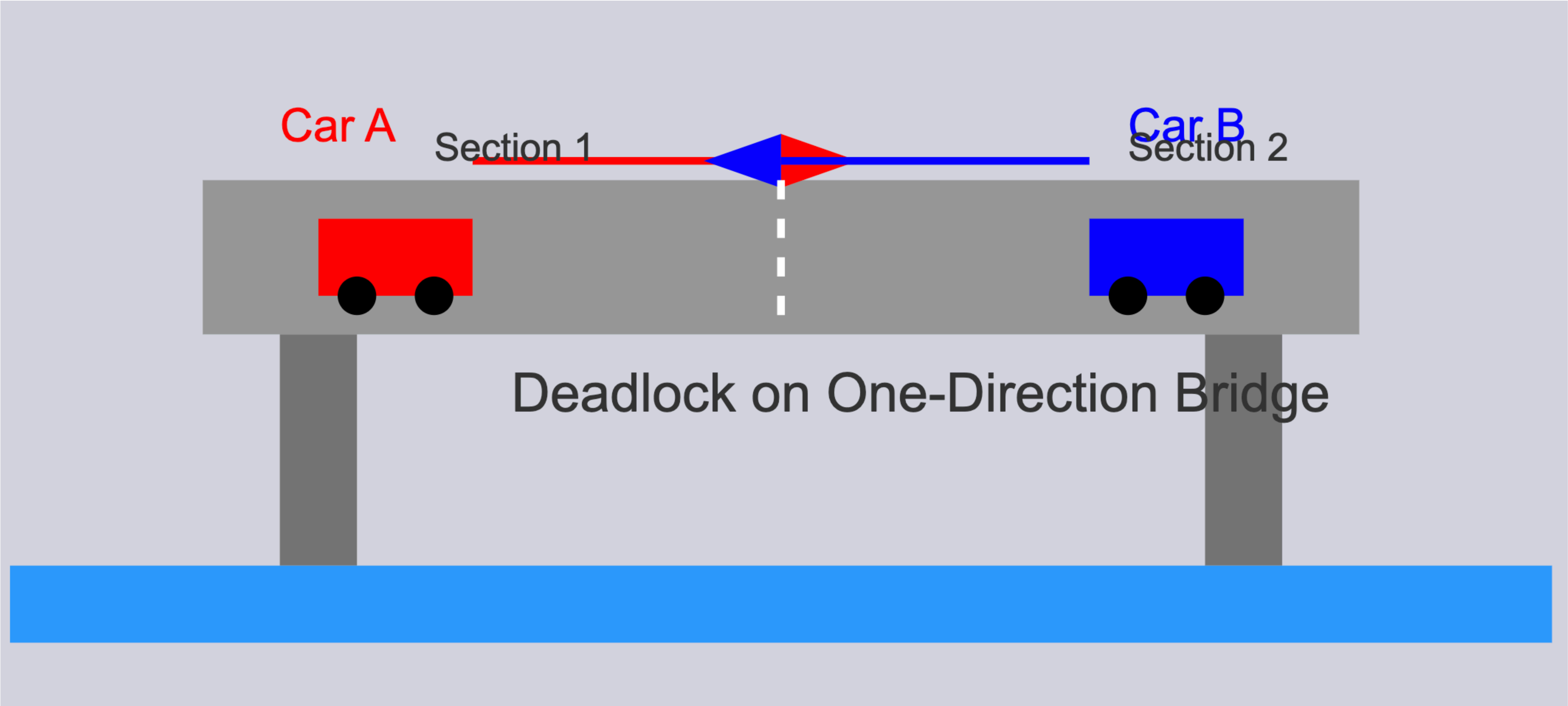
CS202 (003): Operating Systems
Concurrency V

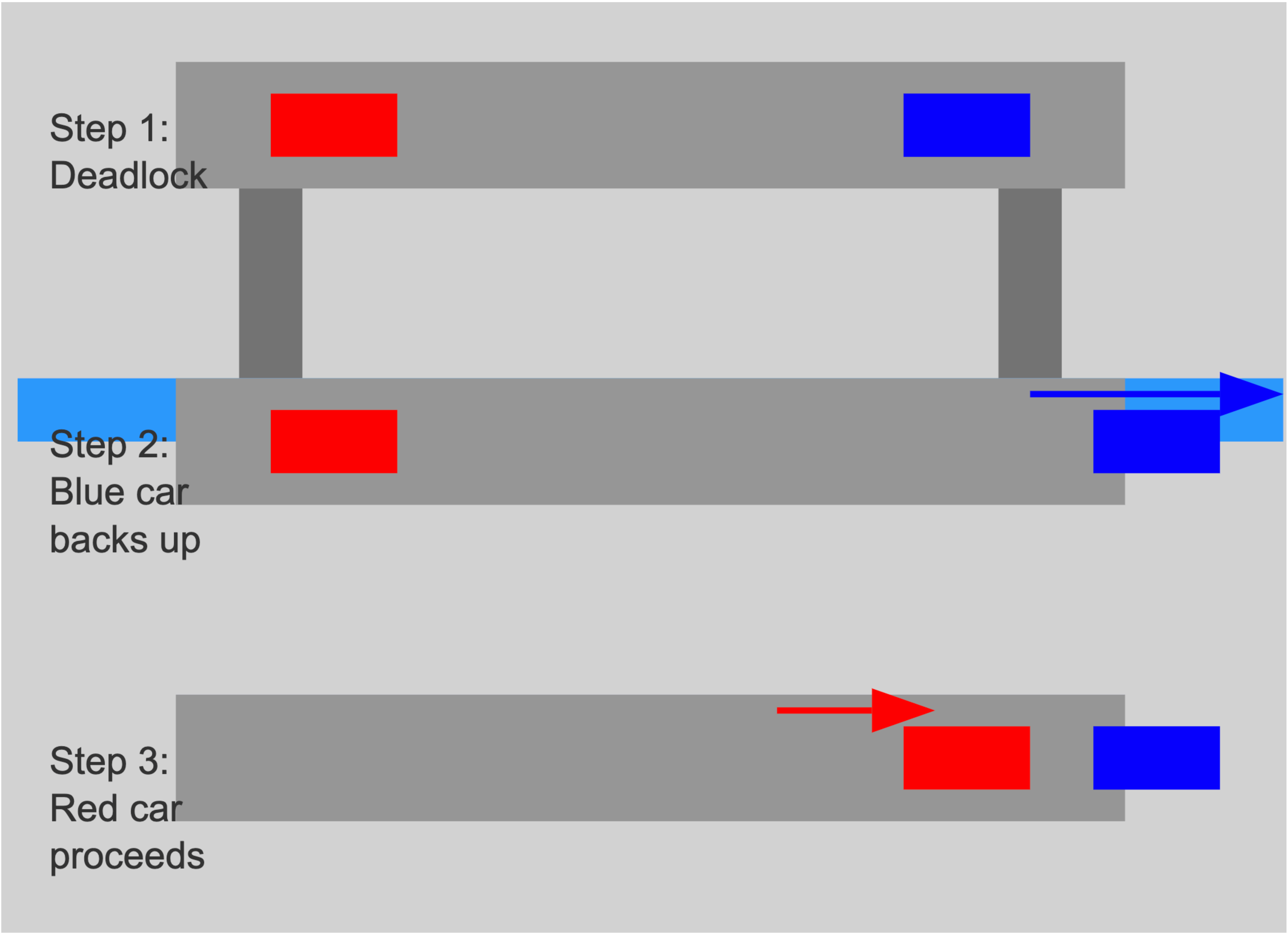
Instructor: Jocelyn Chen

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

Deadlock


Car A
Section 1
Car B
Section 2
Deadlock on One-Direction Bridge

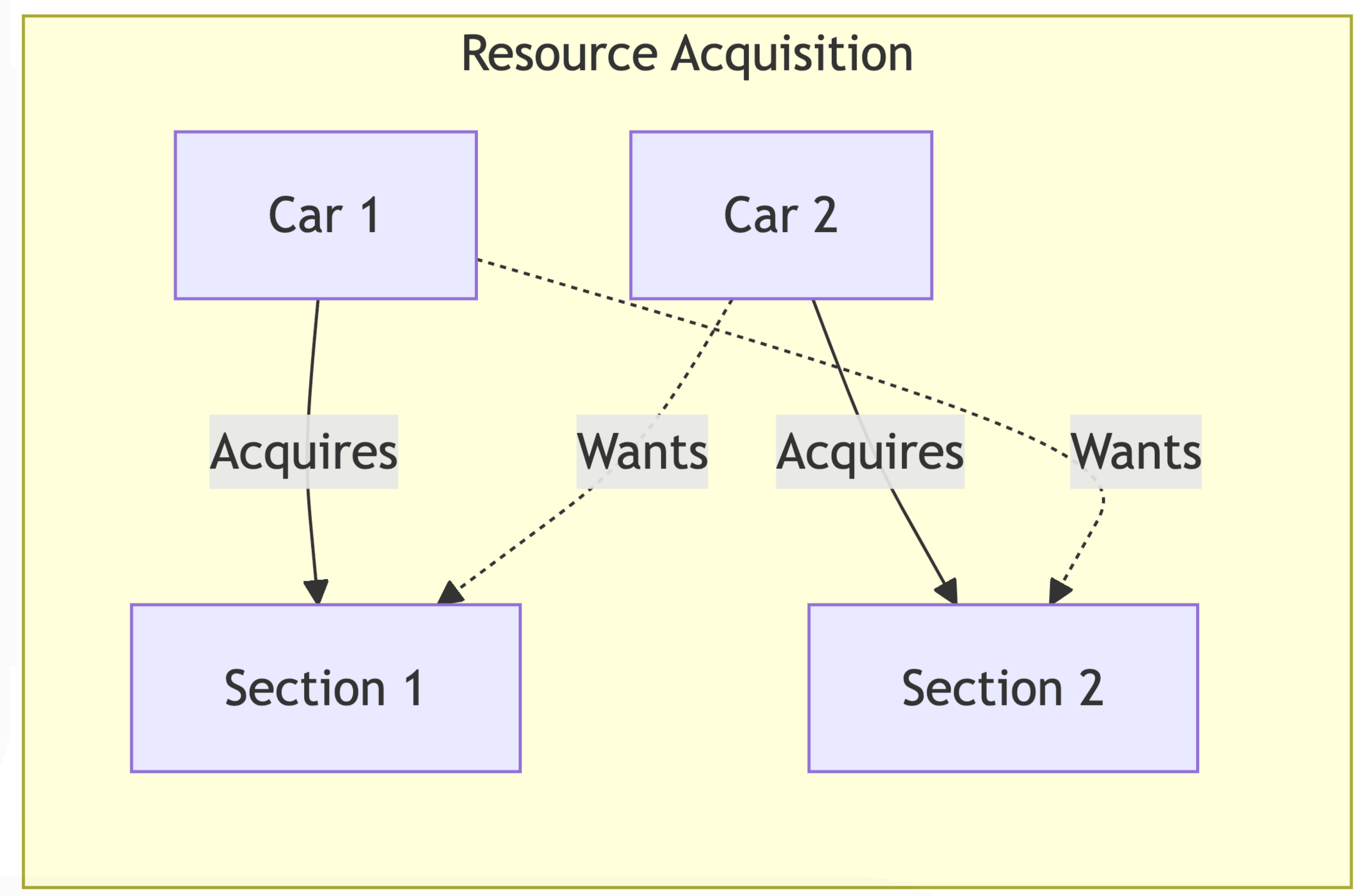
Deadlock


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 class[1]
Negate Any of the Conditions	**Mutual exclusion** put a queue for accessing resources; **Hold and wait** not likely to work; **No preemption** modify virtual memory inside a virtual machine; **Circular dependency** put partial order on locks (=> no cycles)
Static/Dynamic Analysis	Static: detect potential errors without running the code[2] Dynamic: detect during execution, but before deadlock occurs[3]

Check the following if you are curious:

[1]Section 6.5.3 of Modern Operating Systems (Tanenbaum)

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

[3]Savage, 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 reproduce 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

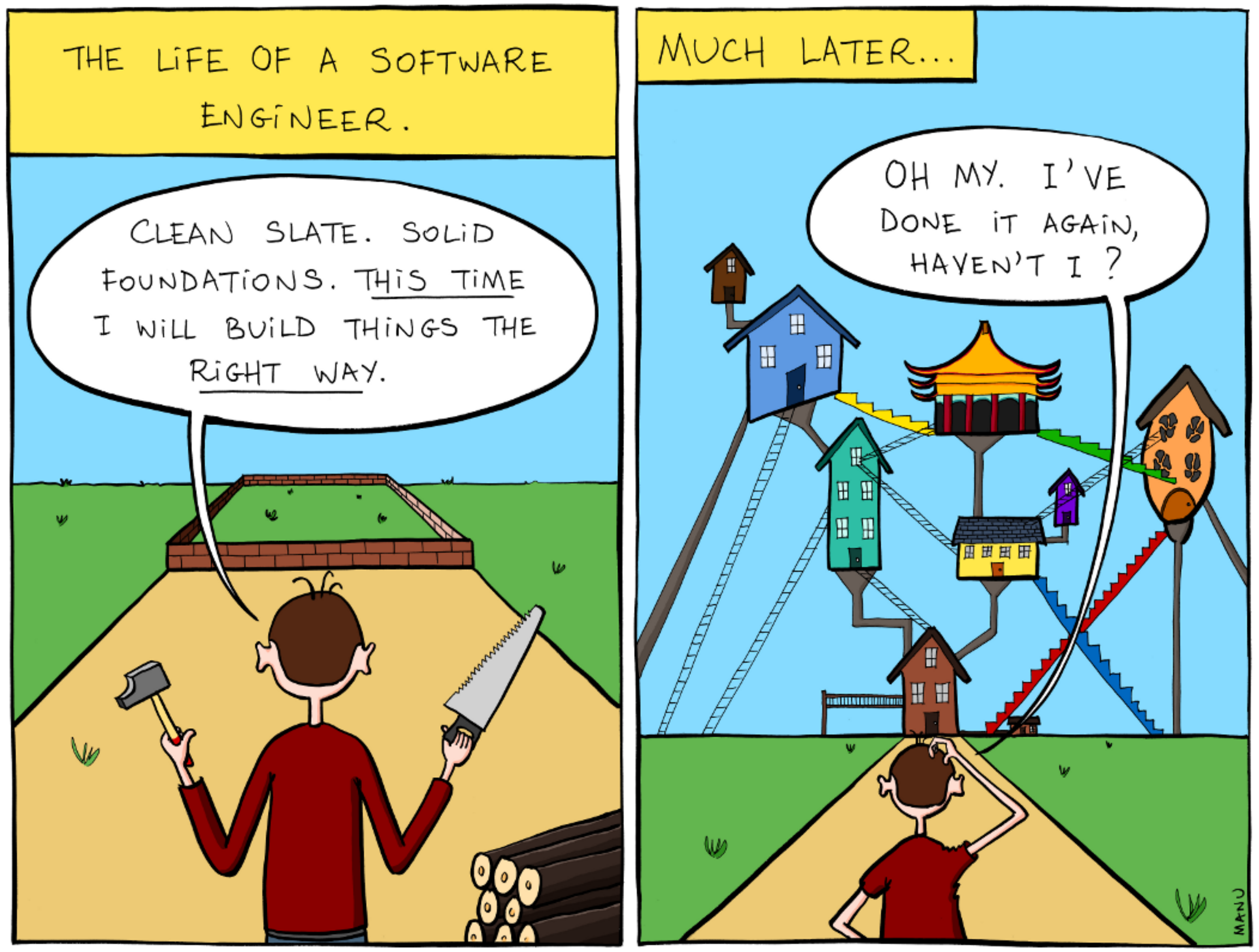
*Look up "MCS locks" if curious

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 guarantee sequential consistency

https://gunshowcomic.com/648

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)

MUST 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
```

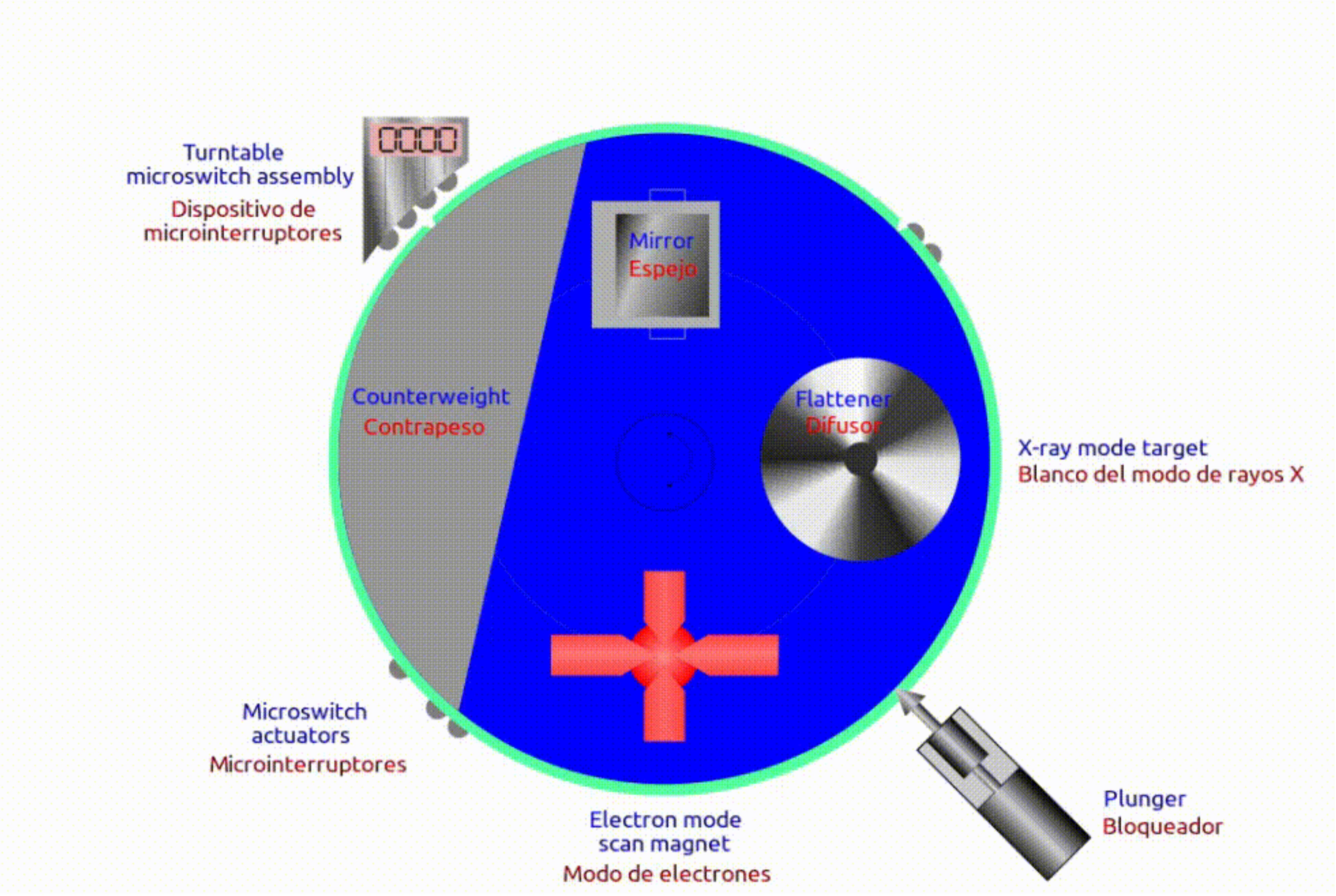
If any memory write after **MFENCE** (in program order) is visible to another CPU, then that other CPU also sees all memory writes before the **MFENCE**

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

"xchg" on x86 includes an implicit memory barrier

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

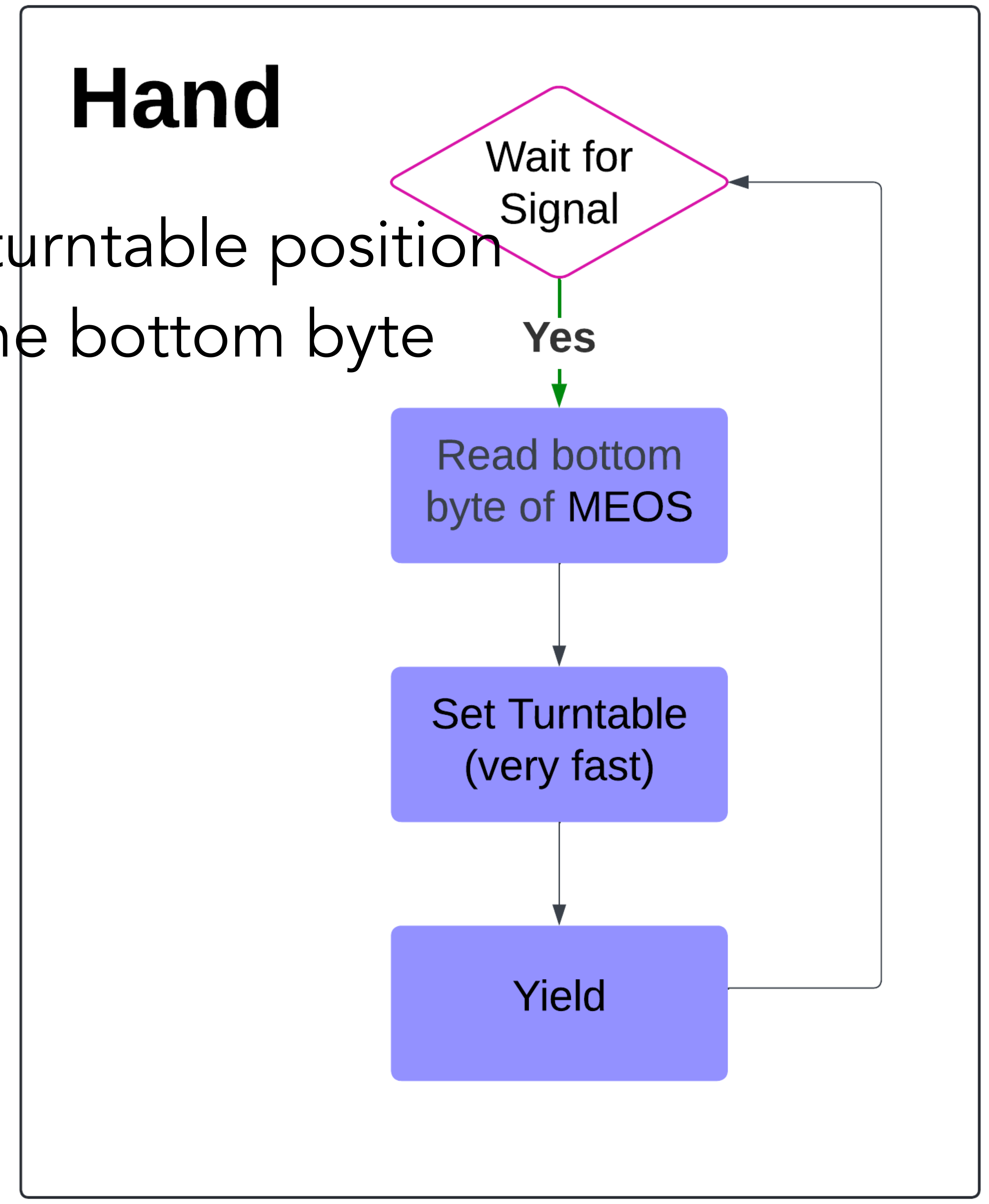
Keyboard

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

Software problem I

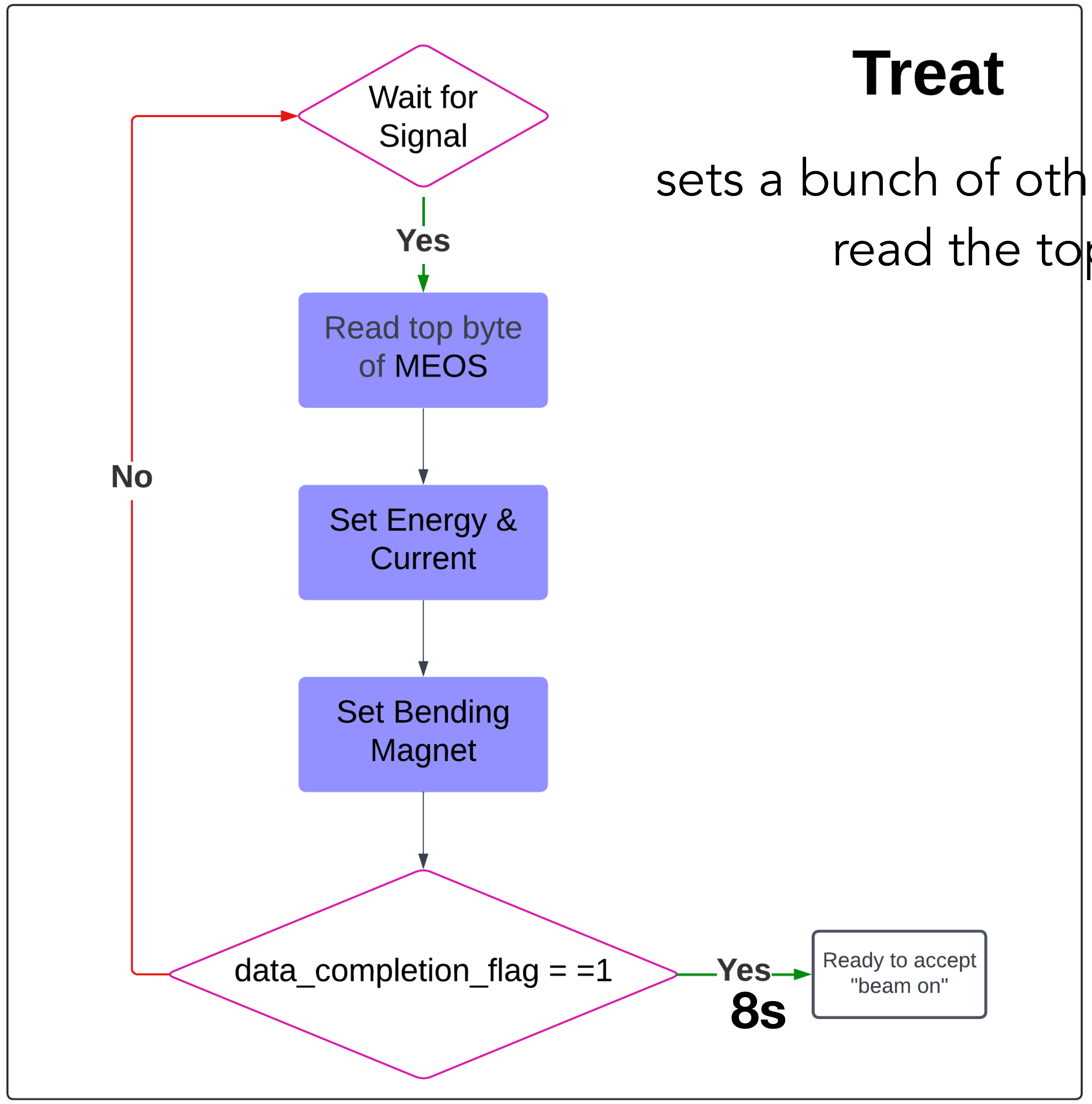
Hand

sets the turntable position
read the bottom byte



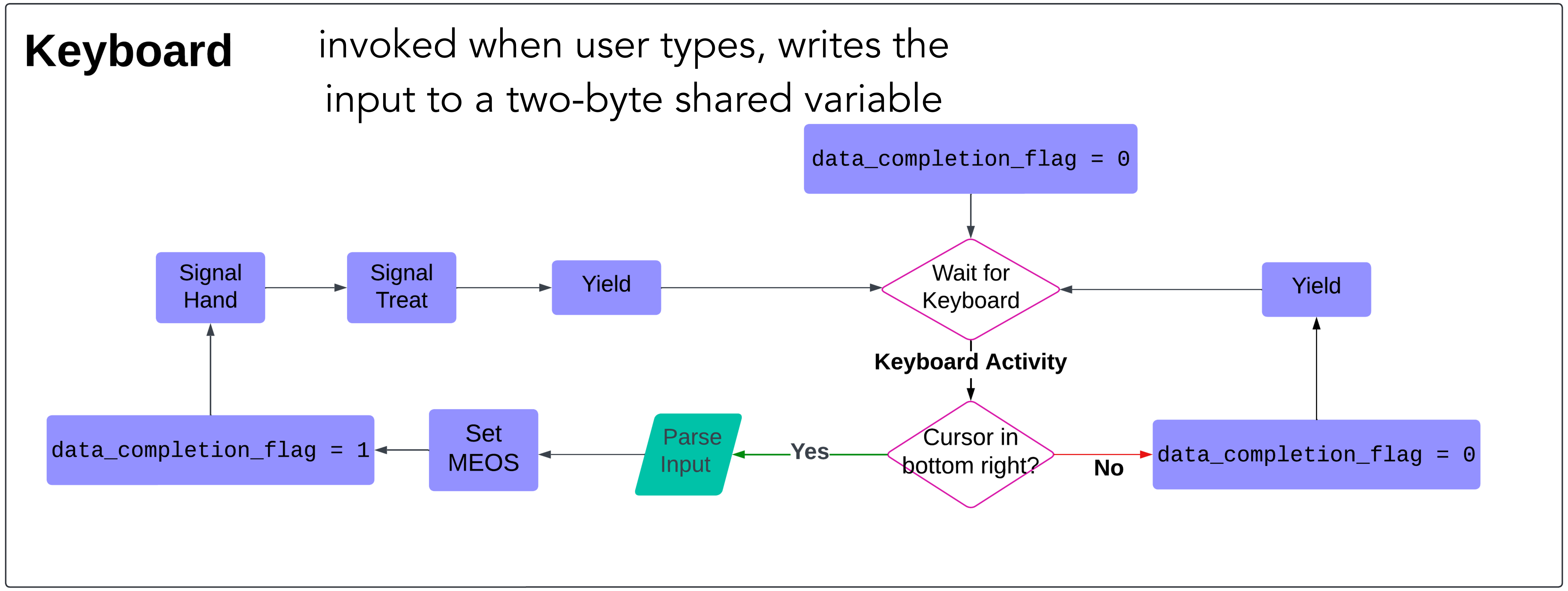
Treat

sets a bunch of other parameters
read the top byte



Keyboard

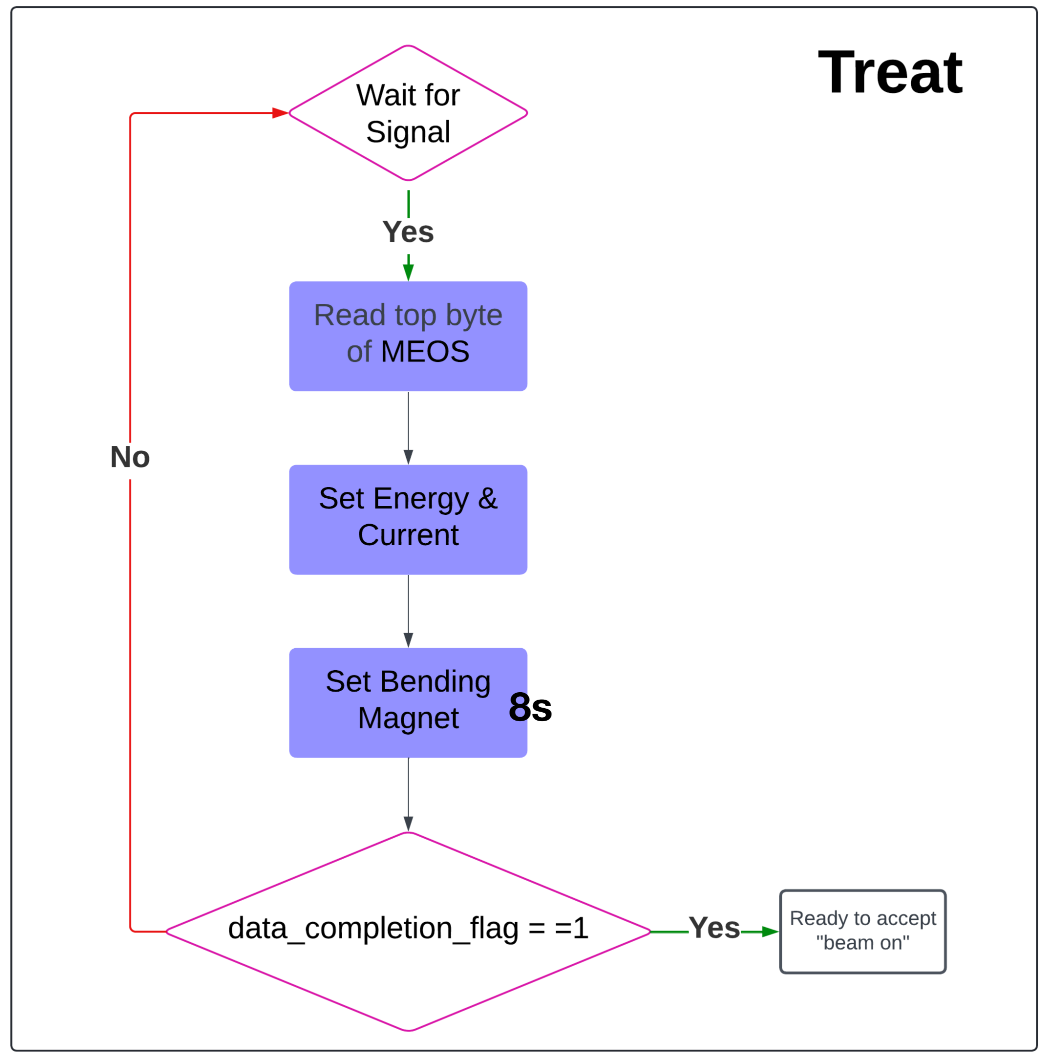
invoked when user types, writes the input to a two-byte shared variable

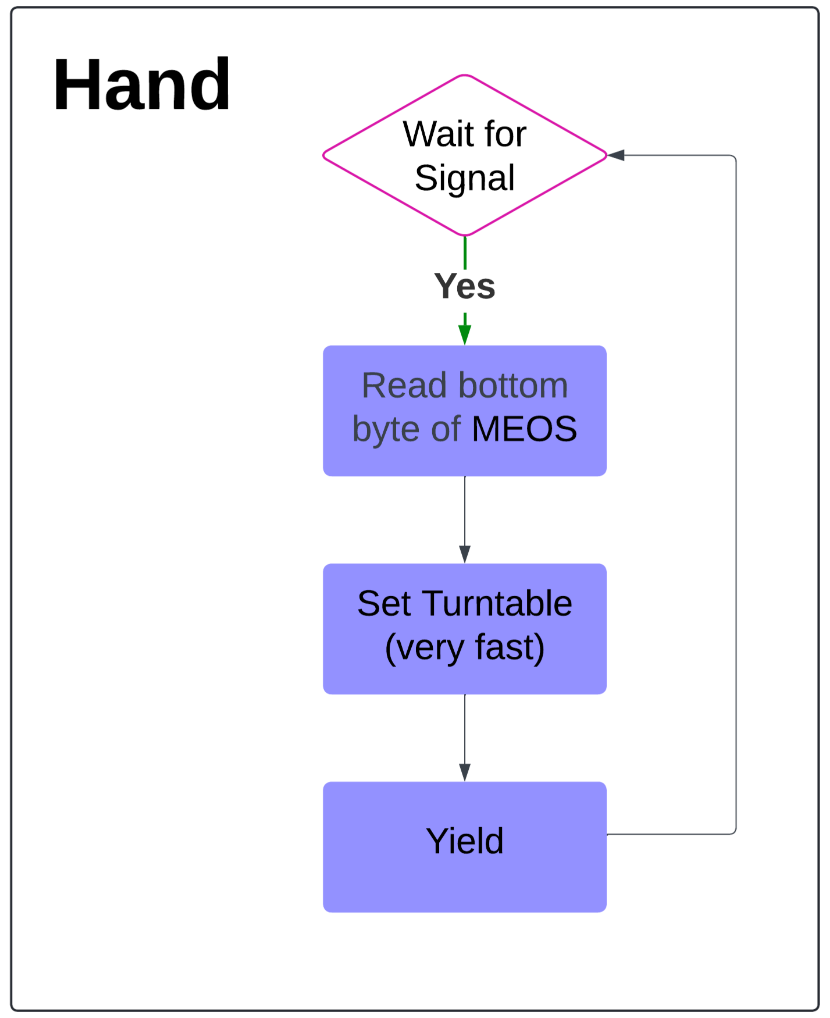


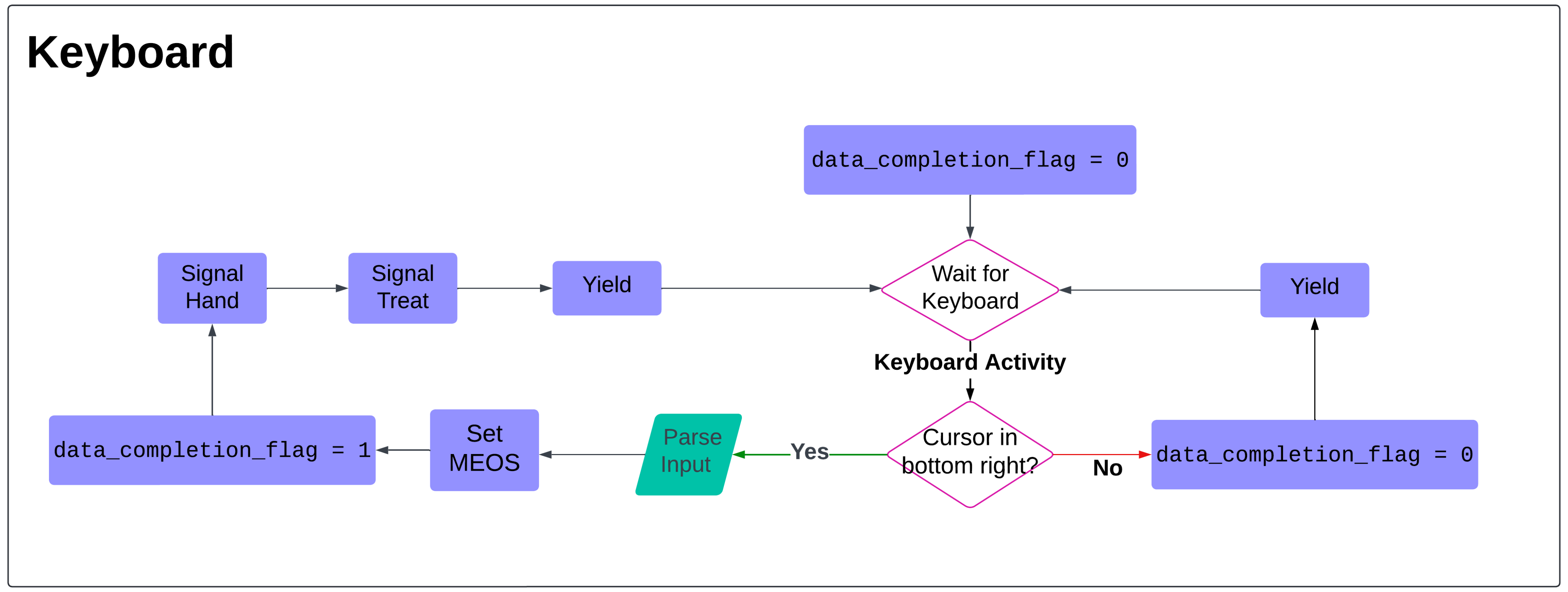
Software problem I

If the operator sets a consistent set of parameters for x (X-ray (photon) mode), realizes that the doctor ordered something different, and then edits very quickly to e (electron) mode, then what happens?

- if the re-editing takes less than 8 seconds, the general parameter setting thread never sees that the editing happened because it's busy doing something else. when it returns, it misses the setup signal
- now the turntable is in 'e' position (magnets)
- but the beam is a high intensity beam because the ‘Treat' never saw the request to go to electron mode
- operator presses BEAM ON -> patient mortally injured



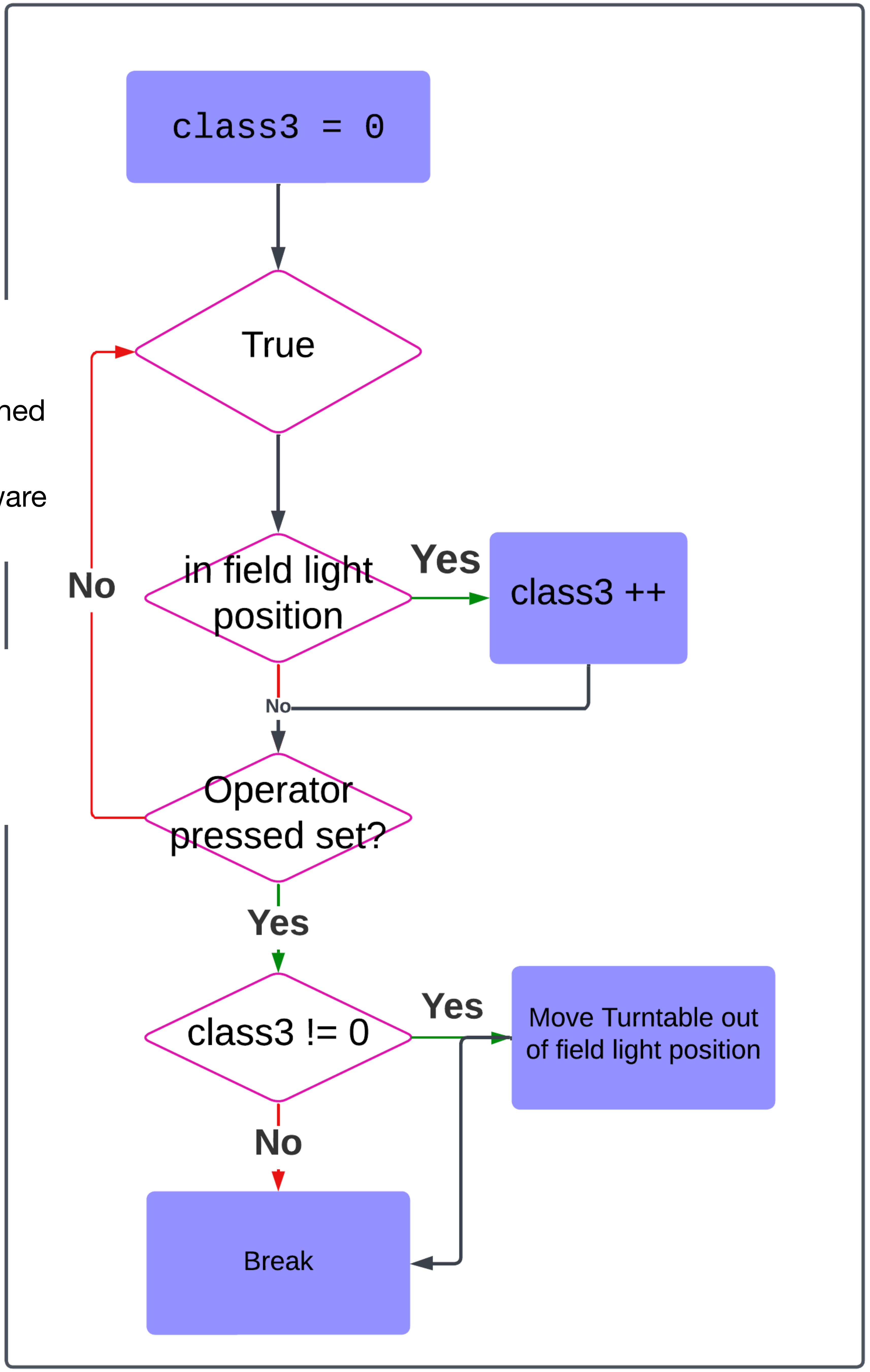




Software problem II

how it's supposed to work:
- operator sets up parameters on the screen
- operator moves turntable to field-light mode, and visually checks that patient is properly positioned
- operator hits "set" to store the parameters
- at this point, the class3 "interlock" (in quotation marks for a reason) is supposed to tell the software to check and perhaps modify the turntable position

So what goes wrong?
- every 256 times that code runs, class3 is set to 0, operator presses 'set', and no repositioning
- operator presses "beam on", and a beam is delivered in field light position, with no scanning magnets or flattener -> patient injured or killed



What else are wrong?

Software Engineering Issues	System Design Failures	Human Errors
No real quality control (lack of unit testing …)	**No end-to-end consistency checks**	**Assume software is always correct**
Complex and poor code	**No backup plan to tolerate error (like using hardware interlocks)**	**“Think” errors are fixed without enough formal reasoning**
Use old code without much thinking	**Not readable error messages**	**Company did not inform the failures, user weren’t required to report failures**
No documentation of software design	**No error documentation**	**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

……