Project Report

Dining Philosophers Problem

Course: IT3070E - Operating Systems

Authors: Group 16

Nguyen Tong Minh - 20204885
Nguyen Cong Đat - 20200137
Ngo Thi Thu Huyen - 20200289
Nguyen Trung Hieu - 20204877

Advisor: TS. Do Quoc Huy

Academic semester: 2022.1

Abstract

The dining philosophers problem is a classical synchronization problem. Taken at face value, it is a pretty meaningless problem, but it is typical of many synchronization problems when allocating resources in operating systems.The problem was designed to illustrate the challenges of avoiding deadlock. In addition, resource starvation, mutual exclusion and livelock are other types of sequence and access problem. To keep these issues to a minimum, we implemented four solutions: Resource hierarchy solution, Arbitrator solution, Limiting the number of diners in the table and Chandy/Misra solution. These solutions works pretty well, the issues are effectively resolved or rarely occur in reality.

We would like to thank Mr. Do Quoc Huy - our Professor who dedicatedly taught us the subject Operating System. Moreover, he gave us the detail guidance that made us complete our project on the topic *Dining Philosophers Problem* easily. It was a great learning experience.

Contents

1 Introduction

In computer science, the dining philosophers problem is an example problem 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.

Problem demonstration Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork. Thus two forks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both forks. The problem is how to design a regimen (a concurrent algorithm) 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.

To sum up, the dining philosopher problem is set on some constraints; however, a few solutions of ours explained below may violate slightly one or more of these constraints, for flexibility and efficiency in synchronization:

- Philosophers are either thinking or eating. They do not talk to each other.
- Philosophers take the left fork then the right fork
- Philosophers can only fetch forks placed between them and their neighbors.
- Philosophers cannot take their neighbors' forks away while they are eating.
- Hopefully no philosophers should starve to death (i.e. wait over a certain amount of time before she acquires both forks).

To cover the problem in general details, we define that each philosopher can reach three major states, in which the state "hungry" is added to represent the philosophers who are waiting for any forks.

- Hungry: When a philosopher p_i wants to eat, if p_i doesn't have the left fork or the right fork, p_i will send a request to the neighbor to have the fork.
- Eating: When a philosopher p_i has both the left and right fork, p_i start eating.
- Thinking: After eating the philosopher goes to the thinking state before the hungry state again.

Real situations 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 a proposal in which each philosopher is instructed to behave as follow:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- put the left fork down;
- put the right fork down;
- repeat from the beginning.

However, they each will think for an undetermined amount of time and may end up holding a left fork thinking, staring at the right side of the plate, unable to eat because there is no right fork, until they starve.

Therefore, a valid process synchronization solution has to handle such deadlock cases as well as tackling other sequence and access problems, starting with the three common ones:

- Mutual Exclusion: If process P is executing in its critical section, then no other processes can be executing in their critical sections.
- Progress: If critical resource still able to serve and there are process want to be executed in critical section then this process can use critical resource.
- Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

2 Methodology

2.1 Resource Hierarchy Solution

Then, we introduce resource hierarchy solution, which is a solution proposed originally by Dijkstra.

- Index the resources (forks) from 1 to 5
- Each of philosopher will always pick up the lower numbered fork first, and then the higher numbered fork, from the two forks they use.
- The order each philosopher puts down the left or right fork first does not matter

Therefore, if four of the five philosophers simultaneously pick up their lower numbered fork, only the highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks. This design will guarantee deadlock-free.

However, resource hierarchy approach remains some significant drawbacks that can be seen throughout procedure.

1. One major limitation to this solution is that it requires the list of resources be completely known in advance. If a unit of work finds out it needs a resource numbered 2 while holding resource 4 and 5, it would have to release the two resources first, acquire 2, and re-acquire 4 and 5. It could cause troubles in efficiency.
2. The resource hierarchy solution is not fair. If philosopher 1 is slow to take a fork, and if philosopher 2 is quick to think and pick its forks back up, then philosopher 1 will never get to pick up both forks. A fair solution must guarantee that each philosopher will eventually eat, no matter how slowly that philosopher moves relative to the others.

2.2 Arbitrator Solution

Arbitrator approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. Along with the external entity, it provides a simple strategy to solve the Dining Philosopher problem:

- Every philosopher must request their (shared) forks from a waiter, who may refuse the request at first in order to avoid a deadlock (left fork first, for convenience).

- Only the unit that has the permission can access, then request shared resources.

- Release a resource does not need permissions from the arbitrator.

This solution is argued for deadlock avoidance: Since only one philosopher has the permission to request forks at once, then that philosopher must receive both of his two requests (for two forks) and return back the permission to the waiter so that the others can request resources. Therefore, the "circular wait" required for deadlock cannot occur. Besides, no starvation exists in such a scenario.

On the other hand, a new central entity (the waiter) is introduced, which would require additional resources. Also, this approach can result in reduced parallelism: if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available. The waiter hence damages the scalability of the whole system, becoming a bottleneck if the number of processors is large. In other words, the arbitrator solution holds the progress issue.

2.3 Limiting the Number of Diners Solution

We introduce the solution: Limiting the Numbers of Diners - an easy way to avoid deadlock. This solution presented by William Stallings is to allow a maximum of n-1 philosophers to sit down at any time. That means maximum number of philosophers on the table should not be more than four. By carrying out all the instructions, one of the philosophers will finally own the fork and prevent any deadlock.

Although this solution can avoid deadlock, it can lead to some issues. If all of four philosophers request the left fork, then the one who is at the left of the person waiting to sit can request the right fork and start eating. However, there exists one person who can eat at the same time as the other, but this is not possible because the left forks are all being requested. This leads to waiting for each other, prolonging the dinner time. Moreover, this solution can not ensure that the one waiting to sit can sit or eat in the future, so it may lead to starvation problem.

2.4 Chandy/Misra Solution

Finally, we introduce Chandy - Misra solution This is a solution proposed by K.Chandy and J.Misra [1] that can work with generalized dining philosophers problem, where the number of philosophers is arbitrary (n philosophers). The solution is decribed below.

We numbered n fork from 1 to n. Initially each fork goes to the philosopher with lower ID, so philosopher p_1 will have 2 forks, philosophers from p_2 to p_{n-1} get the right fork, and philosopher p_n has no fork. At the beginning each fork is marked *dirty*.

There are also specific mechanisms regarding the forks:

- After eating, the forks become *dirty*.

- When philosopher p_i receives a request for a fork, it depends on the state of the fork state. If the fork is *dirty*, then the philosopher will clean the fork, switch it's state to *clean*. Then, the philosopher will pass the fork to the neighbor. If the fork is *clean*, the request is hold off until the philosopher finish eating and the fork becomes *dirty*, then we clean the fork and give it to the neighbor. Note that the fork is *clean* when the philosopher is hungry or eating, so in that case we must wait after the philosopher finish eating to have the fork.

This solution ensures that no philosopher will starve: Any philosopher requests for a *dirty*

fork will have it and keeps the fork *clean*, so that philosopher can hold it until eating. If the philosopher requests a *clean* fork, that request is hold off until the neighbor finish eating, so the philosopher will eventually has the fork and eats, but it might be a long time waiting.

Chandy and Misra's proposal also can avoid deadlock, if it has a good initial state. The proof is detailed in [1] using the precedence graph. The algorithm initializes the philosopher p_1 to hold both the left and right forks, so the system won't go into deadlocks.

Nonetheless, this design violates the rule of 'philosophers do not talk to each other'.

3 Implementation

3.1 Backend

Studying the dining problem, we observe the generality that is shared across different synchronization implementations. Hence, we decide to write a set of modules, according to [3], which are *table.py*, *philosophers.py*, *forks.py*, to model the generality of the problem and, override specific setups to simulate the behaviours of the solutions above.

table.py This module aims to setup a initialization schema for the problem on a synchronization method. Any simulations can only be started by having required resources of the statement, which are forks and philosophers. Therefore, we create two methods $\_invite\_philosophers$ and $\_serve\_forks$ in order to initialize forks and philosophers with corresponding forks being assigned to them. However, based on each solution, there will be a distinct way to create those resources; therefore, we leave the two methods as abstract type and force solution builders to reimplement them.

After all resources are prepared, a simulation are ready to run. At this time, the philosophers, which are processes, will be started with an event handler tracking their states for the sake of animation. Since all simulations share this step, we practice it into method $start\_dining$ so solution implementations do not have to rewrite such a piece of code.

One to notice is that a process has its own memory and other processes cannot access it without a defined mechanism. Therefore, a multiprocessing manager is used to control and share the data (e.g, forks, *mutex*,...) between the processes (philosophers).

philosopher.py This module aims to practice the behaviours of philosophers from the problem statement. Intuitively, we consider and design them as a process. Since every philosopher has a loop procedure in which he thinks, then eats until he is full and leave (a process finishes), the method *run* is built-ready describing this pattern. Furthermore, a philosopher either eats or thinks, so we pack these two unique actions into method *eat* and *think*. Both thinking and eating are simulated by putting the process into sleep in an amount of time. Thinking can be the same for philosophers across implementations, but eating requires synchronization and then each solution will be left to practice its own schema here, after inheriting.

forks.py This module aims to implement the forks as stated. All methods we intend to use above consider the forks as semaphores; hence, we inherit class *Semaphore* from threading library of Python and sign them to the proxy of multiprocessing manager, which helps the manager possibly initialize objects of the forks and manage them throughout memories of multiple processes. A fork is assigned with an *id* to pair with its neighbor philosophers.

3.1.1 Resource Hierarchy Solution

This solution can be thought intuitively of as having one philosopher who takes his right fork first then left fork while the others take left fork then right fork. For simplicity, we also label each philosopher as others solutions, hence, the last philosopher (the philosopher with highest id) will take the right fork first (fork 1) then left fork (fork 5) if they are available.

Algorithm 1 Resource hierachy [2]

semaphore $array[0..4] fork \leftarrow [1, 1, 1, 1, 1]$
  loop forever
    p1 : think
    p2 : wait($mutex$)
    p3 : if philosopher.id = 5
    p4 : wait($fork[(i + 1)\%5]$)
    p5 : wait($fork[i\%5]$)
    p6 : else
    p7 : wait($fork[i\%5]$)
    p8 : wait($fork[(i + 1)\%5]$)
    p9 : end if
    p10: signal($mutex$)
    p11: eat
    p12: signal($fork[i\%5]$)
    p13: signal($fork[(i + 1)\%5]$)

For this solution, we implement two files: *philosophers.py* and *table.py*.

table.py We implement the *table* as much like others solution. Two inherited methods *_serve_forks* and *_invite_philosophers* are implement in a very simple way, almost like the arbitrator solution except we don't need a waiter semaphore.

philosophers.py We implement the class *HierachyPhilosopher* that inherits from *Philosopher*.

Our implementation for this file contains:

- One extra attribute *n_philosopher* to store number of philosophers at the table
- Methods
  - $left\_fork\_first$: return **True** unless the philosopher is the last philosopher
  - *eat*: check if the philosopher is the last philosopher, if $True$, request the right fork, if not available, the philosopher wait, else continue request the left fork, if available, goes to eating state, after finish, release forks, order is not important. Else if $False$, request the right fork then the left fork, the remains is same as mentioned in the above case.

3.1.2 Arbitrator Solution

As known for this method, in order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time until the philosopher has picked up both of their forks. To describe such a mechanism, we implement the waiter as a semaphore *mutex* with capacity 1. Then whenever a philosopher want to request a fork, he has to wait for the *mutex* first.

The implementation starts by inheriting the base backend of philosophers and table. Since the solution do not require the fork to have any specific behaviours, compared to original problems, then we just use the fork that has been implemented on the parent modules. After that, we override the necessary methods to initialize the dining scenario in *table.py* and setup the process synchronization schema in *philosophers.py*.

Algorithm 2 Arbitrator [2]

semaphore $array[0..4] fork \leftarrow [1,1,1,1,1]$
  semaphore $mutex \leftarrow 1$
  loop forever
    p1 : think
    p2 : wait($mutex$)
    p3 : wait($fork[i]$)
    p4 : wait($fork[i+1]$)
    p5 : signal($mutex$)
    p6 : eat
    p7 : signal($fork[i]$)
    p8 : signal($fork[i+1]$)

table.py For the table, where the initialization of the problem is executed, we have to write a specific one based on the requirements of arbitrator solution. With method $\_serve\_forks$, we initialize forks, which are the critical resources, as process-shared semaphores managed by a multiprocessing manager. Those forks will then be passed to corresponding philosophers, which are now processes. On the other hand, the method $\_invite\_philosophers$ is responsible for preparing processes of philosopher. Here, we pass the corresponding left fork (determined by $id\_$ of the forks and philosophers) and right fork as a list (the former being the first element) to each philosopher created. Besides, for the arbitrator solution, we get an another $mutex$ to handle. The practice is to create it under the track of manager and share it across philosophers. At last, a list of run-ready philosophers are returned and start later.

philosophers.py For the philosopher, this is where we add the practice of synchronization solution. We follow the pseudo-code for arbitrator algorithm 2 and write the script for the overridden method *eat* to synchronize philosophers. After requesting forks, a philosopher acts up "eating" by calling method *eat*() of the parent (the one without synchronization).

3.1.3 Limit the Number of Diners

We implemented this solution following by the pseudo code [2] below:

Algorithm 3 Limit number of Diners

semaphore $array[0..4] fork \leftarrow [1,1,1,1,1]$
  semaphore $wait\_to\_sit \leftarrow 4$
  loop forever
    p1 : think
    p2 : wait($wait\_to\_sit$)
    p3 : wait($fork[i]$)
    p4 : wait($fork[i+1]$)
    p5 : eat
    p6 : signal($fork[i]$)
    p7 : signal($fork[i+1]$)
    p8 : signal($wait\_to\_sit$)

Explanation for this code, we limit the number of people requesting fork to 4, that means there exists one person who is not allowed to request fork.

For the detail of implementation, we implement two files: *philosophers.py* and *table.py*.

table.py The *table* is implemented following by the other algorithm of table implementation. The only class is *LimitTable*, which inherits from the abstract class *table*. Two methods $\_serve\_forks$ and $\_invite\_philosophers$ are similar to other implementations of above algorithm. However, there is a different point in method $\_invite\_philosophers$, we add a semaphore *room* to limit the diners requesting the fork to 4. After that we pass this variable to class *LimitPhilosopher* which belongs to *philosophers.py*

philosophers.py We implement a single class *LimitPhilosopher* that inherits from *Philosopher*. Our implementation for this file is identical to above pseudo code: before *eat*, the philosopher have to wait the semaphore

wait_to_sit, then request the fork by call request methods on the fork, and after eating will signal the fork that we finish eating.

3.1.4 Chandy/Misra Solution

In the Chandy - Misra solution implementations, we implements three files: *philosophers.py*, *forsk.py* and *table.py*.

table.py The *table* implementations is very much similar to other algorithm table implementations with a single class *CMTable*

- *CMTable* inherits from the abstract class *Table* so it uses the method *start_dining* from the abstract class.
- method *_serve_forks* has the same implementations with the above algorithm.
- method *_invite_philosophers* is the same with the above algorithm with one adjustment: it also gives the forks to philosopher according to our algorithm initialization described above.

philosophers.py Our *philosophers* implementations is a single class *CMPhilosopher* inherits from *Philosopher* with one adjustment only: before eat, the philosopher will have to request the fork by call request methods on the fork, and after eating will signal the fork that we finish eating.

forks.py This is the largest change in the algorithm implementation. Instead of being a simple Semaphore, we implement the *CMFork* class being a complex object with different attributes and methods. The *CMFork* object is shared among philosophers, which mean shared among processes, and being used for philosopher to make request to each others.

The *CMFork* object has the following attributes and methods:

- Attributes
  - *id_*: the ID of the fork
  - *state*: the fork state
  - *owner_id*: the ID of the philosopher is holding the fork
  - *lock*: a mutex lock to make sure the fork is being accessed by a philosopher at a time.
  - *condition_lock*: a condition variable to wait and signal philosopher waiting for the forks.
- Methods
  - *request* (philosopher_id: *CMPhilosopher*): call to request this fork. The method accept a parameter: philosopher_id which is the ID of the philosopher who is asking for this fork. Three case can happens:
    - The philosopher requesting is the owner of this fork. In this case the mutex lock is called, we change the fork state to *clean* to know that the philosopher is hungry and ready to eat.
    - The philosopher requesting is not the owner and the fork is dirty. In this case we call the mutex lock, change the fork to *clean* and pass it to the request philosopher by changing the fork *owner*
    - The The philosopher requesting is not the owner and the fork is clean. In this case, we call the condition variable to wait. This wait will finish until the current owner of the fork finish eating and issue a *notify* to the condition variable. After that, we proceed as the above case when the fork is *dirty*
  - *finish*: This method is called when the owner of the fork finish eating. The mutex lock is called, we mark the fork as *dirty*,

and then call notify the condition variable, to make the other philosopher requested for the fork know that the fork is finish using and can be give to that philosopher.

3.2 GUI

To animate the problem, we design a simple UI (*gui.py* for UI and *events.py* for event handling) to describe the state of each philosopher over a problem simulation. The graphics of program is shown in the figure 1.

On the right of the UI, the gray buttons are where to select the desired methods, and the red button is responsible for starting the program with the selected synchronization method. Under the panel, there is a notation about colors representing for the states of philosophers during simulation with red being hungry, blue being thinking and green being eating. By changing colors based on states, we hopefully can animate clearly the simulation of the dining philosopher problem, as shown in figure 2.

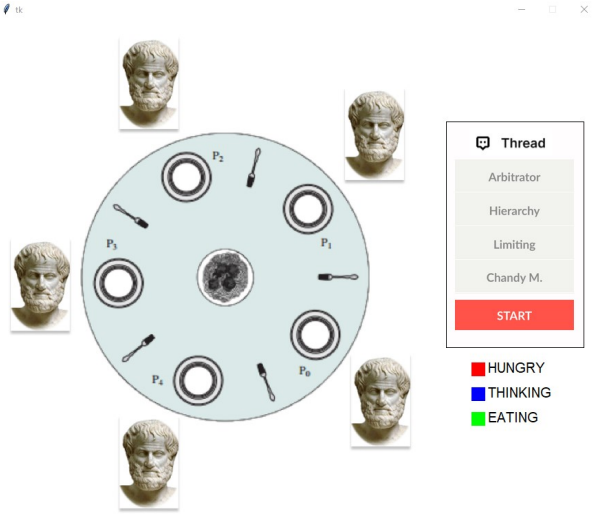

Figure 1: The UI of the simulation.

After a philosopher is full and ends itself, that philosopher will return to the initial state, that is with no border.

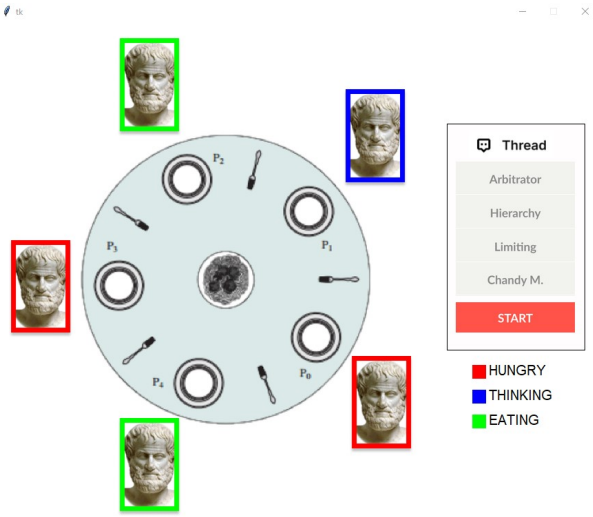

Figure 2: The UI animation during simulation.

4 Conclusion

In this project, we have investigated dining philosopher problem, a classic problem in process synchronization. Specifically, we studied 4 different solutions to the problem: arbitrator solution, resource hierarchy solution, limiting number of diners solution and Chandy/Misra solution. Each solutions has it's own advantages and disadvantages, show multiple approaches toward synchronization. The 4 algorithms were implemented in Python programming language to illustrated how these 4 solutions work, not just theory. The results show that all 4 solutions has no deadlocks or starvation, which prove the correctness of the algorithm empirically. Finally, we had also built a GUI application to demonstrate the dining philosopher problem and its solutions in an intuitive way.

References

[1] J Chandy K.M.; Misra. "The drinking philosophers problem". In: *ACM Transactions on Programming Languages and Systems* (1984).

[2] *Dining philosophers problem.* `https://www.geeksforgeeks.org/dining-philosophers-problem/`.

[3] Lievi Silva. *lievi/dining_philosophers.* 2021. URL: `https://github.com/lievi/dining_philosophers`.