

15-410

“...process_switch(P2) 'takes a while'...”

Yield
Sep. 23, 2013

Dave Eckhardt

Todd Mowry

Synchronization

Thread library due tonight

- Please follow hand-in procedure on Projects page

Synchronization

Thread library due tonight

- Just kidding!

Who has...

- ...read handouts?
- ...unpacked tarball?
- ...issued a system call?
- ...drawn stack pictures?
- ...had a thread killed due to a page fault?
- ...set up your mail client to alert you to .announce posts?

Synchronization

We hope you use the milestones and attack plan

- Pitfalls exist and we hope to steer you away
- Please don't wait “just one week” to find out when the milestones are...
 - ...if you haven't read the handouts yet, please do so *today*.

Take advantage of course staff

- If you see me I may require you to draw pictures
- Because this is very likely to help you

Review material if necessary

- We genuinely expect you to be operating from the “Questions” and “Debugging” lecture material
- If you missed class or were late, please review them as necessary

Outline

Context switch

- Motivated by `yield()`
- This is a *core idea* of this class
 - You will benefit if your P3 context switch is clean and solid
 - There's more than one way to do it
 - Even more than one *good* way
 - As with P2 `thread_fork`, part of the design is figuring out what parameters `context_switch()` should take...
- This lecture is “early”
 - Struggle with it today
 - Hopefully it'll be easier when you struggle with it in P3
- Note: today we'll talk about every kind of thread *but* P2

Mysterious yield()

```
T1() {  
    while (1)  
        yield(T2);  
}
```

```
T2() {  
    while (1)  
        yield(T1);  
}
```

User-space Yield

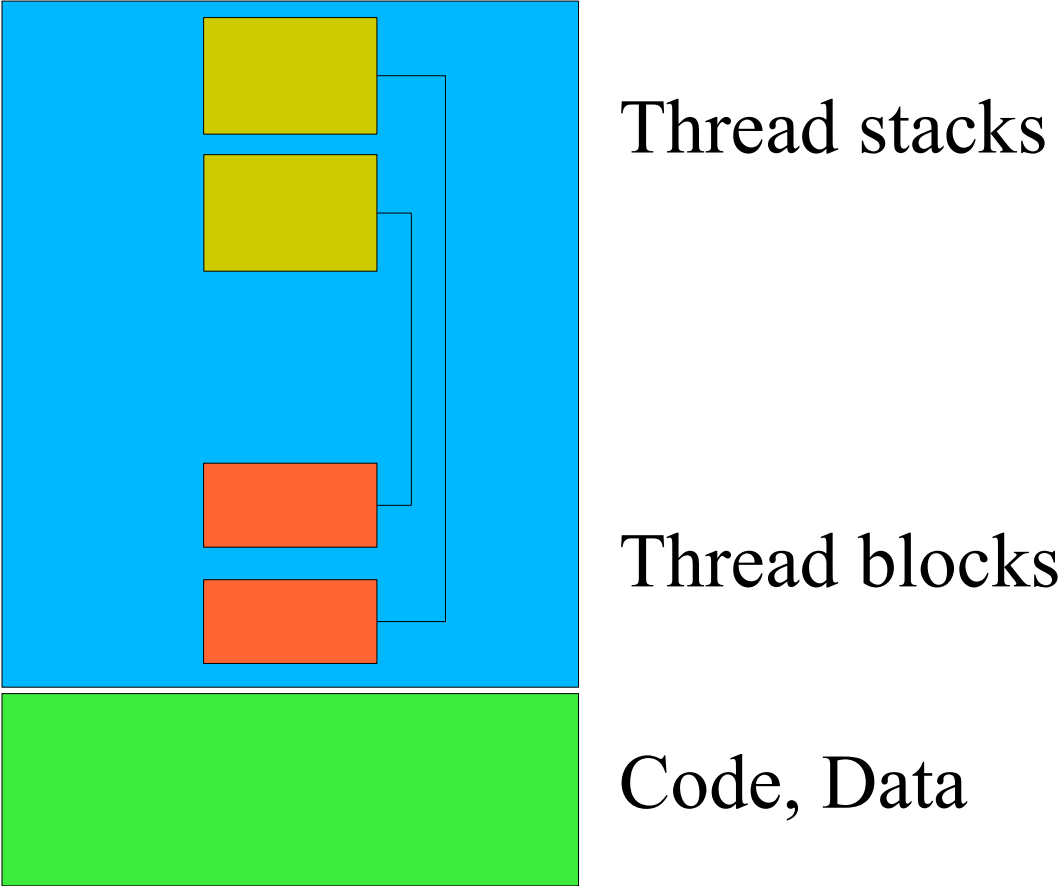
Consider *pure user-space threads*

- You implement threads inside a single-threaded process
- There is no `thread_fork...`
- The opposite of Project 2

What is a thread in that world?

- A stack
- “Thread control block” (TCB)
 - Locator for register-save area
 - Housekeeping information

Big Picture



User-space Yield

yield(user-thread-3)

save my registers on stack

/ magic happens here */*

restore thread 3's registers from thread 3's stack

return; */* to thread 3! */*

Todo List

Save

- General-purpose registers
 - (floating-point registers: omitted)
- Stack pointer
- Program counter

Which value to save for each?

- The value we want the register to have after restore is done

Restore

- Same list as “save”
- Not *our* values: the *target's* values

No magic!

```
/* C+asm() for slide notation only! */  
yield(user-thread-3){  
    save registers on stack    /* asm(...) */  
    tcb->sp = get_esp();        /* asm(...) */  
    tcb->pc = &there;          /* gcc ext. */  
    tcb = findtcb(user-thread-3);  
    set_esp(tcb->sp);         /* asm(...) */  
    jump(tcb->pc);            /* asm(...) */  
there:  
    restore registers from stack /* asm() */  
    return;  
}
```

The Program Counter

What values can the PC (%eip) contain?

- In a pure user-thread environment, thread switch happens *only in yield()*
- Yield sets saved PC to address of first “restore registers” instruction

All non-running threads have the *same* saved PC

- Please make sure this makes sense to you

Remove Unnecessary Code – 1

```
yield(user-thread-3) {
    save registers on stack
    tcb->sp = get_esp();
tcb->pc = &there;
    tcb = findtcb(user-thread-3);
    set_esp(tcb->sp);
    jump(tcb->pc &there);
there:
    restore registers from stack
    return
}
```

Remove Unnecessary Code – 2

```
yield(user-thread-3) {  
    save registers on stack  
    tcb->sp = get_esp();  
tcb->pc = &there;  
    tcb = findtcb(user-thread-3);  
    set_esp(tcb->sp);  
jump(tcb->pc &there);  
there:  
    restore registers from stack  
    return  
}
```

Remove Unnecessary Code – 3

```
yield(user-thread-3) {  
    save registers on stack  
    tcb->sp = get_esp();  
    tcb = findtcb(user-thread-3);  
    set_esp(tcb->sp);  
    restore registers from stack  
    return  
}
```

User Threads vs. Kernel Processes

What if a *process* yields to another?

- “Compare & contrast, in no more than 1,000 words...”

User threads

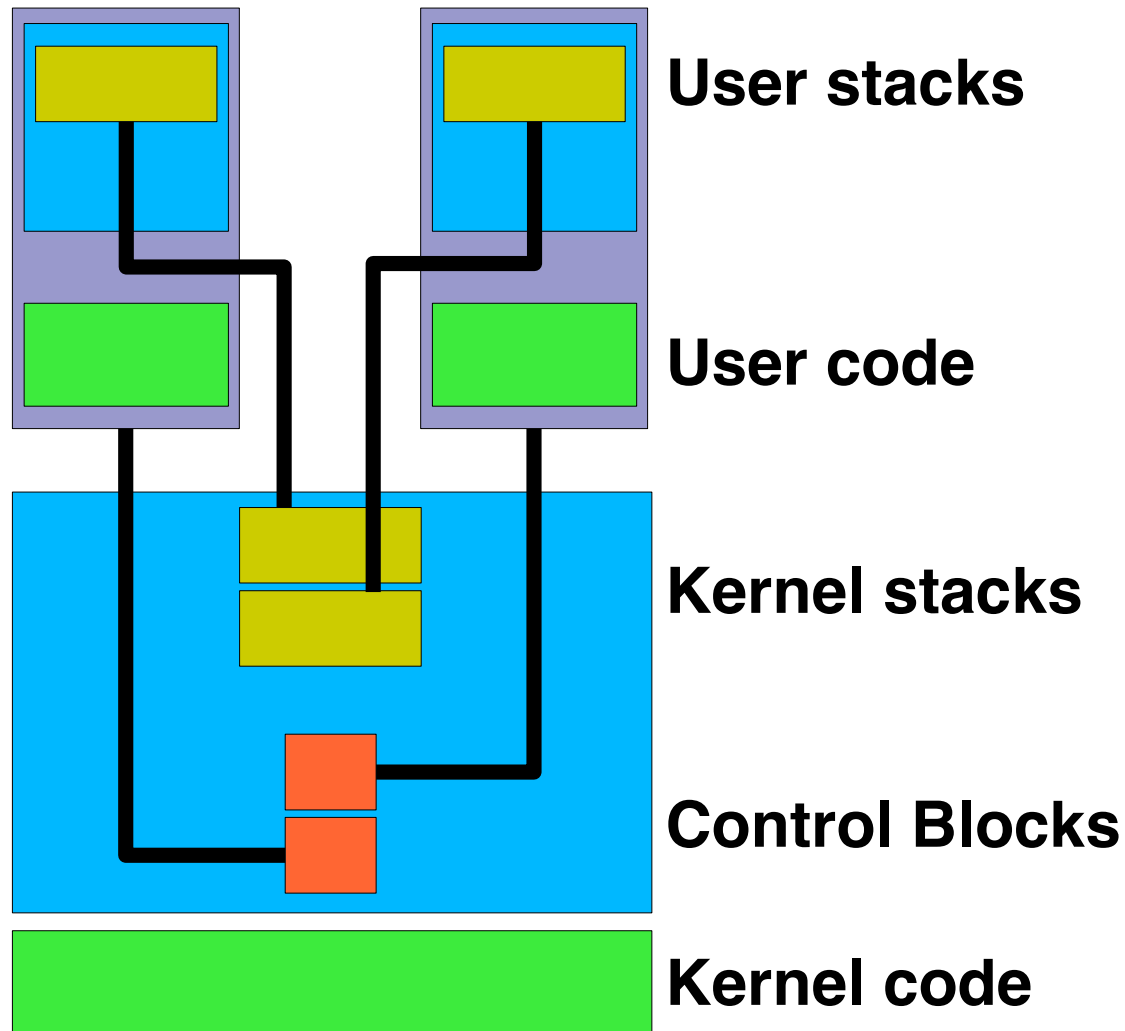
- Share memory
- Threads not protected from each other

Processes

- Do *not* generally share memory
- P1 must *not* modify P2's saved registers

Where are process save areas and control blocks?

Kernel Memory Picture



P1's Yield(P2) steps

P1 calls yield(P2)

Syscall stub: INT 50 ⇒ *boom!*

Processor trap protocol

- Saves some registers on P1's kernel stack
 - This is a *stack switch* (user ⇒ kernel), intel-sys.pdf 5.10
 - Top-of-kernel-stack specified by %esp0
 - Trap frame (x86): %ss & %esp, %eflags, %cs & %eip

Assembly-language wrapper

- Saves more registers
- Starts C trap handler

Then...?

P1's Yield(P2) steps

```
int sys_yield(int pid) {  
    return (process_switch(pid));  
}
```

Assembly-language wrapper

- Restores registers from P1's kernel stack, modulo %eax

Processor return-from-trap protocol (aka IRET)

- Restores %ss & %esp, %eflags, %cs & %eip

INT 50 instruction “completes”

- Back in user-space

P1 yield() library routine returns

What happened to P2??

process_switch(P2) “takes a while”

- When P1 calls it, it “returns” to P2
- When P2 calls it, it “returns” to P1 (eventually)

Inside process_switch()

ATOMICALLY

```
enqueue_tail(runqueue, cur_pcb);  
save registers      /* P1's stack */  
cur_pcb = dequeueID(runqueue, P2);  
stackpointer = cur_pcb->sp;  
restore registers /* P2's stack */  
return;  
  
/* some details omitted */
```

User-mode Yield vs. Kernel-mode

Kernel context switches happen for more reasons

- good old yield(), but also...
- Message passing from P1 to P2
- P1 blocked on disk I/O, so run P2
- *CPU preemption by clock interrupt*

I/O completion Example

P1 calls read()

In kernel

- read() starts disk read
- read() calls `condition_wait(&buffer); /* details vary */`
- `condition_wait()` calls `process_switch()`
 - In general, we *want* somebody else to run
- `process_switch()` returns *to P2*

I/O Completion Example

While P2 is running

- Disk completes read, interrupts P2 into kernel
- Interrupt handler calls `condition_signal(&buffer);`

Now what?

I/O Completion Example

While P2 is running

- Disk completes read, interrupts P2 into kernel
- Interrupt handler calls `condition_signal(&buffer);`

Option 1

- `condition_signal()` marks P1 as runnable, returns
- Interrupt handler returns to P2

I/O Completion Example

While P2 is running

- Disk completes read, interrupts P2 into kernel
- Interrupt handler calls `condition_signal(&buffer);`

Option 1

- `condition_signal()` marks P1 as runnable, returns
- Interrupt handler returns to P2

Option 2

- `condition_signal()` calls `process_switch(P1)` (“only fair”)
- P2 will finish the interrupt handler *much later*
 - Remember in P3 to confront implications of this!

Clock interrupts

P1 doesn't “ask for” clock interrupt

- Clock handler *forces* P1 into the kernel
 - Kernel stack looks like a “system call”
 - As if user process had called `handle_timer()`
 - But it was involuntary

P1 doesn't say who to yield to

- (it didn't make the “system call”)
- *Scheduler* chooses next process

Summary

Similar steps for user space, kernel space

Primary differences

- Kernel has open-ended competitive scheduler
- Kernel more interrupt-driven

Implications for 410 projects

- P2: firmly understand thread stacks
 - `thread_create()` stack setup
 - cleanup
 - race conditions
- P3: firmly understand kernel context switch

Advice: draw pictures of stacks