CSE 421/521 - Operating Systems
Fall 2018

Lecture - X
Deadlocks - I

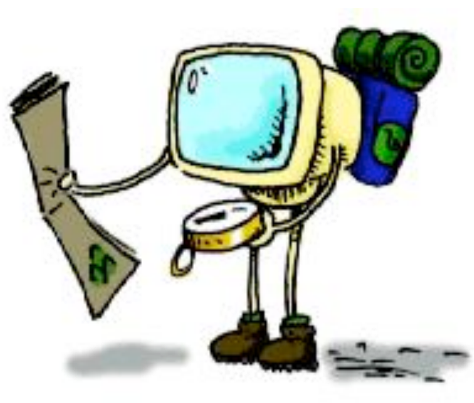
Tevfik Koşar

University at Buffalo

October 2nd, 2018

Roadmap

- The Deadlock Problem
  - Characterization of Deadlock
  - Resource Allocation Graph
  - Deadlock Prevention

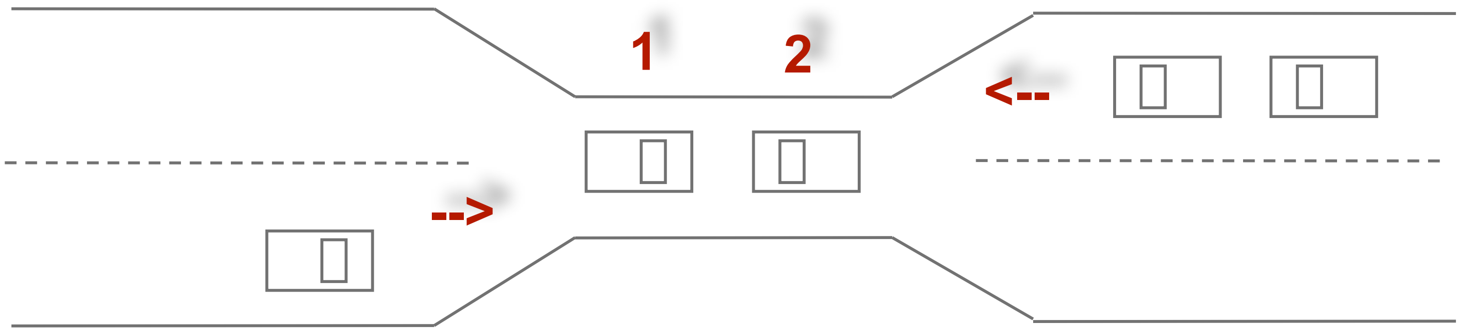


The Deadlock Problem

- 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 another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

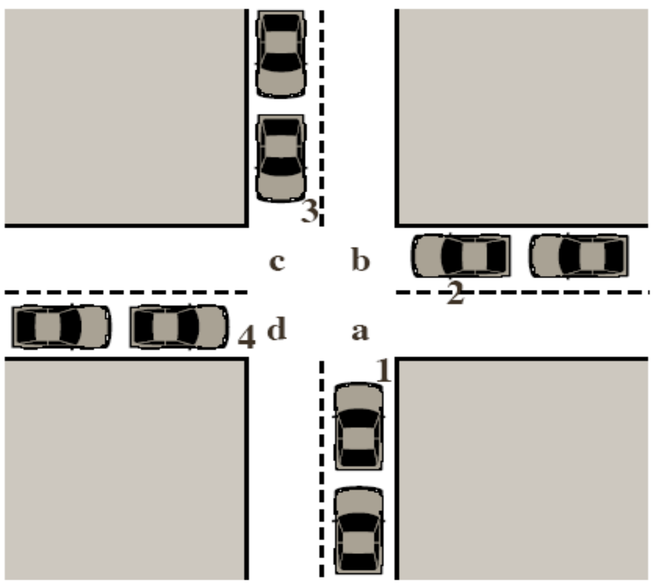
Bridge Crossing Example



- 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 resources 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



(a) Deadlock possible

- 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.

1. **Mutual exclusion:** nonshared resources; only one process at a time can use a specific resource
2. **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
3. **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task

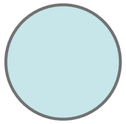
Deadlock Characterization (cont.)

Deadlock can arise if four conditions hold simultaneously.

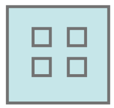
4. **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph (Cont.)

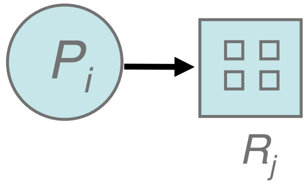
- Process



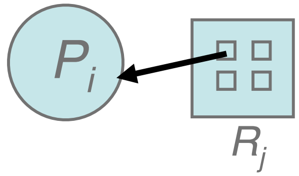
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j

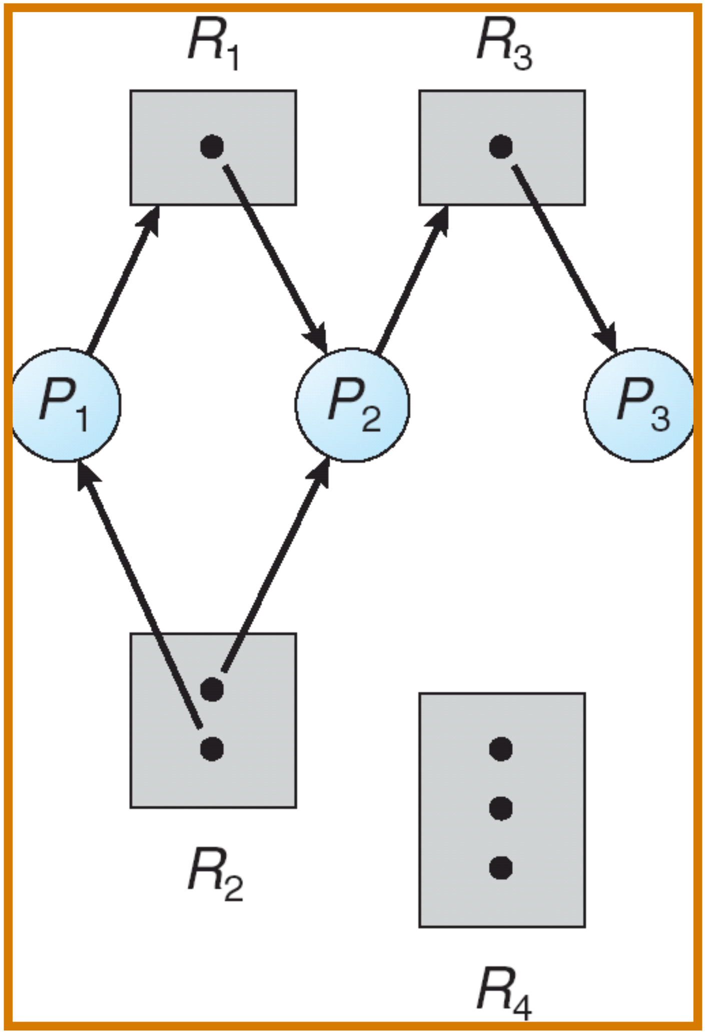


Example

- semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

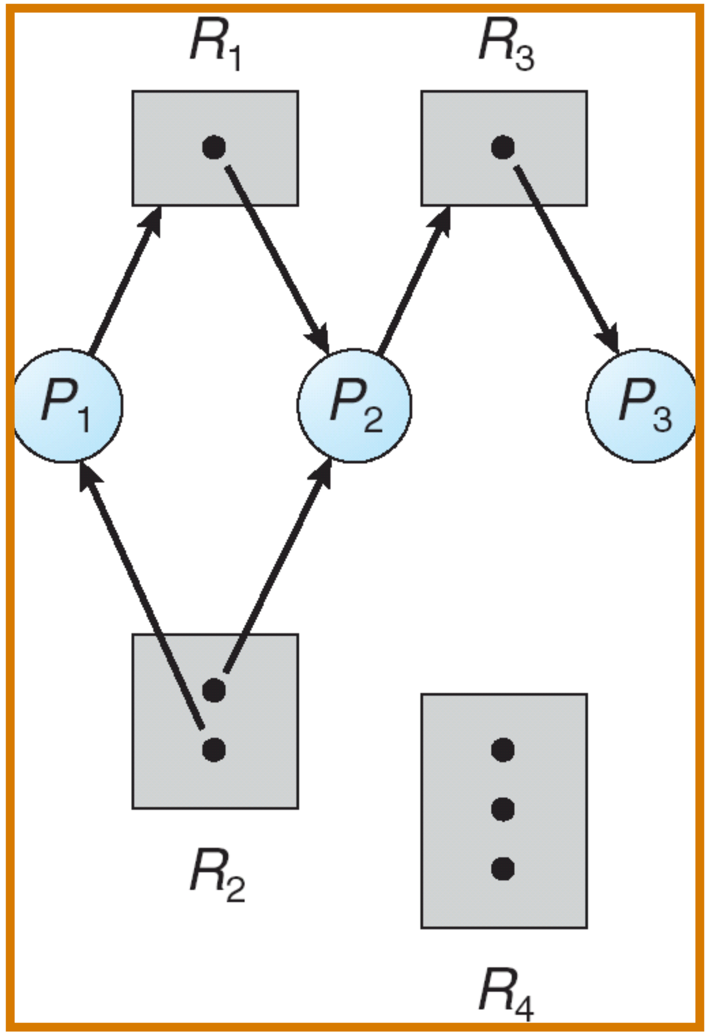
Example of a Resource Allocation Graph



Basic Facts

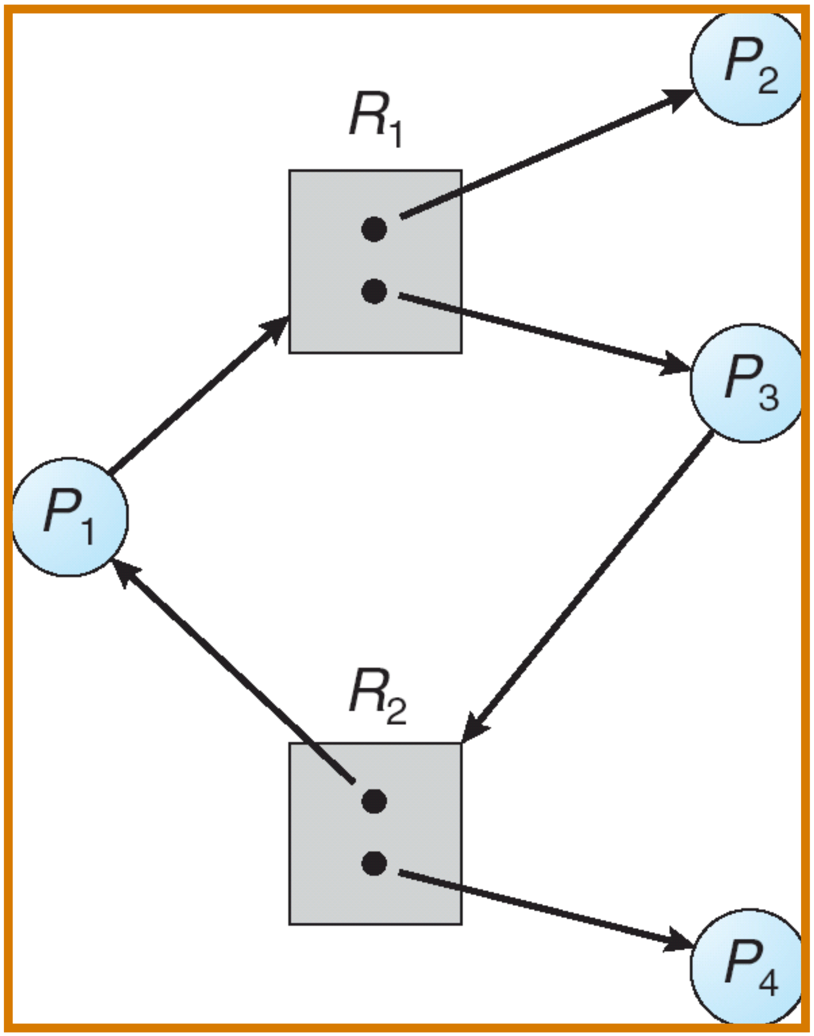
- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$ there may be a deadlock
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Resource Allocation Graph – Example 1



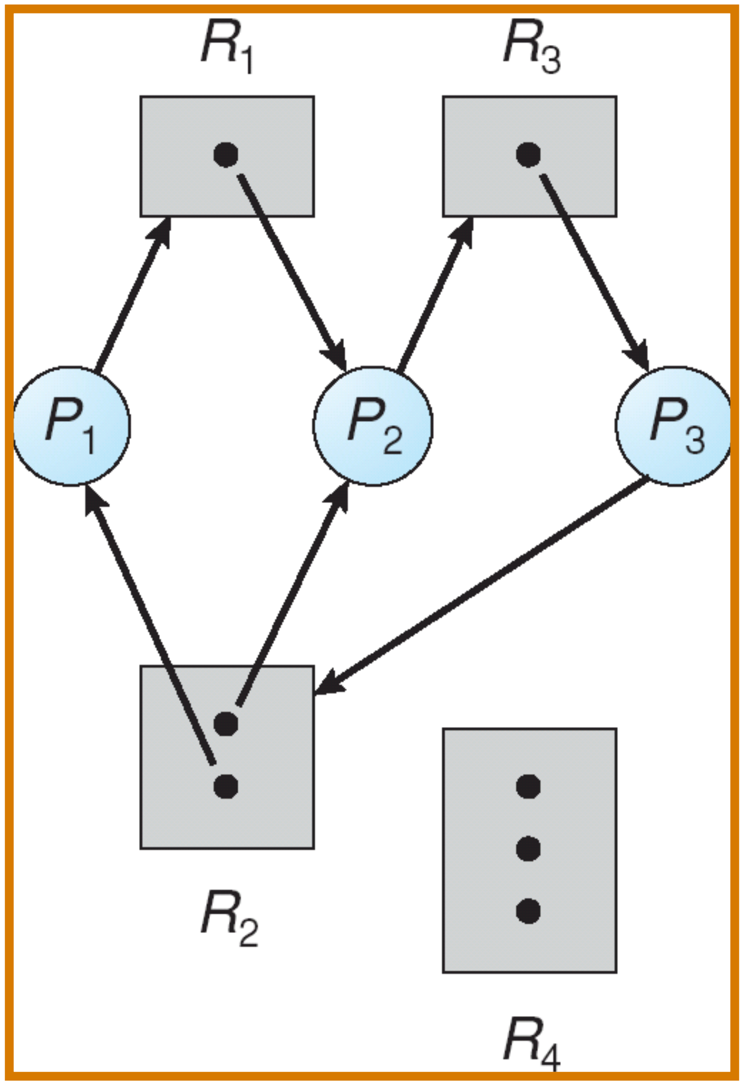
➔ No Cycle, no Deadlock

Resource Allocation Graph – Example 2



➔ Cycle, but no Deadlock

Resource Allocation Graph – example 3



➔ Deadlock

Which Processes deadlocked?

➔ P1 & P2 & P3

Rule of Thumb

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

Exercise

In the code below, three processes are competing for six resources labeled A to F.

a. Using a resource allocation graph (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

`void P0()`
`{`
`  while (true) {`
`    get(A);`
`    get(B);`
`    get(C);`
`    // critical region:`
`    // use A, B, C`
`    release(A);`
`    release(B);`
`    release(C);`
`  }`
`}`
	`void P1()`
	`{`
	`  while (true) {`
	`    get(D);`
	`    get(E);`
	`    get(B);`
	`    // critical region:`
	`    // use D, E, B`
	`    release(D);`
	`    release(E);`
	`    release(B);`
	`  }`
	`}`
		`void P2()`
		`{`
		`  while (true) {`
		`    get(C);`
		`    get(F);`
		`    get(D);`
		`    // critical region:`
		`    // use C, F, D`
		`    release(C);`
		`    release(F);`
		`    release(D);`
		`  }`
		`}`

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
- Ignore the problem and pretend that deadlocks never occur in the system
  - ➔ Programmers should handle deadlocks (UNIX, Windows)

Deadlock Prevention

➔ Ensure one of the deadlock conditions cannot hold

➔ Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
  - Eg. read-only files

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  1. Require process to request and be allocated all its resources before it begins execution
  2. or allow process to request resources only when the process has none.

  Example: Read from DVD to memory, then print.
  1. holds printer unnecessarily for the entire execution
     - Low resource utilization
  2. may never get the printer later
     - 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.

Exercise *(cont.)*

In the code below, three processes are competing for six resources labeled A to F.

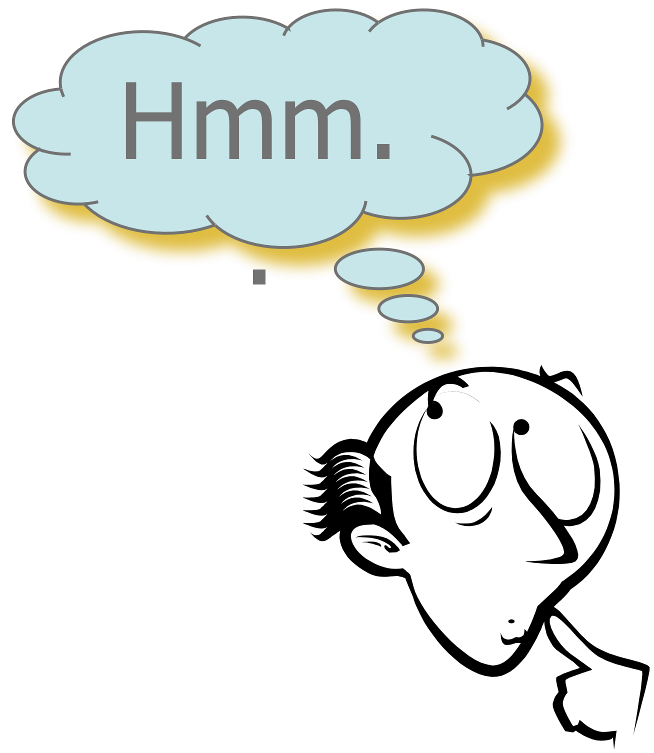
a. <u>Using a resource allocation graph</u> (Silberschatz pp.249-251) show the possiblity of a deadlock in this implementation.

b. Modify the order of some of the `get` requests to prevent the possiblity 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.

<pre>void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}</pre>
	<pre>void P1()
	{
	  while (true) {
	    get(D);
	    get(E);
	    get(B);
	    // critical region:
	    // use D, E, B
	    release(D);
	    release(E);
	    release(B);
	  }
	}</pre>
		<pre>void P2()
		{
		  while (true) {
		    get(C);
		    get(F);
		    get(D);
		    // critical region:
		    // use C, F, D
		    release(C);
		    release(F);
		    release(D);
		  }
		}</pre>

Summary

- The Deadlock Problem
  - Characterization of Deadlock
  - Resource Allocation Graph
  - Deadlock Prevention



- Next Lecture: Deadlocks - II

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 UNR