**Name** – Ashutosh Pratap Rao

**Registration No** – 11611957

**Email** – [ashutoshprataprao@gmail.com](mailto:ashutoshprataprao@gmail.com)

**Github Link** - <https://github.com/ashutoshprataprao>

**Code**  -

|  |
| --- |
|  |
|  |
|  | #include<stdio.h>  #include<pthread.h> |
|  | #include<unistd.h> |
|  | #include<stdbool.h> |
|  | #include<stdlib.h> |
|  |  |
|  | int i,j,k; //Global variable for loops |
|  |  |
|  | int processes,resource; |
|  |  |
|  | int safe[10]; //To store the safe sequence |
|  | bool completed[10]; //To check whether a process is completed or not |
|  | int avail[10]; //available array |
|  | int alloc[10][10]; //allocation matrix for checking how many resources alloacted to a process |
|  | int need[10][10]; //need matrix for checking how many resources need to a process |
|  | int max[10][10]; //Max matrix to check total number of resources required to a process |
|  |  |
|  | //Mutex lock for avail and allocation |
|  | pthread\_mutex\_t mut\_available; |
|  | pthread\_mutex\_t mut\_allocation; |
|  | pthread\_mutex\_t lock; |
|  |  |
|  | bool fun\_allocation(int process,int \*request); |
|  |  |
|  |  |
|  | // Method for displaying status of avail,need,allocate and max. |
|  | void print() |
|  | { |
|  | //display status of available. |
|  | printf("Status of avail:\n"); |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | printf("\tR%d",i); |
|  | } |
|  | printf("\n"); |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | printf("\t%d",avail[i]); |
|  | } |
|  |  |
|  | //display status of allocation |
|  | printf("\nStatus of allocation:\n"); |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | printf("\tR%d",i); |
|  | } |
|  | printf("\n"); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  |  |
|  | printf("P%d\t",i); |
|  | for(j=0;j<resource;j++) |
|  | { |
|  | printf("%d\t",alloc[i][j]); |
|  | } |
|  | printf("\n"); |
|  |  |
|  | } |
|  |  |
|  | //display status of max. |
|  | printf("Status of max:\n"); |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | printf("\tR%d",i); |
|  | } |
|  | printf("\n"); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  |  |
|  | printf("P%d\t",i); |
|  | for(j=0;j<resource;j++) |
|  | { |
|  | printf("%d\t",max[i][j]); |
|  | } |
|  | printf("\n"); |
|  |  |
|  | } |
|  |  |
|  | //display status of need. |
|  | printf("Status of need:\n"); |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | printf("\tR%d",i); |
|  | } |
|  | printf("\n"); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | printf("P%d\t",i); |
|  | for(j=0;j<resource;j++) |
|  | { |
|  | need[i][j]=max[i][j]-alloc[i][j]; |
|  | printf("%d\t",need[i][j]); |
|  | } |
|  | printf("\n"); |
|  |  |
|  | } |
|  | } |
|  |  |
|  |  |
|  | /\*\* |
|  | Method for Bankers algorithm which check safe and unsafe state. |
|  | It return true if system is in safe state, false if unsafe state. |
|  | \*\*/ |
|  |  |
|  | bool Bankers() |
|  | { |
|  | int task[resource]; //A duplicate array for avail. |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | completed[i]=false; |
|  | } |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | task[i]=avail[i]; |
|  | } |
|  | int count1=0; |
|  | while(count1<processes) |
|  | { |
|  | bool found=false; |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | if(completed[i]==false) |
|  | { |
|  | for(j=0;j<resource;j++) |
|  | { |
|  | if(need[i][j]>task[j]) |
|  | { |
|  | break; |
|  | } |
|  | } |
|  | if(j==resource) |
|  | { |
|  | for(k=0;k<resource;k++) |
|  | { |
|  | task[k]=task[k]+alloc[i][k]; |
|  |  |
|  | } |
|  | safe[count1++]=i; |
|  | completed[i]=true; |
|  | found=true; |
|  | } |
|  | } |
|  |  |
|  | } |
|  | if(found==false) |
|  | { |
|  | printf("System is in unsafe state\n"); |
|  | return false; |
|  | } |
|  | } |
|  | printf("Your system is in safe state and safe sequence is\n"); |
|  | printf("<"); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | printf("P%d ",safe[i]); |
|  | } |
|  | printf(">"); |
|  | printf("\n\n"); |
|  | return true; |
|  | } |
|  |  |
|  |  |
|  | //Method for requesting the resources. |
|  | void \*request\_resource(void\* p) |
|  | { |
|  | pthread\_mutex\_lock(&lock); |
|  | int process\_id=(int)p; |
|  | sleep(1); |
|  | int request[resource]; |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | printf("Enter number of instance required for resource R%d: ",i); |
|  | scanf("%d",&request[i]); |
|  | } |
|  |  |
|  | if(fun\_allocation(process\_id,request)==true) |
|  | { |
|  | printf(" Approved\n"); |
|  | sleep(1); |
|  | } |
|  | else |
|  | { |
|  | printf(" Denied\n"); |
|  | sleep(1); |
|  | } |
|  | pthread\_mutex\_unlock(&lock); |
|  | pthread\_exit(NULL); |
|  | } |
|  |  |
|  | bool fun\_allocation(int process,int request[resource]) |
|  | { |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | pthread\_mutex\_lock(&mut\_allocation); |
|  | alloc[process][i]=alloc[process][i]+request[i]; |
|  | pthread\_mutex\_unlock(&mut\_allocation); |
|  |  |
|  | pthread\_mutex\_lock(&mut\_available); |
|  | avail[i]=avail[i]-request[i]; |
|  | pthread\_mutex\_unlock(&mut\_available); |
|  | } |
|  | if(Bankers()==false) |
|  | { |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | pthread\_mutex\_lock(&mut\_allocation); |
|  | alloc[process][i]=alloc[process][i]-request[i]; |
|  | pthread\_mutex\_unlock(&mut\_allocation); |
|  |  |
|  | pthread\_mutex\_lock(&mut\_available); |
|  | avail[i]=avail[i]+request[i]; |
|  | pthread\_mutex\_unlock(&mut\_available); |
|  | } |
|  | return false; |
|  | } |
|  | return true; |
|  | } |
|  |  |
|  | int main() |
|  | { |
|  | pthread\_mutex\_init(&mut\_available,NULL); |
|  | pthread\_mutex\_init(&mut\_allocation,NULL); |
|  | pthread\_mutex\_init(&lock,NULL); |
|  | printf("\nEnter number of processes: "); |
|  | scanf("%d",&processes); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | completed[i]=false; |
|  | } |
|  | printf("\nEnter number of resources: "); |
|  | scanf("%d",&resource); |
|  | printf("\nEnter max matrix: "); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | for(j=0;j<resource;j++) |
|  | { |
|  | scanf("%d",&max[i][j]); |
|  | } |
|  | } |
|  | printf("\nEnter allocation matrix: "); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | for(j=0;j<resource;j++) |
|  | { |
|  | scanf("%d",&alloc[i][j]); |
|  | } |
|  | } |
|  | printf("\nEnter available resource: "); |
|  | for(i=0;i<resource;i++) |
|  | { |
|  | scanf("%d",&avail[i]); |
|  | } |
|  | printf("\nBefore allocation of resources value of need, max, allocate and available:\n"); |
|  | print(); |
|  | Bankers(); |
|  | pthread\_t Threads[processes]; |
|  | int res; |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | res=pthread\_create(&Threads[i],NULL,request\_resource,(void \*)i); |
|  | } |
|  | if(res!=0) |
|  | { |
|  | printf("\nError occurs, with value %d\n",res); |
|  | exit(-1); |
|  | } |
|  | printf("Thread created successfully.\n"); |
|  | for(i=0;i<processes;i++) |
|  | { |
|  | pthread\_join(Threads[i],NULL); |
|  | } |
|  | return 0; |
|  | } |

1. **Description** –

The Bankers Algorithm, sometimes refered as detection algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources and then makes an “s-state” check to test for possible deadlock conditions for all other pending processes, before deciding whether allocation should be allowed to continue.

The Bankers algorithm is run by the operating system whenever a process request resources, the algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in unsafe state.

There are p number of processes and r number of resources in the problem. Mutiple threads will be created through which these process will put request for the resources and resource will be allocated to the these processes if after allocating the resources to these processes it leaves the system in safe sequence. We use here mutex lock so that only one processe can request and grant the resource at the time.

1. **Algorithm** –

It consist of two algorithm.

1.Safety Algorithm

2.Resource Allocation Algorithm

1. Safety Algorithm

Let task and completed is two array of size numberofresorece and numberofprocess

1. task = available; //Copy the value of availabele into task array.
2. Repeat completed[i] = false for i=1,2,3,4…..numberofprocess
3. if completed[i] = = false and need[i][j]<= task[j]
4. goto step 5
5. task = task + allocation
6. completed[i]=true;
7. goto step 3.
8. if completed[i]==true for all i

then system is in safe state.

1. Resource Allocation Algorithm –

Let request[resource] be the request array for process[i]. request[j] = m means process p[i] wants m instance of type R[j]. when a request for resource is made following activity are involved.

1. if request[i] <= need[i][j]

goto step 2;

1. if request[i] < = available

goto step 3;

1. Have the system pretend to have allocated the requested resource to process p[i] by modifying the state as

available = available – request;

allocation = allocation + request;

need = need – request;

1. **Complexity of Algorithm** –

Complexity of print method = O(n\*m) //where n is no. of process and m is resources

Complexity of Bankers method = O(n^2\*m)

Complexity of fun\_allocation method = O(m)

Overall complexity of program is O(n^2\*m).

At the time when n=m then

Overall complexity of program will be O(n^3).

1. **Constraints** –

**Threads** – Thread is an execution unit which consist of its own program counter, a stack and a set of registers. Threads are also know as lightweight processes.

pthread\_t Threads[processes];

int res;

for(i=0;i<processes;i++)

{

res=pthread\_create(&Threads[i],NULL,request\_resource,(void \*)i);

}

**Mutex** - A mutex provide mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

pthread\_mutex\_t mut\_available;

pthread\_mutex\_t mut\_allocation;

pthread\_mutex\_t lock;

pthread\_mutex\_init(&mut\_available,NULL);

pthread\_mutex\_init(&mut\_allocation,NULL);

pthread\_mutex\_init(&lock,NULL);

|  |
| --- |
|  |
|  |  |
|  |  |

**Safe State -** A state is safe if the system can allocate all resource requested by all the processes without entering a deadlock state.

printf("Your system is in safe state and safe sequence is\n");

printf("<");

for(i=0;i<processes;i++)

{

printf("P%d ",safe[i]);

}

printf(">");

1. **Boundary Conditions –**
2. System grants request only if the request will result in a safe state.
3. If number of process is not fixed then it hard to apply which means number of process should be fixed.
4. It allows all requests to be granted in finite amount of time, but one year is also a finite amount of time which means if process takes too much time to grant their request then it should not be apply.
5. **Test Cases –**

Enter number of processes: 5

Enter number of resource: 3

Enter max matrix: 7 5 3 3 2 2 9 0 2 2 2 2 4 3 3

Enter allocation matrix: 0 1 0 2 0 0 3 0 2 2 1 1 0 0 2

Enter available resource: 3 3 2

Status of available

|  |  |  |
| --- | --- | --- |
| R0 | R1 | R2 |
| 3 | 3 | 2 |

Status of allocation

|  |  |  |  |
| --- | --- | --- | --- |
|  | **R0** | **R1** | **R2** |
| **P0** | **0** | **1** | **0** |
| **P1** | **2** | **0** | **0** |
| **P2** | **3** | **0** | **2** |
| **P3** | **2** | **1** | **1** |
| **P4** | **0** | **0** | **2** |

Status of max

|  |  |  |  |
| --- | --- | --- | --- |
|  | **R0** | **R1** | **R2** |
| **P0** | **7** | **5** | **3** |
| **P1** | **3** | **2** | **2** |
| **P2** | **9** | **0** | **2** |
| **P3** | **2** | **2** | **2** |
| **P4** | **4** | **3** | **3** |

Status of need

|  |  |  |  |
| --- | --- | --- | --- |
|  | **R0** | **R1** | **R2** |
| **P0** | **7** | **4** | **3** |
| **P1** | **1** | **2** | **2** |
| **P2** | **6** | **0** | **0** |
| **P3** | **0** | **1** | **1** |
| **P4** | **4** | **3** | **1** |

Your system is in safe state and safe sequence is:

<P1, P3, P4, P0, P2>

**Explanation –**

p=5,r=3

task= 3,3,2

completed= false,false,false,false,false

for i=0

need= 7,4,3

here need > task

so P0 must wait

for i=1

need=1,2 ,2

here need < task

so P1 put in the safe[i]

task=task+allocation

task=5,3,2

completed= false, true, false, false, false

for i=2

need = 6,0,0

here need>task

so P2 must wait

for i=3

need= 0,1,1

here need < task

so P3 put in the safe[i]

task=task+allocation

task=7,4,3

completed=false,true,false,true,false

.

.

.

.

Same as above calculation we calculate for all remaining processes and at the end we get our safe sequence as

<P1,P3,P4,P0,P2>

1. **Have you made minimum 5 revisions of solution on GitHub?**

**Yes.**

<https://github.com/ashutoshprataprao>