CS 423
Operating System Design: The Context Switch

Professor Adam Bates
Spring 2016
Goals for Today

- **Learning Objective:**
  - Gain an understanding of how (and under what circumstances) a processor switches between tasks

- **Announcements, etc:**
  - HW1 will be out on Compass at end of class (Due 2/1)
  - Bring your ideas for special topics to class on Friday

**Reminder:** Please put away devices at the start of class
TA Office Hours (0207 SC)

Bo Teng

Saad Hussain

Yisong Yue

Tuesdays 2:00 - 3:00

Wednesdays 11:00 - 12:00

Thursdays 4:00 - 5:00
Threads provide an illusion of concurrency
  - Given one “real CPU”, give the illusion of multiple “virtual CPUs” each dedicated to a thread
  - The “virtual CPUs” transparently share the “real CPU”

The abstraction is recursive
  - If the virtual CPU abstraction is perfect, a “virtual CPU” is indistinguishable from a “real CPU”
  - Therefore, a “virtual CPU” can play the role of a real CPU and can give the illusion of multiple (2\textsuperscript{nd}-level) “virtual CPUs” transparently sharing the 1\textsuperscript{st}-level virtual CPU.
What’s a ‘real’ CPU?

What is the “real” CPU comprised of at any given moment?
What's a 'real' CPU?

What is the "real" CPU comprised of at any given moment?

Diagram: Program instructions
- Code Segment
- Offset
- Program Counter
What's a ‘real’ CPU?

What is the “real” CPU comprised of at any given moment?
What's a ‘real’ CPU?

What is the “real” CPU comprised of at any given moment?
What’s a ‘real’ CPU?

What is the “real” CPU comprised of at any given moment?
What’s a ‘real’ CPU?

What’s the STATE of a real CPU?

- Code Segment
- Data Segment
- Program Counter
- Offset
- OpCode
- Operand
- Current Instruction
- Heap
- Stack Segment
- Offset
- Stack Pointer
- Stack
- Registers
What’s the **STATE** of a real CPU?
The Context Switch

Load State (Context)

Save State (Context)
The state for processes that are not running on the CPU are maintained in the Process Control Block (PCB) data structure.
The Context Switch

Program Counter

Code Segment
Offset
Program Counter
OpCode Operand

Data Segment

Operand

Stack Segment

Stack Pointer

Heap

Data Segment

Operand

OpCode

Stack

Save State (Context)

Load State (Context)

Registers

Program instructions

Code Segment
Offset
Program Counter
OpCode Operand

Data Segment

Operand

Stack Segment

Stack Pointer

Heap

Data Segment

Operand

Stack

Save State (Context)

Load State (Context)

Registers

Program instructions

Code Segment
Offset
Program Counter
OpCode Operand

Data Segment

Operand

Stack Segment

Stack Pointer

Heap

Data Segment

Operand

Stack
The Context Switch

Note: In thread context switches, heap is not switched!

Save State (Context)

Load State (Context)
The Context Switch

Note: In thread context switches, heap is not switched!

Save State (Context)

Load State (Context)
Thread Context Switch

Note: In thread context switches, heap is not switched!

So who does the context switch, and when??

Save State (Context)

Load State (Context)

Global Variables

Local Variables

Program Counter

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Stack Segment

Stack Pointer

OpCode

Operand

OpCode

Operand

So who does the context switch, and when??

Save State (Context)

Load State (Context)

Global Variables

Local Variables

Program Counter

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Stack Segment

Stack Pointer

OpCode

Operand

OpCode

Operand
Thread Context Switch

Note: In **thread** context switches, heap is not switched!

Solution 1: An Interrupt

Save State (Context)

Load State (Context)
CTX Switch: Interrupt

Running Thread

- Program Counter
- Code Segment
- Offset
- Program Counter
- Stack Segment
- Stack Pointer
- Stack Segment
- Stack Pointer
- Program instructions
- Stack
- Registers

- Program Counter
- Code Segment
- Offset
- Program Counter
- Stack Segment
- Stack Pointer
- Stack Segment
- Stack Pointer
- Program instructions
- Stack
- Registers
Interrupt

Save PC on thread stack
Jump to Interrupt handler
**CTX Switch: Interrupt**

Save PC on thread stack
Jump to Interrupt handler

**Handler**
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)

**Thread Control Block**
CTX Switch: Interrupt

- Save PC on thread stack
- Jump to Interrupt handler

**Handler**
- Save thread state in thread control block (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block
CTX Switch: Interrupt

Save PC on thread stack
Jump to Interrupt handler

Handler
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)
Save PC on thread stack
Jump to Interrupt handler

Handler
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)

Where does it return?
CTX Switch: Interrupt

- Save thread state in thread control block (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)

Where does it return?
What are some examples of context switches due to interrupts?
What are some examples of context switches due to interrupts?

- **Clock Interrupt**: Task exceeds its time slice
What are some examples of context switches due to interrupts?

- **Clock Interrupt**: Task exceeds its time slice
- **I/O Interrupt**: Waiting processes may be preempted
What are some examples of context switches due to interrupts?

- **Clock Interrupt**: Task exceeds its time slice
- **I/O Interrupt**: Waiting processes may be preempted
- **Memory Fault**: CPU attempts encounters a virtual memory address that is not in main memory. OS may resume execution of another process while retrieving the block, then moves process to ready state.
What are some examples of context switches due to interrupts?

- **Clock Interrupt**: Task exceeds its time slice
- **I/O Interrupt**: Waiting processes may be preempted
- **Memory Fault**: CPU attempts encounters a virtual memory address that is not in main memory. OS may resume execution of another process while retrieving the block, then moves process to ready state.
Thread Context Switch

Solution 2: Voluntary yield()

Note: In thread context switches, heap is not switched!
CTX Switch: Yield

Running Thread

- Code Segment
- Offset
- Program Counter
- Stack Segment
- Stack Pointer
- Stack
- Program instructions
- Registers

- Code Segment
- Offset
- Program Counter
- Stack Segment
- Stack Pointer
- Stack
- Program instructions
- Registers
yield()

Save PC on thread stack
Jump to yield() function
CTX Switch: Yield

Save PC on thread stack
Jump to yield() function

yield()
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)

Thread Control Block
Save PC on thread stack
Jump to yield() function

yield()
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)
- Choose next thread
Save PC on thread stack
Jump to yield() function

yield()
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block

Thread Control Block
Save PC on thread stack
Jump to yield() function

yield()
- Save thread state in thread control block
  (SP, registers, segment pointers, ...)
- Choose next thread
- Load thread state from control block
- Pop PC from thread stack (return from handler)
CTX Switch: Yield

Save PC on thread stack
Jump to yield() function

yield()
- Choose next thread
- swapcontext()
Scheduler

Where is the Scheduling Policy?

- Save PC on thread stack
- Jump to yield() function
- yield()
  - Choose next thread
  - swapcontext()
Scheduler

Where is the Scheduling Policy?

Maintains a sorted queue of ready threads

Save PC on thread stack
Jump to yield() function

Thread Control Block

yield()
- NextThreadID = scheduler()
- swapcontext()

Thread Control Block
Issues

- Initialization?
Issues

- Initialization?
  - Stack for each thread needs to be allocated a priori → just malloc() the right sized chunk.
  - Thread control blocks must be created
• **Initialization?**
  - Stack for each thread needs to be allocated a priori → just `malloc()` the right sized chunk.
  - Thread control blocks must be created

• **Interrupt context switch versus yield?**
• Initialization?
  • Stack for each thread needs to be allocated a priori
    → just malloc() the right sized chunk.
  • Thread control blocks must be created

• Interrupt context switch versus yield?
  • Yield is voluntary. What if application never yields?
  • Yield cannot do preemptive scheduling.
User-Level Thread Pkg’s

- Initialization?
  - Stack for each thread needs to be allocated a priori
    → just malloc() the right sized chunk.
  - Thread control blocks must be created

- Interrupt context switch versus yield?
  - Yield is voluntary. What if application did not yield?
  - Yield cannot do preemptive scheduling.
User-Level Thread Pkg’s

- Initialization?
  - Stack for each thread needs to be allocated a priori → just malloc() the right sized chunk.
  - Thread control blocks must be created

- Interrupt context switch versus yield?
  - Yield is voluntary. What if application did not yield?
  - Yield cannot do preemptive scheduling.

API

- `thread_create()`
- `thread_yield()`

Not seen by user

`scheduler()`
Review

- How do you implement a context switch?
  - In cooperative user-level threads?
  - In kernel threads?
- How do you implement a thread scheduler?
- Trade-offs between cooperative and non-cooperative thread models?
Goals for Today

• **Learning Objective:**
  • Gain an understanding of how (and under what circumstances) a processor switches between tasks

• **Announcements, etc:**
  • HW1 will be out on Compass at end of class (Due 2/1)
  • Bring your ideas for special topics to class on Friday
Here’s a more accurate representation of an active process’ layout in memory:

![Diagram of process layout in memory]

- **High address**: Command-line arguments and environment variables
- **Low address**: Program instructions
- **Stack**
- **Heap**
- **Uninitialized data (bss)**: Initialized to zero by exec
- **Initialized data**: Read from program file by exec
- **Text**