The Dining Philosophers

Operating Systems
Baochun Li
University of Toronto

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 (Textbook 31.6)
- The Sleeping Barber Problem (not in the textbook, but in Lab 3)

The Dining Philosophers

The Dining Philosophers Problem

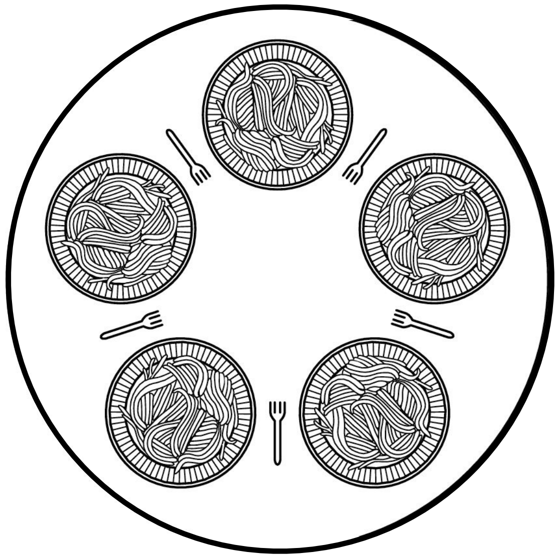
Five philosophers sit at a table

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
```

Is this a valid 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) modulo 5)
putdown_forks(int i)
    putdown_fork(i)
    putdown_fork((i+1) modulo 5)
```

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!

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 solution looks like this — correct?

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

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 pickup_fork() and putdown_fork()?

- 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}
int status[5] = {5 of THINKING}
pickup_forks(int i)
   mutex.down()
   status[i] = HUNGRY
   int left = (i+4) modulo 5, right = (i+1) modulo 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) modulo 5, right = (i+1) modulo 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) modulo 5, right = (i+1) modulo 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!

Changing if to while?

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
```

Changing if to while?

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) modulo 5, right = (i+1) modulo 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) modulo 5, right = (i+1) modulo 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) modulo 5, right = (i+1) modulo 5

    if status[left] == HUNGRY and
       status[(left+4) modulo 5] != EATING then
      status[left] = EATING
      sem[left].up()
    if status[right] == HUNGRY and
       status[(right+1) modulo 5] != EATING then
      status[right] = EATING
      sem[right].up()
    mutex.up()
```

Now you see why we need monitors!

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

Task 1 in Lab 3 asks you 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) modulo 5)
putdown_forks(int i)
   putdown_fork(i)
   putdown_fork((i+1) modulo 5)
```

Towards designing a simpler solution

```
semaphore forks[5]= {5 of 1}
pickup_fork(int i)
  forks[i].down()
putdown_fork(int i)
  forks[i].up()
```

But what about the deadlock?

Making the solution deadlock-free

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

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

What we’ve covered so far

Three Easy Pieces: Chapter 31.6