THREAD SYNCHRONIZATION

A report on package submitted by

HARISH KUMAR A

Roll no. 18PT13

SIVAGIRI VISAKAN T

Roll no. 18PT36

18XT44 - OPERATING SYSTEMS

April 2020

M.Sc. THEORETICAL COMPUTER SCIENCE

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES

PSG COLLEGE OF TECHNOLOGY

COIMBATORE – 641 004.

ABSTRACT

This report aims to cover the basics of synchronization of processes and threads in a computer system and deals in detail about the “Dining Philosophers Problem” and explains a solution to the problem. It also provides an example of a threaded server model and provides a way to synchronize the shared resource among the threads.

THE DINING PHILOSOPHERS PROBLEM

1.1 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 multiple 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 interact with each other and those that share one or many resources require synchronization to function properly. When concurrent processes interact, there are some problems to be solved such as deadlock and synchronization.

One of the classic problems that are used to illustrate the need for synchronization among processes is the dining-philosophers problem. It is often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals. Soon after, Tony Hoare gave the problem its present formulation.

1.2 Dining-Philosophers Problem

This section gives a description of the dining-philosophers problem.

There are 5 philosophers dining at a round table(the original problem by Dijkstra's is stated with five philosophers) with bowls of spaghetti. A fork is placed between each pair of adjacent philosophers as in Figure-1.1 shown below. As a consequence, each philosopher has a fork on his left and on his right-hand side and there will be a total of *N* forks.

Each philosopher has two states - think and eat. The philosophers will alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can only take the fork on their right or the one on their left as they become available and they cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed. The problem is how to design a discipline of behavior such that no philosopher will starve; *i.e.*, each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

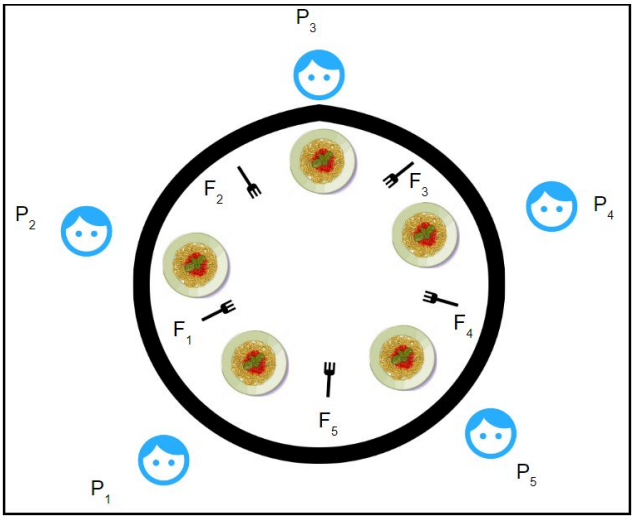

Figure 1.1 An illustration of the dining philosophers problem with five philosophers and five forks in the table.

In a computing system, this can be visualized as five processes requiring two units of resources of the same type for doing some task. There are in total five copies of the resource, but a process is only allowed to take only a fixed two among the five copies and each resource is shared by 2 processes.

1.3 Synchronization preliminaries

This section lists some of the concepts that are required to understand the problems in process synchronization and a method that is helpful ins synchronizing them.

1.3.1 Deadlock

Deadlock is a situation where any process enters a waiting state because another waiting process is holding the demanded resource.

In other words, a Deadlock is said to be present in a system of programs if multiple programs or threads or processes are prevented from using a resource and are waiting for the resource or the result to be left free by another process that acquired the resource.

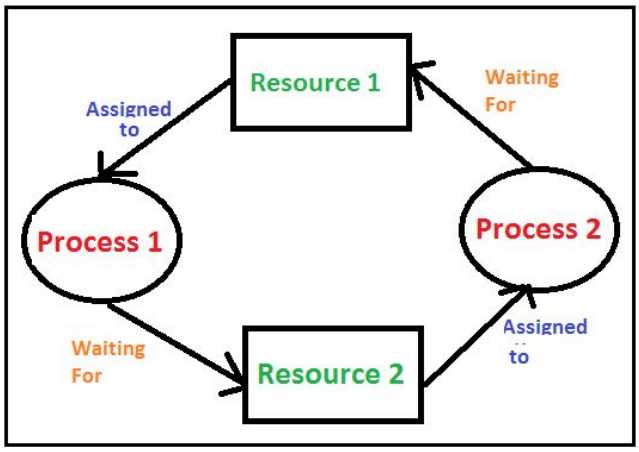

Figure 1.2 A flowchart of a scenario of a deadlock

There is a state of Deadlock as shown in Figure 1.2 where Process 1 is expecting a resource, which happens to be the result of another process, Process 2 which needs the resource, Resource 1 held by Process 1.

Example of a Deadlock:

- A real-world example would be traffic, which is going only in one direction.
- Here, a bridge is considered a resource.
- So, when Deadlock happens, it can be easily resolved if one car backs up (Preempt resources and rollback).
- Several cars may have to be backed up if a deadlock situation occurs.
- So starvation is possible.

1.3.2 Starvation

Resource starvation is a problem encountered in concurrent computing where a process is perpetually denied necessary resources to process its work.

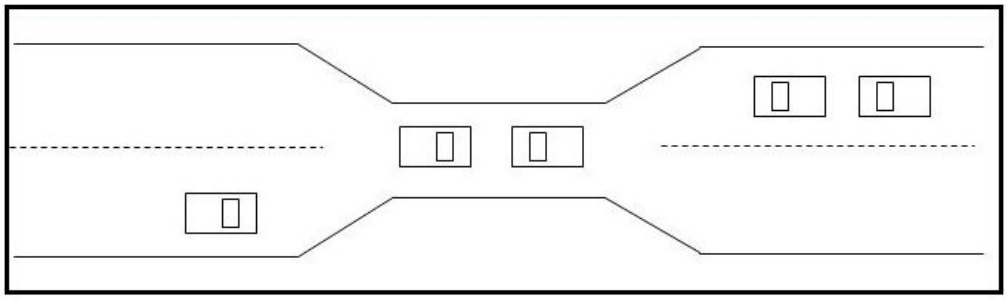

Figure 1.3 A traffic on a bridge

As from the example given above which is reflected in Figure 1.3 for a deadlock resource starvation is also present when the bridge is considered to be the resource. And if vehicles from only one lane are given access to the bridge the resource bridge is being blocked for the other lane by the lane in use. Hence the process; in this case, the vehicle deprived of using the bridge is in resource starvation.

Starvation is usually caused by an overly simplistic scheduling algorithm. For example, if a (poorly designed) multi-tasking system always switches between the first two tasks while a third never gets to run, then the third task is being starved of CPU time. The scheduling

algorithm, which is part of the kernel, is supposed to allocate resources equitably; that is, the algorithm should allocate resources so that no process perpetually lacks necessary resources.

1.3.4 Mutual exclusion

A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access to the resource. Upon releasing the resource, the thread unlocks the mutex.

Mutex comes into the picture when two threads work on the same data at the same time. It acts as a lock and is the most basic synchronization tool. When a thread tries to acquire a mutex, it gains the mutex if it is available, otherwise, the thread is set to sleep condition. Mutual exclusion reduces latency and busy-waits using queuing and context switches. A mutex can be enforced at both the hardware and software levels.

1.3.5 Semaphore

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that is used for process synchronization.

The definitions of wait and signal are as follows :

- Wait: The wait operation decrements the value of its argument S if it is positive. If S is negative or zero, then no operation is performed.
- Signal: The signal operation increments the value of its argument S.

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:

- **Counting Semaphores:**

  These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count is automatically incremented and if the resources are removed, the count is decremented.
- **Binary Semaphores:**

  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when the semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores :

Some of the advantages of semaphores are as follows:

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine-independent code of the microkernel. So they are machine-independent.

1.4 Various Scenarios in the Dining Philosophers Problem

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider the following cases in accordance with the below example:

Considering the number of philosophers be 5, and are named as P_1, P_2, P_3, P_4, P_5 and the forks named as F_1, F_2, F_3, F_4, F_5 respectively are placed as given in Figure 1.4.

Case I (deadlock):

- A philosopher, P_1 feels Hungry;
- Philosopher P_1 thinks until the left fork i.e F_1; is available; when it is, he picks it up;
- Philosopher P_1 thinks until the right fork is available; when it is, pick it up;
- When both forks are held, Philosopher P_1 eats for a fixed amount of time;
- Then, puts the right fork down;
- Then, puts the left fork down;
- Repeat from the beginning with any philosopher.

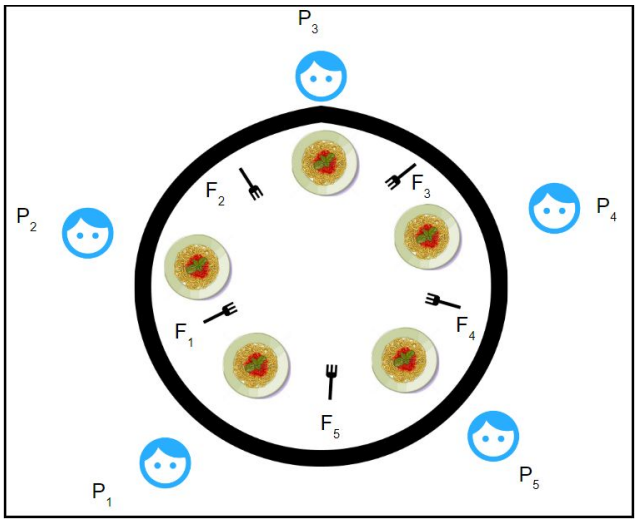

Figure 1.4 Depiction of the philosophers and forks being numbered

This attempted solution fails because it allows the system to reach a 'deadlock state', in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is thinking and waiting; as shown in the Figure-1.5 below for the fork in the right to become available. With the given instructions, this state can be reached, and when it is reached, each philosopher will eternally wait for another (the one to the right) to release a fork.

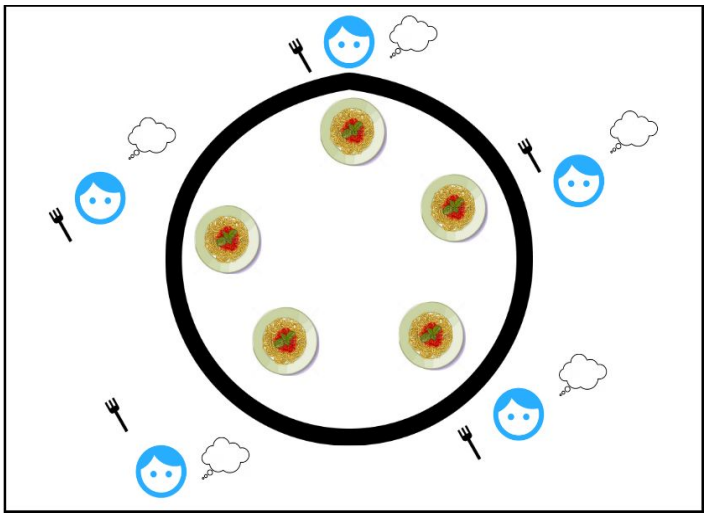

Figure 1.5 Figure depicting the case of a deadlock in the dining philosophers problem

Case II (race condition) :

- Two philosophers P_1 and P_2 feel hungry simultaneously.
- They think until the left fork is available; when it is, pick it up;
- P_1 and P_2 pick their respective left forks namely, F_1 and F_2.
- They think until the right fork is available; when it is, pick it up; (For philosopher P_1 his right fork namely F_5 is free.)
- When both forks are held, eat for a fixed amount of time;
- Then, put the right fork down;
- Then, put the left fork down;
- Repeat from the beginning.

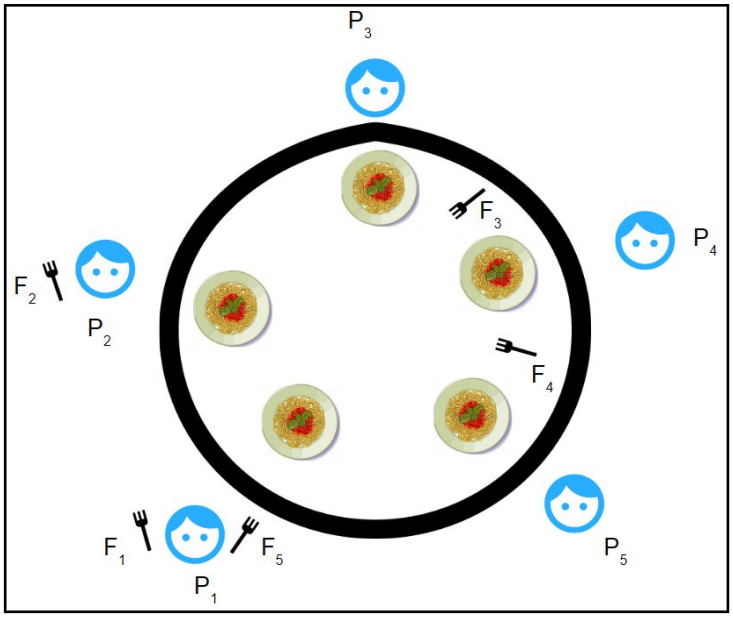

Figure 1.6 Figure depicting case 2

In this case, a philosopher at the right has the respective right fork as shown in Figure 1.6. So, the philosopher eats for a particular amount of time. Eventually, when the philosopher has finished eating, both forks are kept on the table and the other philosopher starts eating. This case might lead to a 'Race Condition'. To remove this Race Condition, Semaphores are used.

Case III (resource starvation) :

- Philosopher P_1 feels hungry.
- The philosopher P_1 thinks until the left fork is available; when it is, pick it up;
- The philosopher thinks until the right fork is available; when it is, pick it up;
- When both forks are held, eat for a fixed amount of time;
- While philosopher P_1 is eating philosopher P_2 feels hungry;
- But his respective right fork is currently in use by philosopher P_1. So philosopher P_2

is blocked from eating and is forced to continue eating.

- Now philosopher P_3 feels hungry.
- According to the rules, philosopher P_3 can wait and think until the left fork is available; as it is in this case fork F_3 is free, philosopher P_3 picks it up;
- The philosopher thinks until the right fork is available namely F_4; when it is, pick it up;
- When both forks are held, eat for a fixed amount of time;
- Then, put the right fork down;
- Then, put the left fork down;
- Repeat from the beginning.

The below Figure 1.7 depicts the above case mentioned.

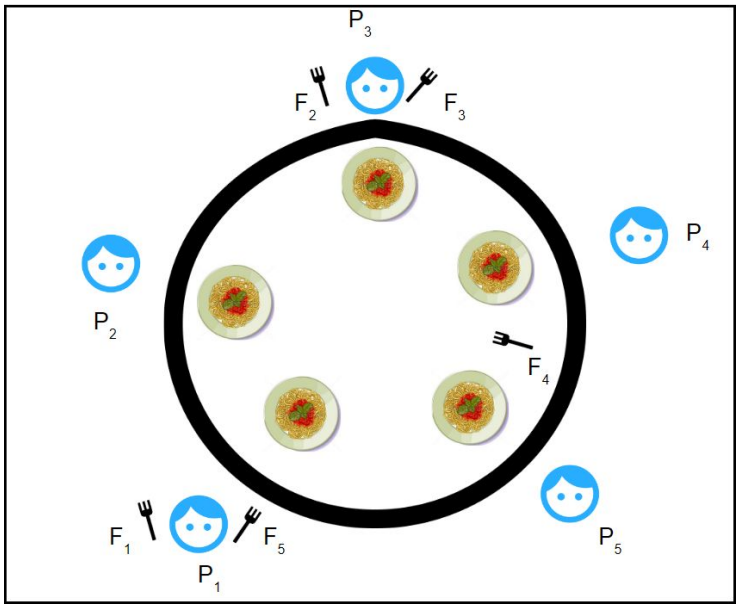

Figure 1.7 Figure depicting case 3

Case IV (case of resource starvation which includes deadlock):

- Say case III is finished as a step, and similarly Philosopher P_2 and P_4 eat, then Philosopher, P_5 feels Hungry, but Philosopher P_1 happens to request for a fork before P_5;
- Philosopher P_1 picks up the fork;
- Philosopher P_1 thinks until the right fork is available; when it is, pick it up;
- When both forks are held, Philosopher P_1 eats for a fixed amount of time;
- Then, puts the right fork down;
- Then, puts the left fork down;
- Repeat from the beginning with any philosopher.

Philosopher P_5 never gets to eat if this case holds for every routine of the algorithm is defined.

Resource Starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual Exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Complex

systems such as operating system kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, and data corruption are to be avoided.

1.5 Solutions to the Dining Philosophers Problem

There have been many variants of solutions proposed overtime for this dining philosophers problem. A valid solution for the dining philosophers problem has two important requirements:

- The algorithm should avoid deadlock i.e it shouldn't have a probability of causing a deadlock
- The algorithm should avoid resource starvation i.e every philosopher should get a chance to eat.

Here two of the solutions namely, the resource hierarchy solution and the waiter solution are addressed and explained.

1.5.1 Resource Hierarchy Solution (arbitrator solution)

This solution was proposed by Dijkstra himself which was to introduce a hierarchy and that hierarchy is to be followed by all the philosophers. Each philosopher can take only two forks adjacent to them.

Since the forks are numbered, among the forks that a philosopher can pick, one of the forks is a fork which is numbered greater than the other fork. There is an extra restriction here that among the two forks, each philosopher will, or rather should always pick the fork with the lower index first, and then the higher indexed fork.

Algorithm:

- Think
- Take lower-indexed fork
- Take higher-indexed fork
- Eat
- Put down forks
- Repeat

In this context, philosopher P_1 will always try to pick the fork, F_1 and then fork F_5, because 1 is the lower-indexed fork and 5 is the highest-indexed fork in accordance with philosopher P_1 is allowed to pick as in Figure 1.8.

Similarly, Philosopher P_2 will always try to pick the fork F_1 and then fork F_2 and so on. This will ensure that at least one philosopher will have two forks and finish his meal and will not lead to a deadlock where there is no progress.

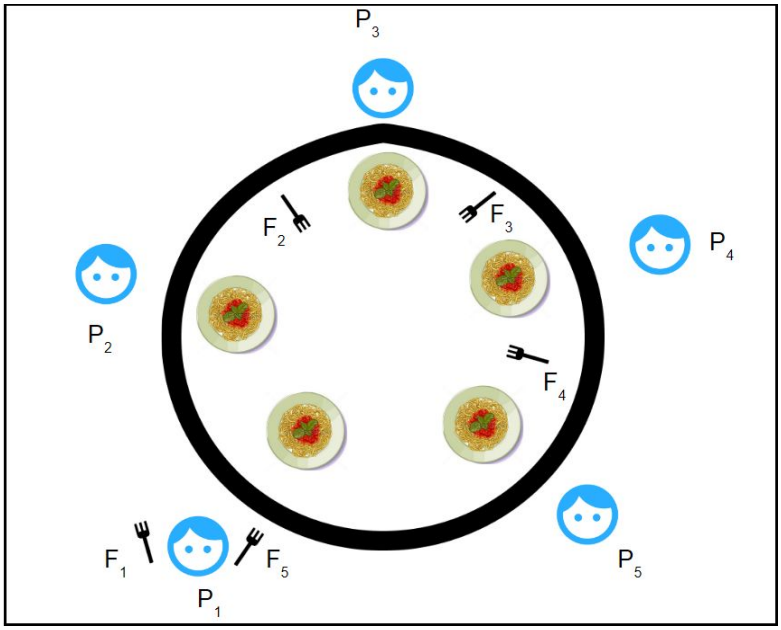

Figure 1.8

Possible worst-case scenario:

The worst case that can happen in this solution is all the philosophers finish thinking at the same time and try to grab the forks at the same instance.

Philosopher P_1 can take only forks F_1 and F_5, lower of which is 1, therefore philosopher P_1 takes the fork F_1. Philosopher P_2 will try to take F_1, which is the lower indexed fork he can

take but will find it is already taken, so philosopher P_2 will wait. Philosopher P_3 will take the fork F_2. Philosopher P_4 will take the fork F_3. Philosopher P_5 will take the fork F_4.

Now, there is a fork remaining free, that can be used by philosopher P_1 to complete his meal as in Figure 1.9 shown above. Hence, even in the worst case, at least one philosopher happens to eat.

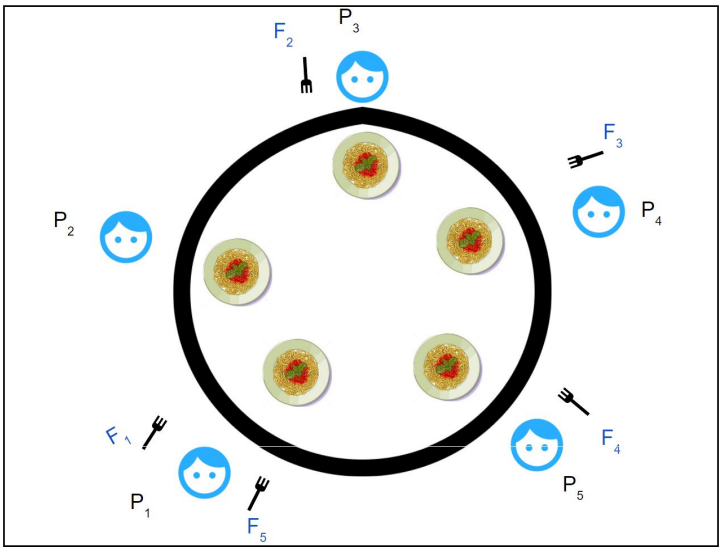

Figure 1.9

After philosopher P_1 finishes his meal, he'll put down the forks F_1 and F_5, which can be used by P_5 to complete his meal and the other philosophers respectively. There is an asymmetry here, as opposed to the naive attempt, where all philosophers go for the left fork first. Some philosophers grab their right fork first and some try to grab their left one first.

Hence, a circle can be avoided here and there is no possibility of deadlock. This assures that there is some kind of progress in the system.

1.5.2 Waiter Solution

A Waiter is introduced into the picture of the dining philosophers and the waiter lays down a rule i.e; only one philosopher at a time can pick up one or a maximum of two forks. This means that they have to ask permission from the waiter in order to eat the spaghetti.

Algorithm:

- Think
- Ask Permission from the waiter to take a fork
- Ask Permission from the waiter to take the second fork
- Eat
- Put down forks
- Repeat

The waiter here is to introduce the idea of a monitor mechanism that monitors and controls the order in which the philosophers think and eat and in the case of process management and process synchronization to be specific it is done using a lock mechanism.

Locks can help in ensuring that any process is uninterrupted by controlling access. Choosing forks need to be atomic. A philosopher, in this case, shouldn't be interrupted when trying to pick up a fork. If that is the case, then this would lead to the same situation as in case I in section 1.4 leading to a deadlock.

Possible worst-case scenario:

The worst case that can happen in this solution is all the philosophers finish thinking at the same time and try to grab the forks at the same instance. Taking case I in section 1.4, there is no chance of giving all of the locks to all the philosophers as the lock is atomic and at least one philosopher gets to eat as shown below in Figure 1.10.

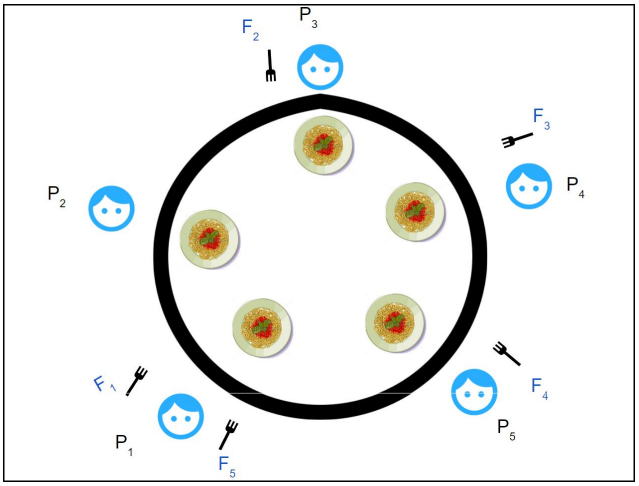

Figure 1.10 at least one of the philosophers get to eat if the waiter is introduced

Thus, this solution avoids deadlock and also starvation in accordance with the introduction of the locking mechanism.

1.6 Operating System Functionalities Used

This section describes the system calls and the APIs provided by the operating system that is used in the development of this package.

1.6.1 shmget

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflag);
```

The `shmget` system call is used to reserve memory in the shared memory segment. The shmget system call shall return the shared memory identifier associated with the key or can be used to create based on the `shmflag.`

1.6.2 shmat

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

The `shmat` function attaches the shared memory segment identified by `shmid` to the address space of the calling process.

The attaching address is specified by shmaddr.

1.6.3 sem_wait

The sem_wait() decrements (locks) the semaphore pointed to by sem.

If the semaphore's value is greater than zero, then the decrement proceeds and the function returns immediately. If the semaphore currently has the value zero, then the call blocks

until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

1.6.4 sem_post

sem_post() increments (unlocks) the semaphore pointed to by sem.

If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

1.7 Tools and technology

The tools and technologies used in the development of this package and a brief description of them are provided in this section.

1.7.1 The C language

C is a statically typed and compiled general-purpose procedural programming language which provides low-level access to the computing system. It was originally developed to make utilities for the Unix operating systems and then the major part of Unix kernel has been written in C.

1.7.2 Qt

Qt is a free and open-source widget toolkit and a cross-platform framework for creating graphical user interfaces. Qt provides APIs to develop programs in various languages such as JavaScript, C++, Python, etc. The Qt framework with C++ has been used in this package.

1.7.3 ncurses

`ncurses` is a programming library providing APIs for developing text-based user interfaces that can be run from right within the terminal. It is a successor to the `curses` library. The bindings for ncurses are available in a variety of languages like Python, C, JavaScript, Ruby, etc. `ncurses` bindings for C has been utilized in this project.

1.8 Workflow

This section describes the implementation of the package using the tools and technologies described previously.

The total number of philosophers is fixed before the start of the program. The states of the philosophers are maintained in a separate structure, named `struct philosopher_info` and the forks' state in a structure named `struct fork_state_info` in a separate file `philosophers_base.h` that is included in source files implementing solutions.

The `pthread.h` library is used to create threads to simulate multiple philosophers. The different threads execute a function `void philosopher(int index)`, whose definitions change for different solutions `naive.c`, `resource_hierarchy.c` and `waiter.c`. Each thread runs independently of others but may wait for some resources that are being used by other threads.

Forks are considered as mutually exclusive objects and are implemented as binary semaphores.

When a particular program is run, it creates a shared memory to store the philosophers' and forks' states. A thread that simulates a philosopher *i* will update the corresponding states for that philosopher. It is to be noted that there is no synchronization required for accessing the memory to update the states, as every thread uses different memory locations to update the state and is not interrupted by others.

This state of the program is implemented as a shared memory so that it can be accessed from the Qt program which runs as an independent process. The Qt program has a GUI visualization of the five philosophers and the table similar to that seen in Figure 1.1. This visualization is updated with the forks moving to appropriate places to reflect the actual state of the dining room. A log is also printed which shows the step by step flow of activities in the

dining room. An option to pause the visualization is provided to pause and understand the particular scenarios described.

The deadlock can be simulated in the naive solution, while there is no possibility for it in other solutions.

The `ncurses` toolkit also provides a text-based visualization of the current state of the system of the philosopher's, without the need to start a separate process for visualization.

1.9 Results and discussions

The solutions proposed discussed in section 1.5, namely resource hierarchy solution and the waiter solution are algorithms that prevent problems in a multiprogramming environment that runs multiple programs concurrently. These two solutions have been implemented in C as a way to control the resource (forks in the dining-philosophers problem) allocation in such a way that it doesn't lead to a deadlock.

A graphical user interface has been implemented with Qt to better visualize the process of the philosopher's dining as the program runs, which can be used.

This idea of many philosophers sharing forks can be extended to a computer system where multiple programs are sharing some common resources such as access to memory, the filesystem, the networking, etc. The cases described in this report can also be visualized in terms of processes and resources instead of philosophers and forks.

In a system with multiple threads/processes running, with shared resources, every process's need for a resource should be addressed. Any process holding such a resource for more than a specific iteration in a period often leads to resource starvation.

A deadlock situation should never occur for an application program and for that the synchronization has to be done properly. This has been in this package.

CONCLUSION

In a real-world scenario where one task which has many other subtasks which share similar task-related resources are often less efficient in the way, these systems of the process(s) are programmed or otherwise scheduled.

An efficient solution that monitors and schedules its threads holistically which not only balances the threads but also does preserve the systematic approach to the process and also ensures a concurrent environment. The above-given solutions in section 1.5 are a testimony for a system of processes handling problems where multiple programs are running concurrently.

BIBLIOGRAPHY

Books and articles

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, 2009. *Operating System Concepts, 8th Edition.*
2. Zuhri Ramadhan, Andysah Putera Utama Siahaan, 2016. *Dining Philosophers Theory and Concept in Operating System Scheduling*.

   Vol18-Issue6.

Websites

1. https://linux.die.net/man/
2. https://invisible-island.net/ncurses/man/
3. https://doc.qt.io/qt-5/reference-overview.html
4. https://pages.mtu.edu/~shene/NSF-3/e-Book/index.html
5. http://web.eecs.utk.edu/~mbeck/classes/cs560/560/notes/Dphil/lecture.html