# CS202 (003): Operating Systems Concurrency V

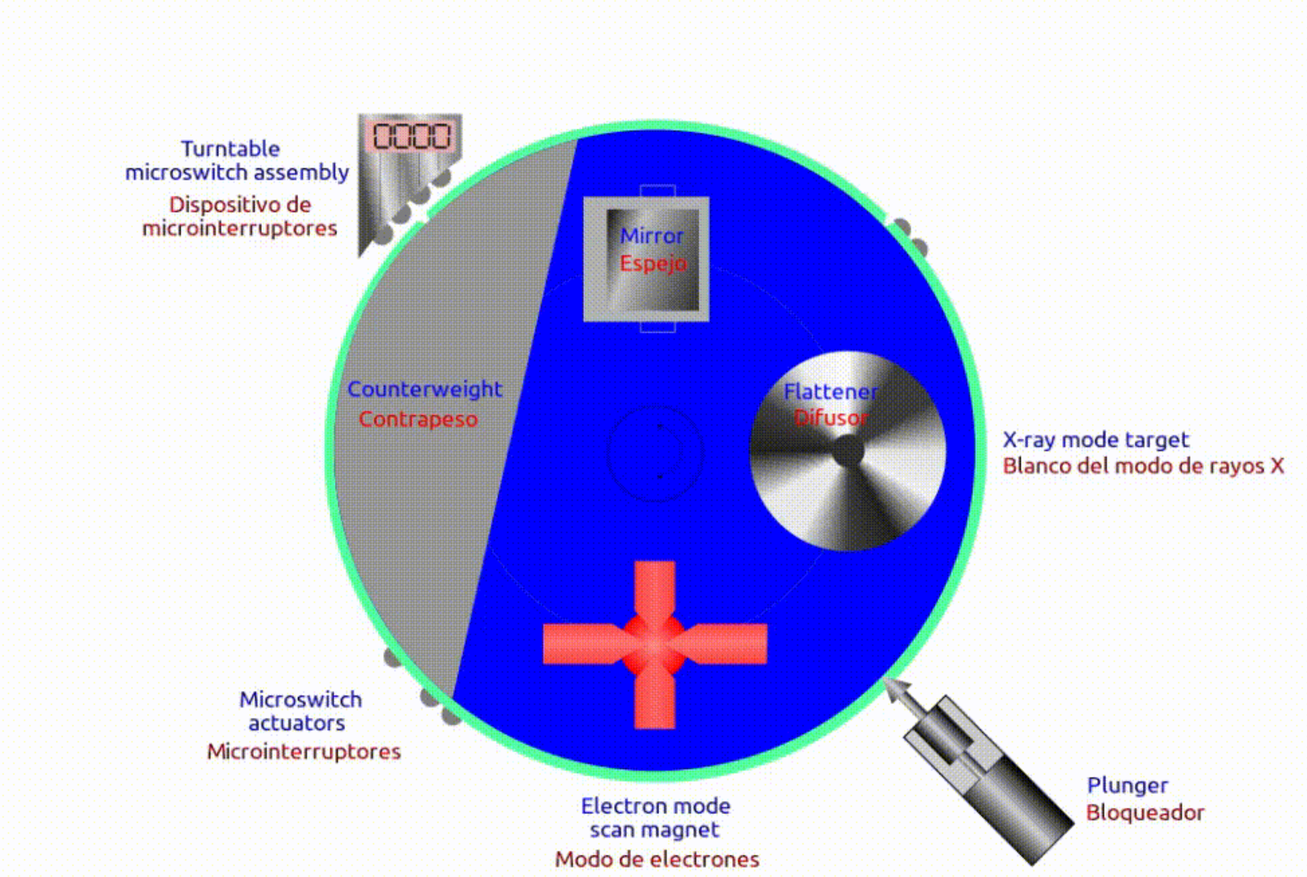
Instructor: Jocelyn Chen

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

# Last Time

# 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

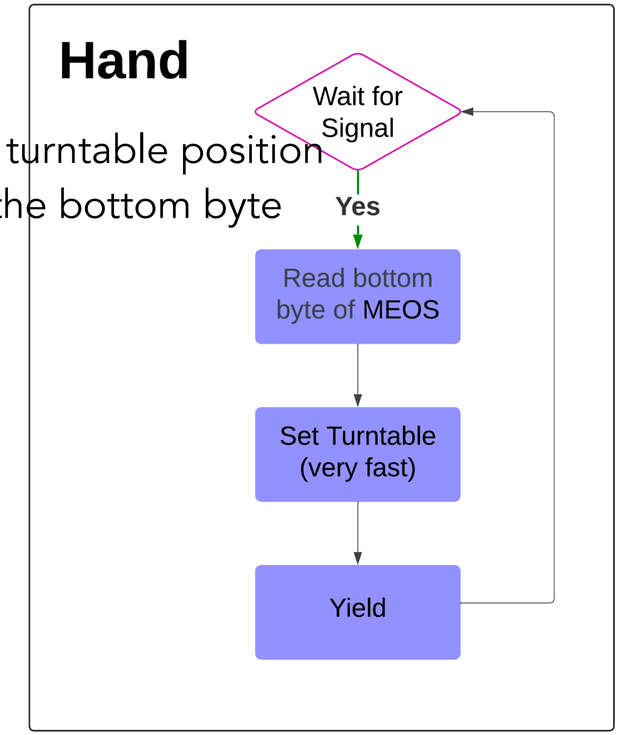
## Keyboard

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

# Software problem I

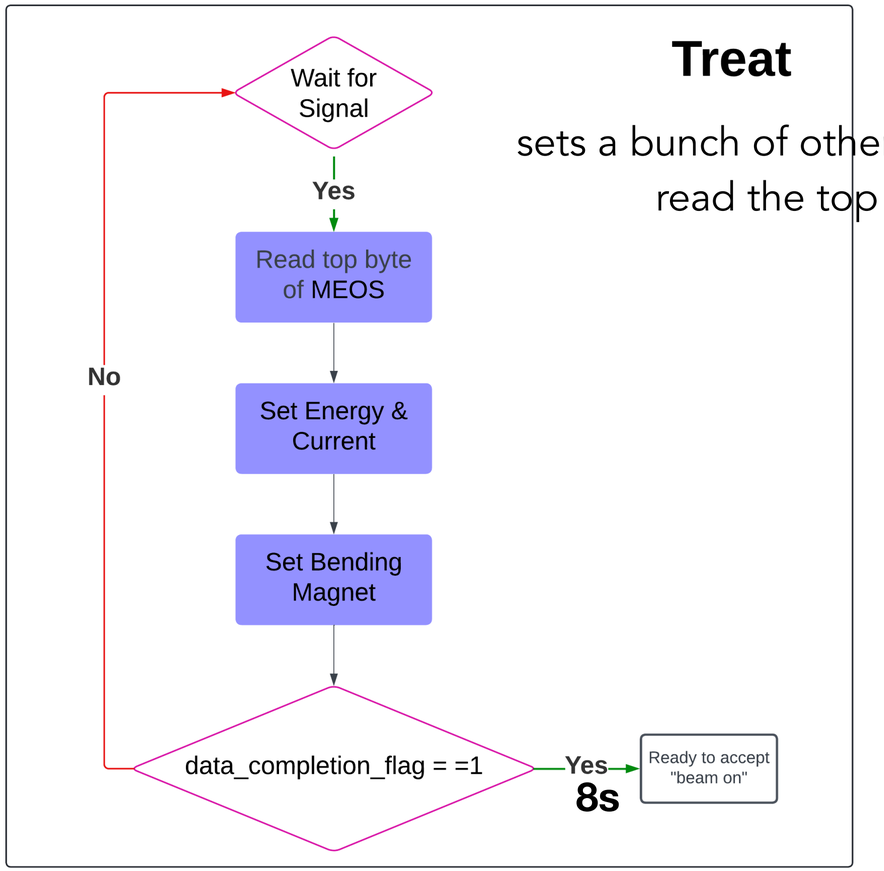
## Hand

sets the turntable position
read the bottom byte



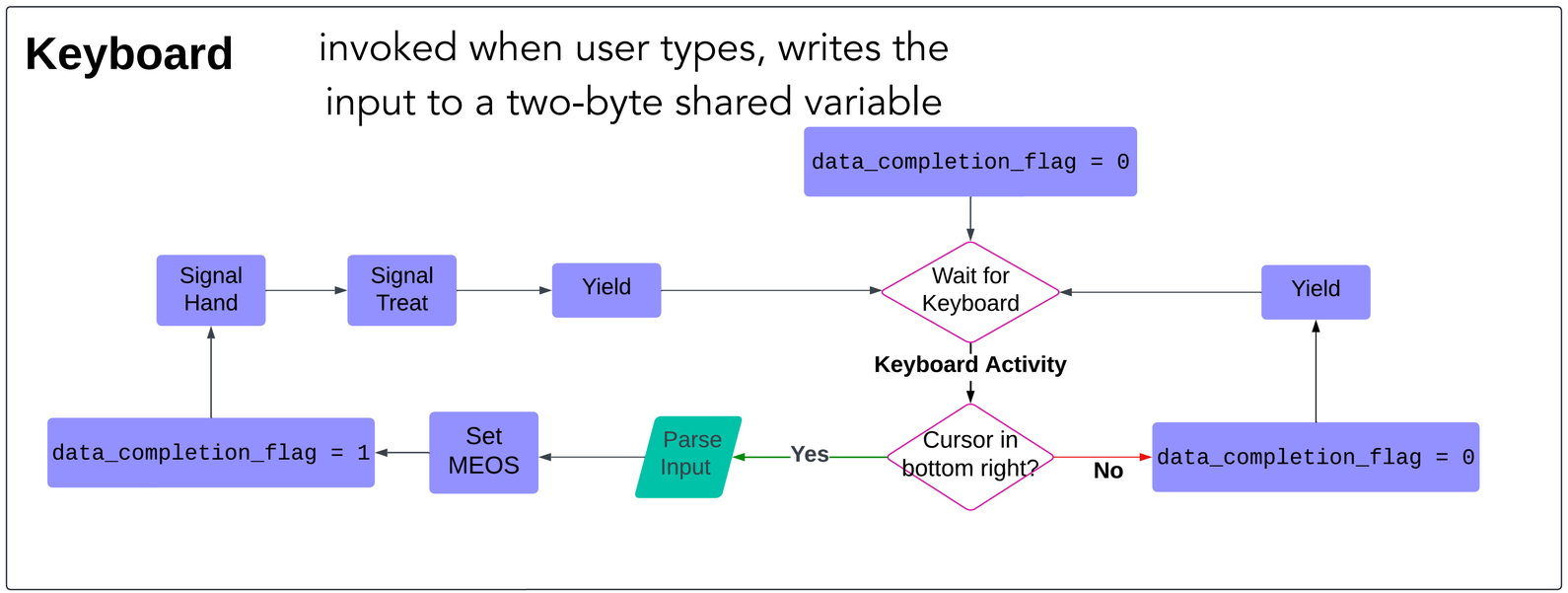

## Treat

sets a bunch of other parameters
read the top byte

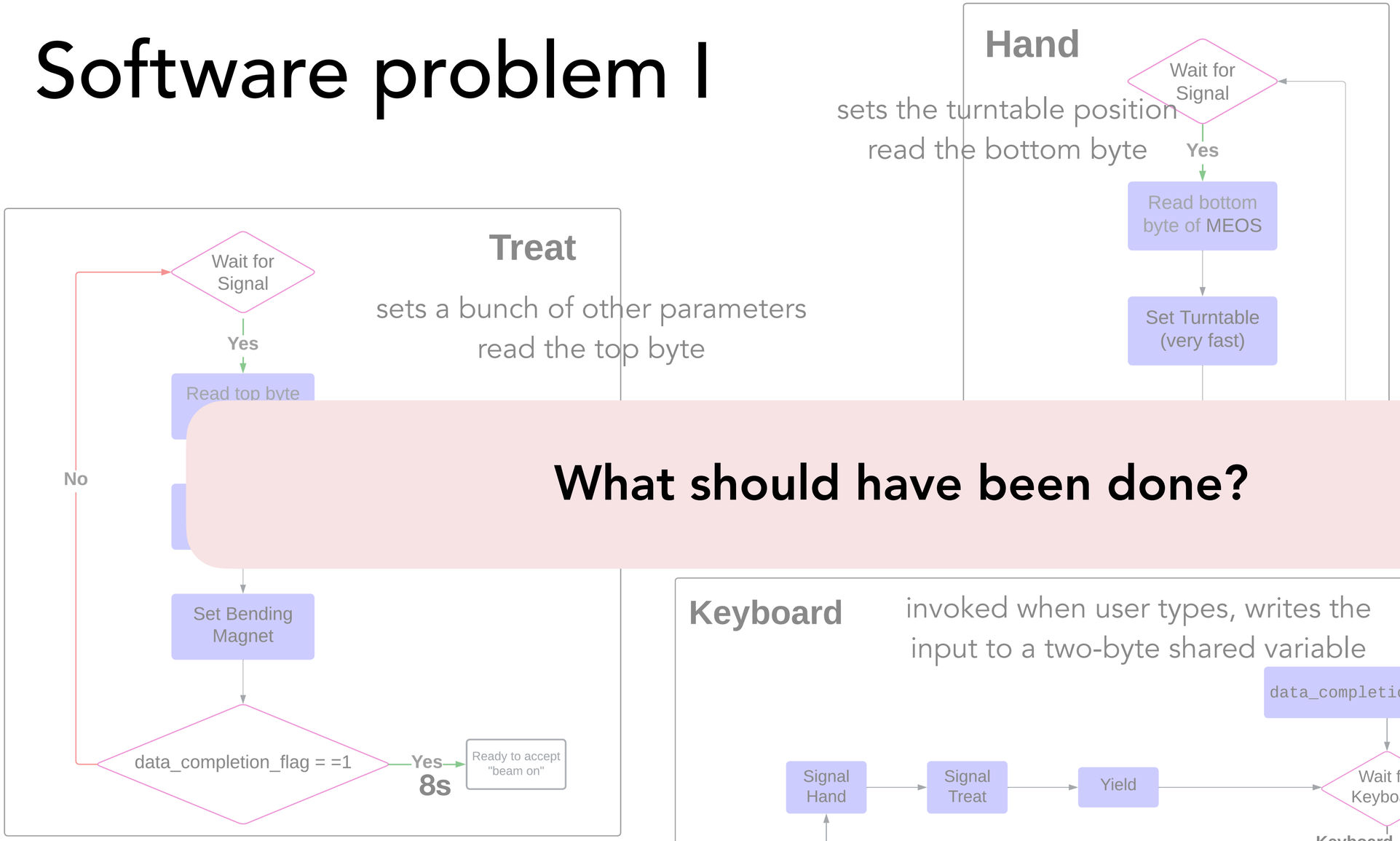



## Keyboard

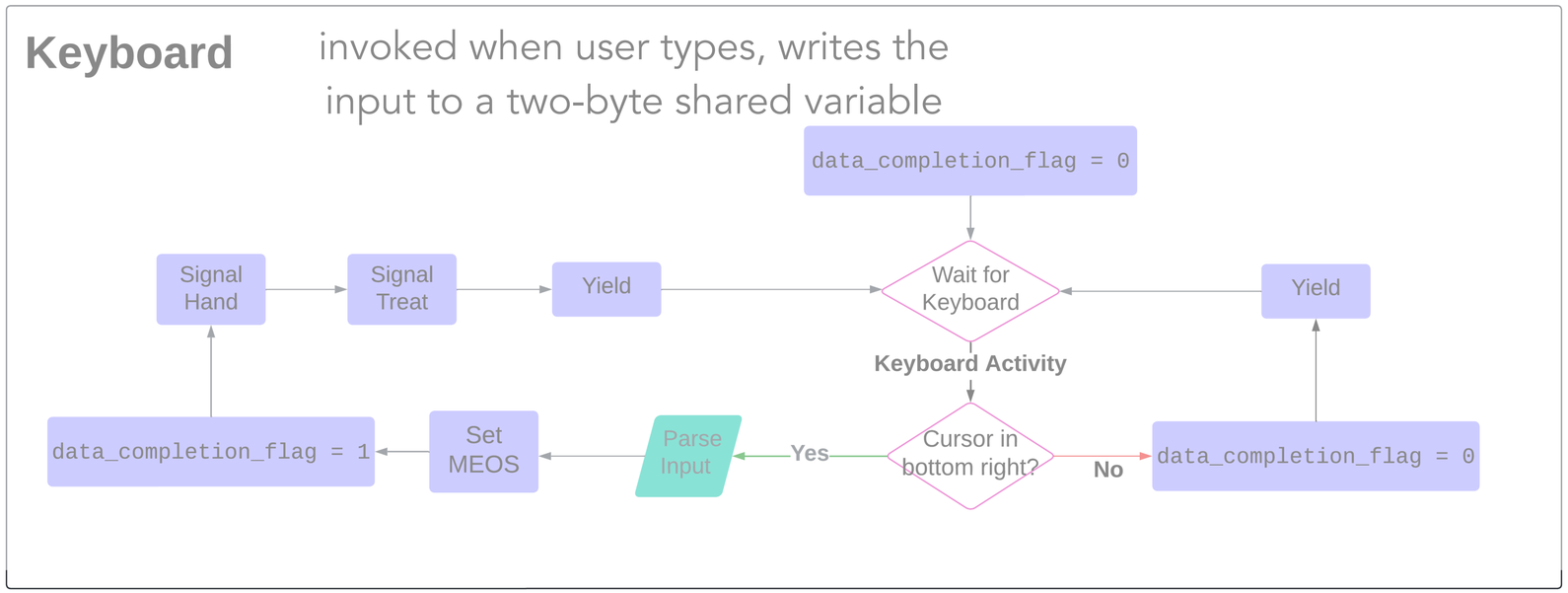
invoked when user types, writes the input to a two-byte shared variable



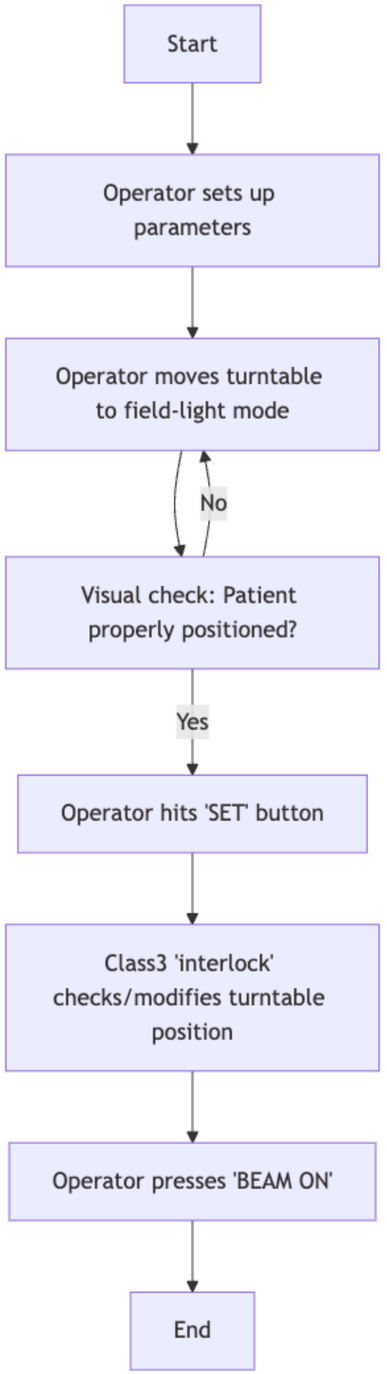
# Software problem I

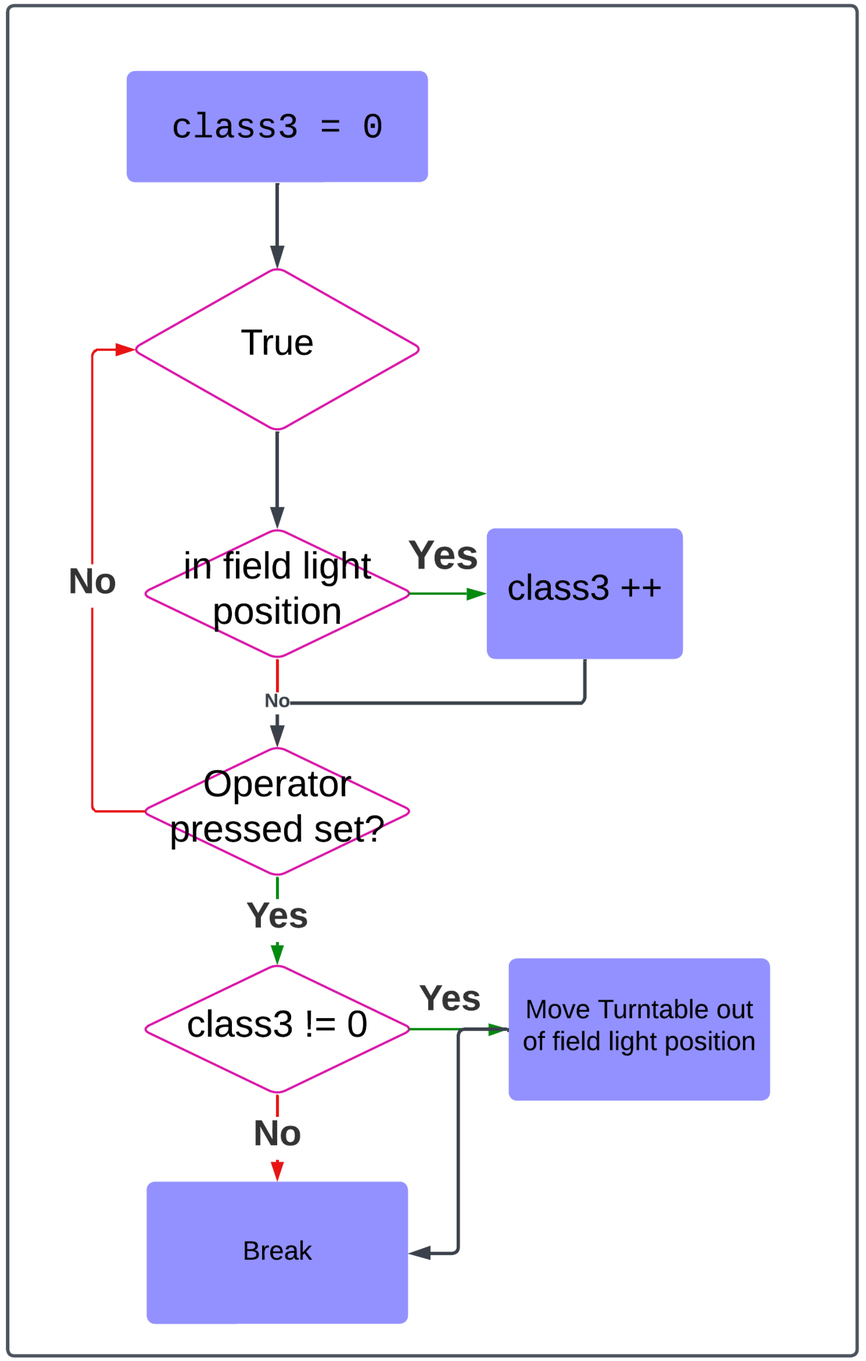



**What should have been done?**



# 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.”*

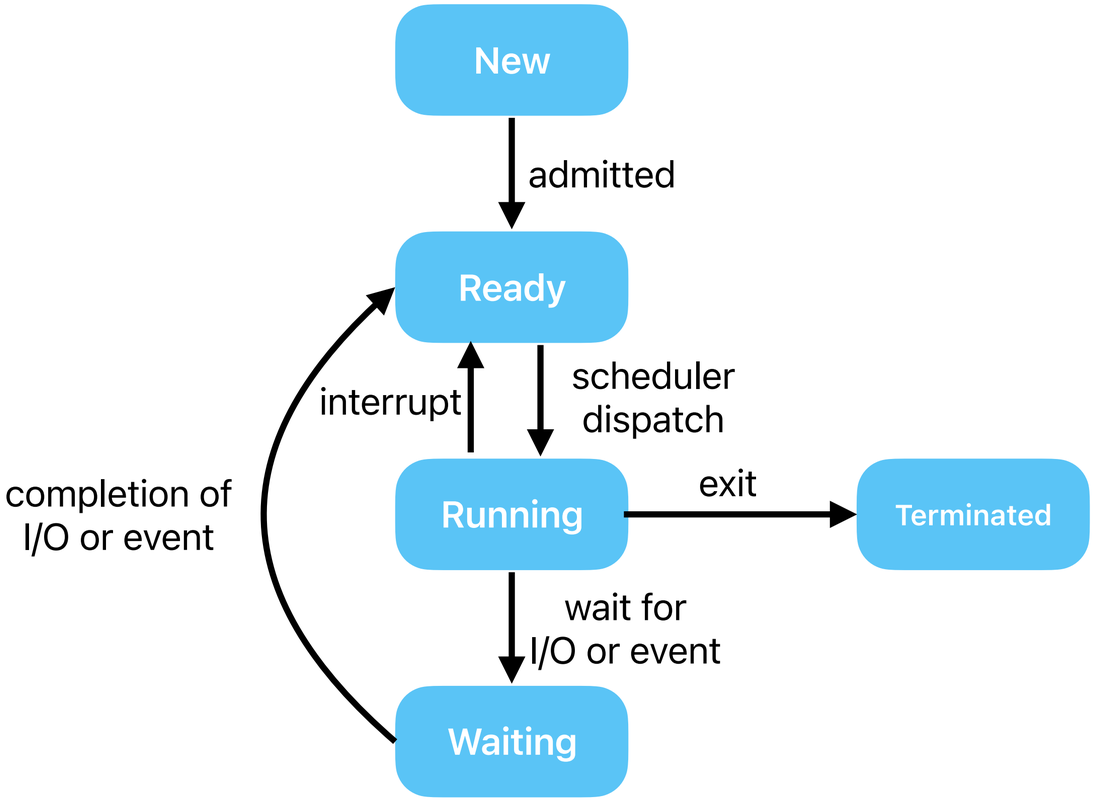

*Theme in building systems: be tolerant of inputs / be strict about outputs!*

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

Have you ever wondered how we decide what next process/thread to run?

Operating system has to decide on this!

# When scheduling decisions happen




(i) switches from running to waiting state
(ii) switches from running to ready state
(iii) switches from waiting to ready
(iv) exits

**Preemptive scheduling**
willing to stop one process from running in order to run another

**(i), (ii), (iii), (iv)**

**Non-preemptive scheduling**
run each job to completion before considering whether to run a new job

**(i), (iv)**

# What are the metrics and criteria for making decisions?

## Turnaround time

Time for each process to complete (from arrival)

## Waiting/Response/Output time

Time spent waiting for something to happen

**Response time:** time between when jobs enters system and starts executing

**Output time:** time from request to first response

## System throughout

# of processes that complete per unit time

## Fairness

(different possible definitions )

Free from starvation

All users get equal time on CPU

Highest priority jobs get most of CPU

……

# We call …

Stopping one running process temporality and resuming (or starting) another process

Context Switch

Context switching has a **cost**!

**CPU time in kernel:** save/restore registers, switch address spaces

**Indirect cost:** TLB shootdown, processor caches, OS caches

More frequent context switches will lead to worse throughput (higher overhead)

# Scheduling disciplines (without I/O)

FCFS/FIFO

SJF and STCF

Round-robin (RR)

# FCFS/FIFO

Run each job until it's done

| Job | Time Needed (s) |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |


**Throughput** = $\frac{3 \text{ jobs}}{30 \text{ seconds}} = 0.1 \text{ jobs/second}$ **Avg Turnaround Time** = $\frac{24 + 27 + 30}{3} = 27$

How can we lower avg turnaround time?


Advantages

- simple
- no starvation
- few context switches

Disadvantages

- short jobs get stuck behind long ones!