```
#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;

//Lab 4 - Demand Paging
//*********************

const size_t MAX_FRAMES=100;
const size_t MAX_PROCS=5;                   //Using indexes 1-4 (4 jobs)
const size_t MAX_PROB=5;                    //Probabilities
const size_t MAX_MIX=5;
const size_t MAX_PAGES=200;
const size_t q=3;
struct mem {
        int process_num;
            int page_num;
            int last_cycle_ref;
            int time_loaded;
            }  memory[MAX_FRAMES];          //memory frame (sort of inverted table)
struct stat {
        int p_faults;
            int residency_t;
            } stats[MAX_PROCS][MAX_PAGES];
int  tot_evictions[MAX_PROCS];
int  tot_faults[MAX_PROCS];
int  tot_residency[MAX_PROCS];
struct rand {
          int rnd1;
              int rnd2;
         } rnd_arr[MAX_PROCS];


double p[MAX_MIX][MAX_PROCS][MAX_PROB];     //Keeps all probabilities
int cycle;                                  //Current cycle
int last_f_used;                            //Used for PRA LIFO
bool start[5];                              //Indicates when a process started
int debug;
```

```
//FUNCTIONS PROTOTYPES
int next_ref(int, int, int , int,fstream&,bool);
int modulo(int , int);
int randomos(fstream&);
int add_translate(int, int );              //Translate word address to virt. page
int find_page(int, int, int);              //Find the frame in which an input virtual page is.
int find_lru_frame(int);                   //Find the least recently used frame
int free_frame(int m, int p);              //Returns the highest free frame or -1 if none
void pager(int,int,char *, int, int,fstream&); //Simulate demand paging
void print_memory(int, int);
void init_stats(int, int);                 //Records page faults & residency time in cycles
void print_stats(int, int);                //Print stats after simulation
void store_process(int,fstream&, int);     //To facilitate quantum switching


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

//Reading parameters from standard input, some validation
   fstream f("./random_numbers.txt", ios::in);
   char *parm; int num_parm;
   int M, P, S, J, N;
   char *R;
   int proc_num;      //carries number of processes for this run (1 or 4)

   if (argc<7) {
      num_parm=7-argc;
        (num_parm > 1) ? parm="parameters" : parm="parameter";
      cout << "you are missing " << num_parm << " " << parm<<"!!!\n";
        exit(1);
   }
   M=atoi(argv[1]); P=atoi(argv[2]); S=atoi(argv[3]); J=atoi(argv[4]); N=atoi(argv[5]);
   R=argv[6];
   if (argc>7) debug=atoi(argv[7]);  //1 for detail without random, 2 - with random
   else         debug=0;

   cout << "\n The machine size is:                     " << M <<
           "\n The page size is:                        " << P <<
               "\n The process size is:                     " << S <<
               "\n The job mix number is:                   " << J <<
               "\n The number of references per process is: " << N <<
               "\n The replacement algorithm is:            " << R <<
               "\n The level of debugging output is:        " << debug << "\n";
```

```
//Initial Machine Memory frames
for (int j=0;j<M/P;++j){
   memory[j].page_num=-1;
     memory[j].process_num = 0;
     memory[j].last_cycle_ref=-1;
}
(J>1)? proc_num=4 : proc_num=1;
//End of input processing

//Assign known probabilities
int i1,j1,k1;
for (int i1=1;i1<=MAX_MIX;++i1){
    for (j1=1;j1<=MAX_PROCS;++j1){
         for (k1=1;k1<=MAX_PROCS;++k1){
            p[i1][j1][k1]=0;
           }
      }
}

if (J==1) p[J][1][1]=1; //A=1
if (J==2)
   for (i1=1;i1<=proc_num;++i1) p[J][i1][1]=1; //A=1
if (J==3)
   for (i1=1;i1<=proc_num;++i1) p[J][i1][4]=1; //1-A-B-C=1
//mix J=4
if (J==4) {
   p[J][1][1]=.75; p[J][1][2]=.25;
   p[J][2][1]=.75; p[J][2][3]=.25;
   p[J][3][1]=.75; p[J][3][2]=.125;p[J][3][3]=.125;
   p[J][4][1]=.50; p[J][4][2]=.125;p[J][4][3]=.125; p[J][4][4]=.25;
}


int w[5];                         //Use indexes 1, 2, 3, 4 - current word for jobs 1,2,3,4
int quantum;                      //quantum
last_f_used=-1;                   //For LIFO
int limit_cycles=N*proc_num;      //Maximum cycles
quantum=1; int process=1; bool
context_switch=false;

init_stats(proc_num,S/P);         //Initialize stats array
for (int fw=1;fw<=proc_num;++fw){ //Set random array to -1 (no prev randoms)
    rnd_arr[fw].rnd1=-1;  rnd_arr[fw].rnd2=-1;
}
```

```
/**************************************************************************************************
* Calculate the loop limit, so we can carry the quantum and keep each process running N references*
* The program executes two loops. The first run with the normal quantum of 3 for m references     *
* and the second completes another k (alternating quantum of 1), where k+m= 4*N.                  *
* If there is one process only or if 4*N%q==0, only the first loop executes.                      *
**************************************************************************************************/
int lim, re, loo;
if (J>1){
   lim=4*N; re=lim%q; loo=lim-re;
     while(re%proc_num !=0){                        //If initially re==0, means 4*N was divisible by q
          loo-=q;
            re +=q;
   }
}
else loo=N;                                         //When job mix is 1 (one process only)

//Main  driver loop  -------------------------------------------------------------------------------------
int p_process;
w[process]=(111*process%S);                        //First memory reference is always the same
for (int ref=1;ref<=loo;++ref){
    cycle=ref;
      pager(w[process],process,R,P,M,f) ;           //call pager with mem ref, process & PRA

      if (J>1){                                     //quantum & switching only relevant for J=2,3,4
          ++quantum;
        if (quantum>q){
             quantum=1; p_process=process;
               ++process;
               if (process>proc_num) process=1;
                 store_process(p_process, f, J);
               context_switch=true;
          }
          else context_switch=false;
      }

      w[process]=next_ref(w[process],process,J,S,f,context_switch); //Gen. next mem ref for process
}

//Remainder of cycles (if 4*N was not divisible by q - quantum is now effectively 1.
if (J>1 && re>0){
   context_switch=false;
   for (int ref1=loo+1;ref1<=limit_cycles;++ref1){
       cycle=ref1;
         pager(w[process],process,R,P,M,f) ;                        //call pager with mem ref, process & PRA
           ++process;
```

```
              if (process>proc_num) process=1;
              if (*(R+1)!='a') context_switch=true;                         //for the random frame evictions
              w[process]=next_ref(w[process],process,J,S,f,context_switch);//Next mem ref for process
        }
   }

   print_stats(proc_num,S/P);  //all processes, calculate and print final statistics and totals
   return 0;
}//END OF MAIN

//----------------------------------------------------------------------------------------------------------------
//FUNCTIONS
//----------------------------------------------------------------------------------------------------------------

int next_ref(int word, int process, int j, int s, fstream& f, bool ctx){
    //word - last address referenced, process= proc. number, j= job mix, s=process size,
      //f - random input file, ctx - context switch T or F

      int next_word; int rnd, rnd2;
      double y;

            if (!ctx) { //context switch did not occur in the main driver (normal quantum)
               rnd=randomos(f);
               if (debug>1)
                  cout << "\n No context switch for RANDOM=" <<rnd2;
               y=rnd/(INT_MAX +1.0);

               //Find probability
             if (y<p[j][process][1]) next_word=modulo(++word,s);                //w+1 - Probability A
             else
                if (y<p[j][process][1]+p[j][process][2])                        //w-5  (A+B)
                     next_word=modulo(word-5,s);
                else
                     if (y<p[j][process][1]+p[j][process][2] + p[j][process][3]) //w+4 (A+B+C)
                          next_word=modulo(word+4,s);
                     else{
                          rnd2=randomos(f);
                              if (debug>1)
                             cout << "\n mix for RANDOM=" <<rnd2;
                              next_word=(rnd2%s);
                     }

           }
           else { //The given process was just switched to (we use the previously stored
                  //random(s) or brand new first reference (unstarted process)
                  rnd=rnd_arr[process].rnd1;
                    if (rnd==-1) {               //First time for this process
                     return (111*process)%s;
                  }
                  else { //Get the last random number for this process from the array
                       if (debug>1)
                            cout << "\n next ref (with CTX switch), RANDOM=" << rnd_arr[process].rnd1;
                       y=rnd/(INT_MAX +1.0);
                          //Find probability
                   if (y<p[j][process][1]) next_word=modulo(++word,s);  //w+1  Probability A
                   else
                      if (y<p[j][process][1]+p[j][process][2])
                           next_word=modulo(word-5,s);
                      else
                           if (y<p[j][process][1]+p[j][process][2] + p[j][process][3])
                                      next_word=modulo(word+4,s);
                           else{
                               rnd2=rnd_arr[process].rnd2;
                                       if (debug>1)
                                      cout << "\n mix for RANDOM=" <<rnd2;
                                       next_word=(rnd2%s);
                           }

                  }
             }

      return next_word;
}
//-------------------------------------------------------------------------------------------------------------------
int modulo(int n, int s){
    return (n+s)%s;
}
//-------------------------------------------------------------------------------------------------------------------
int randomos(fstream& rf){
   long nextrnd;
   rf >> nextrnd;
   return nextrnd;
}
//-------------------------------------------------------------------------------------------------------------------
int add_translate(int w, int p){ //parms: address (word - w) & process size (p)
     div_t x; int page;
     x=div(w,p);
     page=x.quot;
     return page;
```

```
}
//-----------------------------------------------------------------------------------------------------------------
int find_page(int virt, int process, int allframes){
    for (int frame=0; frame<allframes; ++frame){
          if (memory[frame].page_num==virt && memory[frame].process_num==process)
               return frame;
    }
      return -1; //Page fault
}
//-----------------------------------------------------------------------------------------------------------------
int find_lru_frame(int frames){
    int lru_f=0;
      int low=memory[0].last_cycle_ref;
    for (int i=1; i<frames; ++i){
          if (memory[i].last_cycle_ref<low){
               lru_f=i;
               low=memory[i].last_cycle_ref;
        }
    }
      return lru_f;
}

//-----------------------------------------------------------------------------------------------------------------
int free_frame(int m, int p){ //Parms are Machine memory size & page size
    int high=m/p - 1;
      for (int ff=high; ff>=0; --ff){
            if (memory[ff].page_num==-1)
               return ff;
      }
      return -1;  //If no free frame
}
//-----------------------------------------------------------------------------------------------------------------
void pager(int ref,int process,char *r, int p, int m, fstream& f){

       int evict_page, evict_frame, evict_process, findex;
     int page_ref; //Virtual page
       int rnd, rndf, new_frame;
     page_ref=add_translate(ref,p);

     //If the page is in memory==> HIT
       int q=find_page(page_ref,process,m/p);    //Which frame?
       if (q > -1){                               //HIT - virt. page is resident
          if (debug>0)
             cout << "\n"<< process << " references word " << ref << " (page: " << page_ref << ") at time: "<<
                    cycle << " : Hit in frame: " << q;
            memory[q].last_cycle_ref=cycle;
       }
       else{                                     //Page not resident ==> PAGE FAULT
           stats[process][page_ref].p_faults++;  //Update stats for page fault
           new_frame=free_frame(m,p);            //If there is free frame, choose highest

           if (new_frame==-1){  //No free frames ==> make one  EVICTION by the input PRA
                                  //The second letter of the PRA name is unique, so it is used to switch
              switch (*(r+1)){
                        case 'r':   //LRU find frame to evict
                                    findex=find_lru_frame(m/p);
                                    if (debug>1)
                                       cout <<"\n LRU eviction frame=" << findex; //deb
                                    last_f_used=findex;
                                    break;
                        case 'a':   //Random frame selection (out of m/p frames)
                                    rnd=randomos(f);
                                    if (debug>1)
                                       cout << "\n before random eviction, RANDOM=" << rnd; //deb
                                    rndf=rnd%(m/p);             //Randomly select which frame a victim lies in
                                    findex=rndf;
                                    last_f_used=rndf;
                                    break;

                        case 'i':   //LIFO - Not the best performer, so sample of how bad things can be
                                    findex=last_f_used;
                                    break;
                           default: cout << "\n Illegal PRA (or misspelled), please check input\n"; exit(0);
                                    break;

                }//End of switch

                  evict_frame=findex;
                  evict_page=memory[findex].page_num;
                  evict_process=memory[findex].process_num;
                  //Update stats for resident page being evicted from frame findex
                  stats[evict_process][evict_page].residency_t+=cycle - memory[findex].time_loaded;
                  tot_evictions[evict_process]++;
                  memory[findex].page_num=page_ref; //Place new virt. page into the clean frame
                  memory[findex].process_num=process;
                  memory[findex].last_cycle_ref=cycle;
                  memory[findex].time_loaded=cycle;
                  if (debug>0)
                      cout << "\n"<< process << " references word " << ref << " (page: " << page_ref << ") at time: "<<
                           cycle << " : Fault, evicting page: " << evict_page << " of " << evict_process <<
```

```
                              " from frame " << evict_frame << ".";

            }
            else {                                //Make the new page resident in the new found (highest) free frame.

                   memory[new_frame].page_num=page_ref;
                   memory[new_frame].process_num=process;
                   memory[new_frame].last_cycle_ref=cycle;
                   memory[new_frame].time_loaded=cycle;
                   last_f_used=new_frame;
                   if (debug>0)
                      cout << "\n"<< process << " references word " << ref << " (page: " << page_ref << ") at time: "<<
                           cycle << " : Fault using free frame: " << new_frame << ".";

            }

       } //Page Fault
}
//------------------------------------------------------------------------------------------------------------------
void print_memory(int m, int p){
   cout <<"\nMEMORY snapshot\n";
   for (int i=0;i<m/p;++i){
       cout << "\n FRAME=" << i << " Page:" << memory[i].page_num << " Process: " << memory[i].process_num<<
                 " Last_cycle_ref=" << memory[i].last_cycle_ref;
   }
}
//------------------------------------------------------------------------------------------------------------------
void init_stats(int procs, int pages){
    for (int i=1;i<=procs;++i){
         for (int j=0;j<pages;++j){
             stats[i][j].p_faults=0;
             stats[i][j].residency_t=0;
         }
         tot_evictions[i]=0;
         tot_faults[i]=0;
         tot_residency[i]=0;
    }
}
//------------------------------------------------------------------------------------------------------------------
void print_stats(int procs, int pages){
     double avg_residency=0; int overall_faults=0;
     double overall_residency=0;
     double overall_avg_residency=0;
     int    overall_evictions=0;
     bool   no_residency=false;
     bool   no_residency_at_all=false;
     char *desc;

     for (int k1=1;k1<=procs;++k1){
           for (int k2=0;k2<pages;++k2){
               tot_faults[k1]+=stats[k1][k2].p_faults;
                   tot_residency[k1]+=stats[k1][k2].residency_t;
           }
     }
     for (int i=1; i<=procs; ++i){
          no_residency=false;
            (tot_faults[i]>1)? desc="faults " : desc="fault";
          if (tot_evictions[i] > 0){
               //cout << "\n Tot_residency=" << tot_residency[i] << " Tot evictions=" << tot_evictions[i];
             avg_residency=(double)tot_residency[i] / (double) tot_evictions[i];
            }
            else no_residency = true;

            overall_faults+=tot_faults[i];
            overall_residency+=tot_residency[i];
            overall_evictions+=tot_evictions[i];

            if (!no_residency)
             cout << "\nProcess " << i << " had " << tot_faults[i] << " " << desc<<" and " << avg_residency <<
                       "  average residency.";
            else
               cout << "\nProcess " << i << " had " << tot_faults[i] << " " << desc<< ".  " <<
                           "\n    with no evictions, average residency is undefined";

         }

         if ( overall_evictions > 0)
            overall_avg_residency=overall_residency / overall_evictions;
         else
            no_residency_at_all=true;
            desc="faults ";
         if (!no_residency_at_all)
            cout << "\n\n The total number of " << desc << " is: " << overall_faults <<
                    " and the overall average residency is: " << overall_avg_residency<<"\n";
         else
            cout << "\n\n The total number of " << desc << " is: " << overall_faults <<
                  "\n    with no evictions, the overall average residency is undefined\n";

}
//----------------------------------------------------------------------------------------------------------------
```

```
void store_process(int process,fstream& f, int j){
       //This function stores the RANDOM value(s) 1 or 2, for a process that 'context switched' to another, in
       //order to follow the pattern of using the same random numbers as the professor
       if (debug>1)
          cout << "\n IN STORE_PROCESS";
       int rnd, rnd2;
       rnd=randomos(f);
       rnd_arr[process].rnd1=rnd;
       if (debug>1)
          cout << "\n RANDOM1 (stored in rnd_arr[" << process <<"].rnd1)=" << rnd_arr[process].rnd1;
       double y=rnd/(INT_MAX +1.0);
       if (debug>1)
          cout << "\n y=" << y;
       if (y>=p[j][process][1]+p[j][process][2] + p[j][process][3]){ //A+B+C
            rnd2=randomos(f);
            rnd_arr[process].rnd2=rnd2;
            if (debug>1)
               cout << "\n RANDOM2 (stored in rnd_arr[" << process <<"].rnd2)=" << rnd_arr[process].rnd2;
       }
}
//-----------------------------------------------------------------------------------------------------------------
```