

Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1400-1 semester

Zeinab Zali

Session 19: Deadlock characterization and prevention





System Model

- System consists of resources
- Resource types R_1, R_2, \dots, R_m

Examples: 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**

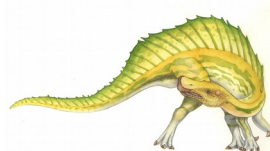




Deadlock in Multithreaded Application

- Two mutex locks are created and initialized:

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





Deadlock in Multithreaded Application

```
/* 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);

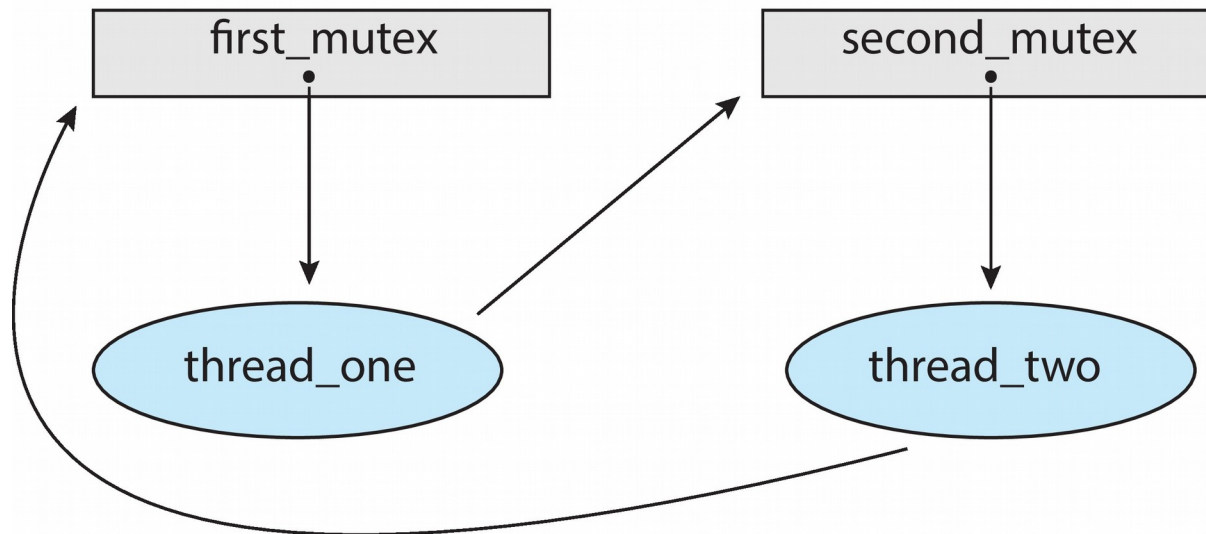
    pthread_exit(0);
}
```





Deadlock in Multithreaded Application

- Deadlock is possible if 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`.
- Can be illustrated with a **resource allocation graph**:





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 $\{P_0, P_1, \dots, P_n\}$ 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 .

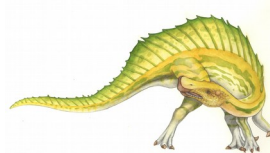
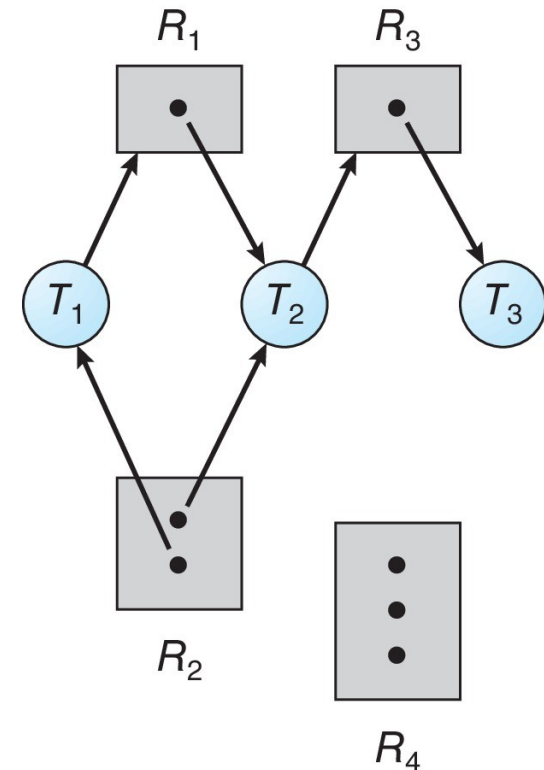
- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed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





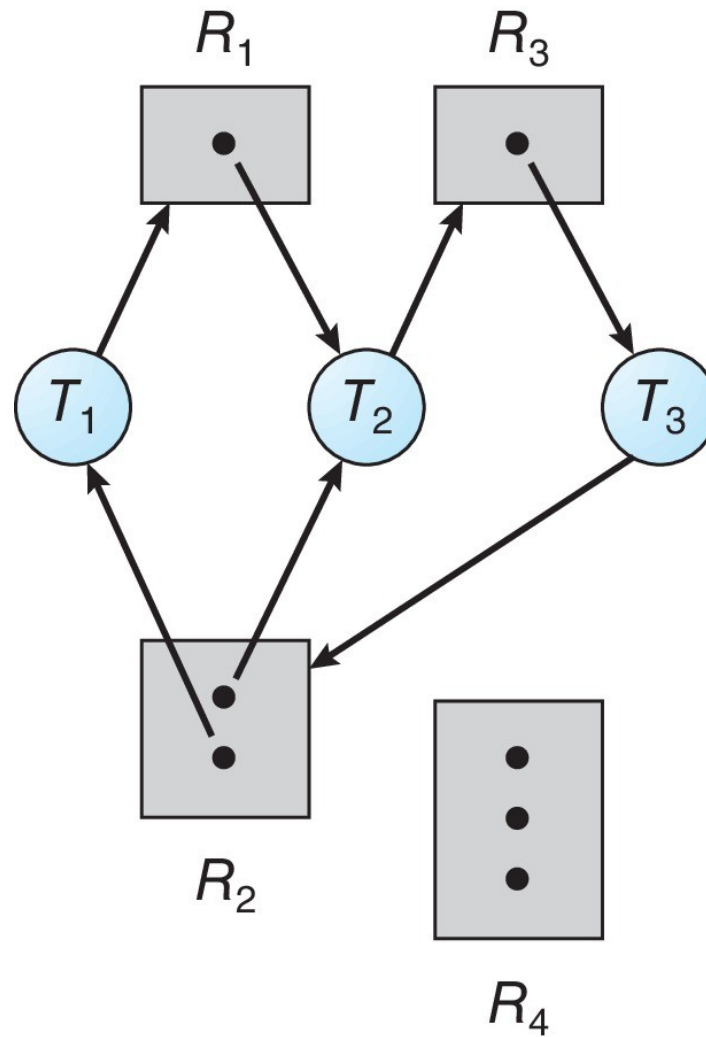
Resource Allocation Graph Example

- 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



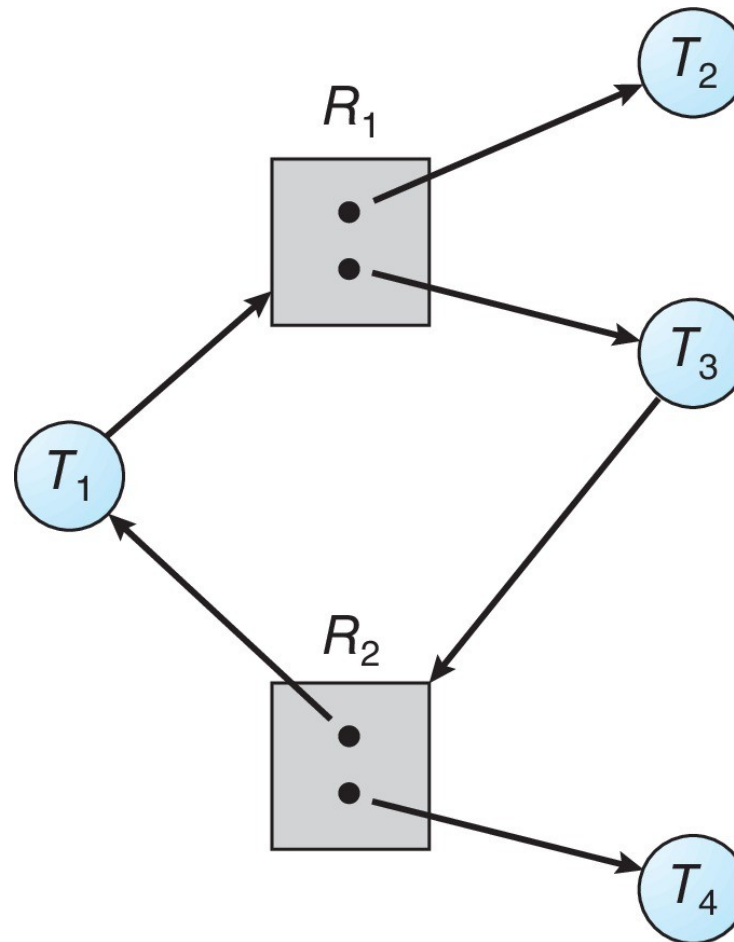


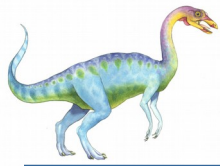
Resource Allocation Graph With A Deadlock





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 پیشگیری
 - Deadlock avoidance اجتناب
- Allow the system to enter a deadlock state and then recover (**detect and recover**)
 - Rollback? Is it useful? Is it possible?
- Ignore the problem and pretend that deadlocks never occur in the system.
 - **Ostrich**



Detect and recover





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.
 - Low resource utilization; starvation possible



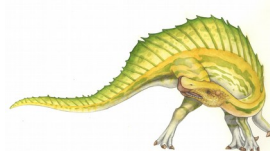


Deadlock Prevention (Cont.)

■ No Preemption –

- 1) 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
- 2) Preempted resources are added to the list of resources for which the process is waiting
- 3) 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



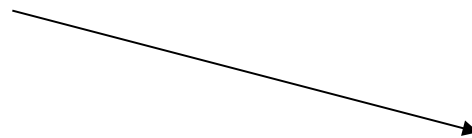


Circular Wait

- 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:

first_mutex = 1
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);
}
```





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);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers \$25 from account A to account B, and Transaction 2 transfers \$50 from account B to account A

