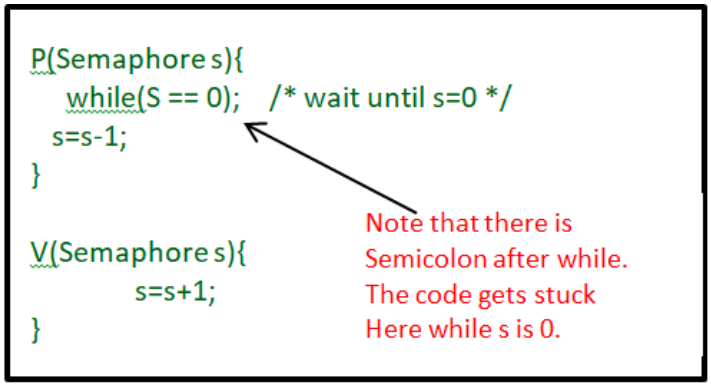
Teaching Assistant

Teaching Assistant (TA) holds office hours where they provide help to students with their programming assignments. The TA's office only has room for one desk with a chair and a computer. There are three chairs outside the office where students may sit and wait if the TA is currently helping another student. If there, are no available chairs in the waiting area, the student shows up later and goes for the programming. When the TA has finished helping a student. He takes the next student and begins to help them. If there are no waiting students, he returns to his chair in the office and takes a nap. If a student shows up and sees the TA sleeping, they sit in his chair and wake him up.
Using threads and semaphores, implement a solution in JAVA that coordinates the activities of the TA and the students.

Semaphore

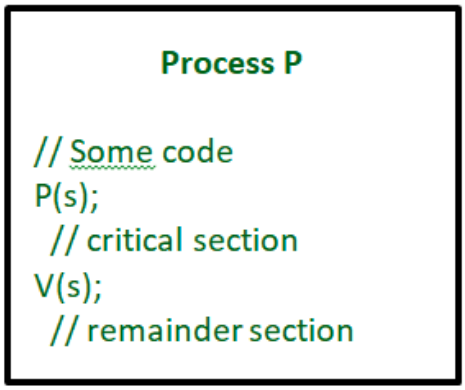
Semaphore is a simply a variable. This variable is used to solve critical section problems and to achieve process synchronization in the multi-processing environment. The two most common kinds of semaphores are Counting Semaphores and Binary Semaphores. Binary semaphores can take the value 0 & 1 only. Counting semaphores can take nonnegative integer values.

Two standard operations, wait and signal are defined on the semaphore. Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care of by signal operation. The wait and signal operations are also called P and V operations. The manipulation of semaphore (S) takes place as follows:



The wait command P(S) decrements the semaphore value by 1. If the resulting value becomes negative then P command is delayed until the condition is satisfied.

The V(S) i.e. signals operation increments the semaphore value by 1.



Mutual exclusion on the semaphore is enforced within P(S) and V(S). If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting. The semaphore operation is implemented as operating system services and so wait and signal are atomic in nature i.e. once started, execution of these operations cannot be interrupted. Thus semaphore is a simple yet powerful mechanism to ensure mutual exclusion among concurrent processes.

The Flow of the Project:

1) Create n students, each will run as a separate thread.

2) Student threads will alternate between programming for a period of time and seeking help from the TA.

3) If the TA is available, students will obtain help

4) If the TA is not available, students will sit in a chair outside the office

5) If no chairs are available, students will resume programming and seek help at a later time

6) If the TA is sleeping, the student will notify TA with a semaphore

7) Use a semaphore to indicate a student in help mode

8) When a TA finishes helping a student, the TA must check to see if there are students waiting for help outside the office.

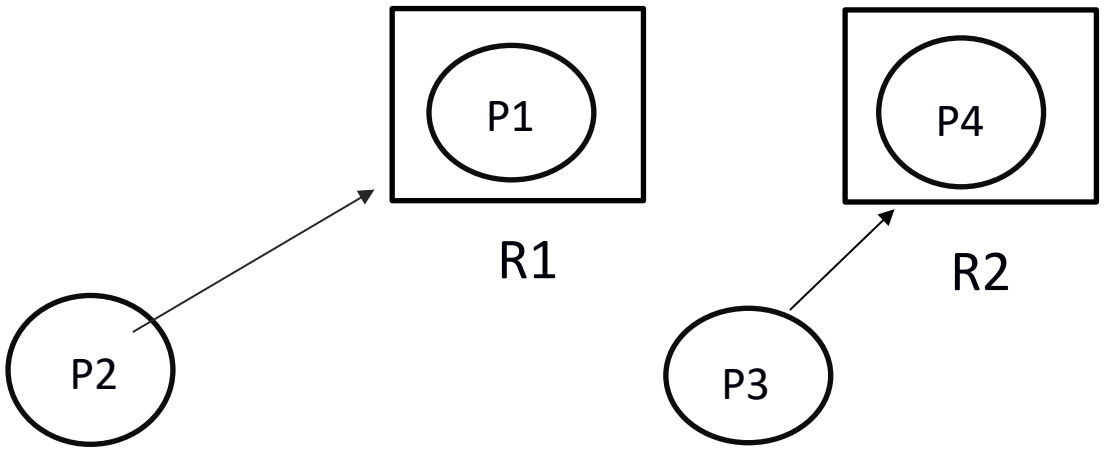
9) The TA must help the students waiting outside the office in the order that they started waiting.

10) If no students are waiting, then the TA can resume napping.

Starvation:

Occurs when one or more threads in our program are blocked from gaining access to a resource and as a result, cannot make a progress.

As we can see:



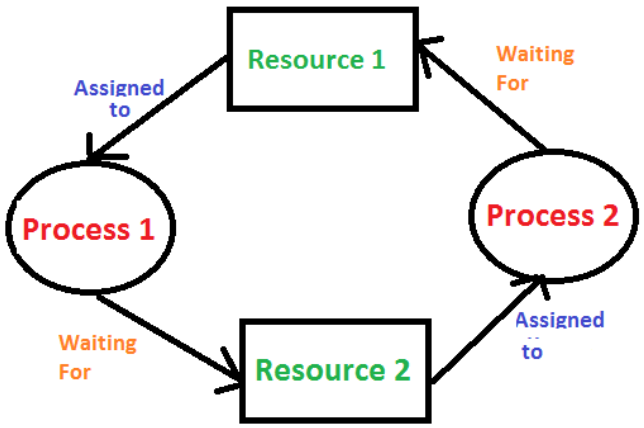
- P2 and P1 are blocked from gaining access to R1 and R2.

In our scenario, a problem of starvation will occur if a student is waiting outside the office to solve a problem with the TA.

We solve this by implementing the waiting data structures as a linked list where the first come the first served.

Deadlock

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Deadlock can arise if the **following four conditions hold simultaneously (Necessary Conditions)**
Mutual Exclusion: Two or more resources are non-shareable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock
There are three ways to handle deadlock
1) Deadlock prevention or avoidance: The idea is to not let the system into a deadlock state.
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process will need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem altogether: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Disadvantages of Deadlock

Here, are cons/ drawback of using deadlock method

- Delays process initiation
- Processes must know future resource need
- Pre-empts more often than necessary
- Dis-allows incremental resource requests
- Inherent preemption losses.

How to avoid deadlock in java

These are some of the guidelines using which we can avoid most of the deadlock situations.

- **Avoid Nested Locks**: This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It's almost impossible to get deadlock situation if you are working with only one object lock.

- **Lock Only What is Required**: You should acquire lock only on the resources you have to work on, for example in above program I am locking the complete Object resource but if we are only interested in one of it's fields, then we should lock only that specific field not complete object.
- **Avoid waiting indefinitely**: You can get deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with maximum time you want to wait for thread to finish.