B3CC: Concurrency

03:Threads (1)

Tom Smeding

What is concurrency?

- Consider multiple tasks being executed by the computer…
  - Tasks are concurrent with respect to each other if:
    - They *may* be executed out-of-order
    - Implies they can be executed at the same time, but *this is not required*
  - Concurrency: deal with lots of things at once

What is parallelism?

- Consider multiple tasks being executed by the computer…
  - Tasks are parallel if they are executed simultaneously:
    - Requires multiple processing elements
    - The primary motivation for parallel programming is to reduce the overall running time (wall clock) of the program: parallel execution
  - Parallelism: do lots of things at once

Question

- What does it mean for an application to be concurrent but not parallel?
  - Give an example
- What does it mean for an application to be parallel but not concurrent?
  - Give an example

Concurrency vs. Parallelism

- **Concurrency:** composition of independently executing processes
- **Parallelism:** simultaneous execution of (possibly related) computations

Concurrency

- Programming with multiple threads of control
  - A tool for *structuring programs* with multiple interactions
  - Examples: GUI, web server, different tasks in a game engine loop, …
- There is no single right answer
- In this course we will discuss several approaches: it is up to you to pick which is right for *your* application

Not easy!

More concurrency

- Concurrency appears on many levels:
  - Threads within a process that share an address space (*multithreading*)
  - Processes on a single system (*multiprogramming / multiprocessing*)
  - Tasks on multiple systems connected by a network (*distributed processing*)

Hierarchy / "threads", "threads" and "threads"

CPU Specifications

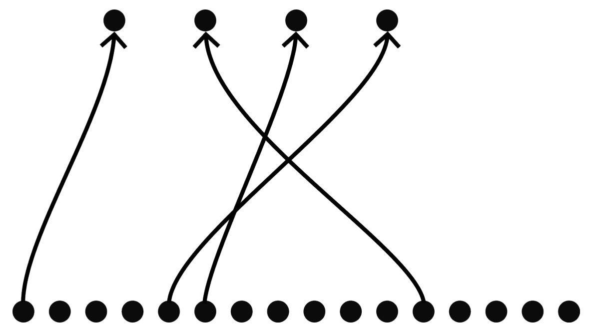
Total Cores	4	× 2 +
Total Threads	8	8

CPU Specifications

Total Cores	16	
# of Performance-cores	8	× 2
# of Efficient-cores	8	× 1 +
Total Threads	24	24

Hierarchy / "threads", "threads" and "threads"

- Physical CPU cores
- Logical CPU cores (simultaneous multithreading / (Intel) hyper-threading) ("threads")



- Kernel threads → *(scheduling: preemptive)* → "*context switching*"
- Processes
- User space threads / green threads / goroutines / … (lightweight) → *(scheduling: either preemptive or cooperative)*

Processes & Threads

- A (kernel) *thread* is…
  - An execution context
  - Contains all the information a CPU needs to execute a (logically sequential) stream of instructions
    - i.e. register set, stack, program counter (a.k.a. instruction pointer), (potentially) thread-local storage
- A *process* is…
  - A running instance of a computer program
  - Consists of at least one (kernel) thread
  - Separate memory space from other processes on the system
- Threads within a process share resources, but execute independently

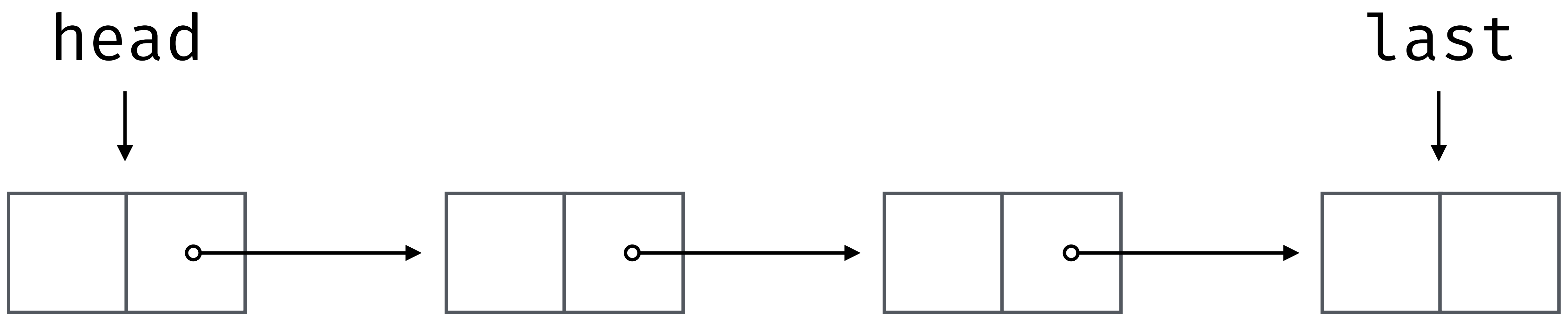
Processes & Threads: Programming Languages

- Many programming languages support threading in some capacity
  - Haskell: *M:N* hybrid threading model mapping *M* user space threads (`forkIO`) onto *N* kernel threads (via `+RTS -N<n>`) → *user space threads*
  - C/C++ provide access to the native threading APIs of the OS; POSIX threads (`pthread_create`) on *nix, and process.h (`_beginthread`) on Windows. Various extensions can be built on top of these (OpenMP, TBB, …)
  - Some interpreted languages (Ruby, Python) support threading for concurrency, but not parallel execution (GIL)
  - Some languages for parallel computing (CUDA, OpenCL) have "threads" in *some* sense, but in an entirely different way… more on that later!

Threads: needs and difficulties

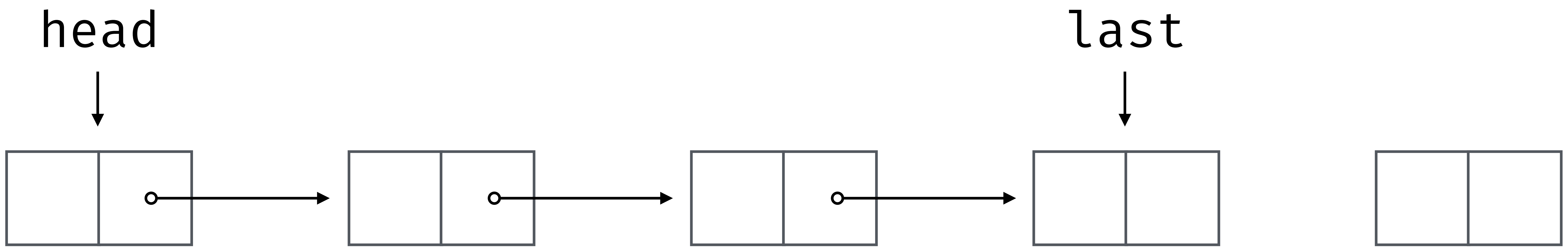
- Concurrent processes (threads) need special support
  - Communication among processes
  - Allocation of processor time
  - Sharing of resources
  - Synchronisation of multiple processes
- Concurrency can be dangerous to the unwary programmer:
  - Sharing global resources (order of read & write operations) → race conditions!
  - Management of allocation of resources (danger of deadlock)
  - Programming errors are difficult to locate (Heisenbugs)

Example: access to a global queue



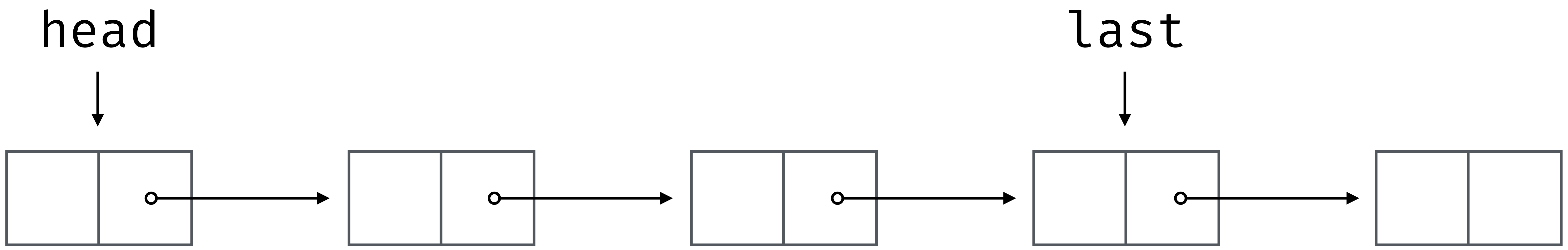
- Inserting:

Example: access to a global queue



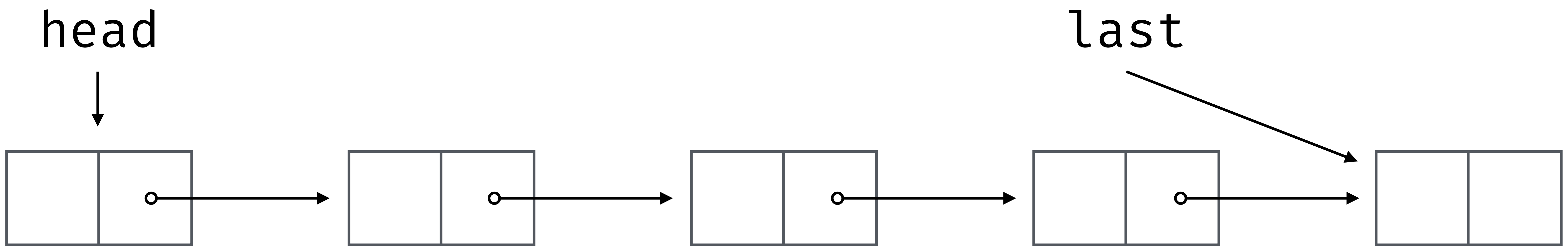
- Inserting:
  - Create new object

Example: access to a global queue



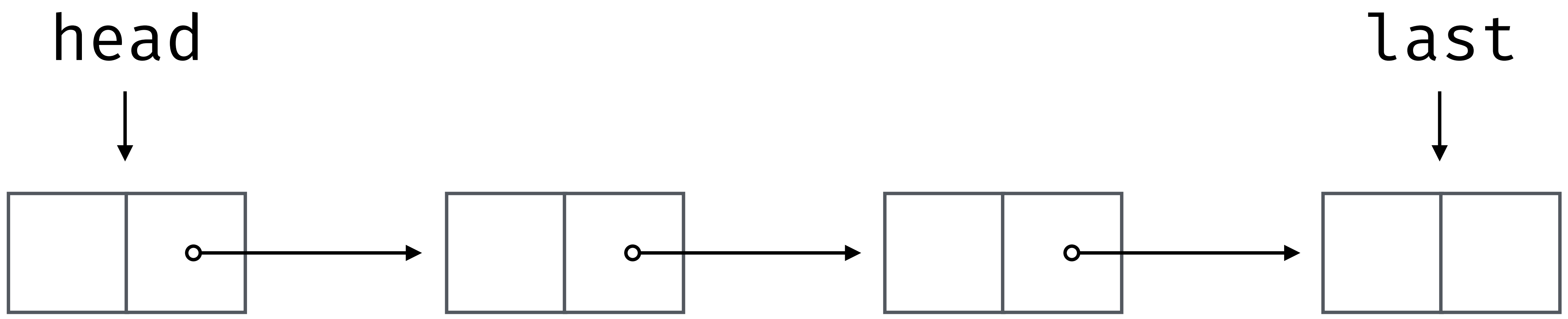
- Inserting:
  - Create new object
  - Set `last->next` to `&new`

Example: access to a global queue



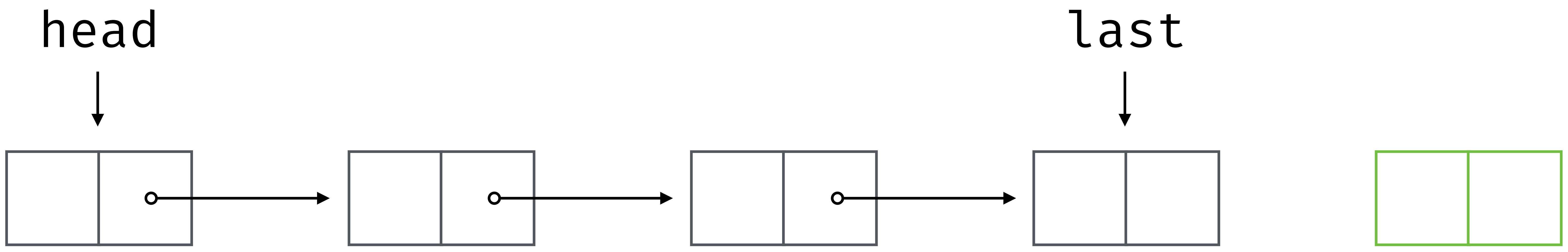
- Inserting:
  - Create new object
  - Set `last->next` to `&new`
  - Set `last` to `&new`

Example: concurrent access to a global queue



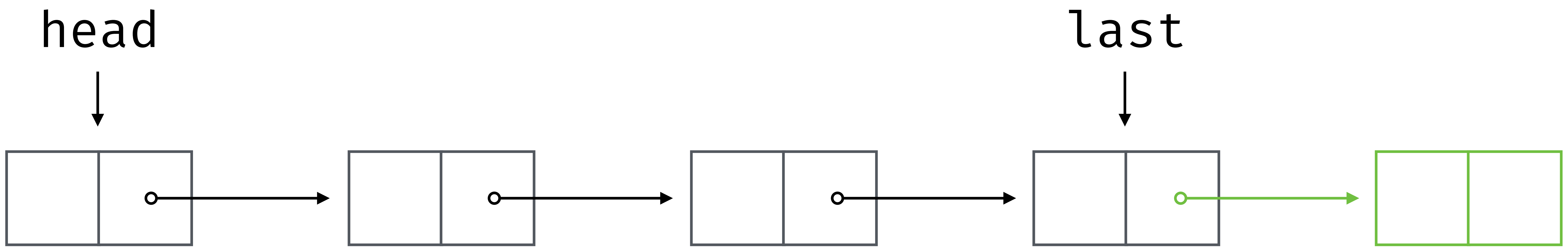
- Thread A:
- Thread B:

Example: concurrent access to a global queue



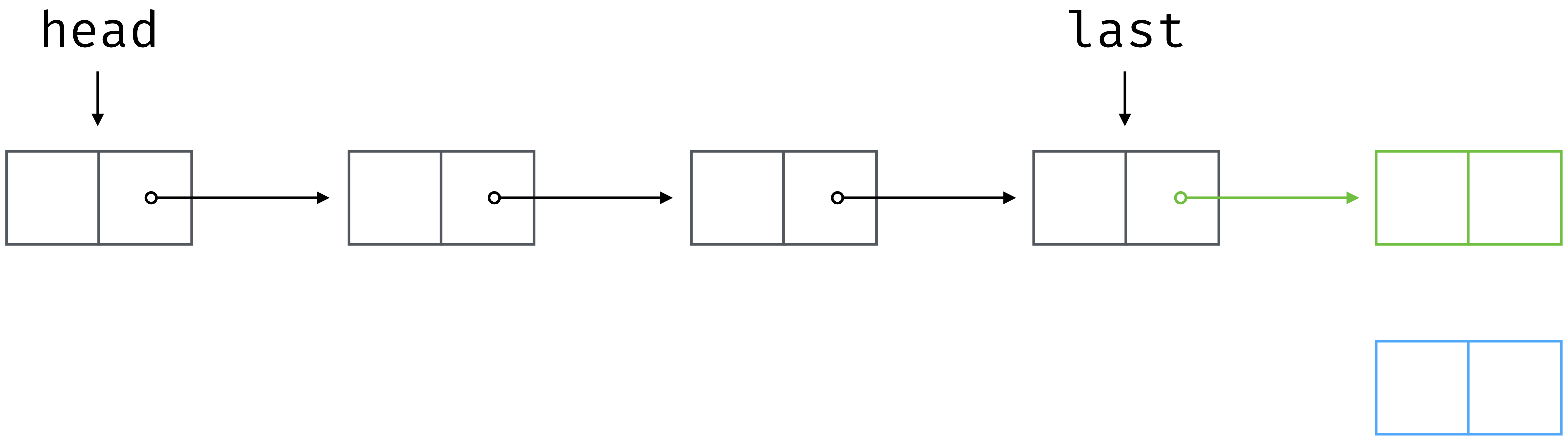
- Thread A:
  - Create new object
- Thread B:

Example: concurrent access to a global queue



- Thread A:
  - Create new object
  - Set `last->next` to `&new`
- Thread B:

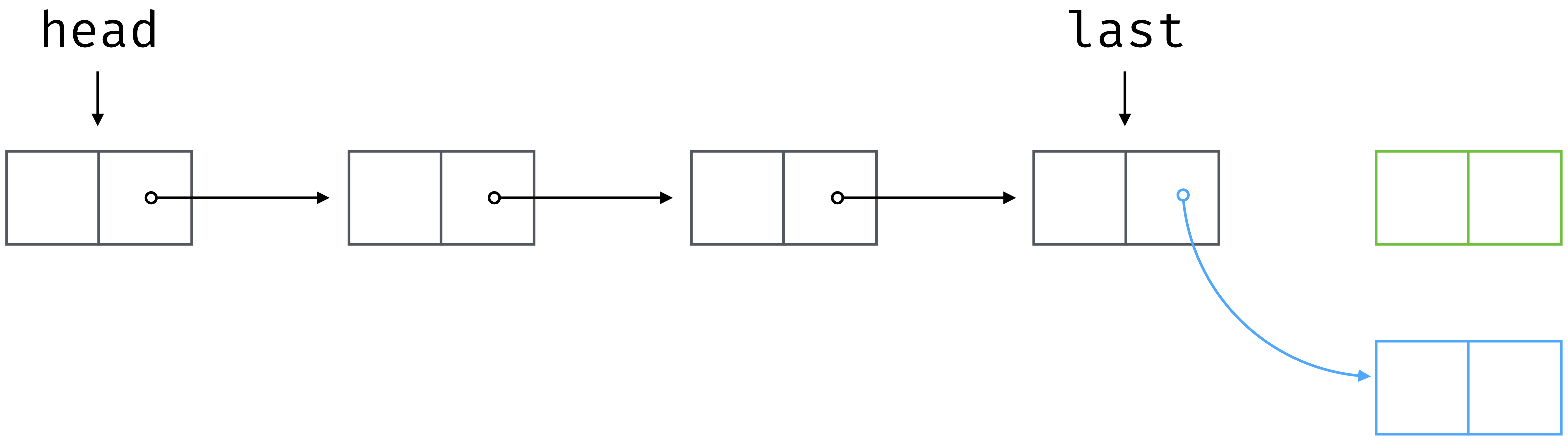
Example: concurrent access to a global queue



- Thread A:
  - Create new object
  - Set `last->next` to `&new`

- Thread B:
  - Create new object

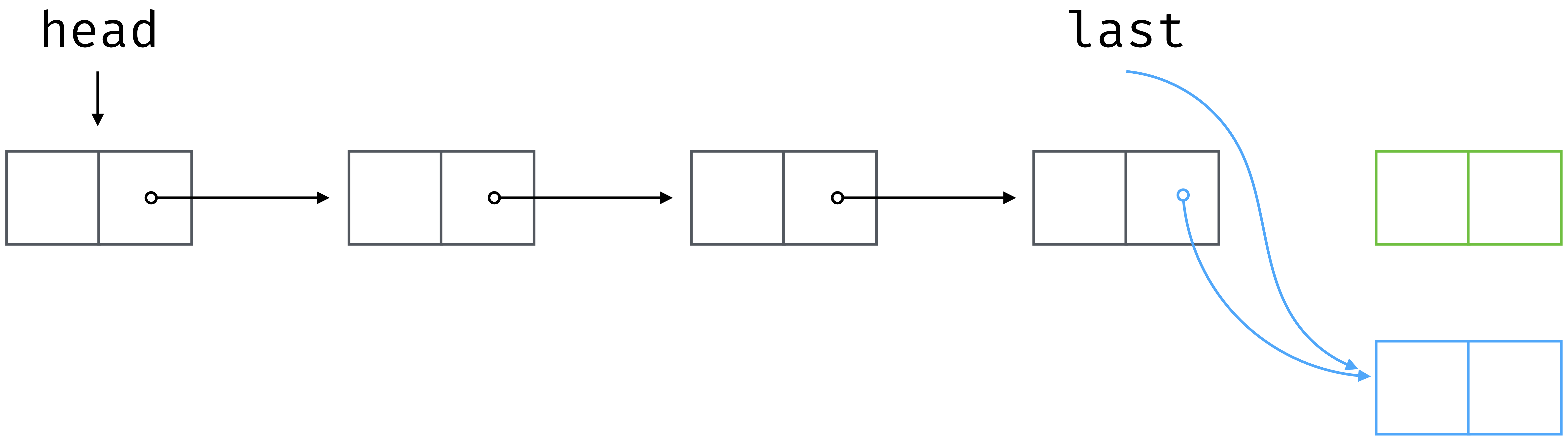
Example: concurrent access to a global queue



- Thread A:
  - Create new object
  - Set `last->next` to `&new`

- Thread B:
  - Create new object
  - Set `last->next` to `&new`

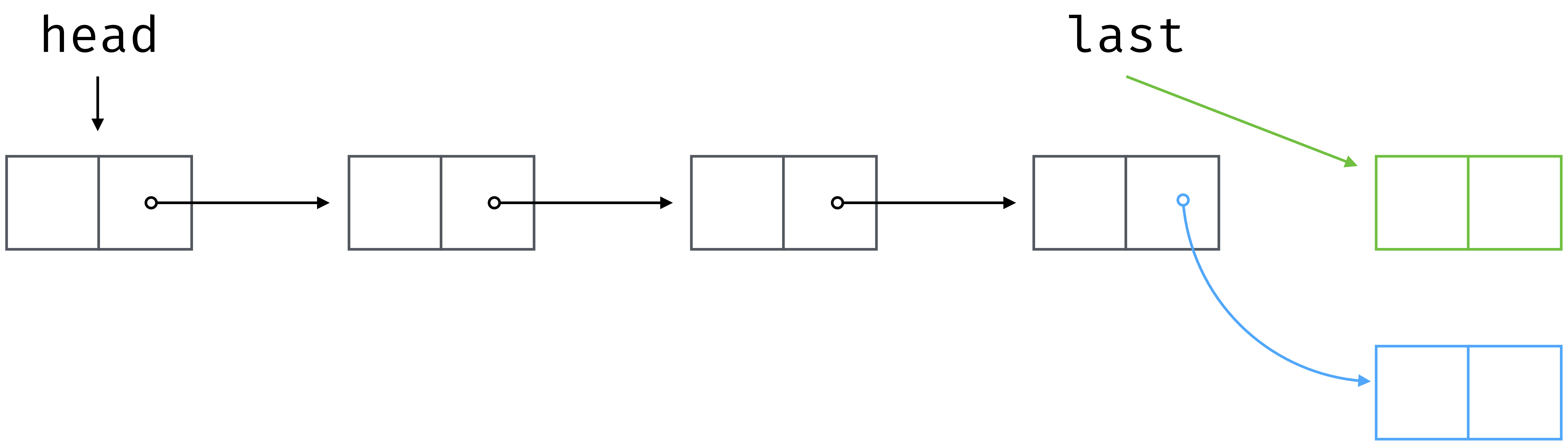
Example: concurrent access to a global queue



- Thread A:
  - Create new object
  - Set `last->next` to `&new`

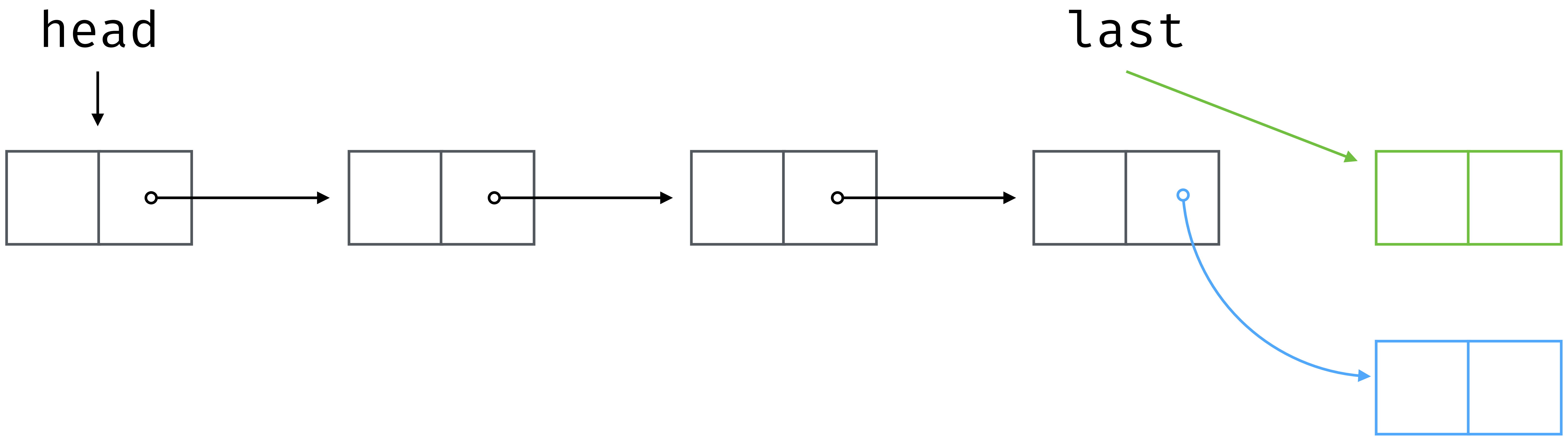
- Thread B:
  - Create new object
  - Set `last->next` to `&new`
  - Set `last` to `&new`

Example: concurrent access to a global queue



- Thread A:
  - Create new object
  - Set `last->next` to `&new`
  - Set `last` to `&new`
- Thread B:
  - Create new object
  - Set `last->next` to `&new`
  - Set `last` to `&new`

Example: concurrent access to a global queue



- Lessons learned
  - We have to control access to shared resources (such as shared variables)
  - We can do this by *controlling access to the code* utilising those shared resources: *critical sections*

Example: concurrent access to a global queue

- Only one thread at a time should have access to the queue:
  - Thread A creates a new object, sets `last->next` pointer
  - Thread A is suspended
  - Thread B is scheduled: since Thread A is currently in `insert`, has to wait
  - Thread A is resumed, the data structure is in the same state as it was when it was suspended
  - Thread A completes operation
  - Thread B is allowed to execute `insert`

Concurrency control

- Processes/threads can
  - Compete for resources
    - Processes may not be aware of each other
    - Execution must not be affected by each other
    - OS is responsible for controlling access
  - Cooperate by sharing a common resource
    - Programmer responsible for controlling access
    - Hardware / OS / programming language may provide support
- Threads of a process usually do not compete, but cooperate

Concurrency control

- We face three control problems:
  - *Mutual exclusion*: critical resources => critical sections
    - Only one thread at a time is allowed in a critical section
    - e.g. only one thread at a time is allowed to send commands to the GPU
  - *Deadlock*: everyone is waiting on everyone else
  - *Starvation*: e.g. when one thread always gets left out :/

Mutual Exclusion

Recall: Example: concurrent access to a global queue

- Only one thread at a time should have access to the queue:
  - Thread A creates a new object, sets `last->next` pointer
  - Thread A is suspended
  - Thread B is scheduled: since Thread A is currently in `insert`, **has to wait**
  - Thread A is resumed, the data structure is in the same state as it was when it was suspended
  - Thread A completes operation
  - Thread B is allowed to execute `insert`

Mutual exclusion

- Mutual exclusion (locking) protects shared resources
  - Only one thread at a time is allowed to access the critical resource
  - Modifications to the resource appear to happen atomically

```
mutex.lock();

... code ...

mutex.unlock();
```

Mutual exclusion

- Who is responsible?
  - *Software approach*: put responsibility on the processes themselves
  - *Systems approach*: provide support within the OS or programming language
- Hardware typically provides special-purpose machine instructions
- NOTE: Use the locking structures that come with your programming language!

… but let's try doing it ourselves anyway

Software approach to mutual exclusion

- Premise
  - 2 threads with *shared memory* (no assumptions about relative thread speed)
  - Elementary mutual exclusion at the level of *memory access*
    - Simultaneous accesses to the same memory location are serialised
- Requirements for the mutex:
  - Only one thread at a time is allowed in the critical section for a resource
  - No deadlock or starvation on attempting to enter/leave the critical section
  - A thread must not be delayed access to a critical section when there is no other thread using it
  - A thread that halts in its non-critical section must do so without interfering with other threads

Mutual exclusion

- Usage conditions:
  - A thread remains inside its critical section for a short time only
    - No potentially blocking operations should be executed inside the critical section

Attempt #1

- The plan:
  - Threads take turns executing the critical section
  - Exploit serialisation of memory access to implement serialisation of access to the critical section
- Employ a shared variable (memory location) `turn` that indicates whose turn it is to enter the critical section

- Thread A:

```
while (turn != 0)
  /* do nothing */ ;

<critical section>

turn = 1;
```

- Thread B:

```
while (turn != 1)
  /* do nothing */ ;

<critical section>

turn = 0;
```

Attempt #1

- Busy waiting (spin lock)
  - Process is always checking to see if it can enter the critical section
  - Implements mutual exclusion
  - Simple
- Disadvantages
  - Process burns resources while waiting
  - Processes *must alternate* access to the critical section
  - If one process fails *anywhere* in the program, the other is permanently blocked

Attempt #2

- The problem:
  - `turn` stores *who* can enter the critical section, rather than whether *anybody* may enter the critical section
- The new plan:
  - Store for each process whether it is in the critical section right now
  - `flag[i]` if process `i` is in the critical section

- Thread A:

```
while (flag[1])
  /* do nothing */ ;

flag[0] = true;
<critical section>
flag[0] = false;
```

- Thread B:

```
while (flag[0])
  /* do nothing */ ;

flag[1] = true;
<critical section>
flag[1] = false;
```

Attempt #2

- Does not guarantee exclusive access
- Race condition: time-of-check to time-of-use (TOCTOU)
- What if a process fails?
  - Outside the critical section: the other is not blocked ✔
  - Inside the critical section: the other is blocked :/ *(however, difficult to avoid)*

Attempt #3

- The goal:
  - Remove the gap between toggling the two flags
- The new updated plan:
  - Move setting the flag to before checking whether we can enter

- Thread A:

```
flag[0] = true;

while (flag[1])
  /* do nothing */ ;

<critical section>
flag[0] = false;
```

- Thread B:

```
flag[1] = true;

while (flag[0])
  /* do nothing */ ;

<critical section>
flag[1] = false;
```

Attempt #3

- Is it working now?
  - No. The gap can cause a *deadlock* now >_>
  - *Deadlock: when each member of a group of threads is waiting for another to take action (e.g. waiting for another to release a lock)*

Attempt #4

- Previous problem:
  - Thread sets its own state before knowing the other threads' states, and *cannot back off*
- The new updated revised plan:
  - Thread retracts its decision if it cannot enter
- Thread A:

```
flag[0] = true;
while (flag[1]) {
  flag[0] = false;
  delay();
  flag[0] = true;
}
<critical section>
flag[0] = false;
```

- Thread B:

```
flag[1] = true;
while (flag[0]) {
  flag[1] = false;
  delay();
  flag[1] = true;
}
<critical section>
flag[1] = false;
```

Attempt #4

- Is it working now?
  - Close, but we may have a livelock =_=
  - *Livelock: The states of the group of threads are constantly changing with regard to each other, but none are progressing (e.g. trying to obtaining a lock, but backing off if it fails)*
  - A special case of resource starvation, and a risk for algorithms which attempt to detect and recover from deadlock

Attempt #5

- Improvements
  - We can solve this problem by combining the first and third attempts
  - In addition to the `flags` we use a variable indicating whose `turn` it is to have *precedence* in entering the critical section

Attempt #5: Peterson's algorithm

- Both threads are courteous and solve a tie in favour of the other
- Algorithm can be generalised to work with *n* threads

- Thread A:

```
flag[0] = true;
turn    = 1;

while (flag[1]
    && turn == 1)
  /* do nothing */ ;

<critical section>

flag[0] = false;
```

- Thread B:

```
flag[1] = true;
turn    = 0;

while (flag[0]
    && turn == 0)
  /* do nothing */ ;

<critical section>

flag[1] = false;
```

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Attempt #5: Peterson's algorithm

- **Statement:** mutual exclusion
  Threads 0 and 1 are never in the critical section at the same time
- **Proof:**
  - If P_0 is in the critical section then
    - `flag[0]` is true
    - `flag[1]` is false OR `turn` is zero OR P_1 is trying to enter the critical section, after setting `flag[1]` to `true` but before setting `turn` to zero
  - For both P_0 and P_1 to be in the critical section
    - `flag[0]` AND `flag[1]` AND `turn=0` AND `turn=1`

Locking: real life

- Again: Peterson's algorithm is a theoretical exercise
- Please use the facilities in your programming language
- If you are implementing a mutex yourself (or are doing the first practical, IBAN!), use the compare-and-swap operation (casIORef)! (explained on Monday)

https://hackage.haskell.org/package/atomic-primops-0.8.8/docs/Data-Atomics.html#v:casIORef

For the practical

IORefs

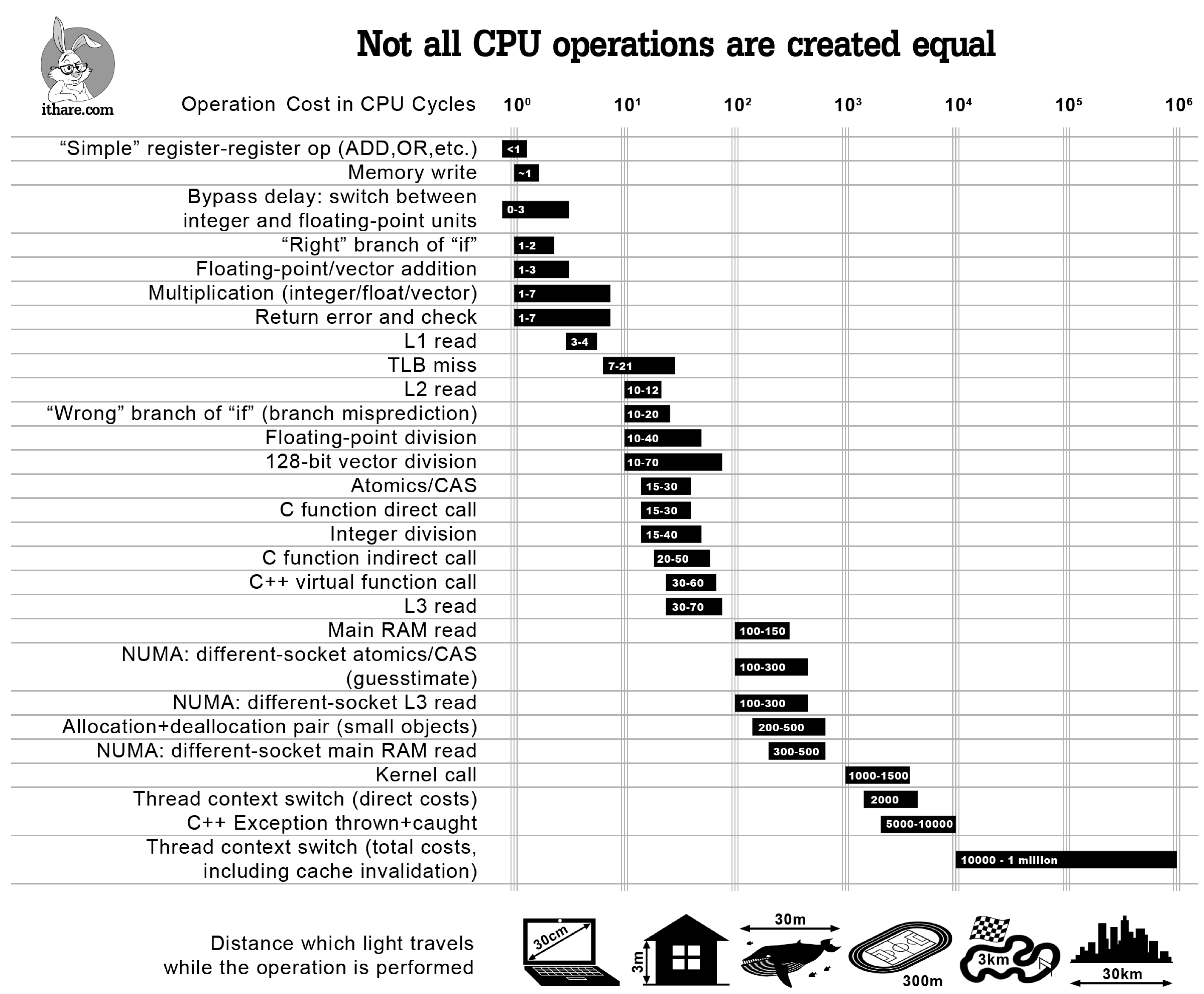
- In most languages variables are mutable by default
- In Haskell, mutable variables must be handled explicitly
  - Notice that whether a variable is mutable is now reflected in its type!

```haskell
import Data.IORef

newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

- More information on Monday
- Check the documentation!
  - https://hackage.haskell.org/package/base-4.17.2.1/docs/Data-IORef.html
  - https://hackage.haskell.org/package/atomic-primops-0.8.8/docs/Data-Atomics.html

Extra slides



http://ithare.com/wp-content/uploads/part101_infographics_v08.png