CS202 (003): Operating Systems Concurrency V

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Quiz Time!

Last Time

Performance issues and tradeoffs

Implementation of spinlocks/ mutexes can be **expensive**

Coarse locks **limit** available parallelism

Fine-grained locking leads to **complexity** and hence **bugs**

Mutex costs:

- instructions to execute "mutex acquire"
- sleep/wake up brings resource cost

Only 1 CPU can execute anywhere in the part of your code protected by a lock

See "filemap.c" in handout

Spinlock costs:

- cross-talk among CPUs
- cache line bounces
- fairness issues

But, you should still start with coarse locks!

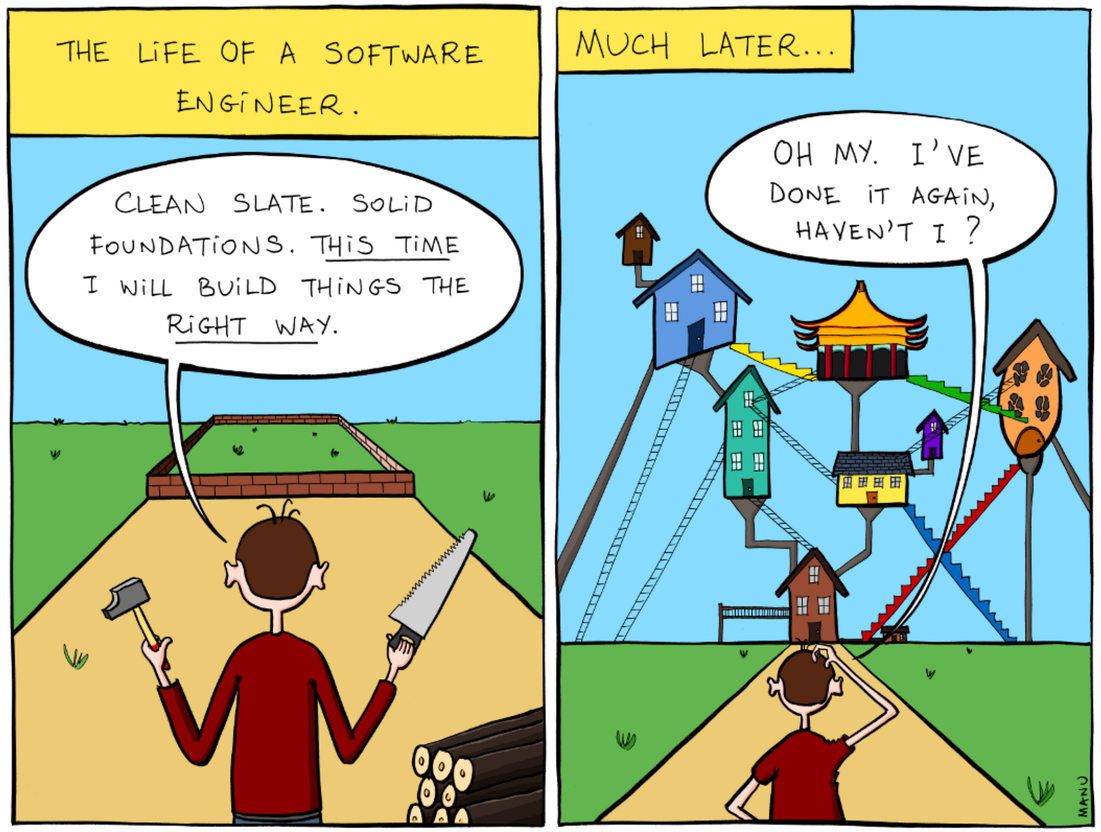
*Look up "MCS locks" if curious

Programmability issues

Loss of modularity

To avoid deadlock, you need to understand how program call each other

You also need to know, whether library functions is thread-safe when you call it. If not, add mutex!



What's the fundamental problem?

Shared memory programming model is hard to use correctly

https://bonkersworld.net/building-software

Some moments of reality about interleaving

Remember sequential consistency?

Modern multi-CPU hardware does not guarantee sequential consistency

https://gunshowcomic.com/648

```
struct foo {
    int abc;
    int def;
};
static int ready = 0;
static mutex_t mutex;
static struct foo* ptr = 0;

void
doublecheck_alloc()
{
    if (!ready) { /* <-- accesses shared variable w/out holding mutex */

        mutex_acquire(&mutex);
        if (!ready) {
            ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
            ready = 1;
        }

        mutex_release(&mutex);

    }
    return;
}
```

Where is the bug?

Yet, if you use mutex correctly…

You don’t have to worry about **arbitrary interleaving**

Critical sections execute atomically

You don’t have to worry about **what hardware is truly doing**

Threading library and compiler do the hard work for you

That does not apply if you do low-level programming

MUST ensure the compiler is not reordering key instructions

MUST know the memory model (of the hardware)

MAY know when to insert memory barriers

```
move $1, 0x10000    # write 1 to memory address 10000
move $2, 0x20000    # write 2 to memory address 20000
MFENCE
move $3, 0x10000    # write 3 to memory address 10000
move $4, 0x30000    # write 4 to memory address 30000
```

If any memory write after **MFENCE** (in program order) is visible to another CPU, then that other CPU also sees all memory writes before the **MFENCE**

"acquire" and "release" in mutexes need memory barriers

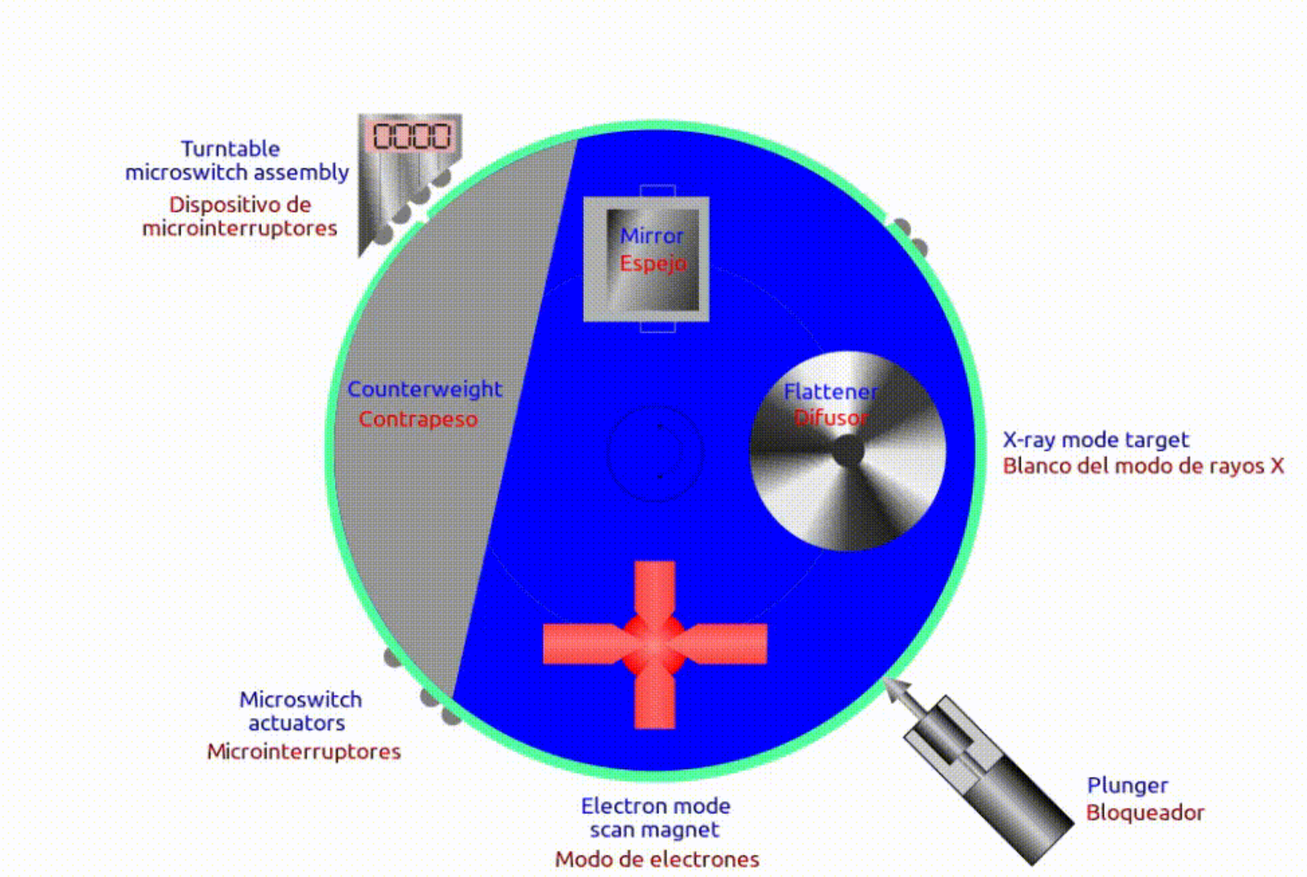
"xchg" on x86 includes an implicit memory barrier

```
struct foo {
    int abc;
    int def;
};
static int ready = 0;
static mutex_t mutex;
static struct foo* ptr = 0;

void
doublecheck_alloc()
{
    if (!ready) { /* <-- accesses shared variable w/out holding mutex */

        mutex_acquire(&mutex);
        if (!ready) {
            ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
            ready = 1;
        }

        mutex_release(&mutex);

    }
    return;
}
```

Where is the bug?

Therac-25

Intended Setting	Beam Energy	Beam Current	Beam Modifier
Electron therapy	5-25 MeV	low	Magnets
X-ray (photon) therapy	25 MeV	high (100x)	Flattener
Field illumination	0	0	None



Therac-25

Intended Setting	Beam Energy	Beam Current	Beam Modifier (determined by the TT)
Electron therapy	5-25 MeV	low	Magnets
X-ray (photon) therapy	25 MeV	high (100x)	Flattener
Field illumination	0	0	None

What can go wrong?

high (100x)	X	Magnets
5-25 MeV	X	Field illumination
25 MeV	X	Field illumination

What actually go wrong?

2 software problems and a bunch of **non-technical problems**

Software problem I

Three threads

Treat

sets a bunch of other parameters
(magnets, energy, current)
read the top byte

Hand

sets the turntable position
read the bottom byte

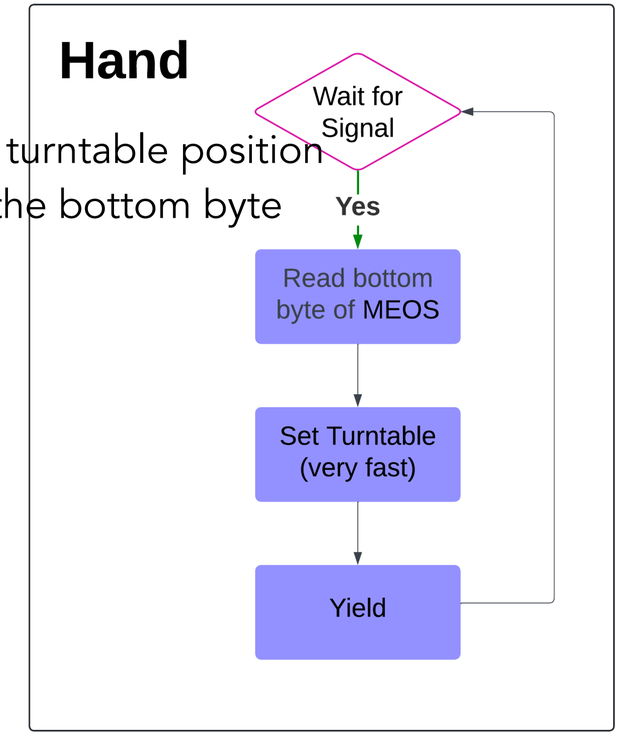
Keyboard

invoked when user types, writes the
input to a two-byte shared variable

Software problem I

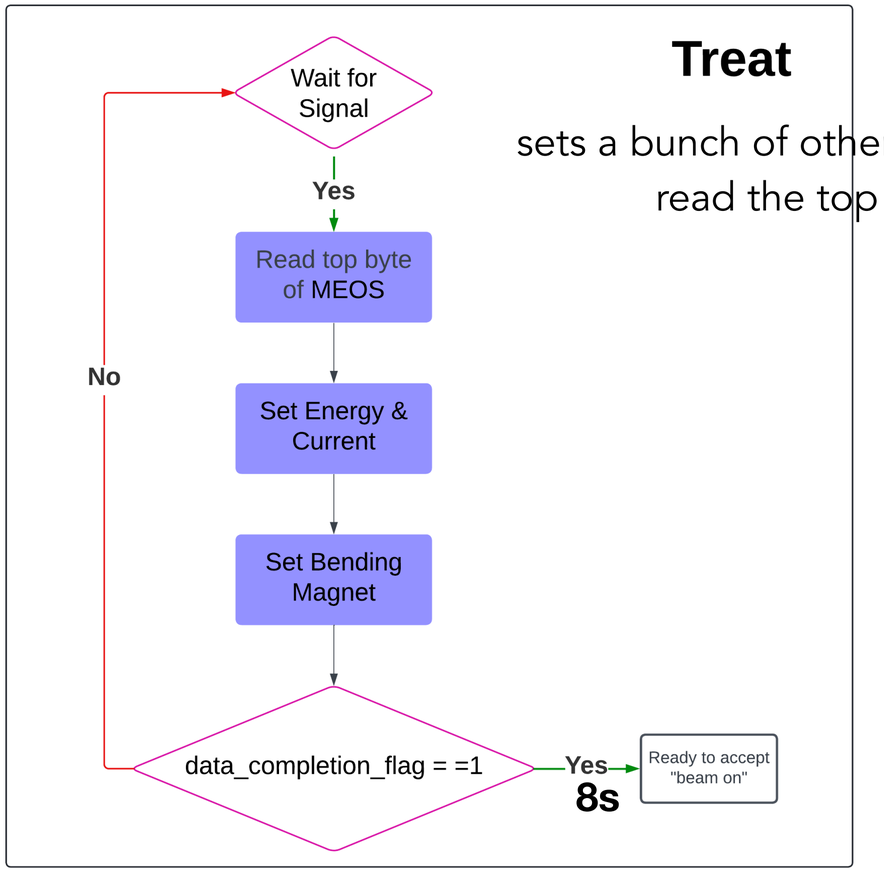
Hand

sets the turntable position
read the bottom byte



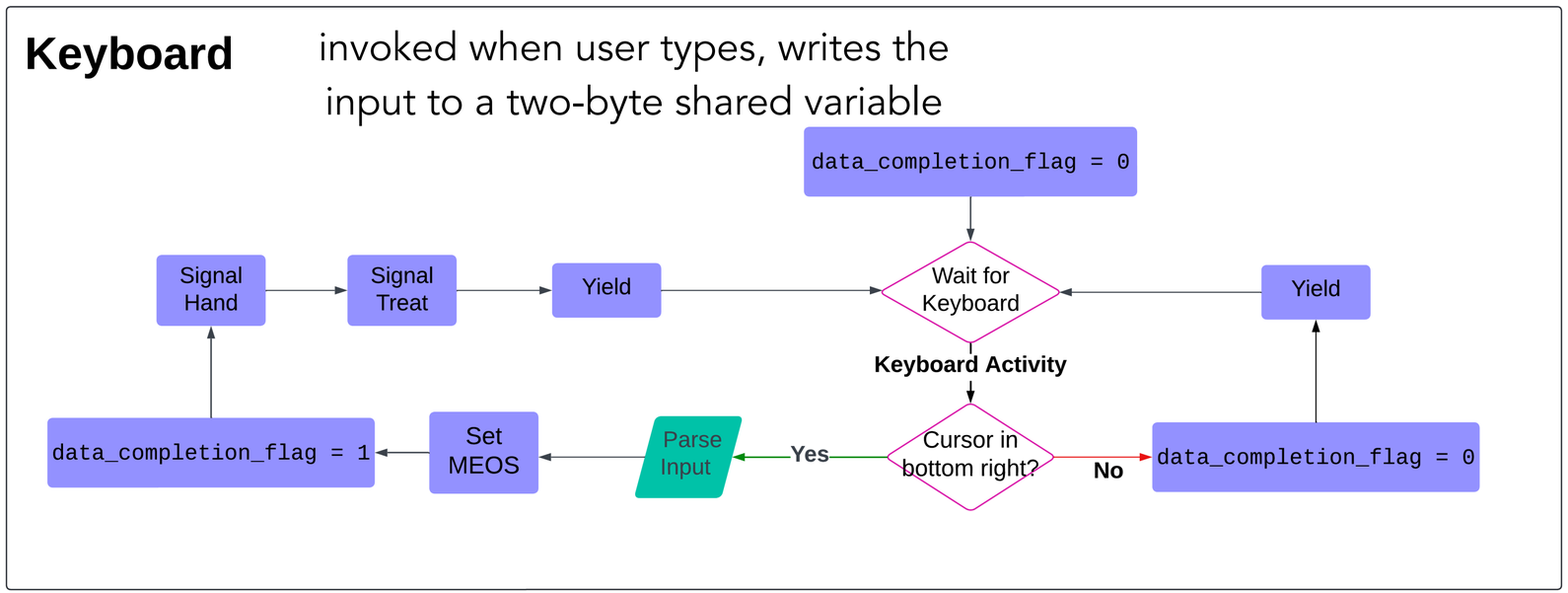
Treat

sets a bunch of other parameters
read the top byte

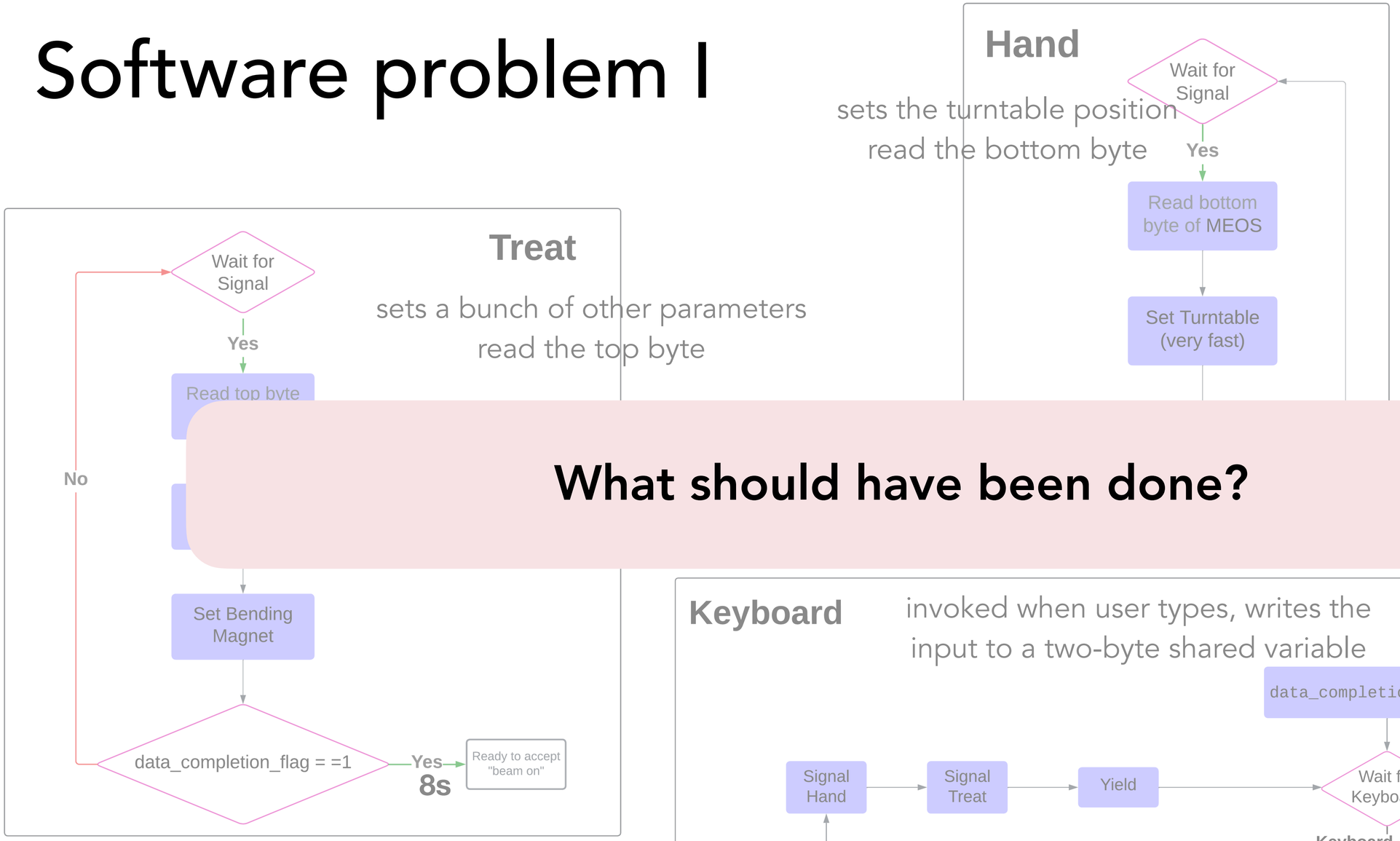


Keyboard

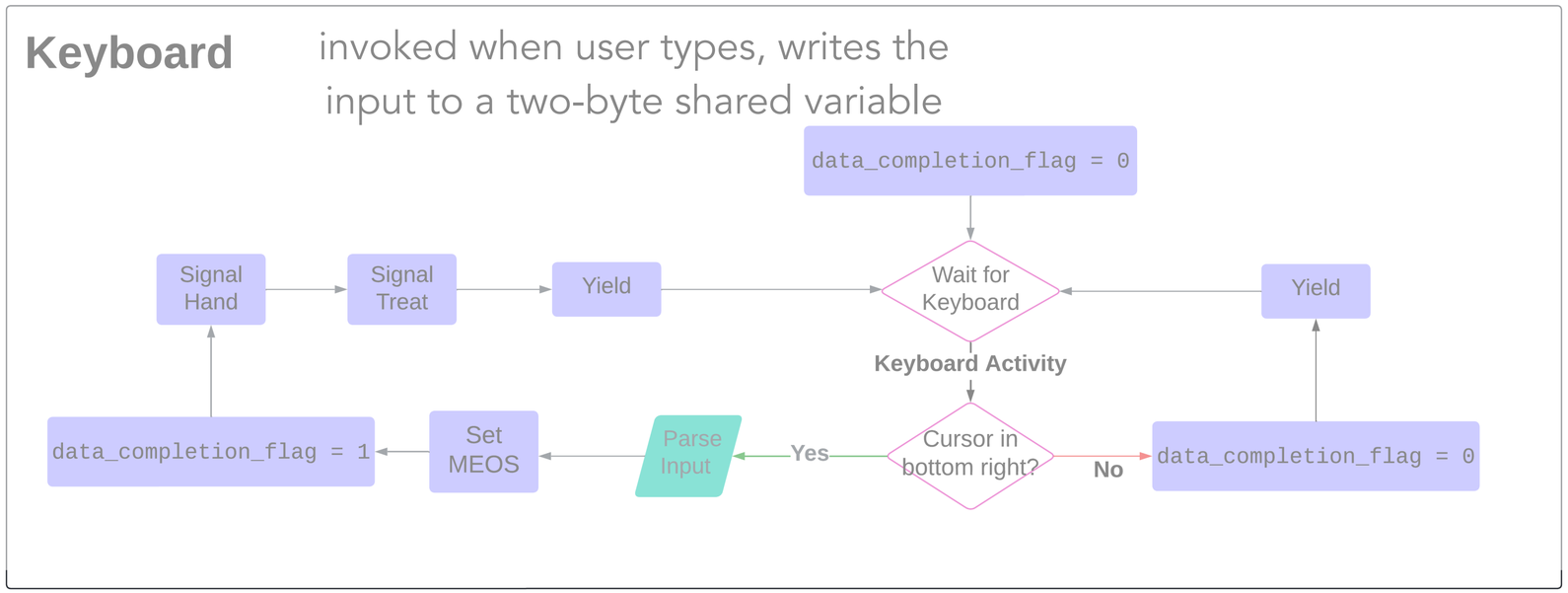
invoked when user types, writes the input to a two-byte shared variable



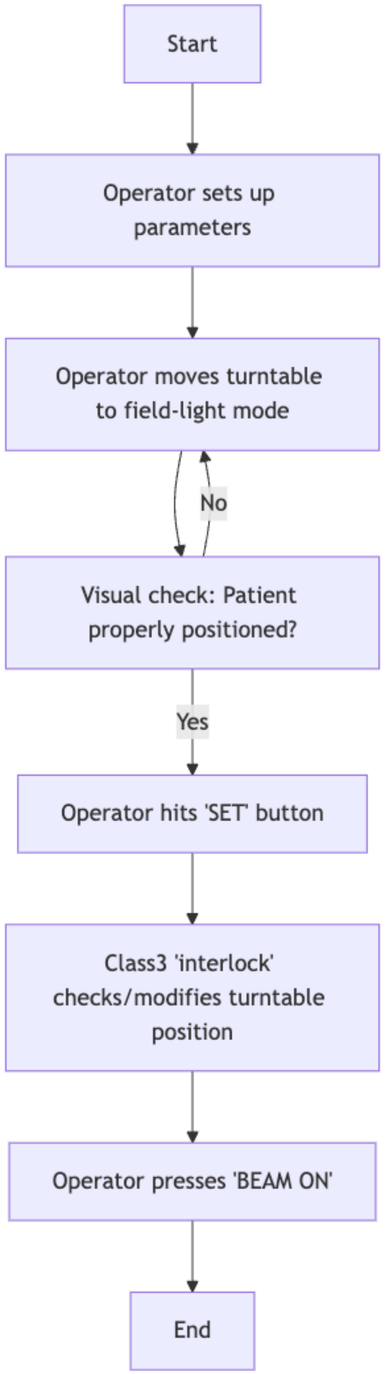
Software problem I

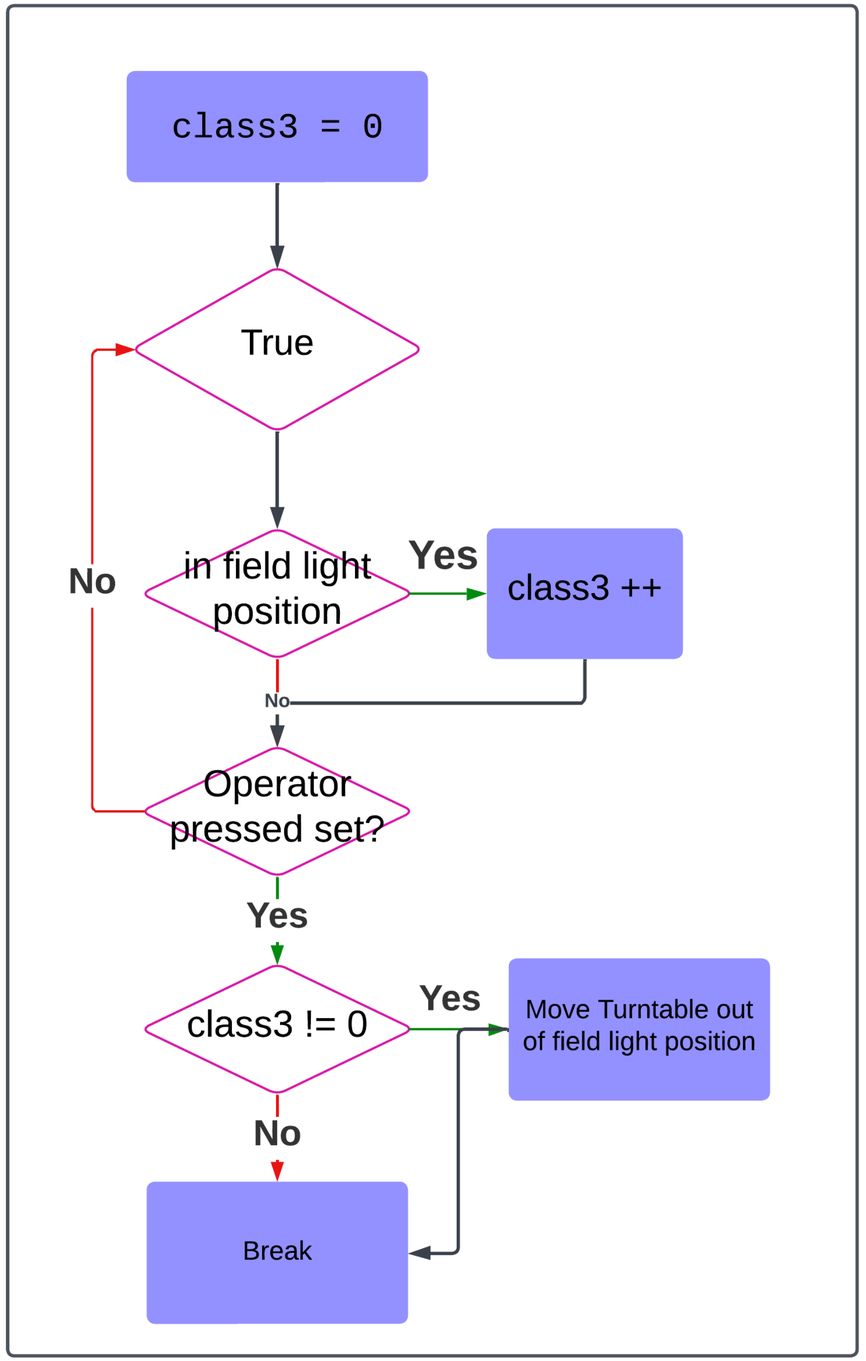


What should have been done?



Software problem II





What else are wrong?

Software Engineering Issues

System Design Failures

Human Errors

What else are wrong?

Software Engineering Issues

No real quality control (lack of unit testing …)

Complex and poor code

Use old code without much thinking

No documentation of software design

System Design Failures

No end-to-end consistency checks

No backup plan to tolerate error (like using hardware interlocks)

Not readable error messages

No error documentation

Human Errors

Assume software is always correct

“Think” errors are fixed without enough formal reasoning

Company did not inform the failures, user weren’t required to report failures

Operators think re-do things will fix the problem

Lack of investigation when failures occur

What should have been done?

Adding a consistency check!

Assume software will make mistakes

Always have back-up failure plans

……

Why are we discussing this?

“There is always another software bug.”

Theme in building systems: be tolerant of inputs / be strict about outputs!

https://medium.com/design-bootcamp/embracing-the-0-bug-policy-a-paradigm-shift-for-bug-free-software-76d18ab53759

Lab 3 is Released Today!

Lab 2 is Due Tomorrow!