15-410

"...What does IRET do, anyway?..."

Exam #1
Feb. 27, 2004

Dave Eckhardt

Bruce Maggs

Synchronization

Final Exam list posted

- You *must* notify us of conflicts in a timely fashion

P3 milestones (completed, right?)

- Read handout, re-read k-spec
- Chosen 3+ weekly joint hacking sessions
- Set up source control repository
- Rough-draft division of labor, rough pseudo-code/outlines
- Typed *some* code...?

Book report topic chosen? Great for airplane time...

Summer internship with SCS Facilities?

A Word on the Final Exam

Disclaimer

- Past performance is not a guarantee of future results

The course will change

- Up to now: “basics”
  - What you need for Project 3
- Coming: advanced topics
  - Design issues
  - Things you won't experience via implemention

Examination will change to match

- More design questions
- Some things you won't have implemented

Outline

Question 1

Question 2

Question 3

Question 4

Question 5

Past Misunderstandings

This is a *C* programming class!

- sizeof (char) == 1 /* 8 bits */
- sizeof (int) == 4 /* 32 bits, *mostly* true now */
- You need to *really* understand pointers

Semantics

- '\0' isn't "just" a 1-byte zero – it's the zero *char*
- Compare 0, '\0', NULL

Other languages are excellent

- ...but very few are ok for writing OS code

Q1 – Definitions (graded *gently*)

XCHG

- instruction, atomically, exchanges

Kernel Stack

- stack used by a thread while running kernel code
- “stack” != “memory”, “stack” != “control block”

Atomic Instruction Sequence

- Must not be interrupted/interleaved, should be short

Exception

- Control transfer to OS, caused by instruction stream

Yield()

Q2 – Interrupt Handling

Misconception City!

- `static` local variable
- What's that ol' `IRET` do, anyway?
- If an interrupt fires in the forest, and nobody hears it...
- “Assume an infinite stream of interrupts...”
- “printf() is a system call”
- Watch out for sneaky stack growth...

`static` local variable??

```
static int ticks_since_boot = 0;
```

What's that all about?

- A weird C trivia question, except...
  - Used in C++ and Java too!

What's the proper scope for `ticks_since_boot`?

- Used by only one procedure
  - Remember, don't specify data items in your interface!!!
  - Specify methods instead
- Used by only one procedure
  - Don't want it to be global
- But local variables “reset” each time procedure is called!
- Unless they are declared “static”!!!

Static = procedure-local persistent variable (oh, and ...)

What's that ol' `IRET` do, anyway?

IRET should not be mysterious

- You used it in P1, will use it *a lot* in P3
- Looking things up in intel-*.pdf is a *good idea*

On interrupt/exception, processor *follows a protocol*

- Saves some state ("trap frame"), typically on stack
- What's that "state" for?
  - Exception: explain what "caused" the exception
  - Interrupt & exception: document "where we were at the time"
  - Handler done? Ram it back into the relevant registers!
    - » `IRET`

So...

- `IRET` pops top of stack into `%EIP, %CS, %EFLAGS (...)`

Other issues with the bad code

IRET happens before function clean-up

- **...leaks “caller's %ebp” each time**
- **True, but we never run that many times**

Registers might be corrupted before PUSHA

- ***Could* happen...**
- **...but not as a result of a *static* local declaration/initialization**

If an interrupt fires in the forest...

What do we mean by a “disabled” interrupt?

- Alternate term: “masked”

Why do we “disable interrupts”?

- To protect an atomic instruction sequence...
- ...which should be “short”...
- ...so it's ok for interfering sequences to...
  - ...die?

If an interrupt fires in the forest...

Why do we “disable interrupts”?

- To protect an atomic instruction sequence...
- ...which should be “short”...
- ...so it's ok for interfering sequences to...
  - ...wait a bit before they can run!

What do we mean by a “disabled” interrupt?

- Alternate term: “deferred”!
- The interrupt controller will remember it until we re-enable

Why should interrupt handlers be “short”?

- Not: longer ones are more likely to throw away interrupts!
  - No length would be safe!
- Because some hardware will get angry if we don't answer...
  - ...or maybe some user code will.

If an interrupt fires in the forest...

Impatient Ethernet

- Interrupts when each packet arrives
- When “ring buffer” overflows, packets will be lost
  - Process them *soon*...

Impatient Disk

- Interrupts when sector is ready
- Say “Oh, and give me the next sector too” *soon*...
  - Or it will have rotated past the head.

Impatient Timer?

- Reloads and starts counting before you process interrupt
- Inter-interrupt period is, well, 10 milliseconds
- (1 *billion* / 1 hundred) instructions...
- That is a deadline, but it's not really a harsh one.

“Assume an infinite stream of interrupts...”

Each interrupt handler invocation uses stack space

- True

“If we have an infinite stream of interrupts...overflow!”

- True
- True of *any* interrupt handler code
  - .c, .S, asm(), ...

Can this happen?

- Each device issues one interrupt, waits for dismissal
  - `outb(...)` in 15-410 x86 support code
- Finite number of devices on system
- How many trap frames can be on stack?

“printf() is a system call”

Reasoning

- `printf()` is a system call
- System calls are slow
- Interrupt handlers should not be slow

`printf()` isn't magic...

- `printf()` is a *library routine*
- ...which sometimes invokes a system call...
- ...if it's not already in the kernel!

kernel `printf()` is a library routine...

- ...which calls the console driver!
- It may or may not be “slow”... (scrolling screen isn't zippy)
- ...but it's not *impossibly* slow.

Sneaky Stack Growth

People generally understand

- Function call sequence begins with pushing parameters
- Then there is a `call` instruction
- What happens *after* the `call`?

Several people claimed

- When `timer_handler()` calls `printf()` and then `outb()`...
- ...“all of those parameters are still on the stack at `POPA`”

Q3: Stack Trace

Many people got this essentially right

Common "oops"

- Assuming `mystery(s1, s2)` because it "seems natural"
- Function table shows `mystery(s2, s1)`

Trouble?

- Review P0 code
- During P3 you may well need to debug from a hex dump

Q3: Stack Trace

```
void main()
{
  printf("Fred!\n");
  exit(99);
}
```

Q3: Stack Trace

```
LC0:
   .ascii “Fred!\12\0”
_main:
   pushl %ebp
   movl %esp,%ebp
   pushl $LC0
   call _printf
   addl $4,%esp ⇐ What's that?
   pushl $99
   call _exit
   addl $4,%esp ⇐ There it is again!!!
   leave
   ret
```

Q4: Deadlock

Many people got this *mostly* right

Key idea

- Four requirements for deadlock
- Four ways to prevent it ("Four Ways to Forgiveness")
- One of them is *commonly* used (locking order)
  - Now you intuitively understand that

Subtle idea

- You can lock *anything* as long as *each lock()* is "in order"
- lock(0); lock(1); lock(33); unlock(33); lock(2);
- Which tool should go at the end?

Q5: Concurrency

“Race condition” / “Thread-safe” still not clear

- Neither one is thread-safe (on either exam)!

Myths

- A: “As long as shared state is changed inside a mutex I'm ok”
- B: “Once cond_wait() returns I'm good to go”
- “Since neither foo() writes to shared state everything is ok”

Myth “A”

"If shared state is changed inside a mutex I'm ok"

- **Not if the decision about *how to change* is outside!**

```
if (queue->start == (queue->end + 1) % QUEUE_LEN)
  return -1;
/* now we mutate NO MATTER WHAT */
mutex_lock(&queue->lock);
queue->buf[queue->end] = data;
queue->end = (queue->end + 1) % QUEUE_LEN;
mutex_unlock(&queue->lock);
```

Myth “B”

"Once cond_wait() returns I'm good to go"

- **You're running with the lock, but are you running *first*?**

```
mutex_lock(&stack->lock);
/* If the stack is empty, wait for data *
if (stack->spot == -1)
  cond_wait(&stack->empty, &stack->lock);
data = stack->buf[stack->spot]; /* It can be -1 again! */
stack->spot--;
mutex_unlock(stack->lock);
```

Shared Myth

"Since neither foo() writes to shared state foo() is ok"

- **What about main()-vs-foo() conflicts?**

```
tid[0] = thr_create(foo, 0); /* foo(0) reads tid[0..1] */
tid[1] = thr_create(foo, 1); /* foo(1) reads tid[0..1] */
```

main() writes tid[0..1], foo() reads tid[0..1]

- **Nary a mutex in sight...**
- **Does `foo(0)` run before or after "`tid[0] =`"?**

Summary

```
90% = 67.5   15 students
80% = 60.0   17 students
70% = 52.5    7 students
<70%          6 students
```