CSE 3320
Operating Systems
Deadlock

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of the Last Class

- Race conditions
- Mutual exclusion and critical regions
- Two simple approaches
  - Disabling interrupt and Lock variables
- Busy waiting
  - Strict alternation, Peterson’s and TSL
- Semaphores
- Mutexes
- Monitors
- Message Passing
- Barrier

Deadlock Definitions

- Two or more processes each blocked and waiting for resources they will never get without drastic actions
  - Something preempts a resource
  - A process is killed
- 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, thus, no process can
  - run
  - release resources
  - be awakened

Resources and Deadlocks (1)

- Examples of computer resources
  - printers
  - tape drives
  - tables
  - software
- Processes need access to resources in a reasonable order
- Suppose a process holds resource A and requests resource B

  Both processes want to have *exclusive access* to A and B!

  - at the same time another process holds B and requests A
  - both are blocked and remain so, *deadlocks*

Resources and Deadlocks (2)

- Deadlocks occur when …
  - processes are granted *exclusive access* to hardware, e.g., I/O devices
  - processes are granted *exclusive access* to software, e.g., database records
  - we refer to these generally as <u>resources</u>
- Pre-emptible resources
  - can be taken away from a process with no ill effects, e.g., Mem
- Non-preemptible resources
  - will cause the process to fail if taken away, e.g., CD burner

In general, deadlocks involve *non-preemptible* and *exclusive* resources!

Resources and Deadlocks (3)

- Sequence of events required to use a resource
  - request the resource
  - use the resource
  - release the resource
- Must wait if request is denied
  - requesting process may be blocked
  - may fail with error code

Resource Acquisition

- Can using semaphores avoid deadlocks?

```
typedef int semaphore;
semaphore resource_1;


void process_A (void) {
        down(&resource_1);
        use_resource_1 ();
        up(&resource_1);
 }
```

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A (void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources();
        up(&resource_2);
        up(&resource_1);
}
```

Using semaphore to protect resources. (a) One resource. (b) Two resources.

But using semaphores wisely !

Resource Acquisition (2)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A (void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources();
        up(&resource_2);
        up(&resource_1);
}

void process_B (void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources();
        up(&resource_2);
        up(&resource_1);
}
```

(a) Deadlock-free code.

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A (void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources();
        up(&resource_2);
        up(&resource_1);
}

void process_B (void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources();
        up(&resource_1);
        up(&resource_2);
}
```

(b) Code with a potential deadlock, why?

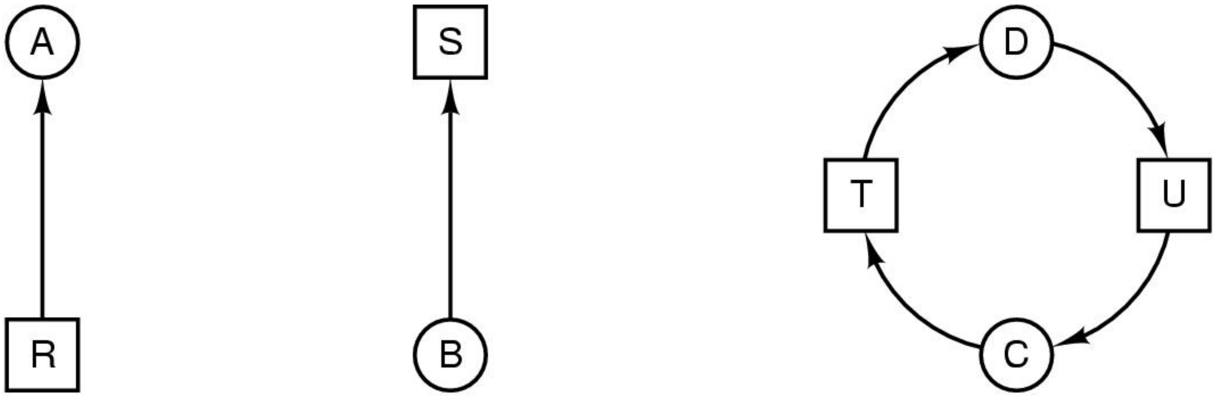
Four Conditions for Deadlock

Coffman (1971)

1. Mutual exclusion condition
   - each resource assigned to 1 process or is available
2. Hold and wait condition
   - process holding resources can request additional
3. No preemption condition
   - previously granted resources cannot be forcibly taken away
4. Circular wait condition
   - must be a circular chain of 2 or more processes
   - each is waiting for resources held by next member of the chain

Deadlock Modeling (1)

- Modeled with directed graphs
  - A cycle means a deadlock involving the processes and resources



(a) (b) (c)

  - resource R assigned to process A
  - process B is requesting/waiting for resource S
  - process C and D are in deadlock over resources T and U

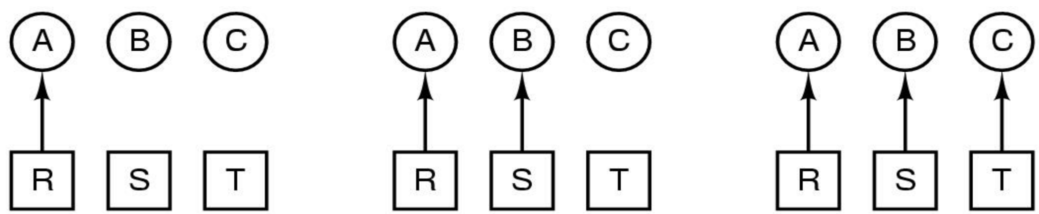
Deadlock Modeling (2)

A	B	C
Request R Request S Release R Release S	Request S Request T Release S Release T	Request T Request R Release T Release R
(a)	(b)	(c)

(a) – (c)
Sequential model
no deadlock,
no parallelism

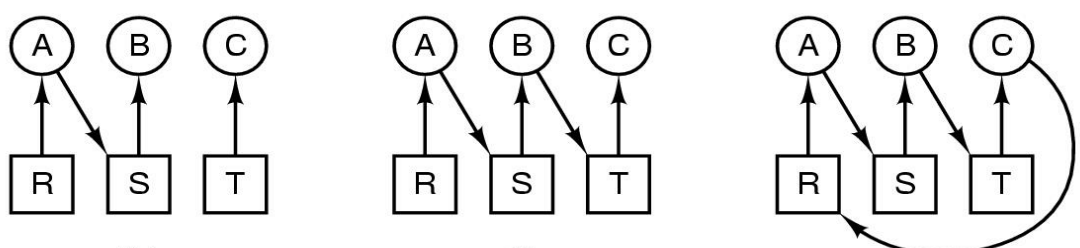
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)



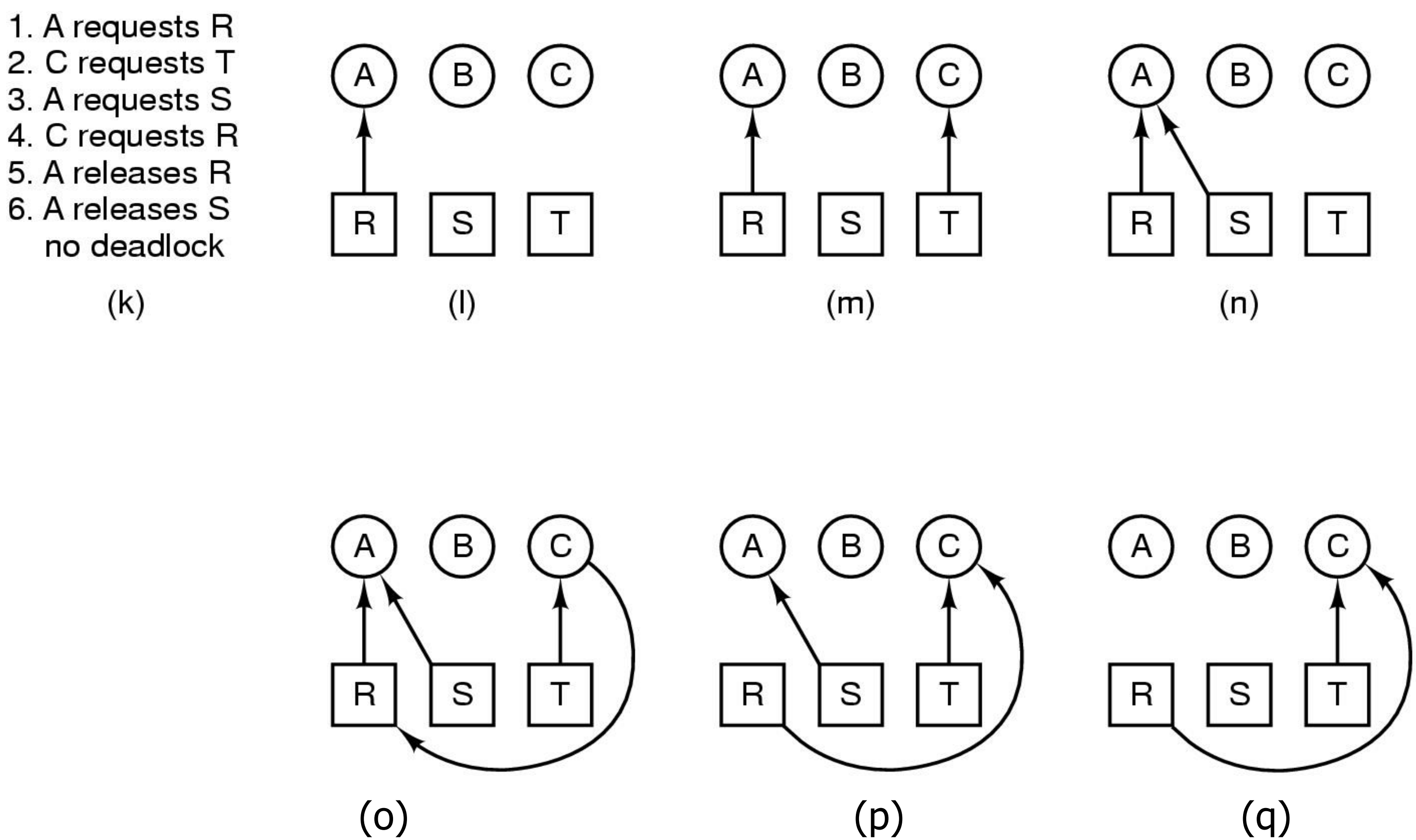
(e) (f) (g)

What if the OS knew the impending deadlock of granting B resource S at step (f)?



(h) (i) (j)

Deadlock Modeling (3)



How deadlock can be avoided by OS’ re-ordering

Dealing with Deadlocks

- Strategies for dealing with Deadlocks
  1. just ignore the problem altogether
  2. detection and recovery
  3. dynamic avoidance
     - careful resource allocation
  4. prevention
     - negating one of the four necessary conditions

The Ostrich Algorithm

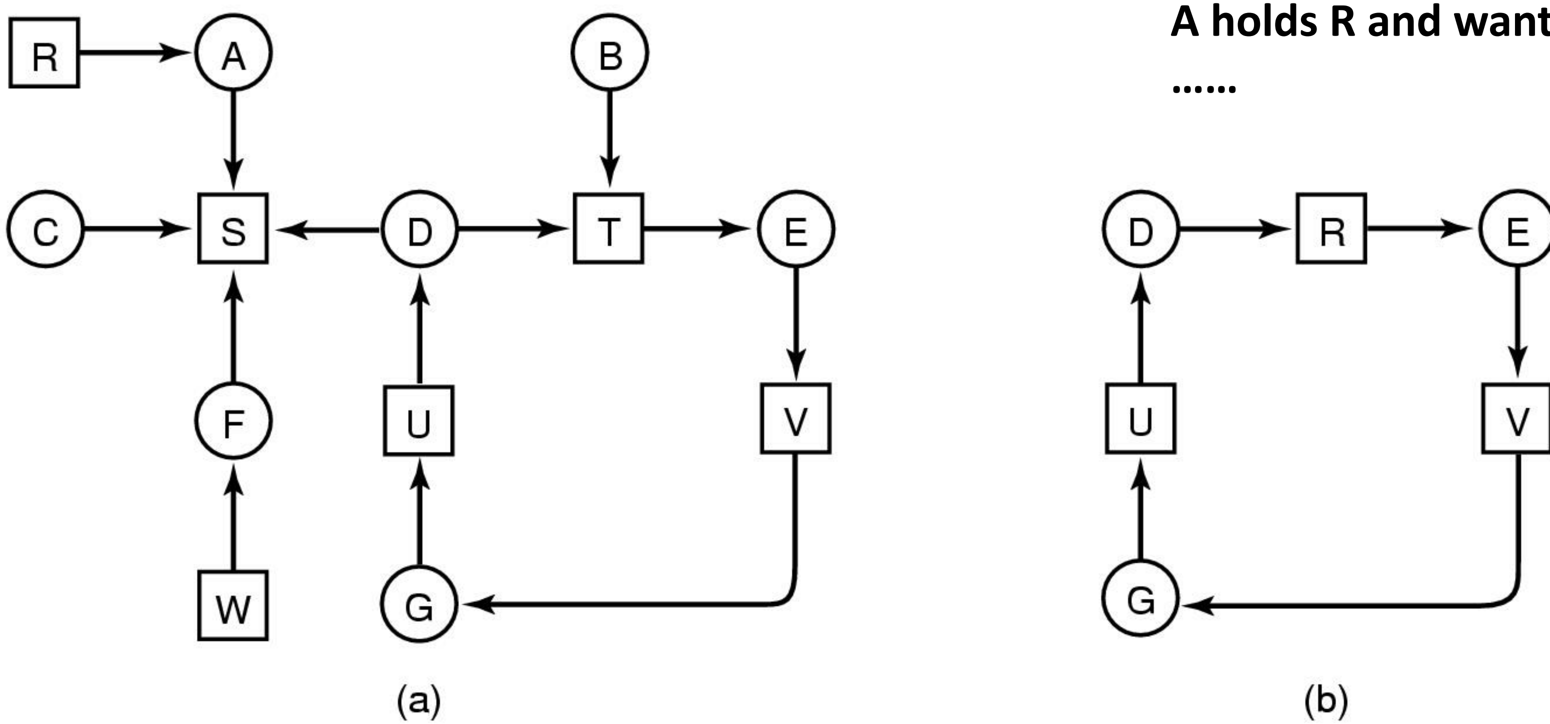
- Pretend there is no problem
- Reasonable if
  - deadlocks occur very rarely
  - cost of prevention is high
- UNIX and Windows take this approach
- It is a trade off between
  - convenience
  - correctness

Detection with One Resource Type

- Assumption: only one resource of each type exists

A holds R and wants S

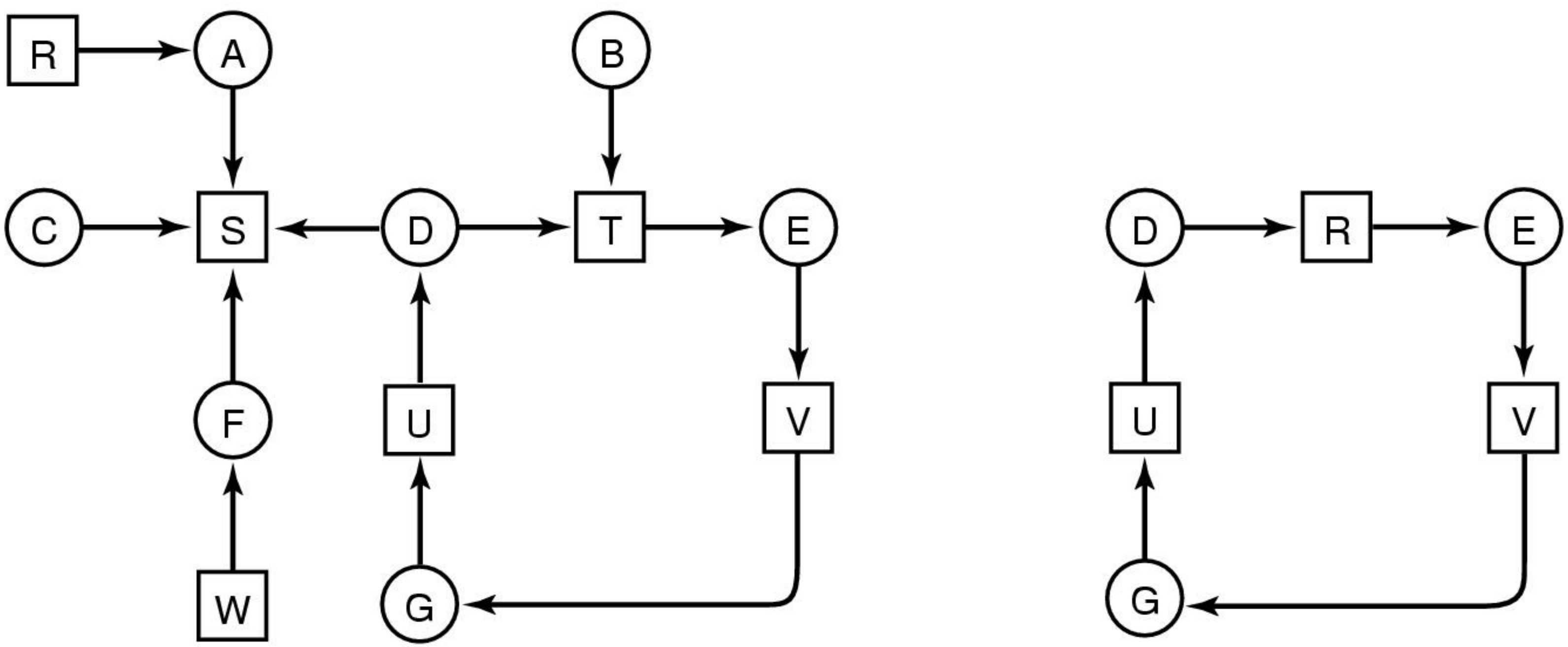
……



- Note the resource ownership and requests
- A cycle can be found within the graph, denoting deadlock

Detect a Cycle in a Graph

- A data structure to find if a graph is a tree that is cycle-free
  - depth-first searching (P.445)
  - Left-right, top-to-bottom: R, A, B, C, S, D, T, E, F



(a) (b)

Detection with Multiple Resources of Each Type (1)

- Deadlock detection algorithm:
  - Two vectors and two matrixes
  - Vector comparison; $A \le B$ means $A_i \le B_i$ for $1 \le i \le m$
  - Observation: Sum_Cij + Aj = E j

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Data structures needed by deadlock detection algorithm

Detection with Multiple Resources of Each Type (2)

- Key: a completed process can release its resources so as to give other processes chances to acquire resources and run
  - Look for a process Pi, If R[i] ≤ A? if so, A = R[i] + C[i]

  Although the algorithm is nondeterministic, the result is always the same

  The scheduling order do not matter

$$
\begin{array}{lcccc}
 & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\
E = (& 4 & 2 & 3 & 1\ )
\end{array}
\qquad
\begin{array}{lcccc}
 & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\
A = (& 2 & 1 & 0 & 0\ )
\end{array}
$$

What if process 2 needs a CD-ROM drive and 2 tape drivers and the plotter?

When to run the deadlock detection algorithm? Why CPU utilization?

Current allocation matrix

$$
C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}
$$

Request matrix

$$
R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}
\begin{array}{l} \text{P1} \\ \text{P2} \\ \text{p3} \end{array}
$$

An example for the deadlock detection algorithm

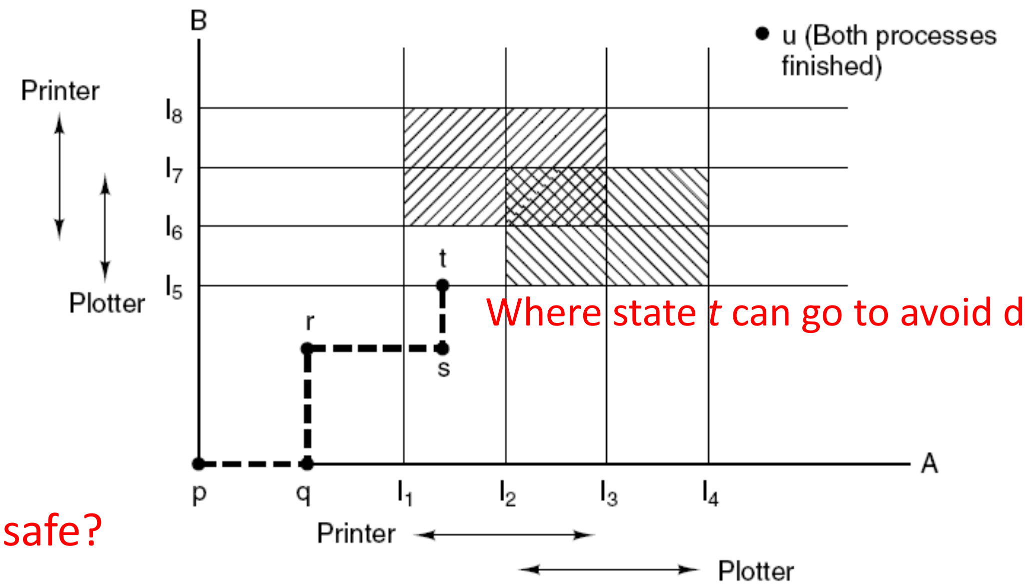
Recovery from Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on the nature of the resource
- Recovery through rollback
  - *checkpoint* a process periodically, resulting a sequence of checkpoint files
  - use this saved state
  - restart the process if it is found deadlocked
  - Processes in database and network applications are not easy to rollback, why?
- Recovery through killing processes
  - crudest but simplest way to break a deadlock
  - kill one of the processes in the deadlock cycle
  - the other processes get its resources
  - choose process that can be rerun from the beginning, not easy!

Will killing a process not in the deadlock cycle help?

Deadlock Avoidance

- Allocate resources wisely to avoid deadlocks
  - But certain information should be available in advance
  - Base: concept of safe states



Where state *t* can go to avoid deadlock?

Which area is unsafe?

What if t is at the intersection of I1 and I5 ?

Two process resource trajectories.

Safe States (w/ one resource type)

- Safe state
  - if it is not deadlocked, and, there is *some* scheduling order in which every process can run to completion even if all of them request their maximum number of resources immediately

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

Why state (a) is safe?

Unsafe States (w/ one resource type)

- Unsafe state
  - there is *no guarantee* of having some scheduling order in which every process can run to completion even if all of them request their maximum number of resources immediately
  - Not the same as a deadlocked state, why? What is the difference?

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

Why state (b) is <u>NOT</u> safe?

The Banker's Algorithm for a Single Resource

- The algorithm models on the way of a banker might deal with a group of customers to whom he has granted lines of credit
  - Not all customers need their maximum credit line simultaneously
  - To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer

(a)

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

Free: 10

(b)

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

Free: 2

(c)

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

Free: 1

Three resource allocation states: (a) safe; (b) safe; (c) unsafe

The Banker's Algorithm for Multiple Resources (1)

- The algorithm looks for a process Pi, If R[i] ≤ A? if so, A = R[i] + C[i]
  - How R is achieved? R = M (Maximum) - C
  - What is the underlying assumption? M info available in advance

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)

E = P + A

If process B requests a scanner, can it be granted? Why?

The Banker's Algorithm for Multiple Resources (2)

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)

After process B was granted a scanner, now process E wants the last scanner, can it be granted? Why?

Why in practice the algorithm is essentially useless?

Deadlock Prevention (1)

Attack the mutual exclusion condition of Coffman Rules

- Some devices (such as printer) can be spooled
  - only the printer daemon uses printer resource
  - thus deadlock for printer eliminated
  - But the disk could be deadlocked, though more unlikely
- Not all devices can be spooled, e.g., process table
- Principle:
  - avoid assigning resource when not absolutely necessary
  - as few processes as possible actually claim the resource

Deadlock Prevention (2)

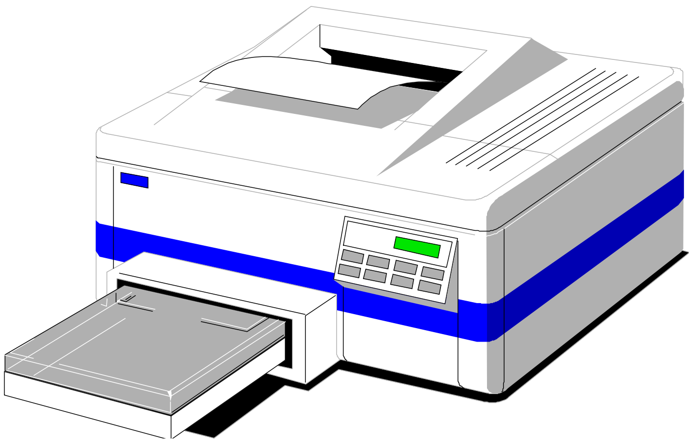
Attack the Hold-and-Wait condition of Coffman Rules

- Require processes to request *all* resources before starting
  - a process never has to wait for what it needs
- Problems
  - may not know required resources at start of run
  - also ties up resources other processes could be using
    - Less concurrency!
- Variation:
  - process must temporarily give up all resources
  - then request all immediately needed

Deadlock Prevention (3)

Attack the No-Preemption Condition of Coffman Rules

- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??



Deadlock Prevention (4)

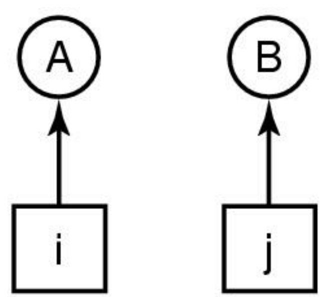
Attack the Circular Wait Condition of Coffman Rules

- A process is entitled only to a single resource at any moment
- Provide a global numbering of all the resources
  - A process can request resources whenever they want to, but all requests must be made in numerical order (or no process requests a resource lower than what it is already holding)

- Why no deadlock?
- Is it feasible in implementation? – what is a good ordering?

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a) Normally ordered resources



(b) a resource graph

Deadlock Prevention Summary

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

Summary of approaches to deadlock prevention

What is the difference between deadlock avoidance and deadlock prevention?

dynamic scheduling vs. static ruling

Two-Phase Locking

- Phase One
  - process tries to lock all records it needs, one at a time
  - if needed record found locked, start over
  - (no real work done in phase one)
- If phase one succeeds, it starts second phase,
  - performing updates
  - releasing locks
- Note similarity to requesting all resources at once
  - Attacking the hold-and-wait condition
- Algorithm works where programmer can arrange
  - program can be stopped and restarted in the first phase, instead of blocking!

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

Re: The Producer-consumer Problem w/ Semaphores

```
#define N 100                         /* number of slots in the buffer */
typedef int semaphore;                /* semaphores are a special kind of int */
semaphore mutex = 1;                  /* controls access to critical region */
semaphore empty = N;                  /* counts empty buffer slots */
semaphore full = 0;                   /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                   /* TRUE is the constant 1 */
          item = produce_item( );     /* generate something to put in buffer */
          down(&empty);               /* decrement empty count */
          down(&mutex);               /* enter critical region */
          insert_item(item);          /* put new item in buffer */
          up(&mutex);                 /* leave critical region */
          up(&full);                  /* increment count of full slots */
     }
}
```

what if the two *downs* in the producer's code were reversed in order, so *mutex* was decremented before empty instead of after it?

```
void consumer(void)
{
     int item;

     while (TRUE) {                   /* infinite loop */
          down(&full);                /* decrement full count */
          down(&mutex);               /* enter critical region */
          item = remove_item( );      /* take item from buffer */
          up(&mutex);                 /* leave critical region */
          up(&empty);                 /* increment count of empty slots */
          consume_item(item);         /* do something with the item */
     }
}
```

Starvation

- Algorithm to allocate a resource
  - may be to give to shortest job first
  - works great for multiple short jobs in a system
- It may cause long jobs to be postponed indefinitely
  - even though not blocked
  - Strict priority may give trouble!
- Solution:
  - First-come, first-serve resource allocation policy

What is the key difference between deadlock and starvation?

Summary

- Deadlocks and its modeling
- Deadlock detection
- Deadlock recovery
- Deadlock avoidance
  - Resource trajectories
  - Safe and unsafe states
  - The banker's algorithm
- Two-phase locking