OS Phase 2

Name: Bhavin Patil
Roll No.: 66

```
#include <bits/stdc++.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
using namespace std;

#define TRUE 1
#define FALSE 0
#define SIZE 300
#define REAL_MEMORY_LEN 300
#define VIRTUAL_MEMORY_LEN 100
#define FRAME_SIZE 10
#define PAGE_SIZE FRAME_SIZE

const int NO_ERROR = 0;
const int OUT_OF_DATA_ERROR = 1;
const int LINE_LIMIT_EXCEEDED_ERROR = 2;
const int TIME_LIMIT_EXCEEDED_ERROR = 3;
const int OPERATION_CODE_ERROR = 4;
const int OPERAND_ERROR = 5;
const int INVALID_PAGE_FAULT_ERROR = 6;

typedef struct ProcessControlBoard
{
   int jId;
   int ttc; //  Total Time Counter
   int tlc; //  Total Limit Counter
   int ttl; //  Total Time Limit
   int tll; //  Total Line Limit
```

```
} PCB;

typedef struct memoryContext
{
   char M[300][4];
   char R[4];
   char IR[4];
   int IC;
   int C;
   char buffer[41];
   FILE *fReadPtr;
   FILE *fWritePtr;
   int PTR;
} MEMORY;

typedef struct interrupts
{
   int PI;
   int SI;
   int TI;
} INTERRUPTS;

void init(char M[300][4], int size)
{
   memset(M, '\0', size * sizeof(char));
}

int readLine(FILE *fptr, char *buffer)
{
   memset(buffer, '\0', 10 * 40);
   int res = fscanf(fptr, "%[^\n]", buffer);
   getc(fptr);
   return res;
}

void writeLine(FILE *fptr, char *content)
{
   fprintf(fptr, "%s", content);
   fputc('\n', fptr);
}
```

```
int allocate(char M[300][4], int limit)
{
   limit = limit / 10;
   srand(time(0));

   int random;
   random = rand() % limit;
   if (M[random * 10][0] != '\0')
   {
       for (int i = 1; i < limit; i++)
       {
           char c = M[((random + i) % limit) * 10][0];
           if (c == '\0')
           {
               return random + i;
           }
       }
   }
   return random;
}

void charToNumber(char *numberInCharArr, int *number, int length)
{
   *number = 0;
   for (int i = 0; i < length; i++)
   {
       *number = (*number * 10) + ((int)numberInCharArr[i] - 48);
   }
}

void checkIfPageFault(char M[300][4], int pAddress, int *isValid)
{
   *isValid = true;

   if ((M[pAddress][3] == '*') || (M[pAddress][3] == '\0'))
   {
       *isValid = false;
   }
}
```

```
void numberToChar(int number, char *numberInCharArr, int length)
{
   int pointer = length - 1;
   while ((number != 0) && (pointer >= 0))
   {
       numberInCharArr[pointer--] = (char)((number % 10) + 48);
       number /= 10;
   }
   while (pointer >= 0)
   {
       numberInCharArr[pointer--] = '0';
   }
}

void createNewPage(MEMORY *memCnt, int offset, int *newPageOffset)
{
   *newPageOffset = allocate(memCnt->M, REAL_MEMORY_LEN);
   char newPageOffsetInChar[4];

   memset(memCnt->M[*newPageOffset * 10], '*', 4 * PAGE_SIZE);

   numberToChar(*newPageOffset, newPageOffsetInChar, 4);
   memcpy(&memCnt->M[(memCnt->PTR * 10) + offset], newPageOffsetInChar,
4);
}

void addressMAP(MEMORY *memCnt, int vAddress, int *pAddress) // PTR = Page
Table Register || PTE = Page Table Entry
{
   int PTRBaseAddr = memCnt->PTR * PAGE_SIZE;
   int PTE = PTRBaseAddr + (vAddress / PAGE_SIZE);
   int pageFound = false;

   checkIfPageFault(memCnt->M, PTE, &pageFound); // Page not Found == Page
Fault

   if (!pageFound)
   {
       int newPageOffset;
```

```
        char newPageOffsetIntCharArr[4];

        int pageTableEntryOffset = PTE % (memCnt->PTR * PAGE_SIZE);

        createNewPage(memCnt, pageTableEntryOffset, &newPageOffset);
        numberToChar(newPageOffset, newPageOffsetIntCharArr, 4);
        memcpy(memCnt->M[PTE], newPageOffsetIntCharArr, 4);
        *pAddress = newPageOffset * 10;
    }
    else
    {
        int targetAddress;
        charToNumber(memCnt->M[PTE], &targetAddress, 4);
        *pAddress = (targetAddress * 10) + (vAddress % PAGE_SIZE);
    }
}

void checkIfValidOperant(char *IR, int *Flag)
{
    *Flag = false;
    if ((('0' <= IR[2]) && (IR[2] <= '9')) && (('0' <= IR[3]) && (IR[3] <=
'9')))
    {
        *Flag = true;
    }
}

void compareString(char *str1, char *str2, int len, int *Equal)
{
    *Equal = true;
    for (int i = 0; i < len; i++)
    {
        if (*(str1 + i) != *(str2 + i))
        {
            *Equal = false;
            break;
        }
    }
}
```

```
void terminate(PCB *jobPCB, INTERRUPTS *interrupts, FILE *fWritePtr, int
errCount, ...)
{
   int erroId;

   va_list argPointer;
   va_start(argPointer, errCount);

   fprintf(fWritePtr, "%s", "Execution Terminated Abnormally\n\n");

   for (int i = 0; i < errCount; i++)
   {
       erroId = va_arg(argPointer, int);

       if (erroId == NO_ERROR)
       {
           cout << "In No Error" << endl;
       }
       else if (erroId == OUT_OF_DATA_ERROR)
       {
           fprintf(fWritePtr, "%s", "~~~~~~ OUT_OF_DATA_ERROR ~~~~~~ \n");
       }
       else if (erroId == LINE_LIMIT_EXCEEDED_ERROR)
       {
           fprintf(fWritePtr, "%s", "~~~~~~ LINE_LIMIT_EXCEEDED_ERROR
~~~~~~ \n");
       }
       else if (erroId == TIME_LIMIT_EXCEEDED_ERROR)
       {
           fprintf(fWritePtr, "%s", "~~~~~~ TIME_LIMIT_EXCEEDED_ERROR
~~~~~~ \n");
       }
       else if (erroId == OPERATION_CODE_ERROR)
       {
           fprintf(fWritePtr, "%s", "~~~~~~ OPERATION_CODE_ERROR ~~~~~~
\n");
       }
       else if (erroId == OPERAND_ERROR)
       {
           fprintf(fWritePtr, "%s", "~~~~~~ OPERAND_ERROR ~~~~~~ \n");
```

```
        }
        else if (erroId == INVALID_PAGE_FAULT_ERROR)
        {
            fprintf(fWritePtr, "%s", "~~~~~~ INVALID_PAGE_FAULT_ERROR
~~~~~~ \n");
        }
    }

    fprintf(fWritePtr, "%s",
"\n---------------------------------------------------------------\n");
    fprintf(fWritePtr, "|  JID  |  TTC  |  TLC  |  TTL  |  TLL  |
SI  |   TI  |   PI  |\n");
    fprintf(fWritePtr, "%s",
"---------------------------------------------------------------\n");
    fprintf(fWritePtr, "|  %3d  |  %3d  |  %3d  |  %3d  |  %3d  |  %3d  |
%3d  |  %3d  |\n ",
            jobPCB->jId, jobPCB->ttc, jobPCB->tlc, jobPCB->ttl,
jobPCB->tll, interrupts->SI, interrupts->SI, interrupts->TI,
interrupts->PI);
    fprintf(fWritePtr, "%s",
"---------------------------------------------------------------\n");

    exit(1);
}

void interruptHandler(PCB *jobPCB, INTERRUPTS *interrupts, FILE
*fileWriter)
{
    if (interrupts->TI == 0)
    {
        if (interrupts->SI == 3)
        {
            terminate(jobPCB, interrupts, fileWriter, 1, NO_ERROR);
        }
        else if (interrupts->PI == 1)
        {
            terminate(jobPCB, interrupts, fileWriter, 1,
OPERATION_CODE_ERROR);
        }
        else if (interrupts->PI == 2)
```

```
        {
            terminate(jobPCB, interrupts, fileWriter, 1, OPERAND_ERROR);
        }
        else if (interrupts->PI == 3)
        {
            // TODO - valid page fault allocate, update page table, adjust
IC if needed
            // TODO - resume execution of 'executeUserProgram()' program
otherwise terminate(6)

            // TODO - Check valid page fault or not and rest of the stuff
            // if(checkIfPageFault())
            terminate(jobPCB, interrupts, fileWriter, 1,
INVALID_PAGE_FAULT_ERROR);
        }
    }
    else if (interrupts->TI == 2)
    {

        if (interrupts->PI == 1)
        {
            terminate(jobPCB, interrupts, fileWriter, 2,
TIME_LIMIT_EXCEEDED_ERROR, OPERATION_CODE_ERROR);
        }
        else if (interrupts->PI == 2)
        {
            terminate(jobPCB, interrupts, fileWriter, 2,
TIME_LIMIT_EXCEEDED_ERROR, OPERAND_ERROR);
        }

        if (interrupts->SI == 1)
        {
            terminate(jobPCB, interrupts, fileWriter, 1,
TIME_LIMIT_EXCEEDED_ERROR);
        }
        else if (interrupts->SI == 2)
        {
            // TODO - Write then terminate
            terminate(jobPCB, interrupts, fileWriter, 1,
TIME_LIMIT_EXCEEDED_ERROR);
```

```
        }
        else if (interrupts->SI == 3)
        {
            terminate(jobPCB, interrupts, fileWriter, 1, NO_ERROR);
        }
        else if (interrupts->PI == 3)
        {
            terminate(jobPCB, interrupts, fileWriter, 1,
TIME_LIMIT_EXCEEDED_ERROR);
        }
    }
}

void executeUserProgram(MEMORY *memContent, PCB *jobPCB, INTERRUPTS
*interrupts)
{
    int pAddress;
    int vAddress;
    int PI = 0;
    int isValid = false;
    int isValidOperant = true;

    jobPCB->ttc = 0;
    memContent->fWritePtr = fopen("output.txt", "w");

    while (true)
    {
        addressMAP(memContent, memContent->IC, &pAddress);

        if (PI != 0)
            break; // PI error checking

        memcpy(memContent->IR, memContent->M[pAddress], 4);

        checkIfValidOperant(memContent->IR, &isValidOperant); // valid
operand or not
        if (!isValidOperant)
            interrupts->PI = 2;
```

```
      if ((memContent->IR[0] == 'L') && (memContent->IR[1] == 'R'))
//--------------LR
      {
          ++jobPCB->ttc;
          if (isValidOperant)
          {
              char vAddressInChar[4];
              charToNumber(&(memContent->IR[2]), &vAddress, 2);
              numberToChar(vAddress, vAddressInChar, 4);
              memcpy(memContent->R, vAddressInChar, 4);
          }
      }

      else if ((memContent->IR[0] == 'S') && (memContent->IR[1] == 'R'))
//---------------SR
      {
          jobPCB->ttc += 2;
          if (isValidOperant)
          {
              charToNumber(&(memContent->IR[2]), &vAddress, 2);
              addressMAP(memContent, vAddress, &pAddress);
              memcpy(memContent->M[pAddress], memContent->R, 4);
          }
      }

      else if ((memContent->IR[0] == 'C') && (memContent->IR[1] == 'R'))
//------------CR
      {
          ++jobPCB->ttc;
          if (isValidOperant)
          {
              int virtualIRStart, virtualRStart;
              int realIRStart, realRStart;

              charToNumber(&memContent->IR[2], &virtualIRStart, 2); //
logical Address
              charToNumber(&memContent->R[2], &virtualRStart, 2);

              addressMAP(memContent, virtualIRStart, &realIRStart); //
              addressMAP(memContent, virtualRStart, &realRStart);
```

```
            compareString(memContent->M[realIRStart],
memContent->M[realRStart], 4 * PAGE_SIZE, &memContent->C);
        }
    }

    else if ((memContent->IR[0] == 'B') && (memContent->IR[1] == 'T'))
//----------BT
    {
        ++jobPCB->ttc;
        if (isValidOperant)
        {
            if (memContent->C == true)
            {
                int jumpOffset;
                charToNumber(memContent->IR + 2, &jumpOffset, 2);
                memContent->IC = jumpOffset - 1;
            }
        }
    }

    else if ((memContent->IR[0] == 'G') && (memContent->IR[1] == 'D'))
//----------GD
    {
        jobPCB->ttc += 2;
        interrupts->SI = 1;
        if (isValidOperant)
        {
            charToNumber(memContent->IR + 2, &vAddress, 2);
            addressMAP(memContent, vAddress, &pAddress);

            int res = readLine(memContent->fReadPtr,
memContent->buffer);
            if (res == -1)
            {
                cout << "UNEXPECTED EOF !!!" << endl;
                exit(1);
            }
```

```
            if ((memContent->buffer[0] == '$') &&
(memContent->buffer[1] == 'E'))
            {
                // cout << "NO DATA FOUND TO READ !!!" << endl;
                exit(1);
            }

            memcpy(memContent->M[pAddress], memContent->buffer, 4 *
PAGE_SIZE);
        }
    }

    else if ((memContent->IR[0] == 'P') && (memContent->IR[1] == 'D'))
//----------PD
    {
        ++jobPCB->ttc;
        ++jobPCB->ttc;
        interrupts->SI = 2;

        if (isValidOperant)
        {
            int PTRBaseAddress = memContent->PTR * PAGE_SIZE;
            int PTE = PTRBaseAddress + (vAddress / PAGE_SIZE);
            checkIfPageFault(memContent->M, PTE, &isValid);

            if (!isValid) // checking whether location actually exit
i.e. PAGE FAULT
            {
                // INVOKE SOME INTERRUPTS
            }
            else // IF exist then put it in output.txt
            {
                charToNumber(memContent->IR + 2, &vAddress, 2);
                addressMAP(memContent, vAddress, &pAddress);

                memcpy(memContent->buffer, memContent->M[pAddress], 4 *
PAGE_SIZE);
                writeLine(memContent->fWritePtr, memContent->buffer);
            }
        }
```

```
        }

        else if (memContent->IR[0] == 'H') //-----------------H
        {
            ++jobPCB->ttc;
            interrupts->SI = 3;
            putc('\n', memContent->fWritePtr);
            putc('\n', memContent->fWritePtr);
            break;
        }
        else
        {
            // Handle OPCODE ERROR
            interrupts->PI = 1;
        }

        ++memContent->IC;
        if (jobPCB->ttc > jobPCB->ttl)
        {
            // Generate Interrupts
            interrupts->TI = 2;
        }
        if (jobPCB->tlc > jobPCB->tll)
        {
            terminate(jobPCB, interrupts, memContent->fWritePtr, 1,
LINE_LIMIT_EXCEEDED_ERROR);
        }

        interruptHandler(jobPCB, interrupts, memContent->fWritePtr);
    }
}

void startExecution(MEMORY *memContent, PCB *job, INTERRUPTS *interrupts)
{
    memContent->IC = 0;
    executeUserProgram(memContent, job, interrupts);
}

int main()
{
```

```
   PCB job;
   job.ttc = 0;
   job.tlc = 0;

   MEMORY memContent;

   INTERRUPTS interrupts;
   interrupts.SI = 0;
   interrupts.PI = 0;
   interrupts.TI = 0;

   init(memContent.M, REAL_MEMORY_LEN * 4);

   memContent.fReadPtr = fopen("input.txt", "r");

   while (!feof(memContent.fReadPtr))
   {
       int res = readLine(memContent.fReadPtr, memContent.buffer);
       if (res == -1)
       {
           cout << "\n\nEncountered EOF\n\n";
           break;
       }
       if (memContent.buffer[0] == '$') // for control cards
       {
           if (memContent.buffer[0] == 'A') //$AMJ
           {
               memContent.PTR = allocate(memContent.M, REAL_MEMORY_LEN);
               memset(memContent.M[memContent.PTR * 10], '*', 4 *
PAGE_SIZE);

               char tempArr[4];

               memcpy(tempArr, memContent.buffer + 4, 4); // copying job
id
               charToNumber(tempArr, &job.jId, 4);

               memcpy(tempArr, memContent.buffer + 8, 4); // copying total
time limit
               charToNumber(tempArr, &job.ttl, 4);
```

```
            memcpy(tempArr, memContent.buffer + 12, 4); // copying
total line limit
            charToNumber(tempArr, &job.tll, 4);
        }
        else if (memContent.buffer[1] == 'D') //$DTA
        {
            startExecution(&memContent, &job, &interrupts);
        }
        else if (memContent.buffer[1] == 'E') //$END
        {
            break;
        }
    }
    else // for program card
    {
        int pageTableOffset = 0;
        int newPageOffset = -1;

        if (feof(memContent.fReadPtr))
        {
            cout << "UNEXPECTED EOF !!!" << endl;
        }

        createNewPage(&memContent, pageTableOffset, &newPageOffset); //
to create new page in PTR
        ++pageTableOffset;

        int internalPageOffset = 0, readerOffset = 0;
        char first;

        while (true)
        {
            first = memContent.buffer[readerOffset];

            if (internalPageOffset > 9) // if page offset goes above 10
indexes
            {
                createNewPage(&memContent, pageTableOffset,
&newPageOffset); // to create new page in PTR
```

```
                    ++pageTableOffset;
                    internalPageOffset = 0;
                }

                if (first == '\0') // if reader offset is on end of line
                {
                    res = readLine(memContent.fReadPtr, memContent.buffer);
                    readerOffset = 0;
                    first = memContent.buffer[readerOffset];
                    if (res == -1)
                    {
                        cout << "UNEXPECTED EOF !!!" << endl;
                        exit(1);
                    }
                }

                if (first = 'H')
                {
                    memContent.M[(newPageOffset * 10) +
internalPageOffset][0] = memContent.buffer[readerOffset];
                    ++internalPageOffset;
                    break;
                }
                else
                {
                    memcpy(memContent.M[(newPageOffset * 10) +
internalPageOffset], &memContent.buffer[readerOffset], 4);
                    readerOffset += 4;
                    ++internalPageOffset;
                }
            }
        }
    }
    return 0;
}
```