Learning Objectives:
• Understand the Kernel/Process Abstraction
• Gain a working knowledge of Mode/Context switches

Announcements:
• C4 template available on Compass! *Due Jan 26*
• MP0 is available on Compass! *Due Jan 29*
• HW0 is available on Compass! *Due Jan 29*

Reminder: Please put away devices at the start of class
Process concept

- A process is the OS abstraction for executing a program with limited privileges

Dual-mode operation: user vs. kernel

- Kernel-mode: execute with complete privileges
- User-mode: execute with fewer privileges

Safe control transfer

- How do we switch from one mode to the other?
Process: an **instance** of a program that runs with limited rights on the machine

- Thread: a sequence of instructions within a process
  - Potentially many threads per process (for now, assume 1:1)
- Address space: set of rights of a process
  - Memory that the process can access
  - Other permissions the process has (e.g., which system calls it can make, what files it can access)
How can we permit a process to execute with only limited privileges?
How can we implement execution with limited privilege?

• Execute each program instruction in a simulator
• If the instruction is permitted, do the instruction
• Otherwise, stop the process
• Basic model in Javascript and other interpreted languages

Ok... but how do we go faster?

• Run the unprivileged code directly on the CPU!
How can we implement execution with limited privilege?

- Execute each program instruction in a simulator
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript and other interpreted languages

Ok... but how do we go faster?
How can we implement execution with limited privilege?

- Execute each program instruction in a simulator
- If the instruction is permitted, do the instruction
- Otherwise, stop the process
- Basic model in Javascript and other interpreted languages

Ok... but how do we go faster?

- Run the unprivileged code directly on the CPU!
A Model of a CPU

Branch Address

Select PC

New PC

Program Counter

CPU Instructions Fetch and Execute

+4

opcode
A CPU with Dual-Mode Operation

Branch Address

+4

Handler PC

Select PC

New PC

Program Counter

CPU Instructions Fetch and Execute

Select Mode

New Mode

Mode

opcode
Privileged instructions
  • Available to kernel
  • Not available to user code

Limits on memory accesses
  • To prevent user code from overwriting the kernel

Timer
  • To regain control from a user program in a loop

Safe way to switch from user mode to kernel mode, and vice versa
Examples?

What should happen if a user program attempts to execute a privileged instruction?
How/when do we switch from user to kernel mode?

1. Interrupts
   • Triggered by timer and I/O devices

2. Exceptions
   • Triggered by unexpected program behavior
   • Or malicious behavior!

3. System calls (aka protected procedure call)
   • Request by program for kernel to do some operation on its behalf
   • Only limited # of very carefully coded entry points
How does the OS know when a process is in an infinite loop?
Hardware device that periodically interrupts the processor

- Returns control to the kernel handler
- Interrupt frequency set by the kernel
  Not by user code!
- Interrupts can be temporarily deferred
  Not by user code!

Interrupt deferral crucial for implementing mutual exclusion
How/when do we switch from kernel to user mode?

1. New process/new thread start
   • Jump to first instruction in program/thread
2. Return from interrupt, exception, system call
   • Resume suspended execution (return to PC)
3. Process/thread context switch
   • Resume some other process (return to PC)
4. User-level upcall (UNIX signal)
   • Asynchronous notification to user program
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?

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?
What’s a ‘real’ CPU?

What’s the **STATE** of a real CPU?
The Context Switch

Save State (Context)

Load 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

Load State (Context)

Save State (Context)
The Context Switch

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

Load State (Context)

Save 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???
Solution 1: An Interrupt
CTX Switch: Interrupt

Running Thread

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

Registers

Code Segment
Offset
Program Counter
Stack Segment
Stack Pointer
Program instructions
Stack
Interrupt
Save PC on thread stack
Jump to Interrupt handler
Save PC on thread stack
Jump to Interrupt handler

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

Thread Control Block
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
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?
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?
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

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

Solution 2: Voluntary yield()
CTX Switch: Yield

Running Thread

Program instructions

Code Segment
Offset
Program Counter

Stack Segment
Stack Pointer

Stack

Program instructions

Code Segment
Offset
Program Counter

Stack Segment
Stack Pointer

Stack

Registers
yield()

Save PC on thread stack
Jump to yield() function
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
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)
Save PC on thread stack
Jump to yield() function

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

Program Counter

Code Segment

Offset

Stack Segment

Stack Pointer

Stack

Program instructions

Save PC on thread stack
Jump to yield() function

Thread Control Block

yield()

- NextThreadID = scheduler()

- swapcontext()

Maintains a sorted queue of ready threads

Thread Control Block

CS 423: Operating Systems Design