CSS430
Deadlocks
Textbook Chapter 7

Instructor: Stephen G. Dame
e-mail: sdame@uw.edu

These slides were adapted from the OSC textbook slides (Silberschatz, Galvin, and Gagne), Professor Munehiro Fukuda and the instructor's class materials.

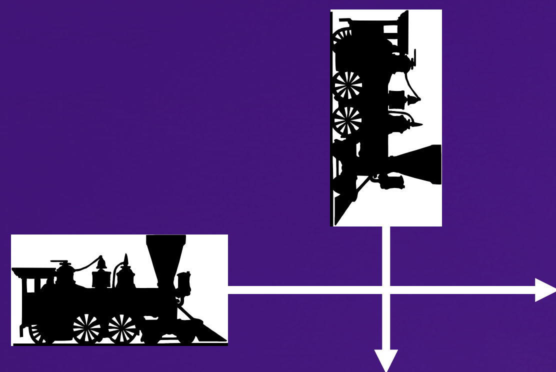
WKP 17

"The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague."

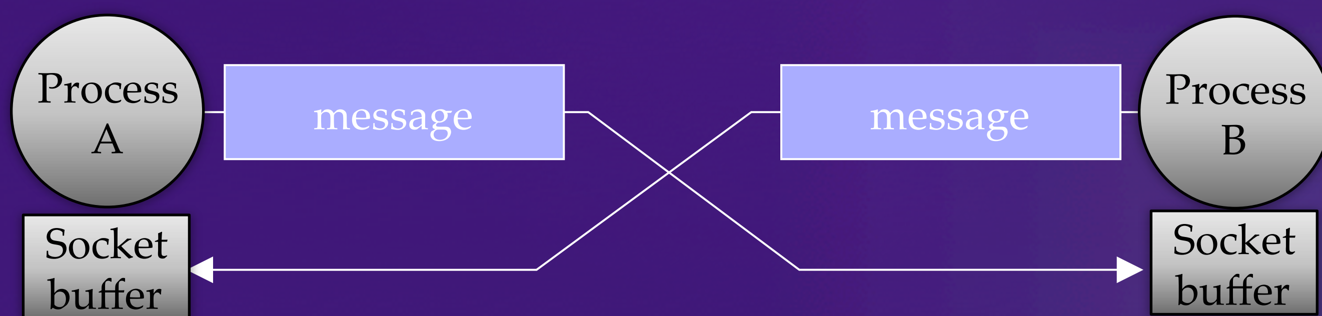
- Edsger Dijkstra

Deadlock Examples 1

Kansas Legislature: “when two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

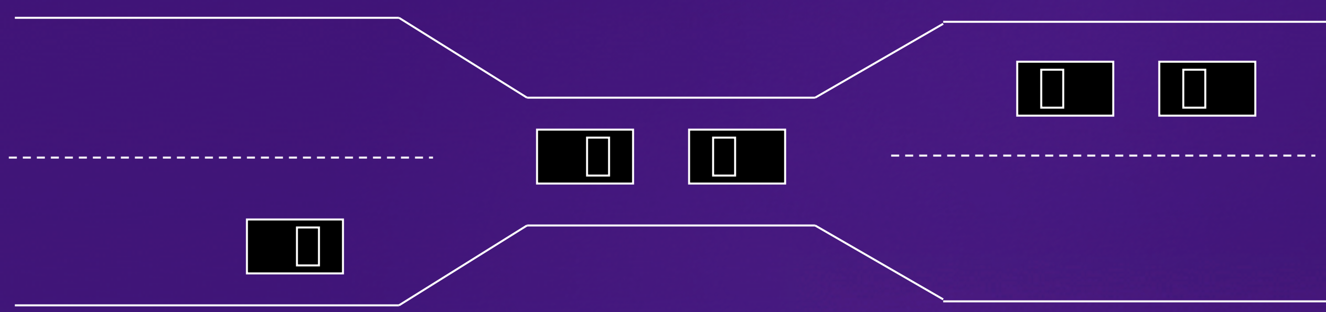


Two processes exchange a long message with each other, but their socket buffer is smaller than the message.



Deadlock Examples 2

Bridge crossing example: traffic only in one direction where two cars are driving from the opposite direction. A deadlock is not resolved unless one gets back up.



Two processes try to go into the same nested critical section in a different order.

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

System Model

- Resource types R_1, R_2, . . ., 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**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

① **Mutual exclusion:** only one process at a time can use a resource.

② **Hold and wait:** a process holding resource(s) is waiting to acquire additional resources held by other processes.

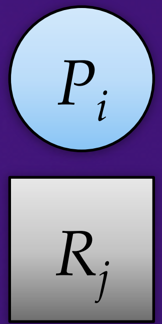
③ **No preemption:** a resource can be released only voluntarily by the process holding it upon its task completion.

④ **Circular wait:** there exists a set {P_0, P_1, …, P_0} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

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



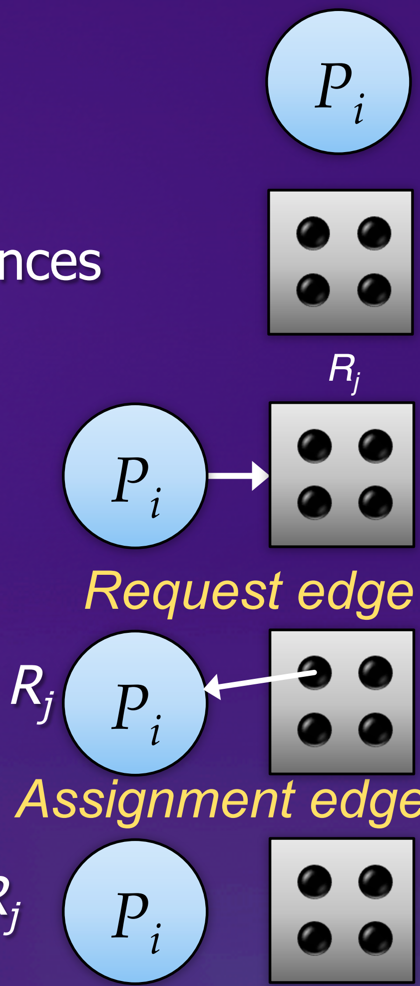
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

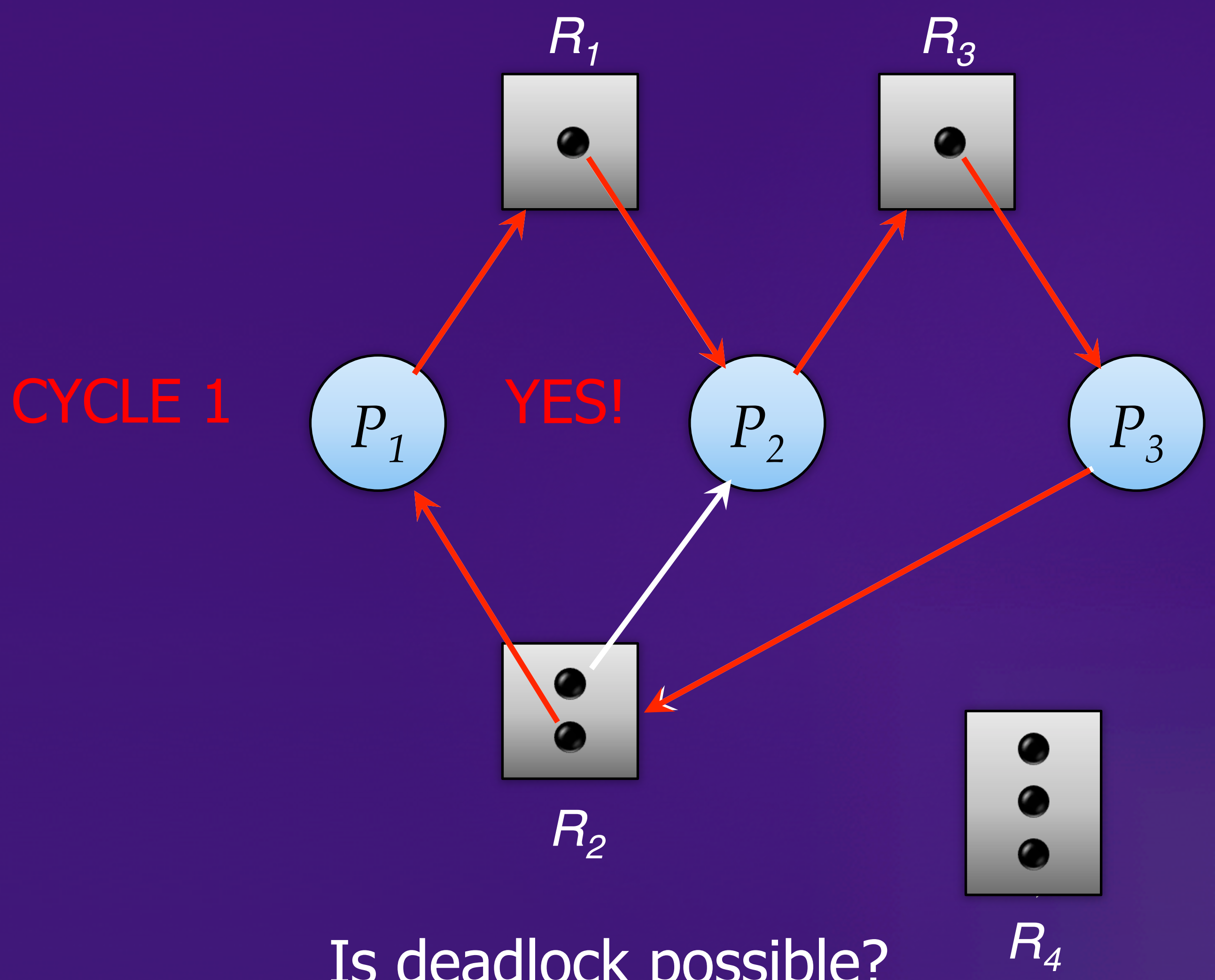
- Process
- Resource Type with 4 instances

Sequence of process resource utilization:

- P_i requests instance of R_j
- P_i is holding an instance of R_j
- P_i releases an instance of R_j

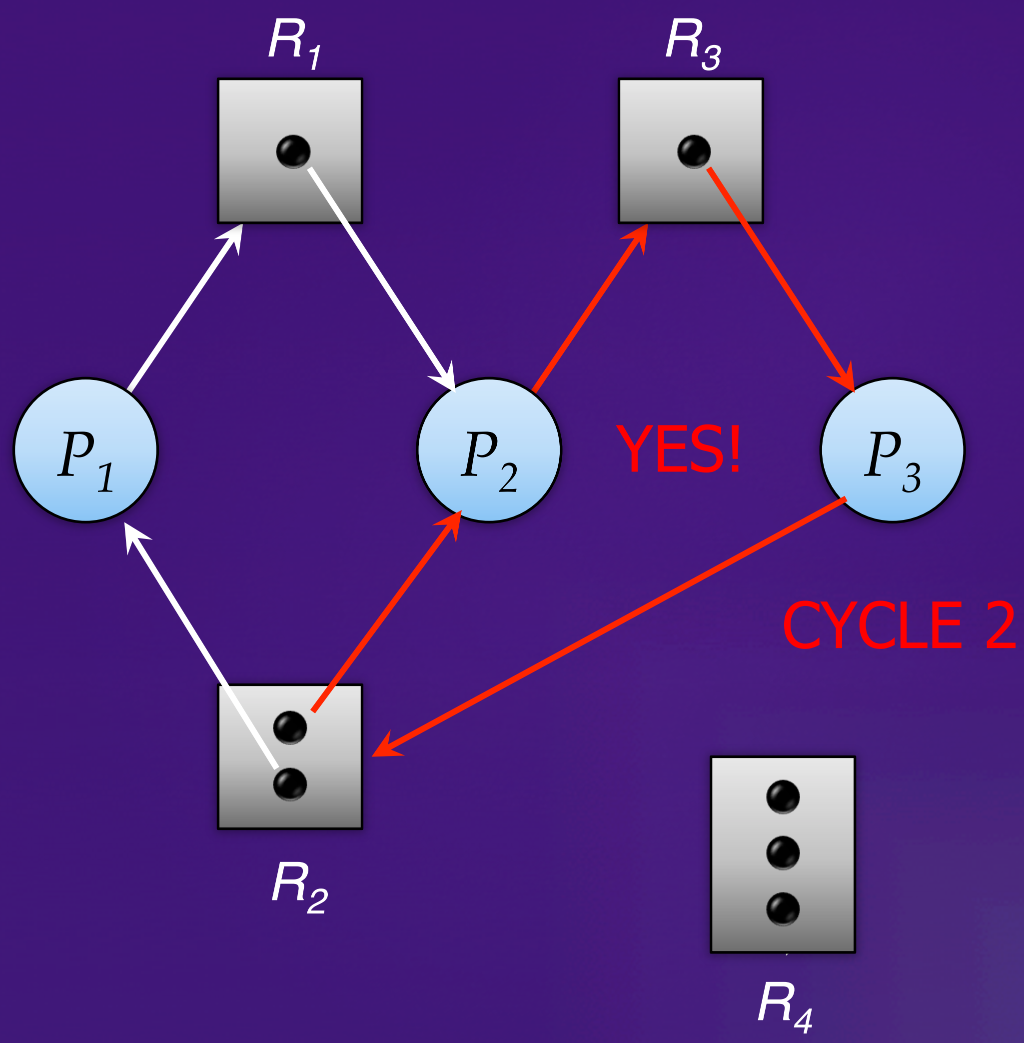


Resource-Allocation Graph

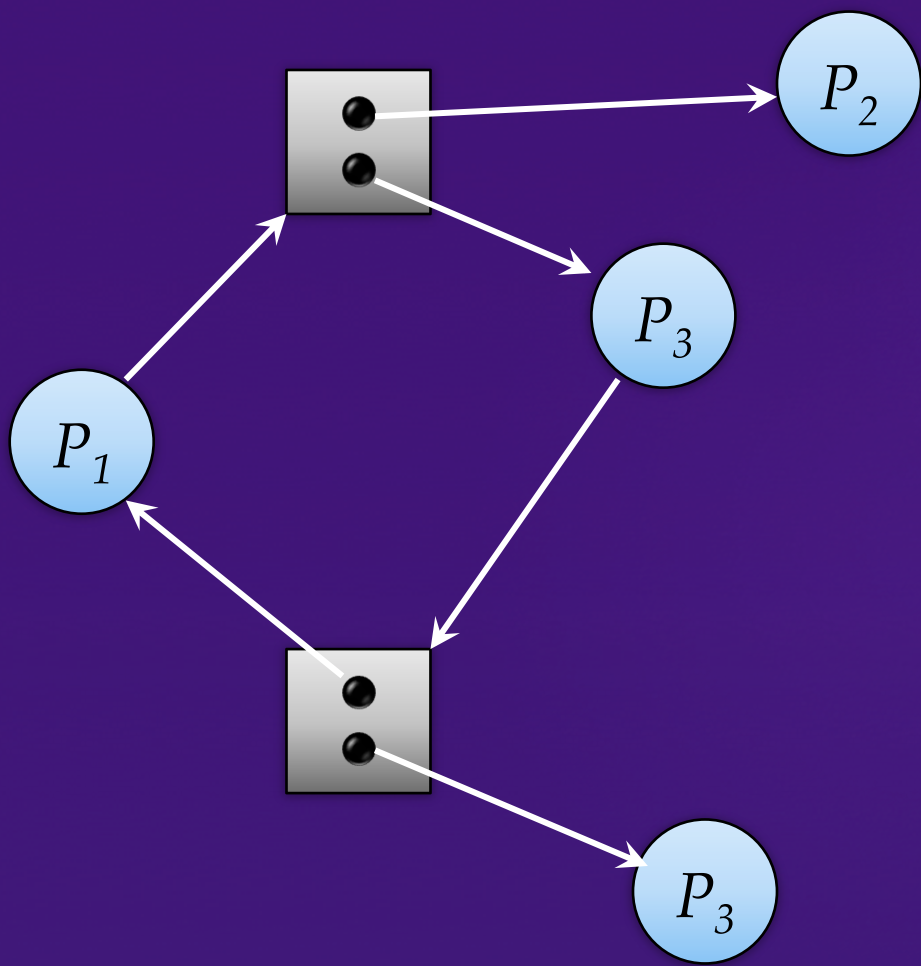


Is deadlock possible?

Resource-Allocation Graph



Resource Allocation Graph With A Cycle But No 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.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state. (Prevention and Avoidance)
- Allow the system to enter a deadlock state and then recover. (Detection and Recovery)
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Restrain one of the following four conditions:

① **Mutual Exclusion** – not required for sharable resources. (but not work always.)

② **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.

- Require a process to request and be allocated all its resources before its execution: Low resource utilization
- Allow process to request resources only when the process has none: starvation possible.

③ **No Preemption** –

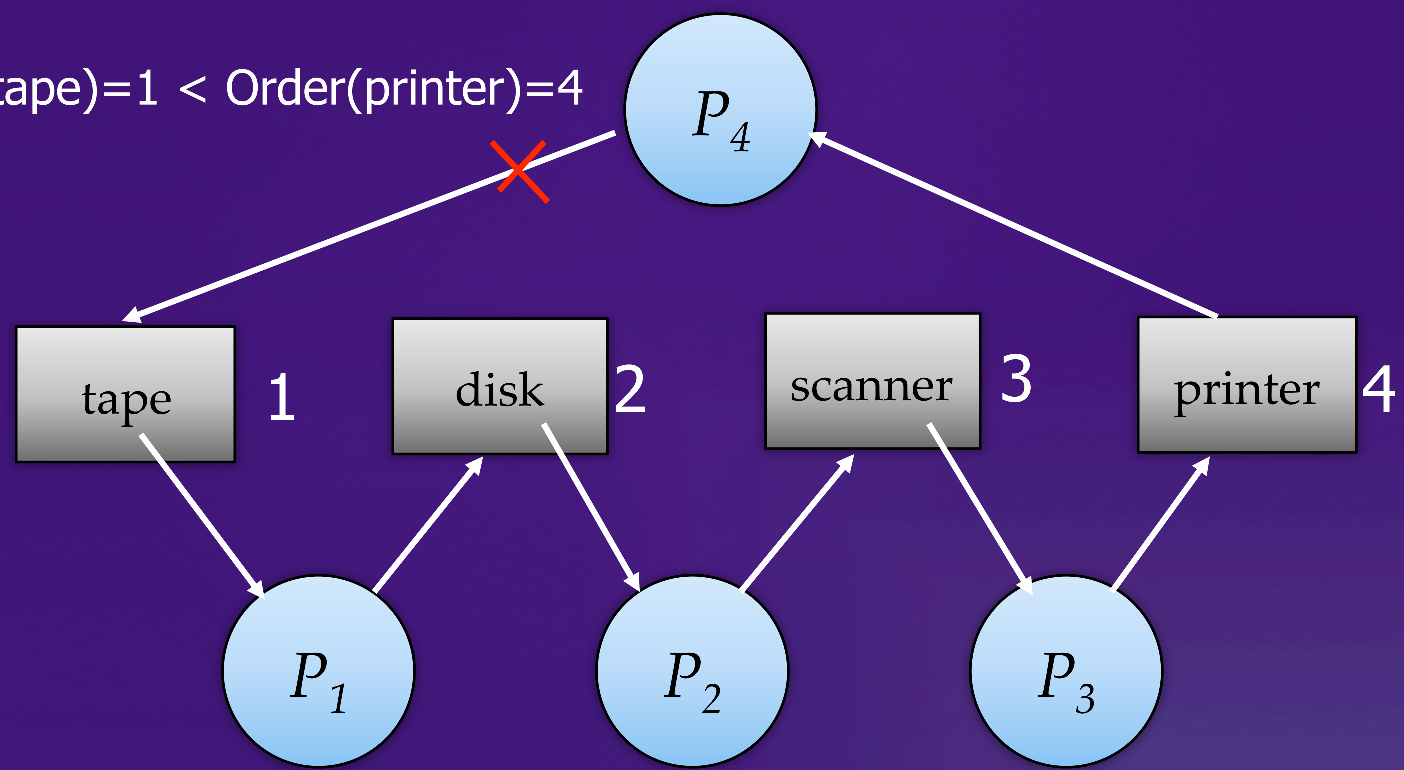
- If a process holding some resources requests another resource that cannot be immediately allocated to it, all resources currently being held are released.
- If a process P1 requests a resource R1 that is allocated to some other process P2 waiting for additional resource R2, R1 is allocated to P1.

④ **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Prevention
Circular Wait

Each process can request resources only in an increasing order of enumeration.

Not allowed Order(tape)=1 < Order(printer)=4



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

  $<P_1, P_2, \ldots, P_n>$

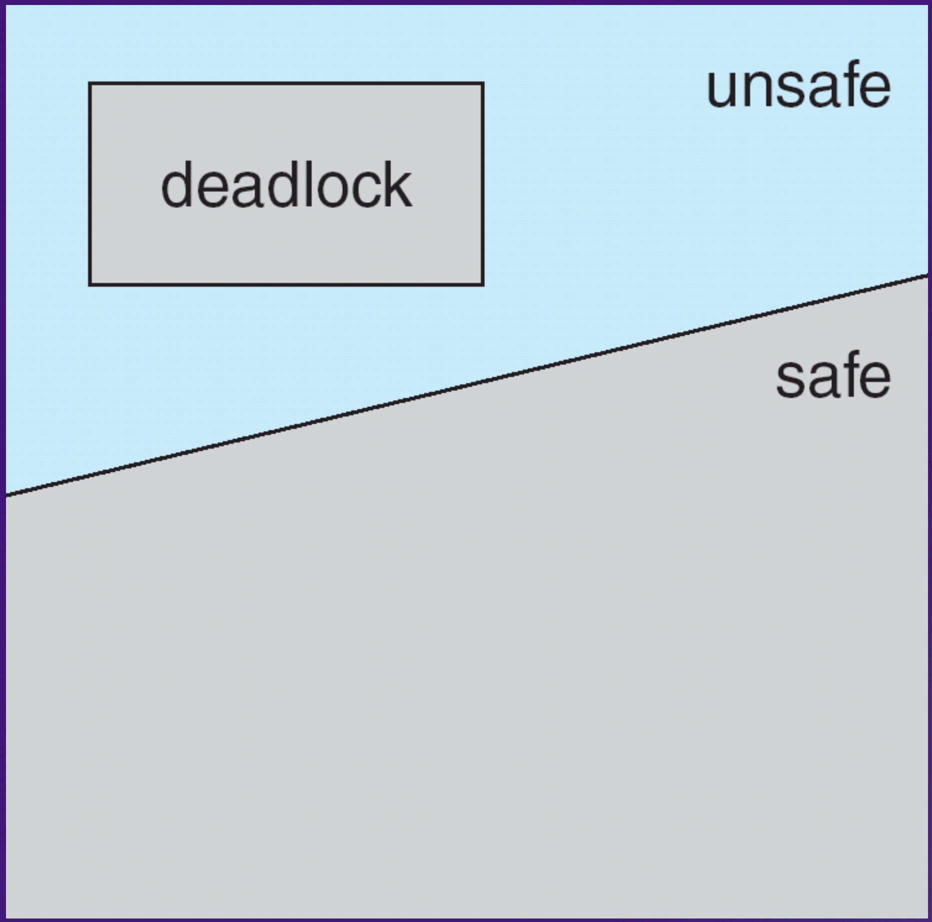
  of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - ✓ If P_i resource needs are not immediately available, then P_i can wait until all P_j have 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.

A system is in a safe state only if there exists a only if there exists a safe sequence.

Basic Facts

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

Safe, Unsafe , Deadlock State



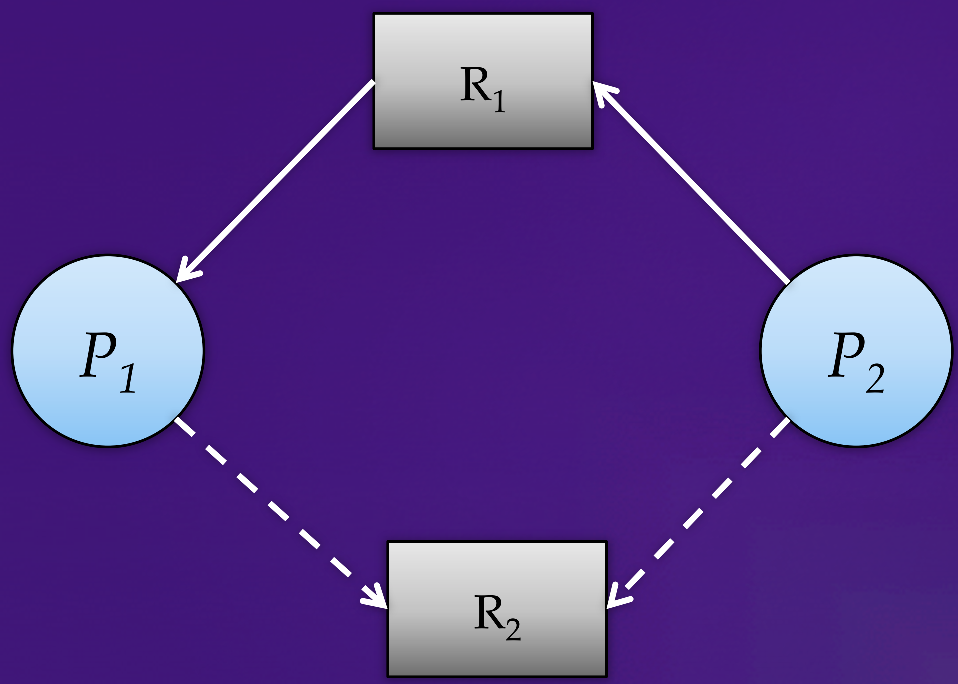
Avoidance algorithms

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

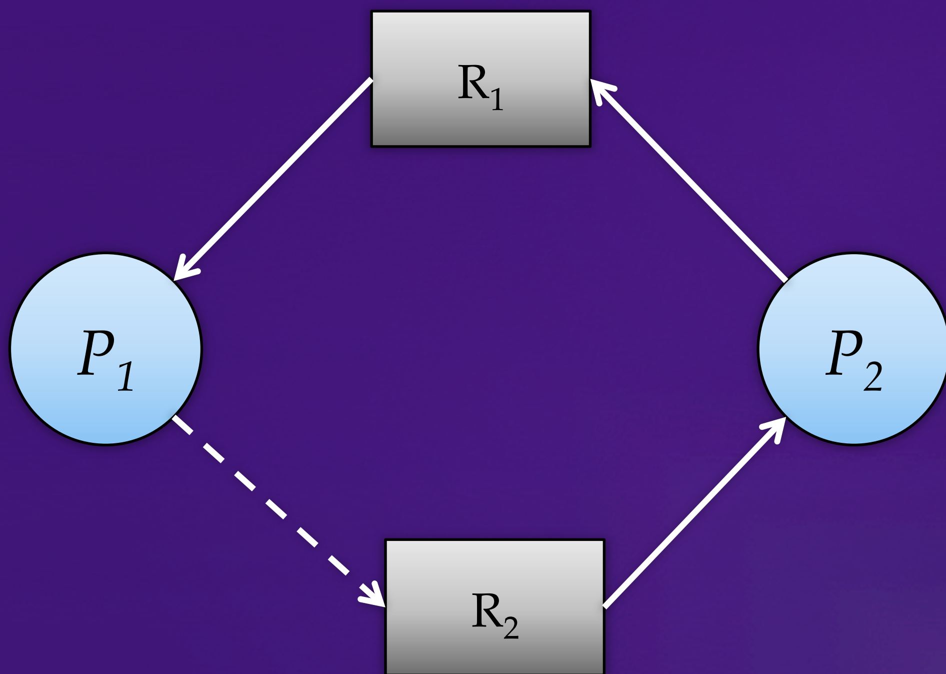
Resource-Allocation Graph Scheme

- **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



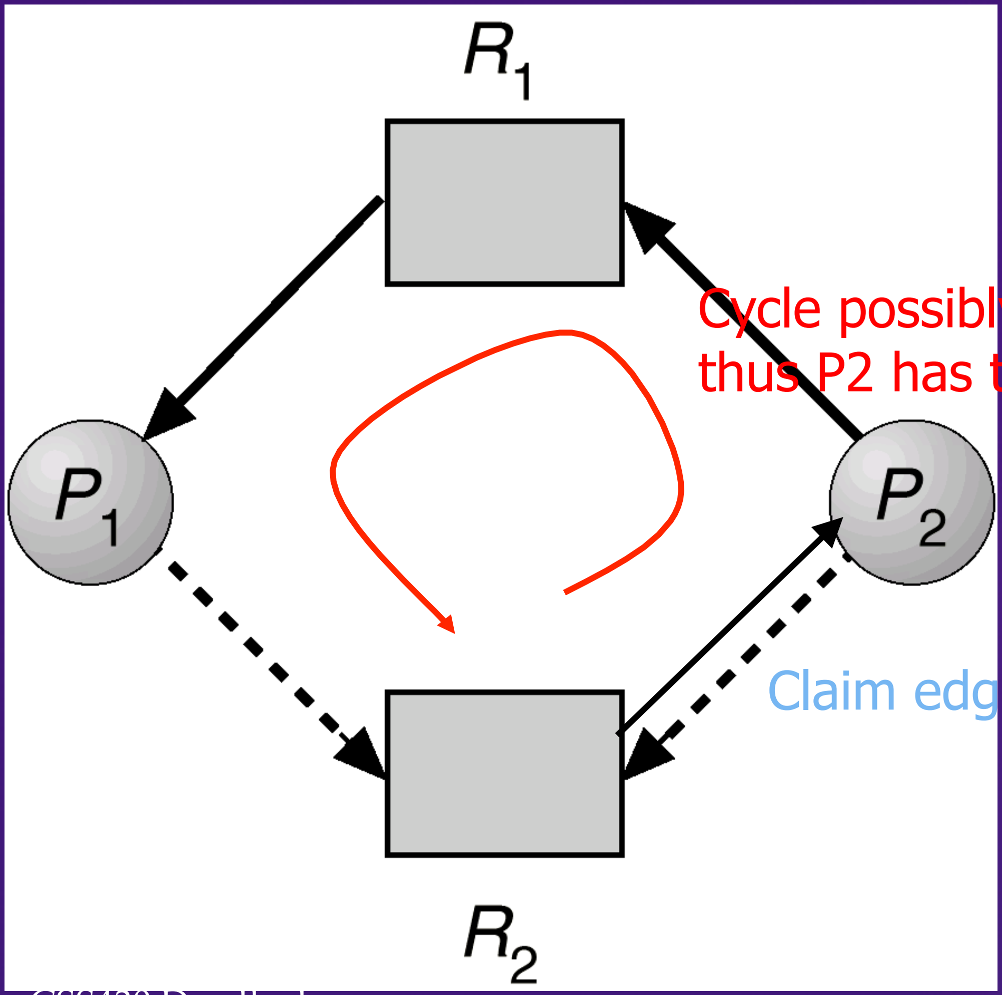
Resource-Allocation Graph Algorithm

- 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

Deadlock Avoidance

Resource-Allocation Algorithm

Processes supply OS with future resource requests



Cycle possibly formed (unsafe state), thus P2 has to wait for a safe state

Claim edge (future request)

Works only with single instance resource types.

Deadlock Avoidance

Banker's Algorithm - Definitions

- Multiple resource 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

Deadlock Avoidance

Banker's Algorithm - Definitions

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

- **Available** – Vector of length m indicates the number of available resources of each type. If $Available[j] = k$, then k instances of resource type R_j are available.
- **Max** : $n \times m$ matrix. Defines the maximum demand of each process. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- **Allocation** : $n \times m$ matrix. Defines the number of resources of each type currently allocated to each process. If $Allocation[i,j] = k$, then process P_i is currently allocated k instances of resource type R_j.
- **Need** : $n \times m$ matrix. Indicates the remaining resource need of each process. If $Need[i,j] = k$, then process P_i may need k more instances of resource type R_j to complete its task. Note that

$$Need[i,j] = Max[i,j] - Allocation[i,j]$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find an *i* such that both:

   (a) *Finish*[*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4

3. *Work* = *Work* + *Allocation*

   Finish[*i*] = *true*

   go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i [j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \le Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. Pretend to allocate requested resources to Pi by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Deadlock Avoidance
Banker's Algorithm

- 5 processes P0 through P4
- Each process must claim **Max** use in advance.
- Resource Types: A (10 instances), B (5instances), and C (7 instances)

Process	Allocation	Max	Need	Initial	Avail
	A B C	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	10 5 7	3 3 2
P1	2 0 0	3 2 2	1 2 2		
P2	3 0 2	9 0 2	6 0 0		
P3	2 1 1	2 2 2	0 1 1		
P4	0 0 2	4 3 3	4 3 1		

Snapshot at time T_0:

Deadlock Avoidance

Banker's Algorithm – P_1 Request (1 0 2)

- Check that Request ≤ Available (that is, (1 0 2) ≤ (3 3 2) ⇒ true
- Execute safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement

Process	Allocation	Max	Need	Initial	Avail
	A B C	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	10 5 7	2 3 0
P1	3 0 2	3 2 2	1 2 2		
P2	3 0 2	9 0 2	6 0 0		
P3	2 1 1	2 2 2	0 1 1		
P4	0 0 2	4 3 3	4 3 1		

Snapshot at time T_1:

Deadlock Avoidance

Banker's Algorithm – P_4 Request (3 3 0)

- Check that Request ≤ Available (that is, (3 3 0) ≤ (3 3 2) ⇒ true

Process	Allocation	Max	Need	Initial	Avail
	A B C	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	10 5 7	0 0 2
P1	2 0 0	3 2 2	1 2 2		
P2	3 0 2	9 0 2	6 0 0		
P3	2 1 1	2 2 2	0 1 1		
P4	3 3 2	4 3 3	4 3 1		

Snapshot at time T_1:

Deadlock Avoidance

Banker's Algorithm – P_0 Request (4 3 0)

- Check that Request ≤ Available (that is, (4 3 0) ≤ (3 3 2) ⇒ false

Process	Allocation	Max	Need	Initial	Avail
	A B C	A B C	A B C	A B C	A B C
P0	0 1 0	7 5 3	7 4 3	10 5 7	3 3 2
P1	2 0 0	3 2 2	1 2 2		
P2	3 0 2	9 0 2	6 0 0		
P3	2 1 1	2 2 2	0 1 1		
P4	0 0 2	4 3 3	4 3 1		

Snapshot at time T_1:

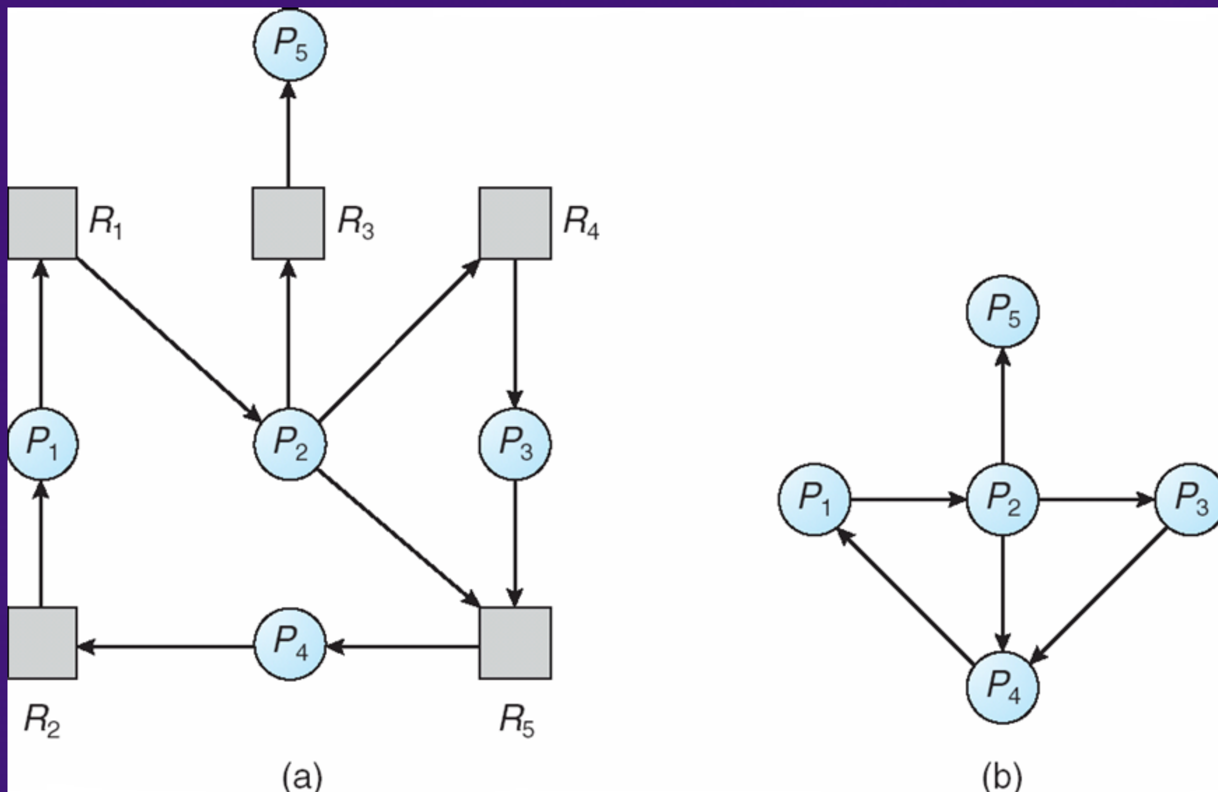
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available**: A vector of length *m* indicates the number of available resources of each type.
- **Allocation**: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*

2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

This algorithm requires an order of O($m \times n^2$) operations to detect whether the system is in deadlocked state.

Detection Algorithm Example

- Five processes P_0 through P_4;
- Three resources types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Sequence <P0, P2, P3, P1, P4> will result in *Finish*[*i*] = true for all *i*

Process	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Snapshot at time T_0

Example (Cont.)

- P_2 requests an additional instance of type C

Process	Request
	A B C
P0	0 0 0
P1	2 0 1
P2	0 0 1
P3	1 0 0
P4	0 0 2

- State of system?
  - ✓ Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - ✓ Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - ✓ How often a deadlock is likely to occur?
  - ✓ How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - ✓ Priority of the process
  - ✓ How long process has computed, and how much longer to completion
  - ✓ Resources the process has used
  - ✓ Resources process needs to complete
  - ✓ How many processes will need to be terminated
  - ✓ Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

Exercises (No turn-in)

1. Why aren't deadlock detection and recovery so attractive?
2. Solve Exercise 7.3, 7.6, 7.9, 7.10, 7.14, and 7.19
3. Can the Java code in the next slide cause a deadlock? If so, write a resource allocation graph with a deadlock.

```java
public class Deadlock {

   public Deadlock( ) {
         Mutex mutex[] = new Mutex[4];

         for ( int i = 0; i < 4; i++ )
            mutex[i] = new Mutex( );

         A threadA = new A( mutex );
         B threadB = new B( mutex );
         C threadC = new C( mutex );

         threadA.start( );
         threadB.start( );
         threadC.start( );
   }


   public static void main( String arg[] ) {
         Deadlock d = new Deadlock( );
   }

   class Mutex{ }

   private class A extends Thread
   {
         private Mutex[] resource;
         public A( Mutex[] m ) {
            resource = m;
         }
         public void run( ) {
            System.out.println( "A started" );
            synchronized ( resource[1] ) {
                  System.out.println( "A got rsc 1" );
                  synchronized ( resource[0] ) {
                     System.out.println( "A got rsc 0" );
                  }
            }
            System.out.println( "A finished" );
         }
   }
private class B extends Thread
   {
         private Mutex[] resource;
         public B( Mutex[] m ) {
            resource = m;
         }
         public void run( ) {
            System.out.println( "B started" );
            synchronized ( resource[3] ) {
                  System.out.println( "B got rsc 3" );
                  synchronized ( resource[0] ) {
                     System.out.println( "B got rsc 0" );
                     synchronized ( resource[2] ) {
                           System.out.println( "B got rsc 2" );
                     }
                  }
            }
            System.out.println( "B finished" );
         }
   }

   private class C extends Thread
   {
         private Mutex[] resource;
         public C( Mutex[] m ) {
            resource = m;
         }
         public void run( ) {
            System.out.println( "C started" );
            synchronized ( resource[2] ) {
                  System.out.println( "C got rsc 2" );
                  synchronized ( resource[1] ) {
                     System.out.println( "C got rsc 1" );
                  }
            }
            System.out.println( "C finished" );
         }
   }
}
```