CSC 4320/6320: Operating Systems

GeorgiaState University

Chapter 07: Synchronization Examples-contd.

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

```
/* the semaphore is shared between threads in the same process */
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor (another locking mechanism to ensure data integrity), POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX Condition Variables

- Thread waiting for the condition **`a == b`** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

/* releases lock, puts the thread **to sleep** instead of busy waiting */

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

/* the signaled thread becomes the owner of the mutex lock and returns control from the call to pthread_cond_wait(). */

Questions

1. Pthreads can be implemented

A) only inside the operating system kernel

B) only at the user level

C) at the user level or inside the operating system kernel ✓

D) only Windows OS

2. When the owner of a mutex lock invokes pthread mutex unlock(), all threads blocked on that mutex's lock are unblocked.

True

False ✓

3. A call to pthread_cond_signal()

A) releases the mutex lock and signals one thread waiting on the condition variable.

B) releases the mutex lock and signals all threads waiting on the condition variable.

C) signals one thread waiting on the condition variable, but does not release the mutex lock. ✓

D) signals all threads waiting on the condition variable, but does not release the mutex lock.

CSC 4320/6320: Operating Systems

Chapter 08: Deadlocks

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - *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**

Deadlock and livelock

Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set

Livelock occurs when a thread continuously attempts an action that fails.

Deadlock in multithreaded application

```c
/* 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);
}
```

```c
/* 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 with Semaphores

- Data:
  - A semaphore s_1 initialized to 1
  - A semaphore s_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

```
wait(s1)
wait(s2)
```

- T_2:

```
wait(s2)
wait(s1)
```

Livelock Example

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&first_mutex);
      if (pthread_mutex_trylock(&second_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&second_mutex);
         pthread_mutex_unlock(&first_mutex);
         done = 1;
     }
     else
         pthread_mutex_unlock(&first_mutex);
   }
   pthread_exit(0);
}
```

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
   int done = 0;

   while (!done) {
     pthread_mutex_lock(&second_mutex);
     if (pthread_mutex_trylock(&first_mutex)) {
         /**
          * Do some work
          */
         pthread_mutex_unlock(&first_mutex);
         pthread_mutex_unlock(&second_mutex);
         done = 1;
     }
     else
         pthread_mutex_unlock(&second_mutex);
   }

   pthread_exit(0);
}
```

What if thread one acquires `first_mutex`, followed by thread two acquiring **second_mutex.** Each thread then invokes **pthread_mutex_trylock()**, which fails, releases their respective locks, and repeats the same actions indefinitely.

Questions

1. A deadlocked state occurs whenever ____.

A) a process is waiting for I/O to a device that does not exist

B) the system has no available free resources

C) every process in a set is waiting for an event that can only be caused by another process in the set ✓

D) a process is unable to release its request for a resource after use

2. Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, while livelock occurs when a thread continuously attempts an action that fails.

True ✓

False

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one thread at a time can use a resource
- **Hold and wait**: a thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**: a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
- **Circular wait**: there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, …, T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

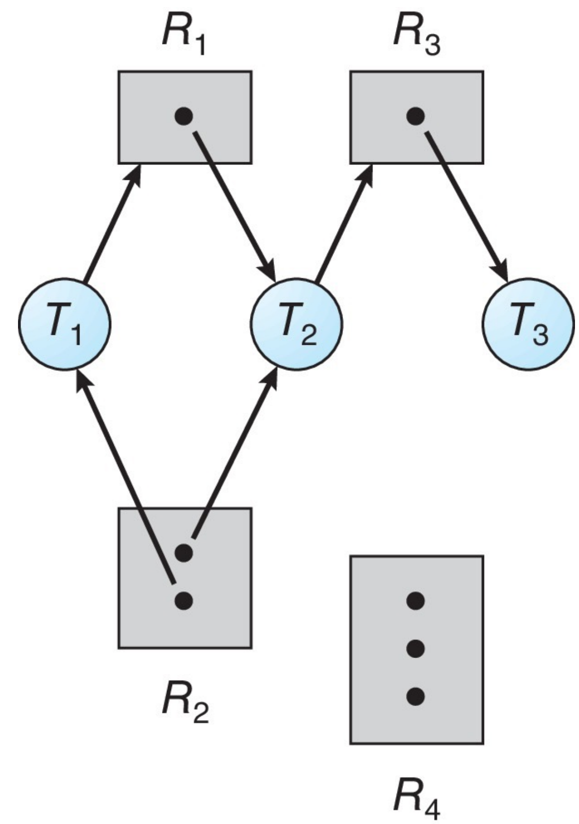
Resource-Allocation Graph

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

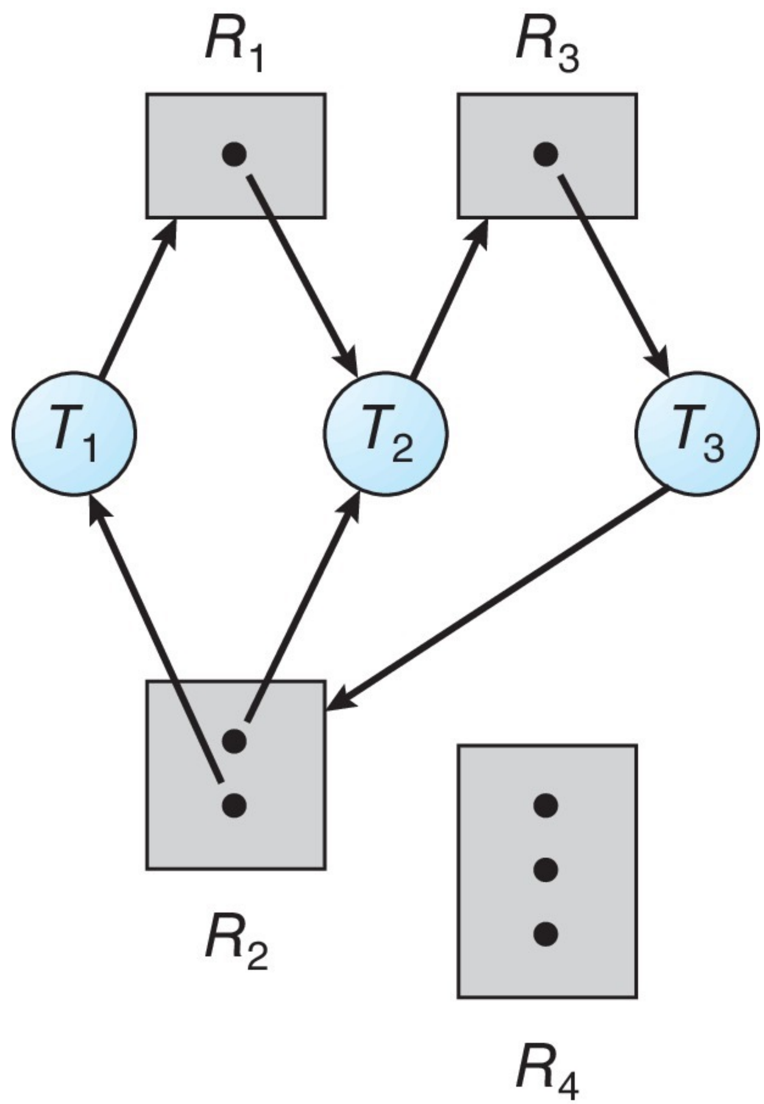
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

Resource Allocation Graph Example

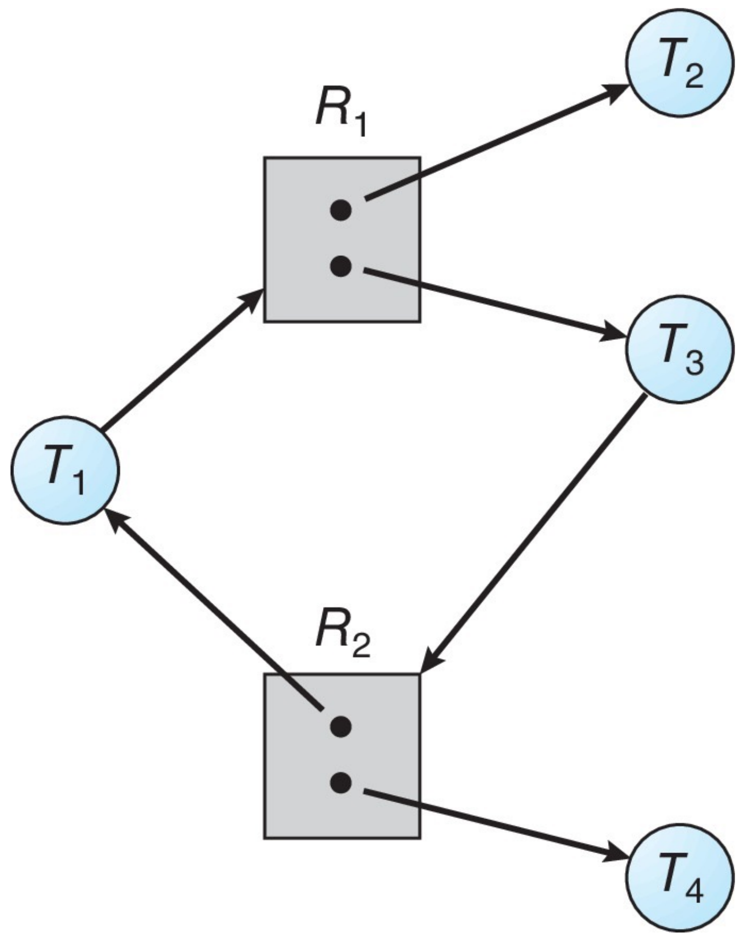
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 holds one instance of R_3



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



Observe that thread T4 may release its instance of resource type R2. That resource can then be allocated to T3, breaking the cycle.

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

Questions

1. A cycle in a resource-allocation graph is _____.

A) a necessary and sufficient condition for deadlock in the case that each resource has more than one instance

B) a necessary and sufficient condition for a deadlock in the case that each resource has exactly one instance ✓

C) a sufficient condition for a deadlock in the case that each resource has more than once instance

D) is neither necessary nor sufficient for indicating deadlock in the case that each resource has exactly one instance

2. If a resource-allocation graph has a cycle, the system must be in a deadlocked state.

True

False ✓