Multi-Object Synchronization

*Throughout the course we will use overheads that were adapted from those distributed from the textbook website. Slides are from the book authors, modified and selected by Jean Mayo, Shuai Wang and C-K Shene.

We must know. We will know.

David Hilbert

Spring 2019

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

Several Considerations

- **Multiprocessor Performance**: Modern computers have increasing numbers of processors. The design of shared objects can have a large impact on multiprocessor performance.
- **Correctness**: For programs with multiple shared objects, we face a problem similar to the once faced when reasoning about atomic loads and stores.
- **Deadlock**: Multiple objects may hold multiple locks. As a result, deadlock could occur.

Multiprocessor Lock Performance

- **Locking**: A lock implies mutual exclusion. Hence, access to a shared object can limit parallelism.
- **Communication of Shared Data**: Performance of a modern processor can vary significantly depending on whether the data needed by the processor is already in its cache or not. On a multiprocessor, shared data protected by a lock will often need to be copied from one cache to another.
- **False Sharing**: The hardware keeps track of shared data at a fixed granularity, often in units of a cache of 32 or 64 bytes. This can cause performance problems if multiple data structures with different sharing behavior fit in the same cache memory.

A Simple Test of Cache Behavior

- Array of 1K counters (small enough to fit in cache), each protected by a separate spinlock.
- The program iterates through the array.
- For each item, it acquires the lock, increments the counter, and releases the lock.
- This loop repeats 1000 times to improve measurement precision.
- Tests:
  - Test 1: one thread loops over array
  - Test 2: two threads loop over disjoint 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)

Scenario	Number of CPU Cycles
One thread, one array	51.2
Two threads, two arrays	52.5
Two threads, one array	197.4
Two threads, alternate elements of one array	127.3

Number of CPU cycles to execute a simple critical section to increment a counter.
Threads assigned to processor cores that do not share a cache.
Performance difference between these cases largely disappears when threads are assigned to cores that share an L2 cache.

Lock Contention

- Locking may remain a bottleneck to good performance on a multiprocessor. Locking a popular item can be a source of contention.
- **MCS locks** (if locks are mostly busy): **MCS** is the initial of the authors of the original paper, John M. **M**ellor-**C**rummey and Michael L. **S**cott*. MCS is an implementation of a spinlock optimized for the case when there are a significant number of waiting threads.
- **RCU locks** (if locks are mostly busy, and data is mostly read-only): **RCU** = **R**ead-**C**opy-**U**pdate. RCU reduces the overhead for read-only at a cost of increased for non-read-only.

*John M. Mellor-Crummey and Michael L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Transactions on Computer Systems*, Volume 9 (1991), No. 1, pp. 21-65.

Problem with Test-and-Set

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

- What happens if many processors try to acquire the lock?
  - Counter value must be communicated from one lock holder to the next.
  - The critical section will take significantly longer on a multiprocessor than on a single processor.

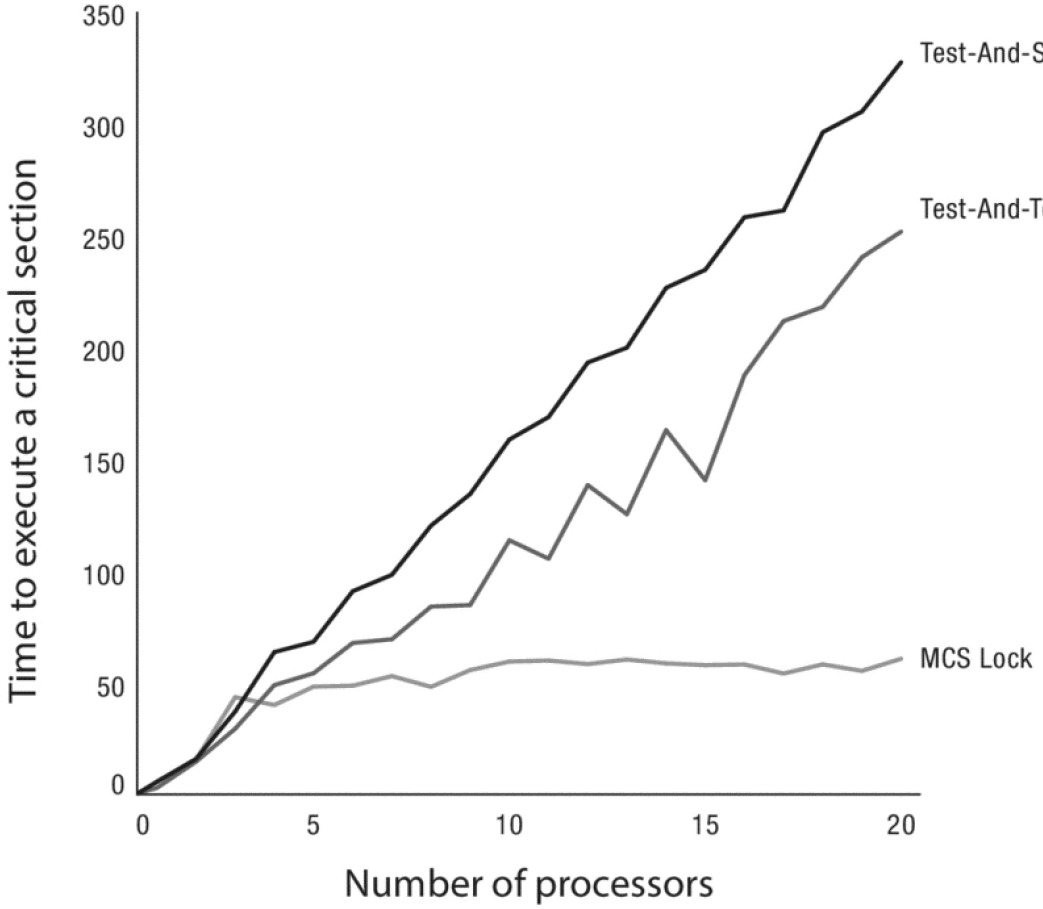
Problem with Test and Test-and-Set

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

How about check the availability of the lock before actually starting to spin?

This way, the chance to start spinning is lower.

Test-and-Test-and-Set Performance



Measurements taken on a 64-core AMD Opteron 6262.
The non-smooth curves are typical of measurements of real system.

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

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` earlier since the last inspection.
- If it has changed, return an error (and loop)
- If it has not changed, set the memory word to a new value

```
CompareAndSwap(*p, old, new)
{
    if (*p != old) {   // if changed
        return FALSE;  // not the same
    }
    *p = new; // if not changed, update
    return TRUE;       // no change
}
```

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

```
MCSLock::acquire() {
  Queue *oldTail = tail;

  myTCB->next = NULL;
  myTCB->needToWait = TRUE;
  while (!CompareAndSwap(&tail,
            oldTail, &myTCB)) {
    oldTail = tail;
  }
  if (oldTail != NULL) {
    oldTail->next = myTCB;
    memory_barrier();
    while (myTCB->needToWait)
      ;
  }
}
```

```
MCSLock::release() {
    if (!CompareAndSwap(&tail,
          myTCB, NULL)) {
       while (myTCB->next == NULL)
          ;
       myTCB->next->needToWait=FALSE;
    }
}
```

```
// Compare-and-Swap is atomic
CompareAndSwap(*p, old, new)
{
    if (*p != old) {
       return FALSE
    }
    *p = new;
    return TRUE;
}
```

try again if someone else changed tail in the meantime

if `tail== myTCB`, no one is waiting. MCS lock is now free.

if `oldTail == NULL`, lock acquired

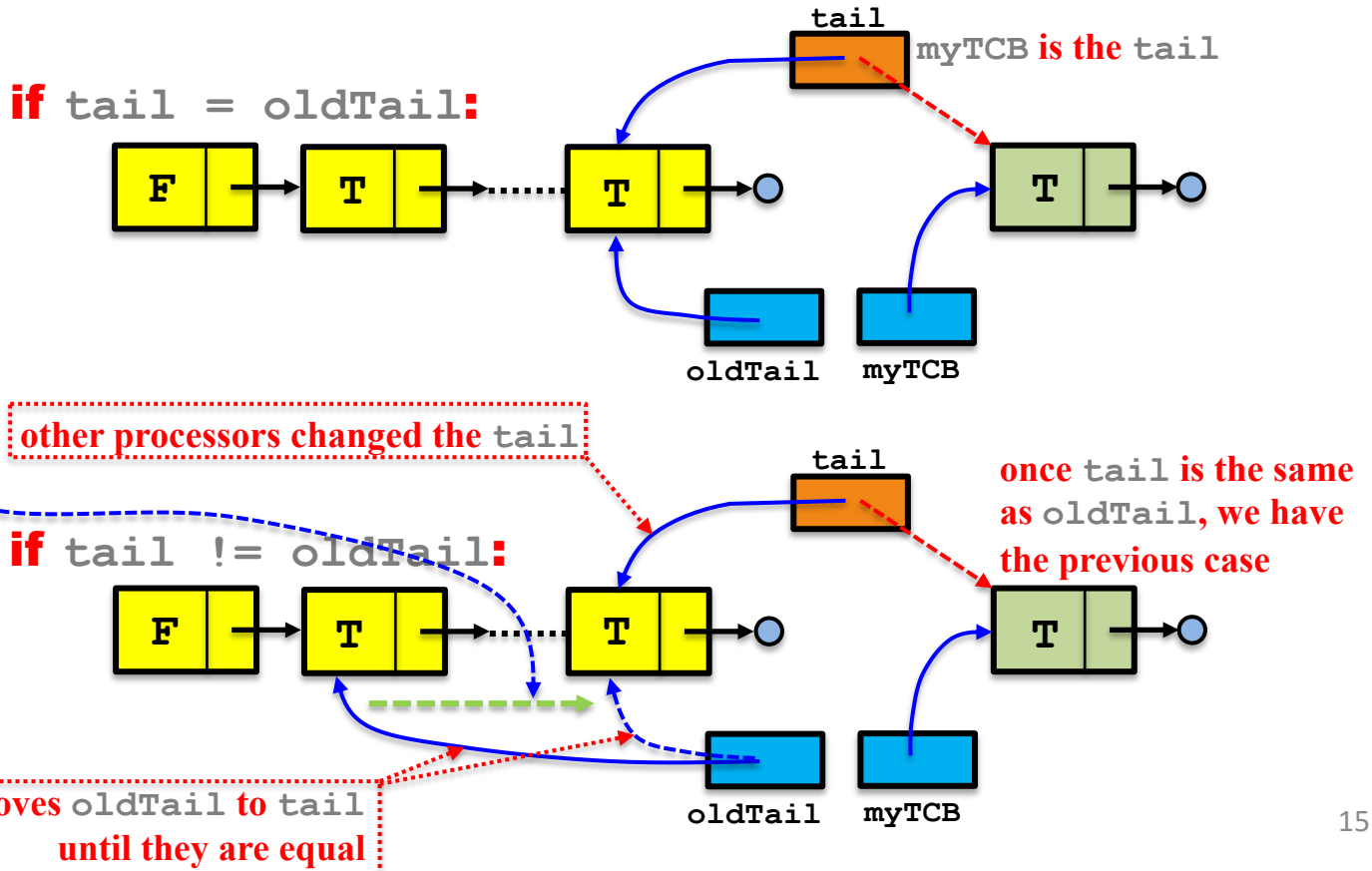
someone is waiting, spin

need to wait and spin

MCS Lock Implementation

MCSLock::acquire()
first part: add to tail

```
myTCB->next = NULL;          // initially no next &
myTCB->needToWait = TRUE;    //    need to wait
while (!CompareAndSwap(&tail, oldTail, &myTCB)) {
    oldTail = tail;
}
```

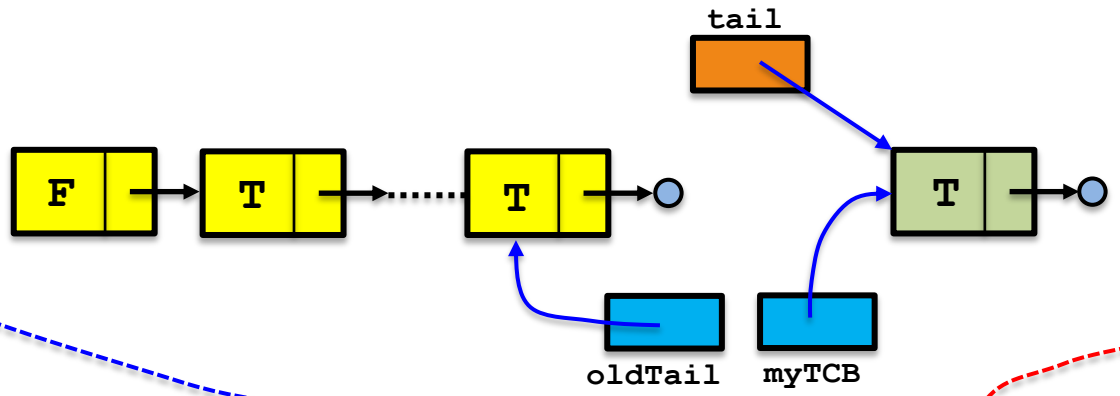


MCS Lock Implementation

MCSLock::acquire()

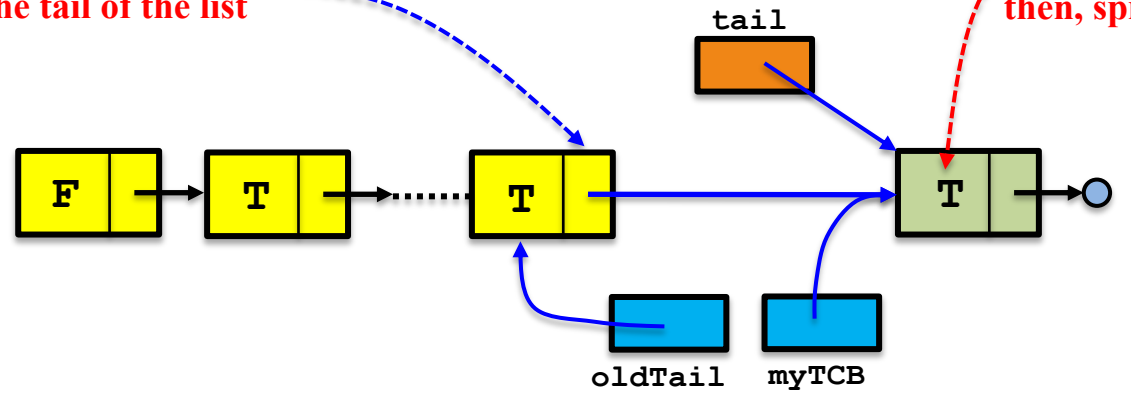
second part: complete the list

```
if (oldTail != NULL) {
    oldTail->next = myTCB;
    memory_barrier();
    while (myTCB->needToWait)
        ;
}
```



link myTCB to the tail of the list

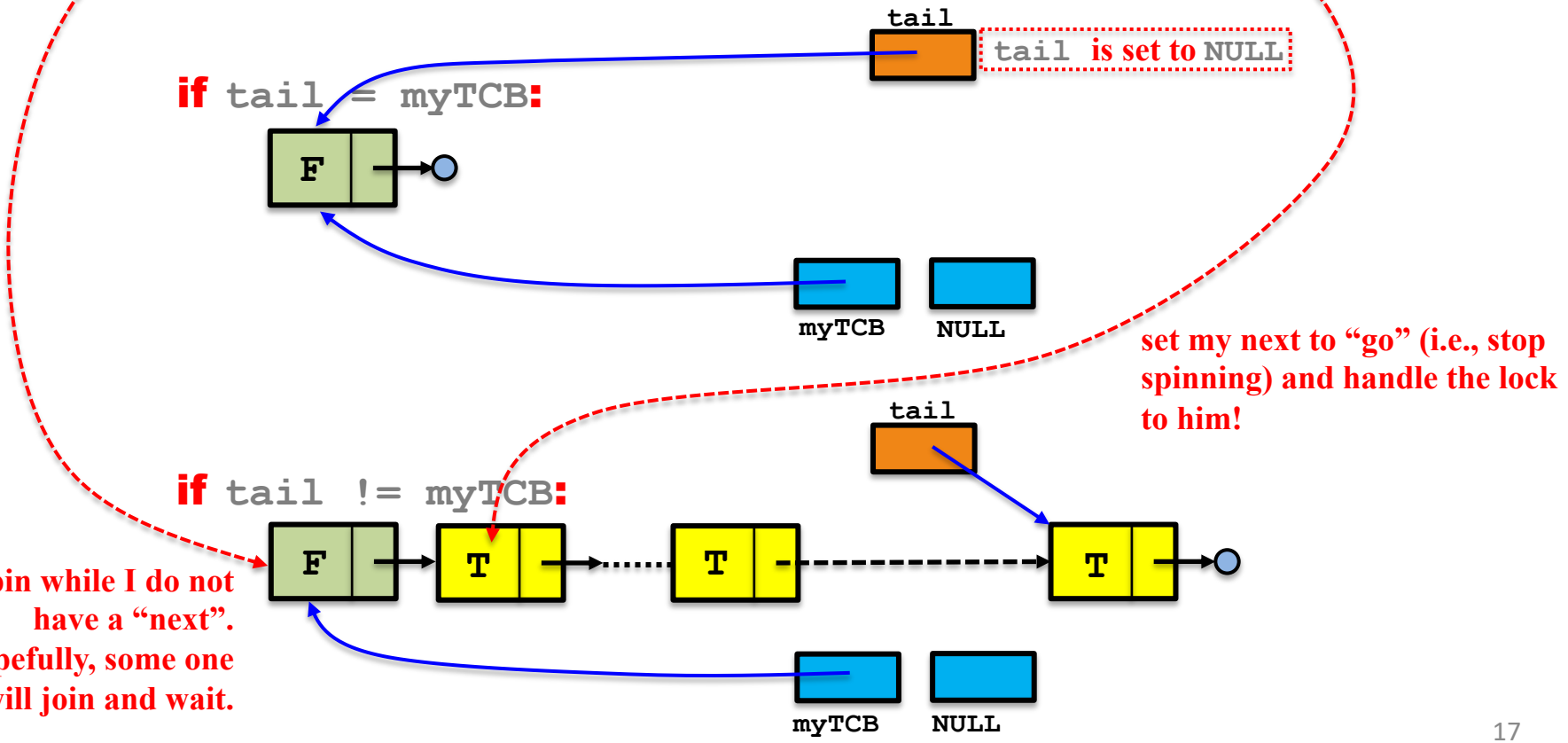
then, spin!



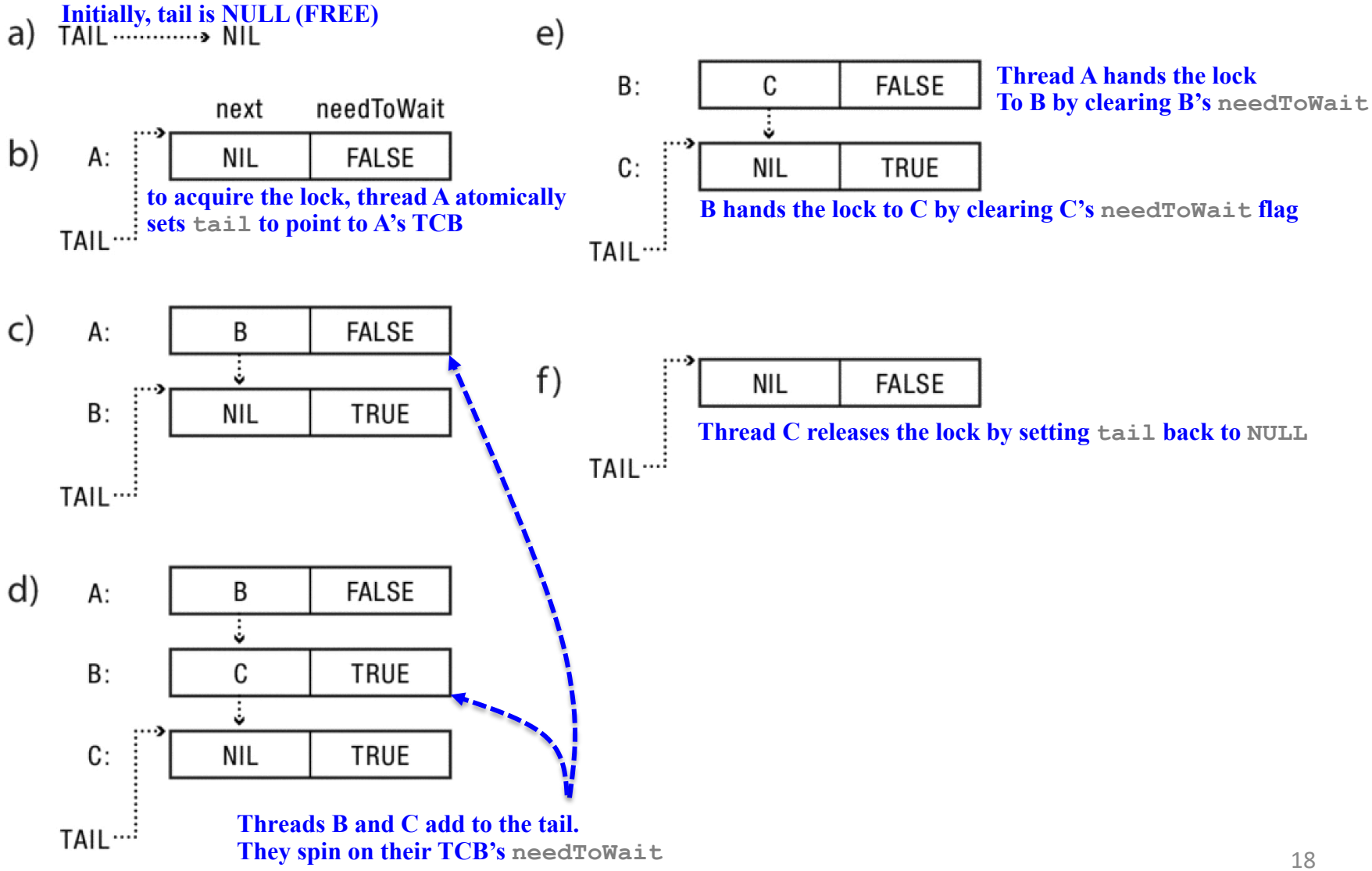
MCS Lock Implementation

MCSLock::release()

```
if (!CompareAndSwap(&tail, myTCB, NULL)) {
    while (myTCB->next == NULL)
        ;
    myTCB->next->needToWait=FALSE;
}
```



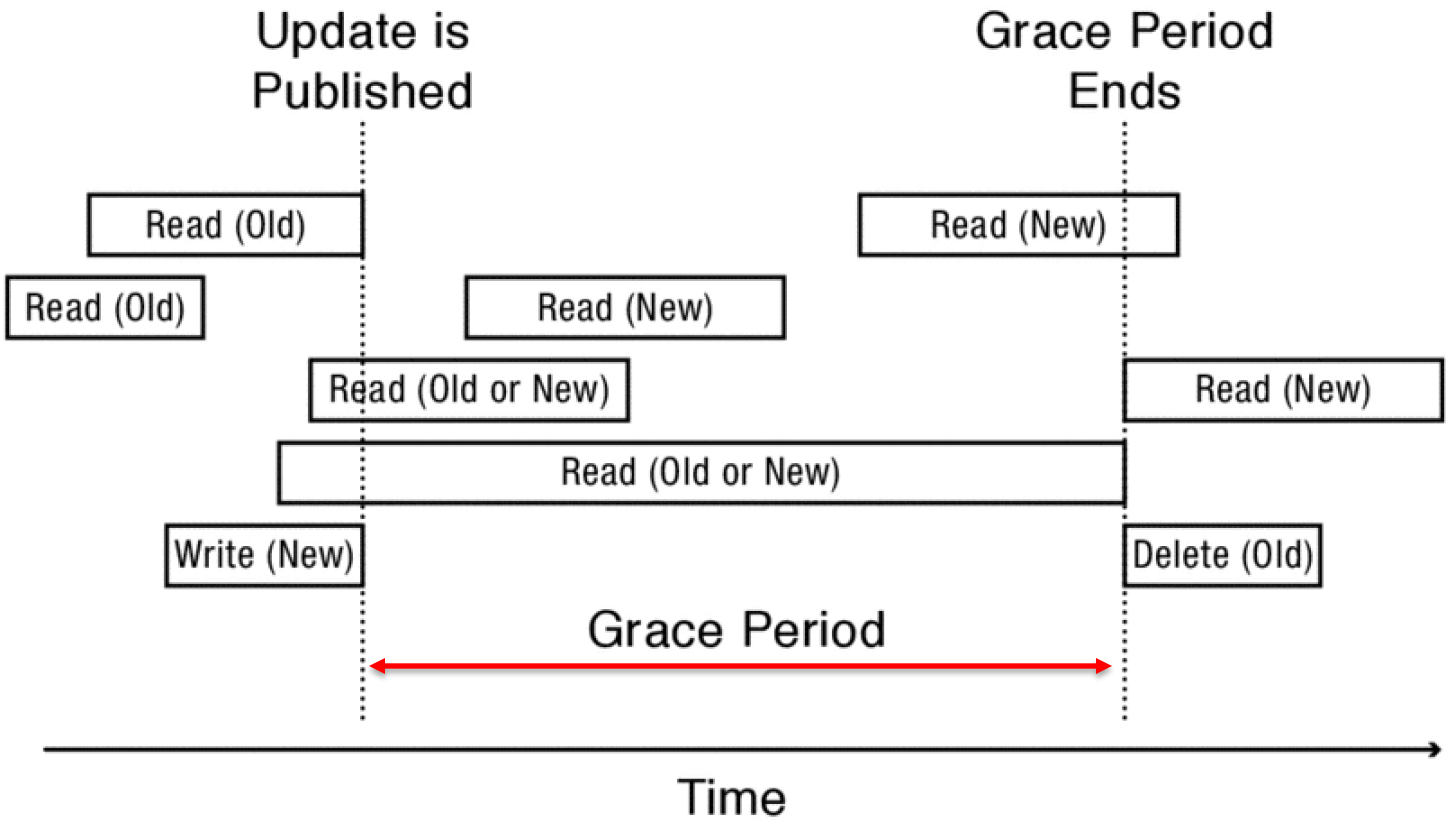
MCS In Operation



Read-Copy-Update: 1/2

- 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
  - Because there may be readers still in progress when an update is made, the shared object must maintain multiple versions of its state to guarantee that an old version is not freed until all readers have finished accessing it.
  - The time from when an update is published until the last reader is done with the previous version is called the *grace period*.
  - The RCU lock uses information provided by the thread scheduler to determine when a grace period ends.

Read-Copy-Update: 2/2



Timeline for an update concurrent with several reads for a data structure accessed by new readers

RCU Lock Implementation: 1/7

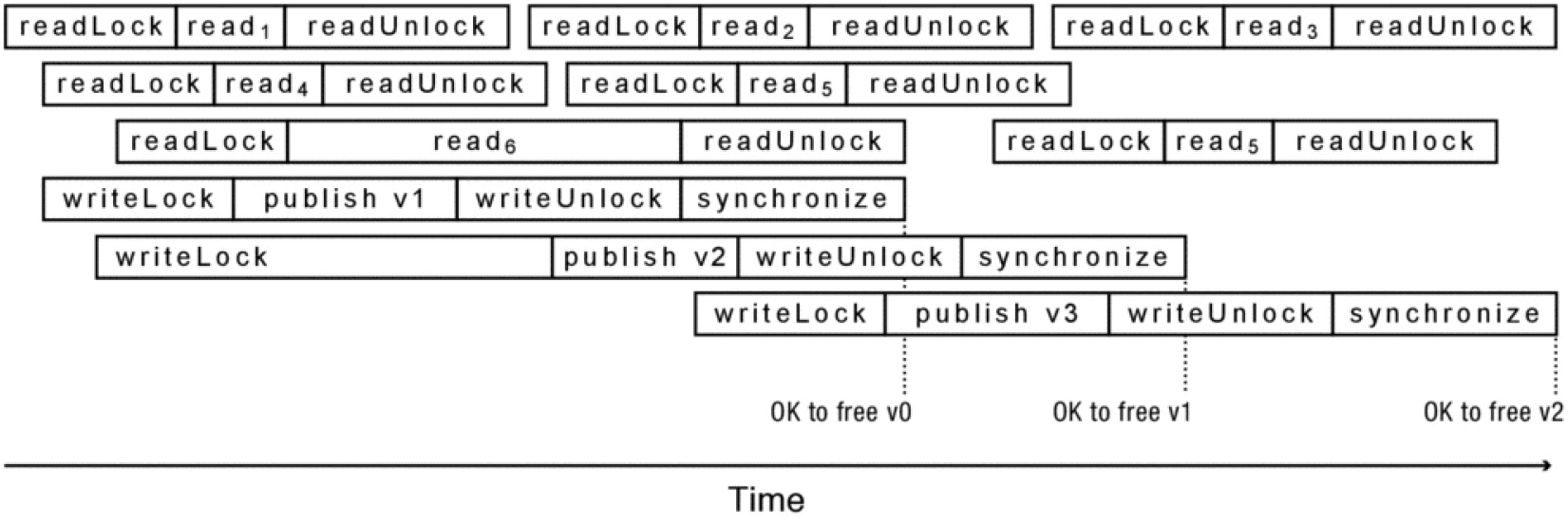
- Readers disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - No read or write lock
- Writer
  - Acquire write lock
  - Compute new data structure
  - Publish new version with atomic instruction
  - Release write lock
  - Wait for time slice on each CPU
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

RCU Clock Implementation: 2/7

- **Here is what we need:**
  - `readLock()` – **enter read-only critical section**
  - `readUnlock()` – **exit read-only critical section**
  - `writeLock()` – **enter read-write critical section**
  - `writeUnlock()` – **exit read-write critical section**
  - `publish()` – **atomically update shared data**
  - `synchronize()` – **wait for all concurrently active readers to exit critical section, to allow for garbage collection of old versions of the object**
  - `quiescentState()` – **Of the read-only threads on this processor who were active during the most recent** `publish()`**, all have exited the critical section.**

RCU Lock Implementation: 3/7

- RCU allows one write at a time, and it allows reads to overlap each other and writes.
- The initial version is v0, and overlapping writes update the version to v1, v2, and then v3.



RCU Lock Implementation: 4/7

- **The central goal of implementing RCU is to minimize the cost of read critical sections:**
  - **The system must allow an arbitrary number of concurrent readers**
  - **Conversely, writes can have high *latency***
  - **In particular, grace periods can be long; however, write overhead (i.e., CPU time needed per write) should be modest.**
- **A common technique for achieve these goals is to integrate the RCU implementation with that of the thread scheduler.**

RCU Lock Implementation: 5/7

- Two things are needed from the scheduler:
  - Read-only critical sections complete without being interrupted
  - Whenever a thread on a processor is interrupted, the scheduler updates some per-processor RCU state. Then, once a write completes, `synchronize()` simply waits for all processors to be interrupted at least once.
  - At that point, the old version of the object is known to be **quiescent** (i.e., no thread has access to the old version other than the writer who changed it).

RCU Lock Implementation: 6/7

Data Declarations

```
// Global state
Spinlock globalSpin;
long     globalCounter;

// One per processor
DEFINE_PER_PROCESSOR
   (static long quiescentCount);

// Per-lock state
Spinlock  writerSpin;
```

Update the Local Counter

```
// called by scheduler whenever that
//    processor is interrupted.
// it updates that processor's
//    quiescentCount to match the
//    current globalCounter.
quiescentState() {
   memory_barrier();
   PER_PROCESSOR_VAR(
      quiescentCount) = globalCounter;
   memory_barrer();
}
```

```
ReadLock() {
   disableInterrupt();
}

ReadUnlock() {
   enableInterrupts();
}
```

Reader Lock and Unlock

Read-only critical sections complete without being interrupted

RCU Lock Implementation: 7/7

```
writeLock() {
  writerSpin.acquire();
}

writeUnlock() {
  writeSpin.release();
}
```

```
publish(void **pp1, void *p2) {
   memory_barrier();
   *pp1 = p2;
   memory_barrier();
}
```

```
synchronize() {
  int p, c;
  globalSpin.acquire();    // update sync global count
    c = ++globalCounter;
  globalSpin.release();
  FOREACH_PROCESSOR(p) {   // once quiescentCount is as large as c,
                           //   on every processor, synchronize() knows
                           //   that no remaining readers can observe
                           //   the old version
    while (PER_PROC_VAR(quiescentCount,p) < c) {
       sleep(10);          // release CPU for 10ms
    }
  }
}
```

Linked List Operations: 1/4

```
typedef struct Elements {
  int key;
  int value;
  struct Elements *next;
} Element;

class RCUList {
  private:
    RCULock rcuLock;
    Element *head;
  public:
    bool search(int key, int *value);
    void insert(Element *item, int value);
    bool remove(int key);
};
```

Linked List Operations: 2/4

```
bool RCUList::search(int key, int *valuep) {
  bool result = FALSE;
  Element *current;

  rckLock.readLock();
  current = head;
  for (current = head; current != NULL; current=current->next) {
    if (current->key == key) {
      *valuep = current->value;
      result = TRUE;
      break;
    }
  }
  rcuLock.readUnlock();
  return result;
}
```

This version allows concurrent search (i.e., read-only).
The linked list is locked by a RCU `ReadLock()` and released by `ReadUnlock()`

Linked List Operations: 3/4

```
bool RCUList::insert(int key, int value) {
  Element *item;

  rcuLock.writeLock();                    // writer lock acquired
    item = (Element *) malloc(sizeof(Element)); // get memory
    item -> key = key;                    // initialize the node
    item -> value = value;
    item => next = head;

    rcuLock.publish(&head, item);         // publish it atomically

  rcuLock.writeUnlock();                  // release writer lock

  rcuLock.synchronize();                  // wait until no reader
}                                         //   has old version
```

Insertion must be a writer.
It always inserts at the end (i.e., the new node becomes the new head)
The list is locked first, a new node is created.
The new node is published and unlocks the list.
Then, wait until no reader has old versions.

Linked List Operations: 4/4

```
bool RCUList::remove(int key) {
  bool found = FALSE;
  Element *prev, *current;

  rcuLock.writeLock();
    for (prev = NULL, current = head; current != NULL;
                 prev=current, current = current->next) {
      if (current->key == key) {    // found the node
        found = TRUE;
        if (prev == NULL)           // it is the head!
          rcuLock.publish(&head, current->next);  // update new head
        else
          rcuLock.publish(&(prev->next), current->next);
      }                             // update the previous node
  rcuLock.writeUnlock();
  if (found) {
    rcuLock.synchronize();          // wait until no reader has old
    free(current);
  }
  return found;
}
```

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

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

Some Terms

- **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 (i.e., released voluntarily)
- **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();`

Thread A	Thread B	Comment
acquire lock1		Lock 1 not available
	acquire lock2	Lock 2 not available
acquire lock2		Thread A waits
	acquire lock1	Thread B waits

deadlock!!!

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 each of `buffer1` and `buffer2` has only 1 empty slot left

Thread A	Thread B	Comment
`buffer1.put()`		Buffer 1 full
	`buffer2.put()`	Buffer 2 full
`buffer1.put()`		Thread A blocks
	`Buffer2.put()`	Thread B blocks

Two Locks & a C.V.

Thread A

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

Thread B

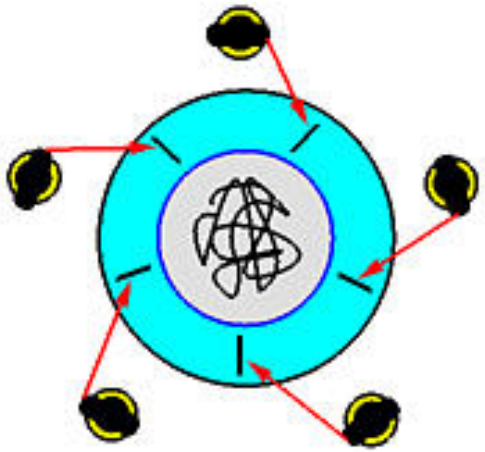
```
lock1.acquire();
   ……
   lock2.acquire();
   ……
   condition.signal(lock2);
   ……
   lock2.release();
   …
lock1.release();
```

Thread A	Thread B	Comment
`lock1.acquire()`		lock1 not available
`lock2.acquire()`		lock2 not available
`cv.wait()`		Thread A waits
	`lock1.acquire()`	Thread B cannot enter

Deadlock!
No one can free A

Dining Philosophers

- Each philosopher must picks up his left chopstick and the his right one before eating.
- It is possible that **ALL** philosophers have their left chopstick, but fail to get their right one.
- Then, we have a deadlock.
- There is a typical "circular waiting"



System Model

- System resources are used in the following way:
  - **Request**: If a process makes a request (i.e., semaphore wait or monitor acquire) to use a system resource which cannot be granted immediately, then the requesting process blocks until it can acquire the resource successfully.
  - **Use**: The process operates on the resource (i.e., in critical section).
  - **Release**: The process releases the resource (i.e., semaphore signal or monitor release).

Deadlock: Definition

- A set of processes is in a **deadlock** state when every process in the set is waiting for an event that can only be caused by another process in the same set.
- The key here is that processes are all in the **waiting state**.

Deadlock Necessary Conditions

- **If a deadlock occurs, then each of the following four conditions must hold.**
  - **Mutual Exclusion**: At least one resource must be held in a non-sharable way.
  - **Hold and Wait**: A process must be holding a resource and waiting for another.
  - **No Preemption**: Resource cannot be preempted.
  - **Circular Waiting**: P_1 waits for P_2, P_2 waits for P_3, …, P_{n-1} waits for P_n, and P_n waits for P_1.

They are NOT Sufficient Conditions

- These conditions are **not sufficient**.
- Even though some processes involve in a circular waiting situation, the system may not be deadlocked.
- Process P_1 waits for a resource being held by process P_2, P_2 waits for a resource being held by process P_3, and P_3 waits for a resource being held by process P_1.
- However, if a process A releases a resource that P_1 needs, this circular waiting is broken.
- Will show examples later.

Question

- How does the naïve solution to the Dining Philosophers problem meet the necessary conditions for deadlock?
  - Mutual Exclusion
  - No Preemption
  - Wait and Hold
  - Circular Waiting
- How can we modify it to prevent deadlock?
  - Lefty-Righty (Weirdo), four-chair, monitor solution, etc.
  - You have learned enough in *Concurrent Computing*.

Preventing Deadlock

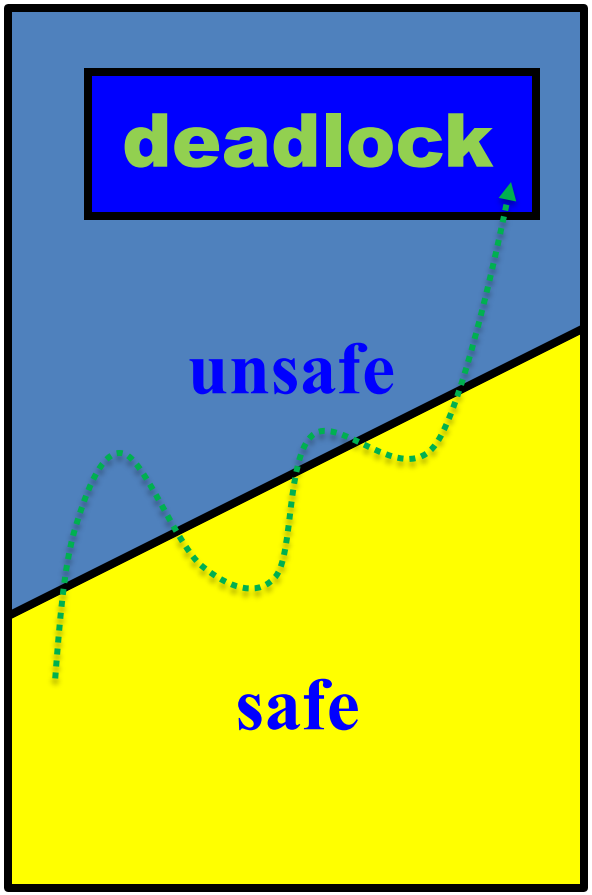
- **Make sure one of the four necessary conditions fail. You learned this in *Concurrent Computing*.**
- **Provide enough number of resources so that they do not have to be shared.**
- **Limit program from doing anything that might lead to deadlock.**
- **Predict the future: If we know what program will 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**

Deadlock Avoidance: 1/5

- Each process provides the maximum number of resources of each type it needs.
- With these information, there are algorithms that can ensure the system will never enter a deadlock state. This is *deadlock avoidance*.
- A sequence of processes $<P_1, P_2, \ldots, P_n>$ is a **safe sequence** if for each process P_i in the sequence, its resource requests can be satisfied by the remaining resources and the sum of all resources that are being held by $P_1, P_2, \ldots, P_{i-1}$. This means we can suspend P_i and run $P_1, P_2, \ldots, P_{i-1}$ until they complete. Then, P_i will have all resources to run.

Deadlock Avoidance: 2/5

- A state is **safe** if the system can allocate resources to each process (up to its maximum, of course) in some order and still avoid a deadlock.
- Thus, a state is **safe** if there is a safe sequence. Otherwise, if no safe sequence exists, the system state is **unsafe**.
- An unsafe state is not necessarily a deadlock state. On the other hand, a deadlock state is an unsafe state.



Deadlock Avoidance: 3/5

- A system has 12 tapes and three processes *A*, *B*, *C*. At time t_0, we have:

	Max needs	Current holding	Will need
A	10	5	5
B	4	2	2
C	9	2	7

3 free tapes

- Then, <*B*, *A*, *C*> is a safe sequence (safe state).
- The system has 12-(5+2+2)=3 free tapes.
- Since *B* needs 2 tapes, it can take 2, run, and return 4. After *B* completes, the system has (3-2)+4=5 tapes. *A* now can take all 5 tapes and run. Finally, *A* returns 10 tapes for *C* to take 7 of them.

Deadlock Avoidance: 4/5

- A system has 12 tapes and three processes *A*, *B*, *C*. At time t_1, *C* has one more tape:

	Max needs	Current holding	Will need
A	10	5	5
B	4	2	2
C	9	3	6

- The system has 12-(5+2+3)=2 free tapes.
- At this point, only *B* can take these 2 and run. It returns 4, making 4 free tapes available.
- But, none of *A* and *C* can run, and a deadlock occurs.
- Thus, granting *C* one more tape makes the system unsafe and leads to a deadlock.

Deadlock Avoidance: 5/5

- A **deadlock avoidance algorithm** ensures that the system is always in a safe state. Therefore, no deadlock can occur.
- Resource requests are granted only if in doing so the system is still in a safe state.
- Consequently, resource utilization may be *lower* than those systems without using a deadlock avoidance algorithm.

Banker's Algorithm: 1/6

- The idea of Banker's algorithm is very simple.
- Suppose you have $1,000 and your brothers Adam, Bill and Cassy borrowed $400, $300 and $100, respectively. Thus, you only have $200!
- What if Adam asks for $600 more?
- You don't have that amount for him. But, you think: if Bill and Cassy will return their $400 and $300 (assuming everyone is honest), respectively, you will have enough money for Adam.
- In this way, you just ask Adam to wait until Bill and Cassy return the money.
- Banker's algorithm works exactly the same way.

Banker's Algorithm: 2/6

- The system has *m* resource types and *n* processes.
- Each process must declare its maximum needs.
- The following arrays are used:
  - *Available*[1..*m*]: one entry for each resource. *Available*[*i*]=*k* means resource type *i* has *k* units available.
  - *Max*[1..*n*,1..*m*]: maximum demand of each process. *Max*[*i*,*j*]=*k* means process *i* needs *k* units of resource *j*.
  - *Allocation*[1..*n*,1..*m*]: resources allocated to each process. *Allocation*[*i*,*j*]=*k* means process *i* is currently allocated *k* units of resource *j*.
  - *Need*[1..*n*,1..*m*]: the remaining resource need of each process. *Need*[*i*,*j*]=*k* means process *i* needs *k* more units of resource *j*. Thus, *Max* = *Allocation* + *Need*.

Banker's Algorithm: 3/6

- We will use $A[i,*]$ to indicate the i-th row of matrix A.
- Given two arrays $A[1..m]$ and $B[1..m]$, $A \leq B$ if $A[i] \leq B[i]$ for all i. Given two matrices $A[1..n,1..m]$ and $B[1..n,1..m]$, $A[i,*] \leq B[i,*]$ if $A[i,j] \leq B[i,j]$ for all j.
- When a resource request is made by process i, this algorithm calls the Resource-Request algorithm to determine if the request can be granted. The Resource-Request algorithm calls the Safety Algorithm to determine if a state is safe.

Banker's Algorithm: 4/6
Safety Algorithm

```
 1.  Let Work[1..m] and Finish[1..n] be two working arrays
 2.  Work = Available;                    // make a working copy
 3.  Finish[i] = FALSE for every i;       // no one finishes yet
 4.  Find an i such that the following conditions are true
 5.    Finish[i] = FALSE;                 // process i not yet finish
 6.    Need[i,*] <= Work;                 //  & its need can be met
 7.  if (no such i exists)                // if no such i found,
 8.    goto Step 13;                      //   check if all done
 9.  else {                               // such an i found !
10.    Work = Work + Allocation[i,*]; // run it and reclaim
11.    goto Step 4;                       // go back to find next one
12.  }
13.  if (Finish[i] = TRUE for all i)// has everyone done?
14.    the system is in a safe state// yes, then we are safe
15.  else                                 // otherwise, we are not safe
16.    The system is not in a safe state;
```

Banker's Algorithm: 5/6
Safety Algorithm

```
int  Work[1..m];      // working array, one entry per resource
bool Finish[1..n];    // working array, one entry per process
int  i;
bool done, found;

Work = Available;     // use Work rather than Available directly
Finish[*] = FALSE;    // no one finishes yet
done = FALSE;         // used to control the while loop
while (!done) {
  for (i=1, found=FALSE; i <= n; i++) { // search a process …
    if (!Finish[i]                      // that has not finished
        && Need[i,*] <= Work) {         //   and can be satisfied
      Work = Work + Allocation;         // allocate and reclaim
      Finish[i] = TRUE;                 // this process is done
      found = TRUE;                     // did find a process
      break;                            // break and find another
    }
  }
  if (!found)                           // if failed to find one
     done = TRUE;                       // then while is done!
}
if (Finish[i] == TRUE for all i)
  system is in a safe state
else
  system is not in a safe state (i.e., unsafe);
```

Work is a working copy of *Available*

find a process whose *Need* can be satisfied.

Let this one run. After it finishes, the *Allocation* is taken Back and puts into the *Available* pool.

break this for and find the next

Banker's Algorithm: 6/6
Resource-Request Algorithm

```
if (Request[i,*] <= Need[i,*]) {          // must request less than Need[]
  if (Request[i,*] <= Available) {        // enough resource to satisfy?
    Available = Available - Request[i,*];     // pretend we can allocate it
    Allocation[i,*] = Allocation[i,*] + Request[i,*]; // reduce Available[]
    Need[i,*] = Need[i,*] - Request[i,*];     // reduce Need[]
    call the safety algorithm;                // after this, is the system safe?
    if (the system is in a safe state)        // if safe, allocation is successful
      allocation is done, process i has the needed resource
    else {                  // otherwise, system is not safe.  Restore all values
      restore Allocation, Need and Available to their previous state;
      process i must wait until resources will become available;
    }
  }
  else                                        // insufficient resource
    process i has to wait because of insufficient resources
else                                          // request must be <= need
  this is an error
```

Example: 1/4

- Consider a system of 5 processes *A*, *B*, *C*, *D* and *E*, and 3 resource types (*X*=10, *Y*=5, *Z*=7). At time t_0, we have

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	3	3	2
B	2	0	0	3	2	2	1	2	2			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- A safe sequence is <*B*,*D*,*E*,*C*,*A*>. Since *B*'s [1,2,2]≤ *Avail*'s [3,3,2], *B* runs. Then, *Avail*=[2,0,0]+[3,3,2]=[5,3,2]. *D* runs next. After this, *Avail*=[5,3,2]+[2,1,1]=[7,4,3]. *E* runs next.
- *Avail*=[7,4,3]+[0,0,2]=[7,4,5]. Since *C*'s [6,0,0]≤*Avail*=[7,4,5], *C* runs. After this, *Avail*=[7,4,5]+[3,0,2]=[10,4,7] and *A* runs.
- There are other safe sequences: <*D*,*E*,*B*,*A*,*C*>, <*D*,*B*,*A*,*E*,*C*>, …

Example: 2/4

- Now suppose process *B* asks for 1 *X* and 2 *Z*s. More precisely, $Request_B$ = [1,0,2]. *Is the system still in a safe state if this request is granted?*
- Since $Request_B$ = [1,0,2] ≤ *Available* = [3,3,2], this request may be granted as long as the system is safe.
- If this request is actually granted, we have the following:

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	2	3	0
B	3	0	2	3	2	2	0	2	0			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

[3,0,2]=[2,0,0]+[1,0,2] [0,2,0]=[1,2,2]-[1,0,2] [2,3,0]=[3,3,2]-[1,0,2]

Example: 3/4

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	2	3	0
B	3	0	2	3	2	2	0	2	0			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- ***Is the system in a safe state after this allocation?***
- **Yes, because the safety algorithm will provide a safe sequence <*B*,*D*,*E*,*A*,*C*>. Verify it yourself.**
- **Hence, *B*'s request of [1,0,2] can safely be made.**

Example: 4/4

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	***Y***	***Z***	***X***	***Y***	***Z***	***X***	***Y***	***Z***	***X***	***Y***	***Z***
A	0	1	0	7	5	3	7	4	3	2	3	0
B	3	0	2	3	2	2	0	2	0			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- **After this allocation, *E*'s request $Request_E$=[3,3,0] cannot be granted since $Request_E$=[3,3,0] ≤[2,3,0] is false.**
- **A's request $Request_A$=[0,2,0] cannot be granted because the system will be unsafe.**
- **If $Request_A$=[0,2,0] is granted, *Available*=[2,1,0].**
- **None of the five processes can finish and the system is unsafe.**

Deadlock Detection

- If a system does not use a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. Thus, we need
  - An algorithm that can examine the system state to determine if a deadlock has occurred. This is a *deadlock detection* algorithm.
  - An algorithm that can help recover from a deadlock. This is a *recovery* algorithm.
- A deadlock detection algorithm does not have to know the maximum need *Max* and the current need *Need*. It uses only *Available*, *Allocation* and *Request*.

Deadlock Detection Algorithm

```
 1.  Let Work[1..m] and Finish[1..n] be two working arrays
 2.  Work = Available;                    // make a working copy
 3.  Finish[i] = FALSE for every i;       // no one finishes yet
 4.  Find an i such that the following conditions are true
 5.    Finish[i] = FALSE;                 // process i not yet finish
 6.    Request[i,*] <= Work;              //  & its request can be met
 7.  if (no such i exists)                // if no such i found,
 8.    goto Step 13;                      //   check if all done
 9.  else {                               // such an i found !
10.    Work = Work + Allocation[i,*]; // run it and reclaim
11.    goto Step 4;                       // go back to find next one
12.  }
13.  if (Finish[i] = TRUE for all i)// has everyone done?
14.    the system is in a safe state// yes, then we are safe
15.  else                                 // otherwise, we are not safe
16.    The system is not in a safe state;
```

In deadlock detection, we do not have **Max** and hence **Need** can not be computed.
Therefore, users just make their **Request[ ]**!
The remaining is all the same as the banker's algorithm.

Deadlock Detection Algorithm

```
int  Work[1..m];      // working array, one entry per resource
bool Finish[1..n];    // working array, one entry per process
int  i;
bool done, found;

Work = Available;     // use Work rather than Available directly
Finish[*] = FALSE;    // no one finishes yet
done = FALSE;         // used to control the while loop
while (!done) {
  for (i=1, found=FALSE; i <= n; i++) { // search a process …
    if (!Finish[i]                      // that has not finished
          && Request[i,*] <= Work) {    // and can be satisfied
      Work = Work + Allocation;         // allocate and reclaim
      Finish[i] = TRUE;                 // this process is done
      found = TRUE;                     // did find a process
      break;                            // break and find another
    }
  }
  if (!found)                           // if failed to find one
     done = TRUE;                       // then while is done!
}
if (Finish[i] == TRUE for all i)
  system is in a safe state
else
  processes with Finish[ ] being FALSE are deadlocked;
```

The detection algorithm is the same as the safety algorithm, except that we don't have `Need[]` as we don't have `Max[]`.

Instead, we just use `Request[]` directly.

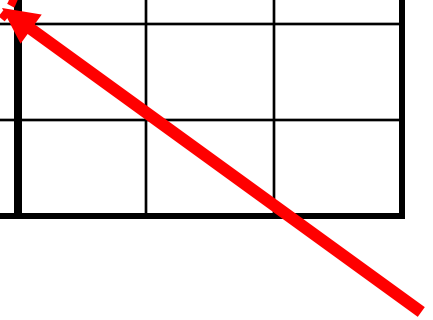
Example: 1/2

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	0	0	0	0	0	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	0			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- Suppose maximum available resource is [7,2,6] and the current state of resource allocation is shown above.
- *Is the system deadlocked?* No. We can run *A* first, making *Available*=[0,1,0].
- Then, we run *C*, making *Available*=[3,1,3]. This is followed by *D*, making *Available*=[5,2,4], and followed by *B* and *E*.

Example: 2/2

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	0	0	0	0	0	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	1			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			



- Suppose *C* requests for one more resource *Z*.
- Now, *A* can run, making *Available*=[0,1,0].
- However, none of *B*, *C*, *D* and *E* can run. Therefore, *B*, *C*, *D* and *E* are deadlocked!

The Use of a Detection Algorithm

- **Frequency**
  - **If deadlocks occur frequently, the detection algorithm should be invoked frequently.**
  - **Once per hour or whenever CPU utilization becomes low (*i.e.*, below 40%). Low CPU utilization usually means more processes are waiting.**

How to Recover: 1/3

- When a detection algorithm determines a deadlock has occurred, the algorithm may inform the system administrator to deal with it. Of, allow the system to *recover* from a deadlock.
- There are two options.
  - Process Termination
  - Resource Preemption
- These two options are not mutually exclusive (*i.e.*, can have both if needed).

Process Termination: 2/3

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- **Problems:**
  - Aborting a process may not be easy. What if a process is updating or printing a large file? The system must find some way to maintain the states of the files and printer before they can be reused.
  - Termination may be determined by the priority/importance of a process.

Resource Preemption: 3/3

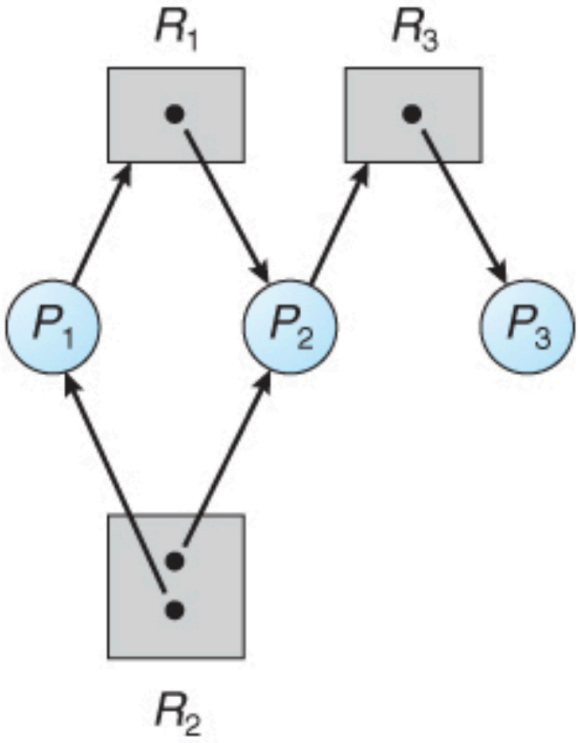
- **Selecting a victim: which resources and which processes are to be preempted?**
- **Rollback: If we preempt a resource from a process, what should be done with that process?**
  - **Total Rollback: abort the process and restart it**
  - **Partial Rollback: rollback the process only as far as necessary to break the deadlock.**
- **Starvation: We cannot always pick the same process as a victim. Some limit must be set.**

Resource Allocation Graph: 1/5

- Deadlock detection may also be graphical.
- Suppose we have m resource types, R_1, R_2, …, R_m, resource type R_i has W_i instances.
- Suppose we have n processes, P_1, P_2, …, P_n, each process is represented by a circular node.
- Each resource is represented by a rectangular node in which the number of "dots" indicate the number of instances.
- If process P_i makes a request of resource R_j, draw a directed edge from P_i to R_j.
- If process P_i receives a resource of resource R_j, draw a directed edge from a dot of R_j to P_i.

Resource Allocation Graph: 2/5

- The right diagram has 3 processes P_1, P_2 and P_3, and 3 types of resource R_1, R_2 and R_3.
- Process P_1 has a resource of R_2 and requests a resource of R_1.
- Process P_2 has a resource of R_1 and a resource of R_2, and requests a resource of R_3.
- Process P_3 has a resource of R_3.

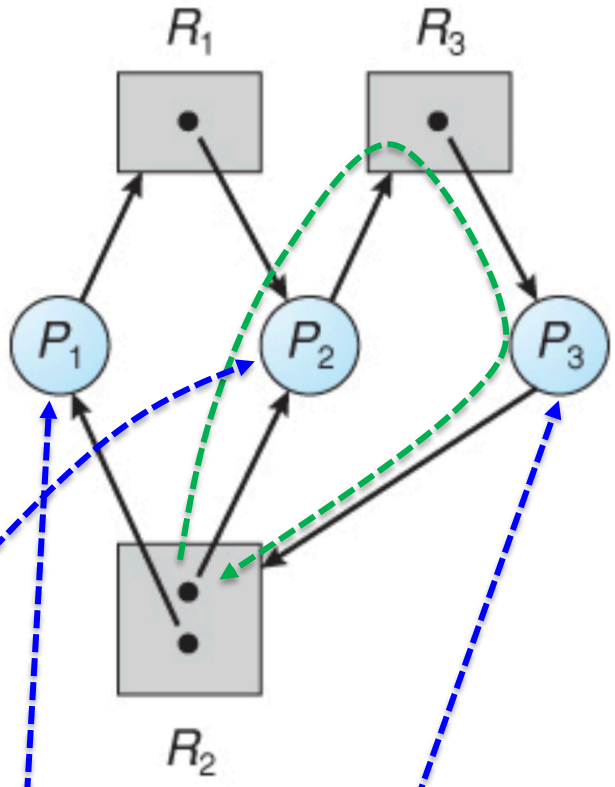


Resource Allocation Graph: 3/5

- The resource allocation graph is a **directed** graph.
- **If there is no directed cycle, the allocation has no deadlock. Why? Your homework problem. Consider the four necessary conditions.**
- If a resource allocation graph has a directed cycle,
  - If only one instance per resource type, we have a deadlock
  - If several instances per resource type, there is a possibility of deadlock.

Resource Allocation Graph: 4/5

- The right diagram has a directed cycle.
- Process P_2 has an instance of R_2, requests an instance of R_3. But, R_3's only instance is allocated to P_3.
- P_3 requests an instance of R_2.
- Hence, we have a circular waiting situation and a deadlock. Use the detection algorithm to justify this.



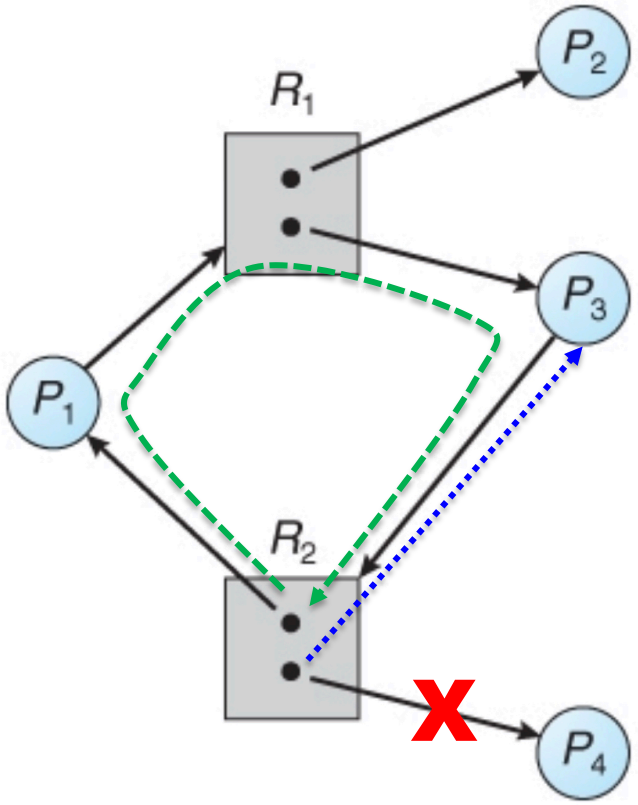
P_2 cannot run because its needed resource is held by P_3.

P_3 cannot run because its needed resource is held by P_1 or P_2.

Can P_1 run and release a resource of R_2 needed by P_3?
No, because R_1 which is held by P_2, is not available.

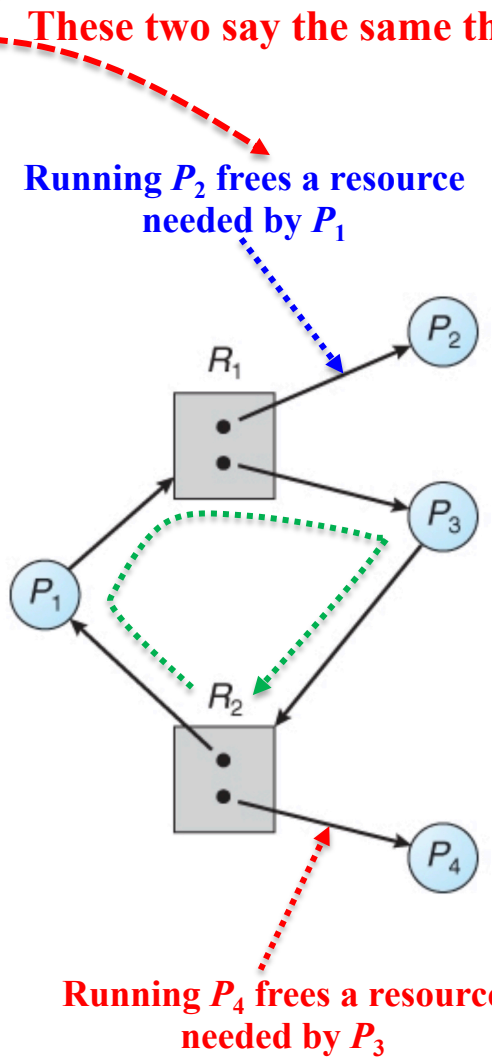
Resource Allocation Graph: 5/5

- But, having a directed cycle does not always mean there is a deadlock.
- In the right diagram, if P_4 releases the instance it holds, P_3 can have it.
- In this case, P_3 can have all the needed resources and run.
- Thus, there is no deadlock even though there is a circular waiting.



Examples: 1/5

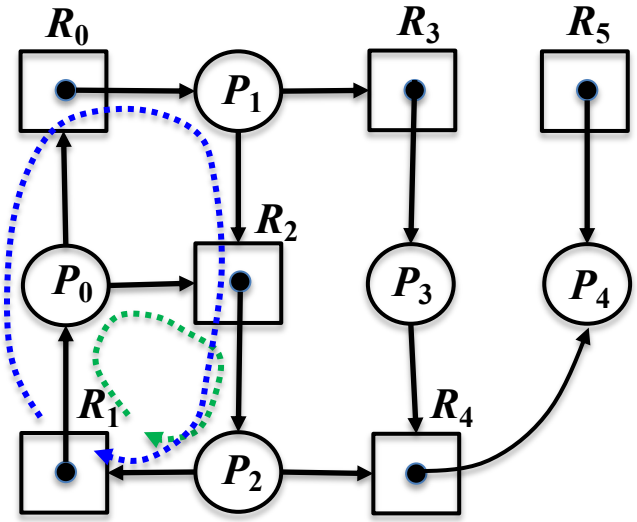
	Allocation		*Request*		*Available*	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	0	1	1	0	0	0
P_2	1	0	0	0		
P_3	1	0	0	1		
P_4	0	1	0	0		



- Because P_4 has *Request* = [0,0] which is ≤ *Available* = [0,0], we can run P_4. After P_4 finishes, it returns its *Allocation* = [0,1], making Available = [0,0] + [0,1] = [0,1].
- Now, we can run P_3 because its *Request* = [0,1] ≤ current *Available* = [0,1]. After P_3 finishes, the new *Available* = P_3's *Allocation* = [1,0] + current *Available* [0,1] = [1,1].
- Now we can run P_1 or P_2.
- There is no deadlock even though there is a cycle.

Examples: 2/5

	Allocation						*Request*					
	R_0	R_1	R_2	R_3	R_4	R_5	R_0	R_1	R_2	R_3	R_4	R_5
P_0	0	1	0	0	0	0	1	0	1	0	0	0
P_1	1	0	0	0	0	0	0	0	1	1	0	0
P_2	0	0	1	0	0	0	0	1	0	0	1	0
P_3	0	0	0	1	0	0	0	0	0	0	1	0
P_4	0	0	0	0	1	1	0	0	0	0	0	0



- *Available* = [0,0,0,0,0,0]
- Because P_4's *Request*=[0,0,0,0,0,0] ≤ *Available* = [0,0,0,0,0,0], run P_4.
- After P_4 finishes, *Available* = [0,0,0,0,0,0] + P_4's *Allocation* = [0,0,0,0,1,1] = [0,0,0,0,1,1].
- Now we can run P_3 because P_3's *Request* = [0,0,0,0,1,0] ≤ *Available* = [0,0,0,0,1,1].
- After P_3 finishes, *Available* = [0,0,0,0,1,1] + P_3's *Allocation* = [0,0,0,1,0,0] = [0,0,0,1,1,1].
- None of P_0's, P_1's and P_2's *Request* can be satisfied.
- Therefore, we have a deadlock.

Dining Philosophers, Again: 1/3

- In the dining philosophers problem, we have philosophers P_i and chopsticks C_j.
- Initially, Philosopher P_i requests chopstick C_i . The following is the initial configuration.
- Use the detection algorithm to allocate a single chopstick.
- Suppose philosopher P_j gets C_j, then P_j requests C_{j+1}.
- Then, new *Allocation*[ ] and *Request*[ ] are obtained.
- Use the detection algorithm to continue chopsticks allocation.

	Allocation					Request				
	C_1	C_2	C_3	C_4	C_5	C_1	C_2	C_3	C_4	C_5
P_1						1				
P_2							1			
P_3								1		
P_4									1	
P_5										1

Dining Philosophers, Again: 2/3

- ***n*** **chopsticks in the middle of table**
- ***n*** **dining philosophers, each can take one chopstick at a time**
- **Can deadlock occur?**
- **Use deadlock detection algorithm AND resource allocation graph to illustrates this possible deadlock.**

Dining Philosophers, Again: 3/3

- ***n*** **chopsticks in the middle of the table**
- ***n*** **philosophers, each takes one chopstick at a time**
- **Philosophers need *k* chopsticks to eat, *k* > 1**
- **Can deadlock occur?**
- **Use deadlock detection algorithm AND resource allocation graph to illustrates this possible deadlock.**

The End