LOVELY PROFESSIONAL UNIVERSITY

Academic Task-3 (Operating System)

School of Computer Science and Engineering Faculty of Technology and Sciences

Name of the faculty member: Prof. Isha

Course Code: CSE 316 Course Title: Operating System

Term: Spring term (2019-2020)

Max. Marks:30

Student Name: Bhagesh Khatri

Student ID: 11914224

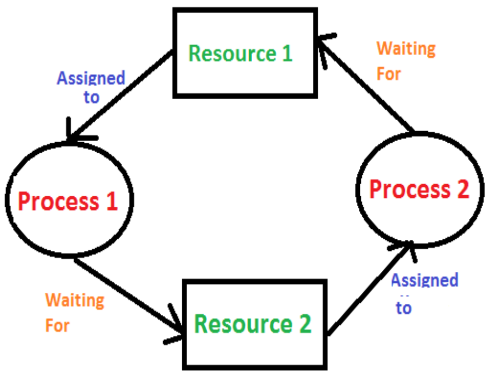
Email Address: khatribhagesh1122@gmail.com

GitHub Link: https://github.com/khatribhagesh1122

Question No. 19

1. Problem in terms of Operating System:

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 same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes 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 following four conditions hold simultaneously (Necessary Conditions)

Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)

Hold and Wait: A process is holding at least one resource and waiting for resources.

No Pre-emption: 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 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) **Ignorance:** Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

2. Description:

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 an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker's Algorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available:

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.

- Available[j] = k means there are **‘k’** instances of resource type **R_j**

Max:

- It is a 2-d array of size ‘**n*m’** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process **P_i** may request at most **‘k’** instances of resource type **R_j**.

Allocation:

- It is a 2-d array of size **‘n*m’** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process **P_i** is currently allocated **‘k’** instances of resource type **R_j**

Need:

- It is a 2-d array of size **‘n*m’** that indicates the remaining resource need of each process.
- Need [i, j] = k means process **P_i** currently allocated **‘k’** instances of resource type **R_j**
- Need [i, j] = Max [i, j] – Allocation [i, j]

Allocation$_i$ specifies the resources currently allocated to process P_i and Need$_i$ specifies the additional resources that process P_i may still request to complete its task.

3. Algorithm:

Steps of Algorithm:

```
1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
   Initialize: Work= Available
   Finish [i]=false; for i=1,2,……,n

2. Find an i such that both
   a) Finish [i]=false
   b) Need_i<=work

   if no such i exists goto step (4)
3. Work=Work + Allocation_i
   Finish[i]= true
   goto step(2)
4. If Finish[i]=true for all i,
   then the system is in safe state.
```

For example,

Available			Processes	Allocation			Max		
A	B	C		A	B	C	A	B	C
3	3	2	P0	0	1	0	7	5	3
			P1	2	0	0	3	2	2
			P2	3	0	2	9	0	2
			P3	2	1	1	2	2	2
			P4	0	0	2	4	3	3

```
        3 3 2          //Available
P1 – 2 0 0
     ______
        5 3 2
P4 – 0 0 2
     ______
        5 3 4
P3 – 2 1 1
     ______
        7 4 5
     Deadlock Detected

        3 3 2          //Available
P3 – 2 1 1
     ______
        5 3 2
P1 – 2 0 0
     ______
        5 3 4
P4 – 0 0 2
     ______
        7 4 5
     Deadlock Detected
```

4. Purpose of use:

Banker's algorithm is a deadlock avoidance algorithm. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Every time a loan has to be granted by the bank; it subtracts the loan amount from the total money the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Banker's algorithm works in a similar way in computers.

Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

5. Code snippet:

```
#include<iostream>

using namespace std;

// Number of processes

const int P = 5;

// Number of resources

const int R = 3;

// Function to find the need of each process

void calculateNeed(int need[P][R], int maxm[P][R],

              int allot[P][R]) {

    // Calculating Need of each P

    for (int i = 0 ; i < P ; i++)

        for (int j = 0 ; j < R ; j++)

            // Need of instance = maxm instance -

            //allocated instance

            need[i][j] = maxm[i][j] - allot[i][j]; }

// Function to find the system is in safe state or not

bool isSafe(int processes[], int avail[], int maxm[][R],

            int allot[][R]) {

    int need[P][R];

    // Function to calculate need matrix

    calculateNeed(need, maxm, allot);
```

```
// Mark all processes as infinish

bool finish[P] = {0};

// To store safe sequence

int safeSeq[P];

// Make a copy of available resources

int work[R];

for (int i = 0; i < R ; i++)

    work[i] = avail[i];

// While all processes are not finished

// or system is not in safe state.

int count = 0;

while (count < P) {

    // Find a process which is not finish and

    // whose needs can be satisfied with current

    // work[] resources.

    bool found = false;

    for (int p = 0; p < P; p++) {

        // First check if a process is finished,

        // if no, go for next condition

        if (finish[p] == 0) {

            // Check if for all resources of

            // current P need is less

            // than work

            int j;
            for (j = 0; j < R; j++)
                if (need[p][j] > work[j])
                    break;
            // If all needs of p were satisfied.
            if (j == R) {
                // Add the allocated resources of
                // current P to the available/work
                // resources i.e.free the resources
                for (int k = 0 ; k < R ; k++)
                    work[k] += allot[p][k];
                // Add this process to safe sequence.
                safeSeq[count++] = p;
                // Mark this p as finished
                finish[p] = 1;
                found = true; } } }
        // If we could not find a next process in safe
        // sequence.
        if (found == false){
            cout << "System is not in safe state";
            return false; } }
    // If system is in safe state then
    // safe sequence will be as below
    cout << "System is in safe state.\nSafe"
         " sequence is: ";
    for (int i = 0; i < P ; i++)

        cout << safeSeq[i] << " ";

    return true;}

// Driver code

int main() {

    int processes[] = {0, 1, 2, 3, 4};

    // Available instances of resources

    int avail[] = {3, 3, 2};

    // Maximum R that can be allocated

    // to processes

    int maxm[][R] = {{7, 5, 3},

                     {3, 2, 2},

                     {9, 0, 2},

                     {2, 2, 2},

                     {4, 3, 3}};

    // Resources allocated to processes

    int allot[][R] = {{0, 1, 0},

                      {2, 0, 0},

                      {3, 0, 2},

                      {2, 1, 1},

                      {0, 0, 2}};

    // Check system is in safe state or not

    isSafe(processes, avail, maxm, allot);

    return 0;}
```

<u>Output:</u>

```
System is in safe state.

Safe sequence is: 1 3 4 0 2
```

6. Description (Example):

Considering a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Available			Processes	Allocation			Max		
A	B	C		A	B	C	A	B	C
3	3	2	P0	0	1	0	7	5	3
			P1	2	0	0	3	2	2
			P2	3	0	2	9	0	2
			P3	2	1	1	2	2	2
			P4	0	0	2	4	3	3

Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2 > satisfies safety requirement.

Time complexity = O(n*n*m) where n = number of processes and m = number of resources.

GitHub Link: https://github.com/khatribhagesh1122