15-410

The Thread
Jan. 30, 2006

Dave Eckhardt

Bruce Maggs

"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

Project 1

- **By end of today...**
  - **Should have run simics once**
  - **Should try to make something happen on the screen**
- **“Soon”**
  - **Should have some progress (at least design) for kbd, timer**

Write good code

- **Console driver will be used *(and extended)* in P3**

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 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

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: spring break

Recommended by previous OS students!

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

Textbook chapters

- **Already: Chapters 1 through 4**
- **Today: Chapter 5 (roughly)**
- **Soon: Chapters 7 & 8**
- **Transactions (7.9) will be deferred**

Outline

Thread = schedulable registers

- **(that's *all* there is)**

Why threads?

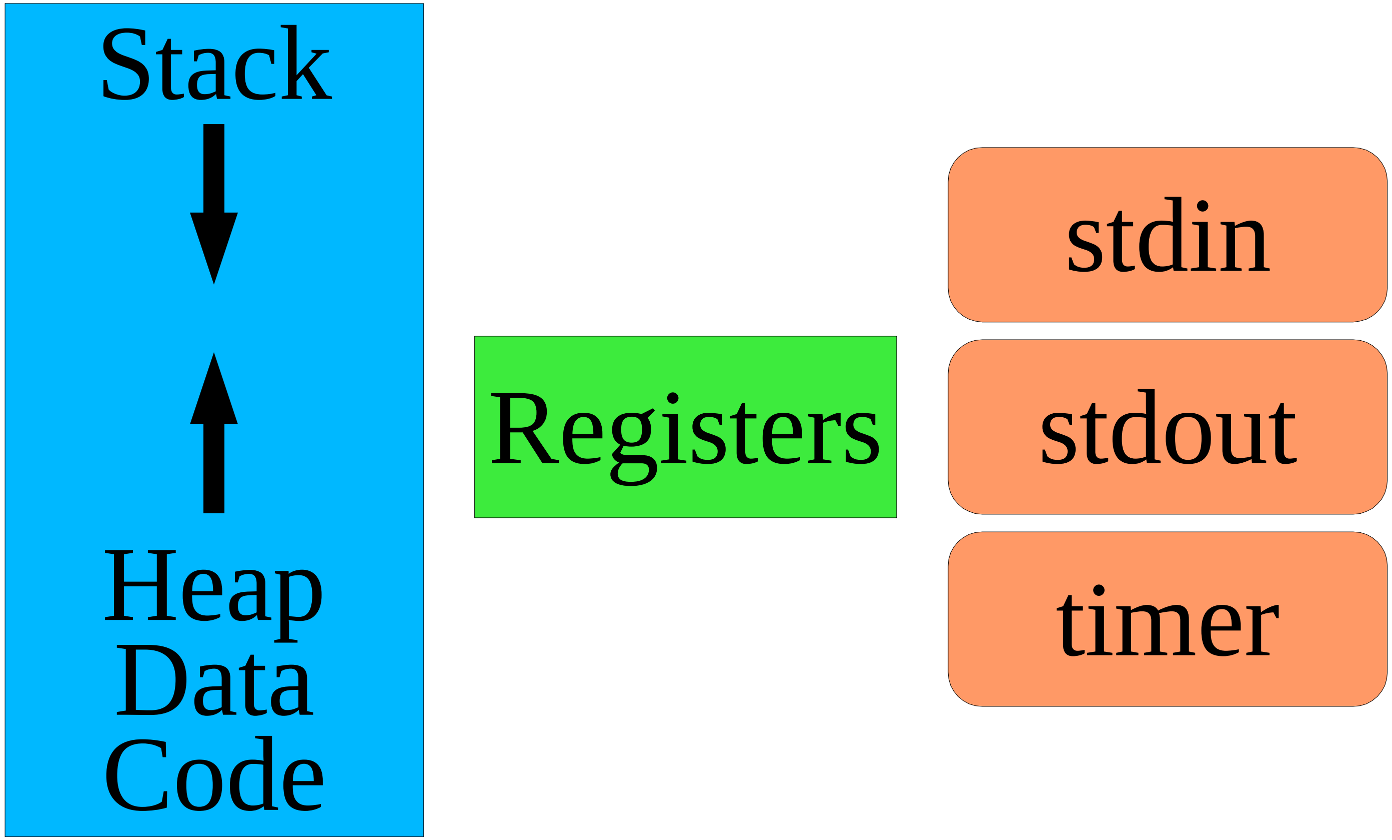
Thread flavors (ratios)

(Against) cancellation

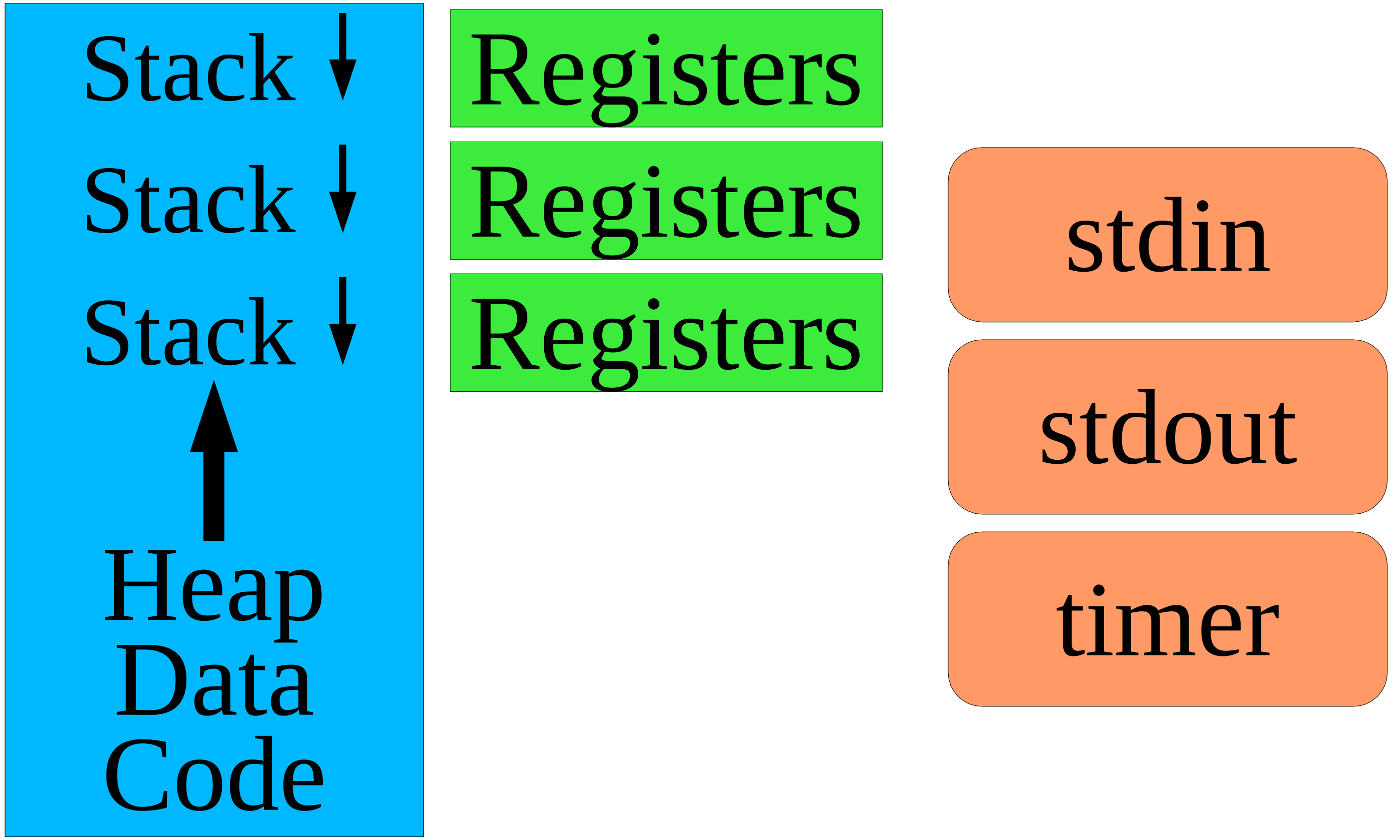
Race conditions

- **1 simple, 1 ouch**
- ***Make sure you really understand this***

Single-threaded Process



Multi-threaded Process



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

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**
  - **Verify that button-release happens in button area of screen**
- **...without JPEG decompressor understanding clicks**

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)

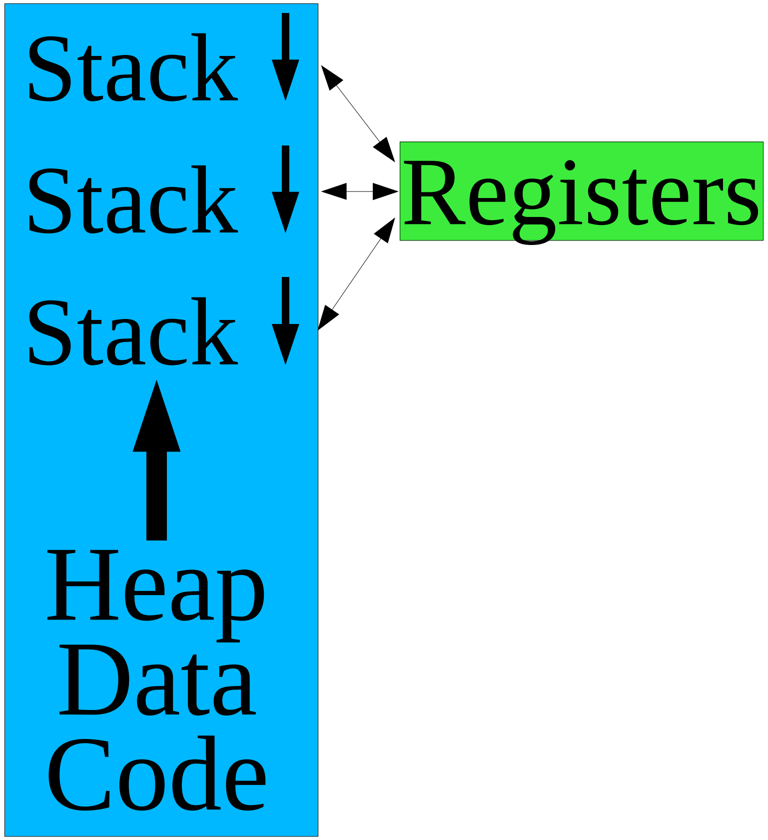
Kernel threads (1:1)

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

- 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

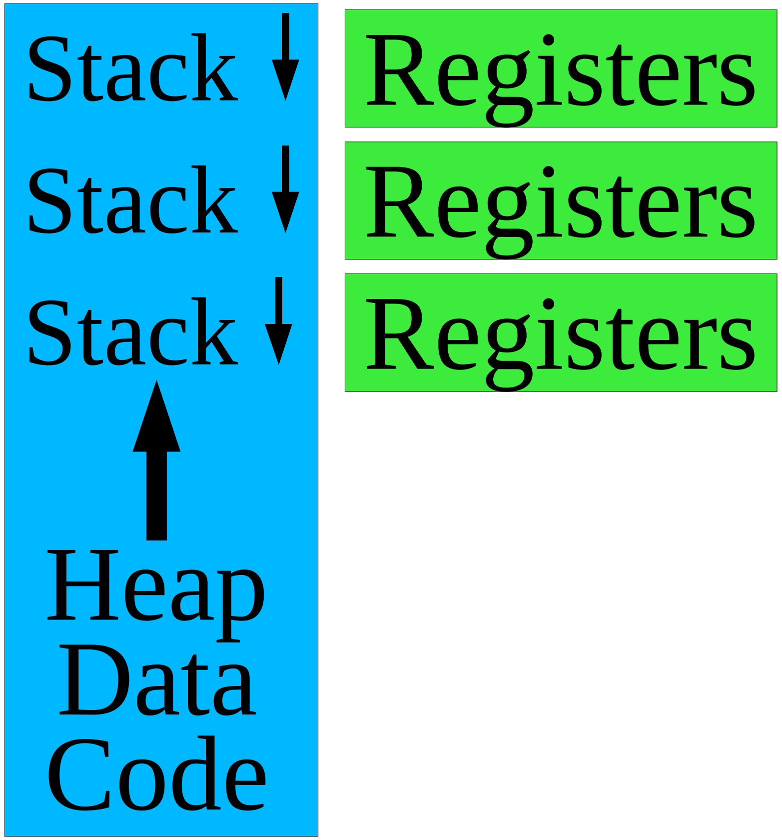
- **Must manually insert many calls to yield()**

- Cannot go faster on multiprocessor machines

Pure kernel threads (1:1)

OS-supported threading

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



Pure kernel threads (1:1)

Every thread is sacred

- Kernel-managed register set
- Kernel stack
- “Real” (timer-triggered) scheduling

Features

+ Program runs faster on multiprocessor

+ CPU-hog threads don't get all the CPU time

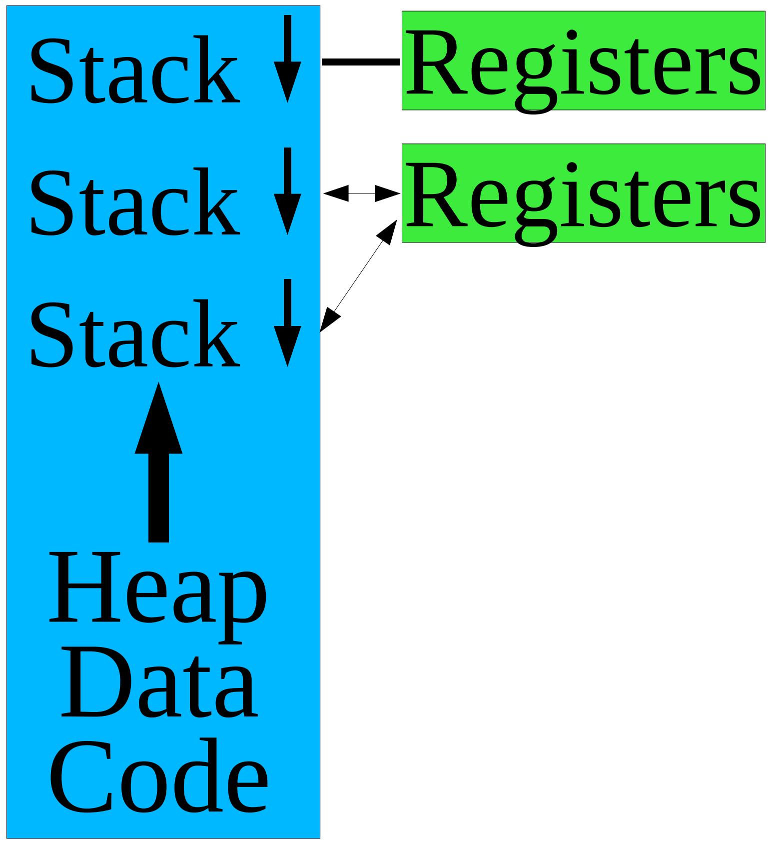
- User-space libraries must be rewritten

- Requires more kernel memory
  - 1 PCB $\Rightarrow$ 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 scheduling works as you expected!

(Against) Thread Cancellation

Thread cancellation

- We don't want the result of that computation
  - (“Cancel button”)

Asynchronous (immediate) cancellation

- Stop execution *now*
  - Free stack, registers
  - Poof!
- Hard to garbage-collect resources (open files, ...)
- Invalidates data structure consistency!

(Against) Thread Cancellation

Deferred ("pretty please") cancellation

- **Write down "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 (IMHO, IDHO)

Race conditions

What you think

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

What really happens (in general)

```
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

T0		*T1*	
`tkt = tmp = n_tkt;`	0		
`++tmp;`	1		
`n_tkt = tmp;`	1		
		`tkt = tmp = n_tkt;`	1
		`++tmp;`	2
		`n_tkt = tmp;`	2
Final value	1		2

Race Condition – Your Bad Luck

T0		*T1*	
`tkt = tmp = n_tkt;`	0		
		`tkt = tmp = n_tkt;`	0
`++tmp;`	1		
		`++tmp;`	1
`n_tkt = tmp;`	1		
		`n_tkt = tmp;`	2
Final value	1		1

Two threads have same "ticket"!

What happened?

Each thread did "something reasonable"

- **...assuming no other thread were touching those objects**
- **...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 begins...

- #!/bin/sh

The kernel does...

- 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`
- **User** `printer` **controls user** `joe`**'s queue access**

Race condition example

Process 0	*Process 1*
`ln -s /bin/lpr /tmp/lpr`	
	`start script /tmp/lpr`
	`[become user "printer"]`
`rm /tmp/lpr`	
`ln -s /my/exploit /tmp/lpr`	
	`run /bin/sh /tmp/lpr`
	`script = open("/tmp/lpr");`
	`execute /my/exploit`

What happened?

Intention

- **Assign privileges to program contents**

What happened?

- **Privileges were assigned to program *name***
- **Program name pointed to 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***