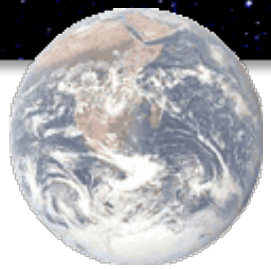
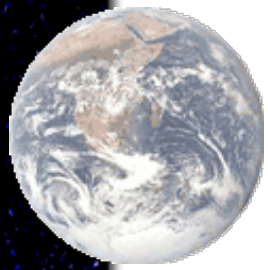
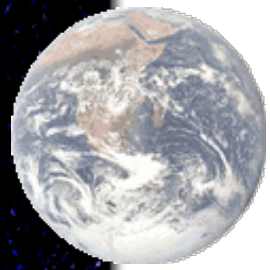


Concurrency: Deadlock



Deadlock

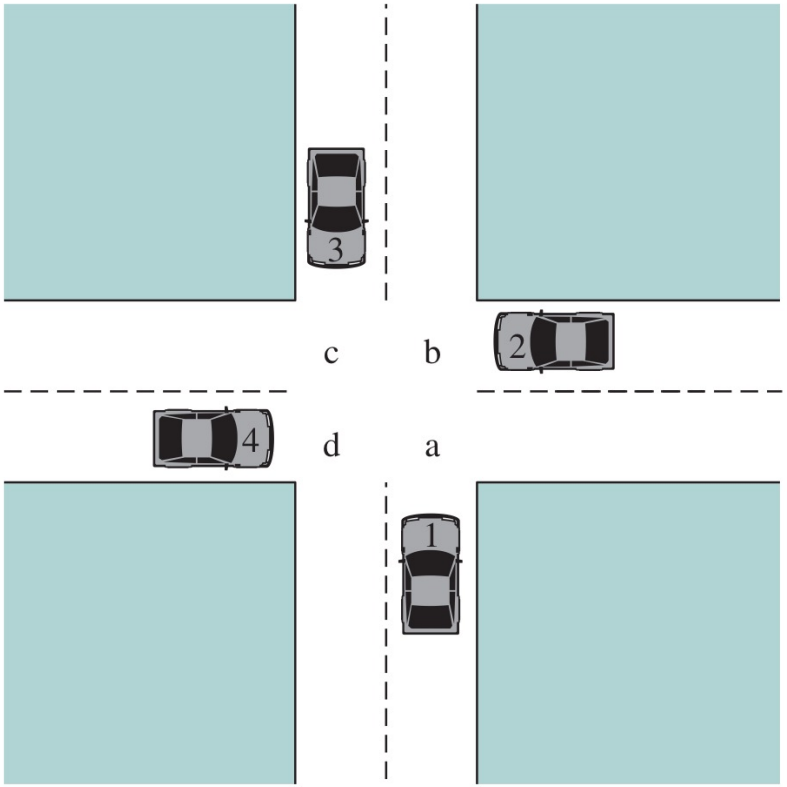
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.
- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- Involve conflicting needs for resources by two or more processes
- No efficient solution



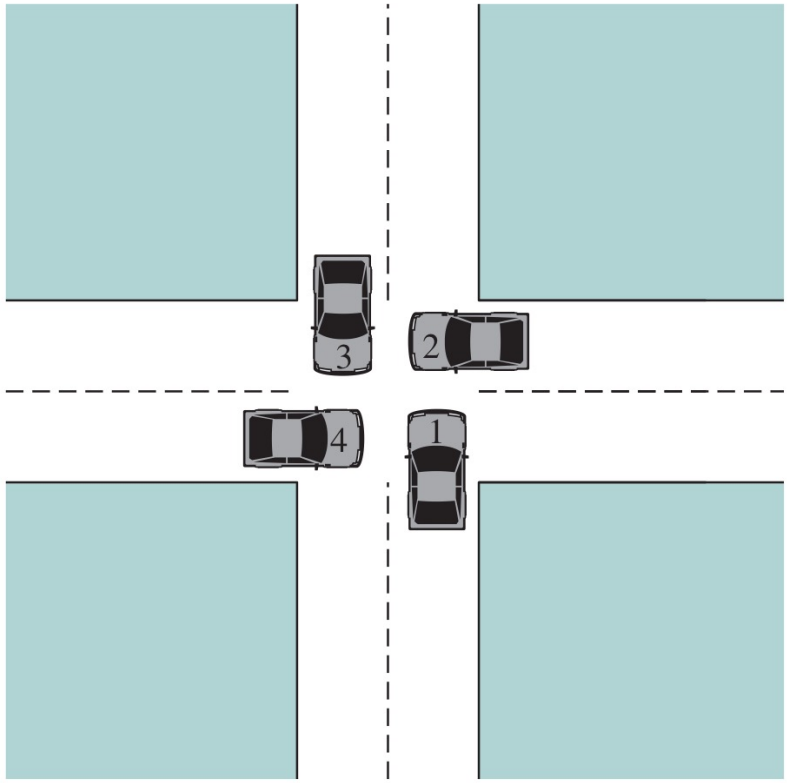
Deadlock and Starvation

- Deadlock is closely related to starvation.
  - Processes wait forever for each other to wake up and/or release resources.
- The difference between deadlock and starvation is subtle.
  - With starvation, there always exists a schedule that feeds the starving party.
    - The situation may resolve itself...if you're lucky.
  - Once deadlock occurs, it cannot be resolved by any possible future schedule.
    - ...though there may exist schedules that avoid deadlock.

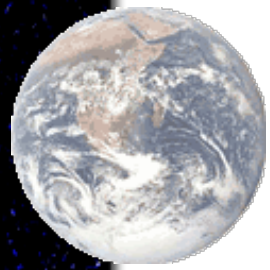
Illustration of Deadlock



(a) Deadlock possible



(b) Deadlock



System Model

A process/thread may utilize a resource in only the following sequence:

1. Request

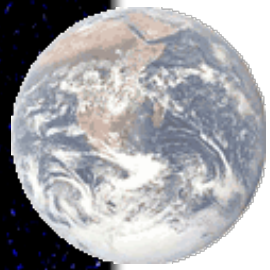
   The procee/thread requests the resource. If the resource can not be granted immediately, then the requesting process/thread must wait until it can acquire the resource.

2. Use

   The process/thread can operate on the resource.

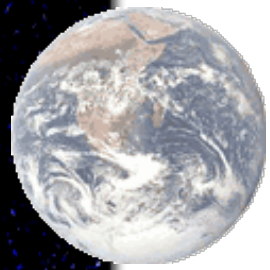
3. Release

   The process/thread releases the resource.



Reusable Resources

- Used by only one process at a time and not depleted by that use.
- Processes obtain resources that they later release for reuse by other processes.
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores.
- Deadlock occurs if each process holds one resource and requests the other.



Examples of Deadlock

- Example 1: P and Q competing for reusable resources D and T.

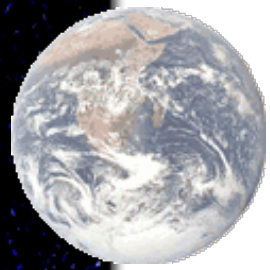
Step	Process P Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Step	Process Q Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

- Example 2: Space is available for allocation of 200KB, and the following sequence of events occur.

P1	P2
…	…
Request 80 Kbytes;	Request 70 Kbytes;
…	…
Request 60 Kbytes;	Request 80 Kbytes;

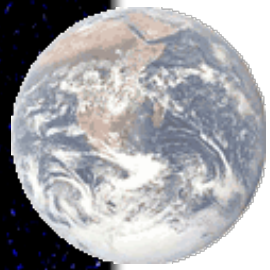
Deadlock occurs if both processes progress to their second request



Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers.
- Deadlock may occur if a Receive message is blocking.
- May take a rare combination of events to cause deadlock.

P1	P2
…	…
Receive (P2);	Receive (P1);
…	…
Send (P2, M1);	Send (P1, M2);



Resource Acquisition (1)

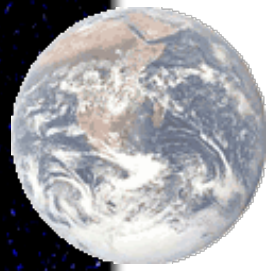
Using a semaphore to protect resources:

```
typedef int semaohore;
semaphore resource_1;

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

One resources

```
typedef int semaohore;
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);
  }
```

Two resources



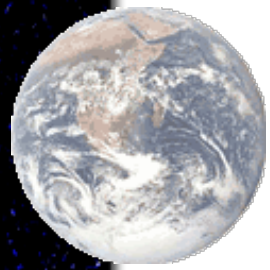
Resource Acquisition (2)

Deadlock-free code

```
typedef int semaohore;
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);
  }
```

Code with a potential deadlock

```
typedef int semaohore;
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);
  }
```



Conditions for Deadlock

- Mutual exclusion

  Only one process may use a resource at a time.

- Hold-and-wait

  A process may hold allocated resources while awaiting assignment of others.

- No preemption

  No resource can be forcibly removed from a process holding it.

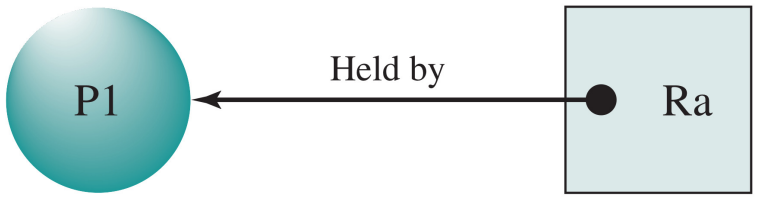
- Circular wait

  A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

Resource Allocation Graphs

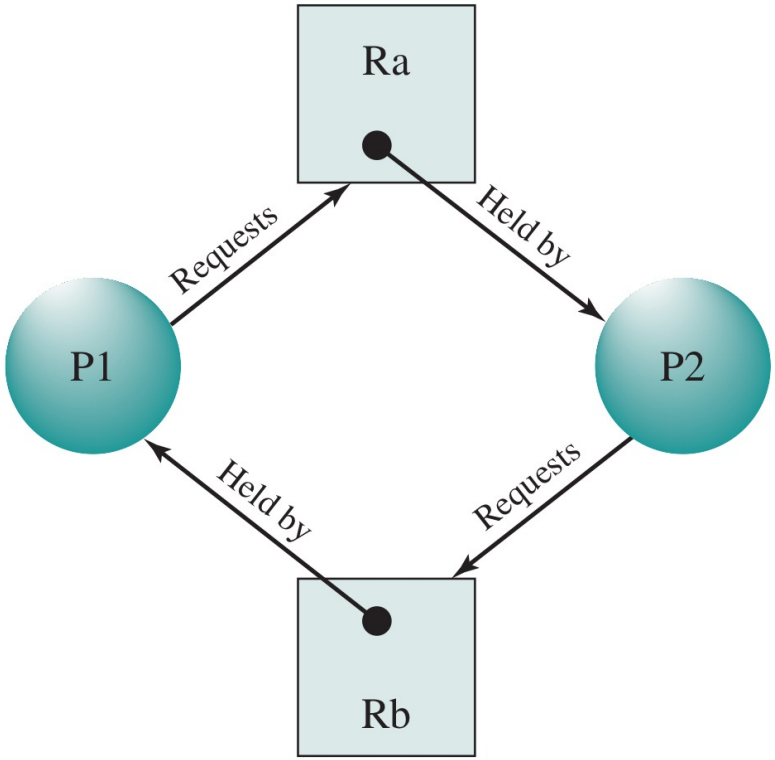
Directed graph that depicts a state of the system of resources and processes

(a) Resource is requested

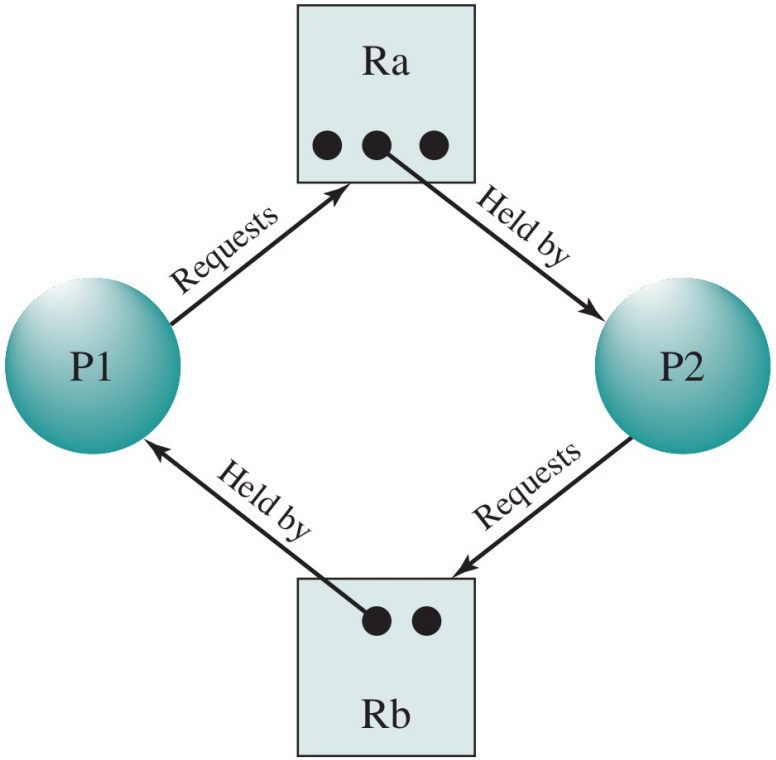


(b) Resource is held

Resource Allocation Graphs

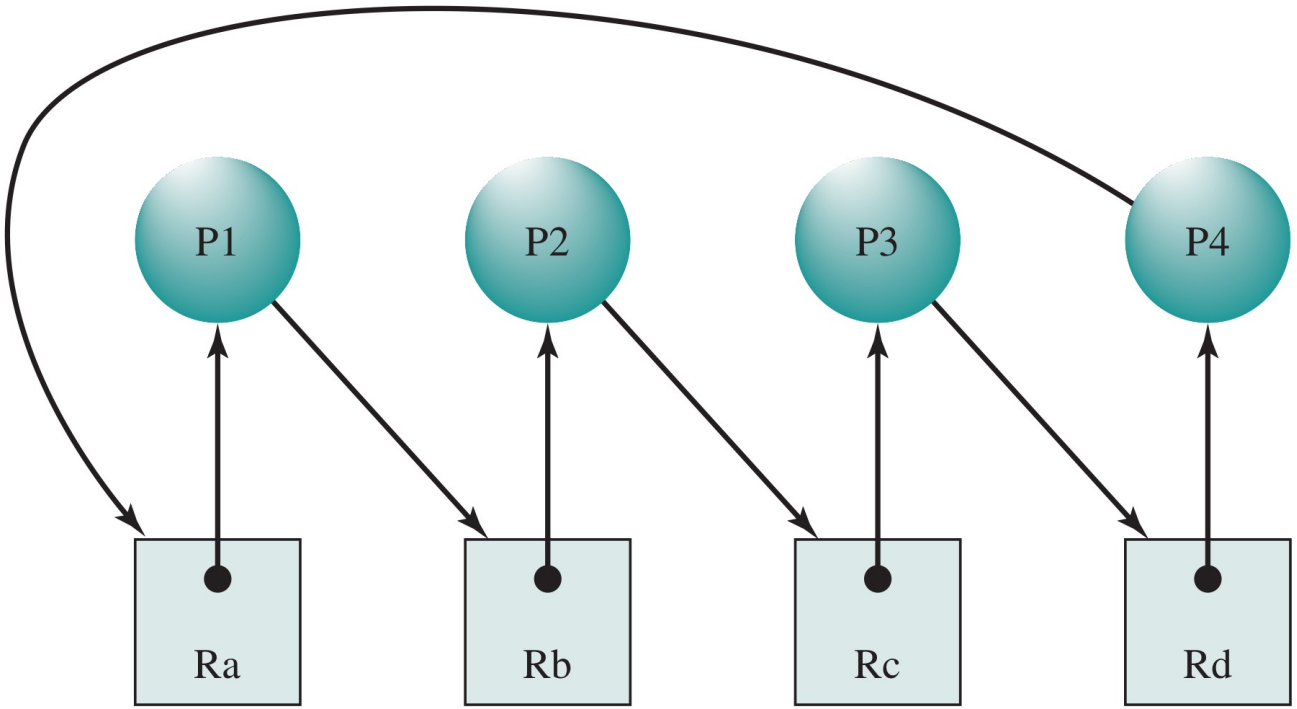


(c) Circular wait



(d) No deadlock

Resource Allocation Graphs



Resource Allocation Graphs (1)

How deadlock occurs.

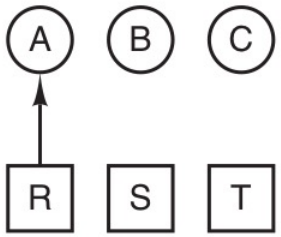
A
Request R
Request S
Release R
Release S
(a)

B
Request S
Request T
Release S
Release T
(b)

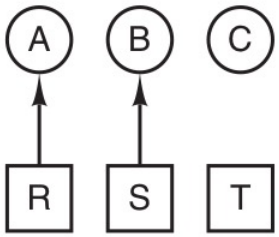
C
Request T
Request R
Release T
Release R
(c)

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

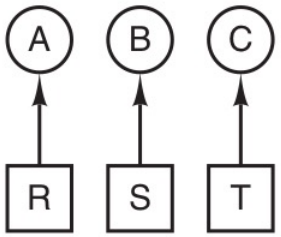
(d)



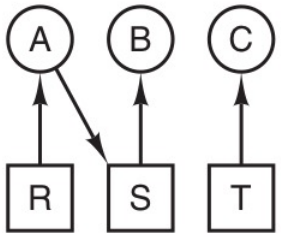
(e)



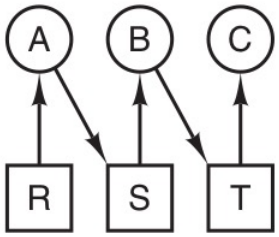
(f)



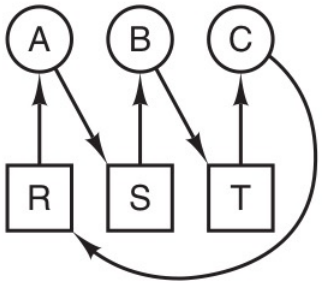
(g)



(h)



(i)



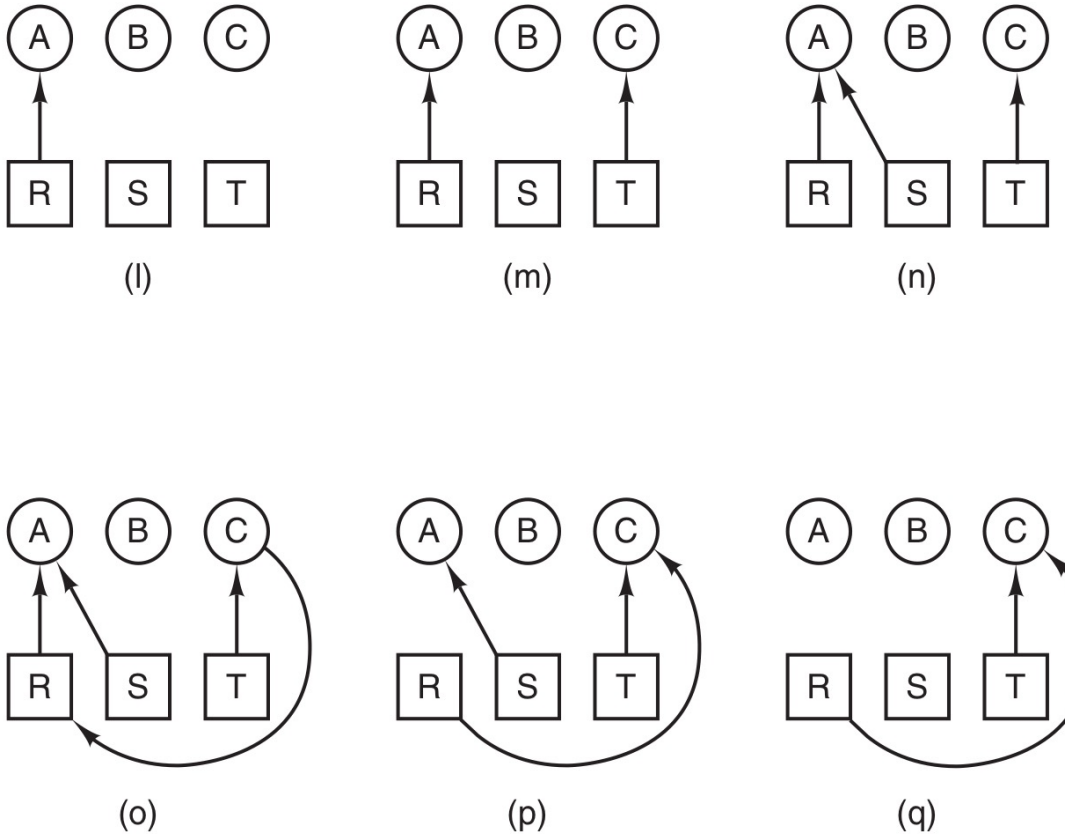
(j)

Resource Allocation Graphs (2)

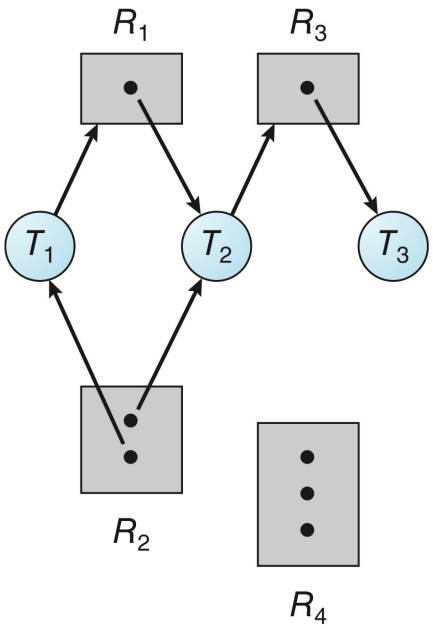
How deadlock can be avoided.

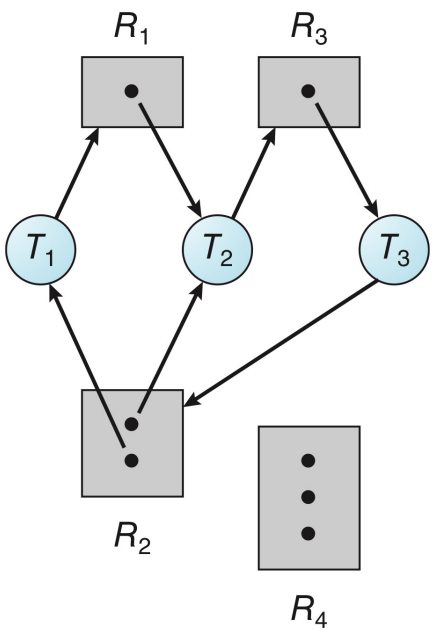
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

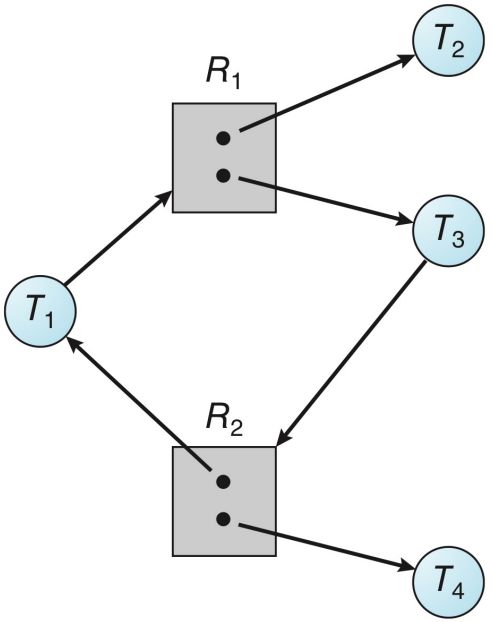
(k)



Resource Allocation Graphs







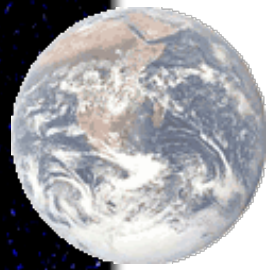
- Graph contains no cycles:

  no deadlock. (i.e., cycle is always a necessary condition for deadlock)

- If graph contains a cycle:

  – if one instance per resource type, then deadlock.

  – if several instances per resource type, then possibility of deadlock.



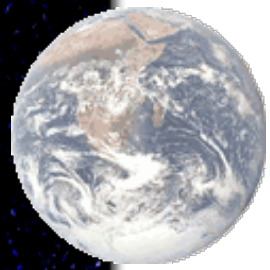
Possibility vs. Existence of Deadlock

Possibility of Deadlock:

- Mutual Exclusion
- No preemption
- Hold and wait

Existence of Deadlock:

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait



Handling of Deadlocks

- Deadlock prevention: deadlock is not possible

  Structurally restrict the way in which processes request resources

- Deadlock avoidance: deadlock is possible, but OS uses advance information to avoid it

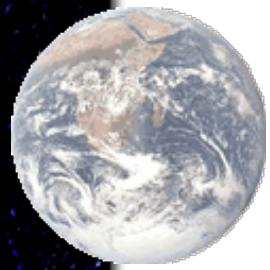
  Require processes to give advance information about the (max) resources they will require; then schedule processes in a way that avoids deadlock.

- Deadlock detection and recovery

  Let deadlocks occur, detect them, and take action.

- Ostrich algorithm

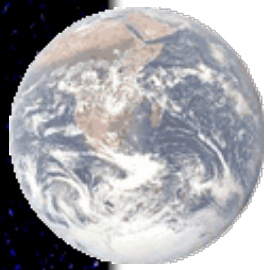
  Ignore the problem and pretend that deadlocks never occur in the system (can be a "solution" sometimes?!...)



Deadlock Prevention

Restrain the ways requests can be made; attack at least one of the four conditions so that deadlocks are impossible to happen:

1. Mutual Exclusion – cannot do much here ...
   - No resource ever assigned exclusively to a single process – spooling.
   - Avoid assigning a resource unless necessary and try to make sure as few processes as possible may actually claim the resource.
2. Hold and Wait
   - Must guarantee that when a process requests a resource, it does not hold any other resources.
   - Require process to request and be allocated all its resources at once.
   - Require a process requesting a resource to release all the resources it currently holds.
   - Low resource utilization; starvation possible



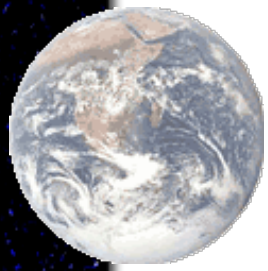
Deadlock Prevention

3. No preemption

   If a process holding some resources requests another resources that cannot be immediately allocated, it releases the held resources and has to request them again (risk for starvation)

4. Circular Wait

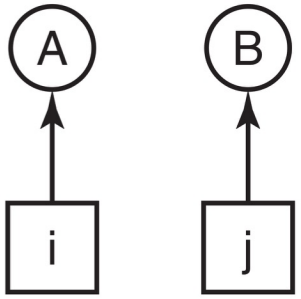
   Define a linear ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

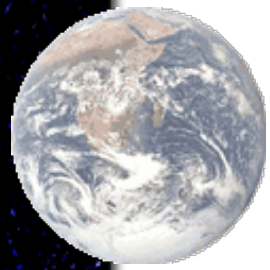


Circular Wait: Resource Ordering

- Global numbering of all the resources.
- The processes can request resources whenever they want to, but all requests must be made in numerical order.

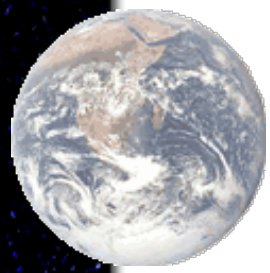
1. Imagesetter
2. Printer
3. Plotter
4. Tape drive
5. Blu-ray drive





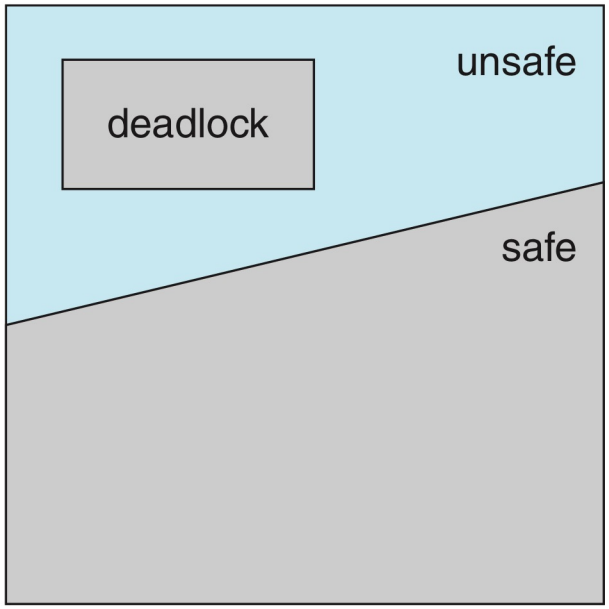
Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Requires knowledge of future process request
- Avoidance = ensure that system will not enter an unsafe state.
- Idea: If satisfying a request will result in an unsafe state, the requesting process is suspended until enough resources are free-ed by processes that will terminate in the meanwhile.



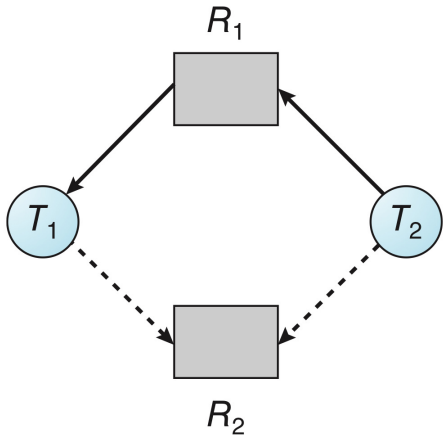
Two Approaches to Deadlock Avoidance

1. Do not start a process if its demands might lead to deadlock.
2. Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

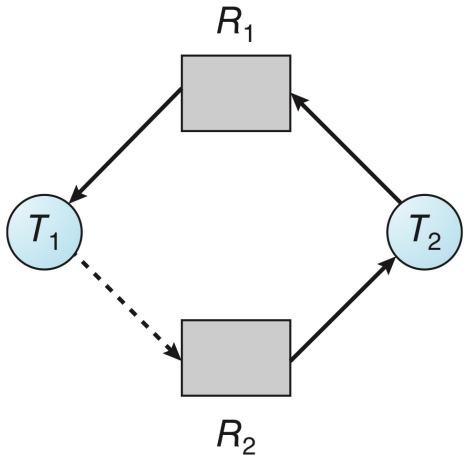


Resource Allocation Denial

- Referred to as the banker's algorithm.
- State of the system is the current allocation of resources to process.
- Safe state is where there is at least one sequence that does not result in deadlock.
- Unsafe state is a state that is not safe.

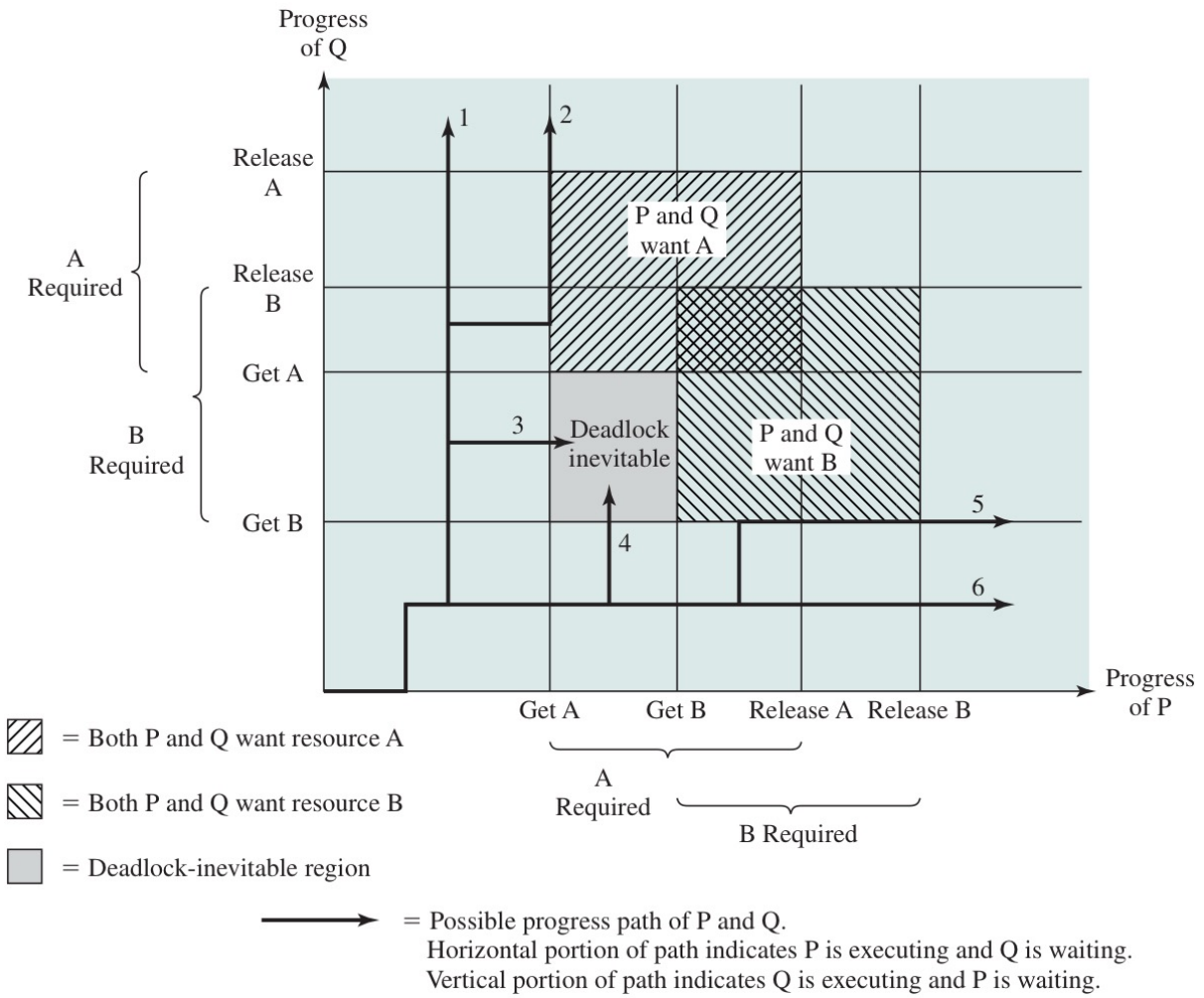


Deadlock avoidance

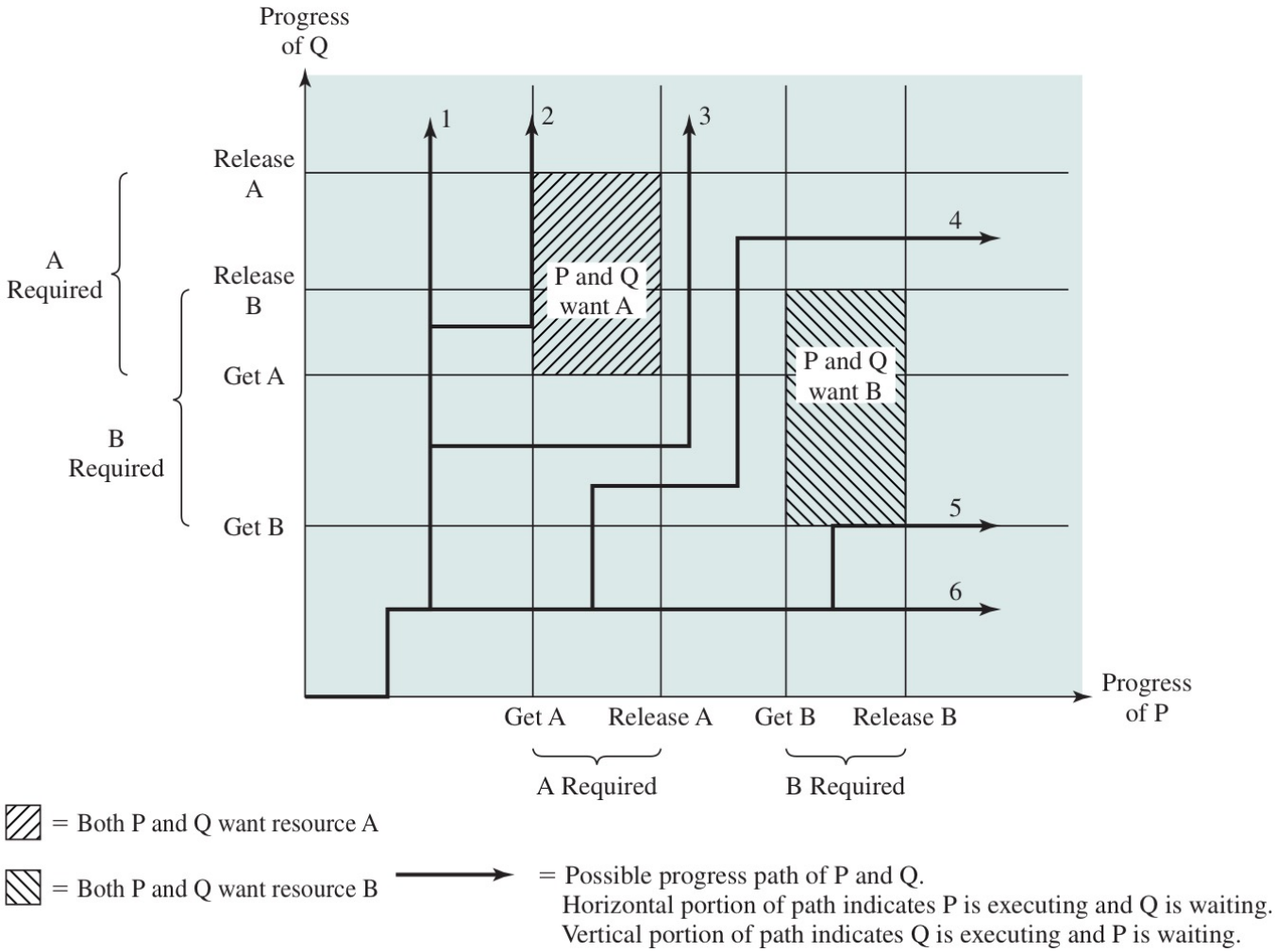


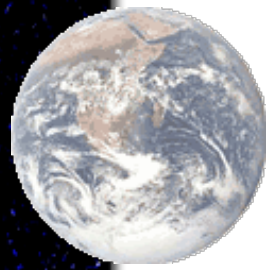
Unsafe state

Resource Trajectories: Deadlock



Resource Trajectories: No Deadlock

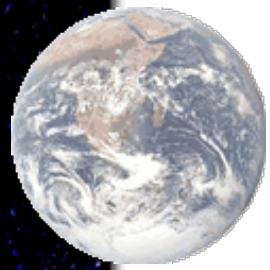




Safety Check

- Safe state = there exists a safe sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of terminating all processes:

  for each P_i, the requests that it can still make can be granted by currently available resources + those held by $P_1, P_2, \ldots, P_{i-1}$
- The system can schedule the processes as follows:
  - if P_i's resource needs are not immediately available, then it can:
    - wait until all $P_1, P_2, \ldots, P_{i-1}$ have finished
    - obtain needed resources, execute, release resources, terminate.
  - then the next process can obtain its needed resources, and so on.



Deadlock Avoidance Data Structures

Resource $= \mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available $= \mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim $= \mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation $= \mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

Safe and Unsafe State (1)

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

Safe and Unsafe State (1)

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

(d) P3 runs to completion

Safe and Unsafe State (2)

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

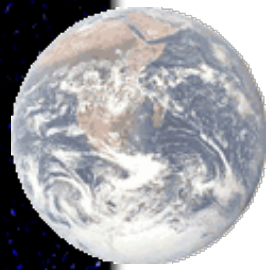
R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

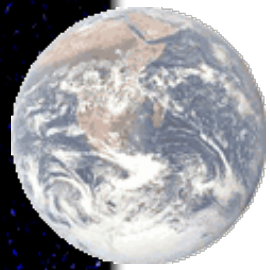


Deadlock Avoidance Logic (1)

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

```
if (alloc [i,*] + request [*] > claim [i,*])
      <error>;                              /* total request > claim*/
else if (request [*] > available [*])
      <suspend process>;
else  {                                     /* simulate alloc */
      <define newstate by:
      alloc [i,*] = alloc [i,*] + request [*];
      available [*] = available [*] - request [*]>;
}
if (safe (newstate))
      <carry out allocation>;
else {
      <restore original state>;
      <suspend process>;
}
```

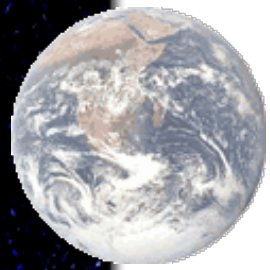
Resource allocation algorithm



Deadlock Avoidance Logic (2)

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
           claim [k,*] - alloc [k,*]<= currentavail;
        if (found) {                  /* simulate execution of Pk */
           currentavail = currentavail + alloc [k,*];
           rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

Banker's algorithm



Deadlock Avoidance More Examples (1)

	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)

Demonstration that the state in (a) is safe

	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)

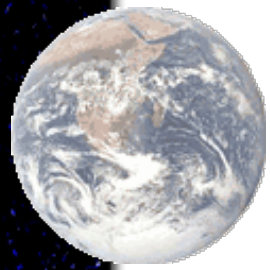
	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Demonstration that the state in (b) is not safe



Deadlock Avoidance More Examples (2)

	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

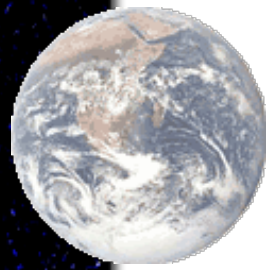
(b)

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

Free: 1

(c)

Three resources allocation states: (a) Safe. (b) Safe. (c) Unsafe.



Deadlock Avoidance More Examples (3)

Consider if the state below is safe or unsafe.

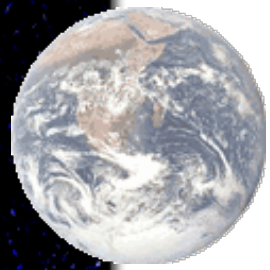
Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	Blu-rays
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

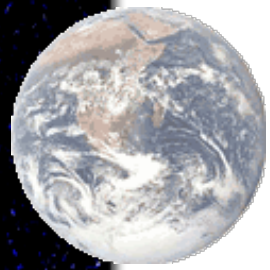
Resources still assigned

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



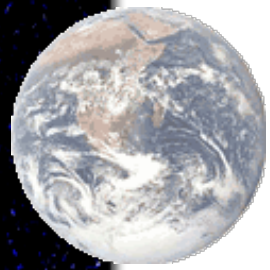
Deadlock Avoidance

- Maximum resource requirement must be stated in advance.
- Processes under consideration must be independent; no synchronization requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.



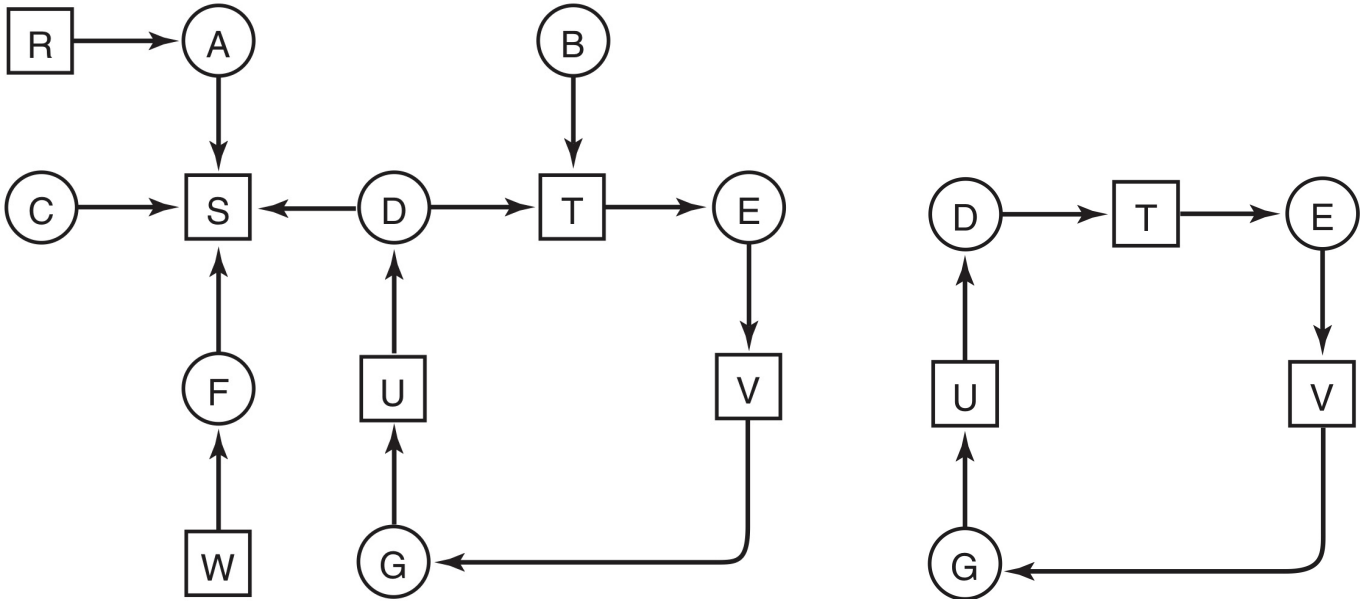
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme



Deadlock Detection - RAG

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



Deadlock Detection: Data Structures

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

Current allocation matrix

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

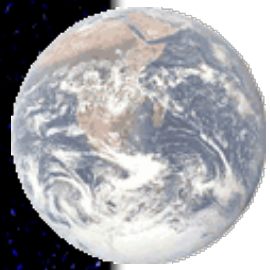
Row n is current allocation to process n

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



Detection Algorithm

	Tape drives	Plotters	Scanners	Blu-rays
E =	4	2	3	1

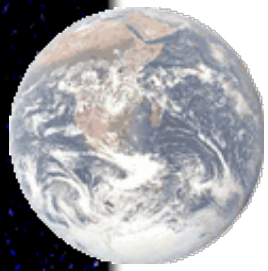
	Tape drives	Plotters	Scanners	Blu-rays
A =	2	1	0	0

Current allocation matrix

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

Request matrix

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



Deadlock Detection

Consider if deadlock is detected in the state below.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

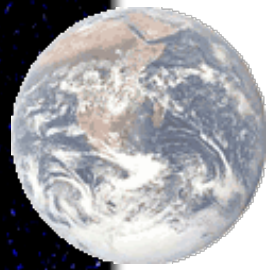
Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

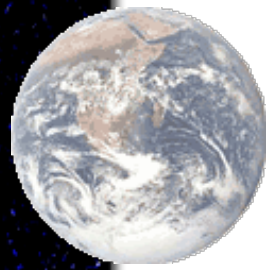
R1	R2	R3	R4	R5
0	0	0	0	1

Available vector



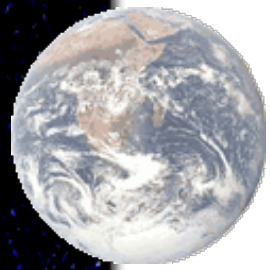
Recovery from Deadlock

- Recovery through preemption
  - Successively preempt resources until deadlock no longer exists.
  - Depends on nature of the resource
- Recovery through rollback
  - Checkpoint a process periodically
  - Back up each deadlocked process to some previously defined checkpoint and restart all process; original deadlock may occur.
- Recovery through killing processes
  - Crudest but simplest way to break a deadlock
  - Abort all deadlocked processes, or successively abort deadlocked processes until deadlock no longer exists
  - Choose process that can be rerun from the beginning



Selection Criteria Deadlocked Processes

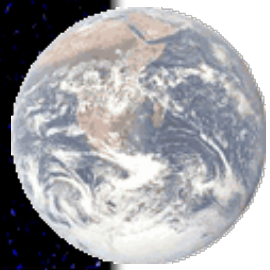
- Least amount of processor time consumed so far
- Least number of lines of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority



Detection Algorithm Usage

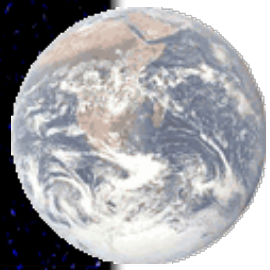
- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
- If algorithm is invoked arbitrarily, there may be many cycles in the resource graph

  → we would not be able to tell which of the many deadlocked processes "caused" the deadlock.



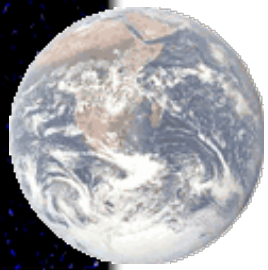
Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
  - Convenience
  - Correctness



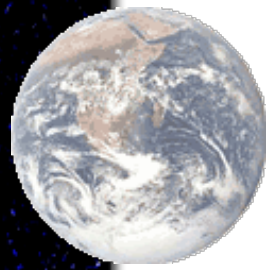
Summary of Deadloak Handling Approaches

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses



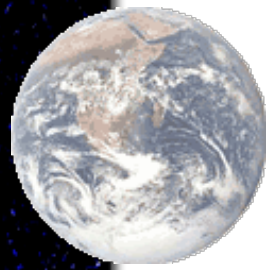
Other Issues

- Two-phase locking
- Communication deadlocks
- Livelock
- Starvation



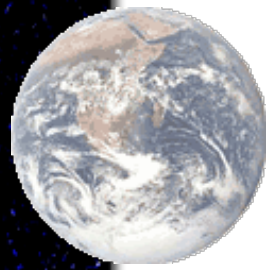
Two-Phase Locking

- 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
- Algorithm works where programmer has arranged things carefully such that the program can be stopped at any point in the first phase and restarted.



Communication Deadlock

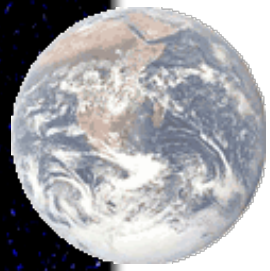
- An example of none-resources deadlock.
- The processes in this type could not complete service if executed independently.
- Mechanisms called timeouts is employed to break deadlocks.
  - Whenever a message is sent to which a reply is expected, a timer is started.
  - If the timer goes off before the reply arrives, the sender of the message assumes that the message has been lost and sends it again.



Livelock

- Almost every table in the OS represents finite resource.
- Most OS, including UNIX and Windows, ignore the problem
  - assumming that most users would prefer an occasional livelock (or even deadlock) to a rule restricting all users to one process, one open file, and one of everything.
- Unpleasant trade-off between convenience and correctness.

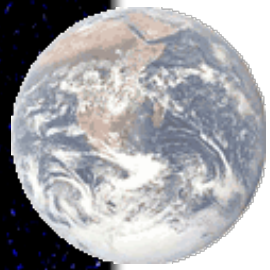
Terms	Definition
Livelock	A situation in which two or more processes continuously change their states in response to changes in other processes without doing any useful work.



Livelock

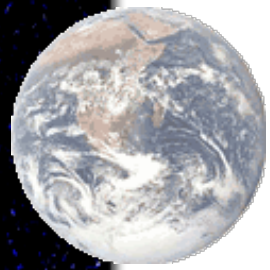
```
void process_A(void) {
  acquire_lock(&resource_1);
  while (try_lock(&resource_2)==FAIL) {
    release_lock(&resource_1);
    wait_fixed_time();
    acquire_lock(&resource_1);
    }
  use_both_resources();
  release_lock(&resource_2);
  release_lock(&resource_1);
  }
```

```
void process_B(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

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



References

1. Modern Operating Systems, 4th edition, Andrew S. Tanenbaum, Herbert Bos
2. Operating Systems, 3rd edition, H.M.Deitel, Pearson Education Limited: Longman.
3. Operating System Concepts, 10th edition, Abraham Silberschatz, Yale University, Peter Baer Galvin, Pluribus Networks, Greg Gagne, Westminster College, 10th edition. Wiley.
4. Operating Systems: Internals and Design Principles, 7th edition, William Stallings.