```
#include <iostream>
#include <cstdlib>
#include <stdlib.h>
#include <stdio.h>
#include <cstring>
#include <string>
#include <sstream>
#include <fstream>
#include <algorithm>
#include <math.h>
using namespace std;
//                                    Banker's Algorithm of allocating resources
//

const size_t MAX_TASKS=20;
const size_t MAX_RES=30;
enum command_t {Initiate, Request, Release, Compute, Terminate}; //Posible commands
enum task_state {blocked, running, aborted};
struct res    {//Resources kept here
               int r_type;
                  int units;
                  } resources[MAX_RES];
struct comm   {//List of allocation commands
               int task_id;
                  command_t cmd;
                  int r_type;
                  int units;
                  } commlist[MAX_RES];

struct tsk  {   //Aray to keep basic info about the tasks
              int           task_id;
                 task_state    state;
                 struct tskact *taskp;
           } task_array[MAX_TASKS];

struct tskact { //This structure keeps the activities for each task
              command_t cmd;
                 int       r_type;
                 int       units;
                 struct tskact *next;   //It is kept as FIFO linked list
                 } task_activity;

struct stats { //Here we keep statistics about tasks, to generate the output
              char * status;
                 int    time;
                 int    wait;
               } t_stats[MAX_TASKS];
```

```
int curr[MAX_TASKS][MAX_RES];       //Current allocation of resources
int maxalc[MAX_TASKS][MAX_RES];     //Maximum claims
int save_curr[MAX_TASKS][MAX_RES];  //To facilitate safety check
int release_curr[MAX_TASKS][MAX_RES]; //To facilitate provisional release

int avail[MAX_RES];
int save_avail[MAX_RES];
int provisional_avail[MAX_RES];    //"in cycle" release of resources

//FUNCTIONS PROTOTYPES

void print_c(struct comm [],int);
void init_storage(int,int);
bool safe(int, int);
void print_matrix(int [][MAX_RES],int, int);
void backup_state(int, int);
void restore_state(int, int);
void print_array(struct tsk [], int);
void print_stats(struct stats [], int);
void update_provisional(int [], int [],int);
bool exceed(int, int);
void init_stats(struct stats [], int);
void grant(int, int, int);
void sort_task(struct tsk [],int);
void init_provisional(int [], int);
void init_matrix(int [][MAX_RES], int, int);
void update_curr(int [][MAX_RES], int, int);

//------------------------------------------------------ MAIN

int main (int argc, char * const argv[]) {

char *in_file;
in_file=argv[1];
fstream in(in_file, ios::in);
if (!in){ cout << "\n Input cannot be opened" <<
                          "\n Please check file: " << in_file << "\n";
    exit(1);
}


char tk[10];
int t1, r1;  //t1 - number of tasks, r1 - number of resources
int c_i=1; int task, resource, u, cyc;
```

```
//Reading the resources pool

in >> tk;
t1=atoi(tk); cout << "\n T=" << t1 << "\n";
in >> tk;
r1=atoi(tk); //# resource types
cout << "\n R=" << r1 << "\n";
for(int i=1; i<=r1;++i){
   in >> tk; u=atoi(tk);
   resources[i].r_type=i;
   resources[i].units=u;
   //cout <<"\n Resource type "  << resources[i].r_type;
   //cout <<"\n  units=" << resources[i].units;
}


//Reading the allocation instructions

char comm_in[10];
in >> tk; strcpy(comm_in,tk);
while(in) {
     //The 3rd character in comm_in, identifies the command
        char comm_id = *(comm_in+2); //cout <<"\n comm_id=" << comm_id;
        //bexit(1);
        switch(comm_id){
              case 'i': //Initiate
                    commlist[c_i].cmd=Initiate;
                              in >> task; in >> resource; in >> u;
                              commlist[c_i].task_id=task;
                              commlist[c_i].r_type=resource;
                              commlist[c_i].units=u;
                              break;
           case 'q': //Request
                    commlist[c_i].cmd=Request;
                              in >> task; in >> resource; in >> u;
                              commlist[c_i].task_id=task;
                              commlist[c_i].r_type=resource;
                              commlist[c_i].units=u;
                              break;
           case 'l': //Release
                    commlist[c_i].cmd=Release;
                              in >> task; in >> resource; in >> u;
                              commlist[c_i].task_id=task;
                              commlist[c_i].r_type=resource;
                              commlist[c_i].units=u;
                              break;
          case 'm': //Compute
```

```
                    commlist[c_i].cmd=Compute;
                              in >> task; in >> cyc;
                              commlist[c_i].task_id=task;
                              commlist[c_i].r_type=0; //CPU is the resource
                              commlist[c_i].units=cyc;
                              break;
            case 'r': //Terminate
                    commlist[c_i].cmd=Terminate;
                              in >> task;
                              commlist[c_i].task_id=task;
                              commlist[c_i].r_type=0;
                              commlist[c_i].units=-1;
                              break;
            default  : cout << "\n char is: " << comm_id;
                       cout << "\ninvalid command encountered. Terminates.\n";
        } //switch
        ++c_i;
        in >> comm_in;
 } //While input
//End of input processing

//Build Available resource array (initial availability of course)
for (int i=1;i<=r1;++i){
    avail[i]=resources[i].units;
}

//Initialize maxalc, curr alocation matrices
init_storage(t1,r1);

/*Build matrix of maximum allocation requested by all tasks for all resources
for (int i=1;i<c_i;++i){
   if (commlist[i].cmd==Initiate){
       maxalc[commlist[i].task_id][commlist[i].r_type]+=commlist[i].units;
   }
}
*/
cout << "\n Command list:\n";
print_c(commlist,c_i);
//cout "\nCurr allocation:\n";
//print_matrix(curr,t1,r1);
//cout << "\nMAX allocation:\n";
//print_matrix(maxalc,t1,r1);

/*************************************************************************
* Built the construct of array of pointers, each of which belongs to one task *
* and points to a linked list that contains all the activities we need to     *
* perform on those tasks.                                                     *
```

```
/*****************************************************************************/
int ind; //struct tskact *p;
for (int i=1; i<t1; ++i){          //Initialize array of pointers to activities
    task_array[i].taskp = NULL;
      task_array[i].task_id=0;
}

//cout << "\n BUILDING THE FIFO\n";
for (int l=1; l<c_i; ++l){
    ind=commlist[l].task_id;

      if (task_array[ind].task_id==0){      //First activity for the task, create entry
          task_array[ind].task_id=ind;      //Initially the array's index=task ids
          task_array[ind].state=running;    //zero excluded
      }
      if ( task_array[ind].taskp==NULL){
       task_array[ind].taskp=(struct tskact *) malloc(sizeof(struct tskact));
          task_array[ind].taskp->cmd     =commlist[l].cmd;
          task_array[ind].taskp->r_type  =commlist[l].r_type;
          task_array[ind].taskp->units   =commlist[l].units;
          task_array[ind].taskp->next    =NULL;
      }

    else{ // The array entry already has an activity(ies) add more
          struct tskact *q; struct tskact *t;
       q=task_array[ind].taskp;
       while (q->next !=NULL) q=q->next;

       t=(struct tskact *) malloc(sizeof(struct tskact));
          t->cmd     =commlist[l].cmd;
          t->r_type  =commlist[l].r_type;
          t->units   =commlist[l].units;
          t->next    =NULL;
       q->next    = t;

      }
}

/* DEBUG
    cout << "\n TASK ARRAY";
      struct tskact *p; char *desc;
    for (int z=1;z<=t1;++z){
         p=task_array[z].taskp;
         while (p!=NULL){
               switch(p->cmd){
                           case Initiate:  desc="initiate";   break;
                               case Request:   desc="request";    break;
```

```
                                case Release:   desc="release";    break;
                                case Compute:   desc="compute";    break;
                                case Terminate: desc="terminate";  break;
                                       default: cout << "invalid command";
                }

               cout <<"\n Element:" << z << "  Command=" << desc << "  R TYPE=" << p->r_type << " UNITS=" << p->units;
               p=p->next;
          }
       }
 END DEBUG */

//PROGRAM EXECUTION
int tasknum; struct tskact *activity;
bool all_tasks_terminated=false;  //We assume that some tasks are ready to run
int cycle=0; bool ac_complete;    //Check if activity was complete

//DEBUG
cout << "\nResource available before run:\n";
for (int i=1;i<=r1;++i){
    cout << "\n resource: " << i << " Units=" << avail[i] << "\n";
}

init_stats(t_stats,t1); int sum_t; int i1;
init_provisional(provisional_avail,r1);     //Empty provisional release array
init_matrix(release_curr,t1,r1);            //and image of current alocation
while (!all_tasks_terminated){              //Not really necessary here, but keeping good form
    cout<< "\n\n ARRAY_TASK before cycle:" << cycle;
       print_array(task_array, t1);
       cout << "\nCurr allocation:\n";
       print_matrix(curr,t1,r1);

        //tc - is Task Counter
        //tc may not corrspond to the task number because
        //we sort task_array, so "blocked" are treated first

       sum_t=0;
    for (int tc=1;tc<=t1;++tc){
         if (task_array[tc].task_id==0) break;        //Meaning task terminated, go to next task
         tasknum=task_array[tc].task_id;
         if (task_array[tc].taskp==NULL){
            task_array[tc].task_id=0;
        break;
         }
         //Debug
          cout << "\n\n TASK ID EXAMINED: " << task_array[tc].task_id << "\n\n";
```

```
activity=task_array[tc].taskp; ac_complete=true;
//DEBUG
char *ac_desc;
switch(activity->cmd){
                    case Initiate:  ac_desc="initiate";   break;
                        case Request:   ac_desc="request";    break;
                        case Release:   ac_desc="release";    break;
                        case Compute:   ac_desc="compute";    break;
                        case Terminate: ac_desc="terminate";  break;
                               default: cout << "invalid command";
}

cout<< "\n\n Checking, task: " << tasknum << " activity:" << ac_desc << " Resource T:" <<
    activity->r_type << "  units:" << activity->units;
cout << "\nResource available, before SWITCH:\n";
for (int i=1;i<=r1;++i){
     cout << "\n resource: " << i << " Units=" << avail[i] << "\n";
}


switch (activity->cmd){
     case Initiate:
           /*********************************************************************************
           * During the course of initiation(which is done before requests), the avail array*
              * still holds the initial availability, which is known in advance from the input *
              * and therefore allows us to check that initiate does not exceed availability    *
              *********************************************************************************/
              if (exceed(activity->r_type,activity->units)){ //ERROR DETECTION
                 cout << "\n Banker aborts task: " << tasknum << " before execution";
                 cout << "\n    Initial claim of task: " << tasknum << " is: " <<
                          activity->units << " units of resource type: " << activity->r_type <<
                              " which exceeds maximum of: " << avail[activity->r_type] << " available";
                     task_array[tc].task_id=0;
                     task_array[tc].state=aborted;
                     t_stats[tasknum].status="aborted";
                     //ABORT
              }
              else {
                maxalc[tasknum][activity->r_type]+=activity->units;
              }
              cout <<"\n MAX allocation after this initiate:\n";
              print_matrix(maxalc,t1,r1);
              cout << "\nCurr allocation, after initiate:\n";
              print_matrix(curr,t1,r1);
              cout << "\nResource available, after initiate:\n";
              for (int i=1;i<=r1;++i){
```

```
                    cout << "\n resource: " << i << " Units=" << avail[i] << "\n";
            }

            break;
case Compute:
          --activity->units;
            if (activity->units >0) ac_complete=false;   //Incomplete activity
          break;
    case Release:
            provisional_avail[activity->r_type]+=activity->units;
            release_curr[tasknum][activity->r_type]+=activity->units;
            /*debug*/ cout << "\n RELEASE Curr allocation:\n";
               print_matrix(release_curr,t1,r1);

            break;
    case Terminate:
          task_array[tc].task_id=0;
            t_stats[tasknum].status="ended";
            for(i1=1;i1<=r1;++i1) maxalc[tasknum][i1]=0; //This task no onger claims resources
          break;
    case Request: // if Request for resource +in-use axceed maximum pledged - abort task
          if (maxalc[tasknum][activity->r_type]<
                activity->units +curr[tasknum][activity->r_type]){
                   task_array[tc].task_id=0;
                   task_array[tc].state=aborted;
                   t_stats[tasknum].status="aborted";
                   ac_complete=false;
                   cout<< "\nTask number: " << tasknum << " asked for:" << activity->units <<
                        " of resource type:" <<activity->r_type << " and has already:" <<
                           curr[tasknum][activity->r_type] << " units of it. It originally asked for:"
                           << maxalc[tasknum][activity->r_type] << " maximum";
            }
            else {
              backup_state(t1,r1); //Save state in case it would be unsafe
                          //Firstly, grant the request temporarily
                          grant(tasknum,activity->r_type, activity->units);

                          //DEBUG
                          cout << "\n CALLING SAFE with:" << " t1=" << t1 << "  r1=" << r1;
                          cout << "\nResource available after temp GRANT and before SAFE:\n";
           for (int i=1;i<=r1;++i){
               cout << "\n resource: " << i << " Units=" << avail[i] << "\n";
                          }
                          cout << "\nCurr allocation:\n";
           print_matrix(curr,t1,r1);
           cout << "\nMAX allocation:\n";
           print_matrix(maxalc,t1,r1);
```

```
                //END DEBUG


                        if (safe(t1, r1)){      //contingent safety
                           restore_state(t1,r1);
                                  grant(tasknum,activity->r_type, activity->units);  //Now grant it for real
                                  task_array[tc].state=running;
                                  cout << "\nREQUEST GRANTED\n";

                        }
                        else { //Request denied, task blocked (unsafe)
                                  cout << "\nREQUEST NOT GRANTED\n";
                                  restore_state(t1,r1);
                                  ac_complete=false;  //Incomplete activity
                             t_stats[tasknum].wait++;
                             task_array[tc].state=blocked;
                        }
                    }
               break;
    default: cout << "\n Other UNKNOWN command";
               break;
  }
  if (ac_complete){ //Incomplete activity is compute that hasn't ended yet, or request denied
      task_array[tc].taskp=task_array[tc].taskp->next;    //Drop the activity after its complete
                                                          //will become NULL if activity is Terminate
  }
  if ( task_array[tc].task_id > 0)                        //one more cycle added to task's time
     t_stats[tasknum].time++;

  //free(activity);
}
update_provisional(avail,provisional_avail,r1);          //Update resource vetor and current allocation
update_curr(release_curr, t1, r1);                        //matrix with resources released in the last cycle
init_provisional(provisional_avail,r1);                   //Initialize the temp storage for next cycle
init_matrix(release_curr,t1,r1);                          //both current allocation and avail resource vetor
++cycle;
sort_task(task_array,t1);  //Sorting task array, so the blocked tasks are ahead of others
for(int l1=1;l1<=t1;++l1){
   sum_t+=task_array[l1].task_id;
}
if (sum_t==0)  all_tasks_terminated=true; sum_t=0;

// DEBUG
if (cycle >22){
   all_tasks_terminated=true;
}
//
```

```
}

//DEBUG
cout << "\nResource available AFTER run:\n";
for (int i=1;i<=r1;++i){
    cout << "\n resource: " << i << " Units=" << avail[i];
}



cout << "\nMAX allocation:\n";
print_matrix(maxalc,t1,r1);
//END DEBUG
print_stats(t_stats,t1);



return 0;
}


//Functions
//-------------------------------------------------------------------------------------------------------------
void print_c(struct comm q[], int len){
   char *comm_desc;
   for (int i=1; i<len;++i){
       switch(q[i].cmd){
                              case Initiate:  comm_desc="initiate";   break;
                                  case Request:   comm_desc="request";    break;
                                  case Release:   comm_desc="release";    break;
                                  case Compute:   comm_desc="compute";    break;
                                  case Terminate: comm_desc="terminate";  break;
                                         default: cout << "invalid command";
         }
         cout <<"\n" << comm_desc << " " << q[i].task_id << "  " << q[i].r_type <<
              "  " << q[i].units;
         cout << "\n";
   }

}
//-------------------------------------------------------------------------------------------------------------
bool safe(int running_tasks, int resources){
```

```
     bool safe; int unfinished=running_tasks; int i, j;
        while (unfinished > 0){
            i=1; safe=false;
         while (i<= running_tasks && !safe){  //Assume it's not safe and look for a task that can complete
              if (curr[i][1] != -1){          // I mark tasks that can complete with -1
                    safe=true;
                        j=1;
                        while (j<= resources) {      // Check if we can satisfy all resources
                            if ((maxalc[i][j]-curr[i][j]) > avail[j]){
                                   safe=false;           //It is enough that we faile once, to try next task
                                      break;
                               }
                               ++j;
                        }
                  }
               ++i;
         }
         if (safe){
            --unfinished;                     //One less task to check
                --i;
               for (j=1;j<=resources;++j){      //Release the resources back to the pool
                   avail[j]+=curr[i][j];
               }
               curr[i][1]=-1;                    //Mark the task as complete for now
         }
         else{
             return false;                    //It was unsafe (no task can complete)
         }
        }
        return true;  //If we made it here, it means all tasks can complete (no unfinished) ==> SAFE

}
//-----------------------------------------------------------------------------------------------------------------
void init_storage(int tasks, int resources){
   for (int i=1;i<=tasks;++i){
       for (int j=1;j<=resources;++j){
            maxalc[i][j]=0; curr[i][j]=0;
         }
   }
}
//-----------------------------------------------------------------------------------------------------------------
void init_matrix(int m[][MAX_RES], int t, int r){
   for (int i=1;i<=t;++i){
       for (int j=1;j<=r;++j){
            m[i][j]=0;
         }
   }
```

```
}

//-------------------------------------------------------------------------------------------------------------------
void init_stats(struct stats s[], int t){
   for (int i=1;i<=t;++i){
       s[i].status="";
          s[i].time=0;
          s[i].wait=0;
   }
}

//-------------------------------------------------------------------------------------------------------------------
void print_matrix(int matrix[][MAX_RES],int t, int r){
   for (int i=1;i<=t;++i){
       cout << "\Task: " << i << "  ";
       for (int j=1;j<=r;++j){
          cout << "Resource type=" << j << " Units allocated=" << matrix[i][j] << "  ";
          }
          cout << "\n";
   }
}
//-------------------------------------------------------------------------------------------------------------------
void backup_state(int t, int r){
    int k1, k2;
      for (k1=1;k1<=t;++k1){
          for (k2=1;k2<=r;++k2){
           save_curr[k1][k2]=curr[k1][k2];
          }
      }
      for (k2=1;k2<=r;++k2){
         save_avail[k2]=avail[k2];
      }
}
//-------------------------------------------------------------------------------------------------------------------
void restore_state(int t, int r){
    int k1, k2;
      for (k1=1;k1<=t;++k1){
          for (k2=1;k2<=r;++k2){
           curr[k1][k2]=save_curr[k1][k2];
          }
      }
      for (k2=1;k2<=r;++k2){
         avail[k2]=save_avail[k2];
      }
}
//-------------------------------------------------------------------------------------------------------------------
void update_provisional(int r1[], int r2[], int r){ //adding resources from r2 array to r1 array
```

```
    int k;
       for (k=1;k<=r;++k){
          r1[k]+=r2[k];
       }
}
//-------------------------------------------------------------------------------------------------------------------
void update_curr(int c[][MAX_RES], int t, int r){ //After "release", the current allocation is reduced
    int k1, k2;
       for (k1=1;k1<=t;++k1){
           for (k2=1;k2<=r;++k2){
            curr[k1][k2]-=c[k1][k2];
           }
       }
}

//-------------------------------------------------------------------------------------------------------------------

void print_array(struct tsk a[], int elements){
    char *desc; struct tskact *p;
    for (int i=1; i<=elements;++i){
          if (a[i].state==blocked)  desc="blocked";
       if (a[i].state==aborted)  desc="aborted";
       if (a[i].state==running)  desc="running";

          cout << "\n Task id=" << a[i].task_id << " state= " << desc;
       p=a[i].taskp;

       if (p!=NULL){
            switch(p->cmd){
                                    case Initiate:  desc="initiate";   break;
                                    case Request:   desc="request";    break;
                                    case Release:   desc="release";    break;
                                    case Compute:   desc="compute";    break;
                                    case Terminate: desc="terminate";  break;
                                    default: cout << "invalid command";
               }

            cout << "\n                              Activity:" << desc;
            cout << "\n                              Resource:" << p->r_type;
            cout << "\n                              Units   :" << p->units;
       }
       else {
            cout << "\n NO MORE ACTIVITIES FOR THIS TASK\n";
       }
       /*
       while (p!=NULL){
           cout << "                              Activity:" << p->cmd;
```

```
                p=p->next;
        }
        */
       }
}
//-------------------------------------------------------------------------------------------------------------------
void print_stats(struct stats s[], int t){
   double percent_w; double wait;
   double time; double tot_time=0; double tot_wait=0; double percent_av;
   for (int i=1;i<=t;++i){
      wait= (double) s[i].wait;
         time = (double) s[i].time;
      if (time > 0) percent_w = (wait/time)*100;
         else          percent_w = 0;
      cout << "\n Task number: " << i;
         if ( !strcmp(s[i].status,"aborted")){ //Task was aborted
             cout << "      aborted";
         }
         else{
            cout << "   " << s[i].time << "  " << s[i].wait <<
         "  " << percent_w << "%";
         tot_time+=s[i].time; tot_wait+=s[i].wait;
         }
   }
   if (tot_time > 0) percent_av = (tot_wait/tot_time)*100;
   else              percent_av = 0;
   cout << "\n Total:          " << tot_time << "  " << tot_wait << "  " << percent_av<<"%\n";
}

//-------------------------------------------------------------------------------------------------------------------
bool exceed(int r, int u){
     if (avail[r]< u)
           return true;
        else
           return false;
}
//-------------------------------------------------------------------------------------------------------------------
void grant(int tn, int r, int u){
     avail[r]-=u;
        curr[tn][r]+=u;
}
//-------------------------------------------------------------------------------------------------------------------
void sort_task(struct tsk v[], int t){
    int temp1; struct tskact *temp2; task_state temp3;
      bool sw;
         do{
               sw = false;
```

```
                for(int i = 1; i <= t-1; i++){
                if(v[i].state > v[i + 1].state){
                    temp1 = v[i].task_id;   v[i].task_id = v[i+1].task_id;    v[i+1].task_id = temp1;
                           temp2 = v[i].taskp;     v[i].taskp=v[i+1].taskp;          v[i+1].taskp=temp2;
                           temp3 = v[i].state;     v[i].state=v[i+1].state;          v[i+1].state=temp3;
                           sw=true;
                       }
                }
            } while(sw);

}
//-------------------------------------------------------------------------------------------------------------------
void init_provisional(int v[], int r){
     for (int i=1;i<=r;++i){
           v[i]=0;
     }
}
//-------------------------------------------------------------------------------------------------------------------
```