Synchronization problems with semaphores

Lecture 4 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman

Chalmers University of Technology | University of Gothenburg

SP1 2020/2021

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Synchronization problems with semaphores

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.

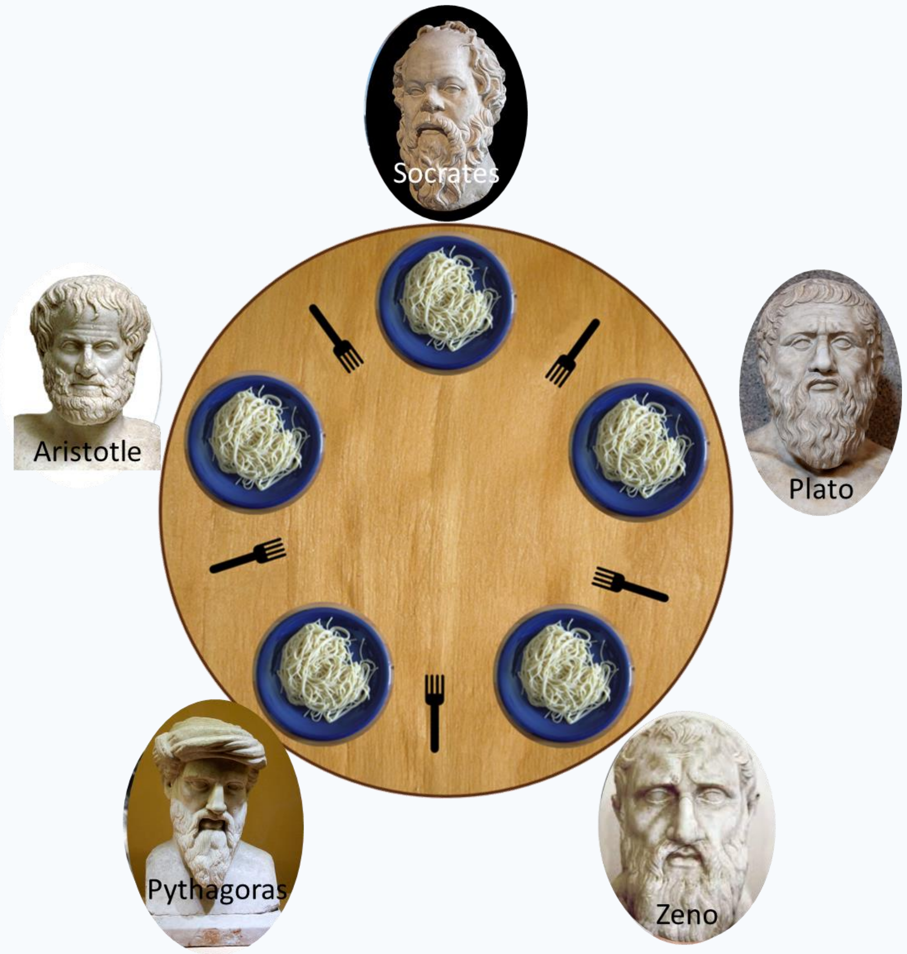
Recall that we occasionally annotate classes with *invariants* using the pseudo-code keyword invariant; invariant; is not a valid Java keyword – that is why we highlight it in a different color – but we will use it to help make more explicit the behavior of classes. We also use at(i) or at(i,j) to indicate the number of threads that are at a location or range of locations. That's not Java either.

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 to the philopher's left and right.
- 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

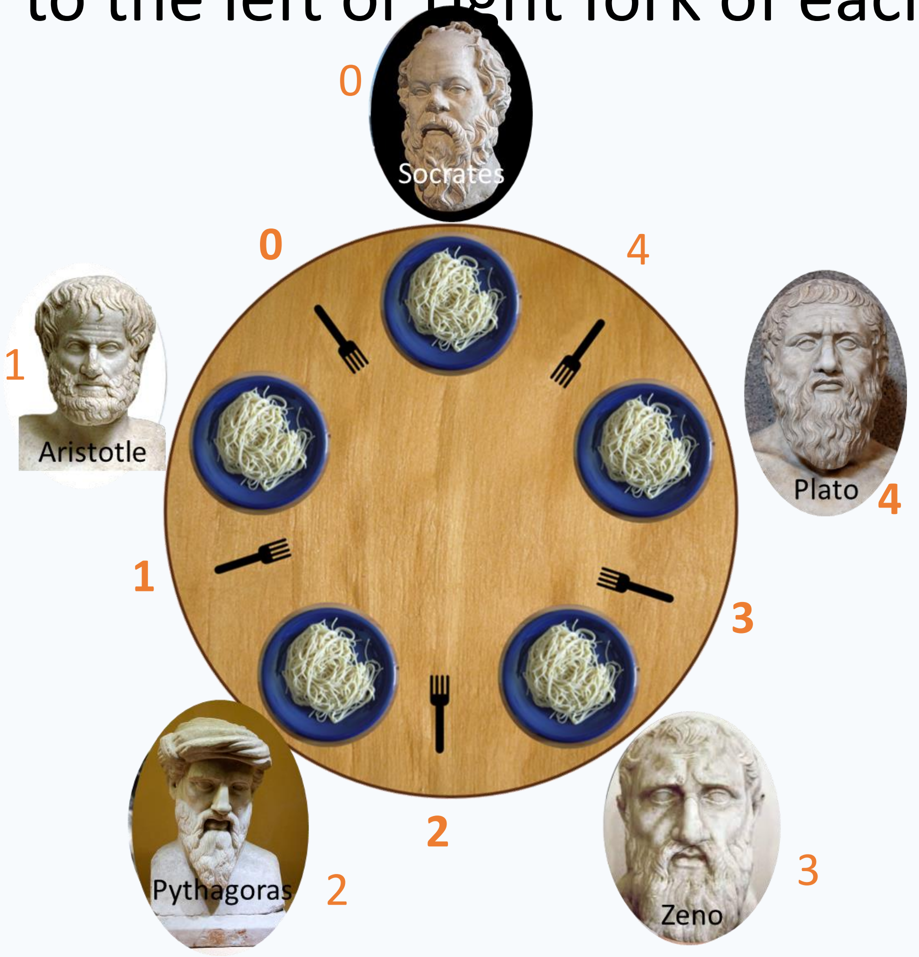
For convenience, we introduce a consistent numbering scheme for forks and philosophers, in a way that it is easy to refer to the left or right fork of each philosopher.

```java
// 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

First solution needs only 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].unlock()` 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];
```

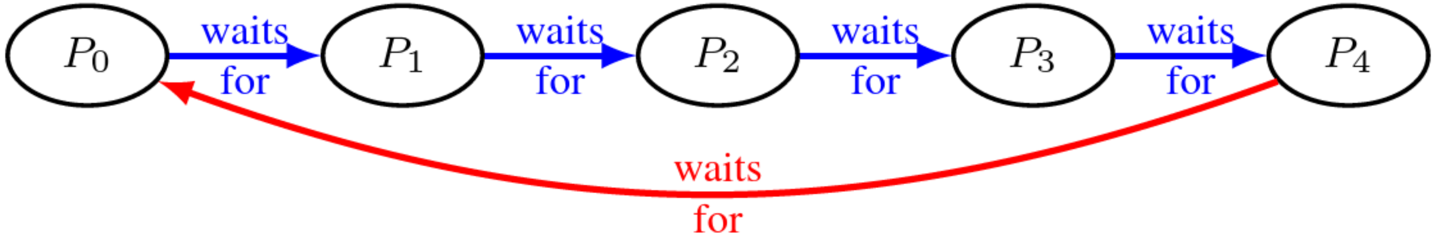
← all forks initially avaliable

```
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:




```
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

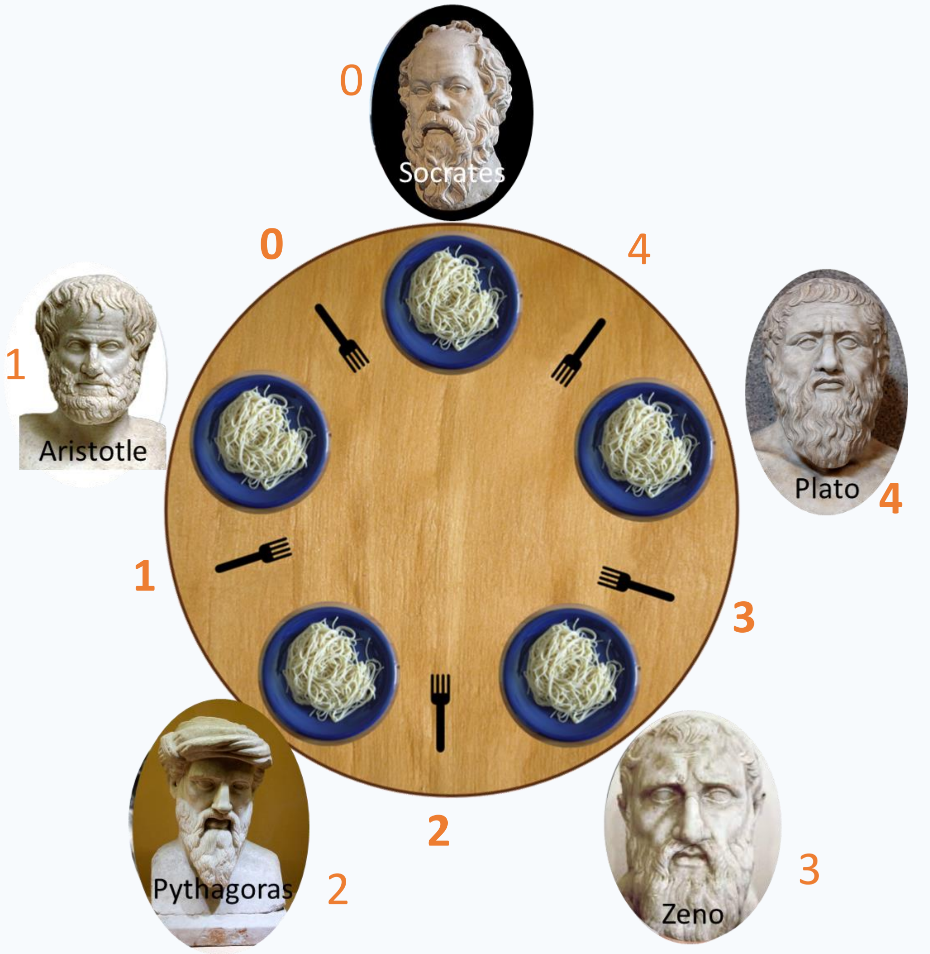
Having one philosopher pick up forks in a different order than the others is sufficient to break the symmetry, and thus to avoid deadlock.

```
public class AsymetricTable 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* from Lecture 2 …:

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* from Lecture 2 …:

1. mutual exclusion: exclusive access to the shared resources,
2. hold and wait: request one resource while holding another,
3. no preemption: resources cannot forcibly be released,
4. circular wait: threads form a circular chain, each waiting for a resource that the next is holding.

Circular wait is a necessary condition 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 removing them from 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 consumers 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	consumer_m
`while (true) {`	`while (true) {`
`  // create a new item`	`  Item item = buffer.get();`
`  Item item = produce();`	`  // do something with ‘item’`
`  buffer.put(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() + at(5,15-17); }
}
```

```
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
}
```

signal t
that the

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 again for `lock` just after it has been signaled to 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 benign because temporary).

Can we execute `up` after `unlock`?

Executing `up` after `unlock`

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

```
7  public T get() {
8   nItems.down();
9   lock.lock();
10  T item =storage.remove();
11  lock.unlock();
12  return item;
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: $\langle y \rangle$

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, ...)
                                                     ) + at(5,15-17); }
```

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

Can we execute `down` after `lock`?

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

Executing `down` after `lock`:

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

Bounded shared buffer

size of 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() +
            + at(6,13-15) == N - nFree.count() - at(4-6,16) ; }
```

```
1  public void put(T item) {
2    // wait until nFree > 0
3    nFree.down();
4    lock.lock(); // lock
5    // store item
6    storage.add(item);
7    nItems.up();   // update nItems
8    lock.unlock(); // release
9  }
```

may deadlock if swapped

OK to swap

```
10 public T get() {
11   // wait until nItems > 0
12   nItems.down();
13   lock.lock(); // lock
14   // retrieve item
15   T item = storage.remove();
16   nFree.up(); // update nFree
17   lock.unlock(); // release
18   return item;
19 }
```

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.

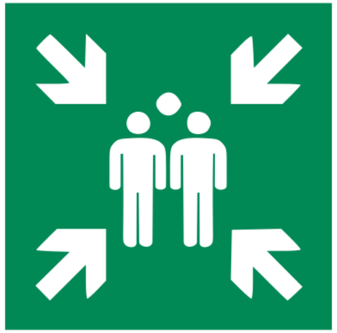
Busy-waiting on the semaphore will not work:

```
  // 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



Capacity 0 forces `up` before first `down`

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

```
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 tread has reaches the barrier

Barriers: variant 1

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

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

Barriers: variant 2

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 (single use)

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

```
// 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
```

total number of expected threads (annotation pointing to `n`)

Barriers with n threads (single use): 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`?

such pairs of wait/signal are called turnstiles

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

Reusable barriers

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

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

returned from

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


```
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;
    }

}
```

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

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!

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

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

Reusable barriers: second attempt


```
public class NonBarrier2 implements Barrier {
   int nDone = 0; // number of done threads
   Semaphore open = new Semaphore(0);
   final int n;

   // initialize barrier for `n' threads
   NonBarrier2(int n) {
     this.n = n;
   }

   // number of threads expected at the barrier
   int expect() {
     return n;
   }

   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.

This is not prevented by strong semaphores: it occurs because the last thread through leaves the gate open (calls `open.up()`)

Reusable barriers: correct solution

Photo by Photnart: Heidelberg Lock, Germany.

Reusable barriers: correct solution

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

Reusable barriers: improved solution

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

```
public class NSemaphoreBarrier extends SemaphoreBarrier {
    Semaphore gate1 = new Semaphore(0);   // first gate
    Semaphore gate2 = new Semaphore(0);   // second gate

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

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

both gates initially closed

open gate1 for n threads

open gate2 for n threads

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

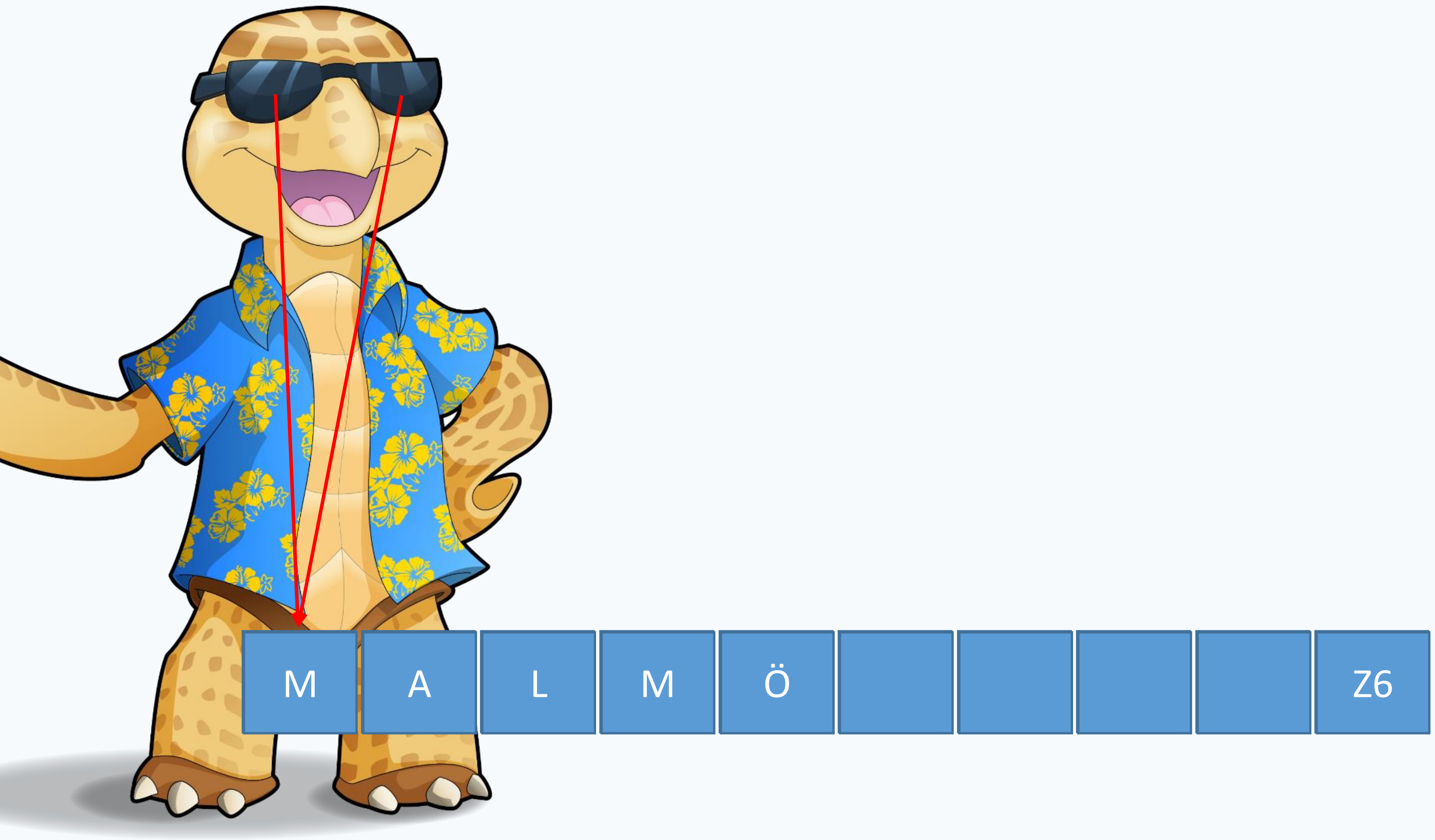
Readers and writers concurrently access shared data:

- 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.

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

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: $\#WRITERS = 0 \lor (\#WRITERS = 1 \land \#READERS = 0)$

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
```

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 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

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

  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