Threads

David A. Eckhardt
School of Computer Science
Carnegie Mellon University

de0u@andrew.cmu.edu

How's it going?

You should have *something* running

- Like, everything but the clock
- If you haven't run simics, today's the day!

Outline

Textbook chapters

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

Thread as schedulable registers

- (that's *all* there is)

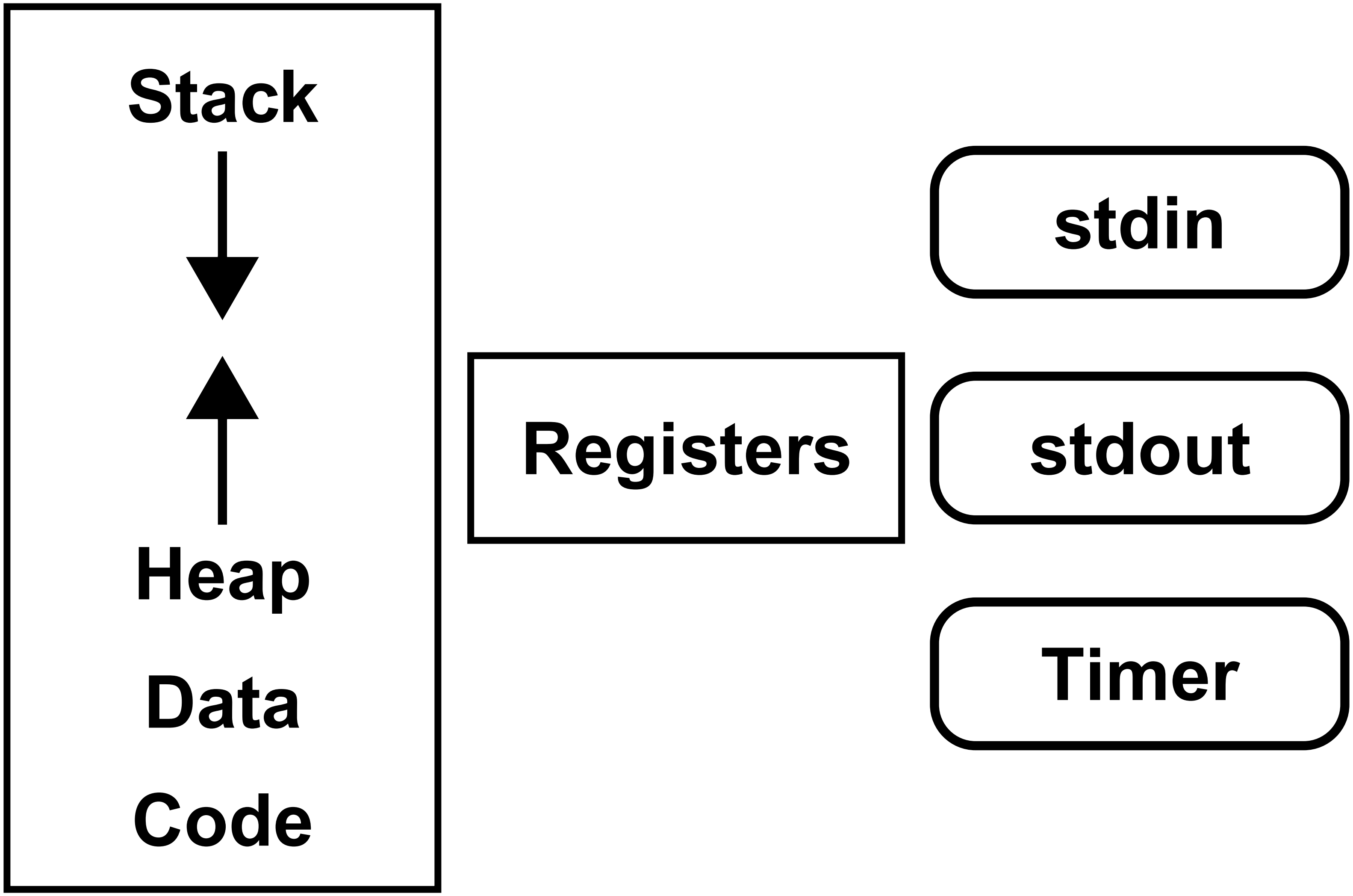
Misc. topics

- Why threads?
- Thread flavors (ratios)
- (Against) cancellation
- Thread-specific Data

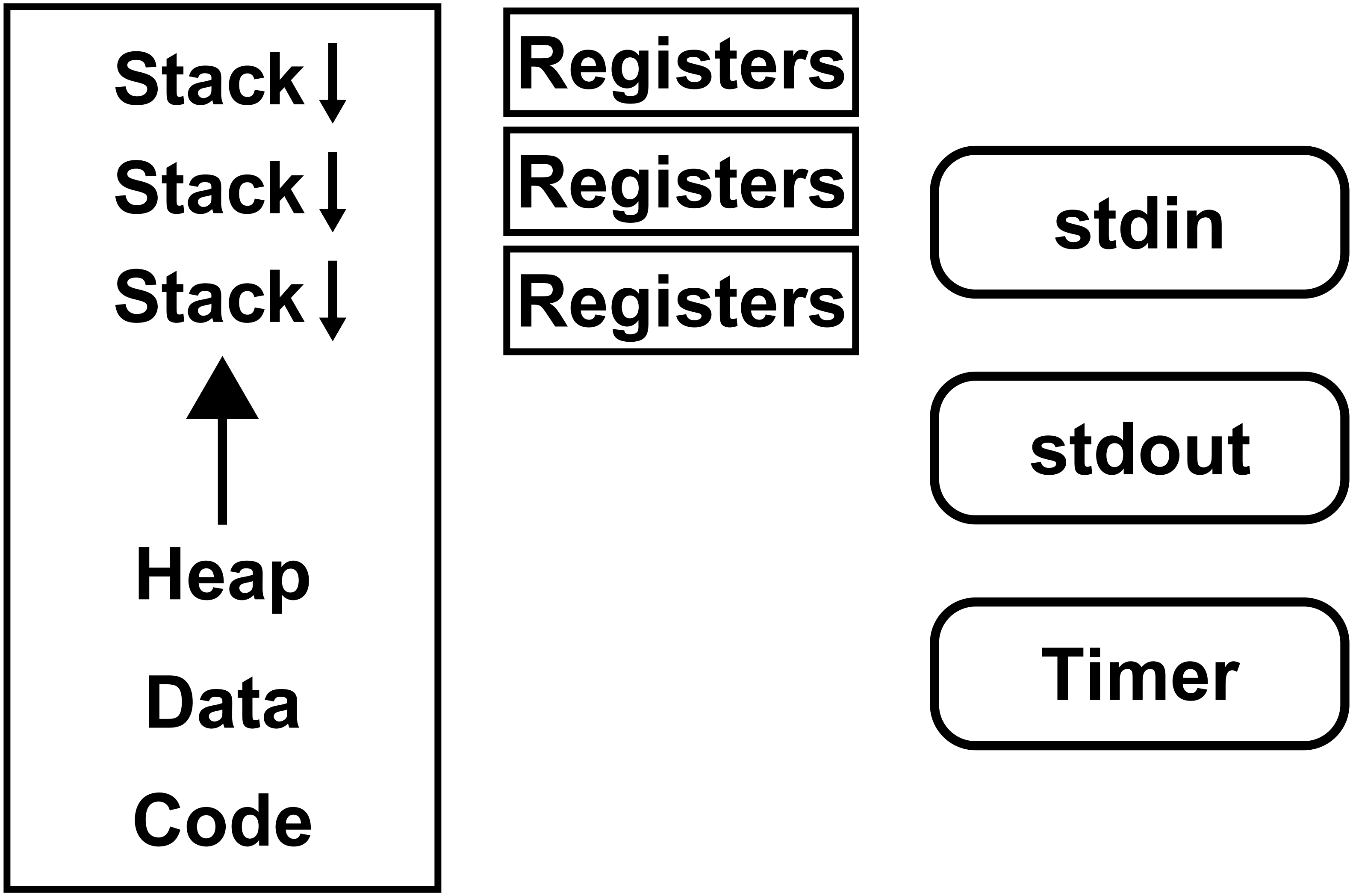
Race conditions

- 1 simple, 1 "ouch"

Single-threaded process



Multi-threaded process



Why threads?

Performance

- Simplistic: copying registers cheaper than copying process
    - also: context switching...
- Looking deeper: cheap access to shared resources

Multiplayer game server

- Many players
- Access (& update) shared world state

1 process per player?

- *Processes* share objects only via system calls
- Hard to make game objects = operating system objects
- Expensive to devote a process per game object

1 *thread* per player

- Easy access to game objects in memory
- Shared access to OS objects (files)

More Performance

Responsiveness

- Conveniently suspend stalled operation
- Allow another operation to progress
- ...without horrible manual coding

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

User-space threads (N:1)

Internal threading

- Processes optionally thread via special library
- Thread switch "just" copies registers
  - register save/restore, stack swap

Features

- No change to operating system
- System call may block *all* threads
  - (special non-blocking system calls can help)
- "Cooperative scheduling" awkward/insufficient
  - How many calls to yield()?
- Does not take advantage of multiprocessor machines

Pure kernel threads (1:1)

OS-supported threading

- OS models thread/process ownership
  - memory regions shared & reference-counted
- Every thread is sacred
  - Kernel-managed register set
  - Kernel stack
  - Independently scheduled

Features

- “Real” (timer-triggered) scheduling
- Takes advantage of multiprocessor machines
- User-space libraries must be rewritten
- Kernel threads may be costly
  - must be created via system call
  - require kernel memory (PCB, stack)

Many-to-many (M:N)

Middle ground

- OS provides kernel threads
- M user threads share N kernel threads

Sharing patterns

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

Features

- Great when it works!

(Against) Cancellation

Thread cancellation

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

Asynchrounous (immediate) cancellation

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

Deferred (“pretty please”) cancellation

- Write down “thread #314, please go away”
- Requires threads to check or define safe cancellation points
- The only safe way (IMHO)

Thread-specific Data

A little anti-sharing

- Threads share code, data, heap
- How to write these?
  - printf("I am thread %d\n", thread_id());
  - thread_status[thread_id()] = BUSY;
  - printf("Client machine %s\n", thread_var(0));

No magic, so...

- thread_id() = system call?
  - too expensive!
- thread_id() = { extern int thread_id; return (thread_id); }
  - shared memory: all int's have same value

Two options

- Think about what's *not* shared...

Thread-specific data: implementation

Reserved register

- Many microprocessors have 32 (or more) user registers
- Devote one to thread data pointer
    - struct thread_private
        - int thread_id;
        - void *thread_vars[N_TH_VAR];
- X86 architecture has *four* general-purpose registers (oops)

Stack trick

- Assume all thread stacks have same size
- Store private data area at top of stack
- Compute “top of stack” given any address within stack
    - “exercise for the reader”

Race conditions

What could go wrong?

- What you think
  - ticket = next_ticket++;
- What really happens (in general)
  - ticket = temp = next_ticket;
  - ++temp;
  - next_ticket = temp;

Murphy's Law (of threading)

- The world is allowed to arbitrarily interleave execution
- Sooner or later it will choose the most painful way

Race condition example

Blow-by-blow

Thread 1	Thread 2
ticket = temp = next_ticket;	
	ticket = temp = next_ticket;
++temp;	
	++temp;
next_ticket = temp;	
	next_ticket = temp;

The #! shell-script hack

What's a "shell script"?

- A file with a bunch of (shell-specific) shell commands

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);

How convenient!

- (Solaris does something similar for Java class files)

The setuid invention

The concept

- A program with stored privileges
- When executed, runs with two identities
  - invoker's identity
  - file owner's identity

Example - printing a file

- Want every user to be able to queue files
- Don't want users to delete other user's files from queue
- Solution
  - Queue directory owned by user "printer"
  - Setuid "queue-file" program
    - Create queue file as user "printer"
    - Copy user data as user "joe"
  - User "printer" controls user "joe"'s access to directory

The race condition

Process 1	Process 2
ln -s /foo/script /tmp/script	
	execl("/tmp/script");
	become "printer"
	execl('/bin/sh", "/tmp/ script");
rm /tmp/script	
ln -s /my/exploit /tmp/script	
	script = open('/tmp/script", 0);
	execute /my/exploit ...

How to solve race conditions?

Carefully analyze operation sequences

Find subsequences which must be ***uninterrupted***

- “Critical section”

Use a ***synchronization mechanism***

- Next time!