Review

- Concurrent Programming is Hard!
  - Non-Determinism
  - Non-Atomicity
- *Critical Sections* simplify things
  - mutual exclusion
  - progress
  - *Need both mutual exclusion and progress!*
- Critical Sections use a *lock*
- But how to implement a lock?

Peterson's Algorithm: *flags & turn*

```
sequential flags, turn

flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 − self
        await (not flags[1 − self]) or (turn == self)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Correctness?

- Want to prove *mutual exclusion*:
  - $T0@cs \land T1@cs \implies \text{False}$
- Use *proof by induction*
- But mutual exclusion itself is not inductive
  - Easy to come up with a mutual exclusive state from which a bad state can be reached
- Need a *stronger property* that is invariant and implies mutual inclusion
  - Tried: $T0@cs \Longrightarrow \neg flags[1] \vee turn = 0$
  - Too strong: Harmony shows that it's not invariant
  - Using Harmony output, weaken property to:
    - $T0@cs \implies \neg flags[1] \vee turn = 0 \vee T1@gate$
- Harmony shows it is an invariant
- Induction steps work!

Does it work in practice?

- No!
- Too inefficient if it did
- Worse: does not work on modern hardware
  - *Data race*: when two cores read/write the same memory location, the outcome is undefined
  - Load/Store operations are not *atomic*!
- Enter: *interlock instructions*

Interlock instructions

- Multiple loads/stores atomically executed
  - Test-and-Set
  - Swap
  - Compare-and-Swap
  - Fetch-and-Add
  - …
  - TAS(x): // test-and-set
    - set x to TRUE
    - return *prior* value of x

Spinlock

- initial state:

  lock = **False**

- enter critical section (acquire lock):

  while tas(?lock): **pass**
  assert lock

- leave critical section (release lock)

  lock = **False**

- Invariants:

  1. at most one thread can have the lock
  2. when T is in the critical section, it has the lock
     - Need both for mutual exclusion!

Alternative lock implementation

- Spinlock bad for simulated parallelism
  - does not work if no pre-emption
  - inefficient when there is pre-emption
- Instead, implement with *context switch*
  - each lock maintains a queue
  - thread goes on lock's queue if lock taken
  - thread resumed if lock released
  - Invariants:
    - thread need lock to execute in critical section
    - at most one thread can have lock
    - if a thread does not execute, it is on exactly one queue
      - either the ready queue or a lock's queue
    - if a thread is not on a queue, then it is executing

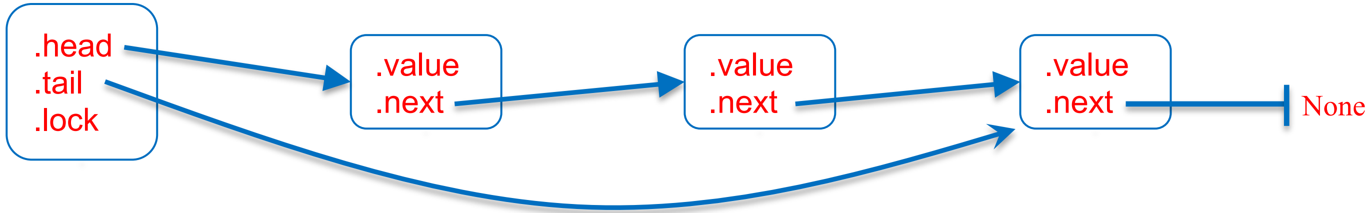
Using locks

- Data structure maintains some application-specific invariant (a type ***is*** an invariant)
    - e.g., in a linked list, there is a head, a tail, a list of nodes such that head points to first node, tail points to the last node, and each node points to the next one except the last, which points to **None**. However, if the list is empty, head and tail are both **None**.
- *You can assume the invariant holds right after obtaining the lock*
- *You must make sure the invariant holds again right before releasing the lock*

Building a Concurrent Queue

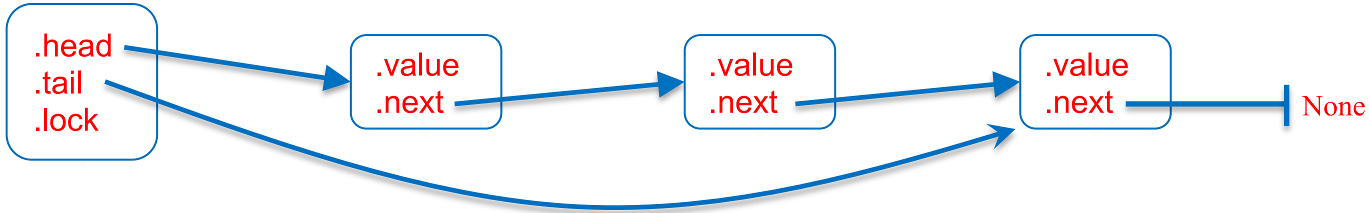
- q = queue.new(): allocate a new queue
- queue.put(q, v): add v to the tail of queue q
- v = queue.get(q): returns None if q is empty or v if v was at the head of the queue

Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

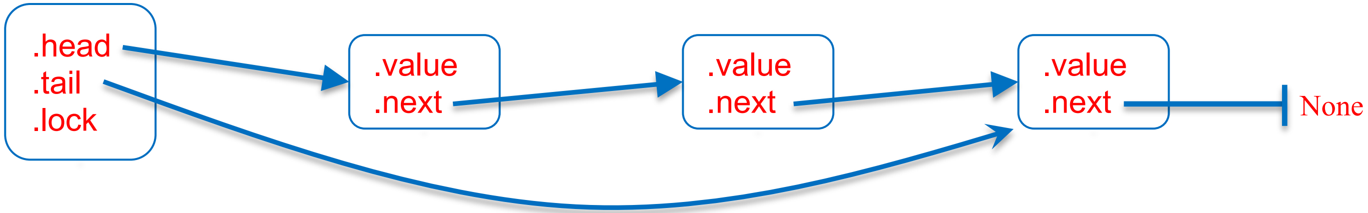
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

dynamic memory allocation

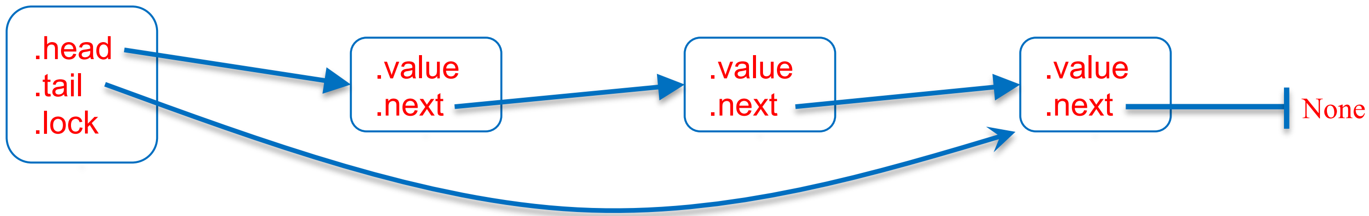
Queue implementation, v1



create empty queue

```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

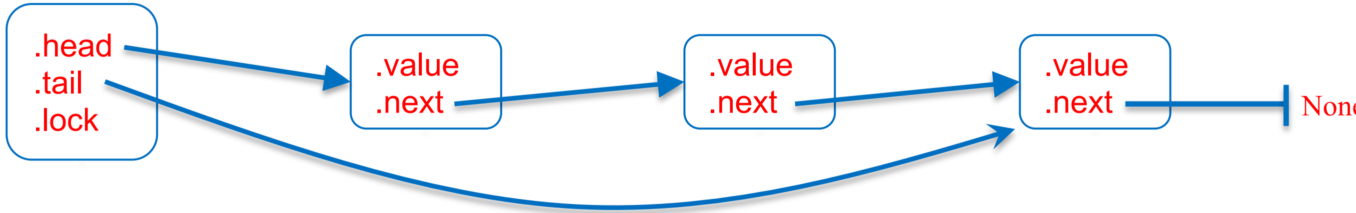
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

allocate node

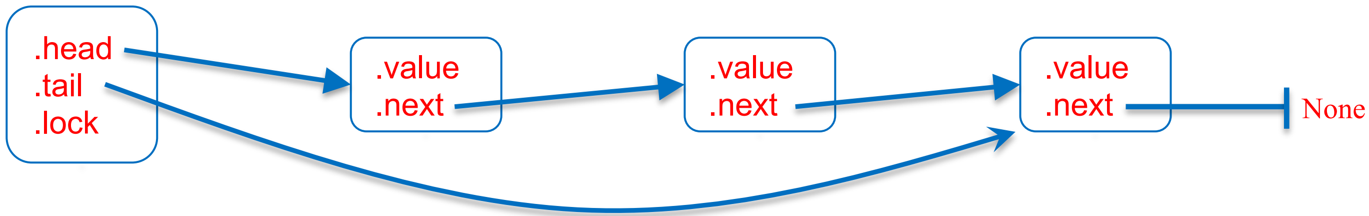
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

Queue implementation, v1

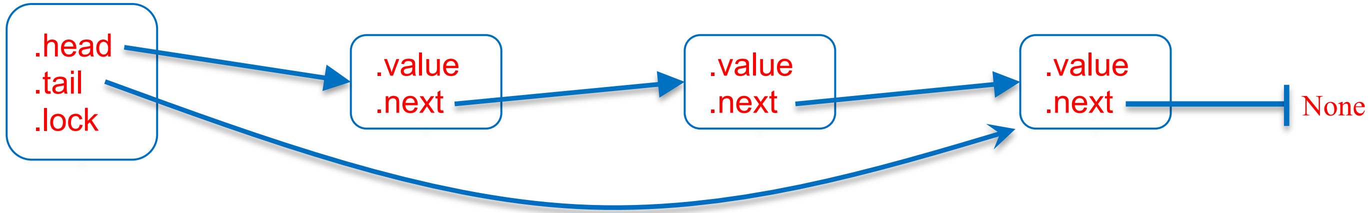



```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

the hard stuff

Queue implementation, v1



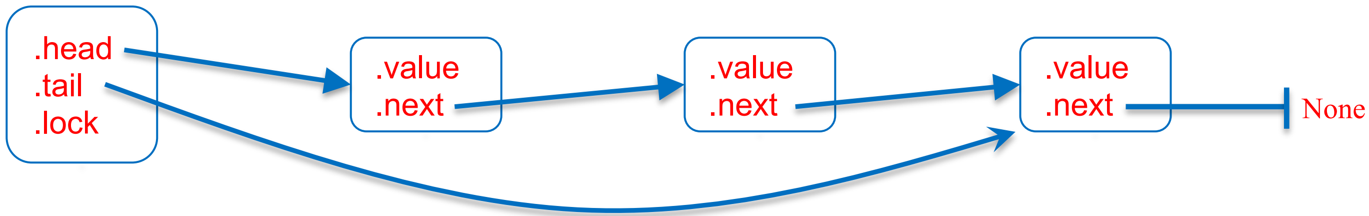

```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    result = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→head == None:
            q→head = q→tail = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

grab lock

the hard stuff

release lock

Queue implementation, v1




```
def get(q):
    acquire(?q→lock)
    let node = q→head:
        if node == None:
            result = None
        else:
            result = node→value
            q→head = node→next
            if q→head == None:
                q→tail = None
            free(node)
    release(?q→lock)
```

malloc'd memory must be explicitly released (cf. C)

Figure 10.2: [code/queue.hny]A basic concurrent queue data structure.

How important are concurrent queues?

- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

How important are concurrent queues?

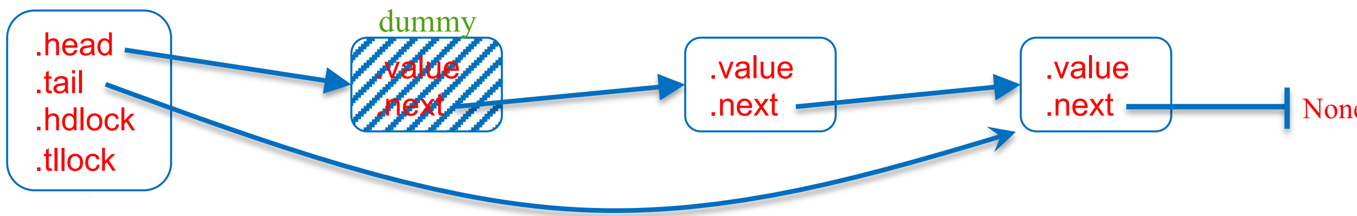
- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

Good performance is critical!

That’s how far we got!

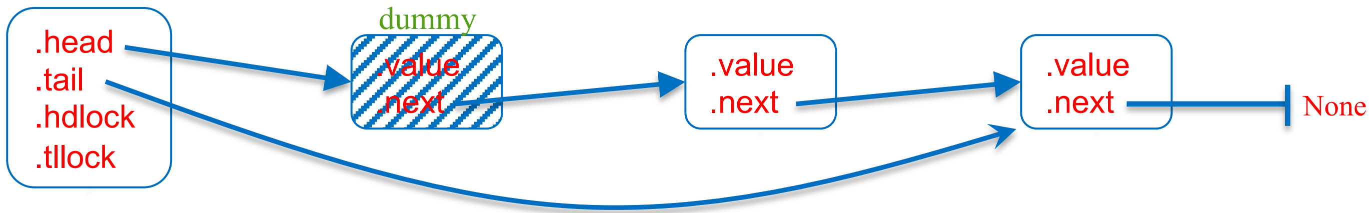
- *now let’s see how we can make concurrent data structures faster by allowing more concurrency*

Concurrent queue v2: 2 locks




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue():
    let dummy = malloc({ .value: (), .next: None }):
        result = { .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->tllock)
        q->tail->next = node
        q->tail = node
        release(?q->tllock)
```

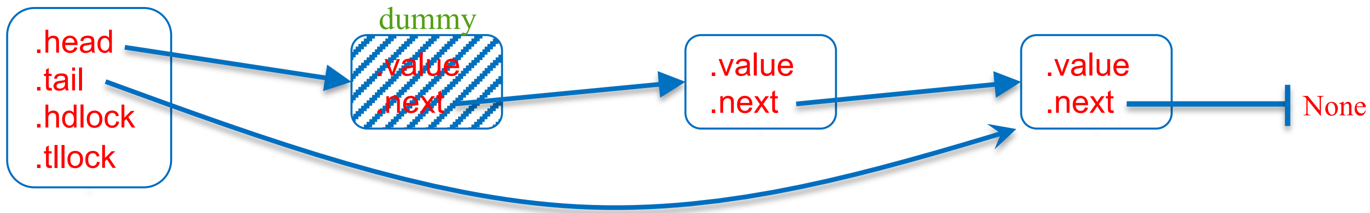
Concurrent queue v2: 2 locks




```
def get(q):
    acquire(?q→hdlock)
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
            free(dummy)
```

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueing and dequeuing items.

Concurrent queue v2: 2 locks

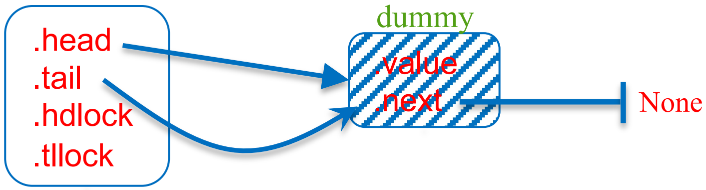



```
def get(q):
    acquire(?q→hdlock)
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
            free(dummy)
```

No contention for concurrent enqueue and dequeue operations!
➔ more concurrency ➔ faster

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.

Concurrent queue v2: 2 locks




```
def get(q):
    acquire(?q→hdlock)
    let dummy = q→head
    let node = dummy→next:
        if node == None:
            result = None
            release(?q→hdlock)
        else:
            result = node→value
            q→head = node
            release(?q→hdlock)
            free(dummy)
```

But also incorrect for today's hardware because of a data race…

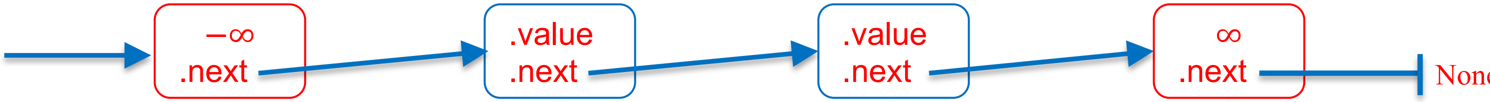
put and get concurrently access *dummy*→next when queue is empty

Figure 10.3: [code/queueMS.hny] A queue with separate locks for enqueuing and dequeuing items.

Global vs. Local Locks

- The two-lock queue is an example of a data structure with *finer-grained locking*
- A global lock is easy, but limits concurrency
- Fine-grained or local locking can improve concurrency, but tends to be trickier to get right

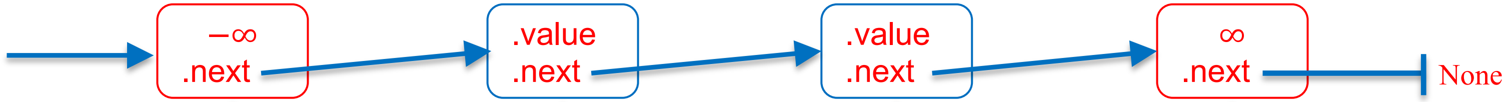
Sorted Integer Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):      # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    let before = lst:
        acquire(?before→lock)
        let after = before→next:
            acquire(?after→lock)
            while after→value < v:
                release(?before→lock)
                before = after
                after = before→next
                acquire(?after→lock)
            result = (before, after)

def LinkedList():
    result = _node(−inf, _node(inf, None))
```

create empty list

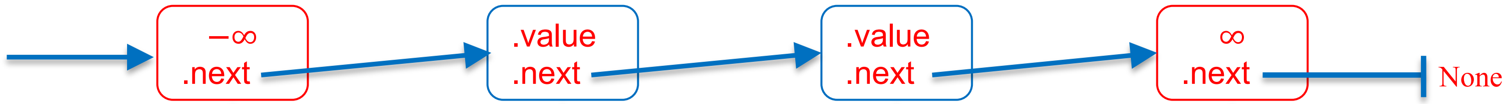
Sorted Integer Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):      # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    let before = lst:
        acquire(?before→lock)
        let after = before→next:
            acquire(?after→lock)
            while after→value < v:
                release(?before→lock)
                before = after
                after = before→next
                acquire(?after→lock)
            result = (before, after)

def LinkedList():
    result = _node(−inf, _node(inf, None))
```

Helper routine to find and lock two consecutive nodes *before* and *after* such that $before \rightarrow value < v \leq after \rightarrow value$

Sorted Integer Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n):      # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    let before = lst:
        acquire(?before→lock)
        let after = before→next:
            acquire(?after→lock)
            while after→value < v:
                release(?before→lock)
                before = after
                after = before→next
                acquire(?after→lock)
            result = (before, after)

def LinkedList():
    result = _node(−inf, _node(inf, None))
```

Helper routine to find and lock two consecutive nodes *before* and *after* such that $before \rightarrow value < v \leq after \rightarrow value$

Hand-over hand locking

(good for data structures without cycles)

Sorted Integer Linked List with Lock per Node

```
def insert(lst, v):
    let before, after = _find(lst, v):
        if after→value != v:
            before→next = _node(v, after)
        release(?after→lock)
        release(?before→lock)

def remove(lst, v):
    let before, after = _find(lst, v):
        if after→value == v:
            before→next = after→next
            release(?after→lock)
            free(after)
        else:
            release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
        result = after→value == v
        release(?after→lock)
        release(?before→lock)
```

Figure 10.4: [code/linkedlist.hny] Linked list with fine-grained locking.

Sorted Integer Linked List with Lock per Node

```
def insert(lst, v):
    let before, after = _find(lst, v):
        if after→value != v:
            before→next = _node(v, after)
        release(?after→lock)
        release(?before→lock)

def remove(lst, v):
    let before, after = _find(lst, v):
        if after→value == v:
            before→next = after→next
            release(?after→lock)
            free(after)
        else:
            release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
        result = after→value == v
        release(?after→lock)
        release(?before→lock)
```

Multiple threads can access the list simultaneously, but they can't *overtake* one another

Figure 10.4: [code/linkedlist.hny] Linked list with fine-grained locking.

How to get more concurrency?

Idea: allow multiple read-only operations to execute concurrently

- In many cases, reads are much more frequent than writes

➔ reader/writer lock

Either:

- multiple readers, or
- a single writer

thus not:

- *a reader and a writer, nor*
- *multiple writers*

Reader/writer lock interface and invariants:

- RW.read_acquire()
  - get a read lock. Multiple threads can have the read lock simultaneously, but no thread can have a write lock simultaneously
- RW.read_release()
  - release a read lock. Other threads may still have the read lock. When the last read lock is released, a write lock may be acquired
- RW.write_acquire()
  - acquire the write lock. Only one thread can have a write lock, and if so no thread can have a read lock
- RW.write_release()
  - release the write lock. Allows other threads to either get a read or write lock

R/W Locks: test for mutual exclusion

```
import RW

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            RW.read_acquire(?rw)
            @rcs: assert atLabel(wcs) == ()
            RW.read_release(?rw)
        else:                          # .write
            RW.write_acquire(?rw)
            @wcs: assert (atLabel(wcs) == { (thread, ()): 1 }) and
                        (atLabel(rcs) == ())
            RW.write_release(?rw)

for i in {1..3}:
    spawn thread()
```

no writer

1 writer and no readers

Figure 11.1: [code/RWtest.hny] Test code for reader/writer locks.

Conditional Waiting

Cornell University Founded A.D. 1865

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Conditional Waiting

- So far we've shown how threads can wait for one another to avoid multiple threads in the critical section
- Sometimes there are other reasons:
  - Wait until queue is non-empty
  - Wait until there are no readers (or writers) in a reader/writer lock
  - …

Busy Waiting: not a good way

- Wait until queue is non-empty:

$$
\begin{array}{l}
\mathit{next} = \textbf{None} \\
\textbf{while } \mathit{next} == \textbf{None}: \\
\quad \mathit{next} = \text{queue.get}(q)
\end{array}
$$

Busy Waiting: not a good way

- Wait until queue is non-empty:

  next = **None**
  while *next* == **None**:
      next = queue.get(*q*)

- *wastes CPU cycles*
- *creates unnecessary contention*

Remember the recruiter...

Asked >100 candidates if they could implement two threads, where one thread had to wait for a signal from the other

none of them were able to do it without hints
only some of them were able to do it with hints

(as far as I know, none of them were Cornell grads ;-)

Can be done with busy-waiting

```
def T0():
    await done      # wait for signal
def T1():
    done = True     # send signal

done = False
spawn T0()
spawn T1()
```

Can be done with busy-waiting

```
def T0():
    await done
def T1():
    done = True

done = False
spawn T0()
spawn T1()
```

we don’t like busy waiting

Can be done with busy-waiting

```
def T0():
    await done
def T1():
    done = True

done = False
spawn T0()
spawn T1()
```

we don’t like
data races either!

Can be done with locks, awkwardly

```
import synch;

def T0():
        acquire(?condition)     # wait for signal
        # no release
def T1():
        # no acquire
        release(?condition)     # send signal

condition = Lock()
acquire(?condition)   # weird stuff during init…
spawn T0()
spawn T1()
```

Can be done with locks, awkwardly

```
import synch;

def T0():
      acquire(?condition)
      # no release
def T1():
      # no acquire
      release(?condition)

condition = Lock()
acquire(?condition)
spawn T0()
spawn T1()
```

locks should be nested

Enter *binary semaphores*

[Dijkstra 1962]

Binary Semaphore

- Boolean variable
- Three operations:
  - *binsema* = BinSema(False or True)
    - initialize *binsema*
  - acquire(?*binsema*)
    - waits until !*binsema* = False, then sets the !binsema to True.
  - release(?*binsema*)
    - set !*binsema* to False
    - can only be called if !*binsema* = True

Dijkstra was Dutch, like some

- He said ***P**robeer-te-verlagen* instead of acquire
- He said ***V**erhogen* instead of release
- Many people still use P/V when talking about semaphore operators
- Easier to remember:
  - ***P***rocure (acquire)
  - ***V***acate (release)

Difference with locks

Locks	Binary Semaphores
Initially “unlocked” (False)	Can be initialized to False or True
Acquired, then *released* by same thread	Can be *acquired* and *released* by different threads
Mostly used to implement critical sections	Can be used to implement critical sections as well as waiting for special conditions

but both are much like “*batons*” that are being passed

Binary Semaphore interface and implementation

```
def tas(lk):
    atomic:
        result = !lk
        !lk = True

def BinSema(acquired):
    result = acquired

def Lock():
    result = BinSema(False)

def acquire(binsema):
    await not tas(binsema)

def release(binsema):
    atomic:
        assert binsema
        !binsema = False
```

sema = BinSema(False or True)

acquire(?*sema*)

release(?*sema*)

Same example with semaphores

```
import synch;

def T0():
    acquire(?condition)   # wait for signal
def T1():
    release(?condition)   # send signal

condition = BinSema(True)
spawn T0()
spawn T1()
```

Same example with semaphores

```
import synch;

condition = BinSema(True)

def T0():
    acquire(?condition)   # wait for signal
def T1():
    release(?condition)   # send signal

spawn T0()
spawn T1()
```

What happens if T0 runs first?
What happens if T1 runs first?

Semaphores can be locks too

- *lk* = BinSema(False) # False-initialized
- acquire(?*lk*) # grab lock
- release(?*lk*) # release lock

Great, what else can one do with binary semaphores??

Conditional Critical Sections

- A critical section with a condition
- For example:
  - queue.get(), but wait until the queue is non-empty
    - don’t want two threads to run code at the same time, but also don’t want any thread to run queue.get() code when queue is empty
  - print(), but wait until the printer is idle
  - RW.read_acquire(), but only if there are no writers in the critical section
  - allocate 100 GPUs, when they become available
  - …

[Hoare 1973]

Multiple conditions

Some conditional critical sections can have multiple conditions:

- R/W lock: readers are waiting for writer to leave; writers are waiting for reader or writer to leave
- bounded queue: dequeuers are waiting for queue to be non-empty; enqueuers are waiting for queue to be non-full
- …

High-level idea: selective baton passing!

- When a thread wants to execute in the critical section, it needs the one baton
- Threads can be waiting for various conditions
  - such threads do not hold the baton
- When a thread with the baton leaves the critical section, it checks to see if there are threads waiting on a condition that now holds
- If so, it passes the baton to one such thread
- If not, the critical section is vacated, and the baton is free to pick up for another thread that comes along

“Split Binary Semaphores” [Hoare 1973]

- Implement baton passing with multiple binary semaphores
- If there are *N* conditions, you’ll need *N*+1 binary semaphores
  - one for each condition
  - one to enter the critical section in the first place
- Invariant: At most one of these semaphores is released (False)
  - If all are acquired (True), baton held by some thread
  - If one semaphore is released, no thread holds the baton
    - if it’s the “entry” semaphore, then no thread is waiting on a condition that holds, and any thread can enter
    - if it’s one of the condition semaphores, some thread that is waiting on the condition can now enter the critical section

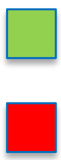
"Split Binary Semaphores" [Hoare 1973]

- Implement baton passing with multiple binary semaphores
- If there are *N* conditions, you'll need *N*+1 binary semaphores
  - one for each condition
  - one to enter the critical section in the first place
- Invariant: At most one of these semaphores is released (False)
  - If all are acquired (True), baton held by some thread
  - If one semaphore is released, no thread holds the baton
    - if it's the "entry" semaphore, then no thread is waiting on a condition that holds, and any thread can enter
    - if it's one of the condition semaphores, some thread that is waiting on the condition can now enter the critical section
      - at most one

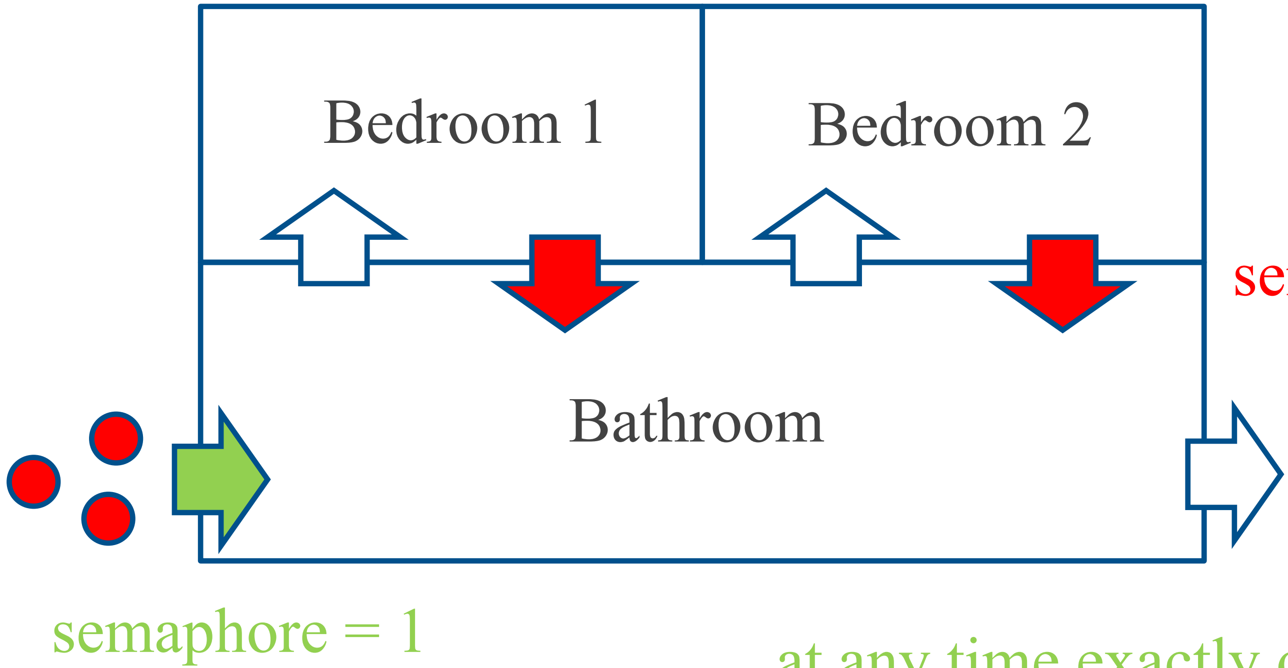
“Split Binary Semaphores” [Hoare 1973]

- Implement baton passing with multiple binary semaphores
- If there are *N* conditions, you’ll need *N*+1 binary semaphores
  - one for each condition
  - one to enter the critical section in the first place
- Invariant: At most one of these semaphores is released (False)
  - If all are acquired (True), baton held by some thread
  - If one semaphore is released, no thread holds the baton
    - if it’s the “entry” semaphore, then no thread is waiting on a condition that holds, and any thread can enter
    - if it’s one of the condition semaphores, some thread that is waiting on the condition can now enter the critical section
      - at most one
      - at least one

Bathroom humor…



3 threads want to enter critical section



Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green (and thus, at most one semaphore is green)

This is a model of:

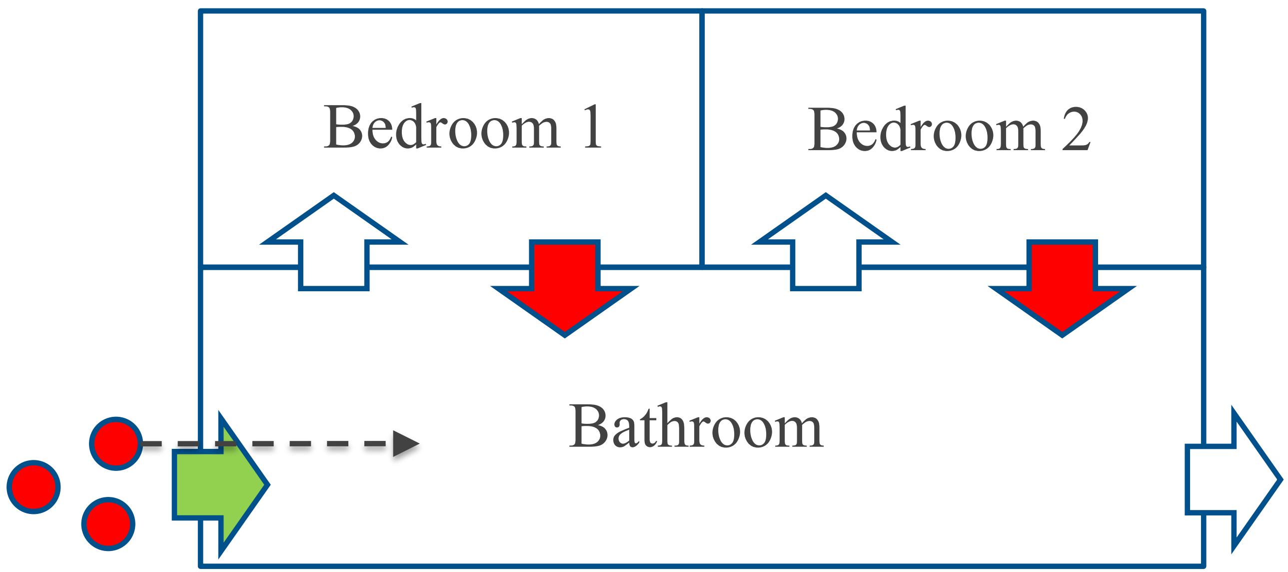
- Reader/writer lock:
  - Bathroom: critical section
  - Bedroom 1: readers waiting for writer to leave
  - Bedroom 2: writers waiting for readers or writers to leave
- Bounded queue:
  - Bathroom: critical section
  - Bedroom 1: dequeuers waiting for queue to be non-empty
  - Bedroom 2: enqueuers waiting for queue to be non-full
- …

Bathroom humor…

holds baton

does not hold baton

3 threads want to enter critical section

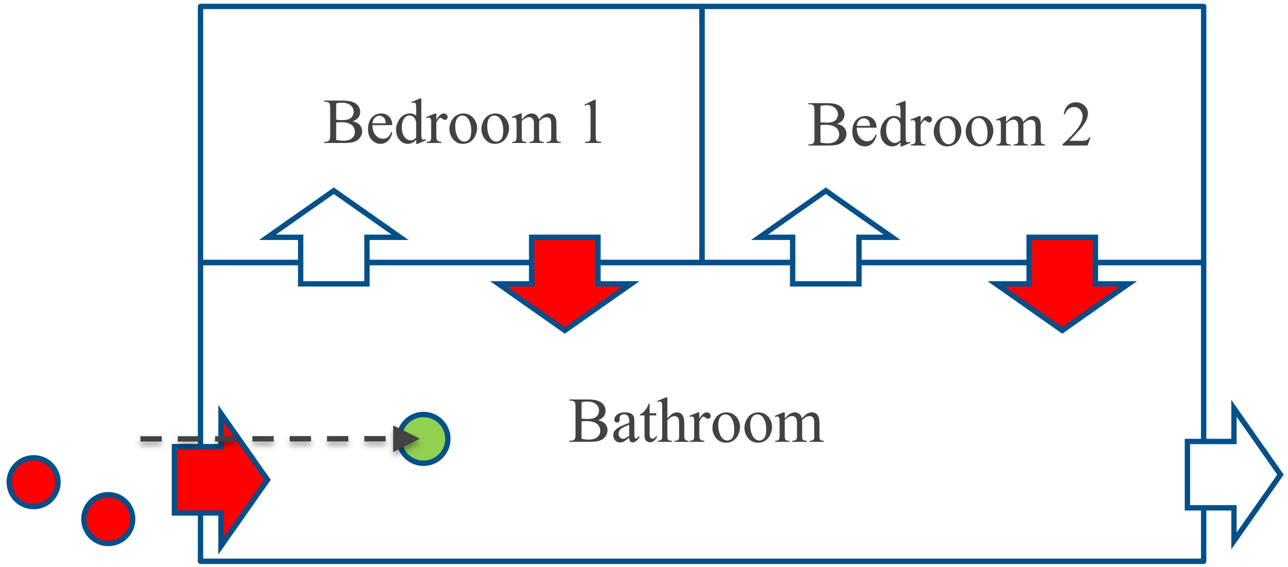


Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green

Bathroom humor…

1 thread entered the critical section



Bathroom: critical section
Bedrooms: waiting conditions

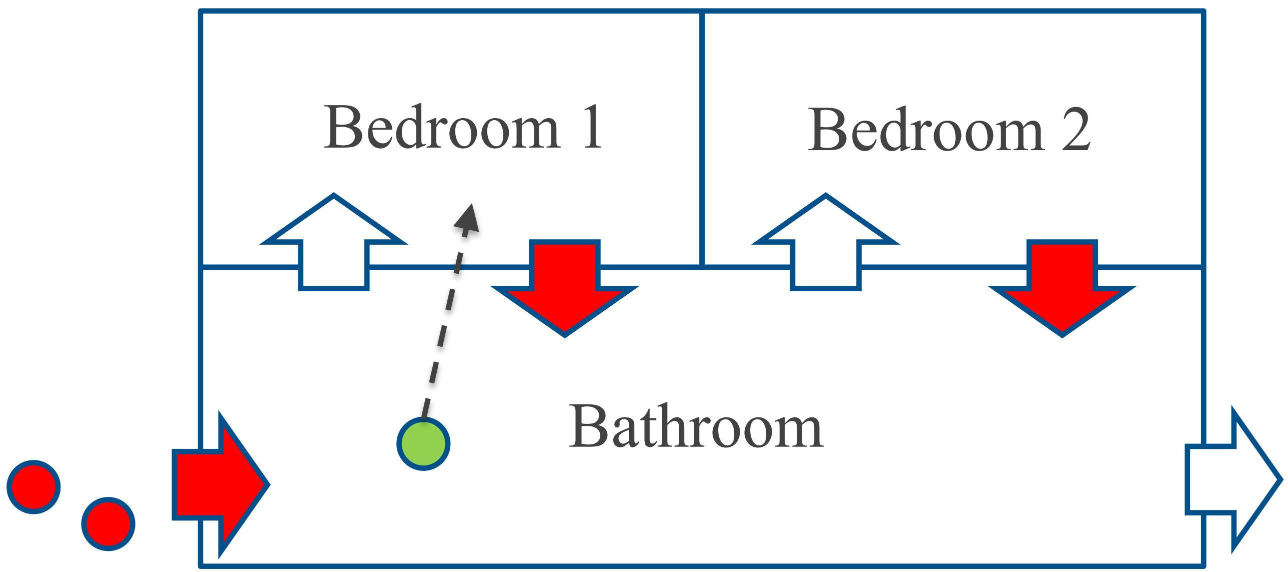
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

thread needs to wait for Condition 1

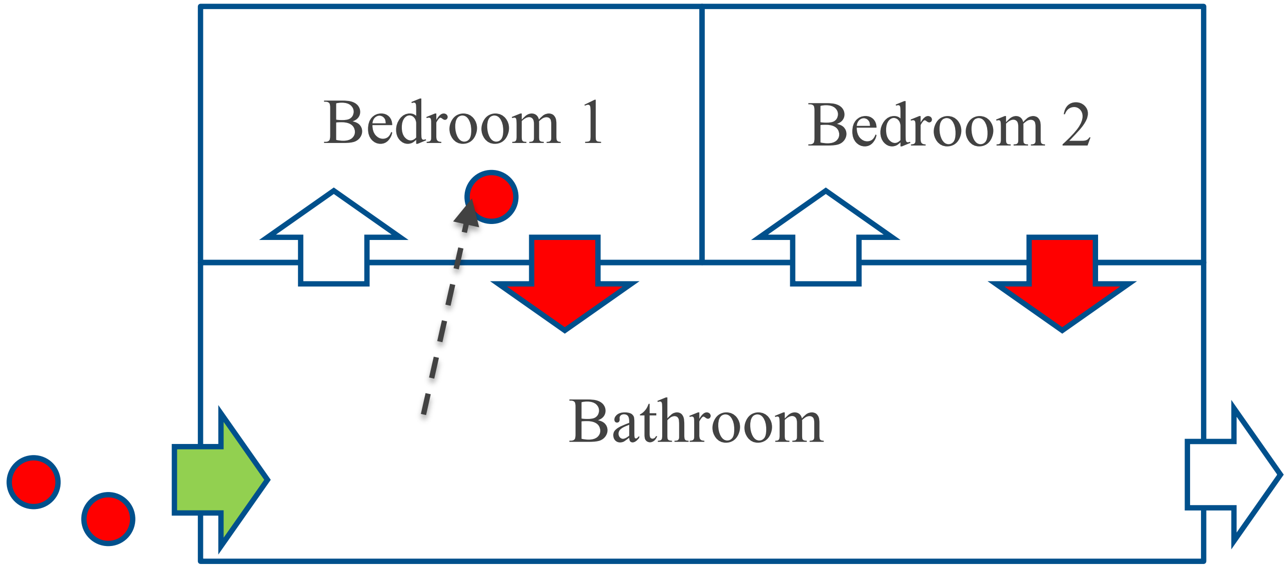


at any time exactly one semaphore or thread is green

Bathroom: critical section
Bedrooms: waiting conditions

Bathroom humor…

no thread waiting for condition that holds



Bathroom: critical section
Bedrooms: waiting conditions

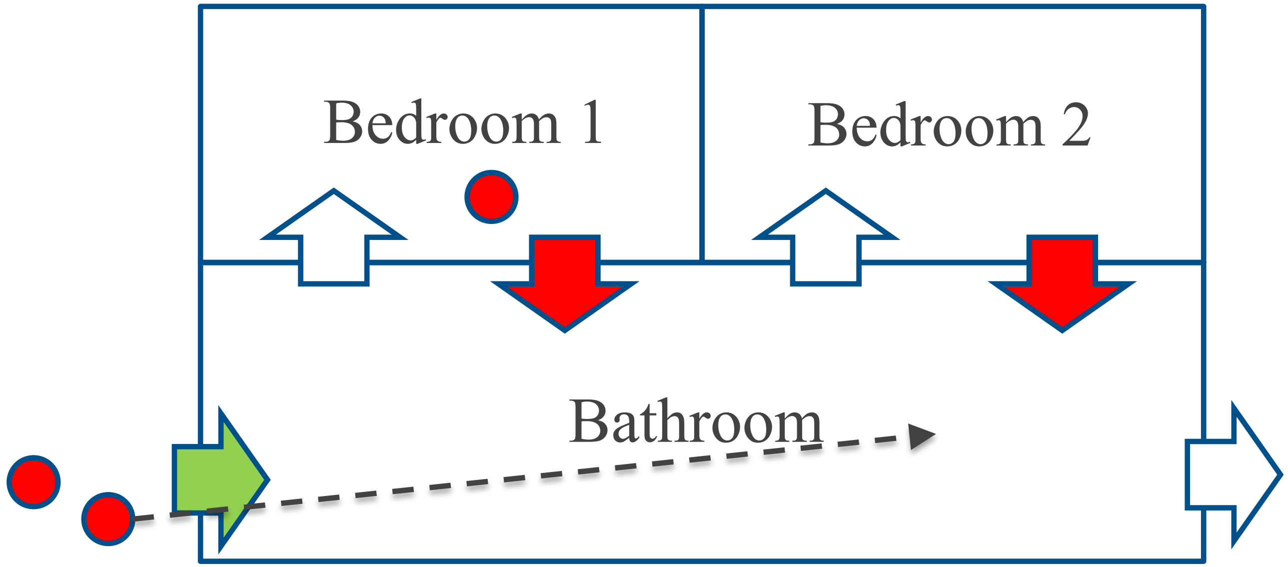
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

another thread can enter the critical section



at any time exactly one semaphore or thread is green

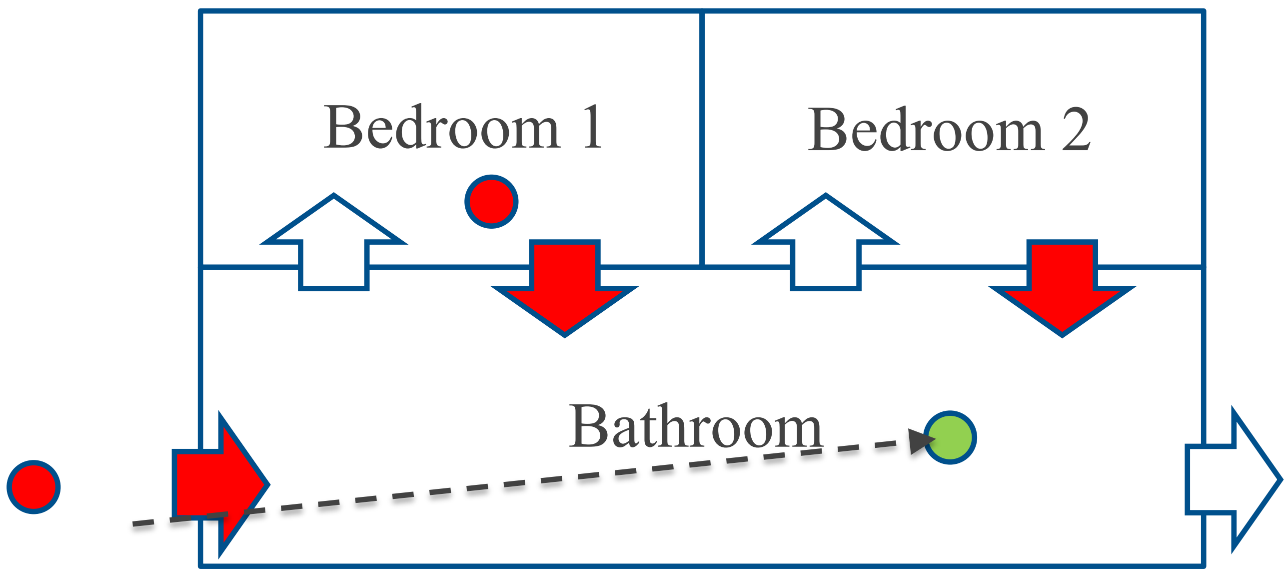
Bathroom: critical section
Bedrooms: waiting conditions

Bathroom humor…

holds baton

does not hold baton

thread entered the critical section



at any time exactly one semaphore or thread is green

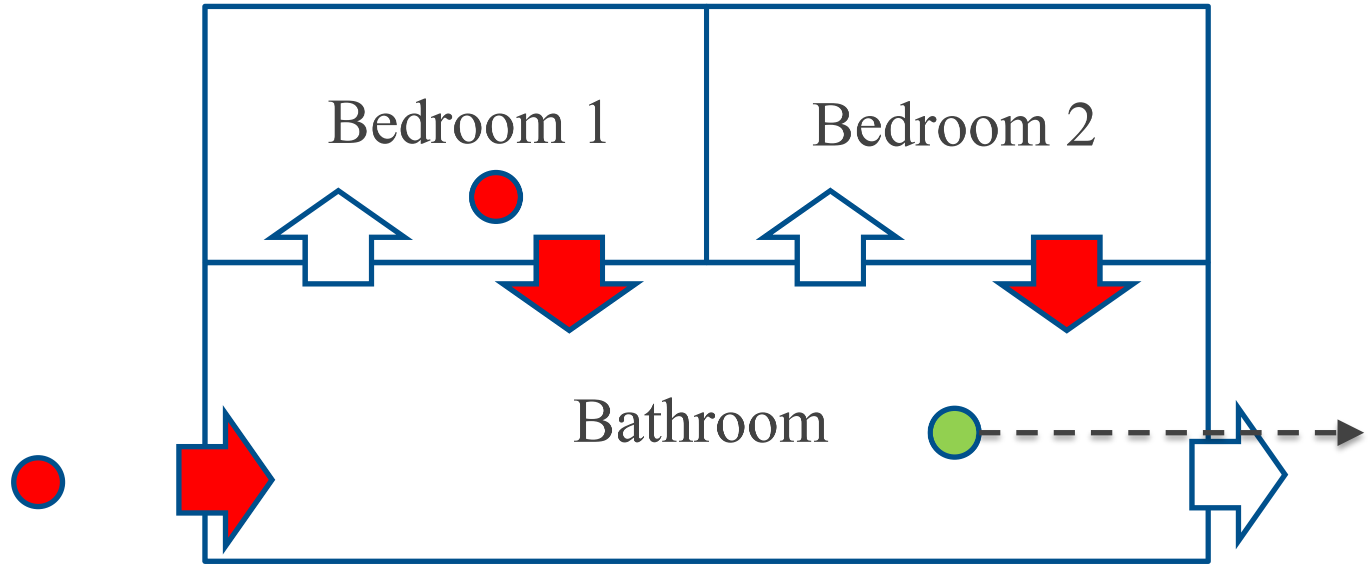
Bathroom: critical section
Bedrooms: waiting conditions

Bathroom humor…

holds baton

does not hold baton

thread enables Condition 1 and wants to leave



Bathroom: critical section
Bedrooms: waiting conditions

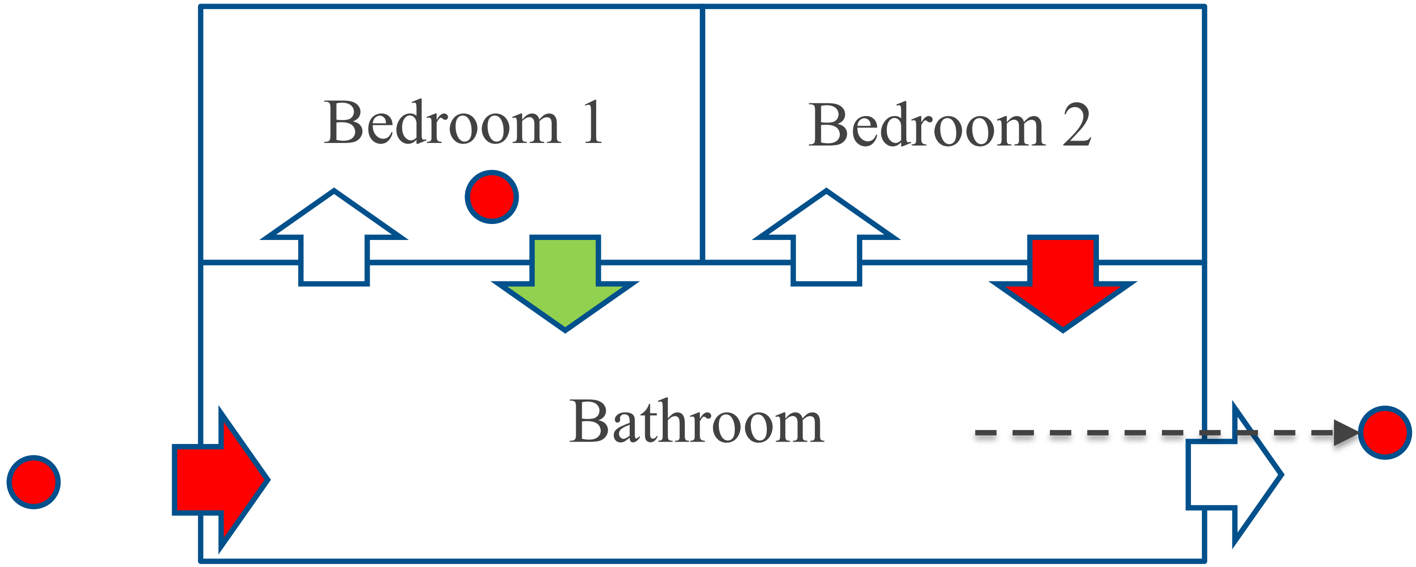
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

thread left, Condition 1 holds



Bathroom: critical section
Bedrooms: waiting conditions

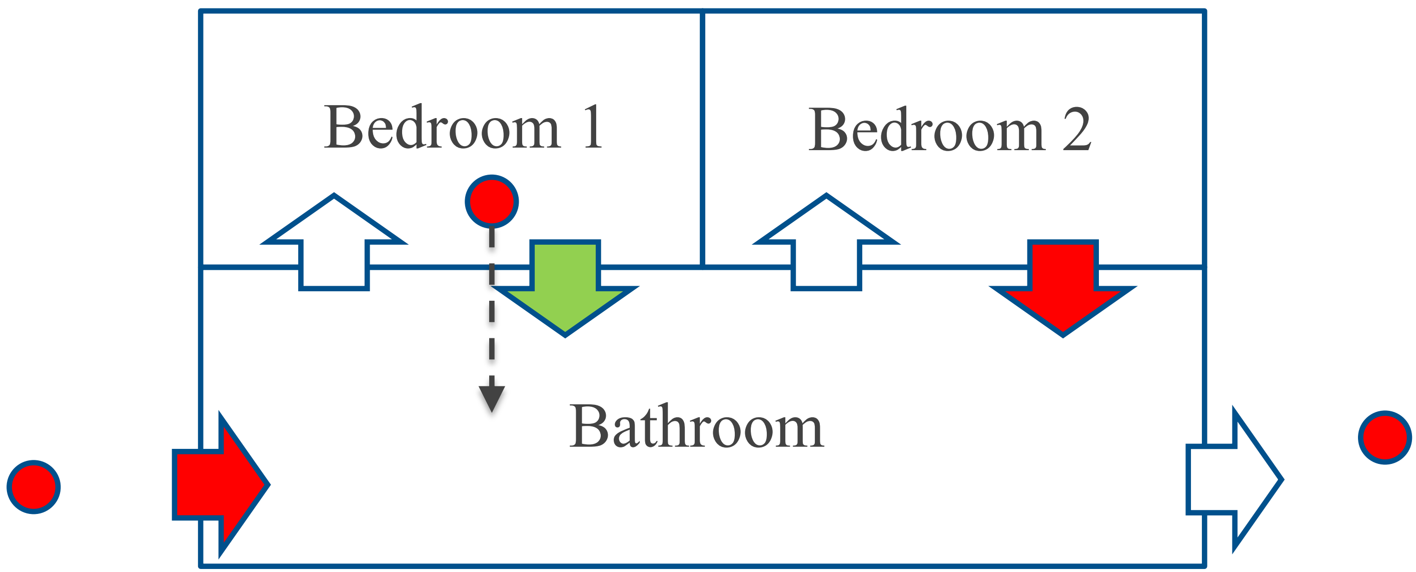
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

first thread (and only first thread) can enter critical section again



Bathroom: critical section
Bedrooms: waiting conditions

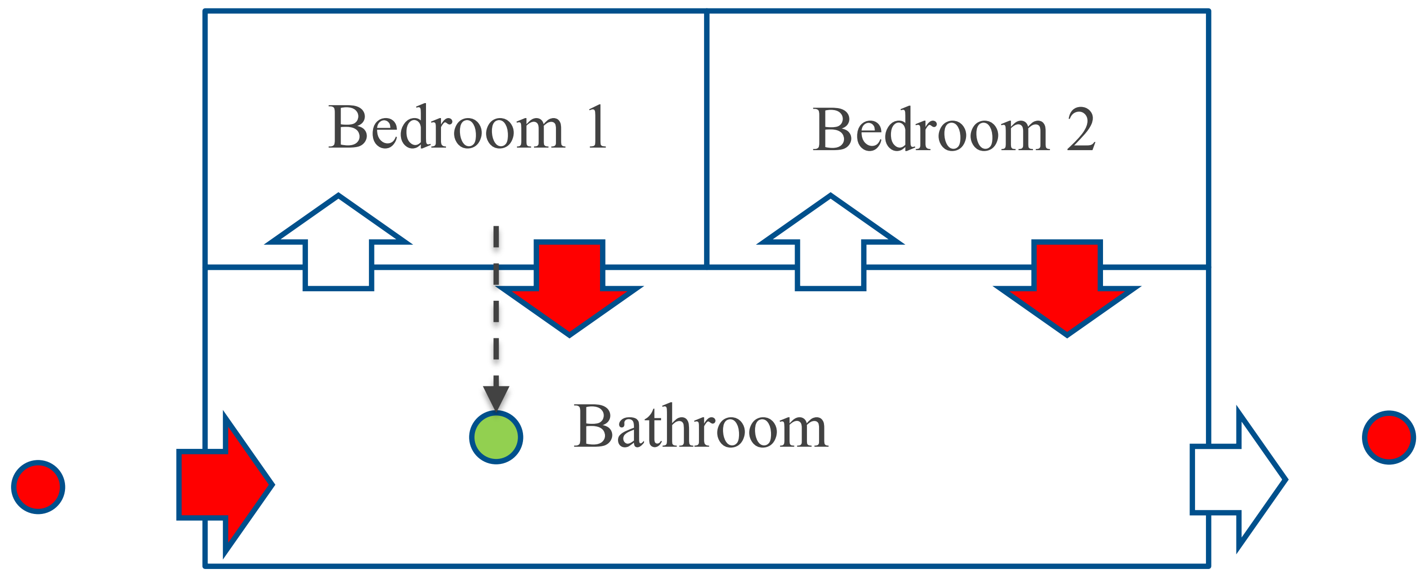
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

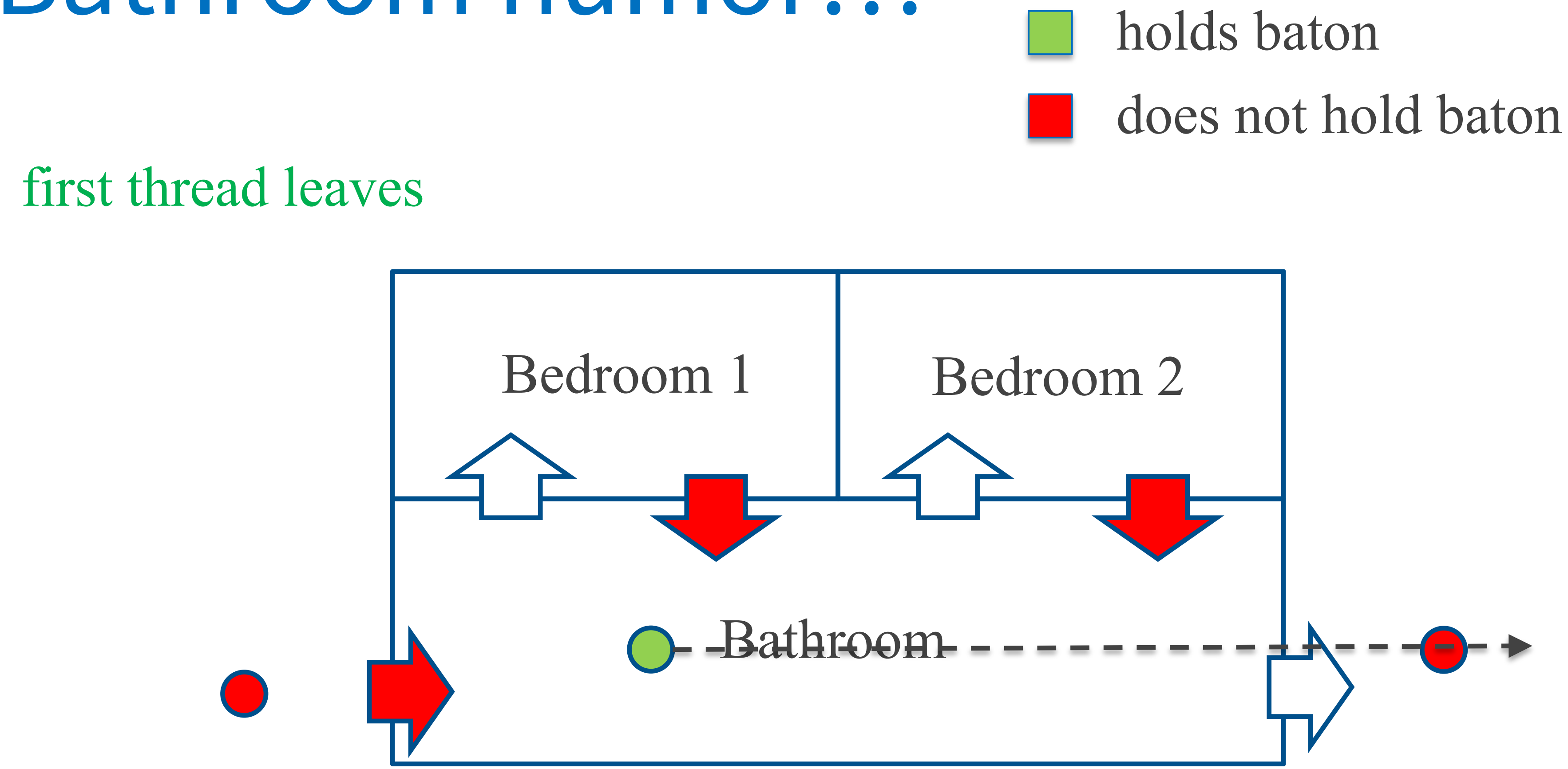
first thread entered critical section again



Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green

Bathroom humor...

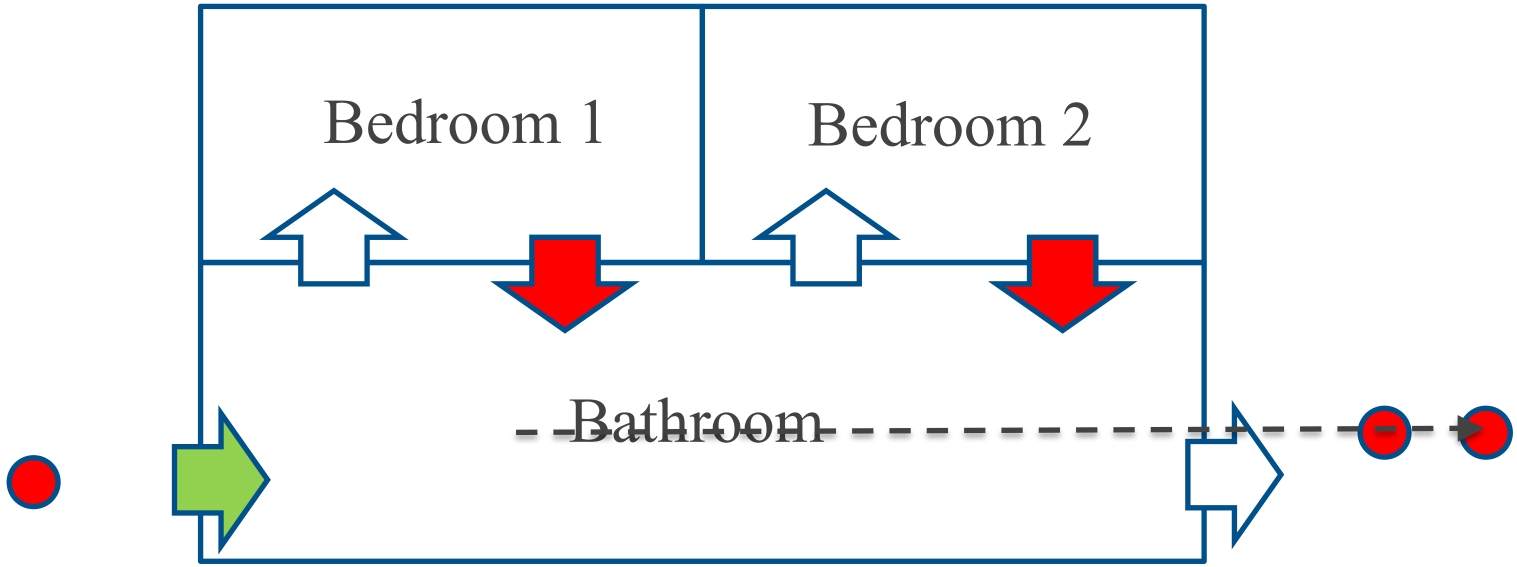


at any time exactly one semaphore or thread is green

Bathroom: critical section
Bedrooms: waiting conditions

Bathroom humor…

first thread done



Bathroom: critical section
Bedrooms: waiting conditions

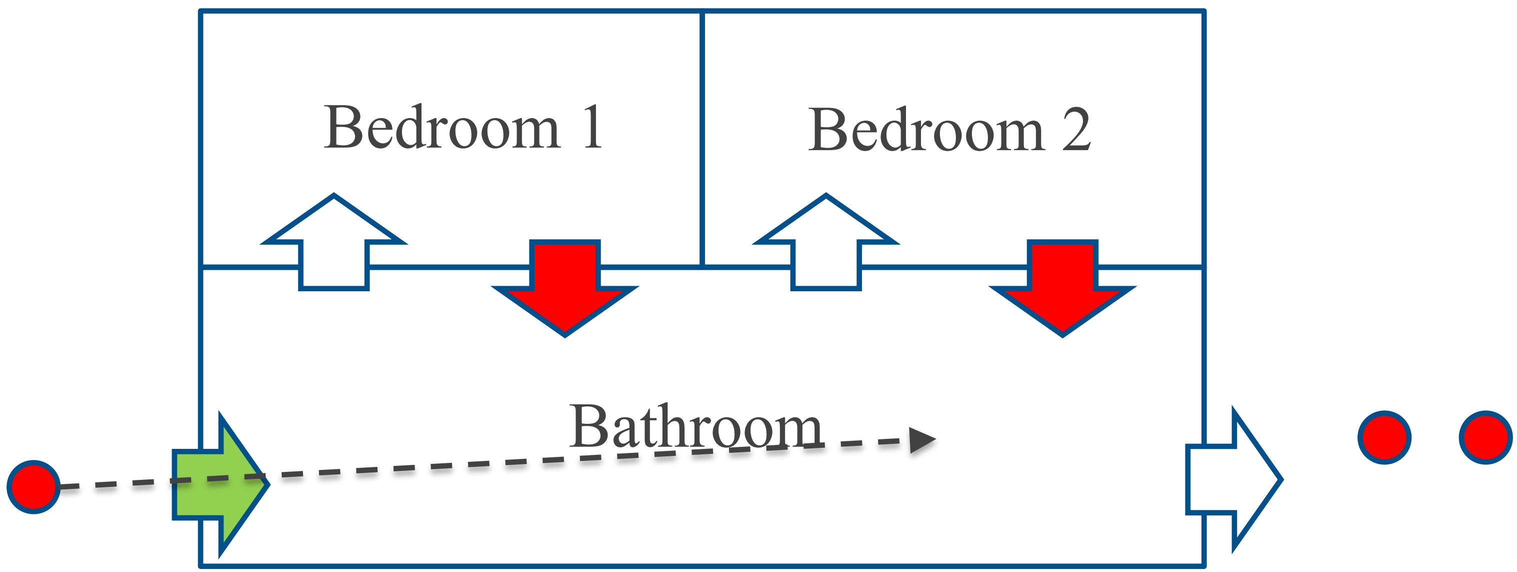
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

one thread wants to enter the critical section

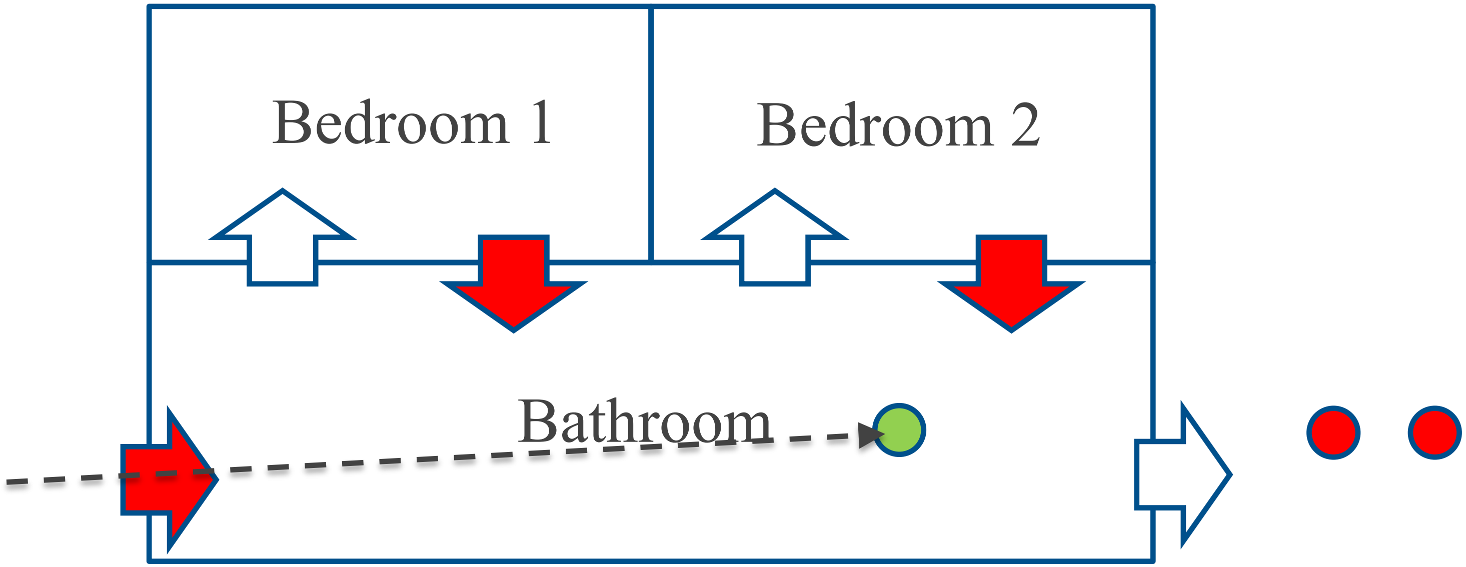


Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green

Bathroom humor…

last thread entered critical section



at any time exactly one semaphore or thread is green

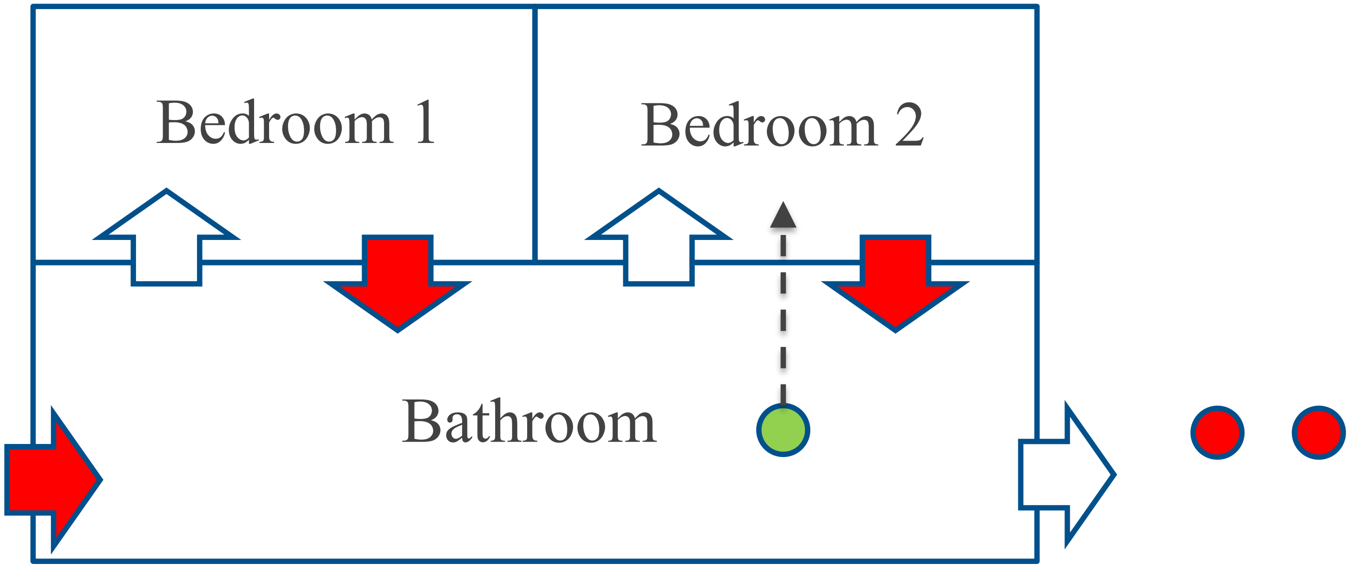
Bathroom: critical section
Bedrooms: waiting conditions

Bathroom humor…

holds baton

does not hold baton

thread needs to wait for Condition 2



Bathroom: critical section
Bedrooms: waiting conditions

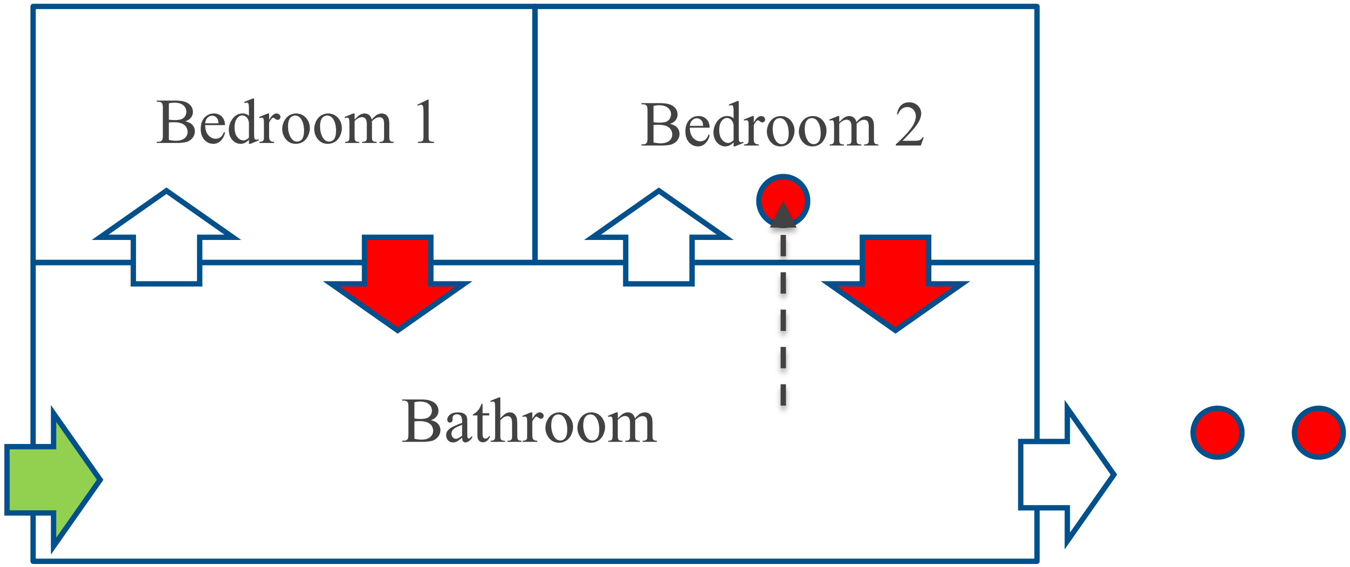
at any time exactly one semaphore or thread is green

Bathroom humor…

holds baton

does not hold baton

thread waiting for Condition 2



at any time exactly one semaphore or thread is green

Bathroom: critical section
Bedrooms: waiting conditions

Let’s build a Reader/Writer lock this way

- You may have seen other ways
- There are many ways that lead to Rome

from https://www.r-bloggers.com/2019/07/all-roads-lead-to-rome-2/

Reader/writer lock interface and invariants:

- RW.read_acquire()
  - get a read lock. Multiple threads can have the read lock simultaneously, but no thread can have a write lock simultaneously
- RW.read_release()
  - release a read lock. Other threads may still have the read lock. When the last read lock is released, a write lock may be acquired
- RW.write_acquire()
  - acquire the write lock. Only one thread can have a write lock, and if so no thread can have a read lock
- RW.write_release()
  - release the write lock. Allows other threads to either get a read or write lock

R/W Locks: test for mutual exclusion

```
import RW

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ .read, .write }) == .read:
            RW.read_acquire(?rw)
            @rcs: assert atLabel(wcs) == ()
            RW.read_release(?rw)
        else:                           # .write
            RW.write_acquire(?rw)
            @wcs: assert (atLabel(wcs) == { (thread, ()): 1 }) and
                        (atLabel(rcs) == ())
            RW.write_release(?rw)

for i in {1..3}:
    spawn thread()
```

no writer

1 writer and no readers

Figure 11.1: [code/RWtest.hny] Test code for reader/writer locks.

Reader/writer lock: implementation

```
from synch import BinSema, acquire, release

def RWlock():
    result = {
            .nreaders: 0, .nwriters: 0, .mutex: BinSema(False),
            .r_gate: { .sema: BinSema(True), .count: 0 },
            .w_gate: { .sema: BinSema(True), .count: 0 }
        }
```

Accounting:

- *nreaders*: #readers in the critical section
- *r_gate.count*: #readers waiting to enter the critical section
- *nwriters*: #writers in the critical section
- *w_gate.count*: #writers waiting to enter the critical section

Invariants:

- if n readers in the critical section, then $nreaders \geq n$
- if n writers in the critical section, then $nwriters \geq n$
- $(nreaders \geq 0 \land nwriters = 0) \lor (nreaders = 0 \land 0 \leq nwriters \leq 1)$

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

enter main gate

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

leave

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

enter reader gate

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

going through

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

leave: let others try too

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

no special waiting condition

Reader/writer lock: read

```
def read_acquire(rw):
    acquire(?rw→mutex)
    if rw→nwriters > 0:
        rw→r_gate.count += 1; release_one(rw)
        acquire(?rw→r_gate.sema); rw→r_gate.count −= 1
    rw→nreaders += 1
    release_one(rw)

def read_release(rw):
    acquire(?rw→mutex); rw→nreaders −= 1; release_one(rw)
```

Note that acquire/release operations alternate

Reader/writer lock: write

```
def write_acquire(rw):
    acquire(?rw→mutex)
    if (rw→nreaders + rw→nwriters) > 0:
        rw→w_gate.count += 1; release_one(rw)
        acquire(?rw→w_gate.sema); rw→w_gate.count −= 1
    rw→nwriters += 1
    release_one(rw)

def write_release(rw):
    acquire(?rw→mutex); rw→nwriters −= 1; release_one(rw)
```

Reader/writer lock: write

different waiting condition

```
def write_acquire(rw):
    acquire(?rw→mutex)
    if (rw→nreaders + rw→nwriters) > 0:
        rw→w_gate.count += 1; release_one(rw)
        acquire(?rw→w_gate.sema); rw→w_gate.count −= 1
    rw→nwriters += 1
    release_one(rw)

def write_release(rw):
    acquire(?rw→mutex); rw→nwriters −= 1; release_one(rw)
```

Reader/writer lock: write

```
def write_acquire(rw):
    acquire(?rw→mutex)
    if (rw→nreaders + rw→nwriters) > 0:
        rw→w_gate.count += 1; release_one(rw)
        acquire(?rw→w_gate.sema); rw→w_gate.count −= 1
    rw→nwriters += 1
    release_one(rw)

def write_release(rw):
    acquire(?rw→mutex); rw→nwriters −= 1; release_one(rw)
```

different waiting gate

Reader/writer lock: leaving

```
def release_one(rw):
    if (rw→nwriters == 0) and (rw→r_gate.count > 0):
        release(?rw→r_gate.sema)
    elif ((rw→nreaders + rw→nwriters) == 0) and (rw→w_gate.count > 0):
        release(?rw→w_gate.sema)
    else:
        release(?rw→mutex)
```

Reader/writer lock: leaving

```
def release_one(rw):
    if (rw→nwriters == 0) and (rw→r_gate.count > 0):
        release(?rw→r_gate.sema)
    elif ((rw→nreaders + rw→nwriters) == 0) and (rw→w_gate.count > 0):
        release(?rw→w_gate.sema)
    else:
        release(?rw→mutex)
```

When leaving critical section:

- if no writers in the Critical Section and there are readers waiting
  then let a reader in
- else if no readers nor writer in the C.S. and there are writers waiting
  then let a writer in
- otherwise
  let any new thread in

Reader/writer lock: leaving

```
def release_one(rw):
    if (rw→nwriters == 0) and (rw→r_gate.count > 0):
        release(?rw→r_gate.sema)
    elif ((rw→nreaders + rw→nwriters) == 0) and (rw→w_gate.count > 0):
        release(?rw→w_gate.sema)
    else:
        release(?rw→mutex)
```

When leaving critical section:

- if no writers in the Critical Section and there are readers waiting
  then let a reader in
- else if no readers nor writer in the C.S. and there are writers waiting
  then let a writer in
- otherwise
  let any new thread in

- Can the two conditions be reversed?
- What is the effect of that?

Making R/W lock starvation-free

- Last implementation suffers from starvation

Making R/W lock starvation-free

- Last implementation suffers from starvation
  - steady stream of new readers lock out writers

Making R/W lock starvation-free

- change the waiting and release conditions:
  - when a reader tries to enter the critical section, wait if there is a writer in the critical section OR if there are writers waiting to enter the critical section
  - exiting reader prioritizes releasing a waiting writer
  - exiting writer prioritizes releasing a waiting reader

See Harmony book

Conditional Critical Sections

We now know of two ways to implement them:

Busy Waiting	Split Binary Semaphores
Wait for condition in loop, acquiring lock before testing condition and releasing it if the condition does not hold	Use a collection of binary semaphores and keep track of state including information about waiting threads
Easy to understand the code	State tracking is complicated
Ok-ish for true multi-core, but bad for virtual threads	Good for both multi-core and virtual threading

Language support?

- Can’t the programming language be more helpful here?
  - Helpful syntax
  - Or at least some library support

“Hoare” Monitors

- Tony Hoare 1974
  - similar construct given by Per Brinch-Hansen 1973
- Syntactic sugar around split binary semaphores

```
single resource:monitor
begin busy:Boolean;
    nonbusy:condition;
  procedure acquire;
    begin if busy then nonbusy.wait;
              busy := true
    end;
  procedure release;
    begin busy := false;
            nonbusy.signal
    end;
  busy := false; comment initial value;
end single resource
```

“condition variable”

wait method

signal method

Hoare Monitors in Harmony

```
import synch

def Monitor():
    result = synch.Lock()

def enter(mon):
    synch.acquire(mon)

def exit(mon):
    synch.release(mon)

def Condition():
    result = { .sema: synch.BinSema(True), .count: 0 }

def wait(cond, mon):
    cond→count += 1
    exit(mon)
    synch.acquire(?cond→sema)
    cond→count −= 1

def signal(cond, mon):
    if cond→count > 0:
        synch.release(?cond→sema)
        enter(mon)
```

Hoare Monitors in Harmony

```
import synch

def Monitor():
    result = synch.Lock()

def enter(mon):
    synch.acquire(mon)

def exit(mon):
    synch.release(mon)

def Condition():
    result = { .sema: synch.BinSema(True), .count: 0 }

def wait(cond, mon):
    cond→count += 1
    exit(mon)
    synch.acquire(?cond→sema)
    cond→count −= 1

def signal(cond, mon):
    if cond→count > 0:
        synch.release(?cond→sema)
        enter(mon)
```

main gate

Hoare Monitors in Harmony

```
import synch

def Monitor():
    result = synch.Lock()
```

main gate

```
def enter(mon):
    synch.acquire(mon)

def exit(mon):
    synch.release(mon)

def Condition():
    result = { .sema: synch.BinSema(True), .count: 0 }
```

waiting gate

```
def wait(cond, mon):
    cond→count += 1
    exit(mon)
    synch.acquire(?cond→sema)
    cond→count −= 1

def signal(cond, mon):
    if cond→count > 0:
        synch.release(?cond→sema)
        enter(mon)
```

Hoare Monitors in Harmony

```
import synch

def Monitor():
    result = synch.Lock()
```

main gate

```
def enter(mon):
    synch.acquire(mon)

def exit(mon):
    synch.release(mon)

def Condition():
    result = { .sema: synch.BinSema(True), .count: 0 }
```

waiting gate

```
def wait(cond, mon):
    cond→count += 1
    exit(mon)
    synch.acquire(?cond→sema)
    cond→count −= 1

def signal(cond, mon):
    if cond→count > 0:
        synch.release(?cond→sema)
        enter(mon)
```

passes control immediately

Example: bounded buffer (aka producer/consumer)

```
import hoare

def BB(size):
    result = {
            .mon: hoare.Monitor(),
            .prod: hoare.Condition(), .cons: hoare.Condition(),
            .buf: { x:() for x in {1..size} },
            .head: 1, .tail: 1,
            .count: 0, .size: size
        }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)
```

Example: bounded buffer (aka producer/consumer)

```
import hoare

def BB(size):
    result = {
            .mon: hoare.Monitor(),
            .prod: hoare.Condition(), .cons: hoare.Condition(),
            .buf: { x:() for x in {1..size} },
            .head: 1, .tail: 1,
            .count: 0, .size: size
        }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)
```

N+1 semaphores abstracted away

Example: bounded buffer (aka producer/consumer)

```
import hoare

def BB(size):
    result = {
            .mon: hoare.Monitor(),
            .prod: hoare.Condition(), .cons: hoare.Condition(),
            .buf: { x:() for x in {1..size} },
            .head: 1, .tail: 1,
            .count: 0, .size: size
        }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)
```

circular buffer

Example: bounded buffer (aka producer/consumer)

```
import hoare

def BB(size):
    result = {
            .mon: hoare.Monitor(),
            .prod: hoare.Condition(), .cons: hoare.Condition(),
            .buf: { x:() for x in {1..size} },
            .head: 1, .tail: 1,
            .count: 0, .size: size
        }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)
```

enter monitor

exit monitor

Example: bounded buffer (aka producer/consumer)

```
import hoare

def BB(size):
    result = {
            .mon: hoare.Monitor(),
            .prod: hoare.Condition(), .cons: hoare.Condition(),
            .buf: { x:() for x in {1..size} },
            .head: 1, .tail: 1,
            .count: 0, .size: size
        }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)
```

wait if full

signal a consumer

Example: bounded buffer (aka producer/consumer)

```
import hoare

def BB(size):
    result = {
        .mon: hoare.Monitor(),
        .prod: hoare.Condition(), .cons: hoare.Condition(),
        .buf: { x:() for x in {1..size} },
        .head: 1, .tail: 1,
        .count: 0, .size: size
    }

def put(bb, item):
    hoare.enter(?bb→mon)
    if bb→count == bb→size:
        hoare.wait(?bb→prod, ?bb→mon)
    bb→buf[bb→tail] = item
    bb→tail = (bb→tail % bb→size) + 1
    bb→count += 1
    hoare.signal(?bb→cons, ?bb→mon)
    hoare.exit(?bb→mon)
```

signal() passes baton *immediately* if there are threads waiting on the given condition variable

Thursday, March 18th, 2021

- Review
  - Conditional Critical Sections
    - Busy waiting
      - Wasteful
    - Split Binary Semaphores
      - Hoare monitors

The Steminist Movement, Inc. and TSMCORNELL present

The Voice of Perseverance's Landing on Mars:

Swati Mohan's Journey to JPL

Saturday, March 20, 2021
7 – 8 PM EST

Guidance and Controls Operations Lead on the **NASA Mars 2020 Mission**

STEM CORNELL

Swati Mohan

From Wikipedia, the free encyclopedia

Swati Mohan is an Indian-American aerospace engineer and was the Guidance and Controls Operations Lead on the NASA Mars 2020 mission.[1]

Contents [hide]

Early life and education [edit]

Mohan was born in Bengaluru, Karnataka, India, and emigrated to the United States when she was one year old.[2][3][4] She became interested in space upon seeing Star Trek at age 9.[5] She had originally planned to be a pediatrician but at the age of 16 took a physics class and decided to study engineering as a way to pursue a career in space exploration.[6][5] She studied Mechanical and Aerospace Engineering at Cornell University, before completing

Swati Mohan

Swati Mohan

Education	Cornell University (B.S.) Massachusetts Institute of Technology (M.S., Ph.D.)
Known for	Work on the Mars 2020 mission
Scientific career	
Institutions	NASA's Jet Propulsion Laboratory
Thesis	*Quantitative Selection and Design of Model Generation Architectures for On-Orbit Autonomous*

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

Help
Learn to edit
Community portal
Recent changes
Upload file

Tools

What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export

Mohan works at NASA's Jet Propulsion Laboratory in Pasadena, California, and is the Guidance & Controls Operations Lead for the Mars 2020 mission.[1] Mohan joined the Mars 2020 team in 2013, shortly after the team was assembled.[15][16] In her role, she was responsible for ensuring the spacecraft that carries the rover was properly oriented during its travel to Mars and when landing on the planet's surface.[15][8][17] She narrated the landing events from inside mission control as the *Perseverance* rover landed on Mars on 18 February 2021.[1] She announced "Touchdown is confirmed," after which the JPL Mission Control Center erupted in celebration, clapping and fist bumping

Split Binary Semaphore rules

- *N*+1 binary semaphores
  - 1 “entry” semaphore and *N* condition semaphores
- Initially only the “entry” semaphore is False (released)
- At most one semaphore can be False
  - ➔ each thread should start with an acquire operation, alternate release and acquire operations, and end on a release
  - ➔ never two acquires or two releases in a row!!!!
- Keep careful track of state in shared variables
  - including one #waiting counter per condition
- Only access variables when all semaphores are True

This “recipe” works for any synchronization problem where the number of conditions is fixed

Reader/writer lock

- 2 waiting conditions ➔ $N = 3$
  - reader waits for no writers
  - writer waits for no readers or writers

Layers of Abstraction

- Note that we have two layers of abstraction:
  - The reader/writer lock object
  - The binary semaphore object
- Both can be used to implement critical section:
  - R/W locks allow multiple readers in a critical section
  - split binary semaphores allow only one thread at a time in a critical section
- These are *not the same critical sections*
  - they occur at different levels of abstraction

Another example: lockbox

- to enter house, you need the key
- to get the key out of the lockbox, you need the code
- the house and the lockbox are both critical sections
- to enter the house you:
  1. open the lockbox
  2. open the house with the key
  3. put the key back in the lockbox and close it
- to lock the house you:
  1. open the lockbox
  2. get the key and lock the house
  3. put the key back in the lockbox and close it

Why is this useful?

- Because it implements an interesting rule:
  - multiple people can get into the house
  - but only if they have lockbox access
- Could design fancier rules, for example:
  - put three marbles in the lockbox
  - to enter the house, you have to remove a marble and take it with you
  - when leaving the house, you have to put the marble back in
- *What does that accomplish?*

Same with R/W locks

- R/W lock:
  - key to the house
  - house allows one writer *or* multiple readers
    - but not both
- Split Binary Semaphore:
  - lockbox
  - + 1 marble (taken by writer)
  - + 1 (tiny) abacus (updated by readers)

Hoare Monitors

- Split Binary Semaphores underneath the "monitor" programming language paradigm
  - monitor: one thread can execute at a time
  - wait(condition variable): thread waits for given condition
  - signal(condition variable): transfer control to a thread waiting for the given condition, if any

Mesa Monitors

- Introduced in the Mesa language
  - Xerox PARC, 1980
- Syntactically similar to Hoare monitors
  - monitors and condition variables
- *Semantically closer to busy waiting approach*
  - wait(condition variable): wait for condition, *but may wake up before condition is not satisfied*
  - notify(condition variable): wake up a thread waiting for the condition, if any, *but don't transfer control*
  - notifyAll(condition variable): wake up all threads waiting for the condition, *but don't transfer control*

This is hugely different from Hoare monitors

Hoare vs Mesa Monitors

Hoare monitors	Mesa monitors
Baton passing approach	Sleep + try again
signal passes baton	notify(all) wakes sleepers

Mesa monitors won the test of time…

Mesa Monitors in Harmony

```
def Condition(lk):
    result = bag.empty()

def wait(c, lk):
    let blocked, cnt, ctx = True, 0, get_context():
        atomic:
            cnt = bag.count(!c, ctx)
            bag.add(c, ctx)
            !lk = False
        while blocked:
            atomic:
                if (not !lk) and (bag.count(!c, ctx) <= cnt):
                    !lk = True
                    blocked = False

def notify(c):
    atomic:
        if !c != bag.empty():
            bag.remove(c, bag.bchoose(!c))

def notifyAll(c):
    !c = bag.empty()
```

Condition: consists of bag of threads waiting

wait: unlock + add thread context to bag of waiters

notify: remove one waiter from the bag of suspended threads

notifyAll: remove *all* waiters from the list of suspended threads

R/W lock with Mesa monitors

```
from synch import *

def RWlock():
    result = {
            .nreaders: 0, .nwriters: 0, .mutex: Lock(),
            .r_cond: Condition(), .w_cond: Condition()
        }
```

Invariants:

- if n readers in the R/W critical section, then $nreaders \geq n$
- if n writers in the R/W critical section, then $nwriters \geq n$
- $(nreaders \geq 0 \land nwriters = 0) \lor (nreaders = 0 \land 0 \leq nwriters \leq 1)$

mutex protects the *nreaders*/*nwriters* variables, not the R/W critical section!

R/W Lock, reader part

```
def read_acquire(rw):
    acquire(?rw→mutex)
    while rw→nwriters > 0:
        wait(?rw→r_cond, ?rw→mutex)
    rw→nreaders += 1
    release(?rw→mutex)

def read_release(rw):
    acquire(?rw→mutex)
    rw→nreaders -= 1
    if rw→nreaders == 0:
        notify(?rw→w_cond)
    release(?rw→mutex)
```

R/W Lock, reader part

```
def read_acquire(rw):
    acquire(?rw→mutex)
    while rw→nwriters > 0:
        wait(?rw→r_cond, ?rw→mutex)
    rw→nreaders += 1
    release(?rw→mutex)

def read_release(rw):
    acquire(?rw→mutex)
    rw→nreaders −= 1
    if rw→nreaders == 0:
        notify(?rw→w_cond)
    release(?rw→mutex)
```

similar to busy waiting

R/W Lock, reader part

```
def read_acquire(rw):
    acquire(?rw→mutex)
    while rw→nwriters > 0:
        wait(?rw→r_cond, ?rw→mutex)
    rw→nreaders += 1
    release(?rw→mutex)

def read_release(rw):
    acquire(?rw→mutex)
    rw→nreaders −= 1
    if rw→nreaders == 0:
        notify(?rw→w_cond)
    release(?rw→mutex)
```

similar to busy waiting

but need this

R/W Lock, writer part

```
def write_acquire(rw):
    acquire(?rw→mutex)
    while (rw→nreaders + rw→nwriters) > 0:
        wait(?rw→w_cond, ?rw→mutex)
    rw→nwriters = 1
    release(?rw→mutex)

def write_release(rw):
    acquire(?rw→mutex)
    rw→nwriters = 0
    notifyAll(?rw→r_cond)
    notify(?rw→w_cond)
    release(?rw→mutex)
```

don’t forget anybody!

Conditional Critical Sections

We now know of *three* ways to implement them:

Busy Waiting	Split Binary Semaphores	Mesa Monitors
Use a lock and a loop	Use a collection of binary semaphores	Use a lock and a collection of condition variables and a loop
Easy to write the code	Just follow the recipe	Notifying is tricky
Easy to understand the code	Tricky to understand if you don't know recipe	Easy to understand the code
Ok-ish for true multi-core, but bad for virtual threads	Best for virtual threading. Thread only runs when it can make progress	Good for both multi-core and virtual threading (but not optimal)

What the recruiter really wanted…

```
from synch import *

done, lock, cond = False, Lock(), Condition()

def T0():
    acquire(?lock)
    while not done:
        wait(?cond, ?lock)
    release(?lock)

def T1():
    acquire(?lock)
    done = True
    notify(?cond)
    release(?lock)

spawn T0()
spawn T1()
```

wait for *done*

notify other thread