Readers/Writers problem

There is a number of threads.
In order to ensure the integrity of the shared data being both read from and written to we need to allow:

- only one writer access to the data at a time
- if a writer is active there must be no active readers
- if no writer is active there can be multiple readers

We also need to make sure that no process misses out entirely.

Three types of solutions:

1. writer preferred - waiting writers go before waiting readers
2. reader preferred – waiting readers go before waiting writers
3. neither preferred - try to treat readers and writers fairly (a simple queue is not good enough we want parallel readers whenever possible)

Both 1 and 2 can lead to indefinite postponement.

Getting the program correct

Programming using low level constructs like semaphores is prone to mistakes.
What is wrong with this semaphore solution to the producer/consumer problem?

```
exclusive_access = Semaphore.new(1)
number_deposited = Semaphore.new(0)

shared_buffer = 0

producer = Thread.new do
   while true
        next_result = whatever
        exclusive_access.wait()
        shared_buffer = next_result
        number_deposited.signal()
        exclusive_access.signal()
   end
end

consumer = Thread.new do
   while true
        exclusive_access.wait()
        number_deposited.wait()
        next_result = shared_buffer
        exclusive_access.signal()
        puts next_result
   end
end
```


Bad programmers

Another popular problem is forgetting to unlock or signal.
We want an automatic (more or less) way of helping programmers lock and unlock.

Java, Ruby and other languages try to avoid or minimise problems by implementing a form of monitor.

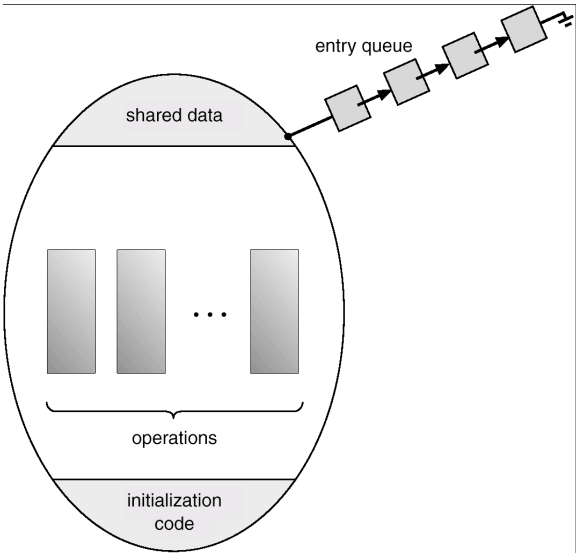
Monitors

Brinch Hansen (1973) Hoare (1974)

You can think of a monitor as an object which only allows one thread to be executing inside it.
It has:

- the shared resource - it can only be accessed by the monitor
- publicly accessible procedures - they do the work
- a queue to get in
- a scheduler - which thread gets access next
- local state - not visible externally except via access procedures
- initialization code
- condition variables

Monitors (cont.)



Example monitor

Here is an example in some Pseudo-code language which includes monitors:

```
monitor Account

 money = 0.00 # the shared resource

  def deposit(amount)
    money = money + amount
  end

  def withdraw(amount)
    if (amount < money)
      money = money - amount
      return true
    else
      return false
   end
  end

  def balance
    return money
  end

end
```


Condition variables

But sometimes our threads have to wait for some condition.

A condition variable is a queue which can hold threads. We have wait and signal operations on condition variables.

`conditionVariable.wait` puts the current thread to sleep on the corresponding queue

`conditionVariable.signal` wakes up one thread from the queue (if there are any waiting)

- No internal state is kept of how many signals and waits there have been.
- Simpler than the similar instructions on semaphores.

A signal with nothing waiting does nothing.

A wait always puts a thread to sleep.

e.g. condition variables

```
monitor SimpleBuffer

    def initialize
        buffer_free = true
        buffer = 0
        empty = new_condition_var
        full = new_condition_var
    end

    def insert(value)
           if !buffer_free
              empty.wait
           buffer = value
           buffer_free = false
           full.signal
    end

    def retrieve
           if buffer_free
              full.wait
           data = buffer
           buffer_free = true
           empty.signal
           return data
    end

end
```

But which thread runs?

But doesn't signal mean we have two threads running in the monitor?

Two choices:

- stop the thread which called signal
- don't start the new one until the current thread leaves the monitor

Usually we use the second answer but:

- the thread may signal on other condition variables as well and we have to make scheduling decisions
- it may also change the conditions again and the next thread shouldn't really run

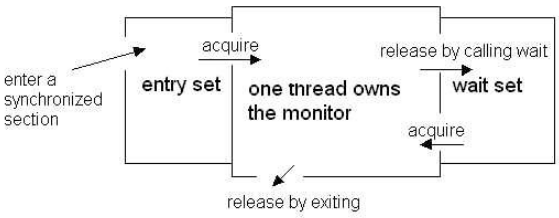
Java monitors

Java has a single lock variable per object (it also has one per class).

Each object also has a *wait set* associated with it (carefully not called a queue).

Synchronized methods must check this variable before allowing entry.

Synchronized blocks check the same variable.



Java monitors (cont.)

There is a count associated with each lock variable.

The count goes up every time a thread which owns the lock on that object calls a synchronized method or block on that object.

And it goes down when it leaves the method or block.

When the count gets to zero the thread exits the monitor and the lock is released.

```
...
   synchronized (anObject) {
    do things to the object;
   }
```


Java monitors are different

- `signal` is called `notify()`.
- It doesn't provide condition variables in the language (but now (1.5 and later) provides them as classes).
- `wait()` and `notify()` have a single set for the whole object, i.e. one condition variable.
- The object can have unsynchronized methods which are not private.
- Also fields which are not private. Not a good idea.
- after a `notify()` running threads run till they leave the `synchronized` area
- programmers are told to use a `while` loop with the conditional `wait`

Before next time

Read from the textbook

5.7.3 The Dining-Philosophers Problem

3.4 Interprocess Communication

3.5 Examples of IPC Systems

4.6.2 Signal Handling