Operating Systems Concepts

Dead Locks

CS 4375, Fall *2025*
Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu
November 10, 2025

Summary

- Process Synchronization
  - Multiprogramming: multiple processes or threads may require consensus to work together
    - How do we achieve that? What issues may occur for shared data?
  - Race Condition
  - Critical-Section Problem
    - Different Solutions

Agenda

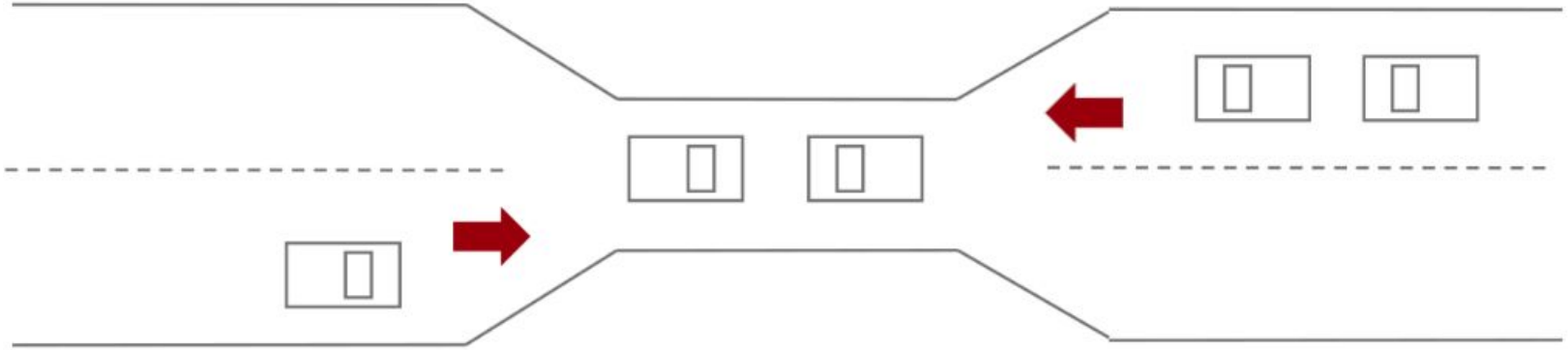
- The Deadlock Problem
  - Characterization of Deadlock
  - Resource Allocation Graph
  - Handling Deadlocks
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection
    - Deadlock Recovery

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example:
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs the other one.
- Example:
  - Semaphores A and B, initialized to 1

P_1	P_2
`wait(A);`	`wait(B);`
`wait(B);`	`wait(A);`

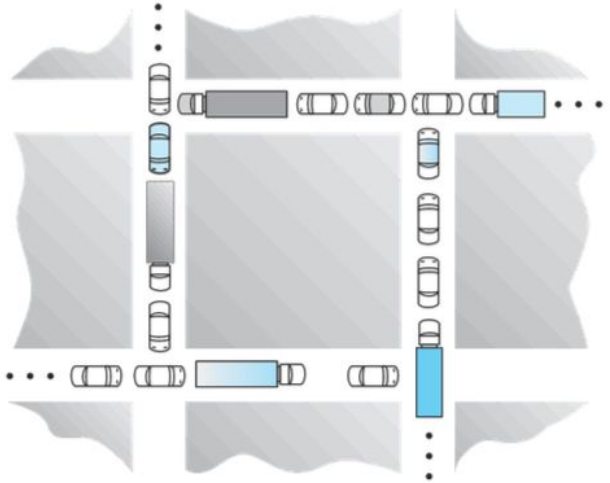
Deadlock Example: Bridge Crossing



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

Deadlock vs. Starvation

- Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes



- Starvation: Indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Deadlock Characterization

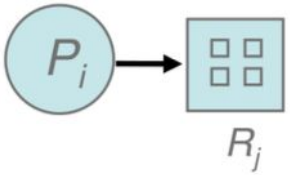
- Deadlock can arise if four conditions hold simultaneously
  - Mutual exclusion: Nonshared resources; only one process at a time can use a specific resource
  - Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption: 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 $\{P_1, P_2, ..., P_n\}$ of waiting processes such that P_1 is waiting for a resource that is held by P_2 , P_2 is waiting for a resource that is held by P_3 , ..., 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_1

Resource Allocation Graph

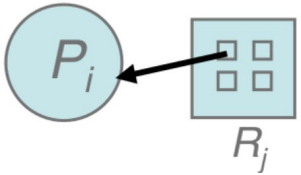
- Process

- Resource type with 4 instances

- P_i requests instance of R_j



- P_i is holding an instance of R_j



Resource Allocation Graph Example

- Semaphores A and B, initialized to 1

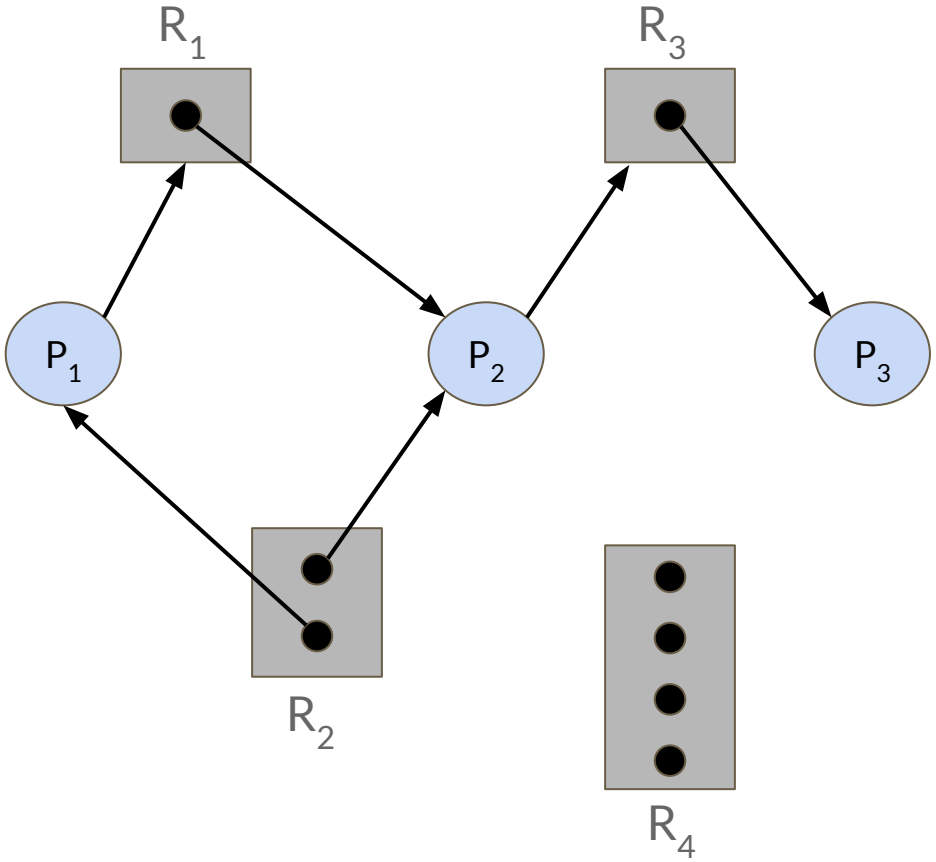
P_1

```
wait(A);
wait(B);
```

P_2

```
wait(B);
wait(A);
```

Resource Allocation Graph Example

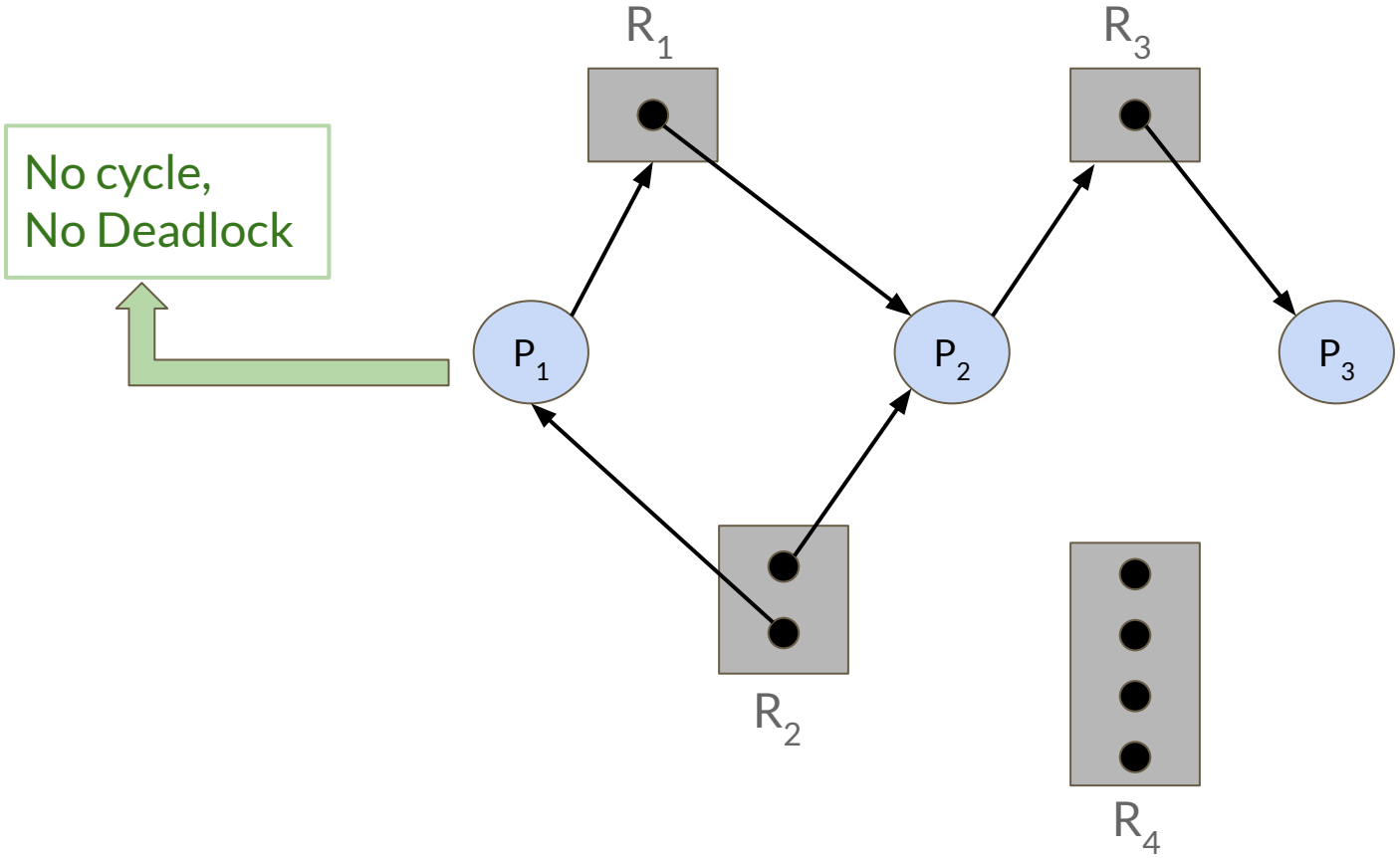


Resource Allocation Graph

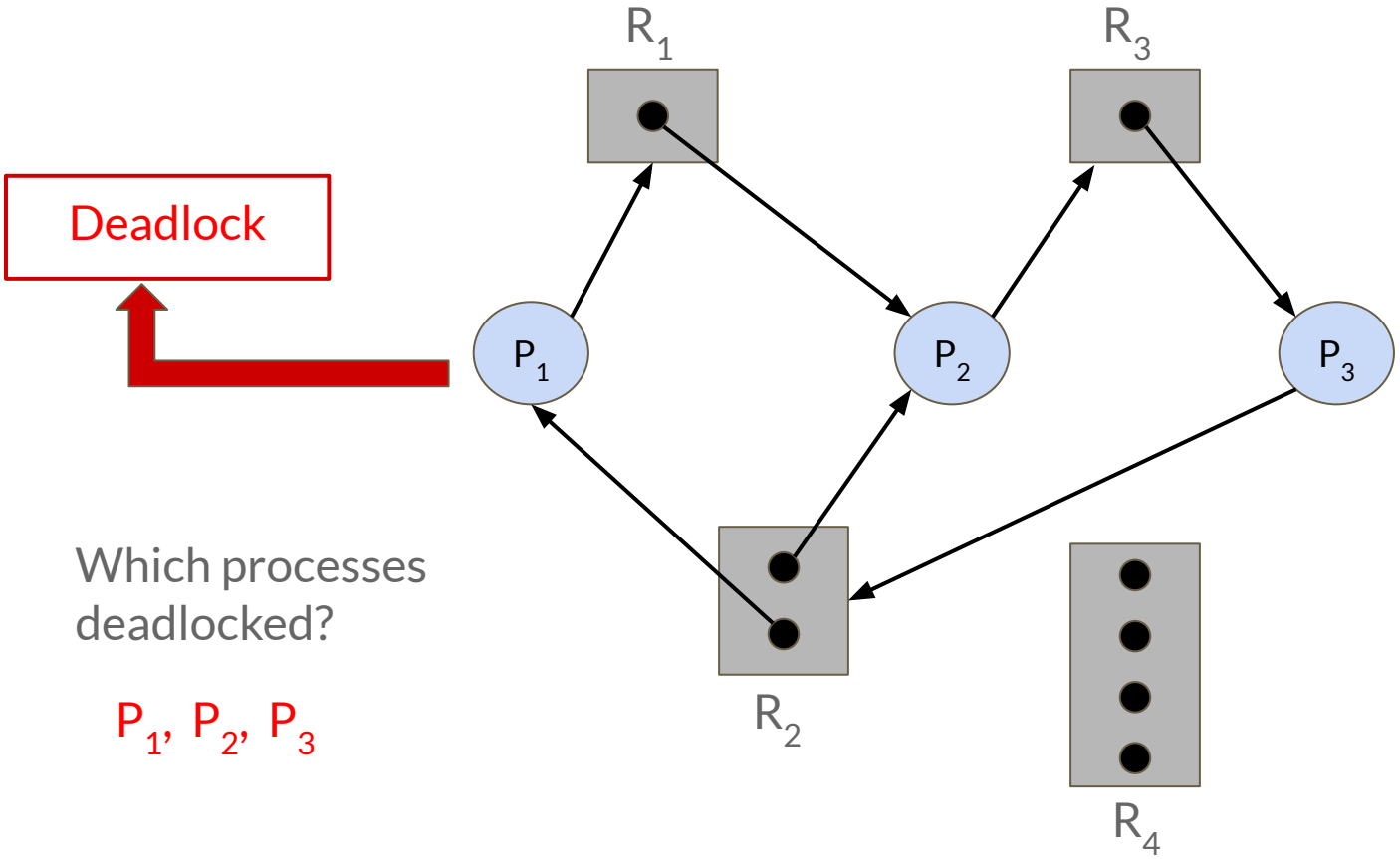
- If graph contains no cycles → No deadlock

- If graph contains a cycle → There may be a deadlock
  - Only one instance per resource type
    - Deadlock
  - Several instances per resource type
    - Possibility of deadlock

Resource Allocation Graph Example



Resource Allocation Graph Example

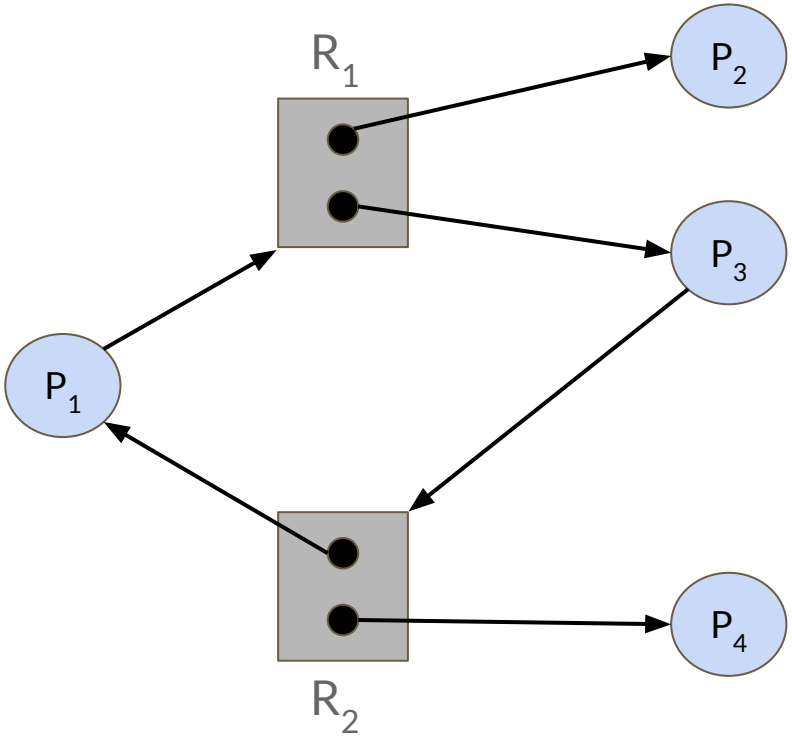


Which processes deadlocked?

P_1, P_2, P_3

Resource Allocation Graph Example

Cycle, but no deadlock



Resource Allocation Graph

- A cycle in the resource allocation graph:
  - Is a necessary condition for a deadlock
  - Is not a sufficient condition for a deadlock

Exercise

- In the code below, three processes are competing for six resources A to F
  - Using a resource allocation graph show the possibility of a deadlock in this implementation.

```
void P1() {
     while (true) {
          get(A);
          get(B);
          get(C);
          // critical region
          // use A, B, C
          release(A);
          release(B);
          release(C);
     }
}
```

```
void P2() {
     while (true) {
          get(D);
          get(E);
          get(B);
          // critical region
          // use D, E, B
          release(D);
          release(E);
          release(B);
     }
}
```

```
void P3() {
     while (true) {
          get(C);
          get(F);
          get(D);
          // critical region
          // use C, F, D
          release(C);
          release(F);
          release(D);
     }
}
```

Handling Deadlocks

- Ignore the problem altogether and pretend that deadlocks never occur in the system.
  - Programmers should handle deadlocks (Linux, Windows)
- Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
  - Deadlock Prevention
    - Provides a set of methods to ensure that at least one of the necessary conditions cannot hold
  - Deadlock Avoidance
- Allow the system to enter a deadlocked state, detect it, and recover.
  - Deadlock Detection
  - Deadlock Recovery

Deadlock Prevention

- Ensure one of the deadlock conditions cannot hold
- Mutual exclusion: Not required for sharable resources. Must hold for nonsharable resources.
    - Example:

      Accessing read-only files does not require mutual exclusion.

Mutual exclusion is inevitable

Deadlock Prevention

- Ensure one of the deadlock conditions cannot hold
- Hold and wait: Must guarantee that whenever a process requests a resource, it does not hold any other resources.

A. Require process to request and be allocated all its resources before it begins.

B. Allow process to request resources only when the process has none.

- Example:
- Reading from DVD to memory and then printing.
- A.1. Holds the printer unnecessarily for the entire execution
  - Low resource utilization
- A.2. May never get all resources
  - Possible starvation
- B. May lose its data

Deadlock Prevention

- Ensure one of the deadlock conditions cannot hold
- 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. Alternatively, we can preempt them only if needed by another process.
    - 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.

This cannot be generally applied to all types of resources (CPU vs mutex)

Deadlock Prevention

- Ensure one of the deadlock conditions cannot hold
- Circular wait: Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.
  - What to do with out of order requests?
  - How to choose the order?
  - What if order changes?

Exercise

- In the code below, three processes are competing for six resources **A to F**
  - Modify the order of some of the **get** requests to prevent the possibility of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

```
void P1() {
     while (true) {
          get(A);
          get(B);
          get(C);
          // critical region
          // use A, B, C
          release(A);
          release(B);
          release(C);
     }
}
```

```
void P2() {
     while (true) {
          get(D);
          get(E);
          get(B);
          // critical region
          // use D, E, B
          release(D);
          release(E);
          release(B);
     }
}
```

```
void P3() {
     while (true) {
          get(C);
          get(F);
          get(D);
          // critical region
          // use C, F, D
          release(C);
          release(F);
          release(D);
     }
}
```

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- Farshad Ghanei from Illinois Tech
- T. Kosar and K. Dantu from University at Buffalo

Announcement

- Homework 4
  - Due on Today at 11.59PM
- Homework 5
  - Will be released tomorrow