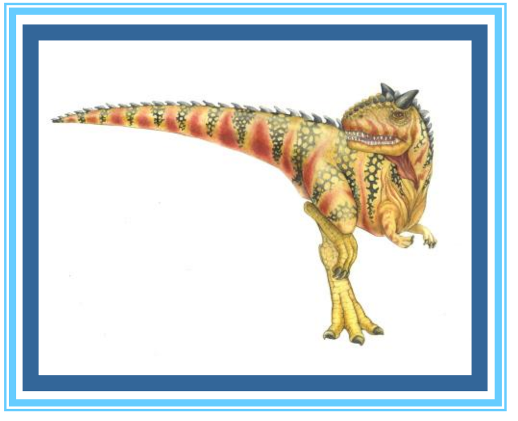
Chapter 7: Synchronization Examples



Outline

- Explain the bounded-buffer synchronization problem
- Explain the readers-writers synchronization problem
- Explain and dining-philosophers synchronization problems
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0 (# of full buffers)
- Semaphore **`empty`** initialized to the value n (# of empty buffers)

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
}
```

Protects against adding into full; empty will reach 0 (blocking) after n productions if consumer never consumes any, i.e., signal(empty)

lock for the CS

Protects against drawing from empty; full is incremented by 1 for every production; 0 represents no item to be consumed, causing the consumer process to wait

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);   Wait if # of full buffers is zero (nothing to consume)
    wait(mutex);  Lock the CS
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);  Increase the # of empty buffers by 1
        ...
      /* consume the item in next consumed */
        ...
   }
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 Only one writer allowed at a time
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

This is known as the *first readers-writers problem*: Access is favored to readers; writers can only access data while there are no readers, i.e., writers may starve.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait(rw_mutex);
            ...
        /* writing is performed */
            ...
        signal(rw_mutex);
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
        wait(mutex); protects modifying read_count
        read_count++;
        if (read_count == 1) /* first reader */
             wait(rw_mutex); only blocks writer access, more
                             readers are always allowed
        signal(mutex);  opens access to read_count

        ...
        /* reading is performed */

        ...
        wait(mutex); again, protects modifying read_count
        read_count--;
        if (read_count == 0) /* last reader */
             signal(rw_mutex); opens writer access if no readers
        signal(mutex); restores access to read_count
}
```

Readers-Writers Problem Variations

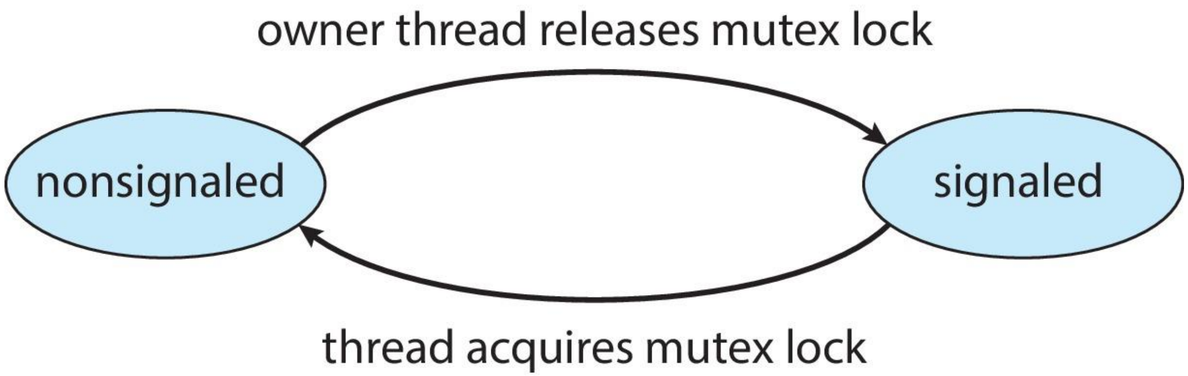
- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
- The “Second reader-writer” problem is a variation of the first reader-writer problem that states:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
- Both the first and second may result in starvation, leading to even more variations
- Best suited to applications where
  - Readers and writers are easily identified
  - There are more readers than writers

Kernel Synchronization - Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - For protecting short code segments
  - Kernel ensures a spinlocking-thread will never be preempted (spinlocks are generally used when the lock will be held for less than two context switches)
- For thread synchronization outside the kernel, Windows also provides user-land **dispatcher objects** which may act as mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects may be in either **signaled-state** (object available) or **non-signaled state** (thread will block)

Kernel Synchronization - Windows

- Mutex dispatcher object



- If a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting
- When the state for the dispatcher object moves to signaled, the kernel can move **one thread** (waiting on a mutex), or **some threads** (waiting on semaphores), or **all threads** (waiting on events), from the waiting state to the ready state so they can resume executing

Kernel Synchronization - Linux

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections (i.e., nonpreempted)
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Atomic integers (all math using them are not interrupted)
  - Mutex locks (mutex_lock() and mutex_unlock())
  - Semaphores
  - Spinlocks
  - Reader-writer versions of both semaphores and spinlocks
- On SMP machines, spinlocks are used (other threads can still run on separate processors)
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- Atomic variables

  `atomic_t` is the type for atomic integer

- Consider the variables

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

- Note that these methods pertain to synchronization available only to kernel developers. For user level programmers, use the POSIX API

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

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

/* critical section */

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

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

POSIX Named Semaphores

- Creating and initializing the semaphore:

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

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **`SEM`**.
  - They can use `sem = sem_open(”SEM”, O_RDWR)`
- Acquiring and releasing the semaphore:

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

/* critical section */

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

POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

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

Related processes are those that have access to this global variable **sem**

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

The second argument: 0: shared among threads, 1: shared among processes

- 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, 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);
```

Here, it releases the mutex lock on entry, and blocks, then regains the mutex on successful return.

Can we use `if (a!=b)`?

- Thread signaling another thread waiting on the condition variable:

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

Java Synchronization

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables

Java Monitors

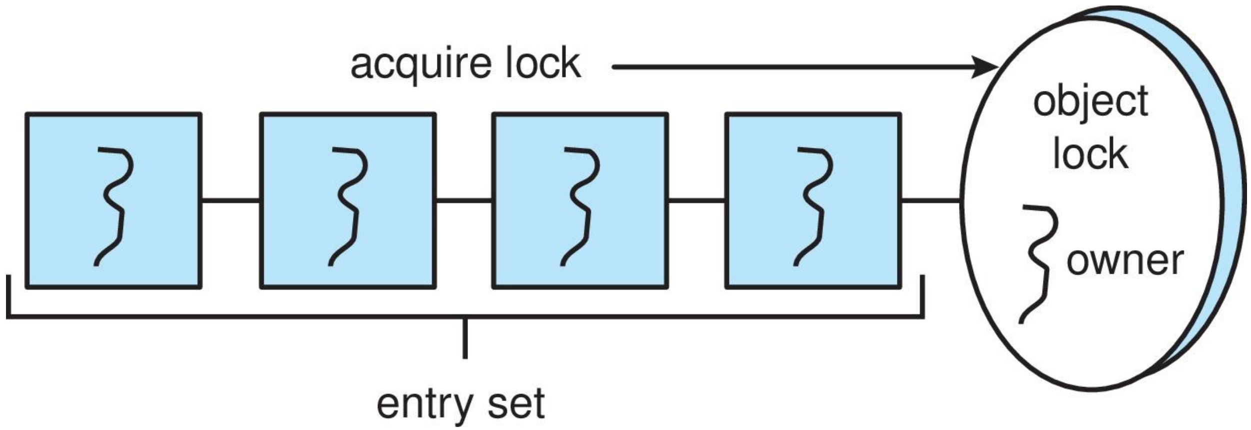
- Every Java object has associated with it a single lock.
- If a method is declared as **`synchronized`**, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the **`synchronized`** method.

Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */  Slide 7.29
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */  Slide 7.30
   }
}
```

Java Synchronization

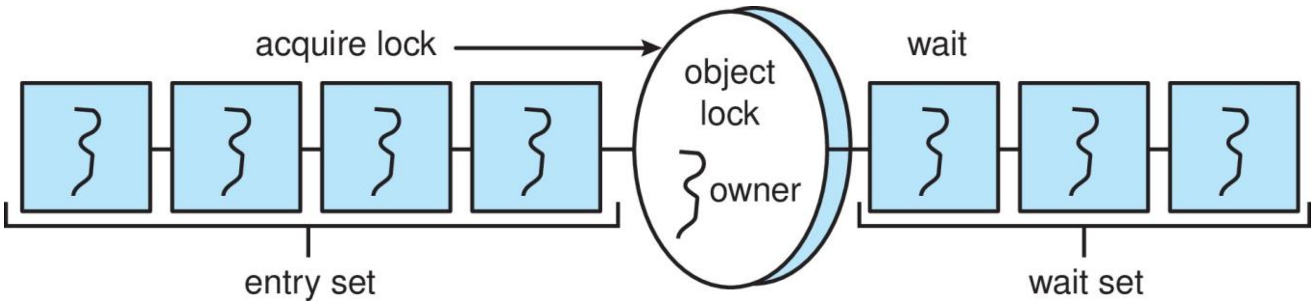
- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:



When the lock is available, JVM arbitrarily picks one thread (usually FCFS though) to become the owner of the object lock and it can enter the *synchronized* method. The lock is released when the thread exits such method.

Java Synchronization

- Similarly, each object also has a **wait set**.
- When a thread calls **`wait()`**:
  1. It releases the lock for the object
  2. The state of the thread is set to blocked
  3. The thread is placed in the wait set for the object



A thread inside the *synchronized* method (owning the lock) may have to wait for certain event (condition). For example, a producer calls insert() and the buffer is full. The thread will release the lock and wait inside the wait set until the condition (one buffer's content consumed) is met to continue.

Java Synchronization

- A thread typically calls wait() when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls **`notify()`**:
  1. An arbitrary thread T is selected from the wait set
  2. T is moved from the wait set to the entry set
  3. Set the state of T from blocked to runnable.
- T can now compete for the lock to check if the condition it was waiting for is now true (so use while-loop instead of if-clause).

Bounded Buffer – Java Synchronization

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   count++;

   notify();
}
```

lock is released, and calling thread (producer) blocks and being placed in the wait set; Will regain the lock and return when notified by consumer

Picks an arbitrary thread from the wait set, and moves it to the entry set, setting its state from *blocked* to *runnable*
If there is no thread in wait set, the call is ignored

Bounded Buffer – Java Synchronization

```java
/* Consumers call this method */
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

Java Reentrant Locks

- Similar to mutex locks (with a fairness feature that favors granting the lock to the longest-waiting thread – default is no particular order)
- The **`finally`** clause ensures the lock will be released in case an exception occurs in the **`try`** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

Returns if the lock is available or already owned by the current thread – thus reentrant

In the `finally` clause to guarantee proper releasing of the lock in case of exceptions

Java Semaphores

- Constructor:

```
Semaphore(int value);
```

- Usage:

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Java Condition Variables

- Condition variables are associated with an **ReentrantLock**.
- Creating a condition variable using **newCondition()** method of **ReentrantLock**:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the **await()** method, and signals by calling the **signal()** method.

Java Condition Variables

- Example (a bunch of workers take turns to do intermittent work):
- Five threads numbered 0 .. 4
- Shared variable **`turn`** indicating which thread's turn it is.
- Thread calls **`doWork()`** when it wishes to do some work. (But it may only do work if it is their turn.)
- If not their turn, wait
- If their turn, do some work for awhile ……
- When completed, notify the thread whose turn is next.
- Necessary data structures:

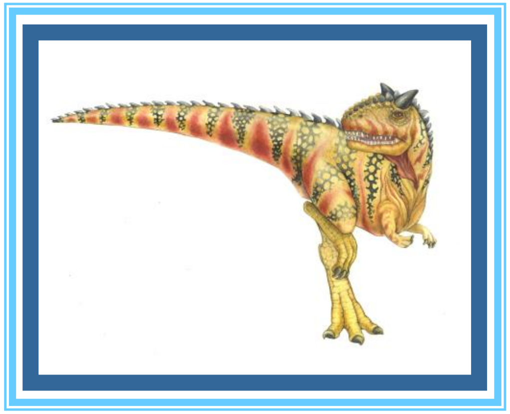
```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```

Here the condition is the circling turn

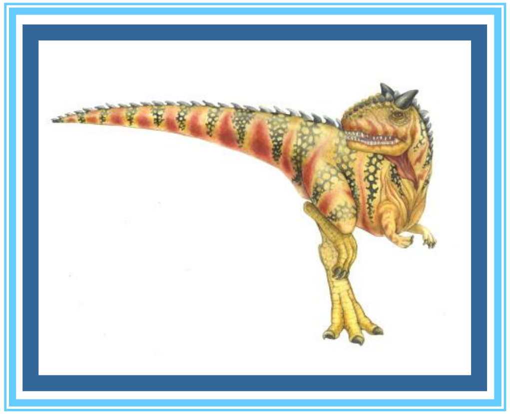
Java Condition Variables

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
  lock.lock();

  try {
    /**
     * If it's not my turn, then wait
     * until I'm signaled.
     */
    if (threadNumber != turn)
      condVars[threadNumber].await();

    /**
     * Do some work for awhile ...
     */

    /**
     * Now signal to the next thread.
     */
    turn = (turn + 1) % 5;
    condVars[turn].signal();
  }
  catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

End of Chapter 7



Chapter 8: Deadlocks



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 with Semaphores

- Data:
  - A semaphore **`S1`** initialized to 1
  - A semaphore **`S2`** initialized to 1
- Two processes P1 and P2
- **`P1:`**

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

- **`P2:`**

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

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 $\{P_0, P_1, \ldots, P_n\}$ 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, …, 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.

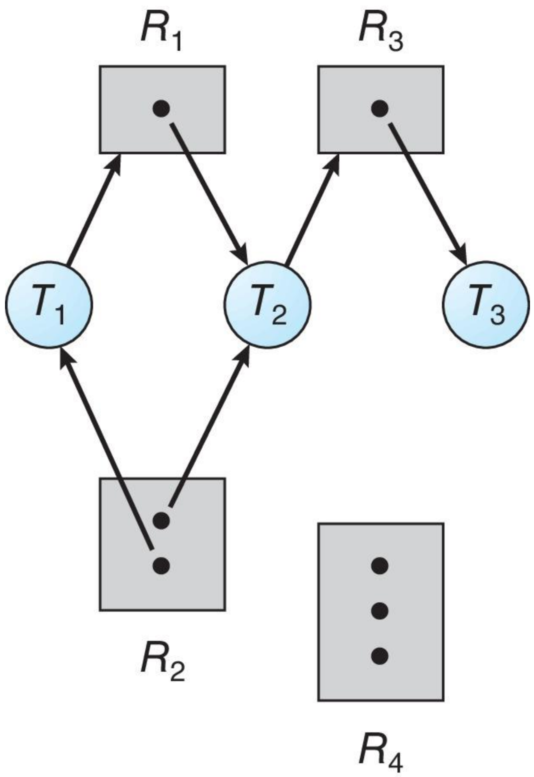
Resource-Allocation Graph

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

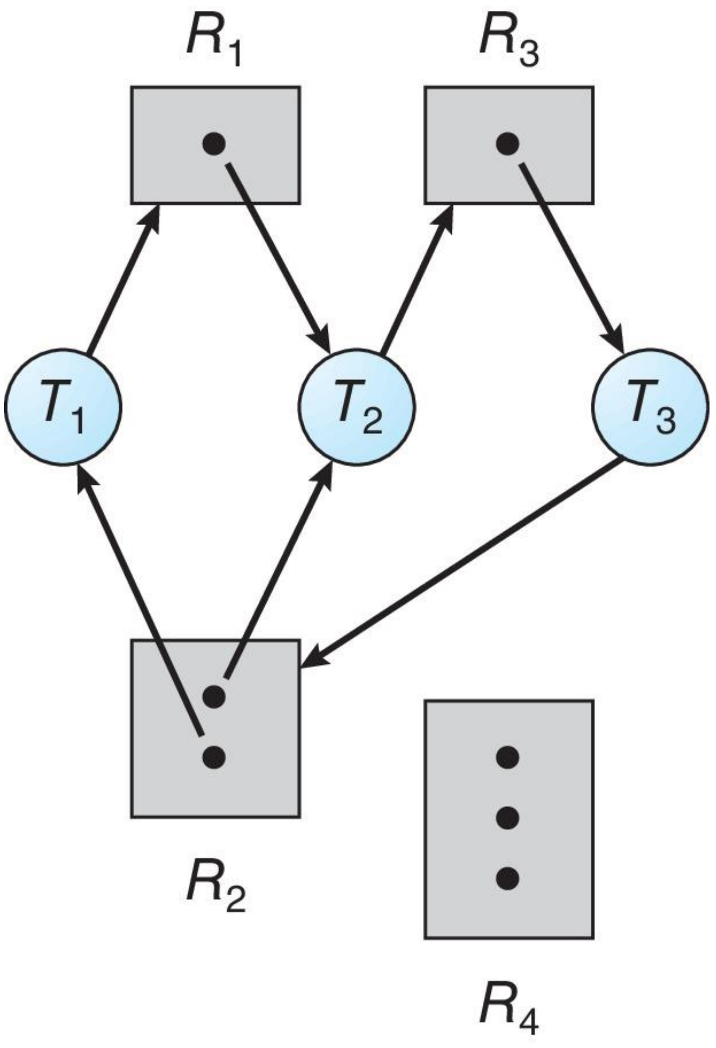
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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

Resource Allocation Graph Example

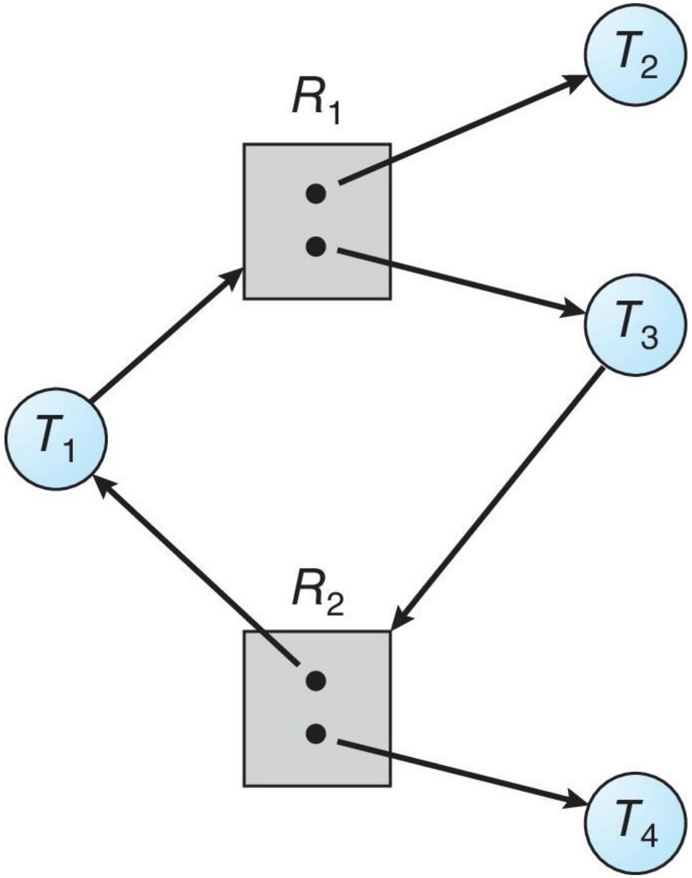
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



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

Methods for Handling Deadlocks

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance (avoid getting into unsafe states)
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system (this is the one used by most operating systems, including Linux and Windows)

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **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.
  - Issues: 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:**
  - Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

`F(first_mutex) = 1`
`F(second_mutex) = 5`

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

```
/* 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);
}
```

End of Chapter 8