Classic Concurrency Problems

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Classic Problems

- Studying concurrency in real-world applications is always difficult
  - 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
  - Based on “everyday life” situations
- We have seen Producer / Consume and Reader / Writer
- Let’s look at a few others in some detail (in whatever pseudo-code)
  - Savings Account (very simple)
  - Barbershop (still pretty easy)
  - Dining Philosophers (difficult and very famous)
- We’ll look at possible solutions, and discuss pros and cons

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
- A simple problem, very similar to producer / consumer at first glance
  - The difference is that one can deposit and withdraw more than one dollar at a time
- Let’s look at a solution with locks/condvars

With Locks/Condvars

```
public class BankAccount {
  int total=0;
  Condvar more_money;
  Lock mutex;

  void deposit(int amount) {
      mutex.lock();
      total += amount;
      mutex.unlock();
      more_money.signal_all();
  }

  void withdraw(int amount) {
     mutex.lock();
      while (amount > total) {
          more_money.wait(mutex);
      }
      total -= amount;
      mutex.unlock();
  }
}
```

- A bit brute-force: we wake up everyone for every deposit!
- Problem: starvation
- Anybody sees why?…

With Locks/Condvars

```
public class BankAccount {
  int total=0;
  Condvar more_money;
  Lock mutex;

  void deposit(int amount) {
      mutex.lock();
      total += amount;
      mutex.unlock();
      more_money.signal_all();
  }

  void withdraw(int amount) {
     mutex.lock();
      while (amount > total) {
          more_money.wait(mutex);
      }
      total -= amount;
      mutex.unlock();
   }
}
```

- A bit brute-force: we wake up everyone for every deposit!
- Problem: starvation
- Anybody sees why?…
- A large withdrawal can constantly be overtaken by a stream of small withdrawals…
  - A: withdraw(10000)
  - B: while (true) { withdraw(1); }
- Before we try to fix this, let's attempt to do the exact same this with semaphores…

With Semaphores

```
int total = 0
Semaphore mutex = 1
Semaphore money = 0
```

```
void deposit(int amount) {
      mutex.P();
      total += amount;
      money.V();
      mutex.V();
 }
```

```
void withdraw(int amount) {
   mutex.P();
    while (amount > total) {
         mutex.V();
         money.P();
         mutex.P();
    }
    total -= amount
    mutex.V();
}
```

This is not very semaphore-like: we're using the total variable to keep track of the money in the account (using a counting semaphore instead comes to mind)

It turns out that this doesn't actually work… any ideas why?

With Semaphores

```
int total = 0
Semaphore mutex = 1
Semaphore money = 0
```

```
void deposit(int amount) {
      mutex.P();
      total += amount;
      money.V();
      mutex.V();
 }
```

```
void withdraw(int amount) {
   mutex.P();
    while (amount > total) {
            mutex.V();
            money.P();
            mutex.P();
    }
    total -= amount
    mutex.V();
}
```

- Thread A: withdraw(500)
- Thread B: withdraw(500)
- Thread C: deposit(1000)
- Only one of A or B is "awakened", and the other ones may sleep forever even though there is enough money in the account for its withdrawal
- No direct equivalent of signal_all() in the monitor solution
  - But we know that we should be able to use any synchronization paradigm as they are all equivalent… that means we need to make the code more complicated

With Semaphores

- One possible solution

```
int total = 0
sem_t mutex = 1
sem_t onedollar = 0
```

```
void deposit(int amount) {
    mutex.P();
    total += amount;
    for (i=0; i < amount; i++)
        onedollar.V();
    mutex.V();
}
```

```
void withdraw(int amount) {
    mutex.P();
    while (amount > 0) {
        mutex.V();
        for (i=0; i < amount; i++) {
            onedollar.P();
            amount--;
        mutex.P();
    }
    total -= amount;
    mutex.V();
}
```

- By calling V() for each dollar, and calling P() for each dollar now we don't have the problem that a withdrawer can "miss" a call to V()
  - But it has high overhead for large $ amounts
- We have another problem, that we have seen before with reader-write, if we have two withdrawals happening concurrently: splitting the amount….

Bank Account with Semaphores

```
int total = 0
sem_t mutex = 1
sem_t onedollar = 0
```

```
void deposit(int amount) {
      mutex.P();
      total += amount;
      for (i=0; i < amount; i++)
          onedollar.V();
      mutex.V();
 }
```

```
void withdraw(int amount) {
   mutex.P();
    while (amount > 0) {
           mutex.V();
           for (i=0; i < amount; i++) {
              onedollar.P();
              amount--;
           mutex.P();
    }
    total -= amount;
    mutex.V();
}
```

- Say two withdrawals for $500 happens are ongoing and $500 is deposited
- With the above code it's possible that each withdrawer gets $250 and then is stuck
- So we have starvation again…

Sequential Withdrawals

- We have a starvation problem in all our previous solutions because withdrawals can happen "simultaneously"
- Let's now opt for a brute-force solution to the starvation problem: force withdrawals to happen in order!
- Let's do this both for our lock/condvar and our semaphore solution….

With Lock/Condvars

```
public class BankAccount {
  int total=0;
  Condvar more_money;
  Lock mutex, withdrawing;

  void deposit(int amount) {
      mutex.lock();
      total += amount;
      mutex.unlock();
      more_money.signal_all();
  }

  void withdraw(int amount) {
     withdrawing.lock()
     mutex.lock();
      while (amount > total) {
          more_money.wait(mutex);
      }
     total -= amount;
     mutex.unlock();
     withdrawing.lock();
  }
```

- By using an additional lock, we can force withdrawals to happen in sequence
- Let’s do it with semaphores…

With Semaphores

```
int total = 0
Semaphore mutex = 1
Semaphore withdrawing = 1
Semaphore money = 0
```

```
void deposit(int amount) {
    mutex.P();
    total += amount;
    money.V();
    mutex.V();
}
```

```
void withdraw(int amount) {
    withdrawing.P();
    mutex.P();
    while (amount > total) {
        mutex.V();
        money.P();
        mutex.P();
    }
    total -= amount
    mutex.V();
    withdrawing.V()
}
```

- Now that withdrawals happen in sequence, we don't have to go dollar-per-dollar and can use single calls for money.V() and money.P()
- This works but is very non-semaphore-like, let's now use a counting semaphore…

With Semaphore

- A nicer, more semaphore-esque solution

```
sem_t balance = 0
sem_t withdrawing = 1
```

```
void deposit(int amount) {
    for (i=0; i < amount; i++)
      balance.V();
 }
```

```
void withdraw(int amount) {
    Withdrawing.P();
   for (i=0; i < amount; i++)
      balance.P();
    withdrawing.V();
}
```

- Using a counting semaphore removes the need for the total variable, which makes the code much better (not while/if statements)
- But then it goes dollar-per-dollar, which has higher overhead again….

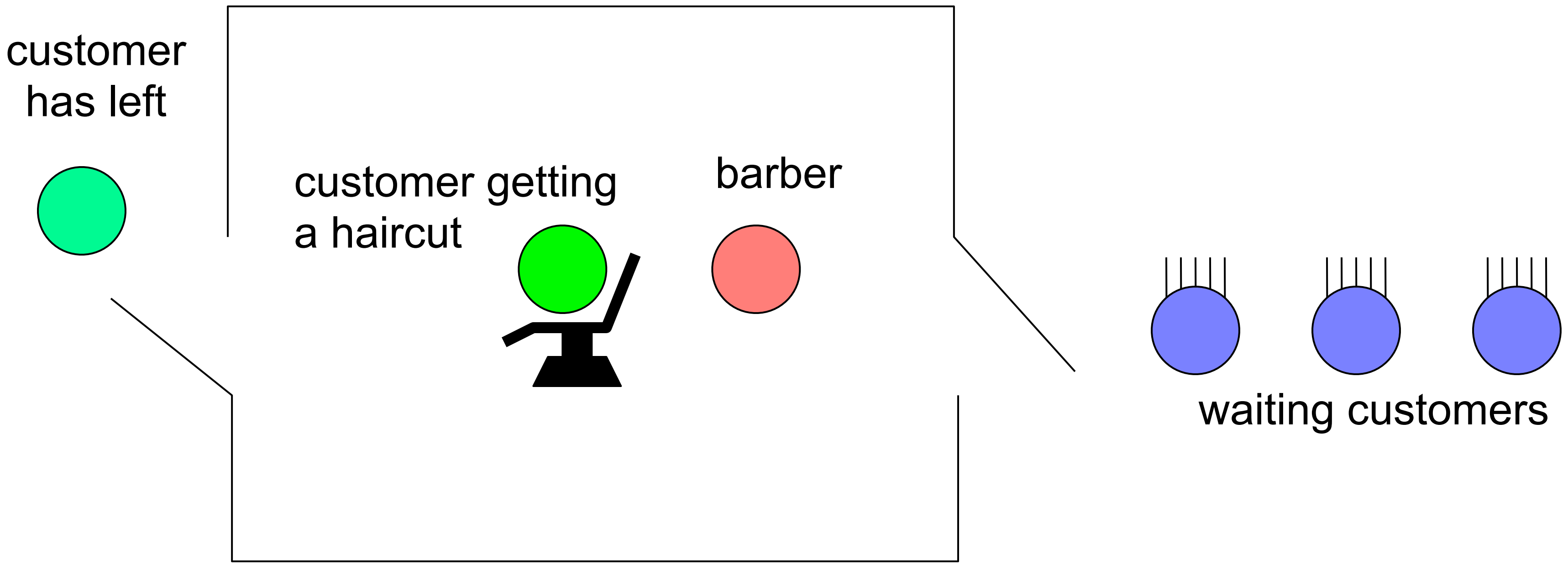
Bank Account

- Each solution has its own “features”
  - Starvation behaviors
    - Which we “fixed” by imposing a sequential order on withdrawals, which is both good and bad
  - Code complexity
  - Overhead
- Depending on the desired behavior and the use case, some solutions may be preferable
- Aiming for a great solutions across all use cases is perhaps not possible, and you can see how one could spend a lot of time designing it
- This is the whole point of these “metaphor” problems: perhaps there is no great solution, but thinking one is a great learning and thought experiment

The BarberShop Problem

- A simpler problem, for which there are great solutions
- It's "just" about thread communication
- The Barber provides a service (i.e., a haircut) to 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
- The Customer
  - waits 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: develop a Barber Shop monitor

The BarberShop



With Locks/Condvars

- 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();
}
```

. . .

BarberShop with locks/cond vars

- 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)

With Locks/Condvar

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

  Condvar chair_occupied;
  Condvar customer_left;
  Condvar barber_available;
  Condvar door_open;
  Condvar haircut_done;

  Lock mutex;

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

BarberShop Implementation

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

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

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

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

With Locks/Condvars

- 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 differently
  - Many solutions available on the Web
  - Some more readable than others, but that's typically pretty subjective
  - **Key point: pick good names for variables/flags**

- Still, it's a lot of code… can we do better with semaphores?
  - After all, semaphores are so easy for communication

BarberShop with Semaphores

- Let's have one binary semaphore per "resource":
  - **left**: the "fact" that the last customer has left (init = 0)
  - **barber**: the "fact" that the barber has finished (init = 0)
  - **door**: the "fact" that the front door is open (init = 0)
  - **chair**: the "fact" that the chair is empty (init = 0)

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

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

```
void finishCut() {
  barber.V(); // say "I am done"
  left.P();   // wait for customer
              // to have left
}
```

BarberShop with Semaphores

- Le

Because all we do is communication, semaphores are very elegant for the barbershop problem!

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

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

```
void finishCut() {
  barber.V(); // say “I am done”
  left.P();      // wait for customer
                   // to have left
}
```

Reader-Writer-like Problems

- Many people have come up with problems that ressemble, more or less, the reader-writer problem
- I just made this one up: you have a cloud, and two companies, A and B, that you charge for use
- A company can have an unlimited number of users in the cloud
- But there can never be users from the two companies in it at the same time
  - (companies are paranoid about industrial espionage)
- Let’s look at one solution…

Cloud Problem

```
void user(int id) {
  mutex.lock()
  // Wait for cloud to be void of the other company
  while (count[1 - id] > 0) {
    go_ahead[id].wait(mutex)
   }
  counts[id]++
  mutex.unlock()

  // Use the server

  mutex.lock();
  count[id]--
  if (count[id] <= 0)  {
    go_ahead[1-id].signal_all();
  }
  mutex.unlock();
}
```

```
int counts[2] = {0,0};
lock mutex;
cond go_ahead[2];

#define id_A 0
#define id_B 1
```

Cloud Problem

```
void user(int id) {
  mutex.lock()
  // Wait for cloud to be void of the other company
  while (count[1 - id] > 0) {
    go_ahead[id].wait(mutex)
   }
  counts[id]++
  mutex.unlock()

  // Use the server

  mutex.lock();
  count[id]--
  if (count[id] <= 0)  {
    go_ahead[1-id].signal_all();
  }
  mutex.unlock();
}
```

```
int counts[2] = {0,0};
lock mutex;
cond go_ahead[2];

#define id_A 0
#define id_B 1
```

Works, but has starvation! (just like the naive reader-writer)

Cloud Problem (#2)

- Let's now say that we have only 3 servers the cloud
- We now need to have users wait for users of their own company to be done using the servers!
- Let's look at the solution, which is a bit more complicated…

Cloud Problem (#2)

```
void user(int id) {
 mutex.lock();
 // Wait for cloud to be void of the other company
 while (waiting[1-id] > 0 || using[1 - id] > 0) {
   go_ahead[id].wait(mutex)
 }
 waiting[id]++;
 while (using[id] >= 3) {
   free_server.wait(mutex)
 }
 waiting[id]--;
 using[id]++
 mutex.unlock();

 // Use the server

 mutex.lock();
 using[id]--
 free_server.signal()
 if (waiting[id] <= 0 && using[id] <= 0)  {
   go_ahead[1-id].signal_all();
 }
 mutex.unlock();
}
```

```
int waiting[2] = {0,0};
int using[2] = {0,0};

lock mutex;
cond go_ahead[2];
cond free_server;


#define id_A 0
#define id_B 1
```

Cloud Problem (#2)

```
void user(int id) {
 mutex.lock();
 // Wait for cloud to be void of the other company
 while (waiting[1-id] > 0 || using[1 - id] > 0) {
   go_ahead[id].wait(mutex)
 }
 waiting[id]++;
 while (using[id] >= 3) {
   free_server.wait(mutex)
 }
 waiting[id]--;
 using[id]++
 mutex.unlock();

 // Use the server

 mutex.lock();
 using[id]--
 free_server.signal()
 if (count[id] <= 0 && waiting[id] <= 0)  {
   go_ahead[1-id].signal_all();
 }
 mutex.unlock();
}
```

```
int waiting[2] = {0,0};
int using[2] = {0,0};

lock mutex;
cond go_ahead[2];
cond free_server;


#define id_A 0
#define id_B 1
```

Works, but has starvation! (just like the naive reader-writer)

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

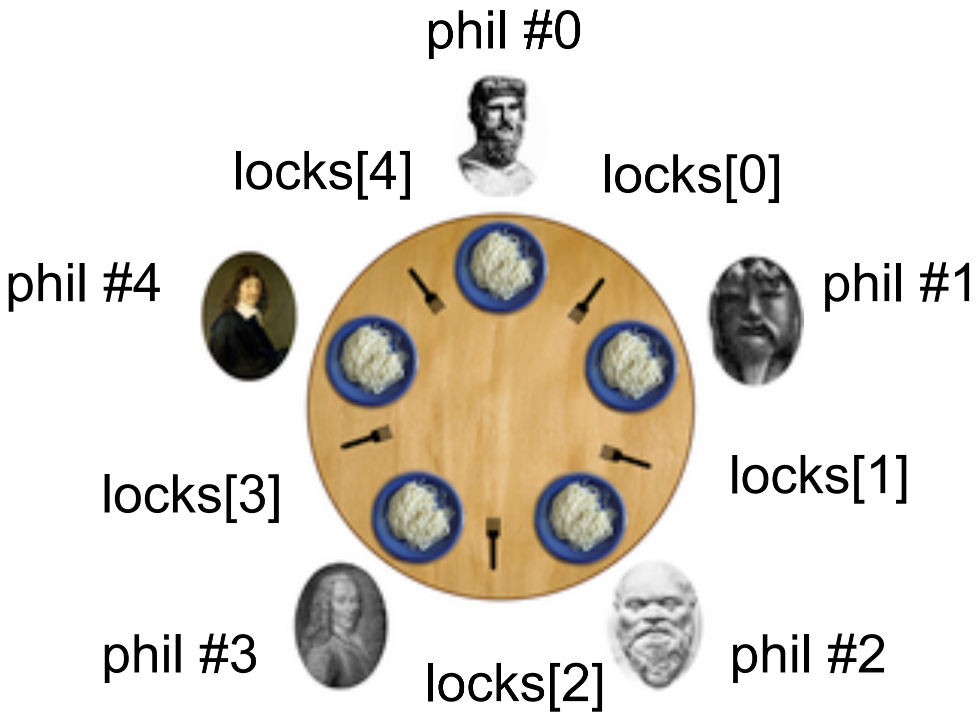
Philosopher Algorithm

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

- Problem: 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
- First Idea: 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)



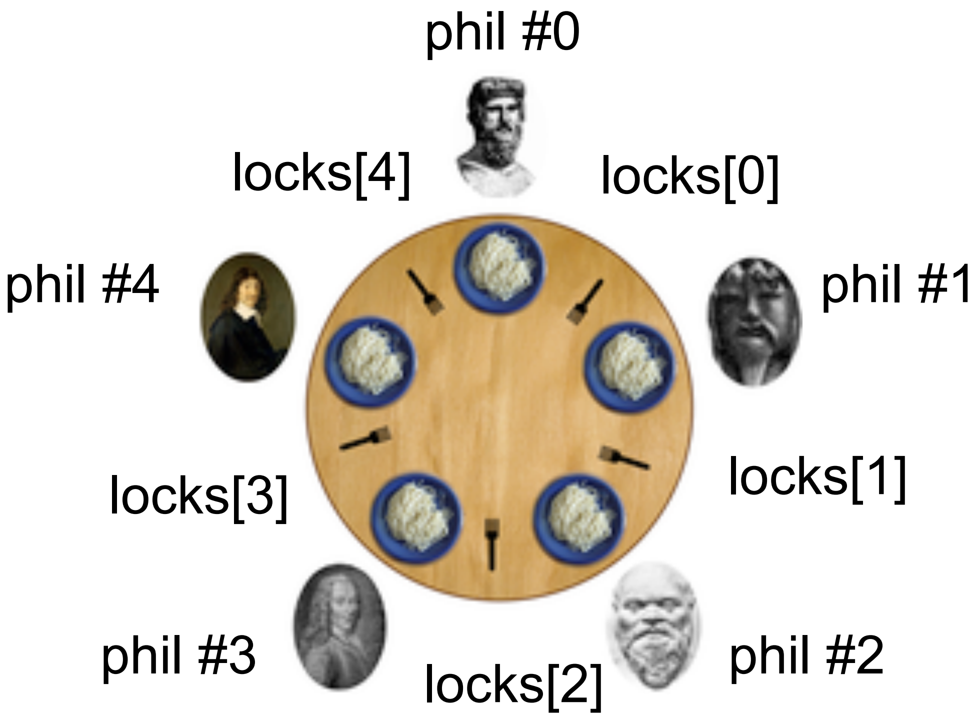
- To eat, philosopher #i must acquire lock[(i+1) % 5] and lock[i]

Implementation Idea #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)]);
}
```

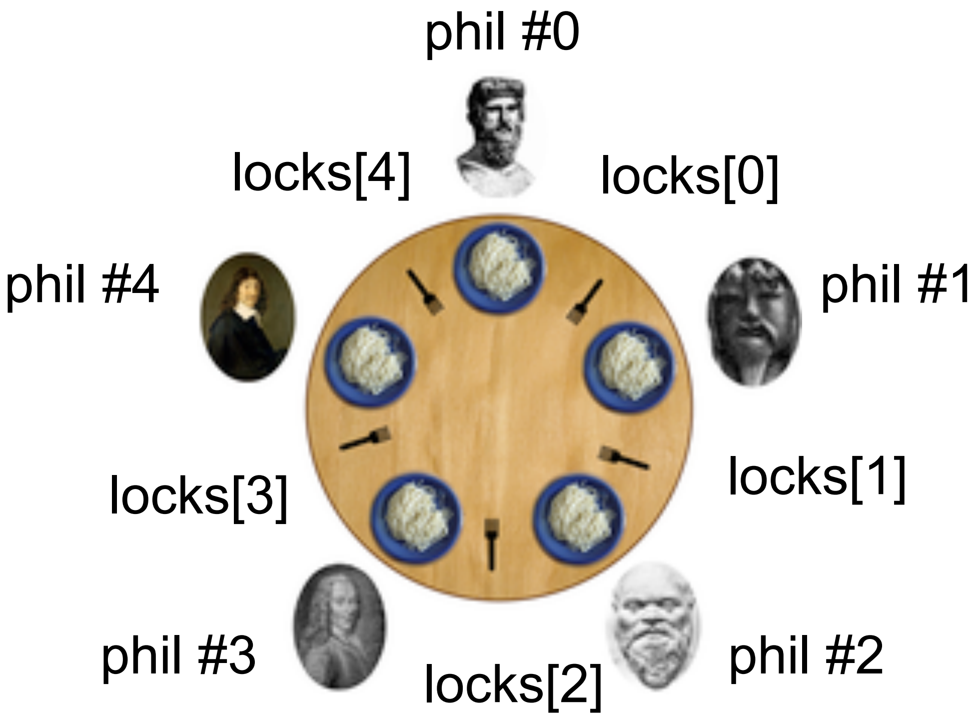


Solution #1

```
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 all threads are 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!

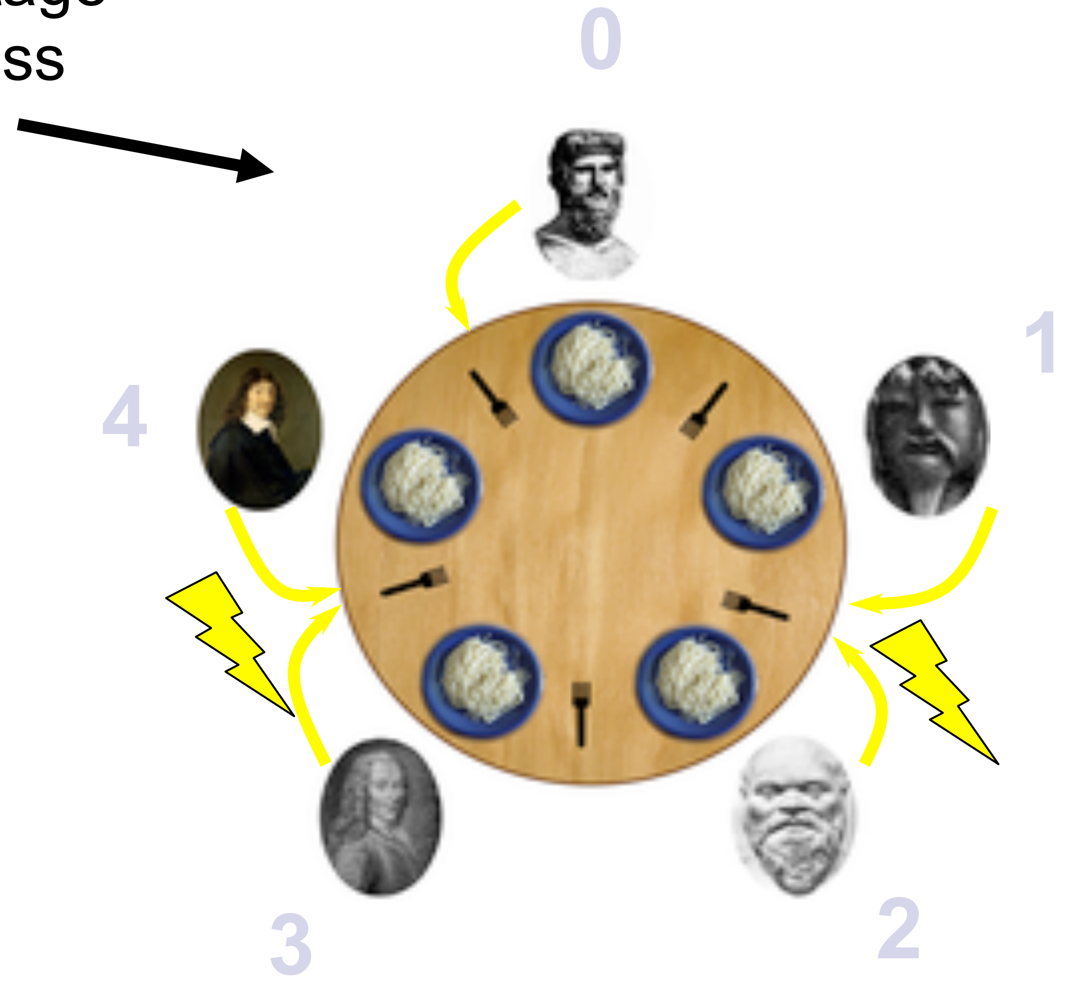
- If P1 gets to f1 before P2
  - P2 does not pick up f2
  - If P4 gets to f3 before P3
    - If P4 gets to f4 before P0
      - P4 eats!
      - P0 doesn't pick up f0
      - P1 eats
    - . . .
- This kind of exhaustive reasoning is very tedious
- But we can see that at least two philosophers can always 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
- At this point we're getting into the "theoretical" domain, while most "systems" people would be ok with what we already have

Solution #4: The Queue

- To guarantee fairness one can use a queue of philosophers
  - If a philosopher finds that he can eat, then great
  - **Otherwise,** he is placed in a queue
  - Only the philosopher at the head of the queue is allowed to eat among those in the queue (and gets removed from the queue)
- 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
- 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
  - Performance
- For some problems it can be very difficult to come up with a good solution that works under all conditions
  - There may be no such solution at all
- For some problems, semaphores are more elegant than monitors, for some others it's the other way around
  - Let's check out "The little book of semaphores!"
- In this course I don't have you do the beaten-to-death "implement dining philosophers" assignment
- But I have the assignment. So if you want the experience, let me know…