**Dining Philosopher problem:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_PHILOSOPHERS 5

#define NUM\_CHOPSTICKS 5

void dine(int n);

pthread\_t philosopher[NUM\_PHILOSOPHERS];

pthread\_mutex\_t chopstick[NUM\_CHOPSTICKS];

int main()

{

// Define counter var i and status\_message

int i, status\_message;

void \*msg;

// Initialise the semaphore array

for (i = 1; i <= NUM\_CHOPSTICKS; i++)

{

status\_message = pthread\_mutex\_init(&chopstick[i], NULL);

// Check if the mutex is initialised successfully

if (status\_message == -1)

{

printf("\n Mutex initialization failed");

exit(1);

}

}

// Run the philosopher Threads using \*dine() function

for (i = 1; i <= NUM\_PHILOSOPHERS; i++)

{

status\_message = pthread\_create(&philosopher[i], NULL, (void \*)dine, (int \*)i);

if (status\_message != 0)

{

printf("\n Thread creation error \n");

exit(1);

}

}

// Wait for all philosophers threads to complete executing (finish dining) before closing the program

for (i = 1; i <= NUM\_PHILOSOPHERS; i++)

{

status\_message = pthread\_join(philosopher[i], &msg);

if (status\_message != 0)

{

printf("\n Thread join failed \n");

exit(1);

}

}

// Destroy the chopstick Mutex array

for (i = 1; i <= NUM\_CHOPSTICKS; i++)

{

status\_message = pthread\_mutex\_destroy(&chopstick[i]);

if (status\_message != 0)

{

printf("\n Mutex Destroyed \n");

exit(1);

}

}

return 0;

}

void dine(int n)

{

printf("\nPhilosopher % d is thinking ", n);

// Philosopher picks up the left chopstick (wait)

pthread\_mutex\_lock(&chopstick[n]);

// Philosopher picks up the right chopstick (wait)

pthread\_mutex\_lock(&chopstick[(n + 1) % NUM\_CHOPSTICKS]);

// After picking up both the chopstick philosopher starts eating

printf("\nPhilosopher % d is eating ", n);

sleep(3);

// Philosopher places down the left chopstick (signal)

pthread\_mutex\_unlock(&chopstick[n]);

// Philosopher places down the right chopstick (signal)

pthread\_mutex\_unlock(&chopstick[(n + 1) % NUM\_CHOPSTICKS]);

// Philosopher finishes eating

printf("\nPhilosopher % d Finished eating ", n);

}

Output:

**Philosopher 2 is thinking**

**Philosopher 2 is eating**

**Philosopher 3 is thinking**

**Philosopher 5 is thinking**

**Philosopher 5 is eating**

**Philosopher 1 is thinking**

**Philosopher 4 is thinking**

**Philosopher 4 is eating**

**Philosopher 2 Finished eating**

**Philosopher 5 Finished eating**

**Philosopher 1 is eating**

**Philosopher 4 Finished eating**

**Philosopher 3 is eating**

**Philosopher 1 Finished eating**

**Philosopher 3 Finished eating**

**Let's Understand How the Above Code is Giving a Solution to the Dining Philosopher Problem?**

We start by importing the libraries pthread for threads and semaphore for synchronization. And create an array of 5 p\_threads representing the philosophers. Create an array of 5 mutexes representing the chopsticks.

**The pthread library is used for multi-threaded programming which allows us to run parallel sub-routines using threads. The <semaphore.h> header is used to perform semaphore operations.**

**We initialise the counter i and status message variable as int and a pointer msg, and intialise the semaphore array.**

**We create the philosopher threads using pthread\_create and pass a pointer to the dine function as the subroutine and a pointer to the counter variable i.**

**All the philosophers start by thinking. We pass chopstick(n) (left) to pthread\_mutex\_lock to wait and acquire lock on it.**

**Then the thread waits on the right((n+1) % NUM\_CHOPSTICKS) chopstick to acquire a lock on it (pick it up).**

**When the philosopher successfully acquires lock on both the chopsticks, the philosopher starts dining (sleep(3)).**

**Once the philosopher finishes eating, we call pthread\_mutex\_unlock on the left chopstick (signal) thereby freeing the lock. Then proceed to do the same on the right chopstick.**

**We loop through all philosopher threads and call pthread\_join to wait for the threads to finish executing before exiting the main thread.**

**We loop thorough the chopstick array and call pthread\_mutex\_destroy to destroy the semaphore array.**