Operating Systems

Dining Philosophers and Sleeping Barber

Hongliang Liang, BUPT

Sep. 2023

We are going to discuss two problems

These are classic thread synchronization problems

They are examples to show how semaphores and monitors can be used to achieve synchronization —

- The Dining Philosophers Problem
- The Sleeping Barber Problem (not in textbook)

The Dining Philosophers

The Dining Philosophers Problem

- Five philosophers sit at a table
- One fork between two neighbouring philosophers
- Philosophers think, grab both forks, eat, put down both forks
- Models exclusive access to a limited number of resources (such as I/O devices)
- Each philosopher is modelled as a thread

```
while true do
  think()
  Pickup left fork
  Pickup right fork
  eat()
  Put down left fork
  Put down right fork
```

A solution

```
philosopher(int i)
  while true do
    think()
    pickup_forks(i)   //
    eat()
    putdown_forks(i)  //
```

```
pickup_forks(int i)
  pickup_fork(i)
  pickup_fork((i+1) % 5)

putdown_forks(int i)
  putdown_fork(i)
  putdown_fork((i+1) % 5)
```

A solution

```
philosopher(int i)
  while true do
    think()
    pickup_forks(i)   //
    eat()
    putdown_forks(i)  //
```

```
pickup_forks(int i)
  pickup_fork(i)
  pickup_fork((i+1) % 5)

putdown_forks(int i)
  putdown_fork(i)
  putdown_fork((i+1) % 5)
```

Correct?

The Problem

It may happen that all five philosophers take their left fork at the same time, and then try to take their right fork, which is taken by a neighbouring philosopher!

No one is able to progress — a **deadlock**

How do we solve this problem?

Intuition: taking the left and right forks needs to be made into one **atomic action**

Second Try: the Dining Philosophers Problem

```
semaphore mutex = 1 // binary semaphore
philosopher(int i)
  while true do
    think()
    mutex.down()  //
    pickup_forks(i)
    eat()
    putdown_forks(i)
    mutex.up()  //
```

The Problem Now

Only one philosopher can be eating at a given time

But we should be able to allow **two** philosophers eating at the same time!

Poor performance!

How do we solve this problem?

First intuition: define a smaller critical section by moving the binary semaphore operations into **pickup_forks()** and **putdown_forks()**

Now the new solution

```
philosopher(int i)
  while true do
    think()
    pickup_forks(i)  //
    eat()
    putdown_forks(i)  //

pickup_forks(int i)
    mutex.down() //
    pickup_fork(i)
    pickup_fork((i+1) % 5)
    mutex.up() //

putdown_forks(int i)
    mutex.down()  //
    putdown_fork(i)
    putdown_fork((i+1) % 5)
    mutex.up()  //
```

Now the new solution

```
philosopher(int i)
  while true do
    think()
    pickup_forks(i)   //
    eat()
    putdown_forks(i)  //

pickup_forks(int i)
    mutex.down() //
    pickup_fork(i)
    pickup_fork((i+1) % 5)
    mutex.up() //

putdown_forks(int i)
    mutex.down()  //
    putdown_fork(i)
    putdown_fork((i+1) % 5)
    mutex.up()  //
```

Correct?

Looks fine so far, but what about pickup_fork()?

The solution looks fine for now, but we haven't implemented **pickup_fork()** and **putdown_fork()** yet!

How do we implement them?

- We do not need to maintain any additional states to know if a fork is available
- Just look at the **status** of two adjacent philosophers

The status of two adjacent philosophers

They can be in one of the three states: eating, thinking, or hungry (waiting for forks to become available)

A philosopher may only eat if both of his neighbours are not eating

What if a philosopher tries to pickup a fork, but it is not available?

- It needs to wait for it to become available — **thread synchronization**
- His neighbour, once finished eating, will have to wake him up

First try: synchronization with semaphores

```
semaphore sem[5]= {5 of 0} // one sem per philosopher
int status[5] = {5 of THINKING}
pickup_forks(int i)
  mutex.down()  //
  status[i] = HUNGRY
  int left = (i+4) % 5
  int right = (i+1) % 5
  if status[left] == EATING or
     status[right] == EATING then
    sem[i].down()  //
  status[i] = EATING
  mutex.up()  //
```

First try: synchronization with semaphores

```
putdown_forks(int i)
  mutex.down()  //
  status[i] = THINKING
  int left = (i+4) % 5
  int right = (i+1) % 5
  if status[left] == HUNGRY then
    sem[left].up()  //
  if status[right] == HUNGRY then
    sem[right].up()  //
  mutex.up()  //
```

Problem with the first try

In pickup_forks(), if a philosopher **i** has failed to pick up both forks, it calls `sem[i].down()`, which **blocks itself, before** calling `mutex.up()` to leave the critical section

No other thread is able to enter the critical section — **deadlock**!

So how do we solve this problem?

How about this solution?

```
pickup_forks(int i)
  mutex.down()  //
  status[i] = HUNGRY
  int left = (i+4) % 5
  int right = (i+1) % 5
  if status[left] == EATING or
     status[right] == EATING then
    mutex.up()  //...
    sem[i].down()  //
    status[i] = EATING
  else
    status[i] = EATING
    mutex.up()  //
```

Still another problem

Philosopher 1 and 4 were both eating at this time

They finish eating at the same time

Philosopher 1 wakes up 2, and 4 wakes up 3, since both 2 and 3 are hungry at the time (2 waiting on sem[2], 3 on sem[3])

Both **sem[2].down()** and **sem[3].down()** are allowed to proceed!

How to solve the problem?

Can we solve the problem by changing **if** to **while** in pickup_forks()?

```
while status[left] == EATING or
      status[right] == EATING do
  mutex.up()  //
  sem[i].down()  //

status[i] = EATING
```

How to solve the problem?

Can we solve the problem by changing **if** to **while** in pickup_forks()?

```
while status[left] == EATING or
      status[right] == EATING do
  mutex.up()  //
  sem[i].down()  //

status[i] = EATING
```

No. We are testing **status[left]** and **status[right]** without acquiring mutual exclusion locks!

Correct implementation of pickup_forks()

```
pickup_forks(int i)
  mutex.down()
  status[i] = HUNGRY
  int left = (i+4) % 5, right = (i+1) % 5
  while status[left] == EATING or
        status[right] == EATING do
     mutex.up()   //
     sem[i].down()   //
     mutex.down()   //
  status[i] = EATING
  mutex.up()
```

Alternative solution: revise putdown_forks()

Alternatively, we can leave `pickup_forks()` as it was

Instead, we revise `putdown_forks()`

- When a philosopher finishes eating, it **only** wakes up a neighbouring philosopher if it is sure that its *other* neighbour is not eating!
- If it does wake up a neighbour, it sets its status to EATING

Alternative solution: revise putdown_forks()

```
pickup_forks(int i)
  mutex.down()
  status[i] = HUNGRY
  int left = (i+4) % 5, right = (i+1) % 5

  if status[left] == EATING or
     status[right] == EATING then
    mutex.up()
    sem[i].down()
  else
    status[i] = EATING
    mutex.up()
```

Alternative solution: revise putdown_forks()

```
putdown_forks(int i)
  mutex.down()  //
  status[i] = THINKING
  int left = (i+4) % 5, right = (i+1) % 5
  if status[left] == HUNGRY and
     status[(left+4) % 5] != EATING then //...
    status[left] = EATING
    sem[left].up()  //
  if status[right] == HUNGRY and
     status[(right+1) % 5] != EATING then //...
    status[right] = EATING
    sem[right].up()  //
  mutex.up()  //
```

Now you see why we need monitors!

Using semaphores is a bit too **tricky**, even when solving a simple synchronization problem

In later Lab, you are asked to implement the Dining Philosophers problem using monitors and condition variables

- The monitor implementation in BLITZ follows MESA semantics
- Keep this in mind when designing your solution

But semaphores are more **powerful** primitives, it allows us to design a simpler solution

Revisiting our initial solution

```
philosopher(int i)
  while true do
    think()
    pickup_forks(i)  //
    eat()
    putdown_forks(i)  //

pickup_forks(int i)
  pickup_fork(i)
  pickup_fork((i+1) % 5)

putdown_forks(int i)
  putdown_fork(i)
  putdown_fork((i+1) % 5)
```

Towards designing a simpler solution

```
semaphore forks[5]= {5 of 1} // instead of philosophers!

pickup_fork(int i)
  forks[i].down()

putdown_fork(int i)
  forks[i].up()
```

Towards designing a simpler solution

```
semaphore forks[5]= {5 of 1} // instead of philosophers!

pickup_fork(int i)
  forks[i].down()

putdown_fork(int i)
  forks[i].up()
```

But what about the deadlock?

Making the solution deadlock-free

Making the solution deadlock-free

```
pickup_forks(int i)
  if i == 4 then
    pickup_fork((i+1) % 5)
    pickup_fork(i)
  else
    pickup_fork(i)
    pickup_fork((i+1) % 5)

putdown_forks(int i)
  putdown_fork(i)
  putdown_fork((i+1) % 5)
```

The Sleeping Barber Problem

The Sleeping Barber Problem

The Sleeping Barber Problem

The sleeping barber —

- While there are customers sitting in a waiting chair, move one customer to the barber chair, and start the haircut
- When done, move to the next customer
- If there are no customers, go to sleep

The customers —

- If the barber is asleep, wake him up for a haircut
- If someone is getting a haircut, wait for the barber to become free by sitting in a waiting chair
- If all waiting chairs are occupied, leave the barber shop

Basic ideas towards a solution

Model the barber and customers as **threads**

Model the number of waiting customers as a **semaphore** (0 or more)

- `semaphore customers = 0 // # of waiting customers`
- Since there is no way to read the current value of this semaphore, we also need an integer variable, say `occupied_chairs`, to keep track of the number of waiting customers
- To protect access to occupied_chairs, we need a mutual exclusion lock — `lock access_lock = UNLOCKED`

Model the state of the barber as a **semaphore**

- `semaphore barber = 0 // Is the barber ready to start?`

Solving the Sleeping Barber Problem

```
semaphore barber = 0 // Is the barber ready to start?
semaphore customers = 0 // number of waiting customers
lock access_lock = UNLOCKED
int occupied_chairs = 0
barber()
  while true do
    customers.down() // wait for (or get) a customer
    acquire(access_lock) //
    occupied_chairs = occupied_chairs - 1
    release(access_lock) //
    barber.up() // the barber is now ready to start
    cut_hair()
```

Solving the Sleeping Barber Problem

```
customer()
  acquire(access_lock) //
  if (occupied_chairs < N) then
    occupied_chairs = occupied_chairs + 1
    release(access_lock) //
    customers.up() // one more customer has taken a chair
    barber.down() // waiting for barber to get ready
    get_haircut()
  else
    release(access_lock) // leave the barber shop
```

Subtleties in Lab

In later Lab, you are asked to print the state of the customers —

- Enter, Sit in a waiting chair, Begin haircut, Finish haircut, Leave

For a successful haircut, from the desired output, the customer and **the barber** go through 7 states sequentially —

- Enter, Sit in a waiting chair, **Start**, Begin haircut, Finish haircut, **End**, Leave

Correctly producing this sequence requires —

- The barber prints **Start** before the customer prints **B**
- The customer prints **F** before the barber prints **End**
- The barber prints **End** before the customer prints **L**

But How? — use semaphores for ordering

```
semaphore barber_done = 0 // Is the barber done?
barber()
  while true do
    customers.down() // wait for (or get) a customer
    acquire(access_lock)
    occupied_chairs = occupied_chairs - 1
    release(access_lock)
    barber.up() // the barber is now ready to start
    cut_hair()
    barber_done.up() // the barber is now done with the haircut
customer()
  acquire(access_lock)
  if occupied_chairs < N then
    occupied_chairs = occupied_chairs + 1
    release(access_lock)
    customers.up() // one more customer has taken a chair
    barber.down() // waiting for barber to get ready
    get_haircut()
    barber_done.down() // waiting for barber to be done
  else
    release(access_lock) // leave the barber shop
```