System calls

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h> // For pid\_t

#include <sys/wait.h>

#include <fcntl.h> // For file related system calls

#include <string.h>

#include <sys/stat.h> // for stat

int main() {

putenv("TERM=xterm");

//indicates the type of terminal being used

while (1) {

printf("\033[H\033[J"); // Clear the terminal screen and move cursor

printf("Lab 4: System Call Demonstration\n");

printf("1. Process Related System Calls\n");

printf("2. File Related System Calls\n");

printf("3. Communication System Calls\n");

printf("4. Information Related System Calls\n");

printf("5. Exit\n");

int choice;

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: // Process Related System Calls

printf("1. fork\n");

printf("2. exit\n");

printf("3. wait\n");

int sub\_choice;

printf("Enter sub-choice: ");

scanf("%d", &sub\_choice);

switch (sub\_choice) {

case 1:

printf("Executing fork system call...\n");

pid\_t child\_pid = fork();

if (child\_pid == 0) {

printf("This is the child process.\n");

} else if (child\_pid > 0) {

printf("This is the parent process. Child PID: %d\n", child\_pid);

} else {

printf("Fork failed.\n");

}

break;

case 2:

printf("Executing exit system call...\n");

exit(0);

break;

case 3:

printf("Executing wait system call...\n");

wait(NULL);

// system call is invoked to make the parent process wait until any child process terminates.

printf("Child process has completed.\n");

break;

default:

printf("Invalid sub-choice\n");

break;

}

break;

case 2: // File Related System Calls

printf("1. open\n");

printf("2. read\n");

printf("3. write\n");

printf("4. close\n");

printf("5. link\n");

printf("6. unlink\n");

printf("7. stat\n");

printf("Enter sub-choice: ");

scanf("%d", &sub\_choice);

switch (sub\_choice) {

case 1:

printf("Executing open system call...\n");

int fd = open("sample.txt", O\_CREAT | O\_WRONLY, 0644);

// (O\_CREAT creates the file if it doesn't exist, and O\_WRONLY opens for writing only).

if (fd != -1) {

printf("File opened for writing.\n");

close(fd);

} else {

printf("Error opening file.\n");

}

break;

case 2:

printf("Executing read system call...\n");

fd = open("sample.txt", O\_RDONLY);

if (fd != -1) {

char content[100];

ssize\_t bytesRead = read(fd, content, sizeof(content));

// types of data present in systypes

// checks if the file is not empty

if (bytesRead > 0) {

content[bytesRead] = '\0';

printf("Content read from file: %s\n", content);

} else {

printf("Error reading file.\n");

}

close(fd);

} else {

printf("Error opening file.\n");

}

break;

case 3:

printf("Executing write system call...\n");

fd = open("sample.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd != -1) {

write(fd, "Hello, world!", 13);

// file ptr, content, size of content

printf("Content written to file.\n");

close(fd);

} else {

printf("Error opening file.\n");

}

break;

case 4:

printf("Executing close system call...\n");

close(sub\_choice); // Replace sub\_choice with actual file descriptor

printf("File closed.\n");

break;

// Hard links allow multiple filenames to refer to the same physical file content.

case 5:

printf("Executing link system call...\n");

link("sample.txt", "sample\_link.txt");

printf("Hard link created.\n");

break;

case 6:

printf("Executing unlink system call...\n");

unlink("sample\_link.txt");

printf("Hard link removed.\n");

break;

case 7:

printf("Executing stat system call...\n");

struct stat st;

if (stat("sample.txt", &st) == 0) {

printf("File Size: %lld bytes\n", (long long)st.st\_size);

printf("File Permissions: %o\n", st.st\_mode & 0777);

} else {

printf("Error getting file information.\n");

}

break;

default:

printf("Invalid sub-choice\n");

break;

}

break;

case 3: // Communication System Calls

printf("1. pipe\n");

printf("2. fifo\n");

printf("Enter sub-choice: ");

scanf("%d", &sub\_choice);

switch (sub\_choice) {

case 1:

printf("Executing pipe system call...\n");

int pipe\_fd[2];

if (pipe(pipe\_fd) == 0) {

printf("Pipe created.\n");

// pipe\_fd[0] representing the read end and pipe\_fd[1] representing the write end.

close(pipe\_fd[0]);

close(pipe\_fd[1]);

} else {

printf("Error creating pipe.\n");

}

break;

case 2:

printf("Executing fifo system call...\n");

mkfifo("myfifo", 0666);

printf("FIFO created.\n");

break;

// Pipes are unidirectional communication channels, and FIFOs (named pipes) provide named communication channels between processes.

default:

printf("Invalid sub-choice\n");

break;

}

break;

case 4: // Information Related System Calls

printf("Executing information related system call...\n");

printf("Current User: %s\n", getenv("USER"));

system("date");

break;

case 5:

printf("Exiting...\n");

exit(0);

default:

printf("Invalid choice\n");

break;

}

printf("Press Enter to continue...");

getchar(); // Consume the newline character left in the input buffer

getchar(); // Wait for user to press Enter

}

return 0;

}

Reader writer (Mutex)  
  
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#ifdef \_WIN32

#include <windows.h>

#define sleep\_ms(ms) Sleep(ms)

#else

#include <unistd.h>

#define sleep\_ms(ms) usleep((ms) \* 1000)

#endif

pthread\_mutex\_t x, wsem;

pthread\_t tid;

int readcount;

void initialize() {

pthread\_mutex\_init(&x, NULL);

pthread\_mutex\_init(&wsem, NULL);

readcount = 0;

}

void \*reader(void \*param) {

int waittime;

waittime = rand() % 5;

pthread\_t tid = pthread\_self();

printf("\nReader with thread id: %lu is trying to enter", tid);

pthread\_mutex\_lock(&x);

readcount++;

if (readcount == 1)

pthread\_mutex\_lock(&wsem);

printf("\nReader with thread id: %lu is inside (%d readers)", tid, readcount);

pthread\_mutex\_unlock(&x);

sleep\_ms(waittime \* 1000); // Sleep in milliseconds

pthread\_mutex\_lock(&x);

readcount--;

if (readcount == 0)

pthread\_mutex\_unlock(&wsem);

pthread\_mutex\_unlock(&x);

printf("\nReader with thread id: %lu has left", tid);

return NULL;

}

void \*writer(void \*param) {

int waittime;

waittime = rand() % 2; // Reduce the sleep time for writers

pthread\_t tid = pthread\_self();

printf("\nWriter with thread id: %lu is trying to enter", tid);

pthread\_mutex\_lock(&wsem);

printf("\nWriter with thread id: %lu has entered", tid);

sleep\_ms(waittime \* 1000); // Sleep in milliseconds

pthread\_mutex\_unlock(&wsem);

printf("\nWriter with thread id: %lu has left", tid);

sleep\_ms(30000); // Sleep for 30 seconds

exit(0);

}

int main() {

int n1, n2, i;

printf("\n----------------------------- ");

printf("\nInput the number of readers: ");

scanf("%d", &n1);

printf("\nInput the number of writers: ");

scanf("%d", &n2);

printf("\n----------------------------- ");

// Initialize the mutexes

initialize();

for (i = 0; i < n1; i++)

pthread\_create(&tid, NULL, reader, NULL);

for (i = 0; i < n2; i++)

pthread\_create(&tid, NULL, writer, NULL);

// Sleep for 30 seconds to allow threads to run

sleep\_ms(30000);

// Cancel the threads to exit the infinite loops

pthread\_cancel(tid);

// Wait for all threads to finish

pthread\_join(tid, NULL);

// Clean up mutexes

pthread\_mutex\_destroy(&x);

pthread\_mutex\_destroy(&wsem);

return 0;

}

Reader writer semaphore  
  
#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

sem\_t x, y;

pthread\_t readerthreads[100], writerthreads[100];

int readercount = 0;

void\* reader(void\* param)

{

sem\_wait(&x);

readercount++;

if (readercount == 1)

sem\_wait(&y);

sem\_post(&x);

printf("%lu thread id Reader is inside\n", pthread\_self());

usleep(300000); // usleep is in microseconds

sem\_wait(&x);

readercount--;

if (readercount == 0)

{

sem\_post(&y);

}

sem\_post(&x);

printf("%lu thread id Reader has left\n", pthread\_self());

return NULL;

}

void\* writer(void\* param)

{

printf("Writer with thread id : %lu is checking to enter\n", pthread\_self());

sem\_wait(&y);

printf("Writer with thread id : %lu is writing\n", pthread\_self());

usleep(300000); // usleep is in microseconds

sem\_post(&y);

printf("Writer with thread id : %lu has finished writing \n", pthread\_self());

return NULL;

}

int main()

{

int num\_readers, num\_writers;

printf("Enter the number of readers: ");

scanf("%d", &num\_readers);

printf("Enter the number of writers: ");

scanf("%d", &num\_writers);

sem\_init(&x, 0, 1);

sem\_init(&y, 0, 1);

for (int i = 0; i < num\_readers; i++)

{

pthread\_create(&readerthreads[i], NULL, reader, NULL);

}

for (int i = 0; i < num\_writers; i++)

{

pthread\_create(&writerthreads[i], NULL, writer, NULL);

}

for (int i = 0; i < num\_readers; i++)

{

pthread\_join(readerthreads[i], NULL);

}

for (int i = 0; i < num\_writers; i++)

{

pthread\_join(writerthreads[i], NULL);

}

sem\_destroy(&x);

sem\_destroy(&y);

return 0;

}

Producer consumer (mutex)  
  
Producer consumer (semaphore)  
  
Dining philosopher (mutex)  
  
Dining philosopher (semaphore)  
  
Cpu scheduling

Page replacement

#include <iostream>

#include <vector>

#include <limits>

#include <sstream>

using namespace std;

// FIFO (First-In, First-Out) Page Replacement Algorithm

int fifo(vector<int>& reference, int frames) {

vector<int> frame(frames, -1);

int page\_faults = 0;

int index = 0;

for (int i = 0; i < reference.size(); i++) {

int page = reference[i];

int is\_fault = 1;

for (int j = 0; j < frames; j++) {

if (frame[j] == page) {

is\_fault = 0;

break;

}

}

if (is\_fault) {

page\_faults++;

frame[index] = page;

index = (index + 1) % frames;

}

}

return page\_faults;

}

// LRU (Least Recently Used) Page Replacement Algorithm

int lru(vector<int>& reference, int frames) {

vector<int> frame(frames, -1);

vector<int> counter(frames, 0);

int page\_faults = 0;

for (int i = 0; i < reference.size(); i++) {

int page = reference[i];

int is\_fault = 1;

for (int j = 0; j < frames; j++) {

if (frame[j] == page) {

is\_fault = 0;

counter[j] = i;

break;

}

}

if (is\_fault) {

page\_faults++;

int min\_count = counter[0], replace = 0;

for (int j = 1; j < frames; j++) {

if (counter[j] < min\_count) {

min\_count = counter[j];

replace = j;

}

}

frame[replace] = page;

counter[replace] = i;

}

}

return page\_faults;

}

// Optimal Page Replacement Algorithm

int optimal(vector<int>& reference, int frames) {

vector<int> frame(frames, -1);

int page\_faults = 0;

for (int i = 0; i < reference.size(); i++) {

int page = reference[i];

int is\_fault = 1;

for (int j = 0; j < frames; j++) {

if (frame[j] == page) {

is\_fault = 0;

break;

}

}

if (is\_fault) {

page\_faults++;

int replace = -1;

for (int j = 0; j < frames; j++) {

int k;

for (k = i + 1; k < reference.size(); k++) {

if (reference[k] == frame[j]) {

if (k > replace)

replace = k;

break;

}

}

if (k == reference.size()) {

replace = j;

break;

}

}

frame[replace] = page;

}

}

return page\_faults;

}

int main() {

int frames, pages;

cout << "Enter the number of frames: ";

cin >> frames;

cout << "Enter the number of pages: ";

cin >> pages;

cin.ignore(); // Consume the newline character.

vector<int> reference;

cout << "Enter the reference string (space-separated): ";

string input;

getline(cin, input);

istringstream iss(input);

int page;

while (iss >> page) {

reference.push\_back(page);

}

int fifo\_faults = fifo(reference, frames);

int lru\_faults = lru(reference, frames);

int optimal\_faults = optimal(reference, frames);

cout << "Total page faults using FIFO: " << fifo\_faults << endl;

cout << "Total page faults using LRU: " << lru\_faults << endl;

cout << "Total page faults using Optimal: " << optimal\_faults << endl;

return 0;

}

Bankers algorithm

#include<stdio.h>

void main() {

int max[10][10], alloc[10][10], need[10][10], avail[10], np, nr;

printf("Enter the number of processes: ");

scanf("%d", &np);

printf("Enter the number of resources: ");

scanf("%d", &nr);

printf("Enter the allocation matrix:\n");

for(int i=0; i<np; i++)

for(int j=0; j<nr; j++)

scanf("%d", &alloc[i][j]);

printf("Enter the max matrix:\n");

for(int i=0; i<np; i++)

for(int j=0; j<nr; j++)

scanf("%d", &max[i][j]);

printf("Enter the available resources:\n");

for(int i=0; i<nr; i++)

scanf("%d", &avail[i]);

for(int i=0; i<np; i++)

for(int j=0; j<nr; j++)

need[i][j] = max[i][j] - alloc[i][j];

int finish[10], safeSeq[10];

for(int i=0; i<np; i++)

finish[i] = 0;

int work[10];

for(int i=0; i<nr; i++)

work[i] = avail[i];

int count=0;

while(count < np) {

int found = 0;

for(int p=0; p<np; p++) {

if(finish[p] == 0) {

int j;

for(j=0; j<nr; j++) {

if(need[p][j] > work[j])

break;

}

if(j == nr) {

for(int k=0 ; k<nr ; k++)

work[k] += alloc[p][k];

safeSeq[count++] = p;

finish[p] = 1;

found = 1;

}

}

}

if(found == 0) {

printf("System is not in safe state due to process: %d\n", count);

return;

}

}

printf("System is in safe state.\nSafe sequence is: ");

for(int i=0; i<np; i++)

printf("%d ",safeSeq[i]);

}

Matrix multiplication

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define MATRIX\_SIZE 3

#define NUM\_THREADS 3

int A[MATRIX\_SIZE][MATRIX\_SIZE];

int B[MATRIX\_SIZE][MATRIX\_SIZE];

int C[MATRIX\_SIZE][MATRIX\_SIZE];

int D[MATRIX\_SIZE][MATRIX\_SIZE];

int E[MATRIX\_SIZE][MATRIX\_SIZE];

pthread\_t threads[NUM\_THREADS];

struct ThreadArgs {

int start\_row;

int end\_row;

};

void \*addMatrices(void \*args) {

struct ThreadArgs \*threadArgs = (struct ThreadArgs \*)args;

for (int i = threadArgs->start\_row; i < threadArgs->end\_row; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

C[i][j] = A[i][j] + B[i][j];

}

}

pthread\_exit(NULL);

}

void \*subMatrices(void \*args) {

struct ThreadArgs \*threadArgs = (struct ThreadArgs \*)args;

for (int i = threadArgs->start\_row; i < threadArgs->end\_row; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

E[i][j] = A[i][j] - B[i][j];

}

}

pthread\_exit(NULL);

}

void \*multiplyMatrices(void \*args) {

struct ThreadArgs \*threadArgs = (struct ThreadArgs \*)args;

for (int i = threadArgs->start\_row; i < threadArgs->end\_row; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

D[i][j] = 0;

for (int k = 0; k < MATRIX\_SIZE; ++k) {

D[i][j] += A[i][k] \* B[k][j];

}

}

}

pthread\_exit(NULL);

}

int main() {

// Input matrices A and B

printf("Enter elements of matrix A:\n");

for (int i = 0; i < MATRIX\_SIZE; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

scanf("%d", &A[i][j]);

}

}

printf("Enter elements of matrix B:\n");

for (int i = 0; i < MATRIX\_SIZE; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

scanf("%d", &B[i][j]);

}

}

// Create thread arguments and divide work

struct ThreadArgs threadArgs[NUM\_THREADS];

int rows\_per\_thread = MATRIX\_SIZE / NUM\_THREADS;

int start\_row = 0;

for (int i = 0; i < NUM\_THREADS; ++i) {

threadArgs[i].start\_row = start\_row;

threadArgs[i].end\_row = start\_row + rows\_per\_thread;

start\_row += rows\_per\_thread;

}

// Create threads for addition

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_create(&threads[i], NULL, addMatrices, (void \*)&threadArgs[i]);

}

// Wait for addition threads to finish

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Print the result matrix C (addition)

printf("Matrix C (Addition):\n");

for (int i = 0; i < MATRIX\_SIZE; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

printf("%d ", C[i][j]);

}

printf("\n");

}

// Create threads for multiplication

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_create(&threads[i], NULL, multiplyMatrices, (void \*)&threadArgs[i]);

}

// Wait for multiplication threads to finish

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Print the result matrix D (multiplication)

printf("Matrix D (Multiplication):\n");

for (int i = 0; i < MATRIX\_SIZE; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

printf("%d ", D[i][j]);

}

printf("\n");

}

//substraction

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_create(&threads[i], NULL, subMatrices, (void \*)&threadArgs[i]);

}

// Wait for addition threads to finish

for (int i = 0; i < NUM\_THREADS; ++i) {

pthread\_join(threads[i], NULL);

}

// Print the result matrix C (addition)

printf("Matrix E (Substraction):\n");

for (int i = 0; i < MATRIX\_SIZE; ++i) {

for (int j = 0; j < MATRIX\_SIZE; ++j) {

printf("%d ", E[i][j]);

}

printf("\n");

}

return 0;

}

Disk scheduling

Phase - 1

#include <algorithm>

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

class VM

{

private:

char buffer[40];

char Memory[100][4];

char IR[4]; // Instruction Register sa

char R[4]; // General purpose Register

bool C; // Toggle Register

int IC; // Instruction counter

int SI; // System Interrupt

fstream infile;

ofstream outfile;

// Initialize all the variables (buffer, memory, IR, R, IC, C, SI)

void init(){

fill(buffer, buffer + sizeof(buffer), '\0');

fill(&Memory[0][0], &Memory[0][0] + sizeof(Memory), '\0');

fill(IR, IR + sizeof(IR), '\0');

fill(R, R + sizeof(R), '\0');

IC = 0;

C = true;

SI = 0;

}

// Reset buffer to null

void resetBuffer(){

fill(buffer, buffer + sizeof(buffer), '\0');

}

// Master Mode

void MOS()

{

switch (SI)

{

case 1:

READ();

break;

case 2:

WRITE();

break;

case 3:

TERMINATE();

break;

}

SI = 0;

}

void LOAD(){

if (infile.is\_open())

{

string s;

while (getline(infile, s))

{

if (s[0] == '$' && s[1] == 'A' && s[2] == 'M' && s[3] == 'J'){

init();

cout<<"New Job started\n";

}

else if (s[0] == '$' && s[1] == 'D' && s[2] == 'T' && s[3] == 'A'){

cout << "Data card loding\n";

resetBuffer();

STARTEXE();

}

else if (s[0] == '$' && s[1] == 'E' && s[2] == 'N' && s[3] == 'D'){

cout << "END of Job\n";

}

else{

cout << "Program Card loding\n";

int length = s.size();

resetBuffer(); //restting the buffer to null for new instructoins

// Buffer <-- Program Card

for (int i = 0; i < length; i++){

buffer[i] = s[i];

}

int buff = 0;

int ref = 0;

// Memory <-- Buffer

while (buff < 40 && buffer[buff] != '\0'){

for (int j = 0; j < 4; j++){

if (buffer[buff] == 'H'){

Memory[IC][j] = 'H';

buff++;

break;

}

Memory[IC][j] = buffer[buff];

buff++;

}

IC++;

}

}

for(int i = 0; i<100; i++)

{

cout<<"M["<<i<<"]\t";

for(int j = 0; j<4; j++ )

{

cout<<Memory[i][j];

}

cout<<endl;

}

}

infile.close(); // Input File closed

}

}

void STARTEXE(){

IC = 0;

EXECUTEUSERPROGRAM();

}

void READ()

{

cout << "Read function called\n";

string data;

getline(infile, data);

int len = data.size();

for (int i = 0; i < len; i++)

{

buffer[i] = data[i];

}

int buff = 0, mem\_ptr = (IR[2] - '0') \* 10;

while (buff < 40 && buffer[buff] != '\0')

{

for (int i = 0; i < 4; i++)

{

Memory[mem\_ptr][i] = buffer[buff];

buff++;

}

mem\_ptr++;

}

resetBuffer();

}

void WRITE()

{

cout << "Write function called\n";

outfile.open("output.txt", ios::app);

for (int i = (IR[2] - '0') \* 10; i < (IR[2] - '0' + 1) \* 10; i++)

{

for (int j = 0; j < 4; j++)

{

if (Memory[i][j] != '\0')

{

outfile << Memory[i][j];

}

}

}

outfile << "\n";

outfile.close();

}

void TERMINATE()

{

outfile.open("output.txt", ios::app);

cout << "Terminate called\n\n";

outfile << "\n\n";

outfile.close();

}

void EXECUTEUSERPROGRAM(){ // Slave Mode

while (IC < 99 && Memory[IC][0] != '\0'){

for (int i = 0; i < 4; i++){

IR[i] = Memory[IC][i];

}

IC++;

//SI= 1-GD, 2-PD, 3-H

// GD

if (IR[0] == 'G' && IR[1] == 'D'){

SI = 1;

MOS();

}

// PD

else if (IR[0] == 'P' && IR[1] == 'D'){

SI = 2;

MOS();

}

// H

else if (IR[0] == 'H'){

SI = 3;

MOS();

return;

}

// LR - LOAD DATA (R <-- memory[IR[2,3]])

else if (IR[0] == 'L' && IR[1] == 'R'){

for (int i = 0; i < 4; i++){

R[i] = Memory[(IR[2] - '0') \* 10 + (IR[3] - '0')][i];

}

}

// SR - STORE (memory[IR[2,3]] <-- R)

else if (IR[0] == 'S' && IR[1] == 'R'){

for (int i = 0; i < 4; i++){

Memory[(IR[2] - '0') \* 10 + (IR[3] - '0')][i] = R[i];

}

}

// CR - COMPARE(R, memory[IR[2,3]])

else if (IR[0] == 'C' && IR[1] == 'R'){

int cnt = 0;

for (int i = 0; i < 4; i++){

if (Memory[(IR[2] - '0') \* 10 + (IR[3] - '0')][i] == R[i]){

cnt++;

}

}

if (cnt == 4){

C = true;

}

else{

C = false;

}

}

// BT (JUMP if toogle is T)

else if (IR[0] == 'B' && IR[1] == 'T'){

if (C){

IC = (IR[2] - '0') \* 10 + (IR[3] - '0');

}

}

}

}

public:

VM()

{

// infile.open("./example\_job.txt", ios::in);

infile.open("./input\_custom.txt", ios::in);

// infile.open("./input\_Phase1.txt", ios::in);

init();

LOAD();

}

};

int main()

{

VM v;

return 0;

}

Phase - 2

#include <algorithm>

#include <fstream>

#include <iostream>

#include <string>

#include <cstdlib>

#include <time.h>

using namespace std;

class PCB{

public:

int jobID, TTL, TLL;

int TTC,LLC;

};

class VM

{

private:

fstream infile;

ofstream outfile;

char Memory[300][4], buffer[40], IR[4], R[4];

bool C;

int IC;

int SI, PI, TI;

PCB pcb;

int PTR;

int PTE;

int real\_address,virtual\_address;

int pageTable[30];

int pageTablePTR;

int page\_fault\_valid = 0;// 1-Valid 0-Invalid

bool Terminate;

int pageNo;

void init()

{

fill(buffer, buffer + sizeof(buffer), '\0');

fill(&Memory[0][0], &Memory[0][0] + sizeof(Memory), '\0');

fill(IR, IR + sizeof(IR), '\0');

fill(R, R + sizeof(R), '\0');

C = true;

IC = 0;

SI = PI = TI= 0;

pcb.jobID = pcb.TLL = pcb.TTL = pcb.TTC = pcb.LLC = 0;

PTR=PTE=pageNo=-1;

fill(pageTable,pageTable + sizeof(pageTable), 0);

pageTablePTR = 0;

Terminate=false;

}

void restBuffer()

{

fill(buffer, buffer + sizeof(buffer), '\0');

}

int Allocate(){

int pageNo;

bool check=true;

while(check){

pageNo = (rand() % 30) ;

if(pageTable[pageNo]==0){

pageTable[pageNo] = 1;

check=false;

}

}

return pageNo;

}

void MOS()

{

//TI & SI

if (TI == 0 && SI == 1)

{

READ();

}

else if (TI == 0 && SI == 2)

{

WRITE();

}

else if (TI == 0 && SI == 3)

{

TERMINATE(0);

}

else if (TI == 2 && SI == 1)

{

TERMINATE(3);

}

else if (TI == 2 && SI == 2)

{

WRITE();

TERMINATE(3);

}

else if (TI == 2 && SI == 3)

{

TERMINATE(0);

}

// TI & PI

else if (TI == 0 && PI == 1)

{

TERMINATE(4);

}

else if (TI == 0 && PI == 2)

{

TERMINATE(5);

}

else if (TI == 0 && PI == 3)

{

if(page\_fault\_valid == 1)

{

pageNo = Allocate();

Memory[PTE][2] = (pageNo / 10) + '0';

Memory[PTE][3] = (pageNo % 10) + '0';

pageTablePTR++;

PI =0;

}

else

{

TERMINATE(6);

}

}

else if (TI == 2 && PI == 1)

{

TERMINATE(7);

}

else if (TI == 2 && PI == 2)

{

TERMINATE(8);

}

else if (TI == 2 && PI == 3)

{

TERMINATE(3);

}

}

void LOAD()

{

if (infile.is\_open())

{

string s;

while (getline(infile, s))

{

if (s[0] == '$' && s[1] == 'A' && s[2] == 'M' && s[3] == 'J')

{

init();

cout << "New Job started\n";

pcb.jobID = (s[4] - '0') \* 1000 + (s[5] - '0') \* 100 + (s[6] - '0') \* 10 + (s[7] - '0');

pcb.TTL = (s[8] - '0') \* 1000 + (s[9] - '0') \* 100 + (s[10] - '0') \* 10 + (s[11] - '0');

pcb.TLL = (s[12] - '0') \* 1000 + (s[13] - '0') \* 100 + (s[14] - '0') \* 10 + (s[15] - '0');

// Get Frame for Page Table

PTR = Allocate()\*10;

for(int i=PTR;i<PTR+10;i++){

for(int j=0;j<4;j++){

Memory[i][j]='\*';

}

}

cout << "\nAllocated Page is for Page Table: " << PTR / 10 << "\n";

cout << "jobID: " << pcb.jobID << "\nTTL: " << pcb.TTL << "\nTLL: " << pcb.TLL << "\n";

}

else if (s[0] == '$' && s[1] == 'D' && s[2] == 'T' && s[3] == 'A')

{

cout << "Data card loding\n";

restBuffer();

STARTEXE();

}

else if (s[0] == '$' && s[1] == 'E' && s[2] == 'N' && s[3] == 'D')

{

cout << "END of Job\n";

for(int i=0;i<300;i++){

if(i==PTR){

cout<<"---Page Table---\n";

}

if(i==PTR+10){

cout<<"--Page Table End--\n";

}

cout<<"[ "<<i<<" ] : ";

for(int j=0;j<4;j++){

cout<<Memory[i][j]<<" ";

}

cout<<'\n';

}

}

else

{

restBuffer();

// Get Frame for Program Page

pageNo = Allocate();

// Memory[PTR + pageTablePTR][0] = '0';

Memory[PTR + pageTablePTR][0] = (pageNo / 10) + '0';

Memory[PTR + pageTablePTR][1] = (pageNo % 10) + '0';

pageTablePTR++;

cout << "Program Card loding\n";

int length = s.size();

// Buffer <-- Program Card

for (int i = 0; i < length; i++)

{

buffer[i] = s[i];

}

int buff = 0;

IC = pageNo \* 10;

int end = IC + 10;

// Memory <-- Buffer

while (buff < 40 && buffer[buff] != '\0' && IC < end)

{

for (int j = 0; j < 4; j++)

{

if (buffer[buff] == 'H')

{

Memory[IC][j] = 'H';

buff++;

break;

}

Memory[IC][j] = buffer[buff];

buff++;

}

IC++;

}

}

}

infile.close(); // card File closed

}

}

void READ()

{

cout<<"Raed Function called\n";

string data;

getline(infile,data);

if(data[0]=='$' && data[1]=='E' && data[2]=='N' && data[3]=='D'){

TERMINATE(1);

return;

}

int n=data.size();

for(int i=0; i<n; i++){

buffer[i]=data[i];

}

int buff=0, memorty\_ptr=real\_address, end=real\_address+10;

while(buff<40 && buffer[buff]!='\0' && memorty\_ptr<end){

for(int i=0; i<4; i++){

Memory[memorty\_ptr][i]=buffer[buff];

buff++;

}

memorty\_ptr++;

}

restBuffer();

SI=0;

}

void WRITE(){

cout<<"Write Function called\n";

pcb.LLC++;

if(pcb.LLC>pcb.TLL){

TERMINATE(2);

return;

}

outfile.open("./output.txt",ios::app);

bool flag=true;

for(int i=real\_address; i<real\_address+9; i++){

for(int j=0; j<4; j++){

if(Memory[i][j]!='\0'){

outfile<<Memory[i][j];

}

else{

flag=false;

break;

}

}

if(!flag){

break;

}

}

SI=0;

outfile<<"\n";

outfile.close();

}

void TERMINATE(int EM){

Terminate=true;

outfile.open("./output.txt",ios::app);

switch(EM){

case 0:

outfile<<"NO ERROR : resuot stored successfully\n";

break;

case 1:

outfile<<"ERROR : Out of Data error\n";

break;

case 2:

outfile<<"ERROR : Line limit Exceeded\n";

break;

case 3:

outfile<<"ERROR : Time limit exceeded\n";

break;

case 4:

outfile<<"ERROR : Operation code error\n";

break;

case 5:

outfile<<"ERROR : Operand error\n";

break;

case 6:

outfile<<"ERROR : Invalid page fault\n";

break;

case 7:

outfile<<"ERROR : Time Limit exceeded and Operation code error\n";

break;

case 8:

outfile<<"ERROR : Time Limit exceeded and Operand error\n";

break;

}

outfile<<"JobID : "<<pcb.jobID<<" ";

outfile<<"IC : "<<IC<<" ";

outfile<<"IR : ";

for(int i=0; i<4; i++){

outfile<<IR[i];

}

outfile<<" ";

outfile<<"SI : "<<SI<<" ";

outfile<<"PI : "<<PI<<" ";

outfile<<"TI : "<<TI<<" ";

outfile<<"TTC : "<<pcb.TTC<<" ";

outfile<<"LLC : "<<pcb.LLC<<" ";

outfile<<"TTL : "<<pcb.TTL<<" ";

outfile<<"TLL : "<<pcb.TLL<<" ";

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

SI=PI=TI=0;

outfile.close();

}

int ADDRESSMAP(int virtual\_address)

{

if (0 <= virtual\_address && virtual\_address < 100)

{

PTE = PTR + (virtual\_address / 10);

if (Memory[PTE][0] == '\*')

{

PI = 3; // Page fault no such page exist

MOS();

}

else

{

string p;

p = Memory[PTE][0];

p += Memory[PTE][1];

int pageNo = stoi(p);

real\_address = pageNo \* 10 + (virtual\_address % 10);

return real\_address;

}

}

else

{

PI = 2; // Operand Error;

MOS();

}

return pageNo \* 10;

}

void STARTEXE(){

IC = 0;

EXECUTEUSERPROGRAM();

}

void EXECUTEUSERPROGRAM()

{ // Slave Mode

while (!Terminate)

{

real\_address = ADDRESSMAP(IC);

if(PI != 0){

return;

}

for (int i = 0; i < 4; i++){

IR[i] = Memory[real\_address][i];

}

IC++;

string op;

op += IR[2];

op += IR[3];

//GD - GET DATA

if (IR[0] == 'G' && IR[1] == 'D')

{

SIMULATION();

page\_fault\_valid = 1;

if (!isdigit(IR[2]) || !isdigit(IR[3]))

{

PI = 2;

MOS();

}

else{

virtual\_address = stoi(op);

real\_address = ADDRESSMAP(virtual\_address);

SI = 1;

MOS();

}

}

// PD - PRINT DATA

else if (IR[0] == 'P' && IR[1] == 'D')

{

SIMULATION();

page\_fault\_valid=0;

if (!isdigit(IR[2]) || !isdigit(IR[3]))

{

PI = 2;

MOS();

}

else{

virtual\_address = stoi(op);

real\_address = ADDRESSMAP(virtual\_address);

SI = 2;

MOS();

}

}

// H - HALT

else if (IR[0] == 'H' && IR[1] == '\0')

{

SIMULATION();

SI = 3;

MOS();

return;

}

// LR - LOAD DATA (Register <- Memory)

else if (IR[0] == 'L' && IR[1] == 'R')

{

SIMULATION();

page\_fault\_valid = 0;

if (!isdigit(IR[2]) || !isdigit(IR[3]))

{

PI = 2;

MOS();

}

else{

virtual\_address = stoi(op);

real\_address = ADDRESSMAP(virtual\_address);

for (int i = 0; i < 4; i++)

{

R[i] = Memory[real\_address][i];

}

}

}

// SR - STORE (Memory <- Register)

else if (IR[0] == 'S' && IR[1] == 'R')

{

SIMULATION();

page\_fault\_valid = 1;

if (!isdigit(IR[2]) || !isdigit(IR[3]))

{

PI = 2;

MOS();

}

else{

virtual\_address = stoi(op);

real\_address = ADDRESSMAP(virtual\_address);

for (int i = 0; i < 4; i++)

{

Memory[real\_address][i] = R[i];

}

}

}

// CR - COMPARE

else if (IR[0] == 'C' && IR[1] == 'R')

{

SIMULATION();

page\_fault\_valid=0;

if (!isdigit(IR[2]) || !isdigit(IR[3]))

{

PI = 2;

MOS();

}

else{

virtual\_address = stoi(op);

real\_address = ADDRESSMAP(virtual\_address);

string s1,s2;

for (int i = 0; i < 4; i++)

{

s1+=Memory[real\_address][i];

s2+=R[i];

}

if (s1 == s2)

{

C = true;

}

else

{

C = false;

}

}

}

// BT (JUMP if toogle is T)

else if (IR[0] == 'B' && IR[1] == 'T')

{

SIMULATION();

page\_fault\_valid=0;

if (!isdigit(IR[2]) || !isdigit(IR[3]))

{

PI = 2;

MOS();

}

else{

if (C)

{

string j;

j+=IR[2];

j+=IR[3];

IC = stoi(j);

}

}

}

else{

PI = 1;

SI = 0;

MOS();

}

}

}

void SIMULATION()

{

if (IR[0] == 'G' && IR[1] == 'D'){

pcb.TTC += 2;

}

else if (IR[0] == 'P' && IR[1] == 'D'){

pcb.TTC += 1;

}

else if (IR[0] == 'H'){

pcb.TTC += 1;

}

else if (IR[0] == 'L' && IR[1] == 'R'){

pcb.TTC += 1;

}

else if (IR[0] == 'S' && IR[1] == 'R'){

pcb.TTC += 2;

}

else if (IR[0] == 'C' && IR[1] == 'R'){

pcb.TTC += 1;

}

else if (IR[0] == 'B' && IR[1] == 'T'){

pcb.TTC += 1;

}

if(pcb.TTC >= pcb.TTL){

TI=2;

MOS();

}

}

public:

VM()

{

// infile.open("C:\\Users\\91766\\OneDrive\\Desktop\\TY\\os\\CP\\phase 2\\input\_phase2.txt", ios::in);

infile.open("./input\_phase2.txt", ios::in);

// infile.open("./input\_Phase1.txt", ios::in);

// infile.open("C:\\Users\\91766\\OneDrive\\Desktop\\TY\\os\\CP\\input\_custom.txt", ios::in);

init();

LOAD();

}

};

int main()

{

VM v;

return 0;

}