**Dining Philosopher Problem Using Semaphores**

**The Dining Philosopher Problem states that K philosophers are seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.**

**Semaphore Solution to Dining Philosopher**

**Each philosopher is represented by the following pseudocode:**

**process P[i]  
 while true do  
 { THINK;  
 PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);  
 EAT;  
 PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])  
 }**

**There are three states of the philosopher: THINKING, HUNGRY, and EATING. Here there are two semaphores: Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or put it down at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.**

**Outline of a philosopher process:**

Var successful: boolean;  
repeat  
 successful:= false;  
 while (not successful)  
  
 if both forks are available then  
 lift the forks one at a time;  
 successful:= true;  
  
 if successful = false  
 then  
 block(Pi);  
 {eat}  
  
 put down both forks;  
  
 if left neighbor is waiting for his right fork  
 then  
 activate (left neighbor);  
 if right neighbor is waiting for his left fork  
 then  
 activate( right neighbor);  
 {think}  
forever

#include *<iostream>*

#include *<thread>*

#include *<mutex>*

#include *<condition\_variable>*

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

std::mutex mutex;

std::condition\_variable S[N];

void test(int phnum)

{

**if** (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

*// state that eating*

state[phnum] = EATING;

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

std::cout << "Philosopher " << phnum + 1 << " takes fork " << LEFT + 1 << " and " << phnum + 1 << std::endl;

std::cout << "Philosopher " << phnum + 1 << " is Eating" << std::endl;

*// notify the philosopher that he can start eating*

S[phnum].notify\_all();

}

}

*// take up chopsticks*

void take\_fork(int phnum)

{

std::unique\_lock<std::mutex> lock(mutex);

*// state that hungry*

state[phnum] = HUNGRY;

std::cout << "Philosopher " << phnum + 1 << " is Hungry" << std::endl;

*// eat if neighbours are not eating*

test(phnum);

*// if unable to eat wait to be signalled*

S[phnum].wait(lock);

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

}

*// put down chopsticks*

void put\_fork(int phnum)

{

std::unique\_lock<std::mutex> lock(mutex);

*// state that thinking*

state[phnum] = THINKING;

std::cout << "Philosopher " << phnum + 1 << " putting fork " << LEFT + 1 << " and " << phnum + 1 << " down" << std::endl;

std::cout << "Philosopher " << phnum + 1 << " is thinking" << std::endl;

test(LEFT);

test(RIGHT);

}

void philosopher(int num)

{

**while** (true) {

take\_fork(num);

put\_fork(num);

}

}

int main()

{

std::**thread** threads[N];

**for** (int i = 0; i < N; i++) {

threads[i] = std::**thread**(philosopher, i);

std::cout << "Philosopher " << i + 1 << " is thinking" << std::endl;

}

**for** (int i = 0; i < N; i++)

threads[i].join();

**return** 0;

}