R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه هشتم – مسائل همزمانی، مانیتور و ارسال پیام

جلسه‌ی گذشته

سمافور و مسائل کلاسیک هم‌روندی

The Producer-Consumer Problem

- An example of the pipelined model
  - *One thread produces data items*
  - *Another thread consumes them*
- Use a bounded buffer between the threads
- The buffer is a shared resource
  - *Code that manipulates it is a critical section*
- Must suspend the producer thread if the buffer is full
- Must suspend the consumer thread if the buffer is empty

Semaphores

- An abstract data type
  - *containing an integer variable (S)*
  - *Two operations: Wait (S) and Signal (S)*

- Alternative names for the two operations
  - *Wait(S) = Down(S) = P(S)*
  - *Signal(S) = Up(S) = V(S)*

- Blitz names its semaphore operations Down and Up

پیاده‌سازی سمافور با میوتکس!؟

Solution?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = ulocked -- Protects access to "cnt"
    m2: Mutex = locked    -- Locked when waiting
```

Down ():

```
Lock(m1)
cnt = cnt - 1
if cnt<0
  Unlock(m1)
  Lock(m2)
endIf
Unlock(m1)
```

Up():

```
Lock(m1)
cnt = cnt + 1
if cnt<=0
  Unlock(m2)
else
  Unlock(m1)
endIf
```

Solution?

```
var cnt: int = 0              -- Signal count
var m1: Mutex = ulocked -- Protects access to "cnt"
    m2: Mutex = locked    -- Locked when waiting
```

Down ():

```
Lock(m1)
cnt = cnt - 1
if cnt<0
   Unlock(m1)
   Lock(m2)
endIf
Unlock(m1)
```

Up():

```
Lock(m1)
cnt = cnt + 1
if cnt<=0
   Unlock(m2)
else
   Unlock(m1)
endIf
```

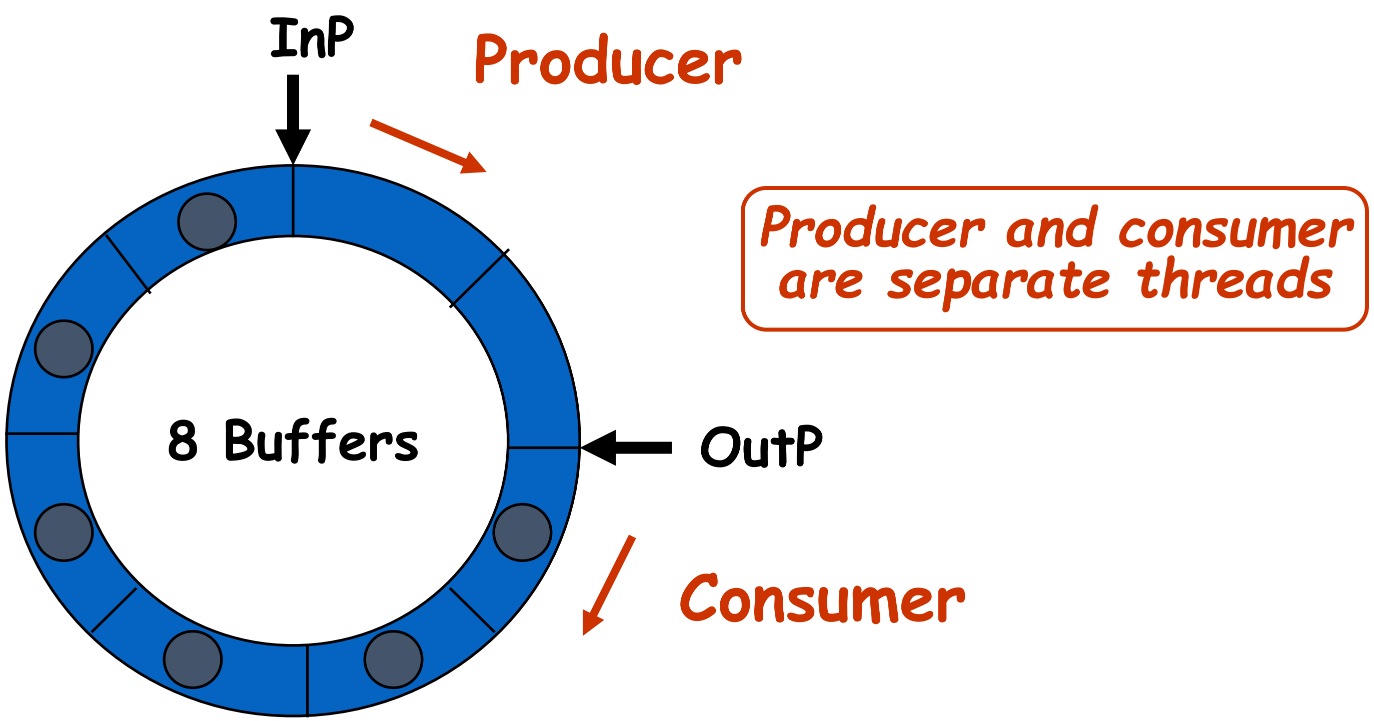
Contains a Race-Condition!

Classical Synchronization Problems

- Producer Consumer (bounded buffer)
- Dining philosophers
- Sleeping barber
- Readers and writers

Producer Consumer Problem

- Also known as the bounded buffer problem



Producer Consumer

```
Global variables
  semaphore full_buffs = 0;
  semaphore empty_buffs = n;
  char buff[n];
  int InP, OutP;
```

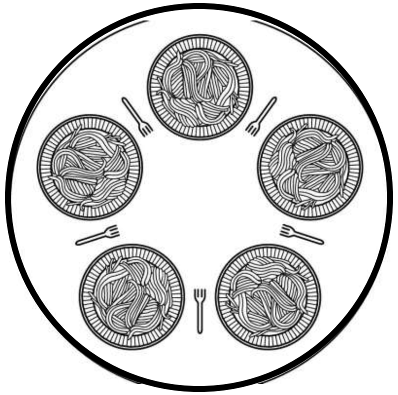
```
thread producer {
   while(1){
      // Produce char c...
      down(empty_buffs)
      buf[InP] = c
      InP = InP + 1 mod n
      up(full_buffs)
   }
}
```

```
thread consumer {
   while(1){
      down(full_buffs)
      c = buf[OutP]
      OutP = OutP + 1 mod n
      up(empty_buffs)
      // Consume char...
   }
}
```

Dining Philosophers Problem

- Five philosophers sit at a table
- One chopstick between each philosopher

(need two to eat)



Each philosopher is modeled with a thread

```
while(TRUE) {
  Think();
  Grab first chopstick;
  Grab second chopstick;
  Eat();
  Put down first chopstick;
  Put down second chopstick;
}
```

- *Why do they need to synchronize? How should they do it?*

Is This a Valid Solution?

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_chopstick(i);
    take_chopstick((i+1)% N);
    Eat();
    put_chopstick(i);
    put_chopstick((i+1)% N);
  }
}
```

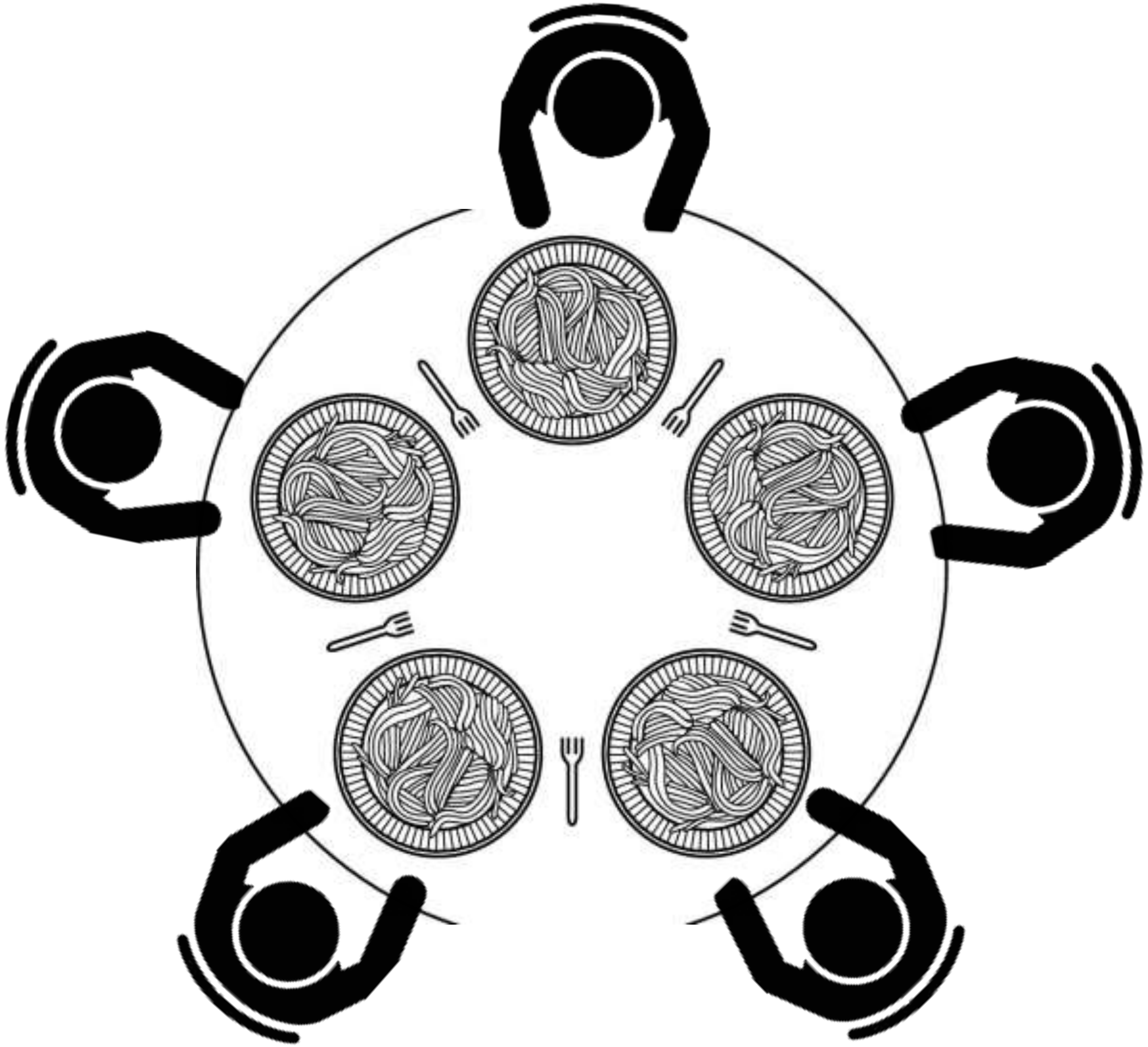
Problems

- Potential for deadlock !

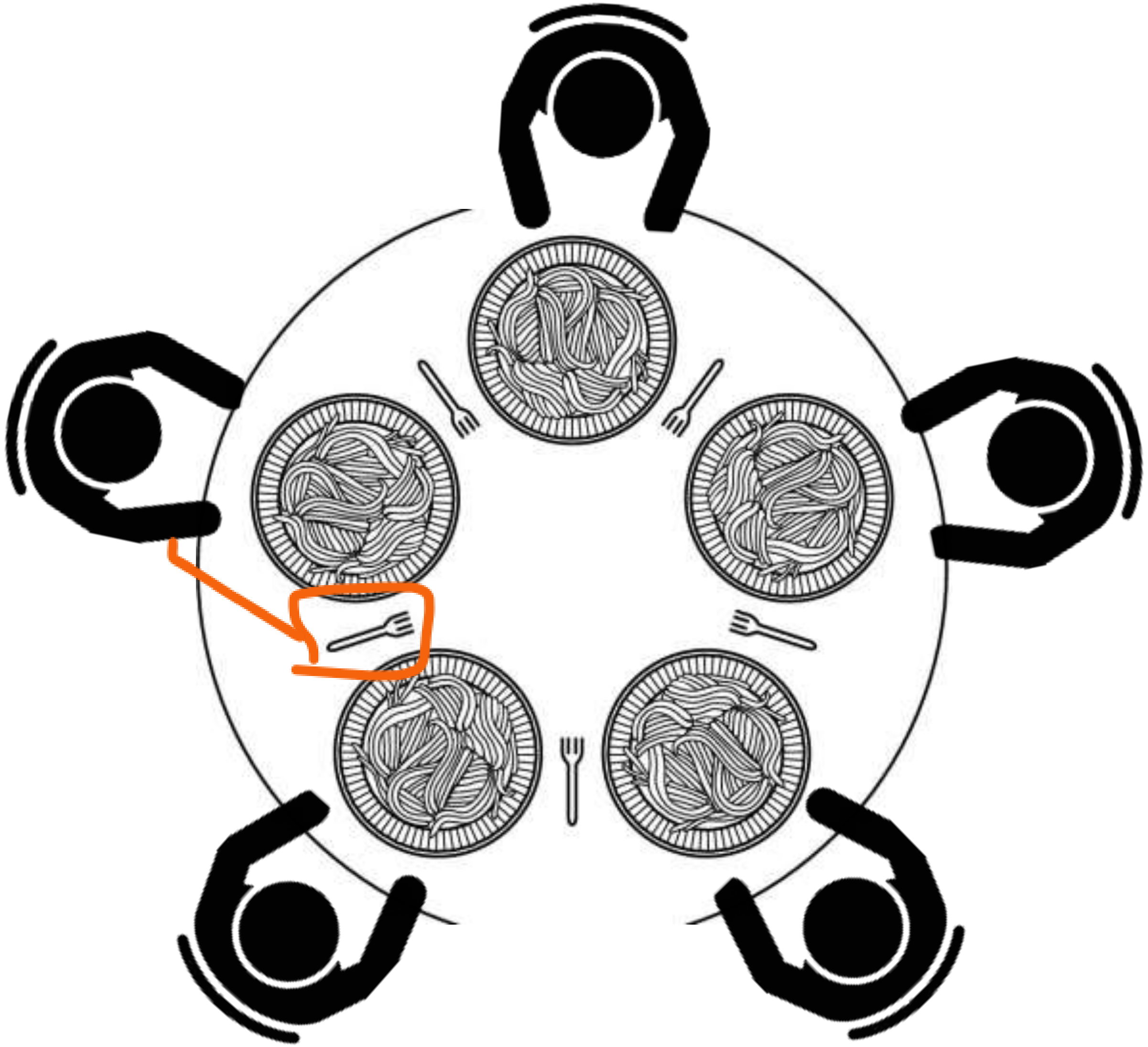
جلسه‌ی جدید

شام فیلسوفان

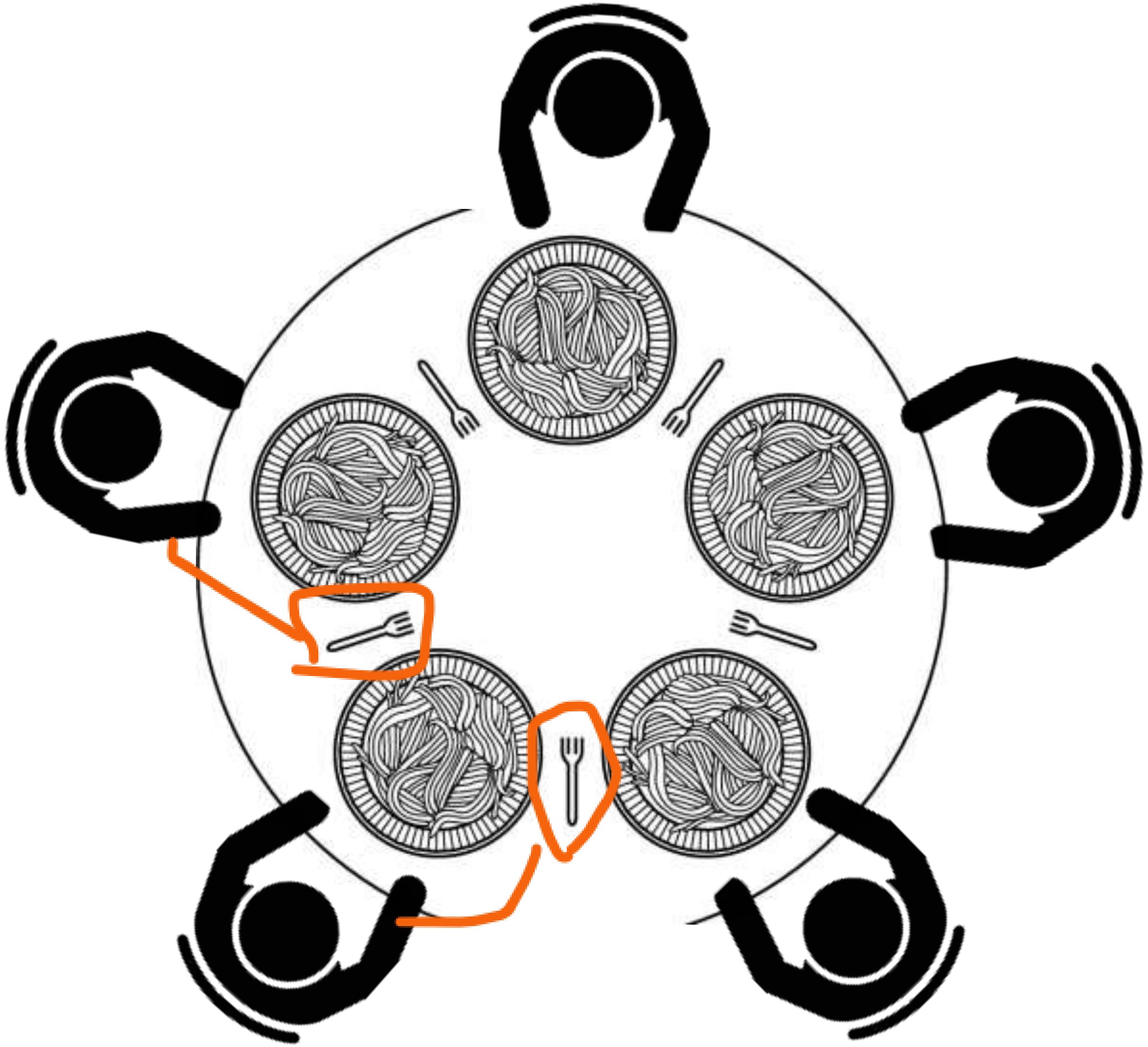
مشکل راه قبلی چه بود؟



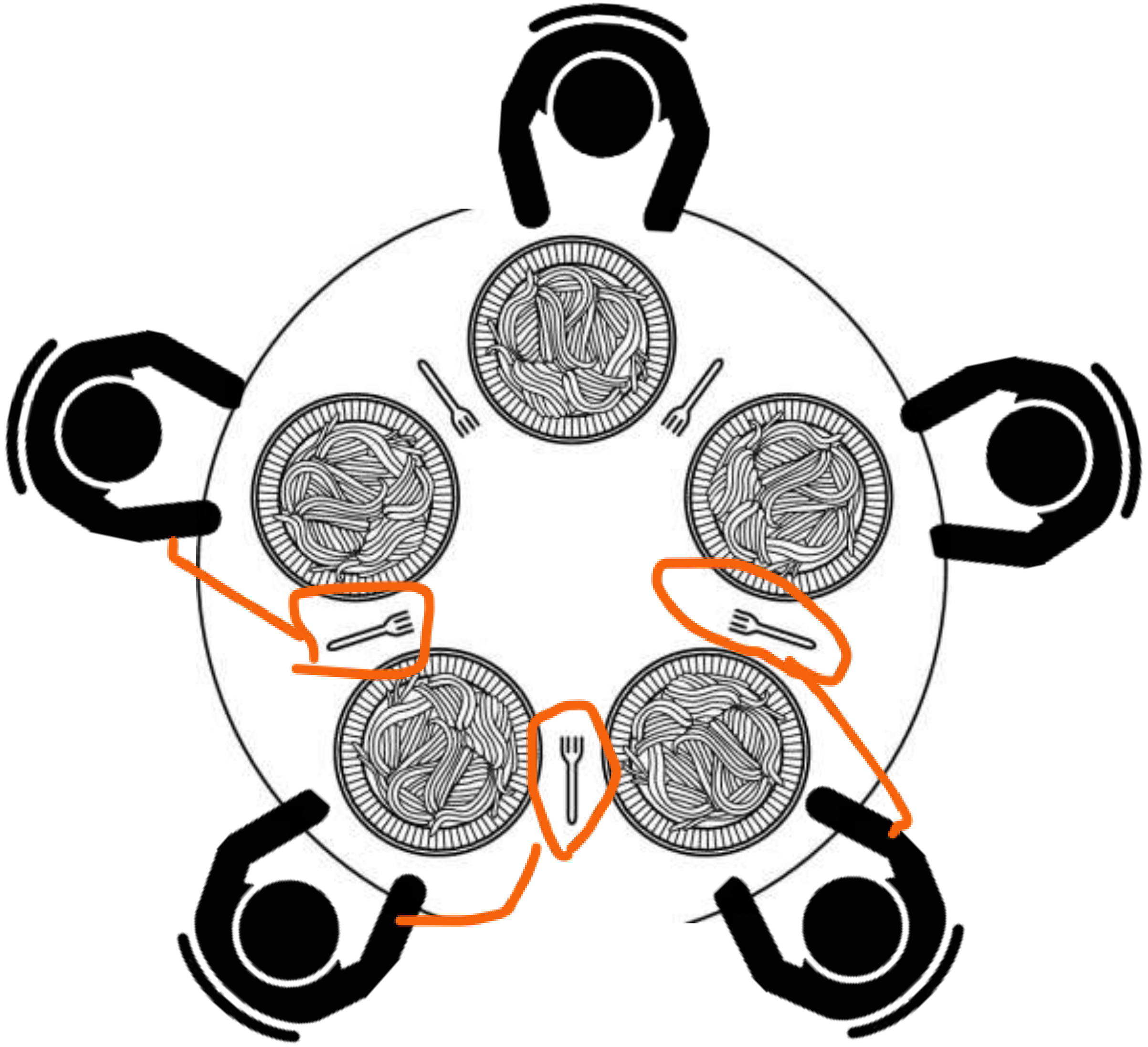
مشکل راه قبلی چه بود؟



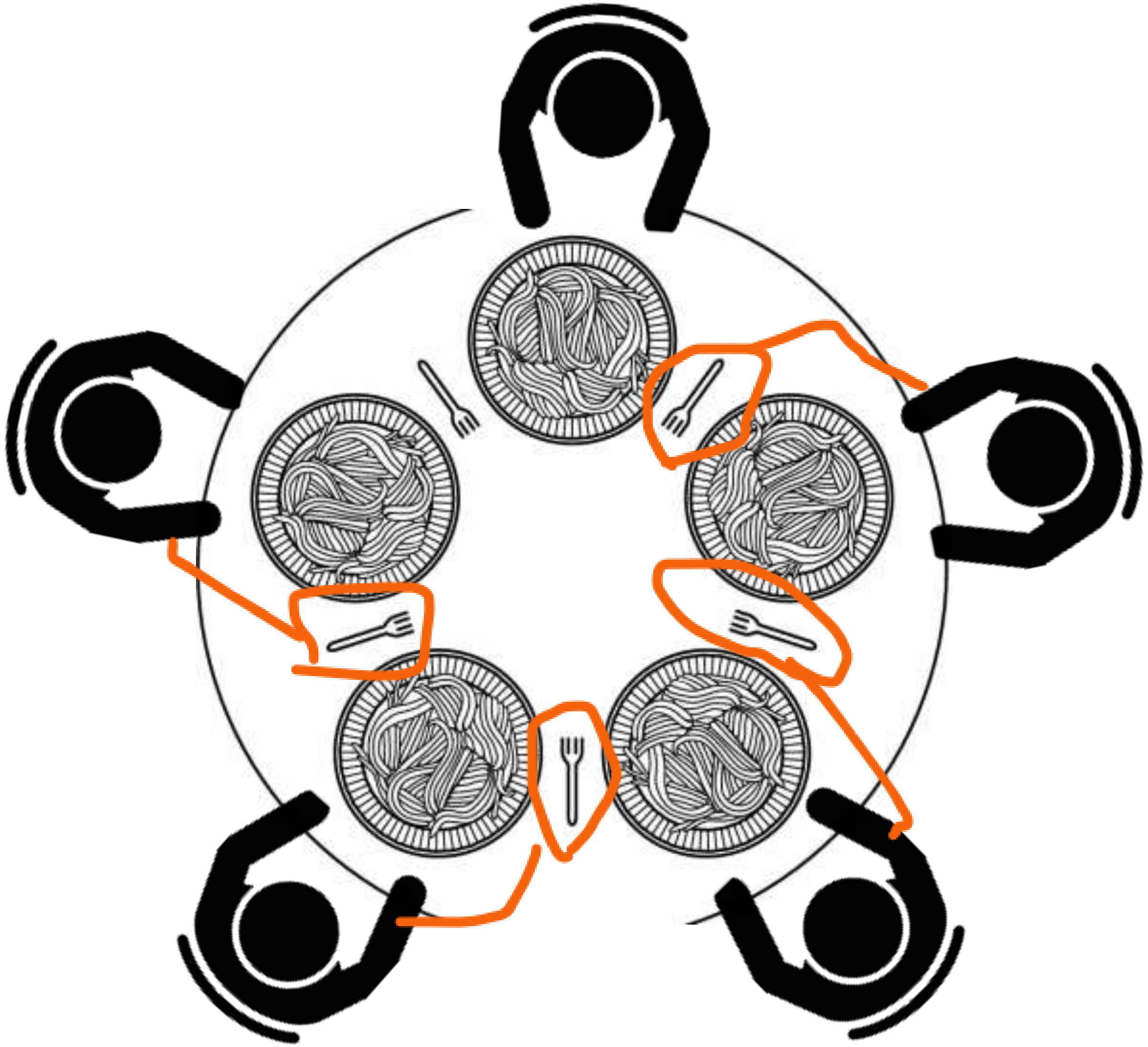
مشکل راه قبلی چه بود؟



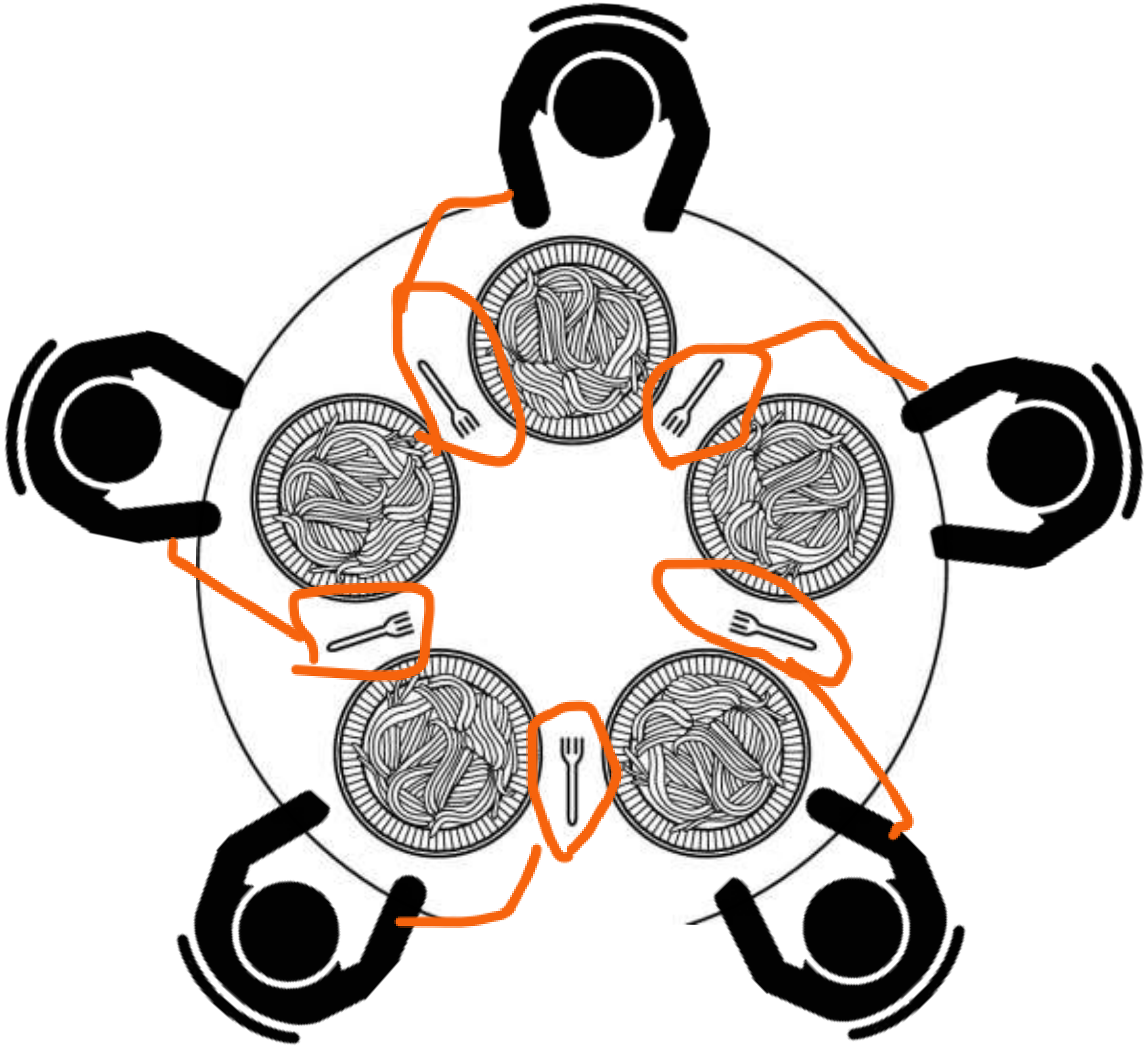
مشکل راه قبلی چه بود؟



مشکل راه قبلی چه بود؟



مشکل راه قبلی چه بود؟



مشکل راه قبلی چه بود؟

اندر مشکلات بن‌بست

- هر ۵ فیلسوف چوب دستشان است!
- یک فیلسوف چوب برداشته بدون اینکه غذا بخورد ☹
- همه با دست راست شروع به غذا خوردن کردند.

- پس اجازه ندهیم حداقل یکی از این‌ها رخ بدهد.

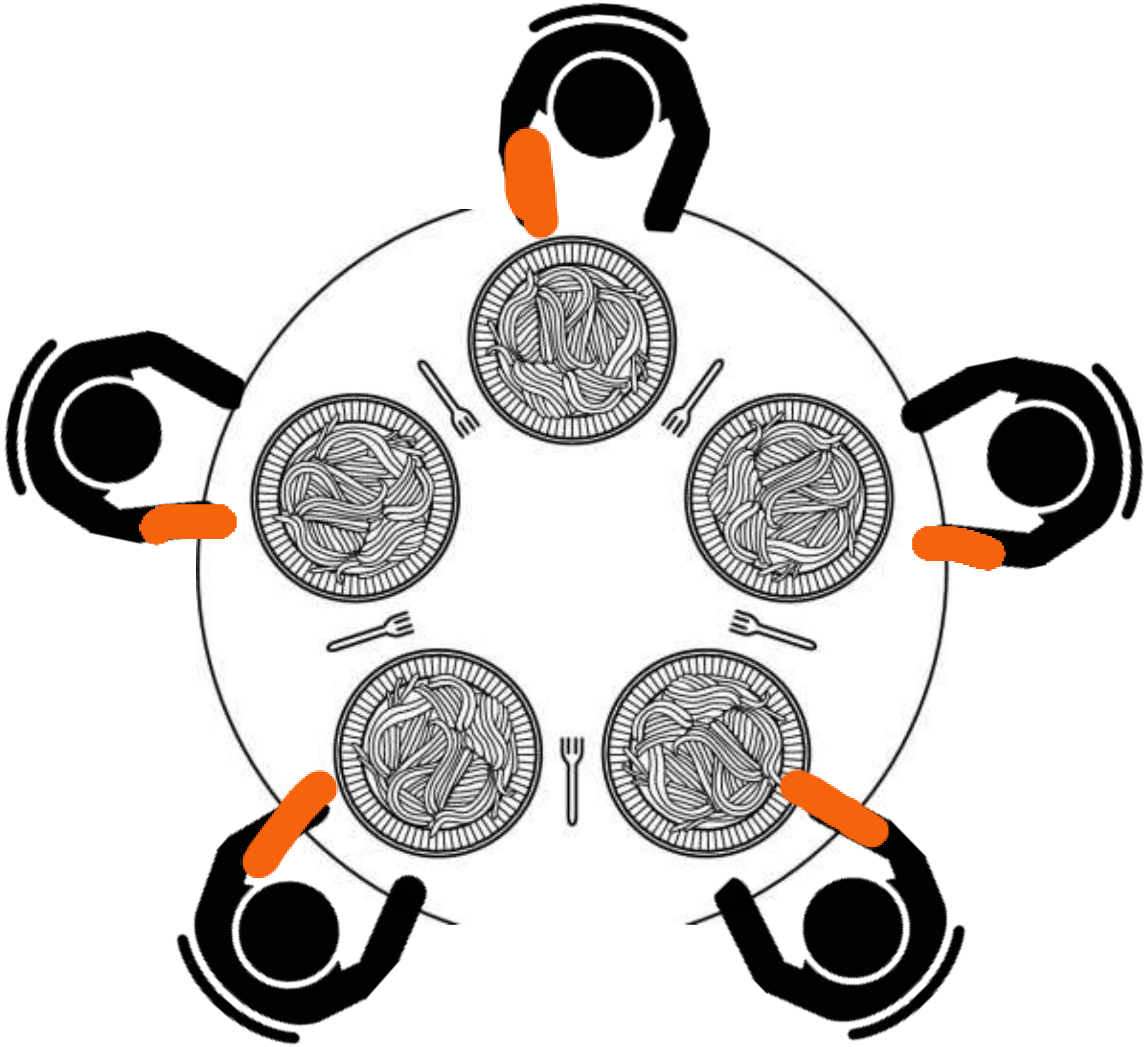
شام فیلسوف‌ها، ایده‌ی ۱

- همه با دست راست شروع به غذا خوردن نکنند.
- فیلسوف‌های فرد با دست راست و فیلسوف‌های زوج با دست چپ شروع کنند.

Idea: Alternating the Chopstick Pickup Order for Odd and Even Philosophers

- **Odd philosophers**: Grab the left chopstick first, then the right.
- **Even philosophers**: Grab the right chopstick first, then the left.

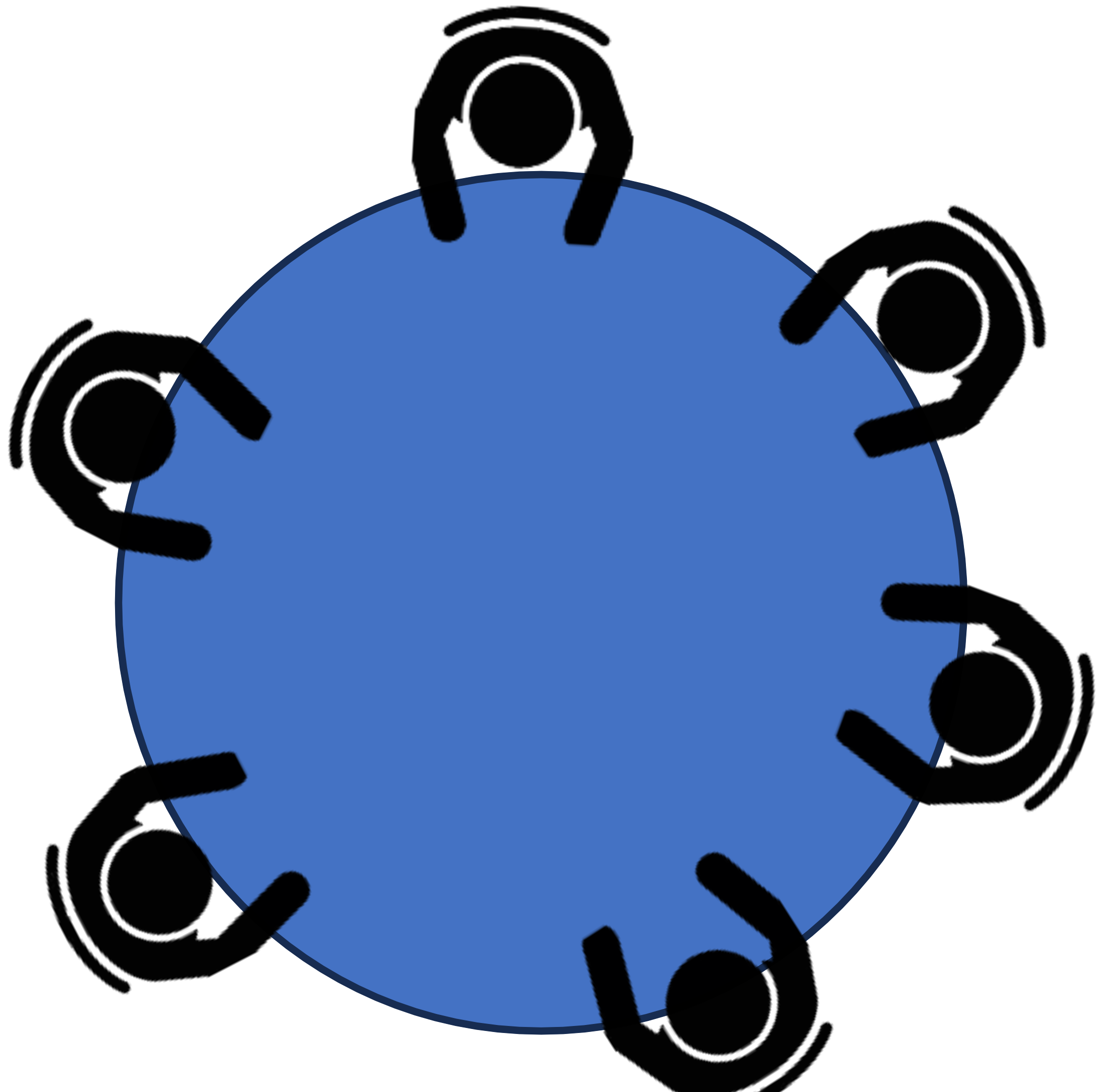
شام فیلسوف‌ها، ایده‌ی ۱



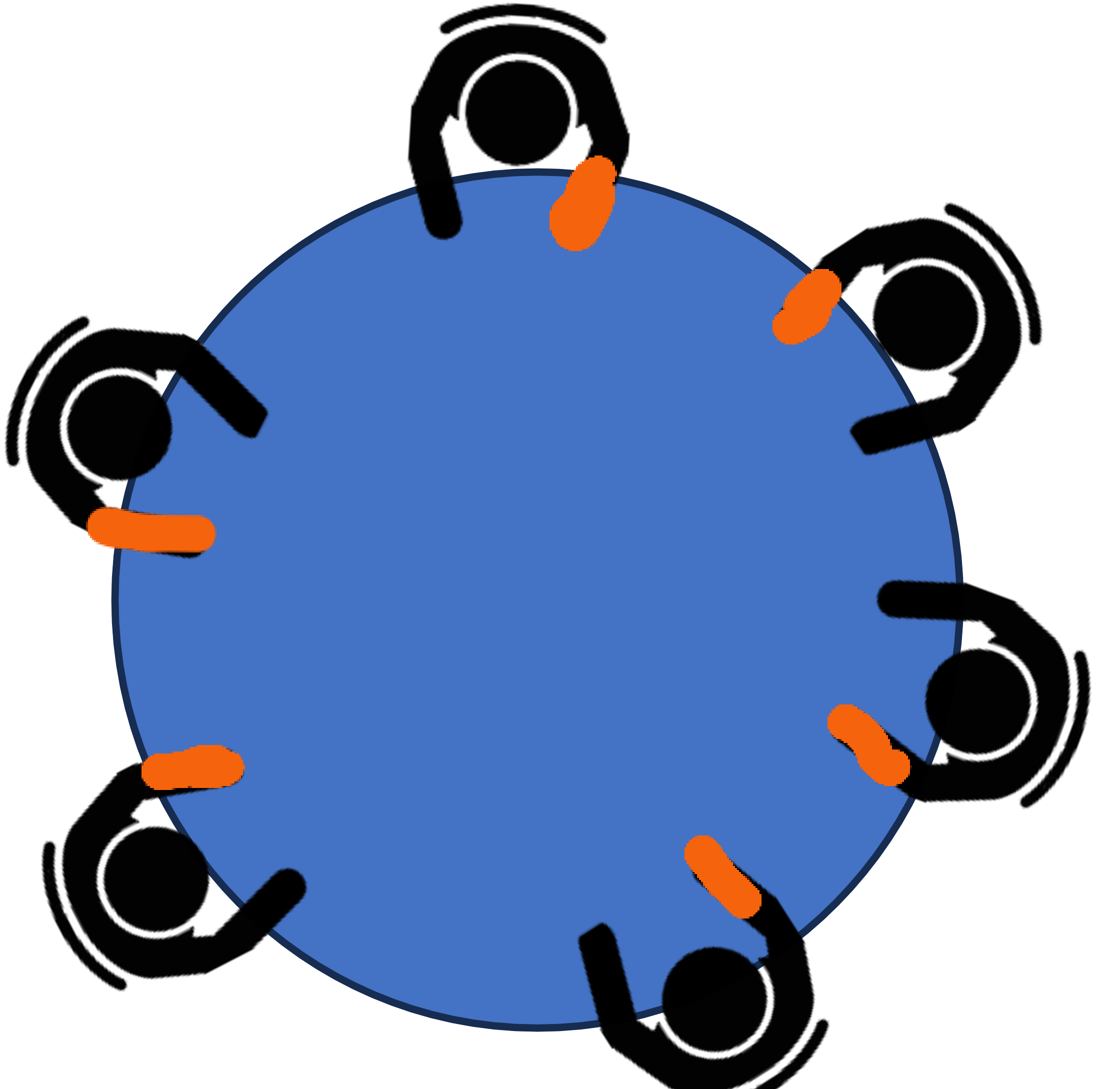
شام فیلسوف‌ها، ایده‌ی ۱

- اگر تعداد متفاوت از ۵ تا بود چطور؟
- مثلا ۶ تا فیلسوف

شام فیلسوف‌ها، با ۶ فیلسوف؟



شام فیلسوف‌ها، با ۶ فیلسوف؟



ایده‌ی ۲: چطوری نذاریم همه‌ی فیلسوف‌ها، چوب بردارند؟

- مثلا حداکثر ۴ تا فیلسوف چوبی دستشون باشه.
- خودتون فکر کنید ;)

ایده‌ی ۳: یک فیلسوف یا هر دو چوب را بردارد یا هیچ کدام

- چطوری؟
- وضعیت چوب خیلی مهم نیست، وضعیت فیلسوف مهمه.
  - *یا در حال فکر*
  - *یا منتظر غذا خوردن (گشنه)*
  - *یا در حال غذا خوردن*

Working Towards a Solution

وضعیت هر فیلسوف

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

برای متغیرهای مشترک

برای منتظر غذا ماندن فیلسوف

Working Towards a Solution

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_chopstick(i);
    take_chopstick((i+1)% N
    Eat();
    put_chopstick(i);
    put_chopstick((i+1)% N);
  }
}
```

take_chopsticks(i)

put_chopsticks(i)

Working Towards a Solution

```
#define N 5

Philosopher() {
  while(TRUE) {
    Think();
    take_chopsticks(i);
    Eat();
    put_chopsticks(i);
  }
}
```

Taking Chopsticks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
take_chopsticks(int i) {
  down(mutex);
  state [i] = HUNGRY;
  test(i);
  up(mutex);
  down(sem[i]);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
     state[LEFT] != EATING &&
     state[RIGHT] != EATING){
   state[i] = EATING;
   up(sem[i]);
 }
}
```

Putting Down Chopsticks

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

```
put_chopsticks(int i) {
  down(mutex);
  state [i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(mutex);
}
```

```
// only called with mutex set!

test(int i) {
 if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING){
   state[i] = EATING;
   up(sem[i]);
 }
}
```

مسئلەی خوانندگان و نویسندگان

Readers and Writers Problem

- Multiple readers and writers want to access a database (each one is a thread)
- Multiple readers can proceed concurrently
- Writers must synchronize with readers and other writers
  - *only one writer at a time !*
  - *when someone is writing, there must be no readers !*

Goals:

- *Maximize concurrency*
- *Prevent starvation*

دو سناریو

- First readers-writers problem
  - *no reader should wait for other readers to finish simply because a writer is waiting.*
- Second readers-writers problem
  - *once a writer is ready, that writer perform its write as soon as possible.*
  - *If a writer is waiting to access the object, no new readers may start reading.*

پیاده‌سازی ReadWriteLock

- از کتاب بخوانید... (بخش ۷٫۱٫۲)

پیاده‌سازی ReadWriteLock

- از کتاب بخوانید... (بخش ۷/۱/۲)

MONITORS

Monitors

- It is difficult to produce correct programs using semaphores
  - *Correct ordering of down and up is tricky!*
  - *Avoiding race conditions and deadlock is tricky!*
  - *Boundary conditions are tricky!*

- Can we get the compiler to generate the correct semaphore code for us?
  - *High level abstractions for synchronization?*

Monitors

- Related shared objects are collected together

- Compiler or programming convention enforces encapsulation/mutual exclusion
  - *Encapsulation*
    - *Local data variables are accessible only via the monitor's entry procedures (like methods)*
  - *Mutual exclusion*
    - *Threads must acquire the monitor's mutex lock before invoking one of its procedures*

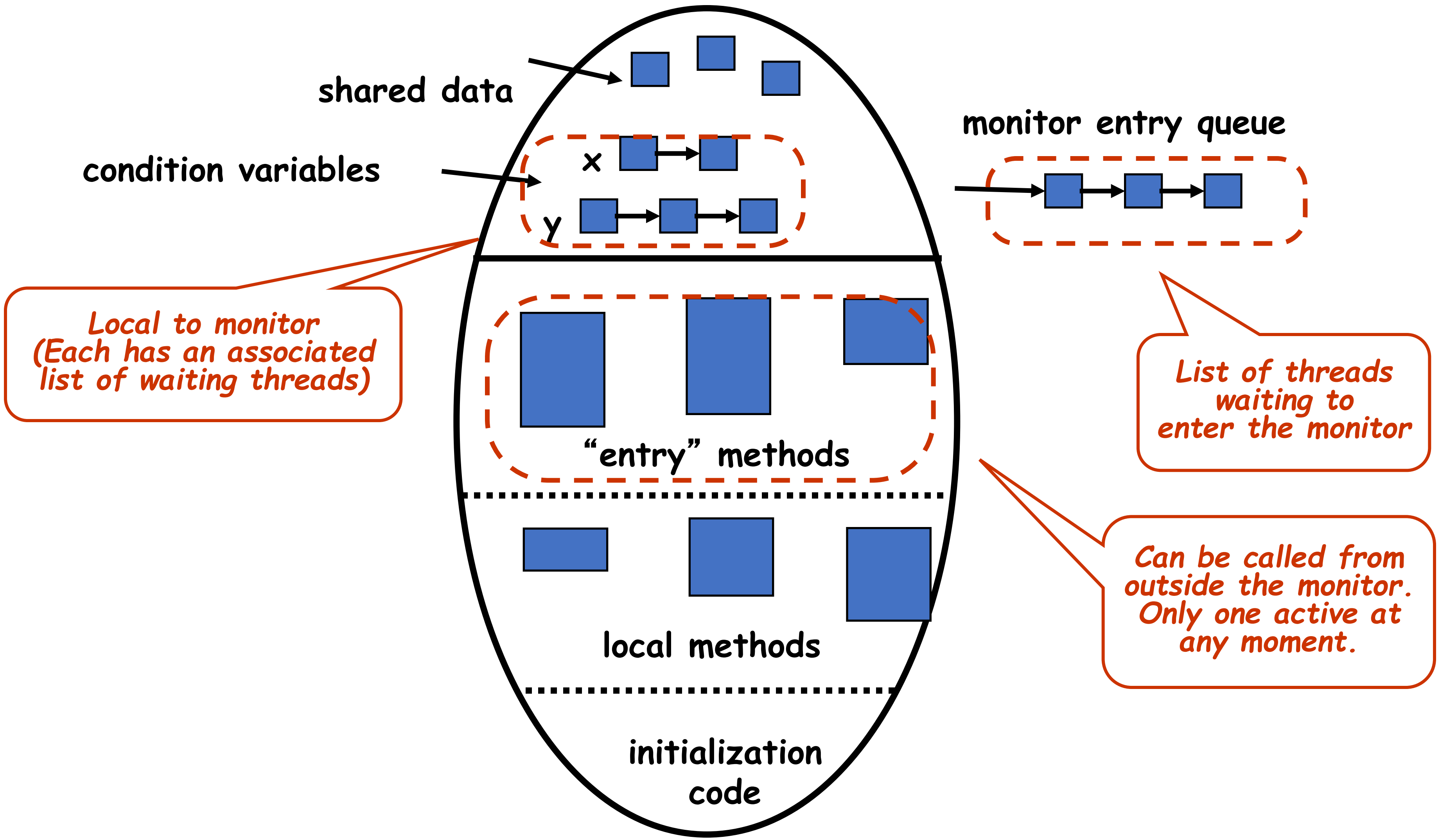
Monitors & Condition Variables

- We need two flavors of synchronization
- Mutual exclusion
  - *Only one at a time in the critical section*
  - *Handled by the monitor’s mutex*
- Condition synchronization
  - *Wait until a certain condition holds*
  - *Signal waiting threads when the condition holds*

Monitors & Condition Variables

- Condition variables (cv) for use within monitors
  - *cv.wait(mon-mutex)*
    - Thread blocked (queued) until condition holds
    - Must not block while holding mutex!
    - Monitor mutex must be released!
    - Monitor mutex need not be specified by programmer if compiler is enforcing mutual exclusion
  - *cv.signal()*
    - Signals the condition and unblocks (dequeues) a thread

Monitor Structures



Monitor Example

```
process Producer
begin
  loop
    <produce char “c”>
    BoundedBuffer.deposit(c)
  end loop
end Producer
```

```
process Consumer
begin
  loop
    BoundedBuffer.remove(c)
    <consume char “c”>
  end loop
end Consumer
```

```
monitor: BoundedBuffer
var  buffer : ...;
     nextIn, nextOut :... ;

   entry deposit(c: char)
     begin
       ...
     end

   entry remove(var c: char)
     begin
       ...
     end

end BoundedBuffer
```

Observations

- That' s much simpler than the semaphore-based solution to producer/consumer (bounded buffer)!
  - *… but where is the mutex?*
  - *… and what do the bodies of the procedures look like?*

- Here we assume the compiler is enforcing mutual exclusion among accesses to a monitor type
  - \- *like synchronized types in Java*

Monitor Example

```
monitor : BoundedBuffer
var buffer             : array[0..n-1] of char
    nextIn,nextOut     : 0..n-1 := 0
    fullCount          : 0..n   := 0
    notEmpty, notFull  : condition
```

```
  entry deposit(c:char)
  begin
    if (fullCount = n) then
        wait(notFull)
    end if

    buffer[nextIn] := c
    nextIn := nextIn+1 mod n
    fullCount := fullCount+1

    signal(notEmpty)
  end deposit
```

```
  entry remove(var c: char)
  begin
    if (fullCount = n) then
      wait(notEmpty)
    end if

    c := buffer[nextOut]
    nextOut := nextOut+1 mod n
    fullCount := fullCount-1

    signal(notFull)
  end remove
```

```
end BoundedBuffer
```

Condition Variables

Condition variables allow processes to synchronize based on some state of the monitor variables

Producer-Consumer Conditions

NotFull condition

NotEmpty condition

- Operations Wait() and Signal() allow synchronization within the monitor
- When a producer thread adds an element...
  - *A consumer may be sleeping*
  - *Need to wake the consumer... Signal*

Condition Variable Semantics

- *Only one thread at a time can execute in the monitor*
- Scenario:
  - *Thread A is executing in the monitor*
  - *Thread A does a signal waking up thread B*
  - *What happens now?*
  - *Signaling and signaled threads can not both run!*
  - *… so which one runs? which one blocks? … and how (on what queue)?*

Monitor Design Choices

- Condition variables introduce two problems for mutual exclusion
- What to do in signal: only one process can be active in the monitor at a time
  - The signaling one is already in
  - The signaled one was in when it waited and will be in again on return from wait
- What to do on wait
  - Must not block holding the mutex!
  - How do we know which mutex to release?
  - What if monitor calls are nested?

Monitor Design Choices

- A signals a condition that unblocks B
  - *Does A block until B exits the monitor?*
  - *Does B block until A exits the monitor?*
  - *Does the condition that B was waiting for still hold when B runs?*
- A signals a condition that unblocks B & C
  - *Is B unblocked, but C remains blocked?*
  - *Is C unblocked, but B remains blocked?*
  - *Are both B & C unblocked, i.e. broadcast signal*
    - … if so, they must compete for the mutex!

Option 1: Hoare Semantics

- What happens when a Signal is performed?
  - *Signaling thread (A) is suspended*
  - *Signaled thread (B) wakes up and runs immediately*
- Result:
  - *B can assume the condition it was waiting for now holds*
  - *Hoare semantics give certain strong guarantees*
- When B leaves monitor, A can run
  - *A might resume execution immediately*
  - *... or maybe another thread (C) will slip in!*

Option 2: MESA Semantics

- What happens when a Signal is performed?
  - *The signaling thread (A) continues*
  - *The signaled thread (B) waits*
  - *When A leaves the monitor, then B resumes*
- <u>Issue:</u> What happens while B is waiting?
  - *Can the condition that caused A to generate the signal be changed before B runs?*
- In MESA semantics a signal is more like a hint
  - *Requires B to recheck the condition on which it waited to see if it can proceed or must wait some*

Example Use of Hoare Semantics

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     if cntFull == N
       notFull.Wait()
     endIf
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```

Hoare Semantics

Example Use of Mesa Semantics

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     while cntFull == N
       notFull.Wait()
     endWhile
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```

MESA Semantics

Example Use of Hoare Semantics

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     ...

  entry remove()
     if cntFull == 0
       notEmpty.Wait()
     endIf
     c = buffer[nextOut]
     nextOut = (nextOut+1) mod N
     cntFull = cntFull - 1
     notFull.Signal()
   endEntry

endMonitor
```

Hoare Semantics

Example Use of Mesa Semantics

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     ...

  entry remove()
     while cntFull == 0
       notEmpty.Wait()
     endWhile
     c = buffer[nextOut]
     nextOut = (nextOut+1) mod N
     cntFull = cntFull - 1
     notFull.Signal()
   endEntry

endMonitor
```

MESA Semantics

Monitors in Blitz

- They are not implemented by the compiler
  - *The monitor lock is managed explicitly in the program*
  - *The wait call on condition variables takes the monitor lock as a parameter*
- They have MESA semantics
  - *When a waiting thread is awoken, you can’t assume that the condition it was waiting for still holds, even if it held when signal was called!*

Implementing Hoare Semantics

- Thread A holds the monitor lock
- Thread A signals a condition that thread B was waiting on
- Thread B is moved back to the ready queue?
  - *B should run immediately!*
  - *The monitor lock must be passed from A to B immediately*
  - *Thread A must be suspended*
- When B finishes it releases the monitor lock
  - *A is blocked, waiting to re-aquire the lock*
  - *A must re-acquire the lock eventually, but perhaps not immediately*

Implementing Hoare Semantics

- The challenge:
  - *Possession of the monitor lock must be passed directly from A to B and then eventually back to A*

Implementing Hoare Semantics

- Recommendation for Project 4 implementation:
  - - *Do not modify the mutex methods provided, because future code will use them*
  - - *Create new classes:*
    - MonitorLock -- similar to Mutex
    - HoareCondition -- similar to Condition

MESSAGE PASSING

Message Passing

- Interprocess Communication
  - *Via shared memory*
  - *Across machine boundaries*
- Message passing can be used for synchronization or general communication
- Processes use send and receive primitives
  - *receive can block (like waiting on a Semaphore)*
  - *send unblocks a process blocked on receive (just as a signal unblocks a waiting process)*

Message Passing Example

- **Producer-consumer example:**
  - *After producing, the producer sends the data to consumer in a message*
  - *The system buffers messages (kept in order)*
  - *The producer can out-run the consumer*
- How does the producer avoid overflowing the buffer?
  - *The consumer sends empty messages to the producer*
  - *The producer blocks waiting for empty messages*
  - *The consumer starts by sending N empty messages*
    - N is based on the buffer size

Message Passing Example

```
const N = 100             -- Size of message buffer
var em: char
for i = 1 to N            -- Get things started by
  Send (producer, &em)    --    sending N empty messages
endFor
```

```
thread consumer
  var c, em: char
  while true
    Receive(producer, &c)    -- Wait for a char
    Send(producer, &em)      -- Send empty message back
    // Consume char...
  endWhile
end
```

Message Passing Example

```
thread producer
  var c, em: char
  while true
    // Produce char c...
    Receive(consumer, &em)    -- Wait for an empty msg
    Send(consumer, &c)        -- Send c to consumer
  endWhile
end
```

Buffering Design Choices

- **Option 1:** **Mailboxes**
  - *System maintains a buffer of sent, but not yet received, messages*
  - *Must specify the size of the mailbox ahead of time*
  - *Sender will be blocked if the buffer is full*
  - *Receiver will be blocked if the buffer is empty*

Buffering Design Choices

- **Option 2: No buffering**
  - *If Send happens first, the sending thread blocks*
  - *If Receive happens first, the receiving thread blocks*
  - *Sender and receiver must Rendezvous (ie. meet)*
  - *Both threads are ready for the transfer*
  - *The data is copied / transmitted*
  - *Both threads are then allowed to proceed*

THE SLEEPING جامانده: BARBER PROBLEM

The Sleeping Barber Problem

The Sleeping Barber Problem

- *Barber:*
  - *While there are people waiting for a hair cut, put one in the barber chair, and cut their hair*
  - *When done, move to the next customer*
  - *Else go to sleep, until someone comes in*

- *Customer:*
  - *If 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 chair*
  - *If all chairs are all full, leave the barbershop*

Designing a Solution

- How will we model the barber and customers?
- What state variables do we need?
  - *.. and which ones are shared?*
  - *…. and how will we protect them?*
- How will the barber sleep?
- How will the barber wake up?
- How will customers wait?
- What problems do we need to look out for?

Recap

- What is the difference between a monitor and a semaphore?
- Why might you prefer one over the other?
- How do the wait/signal methods of a condition variable differ from the wait/signal methods of a semaphore?
- What is the difference between Hoare and Mesa semantics for condition variables?
- What implications does this difference have for code surrounding a wait() call?