Computer architecture so far

What do we have to add to our processor to support a modern operating system?

Single program, communicates via MMIO
Single program, communicates via MMIO

What do we have to add to support a modern operating system?
- Isolation between processes
- System abstraction – Hide details about underlying hardware
- Resource management – CPU, memory, disk, network, ...

Goal: support consistent abstraction to software
Even with changing hardware, drivers, etc!
Aside: The old days

- Old personal operating systems (MS-DOS, CP/M, ...) were very basic
  - The division between OS and user software was not strong
  - OS basically “jalr” into the user software, and “ret” out
  - User software had all access to hardware, including OS files on disk
  - Only one software running at a time!
  - Software failure -> System crash!

- Not much hardware abstraction
  - Each software had to handle each possible video, sound, etc hardware
Aside: The CP/M operating system (1974)

- Control Program/Monitor, created by Digital Research, Inc.
  - Designed for Intel 8080, with less than 64 KiB of memory
  - Massive popularity, massive influence to MS-DOS (1981)
    - A: B: C: device naming, “BIOS”, AAAAAAAA.EXT naming scheme, etc survives until now

- Extremely simple O/S
  - Still used/modified by hobbyists!

Source: Digital Research, Inc.

Source: http://benryves.com/projects/z80computer
Aside: The CP/M operating system (1974)

- Once booted, the CCP command line is presented.

- When executing software, binary is loaded to low part of free memory, and OS simply jumps to that region
  - Always only one execution context (process)

- User software interfaces with OS via BDOS
  - BDOS location is stored as a pointer in “Low storage”
  - Scheme allows contiguous memory for software regardless of memory capacity

- When done execution, simply returns to OS

Simple! Software has exclusive access to machine
OS is effectively just like a library – DOS was very similar
Aside: Something new – multitasking

- Multiple tasks (processes) executing concurrently
  - Multi-user systems, servers with multiple parallel workloads, services, GUI, ...

- Memory usage becomes complicated with multitasking
  - Two binaries cannot be loaded to same memory location, software can be loaded to arbitrary, possibly non-contiguous, locations
  - Will have contention between processes for data memory locations
  - We cannot use absolute addressing any more for jumps and data referencing!
  - No longer simple address model with assumed exclusive access to memory

Address “0x3c0” is encoded as literal. Needs exclusive access guarantee (At compile time?!)
Modern operating systems

- Modern operating systems support user process isolation
- The OS kernel provides a private address space to each process
  1. Each process thinks it has exclusive access to contiguous memory
  2. A process is not allowed to access the memory of other processes
  3. No user process can access OS memory
- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
- The OS kernel lets processes invoke system services (e.g., access files or network sockets) via system calls

Familiar concepts from OS classes!
Architectural support for operating systems

- Operating system must have different capabilities from user processes
  - Typical ISA defines two or more “privilege levels” (e.g., “user”, and “supervisor”)
  - Some instructions and registers that are only accessible for a process executing in supervisor mode
  - Typically, the very first process to execute is given supervisor privilege, and is responsible for spawning future user processes

- Interrupts and exceptions to transition from user to supervisor mode

- Virtual memory to provide private address spaces and abstract the storage resources of the machine
  - User processes executing LW/SW/etc access memory through a hardware virtual memory manager
Topics

- Privilege levels
- Interrupts and exceptions
- Virtual memory
Privilege levels in modern architectures

- RISC-V has three (or more) formally defined levels
  - Machine level, full access to all hardware after initial boot
  - Hypervisor level – For virtualization. Recently formally defined! (2022)
  - Supervisor level – For operating systems
  - User level – For applications

- x86 has “protection rings”
  - Typically only ring 0 and 3 are used
  - Additional ring -1 for hypervisors

- Each process/thread belongs on one level

  Less privileged levels have more restrictions
  - Cannot access some registers
  - Can only access memory via virtual memory, not raw hardware
Example: RISC-V

- Special register, “mstatus” (for “machine status”)
  - Among other information, stores the privilege level of the current process
  - Writing a new value to it can change the privilege level, but only machine mode is allowed to write to it
  - OS runs in machine mode, when user process must be spawned, it first spawns a kernel process which downgrades itself to user mode before jumping to actual user software

- Special ISA instructions to access the special registers
  - One of many “Control Status Register”
  - csrr, csrw instructions, only allowed in machine mode
  - There are many CSRs! Will mention more soon.

x86 typically has separate instructions for each privileged operation
Topics

- Privilege levels
- Interrupts and exceptions
- Virtual memory
Exceptions?

- Event that needs to be processed by the OS kernel. The event is usually unexpected or rare
  - Exceptions cause an exception handler in OS, in higher privilege

![Diagram showing the process and exception handler](image-url)
Typical terminology

- **Exceptions**: Usually events caused by the running process itself
  - Illegal memory access (SEGFAULT), divide-by-zero, system call, etc

- **Interrupts**: Usually events caused by the outside world
  - Timer, I/O completion, keystroke, etc

- Terminology is often used interchangeably...
Handling exceptions

- When an exception happens, the **processor**:  
  - Stops the current process at instruction $I_i$, completing all the instructions up to $I_{i-1}$  
  - Saves the PC of instruction $I_i$ and the reason for the exception in special (privileged) registers  
  - Enables supervisor mode, disables interrupts*, and transfers control to a pre-specified exception handler PC

- After the exception handler finishes, the processor:  
  - Returns control to the user process at instruction $I_i$  
  - User process is oblivious to the interrupt

- If an interrupt is due to an illegal operation, the OS aborts the process  
  - e.g., SEGFAULT
Handling exceptions

- The operating system is responsible for telling the processor how to handle each type of exception
  - Typically via a table of pointers in main memory, each corresponding to a particular exception type
  - A special register is set with a pointer to the table in memory (“mtvec” for RISC-V, “IDTR” for x86)
- For each exception, the CPU transparently consults this register, reads the table, and jumps to the correct handler
  - No software involved in this process. Hardware!

<table>
<thead>
<tr>
<th>INT_NUM</th>
<th>Short Description</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Division by zero</td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td>Single-step interrupt (see trap flag)</td>
<td></td>
</tr>
<tr>
<td>0x02</td>
<td>NMI</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td>Breakpoint (call to the special 1-byte instruction 0xCC, used by debuggers)</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>Overflow</td>
<td></td>
</tr>
<tr>
<td>0x05</td>
<td>Bounds</td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td>Invalid Opcode</td>
<td></td>
</tr>
<tr>
<td>0x07</td>
<td>Coprocessor not available</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td>Double fault</td>
<td></td>
</tr>
<tr>
<td>0x09</td>
<td>Coprocessor Segment Overrun (386 or earlier only)</td>
<td></td>
</tr>
</tbody>
</table>
Exception use #1: CPU scheduling

- The OS kernel schedules processes into the CPU
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- Key enabling technology: Timer interrupts
  - Kernel sets timer, which raises an interrupt after a specified time

![Diagram showing CPU scheduling and timer interrupts](image)
Exception Use #2: Emulating Instructions

- mul x1, x2, x3 is an instruction in the RISC-V ‘M’ extension (x1 = x2 * x3)
  - If ‘M’ is not implemented, this is an illegal instruction

- What happens if we run code for an RV32IM ISA on an RV32I machine?
  - mul causes an illegal instruction exception
  - The exception handler can take over and abort the process... but it can also emulate the instruction!
Emulating Unsupported Instructions

- Program believes it is executing in a RV32IM processor, when it’s actually running in a RV32I
- The IBM System/360 line of machines used this method to build cheap machines that adhere to ISA

What are the downsides?
Slower performance compared to HW implementation!
Exception Use #3: System Calls

- User process has no access to raw hardware resources (not even the keyboard)
  - User process communicates with the OS via system calls (and other methods)
  - The syscall instruction (SYSCALL in x86, ecall in RISC-V) results in a machine-mode exception that can handle the request
    - Arguments and return values following familiar function call conventions
  - Aside: x86 used to assign a special number in the interrupt table (0x80) to handle syscalls. This is still technically supported, but discouraged
    - “int 0x80” vs. “syscall”
RISC-V provides several privileged registers, called control and status registers (CSRs), e.g.,
- mepc: PC of instruction that caused exception
- mcause: cause of the exception (interrupt, illegal instr, etc.)
- mtvec: address of the exception handler
- mstatus: status bits (privilege mode, interrupts enabled, etc.)

RISC-V also provides privileged instructions, e.g.,
- csrr and csrw to read/write CSRs
- mret to return from the exception handler to the process
- Trying to execute these instructions from user mode causes an exception. Normal processes cannot take over the system
System call details for RISC-V

- `ecall` instruction causes an exception, sets `mcause` CSR to a particular value

- Application Binary Interface (ABI) convention defines how process and kernel pass arguments and results
  - Typically, similar conventions as a function call:
    - System call number in `a7`
    - Other arguments in `a0` - `a6`
    - Results in `a0` - `a1` (or in memory)
  - **All** registers are preserved (treated as callee-saved) Why is this?
Typical System Calls

- Accessing files (sys_open/close/read/write/…)
- Using network connections (sys_bind/listen/accept/…)
- Managing memory (sys_mmap/munmap/mprotect/…)
- Getting information about the system or process (sys_gettime/getpid/getuid/…)
- Waiting for a certain event (sys_wait/sleep/yield…)
- Creating and interrupting other processes (sys_fork/exec/kill/…)
- ... and many more!

- Programs rarely invoke system calls directly. Instead, they are used by library/language routines
- Some of these system calls may block the process!
Hello world using x86 system calls

- Old example using using int 0x80

```assembly
section .data
    msg db "hello, world!" ; defining the message

section .text
    global _start ; this is for the linker

_start:
    mov rax, 4 ; Select system call: 4 = sys_write
    mov rbx, 1 ; First argument: 1 = stdout
    mov rcx, msg ; Second argument: pointer to message
    mov rdx, 13 ; Third argument: number of bytes to be written

    int 0x80 ; perform the chosen system call (pass variables
             ; inside registers to the kernel and it will do
             ; the rest)

    mov rax, 1 ; 1 = sys_exit
    mov rbx, 0 ; exit status = 0

    int 0x80 ; again, perform system call, this time sys_exit
```

http://boccelliengineering.altervista.org/junk/asm/assembly1.html
So far...

- **Operating System goals:**
  - Protection and privacy: Processes cannot access each other’s data
  - Abstraction: OS hides details of underlying hardware
    - e.g., processes open and access files instead of issuing raw commands to disk
  - Resource management: OS controls how processes share hardware resources (CPU, memory, disk, etc.)

- **Key enabling technologies:**
  - User mode + supervisor mode w/ privileged instructions
  - Exceptions to safely transition into supervisor mode
  - Virtual memory to provide private address spaces and abstract the machine’s storage resources *(next lecture)*
Context switching

- On a multitasked system, a processor cycles over multiple processes, executing them in small increments.

- Simply jumping between where we left off does not ensure correctness!
  - When we jumped into the kernel-space interrupt handler, the register values are stored in the stack, so they can be reclaimed after exiting the interrupt handler.
    - Remember, all registers are callee-saved in this situation because the user process is unaware.
  - How do we know where to get the next register values? e.g., stack pointer?
Context switching

- Context: The state of the process or thread which must be saved and restored for seamless multiprocessing
  - So far: PC, entirety of the register file (including the stack pointer, x2)
  - In reality, a lot more information including virtual memory state

- Context switching: Storing the context of the current process and loading the context of a new process
  - The processor is (conceptually) oblivious to processes
    - The concept of processes does not exist at the processor level, it’s just executing instructions
  - Like loading the same body (processor) with a different soul (context)
Context switching – Process Control Block

- Context information is managed in the OS via a construct called the Process Control Block (PCB)
  - Again, the processor is completely unaware of this
  - Stores information including the process ID, context state (register values, etc), meta-information for scheduling control (when was it last scheduled? etc)
  - An array of PCBs, one element per process/thread
  - Operating system topic! Only introduced here to connect the dots between architecture and OS

- In Linux, PCB is “struct task_struct”
Context switching – Process Control Block

- The OS software (not the processor hardware) is responsible for context switching, including:
  - Storing the current context to the appropriate PCB
  - Deciding which process to execute (and for how long)
  - Loading the next context from the PCB to the hardware registers
  - Resuming the next process
    - “Resuming” because it is currently suspended while the current process was executing
Aside:
Hardware vs. software context switching

- Some processor designs support hardware handling of context switching operations (e.g., x86)
  - CALL or JMP under special circumstances evoke hardware handling of context switching
  - Processor hardware automatically read/writes the PCB if it is in a specific format

- Unfortunately, most mainstream OSs don’t use it
  - High overhead as some of the hardware-defined context includes some values that are no longer useful in modern OSs
    - e.g., segment registers, will introduce soon
  - Some newer registers are not automatically restored
    - e.g., floating point

Modern processors often omit this feature in 64-bit mode
Aside: x86 way of creating user-level processes

- x86 doesn’t provide a way to explicitly switch to user level
  - Instead, we write code that pretends to return from an interrupt, back into user level
  - Allocate stack space in memory, and populate it with a return address, stack pointer, thread information, ... pretending to be a user level process whose interrupt is being handled
  - Call “IRET” which reads the stack, and “returns” to user level operation
System boot process

- Our RV32I processor, when powered on, starts executing from address 0
  - When powered on, memory is blank... How does OS get there?
  - Short answer: Firmware (e.g., BIOS, UEFI)

- Firmware is usually located in address 0
  - Special ROM/EEPROM/etc hardwired to map to address zero
  - On power on, CPU executes the firmware to load a small “bootloader” from storage and loads it to a special address, and transfers control
  - Bootloader loads the actual OS kernel from storage to memory and transfers control
Why bootloader?

- BIOS (Basic Input/Output System) treated the first sector (512 Bytes) of a storage medium specially (MBR, “Master Boot Record”)
  - BIOS loaded the MBR of the first HDD to memory and executed it
  - Bootloader had to fit in 512 Bytes, and is responsible for finding/loading the OS kernel and executing it
  - Due to complexities of file systems, etc, sometimes two-level bootloaders were used (e.g., GRUB on Linux)
    - Bootloader loads the second bootloader and executes it, which in turn loads the whole kernel

- UEFI (Unified Extensible Firmware Interface) doesn’t use MBR, instead stores bootloaders in a special UEFI partition
  - Still not the whole kernel!