Vishwakarma Institute Of Technology,Pune

SY30
Raj Dugad (04) - 12110101
Akshata Gile (15) - 12110337
Gyaneshwari Patil (22) - 12111379
Shrinivas Hatyalikar (24) – 12110883

OPERATING SYSTEM COURSE PROJECT

OS Phase 1

ASSUMPTIONS:

1. Jobs entered without error in input file
2. No physical separation between jobs
3. Job outputs separated in output file by 2 blank lines
4. Program loaded in memory starting at location 00
5. No multiprogramming, load and run one program at a time
6. SI interrupt for service request

NOTATION

1. M: memory; IR: Instruction Register (4 bytes)
2. IR [1, 2]: Bytes 1, 2 of IR/Operation Code
3. IR [3, 4]: Bytes 3, 4 of IR/Operand Address
4. M[&]: Content of memory location &
5. IC: Instruction Counter Register (2 bytes)
6. R: General Purpose Register (4 bytes)
7. C: Toggle (1 byte)
8. : Loaded/stored/placed into

Here's a breakdown of code

1. The OSCP class has private member variables representing different components of the operating system, such as **memory, registers, instruction counter, interrupt, and a buffer for input/output.**

2. The class defines several member functions including **init(), MOS(), Execute(), and load().**

3. The **init()** function initializes the memory, registers, and other variables to 0.

4. The **MOS()** function handles different interrupt requests based on the value of the SI variable. It performs operations such as reading input, writing output, or terminating the program.

5. The **Execute()** function is the main control loop that fetches instructions from memory, interprets them, and performs corresponding actions based on the instruction type.
6. The **load()** function reads instructions from an input file (input.txt) and loads them into memory. It also executes the instructions when it encounters a specific marker.

7. The **main()** function creates an instance of the OSCP class, opens the input and output files, and calls the load function to start the execution.

Code:

```
#include<iostream>
#include<fstream>
using namespace std;

class OSCP{
    private:
        char M[100][4]; //Memory of 100x4
        char IR[4];     //Instruction Register of 4  bytes for storing 
instructions
        char R[4];      //General purpose Register of 4 bytes
        int IC;         //Instruction Counter Register of 2 bytes;
        bool C;         //Toggle 1 byte
        int SI;         //Interrupt
        char buffer[40];

    //functions
    public:
        void init();
        void load();
        void Execute();
        void MOS();

        fstream infile;
        fstream outfile;

};

void OSCP::init(){
    //Initialise everything to 0
    for(int i=0;i<100;i++){
        for(int j=0;j<4;j++){
            M[i][j]=0;
        }
    }
    IR[0] = {0};
    R[0] = {0};
    C = false;
}
```

```cpp
void OSCP::MOS(){
    cout<<endl<<IR[0]<<IR[1]<<IR[2]<<IR[3];
    if(SI==1){          //For GD therefore READ MODE
        for(int i=0;i<=39;i++){
            buffer[i]='\0';
        }
        infile.getline(buffer,40);
        int k=0;
        int i=IR[2]-48;
        i=i*10;
        for(int l=0;l<10;l++){
            for(int j=0;j<4;j++){
                M[i][j]=buffer[k];
                k++;
            }
            if(k==40){
                break;
            }
            i++;
        }

    }
    else if(SI==2)        //For PD therefore WRITE MODE
    {
        for(int i=0;i<=39;i++)
           buffer[i]='\0';
        int k = 0;
            int i = IR[2]-48;
            i = i*10;
        for( int l=0 ; l<10 ;  ++l)
        {
            for(int j = 0 ; j<4; ++j)
            {
                buffer[k]=M[i][j];
                outfile<<buffer[k];

                k++;
            }
            if(k == 40)
            {
                break;
            }
            i++;
        }

        outfile<<"\n";
    }
    else if(SI == 3)        //Terminate
    {

        outfile<<"\n";
        outfile<<"\n";
    }
}

void OSCP::Execute(){
    while(true){
        for(int i=0;i<4;i++){
            IR[i] = M[IC][i];
        }
        IC++;
        if(IR[0] == 'G' && IR[1] == 'D')    //For Get Data
        {
            SI = 1;
            MOS();
        }
        else if(IR[0] == 'P' && IR[1] == 'D')    // For Print Data
        {
            SI = 2;
            MOS();
        }
        else if(IR [0] == 'H')      //For Hault
        {
            SI = 3;
            MOS();
            break;
        }
        else if(IR[0] == 'L' && IR[1] == 'R')       //for load Register
        {
            int i = IR[2]-48;               //here we take input in string for
eg LR20 so to convert 20 into integer ie ascii of 2 is 50 - 48(ascii of 0)
            i = i*10 + ( IR[3]-48);

            for(int j=0;j<=3;j++)           //loading the contents of memory
into register(general purpose)
                R[j]=M[i][j];

            //for(int j=0;j<=3;j++)
              // cout<<R[j];

            cout<<endl;
        }
        else if(IR[0] == 'S' && IR[1] == 'R')       //for Store Register
        {
            int i = IR[2]-48;               //same as above
            i = i*10 +( IR[3]-48) ;
            //cout<<i;
            for(int j=0;j<=3;j++)           //loading from register to memory
                M[i][j]=R[j];

            cout<<endl;
        }
        else if(IR[0] == 'C' && IR[1] == 'R')       //for Compare Register
        {
            int i = IR[2]-48;                   //same as above
            i = i*10 + (IR[3] - 48);
            //cout<<i;
            int count=0;

            for(int j=0;j<=3;j++)       //for comparing contents at given pos
with contents present in R
                if(M[i][j] == R[j])
                    count++;

            if(count==4)
                C=true;

            //cout<<C;
        }
        else if(IR[0] == 'B' && IR[1] == 'T')       //for Branch on True
        {
            if(C == true)                       //if comparing is true jump to
given location using IC
            {
                int i = IR[2]-48;
                i = i*10 + (IR[3] - 48);

                IC = i;

            }
        }
    }
}
void OSCP::load(){
    //For loading the contents of buffer into memory
    int x=0;
    do{
        for(int i=0;i<=39;i++){   //clearing the buffer
            buffer[i]='\0';
        }

        infile.getline(buffer,40);

        //reading from buffer
        if(buffer[0]== '$' && buffer[1]=='A' && buffer[2]=='M' &&
buffer[3]=='J'){
            init();
        }
        else if(buffer[0]== '$' && buffer[1]=='D' && buffer[2]=='T' &&
buffer[3]=='A'){
            IC=0;
            Execute();
        }
        else if(buffer[0]== '$' && buffer[1]=='E' && buffer[2]=='N' &&
buffer[3]=='D'){
            x=0;
            continue;
        }
        else{
            int k=0;

            for( ; x<100 ; ++x){
                for(int j=0;j<4;++j){
                    M[x][j] = buffer[k];
                    k++;
                }

                if(k==40 || buffer[k]== ' ' || buffer[k]=='\n'){
                    break;
                }
            }
            for(int i=0;i<10;i++){    //here 10 bcz of memory is in block of 
10
                cout<<"M["<<i<<"] :";
                for(int j=0;j<4;j++){
                    cout<<M[i][j];
                }
                cout<<endl;
            }
        }

    }
    while(!infile.eof());   //Till end of the file
}

int main(){
    OSCP os;
    os.infile.open("input.txt", ios::in);
    os.outfile.open("output.txt", ios::out);

    if(!os.infile)
    {
        cout<<"Failure"<<endl;
    }
    else
    {
        cout<<"File Exist"<<endl;
    }

    os.load();
    return 0;

}
```

OS Phase 2

ASSUMPTIONS:

1. Jobs may have program errors
2. PI interrupt for program errors introduced
3. No physical separation between jobs
4. Job outputs separated in output file by 2 blank lines
5. Paging introduced, page table stored in real memory
6. Program pages allocated one of 30 memory block using random number generator
7. Load and run one program at a time
8. Time limit, line limit, out-of-data errors introduced
9. TI interrupt for time-out error introduced
10. 2-line messages printed at termination

NOTATION

1. M: memory
2. IR: Instruction Register (4 bytes)
3. IR [1, 2]: Bytes 1, 2 of IR/Operation Code
4. IR [3, 4]: Bytes 3, 4 of IR/Operand Address
5. M[&]: Content of memory location &
6. IC: Instruction Counter Register (2 bytes)
7. R: General Purpose Register (4 bytes)
8. C: Toggle (1 byte)
9. PTR: Page Table Register (4 bytes)
10. PCB: Process Control Block (data structure)
11. VA: Virtual Address
12. RA: Real Address
13. TTC: Total Time Counter
14. LLC: Line Limit Counter
15. TTL: Total Time Limit
16. TLL: Total Line Limit
17. EM: Error Message
18. : Loaded/stored/placed into

INTERRUPT VALUES

1. SI = 1 on GD
   = 2 on PD
   = 3 on H
2. TI = 2 on Time Limit Exceeded
3. PI = 1 Operation Error
   = 2 Operand Error
   = 3 Page Fault

Error Message Coding

EM	Error
0	No Error
1	Out of Data
2	Line Limit Exceeded
3	Time Limit Exceeded
4	Operation Code Error
5	Operand Error
6	Invalid Page Fault

Here's a breakdown of the code:

The code defines a class called OS, which represents the operating system.

1. It includes various member variables such as **memory arrays (M), registers (IR, R, IC, SI, C, PI, TI, PTR), and a flag (breakFlag)** to control program execution.

2. The PCB struct represents the Process Control Block and stores information about a process.

3. The code defines several member functions of the OS class, including init(), load(), addressMap(), read(), write(), execute_user_program(), MOS(), start_execution(), and Terminate().
4. The **init()** function initializes the OS by resetting all the member variables to their initial values.

5. The **load()** function loads a program into memory by parsing the input file and storing instructions and data in the appropriate memory locations.

6. The **addressMap()** function maps a virtual address to a real address using a page table.

7. The **read()** function reads a line from the input file into a buffer and stores it in memory.

8. The **write()** function writes lines from memory to an output file.

9. The **execute_user_program()** function executes the user program instructions stored in memory.

10. The **MOS()** function handles interrupts and system calls by checking the interrupt and system interrupt flags and executing the corresponding operations.

11. The **start_execution()** function sets the instruction counter (IC) to 0 and starts the execution of the user program.

12. The **Terminate()** function terminates the program and prints appropriate messages based on the termination status.

Overall, this code provides a basic simulation of an operating system with virtual memory and handles system calls, interrupts, and program execution. However, without the complete context and the missing parts of the code (e.g., the main function, input/output file handling), it is difficult to provide a comprehensive analysis or determine the correctness and functionality of the code.

Code:

```
#include <iostream>
#include <fstream>
#include <string.h>
#include <time.h>
using namespace std;

ifstream fin("input.txt");
ofstream fout("output.txt");

char M[300][4]; // Physical Memory
char buffer[40];
char IR[4]; // Instruction Register (4 bytes)
char R[5];  // General Purpose Register (4 bytes)
int IC;     // Instruction Counter Register (2 bytes)
int C;      // Toggle (1 byte)
int SI;     // System Interrupt
int PI;     // Program Interrupt
int TI;     // Timing Interrupt
int PTR;    // Page Table Register
bool breakFlag;

struct PCB
{ // Process Control Block
    int job_id;
    int TTL; // Total TIme Limit
    int TLL; // Total Line Limit
    int TTC; // Total Time Count
    int LLC; // Total Line Count
    void setPCB(int id, int ttl, int tll)
    {
        // Function to set the PCB with the following
        job_id = id;
        TTL = ttl;
        TLL = tll;
        TTC = 0;
        LLC = 0;
    }
};

PCB pcb;

string error[7] = {"No Error", "Out of Data", "Line Limit Exceeded", "Time
Limit Exceeded",
                   "Operation Code Error", "Operand Error", "Invalid Page
Fault"};

void init();
void read(int RA);
void write(int RA);
int addressMap(int VA);
void execute_user_program();
void start_execution();
int allocate();
void load();

void init()
{
    memset(M, '\0', 1200);
    memset(IR, '\0', 4);
    memset(R, '\0', 5);
    C = 0;
    SI = 0;
    PI = 0;
    TI = 0;
    breakFlag = false;
}

void Terminate(int EM, int EM2 = -1)
{
    fout << endl
         << endl;
    // If the program terminated normally, print the appropriate message
    if (EM == 0)
    {
        fout << " terminated normally. " << error[EM] << endl;
    }
    // If the program terminated abnormally, print the appropriate message
    else
    {
        fout << "terminated abnormally due to " << error[EM] << (EM2 != -1 ?
(". " + error[EM2]) : "") << endl;
        fout << "Job Id = " << pcb.job_id << ", IC = " << IC << ", IR = " <<
IR << ", C = " << C << ", R = " << R << ", TTL = " << pcb.TTL << ", TTC = " <<
pcb.TTC << ", TLL = " << pcb.TLL << ", LLC = " << pcb.LLC;
    }
}
void read(int RA)
{
    // Read a line from the input file into a buffer
    fin.getline(buffer, 41);
    // Check if the buffer contains the end-of-job marker
    char temp[5];
    memset(temp, '\0', 5);
    memcpy(temp, buffer, 4);
    // If the end-of-job marker is found, terminate the program
    if (!strcmp(temp, "$END"))
    {
        Terminate(1);
        breakFlag = true;
    }
    // Otherwise, store the line in memory at the specified location
    else
    {
        strcpy(M[RA], buffer);
    }
}

void write(int RA)
{
    // Check if the program has exceeded its line limit
    if (pcb.LLC + 1 > pcb.TLL)
    {
        Terminate(2);
        breakFlag = true;
    }
    // If not, read the specified number of lines from memory and write to
output file
    else
    {
        char str[40];
        int k = 0;
        for (int i = RA; i < (RA + 10); i++)
        {
            for (int j = 0; j < 4; j++)
                str[k++] = M[i][j];
        }
        fout << str << endl;
        pcb.LLC++;
    }
}

int mos(int RA = 0)
{
    // Check if the timer interrupt (TI) is set to 0
    if (TI == 0)
    {
        // If the system interrupt (SI) is set to a non-zero value, handle the
system call
        if (SI != 0)
        {
            switch (SI)
            {
            case 1:
                // Read from input device
                read(RA);
                break;
            case 2:
                // Write to output device
                write(RA);
                break;
            case 3:
                // Terminate the program
                Terminate(0);
                breakFlag = true;
                break;
            default:
                cout << "Error with SI." << endl;
            }
            // Reset the system interrupt (SI) to 0 after handling the system
call
            SI = 0;
        }
        // If the program interrupt (PI) is set to a non-zero value, handle
the interrupt
        else if (PI != 0)
        {
            switch (PI)
            {
            case 1:
                // Read/write to an illegal memory location
                Terminate(4);
                breakFlag = true;
                break;
            case 2:
                // Handle page fault
                Terminate(5);
                breakFlag = true;
                break;
            case 3:
                // Reset the program interrupt (PI) to 0
                PI = 0;
                // Page Fault checking
                char temp[3];
                memset(temp, '\0', 3);
                memcpy(temp, IR, 2);

                // If the instruction is a GD or SR instruction, allocate a
new page frame and update the page table register (PTR)
                if (!strcmp(temp, "GD") || !strcmp(temp, "SR"))
                {
                    int m;
                    do
                    {
                        m = allocate();
                    } while (M[m * 10][0] != '\0');
                    int currPTR = PTR;
                    while (M[currPTR][0] != '*')
                        currPTR++;

                    itoa(m, M[currPTR], 10);

                    cout << "Valid Page Fault, page frame = " << m << endl;
                    cout << "PTR = " << PTR << endl;
                    for (int i = 0; i < 300; i++)
                    {
                        cout << "M[" << i << "] :";
                        for (int j = 0; j < 4; j++)
                        {
                            cout << M[i][j];
                        }
                        cout << endl;
                    }
                    cout << endl;

                    // If the time limit (TTL) is exceeded, set the timer
interrupt (TI) to 2 and handle the interrupt
                    if (pcb.TTC + 1 > pcb.TTL)
                    {
                        TI = 2;
                        PI = 3;
                        mos();
                        break;
                    }
                    pcb.TTC++;

                    // Return 1 to indicate that a page fault occurred
                    return 1;
                }

                // If the instruction is a PD, LR, H, CR, or BT instruction,
terminate the program and handle the interrupt
                else if (!strcmp(temp, "PD") || !strcmp(temp, "LR") ||
!strcmp(temp, "H") || !strcmp(temp, "CR") || !strcmp(temp, "BT"))
                {
                    Terminate(6);
                    breakFlag = true;
                    if (pcb.TTC + 1 > pcb.TTL)
                    {
                        TI = 2;
                        PI = 3;
                        mos();
                        break;
                    }
                    // pcb.TTC++;
                }
                else
                {
                    PI = 1;
                    mos();
                }
                return 0;
            default:
                cout << "Error with PI." << endl;
            }
            PI = 0;
        }
    }
    else
    {
        if (SI != 0)
        {
            switch (SI)
            {
            case 1:
                Terminate(3);
                breakFlag = true;
                break;
            case 2:
                write(RA);
                if (!breakFlag)
                    Terminate(3);
                break;
            case 3:
                Terminate(0);
                breakFlag = true;
                break;
            default:
```

```cpp
                cout << "Error with SI." << endl;
            }
            SI = 0;
        }
        else if (PI != 0)
        {
            switch (PI)
            {
            case 1:
                Terminate(3, 4);
                breakFlag = true;
                break;
            case 2:
                Terminate(3, 5);
                breakFlag = true;
                break;
            case 3:
                Terminate(3);
                breakFlag = true;
                break;
            default:
                cout << "Error with PI." << endl;
            }
            PI = 0;
        }
    }

    return 0;
}

int addressMap(int VA)
{
    // Funtion to accept Virtual Address and return Real Address
    //   0 < VA < 100
    if (0 <= VA && VA < 100)
    {
        int pte = PTR + VA / 10; // pte = main memory location
        if (M[pte][0] == '*')
        {
            PI = 3;
            return 0;
        }
        cout << "In addressMap(), VA = " << VA << ", pte = " << pte << ",
M[pte] = " << M[pte] << endl;
        return atoi(M[pte]) * 10 + VA % 10; // Real Address
    }
    PI = 2;
    return 0;
}

void execute_user_program()
{
    char temp[3], loca[2];
    int locIR, RA;

    while (true)
    {
        if (breakFlag)
            break;

        RA = addressMap(IC); // GD10 here 10 is also a virtual address so
convert to real address
        if (PI != 0)
        {
            if (mos())
            {
                continue;
            }
            break;
        }
        cout << "IC = " << IC << ", RA = " << RA << endl;
        memcpy(IR, M[RA], 4); // Memory to IR, instruction fetched
        IC += 1;

        memset(temp, '\0', 3);
        memcpy(temp, IR, 2);
        for (int i = 0; i < 2; i++)
        {
            if (!((47 < IR[i + 2] && IR[i + 2] < 58) || IR[i + 2] == 0))
            {
                PI = 2;
                break;
            }
            loca[i] = IR[i + 2];
        }

        if (PI != 0)
        {
            mos();
            break;
        }

        // loca[0] = IR[2];
        // loca[1] = IR[3];
        locIR = atoi(loca);

        RA = addressMap(locIR);
        if (PI != 0)
        {
            if (mos())
            {
                IC--;
                continue;
            }
            break;
        }

        cout << "IC = " << IC << ", RA = " << RA << ", IR = " << IR << endl;
        if (pcb.TTC + 1 > pcb.TTL)
        {
            TI = 2;
            PI = 3;
            mos();
            break;
        }

        if (!strcmp(temp, "LR"))
        {
            memcpy(R, M[RA], 4);
            pcb.TTC++;
        }
        else if (!strcmp(temp, "SR"))
        {
            memcpy(M[RA], R, 4);
            pcb.TTC++;
        }
        else if (!strcmp(temp, "CR"))
        {
            if (!strcmp(R, M[RA]))
                C = 1;
            else
                C = 0;
            pcb.TTC++;
        }
        else if (!strcmp(temp, "BT"))
        {
            if (C == 1)
                IC = RA;
            pcb.TTC++;
        }
        else if (!strcmp(temp, "GD"))
        {
            SI = 1;
            mos(RA);
            pcb.TTC++;
        }
        else if (!strcmp(temp, "PD"))
        {
            SI = 2;
            mos(RA);
            pcb.TTC++;
        }
        else if (!strcmp(temp, "H"))
        {
            SI = 3;
            mos();
            pcb.TTC++;
            break;
        }
        else
        {
            PI = 1;
            mos();
            break;
        }
        memset(IR, '\0', 4);
    }
}
```

```
void start_execution()
{
    IC = 0;
    execute_user_program();
}

int allocate()
{
    // Function to randomly generate a number. This number will be reserved
for page table
    srand(time(0));
    return (rand() % 30); // %30 bcz we want the random number between 0 to 29
as we have 30 frames in main memory
}

void load()
{
    int m;                      // Variable to hold memory location
    int currPTR;                // Points to the last empty location in Page
Table Register
    char temp[5];               // Temporary Variable to check for $AMJ, $DTA,
$END
    memset(buffer, '\0', 40); // to make the buffer empty by assigning it with
'\0'

    while (!fin.eof())
    {
        fin.getline(buffer, 41);
        memset(temp, '\0', 5);
        memcpy(temp, buffer, 4);

        if (!strcmp(temp, "$AMJ"))
        {
            init();

            int jobId, TTL, TLL;         //$AMJ 0001 0002 0003
            memcpy(temp, buffer + 4, 4); //+4 bcz jobID
            jobId = atoi(temp);
            memcpy(temp, buffer + 8, 4); //+8 bcz Total time limit
            TTL = atoi(temp);
            memcpy(temp, buffer + 12, 4); //+12 bcz Total Line Limit
            TLL = atoi(temp);
            pcb.setPCB(jobId, TTL, TLL); // set PCB with all the parameters

            PTR = allocate() * 10;   // PTR- Page Table Register it points to 
the starting address.
            memset(M[PTR], '*', 40); // TO make memory assigned with '*'
            currPTR = PTR;
        }
        else if (!strcmp(temp, "$DTA"))
        {
            start_execution();
        }
        else if (!strcmp(temp, "$END"))
        {
            continue;
        }
        else
        {
            if (breakFlag)
                continue;

            do
            {
                m = allocate();
            } while (M[m * 10][0] != '\0'); // To check if the frame is free

            itoa(m, M[currPTR], 10);
            currPTR++;

            strcpy(M[m * 10], buffer); // *10 bcz page size is 10

            cout << "PTR = " << PTR << endl;
            for (int i = 0; i < 300; i++)
            {
                cout << "M[" << i << "] :";
                for (int j = 0; j < 4; j++)
                {
                    cout << M[i][j];
                }
                cout << endl;
            }
            cout << endl;
        }
    }
}

int main()
{
    load();
    fin.close();
    fout.close();
    return 0;
}
```