Lecture 10 — Use of Locks, Reentrancy

Jeff Zarnett & Patrick Lam
{jzarnett, patrick.lam}@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

October 16, 2022

Part I

Use of Locks

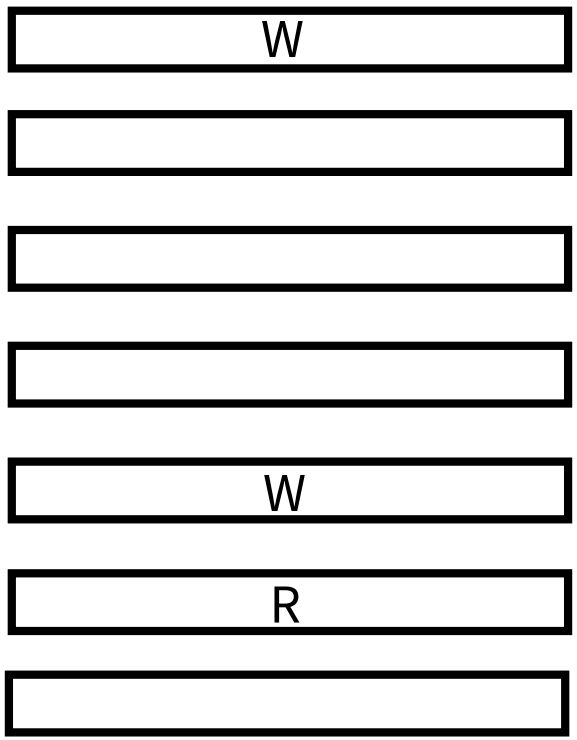
Use of Locks

In previous courses: learned about locking and how it works.

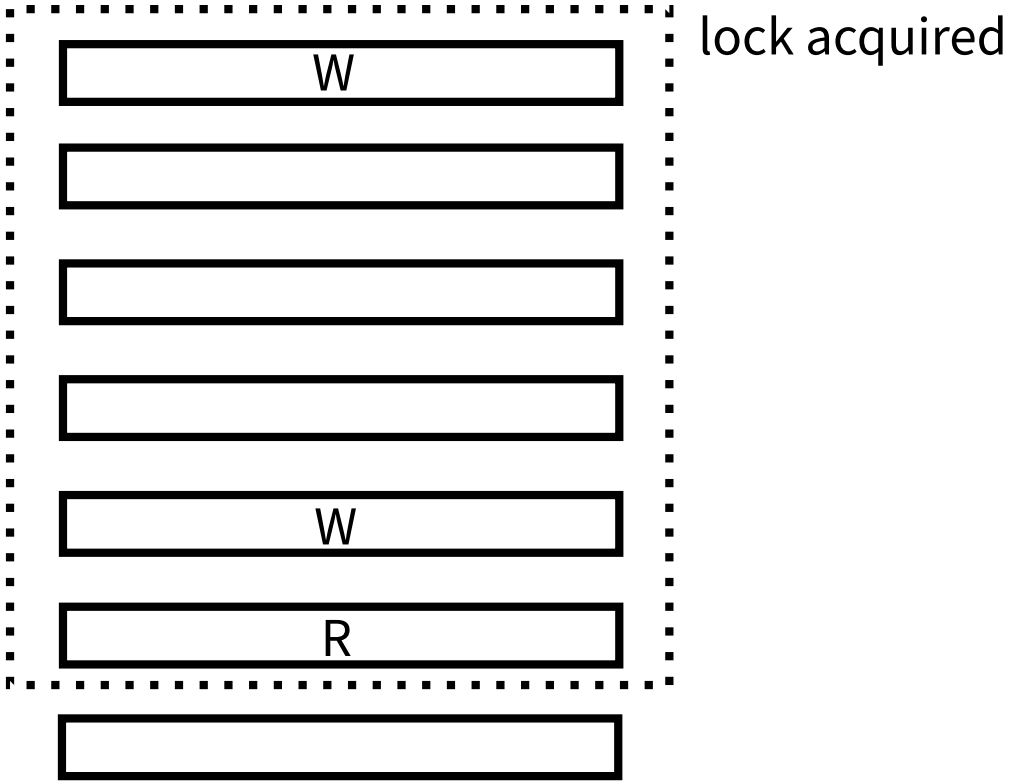
Before: was enough to avoid bad stuff.

Now: Still important, but not enough.

Critical Section Size

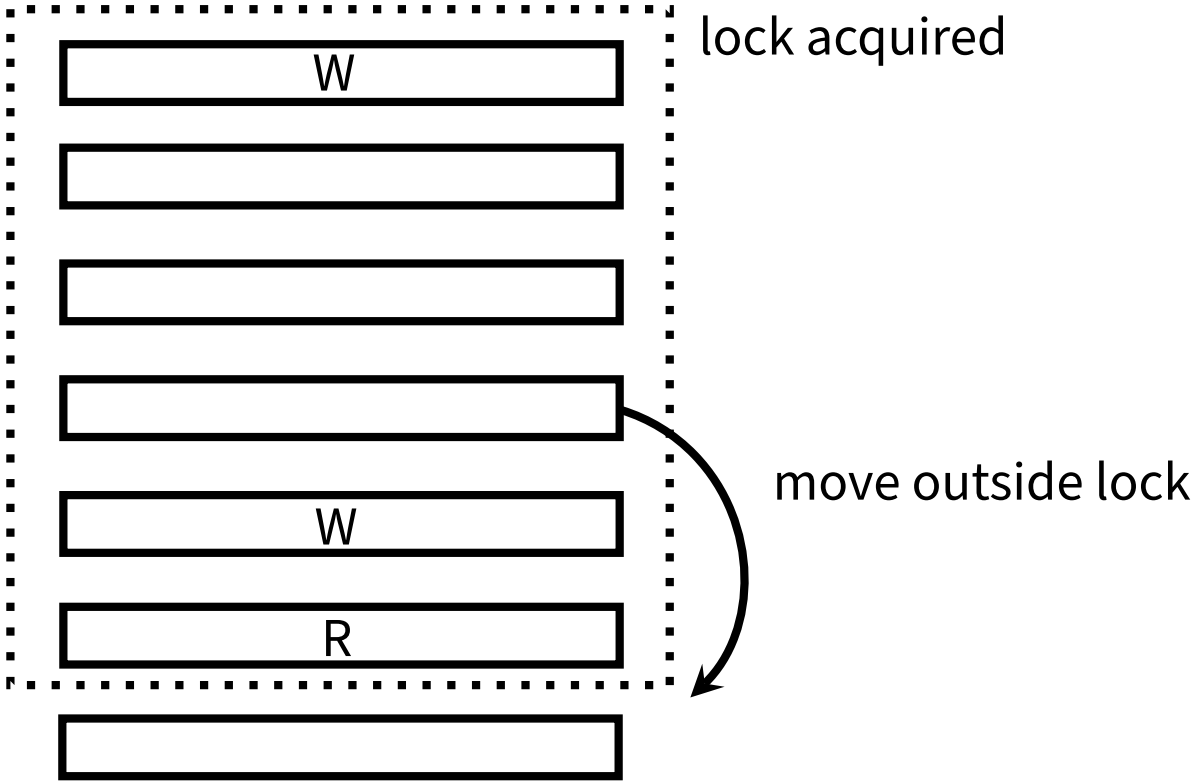


Critical Section Size



Critical sections should be as large as they need to be but no larger.

Rewrite This Block



Sometimes control flow or other very short statements might get swept into the critical section.

Those should be the exception rather than the rule.

Remember the Producer-Consumer Problem?

Short code example from the producer-consumer problem:

```
for _j in 0 .. NUM_THREADS {
  // create consumers
  let spaces = spaces.clone();
  let items = items.clone();
  let buffer = buffer.clone();
  threads.push(
    thread::spawn(move || {
      for _k in 0 .. ITEMS_PER_THREAD {
        let permit = block_on(items.acquire());
        let mut buf = buffer.lock().unwrap();
        let current_consume_space = buf.consumer_count;
        let next_consume_space = (current_consume_space+1) % buf.buffer.len();
        let to_consume = *buf.buffer.get(current_consume_space).unwrap();
        buf.consumer_count = next_consume_space;
        spaces.add_permits(1);
        permit.forget();
        consume_item(to_consume);
      }
    })
  );
}
```

Previously, More Explicit

When we used locks in C (or similar), it was easier to identify what's in the critical section, because we had explicit lock and unlock statements.

The explicit unlock statement, especially, made it much clearer where it ends.

Now, we don't consider the critical section over until the `MutexGuard` (returned by `lock()`) goes out of scope.

And that happens here at the end of the iteration of the loop.

“Do you think you got him?”

What I always say is to analyze this closure one statement at a time and look into which of these access shared variables.

Better to be Safe than Sorry

In a practical sense, the critical section needs to enclose anything that references `buf`.

Looking at that code, how did we do?

Rust is good about not letting you access shared data in an uncontrolled way.

But not so good at making sure nothing extra is there.

Option 1: Manual Scoping

```
for _j in 0 .. NUM_THREADS {
 // create consumers
 let spaces = spaces.clone();
 let items = items.clone();
 let buffer = buffer.clone();
 threads.push(
   thread::spawn(move || {
     for _k in 0 .. ITEMS_PER_THREAD {
       let permit = block_on(items.acquire());
       let to_consume = {
         let mut buf = buffer.lock().unwrap();
         let current_consume_space = buf.consumer_count;
         let next_consume_space = (current_consume_space+1)%buf.buffer.len();
         let to_consume = *buf.buffer.get(current_consume_space).unwrap();
         buf.consumer_count = next_consume_space;
         to_consume
       };
       spaces.add_permits(1);
       permit.forget();
       consume_item(to_consume);
     }
   })
 );
}
```

Option 2: Taking Out the Guard

The other approach to making the `MutexGuard` go out of scope is to actually call `drop()` on it.

This is effective in telling the compiler that it is time for this value to die.

Calling `drop()` moves ownership of the MutexGuard to the `drop` function where it will go out of scope and be removed.

Like for Like

I applied a similar change the producer code as we just discussed about the consumer.

I added some thread sleeps to the original and modified program so it appears that consuming or producing an item actually takes meaningful work.

Benchmarks are created with `hyperfine -warmup 1 -m 5 "cargo run -release"`.

Test Results

Unoptimized version: 2.8 seconds

Optimized version: 1.1 seconds

Locks Have Costs

Keeping the critical section as small as possible is important because it speeds up performance (reduces the serial portion of your program).

But that's not the only reason.

The lock is a resource, and contention for that resource is itself expensive.

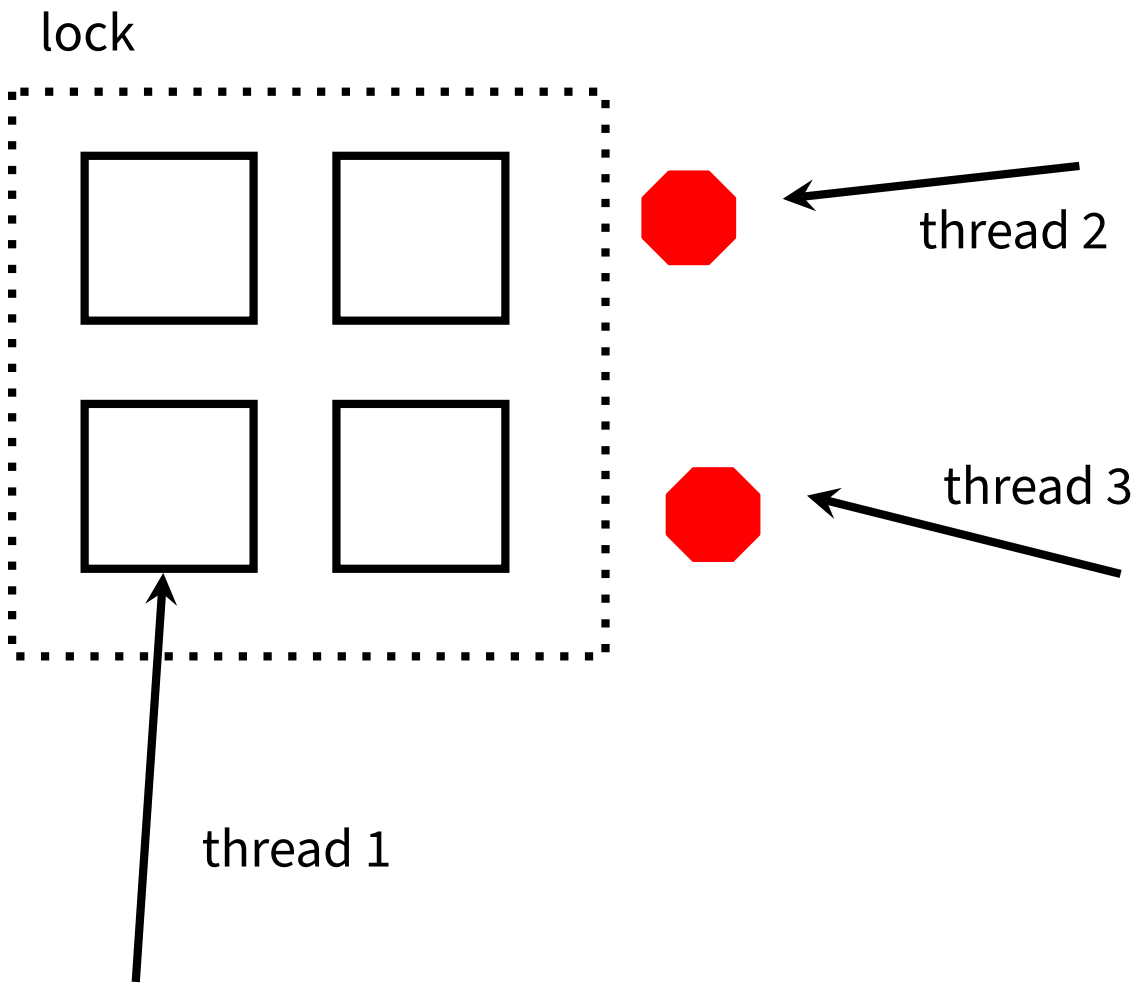
Locking Granularity

Alright, we already know that locks prevent data races.

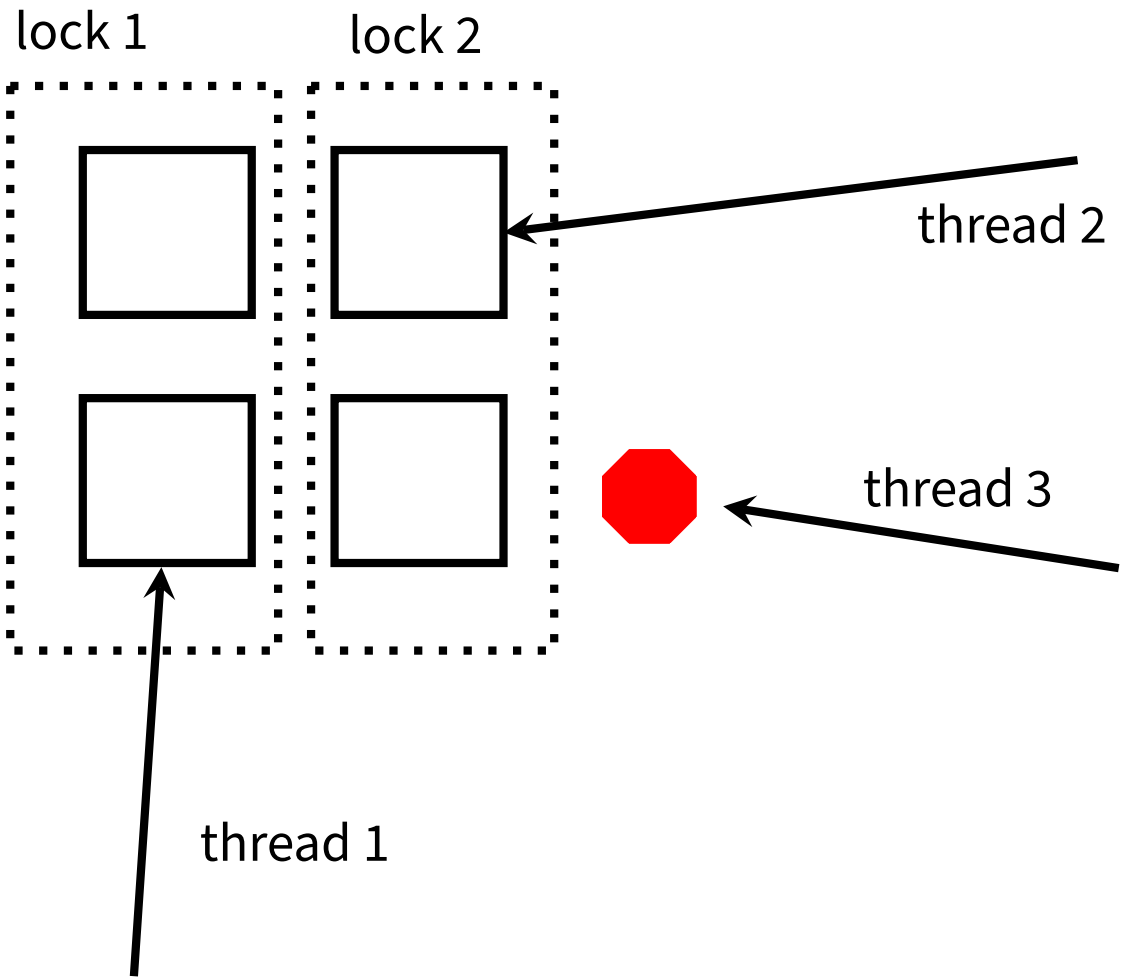
So we need to use them to prevent data races, but it's not as simple as it sounds.

We have choices about the granularity of locking, and it is a trade-off.

Coarse-Grained Lock



Fine-Grained Locks



Coarse-Grained vs Fine-Grained Locking

Coarse-grained locking is easier to write and harder to mess up, but it can significantly reduce opportunities for parallelism.

Fine-grained locking requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc).

Locks' extents constitute their granularity.

Lock Concerns

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e., remaining race conditions).

Lock Overhead

Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention

Most locking time is wasted waiting for the lock to become available.

We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X'. ($X = X'$ is allowed).

Okay, in a formal sense, the four conditions for deadlock are:

1. **Mutual Exclusion**
2. **Hold-and-Wait**
3. **No Preemption**
4. **Circular-Wait**

Simple Deadlock Example

Consider, for instance, two processors trying to get two locks.

Thread 1

```
Get Lock 1
Get Lock 2
Release Lock 2
Release Lock 1
```

Thread 2

```
Get Lock 2
Get Lock 1
Release Lock 1
Release Lock 2
```

Avoiding Deadlocks

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**.

You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

```
let mut thing1 = l1.lock().
    unwrap()
let mut thing2 = l2.lock().
    unwrap()
// protected code
// locks dropped when going out
    of scope
```

```
let mut thing1 = l1.lock().
    unwrap()
let mut thing2 = l2.lock().
    unwrap()
// protected code
// locks dropped when going out
    of scope
```

Trylock

Alternately, you can use trylock.

Recall that Pthreads' `trylock` returns 0 if it gets the lock.

But if it doesn't, your thread doesn't get blocked.

```
    loop {
        let mut m1 = l1.lock().unwrap();
        let m2 = l2.try_lock();
        if m2.is_ok() {
            *m1 += amount;
            *m2.unwrap() -= amount;
            break;
        } else {
            println!("try_lock failed");
            // Go around the loop again and try again
        }
    }
```

This prevents the hold and wait condition.

There Can Be Only One

Photo Credit: Craig Middleton[1]

[1] https://www.flickr.com/photos/craigmiddleton/3579668458

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (1?).

This is *coarse-grained locking*; it does have a couple of advantages:

- it is easier to implement;
- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

Your parallel program will quickly become sequential.

Python puts a lock around the whole interpreter (known as the *global interpreter lock*).

This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if one of the threads is waiting for I/O.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking*.

Fine-Grained Locking

The big advantage: it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Lock Scope Examples

Databases may lock fields / records / tables. (fine-grained $\rightarrow$ coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)

Part II

Reentrancy

Out and In Again

Non-Reentrant C Function

The trivial example of a non-reentrant C function:

```
int tmp;

void swap( int x, int y ) {
    tmp = y;
    y = x;
    x = tmp;
}
```

Why is this non-reentrant?

How can we make it reentrant?

Reentrancy

$\Rightarrow$ A function can be suspended in the middle and
re-entered (called again) before the previous execution returns.

Does not always mean **thread-safe** (although it usually is).

- Recall: **thread-safe** is essentially "no data races".

Moot point if the function only modifies local data, e.g. `sin()`.

Reentrancy Example

Courtesy of Wikipedia (with modifications):

```
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example—Explained (a trace)

```
call swap(&a, &b);
 t = *x;                    // t = 3 (a)
 *x = *y;                   // a = 4 (b)
 call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    t = *x;                 // t = 1 (x)
    *x = *y;                // x = 2 (y)
    *y = t;                 // y = 1
 *y = t;                    // b = 1

Final values:
a = 4, b = 1

Expected values:
a = 4, b = 3
```

Reentrancy Example, Fixed

```
int t;

void swap(int *x, int *y) {
    int s;

    @\alert{s = t}@;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    @\alert{t = s}@;  // restore global variable
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example, Fixed—Explained (a trace)

```
call swap(&a, &b);
s = t;                  // s = UNDEFINED
t = *x;                 // t = 3 (a)
*x = *y;                // a = 4 (b)
call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    s = t;              // s = 3
    t = *x;             // t = 1 (x)
    *x = *y;            // x = 2 (y)
    *y = t;             // y = 1
    t = s;              // t = 3
*y = t;                 // b = 3
t = s;                  // t = UNDEFINED

Final values:
a = 4, b = 3

Expected values:
a = 4, b = 3
```

Remember that in things like interrupt subroutines (ISRs) having the code be reentrant is very important.

Interrupts can get interrupted by higher priority interrupts and when that happens the ISR may simply be restarted, or we pause and resume.

Either way, if the code is not reentrant we will run into problems.

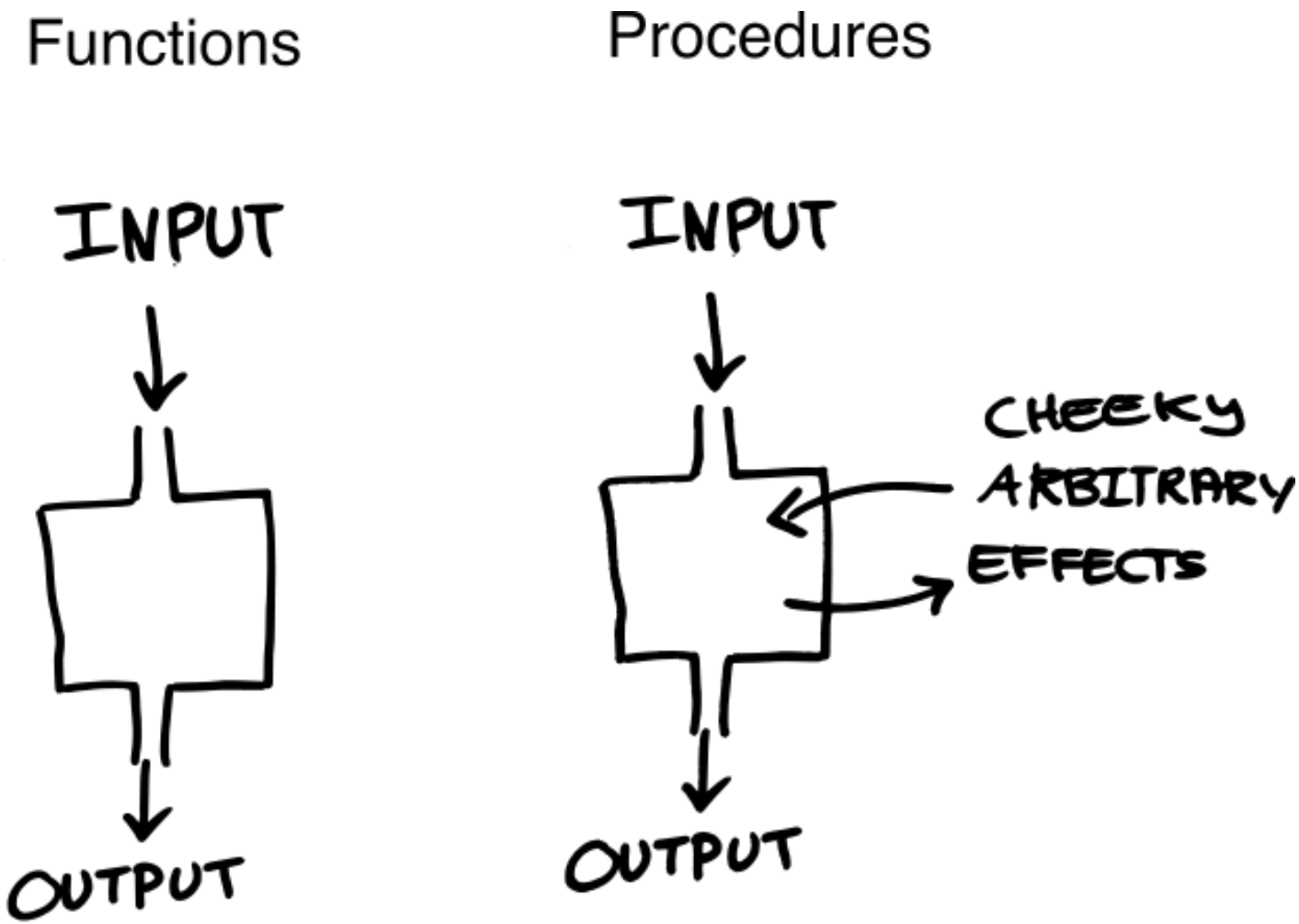
Functional Programming

Interestingly, functional programming languages (NOT procedural like C) such as Scala and so on, lend themselves very nicely to being parallelized.

Why?

Because a purely functional program has no side effects and they are very easy to parallelize.

Any impure function has to indicate that in its function signature.



https://www.fpcomplete.com/blog/2017/04/pure-functional-programming

Joel on Functional

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable.

— Joel Spolsky

OOP: `getBrain().damage()`

Object oriented programming kind of gives us some bad habits in this regard.

We tend to make a lot of `void` methods.

In functional programming these don't really make sense, because if it's purely functional, then there are some inputs and some outputs.

If a function returns nothing, what does it do?

For the most part it can only have side effects which we would generally prefer to avoid if we can, if the goal is to parallelize things.

Functional Programming in Your C++

`algorithms` has been part of C++ since C++11.
C++17 (not well supported yet) introduces parallel and vectorized `algorithms`;
you specify an execution policy, compiler does it:
(`sequenced`, `parallel`, or `parallel_unsequenced`).

Some examples of algorithms:
`sort`, `reverse`, `is_heap`...

Other examples of algorithms:
`for_each_n`, `exclusive_scan`, `reduce`

If you know functional programming (e.g. Haskell), these are:
`map`, `scanl`, `fold1/foldl1`.

Side Effects

Side effects are sort of undesirable (and definitely not functional), but not necessarily bad.

Printing to console is unavoidably making use of a side effect, but it's what we want.

We don't want to call print reentrantly, because two threads trying to write at the same time to the console would result in jumbled output.

Or alternatively, restarting the print routine might result in some doubled characters on the screen.

Purity

The notion of purity is related to side effects.

A function is *pure* if it has no side effects and if its outputs depend solely on its inputs.

Pure functions should be implemented as thread-safe and reentrant.

Previous Example: thread-safety

Is the previous reentrant code also thread-safe?
(This is more what we're concerned about in this course.)

Let's see:

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}
```

Consider two calls: `swap(a, b)`, `swap(c, d)` with
`a = 1, b = 2, c = 3, d = 4`.

Previous Example: thread-safety trace

```
global: t

/* thread 1 */                          /* thread 2 */
a = 1, b = 2;
s = t;      // s = UNDEFINED
t = a;      // t = 1
                                        c = 3, d = 4;
                                        s = t;      // s = 1
                                        t = c;      // t = 3
                                        c = d;      // c = 4
                                        d = t;      // d = 3
a = b;      // a = 2
b = t;      // b = 3
t = s;      // t = UNDEFINED
                                        t = s;      // t = 1

Final values:
a = 2, b = 3, c = 4, d = 3, t = 1

Expected values:
a = 2, b = 1, c = 4, d = 3, t = UNDEFINED
```

Another Definition of Thread-Safe Functions

"A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another, in an undefined order, even if the actual execution is interleaved."

Functional Rust?

Rust does not force a functional style upon your program.

It discourages mutability of data, which is sort of functional-ish.

Internal mutability is a thing, but discouraged.