CMSC421: Principles of Operating Systems

Nilanjan Banerjee

Assistant Professor, University of Maryland
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

Acknowledgments: Some of the slides are adapted from Prof. Mark Corner and Prof. Emery Berger's OS course at Umass Amherst

Announcements

- Homework 2 is out (due Oct 13th)
- Readings from Silberchatz [7th chapter]

Exercise: How do you implement reader writer locks?

Shared Data
- Data set
- Semaphore mutex initialized to 1
- Semaphore wrt initialized to 1
- Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
          wait (wrt) ;

               //    writing is performed

          signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
    } while (TRUE);
```

Monitors

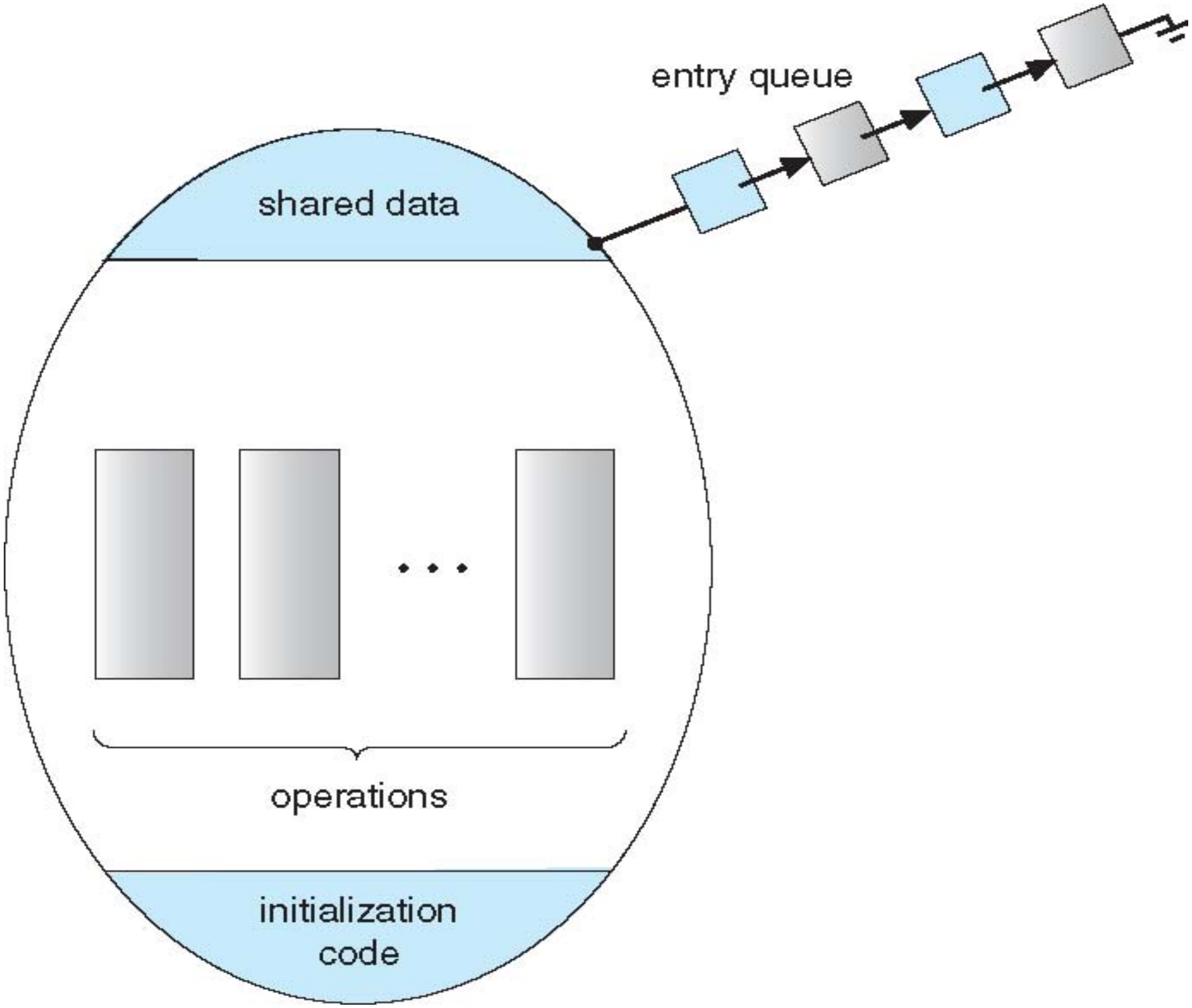
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }

    procedure Pn (…) {……}

    Initialization code (…) { … }
    }
}
```

Monitors



Implementing Locks using Swap

```
void Swap (bool *a, bool *b)
        {
                bool temp = *a;
                *a = *b;
                *b = temp:
        }
```

- Shared Boolean variable lock initialized to FALSE;
- Each process has a local Boolean variable key
- Solution:

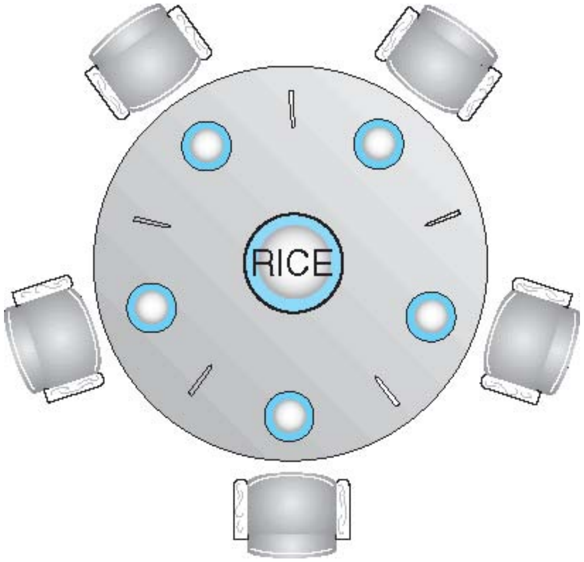
```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
          //    critical section
         lock = FALSE;

 } while (TRUE);
```

Atomic Transactions (Just a Primer!)

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher ***i***:

```
do  {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

              //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

              //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Deadlock teminology

- Deadlock

- Deadlock detection
  - Finds instances of deadlock when threads stop making progress
  - Tries to recover

- Deadlock prevention algorithms
  - Check resource requests & availability

Rules for Deadlock

- All necessary and none sufficient

Rules for Deadlock

- All necessary and none sufficient
- Finite resource
  - Resource can be exhausted causing waiting

Rules for Deadlock

- All necessary and none sufficient
- Finite resource
  - Resource can be exhausted causing waiting
- Hold and wait
  - Hold resource while waiting for another

Rules for Deadlock

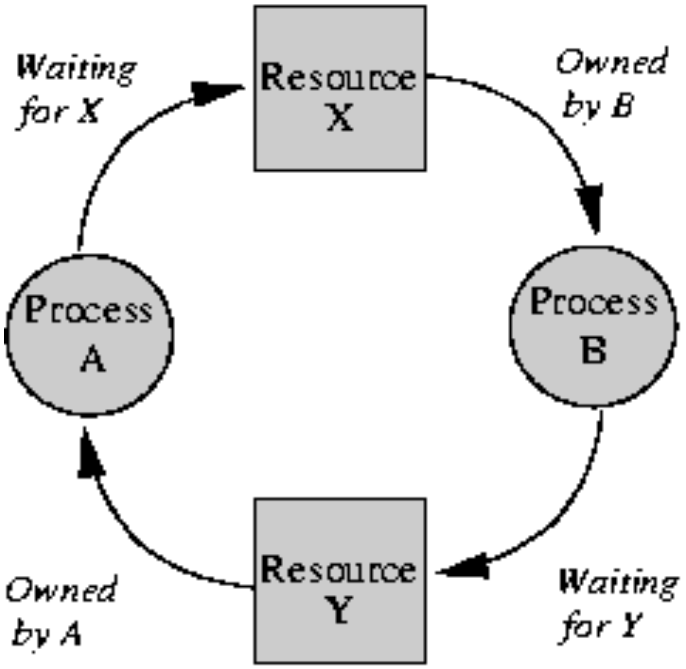
- All necessary and none sufficient
- Finite resource
  - Resource can be exhausted causing waiting
- Hold and wait
  - Hold resource while waiting for another
- No preemption
  - Thread can only release resource voluntarily
  - No other thread or OS can force thread to release

Rules for Deadlock

- All necessary and none sufficient
- Finite resource
  - Resource can be exhausted causing waiting
- Hold and wait
  - Hold resource while waiting for another
- No preemption
  - Thread can only release resource voluntarily
  - No other thread or OS can force thread to release
- Circular wait
  - Circular chain of waiting threads

Circular Waiting

- If no way to free resources (preemption)

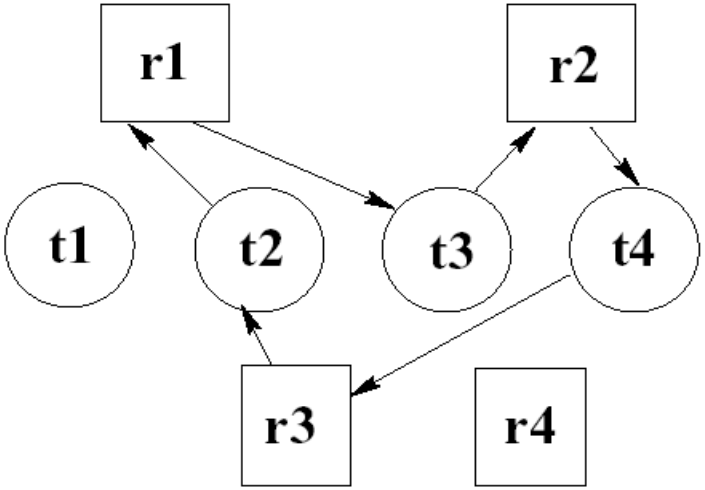


Deadlock Detection

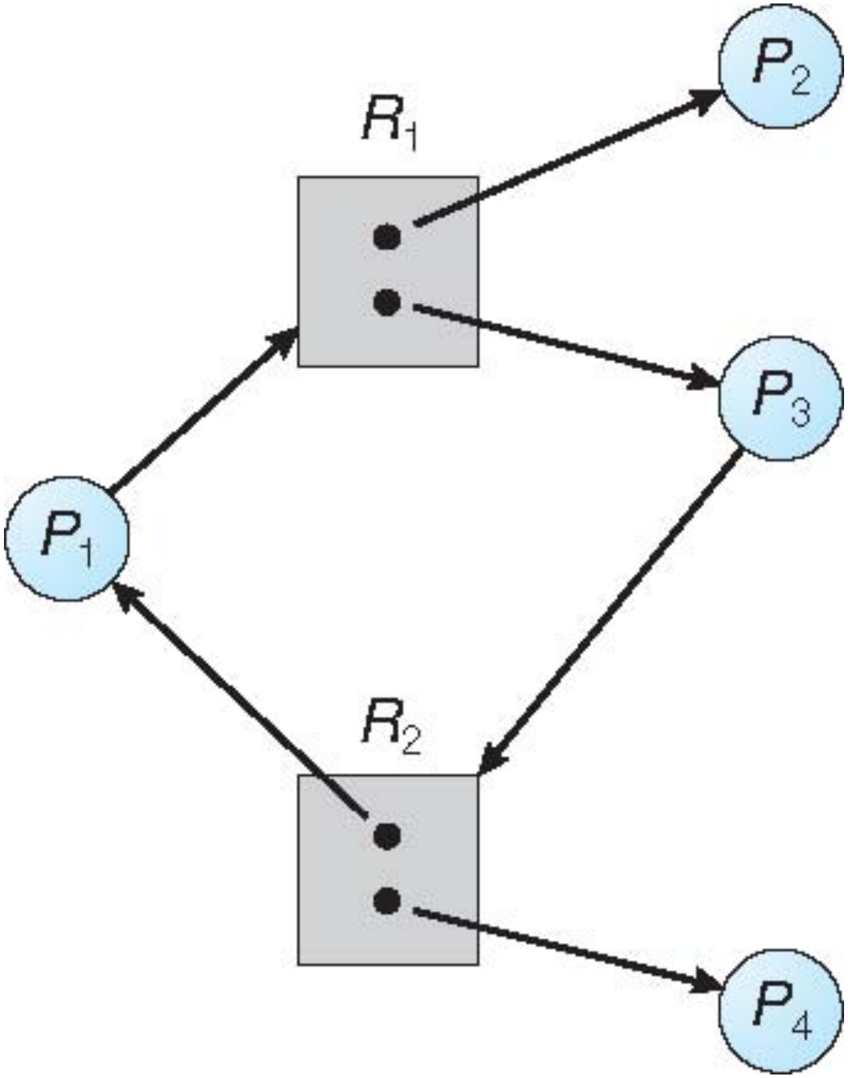
- Define graph with vertices:
  - Resources = {r1, …, rm}
  - Threads or processes = {t1, …, tn}
- Request edge from thread to resource
  - (ti → rj)
- Assignment edge from resource to thread
  - (rj → ti)
  - OS has allocated resource to thread
- Result:
  - No cycles ⇒ no deadlock
  - Cycle ⇒ may deadlock

Example

- Deadlock or not?
- Request edge from thread to resource ti -> rj
  - Thread: requested resource but not acquired it (waiting)
- Assignment edge from resource to thread rj -> ti
  - OS has allocated resource to thread

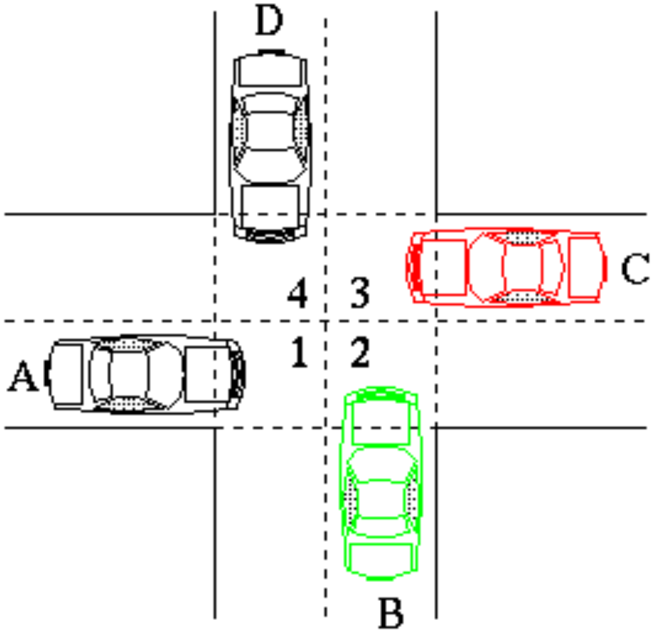


Graph With A Cycle But No Deadlock



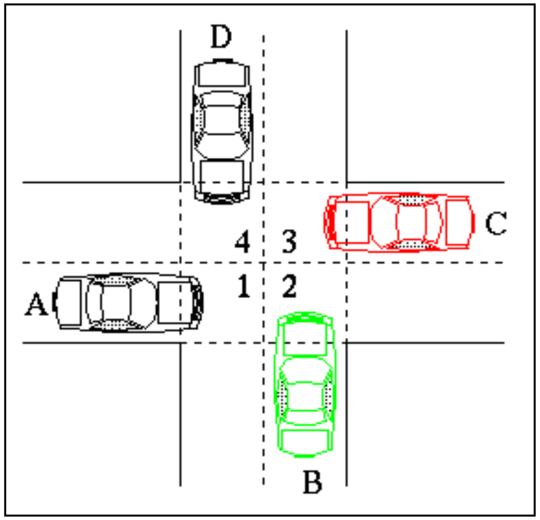
Quick Exercise

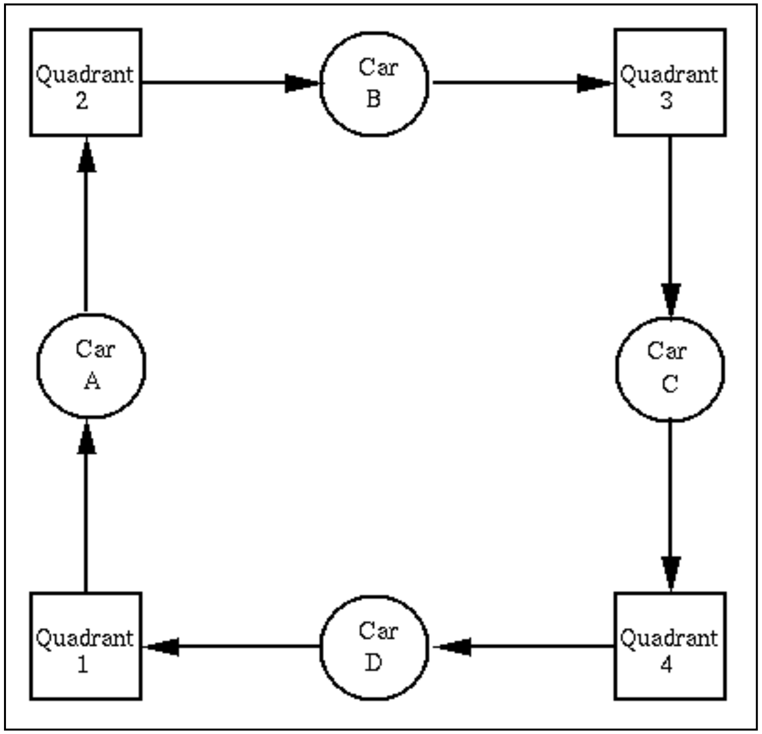
- Draw a graph for the following event:
- Request edge from thread to resource
    - ti -> rj
- Assignment edge from resource to thread
    - rj -> ti



Resource Allocation Graph

- Draw a graph for the following event:





Detecting Deadlock

- Scan resource allocation graph for cycles
  - Then break them!
- Different ways to break cycles:
  - Kill all threads in cycle
  - Kill threads one at a time
    - Force to give up resources
  - Preempt resources one at a time
    - Roll back thread state to before acquiring resource
    - Common in database transactions

Deadlock Prevention

- Instead of detection, ensure at least one of necessary conditions doesn’t hold
    - **Mutual exclusion**
    - **Hold and wait**
    - **No preemption**
    - **Circular wait**

Deadlock Prevention

- **Mutual exclusion**
  - Make resources shareable (but not all resources can be shared)
- **Hold and wait**
  - Guarantee that thread cannot hold one resource when it requests another
  - Make threads request all resources they need first and release all before requesting more

Deadlock Prevention, continued

- **No preemption**
  - If thread requests resource that cannot be immediately allocated to it
    - OS preempts (releases) all resources thread currently holds
  - When all resources available:
    - OS restarts thread
- Not all resources can be preempted!

Deadlock Prevention, continued

- **Circular wait**
  - Impose ordering (numbering) on resources and request them in order
  - *Most important trick to correct programming with locks!*

Avoiding Deadlock

- Cycle in locking graph = deadlock
- Typical solution: **canonical order** for locks
  - Acquire in increasing order
    - E.g., lock_1, lock_2, lock_3
  - Release in decreasing order

- Ensures deadlock-freedom
  - but not always easy to do

Avoiding Deadlock

- Avoiding deadlock: is this ok?

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

```
lock (b);
lock (a);
unlock (a);
unlock (b);
```

Avoiding Deadlock

- Not ok – may deadlock.

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

```
lock (b);
lock (a);
unlock (a);
unlock (b);
```

- Solution: impose canonical order (acyclic)

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

```
lock (a);
lock (b);
unlock (b);
unlock (a);
```

Deadlock Avoidance

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

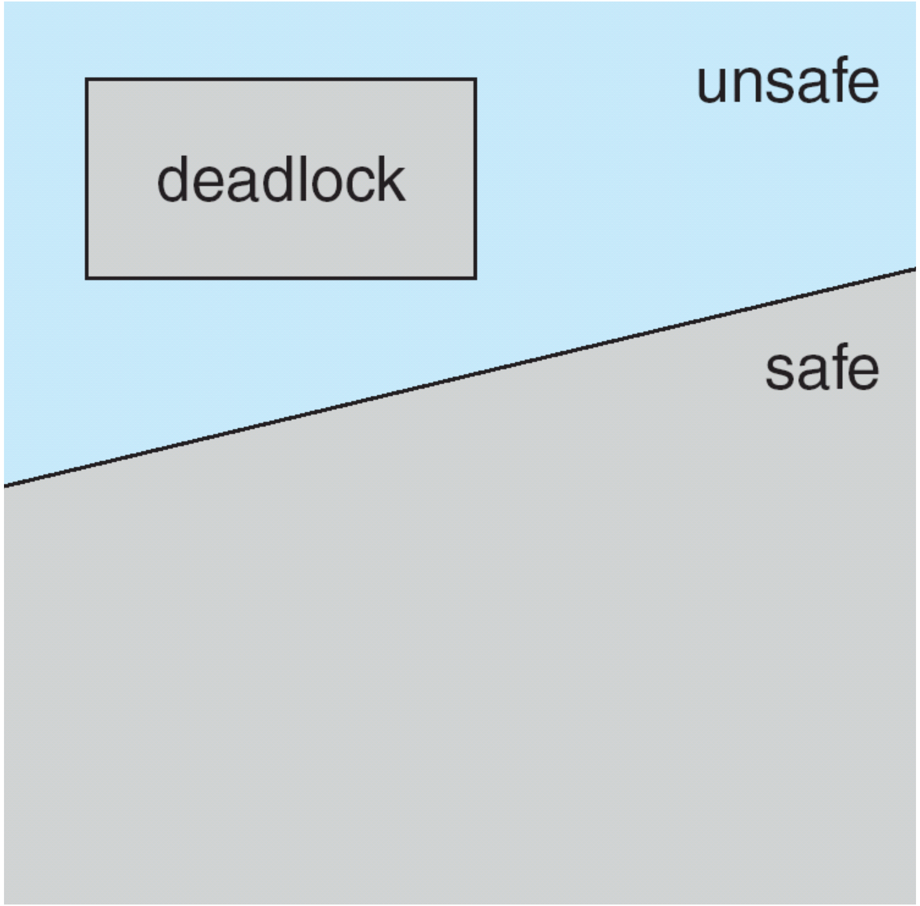
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



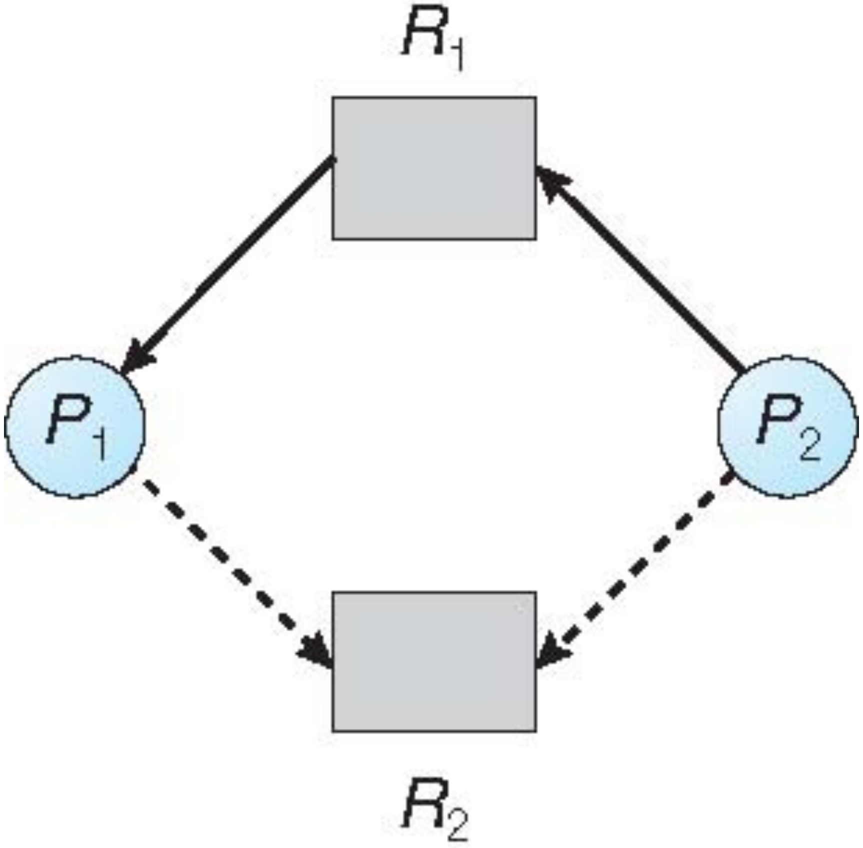
Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker’s algorithm

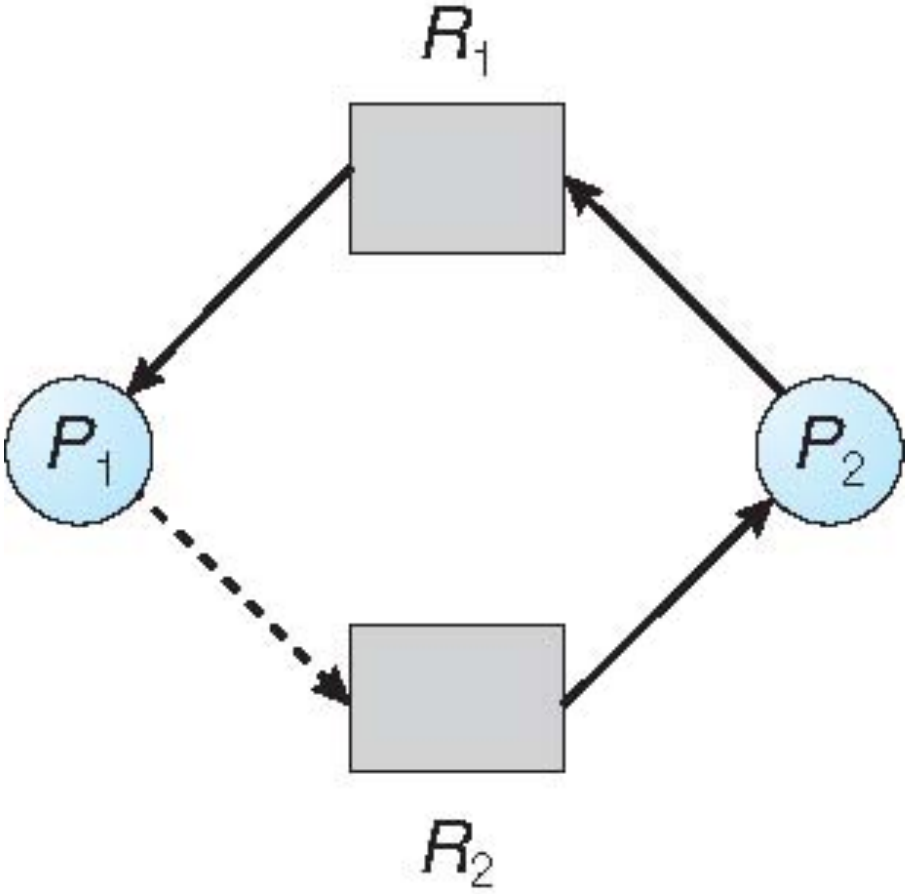
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Example of the Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example of the Banker's Algorithm

- The content of the matrix *Need* is defined to be *Max* – *Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example of the Banker's Algorithm

- Check that Request $\le$ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

An in-class discussion
(surprise : Java swapping)