Synchronization problems with semaphores

Lecture 4 of TDA383/DIT390 (Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg
SP3 2016/2017

Today's menu

Dining philosophers

Producer-consumer

Barriers

Readers-writers

A gallery of synchronization problems

In today's class, we go through several classical synchronization problems and solve them using threads and semaphores.

If you want to learn about many other synchronization problems and their solutions, check out "The little book of semaphores" by A. B. Downey available at `http://greenteapress.com/semaphores/`.

We will use pseudo-code, which simplifies the details of Java syntax and libraries but which can be turned into fully functioning code by adding boilerplate. On the course website you can download fully working implementations of some of the problems.

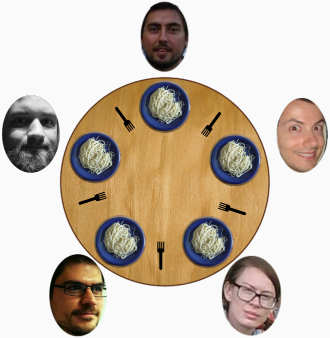
In particular, we occasionally annotate classes with invariants using the pseudo-code keyword **`invariant`**; **`invariant`** is not a valid Java keyword—that is why we highlight it in a different color—but we will use it to help make more explicit the behavior of classes.

Dining philosophers

The dining philosophers

The dining philosophers is a classic synchronization problem introduced by Dijkstra. It illustrates the problem of deadlocks using a colorful metaphor (by Hoare).

- Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers
- Each philosopher alternates between thinking (non-critical section) and eating (critical section)
- In order to eat, a philosopher needs to pick up the two forks that lie at the philopher's left and right sides
- Since the forks are shared, there is a synchronization problem between philosophers (threads)



Dining philosophers: the problem

```
interface Table {
    // philosopher k picks up forks
    void getForks(int k);
    // philosopher k releases forks
    void putForks(int k);
}
```

Dining philosophers problem: implement `Table` such that:

- forks are held exclusively by one philosopher at a time
- each philosopher only accesses adjacent forks

Properties that a good solution should have:

- support an arbitrary number of philosophers
- deadlock freedom
- starvation freedom
- reasonable efficiency: eating in parallel still possible

The philosophers

Each philosopher continuously alternate between thinking and eating; the table must guarantee proper synchronization when eating.

```
Table table; // table shared by all philosophers
```

philosopher$_k$

```
while (true) {
  think();             // think
  table.getForks(k); // wait for forks
  eat();               // eat
  table.putForks(k); // release forks
}
```

Left and right

For convenience, we introduce a consistent numbering scheme for forks and philosophers, in a way that it is easy to refer to the left or right fork of each philosopher.

```
// in classes implementing Table:

// fork to the left of philosopher k
public int left(int k) {
  return k;
}

// fork to the right of philosopher k
public int right(int k) {
  // N is the number of philosophers
  return (k + 1) % N;
}
```

Dining philosophers with locks and semaphores

We use semaphores to implement mutual exclusion when philosophers access the forks. In fact, we only need locks.

```
Lock[] forks = new Lock[N]; // array of locks
```

- one lock per fork
- `forks[i].lock()` to pick up fork `i`: `forks[i]` is held if fork `i` is held
- `forks[i].lock()` to put down fork `i`: `forks[i]` is available if fork `i` is available

Dining philosophers with semaphores: first attempt

In the first attempt, every philosopher picks up the left fork and then the right fork:

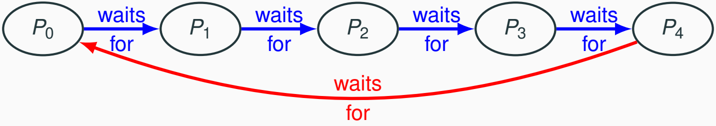
```
public class DeadTable implements Table {
  Lock[] forks = new Lock[N];   ← all forks initially available

  public void getForks(int k) {
    // pick up left fork
    forks[left(k)].lock();
    // pick up right fork
    forks[right(k)].lock();
  }

  public void putForks(int k) {
    // put down left fork
    forks[left(k)].unlock();
    // put down right fork
    forks[right(k)].unlock();
  }
```

Dining philosophers with semaphores: first attempt

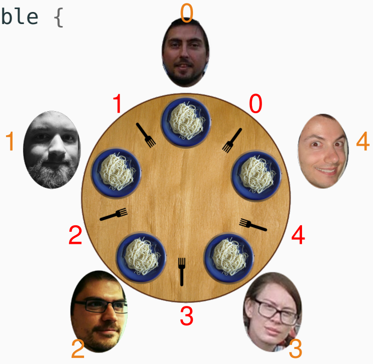
A deadlock may occur because of circular waiting:




```java
public class DeadTable implements Table {
  Lock[] forks = new Lock[N];

  public void getForks(int k) {
    // pick up left fork
    forks[left(k)].lock();
    // pick up right fork
    forks[right(k)].lock();
  }
```

if all philosophers hold left fork: deadlock!

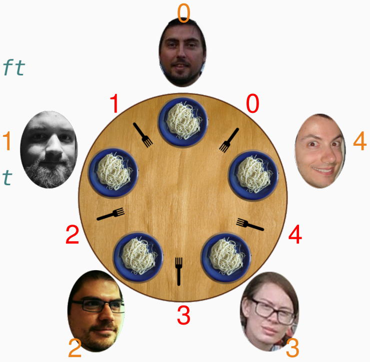


Dining philosophers solution 1: breaking the symmetry

Having one philosopher pick up forks in a different order than the others is sufficient to break the symmetry, and thus to avoid deadlock.

```java
public class AsymmetricTable implements Table {
  Lock[] forks = new Lock[N];

  public void getForks(int k) {
    if (k == N) { // right before left
      forks[right(k)].lock();
      forks[left(k)].lock();
    } else {     // left before right
      forks[left(k)].lock();
      forks[right(k)].lock();
    }
  }
  // putForks as in DeadTable
```



Breaking symmetry to avoid deadlock

Breaking the symmetry is a general strategy to avoid deadlock when acquiring multiple shared resources:

- assign a total order between the shared resources $R_0 < R_1 < \cdots < R_M$
- a thread can try to obtain resource R_i, with $i > j$, only after it has successfully obtained resource R_j

Recall the Coffman conditions in a previous lecture: circular wait is one of the most common conditions for a deadlock to occur.

Dining philosophers solution 2: bounding resources

Limiting the number of philosophers active at the table to `M < N` ensures that there are enough resources for everyone at the table, thus avoiding deadlock.

```
public class SeatingTable implements Table {
  Lock[] forks = new Lock[N];
  Semaphore seats = new Semaphore(M); // # available seats
```

```
  public void getForks(int k) {
    // get a seat
    seats.down();
    // pick up left fork
    forks[left(k)].lock();
    // pick up right fork
    forks[right(k)].lock();
  }
```

```
  public void putForks(int k) {
    // put down left fork
    forks[left(k)].unlock();
    // put down right fork
    forks[right(k)].unlock();
    // leave seat
    seats.up();
  }
```

Starvation-free philosophers

The two solutions to the dining philosophers problem also guarantee freedom from starvation, under the assumption that locks/semaphores (and scheduling) are fair.

In the asymmetric solution (`AsymmetricTable`):

- if a philosopher *P* waits for a fork *k*, *P* gets the fork as soon as *P*'s neighbor holding fork *k* releases it
- *P*'s neighbor eventually releases fork *k* because there are no deadlocks

In the bounded-resource solution (`SeatingTable`):

- at most `M` philosophers are active at the table
- the other `N - M` philosophers are waiting on `seats.down()`
- the first of the `M` philosophers that finishes eating releases a seat
- the philosopher *P* that has been waiting on `seats.down` proceeds
- similarly to the asymmetric solution, *P* also eventually gets the forks

Producer-consumer

Producer-consumer: overview

Producers and consumer exchange items through a shared buffer:

- producers asynchronously produce items and store them in the buffer
- consumers asynchronously consume items after taking them out of the buffer

Producer-consumer: the problem

```
interface Buffer<T> {
        // add item to buffer; block if full
        void put(T item);

        // remove item from buffer; block if empty
        T get();

        // number of items in buffer
        int count();
}
```

Producer-consumer problem: implement `Buffer` such that:

- producers and consumers access the buffer in mutual exclusion
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

Producer-consumer: desired properties

Producer-consumer problem: implement `Buffer` such that:

- producers and consumer access the buffer in mutual exclusion
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

Other properties that a good solution should have:

- support an arbitrary number of producers and consumers
- deadlock freedom
- starvation freedom

Producers and consumers

Producers and consumers continuously and asynchronously access the buffer, which must guarantee proper synchronization.

```
Buffer<Item> buffer;
```

producer$_n$

```
while (true) {
  // create a new item
  Item item = produce();
  buffer.put(item);
}
```

consumer$_m$

```
while (true) {
  Item item = buffer.get();
  // do something with ‘item’
  consume(item);
}
```

Unbounded shared buffer

```
public class UnboundedBuffer<T> implements Buffer<T> {
  Lock lock = new Lock(); // for exclusive access to buffer
  Semaphore nItems = new Semaphore(0); // number of items in buffer
  Collection storage = ...; // any collection (list, set, ...)
  invariant { storage.count() == nItems.count(); }

  public void put(T item) {
    lock.lock(); // lock
    // store item
    storage.add(item);
    nItems.up();   // update nItems
    lock.unlock(); // release
  }

  public int count() {
    return nItems.count(); // locking here?
  }

  public T get() {
    // wait until nItems > 0
    nItems.down();
    lock.lock(); // lock
    // retrieve item
    T item = storage.remove();
    lock.unlock(); // release
    return item;
  }
```

Buffer: method `put`

```java
public void put(T item) {
  lock.lock(); // lock
  // store item
  storage.add(item);
  // update nItems
  nItems.up();
  lock.unlock(); // release
}
```

Buffer: method `put`

```
public void put(T item) {
  lock.lock(); // lock
  // store item
  storage.add(item);
  // update nItems
  nItems.up();
  lock.unlock(); // release
}
```

signal to consumers waiting in `get` that they can proceed

Can we execute `up` after `unlock`?

Buffer: method put

```
public void put(T item) {
  lock.lock(); // lock
  // store item
  storage.add(item);
  // update nItems
  nItems.up();
  lock.unlock(); // release
}
```

signal to consumers waiting in `get` that they can proceed

Can we execute `up` after `unlock`?

Executing `up` after `unlock`:

- no effects on other threads executing `put`: they only wait for `lock`
- if a thread is waiting for `nItems > 0` in `get`: it does not have to wait for `lock` after it can continue
- if a thread is waiting for the lock in `get`: it may return with the buffer in a (temporarily) inconsistent state (broken invariant, but <u>benign</u> because temporary)

Executing `up` after `unlock`

```java
1  public void put(T item) {
2    lock.lock();
3    storage.add(item);
4    lock.unlock();
5    nItems.up();
6  }
```

```java
public T get() {                  7
  nItems.down();                  8
  lock.lock();                    9
  T item = storage.remove();      10
  lock.unlock();                  11
  return item;                    12
}                                 13
```

#	producer `put`	consumer `get`	SHARED
+1	$pc_t : 3$	$pc_u : 8$	`nItems`: 1 `buffer`: $\langle x \rangle$
+2	$pc_t : 3$	$pc_u : 9$	`nItems`: 0 `buffer`: $\langle x \rangle$
+3	$pc_t : 4$	$pc_u : 9$	`nItems`: 0 `buffer`: $\langle x, y \rangle$
+4	$pc_t : 5$	$pc_u : 9$	`nItems`: 0 `buffer`: $\langle x, y \rangle$
+5	$pc_t : 5$	$pc_u : 10$	`nItems`: 0 `buffer`: $\langle x, y \rangle$
+6	$pc_t : 5$	$pc_u : 11$	`nItems`: 0 `buffer`: $\langle y \rangle$
+7	$pc_t : 5$	$pc_u : 12$	`nItems`: 0 `buffer`: $\langle y \rangle$
+8	$pc_t : 5$	done	`nItems`: 0 `buffer`: $\langle y \rangle$
+9	done	done	`nItems`: 1 `buffer`: $\langle y \rangle$

Buffer: method `get`

```
public T get() {
  nItems.down();                   // wait until nItems > 0
  lock.lock();                     // lock
  T item = storage.remove();       // retrieve item
  lock.unlock();                   // release
  return item;
}
```

What happens if another thread gets the lock just after the current threads has decremented the semaphore `nItems`?

- if the other thread is a producer, it does not matter: as soon as `get` resumes execution, there will be one element in storage to remove
- if the other thread is a consumer, it must have synchronized with the current thread on `nItems.down()`, and the order of removal of elements from the buffer does not matter

Buffer: method `get`

```
public T get() {
  nItems.down();
  lock.lock();
  T item = storage.remove();
  lock.unlock();
  return item;
}
```

Can we execute `down` after `lock`?

Buffer: method `get`

```java
public T get() {
  nItems.down();
  lock.lock();
  T item = storage.remove();
  lock.unlock();
  return item;
}
```

Can we execute `down` after `lock`?

Executing `down` after `lock`:

- if the buffer is empty when locking, there is a deadlock!

Bounded shared buffer

```
public class BoundedBuffer<T> implements Buffer<T> {
 Lock lock = new Lock(); // for exclusive access to buffer
 Semaphore nItems = new Semaphore(0); // # items in buffer
 Semaphore nFree = new Semaphore(N);  // # free slots in buffer
 Collection storage = ...; // any collection (list, set, ...)
 invariant { storage.count()
             == nItems.count() == N - nFree.count(); }
```

```
public void put(T item) {
 // wait until nFree > 0
 nFree.down();
 lock.lock(); // lock
 // store item
 storage.add(item);
 nItems.up();   // update nItems
 lock.unlock(); // release
}
```

```
public T get() {
 // wait until nItems > 0
 nItems.down();
 lock.lock(); // lock
 // retrieve item
 T item = storage.remove();
 nFree.up(); // update nFree
 lock.unlock(); // release
 return item;
}
```

Bounded shared buffer

```
public class BoundedBuffer<T> implements Buffer<T> {
 Lock lock = new Lock(); // for exclusive access to buffer
 Semaphore nItems = new Semaphore(0); // # items in buffer
 Semaphore nFree = new Semaphore(N);  // # free slots in buffer
 Collection storage = ...; // any collection (list, set, ...)
 invariant { storage.count()
             == nItems.count() == N - nFree.count(); }
```

size of buffer

```
public void put(T item) {
 // wait until nFree > 0
 nFree.down();
 lock.lock(); // lock
 // store item
 storage.add(item);
 nItems.up();   // update nItems
 lock.unlock(); // release
}
```

```
public T get() {
 // wait until nItems > 0
 nItems.down();
 lock.lock(); // lock
 // retrieve item
 T item = storage.remove();
 nFree.up(); // update nFree
 lock.unlock(); // release
 return item;
}
```

Bounded shared buffer

```java
public class BoundedBuffer<T> implements Buffer<T> {
  Lock lock = new Lock(); // for exclusive access to buffer
  Semaphore nItems = new Semaphore(0); // # items in buffer
  Semaphore nFree = new Semaphore(N);  // # free slots in buffer
  Collection storage = ...; // any collection (list, set, ...)
  invariant { storage.count()
              == nItems.count() == N - nFree.count(); }
```

```java
public void put(T item) {
  // wait until nFree > 0
  nFree.down();
  lock.lock(); // lock
  // store item
  storage.add(item);
  nItems.up();   // update nItems
  lock.unlock(); // release
}
```

may deadlock if swapped

OK to swap

```java
public T get() {
  // wait until nItems > 0
  nItems.down();
  lock.lock(); // lock
  // retrieve item
  T item = storage.remove();
  nFree.up(); // update nFree
  lock.unlock(); // release
  return item;
}
```

may deadlock if swapped

OK to swap

Waiting on multiple conditions?

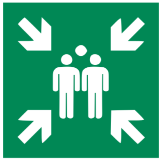
The operations offered by semaphores do not support waiting on multiple conditions (not empty and not full in our case) using one semaphore:

```
// wait until there is space in the buffer
while (!(nItems.count() < N)) {};
// the buffer may be full again when locking!
lock.lock(); // lock
// store item
storage.add(item);
nItems.up();   // update nItems
lock.unlock(); // release
```

Barriers

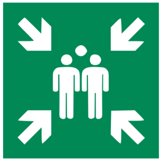
Barriers (also called rendezvous)

A barrier is a form of synchronization where there is a point (the barrier) in a program's execution that all threads in a group have to reach before any of them is allowed to continue



Barriers (also called rendezvous)

A barrier is a form of synchronization where there is a point (the barrier) in a program's execution that all threads in a group have to reach before any of them is allowed to continue



A solution to the barrier synchronization problem for 2 threads using binary semaphores.

```
Semaphore[] done = {new Semaphore(0), new Semaphore(0)};
```

t_0

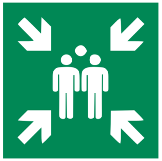
```
// code before barrier
done[t0].up();   // t done
done[t1].down(); // wait u
// code after barrier
```

t_1

```
// code before barrier
done[t1].up();   // u done
done[t0].down(); // wait t
// code after barrier
```

Barriers (also called rendezvous)

> A barrier is a form of synchronization where there is a point (the barrier) in a program's execution that all threads in a group have to reach before any of them is allowed to continue



A solution to the barrier synchronization problem for 2 threads using binary semaphores.

capacity 0 forces up before first down

```
Semaphore[] done = {new Semaphore(0), new Semaphore(0)};
```

t_0	t_1
`// code before barrier`	`// code before barrier`
`done[`t_0`].up();   // t done`	`done[`t_1`].up();   // u done`
`done[`t_1`].down(); // wait u`	`done[`t_0`].down(); // wait t`
`// code after barrier`	`// code after barrier`

up done unconditionally

down waits until the other thread has reached the barrier

Barriers: variant

The solution still works if t_0 performs `down` before `up` — or, symmetrically, if t_1 does the same.

```
Semaphore[] done = new Semaphore(0), new Semaphore(0);
```

t_0	t_1
`// code before barrier`	`// code before barrier`
`done[`t_1`].down(); // wait u`	`done[`t_1`].up();   // u done`
`done[`t_0`].up();   // t done`	`done[`t_0`].down(); // wait t`
`// code after barrier`	`// code after barrier`

This solution is, however, a bit less efficient: the last thread to reach the barrier has to stop and yield to the other (one more context switch).

Barriers: deadlock!

The solution deadlocks if both t_0 and t_1 perform `down` before `up`.

```
Semaphore[] done = new Semaphore(0), new Semaphore(0);
```

t_0	t_1
`// code before barrier`	`// code before barrier`
`done[`t_1`].down(); // wait u`	`done[`t_0`].down(); // wait t`
`done[`t_0`].up();   // t done`	`done[`t_1`].up();   // u done`
`// code after barrier`	`// code after barrier`

There is a circular waiting, because no thread has a chance to signal to the other that it has reached the barrier.

Barriers with n threads

Keeping track of n threads reaching the barrier:

- `nDone`: number of threads that have reached the barrier
- `lock`: to update `nDone` atomically
- `open`: to release the waiting threads ("opening the barrier")

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

thread t_k

total number of expected threads

```
// code before barrier
lock.lock();                    // lock nDone
nDone = nDone + 1;              // I'm done
if (nDone == n) open.up();      // I'm the last: we can go!
lock.unlock();                  // unlock nDone
open.down();                    // proceed when possible
open.up();                      // let the next one go
// code after barrier
```

Barriers with n threads: variant

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

thread t_k

```
// code before barrier
lock.lock();                    // lock nDone
nDone = nDone + 1;              // I'm done
lock.unlock();                  // unlock nDone
if (nDone == n) open.up();      // I'm the last: we can go!
open.down();                    // proceed when possible
open.up();                      // let the next one go
// code after barrier
```

can we open the barrier after `unlock`?

Barriers with *n* threads: variant

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

thread t_k

```
// code before barrier
lock.lock();                   // lock nDone
nDone = nDone + 1;             // I'm done
lock.unlock();                 // unlock nDone
if (nDone == n) open.up();     // I'm the last: we can go!
open.down();                   // proceed when possible
open.up();                     // let the next one go
// code after barrier
```

can we open the barrier after `unlock`?

- in general, reading a shared variable outside a lock may give an inconsistent value
- in this case, however, only the last thread can read `nDone == n` because `nDone` is only incremented

Barriers with n threads: variant

Signaling after unlocking follows the rule of thumb of minimizing the operations under lock (provided it does not affect correctness ☺).

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

thread t_k

```
// code before barrier
lock.lock();                // lock nDone
nDone = nDone + 1;          // I'm done
lock.unlock();              // unlock nDone
if (nDone == n) open.up(); // I'm the last: we can go!
open.down();                // proceed when possible
open.up();                  // let the next one go
// code after barrier
```

such pairs of wait/signal are called turnstiles

Reusable barriers

```
interface Barrier {
  // block until expect() threads have reached barrier
  void wait();

  // number of threads expected at the barrier
  int expect();
}
```

Reusable barrier: implement `Barrier` such that:

- a thread blocks on `wait` until all threads have reached the barrier
- after `expect()` threads have executed `wait`, the barrier is closed again

Threads at a reusable barrier

Threads continuously approach the barrier, which must guarantee that they synchronize each access.

```
Barrier barrier = new Barrier(n); // barrier for n threads
```

thread$_k$

```
while (true) {
  // code before barrier
  barrier.wait();   // synchronize at barrier
  // code after barrier
}
```

Reusable barriers: first attempt

```java
public class NonBarrier1 implements Barrier {
   int nDone = 0; // number of done threads
   Semaphore open = new Semaphore(0);
   final int n;

   // initialize barrier for ‘n’ threads
   NonBarrier1(int n) {
     this.n = n;
   }

   // number of threads expected at the barrier
   int expect() {
     return n;
   }
   // continues in the next slide
```

Reusable barriers: first attempt (cont’d)

```
public class NonBarrier implements Barrier {
   int nDone = 0; // number of done threads
   Semaphore open = new Semaphore(0);
   final int n;

   public void wait() {
     synchronized(this) { nDone += 1; } // I’m done
     if (nDone == n) open.up();  // I’m the last arrived:
                                 // we can go!
     open.down()                 // proceed when possible
     open.up()                   // let the next one go
     synchronized(this) { nDone -= 1; }  // I’ve gone through
     if (nDone == 0) open.down(); // I’m the last through:
   }                              // close barrier!
```

Reusable barriers: first attempt (cont'd)

```java
public class NonBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore open = new Semaphore(0);
  final int n;

  public void wait() {
    synchronized(this) { nDone += 1; } // I'm done
    if (nDone == n) open.up();  // I'm the last arrived:
                                // we can go!
    open.down()                 // proceed when possible
    open.up()                   // let the next one go
    synchronized(this) { nDone -= 1; }  // I've gone through
    if (nDone == 0) open.down(); // I'm the last through:
  }                              // close barrier!
```

What if *n* threads block here until `nDone == n`?

Reusable barriers: first attempt (cont'd)

```java
public class NonBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore open = new Semaphore(0);
  final int n;

  public void wait() {
    synchronized(this) { nDone += 1; } // I'm done
    if (nDone == n) open.up();  // I'm the last arrived:
                                // we can go!
    open.down()                 // proceed when possible
    open.up()                   // let the next one go
    synchronized(this) { nDone -= 1; }  // I've gone through
    if (nDone == 0) open.down(); // I'm the last through:
  }                              // close barrier!
```

What if *n* threads block here until `nDone == n`?

More than one thread may open the barrier (the first `open.up`): this was not a problem in the non-reusable version, but now some threads may be executing `wait` again before the barrier is closed again!

Reusable barriers: first attempt (cont'd)

```java
public class NonBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore open = new Semaphore(0);
  final int n;

  public void wait() {
    synchronized(this) { nDone += 1; } // I'm done
    if (nDone == n) open.up();  // I'm the last arrived:
                                // we can go!
    open.down()                 // proceed when possible
    open.up()                   // let the next one go
    synchronized(this) { nDone -= 1; }  // I've gone through
    if (nDone == 0) open.down(); // I'm the last through:
  }                              // close barrier!
```

What if *n* threads block here until `nDone == 0`?

Reusable barriers: first attempt (cont'd)

```
public class NonBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore open = new Semaphore(0);
  final int n;

  public void wait() {
    synchronized(this) { nDone += 1; } // I'm done
    if (nDone == n) open.up();  // I'm the last arrived:
                                // we can go!
    open.down()                 // proceed when possible
    open.up()                   // let the next one go
    synchronized(this) { nDone -= 1; }  // I've gone through
    if (nDone == 0) open.down(); // I'm the last through:
  }                              // close barrier!
```

What if *n* threads block here until `nDone == 0`?

More than one thread may try to close the barrier (the last `open.down`): deadlock!

Reusable barriers: second attempt

```java
public class NonBarrier2 implements Barrier {
   // same variables as in NonBarrier1

   public void wait() {
     synchronized(this)
     { nDone += 1;                          // I’m done
       if (nDone == n) open.up(); }         // open barrier
     open.down()                            // proceed when possible
     open.up()                              // let the next one go
     synchronized(this)
     { nDone -= 1;                          // I’ve gone through
       if (nDone == 0) open.down(); } // close barrier
}
```

Reusable barriers: second attempt

```
public class NonBarrier2 implements Barrier {
   // same variables as in NonBarrier1

   public void wait() {
     synchronized(this)
     { nDone += 1;                          // I'm done
       if (nDone == n) open.up(); }         // open barrier
     open.down()                            // proceed when possible
     open.up()                              // let the next one go
     synchronized(this)
     { nDone -= 1;                          // I've gone through
       if (nDone == 0) open.down(); }       // close barrier
}
```

Now multiple signaling is not possible. But a fast thread may race through the whole method, and re-enter it before the barrier has been closed, thus getting ahead of the slower threads — which are still in the previous iteration of the barrier.

Reusable barriers: second attempt (cont'd)

A fast thread may race through the whole method, and re-enter it before the barrier has been closed, thus getting ahead of the slower threads — which are still in the previous iteration of the barrier. This is not prevented by strong semaphores: it occurs because the last thread through leaves the gate open (calls `open.up()`)

```
1 public class NonBarrier2 {
2  public void wait() {
3   synchronized(this)
4   {nDone += 1;
5    if (nDone == n) open.up();}
6   open.down()
7   open.up()
8   synchronized(this)
9   {nDone -= 1;
10   if (nDone == 0) open.down();}
11 }
```

(a) All n threads are at 8, with `open.count() == 1`

(b) The fastest thread t_f completes `wait` and re-enters it with `nDone == n - 1`

(c) Thread t_f reaches 6 with `nDone == n`, which it can execute because `open.count() > 0`

(d) Thread t_f reaches 8 again, but it is one iteration ahead of all other threads!

Reusable barriers: correct solution

```java
public class SemaphoreBarrier implements Barrier {
   int nDone = 0; // number of done threads
   final int n;

   // initialize barrier for ‘n’ threads
   SemaphoreBarrier(int n) {
     this.n = n;
   }

   // number of threads expected at the barrier
   int expect() {
     return n;
   }
   // continues in the next slide
```

Reusable barriers: correct solution

```
public class SemaphoreBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore gate1 = new Semaphore(0);  // level-1 gate
  Semaphore gate2 = new Semaphore(1);  // level-2 gate
```

```
void approach() {
  synchronized (this) {
   nDone += 1;       // arrived
   if (nDone == n)   // if last in:
   { gate1.up();     // open gate1
     gate2.down(); } // close gate2
  }
  gate1.down(); // pass gate1
  gate1.up();   // let next pass
}
```

```
void leave() {
  synchronized (this) {
   nDone -= 1;       // going out
   if (nDone == 0)   // if last out:
   { gate2.up();     // open gate2
     gate1.down(); } // close gate1
  }
  gate2.down(); // pass gate2
  gate2.up();   // let next pass
}
```

```
public void wait() { approach(); leave(); }
```

Reusable barriers: improved solution

If the semaphores support adding n to the counter at once, we can write a barrier with fewer semaphore accesses.

both gates initially closed

```
public class NSemaphoreBarrier extends SemaphoreBarrier {
   Semaphore gate1 = new Semaphore(0); // level-1 gate
   Semaphore gate2 = new Semaphore(0); // level-2 gate
```

```
void approach() {
  synchronized (this) {
   nDone += 1;
   if (nDone == n)
     gate1.up(n);   // open gate1 for n threads
  }
  gate1.down(); // pass gate1
  // last thread here closes gate1
}
```

```
void leave() {
  synchronized (this) {
   nDone -= 1;
   if (nDone == 0)
     gate2.up(n);   // open gate2 for n threads
  }
  gate2.down();
  // last thread here closes gate2
}
```

Java semaphores support adding n to counter (`release(n)`). Anyway, `up(n)` need not be atomic, so we can also implement it with a loop.

Readers-writers

Readers-writers: overview

Readers and writers concurrently access shared data:

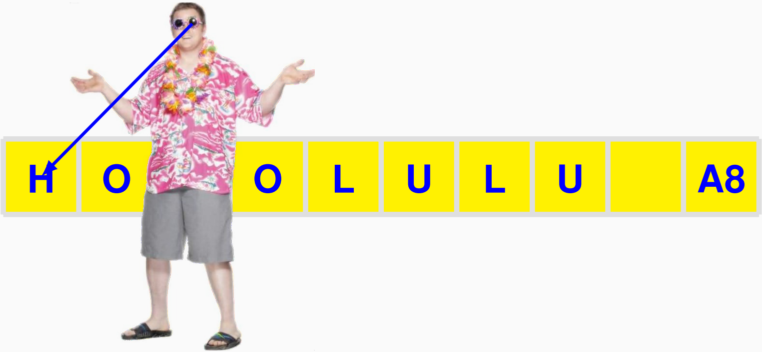
- readers may execute concurrently with other readers, but need to exclude writers
- writers need to exclude both readers and other writers

The problem captures situations common in databases, filesystems, and other situations where accesses to shared data may be inconsistent.

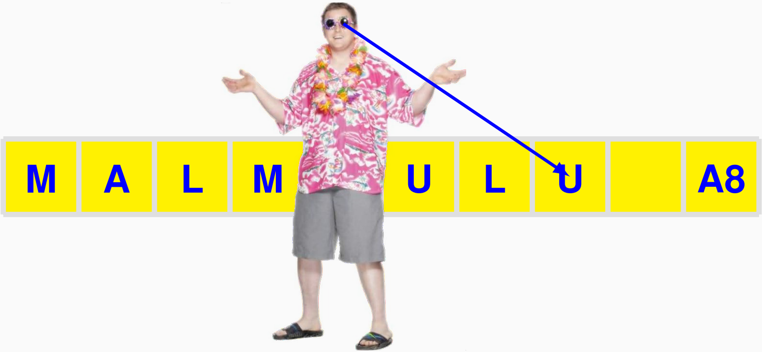
What’s the gate for the flight to Honolulu?

H	O	N	O	L	U	L	U		A8

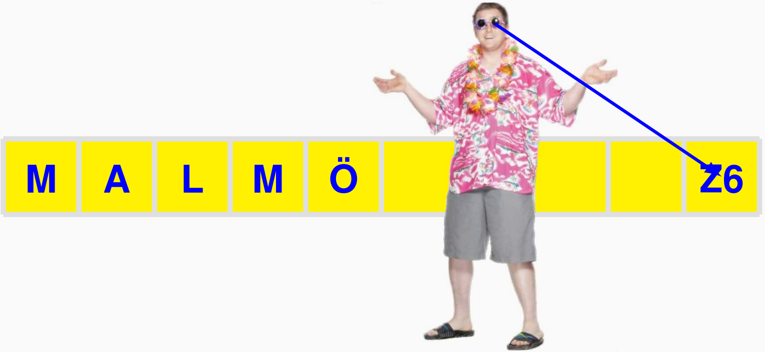
What’s the gate for the flight to Honolulu?



What’s the gate for the flight to Honolulu?



What’s the gate for the flight to Honolulu?



Readers-writers: the problem

```
interface Board<T> {
        // write message 'msg' to board
        void write(T msg);
        // read current message on board
        T read();
}
```

Readers-writers problem: implement `Board` data structure such that:

- multiple reader can operate concurrently
- each writer has exclusive access

Invariant: $\#\text{WRITERS} = 0 \lor (\#\text{WRITERS} = 1 \land \#\text{READERS} = 0)$

Other properties that a good solution should have:

- support an arbitrary number of readers and writers
- no starvation of readers or writers

Readers and writers

Readers and writers continuously and asynchronously try to access the board, which must guarantee proper synchronization.

```
Board<Message> board;
```

reader_n

```
while (true) {
  // read message from board
  Message msg = board.read();
  // do something with ‘msg’
  process(msg);
}
```

writer_m

```
while (true) {
  // create a new message
  Message msg = create();
  // write ‘msg’ to board
  board.write(msg);
}
```

Readers-writers board: `write`

```
public class SyncBoard<T> implements Board<T> {
  int nReaders = 0; // # readers on board
  Lock lock = new Lock(); // for exclusive access to nReaders
  Semaphore empty = new Semaphore(1); // 1 iff no active threads
  T message; // current message
```

```
public void write(T msg) {
  // get exclusive access
  empty.down();
  message = msg; // write (cs)
  // release board
  empty.up();
}
```

```
invariant { nReaders == 0
     ⇔ empty.count() == 1 }
```

Readers-writers board: read

```java
public class SyncBoard<T> implements Board<T> {
  int nReaders = 0; // # readers on board
  Lock lock = new Lock(); // for exclusive access to nReaders
  Semaphore empty = new Semaphore(1); // 1 iff no active threads
  T message; // current message

 public T read() {
   lock.lock();                         // lock to update nReaders
   if (nReaders == 0) empty.down();// if first reader, set not empty
   nReaders += 1;                       // update active readers
   lock.unlock();                       // release lock to nReaders
   T msg = message;                     // read (critical section)
   lock.lock();                         // lock to update nReaders
   nReaders -= 1;                       // update active readers
   if (nReaders == 0) empty.up();  // if last reader, set empty
   lock.unlock();                       // release lock to nReaders
   return msg;
 }
```

Properties of the readers-writers solution

We can check the following properties of the solution:

- `empty` is a binary semaphore
- when a writer is running, no other reader can run
- one reader waiting for a writer to finish also locks out other readers
- a reader signals “empty” only when it is the last reader to leave the board
- deadlock is not possible (no circular waiting)

However, writers can starve: as long as readers come and go with at least one reader always active, writers are shut out of the board.

Readers-writers board without starvation

```java
public class FairBoard<T> extends SyncBoard<T> {
  // held by the next thread to go
  Semaphore baton = new Semaphore(1); // binary semaphore
```

```java
public T read() {
  // wait for my turn
  baton.down();
  // release a waiting thread
  baton.up();
  // read() as in SyncBoard
  return super.read()
}
```

```java
public void write(T msg) {
  // wait for my turn
  baton.down();
  // write() as in SyncBoard
  super.write(msg);
  // release a waiting thread
  baton.up();
}
```

Readers-writers board without starvation

```
public class FairBoard<T> extends SyncBoard<T> {
  // held by the next thread to go
  Semaphore baton = new Semaphore(1); // binary semaphore
```

```
public T read() {
  // wait for my turn
  baton.down();
  // release a waiting thread
  baton.up();
  // read() as in SyncBoard
  return super.read()
}
```

```
public void write(T msg) {
  // wait for my turn
  baton.down();
  // write() as in SyncBoard
  super.write(msg);
  // release a waiting thread
  baton.up();
}
```

Now writers do not starve: suppose a writer is waiting that all active readers leave: it waits on `empty.down()` while holding the `baton`. If new readers arrive, they are shut out waiting for the `baton`. As soon as the active readers terminate and leave, the writer is signaled `empty`, and thus it gets exclusive access to the board.

Readers-writers with priorities

The starvation free solution we have presented gives all threads the same priority: assuming a fair scheduler, writers and readers take turn as they try to access the board.

In some applications it might be preferable to enforce difference priorities:

- $R = W$: readers and writers have the same priority (as in `FairBoard`)
- $R > W$: readers have higher priority than writers (as in `SyncBoard`)
- $W > R$: writers have higher priority than readers