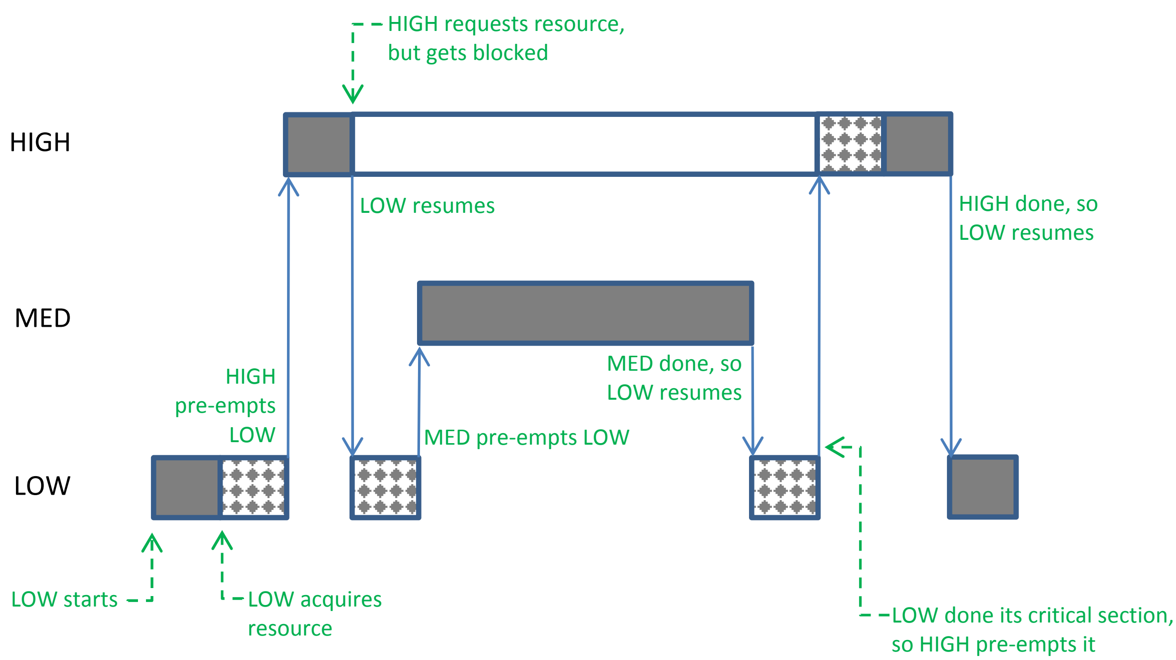
2. Priority Inversion

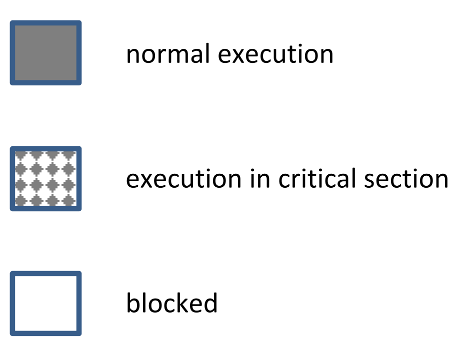
Assume the following:

- two tasks share the same mutually-exclusive resource
- the resource is protected by a locking mechanism (e.g. blocking semaphore)
- the two tasks have fixed priorities indicated by their names: HIGH and LOW
- let us call the time when HIGH or LOW owns the resource as its "critical section" of code
- a third task of fixed medium priority does not require the resource

Then consider the following scenario:



where:



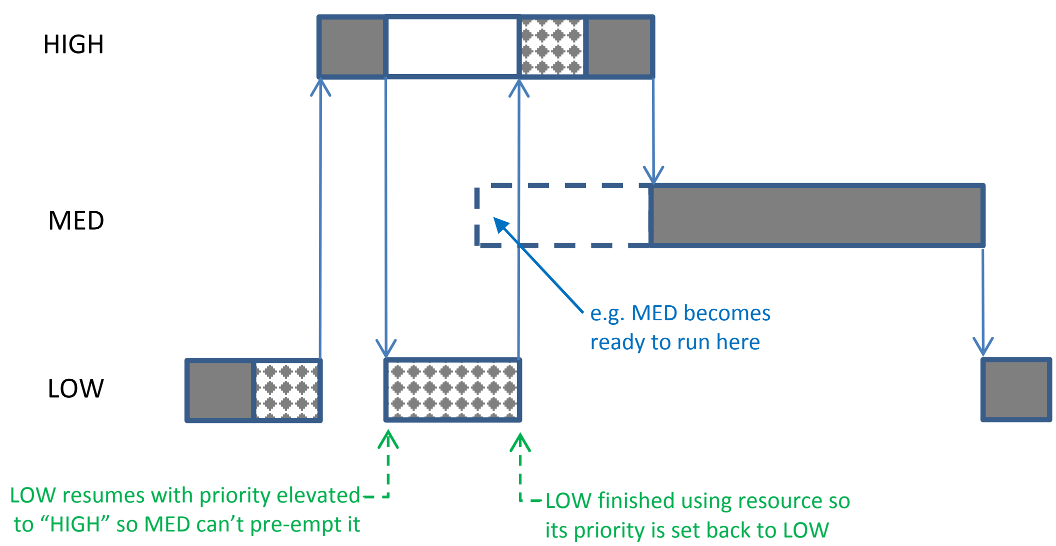
Solution – Priority Inheritance

To invoke priority inheritance, do the following:

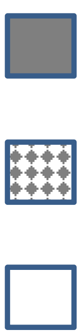
When a higher-priority task gets blocked from executing because a lower-priority task has ownership of a mutually-exclusive shared resource that the higher-priority task also wants, then, temporarily, raise the priority of the lower-priority task to match that of the higher-priority task.

note: SYS/BIOS has "GateMutexPri" to support priority inheritance

Here is the previous scenario, but this time with priority inheritance:



where:



normal execution

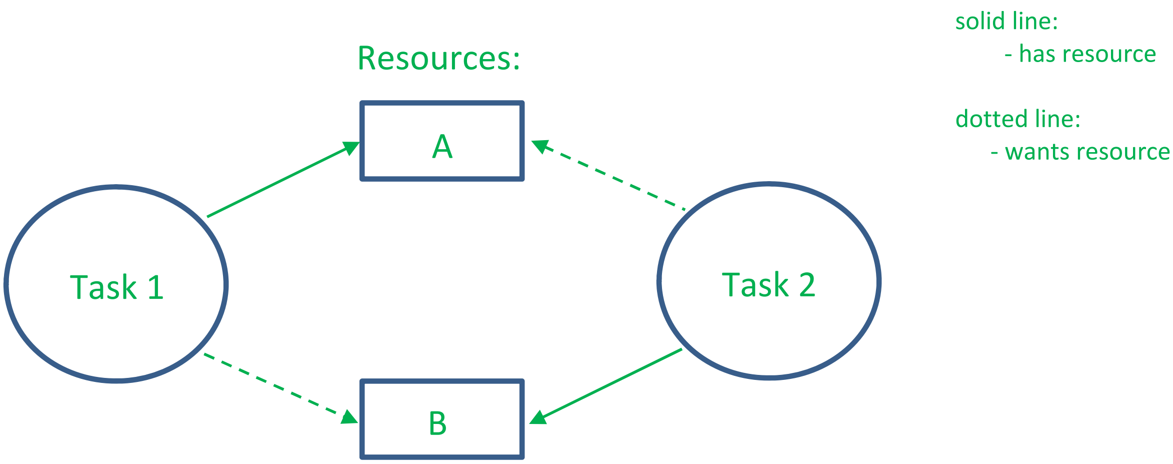
execution in critical section

blocked

3. Deadlock

Definition:

Deadlock in computer programming is when two or more tasks are each waiting for one another to finish before continuing.



Scenario:

1. T1 requests A and receives it
2. T2 requests B and receives it
3. T1 requests B and waits for it
4. T2 requests A and waits for it
5.

Conditions for Deadlock to Occur

All four conditions must be met for deadlock to be possible.

1. mutual exclusion	i.e., only one task can use a resource at one time
2. task can hold and wait	i.e., a task can hold a resource while waiting for another resource
3. circular pend	i.e., circular chain of tasks holding resources needed by others in the chain
4. no pre-emption of resource hold	i.e., a resource can only be released by the task itself

Conditions 1–4: Coffman et. al.

Handling Deadlock

large, general-purpose operating systems → usually the responsibility of the OS to handle deadlock

small RTOSs → usually the responsibility of the user to handle deadlock

Three Basic Ways to Handle Deadlock

1. *Prevent* – follow some policy that guarantees that at least one of the four conditions of deadlock does not exist

   a) remove mutual exclusion

   → not always possible since some resources cannot be shared "simultaneously"

   b) remove multiple hold and wait

   → requires task to acquire *all* resources it needs at the same time

   c) remove circular pend condition

   → requires that resources always be acquired in a certain order
   e.g. task must request resource A before resource B or resource B before resource C , etc
   i.e., in a hierarchical order

   d) allow pre-emption of resource holds ← also, SYS/BIOS has a timeout feature for semaphore pend

   → if a task can't acquire a particular resource, it must surrender *all* resources it holds and try again

2. *Detect and Recover* – allow the four conditions of deadlock to be present and detect when deadlock occurs

   - can be very difficult to do so
   - more appropriate for a large OS
   - a last resort: watchdog times out and issues processor reset

3. *Avoid* - allow the four conditions of deadlock to be present and avoid the occurrence of deadlock

   → dynamically detect if allowing a resource request could cause or lead to deadlock → if yes, don't grant request

Deadlock Avoidance

- each time a task requests a resource, the OS observes the current resource allocation state

- the resource allocation state consists of:
    - number of resources available
    - number of resources already allocated
    - maximum number of resources each task could request

- OS runs an algorithm with the resource allocation state as its input

- if the algorithm determines that granting the new request will not lead to an “unsafe” state, it grants the request

- if the algorithm determines that granting the new request will lead to an “unsafe” state, it denies the request

safe state: will not lead to deadlock

unsafe state: can potentially lead to deadlock

Example of Safe and Unsafe States

Assumptions:

Are the four conditions for deadlock present in this example?

- 3 tasks
- resources = 12 identical blocks of memory
- tasks have equal priority
- each task needs a certain number of blocks of memory to complete its processing
- each task has a maximum number of blocks it can ever require to complete its processing
- it is not important which task completes its processing first

Current State:

Process	Number of Memory Blocks Task Has	Maximum Number of Memory Blocks Task Could Need
Task 1	5	10
Task 2	2	4
Task 3	2	9

Safe or Unsafe?

path to completion: T2
T1
T3

scheduled in this order by OS

→

Task 1 requests another memory block:

Process	Number of Memory Blocks Task Would Have	Maximum Number of Memory Blocks Task Could Need
Task 1	6	10
Task 2	2	4
Task 3	2	9

Safe or Unsafe?

path to completion: T2
T1
T3

→

You do these:

or Task 2 requests another memory block:

Process	Number of Memory Blocks Task Would Have	Maximum Number of Memory Blocks Task Could Need
Task 1		
Task 2		
Task 3		

Safe or Unsafe?

or Task 3 requests another memory block:

Process	Number of Memory Blocks Task Would Have	Maximum Number of Memory Blocks Task Could Need
Task 1		
Task 2		
Task 3		

Safe or Unsafe?