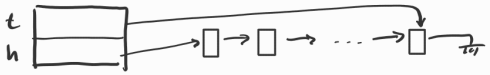
Singly Linked List




```
p = f
if (p == null ptr) return p;
f = p->next;
if (f == null ptr) t = null ptr;
return p
```

* Reading followed by writing actions is what causes issues
* If you have instructions adding and removing them it gets worse

Can we use Atomic to make this work?

need some form of compare & swap
↳ CMP XCHG (compare exchange)

* Lock-Free (but Not Loop free)
* Implemented directly into the architecture
** DOES NOT work for this linked list scenario

```
p = f
if (p == null ptr) return p;
h = p->next;
if (f == null ptr) t = null ptr;
return p
```

This is a point where the assertion might fail incorrectly
↳ if head is null this would be OK if everything tail should also be null

ran sequentially
↳ BUT not in a parallel system
↳ Another thread might catch the invariance
↳ need to use small critical section to avoid this issue

* Assertions are saying "we can't be 100% sure that it works"

Use a spinlock?

* Spinning is expensive so you should typically try and avoid
↳ But not in this class → other things too complex
* In this class: All critical section using spinlock should have a time complexity of O(1)
↳ with small constant (~20-50 instructions

Gheith is LYING TO US!!

removeall() in the code = O(1)
↳ just changes the pointers to null
* there is no garbage collector

* When talking about race conditions, talk about read, modify, write patterns

How can you verify your program works 100% of the time?

* write a code reviewer!! use mathematical models that prove it is correct
* Insert assertions to check the states at the beginning and end
↳understand your invariance and explicitly check it

stop()
You need to context_switch() and delete yourself
↳ WHEN do you delete yourself
* in theory your data should still be accessible through state pointers
↳ BUT: what if another core mallocs your stack

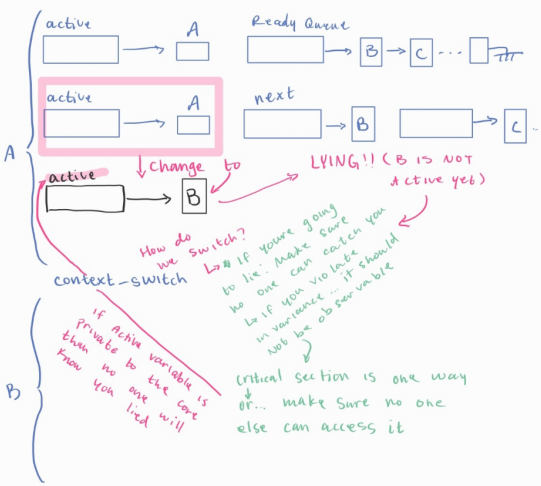
your TCP

③

* Doing stuff after context_switch is easier BUT complicated because you need to pass all the data along
* Doing it during context_switch() means having to implement in assembly
↳also complicated (and need to pass data along)
you can use flags to say theres stuff it needs to run after switching

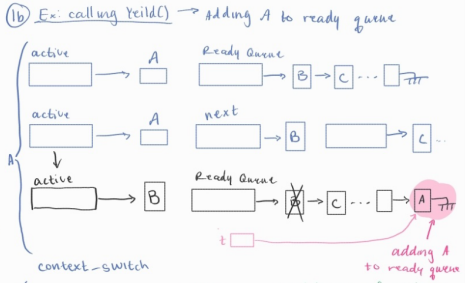
Common Race Conditions

(1a) Ex: Calling Yeild() → setting B to Active



* If you're going to lie: make sure no one can catch you
↳ if you violate invariance: it should not be observable

(1b) Ex: calling Yeild() → Adding A to ready queue



Note: This is harder than previous issue => Ready Queue not isolated to a single core

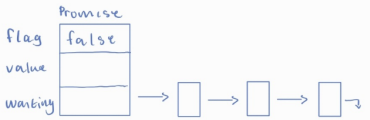
What happens if another thread picks up A BEFORE its been updated and released
↳ New thread might end up somewhere random
↳* Need to implement critical section
↳ can ONLY release lock AFTER context_switch()
↳ BUT you'd have to know exactly where context switch might send you

OR: you can update ready queue AFTER context_switch

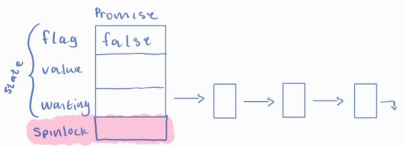
(BETTER) keeps all the operations together

OR: update ready queue in the MIDDLE of context_switch
↳ VERY HARD
↳ probably have to write a function

(2) Promise:



What happens if the value is set at the SAME time that someone is adding themself to the waiting queue?
no one will wake up the thread from the queue



Add a spinlock! — hold the lock for the duration of the context switch

Double Checking Technique:
?? check the state prior to context_switch()
if it's false then context_switch() and check after THEN decide what queue to be added
↳ you lock twice but for MUCH shorter time

* in general: your lock should not depend on another lock
↳ can produce a deadlock
↳ no longer O(1)

Alternative Solution: Give each core an independent ready queue and distribute tasks to each core
↳ high probability of distributing tasks unevenly
↳* Periodically rebalance tasks
↳* This is what most computers do

How do you know how much work a task is?
there's a global array where each core writes how much work a task is
↳ Race conditions don't matter

If there is gross imbalance then a busy core will put a portion of their tasks on a global queue
↳ if its out-of-date its ok... nothing bad will happen

add the task before switch and use a flag to mark as dangerous
↳ during switch → change the flag
↳less expensive than doing actual work during switch

If someone picks up task while dangerous it can just put it back
↳only a small waste

Problem:

```
void notKernelMain() {
    Barrier b{N+1};
    for (i=0; i<N; i++) {
        thread([ ] {
            print("before\n")
            b.sync()
            print("after\n")
        });
    }
    return ;
}
```

```
void KernelMain(){
    notKernelMain();
    while(true)
        yeild();
}
```

very high probability these threads WONT finish before notKernelMain() ends

⇓

This is a PROBLEM because b will get deleted. the usage of b

Need to capture b ⇐ outlives its existence

but CANT use value

⇓

makes copy of the barrier which is USELESS

⇓

Solution:

```
void notKernelMain() {
    Barrier b{N+1};
    for (i=0; i<N; i++) {
        thread([ ] {
            print("before\n")
            b.sync()
            print("after\n")
        });
    }
    b.sync()        → Makes sure all uses of b complete before ending notKernelMain()
    return ;
}
```

How do we make this Reliable?

```
void notKernelMain() {
    Barrier* b = new Barrier(N);
    for (i=0; i<N; i++) {
        thread([b] {
            print("before\n")
            b.sync()
            print("after\n")
        });
    }

    return ;
}
```

* Put B in the heap (malloc)

creates a copy of the REFERENCE of b

*NOTE: There is no reliable way to make sure b is deleted (Simple)
↳ would have to have a counter and free

can delete the extra sync

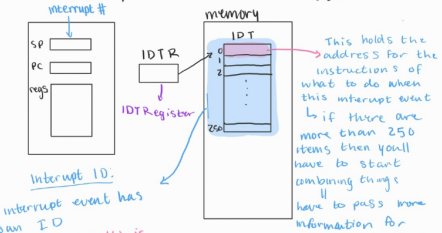
OR implement Garbage Collector once all threads are done

2/7

Ghetth Lied to us in 429

*Note: This will be discussing x86 Architecture

There are a few pointers that EVERY architecture has



This holds the address for the instructions of what to do when this interrupt event
↳ if there are more than 250 items then you'll have to start combining things
⇓
have to pass more information for the interrupt instructions to know who interrupted it

Interrupt ID:

each interrupt event has its own ID

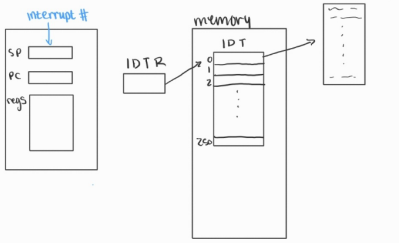
Note: in our project this is the IDT file (Interrupt Disruptor Table) (i think...)

* looks like a Forced Function Call

Before taking an interrupt:

- saves 3 values to the current stack
  - Current PC
  - Flags Registers (ex. carry bit) → Vulnerable: can be modified by anyone!!
  - CS register
    ↳ CS = Code Segment:
    Describes the nature of the code that is running (ex. x64 or x32 mode)
    ↳ The interrupt might be running in a different mode

- x86 uses a special instruction (iret) to return
  ↳ interrupt return
- interrupt handler will uses push a and pop a to save EVERYTHING to the stack
  push all / pop all



*When an interrupt is called the interrupt code is still running on the same hardware
↳ This can cause a problem because it will continue to re-read the interrupt and loop infinitely

⇓ solution

%EFLAGS → x86 register

IF

↳ interrupt flag: tells you if you should ignore interrupts or not
↳ this is what defines if you're interrupted or not

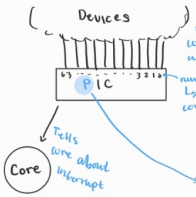
CLI → clear interrupt flag

STI → set interrupt flag

4/7

* There are maskable and non-maskable interrupts... ??

Interrupt Controller

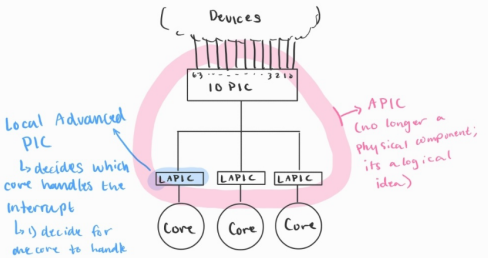


IC is connected to devices with wires

numbered wires
↳ these numbers typically correspond with the IDT
↳ used to be directly wired but now there's a table
⇓
IC is now PIC
↳ programamable IC
will typically also handle interrupt priority

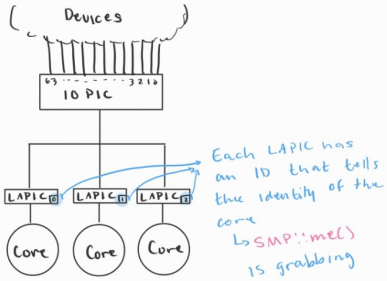
Tells core about interrupt

what if there are multiple cores

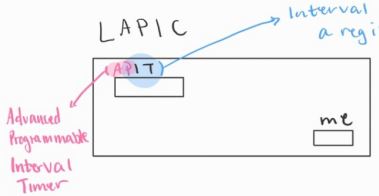


APIC (no longer a physical component; its a logical idea)

Local Advanced PIC
↳ decides which core handles the interrupt
↳ 1) decide for one core to handle
2) first one to receive handles it
3) They all handle it



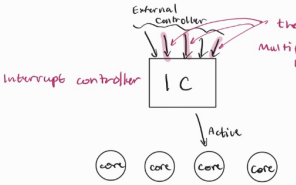
Each LAPIC has an ID that tells the identity of the core
↳ SMP::me() is grabbing this value



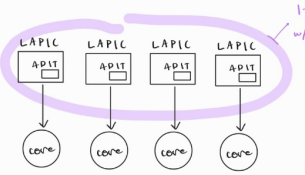
Advanced Programmable Interval Timer

Interval timer:
a register that hardware occasionally decrements
⇓
Generates Random arbitrary interrupts
⇓
Checks for a new interrupt from IOPIC
↳ can be used to force yeild()

Continued from last time...



there may be multiple active interrupts
⇓
only 1 can run
⇓
The controller knows which one has priority
↳ the other ones stay there pending



1-to-1 correspondence w/ cores

- when there's an interrupt there's ATLEAST one active core
- There's an IF flag (interrupt flag) in each core that tells the core there is an interrupt
- Interrupt won't be handled immediately but will happen "soon"

Handling Interrupts

① Turn off IF to prevent other interrupts from happening

② Execute the interrupts

③ iret is called -- returns and pops things back off the stack (including the original IF flag
↳ you could manipulate the stack and change the IF flag

* The LAPIC will have a small buffer that stores a list of pendin interrupts
* Once the interrupt is done it sets the Flip-Flop in the LAPIC to allow the next interrupt through

EOI goes to IC to tell it that the interrupt has been handled and it can stop trying to get it handled
↳ register in the device

```
{
  IF (core == 0) {        // only want to increment on one core so you dont count extra time
    jiffies += 1;
  }
  EOI
  STI();                  // DONT DO THIS!! re-enables IF which opens the possibility for multiple interrupts to be handled at one time
}
```

* Two Rules:
  ① Don't call sti()
  ② Must run in O(1)

Reminder:
CLI: disables interrupts
STI: enables interrupts

Note: you dont have to disable interrupts in prog → hardware will do for you

How to disable Interrupt:
- must be O(1)
- Spinlock must ALSO be O(1)
- A quantum boundary:
  yeild()
  ↳ In our world:
  No Matter What: if the timer reaches it's limit ⇒ Force yeild()
  ↳ Normally this would be optimized to not switch away from a task too quickly

Note: Test threads (threads created by test) should run w/ interrupts enabled

Problem: when you switch threads the thread may be assuming the state of of IF but it may not have been restored
↳ Need to store the des

```
APIT Handler:
{
  update jiffies
  EOI
  yeild()
}
```

Question: How do we ensure IF is enabled?
Solution: Push IF flag during context_switch

Problem: what if you interrupt in the middle of a yeild()?
- will look like calling yeild() recursively
- yeild() is calling and manipulating global variables

what about if you're in the middle of spinlock()?
- could result in deadlock

what about middle of malloc()?
- could mess up malloc

CANT call yeild() ⇒ yeild NOT O(1)
↳ manipulates ready Queue which has a spinlock which is NOT O(1)
↳ you don't know how long you will wait to obtain lock

↳ can you call yeild() while yeilding?
↳ use "busy flag" to check if the thread is yeilding
↳ Only Fixes the yeild problem

* one Idea: have a local ready queue ⇒ allows you * use ready queue in O(1)

NOTE: This is ONLY for the Timer Interrupt
↳ can ignore the timer interrupt b/c you're already yeilding

How does the thread know it's identity?
↳ TCB

```
auto my_tcb = current_thread[SMP::me()]
```
⇓ Same as
```
auto id = SMP::me();
[INTERRUPT]
auto my_tcb = current_thread[id];
```
BIG PROBLEM! SMP::me() changes!
⇓
RACE condition!!

EASY SOLUTION

```
cli();
auto id = SMP::me();
auto my_tcb = current_thread[id];
sti();
```
Disable Interrupts

New Problem!

```
// how do you know interrupts were enabled before?
cli();
auto id = SMP::me();
auto my_tcb = current_thread[id];
sti();
```
wouldn't want to enable them here!

Pretty much the only place where it's ok to disable interrupts → you dont want to disable anywhere else

* Put this in a method and call the method instead of accessing current_thread

Solution!

```
was = Interrupts::disable();   // disables and returns previous state
      .
      .
Interrupts::restore(was);
```
* This needs to be written

* Note: Gheith recommends putting asserts every where to check the IF flag!!

NEW PROBLEM!

O(1)
```
Spin.lock();
.
.
[interrupt]        // can lead to dead lock
.
Spin.unlock();
```

TWO BAD IMPLEMENTATIONS

```
Spin.lock();
disable();
                   // can still interrupt here
enable();
Spin.unlock();
```

```
disable();
Spin.lock();
Spin.unlock();
enable();
```
no longer O(1)

Solution
change spinlock implementation:

```
disable();
while(exchange(taken, 1)) {   // Disables interrupts for this line
  restore();
  // Stuff
  disable();
}
restore();
```

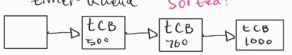
* Note: There may still be problems with this implementation
  Ex: what if a different thread unlocks?

* Note: Relax the disable = O(1) requirement

How To Sleep?

Gheith's first implementation

```
void sleep(int n) {
  target = jiffies per second(n) + jiffies;   // estimated test of time
  while (jiffies ≤ target) {                  // BAD! Blocks the whole core from running
  }
}
```



- check and update during every interrupt
- you technically could just have local timer queues cause they're all being updated at the same time
  ↳ doesn't matter because still can't add every thing to ready queue

* A thread that calls sleep will always have interrupts enabled
  ↳ sleep will always be run by a user process (a test case)

* Anything you do in the APIT needs to be O(1)
  ↳ APIT can't create threads ⇒ not O(1)
  ↳ Give APIT a premade thread to handle the cleanup work that is slower
  ↳ this special thread should prevent preemption
    ↳ can't stop the interrupt but can stop the interrupt from doing anything
      ↳ disabling interrupt handler stops time from advancing

this is ok bc we KNOW it will yeild

* This helper thread will add the things to the ready queue

* helper thread ⇒ offline interrupt handler
  ↳ will never go into ready queue
    ↳ manually call it!

* how do you communicate with helper thread?
  ↳ put some sort of data structure in the thread to send it info
    ↳ (ex. timer ⇒ let the thread figure out who can wake up)

* This is only needed if the next thread has interrupts disabled
  ↳ if interrupts are enabled in the next thread then it could just do the cleanup
  ↳ if interrupts not enabled then it can't
    ⇓
    you don't want interrupts to be disabled while cleaning up

Is this too much work?