# 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*
  - Thread needs lock to enter the critical section
  - Only one thread can get the section's lock

# Specification in the face of Concurrency and Overlap

Is the following a possible scenario?

1. customer X orders a burger
2. customer Y orders a burger (afterwards)
3. customer Y is served a burger
4. customer X is served a burger (afterwards)

# Specification in the face of Concurrency and Overlap

Is the following a possible scenario?

1. customer X orders a burger
2. customer Y orders a burger (afterwards)
3. customer Y is served a burger
4. customer X is served a burger (afterwards)

We've all seen this happen. It's a matter of how things get scheduled!

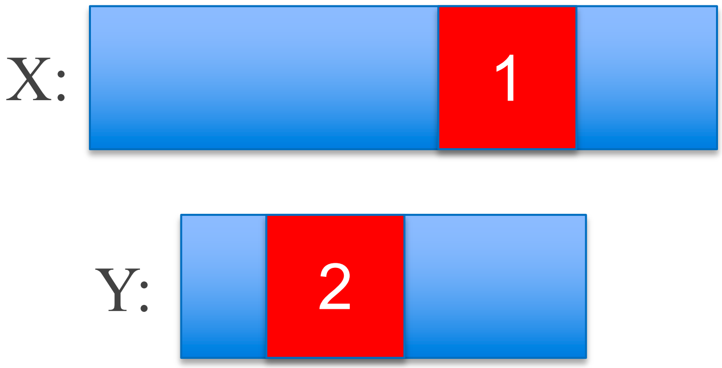
# Specification

- One operation: order a burger
  - result: a burger (at some later time)
- Semantics: the burger manifests itself atomically *sometime during the operation*
- *Atomically*: no two manifestations overlap
- It’s easier to specify something when you don’t have to worry about overlap
  - i.e., you can simply give a sequential specification
- Allows many implementations

# Implementation?

- Suppose the diner has one small hot plate and two cooks
- Cooks use a lock for access to the hot plate
- Possible scenario:

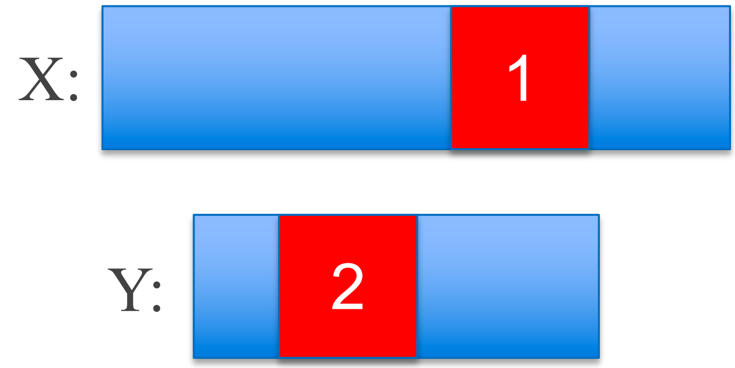
1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases lock
9. customer Y receives burger
10. customer X receives burger



# Implementation?

- Suppose the diner has one small hot plate and two cooks
- Cooks use a lock for access to the hot plate
- Possible scenario:

1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases lock
9. customer Y receives burger
10. customer X receives burger




- *can't happen if Y orders burger after X receives burger*
- *but if operations overlap, any ordering can happen…*

# Queue test program, again

```
import queue

const NOPS = 4
q = queue.Queue()

def put_test(self):
    print("call put", self)
    queue.put(?q, self)
    print("done put", self)

def get_test(self):
    print("call get", self)
    let v = queue.get(?q):
        print("done get", self, v)

nputs = choose {1..NOPS−1}
for i in {1..nputs}:
    spawn put_test(i)
for i in {1..NOPS−nputs}:
    spawn get_test(i)
```

# 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 Specification

```
def RWlock():
    result = { .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    atomically when rw→nwriters == 0:
        rw→nreaders += 1

def read_release(rw):
    atomically rw→nreaders −= 1

def write_acquire(rw):
    atomically when (rw→nreaders + rw→nwriters) == 0:
        rw→nwriters = 1

def write_release(rw):
    atomically rw→nwriters = 0
```

# R/W Locks: test for mutual exclusion

```
import RW

const NOPS = 3

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ "read", "write" }) == "read":
            RW.read_acquire(?rw)
            rcs: assert (countLabel(rcs) >= 1) and (countLabel(wcs) == 0)
            RW.read_release(?rw)
        else: # write
            RW.write_acquire(?rw)
            wcs: assert (countLabel(rcs) == 0) and (countLabel(wcs) == 1)
            RW.write_release(?rw)

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

no writer

1 writer and
no readers

# *Cheating* R/W lock implementation

```
import synch

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

def read_acquire(rw):
    synch.acquire(rw);

def read_release(rw):
    synch.release(rw);

def write_acquire(rw):
    synch.acquire(rw);

def write_release(rw):
    synch.release(rw);
```

# *Cheating* R/W lock implementation

```
import synch

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

def read_acquire(rw):
    synch.acquire(rw);

def read_release(rw):
    synch.release(rw);

def write_acquire(rw):
    synch.acquire(rw);

def write_release(rw):
    synch.release(rw);
```

Allows only one reader to get the lock at a time

Does *not* have the same behavior as the specification

- it is missing behaviors
- no bad behaviors though

# *Busy Waiting* Implementation

```
from synch import Lock, acquire, release

def RWlock():
    result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders −= 1
    release(?rw→lock)

def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

def write_release(rw):
    acquire(?rw→lock)
    rw→nwriters = 0
    release(?rw→lock)
```

# *Busy Waiting* Implementation

```
from synch import Lock, acquire, release

def RWlock():
    result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders −= 1
    release(?rw→lock)

def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

def write_release(rw):
    acquire(?rw→lock)
    rw→nwriters = 0
    release(?rw→lock)
```

Good: has the same behaviors as the implemention

Bad: process is continuously scheduled to try to get the lock even if it's not available

(*Harmony complains about this as well*)

# Conditional Waiting

# Conditional Waiting

- Thus 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:

> *done* = **False**
> **while not** *done*:
>     *next* = queue.get(*q*)
>     *done* = *next* != **None**

# Busy Waiting: not a good way

- Wait until queue is non-empty:

```
done = False
while not done:
    next = queue.get(q)
    done = next != None
```

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

# Enter *binary semaphores*

[Dijkstra 1962]

# Binary Semaphore

- Boolean variable (much like a lock)
- Three operations:
  - *binsema* = BinSema(False or True)
    - initialize *binsema*
  - acquire(?*binsema*)
    - waits until !*binsema* = False, then sets !*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*, usually *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 specification

```
def BinSema(acquired):
    result = acquired

def Lock():
    result = BinSema(False)

def acquired(binsema):
    result = !binsema

def acquire(binsema):
    atomically when not !binsema:
        !binsema = True

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

# Waiting with semaphores

```
import synch;

condition = BinSema(True)

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

spawn T0()
spawn T1()
```

# Waiting 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]

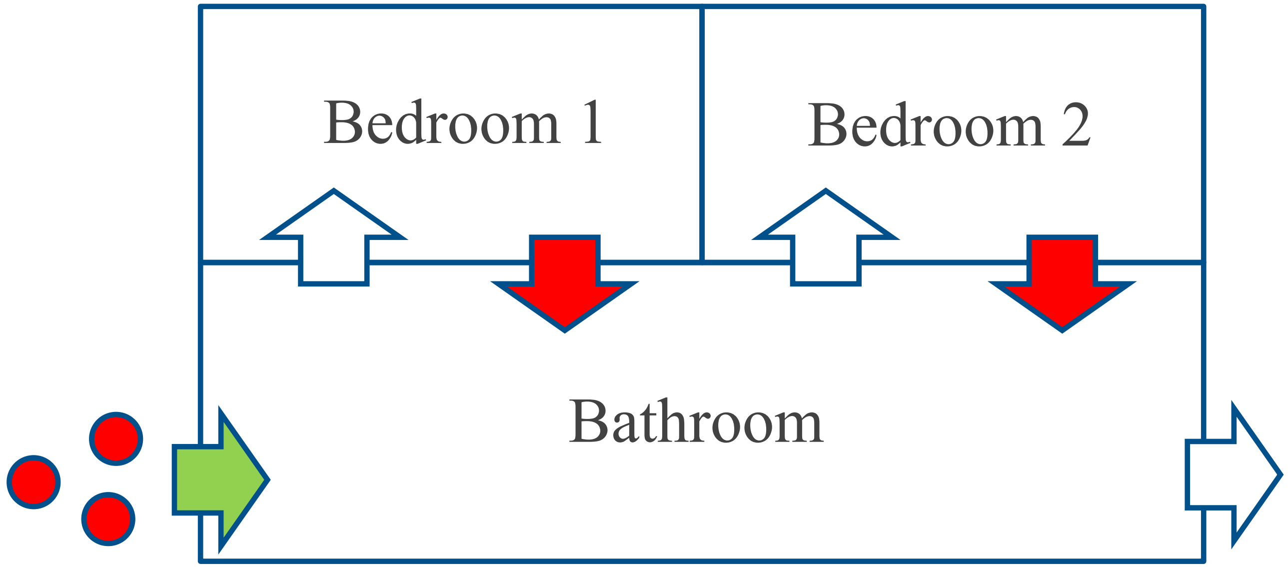
- Implement baton passing with multiple binary semaphores
- If there are *N* conditions, you'll need *N*+1 binary semaphores
  - one for each condition
  - one to enter the critical section in the first place
- Invariant: At most one of these semaphores is released (False)
  - If all are acquired (True), 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…

holds baton

does not hold baton

3 threads want to enter critical section



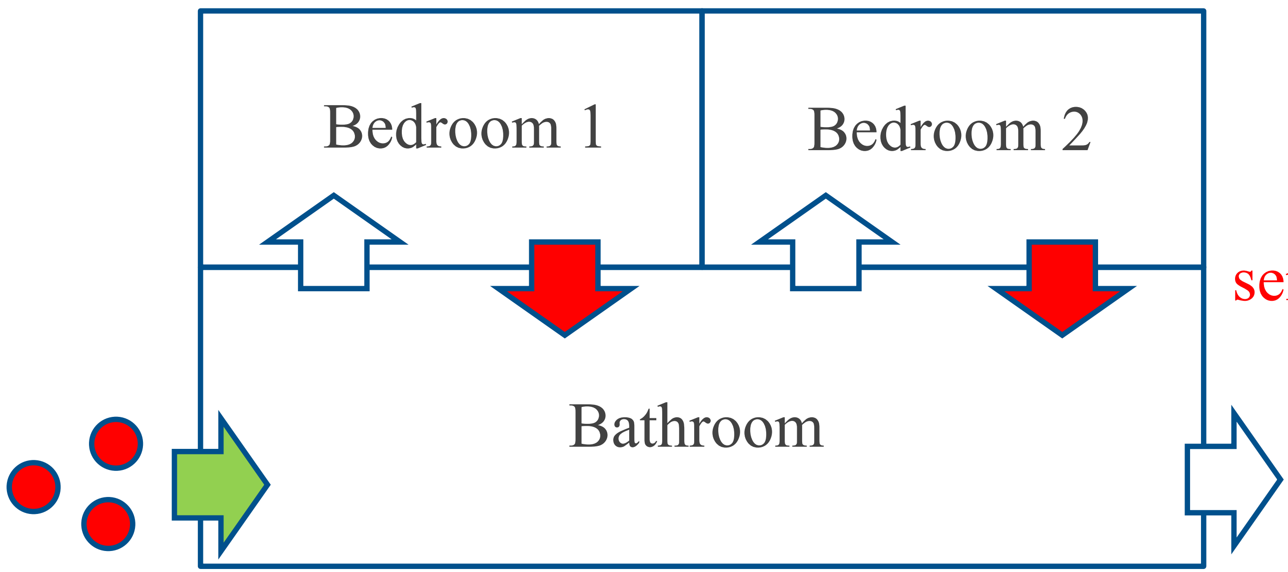

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green

# Bathroom humor…

- holds baton (green)
- does not hold baton (red)

3 threads want to enter critical section




sema = True

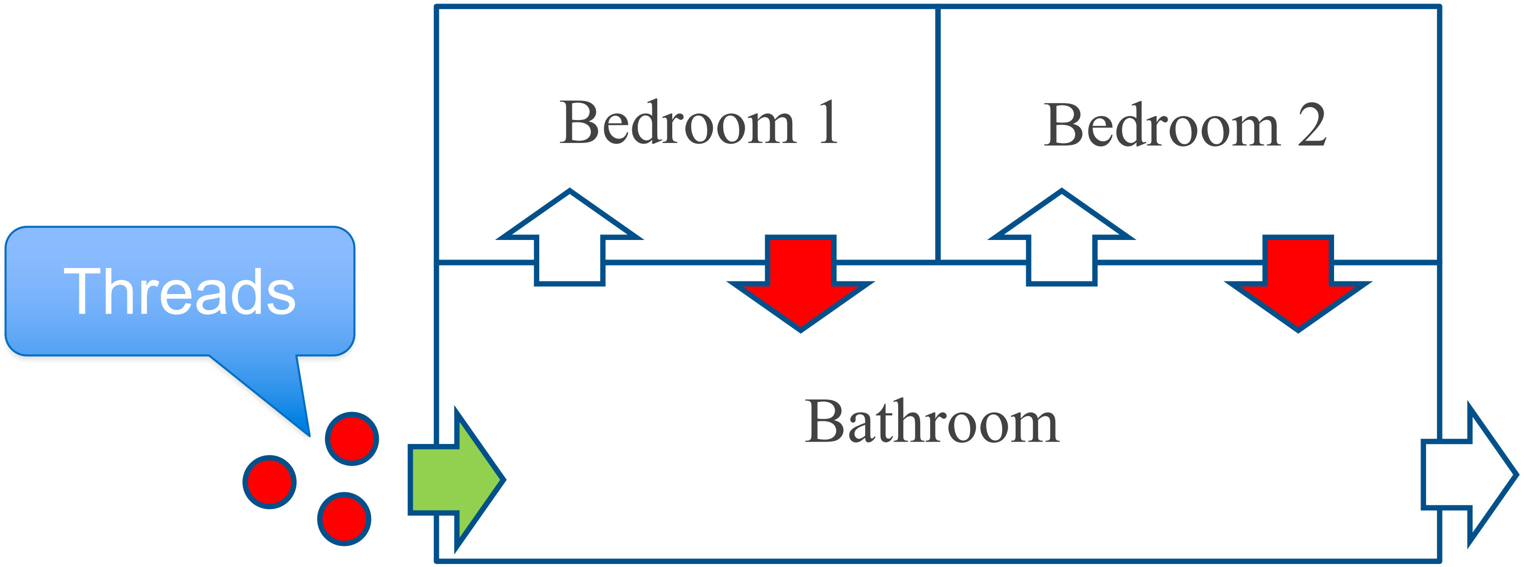
semaphore = False

Bathroom: critical section
Bedrooms: waiting conditions

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

# Bathroom humor…


3 threads want to enter critical section

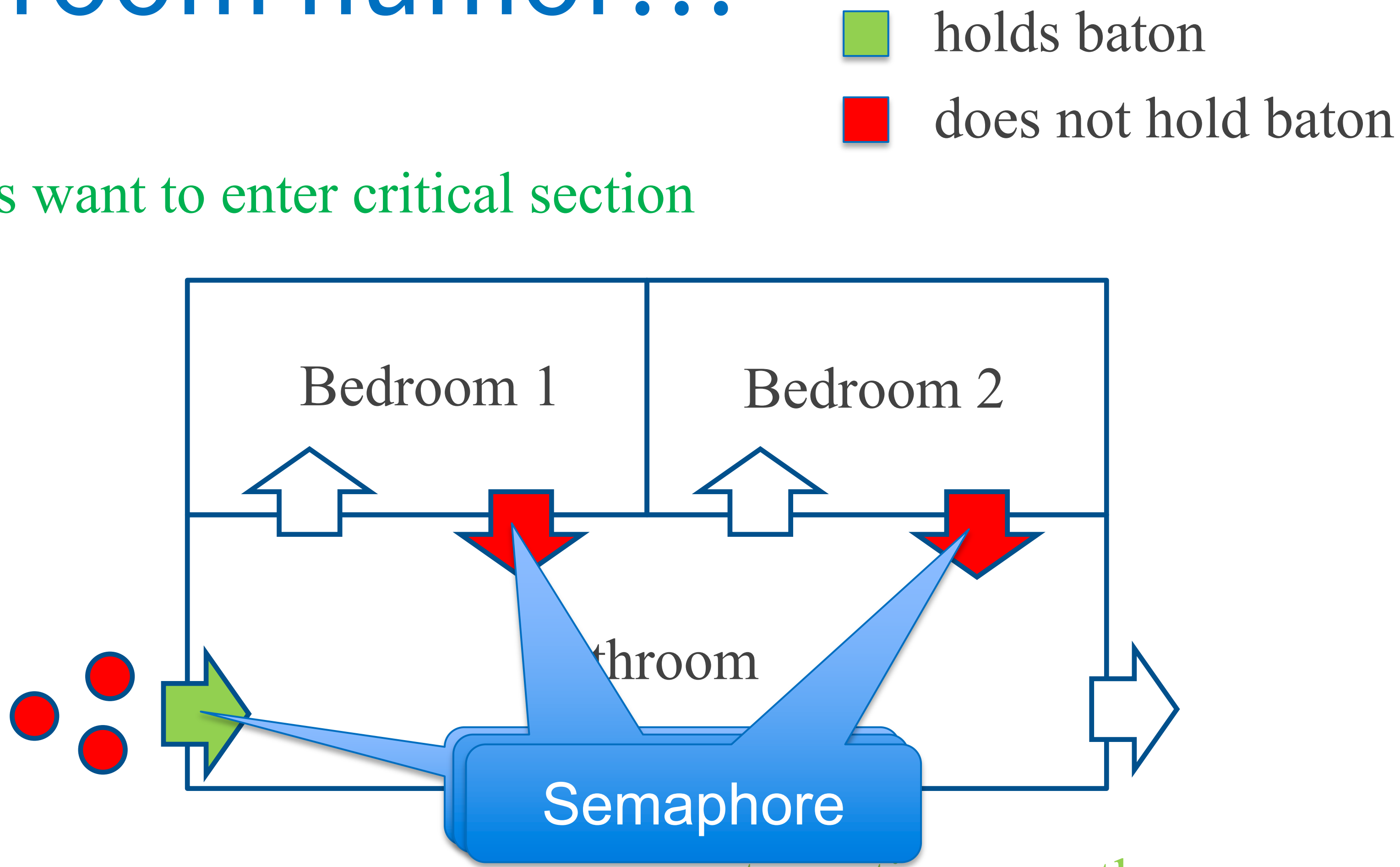



Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green

# Bathroom humor…

3 threads want to enter critical section




at any time exactly one semaphore or thread is green

Bathroom: critical section
Bedrooms: waiting conditions

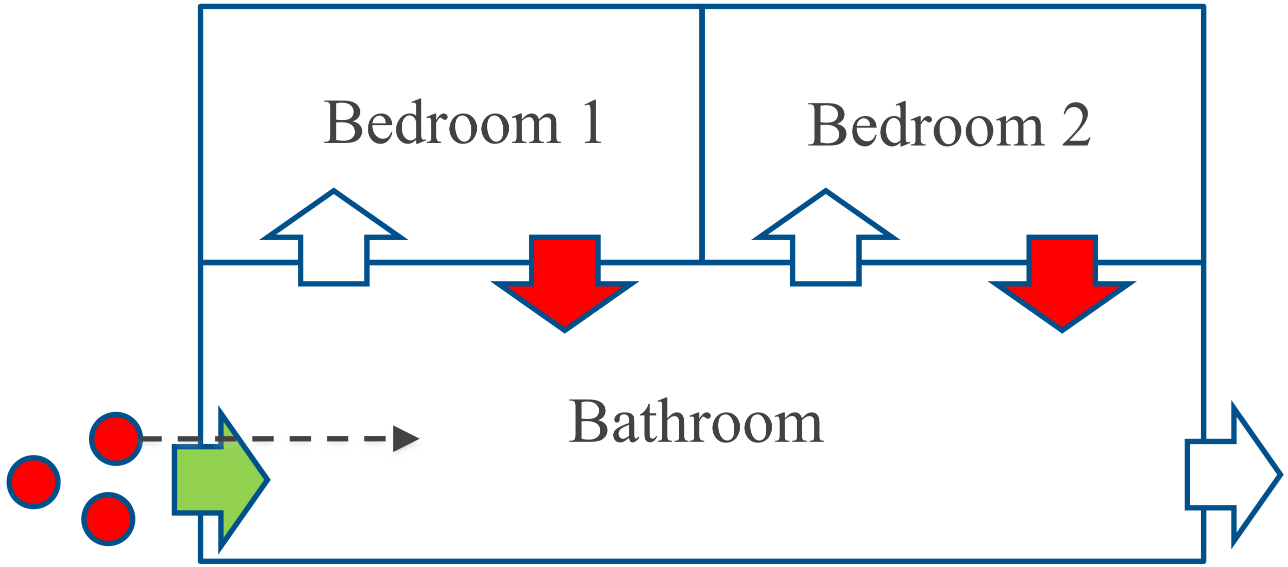
# This is a model of:

- 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




at any time exactly one semaphore or thread is green

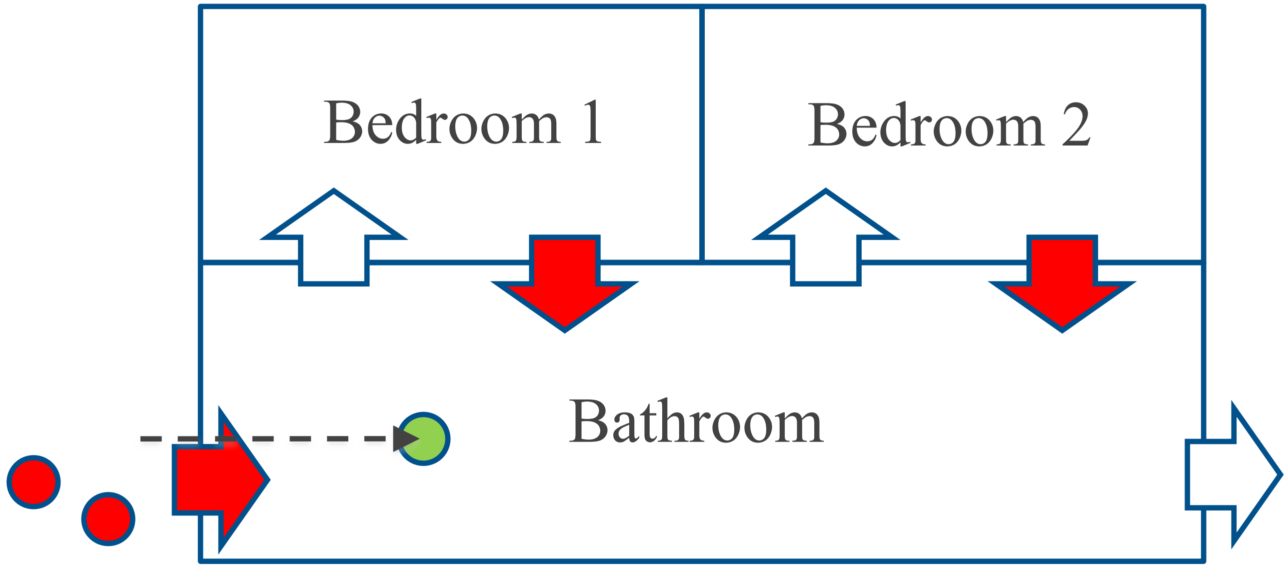
Bathroom: critical section
Bedrooms: waiting conditions

# Bathroom humor...

holds baton

does not hold baton

1 thread entered the critical section




at any time exactly one semaphore or thread is green

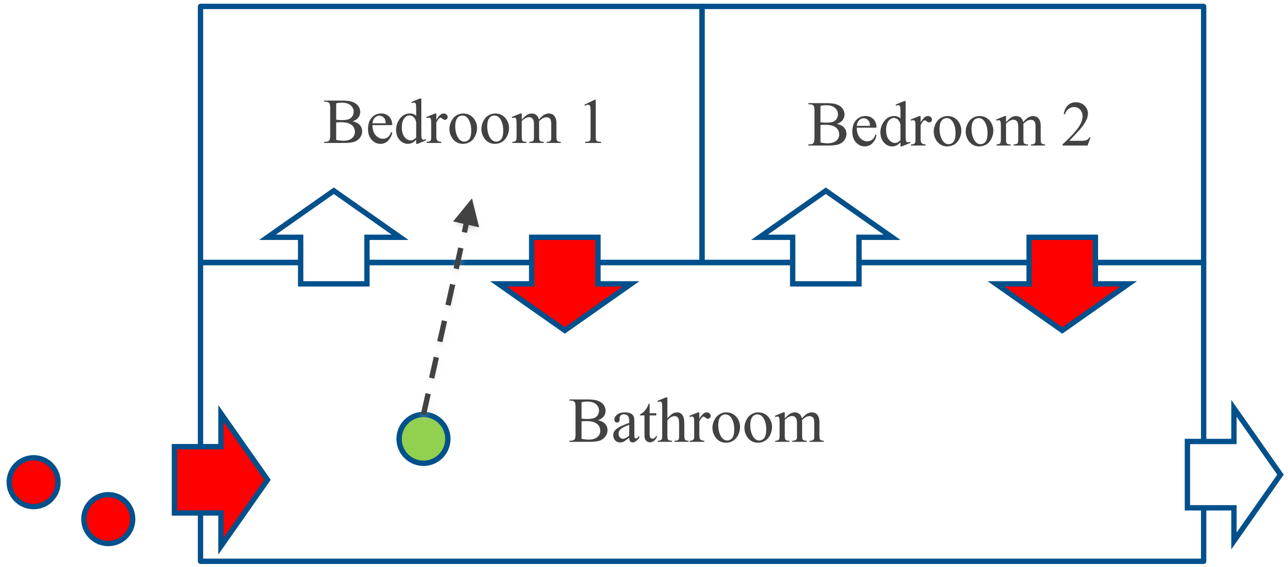
Bathroom: critical section
Bedrooms: waiting conditions

# Bathroom humor…

holds baton

does not hold baton

thread needs to wait for Condition 1




at any time exactly one semaphore or thread is green

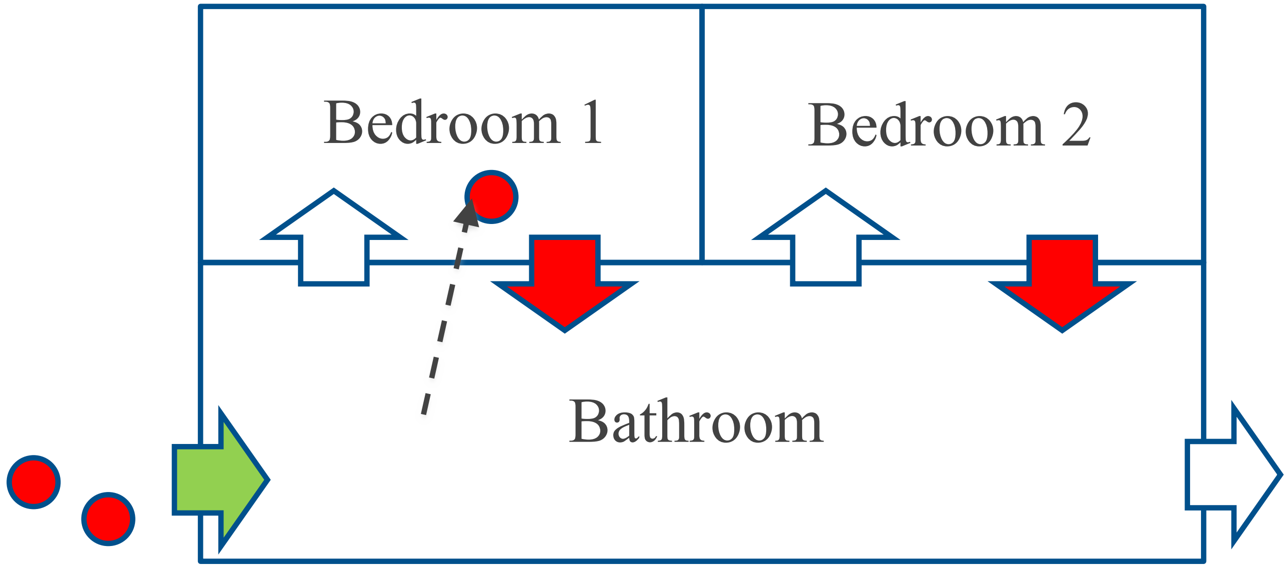
Bathroom: critical section
Bedrooms: waiting conditions

# Bathroom humor…

holds baton

does not hold baton

no thread waiting for condition that holds

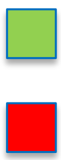



at any time exactly one semaphore or thread is green

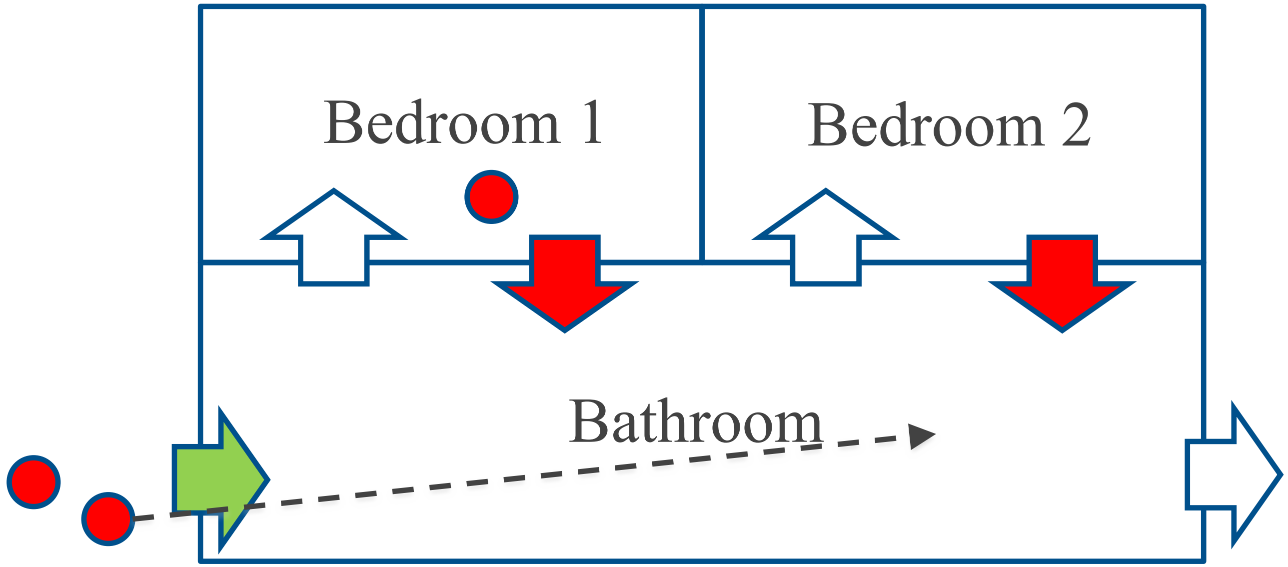
Bathroom: critical section

Bedrooms: waiting conditions

# Bathroom humor…




another thread can enter the critical section




Bathroom: critical section
Bedrooms: waiting conditions

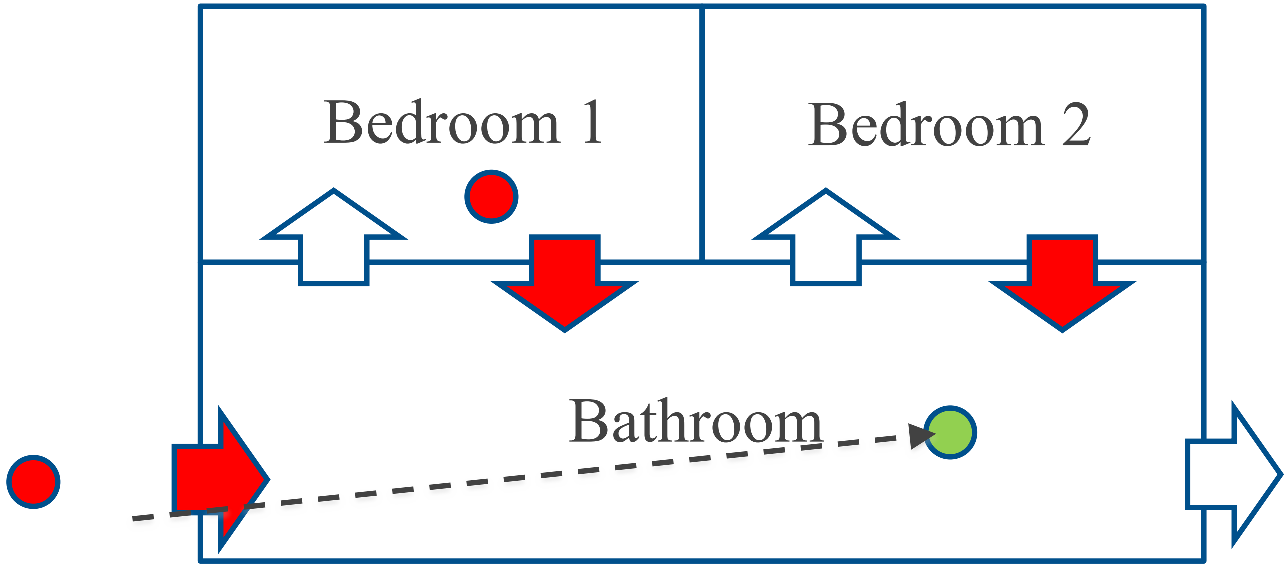
at any time exactly one semaphore or thread is green

# Bathroom humor…

holds baton

does not hold baton

thread entered the critical section




Bathroom: critical section
Bedrooms: waiting conditions

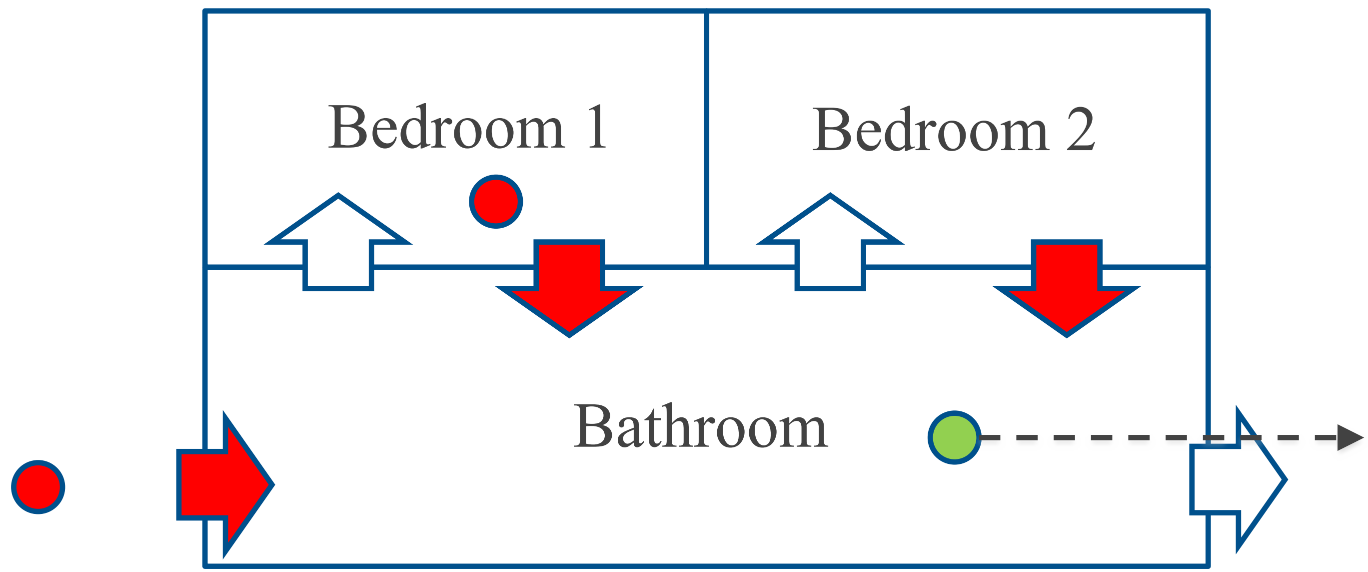
at any time exactly one semaphore or thread is green

# Bathroom humor…

holds baton

does not hold baton

thread enables Condition 1 and wants to leave



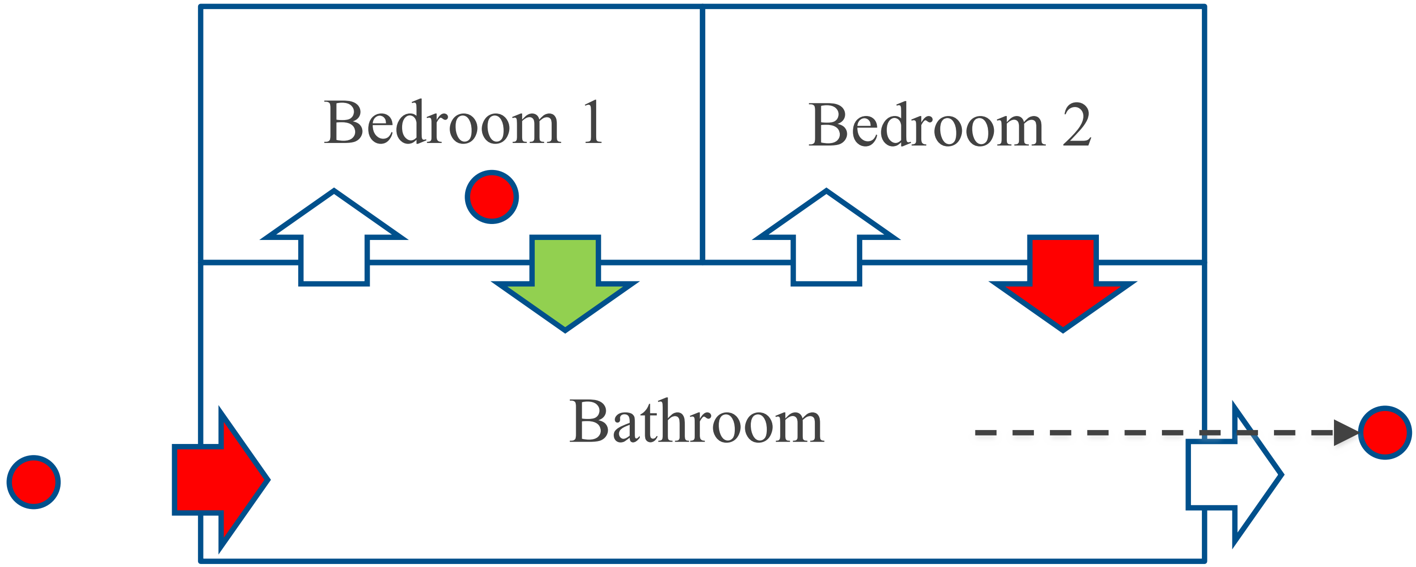

Bathroom: critical section
Bedrooms: waiting conditions

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

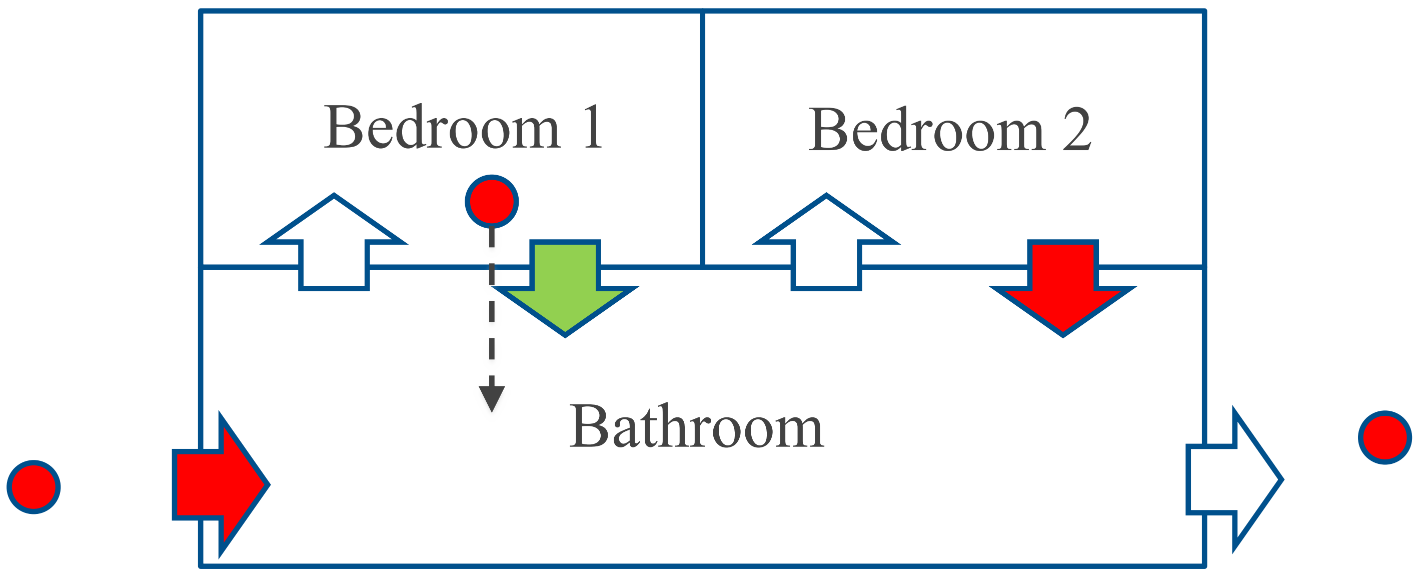
at any time exactly one semaphore or thread is green

# 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

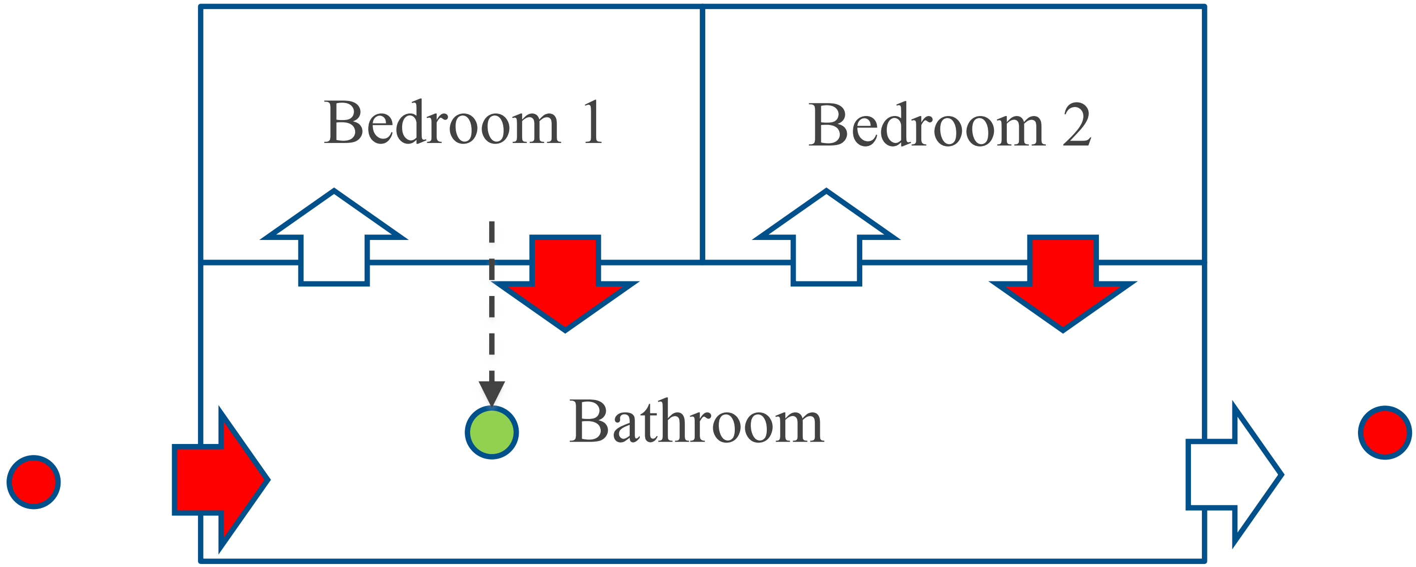
at any time exactly one semaphore or thread is green

# Bathroom humor…

holds baton

does not hold baton

first thread entered critical section again



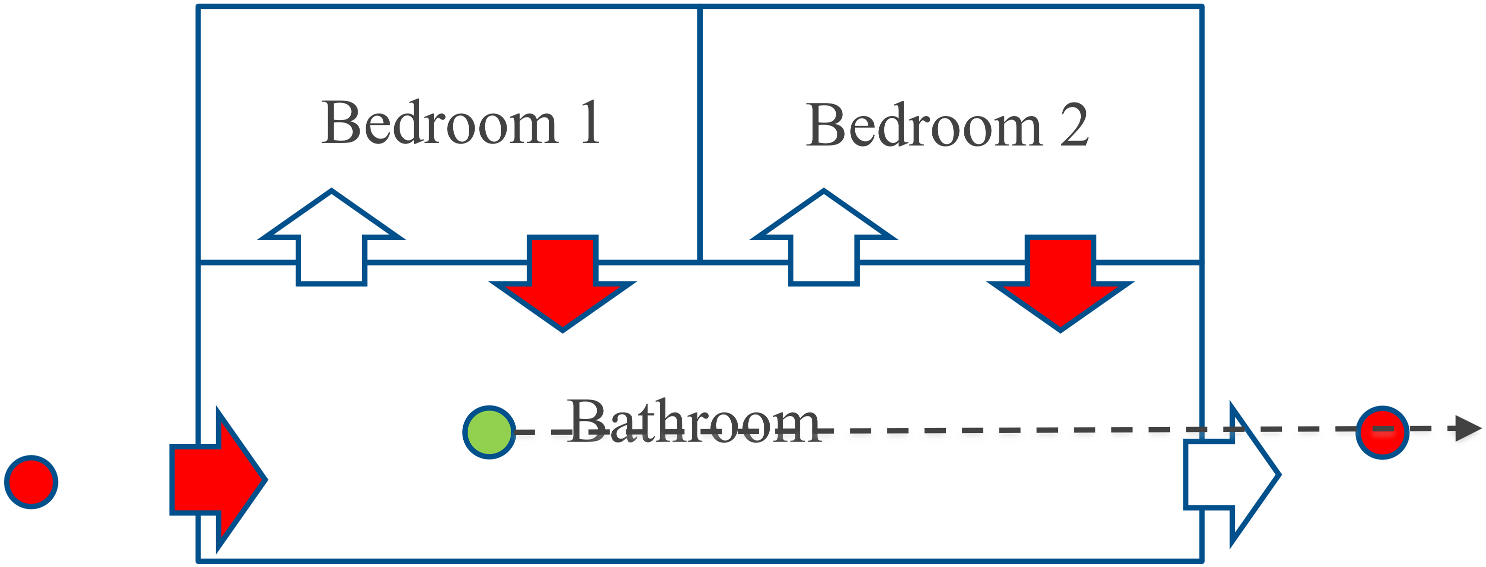

Bathroom: critical section
Bedrooms: waiting conditions

at any time exactly one semaphore or thread is green

# Bathroom humor…

- holds baton (green)
- does not hold baton (red)

first thread leaves




at any time exactly one semaphore or thread is green

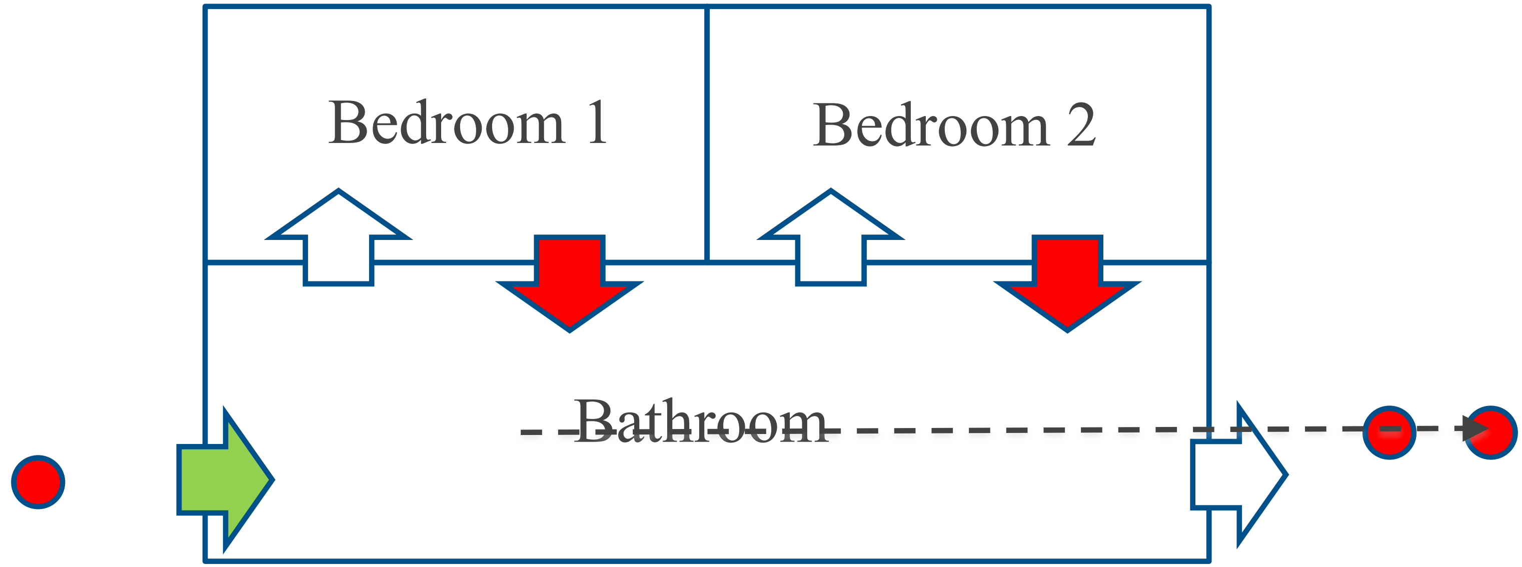
Bathroom: critical section
Bedrooms: waiting conditions

# Bathroom humor…

holds baton

does not hold baton

first thread done




at any time exactly one semaphore or thread is green

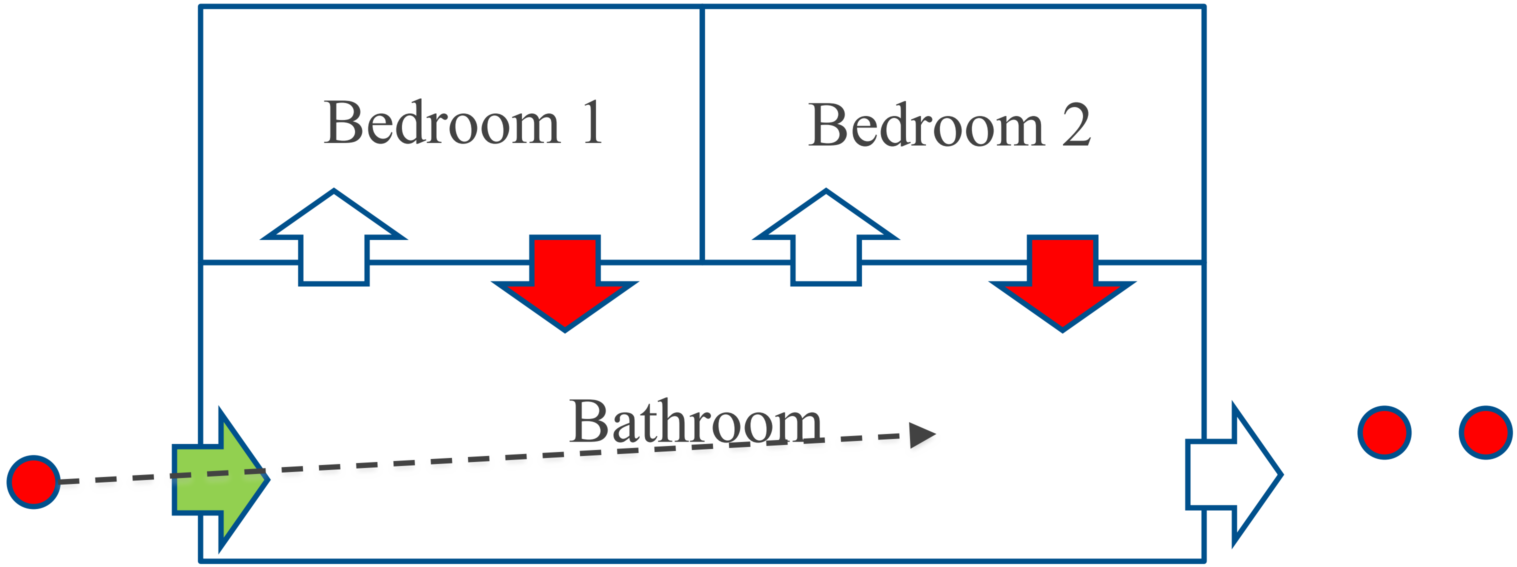
Bathroom: critical section
Bedrooms: waiting conditions

# Bathroom humor...

holds baton

does not hold baton

one thread wants to enter the critical section




Bathroom: critical section
Bedrooms: waiting conditions

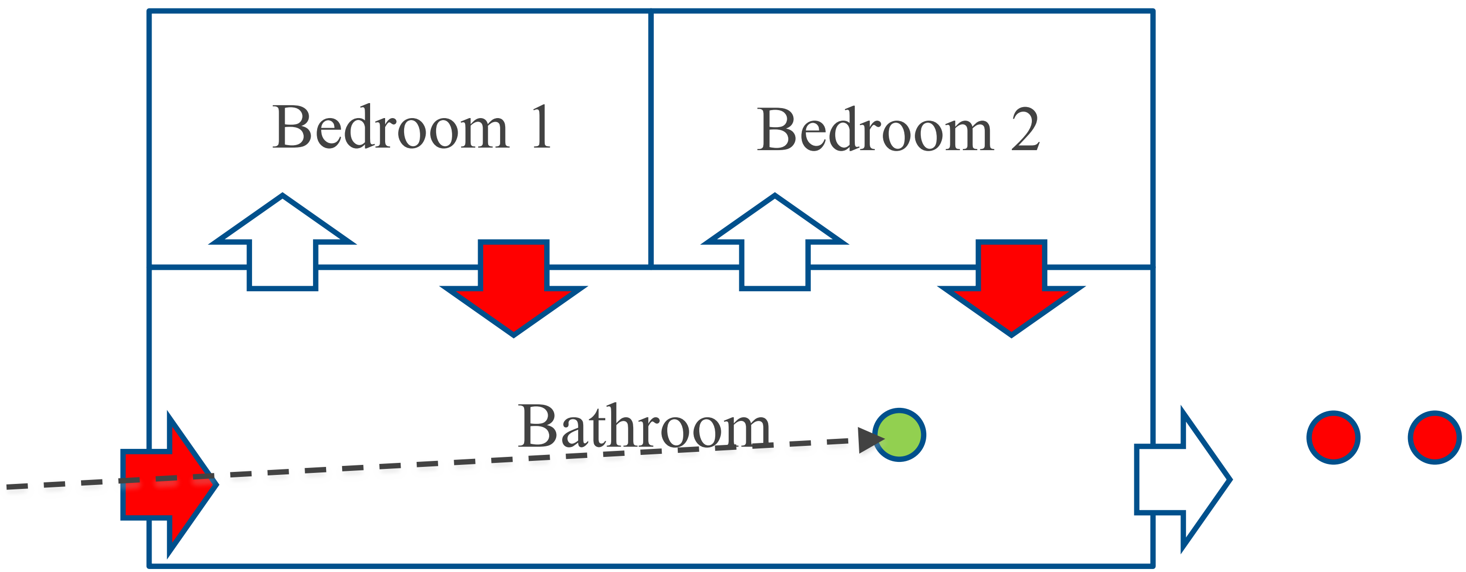
at any time exactly one semaphore or thread is green

# Bathroom humor…

holds baton

does not hold baton

last thread entered critical section




at any time exactly one semaphore or thread is green

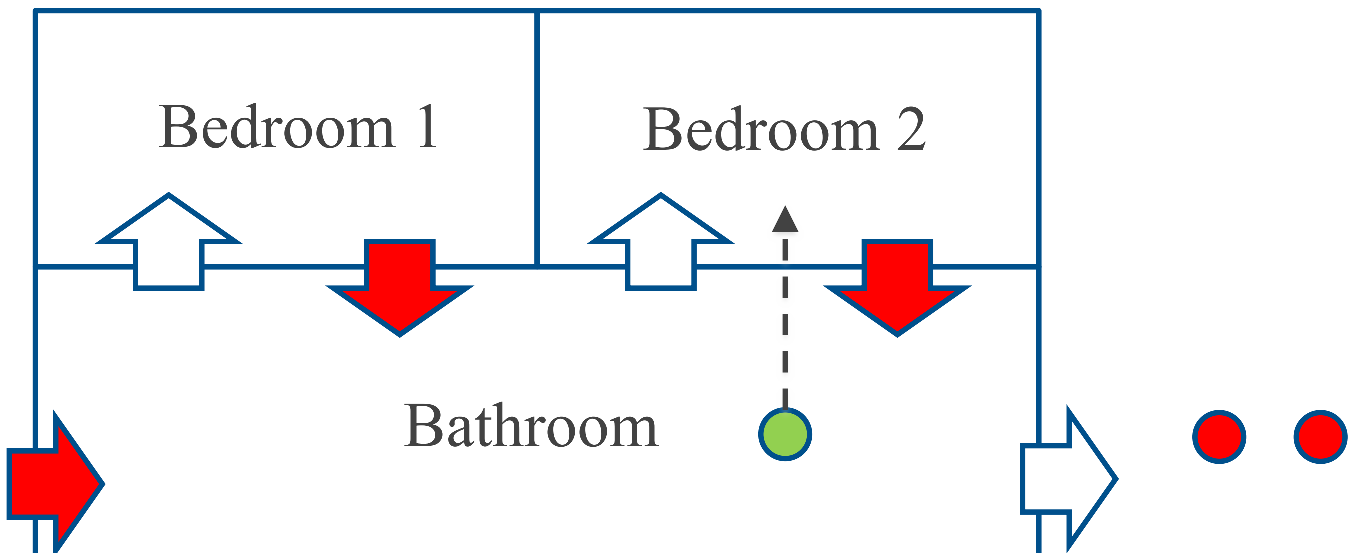
Bathroom: critical section
Bedrooms: waiting conditions

# Bathroom humor…

holds baton

does not hold baton

thread needs to wait for Condition 2




Bathroom: critical section
Bedrooms: waiting conditions

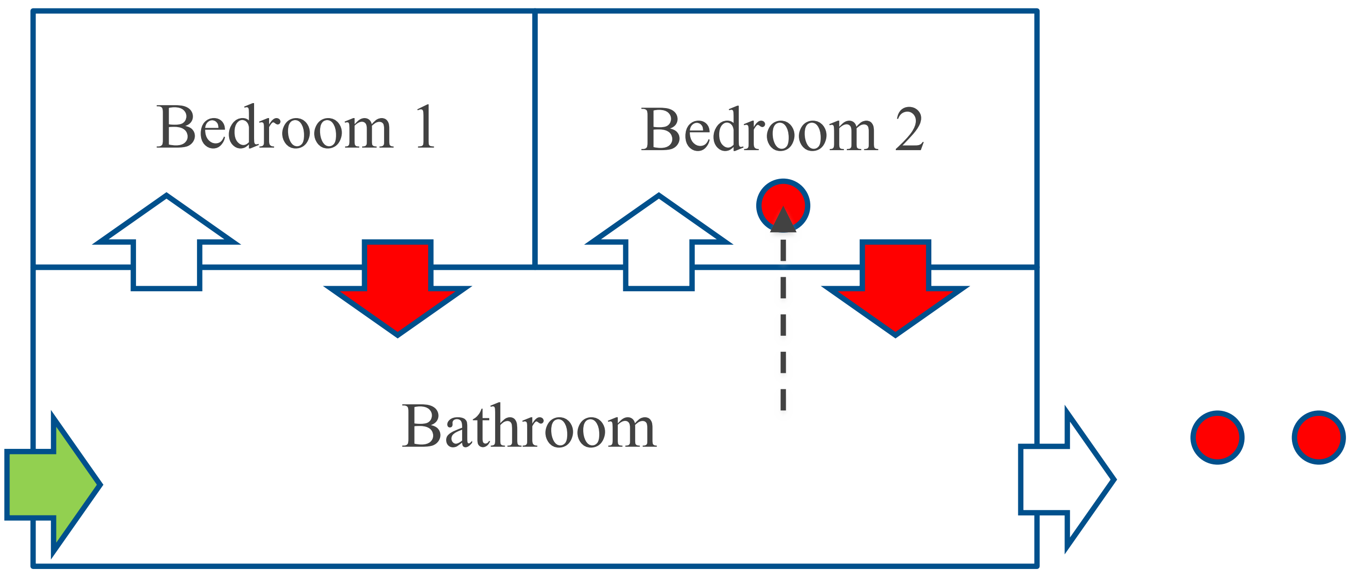
at any time exactly one semaphore or thread is green

# 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 Spec, again

```
def RWlock():
    result = { .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    atomically when rw→nwriters == 0:
        rw→nreaders += 1

def read_release(rw):
    atomically rw→nreaders −= 1

def write_acquire(rw):
    atomically when (rw→nreaders + rw→nwriters) == 0:
        rw→nwriters = 1

def write_release(rw):
    atomically rw→nwriters = 0
```

# 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 $nwriters \geq n$
- $(nreaders \geq 0 \wedge nwriters = 0) \vee (nreaders = 0 \wedge 0 \leq nwriters \leq 1)$

# 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)
```

# 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?

# What happens if…

- Multiple readers are waiting and a writer leaves
- Does it let in all the readers or just one?

# 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 sections:
  - 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)

# 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;          ⇐ “condition variable”
   procedure acquire;
      begin if busy then nonbusy.wait;   ⇐ wait method
               busy := true
      end;
   procedure release;
      begin busy := false;
               nonbusy.signal          ⇐ signal method
      end;
   busy := false; comment initial value;
end single resource
```

# 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()

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

waiting gate

# 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

waiting gate

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

# 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():
    result = bag.empty()

def wait(c, lk):
    var cnt = 0
    let _, ctx = save():
        atomically:
            cnt = bag.multiplicity(!c, ctx)
            !c = bag.add(!c, ctx)
            !lk = False
        atomically when (not !lk) and (bag.multiplicity(!c, ctx) <= cnt):
            !lk = True

def notify(c):
    atomically if !c != bag.empty():
        !c = 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

# Reader/Writer Lock Specification

```
def RWlock():
    result = { .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    atomically when rw→nwriters == 0:
        rw→nreaders += 1

def read_release(rw):
    atomically rw→nreaders −= 1

def write_acquire(rw):
    atomically when (rw→nreaders + rw→nwriters) == 0:
        rw→nwriters = 1

def write_release(rw):
    atomically rw→nwriters = 0
```

# *Busy Waiting* Implementation

```
from synch import Lock, acquire, release

def RWlock():
    result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders −= 1
    release(?rw→lock)

def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

def write_release(rw):
    acquire(?rw→lock)
    rw→nwriters = 0
    release(?rw→lock)
```

# 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 | Good for virtual threading. Thread only runs when it can make progress | Good for both multi-core and virtual threading (but not optimal) |