15-410

"My other car is a cdr" -- Unknown

Exam #1
Mar. 24, 2008

Dave Eckhardt

Roger Dannenberg

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

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

Debugging advice

- **Last semester as I was buying lunch I received a fortune**

Synchronization

Debugging advice

- Last semester as I was buying lunch I received a fortune

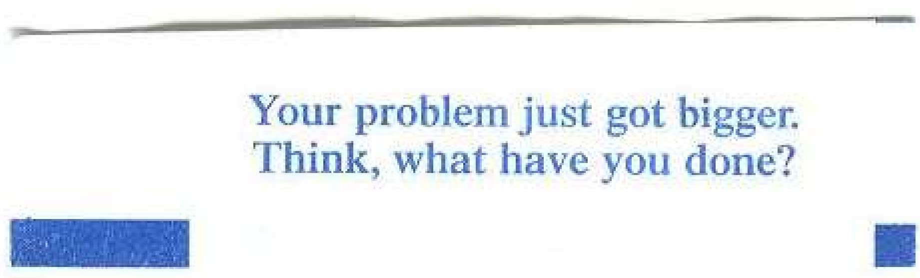

Image credit: Kartik Subramanian

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)

Outline

Question 1

Question 2

Question 3

Question 4

Question 5

Q1 –Short Answer

Progress

- Pretty straightforward
- For critical-section algorithms: “As long as a non-zero number of people want to enter the critical section, somebody will get to enter”.
- (As a general “systems” term: “the system is performing useful work”)

Q1 – Short Answer

User Mode

- **A common glitch – using "mode" without explanation when defining what a user mode is**
- **Key concept: environment in which operations which could disturb other computations are banned; enforced by rules built into hardware (`OUTB` checks `IOPL`; `CLI` won't let user code disable interrupts, etc.)**

Q2 –Trouble at the Warehouse

What was right about the code?

- Lots of mutexes, lots of cvars
- All code accessing shared state held *some* mutex

What was *not* wrong about the code?

- There was not an arbitrary underflow/overflow problem
  - Reasoning is weird but a useful thought exercise
    - » Because adders and subtracters use different loading docks, there can be at most one of each
    - » Inside of that restriction, the one adder and the one subtracter do lock each other out
- "Always use cond_signal(), not cond_broadcast()"
  - Waking too many threads can be an issue
  - But waking too few people risks waking the wrong kind

Q2 – Trouble at the Warehouse

What was wrong?

- One logic error (involving "ready")
- A huge synchronization error
  - Wrong number of mutexes
  - Mutexes doing the wrong job
  - The key issue
    - » Everybody involved in shared state has a "examine, then commit" pattern (aside from trivial cases: ++/--)
    - » If state can change between "examine" and "commit", people will get lost/hung, or state changes will be incorrect
    - » Solution: *one* mutex per *collection* of shared state
    - » Held just long enough for "examine, commit" to be atomic
    - » Recall our "mutex assumptions"

Q2 –Trouble at the Warehouse

General approach

- One mutex
- Multiple condition variables
  - One for each reason somebody should sleep / wake
    - » Loading dock availability
    - » Availability of each kind of stock
    - » Availability of forklift
    - » Etc.

Q3 –Dual-priority Locking

The mission

- Write a “fancy lock”
  - Each thread is either high-priority or low-priority
  - When lock is released, it should go to a high-priority thread if any are waiting
- Objects you need
  - Mutex
    - » You need one to protect competing accesses to state
    - » More than one is asking for trouble –who holds what should be encoded in the state, not in a mutex, which should be held only very briefly
  - Two thread counts, two cvars (note the relationship)
  - Optionally one extra variable
    - » Logically makes sense; got most people into trouble

Q3 –Dual-priority Locking

Frequent hazards

- Leaking memory in init
  - If you got “see course staff”, please do so
- Forgetting about “the third thread”
  - Considered: one unlocker, one high-priority thread which you expect/home will run
  - But another (low-priority) thread might always capture the lock
  - Lock state must somehow make this case visible to the third thread
  - See lecture material for detailed “third thread” example
- Too few / too many cvars
  - Define the key state-change transitions, give each a cvar
- Deadlock, etc.

Q4 –Deadlock

Question tested understanding of multiple details

- Imposing a locking order (to avoid circular wait)
- Safe sequence

Frequent hazards

- Confusing hold&wait vs. circular wait
  - Almost every application involves hold&wait
- Inadequate understanding of safe sequence
- Omissions (e.g., not drawing process/resource graph)

Advice

- Go back and understand this thoroughly
- It is one of the key non-programming concepts of the class

Q5 –Stack Picture

Key elements of solution

- Enough stack frames
- Enough pieces in each stack frame
- Getting the struct in the right place
- Not putting strings in strange places

Graded fairly gently

Breakdown

90% = 67.5	3 students
80% = 60.0	9 students
70% = 52.5	23 students (52 and up)
60% = 45.0	11 students
50% = 37.5	13 students
<50%	7 students

Comparison

- Scores are lower than typical

Implications

We adjusted scores upward

- **Something like 3-5 points**

Score below 70%?

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

Warning...

- **To pass the class you must demonstrate reasonable proficiency on exams (project grades alone are not sufficient)**
- **See syllabus**