

# Classic Concurrency Problems

## ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

### Classic Problems

- Studying concurrency in real-world applications is always difficult
  - These applications have their own idiosyncrasies
  - They are often very large and it would take hours for us to understand how they work
- So people have designed easy-to-understand applications that raise relevant and challenging concurrency issues
  - Modeling "everyday life" situations
- We have looked at
  - Producer / Consumer
  - Reader / Writer
- We'll look at those in some detail (in pseudo-code, not Java)
  - Savings Account (very simple)
  - Barbershop (still pretty easy)
  - Dining Philosophers (difficult and very famous)

### Shared Bank Account

- Consider a bank account shared by multiple people
- There are two operations
  - deposit(): adds money to the account
  - withdraw(): remove money
    - Should block if not enough money
- Let's look at a solution using a monitor

### The Bank Account Monitor

```
Monitor BankAccount {
    int total=0;
    cond more_money;

    void deposit(int amount) {
        total += amount;
        signal_all(more_money);
    }
    void withdraw(int amount) {
        while (amount > total) {
            wait(cond);
        }
        total -= amount;
    }
}
```

- A very simple problem, very similar to producer / consumer
- An optimization would consist in having the minimum withdrawal get priority
- Remember that this is a monitor, thus with (implicit) mutual exclusion on the deposit() and withdraw() methods

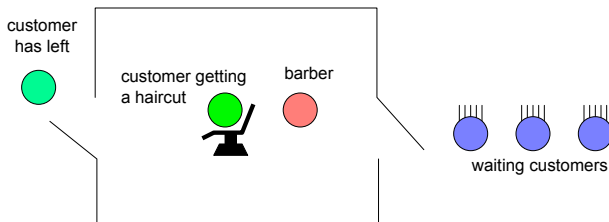
### Bank Account

- A very simple problem, very similar to producer / consumer
- An optimization would consist in having the minimum withdrawal get priority

### The BarberShop Problem

- A pretty simple problem also, just about thread communication
- The Barber provides a service (i.e., a haircut) to incoming customers
  - opens the door to the shop
  - waits for a customer
  - gives a haircut
  - tells the customer to leave
  - waits until the customer has left through the back (it's a polite barber)
- The Customer
  - wait for the door to open
  - enters the barber shop
  - waits until the barber is available
  - waits until the haircut is finished
  - leaves the shop through the back door
- The problem: to develop a Barber Shop monitor

## The BarberShop



## The BarberShop Monitor

- We must implement three methods
  - getHaircut(): called by customers
  - getNextCustomer(): called by the barber when free
  - finishedCut(): called by the barber when done

```
void Barber() {
    while (true) {
        BarberShop.getNextCustomer();
        <Cut hair>
        BarberShop.finishedCut();
    }
}
```

```
void Customer() {
    BarberShop.getHaircut();
}
```

```
void Customer() {
    BarberShop.getHaircut();
}
```

```
void Customer() {
    BarberShop.getHaircut();
    ...
}
```

## The BarberShop Implementation

- We use four boolean flags
  - barber: IDLE / WORKING (barber)
  - left: GONE / STILL\_HERE (customer just serviced)
  - door: OPEN / CLOSED
  - chair: OCCUPIED / FREE
- We use four condition variables
  - The barber waits on
    - chair\_occupied: a customer just sat down (chair = OCCUPIED)
    - customer\_left: the recently served customer just left (left = GONE)
  - The customer waits on
    - door\_open: the entrance is open (door = OPEN)
    - haircut\_done: the haircut is done (barber = IDLE)

## The BarberShop Monitor

```
Monitor BarberShop {
    boolean barber = IDLE;
    boolean chair = FREE;
    boolean left = GONE;
    boolean door = OPEN;

    cond chair_occupied;
    cond customer_left;
    cond barber_available;
    cond door_open;

    void getHaircut() { ... }
    void getNextCustomer() { ... }
    void finishedCut() { ... }
}
```

## BarberShop.getHaircut()

```
void getHaircut() {
    // wait for door to open
    while (door == CLOSED) {
        door_open.wait();
    }
    door = CLOSED;

    // make the barber non-idle
    barber = WORKING;
    chair = OCCUPIED;
    chair_occupied.signal();
    // wait for the barber to be idle
    while (barber == WORKING) {
        haircut_done.wait();
    }
    chair = FREE;
    left = GONE;
    customer_left.signal();
}
```

```
void getNextCustomer() {
    while (chair == FREE) {
        chair_occupied.wait();
    }
}
```

```
void finishCut() {
    barber = IDLE;
    haircut_done.signal();
    while (left == STILL_HERE) {
        customer_left.wait();
    }
    door = OPEN;
    door_open.signal();
}
```

## BarberShop with Monitors

- Overall, a pretty natural solution but that requires a bit of thoroughness
- Different solutions are possible with different flags / condition variables
  - We decided arbitrarily who sets which variables, could be done otherwise
  - Many solutions available on the Web
- One interesting exercise: How to implement this using only Semaphores?
- Turns out is actually easier
- Let's look at it

## BarberShop with Semaphores

- We're going to have one binary semaphore per "resource":
  - left: the "fact" that the last customer has left (init = 0)
  - barber: the barber (init = 0)
  - door: the front door (init = 0)
  - chair: the chair (init = 0)

```
void getHaircut() {
    P(door); // wait for door to be open
    V(chair); // seat in the chair
    P(barber); // wait for barber to be done
    V(left); // leave the shop
}
```

```
void getNextCustomer() {
    V(door); // open the door
    P(chair); // wait for chair to be taken
}
```

```
void finishCut() {
    V(barber); // say "I am done"
    P(left); // leave the shop
}
```

## The Dining Philosophers Problem

- A classical synchronization problem
  - pretty meaningless at face value
  - but representative of many real-world problems
- 5 philosophers sit at a table with 5 plates and 5 forks/chopsticks
- Each philosopher does two things:
  - think for a while
  - eat for a while
  - repeat
- To eat, a philosopher needs **two** forks/chopsticks



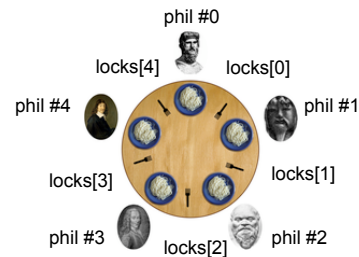
## Philosopher Algorithm

```
void philosopher() {
    <think>
    pickupForks();
    <eat>
    putdownForks();
}
```

- **Question:** how to implement the pickupForks() and putdownForks() methods?
  - putdownForks() is actually straightforward

## "Protected" Forks

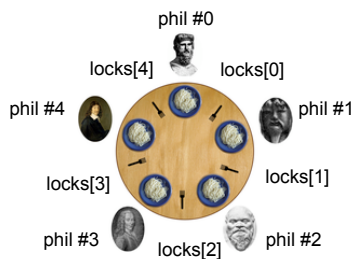
- We need to avoid two philosophers having the same chopstick in hand
- Ideas: Use an array of "locks", one for each fork
  - Acquiring the lock means "getting the fork"
  - Releasing the lock means "giving up the fork"
- These are "conceptual" locks (e.g., may be something else in Java)



- To eat, philosopher #i must acquire lock[(i+4) % 5] and lock[i]

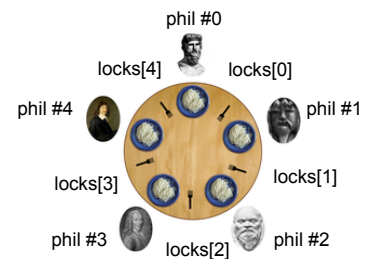
## Implementation Idea #1

```
int right(int phil) {
    return ((phil+4) % 5);
}
int left(int phil) {
    return phil;
}
void pickupForks(int phil) {
    lock(locks[left(phil)]);
    lock(locks[right(phil)]);
}
void putdownForks(int phil) {
    unlock(locks[left(phil)]);
    unlock(locks[right(phil)]);
}
```



## Solution #1

```
int left(int phil) {
    return ((phil + 4) % 5);
}
int right(int phil) {
    return phil;
}
void pickupForks(int phil) {
    lock(locks[left(phil)]);
    lock(locks[right(phil)]);
}
void putdownForks(int phil) {
    unlock(locks[left(phil)]);
    unlock(locks[right(phil)]);
}
```



what is wrong in this solution?

## Solution #1 Deadlocks

- If all philosophers pick up the fork on their left simultaneously and then try to pick up the fork on their right, then we have a **deadlock**
- The deadlock may happen very rarely on a single proc system
  - what are the odds that a thread is interrupted right in between the two calls to pthread\_lock()
- May happen more frequently on a multi-core system
- At any rate, one is never guaranteed that the code will not block at some point in time
  - Think of a server that must stay up for months...
- **Question:** What's a deadlock-free implementation?

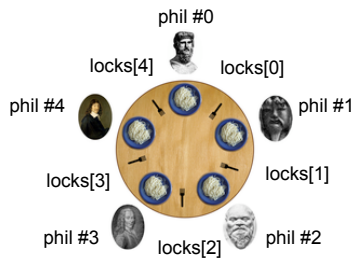
## Solution #2

- A simple Idea: make the solution asymmetrical
  - Odd-numbered philosophers start with the left fork
  - Even-numbered philosophers start with the right fork

```
void pickupForks(int phil) {
    if (phil % 2 == 0) {
        lock(locks[right(phil)]);
        lock(locks[left(phil)]);
    } else {
        lock(locks[left(phil)]);
        lock(locks[right(phil)]);
    }
}
```

## Solution #2 doesn't Deadlock!

- If P1 gets to f1 before P2
  - P2 does not pick up f2
  - If P4 gets to f3 before P3
    - If P4 gets to f4 before P0
      - P4 eats!
      - P0 doesn't pick up f0
      - P1 eats
    - ...
- This kind of exhaustive reasoning is very tedious
- But we can see that at least two philosophers can always eat no matter what
- Formal reasoning for something like this can be very difficult

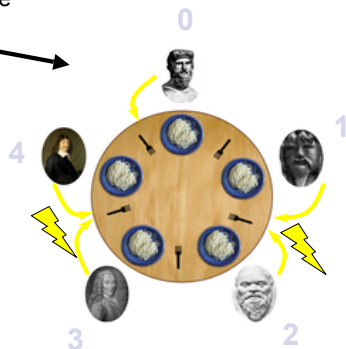


## Solution #2 isn't so great...

- Small possibility of **starvation**
  - A philosopher could put down a fork and pick it right back up
  - But this depends upon the way in which threads are implemented
  - And requires that a philosopher's think time could be 0 seconds
- Biggest problem: the implementation is unfair
  - One of the threads has an advantage over the others
  - Philosopher 0 doesn't face a lot of competition when picking up the fork on its right
  - Let's see this on a picture

## Solution #2 is Unfair

Unfair advantage because of less competition



## Towards a Fair Solution

- How can we not give an unfair advantage to Philosopher 0?
- The problem is that it's a jungle out there
  - There is no communication between philosophers
  - They have their eyes on the forks, and not on each other
- New idea:
  - When a philosopher wants to eat, he checks the forks
  - If they are available, he eats
  - otherwise, he waits on a condition variable
    - one condition variable per philosopher
  - when a philosopher finishes eating he checks to see if his neighbors are waiting
  - if so, he signals them so that they can recheck the forks
- Major difference: everything is about philosopher state not about the forks
  - THINKING, HUNGRY, EATING

## Solution #3

```
void pickupForks(int phil) {
    lock(mutex); // enter critical section
    state[phil] = HUNGRY;
    while ((state[left(phil)] == EATING) ||
           (state[right(phil)] == EATING)) {
        wait(cond[phil], mutex);
    }
    state[phil] = EATING;
    unlock(mutex); // leave critical section
}
```

```
void putdownForks(int phil) {
    lock(mutex); // enter critical section
    if (state[left(phil)] == HUNGRY)
        signal(cond[left(phil)]);
    if (state[right(phil)] == HUNGRY)
        signal(cond[right(phil)]);
    state[phil] = THINKING;
    unlock(mutex); // leave critical section
}
```

- One lock for mutual exclusion
- One array of condition variables, one per philosopher
- All philosophers are equal
- Still a problem :(

## Solution #3 not that good...

- Risk of starvation
  - There could be a ping-pong effect
    - P0 and P2 get to eat
    - P1 and P3 get to eat
    - P0 and P2 get to eat
    - ....
    - P4 never gets to eat!
- This is rare, but could happen in the long run
- It would be nice to have something that is guaranteed to work well and fairly

## Solution #4: The Queue

- To guarantee fairness one can use a queue of philosophers
  - When a philosopher finds that he cannot eat, he is placed on a queue
  - Only the philosopher at the head of the queue is allowed to eat next
- Problem
  - A philosopher could find that he can pickup forks BUT he is not at the head of the queue
  - In this case he has to wait
  - Hence philosophers cannot eat as much as they want
  - So it's fair, but not very efficient
- Possible Solution
  - Allows philosophers to jump ahead in the queue when they use forks that are not needed by anybody ahead of them in the queue

## Solution #5: The Deli

- Use numbers (the "Deli" model)
  - When hungry, a philosopher takes a number
  - If a philosopher is hungry and so are his neighbors, the one with the lowest number gets to eat
  - Numbers always increase
- Works pretty well, but still can lead to poor performance with too much blocking
- Some solutions use a mix of everything we've seen so far...
- It turns out that having a deadlock-free and fair solution is rather difficult in theory
- Some of the solutions we have seen are good, but could potentially break down in particular situations
  - Depending on thinking / eating times
  - Depending on the number of philosophers

## Conclusion

- Main things to worry about
  - Deadlock
  - Starvation
  - Fairness
- For some problems it can be very difficult to come up with a good solution that works under all conditions

## Conclusion

- There are many solutions
  - difficult to find the best one
  - especially because it depends on the "workload"
    - 2 hungry philosophers, 7 not-so-hungry ones
    - ...
- Lessons
  - One must worry about deadlock
  - One must worry about starvation
    - enforce that each thread gets unblocked every now and then (queue, numbers)
    - can be detrimental to performance
  - Worry about treating all threads equally (not always needed, but often)