CS 423
Operating System Design: The Kernel Abstraction

Tianyin Xu

* Thanks for Prof. Adam Bates for the slides.
Let’s start with some questions.
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 Abstraction

**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?
Thought Experiment

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
Thought Experiment

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?
Thought Experiment

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

opcode
A CPU with Dual-Mode Operation

Branch Address

Selector PC

New PC

Program Counter

CPU Instructions Fetch and Execute

New Mode

Mode

opcode

Handler PC

+4
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 Timer

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
Kernel -> User Switches

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 is the CPU’s behavior defined by at any given moment?
What is the CPU’s behavior defined by at any given moment?
What is the CPU’s behavior defined by at any given moment?
CPU State

What is the CPU’s behavior defined by at any given moment?

- Program instructions
  - Code Segment
    - OFFSET
  - Program Counter
    - OpCode, Operand
  - Current Instruction
- Data Segment
  - OFFSET
  - Data Operand
- Heap
- Stack Segment
  - Offset
  - Stack Pointer
- Stack
What is the CPU’s behavior defined by at any given moment?

CPU State

Registers

Program instructions

- Code Segment
- Offset
- Program Counter

- OpCode
- Operand

Current Instruction

Data Segment

Offset

Data Operand

Heap

Stack Segment

Offset

Stack Pointer

Stack
CPU State

What defines the **STATE** of the CPU?

- Program Counter
  - Code Segment
  - Offset
  - Program

- Current Instruction
  - OpCode
  - Operand

- Data Segment
  - Offset
  - Data Operand

- Heap

- Stack Segment
  - Offset
  - Stack Pointer

- Stack

- Registers

What defines the **STATE** of the CPU?
What's a 'real' CPU?

What's the **STATE** of a real CPU?

- **Program Counter**
  - Current Instruction
  - Offset
- **Code Segment**
- **Data Segment**
  - Data Operand
  - Offset
- **Operand**
- **Heap**
- **Stack Segment**
  - Offset
  - Stack Pointer
- **Stack**
- **Registers**

Program instructions
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

Save State (Context)

Load State (Context)
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!

Program Counter

Code Segment
Offset
Program Counter

Stack Segment

Stack Pointer

Stack

Data Segment

Heap

Operand

Data Segment

Operand

Stack Segment

Stack Pointer

Stack

Program Counter

Code Segment
Offset
Program Counter

Operand

Stack Segment

Stack Pointer

Registers

Global Variables

Load State (Context)

Local Variables

Save State (Context)

Registers

Stack Segment

Stack Pointer

Stack
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)

Program Counter

Code Segment Offset

Program Counter

Code Segment Offset

Stack Segment

Stack Pointer

Stack Segment

Stack Pointer

Data Segment

Data Segment

Operands

Stack

Stack

Save State (Context)

Load State (Context)

Registers

Local Variables

Global Variables

OpCode

Operand

OpCode

Operand

So who does the context switch, and when???
Thread Context Switch

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

**Solution 1: An Interrupt**

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Registers

Local Variables

Global Variables

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Solution 1:

An Interrupt

Save State (Context)

Load State (Context)

Registers

Global Variables

Local Variables

Program instructions

Code Segment

Offset

Program Counter

Stack Segment

Stack Pointer

Data Segment

Data Operand

Heap
CTX Switch: Interrupt

Running Thread

Program instructions

Code Segment
Offset

Program Counter

Stack Segment

Stack Pointer

Stack Segment

Stack

Program instructions

Code Segment
Offset

Program Counter

Stack Segment

Stack Pointer

Stack

Registers

Registers
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
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

Program Counter
Code Segment
Offset
Stack Segment
Stack Pointer
Stack Segment
Stack

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)

Thread Control Block
Registers

Thread Control Block
Registers

Thread Control Block
Registers

Stack Segment
Stack Pointer
Stack Segment
Stack

Program Counter
Code Segment
Offset
Program Counter
Code Segment
Offset
Program Counter
Code Segment
Offset
Program Counter
Code Segment
Offset
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)

Where does it return?
CTX Switch: Interrupt

Program Counter

Code Segment
Offset

Stack Segment

Stack Pointer

Stack Segment

Stack Pointer

Program instructions

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
- 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 to access 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

Registers

Program instructions

Code Segment Offset

Program Counter

Stack Segment

Stack Pointer

Stack

Registers
CTX Switch: Yield

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, …)
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
- Choose next thread
- Load thread state from control block
CTX Switch: Yield

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()

Thread Control Block

Thread Control Block
Scheduler

Where is the Scheduling Policy?

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

Maintains a sorted queue of ready threads

Thread Control Block

Save PC on thread stack
Jump to yield() function

Program instructions

Stack Segment

Stack Pointer

Program Counter

Code Segment

Offset

Registers

Stack

Program instructions

Stack Segment

Stack Pointer

Program Counter

Code Segment

Offset

Registers

Stack