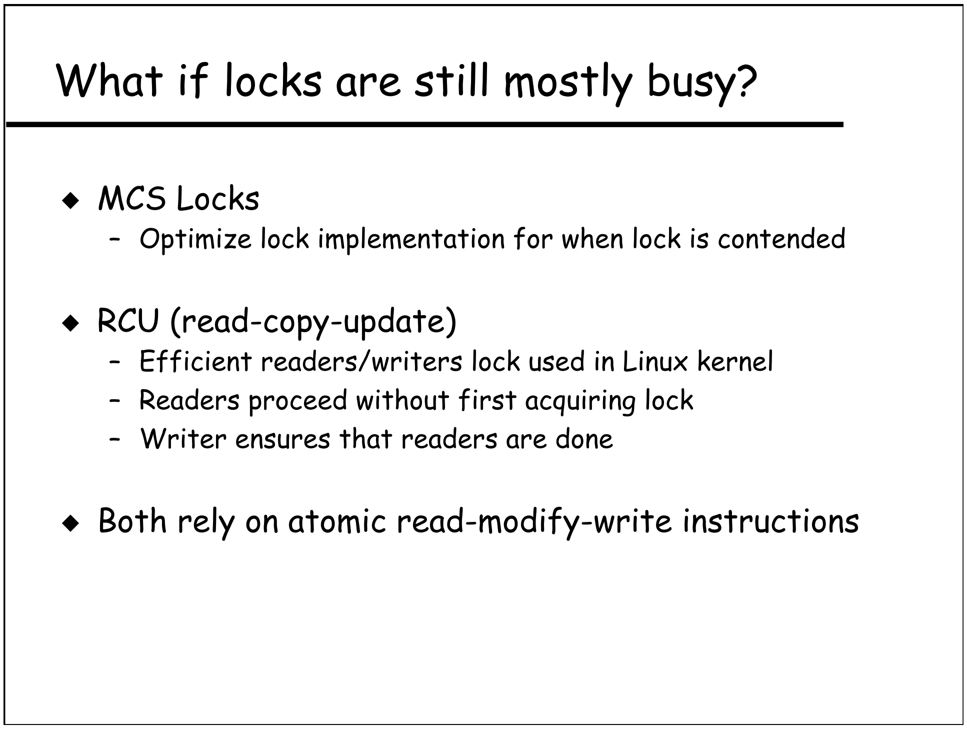
- Test 1: one thread loops over array
- Test 2: two threads loop over different arrays
- Test 3: two threads loop over single array
- Test 4: two threads loop over alternate elements in single array

Results (64 core AMD Opteron)

One thread, one array	51 cycles
Two threads, two arrays	52 cycles
Two threads, one array	197 cycles
Two threads, odd/even	127 cycles

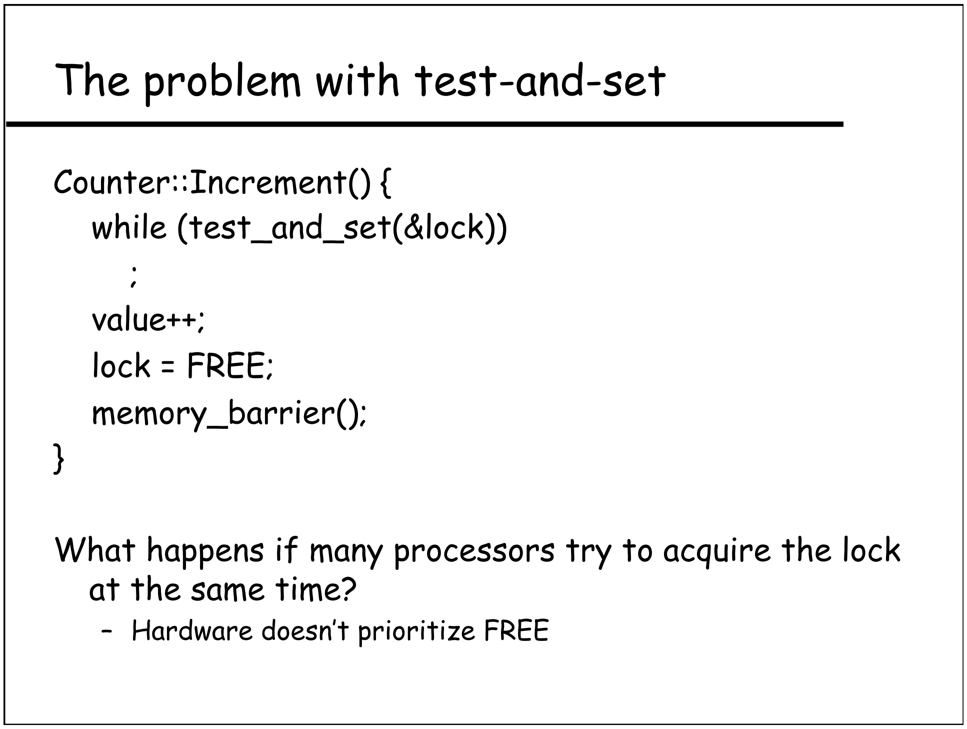
Reducing lock contention

- Fine-grained locking
  - Partition object into subsets, each protected by its own lock
  - Example: hash table buckets
- Per-processor data structures
  - Partition object so that most/all accesses are made by one processor
  - Example: per-processor heap
- Ownership/staged architecture
  - Only one thread at a time accesses shared data
  - Example: pipeline of threads



What if locks are still mostly busy?

- MCS Locks
  - Optimize lock implementation for when lock is contended
- RCU (read-copy-update)
  - Efficient readers/writers lock used in Linux kernel
  - Readers proceed without first acquiring lock
  - Writer ensures that readers are done
- Both rely on atomic read-modify-write instructions



The problem with test-and-set

```
Counter::Increment() {
   while (test_and_set(&lock))
      ;
   value++;
   lock = FREE;
   memory_barrier();
}
```

What happens if many processors try to acquire the lock at the same time?

- Hardware doesn't prioritize FREE

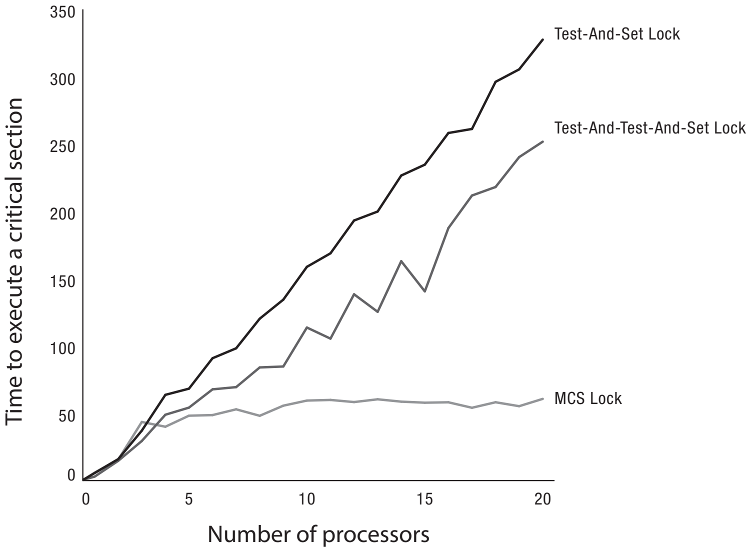
The problem with test-&-test-and-set

```
Counter::Increment() {
   while (lock == BUSY && test_and_set(&lock))
      ;
   value++;
   lock = FREE;
   memory_barrier();
}
```

What happens if many processors try to acquire the lock?

- Lock value pings between caches

Test (and test) and set performance

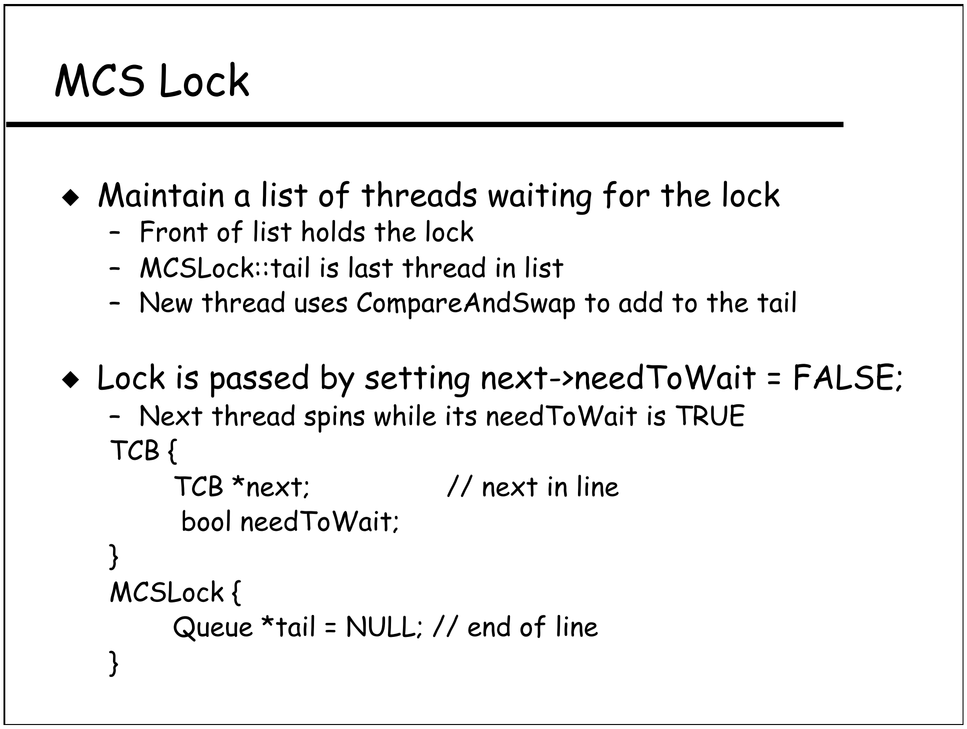


Some Approaches

- Insert a delay in the spin loop
  - Helps but acquire is slow when not much contention
- Spin adaptively
  - No delay if few waiting
  - Longer delay if many waiting
  - Guess number of waiters by how long you wait
- MCS
  - Create a linked list of waiters using compareAndSwap
  - Spin on a per-processor location

Atomic CompareAndSwap

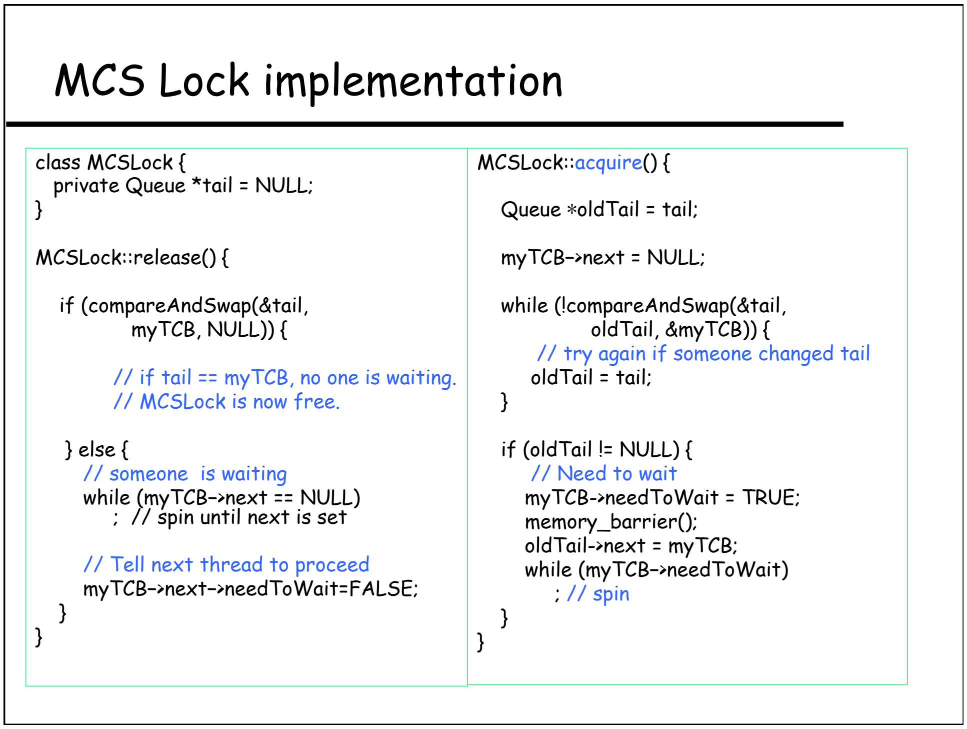
- Operates on a memory word
- Check that the value of the memory word hasn't changed from what you expect
  - E.g., no other thread did compareAndSwap first
- If it has changed, return an error (and loop)
- If it has not changed, set the memory word to a new value



MCS Lock

- Maintain a list of threads waiting for the lock
  - Front of list holds the lock
  - MCSLock::tail is last thread in list
  - New thread uses CompareAndSwap to add to the tail
- Lock is passed by setting next->needToWait = FALSE;
  - Next thread spins while its needToWait is TRUE

```
TCB {
        TCB *next;              // next in line
         bool needToWait;
}
MCSLock {
        Queue *tail = NULL; // end of line
}
```

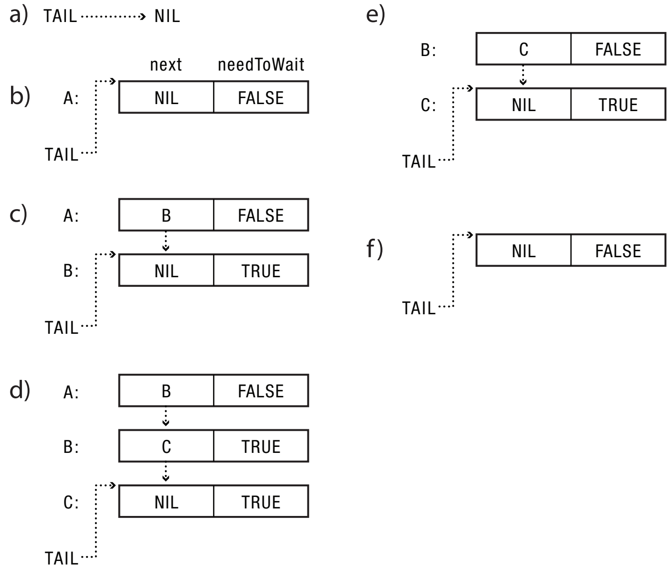


MCS Lock implementation

```
class MCSLock {
   private Queue *tail = NULL;
}

MCSLock::release() {

    if (compareAndSwap(&tail,
                myTCB, NULL)) {

            // if tail == myTCB, no one is waiting.
            // MCSLock is now free.

     } else {
        // someone  is waiting
        while (myTCB->next == NULL)
             ;  // spin until next is set

        // Tell next thread to proceed
        myTCB->next->needToWait=FALSE;
    }
}
```

```
MCSLock::acquire() {

    Queue *oldTail = tail;

    myTCB->next = NULL;

    while (!compareAndSwap(&tail,
                oldTail, &myTCB)) {
        // try again if someone changed tail
        oldTail = tail;
    }

    if (oldTail != NULL) {
        // Need to wait
        myTCB->needToWait = TRUE;
        memory_barrier();
        oldTail->next = myTCB;
        while (myTCB->needToWait)
             ; // spin
    }
}
```

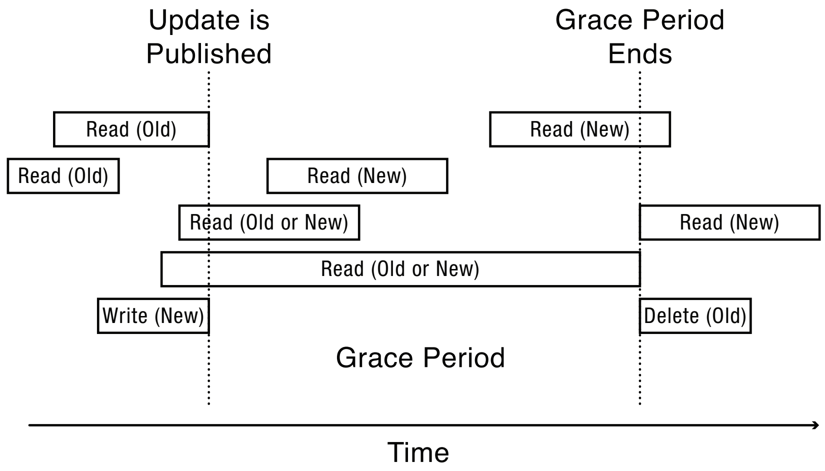
MCSLock in operation



Read-Copy-Update

- Goal: very fast reads to shared data
  - Reads proceed without first acquiring a lock
  - OK if write is (very) slow
- Restricted update
  - Writer computes new version of data structure
  - Publishes new version with a single atomic instruction
- Multiple concurrent versions
  - Readers may see old or new version
- Integration with thread scheduler
  - Guarantee all readers complete within grace period, and then garbage collect old version

Read-Copy-Update



Read-Copy-Update implementation

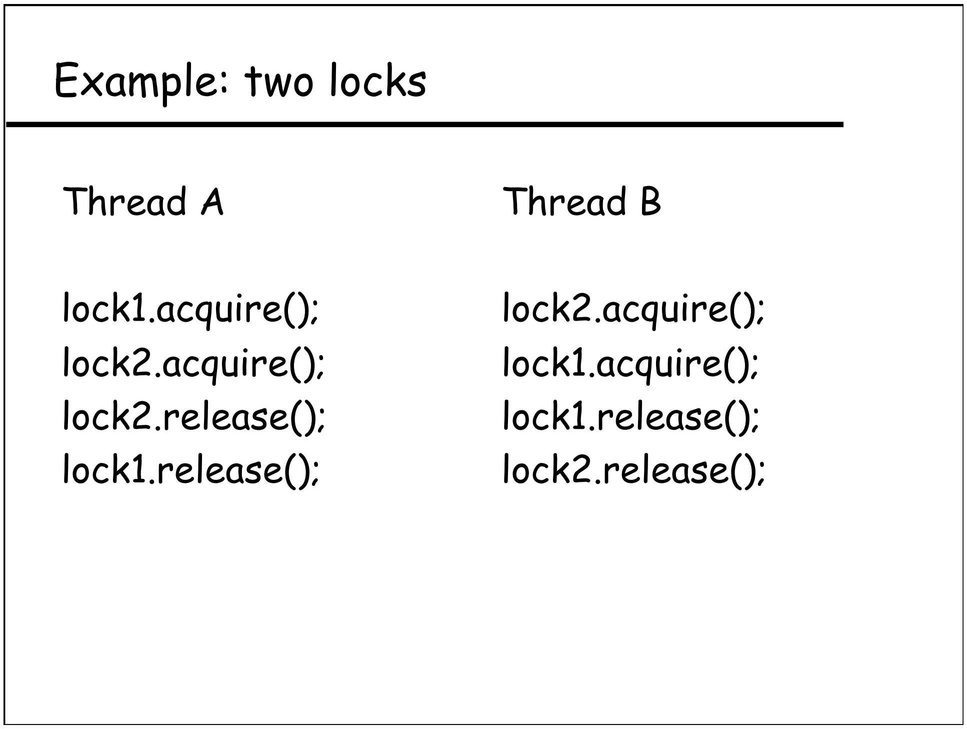
- Readers disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - No read or write lock
- Writer
  - Acquire write lock
  - Compute new data structure
  - Publish new version with atomic instruction
  - Release write lock
  - Wait for time slice on each CPU
  - Only then, garbage collect old version of data structure

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

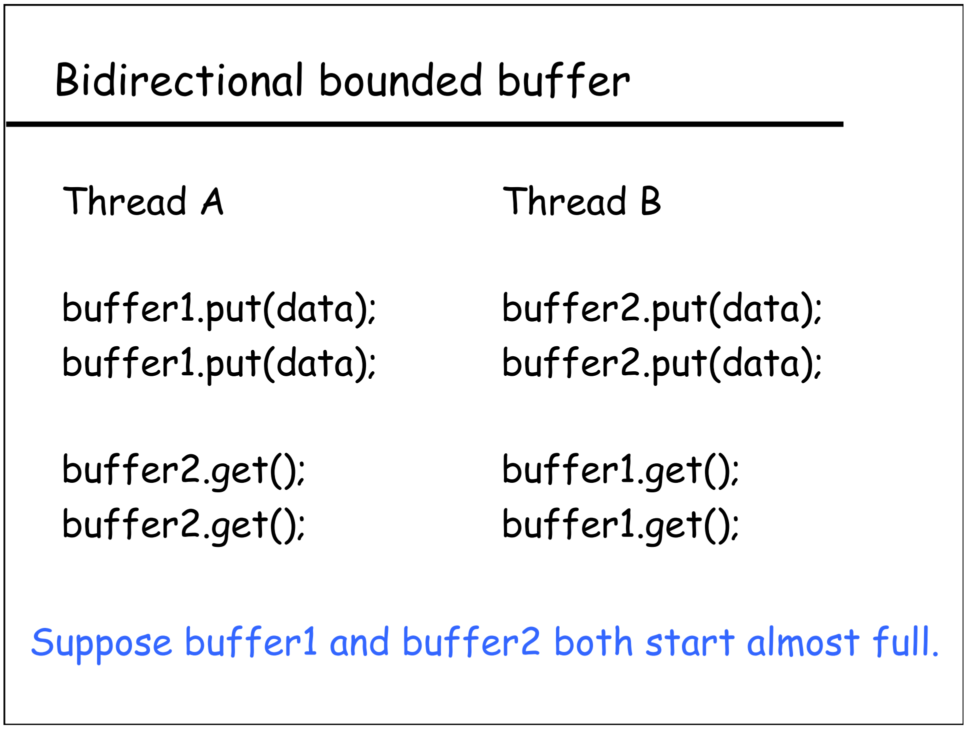
Deadlock definition

- 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

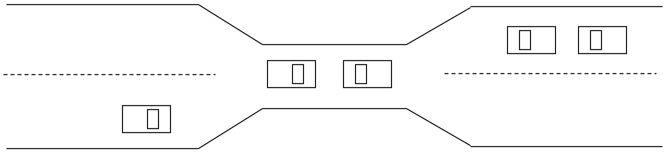
```
Thread A               Thread B

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                         Thread B

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

The bridge-crossing example

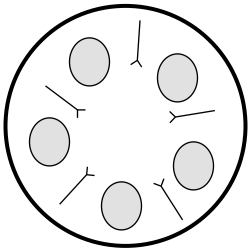


- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

The dining philosophers problem

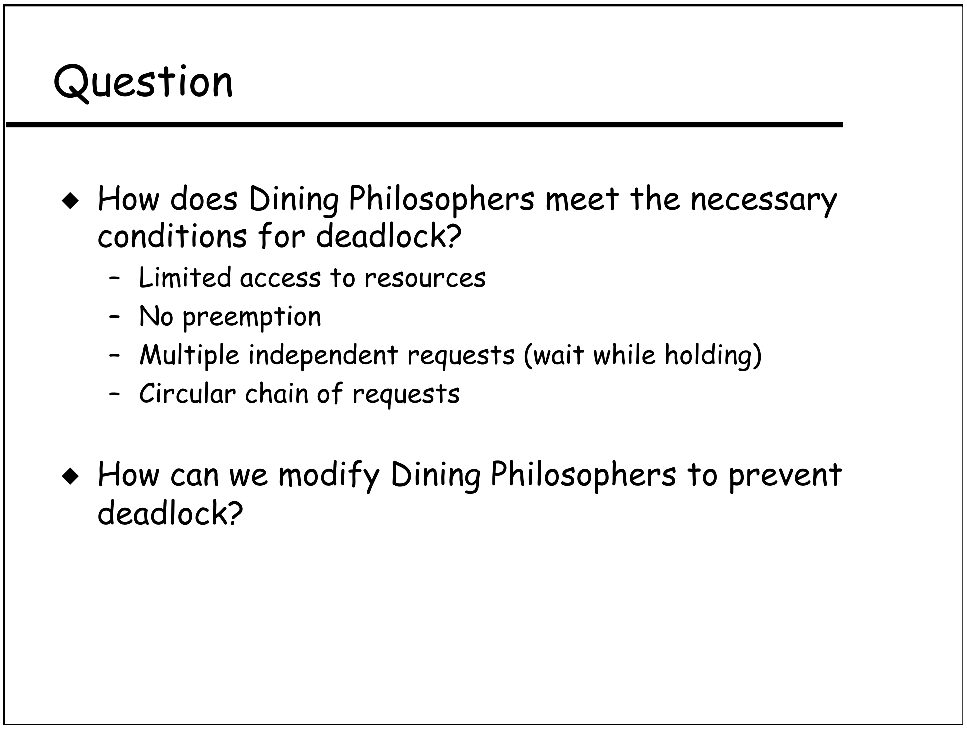
- Five philosophers around a table --- thinking or eating
- Five plates of spaghetti + five forks (placed between each plate)
- The spaghetti is so slippery that a philosopher needs two forks to eat it.

```
void philosopher (int i) {
   while (TRUE) {
       think();
       take_fork (i);
       take_fork ((i+1) % 5);
       eat();
       put_fork (i);
       put_fork ((i+1) % 5);
   }
}
```



Necessary conditions for deadlock

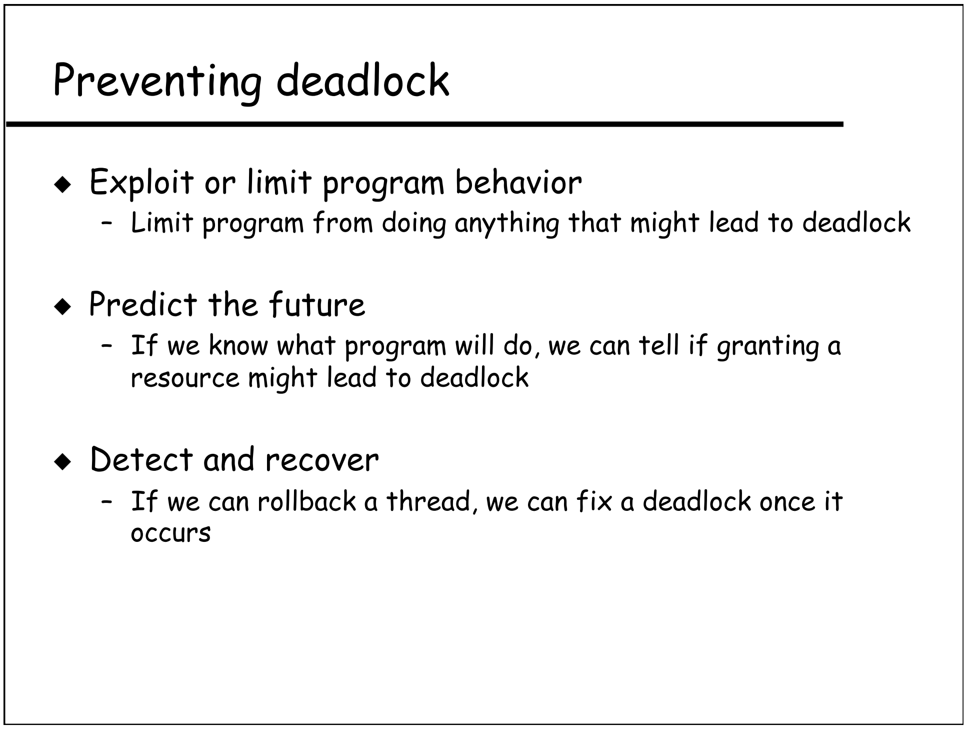
- Limited access to resources
  - If infinite resources, no deadlock!
- No preemption
  - If resources are virtual, can break deadlock
- Multiple independent requests
  - “wait while holding”
- Circular chain of requests



Question

- How does Dining Philosophers 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 Philosophers 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

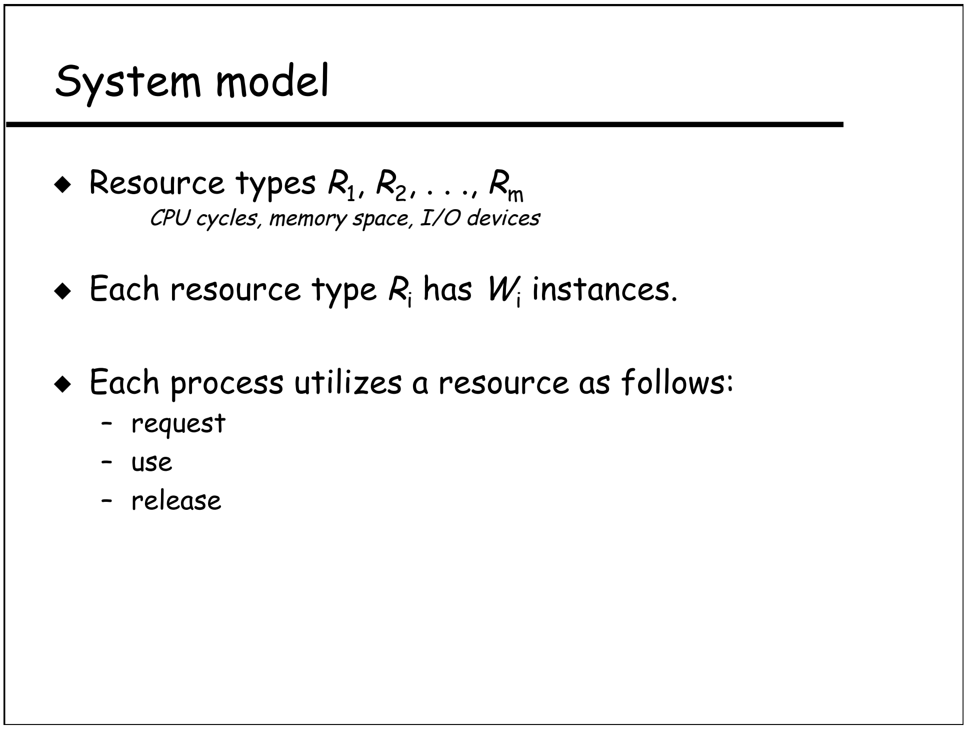
Exploit or limit behavior

- 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

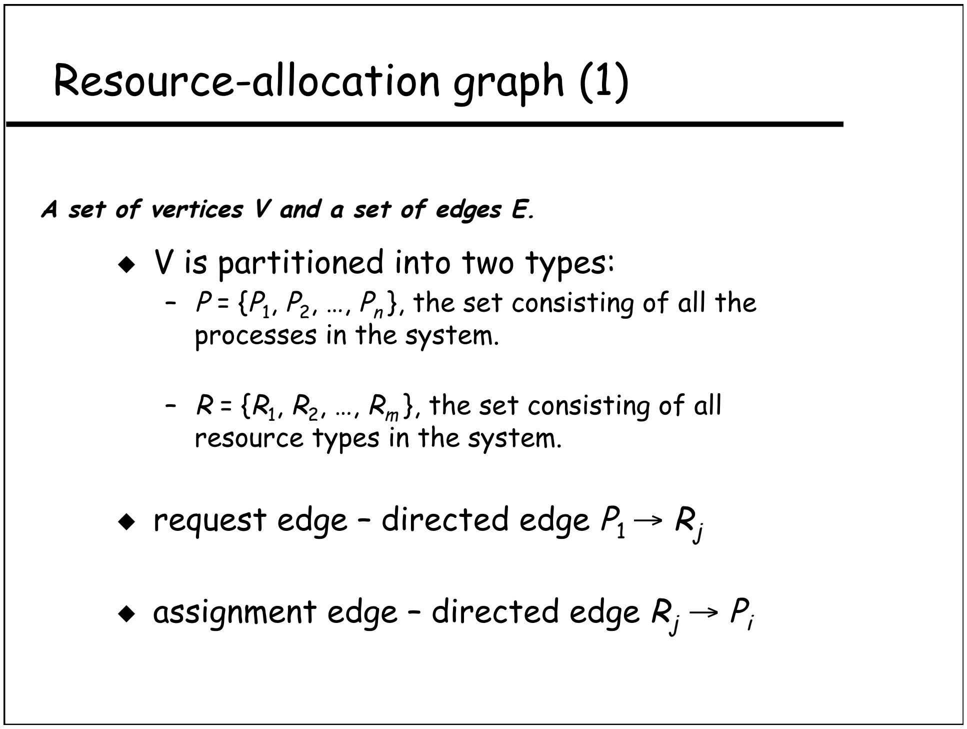
Thread 1	Thread 2
1. Acquire A	1.
2.	2. Acquire B
3. Acquire C	3.
4.	4. Wait for A
5. If (maybe) Wait for B	

How can we make sure to avoid deadlock?



System model

- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

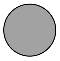


Resource-allocation graph (1)

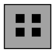
A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

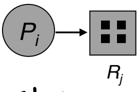
Resource-allocation graph (2)

- Process 

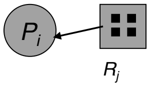
- Resource type with 4 instances



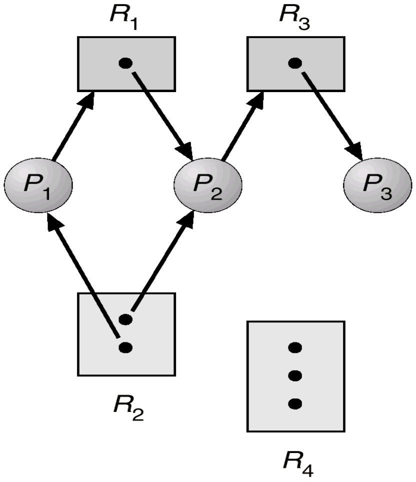
- P_i requests instance of R_j



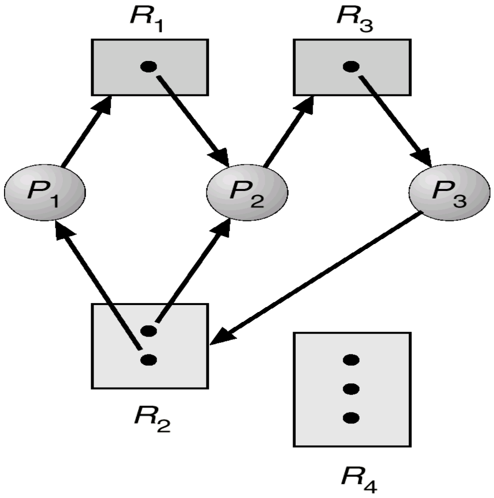
- P_i is holding an instance of R_j



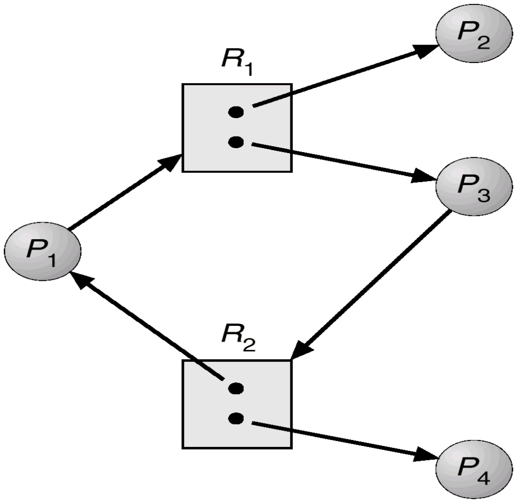
Example: resource-allocation graph



Resource-allocation graph with a deadlock



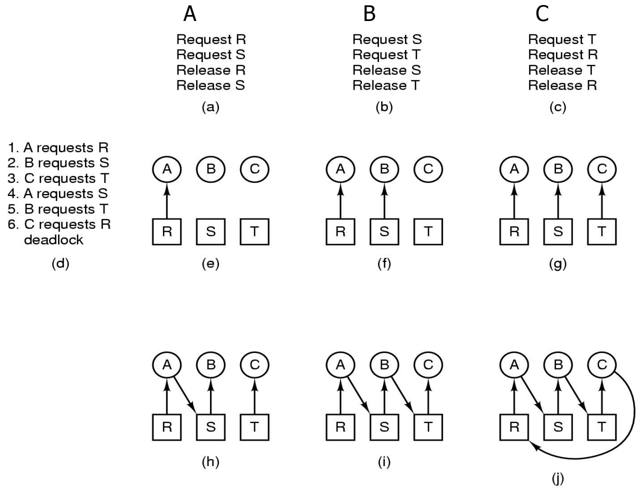
Resource-allocation graph with a cycle but no deadlock



Resource allocation graph vs. deadlock?

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

How deadlocks occur?



A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

C

Request T
Request R
Release T
Release R

(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)

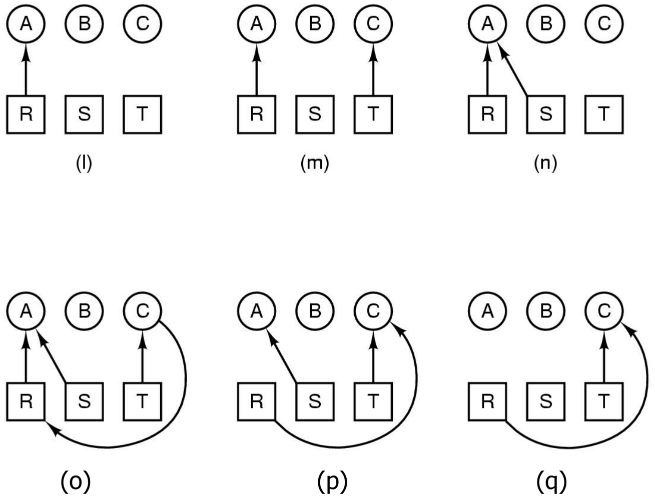
How deadlocks can be avoided

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

(k)

Block process B when it asks for S.



Deadlock detection: data structures

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Data structures needed by deadlock detection algorithm

Deadlock detection: example

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array} \qquad A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

An example for the deadlock detection algorithm

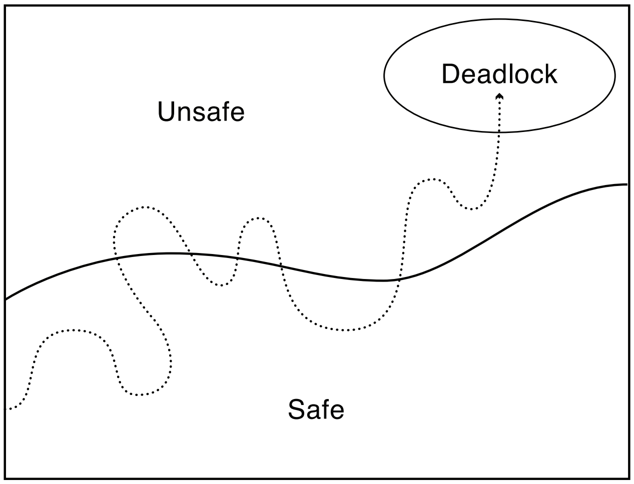
Methods for handling deadlocks

- Ensure that the system will *never* enter a deadlock state. ***(deadlock prevention and avoidance)***
  - * problems: low device utilization, reduced throughput
  - * avoidance also requires prediction of resource needs
- Allow the system to enter a deadlock state and then recover. ***(deadlock detection and recovery)***
  - * costly; sometimes impossible to recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

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



Safe and unsafe states

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Demonstration that the state in **(a)** is safe

Safe and unsafe states

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

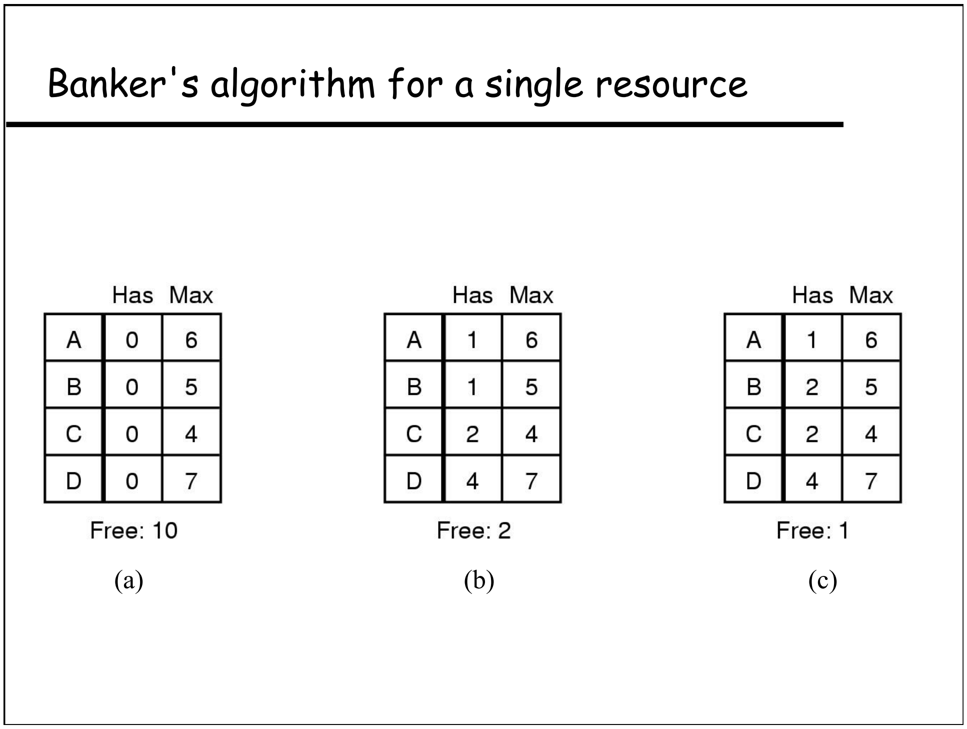
Demonstration that the state in **(b)** is not safe

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



Banker's algorithm for a single resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

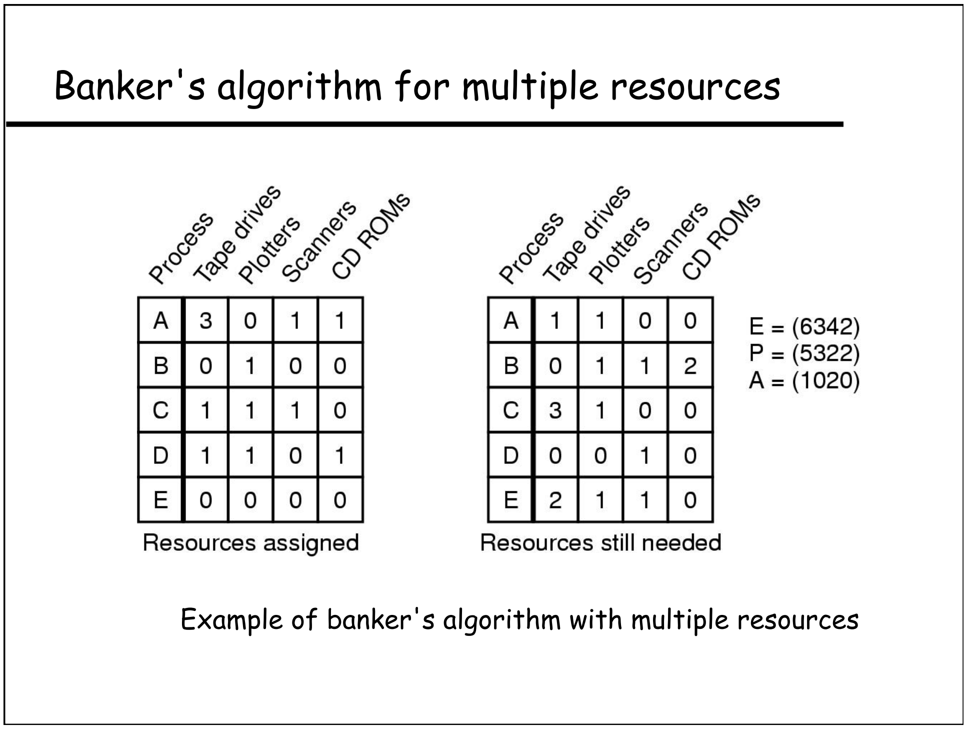
Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)



Banker's algorithm for multiple resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Example of banker's algorithm with multiple resources

Banker's algorithm: data structures

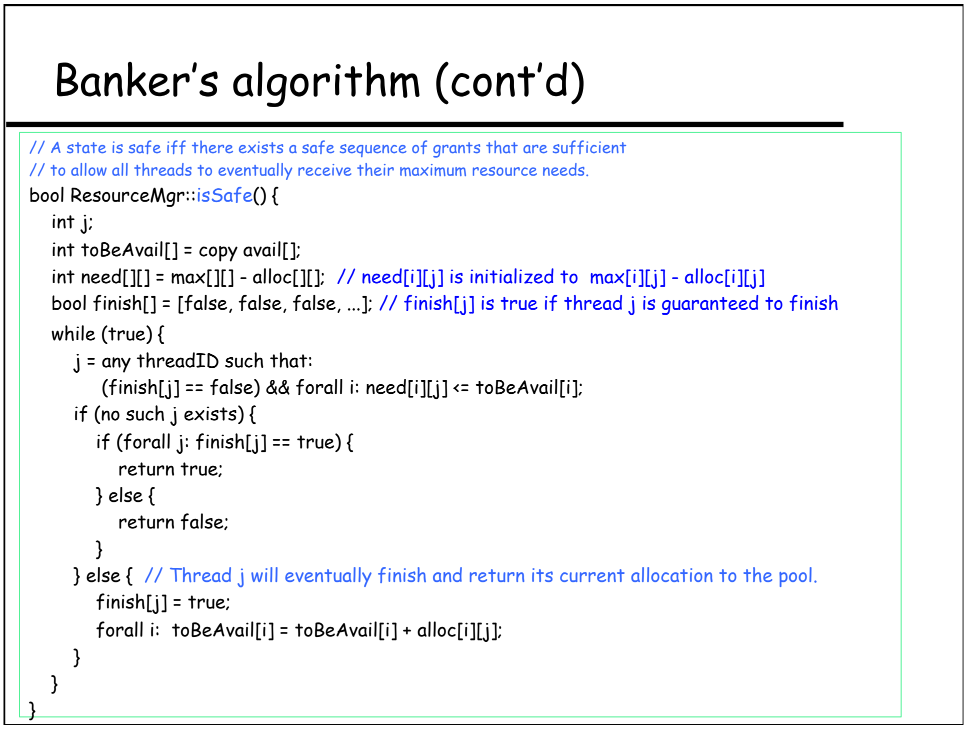
Let n = number of processes, and m = number of resources types.

- ***Available:*** Vector of length m. If avail [j] = k, there are k instances of resource type R_j available.
- ***Max:*** n x m matrix. If *max* [i,j] = k, then process P_j may request at most k instances of resource type R_i
- ***Allocation:*** n x m matrix. If alloc[i,j] = k then P_j is currently allocated k instances of R_i
- ***Need:*** n x m matrix. If *Need*[i,j] = k, then P_j may need k more instances of R_i to complete its task.

Need [i,j] = *Max*[i,j] – *Allocation* [i,j].

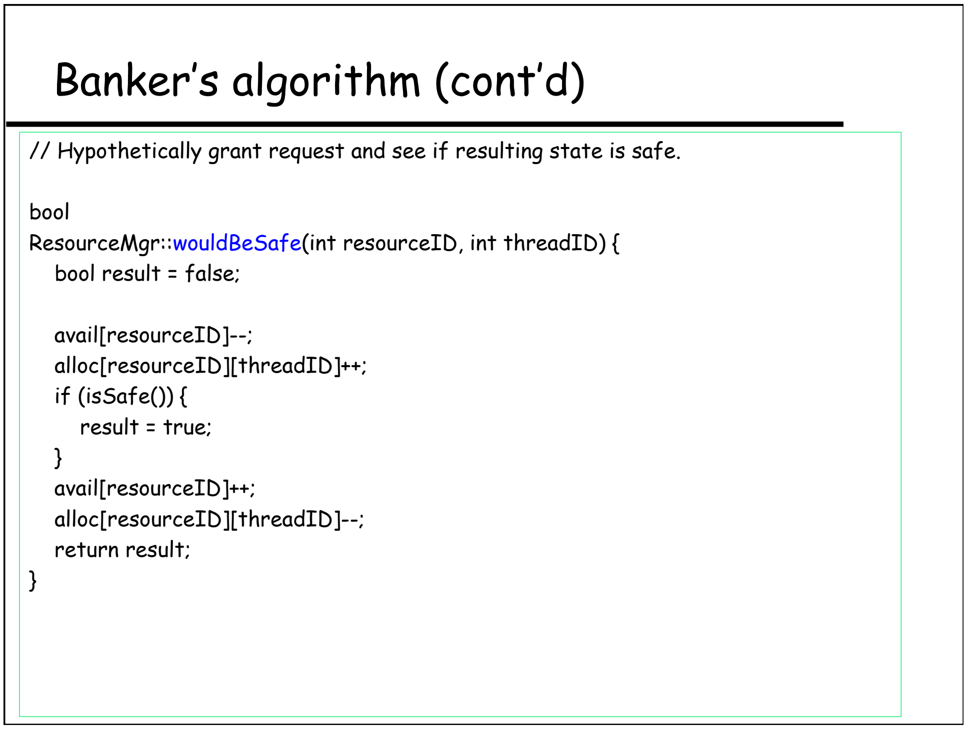
Banker's algorithm

```
class ResourceMgr {
  private:
    Lock lock;
    CV cv;
    int r;          // Number of resources
    int t;          // Number of threads
    int avail[];    // avail[i]: instances of resource i available
    int max[][];    // max[i][j]: max of resource i needed by thread j
    int alloc[][];  // alloc[i][j]: current allocation of resource i to thread j
  ...
}

// Invariant: the system is in a safe state.
ResourceMgr::Request(int resourceID, int threadID) {
    lock.Acquire();
    assert(isSafe());
    while (!wouldBeSafe(resourceID, threadID)) {
        cv.Wait(&lock);
    }
    alloc[resourceID][threadID]++;
    avail[resourceID]--;
    assert(isSafe());
    lock.Release();
}
```



Banker's algorithm (cont'd)

```
// A state is safe iff there exists a safe sequence of grants that are sufficient
// to allow all threads to eventually receive their maximum resource needs.
bool ResourceMgr::isSafe() {
   int j;
   int toBeAvail[] = copy avail[];
   int need[][] = max[][] - alloc[][];  // need[i][j] is initialized to  max[i][j] - alloc[i][j]
   bool finish[] = [false, false, false, ...]; // finish[j] is true if thread j is guaranteed to finish
   while (true) {
      j = any threadID such that:
          (finish[j] == false) && forall i: need[i][j] <= toBeAvail[i];
      if (no such j exists) {
         if (forall j: finish[j] == true) {
            return true;
         } else {
            return false;
         }
      } else {  // Thread j will eventually finish and return its current allocation to the pool.
         finish[j] = true;
         forall i:  toBeAvail[i] = toBeAvail[i] + alloc[i][j];
      }
   }
}
```



Banker's algorithm (cont'd)

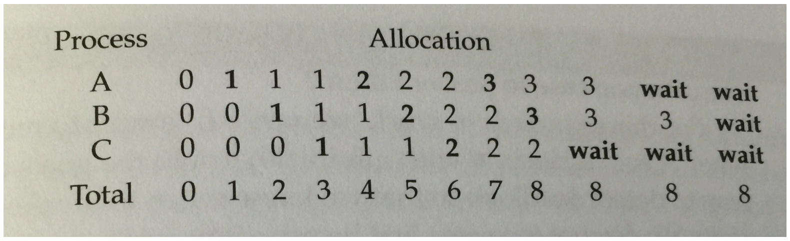
```
// Hypothetically grant request and see if resulting state is safe.

bool
ResourceMgr::wouldBeSafe(int resourceID, int threadID) {
   bool result = false;

   avail[resourceID]--;
   alloc[resourceID][threadID]++;
   if (isSafe()) {
      result = true;
   }
   avail[resourceID]++;
   alloc[resourceID][threadID]--;
   return result;
}
```

Why we need Banker's algorithm?

8 pages of memory available

Three processes: A, B, C which need 4, 5, 5 pages respectively

The following would leads to deadlock



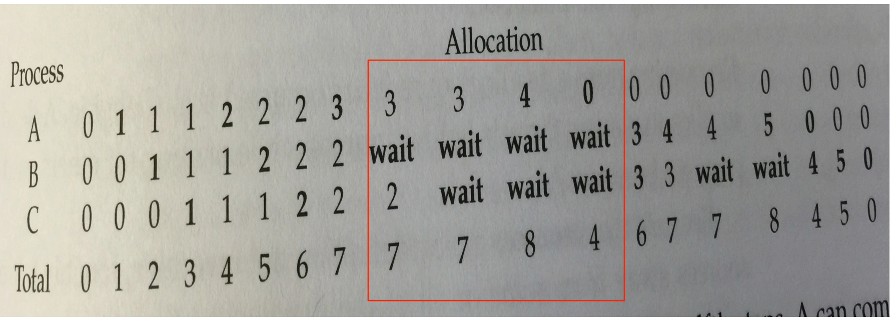
Process	Allocation											
A	0	1	1	1	2	2	2	3	3	3	wait	wait
B	0	0	1	1	1	2	2	2	3	3	3	wait
C	0	0	0	1	1	1	2	2	2	wait	wait	wait
Total	0	1	2	3	4	5	6	7	8	8	8	8

Why we need Banker's algorithm?

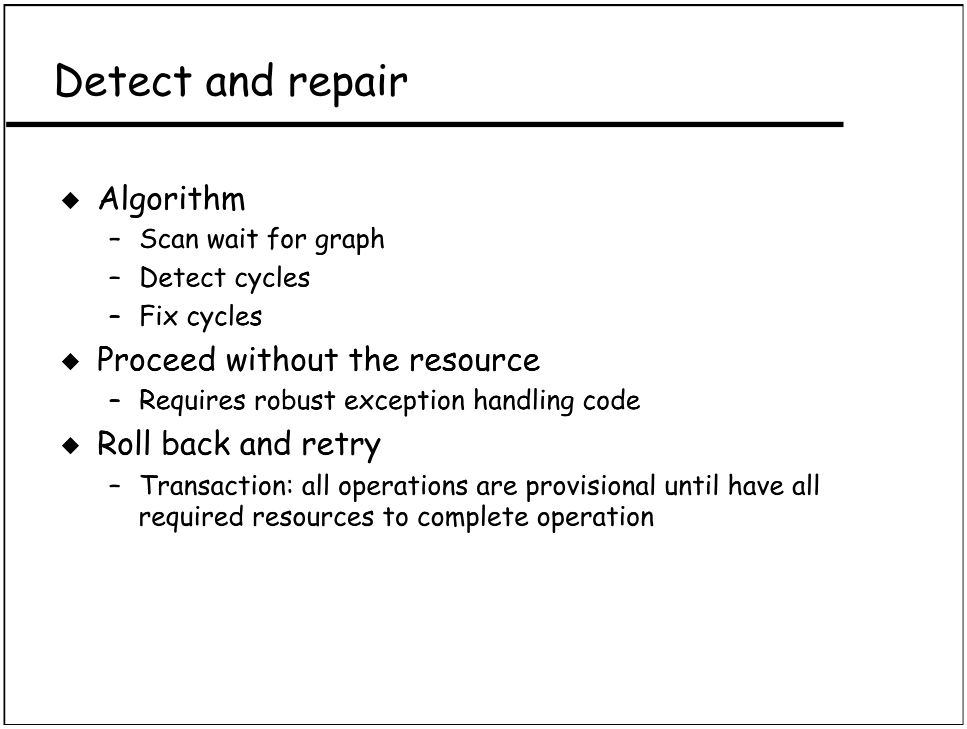
8 pages of memory available

Three processes: A, B, C which need 4, 5, 5 pages respectively

The following would work!



Process	Allocation																		
A	0	1	1	1	2	2	2	3	3	3	4	0	0	0	0	0	0	0	0
B	0	0	1	1	1	2	2	2	wait	wait	wait	wait	3	4	4	5	0	0	0
C	0	0	0	1	1	1	2	2	2	wait	wait	wait	3	3	wait	wait	4	5	0
Total	0	1	2	3	4	5	6	7	7	7	8	4	6	7	7	8	4	5	0



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