CS 695: Virtualization and Cloud Computing

Lecture 2: Review of Operating Systems Concepts

Mythili Vutukuru
IIT Bombay
Spring 2021
Recap of Operating Systems

• The following OS concepts are required to understand virtualization
  • The concept of a process
  • Virtual memory, paging, (segmentation)
  • User mode and kernel mode of a process
  • Interrupt/trap processing in kernel mode
  • I/O handling
The concept of a process

• A process is a running program
• User writes a program, compiles it to generate an executable
• Executable contains machine/CPU instructions
  • Every CPU architecture (e.g., x86) defines a certain set of instructions
  • Compiler translates high level language code to instructions the CPU can run
• To create a process, OS allocates memory in RAM for the memory image of the process, containing
  • Code and static/global data from the executable
  • Heap memory for dynamic memory allocations (e.g., malloc)
  • Stack to store arguments/return address/local variables during function calls
• All instructions and variables in the memory image are assigned memory addresses
  • Starting at 0, up to some max value (4GB in 32-bit systems)
Memory image of a process

Address = 0
Code/data
from compiled executable
(machine instructions, static/global data)

Address = 12KB
Heap
Address of this memory returned by malloc

Unused addresses

Stack
New stack frame pushed by function call
Contains arguments, local vars, return address etc.
Popped when function returns

Max Address = 4GB

Heap and stack grow towards each other
Process execution

• When a process is run, CPU executes the code in the memory image

• When a process runs on the CPU, the **CPU registers** hold values related to the process execution
  • The **program counter** (PC, or EIP in x86) has address of current instruction
  • The CPU fetches the current instruction, decodes it, and executes it
  • Any variables needed for operations are loaded from process memory into **general purpose CPU registers** (EAX, EBX, ECX, EDX etc in x86)
  • After instruction completes, values are stored from registers into memory
  • The **stack pointer** (SP, or ESP in x86) has address of top of stack (current stack frame holds arguments/variables of the current function that is running)

• The set of values of all CPU registers pertaining to a process execution is called its **CPU context**
CPU context during execution

- EIP (PC)
- EAX
- EDX
- ESP

Code/data
- Heap
- Stack
- ....
Concurrent execution, context switching

• To run a process, OS allocates memory, loads CPU context
  • EIP points to instructions, ESP points to stack of process, registers have process data
  • CPU now begins to run the process

• OS runs multiple process **concurrently** by multiplexing on the same CPU

• **Context switch:** After running a process for some time, OS switches from one process to another

• How does context switch happen? OS saves the CPU context of the old process and loads the context of new process
  • When EIP points to instruction of new process, new process starts to run

• Where is the context saved?
  • OS has a data structure called **Process Control Block (PCB)** for each process
  • PCB (specifically, a PCB field called kernel stack) temporarily stores context of a process when it is not running
Context switch

CPU has switched execution from blue process to green process

User memory
Kernel memory

Process control blocks
Virtual memory

• Addresses assigned in memory image, that are used by CPU to load and store are virtual/logical addresses
  • These virtual addresses are not actual addresses occupied by instructions/data of process, but assigned from 0 for convenience

• Actual addresses where instructions/data bytes of process are stored are called physical addresses
  • RAM hardware needs physical addresses to fetch bytes

• CPU requests code/data at virtual addresses
  • Translated to physical address so that RAM can fetch it

• Memory is allocated at granularity of pages (usually 4KB)
  • Logical pages of a process are stored in physical frames in memory
  • Logical page numbers translated to physical frame numbers
Virtual and physical address spaces

CPU

Fetch address 5KB

Fetch address Y+1KB

Page (4KB)

Address = 0

Address = 4GB

Heap

Address = X

Address = Y

Stack

Address = Z

Virtual address space

Physical address space
Address translation with paging

• On every memory access, a piece of hardware called MMU (memory management unit) translates virtual addresses to physical addresses

• Page table of a process stores the mapping of logical page number → physical frame number
  • OS builds the page table when allocation memory
  • MMU uses this page table to translate addresses

• MMU looks up CR3 register of CPU (x86) which stores location of page table of current process
  • Looks up page number in page table to translate address
  • CR3 reset on every context switch

• Recent address translations cached in TLB (Translation Lookaside Buffer) located within MMU
CPU

MMU

CR3

Virtual address space

Physical address space

Page table (page 1 = frame y)

Virtual address

Page number

Offset

Using page table

Physical address

Frame number

Offset

Address = X

Address = Y

Address = Z

Fetch address 5KB

Fetch address Y+1KB

Virtual address space

Page

Physical address space

Address = Y
Hierarchical page tables

• Modern operating systems use hierarchical page tables
• 32-bit virtual address space (4GB), $2^{12}$ byte (4KB) pages
  • Each process can have up to $2^{32}/2^{12} = 2^{20}$ pages
  • Each page has a page table entry (PTE), so $2^{20}$ PTEs per process
  • Assuming each PTE is 4 bytes, all page table entries occupy $4 \times 2^{20} = 4$MB
• Cannot store a large page table contiguously in memory, so page table of a process is stored in memory in page sized chunks
  • Each 4KB page stores $2^{10}$ PTEs, so $2^{20}/2^{10} = 2^{10}$ pages to store all PTEs
  • Pointers to these “inner” pages stored in an outer page directory
  • In 32-bit architectures, one outer page can store physical frame numbers of all $2^{10}$ inner page table pages, so 2-level page table
  • More levels in page table for 64-bit architectures
Page table lookup (walking the page table)

- **CPU’s CR3 has physical address of outermost page directory**
- Physical frame numbers of $2^{10}$ inner pages stored in outer page directory
- Multiple such levels of page tables can exist

**Diagram Details**

- **Page table entry (PTE)**
  - Stores physical frame number
  - **Array of PTEs**
    - Split among multiple pages
    - $2^{32}/2^{12} = 2^{20}$ pages
    - $2^{20}$ PTEs in $2^{10}$ pages

**Virtual Address**

- MMU
  - 10 bits
  - Frame number
  - 10 bits
  - Offset
  - 12-bit offset

**Physical Address**

- Mapping cached in TLB
Demand Paging

- CPU
  - Fetch address 5KB
  - Page fault

- Page
  - Address = 0
  - Address = X
  - Address = Y
  - Address = Z
  - Address = 4GB

- Heap
- Stack
- Virtual address space
- Physical address space

- Another victim page swapped to disk
- Physical frame assigned to page
- Page table updated
- CPU instruction rerun
User mode and kernel mode

- Two kinds of CPU instructions
  - Unprivileged instructions (regular program code) – part of user code
  - Privileged instructions (access to hardware etc.) – part of OS code
- Modern CPUs have multiple privilege levels (rings)
  - Privileged instructions are executed only when CPU is at high privilege level
- User code runs at low privilege level / user mode (CPU set to ring 3 in x86)
  - If user runs privileged instructions, error is thrown
- OS code runs at high privilege level / kernel mode (CPU set to ring 0 in x86)
  - Allowed to run privileged instructions
- Privilege level checked by CPU and MMU (every page has privilege bit)
- When user process needs to perform privileged action, must jump to privileged OS code, set CPU to high privilege, and then perform the action
Where is OS code located?

- OS is part of high virtual address space of every process.
- Page table of process maps these kernel addresses to location of kernel code in memory.
- Only one copy of OS code in memory, but mapped into virtual address space/page table of every process.
- Process jumps to high virtual addresses to run OS code.
Kernel mode execution

• When does a process go from user mode to kernel mode?
  • **System call**: user requests some privileged action from OS (e.g., read syscall)
  • **Program fault**: hardware raises a fault when user does some invalid action (e.g., privileged instruction in user mode, page fault)
  • **Interrupt**: I/O devices request attention (e.g., packet has arrived on NIC)
  • All these are called *traps* in general

• How is a trap handled?
  • Change CPU privilege level to high privilege/kernel mode
  • Jump to OS code that handles trap and run it
  • Return to user code after handling trap
  • Can choose to return to user code of another process too (if context switch)
What happens upon a trap?

• **Trap handling begins by running a CPU instruction** (*int n in x86*)
  - Invoked by system call code or triggered by hardware event

• **What happens during execution of int n?**
  - Change CPU privilege level
  - Lookup **Interrupt Descriptor Table (IDT)** with index “n” to get address of kernel code that handles this trap → set EIP to this value
  - Use **kernel stack** of process (in PCB) as the main stack → set ESP to this value
  - Start saving user context onto the (kernel) stack

• **Next, kernel code to handle the trap runs**
  - Save more user context, beyond what is saved by hardware instruction
  - System call processing, interrupt handling etc.

• **Finally, kernel invokes **iret** (x86) instruction to return back to user mode**
  - Reverses changes of *int n*
Trap handling

CPU

EIP (PC)
EAX
.....
EDX
ESP

int n executed
ring 0

Set EIP = IDT[n]

Set ESP = top of kernel stack

Code/data
Heap
Stack
.....

Kernel stack of process (part of PCB kernel data structure)

User context saved
Interrupt Descriptor Table (IDT)

• Every interrupt has a number (IRQ)
  • E.g., different hardware devices get different numbers, system call gets a different number
• IDT has an entry for every interrupt number (IRQ)
  • IDT entry specifies values of EIP and a few such CPU registers
  • CPU uses this EIP to locate kernel interrupt handling code
• Pointer to IDT is stored in CPU register
  • Setting IDT pointer in CPU is a privileged operation, done by OS
  • Much like how setting CR3 to page table is a privileged operation
Segmentation: An alternate to paging

- Code/data
- Heap
- Stack
- Virtual address space
- Physical address space

Base = Start address of segment
Limit = End address of segment

Base = BHeap
Limit = LHeap

Base = BStack
Limit = LStack

Base = BCode
Limit = LCode

CPU

Fetch address at 1KB
In code segment

Fetch address BCode +1KB

Base = Start address of segment
Limit = End address of segment
CPU

Segment descriptor table (BCode, BLimit, ...)

Virtual address space

Physical address space

MMU

CPU

Fetch address at 1KB in code segment

Fetch address BCode+1KB

Virtual address

Segment

Offset in segment

Add segment base to offset

Linear/physical address

Address

Base = BCode

Limit = LCode
Use of segmentation today

• What do operating systems use today? Segmentation or Paging?
  • Hardware is built to do segmentation+paging, but only paging in practice
  • Modern OSes use “flat” segments: Base = 0, Limit = MAX (4GB)
    • Virtual address remains same after segmentation-based translation

• Paging is used to manage virtual memory in reality
  • After dummy translation with segmentation, paging does actual translation
  • After adding segment base (0), linear address is translated using page table

• Segmentation is used to check permissions (along with paging)
  • Modern OSes have different flat segments for different privilege levels
  • CS has two different values in user and kernel mode (base=0, limit=MAX, only privilege level is different)
  • CS has high privilege level when executing privileged instructions
I/O subsystem: system calls, interrupts

- Processes use system calls to access I/O devices
  - Process P1 makes system call, goes into kernel mode
  - OS device driver initiates I/O request to device (e.g., disk, network card)
  - If the system call is blocking (i.e., cannot be completed right away), OS performs context switch to another process P2
  - When request completes, I/O device raises interrupt
  - P2 goes into kernel mode, handles interrupt, marks P1 as ready to run
  - P1 runs at a later time when scheduled by OS scheduler

- DMA: I/O devices perform Direct Memory Access to store I/O data in memory
  - When initiating I/O request, device driver provides (physical) address of memory buffer in which to store I/O data (e.g., disk block)
  - Device first stores data in DMA buffer before raising interrupt
  - Interrupt handler need not copy data from device memory to RAM
Summary of OS concepts

• The concept of a process, memory image, CPU context
  • CPU context is saved in PCB/kernel stack during user/kernel mode transitions of a process, as well as during context switch between processes

• Virtual memory, paging, address translation by MMU using page table
  • OS is mapped into the virtual address space of every process
  • Modern OSes use flat segments + paging

• User mode and kernel mode of a process, privileged instructions, CPU privilege levels
  • Privileged instructions in OS code run at highest CPU privilege level (ring 0)

• Interrupt/trap processing in kernel mode
  • Change privilege level, save user context, jump to kernel code, handle trap

• I/O handling