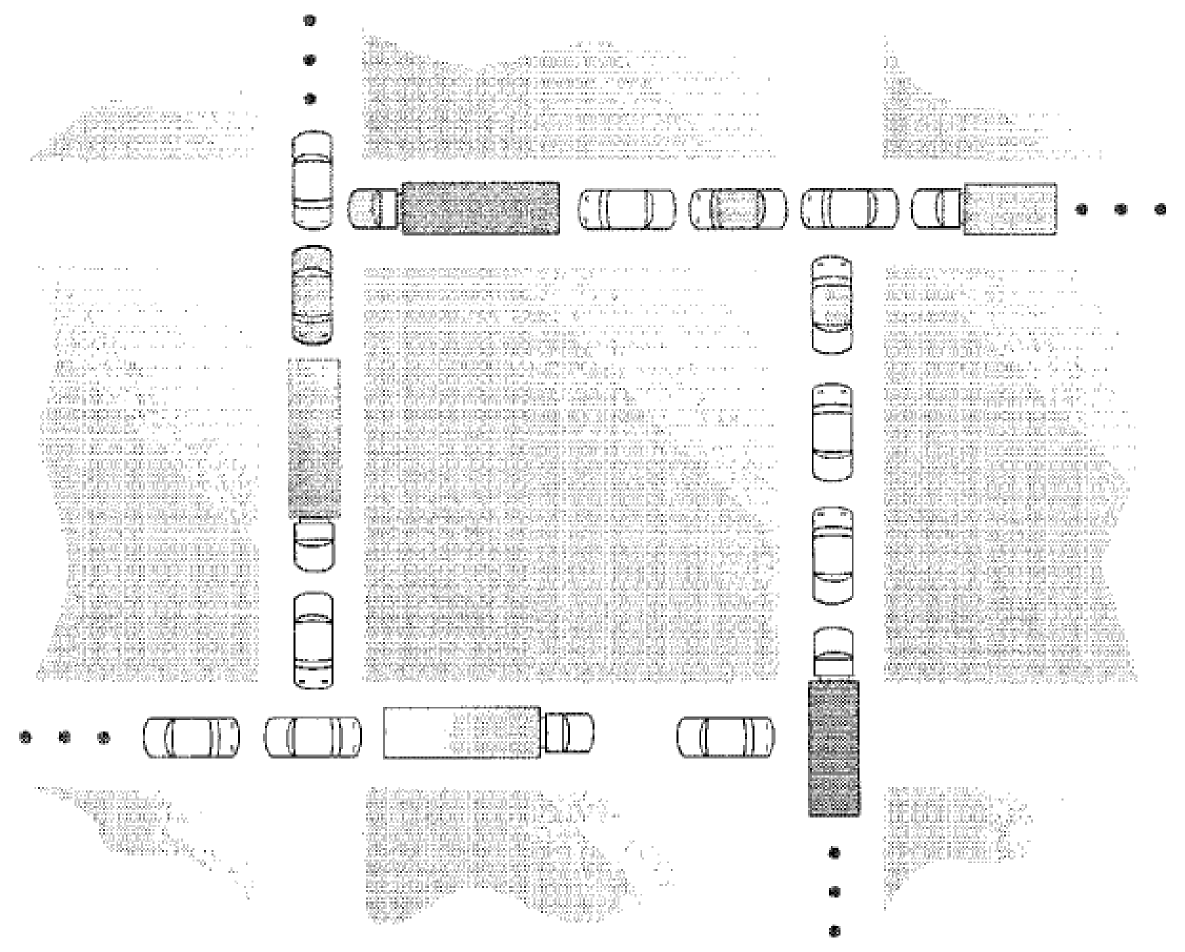
This image cannot currently be displayed.

Chapter 7: Deadlocks

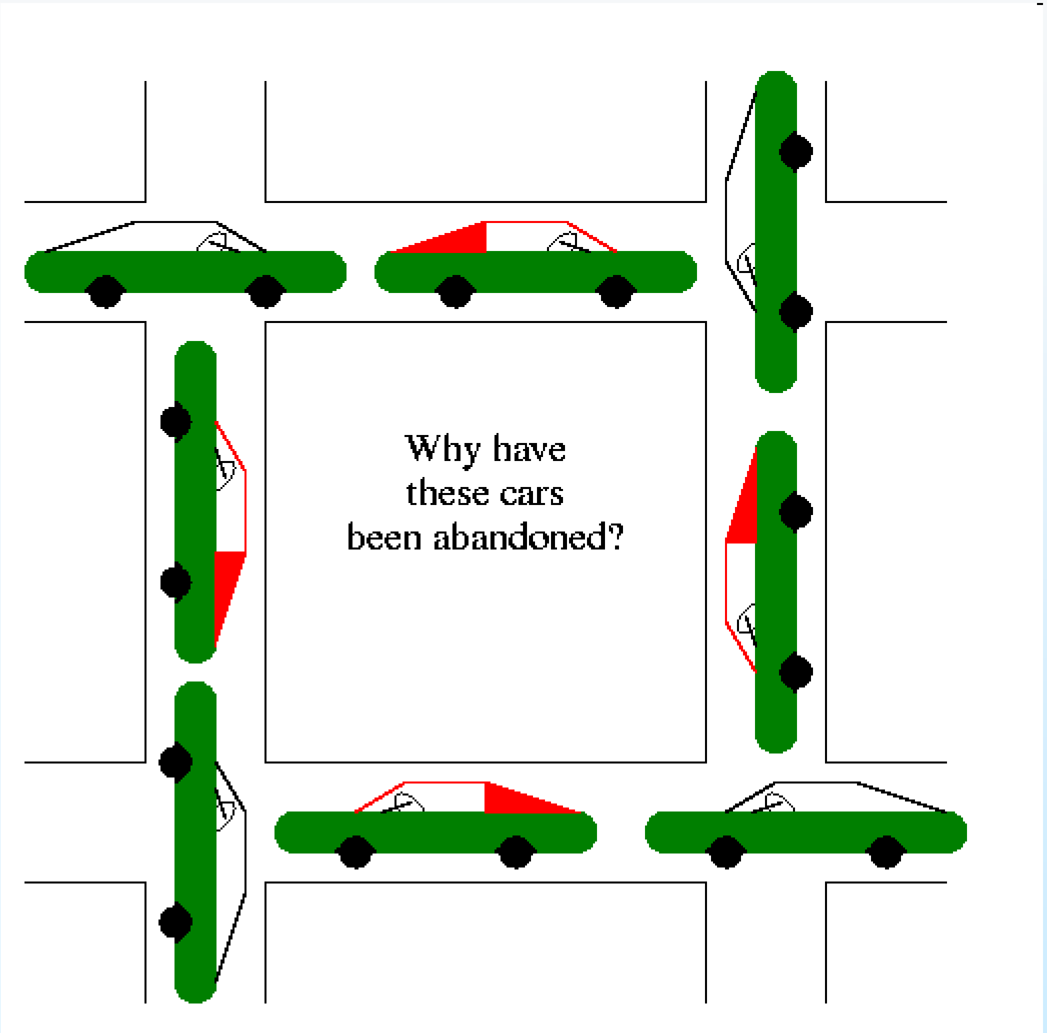
Chapter 7: Deadlocks

- **The Deadlock Problem**
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

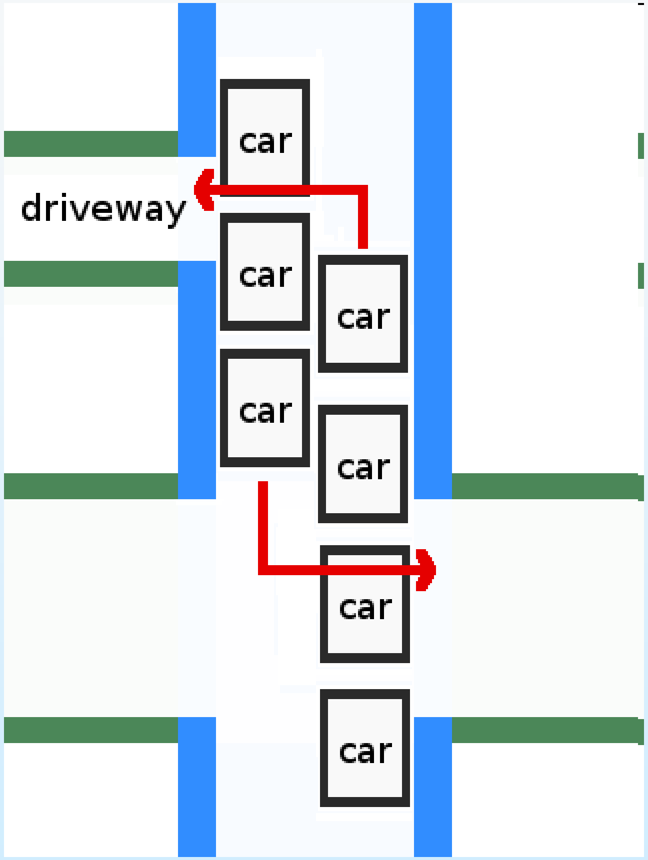
Traffic deadlock



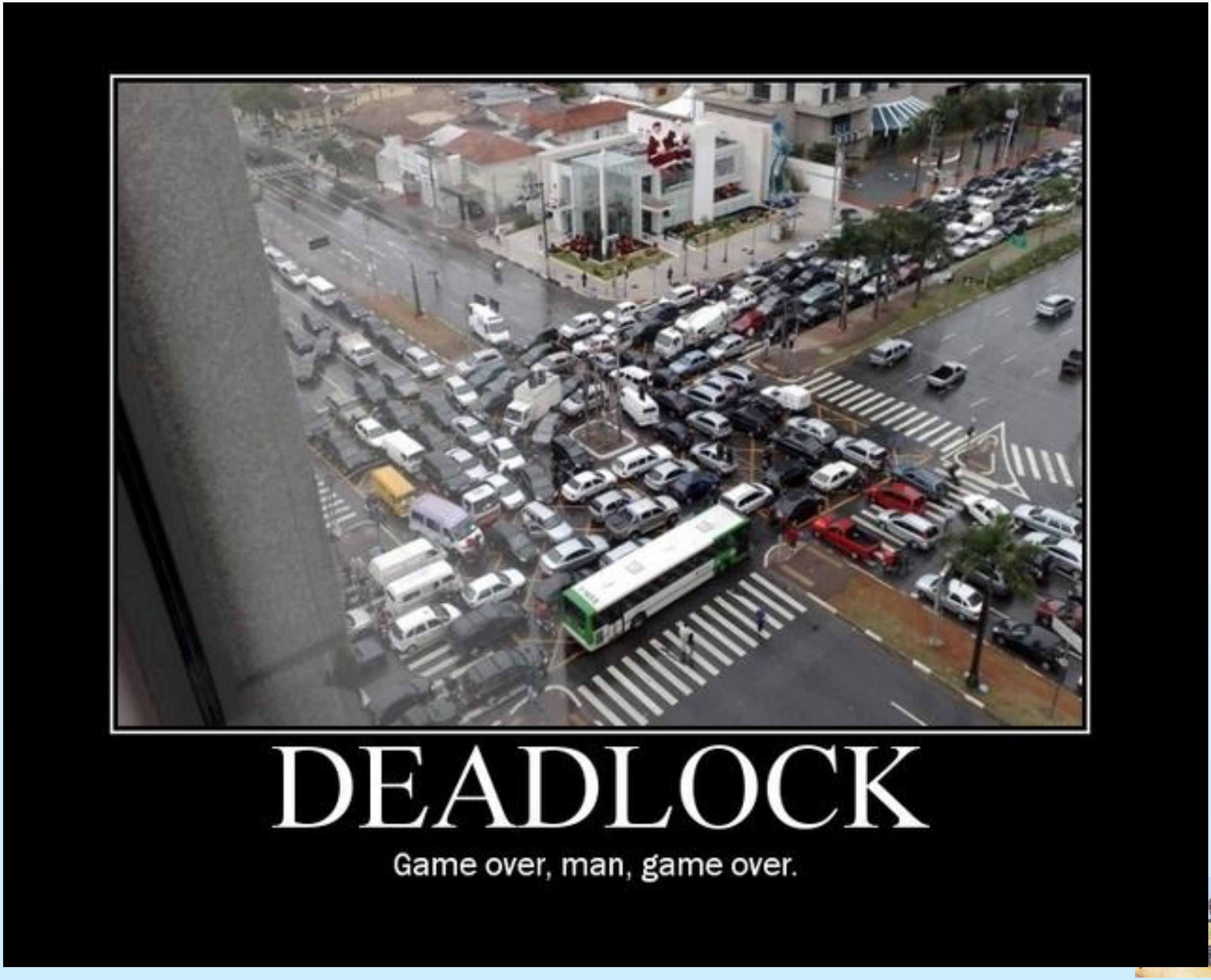
Funny deadlock



Another deadlock



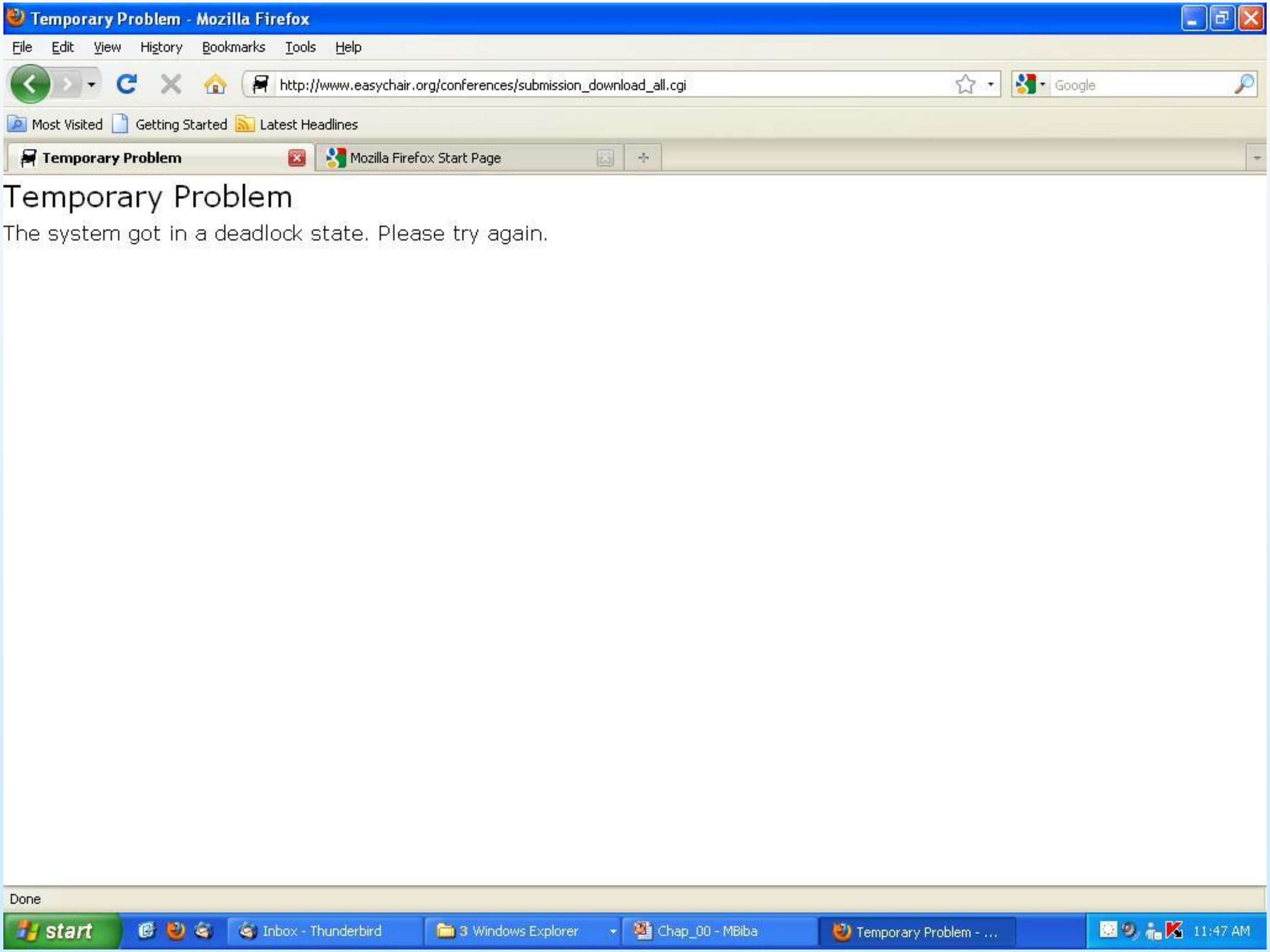
Deadlock: Game over



Where do u go now?

http://go.funpic.hu

Good morning to you!



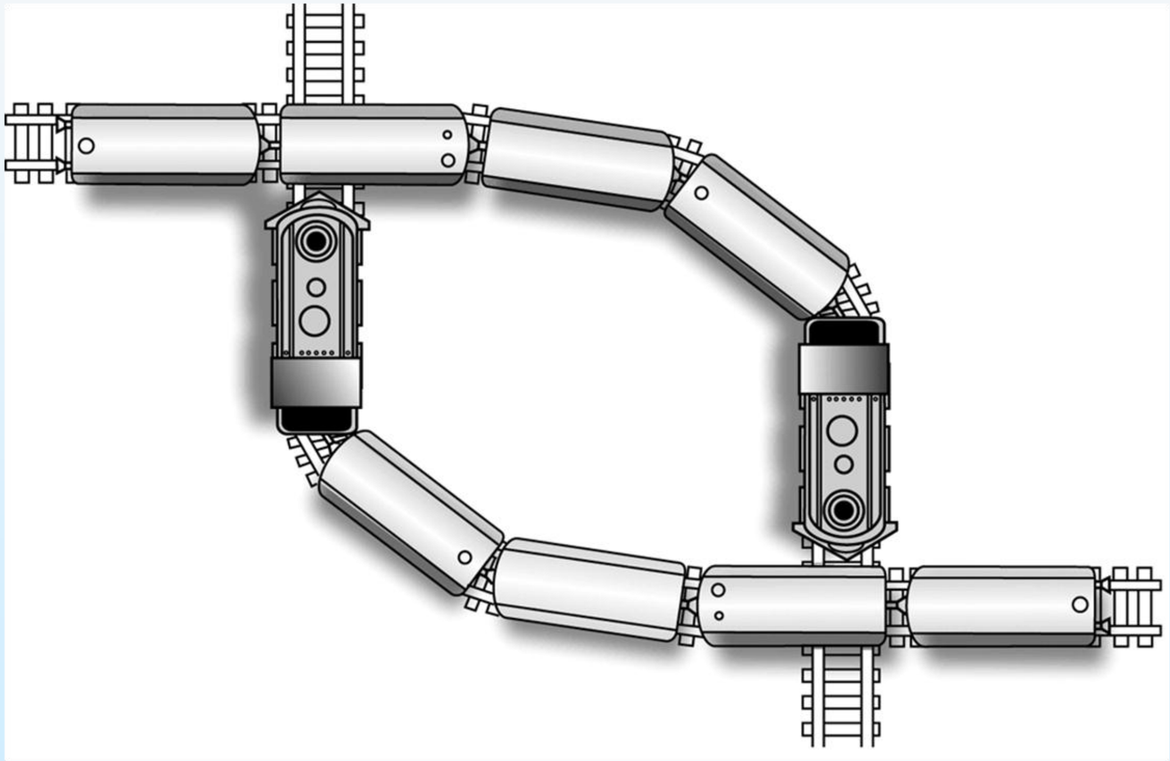
Temporary Problem

The system got in a deadlock state. Please try again.

Deadlocks

- In a multiprogramming environment **several** processes may compete for a **finite** number of resources.
- A process requests resources; and if the resources are **not available** at that time the process enters a **waiting state**.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by **other waiting processes**. This situation is called a **deadlock.**
- The best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part:
  - "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.“ ☺

Train Deadlock



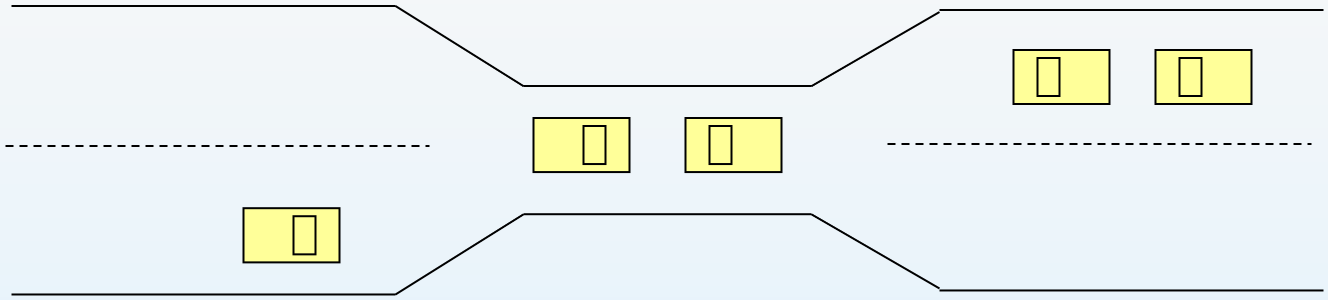
Resources

- A system consists of a **finite** number of resources to be distributed among competing processes.
- The resources are partitioned into **several types**, each consisting of some number of identical **instances**.
  - Memory *space,* CPU cycles, files, and I/O devices (such as printers and DVD drives)
  - If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type *printer* may have five instances.

The Deadlock Problem

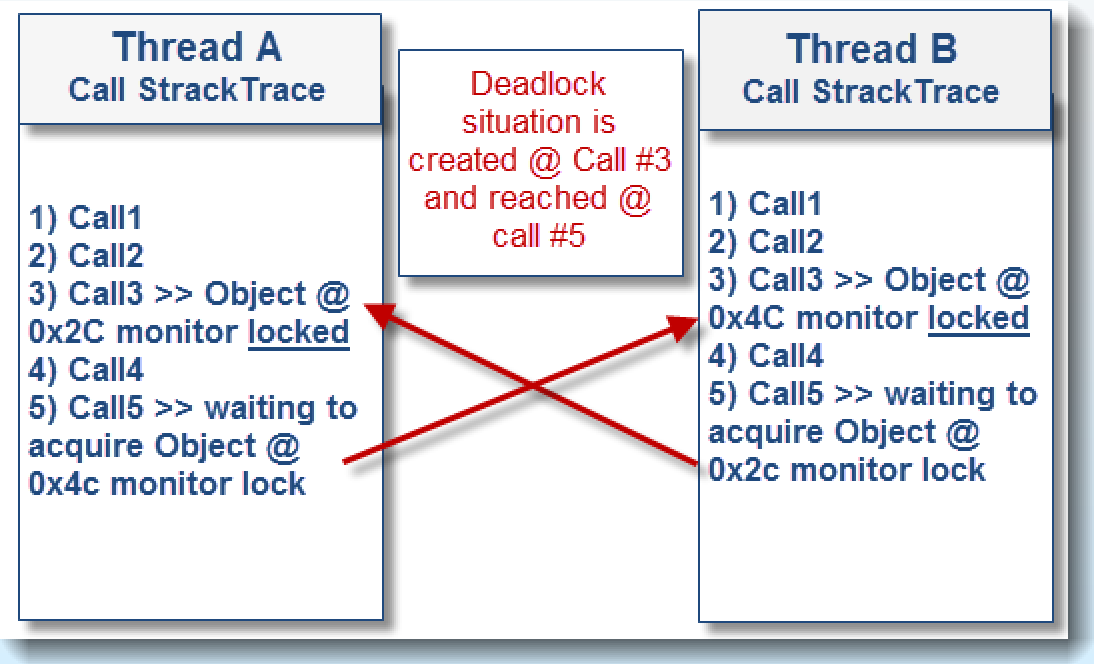
- **A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.**
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.

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.

Threads in deadlock



Thread A
Call StrackTrace

1) Call1
2) Call2
3) Call3 >> Object @ 0x2C monitor <u>locked</u>
4) Call4
5) Call5 >> waiting to acquire Object @ 0x4c monitor lock

Deadlock situation is created @ Call #3 and reached @ call #5

Thread B
Call StrackTrace

1) Call1
2) Call2
3) Call3 >> Object @ 0x4C monitor <u>locked</u>
4) Call4
5) Call5 >> waiting to acquire Object @ 0x2c monitor lock

Chapter 7: Deadlocks

- The Deadlock Problem
- **System Model**
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from 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**. If the request **cannot be granted** immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
  - **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
  - **Release**. The process releases the resource.
  - The request and release of resources are **system calls**.
    - Examples are the request() and release() device, open() and close() file, and allocate() and free() memory system calls.

Processes in deadlock

- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
  - The events with which we are mainly concerned here are **resource acquisition and release**.
  - The resources may be either
    - **physical resources** (for example, printers, tape drives, memory space, and CPU cycles) or
    - **logical resources** (for example, files, semaphores, and monitors).
  - However, other types of events may result in deadlocks.

CD-RW deadlock

- To illustrate a deadlock state, consider a system with **three** CD-RW drives.
- Suppose each of three processes holds one of these CD-RW drives.
  - If each process now **requests another drive**, the three processes will be in a deadlock state.
  - Each is waiting for the event "CD-RW is released," which can be caused only by one of the other waiting processes.
  - The CD-RW example is a deadlock involving the same resource type.

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- **Deadlock Characterization**
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released **only voluntarily** by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ 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.

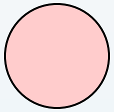
Resource-Allocation Graph

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

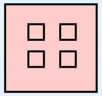
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

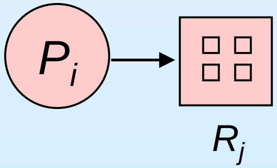
- Process



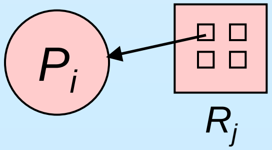
- Resource Type with 4 instances



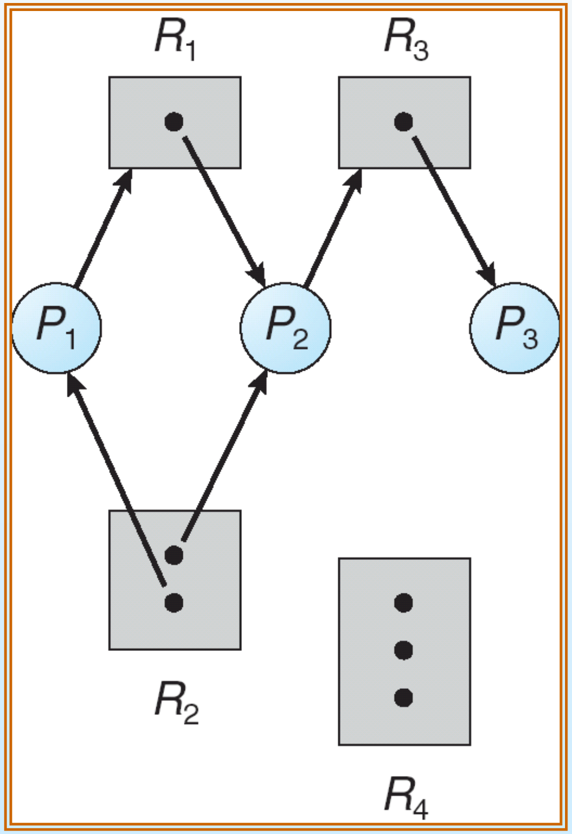
- P_i requests instance of R_j



- P_i is holding an instance of R_j



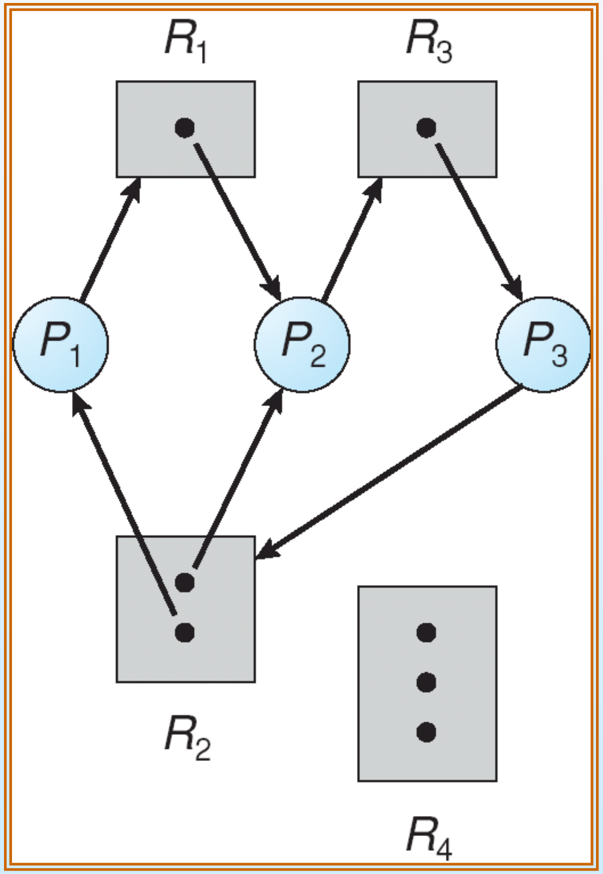
Example of a Resource Allocation Graph



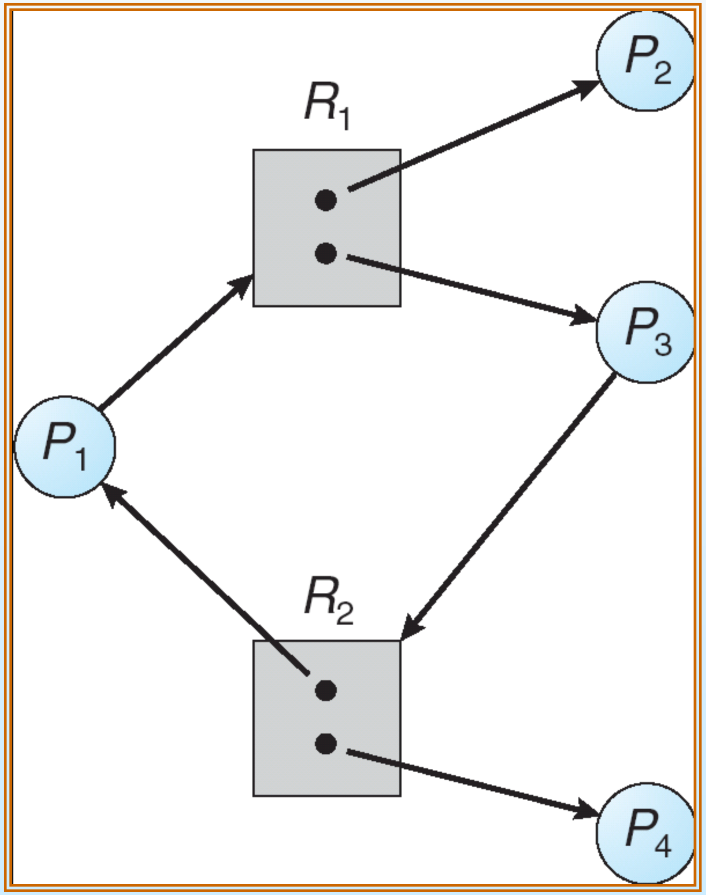
Basic Facts

- 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**.

Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- **Methods for Handling Deadlocks**
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Methods for Handling Deadlocks

- Ensure that the system **will never enter** a deadlock state.
- Allow the system to **enter** a deadlock state and then **recover**.
- Ignore the problem and pretend that deadlocks **never occur** in the system

From deadlock to restart

- If a system neither ensures that a deadlock will **never occur** nor provides a mechanism for deadlock **detection** and recovery, then we may arrive at a situation where the system **is in a deadlocked state yet has no way of recognizing what has happened**.
- In this case the undetected deadlock will result in deterioration of the system's performance:
  - because resources are being held by processes that **cannot run** and
  - because more and more processes, as they make requests for resources, will enter a **deadlocked state**.
- Eventually, the system will stop functioning and will need to be **restarted** manually.

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- **Deadlock Prevention**
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Deadlock Prevention

- For a deadlock to occur, each of the four necessary conditions must hold.
- By **ensuring** that at least one of these conditions **cannot hold**, *we* can *prevent* the occurrence of a deadlock.
  - Deadlock-prevention algorithms, prevent deadlocks by **restraining how requests can be made**.

Mutual Exclusion

- **Mutual Exclusion** – not required for sharable resources; **must hold for nonsharable resources**.
  - For example, a printer cannot be **simultaneously shared** by several processes.
  - Read-only files are a good example of a **sharable resource**. If several processes attempt to open a read-only file at the same time they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.
- **In general! however we cannot prevent deadlocks by denying the mutual-exclusion condition**, because some resources are intrinsically non sharable.

Hold and Wait

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - **Protocol 1:** Require process to request and be allocated all its resources before it begins execution, or
  - **Protocol 2:** Allow process to request resources only when the process has none.

Hold and Wait

- Example: consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.
  - If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. **It will hold the printer for its entire execution, even though it needs the printer only at the end.**
  - The second method allows the process to **request initially only** the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must **then again request** the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.
- Disadvantages: Low **resource utilization** and **possible starvation**.

No Preemption

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then **all resources currently being held are released**.
  - **Preempted resources** are added to the list of resources for which the process is waiting.
  - Process will be **restarted** only when it can regain its old resources, as well as the new ones that it is requesting.
- **Alternatively**, if a process requests some resources, we first check whether they are available.
  - If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is **waiting** for additional resources.
  - If so, we **preempt the desired resources from the waiting process** and allocate them to the requesting process.

Circular wait

- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that **each process requests resources in an increasing order of enumeration**.
- For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function *F* might be defined as follows:
  - *F* (tape drive) = 1
  - *F* (disk drive) = 5
  - *F* (printer) = 12
- **Using the function F, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.**
- Alternatively, we can require that, whenever a process requests an instance of resource type *Rj,* it has **released** any resources *Ri* such that $F(Ri) >= F(Rj)$.

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- **Deadlock Avoidance**
- Deadlock Detection
- Recovery from Deadlock

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.

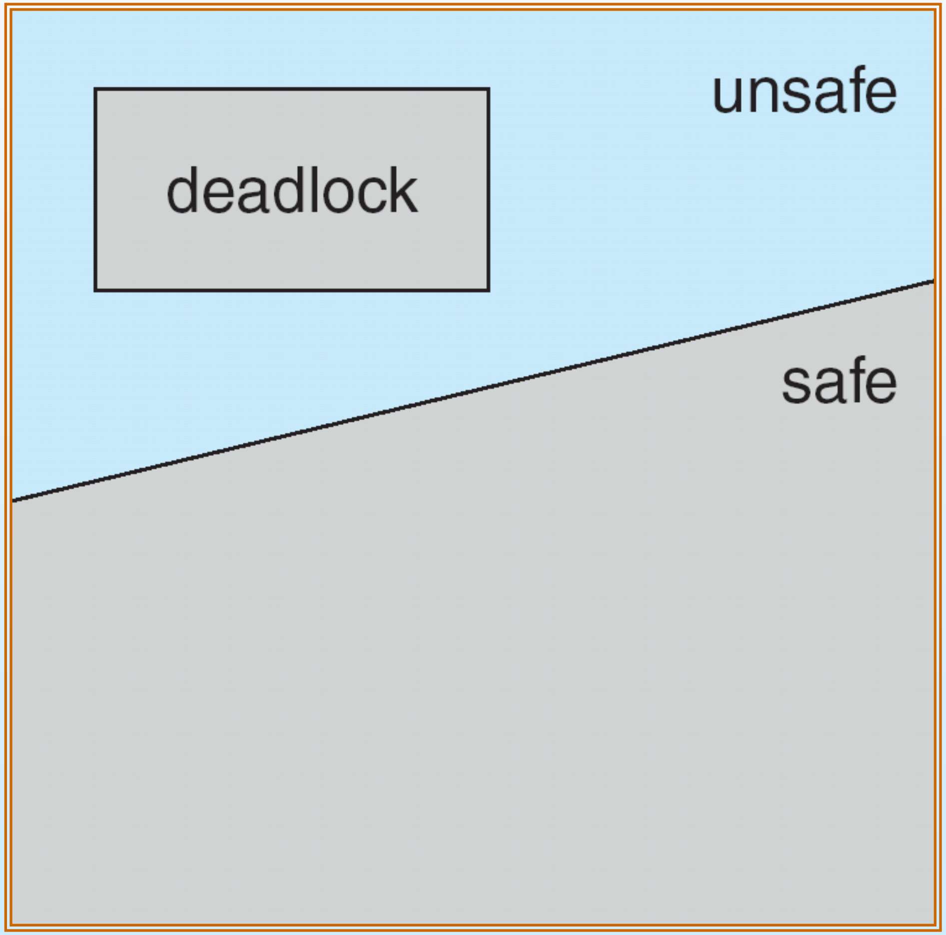
Safe State

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

Basic Facts

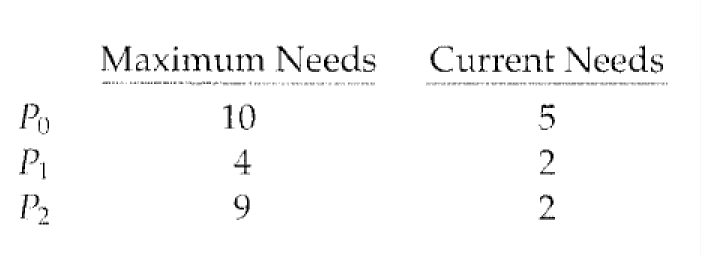
- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



Example

- Consider a system with 12 magnetic tape drives and three processes: *P0, P1,* and *P2.* Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process *P2* may need up to 9 tape drives.
- Suppose that, at time T0 we have:



	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- At time T0, the system is in a safe state. The sequence < P1, P0, *P2*> satisfies the safety condition.
- The system can go from a safe state to an unsafe state. Suppose that, **at time T1, P2 requests and is allocated one more tape drive.**
  - The system is no longer in a safe state.
  - Only process *P1* can be allocated all its tape drives. Suppose it returns all them and we have 4 tapes free!!!
  - P0 may request other 5 tapes and wait
  - P2 may request other 6 tapes and wait

Avoidance algorithms

- Single instance of a resource type.
  - **Use a resource-allocation graph**

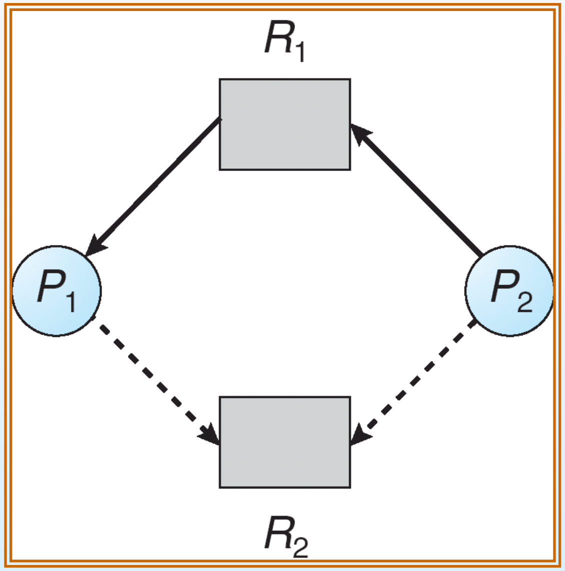
- Multiple instances of a resource type.
  - **Use the banker's algorithm**

Resource-Allocation Graph Scheme

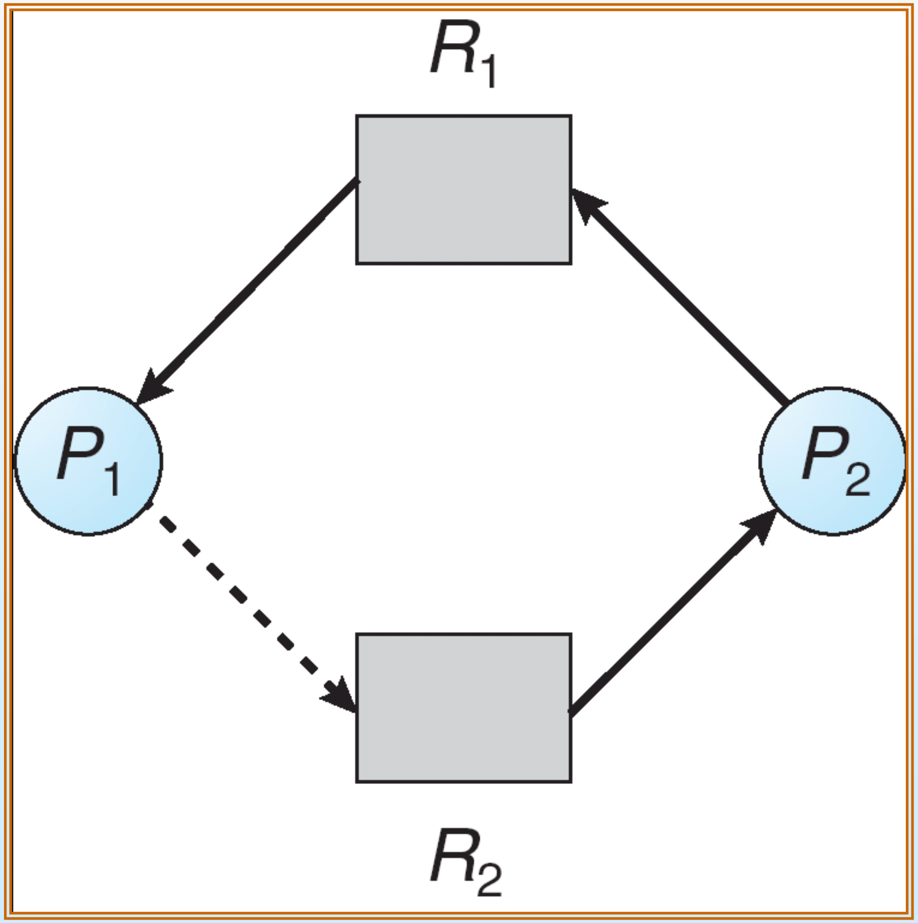
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j **may request** resource R_j; represented by a dashed line.
- **Claim edge converts to request edge** when a process requests a resource.
- **Request edge converted to an assignment edge** when the resource is allocated to the process.
- 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

- Suppose that process Pi requests resource *Rj.*
- The request can be granted only if converting the request edge *Pi -> Rj* to an assignment edge *Rj -> Pi* **does not result in the formation of a cycle in the resource-allocation graph.**
- Note that we check for safety by using a **cycle-detection algorithm**.
- An algorithm for detecting a cycle in this graph requires an order of $\boldsymbol{n^2}$ operations, where *n* is the number of processes in the system.



Unsafe State In Resource-Allocation Graph



Suppose that P2 requests *R2.* Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph A cycle indicates that the system is in an unsafe state.

If *P1* requests *R2,* and *P2* requests *R1* , then a deadlock will occur.

Banker's Algorithm (Djikstra)

- 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**.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers. ☺

Data Structures for the Banker's Algorithm

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

- ***Available***: Vector of length *m*. If Available[*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_i may request at most *k* instances of resource type R_j.
- ***Allocation***: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of $R_{j.}$
- ***Need***: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.

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

Safety Algorithm

- To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that X <= Y if and only if X[i] <= *Y[i]* for all i = 1, 2, ... , 11. For example, if X (1,7,3,2) and Y = (0,3,2,1), then Y < X. Y < X if Y < X and Y /= X.

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

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

- **This algorithm may require an order of m x n^2 operations to determine whether a state is safe.**

Resource-Request Algorithm for Process P_i

- We now describe the algorithm which determines if requests can be safely granted.
- *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 $\boldsymbol{Request_i} \leq \boldsymbol{Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $\boldsymbol{Request_i} \leq \boldsymbol{Available}$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. **Pretend** to allocate requested resources to P_i 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 Pi.*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), 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 (Cont.)

- 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: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ 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 1	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? **No. Resources not available.**
- Can request for (0,2,0) by P_0 be granted? **No. We go in unsafe state.**

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- **Deadlock Detection**
- Recovery from Deadlock

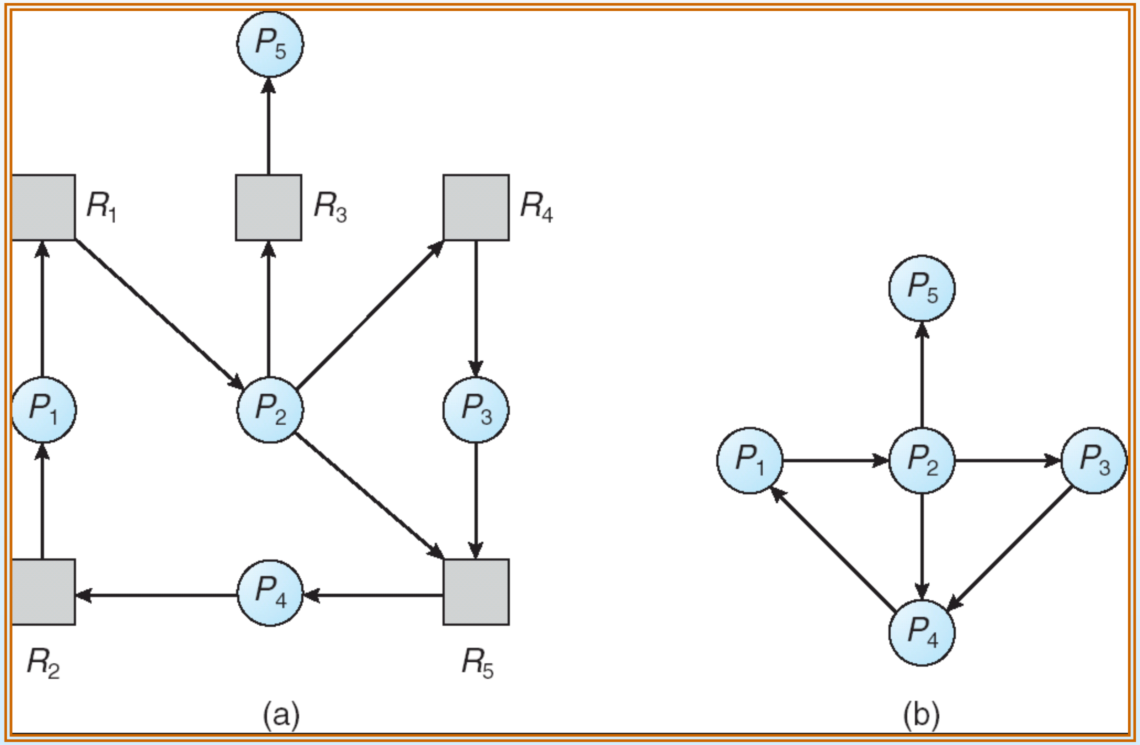
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme
- At this point, however we note that a detection-and-recovery scheme requires **overhead** that includes:
  - not only the **run-time costs** of maintaining the necessary information and executing the detection algorithm
  - but also the **potential losses** inherent in recovering from a deadlock.

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.
- **Complexity: 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, …, *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.

Detection Algorithm (Cont.)

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.**

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- No deadlock: Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	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?

- 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.
- If deadlocks occur **frequently**, then the detection algorithm should be **invoked frequently.**
  - Resources allocated to deadlocked processes will be **idle** until the deadlock can be broken.
  - In addition, the number of involved processes in the deadlock cycle **may grow**.

Detection-Algorithm Usage

- Deadlocks occur only when some process makes a request that **cannot be granted immediately**.
    - This request may be the final request that completes a chain of waiting processes.
    - In the extreme, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
        - In this case, we can identify not only the deadlocked set of processes but also the **specific process that "caused" the deadlock**.

Detection-Algorithm Computational Overhead

- If the deadlock-detection algorithm is invoked for every resource request, this will incur a **considerable overhead** in computation time.
  - A less expensive alternative is simply to invoke the algorithm at less frequent intervals –f or example, once per hour or whenever CPU utilization drops below 40 percent.
  - ***(A deadlock eventually cripples system throughput and causes CPU utilization to drop.)***
  - If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph.
    - In this case, we would generally not be able to tell which of the many deadlocked processes **"caused“ the deadlock**.

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- **Recovery from Deadlock**

Deadlock recovery

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
  - One possibility is to **inform the operator** that a deadlock has occurred and to let him deal with the deadlock manually.
  - Another possibility is to let the system **recover** from the deadlock automatically.
    - There are two options for breaking a deadlock:
      - **Simply to abort one or more processes to break the circular wait.**
      - **The other is to preempt some resources from one or more of the deadlocked processes.**

Recovery from Deadlock: Process Termination

- **Abort all** deadlocked processes.
  - This method clearly will break the deadlock cycle, but at **great expense**; the deadlocked processes may have computed for a long time, and the results of these **partial computations must be discarded** and probably will have to be recomputed later.
- **Abort one process at a time** until the deadlock cycle is eliminated.
  - This method incurs considerable overhead, since, after each process is aborted, **a deadlock-detection algorithm must be invoked** to determine whether any processes are still deadlocked.

Order to abort processes

- Aborting a process may not be easy.
  - If the process was in the midst of **updating** a file, terminating it will leave that file in an incorrect state.
  - Similarly, if the process was in the **midst of printing data** on a printer, the system must reset the printer to a correct state before printing the next job.

Minimum cost of process abortion

- If the partial termination method is used, then we must determine **which deadlocked process (or processes) should be terminated**.
- This determination is a policy decision, **similar to CPU-scheduling decisions**. The question is basically an economic one; we should abort those whose termination will incur the **minimum cost**.
- Unfortunately, the term **minimum cost** is not a precise one.
- 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

- To eliminate deadlocks using resource preemption, we:
  - successively preempt some resources from processes and
  - give these resources to other processes until the deadlock cycle is broken.
- **Selecting a victim** – minimize cost.
- **Rollback** – return to some safe state, restart process from that state.
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor.

Selecting a victim in Resource preemption

- **Selecting a victim.**
  - Which resources and which processes are to be preempted?
  - As in process termination, we must determine **the order of preemption** to minimize cost.
  - Cost factors may include such parameters as:
    - the **number of resources** a deadlocked process is holding
    - **amount of time** the process has thus far consumed during its execution.

Rollback in Resource preemption

- If we preempt a resource from a process, what should be done with that process?
  - Clearly, it cannot continue with its normal execution if it is missing some needed resource.
  - We must **roll back** the process to some **safe state** and restart it from that state.
- Since, in general, it is difficult to determine what a safe state is, the simplest solution is a **total rollback**:
  - Abort the process and then **restart it**.
  - Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to **keep more information** about the state of all running processes.

Starvation in Resource preemption

- How do we ensure that starvation will not occur?
  - That is, how can we guarantee that resources will not always be **preempted from the same process**?
  - In a system where victim selection is based primarily on **cost factors**, it may happen that the same process is always picked as a victim. ☺
- As a result, this process never completes its designated task, a starvation situation that must be dealt with in any practical system.
  - Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor. (**a kind of Aging**)

Readings

- **Silberschatz. Chapter 7.**

This image cannot currently be displayed.

End of Chapter 7

What about deadlocks in distributed systems?

- **A good question for Master students**

Orchestra