EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Slides from Christopher Kruegel, UCSB

Deadlocks

When processes try to acquire resources concurrently they may end up “stuck”

Example:

1. Process A needs P, Q
2. Process B needs Q, P
3. Process A gets P
4. Process B gets Q
5. Process A tries to get Q and blocks
6. Process B tries to get P and blocks

Resources

Examples of computer resources

- Printers
- Tape drives
- Tables (e.g., in memory)
- individual data structures

Resources can be available

- In a single instance (e.g., one printer)
- In multiple identical copies (e.g., an array of tape drives)

Resources can be

- Preemptable: the resource can be taken away from a process with no negative side-effects
- Nonpreemptable: taking away the resource will cause the process to fail

Accessing Resources

Deadlocks occur when processes are granted exclusive access to non-preemptable resources

Sequence of events required to use a resource

- Request the resource
- Use the resource
- Release the resource

If request is denied

- Requesting process may be blocked
- May fail with error code

Defining Deadlocks

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 the release of a currently held resource

None of the processes can

- Run
- Release resources
- Be awakened

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

Remember Producer/Consumer with a Mistake …

Producer

```
item = produce_item()

mutex.down();
empty.down();

insert_item(item);

mutex.up();
full.up();
```

Consumer

```
full.down();
mutex.down();

item =
remove_item();

mutex.up();
empty.up();

consume_item(item);
```

Four Conditions for Deadlock

1. **Mutual exclusion condition**

   Each resource is assigned to exactly one process or is available

2. **Hold and wait/spin condition**

   A process holding resources can request additional ones

3. **No preemption condition**

   Previously granted resources cannot forcibly be taken away

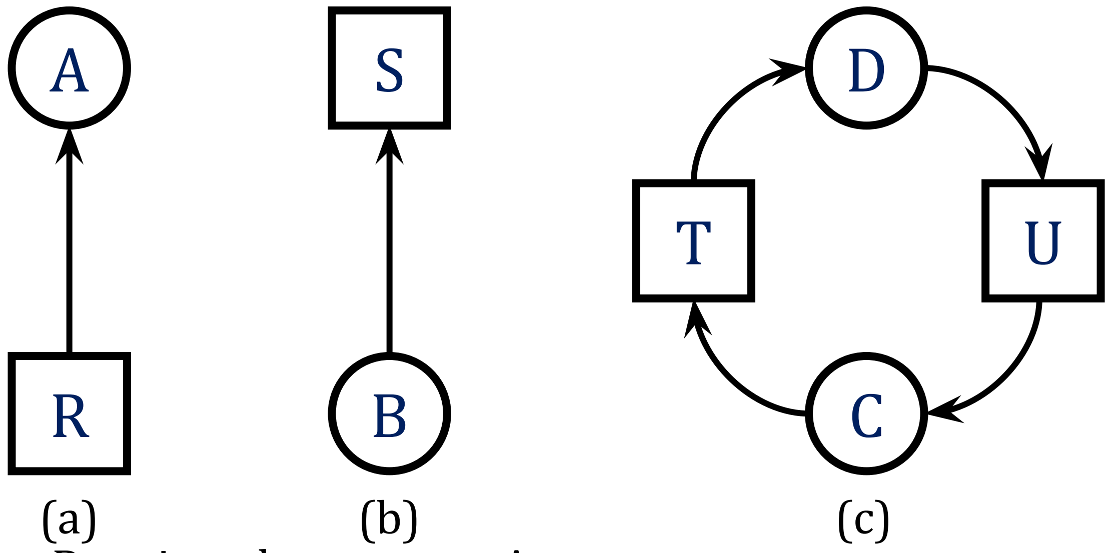
4. **Circular wait condition**

   There must be a circular chain of two or more processes, each of which is waiting for a resource held by next member of the chain

Deadlock Modeling

Modeled with directed graphs

- Processes: circles
- Resources: squares



(a) Resource R assigned to process A

(b) Process B is requesting/waiting for resource S

(c) Process C and D are in deadlock over resources T and U

A simple Example

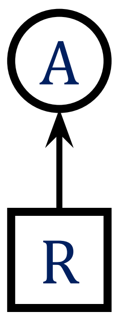
A

- Request R
  Request S

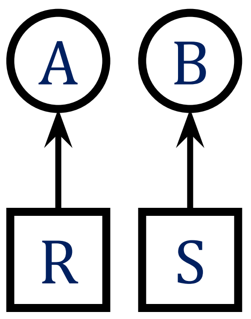
B

- Request S
  Request R

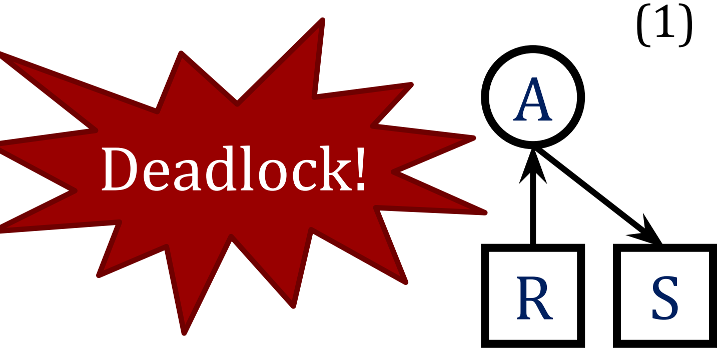
1. A requests R
2. B requests S
3. A request S
4. B requests R



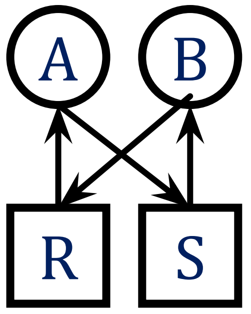
(1)



(2)



(4)



(5)

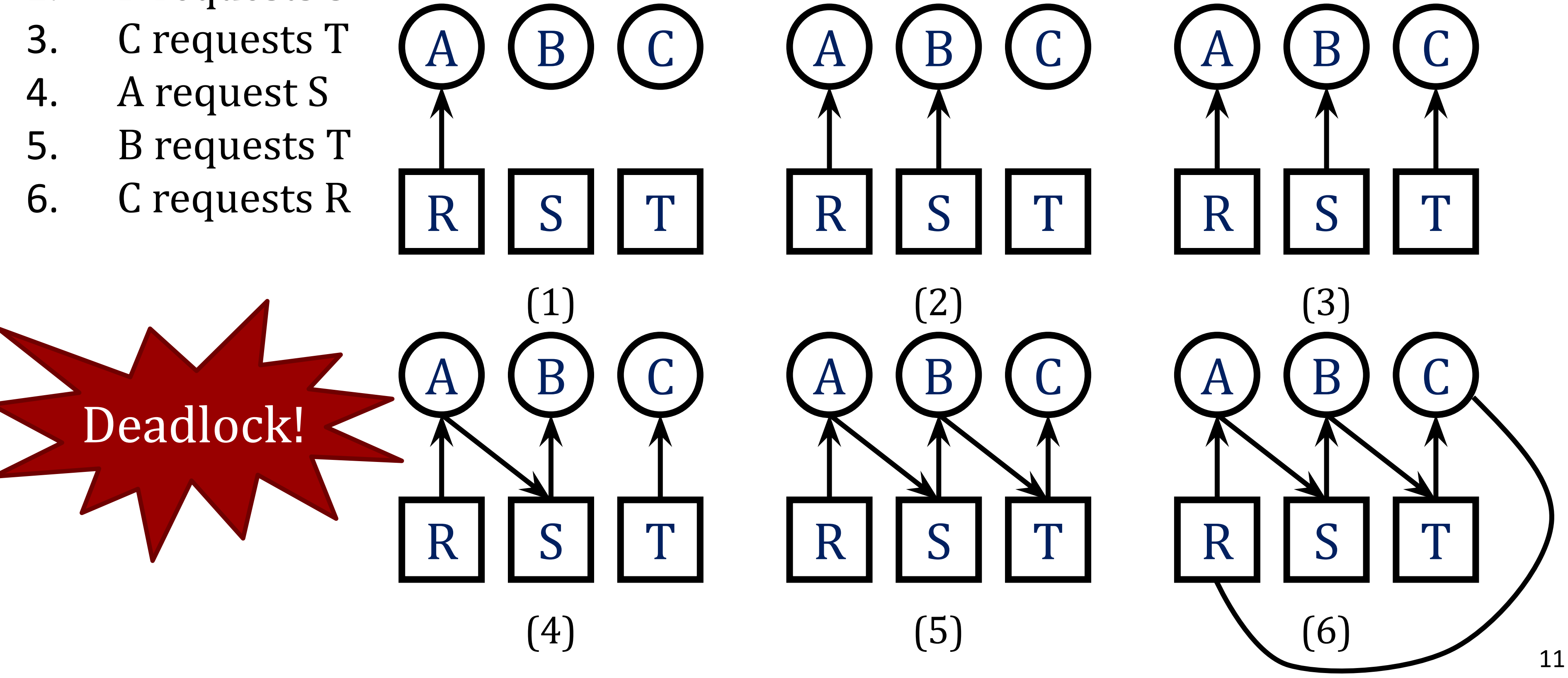
An Example

A
- Request R
- Request S
- Release R
- Release S

B
- Request S
- Request T
- Release S
- Release T

C
- Request T
- Request R
- Release T
- Release R

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

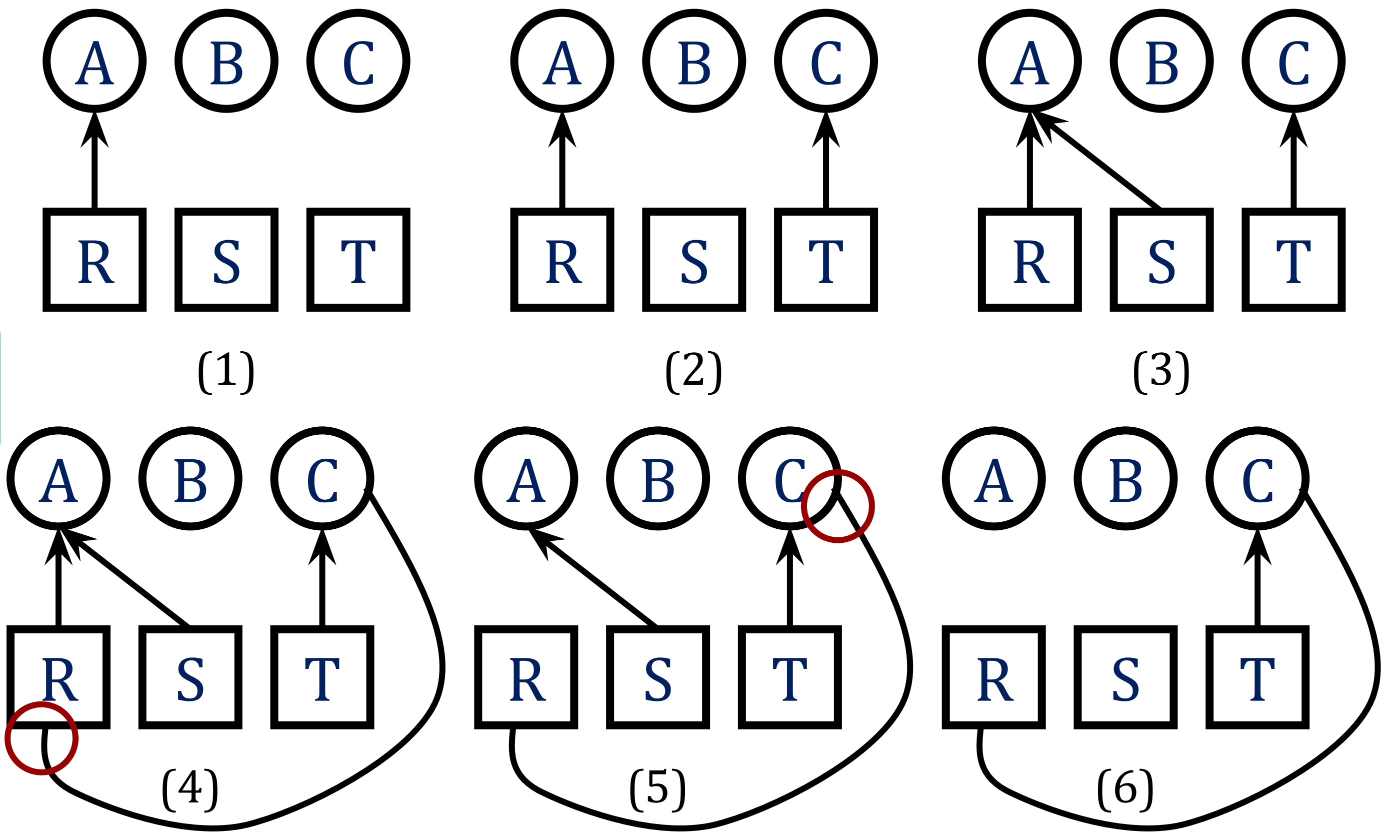


Another Example

A	B	C
● Request R	Request S	● Request T
● Request S	Request T	● Request R
● Release R	Release S	Release T
● Release S	Release T	Release R

1. A requests R
2. C requests T
3. A requests S
4. C request R
5. A releases R
6. A releases S

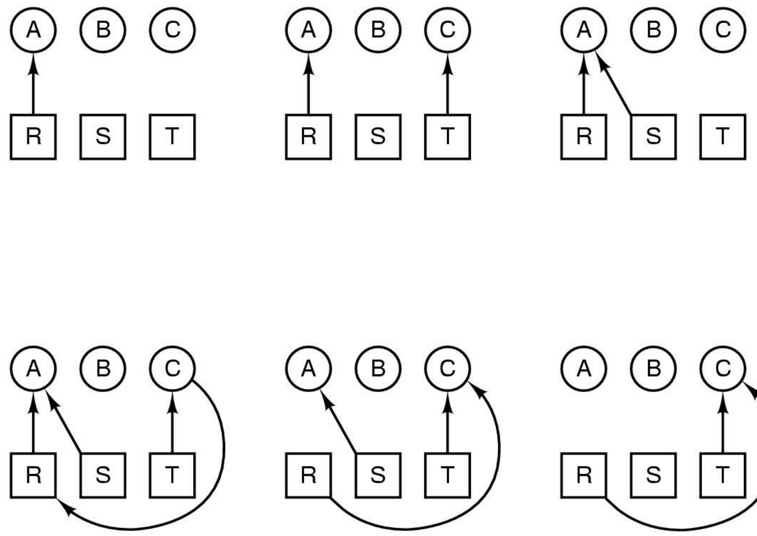
No Deadlock!



Another Example

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock



Dealing With Deadlocks

Just ignore the problem altogether

- Bad things happen!

Detection and recovery

- Let them occur and deal with it

Dynamic avoidance

- Careful resource allocation

Prevention

- Negating one of the four necessary conditions

Dealing With Deadlocks

Just ignore the problem altogether

Detection and recovery

Dynamic avoidance

Prevention

The Ostrich Algorithm

Pretend there is no problem

Reasonable if

- Deadlocks occur very rarely
- Cost of prevention is high

UNIX and Windows takes this approach for many resources

It is a trade off between

- Convenience
- Correctness

Dealing With Deadlocks

Just ignore the problem altogether

Detection and recovery

Dynamic avoidance

Prevention

Detection And Recovery

Let deadlocks happen and deal with the situation

Need to detect:

– Deadlock detection algorithms

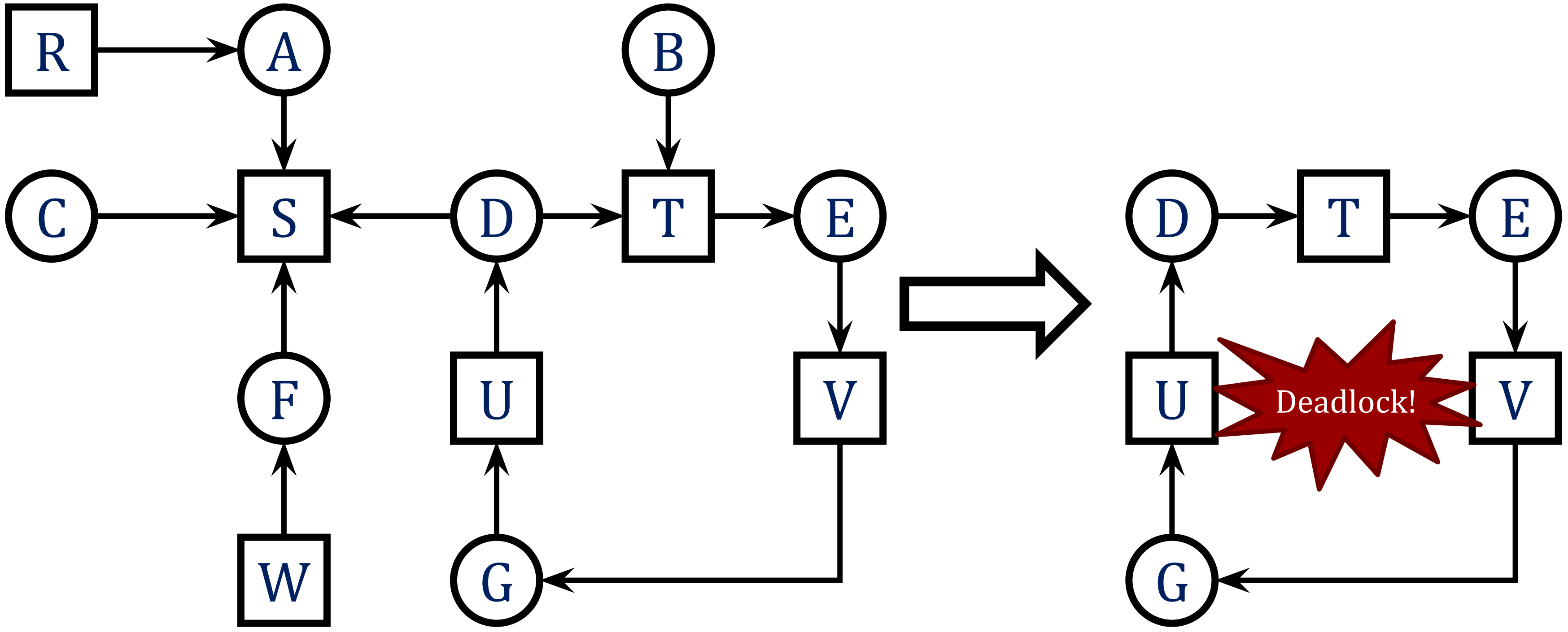
Need to recover:

– Preemption, Rollback, Killing

Detection with One Resource of Each Type

Note the resource ownership and requests

If a cycle can be found within the graph, then there is a deadlock



Detection with One Resource of Each Type

L: list of nodes

Arcs can be marked to indicate that they have been inspected

1. For each node N in the graphs do the following
2. L := empty, arcs all unmarked
3. Add current node to L and check if it appears two times
    1. Yes: there is a cycle
    2. No: continue
4. Are there outgoing, unmarked arcs? If not go to step 6
5. Pick randomly an unmarked arc and mark it, follow the arc to the node and go to step 3
6. Remove current node from the list, go back to the previous node, and jump to step 3. If this is the root node then there are no cycles

Detection with Multiple Resources of Each Type

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

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

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

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

Invariant

Sum of current resource allocation + resources available = resources that exist

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Detection with Multiple Resources of Each Type

- Comparing vectors:

$$A < B \quad \text{iff}$$
$$\forall i \in 1 \ldots n : A_i < B_i$$

- Initially all processes are unmarked (not deadlocked)
- Look for a process P_i for which the corresponding row in request matrix (R_i) is less than or equal to availability vector (A)
  - If such process exists add the corresponding row of the current allocation matrix (C_i) to A, mark the process and restart to look
  - If there is no such process then exit
- At the end, unmarked processes are in deadlock

Detection with Multiple Resources of Each Type

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

Current allocation matrix

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

Request matrix

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

Detection with Multiple Resources of Each Type

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

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \qquad R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Detection with Multiple Resources of Each Type

Find i s.t., $R_i < A$

$i = 3, A \mathrel{+}= C_i$

$E = (4\ \ 2\ \ 3\ \ 1)$

$A = (2\ \ 1\ \ 0\ \ 0)$

$$C = \begin{array}{c} \\ \\ \Rightarrow \end{array} \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$\begin{array}{c} \Rightarrow \\ \Rightarrow \\ \Rightarrow \end{array} \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ \cancel{2} & \cancel{1} & \cancel{0} & \cancel{0} \end{pmatrix}$$

Detection with Multiple Resources of Each Type

Find i s.t., $R_i < A$

$i = 2, A += C_i$

$E = (4 \;\; 2 \;\; 3 \;\; 1)$

$A = (2 \;\; 2 \;\; 2 \;\; 0)$

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} \qquad R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ \cancel{1} & \cancel{0} & \cancel{1} & \cancel{0} \\ \cancel{2} & \cancel{1} & \cancel{0} & \cancel{0} \end{pmatrix}$$

Detection with Multiple Resources of Each Type

Find i s.t., $R_i < A$

$i = 1, A \mathrel{+}= C_i$

$E = (4\ \ 2\ \ 3\ \ 1)$

$A = (4\ \ 2\ \ 2\ \ 1)$

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

$$\rightarrow R = \begin{pmatrix} \cancel{2} & \cancel{0} & \cancel{0} & \cancel{1} \\ \cancel{1} & \cancel{0} & \cancel{1} & \cancel{0} \\ \cancel{2} & \cancel{1} & \cancel{0} & \cancel{0} \end{pmatrix}$$

Detection with Multiple Resources of Each Type

$E = (4 \;\; 2 \;\; 3 \;\; 1)$

$A = (4 \;\; 2 \;\; 3 \;\; 1)$

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

$$R = \begin{pmatrix} \cancel{2} & \cancel{0} & \cancel{0} & \cancel{1} \\ \cancel{1} & \cancel{0} & \cancel{1} & \cancel{0} \\ \cancel{2} & \cancel{1} & \cancel{0} & \cancel{0} \end{pmatrix}$$

Deadlock Detection

Suppose, P3 needs 2 Tapes,
1 Plotter, 0 Scanners, and 1 CD Rom

$$E = \begin{array}{cccc} \text{Tape Drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array}$$

$$A = (2 \quad 1 \quad 0 \quad 0)$$

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

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix}$$

Dealing With Deadlocks

Just ignore the problem altogether

Detection and recovery

Dynamic avoidance

Prevention

Recovery from Deadlock

Once detected, how to recover from deadlock?

- Recovery through preemption
  - Take a resource from some other process
  - Depends on nature of the resource
- Recovery through rollback
  - Checkpoint a process periodically
  - Use this saved state
  - Restart the process if it is found deadlocked

Recovery from Deadlock

Once detected, how to recover from deadlock?

- 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

Dealing With Deadlocks

Just ignore the problem altogether

Detection and recovery

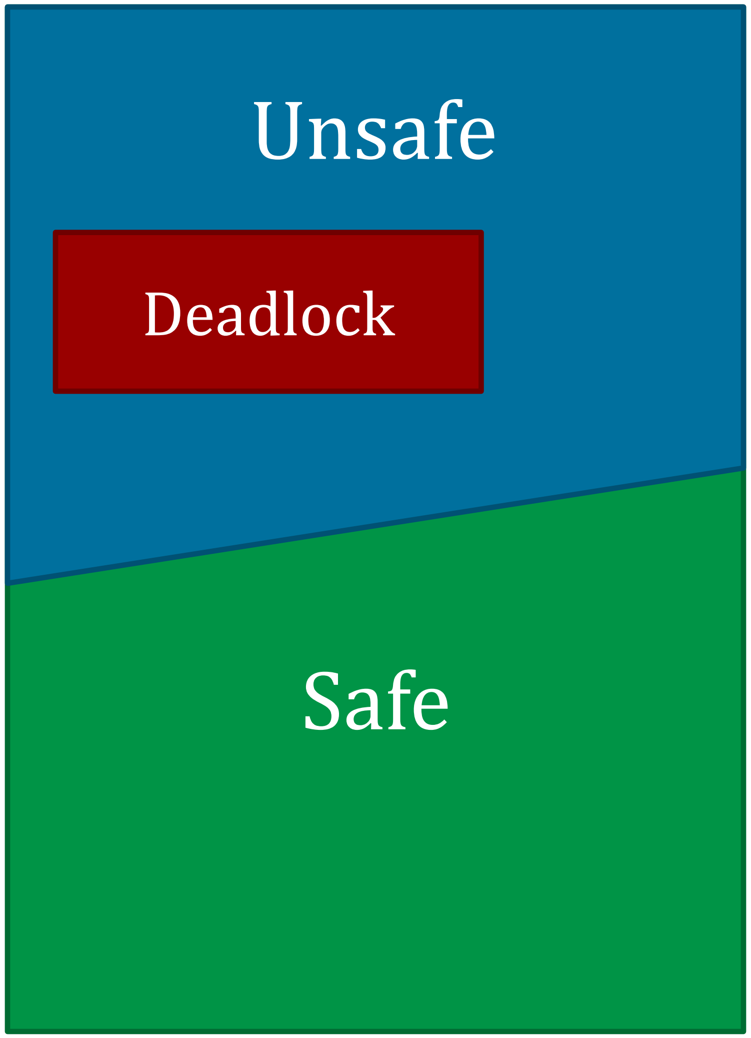
Dynamic avoidance

Prevention

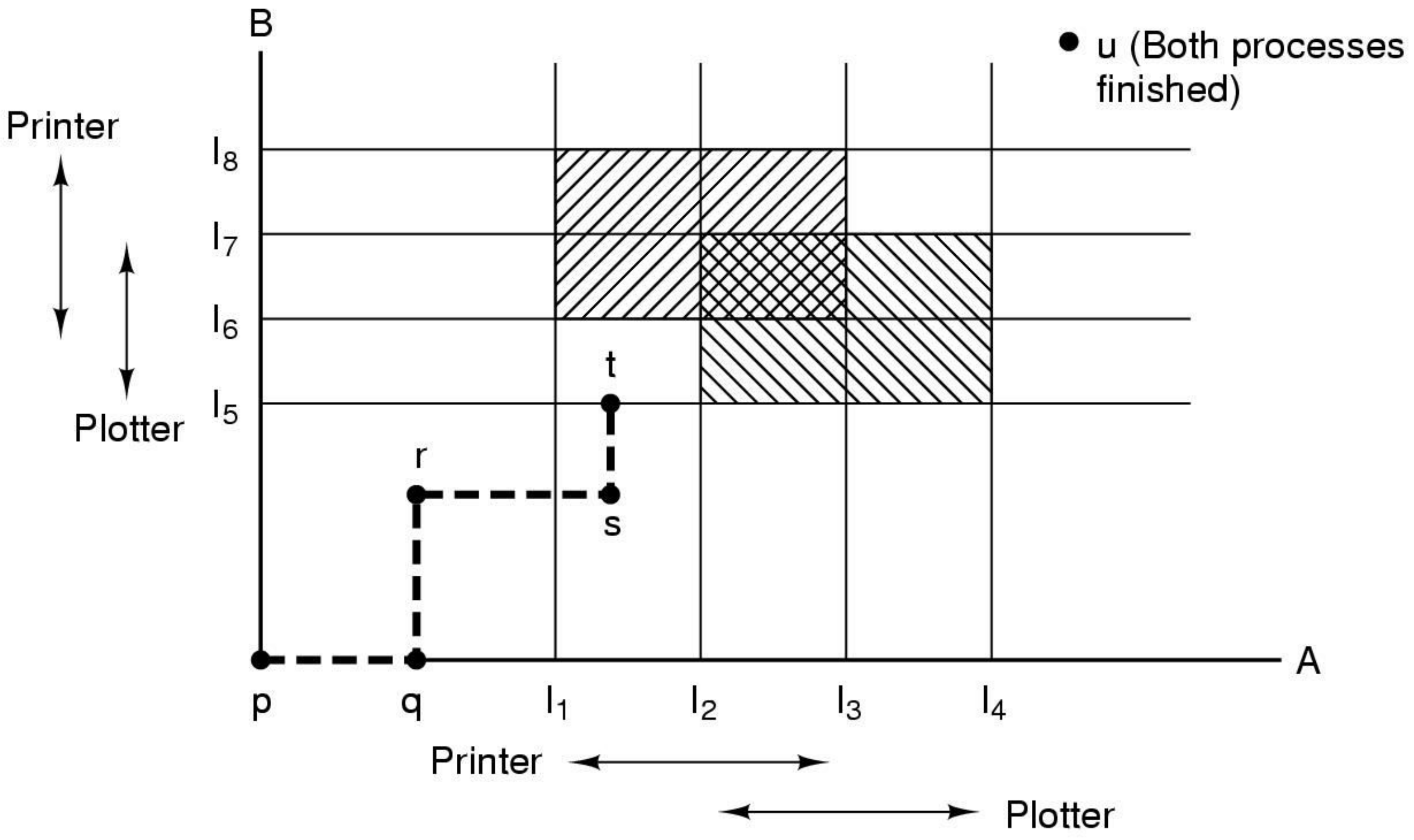
Deadlock Avoidance

- When a process requests a resource the system must decide if resource should be granted
- To avoid deadlocks system should stay in *safe state*
- State: matrices C, R, E, A
- Safe state: there is currently no deadlocked process and there is some scheduling order in which every process can run to completion, even if all the processes request all the resources at the same time

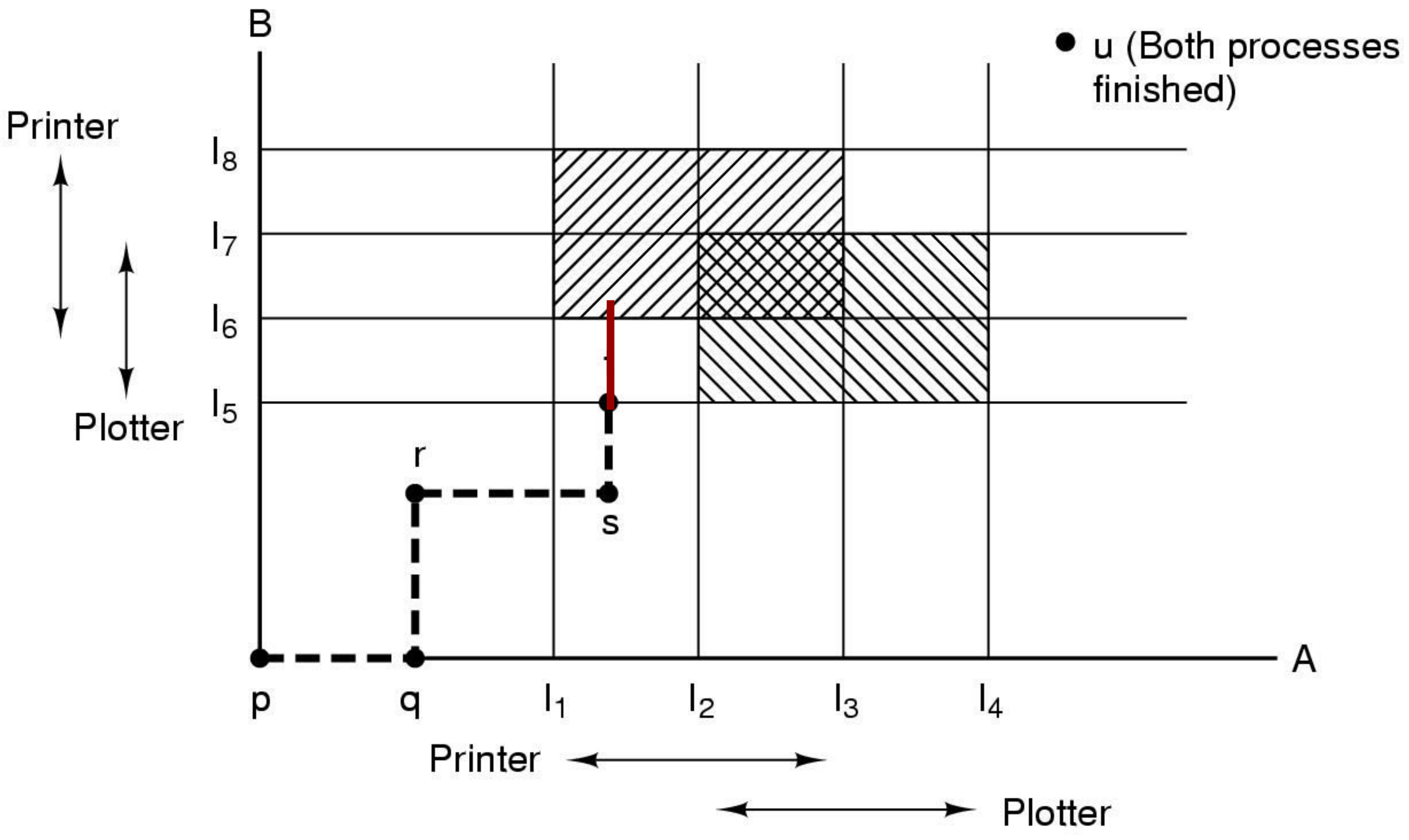
Deadlock Avoidance – States



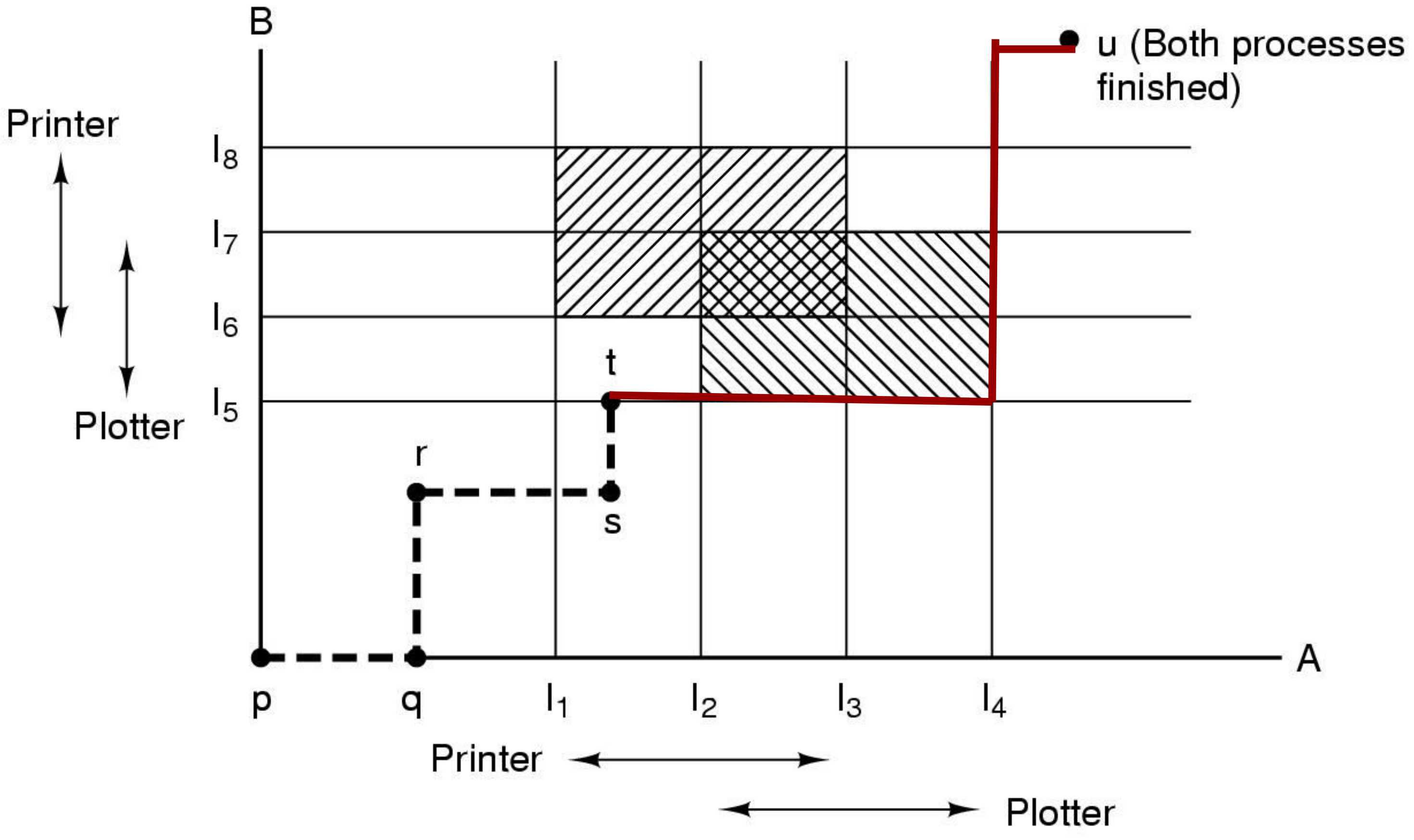
Resource Trajectories



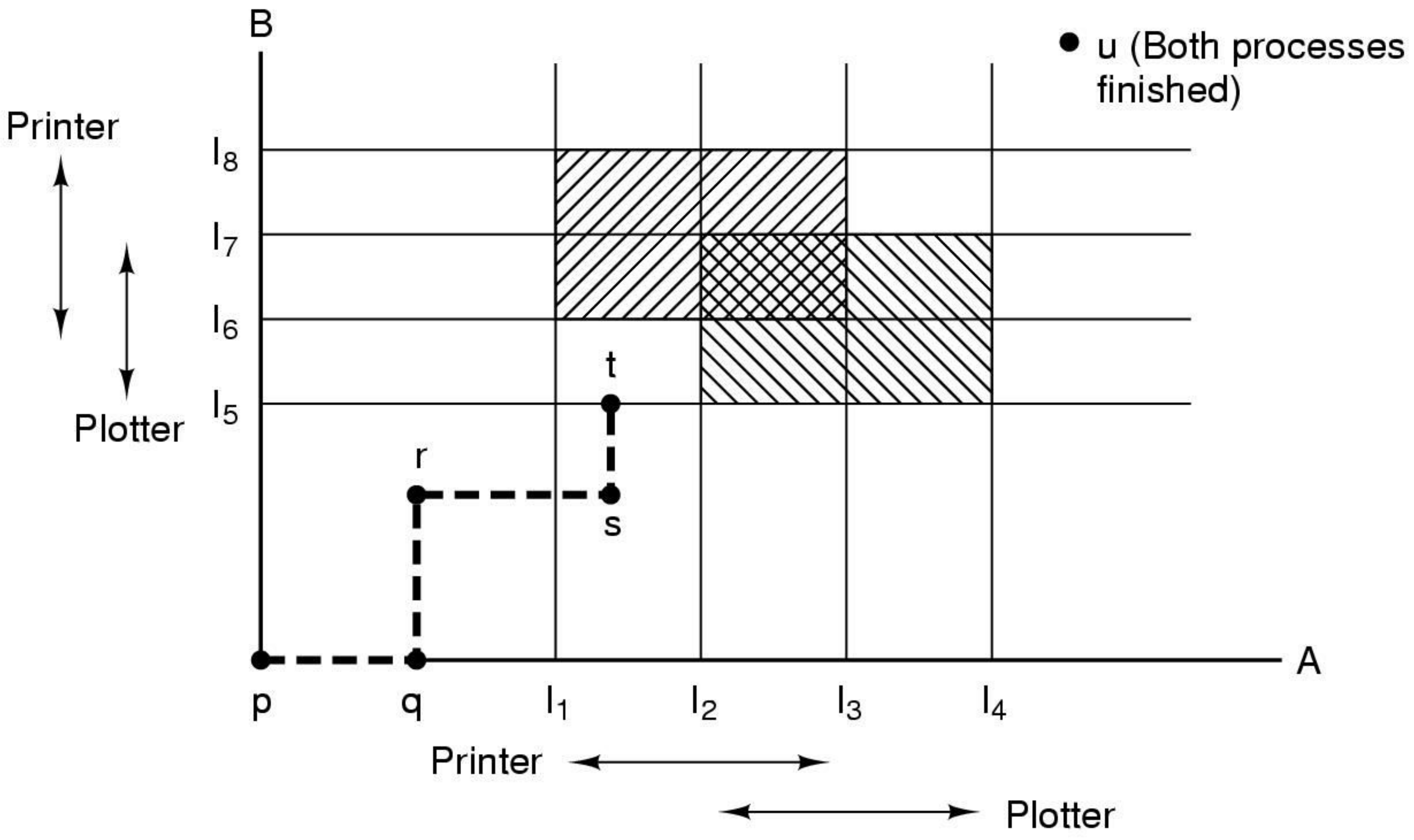
If we schedule B, what happens



A schedule that works

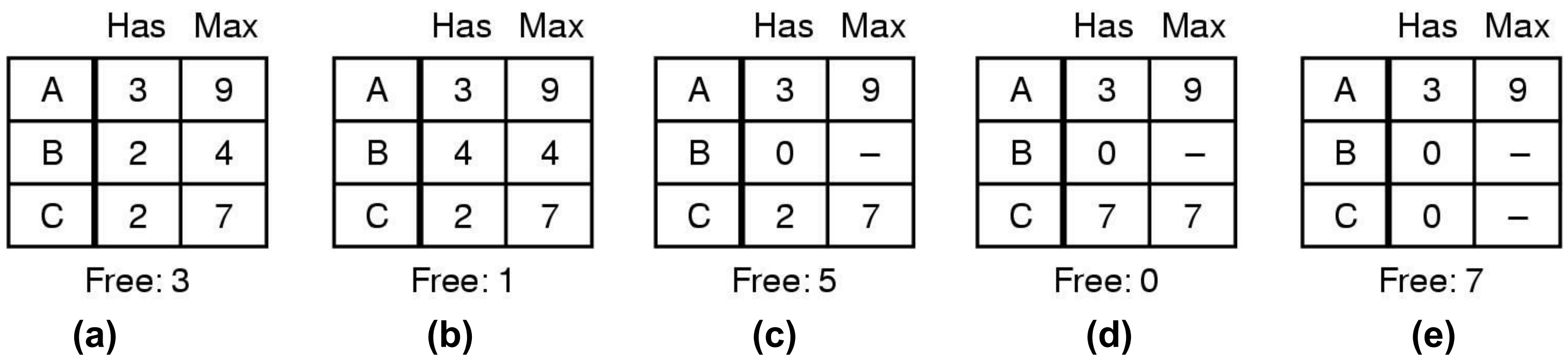


Resource Trajectories



Safe and Unsafe States

- State (a) is safe
  - Max possible allocation is A=6, B=2, C=5
  - Three are free, give two to B and let it run to completion
- State (c)
  - Max possible allocation is A=6, C=5
  - Five are free give to C and let it run to completion



(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(c)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(d)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(e)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

Total number of resources: 10

Safe and Unsafe States

Safe state: there is currently no deadlocked process and there is some scheduling order in which every process can run to completion, even if all the processes request all the resources at the same time

IS state (a) safe?

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

Total number of resources: 10

Safe and Unsafe States

Safe state: there is currently no deadlocked process and there is some scheduling order in which every process can run to completion, even if all the processes request all the resources at the same time

IS state (a) safe?

- Max possible allocation is A=6, B=2, C=5
- Three are free, give two to B and let it run to completion

	Has	Max
A	**3**	**9**
B	**2**	**4**
C	**2**	**7**

Free: 3
(a)

Total number of resources: 10

Safe and Unsafe States

Safe state: there is currently no deadlocked process and there is some scheduling order in which every process can run to completion, even if all the processes request all the resources at the same time

IS state (a) safe?

- Max possible allocation is A=6, B=2, C=5
- Three are free, give two to B and let it run to completion

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

Total number of resources: 10

Safe and Unsafe States

Safe state: there is currently no deadlocked process and there is some scheduling order in which every process can run to completion, even if all the processes request all the resources at the same time

IS state (a) saf

– Max possi

– Three are on

A is Safe!!

State (c)

– Max possible allocation is A=6, C=5

– Five are free give to C and let it run to completion

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	--
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	--
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	--
C	0	--

Free: 7
(e)

Total number of resources: 10

Safe and Unsafe States

Safe state: there is currently no deadlocked process and there is some scheduling order in which every process can run to completion, even if all the processes request all the resources at the same time

State (a) is safe

- Max possible allocation is A=6, B=2, C=5
- Three are free, give two to B and let it run to completion

State (c)

- Max possible allocation is A=6, C=5
- Five are free give to C and let it run to completion

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	--
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	--
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	--
C	0	--

Free: 7
(e)

Total number of resources: 10

Moving to an Unsafe State

- Imagine if scheduler from State (a) gives one resource to A

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	0	--
C	2	7

Free: 4
(d)

Deadlock!

Total number of resources: 10

Moving to an Unsafe State

- From State (a) one resource is given to A
- State (b) is unsafe because there is not a scheduling order in which every process can run to completion if all the processes request all the resources at the same time
- B gets 2 and returns 4, but both A and C need 5

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

Safe and Unsafe States

Unsafe states are not necessarily deadlocked

- With a lucky sequence, all processes may complete
- However, we cannot guarantee that they will complete (not deadlock)

Safe states guarantee we will eventually complete all processes

Deadlock avoidance algorithm

- Only grant requests that result in safe states

The Banker's Algorithm

- Algorithm considers each request and examines if it leads to a safe state
  - Check if there are enough resources to satisfy at least one process
  - Sum the resources of the process to those available, mark the process and iterate
  - If at the end there are processes that are left unmarked the process would lead to an unsafe state

- If granting the request would lead to an unsafe state then resource is not granted

The Banker’s Algorithm

Modeled after a banker with customers

- The banker has a limited amount of money to loan customers
  - Limited number of resources
- Each customer can borrow money up to the customer’s credit limit
  - Maximum number of resources required

Basic idea

- Keep the bank in a safe state
  - So all customers are happy even if they all request to borrow up to their credit limit at the same time.
- Customers wishing to borrow such that the bank would enter an unsafe state must wait until somebody else repays their loan such that the transaction becomes safe.

Banker’s Algorithm for Multiple Resources (2)

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, system will eventually deadlock.
2. Assume the process of row chosen requests all resources needed and finishes. Mark that process as terminated, add its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (safe state) or no process is left whose resource needs can be met (deadlock)

The Banker's Algorithm for a Single Resource

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

Free: 10
(a)

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

Free: 2
(b)

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

Free: 1
(c)

Three resource allocation states:

(a) safe
(b) safe
(c) unsafe

The Banker's Algorithm

- B requests a scanner. Should the scanner be granted?
- Then E requests the last scanner. Should it be granted?

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)

Demo Spreadsheet

The Banker’s Algorithm

Not commonly used in practice

It is difficult (sometimes impossible) to know in advance

- the resources a process will require
- the number of processes in a dynamic system

Dealing With Deadlocks

Just ignore the problem altogether

Detection and recovery

Dynamic avoidance

Prevention

Deadlock Prevention

Necessary Conditions for Deadlock:

1. Mutual exclusion condition

    Each resource is assigned to exactly one process or is available

2. Hold and wait condition

    A process holding resources can request additional ones

3. No preemption condition

    Previously granted resources cannot forcibly be taken away

4. Circular wait condition

    There must be a circular chain of two or more processes, each of which is waiting for a resource held by next member of the chain

Prevent Deadlock by invalidating at least 1 condition

Attacking the Mutual Exclusion Condition

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

Attacking the Hold and Wait Condition

- Require processes to request all their resources before starting
  - A process never has to wait for what it needs

- Problems
  - Process may not know required resources at start of run
  - Ties up resources that other processes could be using

- Possible solution
  - Process must give up all resources before acquiring a new one
  - Then request all needed resources at once

Two-Phase Locking

- Phase One
  - Process tries to lock all records it needs, one at a time
  - If needed record found locked, release all the locks and start over

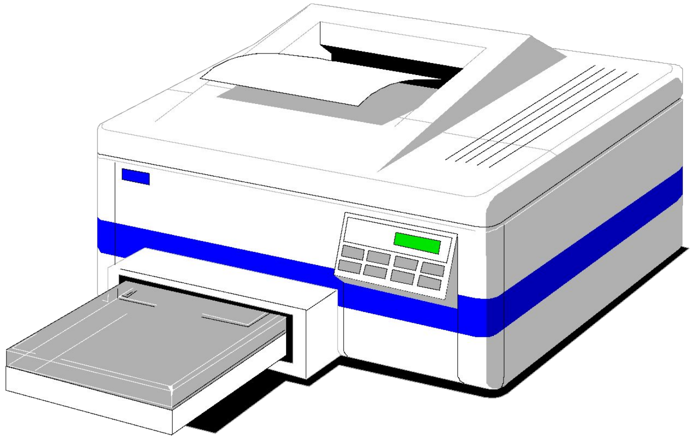
- If phase one succeeds, it starts second phase
  - Performing updates
  - Releasing locks

- Similar to requesting all resources at once

Attacking the No Preemption Condition

- This is not a very appealing option

- Consider a process that is using a printer
  - Let process go halfway through its job
  - Then forcibly take away printer
  - Results can be unpredictable



Attacking the Circular Wait Condition

- Require a process to request/hold only one resource at a time
  - Not realistic

- Provide global numbering of resources and require ordered acquisitions
  - A process holding resource *j* cannot ask for resource *i*, with $i < j$
  - The resulting resource graph guaranteed to be cycle-free

Livelock

- Sometimes processes can never be deadlocked, but still not make progress

- Consider this algorithm, they keep running... but, it is possible for neither to ever make progress

- Lots of real examples of this in OS, e.g. with two phased locking

```
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);
}
```

Starvation

- A process never receives the resource it is waiting for, despite the resource (repeatedly) becoming free, the resource is always allocated to another waiting process.

- Algorithm to allocate a resource
  - May be to give to shortest job first

- Works great for multiple short jobs in a system

- May cause long job to be postponed indefinitely
  - Even though not deadlocked

- Solution:
  - First-come, first-serve policy

Deadlock in Linux

Focus on Synchronization

- Linux takes the "Ostrich" approach to solving user deadlocks; but there is enormous focus on preventing kernel deadlock, starvation and livelock
- Resource issues arise on kernel data structures, e.g., task table, inode table...
- Most of the focus goes to synchronization and prevention.

Locking In the Linux kernel

- semaphores/counting semaphores
  - multiple count resources blocking
  - example: free objects on a list
- mutexes
  - can be reader/writer
  - single count resources blocking
  - example: file access
- spinlocks
  - can be reader/writer
  - reader: multiple access non-blocking
  - writer: single access non-blocking
  - can not block with spinlock held
- atomic operations
  - incrementing/decrementing counters
- RCU
  - read copy update

Locking in the Linux Kernel

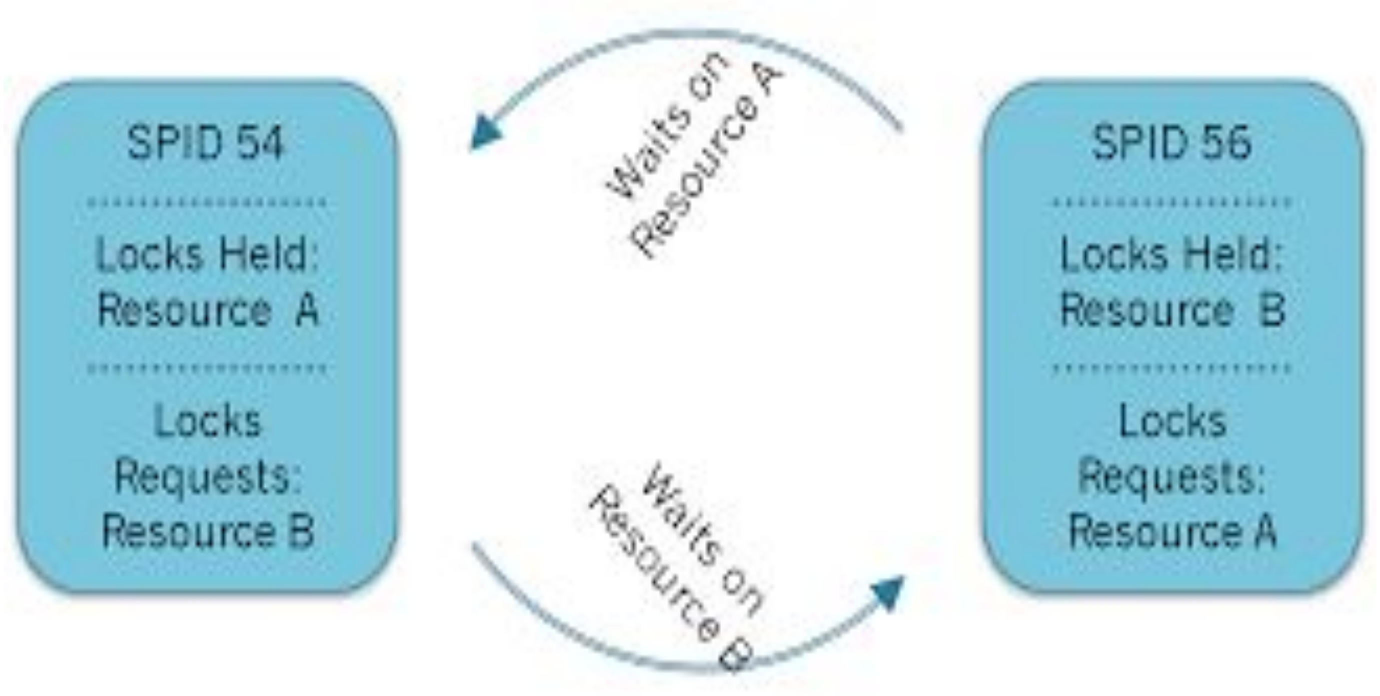
- spinlocks: non-blocking/busy-wait
  - simple spinlock: spinlock_t
    - spinlock(), spinlock_irq(), spin_trylock(), spin_unlock
  - nested spinlock: nested_spinlock_t
    - spinlock(), spinlock_irq(), spin_trylock(), spin_unlock
  - reader/writer spinlock: rw_spinlock_t
    - spin_readlock(), spin_writelock(), spinunlock()
- mutex & semaphores: blocking/waiting
  - simple mutex: mutex_t
    - down(), down_try(), up()
  - nested mutex: nested_mutex_t
    - down(), down_try(), up()
  - reader/writer mutex rw_mutex_t
    - downread(), downwrite(), up()

Problems with locks in the Linux Kernel

- Deadlock
  - one or more processes permanently stuck on spinlocks or blocked on semaphores without making progress.
  - default behavior: panic if no reschedule in 30s
- Livelock
  - one or more processes permanently stuck in a loop not making any progress.
  - default behavior: panic if no reschedule in 60s
- Starvation
  - One or more processes never getting CPU time even though they are runnable.
  - One or more CPUs stuck on a spinlock without ever acquiring it.
  - default behavior: SIGKILL if no reschedule in 120s

Deadlock

- One or more process permanently stuck is a “deadly embrace”



deadlocks In the Linux kernel

- Multi thread
  - thread A locks resource X
  - thread B lock resource Y
  - thread A attempts to lock resource Y
  - thread B attempts to lock resource X
- Single thread
  - thread A calls procedure M which locks resource X
  - procedure M calls procedure N which attempts to lock resource X
- Interrupt handling
  - thread A locks resource X
  - interrupt occurs
  - ISR attempts to lock resource X
- livelocks versus deadlocks
  - deadlock: stuck on a single lock
  - livelock: looping down and up a potentially long call chain.

Multi-thread deadlocks

- Solution:
  - strict lock hierarchy enforced(enum)
    - X < Y < Z
  - kernel locks are ordered in terms of scope
    - X scope is a superset of Y scope
    - Y scope is a superset of Z scope
  - X must be locked before Y which must be locked before Z
    - thread A locks resource X
    - thread B attempts to lock resource X
    - thread A locks resource Y
    - thread A releases locks Y then X
    - thread B is granted resource X
    - thread B locks resource Y
    - thread B releades locks Y then X

Linux lock hierarchy

- Lock types in Linux kernel are ordered from 1 to N
- Low numbered/ordered locks are coarser than high numbered locks and must be acquired first

  example:
  - task_list_lock protect a list of struct_tasks
  - task_lock protects an individual struct_task
  - order of task_list_lock < order of task_lock
    a. lock(task_list_lock);
    b. lock_task(&task-being-locked);

- If you need lock A of order 1 and lock B of order 2 you must:
  – lock(A) before lock(B)

    or
  – lock(B) then trylock(A)

- You must unlock in reverse locking order
  – lock(A) lock(B) … unlock(B) unlock(A)
  – lock(B) trylock(A) … unlock(B) unlock(A)

- Well documented in ./GIT/linux/linux/Documentation/locking

Single threaded deadlocks

Thread A calls procedure M which locks resource X

procedure M calls procedure N which attempts to lock resource X

- Solution:
  - recursive locks:
    - Thread A is granted the same lock multiple times in a call chain
      - each lock attempt increments a depth count
      - each unlock decrements a depth count
    - Thread A calls M which locks recursive lock X
      - lock X is aquired, depth count incremented to 1
    - M calls N which locks recursive lock X
      - depth count is incremented to 2
    - N unlocks X
      - depth count is decremented to 1
    - N returns to M
      - depth count is decremented to 0, X is unlocked

Interrupt deadlocks

Interrupt handling:

thread A locks resource X
interrupt occurs
ISR attempts to lock resource X

- Solution:
  - spinlock_IRQ()/spinunlock_IRQ()
    - disable interrupts before acquiring lock X
    - unlock X then enable interrupts
    - thread A does spinlock_IRQ(X)
      - » disable interrupts then locks X
        - interrupts are blocked
    - thread A runs than does spinunlock_IRQ(X)
      - » unlock X then enables interrupts
    - interrupt occurs
      - » ISR locks X, runs then unlock X