Supporting an Operating System

CSE 378 Spring 2009

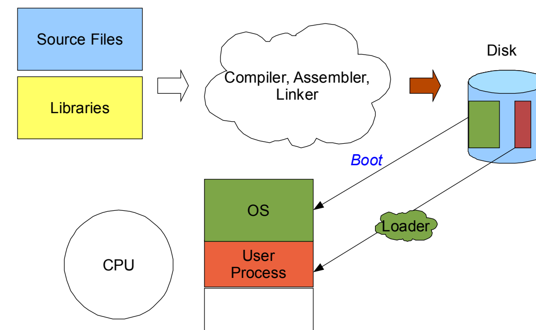
Topics

- We've seen how the ISA supports individual user applications
  - You know how to “compile” C code, say, to assembler
  - The instruction set we've seen is enough to implement all of C (for instance)
- *How do computer systems work?*
  - *What's the role of the ISA?*
  - *What's the role of the OS?*
  - *How are these related to the roles of the compiler, linker, loader?*

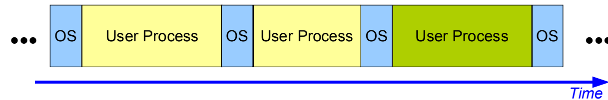
Roles of the OS

- Resource management
  - Allocate CPU time, memory, disk space, disk use, network bandwidth, etc.
- Protection
  - Limit access to resources to authorized users
- Key notion to today's topics: *process*
  - A process is a program in execution

The Hardware/Software Interface



Background: View of Runtime



- Use of the CPU alternates between the OS and user processes
- On entry to OS, process *state* is saved
  - PC and register contents written to save area in OS memory
- Process *dispatch* is handing CPU back to a user process
  - Reload register contents, and jump back to saved PC

Context For Today's Topics

- The goal is "a computer system" that can do the sorts of things you're used to:
  - I/O
  - Run more than one app at a time, etc.
- Achieving this high level goal is an orchestration of:
  - Software: the OS, linker, compiler, libraries, etc. have cooperating roles
  - ISA: must provide some key functionality
    - This is particularly the case for allowing the OS to do things its responsible for
  - Hardware: must implement the ISA

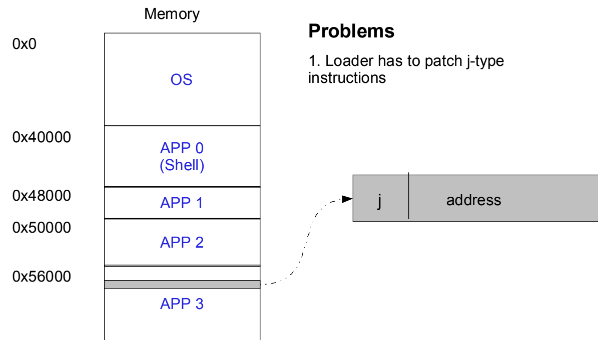
Today vs. Tomorrow

- For concreteness, we'll be talking today about the Cebollita architecture
  - It contains most concepts required to understand a current (desktop) computer
  - The techniques used lie somewhere in the past on the computer's evolutionary tree
- We'll look at more modern virtual memory concepts a little later in the course
  - More sophisticated OS facilities and implementations are CSE 451

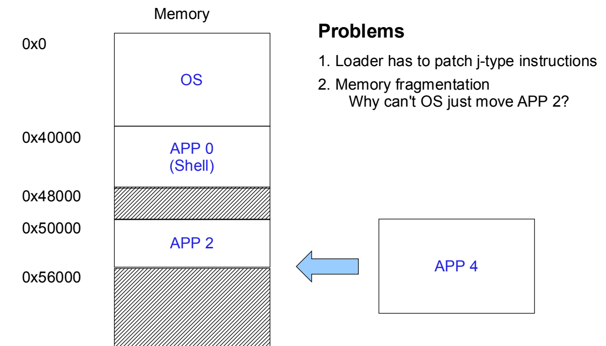
Topics

- Today:
  - Address Translation (Virtual Memory)
    - Managing main memory
  - Exceptions
    - Managing the CPU
  - *Memory Mapped Devices*
    - *An implementation technique*
- Friday:
  - I/O
  - System calls

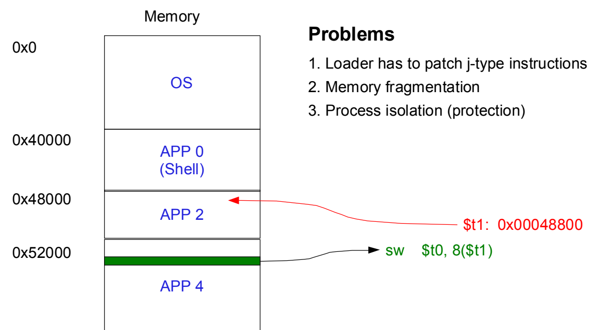
Address Translation: Why?



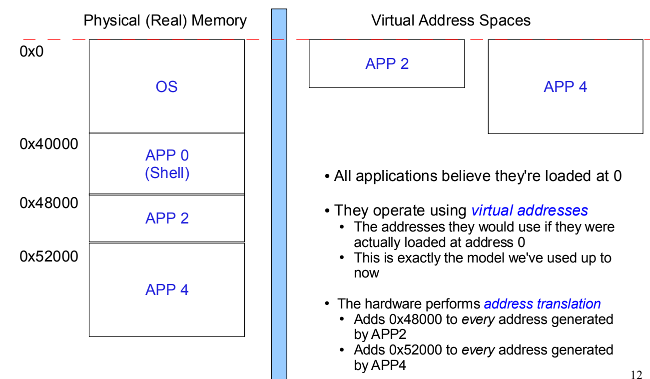
Address Translation: Why?



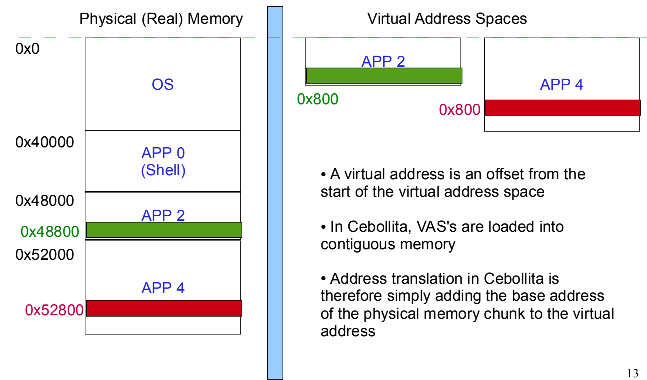
Address Translation: Why?



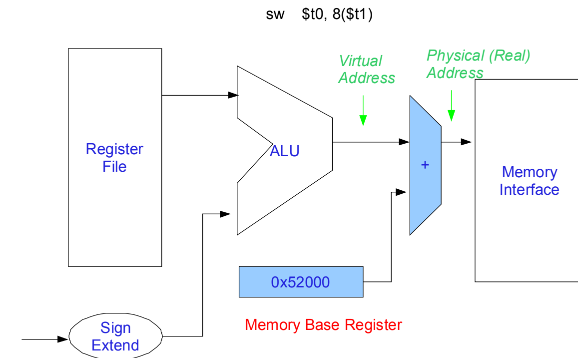
Cebollita Address Translation



Cebollita Address Translation



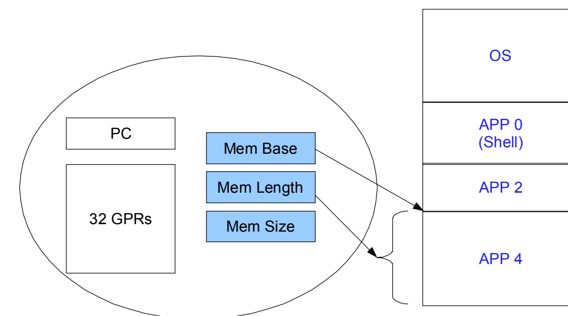
(Partial) Implementation



Address Translation: Cebollita ISA

- The Cebollita ISA specifies:
  - How (and when) address translation is performed
  - Three (new) registers:
    - Memory Base Register
      - (Real) base address of process
      - *real address = virtual address + memory base*
    - Memory Length Register
      - Size of process image (in bytes)
      - *If (virtual address > memory length) then* ERROR
    - Memory Size Register
      - Size of real (physical) memory
      - A convenience for the OS

Updated View of CPU



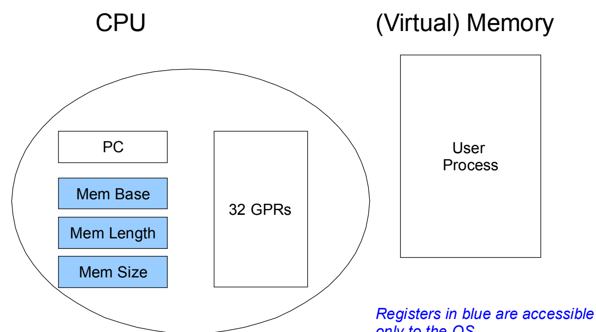
Address Translation: OS

- First pass overview:
  - The loader (OS) finds enough contiguous memory to hold the new process image
  - The process image is read from disk into that memory
  - The memory base and length registers are set
    - *Problem: What happens when mem base register is set?*
  - The OS jumps to the entry point of the new process

Address Translation: Protection

- Process isolation is the most important benefit of (the Cebollita) address translation
- Process isolation requires checking access permission per-instruction
  - There must be hardware support
- Virtual addresses provide isolation by making it impossible for one process to *name* the memory belonging to another process
  - There is no memory address a process can utter that names the physical memory used by another process (so long as the OS makes no mistakes in setting the base register)

Updated View of ISA



Registers in blue are accessible only to the OS.
(Why?)
(How?)

The Exception Architecture

- The exception architecture defines how transitions between running user code and the OS take place
- Example reasons to want to transition from the user app to the OS
  - The user app wants to do something only the OS can do
    - E.g., read or write disk
    - Invoking a procedure in the OS is a "system call" (syscall)
  - The user app has made some kind of mistake
    - E.g., referenced a virtual address that is larger than the size of its address space
    - These are called "exceptions"
  - Some device needs attention
    - E.g., a packet has arrived off the network
    - These are called "interrupts"

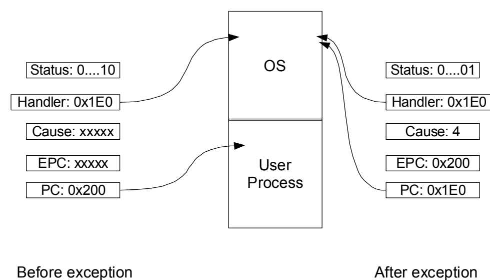
The Cebollita Exception Architecture

- The ISA specifies four new registers:
  - EPC: the PC when the exception/interrupt occurred
  - Handler Address: what the PC should be set to
    - This is the address of the OS's *trap handler* routine
  - Cause: indicates what happened
    - E.g., 4 means addressing exception; 16 means overflow; 2 means disk needs attention
  - Status: a bit mask
    - Privilege bit: set to 1 if the OS is currently running; 0 if a user app
    - Interrupt enable bit: set to 0 to disable "raising exceptions/interrupts

Exceptions: ISA

- When an exception or interrupt occurs, if the interrupt enable bit is 0, ignore (or leave pending)
- Otherwise, do all this in the current cycle:
  - Save the current value of the PC in register EPC
  - Set the PC to the value of the trap handler address register
  - Set the value of the Cause register
  - Update the status register:
    - Set the privilege bit on (because the next instruction to be executed is the trap handler, in the OS)
    - Set the interrupt enable bit off

Exceptions: ISA Mechanics



Before exception

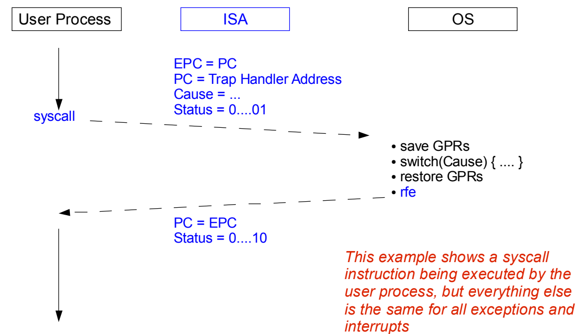
After exception

This is the only way for a user process to "enter" the OS

Exceptions: OS Trap Handler

- First thing trap handler does is save the 31 general purpose registers (GPRs), plus the EPC
- Next, it looks at the cause register, and decides what to do:
  - E.g., if it's a syscall, it branches to the syscall handler code
  - If it's a fatal program error, it terminates the process, etc.
- When done, OS picks a *runnable* process and *dispatches* it
  - Loads 31 GPRs using saved register values
  - Causes, simultaneously, a branch to where the process was last executing (saved EPC) and change of Status register to unprivileged and interrupts enabled
    - There's a special instruction for this: rfe

Another View of Mechanics

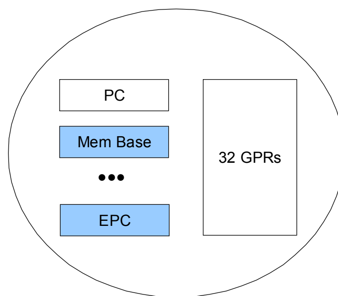


This example shows a syscall instruction being executed by the user process, but everything else is the same for all exceptions and interrupts

(Difficult) Comprehension Check

Do OS instructions issue real addresses or virtual addresses?

Summary to this Point

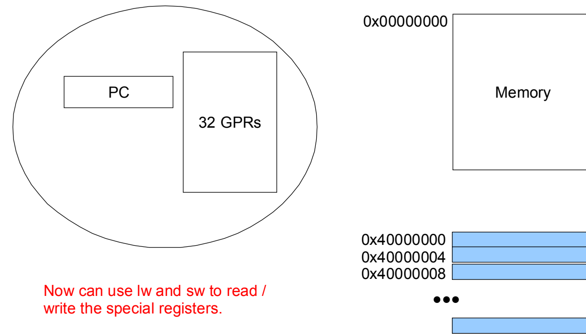


- Requirements:
  - Must have instructions to read/write special registers
    - The OS needs them
  - Can't let user processes read/write special registers
- How can we do that?

One Approach

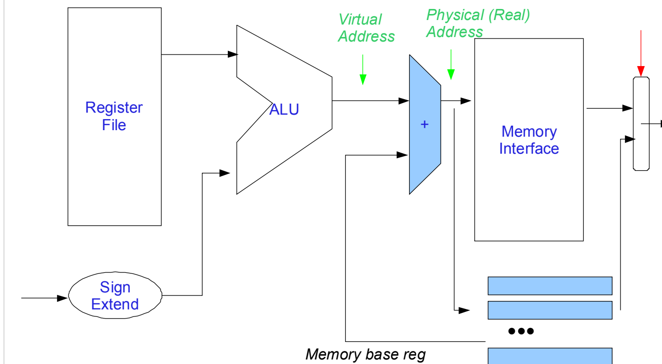
- Define new instructions for reading / writing special registers
  - ReadSpecial $8, $membase
  - WriteSpecial $membase, $8
- The new instructions cause an error unless the CPU is operating in privileged mode
  - *"Privileged instruction exception"*
- Et voila...
- If you look at the MIPS Programmer Manual, you'll see the bulk of the instructions there are of this sort

Cebollita ISA: Memory Mapped Devices



Now can use lw and sw to read / write the special registers.

Idea of the Implementation



Why Does This Work?

- OS can read / write these registers
- If a user process tries to read or write them:
  - It has to construct and use an address no smaller than 0x40000000
  - That causes an addressing exception
    - The memory allocated to it is smaller than that
      → The address it issued is larger than the memory length register
      → Addressing exception
- A side-effect of this design is that no virtual address space can be bigger than 0x40000000 bytes long
  - Not a worry for us...
- Note: Physical memory could still be 2^{32} bytes
  - There'd be a chunk "missing"

Summary

- Processes issue virtual addresses; hardware performs address translation to get real address
  - Allows flexibility in use of main memory
  - Provides process isolation / protection
- CPU have privileged and unprivileged modes
  - Allows OS to do things user processes can't
- Exception architecture ensures that:
  - The only way to go into privileged mode is to enter the OS
  - The only place to enter the OS is the trap handler
- Memory mapped devices exploits these properties to provide access to control registers without needing new instructions