Outline for Today

• Announcements
  – How office hours for TA and UTAs work: announced around assignment schedule
  – Those who didn’t get your photo taken last time, do it today - signup sheet, associating your pic with frame number on disk. Tips for “good” photos.
  – Groups - if you haven’t sent me email DO IT TODAY!
  – If you weren’t here last time, fill out who’s who questionnaire (doc format on the web page - soon).
  – Plug for Soph. and Juniors to do undergrad research experiences - CRA awards

• Lecture: Review of computer architecture

Basic Storyline

• (Review) Computing from the (purely) architectural point of view: instruction cycle, register state, DMA I/O, interrupts.
• Introduce execution of user programs into the picture and we want to restrict user code from having direct access to (at least) I/O -> protected instructions, kernel/user modes, system calls.
• Add sharing among multiple users -> memory protection, timers, instructions to assist synchronization, process abstraction.

The Big Picture

• The Five Classic Components of a Computer
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)
- 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!

Example: MIPS

<table>
<thead>
<tr>
<th>Register-Register</th>
<th>31 26 25 21 10 16 12 8 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register-Immediate</th>
<th>31 26 25 21 10 16 12 8 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
</tr>
</tbody>
</table>

Branch, Load, Store

<table>
<thead>
<tr>
<th>31 26 25 21 10 16 12 8 5 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>31 26 25 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
</tr>
</tbody>
</table>

So, how many memory locations can we address? Can we tell how much memory the machine has?

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 cond codes, etc.)
  - increment PC (unless jump)

An Abstract View of the Implementation

Is there something missing here?
**Program Stack**

- A Stack is used for:
  - passing parameters (function, method, procedure, subroutine)
  - storing local variables
- Well defined register is stack pointer

A stack frame (Activation Record)

- Return results
- Return Address
- Arg1
- Arg2
- Local variables

- First few return results and arguments are mapped to specific registers (calling conventions)

---

**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?

---

**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, must do something when they occur.
- Exception is potential problem with program
  - 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.
- Accessing vector table, in memory, it jumps to address of appropriate interrupt service routine for this event.
- 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
- 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

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 protected processor register.
- Privileged instructions can only be executed in kernel mode (I/O instructions).
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
- Disabling interrupts is a protected operation
  - Only the kernel can execute it
  - User vs. kernel mode
  - Mode bit in CPU status register
- Other protected operations
  - Installing interrupt handlers
  - Manipulating CPU state (saving/restoring status registers)
- Changing modes
  - Interrupts
  - System calls (trap instruction)

CPU handles interrupt

- CPU stops current operation, goes into kernel mode, saves current program counter and other processor state needed to continue at interrupted instruction.
- Accessing vector table, in memory, jump to address of appropriate interrupt service routine for this event.
- Handler 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

- CPU stops current operation, goes into kernel mode, saves current program counter and other processor state needed to continue at interrupted instruction.
- Accessing vector table, in memory, jump to address of appropriate interrupt service routine 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.
- Setting timer is privileged.

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.
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?
**What HW support is available in architecture?

Memory Hierarchy 101

Very fast 1ns clock
Multiple Instructions per cycle

SRAM, Fast, Small
Expensive

“CPU-DRAM gap”
memory system architecture
(CPS 104)

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

Magnetic, Really Slow,
Really Big, Really Cheap

=> Cost Effective Memory System (Price/Performance)

Role of MMU Hardware and OS

• VM address translation must be very cheap (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

TLB serves as a cache of PT entries.

Physical Memory
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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld r2, count</td>
<td>ld r2, count</td>
</tr>
<tr>
<td>add r1, r2, r3</td>
<td>add r1, r2, r3</td>
</tr>
<tr>
<td>st count, r1</td>
<td>st count, r1</td>
</tr>
</tbody>
</table>

Memory

Solution: Atomic Sequence of Instructions

```
T1
begin atomic
ld (count)
add
switch
end atomic

T2
begin atomic
ld (count)
add
st (count+1)
end atomic

count+1
switch
```

- Atomic Sequence
  - Appears to execute to completion without any intervening operations

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 saved on context switch
- Exceptions and Interrupts
  - what drives OS
- Mode bit, Privileged Instructions
  - kernel structure
- Memory Hierarchy
  - MMU, access characteristics of levels
- Concurrency
  - atomic sequences, synchronization