CS 316: Privileged Mode, Exceptions and Interrupts

Kavita Bala

Fall 2007

Computer Science

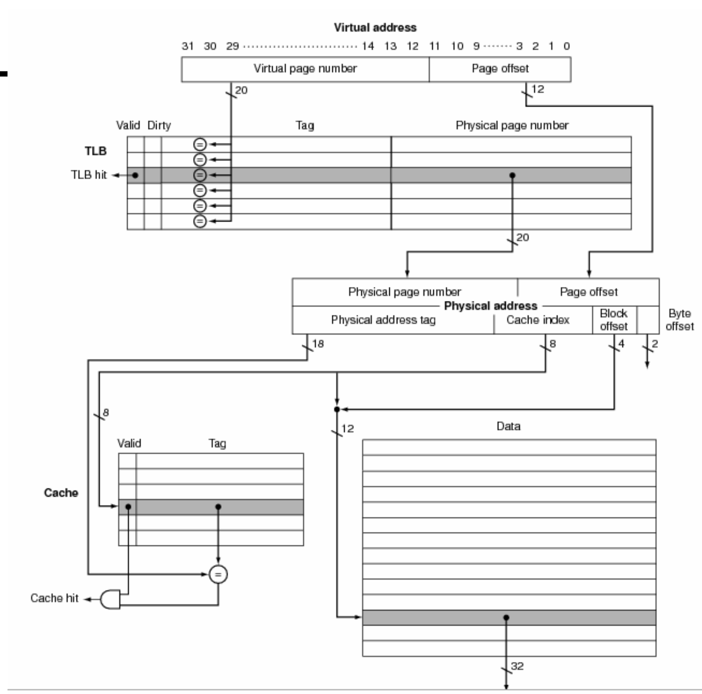
Cornell University

Announcements

- PA 4 is due next Monday (Nov 12)
- In recitation, will be available to answer questions
- HW 1 finite state machine

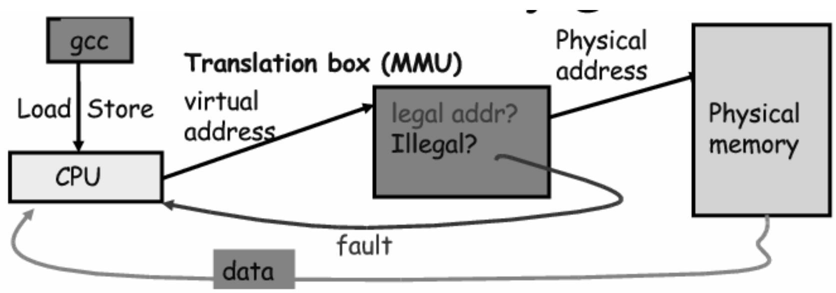
Caches/TLBs/VM

- Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
  1. Where can block be placed?
  2. How is block found?
  3. What block is replaced on miss?
  4. How are writes handled?



Address Translation

- Translation is done through the page table
  - A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault

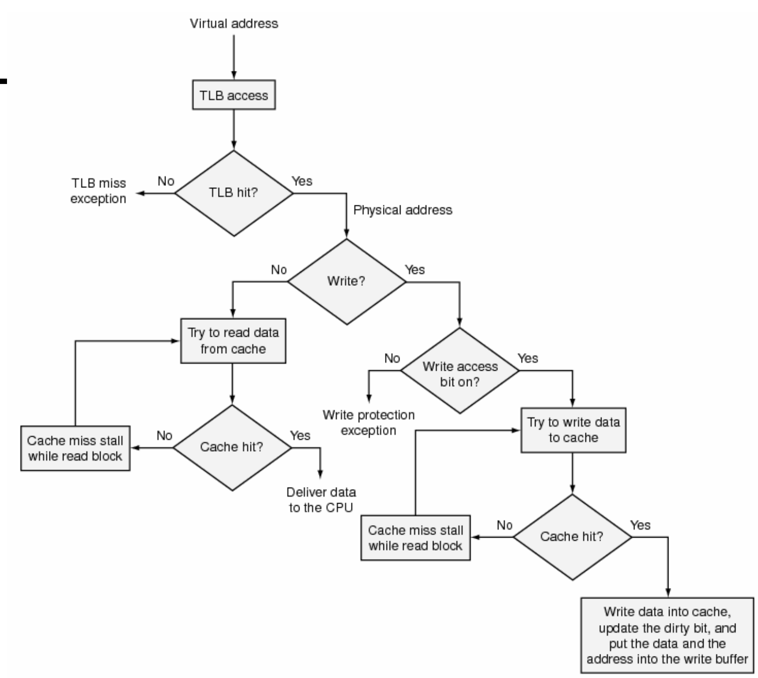


Hardware/Software Boundary

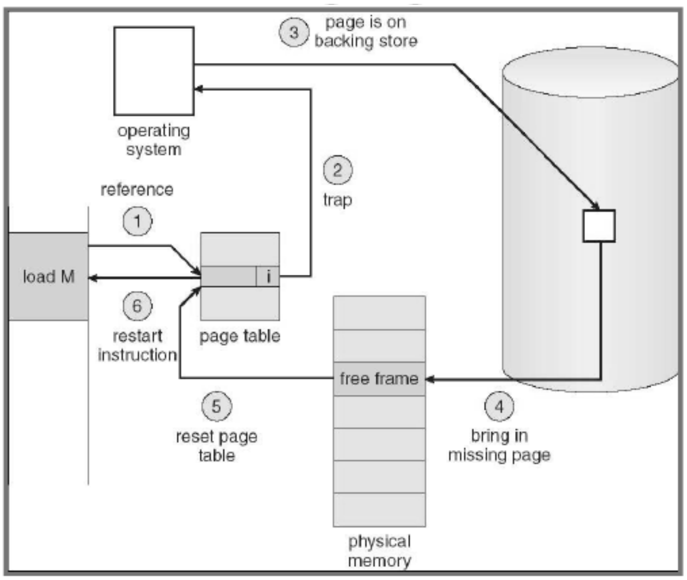
- Virtual to physical address translation is assisted by hardware
- Hardware
  - Translation Lookaside Buffer (TLB) that caches the recent translations
    - TLB access time is part of the cache hit time
    - May allot an extra stage in the pipeline for TLB access
  - TLB miss
    - Can be in software (kernel handler) or hardware

Hardware/Software Boundary

- Virtual to physical address translation is assisted by hardware
- Software
  - Page table storage, fault detection and updating
    - Page faults result in interrupts (precise) that are then handled by the OS
    - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables



Paging



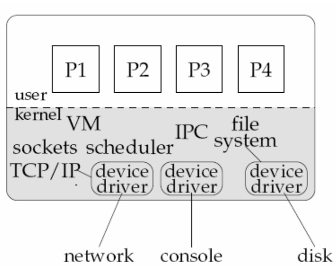
Example: paging to disk

- Compiler (gcc) needs a new page of memory
- Trap/exception into OS
  – OS gets page from application (vi)
- If page is clean, give up page to gcc
- If page is dirty, … only copy in memory
  – Write to disk, before giving it to gcc
- Mark page invalid in vi
  – (if vi needs this soon, it will trap)

Privileged Mode, Exceptions, Traps and Interrupts

Privilege Levels

- Some processor functionality cannot be made accessible to untrusted user applications
  - e.g. HALT, change MMU settings, set clock, reset devices, manipulate device settings, …
- Need to have a designated mediator between untrusted/untrusting applications
  - The operating system (OS)



Privilege Mode

- Need to delineate between untrusted applications and OS code
  - Use a “privilege mode” bit in the processor
  - 0 = Untrusted = user, 1 = Trusted = OS
- Privilege mode bit indicates if the current program can perform privileged operations
  - On system startup, privilege mode is set to 1, and processor jumps to a well-known address
  - The OS boot code resides at this address
  - The OS sets up the devices, initializes the MMU, loads applications, and resets the privilege bit before invoking the application
- Applications must transfer control back to OS for privileged operations

Terminology

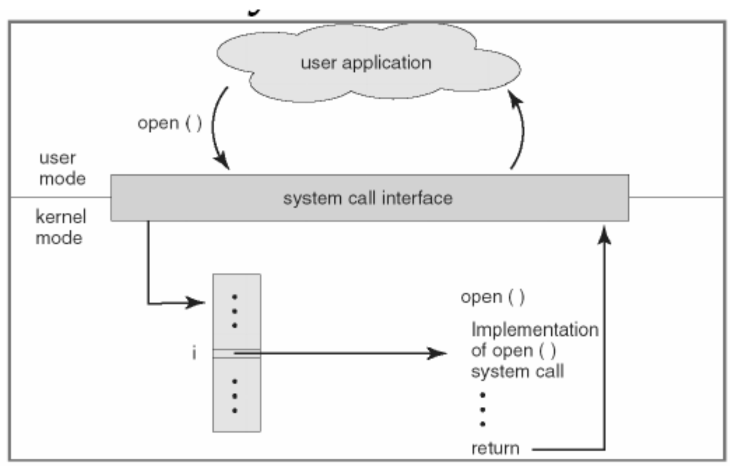
- Trap
  - Any kind of a control transfer to the OS
- Syscall
  - Synchronous, program-initiated control transfer from user to the OS to obtain service from the OS
  - e.g. SYSCALL
- Exception
  - Asynchronous, program-initiated control transfer from user to the OS in response to an exceptional event
  - e.g. Divide by zero
- Interrupt
  - Asynchronous, device-initiated control transfer from user to the OS
  - e.g. Clock tick, network packet

Sample System Calls

- Print character to screen
  - Needs to multiplex the shared screen resource between multiple applications
- Send a packet on the network
  - Need to manipulate the internals of a device
- Allocate a page
  - Needs to update page tables & MMU

System Calls

- A system call is a controlled transfer of execution from unprivileged code to the OS
  - An alternative is to make OS code read-only, and allow applications to just jump to the desired system call routine. Why is this a bad idea?
- A SYSCALL instruction transfers control to a system call handler at a fixed address
  - On the MIPS, v0 holds the syscall number, which specifies the operation the application is requesting

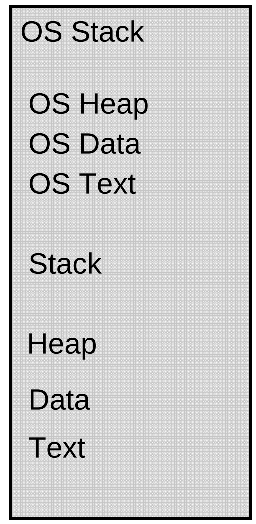


Where does OS live?

- In its own address space?
  - But then syscall would have to switch to a different address space (not possible on most architectures)
  - Also harder to deal with syscall arguments passed as pointers
- So in the same address space as process
  - Use protection bits to prevent user code from writing kernel

Full System Layout

- Typically all kernel text, most data
  - At same VA in every address space
  - Map kernel in contiguous physical memory when boot loader puts kernel into PM
- The OS is omnipresent and steps in where necessary to aid application execution
  - Typically resides in high memory
- When an application needs to perform a privileged operation, it needs to invoke the OS



SYSCALL instruction

- SYSCALL instruction does an atomic jump to a controlled location
  - Switches the sp to the kernel stack
  - Saves the old (user) SP value
  - Saves the old (user) PC value (= return address)
  - Saves the old privilege mode
  - Sets the new privilege mode to 1
  - Sets the new PC to the kernel syscall handler

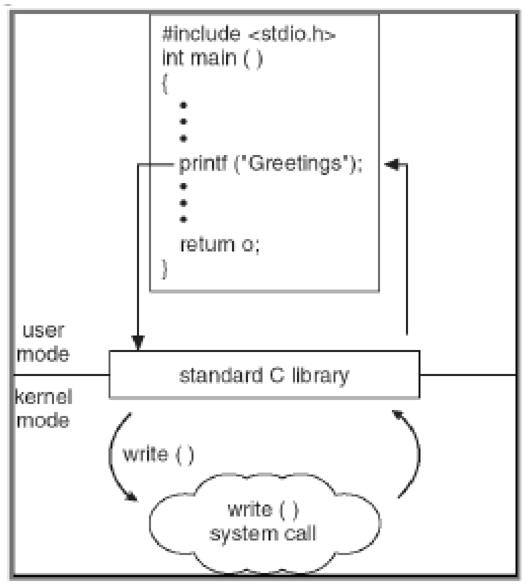
SYSCALL instruction

- Kernel system call handler carries out the desired system call
  - Saves callee-save registers
  - Examines the syscall number
  - Checks arguments for sanity
  - Performs operation
  - Stores result in v0
  - Restores callee-save registers
  - Performs a “return from syscall” instruction, which restores the privilege mode, SP and PC

Libraries and Wrappers

- Compilers do not emit SYSCALL instructions
  - They do not know the interface exposed by the OS
- Instead, applications are compiled with standard libraries, which provide “syscall wrappers”
  - printf() -> write(); malloc() -> sbrk(); recv(); open(); close(); …
- Wrappers are:
  - written in assembler,
  - internally issue a SYSCALL instruction,
  - pass arguments to kernel,
  - pass result back to calling application

- Advantages?
  – Portability



Exceptions

- System calls are control transfers to the OS, performed under the control of the user program
- Sometimes, need to transfer control to the OS at a time when the user program least expects it
  – Division by zero,
  – Alert from power supply that electricity is going out,
  – Alert from network device that a packet just arrived,
  – Clock notifying the processor that clock just ticked
- Some of these causes for interruption of execution have nothing to do with the user application
- Need a (slightly) different mechanism, that allows resuming the user application

Interrupts & Exceptions

- On an interrupt or exception
  - Switches the sp to the kernel stack
  - Saves the old (user) SP value
  - Saves the old (user) PC value
  - Saves the old privilege mode
  - Saves cause of the interrupt/privilege
  - Sets the new privilege mode to 1
  - Sets the new PC to the kernel interrupt/exception handler

Interrupts & Exceptions

- Kernel interrupt/exception handler handles the event
  - Saves all registers
  - Examines the cause
  - Performs operation required
  - Restores all registers
  - Performs a “return from interrupt” instruction, which restores the privilege mode, SP and PC

Syscall vs. Interrupt

- The differences lie in how they are initiated, and how much state needs to be saved and restored
- Syscall requires much less state saving
  - Caller-save registers are already saved by the application
- Interrupts typically require saving and restoring the full state of the processor
  - Because the application got struck by a lightning bolt without anticipating the control transfer

Terminology

- Trap
  - Any kind of a control transfer to the OS
- Syscall
  - Synchronous, program-initiated control transfer from user to the OS to obtain service from the OS
  - e.g. SYSCALL
- Exception
  - Asynchronous, program-initiated control transfer from user to the OS in response to an exceptional event
  - e.g. Divide by zero
- Interrupt
  - Asynchronous, device-initiated control transfer from user to the OS
  - e.g. Clock tick, network packet

Input / Output

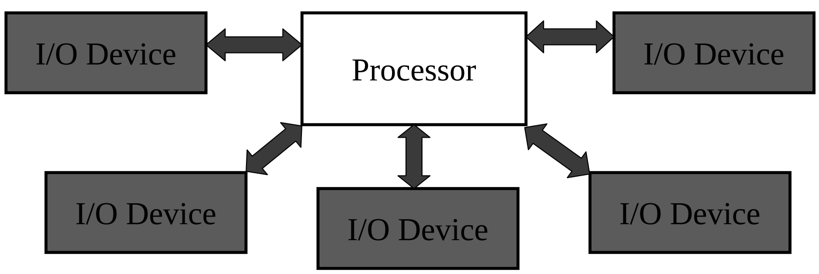
Challenge

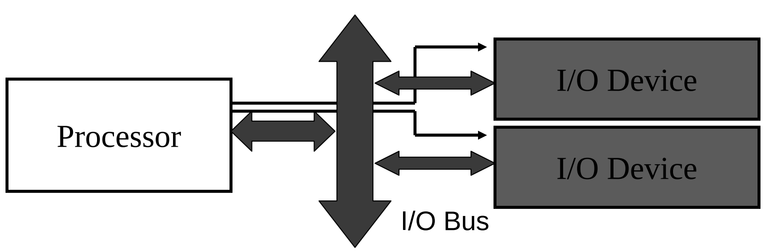
- How do we interface to other devices
  - Keyboard
  - Mouse
  - Disk
  - Network
  - Printer
  - …

Communication Interface

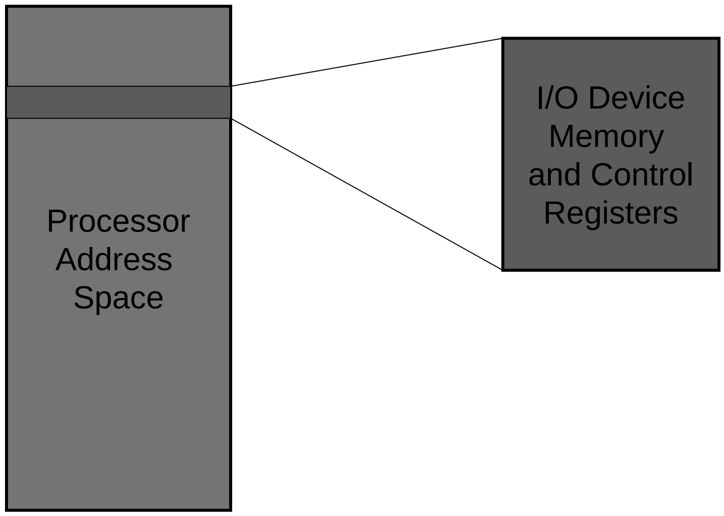
- Programmed I/O
  - CPU has dedicated, special instructions
  - CPU has additional output signals (I/O bus)
  - Instruction specifies device and operation
  - Protection: I/O instructions are privileged
- Memory-mapped I/O
  - Device communication goes over the memory bus
  - Reads/Writes to special addresses are converted into I/O operations
  - Each device appears as if it is part of the memory address space
  - Protection: device protected by the MMU

Programmed I/O





Memory-Mapped I/O



Memory-Mapped I/O

- Basic Idea – Make control registers and I/O device memory appear to be part of the system's main memory
  - Reads and writes to that region of the memory are translated by OS into device accesses
  - Easy to support variable numbers/types of devices
    - Just map them onto different regions of memory
    - Managing new devices is now memory allocation
    - Example: accessing memory on a PCMCIA card
      - Once card memory mapped into address space, just hand out pointers like conventional memory

Memory-Mapped I/O

- Basic Idea – Make control registers and I/O device memory appear to be part of the system’s main memory
  - Accessing I/O device registers and memory can be done by accessing data structures via the device pointers
    - Most device drivers are now written in C. Memory mapped I/O makes it possible without special changes to the compiler

Processor Memory Maps

- Define which regions of the address space are allocated to memory and/or I/O
- Some processors/operating systems use fixed memory maps
- Others use variable mappings
  - Unix/Linux has virtual memory system, maps I/O devices onto memory only when requested. This allows more flexibility for a system to accommodate variable number I/O devices