Lecture 2: Multiprogramming and Dual Mode Operation

23 June 2015
Charles Reiss
http://cs162.eecs.berkeley.edu/
Recall: What is an OS

Referee
- Resource sharing, protection, isolation

Illusionist
- Clean, easy abstractions

Glue: Common services
- Storage
- Window systems
- Authorization
- Networking
Recall: Virtual Machines

Definition: Software emulation of abstract machine
- Programs believe they own the machine
- Simulated "hardware" has the features we want

Two types:
- Process VM: run a single program in the VM
  - Example: Typical OS
- System VM: run entire OS + its applications in the VM
  - Example: Virtualbox, VMWare, Parallels, Xen, ...
Recall: Dual Mode Operation

**Hardware** provides at least two modes:
- "**Kernel**" (or "supervisor" or "protected")
  - Unix "kernel" runs here (part of OS running all the time)
- "**User**" mode
  - Normal programs run here
  - Even if "administrator" or "superuser" or "sudo"

Some operations **prohibited** in user mode
- e.g. changing the page table pointer
Outline: Four OS concepts

Threads: Execution context
- Program counter, registers, stack

Address spaces w/ translation
- Program's view of memory distinct from the physical machine

Process: an instance of a running program
- Address space + one (or more) threads

Dual mode operation
- Only the "system" can access certain resources
- Combined with translation, isolates user programs from each other
Review (61C): Instruction Cycle

The instruction cycle

Processor

PC: next

Instruction fetch

Decode

decode

Execute

Registers

ALU

Memory

data

instruction
Review (61C): Instruction Cycle

Execution sequence:
- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers/mem
- PC = Next Instruction(PC)
- Repeat
Outline: Four OS concepts

Threads: Execution context
- Program counter, registers, stack

Address spaces w/ translation
- Program's view of memory distinct from the physical machine

Process: an instance of a running program
- Address space + one (or more) threads

Dual mode operation
- Only the "system" can access certain resources
- Combined with translation, isolates user programs from each other
Threads of Control

Definition: **Single unique execution context**
- Program Counter, Registers, Stack

A thread is *executing* on a processor when it is resident in that processor's registers

Registers hold the root state of the thread:
- The rest is "in memory"
  - Including program counter – currently executing instruction

Registers point to thread state in memory:
- Stack pointer to the top of *the thread's* (own) stack
Multiprogramming: Multiple threads

- Proc 1
- Proc 2
- Proc n
- OS

Diagram showing memory layout with code, static data, heap, and stack for multiple processes.
Illusion of Multiple Processors

Thread are virtual cores

Multiple threads: Multiplex in time
Illusion of Multiple Processors

Contents of virtual core (thread):
- Program counter, stack pointer
- Registers

Where is it?
- On the real core, or
- In memory – called the thread control block (TCB)
Illusion of Multiple Processors

At **T1**: vCPU1 on real core; vCPU2 in memory

At **T2**: vCPU1 in memory; vCPU2 on real core

What happened?
- OS ran [how?]
- Saved PC, SP, ... in vCPU1's thread control block (memory)
- Loaded SP, ... from vCPU2's thread control block, jumped to PC
Simple Multiprogramming Problems

All vCPUs share the non-CPU resources
- Memory, I/O devices

Each thread can read/write the data of others

Threads can overwrite OS functions

Unusable? No.
- Embedded applications (sometimes)
- MacOS 1-9
- Windows 3.1
- Windows 95-ME (sort of)
Outline: Four OS concepts

Threads: Execution context
   - Program counter, registers, stack

Address spaces w/ translation
   - Program's view of memory distinct from the physical machine

Process: an instance of a running program
   - Address space + one (or more) threads

Dual mode operation
   - Only the "system" can access certain resources
   - With translation, isolates user programs from each other/the OS
Address Space

Definition: **set of accessible addresses and the state associated with them**

- $2^{32}$ on a 32-bit process

State: what happens when you read or write

- Nothing?
- Regular memory
- I/O operation
- Crash
- Communicate with other program
Address Space

Processor registers

PC:

SP:

0x000...

0xFFFF...

code

instruction

Static Data

heap

stack
Trivial Address Spaces: Base and Bound w/o translation

Program address

Base 1000...

Bound 1100...

Protects OS, other programs

No additions to address computation

Requires relocating program

Exception!
Relocation?

jal printf

000011 XX XXXXXXXX XXXXXXXX XXXXXXXX

opcode for jal

address of printf (shifted right by 2)
Address Space Translation

Program address space **distinct** from physical one
Adding Translation: Base and Bound

Program address

Base

Bound

<=

+
Adding Translation: Base and Bound

No runtime relocation

Program memory still limited

Still provides protection
Virtual Address Translation/Paging

Maybe you remember this from 61C?
  − Regardless, more on this later in the course

Gives every process the **whole address space**

Break into pages (~4K chunks)

Allows unallocated "holes"
  − and other tricks
Outline: Four OS concepts

Threads: Execution context
- Program counter, registers, stack

Address spaces w/ translation
- Program's view of memory distinct from the physical machine

Process: an instance of a running program
- Address space + one or more threads

Dual mode operation
- Only the "system" can access certain resources
- With translation, isolates user programs from each other/the OS
The Process

Process: execution environment with Restricted Rights
- **Address Space with One (or More) Threads**
  - Today: Just one thread
  - Owns memory (address space)
  - Owns file descriptors, file system context, ...

Why processes?
- **Protected from each other!**
- OS protected from them

Tradeoff: protection and efficiency
- Threads more efficient than processes (later)

Application: one or more processes
Protection

Why?
- Reliability: buggy programs only hurt themselves
- Security and privacy: trust programs less
- Fairness: share of disk, CPU

Mechanisms:
- **Address translation**: address space only contains its own data
- **Privileged instructions, registers**
- **Syscall processing** (e.g. enforce file access rights)
Outline: Four OS concepts

Threads: Execution context
- Program counter, registers, stack

Address spaces w/ translation
- Program's view of memory distinct from the physical machine

Process: an instance of a running program
- Address space + one (or more) threads

Dual mode operation
- Only the "system" can access certain resources
- With translation, isolates user programs from each other/the OS
Dual Mode Operation

**Hardware** provides to modes
- "Kernel" (or "supervisor" or "protected")
- "User" mode

Hardware support
Dual Mode Operation: HW Support

1 bit of state (user/kernel mode bit)

Certain actions only permitted in kernel mode

User->kernel transition sets kernel mode, saves user PC
  - Only transition to OS-designated addresses
  - OS can save the rest of the user state (if it needs to)

Kernel->user transition sets user mode, restores user PC
# UNIX System Structure

<table>
<thead>
<tr>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>Standard Libs</td>
<td>Hardware</td>
</tr>
<tr>
<td>(the users)</td>
<td>shells and commands, compilers and interpreters, system libraries</td>
<td>terminal controllers, terminals</td>
</tr>
<tr>
<td>Standard Libs</td>
<td>system-call interface to the kernel</td>
<td>device controllers, disks and tapes</td>
</tr>
<tr>
<td></td>
<td>signals terminal handling, character I/O system, terminal drivers</td>
<td>memory controllers, physical memory</td>
</tr>
<tr>
<td></td>
<td>file system, swapping block I/O system, disk and tape drivers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU scheduling, page replacement, demand paging, virtual memory</td>
<td></td>
</tr>
</tbody>
</table>
User/Kernel Mode

User Mode

Kernel Mode

exec
syscall
ret
rfi
interrupt
exception

Limited HW access
Full HW access
Types of (U->K) Mode Switches

System call (syscall)
  - "Function call" to the kernel
  - Example: exit, read
  - Kernel reads syscall id + args from user registers

Interrupt
  - External asynchronous events: timer, I/O

Trap or Exception
  - Special event in process
  - Protection violation (segfault), divide by zero, ...
A brief terminology note

Architectures have a lack of agreement about the names
- Interrupt
- Trap
- Exception

Sorry.

We will try to be consistent in this course.
Where do mode transfers go?

Cannot let user program specify! (Why?)

Solution: *Interrupt Vector*

```c
intrpHandler_i () {
  ...
}
```
Mode Transfer Hygiene

*Very careful* kernel code packs up user state

Must handle weird buggy/malicious user state:
- Syscall with null pointers
- Return instruction out of bounds
- User stack pointer out of bounds

Cannot let user corrupt kernel

User program should not know interrupt occurred
The Kernel Stack

Interrupt handlers want a stack
System call handlers want a stack
Can't use the user stack [Why?]
The Kernel Stack

Solution: Two stacks
- Each OS thread has kernel stack and user stack

Place to save user registers during interrupt
Before Interrupt

code:

foo () {
  while (...) {
    x = x + 1;
    y = y - 2;
  }
}

stack:

Registers

SS: ESP
CS: EIP
EFLAGS
other registers: EAX, EBX, ...

Kernel
code:

handler() {
  pusha
  ...
}

Exception Stack

Stack
During Interrupt

code:
foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

stack:

Registers:
- SS: ESP
- CS: EIP
- EFLAGS
- other registers: EAX, EBX, ...

Kernel:

handler() {
    pusha
    ...
}

Exception Stack:
- SS
- ESP
- EFLAGS
- CS
- EIP
- error
Handling interrupts safely

Interrupts disabling
- On entry to handler (by hardware)
- Reenabled on completion
- Explicitly by OS: (cli/sti on x86)
Why disable interrupts?

Interrupt while saving state during interrupt handler

Interrupt while restoring state to return from interrupt/syscall

Interrupt while reconfiguring interrupt handlers

Interrupt while selecting a new process to run
Handling interrupts safely

Interrupts disabling
- On entry to handler (by hardware)
- Reenabled on completion
- Explicitly by OS: (cli/sti on x86)
  - When selecting process to run [Why?]
  - Just before returning from interrupt/syscall [Why?]

Handler must be quick
- Can't hold up whatever it's interrupting
- Usually just queue work for later (wake up a thread)
Handling interrupts safely

**Atomically** change all of these:
- User/kernel mode
- Program counter
- Memory protection

→ "Return from interrupt" (RFI) instruction

Also needed for syscalls, etc.
Handling system calls safely

Same work as interrupts to enter/leave

Copy arguments from user registers/memory (!)
  - Carefully check memory locations!

Validate arguments
  - Kernel can't crash if user code is broken/malicious

Do operation

Copy result back into user registers/memory
  - Carefully check memory locations!
Addt'l HW support for interrupts

Multiple layers of interrupts
  – Priority levels
  – Some non-maskable (can't be disabled)
    • segmentation fault in segmentation fault handler
    • "watchdog" timer
Outline: Four OS concepts

Threads: Execution context
  – Program counter, registers, stack

Address spaces w/ translation
  – Program's view of memory distinct from the physical machine

Process: an instance of a running program
  – Address space + one or more threads

Dual mode operation
  – Only the "system" can access certain resources
  – With translation, isolates user programs from each other/the OS
Putting it together

Mode transfer and translation

Switching between processes

A brief note on hardware threads
Putting it together

Mode transfer and translation

Switching between processes

A brief note on hardware threads
Mode Transfer and Translation

Mode transfer should change translation map

Examples:
- Ignore base + bound in kernel mode
- Page tables with "kernel mode only" bits
Base and Bound: Load Process 2

sysmode: 1
Base: ????
Bound: ????
uPC: ????
PC: ...
regs: ...

(code: 0000...
static data: 0000...
heap: 0000...
stack: 0000...)

(Effectively 0000...)

(code: FFFF...
static data: FFFF...
heap: FFFF...
stack: FFFF...)

(Effectively FFFF...)
Base and Bound: Just After Return

- **Proc 1**
- **Proc 2**
- **Proc n**
- **OS**

- **Base**: 1000...
- **Bound**: 1100...
- **uPC**: ???...
- **sysmode**: 0

- **PC**
- **regs**...

- **0000...**
- **1000...**
- **1100...**
- **FFFF...**

- **Code**
- **Static data**
- **Heap**
- **Stack**
Base and Bound: Handling Interrupt

Need to setup OS's stack and save program registers!
Putting it together

Mode transfer and translation

Switching between processes

A brief note on hardware threads
Switching between processes

Just two mode transfers!
Base and Bound: Switch Processes

**Base and Boundary Addresses**

- **Base** refers to the starting address where the program code begins.
- **Bound** refers to the ending address where the program code ends.

**Address Spaces**

- **Code**: Contains the executable program code.
- **Static Data**: Stores data that does not change at runtime.
- **Heap**: Allocates and deallocates memory dynamically.
- **Stack**: Stores temporary data used by the program.

**Register Values**

- **sysmode**: Indicates the mode of operation.
- **uPC**: Program Counter (essentially 0000...).
- **PC**: Program Counter (effectively FFFF...).
- **regs**: General-purpose registers.

**Memory Layout**

- The memory is divided into sections, each with a specific address range.
- The diagram illustrates how different processes (Proc 1, Proc 2, ..., Proc n) share and access these memory spaces.
Base and Bound: Switch Processes

- **Proc 1**
- **Proc 2**
- **...**
- **Proc n**

**OS**

**sysmode**

- **0**

**Base**

- **1100...** (effectively 0000...)

**Bound**

- **1200...** (effectively FFFF...)

**uPC**

- **????...**

**PC**

- **0000 ...**

**regs**

- **...**

**code**

**static data**

**heap**

**stack**

**0000...**

**1000...**

**1100...**

**FFFF...**
Representing Processes

- **Proc 1**, **Proc 2**, **Proc n**
- **OS**

- **Sysmode**: 0
- **Base**: 1100...
- **Bound**: 1200...
- **uPC**: ????
- **PC**: 0000...
- **Regs**: 00FF...

- **Code**: RTU
- **Static Data**: Heap
- **Heap**: Stack
- **Stack**: (effectively 0000...)
- **(effectively FFFF...)**
Process Control Block (PCB)

Kernel representation of each process
- Status (running, ready, blocked)
- Register state (if not ready)
  - The thread control block – for now, one thread per process
- Process ID
- Execution time
- Memory translations, ...

Scheduler maintains a data structure of PCBs

Scheduling algorithm decides what to run
The Scheduler

if ( readyProcesses(PCBs) ) {
    nextPCB = selectProcess(PCBs);
    run( nextPCB );
} else {
    run_idle_process();
}
Putting it together: web server
Putting it together

Mode transfer and translation

Switching between processes

A brief note on hardware threads
Simultaneous Multithreading AKA Hyperthreading

Hardware technique
- Reuse instruction-level parallelism for *independent instructions*

Extra hardware registers!

Exposed to OS like multiple cores (just slow)
- Hardware is doing "same" trick as OS

Colored blocks show instructions executed
Conclusion: Four OS concepts

Thread
- Single unique execution context
- Program Counter, Registers, Stack

Address Space w/ Translation
- Programs execute in an address space that is distinct from the memory space of the physical machine

Process
- An instance of an executing program is a process consisting of an address space and one or more threads of control

Dual Mode operation/Protection
- Only the “system” has the ability to access certain resources
- The OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses
Break