Dining Philosophers: A Synchronization Problem

CS347 Operating Systems

Rathour Param Jitendrakumar
190070049

Department of Electrical Engineering
Indian Institute of Technology Bombay

Spring 2021-22

> *When you go to sleep make sure there is someone to wake you up.*
> *(Prof. Mythili Vutukuru)*

Outline

Problem Formulation

The Setup

- N philosophers denoted by P_i, $i \in [N] \triangleq \{0, ..., N-1\}$ around a circular table.
- The table contains
  - N plates - a plate in front of each philosopher denoted by p_i, $i \in [N]$.
  - N forks - in between two consecutive plates denoted by f_i, $i \in [N]$.
  - a huge bowl of spaghetti in the centre of table.
- P_i has p_i in their front and $f_i, f_{(i+1)\%N}$ to their right and left respectively.

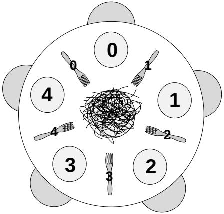

Figure: Example[1] when $N = 5$

---

[1] Downey Allen B. The Little Book of Semaphores

Problem Formulation

The Philosopher

- A philosopher can start eating only after picking up the forks on their left and right.
- Till they start eating, they will be ‘thinking’.
- Generic Behaviour of a philosopher:

```
while (True){
    think()
    pick_up_forks()
    eat()
    put_down_forks()
}
```

Problem Formulation

The Problem

- Write `pick_up_forks` and `put_down_forks` satisfying the following
- Constraints
  - A fork can be used by only one philosopher at any instant.
  - No deadlock should occur.
  - No philosopher should starve forever.
  - At least two philosophers can eat at same time.
- Assumptions
  - `think` and `eat` are known (possibly unique for each philosophers).
  - `eat` has to terminate.
- The intuition here is that the philosophers represent the threads and forks represent the resources needed for these threads to proceed.
- A complicated problem as a thread can possibly context switch anytime during its execution

Problem Formulation

The Notation

- The right and left philosopher for i^{th} philosopher are given by:

```
right_p(i) = (i-1)\%N
left_p(i) = (i+1)\%N
```

- The right and left fork for i^{th} philosopher are given by:

```
right_f(i) = i
left_f(i) = (i+1)\%N
```

- We may refer to philosophers as threads
- A philosopher P_0 successfully completes/finishes when it goes back to thinking.
- A scheduled thread is active when it has run at least once.

Semaphores

Introduction

- Semaphores are used to achieve synchronization between threads.
- A semaphore is essentially a variable with an underlying counter
- The counter value can't be accessed once it is initialised with a suitable value.
- For a semaphore variable `s`,
  - When a thread calls `down(s)`, the counter is decremented and the thread is blocked if the counter value becomes negative.
  - When a thread calls `up(s)`, the counter is incremented and any one of blocked threads is woken up ('ready to run' again).

Semaphores – Focusing on Forks

Incorrect Solution

- Create N semaphore variables, one for each fork denoted by $s_i = 1, i \in [N]$.
- Pseudocode:

```
function pick_up_forks(philosopher i){
    down(s_{right_f(i)})
    down(s_{left_f(i)})
}
```

```
function put_down_forks(philosopher i){
    up(s_{left_f(i)})
    up(s_{right_f(i)})
}
```

Semaphores – Incorrect Solution

Example – Deadlock

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2 (another example would be P_0, P_2, P_1.)
- P_0 will get the right fork f_0 by calling `down(s_{right_f(0)}) = down(s_0)` which will make $s_0 = 0$. Now, suppose P_0 gets context switched then P_1 begins.
- P_1 will get the right fork f_1 by calling `down(s_{right_f(1)}) = down(s_1)` which will make $s_1 = 0$. Now, suppose P_1 gets context switched then assuming P_2 begins,
- P_2 will get the right fork f_2 by calling `down(s_{right_f(2)}) = down(s_2)` which will make $s_2 = 0$. Now, every P_i will have a single fork f_i.
- Now, if any thread P_i executes `down(s_{left_f(i)})`, then for $f_{\text{left}(i)}$, $s_{\text{left}(i)} = -1$. Hence, that thread will be sent to sleep.
- Each thread will try to access their 'left fork' and will be sent to eternal sleep. A *deadlock*!

Semaphores

Incorrect Solution – Why Deadlock?

Proof.

- Intuitively, suppose each philosopher simultaneously picks up the fork to their right, then all forks are occupied. There are no ‘available forks’ to any philosopher’s left.
- Formally, if each thread gets context switched just after executing `down(s_{right_f(i)})`, then this will result in $s_i = 0, i \in [N]$.
- Now, if any thread P_i gets scheduled and executes `down(s_{left_f(i)})`, then for that fork’s semaphore $s_{\text{left}(i)} = -1$. Hence, that thread will be sent to sleep.
- Similarly, each thread will try to access their ‘left fork‘ and will be sent to sleep.
- To awake them, some thread must give signal which is not possible as all threads are sleeping. A deadlock!
- Note that the above case is possible for any scheduling of threads as we haven’t made any assumption on scheduling in the proof.

$\square$

Semaphores

Correct Solution

- In addition to the N semaphore variables, one for each fork $s_i = 1, i \in [N]$,
- create another semaphore variable called $max = N - 1$, denoting the maximum number of philosophers allowed to eat at any instant.
- Revised pseudocode:

```
function pick_up_forks(philosopher i){
    down(max)
    down(s_{right_f(i)})
    down(s_{left_f(i)})
}
```

```
function put_down_forks(philosopher i){
    up(s_{left_f(i)})
    up(s_{right_f(i)})
    up(max)
}
```

Semaphores – Correct Solution

Example 1

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2
- Let's assume that no context switches can occur during the execution of `pick_up_forks` and `put_down_forks` (unless it sleeps due to semaphore).
- P_0 will get both it's 'right and left fork' (f_0 and f_1) and P_0 will start eating.
- Now, one of the two things can happen:
  - P_0 successfully completes everything
  - P_0 gets context switched out before it can put down its both forks.

Semaphores – Correct Solution

Example 1

Example (continued)

- In the first case, P_1 will get both it's 'right and left fork' and it will start eating using f_1 and f_2, whereas in the second case P_1 will only be able to get one fork (f_2) and will go to sleep.
- P_0 will complete eating and thus its job at sometime due to assumption that eat has to terminate. So, forks f_0 and f_1 will be available later for other threads.
- Even in the 2nd case after completion of P_0, P_1 will wake up and start eating using f_1 and f_2.
- Hence, P_2 will get a fork f_0 and will go to sleep waiting for P_1 to complete.
- Once P_1 completes eventually, P_2 will be woken up and it will also get completed.

Semaphores – Correct Solution

Example 2

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2
- If any thread is able to pick up both forks, it will eventually finish and the analysis will be similar to Example 1.

Semaphores – Correct Solution

Example 2

Example (continued)

- So let's try the worst case, when each thread gets context switched just after picking up right fork, now the *max* semaphore comes into picture.
- P_0 and P_1 will get their right fork f_0, f_1 respectively.
- Only two of three threads can be active at a time. As both P_0 and P_1 are yet not completed, When P_2 tries to pick up forks, it is sent to sleep as *max* becomes -1.
- So, P_0 and P_1, will be able to pick up atleast one fork i.e. the uncommon forks.
- Also, one of them will surely pick up the common fork between them f_1. So that thread will start eating.
- Eventually, it will finish and now the other thread will pick up f_1 and start eating .
- Now, P_2 will be woken up and once the older thread finishes, P_2 will start eating.

Semaphores – Correct Solution

Proof.

- WLOG say initially P_i is scheduled earlier than P_j for all $j > i$ where, $i, j \in [N]$.
- If we are able to show that any thread is able to pick up both forks, then it will eventually finish and now we will be left with same forks but one less thread.
- Due to *max*, at most $N - 1$ threads have begun but there are N forks. So, by pigeonhole principle one thread (say P_i) will get two forks which will be done by `pick_up_forks`.
- Semaphore variables, ensure that multiple threads can't access same fork at a time.

☐

Semaphores – Correct Solution

Proof (continued).

- So eventually P_i will start eating and finish eating.
- Then, P_i will execute `put_down_forks` and will free its forks for the neighbours.
- So eventually P_i will finish and we will be left with one less thread.
- If one thread was able to finish when a total of k threads were active, then one thread can definitely finish when a total of $k - 1$ threads were active as we can always add a thread which does nothing.
- So, we have recursively shown that all threads will finish.

□

Condition Variables

Introduction

- Condition variables are also used to achieve synchronization between threads.
- They communicate between threads when certain conditions becomes true.
- For a condition variable `cv`,
  - When a thread calls `wait(cv)`, it is added to a list of waiting threads for `cv` and is blocked. This list is maintained for every condition variable.
  - When a thread calls `signal(cv)`, any one of blocked threads is woken up (‘ready to run’ again). There is no immediate context switch, it will be scheduled later.

Condition Variables – Focusing on Philosophers

Incorrect Solution

- Create N condition variables, one for each philosopher denoted by $c_i, i \in [N]$.
- Create N state variables, one for each philosopher denoted by $x_i = T$, where $i \in [N]$ and $x_i \in \{E, T\}$ denoting whether the philosopher is Eating or Thinking.
- Pseudocode:

```
function pick_up_forks(philosopher i){
    while (s_{right_p(i)} = E OR s_{left_p(i)} = E)
        wait(c_i)
    s_i = E
}
```

```
function put_down_forks(philosopher i){
    s_i = T
    if (s_{right_p(right_p(i)} = T)
        signal(c_{right_p(i)}
    if (s_{left_p(left_p(i)} = T)
        signal(c_{left_p(i)}
}
```

Condition Variables – Incorrect Solution

Example 1 – Deadlock

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2
- For P_0, both its 'right and left neighbour' (P_2 and P_1) are thinking. So, P_0 exits while loop and changes its state to eating.
- Now when P_1 `pick_up_forks` is executed, the while loop condition fails as P_0 is still eating.
- Suppose, P_1 is context switched just before it can go to sleep. Also, same happens with P_2.
- Now when P_0 completes eating it will execute `put_down_forks`, which changes its state back to thinking and sends the signal to wake up P_1 and P_2 using c_1 and c_2 respectively.
- Now, when P_1 and P_2 will come back they will execute the wait statement ignorant of the fact that P_0 is already completed
- So both P_1 and P_2, will go to eternal sleep, called a *missed wakeup* problem and a deadlock!

Condition Variables – Incorrect Solution

Example 2 – Race Condition

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2
- For P_0, both its ‘right and left philosopher’ (P_2 and P_1) are thinking. So, P_0 exits while loop.
- Now, suppose P_0 gets context switched before P_0 changes its state to eating.
- For P_1, both its ‘right and left neighbour’ (P_0 and P_2) are thinking. So, P_1 exits while loop and changes its state to eating.
- Now even if P_2 is scheduled next, it will remain in while loop and go to sleep as P_1 is eating. So P_0 comes back and changes its state to eating.
- The above will imply two neighbouring philosophers are eating simultaneously by using a common fork which should not happen, a *race condition*!

Condition Variables

Correct Solution

- Pseudocode:

```
function pick_up_forks(philosopher i){
    lock(mutex)
    while (s_{right_p(i)} = E OR s_{left_p(i)} = E)
        wait(c_i, mutex)
    s_i = E
    unlock(mutex)
}
```

```
function put_down_forks(philosopher i){
    lock(mutex)
    s_i = T
    if (s_{right_p(right_p(i)} = T)
        signal(c_{right_p(i)}
    if (s_{left_p(left_p(i)} = T)
        signal(c_{left_p(i)}
    unlock(mutex)
}
```

Condition Variables – Correct Solution

Example 1

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2
- P_0 first acquires the lock, then as both its ‘right and left neighbour’ (P_2 and P_1) are thinking, P_0 exits while loop and changes its state to eating and releases the lock.
- Now when P_1 `pick_up_forks` is executed it acquires the lock but the while loop condition fails as P_0 is still eating.
- Suppose, P_1 is context switched just before it can go to sleep. Then as P_1 has not yet released the lock, P_0, P_2 will keep waiting for `put_down_forks` and `pick_up_forks` respectively.
- The lock will be released only when P_1 comes back and executes `wait`. The lock is released only after ensuring that list of waiting processes contains P_1. Then P_1 goes to sleep.
- If P_2 is scheduled earlier than P_0, then it will go to sleep eventually (directly or like in P_1).

Condition Variables – Correct Solution

Example 1

Example (continued)

- When P_0 completes eating it will execute `put_down_forks` and acquire the lock then change its state back to thinking and signal P_1 and P_2 using c_1 and c_2 respectively.
- Now, whoever among P_1 and P_2 will come back first (say P_1) will reacquire lock and will release the lock only after changing its state and then it can start eating.
- Even if there are more random context switches P_1 will start eating first as P_2 will have to wait till the P_1 state changes back to thinking. So, P_1 will eventually finish.
- Then the last thread will do the formality.

Condition Variables – Correct Solution

Example 2

Example

- Say for $N = 3$ case, the schedule is P_0, P_1, P_2
- P_0 executes `pick_up_forks` and first acquires the lock, as both its 'right and left philosopher' (P_2 and P_1) are thinking. So, P_0 exits while loop.
- Now, suppose P_0 gets context switched before P_0 changes its state to eating.
- As P_0 still hasn't released the lock, whichever thread is scheduled next (say P_1), it can't acquire the lock in `pick_up_forks` and will be put to sleep.
- If P_2 is scheduled earlier than P_0, then it will go to sleep eventually (directly or like in P_1).
- When P_0 comes back, it will change its state and release the lock and signal any one of neighbours (say P_1) to wake up. Also, P_0 can now start eating.

Condition Variables – Correct Solution

Example 2

Example (continued)

- Now, P_1 will acquire lock but it will be put to sleep as P_0's state is eating. The lock is released only after ensuring that list of waiting processes contains P_1. Then P_1 goes to sleep.
- When P_0 completes eating it will execute `put_down_forks` and acquire the lock then change its state back to thinking and signal P_1 and P_2 using c_1 and c_2 respectively.
- Now, whoever among P_1 and P_2 will come back first (say P_1) will reacquire lock and will release the lock only after changing its state and then it can start eating.
- Even if there are more random context switches P_1 will start eating first as P_2 will have to wait till the P_1 state changes back to thinking. So, P_1 will eventually finish.
- Then the last thread will do the formality.

Condition Variables – Correct Solution

The proof's structure is similar to the proof for Semaphores

Proof.

- WLOG say initially P_i is scheduled earlier than P_j for all $j > i$ where, $i, j \in [N]$.
- If we are able to show that any thread is able to pick up both forks, then it will eventually finish and now we will be left with same forks but one less thread.
- The condition variables are set up in such a way that either the thread will guaranteedly pick up both forks (say P_i) or it can't pick any.
- Now once P_i `pick_up_forks` starts execution, P_i will acquire a lock. So, even in the case of context switching its neighbours simply can't acquire lock in `pick_up_forks` until P_i completes `pick_up_forks`
- Intuitively, we can be sure that the state changes only when both forks are available. Hence, multiple threads can't access same fork at a time.

□

Condition Variables – Correct Solution

Proof (continued).

- So eventually P_i will start eating and finish eating.
- P_i will execute `put_down_forks` and change its state allowing its neighbours to get forks.
- If one thread was able to finish when a total of k threads were active, then one thread can definitely finish when a total of $k - 1$ threads were active since we can always add a thread which does nothing.
- So, we have recursively shown that all threads will finish.

□

References

Downey Allen B.
The Little Book of Semaphores (2nd Edition).
v2.1.5 edition, 2019.

IIT Bombay Mythili Vutukuru.
Concurrency: Slides and practice problems.
URL: `https://www.cse.iitb.ac.in/~mythili/os/`.