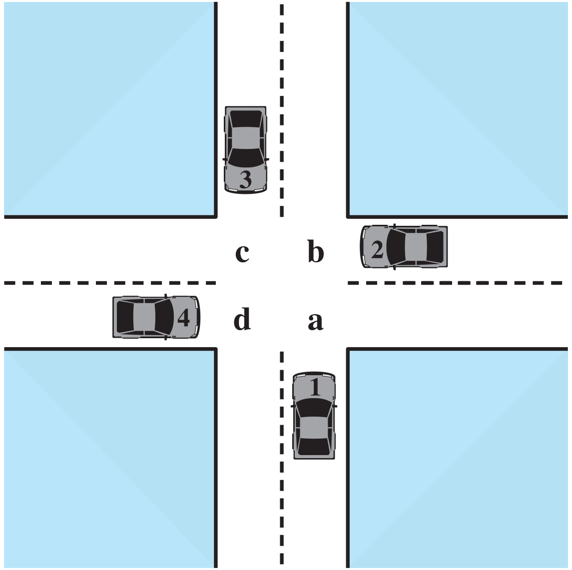
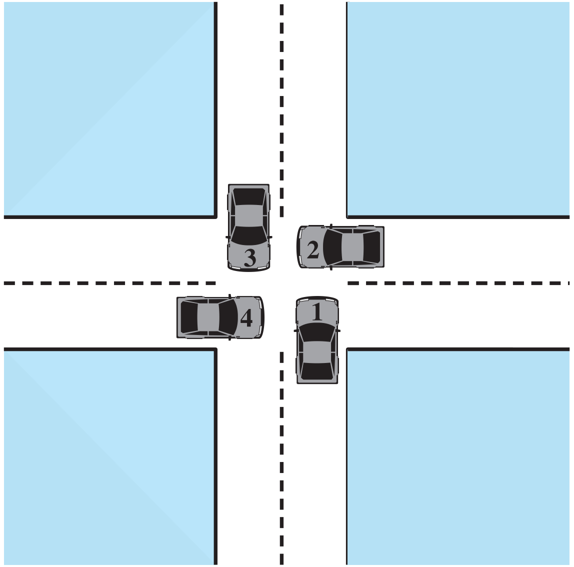
OPERATING SYSTEMS

CONCURRENCY: DEADLOCK AND STARVATION

Deadlock: an illustration



(a) Deadlock possible



(b) Deadlock

Deadlock

- To understand **Deadlock** we need to consider each **process** as an entity that either **requests** or **holds resources**
- Given a set of processes that **either compete** for system resources **or communicate** with each other, the set is **deadlocked** if processes are **permanently blocked**
  - **each process** is blocked **awaiting** an event that can only be triggered by **another blocked process**

No efficient solution in the general case

Resource Categories

Reusable

- Can be safely used by only one process at a time and is not depleted by that use
- Example: processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- One that can be created (produced) and destroyed (consumed)
- Example: interrupts, signals, messages, and information In I/O buffers

Example of Processes competing for Reusable Resources

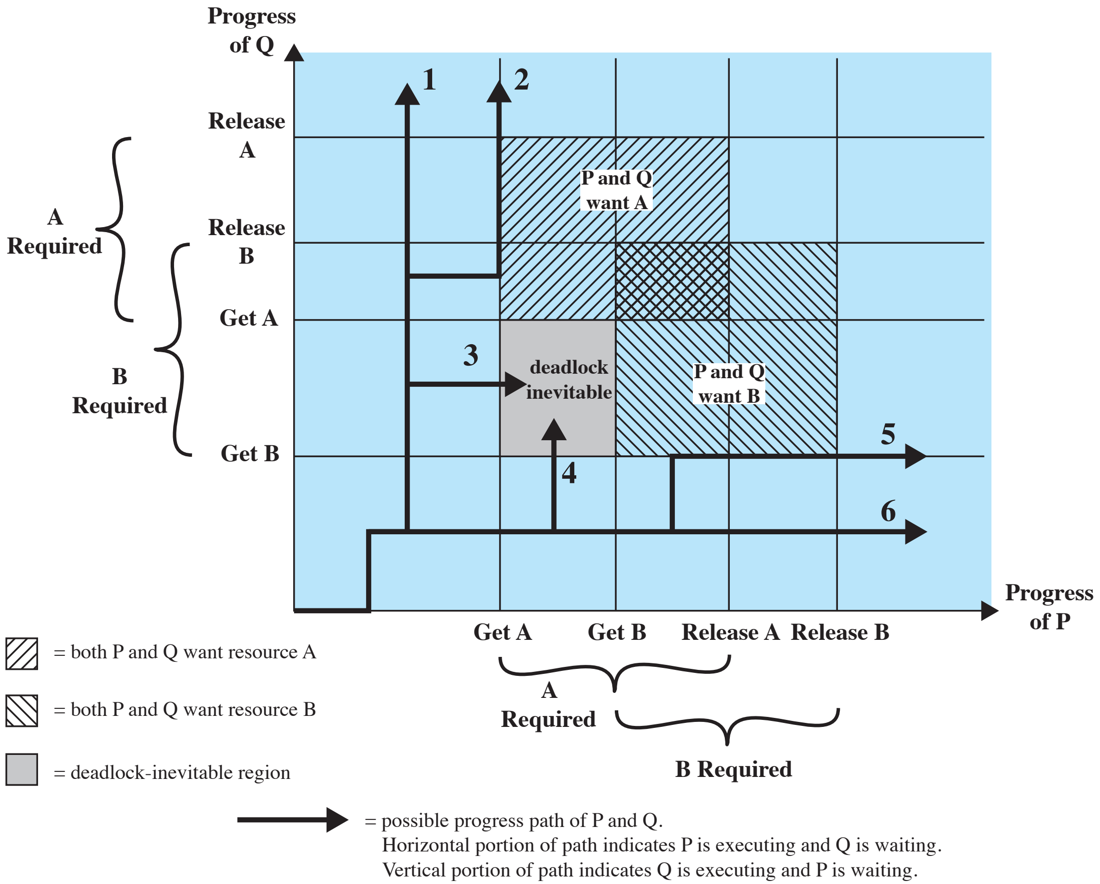
Process P

Step	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)

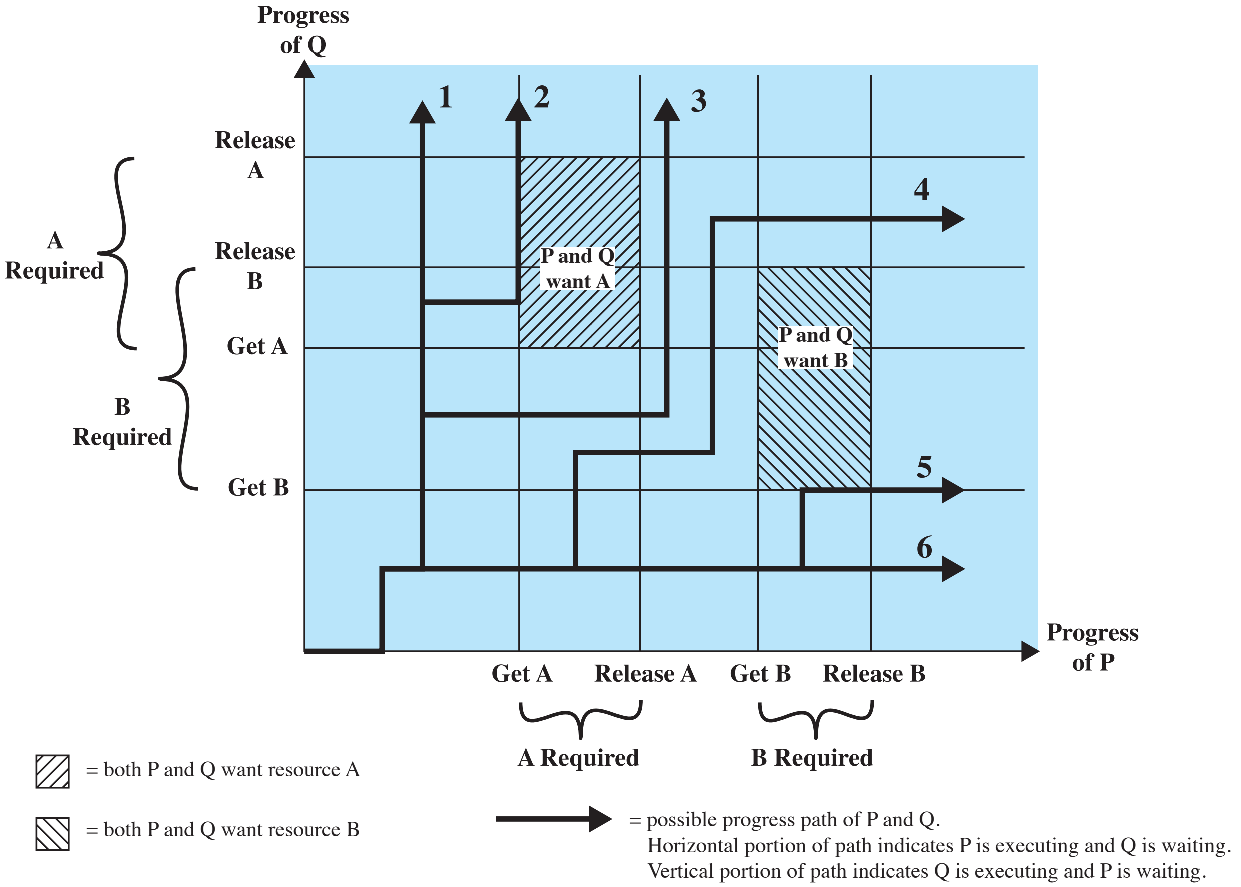
Process Q

Step	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 of Deadlock



Example of No Deadlock



Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

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

- Deadlock occurs if the Receive is blocking

Conditions for Deadlock

necessary conditions: Mutual Exclusion, Hold-and-Wait, No Pre-emption

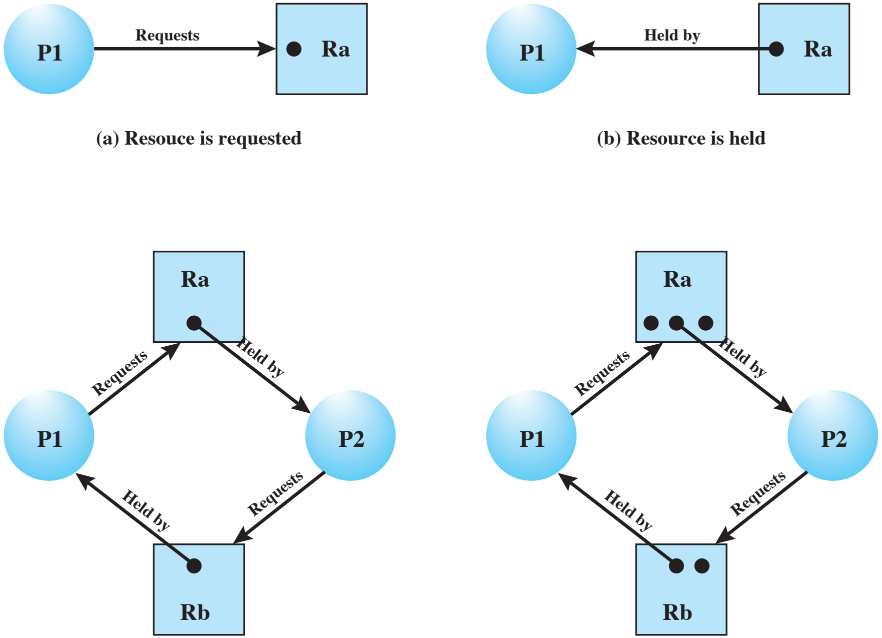
sufficient condition: Circular Wait

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• Only one process may use a resource at a time • No process may access a resource until that has been allocated to another process	• A process may hold allocated resources while awaiting assignment of other resources	• No resource can be forcibly removed from a process holding it	• 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

Vertices are either processes or resources.

Arcs represent the processes requesting or holding resources.



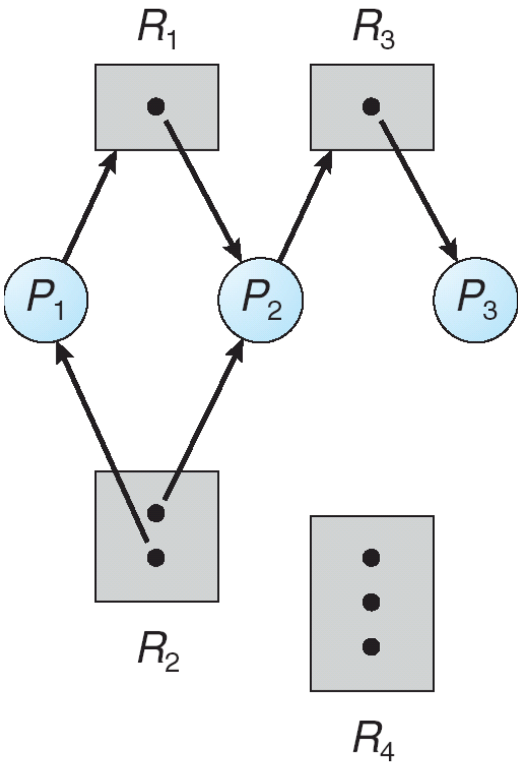
(a) Resouce is requested

(b) Resource is held

(c) Circular wait

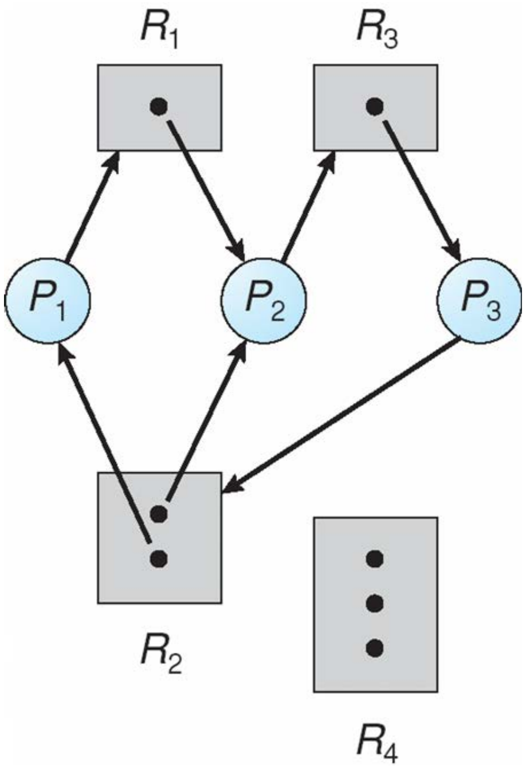
(d) No deadlock

Example of a Resource Allocation Graph



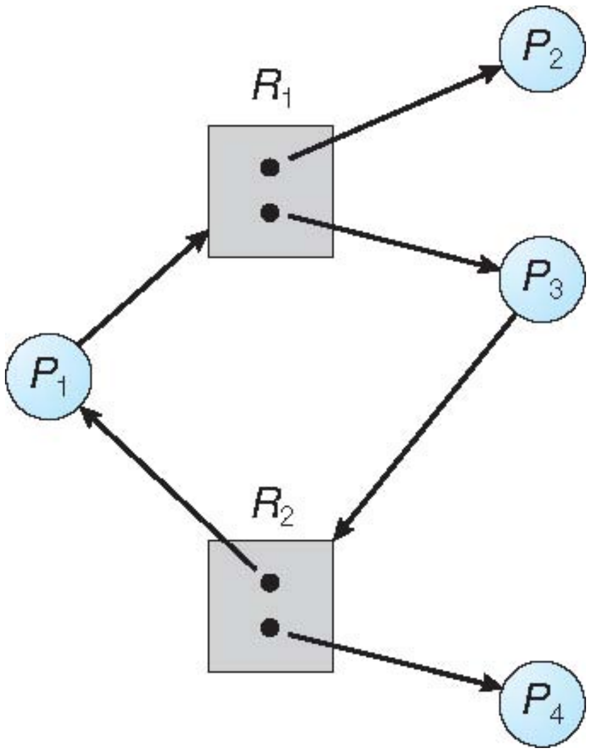
No cycles, no deadlocks

Resource allocation graph with a deadlock



One cycle → deadlock

Graph with a Cycle but No Deadlock



One cycle but no deadlock
at least one of the processes
holding the resources
is not part of the cycle

Deadlock Approaches

There is no single effective strategy that can deal with all types of deadlock

Three approaches are common

Deadlock Prevention

Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening

Deadlock Avoidance

Do not grant a resource request if this allocation might lead to deadlock

Deadlock Detection

Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

Deadlock Prevention

Deadlock Prevention
Restrain the ways request can be made

Mutual Exclusion

- **not required for sharable** resources (e.g., read-only files);
- **must hold for non-sharable** resources

Hold and Wait

must guarantee that whenever a process requests a resource, it does not hold any other resources

- Require process to request and be allocated **all** its **resources before** it begins execution, **or** allow process to request **resources only when** the process has **none allocated** to it.
- Low resource utilization; starvation possible

Deadlock Prevention
Restrain the ways request can be made

No Preemption

- If a process that is holding some resources **requests** another **resource** that **cannot be** immediately **allocated** to it, then **all resources** currently being held are **released**
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular Wait

- impose a **total ordering of all resource types**, and require that **each process requests resources in an increasing order of enumeration**

Deadlock Avoidance

Deadlock Avoidance

**Allows the three necessary conditions
but makes judicious choices
to assure that the deadlock point is never reached**

A decision is made **dynamically** whether the current resource allocation **request will**, if granted, **potentially** lead to a **deadlock**

Requires knowledge of future process requests

Two Approaches to Deadlock Avoidance

Process Initiation Denial

Do not start a process if its demands might lead to deadlock

Resource Allocation Denial

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 reflects the **current allocation** of resources to processes
- **Safe state** is one in which there is **at least one sequence of resource allocations** to processes that **does not result in a deadlock**
- **Unsafe state** is a state that is not safe

Banker's Algorithm

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

(a) global data structures

```
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 >;
}
```

(b) resource allocation algorithm

Banker's Algorithm

```
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);
}
```

(c) test for safety algorithm (banker's algorithm)

Determination of a Safe 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	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

Determination of a Safe 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

Determination of a Safe State

	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

Determination of a Safe State

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
9	3	4

(d) P3 runs to completion

Determination of an Unsafe 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	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

Determination of an Unsafe 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 Advantages

- It is less restrictive than deadlock prevention
- It is not necessary to preempt and rollback processes, as in deadlock detection

Deadlock Avoidance Restrictions

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

Deadlock Detection

Deadlock Strategies

Deadlock prevention strategies are very conservative

Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

Resource requests are granted whenever possible

Deadlock Detection

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

Deadlock Detection Algorithm

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur

Advantages

- It leads to early detection
- The algorithm is relatively simple

Disadvantage

- Frequent checks consume considerable processor time

Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Recovery from Deadlock: Process Termination

- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Integrated Deadlock Strategy

- Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use **different strategies in different situations**
  - **Group resources** into a number of different resource classes
  - Use the **linear ordering strategy** defined previously for the prevention of circular wait to prevent deadlocks between resource classes
  - Within a **resource class**, use the **algorithm** that is most **appropriate** for that class

Classes of resources

- **Swappable space**
  - Blocks of memory on secondary storage for use in swapping processes
- **Process resources**
  - Assignable devices, such as tape drives, and files
- **Main memory**
  - Assignable to processes in pages or segments
- **Internal resources**
  - Such as I/O channels

Class Strategies

- **Swappable space**
  - Prevention of deadlocks by requiring that all of the required resources that may be used be **allocated at one time**, as in the hold-and-wait prevention strategy
    - This strategy is reasonable if the maximum storage requirements are known
- **Process resources**
  - Avoidance will often be effective in this category, because it is reasonable to expect processes to declare ahead of time the resources that they will require in this class
    - Prevention by means of resource ordering within this class is also possible
- **Main memory**
  - Prevention by preemption appears to be the most appropriate strategy for main memory
    - When a process is preempted, it is simply swapped to secondary memory, freeing space to resolve the deadlock
- **Internal resources**
  - Prevention by means **of** resource ordering can be used

Conclusions from A. Tanenabum, Modern Operating Systems

If ever there was a subject that was investigated mercilessly during the early days of operating systems, it was deadlock.

The reason for this is that deadlock is a nice little graph theory problem that one mathematically-inclined graduate student can get his jaws around and chew on for 3 or 4 years,

All kind of algorithms were devised, each one more exotic, and less practical than the previous one.

When an operating system wants to do deadlock detection or prevention, which few of them do, they use one of the methods discussed in this chapter.