Outline for Today

- **Announcements**
  - **How office hours for UTAs work: announced around Nachos assignments.**
  - **Those who didn't get your photo taken last time, we will do it again *next* time (or you can provide a picture – jpg, 216x216 px)**
  - **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).**
  - **Plug for Soph. and Juniors to do undergrad research experiences - CRA awards**
- **Lecture: Review of computer architecture**

Groups So Far

Jake Palmer
Mike Fliss
Jennifer Bedell
Ryan Kazanciyan

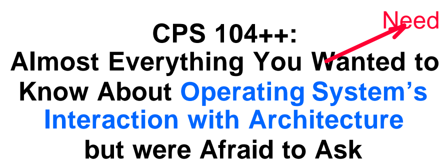
Jeremy Morgan
Blake Byrnes
Dave Frist
Andrew Preslar

Amin Aminfar
Jason Donald
Neal Dongre
Peter Kraft

Wan Chun Chen
Janet Ou
Todd Dolinsky
David Wang

Julie Kempton
Dan Drewnowski
Alex Martinez

Chris Traver
Maureen Hurtgen
Adam Mercer



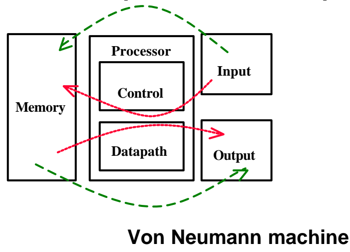
CPS 104++:
Almost Everything You Wanted to Know About Operating System's Interaction with Architecture but were Afraid to Ask

Basic Storyline – Evolution of HW Support

- **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.**

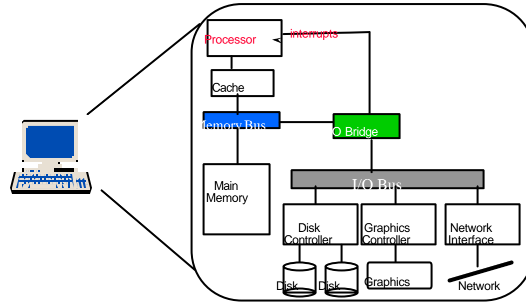
The Big Picture

- **The Five Classic Components of a Computer**

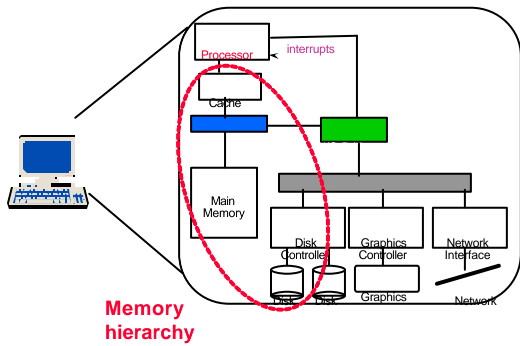


Von Neumann machine

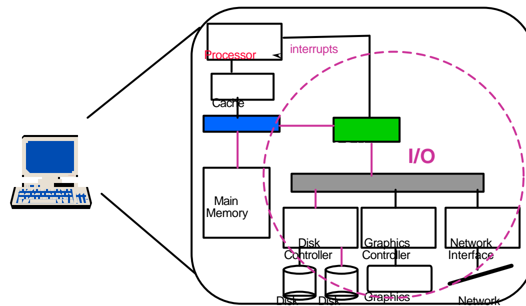
System Organization



System Organization



System Organization



What do we need to know about the Processor?

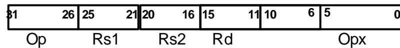
- Size (# bits) of *effective memory addresses* that can be generated by the program and therefore, the amount of memory that can be accessed.
- Information that is crucial *process state* or *execution context* describing the execution of a program (e.g. program counter, stack pointer). This is stuff 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

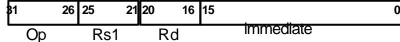
- **32-bit fixed format instruction**
- **32 (32,64)-bit GPR (general purpose registers)**
- **Status registers (condition codes)**
- **Load/Store Architecture**
  - **Only accesses to memory are with `load/store` instructions**
  - **All other operations use registers**
  - **addressing mode: base register + 16-bit offset**
- **Not Intel x86 architecture!**

Example: MIPS

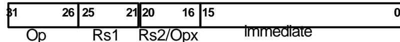
Register-Register



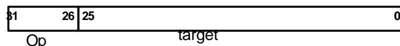
Register-Immediate



Branch, Load, Store



Jump / Call



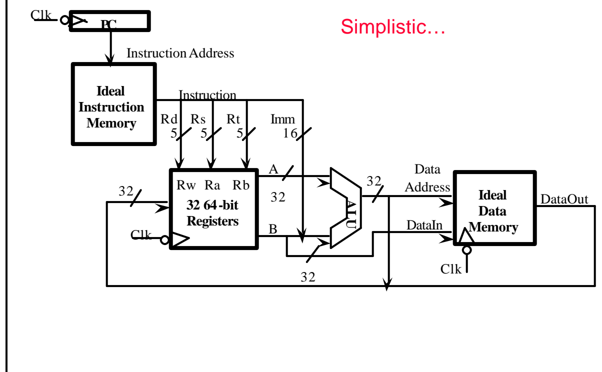
So, how many memory locations can we address?
Can we tell how much memory the machine has?

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

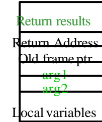
Simplistic…

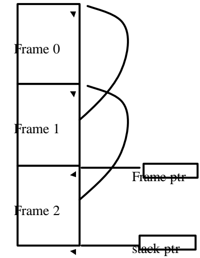


Program Stack

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

A stack frame (Activation Record)





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?

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, 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

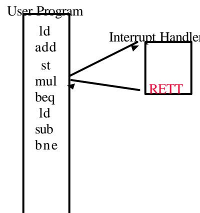
* At what point in the execution cycle does this make sense?

** 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



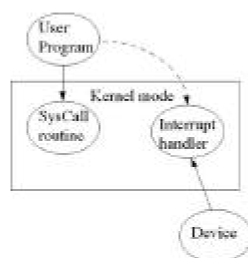
- Invoke specific kernel routine based on type of interrupt
  - interrupt/exception handler
- Must determine what caused interrupt
  - could use software to examine each device
  - PC = interrupt_handler
- Vectored Interrupts
  - PC = interrupt_table[i]
  - kernel initializes table at boot time
- Clear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

Context Switches

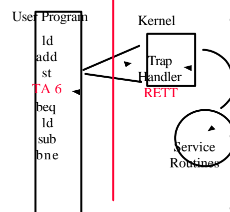
- Save current execution context
  - Save registers and program counter
  - information about the context (e.g., ready, blocked)
- Restore other context
- Need data structures in kernel to support this
  - process control block
- Why do we context switch?
  - Timeslicing: HW clock tick
  - I/O begin 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 v.s. 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 IUnterrupt (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.
- 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-interruptable 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.
- 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 .

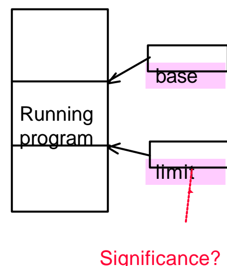
Timer Operation

- 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

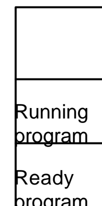
Protection:

- Simplest scheme uses base and limit registers, loaded by OS (privileged operation) before starting program.
- Issuing an address out of range causes an exception.



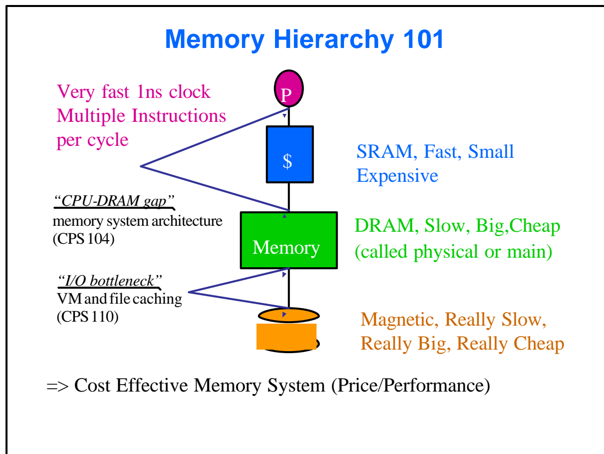
Allocation

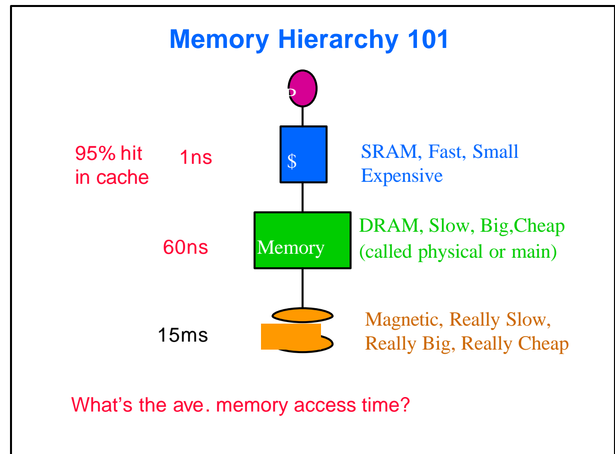
- Disjoint programs have to occupy different cells in memory (or the same cells at different times - swapping*)
- Memory management has to determine where, when, and how** code and data are loaded into memory



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

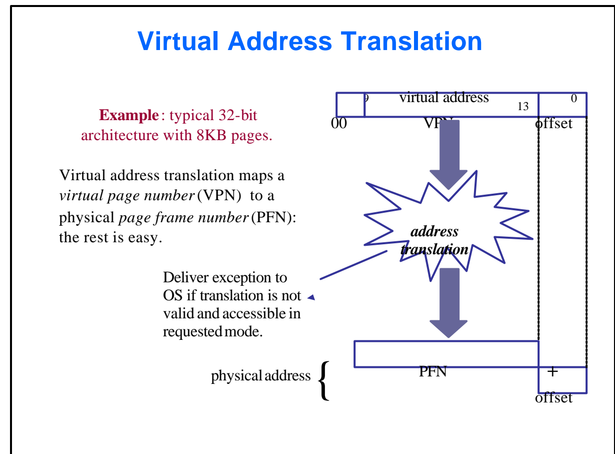
**What HW support is available in architecture? MMU

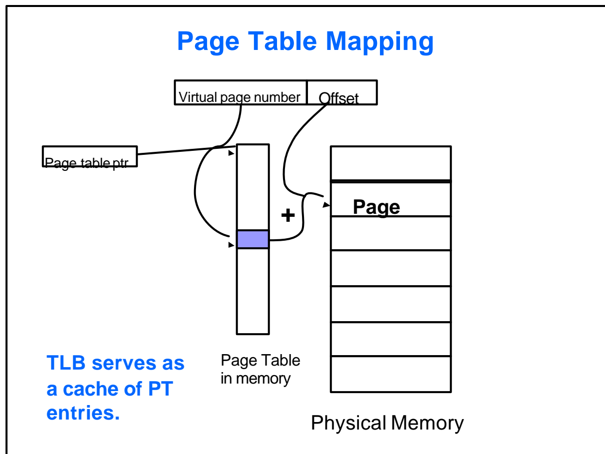


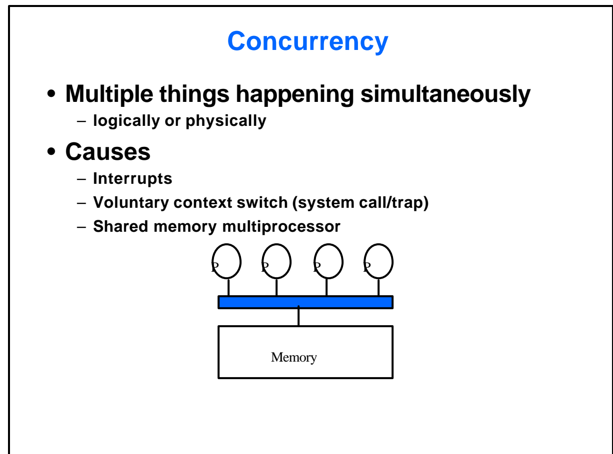


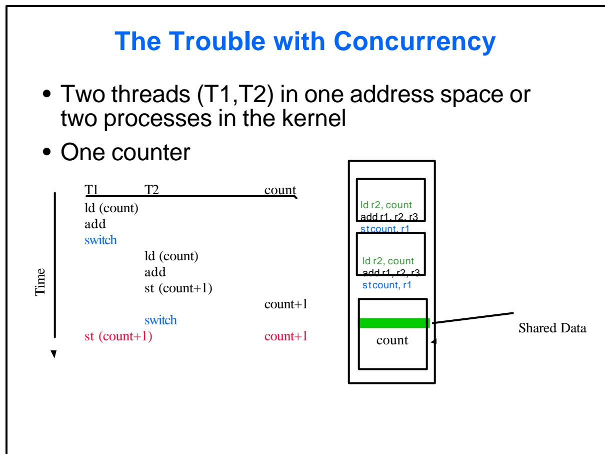
Role of MMU Hardware and OS

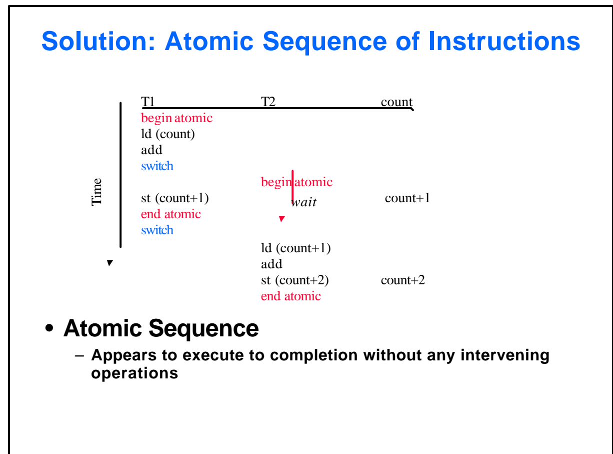
- VM address translation must be very cheap (on average).
  - Every instruction includes one or two memory references.
    - » (including the reference to the instruction itself)
- VM translation is supported in hardware by a *M*emory *M*anagement *U*nit or *MMU*.
  - The addressing model is defined 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.




Page Table Mapping
Virtual page number
Offset
Page table ptr
+
Page
TLB serves as a cache of PT entries.
Page Table in memory
Physical Memory


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


The Trouble with Concurrency
Two threads (T1,T2) in one address space or two processes in the kernel
One counter
T1
T2
count
ld (count)
add
switch
ld (count)
add
st (count+1)
count+1
switch
st (count+1)
count+1
Time
Shared Data
count


Solution: Atomic Sequence of Instructions
T1
T2
count
begin atomic
ld (count)
add
switch
begin atomic
wait
st (count+1)
count+1
end atomic
switch
ld (count+1)
add
st (count+2)
count+2
end atomic
Time
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