Process create and terminate

1. Process Creation and Termination

🔹 `fork()`

Purpose: Creates a new process by duplicating the current process.

Prototype:

cpp
CopyEdit
```
pid_t fork();
```

Returns:

- `0` to child
- Child's PID to parent
- `-1` on error

Example:

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        std::cout << "Child process\n";
    } else if (pid > 0) {
        std::cout << "Parent process, child PID: " << pid << "\n";
    } else {
        std::cerr << "Fork failed\n";
    }
    return 0;
}
```

- ### 🔚 `exit()` and `_exit()`

Purpose: Terminates a process.

Header: `<cstdlib>` for `exit()`, `<unistd.h>` for `_exit()`

Prototype:

cpp
CopyEdit
```
void exit(int status);
void _exit(int status);
```

Example:

cpp
CopyEdit
```
exit(0);
```

`exec()`

```
execlp("ls", "ls", "-l", NULL);
```

`wait()`

wait(&status);

CPU Scheduling

C++ Implementation of SRJF (Preemptive):

- It's the **preemptive version of Shortest Job First (SJF)**.
- The process with the **shortest remaining time** is executed first.
- If a new process arrives with a shorter remaining time than the current one, the CPU **preempts** and switches to the new process.

SJF (Shortest Job First) - Non-Preemptive

- **Selects the process with the shortest burst time from the list of arrived processes.**
- **Once a process starts executing, it runs to completion (non-preemptive).**
- **It minimizes average waiting time but may cause starvation for longer jobs.**

Round Robin Scheduling

- **Each process gets a fixed time quantum.**
- **Processes are executed in FIFO order.**
- **If a process doesn’t finish in its time quantum, it's moved to the back of the queue.**
- **It's preemptive, and fair for time-sharing systems.**

Priority Scheduling (Non-Preemptive)

- **Each process is assigned a priority.**
- **The CPU is assigned to the process with the highest priority (lowest number = higher priority).**
- **If two processes have the same priority, use arrival time as a tiebreaker.**

FCFS

file management

Summary of System Calls Used

System Call	Description
`open()`	Opens or creates a file
`write()`	Writes bytes to a file
`read()`	Reads bytes from a file
`close()`	Closes an opened file descriptor
`unlink()`	Deletes a file
`rename()`	Renames or moves a file

`void create_file(const char* filename, const char* content)`

Purpose: Creates a new file (or truncates if it exists) and writes the given content into it.

System Calls Used:

- `open()` with flags: `O_CREAT | O_WRONLY | O_TRUNC`
- `write()`
- `close()`

Permissions: `0644` → Owner read/write, others read.

`void delete_file(const char* filename)`

Purpose: Deletes a file from the filesystem.

System Call Used: `unlink()`

`void copy_file(const char* source, const char* destination)`

Purpose: Copies content from a source file to a new destination file.

System Calls Used:

- `open()` for source (read-only) and destination (write + create/truncate)
- `read()` and `write()` in a loop
- `close()`

Buffer: 1024 bytes used for chunked copying.

`void move_file(const char* old_name, const char* new_name)`

Purpose: Moves or renames a file.

System Call Used: `rename()`

```
int main() {
    const char* file1 = "test.txt";
    const char* file2 = "copy.txt";
    const char* file3 = "moved.txt";

    create_file(file1, "Hello, this is a test file.\n");
    copy_file(file1, file2);
    move_file(file2, file3);
    delete_file(file1);

    return 0;
}
```

shell scripting

1. Script Header

#!/bin/bash

This directive specifies the interpreter to be used, in this case, the GNU Bourne Again Shell (Bash).

2. Variable Declaration and Usage

name="Uthpol"
echo "Hello, $name"

- No whitespace around =.
- Variables are referenced using the $ symbol.

3. User Input

read -p "Enter your name: " username
echo "Welcome, $username"

- The read command captures input from the terminal.

4. Conditional Expressions

A. Basic If-Else

```
if [ "$age" -ge 18 ]; then
  echo "Adult"
else
  echo "Minor"
fi
```

B. If-Elif-Else

```
if [ "$marks" -ge 90 ]; then
  echo "Grade: A"
elif [ "$marks" -ge 75 ]; then
  echo "Grade: B"
else
  echo "Grade: C"
fi
```

C. Nested Conditions

```
if [ "$logged_in" = "yes" ]; then
  if [ "$role" = "admin" ]; then
    echo "Welcome, Admin"
  fi
fi
```

D. Logical Operators

```
if [ "$age" -ge 18 ] && [ "$citizen" = "yes" ]; then
  echo "Eligible to vote"
fi


if [ "$age" -ge 18 ] || [ "$has_permit" = "yes" ]; then
  echo "Access granted"
fi
```

E. File Conditions

```
if [ -f "file.txt" ]; then
  echo "Regular file exists"
fi

if [ -d "dir_name" ]; then
  echo "Directory exists"
fi
```

- Common file test flags:
    - -e: file exists
    - -f: regular file
    - -d: directory
    - -r: readable
    - -w: writable
    - -x: executable

5. Loop Constructs

- **For Loop (numeric range)**

```
for i in {1..5}
do
  echo "Iteration $i"
done
```

- **C style For Loop**

```
for ((i=1; i<=5; i++))
do
  echo "i = $i"
done
```

- **While Loop**

```
count=1
while [ $count -le 5 ]
do
  echo "Count: $count"
  ((count++))
done
```

6. Arithmetic Operations

```
a=10
b=5

sum=$((a + b))
diff=$((a - b))
prod=$((a * b))
quot=$((a / b))
mod=$((a % b))

echo "Sum: $sum"
```

- Enclose expressions in $(()) for integer arithmetic.

7. String Operations

```bash
str="Hello World"

echo "Length: ${#str}"
echo "Substring: ${str:6:5}"
echo "Replace: ${str/World/Bash}"
```

8. Functions

```bash
greet() {
  echo "Good day, $1!"
}

greet "Uthpol"
```

Functions encapsulate logic and may accept parameters for modular scripting.

Thread Code

Class: MyThread.java

```java
package lab2;

public class MyThread implements  Runnable{
   private  String name;
   private  int age;

   public MyThread(String name, int age) {
       this.name = name;
       this.age = age;
   }

   @Override
   public void run() {

       try {
           for(int i=0; i<=3; i++){
               System.out.println(i +" Helle "+name+ "! Age: "+age);
               Thread.sleep(1000);
           }
       }catch (Exception e){
           System.out.println("Exceptions");
       }
   }
}
```

Class: Main.java

```java
public class Main {
   public static void main(String[] args) {


           Thread t1 =new Thread(new ThreadCreation(23,"Anis"));
           Thread t2 = new Thread(new ThreadCreation(19,"Uthpol"));

           t1.start();
           t2.start();

           try{
               t1.join();
               t2.join();
           }catch(Exception e){
               System.out.println("Thread interrupted");
           }
           System.out.println("Main thread finished.");
   }
}
```

deadlock simulation

Class: ResourecsA.java

```java
package DeadlockSimulation;

public class ResourcesA implements Runnable{
   private final Object lockA;
   private  final Object lockB;

   public ResourcesA(Object lockA, Object lockB) {
       this.lockA = lockA;
       this.lockB = lockB;
   }

   @Override
   public void run() {
       synchronized (lockA){
           System.out.println("Thread A acquired LockA");
           synchronized (lockB){
               System.out.println("Thread B acquired LockB");
           }
       }
   }
}
```

Class: ResourcesB.java

```java
package DeadlockSimulation;

public class ResourcesB implements  Runnable{
   private final  Object lockA;
   private final  Object lockB;

   public ResourcesB(Object lockA, Object lockB) {
       this.lockA = lockA;
       this.lockB = lockB;
   }


   @Override
   public void run() {
       synchronized (lockB){
           System.out.println("Thread B acquired LockB");
           synchronized (lockA){
               System.out.println("Thread A acquired LockA");
           }
       }
   }
}
```

Class: Main.java

```
package DeadlockSimulation;

public class Main {
   public static void main(String[] args) {
      final Object lockA = new Object();
      final Object lockB = new Object();

      Thread th1 =new Thread(new ResourcesA(lockA,lockB));
      Thread th2 = new Thread(new ResourcesB(lockA,lockB));

      th1.start();
      th2.start();

       System.out.println("finshed threads");
   }
}
```

Output

```
Thread A acquired LockA
Thread B acquired LockB
…………………………………………………………………..
```

code

1. File Management

Summary of System Calls Used

System Call	Description
`open()`	Opens or creates a file
`write()`	Writes bytes to a file
`read()`	Reads bytes from a file
`close()`	Closes an opened file descriptor
`unlink()`	Deletes a file
`rename()`	Renames or moves a fil

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>     // open()
#include <unistd.h>    // read(), write(), close(), unlink(),
#include <string.h>
#include <errno.h>

#define BUF_SIZE 1024
```

```
// Function to create a file and write data
void create_file(const char* filename, const char* content) {
    int fd = open(filename, O_CREAT | O_WRONLY |
O_TRUNC, 0644);
    if (fd == -1) {
        perror("Create");
        return;
    }
    write(fd, content, strlen(content));
    close(fd);
    printf("File '%s' created successfully.\n", filename);
}


// Function to delete a file
void delete_file(const char* filename) {
    if (unlink(filename) == -1) {
        perror("Delete");
        return;
    }
    printf("File '%s' deleted successfully.\n", filename);
}


// Function to move (rename) a file
void move_file(const char* old_name, const char* new_name) {
    if (rename(old_name, new_name) == -1) {
        perror("Move");
        return;
    }
    printf("File moved from '%s' to '%s'.\n", old_name,
new_name);
}
```

```
// Function to copy a file
void copy_file(const char* source, const char* destination) {
    int src = open(source, O_RDONLY);
    if (src == -1) {
        perror("Open source");
        return;
    }

    int dest = open(destination, O_CREAT | O_WRONLY |
O_TRUNC, 0644);
    if (dest == -1) {
        perror("Create destination");
        close(src);
        return;
    }

    char buffer[BUF_SIZE];
    ssize_t bytes;
    while ((bytes = read(src, buffer, BUF_SIZE)) > 0) {
        write(dest, buffer, bytes);
    }

    close(src);
    close(dest);
    printf("File copied from '%s' to '%s'.\n", source, destination);
}
```

```
int main() {
    const char* file1 = "test.txt";
    const char* file2 = "copy.txt";
    const char* file3 = "moved.txt";

    create_file(file1, "Hello, this is a test file.\n");
    copy_file(file1, file2);
    move_file(file2, file3);
    delete_file(file1);

    return 0;
}
```

2. C++ Implementation of SRJF (Preemptive)

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

using namespace std;

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

```
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;
        processes[i].remaining = processes[i].burst;
    }

    int complete = 0, current_time = 0;
    int min_remain = 1e9;
    int shortest = -1;
    bool found = false;

    while (complete < n) {
        shortest = -1;
        min_remain = 1e9;
        found = false;

        // Find process with minimum remaining time at current_time
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival <= current_time &&
                processes[i].remaining > 0 &&
                processes[i].remaining < min_remain) {
                min_remain = processes[i].remaining;
                shortest = i;
                found = true;
            }
        }

        if (!found) {
            current_time++;
            continue;
        }

        // Execute process
        processes[shortest].remaining--;
        current_time++;
```

```
        // If process finished
        if (processes[shortest].remaining == 0) {
            complete++;
            processes[shortest].completion = current_time;
            processes[shortest].turnaround = processes[shortest].completion -
processes[shortest].arrival;
            processes[shortest].waiting = processes[shortest].turnaround -
processes[shortest].burst;
        }
    }

    // Output
    double total_wt = 0, total_tat = 0;

    cout << "\nPID\tArrival\tBurst\tWaiting\tTurnaround\tCompletion\n";
    for (const auto& p : processes) {
        cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
             << p.waiting << "\t" << p.turnaround << "\t\t" << p.completion << "\n";
        total_wt += p.waiting;
        total_tat += p.turnaround;
    }

    cout << "\nAverage Waiting Time: " << total_wt / n;
    cout << "\nAverage Turnaround Time: " << total_tat / n << endl;

    return 0;
}
```

3. SJF (Shortest Job First) - Non-Preemptive

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

using namespace std;

struct Process {
    int pid;
    int arrival;
    int burst;
    int completion;
    int turnaround;
    int waiting;
```

```
    bool done = 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 current_time = 0, completed = 0;
    double total_waiting = 0, total_turnaround = 0;

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

        // Find the process with the shortest burst time among those that have arrived
        for (int i = 0; i < n; ++i) {
            if (!processes[i].done && processes[i].arrival <= current_time) {
                if (processes[i].burst < min_burst) {
                    min_burst = processes[i].burst;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            Process &p = processes[idx];
            p.completion = current_time + p.burst;
            p.turnaround = p.completion - p.arrival;
            p.waiting = p.turnaround - p.burst;

            current_time = p.completion;
            p.done = true;
            completed++;

            total_waiting += p.waiting;
```

```
            total_turnaround += p.turnaround;
        } else {
            current_time++; // CPU is idle
        }
    }

    cout << "\nPID\tArrival\tBurst\tWaiting\tTurnaround\tCompletion\n";
    for (const auto &p : processes) {
        cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
             << p.waiting << "\t" << p.turnaround << "\t\t" << p.completion << "\n";
    }

    cout << "\nAverage Waiting Time: " << total_waiting / n;
    cout << "\nAverage Turnaround Time: " << total_turnaround / n << "\n";

    return 0;
}
```

4. Round Robin Scheduling

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

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;
vector<bool> in_queue(n, false);

int current_time = 0;
int completed = 0;

// Sort by arrival time initially
sort(processes.begin(), processes.end(), [](Process a, Process b) {
    return a.arrival < b.arrival;
});

q.push(0);
in_queue[0] = true;
current_time = processes[0].arrival;

while (!q.empty()) {
    int idx = q.front();
    q.pop();

    Process &p = processes[idx];

    if (p.remaining > quantum) {
        current_time = max(current_time, p.arrival) + quantum;
        p.remaining -= quantum;
    } else {
        current_time = max(current_time, p.arrival) + p.remaining;
        p.remaining = 0;
        p.completion = current_time;
        p.turnaround = p.completion - p.arrival;
        p.waiting = p.turnaround - p.burst;
        completed++;
    }
```

```
        // Enqueue newly arrived processes
        for (int i = 0; i < n; i++) {
            if (i != idx && !in_queue[i] && processes[i].arrival <= current_time &&
processes[i].remaining > 0) {
                q.push(i);
                in_queue[i] = true;
            }
        }

        // Re-enqueue current process if it's not finished
        if (p.remaining > 0) {
            q.push(idx);
        }
    }

    // Output
    double total_wt = 0, total_tat = 0;

    cout << "\nPID\tArrival\tBurst\tWaiting\tTurnaround\tCompletion\n";
    for (const auto& p : processes) {
        cout << p.pid << "\t" << p.arrival << "\t" << p.burst << "\t"
             << p.waiting << "\t" << p.turnaround << "\t\t" << p.completion << "\n";
        total_wt += p.waiting;
        total_tat += p.turnaround;
    }

    cout << "\nAverage Waiting Time: " << total_wt / n;
    cout << "Average Turnaround Time: " << total_tat / n << endl;

    return 0;
}
```

5. Priority Scheduling (Non-Preemptive)

```
#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 finished = false;
};

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

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

    int current_time = 0;
    int completed = 0;
```

```
    while (completed < n) {
        int idx = -1;
        int min_priority = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (!p[i].finished && p[i].arrival <= current_time) {
                if (p[i].priority < min_priority ||
                   (p[i].priority == min_priority && p[i].arrival < p[idx].arrival))
{
                    min_priority = p[i].priority;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            current_time = max(current_time, p[idx].arrival) + p[idx].burst;
            p[idx].completion = current_time;
            p[idx].turnaround = p[idx].completion - p[idx].arrival;
            p[idx].waiting = p[idx].turnaround - p[idx].burst;
            p[idx].finished = true;
            completed++;
        } else {
            current_time++;
        }
    }

    double total_wait = 0, total_turnaround = 0;

    cout <<
"\nPID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\tCompletion\n";
    for (auto &proc : p) {
        cout << proc.pid << "\t" << proc.arrival << "\t" << proc.burst <<
"\t" << proc.priority
```

```
            << "\t\t" << proc.waiting << "\t" << proc.turnaround << "\t\t" <<
proc.completion << "\n";
        total_wait += proc.waiting;
        total_turnaround += proc.turnaround;
    }

    cout << "\nAverage Waiting Time: " << total_wait / n;
    cout << "\nAverage Turnaround Time: " << total_turnaround / n <<
"\n";

    return 0;
}
```

6. FCFS

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

using namespace std;

struct Process {
    int pid;        // Process ID
    int arrival;    // Arrival Time
    int burst;      // Burst Time
    int start;      // Start Time
    int completion; // Completion Time
    int turnaround; // Turnaround Time
    int waiting;    // Waiting Time
};

// Sort by arrival time
bool arrivalCmp(Process a, Process b) {
    return a.arrival < b.arrival;
}

int main() {
    int n;
```

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

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

    sort(p.begin(), p.end(), arrivalCmp);

    int currentTime = 0;
    for (int i = 0; i < n; ++i) {
        p[i].start = max(currentTime, p[i].arrival);
        p[i].completion = p[i].start + p[i].burst;
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        currentTime = p[i].completion;
    }

    double avgWaiting = 0, avgTurnaround = 0;

    cout << "\nPID\tArrival\tBurst\tStart\tCompletion\tWaiting\tTurnaround\n";
    for (const auto& proc : p) {
        cout << proc.pid << "\t" << proc.arrival << "\t" << proc.burst << "\t"
             << proc.start << "\t" << proc.completion << "\t\t"
             << proc.waiting << "\t" << proc.turnaround << "\n";
        avgWaiting += proc.waiting;
        avgTurnaround += proc.turnaround;
    }

    cout << "\nAverage Waiting Time: " << avgWaiting / n;
    cout << "\nAverage Turnaround Time: " << avgTurnaround / n << "\n";

    return 0;
}
```

7. Preemptive Priority Scheduling

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

struct Process {
    int pid, arrival, burst, priority;
    int remaining, completion, turnaround, waiting;
    bool finished = false;
};

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

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

    int current_time = 0, completed = 0;

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

        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= current_time && !p[i].finished && p[i].priority <
min_priority && p[i].remaining > 0) {
                min_priority = p[i].priority;
                idx = i;
```

```
            }
        }

        if (idx != -1) {
            p[idx].remaining--;
            current_time++;

            if (p[idx].remaining == 0) {
                p[idx].completion = current_time;
                p[idx].turnaround = p[idx].completion - p[idx].arrival;
                p[idx].waiting = p[idx].turnaround - p[idx].burst;
                p[idx].finished = true;
                completed++;
            }
        } else {
            current_time++;
        }
    }

    double total_wait = 0, total_turnaround = 0;

    cout <<
"\nPID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\tCompletion\n";
    for (auto &proc : p) {
        cout << proc.pid << "\t" << proc.arrival << "\t" << proc.burst << "\t" <<
proc.priority
             << "\t\t" << proc.waiting << "\t" << proc.turnaround << "\t\t" <<
proc.completion << "\n";
        total_wait += proc.waiting;
        total_turnaround += proc.turnaround;
    }

    cout << "\nAverage Waiting Time: " << total_wait / n;
    cout << "\nAverage Turnaround Time: " << total_turnaround / n << "\n";

    return 0;
}
```

Race Condition

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

// Shared variable
int shared = 1;

// Function declarations
void *function1();
void *function2();

int main()
{
    pthread_t thread1, thread2;

    // Create threads
    pthread_create(&thread1, NULL, function1, NULL);
    pthread_create(&thread2, NULL, function2, NULL);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("The final value of shared variable is: %d\n", shared);

    return 0;
}

void *function1()
{
    int x;
    x = shared;
    printf("Thread1 reads the value of shared variable as %d\n", x);
    x++;
    printf("Local updation by thread1: %d\n", x);
    sleep(1); // Thread1 is preempted by thread2
    shared = x;
    printf("Value of shared variable updated by thread1 is: %d\n", shared);

    return NULL;
}
```

```
void *function2()
{
    int x;
    x = shared;
    printf("Thread2 reads the value of shared variable as %d\n", x);
    x--;
    printf("Local updation by thread2: %d\n", x);
    sleep(1); // Thread2 is preempted by thread1
    shared = x;
    printf("Value of shared variable updated by thread2 is: %d\n", shared);

    return NULL;
}
```

Semaphore

```c
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *function1();
void *function2();
int shared = 1;
pthread_mutex_t mutex;
int main()
{
	pthread_mutex_init(&mutex, NULL); //initialize mutex lock
	pthread_t thread1, thread2;
	pthread_create(&thread1, NULL, function1,NULL);
	pthread_create(&thread2, NULL, function2, NULL);
	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);
	printf("The final value of shared value is : %d\n", shared);
	return 0;
}
void *function1()
{
	int x;
	printf("Thread1 trying to acquire lock\n");
	pthread_mutex_lock(&mutex);
	printf("Thread1 aquire lock\n");
	x=shared;
	printf("Thread1 reads the value of shared variable as %d\n", x);
	x++;
	printf("Local updation by thread1 : %d\n",x);
	sleep(1); //thread1 is preempted by thread2
	shared=x; //thread 1 updates the value of shared variable
	printf("value of shared variable update by thread1 is : %d\n", shared);
	pthread_mutex_unlock(&mutex);
	printf("Thread1 unlock the mutex\n");
}
void *function2()
{
	int x;
	printf("Thread2 trying to acquire lock\n");
	pthread_mutex_lock(&mutex);
	printf("Thread2 aquire lock\n");
	x=shared;
	printf("Thread2 reads the value of shared variable as %d\n", x);
	x--;
```

```
    printf("Local updation by thread2 : %d\n",x);
    sleep(1); //thread2 is preempted by thread2
    shared=x; //thread 2 updates the value of shared variable
    printf("value of shared variable update by thread2 is : %d\n", shared);
    pthread_mutex_unlock(&mutex);
    printf("Thread2 unlock the mutex\n");
}
```

Mutex for race condition

```c
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *function1();
void *function2();
int shared = 1;
pthread_mutex_t mutex;
int main()
{
	pthread_mutex_init(&mutex, NULL); //initialize mutex lock
	pthread_t thread1, thread2;
	pthread_create(&thread1, NULL, function1,NULL);
	pthread_create(&thread2, NULL, function2, NULL);
	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);
	printf("The final value of shared value is : %d\n", shared);
	return 0;
}
void *function1()
{
	int x;
	printf("Thread1 trying to acquire lock\n");
	pthread_mutex_lock(&mutex);
	printf("Thread1 aquire lock\n");
	x=shared;
	printf("Thread1 reads the value of shared variable as %d\n", x);
	x++;
	printf("Local updation by thread1 : %d\n",x);
	sleep(1); //thread1 is preempted by thread2
	shared=x; //thread 1 updates the value of shared variable
	printf("value of shared variable update by thread1 is : %d\n", shared);
	pthread_mutex_unlock(&mutex);
	printf("Thread1 unlock the mutex\n");
}
void *function2()
{
	int x;
	printf("Thread2 trying to acquire lock\n");
	pthread_mutex_lock(&mutex);
	printf("Thread2 aquire lock\n");
	x=shared;
	printf("Thread2 reads the value of shared variable as %d\n", x);
	x--;
```

```
        printf("Local updation by thread2 : %d\n",x);
        sleep(1); //thread2 is preempted by thread2
        shared=x; //thread 2 updates the value of shared variable
        printf("value of shared variable update by thread2 is : %d\n", shared);
        pthread_mutex_unlock(&mutex);
        printf("Thread2 unlock the mutex\n");
}
```

Reader Writter problem

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <shared_mutex>
#include <chrono>
using namespace std;

shared_mutex rw_mutex;

void reader(int id) {
    while (true) {
        rw_mutex.lock_shared();
        cout << "Reader " << id << " is reading...\n";
        this_thread::sleep_for(chrono::milliseconds(500));
        cout << "Reader " << id << " done reading\n";
        rw_mutex.unlock_shared();
        this_thread::sleep_for(chrono::milliseconds(1000));
    }
}

void writer(int id) {
    while (true) {
        rw_mutex.lock();
        cout << "Writer " << id << " is writing...\n";
        this_thread::sleep_for(chrono::milliseconds(1000));
        cout << "Writer " << id << " done writing\n";
        rw_mutex.unlock();
        this_thread::sleep_for(chrono::milliseconds(1500));
    }
}

int main() {
    thread r1(reader, 1), r2(reader, 2), w1(writer, 1);
    r1.join(); r2.join(); w1.join();
    return 0;
}
```

producer Consumer

```
#include <iostream>
#include <thread>
#include <mutex>
#include <semaphore.h>
#include <unistd.h> // for sleep()
using namespace std;

int buffer; // shared buffer (size = 1)
mutex mtx;
sem_t empty, full;

void producer() {
    int item = 1;
    for (int i = 0; i < 5; ++i) {
        sem_wait(&empty);
        {
            lock_guard<mutex> lock(mtx);
            buffer = item;
            cout << "Producer produced: " << buffer << endl;
            item++;
        }
        sem_post(&full);
        sleep(1);
    }
}

void consumer() {
    int item;
    for (int i = 0; i < 5; ++i) {
        sem_wait(&full);
        {
            lock_guard<mutex> lock(mtx);
            item = buffer;
            cout << "Consumer consumed: " << item << endl;
        }
        sem_post(&empty);
        sleep(1);
    }
}

int main() {
    sem_init(&empty, 0, 1); // buffer initially empty
    sem_init(&full, 0, 0);  // no full slots initially
    thread prod(producer);
```

```
    thread cons(consumer);

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

    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```