EE108B
Lecture 14
Virtual Memory

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

- Reminders
  - Lab3 and PA2 (part a) are due on today

Review: Hardware Support for Operating System

- Operating system
  - Manages hardware resources (CPU, memory, I/O devices)
    - On behalf of user applications
  - Provides protection and isolation
  - Virtualization
    - Processes and virtual memory
- What hardware support is required for the OS?
  - Kernel and user mode
    - Kernel mode: access to all state including privileged state
    - User mode: access to user state ONLY
  - Exceptions/interrupts
  - Virtual memory support

MIPS Interrupts

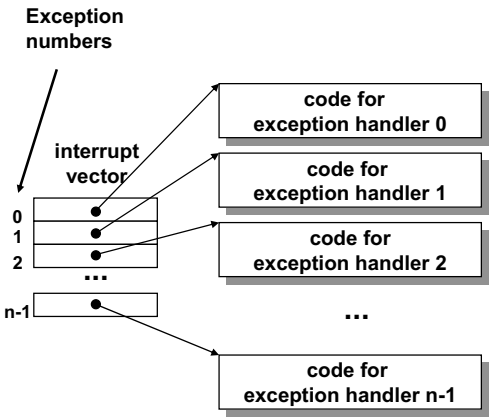
- What does the CPU do on an exception?
  - Set `EPC` register to point to the restart location
  - Change CPU mode to kernel and disable interrupts
  - Set `Cause` register to reason for exception also set `BadVaddr` register if address exception
  - Jump to interrupt vector
- Interrupt Vectors
  - `0x8000 0000` : TLB refill
  - `0x8000 0180` : All other exceptions
- Privileged state
  - `EPC`
  - `Cause`
  - `BadVaddr`

A Really Simple Exception Handler

```
xcpt_cnt:
    la      $k0, xcptcount      # get address of counter
    lw      $k1, 0($k0)         # load counter
    addu    $k1, 1              # increment counter
    sw      $k1, 0($k0)         # store counter
    eret                        # restore status reg: enable
                                # interrupts and user mode
                                # return to program (EPC)
```

- Can't survive nested exceptions
  - OK since does not re-enable interrupts
- Does not use any user registers
  - No need to save them

Interrupt vectors
Accelerating handler dispatch



- 1. Each type of interrupt has a unique exception number k
- 2. Jump table (interrupt vector) entry k points to a function (exception handler).
- 3. Handler k is called each time exception k occurs

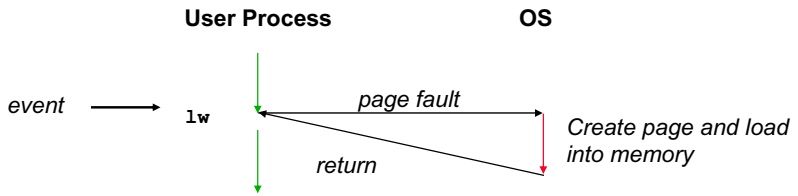
But the CPU is much faster than I/O devices and most OSs use a common handler anyway

Make the common case fast!

Exception Example #1

- Memory Reference
  - User writes to memory location
  - That portion (page) of user's memory is currently not in memory
  - Page handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try

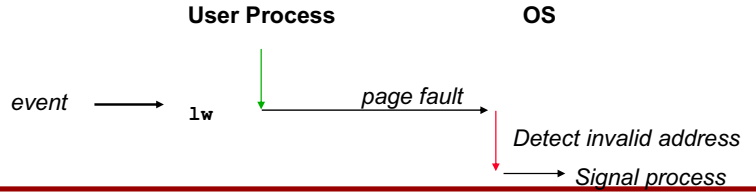
```
int a[1000];
main ()
{
    a[500] = 13;
}
```



Exception Example #2

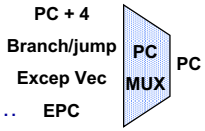
- Memory Reference
  - User writes to memory location
  - Address is not valid
  - Page handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with "segmentation fault"

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```



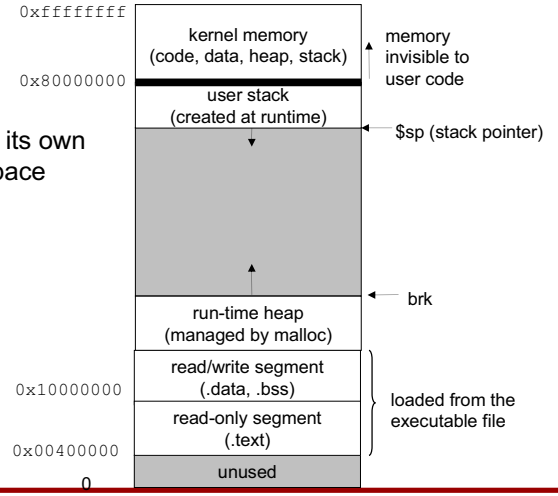
Review: OS ↔ User Code Switches

- System call instructions (syscall)
  - Voluntary transfer to OS code to open a file, write data, change working directory, …
  - Jumps to predefined location(s) in the operating system
  - Automatically switches from user mode to kernel mode
- Exceptions & interrupts
  - Illegal opcode, device by zero, …, timer expiration, I/O, …
  - Exceptional instruction becomes a jump to OS code
    - Precise exceptions
  - External interrupts attached to some instruction in the pipeline
  - New state: EPC, Cause, BadVaddr registers



Process Address Space

- Each process has its own private address space

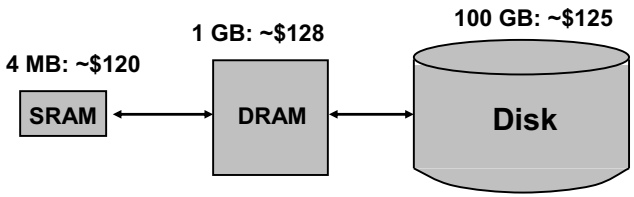


Motivations for Virtual Memory

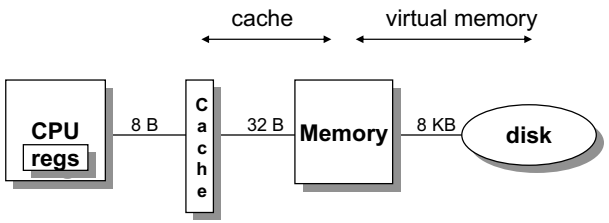
- #1 Use Physical DRAM as a Cache for the Disk
  - Address space of a process can exceed physical memory size
  - Sum of address spaces of multiple processes can exceed physical memory
- #2 Simplify Memory Management
  - Multiple processes resident in main memory.
    - Each process with its own address space
  - Only "active" code and data is actually in memory
    - Allocate more memory to process as needed.
- #3 Provide Protection
  - One process can't interfere with another.
    - because they operate in different address spaces.
  - User process cannot access privileged information
    - different sections of address spaces have different permissions.

Motivation #1: DRAM a "Cache" for Disk

- Full address space is quite large:
  - 32-bit addresses: ~4,000,000,000 (4 billion) bytes
  - 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes
- Disk storage is ~100X cheaper than DRAM storage
- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Levels in Memory Hierarchy

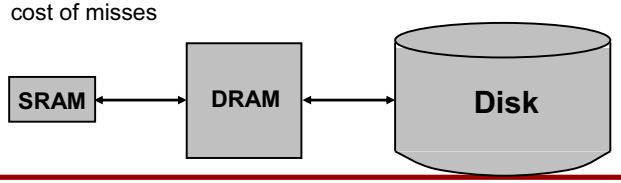


	Register	Cache	Memory	Disk Memory
size:	128 B	<4MB	< 16 GB	> 100 GB
Speed(cycles):	0.5-1	1-20	80-100	5-10 M
$/Mbyte:		$30/MB	$0.128/MB	$0.001/MB
line size:	8 B	32 B	8 KB	

larger, slower, cheaper

DRAM vs. SRAM as a "Cache"

- DRAM vs. disk is more extreme than SRAM vs. DRAM
  - Access latencies:
    - DRAM ~10X slower than SRAM
    - Disk ~**100,000X** slower than DRAM
  - Importance of exploiting spatial locality:
    - First byte is ~**100,000X** slower than successive bytes on disk
      - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
  - Bottom line:
    - Design decisions made for DRAM caches driven by enormous cost of misses

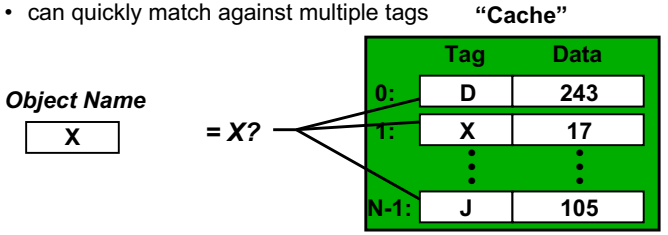


Impact of These Properties on Design

- If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?
  - Line size?
    - Large, since disk better at transferring large blocks and minimizes miss rate
  - Associativity?
    - High, to mimimize miss rate
  - Write through or write back?
    - Write back, since can't afford to perform small writes to disk

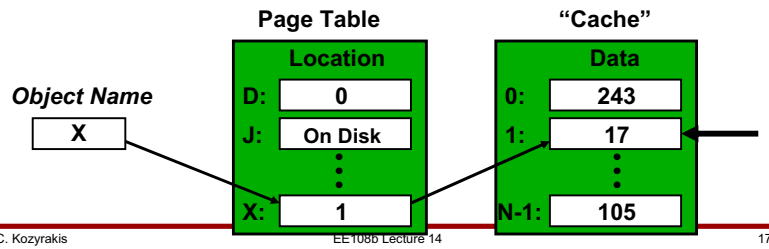
Locating an Object in a "Cache"

- SRAM Cache
  - Tag stored with cache line
  - Maps from cache block to memory blocks
    - From cached to uncached form
  - No tag for block not in cache
  - Hardware retrieves information
    - can quickly match against multiple tags



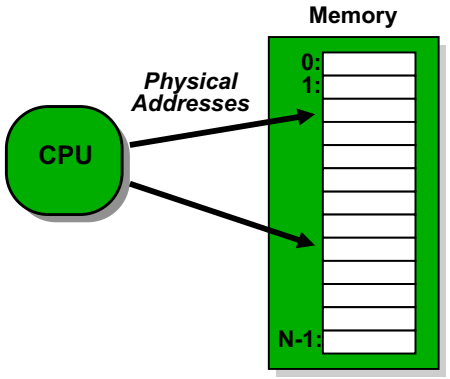
Locating an Object in a "Cache" (cont.)

- DRAM Cache
  - Each allocate page of virtual memory has entry in page table
  - Mapping from virtual pages to physical pages
    - From uncached form to cached form
  - Page table entry even if page not in memory
    - Specifies disk address
  - OS retrieves information



A System with Physical Memory Only

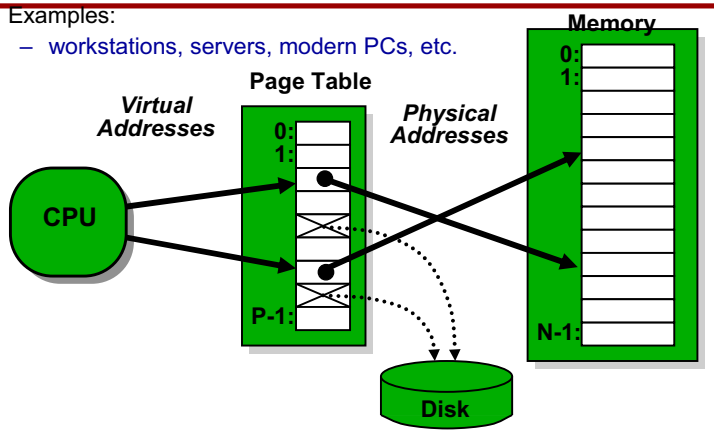
- Examples:
  - most Cray machines, early PCs, nearly all embedded systems, etc.



Addresses generated by the CPU point directly to bytes in physical memory

A System with Virtual Memory

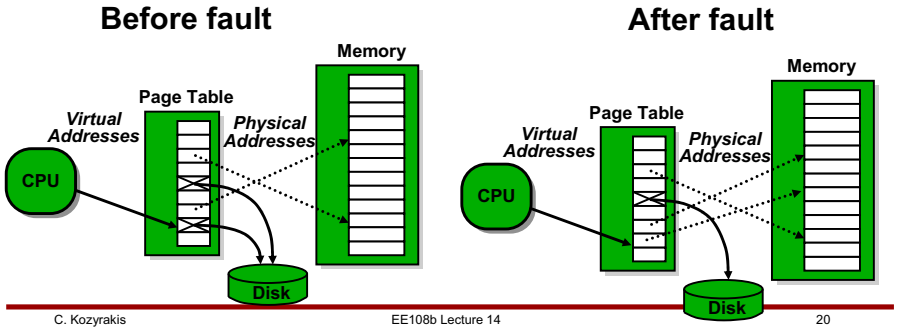
- Examples:
  - workstations, servers, modern PCs, etc.



Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)

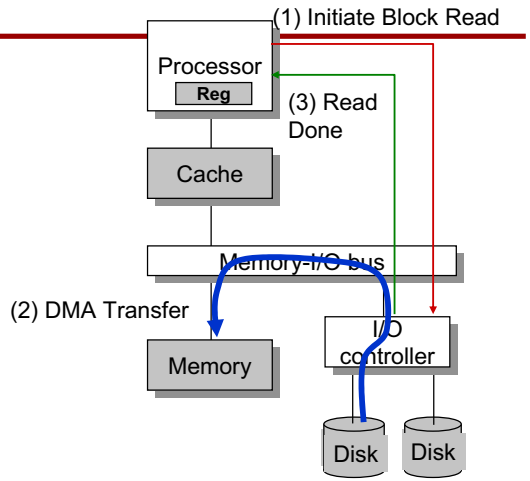
Page Faults (Similar to "Cache Misses")

- What if an object is on disk rather than in memory?
  - Page table entry indicates virtual address not in memory
  - OS exception handler invoked to move data from disk into memory
    - current process suspends, others can resume
    - OS has full control over placement, etc.



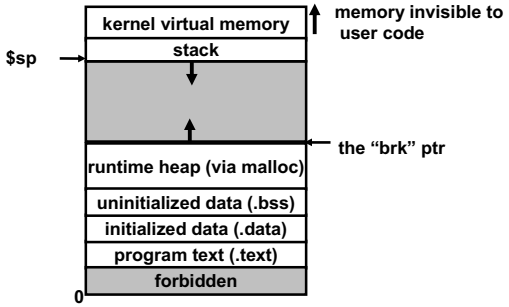
Servicing a Page Fault

- Processor Signals Controller
  - Read block of length P starting at disk address X and store starting at memory address Y
- Read Occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- I / O Controller Signals Completion
  - Interrupt processor
  - OS resumes suspended process



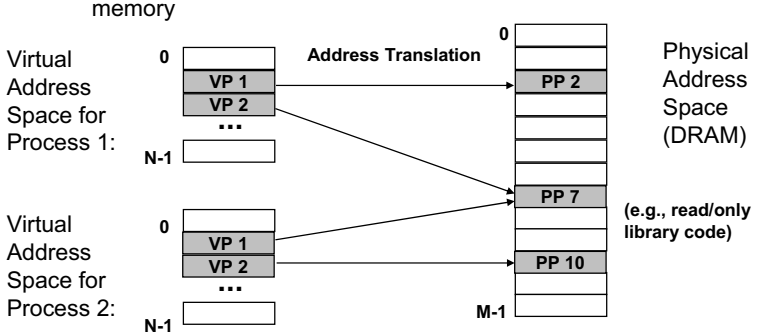
Motivation #2: Memory Management

- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
  - what if two processes access something at the same address?



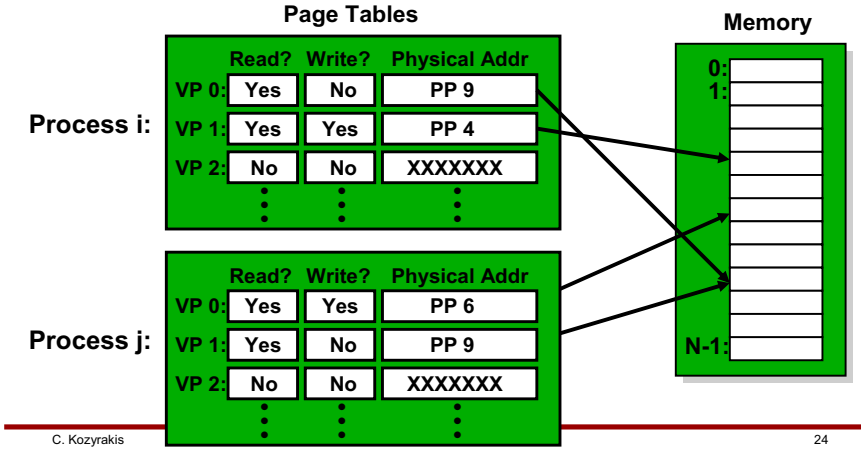
Solution: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
  - blocks are called "pages" (both virtual and physical)
- Each process has its own virtual address space
  - operating system controls how virtual pages as assigned to physical memory

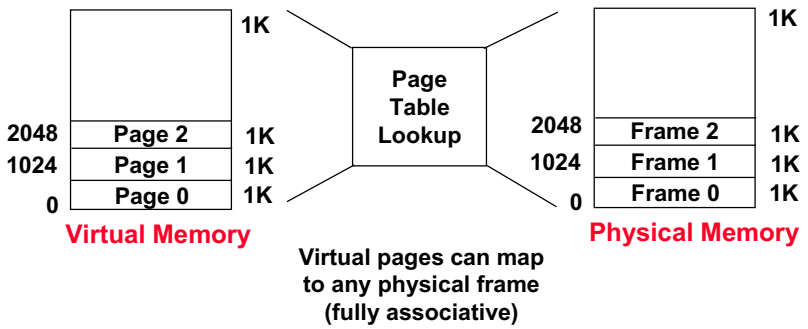


Motivation #3: Protection

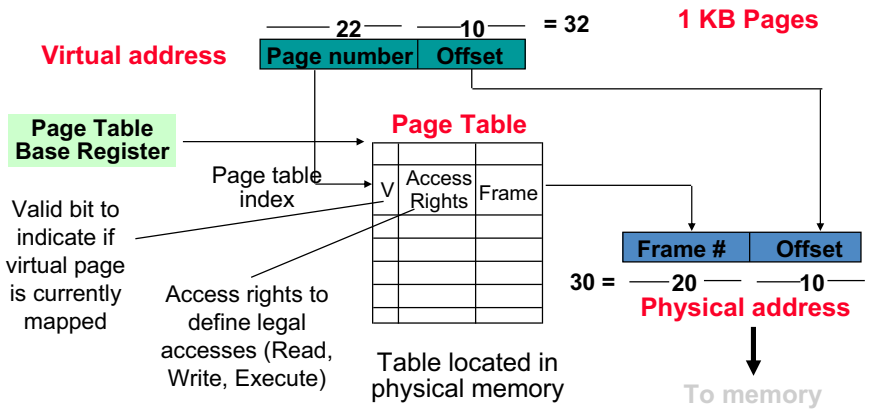
- Page table entry contains access rights information
  - hardware enforces this protection (trap into OS if violation occurs)



Paging Address Translation



Page Table/Translation



Process Address Space

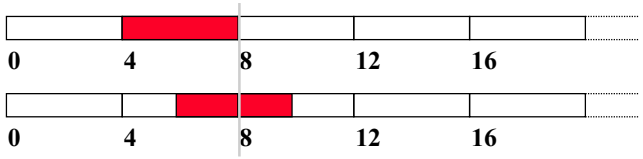
- Each process has its own address space
  - Each process must therefore have its own translation table
  - Translation tables are changed only by the OS
  - A process cannot gain access to another process's address space
- Context switches
  - Steps
    1. Save previous process' state (registers, PC) in the Process Control Block (PCB)
    2. Change page table pointer to new process's table
    3. Flush the TLB (discussed later)
    4. Initialize PC and registers from new process' PCB
  - Context switches occur on a *timeslice* (when the *scheduler* determines the next process should get the CPU) or when blocking for I/O

Translation Process

- Valid page
  - Check access rights (R, W, X) against access type
    - Generate physical address if allowed
    - Generate a protection fault if illegal access
- Invalid page
  - Page is not currently mapped and a *page fault* is generated
- Faults are handled by the operating system
  - Protection fault is often a program error and the program should be terminated
  - Page fault requires that a new frame be allocated, the page entry is marked valid, and the instruction restarted
- *Frame table* has mapping from physical address to virtual address and tracks used frames

Unaligned Memory Access

- Memory access might be aligned or unaligned



- What happens if unaligned address access straddles a page boundary?
  - What if one page is present and the other is not?
  - Or, what if neither is present?
- MIPS architecture disallows unaligned memory access
- Interesting legacy problem on 80x86 which does support unaligned access

Page Table Problems

- "Internal" Fragmentation resulting from fixed size pages since not all of page will be filled with data
  - Problem gets worse as page size is increased
- Each data access now takes two memory accesses
  - First memory reference uses the page table base register to lookup the frame
  - Second memory access to actually fetch the value
- Page tables are large
  - Consider the case where the machine offers a 32 bit address space and uses 4 KB pages
    - Page table contains $2^{32} / 2^{10} = 2^{22}$ entries
    - Assume each entry contains ~32 bits
    - Page table requires 4 MB of RAM per process!

Improving Access Time

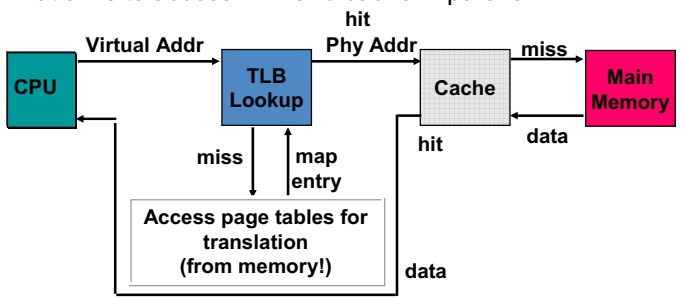
- But we know how to fix this problem, let's cache the page table and leverage the principles of locality
- Rather than use the normal caches, we will use a special cache known as the Translation Lookaside Buffer (TLB)
  - All accesses are looked up in the TLB
    - A hit provides the physical frame number
    - On a miss, look in the page table for the frame
  - TLBs are usually smaller than the cache
    - Higher associativity is common
    - Similar speed to a cache access
    - Treat like a normal cache and use write back vs. write through

TLB Entries

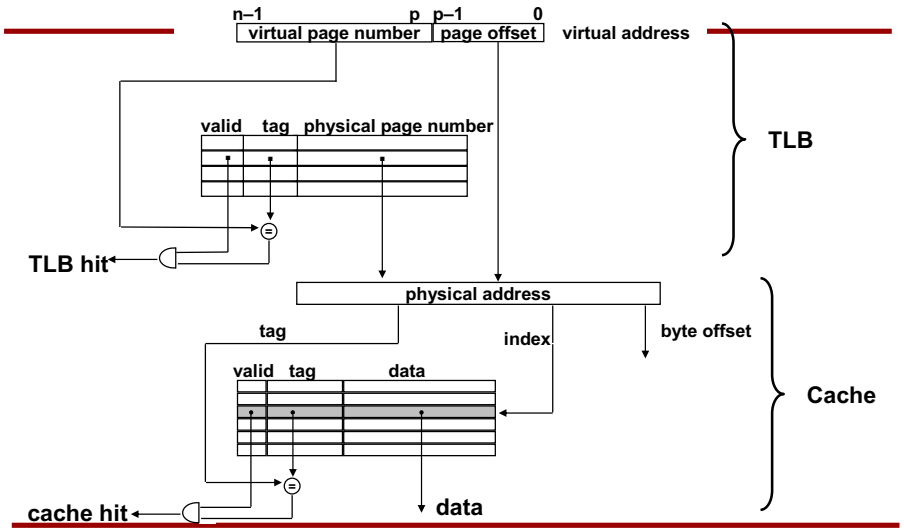
- Just holds a cached Page Table Entry (PTE)
- Additional bits for LRU, etc. may be added

Virtual Page	Physical Frame	Dirty	LRU	Valid	Access Rights

- Optimization is to access TLB and cache in parallel



Address Translation with a TLB



TLB Miss Handler

- A TLB miss can be handled by software or hardware
  - Miss rate : 0.01% – 1%
- In software, a special OS handler looks up the value in the page table
  - Ten instructions on R3000/R4000
- In hardware, microcode or a dedicated finite state machine (FSM) looks up the value
  - Page tables are stored in regular physical memory
  - It is therefore likely that the data is in the L2 cache
  - Advantages
    - No need to take an exception
    - Better performance but may be dominated by cache miss

TLB Caveats

- What happens on a context switch?
  - If the TLB entries have a Process ID (PID) associated with them, then nothing needs to be done
  - Otherwise, the OS must flush the entries in the TLB
- Limited Reach
  - 64 entry TLB with 8KB pages maps 0.5 MB
  - Smaller than many L2 caches
  - TLB miss rate > L2 miss rate!
  - Motivates larger page size

TLB Case Study: MIPS R2000/R3000

- Consider the MIPS R2000/R3000 processors
  - Addresses are 32 bits with 4 KB pages (12 bit offset)
  - TLB has 64 entries, fully associative
  - Each entry is 64 bits wide:

20	6	6	20	1	1	1	1	8
Virtual Page	PID	0	Physical Page	N	D	V	G	0

PID	Process ID
N	Do not cache memory address
D	Dirty bit
V	Valid bit
G	Global (valid regardless of PID)

Page Size

- Larger Pages
  - Advantages
    - Smaller page tables
    - Fewer page faults and more efficient transfer with larger applications
    - Improved TLB coverage
  - Disadvantages
    - Higher internal fragmentation
- Smaller Pages
  - Advantages
    - Improved time to start up small processes with fewer pages
    - Internal fragmentation is low (important for small programs)
  - Disadvantages
    - High overhead in large page tables
- General trend toward larger pages
  - 1978: 512 B
  - 1984: 4 KB
  - 1990: 16 KB
  - 2000: 64 KB

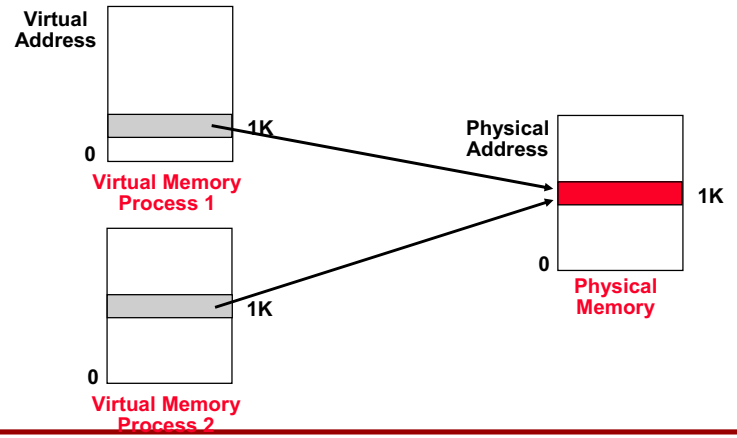
Multiple Page Sizes

- Some machines support multiple page sizes to balance competing goals
  - Page size dependent upon application
    - OS kernel uses large pages
    - User applications use smaller pages
    - SPARC: 8KB, 64KB, 1 MB, 4MB
    - MIPS R4000: 4KB – 16 MB
  - Results
    - Complicates TLB design since must account for page size

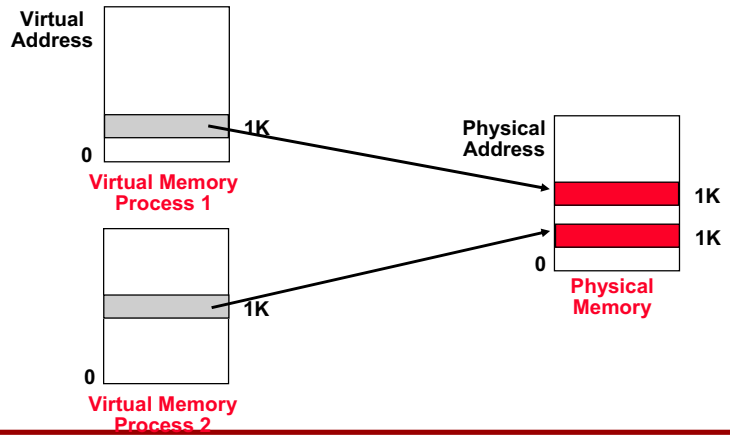
Page Sharing

- Another benefit of paging is that we can easily share pages by mapping multiple pages to the same physical frame
- Useful in many different circumstances
  - Useful for applications that need to share data
  - Read only data for applications, OS, etc.
  - Example: Unix `fork()` system call creates second process with a "copy" of the current address space
- Often implemented using *copy-on-write*
  - Second process has read-only access to the page
  - If second process wants to write, then a page fault occurs and the page is copied

Page Sharing (cont)



Copy-on-Write



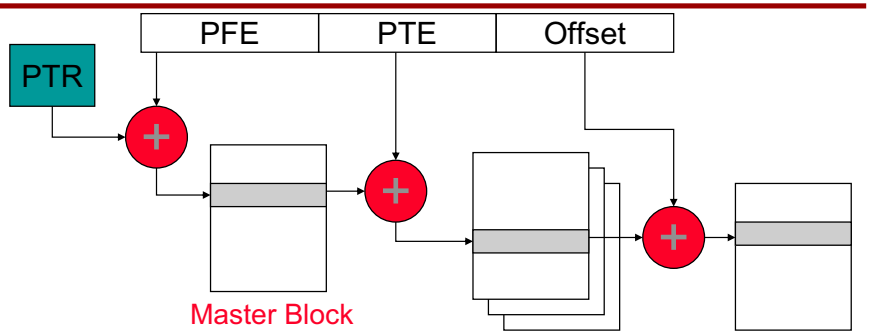
Page Table Size

- Page table size is proportional to size of address space
  - Even when applications use few pages, page table is still very large
- Example: Intel 80*x*86 Page Tables
  - Virtual addresses are 32 bits
  - Page size is 4 KB
  - Total number of pages: $2^{32} / 2^{12} = 1$ Million
  - Page Table Entry (PTE) are 32 bits wide
    - 20 bit Frame address
    - Dirty bit, accessed bit, valid (present) bit
    - 9 flag and unused bits
  - Total page table size is therefore $2^{20} \times 4$ bytes = 4 MB
  - But, only a small fraction of those pages are actually used!
- The large page table size is made even worse by the fact that the page table must be in memory
  - Otherwise, what would happen if the page table were suddenly swapped out?

Paging the Page Tables

- One solution is to use multi-level page tables
  - If each PTE is 4 bytes, then each 4 KB frame can hold 1024 PTEs
  - Use an additional "master block" 4 KB frame to index frames containing PTEs
    - Master frame has 1024 PFEs (Page Frame Entries)
    - Each PFE points to a frame containing 1 KB PTEs
    - Each PTE points to a page
  - Now, only the "master block" must be fixed in memory

Two-level Paging



- Problems
  - Multiple page faults
    - Accessing a PTE page table can cause a page fault
    - Accessing the actual page can cause a second page fault
  - TLB plays an even more important role

Virtual Memory Overview

- We can extend our caching idea one step further and use secondary storage as one of the elements in our memory hierarchy
- *Virtual memory* is the process of using secondary storage to to create an illusion of an even larger memory space without incurring the cost
- Removes programming burdens of managing limited physical memory

Virtual Memory Approach

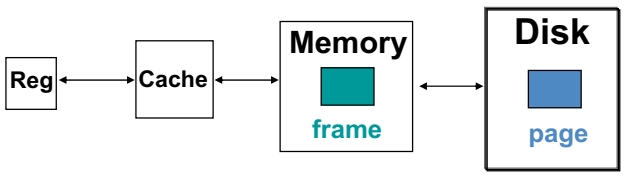
- Presents the illusion of a large address space to every application
  - Each process or program believes it has its own complete copy of physical memory (its own copy of the address space)
- Strategy
  - Pages not currently in use can be written back to secondary storage
  - When page ownership changes, save the current contents to disk so they can be restored later
- Costs
  - Address translation is still required to convert virtual addresses into physical addresses
  - Writing to the disk is a slow operation
  - TLB misses can limit performance

Virtual Addresses

- Each virtual address can be mapped into one of three categories
  - A physical memory address (by way of TLB or PT)
  - A location on disk (we suffer a page miss)
  - Nothing – corresponds to an invalid memory address

Page Misses

- If a page miss corresponds to a valid address that is no longer available, then it must reloaded
  - Locate the contents on disk that were swapped out
  - Find an available physical frame (or evict another frame)
  - Write the contents into the frame
  - Update the page table mappings (for both this process and the one that was evicted if necessary)
- Pages are fetched only on *demand*, when accessed



Page Fault

- Programs use virtual address which may include pages that currently reside in secondary storage
- If an entry is not found then the address is not currently located in physical memory, and a *page fault* is said to have occurred
  - A page fault is handled by the operating system
  - The item must be first fetched from disk storage and loaded into a physical frame
  - A page table entry must be created
  - The access is repeated