Semaphores...
... or how to get “in synch”

6th November 2018 —

Synchronization

Definition

Cause to occur or operate at the same time or rate (Oxford English Dictionary)

- Events - any number of them.
- Relationship between events - before, during, after
- Requirements on them: certain order necessary

Synchronization

Example

Event A must happen before Event B

Example

Events A and B must not happen at the same time.

Before? At the same time?

- In real life we use a clock. A before B? We look at the clock.
- In computers: often not possible
  - a clock may simply be not helpful
  - we don't know when events occur

Execution Model

- How is a computer program executed?
  - Fetch - execute
  - One instruction after the other
- Synchronization is trivial:
  - Order of events: Look at program
  - Statement A before B: A will be executed first
- If it would be that easy...

Parallelism

Two ways to make it more complicated

Multiple processors

Not easy to know if a statement on one processor is executed before a statement on another

single processor is running multiple threads of execution

In general we have no control over when each thread runs. Can't tell when statements in different threads will be executed.

Parallelism

- No difference for our purposes - same issues involved
- We only know the order of instructions executed within one thread

Real World Example

Example

- Alice and Bob live in different cities
- Alice wants to know who had lunch first that day
- How can she find out?

Clocks and Computers

- Computer clocks are accurate, but with limited precision
- Nobody keeps track what events happen at what time
  - too many things
  - too fast
  - not everything is important to track

Puzzle 1

Bob is a nice guy. He is following Alice's instructions.

Can Alice guarantee that she will eat lunch before Bob?

Solution

Solution

- Alice instructs Bob not to eat lunch until she calls
- Alice makes sure not to call until she had lunch

- Seems trivial - but is a real solution
- Message Passing

Time-line

Alice	Bob
A1 Eat breakfast	B1 Eat breakfast
A2 Work	B2 ...
A3 Eat Lunch	B3 Wait for a Call
A4 Call Bob	B4 Eat Lunch

Order of events

Definition

$A1 < A2$: A1 happened before A2

We know

$$A1 < A2 < A3 < A4$$
$$B1 < B2 < B3 < B4$$

We don't know

$A1 < B1$

With message passing we know

sequentially - concurrently

We can say

- Alice and Bob ate lunch sequentially
  - because we know the order of the events
- Alice and Bob ate breakfast concurrently
  - because we don't know the order of the events

Concurrent

Definition

Two events are concurrent if we cannot tell by looking at the program which will happen first.

Puzzle 2

Concurrent Programs are often non-deterministic: we can't tell, by looking at the program, what will happen when it executes.

What is the output of this program:

Thread A

- print "yes"

Thread B

- print "no"

Non-determinism

- makes concurrent programs hard to debug
  - program may run 1000 times
  - crash at 1001st execution
- difficult to find by testing
- avoidable by careful programming

Shared Variables

- Most variables in most threads are local
  - belong to single thread
  - no other thread can access them
  - no interaction - few synchronization problems
- Some variables may be shared
  - global variables
  - global objects

Shared Variables

- Threads communicate
  - one thread writes value into common variable
    - other thread reads value from common variable
    - we can't tell if reader sees old or new value - serialization problem
  - Other issues: Concurrent writes, concurrent updates

Concurrent Writes

What happens here?

Thread A	Thread B
A1 x = 5	B1 x = 7
A2 print x	

Concurrent Writes

What happens here?

Thread A	Thread B
A1 x = 5	B1 x = 7
A2 print x	

Answer

It depends.

$A1 < A2 < B1$: output is 5, final value is 7.

Puzzle 3-5

What happens here?

Thread A	Thread B
A1 x = 5	B1 x = 7
A2 print x	

Concurrent Updates

Update:

- Reads a variable,
- changes the value and
- writes the variable.
- Let's look at an example:

Counting... (puzzle 6)

Counting things...

Thread A

1. count = count + 1

Thread B

1. count = count + 1

- What's the problem?

high level language and reality

count=count+1 is not what really happens....

what really happens...

Thread A	Thread B
A1 LOAD AX,M[count]	B1 LOAD AX,M[count]
A2 ADD AX,1	B2 ADD AX,1
A3 LD M[count],AX	B3 LD M[count],AX

Now consider $A1 < B1 < B2 < B3 < A2 < A3$

Race Condition

- A race condition occurs, when the behaviour of a program depends on the interleaving of different threads access to shared memory
- can become a bug, if the order does not correspond to the intention of the programmer
- non-deterministic behaviour, difficult to debug

Atomic Operations

- Problem exists, since incrementing count can be interrupted
- We can't tell if count=count+1 (or count++) work in a single step and cannot be interrupted or not

Definition

An operation that cannot be interrupted is called **atomic**

- There are other constructs "look" atomic
- But: if we don't know - assume they are not

Not Atomic (obviously)

- strncpy (str1, str2, n)
- something = someclass.new(...)
- std::cout << “whatever”
- ptr = malloc(size)
- ...

Real Life Updates

- Real-Live updates are more complex than incrementing
  - add element to linked list
  - transfer money between bank accounts
- Updates often require mutual exclusion

Semaphores

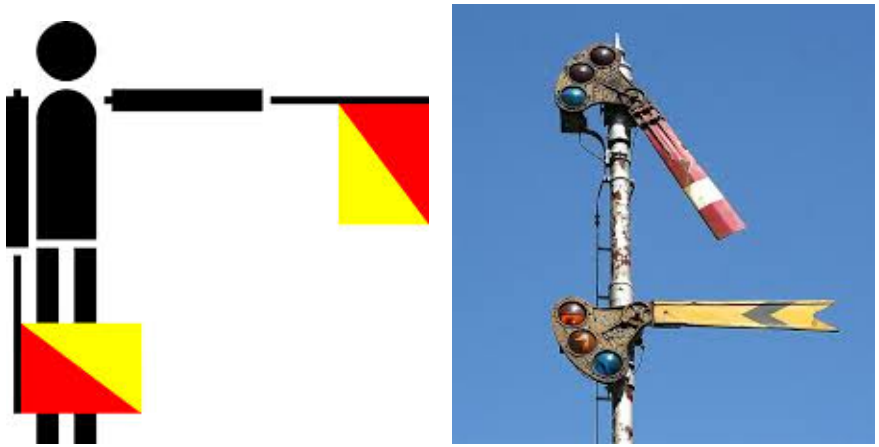


Figure: Real-life Semaphores

Semaphores

- Invented by Edsger Dijkstra 1965

Semaphores

Definition

A semaphore is like an integer, but

- you can initialize it to any integer
- you can only increment and decrement thereafter
- you cannot read the current value
- if a thread decrements the semaphore, and the value is negative, the thread is blocked
- if a thread increments the semaphore and other threads are waiting, one thread gets unblocked

Consequences

Fact

You cannot read the current value

Corollary

You don't know whether the thread will block when decrementing the semaphore

Assumption

If a thread gets woken up when the semaphore is incremented

Fact

both threads continue concurrently. We don't know which thread, if either, will continue immediately after.

Warning

pthread semaphore implementation

- Offers “sem_getvalue”
- reads value of semaphore
- OK, iff: Programmer reads manual

CAVEAT

The value of the semaphore may already have changed by the time sem_getvalue() returns.

Value

Meaning of the value

Syntax

.

Constructor

```
1 fred = Semaphore(1)
```

Semaphore Operations

```
1 fred.P()
2 fred.V()
```

Semaphore Operations

```
1 fred.increment()
2 fred.decrement()
```

Semaphore Operations

```
1 fred.increment_and_wake_a_waiting_process_if_any()
```

Reality(pthreads)

Constructor

```
1 int fred = sem_init(sem, 0, n);
```

Semaphore Operations

```
1 sem_signal(fred);
2 sem_post(fred);
```

Why Semaphores

- Why are semaphores useful?
  - we don't **need** semaphores to solve synchronization problems,
  - but there are some advantages to using them:
- Semaphores
  - impose constraints → help programmers avoid errors
  - often lead to clean and organized solutions → easy to demonstrate their correctness.
  - can be implemented efficiently on many systems → portable and usually efficient solutions

Signaling

- Simplest use for semaphores: Signal another thread
- Thread A has to finish something before thread B can continue
- Thread A has to tell thread B - signal

Signaling

Thread A

1. eat breakfast
2. work
3. eat lunch
4. sem.signal()

Thread B

1. eat breakfast
2. sem.wait()
3. eat lunch

Rendezvous

PUZZLE 7

Generalize the signal pattern so that it works both ways. Thread A has to wait for Thread B and vice versa.

Thread A

1. statement a1
2. statement a2

Thread B

1. statement b1
2. statement b2

Requirements

- guarantee that a1 happens before b2
- guarantee that b1 happens before a2
- we don't care about the order of a1 and b1

In writing your solution, be sure to specify the names and initial values of your semaphores (little hint there).

Now think about it!

Solution

Thread A

1. `statement a1`
2. `aArrived.signal()`
3. `bArrived.wait()`
4. `statement a2`

Thread B

1. `statement b1`
2. `bArrived.signal()`
3. `aArrived.wait()`
4. `statement b2`

You might have tried....

Thread A

1. `statement a1`
2. `aArrived.signal()`
3. `bArrived.wait()`
4. `statement a2`

Thread B

1. `statement b1`
2. `aArrived.wait()`
3. `bArrived.signal()`
4. `statement b2`

This solution also works, although it is probably less efficient, since it might have to switch between A and B one time more than necessary.

You might have tried....

Thread A

1. `statement a1`
2. `bArrived.wait()`
3. `aArrived.signal()`
4. `statement a2`

Thread B

1. `statement b1`
2. `aArrived.wait()`
3. `bArrived.signal()`
4. `statement b2`

How good is this?

Not good. Deadlock.

Mutex - Puzzle 8

PUZZLE

Add semaphores to the following example to enforce mutual exclusion to the shared variable count.

Thread A

1. count = count + 1

Thread B

1. count = count + 1

Mutual exclusion hint

1. Create a semaphore named mutex that is initialized to 1.

    1. A value of one means that a thread may proceed and access the shared variable;
    1. a value of zero means that it has to wait for another thread to release the mutex.

Mutual exclusion solution

Thread A

```
1  mutex.wait();
2          //critical section
3          count = count + 1;
4  mutex.signal();
```

Thread B

```
1  mutex.wait();
2          //critical section
3          count = count + 1;
4  mutex.signal();
```

Multiplex

PUZZLE 9

Generalize the Mutex so that it allows multiple threads into the critical section but enforces an upper limit (i.e. no more than n threads may enter the critical section)

- This is called a **multiplex**.
- Problem may be known from nightclubs. Bouncer enforces the constraints there.
- Do this with semaphores.

Multiplex hints

No hints this time. Too trivial.

Multiplex Solution

All Threads

```
multiplex.wait()
   //critical section
multiplex.signal()
```

Barrier

Remember the Rendezvous?

- Solution presented unfortunately does not work for more than two threads

Barrier

PUZZLE 10

Generalize the rendezvous solution to work for more than two threads. Every thread should run the following code:

```
rendezvous
critical point
```

Requirement: No thread executes critical point until after all threads have executed rendezvous

Assume n threads, stored in a variable accessible from all threads

Barrier Hint

Variables

n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)

Barrier non-solution

Code

```
1  // Barrier - rendezvous for more than one thread
2  mutex.wait();
3          count = count + 1;
4  mutex.signal();
5  if (count == n)
6          barrier.signal();
7  barrier.wait();
8  // critical point !
```

Why is this a non-solution? (Puzzle 11)

Non-Solution-explanation

- Deadlock.
- What happens after the nth thread executes the signal()?
- Does this code always produce a deadlock?

PUZZLE 12

Fix the problem!

Barrier Solution

```
1  // Barrier - rendezvous for more than one thread
2  mutex.wait();
3          count = count + 1;
4  mutex.signal();
5  if (count == n)
6          barrier.signal();
7  barrier.wait();
8  barrier.signal();
9  // critical point !
```

Barrier Solution

- one more signal is sufficient
- each threads that wakes up signals another thread.
- pattern: wait() and signal() in rapid succession
- turnstile
- Reading count out of locked section - here no problem, but generally it is not good
- Question: what state is the turnstile (semaphore barrier) in after all threads came through?

Alternative solution

- Only one thread can pass through mutex
- Only one thread can pass through turnstile

Alternative solution

- Why not put turnstile inside the mutex? Like this:

```
1  mutex.wait();
2  count = count + 1;
3  if (count == n)
4          barrier.signal();
5  barrier.wait();
6  barrier.signal();
7  mutex.signal();
8  // --- critical point ---;
```

- Opinions?

Deadlocked again

- bad idea - can cause a deadlock.
  - first thread enters the mutex
  - blocks when it reaches the turnstile.
  - mutex is locked, no other threads can enter
  - condition, count==n, will never be true
  - no one will ever unlock the turnstile

common source of deadlocks:

blocking on a semaphore while holding a mutex.

Reusable Barrier

Task

Often a set of cooperating threads will perform a series of steps in a loop and synchronize at a barrier after each step. For this application we need a reusable barrier that locks itself after all the threads have passed through.

PUZZLE 13

Rewrite the barrier solution so that after all the threads have passed through, the turnstile is locked again.

Reusable Barrier Non-Solution

```
1  mutex.wait();
2          count += 1;
3  mutex.signal();
4  if (count == n)
5          turnstile.signal();
6  turnstile.wait();
7  turnstile.signal();
8  // ---critical point----
9  mutex.wait();
10         count -= 1;
11 mutex.signal();
12 if (count == 0)
13         turnstile.wait();
```

- pretty much the same as before
- use the mutex to protect access to count
- not quite correct. Why? (Puzzle 14)

Reusable Barrier Non-Solution

```
1   mutex.wait();          count += 1;
2   mutex.signal();
3   if (count == n)
4           turnstile.signal();
5   turnstile.wait();
6   turnstile.signal();
7   // ---critical point----
8   mutex.wait();
9           count -= 1;
10  mutex.signal();
11  if (count == 0)
12          turnstile.wait();
```

- No protection from reading count
- multiple threads may call signal in line 6
- multiple threads may call wait in line 12
- deadlock!

Puzzle

PUZZLE 15

Fix this!

Reusable Barrier Non-Solution 2

```
1           mutex.wait();
2                   count += 1;
3                   if (count == n)
4                           turnstile.signal();
5           mutex.signal();
6           turnstile.wait();
7           turnstile.signal();
8           // ---- critical point ----
9           mutex.wait();
10                  count -= 1;
11                  if (count == 0) {
12                          mutex.signal();
13                          turnstile.wait();
14                  }
15                  else
16                          mutex.signal();
```

- fixes previous error
- subtle problem remains
- remember: when finished, thread will go back the the rendezvous-code and start again
- What's the problem? (Puzzle 16)

Reusable Barrier

- thread can
  - pass through second mutex
  - loop around and
  - pass through first mutex - getting ahead of all other threads
- Solution: use two turnstiles

Reusable Barrier - Two Turnstiles

Variables

turnstile = Semaphore(0)
turnstile2 = Semaphore(1)
mutex = Semaphore(1)

- All arrive at the first: we lock the second and open the first
- All arrive at the second: we lock the first and open the second

Reusable Barrier Solution

```
1   mutex.wait();
2           count += 1;
3           if (count == n) {
4                   turnstile2.wait(); // lock the second
5                   turnstile.signal(); // unlock the first
6           }
7   mutex.signal();
8   turnstile.wait(); // first turnstile
9   turnstile.signal();
10  //--- critical point ---
11  mutex.wait():
12          count -= 1;
13          if (count == 0) {
14                  turnstile.wait(); // lock the first
15                  turnstile2.signal();
16          } // unlock the second
17  mutex.signal();
18  turnstile2.wait(); // second turnstile
19  turnstile2.signal();
```

Preloaded Turnstile

- turnstile is versatile
- but: forces threads to go through sequentially
- maybe more context switching than necessary
- simplification: last thread preloads sufficiently many signals to let the right number of threads through
- nth thread
  - preloads one signal per thread
  - takes the last signal, so turnstile is locked again

Preloaded Turnstile

```
1  mutex.wait();
2          count += 1;
3          if (count == n) {
4                  for(i=1;i<n;i++)
5                          turnstile.signal();
6          } // unlock the first
7  mutex.signal();
8  turnstile.wait(); // first turnstile
9  // --- critical point ---
10 mutex.wait():
11         count -= 1;
12         if (count == 0) {
13                 for(i=1;i<n;i++)
14                 turnstile2.signal();
15         } // unlock the second
16 mutex.signal();
17 turnstile2.signal();
```

Queue

- Semaphores can represent queues
- initial value is 0
- usually it is not possible to signal unless a thread is waiting
- so the semaphore never is positive

Dancers

- Imagine: Threads represent 2 classes of ballroom dancers
  - leaders
  - followers
- who wait in queues before entering the dance floor
- When a leader arrives, it checks if there is a follower waiting
  - if so - dance
  - if not - wait
- same stuff is done by followers

Dancers

PUZZLE 17

Write code for leaders and followers

Dancers - Hint

```
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
```

- `leaderQueue` is the queue where leaders wait
- and `followerQueue` is the queue where followers wait.

Queue Solution

```
1  // leader
2  followerQueue.signal();
3  leaderQueue.wait();
4  dance();
5  // follower
6  leaderQueue.signal();
7  followerQueue.wait();
8  dance();
```

- Almost trivial - a rendezvous
- Each leader signals one follower
- Each follower signals one leader
  - so they proceed in pairs
- or.... do they? Do you see any issues here?

Queue Issues

```
1  // leader
2  followerQueue.signal();
3  leaderQueue.wait();
4  dance();
5  // follower
6  leaderQueue.signal();
7  followerQueue.wait();
8  dance();
```

- It is not clear whether they actually proceed in pairs
- any number of threads my accumulate before dance()
- any number of leaders may dance before any followers do
- this may be an issue (depending on the semantics of dance())

Exclusive Queue

- Let's make it more interesting:
- additional constraint:
  - each leader can invoke `dance` concurrently with only one follower
  - and vice versa.
- In other words, you got to dance with the one that you met at the dance-floor

Puzzle 18

write a solution to this "exclusive queue" problem.

Exclusive Queue - Hint

Here are the variables I used in my solution:

```
leaders = followers = 0
mutex = Semaphore(1)
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
rendezvous = Semaphore(0)
```

- <1|handout:0>`leaders` and `followers`: counters keeping track of the number of dancers of each kinds that are waiting
- <2|handout:0>The `mutex` guarantees exclusive access to the counters.
- <3|handout:0>`leaderQueue` and `followerQueue` are the queues where dancers wait.

Exclusive Queue - Leaders

```
1  mutex.wait();
2  if (followers > 0) {
3          followers--;
4          followerQueue.signal();
5  } else {
6          leaders++;
7          mutex.signal();
8          leaderQueue.wait();
9  }
10 dance();
11 rendezvous.wait();
12 mutex.signal()
```

Exclusive Queue - Followers

```
1  mutex.wait();
2  if (leaders > 0) {
3          leaders--;
4          leaderQueue.signal();}
5  else {
6          followers++;
7          mutex.signal();
8          followerQueue.wait();
9  }
10 dance();
11 rendezvous.signal();
```

FIFO-Queue

- Multiple threads waiting in a queue - no way to tell which thread will be woken
- some implementations require particular order
- semantics of semaphores don't guarantee any order
- but: we can build the order we want

PUZZLE 18

use semaphores to build a first-in-first-out queue.
No assumptions on behaviour of semaphore-signals allowed!

FIFO-Queue Hint

- use one semaphore per thread
- organize semaphores in a queue
- add semaphore to queue when waiting in the FIFO-Queue
- remove semaphore from queue when signaling
- queue supports: queue.add() and queue.remove()

FIFO-Queue Solution

```
1   // global variables
2           queue = Queue();
3           mutex = semaphore(1);
4   // thread-local variables
5           mySem = Semaphore(0);
6   // wait
7           mutex.wait();
8                   queue.add(mySem);
9           mutex.signal();
10          mySem.wait();
11  // signal
12          mutex.wait()
13                  sem = queue.remove();
14          mutex.signal;
15          sem.signal();
```

```
1   check if we should actually add ourselves to the queue missing
```

Producer-consumer

- Often division of labour between threads
- common pattern: some threads are producers, some consumers
- producers create some items and add them to a common data-structure
- consumers take items form the common data-structure and process them

real life example

- Example: Event-driven programs
  - events: mouse-click, key-press, network packet arrives, disk block is read
  - on events, a producer thread creates event-packet and adds it to an event-buffer
  - consumer-threads take packets from buffer to process them
- Synchronization constraints
  - need exclusive access to buffer
  - consumer threads have to wait if buffer is empty

Base Code

Producer

```
1 while (1) {
2         event = waitForEvent();
3         buffer.add(event);
4 }
```

Consumer

```
1 while (1) {
2         event = buffer.get();
3         event.process();
4 }
```

Producer Consumer

Puzzle 18

Add synchronization statements to the producer and consumer code to enforce the synchronization constraints.

Produce Consumer Hints

Here are the variables you might want to use:

```
1  mutex = Semaphore(1)
2  items = Semaphore(0)
3  local event
```

- `mutex` provides exclusive access to the buffer
- When `items` is positive, it indicates the number of items in the buffer.
- When it is negative, it indicates the number of consumer threads in queue.
- `event` is a **local variable**, which in this context means that each thread has its own version.

Producer-consumer solution

Producer

```
1  while (1) {
2          event = waitForEvent();
3          mutex.wait();
4                  buffer.add(event);
5                  items.signal();
6          mutex.signal();
7  }
```

Consumer

Consumer

```
1  while (1) {
2          items.wait();
3          mutex.wait();
4                  event = buffer.get();
5          mutex.signal();
6          event.process();
7  }
```

Producer

Producer

```
1 while (1) {
2         event = waitForEvent();
3         mutex.wait();
4                 buffer.add(event);
5                 items.signal();
6         mutex.signal();
7 }
```

Hmmm...

Anything we could do better here?

Better Producer

```
1  while (1) {
2          event = waitForEvent();
3          mutex.wait();
4                  buffer.add(event);
5          mutex.signal();
6          items.signal();
7  }
```

Consumer Issue

Consumer

```
1  while (1) {
2          items.wait();
3          mutex.wait();
4                  event = buffer.get();
5          mutex.signal();
6          event.process();
7  }
```

Hmmm...

We said, items keeps track of the number of items in the buffer. Is this the case?

Let's try to fix this

Consumer

```
1  while (1) {
2          mutex.wait();
3                  items.wait();
4                  event = buffer.get();
5          mutex.signal();
6          event.process();
7  }
```

Hmmm...

Is this a good idea?

Deadlock

Can cause a deadlock:

- Buffer is empty
  - consumer arrives
  - gets the mutex
  - blocks on `items`.
- producer arrives
  - blocks on `mutex`

... and the system comes to a grinding halt.

Finite Buffer

- So far: buffer had virtually infinite size
- often: not true
  - disk requests, network packets: some upper limit exists
- Assume: we know size of buffer: `bufferSize`
- Semaphore keeps track of number of items

Finite Buffer

- can we write:

```
1  if items >= bufferSize:
2  block()
```

Finite Buffer

- No...
- Can't read value of semaphore
- only signal and wait are allowed!

PUZZLE 19

write producer-consumer code that handles the finite-buffer constraint.

Finite Buffer Hint

- Add a second semaphore to keep track of the number of available spaces in the buffer.

```
1  mutex = Semaphore(1)
2  items = Semaphore(0)
3  spaces = Semaphore(buffer.size())
```

- consumer removes an item:signal spaces
- producer arrives: decrement spaces
  - might block until the next consumer signals

Finite Buffer Solution

```
1   //Consumer
2   items.wait();
3   mutex.wait();
4           event = buffer.get();
5   mutex.signal();
6   spaces.signal();
7   event.process();
8   //Producer
9   event = waitForEvent();
10  spaces.wait();
11  mutex.wait();
12          buffer.add(event);
13  mutex.signal();
14  items.signal();
```

Readers-Writers Problem

- concurrent modification of common data
  - data-structure, file-system, database
- while modifications are done, reading should also be forbidden
- asymmetric solution:
  - readers and writers are different

Readers-Writers Constraints

Constraints

1. Any number of readers can be in the critical section simultaneously.
2. Writers must have exclusive access to the critical section.

PUZZLE 20

Puzzle: Use semaphores to enforce these constraints, while allowing readers and writers to access the data structure, and avoiding the possibility of deadlock.

Readers-Writers Hints

Here is a set of variables that is sufficient to solve the problem.

```
1  int readers = 0
2  mutex = Semaphore(1)
3  roomEmpty = Semaphore(1)
```

- `readers` keeps track of how many readers are in the room
- `mutex` protects the shared counter
- `roomEmpty` is 1 if there are no threads (readers or writers) in the critical section

Readers-writers solution

Writers-code simple:

```
1 roomEmpty.wait();
2         //critical section for writers
3 roomEmpty.signal();
```

Can the writer be sure the room is empty when he exits?

Readers-writers solution

Ideas:

- First reader checks if room empty
- if so, he proceeds and ensures that writers are barred
- subsequent readers can enter
- last one ensures that writers can proceed

Readers-writers solution

```
1  mutex.wait();
2          readers += 1;
3          if (readers == 1)
4                  roomEmpty.wait();      // first in locks
5  mutex.signal();
6  # critical section for readers
7  mutex.wait();
8          readers -= 1;
9          if (readers == 0)
10                 roomEmpty.signal();    // last out unlocks
11 mutex.signal();
```

Light-switch

- patterns similar to this reader code are common
  - first thread into a section locks a semaphore (or queues)
  - last one out unlocks it

The name of the pattern is **Light-switch**,

Light-switch Class

```
1   class Lightswitch:
2   // constructor:
3           self.counter = 0;
4           self.mutex = Semaphore(1);
5   // lock
6   void lock(self, semaphore){
7           self.mutex.wait();
8                   self.counter++;
9                   if (self.counter == 1)
10                          semaphore.wait();
11          self.mutex.signal();
12  }
13  // unlock
14  void unlock(self, semaphore){
15          self.mutex.wait();
16                  self.counter--;
17                  if (self.counter == 0)
18                          semaphore.signal();
19          self.mutex.signal();
20  }
```

Readers-Writers with Light-switches

```
1  readLightswitch = Lightswitch();
2  roomEmpty = Semaphore(1);
3  // readLightswitch is a shared Lightswitch
4  // object whose counter is initially zero.
5  // --- Writers ---
6  roomEmpty.wait();
7          //critical section for writers
8  roomEmpty.signal();
9  // --- Readers ---
10 readLightswitch.lock(roomEmpty)
11         // critical section for readers
12 readLightswitch.unlock(roomEmpty)
```

PUZZLE 21

There is one problem with this solution

What happens if...

- there are lots of readers
- so readers keep coming ....
- and writers keep waiting ...
- starvation

PUZZLE

PUZZLE 22

Extend this solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.

hint

- add a turnstile for the readers
- writers can lock turnstile
- writers pass through same turnstile
- but if inside the turnstile - lock `roomEmpty`
- if writer has to wait for empty room, new readers queue at the turnstile
- when last reader leaves, we know a writer will be able to enter

hint

So we will need

```
1 readSwitch = Lightswitch();
2 roomEmpty = Semaphore(1);
3 turnstile = Semaphore(1);
```

- `readSwitch` keeps track of how many readers are in the room
- it locks `roomEmpty` when the first reader enters and unlocks it when the last reader exits.
- `turnstile` is a turnstile for readers and a mutex for writers

No Starve Solution - Writers

```
1  // writers
2  turnstile.wait();
3          roomEmpty.wait();
4          // critical section for writers
5  turnstile.signal();
6  roomEmpty.signal();
```

```
1  // readers
2  turnstile.wait();
3  turnstile.signal();
4  readSwitch.lock(roomEmpty);
5          // critical section for readers
6  readSwitch.unlock(roomEmpty);
```

Priority to Writers

- Maybe good idea to give more priority to writers.
- For example: writers are making time-critical updates to a data structure
- best to minimize the number of readers that see the old data

PUZZLE

Write a solution to the readers-writers problem that gives priority to writers

Hint...

The variables we are going to use:

```
1 readSwitch = Lightswitch()
2 writeSwitch = Lightswitch()
3 noReaders = Semaphore(1)
4 noWriters = Semaphore(1)
```

Solution - Writers have priority

```
1  noReaders.wait();
2          readSwitch.lock(noWriters);
3  noReaders.signal();
4          // critical section for readers
5  readSwitch.unlock(noWriters);
```

```
1  writeSwitch.lock(noReaders);
2          noWriters.wait();
3                  // critical section for writers
4          noWriters.signal();
5  writeSwitch.unlock(noReaders);
```

No-starve Mutex

- Starvation is always an issue when using semaphores.
- Starvation may be (and usually is) unacceptable
- requirement of **bounded waiting**
  - i.e. time a thread waits has to be provably finite
- In part: responsibility for starvation at scheduler
- if a thread never gets scheduled...
- we have to make assumptions on the behaviour of the scheduler

Scheduler assumptions

strong assumption

we assume that the scheduler uses one of the many algorithms that can be proven to enforce bounded waiting.

weaker assumptions

if we cannot use the strong assumption, we have to assume weaker properties

Assumptions

Property 1

if there is only one thread that is ready to run, the scheduler has to let it run.

- very weak assumption - but will be true with any practical scheduler
- we can build a system that is provably free of starvation
- generally non-trivial to write programs that are free from starvation unless we make a stronger assumption

Assumptions

Property 2

if a thread is ready to run, then the time it waits until it runs is bounded.

- we have been assuming this property implicitly (and will continue to)
- but there may be schedulers not guaranteeing this!
- and with semaphore it gets worse
- we have to make assumptions about them

Assumptions

Property 3

if there are threads waiting on a semaphore when a thread executes `signal`, then one of the waiting threads has to be woken.

- seems trivial - but isn't (well, almost isn't)
- bars the following behaviour:
  - thread signals semaphore (while others are blocked on them)
  - same thread waits on semaphore
  - thread does not block (because the signals has not yet been consumed by another thread)

Assumptions

- Property 3 makes it possible to avoid starvation
- but even for a mutex, it is not easy
- e.g., imagine three threads running the following code:

Thread code

```
1 while (True) {
2         mutex.wait();
3                 // critical section
4         mutex.signal();
5 }
```

Assumptions

- A and B take turns, C starves
- proves that the mutex is vulnerable to starvation
- One solution is: change the implementation of the semaphore so that it guarantees a stronger property:

Property 4

if a thread is waiting at a semaphore, then the number of threads that will be woken before it is bounded.

e.g if semaphore would use a FIFO-Queue, this would be guaranteed.

Assumptions

Weak Semaphore

Semaphores that have property 3

Strong Semaphore

Semaphores that have property 4

Weak Semaphore

Vulnerable to starvation

Mutex without starvation

- Dijkstra thought: not possible to have starvation free solution with weak semaphores
- But it can be done.... (Morris, J.M., 1979)

PUZZLE 24

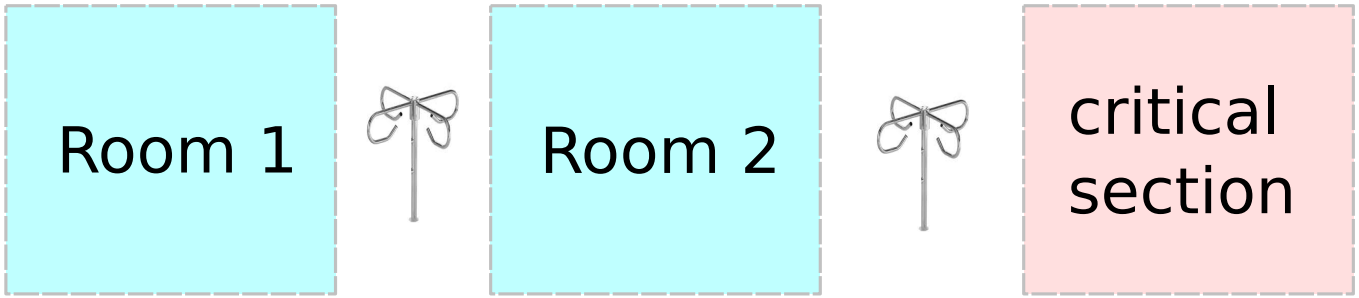
write a solution to the mutual exclusion problem using weak semaphores.

- should guarantee: once a thread arrives, there is a bound number of threads that can proceed ahead of it
- ... consider using mechanisms we know already

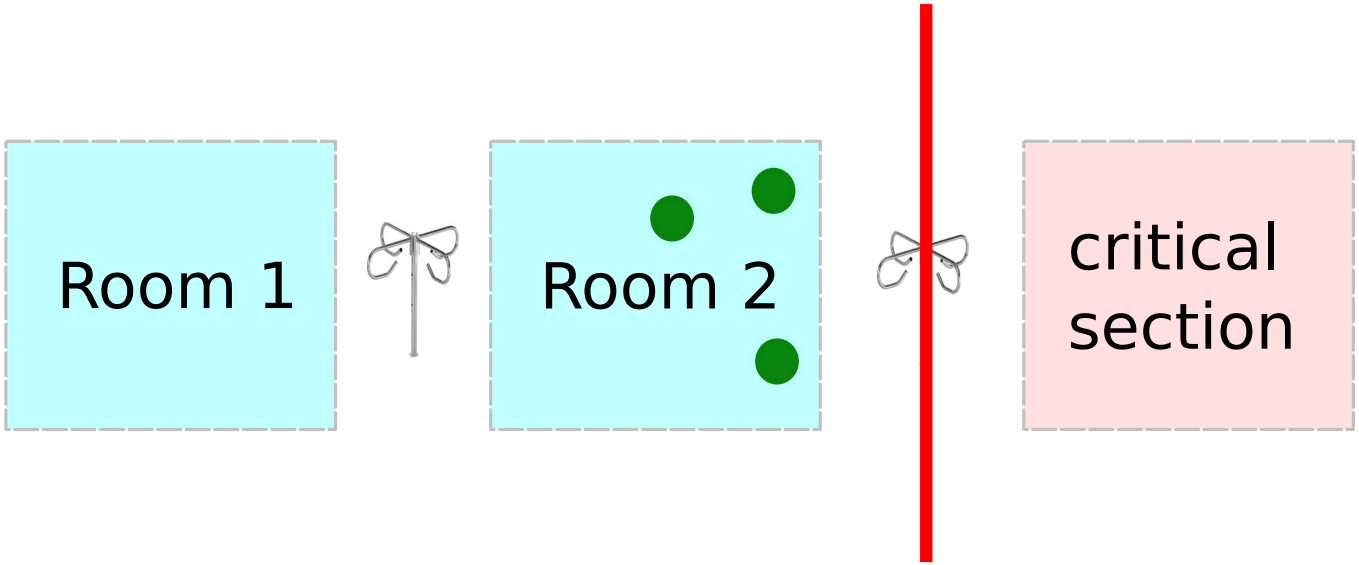
hints

- use turnstiles
- similar to reusable barrier solution
- two turnstiles - two waiting rooms
- idea: initially turnstile 1: open, turnstile 2: closed

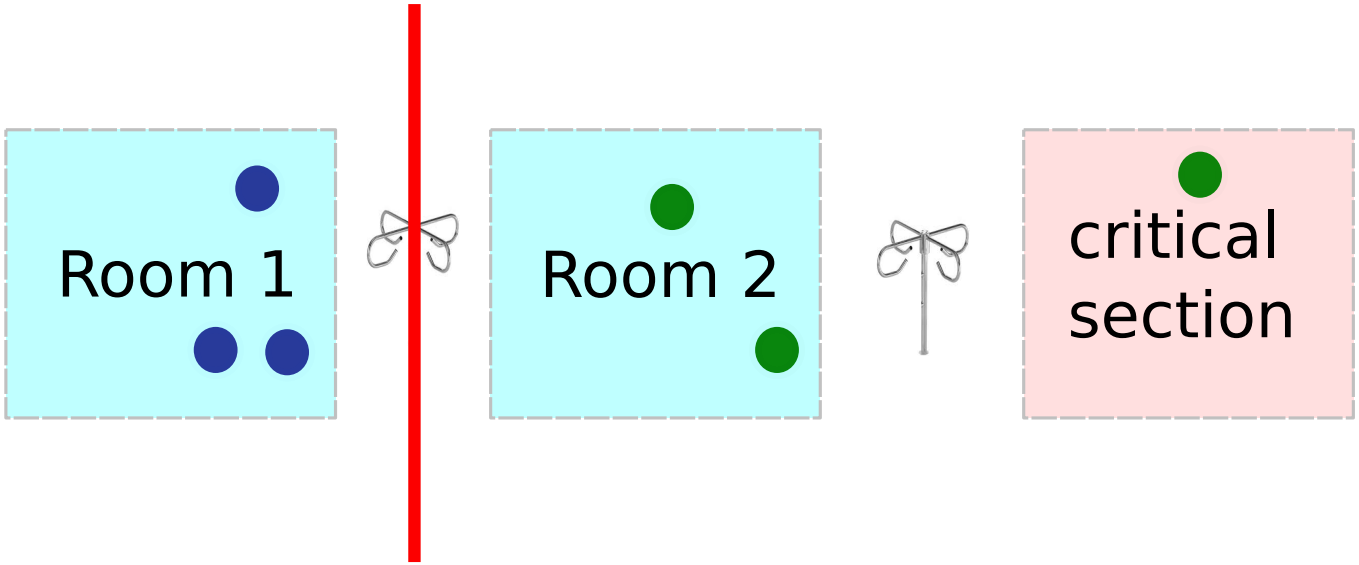
idea



idea



idea



variables

```
1  room1 = room2 = 0;
2  mutex = Semaphore(1);
3  t1 = Semaphore(1);
4  t2 = Semaphore(0);
```

- room1, room2: counters
- mutex: to protect the counters
- t1,t2: turnstiles

let's develop the solution

```
1  // entering room 1 - and counting
2  mutex.wait();
3          room1 ++;
4  mutex.signal();
5  // waiting on turnstile 1 to enter room 2
6  t1.wait();
7  // we have passed the first turnstile
8  // so we are in room 2 here!
```

let's develop the solution

```
1  // entering room 1 - and counting
2  mutex.wait();
3          room1 ++;
4  mutex.signal();
5  // waiting on turnstile 1 to enter room 2
6  t1.wait();
7  // entering room 2 - we need to adapt the counters
8  room2++;
9  mutex.wait();
10         room1--;
```

let's develop the solution

```
1   // entering room 1 - and counting
2   mutex.wait();
3           room1 ++;
4   mutex.signal();
5   // waiting on turnstile 1 to enter room 2
6   t1.wait();
7   // entering room 2 - we need to adapt the counters
8   room2++;
9   mutex.wait();
10          room1--;
11          if (room1 == 0){
12                  mutex.signal();
13                  t2.signal()M
14          } else {
15                  mutex.signal();
16                  t1.signal();
17          }
18  // end of room 2
```

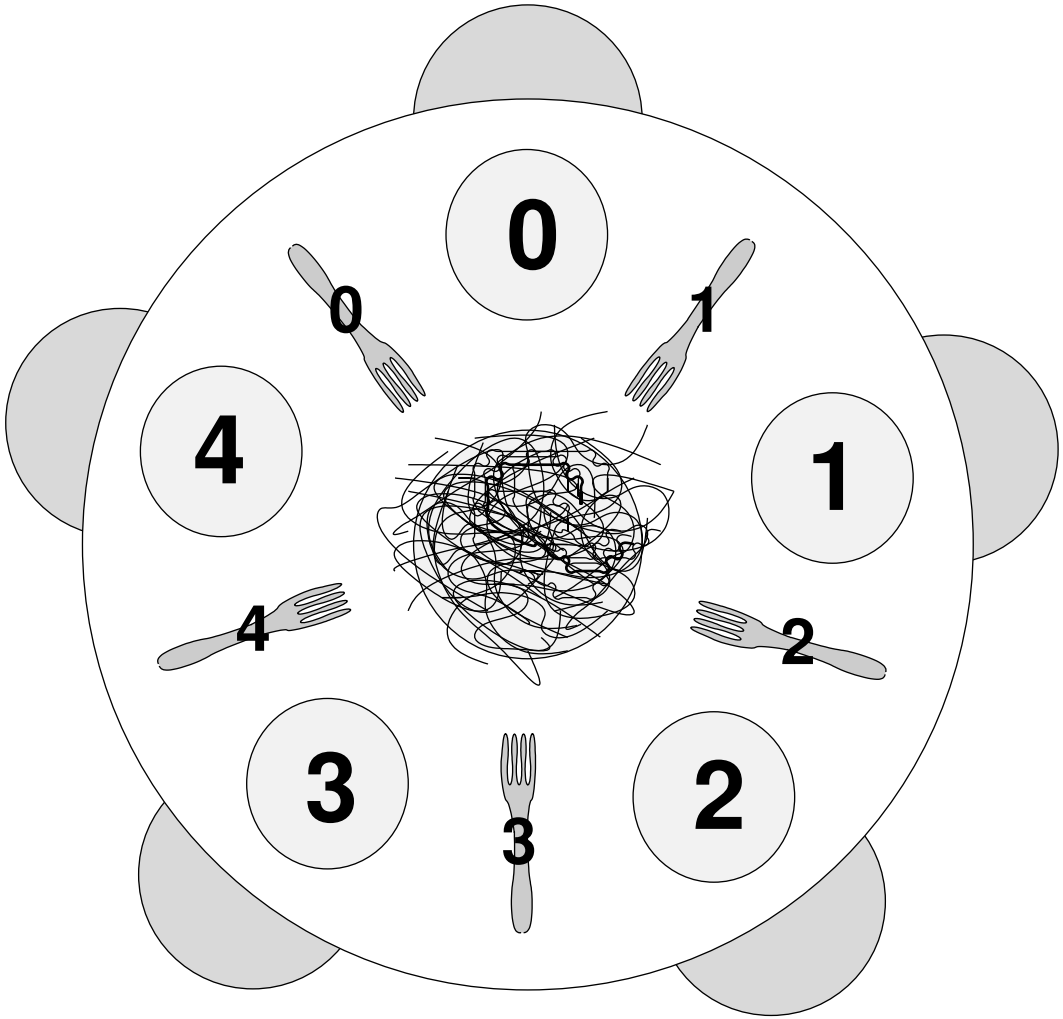
let's develop the solution

```
1   // entering room 1 - and counting
2   mutex.wait();
3           room1 ++;
4   mutex.signal();
5   // waiting on turnstile 1 to enter room 2
6   t1.wait();
7   // entering room 2 - we need to adapt the counters
8   room2++;
9   mutex.wait();
10          room1--;
11          if (room1 == 0){
12                  mutex.signal();
13                  t2.signal()M
14          } else {
15                  mutex.signal();
16                  t1.signal();
17          }
18  // end of room 2
19  t2.wait();
20  room2--;
21  // critical section
```

let's develop the solution

```
1   // entering room 1 - and counting
2   mutex.wait();
3           room1 ++;
4   mutex.signal();
5   // waiting on turnstile 1 to enter room 2
6   t1.wait();
7   // entering room 2 - we need to adapt the counters
8   room2++;
9   mutex.wait();
10          room1--;
11          if (room1 == 0){
12                  mutex.signal();
13                  t2.signal()M
14          } else {
15                  mutex.signal();
16                  t1.signal();
17          }
18  // end of room 2
19  t2.wait();
20  room2--;
21  // critical section
22  if (room2==0) t1.signal else t2.signal;
```

Dining Philosophers



Dining Philosophers

- Classical Problem by Dijkstra
- standard features:
  - table with five plates
  - five forks (or chopsticks)
  - a big bowl of spaghetti
  - five philosophers
- Philosophers represent interacting threads

Philosophers

- execute following code

```
1 while(true) {
2 think();
3 get_forks();
4 eat();
5 put_forks();
6 }
```

Dining Philosophers

- forks represent resources that the threads have to hold exclusively
- What that makes the problem interesting, unrealistic, and unsanitary, is
- the philosophers need *two* forks to eat
- a hungry philosopher might have to wait for a neighbour to put down a fork.

Dining Philosophers

- Assume
  - the philosophers have a local variable `i` identifying each philosopher with a value in (0..4)
  - forks are numbered from 0 to 4
  - Philosopher i has fork i on the right and fork $(i+1)\%5$ on the left
  - the philosophers know how to `think` and `eat`

Dining Philosophers

Our Job

write a version of `get_forks` and `put_forks` that satisfies the following constraints:

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

Dining Philosophers

- no assumptions about how long `eat` and `think` take
  - except `eat` has to terminate eventually
- to refer to forks, we can use the functions `left` and `right`:

```
1  int left(i) {return i;}
2  int right(i) {return (i + 1) % 5;}
```

Dining Philosophers

One Semaphore per fork:

```
1  forks = Semaphore[5]; // all initialised with 1
```

Here is an initial attempt at get_fork and put_fork:

```
1  void get_forks(i) {
2          fork[right(i)].wait();
3          fork[left(i)].wait();
4  }
5  void put_forks(i) {
6          fork[right(i)].signal();
7          fork[left(i)].signal();
8  }
```

Dining Philosophers

```
1  void get_forks(i) {
2          fork[right(i)].wait();
3          fork[left(i)].wait();
4  }
5  void put_forks(i) {
6          fork[right(i)].signal();
7          fork[left(i)].signal();
8  }
```

PUZZLE 25

what's wrong?

Dining Philosophers - Deadlock

```
1  void get_forks(i) {
2          fork[right(i)].wait();
3          fork[left(i)].wait();
4  }
5  void put_forks(i) {
6          fork[right(i)].signal();
7          fork[left(i)].signal();
8  }
```

Deadlock

Possible that all philosophers execute line 2 and get interrupted

Dining Philosophers - Deadlock

PUZZLE

think about solutions to this problem that prevent deadlock

Hint

small changes prevent deadlock.

Hint

- If only four philosophers are allowed at the table at a time, deadlock is impossible.

PUZZLE

- convince yourself that this claim is true
- write code that limits the number of philosophers at the table.

Claim

- only four philosophers at the table
- in the worst case each one picks up a fork
- there is a fork left on the table
- that fork has two neighbours, each of which is holding another fork
- either of these neighbours can pick up the remaining fork and eat.

Solution

```
1  footman = Semaphore(4);
2  void get_forks(i) {
3          footman.wait();
4          fork[right(i)].wait();
5          fork[left(i)].wait();
6  }
7  void put_forks(i) {
8          fork[right(i)].signal();
9          fork[left(i)].signal();
10         footman.signal();
11 }
```

Starvation

- Can starvation happen?
- Not, if footman has property 4 (is a strong semaphore)
  - we assume strong semaphores from now on

Alternatives

- by controlling the number of philosophers, we can avoid deadlock.
- Alternative: change the order in which the philosophers pick up forks.
- So far the philosophers are “righties”: they pick up the right fork first.
- But what happens if Philosopher 0 is a lefty?

PUZZLE

prove that if there is at least one lefty and at least one righty, then deadlock is not possible.

Proof

- deadlock can only occur when all 5 philosophers are holding one fork and waiting, forever, for the other
- Otherwise, one of them could get both forks, eat, and leave.
- proof by contradiction
  - assume that deadlock is possible
  - choose one of the supposedly deadlocked philosophers
  - If she's a lefty/righty
    - you can prove that the philosophers are all lefties/righties
    - contradiction
    - therefore, deadlock is not possible.

Tanenbaum's Solution

- Each philosopher has
  - a state variable per philosopher indicating whether the philosopher is
    - thinking
    - eating
    - or waiting to eat ("hungry")
  - and a semaphore indicating whether the philosopher can start eating

```
1  state = int[5]; // initialized to thinking
2  sem = Semaphore[5]; //initialized to 0;
3  mutex = Semaphore(1);
```

Tanenbaum's Solution

```
1   void get_fork(i) {
2           mutex.wait();
3                   state[i] = "hungry";
4                   test(i);
5           mutex.signal();
6           sem[i].wait();
7   }
8   void put_fork(i) {
9           mutex.wait()
10                  state[i] = "thinking";
11                  test(right(i));
12                  test(left(i));
13          mutex.signal()
14  }
15  void test(i) {
16          if ((state[i] == "hungry") &&
17                   (state[left (i)]  != "eating") &&
18                   (state[right (i)]  != "eating")) {
19                                  state[i] = "eating";
20                                  sem[i].signal();
21          }
22  }
```

Tanenbaum's Solution

- exclusive access guaranteed?
  - YES: thread only allowed to proceed if both forks free
- Deadlock?
  - no deadlock possible:
    - only one common semaphore: mutex.
    - no wait inside mutex
- Starvation?

PUZZLE

Check if starvation possible

Starvation

- Starvation is possible:
- thread 0 wants to eat (needs forks 0,1)
- 2 and 4 are eating (using forks 2,3,4,0)
- 1 and 3 are hungry
- 2 gets up, 1 sits down (1,2,4,0)
- then 4 gets up, 3 sits down (1,2,3,4)
- 3 gets up, 4 sits down ... (1,2,4,0)
- 1 gets up, 2 sits down ... (2,3,4,0)

Barbershop Problem

- Author: Dijkstra

Problem

A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

Barbershop Problem

- Customer threads should invoke a function named `getHairCut`.
- If a customer thread arrives when the shop is full, it can invoke `balk`, which does not return.
- Barber threads should invoke `cutHair`.
- When the barber invokes `cutHair` there should be exactly one thread invoking `getHairCut` concurrently.

Try to solve this at home...