BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Deadlock Handling

Notice: The slides for this lecture have been largely based on those accompanying an earlier edition of the course text *Operating Systems Concepts, 9th ed.*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source. Revised by X.M. based on Professor Perrone's notes.

Potential Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Why "potential"?

The code may not cause deadlock if one thread grabs both locks before the other.

If both threads hold on the one lock before trying the second lock, a deadlock is ensured.

http://www.eg.bucknell.edu/~cs315/Fall13/code/deadlock/deadlock.c

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. (*recover*)
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention

Constrain the ways requests can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

Deadlock Prevention

Constrain the ways request can be made.

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

Prevention and Avoidance

- In deadlock prevention, we try to break one of the four necessary conditions for deadlock.
  - Doing so doesn't require any extra knowledge, we just examine the current status of the resources and requests to make a decision and take actions
- If extra knowledge is available, or extra constrains are allowed, we can avoid the deadlocks.

Deadlock Avoidance

The system has additional *a priori* information.

- The 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 States

- Sequence **<P_1, P_2, …, P_n>** is ***safe*** if for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus the 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.
- The system is in a ***safe state*** if there exists a safe sequence for all processes.
- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a **safe state**.

Safe Sequence and State Example

Table: Resource allocation

	Maximum Needs	Allocated
P0	10	5
P1	4	2
P2	5	2

Table: Total resources

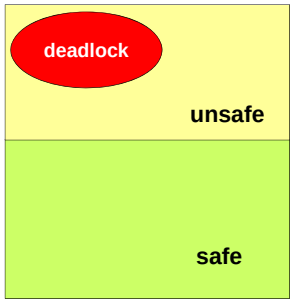
	Total Count
R	12

Safe sequence 1: [<p1,2>, <p0, 5>, <p2,3>]
Safe sequence 2: [<p2,3>, <p0, 5>, <p1,2>]
More safe sequences?

Basic Facts

- If a system is in a safe state there can be no deadlock.
- If a system is in unsafe state, there exists the **possibility** of deadlock.
- **Avoidance** strategies ensure that a system will never enter an unsafe state.

Safe, Unsafe, and Deadlock States

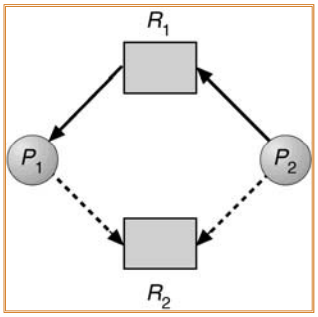


Resource-Allocation Graph Algorithm

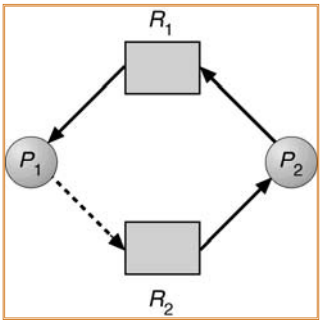
Goal: prevent the system from entering an unsafe state.

- Applicable only when there is a single instance of each resource type.
- *Claim edge* $P_i \rightarrow R_j$ indicates 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.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed ***a priori*** in the system.
- If there is no cycle as the result of allocation, the system is safe.

Resource-Allocation Graph for Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Applicable when there are multiple instances of each resource type.
- In a bank, the cash must never be allocated in a way such that it cannot satisfy the need of **all its customers**.
- Each process must state a priori the maximum number of instances of each kind of resource that it will ever need.
- 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.

Banker's Algorithm: Data Structures

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

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:
   Work = Available
   Finish [*i*] = *false* for *i* = 0,1,2,3, …, *n-1*.
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.

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:
   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;
   - *If safe ⇒ the resources are allocated to P_i. (run safe algorithm)*
   - *If unsafe ⇒ P_i 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 (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 (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) (Cont.)

- Check that Request $\leq$ Available (that is, (1,0,2) $\leq$ (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 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 <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?