## Pthread Semaphore

- Pthread semaphores for multi-process and multi-thread programming:
  - Pthread Mutex Lock (binary semaphore)
  - Pthread Semaphore (general counting semaphore)

## Pthread Mutex Lock

```
#include <pthread.h>
/*declare a mutex variable*/
pthread_mutex_t  mutex ;

/* create a mutex lock */
pthread_mutex_init (&mutex, NULL) ;

/* acquire the mutex lock  */
pthread_mutex_lock(&mutex) ;

/* release the mutex lock  */
pthread_mutex_unlock(&mutex) ;
```

## Using Pthread Mutex Locks

- Use mutex locks to solve critical section problems:

```
#include <pthread.h>
pthread_mutex_t  mutex ;
…
pthread_mutex_init(&mutex, NULL) ;
…
pthread_mutex_lock(&mutex) ;

/*** critical section ***/

pthread_mutex_unlock(&mutex) ;
```

## Pthread Semaphores

```
#include <semaphore.h>
/*declare a pthread semaphore*/
sem_t sem ;

/* create and initialize a semaphore */
sem_init (&sem, flag, initial_value) ;

/* wait() operation  */
sem_wait(&sem) ;

/* signal() operation  */
sem_post(&sem) ;
```

## Using Pthread semaphore

- Using Pthread semaphores for counters shared by multiple threads:

```
#include <semaphore.h>
sem_t  counter ;
…
sem_init(&counter, 0, 0) ;  /* initially 0 */
…
sem_post(&counter) ;    /* increment */
…
sem_wait(&counter) ;    /* decrement */
```

CSE3221
Operating System Fundamentals

No. 7

# Deadlocks

*Prof. Hui Jiang*
*Dept of Computer Science and Engineering*
*York University*

## What is Deadlock?

- **A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.**
  - → **never change state again**
- **Example I:**
  - **System has 2 tape drives.**
  - **$P_1$ and $P_2$ each hold one tape drive and each needs another one.**
- **Example II:**
  - **semaphores *A* and *B*, initialized to 1**

| $P_0$ | $P_1$ |
|---|---|
| *wait (A);* | *wait(B)* |
| *wait (B);* | *wait(A)* |
| … | … |

## System Model

- A set of processes to compete resources
- Resources are partitioned into several types $R_1, R_2, \ldots, R_m$
  - Memory space, CPU cycles, files, I/O devices
- Several instances for each resource type:
  - All instances for a resource type are identical
- Request-Use-Release model
  - Request: the process must wait if the request can not be granted immediately.
  - Use: the process can operate on the resource
  - Release: the process release the resource for others to use.

## Deadlock Characterization

**Deadlock can arise only 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 $\{P_0, P_1, \ldots, P_n, P_0\}$ 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 $P_2$, …, $P_{n-1}$ is waiting for a resource that is held by $P_n$, and $P_n$ is waiting for a resource that is held by $P_0$.

## Resource-Allocation Graph (I)

**A set of vertices *V* and a set of edges *E*.**

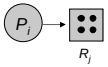
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge** – directed edge $P_1 \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

## Resource-Allocation Graph (II)

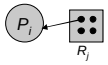
- **Process**

- **Resource Type with 4 instances**

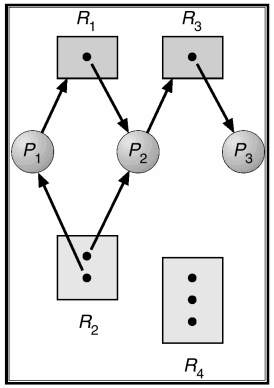
- **$P_i$ requests instance of $R_j$**




- **$P_i$ is holding an instance of $R_j$**



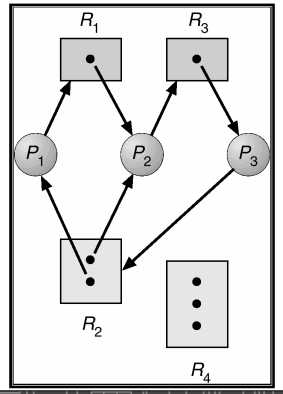

## Example of a Resource Allocation Graph



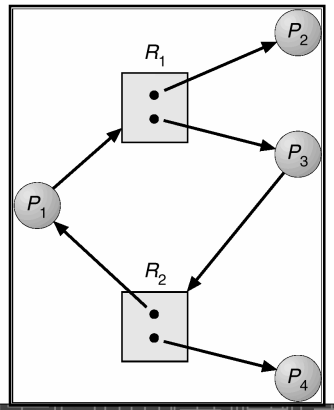
## Basic Facts

- 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.

## Resource Allocation Graph With A Deadlock




## Resource Allocation Graph With A Cycle But No Deadlock




## Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - Deadlock Prevention
  - Deadlock Avoidance
- Allow the system to enter a deadlock state and then detect and recover.
- Ignore the deadlock problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

## Deadlock Prevention(I)

**Restrain the ways request can be made.**

- **Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.**
  - **→ Nothing can be done for non-sharable resources**
- **Hold and Wait – must require that whenever a process requests a resource, it does not hold any other resources.**
  - **Protocol 1: require process to request and be allocated all its needed resources in one chunk before it begins execution.**
  - **Protocol 2: allow process to request resources only when the process has none.**
  - **Problems: low resource utilization; starvation possible.**

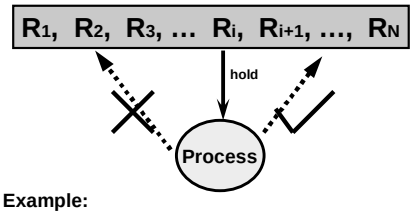
## Deadlock Prevention (II)

- **No Preemption –**
  - **Allow 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 can be preempted by other processes.
  - 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.
  - If a process requests some resources:
    - If available, allocate
    - If not available but used by a waiting process, preempt that resource
    - Otherwise, the process wait (its occupied resources may be preempted by others)

## Deadlock Prevention (III)

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

Organize all resources in a linear order




**Example:**

**Tape device → disk device → printer**