Concurrency

CS 5007: Systems

Adrienne Slaughter, Joe Buck

Northeastern University

April 2, 2019

Review from Last Time

- Threads, Processes
- Viewing threads & processes
- POSIX and threading on UNIX (pthread)
- Using pthreads in C
- Briefly: sharing data in threads

Issues to address

Concurrency: Two things are ***concurrent*** if we cannot tell what will happen first just by looking at the code.

Synchronization: The Simple Case

So far:

Program

instruction 1

instruction 2

instruction 3

Synchronization: The Simple Case

So far:

Program

instruction 1
instruction 2
instruction 3

- Computers execute one instruction after another in sequence

Synchronization: The Simple Case

So far:

Program

instruction 1
instruction 2
instruction 3

- Computers execute one instruction after another in sequence
- Synchronization is trivial: we can tell the order of events by looking at the program.

Synchronization: The Simple Case

So far:

Program

instruction 1

instruction 2

instruction 3

- Computers execute one instruction after another in sequence
- Synchronization is trivial: we can tell the order of events by looking at the program.
- Instruction 1 comes before Instruction 2 comes before Instruction 3

Synchronization: More Complex

Processor A	Processor B
instruction 1	instruction 1
instruction 2	instruction 2
instruction 3	instruction 3

Synchronization: More Complex

Processor A	Processor B
instruction 1	instruction 1
instruction 2	instruction 2
instruction 3	instruction 3

- ***Case 1:*** The computer is parallel (has multiple processors running at the same time)

Synchronization: More Complex

Processor A	Processor B
instruction 1	instruction 1
instruction 2	instruction 2
instruction 3	instruction 3

- ***Case 1:*** The computer is parallel (has multiple processors running at the same time)
  - $\Rightarrow$ it's not easy to know if a statement on one processor is executed before a statement on another

Synchronization: *More Complex*

Processor A	Processor B
instruction 1	instruction 1
instruction 2	instruction 2
instruction 3	instruction 3

- ***Case 1:*** The computer is parallel (has multiple processors running at the same time)
  - $\Rightarrow$ it's not easy to know if a statement on one processor is executed before a statement on another
- ***Case 2:*** A single processor is running multiple threads of execution.

Synchronization: More Complex

Processor A	Processor B
instruction 1	instruction 1
instruction 2	instruction 2
instruction 3	instruction 3

- ***Case 1:*** The computer is parallel (has multiple processors running at the same time)
  - $\Rightarrow$ it's not easy to know if a statement on one processor is executed before a statement on another
- ***Case 2:*** A single processor is running multiple threads of execution.
  - If there are multiple threads, then the processor can work on one for a while, then switch to another, and so on.

Synchronization: More Complex

Processor A	Processor B
instruction 1	instruction 1
instruction 2	instruction 2
instruction 3	instruction 3

- ***Case 1:*** The computer is parallel (has multiple processors running at the same time)
  - ⇒ it's not easy to know if a statement on one processor is executed before a statement on another
- ***Case 2:*** A single processor is running multiple threads of execution.
  - If there are multiple threads, then the processor can work on one for a while, then switch to another, and so on.

In both cases: within one processor or thread, we know the order of execution, but between processors (or threads) it is impossible to tell.

Example

- You have a friend Ben.

Example

- You have a friend Ben.
- One day, you start to wonder: who ate lunch first today? You, or Ben? How do you find out?

Example

- You have a friend Ben.
- One day, you start to wonder: who ate lunch first today? You, or Ben? How do you find out?
- Call him and ask!
  - What if you had lunch starting at 11:59, and he had lunch starting at 12:01?

Example

- You have a friend Ben.
- One day, you start to wonder: who ate lunch first today? You, or Ben? How do you find out?
- Call him and ask!
  - What if you had lunch starting at 11:59, and he had lunch starting at 12:01?
- Unless you know that both of you have accurate clocks, you can't be sure who ate first.

New Goal: Guarantee that you eat before Ben

One approach is to tell Ben not to eat lunch until you call. Then, call him after you each lunch.

Your instructions:

```
Eat breakfast
Work
Eat lunch
Call Ben
```

Ben's Instructions:

```
Eat breakfast
Wait for a call
Eat lunch
```

New Goal: Guarantee that you eat before Ben

One approach is to tell Ben not to eat lunch until you call. Then, call him after you each lunch.

Your instructions:

```
Eat breakfast
Work
Eat lunch
Call Ben
```

Ben’s Instructions:

```
Eat breakfast
Wait for a call
Eat lunch
```

You and Ben ate lunch ***sequentially***: we know the order of events

New Goal: Guarantee that you eat before Ben

One approach is to tell Ben not to eat lunch until you call. Then, call him after you each lunch.

Your instructions:

```
Eat breakfast
Work
Eat lunch
Call Ben
```

Ben's Instructions:

```
Eat breakfast
Wait for a call
Eat lunch
```

You and Ben ate lunch ***sequentially***: we know the order of events

You and Ben ate breakfast ***concurrently***, because we don't know the order.

Determinism vs Nondeterminism

What gets printed?

Thread A

```
print "hello"
```

Thread B

```
print "goodbye"
```

Determinism vs Nondeterminism

What gets printed?

Thread A

```
print "hello"
```

Thread B

```
print "goodbye"
```

Non-deterministic: It's not possible to tell what will happen when a program executes simply by looking at it.

Determinism vs Nondeterminism

What gets printed?

Thread A

```
print "hello"
```

Thread B

```
print "goodbye"
```

Non-deterministic: It's not possible to tell what will happen when a program executes simply by looking at it.

(I can't demonstrate this...)

Determinism vs Nondeterminism

What gets printed?

Thread A

```
print "hello"
```

Thread B

```
print "goodbye"
```

Non-deterministic: It's not possible to tell what will happen when a program executes simply by looking at it.

(I can't demonstrate this...)

Deterministic: We know exactly how a program will behave when executed.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

int my_count = 5;

void *thread1(void *vargp)
{
Iprintf("thread1: %d\n", my_count);
Ireturn NULL;
}

void *thread2(void *vargp){
Iprintf("thread2: %d\n", my_count++);
Ireturn NULL;
}

int main()
{
    pthread_t tid;
    pthread_t tid2;
    printf("Before Thread\n");

    pthread_create(&tid, NULL, thread1, NULL);
    pthread_create(&tid2, NULL, thread2, NULL);

    pthread_join(tid, NULL);
    pthread_join(tid2, NULL);

    printf("After Thread\n");
    exit(0);
}
```

Listing 1: Sharing data via a variable

Sharing Variables via threads

One common way for threads to share data is to share variables that live outside the threads

1. One thread reads a variable that another thread has written to

Sharing Variables via threads

One common way for threads to share data is to share variables that live outside the threads

1. One thread reads a variable that another thread has written to
2. Two (or more) threads write to a single variable

Sharing Variables via threads

One common way for threads to share data is to share variables that live outside the threads

1. One thread reads a variable that another thread has written to
2. Two (or more) threads write to a single variable
3. Two (or more) threads read *and* write a single variable (update)

Sharing Variables via threads

One common way for threads to share data is to share variables that live outside the threads

1. One thread reads a variable that another thread has written to
2. Two (or more) threads write to a single variable
3. Two (or more) threads read *and* write a single variable (update)
4. Two (or more) threads reading the same variable rarely causes problems

Case 1: One thread writes; another reads

Case 1: One thread writes; another reads

- If the threads are ***unsynchronized***, we can't tell whether the reader will see the original value or the written value

Case 1: One thread writes; another reads

- If the threads are ***unsynchronized***, we can't tell whether the reader will see the original value or the written value
- One constraint we can enforce is: Reader should not read until after the writer writes.
  - This is like lunch with Ben: He's not going to eat until I tell him I ate.

Case 1: One thread writes; another reads

- If the threads are ***unsynchronized***, we can't tell whether the reader will see the original value or the written value
- One constraint we can enforce is: Reader should not read until after the writer writes.
  - This is like lunch with Ben: He's not going to eat until I tell him I ate.
- That is, synchronization by *policy* or *convention*

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5?

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
1 x = 5
2 print x
```

Thread B

```
1 x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$
- yield output 7 and final value 7?

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$
- yield output 7 and final value 7? $a1 < b1 < a2$

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$
- yield output 7 and final value 7? $a1 < b1 < a2$
- yield output 5 and final value 7?

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$
- yield output 7 and final value 7? $a1 < b1 < a2$
- yield output 5 and final value 7? $a1 < a2 < b1$

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$
- yield output 7 and final value 7? $a1 < b1 < a2$
- yield output 5 and final value 7? $a1 < a2 < b1$
- Is there a path that yields output 7 and final value 5?

Case 2: Concurrent Writes

We care about 2 things here:
What gets printed?
What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

Execution path: Order of execution

What are the possible execution paths that:

- yield output 5 and final value 5? $b1 < a1 < a2$
- yield output 7 and final value 7? $a1 < b1 < a2$
- yield output 5 and final value 7? $a1 < a2 < b1$
- Is there a path that yields output 7 and final value 5? Nope.

Case 3: Concurrent Updates

Update: Read the value of a variable, compute a new value based on the old value, and write the new value to the variable.

Thread A

```
x = x + 1
```

Thread B

```
x = x + 1
```

What could go wrong?

Concurrent Updates

What if we re-write the code such:

Thread A

```
temp = x
x = temp + 1
```

Thread B

```
temp = x
x = temp + 1
```

Concurrent Updates

What if we re-write the code such:

Thread A

```
temp = x
x = temp + 1
```

Thread B

```
temp = x
x = temp + 1
```

What happens with this order of execution:

$$a1 < b1 < b2 < a2$$

Concurrent Updates

What if we re-write the code such:

Thread A

```
temp = x
x = temp + 1
```

Thread B

```
temp = x
x = temp + 1
```

What happens with this order of execution:

$$a1 < b1 < b2 < a2$$

- Both threads *read* the same value, so they both *write* the same value...

Concurrent Updates

What if we re-write the code such:

Thread A

```
1 temp = x
2 x = temp + 1
```

Thread B

```
1 temp = x
2 x = temp + 1
```

What happens with this order of execution:

$$a1 < b1 < b2 < a2$$

- Both threads *read* the same value, so they both *write* the same value...
- Would the same thing happen if we had written `x++`?

Concurrent Updates

What if we re-write the code such:

Thread A

```
temp = x
x = temp + 1
```

Thread B

```
temp = x
x = temp + 1
```

What happens with this order of execution:

$$a1 < b1 < b2 < a2$$

- Both threads *read* the same value, so they both *write* the same value...
- Would the same thing happen if we had written `x++`?
- Maybe... depends on the computer.

Concurrent Updates

What if we re-write the code such:

Thread A

```
temp = x
x = temp + 1
```

Thread B

```
temp = x
x = temp + 1
```

What happens with this order of execution:

$$a1 < b1 < b2 < a2$$

- Both threads *read* the same value, so they both *write* the same value...
- Would the same thing happen if we had written `x++`?
- Maybe... depends on the computer.
- ***atomic***: An operation that cannot be interrupted.

Mutex

To write concurrent programs and ensure they are correct, we:

Mutex

To write concurrent programs and ensure they are correct, we:

- Assume all updates and all writes are **not** atomic

Mutex

To write concurrent programs and ensure they are correct, we:

- Assume all updates and all writes are **not** atomic
- Use synchronization to control concurrent access to shared resources.

Mutex

To write concurrent programs and ensure they are correct, we:

- Assume all updates and all writes are **not** atomic
- Use synchronization to control concurrent access to shared resources.
- “Mutual Exclusion”

Mutex

To write concurrent programs and ensure they are correct, we:

- Assume all updates and all writes are **not** atomic
- Use synchronization to control concurrent access to shared resources.
- "Mutual Exclusion"
- Guarantees that only one thread accesses a shared section of code at a time

Mutex Example

Semaphore and Mutex

How do we add a mutex to this?

Thread A

```
count = count + 1;
```

Thread B

```
count = count + 1;
```

Semaphore and Mutex

How do we add a mutex to this (semantically)?

Thread A

```
count = count + 1;
```

Thread B

```
count = count + 1;
```

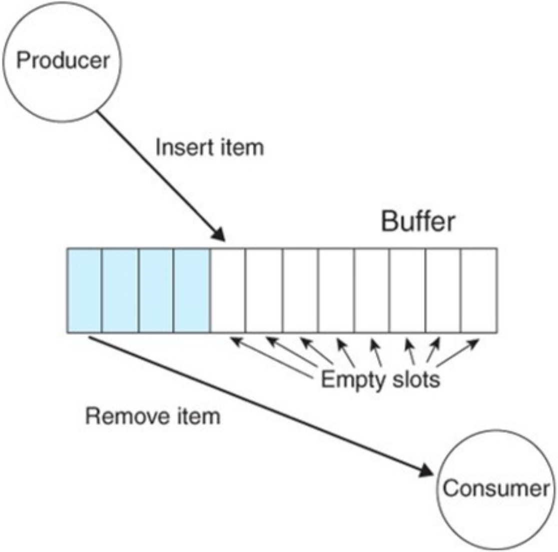
Mutexes in C

```c
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict
    attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mutexes in C

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>


int my_count = 5;

pthread_mutex_t lock;

void *thread1(void *vargp){
    pthread_mutex_lock(&lock);
    printf("thread1: %d\n", my_count);
    pthread_mutex_unlock(&lock);
    return NULL;
}

void *thread2(void *vargp){
    pthread_mutex_lock(&lock);
    printf("thread2: %d\n", my_count++);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(){
    // Initialize mutex
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    pthread_t tid;
    pthread_t tid2;
    printf("Before Thread\n");
    pthread_create(&tid, NULL, thread1, NULL);
    pthread_create(&tid2, NULL, thread2, NULL);
```

Producer-Consumer Problem



Dining Philosophers[1]

A philosopher sits at each spot.
Each philosopher follows the given process:

- Think until the left fork is available; when it is, pick it up
- Think until the right fork is available; when it is, pick it up
- When both forks are held, eat for a fixed amount of time
- Put the right fork down
- Put the left fork down
- Repeat from the beginning

[1] https://en.wikipedia.org/wiki/Dining_philosophers_problem

Dining Philosophers[2]

- Each philosopher is a thread, doing some work
- The plate of food is the data on which the philosopher is acting
- The forks are the resources the philosopher needs to do the work

[2] https://en.wikipedia.org/wiki/Dining_philosophers_problem

Semaphore

- A ***semaphore*** is a data structure that is useful for solving a variety of synchronization problems

[3] http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Semaphore

- A ***semaphore*** is a data structure that is useful for solving a variety of synchronization problems
- A semaphore is like an integer, with three differences[3]:

[3] http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Semaphore

- A ***semaphore*** is a data structure that is useful for solving a variety of synchronization problems
- A semaphore is like an integer, with three differences[3]:
  1. When you create the semaphore:
     - You can initialize its value to any integer.
     - After that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one).
     - You cannot read the current value of the semaphore.

[3] http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Semaphore

- A ***semaphore*** is a data structure that is useful for solving a variety of synchronization problems
- A semaphore is like an integer, with three differences[3]:
  1. When you create the semaphore:
     - You can initialize its value to any integer.
     - After that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one).
     - You cannot read the current value of the semaphore.
  2. When a thread decrements the semaphore:
     - If the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

[3] http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Semaphore

- A ***semaphore*** is a data structure that is useful for solving a variety of synchronization problems
- A semaphore is like an integer, with three differences[3]:
  1. When you create the semaphore:
     - You can initialize its value to any integer.
     - After that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one).
     - You cannot read the current value of the semaphore.
  2. When a thread decrements the semaphore:
     - If the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.
  3. When a thread increments the semaphore:
     - If there are other threads waiting, one of the waiting threads gets unblocked

[3] `http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf`

Semaphore: Implementation

Semaphor operations

- `createSemaphore()`

Semaphore: Implementation

Semaphor operations

- `createSemaphore()`
- `increment()` `signal()` `V()`

Semaphore: Implementation

Semaphor operations

- `createSemaphore()`
- `increment()` `signal()` `V()`
- `decrement()` `wait()` `P()`

Semaphore: Implementation

Semaphor operations

- `createSemaphore()`
- `increment()` `signal()` `V()`
- `decrement()` `wait()` `P()`
- `destroySemaphor()`

Why Semaphores?

- Semaphores impose deliberate constraints that help programmers avoid errors.

Why Semaphores?

- Semaphores impose deliberate constraints that help programmers avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.

Why Semaphores?

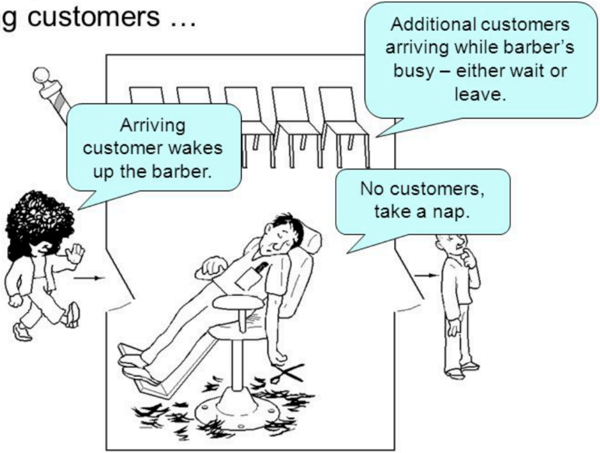
- Semaphores impose deliberate constraints that help programmers avoid errors.
- Solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness.
- Semaphores can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient.

Using Semaphores for mutual exclusion

In order for a thread to access a shared variable, it has to "get" the mutex; when it is done, it "releases" the mutex. Only one thread can hold the mutex at a time.

Sleeping Barber

One barber, one barber chair and *n* chairs for waiting customers …



Sleeping Barber

We have a barber that works when there are customers, and sleeps when there are none.

- We have a hypothetical barber shop with one barber.
- The barber has one barber's chair in a cutting room and a waiting room containing a number of chairs in it.
- When the barber finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting.
- If there are, he brings one of them back to the chairs and cut their hair.
- If there are none, he returns to the chair and sleeps in it.
- Each customer, when they arrive, looks to see what the barber is doing.
- If the barber is sleeping, the customer wakes him up and sits in the cutting room chair.
- If the barber is cutting hair, the customer stays in the waiting room.
- If there is a free chair in the waiting room, the customer sits in it and waits their turn.

Sleeping Barber[4]

- The barber is the thread that does work.

[4] https://en.wikipedia.org/wiki/Dining_philosophers_problem

Sleeping Barber[4]

- The barber is the thread that does work.
- The customer is the data on which the philosopher is acting

[4] https://en.wikipedia.org/wiki/Dining_philosophers_problem

Sleeping Barber[4]

- The barber is the thread that does work.
- The customer is the data on which the philosopher is acting
- The waiting room is the thing that controls whether there is more data to process

[4] https://en.wikipedia.org/wiki/Dining_philosophers_problem

Sleeping Barber Solution

- Global variable: `waiting`

Sleeping Barber Solution

- Global variable: `waiting`
  - Use a mutex to control access

Sleeping Barber Solution

- Global variable: `waiting`
  - Use a mutex to control access
- Signal to the barber it can sleep

Sleeping Barber Solution

- Global variable: `waiting`
  - Use a mutex to control access
- Signal to the barber it can sleep
  - Use customer semaphor

Sleeping Barber Solution

- Global variable: `waiting`
  - Use a mutex to control access
- Signal to the barber it can sleep
  - Use customer semaphor
- Signal the customer to wakeup if all barbers are busy

Sleeping Barber Solution

- Global variable: `waiting`
  - Use a mutex to control access
- Signal to the barber it can sleep
  - Use customer semaphor
- Signal the customer to wakeup if all barbers are busy
  - Use barber semaphore

In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

- The problems are all due to the fact that all tasks take an unknown amount of time.

In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

- The problems are all due to the fact that all tasks take an unknown amount of time.
- A customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps.

In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

- The problems are all due to the fact that all tasks take an unknown amount of time.
- A customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps.
- The barber is now waiting for a customer, but the customer is waiting for the barber.

In practice, there are a number of problems that can occur that are illustrative of general scheduling problems.

- The problems are all due to the fact that all tasks take an unknown amount of time.
- A customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps.
- The barber is now waiting for a customer, but the customer is waiting for the barber.
- In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Barber; a psuedocode solution

```
# The first two are mutexes (only 0 or 1 possible)
Semaphore barberReady = 0
Semaphore accessWRSeats = 1     # if 1, the number of seats in the waiting room can be incremented or decremented
Semaphore custReady = 0         # the number of customers currently in the waiting room, ready to be served
int numberOfFreeWRSeats = N     # total number of seats in the waiting room

def Barber():
  while true:                   # Run in an infinite loop.
    wait(custReady)             # Try to acquire a customer - if none is available, go to sleep.
    wait(accessWRSeats)         # Awake - try to get access to modify # of available seats, otherwise sleep.
    numberOfFreeWRSeats += 1    # One waiting room chair becomes free.
    signal(barberReady)         # I am ready to cut.
    signal(accessWRSeats)       # Don't need the lock on the chairs anymore.
    # (Cut hair here.)

def Customer():
    wait(accessWRSeats)         # Try to get access to the waiting room chairs.
    if numberOfFreeWRSeats > 0: # If there are any free seats:
      numberOfFreeWRSeats -= 1  #   sit down in a chair
      signal(custReady)         #   notify the barber, who's waiting until there is a customer
      signal(accessWRSeats)     #   don't need to lock the chairs anymore
      wait(barberReady)         #   wait until the barber is ready
      # (Have hair cut here.)
    else:                       # otherwise, there are no free seats; tough luck --
      signal(accessWRSeats)     #   but don't forget to release the lock on the seats!
      # (Leave without a haircut.)
```

Definitions/Vocab

- ***deadlock***
- ***semaphore***
- ***mutex***
- ***concurrency***