15-410

"My other car is a cdr" -- Unknown

Exam #1
Mar. 5, 2012

Dave Eckhardt

Synchronization

Checkpoint 2 - alerts

- Please read the handout warnings about context switch and mode switch and IRET *very carefully*
  - Each warning is there because of a big mistake which was very painful for previous students

Asking for trouble

- If your code isn't in your 410 AFS space every day, you are asking for trouble
- If your code isn't built and tested on Andrew Linux every two or three days, you are asking for trouble
- If you aren't using source control, that is probably a mistake

Upcoming Events

Google "Summer of Code"

- http://code.google.com/soc/
- Hack on an open-source project
  - And get paid (possibly get recruited, probably not a lot)
- Projects with CMU connections: Plan 9, OpenAFS (see me)

CMU SCS "Coding in the Summer"?

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

Synchronization

Crash box

- How many people have had to wait in line to run code on the crash box?
  - How long?

Synchronization

Debugging advice

- Once as I was buying lunch I received a fortune

Synchronization

Debugging advice

- Once as I was buying lunch I received a fortune

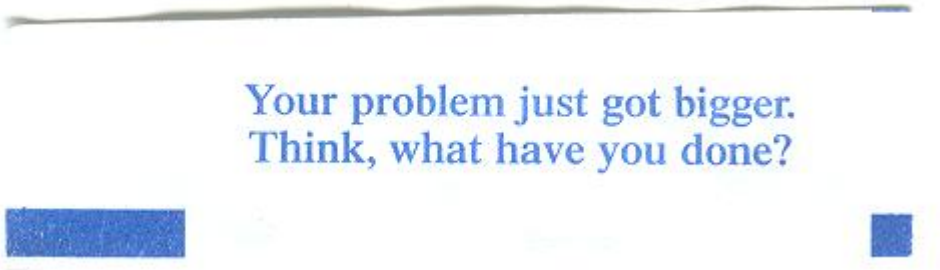

Image credit: Kartik Subramanian

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 implementation

Examination will change to match

- More design questions
- Some things you won't have implemented (text useful!!)
- Still 3 hours, but more stuff (~100 points, ~7 questions)

“See Course Staff”

If your paper says “see course staff”...

- **...you should!**

This generally indicates a serious misconception...

- **...which we fear will seriously harm code you are writing now...**
- **...which we believe requires personal counseling, not just a brief note, to clear up.**

Outline

Question 1

Question 2

Question 3

Question 4

Question 5

Q1a – "Why is text read-only?"

Expected

- Detect accidental writes (instead of corrupting execution)
- Make it easier to share program text across programs
  - This *can* be done with copy-on-write
  - But it can also be done in non-COW kernels
    - » It was done in non-COW kernels for many years
    - » Occasionally somebody still writes a non-COW kernel
  - Read-only text enables sharing between "unrelated" programs
    - » User A's /bin/bash and User B's (hard for COW)

Admissible, depending on details

- Relationship between executability and writability
  - But note: if an r/x segment "happens" to overlap with an r/w segment, then segmentation isn't protecting text
- ROM code is *really* r/o, so if a program will ever run out of ROM it can't plan on writing to its text

Q1a – "Why is text read-only?"

What you should *not* tell us

- Prevents one program from overwriting another program's text
  - First, *independent address spaces* are how kernels isolate programs from each other
  - Second, protecting text but not data/stack wouldn't be a lot of protection
- Prevents network attacker from changing code
  - Not really preventive: attacker can put code somewhere else
    - » See "return-oriented programming"

Q1b – "I would like to assume..."

Basic idea: cost-benefit analysis

- What might you *gain* by assuming X?
  - Is it really a noticeable gain?
- What might you *lose* by assuming X?
  - If !X is wildly unlikely and easy to detect, then maybe the loss is "once in a long while I need to apologize and nobody will be mad"
  - If !X is plausible and would lead to disaster, then assuming X will plausibly lead to disaster

As system designers:

- You will need to "bake assumptions into your design"
- You should give real thought to which assumptions to "bake in"
- This pattern represents the most-basic "real thought"

Q2 – Memory arbitration

The key insight

- Exam sample code starves
- Because the problem is small (few players), it's easy to solve starvation with a little state

Common issues

- `get_ticks()`
  - It's a system call, but we're *below* that level
  - This “clock” needs to tick *much* more often than that!
  - (It's easy to maintain a good timestamp yourself.)
- `genrand()`
  - Hardware entropy exists, but not in infinite supply!
  - Genuine randomness is overkill... ECE's avoid overkill
- “mistakes”
  - Solution starves... solution doesn't progress...

Q3 – Trouble in the barbershop

Rueful warning

- If you were unable to find a problem, this is a serious issue
  - We intended one bug... class found three...
  - (It's really hard to insert *just one* concurrency bug)

Serious issues to avoid

- Misunderstanding how mutexes and cvars work (!!)
  - `cond_wait()` drops and reacquires the mutex! This is a fundamental part of what it does, and this absolutely must be understood.
- "Sometimes a customer misses a seat that is just opening up"
  - True, but the universe works that way ("It's a feature, not a bug")
- Solutions that exhibit "Paradise Lost"
  - You should *automatically* check for this

Q3 – Trouble in the barbershop

Somewhat serious

- Impossible/unclear execution trace
  - You need to be able to reason about these issues and communicate them to others.
  - Our exact format is not 100% necessary, but you need something at least that descriptive and clear.

Other notable issues

- Fix adds starvation

Q4 – "super semaphores"

Question goal

- "Write a synchronization object" - typical exam question

The lurking threats

- Deadlock – *easy* if `sem_wait()` does hold&wait
- Other progress failures
  - Core pattern: enough resources are free that thread at "head of queue" could be running, but it isn't

Design dangers

- "Paradise lost" - again, form the habit of checking for this
- One signal+wait per resource acquired: many ways to lose signals/have the "wrong thread" proceed
- Peering inside a cvar / adding `cond_xxx()` to cvar interface
  - *Many* things in this space don't work or make multi-processor-friendly implementations harder

Q4 – “super semaphores”

Distasteful

- “Just wake everybody up!”
  - It's painfully wasteful to wake up many threads if only one can make progress...
  - It's *especially* painfully wasteful to wake up many threads if zero can make progress right now!
- This doesn't mean “*cond_broadcast()* is always wrong”
  - But you should be able to say why it's right to wake up some group of threads

Q5 – Omitting the frame pointer

Why omit the frame pointer?

- It occupies a whole register that could be used for other things
  - x86-32 is unusually lacking in registers
  - Also, since the x86-32 calling convention was written, compilers are *vastly smarter* about register allocation
- Meanwhile, nobody uses it!
  - Individual functions don't need %ebp to correctly unwind the stack before returning (Part A of the question)
  - People don't routinely call traceback() or things like it
- So it makes sense to remove the costs from day-to-day operation and impose costs (even if higher) on debuggers and similar code

Q5 – Omitting the frame pointer

What do we need to accomplish?

- Given the address of a return address, find "everything"
  - Find the address of the next return address
    - » We already know how to find parameters relative to a return address

What does the "new code style" (Part A) provide?

- During the execution of a function, `%esp` is always X bytes below the return address
  - If we knew, for each function, that height...

Observations about real systems

- "Stack height" may not be exactly constant
  - Table may need to map from program-counter value to stack height
- Debuggers need to know register occupancy too
  - Also a function of program-counter

Q5 – Omitting the frame pointer

Conceptual hazards

- Confusions between *function* (a piece of code with static properties" and *function invocation* (one function may be invoked many times)!
  - "Store each function's `%ebp` in the table"
    - » *Impossible* given recursion
  - "Store each function's caller in the table"
    - » Functions don't *have* unique callers! Consider `printf()`!

“Design” in this exam

Reminder...

- Final exam will focus more on “design”
  - On this exam, design was best represented by super-semaphores and omit-frame-pointer questions ...

Breakdown

```
90% = 67.5   19 students (66 and up)
80% = 60.0   22 students
70% = 52.5   16 students (52 and up)
60% = 45.0    4 students (44 and up)
50% = 37.5    8 students (37 and up)
<50%          7 students
```

Comparison/calibration

- There were more high scores than is typical
- There were more worrisome scores than is typical

Implications

Score under 51?

- Form a theory of "what happened"
  - Not enough textbook time?
  - Not enough reading of partner's code?
  - Lecture examples "read" but not grasped?
  - Sample exams "scanned" but not solved?
- Probably plan to do better on the final exam

Score at/below 35?

- Something went *dangerously* wrong
  - It's important to figure out what!
- Passing the final exam may be a *serious* challenge
- To pass the class you must demonstrate proficiency on exams (not just project grades)
- "See instructor" is probably a good idea

Implications

"Special anti-course-passing syndrome":

- You got only the "mercy points" on several questions
- Extreme case: *no* question was convincingly answered
  - It is very important that you don't have *two* exams without evidence that *some* topics have been mastered!
    - » So if this exam looks that way, you should definitely at least "see course staff" to reduce the likelihood that both do!

This is not a real slide

This slide and the ones which follow are scratch slides from/for other semesters

Breakdown

```
90% = 67.5    8 students (66 and up)
80% = 60.0   15 students (59 and up)
70% = 52.5    9 students (51 and up)
60% = 45.0    5 students
50% = 37.5    3 students
<50%          3 students
```

Comparison/calibration

- People took longer than usual on the exam
- Grades aren't unusually low

Breakdown

```
90% = 67.5    3 students
80% = 60.0   16 students
70% = 52.5   23 students (52 and up)
60% = 45.0   10 students
50% = 37.5    0 students
<50%          0 students
```

Comparison

- Noticeably fewer “A's” than typical
- Also noticeably fewer “R's”

Breakdown

```
90% = 63.0   16 students (3 got 69/70)
80% = 56.0   26 students
70% = 49.0   20 students
60% = 42.0    9 students
50% = 35.0    4 students
<50%          2 students
```

Comparison

- Scores were “reasonably shaped”
- Probably a few more A's than typical

Implications

Score below 70%?

- Something went really wrong!
- You are strongly advised to debug the situation
- To pass the class you must demonstrate reasonable proficiency on exams (project grades alone are not sufficient)
- See syllabus

Above 70%?

- Probably a 50/50 chance that final-exam score will be one grade lower...

Summary

90% = 72.0 7 students

80% = 64.0 23 students

70% = 56.0 14 students

60% = 48.0 6 students

<60% 2 students

Comparison

- This is a roughly-typical mix for the mid-term
- More B's, fewer A's & C's

Summary

90% = 67.5 10 students

80% = 60.0 18 students

70% = 52.5 17 students (52 and up)

60% = 45.0 6 students

<60% 1 student

Comparison

- This is a roughly-typical mix for the mid-term
- More C's, fewer D's, fewer R's

Implications

Score under 55?

- Form a theory of "what happened"
  - Not enough textbook time?
  - Not enough reading of partner's code?
  - Lecture examples "read" but not grasped?
  - Sample exams "scanned" but not solved?
- Probably plan to do better on the final exam

Score below 42?

- Something went rather wrong
  - It's important to figure out what!
- Passing the final exam may be a serious challenge
- To pass the class you must demonstrate some proficiency on exams (not just project grades)

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)

Synchronization

Checkpoint 3 – Friday, file drop (see announcement)

- Suggestions
  - You now know how long VM and context switch take
    - » Plus fork() or exec()
  - There's a lot more to do
    - » Code, but also design (vanish()/wait()!) and debug
  - We'll ask you to put together a schedule... please do.
- Reminders
  - context switch ≠ mode switch
    - » Identify scenarios with one and not the other
  - context switch ≠ interrupt
    - » Later it will be invoked in other circumstances
  - If you don't see the differences, contact course staff!

Synchronization

Checkpoint 2 – Wednesday, in cluster

- Reminder: context switch ü 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?

Synchronization

Google "Summer of Code"

- http://code.google.com/soc/
- Hack on an open-source project
  - And get paid
  - And quite possibly get recruited

CMU SCS "Coding in the Summer"

Synchronization

Computer Club movie night

- "The Net"
  - "Her driver's license. Her credit cards. Her bank accounts. Her identity. DELETED."
- Tuesday 17:30, Wean 7500

However....

Synchronization

Checkpoint schedule

- Wednesday during class time
- Meet in Wean 5207
  - If your group number *ends* with
    - » 0-2 try to arrive 5 minutes early
    - » 3-5 arrive at 10:42:30
    - » 6-9 arrive at 10:59:27
- Preparation
  - Your kernel should be in mygroup/p3ck1
  - It should load one program, enter user space, gettid()
    - » Ideally lprintf() the result of gettid()
  - We will ask you to load & run a test program we will name
  - Explain which parts are “real”, which are “demo quality”