Outline for Today

• Objective of Today’s Lecture: Review of computer architecture
• C++ Pitfalls

CPS 104++:
Almost Everything You Wanted to Know About Operating System’s interaction with Architecture but were Afraid to Ask

What does an OS do?

• Manipulate and Control Programs
  – Place and remove them from memory (treat as data)
  – Start them, “Freeze” (stop) them, “thaw” (restart) them
  – Set Memory management hardware to monitor their use of memory

• Need to know (from 104)
  – Programs consist of instructions, each a “word” (32 bits on MIPS)
  – Programs and data reside in memory
  – Memory is an array of bytes (2^32 on MIPS)
  – Program, and data structures reside in “blocks” of memory (sequentially addressed bytes) (a byte is 8 bits, 4 bytes per word
  – To freeze a program, stop its execution, and save its “state”
  – State: all registers, memory, status flags, mode bit, memory management tables, interrupt status bits that the program can access, or which affect how the program executes indirectly
OS Services: I/O

• OS provides I/O device ABSTRACTIONS to programs
  – Files
  – Communication channels
• You need to know
  – Something about the underlying I/O hardware
    – Disks
      • Large number of addressable records, each holding maybe 8K bytes
      • Records placed on disk in concentric tracks, on many physical sides
      – Seek to move arm to proper "cylinder"
      – Wait while proper record spins under R/W head
      – Use electronics to select which surface to Read/Write
      – Must operate on whole record
    • OS defines logical structure on top of this, to present file abstraction

Basic Storyline – Evolution of HW Support

• The bare machine: instruction cycle, register state, DMA I/O, interrupts.
• Add an OS, to safely and efficiently start and execute user programs
• But user programs might damage the OS
  – better restrict user code from having direct access to (at least ) I/O
  – Need: protected instructions, kernel/user modes, system calls.
• Add sharing among multiple users
  – Need: memory protection, timers, instructions to assist synchronization, process abstraction.

The Big Picture

• The Five Classic Components of a Computer

Von Neumann machine
What do we need to know about the Processor?

• Size (# bits) of effective memory addresses that can be generated by the program and therefore, the amount of memory that can be accessed.
• Information that is crucial: process state or execution context describing the execution of a program (e.g. program counter, stack pointer). This is stuff that needs to be saved and restored on context switch.
• When the execution cycle can be interrupted. What is an indivisible operation in given architecture?

A "Typical" RISC Processor

• 32-bit fixed format instruction
• 32 (32,64)-bit GPR (general purpose registers)
• 32 Floating-point registers (not used by OS, but part of state)
• Status registers (condition codes)
• Load/Store Architecture
  – Only accesses to memory are with load/store instructions
  – All other operations use registers
  – addressing mode: base register + 16-bit offset
• Not Intel x86 architecture!

Executing a Program

• Thread of control (program counter)

• Basic steps for program execution (execution cycle)
  – fetch instruction from Memory(PC), decode it
  – execute the instruction (fetching any operands, storing result, setting condition codes, etc.)
  – increment PC (unless jump)
An Abstract View of the Implementation

Program Stack

• Some known register is stack pointer
• Stack is used for
  – passing parameters (function, method, procedure, subroutine)
  – storing local variables

Variable Storage Class in C/C++

• Local: Will vanish when current procedure returns + can be accessed only by this procedure
  – Declaration is inside the procedure/method
• Global: Will remain around till program exits + are accessible to all procedures
• Heap: Will remain around until “deleted”. Allocated by “new”. Accessible through the pointer that “new” returns

CPS I 10

Alvin R. Lebeck
Memory Pitfalls

- Allocating space for a pointer, but not creating the data the pointer points to, or not setting the pointer to point anywhere
  - `Int *x;` // Reserves (allocates) space for a pointer to an int. // But THIS puts nothing into the pointer – the pointer // points to an unknown area of memory
  - `Int *x[50];` // Reserves space for 50 pointers, but fills in none of // them.
  - `Int *x = new int[50];` // Reserves space on the heap for an array // of 50 ints. None of these ints are initialized.
  - `Int **x = new *int[50];` // Reserves space on the heap for 50 // pointers to int.
- I’m NOT certain that these constructs work as I’ve stated. Check them, and tell me, so I can correct this slide!
- Allocating a local variable, returning a pointer to it.

What do we need to know about the Processor?

- **Size (in bits) of effective memory addresses** that can be generated by the program and therefore, the amount of memory that can be accessed.
- Information that is crucial: **process state or execution context** describing the execution of a program (e.g. program counter, stack pointer). This is stuff that needs to be saved and restored on context switch.
- When the execution cycle can be interrupted. What is an indivisible operation in given architecture?

Interrupts are a Key Mechanism
Role of Interrupts in I/O

So, the program needs to access an I/O device ...

- Start an I/O operation (special instructions or memory-mapped I/O)
- Device controller performs the operation asynchronously (in parallel with) CPU processing (between controller's buffer & device).
- If DMA, data transferred between controller's buffer and memory without CPU involvement.
- Interrupt signals I/O completion when device is done.

First instance of concurrency we've encountered - I/O Overlap

Interrupts and Exceptions

- Unnatural change in control flow
- Interrupt is external event
  - devices: disk, network, keyboard, etc.
  - clock for timeslicing
  - These are useful events. OS must do something when they occur.
- Exception announces potential problem with program. OS must handle these, also.
  - segmentation fault
  - bus error
  - divide by 0
  - Don't want my bug to crash the entire machine
  - page fault (virtual memory...)

CPU handles interrupt

- CPU stops current operation *, saves current program counter and other processor state needed to continue at interrupted instruction.
- Hardware accesses vector table in memory, jumps to address of appropriate interrupt service routine for this event.
- Service routine (handler) does what needs to be done.
- Restores saved state at interrupted instruction

* At what point in the execution cycle does this make sense?
** Need someplace to save it!
Data structures in OS kernel.
An Execution Context

- The state of the CPU associated with a thread of control (process)
  - general purpose registers (integer and floating point)
  - status registers (e.g., condition codes)
  - program counter, stack pointer
- Need to be able to switch between contexts
  - better utilization of machine (overlap I/O of one process with computation of another)
  - timeslicing: sharing the machine among many processes
  - different modes (Kernel v.s. user)

Handling an Interrupt/Exception

- Invoke specific kernel routine based on type of interrupt
  - interrupt/exception handler
- Must determine what caused interrupt
  - could use software to examine each device
  - PC set to address of interrupt handler by the hardware when interrupt event occurs (fixed location)
- Vectored Interrupts
  - PC = interrupt_table[i]
  - kernel initializes table at boot time
- Clear the interrupt
- May return from interrupt (RETT) to different process (e.g., context switch)

Context Switches

- Save current execution context
  - Save registers and program counter
  - information about the context (e.g., ready, blocked)
- Restore other context
- Need data structures in kernel to support this
  - process control block
- Why do we context switch?
  - Timeslicing: HW clock tick
  - I/O begin and/or end
- How do we know these events occur?
  - Interrupts...
Crossing Protection Boundaries

- For a user to do something "privileged", it must invoke an OS procedure providing that service. How?

  - System Calls
    - special trap instruction that causes an exception which vectors to a kernel handler
    - parameters indicate which system routine called
    - The interrupt action (done by hardware) changes both PC and Kernel/User mode bit.

A System Call

- Special Instruction to change modes and invoke service
  - read/write I/O device
  - create new process
- Invokes specific kernel routine based on argument
- kernel defined interface
- May return from trap to different process (e.g., context switch)
- RETT, instruction to return to user process

User / Kernel Modes

- Hardware support to differentiate between what we'll allow user code to do by itself (user mode) and what we'll have the OS do (kernel mode).
- Mode indicated by status bit in a protected processor register.
- Privileged instructions can only be executed in kernel mode (I/O instructions).
X Execution Mode

- What if interrupt occurs while in interrupt handler?
  - Problem: Could lose information for one interrupt
    - Clear of interrupt #1, clears both #1 and #2
  - Solution: disable interrupts (done automatically for at least a few instructions)

- Disabling interrupts is a protected operation
  - Only the kernel can execute it
  - user v.s. kernel mode
    - mode bit in CPU status register

- Other protected operations
  - Installing interrupt handlers (placed in OS-owned memory)
  - Manipulating CPU state (saving/restoring status registers)
  - Changing table that control memory access

- Changing modes
  - Interrupts
  - System calls (trap instruction)

X CPU Handles Interrupt (with User Code)

- CPU stops current operation, goes into kernel mode, saves current program counter and other processor state needed to continue at interrupted instruction.
- Hardware accesses vector table, in memory, jumps to address of appropriate interrupt handler for this event.
- Handler (software) does what needs to be done.
- Restores saved state at interrupted instruction.
  Returns to user mode.

Multiple User Programs

- Sharing system resources requires that we protect programs from other incorrect programs.
  - Protect from a bad user program walking all over the memory space of the OS and other user programs (memory protection).
  - Protect from runaway user programs never relinquishing the CPU (e.g., infinite loops) (timers).
  - Preserving the illusion of non-interruptible instruction sequences (synchronization mechanisms - ability to disable/enable interrupts, special "atomic" instructions).
CPU Handles Interrupt (Multiple Users)

- CPU stops current operation, goes into kernel mode, saves current program counter and other processor state needed to continue at interrupted instruction.
- Hardware accesses vector table in memory, jumps to address of appropriate interrupt handler for this event.
- Handler does what needs to be done.
- Restores saved state at interrupted instruction (with multiple processes, it is the saved state of the process that the scheduler selects to run next). Returns to user mode.

Timer Operation

- Timer set to generate an interrupt in a given time.
- OS uses it to regain control from user code.
- Sets timer before transferring to user code.
- When time expires, the executing program is interrupted and the OS is back in control.
- Prevents monopolization of CPU
- Setting timer is privileged.

Extra Credit Problem

- Two parallel loops run so that they share variable C, but not other variables. The timer switches CPU attention from one to the other at random.
  Each loop is:
  ```
  for (i=0; i<10; i++) C+=1;
  ```
- Variable C is initially 0. When both programs finish, C’s value is 12. How can this happen?
Issues of Sharing Physical Memory

Protection:

• Simplest scheme uses base and limit registers, loaded by OS (privileged operation) before starting program.

• Issuing an address out of range causes an exception.

Significance?

Sharing Physical Memory

Allocation

• Disjoint programs have to occupy different cells in memory (or the same cells at different times - swapping*)

• Memory management has to determine where, when, and how** code and data are loaded into memory

* Where is it when it isn’t in memory? Memory Hierarchy

**What HW support is available in architecture? MMU

Memory Hierarchy 101

Very fast 1ns clock
Multiple Instructions per cycle

SRAM, Fast, Small
Expensive “Cache”

DRAM, Slow, Big,Cheap
(called physical or main)

Magnetic, Really Slow,
Really Big, Really Cheap

=> Cost Effective Memory System (Price/Performance)
Memory Hierarchy 101

- **SRAM**, Fast, Small
  - Expensive “Cache”
- **DRAM**, Slow, Big, Cheap
  - (called physical or main)
- **Magnetic**, Really Slow, Really Big, Really Cheap

What’s the average memory access time?

Role of MMU Hardware and OS

- VM address translation must be very fast (on average).
  - Every instruction includes one or two memory references.
  - (including the reference to the instruction itself)
- VM translation is supported in hardware by a **Memory Management Unit or MMU**.
  - The addressing model is defined by the CPU architecture.
  - The MMU itself is an integral part of the CPU.
- The role of the OS is to install the virtual-physical mapping and intervene if the MMU reports a violation.

Virtual Address Translation

**Example**: typical 32-bit architecture with 8KB pages.

Virtual address translation maps a **virtual page number** (VPN) to a **physical page frame number** (PFN): the rest is easy.

Deliver exception to OS if translation is not valid and accessible in requested mode.
Page Table Mapping

Concurrency
- Multiple things happening simultaneously
  - logically or physically
- Causes
  - Interrupts
  - Voluntary context switch (system call/trap)
  - Shared memory multiprocessor

The Trouble with Concurrency
- Two threads (T1,T2) in one address space or two processes in the kernel
- One counter (in memory)
  - Result NOT correct
Solution: Atomic Sequence of Instructions

- **Atomic Sequence**
  - Appears to execute to completion without any intervening operations
  - Result is correct

HW Support for Atomic Operations

- Could provide direct support in HW
  - Atomic increment
    - Insert node into sorted list??
- Just provide low level primitives to construct atomic sequences
  - called synchronization primitives
    - LOCK(counter->lock);
    - counter->value = counter->value + 1;
    - UNLOCK(counter->lock);
- `test&set (x)` instruction: returns previous value of x and sets x to "1"
  - LOCK(x) => while (test&set(x));
  - UNLOCK(x) => x = 0;

Summary

- Fetch, Execute Cycle
  - Thread of control, indivisible operations, dynamic memory reference behavior
- Execution Context
  - What needs to be saved on context switch
- Exceptions and Interrupts
  - What triggers most OS actions
- Mode bit, Privileged Instructions
  - Kernel/user program distinction, privileges
- Memory Hierarchy
  - MMU, access characteristics of levels
- Concurrency
  - Atomic sequences, synchronization