Introduction to System Programming

Deadlock

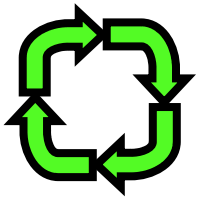
Minal Shah

Outline

- Introduction
- Principles of Deadlock
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection

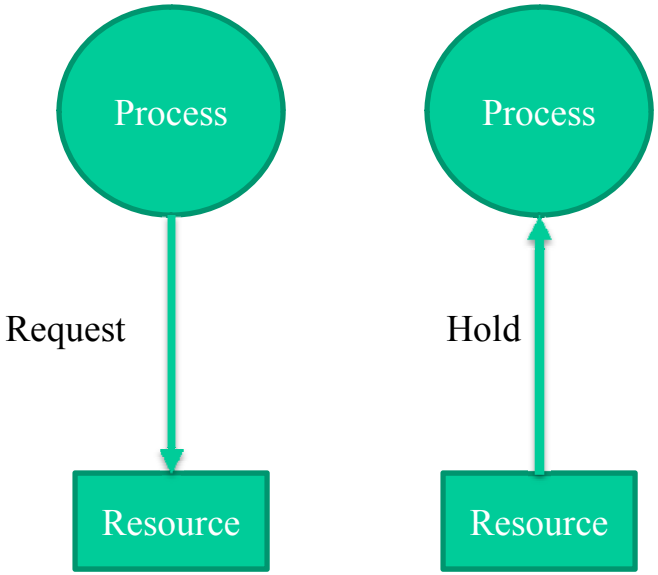
Preemptable and non-preemptable resource

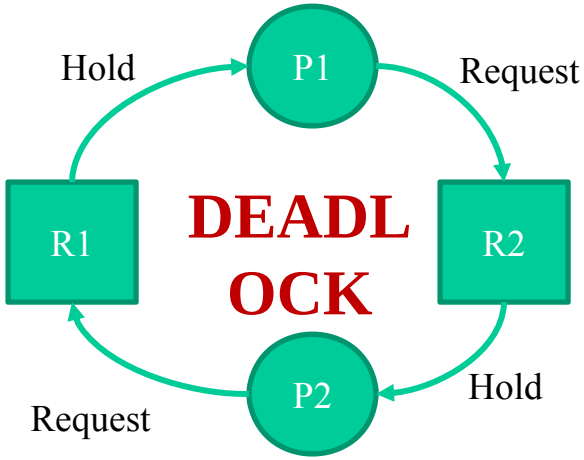
- **Preemptable**:- Preemptive resources are those which **can be taken away from a process without causing any ill effects** to the process.
  - Example:- Memory.
- **Non-preemptable**:- Non-pre-emptive resources are those which **cannot be taken away from the process without causing any ill effects** to the process.
  - Example:- CD-ROM (CD recorder), Printer.



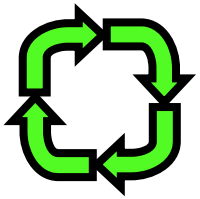
What is Deadlock?

- 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**.
- Deadlocks are a **set of blocked processes** each **holding a resource and waiting to acquire a resource held by another process**.



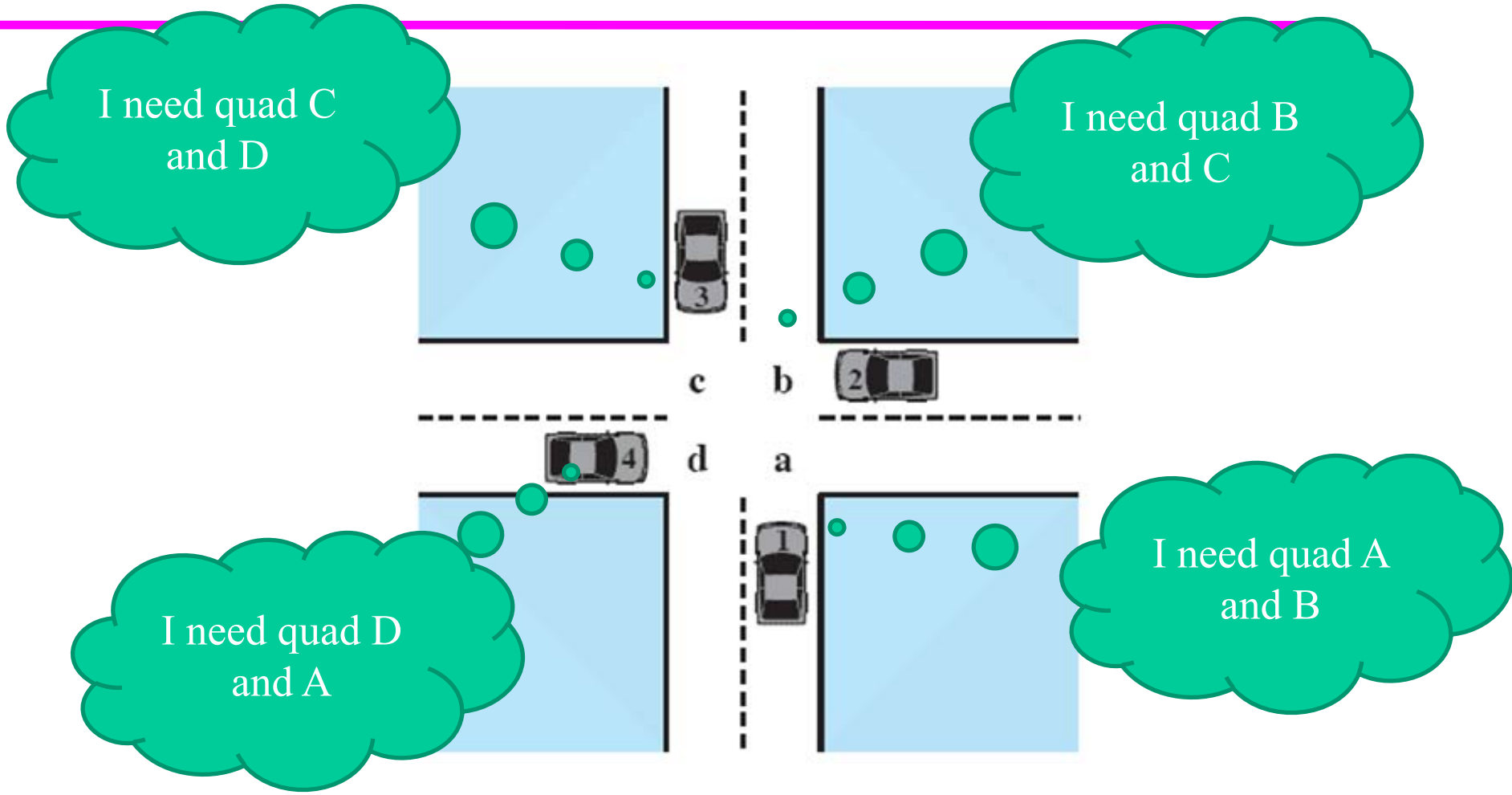


What is a deadlock?

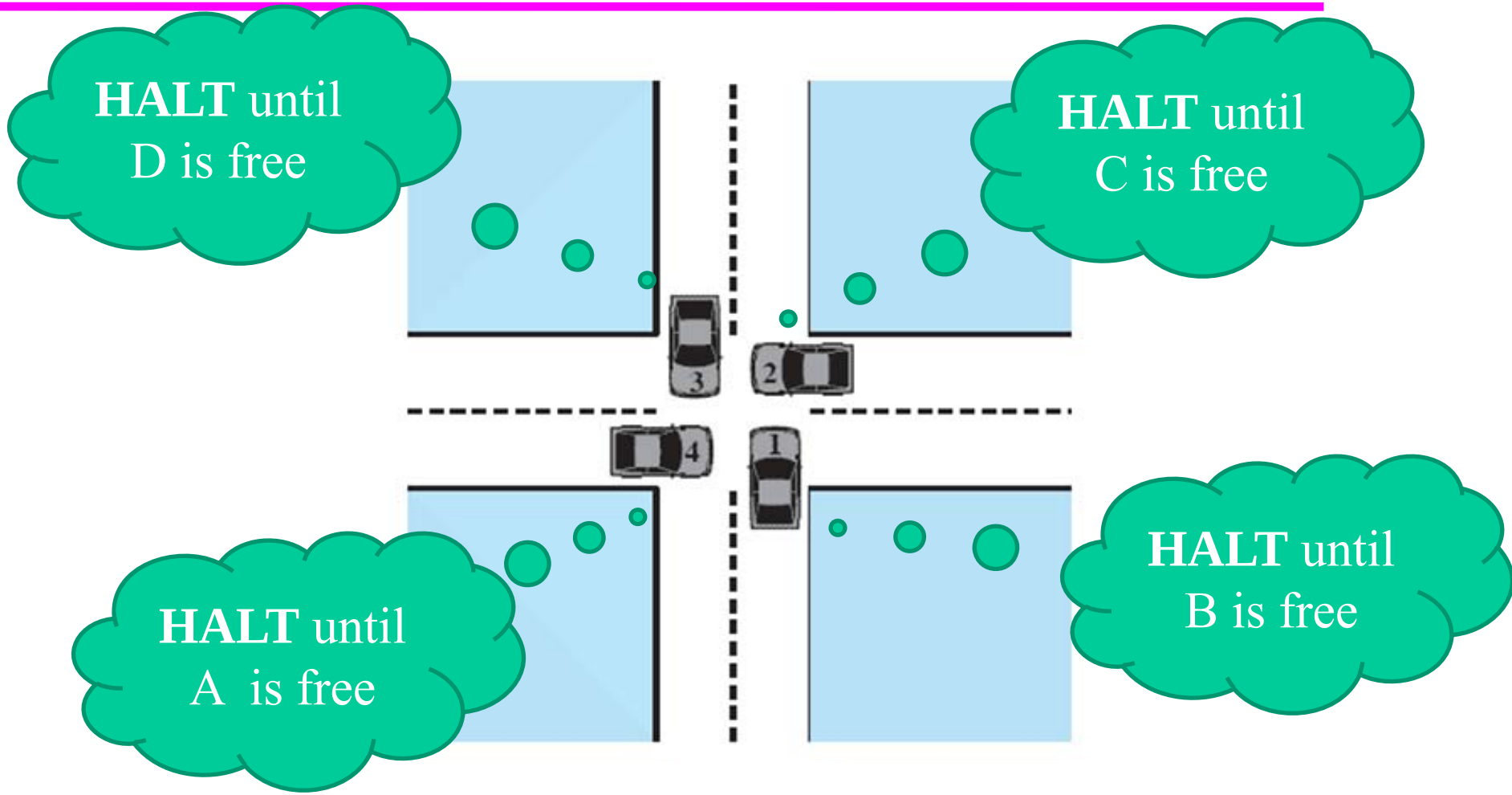


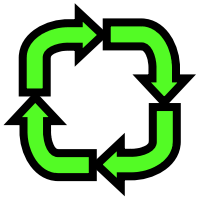
- Formal definition:
  "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause."
- Usually, the event is release of a currently held resource
- In deadlock, none of the processes can
  - Run
  - Release resources
  - Be awakened

Potential Deadlock



Actual Deadlock





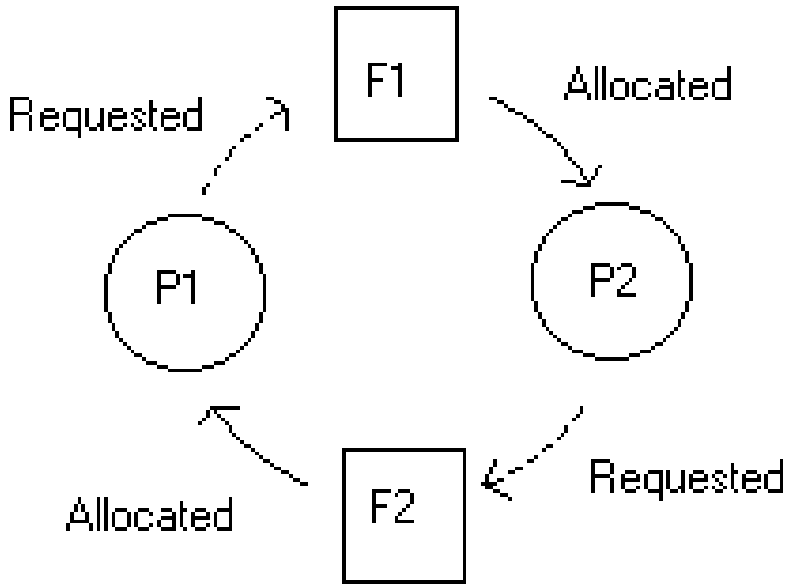
Deadlock: another example

- John borrowed book A from the library.
  - John is holding book A but also needs book B to complete his assignment
- Marry borrowed book B from the library.
  - Marry is holding book B but also needs book A to complete her homework
- This is a deadlock situation. As a result, no one can complete his/her work.

Seven Cases of Deadlocks

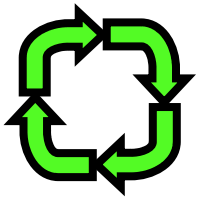
1. Deadlocks on file requests
2. Deadlocks in databases
3. Deadlocks in dedicated device allocation
4. Deadlocks in multiple device allocation
5. Deadlocks in spooling
6. Deadlocks in disk sharing
7. Deadlocks in a network

Case 1 : Deadlocks on File Requests



- If jobs can request and hold files for duration of their execution, deadlock can occur.
- Any other programs that require F1 or F2 are put on hold as long as this situation continues.
- Deadlock remains until a programs is withdrawn or forcibly removed and its file is released.

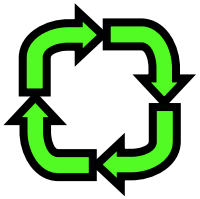
Case 2 : Deadlocks in Databases



1. P1 accesses R1 and locks it.
2. P2 accesses R2 and locks it.
3. P1 requests R2, which is locked by P2.
4. P2 requests R1, which is locked by P1.

- Deadlock can occur if 2 processes access & lock records in database.
- 3 different levels of locking :
  - entire database for duration of request
  - a subsection of the database
  - individual record until process is completed.
- If don't use locks, can lead to a **race condition**.

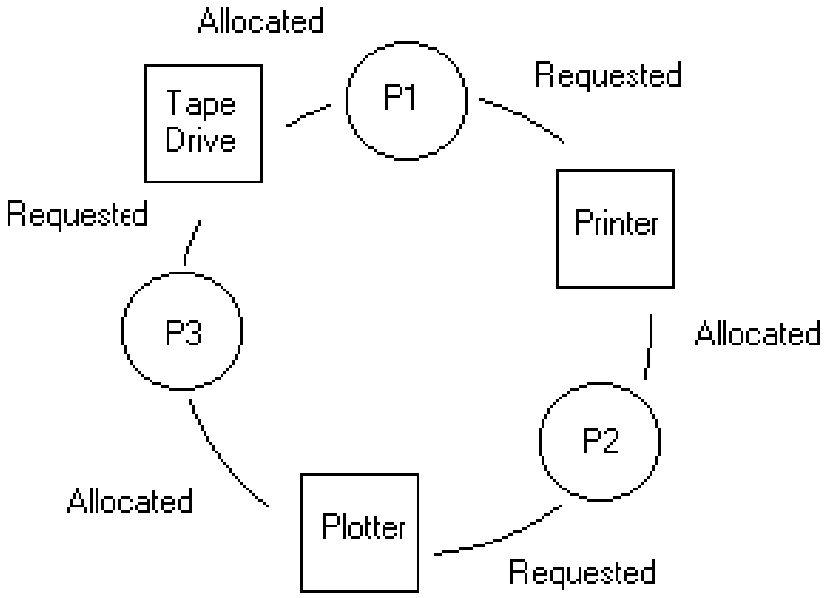
Case 3: Deadlocks in Dedicated Device Allocation



1. P1 requests tape drive 1 and gets it.
2. P2 requests tape drive 2 and gets it.
3. P1 requests tape drive 2 but is blocked.
4. P2 requests tape drive 1 but is blocked.

- Deadlock can occur when there is a limited number of dedicated devices.
- E.g., printers, plotters or tape drives.

Case 4 : Deadlocks in Multiple Device Allocation

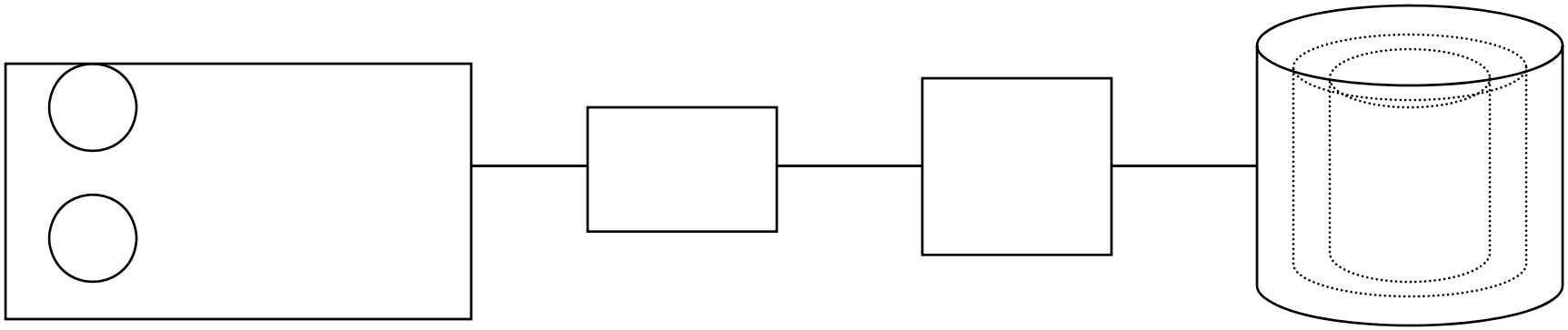


- Deadlocks can happen when several processes request, and hold on to, dedicated devices while other processes act in a similar manner.

Case 5 : Deadlocks in Spooling

- Most systems have transformed dedicated devices such as a printer into a sharable device by installing a high-speed device, a disk, between it and the CPU.
- Disk accepts output from several users and acts as a temporary storage area for all output until printer is ready to accept it (**spooling**).
- If printer needs all of a job's output before it will begin printing, but spooling system fills available disk space with only partially completed output, then a deadlock can occur.

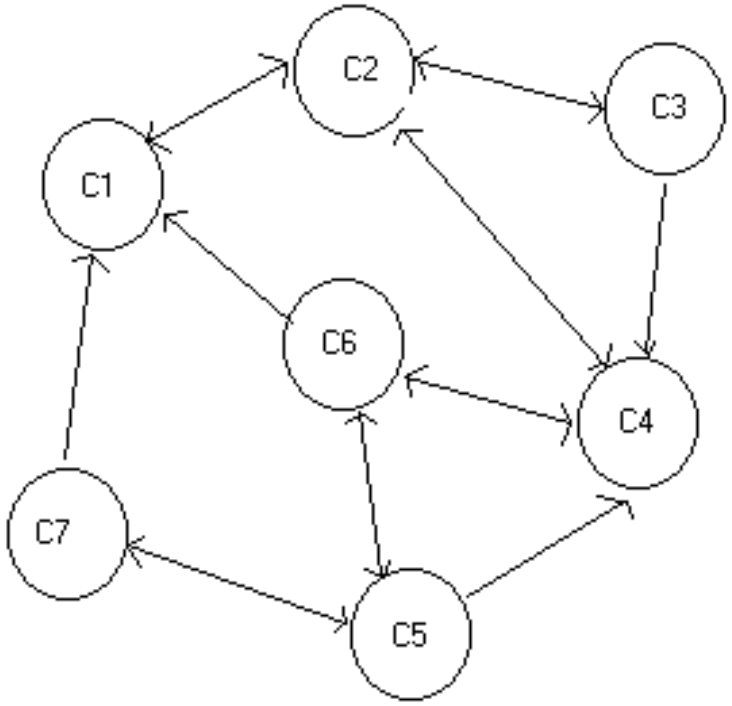
Case 6 : Deadlocks in Disk Sharing

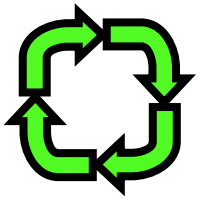


- Disks are designed to be shared, so it's not uncommon for 2 processes access different areas of same disk.
- Without controls to regulate use of disk drive, competing processes could send conflicting commands and deadlock the system.

Case 7: Deadlocks in a Network

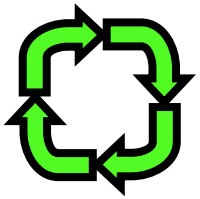
- A network that's congested (or filled large % of its I/O buffer space) can become deadlocked if it doesn't have protocols to control flow of messages through network.





Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources units that they later release for reuse by other processes
- Examples : 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



Reusable Resources

Process P

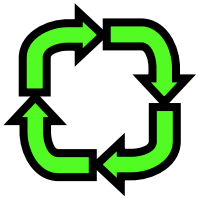
Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

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)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Deadlock occurs if the multiprogramming system executes the to process as p0,p1,q0,q1,p2,q2



Reusable Resources

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

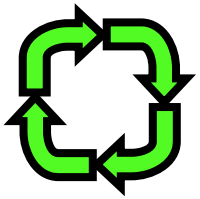
P1

```
. . .
Request 80 Kbytes;
. .
Request 60 Kbytes;
```

P2

```
. . .
Request 70 Kbytes;
. . .
Request 80 Kbytes;
```

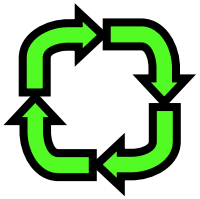
- Deadlock occurs if both processes progress to their second request.



Consumable Resources

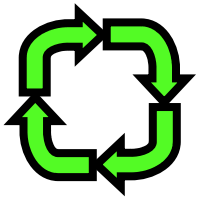
- Is one that can be created (produced) and destroyed (consumed)
- Examples :Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

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



Cycle of Accessing a Resources

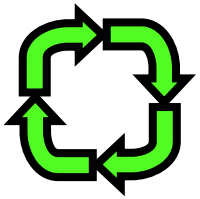
1. Request : A process, needing a resources, will request the OS for assignment of the needed resources. Then the process waits, till OS assigns it an instance of the requested resources.
2. Assignment: The OS will assign to the requesting process an instance of the requested resources, whenever, it is available. Then , the process comes out of its waiting state.
3. Use : The process will use the assigned resources. In case, the resource is non-sharable, the process will have exclusive access to it.
4. Release : After the process finished with the use of assigned resources , it will return the resources to the system pool. The release resource can now be assigned to another waiting process.



Cycle of Accessing a Resources

- The “Request” and “Release” are System Call.
- IF a resource is protected by a Semaphore, then the “Request” would comprise a “Wait” call on the Semaphore and the “Release “ would comprise a “Signal” call on the Semaphore

.



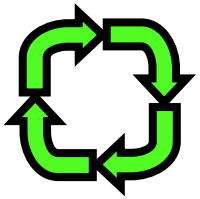
Example

- A system contains one tape and one printer and two processes Pi, Pj that use these resources as follows

Process Pi	Process Pj
Request Tape	Request Printer
Request Printer	Request Tape
Use Tape and Printer	Use Tape and Printer
Release Printer	Release Tape
Release Tape	Release Printer

- Show that the set of process {Pi,Pj} is in deadlock state.

.



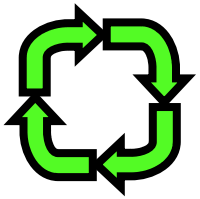
Solution

- Resources requests by Pi, Pj take place in the following order:

1. Process Pi requests the tape.
2. Process Pj requests the printer.
3. Process Pi requests the printer
4. Process Pj requests the tape.

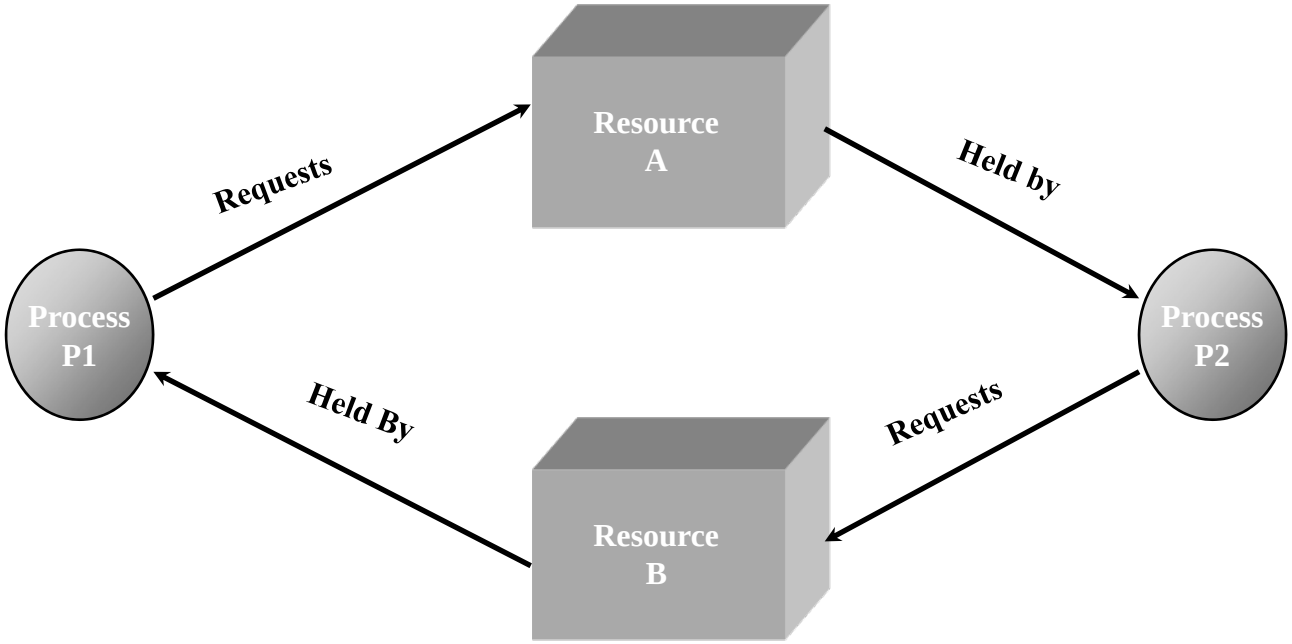
- The first two (1) and (2) requests are granted immediately, because a tape and a printer exits in the system.
- Now process Pi holds the tape and Pj holds the printer.
- When Pi asks for the printer, it is blocked until Pj releases the printer.
- Similarly, Pj is block until Pi releases the tape.
- So the set of processes {Pi,Pj} is in deadlock state.

Conditions of Deadlock



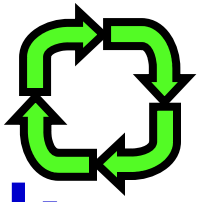
- Necessary (but not sufficient)
  - Mutual exclusion – Everyone abides by the rules
    - only one process may use a resource at a time.
    - no process may access resource allocated to another.
  - Hold-and-wait
    - a process may hold allocated resources while awaiting assignment of other resources.
  - No preemption
    - no resource can be forced to free a resource.
- Circular wait (sufficient)
  - a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain (consequence of the first three conditions)
- Other conditions are necessary but not sufficient for deadlock - all four conditions must hold for deadlock - Unresolvable circular wait is the definition of deadlock!

Circular Wait



Methods for Handling Deadlocks

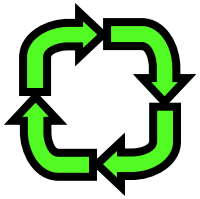
- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX



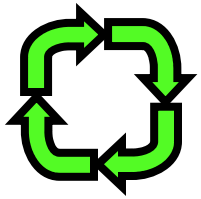
Strategies for dealing with deadlock

1. Just **ignore** the problem.
2. **Detection** and **recovery**.
   - Let deadlocks occur, detect them and take action.
3. Dynamic **avoidance** by careful resource allocation.
4. **Prevention**, by structurally negating (killing) one of the four required conditions.

Deadlock ignorance (Ostrich Algorithm)

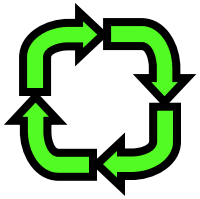


- When storm approaches, an ostrich puts his head in the sand (ground) and pretend (imagine) that there is no problem at all.
- **Ignore** the **deadlock** and **pretend** that **deadlock never occur**.
- Reasonable if
  - deadlocks **occur very rarely**
  - **difficult to detect**
  - **cost** of prevention is **high**
- UNIX and Windows takes this approach



The Ostrich Algorithm

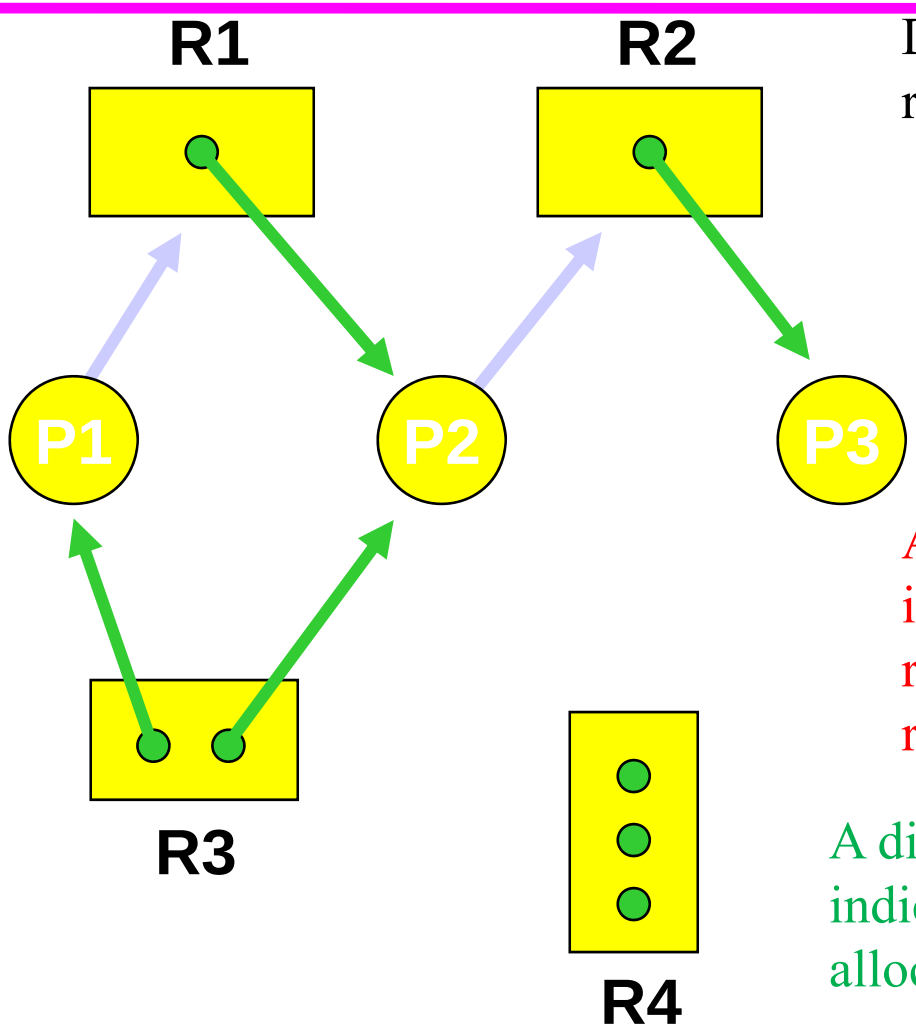
- Pretend there's no problem
- Reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting
- Trade off between convenience and correctness



Deadlock detection for single resource

- Algorithm for detecting deadlock for single resource
  1. For each node, N in the graph, perform the following five steps with N as the starting node.
     i. Initialize L to the empty list, designate all arcs as unmarked.
     ii. Add current node to end of L, check to see if node now appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.
     iii. From given node, see if any unmarked outgoing arcs. If so, go to step 4; if not, go to step 5.
     iv. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 2.
     v. If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 2.

Resource Allocation Graph



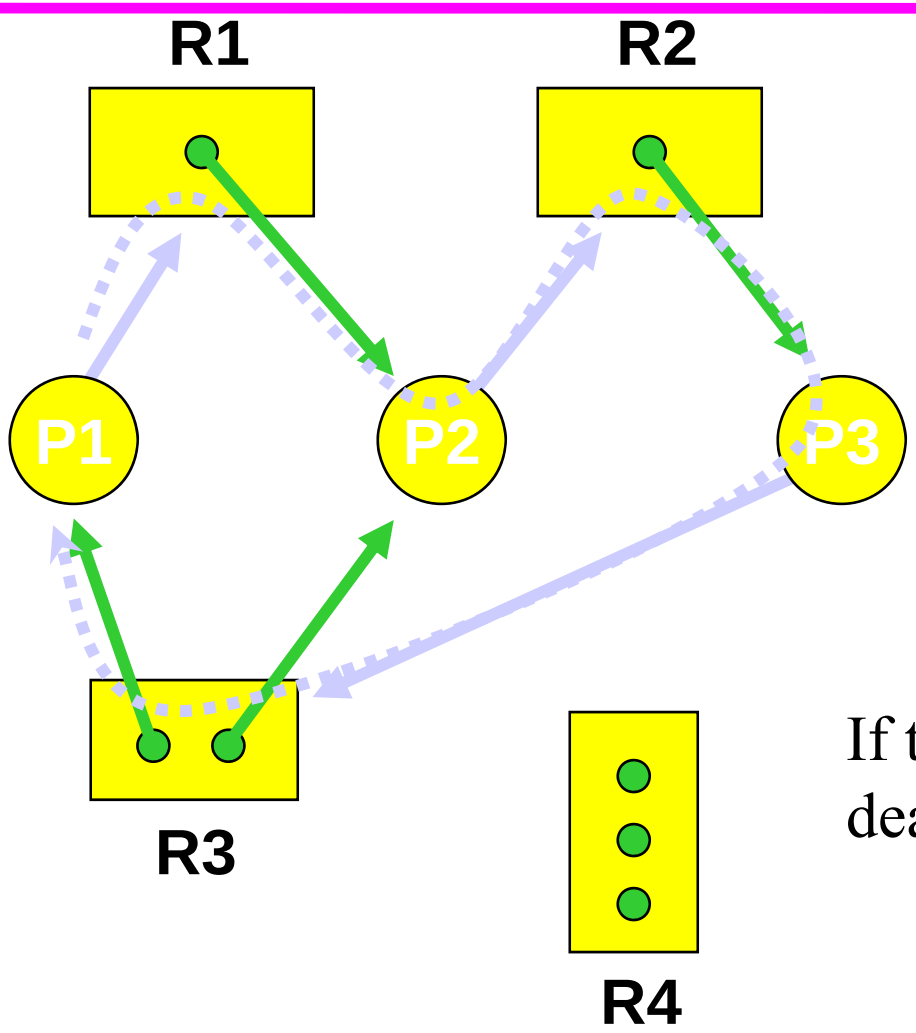
Deadlocks can be described using resource allocation graph.

Vertices: circles are Processes, rectangles are Resources.

A directed edge from Pi to Rj indicates process Pi has requested an instance of resource Rj

A directed edge from Rj to Pi indicates resource Rj has been allocated to process Pi

Resource Allocation Graph



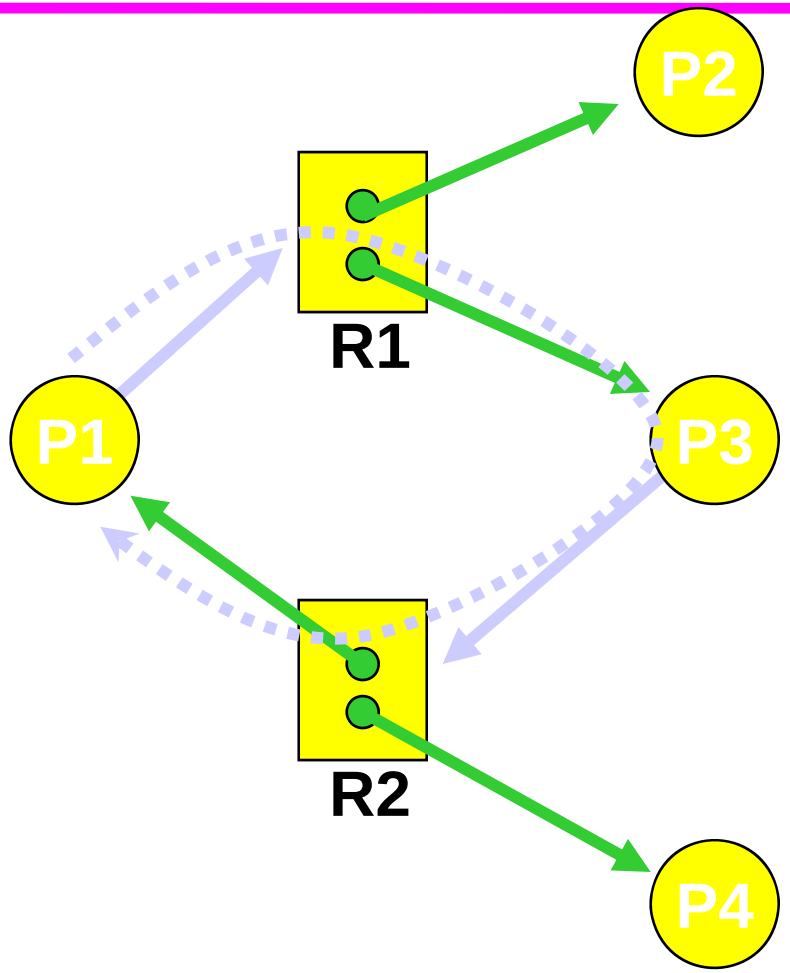
Is there a cycle (ie. some number of vertices connected in a closed chain)?

If a graph contains no cycles, then no process in the system is deadlocked

If the graph contains a cycle, deadlock MAY exist.

Is there deadlock? Yes

Deadlock?

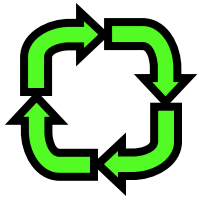


Is there a cycle?

Yes

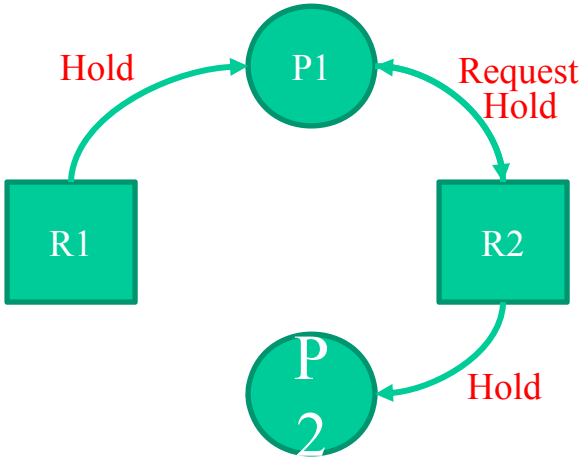
Is there deadlock?

No



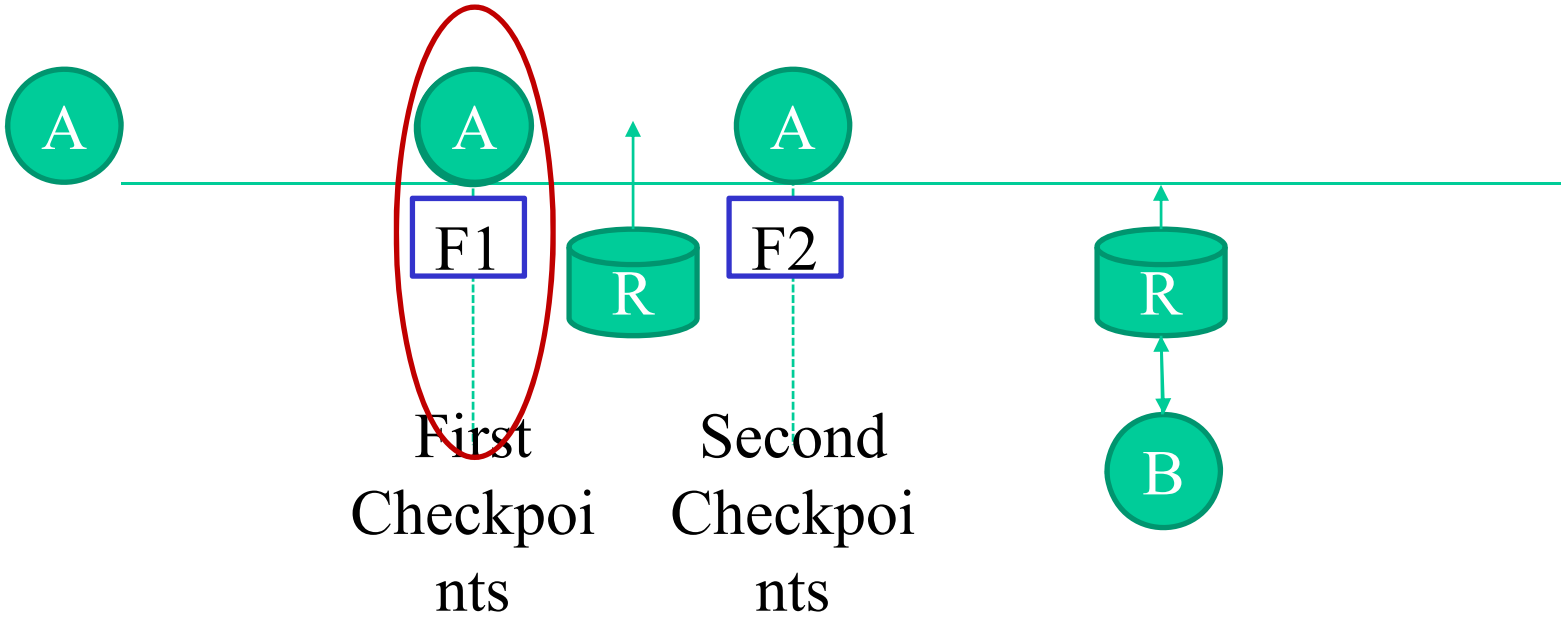
Deadlock recovery

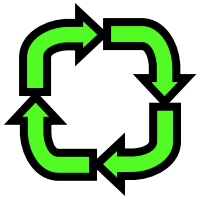
1. Recovery through pre-emption
   - In this method **resources are temporarily taken away** from its current owner and give it to another process.
   - The **ability to take a resource away from a process**, have **another process use it**, and then **give it back without the process** noticing it is highly **dependent on the nature of the resource**.
   - Recovering this way is frequently difficult or impossible.



Deadlock recovery

2. Recovery through rollback
   - **PCB (Process Control Block)** and **resource state** are **periodically saved at "checkpoint"**.
   - When **deadlock is detected**, **rollback the preempted process up to the previous safe state** before it acquired that resource.
   - **Discard the resource manipulation** that occurred after that checkpoint.
   - Start the process after it is determined it can run again.

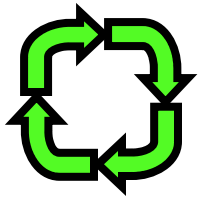




Deadlock recovery

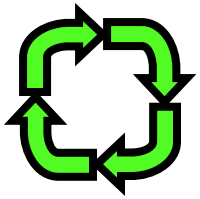
3. Recovery through killing processes
   - The simplest way to break a deadlock is to **kill one or more processes**.
     - Kill all the process involved in deadlock
     - Kill process one by one.
       - After killing each process check for deadlock
         - » If deadlock recovered then stop killing more process
         - » Otherwise kill another process

Prevent Deadlock



- Eliminate Mutual Exclusion
  - Use non-sharable resources
- Eliminate Hold and wait
  - Guarantee that when a process requests a resource, it does not hold any other resources
    - System calls requesting resources precede all others
    - A process can only request resources when it has none
    - Usually results in low utilization of resources
- Allow Preemption
  - If a process holds resources and requests more that cannot be allocated, all its other resources are preempted
    - If you can't hold all, you can't hold any
    - Process is restarted only when it can have all
    - Works for resources whose state can be easily saved and restored later such as registers or memory.

Prevent Deadlock

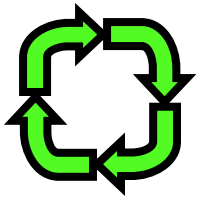


- Eliminate Circular Wait
  - Impose a total ordering of all resources (transitivity, antisymmetry)
  - Require that all processes request resources in increasing order.
  - Whenever a process requests a resource, it must release all resources that are lower
- With this rule, the resource allocation graph can never have a cycle.
  - May be impossible to find an ordering that satisfies everyone

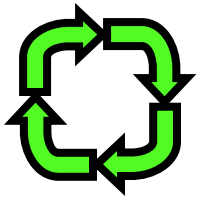
1 ≡ Card reader
2 ≡ Printer
3 ≡ Plotter
4 ≡ Tape drive
5 ≡ Card punch

Processes request resources whenever, but must be made in numerical order.
A process may request first printer and then a tape drive (order: 2, 4), but not a plotter and then a printer (order: 3, 2).

Avoid Deadlock



- Allow general requests, but grant only when ***safe***
- Assume we know the maximum requests (claims) for each process
  - Process must state it needs
    - Ie. max of 5 A objects, 3 B objects, 2 C objects.
  - Do not need to use its max claims
    - Ie. Ok to set max=5 and only use 3
  - Can make requests at any time and in any order
- Process Initiation Denial
  - Track current allocations
  - Assume all processes may make maximum requests at the same time
  - Only start process if it can't result in deadlock regardless of allocations

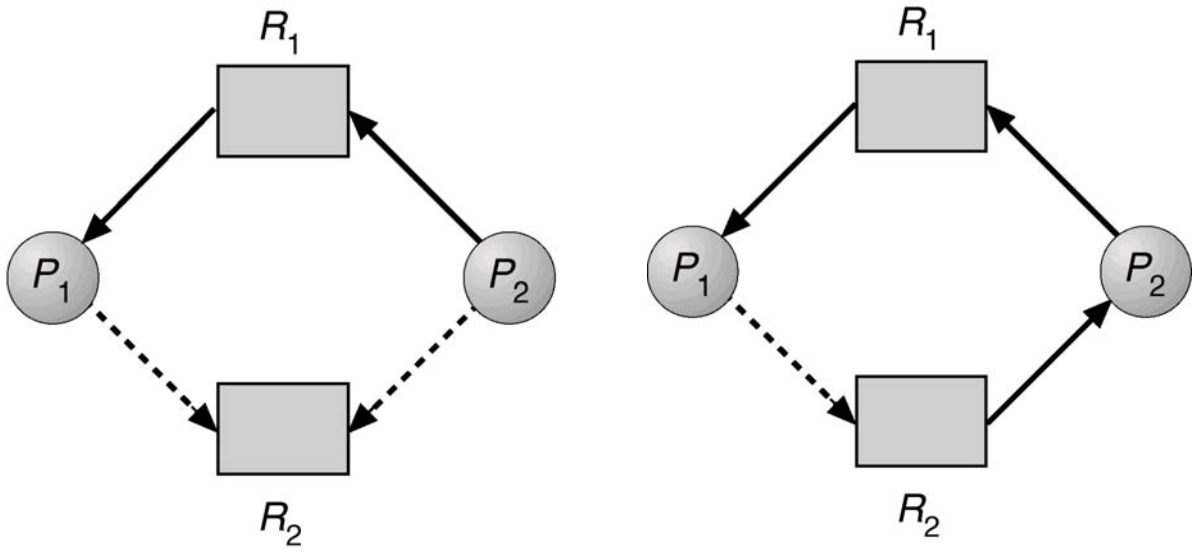


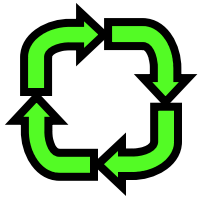
Safe State

- A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.
- Safe sequence $<P_1, P_2, ..., P_n>$:
  if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j with $j < i$.
- If no such sequence exists, then the system is said to be *unsafe*.

Safe State

- A request can be granted only if converting the request edge to assignment edge does not result in the formation of a cycle in the resource-allocation graph.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state. Otherwise, process have to wait.



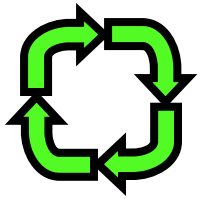


Safe State

- Example: 12 tape drives, 3 processes.

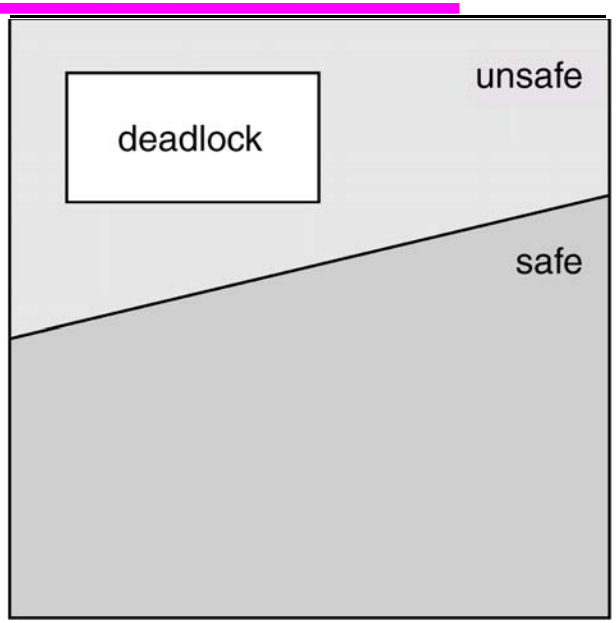
	max needs	current holding
P0	10	5
P1	4	2
P2	9	2

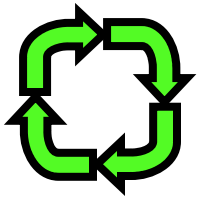
- What sequence leads to safe and unsafe state.
- Safe for <P1,P0,P2>
- If P2 request one more, system go into unsafe state.



Basic Facts

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

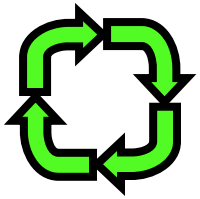




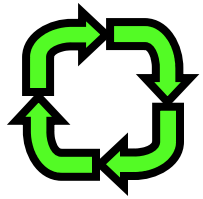
Deadlock avoidance

- Deadlock can be avoided by **allocating resources carefully**.
- Carefully **analyze each resource** request to **see if it can be safely granted**.
- Need an algorithm that can always avoid deadlock by making right choice all the time **(Banker's algorithm).**
- Banker's algorithm for single resource
- Banker's algorithm for multiple resource

Banker's Algorithm



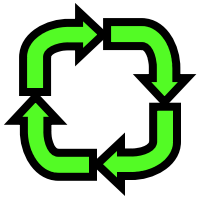
- Banker's Algorithm is used to determine whether a process's request for allocation of resources be safely granted immediately.
- The grant of request be deferred to a later stage.
- For the banker's algorithm to operate, each process has to a priori specify its maximum requirement of resources.
- A process is admitted for execution only if its maximum requirement of resources is within the system capacity of resources.
- The Banker's algorithm is an example of resource allocation policy that avoids deadlock.



Banker's Algorithm

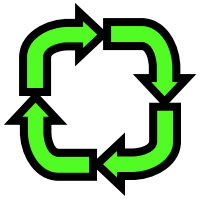
- For the banker's algorithm to work, it should know three things:
  - How much of each resource each person could maximum request [MAX]
  - How much of each resource each person currently holds [Allocated]
  - How much of each resource is available in the system for each person [Available]
- So we need MAX and REQUEST.
- If REQUEST is given MAX = ALLOCATED + REQUEST
- NEED = MAX – ALLOCATED
- A resource can be allocated only for a condition.
  - REQUEST<= AVAILABLE or else it waits until

Banker's Algorithm



- Following **Data structures** are used to implement the Banker's Algorithm:
- Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.
- **Available :**
  - It is a 1-d array of size **'m'** indicating the number of available resources of each type.
  - Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$
- **Max :**
  - It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
  - Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Banker's Algorithm



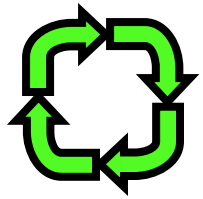
- **Allocation :**
  - It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
  - Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$
- **Need :**
  - It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
  - Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$ for its execution.
  - Need [i, j] = Max [i, j] – Allocation [i, j]
  - $Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.
- Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

- The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
   Initialize: Work = Available
   Finish[i] = false; for i=1, 2, 3, 4….n
2. Find an i such that both
   a) Finish[i] = false
   b) $Need_i$ <= Work
   if no such i exists goto step (4)
3. Work = Work + Allocation[i]
   Finish[i] = true
   goto step (2)
4. if Finish [i] = true for all i
   then the system is in a safe state

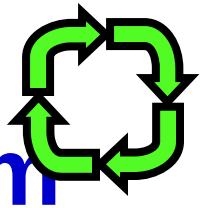
Resource-Request Algorithm



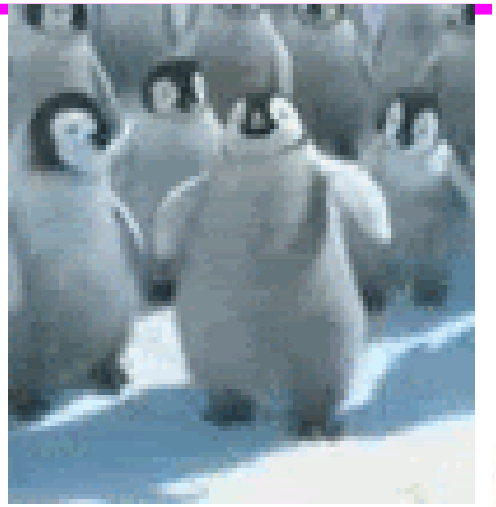
- Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

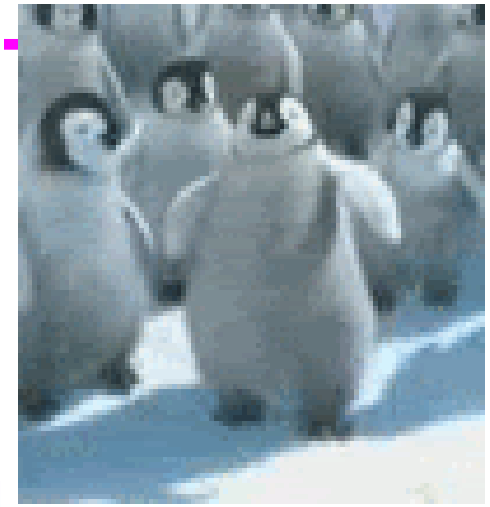
1. If $Request_i <= Need_i$
   Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i <= Available$
   Goto step (3); otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
   $Available = Available – Request_i$
   $Allocation_i = Allocation_i + Request_i$
   $Need_i = Need_i – Request_i$

Disadvantages of Banker's Algorithm



- It requires the number of processes to be fixed; no additional processes can start while it is executing.
- It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
- It allows all requests to be granted in finite time, but one year is a finite amount of time.
- Similarly, all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
- All processes must know and state their maximum resource need in





Minal Shah