



東南大學  
SOUTHEAST UNIVERSITY

# OPERATING SYSTEM CONCEPTS

.....

## Chapter 7. Deadlocks

A/Prof. Kai Dong



## Warm-up

### *Bugs in Modern Applications*

| Application | What is does    | Non-Deadlock | Deadlock |
|-------------|-----------------|--------------|----------|
| MySQL       | Database Server | 14           | 9        |
| Apache      | Web Server      | 13           | 4        |
| Mozilla     | Web Browser     | 41           | 16       |
| OpenOffice  | Office Suite    | 6            | 2        |
| Total       |                 | 74           | 31       |

- [1] han Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, “Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics”, ASPLOS’08, March 2008, Seattle, Washington



## Warm-up

### Non-deadlock Bugs

- A large fraction (97%) of non-deadlock bugs are either **atomicity violation** or **order violation**.
- Atomicity violation bugs

```
1  /* Thread 1 */
2  if (thd->proc_info) {
3      ...
4      fputs(thd->proc_info, ...);
5      ...
6  }
```

```
1  /* Thread 2 */
2  thd->proc_info = NULL;
3
4
5
6  //
```

- Order violation Bugs

```
1  /* Thread 1 */
2  void init() {
3      ...
4      mThread = PR_CreateThread(mMain,
5                               ...);
6      ...
7  }
```

```
1  /* Thread 2 */
2  void mMain(...) {
3      ...
4      mState = mThread->State;
5      ...
6  }
7  //
```



## Warm-up

### Deadlock Bugs

- Why Do Deadlocks Occur?
  - Complex dependencies arise between components in large code bases.
    - » The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system.
  - The nature of encapsulation — Modularity does not mesh well with locking.
    - » The Java Vector class and the method `AddAll()`. The routine acquires two locks in some arbitrary order. If some other thread calls `v2.AddAll(v1)`, a deadlock can happen.

```
1 Vector v1, v2;  
2 v1.AddAll(v2);
```

# Objectives



- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system



# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection
7. Recovery from Deadlock



# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection
7. Recovery from Deadlock



## System Model

- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$ 
  - CPU cycles, memory space, I/O devices
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - *request*
  - *use*
  - *release*





# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection
7. 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  $\{P_0, P_1, \dots, P_n\}$  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  $P_2$ ,  $\dots$ ,  $P_{n-1}$  is waiting for a resource that is held by  $P_n$ , and  $P_n$  is waiting for a resource that is held by  $P_0$ .
- Necessary, but NOT sufficient condition(s).



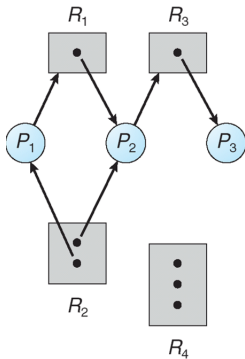
# Deadlock Characterization

## Resource Allocation Graph

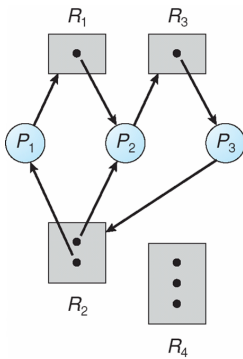
- How to describe a deadlock?
- A resource-allocation graph, which contains a set of vertices  $V$  and a set of edges  $E$ .
  - Two types of  $V$ 
    - » Processes —  $P = \{P_1, P_2, \dots, P_n\}$
    - » Resources —  $R = \{R_1, R_2, \dots, R_m\}$
  - Two types of  $E$ 
    - » Request edge — directed edge  $P_i \rightarrow R_j$
    - » Assignment edge — directed edge  $R_j \rightarrow P_i$

# Deadlock Characterization

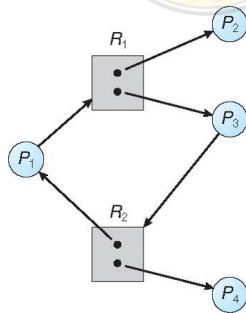
## Resource Allocation Graph (contd.)



(a)



(b)



(c)



# Deadlock Characterization

## Resource Allocation Graph (contd.)

- Some basic facts
  - If graph contains no cycles  $\Rightarrow$  no deadlock
  - If graph contains a cycle  $\Rightarrow$ 
    - » if only one instance per resource type, then deadlock
    - » if several instances per resource type, possibility of deadlock



# Contents

1. System Model
2. Deadlock Characterization
- 3. Methods for Handling Deadlocks**
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection
7. Recovery from Deadlock



## Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state
  - **Deadlock prevention** — dealing with necessary conditions
  - **Deadlock avoidance** — dealing with safe state
- Allow the system to enter a deadlock state and then recover
  - **Deadlock detection**
  - **Recovery from deadlock**
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX



# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
- 4. Deadlock Prevention**
5. Deadlock Avoidance
6. Deadlock Detection
7. Recovery from Deadlock





# Deadlock Prevention

## Circular Wait

- Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - total ordering
  - partial ordering
- Enforcing lock ordering by lock address

```
1  do_something(mutex_t *m1, mutex_t *m2) {  
2      ...  
3      if (m1 > m2) {  
4          pthread_mutex_lock(m1);  
5          pthread_mutex_lock(m2);  
6      }  
7      else {  
8          pthread_mutex_lock(m2);  
9          pthread_mutex_lock(m1);  
10     }  
11     ...  
12 }  
13  
14 /* An ordering, or hierarchy, does not in itself prevent deadlock */
```



## Deadlock Prevention

### *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
- Note that the solution is problematic:
  - Encapsulation works against us
  - Decreased concurrency



# Deadlock Prevention

## 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.

```
1  top:
2      lock(L1);
3      if (trylock(L2) == -1) {
4          unlock(L1);
5          goto top;
6      }
```

- Note that the solution is problematic:
  - Encapsulation works against us
  - Livelock — It is possible that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks.
    - » Solution is to add a random delay before looping back and trying the entire thing over again.



# Deadlock Prevention

## Mutual Exclusion

- Mutual Exclusion — not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- In general, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable
- Wait-free concurrency

```
1  /* atomically increment a value by a certain amount */
2  int CompareAndSwap(int *address, int expected, int new) {
3      if (*address == expected) {
4          *address = new;
5          return 1;
6      }
7      return 0;
8  }
9  void AtomicIncrement(int *value, int amount) {
10     do {
11         int old = *value;
12         } while (CompareAndSwap(value, old, old + amount) == 0);
13 }
```



# Deadlock Prevention

## Mutual Exclusion (contd.)

```
1  /* inserts at the head of a list */
2  void insert(int value) {
3      node_t *n = malloc(sizeof(node_t));
4      assert(n != NULL);
5      n->value = value;
6      pthread_mutex_lock(listlock);
7      n->next = head;
8      head = n;
9      pthread_mutex_unlock(listlock);
10 }
11
12 /* atomic list insertion: */
13 void insert(int value) {
14     node_t *n = malloc(sizeof(node_t));
15     assert(n != NULL);
16     n->value = value;
17     do {
18         n->next = head;
19     } while (CompareAndSwap(&head, n->next, n) == 0);
20 }
```



# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
- 5. Deadlock Avoidance**
6. Deadlock Detection
7. Recovery from Deadlock



## Deadlock Avoidance

- Deadlock avoidance — via scheduling
  - Assume we have two processors ( $CPU_1$ ,  $CPU_2$ ) and four threads ( $T_1$ ,  $T_2$ ,  $T_3$ ,  $T_4$ ) which must be scheduled upon them. Assume further we know that each thread will grab some locks  $L_1$  or  $L_2$  as follows.

|       | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|-------|-------|-------|-------|-------|
| $L_1$ | yes   | yes   | no    | no    |
| $L_2$ | yes   | yes   | yes   | no    |

| $CPU_1$ | $T_1$ | $T_2$ |  |
|---------|-------|-------|--|
| $CPU_2$ | $T_3$ | $T_4$ |  |



## 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





# Deadlock Avoidance

## Safe State

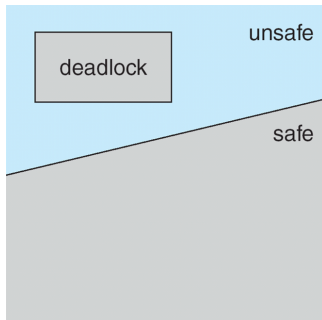
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence  $\langle P_1, P_2, \dots, P_n \rangle$  of ALL the processes in the systems such that for each  $P_i$ , the resources that  $P_i$  can still request can be satisfied by currently available resources + resources held by all the  $P_j$ , with  $j < i$ 
  - 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



## Deadlock Avoidance

### Safe State (contd.)

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





## Deadlock Avoidance

### Safe State (contd.)

- Consider a system with 12 magnetic tape drives and three processes:  $P_0, P_1, P_2$ .

|       | Maximum Needs | Current Needs |
|-------|---------------|---------------|
| $P_0$ | 10            | 5             |
| $P_1$ | 4             | 5             |
| $P_2$ | 9             | 2             |

- At time  $t_0$ , the system is in a safe state. The sequence  $\langle P_1, P_0, P_2 \rangle$  satisfies the safety condition.
- Show by an example that a system can go from a safe state to an unsafe state.
- Suppose that, at time  $t_1$ , process  $P_2$  requests and is allocated one more tape drive. The system is no longer in a safe state.



# Deadlock Avoidance

## *Avoidance Algorithms*

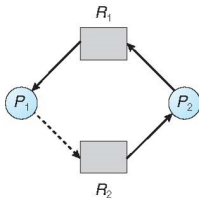
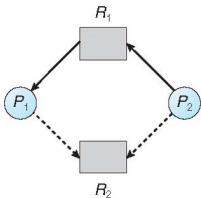
- Single instance of a resource type
  - Use the **resource-allocation-graph algorithm**
- Multiple instances of a resource type
  - Use the **banker's algorithm**



# Deadlock Avoidance

## Resource-Allocation-Graph Algorithm

- Claim edge:  $P_i \dashrightarrow R_j$  indicated that process  $P_i$  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





## Deadlock Avoidance

### *Resource-Allocation-Graph Algorithm (contd.)*

- Suppose that process  $P_i$  requests a resource  $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

# Deadlock Avoidance

## *Banker's Algorithm*



- Multiple instances
- Assumptions:
  - 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



## Deadlock Avoidance

### Banker's Algorithm (contd.)

- $n$  = number of processes
- $m$  = number of resources types.
- **Available**: Vector of length  $m$ . If  $Available[j] = k$ , then there are  $k$  instances of resource type  $R_j$  available
- **Max**:  $n \times m$  matrix. If  $Max[i, j] = k$ , then process  $P_i$  may request at most  $k$  instances of resource type  $R_j$
- **Allocation**:  $n \times m$  matrix. If  $Allocation[i, j] = k$  then  $P_i$  is currently allocated  $k$  instances of  $R_j$
- **Need**:  $n \times m$  matrix. If  $Need[i, j] = k$ , then  $P_i$  may need  $k$  more instances of  $R_j$  to complete its task

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





## Deadlock Avoidance

### Safety Algorithm

1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively.

Initialize:

$Work = Available$

$Finish[i] = false$ , for  $i = 0, 1, \dots, n - 1$

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



## Deadlock Avoidance

### Resource-Request Algorithm

$Request_i$  = 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  $Request_i \leq Need_i$  go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $Request_i \leq Available$ , 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  $\Rightarrow$  the resources are allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored



# Deadlock Avoidance

## In Class Exercise

A computer system has 3 types of resources A, B, and C with different numbers of instances. There are 4 running processes  $P_1, P_2, P_3, P_4$ . The total resources, the resource's *Allocation* and *Max* matrices for the four processes are shown as follows:

| Process | Allocation |   |   | Max |   |   |
|---------|------------|---|---|-----|---|---|
|         | A          | B | C | A   | B | C |
| $P_1$   | 1          | 3 | 1 | 6   | 5 | 3 |
| $P_2$   | 0          | 2 | 2 | 3   | 5 | 3 |
| $P_3$   | 2          | 0 | 0 | 3   | 5 | 2 |
| $P_4$   | 0          | 1 | 3 | 2   | 4 | 3 |
| total   | 6          | 9 | 6 |     |   |   |

1. What are the matrices *Need* and *Available* for the system?
2. Please check if the system is currently deadlocked. Show your steps clearly.
3. At the current state, if  $P_2$  requests additional resources  $[1, 0, 0]$ , can the request be granted without any possible deadlock? Show your steps clearly.
4. At the current state, if  $P_1$  requests additional resources  $[1, 0, 0]$ , can the request be granted without any possible deadlock? Show your steps clearly.



# Deadlock Avoidance

## Key

- $n = 4, m = 3$ , Allocation and Max are already defined.
- Key to Q1: Compute Available and Need

| Process   | Need |   |   |
|-----------|------|---|---|
|           | A    | B | C |
| $P_1$     | 5    | 2 | 2 |
| $P_2$     | 3    | 3 | 1 |
| $P_3$     | 1    | 5 | 2 |
| $P_4$     | 2    | 3 | 0 |
| Available | 3    | 3 | 0 |

- Key to Q3 Q4: Resource-Request Alg.
- Assume the request is granted, use the Safety Alg. to determine whether the system is in safe state.

- Key to Q2: Safety Alg.

1.  $Work = Available$ ,  
 $Finish[i] = false, 0 \leq i < n$ .
2. Find an  $i = 4$ .
3.  $Work = Work + Allocation_i$ ,  
 $Finish[4] = true$ 

|      |   |   |   |
|------|---|---|---|
|      | A | B | C |
| Work | 3 | 4 | 3 |
4. Go to step 2.
5. Find an  $i = 2$ .
6.  $Work = < 3, 6, 5 >$ ,  
 $Finish[2] = true$
7. Go to step 2.
8. Find an  $i = 3$ .

- Safe sequence:  $< P_4, P_2, P_3, P_1 >$



# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
- 6. Deadlock Detection**
7. Recovery from Deadlock



## Deadlock Detection

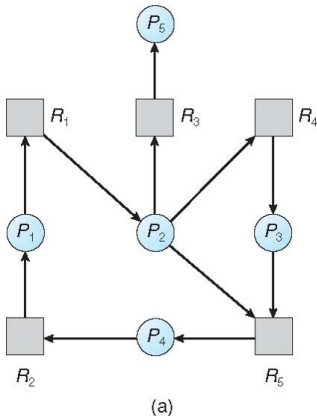
### *Single Instance of Each Resource Type*

- 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
- 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

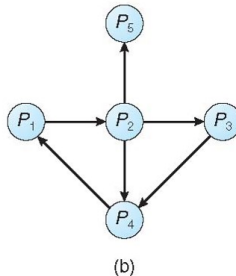


# Deadlock Detection

## Wait-for Graph



Resource-Allocation Graph



Wait-for Graph



## Deadlock Detection

### *Several Instances of a Resource Type*

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process
- **Request:** An  $n \times m$  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  $R_j$ .





## Deadlock Detection

### Detection Algorithm

1. Let *Work* and *Finish* be vectors of length  $m$  and  $n$ , respectively.

Initialize:

$Work = Available$

For  $i = 1, 2, \dots, n$ , 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.  $P_i$  is deadlocked.



# Deadlock Detection

## *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.



# Contents

1. System Model
2. Deadlock Characterization
3. Methods for Handling Deadlocks
4. Deadlock Prevention
5. Deadlock Avoidance
6. Deadlock Detection
- 7. Recovery from Deadlock**



# 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
  - Is process interactive or batch?

# 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