LECTURE 24 · CSE 333 · Summer 2026

To Lock or not to Lock

Discussion: Last lecture this summer (for me!)

Reminders

Assessments:

- Exercise 9 due Thursday at 11:59pm
- Homework 4 due Thursday at 11:59pm

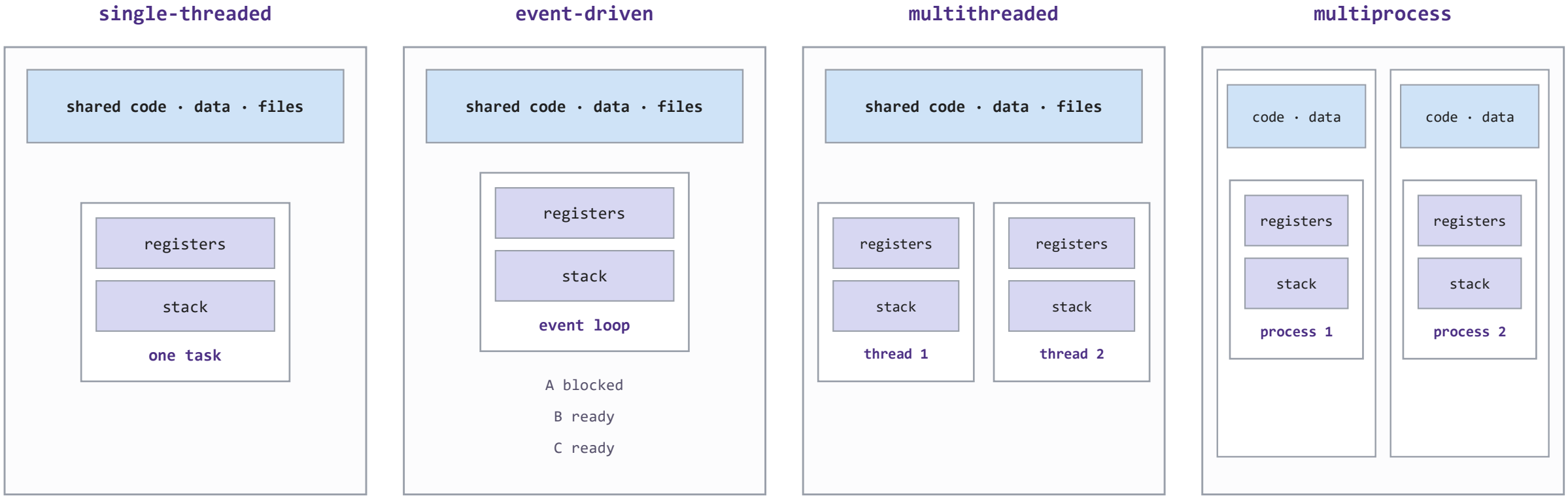
General:

- Today: Fun Lecture
- Thursday:
  - Section is extra office hours!
- Friday:
  - Guest Lecture: CUDA Programming
  - I'll be gone so no office hours

Review

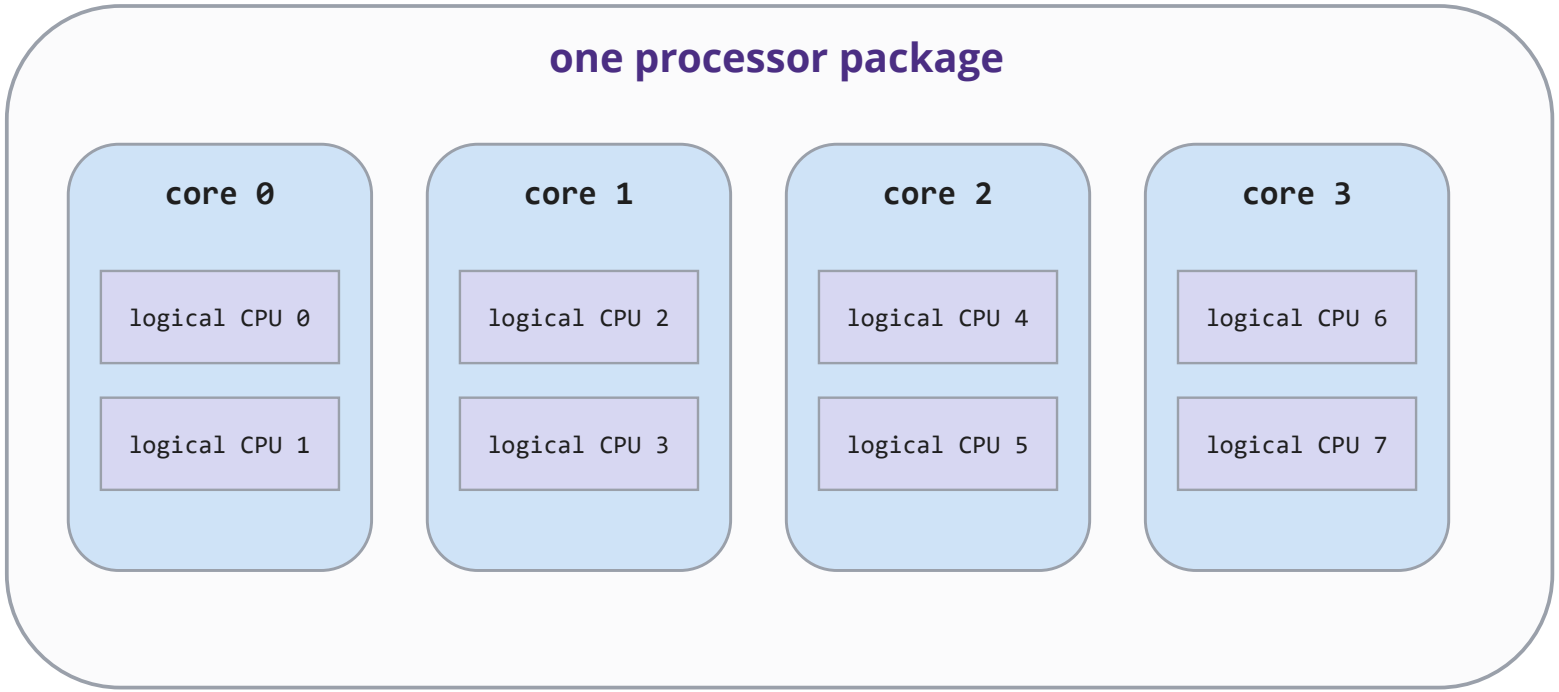
REVIEW

Execution contexts and memory



REVIEW

Packages, cores, and logical CPUs



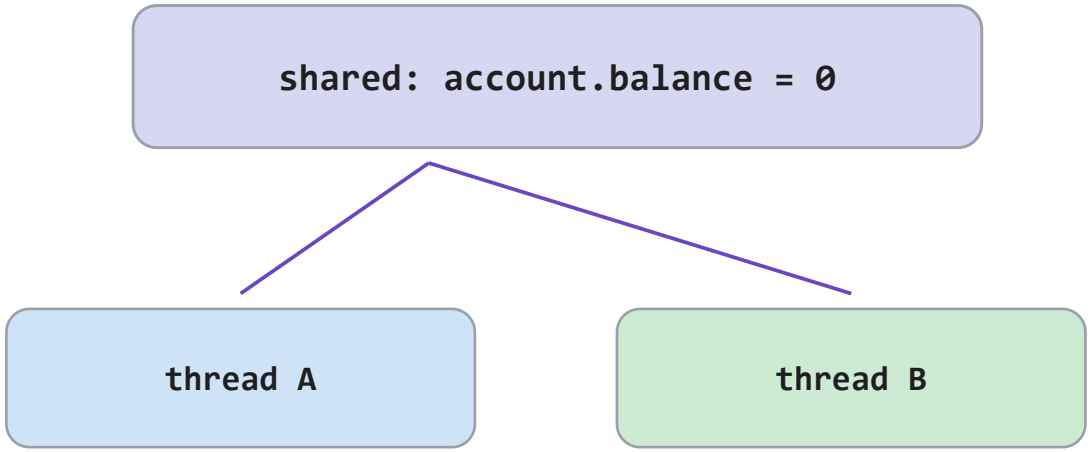
Definitions

- A machine may have one or more processor packages
- Each package contains one or more physical cores
- A core may expose one or more logical CPUs
- Linux schedules software threads onto logical CPUs

REVIEW

Multi-threading

```
1  typedef struct {
2    int balance;
3  } Account;
4
5  Account account = {.balance = 0};
```



Both threads can address the same object because pthreads share one process address space.

REVIEW

Not all operations are thread-safe

```
1  void Deposit(Account* account, int amount) {
2    account->balance += amount;
3  }
4
5  // thread A: Deposit(&account, 1);
6  // thread B: Deposit(&account, 1);
```

A C data race

- A and B access the same balance object
- Both operations include a write
- No synchronization orders those accesses
- The C program therefore has undefined behavior

REVIEW

Non-atomic operations

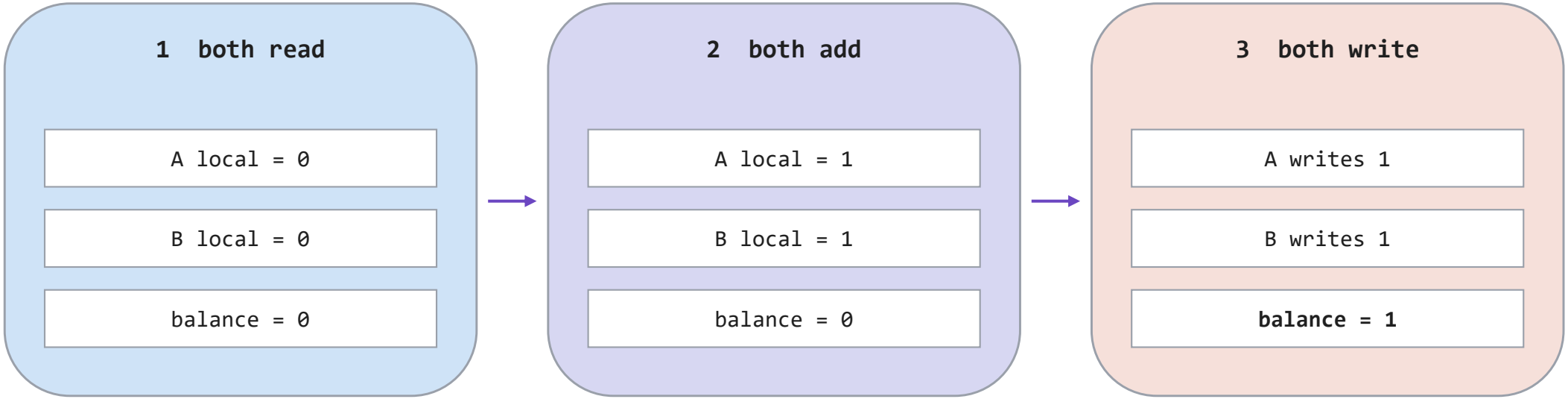
```
1  account->balance += 1;
2
3  // conceptual operations
4  local = account->balance;  // read shared memory
5  local = local + 1;         // modify private value
6  account->balance = local;  // write shared memory
```

One statement is not one atomic action

- balance is one shared memory location
- local represents private register state
- A thread may stop between these operations
- Another core may access balance simultaneously

REVIEW

Race condition



Both threads computed 1 from the same old value. B's store replaces A's store instead of adding another deposit.

REVIEW

One solution is to use a mutex

```
1  typedef struct {
2    pthread_mutex_t mutex;
3    int balance;
4  } Account;
5
6  int Account_Init(Account* account) {
7    account->balance = 0;
8    return pthread_mutex_init(&account->mutex, NULL);
9  }
10
11 int Account_Destroy(Account* account) {
12   return pthread_mutex_destroy(&account->mutex);
13 }
```

One object, one locking rule

- ❖ Initialize before publishing Account to workers
- ❖ The mutex and balance share one object lifetime
- ❖ Destroy after all worker threads have stopped
- ❖ Destroy only while the mutex is unlocked

Example: mutex

```
1  int Account_Deposit(Account* account, int amount) {
2    int rc = pthread_mutex_lock(&account->mutex);
3    if (rc != 0) return rc;
4
5    account->balance += amount;
6
7    return pthread_mutex_unlock(&account->mutex);
8  }
```

The data race is removed

- Lock before reading or writing balance
- Only the mutex owner executes the update
- Unlock after the invariant is restored
- Every balance access follows this mutex rule

REVIEW

fork()

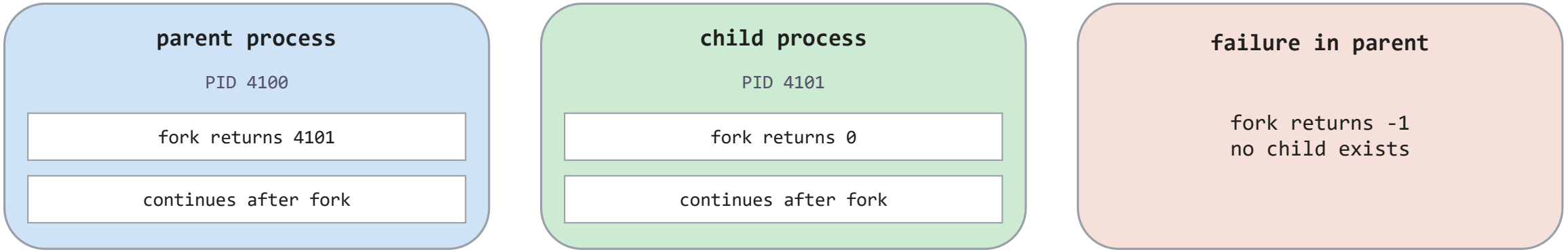
```
1  int x = 0;
2  pid_t pid = fork();
3  if (pid == 0) {
4      x = 1;
5  } else {
6      x = 2;
7  }
8  printf("x = %d\n", x);
```

Parallel executions

- The child follows the pid == 0 branch and prints 1
- The parent follows the other branch and prints 2
- Each process changes its own x
- The scheduler determines which line prints first

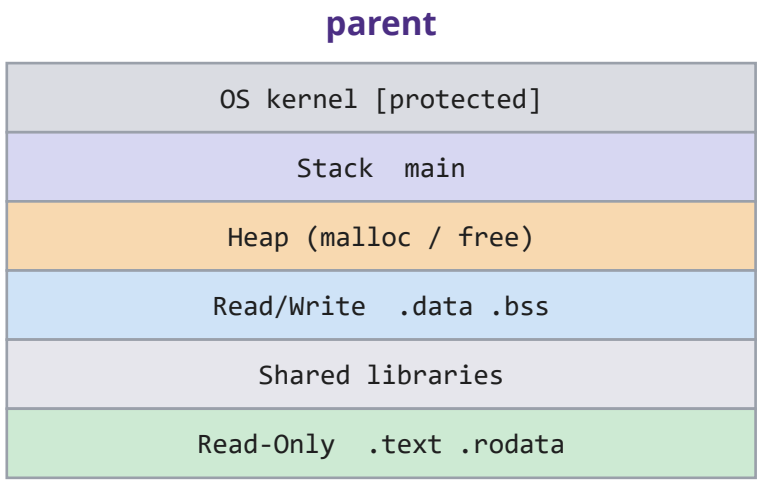
fork has three return cases

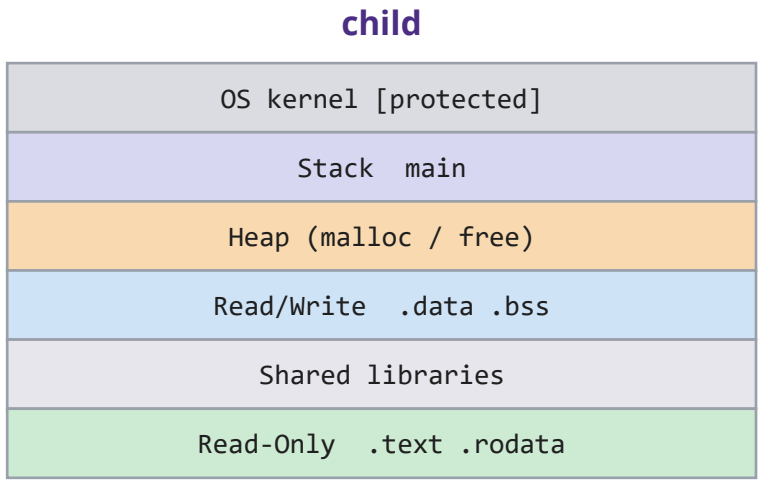
C17 · POSIX process APIs



The return value identifies which execution path is running. It does not determine which process runs first.

Generates two address spaces





The contrast

- Two copies of x
- Neither write is visible
- No lock needed
- No way to share

Lock-free

Two calls can lose one update

```
1  uint64_t total = 40;
2
3  void CountOne() {
4    total++;
5  }
```

One possible timing

- A reads 40
- B reads 40
- A writes 41
- B writes 41

How could we fix it?

Brainstorm ways to make sure both calls are counted.

```
1  void CountOne() {
2    // ? several threads call this
3    total++;
4  }
```

Think about who writes total, and when.

Predict first: Write an answer down before we move on.

The real problem is shared mutable state

`shared`

Several threads can reach total.

`mutable`

Those threads can change total.

`overlap`

Their accesses can happen together.

If any one of these disappears, this particular race disappears.

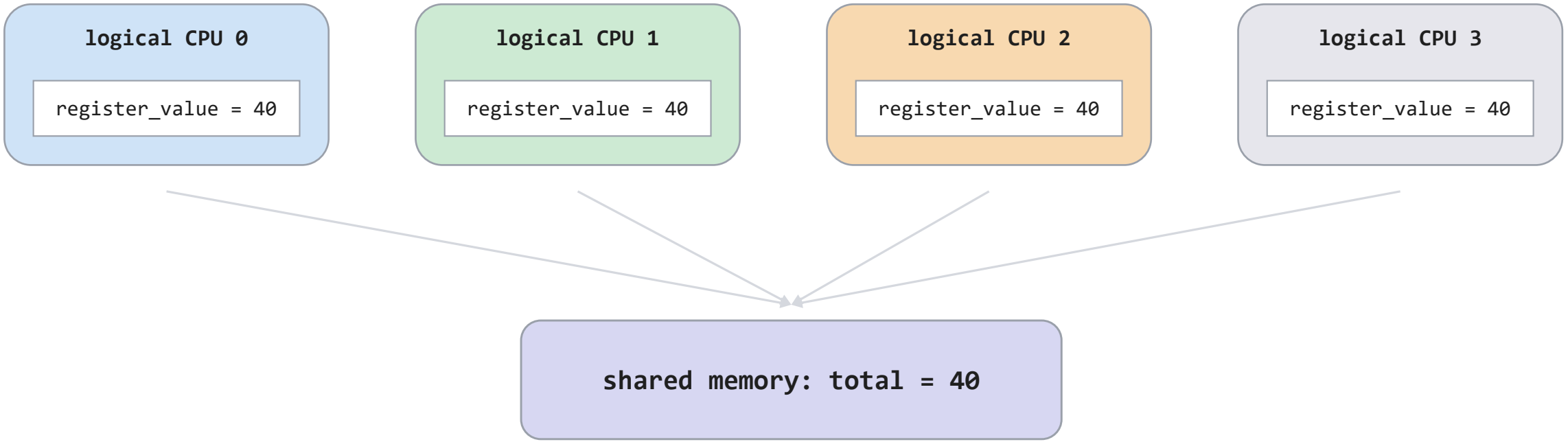
total++ uses a private local value

```
1  total++;
2
3  // conceptual operations from the threads lecture
4  register_value = total;
5  register_value = register_value + 1;
6  total = register_value;
```

Where the values live

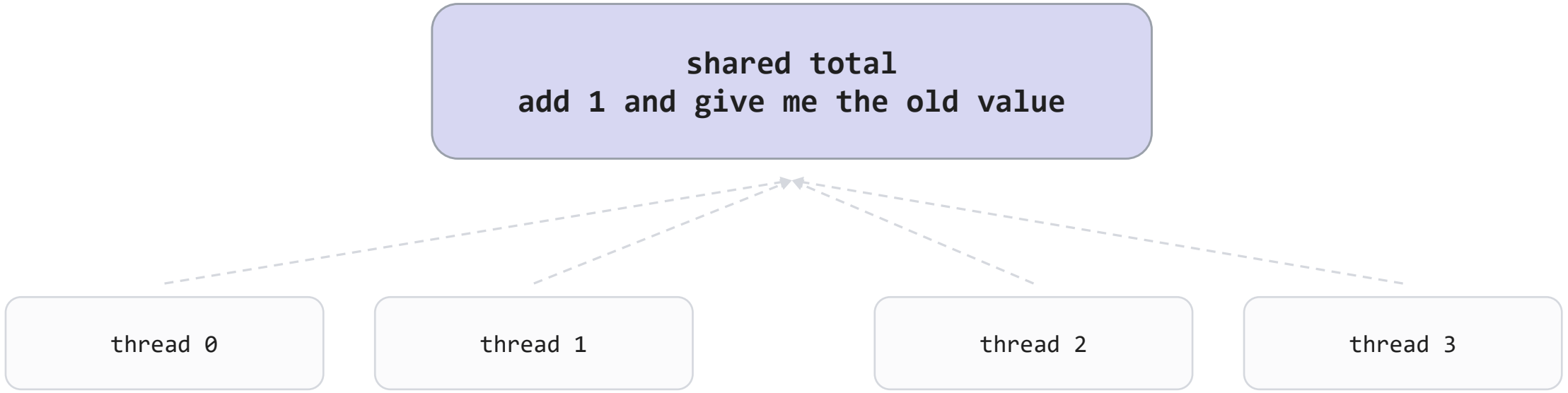
- total is one shared memory location
- register_value is private execution state
- another thread can run between these steps

Four threads can hold four old values



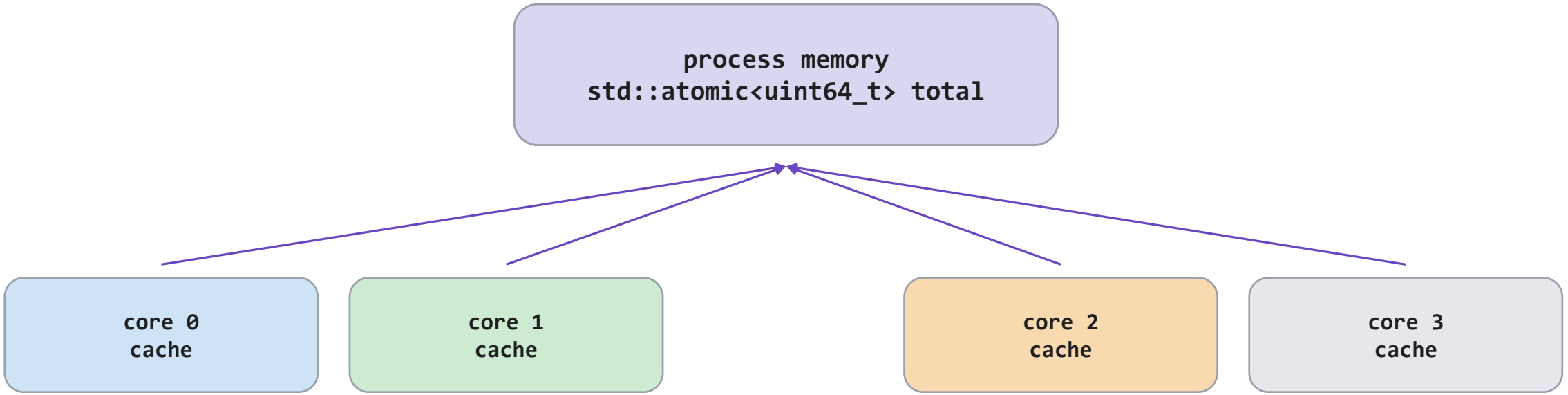
Registers belong to the running logical CPU. The shared object does not.

Could the add happen as one shared operation?

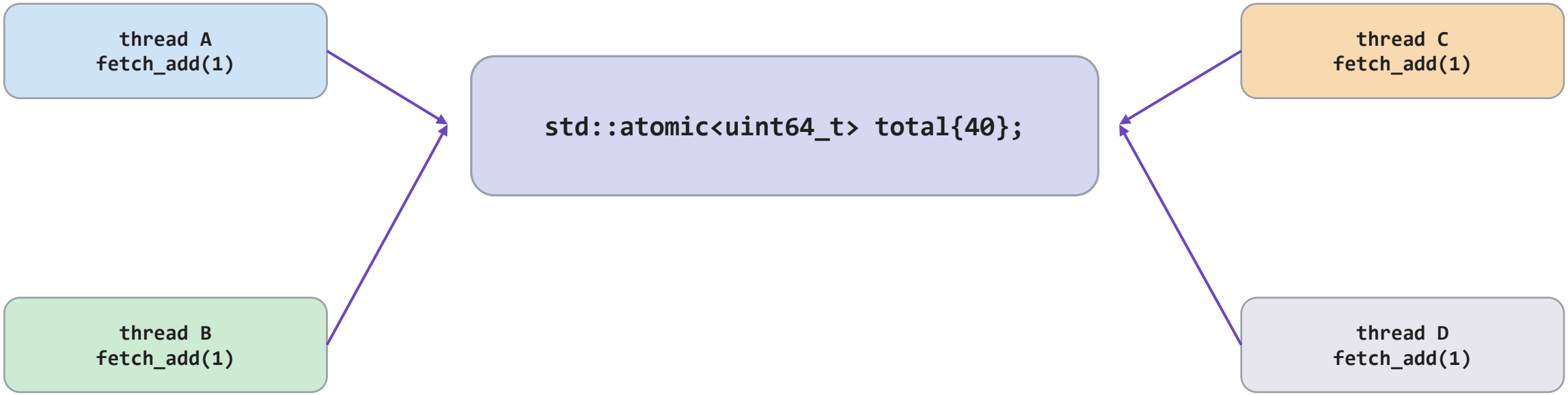


Predict first: What would the CPU need to coordinate?

An atomic is ordinary shared memory



Four threads use the same atomic object



The threads share the object, not one permanent cache copy.

fetch_add updates without a guessed new value

```
1  std::atomic<uint64_t> total{40};
2
3  void CountOne() {
4    total.fetch_add(1);
5  }
6
7  uint64_t old = total.fetch_add(5);
```

Read the name literally

- fetch the old value
- add the argument
- publish the new value as one atomic operation
- return the old value to the caller

Four atomic additions produce four results

caller	operation	value returned	total afterward
A	fetch_add(1)	40	41
C	fetch_add(1)	41	42
B	fetch_add(1)	42	43
D	fetch_add(1)	43	44

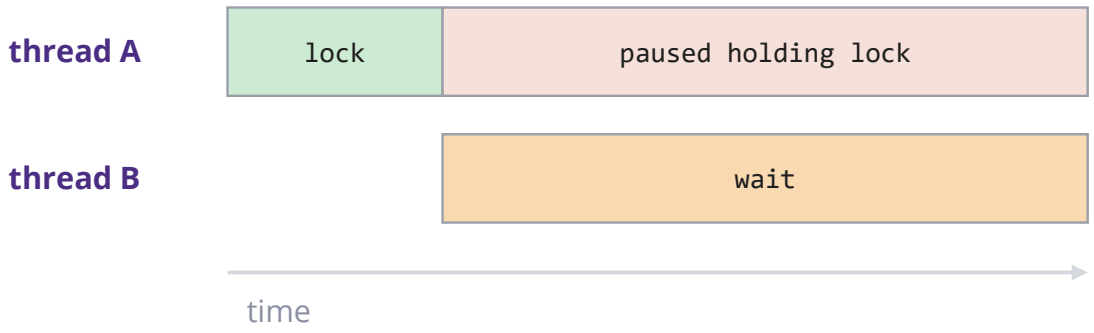
The order may change. The four distinct updates do not disappear.

The atomic toolbox

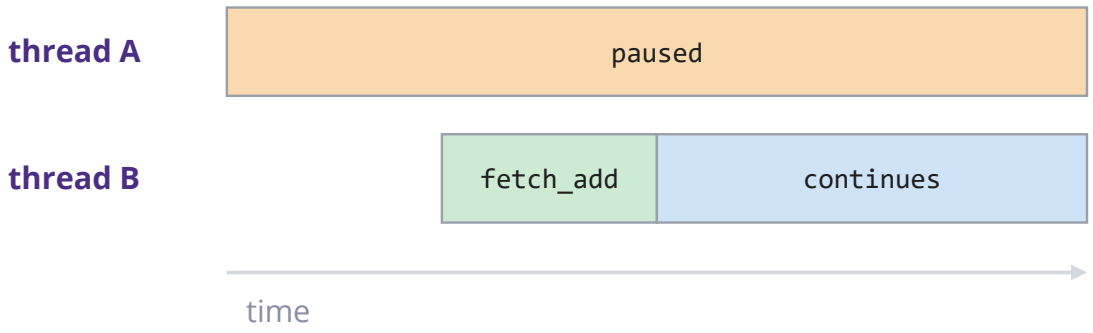
operation	what it does	what the caller gets
load()	read the current value	returns the value
store(value)	replace the value	returns nothing
exchange(value)	replace unconditionally	returns the old value
fetch_add / fetch_sub	change a number or pointer	returns the old value
fetch_and / fetch_or / fetch_xor	change integer bits	returns the old value
++, --, +=, -=, &=, |=, ^=	operator spellings	returns per operator
compare_exchange	replace only if unchanged	reports success

Choose the operation that matches the shared-state change.

What lock-free means here



mutex version

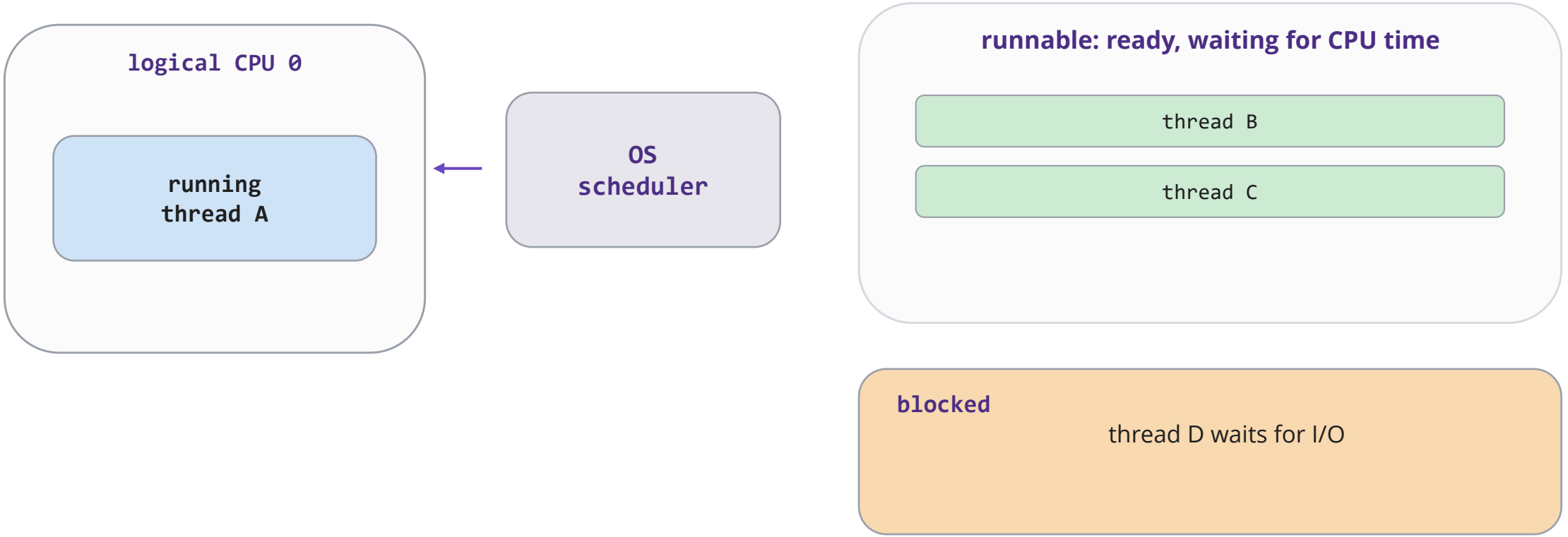


lock-free atomic version

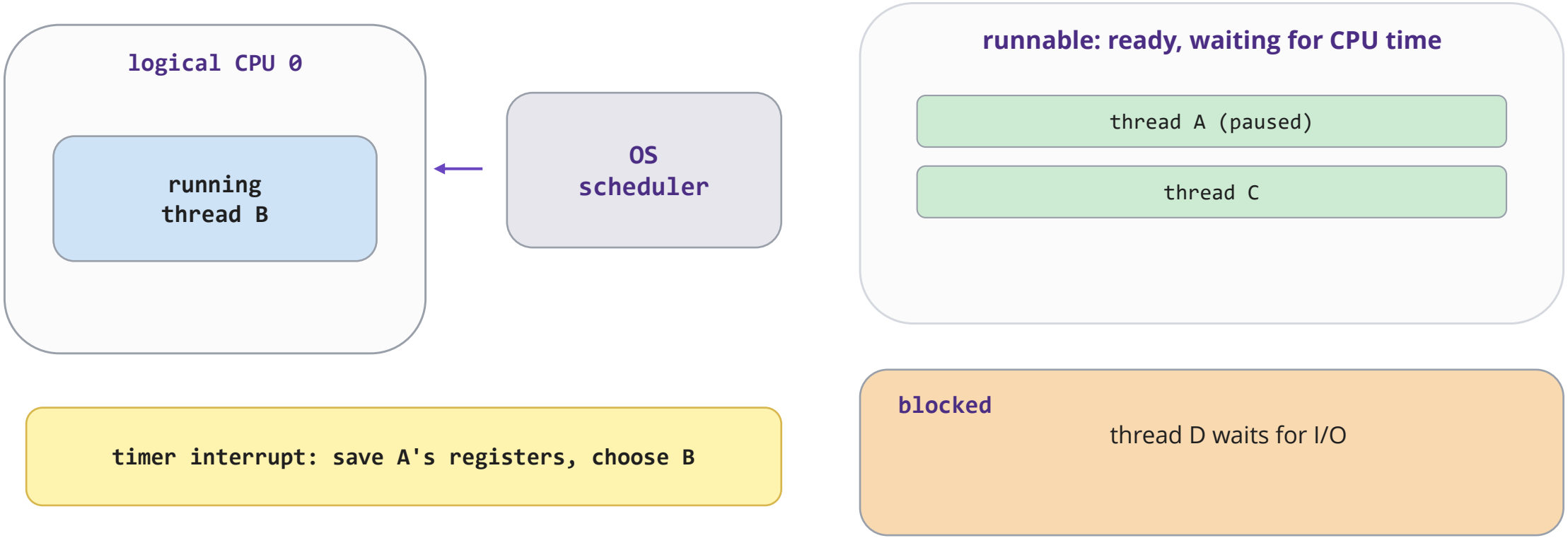
A paused caller does not own a lock that every other caller needs.

Step 1 of 3

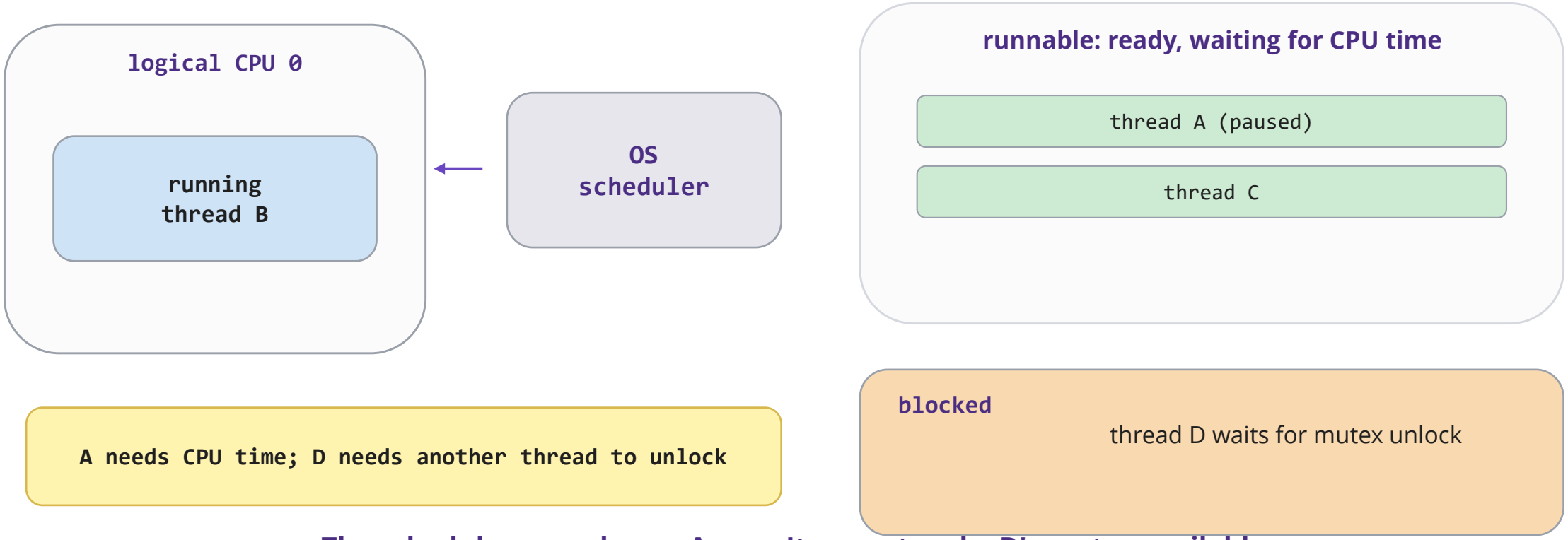
Running, runnable, and blocked



The OS can pause a running thread

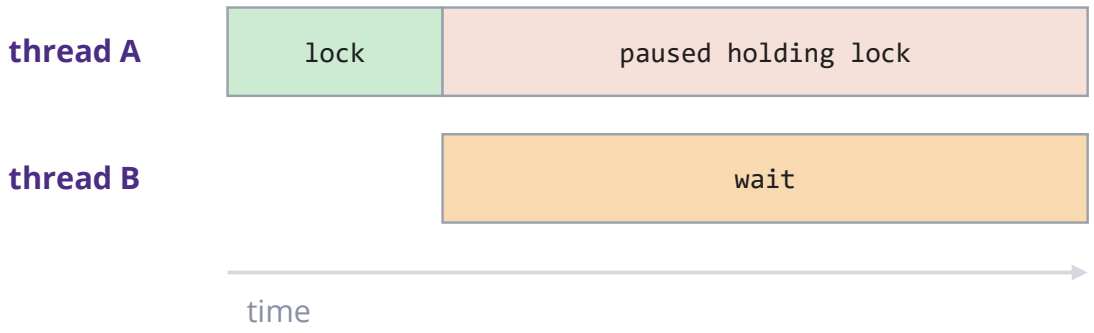


Paused and waiting are different

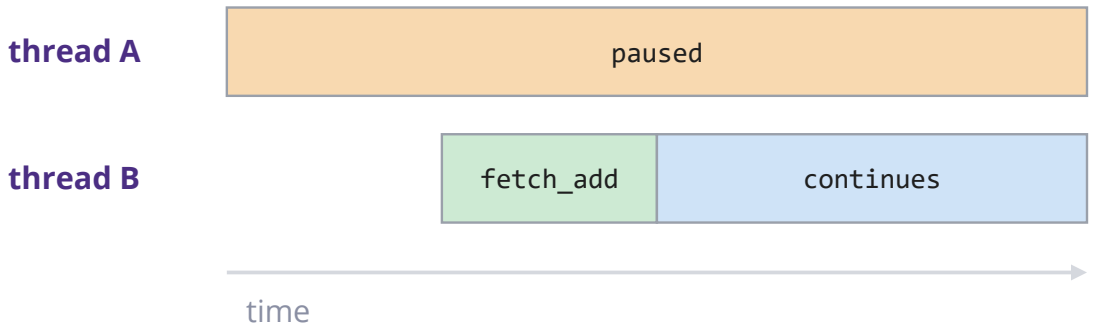


The scheduler may choose A now. It cannot make D's mutex available.

What lock-free means here



mutex version

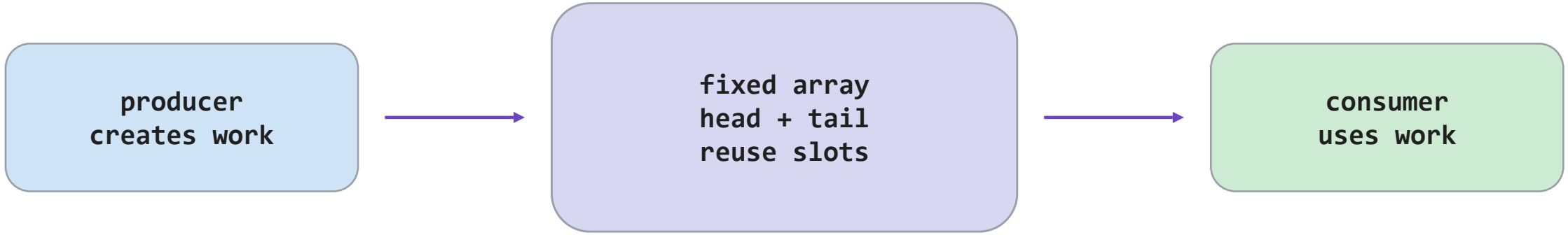


lock-free atomic version

A paused caller does not own a lock that every other caller needs.

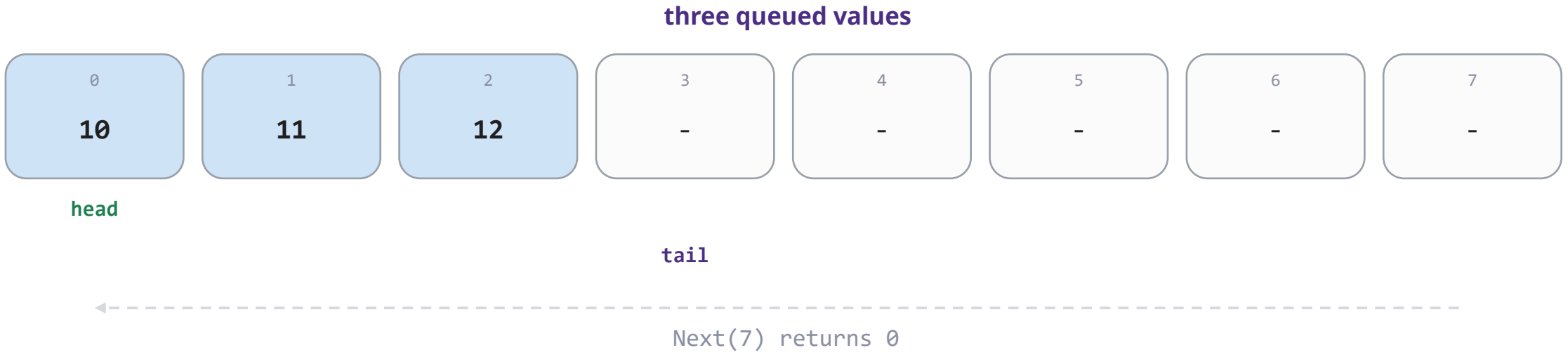
A complex example

A ring buffer passes bounded work



Fixed capacity avoids allocation while the queue is in use.

A ring buffer reuses a fixed array



Two indices

- head: next value to remove
- tail: next free slot
- both wrap back to zero

Start with a single-threaded ring buffer

```
1  template <size_t Capacity>
2  class RingBuffer {
3   private:
4    std::array<int, Capacity> data_{};
5    size_t head_ = 0;
6    size_t tail_ = 0;
7
8    size_t Next(size_t index) const {
9      return (index + 1) % Capacity;
10   }
11 };
```

State

- data_ stores the values
- head_ selects the next pop
- tail_ selects the next push
- Next wraps an index

Single-threaded TryPush

```
1  bool TryPush(int value) {
2    const size_t next = Next(tail_);
3    if (next == head_) {
4      return false;  // full
5    }
6    data_[tail_] = value;
7    tail_ = next;
8    return true;
9  }
```

Push order

- compute the next tail
- stop if it would reach head
- write the value
- move tail

Step 1 of 3

TryPush(99)

head = 5, tail = 7

1 check

`next = 0;  0 != head`

2 write slot

`data[7] = 99`

3 move tail

`tail = 0`

Step 2 of 3

TryPush(99)

head = 5, tail = 7

1 check

`next = 0;  0 != head`

2 write slot

`data[7] = 99`

3 move tail

`tail = 0`

Step 3 of 3

TryPush(99)

head = 5, tail = 0

1 check

`next = 0;  0 != head`

2 write slot

`data[7] = 99`

3 move tail

`tail = 0`

Single-threaded TryPop

```
1  bool TryPop(int* value) {
2    if (head_ == tail_) {
3      return false;  // empty
4    }
5    *value = data_[head_];
6    head_ = Next(head_);
7    return true;
8  }
```

Pop order

- stop when head equals tail
- copy the value out
- move head
- the old slot becomes reusable

Step 1 of 3

TryPop(&value)

head = 5, tail = 0

1 check

`head != tail`

2 copy slot

`*value = data[5]`

3 move head

`head = 6`

Step 2 of 3

TryPop(&value)

head = 5, tail = 0

0	1	2	3	4	5	6	7
-	-	-	-	-	50	60	99

head

tail

1 check

`head != tail`

2 copy slot

`*value = data[5]`

3 move head

`head = 6`

Step 3 of 3

TryPop(&value)

head = 6, tail = 0

1 check

`head != tail`

2 copy slot

`*value = data[5]`

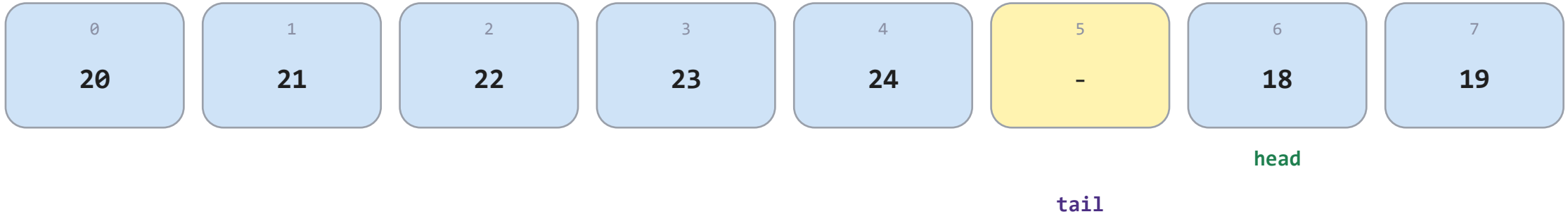
3 move head

`head = 6`

One empty slot separates empty from full

empty: head == tail

full: Next(tail) == head



What changes with two threads?

One producer calls TryPush while one consumer calls TryPop.

```
1  // producer changes tail_ and data_[tail_]
2  // consumer changes head_ and reads data_[head_]
3
4  // ? Which state is shared?
```

Mark the readers and writers for each field.

Predict first: Write an answer down before we move on.

The mutex protects the whole ring state

```
template <size_t Capacity>
class LockedRingBuffer {
 private:
  std::array<int, Capacity> data_{};
  size_t head_ = 0;
  size_t tail_ = 0;
  std::mutex mutex_;
};
```

One simple rule

- lock before reading or changing head_
- lock before reading or changing tail_
- lock before accessing an occupied slot

Mutex-protected TryPush

```
1  bool TryPush(int value) {
2    std::lock_guard<std::mutex> guard(mutex_);
3    const size_t next = Next(tail_);
4    if (next == head_) {
5      return false;
6    }
7    data_[tail_] = value;
8    tail_ = next;
9    return true;
10 }
```

What changed

- the ring algorithm stayed the same
- lock_guard covers every shared access
- destruction of guard unlocks on every return

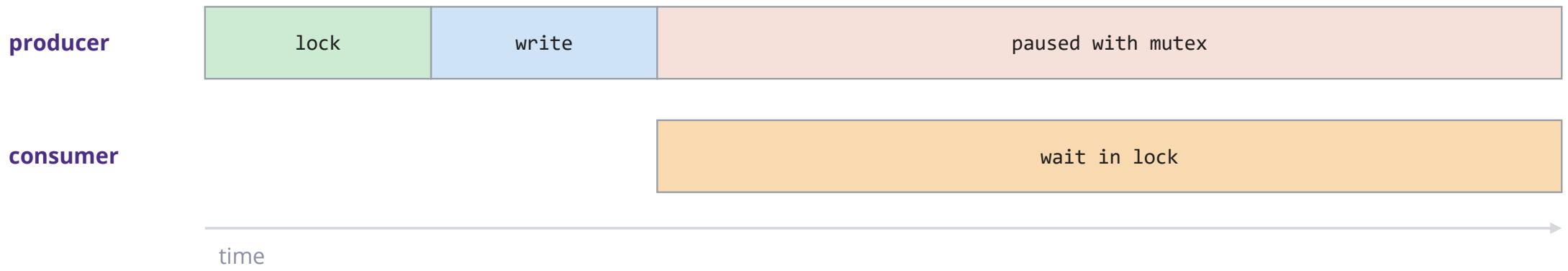
Mutex-protected TryPop

```
1  bool TryPop(int* value) {
2    std::lock_guard<std::mutex> guard(mutex_);
3    if (head_ == tail_) {
4      return false;
5    }
6    *value = data_[head_];
7    head_ = Next(head_);
8    return true;
9  }
```

Same rule

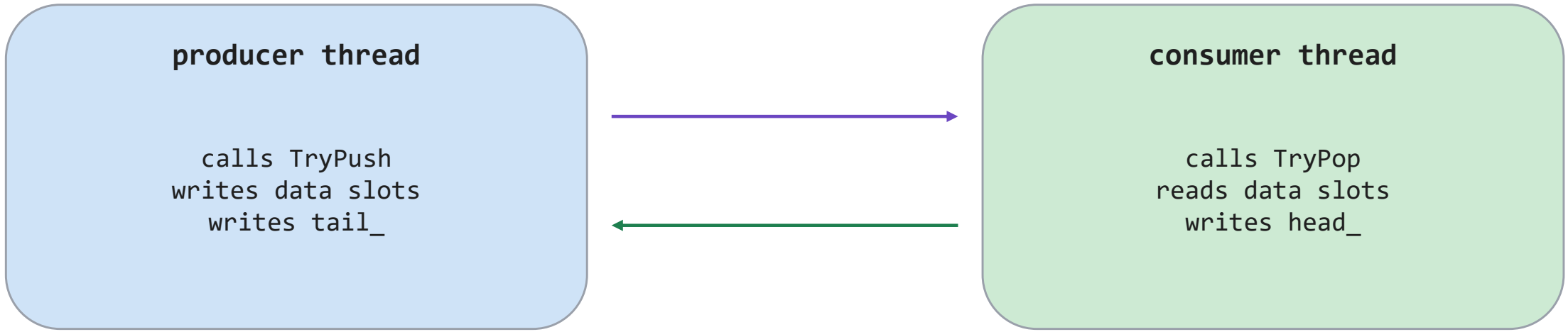
- pop uses the same mutex as push
- producer and consumer cannot overlap ring access
- the result is straightforward to reason about

A paused producer can block the consumer



The consumer cannot inspect a different slot while the producer owns the mutex.

First specify the contract



SPSC is part of the type's contract, not a suggestion.

Each index has one writer

shared state	who writes	who reads
tail_	producer	consumer checks for empty
head_	consumer	producer checks for full
data_[slot]	producer before publish	consumer after observing publish

Single writer does not mean unshared: the other thread still reads the index.

The lock-free ring has atomic indices

```
1  template <size_t Capacity>
2  class SpscRingBuffer {
3   private:
4     std::array<int, Capacity> data_{};
5     std::atomic<size_t> head_{0};
6     std::atomic<size_t> tail_{0};
7  };
```

What stays ordinary

- the fixed array still stores ordinary int values
- head_ and tail_ are shared atomic objects
- there is no mutex field
- the SPSC contract still limits the callers

Each operation starts with local snapshots

producer

```
1  size_t tail = tail_.load();
2  size_t head = head_.load();
3  size_t next = Next(tail);
```

consumer

```
1  size_t head = head_.load();
2  size_t tail = tail_.load();
3  size_t next = Next(head);
```

Why local copies help

- the method reasons from one observed pair of indices
- ordinary arithmetic uses private local variables
- only publication back to the shared index is atomic

Slot access comes before index publication

Moving tail announces a ready slot. Moving head announces a reusable slot.

CAS moves an index only if it is unchanged

In SPSC, only one thread writes each index, so this CAS should succeed.

SPSC TryPush with CAS

```
1  bool TryPush(int value) {
2    size_t tail = tail_.load();
3    const size_t next = Next(tail);
4    if (next == head_.load()) {
5      return false;
6    }
7    data_[tail] = value;
8    return tail_.compare_exchange_strong(tail, next);
9  }
```

Publish in this order

- read the current indices
- stop if the ring is full
- write the slot
- CAS tail to publish the slot

SPSC TryPop with CAS

```
1  bool TryPop(int* value) {
2    size_t head = head_.load();
3    if (head == tail_.load()) {
4      return false;
5    }
6    *value = data_[head];
7    const size_t next = Next(head);
8    return head_.compare_exchange_strong(head, next);
9  }
```

Release in this order

- read the current indices
- stop if the ring is empty
- copy the slot
- CAS head to release the slot

Atomic behavior is not always lock-free

```
1  bool IsLockFree() const {
2    return head_.is_lock_free() &&
3           tail_.is_lock_free();
4  }
5
6  // The progress claim depends on this result.
```

Before claiming lock-free

- the algorithm must contain no blocking lock
- the atomic index operations must be lock-free here
- the SPSC caller contract must be followed
- performance still requires measurement

Producer and consumer can overlap

step	producer	consumer	ring
1	read tail = 5	read head = 2	slots 2, 3, 4 occupied
2	write data[5] = 99	read data[2]	different slots
3	CAS tail 5 -> 6	CAS head 2 -> 3	publish and release
4	continue	continue	neither owns a mutex

The contract keeps them on different occupied/free slots.

Reminders

Assessments:

- Exercise 9 due Thursday at 11:59pm
- Homework 4 due Thursday at 11:59pm

General:

- Today: Fun Lecture
- Thursday:
  - Section is extra office hours!
- Friday:
  - Guest Lecture: CUDA Programming
  - I'll be gone so no office hours

Questions?

Thanks for coming - see you at the final!