

CS202 (003): Operating Systems

Concurrency V

Instructor: Jocelyn Chen

Quiz Time!

Last Time

Deadlock

T1:	T2:
acquire(mutexA);	acquire(mutexB);
acquire(mutexB);	acquire(mutexA);
// do some stuff	// do some stuff
release(mutexB);	release(mutexA);
release(mutexA);	release(mutexB);

Example 1

Deadlock

```
class M {  
    private:  
        Mutex mutex_m;  
        // instance of monitor N  
        N another_monitor;  
  
        // Assumption: no other objects  
        // in the system hold a pointer  
        // to our "another_monitor"  
  
    public:  
        M();  
        ~M();  
        void methodA();  
        void methodB();  
};
```

Example 2: Code see handout

```
class N {  
    private:  
        Mutex mutex_n;  
        Cond cond_n;  
        int navailable;  
  
    public:  
        N();  
        ~N();  
        void* alloc(int nwanted);  
        void free(void*);  
}
```

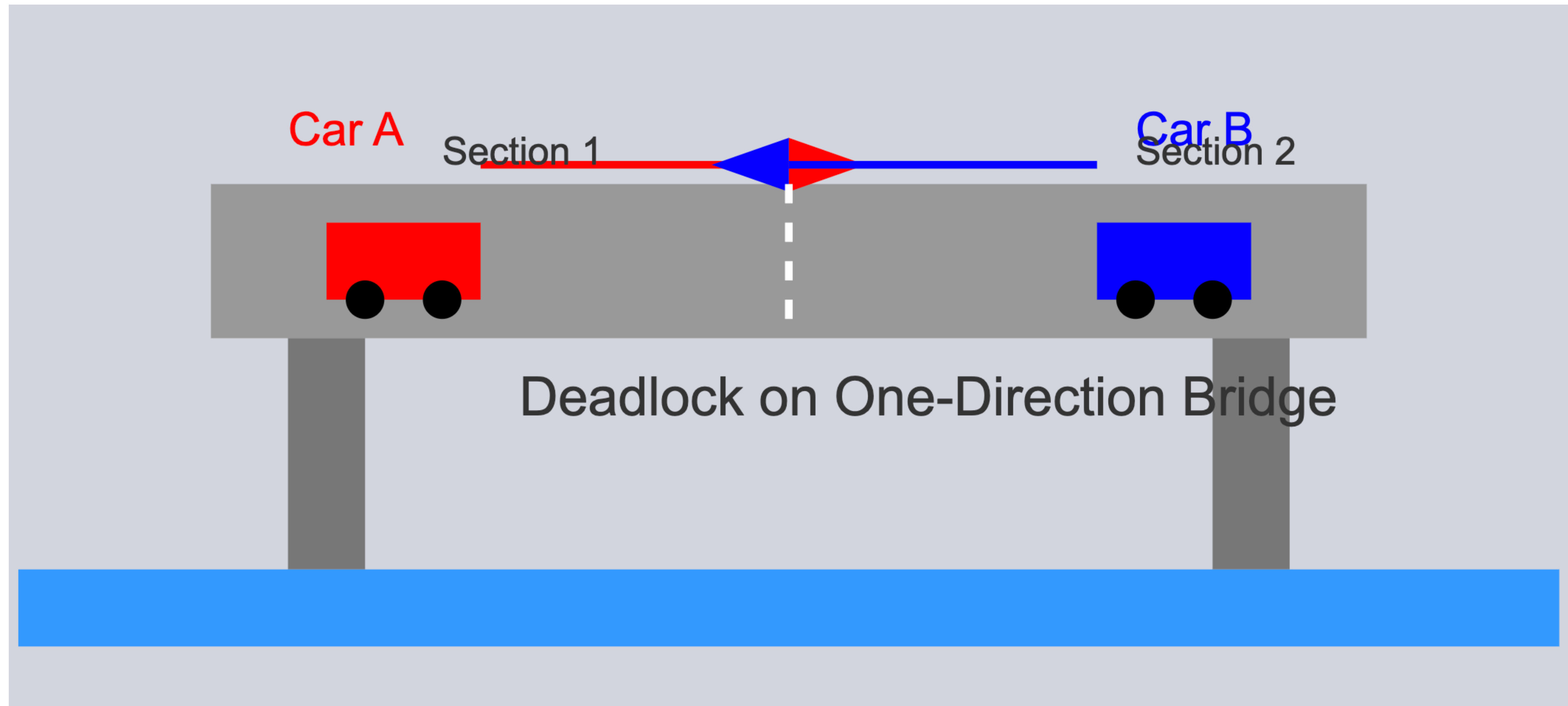
M:
acquire(&mutex_m);
n.alloc(nwanted)

N:
acquire(&mutex_n)
navailable < nwanted
release(&mutex_n)



acquire(&mutex_m);

Deadlock

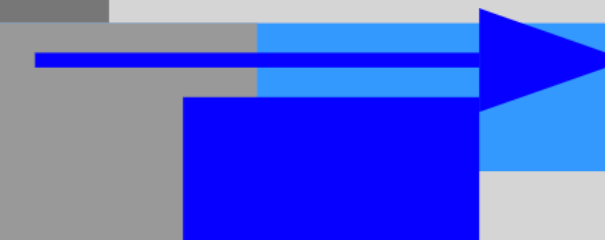


Deadlock

Step 1:
Deadlock



Step 2:
Blue car
backs up



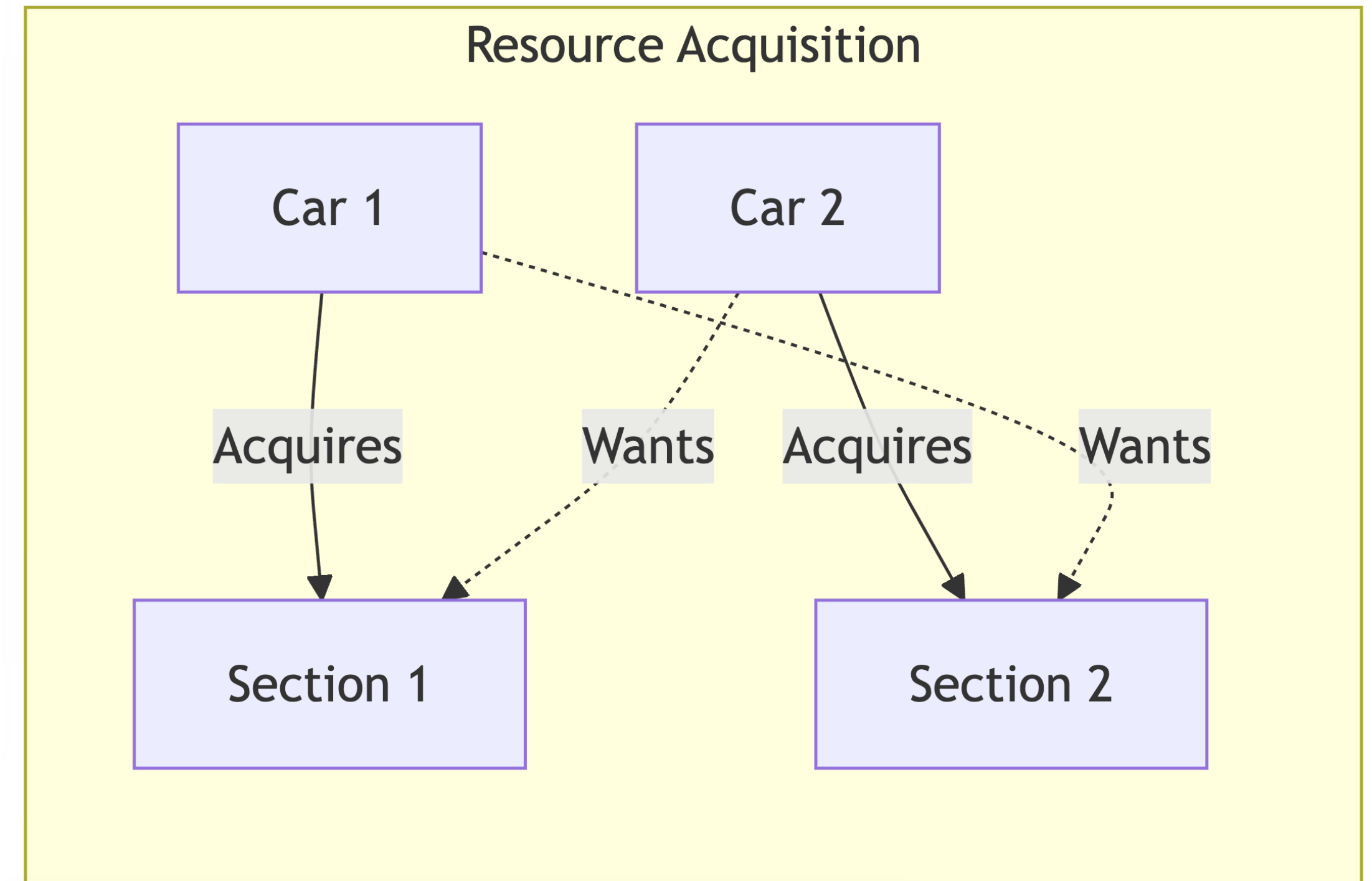
Step 3:
Red car
proceeds



Deadlock

Happens when **all four** conditions are present:

- (1) Mutual exclusion
- (2) Hold and wait
- (3) No pre-emption
- (4) Circular wait



Preventing deadlock

Ignore It!

“admit defeat”

Detect & Recover

Works in development, not really viable for production

Avoid Algorithmically

There are ways but we don't cover them in this class¹

Negate Any of the Conditions

Mutual exclusion
put a queue for
accessing resources

Hold and wait
not likely to work

No preemption
not likely to work

Circular dependency
put partial order on locks
(=> no cycles)

Static/Dynamic Analysis

Static: detect potential errors without running the code²

Dynamic: detect (potential) error during/after execution³

Check the following if you are curious:

¹Section 6.5.3 of Modern Operating Systems (Tanenbaum)

²Engler, D. and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks.

³Savage, S., M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs.

Other progress issues

Starvation

Thread waiting indefinitely
(if low priority and/or resource is contended)

Priority Inversion

T1: (highest priority)	T2: (middle priority)	T3: (lowest priority)
		hold the lock
start		preempt T3
waiting for lock		
	start	
	running	

Why does T2 control the CPU?

Priority inversion - potential fixes

Solution 1

T1: (highest priority)	T2: (middle priority)	T3: (highest priority)
		hold the lock
start		
waiting for lock		
		finish T3
		release the lock
acquire the lock		
running		

.....

Solution 2

T1: (highest priority)	T2: (middle priority)	T3: (lowest priority)
		hold the lock
start		
waiting for lock		
		disable interrupt
		finish T3
		release the lock
acquire the lock		
running		

.....

Solution 3

Don't handle it.

Design the code wisely so that only adjacent priority processes/threads share the lock

Performance issues and tradeoffs

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

Mutex costs:

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

Spinlock costs:

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

Coarse locks **limit**
available parallelism

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

**But, you should still
start with coarse locks!**

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

**See “filemap.c” in
handout**

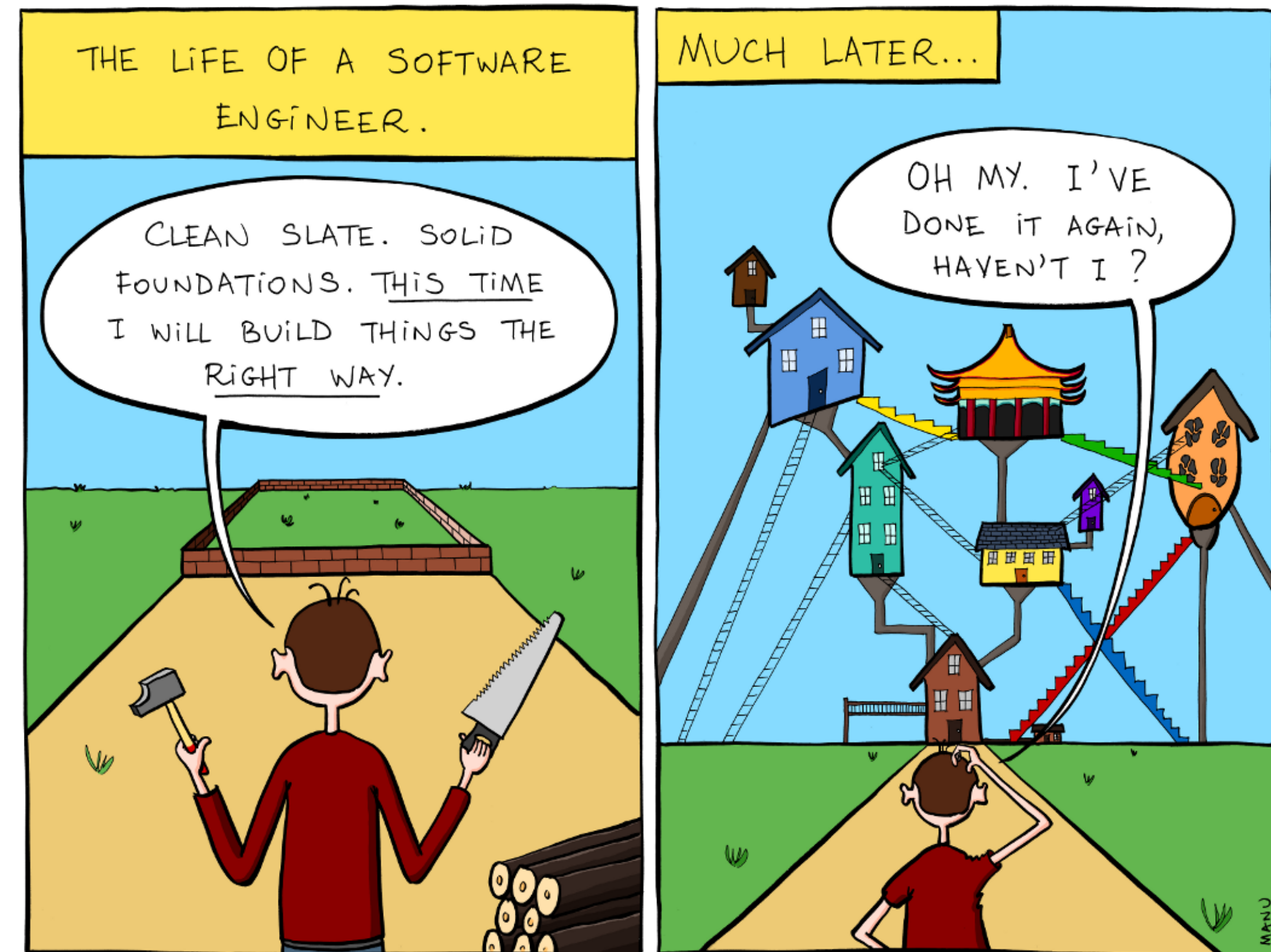
Programmability issues

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?

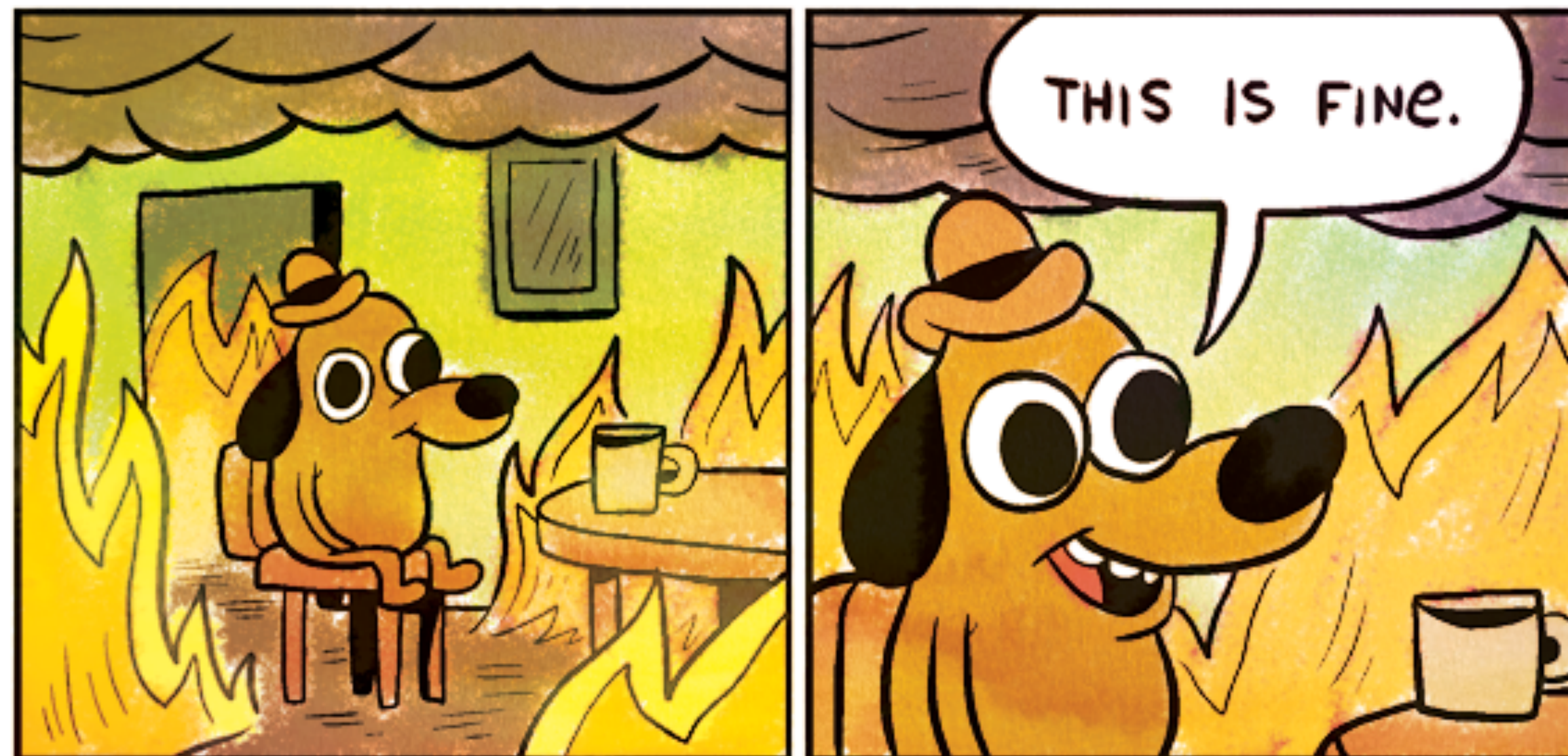


Shared memory programming model is hard to use correctly

Some moments of reality about interleaving

Remember sequential consistency?

Modern multi-CPU hardware does not guarantee sequential consistency



```
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 */

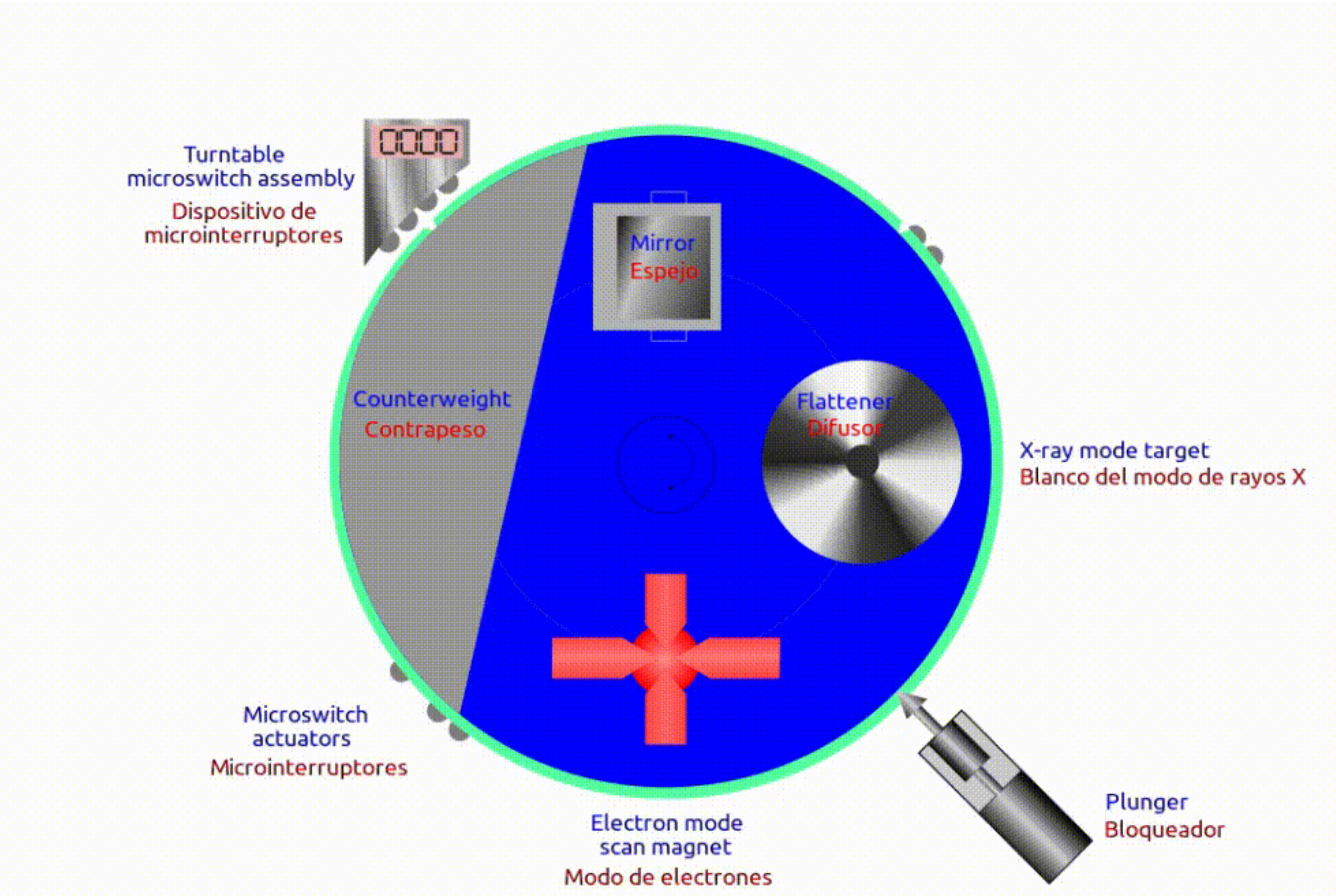
        mutex_acquire(&mutex);
        if (!ready) {
            ptr = alloc_foo(); /* <-- sets ptr to be non-zero */
            ready = 1;
        }

        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

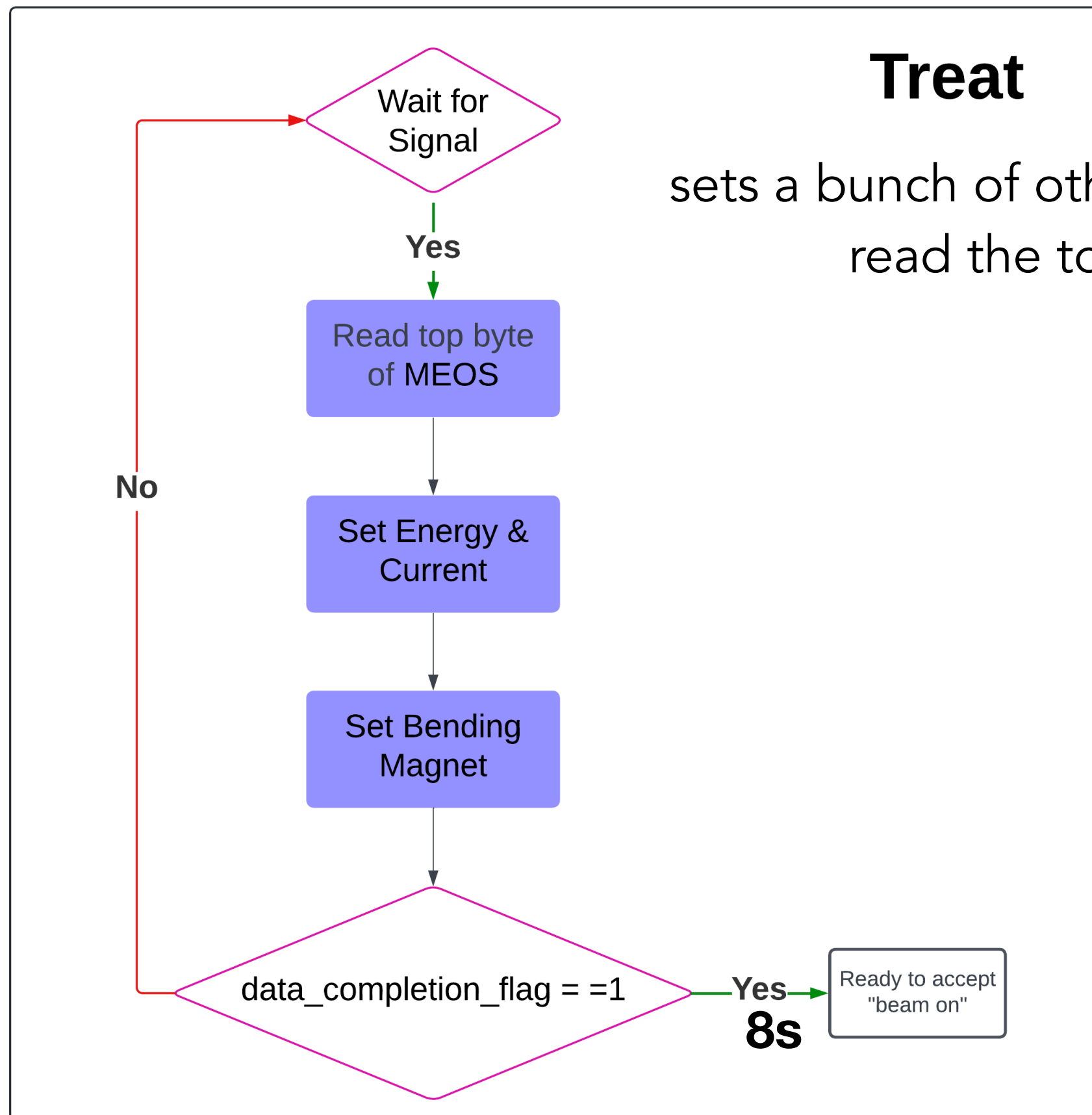
Hand

sets the turntable position
read the bottom byte

Keyboard

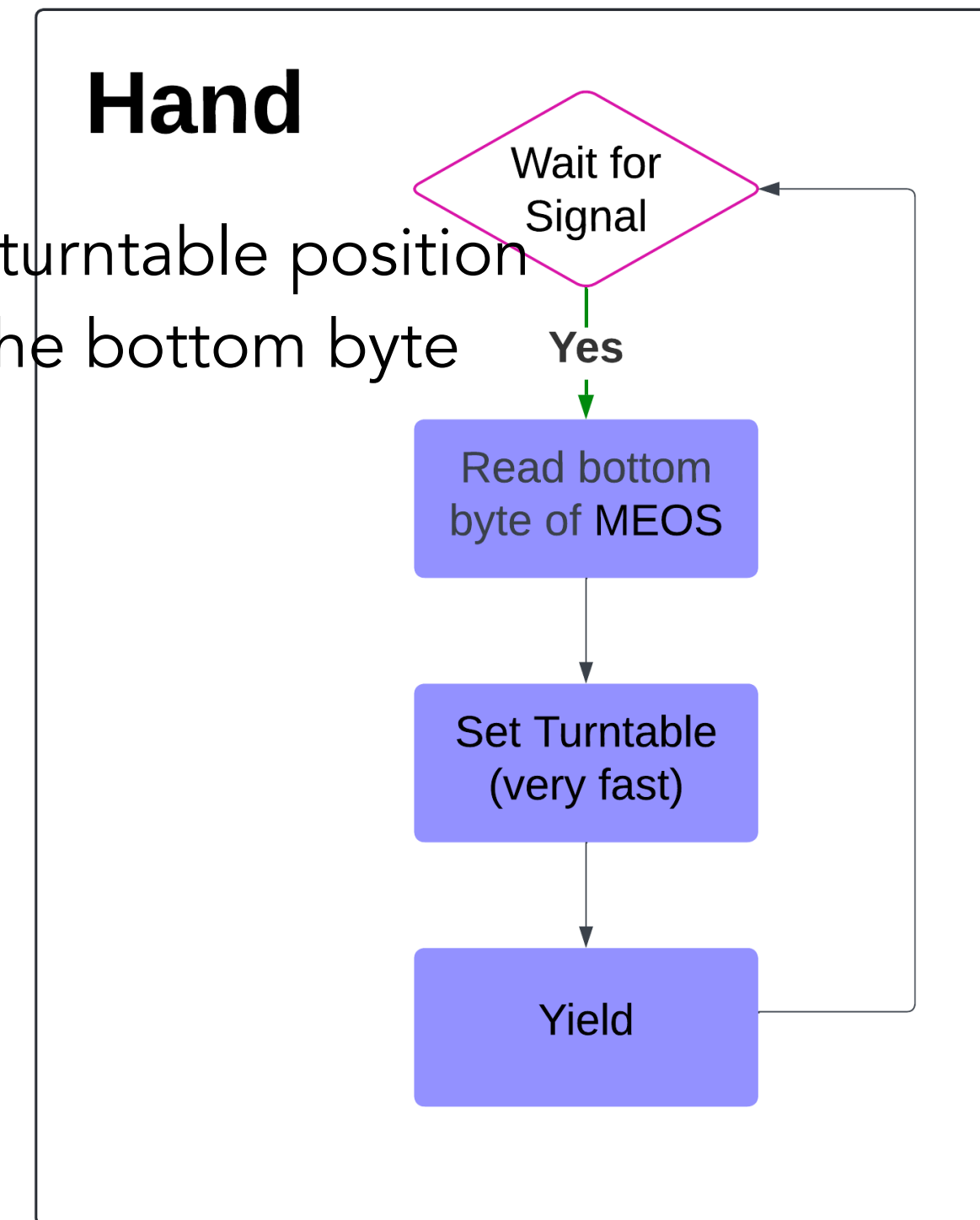
invoked when user types, writes the
input to a two-byte shared variable

Software problem I



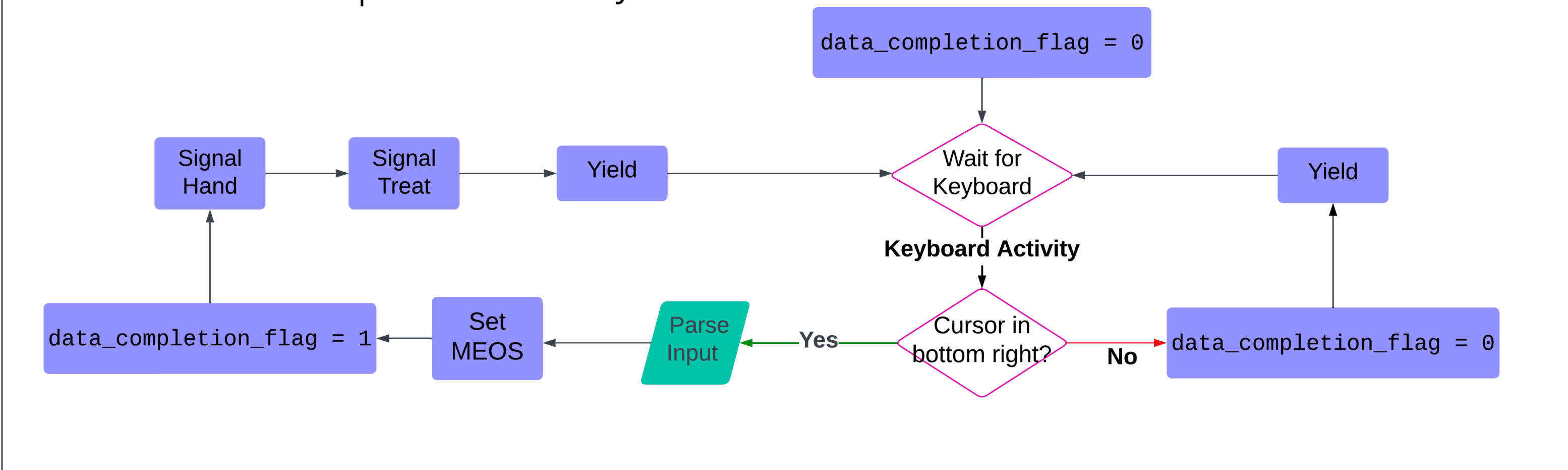
Hand

sets the turntable position
read the bottom byte

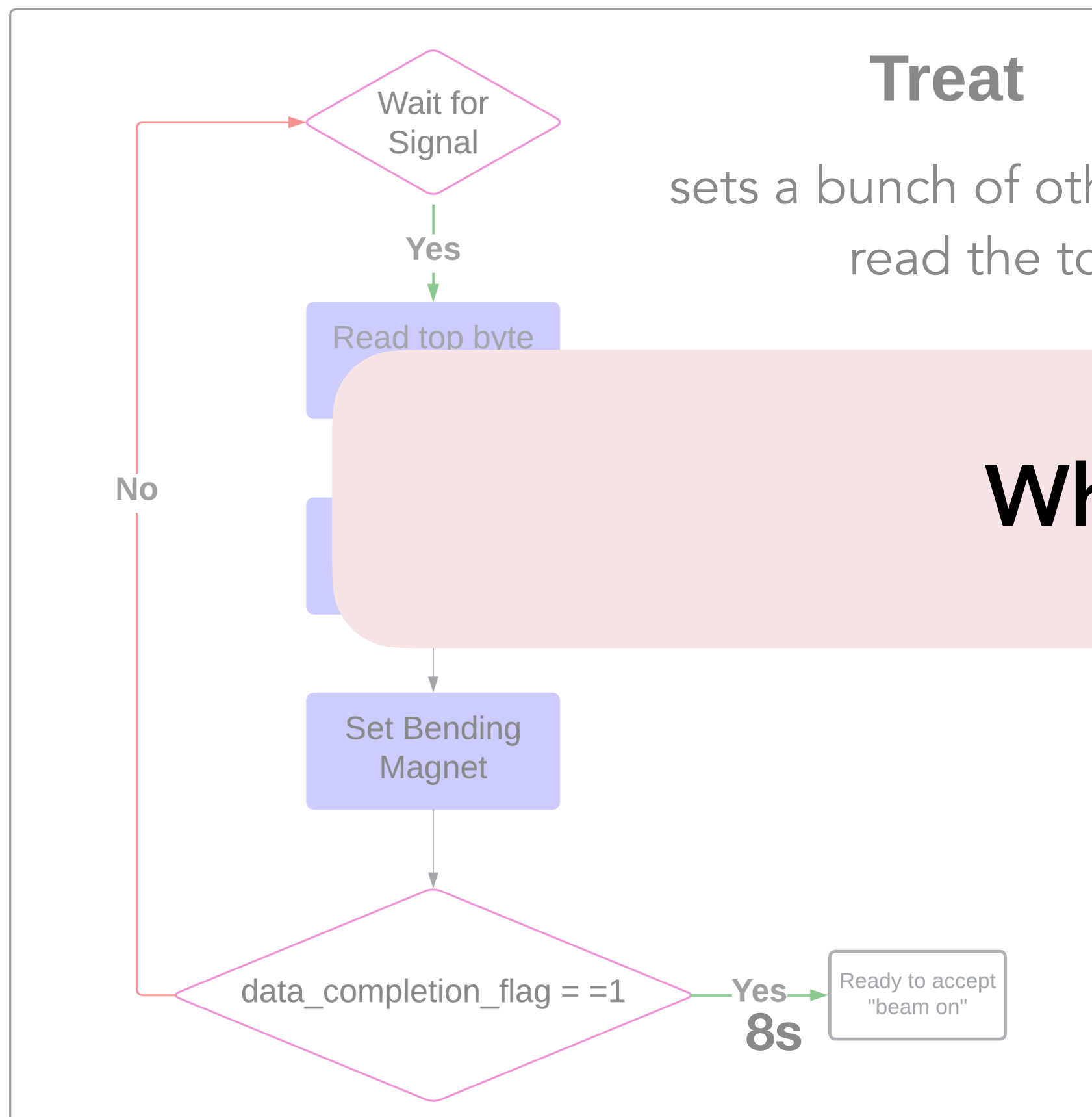


Keyboard

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

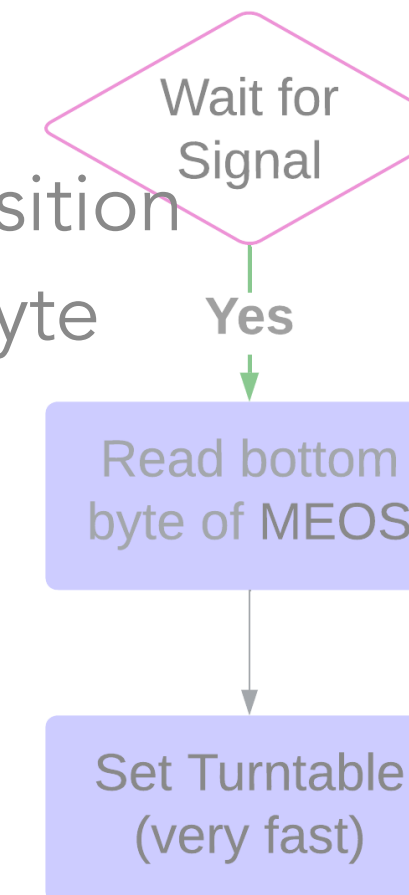


Software problem I



Hand

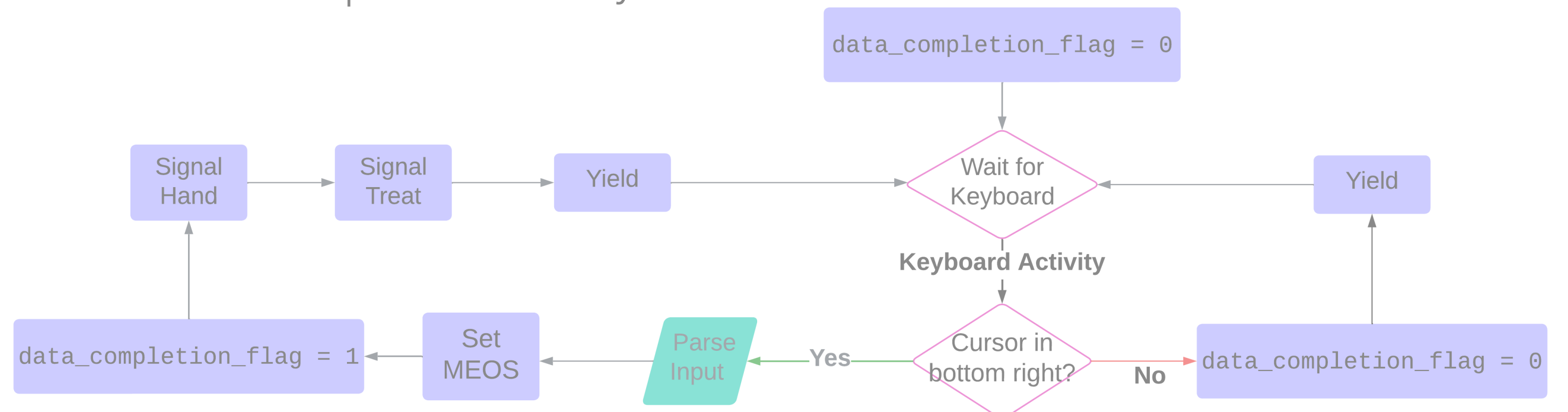
sets the turntable position
read the bottom byte



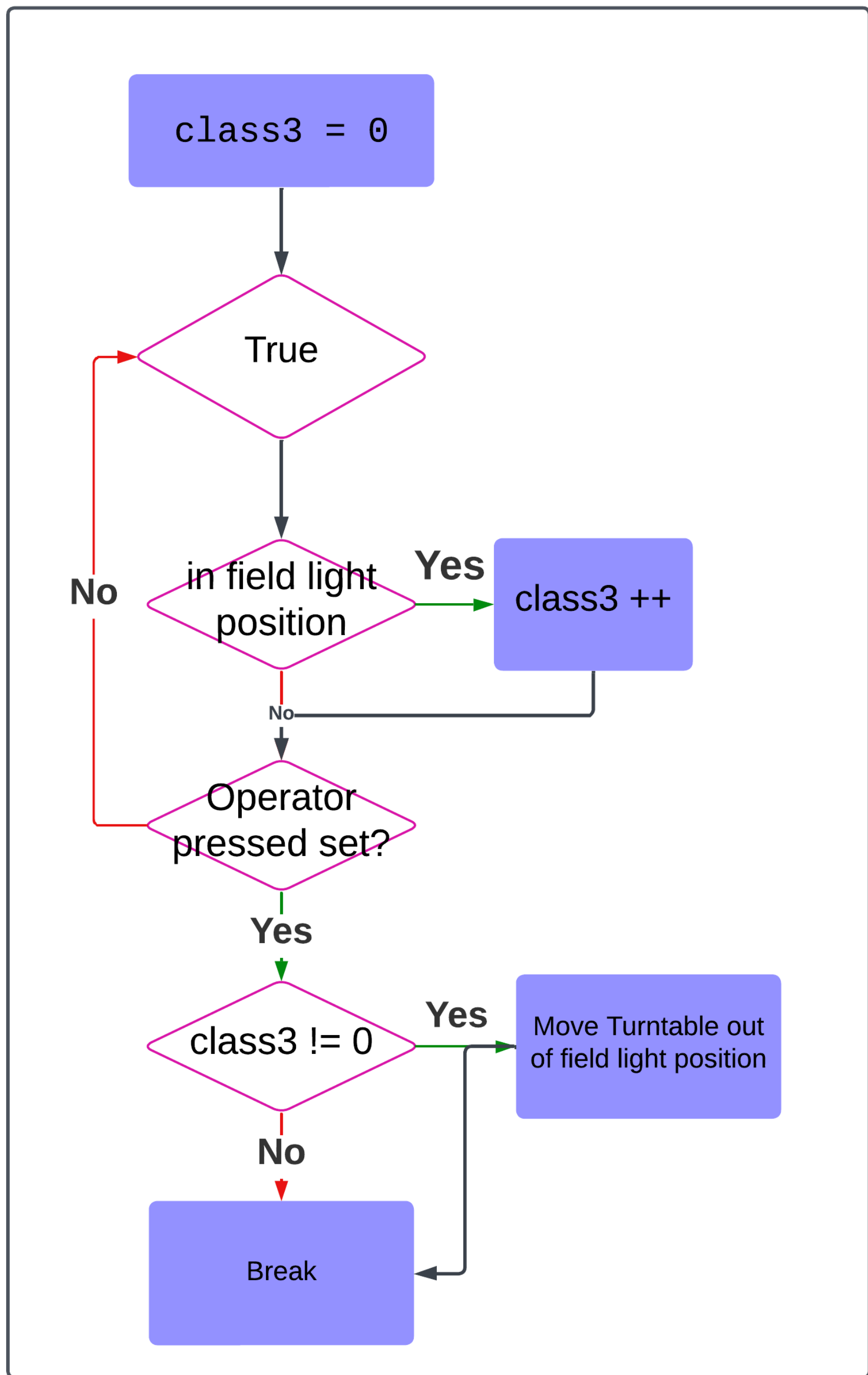
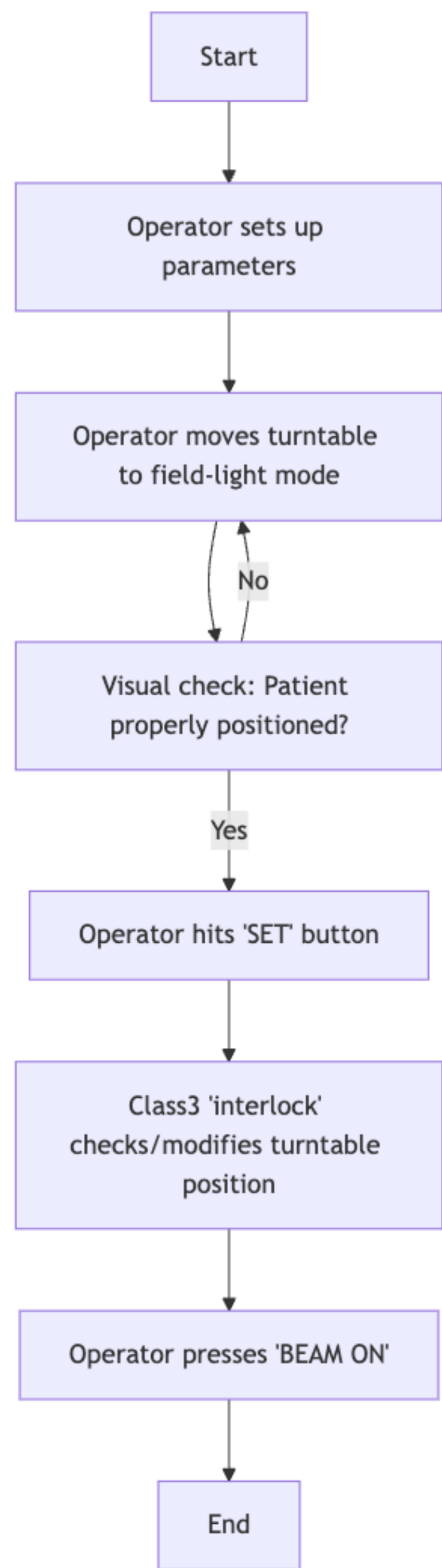
What should have been done?

Keyboard

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



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!

Lab 3 is Released Today!

Lab 2 is Due Tomorrow!