“Real concurrency – in which one program actually continues to function while you call up and use another – is more amazing but of small use to the average person. How many programs do you have that take more than a few seconds to perform any task?” – NYT, 4/25/1989
Synchronization

Partner sign-up!

- Approximately 14 students un-partnered
- I have spammed the un-signed... note that HH B103 is a good place to meet potential partners!

Project 1

- By end of Wednesday...
  - Console (output) should be “doing something”, “not far”
  - Should have “some progress” for kbd, timer
    » Should really have at least “solid design”
    » Better to have handled one interrupt once

Write good code

- Console driver will be used \textit{(and extended)} in P3
Synchronization

**Simics issues**

- Simics doesn't simulate time with 100% accuracy
  - Mentioned in handout, but:
    - Sometimes it runs slower (“of course”)
    - Sometimes it runs faster (!)

- Simics doesn't blink
  - Not your fault

- Arrow keys may not work “so well”
  - If you get a strange message, tell us about your setup
  - Crash box!
Readings

Textbook chapters

- OSC
  - Already: Chapters 1 through 3
  - Today: Chapter 4 (roughly)
  - Soon: Chapters 6 & 7
    - Transactions (6.9) will be deferred

- OS:P+P
  - Already: Chapters 1 through 3
  - Today: Chapter 4 (roughly/partly)
  - Soon: Chapter 6
Book Report Goals

There's more than one way to do it

- But you don't have time to try all the ways in 410
- Reading about other ways is good, maybe fun

Habituation

- Long-term career development requires study

Writing skills (a little!)

- “Summarizing” a book in a page is tough
Book Report Goals

Some of you are going to grad. school

Some of you are wondering about grad. school

Some of you are in grad. school
  - You should be able to read a Ph.D. dissertation

More generally
  - Looking at something in depth is different
  - Not like a textbook
Book Report

Read the “handout”

Browse the already-approved list

Pick something (soon)
  - “Don't make me stop the car...”

Read a bit before you sleep at night
  - or: before you sleep in the morning
  - and/or: Thanksgiving break / Spring break

Assignment recommended by previous OS students!
  - They recommend starting early, too
Road Map

Thread lecture

Synchronization lectures
  ▪ Probably three

Yield lecture

This is important
  ▪ When you leave here, you will use threads
  ▪ Understanding threads will help you understand the kernel

Please make sure you understand threads
  ▪ We'll try to help by assigning you P2
Outline

Thread = schedulable registers
  - (that's *all* there is)

Why threads?

Thread flavors (ratios)

(Against) cancellation

Race conditions
  - 1 simple, 1 ouch
    - *Make sure you really understand this*
Single-threaded Process

Stack

Heap

Data

Code

Registers

stdin

stdout

timer
Multi-threaded Process

Stack ↓  ↓  ↓
Stack
Stack
Heap
Data
Code

Registers

stdin
stdout
timer
What does that *mean*?

**Three stacks**
- Three sets of “local variables”

**Three register sets**
- Three stack pointers
- Three %eax's (etc.)

**Three *schedulable RAM mutators***
- (heartfelt but partial apologies to the ML crowd)

**Three potential bad interactions**
- A/B, A/C, B/C ... this pattern gets worse fast...
Why threads?

- Shared access to data structures
- Responsiveness
- Speedup on multiprocessors
Shared access to data structures

Database server for multiple bank branches

- Verify multiple rules are followed
  - Account balance
  - Daily withdrawal limit
- Multi-account operations (transfer)
- Many accesses, each modifies tiny fraction of database

Server for a multi-player game

- Many players
- Access (& update) shared world state
  - Scan multiple objects
  - Update one or two objects
Shared access to data structures

Process per player?
- *Processes* share objects only via system calls
- Hard to make game objects = operating system objects

Process per game object?
- “Scan multiple objects, update one”
- Lots of message passing between processes
- Lots of memory wasted for lots of processes
- Slow
Shared access to data structures

**Thread per player**
- Game objects inside single memory address space
- Each thread can access & update game objects
- Shared access to OS objects (files)

**Thread-switch is cheap**
- Store N registers
- Load N registers
Responsiveness

“Cancel” button vs. decompressing large JPEG

- Handle mouse click *during* 10-second process
  - Map (x,y) to “cancel button” area
  - Change color / animate shadow / squeak / ...
  - Verify that button-release happens in button area of screen
- ...without JPEG decompressor understanding clicks
- Actually *stopping* the decompressor is a separate issue
  - Threads allow the user to register intent while it’s running
Multiprocessor speedup

More CPUs can't help a single-threaded process!

PhotoShop color dither operation

- Divide image into regions
- One dither thread per CPU
- Can (sometimes) get linear speedup
Kinds of threads

User-space (N:1)
Kernel threads (1:1)
Many-to-many (M :N)
User-space threads (N:1)

**Internal threading**
- Thread library adds threads to a process
- Thread switch “just swaps registers”
  - Small piece of asm code
  - Maybe called yield()
User-space threads (N:1)

+ No change to operating system

- Any system call probably blocks all “threads”
  - “The process” makes a system call
  - Kernel blocks “the process”
  - (special non-blocking system calls can help)

- “Cooperative scheduling” awkward/insufficient
  - Must manually insert many calls to yield()

- Cannot go faster on multiprocessor machines
Pure kernel threads (1:1)

**OS-supported threading**
- OS knows thread/process ownership
- Memory regions shared & reference-counted

---

Diagram:
- Stack
- Heap
- Data
- Code
- Registers
Pure kernel threads (1:1)

“Every thread is sacred”
- Kernel-managed register set
- Kernel stack for when the thread is running kernel code
- “Real” (timer-triggered) scheduling

Features
+ Program runs faster on a multiprocessor
+ CPU-hog threads don't get all the CPU time
- User-space libraries must be rewritten to be “thread safe”
- Requires more kernel memory
  - 1 PCB $\Rightarrow$ 1 TCB + N tCB's,
  - 1 k-stack $\Rightarrow$ N k-stacks
Many-to-many (M:N)

**Middle ground**

- OS provides kernel threads
- M user threads *share* N kernel threads
Many-to-many (M:N)

Sharing patterns

- Dedicated
  - User thread 12 owns kernel thread 1
- Shared
  - 1 kernel thread per hardware CPU
  - Each kernel thread executes next runnable user thread
- Many variations, see text

Features

- Great when all the schedulers work together as you expected!
(Against) Thread Cancellation

Thread cancellation

- We don't want the result of that computation
  - ("Cancel button")
- Two kinds – “asynchronous”, “deferred”

Asynchronous (immediate) cancellation

- Stop execution *now*
  - Run 0 more instructions (at least, in user space)
  - Free stack, registers
  - Poof!
- Hard to garbage-collect resources (open files, ...)
- Difficult to maintain data-structure consistency!
(Against) Thread Cancellation

Deferred ("pretty please") cancellation

- Write down “Dear Thread #314, Please go away.”
- Threads must check for cancellation
- Or define safe cancellation points
  - “Any time I call close() it's ok to zap me”

The only safe way

- Unless your threads are running very unusual code!
Race conditions

**What you think**

```c
ticket = next_ticket++; /* 0 ⇒ 1 */
```

**What really happens (in general)**

```c
ticket = temp = next_ticket;    /* 0 */
++temp;                      /* 1, but not visible */
next_ticket = temp;          /* 1 is visible */
```
Murphy's Law (of threading)

The world may *arbitrarily* *interleave* execution

- **Multiprocessor**
  - N threads executing instructions *at the same time*
  - Of course effects are interleaved!

- **Uniprocessor**
  - Only one thread running at a time...
  - But N threads runnable, timer counting down toward zero...

The world will choose the *most painful* interleaving

- “Once chance in a million” happens every minute
## Race Condition – Your Hope

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>tkt = tmp = n_tkt; 0</td>
<td>tkt = tmp = n_tkt; 1</td>
</tr>
<tr>
<td>++tmp; 1</td>
<td>++tmp; 2</td>
</tr>
<tr>
<td>n_tkt = tmp; 1</td>
<td>n_tkt = tmp; 2</td>
</tr>
</tbody>
</table>

T0 has ticket 0, T1 has ticket 1. next_tkt has value 2. Your boss is happy.
# Race Condition – Your Bad Luck

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tkt = tmp = n_tkt); 0</td>
<td>(tkt = tmp = n_tkt); 0</td>
</tr>
<tr>
<td>(++tmp);</td>
<td>(++tmp);</td>
</tr>
<tr>
<td>(n_tkt = tmp); 1</td>
<td>(n_tkt = tmp); 1</td>
</tr>
</tbody>
</table>

T0 has ticket 0, T1 has ticket 0. next\_tkt has value 1. Your boss is not entirely happy.
What happened?

Each thread did “something reasonable”
- ...assuming no other thread were touching those objects
- ...that is, assuming “mutual exclusion”

The world is cruel
- Any possible scheduling mix will happen sometime
- The one you fear will happen...
- The one you didn't think of will happen...
The #! shell-script hack

What's a “shell script”?  
- A file with a bunch of (shell-specific) shell commands
  ```
  #!/bin/sh
  echo "My hovercraft is full of eels."
  sleep 10
  exit 0
  ```
- Or: a security race-condition just waiting to happen...
The `#!` shell-script hack

**What's "#!"?**
- A venerable hack

**You say**
- `execl("/foo/script", "script", "arg1", 0);`

/foo/script “executable file” begins...
- `#!/bin/sh`

**The kernel rewrites your system call...**
- `execl("/bin/sh" "/foo/script" "arg1" , 0);`

**The shell does**
- `open("/foo/script", O_RDONLY, 0);`
The setuid invention

U.S. Patent #4,135,240
- Dennis M. Ritchie
- January 16, 1979

The concept
- A program with stored privileges
- When executed, runs with two identities
  - invoker's identity
  - program owner's identity
- Can switch identities at will
  - Open some files as invoker
  - Open other files as program-owner
Setuid example - printing a file

Goals

- Every user can queue files
- Users cannot delete other users' files

Solution

- Queue directory owned by user printer
- Setuid queue-file program
  - Create queue file as user printer
  - Copy joe's data as user joe
- Also, setuid remove-file program
  - Allows removal only of files you queued
- User printer mediates user joe's queue access
Race condition example

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ln -s /bin/lpr /tmp/lpr</code></td>
<td><code>run /tmp/lpr</code></td>
</tr>
<tr>
<td></td>
<td><code>[setuid to user &quot;printer&quot;]</code></td>
</tr>
<tr>
<td></td>
<td><code>start &quot;/bin/sh /tmp/lpr...&quot;</code></td>
</tr>
<tr>
<td><code>rm /tmp/lpr</code></td>
<td></td>
</tr>
<tr>
<td><code>ln -s /my/exploit /tmp/lpr</code></td>
<td><code>script = open(&quot;/tmp/lpr&quot;);</code></td>
</tr>
<tr>
<td></td>
<td><code>execute /my/exploit</code></td>
</tr>
</tbody>
</table>
What happened?

**Intention**
- Assign privileges to program contents

**What happened?**
- First, name was mapped to privileges
  - (name $\Rightarrow$ file, file $\Rightarrow$ privileges)
- Next, program name was re-bound to a different file
- Then, name was mapped to contents
  - (name $\Rightarrow$ different file, different file $\Rightarrow$ different contents)

How would you fix this?
How to solve race conditions?

Carefully analyze operation sequences

Find subsequences which must be **uninterrupted**
  - “Critical section”

Use a **synchronization mechanism**
  - Next time!
Summary

Thread: What, why

Thread flavors (ratios)

Race conditions
  - Make sure you really understand this
Further Reading

Setuid Demystified
  - Hao Chen, David Wagner, Drew Dean
  - “Abandon hope all ye who enter here”

The “cancel button problem”
  - “Attentiveness: Reactivity at Scale”
    - Gregory S. Hartman
    - CMU-ISR-10-111.pdf
    - (on the book-report list)