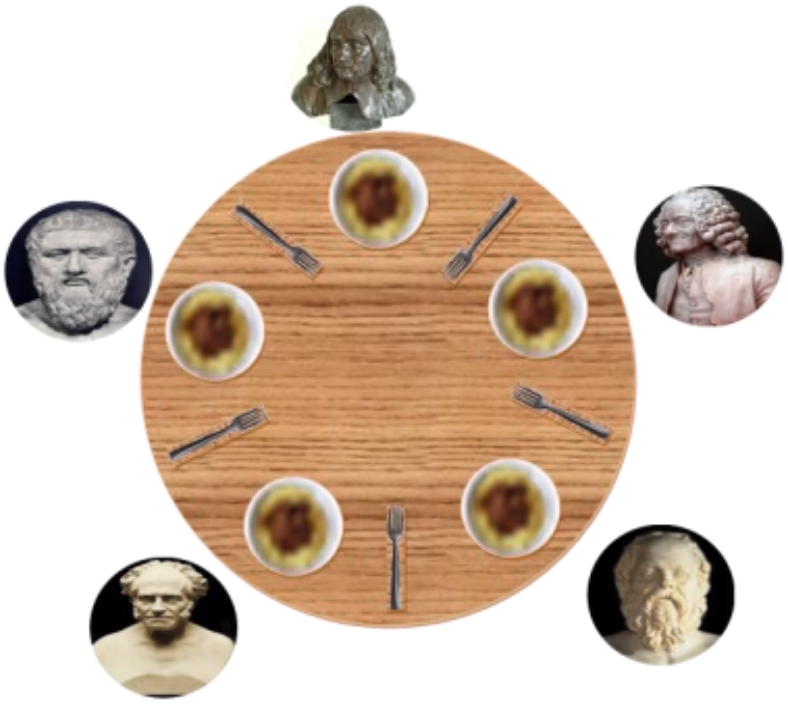


The Dining Philosopher's Problem

Table of Contents

Theoretical Background

The operating system responsible for the proper functioning of the computer. And at any instance of a computer's operation, multiple processes are being run on the computer. A process is an instance of a program in execution. Whereas a program can be considered as a list of commands that are stored in a file in memory. When the program is executed, its running instance is known as a process. When a computer boots up, many processes are created or started by the operating system which in turn perform essential and non-essential tasks. Essential tasks may include process of the File Explorer, Antivirus and the process of the graphics driver. Non-essential tasks may include periodic checking of the email, third party software, game clients etc.

Process Synchronization

One major problem that came into existence by using computers which contain multiple processes running in parallel regardless the system in single core or multi core is the synchronization problem. In an operating system, multiple processes when run in parallel require access to the shared resources. However, if multiple processes try to access these shared resources at the same time, data integrity may be lost and it may lead to data inconsistency throughout the system. Therefore, some sort of order and coordination is required between these processes, this is known as data synchronization. Synchronization is very important in an operating system to ensure proper functioning of a computer. From the perspective of the end user, the multiprocessor system should not allow disparity of data. By synchronization, we can ensure that the end users' data is not tampered with as the consumer shouldn't be concerned with the working of the various processors. It should be made sure that the end user feels that his/her data is the same even though tasks are divided or are running in parallel and numerous processes share and read/write on the same data.

Race Conditions

In inter-process Communication, when a shared resource is accessed by the process or when two processes are executing the same code, there is a very high risk that the data integrity is lost and the value of the shared variable is rendered incorrect. When two or more processes are reading and writing some shared data, the final result usually depends on who runs precisely when, these situations are known as race conditions. Race conditions are widely common in parallel systems and especially in multiprocessor systems. Whenever a computer system is developed, the computer designers are required to put extensive care into solutions that avoid race conditions generally.

For example, if we consider the following code, where x is the shared data.

```
int i, x = 0;
for(i = 0; i<5; i++)
{
x++;
}
```

We would think that after the execution of this code by multiple threads, the value of y will be 5. But we would in fact, be wrong because each thread changes the value of the shared variable y and the data integrity at the end is lost. This condition is known as the race condition. Solving synchronization problems involves solving race conditions and all the primitives that are designed to solve synchronization problem showcase their effectiveness in solving race conditions

Critical Section

In order to avoid race conditions, we need to make sure that no two processes access the shared data at the same time. Which means that no files and memory which are shared between the two parallel mechanisms, are accessed at the same time. This is known as mutual exclusion and is usually implemented in some forms to avoid synchronization problems. In order to classify the section of the program where any shared data is used, programs/engineers usually use the term critical section. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

Proposed Solutions

Spinlocks

A spin lock is a very effective way of achieving mutual exclusion between processes. In parallel operating systems and applications, spinlocks are widely utilized. A spin lock is a lock which causes the thread or the process which is trying to acquire it to wait in a spin state. During this state, the process is repeatedly checking to see if the lock is available or not. So essentially, we can say that a lock that uses busy waiting is called a spin lock. Due to the fact that they avoid operating system rescheduling, we can use spinlocks without any doubt if the process or the threads are supposed to be short-lived.

Semaphores

A semaphore is a variable which is used to count the number of wakeups which may be saved for some future use. If a semaphore has value 0, it means that no wakeups are saved. If the semaphore has a positive value, it means that some sort of wakeup is pending.

TSL Instructions

The TSL (Test and Set Lock) allows binary value hardware synchronization. It is an instruction which return the old value of a memory location and sets the location value to 1. This operation is atomic. Due to this, if multiple processes access the location. If a process is current performing testing and setting, the other processes are not allowed to test and set the variable. This mechanism is utilized to develop a lock and solve the synchronization problem.

Dining Philosopher's Problem

Background

The dining philosopher's problem was a synchronization problem which was posed by Dijkstra in 1965. This problem became so famous that it is widely considered as the hallmark of synchronization problems. Now whenever a new synchronization primitive is introduced to the world, it seems compulsory to demonstrate its effectiveness in solving the Dining Philosopher's problem.

The Problem

The crux of the problem is as follows: Five philosophers are seated on a roundtable. The philosophers are having spaghetti as dinner. The problem is that the spaghetti is very slippery and requires two forks to eat. However, there is only one fork per philosopher. So, whenever a philosopher needs to eat, he/she needs two forks to eat.

The philosopher's lead a rather simple life which consists of alternation between eating and thinking. This simplicity is obviously an abstraction. Now whenever a philosopher becomes hungry, he tries to acquire his left and right forks one by one. If the forks are acquired, the philosopher starts to eat. If the forks are not acquired, the philosopher remains hungry.

The problem is to design such an algorithm which ensures that no philosopher starves forever and continues his life which alternates between thinking and eating. Now the intuitive solution would be to make all the philosophers think until the left fork is available. When available, pick it up and then see if the right fork is available and pick it up. Then put right fork down and then put the left fork down and then repeat. This solution looks that it might work however it will fail because in the scenario that a philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available. With the given instructions, this state can be reached, and when it is reached, each philosopher will wait for another (the one to the right) to release a fork and there will be no progress. This will cause the system to reach a deadlock.

If we consider a solution in which a philosopher waits a certain amount of time before picking up a fork, we could still reach a state where all the philosophers pick up their left forks simultaneously and then finding out that the right forks are not available, putting their

forks down and then picking the left forks again simultaneously, even though the program seems to continue forever, it would actually make no progress since no one would be able to eat

Proposed Solution

By looking at the problem, it can easily be deduced that some sort of synchronization is required in order to solve it. The following is the proposed solution:

Solution Primitives

Semaphores

- An array of semaphores to allow processes to block and unblock after testing whether the forks are available or not
- Each philosopher was given a single semaphore from the semaphore array

Mutex

- A mutex lock was used to enter and exit the critical region

Threads

- An array of threads was used in the code to simulate parallelism
- Each thread was used to represent a single philosopher

State Array

- An array was used which was used to keep track of the state of the philosophers
- State was changed in the array whenever the behavior of the philosopher was changed

Proposed Mechanism

- First of all, an array of threads was created to introduce parallelism in the program where each thread represented a philosopher
- An array named “state” was created which kept track of the states of all the philosophers
- The central code of the philosophers revolved around thinking for a random amount of time. After this random time, the philosopher locked the mutex (i.e. enters the critical region) and the state of the philosopher is changed to HUNGRY and then the adjacent (LEFT and RIGHT) forks were tested.
- If the forks were acquired, the state of the philosopher is changed to EATING
- If the adjacent forks were not available. the philosopher was blocked using the sem_post(); function on the semaphore array index
- In order to acquire the forks, the philosopher looked at the adjacent philosophers and took the forks only if the LEFT and RIGHT philosophers were not eating

- The critical section of the code is where the forks were tested and the state array was updated. This critical section was protected via a mutex to allow synchronization and mutual exclusion
- After the philosopher stopped eating, the state was changed to THINKING again and the adjacent philosophers were tested if they can eat again, if they can, the blocked processes were also unblocked during this step

Performance

Pros

- The proposed solution is deadlock free
- In the proposed solution parallelism is achieved for a generic number of philosophers (only 5 visualized in the code).
- Mutual Exclusion is achieved
- There is no resource starvation in the proposed code
- The data remains consistent throughout as the shared variables are properly protected
- Proper synchronization is achieved

Cons

- If the random amount of wait time is high, CPU time could be wasted
- If due to some issue, a thread (philosopher) goes to sleep while it has obtained the lock, the program might not be able to move forward

Future Improvements

- Using forks to introduce multiple processes to represent philosophers
- Using multiple terminals to showcase philosophers

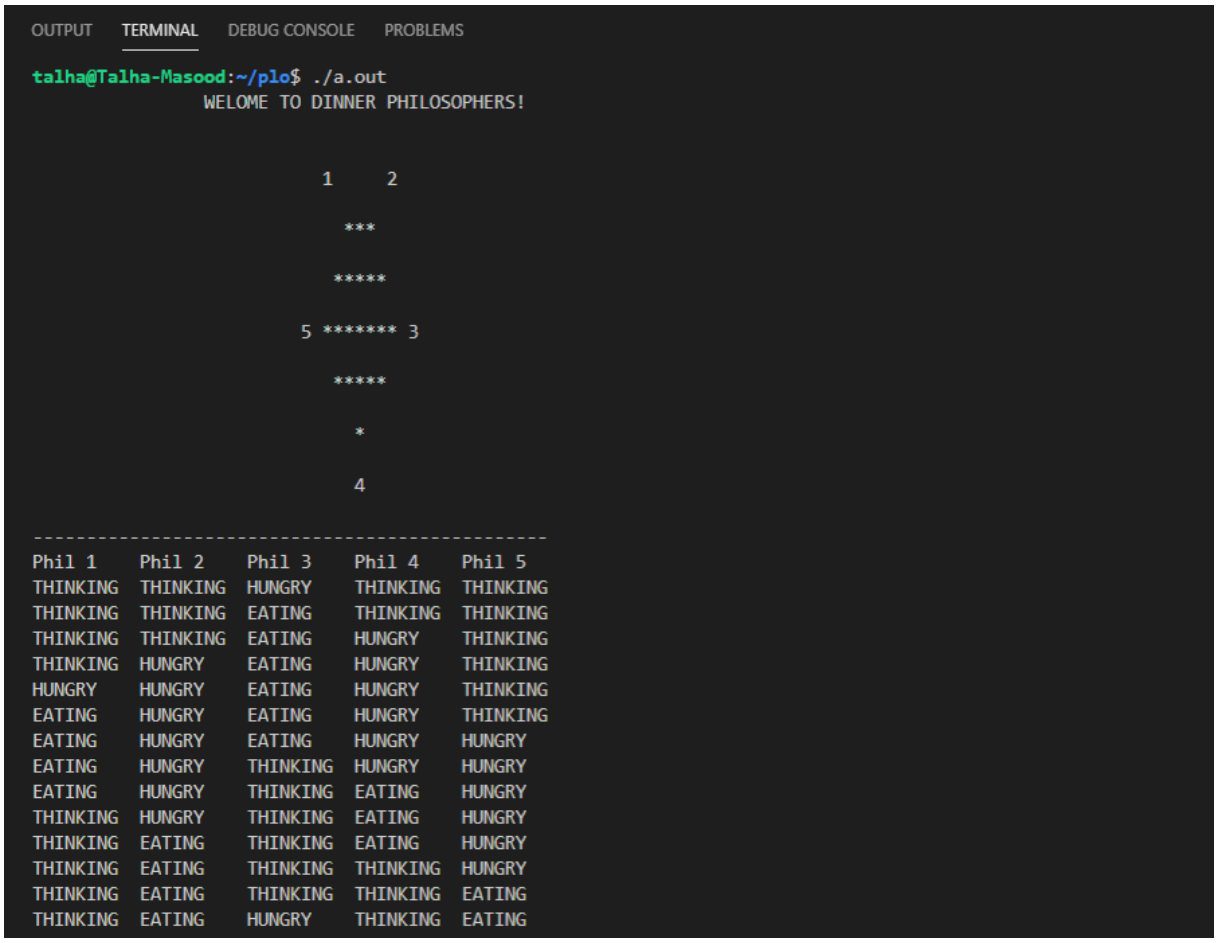
Alternative solutions

- Synchronization via disabling interrupts
- Using barriers
- Using spinlocks
- Using named pipes

Conclusion

In this task, a popular synchronization problem was successfully solved using built-in Linux primitives and the problem was presented visually through implementation in C.

Output


OUTPUT
TERMINAL
DEBUG CONSOLE
PROBLEMS
talha@Talha-Masood:~/plo$ ./a.out
WELOME TO DINNER PHILOSOPHERS!
1 2
***
*****
5 ******* 3
*****
*
4
Phil 1 Phil 2 Phil 3 Phil 4 Phil 5
THINKING THINKING HUNGRY THINKING THINKING
THINKING THINKING EATING THINKING THINKING
THINKING THINKING EATING HUNGRY THINKING
THINKING HUNGRY EATING HUNGRY THINKING
HUNGRY HUNGRY EATING HUNGRY THINKING
EATING HUNGRY EATING HUNGRY THINKING
EATING HUNGRY EATING HUNGRY HUNGRY
EATING HUNGRY THINKING HUNGRY HUNGRY
EATING HUNGRY THINKING EATING HUNGRY
THINKING HUNGRY THINKING EATING HUNGRY
THINKING EATING THINKING EATING HUNGRY
THINKING EATING THINKING THINKING HUNGRY
THINKING EATING THINKING THINKING EATING
THINKING EATING HUNGRY THINKING EATING