Semaphores
Reference: LBoS

- Little Book of Semaphores, by Allen Downey
  - Focuses on synchronization using semaphores
  - Includes classical and non-traditional problems
  - Lots of sample code in quasi-Python syntax
Semaphore rules

1. When you create the semaphore, you can initialize its value to any integer, but 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.
# Initialization & Operations

**Operation names?**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred = Semaphore(1)</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.increment_and_wake_a_waiting_process_if_any()</td>
</tr>
<tr>
<td>2</td>
<td>fred.decrement_and_block_if_the_result_is_negative()</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.increment()</td>
</tr>
<tr>
<td>2</td>
<td>fred.decrement()</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.signal()</td>
</tr>
<tr>
<td>2</td>
<td>fred.wait()</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.V()</td>
</tr>
<tr>
<td>2</td>
<td>fred.P()</td>
</tr>
</tbody>
</table>

**2.3 Why semaphores?**

In most programming environments, an implementation of semaphores is available as part of the programming language or the operating system. Dijkstra, who wisely realized that a meaningless name is better than a misleading name, proposed different names in different environments. Actually, I don't think the world is likely to embrace either of these names soon. In general, there is no way to know before a thread decrements a semaphore whether it will block or not (in specific cases you might be able to prove that it will or will not). Semaphores can be implemented efficiently on many systems, so solutions using semaphores are often clean and organized, making it easy to demonstrate their correctness. Solutions using semaphores are often portable and usually easier to debug. Errors are usually easier to locate than deadlocks, which are the original names in different languages. Even if the other pairs are completely meaningless to people who speak Dutch, aren't completely meaningless to people who speak Dutch. When you signal a semaphore, you don't necessarily know whether another thread will get woken up, both threads continue running concurrently. There is no way to know after which thread will get woken up, both threads continue running concurrently. There is no way to know whether another thread will get woken up when you signal a semaphore. It's true that we don't describe what they are often used for. I consider the other pairs misleading because they use semaphores to solve synchronization problems, and the other pairs are the original names in different environments.
Basic synchronization patterns

1. Rendezvous
2. Mutual exclusion (Mutex)
3. Multiplex
4. Generalized rendezvous
   - Barrier & Turnstile
1. Rendezvous

Problem: Ensure that $a_1 < b_2$, $b_1 < a_2$

Hint: use the following variables

```python
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
1. Rendezvous

```
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
```

Here is my solution, based on the previous hint:

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```

While working on the previous problem, you might have tried something like this:

```
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
```

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.

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.

3.2.3 Deadlock #1

Again, while working on the previous problem, you might have tried something like this:

```
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
```

If so, I hope you rejected it quickly, because 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. Mutual exclusion

Problem: Ensure that critical sections do not overlap

Thread A

\[
\text{count} = \text{count} + 1
\]

Thread B

\[
\text{count} = \text{count} + 1
\]

Hint: use the following variable

\[
\text{mutex} = \text{Semaphore}(1)
\]
2. Mutual exclusion

mutex = Semaphore(1)

Thread A
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()

Thread B
mutex.wait()
   # critical section
   count = count + 1
mutex.signal()
3. Multiplex

To allow multiple threads to run in the critical section, just initialize the semaphore to $n$, which is the maximum number of threads that should be allowed.

At any time, the value of the semaphore represents the number of additional threads that may enter. If the value is zero, then the next thread will block until one of the threads inside exits and signals. When all threads have exited the value of the semaphore is restored to $n$.

Since the solution is symmetric, it's conventional to show only one copy of the code, but you should imagine multiple copies of the code running concurrently in multiple threads.

Listing 3.1: Multiplex solution

```java
1 multiplex.wait()
2 critical section
3 multiplex.signal()
```

What happens if the critical section is occupied and more than one thread arrives? Of course, what we want is for all the arrivals to wait. This solution does exactly that. Each time an arrival joins the queue, the semaphore is decremented, so that the value of the semaphore (negated) represents the number of threads in queue.

When a thread leaves, it signals the semaphore, incrementing its value and allowing one of the waiting threads to proceed.

Thinking again of metaphors, in this case I find it useful to think of the semaphore as a set of tokens (rather than a lock). As each thread invokes `wait`, it picks up one of the tokens; when it invokes `signal`, it releases one. Only a thread that holds a token can enter the room. If no tokens are available when a thread arrives, it waits until another thread releases one.

In real life, ticket windows sometimes use a system like this. They hand out tokens (sometimes poker chips) to customers in line. Each token allows the holder to buy a ticket.

3.5 Barrier

Consider again the Rendezvous problem from Section 3.2. A limitation of the solution we presented is that it does not work with more than two threads.

Puzzle: Generalize the rendezvous solution. Every thread should run the following code:

Listing 3.2: Barrier code

```java
1 rendezvous
2 critical point
3 Permits $N$ threads through into their critical sections
```

```java
1 multiplex = Semaphore(N)
2 multiplex.wait()
3 critical section
4 multiplex.signal()
```
4. Generalized Rendezvous

Problem: Generalize the rendezvous solution. Every thread should run the following code

```plaintext
1 rendezvous
2 critical point
```

Hint: use the following variables

```plaintext
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```
3.5 Barrier hint

For many of the problems in this book I will provide hints by presenting the
variables I used in my solution and explaining their roles.

Listing 3.3: Barrier hint

1 \( n = \text{the number of threads} \)
2 \( \text{count} = 0 \)
3 \( \text{mutex} = \text{Semaphore}(1) \)
4 \( \text{barrier} = \text{Semaphore}(0) \)
4. Generalized Rendezvous

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rendezvous</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mutex.wait()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>count = count + 1</td>
</tr>
<tr>
<td>5</td>
<td>mutex.signal()</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>if count == n: barrier.signal()</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>barrier.wait()</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>} “turnstile”</td>
</tr>
<tr>
<td>11</td>
<td>barrier.signal()</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>critical point</td>
<td></td>
</tr>
</tbody>
</table>
4. Generalized Rendezvous

```python
1    rendezvous
2
3    mutex.wait()
4        count = count + 1
5    mutex.signal()
6
7    if count == n: turnstile.signal()
8
9    turnstile.wait()
10   turnstile.signal()
11
12    critical point
```

what is the value of turnstile when all threads reach the critical point?
4. Generalized Rendezvous

```
1    rendezvous
2
3    mutex.wait()
4     count = count + 1
5    mutex.signal()
6
7    if count == n: turnstile.signal()
8
9    { turnstile.wait()
10       turnstile.signal()
11}
12    critical point
```

value of turnstile is in range [1,n]

can we eliminate this non-determinism?

threads may be preempted here

between 1 and n threads signal here

balanced # of waits/signals
4. Generalized Rendezvous

```java
1     rendezvous
2
3     mutex.wait()
4     count = count + 1
5     if count == n: turnstile.signal()
6     mutex.signal()
7
8     turnstile.wait()
9     turnstile.signal()
10
11    critical point
```

value of `turnstile` at critical point is predictably 1 (but it is no longer a usable barrier)
4. Generalized Rendezvous

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

Listing 3.8: Reusable barrier non-solution

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

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.

Does this work reliably?

A thread may “lap” the other threads, mess up count, and fail to block before the critical point the next time.
4. Generalized Rendezvous

Problem: Build a generalized, reusable rendezvous solution — i.e., where threads all rendezvous again after each time through the CS

Hint: use the following variables

```
1  turnstile = Semaphore(0)
2  turnstile2 = Semaphore(1)
3  mutex = Semaphore(1)
```
3.6 Reusable barrier

3.6.4 Reusable barrier hint

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.

Listing 3.9: Reusable barrier hint

```python
1 turnstile = Semaphore(0)
2 turnstile2 = Semaphore(1)
3 mutex = Semaphore(1)
```
4. Generalized Rendezvous

```python
# rendezvous
mutex.wait()
count += 1
if count == n:
    turnstile2.wait()
    turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

# critical point
mutex.wait()
count -= 1
if count == 0:
    turnstile.wait()
    turnstile2.signal()
mutex.signal()

turnstile2.wait()
turnstile2.signal()
```

lock the second turnstile, unlock the first

first turnstile

lock the first turnstile, unlock the second

second turnstile
class Barrier:
    def __init__(self, n):
        self.n = n
        self.count = 0
        self.mutex = Semaphore(1)
        self.turnstile = Semaphore(0)
        self.turnstile2 = Semaphore(1)

    def phase1(self):
        self.mutex.wait()
        self.count += 1
        if self.count == self.n:
            self.turnstile2.wait()
            self.turnstile.signal()
            self.mutex.signal()
            self.turnstile.wait()
            self.turnstile.signal()

    def phase2(self):
        self.mutex.wait()
        self.count -= 1
        if self.count == 0:
            self.turnstile.wait()
            self.turnstile2.signal()
            self.mutex.signal()
            self.turnstile2.wait()
            self.turnstile2.signal()

    def wait(self):
        self.phase1()
        self.phase2()
Classical synchronization problems

1. Producer/Consumer
2. Readers/Writers
3. Dining Philosophers
Problem: producer & consumer threads repeatedly accessing a finite, non-thread-safe buffer

Hint: use the following variables

```java
# Producer
item = produce()
buffer.put(item)

# Consumer
item = buffer.get()
consume(item)
```

```java
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.capacity())
```
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.capacity())
1. Producer/Consumer (revisited)

# Producer

```python
item = produce()
spaces.wait()
mutex.wait()
buffer.put(item)
mutex.signal()
items.signal()
```

# Consumer

```python
items.wait()
mutex.wait()
    item = buffer.get()
mutex.signal()
spaces.signal()
consume(item)
```
2. Readers/Writers

Problem: unlimited # of readers allowed to access shared resource at once, but at most one writer; no readers while writer is accessing resource

- i.e., categorical mutex

- can model access to the resource as a “room”, where any # of readers may occupy the room, but it must be vacated for a single writer to enter

Hint: use the following variables

\[
\begin{align*}
    n\text{\_readers} &= 0 \\
    \text{mutex} &= \text{Semaphore}(1) \\
    \text{roomEmpty} &= \text{Semaphore}(1)
\end{align*}
\]
n_readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
2. Readers/Writers

# Readers

mutex.wait()

n_readers += 1

if n_readers == 1:
    roomEmpty.wait()

mutex.signal()

# critical section

mutex.wait()

n_readers -= 1

if n_readers == 0:
    roomEmpty.signal()

mutex.signal()

# Writers

roomEmpty.wait()

# critical section

roomEmpty.signal()
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)

    def lock(self, switch):
        self.mutex.wait()
        self.counter += 1
        if self.counter == 1:
            switch.wait()
        self.mutex.signal()

    def unlock(self, switch):
        self.mutex.wait()
        self.counter -= 1
        if self.counter == 0:
            switch.signal()
        self.mutex.signal()

- Encapsulates “first-in locks, last-out unlocks” synchronization semantic

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

# Writers
roomEmpty.wait()
    # critical section
roomEmpty.signal()

# Readers
readSwitch.lock(roomEmpty)
    # critical section
readSwitch.unlock(roomEmpty)
```
2. Readers/Writers with Lightswitch

- Problem: a constant stream of readers into the room may starve writers!

- How to guarantee entry into room for a newly arrived writer?

- Hint:

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

# Writers
roomEmpty.wait()
    # critical section
roomEmpty.signal()

# Readers
readSwitch.lock(roomEmpty)
    # critical section
readSwitch.unlock(roomEmpty)
```
roomEmpty = Semaphore(1)
readSwitch = Lightswitch()
turnstile = Semaphore(1)
2. No-starve Readers/Writers

```python
roomEmpty  = Semaphore(1)
readSwitch = Lightswitch()
turnstile  = Semaphore(1)

# Readers
turnstile.wait()
turnstile.signal()
readSwitch.lock(roomEmpty)
# critical section
readSwitch.unlock(roomEmpty)

# Writers
turnstile.wait()
roomEmpty.wait()
turnstile.signal()
roomEmpty.signal()

writer blocks turnstile while waiting on roomEmpty, preventing readers from filing into room

when last reader leaves the room, writer enters and releases turnstile
```
3. Dining Philosophers

Problem: Philosophers are seated about a round table, each with a plate of spaghetti in front of, and a fork to either side of them — adjacent philosophers share a fork

- Philosophers alternate between thinking and eating
- To eat, a philosopher needs to use both forks
  - A fork can only be in use by one philosopher
- Philosophers should not be starved (of spaghetti), and cannot predict how others will behave
3. Dining Philosophers

- Simple setup: model forks as semaphores

```python
forks = [Semaphore(1) for i in range(5)]

# philosopher id → fork id mapping functions
def left(i): return i
def right(i): return (i + 1) % 5
```
3. DP: Naive solution

def get_forks(i):
    fork[left(i)].wait()
    fork[right(i)].wait()

def put_forks(i):
    fork[left(i)].signal()
    fork[right(i)].signal()

- Potential deadlock! All philosophers obtain left fork and starve
3. DP: Global mutex

```python
def get_forks(i):
    mutex.wait()
    fork[left(i)].wait()
    fork[right(i)].wait()
    mutex.signal()

def put_forks(i):
    fork[left(i)].signal()
    fork[right(i)].signal()
```

- May prohibit a philosopher from eating when their forks are available
- Fails to maximize concurrency
3. DP: Thread limit

```python
footman = Semaphore(4)

def get_forks(i):
    footman.wait()
    fork[left(i)].wait()
    fork[right(i)].wait()

def put_forks(i):
    fork[left(i)].signal()
    fork[right(i)].signal()
    footman.signal()
```

- How realistic is this approach?
3. DP: Resource ordering

- Order all required resources and request only in increasing order
- Prevents a cycle in the resource allocation graph
- How realistic is this approach?

```
def get_forks(i):
    for i in sorted([left(i), right(i))):
        fork[i].wait()
```
3. DP: Tanenbaum’s solution

- Idea: philosophers announce their state $\in \{\text{thinking, eating, hungry}\}$
- Can only eat if neighbors are both not eating
- When done eating, check if neighbor is hungry and help them eat, if possible

```python
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)

def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' \
        and state[left(i)] != 'eating' \
        and state[right(i)] != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```
(let’s mess with this guy)
(starves)
Summary
Concurrency is desirable

- Can help improve CPU and I/O utilization
  - By blocking only part of a task/process instead of the whole thing
- May leverage parallelism for increase in performance
  - Limited by parallel portion of workload (Amdhal’s/Gustafson’s)
- May also help logically partition a task into discrete subtasks
Concurrent programming relies on the OS & HW

- The kernel is the original concurrent program
- Without kernel-level threads, we cannot translate user-level concurrency into performance gains
- Hardware support is needed to build robust and efficient mechanisms for concurrent programming
  - E.g., atomic instructions, interrupt mechanisms
Concurrent tasks overlap non-deterministically, and when they access shared data, we may end up with race conditions.

Synchronizing concurrent tasks to eliminate race conditions while maximizing efficiency, eliminating starvation, etc., is hard!

- Requires thinking in multiple dimensions and accounting for nearly infinite scenarios.

- When not done carefully, may entangle application and synchronization logic, and make code difficult to maintain.