Operating Systems (A)
(Honor Track)

Lecture 17: Deadlocks

Yao Guo (郭耀)

Peking University

Fall 2021

Acknowledgements: Prof. Xiangqun Chen & Tao Wang at PKU and Prof. Yuanyuan Zhou at UCSD

This Lecture

Deadlocks

Deadlock overview

Dealing with deadlocks

Buzz Words

- Resource allocation graph (RAG)
- Wait-for graph (WFG)
- Ostrich algorithm
- Deadlock prevention
- Deadlock avoidance
- Banker's algorithm
- Deadlock detection
- Deadlock recovery

This Lecture

Deadlocks

Deadlock overview

Dealing with deadlocks

Traffic Deadlock (Gridlock)

Deadlock

- Synchronization is a live gun – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already
- More generally, processes that allocate multiple resources, generate dependencies on those resources
  - Locks, semaphores, monitors, etc., just represent the resources that they protect
- If one process tries to allocate a resource that a second process holds, and vice-versa, they can never make progress
- We call this situation deadlock, and we'll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions

Let's Start with Resource

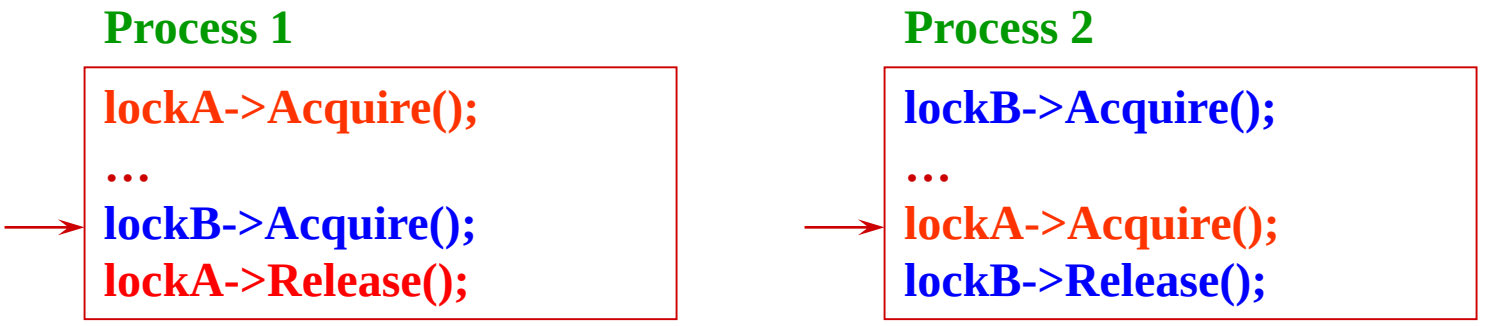
- A **resource** is a commodity needed by a process
- Resources can be either:
  - **Serially** reusable**:** e.g., CPU, memory, disk space, I/O devices, files
    acquire → use → release
  - Consumable**:** produced by a process, needed by a process; e.g., messages, buffers of information, interrupts
    create → acquire → use (consumed)
    Resource ceases to exist after it has been used, so it is not released

Resource (2)

- Resources can also be either:
  - preemptible: e.g., CPU, or
  - non-preemptible: e.g., tape drives
- Resources can be either:
  - shared among several processes or
  - dedicated exclusively to a single process
- Resources can also be either:
  - hardware: e.g., printer, or
  - software/data: e.g., a database entry

Deadlock Definition

- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized
- 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.



Using Semaphore to Share Resource

```
                Process P();                    1   Process Q();
2   { A.Down();                                 6   { A.Down();
3     B.Down();                                       B.Down();
        use both resource                               use both resource
4     B.Up();                                         B.Up();
5     A.Up(); }                                       A.Up(); }
```

1 External Semaphore A(1), B(1);

2 External Semaphore A(0), B(1);

3 External Semaphore A(0), B(0);

4 External Semaphore A(0), B(1);

5 External Semaphore A(1), B(1);

But Deadlock Can Happen!

```
1 ⟶ Process P();
2 ⟶ { A.Down();
      B.Down();
        use both resources
      B.Up();
      A.Up(); }
```

```
1 ⟶ Process Q();
3 ⟶ { B.Down();
      A.Down();
        use both resources
      A.Up();
      B.Up(); }
```

```
1  External Semaphore A(1), B(1);
2  External Semaphore A(0), B(1);
3  External Semaphore A(0), B(0);
```

DEADLOCK IS COMING!

Deadlock vs. Starvation

- Is deadlock the same as starvation (or indefinitely postponed)?
  - A process is indefinitely postponed if it is delayed repeatedly over a *long* period of time while the attention of the system is given to other processes
  - i.e., logically the process may proceed but the system never gives it the CPU in reality

Conditions for Deadlock

- What conditions should exist in order to lead to a deadlock?

Necessary and Sufficient Conditions for Deadlock

- Mutual exclusion
  - Processes claim **exclusive** control of the resources they require
- Hold-and-wait condition
  - Processes hold resources already allocated to them while waiting for additional resources
- No preemption condition
  - Resources cannot be removed from the processes holding them until used to completion
- Circular wait condition
  - A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain

Resource Allocation Graph

- Deadlock can be described using a resource allocation graph (RAG)
- The RAG consists of a set of vertices $P=\{P_1, P_2, \ldots, P_n\}$ of processes and $R=\{R_1, R_2, \ldots, R_m\}$ of resources
  - A directed edge from a process to a resource, $P_i \rightarrow R_j$, means that P_i has requested R_j
  - A directed edge from a resource to a process, $R_j \rightarrow P_i$, means that R_j has been allocated by P_i
  - Each resource has a fixed number of units
- If the graph has no cycles, deadlock cannot exist
- If the graph has a cycle, deadlock may exist

Resource Allocation Graph

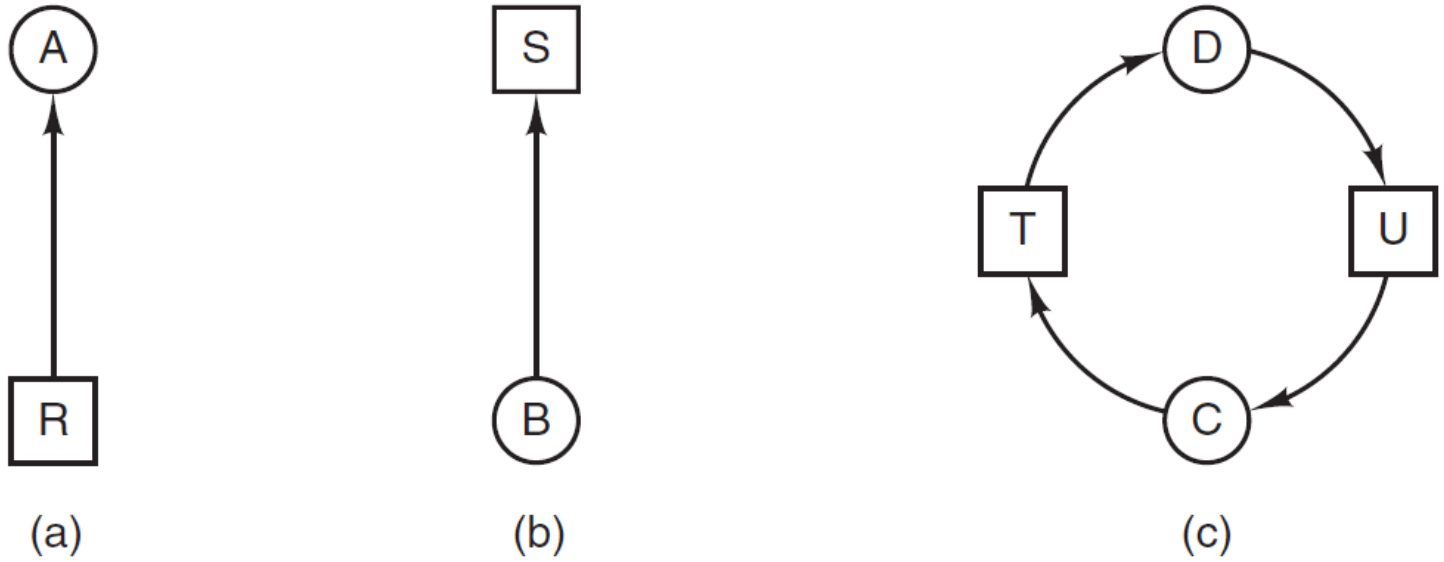
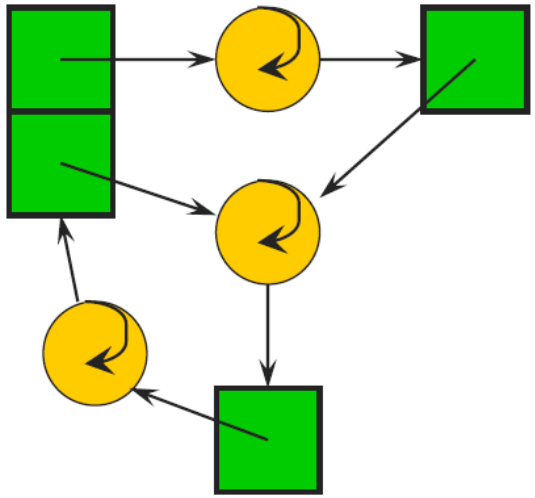


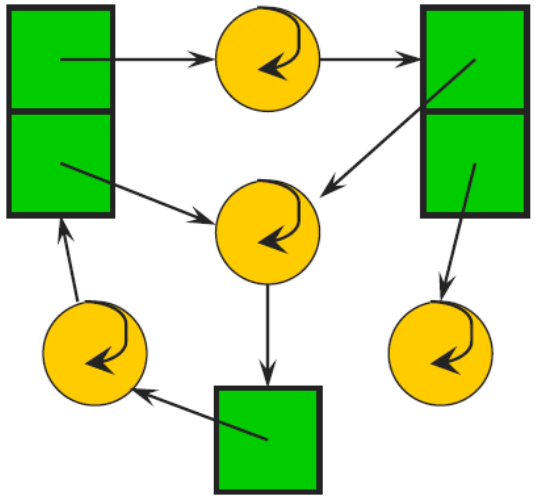
Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

RAG Examples



Deadlock?

Yes! There exist a cycle.



Deadlock?

No. There exist a cycle, BUT?

A Simpler Case

- If all resources are single unit and all processes make single requests, then we can represent the resource state with a simpler waits-for graph (WFG)
- The WFG consists of a set of vertices $P=\{P_1, P_2, \ldots, P_n\}$ of processes
  - A directed edge $P_i \rightarrow P_j$ means that P_i has requested a resource that P_j currently holds
- If the graph has no cycles, deadlock cannot exist
- If the graph has a cycle, deadlock exists

Questions to Ponder

- How many different deadlock scenarios if there are two (2) nodes in a WFG?
- How many different deadlock scenarios if there are three (3) nodes in a WFG?
- How many different deadlock scenarios if there are four (4) nodes in a WFG?
- How many different deadlock scenarios if there are N nodes in a WFG?

This Lecture

Deadlocks

Deadlock overview

Dealing with deadlocks

Dealing with Deadlocks

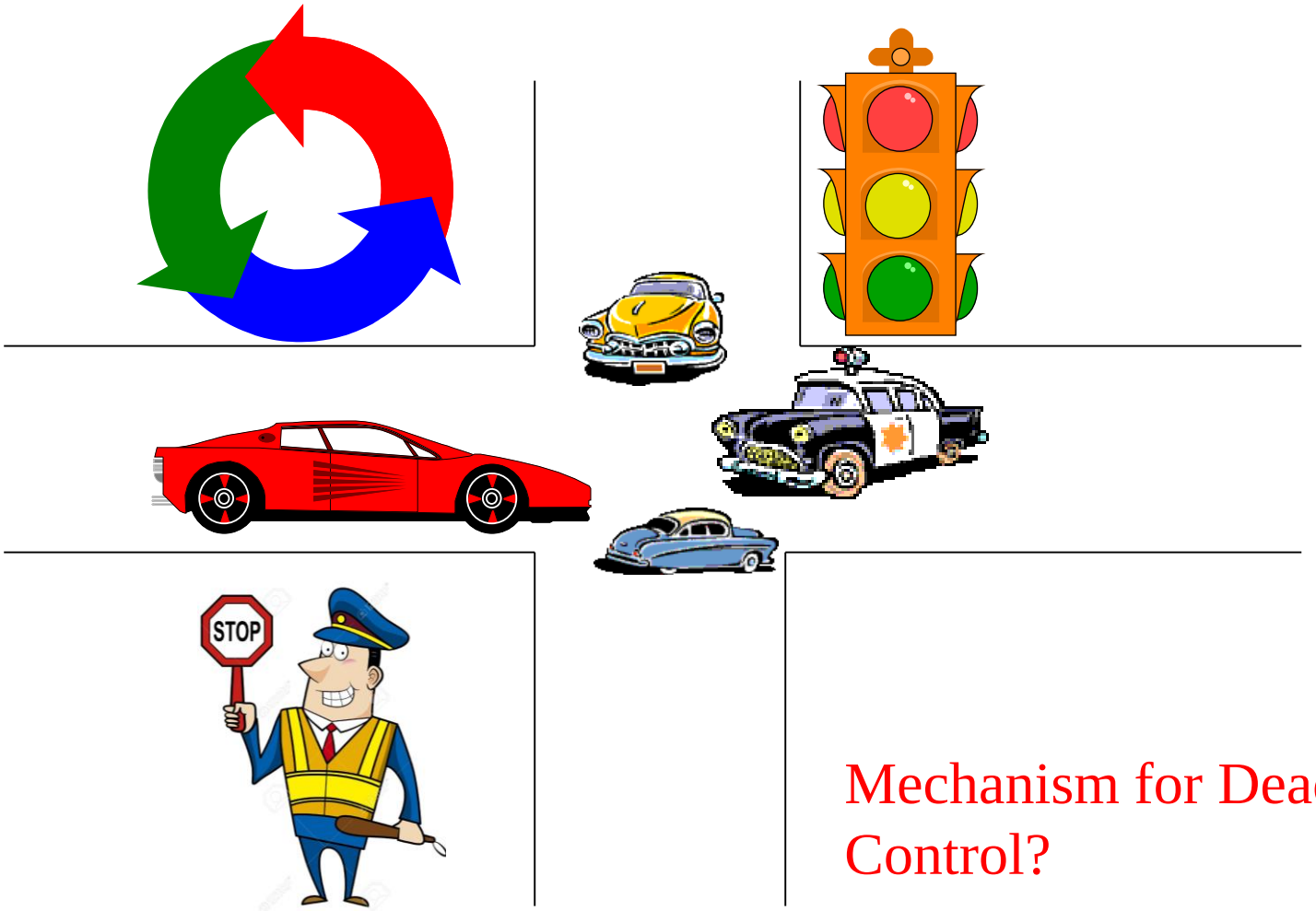
- There are four approaches for dealing with deadlock
  - Ignore it (the ostrich algorithm) – how lucky do you feel?
  - Prevention – make it impossible for deadlock to happen
  - Avoidance – control allocation of resources control allocation of resources
  - Detection and Recovery – look for a cycle in dependencies

The Ostrich Algorithm

- Don't do anything, simply restart the system (stick your head into the sand, pretend there is no problem at all)

- Rational: make the common path faster and more reliable
  - Other deadlock handling algorithms are expensive
  - If deadlock occurs only rarely, it is not worth the overhead to implement any of these algorithms

Dealing with Deadlocks



Mechanism for Deadlock Control?

Deadlock Prevention

- Prevention: Ensure that at least one of the necessary conditions cannot happen
  - *Mutual exclusion*
    - Make resources sharable (not generally practical)
  - *Hold and Wait*
    - Process cannot hold one resource when requesting another
    - Process requests, releases all needed resources at once
  - *Preemption*
    - OS can preempt resource (costly)
    - Virtualization. Normally a virtualized resource can be preempted (rather than a physical resource such as a printer)
  - *Circular wait*
    - Impose an ordering (numbering) on the resources and request them in order (popular implementation technique)

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 the second phase,
  - performing updates
  - releasing locks
- Note: similar to requesting all resources at once

Break Circular Wait Condition

- Method 1: request one resource at a time. Release the current resource when request the next one
- Method 2: global ordering of resources
  - Requests have to be made in increasing order
  - req(resource1), req(resource2), …
  - Why no circular wait?

Summary of Deadlock Prevention

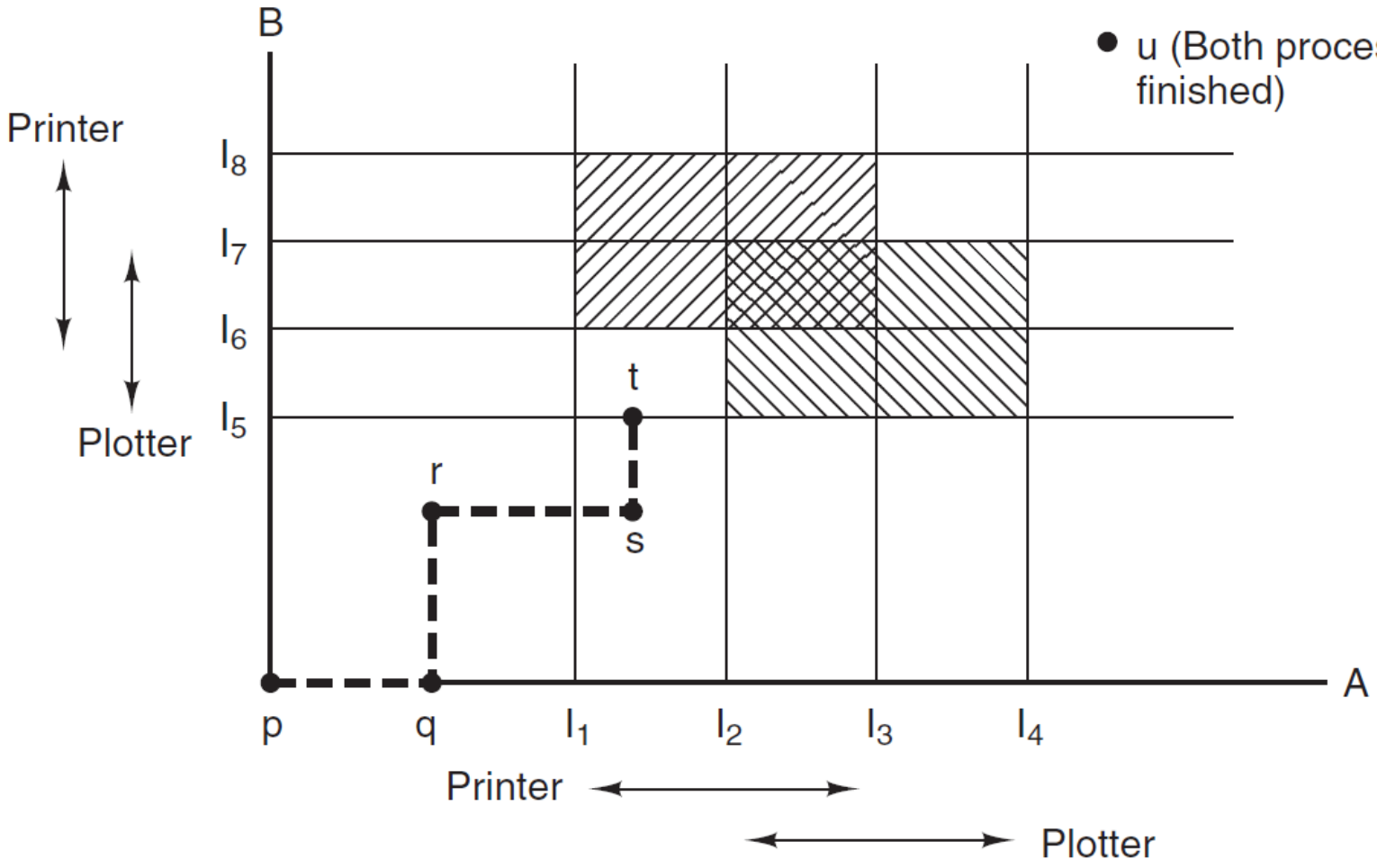
Condition	How to break it
Mutual Exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Virtualize the physical resources and then take resources away
Circular wait	Order resources numerically

Deadlock Avoidance

- Goal of deadlock avoidance: impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches

- The system needs to know the resource requirement ahead of time
- Banker's Algorithm (Dijkstra, 1965)

Resource Trajectory



Two process resource trajectories

Safe State and Unsafe State

- Safe State
  - There is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
  - From safe state, the system can guarantee that all processes will finish
- Unsafe state: no such guarantee
  - Not a deadlocked state
  - Some process may be able to complete

Banker's Algorithm

- The Banker's Algorithm is the classic approach to deadlock avoidance for resources with multiple units

1. Assign a credit limit to each customer (process)
   - Maximum credit claim must be stated in advance
2. Reject any request that leads to a dangerous state
   - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
   - A recursive reduction procedure recognizes dangerous states

How to Compute Safety

Given:

n kinds of resources

p processes

Set P of processes

struct {resource needs[n], owns[n]} ToDo[p]

resource_available[n]

while there exists a p in P such that

for all i (ToDo[p].needs[i] <= available[i]) // pick one that can be satisfied

{ do for all i //return the allocated resources back to system

available[i] += ToDo[p].owns[i];

P = P - p;

}

If P is empty then the system is safe

Need = Max - Alloc

	A	B	C
Total	10	5	5
Available	3	3	2

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	2	0	0		3	2	2		1	2	2
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

Is Allocation (1 0 2) to P1 Safe?

	A	B	C
Total	10	5	5
Available	3	3	2

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	2	0	0		3	2	2		1	2	2
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

Is Allocation (1 0 2) to P1 Safe?

	A	B	C
Total	10	5	5
Available	2	3	0

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	3	0	2		3	2	2		0	2	0
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

And Run Safety Test

	A	B	C
Total	10	5	5
Available	2	3	0

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	3	0	2		3	2	2		0	2	0
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

If P1 requests max resources, it can complete

Allocate to P1, Then

	A	B	C
Total	10	5	5
Available	2	1	0

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	3	2	2		3	2	2		0	0	0
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

Release – P1 Finishes

	A	B	C
Total	10	5	5
Available	5	3	2

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	0	0	0		3	2	2		3	2	2
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

Now P3 can acquire max resources and release

Release – P3 Finishes

	A	B	C
Total	10	5	5
Available	7	4	3

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	0	0	0		3	2	2		3	2	2
P2	3	0	0		9	0	2		6	0	2
P3	0	0	0		2	2	2		2	2	2
P4	0	0	2		4	3	3		4	3	1

Now P4 can acquire max resources and release

Release – P4 Finishes

	A	B	C
Total	10	5	5
Available	7	4	5

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	0	0	0		3	2	2		3	2	2
P2	3	0	0		9	0	2		6	0	2
P3	0	0	0		2	2	2		2	2	2
P4	0	0	0		4	3	3		4	3	3

Now P2 can acquire max resources and release

Release – P2 Finishes

	A	B	C
Total	10	5	5
Available	10	4	5

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	0	0	0		3	2	2		3	2	2
P2	0	0	0		9	0	2		9	0	2
P3	0	0	0		2	2	2		2	2	2
P4	0	0	0		4	3	3		4	3	3

Now P0 can acquire max resources and release

So P1 Allocation (1 0 2) Is Safe

	A	B	C
Total	10	5	5
Available	2	3	0

Process	Alloc				Max				Need		
	A	B	C		A	B	C		A	B	C
P0	0	1	0		7	5	3		7	4	3
P1	3	0	2		3	2	2		0	2	0
P2	3	0	0		9	0	2		6	0	2
P3	2	1	1		2	2	2		0	1	1
P4	0	0	2		4	3	3		4	3	1

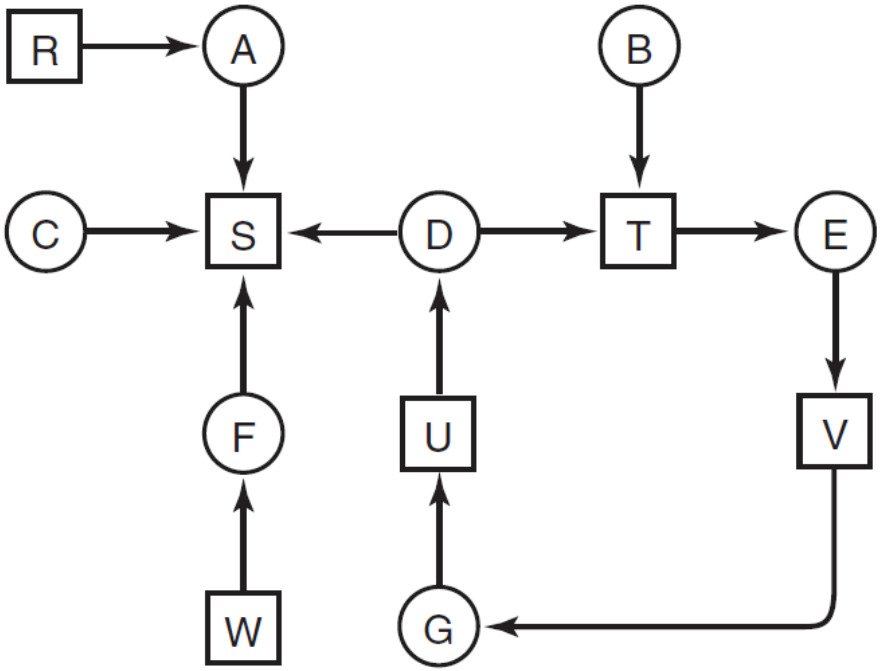
Detection and Recovery

- Detection and recovery
  - If we don't have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it
- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock
- Possible, but expensive (time-consuming)
  - Implemented in VMS (DEC)
  - Run detection algorithm when resource request times out

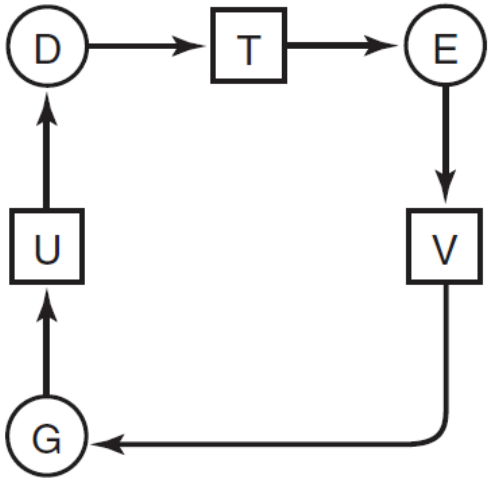
Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
  - If a cycle is found, may preempt resource (force a process to release)
- Expensive
  - Many processes and resources to traverse
- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many processes are likely to be affected when it occurs

Deadlock Detection: Example



(a) A resource graph



(b) A cycle extracted from (a).

Deadlock Detection: Algorithm

1. For each node, *N*, in the graph, perform the following five steps with *N* as the starting node.
2. Initialize *L* to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of *L* and check to see if the node now appears in *L* two times. If it does, the graph contains a cycle (listed in *L*) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

Other Detection Algorithms

- More efficient algorithm do exist
  - For example, [Even 1979]

- The previous algorithm only works for situation with one resource of each type
- What if there are multiple resources of each type?
  - Read *Textbook 6.4.2. Deadlock Detection with Multiple Resources of Each Type*

Deadlock Recovery

Once a deadlock is detected, we have two options…

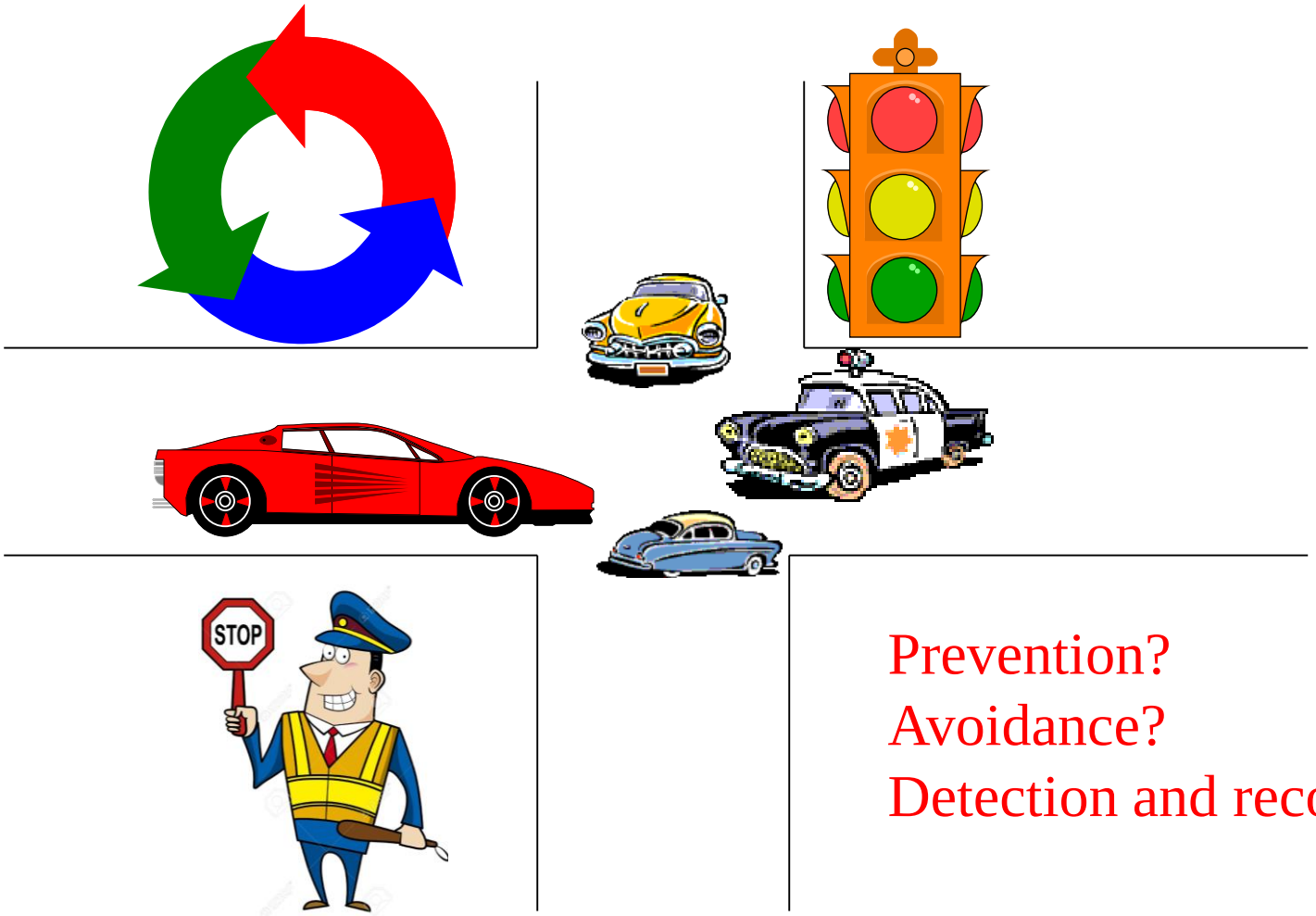
1. Abort processes

- Abort all deadlocked processes
  - Processes need start over again
- Abort one process at a time until cycle is eliminated
  - System needs to rerun detection after each abort

2. Preempt resources (force their release)

- Need to select process and resource to preempt
- Need to rollback process to previous state
- Need to prevent starvation

Dealing with Deadlocks



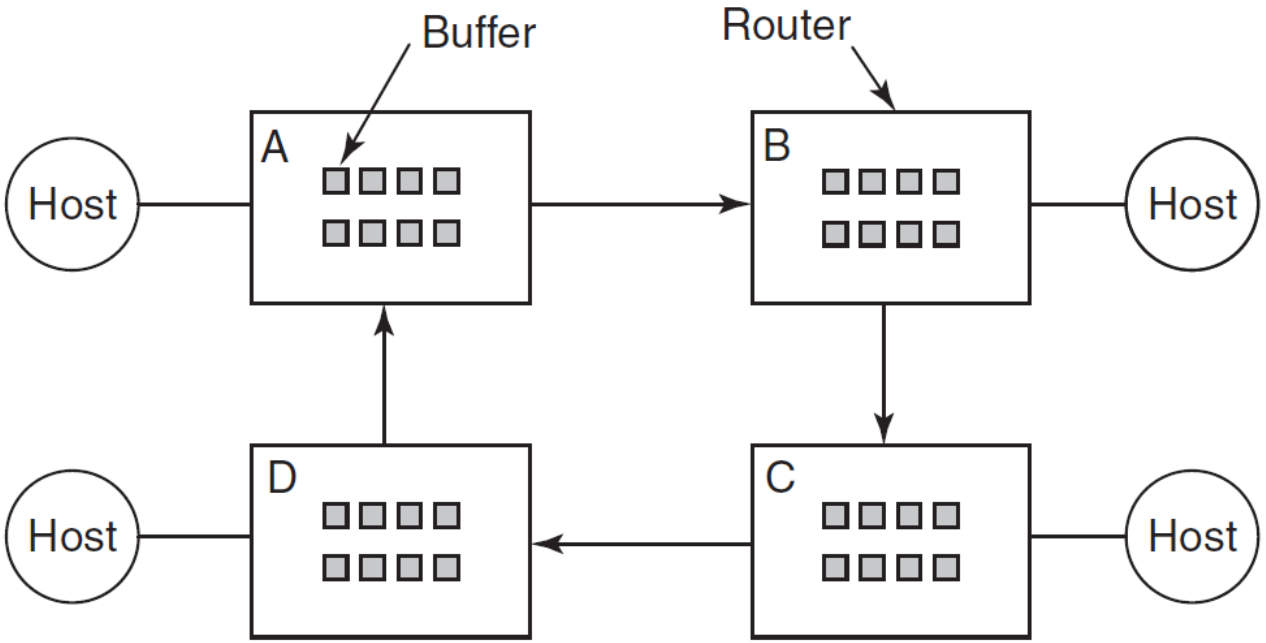
Prevention?
Avoidance?
Detection and recovery?

Communication Deadlocks

- Not all deadlocks are resource deadlocks
- In communication systems
  - Suppose two or more processes communicate by sending messages.
  - Process *A* sends a request message to process *B*, and then blocks until *B* sends back a reply message.
- What can go wrong?
- How to deal with communication deadlocks?

Resource Deadlock in Communication

- Not all deadlocks in communication systems are communication deadlocks.
- An example of resource deadlock here:



Livelock

- Bad things might happen even every process tries to behave politely
  - A process tries to be polite by giving up the locks it already acquired whenever it notices that it cannot obtain the next lock it needs.
- Livelock: no "real" progress even if nobody is really blocked

```
void process_A(void) {
        acquire_lock(&resource_1);
        while (try_lock(&resource_2) == FAIL) {
                release_lock(&resource_1);
                wait_fixed_time();
                acquire_lock(&resource_1);
        }
        use_both_resources( );
        release_lock(&resource_2);
        release_lock(&resource_1);
}

void process_A(void) {
        acquire_lock(&resource_2);
        while (try_lock(&resource_1) == FAIL) {
                release_lock(&resource_2);
                wait_fixed_time();
                acquire_lock(&resource_2);
        }
        use_both_resources( );
        release_lock(&resource_1);
        release_lock(&resource_2);
}
```

A Real-world Exmaple

- Fork(): Process-table slots are finite resources
  - If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again
- Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 children.
- After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table.
- Each of the 10 original processes now sits in an endless loop forking and failing.

  ➔ Resulting in a livelock.

Summary

- Deadlock overview
  - Deadlock conditions – mutual exclusion, hold-and-wait, no resource preemption, circular wait
- Dealing with deadlocks
  - Ignore it – Living life on the edge
  - Prevention – Make one of the four conditions impossible
  - Avoidance – Banker's algorithm (control allocation to find a safe way)
  - Detection and recovery – Look for a cycle, preempt or abort
- Next lecture: File System