1)

```
   #include <iostream>
   #include <unistd.h>
   #include <climits>

   int main() {
       // Check and print the number of clock ticks
       std::cout << "No. of clock ticks: " << sysconf(_SC_CLK_TCK) << std::endl;

       // Check and print the max number of child processes
       std::cout << "Max. no. of child processes: " << sysconf(_SC_CHILD_MAX) <<
std::endl;

       // Check and print the max path length
       std::cout << "Max. path length: " << pathconf("/", _PC_PATH_MAX) <<
std::endl;

       // Check and print the max number of characters in a file name
       std::cout << "Max. no. of characters in a file name: " << pathconf("/",
_PC_NAME_MAX) << std::endl;

       // Check and print the max number of open files per process
       std::cout << "Max. no. of open files/process: " << sysconf(_SC_OPEN_MAX) <<
std::endl;

       return 0;
   }
```

2)a) Copy a file using system calls in Unix:

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

int main() {
    const char* sourceFile = "source.txt";
    const char* destinationFile = "destination.txt";

    // Open the source file for reading
    int source_fd = open(sourceFile, O_RDONLY);
    if (source_fd == -1) {
        perror("Error opening source file");
        return 1;
    }

    // Create or open the destination file for writing
    int dest_fd = open(destinationFile, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
    if (dest_fd == -1) {
```

```
        perror("Error opening destination file");
        close(source_fd);
        return 1;
    }

    char buffer[4096];
    ssize_t bytes_read;

    // Copy data from source to destination
    while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
        ssize_t bytes_written = write(dest_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("Error writing to destination file");
            close(source_fd);
            close(dest_fd);
            return 1;
        }
    }

    // Close file descriptors
    close(source_fd);
    close(dest_fd);

    std::cout << "File copied successfully!" << std::endl;

    return 0;
}
```

2b). Output the contents of the Environment list:

```
cpp
#include <iostream>

extern char** environ;

int main() {
    char** env = environ;

    while (*env != nullptr) {
        std::cout << *env << std::endl;
        env++;
    }

    return 0;
}
```

3. a. Emulate the UNIX ln command

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " source_file target_file" <<
std::endl;
        return 1;
    }

    const char *source_file = argv[1];
    const char *target_file = argv[2];

    if (link(source_file, target_file) == 0) {
        std::cout << "Hard link created: " << target_file << " -> " << source_file
<< std::endl;
        return 0;
    } else {
        perror("Error creating hard link");
        return 2;
    }
}
```

3b)C++ program that creates a child process from a parent process using `fork()` and both the parent and child processes count up to 5 and display their counts:

```
cpp
#include <iostream>
#include <unistd.h>

int main() {
    pid_t child_pid;

    // Fork a child process
    child_pid = fork();

    if (child_pid == -1) {
        std::cerr << "Fork failed." << std::endl;
        return 1;
    }

    // Both parent and child processes continue from here

    for (int i = 1; i <= 5; i++) {
        if (child_pid == 0) {
            // Child process
            std::cout << "Child Count: " << i << std::endl;
        } else {
```

```
            // Parent process
            std::cout << "Parent Count: " << i << std::endl;
        }
        sleep(1); // Sleep for 1 second
    }

    return 0;
}
```

4)C++ program where two processes communicate using shared memory

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

// Define the shared memory key
#define SHM_KEY 1234
// Define the size of the shared memory segment
#define SHM_SIZE 1024

int main() {
    // Create a key for the shared memory segment
    key_t key = ftok(".", SHM_KEY);
    if (key == -1) {
        perror("ftok"); // Print an error message if ftok fails
        exit(1);
    }

    // Create (or get) a shared memory segment
    int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget"); // Print an error message if shmget fails
        exit(1);
    }

    // Attach the shared memory segment to the process's address space
    char *shm_ptr = (char *)shmat(shmid, NULL, 0);
    if (shm_ptr == (char *)(-1)) {
        perror("shmat"); // Print an error message if shmat fails
        exit(1);
    }

    // Parent process writes a message to shared memory
    std::string message = "Hello, shared memory!";
    std::strcpy(shm_ptr, message.c_str());
```

```
    // Fork a child process
    pid_t child_pid = fork();

    if (child_pid == -1) {
        perror("fork"); // Print an error message if fork fails
        exit(1);
    }

    if (child_pid == 0) {
        // Child process reads from shared memory and prints
        std::cout << "Child process reads: " << shm_ptr << std::endl;

        // Detach the shared memory segment from the child process
        if (shmdt(shm_ptr) == -1) {
            perror("shmdt"); // Print an error message if shmdt fails
            exit(1);
        }
    } else {
        // Parent process waits for the child to finish
        wait(NULL);

        // Detach the shared memory segment from the parent process
        if (shmdt(shm_ptr) == -1) {
            perror("shmdt"); // Print an error message if shmdt fails
            exit(1);
        }

        // Remove the shared memory segment
        if (shmctl(shmid, IPC_RMID, NULL) == -1) {
            perror("shmctl"); // Print an error message if shmctl fails
            exit(1);
        }
    }

    return 0;
}
```

5)C++ prog the producer-consumer problem using semaphores in UNIX

```
#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <vector>

// Define the maximum number of items in the buffer
#define MAX_BUFFER_SIZE 5
```

```
// Define the number of producer and consumer threads
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

// Shared variables
std::vector<int> buffer; // Shared buffer
sem_t mutex;             // Semaphore for mutual exclusion
sem_t empty;             // Semaphore for tracking empty slots in the buffer
sem_t full;              // Semaphore for tracking filled slots in the buffer

// Producer function
void* producer(void* arg) {
    int item = *((int*)arg);
    while (true) {
        sleep(1); // Simulate time to produce an item

        sem_wait(&empty); // Wait for an empty slot in the buffer
        sem_wait(&mutex); // Enter critical section

        buffer.push_back(item); // Produce an item and add it to the buffer
        std::cout << "Produced: " << item << ", Buffer size: " << buffer.size() <<
std::endl;

        sem_post(&mutex); // Exit critical section
        sem_post(&full);  // Signal that a slot in the buffer is filled
    }
    return NULL;
}

// Consumer function
void* consumer(void* arg) {
    while (true) {
        sleep(1); // Simulate time to consume an item

        sem_wait(&full);  // Wait for a filled slot in the buffer
        sem_wait(&mutex); // Enter critical section

        int item = buffer.back(); // Consume an item from the buffer
        buffer.pop_back();
        std::cout << "Consumed: " << item << ", Buffer size: " << buffer.size() <<
std::endl;

        sem_post(&mutex); // Exit critical section
        sem_post(&empty); // Signal that a slot in the buffer is empty
    }
    return NULL;
}

int main() {
    // Initialize semaphores
```

```
    sem_init(&mutex, 0, 1);        // Mutex semaphore
    sem_init(&empty, 0, MAX_BUFFER_SIZE); // Empty semaphore (buffer slots
available)
    sem_init(&full, 0, 0);         // Full semaphore (buffer slots filled)

    // Create producer and consumer threads
    pthread_t producer_threads[NUM_PRODUCERS];
    pthread_t consumer_threads[NUM_CONSUMERS];

    for (int i = 0; i < NUM_PRODUCERS; ++i) {
        int* item = new int(i);
        pthread_create(&producer_threads[i], NULL, producer, (void*)item);
    }

    for (int i = 0; i < NUM_CONSUMERS; ++i) {
        pthread_create(&consumer_threads[i], NULL, consumer, NULL);
    }

    // Join threads
    for (int i = 0; i < NUM_PRODUCERS; ++i) {
        pthread_join(producer_threads[i], NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; ++i) {
        pthread_join(consumer_threads[i], NULL);
    }

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

6.Round Robin scheduling algorithm

```
cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

struct Process {
    char name;
    int arrivalTime;
    int burstTime;
    int remainingTime;
};
```

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

    vector<Process> processes(n);

    // Input process information: name, arrival time, and burst time
    for (int i = 0; i < n; i++) {
        processes[i].name = 'A' + i;
        cout << "Enter arrival time for process " << processes[i].name << ": ";
        cin >> processes[i].arrivalTime;
        cout << "Enter burst time for process " << processes[i].name << ": ";
        cin >> processes[i].burstTime;
        processes[i].remainingTime = processes[i].burstTime;
    }

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

    // Initialize variables for tracking time and waiting times
    int currentTime = 0;
    queue<Process> readyQueue;
    vector<int> waitingTimes(n, 0);

    while (true) {
        // Find processes that have arrived but not completed
        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0) {
                readyQueue.push(processes[i]);
            }
        }

        if (readyQueue.empty()) {
            // No processes in the queue, check if all processes are completed
            bool allProcessesCompleted = true;
            for (int i = 0; i < n; i++) {
                if (processes[i].remainingTime > 0) {
                    allProcessesCompleted = false;
                    break;
                }
            }
            if (allProcessesCompleted) {
                break;  // All processes completed, exit the loop
            } else {
                currentTime++;  // No processes to execute, time passes
            }
        } else {
```

```
            // Execute a process with the given time quantum
            Process currentProcess = readyQueue.front();
            readyQueue.pop();
            int executeTime = min(currentProcess.remainingTime, quantum);
            currentProcess.remainingTime -= executeTime;
            currentTime += executeTime;

            // Update waiting times for other processes in the queue
            for (int i = 0; i < n; i++) {
                if (processes[i].arrivalTime <= currentTime &&
processes[i].remainingTime > 0 && processes[i].name != currentProcess.name) {
                    waitingTimes[i] += currentTime - processes[i].arrivalTime;
                }
            }

            if (currentProcess.remainingTime > 0) {
                readyQueue.push(currentProcess);
            }
        }
    }

    // Calculate turnaround times and print results
    vector<int> turnaroundTimes(n, 0);
    double totalWaitingTime = 0;
    double totalTurnaroundTime = 0;

    for (int i = 0; i < n; i++) {
        turnaroundTimes[i] = processes[i].burstTime + waitingTimes[i];
        totalWaitingTime += waitingTimes[i];
        totalTurnaroundTime += turnaroundTimes[i];
    }

    double averageWaitingTime = totalWaitingTime / n;
    double averageTurnaroundTime = totalTurnaroundTime / n;

    cout << "Process\tWaiting Time\tTurnaround Time" << endl;
    for (int i = 0; i < n; i++) {
        cout << processes[i].name << "\t" << waitingTimes[i] << "\t" <<
turnaroundTimes[i] << endl;
    }

    cout << "Average Waiting Time: " << averageWaitingTime << endl;
    cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;

    return 0;
}
```

7. C++ program for implementing a priority-based scheduling algorithm and

calculating average waiting time and average turnaround time in a Unix environment.

cpp

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct Process {
    int processID;
    int burstTime;
    int priority;
    int waitingTime;
    int turnaroundTime;
};

bool comparePriority(const Process &a, const Process &b) {
    return a.priority < b.priority;
}

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

    vector<Process> processes(numProcesses);

    for (int i = 0; i < numProcesses; i++) {
        processes[i].processID = i + 1;
        cout << "Enter burst time for process " << i + 1 << ": ";
        cin >> processes[i].burstTime;
        cout << "Enter priority for process " << i + 1 << ": ";
        cin >> processes[i].priority;
    }

    sort(processes.begin(), processes.end(), comparePriority);

    processes[0].waitingTime = 0;
    processes[0].turnaroundTime = processes[0].burstTime;

    for (int i = 1; i < numProcesses; i++) {
        processes[i].waitingTime = processes[i - 1].waitingTime + processes[i -
1].burstTime;
        processes[i].turnaroundTime = processes[i].waitingTime +
processes[i].burstTime;
    }

    double totalWaitingTime = 0;
    double totalTurnaroundTime = 0;
```

```
    for (const Process &p : processes) {
        totalWaitingTime += p.waitingTime;
        totalTurnaroundTime += p.turnaroundTime;
    }

    double averageWaitingTime = totalWaitingTime / numProcesses;
    double averageTurnaroundTime = totalTurnaroundTime / numProcesses;

    cout << "Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n";
    for (const Process &p : processes) {
        cout << p.processID << "\t\t" << p.burstTime << "\t\t" << p.priority <<
"\t\t" << p.waitingTime << "\t\t" << p.turnaroundTime << endl;
    }

    cout << "\nAverage Waiting Time: " << averageWaitingTime << endl;
    cout << "Average Turnaround Time: " << averageTurnaroundTime << endl;

    return 0;
}
```

8.C++ program that acts as a sender to send data to a message queue and a receiver to read data from the same message queue......run the sender and receiver in separate terminals.

cpp

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

using namespace std;

// Define a structure for the message data
struct Message {
    long mtype;
    char mtext[100];
};

int main() {
    key_t key;
    int msgid;
    Message message;

    // Step 1: Create a key for the message queue
    key = ftok("message_queue_key", 'A');
```

```
    if (key == -1) {
        perror("ftok");
        exit(1);
    }

    // Step 2: Create or open the message queue
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    // Sender: Send data to the message queue
    message.mtype = 1; // Message type (you can use different types for different
purposes)
    strcpy(message.mtext, "Hello, this is a message from the sender!");

    // Step 3: Send the message to the queue
    if (msgsnd(msgid, &message, sizeof(message.mtext), 0) == -1) {
        perror("msgsnd");
        exit(1);
    }

    cout << "Data sent to message queue." << endl;

    return 0;
}

//sender

#include <iostream>
#include <cstring>
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

using namespace std;

// Define a structure for the message data
struct Message {
    long mtype;
    char mtext[100];
};

int main() {
    key_t key;
    int msgid;
    Message message;
```

```
    // Step 1: Create a key for the message queue (use the same key as in the
sender)
    key = ftok("message_queue_key", 'A');
    if (key == -1) {
        perror("ftok");
        exit(1);
    }

    // Step 2: Create or open the message queue
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    // Receiver: Read data from the message queue
    // Step 3: Receive a message from the queue with message type 1
    if (msgrcv(msgid, &message, sizeof(message.mtext), 1, 0) == -1) {
        perror("msgrcv");
        exit(1);
    }

    cout << "Data received from message queue: " << message.mtext << endl;

    return 0;
}
```