15-410

"My other car is a cdr" -- Unknown

Exam #1
Feb. 29, 2016

Dave Eckhardt

Todd Mowry

Synchronization

Checkpoint schedule

- Wednesday during class time
- Meet in GHC 3000 *(not 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”

Synchronization

Asking for trouble?

- If your code isn't in your 410 AFS space every day, you are asking for trouble
  - Roughly 2/3 of groups have blank REPOSITORY directories...
- 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
- GitHub sometimes goes down!
  - S'13: on P4 hand-in day (really!)

Synchronization

Google "Summer of Code"

- http://code.google.com/soc/
- Hack on an open-source project
  - And get paid
  - And quite possibly get recruited
- Projects with CMU connections: Plan 9, OpenAFS (see me)

CMU SCS "Coding in the Summer"

Synchronization

Book report!

- **Hey, “Mid-Semester Break” is just around the corner!**

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

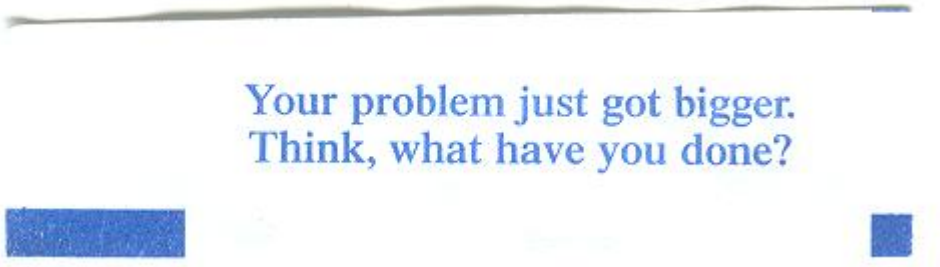


Image credit: Kartik Subramanian

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 could be more stuff (~100 points, ~7 questions)

“See Course Staff”

If your exam 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 – runnable/blocked threads

What we were testing

- Definitions of runnable, blocked
- Understanding of typical kernel entry/exit paths
- [What "kernel entry" means]

Conceptual mayhem ensued

- A trap is *not* an interrupt
- "Leave the kernel" does not mean "become descheduled" (or "become scheduled"); "enter the kernel" likewise
- Entering the kernel does *not* change a thread from running to runnable – it keeps running!
- Scheduling a thread does not cause it to leave the kernel – it keeps running in the kernel, at least for a while, maybe for a *long* while – memmove()
- "Runnable" means *not running* – getpid() can't make that happen

Q1a – runnable/blocked threads

Specific alarming items

- "A runnable thread enters the kernel because it invokes some system call"
  - Fundamentally, the running-to-runnable transition is about a *surprise loss of the CPU the thread was running on*
- "An I/O interrupt blocks a thread"
  - Typically, an I/O interrupt *unblocks* a thread

Concepts to be very clear on

- Entering the kernel (trap/exception/interrupt)
- Leaving the kernel (after trap/exception/interrupt)
- Definition of running/runnable/blocked, also transitions: user/kernel, running/runnable/blocked
  - Very soon you will be implementing these, so it is important that you have a crisp sense of what they mean!

Please heed context-switch warnings in handout!!!

Q1b – “Thread cancellation”

Many high scores on this part

- Good!

A few common issues

- Voluntary exiting isn't what thread cancellation is about
- The kernel suddenly deciding to slay a thread isn't what thread cancellation is about
  - BTW kernels should *not* capriciously slay threads!
  - It might feel that way as a novice programmer, but as an OS expert you should form an organized understanding of causality

Q2 – Critical-Section Problem

What we were testing

- Ability to find a bounded-waiting problem
- Ability to write a clear execution trace
- Ability to solve a bounded-waiting problem

Odd feature of the problem

- This code was discussed in class!

Many scores were high

- Good!

Q2 – Critical-Section Problem

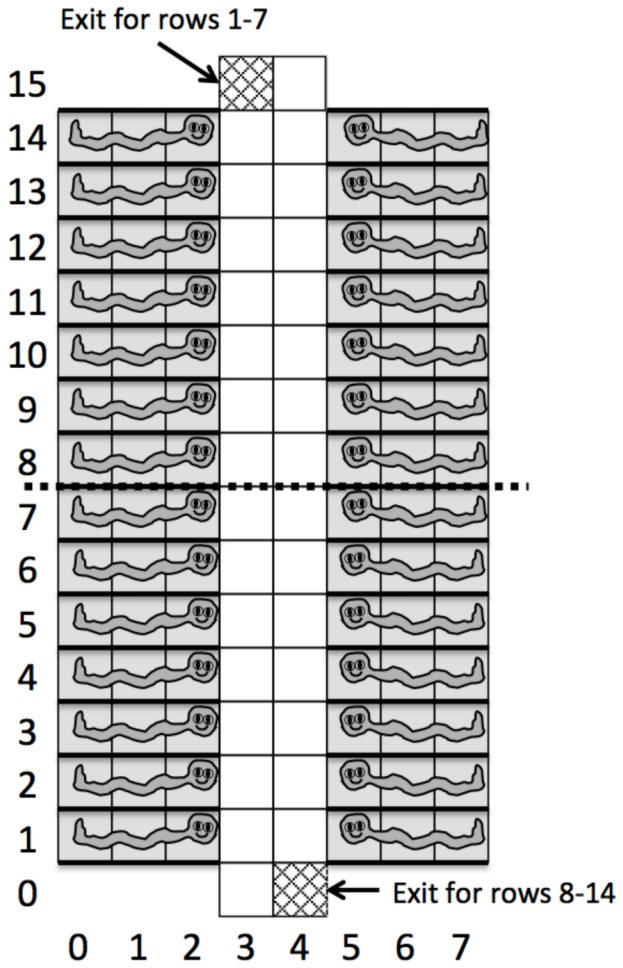
Some disturbing features were observed

- Many traces were not easy to read
  - It is to your benefit to be good about thinking scenarios through, and notation matters
  - Plus, you still have a final exam to take...
- A few people misinterpreted the code (that can happen)
- Roughly 10% of suggestions for fixing the problem made it worse
  - Spin-waiting
  - Deadlock

If you had trouble with this question...

- ...please figure out why and how to practice. This is core material.

Q3 – Deadlock


Exit for rows 1-7
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Exit for rows 8-14
0 1 2 3 4 5 6 7

Q3 – Deadlock

Parts of the problem

- Explain how deadlock can happen
  - 4 necessary conditions
- Deadlock avoidance
- Deadlock prevention

Q3 – Deadlock

Explain how deadlock can happen

- 4 necessary conditions
- Most people did well on this
- Common mistakes:
  - Poor explanations of:
    - » Cyclic waiting
    - » No preemption

Q3 – Deadlock

Deadlock *avoidance*

- Many people struggled with at least some parts of this
- First sub-part: resource pre-declarations for R13, R7
  - Full path to exit
  - Most people got this

Q3 – Deadlock

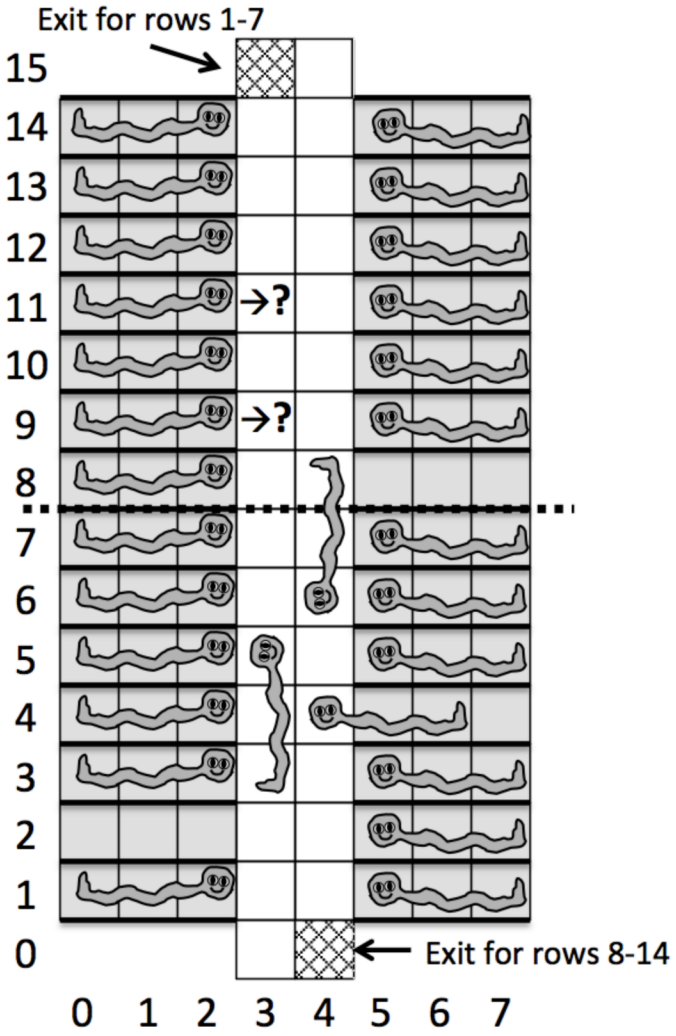
Deadlock *avoidance*

- **Second sub-part: specific resource scenarios**
- **(a) show safe sequence?**
  - **Proceed *sequentially***

Q3 – Deadlock

Deadlock *avoidance*

- **Second sub-part: specific resource scenarios**
- **(b) & (c): safe requests?**
  - **Common mistake:**
    - » **Ignoring safe sequence**
      - Carefully moving snakes
      - Not *sequential*
    - » **Confusing:**
      - Safe seq &
      - Necessary deadlock
  - **Many people got L9**
  - **Most people missed L11**



Q3 – Deadlock

Deadlock *prevention*

- Well-reasoned discussion of 4 requirements is important
- Arguing that it is not possible:
  - Ok if reasoning is sound
- Breaking the computation up into phases (e.g., top then bottom):
  - Ok if clear that implementation is practical and does not deadlock, etc.
- Many people did poorly on this one
  - Confusing prevention with avoidance

Q4 – "CountDown latches"

Question goal

- Slight modification of typical "write a synchronization object" exam question

General conceptual problems

- "x() takes a pointer" does *not* mean "x() must call malloc()"
- Assigning to a function parameter changes the *local copy*
  - It has no effect on the calling function's value
  - C isn't C++ or Pascal (luckily!)
- Everything must be initialized and destroyed
- See course staff about any general conceptual problems revealed by this specific exam

Q4 – "CountDown latches"

"Be careful out there"

- Deadlock scenarios
- Memory leaks
- Busy-wait/spin-loop – use an accepted synch object!
- Waking up threads when it really doesn't make sense
  - Use cond_broadcast() rarely – one "ok case" is when the number of threads to awaken is genuinely uncertain

Question-specific conceptual problems

- If a data structure is full of threads, it can't be destroyed without some kind of synchronization
  - Clearly stated in the problem text: abort()/destroy() problem
  - Also available: countdown()/destroy()
- Count must peg instead of going negative or threads can get stuck

Q5 – swexn() return protocol

Question goals

- Test understanding of thread execution state

Question

- Why don't swexn() handlers just return (the way Unix signal handlers do)?
- Note implicit assumption: we frequently want to re-execute the troubled instruction
  - This is true! Not just for page faults, not just in user mode

Q5 – swexn() return protocol

The key problem with "return"

- Execution state consists of more than just %EIP!
  - Instructions depend on (potentially) every bit of every register as inputs

Outcomes

- Many answers discussed why/how to *not* re-run the troubled instruction
  - Warning: much of what an OS does is supposed to be invisible!

Warning

- Some answers discussed running swexn() handlers in kernel mode
  - This is a serious conceptual misunderstanding!

Breakdown

```
90% = 58.5    4 students (57.0 and up)
80% = 52.0    8 students (51.5 and up)
70% = 45.5   12 students
60% = 39.0   26 students
50% = 32.5   12 students
40% = 26.0    5 students (25.0 and up)
<40%          0 students
```

Comparison/calibration

- These scores are low – maybe 10% too low?
- Some adjustment is possible after detailed analysis

Implications

Score below 39?

- 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?
- It is important to do better on the final exam
  - Historically, an explicit plan works a lot better than “I'll try harder”
  - Strong suggestion: draft plan, see instructor

Implications

Score below 32?

- Something went *dangerously* wrong
  - It's important to figure out what!
- Beware of "triple whammy"
  - Low score on deadlock *and* CDL *and* critical-section
    - » Those questions are the "core material"
    - » Strong scores on Q1+Q5 don't make up for serious trouble with core material
    - » This was a comparatively easy critical-section question
- Passing the final exam may be a *serious* challenge
- *Passing the class may not be possible!*
  - To pass the class you must demonstrate proficiency on exams (not just project grades)
- See instructor

Implications

"Special anti-course-passing syndrome":

- **Only "mercy points" received on several questions**
- **Extreme case: *no* question was convincingly answered**
  - **It is *not possible to pass the class* if both exams show no evidence that the core topics were mastered!**