**Examples of starvition:**

* **the customer has to wait for a long time because he doesn't know that The previous customer has already done and there is another customer talking his turn.**
* **The barber calls out the customer randomly.**

**Explanation for real world application and how did apply the problem:**





**What is deadlock?!!**

* Waiting processes may never change the states



what is deadlock sleeping problem??

sleeping barber problem is a classic inter-process communication and synchronization problem between multiple operating system processes.



What are the main causes of deadlocks and can they be prevented?

* Mutually exclusive events
* Circular wait
* No preemption
* Hold and wait

sleeping barber problem as pseudocode :

* The barber sleeps when there is no customer in the waiting room. Therefore, when a customer arrives, he should wake up the barber
* As we’ve seen from the figure, customers can enter the waiting room and should wait for whether the barber is available or not
* If other customers keep coming while the barber is cutting a customer’s hair, they sit down in the waiting room
* Customers leave the barbershop if there is no emp ty chair in the waiting room

**The figure below shows the main characteristics of the problem:**



**Methods for Handling Deadlocks**

* Generally speaking there are three ways of handling deadlocks:
* Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
* Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
* Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.
* In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. ( Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm. )
* Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
* If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

**There are four main causes for deadlocks :-**

1- Mutually exclusion events

2- Hold and wait

3- No preemption

4- Circular wait

**Deadlock Prevention**

**Mutually Exclusion Events**

* Shared resources such as read-only files do not lead to deadlocks.
* Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

**Hold and Wait**

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

- Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.

- Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

- Either of the methods described above can lead to starvation if a process requires one or more popular resources.

**No Preemption**

* Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
* One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, ( preempted ), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
* Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
* Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

**Circular Wait**

-One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing ( or decreasing ) order.

-In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.

-One big challenge in this scheme is determining the relative ordering of the different resources

**The problem of starvation can be solved with a first-in first-out (FIFO) queue. The semaphore would provide two functions: wait() and signal() , which in terms of C code would correspond to P() and V() , respectively**

* **Example in our project**
* This is an example of the sleeping barber
* Customers come to the barber one after the other, and the place becomes crowded with customers. In order for the crowd to decrease, the barber must shave for the one who came first
* .