CPU Virtualization

CS 450: Operating Systems
Michael Lee <lee@iit.edu>
Agenda

- Central question: how to implement time-sharing?
  - While maintaining OS control & maximizing performance
- “Limited direct execution”
- Mechanics of context switches
Direct Execution

- OS loads process program/data/args into predefined location(s), then points PC at entry point (e.g., `main`)
- When program terminates (e.g., return from `main`), OS cleans up process footprint (data/metadata)
Direct Execution

- Problems:
  - No concurrency
  - Process is unchecked; can wreak havoc on system!
Limited Direct Execution

- Must prevent user from:
  - accessing arbitrary memory addresses
  - executing “dangerous” instructions
    - e.g., access to I/O and system registers
**Limited Direct Execution**

- Must prevent user from:
  - accessing arbitrary memory addresses
  - executing “dangerous” instructions
  - e.g., access to I/O and system registers

province of VM (later)

focus on this first
Recall: kernel vs. user mode

- Privileged instructions can only be executed in kernel mode
  - (what happens when user attempts to run?)
- On x86: CPL flag in CS register — 0 = kernel, 3 = user
- After system boot, OS switches to user mode before ceding control to process
System Calls

- When user needs to perform I/O, invoke kernel-mode OS functions via system calls
  - e.g., `printf(...)` → `write(...)`
- Looks like a regular function call, but isn’t!
char *str = "hello world";
int len = strlen(str);
write(1, str, len);
...

movl len, %edx
movl str, %ecx
movl $1, %ebx  # syscall num
movl $4, %eax  # trap instr
int $0x80
...

...
Trap Mechanism

interrupt vector

[128]

... (mode switch) syscall

[4]

syscall table

OS

sys_write

syscall table

Process

Memory

movl $4, %eax

int $0x80
- x86-64 adds `syscall` instruction — avoids trap mechanism
- much faster! (software interrupts are expensive)
- but traps still used for other things

```
movq $4, %rax
syscall
```

![Diagram showing the process of system call execution with IA32_LSTAR model-specific register (MSR) and process context.]
General Interrupt Mechanism

IDTR (base address register) — populated by privileged `lidtr` instruction

- **0-31**: reserved for CPU-generated
- **32-255**: software configurable (for sw/hw interrupts)

*(not all can be triggered from user mode!)*
An interrupt gate or trap gate references an exception- or interrupt-handler procedure that runs in the context of a task gate (see Section 7.3, “Task Switching”). If index points to a task gate, the processor executes a task switch to the exception- or interrupt-handler task in a manner similar to a CALL to a call gate (see Section 5.8.2, “Gate Descriptors,” through Section 5.8.6, “Returning from a Called Procedure”). If index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the IDT. If the index points to an interrupt gate or trap gate, the processor handles calls to exception- and interrupt-handler procedures similar to the way it handles calls with a CALL to a call gate.

Figure 6-2. IDT Gate Descriptors

from Intel 64 and IA-32 Software Developer’s Manual, Volume 3
- Problem: when transitioning to OS code, process state may be lost (e.g., PC, SP, etc.)
- Should save in case we return to process after servicing trap
Saving Process State

- Hardware automatically saves current context during trap
  - Where?
    - On *kernel stack* — automatically activated on mode switch
- Every process has its own separate kernel stack — used to keep track of kernel state (e.g., while handling I/O)
Figure 3-1. Kernel stack after an int instruction.

from xv6 commentary
Restoring Process State

- “return from trap” instruction: iret — pops and restores trap frame and returns to process in user mode

(On x86-64, sysret instead; loads PC from %rcx)
- Do we always immediately return to trapping process?
  - No! (Why not?)
    - Process may be blocked (due to I/O request)
    - Scheduling decision
Context Switch

1. Trap to kernel; save trap frame on kernel stack
2. Save outgoing process context on kernel stack
3. Switch to different kernel stack
4. Restore incoming process context from kernel stack
5. iret (restore trap frame from kernel stack)
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret

trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp
  iret
Cooperative Multitasking, done!

- aka non-preemptive multitasking
- Only context switch on trap to OS that results in:
  - process termination
  - process blocking
- Can also add “yield” system call to voluntarily cede control
Preemptive Multitasking

- Must guarantee that OS regains control periodically
- Hardware assistance: schedule periodic clock interrupt at fixed time intervals (e.g., 1ms)
  - Decide whether to perform context switch after some number of intervals (typically ~100ms)
Decision = Policy

- Context switch is merely a mechanism
  - Carried out by low level dispatcher
- When to carry out context switch is decided by the scheduler
  - Scheduling policies/algorithms, next!