Advanced Systems Programming

Threads and Concurrency

Till Smejkal

September 24th, 2026

Technische Universität Dresden

About this course

- One day, three blocks, six exercises.
- The goal is code that is still correct when two things happen at once — and knowing when that is worth the trouble.
- You know C, and you have met modern C++. Both have threads. One of them will let you forget to unlock.

About this course

- One day, three blocks, six exercises.
- The goal is code that is still correct when two things happen at once — and knowing when that is worth the trouble.
- You know C, and you have met modern C++. Both have threads. One of them will let you forget to unlock.
- Almost everything here can be got wrong in a way that works on your machine, today, and fails in production in three weeks.

About this course

- One day, three blocks, six exercises.
- The goal is code that is still correct when two things happen at once — and knowing when that is worth the trouble.
- You know C, and you have met modern C++. Both have threads. One of them will let you forget to unlock.
- Almost everything here can be got wrong in a way that works on your machine, today, and fails in production in three weeks.
- Please interrupt and ask.

About this course

- One day, three blocks, six exercises.
- The goal is code that is still correct when two things happen at once — and knowing when that is worth the trouble.
- You know C, and you have met modern C++. Both have threads. One of them will let you forget to unlock.
- Almost everything here can be got wrong in a way that works on your machine, today, and fails in production in three weeks.
- Please interrupt and ask.

Key idea

Concurrency bugs are not rare. They are rarely *observed*, which is a different thing and a much worse one.

The plan for today

Block 1	Threads, the POSIX API, and `std::thread`	Ex. 1, 2
Block 2	Races, atomics, mutexes, and never unlocking by hand	Ex. 3, 4
Block 3	Semaphores, condition variables, deadlocks, futures	Ex. 5, 6

The plan for today

Block 1	Threads, the POSIX API, and `std::thread`	Ex. 1, 2
Block 2	Races, atomics, mutexes, and never unlocking by hand	Ex. 3, 4
Block 3	Semaphores, condition variables, deadlocks, futures	Ex. 5, 6

Roughly half slides and half typing. The third block is the long one: it carries the two hardest exercises, so the long break comes before it.

The plan for today

Block 1	Threads, the POSIX API, and `std::thread`	Ex. 1, 2
Block 2	Races, atomics, mutexes, and never unlocking by hand	Ex. 3, 4
Block 3	Semaphores, condition variables, deadlocks, futures	Ex. 5, 6

Roughly half slides and half typing. The third block is the long one: it carries the two hardest exercises, so the long break comes before it.

The exercises take one program — the log analyser from the C++ course — and make it concurrent. By the end of the day it is about three times faster — not sixteen.

The plan for today

Block 1	Threads, the POSIX API, and `std::thread`	Ex. 1, 2
Block 2	Races, atomics, mutexes, and never unlocking by hand	Ex. 3, 4
Block 3	Semaphores, condition variables, deadlocks, futures	Ex. 5, 6

Roughly half slides and half typing. The third block is the long one: it carries the two hardest exercises, so the long break comes before it.

The exercises take one program — the log analyser from the C++ course — and make it concurrent. By the end of the day it is about three times faster — not sixteen.

There is a seventh, on futures and `std::async`. We will cover the material; you take that one home.

Getting the code

```
$ git clone \
   https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-threads.git
$ cd sysprog-threads && make check
```

Getting the code

```
$ git clone \
   https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-threads.git
$ cd sysprog-threads && make check
```

It is the log analyser from the C++ course, finished and working — and single-threaded. You do not need to have been there; everything it already does is given.

Getting the code

```
$ git clone \
   https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-threads.git
$ cd sysprog-threads && make check
```

It is the log analyser from the C++ course, finished and working — and single-threaded. You do not need to have been there; everything it already does is given.

You need GCC 14 or newer, or Clang 18 or newer. All seven tests fail on a fresh clone, and `make check` stops at the first one.

Getting the code

```
$ git clone \
   https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-threads.git
$ cd sysprog-threads && make check
```

It is the log analyser from the C++ course, finished and working — and single-threaded. You do not need to have been there; everything it already does is given.

You need GCC 14 or newer, or Clang 18 or newer. All seven tests fail on a fresh clone, and `make check` stops at the first one.

Solutions live in the same repository, one commit per exercise:

```
$ git show ex3
$ git diff main ex3
```

Why bother: keeping the program responsive

A browser downloading a large file. One thread does the download; the rest of the program keeps answering the user.

Why bother: keeping the program responsive

A browser downloading a large file. One thread does the download; the rest of the program keeps answering the user.

Without threads you have two options, and both are worse:

- block, and the window stops repainting until the file arrives;

Why bother: keeping the program responsive

A browser downloading a large file. One thread does the download; the rest of the program keeps answering the user.

Without threads you have two options, and both are worse:

- block, and the window stops repainting until the file arrives;
- turn the whole program inside out into a state machine driven by one event loop, so that no operation is ever allowed to take long.

Why bother: keeping the program responsive

A browser downloading a large file. One thread does the download; the rest of the program keeps answering the user.

Without threads you have two options, and both are worse:

- block, and the window stops repainting until the file arrives;
- turn the whole program inside out into a state machine driven by one event loop, so that no operation is ever allowed to take long.

Key idea

A thread lets you write a slow operation as a straight line of code while somebody else keeps the lights on.

Why bother: getting the answer sooner

The other reason: the work divides, and the machine has more than one core.

- Parse eight log files instead of one at a time.
- Same computation, different data — the shape most numerical work has.

Why bother: getting the answer sooner

The other reason: the work divides, and the machine has more than one core.

- Parse eight log files instead of one at a time.
- Same computation, different data — the shape most numerical work has.

This only pays on a machine with cores to spare. On one core, splitting CPU-bound work into threads makes it slower: same instructions, plus the scheduling.

Why bother: getting the answer sooner

The other reason: the work divides, and the machine has more than one core.

- Parse eight log files instead of one at a time.
- Same computation, different data — the shape most numerical work has.

This only pays on a machine with cores to spare. On one core, splitting CPU-bound work into threads makes it slower: same instructions, plus the scheduling.

Pitfall

Two different goals, two different designs

Responsiveness wants a thread *per activity*. Throughput wants a thread *per core*. Confusing the two is how programs end up with four hundred threads.

CPU-bound and I/O-bound

	CPU-bound	I/O-bound
Waiting for	nothing, it is computing	disk, network, a user
Threads help by	using more cores	overlapping the waiting
Useful threads	about one per core	many more than cores
On one core	slower, never faster	still much faster

CPU-bound and I/O-bound

	CPU-bound	I/O-bound
Waiting for	nothing, it is computing	disk, network, a user
Threads help by	using more cores	overlapping the waiting
Useful threads	about one per core	many more than cores
On one core	slower, never faster	still much faster

Today's exercise is both at once: reading the files is I/O, parsing them is CPU. That mixture is normal, and it is why the speedup has to be measured.

CPU-bound and I/O-bound

	CPU-bound	I/O-bound
Waiting for	nothing, it is computing	disk, network, a user
Threads help by	using more cores	overlapping the waiting
Useful threads	about one per core	many more than cores
On one core	slower, never faster	still much faster

Today's exercise is both at once: reading the files is I/O, parsing them is CPU. That mixture is normal, and it is why the speedup has to be measured.

Best practice

Find out which one you have before you add a thread. The answer changes what the right design is, and guessing it is free but usually wrong.

So this sounds great

It is not free. Two things stop being true the moment there is a second thread:

- statements no longer happen in the order they are written — not across threads, and not always within one;

So this sounds great

It is not free. Two things stop being true the moment there is a second thread:

- statements no longer happen in the order they are written — not across threads, and not always within one;
- data you thought was yours can change between two lines that look adjacent.

So this sounds great

It is not free. Two things stop being true the moment there is a second thread:

- statements no longer happen in the order they are written — not across threads, and not always within one;
- data you thought was yours can change between two lines that look adjacent.

Every bug in this lecture is one of those two.

So this sounds great

It is not free. Two things stop being true the moment there is a second thread:

- statements no longer happen in the order they are written — not across threads, and not always within one;
- data you thought was yours can change between two lines that look adjacent.

Every bug in this lecture is one of those two.

Key idea

Rule #1. Never make any assumption about the order of execution or about timing. If the code needs an order, you have to say so, with a lock or with a signal. Anything you merely expect, you have already got wrong.

BLOCK 1 / 3

Threads and how to make them

- Processes and threads
- POSIX threads
- Threads in C++

Exercises 1 and 2

What a process owns

A process is the unit the operating system protects:

- an address space of its own — one process cannot see another's memory, and that is enforced by hardware;
- resources: open files, sockets, shared memory segments;
- an identity — it is the thing permissions are checked against.

What a process owns

A process is the unit the operating system protects:

- an address space of its own — one process cannot see another's memory, and that is enforced by hardware;
- resources: open files, sockets, shared memory segments;
- an identity — it is the thing permissions are checked against.

A *thread* owns none of that. It belongs to a process, and it shares everything the process has with every other thread in it.

What a process owns

A process is the unit the operating system protects:

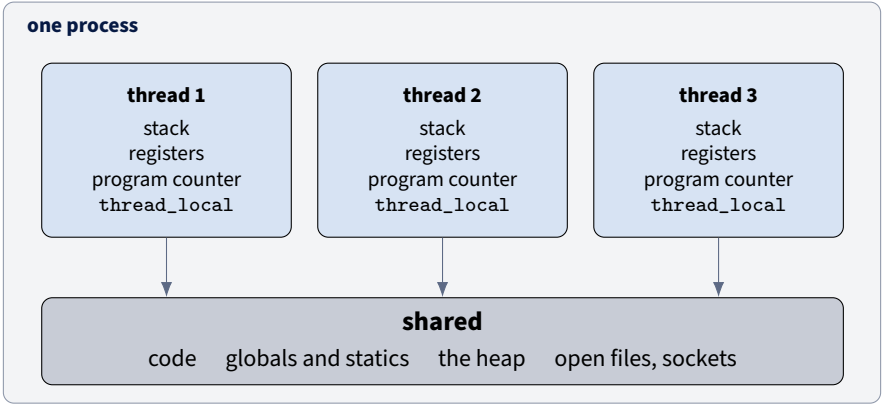
- an address space of its own — one process cannot see another's memory, and that is enforced by hardware;
- resources: open files, sockets, shared memory segments;
- an identity — it is the thing permissions are checked against.

A *thread* owns none of that. It belongs to a process, and it shares everything the process has with every other thread in it.

Key idea

Two processes share nothing unless they ask. Two threads share everything unless you arrange otherwise. That is the whole difference, and every other difference follows from it.

What is shared and what is not



The things you forget are per-thread

- **The stack.** Every local variable is private — which is why the easiest way to avoid a race is to not share in the first place.

The things you forget are per-thread

- **The stack.** Every local variable is private — which is why the easiest way to avoid a race is to not share in the first place.
- `errno`**.** It looks like a global and has to be per-thread, or two failing calls would overwrite each other. On Linux it is a macro that expands to a thread-local lookup.

The things you forget are per-thread

- **The stack.** Every local variable is private — which is why the easiest way to avoid a race is to not share in the first place.
- `errno`**.** It looks like a global and has to be per-thread, or two failing calls would overwrite each other. On Linux it is a macro that expands to a thread-local lookup.
- **Thread-local variables.** `thread_local` gives one instance per thread, constructed on first use and destroyed when the thread ends.

The things you forget are per-thread

- **The stack.** Every local variable is private — which is why the easiest way to avoid a race is to not share in the first place.
- `errno`**.** It looks like a global and has to be per-thread, or two failing calls would overwrite each other. On Linux it is a macro that expands to a thread-local lookup.
- **Thread-local variables.** `thread_local` gives one instance per thread, constructed on first use and destroyed when the thread ends.

And the things you forget are *not* per-thread: `static` locals inside a function, the `rand()` state, the current working directory, and the file-descriptor table.

The things you forget are per-thread

- **The stack.** Every local variable is private — which is why the easiest way to avoid a race is to not share in the first place.
- `errno`**.** It looks like a global and has to be per-thread, or two failing calls would overwrite each other. On Linux it is a macro that expands to a thread-local lookup.
- **Thread-local variables.** `thread_local` gives one instance per thread, constructed on first use and destroyed when the thread ends.

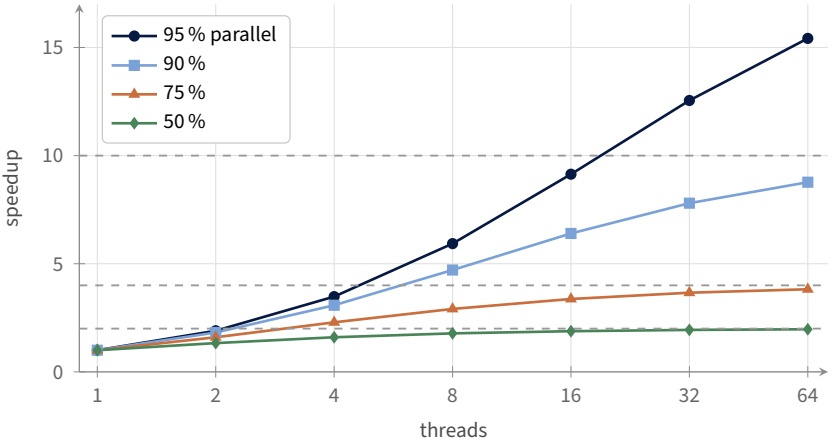
And the things you forget are *not* per-thread: `static` locals inside a function, the `rand()` state, the current working directory, and the file-descriptor table.

Pitfall

“It worked when I called it from one thread”

A function that keeps state in a `static` is not reentrant. `strtok`, `gmtime` and `rand` all do; their `_r` variants exist for this reason.

How much can this possibly buy?



The POSIX thread interface

A standardised C interface for threads on Unix-like systems — IEEE POSIX 1003.1c, which is why everything in it is called `pthread_something`.

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

The POSIX thread interface

A standardised C interface for threads on Unix-like systems — IEEE POSIX 1003.1c, which is why everything in it is called `pthread_something`.

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

Compile *and* link with `-pthread`:

```
$ gcc -pthread -o prog prog.c
```

The POSIX thread interface

A standardised C interface for threads on Unix-like systems — IEEE POSIX 1003.1c, which is why everything in it is called `pthread_something`.

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

Compile *and* link with `-pthread`:

```
$ gcc -pthread -o prog prog.c
```

Not `-lpthread`: that links the library but skips the defines the headers need, and on glibc it then misbehaves.

The POSIX thread interface

A standardised C interface for threads on Unix-like systems — IEEE POSIX 1003.1c, which is why everything in it is called `pthread_something`.

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

Compile *and* link with `-pthread`:

```
$ gcc -pthread -o prog prog.c
```

Not `-lpthread`: that links the library but skips the defines the headers need, and on glibc it then misbehaves.

Best practice

`man pthreads` is a genuinely good overview, and every function has its own page. You will need them; the interface is large and the names are not guessable.

Creating a thread

```
1 int pthread_create(pthread_t *thread,
2                    const pthread_attr_t *attr,
3                    void *(*start_routine)(void *),
4                    void *arg);
```

Creating a thread

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

- `thread` — out-parameter, the new thread's id.
- `attr` — stack size, detach state, scheduling. `NULL` for the defaults.
- `start_routine` — where it starts. `void *` in, `void *` out.
- `arg` — handed over untouched.

Creating a thread

```
1 int pthread_create(pthread_t *thread,
2                    const pthread_attr_t *attr,
3                    void *(*start_routine)(void *),
4                    void *arg);
```

- `thread` — out-parameter, the new thread's id.
- `attr` — stack size, detach state, scheduling. `NULL` for the defaults.
- `start_routine` — where it starts. `void *` in, `void *` out.
- `arg` — handed over untouched.

Pitfall

The return value is the error

It does *not* set `errno`, so use `strerror(err)`, not `perror`.

Getting data in

One `void *`, for everything. For a single value that fits, people cast it through the pointer:

```
1 pthread_create(&tid, NULL, worker, (void *)(long)i);   /* in  */
2 long i = (long)arg;                                    /* out */
```

Getting data in

One `void *`, for everything. For a single value that fits, people cast it through the pointer:

```
pthread_create(&tid, NULL, worker, (void *)(long)i);   /* in  */
long i = (long)arg;                                    /* out */
```

Legal, ugly, and limited to one word. For anything real you pass a pointer to a struct — and then the question is who owns it and how long it lives.

Getting data in

One `void *`, for everything. For a single value that fits, people cast it through the pointer:

```
1 pthread_create(&tid, NULL, worker, (void *)(long)i);   /* in  */
2 long i = (long)arg;                                    /* out */
```

Legal, ugly, and limited to one word. For anything real you pass a pointer to a struct — and then the question is who owns it and how long it lives.

Key idea

The type system has stopped helping. Whatever you cast in, you cast back out, and nothing checks that the two agree.

One object each, not one object

```
struct task { int number; const char *path; };

struct task *tasks = calloc(n, sizeof(*tasks));        /* one each */

for (int i = 0; i < n; ++i) {
        tasks[i].number = i;
        pthread_create(&threads[i], NULL, worker, &tasks[i]);
}
```

One object each, not one object

```
1  struct task { int number; const char *path; };
2
3  struct task *tasks = calloc(n, sizeof(*tasks));        /* one each */
4
5  for (int i = 0; i < n; ++i) {
6          tasks[i].number = i;
7          pthread_create(&threads[i], NULL, worker, &tasks[i]);
8  }
```

`tasks` outlives every thread and each gets a different element.

One object each, not one object

```
1  struct task { int number; const char *path; };
2
3  struct task *tasks = calloc(n, sizeof(*tasks));        /* one each */
4
5  for (int i = 0; i < n; ++i) {
6          tasks[i].number = i;
7          pthread_create(&threads[i], NULL, worker, &tasks[i]);
8  }
```

`tasks` outlives every thread and each gets a different element.

Pitfall

Passing `&i`

The loop variable gives every thread a pointer to the *same* `int`, which the loop then keeps changing. This is exercise 1, and the single most common pthreads bug there is.

Getting data out, and waiting

```
1 void  pthread_exit(void *retval);
2 int   pthread_join(pthread_t thread, void **retval);
```

Getting data out, and waiting

```
1 void  pthread_exit(void *retval);
2 int   pthread_join(pthread_t thread, void **retval);
```

A thread ends by returning from its start routine or by calling `pthread_exit`. Both hand over one `void *`.

Getting data out, and waiting

```c
void  pthread_exit(void *retval);
int   pthread_join(pthread_t thread, void **retval);
```

A thread ends by returning from its start routine or by calling `pthread_exit`. Both hand over one `void *`.

`pthread_join` blocks until that thread has ended and gives you the value. It also releases the thread's stack — a thread nobody joins keeps its resources until the process exits.

Getting data out, and waiting

```
1 void  pthread_exit(void *retval);
2 int   pthread_join(pthread_t thread, void **retval);
```

A thread ends by returning from its start routine or by calling `pthread_exit`. Both hand over one `void *`.

`pthread_join` blocks until that thread has ended and gives you the value. It also releases the thread's stack — a thread nobody joins keeps its resources until the process exits.

Pitfall

`exit` is not `pthread_exit`

`exit()` ends the whole process from whichever thread calls it. `pthread_exit()` ends only the calling thread. Calling `exit` where you meant `pthread_exit` takes down fifteen healthy threads without a word.

Exercise 1 — `square_all`

In `pthreads.c`:

```
1 int square_all(int count, long *results);
```

Start `count` threads. Each prints which thread it is and hands back the square of its own number; `results[i]` ends up with thread `i`'s value.

Exercise 1 — `square_all`

In `pthreads.c`:

```
1 int square_all(int count, long *results);
```

Start `count` threads. Each prints which thread it is and hands back the square of its own number; `results[i]` ends up with thread `i`'s value.

It is already written. It compiles, it runs, and `./test-pthreads` fails — not always on the first round, but somewhere in the forty.

Exercise 1 — `square_all`

In `pthreads.c`:

```
1 int square_all(int count, long *results);
```

Start `count` threads. Each prints which thread it is and hands back the square of its own number; `results[i]` ends up with thread `i`'s value.

It is already written. It compiles, it runs, and `./test-pthreads` fails — not always on the first round, but somewhere in the forty.

- Every thread is handed `&i`. How many `i`s are there?
- `pthread_create` returns once the thread has been *created*. Nothing says it has run an instruction yet.
- The fix is not a lock. Give each thread something of its own that lives until the join.

```
$ make test-pthreads && ./test-pthreads
```

The same thing, with types

```
#include <thread>

void work() { /* ... */ }

int main()
{
        std::thread t{work};
        t.join();
}
```

The same thing, with types

```
#include <thread>

void work() { /* ... */ }

int main()
{
        std::thread t{work};
        t.join();
}
```

On this system that is a wrapper around `pthread_create`. What it adds is not speed — it is that the compiler now checks things:

- any callable will do: a function, a lambda, a member function;

The same thing, with types

```
#include <thread>

void work() { /* ... */ }

int main()
{
        std::thread t{work};
        t.join();
}
```

On this system that is a wrapper around `pthread_create`. What it adds is not speed — it is that the compiler now checks things:

- any callable will do: a function, a lambda, a member function;
- arguments keep their types, so there is no `void *` and no cast.

Passing arguments

```
1 void parse(std::string path, int id);
2
3 std::thread t{parse, "app.log", 7};        // copied into the thread
```

Passing arguments

```
1  void parse(std::string path, int id);
2
3  std::thread t{parse, "app.log", 7};        // copied into the thread
```

The arguments are *copied* into the thread's own storage — which is exactly what exercise 1 had to do by hand. To pass a reference you have to say so:

```
4  void fill(std::vector<Entry>& out);
5
6  std::thread t{fill, std::ref(entries)};    // without std::ref: a copy
```

Passing arguments

```
1  void parse(std::string path, int id);
2
3  std::thread t{parse, "app.log", 7};        // copied into the thread
```

The arguments are *copied* into the thread's own storage — which is exactly what exercise 1 had to do by hand. To pass a reference you have to say so:

```
4  void fill(std::vector<Entry>& out);
5
6  std::thread t{fill, std::ref(entries)};    // without std::ref: a copy
```

Pitfall

`std::ref` is a promise about lifetime

Whatever you refer to has to outlive the thread. The `join` is what makes that true — which is why the join usually belongs in the same scope as the data.

`std::thread` is not RAII

A `std::thread` that still refers to a running thread is *joinable*. Destroying a joinable thread does not join it and does not detach it:

```
1  void process(std::vector<std::string> const& paths)
2  {
3          std::thread t{load, paths};
4          if (paths.empty())
5                  return;              // ~thread() -> std::terminate()
6
7          t.join();
8  }
```

`std::thread` is not RAII

A `std::thread` that still refers to a running thread is *joinable*. Destroying a joinable thread does not join it and does not detach it:

```
1  void process(std::vector<std::string> const& paths)
2  {
3          std::thread t{load, paths};
4          if (paths.empty())
5                  return;              // ~thread() -> std::terminate()
6
7          t.join();
8  }
```

Compiler error

```
terminate called without an active exception
```

`std::thread` is not RAII

A `std::thread` that still refers to a running thread is *joinable*. Destroying a joinable thread does not join it and does not detach it:

```
1  void process(std::vector<std::string> const& paths)
2  {
3          std::thread t{load, paths};
4          if (paths.empty())
5                  return;              // ~thread() -> std::terminate()
6
7          t.join();
8  }
```

Compiler error

```
terminate called without an active exception
```

Deliberate: a silent join hides an unbounded wait in a destructor.

`std::jthread`, which is

```
1 {
2         std::jthread t{load, paths};
3
4         if (paths.empty())
5                 return;                 // ~jthread() joins.  Fine.
6 }                                       // and here too
```

`std::jthread`, which is

```
1 {
2         std::jthread t{load, paths};
3
4         if (paths.empty())
5                 return;                 // ~jthread() joins.  Fine.
6 }                                       // and here too
```

C++20's `std::jthread` joins in its destructor. Every exit from the scope — falling off the end, returning early, an exception unwinding through — waits for the thread.

`std::jthread`, which is

```
{
        std::jthread t{load, paths};

        if (paths.empty())
                return;                 // ~jthread() joins.  Fine.
}                                       // and here too
```

C++20's `std::jthread` joins in its destructor. Every exit from the scope — falling off the end, returning early, an exception unwinding through — waits for the thread.

Best practice

Use `std::jthread`. Reach for `std::thread` only when you genuinely need to outlive the scope, and then say why in a comment.

A pool in four lines

```
{
        std::vector<std::jthread> workers;

        for (auto const& path : paths)
                workers.emplace_back(load_one, path);
}                               // all of them joined, in order
```

A pool in four lines

```
{
        std::vector<std::jthread> workers;

        for (auto const& path : paths)
                workers.emplace_back(load_one, path);
}                               // all of them joined, in order
```

No cleanup code, no join loop, and correct if the body throws. This is the shape of every exercise from here on.

A pool in four lines

```
{
        std::vector<std::jthread> workers;

        for (auto const& path : paths)
                workers.emplace_back(load_one, path);
}                               // all of them joined, in order
```

No cleanup code, no join loop, and correct if the body throws. This is the shape of every exercise from here on.

Pitfall

The closing brace is a blocking call

It waits for the slowest thread. That is what you wanted — but it is worth knowing that the brace is where the time goes.

How many threads?

```
1 unsigned n = std::thread::hardware_concurrency();
```

How many threads?

```
1 unsigned n = std::thread::hardware_concurrency();
```

A hint, not a promise. It may return 0 — “no idea” — so always have a fallback. It counts *hardware threads*, so on a machine with eight cores and SMT it says sixteen, and those sixteen are not sixteen cores’ worth of arithmetic.

How many threads?

```
1 unsigned n = std::thread::hardware_concurrency();
```

A hint, not a promise. It may return 0 — “no idea” — so always have a fallback. It counts *hardware threads*, so on a machine with eight cores and SMT it says sixteen, and those sixteen are not sixteen cores’ worth of arithmetic.

It also does not know about your cgroup. In a container limited to two CPUs it will still cheerfully say sixteen.

How many threads?

```cpp
unsigned n = std::thread::hardware_concurrency();
```

A hint, not a promise. It may return 0 — “no idea” — so always have a fallback. It counts *hardware threads*, so on a machine with eight cores and SMT it says sixteen, and those sixteen are not sixteen cores’ worth of arithmetic.

It also does not know about your cgroup. In a container limited to two CPUs it will still cheerfully say sixteen.

Best practice

Use it to *size* a pool, never to decide whether to parallelise. And make it configurable — the exercise’s `analyse` takes `-j` for exactly this reason.

Exercise 2 — one thread per file

In `load.cc`:

```
1 LoadResult load_threaded(std::span<std::string const> paths);
```

One thread per log file. It has to produce exactly what `load_serial` produces — same entries, same order — only sooner.

Exercise 2 — one thread per file

In `load.cc`:

```
1 LoadResult load_threaded(std::span<std::string const> paths);
```

One thread per log file. It has to produce exactly what `load_serial` produces — same entries, same order — only sooner.

- `read_file(path)` is given, and touches nothing outside itself.
- Give every worker its own slot to write into. Size the vector *before* any thread starts.
- Merge afterwards, in index order. That is what keeps the output identical however the threads were scheduled.

```
$ make test-load && ./test-load
```

BLOCK 2 / 3

Shared state and how to protect it

- Race conditions
- How locks are built
- Mutexes
- Never forgetting to unlock

Exercises 3 and 4

One statement, three instructions

Two threads, one counter, one line of code each:

```
1 ++count;
```

One statement, three instructions

Two threads, one counter, one line of code each:

```
1 ++count;
```

There is no such instruction. What the CPU is asked to do is:

```
2 LOAD  count -> register
3 INC   register
4 STORE register -> count
```

One statement, three instructions

Two threads, one counter, one line of code each:

```
++count;
```

There is no such instruction. What the CPU is asked to do is:

```
LOAD  count -> register
INC   register
STORE register -> count
```

Three steps, and the thread can be taken off the core between any two of them — or run on a second core at the same time, which is worse: then there is no “between” at all.

One statement, three instructions

Two threads, one counter, one line of code each:

```
1 ++count;
```

There is no such instruction. What the CPU is asked to do is:

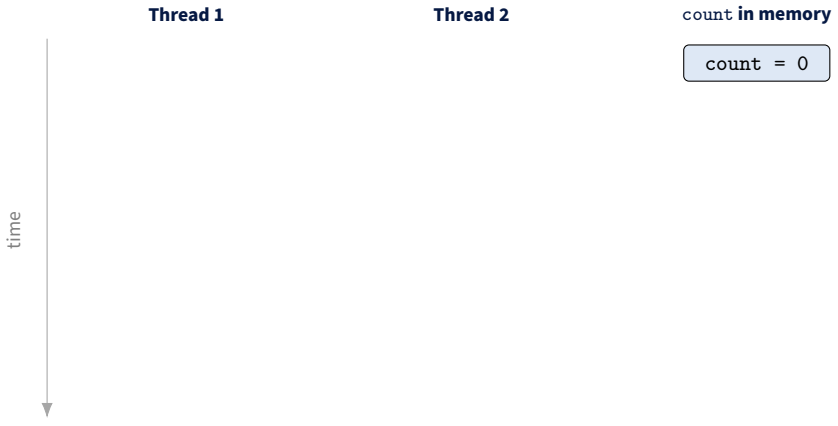
```
2 LOAD  count -> register
3 INC   register
4 STORE register -> count
```

Three steps, and the thread can be taken off the core between any two of them — or run on a second core at the same time, which is worse: then there is no “between” at all.

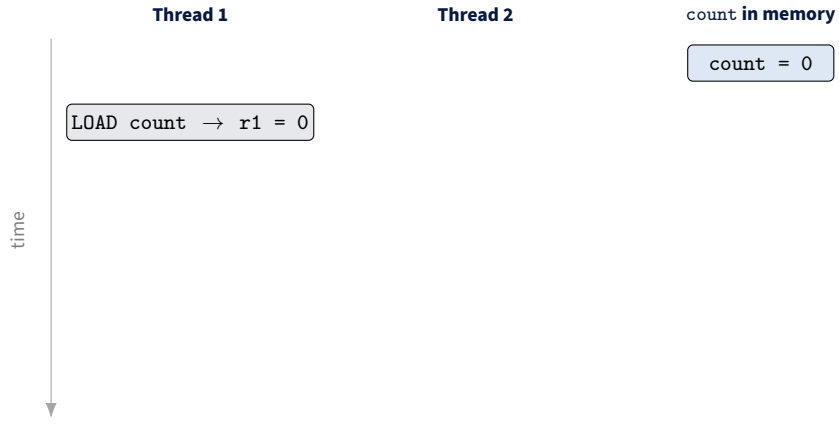
Key idea

Atomicity is not a property of a line of source. It is a property the hardware gives you, and by default it does not.

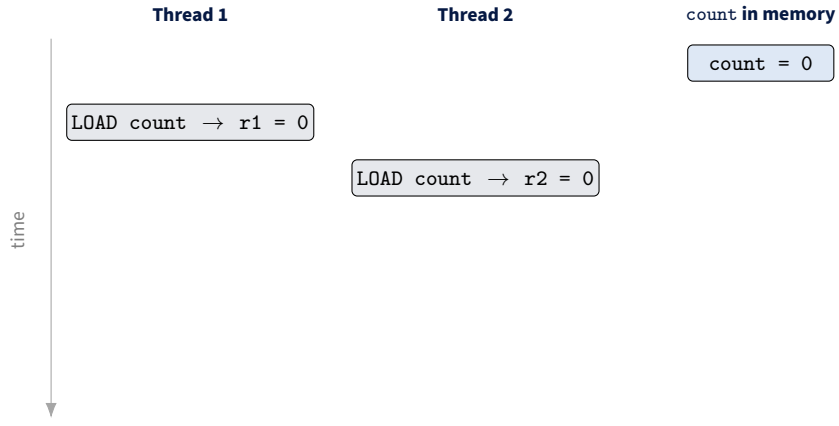
Two increments, one result



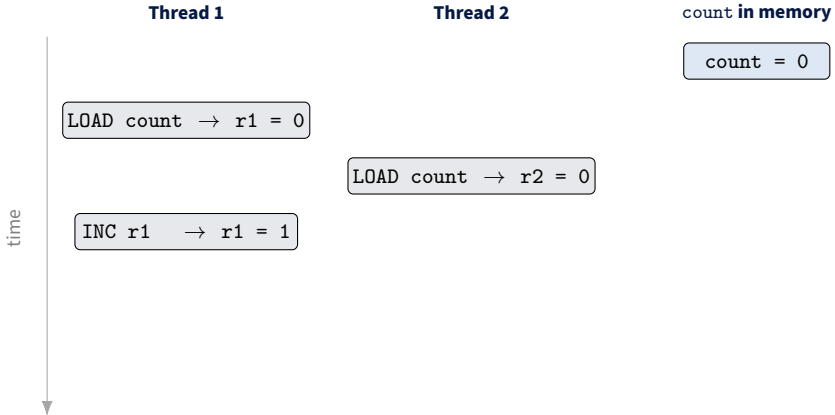
Two increments, one result



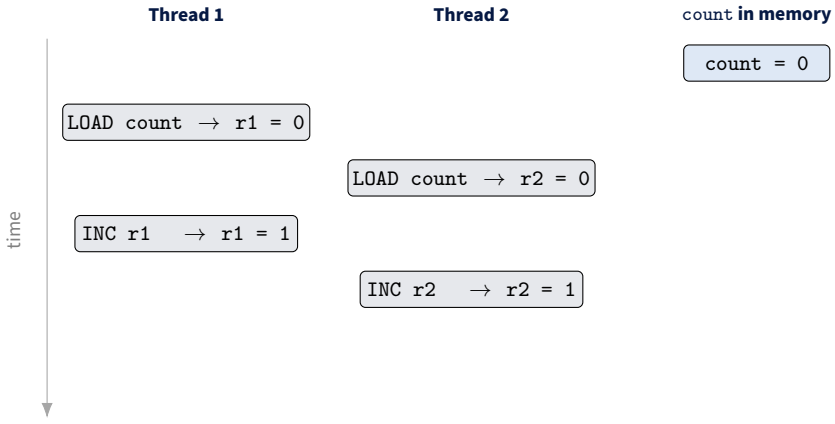
Two increments, one result



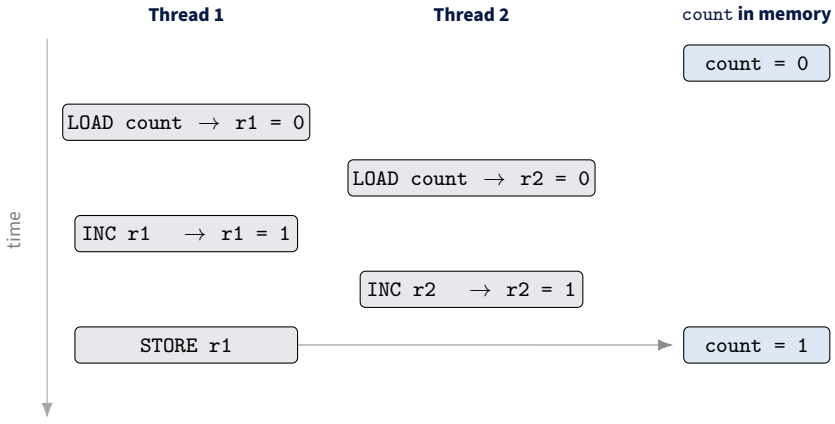
Two increments, one result



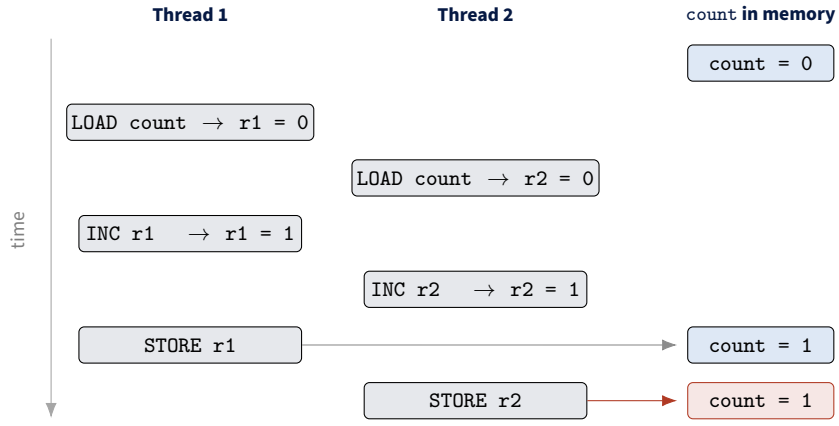
Two increments, one result



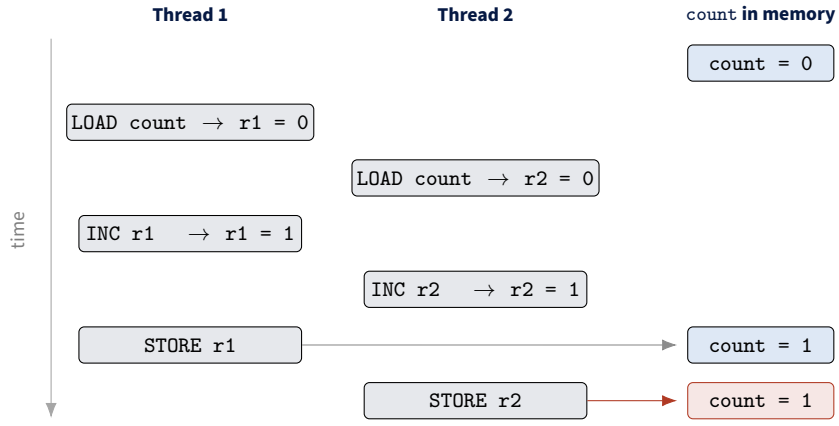
Two increments, one result



Two increments, one result



Two increments, one result



two increments, one result

Critical sections

A *critical section* is a piece of code that must not be entered by two threads at once, because it reads and writes state that they share.

Critical sections

A *critical section* is a piece of code that must not be entered by two threads at once, because it reads and writes state that they share.

The counter above is the smallest possible one. Real ones are bigger: the three lines that push onto a list, the five that rebalance a tree, the one that hands out the next free slot.

Critical sections

A *critical section* is a piece of code that must not be entered by two threads at once, because it reads and writes state that they share.

The counter above is the smallest possible one. Real ones are bigger: the three lines that push onto a list, the five that rebalance a tree, the one that hands out the next free slot.

Two things make a critical section:

- **shared** — more than one thread can reach the data;
- **mutable** — at least one of them writes.

Critical sections

A *critical section* is a piece of code that must not be entered by two threads at once, because it reads and writes state that they share.

The counter above is the smallest possible one. Real ones are bigger: the three lines that push onto a list, the five that rebalance a tree, the one that hands out the next free slot.

Two things make a critical section:

- **shared** — more than one thread can reach the data;
- **mutable** — at least one of them writes.

Key idea

Any number of threads may read the same data at the same time. It is the first writer that creates the problem — which is why the cheapest fix is always to stop sharing, not to lock better.

It is worse than a wrong number

A lost increment is the friendly version. What exercise 3 actually does:

```
1 result.entries.insert(result.entries.end(), part.begin(), part.end());
```

It is worse than a wrong number

A lost increment is the friendly version. What exercise 3 actually does:

```
1 result.entries.insert(result.entries.end(), part.begin(), part.end());
```

- two threads read the same `end()` and both write there;

It is worse than a wrong number

A lost increment is the friendly version. What exercise 3 actually does:

```
1 result.entries.insert(result.entries.end(), part.begin(), part.end());
```

- two threads read the same `end()` and both write there;
- one reallocates while the other is copying into the old buffer — a use-after-free;

It is worse than a wrong number

A lost increment is the friendly version. What exercise 3 actually does:

```
1 result.entries.insert(result.entries.end(), part.begin(), part.end());
```

- two threads read the same `end()` and both write there;
- one reallocates while the other is copying into the old buffer — a use-after-free;
- both free the old buffer.

It is worse than a wrong number

A lost increment is the friendly version. What exercise 3 actually does:

```
1 result.entries.insert(result.entries.end(), part.begin(), part.end());
```

- two threads read the same `end()` and both write there;
- one reallocates while the other is copying into the old buffer — a use-after-free;
- both free the old buffer.

Compiler error

```
corrupted size vs. prev_size while consolidating
Aborted (core dumped)
```

It is worse than a wrong number

A lost increment is the friendly version. What exercise 3 actually does:

```
1 result.entries.insert(result.entries.end(), part.begin(), part.end());
```

- two threads read the same `end()` and both write there;
- one reallocates while the other is copying into the old buffer — a use-after-free;
- both free the old buffer.

Compiler error

```
corrupted size vs. prev_size while consolidating
Aborted (core dumped)
```

Pitfall

A data race is undefined behaviour

Not “a wrong answer”. The standard says nothing at all about a program with a race in it, and the optimiser has already assumed there wasn’t one.

ThreadSanitizer

Do not rely on luck. ThreadSanitizer instruments every access to memory and records which thread did it and what it was ordered against. When it finds two accesses to one address that nothing ordered, it says so — whether or not this particular run went wrong.

```
$ g++ -fsanitize=thread -O1 -g ...    # compile and link with it
$ make tsan                           # in the exercise
```

ThreadSanitizer

Do not rely on luck. ThreadSanitizer instruments every access to memory and records which thread did it and what it was ordered against. When it finds two accesses to one address that nothing ordered, it says so — whether or not this particular run went wrong.

```
$ g++ -fsanitize=thread -O1 -g ...    # compile and link with it
$ make tsan                           # in the exercise
```

The costs: roughly 5–15× slower, five to ten times the memory, and everything has to be built with it. Which is why it is not the default and why it is worth running anyway.

ThreadSanitizer

Do not rely on luck. ThreadSanitizer instruments every access to memory and records which thread did it and what it was ordered against. When it finds two accesses to one address that nothing ordered, it says so — whether or not this particular run went wrong.

```
$ g++ -fsanitize=thread -O1 -g ...    # compile and link with it
$ make tsan                           # in the exercise
```

The costs: roughly 5–15× slower, five to ten times the memory, and everything has to be built with it. Which is why it is not the default and why it is worth running anyway.

Best practice

Run your test suite under TSan in CI, not on your laptop the day something breaks. It finds races that did not fail.

What it tells you

Exercise 1, before you fix it:

ThreadSanitizer

```
WARNING: ThreadSanitizer: data race (pid=3136796)
  Read of size 4 at 0x7ffd052ac060 by thread T1:
    #0 worker pthreads.c:17

  Previous write of size 4 at 0x7ffd052ac060 by main thread:
    #0 square_all pthreads.c:53
    #1 main test-pthreads.cc:18

  Location is stack of main thread.

  Thread T1 created by main thread at:
    #1 square_all pthreads.c:54

SUMMARY: ThreadSanitizer: data race pthreads.c:17 in worker
```

The obvious way to lock

```
1  int locked = 0;
2
3  while (locked == 1)      /* wait until it is free  */
4          ;
5  locked = 1;              /* claim it               */
6
7  /* critical section */
8
9  locked = 0;              /* release it             */
```

broken

The obvious way to lock

```
1  int locked = 0;

3  while (locked == 1)      /* wait until it is free  */
4          ;
5  locked = 1;              /* claim it               */

7  /* critical section */

9  locked = 0;              /* release it             */
```

broken

Two threads can both read `locked == 0` and then both write `1`. The test and the claim are two operations — and the whole point was to stop two operations being separable.

The obvious way to lock

```
1  int locked = 0;
2
3  while (locked == 1)      /* wait until it is free  */
4          ;
5  locked = 1;              /* claim it               */
6
7  /* critical section */
8
9  locked = 0;              /* release it             */
```

Two threads can both read `locked == 0` and then both write `1`. The test and the claim are two operations — and the whole point was to stop two operations being separable.

Key idea

We have replaced a critical section with a smaller critical section. This does not converge — it needs help from the hardware.

Compare-and-swap

One instruction, on every CPU worth the name, that does three things without anyone getting in between:

1. read a memory location;
2. compare it with the value you expected;
3. if it matches, write the new value. If not, do nothing.

Either way it tells you what it found.

Compare-and-swap

One instruction, on every CPU worth the name, that does three things without anyone getting in between:

1. read a memory location;
2. compare it with the value you expected;
3. if it matches, write the new value. If not, do nothing.

Either way it tells you what it found.

On x86 it is `cmpxchg`, with a `lock` prefix on a multiprocessor. It is not privileged — which is the difference between this and turning off interrupts.

Compare-and-swap

One instruction, on every CPU worth the name, that does three things without anyone getting in between:

1. read a memory location;
2. compare it with the value you expected;
3. if it matches, write the new value. If not, do nothing.

Either way it tells you what it found.

On x86 it is `cmpxchg`, with a `lock` prefix on a multiprocessor. It is not privileged — which is the difference between this and turning off interrupts.

Key idea

Every lock, every atomic counter and every lock-free queue in existence is built on this one idea: an indivisible read-modify-write.

A spinlock, the hard way

```
1  static bool cas(unsigned long *dest, unsigned long new_val,
2                                       unsigned long cmp_val)
3  {
4          unsigned long tmp;
5
6          asm volatile(
7              "lock cmpxchg %1, %3 \n\t"
8              : "=a" (tmp)          /* out: EAX, what was there  */
9              : "r"  (new_val),     /* in:  any register         */
10               "0"  (cmp_val),     /* in:  EAX, what we expect  */
11               "m"  (*dest)        /* in:  the memory operand   */
12             : "memory", "cc");    /* clobbers memory and flags */
13
14         return tmp == cmp_val;
15 }
16
17 void lock(lock_t *l)   { while (!cas(l, 1, 0)) ; }
18 void unlock(lock_t *l) { *l = 0; }
```


hand-rolled

A spinlock, the portable way

C11 has the same thing without the assembler:

```
1 #include <stdatomic.h>
2
3 void lock(atomic_flag *l)   { while (atomic_flag_test_and_set(l)) ; }
4 void unlock(atomic_flag *l) { atomic_flag_clear(l); }
```

A spinlock, the portable way

C11 has the same thing without the assembler:

```
#include <stdatomic.h>

void lock(atomic_flag *l)   { while (atomic_flag_test_and_set(l)) ; }
void unlock(atomic_flag *l) { atomic_flag_clear(l); }
```

And it is still the naive version: eight threads calling `test_and_set` all *write* to the same cache line, and spend their time taking it away from each other.

```
while (flag.exchange(true, std::memory_order_acquire)) {
        while (flag.load(std::memory_order_relaxed))   // read-only spin
                __builtin_ia32_pause();                // and be polite
}
```

A spinlock, the portable way

C11 has the same thing without the assembler:

```
1  #include <stdatomic.h>
2
3  void lock(atomic_flag *l)   { while (atomic_flag_test_and_set(l)) ; }
4  void unlock(atomic_flag *l) { atomic_flag_clear(l); }
```

And it is still the naive version: eight threads calling `test_and_set` all *write* to the same cache line, and spend their time taking it away from each other.

```
5  while (flag.exchange(true, std::memory_order_acquire)) {
6          while (flag.load(std::memory_order_relaxed))   // read-only spin
7                  __builtin_ia32_pause();                // and be polite
8  }
```

Key idea

Getting a spinlock to merely not be embarrassing takes a page of care — and `std::mutex` already spins before it sleeps, for free.

When an atomic is the whole answer

Sometimes there is no critical section, because the operation is already one instruction:

```
1  std::atomic<int> skipped{0};
2
3  ++skipped;                          // atomic read-modify-write
4  skipped.fetch_add(1);               // the same thing, spelled out
5  int now = skipped.load();
```

When an atomic is the whole answer

Sometimes there is no critical section, because the operation is already one instruction:

```
std::atomic<int> skipped{0};

++skipped;                          // atomic read-modify-write
skipped.fetch_add(1);               // the same thing, spelled out
int now = skipped.load();
```

No lock, no waiting, nothing to forget — and for a counter, a flag or a pointer swap, faster than a mutex.

When an atomic is the whole answer

Sometimes there is no critical section, because the operation is already one instruction:

```
1  std::atomic<int> skipped{0};
2
3  ++skipped;                          // atomic read-modify-write
4  skipped.fetch_add(1);               // the same thing, spelled out
5  int now = skipped.load();
```

No lock, no waiting, nothing to forget — and for a counter, a flag or a pointer swap, faster than a mutex.

Pitfall

Two atomic operations are not one atomic operation

It stops being the answer the moment two pieces of data have to agree — `size` and `capacity`. An atomic for a counter; a mutex for an invariant.

The part we are not going to cover

Atomics take a memory ordering, and the default is the strict one:

```
counter.fetch_add(1);                                  // seq_cst, the default
counter.fetch_add(1, std::memory_order_relaxed);   // faster, and subtle
```

The part we are not going to cover

Atomics take a memory ordering, and the default is the strict one:

```
counter.fetch_add(1);                                  // seq_cst, the default
counter.fetch_add(1, std::memory_order_relaxed);   // faster, and subtle
```

Both the compiler and the CPU reorder memory operations, and the weaker orderings let them keep doing it around your atomic. Used correctly they are worth real performance; used incorrectly they produce code that is correct on x86 and broken on ARM.

The part we are not going to cover

Atomics take a memory ordering, and the default is the strict one:

```
counter.fetch_add(1);                                  // seq_cst, the default
counter.fetch_add(1, std::memory_order_relaxed);   // faster, and subtle
```

Both the compiler and the CPU reorder memory operations, and the weaker orderings let them keep doing it around your atomic. Used correctly they are worth real performance; used incorrectly they produce code that is correct on x86 and broken on ARM.

Best practice

Leave the default. `seq_cst` is what everybody's intuition already assumes, and if you ever need `relaxed` you will need a week and a copy of the standard, not a slide.

std::mutex

A spinlock waits by *running*: the threads that did not get in burn cores producing nothing. A mutex waits by sleeping — it hands the core back and lets the kernel wake it.

```
#include <mutex>

std::mutex mtx;        // no init, no destroy, no copy, no move

void work()
{
        mtx.lock();
        /* critical section */
        mtx.unlock();
}
```

std::mutex

A spinlock waits by *running*: the threads that did not get in burn cores producing nothing. A mutex waits by sleeping — it hands the core back and lets the kernel wake it.

```
1  #include <mutex>
2
3  std::mutex mtx;        // no init, no destroy, no copy, no move
4
5  void work()
6  {
7          mtx.lock();
8          /* critical section */
9          mtx.unlock();
10 }
```

Pitfall

This slide is still wrong

Calling `lock()` and `unlock()` by hand is what exercise 4 is about.

Spin or sleep?

	spinlock	`std::mutex`
waits by	running	sleeping
costs	a core, the whole wait	one context switch per wait

Spin or sleep?

	spinlock	`std::mutex`
waits by	running	sleeping
costs	a core, the whole wait	one context switch per wait

The folklore follows: spin if the wait is shorter than a context switch, sleep otherwise.

Spin or sleep?

	spinlock	`std::mutex`
waits by	running	sleeping
costs	a core, the whole wait	one context switch per wait

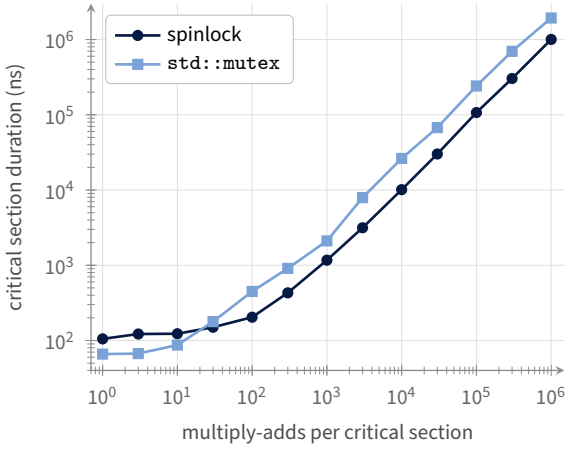
The folklore follows: spin if the wait is shorter than a context switch, sleep otherwise.

Best practice

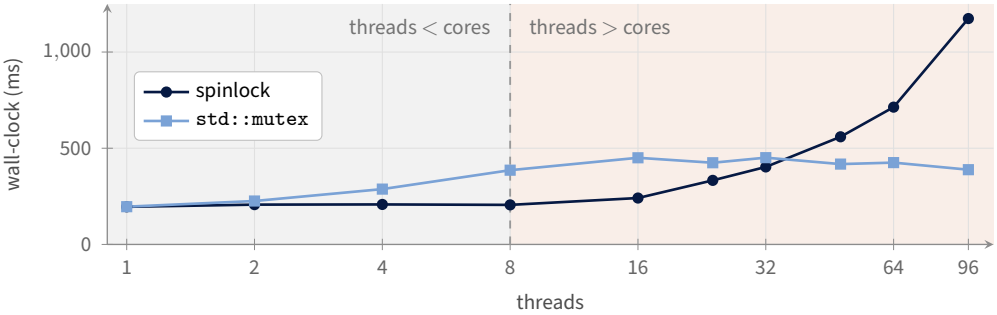
Never assume which is faster — measure. Which way it goes depends on your hardware, your libc and how contended the lock actually is, and none of those are things to guess at.

Sleep vs. spin — measured

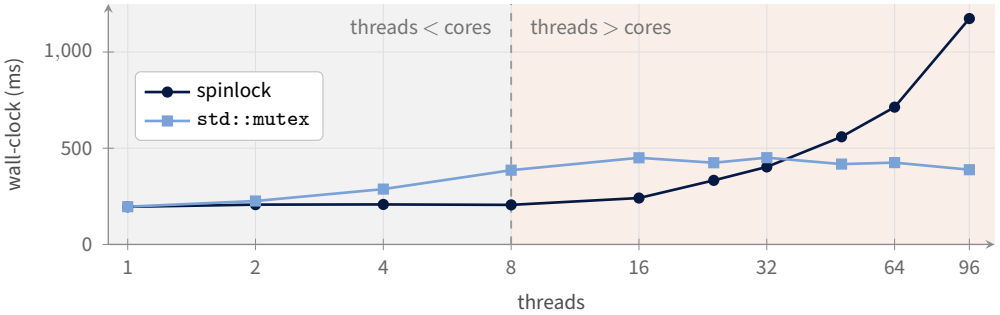
```cpp
template <typename Lock>
double run(Lock& l, int units, int sections)
{
    auto worker = [&] {
        for (int i = 0; i < sections / 8; ++i) {
            std::scoped_lock g{l};
            work(units);   // n multiply-adds
        }
    };

    auto t0 = clock::now();

    {
        std::vector<std::jthread> pool;

        for (int t = 0; t < 8; ++t)
            pool.emplace_back(worker);
    }                  // joined here

    return ms_since(t0);
}
```



Sleep vs. spin — oversubscribed



Sleep vs. spin — oversubscribed



Key idea

Lock-holder preemption: past a few threads per core the scheduler takes the core away from the thread holding the lock, and every spinner burns a quantum waiting for a thread that is not running.

How much should one lock protect?

- **Coarse** — one lock for the whole structure. Trivially correct, and every thread queues behind every other.
- **Fine** — one lock per bucket, per node, per shard. Threads working on different parts proceed at the same time, and now you have several locks, which is block 3's problem.

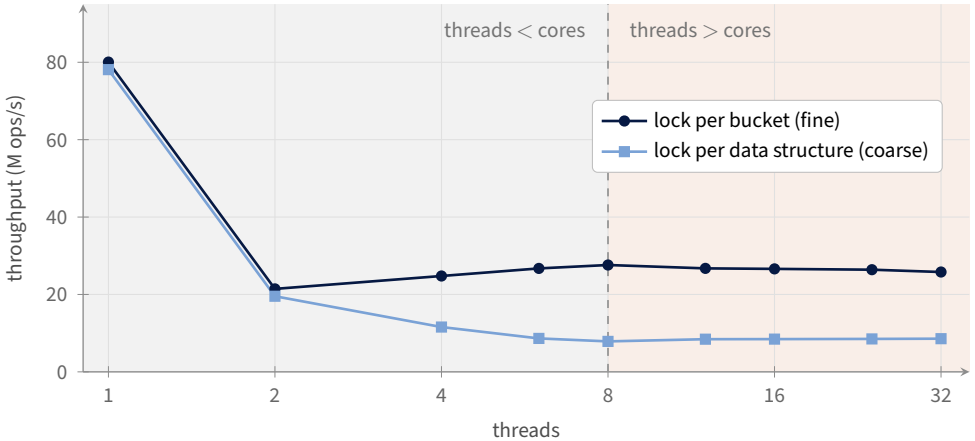
How much should one lock protect?

- **Coarse** — one lock for the whole structure. Trivially correct, and every thread queues behind every other.
- **Fine** — one lock per bucket, per node, per shard. Threads working on different parts proceed at the same time, and now you have several locks, which is block 3's problem.

Key idea

The cost of getting this wrong is not subtle, and it is not a constant factor — it changes the *shape* of the scaling curve.

The cost of getting it wrong



Exercise 3 — the shared result

In `load.cc`, `load_shared` is written for you, and every worker appends straight into one `LoadResult`.

Exercise 3 — the shared result

In `load.cc`, `load_shared` is written for you, and every worker appends straight into one `LoadResult`.

Run it. Then run it again — it is wrong in a different way each time. Then run it under the sanitizer, which does not have to get lucky.

Exercise 3 — the shared result

In `load.cc`, `load_shared` is written for you, and every worker appends straight into one `LoadResult`.

Run it. Then run it again — it is wrong in a different way each time. Then run it under the sanitizer, which does not have to get lucky.

- Give the shared state a `std::mutex` and take it around every access.
- Use `lock()` and `unlock()` explicitly this time. Exercise 4 is about why that was a bad idea.
- `read_file()` belongs *outside* the critical section. Put it inside and eight threads will politely parse one file at a time.

```
$ make test-shared && ./test-shared
$ make tsan
```

The bug, first

```
1  mtx.lock();
2
3  if (not part) {
4          result.errors.push_back(path);
5          return;                          // <- still holding the lock
6  }
7
8  append(result, *std::move(part));
9
10 mtx.unlock();
```

has an exit

The bug, first

```
1  mtx.lock();
2
3  if (not part) {
4          result.errors.push_back(path);
5          return;                          // <- still holding the lock
6  }
7
8  append(result, *std::move(part));
9
10 mtx.unlock();
```

Every other worker now waits for a thread that has already finished.

The bug, first

```
1  mtx.lock();
2
3  if (not part) {
4          result.errors.push_back(path);
5          return;                          // <- still holding the lock
6  }
7
8  append(result, *std::move(part));
9
10 mtx.unlock();
```

has an exit

Every other worker now waits for a thread that has already finished.

Key idea

The dangerous edit is not this `return`. It is the one somebody adds six months from now, to a function whose locking they did not read.

`std::lock_guard` and `std::scoped_lock`

```
1  void work()
2  {
3          std::scoped_lock const lock{mtx};
4
5          if (not part) {
6                  result.errors.push_back(path);
7                  return;              // unlocked here
8          }
9
10         append(result, *std::move(part));
11 }                                     // and here
```

`std::lock_guard` and `std::scoped_lock`

```
1  void work()
2  {
3          std::scoped_lock const lock{mtx};
4
5          if (not part) {
6                  result.errors.push_back(path);
7                  return;              // unlocked here
8          }
9
10         append(result, *std::move(part));
11 }                                     // and here
```

The constructor locks and the destructor unlocks — on every path out: returns, breaks, exceptions, all of them.

`std::lock_guard` and `std::scoped_lock`

```
1  void work()
2  {
3          std::scoped_lock const lock{mtx};
4
5          if (not part) {
6                  result.errors.push_back(path);
7                  return;                // unlocked here
8          }
9
10         append(result, *std::move(part));
11 }                                       // and here
```

The constructor locks and the destructor unlocks — on every path out: returns, breaks, exceptions, all of them.

- `std::lock_guard` (C++11) — one mutex, nothing else.
- `std::scoped_lock` (C++17) — one *or more*, and the multi-lock case is deadlock-free. Use this one.

`std::unique_lock`, when you need more

```
1  std::unique_lock lock{mtx};
2
3  cv.wait(lock, [&]{ return ready; });    // the condvar unlocks and relocks
4
5  lock.unlock();                          // release early, on purpose
6  expensive_thing_not_touching_shared_state();
7  lock.lock();                            // and take it back
```

`std::unique_lock`, when you need more

```
1  std::unique_lock lock{mtx};
2
3  cv.wait(lock, [&]{ return ready; });    // the condvar unlocks and relocks
4
5  lock.unlock();                          // release early, on purpose
6  expensive_thing_not_touching_shared_state();
7  lock.lock();                            // and take it back
```

Heavier than `scoped_lock` — it carries a flag saying whether it currently holds the mutex — and it is the only one a `condition_variable` accepts, because waiting means unlocking and relocking.

`std::unique_lock`, when you need more

```
1  std::unique_lock lock{mtx};
2
3  cv.wait(lock, [&]{ return ready; });    // the condvar unlocks and relocks
4
5  lock.unlock();                          // release early, on purpose
6  expensive_thing_not_touching_shared_state();
7  lock.lock();                            // and take it back
```

Heavier than `scoped_lock` — it carries a flag saying whether it currently holds the mutex — and it is the only one a `condition_variable` accepts, because waiting means unlocking and relocking.

Best practice

`scoped_lock` by default. `unique_lock` when you wait on a condition variable, or genuinely need to unlock early — and then say why.

The rule

You should never type `.lock()`.

The rule

You should never type `.lock()`.

Not “you should be careful to match it”. Never. If you have written `mtx.lock()`, the next line of the review is “why is this not a guard”.

The rule

You should never type `.lock()`.

Not “you should be careful to match it”. Never. If you have written `mtx.lock()`, the next line of the review is “why is this not a guard”.

The same applies to `unlock()`, to `pthread_mutex_lock` in C++ code, and to any “I’ll release it at the end” anywhere.

The rule

You should never type `.lock()`.

Not “you should be careful to match it”. Never. If you have written `mtx.lock()`, the next line of the review is “why is this not a guard”.

The same applies to `unlock()`, to `pthread_mutex_lock` in C++ code, and to any “I’ll release it at the end” anywhere.

Key idea

This is not a style preference. It is the difference between a bug that cannot happen and a bug that has not happened yet.

A guard still leaves one thing open

A guard fixes this function. It does nothing about the next one: a `mutex` declared next to the `LoadResult` it protects has nothing connecting the two but a comment, and nothing stops a caller touching one without the other.

A guard still leaves one thing open

A guard fixes this function. It does nothing about the next one: a `mutex` declared next to the `LoadResult` it protects has nothing connecting the two but a comment, and nothing stops a caller touching one without the other.

So put the data and the lock in the same object, make both private, and expose only operations that have already locked:

```
class Sink {
public:
        void add(FilePart&& part)
        {
                std::scoped_lock const lock{mtx_};
                /* ... */
        }
private:
        std::mutex mtx_;        // unreachable from outside
        LoadResult result_;     // unreachable without it
};
```

Exercise 4 — a lock nobody can forget

`load_guarded` in `load.cc` is exercise 3 again, with the error handling real code has: two of the files do not exist.

Exercise 4 — a lock nobody can forget

`load_guarded` in `load.cc` is exercise 3 again, with the error handling real code has: two of the files do not exist.

`./test-guard` runs it with a deadline, because this one does not fail — it hangs.

Exercise 4 — a lock nobody can forget

`load_guarded` in `load.cc` is exercise 3 again, with the error handling real code has: two of the files do not exist.

`./test-guard` runs it with a deadline, because this one does not fail — it hangs.

- **4a:** find the way out of the critical section that does not pass the `unlock`, and use `std::scoped_lock` so that it cannot matter.
- **4b:** then notice that 4a only fixed today. Move the state and its mutex into `Sink` in `sink.h`, so the call site has nothing left to get wrong.
- `Sink` is used again by exercise 5's worker pool, so it has to survive several threads calling `add()` at once — which the test checks.

```
$ make test-guard && ./test-guard
```

BLOCK 3 / 3

Coordination and results

- Semaphores and condition variables
- Deadlocks
- Promises and futures
- `std::async`
- Wrapping up

Exercises 5 and 6

Semaphores

A counter, plus a queue of threads waiting on it.

- **down** (wait, `P`): decrement. If it would go below zero, sleep until somebody raises it.
- **up** (post, `V`): increment, and wake one sleeper.

Semaphores

A counter, plus a queue of threads waiting on it.

- **down** (wait, `P`): decrement. If it would go below zero, sleep until somebody raises it.
- **up** (post, `V`): increment, and wake one sleeper.

The count is what makes it more than a flag: initialise it to the number of free slots and `down` means “claim one, or wait for one”.

Semaphores

A counter, plus a queue of threads waiting on it.

- **down** (wait, `P`): decrement. If it would go below zero, sleep until somebody raises it.
- **up** (post, `V`): increment, and wake one sleeper.

The count is what makes it more than a flag: initialise it to the number of free slots and `down` means “claim one, or wait for one”.

Initialise it to 1 and you have a mutex — except that a semaphore has no owner, and any thread may post one that another took. Which is what makes it a good signal and a bad mutex.

Semaphores

A counter, plus a queue of threads waiting on it.

- **down** (wait, `P`): decrement. If it would go below zero, sleep until somebody raises it.
- **up** (post, `V`): increment, and wake one sleeper.

The count is what makes it more than a flag: initialise it to the number of free slots and `down` means “claim one, or wait for one”.

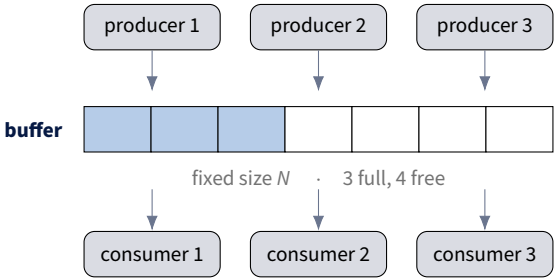
Initialise it to 1 and you have a mutex — except that a semaphore has no owner, and any thread may post one that another took. Which is what makes it a good signal and a bad mutex.

Pitfall

No owner, no help

A semaphore cannot be RAII-wrapped in any useful way, cannot detect recursive use, and cannot tell you who is holding it.

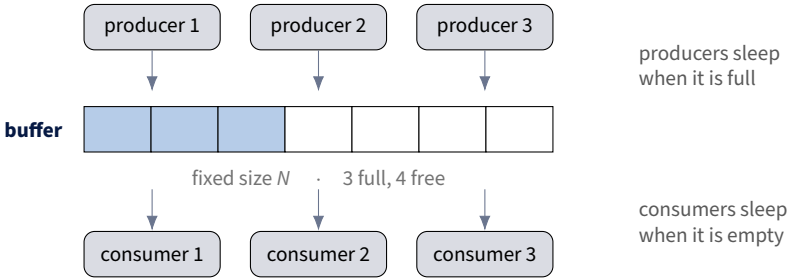
The producer-consumer problem



producers sleep when it is full

consumers sleep when it is empty

The producer-consumer problem



Any number of each, in any ratio. The buffer is fixed size on purpose: an unbounded one just moves the problem to the moment you run out of memory.

The classical solution

Two semaphores, counting the two things that can run out:

Producer

```
sem_t empty;   /* init N */
sem_t full;    /* init 0 */

while (1) {
    item = produce();

    sem_wait(&empty);
    put(item);
    sem_post(&full);
}
```

Consumer

```
while (1) {
    sem_wait(&full);
    item = take();
    sem_post(&empty);

    consume(item);
}
```

The classical solution

Two semaphores, counting the two things that can run out:

Producer

```
sem_t empty;   /* init N */
sem_t full;    /* init 0 */

while (1) {
    item = produce();

    sem_wait(&empty);
    put(item);
    sem_post(&full);
}
```

Consumer

```
while (1) {
    sem_wait(&full);
    item = take();
    sem_post(&empty);

    consume(item);
}
```

Pitfall

With more than one of each, this is not finished

`put` and `take` still touch the buffer, so they need a mutex too. Semaphores count; they do not exclude.

Condition variables

A semaphore mixes three ideas: a counter, a sleep queue, and a signal. You can only wait for one condition with it — “the count is above zero”.

Condition variables

A semaphore mixes three ideas: a counter, a sleep queue, and a signal. You can only wait for one condition with it — “the count is above zero”.

A condition variable separates them out. It is a sleep queue and nothing else: no counter, no state, no memory of anything.

- `wait` — sleep on it until somebody signals.
- `notify_one` / `notify_all` — wake one, or all of them.

Condition variables

A semaphore mixes three ideas: a counter, a sleep queue, and a signal. You can only wait for one condition with it — “the count is above zero”.

A condition variable separates them out. It is a sleep queue and nothing else: no counter, no state, no memory of anything.

- `wait` — sleep on it until somebody signals.
- `notify_one` / `notify_all` — wake one, or all of them.

Because it holds no state, the condition you are waiting for lives in your own variables, protected by your own mutex — and *you* check it.

Condition variables

A semaphore mixes three ideas: a counter, a sleep queue, and a signal. You can only wait for one condition with it — “the count is above zero”.

A condition variable separates them out. It is a sleep queue and nothing else: no counter, no state, no memory of anything.

- `wait` — sleep on it until somebody signals.
- `notify_one` / `notify_all` — wake one, or all of them.

Because it holds no state, the condition you are waiting for lives in your own variables, protected by your own mutex — and *you* check it.

Key idea

A condition variable is not the condition. It is only the doorbell.

Why it needs the mutex

Waiting is two steps — check the condition, then go to sleep — and if anything happens in between, the wakeup is lost and you sleep forever.

Waiter

```
if (!ready)          // 1. check


                     //
    sleep();         // 3. sleep -- forever
```

Signaller

```
                     // 2. it happens
ready = true;
notify();            //    ... to nobody
```

Why it needs the mutex

Waiting is two steps — check the condition, then go to sleep — and if anything happens in between, the wakeup is lost and you sleep forever.

Waiter

```
if (!ready)          // 1. check


                     //
    sleep();         // 3. sleep -- forever
```

Signaller

```
                     // 2. it happens
ready = true;
notify();            //    ... to nobody
```

So `wait` takes the mutex from you and releases it *as part of* going to sleep, indivisibly — and takes it again before returning. That is why the signaller cannot slip through the gap, and why `wait` needs a `std::unique_lock` rather than a `scoped_lock`.

Condition variables in practice

POSIX

```
pthread_mutex_t m;
pthread_cond_t  cv;

pthread_mutex_lock(&m);
while (!ready)
    pthread_cond_wait(&cv, &m);
/* ready is true, m is held */
pthread_mutex_unlock(&m);


pthread_mutex_lock(&m);
ready = 1;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&m);
```

C++

```
std::mutex              m;
std::condition_variable cv;

{
    std::unique_lock lock{m};
    cv.wait(lock, []{ return ready; });
    /* ready is true, m is held */
}


{
    std::scoped_lock const lock{m};
    ready = true;
}
cv.notify_one();
```

Three rules, and why

1. **Hold the mutex while you change the condition.** Otherwise you are back at the lost wakeup on the previous slide.

Three rules, and why

1. **Hold the mutex while you change the condition.** Otherwise you are back at the lost wakeup on the previous slide.
2. **Always use the predicate form.** `wait` may return with nobody having notified — a *spurious wakeup* — and it is also what makes two woken waiters correct when one item arrived.

Three rules, and why

1. **Hold the mutex while you change the condition.** Otherwise you are back at the lost wakeup on the previous slide.
2. **Always use the predicate form.** `wait` may return with nobody having notified — a *spurious wakeup* — and it is also what makes two woken waiters correct when one item arrived.
3. **Notify after releasing the mutex.** A thread woken while the mutex is still held goes straight back to sleep, on the mutex.

Three rules, and why

1. **Hold the mutex while you change the condition.** Otherwise you are back at the lost wakeup on the previous slide.
2. **Always use the predicate form.** `wait` may return with nobody having notified — a *spurious wakeup* — and it is also what makes two woken waiters correct when one item arrived.
3. **Notify after releasing the mutex.** A thread woken while the mutex is still held goes straight back to sleep, on the mutex.

`notify_one` when any single waiter will do; `notify_all` when the change affects everybody — a shutdown, for instance.

Three rules, and why

1. **Hold the mutex while you change the condition.** Otherwise you are back at the lost wakeup on the previous slide.
2. **Always use the predicate form.** `wait` may return with nobody having notified — a *spurious wakeup* — and it is also what makes two woken waiters correct when one item arrived.
3. **Notify after releasing the mutex.** A thread woken while the mutex is still held goes straight back to sleep, on the mutex.

`notify_one` when any single waiter will do; `notify_all` when the change affects everybody — a shutdown, for instance.

Pitfall

`notify_one` on a shutdown

Four consumers asleep, one gets woken, three wait forever, and the join never returns. This is the most common way exercise 5 hangs.

Exercise 5 — the work queue

In `queue.h`:

```
template <typename T> class Queue {
        explicit Queue(std::size_t capacity);
        bool             push(T value);   // false once closed
        std::optional<T> pop();           // nullopt once closed and drained
        void             close();
};
```

One mutex, two condition variables — one for empty, one for full.

Exercise 5 — the work queue

In `queue.h`:

```
template <typename T> class Queue {
        explicit Queue(std::size_t capacity);
        bool             push(T value);   // false once closed
        std::optional<T> pop();           // nullopt once closed and drained
        void             close();
};
```

One mutex, two condition variables — one for empty, one for full.

- Every predicate must mention `closed_`, or a thread already asleep stays asleep when the queue closes.
- Closed is not finished: `pop` keeps handing out what is queued.
- `close()` wakes *everybody*, on both condition variables.

Exercise 5 — the work queue

In `queue.h`:

```
template <typename T> class Queue {
        explicit Queue(std::size_t capacity);
        bool             push(T value);   // false once closed
        std::optional<T> pop();           // nullopt once closed and drained
        void             close();
};
```

One mutex, two condition variables — one for empty, one for full.

- Every predicate must mention `closed_`, or a thread already asleep stays asleep when the queue closes.
- Closed is not finished: `pop` keeps handing out what is queued.
- `close()` wakes *everybody*, on both condition variables.

`load_pool` is written for you, and already uses it.

```
$ make test-queue && ./test-queue
```

What a deadlock is

Two or more threads, each waiting for something only another one of them can release. None of them will ever run again, and none of them will ever give up what it is holding, because it cannot get past the line it is on.

What a deadlock is

Two or more threads, each waiting for something only another one of them can release. None of them will ever run again, and none of them will ever give up what it is holding, because it cannot get past the line it is on.

From the outside it looks like nothing. No crash, no error, no log line — the process is still there, using no CPU at all.

What a deadlock is

Two or more threads, each waiting for something only another one of them can release. None of them will ever run again, and none of them will ever give up what it is holding, because it cannot get past the line it is on.

From the outside it looks like nothing. No crash, no error, no log line — the process is still there, using no CPU at all.

Which is why it usually gets diagnosed as “it’s slow”.

What a deadlock is

Two or more threads, each waiting for something only another one of them can release. None of them will ever run again, and none of them will ever give up what it is holding, because it cannot get past the line it is on.

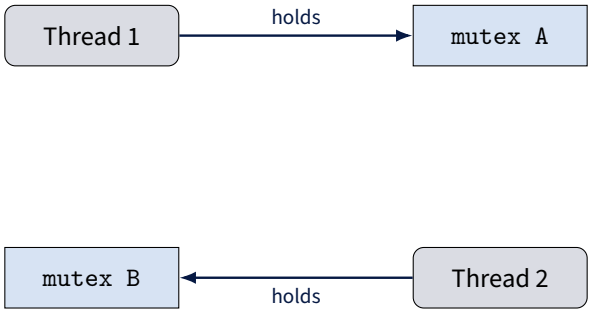
From the outside it looks like nothing. No crash, no error, no log line — the process is still there, using no CPU at all.

Which is why it usually gets diagnosed as “it’s slow”.

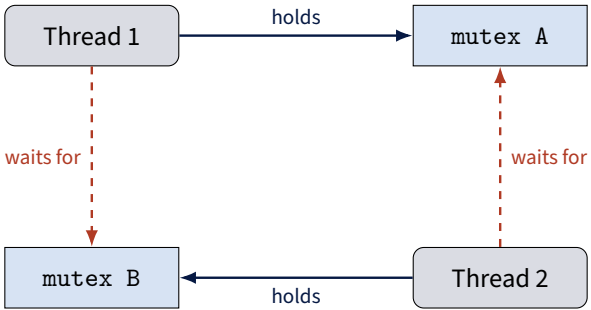
Key idea

A deadlock is not a timing accident you got unlucky with. It is a property of the code: the cycle was always there, and the schedule only decides which day you find out.

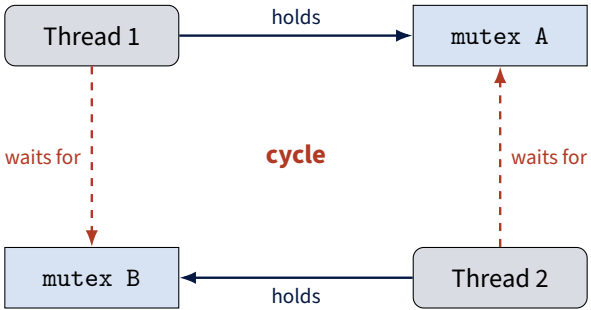
The cycle



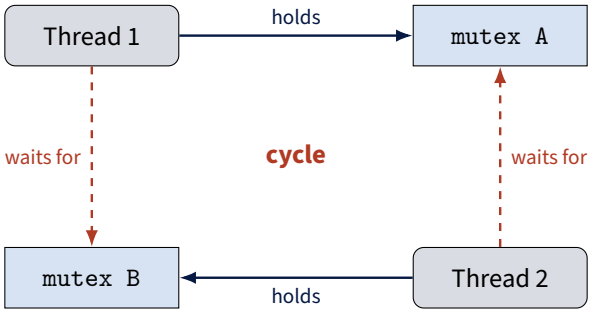
The cycle


Thread 1
holds
mutex A
waits for
waits for
mutex B
holds
Thread 2

The cycle


Thread 1
holds
mutex A
waits for
cycle
waits for
mutex B
holds
Thread 2

The cycle



Draw who holds what and who wants what. If the arrows form a loop, nothing in that loop will ever move — and if they never form a loop, a deadlock is impossible.

Four conditions, four ways out

A deadlock needs all four of these at once (Coffman, 1971). Break any one and it cannot happen:

Condition	Breaking it
mutual exclusion	don’t share — copy, or give each thread its own
hold and wait	take every lock you need at once
no preemption	`try_lock`, and back off if you fail
circular wait	a global order on locks

Four conditions, four ways out

A deadlock needs all four of these at once (Coffman, 1971). Break any one and it cannot happen:

Condition	Breaking it
mutual exclusion	don't share — copy, or give each thread its own
hold and wait	take every lock you need at once
no preemption	`try_lock`, and back off if you fail
circular wait	a global order on locks

In practice the last two are what you use, and the first is what you should have done instead.

Four conditions, four ways out

A deadlock needs all four of these at once (Coffman, 1971). Break any one and it cannot happen:

Condition	Breaking it
mutual exclusion	don't share — copy, or give each thread its own
hold and wait	take every lock you need at once
no preemption	`try_lock`, and back off if you fail
circular wait	a global order on locks

In practice the last two are what you use, and the first is what you should have done instead.

Key idea

The cheapest deadlock to fix is the one where you notice the two threads did not need to share that data in the first place.

Bug 1: the lock that is never released

```
1  lock(&table_lock);
2
3  place = find_free_table();
4
5  if (place < 0) {
6          printf("thread %lu gives up\n", pthread_self());
7          return NULL;                    /* <- lock still held */
8  }
9
10 unlock(&table_lock);
```

lock1.c

Bug 1: the lock that is never released

```
1  lock(&table_lock);
2
3  place = find_free_table();
4
5  if (place < 0) {
6          printf("thread %lu gives up\n", pthread_self());
7          return NULL;                    /* <- lock still held */
8  }
9
10 unlock(&table_lock);
```


lock1.c

One thread leaves early; everybody else waits for a lock whose owner has terminated. Strictly not a deadlock — no cycle, just an abandoned lock — and indistinguishable from one at three in the morning.

Bug 1: the lock that is never released

```
1  lock(&table_lock);
2
3  place = find_free_table();
4
5  if (place < 0) {
6          printf("thread %lu gives up\n", pthread_self());
7          return NULL;                    /* <- lock still held */
8  }
9
10 unlock(&table_lock);
```


lock1.c

One thread leaves early; everybody else waits for a lock whose owner has terminated. Strictly not a deadlock — no cycle, just an abandoned lock — and indistinguishable from one at three in the morning.

In C++ it cannot happen: the guard's destructor runs on the way out.

Bug 2: two locks, two orders

Person 1

```
while (1) {
    lock(&paper);
    printf("got paper\n");

    lock(&printer);
    printf("got printer\n");

    print_document();

    unlock(&printer);
    unlock(&paper);
}
```

Person 2

```
while (1) {
    lock(&printer);
    printf("got printer\n");

    lock(&paper);
    printf("got paper\n");

    print_document();

    unlock(&paper);
    unlock(&printer);
}
```

Bug 2: two locks, two orders

Person 1

```
while (1) {
    lock(&paper);
    printf("got paper\n");

    lock(&printer);
    printf("got printer\n");

    print_document();

    unlock(&printer);
    unlock(&paper);
}
```

Person 2

```
while (1) {
    lock(&printer);
    printf("got printer\n");

    lock(&paper);
    printf("got paper\n");

    print_document();

    unlock(&paper);
    unlock(&printer);
}
```

`lock2.c`. It prints happily for a while — most of the time one of them gets both before the other starts — and then one day person 1 has the paper and person 2 has the printer.

Bug 2: two locks, two orders

Person 1

```
while (1) {
    lock(&paper);
    printf("got paper\n");

    lock(&printer);
    printf("got printer\n");

    print_document();

    unlock(&printer);
    unlock(&paper);
}
```

Person 2

```
while (1) {
    lock(&printer);
    printf("got printer\n");

    lock(&paper);
    printf("got paper\n");

    print_document();

    unlock(&paper);
    unlock(&printer);
}
```

`lock2.c`. It prints happily for a while — most of the time one of them gets both before the other starts — and then one day person 1 has the paper and person 2 has the printer. That is the whole bug: two functions that disagreed about which lock comes first.

Bug 3: waiting for yourself

```
1  void callme(void)
2  {
3          lock(&lock2);
4          printf(".\n");
5          callme();               /* and here we go again */
6          unlock(&lock2);
7  }
```


lock3.c

Bug 3: waiting for yourself

```
1  void callme(void)
2  {
3          lock(&lock2);
4          printf(".\n");
5          callme();               /* and here we go again */
6          unlock(&lock2);
7  }
```


lock3.c

One thread, one lock, nobody else involved: the second acquisition waits for the first, held by the thread doing the waiting. Rarely this obvious — usually it is one locked method calling another, three files apart.

Bug 3: waiting for yourself

```
1  void callme(void)
2  {
3          lock(&lock2);
4          printf(".\n");
5          callme();               /* and here we go again */
6          unlock(&lock2);
7  }
```

lock3.c

One thread, one lock, nobody else involved: the second acquisition waits for the first, held by the thread doing the waiting. Rarely this obvious — usually it is one locked method calling another, three files apart.

Pitfall

Locking a `std::mutex` twice is undefined

Not “it deadlocks”. Undefined. In practice it deadlocks.

Prevention: an order, and a way to not need it

The rule. Decide, once, that the store's lock always comes before the index's. Write it down. Apply it everywhere, without exception.

Prevention: an order, and a way to not need it

The rule. Decide, once, that the store's lock always comes before the index's. Write it down. Apply it everywhere, without exception.

The tool. `std::scoped_lock` takes any number of mutexes as one step: acquire what you can, release everything if you cannot get the rest, retry.

```
1 void merge(Store& store, SourceIndex& index, ...)
2 {
3         std::scoped_lock const lock{store.mutex(), index.mutex()};
4         /* both held, in no particular order, no cycle possible */
5 }
```

Prevention: an order, and a way to not need it

The rule. Decide, once, that the store's lock always comes before the index's. Write it down. Apply it everywhere, without exception.

The tool. `std::scoped_lock` takes any number of mutexes as one step: acquire what you can, release everything if you cannot get the rest, retry.

```
1 void merge(Store& store, SourceIndex& index, ...)
2 {
3         std::scoped_lock const lock{store.mutex(), index.mutex()};
4         /* both held, in no particular order, no cycle possible */
5 }
```

Best practice

Do both. The algorithm makes today correct; the written-down order is what survives the next person adding a third lock.

Exercise 6 — two locks

First, the museum. Three programs, three different ways to stop:

```
$ make -C deadlocks && ./deadlocks/lock1
```

Work out which bug is which, and which one of the three is a cycle.

Exercise 6 — two locks

First, the museum. Three programs, three different ways to stop:

```
$ make -C deadlocks && ./deadlocks/lock1
```

Work out which bug is which, and which one of the three is a cycle.

Then `index.cc`: `merge` takes the store's lock then the index's, `rebuild` the other way round. The single-threaded checks pass; the concurrent one stops dead.

Exercise 6 — two locks

First, the museum. Three programs, three different ways to stop:

```
$ make -C deadlocks && ./deadlocks/lock1
```

Work out which bug is which, and which one of the three is a cycle.

Then `index.cc`: `merge` takes the store's lock then the index's, `rebuild` the other way round. The single-threaded checks pass; the concurrent one stops dead.

- Fix it twice — first by imposing an order, then with one `std::scoped_lock` over both.
- Mind the `_locked` suffix: calling a locking member while holding the lock is bug 3.

```
$ make test-index && ./test-index
```

Two things a thread cannot do

It cannot return anything. The function it runs returns `void`, so every result has to come out through a captured reference — which is shared mutable state, which needs a lock, which is the whole of block 2 for what should have been a return value.

Two things a thread cannot do

It cannot return anything. The function it runs returns `void`, so every result has to come out through a captured reference — which is shared mutable state, which needs a lock, which is the whole of block 2 for what should have been a return value.

It cannot fail. An exception that escapes the thread function does not propagate to whoever started the thread. There is nowhere for it to go:

Compiler error

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  cannot read app.log
Aborted (core dumped)
```

Two things a thread cannot do

It cannot return anything. The function it runs returns `void`, so every result has to come out through a captured reference — which is shared mutable state, which needs a lock, which is the whole of block 2 for what should have been a return value.

It cannot fail. An exception that escapes the thread function does not propagate to whoever started the thread. There is nowhere for it to go:

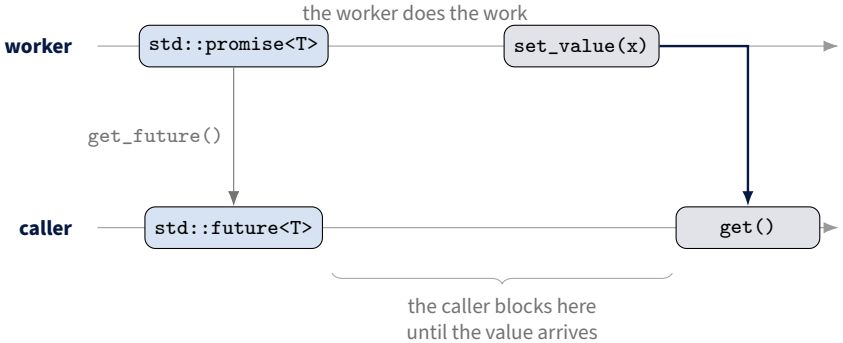
Compiler error

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  cannot read app.log
Aborted (core dumped)
```

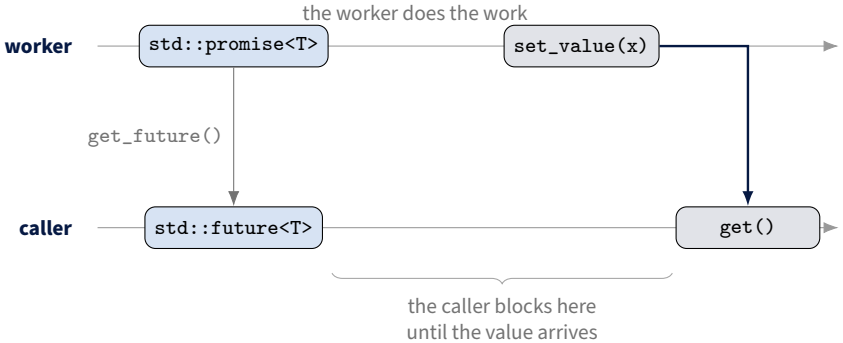
Key idea

A future is the missing return path. It carries the value *or* the exception, whichever happened, to whoever asks for it.

One channel, two ends



One channel, two ends



The promise is the writing end and the future is the reading end. They are created together, and each may be moved into whichever thread needs it.

Using them directly

```
std::promise<int> promise;
std::future<int>  result = promise.get_future();

std::jthread worker{[p = std::move(promise)]() mutable {
        p.set_value(compute());          // or p.set_exception(...)
}};

int value = result.get();                // blocks until it is there
```

Using them directly

```
1  std::promise<int> promise;
2  std::future<int>  result = promise.get_future();
3
4  std::jthread worker{[p = std::move(promise)]() mutable {
5          p.set_value(compute());          // or p.set_exception(...)
6  }};
7
8  int value = result.get();                // blocks until it is there
```

`get()` blocks until the value exists, and may be called *once*.

Using them directly

```
1  std::promise<int> promise;
2  std::future<int>  result = promise.get_future();
3
4  std::jthread worker{[p = std::move(promise)]() mutable {
5          p.set_value(compute());          // or p.set_exception(...)
6  }};
7
8  int value = result.get();                // blocks until it is there
```

`get()` blocks until the value exists, and may be called *once*.

Pitfall

A promise that is never set

The future does not hang: `get()` throws `broken_promise`. Merciful, and still a bug.

Exceptions take the same path

In the worker

```
try {
    p.set_value(read(path));
} catch (...) {
    p.set_exception(
        std::current_exception());
}
```

In the caller

```
try {
    auto e = result.get();
} catch (std::exception const& e) {
    std::println("failed: {}",
                 e.what());
}
```

Exceptions take the same path

In the worker

```
try {
    p.set_value(read(path));
} catch (...) {
    p.set_exception(
        std::current_exception());
}
```

In the caller

```
try {
    auto e = result.get();
} catch (std::exception const& e) {
    std::println("failed: {}",
                 e.what());
}
```

Stored, carried across the thread boundary, and re-thrown out of `get()` — with the original type and the original message.

Exceptions take the same path

In the worker

```
try {
    p.set_value(read(path));
} catch (...) {
    p.set_exception(
        std::current_exception());
}
```

In the caller

```
try {
    auto e = result.get();
} catch (std::exception const& e) {
    std::println("failed: {}",
                 e.what());
}
```

Stored, carried across the thread boundary, and re-thrown out of `get()` — with the original type and the original message.

Key idea

This is the half of futures that matters most. Error handling across threads without it means inventing your own error channel, and everybody who invents one forgets a case.

Waiting without committing

```
if (result.wait_for(100ms) == std::future_status::ready)
        use(result.get());
else
        std::println("still working...");
```

Waiting without committing

```
if (result.wait_for(100ms) == std::future_status::ready)
        use(result.get());
else
        std::println("still working...");
```

Three statuses: `ready`, `timeout`, and `deferred` — the last one meaning “this has not started and will not until you call `get()`”. We will come back to that on the next slide, because it catches people.

Waiting without committing

```
if (result.wait_for(100ms) == std::future_status::ready)
        use(result.get());
else
        std::println("still working...");
```

Three statuses: `ready`, `timeout`, and `deferred` — the last one meaning “this has not started and will not until you call `get()`”. We will come back to that on the next slide, because it catches people.

`wait()` blocks without taking the value, so you can check readiness and still `get()` later.

Waiting without committing

```
if (result.wait_for(100ms) == std::future_status::ready)
        use(result.get());
else
        std::println("still working...");
```

Three statuses: `ready`, `timeout`, and `deferred` — the last one meaning “this has not started and will not until you call `get()`”. We will come back to that on the next slide, because it catches people.

`wait()` blocks without taking the value, so you can check readiness and still `get()` later.

Pitfall

Polling `wait_for(0s)` in a loop

That is a busy-wait with extra steps. If you want to do other work while waiting, you wanted a queue, not a future.

std::async

A thread, a promise, a future and the exception handling, in one call:

```
1 std::future<FilePart> result =
2         std::async(std::launch::async, read_file, path);
3
4 auto part = result.get();      // value, or the exception it threw
```

std::async

A thread, a promise, a future and the exception handling, in one call:

```
1 std::future<FilePart> result =
2         std::async(std::launch::async, read_file, path);
3
4 auto part = result.get();      // value, or the exception it threw
```

That is the whole of the previous section, done for you. For “run this and give me the answer later”, it is the right tool and there is nothing to write.

std::async

A thread, a promise, a future and the exception handling, in one call:

```
1 std::future<FilePart> result =
2         std::async(std::launch::async, read_file, path);
3
4 auto part = result.get();      // value, or the exception it threw
```

That is the whole of the previous section, done for you. For “run this and give me the answer later”, it is the right tool and there is nothing to write.

It has two traps. Both of them are on the next two slides, and both of them make your program quietly not concurrent.

std::async

A thread, a promise, a future and the exception handling, in one call:

```
1 std::future<FilePart> result =
2         std::async(std::launch::async, read_file, path);
3
4 auto part = result.get();      // value, or the exception it threw
```

That is the whole of the previous section, done for you. For “run this and give me the answer later”, it is the right tool and there is nothing to write.

It has two traps. Both of them are on the next two slides, and both of them make your program quietly not concurrent.

Best practice

Reach for `std::async` before `std::thread` when what you want is a result.
Reach for a pool when you have more work than cores.

Trap 1: the policy

```
1 std::async(f);                          // async | deferred -- either!
2 std::async(std::launch::async, f);      // definitely another thread
3 std::async(std::launch::deferred, f);   // definitely not
```

Trap 1: the policy

```
1 std::async(f);                          // async | deferred -- either!
2 std::async(std::launch::async, f);      // definitely another thread
3 std::async(std::launch::deferred, f);   // definitely not
```

With the default policy the implementation may choose `deferred`, which means: do not run it at all until somebody calls `get()`, and then run it *on that thread*.

Trap 1: the policy

```
std::async(f);                          // async | deferred -- either!
std::async(std::launch::async, f);      // definitely another thread
std::async(std::launch::deferred, f);   // definitely not
```

With the default policy the implementation may choose `deferred`, which means: do not run it at all until somebody calls `get()`, and then run it *on that thread*.

So a loop that starts ten tasks and then gathers them runs all ten, one after another, inside the gathering loop. The code looks parallel, the profile does not, and nothing warns you.

Trap 1: the policy

```
std::async(f);                          // async | deferred -- either!
std::async(std::launch::async, f);      // definitely another thread
std::async(std::launch::deferred, f);   // definitely not
```

With the default policy the implementation may choose `deferred`, which means: do not run it at all until somebody calls `get()`, and then run it *on that thread*.

So a loop that starts ten tasks and then gathers them runs all ten, one after another, inside the gathering loop. The code looks parallel, the profile does not, and nothing warns you.

Best practice

Always pass `std::launch::async` explicitly. If you did not want another thread you did not want `async`.

Trap 2: the destructor

```
1 std::async(std::launch::async, f);   // temporary future, dies here
2 g();                                 // so this waits for f
```


not async

Trap 2: the destructor

```
1 std::async(std::launch::async, f);   // temporary future, dies here
2 g();                                 // so this waits for f
```

not async

`std::async`'s future blocks in its destructor until the task finishes, so ignoring it gives you a synchronous call. No other future does this.

```
3 auto pending = std::async(std::launch::async, f);   // keep it
4 g();                                                // now they overlap
5 pending.get();
```

Trap 2: the destructor

```
1 std::async(std::launch::async, f);   // temporary future, dies here
2 g();                                 // so this waits for f
```

not async

`std::async`'s future blocks in its destructor until the task finishes, so ignoring it gives you a synchronous call. No other future does this.

```
3 auto pending = std::async(std::launch::async, f);   // keep it
4 g();                                                // now they overlap
5 pending.get();
```

Pitfall

The same applies to a vector of futures

It blocks in *order*, at the closing brace.

Exercise 7 — take this one home

In `parallel.cc`:

```
std::future<FileSummary> summarise_async(std::string path);
std::vector<FileSummary> summarise_all(std::span<std::string const> paths);
void                     FirstFailure::report(std::string path);
```

Exercise 7 — take this one home

In `parallel.cc`:

```
std::future<FileSummary> summarise_async(std::string path);
std::vector<FileSummary> summarise_all(std::span<std::string const> paths);
void                     FirstFailure::report(std::string path);
```

- Spell out `std::launch::async`, and capture the path *by value* — the caller's string may be gone by the time it runs.
- An unreadable file throws inside the task; the test checks it comes back out of `get()` rather than killing the process.
- `FirstFailure` is a promise, not a return value: several workers may fail at once and exactly one report must win.

Exercise 7 — take this one home

In `parallel.cc`:

```
std::future<FileSummary> summarise_async(std::string path);
std::vector<FileSummary> summarise_all(std::span<std::string const> paths);
void                     FirstFailure::report(std::string path);
```

- Spell out `std::launch::async`, and capture the path *by value* — the caller's string may be gone by the time it runs.
- An unreadable file throws inside the task; the test checks it comes back out of `get()` rather than killing the process.
- `FirstFailure` is a promise, not a return value: several workers may fail at once and exactly one report must win.

The one we do not type together. The worked answer is a commit away:

```
$ make test-parallel && ./test-parallel
$ git diff main ex7
```

Choosing the tool

What you have	What to reach for
nothing shared	nothing. Stop here
a counter, a flag, a pointer swap	`std::atomic`
shared data with an invariant	`std::mutex` + `scoped_lock`
wait until something becomes true	`std::condition_variable`
wait for one value from one task	`std::future`
a stream of work, more items than cores	a queue and a fixed pool

Choosing the tool

What you have	What to reach for
nothing shared	nothing. Stop here
a counter, a flag, a pointer swap	`std::atomic`
shared data with an invariant	`std::mutex` + `scoped_lock`
wait until something becomes true	`std::condition_variable`
wait for one value from one task	`std::future`
a stream of work, more items than cores	a queue and a fixed pool

The first row is the one people skip. Most of the locking in most programs exists because two threads were given the same object when they could have had one each.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.
- Every mutex is private to the object it protects, and it is written down what it protects.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.
- Every mutex is private to the object it protects, and it is written down what it protects.
- Wherever two locks are held at once, there is one documented order — or a `std::scoped_lock` over both.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.
- Every mutex is private to the object it protects, and it is written down what it protects.
- Wherever two locks are held at once, there is one documented order — or a `std::scoped_lock` over both.
- Every `wait` has a predicate, and every predicate mentions the shutdown condition.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.
- Every mutex is private to the object it protects, and it is written down what it protects.
- Wherever two locks are held at once, there is one documented order — or a `std::scoped_lock` over both.
- Every `wait` has a predicate, and every predicate mentions the shutdown condition.
- Every queue is bounded, and closing it wakes everybody.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.
- Every mutex is private to the object it protects, and it is written down what it protects.
- Wherever two locks are held at once, there is one documented order — or a `std::scoped_lock` over both.
- Every `wait` has a predicate, and every predicate mentions the shutdown condition.
- Every queue is bounded, and closing it wakes everybody.
- Every thread is joined. `jthread` counts.

Before you ship it

- It has been run under `-fsanitize=thread`, with the real test suite, more than once.
- There is no `.lock()` anywhere. Every lock is a guard.
- Every mutex is private to the object it protects, and it is written down what it protects.
- Wherever two locks are held at once, there is one documented order — or a `std::scoped_lock` over both.
- Every `wait` has a predicate, and every predicate mentions the shutdown condition.
- Every queue is bounded, and closing it wakes everybody.
- Every thread is joined. `jthread` counts.
- Somebody measured it, and the speedup is a number, not a hope.

What we did not cover

- The memory model in earnest: acquire/release, `relaxed`, fences, and why ARM is not x86.
- Lock-free data structures, hazard pointers, RCU.
- `std::latch`, `std::barrier` — C++20, and genuinely useful for phase-structured work.
- Coroutines, unless we have time for the backup slides.
- Senders/receivers (C++26), which is where the standard is heading.
- Anything about GPUs, message passing, or more than one machine.
- Thread pools as such, the parallel algorithms (`std::execution::par`), and `std::shared_mutex` — casualties of a six-hour day rather than of taste.

What we did not cover

- The memory model in earnest: acquire/release, `relaxed`, fences, and why ARM is not x86.
- Lock-free data structures, hazard pointers, RCU.
- `std::latch`, `std::barrier` — C++20, and genuinely useful for phase-structured work.
- Coroutines, unless we have time for the backup slides.
- Senders/receivers (C++26), which is where the standard is heading.
- Anything about GPUs, message passing, or more than one machine.
- Thread pools as such, the parallel algorithms (`std::execution::par`), and `std::shared_mutex` — casualties of a six-hour day rather than of taste.

Worth reading next: Williams, *C++ Concurrency in Action*; the `cppreference` pages on `<thread>`, `<mutex>` and `<atomic>`; and `man pthreads` for what is actually underneath.

One thing to take away

Rule #1 was the whole lecture.

Never assume anything about the order of execution or about timing. Every bug today was somebody assuming one anyway:

- that the thread would read `i` before the loop changed it;
- that two increments would not overlap;
- that the unlock would be reached;
- that both functions would take the locks in the same order;
- that the consumer would be waiting when the signal arrived.

One thing to take away

Rule #1 was the whole lecture.

Never assume anything about the order of execution or about timing. Every bug today was somebody assuming one anyway:

- that the thread would read `i` before the loop changed it;
- that two increments would not overlap;
- that the unlock would be reached;
- that both functions would take the locks in the same order;
- that the consumer would be waiting when the signal arrived.

If the code needs an order, say so — with a lock, a signal, or by not sharing. Anything you merely expect, you have already got wrong.

Questions?

Thank you.

A coroutine is not a thread

A coroutine is a function that can stop in the middle and be resumed later. That is the entire idea.

- Nothing runs in parallel. There is one thread, and it is yours.
- Nothing needs a lock, because nothing is shared.
- What it buys is the *shape* of the code: the loop that produces values and the loop that consumes them can both be written as loops.

A coroutine is not a thread

A coroutine is a function that can stop in the middle and be resumed later. That is the entire idea.

- Nothing runs in parallel. There is one thread, and it is yours.
- Nothing needs a lock, because nothing is shared.
- What it buys is the *shape* of the code: the loop that produces values and the loop that consumes them can both be written as loops.

Which is the opposite of the callback or state machine you would otherwise have to write, where one of the two loops gets turned inside out for the other's benefit.

A coroutine is not a thread

A coroutine is a function that can stop in the middle and be resumed later. That is the entire idea.

- Nothing runs in parallel. There is one thread, and it is yours.
- Nothing needs a lock, because nothing is shared.
- What it buys is the *shape* of the code: the loop that produces values and the loop that consumes them can both be written as loops.

Which is the opposite of the callback or state machine you would otherwise have to write, where one of the two loops gets turned inside out for the other's benefit.

Key idea

Concurrency is about *structure*. Parallelism is about *execution*. Coroutines are the first, and only sometimes help with the second.

The three keywords

A function is a coroutine because its body contains one of these. Nothing in the declaration says so:

`co_await`	suspend until something else is ready
`co_yield`	hand a value to the caller and suspend
`co_return`	finish

The three keywords

A function is a coroutine because its body contains one of these. Nothing in the declaration says so:

`co_await`	suspend until something else is ready
`co_yield`	hand a value to the caller and suspend
`co_return`	finish

On the first call the compiler allocates a *coroutine frame* holding the parameters, the locals and where in the body you were.

The three keywords

A function is a coroutine because its body contains one of these. Nothing in the declaration says so:

`co_await`	suspend until something else is ready
`co_yield`	hand a value to the caller and suspend
`co_return`	finish

On the first call the compiler allocates a *coroutine frame* holding the parameters, the locals and where in the body you were.

Pitfall

Reference parameters do not survive

The frame copies the *parameters*, so a reference parameter copies the reference — and what it pointed at is very likely gone by the time the body runs. Exercise 1's bug, in a new costume. Take by value.

What C++20 actually gave you

C++20 shipped the language feature and almost none of the library. The return type has to supply a `promise_type` telling the compiler what suspending, resuming and returning mean:

```
struct Generator {
    struct promise_type {
        int current;
        Generator           get_return_object();
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int v) { current = v; return {}; }
        void                return_void() {}
        void                unhandled_exception() { throw; }
    };
    std::coroutine_handle<promise_type> handle;
    /* ... iterator, destructor, move operations ... */
};
```

What C++20 actually gave you

C++20 shipped the language feature and almost none of the library. The return type has to supply a `promise_type` telling the compiler what suspending, resuming and returning mean:

```
struct Generator {
    struct promise_type {
        int current;
        Generator           get_return_object();
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int v) { current = v; return {}; }
        void                return_void() {}
        void                unhandled_exception() { throw; }
    };
    std::coroutine_handle<promise_type> handle;
    /* ... iterator, destructor, move operations ... */
};
```

A page of boilerplate for the obvious behaviour. Which is why nobody wrote coroutines in C++20.

`std::generator`, which you can use

C++23 provides the one that everybody was writing by hand:

```
1  #include <generator>
2
3  std::generator<std::string> lines(std::string path)
4  {
5          std::ifstream input{path};
6          std::string   line;
7
8          while (std::getline(input, line))
9                  if (not line.empty())
10                         co_yield line;
11 }
12
13 for (auto const& line : lines("app.log"))
14         handle(line);
```

`std::generator`, which you can use

C++23 provides the one that everybody was writing by hand:

```
#include <generator>

std::generator<std::string> lines(std::string path)
{
        std::ifstream input{path};
        std::string   line;

        while (std::getline(input, line))
                if (not line.empty())
                        co_yield line;
}

for (auto const& line : lines("app.log"))
        handle(line);
```

The file is never in memory; one line is. And it is a proper range, so `| std::views::take(5)` reads five lines and stops.

Where this is going, and where it is not

What is genuinely useful today: generators, and the async frameworks that build their own task types on top of the language feature — `cppcoro`, `libunifex`, ASIO's `awaitable`.

Where this is going, and where it is not

What is genuinely useful today: generators, and the async frameworks that build their own task types on top of the language feature — `cppcoro`, `libunifex`, ASIO's `awaitable`.

What is still missing from the standard: a `task<T>`, and any notion of an executor to run one on. Senders and receivers are the C++26 answer, and until then `co_await` in application code means picking a library.

Where this is going, and where it is not

What is genuinely useful today: generators, and the async frameworks that build their own task types on top of the language feature — `cppcoro`, `libunifex`, ASIO's `awaitable`.

What is still missing from the standard: a `task<T>`, and any notion of an executor to run one on. Senders and receivers are the C++26 answer, and until then `co_await` in application code means picking a library.

And the thing to be clear about: none of this makes anything parallel. A thousand coroutines on one thread are still one thread. They are how you *structure* a thousand concurrent operations — what runs them is still a thread, or a pool of them.

Where this is going, and where it is not

What is genuinely useful today: generators, and the async frameworks that build their own task types on top of the language feature — `cppcoro`, `libunifex`, ASIO's `awaitable`.

What is still missing from the standard: a `task<T>`, and any notion of an executor to run one on. Senders and receivers are the C++26 answer, and until then `co_await` in application code means picking a library.

And the thing to be clear about: none of this makes anything parallel. A thousand coroutines on one thread are still one thread. They are how you *structure* a thousand concurrent operations — what runs them is still a thread, or a pool of them.

Best practice

Use `std::generator` where a lazy sequence is the natural shape. For everything else in C++23, the futures and `std::async` from earlier are the ones that exist.