KOÇ UNIVERSITY

Deadlocks

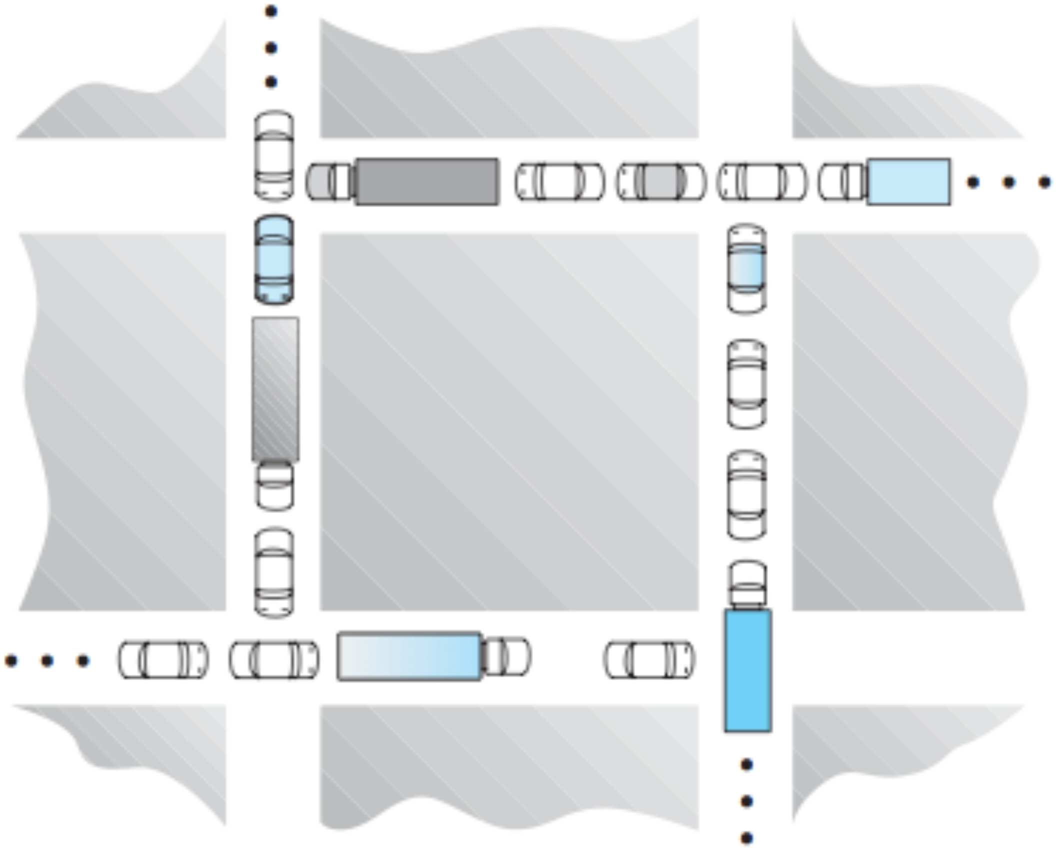
Didem Unat

Lecture 15

COMP304 - Operating Systems (OS)

Traffic Example

- Traffic only in one direction.
- Each street can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



Mutex Code Example

```
/* thread-1 runs in this function */
void *do_work_one(void *param) {
    mutex_lock(&first_mutex);
    mutex_lock(&second_mutex);

    /* Do some work */

    mutex_unlock(&second_mutex);
    mutex_unlock(&first_mutex);
}
```

```
/* thread-2 runs in this function */
void *do_work_two(void *param) {
    mutex_lock(&second_mutex);
    mutex_lock(&first_mutex);

    /* Do some work */

    mutex_unlock(&first_mutex);
    mutex_unlock(&second_mutex);
}
```

In this example, thread one attempts to acquire the mutex locks in the order (1) first mutex, (2) second mutex, while thread two attempts to acquire the mutex locks in the order (1) second mutex, (2) first mutex**. Deadlock** is possible if thread one acquires first mutex while thread two acquires second mutex.

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  Ex. 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

The deadlock problem: A set of blocked processes each holding a resource and waiting to acquire another resource held by another process in the set.

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 is waiting for a resource that is held by P_0.

Resource-Allocation Graph

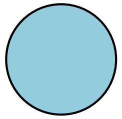
Deadlocks can be described in terms of a directed graph

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

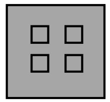
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge:** directed edge $P_i \rightarrow R_j$
- **assignment edge:** directed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

Resource-Allocation Graph

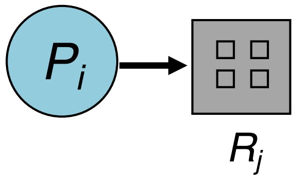
- Process



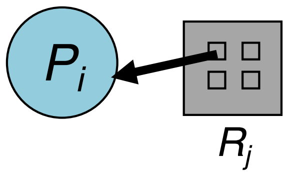
- Resource type with 4 instances



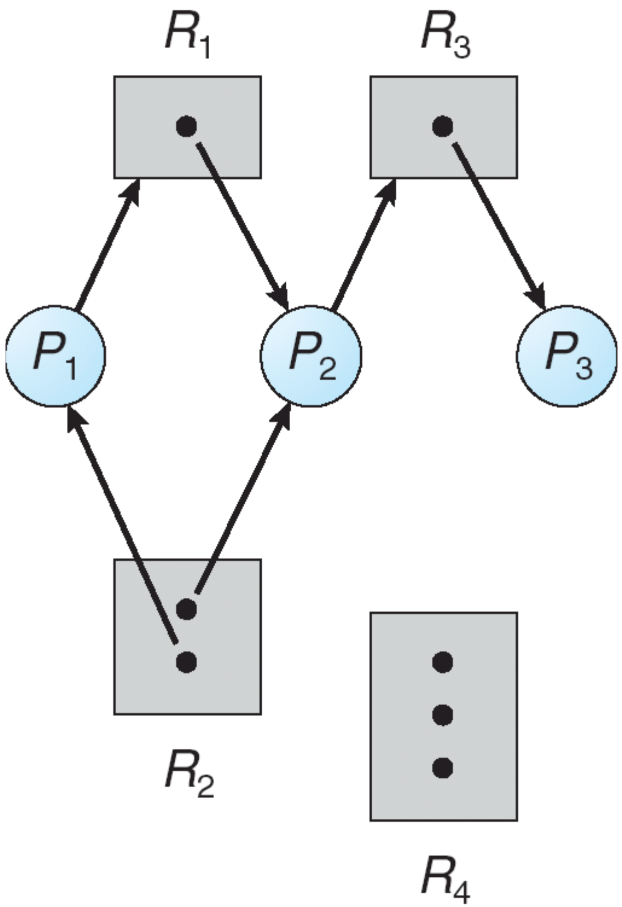
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example Resource Allocation Graph



Resource instances:

◦ One instance of resource type *R1*

◦ Two instances of resource type *R2*

◦ One instance of resource type *R3*

◦ Three instances of resource type *R4*

Process states:

◦ Process *P1* is holding an instance of resource type *R2* and is waiting for an instance of resource type *R1*.

◦ Process *P2* is holding an instance of *R1* and an instance of *R2* and is waiting for an instance of *R3*.

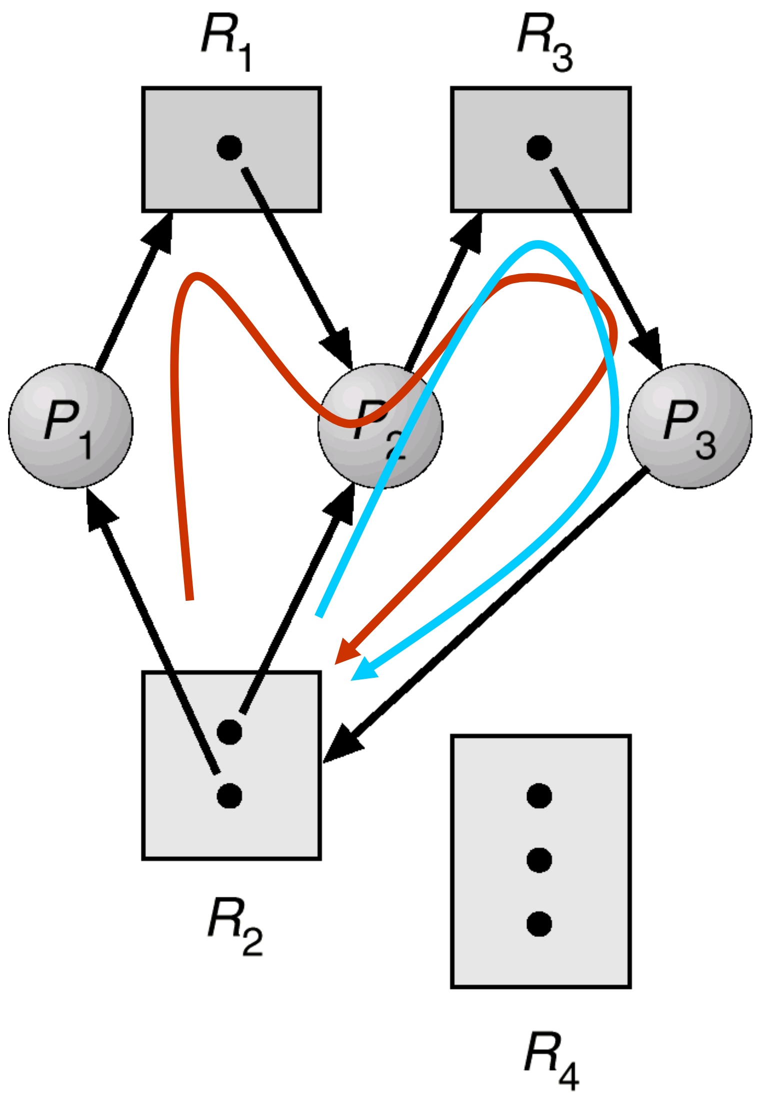
◦ Process *P3* is holding an instance of *R3*.

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.

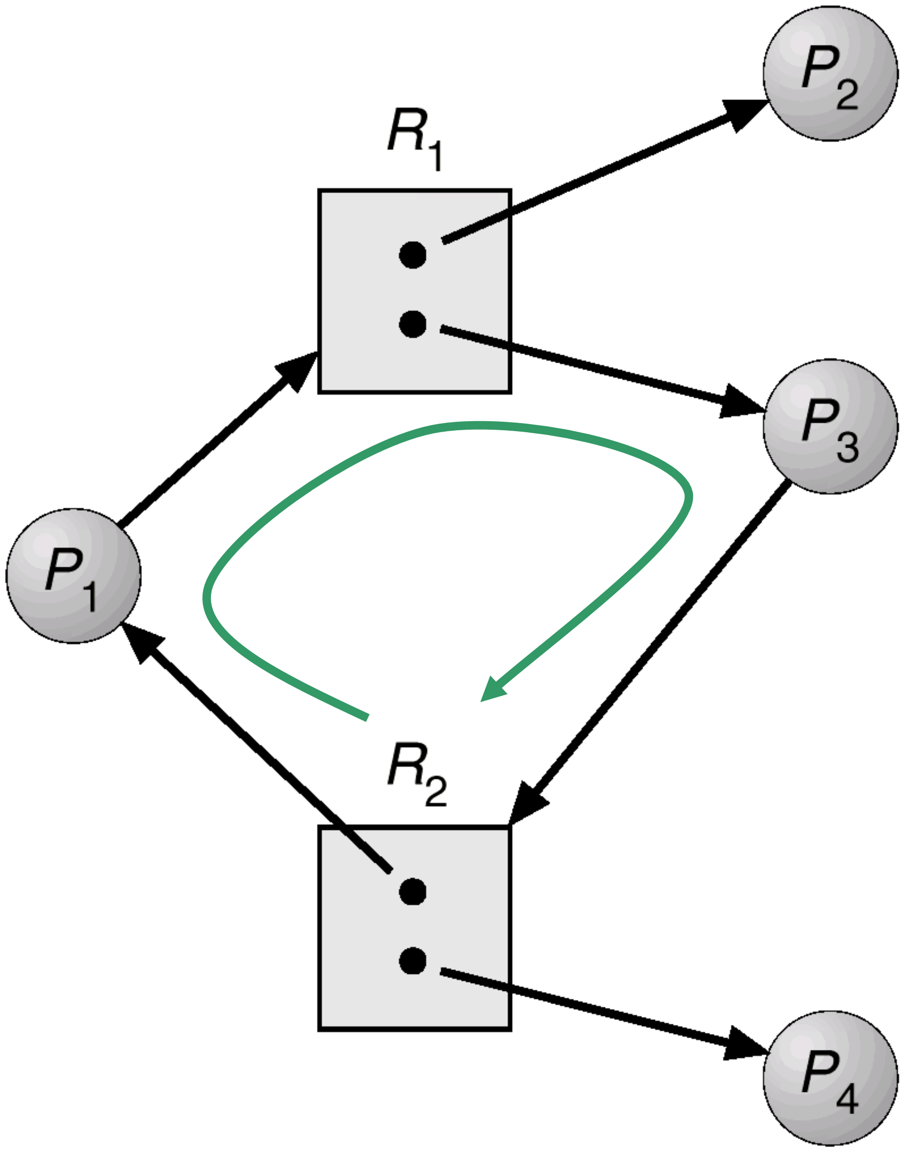
Resource Allocation Graph-1



A resource allocation graph with a cycle

Is there a deadlock?

Resource Allocation Graph-2



A resource allocation graph with a cycle

Is there a deadlock?

Deadlock Examples

- Graph-1
  - Deadlock because P1, P2 and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or P2 to release resource R2. In addition process P1 is waiting for process P2 to release resource R1

- Graph-2
  - No deadlock
  - P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state.

Deadlock prevention or avoidance

- Allow the system to enter a deadlock state and then recover.

Deadlock detection and recovery

- Ignore the problem and pretend that deadlocks never occur in the system

Which one of these methods is used most commonly?

Deadlock Prevention

Methods for ensuring that at least one of the four conditions cannot hold

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.

- **Hold and Wait** –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.
  - Dining Philosophers' problem: Have both chopsticks otherwise put down the chopstick
  - Low resource utilization; starvation possible.

Deadlock Prevention

- **No Preemption**
  - If a process that is holding some resource requests another resource that cannot be immediately allocated to it, then all resources currently being held are released (<u>preempted</u>).
  - 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 <u>each process requests resources in an increasing order of enumeration</u>.

Total Ordering of Locks

```
/* thread one runs in this function */
void *do_work_one(void *param) {
    mutex_lock(&first_mutex);
    mutex_lock(&second_mutex);

    /* Do some work */

    mutex_unlock(&second_mutex);
    mutex_unlock(&first_mutex);
}
```

```
/* thread two runs in this function */
void *do_work_two(void *param) {
    mutex_lock(&second_mutex);
    mutex_lock(&first_mutex);

    /* Do some work */

    mutex_unlock(&first_mutex);
    mutex_unlock(&second_mutex);
}
```

BSD operating system uses a lock-order verifier, known as **witness**.

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex. Witness records the relationship that first mutex must be acquired before second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

Example in the Handout

```
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);
}
```

Deadlock Prevention- Summary

- Ensure that at least one of the four conditions cannot hold
- Conservative approach
- Prevention leads to
  - Low utilization of devices
  - Low throughput
  - Frequent starvation

- Alternative method is deadlock avoidance
  - Requires additional information about how resources to be requested

Acknowledgments

- These slides are adapted from
  - Öznur Özkasap (Koç University)
  - Operating System and Concepts (9th edition) Wiley