```
#include <iostream.h>
#include <fstream.h>


typedef struct page
{
   int  size;
   int  pageAddress;
   int  memAddress;
   bool empty;
   bool dirty;
   int  accessOrder;
   int  accessStart;
   int  jobId;
} page;

typedef struct job
{
   int jobId;
   int size;
} job;

static int pageFaultCount = 0;
static int pageHitCount = 0;
static int orderCounter;
static int startCounter;
static page virtualPage[4];
static job jobArray[10];
static int jobCount = 0;
static int currentJob = 0;

typedef struct newJob
{
   int jobId;
   int jobSize;
} newJob;

typedef struct switchCpu
{
   int jobId;
} switchCpu;

typedef struct readAccess
{
   int address;
} readAccess;

typedef struct writeAccess
{
   int address;
} writeAccess;


typedef struct endJob
{
   int jobId;
} endJob;

typedef enum OPCODE
{
   NEW_JOB        = 1,
   SWITCH_CPU     = 2,
   READ_ACCESS    = 3,
   WRITE_ACCESS   = 4,
   END            = 5
} OPCODE;

typedef enum REPTYPE
{
   FIFO,
   LRU,
   OPTIMAL
} REPTYPE;
typedef struct opCode
{
   OPCODE type;
   union
   {
      newJob      uNewJob;
```

```
      switchCpu   uSwitchCpu;
      readAccess  uReadAccess;
      writeAccess uWriteAccess;
      endJob      uEndJob;
   };
} opCode;

typedef struct opCodeList
{
   opCode* current;
   opCodeList* next;
} opCodeList;

opCodeList* ReadInputFile(ifstream& inStream)
{
   int         temp;
   opCodeList* first;
   opCodeList* opList;
   opCode* op;

   first = opList = new opCodeList;
   while (!inStream.eof())
   {
      op = opList->current = new opCode;

      // Get the OP Code
      inStream >> temp;
      op->type = (OPCODE) temp;

      if (inStream.eof())
      {
        opList->next = NULL;
        break;
      }

      switch (op->type)
      {
      case NEW_JOB:

         // read job number
         inStream >> temp;
         op->uNewJob.jobId = temp;

         // read job size
         inStream >> temp;
         op->uNewJob.jobSize = temp;
         break;

      case SWITCH_CPU:

         // read ID
         inStream >> temp;
         op->uSwitchCpu.jobId = temp;
         break;

      case READ_ACCESS:

         // read address
         inStream >> temp;
         op->uReadAccess.address = temp;
         break;

      case WRITE_ACCESS:

         // read address
         inStream >> temp;
         op->uWriteAccess.address = temp;
         break;

      case END:

         // read ID
         inStream >> temp;
         op->uEndJob.jobId = temp;
         break;

      }

      opList->next = new opCodeList;
```

```
      opList = opList->next;
      opList->next = NULL;
   }

   return first;
}

job*  GetJob(int jobId)
{
   for (int c=0; c< jobCount ; c++)
   {
      if (jobArray[c].jobId == jobId)
         return &jobArray[c];
   }

   return NULL;
}

void Init()
{
   pageHitCount = pageFaultCount = 0;

   for (int c=0; c<4; c++)
   {
      virtualPage[c].pageAddress = c * 1000;
      virtualPage[c].empty = true;
      virtualPage[c].dirty = false;
      virtualPage[c].size = 1000;
   }
}

void DeleteJob(int jobId)
{
   job* job = GetJob(jobId);

   for (int c=0; c<4; c++)
   {
      if (virtualPage[c].jobId == jobId)
      {
         virtualPage[c].pageAddress = c * 1000;
         virtualPage[c].empty = true;
         virtualPage[c].dirty = false;
         virtualPage[c].size = 1000;
      }
   }
}

page* GetPage(int jobId, int address)
{
   job* job = GetJob(jobId);
   for (int c=0; c<4; c++)
   {
      if (!virtualPage[c].empty &&
         jobId == virtualPage[c].jobId &&
         virtualPage[c].memAddress < address &&
         (virtualPage[c].memAddress + virtualPage[c].size)  > address
         )
      {
         return &virtualPage[c];
      }

   }

   return NULL;
}

page* GetFirstEmpty(int jobId)
{
   job* job = GetJob(jobId);
   for (int c=0; c<4; c++)
   {
      if (virtualPage[c].empty)
         return &virtualPage[c];
   }

   return NULL;
}

page* GetPageReplace(int jobId, REPTYPE replacementType, opCodeList* list)
```

```
{
   job* job = GetJob(jobId);
   opCodeList* start = list;

   if (replacementType == FIFO)
   {
      int min = virtualPage[0].accessStart;
      int minIndex = 0;

      for (int c=1; c<4; c++)
      {
         if (min> virtualPage[c].accessStart)
         {
            min = virtualPage[c].accessStart;
            minIndex = c;
         }
      }

      return &virtualPage[minIndex];
   }

   else if (replacementType == LRU)
   {
      int min = virtualPage[0].accessOrder;
      int minIndex = 0;

      for (int c=1; c<4; c++)
      {
         if (min> virtualPage[c].accessOrder)
         {
            min = virtualPage[c].accessOrder;
            minIndex = c;
         }
      }

      return &virtualPage[minIndex];
   }

   else if (replacementType == OPTIMAL)
   {
      int refrencePage[4];

      for (int c=0; c<4; c++)
      {
         int counter =0;
         list = start;
         refrencePage[c] = 1000;
         while (list->next != NULL)
         {
            opCode* op = list->current;
            if ( (op->type == WRITE_ACCESS || op->type == READ_ACCESS) &&
               virtualPage[c].memAddress < op->uReadAccess.address &&
               (virtualPage[c].memAddress + virtualPage[c].size)  > op->uReadAccess.address
               )
            {
               refrencePage[c] = counter;
               break;
            }
            counter++;
            list = list->next;
         }
      }

      int max = refrencePage[0];
      int maxIndex = 0;

      for (int c=1; c<4; c++)
      {
         if (refrencePage[c] > max)
         {
            max = refrencePage[c];
            maxIndex = c;
         }
      }

      return &virtualPage[maxIndex];
   }

   return NULL;
```

```
}

void run(opCodeList* opList, REPTYPE replacementType)
{
   Init();

   opCode* op;
   opCodeList* list= opList;


   switch (replacementType)
   {
   case FIFO:
      cout << "\n*** First In First Out ***\n\n";
      break;
   case LRU:
      cout << "\n*** Least Recently Used ***\n\n";
      break;
   case OPTIMAL:
      cout << "\n*** Optimal ***\n\n";
   }
   while (list->next != NULL)
   {
      op = list->current;

      switch (op->type)
      {
      case NEW_JOB:

         jobArray[jobCount].jobId =
            op->uNewJob.jobId;

         jobArray[jobCount ++ ].size =
            op->uNewJob.jobSize;

         cout << "Job " << op->uNewJob.jobId
             << "  Size = " <<
             op->uNewJob.jobSize << endl;
         break;

      case SWITCH_CPU:

         currentJob = op->uSwitchCpu.jobId;

         cout << "Switch to job " << currentJob << endl;
         break;

      case WRITE_ACCESS:
      case READ_ACCESS:
         {
            if (op->type == WRITE_ACCESS)
               cout << "WRITE  " << op->uReadAccess.address << endl;
            else
               cout << "READ  " << op->uReadAccess.address << endl;

            // Check for memory violation
            job* j = GetJob(currentJob);
            if (op->uReadAccess.address >=  ((int)(j->size/1000) + 1) * 1000)
            {
               cout << "   MEMORY VIOLATION\n";
               break;
            }

            // Check for page hit
            page* p = GetPage(currentJob, op->uReadAccess.address);
            if (p)
            {
               pageHitCount ++;
               if (op->type == WRITE_ACCESS)
                  p->dirty = true;
               p->accessOrder = orderCounter ++;
               p->jobId = currentJob;
               cout << " LOCATION  " <<
                  p->pageAddress + (op->uReadAccess.address % 1000) << endl;
               break;
            }

            // Check for page fault (free block)
            p = GetFirstEmpty(currentJob);
```

```
            if (p)
            {
               pageFaultCount ++;
               p->empty = false;
               if (op->type == WRITE_ACCESS)
                  p->dirty = true;
               p->memAddress = ((int) (op->uReadAccess.address / 1000) ) * 1000;
               p->accessOrder = orderCounter ++;
               p->accessStart = startCounter ++;
               p->jobId = currentJob;
               cout << " Page Fault.\n"
                   << " Using Free Block.\n"
                   << " LOCATION  " <<
                   p->pageAddress + (op->uReadAccess.address % 1000) << endl;;
               break;
            }

            // Check for page fault (Page Replacement)
            p = GetPageReplace(currentJob, replacementType, list->next);
            if (p)
            {
               p->memAddress = ((int) (op->uReadAccess.address / 1000) ) * 1000;
               p->accessOrder = orderCounter ++;
               p->accessStart = startCounter ++;
               pageFaultCount ++;
               p->jobId = currentJob;
               cout << " Page Fault.\n"
                   << " Page Replacement.\n";

               if (p->dirty)
               {
                  cout << "      Page Out\n";
                  p->dirty = false;
               }

               cout << " LOCATION  " <<
                   p->pageAddress + (op->uReadAccess.address % 1000) << endl;

               if (op->type == WRITE_ACCESS)
                  p->dirty = true;
               break;
            }

            // If none, it is an error
            cout << " No block found\n";
         }
         break;

      case END:

         DeleteJob(op->uEndJob.jobId);
         cout << "End job" << op->uEndJob.jobId << endl;
         break;
      }

      list = list->next;
   }

   cout << endl << "Page Hits = " << pageHitCount << endl;
   cout << "Page Faults = " << pageFaultCount << endl;
}


int main()
{
   opCodeList* opList;

   ifstream inStream("Vm.dat");

   if (!inStream)
   {
      cout << "Can't find input file\n";
      return 1;
   }

   opList = ReadInputFile(inStream);

   run(opList, FIFO);
   run(opList, LRU);
```

```
    run(opList, OPTIMAL);

    return 0;
}
```