15-410

“...What does BSS stand for, anyway?...”

Exam #1
Oct. 25, 2004

Dave Eckhardt
Synchronization

Final Exam list posted
  - You *must* notify us of conflicts in a timely fashion

Checkpoint 2 – Wednesday, in cluster

Book report topic chosen? Great for airplane time...

Upcoming events
  - 15-412
  - 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 implementation

Examination will change to match

- More design questions
- Some things you won't have implemented
Outline

Question 1
Question 2
Question 3
Question 4
Question 5
Q1 – Definitions (graded *gently*)

**BSS**
- Blank storage space?
- Blank static storage?
- Block started by symbol
  - According to Wikipedia
    » Directive for IBM 704 assembler (1950's)
  - Where all the zeroes go when you erase the blackboard

**inb()**
- Is not a system call

**trap frame**
- Execution state the CPU saves on interrupt/exception/trap
Q2 – Interrupt Handling

The “1024 registers problem”
  • Can't afford to save 1024 registers millions of times/sec.

Solution
  • Ok, don't save all the registers!
  • Save the ones you'll use while running the interrupt handler.
Q2 – Interrupt Handling

Second problem

- How do I know which registers the interrupt handler will use?

Solutions

- Write whole interrupt handler in assembly language (urgh).
- Special compiler flags
  - While compiling foo.c, use only registers 0..16
    » Wrapper can save and restore only those 16
  - Treat *all* registers as callee-save
    » Maybe less efficient, maybe doesn’t matter
Q3: Implicit Thread Exit

```c
int main() {
    void *status;
    thr_init(16*1024);
    thr_join(
        thr_create(foo, (void *) 0),
        NULL, &status);
    thr_exit(status);
}
```

What if it said “return(status)” instead?
Q3: Implicit Thread Exit

Problem: return(s) means different things

- Random procedure: return to caller
- main(), without threads: exit(s)
- main(), with threads: thr_exit(s)

How is “exit()” case handled?

- _main(), which calls exit(main(argc, argv));

How can we extend this approach?

- _main() could do something different
  s = main(argc, argv);
  if (thr_init_happened) thr_exit(s);
  else exit(s);
Q3: Implicit Thread Exit

Other approaches

- Leave \_main() alone but change exit() wrapper
- Asking thr\_init() to patch the stack
  - ...so main() returns to something\_special() instead of to \_main()

Stack patching

- Issue: how to locate main()’s return address on stack?
  - One approach: know start of main(), length of main()
- Issue: can \textit{not} set main()’s return address to thr\_exit()...
  - Where does thr\_exit() look for status value?
Q4: Deadlock

This is a deadlock question

- Lots of systems contain deadlock
- Deadlock is hard to deal with
  - Usually can't "define it away"
  - If you try, you probably end up with starvation instead
  - There is often no really satisfying solution

Should be easy to see the deadlock in this problem

- CD burners are inherently exclusive-access
- Preempting a CD burner breaks the product
  - Device driver won't let you do that, so non-preemption is natural
- Loop around alloc_drive(BURNER) is exactly hold&wait
- Application wants any burner, so you get cycles
Q4: Deadlock

Approaches

- Prevention
  - Banning mutual exclusion or non-preemption isn't really feasible
  - Banning hold&wait is possible
    » Popular: allocate all burners at once
      • Also popular: starving large requests
      • There is an *inherent tension* here
    » Popular: allocate as many burners as currently available
      • Problem: burning 100 copies 1-by-1 is prohibitive
      • Note: that is *not* “high throughput”!
  - Banning cycles is odd...
    » Result: given thread can allocate only random subset of drives
    » Easy to approximate 1-by-1...
Q4: Deadlock

Approaches

▪ Avoidance
  ▪ Natural
  ▪ Need to watch out for starvation/inefficiency here too

▪ Detection/recovery
  ▪ Rebooting the machine means a machine full of bad discs...

Summary

▪ “Job scheduling” is hard
  ▪ Throughput vs. starvation is often an issue

▪ Real problems often contain painful messy issues
  ▪ Can’t find perfect solution if there isn’t one.
Q5: mutex_unlock()

void mutex_lock(mutex_t *m) {
    while (xchg(&m->status, LOCKED) != UNLOCKED)
        yield(m->owner);
    m->owner = gettid();
}
Q5: `mutex_unlock()`

```c
void mutex_unlock_one(mutex_t *m) {
    m->owner = -1;
    m->status = UNLOCKED;
}

void mutex_unlock_two(mutex_t *m) {
    m->status = UNLOCKED;
    m->owner = -1;
}
```

**What is desirable about #2?**

**Why is #2 subtly but horribly wrong?**
Q5: mutex_unlock()

```c
void mutex_unlock_one(mutex_t *m) {
    m->owner = -1;
    m->status = UNLOCKED;
}

void mutex_unlock_three(mutex_t *m) {
    m->owner = -1;
    m->status = UNLOCKED;
    yield(-1);
}
```

What desirable feature does the `yield()` add to mutexes?

What assumption argues the other way?
Summary

90% = 67.5 7 students
80% = 60.0 28 students
70% = 52.5 13 students
60% 8 students
<60% 9 students