15-410

"My other car is a cdr" -- Unknown

Exam #1
Mar. 19, 2007

Dave Eckhardt

Synchronization

Checkpoint 2 –Friday, in cluster

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

Google “Summer of Code”

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

CMU SCS “Coding in the Summer”

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

Starvation

- Key concept: repeatedly losing a resource-acquisition "game" to the point of genuine, observable unfairness
  - Definitely not: "I want Disk 3, but somebody locks it and enters an infinite loop"
  - In practice, frequently involves trying to acquire multiple resources
- Starvation happens even when there is no deadlock
- Starvation happens even when the system makes progress
- Starvation vs. critical-section "bounded waiting"
  - Some relationship, but not two names for the same thing

Q1 –Short Answer

Thread-safe

- **Key concept: “Can be simultaneously invoked by multiple threads with correct states and answers resulting”**
- **Mutexes (or other locks) needed *when multiple threads access shared state***
  - **If no state is shared, thread-safety doesn't require locks!**
  - **Sometimes state can be shared without locks!**

Q2 –Process Model

"Write code to probe the kernel's response to attemping to read/write to the ____ region, which should/shouldn't be allowed"

Concepts

- What is a region?
- Which C constructs are located in which regions?

```
char *s = "Hello, Sailor!";
char s[] = "Hello, Sailor!";
```

- Which system calls {write,read} memory
  - (new_pages(), remove_pages() don't!)

Tests: What are the parts? How do parts fit together?

Q3 – Test Condition Variables

The mission

- Write test code: if two threads are awaiting a condition, and the condition is signaled, exactly one thread is awakened. Test should complete in 10 seconds.

The answer?

Q3 – Test Condition Variables

The mission

- Write test code: if two threads are awaiting a condition, and the condition is signaled, exactly one thread is awakened. Test should complete in 10 seconds.

The answer?

- [left as an exercise for the reader]

Q3 – Test Condition Variables

The parts

- Launch two threads (popular, ok: three)
- Measure when two threads are awaiting a condition
  - (not quite as good: "convince" threads)
  - (not quite as good: verify thread state looks like what *your* condition variables do)
  - Subtlety: the best test code doesn't wait for threads to *stop*
- Signal the condition variable
- "Allow some time" for threads to awaken!
- Measure how many threads wake up
  - Detect "2 threads awaken"
  - Detect "0 threads awaken"
- Make use of / comply with the 10-second requirement

Q3 – Test Condition Variables

Key issues

- Must hold mutex before you ask cond_wait() to unlock it
- "Semi-locked state" is generally unwise
  - Some people manipulate an int while holding a mutex
  - Other people peer at the int while not holding the mutex
  - The peering thread learns much less than it might seem
    - » State change of the int is inherently disconnected from other state changes of the manipulating thread
    - » On a machine with "modern memory"...
      - The int's state change may be "from the future"!

Q3 – Test Condition Variables

Advice

- Turn text into checklist ("The parts" above)
- Think about measurement
  - What are you trying to measure?
  - How can you most simply measure each thing?
- Think about race conditions
  - I want *this* to happen, then *that*
  - What *other* event orders are possible?

Q3 – Test Condition Variables

C

```
typedef struct nexus { ... } *nexus_p;

void nexus_init(nexus_p np)
{
  np ...?
}
```

Q3 – Test Condition Variables

C

```
typedef struct nexus { ... } *nexus_p;

void nexus_init(nexus_p np)
{
  /* pick one of these two */
  np = malloc(sizeof (struct nexus));
  np = malloc(sizeof (*np));
}
```

Q3 – Test Condition Variables

C

```
typedef struct nexus { ... } *nexus_p;

void nexus_init(nexus_p np)
{
  np = malloc(sizeof (*np));
  /* why is this fundamentally wrong? */
}
```

Q4 –Inscrutable Code

Tested

- What do the building blocks do?
  - thread_fork, set_status(), vanish()
- How do they fit together? (Even when “abused”)
- What does a piece of code *do*?
  - (especially: code written by somebody else)
  - (not: What does it *look* like it does? What are its hopes?)
- Detect a race condition when shown thread code

Residual confusion

- Relationship of thread_fork to thread-stack creation
- Relationship of vanish() to thread-stack removal
- One hazard: “Read your partner's code”

Q5 –Deadlock

Key issues

- Drawing a process/resource graph
  - Graded somewhat gently, but you need to clearly:
    - » Differentiate between actors and objects
    - » Differentiate between requesting and owning
    - » Recognize and portray deadlock
- Describing the deadlock
  - Make sure you can think deadlocks through in terms of the four necessary ingredients –*each* deadlock exhibits *all four*
- Fixing the problem
  - This time the “standard solution” helps
  - But there is still a “Which one is better?” design step

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**

Implications

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