Operating System Lab-Programs

Q1)Write a Program to implement child process.

```
#include <windows.h>

#include <iostream>


int main() {

    STARTUPINFO si;

    PROCESS_INFORMATION pi;



    ZeroMemory(&si, sizeof(si));

    si.cb = sizeof(si);

    ZeroMemory(&pi, sizeof(pi));



    // Command to run (e.g., notepad)

    const char* command = "notepad.exe";



    if (!CreateProcess(

        NULL,           // No module name (use command line)

        (LPSTR)command,     // Command line

        NULL,           // Process handle not inheritable

        NULL,           // Thread handle not inheritable

        FALSE,          // Set handle inheritance to FALSE

        0,              // No creation flags

        NULL,           // Use parent's environment block

        NULL,           // Use parent's starting directory

        &si,            // Pointer to STARTUPINFO structure
```

```
    &pi)                // Pointer to PROCESS_INFORMATION structure
) {
    std::cerr << "CreateProcess failed (" << GetLastError() << ").\n";
    return 1;
}

std::cout << "Child process created. PID: " << pi.dwProcessId << std::endl;

// Wait until child process exits
WaitForSingleObject(pi.hProcess, INFINITE);

// Close process and thread handles
CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    std::cout << "Child process finished." << std::endl;

    return 0;
}
```

Q2) write a c++ program to implemented multithreding.

```
#include <iostream>
#include <thread>
#include <vector>
```

```
// Function to be executed by each thread
void printNumbers(int thread_id, int count) {
    for (int i = 1; i <= count; ++i) {
        std::cout << "Thread " << thread_id << " prints: " << i << std::endl;
    }
}

int main() {
    const int numThreads = 3;   // Number of threads
    const int count = 5;        // Numbers each thread prints

    std::vector<std::thread> threads;

    // Create threads
    for (int i = 1; i <= numThreads; ++i) {
        threads.push_back(std::thread(printNumbers, i, count));
    }

    // Wait for all threads to finish
    for (auto &t : threads) {
        t.join(); // Join each thread
    }

    std::cout << "All threads finished execution." << std::endl;
```

```
    return 0;
}
```

OUTPUT : -

```
Thread 1 prints: 1
Thread 2 prints: 1
Thread 3 prints: 1
Thread 1 prints: 2
Thread 2 prints: 2
Thread 3 prints: 2
Thread 1 prints: 3
Thread 2 prints: 3
Thread 3 prints: 3
...
All threads finished execution.
```

Q3) Simulating a virtual memory system with page replacement algorithms (Eg: FIFO or LRU).

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <list>
```

```
using namespace std;


// Function to simulate FIFO page replacement
int fifoPageReplacement(vector<int>& pages, int frames) {
    queue<int> q;
    unordered_map<int, bool> inMemory;
    int pageFaults = 0;

    for (int page : pages) {
        if (!inMemory[page]) {
            // Page fault occurs
            pageFaults++;
            if (q.size() == frames) {
                int old = q.front();
                q.pop();
                inMemory[old] = false;
            }
            q.push(page);
            inMemory[page] = true;
        }
    }

    return pageFaults;
}


// Function to simulate LRU page replacement
```

```
int lruPageReplacement(vector<int>& pages, int frames) {
    list<int> lruList;
    unordered_map<int, list<int>::iterator> pageMap;
    int pageFaults = 0;

for (int page : pages) {
    // Page not in memory
    if (pageMap.find(page) == pageMap.end()) {
        pageFaults++;
        if (lruList.size() == frames) {
            int last = lruList.back();
            lruList.pop_back();
            pageMap.erase(last);
        }
}
else {
    // Page in memory, remove from current position
    lruList.erase(pageMap[page]);
}
// Insert page at front (most recently used)
lruList.push_front(page);
        pageMap[page] = lruList.begin();
    }

    return pageFaults;
}
```

```
int main() {
    int numFrames;
    int numPages;

    cout << "Enter number of frames: ";
    cin >> numFrames;

    cout << "Enter number of pages in reference sequence: ";
    cin >> numPages;

    vector<int> pages(numPages);
    cout << "Enter page reference sequence: ";
    for (int i = 0; i < numPages; i++) {
        cin >> pages[i];
    }

    int fifoFaults = fifoPageReplacement(pages, numFrames);
    int lruFaults = lruPageReplacement(pages, numFrames);

    cout << "\nFIFO Page Faults: " << fifoFaults << endl;
    cout << "LRU Page Faults: " << lruFaults << endl;

    return 0;
}
```

OUTPUT :

Enter number of frames: 3

Enter number of pages in reference sequence: 12

Enter page reference sequence: 1 2 3 4 1 2 5 1 2 3 4 5

FIFO Page Faults: 9

LRU Page Faults: 8

Q4) Write a Program to implement Inter process Communication(IPC).

Parent process (Server) writes to the pipe, child process (Client) reads):

```
#include <windows.h>

#include <iostream>


using namespace std;


int main() {

    const char* pipeName = "\\\\.\\pipe\\MyPipe";


// Create a named pipe

HANDLE hPipe = CreateNamedPipe(

    pipeName,

    PIPE_ACCESS_OUTBOUND,   // write access
```

```
    PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
    1,
    1024,
    1024,
    0,
    NULL
);

if (hPipe == INVALID_HANDLE_VALUE) {
    cerr << "Failed to create pipe. Error: " << GetLastError() << endl;
    return 1;
}

    cout << "Waiting for client to connect..." << endl;
    BOOL connected = ConnectNamedPipe(hPipe, NULL) ? TRUE : (GetLastError() ==
ERROR_PIPE_CONNECTED);

    if (connected) {
        cout << "Client connected. Sending message..." << endl;
        const char* message = "Hello from server!";
        DWORD bytesWritten;
        WriteFile(hPipe, message, (DWORD)strlen(message) + 1, &bytesWritten, NULL);
        cout << "Message sent to client." << endl;
    }

CloseHandle(hPipe);
```

```
    return 0;
}
```

Client Process (reads from the pipe)

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

using namespace std;


int main() {
    const char* pipeName = "\\\\.\\pipe\\MyPipe";


    HANDLE hPipe;
    while (true) {
hPipe = CreateFile(
    pipeName,
    GENERIC_READ,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL
);


if (hPipe != INVALID_HANDLE_VALUE)
    break;
```

```
if (GetLastError() != ERROR_PIPE_BUSY) {
    cerr << "Could not open pipe. Error: " << GetLastError() << endl;
    return 1;
}
    // Wait for pipe to be available
    if (!WaitNamedPipe(pipeName, 20000)) {
        cerr << "Could not open pipe: 20 sec timeout" << endl;
        return 1;
    }
}

char buffer[1024];
DWORD bytesRead;
if (ReadFile(hPipe, buffer, sizeof(buffer), &bytesRead, NULL)) {
    cout << "Received from server: " << buffer << endl;
} else {
    cerr << "ReadFile failed. Error: " << GetLastError() << endl;
}

CloseHandle(hPipe);
    return 0;
}
```

OUTPUT:

Server console:

Waiting for client to connect...

Client connected. Sending message...

Message sent to client.

Client console:

Received from server: Hello from server!

Q5) Producer consumer problem.(Synchronization using semaphores)

```
#include <windows.h>

#include <iostream>

#include <queue>

#include <thread>


using namespace std;


// Buffer

queue<int> buffer;

const int BUFFER_SIZE = 5;


// Semaphores

HANDLE empty; // Counts empty slots

HANDLE full;  // Counts filled slots

HANDLE mutex; // Binary semaphore for mutual exclusion


void producer(int items) {

    for (int i = 1; i <= items; i++) {

        WaitForSingleObject(empty, INFINITE); // Wait if no empty slots

        WaitForSingleObject(mutex, INFINITE); // Enter critical section
```

```
        buffer.push(i);
        cout << "Produced: " << i << endl;

        ReleaseSemaphore(mutex, 1, NULL); // Leave critical section
        ReleaseSemaphore(full, 1, NULL);  // Increment filled slots

        this_thread::sleep_for(chrono::milliseconds(100));
    }
}

void consumer(int items) {
    for (int i = 1; i <= items; i++) {
        WaitForSingleObject(full, INFINITE);  // Wait if buffer empty
        WaitForSingleObject(mutex, INFINITE); // Enter critical section

        int item = buffer.front();
        buffer.pop();
        cout << "Consumed: " << item << endl;

        ReleaseSemaphore(mutex, 1, NULL); // Leave critical section
        ReleaseSemaphore(empty, 1, NULL); // Increment empty slots

        this_thread::sleep_for(chrono::milliseconds(150));
    }
}
```

```
int main() {
    int items = 10;

    // Create semaphores
    empty = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
    full = CreateSemaphore(NULL, 0, BUFFER_SIZE, NULL);
    mutex = CreateSemaphore(NULL, 1, 1, NULL);

    // Create threads
    thread prod(producer, items);
    thread cons(consumer, items);

    prod.join();
    cons.join();

    // Close handles
    CloseHandle(empty);
    CloseHandle(full);
    CloseHandle(mutex);

    return 0;
}
```

OUTPUT :

```
Produced: 1
Consumed: 1
Produced: 2
```

Produced: 3

Consumed: 2

Produced: 4

Consumed: 3

...

Q6) Simple Memory Allocation and Deallocation.

1. Stack Memory (Automatic Allocation)

Variables declared normally are allocated on the stack and **automatically deallocated** when they go out of scope

```
#include <iostream>

using namespace std;


int main() {
    int a = 10;          // Stack allocation
    int arr[5] = {1, 2, 3, 4, 5}; // Stack array


    cout << "Stack variable a: " << a << endl;
    cout << "Stack array arr[2]: " << arr[2] << endl;


    return 0;
}
```

2. Heap Memory (Dynamic Allocation)

Heap memory must be **manually allocated and deallocated** using new and delete.

Single Variable

```
#include <iostream>

using namespace std;


int main() {

    // Allocate memory dynamically

    int* ptr = new int;

    *ptr = 25;


    cout << "Dynamically allocated int: " << *ptr << endl;


    // Deallocate memory

    delete ptr;

    ptr = nullptr; // Avoid dangling pointer


    return 0;

}
```

Array Allocation

```
#include <iostream>

using namespace std;


int main() {

    int n = 5;


    // Allocate array dynamically

    int* arr = new int[n];
```

```
for (int i = 0; i < n; i++) {
    arr[i] = i * 10;
    cout << "arr[" << i << "] = " << arr[i] << endl;
}
    // Deallocate array memory
    delete[] arr;
    arr = nullptr;

    return 0;
}
```

Output :

1. Stack Memory Example

Code snippet:

```
int a = 10;
int arr[5] = {1, 2, 3, 4, 5};

cout << "Stack variable a: " << a << endl;
cout << "Stack array arr[2]: " << arr[2] << endl;
```

Stack variable a: 10

Stack array arr[2]: 3

2.Heap Memory Example (Single Variable)

Code snippet:

```
int* ptr = new int;

*ptr = 25;

cout << "Dynamically allocated int: " << *ptr << endl;

delete ptr;
```

Dynamically allocated int: 25

3. Heap Memory Example (Array)

Code snippet:

```
int n = 5;

int* arr = new int[n];

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

    arr[i] = i * 10;

    cout << "arr[" << i << "] = " << arr[i] << endl;

}

delete[] arr;
```

arr[0] = 0

arr[1] = 10

arr[2] = 20

arr[3] = 30

arr[4] = 40

Summary of Output ;

Stack variable a: 10

Stack array arr[2]: 3

Dynamically allocated int: 25

arr[0] = 0

arr[1] = 10

arr[2] = 20

arr[3] = 30

arr[4] = 40

Part-B

Q1)

```
#include <iostream>

#include <fstream>

using namespace std;


int main() {

fstream file("example.txt", ios::in | ios::out | ios::app); // Open for read/write, append mode


if (!file) {

    cerr << "Error opening file!" << endl;

    return 1;

}


// Write new data

file << "Appending a new line to the file." << endl;


// Move to beginning for reading

file.seekg(0);
```

```
string line;
cout << "Updated file contents:" << endl;
while (getline(file, line)) {
    cout << line << endl;
}
    file.close();
    return 0;
}
```

OUTPUT :

```
Assume example.txt initially contains:

Hello, this is a line in the file.
File I/O in C++ is simple!

After running the fstream program:

Updated file contents:
Hello, this is a line in the file.
File I/O in C++ is simple!
Appending a new line to the file.
```

Q2) Handling signals

```
#include <iostream>
#include <csignal>
#include <unistd.h>
using namespace std;

void handleSignal(int signum) {
    switch (signum) {
        case SIGINT:
            cout << "\nSIGINT received (Ctrl+C). Exiting..." << endl;
            break;
        case SIGTERM:
            cout << "\nSIGTERM received. Terminating..." << endl;
            break;
        case SIGABRT:
            cout << "\nSIGABRT received. Aborting..." << endl;
            break;
        default:
            cout << "\nSignal " << signum << " received." << endl;
    }
    exit(signum);
}
```

```
int main() {
    signal(SIGINT, handleSignal);
    signal(SIGTERM, handleSignal);
    signal(SIGABRT, handleSignal);

    cout << "Try pressing Ctrl+C or sending SIGTERM/SIGABRT.\n";

    while (true) {
        cout << "Program running..." << endl;
        sleep(1);
    }

    return 0;
}
```

OUTPUT:

Running...

Running...

Running...

^C

Interrupt signal (2) received.

Performing cleanup before exiting...

Q3) Scheduling Algorithms.

a) scheduling algorithms (FCFS):Processes are scheduled in the order they arrive.

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

struct Process {
    int pid;      // Process ID
    int arrival;  // Arrival Time
    int burst;    // Burst Time
    int completion;
    int turnaround;
    int waiting;
};

int main() {
```

```
int n;
cout << "Enter number of processes: ";
cin >> n;

vector<Process> processes(n);
for (int i = 0; i < n; i++) {
    processes[i].pid = i + 1;
    cout << "Enter arrival time and burst time for process " << i + 1 << ": ";
    cin >> processes[i].arrival >> processes[i].burst;
}

// Sort by arrival time (optional if input is not sorted)
for (int i = 0; i < n-1; i++) {
    for (int j = i+1; j < n; j++) {
        if (processes[i].arrival > processes[j].arrival) {
            swap(processes[i], processes[j]);
        }
    }
}

int currentTime = 0;
double totalTAT = 0, totalWT = 0;

for (int i = 0; i < n; i++) {
    if (currentTime < processes[i].arrival)
        currentTime = processes[i].arrival;

    processes[i].completion = currentTime + processes[i].burst;
    processes[i].turnaround = processes[i].completion - processes[i].arrival;
    processes[i].waiting = processes[i].turnaround - processes[i].burst;

    currentTime = processes[i].completion;

    totalTAT += processes[i].turnaround;
    totalWT += processes[i].waiting;
}

cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
for (auto &p : processes) {
    cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
        << p.completion << "\t" << p.turnaround << "\t" << p.waiting << endl;
```

```
    }

    cout << "\nAverage Turnaround Time: " << totalTAT/n << endl;
    cout << "Average Waiting Time: " << totalWT/n << endl;

    return 0;
}
```

OUTPUT :

Enter number of processes: 3

Enter arrival time and burst time for process 1: 0 5

Enter arrival time and burst time for process 2: 1 3

Enter arrival time and burst time for process 3: 2 8

PID	AT	BT	CT	TAT	WT
1	0	5	5	5	0
2	1	3	8	7	4
3	2	8	16	14	6

Average Turnaround Time: 8.66667

Average Waiting Time: 3.33333

b) Shortest Job First (SJF) : The process with the smallest execution time is scheduled next.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Process {
    int pid;
    int arrival;
```

```
    int burst;
    int completion;
    int turnaround;
    int waiting;
    bool completed = false;
};

int main() {
int n;
cout << "Enter number of processes: ";
cin >> n;

vector<Process> processes(n);
for (int i = 0; i < n; i++) {
    processes[i].pid = i + 1;
    cout << "Enter arrival time and burst time for process " << i + 1 << ": ";
    cin >> processes[i].arrival >> processes[i].burst;
}

int completed = 0, currentTime = 0;
double totalTAT = 0, totalWT = 0;

while (completed < n) {
    int idx = -1;
    int minBurst = 1e9;

    for (int i = 0; i < n; i++) {
        if (!processes[i].completed && processes[i].arrival <= currentTime) {
            if (processes[i].burst < minBurst) {
                minBurst = processes[i].burst;
                idx = i;
            }
        }
    }

    if (idx != -1) {
        processes[idx].completion = currentTime + processes[idx].burst;
        processes[idx].turnaround = processes[idx].completion -
processes[idx].arrival;
        processes[idx].waiting = processes[idx].turnaround - processes[idx].burst;
        processes[idx].completed = true;
```

```
        currentTime = processes[idx].completion;
        totalTAT += processes[idx].turnaround;
        totalWT += processes[idx].waiting;
        completed++;
    } else {
        currentTime++; // Idle CPU
    }
}

cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
for (auto &p : processes) {
    cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
         << p.completion << "\t" << p.turnaround << "\t" << p.waiting << endl;
}

cout << "\nAverage Turnaround Time: " << totalTAT / n << endl;
cout << "Average Waiting Time: " << totalWT / n << endl;

return 0;
}
```

OUTPUT :

```
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0 6
Enter arrival time and burst time for process 2: 1 8
Enter arrival time and burst time for process 3: 2 7
Enter arrival time and burst time for process 4: 3 3


PID    AT    BT    CT    TAT    WT
1      0     6     6     6      0
4      3     3     9     6      3
3      2     7     16    14     7
2      1     8     24    23     15

Average Turnaround Time: 12.25
Average Waiting Time: 6.25
```

C) Round Robin(RR): Each process is assigned a fix time slice(quantum)and is cycled through in a circular fashion.

```
#include <iostream>
```

```
#include <vector>
#include <queue>
using namespace std;

struct Process {
    int pid;
    int arrival;
    int burst;
    int remaining;
    int completion;
    int turnaround;
    int waiting;
};

int main() {
    int n, quantum;
    cout << "Enter number of processes: ";
    cin >> n;
    vector<Process> processes(n);

    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter arrival time and burst time for process " << i + 1 << ": ";
        cin >> processes[i].arrival >> processes[i].burst;
        processes[i].remaining = processes[i].burst;
    }

    cout << "Enter time quantum: ";
    cin >> quantum;

    queue<int> q;
    int currentTime = 0;
    vector<bool> inQueue(n, false);
    int completed = 0;
    double totalTAT = 0, totalWT = 0;

    // Add processes that arrive at time 0
    for (int i = 0; i < n; i++) {
        if (processes[i].arrival <= currentTime) {
            q.push(i);
            inQueue[i] = true;
```

```
    }
}

while (completed < n) {
    if (q.empty()) {
        currentTime++;
        for (int i = 0; i < n; i++) {
            if (!inQueue[i] && processes[i].arrival <= currentTime) {
                q.push(i);
                inQueue[i] = true;
            }
        }
        continue;
    }

    int idx = q.front();
    q.pop();

    if (processes[idx].remaining <= quantum) {
        currentTime += processes[idx].remaining;
        processes[idx].remaining = 0;
        processes[idx].completion = currentTime;
        processes[idx].turnaround = processes[idx].completion -
processes[idx].arrival;
        processes[idx].waiting = processes[idx].turnaround - processes[idx].burst;

        totalTAT += processes[idx].turnaround;
        totalWT += processes[idx].waiting;
        completed++;
    } else {
        currentTime += quantum;
        processes[idx].remaining -= quantum;
    }

    // Add processes that have arrived during execution
    for (int i = 0; i < n; i++) {
        if (!inQueue[i] && processes[i].arrival <= currentTime) {
            q.push(i);
            inQueue[i] = true;
        }
    }
}
```

```
    // If process is not finished, put it back in the queue
    if (processes[idx].remaining > 0) {
        q.push(idx);
    }
}

cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
    for (auto &p : processes) {
        cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
             << p.completion << "\t" << p.turnaround << "\t" << p.waiting << endl;
    }

    cout << "\nAverage Turnaround Time: " << totalTAT / n << endl;
    cout << "Average Waiting Time: " << totalWT / n << endl;

    return 0;
}
```

Sample OUTPUT:

```
Enter number of processes: 3
Enter arrival time and burst time for process 1: 0 5
Enter arrival time and burst time for process 2: 1 3
Enter arrival time and burst time for process 3: 2 8

Enter time quantum: 2
PID    AT    BT    CT    TAT    WT
1      0     5     9     9      4
2      1     3     7     6      3
3      2     8     17    15     7

Average Turnaround Time: 10
Average Waiting Time: 4.66667
```

D) Priority Scheduling : Process are scheduled based on priority levels ,where higher priority processes are executed first.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
struct Process {
    int pid;
    int arrival;
    int burst;
    int priority;
    int completion;
    int turnaround;
    int waiting;
    bool completed = false;
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> processes(n);
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        cout << "Enter arrival time, burst time, and priority for process " << i + 1 << ":
";
        cin >> processes[i].arrival >> processes[i].burst >> processes[i].priority;
    }

    int completed = 0, currentTime = 0;
    double totalTAT = 0, totalWT = 0;

    while (completed < n) {
      int idx = -1;
      int highestPriority = 1e9;

      for (int i = 0; i < n; i++) {
        if (!processes[i].completed && processes[i].arrival <= currentTime) {
          if (processes[i].priority < highestPriority) {
            highestPriority = processes[i].priority;
            idx = i;
          }
          // If priorities are same, use arrival time as tie-breaker
          else if (processes[i].priority == highestPriority) {
            if (processes[i].arrival < processes[idx].arrival)
              idx = i;
```

```
            }
        }
    }

    if (idx != -1) {
        processes[idx].completion = currentTime + processes[idx].burst;
        processes[idx].turnaround = processes[idx].completion -
processes[idx].arrival;
        processes[idx].waiting = processes[idx].turnaround - processes[idx].burst;
        processes[idx].completed = true;

        currentTime = processes[idx].completion;
        totalTAT += processes[idx].turnaround;
        totalWT += processes[idx].waiting;
        completed++;
    } else {
        currentTime++; // CPU idle
    }
}

cout << "\nPID\tAT\tBT\tPR\tCT\tTAT\tWT\n";
for (auto &p : processes) {
    cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
        << p.priority << "\t" << p.completion << "\t"
        << p.turnaround << "\t" << p.waiting << endl;
}

cout << "\nAverage Turnaround Time: " << totalTAT / n << endl;
cout << "Average Waiting Time: " << totalWT / n << endl;

return 0;
}
```

Sample OUTPUT :

```
Enter number of processes: 4
Enter arrival time, burst time, and priority for process 1: 0 5 2
Enter arrival time, burst time, and priority for process 2: 1 3 1
Enter arrival time, burst time, and priority for process 3: 2 8 4
Enter arrival time, burst time, and priority for process 4: 3 6 2
PID    AT    BT    PR    CT    TAT    WT
1      0     5     2     5     5      0
2      1     3     1     8     7      4
```

4 3 6 2 14 11 5
3 2 8 4 22 20 12

Average Turnaround Time: 10.75
Average Waiting Time: 5.25