LAB # 10
Inter Process Communication II

Deadlock

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock can be defined as the *permanent blocking of a set of processes that either* compete for system resources, or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case.

Deadlock Avoidance

Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether the process should wait, or not. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's Algorithm

Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type. The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes a "safe state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

• **Available**. A vector of length m indicates the number of available resources of each type. If ***Available***[j] equals k, then k instances of resource type Rj are available.

• **Max**. An $n \times m$ matrix defines the maximum demand of each process. If ***Max***[i][j] equals k, then process Pi may request at most k instances or resource type Rj.

• **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.

• **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that ***Need***[i][j] equals ***Max***[i][j] − ***Allocation***[i][j].

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as $\boldsymbol{Allocation}_i$ and $\boldsymbol{Need}_i$. The vector $\boldsymbol{Allocation}_i$ specifies the resources currently allocated to process Pi; the vector $\boldsymbol{Need}_i$ specifies the additional resources that process Pi may still request to complete its task.

Safety Algorithm

We can now present the algorithm for finding out whether a system is in a safe state or not. This algorithm can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize
   Work = ***Available*** and ***Finish***[*i*] = ***false*** for *i* = 0, 1, ..., *n* − 1.
2. Find an index *i* such that both
   a. ***Finish***[*i*] == ***false***
   b. ***Need***$_i \leq$***Work***
   If no such i exists, go to step 4.
3. ***Work*** =***Work*** + ***Allocation***$_i$
   Finish[*i*] = ***true***
   Go to step 2.
4. If ***Finish***[*i*] == ***true*** for all *i*, then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.
Let ***Request***$_i$ be the request vector for process P_i. If ***Request***$_i$ [*j*] == *k*, then process P_i wants *k* instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1. If ***Request***$_i \leq$***Need***$_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Request***$_i \leq$ ***Available,*** go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:
   Available = ***Available***–***Request***$_i$;
   Allocation$_i$ = ***Allocation***$_i$ + ***Request***$_i$;
   Need$_i$ = ***Need***$_i$ –***Request***$_i$;

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for ***Request***$_i$, and the old resource-allocation state is restored.

Time Boxing

Activity Name	Activity Time	Total Time
Login Systems + Setting up Linux Environment	3 mints + 7 mints	10 mints
Walk through Theory & Tasks	60 mints	60 mints
Implement Tasks	80 mints	80 mints
Evaluation Time	30 mints	30 mints
	Total Duration	180 mints

Objectives/Outcomes

- To learn a inter process communication and learn how to avoid deadlocks using Banker's algorithm.

Lab Tasks/Practical Work

1. Write a short note on Banker's algorithm stating its main purpose and working mechanism.
2. Implement the Banker's Algorithm explained above in C language.