**DINING PHILOSOPHERS PROBLEM**

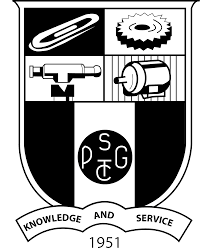
**PROJECT REPORT**

SUBMITTED BY

HASHINI SS – 18PD11

NIKKITHA GS – 18PD21

**SUBJECT**: OPERATING SYSTEMS



Department of Applied Mathematics and Computational Sciences

PSG College of Technology,

Coimbatore -4

**ABSTRACT:**

An implementation of a solution to the Dining Philosophers problem (a classic multi-thread synchronisation problem) using p-threads and semaphores in C programming language. This particular implementation demonstrates the usage of multiple preemptible and cooperative threads of different priorities, as well as mutexes, threads and locks. Avoiding deadlock is one of the main challenges along with ensuring mutual exclusion and non-starvation.

* 1. **INTRODUCTION:**

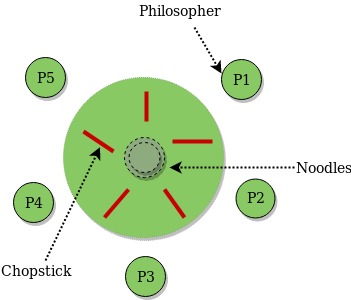
The operating system is a program that links the user and the computer system. This operating system must be capable of controlling resource usage. In the process of designing the operating system, there is a common foundation called concurrency. Concurrent processes are when the processes work at the same time. This is called the multitasking operating system. Concurrent processes can be completely independent of the other but can also interact with each other.

Processes that require synchronization to interact properly are controlled. However, the concurrent processes that interact, there are some problems to be solved such as deadlock and synchronization. One of the classic problems that can illustrate these problems is the Dining Philosophers Problem.

* 1. **DESCRIPTION:**

A dining philosophers problem is a classic synchronization problem. Though, it does not notably represent a real-world problem, it provides a significant learning value, particularly in process synchronization. It is defined in our textbook as follows:

There are 5 philosophers who spend their time just thinking and eating. They sit at the table and each has his/her own chair. There is a rice bowl in the centre of the table and there are 5 chopsticks which of each, is laid next to the philosopher’s hand as shown below. When a philosopher thinks, he will not interact with others. Once in a while, a philosopher is hungry and tries to pick up chopsticks on his left and right-hand sides. A philosopher can pick only one chopstick at a time. When a hungry philosopher has chopsticks on both hands, he can start eating. When he is done eating, he puts both chopsticks down and starts thinking again.



**FIG 1.2.1 Dinner Time**

* 1. **SYSTEM CALLS USED:**

**pthread\_create:**

int pthread\_create(pthread\_t \**thread*, const pthread\_attr\_t \**attr*, void \*(\**start*)(void \*), void \**arg*);

The pthread\_create() function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes are used (see [pthread\_attr\_init()](https://www.mkssoftware.com/docs/man3/pthread_attr_init.3.asp)). If the attribute object is modified later, the thread's attributes are not affected. If *thread* is not NULL, pthread\_create() stores the ID of the created thread in the location referenced by *thread*.

At creation, the thread executes *start*, with *arg* as its sole argument. The calling function must ensure that *arg* remains valid for the new thread throughout its lifetime. If *start* returns, the effect as if there was an implicit call to [pthread\_exit()](https://www.mkssoftware.com/docs/man3/pthread_exit.3.asp) using the return value of *start* as the exit status. Note that the thread in which main() was originally invoked differs from this. When it returns from main(), the effect as if there was an implicit call to [exit()](https://www.mkssoftware.com/docs/man3/exit.3.asp) using the return value of main() as the exit status.

If pthread\_create() fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

**pthread\_join:**

int pthread\_join(pthread\_t *thread*, void \*\**value\_ptr*)

The *pthread\_join*() function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread\_join*() call with a non-NULL *value\_ptr* argument, the value passed to *[pthread\_exit](https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_exit.html)*[()](https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_exit.html) by the terminating thread shall be made available in the location referenced by *value\_ptr*. When a *pthread\_join*() returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread\_join*() specifying the same target thread are undefined. If the thread calling *pthread\_join*() is canceled, then the target thread shall not be detached.

**pthread\_mutex\_lock:**

int pthread\_mutex\_lock(pthread\_mutex\_t \**mutex*);

The pthread\_mutex\_lock() function locks the specified mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD\_MUTEX\_NORMAL, deadlock detection is not provided. Attempting to relock the mutex causes deadlock.

If the mutex type is PTHREAD\_MUTEX\_ERRORCHECK, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error is returned.

If the mutex type is PTHREAD\_MUTEX\_RECURSIVE, the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Every time a thread relocks this mutex, the lock count is incremented by one. The mutex must be unlocked as many time as it was locked for the mutex to be released.

If the mutex type is PTHREAD\_MUTEX\_DEFAULT, attempting to recursively lock the mutex results in undefined behavior.

**TOOLS AND TECHNOLOGY:**

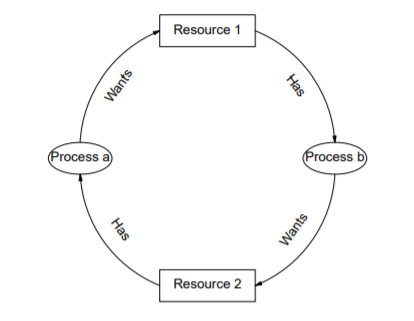
#### Mutex Locks:

#### There are a couple of mechanisms that implement locks. Mutex (mutual exclusion) is the fundamental synchronization technique. The idea is simple: whenever a work unit accesses the **critical section**, it first needs a lock that guarantees no one else at this time is accessing the critical section. When the work unit exits the critical section, it returns the lock for other work units to access.

In the dining philosopher problem, we can implement an algorithm with mutexes that guarantee the philosophers not to be interrupted when they are changing their states.

**Deadlock:**

Another danger of concurrent programming is deadlock. A race condition is produces incorrect results, where a deadlock results in the deadlocked processes never making any progress. In the simplest form it’s a process waiting for a resource held by a second process that’s waiting for a resource that the first holds.



**FIG 1.4.1 Deadlock**

More formally there are 4 conditions for deadlock:

• Mutual Exclusion: Resources can only be held by at most one process. A process cannot deadlock on a sharable resource.

• Hold and Wait: A process can hold a resource and block waiting for another.

• No Pre-emption: The operating system cannot take a resource back from a process that has it.

• Circular Wait: A process is waiting for a resource that another process has which is waiting for a resource that the first has.

A deadlock condition in the simulation dining philosophers problem occurs when at one time; all the philosophers get hungry simultaneously, and all philosophers take the chopsticks in his left hand. By the time the philosophers try to take the chopsticks in the right hand, there will be a deadlock condition since all philosophers will both waiting for chopsticks on the right.

#### POSIX Threads:

#### POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.Here are a few characteristics of p-threads.

#### LIGHT WEIGHT:

#### When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

#### EFFICIENT:

#### The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.

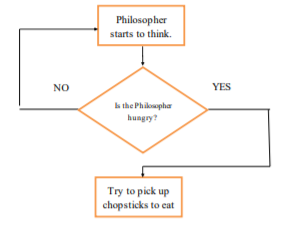
#### For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer.

#### Shared Memory Model:

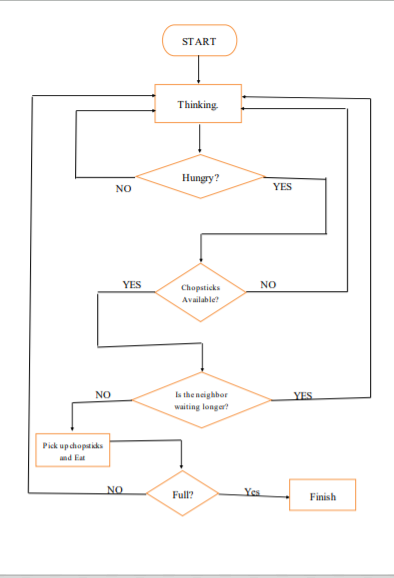
#### All threads have access to the same global, shared memory .Threads also have their own private data.Programmers are responsible for synchronizing access (protecting) globally shared data.

* 1. **WORKFLOW:**

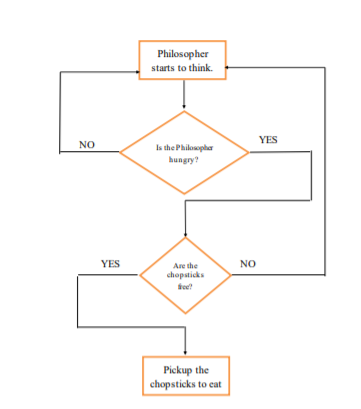
There are five bowls of noodles in front of each philosopher and one chopstick in between each philosopher. The philosophers spend time thinking (when full) and eating (when hungry). When hungry, the philosopher will take two chopsticks (in the left hand and right hand) and eat. If there are philosophers who took two chopsticks, then the next two philosophers who was eating must wait until chopsticks put back. It can be implemented with the WAIT and SIGNAL.



**FIG 1.5.1 Flowchart for Deadlock**

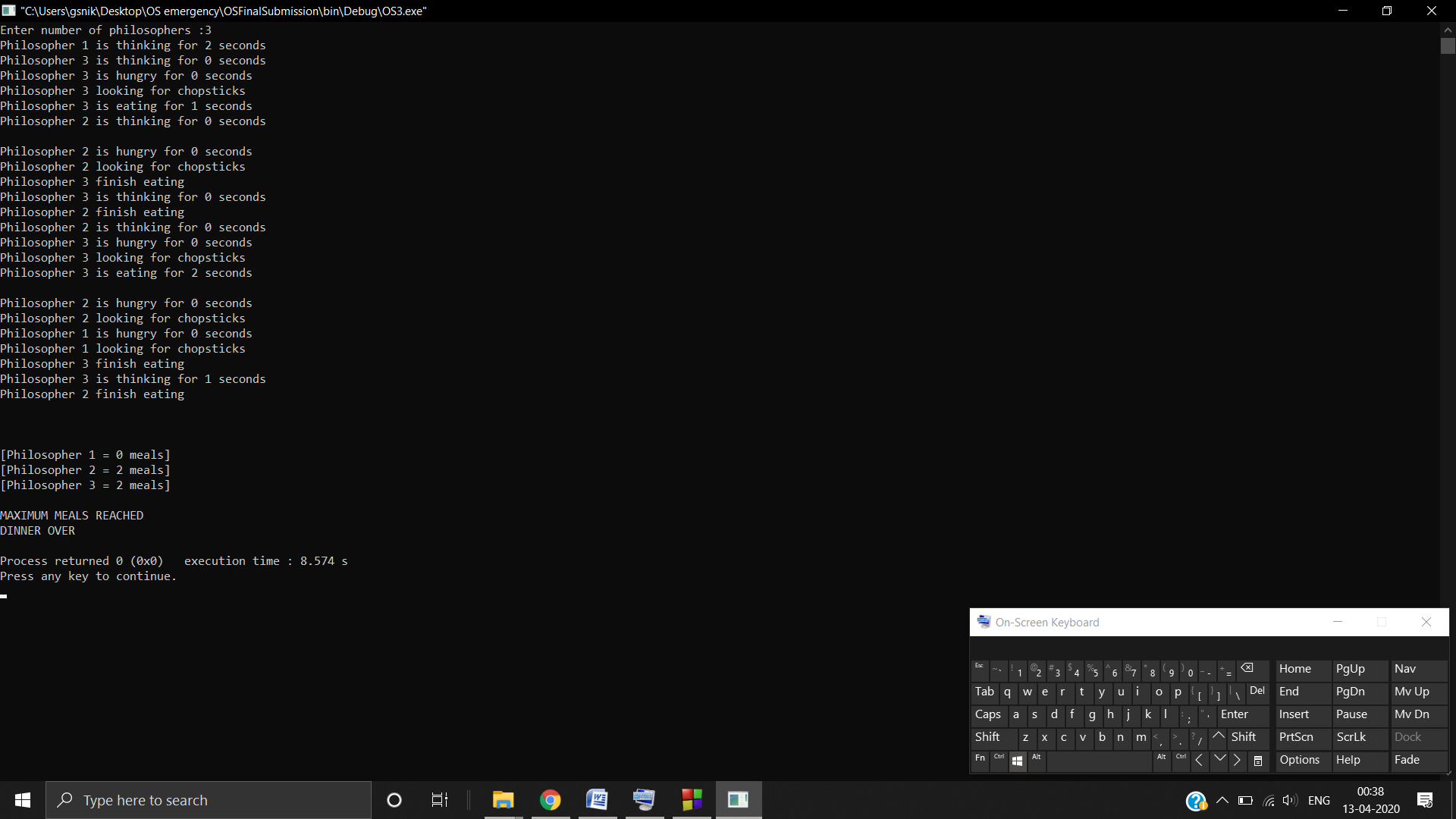


**FIG 1.5.2 Flowchart for problem solution**



**FIG 1.5.3 Flowchart to indicate starvation**

* 1. **RESULTS AND DISCUSSION:**



**FIG 1.6.1 Sample Output**

**Our solution:**

1. The philosopher starts in THINKING state, then he/she is in HUNGRY state.

Then if the neighbouring philosophers are not eating then he/she goes into EATING state.

2. Returning chopstick after eating i.e. changing the state of the philosopher also checking if the meals reached their limit and exiting the program.

3. The philosopher is hungry and checks for chopstick availability in obtain\_chopstick function.

4. Checking the neighbours and change state to EATING in check\_chopstick function.

**NOTE**: There is a lock to keep CR safe while obtaining and returning chopsticks.

**1.7 CONCLUSION:**

The Dining Philosophers problem is a classic concurrency problem dealing with synchronization.

* 1. **BIBLIOGRAPHY:**

**1.8.1. Books:**

“The Dining Philosophers Problem” by Ras Dr. Jim

“Operating System Concepts” by Albert Silberschatz

“Operating Systems: Internals and Design Principles” by William Stallings

**1.8.2 Websites:**

[http://www.iosrjournals.org/iosr-jce/papers/Vol18-issue6/Version- 4/H1806044550.pdf](http://www.iosrjournals.org/iosr-jce/papers/Vol18-issue6/Version-%20%20%20%20%20%20%20%20%20%204/H1806044550.pdf)

<https://docs.zephyrproject.org/latest/samples/philosophers/README.html>

<https://www.youtube.com/watch?v=NbwbQQB7xNQ>

[https://www.geeksforgeeks.org/dining-philosopher-problem-using semaphores/](https://www.geeksforgeeks.org/dining-philosopher-problem-using%20semaphores/)

<https://www.youtube.com/watch?v=9f1oOMX3mP4>