Semaphores...

... or how to get “in synch”

6th November 2018

P. Lipp 2018

Contents

1 Preface

Background and Copyright

Synchronization

Definition 1. 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 2. Event A must happen before Event B

Example 3. 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 4. • Alice and Bob live in different cities

- Alice wants to know who had lunch first that day
- How can she find out?

Alice calls Bob
Bob says: 12:01. Alice had lunch at 11:59.

But: Are these clocks accurate
Who knows....

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

Your activity required: Puzzles...

- Go to https://sweb.student.iaik.tugraz.at/quiz/sync/
- Enter your immatrikulation number
- Ready?

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

A1 Eat breakfast

A2 Work

A3 Eat Lunch

A4 Call Bob

Bob

B1 Eat breakfast

B2 ...

B3 Wait for a Call

B4 Eat Lunch

Order of events

Definition 5. $A1$ $A2$: 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
$A4$ $B4$

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

Order
Sometimes we can tell, after the program runs, which happened first.

Order
Often we cannot tell, after the program runs, which happened first.

Order
Even if we can, there is no guarantee that we will get the same result the next time.

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"

Answer
We don't know. May be "yes no" - or "no yes".

Answer
We don't know. May be "yes no" - or "no yes". Or "yneos".

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

```
A1  x = 5
A2  print x
```

Thread B

```
B1  x = 7
```

Concurrent Writes

Answer
It depends.

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

Puzzle 3-5

What happens here?
Thread A

A1	x = 5
A2	print x

Thread B

B1	x = 7

Puzzle
What path yields output 5 and final value 5?

$B1$ $A1$ $A2$

Puzzle
What path yields output 7 and final value 7?

$A1$ $B1$ $A2$

Puzzle
Is there a path that yields output 7 and final value 5?

Is there a path that yields output 7 and final value 5?
No. Can you prove it?

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

A1 LOAD AX,M[count]

A2 ADD AX,1

A3 LD M[count],AX

Thread B

B1 LOAD AX,M[count]

B2 ADD AX,1

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 7. 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

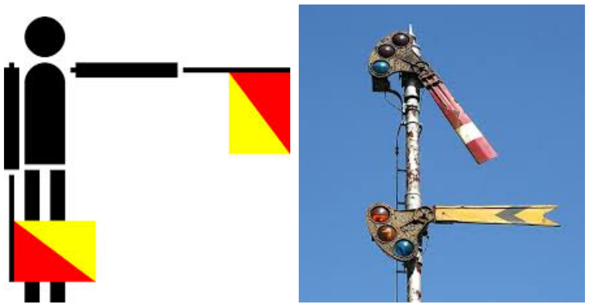


Figure 1: Real-life Semaphores

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

Semaphores
A system of sending messages by holding the arms or two flags or poles in certain positions according to an alphabetic code. for signaling, consisting of an upright with movable parts.

Semaphores

- Invented by Edsger Dijkstra 1965

Semaphores

Definition 8. 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 9. *You cannot read the current value*

Corollary 10. *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 11. *both threads continue concurrently. We don't know which thread, if either, will continue immediately after.*

Fact 12. *When you signal a semaphore, you don't necessarily know whether another thread is waiting, so the number of unblocked threads may be zero or one.*

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

- Value is positive:
  - number of threads that can decrement without blocking
- Value is negative:
  - number of threads that have blocked and are waiting.
- Value is zero
  - there are no threads waiting, but if a thread tries to decrement, it will block.

Syntax

.

Constructor

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

Semaphore Operations

```
fred.P()
fred.V()
```

Semaphore Operations

```
fred.increment()
fred.decrement()
```

Semaphore Operations

```
fred.increment_and_wake_a_waiting_process_if_any()
fred.decrement_and_block_if_the_result_is_negative()
```

We will be using here

```
fred.signal()
fred.wait()
```

Reality(pthreads)

Constructor

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

Semaphore Operations

```
sem_signal(fred);
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

2 Basic Patterns

2.1 Signaling

Signaling

Possibly the simplest use for a semaphore is signaling, which means that one thread sends a signal to another thread to indicate that something has happened.

Signaling makes it possible to guarantee that a section of code in one thread will run before a section of code in another thread; in other words, it solves the serialization problem.

Assume that we have a semaphore named sem with initial value 0, and that Threads A and B have shared access to it.

The word statement represents an arbitrary program statement. To make the example concrete, imagine that a1 reads a line from a file, and b1 displays the line on the screen. The semaphore in this program guarantees that Thread A has completed a1 before Thread B begins b1.

Here's how it works: if thread B gets to the wait statement first, it will find the initial value, zero, and it will block. Then when Thread A signals, Thread B proceeds.

Similarly, if Thread A gets to the signal first then the value of the semaphore will be incremented, and when Thread B gets to the wait, it will proceed immediately. Either way, the order of a1 and b1 is guaranteed.

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

Signaling

This use of semaphores is the basis of the names signal and wait, and in this case the names are conveniently mnemonic. Unfortunately, we will see other cases where the names are less helpful. Speaking of meaningful names, sem isn't one. When possible, it is a good idea to give a semaphore a name that indicates what it represents. In this case a name like a1Done might be good, so that a1done.signal() means "signal that a1 is done," and a1done.wait() means "wait until a1 is done."

2.2 Rendezvous

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!

Hint

The chances are good that you were able to figure out a solution, but if not, here is a hint. Create two semaphores, named aArrived and bArrived, and initialize them both to zero. As the names suggest, aArrived indicates whether Thread A has arrived at the rendezvous, and bArrived likewise.

Time to think before you turn the page

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.

Idea

If A arrives first, it waits for B. When B arrives, it wakes A and might proceed immediately to its wait in which case it blocks, allowing A to reach its signal, after which both threads can proceed. Think about the other possible paths through this code and convince yourself that in all cases neither thread can proceed until both have arrived.

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*.

Deadlock

It has a serious problem.

Assuming that A arrives first, it will block at its wait. When B arrives, it will also block, since A wasn't able to signal aArrived. At this point, neither thread can proceed, and never will. This situation is called a deadlock and, obviously, it is not a successful solution of the synchronization problem. In this case, the error is obvious, but often the possibility of deadlock is more subtle. We will see more examples later.

2.3 Mutex

Mutex

A second common use for semaphores is to enforce mutual exclusion. We have already seen one use for mutual exclusion, controlling concurrent access to shared variables. The mutex guarantees that only one thread accesses the shared variable at a time.

A mutex is like a token that passes from one thread to another, allowing one thread at a time to proceed. For example, in The Lord of the Flies a group of children use a conch as a mutex. In order to speak, you have to hold the conch. As long as only one child holds the conch, only one can speak.

Similarly, 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.

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

- Time to think before you turn the page

Mutual exclusion hint

1. Create a semaphore named mutex that is initialized to 1.
    (a) A value of one means that a thread may proceed and access the shared variable;
    (b) a value of zero means that it has to wait for another thread to release the mutex.

- Time to think before you turn the page

Mutual exclusion solution

Thread A

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

Thread B

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

Mutual exclusion solution

Since mutex is initially 1, whichever thread gets to the wait first will be able to proceed immediately. Of course, the act of waiting on the semaphore has the effect of decrementing it, so the second thread to arrive will have to wait until the first signals.

I have indented the update operation to show that it is contained within the mutex.

In this example, both threads are running the same code. This is sometimes called a symmetric solution. If the threads have to run different code, the solution is asymmetric. Symmetric solutions are often easier to generalize. In this case, the mutex solution can handle any number of concurrent threads without modification. As long as every thread waits before performing an update and signals after, then no two threads will access count concurrently.

Often the code that needs to be protected is called the critical section, I suppose because it is critically important to prevent concurrent access.

In the tradition of computer science and mixed metaphors, there are several other ways people sometimes talk about mutexes. In the metaphor we have been using so far, the mutex is a token that is passed from one thread to another.

In an alternative metaphor, we think of the critical section as a room, and only one thread is allowed to be in the room at a time. In this metaphor, mutexes are called locks, and a thread is said to lock the mutex before entering and unlock it while exiting. Occasionally, though, people mix the metaphors and talk about “getting” or “releasing” a lock, which doesn’t make much sense.

Both metaphors are potentially useful and potentially misleading. As you work on the next problem, try out both ways of thinking and see which one leads you to a solution.

2.4 Multiplex

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.
- Time to think before you turn the page

Multiplex hints

No hints this time. Too trivial.

Multiplex Solution

All Threads

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

2.5 Barrier

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

- Time to think before you turn the page

Barrier Hint

Variables

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

- count keeps track of how many threads have arrived.
- mutex provides exclusive access to count so that threads can increment it safely.
- barrier is locked (zero or negative) until all threads arrive; then it should be unlocked (1 or more).
- Time to think before you turn the page

Barrier non-solution

Code

```
// Barrier - rendezvous for more than one thread
mutex.wait();
        count = count + 1;
mutex.signal();
if (count == n)
        barrier.signal();
barrier.wait();
// critical point !
```

Since count is protected by a mutex, it counts the number of threads that pass. The first $n-1$ threads wait when they get to the barrier, which is initially locked. When the nth thread arrives, it unlocks the barrier. Puzzle: What is wrong with this solution?

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

- Time to think before you turn the page

Non-Solution-explanation

The problem is a deadlock. An an example, imagine that n = 5 and that 4 threads are waiting at the barrier. The value of the semaphore is the number of threads in queue, negated, which is -4. When the 5th thread signals the barrier, one of the waiting threads is allowed to proceed, and the semaphore is incremented to -3. But then no one signals the semaphore again and none of the other threads can pass the barrier. This is a second example of a deadlock. Puzzle: Does this code always create a deadlock? Can you find an execution path through this code that does not cause a deadlock? Puzzle: Fix the problem.

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

PUZZLE 12

Fix the problem!

- Time to think before you turn the page

Barrier Solution

The only change is another signal after waiting at the barrier. Now as each thread passes, it signals the semaphore so that the next thread can pass. This pattern, a wait and a signal in rapid succession, occurs often enough that it has a name; it's called a turnstile, because it allows one thread to pass at a time, and it can be locked to bar all threads. In its initial state (zero), the turnstile is locked. The nth thread unlocks it and then all n threads go through. It might seem dangerous to read the value of count outside the mutex. In this case it is not a problem, but in general it is probably not a good idea. We will clean this up in a few pages, but in the meantime, you might want to consider these questions: After the nth thread, what state is the turnstile in? Is there any way the barrier might be signalled more than once?

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:

```
mutex.wait();
count = count + 1;
if (count == n)
        barrier.signal();
barrier.wait();
barrier.signal();
mutex.signal();
// --- critical point ---;
```

- Opinions?
  - Time to think before you turn the page

Deadlocked again

This turns out to be a bad idea because it can cause a deadlock.

Imagine that the first thread enters the mutex and then blocks when it reaches the turnstile. Since the mutex is locked, no other threads can enter, so the condition, `count==n`, will never be true and no one will ever unlock the turnstile.

In this case the deadlock is fairly obvious, but it demonstrates a 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.

- Time to think before you turn the page

Reusable Barrier Non-Solution

```
mutex.wait();
        count += 1;
mutex.signal();
if (count == n)
        turnstile.signal();
turnstile.wait();
turnstile.signal();
// ---critical point----
mutex.wait();
        count -= 1;
mutex.signal();
if (count == 0)
        turnstile.wait();
```

- pretty much the same as before
- use the mutex to protect access to count
- *not quite correct. Why? (Puzzle 14)*
  - Time to think before you turn the page

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

```
        mutex.wait();
                count += 1;
                if (count == n)
                        turnstile.signal();
        mutex.signal();
        turnstile.wait();
        turnstile.signal();
        // ---- critical point ----
        mutex.wait();
                count -= 1;
                if (count == 0) {
                        mutex.signal();
                        turnstile.wait();
                }
                else
                        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)*

This attempt fixes the previous error, but a subtle problem remains.

In both cases the check is inside the mutex so that a thread cannot be interrupted after changing the counter and before checking it. Tragically, this code is still not correct. Remember that this barrier will be inside a loop. So, after executing the last line, each thread will go back to the rendezvous. Puzzle: Identify and fix the problem.

- Time to think before you turn the page

Reusable Barrier

As it is currently written, this code allows a precocious thread to pass through the second mutex, then loop around and pass through the first mutex and the turnstile, effectively getting ahead of the other threads by a lap.

To solve this problem we can use two turnstiles.

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

Initially the first is locked and the second is open. When all the threads arrive at the first, we lock the second and unlock the first. When all the threads arrive at the second we relock the first, which makes it safe for the threads to loop around to the beginning, and then open the second.

Reusable Barrier Solution

```
mutex.wait();
        count += 1;
        if (count == n) {
                turnstile2.wait(); // lock the second
                turnstile.signal(); // unlock the first
        }
mutex.signal();
turnstile.wait(); // first turnstile
turnstile.signal();
//--- critical point ---
mutex.wait():
        count -= 1;
        if (count == 0) {
                turnstile.wait(); // lock the first
                turnstile2.signal();
        } // unlock the second
mutex.signal();
turnstile2.wait(); // second turnstile
turnstile2.signal();
```

Comments

This solution is sometimes called a **two-phase barrier** because it forces all the threads to wait twice: once for all the threads to arrive and again for all the threads to execute the critical section.

Unfortunately, this solution is typical of most non-trivial synchronization code: it is difficult to be sure that a solution is correct. Often there is a subtle way that a particular path through the program can cause an error.

To make matters worse, testing an implementation of a solution is not much help. The error might occur very rarely because the particular path that causes it might require a spectacularly unlucky combination of circumstances. Such errors are almost impossible to reproduce and debug by conventional means.

The only alternative is to examine the code carefully and "prove" that it is correct. I put "prove" in quotation marks because I don't mean, necessarily, that you have to write a formal proof (although there are zealots who encourage such lunacy).

The kind of proof I have in mind is more informal. We can take advantage of the structure of the code, and the idioms we have developed, to assert, and then demonstrate, a number of intermediate-level claims about the program. For example:

1. Only the n th thread can lock or unlock the turnstiles.
2. Before a thread can unlock the first turnstile, it has to close the second, and vice versa; therefore it is impossible for one thread to get ahead of the others by more than one turnstile.

By finding the right kinds of statements to assert and prove, you can sometimes find a concise way to convince yourself (or a sceptical colleague) that your code is bulletproof.

Preloaded Turnstile

One nice thing about a turnstile is that it is a versatile component you can use in a variety of solutions. But one drawback is that it forces threads to go through sequentially, which may cause more context switching than necessary.

In the reusable barrier solution, we can simplify the solution if the thread that unlocks the turnstile preloads the turnstile with enough signals to let the right number of threads through.

The syntax I am using here assumes that `signal` can take a parameter that specifies the number of signals. This is a non-standard feature, but it would be easy to implement with a loop. The only thing to keep in mind is that the multiple signals are not atomic; that is, the signaling thread might be interrupted in the loop. But in this case that is not a problem.

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

```
mutex.wait();
        count += 1;
        if (count == n) {
                for(i=1;i<n;i++)
                        turnstile.signal();
        } // unlock the first
mutex.signal();
turnstile.wait(); // first turnstile
// --- critical point ---
mutex.wait():
        count -= 1;
        if (count == 0) {
                for(i=1;i<n;i++)
                turnstile2.signal();
        } // unlock the second
mutex.signal();
turnstile2.signal();
```

When the nth thread arrives, it preloads the first turnstile with one signal for each thread. When the nth thread passes the turnstile, it “takes the last token” and leaves the turnstile locked again.

The same thing happens at the second turnstile, which is unlocked when the last thread goes through the mutex.

2.6 Queue

Queue

Semaphores can also be used to represent a queue. In this case, the initial value is 0, and usually the code is written so that it is not possible to signal unless there is a thread waiting, so the value of the semaphore is never positive.

For example, imagine that threads represent ballroom dancers and that two kinds of dancers, leaders and followers, wait in two queues before entering the dance floor. When a leader arrives, it checks to see if there is a follower waiting. If so, they can both proceed. Otherwise it waits.

Similarly, when a follower arrives, it checks for a leader and either proceeds or waits, accordingly.

Puzzle: write code for leaders and followers that enforces these constraints.

- Time to think before you turn the page

Dancers - Hint

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

- `leaderQueue` is the queue where leaders wait
- and `followerQueue` is the queue where followers wait.
  - Time to think before you turn the page

Queue solution

Here is the code for leaders:

```
followerQueue.signal()
leaderQueue.wait()
dance()
```

And here is the code for followers:

```
leaderQueue.signal()
followerQueue.wait()
dance()
```

This solution is about as simple as it gets; it is just a Rendezvous. Each leader signals exactly one follower, and each follower signals one leader, so it is guaranteed that leaders and followers are allowed to proceed in pairs. But whether they actually proceed in pairs is not clear. It is possible for any number of threads to accumulate before executing `dance`, and so it is possible for any number of leaders to `dance` before any followers do. Depending on the semantics of `dance`, that behaviour may or may not be problematic.

To make things more interesting, let's add the additional constraint that each leader can invoke `dance` concurrently with only one follower, and vice versa. In other words, you got to dance with the one that brought you.

Puzzle: write a solution to this "exclusive queue" problem.

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. Time to think before you turn the page

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.
- <4|handout:0>`rendezvous` is used to check that both threads are done dancing.
- <5|handout:0>It's OK if only 2 threads are dancing at any point in time

Exclusive Queue

When a leader arrives, it gets the mutex that protects `leaders` and `followers`. If there is a follower waiting, the leader decrements `followers`, signals a follower, and then invokes `dance`, all before releasing `mutex`. That guarantees that there can be only one follower thread running `dance` concurrently.

If there are no followers waiting, the leader has to give up the mutex before waiting on `leaderQueue`.

Exclusive Queue

When a follower arrives, it checks for a waiting leader. If there is one, the follower decrements `leaders`, signals a leader, and executes `dance`, all without releasing `mutex`. Actually, in this case the follower *never* releases `mutex`; the leader does. We don't have to keep track of which thread has the mutex because we know that one of them does, and either one of them can release it. In my solution it's always the leader.

When a semaphore is used as a queue[1], I find it useful to read "wait" as "wait for this queue" and signal as "let someone from this queue go."

In this code we never signal a queue unless someone is waiting, so the values of the queue semaphores are seldom positive. It is possible, though. See if you can figure out how.

2.7 FIFO-Queue

FIFO-Queue

If there is more than one thread waiting in queue when a semaphore is signalled, there is usually no way to tell which thread will be woken. Some implementations wake threads up in a particular order, like first-in-first-out, but the semantics of semaphores don't require any particular order. Even if your environment doesn't provide first-in-first-out queuing, you can build it yourself.

[1]A semaphore used as a queue is very similar to a condition variable. The primary difference is that threads have to release the mutex explicitly before waiting, and reacquire it explicitly afterwards (but only if they need it).

Puzzle: use semaphores to build a first-in-first-out queue. Each time the FIFO is signalled, the thread at the head of the queue should proceed. If more than one thread is waiting on a semaphore, you should not make any assumptions about which thread will proceed when the semaphore is signalled.

For bonus points, your solution should define a class named `FIFO` that provides methods named `wait` and `signal`.

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

```
// global variables
        queue = Queue();
        mutex = semaphore(1);
// thread-local variables
        mySem = Semaphore(0);
// wait
        mutex.wait();
                queue.add(mySem);
        mutex.signal();
        mySem.wait();
// signal
        mutex.wait()
                sem = queue.remove();
        mutex.signal;
        sem.signal();
```

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

You can assume that there is a data structure named `Queue` that provides methods named `add` and `remove`, but you should not assume that the Queue is thread-safe; in other words, you have to enforce exclusive access to the Queue.

3 Classical Synchronization Problems

3.1 Introduction

In this chapter we examine the classical problems that appear in nearly every operating systems textbook. They are usually presented in terms of real-world problems, so that the statement of the problem is clear and so that students can bring their intuition to bear.

For the most part, though, these problems do not happen in the real world, or if they do, the real-world solutions are not much like the kind of synchronization code we are working with.

The reason we are interested in these problems is that they are analogous to common problems that operating systems (and some applications) need to solve. For each classical problem I will present the classical formulation, and also explain the analogy to the corresponding OS problem.

3.2 Producer-consumer problem

In multi-threaded programs there is often a division of labour between threads. In one common pattern, some threads are producers and some are consumers. Producers create items of some kind and add them to a data structure; consumers remove the items and process them.

Event-driven programs are a good example. An "event" is something that happens that requires the program to respond: the user presses a key or moves the mouse, a block of data arrives from the disk, a packet arrives from the network, a pending operation completes.

Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them. In this case, the consumers are called "event handlers."

There are several synchronization constraints that we need to enforce to make this system work correctly:

- While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Assume that producers perform the following operations over and over:

Producer

```
event = waitForEvent()
buffer.add(event)
```

Also, assume that consumers perform the following operations:

Consumer

```
event = buffer.get()
event.process()
```

As specified above, access to the buffer has to be exclusive, but `waitForEvent` and `event.process` can run concurrently.

Puzzle: 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:

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

Not surprisingly, `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. So far we have been assuming that all threads have access to all variables, but we will sometimes find it useful to attach a variable to each thread.

There are a number of ways this can be implemented in different environments:

- If each thread has its own run-time stack, then any variables allocated on the stack are thread-specific.
- If threads are represented as objects, we can add an attribute to each thread object.
- If threads have unique IDs, we can use the IDs as an index into an array or hash table, and store per-thread data there.

In most programs, most variables are local unless declared otherwise, but in this book most variables are shared, so we will assume that that variables are shared unless they are explicitly declared `local`.

Producer-consumer solution

Producer

```
while (1) {
        event = waitForEvent();
        mutex.wait();
                buffer.add(event);
                items.signal();
        mutex.signal();
}
```

The producer doesn't have to get exclusive access to the buffer until it gets an event. Several threads can run `waitForEvent` concurrently.

The `items` semaphore keeps track of the number of items in the buffer. Each time the producer adds an item, it signals `items`, incrementing it by one.

Consumer

Consumer

```
while (1) {
        items.wait();
        mutex.wait();
                event = buffer.get();
        mutex.signal();
        event.process();
}
```

Again, the buffer operation is protected by a mutex, but before the consumer gets to it, it has to decrement `items`. If `items` is zero or negative, the consumer blocks until a producer signals.

Producer

Producer

```
while (1) {
        event = waitForEvent();
        mutex.wait();
                buffer.add(event);
                items.signal();
        mutex.signal();
}
```

Hmmm…
Anything we could do better here?

$\boxed{\text{Time to think before you turn the page}}$ Although this solution is correct, there is an opportunity to make one small improvement to its performance. Imagine that there is at least one consumer in queue when a producer signals `items`. If the scheduler allows the consumer to run, what happens next? It immediately blocks on the mutex that is (still) held by the producer.

Blocking and waking up are moderately expensive operations; performing them unnecessarily can impair the performance of a program. So it would probably be better to rearrange the producer like this:

Better Producer

```
while (1) {
        event = waitForEvent();
        mutex.wait();
                buffer.add(event);
        mutex.signal();
        items.signal();
}
```

Now we don't bother unblocking a consumer until we know it can proceed (except in the rare case that another producer beats it to the mutex).

Consumer Issue

Consumer

```
while (1) {
        items.wait();
        mutex.wait();
                event = buffer.get();
        mutex.signal();
        event.process();
}
```

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

Time to think before you turn the page

But looking at the consumer code, we see the possibility that several consumers could decrement `items` before any of them gets the mutex and removes an item from the buffer. At least for a little while, `items` would be inaccurate.

Let's try to fix this

Consumer

```
while (1) {
        mutex.wait();
                items.wait();
                event = buffer.get();
        mutex.signal();
        event.process();
}
```

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.

This is a common error in synchronization code: any time you wait for a semaphore while holding a mutex, there is a danger of deadlock. When you are checking a solution to a synchronization problem, you should check for this kind of deadlock.

Finite Buffer

In the example I described above, event-handling threads, the shared buffer is usually infinite (more accurately, it is bounded by system resources like physical memory and swap space).

In the kernel of the operating system, though, there are limits on available space. Buffers for things like disk requests and network packets are usually fixed size. In situations like these, we have an additional synchronization constraint:

- If a producer arrives when the buffer is full, it blocks until a consumer removes an item.

Assume that we know the size of the buffer. Call it `bufferSize`. Since we have a semaphore that is keeping track of the number of items, it is tempting to write something like

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

But we can’t. Remember that we can’t check the current value of a semaphore; the only operations are `wait` and `signal`.

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

Time to think before you turn the page

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. Time to think before you turn the page

Finite Buffer Hint

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

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

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

Finite Buffer Solution

```
//Consumer
items.wait();
mutex.wait();
        event = buffer.get();
mutex.signal();
spaces.signal();
event.process();
//Producer
event = waitForEvent();
spaces.wait();
mutex.wait();
        buffer.add(event);
mutex.signal();
items.signal();
```

3.3 Readers-writers problem

Readers-Writers Problem

The next classical problem, called the Reader-Writer Problem, pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

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

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

The exclusion pattern here might be called **categorical mutual exclusion**. A thread in the critical section does not necessarily exclude other threads, but the presence of one category in the critical section excludes other categories.

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 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. Time to think before you turn the page

Readers-Writers Hints

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

```
int readers = 0
mutex = Semaphore(1)
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

The counter `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, and 0 otherwise. This demonstrates the naming convention I use for semaphores that indicate a condition. In this convention, "wait" usually means "wait for the condition to be true" and "signal" means "signal that the condition is true".

Readers-writers solution

Writers-code simple:

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

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

When the writer exits, can it be sure that the room is now empty? Yes, because it knows that no other thread can have entered while it was there.

Readers-writers solution

The code for readers is similar to the barrier code we saw in the previous section. We keep track of the number of readers in the room so that we can give a special assignment to the first to arrive and the last to leave.

The first reader that arrives has to wait for `roomEmpty`. If the room is empty, then the reader proceeds and, at the same time, bars writers. Subsequent readers can still enter because none of them will try to wait on `roomEmpty`.

If a reader arrives while there is a writer in the room, it waits on `roomEmpty`. Since it holds the mutex, any subsequent readers queue on `mutex`.

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

```
mutex.wait();
        readers += 1;
        if (readers == 1)
                roomEmpty.wait();     // first in locks
mutex.signal();
# critical section for readers
mutex.wait();
        readers -= 1;
        if (readers == 0)
                roomEmpty.signal();   // last out unlocks
mutex.signal();
```

Readers-Writers

The code after the critical section is similar. The last reader to leave the room turns out the lights—that is, it signals `roomEmpty`, possibly allowing a waiting writer to enter.

Again, to demonstrate that this code is correct, it is useful to assert and demonstrate a number of claims about how the program must behave. Can you convince yourself that the following are true?

- Only one reader can queue waiting for `roomEmpty`, but several writers might be queued.
- When a reader signals `roomEmpty` the room must be empty.

Patterns similar to this reader code are common: the first thread into a section locks a semaphore (or queues) and the last one out unlocks it. In fact, it is so common we should give it a name and wrap it up in an object.

The name of the pattern is **Light-switch**, by analogy with the pattern where the first person into a room turns on the light (locks the mutex) and the last one out turns it off (unlocks the mutex). Here is a class definition for a Light-switch:

```
class Lightswitch:
// constructor
        self.counter = 0;
        self.mutex = Semaphore(1);
// lock
def lock(self, semaphore):
        self.mutex.wait()
                self.counter ++;
                if (self.counter == 1)
                        semaphore.wait();
                self.mutex.signal();
// unlock
def unlock(self, semaphore):
        self.mutex.wait();
                self.counter--;
                if (self.counter == 0)
                        semaphore.signal()
        self.mutex.signal()
```

`lock` takes one parameter, a semaphore that it will check and possibly hold. If the semaphore is locked, the calling thread blocks on `semaphore` and all subsequent threads block on `self.mutex`. When the semaphore is unlocked, the first waiting thread locks it again and all waiting threads proceed.

If the semaphore is initially unlocked, the first thread locks it and all subsequent threads proceed.

`unlock` has no effect until every thread that called `lock` also calls `unlock`. When the last thread calls `unlock`, it unlocks the semaphore.

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**,

Readers-Writers with Light-switches

```
readLightswitch = Lightswitch();
roomEmpty = Semaphore(1);
// readLightswitch is a shared Lightswitch
// object whose counter is initially zero.
// --- Writers ---
roomEmpty.wait();
        //critical section for writers
roomEmpty.signal();
// --- Readers ---
readLightswitch.lock(roomEmpty)
        // critical section for readers
readLightswitch.unlock(roomEmpty)
```

3.3.1 Starvation

Starvation

In the previous solution, is there any danger of deadlock? In order for a deadlock to occur, it must be possible for a thread to wait on a semaphore while holding another, and thereby prevent itself from being signalled.

In this example, deadlock is not possible, but there is a related problem that is almost as bad: it is possible for a writer to starve.

If a writer arrives while there are readers in the critical section, it might wait in queue forever while readers come and go. As long as a new reader arrives before the last of the current readers departs, there will always be at least one reader in the room.

This situation is not a deadlock, because some threads are making progress, but it is not exactly desirable. A program like this might work as long as the load on the system is low, because then there are plenty of opportunities for the writers. But as the load increases the behaviour of the system would deteriorate quickly (at least from the point of view of writers).

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

PUZZLE

PUZZLE 22

Extend this solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter. Time to think before you turn the page

hint...

Here's a hint. You can add a turnstile for the readers and allow writers to lock it. The writers have to pass through the same turnstile, but they should check the `roomEmpty` semaphore while they are inside the turnstile. If a writer gets stuck in the turnstile it has the effect of forcing the readers to queue at the turnstile. Then when the last reader leaves the critical section, we are guaranteed that at least one writer enters next (before any of the queued readers can proceed).

hint

So we will need

```
readSwitch = Lightswitch();
roomEmpty = Semaphore(1);
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

```
// writers
turnstile.wait();
        roomEmpty.wait();
        // critical section for writers
turnstile.signal();
roomEmpty.signal();
```

```
// readers
turnstile.wait();
turnstile.signal();
readSwitch.lock(roomEmpty);
        // critical section for readers
readSwitch.unlock(roomEmpty);
```

Priority to Writers

When the last reader leaves, it signals `roomEmpty`, unblocking the waiting writer. The writer immediately enters its critical section, since none of the waiting readers can pass the turnstile.

When the writer exits it signals `turnstile`, which unblocks a waiting thread, which could be a reader or a writer. Thus, this solution guarantees that at least one writer gets to proceed, but it is still possible for a reader to enter while there are writers queued.

Depending on the application, it might be a good idea to give more priority to writers. For example, if writers are making time-critical updates to a data structure, it is best to minimize the number of readers that see the old data before the writer has a chance to proceed.

In general, though, it is up to the scheduler, not the programmer, to choose which waiting thread to unblock. Some schedulers use a first-in-first-out queue, which means that threads are unblocked in the same order they queued. Other schedulers choose at random, or according to a priority scheme based on the properties of the waiting threads.

If your programming environment makes it possible to give some threads priority over others, then that is a simple way to address this issue. If not, you will have to find another way.

Puzzle 23

: Write a solution to the readers-writers problem that gives priority to writers. That is, once a writer arrives, no readers should be allowed to enter until all writers have left the system.

Time to think before you turn the page

Hint...

The variables we are going to use:

```
readSwitch = Lightswitch()
writeSwitch = Lightswitch()
noReaders = Semaphore(1)
noWriters = Semaphore(1)
```

Solution - Writers have priority

If a reader is in the critical section, it holds `noWriters`, but it doesn't hold `noReaders`. Thus if a writer arrives it can lock `noReaders`, which will cause subsequent readers to queue.

When the last reader exits, it signals `noWriters`, allowing any queued writers to proceed.

When a writer is in the critical section it holds both `noReaders` and `noWriters`. This has the (relatively obvious) effect of insuring that there are no readers and no other writers in the critical section. In addition, `writeSwitch` has the (less obvious) effect of allowing multiple writers to queue on `noWriters`, but keeping `noReaders` locked while they are there. Thus, many writers can pass through the critical section without ever signaling `noReaders`. Only when the last writer exits can the readers enter.

Of course, a drawback of this solution is that now it is possible for *readers* to starve (or at least face long delays). For some applications it might be better to get obsolete data with predictable turnaround times.

3.4 No-starve Mutex

No-starve Mutex

In the previous section, we addressed what I'll call **categorical starvation**, in which one category of threads (readers) allows another category (writers) to starve. At a more basic level, we have to address the issue of **thread starvation**, which is the possibility that one thread might wait indefinitely while others proceed.

For most concurrent applications, starvation is unacceptable, so we enforce the requirement of **bounded waiting**, which means that the time a thread waits on a semaphore (or anywhere else, for that matter) has to be provably finite.

In part, starvation is the responsibility of the scheduler. Whenever multiple threads are ready to run, the scheduler decides which one or, on a parallel processor, which set of threads gets to run. If a thread is never scheduled, then it will starve, no matter what we do with semaphores.

So in order to say anything about starvation, we have to start with some assumptions about the scheduler. If we are willing to make a strong assumption, we can assume that the scheduler uses one of the many algorithms that can be proven to enforce bounded waiting. If we don't know what algorithm the scheduler uses, then we can get by with a weaker assumption:

> Property 1: if there is only one thread that is ready to run, the scheduler has to let it run.

If we can assume Property 1, then we can build a system that is provably free of starvation. For example, if there are a finite number of threads, then any program that contains a barrier cannot starve, since eventually all threads but one will be waiting at the barrier, at which point the last thread has to run.

In general, though, it is non-trivial to write programs that are free from starvation unless we make the stronger assumption:

> Property 2: if a thread is ready to run, then the time it waits until it runs is bounded.

In our discussion so far, we have been assuming Property 2 implicitly, and we will continue to. On the other hand, you should know that many existing systems use schedulers that do not guarantee this property strictly.

Even with Property 2, starvation rears its ugly head again when we introduce semaphores. In the definition of a semaphore, we said that when one thread executes `signal`, one of the waiting threads gets woken up. But we never said which one. In order to say anything about starvation, we have to make assumptions about the behaviour of semaphores.

The weakest assumption that makes it possible to avoid starvation is:

> 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.

This requirement may seem obvious, but it is not trivial. It has the effect of barring one form of problematic behaviour, which is a thread that signals a semaphore while other threads are waiting, and then keeps running, waits on the same semaphore, and gets its own signal! If that were possible, there would be nothing we could do to prevent starvation.

With Property 3, it becomes possible to avoid starvation, but even for something as simple as a mutex, it is not easy. For example, imagine three threads running the following code:

```
while (True) {
        mutex.wait();
                # critical section
        mutex.signal();
}
```

The `while` statement is an infinite loop; in other words, as soon as a thread leaves the critical section, it loops to the top and tries to get the mutex again.

Imagine that Thread A gets the mutex and Thread B and C wait. When A leaves, B enters, but before B leaves, A loops around and joins C in the queue. When B leaves, there is no guarantee that C goes next. In fact, if A goes next, and B joins the queue, then we are back to the starting position, and we can repeat the cycle forever. C starves.

The existence of this pattern proves that the mutex is vulnerable to starvation. One solution to this problem is to 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.

For example, if the semaphore maintains a first-in-first-out queue, then Property 4 holds because when a thread joins the queue, the number of threads ahead of it is finite, and no threads that arrive later can go ahead of it.

A semaphore that has Property 4 is sometimes called a **strong semaphore**; one that has only Property 3 is called a **weak semaphore**. We have shown that with weak semaphores, the simple mutex solution is vulnerable to starvation. In fact, Dijkstra conjectured that it is not possible to solve the mutex problem without starvation using only weak semaphores.

In 1979, J.M. Morris refuted the conjecture by solving the problem, assuming that the number of threads is finite [2]. If you are interested in this problem, the next section presents his solution. If this is not your idea of fun, you can just assume that semaphores have Property 4 and go on to the next section.

Puzzle: write a solution to the mutual exclusion problem using weak semaphores. Your solution should provide the following guarantee: once a thread arrives and attempts to enter the mutex, there is a bound on the number of threads that can proceed ahead of it. You can assume that the total number of threads is finite.

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)

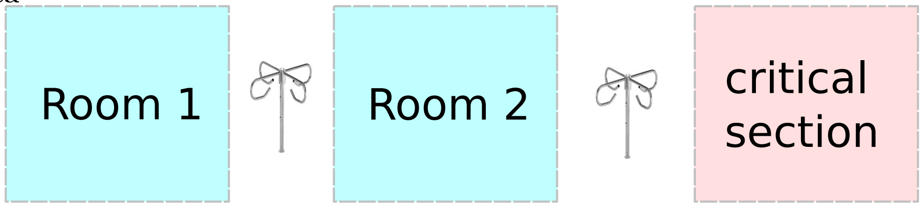
PUZZLE 24
write a solution to the mutual exclusion problem using weak semaphores.

- 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

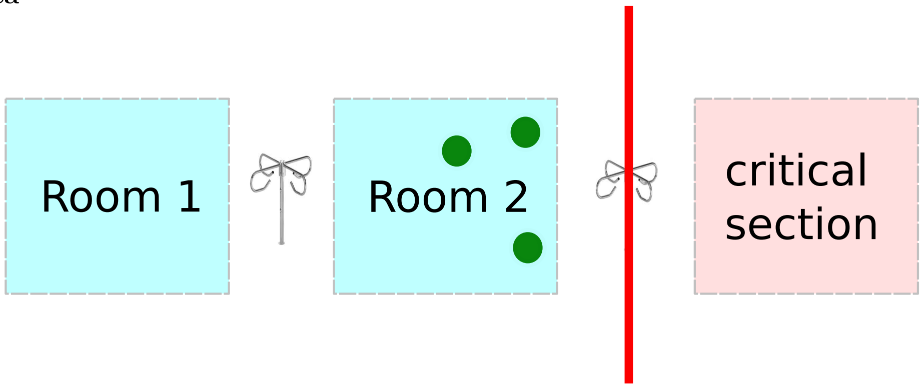
hints

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

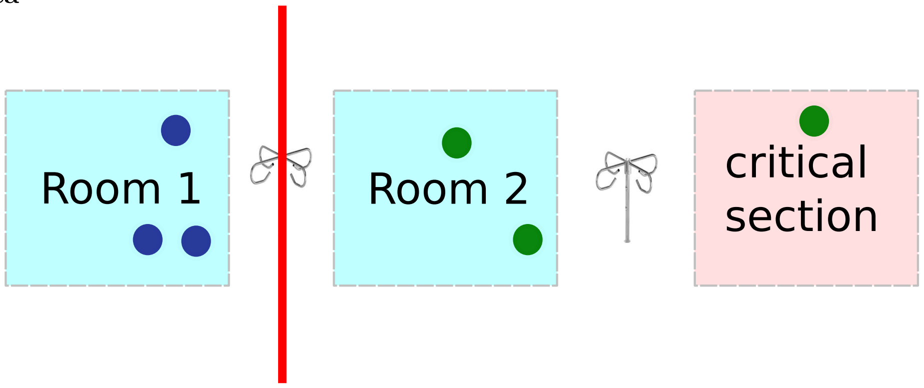
idea



idea



idea



variables

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

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

let's develop the solution

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

let's develop the solution

```
// entering room 1 - and counting
mutex.wait();
        room1 ++;
mutex.signal();
// waiting on turnstile 1 to enter room 2
t1.wait();
// entering room 2 - we need to adapt the counters
room2++;
mutex.wait();
        room1--;
```

let's develop the solution

```
// entering room 1 - and counting
mutex.wait();
        room1 ++;
mutex.signal();
// waiting on turnstile 1 to enter room 2
t1.wait();
// entering room 2 - we need to adapt the counters
room2++;
mutex.wait();
        room1--;
        if (room1 == 0){
                mutex.signal();
                t2.signal()M
        } else {
                mutex.signal();
                t1.signal();
        }
// end of room 2
```

let's develop the solution

```
// entering room 1 - and counting
mutex.wait();
        room1 ++;
mutex.signal();
// waiting on turnstile 1 to enter room 2
t1.wait();
// entering room 2 - we need to adapt the counters
room2++;
```

```
mutex.wait();
        room1--;
        if (room1 == 0){
                mutex.signal();
                t2.signal()M
        } else {
                mutex.signal();
                t1.signal();
        }
// end of room 2
t2.wait();
room2--;
// critical section
```

let’s develop the solution

```
// entering room 1 - and counting
mutex.wait();
        room1 ++;
mutex.signal();
// waiting on turnstile 1 to enter room 2
t1.wait();
// entering room 2 - we need to adapt the counters
room2++;
mutex.wait();
        room1--;
        if (room1 == 0){
                mutex.signal();
                t2.signal()M
        } else {
                mutex.signal();
                t1.signal();
        }
// end of room 2
t2.wait();
room2--;
// critical section
if (room2==0) t1.signal else t2.signal;
```

3.5 Dining Philosophers

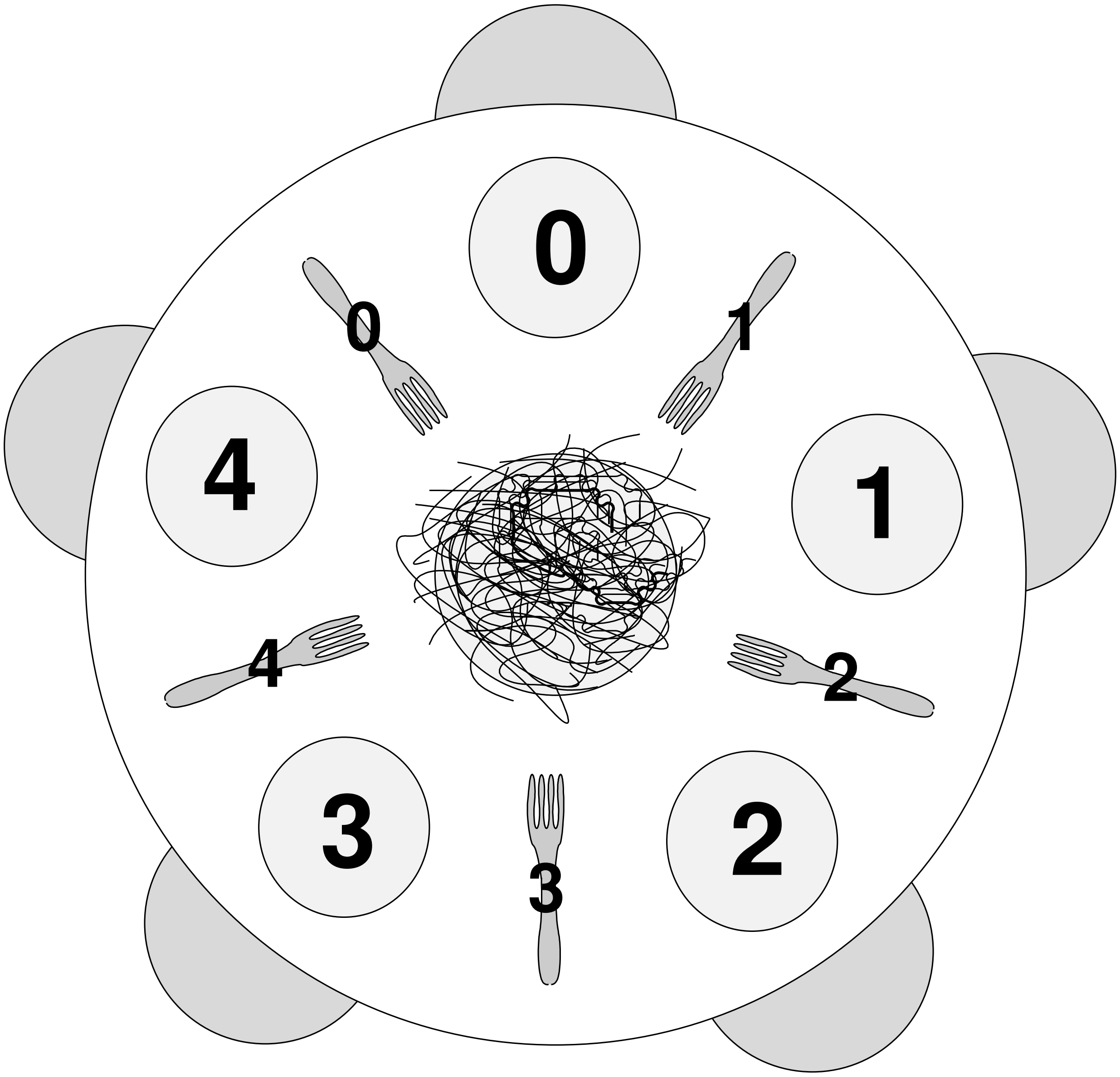
Dining Philosophers

The Dining Philosophers Problem was proposed by Dijkstra in 1965, when dinosaurs ruled the earth [1]. It appears in a number of variations, but the standard features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

```
while(true) {
think();
get_forks();
eat();
put_forks();
}
```

The forks represent resources that the threads have to hold exclusively in order to make progress. The thing that makes the problem interesting, unrealistic, and unsanitary, is that the philosophers need *two* forks to eat, so a hungry philosopher might have to wait for a neighbour to put down a fork.

Assume that the philosophers have a local variable `i` that identifies each philosopher with a value in (0..4). Similarly, the forks are numbered from 0 to 4, so that Philosopher i has fork i on the right and fork $(i\ 1)\%5$ on the left. Here is a diagram of the situation:



Assuming that the philosophers know how to `think` and `eat`, our job is to 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.

The last requirement is one way of saying that the solution should be efficient; that is, it should allow the maximum amount of concurrency.

We make no assumption about how long `eat` and `think` take, except that `eat` has to terminate eventually. Otherwise, the third constraint is impossible—if a philosopher keeps one of the forks forever, nothing can prevent the neighbours from starving.

To make it easy for philosophers to refer to their forks, we can use the functions `left` and `right`:

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

The `%` operator wraps around when it gets to 5, so `(4 + 1) % 5 = 0`.

Since we have to enforce exclusive access to the forks, it is natural to use a list of Semaphores, one for each fork. Initially, all the forks are available.

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

This notation for initializing a list might be unfamiliar to readers who don't use Python. The `range` function returns a list with 5 elements; for each element of this list, Python creates a Semaphore with initial value 1 and assembles the result in a list named `forks`.

Here is an initial attempt at `get_fork` and `put_fork`:

```
void get_forks(i) {
        fork[right(i)].wait();
        fork[left(i)].wait();
}
void put_forks(i) {
        fork[right(i)].signal();
        fork[left(i)].signal();
}
```

It's clear that this solution satisfies the first constraint, but we can be pretty sure it doesn't satisfy the other two, because if it did, this wouldn't be an interesting problem and you would be reading this chapter.

Puzzle: what's wrong?

- Time to think before you turn the page

The problem is that the table is round. As a result, each philosopher can pick up a fork and then wait forever for the other fork. Deadlock!

Puzzle: write a solution to this problem that prevents deadlock.

Hint: one way to avoid deadlock is to think about the conditions that make deadlock possible and then change one of those conditions. In this case, the deadlock is fairly fragile—a very small change breaks it.

Solution

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

We can control the number of philosophers at the table with a Multiplex named `footman` that is initialized to 4. Then the solution looks like this:

```
footman = Semaphore(4);
void get_forks(i) {
        footman.wait();
        fork[right(i)].wait();
        fork[left(i)].wait();
}
void put_forks(i) {
        fork[right(i)].signal();
        fork[left(i)].signal();
        footman.signal();
}
```

In addition to avoiding deadlock, this solution also guarantees that no philosopher starves. Imagine that you are sitting at the table and both of your neighbours are eating. You are blocked waiting for your right fork. Eventually your right neighbour will put it down, because `eat` can't run forever. Since you are the only thread waiting for that fork, you will necessarily get it next. By a similar argument, you cannot starve waiting for your left fork.

Therefore, the time a philosopher can spend at the table is bounded. That implies that the wait time to get into the room is also bounded, as long as `footman` has Property 4.

This solution shows that by controlling the number of philosophers, we can avoid deadlock. Another way to avoid deadlock is to change the order in which the philosophers pick up forks. In the original non-solution, the philosophers are "righties"; that is, 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.

- Time to think before you turn the page

Hint: 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.

The proof works by contradiction. First, assume that deadlock is possible. Then choose one of the supposedly deadlocked philosophers. If she's a lefty, you can prove that the philosophers are all lefties, which is a contradiction. Similarly, if she's a righty, you can prove that they are all righties. Either way you get a 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

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

Tanenbaum's Solution

```
void get_fork(i) {
        mutex.wait();
                state[i] = "hungry";
                test(i);
        mutex.signal();
        sem[i].wait();
}
void put_fork(i) {
        mutex.wait()
                state[i] = "thinking";
                test(right(i));
                test(left(i));
        mutex.signal()
}
void test(i) {
        if ((state[i] == "hungry") &&
                 (state[left (i)]  != "eating") &&
                 (state[right (i)]  != "eating")) {
                                state[i] = "eating";
                                sem[i].signal();
        }
}
```

Tanenbaum's Solution

The `test` function checks whether the *i*th philosopher can start eating, which he can if he is hungry and neither of his neighbours are eating. If so, the `test` signals semaphore i.

There are two ways a philosopher gets to eat. In the first case, the philosopher executes `get_forks`, finds the forks available, and proceeds immediately. In the second case, one of the neighbours is eating and the philosopher blocks on its own semaphore. Eventually, one of the neighbours will finish, at which point it executes `test` on both of its neighbours. It is possible that both tests will succeed, in which case the neighbours can run concurrently. The order of the two tests doesn't matter.

In order to access `state` or invoke `test`, a thread has to hold `mutex`. Thus, the operation of checking and updating the array is atomic. Since a philosopher can only proceed when we know both forks are available, exclusive access to the forks is guaranteed.

No deadlock is possible, because the only semaphore that is accessed by more than one philosopher is `mutex`, and no thread executes `wait` while holding `mutex`.

But again, starvation is tricky.

Puzzle: Either convince yourself that Tanenbaum's solution prevents starvation or find a repeating pattern that allows a thread to starve while other threads make progress.

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

The original barbershop problem was proposed by Dijkstra. A variation of it appears in Silberschatz and Galvin's *Operating Systems Concepts* .

> 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.

To make the problem a little more concrete, I added the following information:

- 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.

Bibliography

References

[1] Edsger Dijkstra. Cooperating sequential processes, 1965. Reprinted in *Programming Languages*, F. Genuys, ed., Academic Press, New York 1968.

[2] Joseph M. Morris. A starvation-free solution to the mutual exclusion problem. *Information Processing Letters*, 8(2):76–80, February 1979.