The Thread

Dave Eckhardt
de0u@andrew.cmu.edu

Synchronization

- If you haven't run simics yet
  - You could be in ***real trouble***
  - Your screen driver should be done (at least)
- This isn't like other programming
  - C (not C++, not Java) – things don't happen for you
  - Assembly language
  - Hardware isn't clean
- Project 1 is a ***warm-up***
  - Next stop: thread library

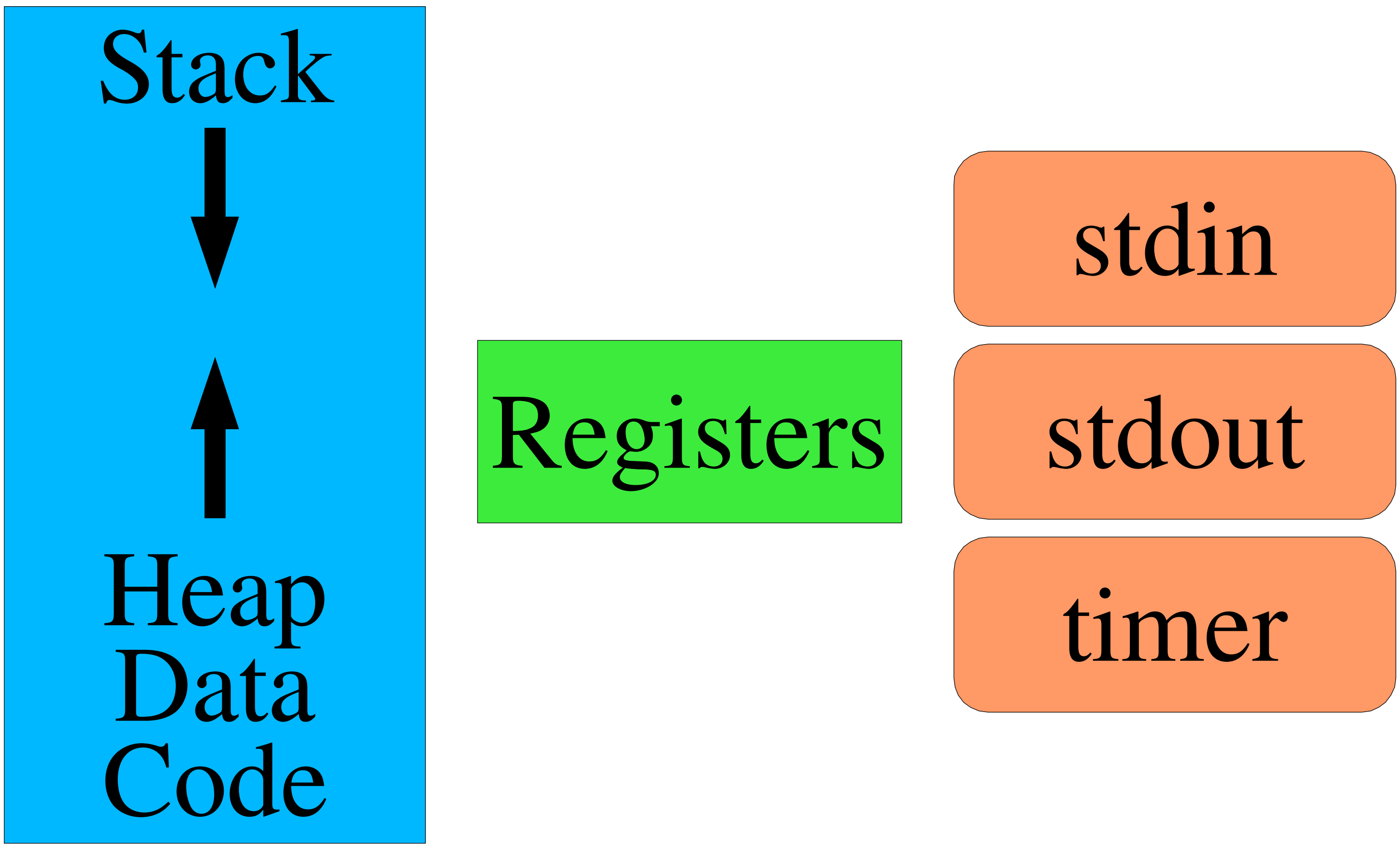
Outline

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

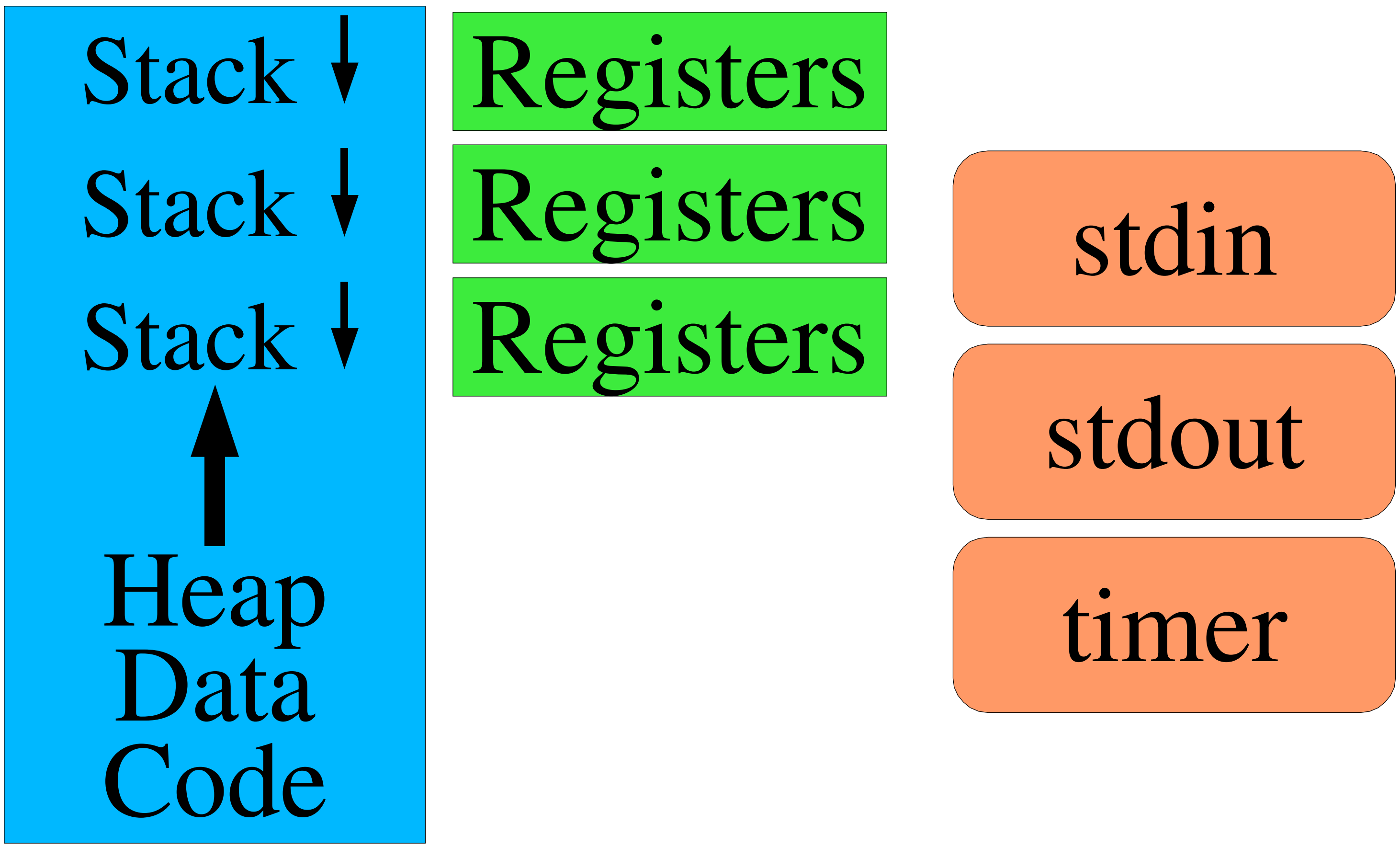
Outline

- Thread = schedulable registers
  - (that's ***all*** there is)
- Why threads?
- Thread flavors (ratios)
- (Against) cancellation
- Race conditions
  - 1 simple, 1 ouch

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
- Server for a multi-player game
  - Many players
  - Access (& update) shared world state
    - Scan multiple objects
    - Update one or two objects

Why threads?

- 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*

Why threads?

- ***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
    - Check that button-release happens in same area
  - ...without JPEG decompressor understanding clicks

Multiprocessor speedup

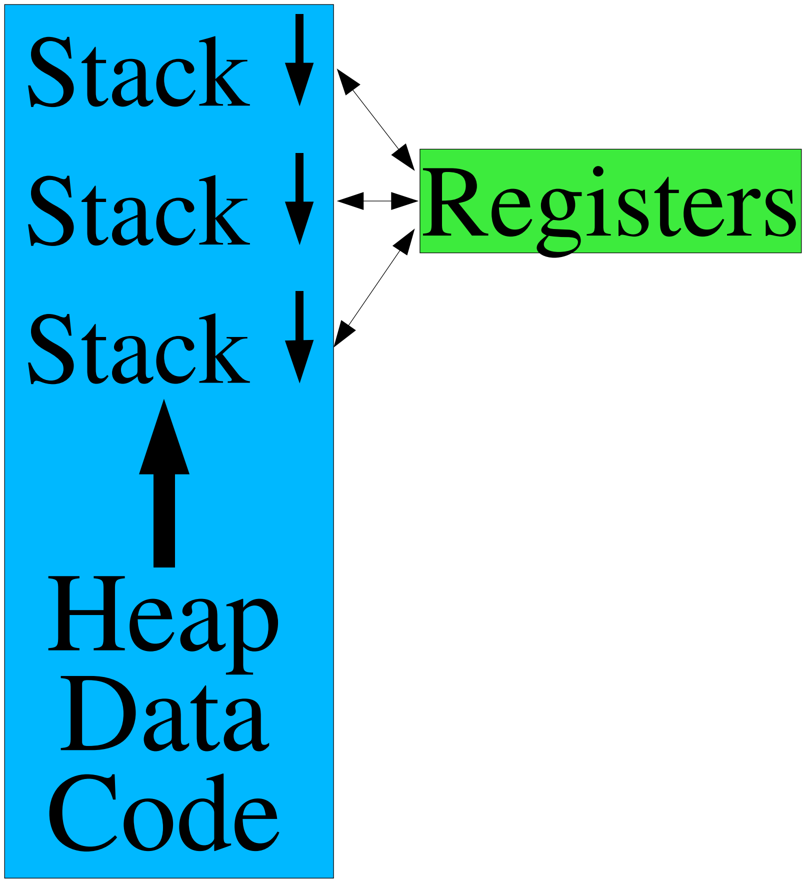
- More CPUs can't help a single-threaded process!
- PhotoShop color dither operation
  - Divide image into regions
  - One dither thread per CPU
  - Can (sometimes) get linear speedup

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

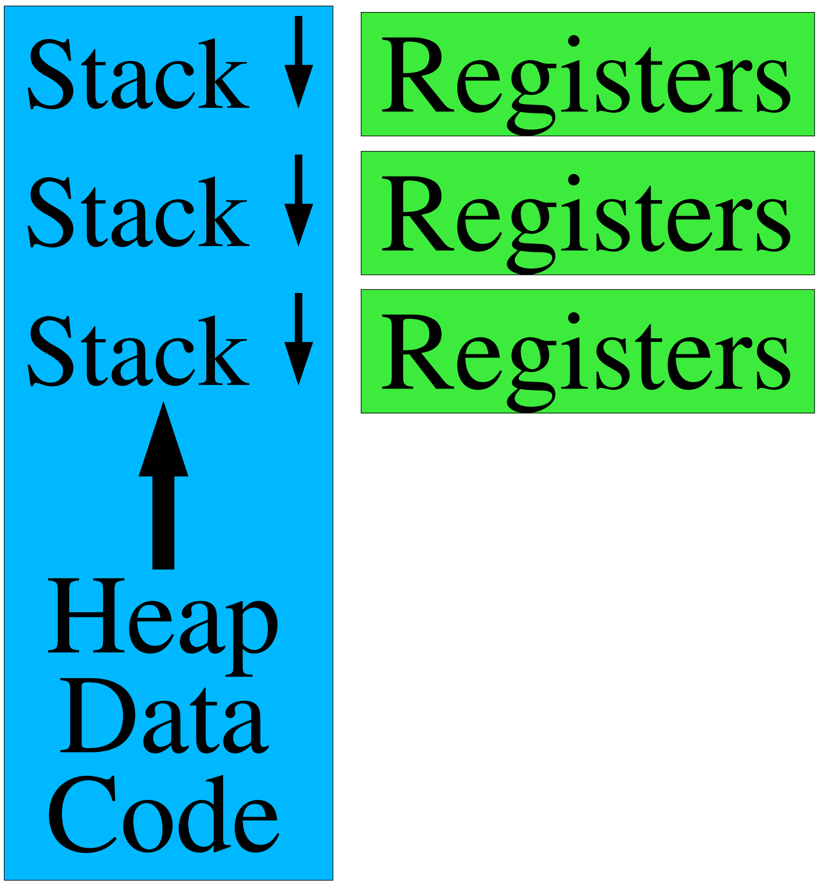


User-space threads (N:1)

- No change to operating system
- System call may block all “threads”
  - 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

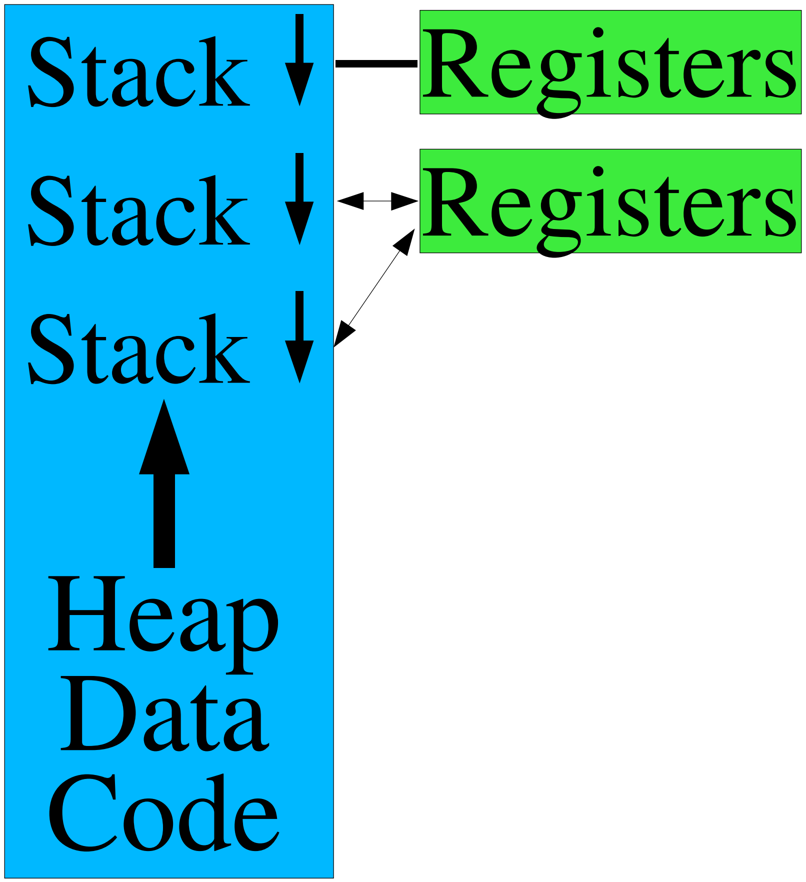


Pure kernel threads (1:1)

- Every thread is sacred
  - Kernel-managed register set
  - Kernel stack
  - “Real” (timer-triggered) scheduling
- Features
  - Program runs faster on multiprocessor
  - User-space libraries must be rewritten
  - Require kernel memory (PCB, stack)

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
    - 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)

Race conditions

- 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 may ***arbitrarily interleave*** execution
- It will choose the ***most painful*** way
  - "Once chance in a million" happens every minute

Race condition example

T0	*T1*
`ticket = temp = next_ticket;`	
	`ticket = temp = next_ticket;`
`++temp;`	
	`++temp;`
`next_ticket = temp;`	
	`next_ticket = temp;`

Effect: temp += 1; ***/* not 2 */***

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
```

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 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
    - file owner's identity

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

Race condition example

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

Thread-specific Data

- 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 is %s\n",
  thread_var(0));
```

- Need a ***little*** anti-sharing

Thread-specific Data

- thread_id() = system call?
  - too expensive!
- Simple C routine?

```
int thread_id(void) {
  extern int thread_id;
  return (thread_id);
}
```

  - shared memory: all int's have same value
- Think about what's ***not*** shared...

TSD: Reserved register

- Many microprocessors have 32 user registers
  - Devote one to thread data pointer
  - 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 stack address***
    - "exercise for the reader"