15-410

"..."I'll be reasonable as soon as I get everything I want"..."

Exam #1
Mar. 6, 2006

Dave Eckhardt

Synchronization

Checkpoint 2 – Wednesday, in cluster

- Reminder: context switch ≠ interrupt
  - Later other things will invoke it too

Upcoming events

- 15-412 (Fall)
  - If you want more time in the kernel after 410...
  - If you want to see what other kernels are like, from the inside
- 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 (text useful)
- Still 3 hours, but more stuff

Outline

Question 1

Question 2

Question 3

Question 4

Question 5

Q1 – Short Answer

"By the book"

- M:N threads
- Starvation

"Write an x86 instruction which accesses text, rodata, bss"

- Key insight: *every* instruction "accesses" text!
- Need a way to move something from one region to another
  - If a program has *no* data, then rodata and bss can be on adjacent pages... hmm...
  - PUSHL 8(%EBP) – if your %ESP points into bss
  - MOVSL – magic string-move instruction

Q2 – The Mysterious argv[]

Question asks for "argv[] including the strings"

- Showing just the strings wasn't enough

Question asks for addresses

- Many solutions without addresses were excessively abstract

main() has *two* parameters – argc, argv

- argv[] is an array of pointers to strings
- argv[] is not a bunch of parameters to main()
  - stack frames showing lots of pointer parameters to main() were wrong

Q2 – The Mysterious argv[]

Smaller issues

- strings in the heap
- argv[] in the heap
- huge/weird padding between strings

Suggestions

- Some mistakes should not have been possible given P0
- If you can't draw a picture of something, you might not understand it
  - Groups had this problem in P2
  - Skipping pictures is a way to hurt yourself in P3

Q3 – IRET

Motivations for IRET

- Privilege-change case
  - "Useful" to atomically switch privilege level and program counter, since user typically can't run kernel code
  - Not the P1 (exam problem) case
- x86 multi-segment fun
  - If you're returning from one code segment to another, "useful" to atomically switch code segment and program counter, since each is meaningless without the other
  - Not the P1 (exam problem) case
    - All %CS values are the same – no need to restore %CS
  - Not necessarily related to interrupts (there is a "far return" instruction which undoes a "far call")

Q3 – IRET

Our hopes for the question

- What needs to be done?
- What does _____ mean?
- What would happen if _____?

Concern is one step

- Some concerns can be addressed

Q3 – IRET

Concerns and responses

- An interrupt might happen during the “IRET sequence”
  - A new trap frame would be pushed on the stack
  - The %EIP in that frame would be in the middle of your “IRET sequence”
  - As long as your %ESP is “reasonable”, this is all ok
- “The trap frame parts are in the wrong order”
  - You can move them around
    - » Push two registers onto the stack
    - » Yank, XCHG, push
    - » Restore, restore

Q3 – IRET

A common approach

```
pop %eip
pop %cs
pop %eflags
```

Q3 – IRET

A common approach

```
pop %eip    # typically called “RET”
pop %cs     # will this be executed?
pop %eflags # will this be executed?
```

A common misconception

- Return from interrupt requires more info than trap frame
- CPU stores the “other information” somewhere else
- The “trap, then return” story needs to work
  - If you can take a trap while servicing a trap, you need to stack the “other information” somewhere... why not use the stack?

Q4 – setjmp()/longjmp()

Concepts

- What does it mean to be executing a function?
  - What is the state of the function?

The todo list

- %eip – of setjmp()'s *caller*
- %esp, %ebp – or everything the caller does will be wrong
- callee-save registers
  - setjmp()'s caller is allowed to depend on “setjmp()” preserving them
- %eax – longjmp()'s param must become setjmp()'s return value

Q4 – setjmp()/longjmp()

Common issues

- Save/restore only %eip
  - Needs to be setjmp()'s *caller's* %eip, which is up on the stack
  - Restoring *only* %eip guarantees broken run-time environment
- Save/restore only %esp – also severely broken
- Missing some registers
- Not quite getting back

P3 suggestions

- Written todo list, often derived from pictures

Q5 – Deadlock / Race

```
void process_ba(int *workload)
{
  for(int i=0; i< 100; i++) {
    mutex_lock(&mutex_b);          /*BA1*/
    mutex_lock(&mutex_a);          /*BA2*/
    (*workload)++;                 /*BA3*/
    mutex_unlock(&mutex_a);        /*BA4*/
    mutex_unlock(&mutex_b);        /*BA5*/
  }
}
```

Q5 – Deadlock / Race

Key concepts

- Deadlock ingredients
- Atomicity versus race conditions

How “try-lock” doesn't save you

Determinism

Try-Lock – Your Hope

AB	*BA*
`mutex_lock(&a);`	
	`mutex_lock(&b);`
`mutex_lock(&b);`	
	`mutex_is_unlocked(&a)`
	`mutex_unlock(&b);`

Try-Lock – Your Bad Luck

AB	*BA*
	`mutex_lock(&b);`
	`mutex_is_unlocked(&a)`
`mutex_lock(&a);`	
	`mutex_lock(&a);`
`mutex_lock(&b);`	

Q5 – Deadlock / Race

What went wrong?

- `process_ba()` checked something and committed to action
- But `process_ab()` was still running
- So the “checked-for” condition wasn't true any more

Q5 – Deadlock / Race

Atomicity

- "Is it safe to assume that line BA4 executes atomically even though it may be composed of multiple instructions?"
  - Certainly not in general!
  - But sometimes we *must* have atomicity (with respect to interfering sequences)
  - When we must have atomicity, we use a synchronization primitive, such as mutex_lock()
  - Then we may assume atomicity—because we built it

Q5 – Deadlock / Race

Determinism

- What is the final value of `workload`?
- Using original `process_ba()`
  - *If* we didn't deadlock, there must have been 200 increments
    - » Protected by `mutex_b` (and `mutex_a`!)
    - » That looks deterministic...
- Using replacement `process_ba()`
  - If we didn't deadlock, there have been 100..200 increments
    - » That's not so deterministic

Q5 – Deadlock / Race

Determinism

- What is the final value of `workload`?
- Using original `process_ba()`
  - *If* we didn't deadlock, there must have been 200 increments
    - » Protected by `mutex_b` (and `mutex_a`!)
    - » That looks deterministic...
- Using replacement `process_ba()`
  - If we didn't deadlock, there have been 100..200 increments
    - » That's not so deterministic
- Say, what was the *initial* value of workload?
  - "int workload"

Summary

90% = 67.5 8 students

80% = 60.0 16 students

70% = 52.5 12 students

60% = 45.0 10 students

<60% 4 students

Comparison

- **This is a typical mix for the mid-term exam**

Implications

Score below 52?

- Figure out what happened
- Probably plan to do better on the final exam

Score below 45?

- Something went *very* wrong
- Passing the final exam may be a serious challenge
- To pass the class you must demonstrate some proficiency on exams (project grades alone are not sufficient)