Deadlocks

- Process concept √
- Process scheduling √
- Interposes communication x
- Deadlocks
- Threads √

1

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

2.4-2

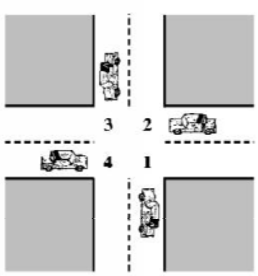
Deadlock-examples

- Process-1 requests the printer, gets it
- Process-2 requests the tape unit, gets it
- Process-1 requests the tape unit, waits
- Process-2 requests the printer, waits deadlocked!
- Hence process-1 and process-2 are deadlocked

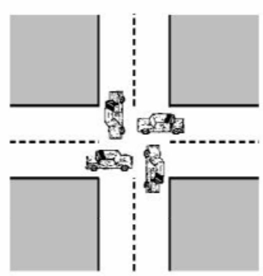
2.4-3

Deadlock-examples (cont--)

Traffic Deadlock Example from Stallings' Text



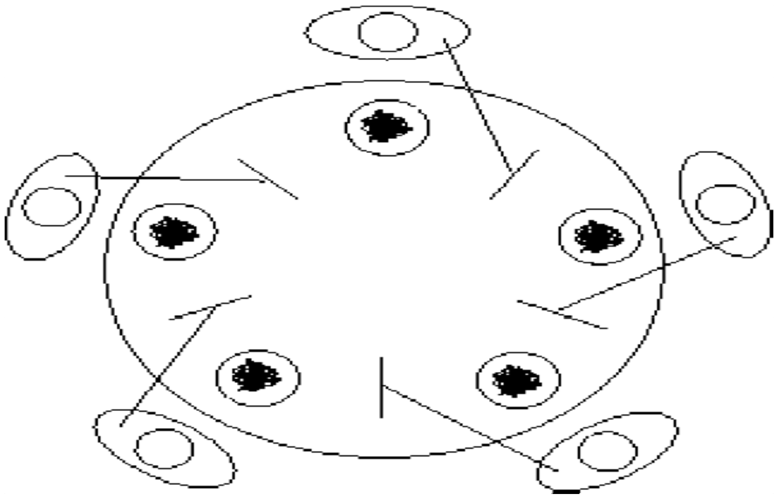
(a) Deadlock possible



(b) Deadlock

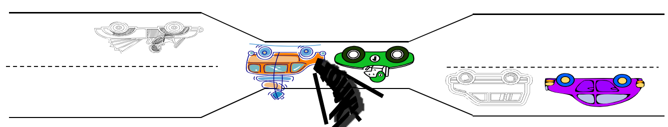
2.4-4

Deadlock-examples (cont--) :Dinning philosophers



2.4-5

Bridge Crossing Example



- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)

2.4-6

Deadlock Definition

- **Formal definition :**
- *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*
- Usually the event is release of a currently held resource
- None of the processes can …

– run

– release resources

– be awakened

2.4-7

Resources

- A resource is something a process(thread) can wait for
- Resources can grouped into:

-Reusable

-Consumable

Deadlock occurs because processes(thread) are in contention for resources

2.4-8

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes

2.4-9

Reusable Resources

- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and Semaphores
- Deadlock occurs if each process holds one dedicated resource and requests another held by another process

2.4-10

Reusable Resources

- Space is available for allocation of 200Kbytes, and the following sequence of events occur
- P1 requests 80Kbytes,P2 requests 70Kbytes,P1 requests 60Kbytes,and P2 requests 80Kbytes
- Deadlock occurs if both processes progress to their second request

2.4-11

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

2.4-12

Necessary Conditions for a Deadlock

- **Mutual exclusion**

– Only one process may use a resource at a time

- **Hold-and-wait**

– A process may hold allocated resources while awaiting assignment of others

2.4-13

Necessary Conditions for a Deadlock

- **No preemption**

– No resource can be forcibly removed form a process holding it

- **Circular wait**

– A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

2.4-14

Necessary Conditions for a Deadlock

Circular Wait: There exists a set of processes: {P1, P2, ..., Pn} such that

- P1 is waiting for a resource held by P2
- P2 is waiting for a resource held by P3
- ...
- Pn-1 is waiting for a resource held by Pn
- Pn is waiting for a resource held by P1

2.4-15

No Deadlock Situation

If you can *prevent at least one of the necessary deadlock conditions* then you won't have a DEADLOCK

2.4-16

Ways of Handling Deadlock

- Deadlock Prevention
- Deadlock Detection
- Deadlock Avoidance
- Deadlock Recovery

2.4-17

Deadlock Prevention

- Remove the possibility of deadlock occurring by denying one of the four necessary conditions:
  - Mutual Exclusion (Can we share everything?)
  - Hold & Wait
  - No preemption
  - Circular Wait

2.4-18

Denying the "Hold & Wait"

- Implementation
  - A process is given its resources on a "ALL or NONE" basis
  - Either a process gets ALL its required resources and proceeds or it gets NONE of them and waits until it can

2.4-19

Denying "No preemption"

- Implementation
  - When a process is refused a resource request, it MUST release all other resources it holds
  - Resources can be removed from a process before it is finished with them

2.4-20

Denying "Circular Wait"

- Implementation
  - Resources are uniquely numbered
  - Processes can only request resources in linear ascending order
  - Thus preventing the circular wait from occurring

2.4-21

Deadlock Avoidance

- Allow the chance of deadlock occur
- But avoid it happening..
- Check whether the next state (change in system) may end up in a deadlock situation

2.4-22

Banker's Problem

Customer	Max. Need	Present Loan	Claim
c1	800	410	390
c2	600	210	390

- Suppose total bank capital is 1000 MTL
- Current cash : 1000- (410+210) = 380 MTL

2.4-23

Dijkstra's Banker's Algorithm

- Definitions
  - Each process has a LOAN, CLAIM, MAXIMUM NEED
    - LOAN: current number of resources held
    - MAXIMUM NEED: total number resources needed to complete
    - CLAIM: = (MAXIMUM - LOAN)

2.4-24

Assumptions

- Establish a LOAN ceiling (MAXIMUM NEED) for each process
  - MAXIMUM NEED < total number of resources available (ie., capital)
- Total loans for a process must be less than or equal to MAXIMUM NEED
- Loaned resources must be returned back in finite time

2.4-25

Algorithm

1. Search for a process with a claim that can satisfied using the current number of remaining resources (ie., tentatively grant the claim)
2. If such a process is found then assume that it will return the loaned resources.
3. Update the number of remaining resources
4. Repeat steps 1-3 for all processes and mark them

2.4-26

- DO NOT GRANT THE CLAIM if at least one process can not be marked.

- Implementation
  - A resource request is only allowed if it results in a SAFE state
  - The system is always maintained in a SAFE state so eventually all requests will be filled

2.4-27

- Advantages
  - It works
  - Allows jobs to proceed when a prevention algorithm wouldn't
- Problems
  - Requires there to be a fixed number of resources
  - What happens if a resource goes down?
  - Does not allow the process to change its Maximum need while processing

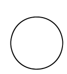
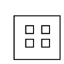
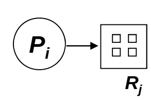
2.4-28

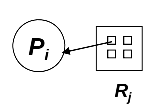
Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph consisting of 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 $P_\mathrm{i} \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Slide 29

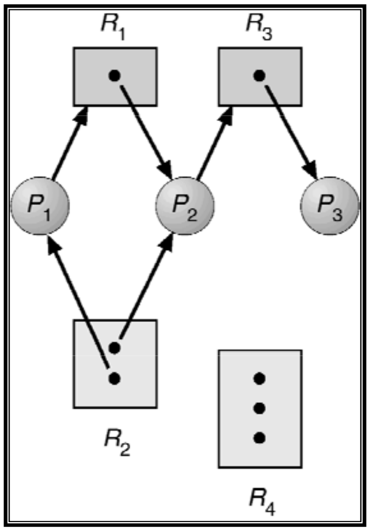
Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Slide 30

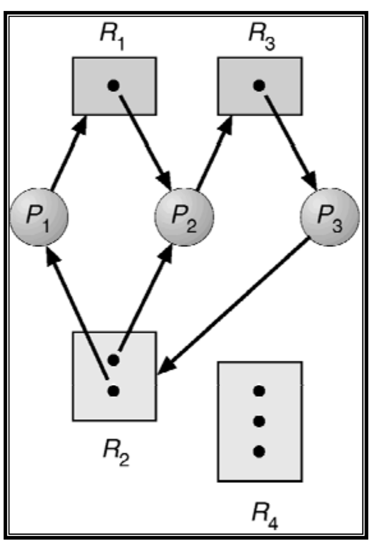
Example of a Resource Allocation Graph



- *Set P contains …*
- *Set R contains …*
- *Set E contains …*
- *How many of instances of each resource are there?*
- *What is each process holding and requesting?*

Slide 31

Resource Allocation Graph With A Deadlock



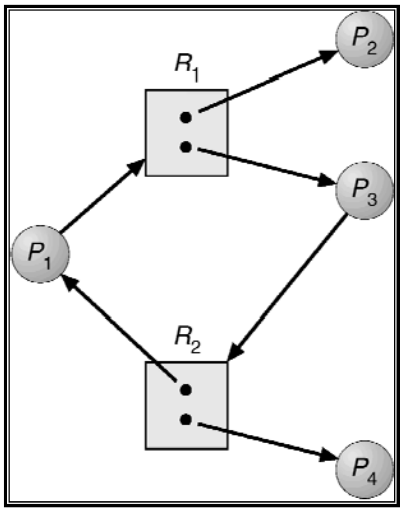
What are the cycles?

P1->R1->P2->R3->P3->R2-> P1

P2->R3->P3->R2->P2

Slide 32

Resource Allocation Graph With A Cycle But No Deadlock



Cycle:

P1->R1->P3->R2->P1

No deadlock as P4 can release R2 which can they be allocated to P3 breaking the cycle

Slide 33

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.

Slide 34

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.

Slide 35

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 safe sequence of all processes.
- Sequence $<P_1, P_2, \ldots, P_n>$ is safe if 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 $j<i$.
  - 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.

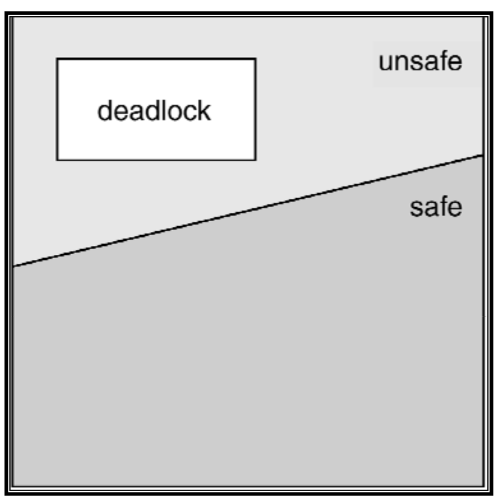
Slide 36

Safe State

- 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.

Slide 37

Safe, Unsafe , Deadlock State



Slide 38

Banker's Algorithm

- Handles multiple instances of resources types that the resource-allocation graph algorithm can not
- 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.

Slide 39

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*].

Slide 40

Safety Algorithm

Tells us whether or not a system is in a safe state

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:
   Work = *Available*
   Finish [i] = *false* for $i = 1,2, \ldots, n$.

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

Slide 41

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:

Available
A B C
3 3 2

	Allocation	*Max*	*Need*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2
P_2	3 0 2	9 0 2	6 0 0
P_3	2 1 1	2 2 2	0 1 1
P_4	0 0 2	4 3 3	4 3 1

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.
- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Slide 42

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 P_i by modifying the state as follows:

   Available = Available = Request;

   $Allocation_i = Allocation_i + Request$;

   $Need_i = Need_i - Request_i$;

   - *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*

Slide 43

Example P_1 Requests (1,0,2)

- Check that $Request_1 \le$ Available (that is, $(1,0,2) \le (3,3,2) \Rightarrow$ *true. Pretend request fulfilled and arrive at this new state:*
- *Available*

  A B C

  2 3 0

	Allocation	*Need*
	A B C	*A B C*
P_0	0 1 0	7 4 3
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. $Request_1$ can be granted
- Can request for (3,3,0) by P_4 be granted? Request not ≤ Available
- Can request for (0,2,0) by P_0 be granted? Available reduced to <2,1,0> and no process can continue

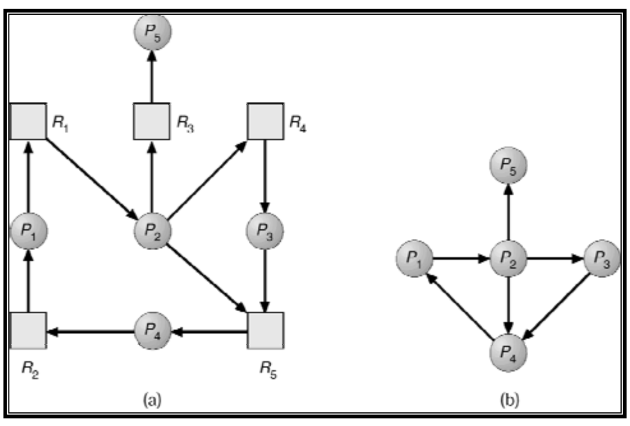
Slide 44

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.
- 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.

Slide 45

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph ***Corresponding wait-for graph***

Slide 46

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.

Slide 47

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.

Slide 48

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:

	Available
	A B C
	0 0 0

	Allocation	*Need*
	A B C	*A B C*
P_0	0 1 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

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in *Finish*[i] = true for all i.

Slide 49

Example (Cont.)

- P_2 requests an additional instance of type C.

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
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.

Slide 50

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.

Slide 51

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.

Slide 52

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.

Slide 53