## Overview

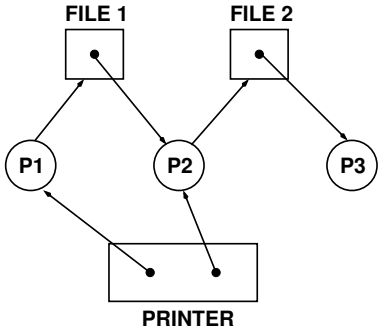
- No self-respecting course on operating systems can omit deadlock from its syllabus so we briefly cover it here
- We will describe the problem itself, illustrate it with resource allocation diagrams, examine the conditions under which it arises and finish by examining some of the methods proposed for solving it

## Definition

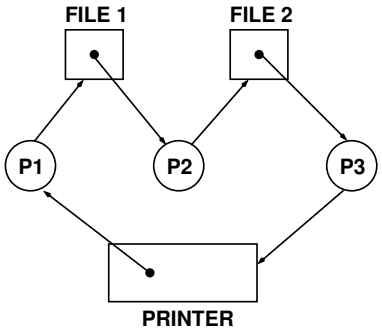
- Resources can be considered as belonging certain categories (files, printers, semaphores. . . ) with resource instances allocated to requesting processes
- A set of processes (or threads) is deadlocked when every process in the set is waiting for a resource which is being held by another process in the set
- Two conditions must hold:
  1. Every process in the set is waiting
  2. Resources waited on are held within the set

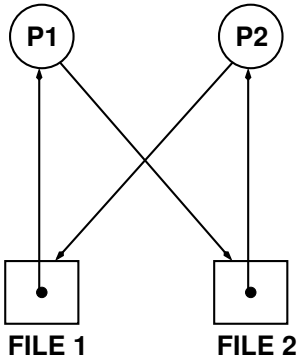
## Resource Allocation Graph

- An arrow from resource to process represents a resource allocation
- An arrow from process to resource represents a resource request




## Depicting Deadlock



## Deadlock Detection

- If the resource allocation graph does not contain a cycle then it cannot be deadlocked
- If the resource allocation graph does contain a cycle then it may be deadlocked
- Whether a resource allocation graph that does contain a cycle is deadlocked or not depends on the number of resource instances available
- It turns out that a cycle (circular wait) is not the only condition that must hold for deadlock to occur...

## Necessary and Sufficient Conditions

- Four conditions must be true (necessary) for deadlock to occur and once all four are true the system is deadlocked (sufficient):
  1. Mutual Exclusion: Processes have exclusive ownership of resources once acquired i.e. there is no sharing
  2. Hold and Wait: Processes hold onto currently held resources while waiting for others
  3. No Preemption: Resources are only released voluntarily by a process i.e. they cannot be forcefully taken back by the operating system
  4. Circular Wait: A cycle exists in the resource allocation graph

## Approaches to the Problem

- Prevent deadlock: Given all of the conditions above must be true for deadlock to occur if we make one of them false the problem of deadlock disappears
- Avoid deadlock: Check before allocating a resource whether if by allocating it we might end up deadlocked; if we might then do not allocate it
- Allow and detect deadlock: Detect deadlock when it happens and attempt to recover from it
- Do nothing: Accept the problem, try to avoid it by programming carefully and if it happens...

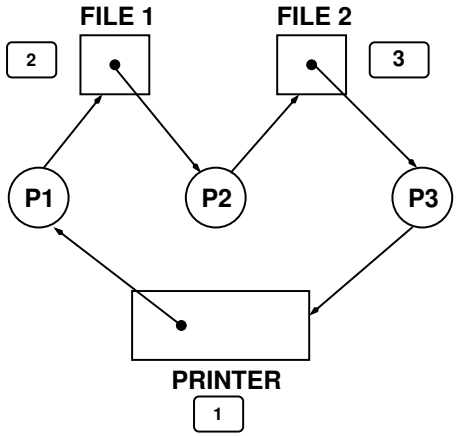
## Deadlock Prevention

- We need to ensure at least one of the necessary conditions is at all times false...
- Mutual Exclusion: All resources must be made shareable, not feasible with some resources e.g. printer
- Hold and Wait: Insist a process gives up some resources if waiting on another; many difficulties e.g. a process may starve as it waits for some set of popular resources to each be free upon request

## Deadlock Prevention

- No Preemption: Process requests an unavailable resource and we allow OS to take it from its current holder (after saving its state so it can be restored later); this will work for registers (as in traditional preemption) but not for other resources like a printer
- Circular Wait: Number all resources and insist a process request resources in increasing numerical order; now it's impossible for a process to hold a high priority resource and be waiting on a lower one; no cycles can arise but it is impractical to ask programmers to request resources in a given order plus resources are dynamic, they come and go

## Deadlock Prevention




## Deadlock Avoidance

- Deadlock prevention guarantees no deadlock but too impractical an approach for the real world
- Deadlock avoidance requires processes declare in advance resources they will need (note the unlikelihood of this in the real world also)
- Each time a request is made the OS decides whether or not to grant the resource
- A safety assurance algorithm is employed:
  1. System begins in a safe state
  2. The OS never allocates a resource which could lead to unsafe state (a conservative approach)

## Banker's Algorithm

- The Banker's Algorithm is used as the safety assurance algorithm and is used to track and manage current resource allocations
- When a resource request arrives a tentative allocation is made to produce a new state
- The Banker's Algorithm is then then applied to the new state to check whether it is a safe or unsafe one:
  1. If the new state is safe the resource is granted
  2. If the new state is unsafe the allocation is refused
- All of this is time-consuming (not to mention impractical)

## Deadlock Detection and Recovery

- This approach is a more optimistic one: assume deadlock is unlikely to occur but if it does happen, detect and recover from it
- A deadlock detection algorithm runs periodically (like the one used by the Banker's Algorithm)
- When deadlock is detected we generally have two options:
  1. Terminate a process
  2. Preempt some resources from an owner process

## Process Termination

- Some caution must be exercised before killing a process since doing so may leave system in inconsistent state e.g. if the victim is in the middle of updating some data or a file
- Selection criteria:
  1. Terminate low priority processes first
  2. Allow processes to run if nearly done (but how would the OS know?) and only terminate young processes
  3. Terminate those processes holding the most popular or largest number of resources

## Resource Preemption

- Rather than the drastic killing of a process, we might take back some of the resources it holds
- The resources taken from some process are given to another, while the victim is asked to wait
- When a resource is preempted the victim process's state must be rolled back to some previous and consistent state before it had acquired resource
- The rollback point may be identified by a programmer or the OS and the process's state must be saved at that point so rollback can later be implemented if need be

## Do Nothing

- The most common approach is to the problem of deadlock is to do nothing and is the one adopted by most operating systems
- If deadlock arises and e.g. some process or processes "stop responding" then it is up to the user to take appropriate action
- Some care can be taken to help alleviate the problem e.g. when coding in the Linux kernel rules govern in what order locks should be acquired and released in an attempt to prevent cycles that could lead to deadlock