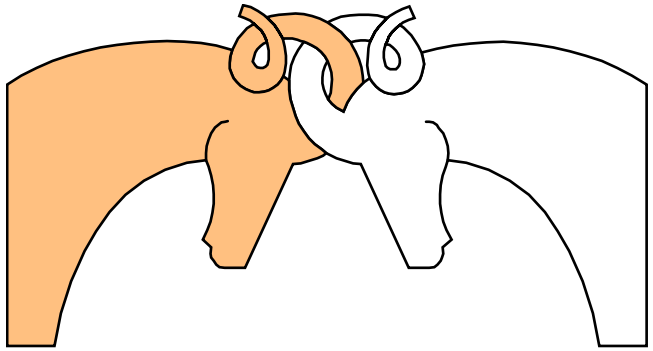
CIS 520, *Operating Systems Concepts*

Lecture 4

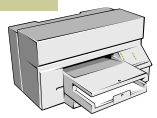
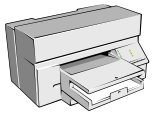
Deadlocks (Part 1 of 2)



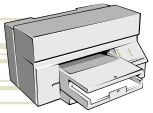
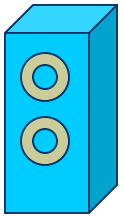
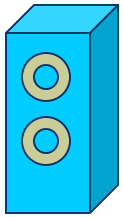
Again, What is a Resource?

- Something a process can use
  - Disc, tape, a communications channel, a chunk of memory, a CPU...
- It can be *shareable* (e.g., read-only memory or read-only file) or *non-shareable* (e.g., a read/write memory)
- Resources are naturally grouped by *resource types*, within which they are indistinguishable and interchangeable

Examples of *Instances* of a Resource Type

- Three *printers* in the same room (but, probably not three printers in different buildings)

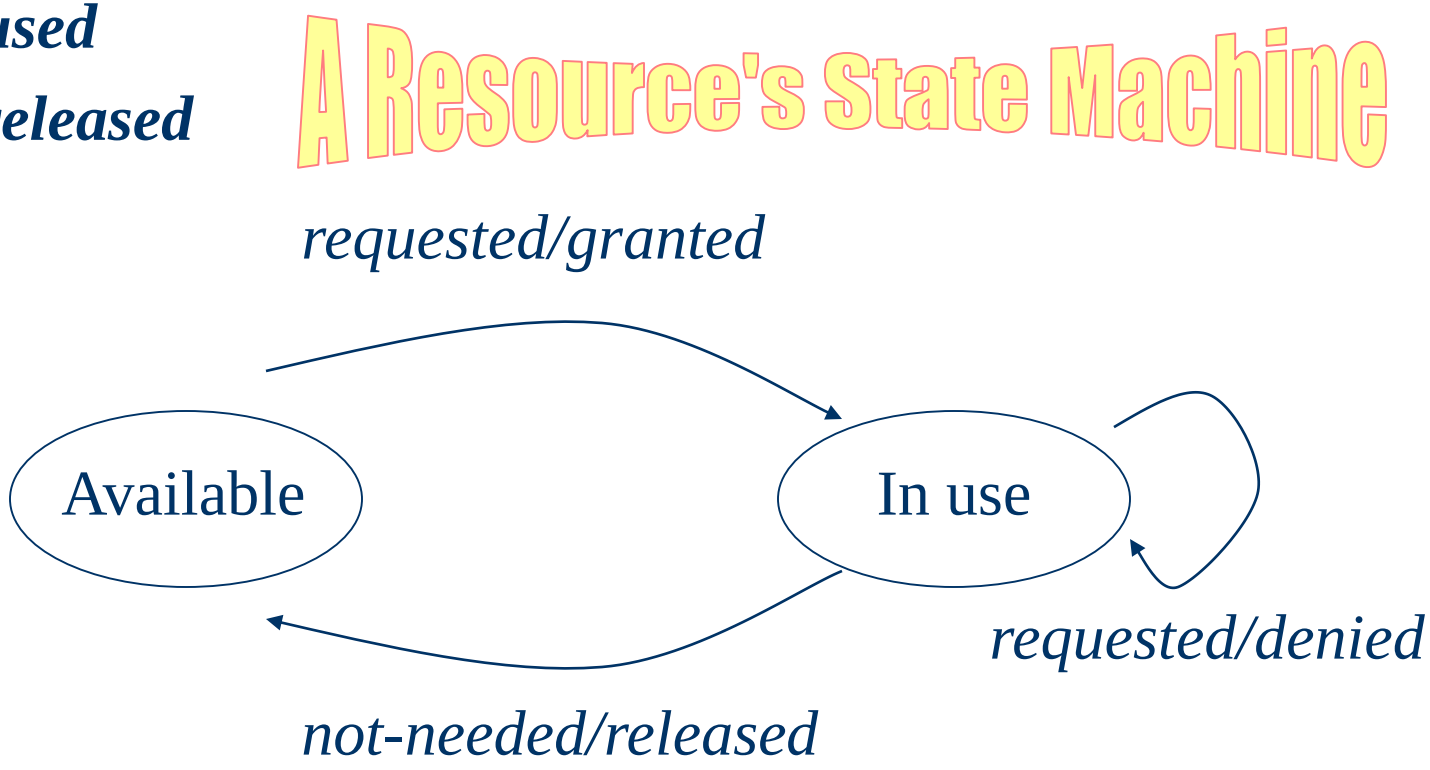
- Two *tape drives*
- A *cell* in an array in the *Producer/Consumer* problem

		*	!	@	#	$	%	&		

- A *customer* in the *Sleeping Barber* problem

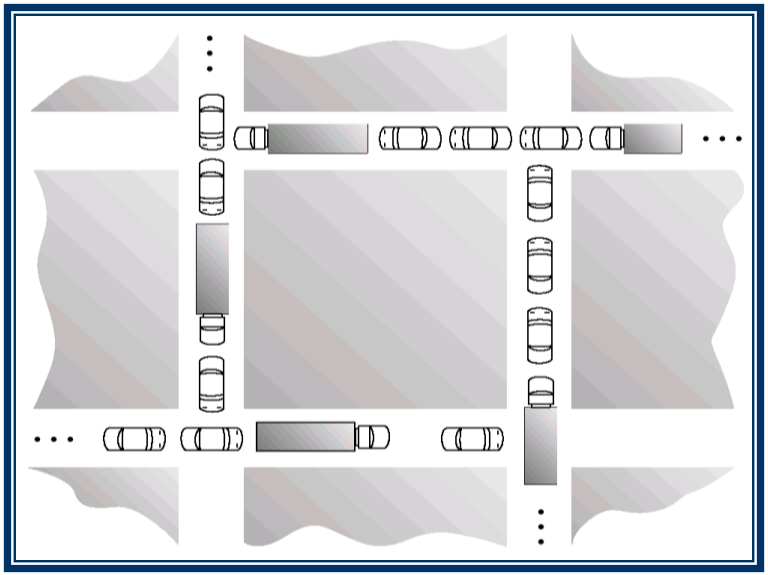
What is Done with a Resource?

- **It is *requested***
- **It is *used***
- **It is *released***



Deadlock (definition)

- A set of processes is in a *deadlock*, if **each** process is waiting for an event that can be caused only by another process in the set



Another Example

1. The system has one disk and one tape drive
2. One process requests the disk and gets it
3. Another process requests the tape and gets it
4. The first process is requesting the tape now and waiting for it, but...
5. The second process is requesting the disk and waiting for it...

A Step Aside: *Necessary* and *Sufficient*

$$A \rightarrow B \qquad (\overline{A} \vee B)$$

A is ***sufficient*** for **B**

and

B is ***necessary*** for **A**

Examples

- For ***n*** to be divisible by 4 it is *necessary* that ***n*** be even (but is it *sufficient*?)
- For ***n*** to be divisible by 3 it is **neither** *necessary* **nor** *sufficient* that ***n*** be even
- Having a passing grade in the OS course is *necessary* for graduating (but is it *sufficient?*)
- Studying deadlocks is *necessary* for getting a passing grade in the OS course (but is it *sufficient*)
- Knowing everything in the book is *sufficient* for passing the OS course (but is it *necessary*?)

Necessary and Sufficient

Of course, $A \rightarrow B$ does not mean that $B \rightarrow A$,

but if both $A \rightarrow B$ and $B \rightarrow A$ hold, then

A is *necessary and sufficient for* **B**, and vice versa.

- Thus, **learning Operating Systems** is *necessary* **and** *sufficient* for achieving happiness in life

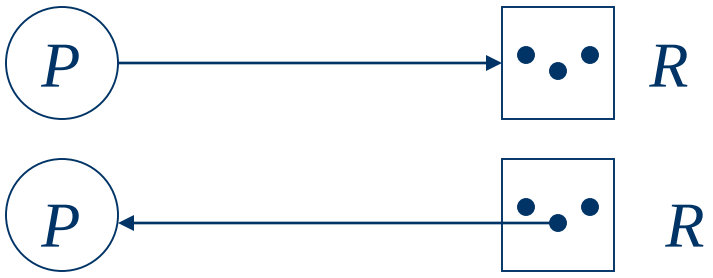
Four *Necessary* Conditions for Deadlock

1. ***Mutual Exclusion:*** At least one resource must be held in a non-shareable mode
2. ***Hold and Wait:*** Each process must hold *at least one* resource while waiting for another resource
3. ***No Preemption:*** Only a process that holds a resource can release it
4. ***Circular Wait:*** $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow \ldots \rightarrow P_n$

Let us check one-by-one if any (or all) is ***sufficient***

A Tool: *Resource Allocation Graph*

- Is a directed graph G = (*V*, *E*)
  - There two types of vertices: *Processes* $\{P_i\}_{i=1}^{n}$ and *Resources* $\{R_i\}_{i=1}^{m}$.
  - Edges **always** connect processes and resources (such graphs are called *bipartite*):

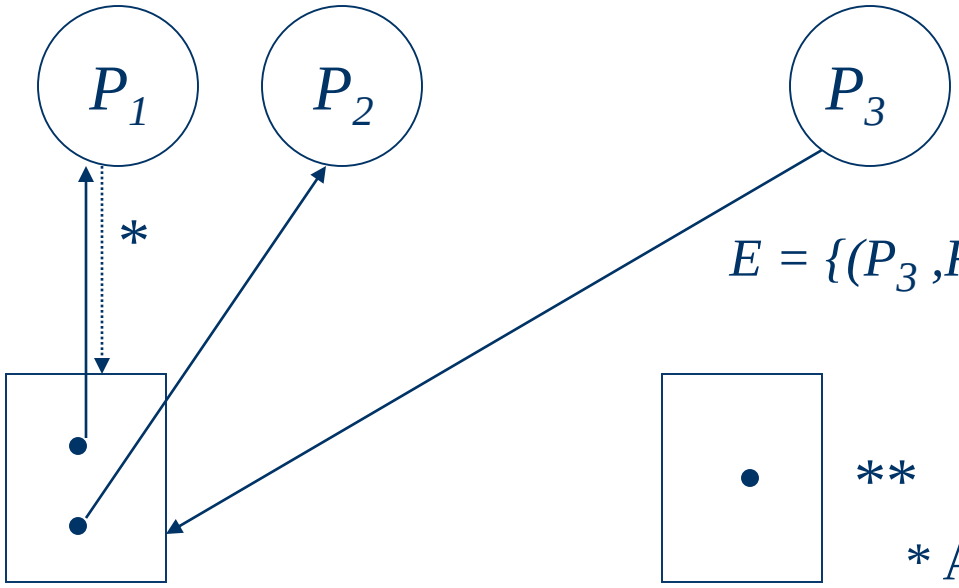


A *request* edge (*P* is waiting)

An *assignment* edge

An Example

$V = \{P_1, P_2, P_3\} \cup \{R_1, R_2\}$

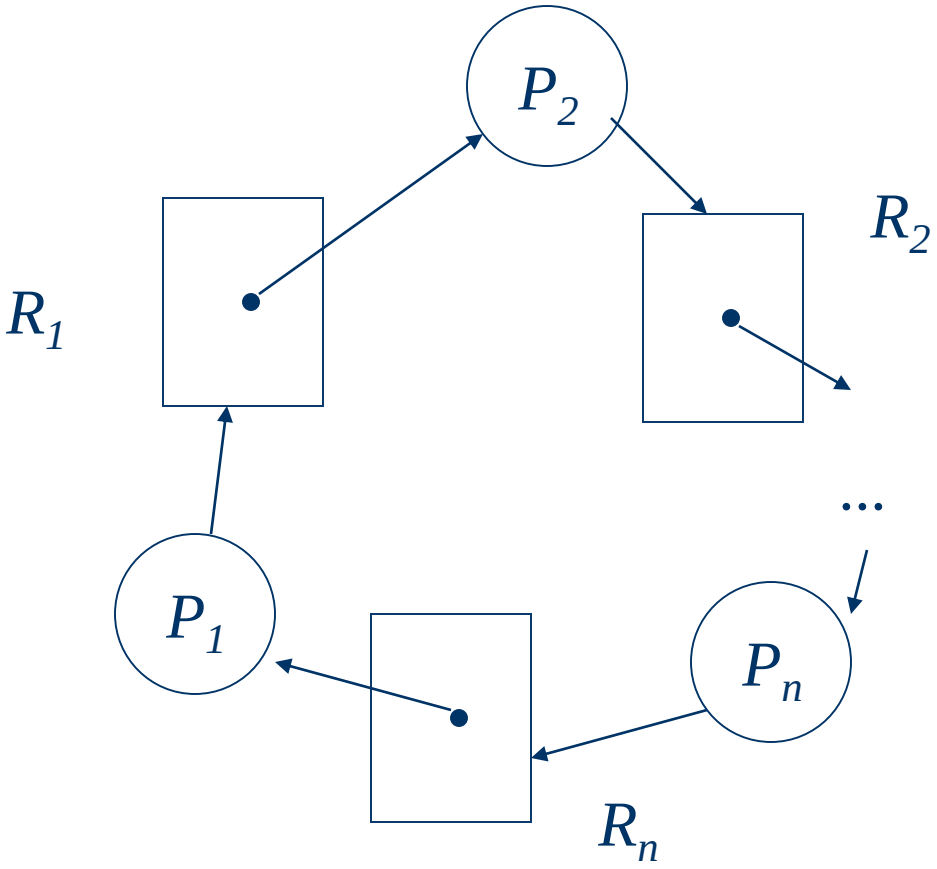


$E = \{(P_3, R_1)\} \cup \{(R_1, P_1), (R_1, P_2)\}$

* After a resource is granted, the request edge is deleted

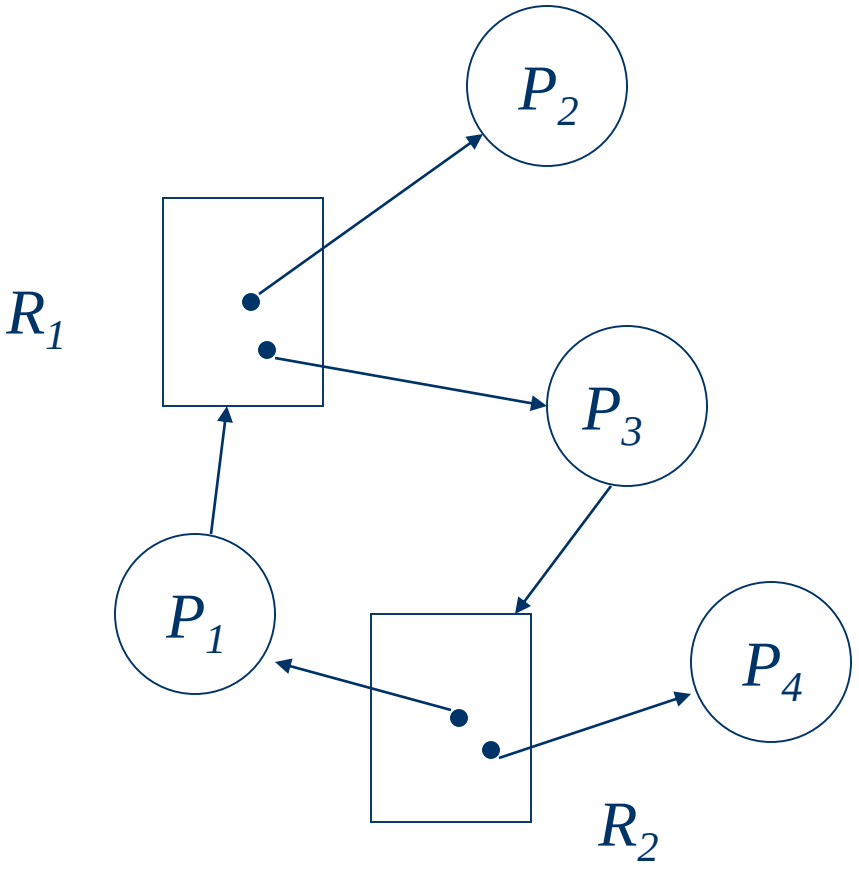
** A *single-instance* resource

Presence of a Cycle in a Single-Instance Resource System...



... is a *necessary* **and** *sufficient* condition for a deadlock

Presence of a Cycle in a multiple-instance resource system



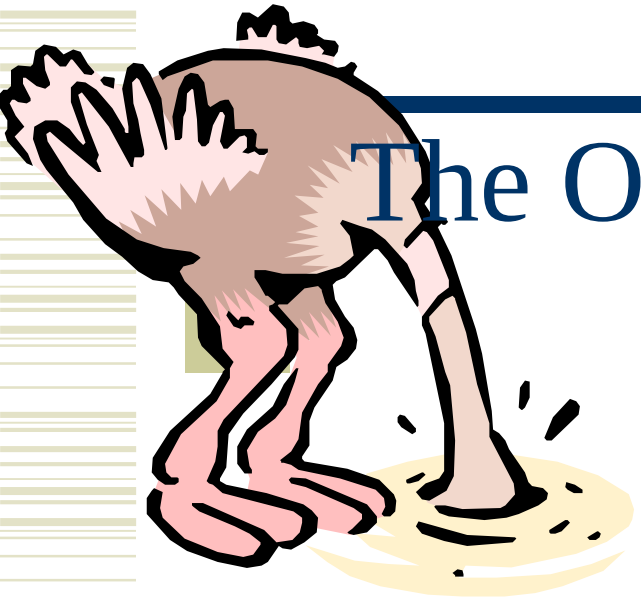
Is **not** *sufficient* (but still, necessary, of course), as shown in this example. Consider what *may* happen when P_4 finishes and releases its resource.

The Four Strategies for Dealing with Deadlocks

1. The *Ostrich Algorithm* (Stick you head in the sand and pretend there is no problem at all)
2. *Prevention* by negating one of the four necessary conditions
3. *Detection* and *recovery*
4. Dynamic *avoidance* by specially designed resource allocation

e Ostrich Algorithm (a case study)

- In Unix, the total number of processes in the system is determined by the (configurable) number of entries in the Process Table. Say it is configured to 100.
- When creating a process (via the *fork* request) fails because the table is full, a typical programming practice is is to wait for a random time interval and try again
- If ten processes are running, each of which needs to create 12 other processes, and has already created nine other processes, they will all sit in endless loops—a deadlock!



The Ostrich Algorithm (a case study cont.)

- Similarly, the maximum number of open files is restricted by the size of the *i-node* (table)—we will study that later, so the same problem may occur when we fill that table up
- Swap space on the disk (again, we will study that later) is another example of a limited resource

But...

***Unix* ignores these problems: they don't occur that often, and the cost of preventing or detecting them is too high for a *general* operating system.**

Deadlock Prevention (A case study)

- A mathematician says: "*An algorithm for preventing such a terrible problem must be implemented at all costs!*"

Deadlock Prevention (Another case study)

- An engineer asks:
  1. How often does the system fail because of a deadlock?
  2. How often does the system fail because of other reasons?
  3. What is the cost of implementing deadlock prevention?

Deadlock Prevention (cont.)

- An engineer determines:
  1. The probability p_d of the system failing because of a deadlock
  2. The probability p_o of the system failing because of another reason
  3. The cost c_d of implementing deadlock prevention

Deadlock Prevention (cont.)

- An engineer makes a recommendation:
  1. If $p_d << p_o$, don't even bother!
  2. Otherwise, check c_d. If it is too high, let PR people deal with it and never again think about fixing this problem.
- The cost is found too high, and no prevention is implemented as the result; the CTO is notified.
- The marketing people write a beautiful variegated brochure explaining that the system *both* prevents and detects deadlocks, and the CTO travels around promoting the story...

Deadlock Prevention (cont.)

- Remember? **A** is *necessary* for **B** if and only if—by definition

$$\mathbf{B \rightarrow A}$$

- If we ensure **A** is false, we then prevent **B**.
- The four such necessary conditions for a deadlock are
  1. Mutual Exclusion
  2. Hold and Wait
  3. No Preemption
  4. Circular Wait

Deadlock Prevention: Eliminating One of the Four Necessary Conditions

1. **Mutual Exclusion:** Each resource is *either* currently assigned to only one process *or* is available

2. **Hold and Wait:** A process holding a resource can request new resources

3. **No Preemption:** A resource previously granted cannot be taken away from a process. (It may be released only by a holding process.)

4. **Circular Wait:** There is a cyclic chain of processes and resources connected, respectively, by the request and assignment edges

Deadlock Prevention
Eliminating (1) *Mutual Exclusion*

- **Mutual Exclusion:** (What would happen, if two processes wrote to the same file simultaneously?) We can try to eliminate problems with certain output devices by spooling everything, but
  1. Not all devices can be spooled
  2. The disk space itself can become the very resource on which the processes are deadlocked

In general, eliminating mutual exclusion is unimplementable.

Deadlock Prevention Eliminating (2) *Hold and Wait*

- **Hold and Wait:** can be eliminated by enforcing a *protocol* so that all needed resources be allocated ahead of time.

That is implementable, but it has

Problems:

1. Low resource utilization as the resources are held idle
2. Often a process does not know all its resources when it starts executing (consider LISP machines, where programs may self-construct!)

Deadlock Prevention
Eliminating (3) *No Preemption*

3.No Preemption: Can be eliminated by either taking away **all** resources from a process that is waiting for a resource that cannot be granted, or—alternatively—preempting the processes that hold the resources needed by that process. (In either case, the preempted processes can be restarted later and granted all the resources they need.)

Even less promising than (2) because of the following

Problem: This works with memory and CPU, but does not quite work with I/O devices like printers. (Why?)

Deadlock Prevention
Eliminating *Circular Wait*

4. **Circular wait:** Can be dealt with by assigning to each resource of type *r* a number $v(r)$ (all numbers are different) and enforcing the following protocol:

- When a process *P* that requests some units of a resource *R*, the request is granted if and only if

$$v(R) > v(r) \quad \forall r \in \{r: r \text{ is held by } P\}$$

Deadlock Prevention
Eliminating *Circular Wait* (cont.)

Why does this protocol prevent a deadlock?

Proof: Suppose, there is a deadlock. Then, there must be a cycle:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow \ldots \rightarrow P_n \rightarrow R_n \rightarrow P_1,$$

but, according, to the protocol, that results in

$$\nu(R_n) < \nu(R_1) < \nu(R_2) < \ldots < \nu(R_n),$$

which is impossible.

Deadlock Prevention
Eliminating *Circular Wait* (cont.)

- The problem with that protocol is that, in practice, it is very hard to order resources so that the ordering makes sense.
- Another approach to eliminating circular wait is to request that the process may have only one resource at any time (i.e., release one resource before asking for another one). But then how would one copy a disk image to a tape?..

The Four Strategies for Dealing with Deadlocks

1. The *Ostrich Algorithm* (Stick you head in the sand and pret[illegible]blem at all)
2. *Prevention* by [illegible] the four necessary conditions
3. *Detection* and *recovery*
4. Dynamic *avoidance* by specially designed resource allocation

Done