Chapter 3: Deadlocks

Overview

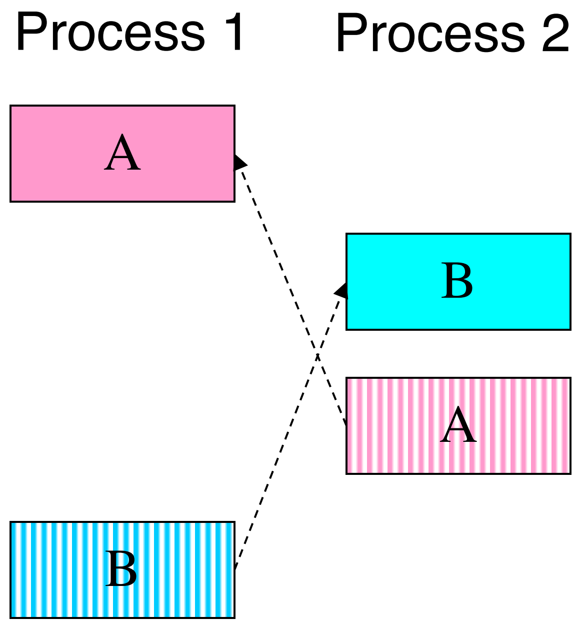
- Resources
- Why do deadlocks occur?
- Dealing with deadlocks
  - Ignoring them: ostrich algorithm
  - Detecting & recovering from deadlock
  - Avoiding deadlock
  - Preventing deadlock

Resources

- Resource: something a process uses
  - Usually limited (at least somewhat)
- Examples of computer resources
  - Printers
  - Semaphores / locks
  - Tables (in a database)
- Processes need access to resources in reasonable order
- Two types of resources:
  - Preemptable resources: can be taken away from a process with no ill effects
  - Nonpreemptable resources: will cause the process to fail if taken away

When do deadlocks happen?

- Suppose
  - Process 1 holds resource A and requests resource B
  - Process 2 holds B and requests A
  - Both can be blocked, with neither able to proceed
- Deadlocks occur when …
  - Processes are granted exclusive access to devices or software constructs (resources)
  - Each deadlocked process needs a resource held by another deadlocked process



DEADLOCK!

Using resources

- Sequence of events required to use a resource
  - Request the resource
  - Use the resource
  - Release the resource
- Can’t use the resource if request is denied
  - Requesting process has options
    - Block and wait for resource
    - Continue (if possible) without it: may be able to use an alternate resource
    - Process fails with error code
  - Some of these may be able to prevent deadlock…

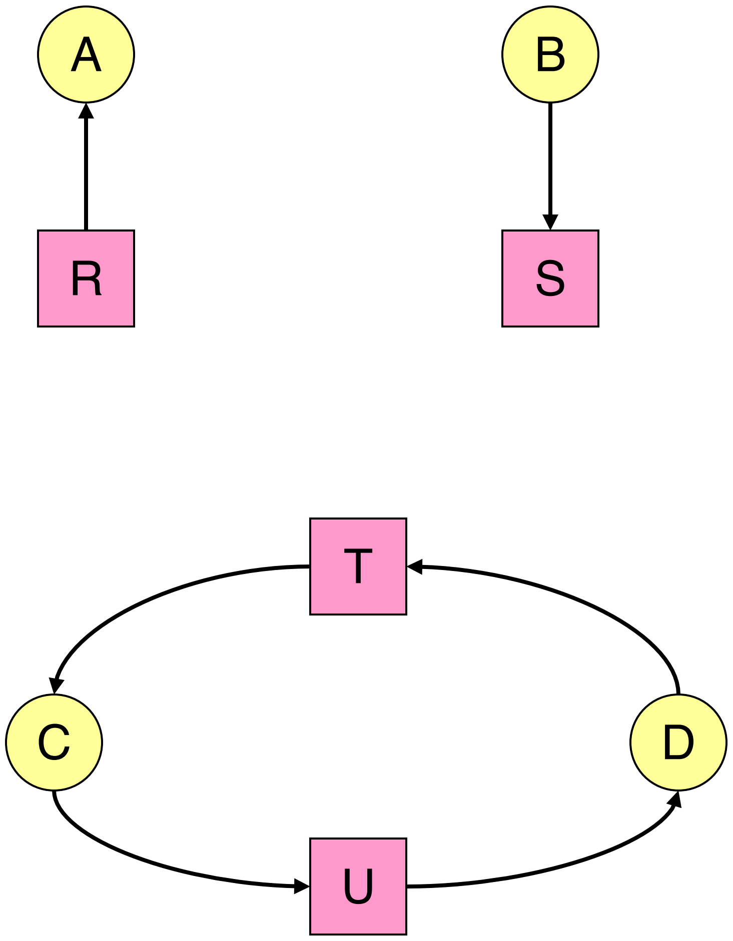
What is a deadlock?

- Formal definition:
  "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause."
- Usually, the event is release of a currently held resource
- In deadlock, none of the processes can
  - Run
  - Release resources
  - Be awakened

Four conditions for deadlock

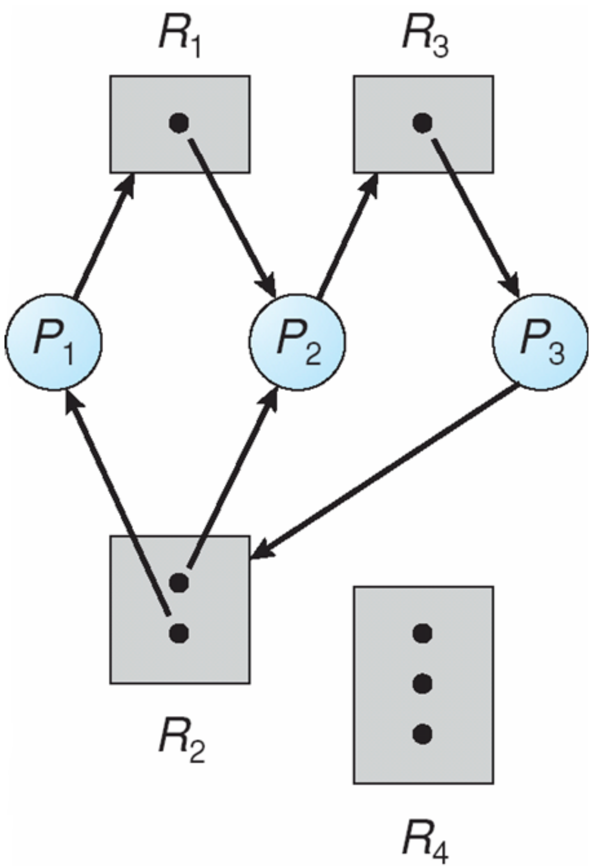
- Mutual exclusion
  - Each resource is assigned to at most one process
- Hold and wait
  - A process holding resources can request more resources
- No preemption
  - Previously granted resources cannot be forcibly taken away
- Circular wait
  - There must be a circular chain of 2 or more processes where each is waiting for a resource held by the next member of the chain

Resource allocation graphs

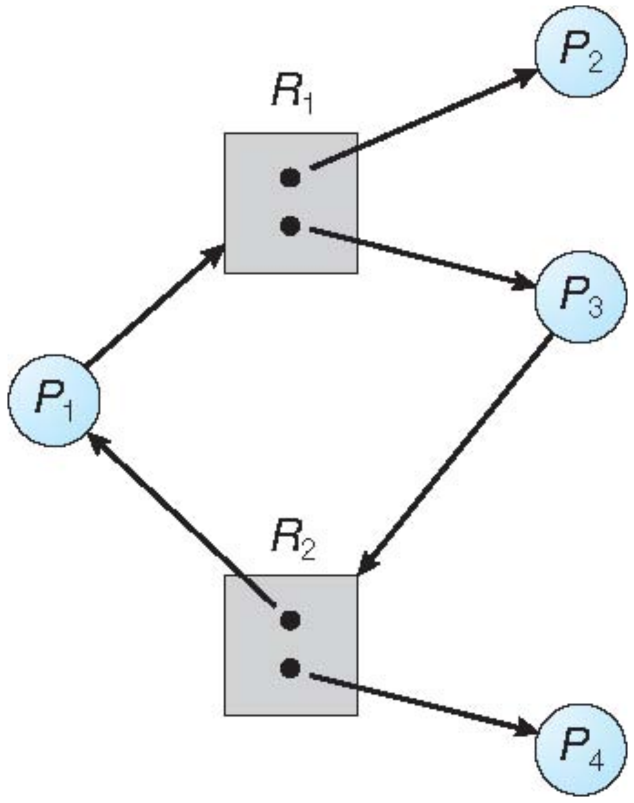


- Resource allocation modeled by directed graphs
- Example 1:
  - Resource R assigned to process A
- Example 2:
  - Process B is requesting / waiting for resource S
- Example 3:
  - Process C holds T, waiting for U
  - Process D holds U, waiting for T
  - C and D are in deadlock!

Resource Allocation Graph: Multiple Resources



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
    - necessary and sufficient condition
  - if several instances per resource type, possibility of deadlock
    - necessary condition

Dealing with deadlock

- How can the OS deal with deadlock?
  - Ignore the problem altogether!
    - Hopefully, it'll never happen…
  - Detect deadlock & recover from it
  - Dynamically avoid deadlock
    - Careful resource allocation
  - Prevent deadlock
    - Remove at least one of the four necessary conditions
- We'll explore these tradeoffs

Getting into deadlock

A

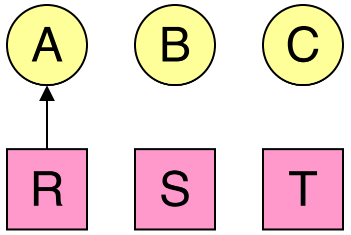
```
Acquire R
Acquire S
Release R
Release S
```

B

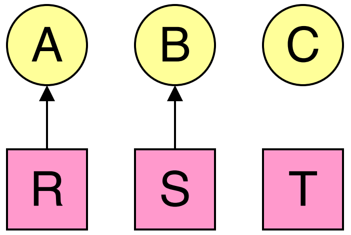
```
Acquire S
Acquire T
Release S
Release T
```

C

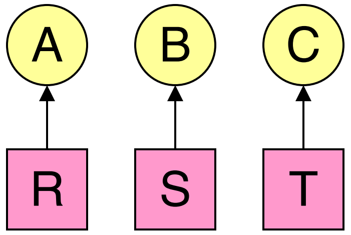
```
Acquire T
Acquire R
Release T
Release R
```



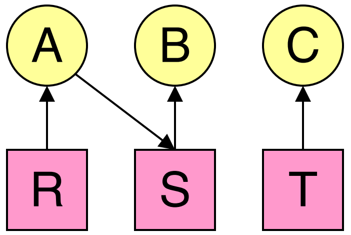
```
Acquire R
```



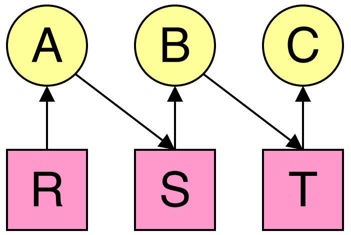
```
Acquire S
```



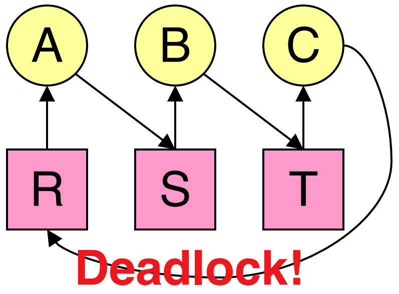
```
Acquire T
```



```
Acquire S
```



```
Acquire T
```



```
Acquire R
```

Not getting into deadlock…

- Many situations *may* result in deadlock (but don't have to)
  - In previous example, A could release R before C requests R, resulting in no deadlock
  - Can we always get out of it this way?
- Find ways to:
  - Detect deadlock and reverse it
  - Stop it from happening in the first place

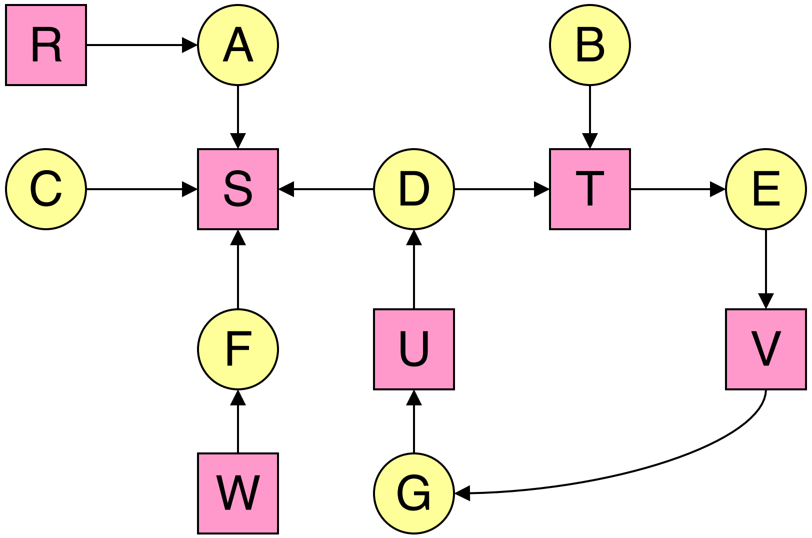
The Ostrich Algorithm

- Pretend there’s no problem
- Reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting
- Trade off between convenience and correctness

Detecting deadlocks using graphs

- Process holdings and requests in the table and in the graph (they're equivalent)
- Graph contains a cycle => deadlock!
  - Easy to pick out by looking at it (in this case)
  - Need to mechanically detect deadlock
- Not all processes are deadlocked (A, C, F not in deadlock)

Process	Holds	Wants
A	R	S
B		T
C		S
D	U	S,T
E	T	V
F	W	S
G	V	U



Deadlock detection algorithm

- General idea: try to find cycles in the resource allocation graph
- Algorithm: depth-first search at each node
  - Mark arcs as they're traversed
  - Build list of visited nodes
  - If node to be added is already on the list, a cycle exists!
- Cycle == deadlock

```
For each node N in the graph {
  Set L = empty list
  unmark all arcs
  Traverse (N,L)
}
If no deadlock reported by now,
there isn't any

define Traverse (C,L)  {
  If C in L, report deadlock!
  Add C to L
  For each unmarked arc from C {
    Mark the arc
    Set A = arc destination
    /* NOTE: L is a
       local variable */
    Traverse (A,L)
  }
}
```

Resources with multiple instances

- Previous algorithm only works if there’s one instance of each resource
- If there are multiple instances of each resource, we need a different method
  - Track current usage and requests for each process
  - To detect deadlock, try to find a scenario where all processes can finish
  - If no such scenario exists, we have deadlock

Deadlock detection algorithm

	A	B	C	D
Avail	2	3	0	1

Hold

Process	A	B	C	D
1	0	3	0	0
2	1	0	1	1
3	0	2	1	0
4	2	2	3	0

Want

Process	A	B	C	D
1	3	2	1	0
2	2	2	0	0
3	3	5	3	1
4	0	4	1	1

```
current=avail;
for (j = 0; j < N; j++) {
 for (k=0; k<N; k++) {
  if (finished[k])
   continue;
  if (want[k] < current) {
   finished[k] = 1;
   current += hold[k];
   break;
  }
  if (k==N) {
   printf “Deadlock!\n”;
   // finished[k]==0 means process is in
   // the deadlock
   break;
  }
}
```

Note: want[j],hold[j],current,avail are arrays!

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

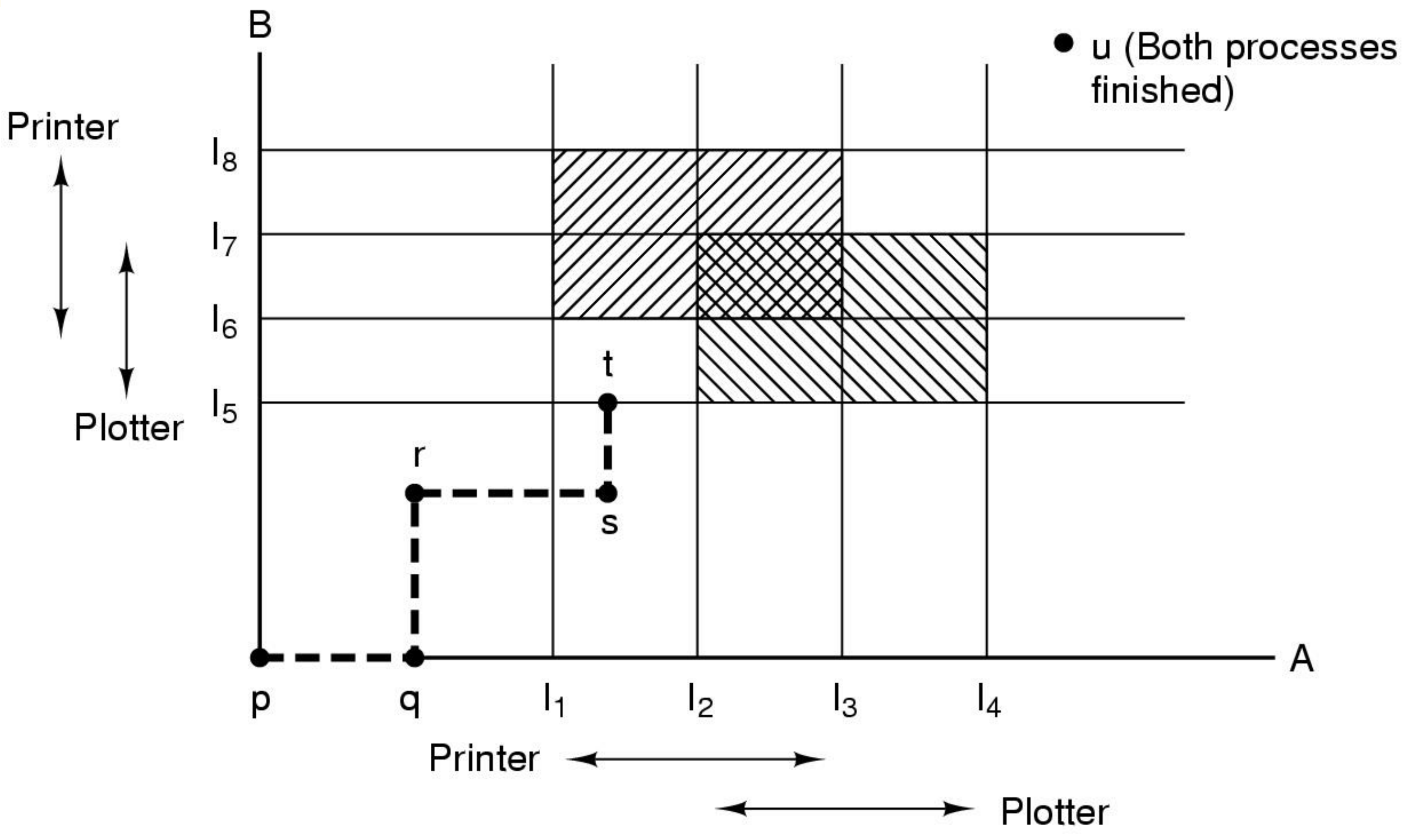
Recovering from deadlock: options

- Recovery through resource preemption
  - Take a resource from some other process
  - Depends on nature of the resource and the process
- Recovery through rollback
  - Checkpoint a process periodically
  - Use this saved state to restart the process if it is found deadlocked
  - May present a problem if the process affects lots of “external” things
- Recovery through killing processes
  - Crudest but simplest way to break a deadlock: kill one of the processes in the deadlock cycle
  - Other processes can get its resources
  - Preferably, choose a process that can be rerun from the beginning
    - Pick one that hasn’t run too far already

Deadlock Recovery: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Resource trajectories



Two process resource trajectories

Safe and unsafe states

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Max
A	3	9
B	0	-
C	2	7

Free: 5

	Has	Max
A	3	9
B	0	-
C	7	7

Free: 0

	Has	Max
A	3	9
B	0	-
C	0	-

Free: 7

Demonstration that the first state is safe

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

	Has	Max
A	4	9
B	0	-
C	2	7

Free: 4

Demonstration that the second state is unsafe

Banker's Algorithm for a single resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

Any sequence finishes

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

C,B,A,D finishes

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Deadlock (unsafe state)

- Bankers' algorithm: before granting a request, ensure that a sequence exists that will allow all processes to complete
  - Use previous methods to find such a sequence
  - If a sequence exists, allow the requests
  - If there's no such sequence, deny the request
- Can be slow: must be done on each request!

Banker's Algorithm for multiple resources

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Example of banker's algorithm with multiple resources

Preventing deadlock

- Deadlock can be completely prevented!
- Ensure that at least one of the conditions for deadlock never occurs
  - Mutual exclusion
  - Circular wait
  - Hold & wait
  - No preemption
- Not always possible…

Eliminating mutual exclusion

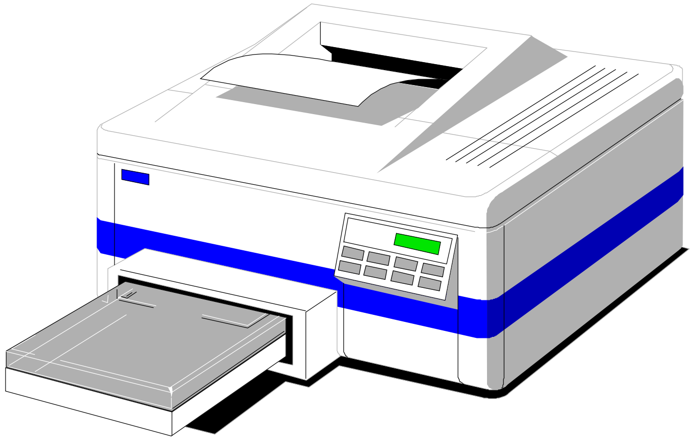
- Some devices (such as printer) can be spooled
  - Only the printer daemon uses printer resource
  - This eliminates deadlock for printer
- Not all devices can be spooled
- Principle:
  - Avoid assigning resource when not absolutely necessary
  - As few processes as possible actually claim the resource

Attacking “hold and wait”

- Require processes to request resources before starting
  - A process never has to wait for what it needs
- This can present problems
  - A process may not know required resources at start of run
  - This also ties up resources other processes could be using
    - Processes will tend to be conservative and request resources they might need
- Variation: a process must give up all resources before making a new request
  - Process is then granted all prior resources as well as the new ones
  - Problem: what if someone grabs the resources in the meantime—how can the process save its state?

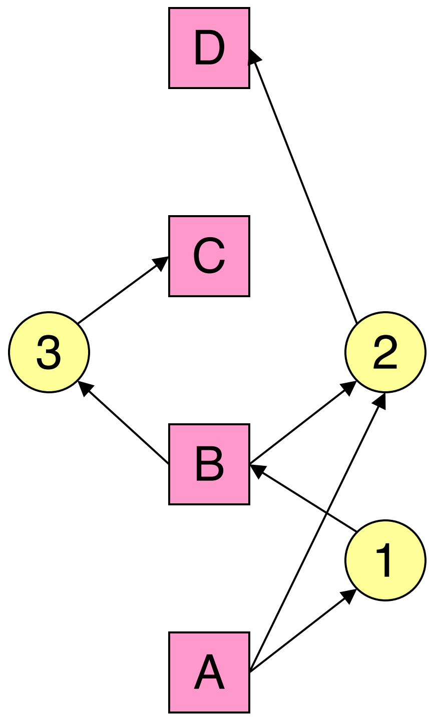
Attacking “no preemption”

- This is not usually a viable option
- Consider a process given the printer
  - Halfway through its job, take away the printer
  - Confusion ensues!
- May work for some resources
  - Forcibly take away memory pages, suspending the process
  - Process may be able to resume with no ill effects



Attacking "circular wait"

- Assign an order to resources
- Always acquire resources in numerical order
  - Need not acquire them all at once!
- Circular wait is prevented
  - A process holding resource *n* can't wait for resource *m* if $m < n$
  - No way to complete a cycle
    - Place processes above the highest resource they hold and below any they're requesting
    - All arrows point up!



Deadlock prevention: summary

- Mutual exclusion
  - Spool everything
- Hold and wait
  - Request all resources initially
- No preemption
  - Take resources away
- Circular wait
  - Order resources numerically

Example: two-phase locking

- Phase One
  - Process tries to lock all data it needs, one at a time
  - If needed data found locked, start over
  - (no real work done in phase one)
- Phase Two
  - Perform updates
  - Release locks
- Note similarity to requesting all resources at once
- This is often used in databases
- It avoids deadlock by eliminating the "hold-and-wait" deadlock condition

“Non-resource” deadlocks

- Possible for two processes to deadlock
  - Each is waiting for the other to do some task
- Can happen with semaphores
  - Each process required to do a down() on two semaphores (mutex and another)
  - If done in wrong order, deadlock results
- Semaphores could be thought of as resources…

Starvation

- Algorithm to allocate a resource (akin to scheduling)
  - Give the resource to the shortest job first
  - Works great for multiple short jobs in a system
  - May cause long jobs to be postponed indefinitely
- Solution
  - First-come, first-serve policy
- Starvation can lead to deadlock
  - Process starved for resources can be holding (other) resources
  - If those resources aren't used and released in a timely fashion, shortage could lead to deadlock
- Is this in general or for deadlocks?