LECTURE 22 · CSE 333 · Summer 2026

More Parallelism

Discussion: What's the favorite thing you've learned in CSE 333?

Reminders

Assessments:

- Exercise 8 due Friday at 11:59pm
- Homework 4 due next Thursday at 11:59pm

General:

- Final Exam
  - CSE2 G10 from 1:30-3:30pm on Tuesday, August 18
  - Makeup exam: 5-7pm on different days. Emails have been sent out for confirmation.
- All work due on last Friday of the course.

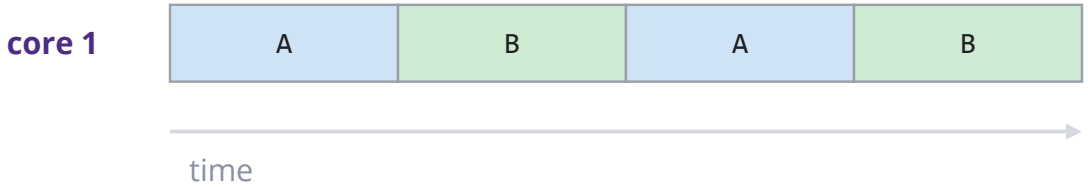
Review

REVIEW

Concurrency and parallelism

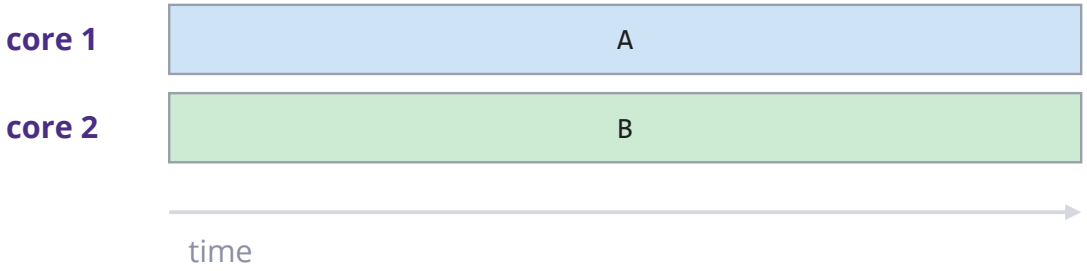
`concurrency`

Several tasks make progress during the same interval.



`parallelism`

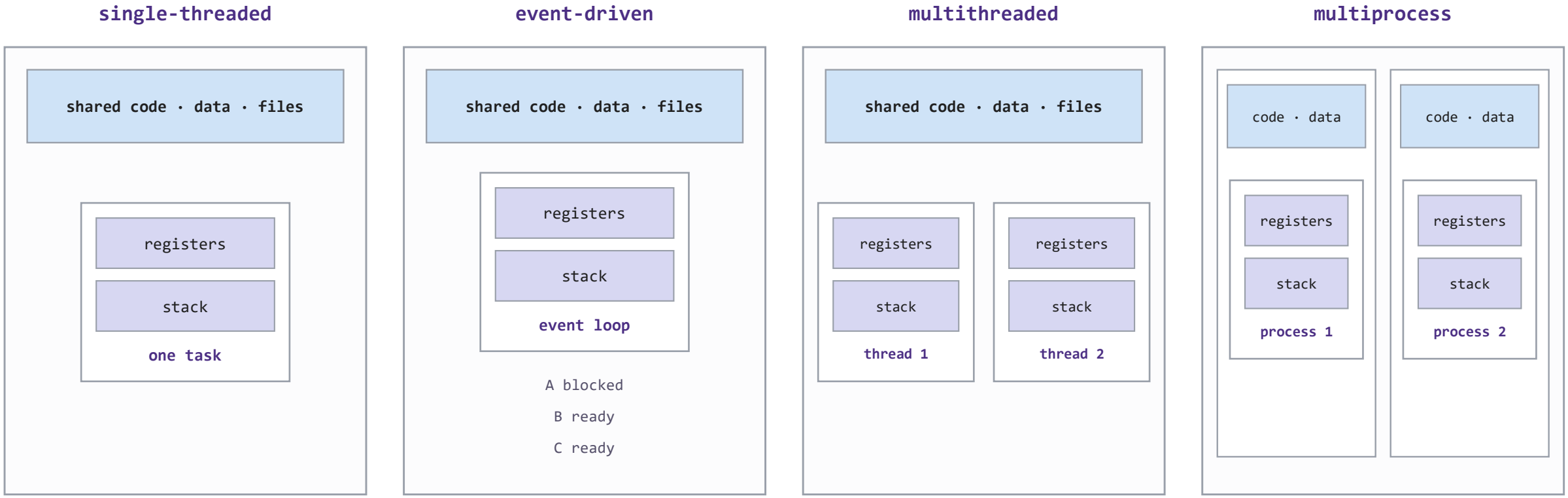
Several instructions execute at the same instant.



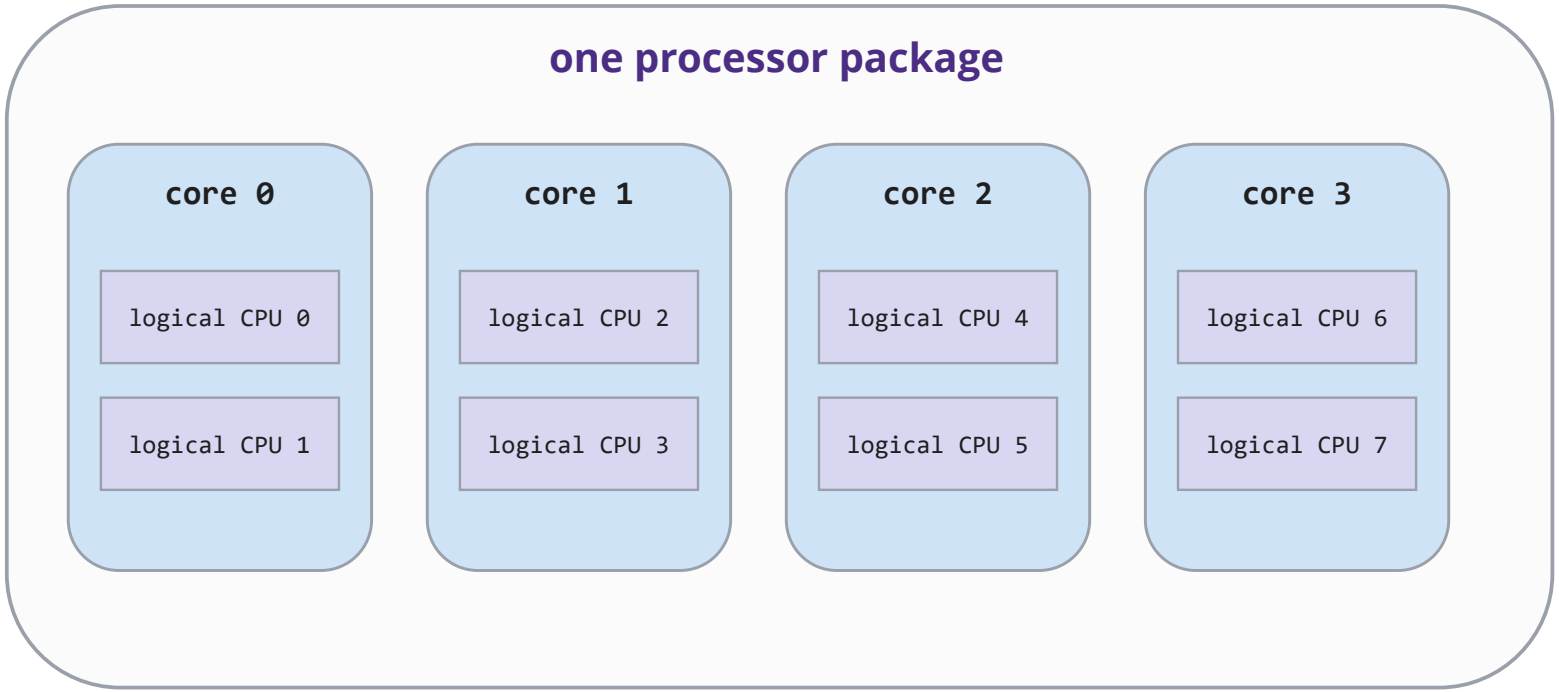
Concurrency can improve an I/O-bound server on one core. Parallelism needs multiple cores and helps CPU-bound work.

REVIEW

Execution contexts and memory



Packages, cores, and logical CPUs

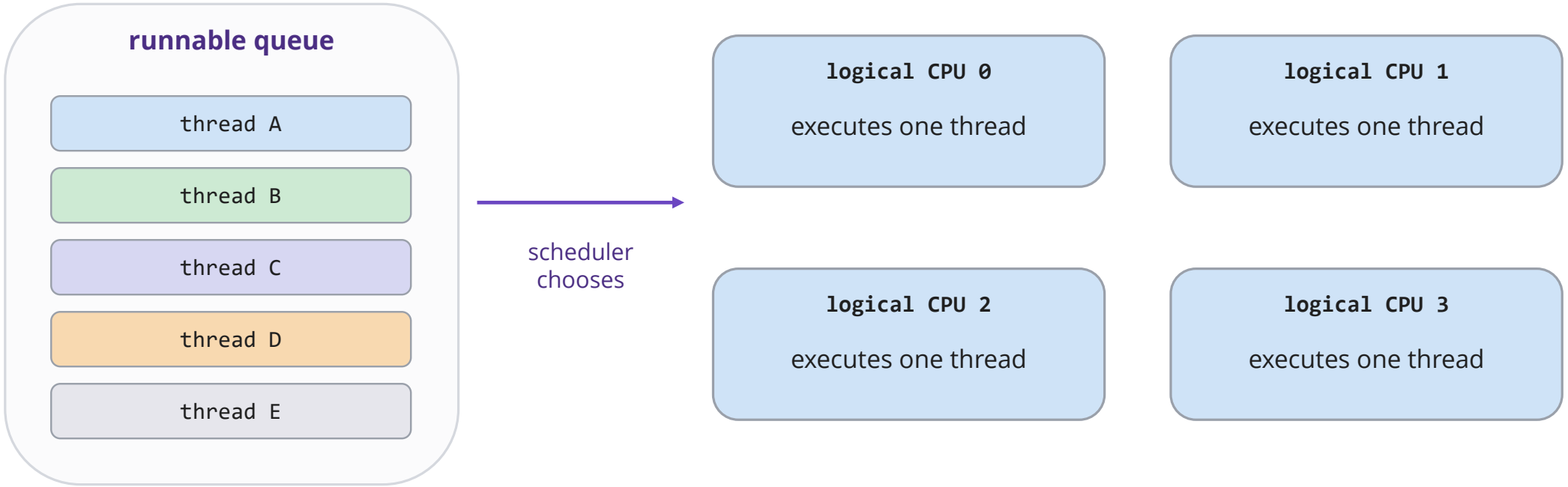


Definitions

- 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

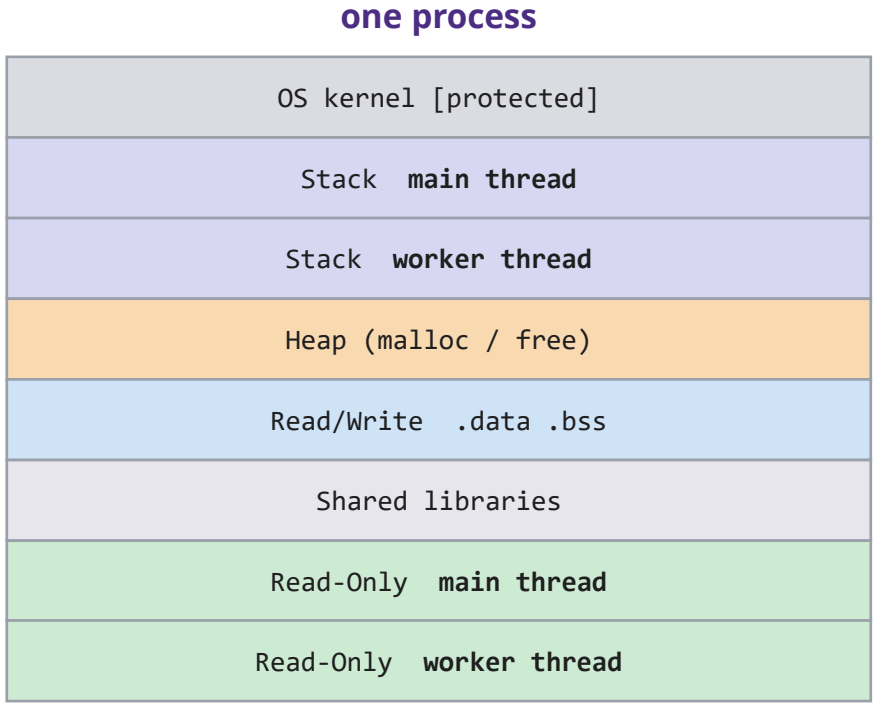
REVIEW

The OS schedules runnable threads



A timer interrupt can return a running thread to the queue. A blocked thread leaves the queue until its event completes.

After pthread_create



What changed?

- The worker receives its own program counter, registers, and stack pointer, and stack!
- No second heap or global-data segment is created
- Both threads can reach the same heap objects and globals
- Both use the process's open file descriptors

Version 1: Single-threaded

```c
#include <stdio.h>
#include <stdlib.h>

void PrintOneToFive(void) {
  for (int value = 1; value <= 5; value++) {
    printf("%d\n", value);
  }
}

int main(void) {
  PrintOneToFive();
  return EXIT_SUCCESS;
}
```

One call on a singlethread

- main runs on the process's initial thread
- PrintOneToFive is an ordinary function call
- value lives in a frame on the main thread's stack
- The call returns before main continues

Version 2: Two-threaded

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* PrintOneToFive(void* unused) {
  for (int value = 1; value <= 5; value++) {
    printf("%d\n", value);
  }
  return NULL;
}

int main(void) {
  pthread_t thread;
  int rc = pthread_create(&thread, NULL, PrintOneToFive, NULL);
  if (rc != 0) return EXIT_FAILURE;
  rc = pthread_join(thread, NULL);
  if (rc != 0) return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

pthread_join

- Blocks the calling thread until this worker terminates
- Reclaims the worker's pthread bookkeeping
- NULL means main does not need the worker's result
- main returns only after the five lines are printed

Version #3: Multi-threaded (no-args)

```
1  pthread_t threads[5];
2  int created = 0;
3
4  for (int i = 0; i < 5; i++) {
5    int rc = pthread_create(&threads[created], NULL,
6                            PrintOneToFive, NULL);
7    if (rc != 0) break;
8    created++;
9  }
10
11 for (int i = 0; i < created; i++) {
12   pthread_join(threads[i], NULL);
13 }
```

Five no-argument workers

- ❖ All five workers run the complete 1-through-5 loop
- ❖ The first loop creates without waiting
- ❖ The second loop eventually reclaims every worker
- ❖ Their output lines may interleave in many orders

Version #4: Multi-threaded (with args)

```
1  pthread_t threads[5];
2  int created = 0;
3
4  for (int value = 1; value <= 5; value++) {
5    int* number = malloc(sizeof(*number));
6    if (number == NULL) break;
7    *number = value;
8    int rc = pthread_create(&threads[created], NULL, PrintNumber, number);
9    if (rc != 0) { free(number); break; }
10   created++;
11 }
12
13 for (int i = 0; i < created; i++) {
14   pthread_join(threads[i], NULL);
15 }
```

Main thread owns

- ❖ malloc creates a distinct int object each iteration
- ❖ The object remains alive after the loop iteration ends
- ❖ Successful create transfers ownership to the worker
- ❖ Failed create leaves ownership with main, which frees it

Synchronization

(with locks!)

The milk

The rule in the flat: if there is no milk, go and buy some.

```
1  if (!milk) {
2    buy milk
3  }
```

Living alone

- Check the fridge, see nothing, go to the shop
- Come back with one carton
- There is no concurrent roommate between check and purchase

The milk

The rule in the flat: if there is no milk, go and buy some.

```
1  if (!milk) {
2    buy milk
3  }
```

Living alone

- Check the fridge, see nothing, go to the shop
- Come back with one carton
- There is no concurrent roommate between check and purchase

Living with a roommate

- You check the fridge and leave
- They check the fridge and leave
- **Two cartons.** You both read the same state before either of you wrote to it.

The note

Does leaving a note on the fridge fix it?

```
1  if (!note) {
2    if (!milk) {
3      leave note;  buy milk;  remove note
4    }
5  }
```

A. yes, this is correct now

B. no, you could end up with no milk at all

C. no, you could still end up with two cartons

D. it works, but only if you are lucky

Predict first: Where can the other person be when you run line 1?

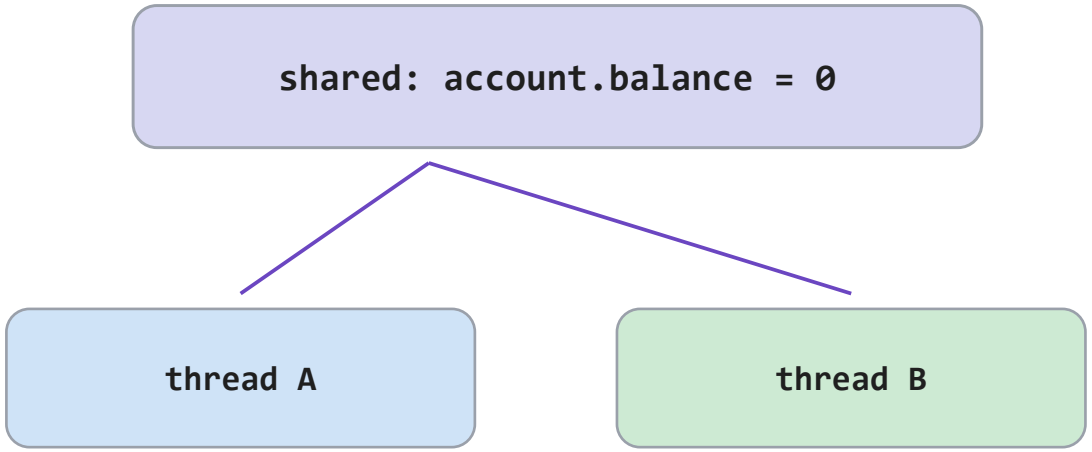
Checking and claiming

	you	your roommate	state
1	check note: none		no note, no milk
2		check note: none	no note, no milk
3	check milk: none		no note, no milk
4		check milk: none	no note, no milk
5	leave note, buy milk		milk
6		leave note, buy milk	two cartons

Checking the note and writing the note are two separate steps, so we solved the problem by adding a second copy of it.

Two threads share one Account

```
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.

The first Deposit is not 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

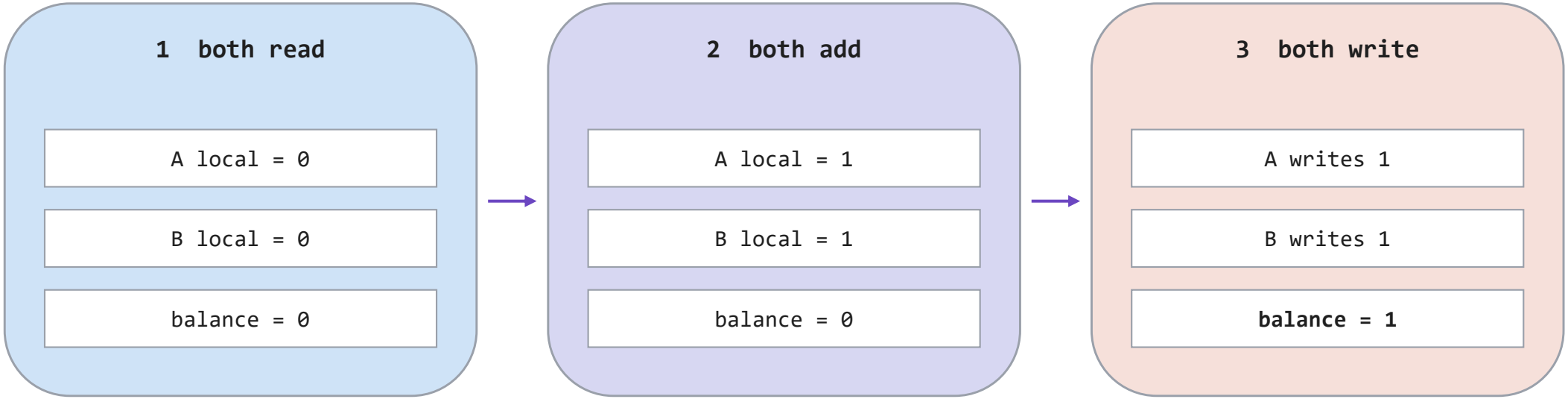
One deposit is a read-modify-write sequence

```
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

Two deposits can produce one update



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

The critical section is the complete deposit

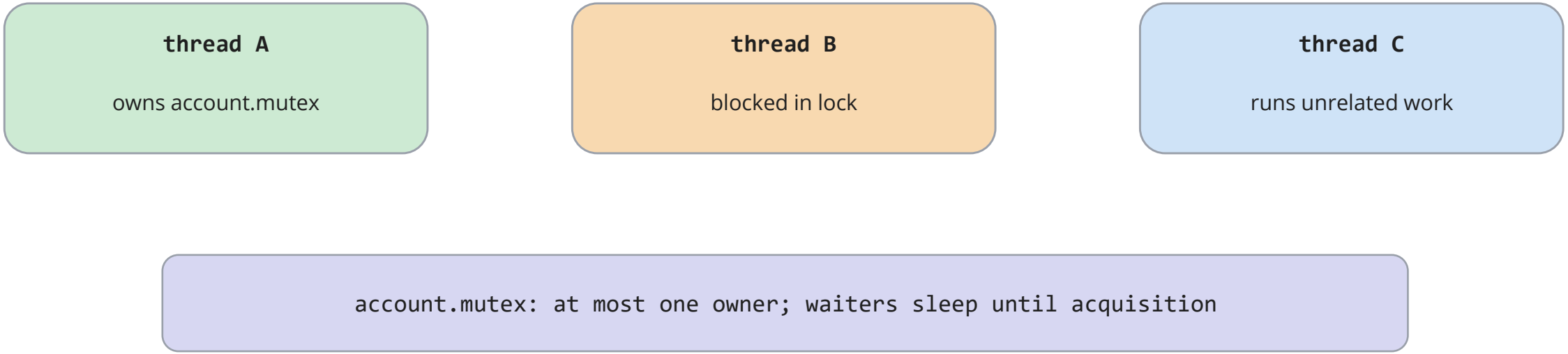
```
1  void Deposit(Account* account, int amount) {
2    account->balance += amount;
3  }
```

protected invariant

balance equals the sum of
all completed deposits

A critical section contains every shared access that must act as one indivisible state transition.

A mutex gives one thread temporary ownership



Mutual exclusion prevents overlapping critical sections. It does not prevent unrelated threads from running.

Store the mutex beside the protected balance

```c
typedef struct {
  pthread_mutex_t mutex;
  int balance;
} Account;

int Account_Init(Account* account) {
  account->balance = 0;
  return pthread_mutex_init(&account->mutex, NULL);
}

int Account_Destroy(Account* account) {
  return pthread_mutex_destroy(&account->mutex);
}
```

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

Lock the complete Deposit operation

```
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

Readers follow the same lock rule

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

Protection is a protocol

- A read conflicts with a concurrent Deposit write
- Locking only writers does not order the reader
- Copy balance while holding the mutex
- Use the copied int after unlocking

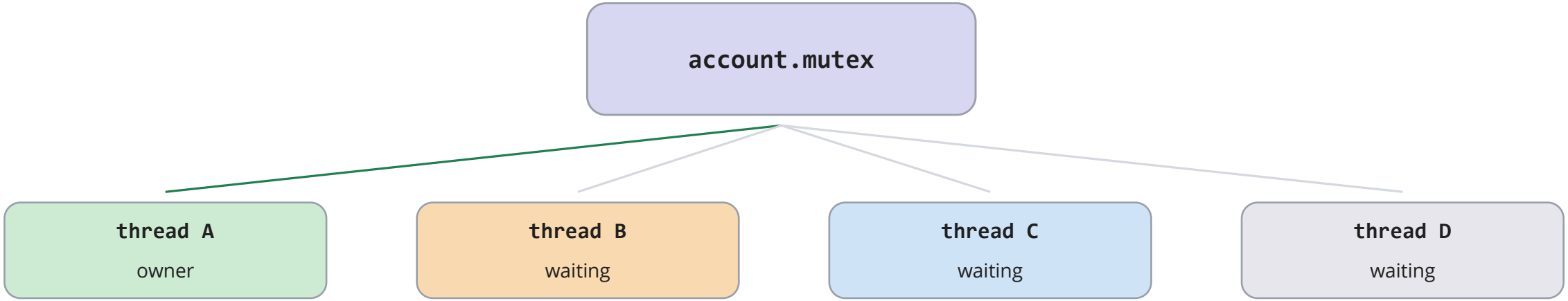
Correct locking can still be expensive

```
1  int Account_DepositMany(Account* account, int count) {
2    for (int i = 0; i < count; i++) {
3      int rc = pthread_mutex_lock(&account->mutex);
4      if (rc != 0) return rc;
5      account->balance += 1;
6      rc = pthread_mutex_unlock(&account->mutex);
7      if (rc != 0) return rc;
8    }
9    return 0;
10 }
```

One lock acquisition per dollar

- The data race is gone
- Every iteration enters the same critical section
- Competing threads repeatedly block and wake
- The mutex serializes nearly all useful work

Contention turns parallel workers into a queue



Contention means several threads want the same lock. Waiting preserves correctness, but blocking, wakeups, cache movement, and serialized work reduce throughput.

Batch private work, then lock once

```
1  int Account_DepositMany(Account* account, int count) {
2    int local_total = 0;
3    for (int i = 0; i < count; i++)
4      local_total += 1;  // private stack variable
5
6    int rc = pthread_mutex_lock(&account->mutex);
7    if (rc != 0) return rc;
8    account->balance += local_total;
9    return pthread_mutex_unlock(&account->mutex);
10 }
```

Reduce lock frequency

- ❖ local_total belongs to this thread's stack
- ❖ The loop needs no shared-memory coordination
- ❖ One critical section merges the final amount
- ❖ Four million acquisitions can become four

Keep slow independent work outside the lock

```
1  // pthread error checks omitted for focus
2  int Account_Deposit(Account* account, int amount) {
3    pthread_mutex_lock(&account->mutex);
4    account->balance += amount;
5    int new_balance = account->balance;
6    pthread_mutex_unlock(&account->mutex);
7
8    PrintReceipt(amount, new_balance);  // no Account access
9    return 0;
10 }
```

Protect Account, not the receipt printer

- Copy the needed result before unlocking
- Restore the invariant before releasing ownership
- PrintReceipt uses only private copies
- Other deposits proceed while the receipt is printed

Transfer extends the same example to two Accounts

```
1  typedef struct {
2    pthread_mutex_t mutex;
3    int id;
4    int balance;
5  } Account;
6
7  // pthread error checks omitted for focus
8  void Transfer(Account* from, Account* to, int amount) {
9    pthread_mutex_lock(&from->mutex);
10   pthread_mutex_lock(&to->mutex);
11   from->balance -= amount;
12   to->balance += amount;
13   pthread_mutex_unlock(&to->mutex);
14   pthread_mutex_unlock(&from->mutex);
15 }
```

One transfer needs two invariants

- Each balance remains protected by its own mutex
- The transfer must update both balances as one operation
- The function therefore owns both mutexes together
- Acquiring more than one mutex introduces a new hazard

Cycle

thread A

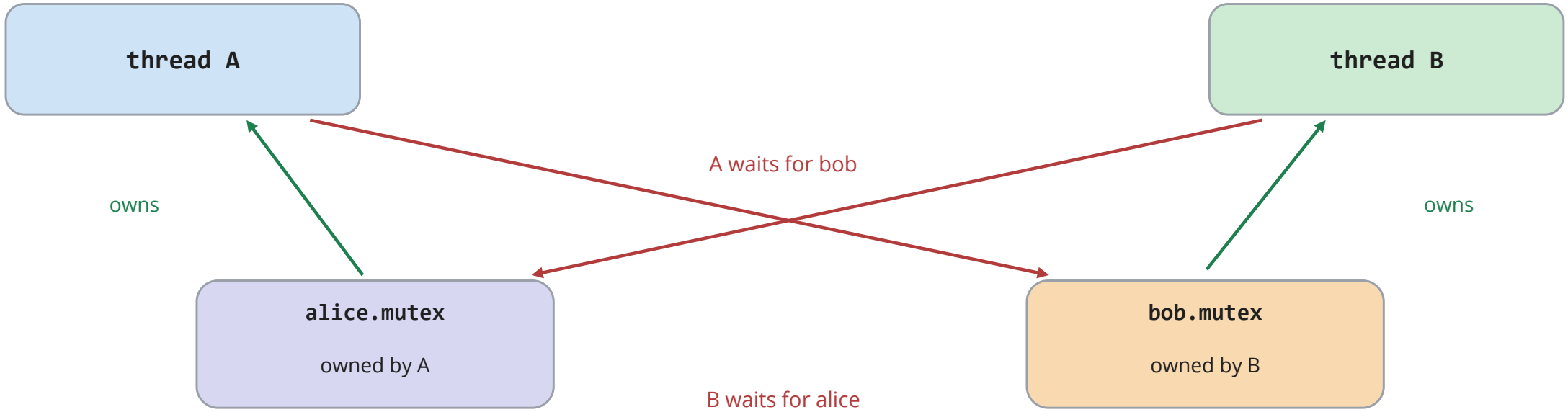
```
1  // thread A: Transfer(&alice, &bob, 10)
2  pthread_mutex_lock(&alice.mutex);
3  pthread_mutex_lock(&bob.mutex);
4  // update both balances
```

thread B

```
1  // thread B: Transfer(&bob, &alice, 20)
2  pthread_mutex_lock(&bob.mutex);
3  pthread_mutex_lock(&alice.mutex);
4  // update both balances
```

If A owns alice while B owns bob, each next lock waits for the mutex held by the other thread.

Deadlock is a cycle of ownership and waiting



Neither blocked thread can execute an unlock. The cycle cannot resolve itself.

Four conditions make this deadlock possible

1 mutual exclusion

one owner per account mutex

2 hold and wait

each thread holds one while requesting another

3 no preemption

only the owner releases its mutex

4 circular wait

A waits for B while B waits for A

All four are required. A global lock order prevents circular wait.

Acquire Account mutexes by stable ID

```
1  // Account IDs are unique and immutable
2  // pthread error checks omitted for focus
3  void Transfer(Account* from, Account* to, int amount) {
4    if (from == to) return;
5    Account* first  = from->id < to->id ? from : to;
6    Account* second = from->id < to->id ? to : from;
7
8    pthread_mutex_lock(&first->mutex);
9    pthread_mutex_lock(&second->mutex);
10   from->balance -= amount;
11   to->balance += amount;
12   pthread_mutex_unlock(&second->mutex);
13   pthread_mutex_unlock(&first->mutex);
14 }
```

One global order

- ❖ IDs are unique, immutable, and totally ordered
- ❖ A thread may wait before owning both, but no cycle forms
- ❖ Release in reverse order to preserve clear nesting
- ❖ Handle from == to before locking the same mutex twice

C++17 expresses the same ownership rules with RAII

one mutex

```
1  class Account {
2   public:
3    void Deposit(int amount) {
4      std::lock_guard<std::mutex> guard(mutex);
5      balance += amount;
6    }
7   private:
8    friend void Transfer(Account&, Account&, int);
9    std::mutex mutex;
10   int balance = 0;
11 };
```

two mutexes

```
1  void Transfer(Account& from, Account& to, int amount) {
2    if (&from == &to) return;
3    std::scoped_lock guard(from.mutex, to.mutex);
4    from.balance -= amount;
5    to.balance += amount;
6  }  // both mutexes unlock
```

lock_guard releases one mutex at scope exit. scoped_lock acquires several mutexes with deadlock avoidance.

Multiprocessing

Why use a process

pthread_create

```
1  int value = 0;
2  pthread_create(&t, nullptr, &Worker, &value);
3  // main and Worker can both write value
```

fork

```
1  int value = 0;
2  pid_t pid = fork();
3  // parent and child get separate logical copies
```

Why use a process

pthread_create

```
1  int value = 0;
2  pthread_create(&t, nullptr, &Worker, &value);
3  // main and Worker can both write value
```

fork

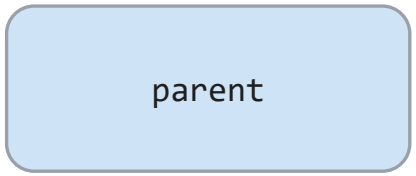
```
1  int value = 0;
2  pid_t pid = fork();
3  // parent and child get separate logical copies
```

Threads make shared-memory communication easy and they get to run on their own CPU cores. So why would we use processes?

Processes have memory isolation. One worker's ordinary writes do not change another worker's objects. Some common reasons are (1) Security boundary and (2) Memory limit

fork()

```
1  pid_t fork();
```

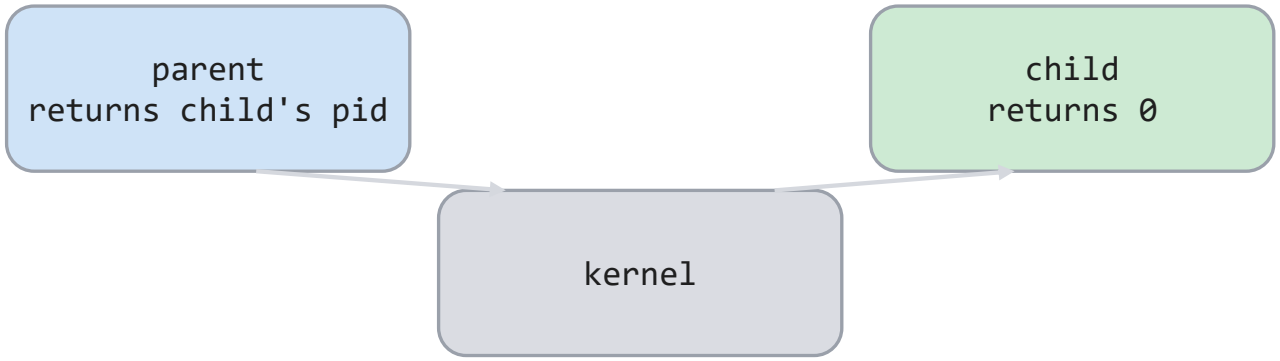


One process. Nothing has happened yet.

fork()

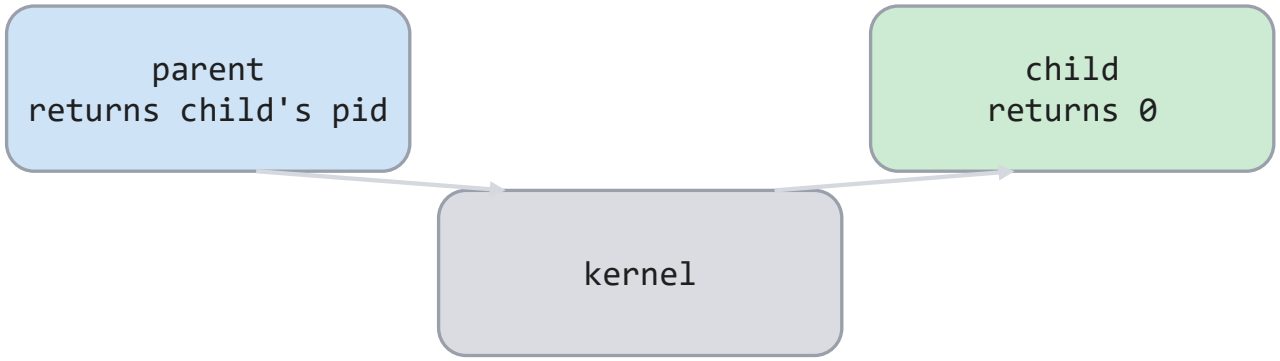
```
1  pid_t fork();
```

The kernel clones the process and both copies return from the one call. The descriptor table is cloned too. Pages are shared until written, so the copy is lazier than the picture suggests.



fork()

```
1  pid_t fork();
```



The kernel clones the process and both copies return from the one call. The descriptor table is cloned too. Pages are shared until written, so the copy is lazier than the picture suggests.

“Called one, returns twice”

Both processes continue

```
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

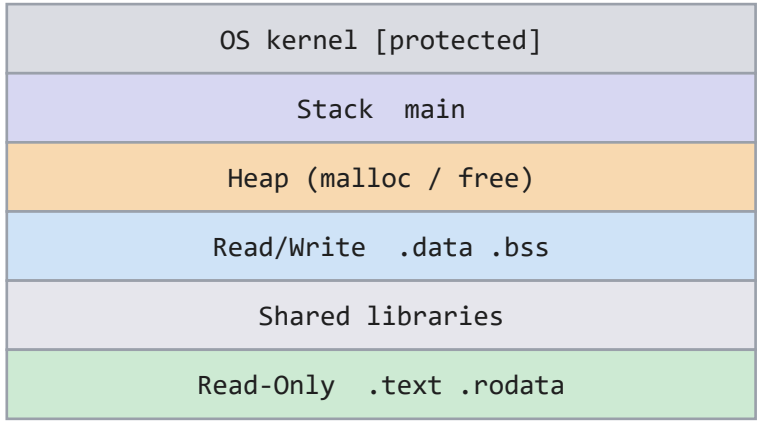
Parent and child are independently runnable

The same scheduler rules still apply

- On one logical CPU, parent and child are interleaved
- On different logical CPUs, they may execute in parallel
- Either process may reach its next instruction first
- wait and blocking I/O make a process non-runnable

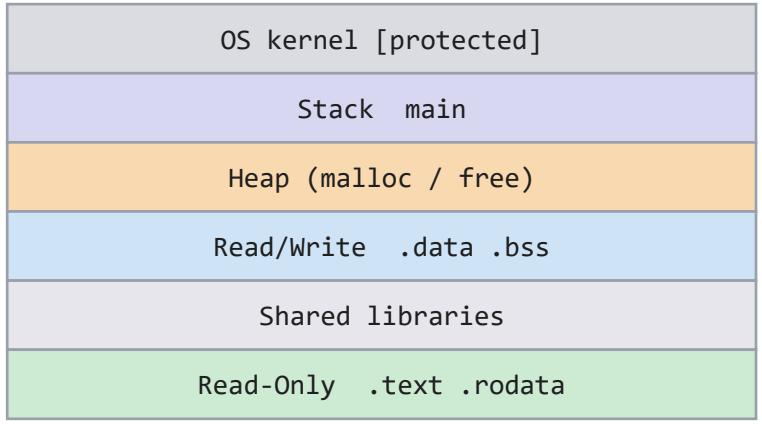
Two address spaces

parent



x = 2

child



x = 1

The contrast

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

Threads and processes

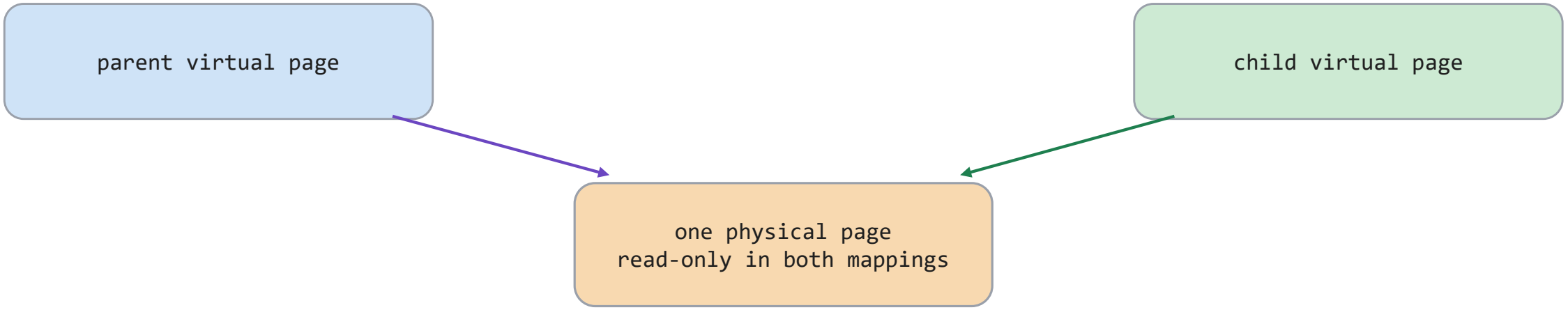
program state	pthread_create()	fork()
stack	one per thread	copied into child
heap and globals	shared	separate copies
file descriptors	shared table	inherited table copy
ordinary communication	read the same object	pipe, socket, file, or shared-memory mapping
failure isolation	low	higher

What a worker costs in each execution model

	fork()	pthread_create()
time to create one	around 0.3 ms	around 0.03 ms
workers per second per core	around 3,000	around 33,000
memory added	different virtual address spaces	one stack
shares memory with the creator	no	yes
needs locks	no	yes

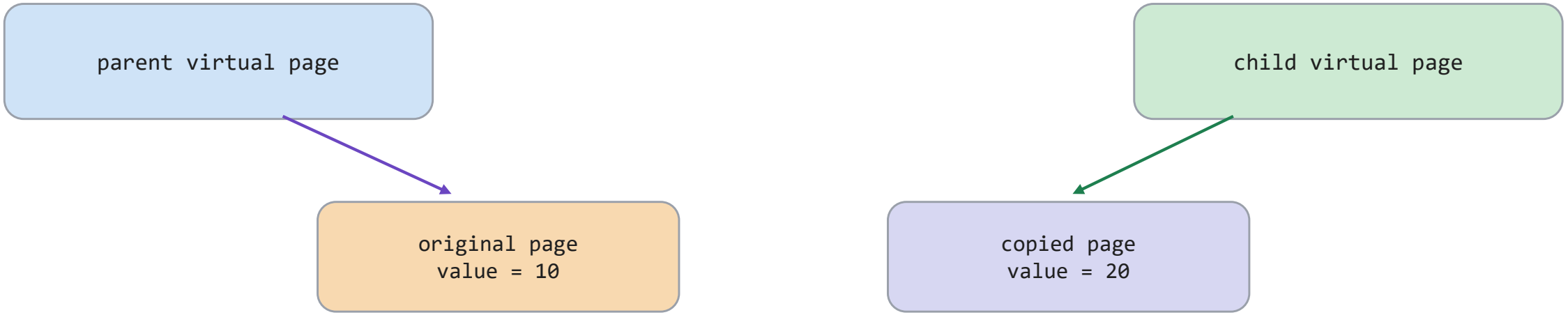
Measured on attu. The values depend on hardware and system load but the ratio is the useful comparison.

Copy-on-write avoids an immediate full copy



Immediately after fork, unchanged private pages may share physical memory while remaining logically isolated.

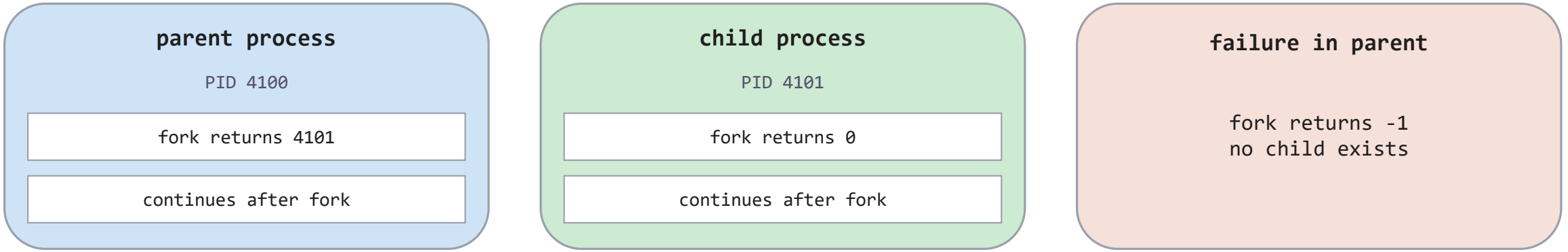
A write creates one private physical page



The kernel copies only the page being written; parent and child then map different physical frames.

fork has three return cases

C17 · POSIX process APIs

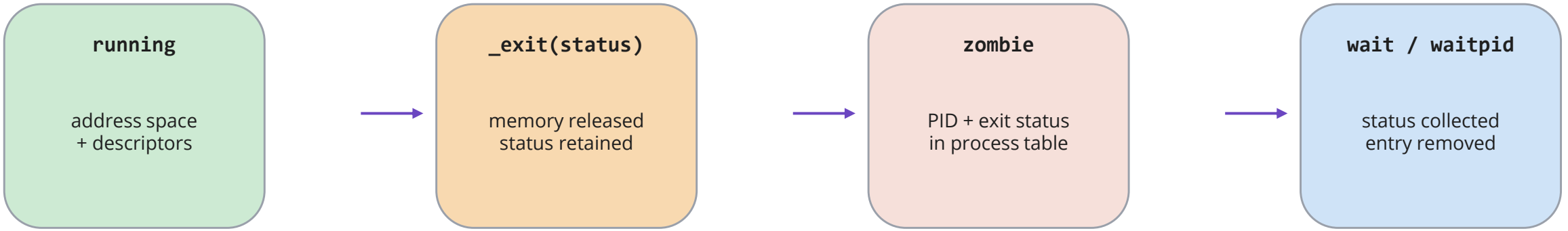


The return value identifies which execution path is running. It does not determine which process runs first.

Example: fork() return values

```
1  pid_t pid = fork();
2  if (pid < 0) {
3    perror("fork");          // fork sets errno
4    return EXIT_FAILURE;     // no child exists
5  }
6  if (pid == 0) {
7    HandleClient(client_fd);
8    _exit(EXIT_SUCCESS);     // do not flush copied stdio buffers
9  }
10 int status;
11 waitpid(pid, &status, 0);  // wait for this child
```

A terminated child leaves an exit status

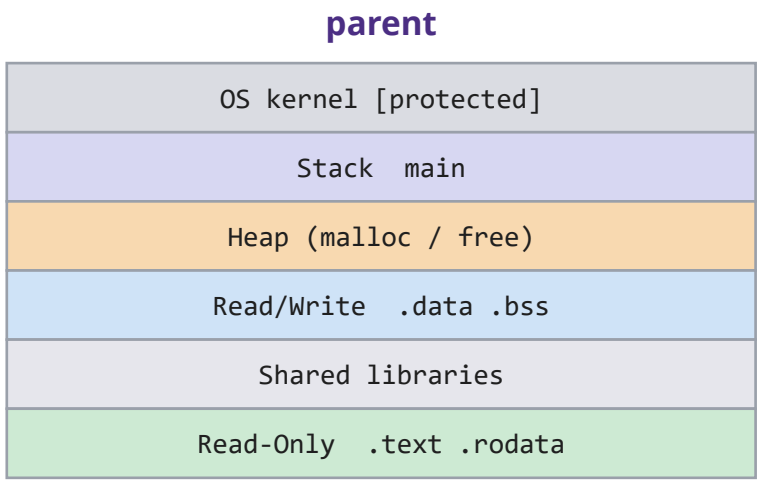


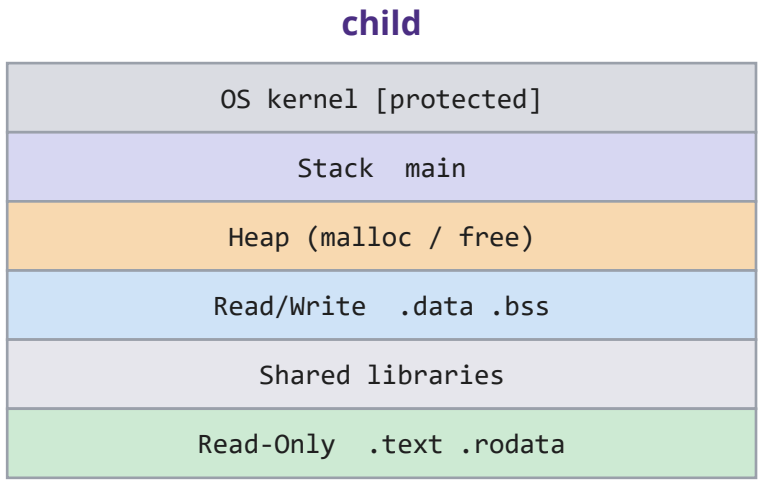
Once a child exits, it is a "zombie“ process that is not executing and retains no user address space. It does leave behind a small record in the kernel that can be read by the parent process.

Example: waiting on child process

```
1  pid_t pid = fork();
2  if (pid < 0) {
3    perror("fork");          // fork sets errno
4    return EXIT_FAILURE;     // no child exists
5  }
6  if (pid == 0) {
7    HandleClient(client_fd);
8    _exit(EXIT_SUCCESS);     // do not flush copied stdio buffers
9  }
10 int status;
11 waitpid(pid, &status, 0);  // wait for this child
```

Two address spaces (but…)





What about file descriptors?

Inherited descriptors

```
1  int client_fd = accept(listen_fd, nullptr, nullptr);
2  pid_t pid = fork();
3
4  // both processes now have client_fd open,
5  // pointing at the same connection
6
7  if (pid == 0) {
8    close(listen_fd);   // the child does not accept
9  } else {
10   close(client_fd);   // the parent does not serve
11 }
```

Why inherited descriptors matter

- Each side closes the one it does not need
- **Forget those close(…)** calls and the connection does not close.

Each process closes one descriptor

```
1  int fd = accept(listen_fd, nullptr, nullptr);
2  pid_t pid = fork();
3  if (pid == 0) {
4    close(listen_fd);
5    HandleClient(fd);
6    close(fd); // child closes its copy
7    _exit(EXIT_SUCCESS);
8  } else {
9    close(fd);  // parent closes its copy
10 }
```

Two descriptor entries, one connection

- The child does not accept, so it closes listen_fd
- The parent does not serve this client, so it closes fd
- The child closes fd after HandleClient returns
- The connection closes after the last copy closes

Reminders

Assessments:

- Exercise 8 due Friday at 11:59pm
- Homework 4 released soon (tricky in a different way!)

General:

- Tim is doing a GPU lecture on Week 9 Friday
- Final Exam
  - CSE2 G10 from 1:30-3:30pm on Tuesday, August 18
  - Makeup exam: 5-7pm on different days. Email coming soon!

Questions?