

15-410

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

Yield
Sep. 27, 2004

Dave Eckhardt

Bruce Maggs

Outline

Project 2 Q&A

Context switch

- Motivated by `yield()`
- This is a *core idea* of this class

Mysterious yield()

```
process1() {  
    while (1)  
        yield(P2);  
}
```

```
process2() {  
    while (1)  
        yield(P1);  
}
```

User-space Yield

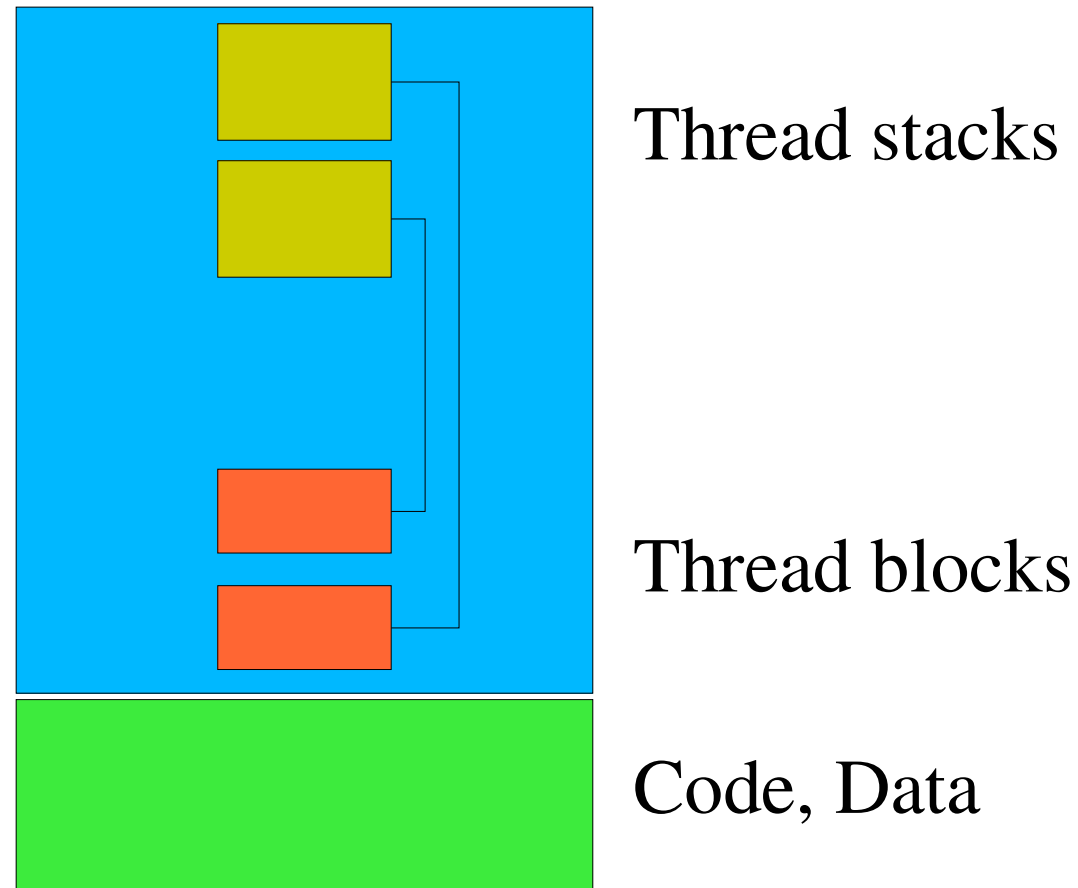
Consider *pure user-space threads*

- The opposite of Project 2

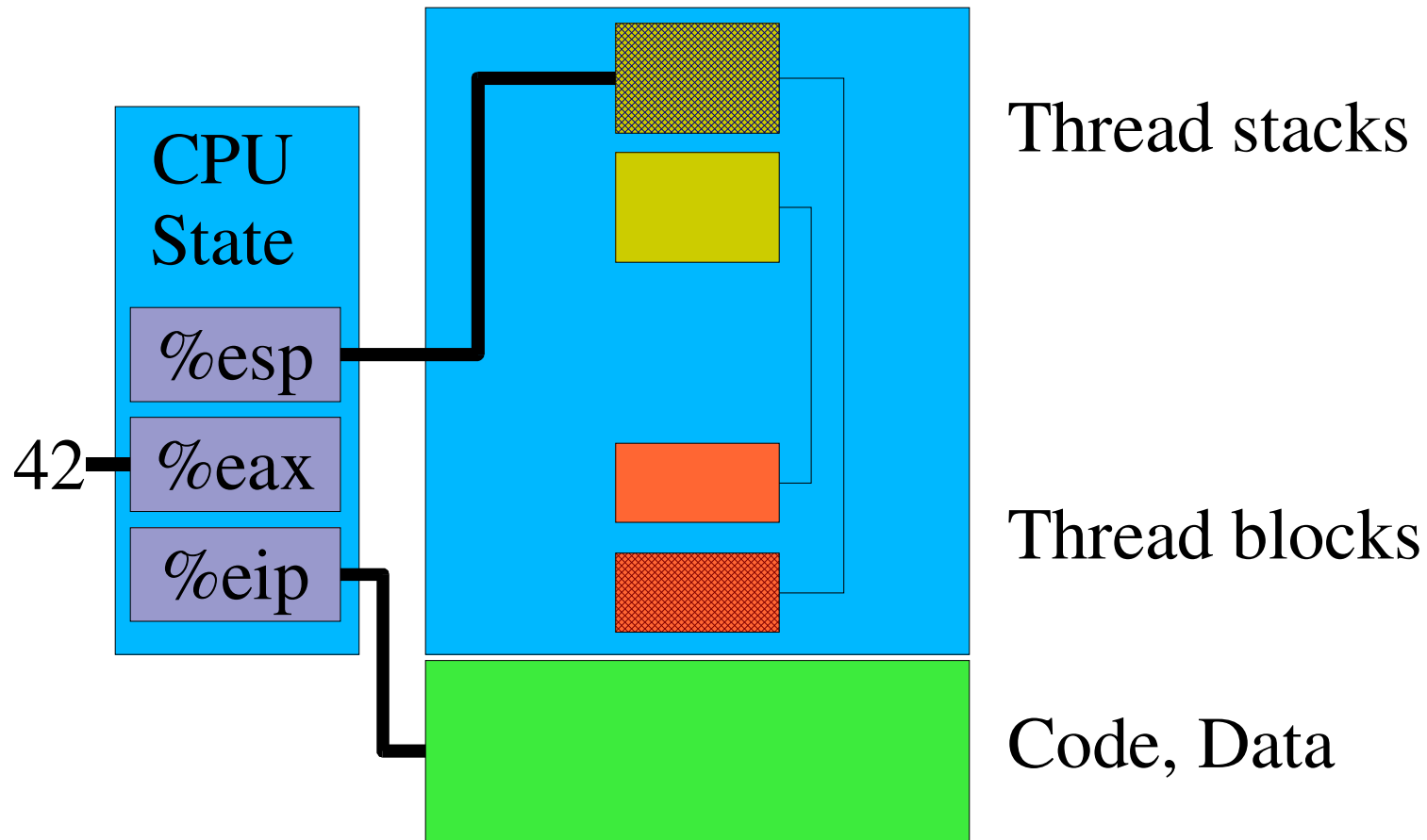
What is a thread?

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

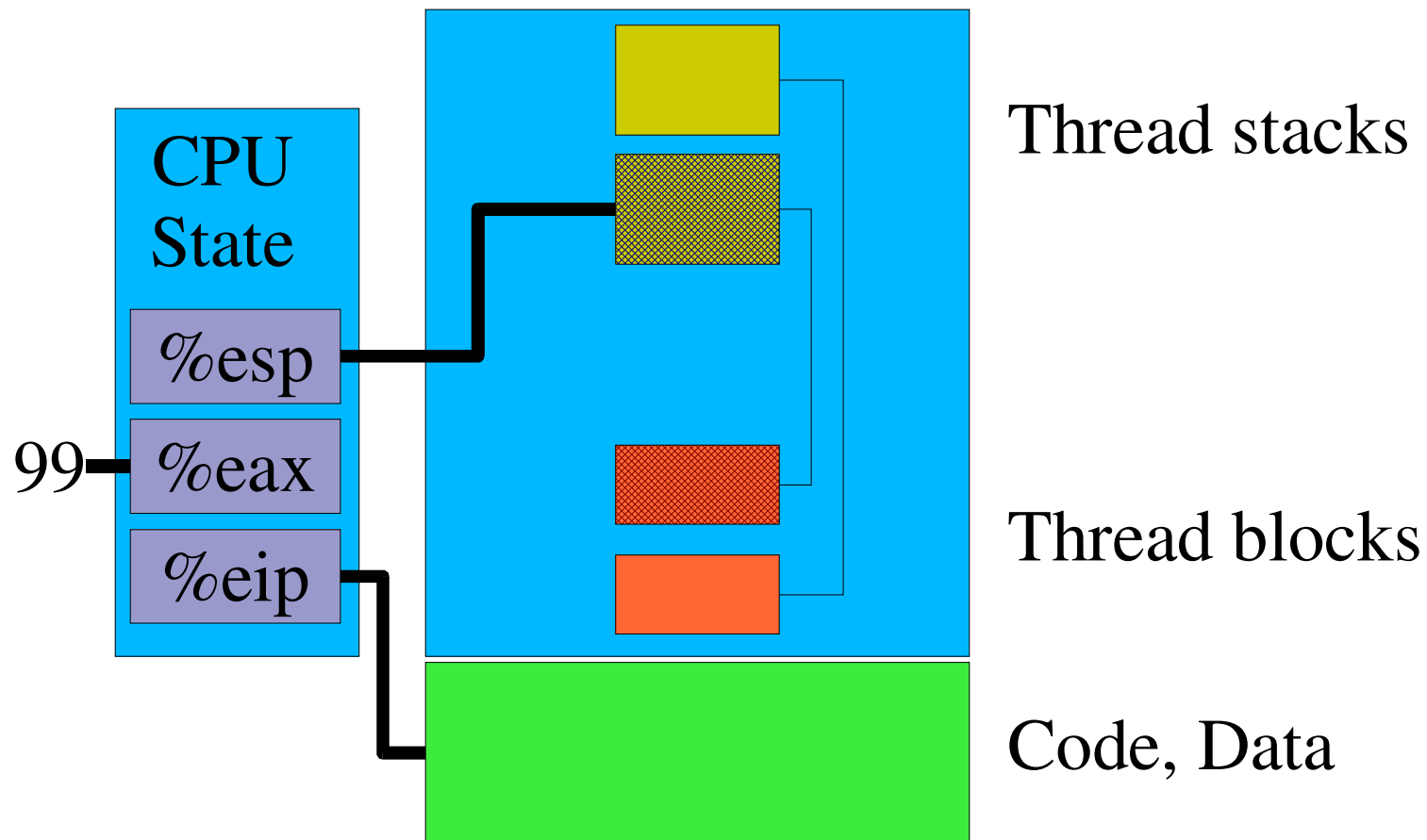
Big Picture



Big Picture



Running the Other Thread



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

General-purpose registers

Stack pointer

Program counter

No magic!

yield(user-thread-3)

```
    save registers on stack    /* asm(...) */
    tcb->sp = get_esp();        /* asm(...) */
    tcb->pc = &there;
    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?

- Thread switch happens *only in yield*
- Yield sets saved PC to start of “restore registers”

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

- Please make sure this makes sense to you

Remove Unnecessary Code

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(there);
```

there:

restore *registers* from stack

return

Remove Unnecessary Code

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

User threads

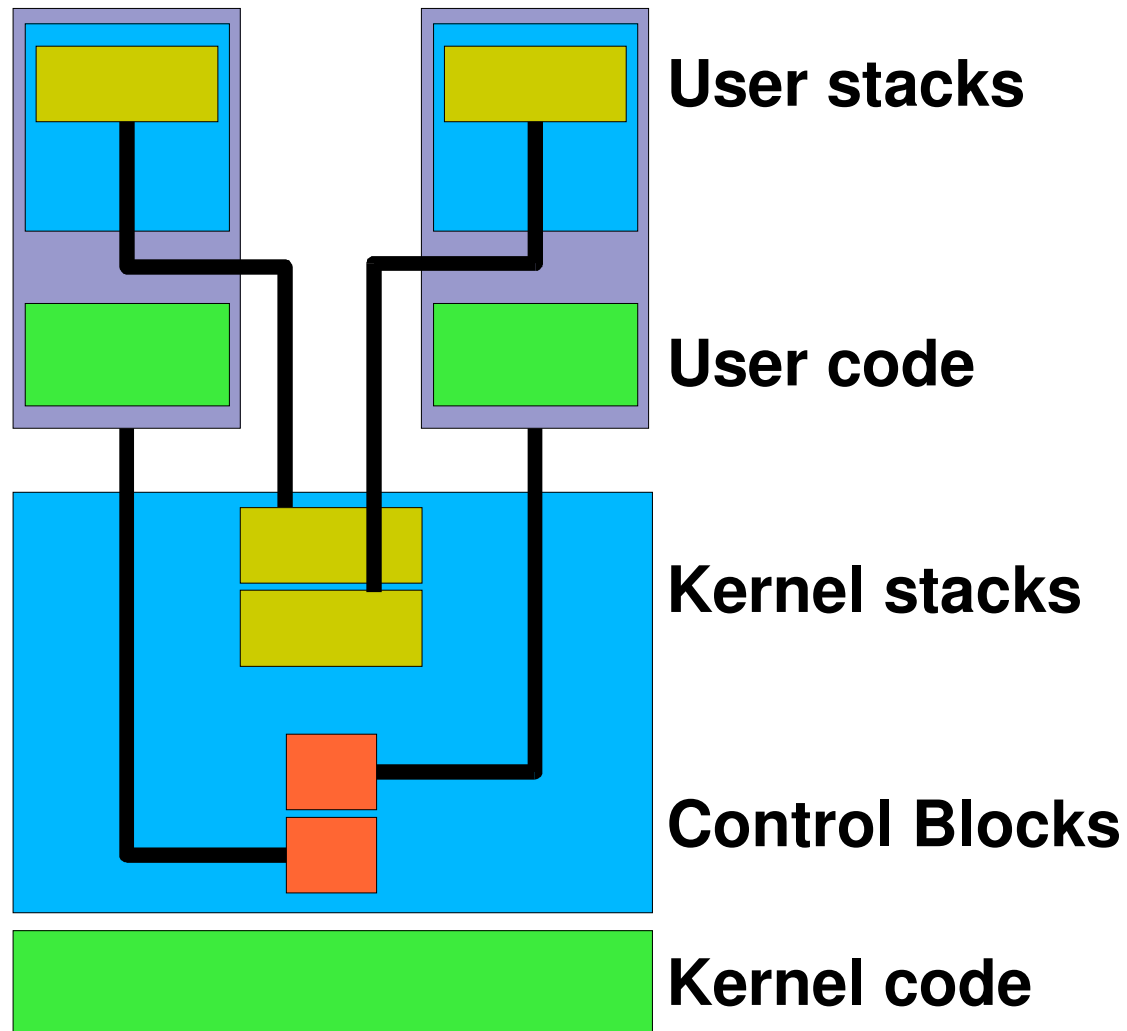
- Share memory
- Threads not protected from each other

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



Yield steps

P1 calls yield(P2)

INT 50 $\Rightarrow$ *boom!*

Processor trap protocol

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

Assembly-language stub

- Saves more registers
- Starts C trap handler

Yield steps

handle_yield()

- `return(process_switch(P2))`

Assembly-language stub

- Restores registers from P1's kernel stack

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 = dequeue(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

- `yield()`
- Message passing from P1 to P2
- P1 sleeping 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()
- 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);`

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 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