



CHALMERS

Synchronization problems with monitors

Lecture 6 of TDA383/DIT390 (Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg

SP3 2016/2017

Today's menu

Resource allocator

Barriers

Readers-writers

Dining philosophers

Sleeping barber

A gallery of synchronization problems

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

We will mostly use **pseudo-code** representing monitor classes with **signal and continue** discipline, 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.

Monitors in pseudo-code

We declare monitor classes by adding the pseudo-code keyword `monitor` to regular Java classes. Note that `monitor` is **not** a valid Java keyword—that is why we highlight it in a different color—but we will use it to simplify the presentation of monitors.

Turning a pseudo-code monitor class into a proper Java class is straightforward:

- mark all attributes as **private**
- add **locking** to all public methods

Details on how to implement monitors in Java are presented in the class on monitors.

We also occasionally annotate monitor 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.

Resource allocator

Resource allocator: the problem

An **allocator** grants **users** exclusive access to a number of resources:

- **users** asynchronously request resources and release them back
- the **allocator** ensures resources are given exclusively to one user at a time, and keeps tracks of how many resources are available

```
interface Allocator<T> {  
    // get 'n' resources; block if not available  
    Set<T> request(int n);  
  
    // release 'resources'  
    void release(Set<T> resources);  
}
```

Resource allocator problem: implement Allocator such that:

- an arbitrary number of users can access the allocator
- users are granted exclusive access to resources

Users

Users continuously and asynchronously access the allocator, which must guarantee proper synchronization.

```
Allocator<Resource> allocator;
```

user_k

```
while (true) {  
    // how many resources are needed?  
    int n = howMany();  
    // get resources from allocator  
    Set<Resource> res = allocator.request(n);  
    // do something with resources  
    for (Resource r: res) use(r);  
    // release resources  
    allocator.release(res); // we assume |res| does not change  
}
```

Allocator: request resources

The allocator problem is similar to the producer-consumer: users waiting for resources block, and get unblocked when some other users releases them. The important thing is to only **acquire all resources at once** when they are all available.

```
monitor class MonitorAllocator<T> implements Allocator<T> {
    int available; // number of available resources
    // new resources have become available
    Condition moreAvailable = new Condition();
    Set<T> storage = new Set<>(); // actual resources

    public Set<T> request(int n) {
        while (available < n)
            moreAvailable.wait();
        // now n resources are available
        available -= n; // update count
        return storage.removeAll(n); // remove and return n resources
    }
}
```

do not update available until done waiting!

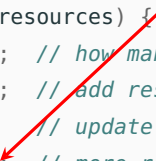
Allocator: release resources

When releasing resources, we do not know how many resources each waiting thread is waiting for. Therefore, we have to signal **all waiting** users; each will have a chance to check whether enough resources are available.

```
monitor class MonitorAllocator<T> implements Allocator<T> {
    int available; // number of available resources
    // new resources have become available
    Condition moreAvailable = new Condition();
    Set<T> storage = new Set<>(); // actual resources

    public void release(Set<T> resources) {
        int n = resources.count(); // how many resources?
        storage.addAll(resources); // add resources to storage
        available += n; // update count
        moreAvailable.signalAll(); // more resources are available
    }
}
```

must signal all waiting threads



Allocator: targeted signaling

MonitorAllocator may be somewhat inefficient: if there are many users and few resources in the system, every time there is a release all threads are unblocked and resumed, but most of them will have to go back waiting. In these conditions, it may be better to use multiple condition variables to **signal precisely** how many resources have become available.

```
monitor class TargetedAllocator<T> extends MonitorAllocator<T> {
    Condition[] have = new Condition[N]; // N = total # of resources

public Set<T> request(int n) { public void release(Set<T> resources) {
    // wait for n resources      int n = resources.count();
while (available < n)          storage.addAll(resources);
    have[n - 1].wait();        available += n;
available -= n;               for (int k = available; 0 < k; k--)
return storage.removeAll(n);   have[k - 1].signal();
}                               }
```

Allocator: targeted signaling

MonitorAllocator may be somewhat inefficient: if there are many users and few resources in the system, every time there is a release all threads are unblocked and resumed, but most of them will have to go back waiting. In these conditions, it may be better to use multiple condition variables to **signal precisely** how many resources have become available.

```
monitor class TargetedAllocator<T> extends MonitorAllocator<T> {
    Condition[] have = new Condition[N]; // N = total # of resources

    public Set<T> request(int n) {
        // wait for n resources
        while (available < n)
            have[n - 1].wait();
        available -= n;
        return storage.removeAll(n);
    }

    public void release(Set<T> resources) {
        int n = resources.count();
        storage.addAll(resources);
        available += n;
        for (int k = available; 0 < k; k--)
            have[k - 1].signal();
    }
}
```

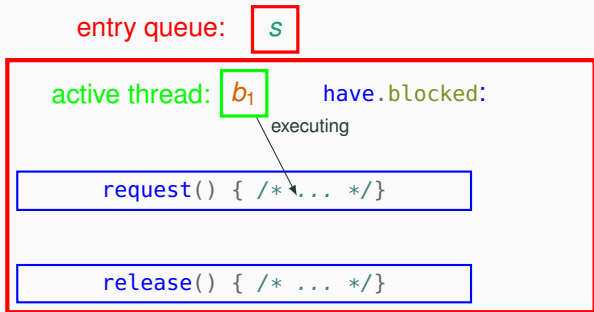
arrays indexed from 0

signal threads waiting for $1 \leq k \leq \text{available resources}$

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

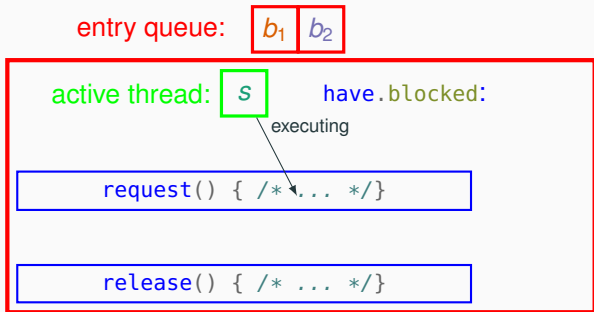


who has the resource: allocator

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

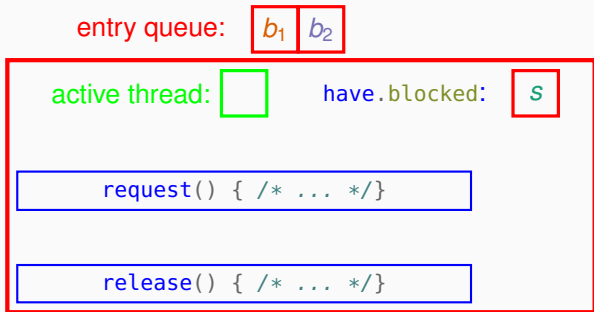


who has the resource: b_1

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

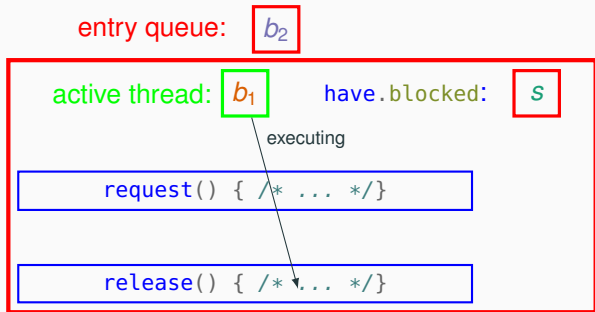


who has the resource: b_1

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

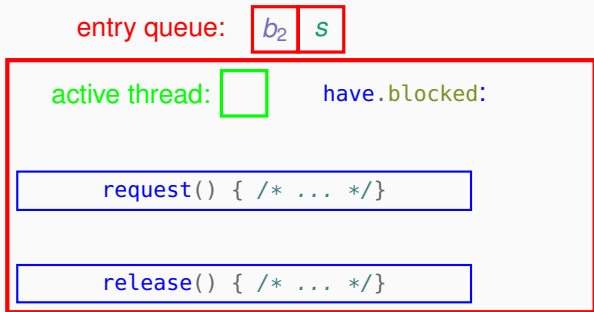


who has the resource: b_1

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

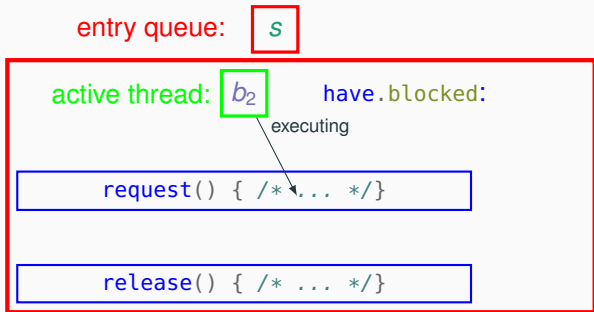


who has the resource: allocator

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

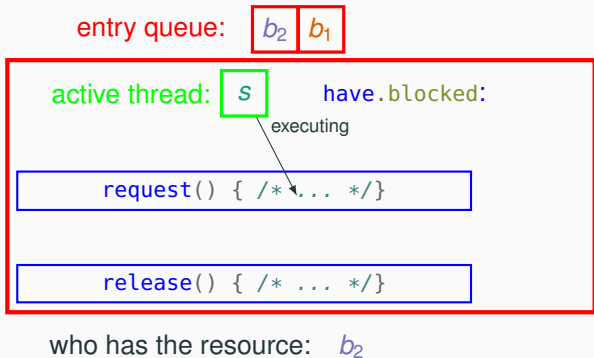


who has the resource: allocator

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

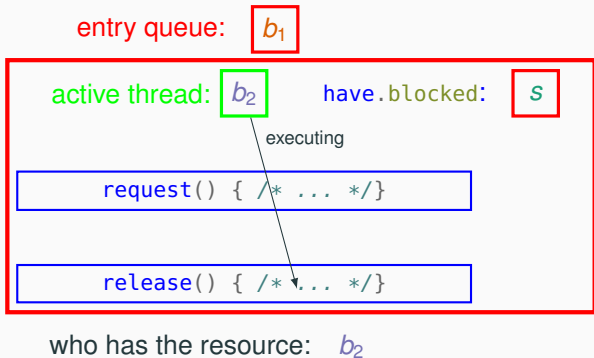
For simplicity, suppose there is a single resource in the system.



Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

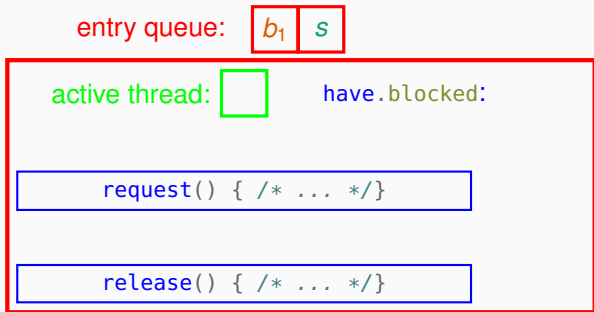
For simplicity, suppose there is a single resource in the system.



Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.

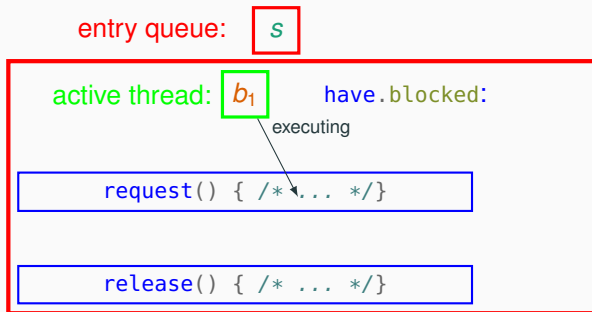


who has the resource: allocator

Allocator: targeted signaling — fairness analysis

Neither TargetedAllocator nor MonitorAllocator satisfy starvation freedom: two users b_1 and b_2 can alternate getting the resources and starve another user s .

For simplicity, suppose there is a single resource in the system.



who has the resource: allocator

This is the same as the initial state, so this scenario can **repeat indefinitely**.

Fair allocator

Starvation occurs because new calls to request take over waiting calls. To avoid starvation, we let **new requests yield to waiting ones**:

```
public Set<T> request(int n) {  
    // wait until n resources are available  
    // and there are no waiting users  
    while (available < n || ∃ k: !have[k].isEmpty())  
        have[n - 1].wait();  
    available -= n;    0 ≤ k < N  
    return storage.removeAll(n);  
}
```

By waiting until **all previous waiting** requests have been served, we have a first-come-first-served policy for assigning resources.

Fair allocator

Starvation occurs because new calls to request take over waiting calls. To avoid starvation, we let **new requests yield to waiting ones**:

```
public Set<T> request(int n) {  
    // wait until n resources are available  
    // and there are no waiting users  
    while (available < n || ∃ k: !have[k].isEmpty())  
        have[n - 1].wait();  
    available -= n;    0 ≤ k < N  
    return storage.removeAll(n);  
}
```

By waiting until **all previous waiting** requests have been served, we have a first-come-first-served policy for assigning resources.

This is not directly implementable in Java because `Condition` does not offer `isEmpty`, but one can emulate it by explicitly keeping track of waiting processes. We will follow a similar approach for fairness in the readers-writers problem with monitors.

Barriers

Reusable barriers — recap

```
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  
}
```

Barriers: first attempt

As a first attempt, we use a condition variable `allDone` to signal when the last thread has arrived at the barrier.

```
monitor class NonBarrier1 implements Barrier {
    final int n;                // number of expected threads
    int nDone = 0;             // number of arrived threads
    Condition allDone = new Condition(); // all threads arrived

    public void wait()
    { nDone += 1;
      if (nDone == n) allDone.signalAll(); // if last, signal all
      else while (nDone < n) allDone.wait(); // else wait for allDone
      nDone -= 1; }
}
```

Barriers: first attempt

As a first attempt, we use a condition variable `allDone` to signal when the last thread has arrived at the barrier.

```
monitor class NonBarrier1 implements Barrier {
    final int n; // number of expected threads
    int nDone = 0; // number of arrived threads
    Condition allDone = new Condition(); // all threads arrived

    public void wait()
    { nDone += 1;
      if (nDone == n) allDone.signalAll(); // if last, signal all
      else while (nDone < n) allDone.wait(); // else wait for allDone
      nDone -= 1; }
}
```

- The **while** loop around `allDone.wait()` is needed because of the **signal and continue** discipline (and possible spurious wakeups)
- Unfortunately, it also keeps on blocking all threads but the last one, since they all read `nDone == n - 1 < n` when unblocked

Barriers: second attempt

As a second attempt, we remove the decrement to `nDone`, so that all waiting threads can continue.

```
monitor class DisposableBarrier implements Barrier {
    final int n;                // number of expected threads
    int nDone = 0;              // number of arrived threads
    Condition allDone = new Condition(); // all threads arrived

    public void wait()
    { nDone += 1;
      if (nDone == n) allDone.signalAll(); // if last, signal all
      else while (nDone < n) allDone.wait(); } // else wait for allDone
}
```

This works in the first iteration, but it is **not reusable**: after the last thread signals `allDone`, the barrier remains open indefinitely.

Barriers: correct solution

A correct solution uses another variable `round` to keep track of how many times the barrier has been used. The last thread in each round resets `nDone` and increments `round` for the next iteration; other threads only block as long as the current turn is not complete.

```
monitor class TurnBarrier implements Barrier {
    int round = 0; /* other variables as in previous attempts */
    public void wait()
    { nDone += 1;
      int myRound = round; // what round am I in?
      if (nDone == n) { // if last at barrier:
        allDone.signalAll(); // signal all
        nDone = 0; // reset counter
        round += 1; // increase round
      } else while (nDone < n // else wait for allDone
                   && myRound == round) // while the turn
        allDone.wait(); // has not changed
    }
}
```

Barriers: correct solution

A correct solution uses another variable `round` to keep track of how many times the barrier has been used. The last thread in each round resets `nDone` and increments `round` for the next iteration; other threads only block as long as the current turn is not complete.

```
monitor class TurnBarrier implements Barrier {
    int round = 0; /* other variables as in previous attempts */
    public void wait()
    { nDone += 1; to avoid overflows, count modulo 2: round = (round+1)%2
      int myRound = round; // what round am I in?
      if (nDone == n) { // if last at barrier:
        allDone.signalAll(), // signal all
        nDone = 0; // reset counter
        round += 1; // increase round
      } else while (nDone < n // else wait for allDone
        && myRound == round) // while the turn
        allDone.wait(); } // has not changed
}
```

Readers-writers

Readers-writers: the problem — a new variant

```
interface AccessBoard {  
    void beginRead(); // get read access to board  
    void endRead();   // release read access  
    void beginWrite(); // get write access to board  
    void endWrite();  // release write access  
}
```

Readers-writers problem: implement AccessBoard such that:

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

Invariant: $\#WRITERS = 0 \vee (\#WRITERS = 1 \wedge \#READERS = 0)$

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.

```
AccessBoard board;
```

reader_{*n*}

```
while (true) {  
    board.beginRead();  
    // read messages  
    board.endRead();  
}
```

writer_{*m*}

```
while (true) {  
    board.beginWrite();  
    // write messages  
    board.endWrite();  
}
```

Where is the reading and writing?

In the semaphore version of the readers-writers problem, we had single methods `read` and `write` that guaranteed synchronization and performed the actual reading and writing.

In contrast, the monitor version that we are describing now has methods to synchronize the `begin` and `end` of reading and writing, but the `actual reading and writing` is left implicit:

- the actual writing could be performed in a monitor method, since it operates in mutual exclusion w.r.t. both readers and writers
- however, the actual reading should not be performed in a monitor method, since it would prevent multiple readers from actually operating concurrently

Where is the reading and writing?

In the semaphore version of the readers-writers problem, we had single methods `read` and `write` that guaranteed synchronization and performed the actual reading and writing.

In a concrete implementation, the **actual reading and writing** could operate on a different (non-synchronized) shared object:

- readers and writers **must** follow the protocol of calling `beginRead`/`beginWrite` before reading/writing the other shared object, and calling `endRead`/`endWrite` when they are done reading/writing it
- this way, synchronization is guaranteed for **any shared object** that can be read and written, and the only actual concurrency is the one permitted by the problem (multiple readers, and no writers)

Readers-writers: condition variables

Since we want to allow multiple readers on the board at the same time, we cannot simply give the lock on the whole monitor to a single thread. Instead, we use **condition variables** to notify threads when it is OK to read or to write.

```
monitor class MonitorBoard implements ActiveBoard {
    int nReaders = 0; // # readers active on the board
    int nWriters = 0; // # writers active on the board

    Condition readOK = new Condition(); // readers can access board
    Condition writeOK = new Condition(); // a writer can access board

    invariant { nWriters == 0 || (nWriters == 1 && nReaders == 0) }
```

Readers-writers: entry/exit methods

Entry methods `beginRead` and `beginWrite`:

- **wait** until it is OK to read/write
- **increment** the number of readers/writers

Exit methods `endRead` and `endWrite`:

- **decrement** the number of readers/writers
- **signal** waiting readers/writers

The tricky part is **when** to **signal** and when to **wait**: different choices lead to different priorities of readers vs. writers.

Readers-writers: entry methods — first version

In the first version, the waiting conditions follow directly from the class invariant: readers wait until there are no writers, and writers wait until there are neither readers nor writers.

```
public void beginRead() {  
    // wait until:  
    // no active writers  
    while (nWriters > 0)  
        readOK.wait();  
    nReaders += 1;  
    // more readers welcome  
    readOK.signalAll();  
}
```

```
public void beginWrite() {  
    // wait until:  
    // no active writers and  
    // no active readers  
    while (nWriters > 0  
           || nReaders > 0)  
        writeOK.wait();  
    nWriters += 1; // nWriters == 1  
}
```

Readers-writers: entry methods — first version

In the first version, the waiting conditions follow directly from the class invariant: readers wait until there are no writers, and writers wait until there are neither readers nor writers.

```
public void beginRead() {
    // wait until:
    // no active writers
    while (nWriters > 0)
        readOK.wait();
    nReaders += 1;
    // more readers welcome
    readOK.signalAll();
}

public void beginWrite() {
    // wait until:
    // no active writers and
    // no active readers
    while (nWriters > 0
           || nReaders > 0)
        writeOK.wait();
    nWriters += 1; // nWriters == 1
}
```

The `readOK.signalAll()` at the end of `beginRead` is not needed: a reader only waits when `nWriters > 0`, so as long as `endWrite` unblocks all readers there will not be waiting readers there.

Readers-writers: exit methods — first version

```
public void endRead() {  
    nReaders -= 1;  
    // if last reader: resume  
    // one waiting writer  
    if (nReaders == 0)  
        writeOK.signal();  
    // more readers welcome  
    readOK.signalAll();  
}
```

```
public void endWrite() {  
    nWriters -= 1; // nWriters == 0  
    // resume one waiting writer  
    // and all waiting readers  
    readOK.signalAll();  
    writeOK.signal();  
}
```

Readers-writers: exit methods — first version

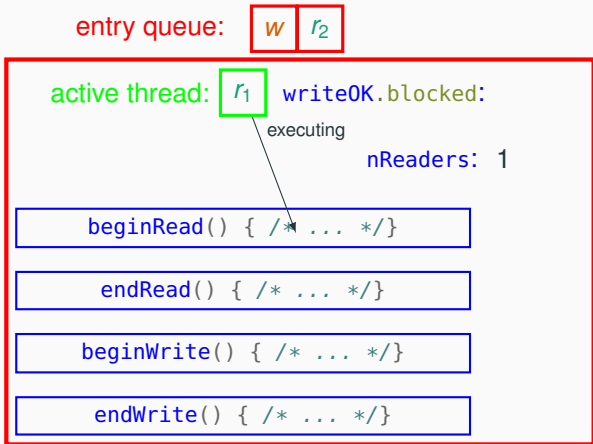
```
public void endRead() {
    nReaders -= 1;
    // if last reader: resume
    // one waiting writer
    if (nReaders == 0)
        writeOK.signal();
    // more readers welcome
    readOK.signalAll();
}
```

```
public void endWrite() {
    nWriters -= 1; // nWriters == 0
    // resume one waiting writer
    // and all waiting readers
    readOK.signalAll();
    writeOK.signal();
}
```

Similarly as in `beginRead`, the `readOK.signalAll()` at the end of `endRead` is not needed: a reader only waits when `nWriters > 0`, so as long as `endWrite` unblocks all readers there will not be waiting readers there.

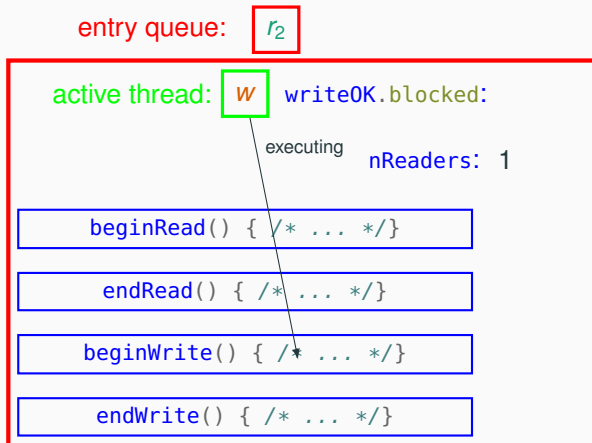
Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.



Readers-writers: the first version prioritizes readers

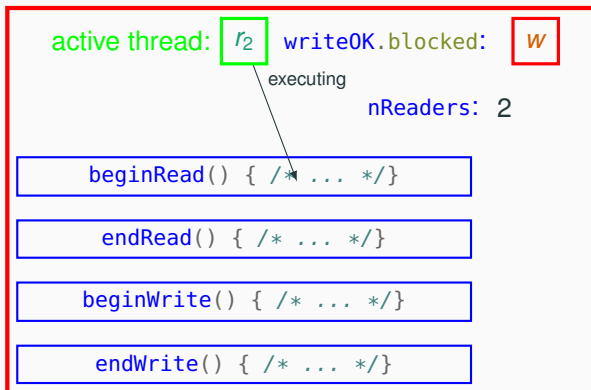
The first version ensures **mutual exclusion** but gives **priority** to readers over writers.



Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.

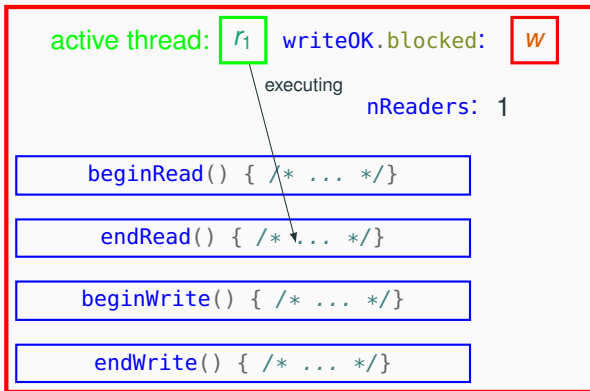
entry queue:



Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.

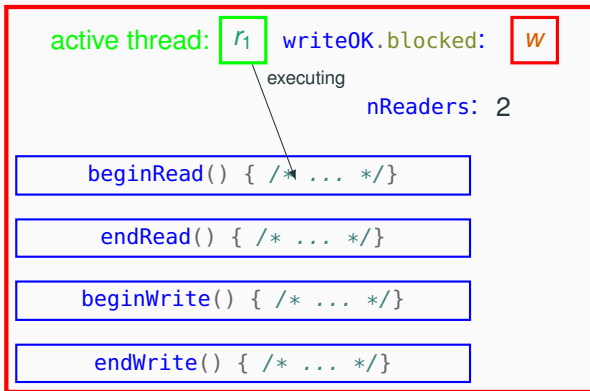
entry queue:



Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.

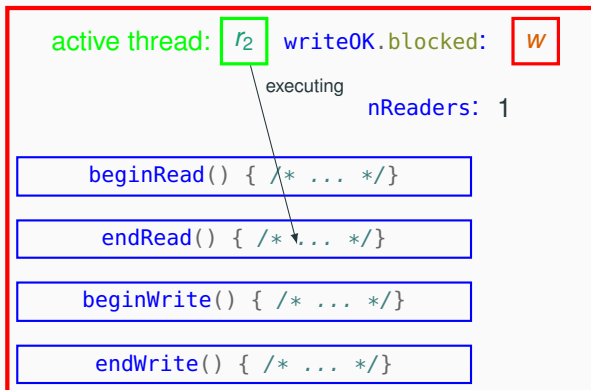
entry queue:



Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.

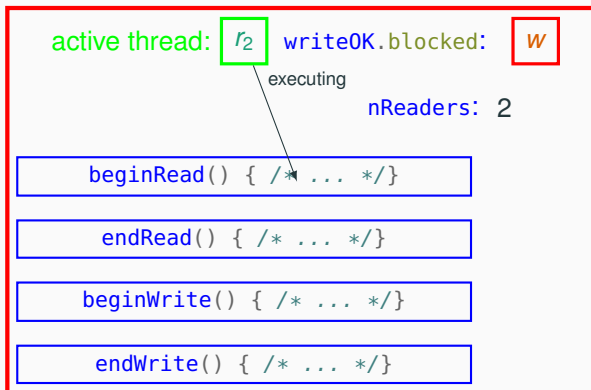
entry queue:



Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.

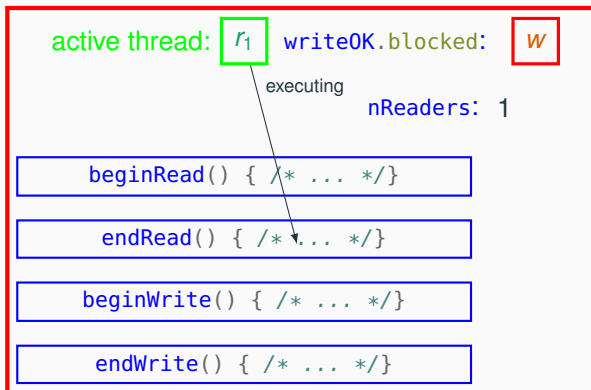
entry queue:



Readers-writers: the first version prioritizes readers

The first version ensures **mutual exclusion** but gives **priority** to readers over writers.

entry queue:



Déjà vu!

Readers-writers: the first version prioritizes readers

The first version is **correct** in that it ensures **mutual exclusion** according to the readers-writers invariant.

However, it gives **priority** to readers over writers:

- new readers can enter the monitor without waiting as long as a reader is active
- waiting writers have to wait until the last reader calls `endRead` and signals `writeOK`
- as long as readers keep arriving and queuing for entering the monitor, the waiting writers will never execute

Conversely, waiting readers are unblocked as soon as the current writer finishes — independent of whether there are other writers waiting.

Readers-writers: towards a fair solution

We show a **fair solution**, which gives equal priority to readers and writers, **using the signal and wait** signaling discipline:

- **readers** give **priority to waiting writers**, so that a reader will be able to begin reading as soon as the writers that have been waiting longer finish
- **writers** give **priority to waiting readers**, so that a writer will be able to begin writing as soon as the readers that have been waiting longer finish

Implementing the same behavior under the signal and continue discipline would require more explicit bookkeeping—a less elegant solution, which we do not discuss explicitly.

This example shows how the semantics of signal and wait is easier to understand and use in programs; unfortunately, it is not the semantics available in most programming languages.

Readers-writers: entry methods — fair solution

Entry methods `beginRead` and `beginWrite`:

- **wait** until it is OK to read/write
- **increment** the number of readers/writers

```
public void beginRead() {  
    // wait until:  
    // no active writers and  
    // no blocked writers  
    if (nWriters > 0  
        || !writeOK.isEmpty())  
        readOK.wait();  
    nReaders += 1;  
    // more readers welcome  
    readOK.signal();  
}
```

```
public void beginWrite() {  
    // wait until:  
    // no active writers and  
    // no active readers  
    if (nWriters > 0  
        || nReaders > 0)  
        writeOK.wait();  
    nWriters += 1;  
}
```

give priority to waiting writers over readers

Readers-writers: entry methods — fair solution

Entry methods `beginRead` and `beginWrite`:

- **wait** until it is OK to read/write
- **increment** the number of readers/writers

```
public void beginRead() {  
    // wait until:  
    // no active writers and  
    // no blocked writers  
    if (nWriters > 0  
        || !writeOK.isEmpty())  
        readOK.wait();  
    nReaders += 1;  
    // more readers welcome  
    readOK.signal();  
}
```

```
public void beginWrite() {  
    // wait until:  
    // no active writers and  
    // no active readers  
    if (nWriters > 0  
        || nReaders > 0)  
        writeOK.wait();  
    nWriters += 1;  
}
```

give priority to waiting writers over readers

no waiting loop under `signal` and `wait` (a loop would lose signals)

Readers-writers: exit methods — fair solution

Exit methods `endRead` and `endWrite`:

- **decrement** the number of readers/writers
- **signal** waiting readers/writers

```
public void endRead() {  
    nReaders -= 1;  
    // if last reader: one  
    // waiting writer can resume  
    if (nReaders == 0)  
        writeOK.signal();  
}
```

```
public void endWrite() {  
    nWriters -= 1;  
    // if no waiting readers: one  
    // waiting writer can resume  
    if (readOK.isEmpty())  
        writeOK.signal();  
    else  
        // otherwise: one waiting  
        // reader can resume  
        readOK.signal();  
}
```

give priority to waiting readers over writers

`beginRead` signals `readOK.signal()` to other waiting readers

Readers-writers with priorities

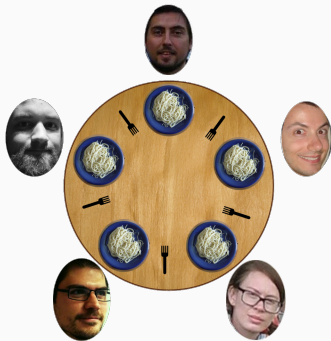
Signaling readers prioritize writers, and signaling writers prioritize readers. This provides a **starvation free solution**: writers and readers take turn as they try to access the board.

It is not difficult to adapt `MonitorBoard`'s implementation to implement **different priorities** (readers over writers, or writers over readers).

Dining philosophers

Dining philosophers: the problem — recap

```
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

Our solution now uses again the **signal and continue** discipline.

The state of the forks

We model the **forks' state** with an array `forks`: `fork[k] == -1` if fork `k` is on the table; otherwise `fork[k]` is the number of the philosopher holding fork `k`. Similarly, condition variable `available[k]` is used to signal that fork `k` has **become available**.

```
monitor class MonitorTable implements Table {
    int[] forks = new int[N] { -1 }; // initialized to all -1
    Condition[] available = new Condition[N];

    invariant {  $\forall k: -1 \leq \text{fork}[k] < N$  }

    void putForks(int k) {
        forks[left(k)] = -1;           // put down left fork
        available[left(k)].signal();  // notify waiting thread
        forks[right(k)] = -1;        // put down right fork
        available[right(k)].signal(); // notify waiting thread
    }
}
```

Picking up forks

We rely on the monitor's **implicit locks** to acquire both forks **atomically**.

```
void getForks(int k) {  
    while (forks[left(k)] >= 0)  
        available[left(k)].wait(); // wait for left fork  
    forks[left(k)] = k;           // pick up left fork  
    while (forks[right(k)] >= 0)  
        available[right(k)].wait(); // wait for right fork  
    forks[right(k)] = k;         // pick up right fork  
}
```

If philosopher k blocks waiting for the right fork, it will release the lock on the monitor, but k 's left neighbor will still find the philosopher k 's left fork **unavailable**. Can this determine a deadlock?

Deadlock analysis

If P_k blocks waiting for the right fork, it will release the lock on the monitor, but P_k 's left neighbor P_{k-1} will still find the philosopher P_k 's left fork **unavailable**. Can this determine a deadlock?

Suppose P_k is waiting for the right fork while holding the left fork.

- Since monitor methods are executed atomically, when P_k started executing `getForks`, P_{k+1} was already holding P_k 's right fork, and thus P_{k+1} started executing `getForks` before P_k did.
- Let e_k be the time when P_k starts executing `getForks`.
- A deadlock occurs only if there is circular waiting, which is possible only if **every** P_j is waiting for the right fork from P_j 's neighbor P_{j+1} .
- The condition for deadlock is thus $e_k > e_{k+1} > \dots > e_{k-1} > e_k$, which is contradictory: a deadlock is impossible.

Fairness analysis

Can a philosopher **starve** by never getting access to both forks?

Suppose P_k is waiting for the **right** fork while holding the left fork:

- As soon as P_k 's right neighbor P_{k+1} is done eating, P_k will be moved to the entry queue of the monitor.
- P_{k+1} is the only philosopher who can take the fork again, but in order to do so P_{k+1} has to re-enter the monitor by queuing at the entry after P_k .
- Therefore P_k will get to have both forks, and will be able to eat.

Suppose P_k is waiting for the **left** fork:

- As soon as P_k 's left neighbor P_{k-1} is done eating, P_k will be moved to the entry queue of the monitor.
- P_{k-1} is the only philosopher who can take the fork again, but in order to do so P_{k-1} has to re-enter the monitor by queuing at the entry after P_k .
- Therefore P_k will get the left fork at the next attempt.
- As explained above, P_k will be able to eat from this configuration.

Sleeping barber

The sleeping barber

A **barbershop** has a barber chair and some chairs for waiting customers.

- If there are no waiting customer, the barber takes a nap
- If there are customers waiting in the chairs, the barber calls one up for a haircut
- If a customer enters the barbershop and finds the barber sleeping, the customer wakes up the barber and gets a haircut immediately
- If a customer arrives when the barber is working, the customer sits in one of the available chairs
- If a customer arrives and finds no empty chairs, the customer leaves right away



Our solution now uses again the **signal and continue** discipline.

The sleeping barber: motivation

The sleeping barber is a **synchronization** problem between two **asymmetric** parties — a customer and the barber.

- They are **asymmetric** because barber and customers have different obligations and behavior
- They abstract a **client/server** relationship between threads
- In a way, the problem is an asymmetric **variant** of barrier synchronization

Sleeping barber: the problem

```
interface Barbershop {  
    // try to get a haircut  
    // block if waiting slots are available  
    void enter();  
  
    // serve next customer  
    // block if no customers  
    void serve();  
}
```

Sleeping barber problem: implement Barbershop such that:

- barber and customers behave as explained above
- it supports an arbitrary number of customers (possibly changing over time)
- it ensures starvation freedom

Barber and customers

The customers continuously and asynchronously arrive at the barbershop; the barber continuously try to serve customers.

Barbershop shop;

customer_k

```
while (true) {  
  shop.enter();  
  // go on with your life  
}
```

barber

```
while (true) {  
  shop.serve();  
}
```

Barbershop with monitors

```
monitor class MonitorBarbershop implements Barbershop {
    int freeChairs = n;           // n waiting chairs
    Condition barber = new Condition(); // signals barber available
    Condition customer = new Condition(); // signals customer available

    public void enter() {
        // if waiting chairs available
        if (freeChairs > 0) {
            // sit down
            freeChairs -= 1;
            // wake up barber
            customer.signal();
            // wait turn
            barber.wait();
        }
    }

    public void serve() {
        // wait for customers
        while (freeChairs == 0)
            customer.wait();
        // call waiting customer
        barber.signal();
        // customer moves to
        // barber chair
        freeChairs += 1;
        // do haircut
    }
}
```

Barbershop with monitors

```
monitor class MonitorBarbershop implements Barbershop {  
    int freeChairs = n;           // n waiting chairs  
    Condition barber = new Condition(); // signals barber available  
    Condition customer = new Condition(); // signals customer available
```

```
    public void enter() {  
        // if waiting chairs available  
        if (freeChairs > 0) {  
            // sit down  
            freeChairs -= 1;  
            // wake up barber  
            customer.signal();  
            // wait turn  
            barber.wait();  
        }  
    }  
}
```

no effect
if barber
already
is working



↑
should this be in a loop?

```
    public void serve() {  
        // wait for customers  
        while (freeChairs == n)  
            customer.wait();  
        // call waiting customer  
        barber.signal();  
        // customer moves to  
        // barber chair  
        freeChairs += 1;  
        // do haircut  
    }  
}
```

↖
at least one
customer waiting

Barbershop with monitors

```
monitor class MonitorBarbershop implements Barbershop {  
    int freeChairs = n;           // n waiting chairs  
    Condition barber = new Condition(); // signals barber available  
    Condition customer = new Condition(); // signals customer available
```

```
    public void enter() {  
        // if waiting chairs available  
        if (freeChairs > 0) {  
            // sit down  
            freeChairs -= 1;  
            // wake up barber  
            customer.signal();  
            // wait turn  
            barber.wait();  
        }  
    }
```

no effect
if barber
already
is working



should this be in a loop?
only for spurious wakeups

```
    public void serve() {  
        // wait for customers  
        while (freeChairs == 0) {  
            customer.wait();  
            // call waiting customer  
            barber.signal();  
            // customer moves to  
            // barber chair  
            freeChairs += 1;  
            // do haircut
```

at least one
customer waiting

