CS61C : Machine Structures

Lecture 7.1.1
VM I

2004-08-2

Kurt Meinz

inst.eecs.berkeley.edu/~cs61c

Outline

- Interrupts Review
- Virtual Memory

SPIM/Proj4 I/O Simulation

- Simulate 1 I/O device: memory-mapped terminal (keyboard + display)
  - Read from keyboard (receiver); 2 device regs
  - Writes to terminal (transmitter); 2 device regs

Receiver Control `0xffff0000`	Unused (00...00)	(I.E.) Ready
Receiver Data `0xffff0004`	Unused (00...00)	Received Byte
Transmitter Control `0xffff0008`	Unused (00...00)	(I.E.) Ready
Transmitter Data `0xffff000c`	Unused	Transmitted Byte

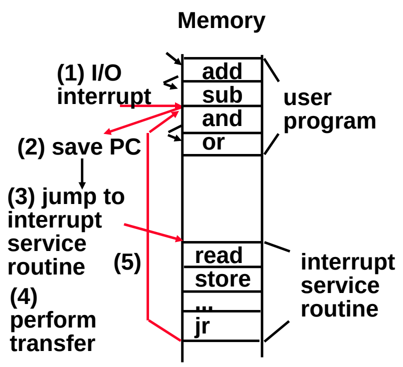
SPIM I/O

- Control register rightmost bit (0): Ready
  - Receiver: Ready==1 means character in Data Register not yet been read;
    1 ⇒ 0 when data is read from Data Reg
  - Transmitter: Ready==1 means transmitter is ready to accept a new character;
    0 ⇒ Transmitter still busy writing last char
    - I.E. bit discussed later
- Data register rightmost byte has data
  - Receiver: last char from keyboard; rest = 0
  - Transmitter: when write rightmost byte, writes char to display

Definitions for Clarification

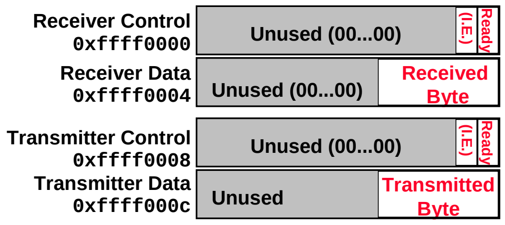
- Exception: signal marking that something "out of the ordinary" has happened and needs to be handled
- Interrupt: asynchronous exception
- Trap: synchronous exception
- Note: These are different from the book's definitions.
  - All I care about: that you know the difference between sync and async.

Interrupt Driven Data Transfer



SPIM I/O Simulation: Interrupt Driven I/O

- I.E. stands for Interrupt Enable
- Set Interrupt Enable bit to 1 have interrupt occur whenever Ready bit is set



Generalizing Interrupts

- We can handle all sorts of exceptions with interrupts.
- Big idea: jump to handler that knows what to do with each interrupt, then jump back
- Our types: syscall, overflow, mmio ready.

OS: I/O Requirements

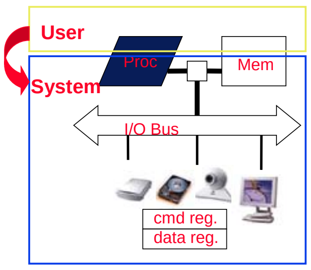
- The OS must be able to prevent:
  - The user program from communicating with the I/O device directly
- If user programs could perform I/O directly:
  - No protection to the shared I/O resources
- 3 types of communication are required:
  - The OS must be able to give commands to the I/O devices
  - The I/O device notify OS when the I/O device has completed an operation or an error
  - Data transfers between memory and I/O device

Kernel/User Mode

- Generally restrict device access to OS
- HOW?
- Add a "mode bit" to the machine: K/U
- Only allow SW in "kernel mode" to access device registers
- If user programs could access device directly?
  - could destroy each others data, ...
  - might break the devices, …

Crossing the System Boundary

- System loads user program into memory and 'gives' it use of the processor
- Switch back
  - SYSCALL
    - request service
    - I/O
  - TRAP (overflow)
  - Interrupt



Syscall

- How does user invoke the OS?
  - `syscall` instruction: invoke the kernel (Go to 0x80000080, change to kernel mode)
  - By software convention, `$v0` has system service requested: OS performs request

Handling a Single Interrupt (1/3)

- An interrupt has occurred, then what?
  - Automatically, the hardware copies PC into EPC ($14 on cop0) and puts correct code into Cause Reg ($13 on cop0)
  - Automatically, PC is set to 0x80000080, process enters kernel mode, and interrupt handler code begins execution
  - Interrupt Handler code: Checks Cause Register (bits 5 to 2 of $13 in cop0) and jumps to portion of interrupt handler which handles the current exception

Handling a Single Interrupt (2/3)

- Sample Interrupt Handler Code

```
.text 0x80000080
mfc0  $k0,$13  # $13 is Cause Reg
sll   $k0,$k0,26  # isolate
srl   $k0,$k0,28  #   Cause bits
```

- Notes:
  - Don't need to save `$k0` or `$k1`
    - MIPS software convention to provide temp registers for operating system routines
    - Application software cannot use them
  - Can only work on CPU, not on cop0

Handling a Single Interrupt (3/3)

  - When the interrupt is handled, copy the value from EPC to the PC.
  - Call instruction `rfe` (return from exception), which will return process to user mode and reset state to the way it was before the interrupt
- What about multiple interrupts?

Modified Interrupt Handler (1/3)

- Problem: When an interrupt comes in, EPC and Cause get overwritten *immediately* by hardware. Lost EPC means loss of user program.
- Solution: Modify interrupt handler. When first interrupt comes in:
  - disable interrupts (in Status Register)
  - save EPC, Cause, Status and Priority Level on Exception Stack
  - re-enable interrupts
  - continue handling current interrupt

Modified Interrupt Handler (2/3)

- When next (or any later) interrupt comes in:
  - interrupt the first one
  - disable interrupts (in Status Register)
  - save EPC, Cause, Status and Priority Level (and maybe more) on Exception Stack
  - determine whether new one preempts old one
    - if no, re-enable interrupts and continue with old one
    - if yes, may have to save state for the old one, then re-enable interrupts, then handle new one

Modified Interrupt Handler (3/3)

- Notes:
  - Disabling interrupts is dangerous
  - So we disable them for as short a time as possible: long enough to save vital info onto Exception Stack
- This new scheme allows us to handle many interrupts effectively.

Details not covered

- MIPS has a field to record all pending interrupts so that none are lost while interrupts are off; in Cause register
- The Interrupt Priority Level that the CPU is running at is set in memory
- MIPS has a field in that can mask interrupts of different priorities to implement priority levels; in Status register
- MIPS has limited nesting of saving KU,IE bits to recall in case higher priority interrupts; in Status Register

Things to Remember

- Kernel Mode v. User Mode: OS can provide security and fairness
- Syscall: provides a way for a programmer to avoid having to know details of each I/O device
- To be acceptable, interrupt handler must:
  - service all interrupts (no drops)
  - service by priority
  - make all users believe that no interrupt has occurred

Improving Data Transfer Performance

- Thus far: OS give commands to I/O, I/O device notify OS when the I/O device completed operation or an error
- What about data transfer to I/O device?
  - Processor busy doing loads/stores between memory and I/O Data Register
- Ideal: specify the block of memory to be transferred, be notified on completion?
  - Direct Memory Access (DMA) : a simple computer transfers a block of data to/from memory and I/O, interrupting upon done

Example: code in DMA controller

- DMA code from Disk Device to Memory

```
         .data
Count:   .word  4096
Start:   .space 4096
         .text
Initial: lw $s0, Count # No. chars
         la $s1, Start # @next char
Wait:    lw $s2, DiskControl
         andi $s2,$s2,1 # select Ready
         beq $s2,$0,Wait # spinwait
         lb $t0, DiskData # get byte
         sb $t0, 0($s1) # transfer
         addiu $s0,$s0,-1 # Count--
         addiu $s1,$s1,1  # Start++
         bne   $s0,$0,Wait # next char
```

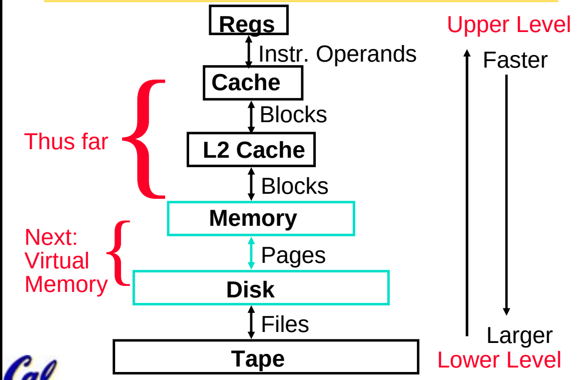
- DMA "computer" in parallel with CPU

VM

Generalized Caching

- We've discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - Filesystem cache
  - Web page cache
  - Game Theory databases / tablebases
  - Software memoization
  - Others?
- Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.

Another View of the Memory Hierarchy



Memory Hierarchy Requirements

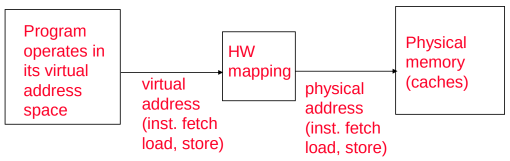
- What else might we want from our memory subsystem? …
  - Share memory between multiple processes but still provide protection – don't let one program read/write memory from another
    - Emacs on star
  - Address space – give each process the illusion that it has its own private memory
    - Implicit in our model of a linker
- Called Virtual Memory

Virtual Memory Big Ideas

- Each address that a program uses (pc, $sp, $gp, .data, etc) is fake.
- Processor inserts new step:
  - Every time we reference an address (in IF or MEM) …
  - Translate fake address to real one.

virtual physical

VM Ramifications


Program operates in its virtual address space
virtual address (inst. fetch load, store)
HW mapping
physical address (inst. fetch load, store)
Physical memory (caches)

- Immediate consequences:
  - Each program can operate in isolation!
  - OS can decide where and when each goes in memory!
  - HW/OS can grant different rights to different processes on same chunk of physical mem!
- Big question:
  - How do we manage the VA➔PA mappings?

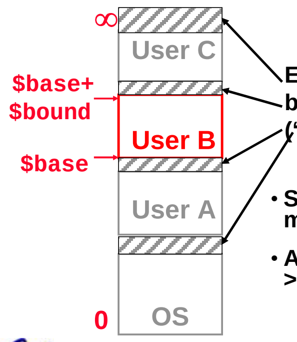
Analogy

- Book title like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call number
- On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights

VM

- Ok, now how do we implement it?
- Simple solution:
  - Linker assumes start addr at 0x0.
  - Each process has a $base and $bound:
    - $base: start of physical address space
    - $bound: size of physical address space
  - Algorithms:
    - VA➔PA Mapping: PA = VA + $base
    - Bounds check: VA < $bound

Simple Example: Base and Bound Reg



☹ so what's wrong?

Enough space for User D, but discontinuous ("fragmentation problem")

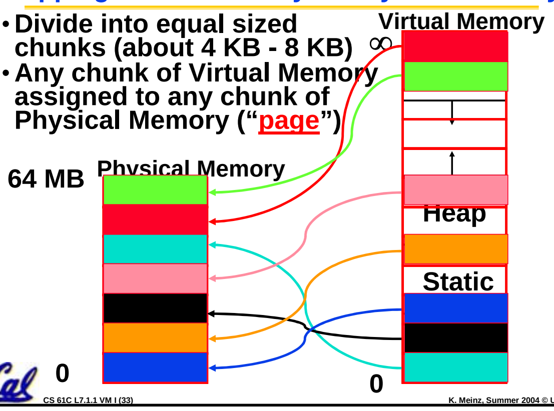
- Same flaws as freelist malloc!
- Also: what if process size > mem
- What to do??

VM Observations

- Working set of process is small, but distributed all over address space ➔
  - Arbitrary mapping function,
    - keep working set in memory
    - rest on disk or unallocated.
- Fragmentation comes from variable-sized physical address spaces
  - Allocate physical memory in fixed-sized chunks (1 mapping per chunk)
  - FA placement of chunks
    - i.e. any V chunk of any process can map to any P chunk of memory.

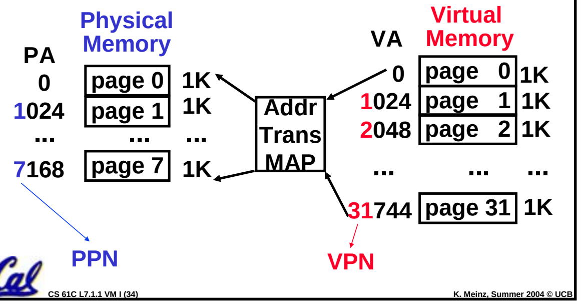
Mapping Virtual Memory to Physical Memory

- Divide into equal sized chunks (about 4 KB - 8 KB)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")



Paging Organization

1KB Pages	Page is unit of mapping	Page also unit of transfer from disk to physical memory



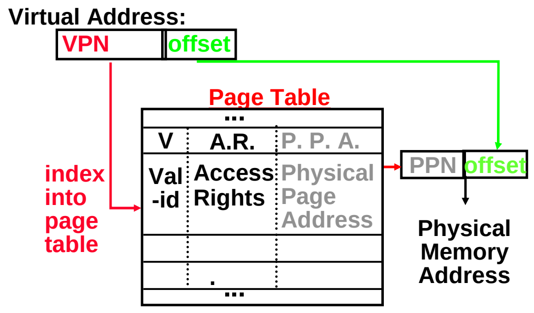
Virtual Memory Mapping Function

Page Number	Offset

- Use table lookup ("Page Table") for mappings: V Page number is index
- Mapping Function
  - Physical Offset = Virtual Offset
  - Physical Page Number = PageTable[Virtual Page Number]

FYI: P.P.N. also called "Page Frame" or "Frame #".

Address Mapping: Page Table



Page Table located in physical memory

Page Table

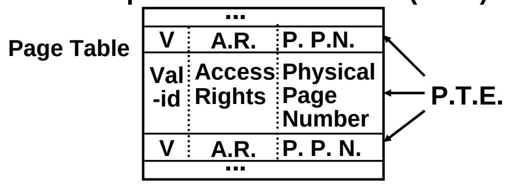
- A page table: mapping function
  - There are several different ways, all up to the operating system, to keep this data around.
  - Each process running in the operating system has its own page table
    - Historically, OS changes page tables by changing contents of Page Table Base Register
      - Not anymore! We'll explain soon.

Requirements revisited

- Remember the motivation for VM:
- Sharing memory with protection
  - Different physical pages can be allocated to different processes (sharing)
  - A process can only touch pages in its own page table (protection)
- Separate address spaces
  - Since programs work only with virtual addresses, different programs can have different data/code at the same address!

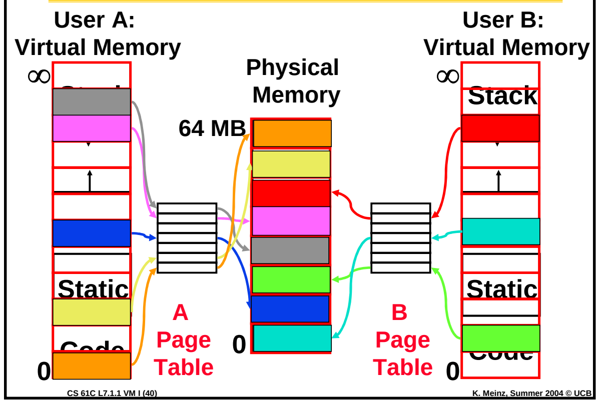
Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid (V = 0)



- If valid, also check if have permission to use page: Access Rights (A.R.) may be Read Only, Read/Write, Executable

Paging/Virtual Memory Multiple Processes



Comparing the 2 levels of hierarchy

Cache Version	Virtual Memory vers.
Block or Line	Page
Miss	Page Fault
Block Size: 32-64B	Page Size: 4K-8KB
Placement: Direct Mapped, N-way Set Associative	Fully Associative
Replacement: LRU or Random	Least Recently Used (LRU)
Write Thru or Back	Write Back

Notes on Page Table

- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Will add details, but Page Table is essence of Virtual Memory