Deadlocks

Contents of Lecture:

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention

References for This Lecture:

- ✓ *Abraham Silberschatz, Peter Bear Galvin and Greg Gagne, Operating System Concepts, 9th Edition,* ***Chapter 7***

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.

- The resources may be partitioned into several types (or classes (R1, R2, . . ., Rm)), each consisting of some number of identical instances.
  - ✓ CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.
  - ✓ If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances

- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request:** The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
  2. **Use:** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
  3. **Release:** The process releases the resource.

Deadlock Characterization

❖ In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

Necessary Conditions

❖ A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a nonsharable mode; only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption:** Resources cannot be preempted; a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait:** There exists a set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Resource-Allocation Graph

❖ Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.

❖ This graph consists of a set of **vertices V** and a set of **edges E**.

- ✓ The **set of vertices V** is partitioned into two different types of nodes:
  - P = {P1, P2, ..., Pn}, the set consisting of all the **active processes** in the system.
  - R = {R1, R2, ..., Rm}, the set consisting of all **resource types** in the system.

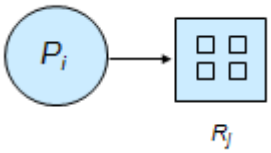
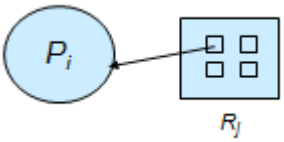
- ✓ A directed edge from process Pi to resource type Rj is denoted by Pi → Rj ; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
  - A directed edge Pi → Rj is called a **request edge**.

- ✓ A directed edge from resource type Rj to process Pi is denoted by Rj → Pi ; it signifies that an instance of resource type Rj has been allocated to process Pi .
  - A directed edge Rj → Pi is called an **assignment edge**.

❖ **Representation of graph:**

- ✓ Represent each process Pi as a circle
- ✓ Each resource type Rj as a rectangle.
- ✓ Since resource type Rj may have more than one instance, represent each such instance as a dot within the rectangle.

- ✓ Note that a request edge points to only the rectangle Rj , whereas an assignment edge must also designate one of the dots in the rectangle.

▪ Process	
▪ Resource Type with 4 instances	
▪ Pi requests instance of Rj	
▪ Pi is holding an instance of Rj	

❖ The resource-allocation graph shown in next figure (Figure 7.1) depicts the following situation:

- ✓ The sets P, R, and E:
  - P = {P1, P2, P3}
  - R = {R1, R2, R3, R4}
  - E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

- ✓ Resource instances:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4

- ✓ Process states:
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
  - Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
  - Process P3 is holding an instance of R3.

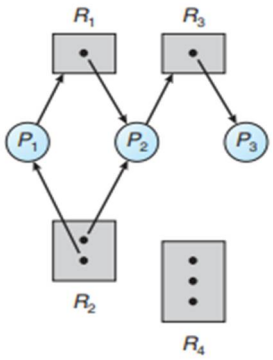


Figure 7.1 Resource-allocation graph.

❖ **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

❖ **Example of graph with a deadlock:** Two cycles exist in the next figure (Figure 7.2):

- ✓ P1 → R1 → P2 → R3 → P3 → R2 → P1
- ✓ P2 → R3 → P3 → R2 → P2

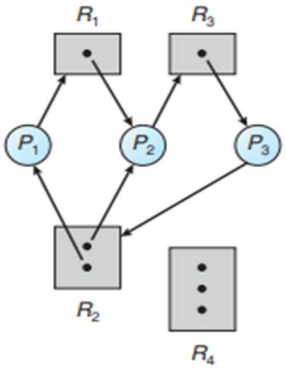


Figure 7.2 Resource-allocation graph with a deadlock.

❖ **Example of graph without a deadlock:** Consider the resource-allocation graph in next figure (Figure 7.3). also have a cycle:

- ✓ P1 → R1 → P3 → R2 → P1

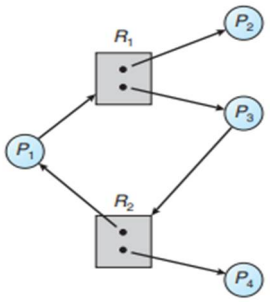


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

Methods for Handling Deadlocks

- Generally speaking, we can deal with the deadlock problem in one of three ways:
  - ✓ Can use a protocol to prevent or avoid deadlocks, ensuring that the system will **never** enter a deadlocked state.
    - Deadlock **prevention**
    - Deadlock **avoidence**
  - ✓ Can allow the system to enter a deadlocked state, **detect** it, and recover.
  - ✓ Can ignore the problem altogether and pretend that deadlocks never occur in the system (used by most operating systems, including UNIX).

Deadlock Prevention

- To occur a deadlock, there are four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, can prevent the occurrence of a deadlock.

1. **Mutual Exclusion:** not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

2. **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**

3. **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

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

❖ **Next figure (Figure 7.4) show 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);
}
```

Figure 7.4 Deadlock example

❖ **Next figure (Figure 7.5) show 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);
}
```

Figure 7.5 Deadlock example with lock ordering.