OPERATING SYSTEMS

Deadlocks

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Deadlock Avoidance

Suresh Jamadagni
Department of Computer Science

Deadlock Avoidance

- Deadlock-prevention algorithms, prevent deadlocks by limiting how requests can be made.
- The limits ensure that at least one of the necessary conditions for deadlock cannot occur.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.
- With this additional knowledge of complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

Deadlock Avoidance

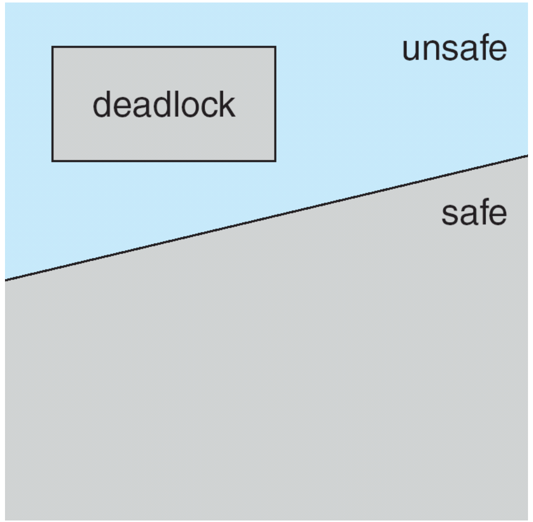
- The simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need.
- Given this apriori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State

- A state is ***safe*** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
- A sequence of processes $<P1, P2, ..., Pn>$ is a safe sequence for the current allocation state if, for each Pi , the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with $j < i$.
- In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished.
- When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
- When Pi terminates, $Pi+1$ can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be ***unsafe***

Safe, Unsafe and Deadlock States

- A safe state is not a deadlocked state.
- Conversely, a deadlock state is an unsafe state.
- Not all unsafe states are deadlocks.
- An unsafe state may lead to a deadlock.
- As long as the state is safe, the OS can avoid unsafe states.
- In an unsafe state, the OS cannot prevent processes from requesting resources in such a way that a deadlock occurs.
- The behavior of processes controls unsafe states.
- If a system is in safe state → no deadlocks
- If a system is in unsafe state → possibility of deadlock
- Goal for Avoidance → ensure that a system will never enter an unsafe state.



Example - Safe, Unsafe and Deadlock States

- Consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2.
- Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.
- Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives. (Thus, there are three free tape drives.)

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

- At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition. Process *P*1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *P*0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P*2 can get all its tape drives and return them
- At time *t*1, process *P*2 requests and is allocated one more tape drive. The system is no longer in a safe state.
- Since process *P*0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process *P*2 may request six additional tape drives and have to wait, resulting in a deadlock

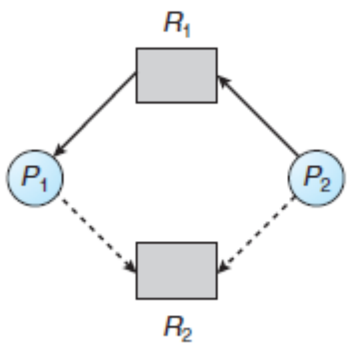
Deadlock avoidance algorithms

- Avoidance algorithms ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.
- Thus, resource utilization may not be optimal

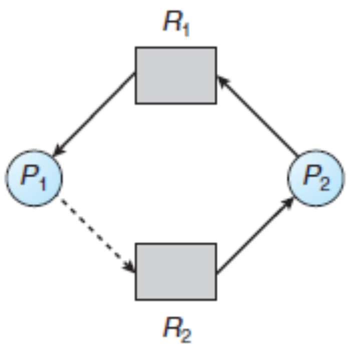
Resource Allocation Graph algorithm

- In addition to the request and assignment edges in a resource allocation graph, we introduce a new type of edge, called a **claim edge**
- A claim edge *Pi* → *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process *Pi* requests resource *Rj* , the claim edge *Pi* → *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi* , the assignment edge *Rj* → *Pi* is reconverted to a claim edge *Pi* → *Rj*
- The resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
- Now 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.

Resource Allocation Graph algorithm example



Resource-allocation graph for deadlock avoidance.



An unsafe state in a resource-allocation graph.

- Consider the above resource-allocation graph.
- Suppose that *P*2 requests *R*2. Although *R*2 is currently free, we cannot allocate it to *P*2, since this action will create a cycle in the graph below.
- A cycle indicates that the system is in an unsafe state.
- If *P*1 requests *R*2, then a deadlock will occur.

Banker's algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.
- We need the following data structures, where n is the number of processes in the system and m is the number of resource types

Banker's algorithm data structures

- **Available –** A vector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] equals *k,* then *k* instances of resource type *Rj* are available.
- **Max**. - An $n \times m$ matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k,* then process *Pi* may request at most *k* instances of resource type *Rj* .
- **Allocation** - An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k,* then process *Pi* is currently allocated *k* instances of resource type *Rj* .
- **Need** - An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] equals *k,* then process *Pi* may need *k* more instances of resource type *Rj* to complete its task.
- ***Need***[*i*][*j*] = ***Max***[*i*][*j*] − ***Allocation***[*i*][*j*].
- Treat each row in the matrices ***Allocation*** and ***Need*** as vectorsThe vector ***Allocationi*** specifies the resources currently allocated to process *Pi* ; the vector ***Needi*** specifies the additional resources that process *Pi* may still request to complete its task
- These data structures vary over time in both size and value.

Banker's algorithm for determining 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*** and ***Finish***[i] = ***false*** for $i = 0, 1, ..., n - 1$.

2. Find an index 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 $m \times n^2$ operations to determine whether a state is safe.

Banker's algorithm for determining whether requests can be safely granted

Let ***Request**i* be the request vector for process *Pi* . If ***Request**i* [*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj* .

When a request for resources is made by process *Pi* , the following actions are taken:

1. If ***Request**i* ≤ ***Need**i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Request**i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

    Available = ***Available***–***Request**i* ;

    ***Allocation**i* = ***Allocation**i* + ***Request**i* ;

    Need**i* = ***Need**i* –Request**i* ;

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources.

If the new state is unsafe, then *Pi* must wait for ***Request**i* **,** and the old resource-allocation state is restored.

Banker’s algorithm example

- Consider a system with five processes *P*0 through *P*4 and three resource types *A, B,* and *C.*
- Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances.
- Suppose that, at time *T*0, the following snapshot of the system has been taken:

	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	

- Sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.
- A request for (3,3,0) by *P*4 cannot be granted, since the resources are not available.
- A request for (0,2,0) by *P*0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering
sureshjamadagni@pes.edu