

Announcements

- HW-1 due today, 5pm at Gates Hall 310
 - No extensions, no exceptions
 - We will post the solutions tonight
- Quiz 1, Wed 10/22, 6-9pm, Cubberley auditorium
 - Lectures 1-9, closed books
 - Bring calculator + 1 page of notes
 - No lecture on 10/22 (extra office hours instead)
- Review session for quiz 1: Fri 10/17, 11am, Skilling 193
 - Look at the sample quiz 1
 - Prepare for Q&A

Lecture 8:

Architectural Support for Operating Systems & Virtual Machines

Department of Electrical Engineering
Stanford University

<http://eeclass.stanford.edu/ee282>

Architectural Support for System Software

- The role of the operating system
 - Resource management on behalf of programs
 - Program believes it has unlimited access to infinite resources
 - Protection and/or safe sharing
- Operating modes
- Processes and threads
- Mutual exclusion
- Virtual memory
- Direct memory access
- Direct user access (minimize OS involvement)
- Polling and Interrupts

User Processes and the Operating System

- The operating system provides an API to user processes
 - System calls
 - Pipes/Sockets
- This API gives abstracted access to the hardware on the machine
 - CPUs, memory, disks, network
- User processes run in a fundamentally different mode from the OS
 - i.e. a root process can't do as much as the OS
 - The OS has access to ISA features unavailable to the user process
- The system call is the mechanism for moving from user to kernel
 - It's not necessarily that cheap
 - Might require a pipeline flush
 - x86 has 3 ways to do system calls as they've optimized over the years

Processes and Threads

- Thread
 - Context (i.e. register state)
 - Program counter
 - Address space
 - Privilege
- Multi-threaded programming
 - Multiple threads sharing a single address space
 - All memory is shared among threads
 - No protection between threads; can step on each other's toes
 - All threads have the same privilege
- Multi-process programming
 - Multiple threads, each with independent address spaces
 - No memory is shared between processes (unless you ask the OS)
 - Requires extra mechanisms for communications
 - Each process can have a different privilege (e.g. OpenSSH)

Interprocess Communication (the other IPC)

- Sockets (TCP/IP, unix domain, etc.)
- Pipes (e.g. "ps awx | grep root")
- Shared memory
 - Like threads but requires explicit set up
- Files

- These things are important and all have overheads that architects can address.

Mutual Exclusion

- Motivation
 - How to ensure that ≥ 2 concurrent processes cannot simultaneously access the same data or execute same code
 - With chip-multiprocessors they may be truly parallel
 - Necessary for programs that share data and OS services
 - E.g. two editor processes updating the same file
- Can we use regular load/store instructions to do mutual exclusions?

```
L1: load flag;
    if (flag == 0) store flag=1;
    else goto L1;
    Work(); /* need exclusive access */
    store flag=0;
```

 - Does this work correctly on uniprocessors or multi-processors?

HW Support for Mutual Exclusion & Synchronization

- Atomic instructions: many flavors, same goal
 - Atomic exchange
 - Atomically exchange values in register – memory location
 - Atomic test & set instruction
 - Test if value is 0 and set to 1 if test is successful
 - Atomic compare & swap instruction
 - Test if value is 0 and set it to other value if test is successful
 - Atomic fetch and increment
 - Read old value and store +1
 - Load-linked and store-conditional instructions
 - LL: Load & remember old value
 - SC: Store if old value still in memory
- Implementation: need support from CPU, caches, and memory controller
- Can be used to implement higher level synchronization constructs
 - Locks, barriers, semaphores, ... (see CS140 & CS315A)

Our Simple Example Revisited

- New version assuming atomic exchange
 - Initial value of Reg=1 and flag=0

```
L1: atom_exchange Reg, flag;
    If (Reg == 1) goto L1;
    Work(); /* exclusive access */
    Reg = 1;
    store flag = 0;
```

- Does this work correctly on uniprocessors or multi-processors?

Example: Implementation of Spin Locks

- Spin lock: try to find lock variable 0 before proceeding further

With atomic exchange

```
try:    li R2, #1
lockit: lw R3, 0(R1)      #load var
        bnez R3, lockit   #not free=>spin
        exch R2, 0(R1)    #atomic exchange
        bnez R2, try      #already locked?
```

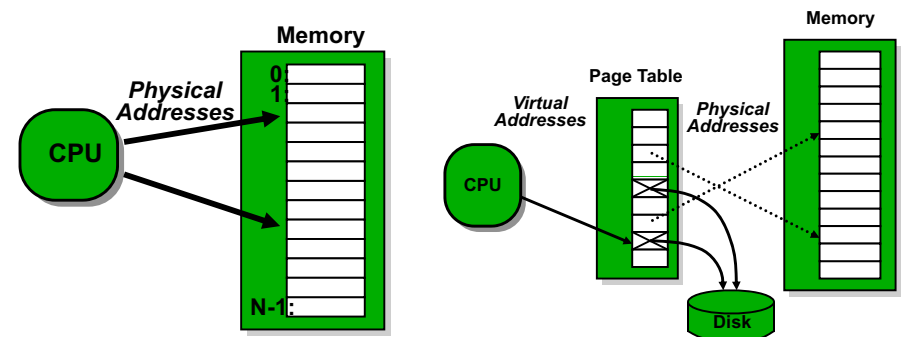
With Load-linked & Store-conditional

```
lockit: ll R2, 0(R1)     #load linked
        bnez R2, lockit   #not free=>spin
        li R2, #1        #locked value
        sc R2, 0(R1)     #store
        beqz R2, lockit  #branch if store fails
```

Scheduling

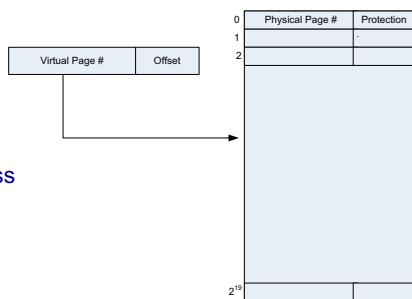
- Generally done completely in the operating system, but...
- Context switches can be improved
 - Don't flush the cache, TLB, etc.
 - Add process ID along with virtual address
 - Provide instructions for quickly saving/restoring registers
- Don't waste power spinning
 - Use Monitor/Mwait
 - Monitor a memory location
 - Mwait → puts core to sleep
 - Other core writes memory location to wake up original core
- Multicore adds a whole new dimension to scheduling!
 - Cache interference
 - Memory bandwidth contention
 - Power issues

Virtual Memory Reminder



Page Tables

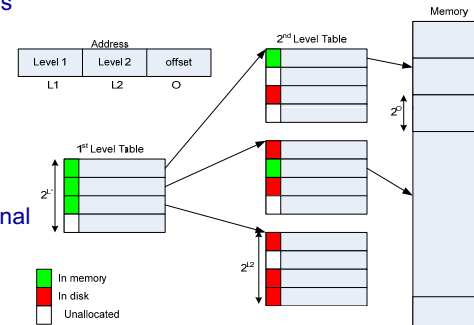
- Simple approach
 - 1 page table per process
 - Flat organization: 1 entry per page



- Problem: can be very large
 - E.g. 32-bit address, 8KB pages
 - $2^{19} * 4$ bytes = 2 MBytes per process
 - Try again with 64-bit addresses ☹
- Most of entries are actually unused!

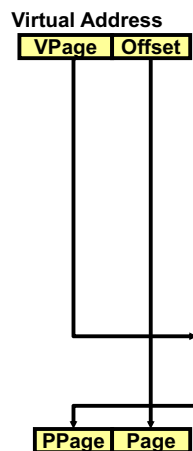
Hierarchical or Multi-level Page Tables

- 1st level page table
 - A page table for page tables
- 2nd level page tables
 - Allocated only if one of its entries corresponds to allocated data



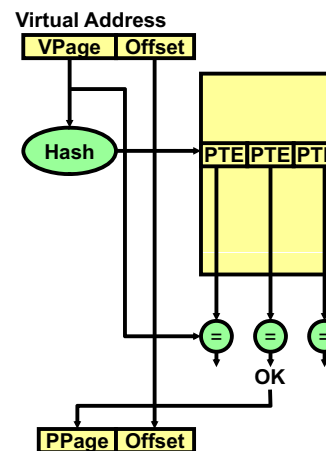
- Advantages
 - Page table space proportional to allocated memory
 - Can page the page tables
- Disadvantage: complexity
 - Especially if TLB miss handled in hardware

Inverted Page Tables



- Store entries only for pages in main memory
 - # entries = # physical pages
 - Much smaller number of entries
- Match virtual page number against every tag in the table
 - Associative read => expensive or slow
 - E.g. 1GB DRAM => 256K entries
- Miss in page table implies page is not in main memory
 - Fetch from disk
 - Need to lookup disk location elsewhere

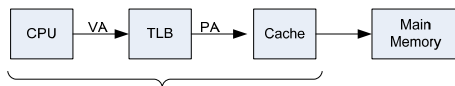
Hashed Inverted Page Tables



- Use hash function to choose a set of M entries
 - Usually $M > 2$
- Need $M * P$ entries for P physical pages to make up for hash losses
 - PowerPC uses $M=8$
- Match VA page against tag in each of M entries
 - Simpler/faster than full associative search

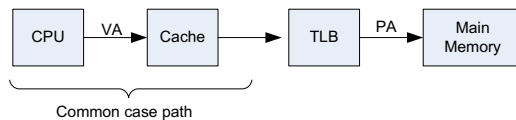
Virtual Memory & Caches

- Physically-addressed L1 caches



- Simple but hit time gets longer

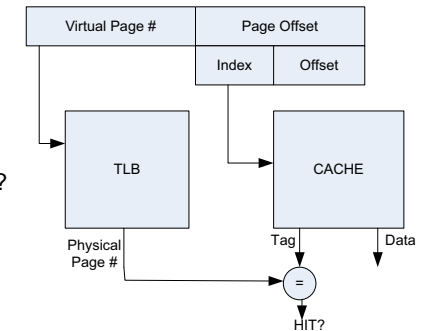
- Virtually-addressed L1 caches



- TLB off the critical path for cache hits
- Potential for cache aliases...

The Compromise: Virtually-indexed, Physically Tagged Caches

- TLB & cache access in parallel
 - Use un-translated portion of address to index cache
 - Retried tag and TLB translation in parallel
 - Use translated address for hit/miss comparison



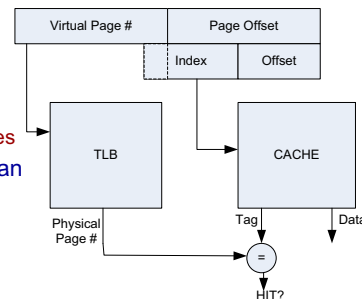
- How big of a cache can I index this way?

Larger Virtually-indexed, Physically Tagged Caches

- Use a few bits of VPN for indexing cache
 - Each bit doubles the size of cache

- Problem: aliases or synonyms

- Imagine two different virtual pages pointing to the same physical page
 - Can be from same or separate processes
- If VPN bits used are different then we can get the same data in the cache twice
- Can end up with read/write inconsistencies...



- How do we handle this?

Solutions for the Aliases Problem

- Stick to the smaller caches
 - Page size times associativity
 - Aliases cannot occur
- Search for aliases in parallel
 - In a 64KB 4-way cache with 8K pages \Rightarrow 2 potential indices for aliases
 - Check in both places on every access (sequentially == slow)
- Track in L2 (assuming inclusion)
 - L2 can detect if same block loaded twice in L1
 - Used in MIPS R10k
- OS solution (coloring)
 - If a physical page is shared, all virtual pages pointing to it must have few LS bits in common
 - Ensures that $\text{index}(VA1) = \text{index}(VA2)$

TLB & Performance

- TLB should have two qualities
 - Fast access (unless we use virtually addressed caches)
 - High coverage
 - Coverage = # translation * size of pages
- Why is coverage a problem?
 - Imagine a 128-entry TLB for 8 KB pages
 - Coverage = 1MB
 - This is <1% of the DRAM available in most computers
 - Can have TLB misses for data available in memory
 - Coverage issue is getting worse
 - Memory capacity is growing by 60% per year...
- We can make the TLB larger, but then it gets slower...

TLB Design Considerations

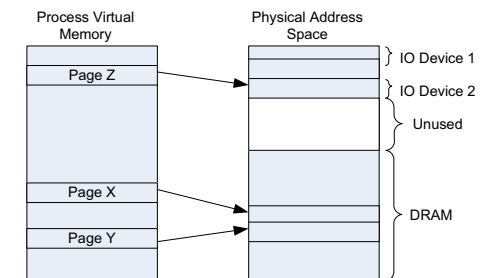
- ISA-level solution 1: combine segmentation with paging
 - Use segmentation to improve coverage with few TLB entries
- ISA-level solution 2: multiple page sizes
 - Super-pages: 16KB, 64KB, 256KB, 1MB, ...
 - OS is more complicated: external fragmentation
 - TLB match logic is also a little more complicated
 - Need guarantee of a single match in some cases...
- Hardware solution 1: multi-level TLBs
 - Small, fully associative first level TLB
 - Large, associative 2nd level TLB
- Hardware solution 2: separate TLBs
 - For instructions and data

TLB Miss/Fault

- What happens when there's a miss in the TLB?
 - Hardware miss handler to walk the page table (e.g. x86)
 - Fast
 - Fixed page table structure
 - Software miss handler (e.g. alpha)
 - Slower
 - Allows for arbitrary page table structures
 - Allows for software optimizations (prefetching?)
- What happens when there's a page fault?
 - Trap to the VM or OS to figure out what to do in software
 - Save current context so a restart can happen
 - Expensive
 - Pipeline flush, cache misses, TLB misses, etc.
 - Allows for paging to disk, across the network, etc.
 - Memory mapped files

Memory-Mapped I/O

- Loads/stores used to read/write memory locations on devices
 - Vs. special I/O bus with special instructions (like x86 in/out)
- I/O registers mapped to physical address space
 - Bridges forward accesses
- OS provides protection
- Uncached
- Slow
 - 3000 cycles to device on PCI bus (p4 3GHz)
- Often used for device configuration



Direct Memory Access (DMA)

- DMA: a separate engine for block transfers between I/O & memory
 - CPU describes the transfer to the DMA engine
 - *From address*: source of data (memory or I/O buffer)
 - *To address*: destination of data (memory or I/O buffer)
 - *Size*: number of bytes to move
 - *Type*: type of transfer (sequential vs. strided special flow control)
 - DMA engine performs memory/disk read/writes
 - CPU can execute other code during the DMA transfer
- DMA engine: a FSM, an adder, and some buffering
 - FSM: repeats read-from/write-to pairs until end of transfer
 - Adder: used to increment to and from addresses
 - Buffer: temporarily buffers source data
 - Allows timing decoupling between the two devices
 - Simplifies transfers between devices with different bursts

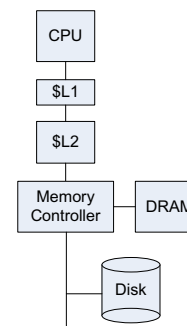
Direct Memory Access (DMA)

- DMA engines typically integrated with memory and I/O controllers
- Multi-channel DMA engines are also common
 - Allow multiple overlapping DMA transfers
 - Disk to/from memory, network to/from memory
 - Lower latency, better utilization of memory bandwidth
- Multi-channel DMA engine hardware
 - A set of registers per channel
 - Separate buffers or a shared pool of buffers
 - Address adders can be shared
 - An arbitration mechanism
 - Priority based or round-robin between non-blocked

DMA & Virtual Memory

- Do we use virtual or physical memory addresses to set up DMA?
- Option 1: Physical address DMA
 - Multi-page transfers only if sequential in physical space
 - Memory page(s) must be pinned in memory during DMA
 - No page fault during the DMA accesses
 - This can waste a significant amount of memory
 - Simple yet restrictive
 - OS access only
- Option 2: Virtual address DMA
 - Need a TLB in the DMA (shared between channels)
 - Generally requires OS hacks to keep this managed
 - Can handle multi-page transfers is sequential in virtual space
 - Must be able to handle DMA TLB miss
 - Can handle page-fault
 - Flexible yet complicated

DMA & Cache Hierarchies

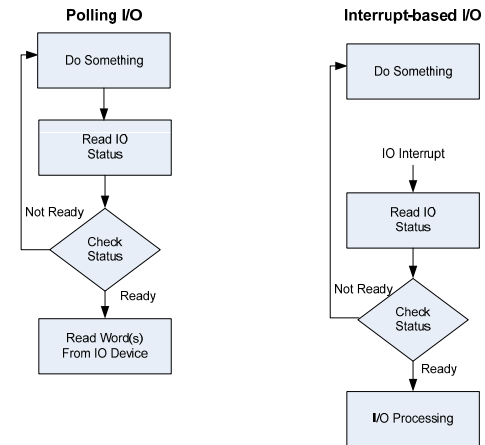


- What if caches hold dirty data involved in DMA transfers?
 - Stale data, can end up with program error
- Software solution: DMA from uncached addresses
 - Flush data from cache before DMA starts
 - Selectively or all contents of cache
 - Mark pages involved as uncached during DMA
 - Use virtual memory system
- Hardware solution: cache coherence
 - Memory addresses are snooped by the L2 as well
 - If L2 hit & line dirty, cache provides the data
 - If L2 miss or line clean, memory provides the data
- Where does data go when a device writes memory?
 - Generally memory
 - What about placing in the cache?
 - Can lead to significant performance improvement if it's used
 - Can lead to cache pollution and significant performance degradation

Virtual Memory & IO Devices: Direct User Access

- Goal: avoid switching to OS to communicate with IO devices
 - E.g. to send/receive network messages
- Protection & sharing by virtualizing the IO resources
 - Divide buffer space in network card to isolated sub-buffers
 - Map a sub-buffer to the address space of the user program
 - Using virtual memory mechanisms
 - Use regular load/stores to the sub-buffer to send/receive packets
 - Multiple processes may be doing this in parallel...
- What if a user program exceeds allocated buffer space or attempts to run without allocated buffer space?
 - Virtual memory exception
 - OS kicks in and does resource allocation and buffering as it does for other resources

I/O Control Models



Polling vs. Interrupts

- Polling
 - Low latency if polling frequently
 - High overhead if device is not ready
 - Must poll often enough
 - Getting stuck in a loop means you never poll
- Interrupts
 - Potentially long latency between interrupt and processing
 - High overhead if device interrupts frequently
 - Interrupt coalescing can be used to reduce frequency
 - At the cost of greater latency
- It's possible to switch between the two, but knowing which to use is a challenge.

Interrupts

- Where does the interrupt go in a multicore?
 - Typically round robin assignment device interrupts to cores
 - E.g. NIC0 → Core0, NIC1 → Core1, etc.
 - What about 10GigE NICs? Need more than one core to do processing.
 - Create virtual NICs and assign different cores to different virtual NICs
 - Implement an algorithm that assigns packets to cores based on address
- How exactly is the processor notified of an interrupt?
 - Dedicated interrupt pins
 - Message signaled interrupts (MSI)
- How are interrupts processed?
 - Go to a main interrupt handler and probe everything (very slow)
 - Vector to a proper handler based on the interrupt

Summary

- OS: a very important application for architects
- Architectural support for OS
 - Operating modes
 - Processes and threads
 - Support for fast context switching
 - Mutual exclusion
 - Virtual memory
 - Watch out for interactions with caching
 - Watch out for TLB performance & coverage
 - DMA
 - Interrupts/Polling
- Everything said today applies to virtual machines
 - Lots of architectural work going on to help out VMs