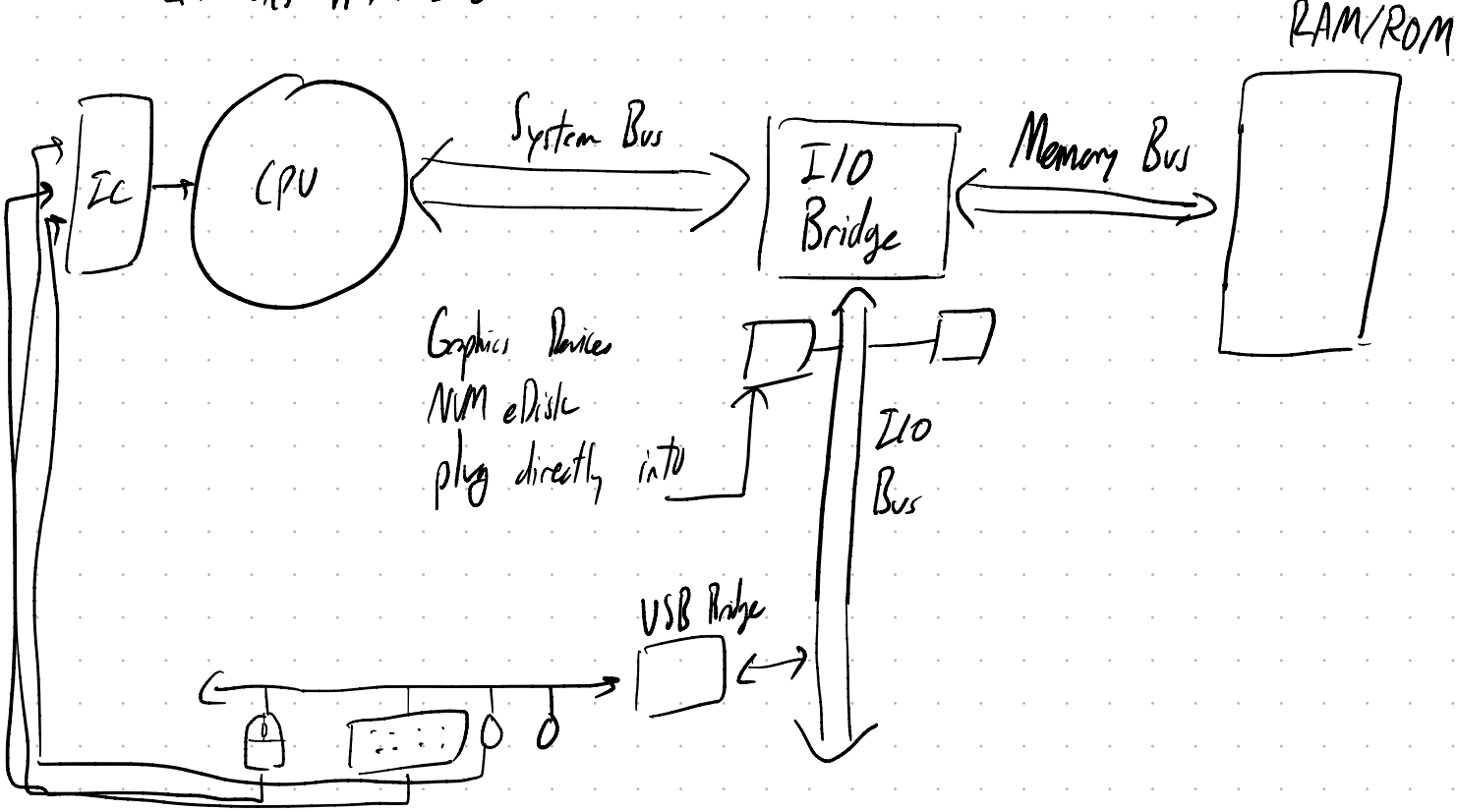
9/12

Little Book of Semaphores

Processor Fetch Decode Execute Update PC
- Loop - don't point to next instr, point back to some place
- Inefficiencies with I/O



USB Bridge speaks cheaper language

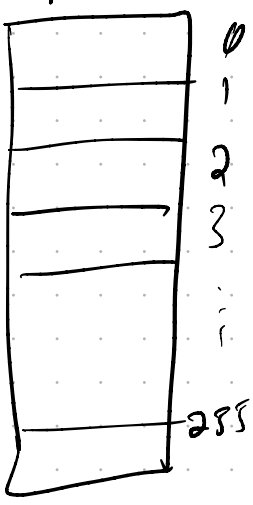
If you click key on keyboard, what happens?
- USB Bus → I/O Bus → System Bus
- 1st Opt
  - Can poll keyboard and ask if input
    - Expensive, most likely doesn't have - humans are slow
    - Wasting CPU time, buses, etc
- 2nd

- Logical wire directly from keyboard to CPU - Still kinda expensive
- It has something change state of wire
- Sending message through buses
- Wire just has info keyboard was instructed, then core goes and asks keyboard for keyboard
  - Logical wire just says there was interrupt, which device interrupted
- Can't have as many wires to core as number of devices - variable # of devices
- Instead one device all devices connected to, devices interrupt passed through one wire into core - Interrupt Controller

fetch decode execute - Sends interrupts to core core at a time info about device number

FDX cycle stops normal cycle fetches from custom place when interrupted
- All done by hardware

256 types of interrupts



IDT - interrupt descriptor table
- ids of interrupt types
- Pointer to piece of code that knows how to handle each type of interrupt - ISR - interrupt service routine

IC sends id to core, core follows ISR to fetch info
IDTR (points to IDT) saved in memory

- Points to beginning and saves size

IDT pointers also carries mode to run in (16-bit, 32-bit, etc.)

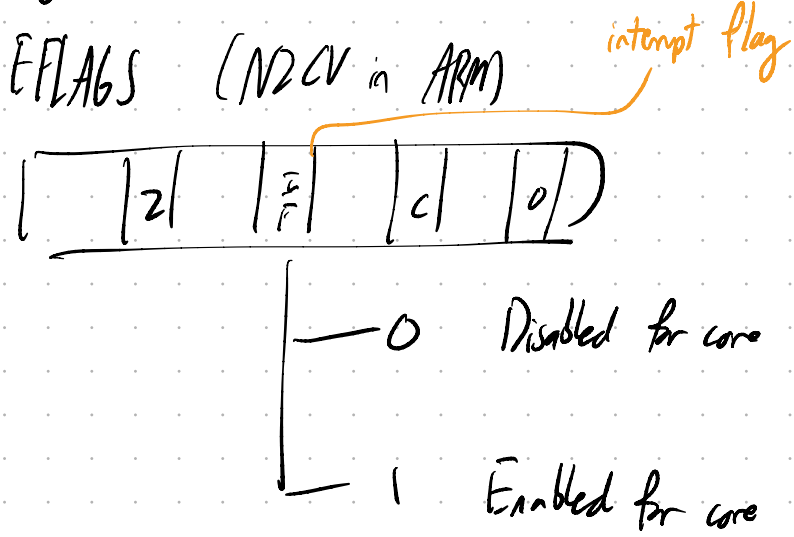
ISR is exit handler in our case (Program counter, current mode, flags)

ISR typically pushes everything onto stack so didn't know would be interrupted
- Otherwise conflict btwn registers in previous process and ISR
- Now risk of overflowing stack if interrupt during interrupt handler
  - Usually just disable interrupt during interrupt handler

By time running ISR, interrupts disabled until returns
- How to disable then enable?

x86

EFLAGS (NZCV in ARM)



pushf pushes all flags onto stack

popf pops

(Can use in context_switch - just need to change stack alignment)

When to enable/disable interrupts?

tcb = active[SMP::me()]

- Can be interrupted during call
- Get SMP::me() = 1, then interrupted before indexing into array, force yield, the TCB can get picked up from another core, then (SMP::me) value is wrong

---

SpinLock - Interrupt interaction

- Both protect against race conditions

Interrupts can bring race conditions even to single-core - SpinLock cannot protect

```
→ ldr x9, ...
  add x9, x9, #1
  str x9
```

— Interrupted here, interrupt handle modifies same X9, X9 not right

↓

Use interrupt safe lock

↳ Disable interrupts around critical conditions

Disable interrupt protects core against itself, SpinLock protects against other cores

↳ Both need to be OK() otherwise becomes unresponsive

↳ Sometimes will need both

disable()
spin_lock() ——— Not O(1)
↳ Spinning can take arbitrarily long, can disable for a long

[Critical O(1)

spin_unlock()
enable()

spin_lock()
disable() ——— Can get interrupted in between
- Can get deadlocked

[Critical O(1)

enable()
spin_unlock()

ISL - Specialized lock to combine
- While spinning, disables then enables interrupts

1) Don't use locks if possible
2) Use BlockingLock() if possible - can't use in yield() or inside BlockingLock
3) Use ISL

9/14

Implement ISL to solve problem w/ Spinlock + Interrupt disables

```
disable();
while (flag.exchange(true)) {      <- Atomically swap to true
    enable();
    disable;
}
```

Lock — Looks like cancels out, but enable allows pending interrupts

Unlock `flag.set(false)`

```
thread([]{ ... });
```

↓

stuff here is work done on behalf of the thread

↓

imagine calls yield() → `void yield(){ ... }`

↓

At some point, code is for core to do work for itself

- When messing w/ ready-queue, not thread that's running but core is running

Interrupts are similar

- When thread runs, pit runs, causes interrupt, pitHandler() runs
  - Thread stops running, doing own housekeeping stuff, doing core work

Who can call yield()? The thread or the core?

- Core is always calling everything btw
- Threads are yielding not cores
  - Yield means let another thread run, core itself doesn't yield

stop();
- Thread thing, cores don't stop

Spinlocks - useless

IRQ - Use over Spinlocks

BlockingLocks - Don't spin, instead has queue to add self to
- Threads add self to queue when blocked
- To block, add yourself to waiting queue and context switch to another thread
  - Cores can't block, threads block

init is to convient place - creates no threads → threads show up out of nowhere
- calls stop, malloc which uses BlockingLock

threads using Spinlocks are wasteful
- Cores can spin, if running and nothing else to do, can just spin

threads shouldn't disable interrupts - should be able to be preempted

Only threads can block, heap uses BlockingLock, only threads can do heap things

InterruptHandler runs w/ interrupts disabled
- Doing a core thing, so calling yield() is funky
- Behaving like a thread in context where doing core-only things
- Why problem?
  - Shouldn't be trying to get Spinlock when interrupts disabled
    - Calling yield() means use heap, which uses lock
    - Cannot do lots of work w/ interrupts disabled

AptHandler (interrupt handler) has interrupts disabled

Can add cooperative thread in init,

```
thread_init() {
    while(true) {
        delete_stuff();
        yield();
    }
}
```

- Calls yield in the middle - yield is not short function
- Not good

Solutions

Don't do complex work in interrupt handler
- Off-level interrupt handler. Real interrupt handler runs really fast, when return, returns to another handler w/ interrupt handler w/ interrupts enabled to do more complex stuff instead of where it was
- In theory can get interrupted in off-level handler + never return back to where it was

Don't use ZLL in ready_queue
- Use ready_queue per core, don't need ZLL
  - Can have imbalance where one core can have many things to do, another has none
  - Loses parallelism
- Background activity to rebalance queues

Blocking Lock
Have busy flag, if true, don't spin but add self to waiting queue
- Then switch to another thread + give chance to run
- Details
  - What if try to add to queue then switch + ready queue is empty?
    - Know why come back - b/c lock is unlocked - ready queue is empty?
      - Can have idle thread - make initial TCB idle thread

- What if X gets lock + succeeds, A tries to get lock, is alr taken, add to queue, X releases lock after A tries to lock + before A adds self to queue - X sees queue is empty, doesn't see A is there
- Have ISL for this
  - Core is spinning, not thread

Have waiting queue waiting for all other threads to be added - BlockingBarrier