CSE 451: Operating Systems

Lab Section: Week 4

Today

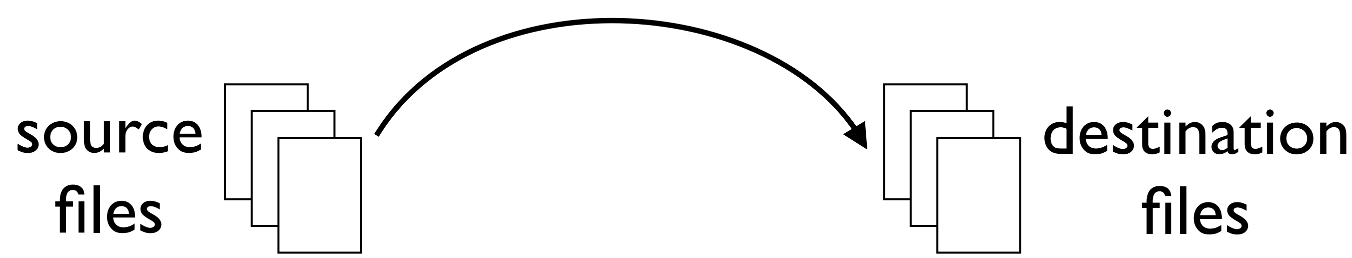
- Project 3
- Synchronization
  (this may be useful for tomorrow's quiz☺)

Reminder ...

- Project 1 due last night!
  - dropbox is closed
- Project 2 due last night!
  - but you can submit late
  - late penalty is 0.5 grade points per day (I think ...)

Project 3

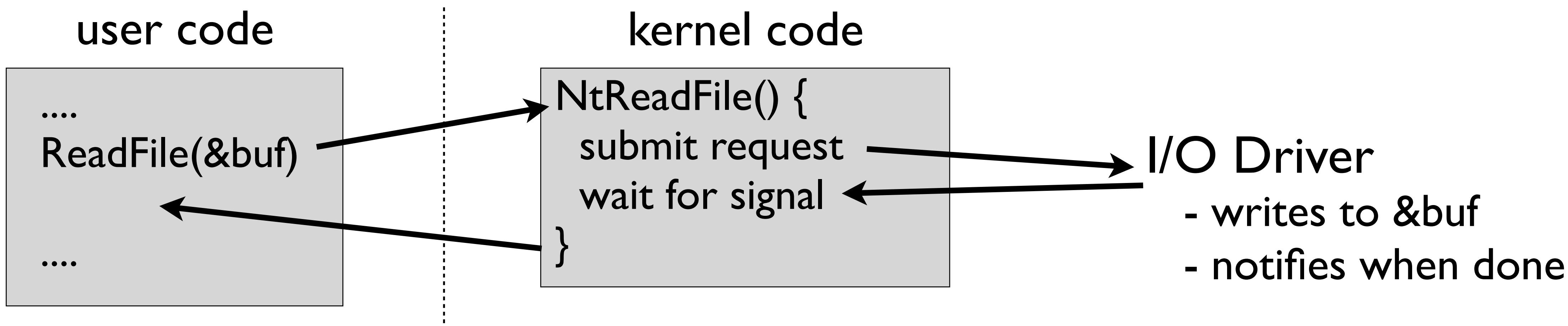
- File copy program
  - implement entirely in user-space (no kernel hacking ☹)



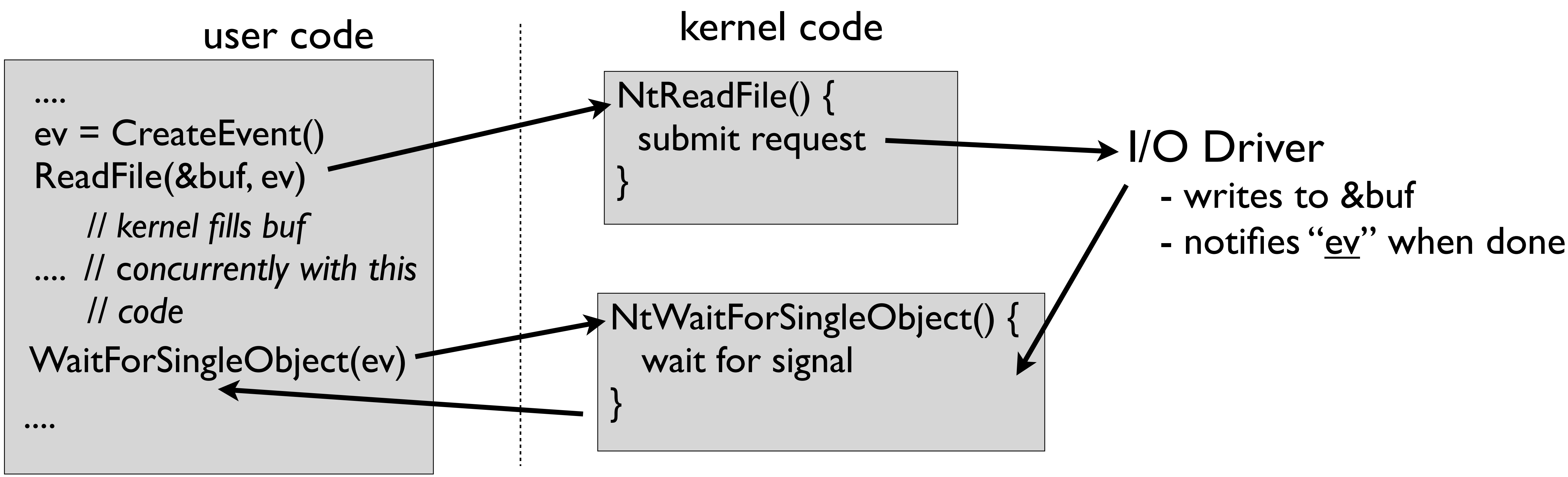
- Three parts
  - implement using multithreading + synchronous I/O
  - implement using single-threading + asynchronous I/O
  - analyze the performance of both implementations
    (more on this next week)

I/O in Windows

- Synchronous



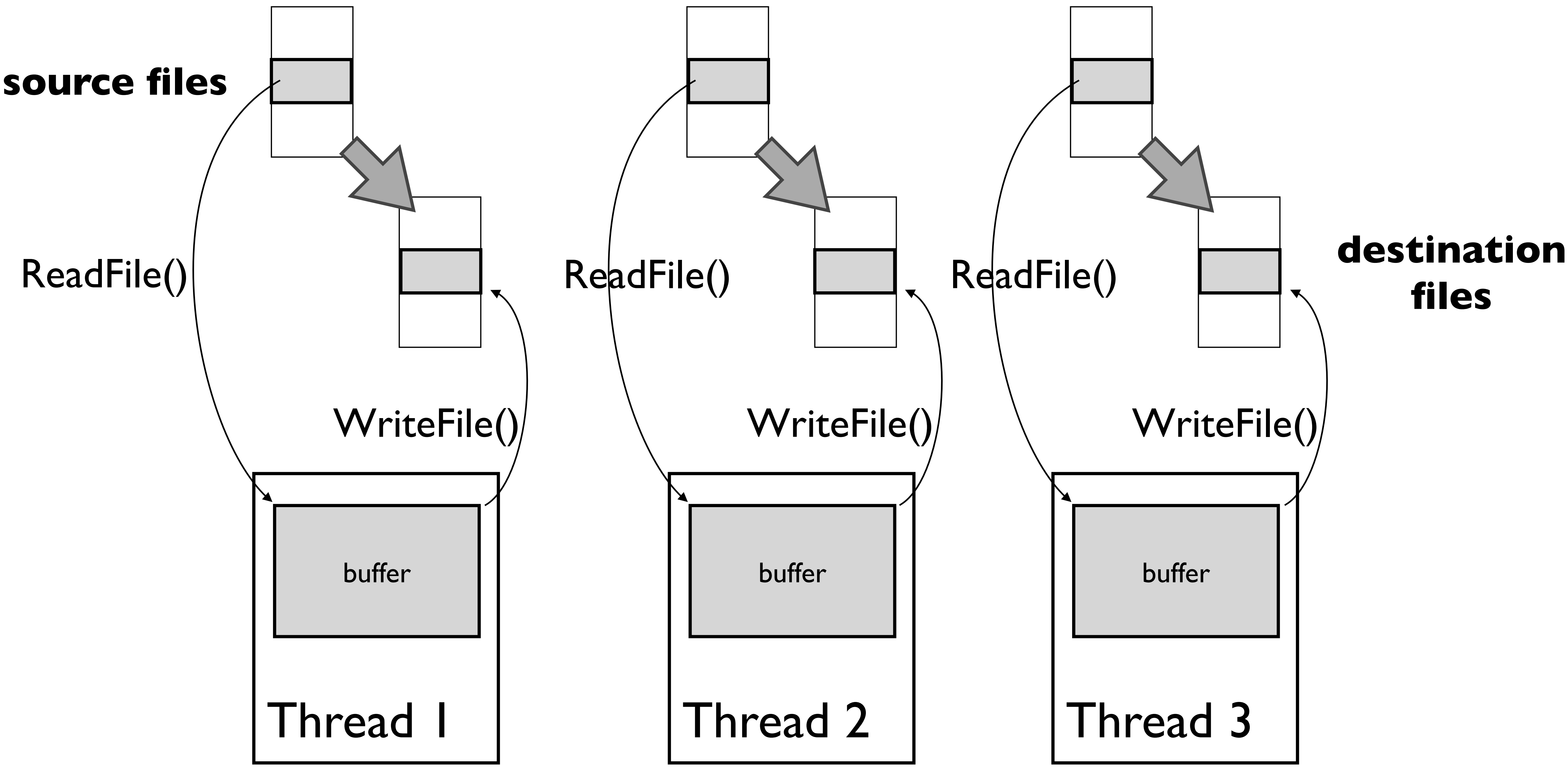
- Asynchronous



I/O in Windows

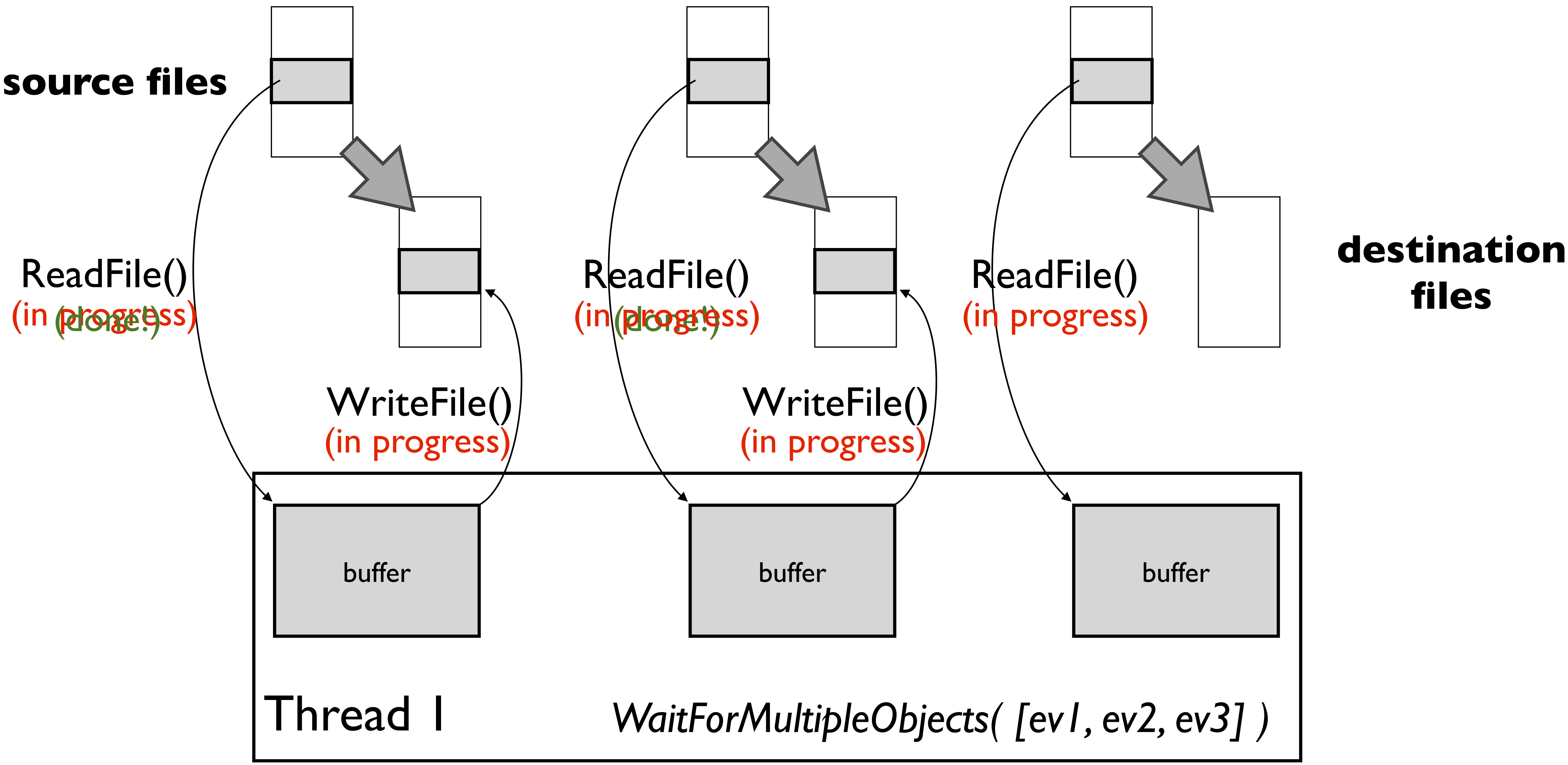
- Advantages of sync I/O?
  - easier to program
    (don't have to explicitly synchronize with I/O driver)
- Advantages of async I/O?
  - more efficient, potentially
    (you can "overlap" work with the I/O request)
- How do we make sync I/O go faster?
  - use more threads!

Project 3
Multithreaded + Synchronous I/O



Project 3

Single-threaded + Asynchronous I/O



Today

- ~~Project 3~~
- Synchronization
  (this may be useful for tomorrow's quiz☺)

Why do we need synchronization?

- Safe data sharing
  - bank account example:

```
withdraw(account, value) {
    balance = get_balance(account)
    balance -= value
```

```
withdraw(account, value) {
    balance = get_balance(account)
    balance -= value
    put_balance(account, balance)
}
```

oops!

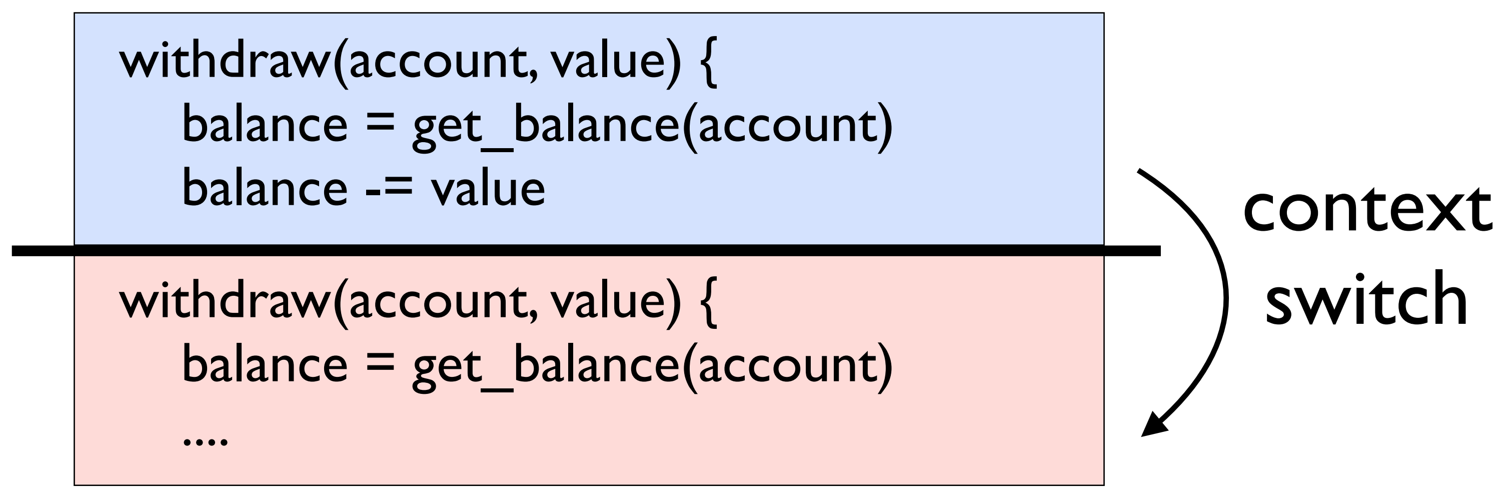
```
    put_balance(account, balance)
}
```

cpu1

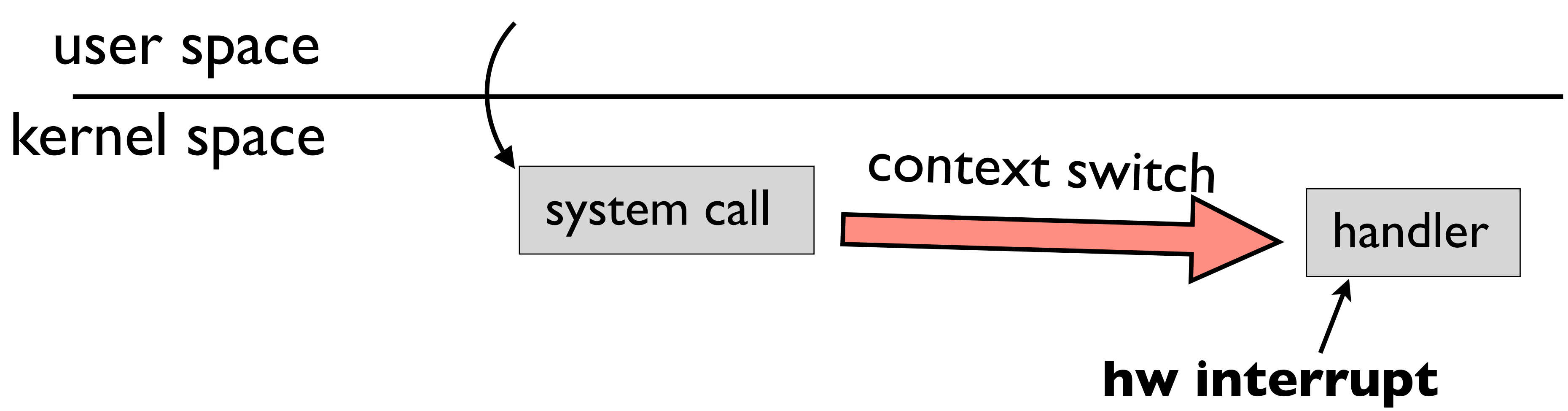
cpu2

Why do we need synchronization on single-processor computers?

- Preemption



- Interrupts



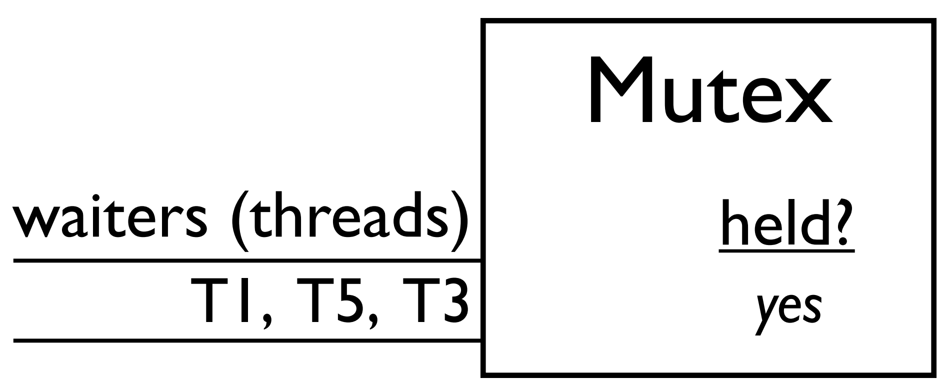
Type of synchronization

(we'll talk about these today)

- Locks / Mutexes
- Semaphores
- Condition variables
- Monitors (= mutexes + condition variables)

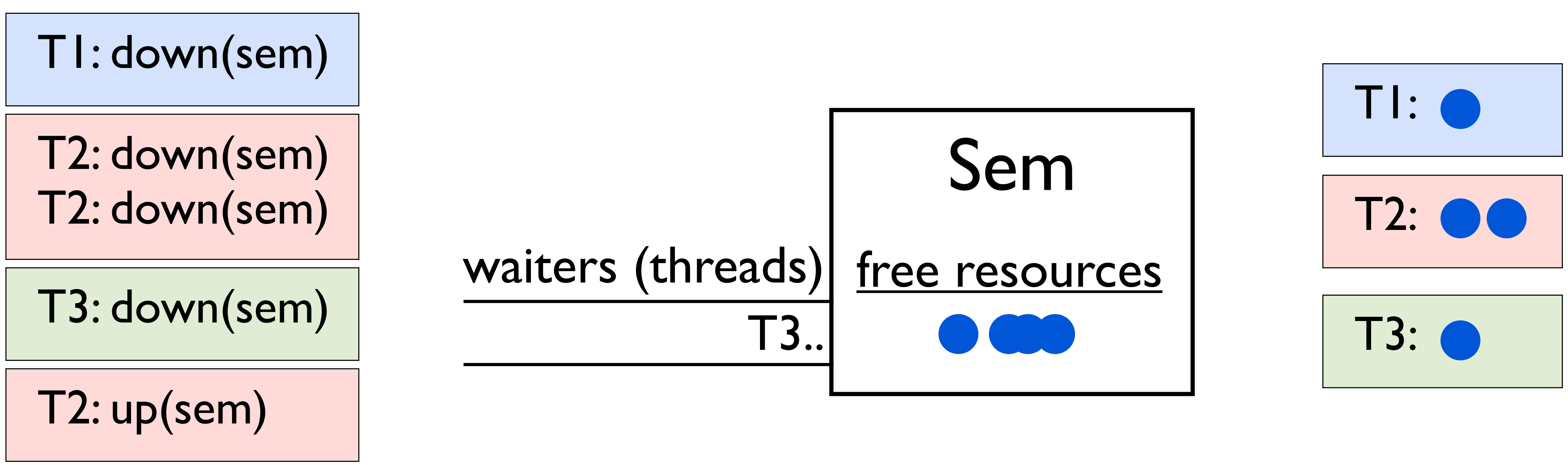
Locks / Mutexes

- MUTual EXclusion
  - lock / acquire
  - unlock / release
- Spinlocks
  - acquire: busy wait (spin) until the lock is released
- Blocking / queueing mutexes



Semaphores

- Operations
  - P (or more sanely: down / wait)
  - V (or more sanely: up / signal)



- Binary semaphore
  - initial count = 1
  - same as a mutex

Semaphores

- What are counting semaphores good for?
  - resource allocation!

- Example: memory allocation w/ quotas
  - sem.count initialized to the memory quota in bytes
  - on malloc(*n*): call down(sem, *n*)
  - on corresponding free(): call up(sem, *n)*

- Example: RPC windowing
  - want no more than *n* RPCs outstanding at any time
  - sem.count initialized to *n*

- Example: bounded buffer
  - see lecture slides ....

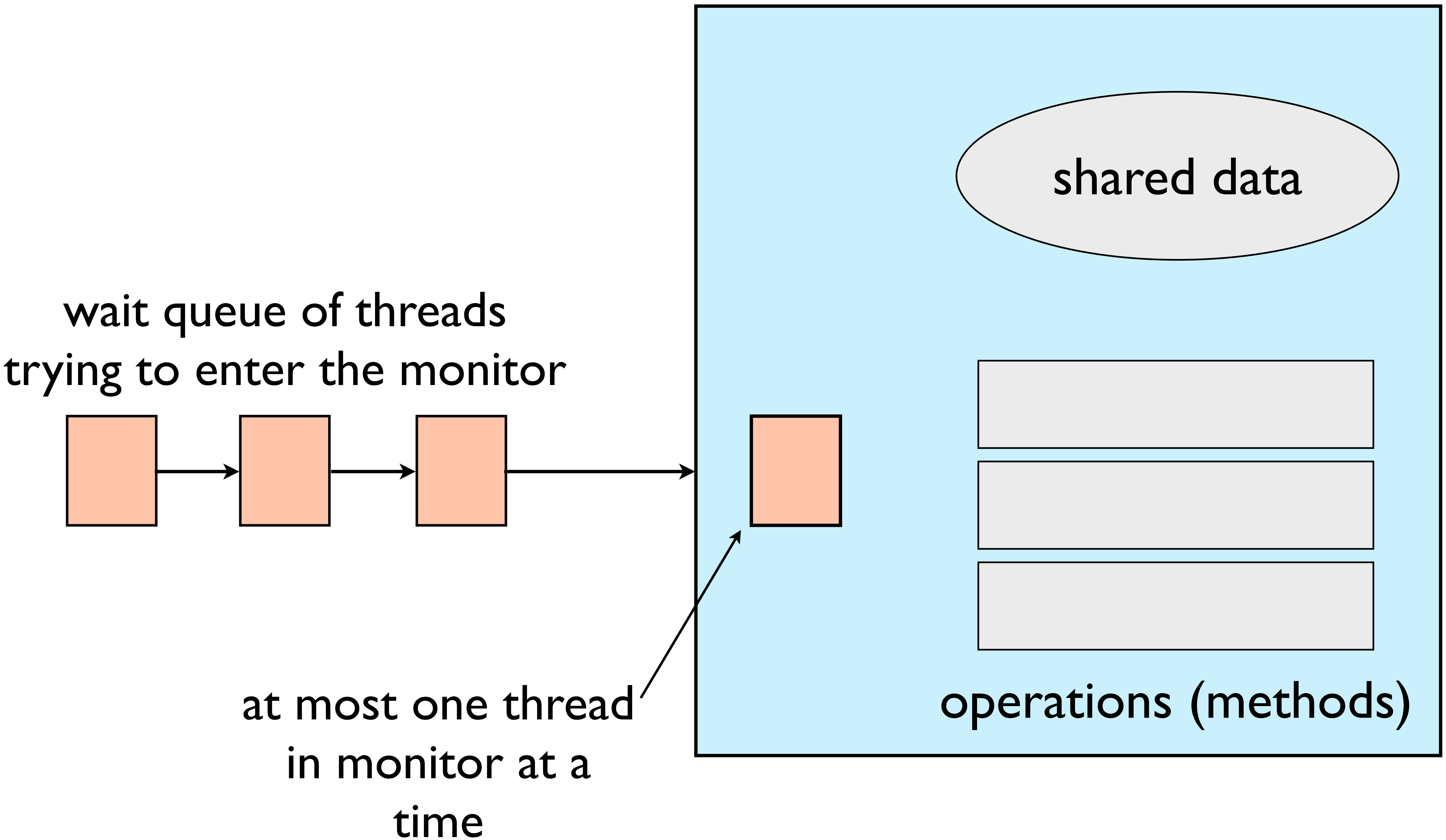
Problems with semaphores (and locks)

- No connection between lock and the data it guards
- Easy to:
  - forget to acquire a lock
  - forget to release a lock
  - use the wrong lock

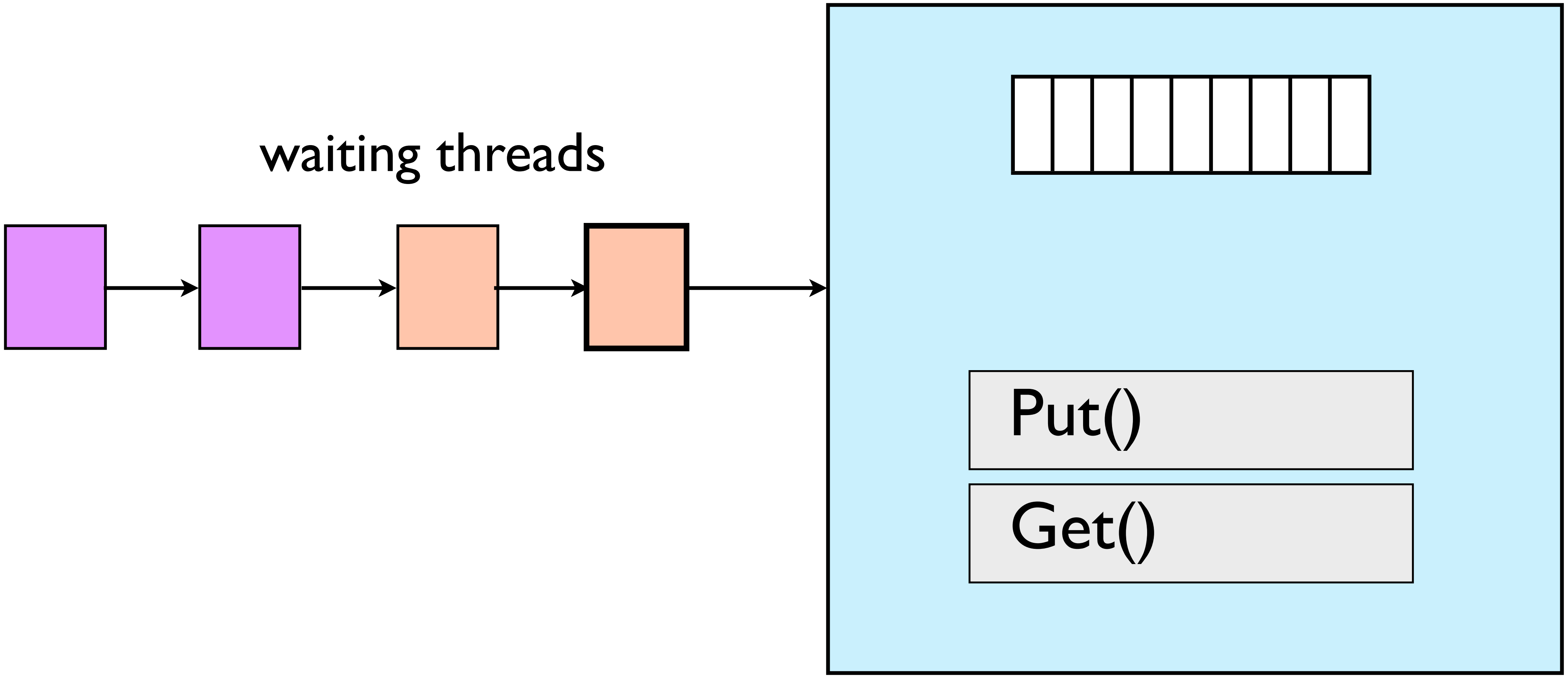
Monitors

- A <u>programming language</u> construct
  - synchronization code added by the compiler
- Essentially a class
  - shared private data
  - methods
  - automatic synchronization

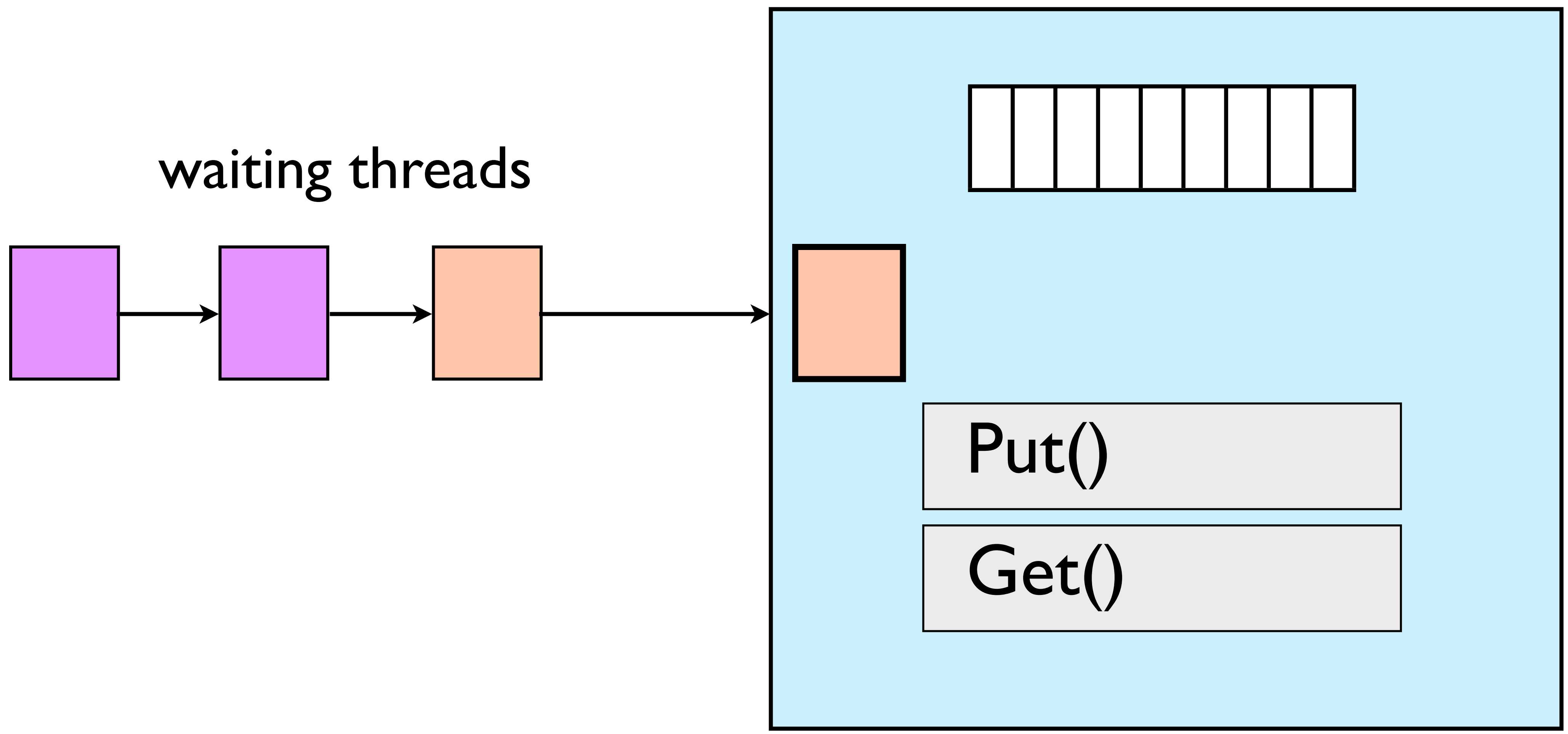
A monitor


shared data
wait queue of threads
trying to enter the monitor
at most one thread
in monitor at a
time
operations (methods)

Monitor example: a workqueue

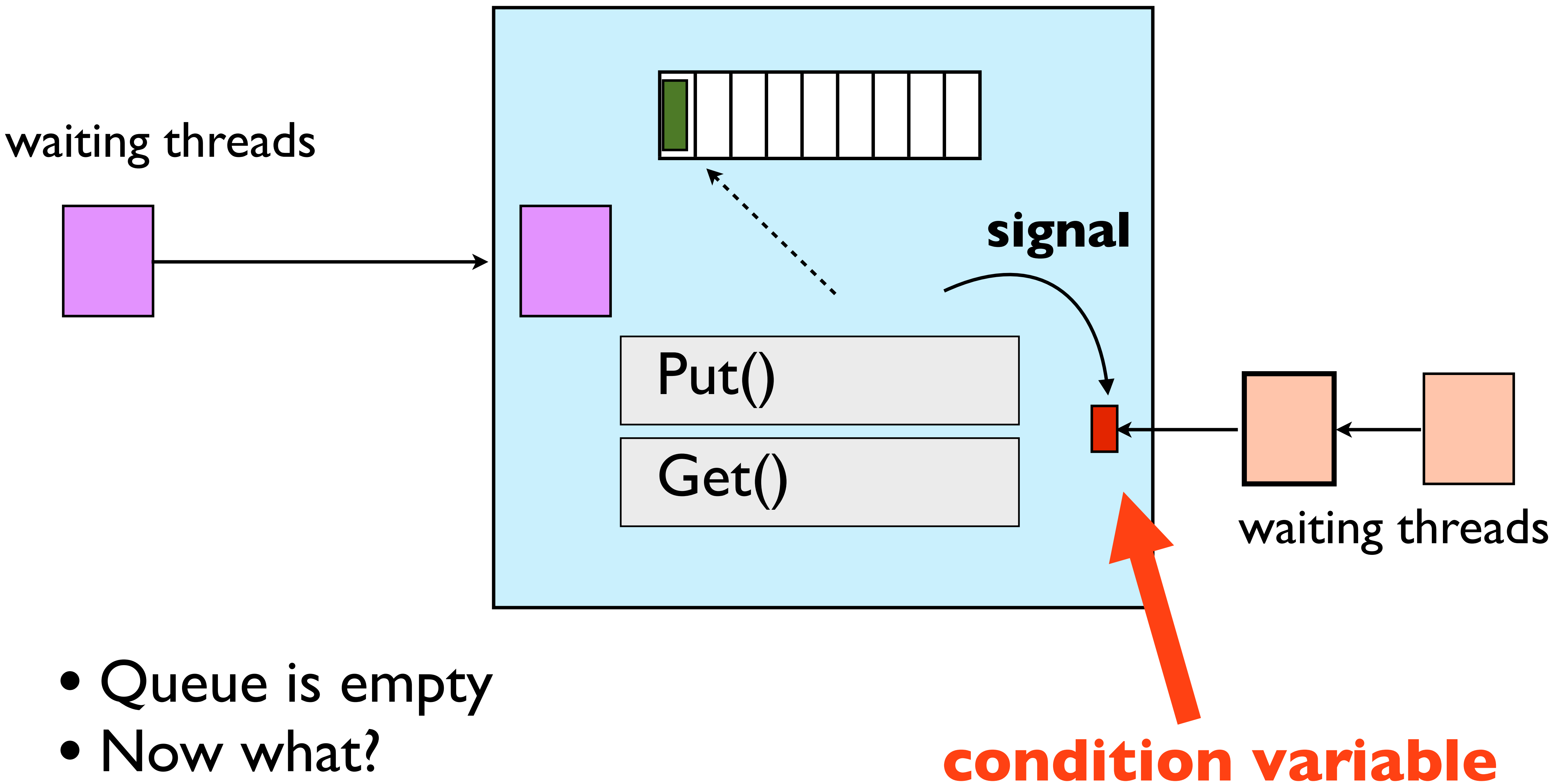


Monitor example: a workqueue



- Queue is empty
- Now what?

Monitor example: a workqueue



- Queue is empty
- Now what?

Condition Variables

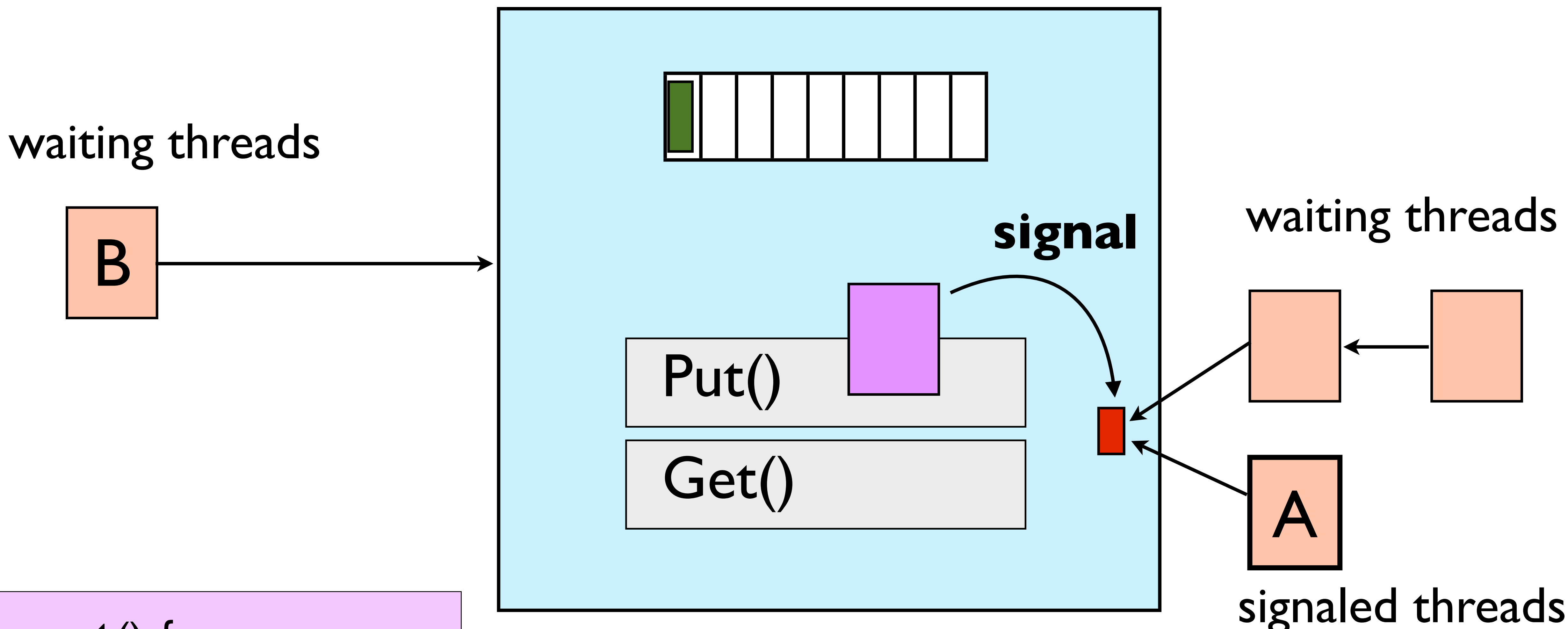
- wait(c)
  - release monitor lock
  - wait for a signal
  - then recapture monitor lock
- signal(c)
  - wake up at most one waiting thread
  - if no waiting threads, signal is lost!
- broadcast(c)
  - wake up all waiting threads

Workqueue pseudocode

```
Monitor {
    Queue      q
    Condition  notEmpty

    put(w) {
        q.push(w)
        signal(notEmpty)
    }

    get() {
        if (q.empty)
            wait(notEmpty)
        q.pop()
    }
}
```

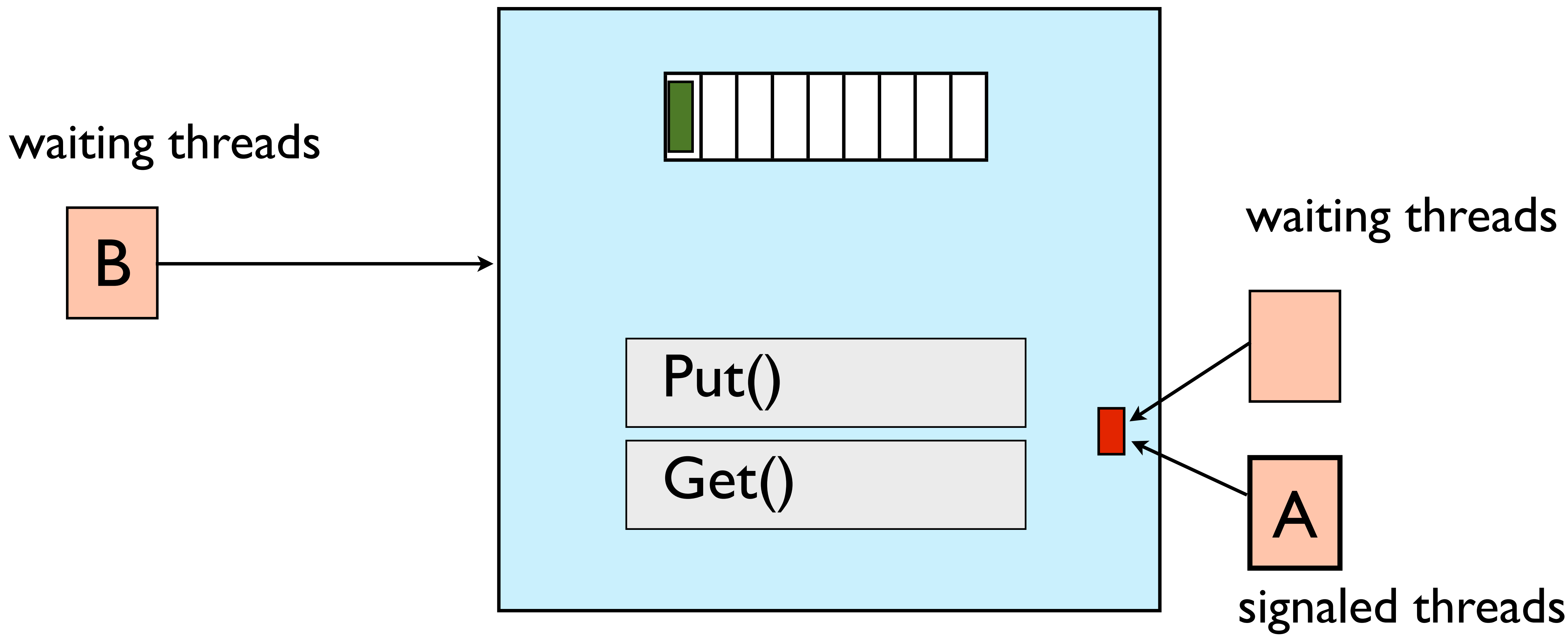
hmm..... is this right?

Monitor example: a workqueue



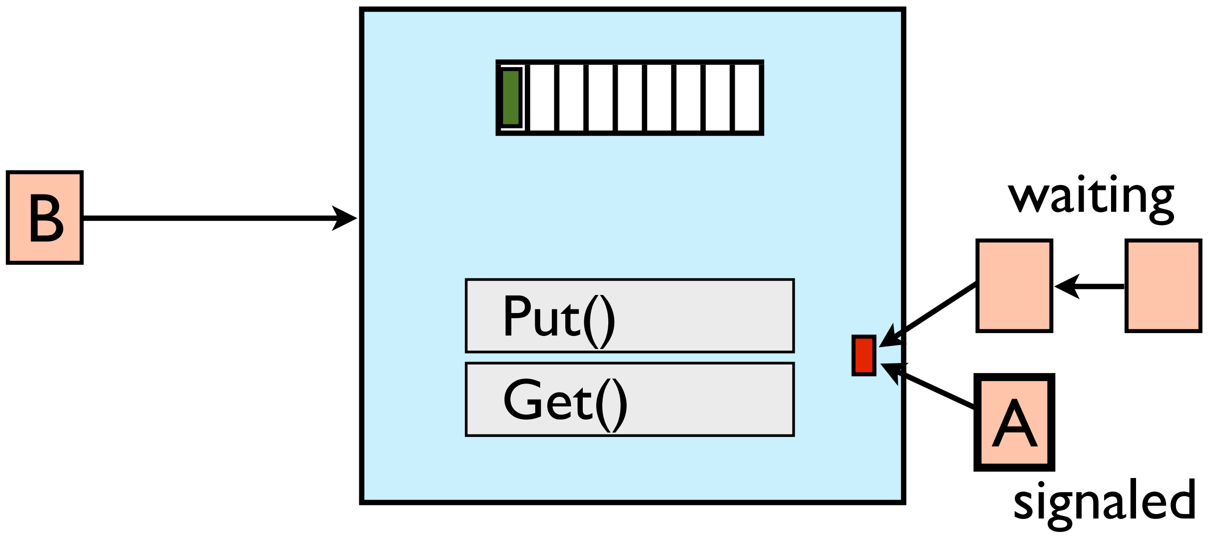

```
put() {
        q.push()
        signal(isEmpty)
}
```

Monitor example: a workqueue



- Put() was just called
- Who enters monitor next: A or B?

Monitor example: a workqueue



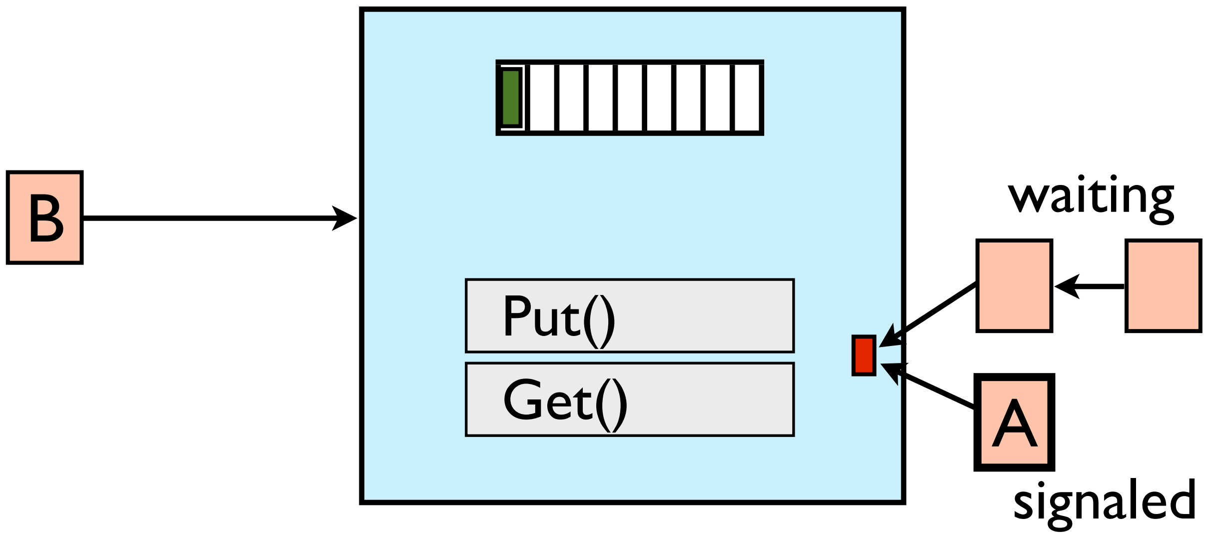
- What if A enters next? (Hoare style monitors)
  - this works fine

```
get() {
        if (q.empty)
                wait(notEmpty)
        q.pop()
}
```

pops the new item → q.pop()

A is blocked here (after wait(notEmpty))

Monitor example: a workqueue



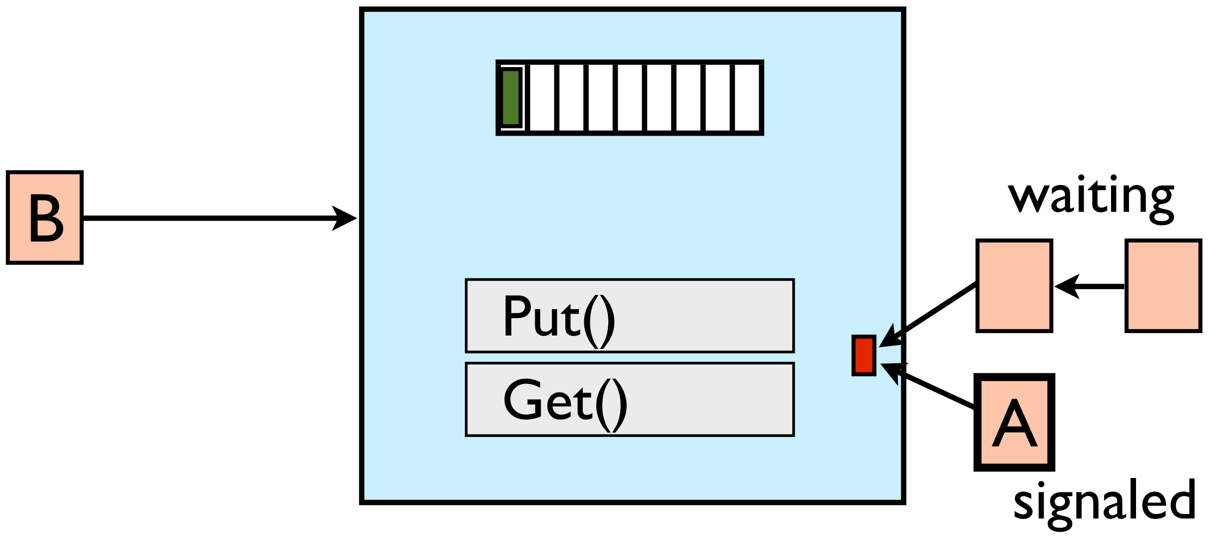
- What if B enters next? (Mesa style monitors)
  - B pops item
  - A sees an empty queue!

```
get() {
        if (q.empty)
                wait(notEmpty)
        q.pop()
}
```

oops! → q.pop()

A is blocked here (after wait(notEmpty))

Monitor example: a workqueue



- What if B enters next? (Mesa style monitors)
  - B pops item
  - A sees an empty queue!

```
get() {
    while (q.empty)
        wait(notEmpty)
    q.pop()
}
```

fix: need a while loop!

A is blocked here

Monitor scheduling choices

- **Hoare[1] monitors:** signal(c) means
  - run waiter immediately
  - must restore monitor invariants before signalling
    - can't leave a mess for the waiter!
- **Mesa[2] monitors:** signal(c)
  - waiter is made ready, but the signaller continues
  - waiter runs some time later
  - being woken up is only a hint something changed
    - condition might not hold
    - must recheck (hence the **while** loop)

[1]Tony Hoare (Turing Award 1980)
[2]Mesa programming language, by Butler Lampson (Turing award 1992)

Pseudocode for Mesa monitors

(everyone uses Mesa semantics these days)

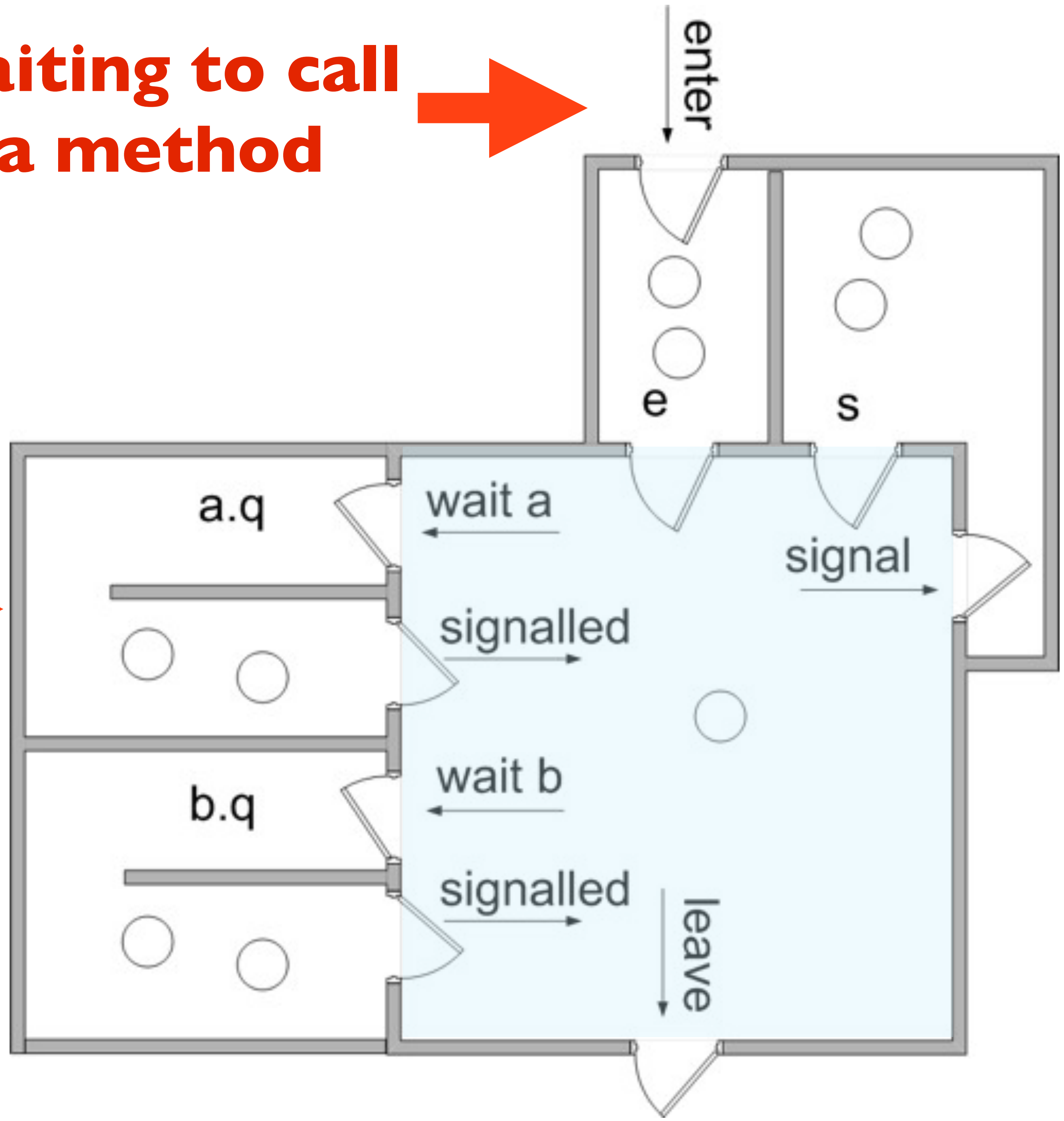
```
Monitor {
    Queue      q
    Condition  notEmpty

    put(w) {
        q.push(w)
        signal(notEmpty)
    }

    get() {
        while (q.empty)
            wait(notEmpty)
        q.pop()
    }
}
```

need to recheck every time we wake up

Hoare monitors

waiting to call a method

condition var wait queues



leave monitor after you signal, so a signalled thread can run

diagram courtesy of wikipedia

Mesa monitors

rejoin entrance queue when signalled

waiting to call a method

condition var wait queues

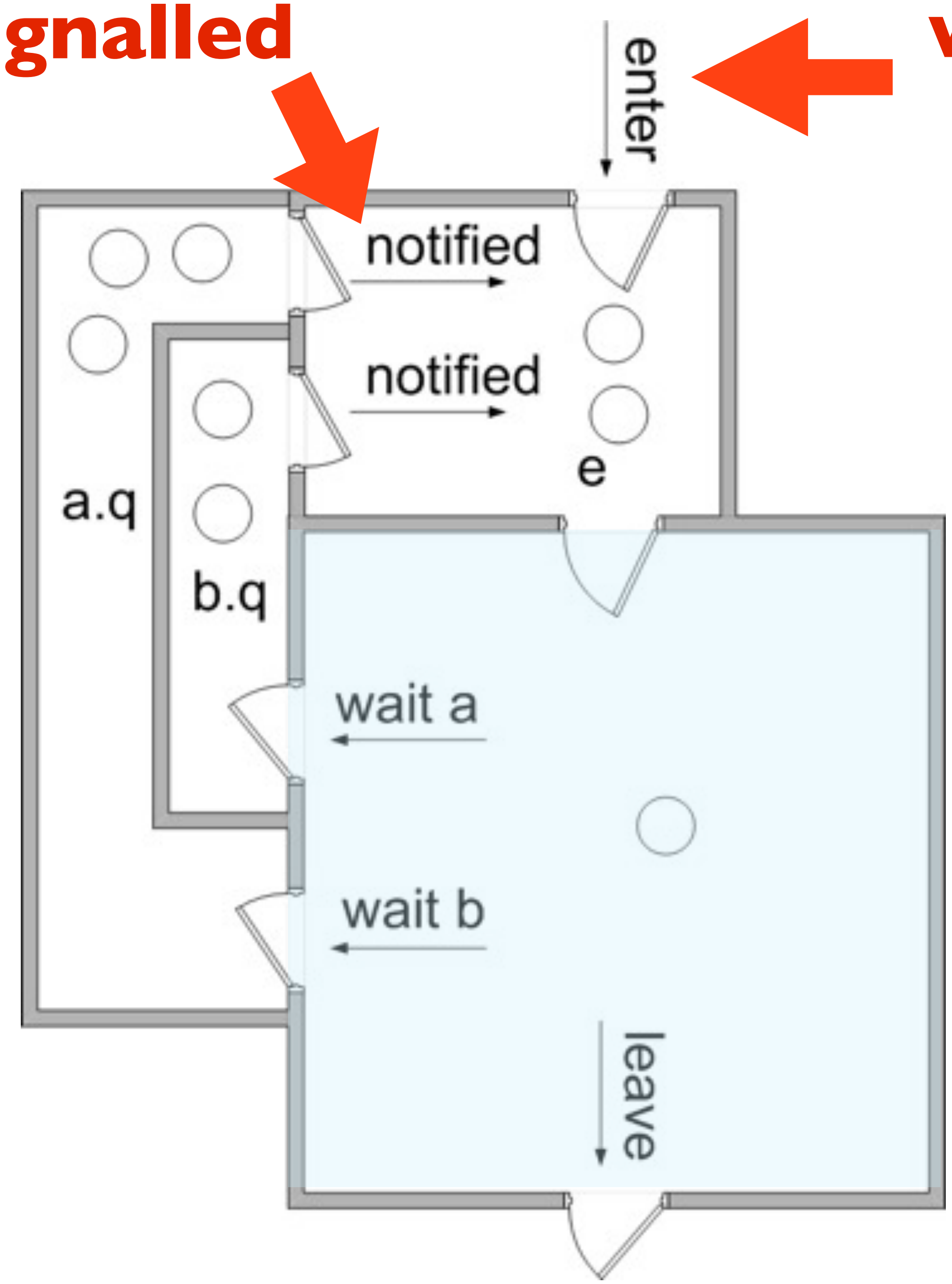


diagram courtesy of wikipedia

Monitors Summary

- Language and compiler support
  - mutual exclusion for methods
  - condition variables for waiting
- Problems?
  - heavyweight: no fine-grained locking
- Java:
  - monitors if you want them (Mesa scheduling)
  - locks and condition variables, too