OPERATING SYSTEMS

Deadlocks

Suresh Jamadagni
Department of Computer Science

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 Prevention, Deadlock Example

Suresh Jamadagni
Department of Computer Science

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
  - Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  - Allow the system to enter into deadlocked state, detect it, and recover.
  - Ignore the problem all together and pretend that deadlocks never occur in the system.

4 Necessary conditions for deadlock to occur

- **Mutual Exclusion**
  - At least one resource must be non sharable
    - Ex. printers and tape drives, mutex locks
  - Sharable resources do not require mutual exclusion
    - Ex . Read-only files
- **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 allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **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** –
  - Each resource will be assigned with a numerical number.
  - A process can request the resources increasing/decreasing order of numbering.

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Resources={R1,R2,.....Rm}
- Resources are assigned unique number
- Each process request resource in increasing order enumeration
- we define a one-to-one function F: R→N, where N is the set of natural numbers
- Protocol 1: Process makes request for Ri and then for Rj. Resources Rj request is allowed if and only if F(Rj)>F(Ri).
- Protocol 2: Process requesting an instance of resource type Rj, must have released any resource Ri , such that F(Ri)>=F(Rj).
- If these protocols are used then circular wait will not exist.

If these two protocols are used, then the circular-wait condition cannot hold.

Proof by contradiction:

- We can demonstrate this fact that by assuming that circular wait condition cannot hold
- Consider set of processes P={P0,P1,………Pn}
- Let us consider process P0 is waiting for resource held by P1, P1 waiting for P2,……..Pn-1 is waiting resource held by Pn, Pn is waiting for resources held by P0.
- Generalizing this, Process Pi is waiting for resources Ri, Ri is held by Pi+1 and it is making request for Ri+1
- We must have F(Ri)<F(Ri+1)
- But this condition means that $F(R0) < F(R1) < \ldots < F(Rn) < F(R0)$.
- By transitivity, $F(R0) < F(R0)$, which is impossible
- Therefore no circular wait.

Circular Wait

Example

- F(tape drive)=1
- F(disk drive)=5
- F(printer)=12
- F(tape drive)<F(printer)

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Invalidating the circular wait condition is most common.
- Assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

  first_mutex = 1

  second_mutex = 5

  code for **thread_two** could not be written as follows:

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);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount) {
  mutex lock1, lock2;
  lock1 = get_lock(from);
  lock2 = get_lock(to);
  acquire(lock1);
    acquire(lock2);
      withdraw(from, amount);
      deposit(to, amount);
    release(lock2);
  release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

OPERATING SYSTEMS

Deadlock Detection, Algorithm

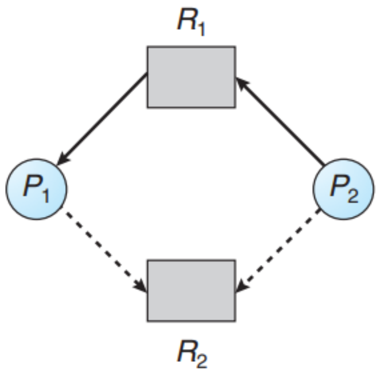
Chandravva Hebbi
Department of Computer Science

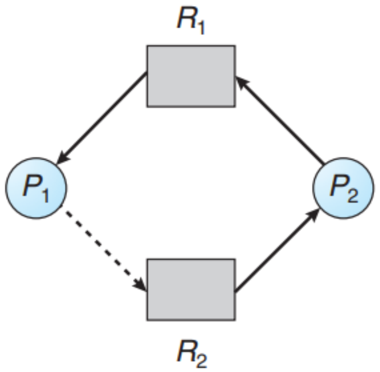
Deadlock Detection

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

Single Instance of Each Resource Type

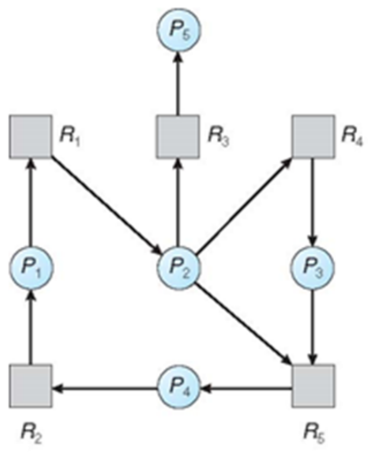
- In a resource allocation graph, a claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge is represented in the graph by a dashed line.
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, 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





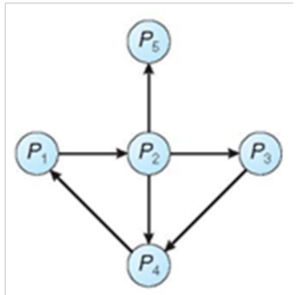
Resource-Allocation Graph and Wait-for Graph

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q
- A **deadlock exists** in the system if and only if the **wait-for graph contains a cycle**.
- 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



(a)

Resource-Allocation Graph



(b)

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. If Available[j] = k, then k instances of resource type R_j are available

- **Max**: An n × 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**: An ***n* x *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

- **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 $\boldsymbol{Allocation_i \neq 0}$, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:

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

   (b) $\boldsymbol{Request_i \leq Work}$

   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 ≤ ***i*** ≤ ***n***, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then ***P$_i$*** is deadlocked. If **Finish[i] == true** for all i, then the system is in a safe state

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	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 <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example of Detection Algorithm (Cont.)

- $\boldsymbol{P_2}$ 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 $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

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.

THANK YOU

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