- **PROJECT NAME**: Deadlock Detection
- **SUBJECT:** Operating System Lab (COM-311)
- **SUBMITTED TO:** Ms. Pragti Jamwal and Ms. Swati Goyal
- **GROUP MEMBERS:** Aaditya Nargotra, Raghav Jamwal, Manav Singh Jamwal, Abhimanyu Peshin
- **ROLL NUMBERS:** 2021A1R126, 2021A1R128, 2021A1R137,2021A1R150

<u>Abstract</u>: Sharing and allocating system resources between running and new processes is an operating system task. In a multiprocessing environment, allocation of resources can become a point of conflict. Such conflict may lead the system into a state known as deadlock. The scope of the topic is to identify the conditions for deadlocks and identify the methods for detection.

<u>Introduction:</u> Parallel processing is driven by the utility maximization of multiple resources. System resources however, are not infinite, and therefore, need to be allocated properly for tasks to be completed. When dealing with resource allocation, it is likely that conflicts may appear. A set of resources for example, may need to be obtained for a process to begin execution, but the entire, or part of, this resource set is occupied by other processes. There are many issues pertaining to efficient resource allocation, but the fundamental ones are those involved with the conflict free utilization.

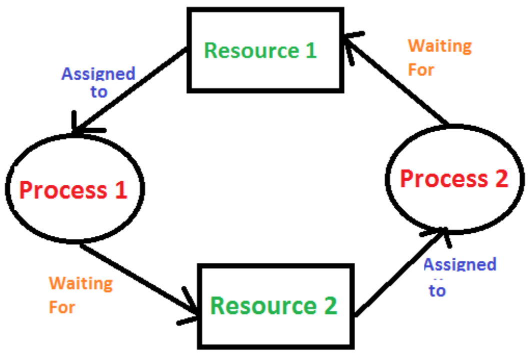
Processes and Resources: It is apparent that the notions of process and resource are tightly linked. A process is a task, identified as a sequence of executing instructions, or a collection of instructions forming a program. The responsibility of managing those processes lays on the operating system which enforces when and how they are executed. A resource, on the other hand, is an inclusive term of all system resources such as printers, disks, tape-drives, processor and shared memory capacities. However, resources are not treated equally by the operating system, and depending on their type, handle processes differently. Non-pre-emptiable resources are used by processes that require uninterrupted resource utilization to be completed. Allowing another process to even temporarily interrupt and use that resource would jeopardize the outcome of the interrupted process. Pre-emptiable resources on the other hand, require operating system control for processes to safely "exchange" the utilization of the resource. Processes that require the exclusive use of a resource are considered mutually exclusive. Mutual exclusion determines how a process would behave while competing for a specific resource, while another one is using that resource. Metaphorically, resources can be treated as a path, where after a certain point, only one process can "walk" on it. The critical region is an artificial term describing the exclusivity of a process over the utilization of a resource. A process needs the exclusive usage of a system resource, for a specific time, in order to complete its assigned task. An example could be the usage of any resource. If we assume that the process in question is P1, then when it "enters" the critical region

(CR), between points A and B, it should finish the assigned task without any other process, P2, being able to execute while P1 is still within the critical region. A Critical Region illustration: a process P1 is already within the critical region using the resource exclusively. As soon P1 exits (after point B), P2 will be allowed to enter the region. Mutual exclusion is the notion of guaranteeing that only one process is within the critical region at any given instance.

Deadlock Conditions: Processes do not run constantly from the time they are created until their termination; they can be identified in three different states: ready, running and blocked. At the ready state, a process is stopped, allowing some other process to run; at the running state, a process is utilizing some resource, and at the blocked state, the process is stopped and will not start running again until something triggers it to restart. A common undesirable condition is described as deadlock, which is when two processes are at a running state, and have acquired some resources, but need to exchange resources to continue. Both processes are waiting for the other to release a "requested" resource that neither one will ever do; as a result, neither one is making any progress, therefore, reaching a deadlock state. Many scenarios have been constructed to illustrate deadlock conditions with the most popular being the Dinning Philosopher's Problem. [1] In this scenario, five philosophers have five dishes of spaghetti in front of them and five forks, one on each side of the dish. The philosophers need both forks (left and right) in order to eat. During their dinner, they perform only two mutually exclusive operations, think or eat. The problem is a simplistic parallelism between processes (the philosophers) trying to obtain resources (the forks); while they are either at a running state (eating). or at a thinking state (ready). A possible deadlock condition would occur, if all five philosophers would simultaneously pick up their left forks. No right fork would be available and the dinner would end-up in a deadlock state.

<u>Deadlock Detection Algorithm</u>: Four conditions need to be occurred simultaneously for a deadlock to occur.

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait



Steps of Algorithm:

Step 1

1. Let **Work(vector)** length = m
2. **Finish(vector)** length = n
3. Initialize Work= Available.
    1. if Allocation = 0 ∀ i ∈ [0, N-1], then Finish[i] = true;
       otherwise, Finish[i]= false.

Step 2

1. Find an index i such that both
    1. Finish[i] == false
    2. Work >= Request

    If there exists no i, go to **step 4**.

Step 3

1. Work += Allocation
2. Finish[i] = true

Go to **Step 2**.

Step 4

1. For some i in [0, N), if Finish[i]==false, then the system is considered in a deadlock state. Moreover, if Finish[i]==false the process P_i is deadlocked.

Conclusion: Deadlocks are an undesirable but intriguing system condition. There are multiple variable conditions that can lead to deadlocks and as a result make them very complex and difficult to prevent. Recovering from deadlocks also has its own challenges, particularly because it is difficult to reinstate the system, to a similar one prior to deadlock, in order to complete interrupted processing. There are some formalized methods via resource trajectories and state diagrams, which can to some extent, predict deadlocks. However, completely restricting their occurrence requires a firmly designed operating system along with very efficient synchronization algorithms.