Race conditions

ID2010

2022

Race condition

- IF multiple processes or threads compete for a non-sharable and mutable resource
- AND access to the resource is not controlled
- THEN there is a *race* to modify the resource

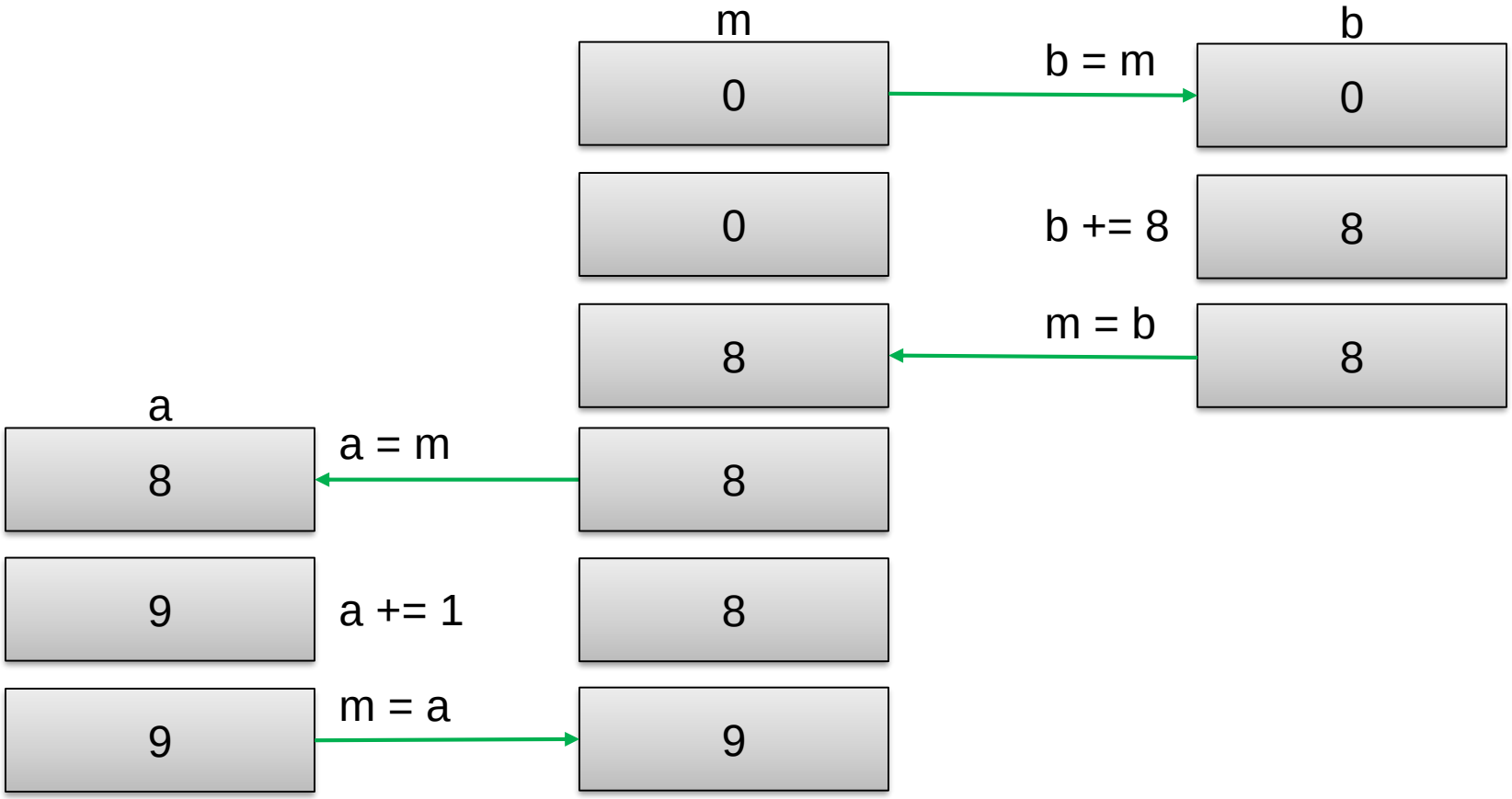
Race condition

- Mutable (writable) resources:
  - register, memory, file, hardware port
- Non-shared access:
  - Read-Modify-Write
- Shared and uncontrolled access:
  - Read-Modify-Read-Write-Modify-Write

Race condition

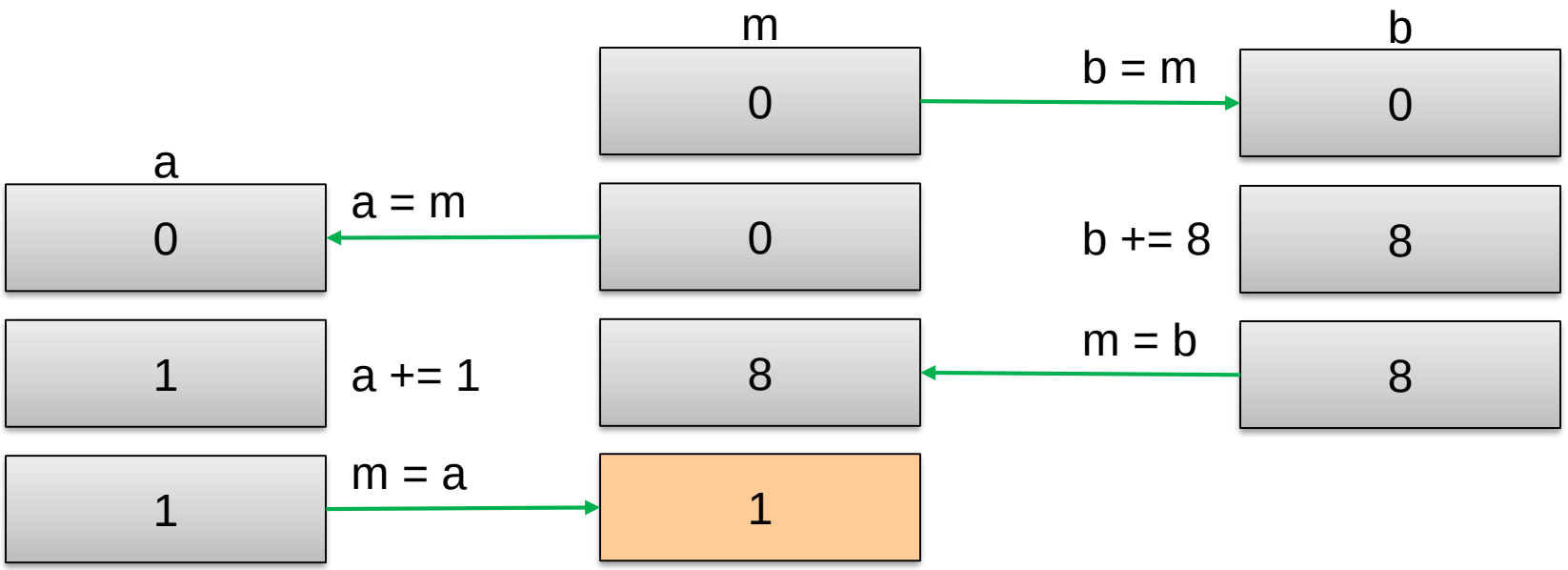
- State is corrupted by overlapping updates
- The race condition means that a corruption is *possible*
- The probability of a corruption actually happening depends on the system
- non-deterministic errors, bugs are hard to find

Race condition on a variable



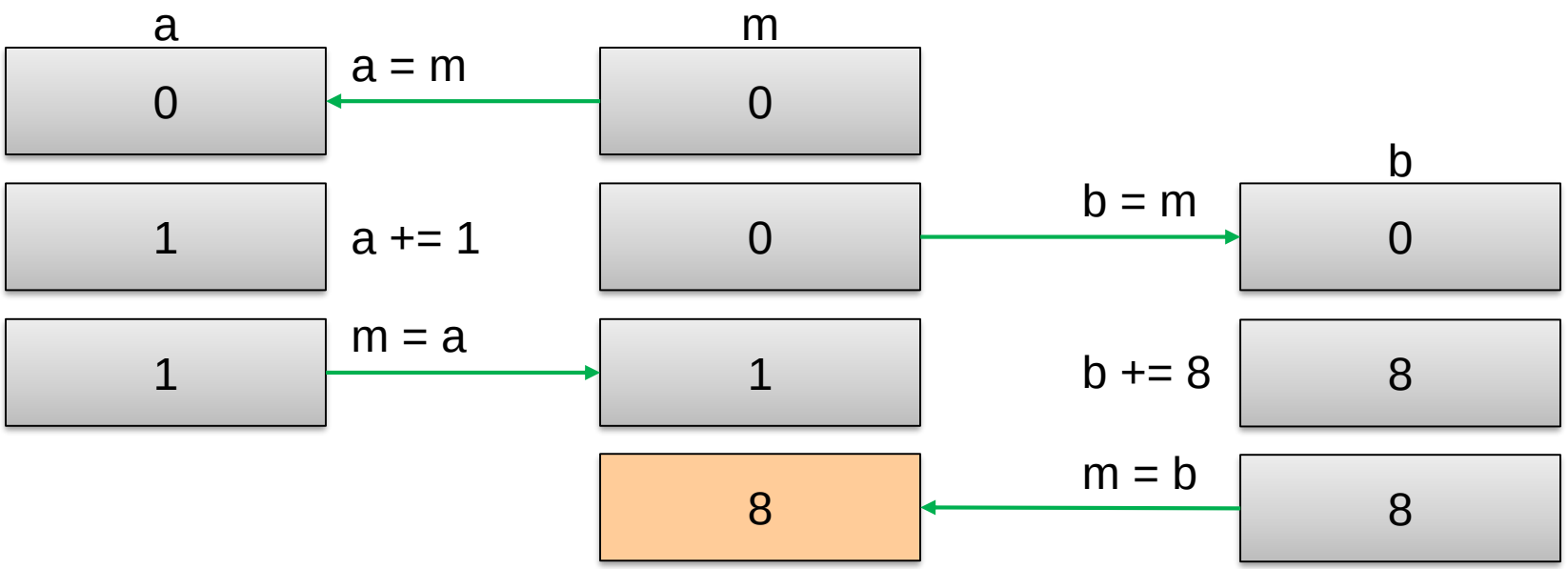
Thread a and thread b updates m separately.
The value of m is 9 (correct).

Race condition on a variable



The update from thread b is overwritten by thread a.
The value of m is 1, not 9 (wrong by 8).

Race condition on a variable



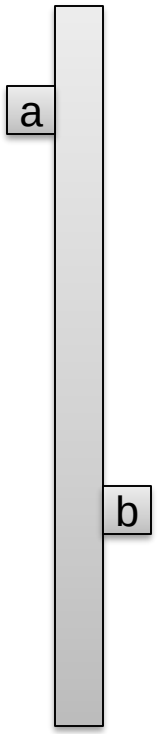
The update from thread a is overwritten by thread b.
The value of m is 8, not 9 (wrong by 1).

Race condition on a variable

- The race condition is a possible bug
- The bug only happens when a and b overlap
- Errors in variable m accumulate over time

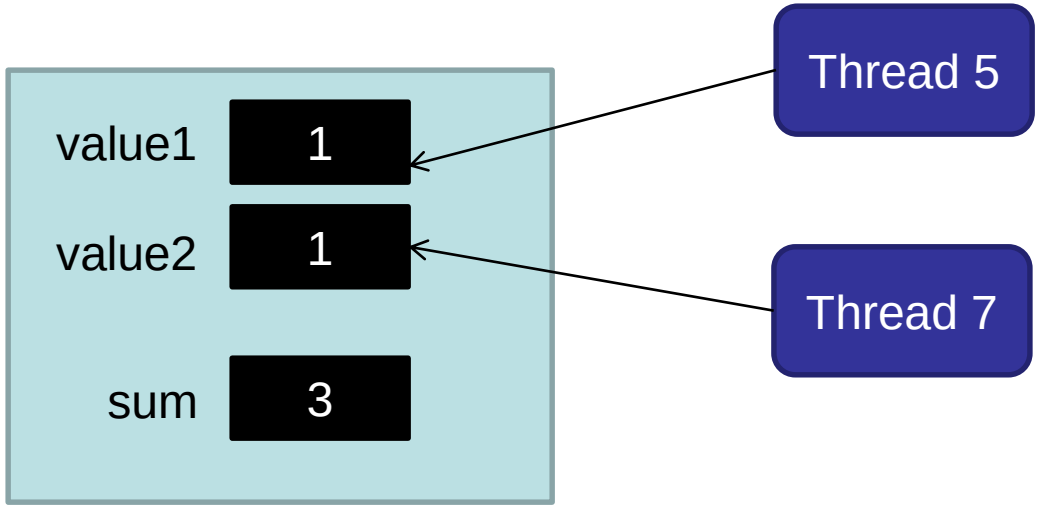
  and may not be recognized at first:

  0 + 1 + 0 + 8 + 1+ 1 + 0 + 8 +...



Generalizations

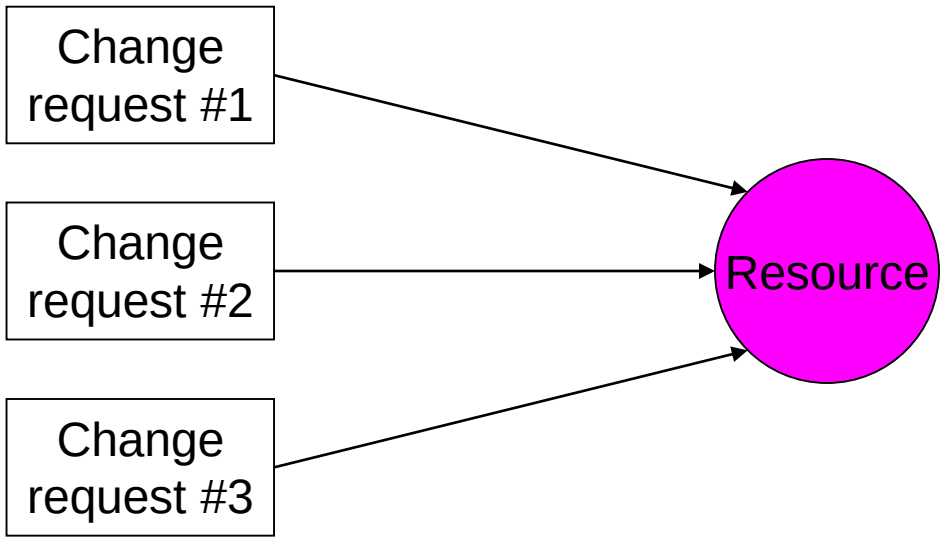
- A single variable is a special case
- of a block of data that must be internally and externally *consistent*



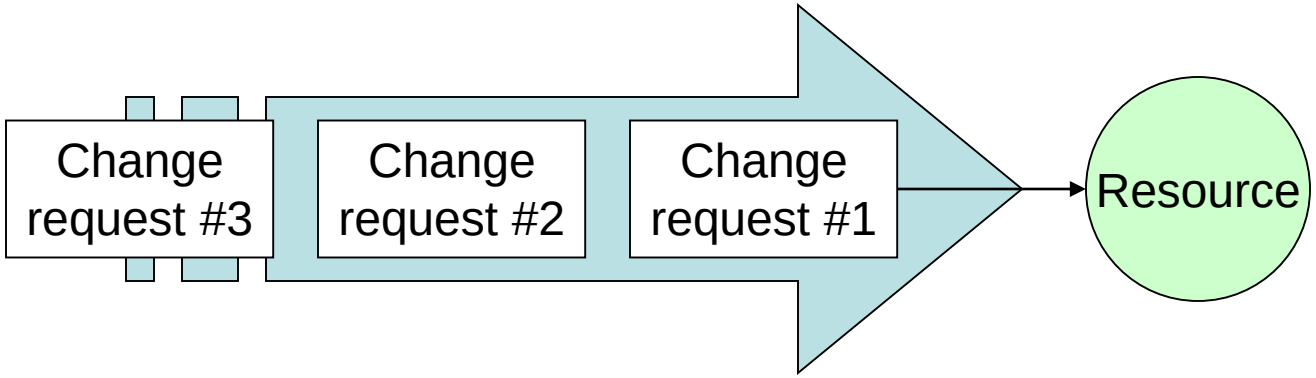
History

- 'Race condition' was used in 1954 referring to signals in electrical circuits
- Logic gates in particular may suffer if signals arrive at different times
- In software also known as 'data race'

Non-serialized operations



Serialized operations



The resource is protected by a mechanism which enforces serial access to it:

- Operating system
- Database manager
- Programming language synchronization

Synchronization

- Only one process may execute the critical section of code
- Acquire exclusive rights
- Execute critical section
- Release exclusive rights

Synchronization

```
...
acquire_permission(s);
a = m;
a = a + k;
m = a;
release_permission(s);
...
```

The entity s controls access to m.

Synchronization

- Semaphore – guards n resources
  - Any thread can return a resource
- Mutex – guards 1 resource
  - Only the currently owning thread can return the resource
- Threads block until the resource is granted

Semaphore

- *Counts the nof available resources*

cupSem

3

Semaphore

- *Counts the nof available resources*

1. Decrement counter

Semaphore

- *Counts the nof available resources*

cupSem

2

1. Decrement counter
2. Use resource

Semaphore

- *Counts the nof available resources*

1. Decrement counter
2. Use resource
3. Increment counter

Semaphore

- *Counts the nof available resources*

cupSem

3

No identification of resources.

1. Decrement counter
2. Use resource
3. Increment resource

Semaphore

- *Counts the nof available resources*

cupSem

3

1. Decrement counter
2. Use resource
3. Increment resource

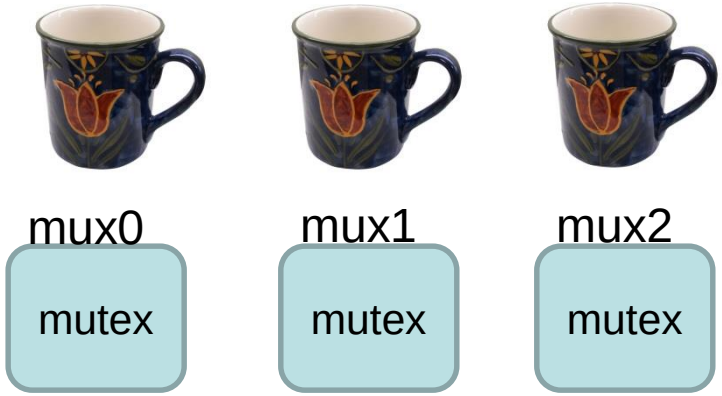
No identification of resources.
Increment and decrement are atomic.

Semaphore caveats

- Processes (i.e. programmers) must follow the protocol and **not**:
  - Forget to return a resource after use
  - Return a resource that was not requested
  - Hold a resource for too long
  - Use a resource anyway

Mutex

- A binary semaphore (one resource)



Mutex

- A binary semaphore (one resource)

1. Acquire

Mutex

- A binary semaphore (one resource)

1. Acquire
2. Use

Mutex

- A binary semaphore (one resource)

1. Acquire
2. Use
3. Return

Mutex

- A binary semaphore (one resource)
- The process acquiring the resource is the only one that can return it (ownership)
- The *synchronized* keyword in Java use any object as a monitor (a kind of mutex).

Monitor

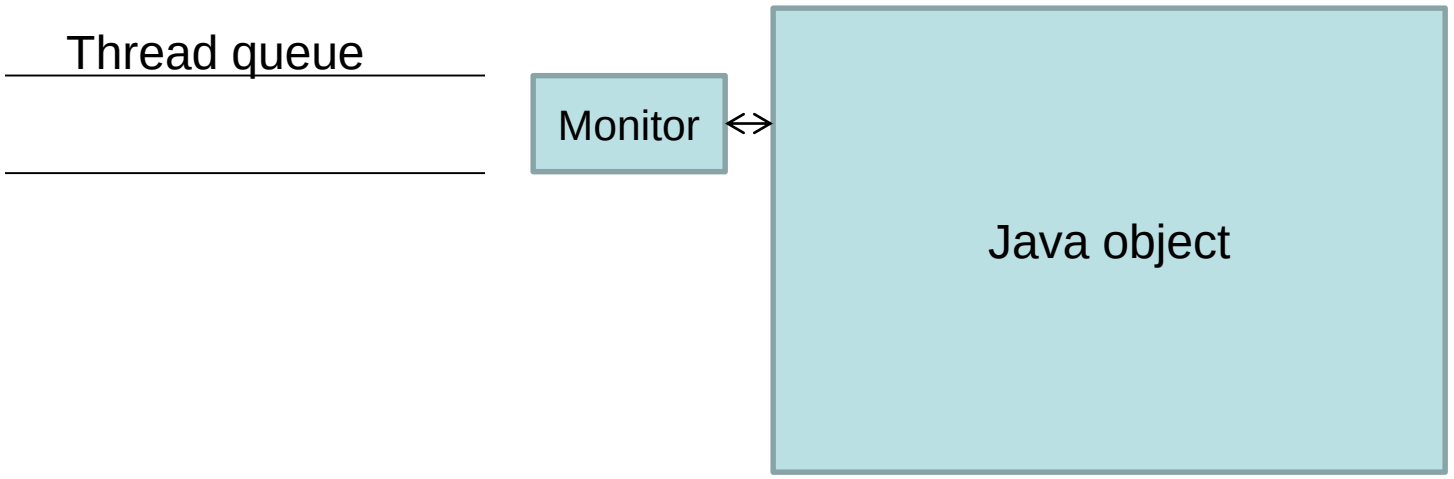
- A mutex with a thread queue

- Threads queue for the mutex
- Threads can yield the resource (wait)
- Threads can alert other threads (notify)

- In Java, every object has a monitor
- The *synchronized* keyword

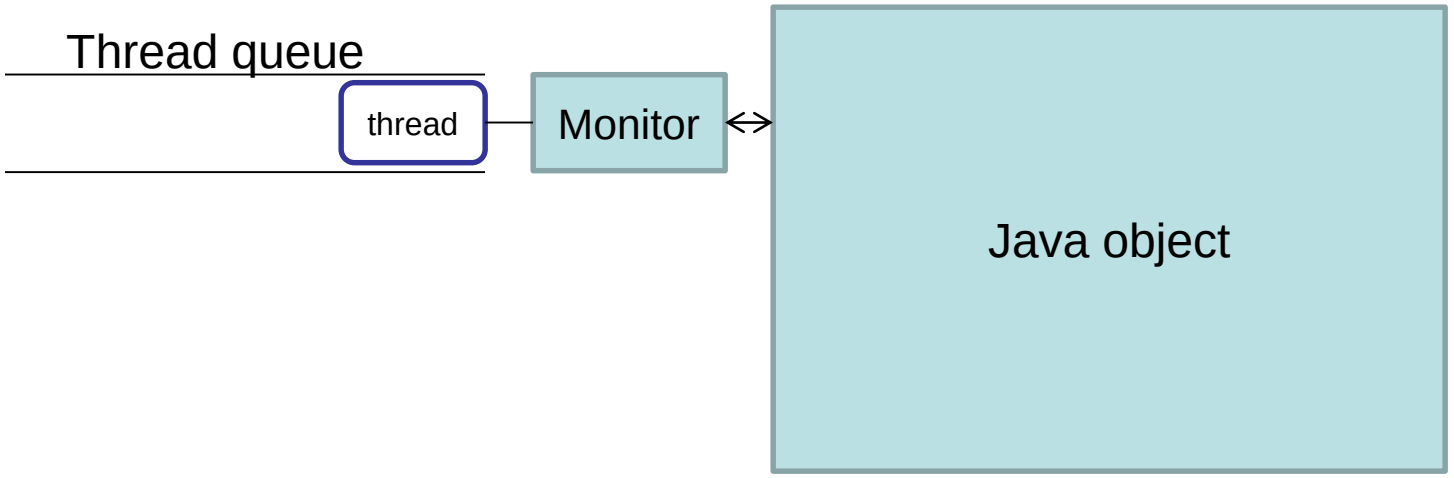
Monitor

- A mutex with a thread queue



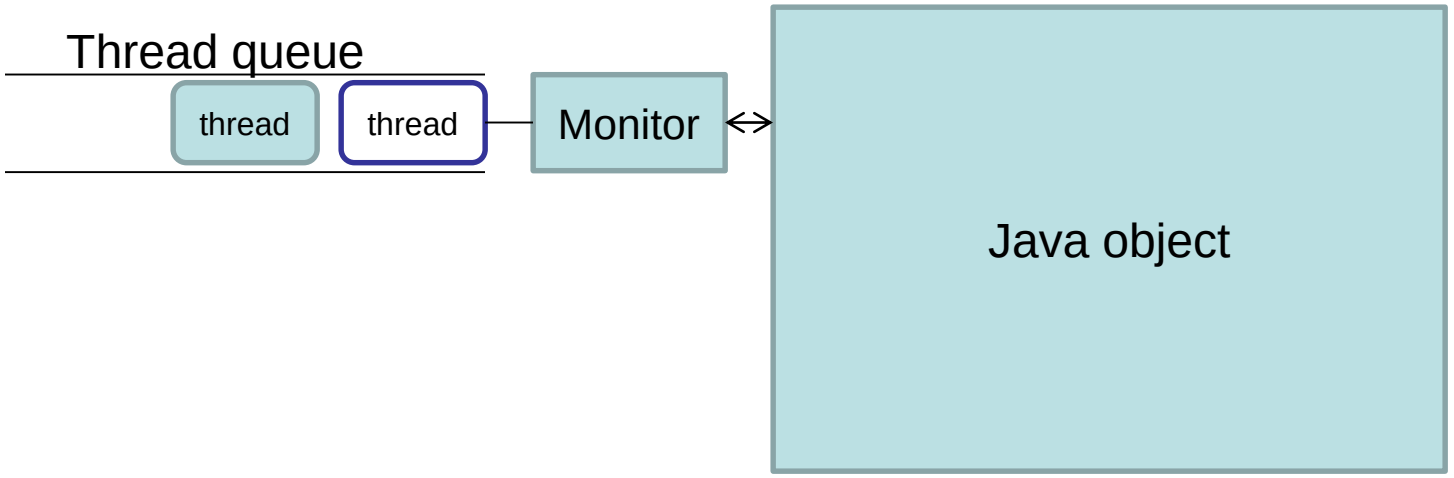
Monitor

- A mutex with a thread queue



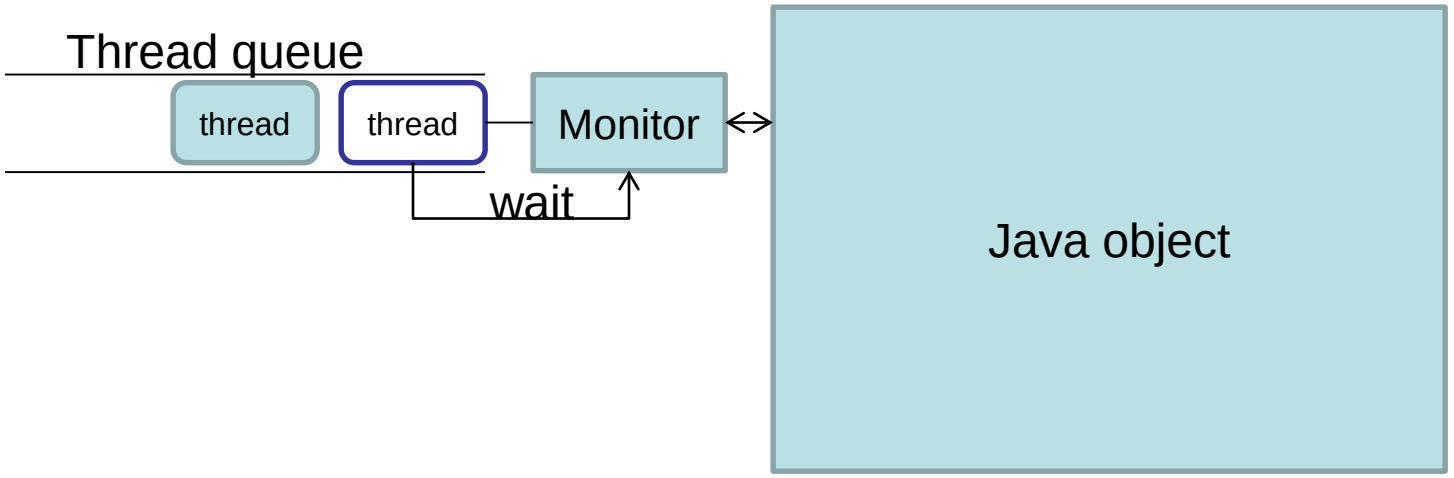
Monitor

- A mutex with a thread queue



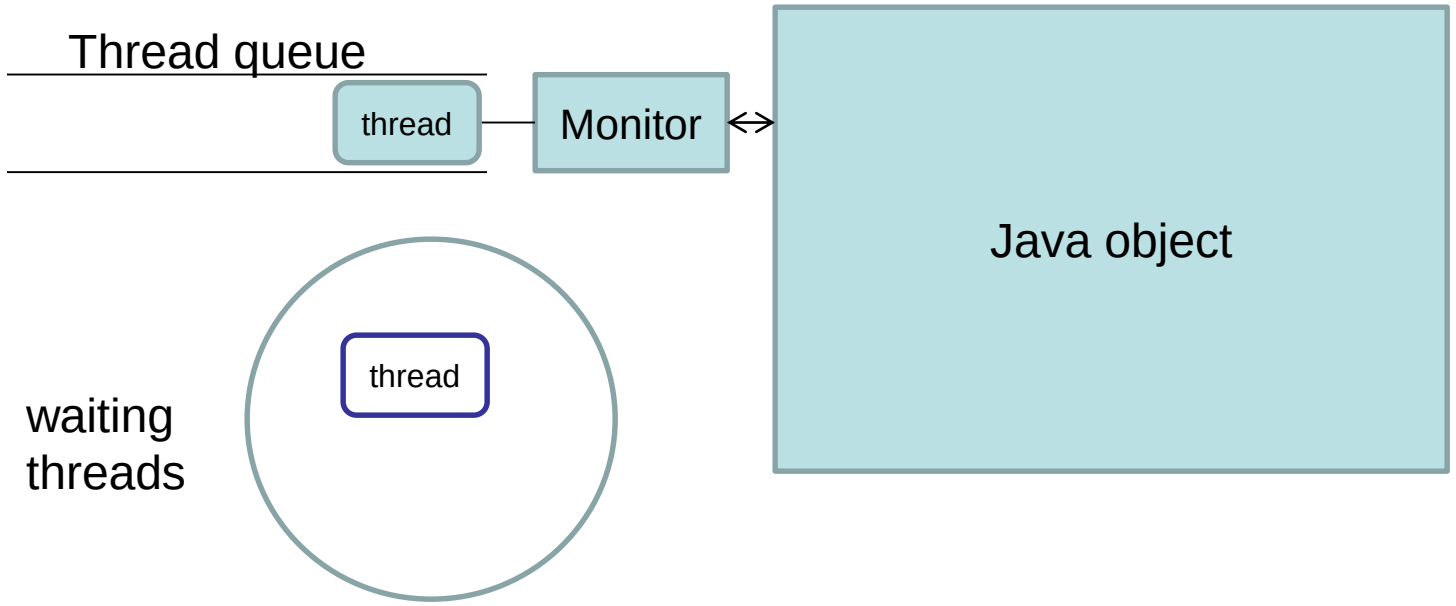
Monitor

- A mutex with a thread queue



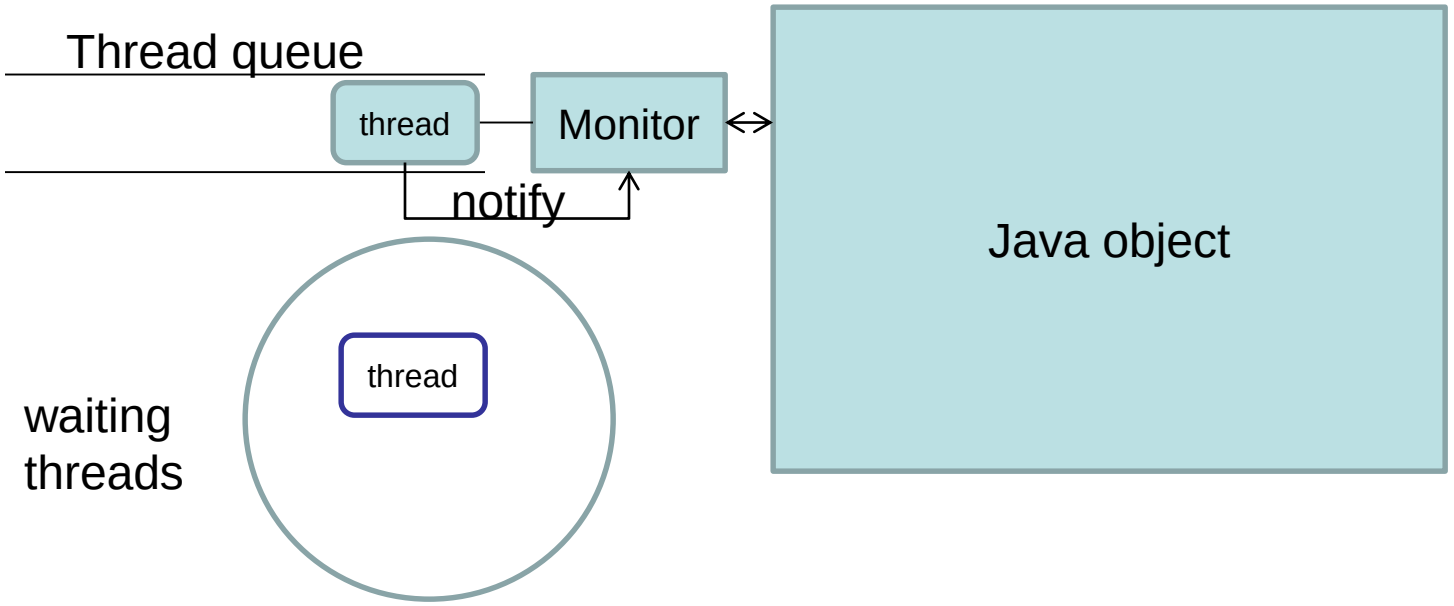
Monitor

- A mutex with a thread queue



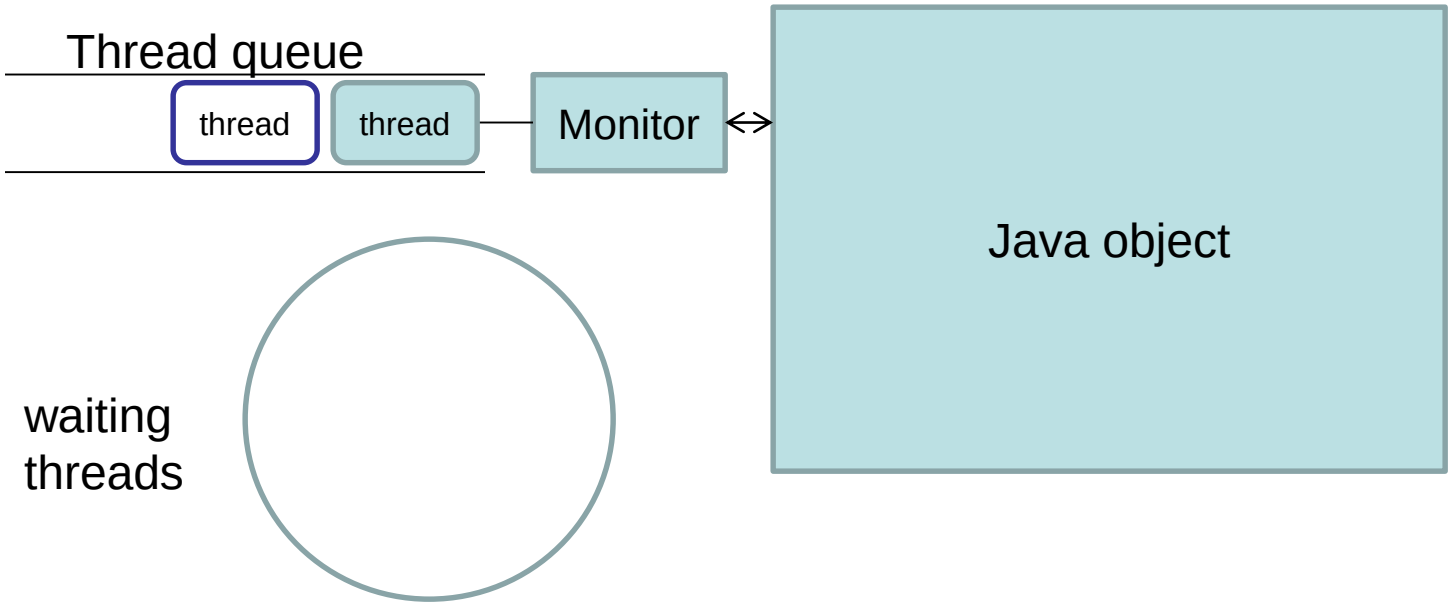
Monitor

- A mutex with a thread queue



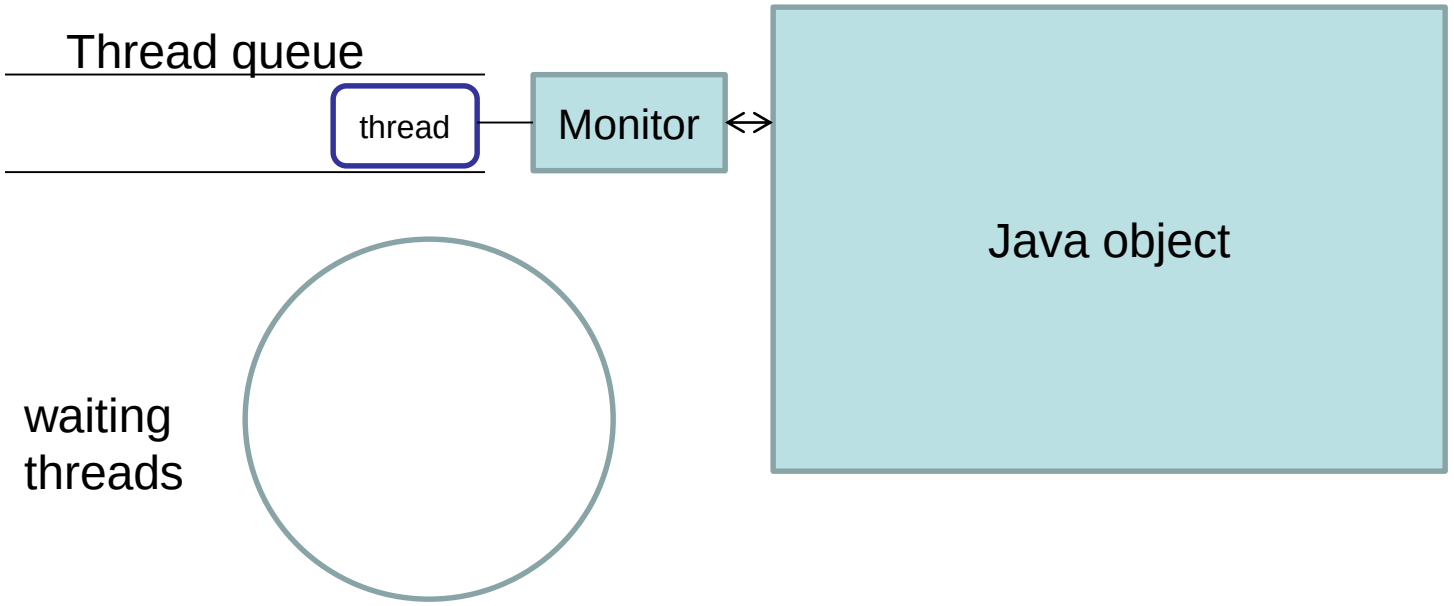
Monitor

- A mutex with a thread queue



Monitor

- A mutex with a thread queue



Some unsharable resources

- Process memory (part of)
- Firmware memory (NVRAM, EEPROM)
- Filesystems
- Databases
- Communication lines
- Network adapters
- User interface devices

Lock-free code

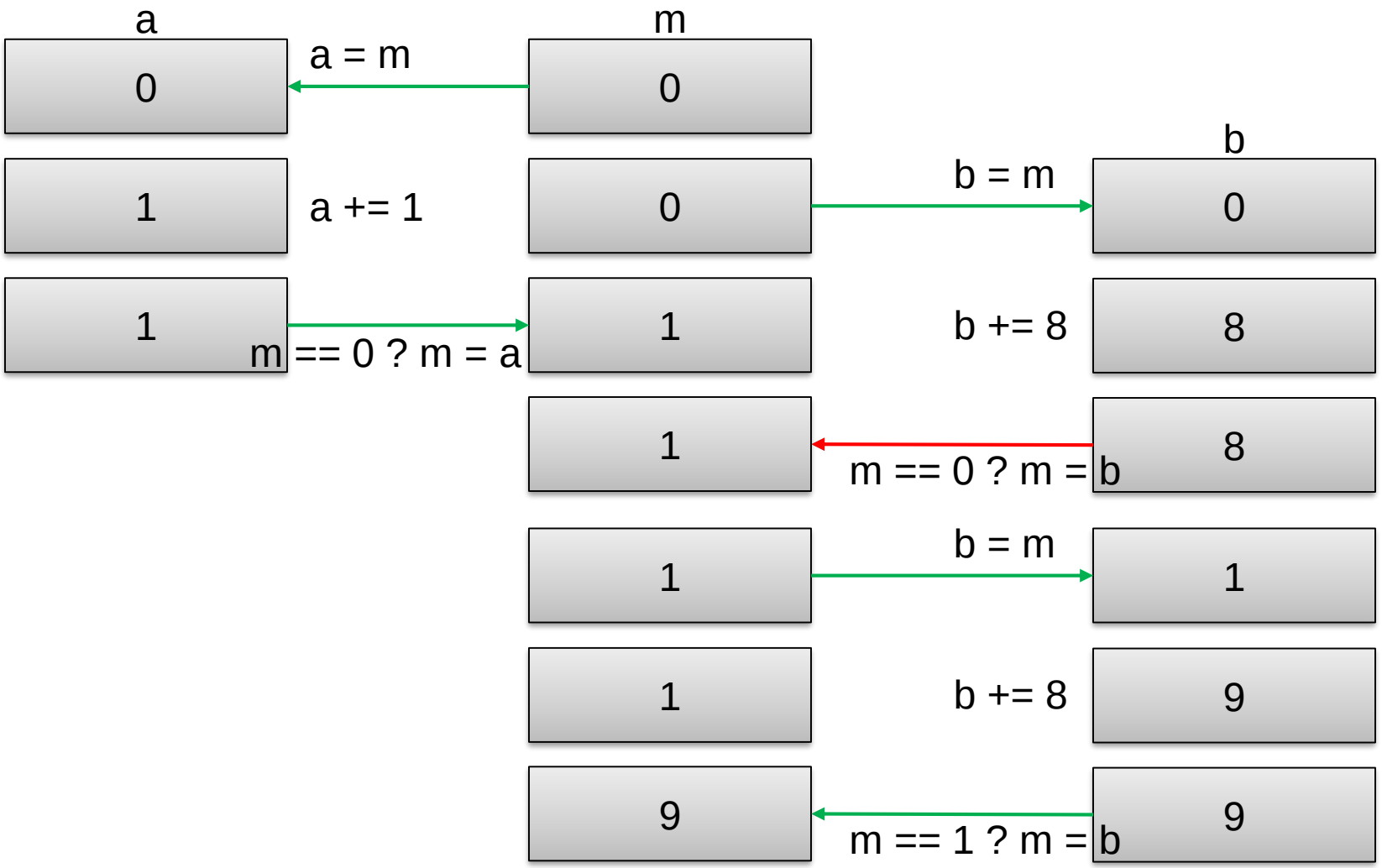
- Semaphores, mutexes, and monitors may block (suspend) the thread
- The thread is locked in waiting
- The thread now depends on other threads:
  - not being locked in turn
  - not being delayed
  - being bug-free

Lock-free code

- Lock-free code, non-blocking code

```
while true {

  previousState = getCurrentState()

  nextState = previousState + operation

  if atomicCompareAndSet(previousState, nextState)
       break

  // else try again
}
```

Lock-free code

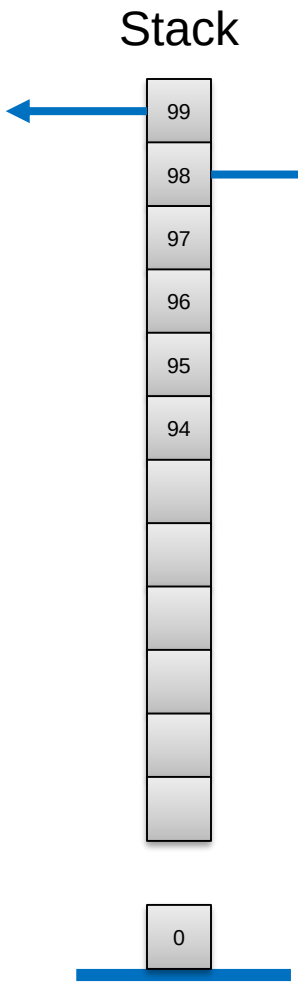


Lock-free code

- Complicated and inefficient algorithms
- The ABA problem:
  - thread X reads state A
  - while X is suspended, thread Y removes A, puts on B, removes B, puts back a different A
  - thread X updates since A==A, but they are in fact not the same A

Example ABALinkedList

Thread 1
Pop the top number off
and save it

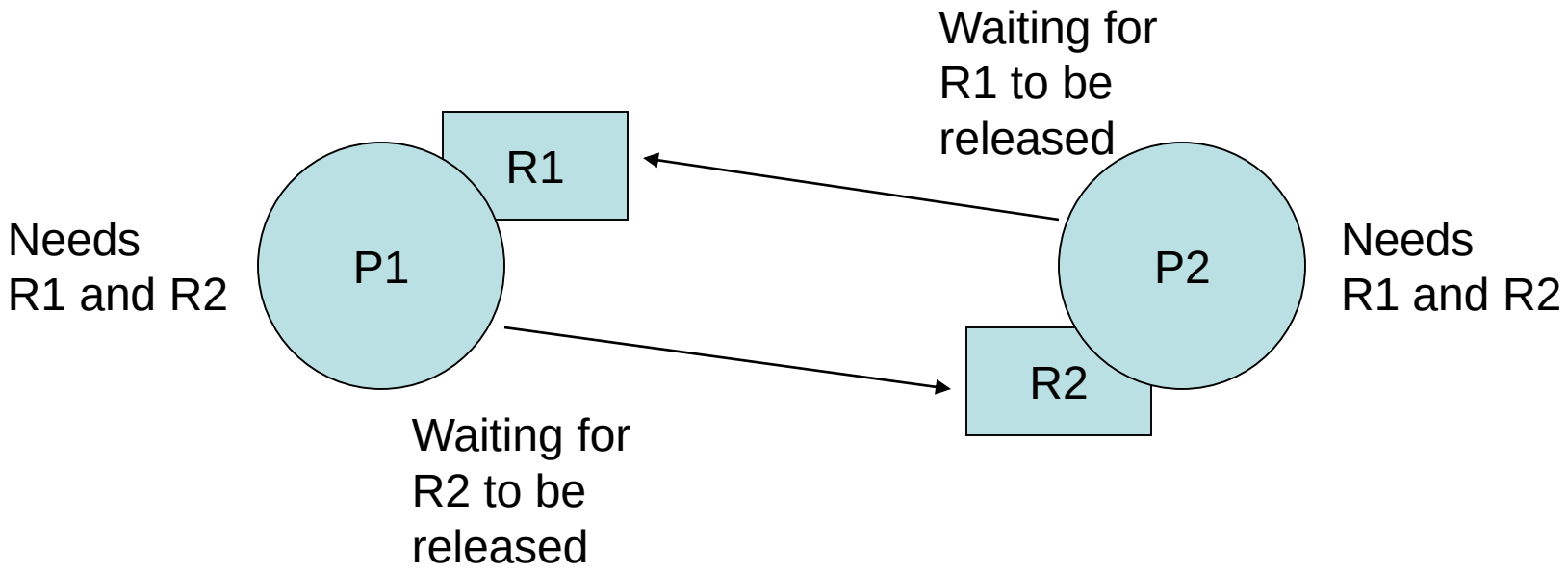


Thread 2
Pop top – 1 off and save it:
a = pop()
b = pop()
push(a)
save b

Consistency assumption:
When the stack is empty,
every number originally on the stack
is found exactly once in either
thread 1 or in thread 2.

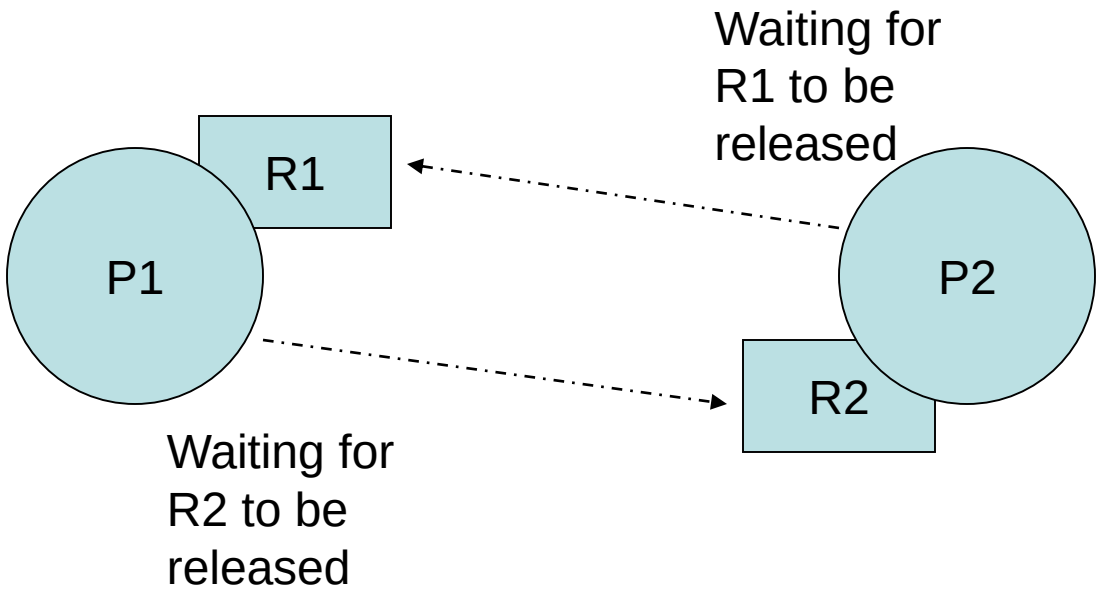
Deadlock

- Processes wait forever for all required resources to be released



Livelock

- Processes attempt to break the deadlock by time-out and release, but no-one wins



Famous race conditions

- The Therac-25 radiation therapy machine (1987)
  - Safety flag was not set but incremented (wrap) + fast manual input outran proper radiation shielding
- Northeast US electrical blackout (2003)
  - Alert message systems broke down; human operators unaware of critical situation – simultaneous write access: alert sys loop.
- Citibank withdrawal fraud (2012)
  - Multiple withdrawals from ATMs within a 60-second window did not accumulate debit on the account

Further fun reading

- Dijkstra, The Dining Philosophers problem
- Chandy/Misra solution to DPh problem
- Producer-consumer problem
- Sleeping barber problem
- Readers-writers problem
- Cigarette smokers problem
- ABA problem

End