CISC 3320

# C24a Deadlock and Resource Allocation Graph

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

# Acknowledgement


- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

# Outline

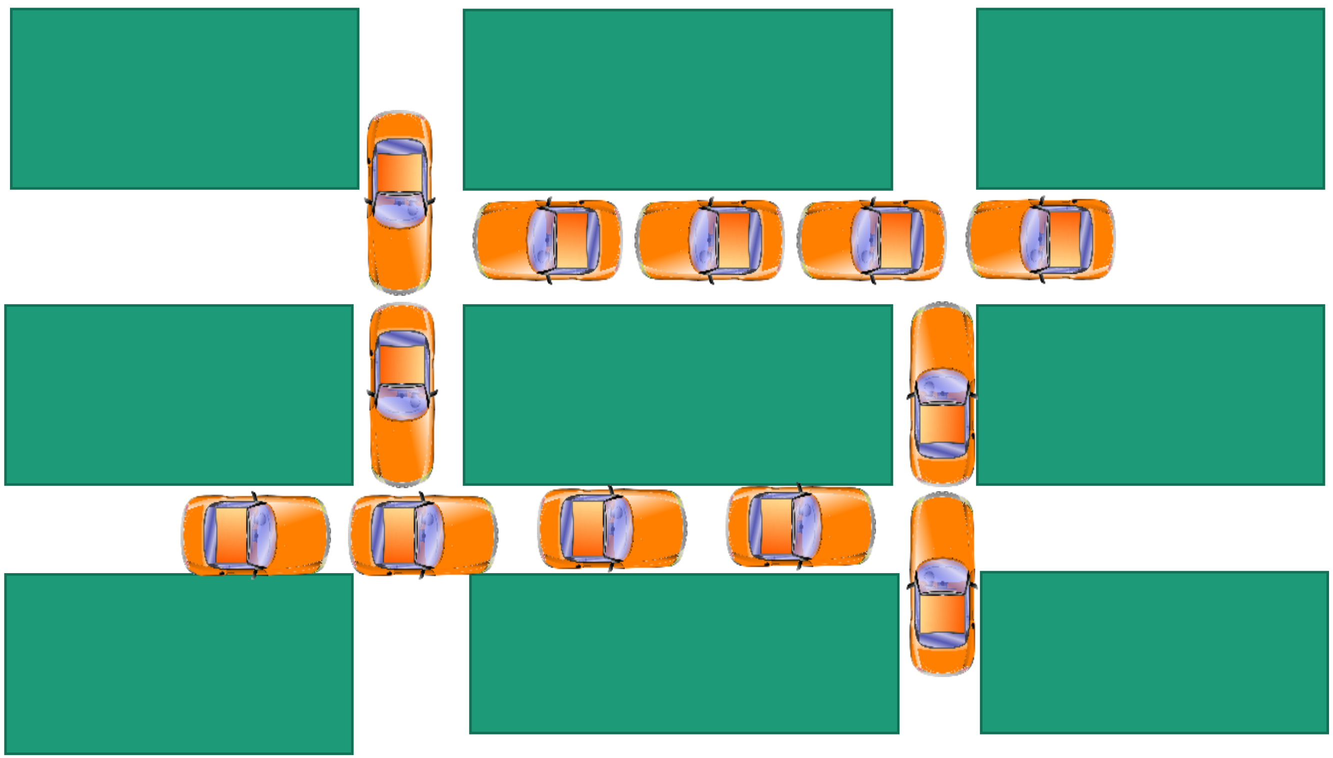
- System Model
- Deadlock Characterization (Necessary Conditions)
- Resource Allocation Graph
- Deadlock in Multithreaded Applications
- Overview of Methods for Handling Deadlocks

- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

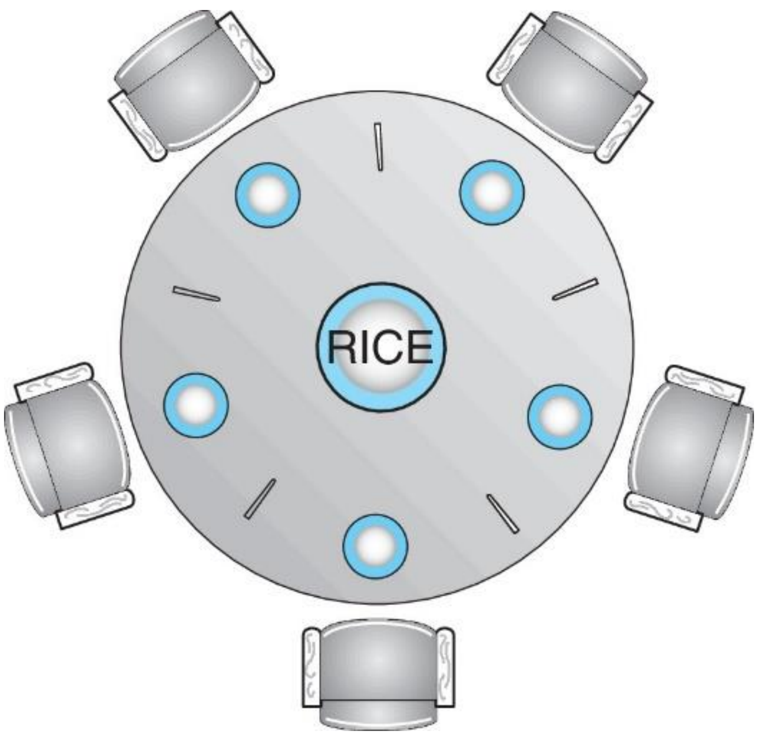
# Problem when Sharing Resources

- A proposed law by the Kansas State Legislature (Botkin and Harlow, 1953)
  - “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

# This can also happen …



# The Dining Philosophers




```
while (true){
  wait (chopstick[i] );
  wait (chopStick[ (i + 1) % 5] );

  /* eat for awhile */
  signal (chopstick[i] );
  signal (chopstick[ (i + 1) % 5] );

  /* think for awhile */
}
```

- What is the problem with this algorithm?

# System Model

- System consists of resources
- Resource types R1, R2, . . ., Rm
  - Examples
    - CPU cycles, memory space, I/O devices
- Each resource type Ri has Wi instances.
- A set of processes, and each process utilizes a resource as follows:
  - request
  - use
  - release

# Deadlock

- Every process in the set is waiting for an event to be triggered by another in the set (request or release resource)

# Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously. (the 4 necessary conditions for deadlocks)
- **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 $P_0$.

# Questions?

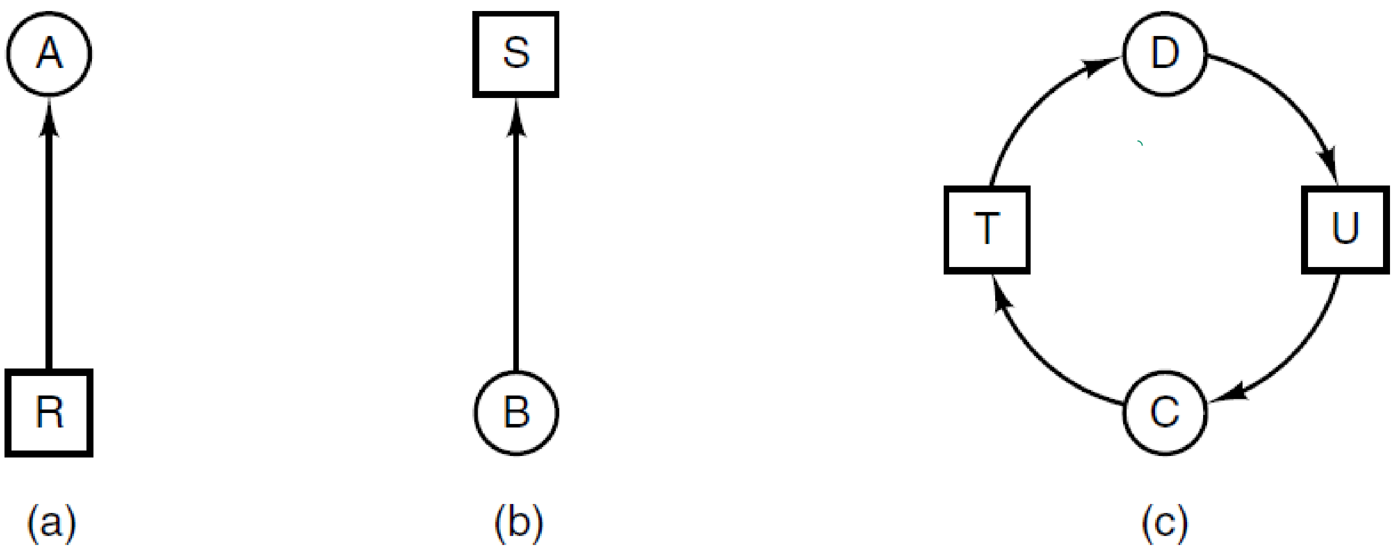
- Concept of deadlock
- Necessary conditions of deadlock

# 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 (drawn in <u>ovals</u>)
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system (drawn in <u>rectangles</u>)
- **request edge** – directed edge $P_i \rightarrow R_j$
  - $P_i$ requests or waits for $R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$
  - *$R_j$ is assigned to or is held by $P_i$*

# Resource-Allocation Graph: Example 1

- Can you describe the graphs in English? (Hint: oval: process; rectangle: resource; arrow: Resource → Process, Process → Resource, i.e., is being held/assigned to or requests by/waiting for)



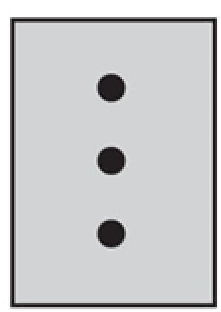

- Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock. [Figure 6-3 in Tanenbaum & Bos, 2014]

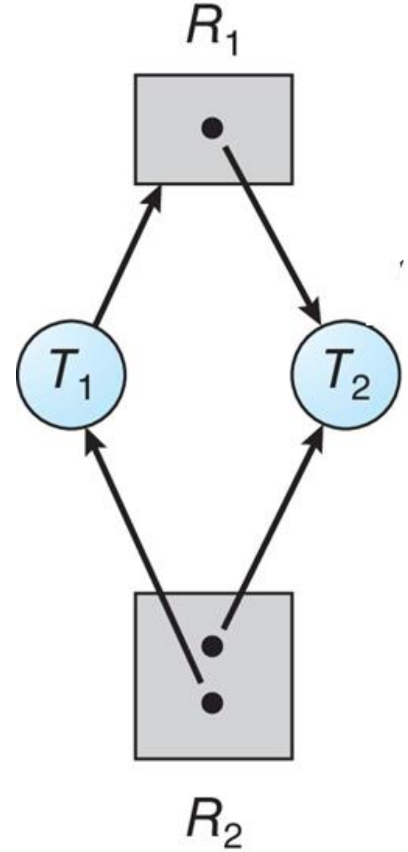
# Questions?

- Concept of resource allocation graph
- Examples of simple resource allocation graph
  - Each type of resources has only a single instance
- What if a type of resource has multiple instances?

# Resource with Multiple Instances

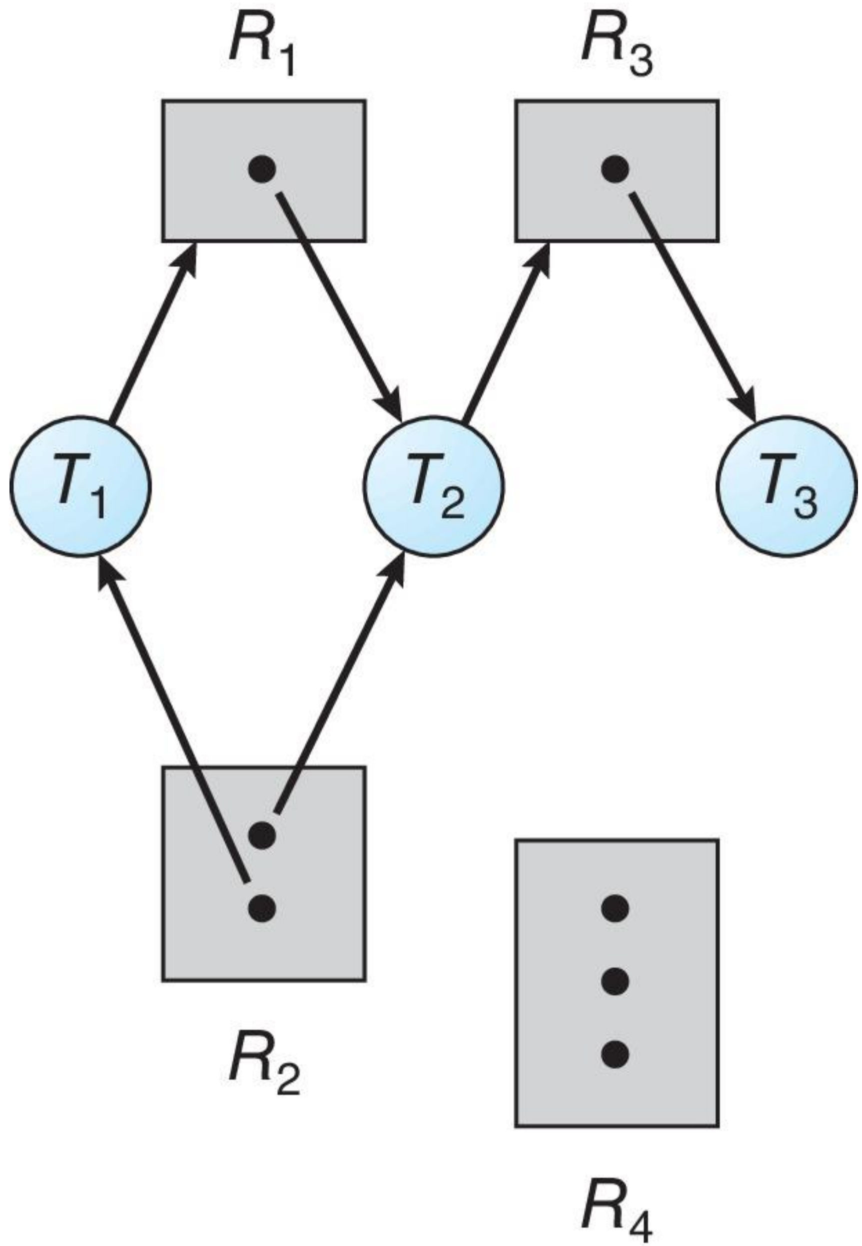
- A type of resource may have multiple instances
- Notations





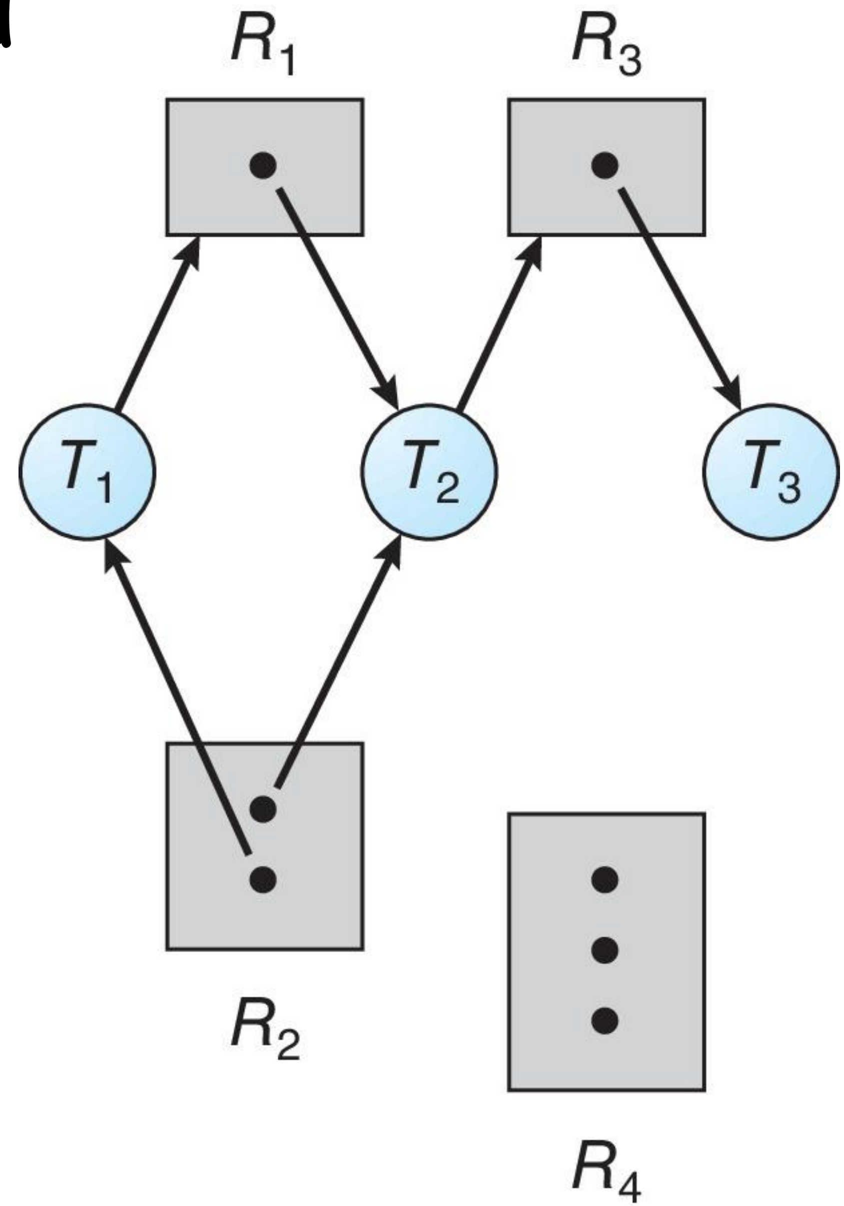
# Resource Allocation Graph: Example 2

- Can you draw the resource allocation graph for the following scenario?
  - 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


R1
R3
T1
T2
T3
R2
R4

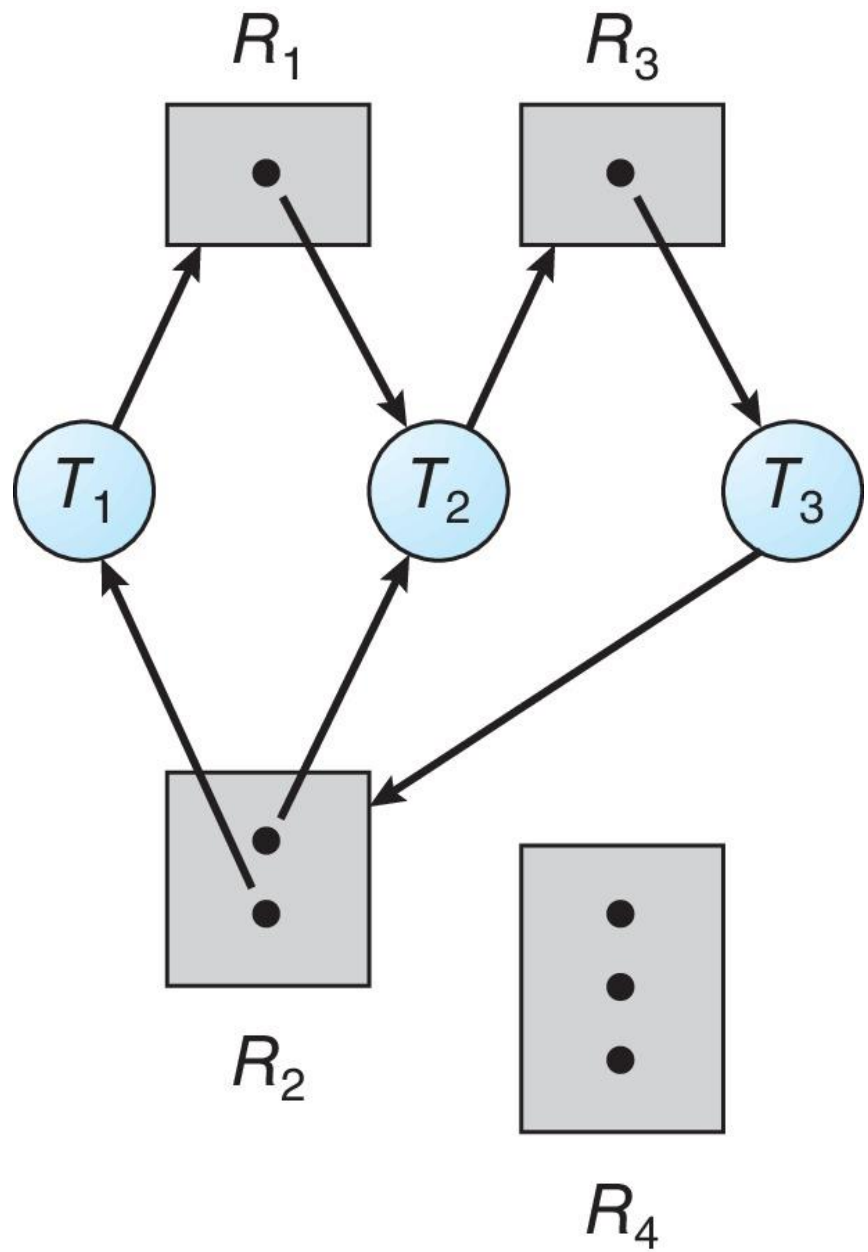
# Is There a Dead Lock?

- Mutual exclusion?
- Hold and wait?
- No preemption?
- Circular wait?



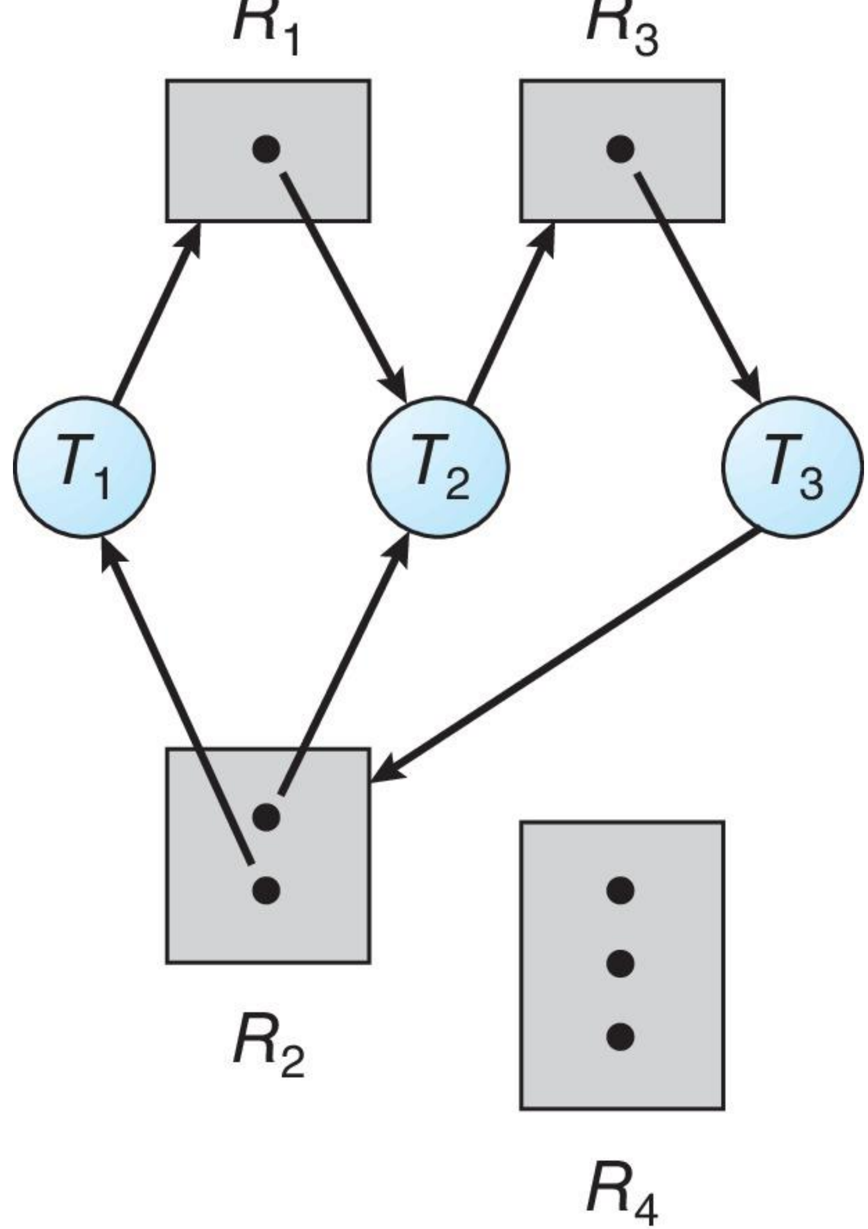
# Resource Allocation Graph: Example 3

- Can you draw the resource allocation graph for the following scenario?
  - 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, and is waiting for an instance of R2*


$R_1$
$R_3$
$T_1$
$T_2$
$T_3$
$R_2$
$R_4$

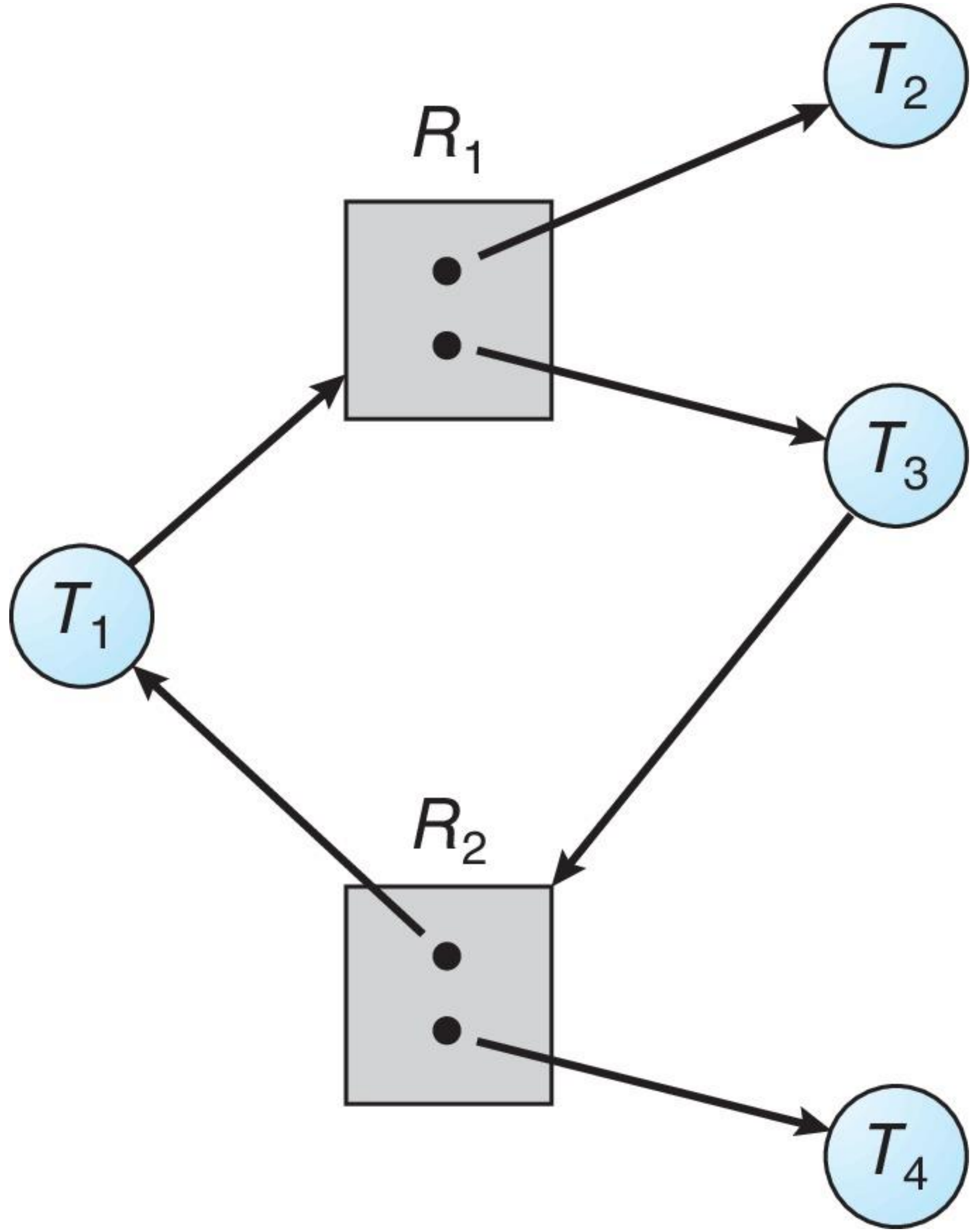
# Is There a Dead Lock?

- Mutual exclusion?
- Hold and wait?
- No preemption?
- Circular wait?



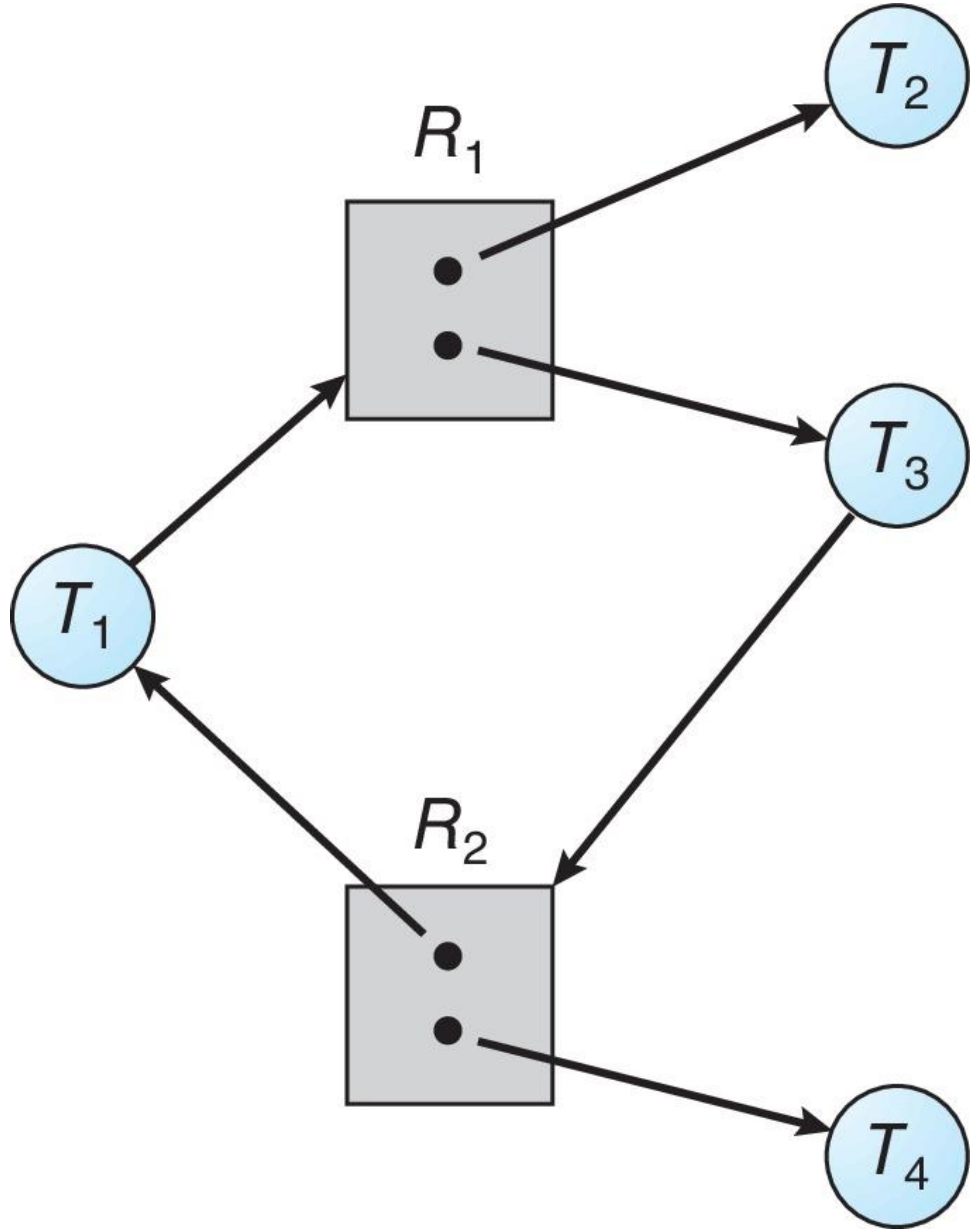
# Resource Allocation Graph: Example 4

- Can you draw the resource allocation graph for the following scenario?
  - Two instances of R1
  - Two instances of R2
  - T1 holds one instance of R2 and is waiting for an instance of R1
  - T2 holds one instance of R1
  - T3 holds one instance of R1 and is waiting for an instance of R2
  - T4 is waiting for an instance of R2


R1
R2
T1
T2
T3
T4

# Is There a Dead Lock?

- Mutual exclusion?
- Hold and wait?
- No preemption?
- Circular wait?



# Determine Existence of Deadlocks

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only <u>one</u> instance per resource type, then deadlock
  - if <u>several</u> instances per resource type, <u>possibility</u> of deadlock

# Resource Allocation Graph: Example 5

- What's the resource allocation graph?
  - 2 processes, P1 and P2 share two 2 CD-RW drives (D1, D2)
  - P1 is using D1, P2 is using D2
  - P1 requests D2 before releasing D1; P2 requests D1 before releasing D2
- Is there a deadlock?

# Questions?

- Resource allocation graph
- Determine existence of deadlock using resource allocation graph

# Deadlock and Scheduling

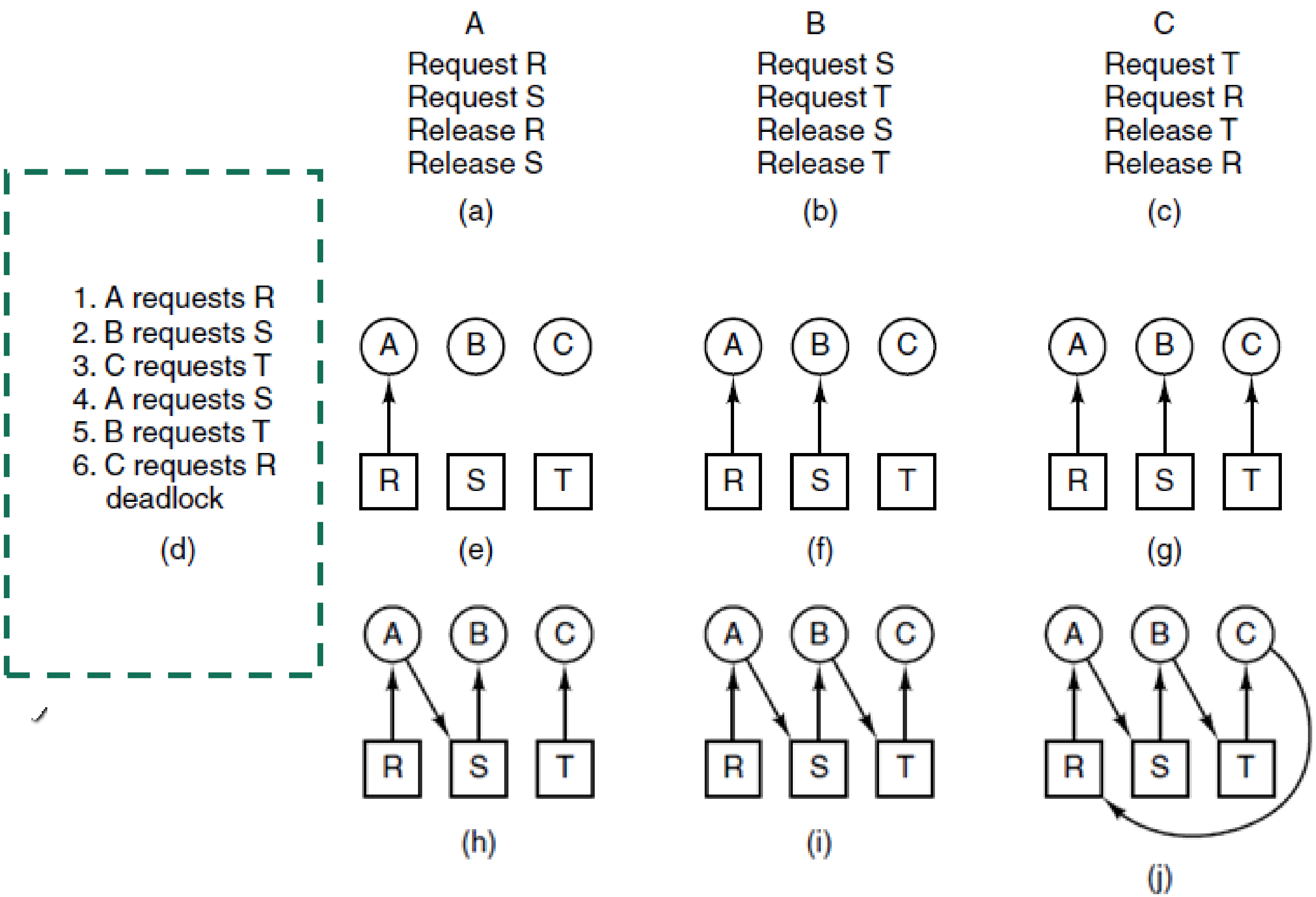
- Two examples
  - A generic example
    - Resource sharing and deadlock
  - A Pthread semaphore example
    - Semaphore and mutexes are resources.

# Resource Allocation and Scheduling: Example

- Three processes: A, B, C
- Three resources: R, S, T
- Each process's requests and release schedule is in the sequence below:

| A | B | C |
|---|---|---|
| Request R | Request S | Request T |
| Request S | Request T | Request R |
| Release R | Release S | Release T |
| Release S | Release T | Release R |
| (a) | (b) | (c) |

# OS Schedule with Deadlock



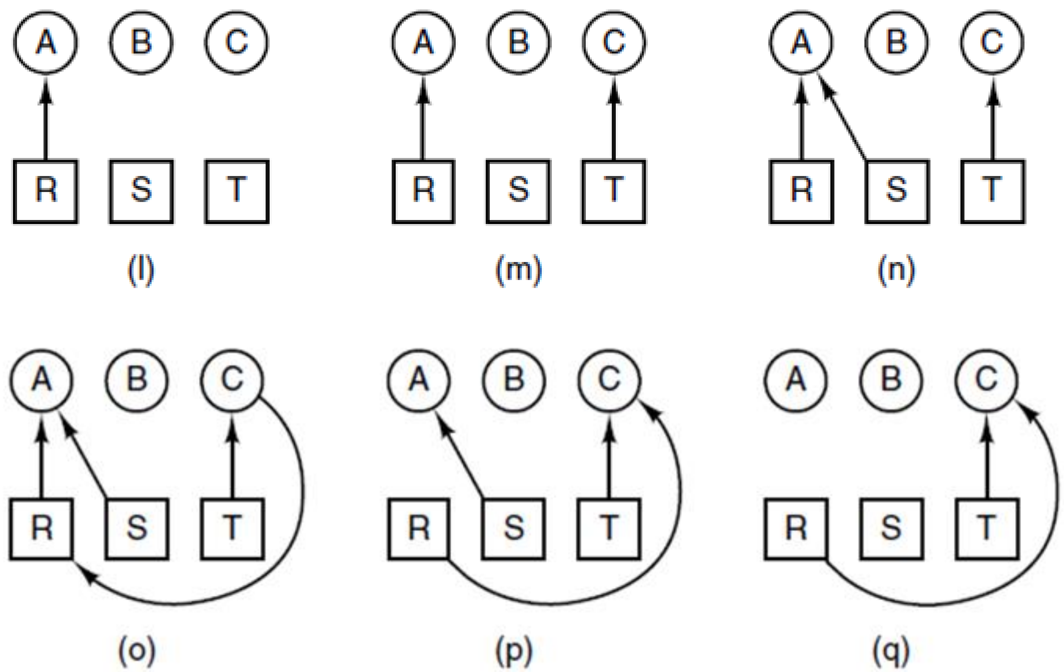
# Schedule without Deadlock

| A | B | C |
|---|---|---|
| Request R | Request S | Request T |
| Request S | Request T | Request R |
| Release R | Release S | Release T |
| Release S | Release T | Release R |
| (a) | (b) | (c) |

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

(k)



# Semaphores or Mutexes are Resources

- Access non-preemptive resource with semaphore (request, use, release)
  - down/signal/P; up/wait/V

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
      down(&resource_1);
      use_resource_1( );
      up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}
```

(b)

- [Figure 6-1 in Tanenbaum & Bos, 2014 (a) one resource (b) two resources]

# Coding Style Matters

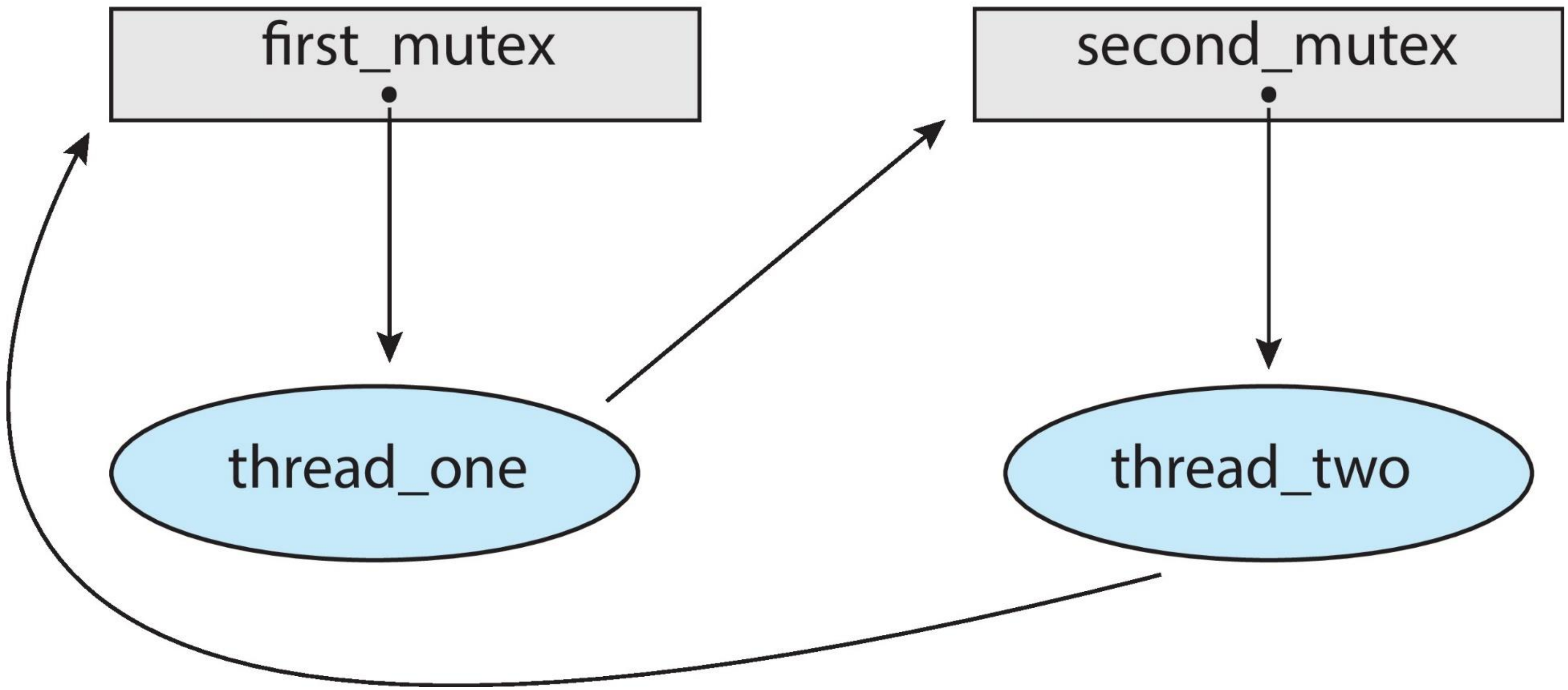
- Two mutex locks are created an initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

- Shared in the following fashion (next slide)
- Is there a dead lock?
  - Hint: mutex/binary semaphore; 0 or 1, available or not available; i.e., one instance per resource type)

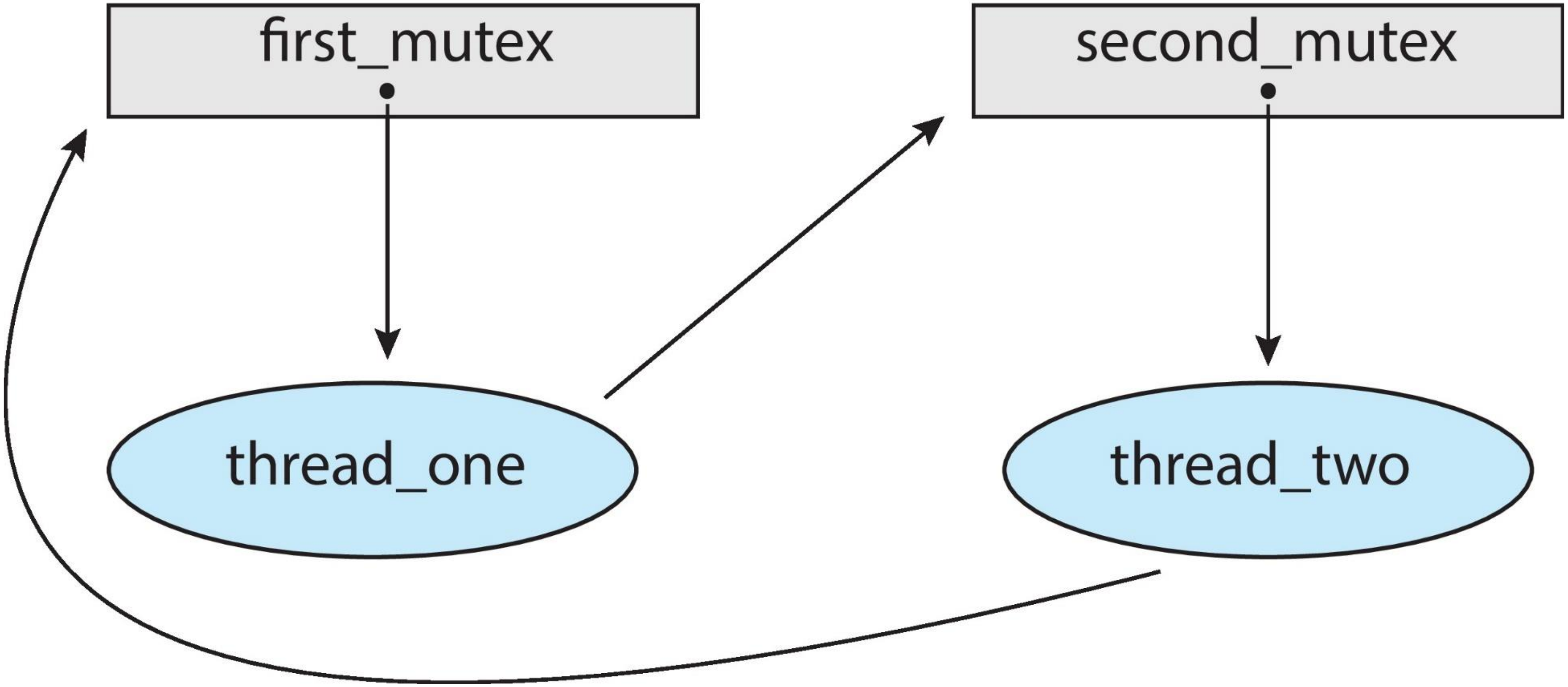
```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   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);
}
```

# Illustration using Resource Allocation Graph



# Resource-Allocation Graph: Example 6

- Describe the following resource allocation graph?



# Deadlock Scenario

- Deadlock occurs when
  - Thread 1 acquires `first_mutex` and thread 2 acquires `second_mutex`;
  - Thread 1 then waits for `second_mutex` and thread 2 waits for `first_mutex`.
- which is illustrated in the **resource allocation graph**

# Subtle Coding Styles

## Deadlock free

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

## Deadlock

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

- [Figure 6-2 in Tanenbaum & Bos, 2014]

# Remarks

- Whether the deadlock happens or not depends on the result of a race (or scheduling)
  - Difficult to debug because it only happens sporadically
- Difference between deadlock free and deadlocked code is subtle in coding style

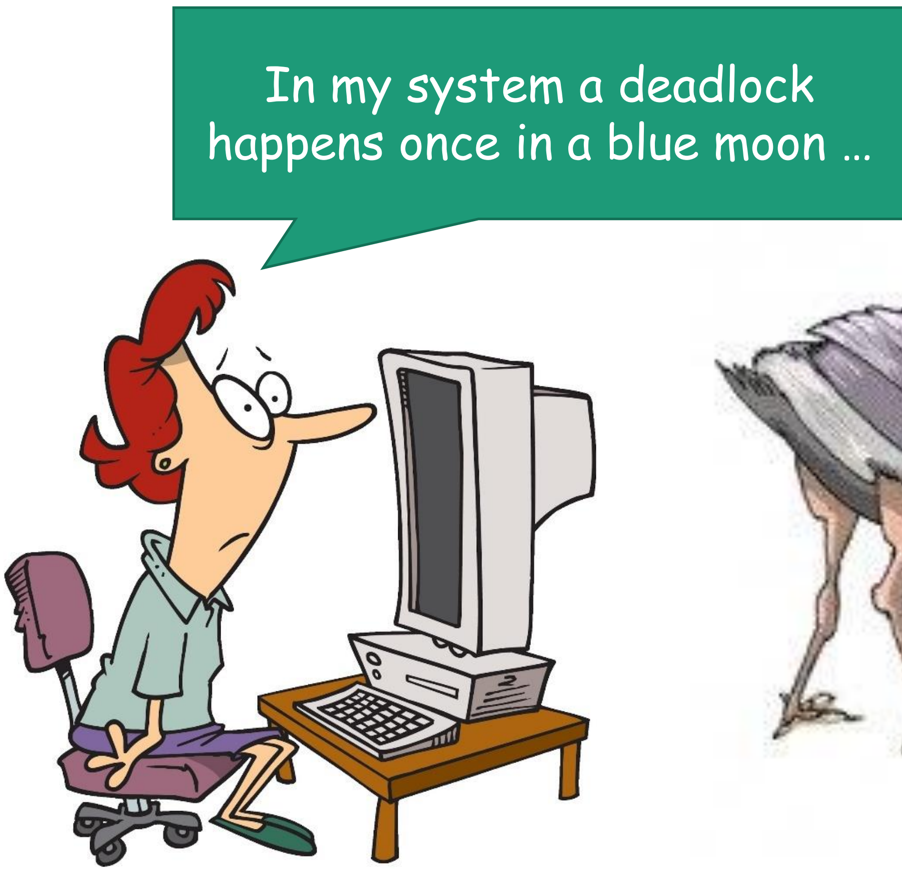
# Questions?

- Synchronization tools are resources
- Subtle to write deadlock-free code, and difficult to debug
- How do we deal with deadlocks?

# Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention (by structurally negating one of the four required conditions)
  - Deadlock avoidance (by carefully allocating resources)
- Allow the system to enter a deadlock state and then recover
  - Deadlock detection and recovery (Let deadlocks occur, detect them, and then take action)
- Ignore the problem and pretend that deadlocks never occur in the system.
  - The **Ostrich algorithm**

# The Ostrich Algorithm




# Questions?

- System Model
- Deadlock in Multithreaded Applications
- Deadlock Characterization and Resource Allocation Graph
- Methods for Handling Deadlocks
  - Presentation, avoidance, detection & recovery
  - The Ostrich Algorithm