Intro

0.1 Synchronization

From WikipediA:
Synchronization is the coordination of events to operate a system in unison. The familiar conductor of an orchestra serves to keep the orchestra in time. Systems operating with all their parts in synchrony are said to be synchronous or in sync.

In Computer Science it is more complicated: there are many **synchronization constraints**. Examples:

- Serialization: Event A must happen before Event B.
- Mutual exclusion: Events A and B must not happen at the same time.

Clock idea: preset a time when the next event is allowed to starts. Probably will not work - often there is no clock to refer to, or multiple clocks – which are slightly off. Further problems: there is no guarantee how long a process or a thread will take to finish execution (think about context switching).

Solving above with messages:

```
You                        Bob
 a1:  Eat breakfast         b1:  Eat breakfast
 a2:  Work                  b2:  Wait for a call
 a3:  Eat lunch             b3:  Eat lunch
 a4:  Call Bob
```

$$a1 < a2 < a3 < a4$$
$$b1 < b2 < b3$$

where the relation $a1 < a2$ means that a1 happened before a2.

Message passing (call) guarantees $b2 > a4$. Which gives us:

$$b3 > b2 > a4 > a3$$

Definitions:

Events A and B are executed **sequentially** if there is a guarantee that A always happens before B (or B always before A). Otherwise we say that A and B are executed **concurrently**.

Note that concurrent does not mean simultaneously.

Non-determinism – the exact order of instructions is not defined.

print 1 print 2

Notes:

- Execution may be non-deterministic, but final result is deterministic. So such algorithm will still be called deterministic.
- How useful are non-deterministic algorithms? Consider algorithm for testing if number N is prime: 100 times choose a random number x from 1 to $\sqrt{N}$ and if x divides N print `not prime` otherwise print `possibly prime`.
- Another example: Monty Hall problem .....

0.2 Shared variables

Threads have local variables, local variable of one thread are not available to any other part of the program. Sometimes threads have to access same variable (object/data).

Thread A	Thread B
x = 5	x = 7
print x	print x

result is non-deterministic. What are the possible output/value pairs?

Race condition situation when result depends on the timing of 2 or more events.

Race conditions are very rarely part of the design, usually very hard to find bugs.

Race conditions may be hard to notice in your code:

Thread A	Thread B
count = count + 1	count = count + 1

`count = count + 1` is not **atomic** operation. It is Read, Increment, Write.

Definition: an operation that cannot be interrupted is said to be **atomic**.

Assume `count` is a shared variable:

Thread A	Thread B
a1: t = count	b1: t = count
a2: ++t;	b2: ++t;
a3: count = t	b3: count = t

Now – order a1-a2-a3-b1-b2-b3 produces 2, while a1-b1-a2-b2-a3-b3 will produce 1.

What about this?

Thread A	Thread B
++count;	++count;

Is increment of integer (`int`) an atomic operation?

From intel.com `https://software.intel.com/en-us/forums/watercooler-catchall/topic/308940` *The rules for atomic operations may be found in Chapter 7, Locked Atomic Operations, of the IA-32 Intel Architecture Software Developers Manual, Volume 3: System Programming Guide (http://developer.intel.com/design/Pentium4/documentation.htm). Here, it is guaranteed that simple loads or stores will be automatically atomic as long as the memory location is aligned on the appropriate boundary (16-bit boundary for 16-bit values, 32-bit boundary for 32-bit values, and so forth). In addition, simple loads or stores that are not aligned on the appropriate boundary are still guaranteed to be executed atomically if the 16, 32, or 64-bit values fit completely within a 32-byte cache line. Loads and stores that cross cache lines are not guaranteed to be executed atomically. In these cases, you can use the LOCK prefix to guarantee atomic operation of the simple load or store.*

INC and DEC belong to the family of instructions that can read, modify, and write a data value in memory. Thus, their operation is not guarnateed to be atomic unless the LOCK prefix is used for these instructions (when referencing a location in memory). The XCHG instruction automatically causes the LOCK behavior to occur regardless of whether the prefix is used or not.

Rule of thumb – assume the worst at all times, nothing is atomic unless explicitly mentioned.

0.3 Mutex

Mutex – mutual exclusion. Plural: **mutexes**.

Desired properties:

1. *Mutual exclusion:* two threads/processes can never be in the critical section at the same time

2. *Progress* is defined as the following: if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next. This selection cannot be postponed indefinitely. A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

3. *Bounded waiting* or bounded bypass means that the number of times a process is bypassed by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system.

Implementing mutual exclusion:
Attempt 1:
shared variable:

```
threadId = 0
```

Thread A	Thread B
`while ( 1 ) do {`	`while ( 1 ) do {`
`  while ( threadId == 1 ) do {}`	`  while ( threadId == 0 ) do {}`
`  critical section code`	`  critical section code`
`  threadId = 1`	`  threadId = 0`
`  non critical code`	`  non critical code`
`}`	`}`

Mutual exclusion – check.

Progress - no. Threads have to alternate, what if "non critical code" section of thread A are much faster than of thread B? Imaging a situation when A's critical and B's non critical sections start almost simultateously. A critical and A non critical will execute, while B is still in it's non critical. A can start another critical section, but it cannot, since critical sections of A and B have to alternate.

Attempt 2:
shared variable:

```
threadAisin = 0
threadBisin = 0
```

```
Thread A
while ( 1 ) do {
  while ( threadBisin == 1 ) do {}
  threadAisin = 1
  critical section code
  threadAisin = 0
  non critical code
}
```

```
Thread B
while ( 1 ) do {
  while ( threadAisin == 1 ) do {}
  threadBisin = 1
  critical section code
  threadBisin = 0
  non critical code
}
```

Instead of telling "You are next" we use "I'm in" and "I'm out". This solves the alternation problem from the previous attempt, but breaks mutual exclusion!

Consider the following sequence when both threads are outside of their critical sections – for example both threads just started:

```
A: while ( threadBisin == 1 ) do {} //threadBisin is 0
B: while ( threadAisin == 1 ) do {} //threadAisin is 0
A: threadAisin = 1 // but it's too late, thread B is already past the guard-line
B: threadBisin = 1 // but it's too late, thread A is already past the guard-line
A and B start critical section code concurrently!
```

Attempt 3:
shared variable:

```
threadAwantsin = 0
threadBwantsin = 0
```

Thread A	Thread B
`while ( 1 ) do {`	`while ( 1 ) do {`
`  threadAwantsin = 1`	`  threadBwantsin = 1`
`  while ( threadBwantsin == 1 ) do {}`	`  while ( threadAwantsin == 1 ) do {}`
`  critical section code`	`  critical section code`
`  threadAwantsin = 0`	`  threadBwantsin = 0`
`  non critical code`	`  non critical code`
`}`	`}`

Instead of "I'm in" and "I'm out" use "I want in". Also notice that the thread is very polite and even if it wants in it will wait if the other thread also wants in.

Notice the reverse order of setting flag and testing the other thread flag (guard line). The swap gives us mutual exclusion back – remember it was broken in attempt 2.

Unfortunately these 2 threads are too polite. Consider the following sequence when both threads are outside of their critical sections – for example both threads just started:

```
A: threadAwantsin = 1
B: threadBwantsin = 1
A: while ( threadBwantsin == 1 ) do {} //waiting politely
B: while ( threadAwantsin == 1 ) do {} //waiting politely
A and B wait forever
```

This is an example of a **deadlock** = "ultimate no progress"

Attempt 4: hackish fix to the previous attempt
Shared variable:

```
threadAwantsin = 0
threadBwantsin = 0
```

```
Thread A
while ( 1 ) do {
  threadAwantsin = 1
  while ( threadBwantsin == 1 ) do {
    threadAwantsin = 0
    sleep ( delta )
    threadAwantsin = 1
  }
  critical section code
  threadAwantsin = 0
  non critical code
}
```

```
Thread B
while ( 1 ) do {
  threadBwantsin = 1
  while ( threadAwantsin == 1 ) do {
    threadBwantsin = 0
    sleep ( delta )
    threadBwantsin = 1
  }
  critical section code
  threadBwantsin = 0
  non critical code
}
```

Almost impossible problem: the 2 threads execute lines exactly a_1-b_1-a_2-b_2-...-a_i-b_i-... and have exactly the same delay - this will result in the same problem as in attempt 3. But chances of that are very small.

Note: the inner `while` loop looks like an attempt to implement priorities – think that higher priority thread will have a smaller `delta`. Fails as `Win32` priorities did. See next paragraph.

Another problem: starvation of one of the threads is possible - similar to attempt 2. Write a sequence of instructions which will prove that.

Attempt 5 – Dekker's algorithm
shared variable:

```
threadAwantsin = 0
threadBwantsin = 0
favored = A
```

```
Thread A                                  Thread B
while ( 1 ) do {                          while ( 1 ) do {
  threadAwantsin = 1                        threadBwantsin = 1
  while ( threadBwantsin == 1 ) do {        while ( threadAwantsin == 1 ) do {
    if ( favored == B ) do {                  if ( favored == A ) do {
      threadAwantsin = 0                        threadBwantsin = 0
      while ( favored == B ) do {}              while ( favored == A ) do {}
      threadAwantsin = 1                        threadBwantsin = 1
    }                                         }
  }                                         }
  critical section code                     critical section code
  favored = B                               favored = A
  threadAwantsin = 0                        threadBwantsin = 0
  non critical code                         non critical code
}                                         }
```

This is a correct algorithm, so instead of showing a single case when it's not working properly we have to show that it works correctly in all cases:

- if only one thread wants to get in, it's skips `while ( threadAwantsin == 1 ) ...` and enters critical section
- B is in critical section and A wishes to enter.
  - **A is favored thread**
    A then spins in practically empty `while ( threadBwantsin == 1 ) { if ( favored == B ) do {NOP} }` doing busy waiting for B to finish and set `threadBwantsin = 0`
    1) B finishes critical section
    2) B sets `threadBwantsin = 0;`
    A sees `threadBwantsin = 0;` exits the `while` and executes its critical section.
    Notice that when B is finished executing non critical code and wraps back to `threadBwantsin = 1; while ( threadAwantsin == 1 ) do`
    it will get stuck on busy waiting `if ( favored == A ) do {}` letting A to proceed to critical section
  - **B is favored thread**. Explain how this scenario is possible! A then enters in `while ( threadBwantsin == 1 )` , resets `threadAwantsin = 0` and gets stuck on `if ( favored == B ) do {}` letting B to finish its critical section and set `favored = A`. At this moment A continues to set `threadAwantsin = 1` and even if B manages to wrap around and get to `while ( threadAwantsin == 1 ) do` A will continue into critical section first, since `favored = A`
- if both threads want to enter, then only the favored thread will be allowed to.

0.4 Amdahl's Law

Program speedup r is calculated as

$$r = \frac{1}{1 - p + \frac{p}{s}}$$

where p is the fraction of the time that is effected by speedup and s is the speedup. When applied to a parallel algorithm p is the fraction of the time that is "parallized" and s is the number of cores available to the algorithm.

Example:

- half of the algorithm may be parallelized to 2 cores:

$$r = \frac{1}{1 - \frac{1}{2} + \frac{(\frac{1}{2})}{2}} = \frac{4}{3}$$

  algorithm is 33% faster.

- half of the algorithm may be parallelized to 4 cores:

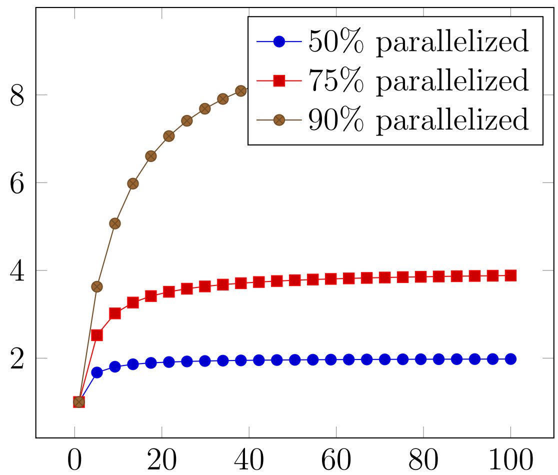
$$r = \frac{1}{1 - \frac{1}{2} + \frac{(\frac{1}{2})}{4}} = \frac{8}{5}$$

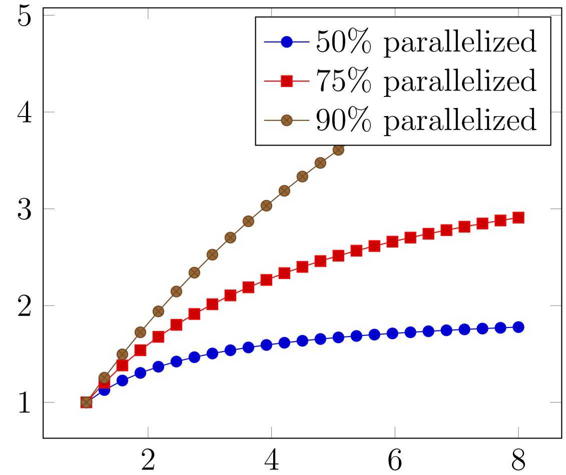
  algorithm is 60% faster.

Simple calculus:

$$\lim_{s\to\infty} \frac{1}{1 - p + \frac{p}{s}} = \frac{1}{1 - p + 0} = \frac{1}{1 - p}$$

Up to 100 cores:





Up to 8 cores:
Amdahl's law does not take into account concurrency overhead.

0.5 Mutex

pthread example – pre-C++11.

```c
/* mutex.c                                          */
/* to build on Linux, link with the pthread library: */
/* gcc mutex1.c -lpthread -DTHREADS=10               */

#include <stdio.h>   /* printf       */
#include <pthread.h> /* thread stuff */

/* #define THREADS 10 */

int Count = 0;
pthread_mutex_t count_mutex;

void *Increment( void *p )
{
    int i;
    for( i = 0; i < 1000000/THREADS; i++ ) {
        pthread_mutex_lock( &count_mutex );
        Count++;
        pthread_mutex_unlock( &count_mutex );
    }
    return 0;
}

int main( void )
{
    int i;
    pthread_t thread_id[THREADS];
    pthread_mutex_init( &count_mutex, NULL );
    for( i = 0; i < THREADS; i++ ) {
        pthread_create( &thread_id[i], 0, Increment, 0 );
    }
    for( i = 0; i < THREADS; i++ ) {
        pthread_join( thread_id[i], 0 );
    }
    pthread_mutex_destroy( &count_mutex );
    printf( "Count = %i\n", Count );
    return 0;
}
```

win32 example

```c
/* wmutex.c */
#include <windows.h>
#include <stdio.h>

struct ThreadParams {
    HANDLE mutex;
    int tid;
};

DWORD WINAPI ThreadFn( LPVOID p )
{
    DWORD result;
    int done = FALSE;
    struct ThreadParams *params = ( struct ThreadParams * )p;
    HANDLE mutex = params->mutex;
    int tid = params->tid;
    while( !done ) {
/*****************************************************************
 * DO NOT set the timeout to zero, unless your threads have some other
 * work to do. If the thread can do nothing until it obtains the mutex,
 * then don't keep checking that it is available. Use INFINITE instead
 * of 0 (zero) for the timeout. If you use 0, and don't obtain the
 * mutex immediately (the likely outcome) your program will be wasting
 * CPU cycles in a kind of busy wait loop. This is just an example
 * of how you would be able to do other work, if necessary.
 *
 * If you don't understand the above, use INFINITE and forget about it.
 *****************************************************************/
        result = WaitForSingleObject( mutex, 0L );
        switch( result ) {
                /* Acquired the mutex */
            case WAIT_OBJECT_0:
                printf( "ThreadId␣%i:␣acquired␣mutex\n", tid );
                done = TRUE;
                /* Pretend to do some work */
                printf( "ThreadId␣%i:␣working...\n", tid );
                Sleep( 1000 );
                if( !ReleaseMutex( mutex ) )
                    printf( "Error␣releasing␣mutex:␣%i\n", GetLastError() );
                printf( "ThreadId␣%i:␣released␣mutex\n", tid );
                break;
                /* The mutex wasn't available yet */
            case WAIT_TIMEOUT:
                /*printf("ThreadId %d: no mutex yet\n", tid);*/
                break;
        }
    }
    return TRUE;
```

```
}

int main( int argc, char **argv )
{
    int i;
    HANDLE *threads;
    struct ThreadParams* params;
    DWORD ThreadID;
    HANDLE mutex;
    int num_threads = 4;
    /* The user can specify the number of threads */
    if( argc > 1 )
        num_threads = atoi( argv[1] );
    /* Allocate memory for threads and parameters */
    threads = malloc( num_threads * sizeof( HANDLE ) );
    params = malloc( num_threads * sizeof( struct ThreadParams ) );
    /* Create mutex */
    mutex = CreateMutex( NULL, FALSE, NULL );
    if( mutex == NULL ) {
        printf( "Error␣creating␣mutex:␣%d\n", GetLastError() );
        return 1;
    }
    /* Create the threads */
    for( i = 0; i < num_threads; i++ ) {
        params[i].mutex = mutex;
        params[i].tid = i;
        threads[i] = CreateThread( NULL, 0, ThreadFn, &params[i], 0, &ThreadID );
        if( threads[i] == NULL ) {
            printf( "Error␣creating␣thread:␣%d\n", GetLastError() );
            return 1;
        }
    }
    /* Wait for each thread */
    WaitForMultipleObjects( num_threads, threads, TRUE, INFINITE );
    /* Release thread handles */
    for( i = 0; i < num_threads; i++ )
        CloseHandle( threads[i] );
    /* Release mutex */
    CloseHandle( mutex );
    /* Clean up memory */
    free( threads );
    free( params );
    return 0;
}
```

C++11 example

```
// note that actually synchronization is not required at all
// all threads write to different parts of array

#include <thread>
#include <mutex>

int julia( int x, int y )
{
    const double scale = 0.05; // maximum zoom is 2
    double jx = scale * (double)(DIM/2 - x)/(DIM/2);
    double jy = scale * (double)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    for ( int i=0; i<255; ++i ) {
        a = a * a + c;
        if (a.magnitude2() > 1000) return i;
    }

    return 255;
}

static std::mutex write_lock;

#ifdef GP1
void generate_pixel( unsigned char* red, unsigned char* green, unsigned char* blu
{
    int offset = x + y * DIM;
    int juliaValue = julia( x, y );

    write_lock.lock();
    red  [offset] = 0;
    green[offset] = juliaValue;
    blue [offset] = 255-juliaValue;
    write_lock.unlock();
}
#endif

#ifdef GP2
void generate_pixel( unsigned char* red, unsigned char* green, unsigned char* blu
{
    int offset = x + y * DIM;
    int juliaValue = julia( x, y );

    std::lock_guard<std::mutex> only_one_thread_writes_at_a_time( write_lock );
    red  [offset] = 0;
    green[offset] = juliaValue;
```

```
    blue [offset] = 255-juliaValue;
}
#endif

// thread
void generate_set( unsigned char* red, unsigned char* green, unsigned char* blue,
{
    for (int y=0; y<DIM; y++) {
        for (int x=start_x; x<DIM; x+=step) {
            generate_pixel( red, green, blue, x, y );
        }
    }
 }

int main( )
{
    // .....
    int const num_threads = 8;
    std::thread t[num_threads];

    //Launch a group of threads
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread( generate_set, red, green, blue, num_threads, i );
    }

    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }

    // .....
}
```

0.6 Semaphores

Semaphores were invented by Edsger Dijkstra – Dutch computer scientist.

Semaphore is a generalization of mutex, i.e. mutex is a semaphore with count 1 (**ALMOST**).

Semaphore has 3 operations

- initialization (takes an integer)
- wait (P)
- signal (V)

Thread that desires to enter a critical section executes `wait`, if semaphore count is not zero, it is allowed to enter. If semaphore was created with count n, up to n threads will be allowed in their critical section. The next thread will block on `wait` will another thread executes `signal`.

0.7 Basic syncronization patterns

Signaling: one thread sends a signal to another thread to indicate that something has happened. Or in other words require that a1 happens before b2:

```
sem = Semaphore(0)
```

`a1`	`b1`
`sem.signal()`	`sem.wait()`
`a2`	`b2`

Rendezvous:
Given 2 threads:

`a1`	`b1`
`a2`	`b2`

rendezvous requires that

- a1 happens before b2
- b1 happens before a2

(same as signalling, but symmetric)
One may quickly write something like this (wait-signal pair):

```
semA = Semaphore(0)
semB = Semaphore(0)
```

`a1`	`b1`
`semB.wait()`	`semA.wait()`
`semA.signal()`	`semB.signal()`
`a2`	`b2`

Unfortunately results in a guaranteed (and very popular) deadlock. Solution:
Correct solution:

```
semA = Semaphore(0)
semB = Semaphore(0)
```

a1	b1
semA.signal()	semB.signal()
semB.wait()	semA.wait()
a2	b2

Multiplex: allow multiple (up to a fix number N) of threads to execute critical section at the same time. Possibly the most common pattern involving semaphores:

```
sem = Semaphore(N) // max number of threads inside critical section
```

```
sem.wait()
critical point
sem.signal()
```

Barrier: rendezvous for more than 2 thread. The synchronization requirement is that no thread executes `critical point` until after all threads have executed `rendezvous`.

```
rendezvous
critical point
```

Attempt 1:

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
mutex.wait()
count = count +1
mutex.signal()

if ( count == n ) barrier.signal()

barrier.wait()

critical point
```

Rational: the first $n-1$ threads get blocked on `barrier.wait()`. When the n^{th} thread arrives it sends a signal to unblock.

Deadlock!

To solve it we need another `barrier.signal()` before critical point.

Attempt 2:

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
mutex.wait()
count = count +1
mutex.signal()

if ( count == n ) barrier.signal()

barrier.wait()
barrier.signal() // added

critical point
```

Now we have a rapid sequence of wait-signal that lets all thread to continue after the n^{th} thread arrives.

```
barrier.wait()
barrier.signal()
```

it's called **turnstile**.

Question: is it a problem if 2 threads send `signal` sequentially (without `wait` in between)?

Reusable barrier: in many applications the thread will be running a `while`-loop with a barrier inside. We need to implement logic that locks barrier for the next iteration. First idea is to use `count` variable – make sure it drops to 0 before we continue.

Attempt 1:

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
while (1) {
  rendezvous
  mutex.wait()
  count = count +1
  mutex.signal()

  if ( count == n ) barrier.signal()

  barrier.wait()
  barrier.signal()

  critical point

  mutex.wait()
  count = count -1
  mutex.signal()

  if ( count == 0 ) barrier.wait()
}
```

This solution is not correct. It is only correct when `signal` and `wait` from all threads alternate. Which is not guaranteed. 2 threads may execute `signal` in a sequence (see question from above) and that may let another thread to continue into the next iteration before all threads arrive to the barrier. Also 2 threads may may execute `wait` in a sequence which will result in a deadlock.

We may try to defend `signal` and `wait` by a mutex – reuse the one we are using for `count`:

Attempt 2:

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
while (1) {
  rendezvous
  mutex.wait()
  count = count +1
  if ( count == n ) barrier.signal() // moved
  mutex.signal()


  barrier.wait()
  barrier.signal()

  critical point

  mutex.wait()
  count = count -1
  if ( count == 0 ) barrier.wait() // moved
  mutex.signal()

}
```

BUT a thread can still go ahead by an iteration.

Attempt 3:
Think at home:

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

```
while (1) {
  rendezvous
  mutex.wait()
  count = count +1
  if ( count == n ) barrier.signal() // moved
  mutex.signal()


  barrier.wait()
  barrier.signal()

  critical point

  mutex.wait()
  count = count -1
  if ( count == n-1 ) barrier.wait() // moved
  mutex.signal()

}
```

Switching to $n-1$ so that the "first thread" decrements the counter sends the `wait`.

Attempt 4 - working solution

```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
barrier2 = Semaphore(1) // noticed the value
```

```
while (1) {
  rendezvous
  mutex.wait()
  count = count +1
  if ( count == n ) {
    barrier2.wait()
    barrier.signal()
  }
  mutex.signal()


  barrier.wait()
  barrier.signal()

  critical point

  mutex.wait()
  count = count -1
  if ( count == 0 ) {
    barrier.wait()
    barrier2.signal()
  }
  mutex.signal()

  barrier2.wait()
  barrier2.signal()

}
```

May be useful

- wrap barrier in a class
- consider `preloaded turnstile`: instead of the actual turnstile behavior when each thread lets the next thread in, make the first thread send all n signals. It cannot be done with mutexes, since 2 consecutive signals is the same as one. Use semaphores.

0.8 Producer-consumer

Very popular situation in event-driven programming: several threads called *producers* generate events which are placed into a buffer. Several other threads *consumers* are processing those events by removing them from the buffer.

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

Producer code	Consumer code
`event = wait_for_event()`	`items.wait()`
`mutex.wait()`	`mutex.wait()`
`buffer.push( event )`	`event = buffer.pop()`
`items.signal()`	`mutex.signal()`
`mutex.signal()`	`process( event )`

This is correct implementation, but it has a small problem – `items.signal` is inside

```
mutex.wait()
.....
items.signal()
mutex.signal()
```

which will cause several unnecessary context switches:

- producer `items.signal`
- consumer passes `items.wait` and immediately blocked by `mutex.wait`
- producer `mutex.signal`
- consumer passes `mutex.wait`

Remember that blocked `mutex.wait` is an expensive operation.
Solution:

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

Producer code	Consumer code
`event = wait_for_event()`	`items.wait()`
`mutex.wait()`	`mutex.wait()`
`buffer.push( event )`	`event = buffer.pop()`
`mutex.signal() // swapped`	`mutex.signal()`
`items.signal() // swapped`	`process( event )`

The above solution assumes infinite buffer. What if that is not possible?

Solution for producer-consumer problem with finite buffer:

```
mutex = Semaphore(1)
items = Semaphore(0)
space = Semaphore( buffer.size() ) // <<<<
```

```
Producer code                          Consumer code

event = wait_for_event()               items.wait()
space.wait() // <<<<                   mutex.wait()
mutex.wait()                           event = buffer.pop()
buffer.push( event )                   space.signal() // <<<<
mutex.signal()                         mutex.signal()
items.signal()                         process( event )
```

0.9 Readers-writers problem

There is a common resource (data-structure) that is read by some threads and written by others. Reads may be executed concurrently, but writes have to be executed with exclusive access.

```
roomEmpty = Semaphore(1)
readers_mutex = Semaphore(1)
int readers = 0
```

```
Writer code
===========
roomEmpty.wait()
critical section
roomEmpty.signal()
```

```
Reader code
===========
readers_mutex.wait()
++readers
if ( readers == 1 ) roomEmpty.wait()
readers_mutex.signal()

critical section

readers_mutex.wait()
--readers
if ( readers == 0 ) roomEmpty.signal()
readers_mutex.signal()
```

The above code uses *lightswitch* pattern, where mutex `roomEmpty` is the switch.

Note a problem: if readers are constantly coming keeping the counter `readers` positive, then writers will wait indefinitely. Such situation is called *starvation*.