


# CSC 369

Week 7: Virtual Memory Mechanisms  
Reading: Text, Ch. 4.3, 4.7




CSC69H15 1 3/6/2007

## Lecture Overview

This week we'll cover more paging mechanisms:


- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)
  - Demand paged virtual memory (space)
- Recap address translation
- Advanced Functionality
  - Sharing memory
  - Copy on Write
  - Mapped files



CSC69H15 3/6/2007

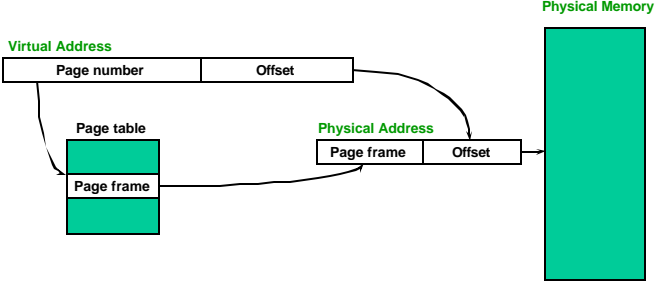
## Last Time

- We talked about memory management with the restriction that processes had to be entirely in or out of physical memory
  - Partitioning (fixed or dynamic)
  - Paging
  - Segmentation
- We discussed translation between **logical** addresses used by program and **physical** addresses used by machine
  - With virtual memory schemes, logical addresses are called **virtual addresses**




CSC69H15 3/6/2007

## Recall Paged Address Translation



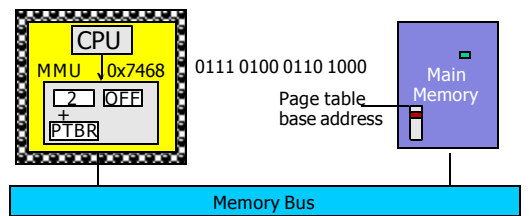
The diagram illustrates the process of paged address translation. It starts with a **Virtual Address** consisting of a **Page number** and an **Offset**. The **Page number** is used to look up the **Page frame** in the **Page table**. The **Page frame** and the **Offset** are then combined to form the **Physical Address**, which is used to access the **Physical Memory**.



CSC69H15 3/6/2007

### Recall paged address translation

- 32-bit virtual address, 4K (4096 bytes) pages
  - Page size, virtual address size set by MMU hardware
  - Offset must be 12 bits ( $2^{12} = 4096$ )
  - Leaves 20 bits for virtual page number (VPN)



### Details of calculation

- Program generates virtual address 0x7468
  - CPU and MMU see binary 0111 0100 0110 1000
  - Virtual page is 0x7, offset is 0x468
- Page table entry 0x7 contains 0x2
  - Page frame number is 0x2
  - Seventh virtual page is stored in second physical frame
- Physical address =  $0x2 \ll 12 + 0x468 = 0x2468$
- MMU hardware generates address of page table entry, does lookup without OS
- OS has to load PTBR for new process on context switch

### The Page Table

- Simplest version
  - a linear array of page table entries, 1 entry per page
  - Stored in memory, attached to process structure
  - Virtual page number (VPN) is array index

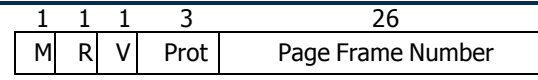
```

struct addrspace {
  paddr_t pgtbl;
  ...
}

struct addrspace *
as_create(void) {
  struct addrspace *as =
    kmalloc(sizeof(struct addrspace));
  int nentries = (unsigned)(-1) >> 13;
  int npages = DIVROUNDUP(nentries*
    sizeof(pte_t), PAGE_SIZE);
  as->pgtbl = getppages(npages);
  ...
}

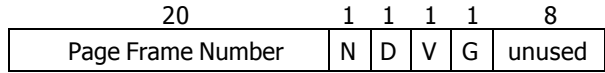
```

### Page Table Entries



- Page table entries (PTEs) control mapping
  - Modify bit (M) says whether or not page has been written
    - Set when a write to a page occurs
  - Reference bit (R) says whether page has been accessed
    - Set when a read or write to the page occurs
  - Valid bit (V) says whether PTE can be used
    - Checked on each use of virtual address
  - Protection bits specify what operations are allowed on page
    - Read/write/execute
  - Page frame number (PFN) determines physical page
  - Not all bits are provided by all architectures

### MIPS R2000 Page Table Entry



- N == not cached
- D == dirty (meaning "writable", not set by hw)
- V == valid
- G == global (can be used by all processes)
- Maximum  $2^{20}$  physical pages, each 4 kB → maximum 4GB of physical RAM

### Paging Limitations- Time

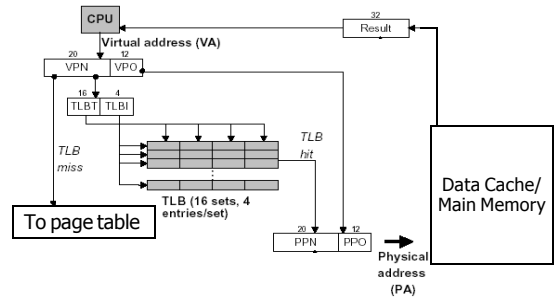
- Memory reference overhead (time)
  - 2 references per address lookup (first page table, then actual memory)
  - Solution: use a hardware cache of lookups
- Translation Lookaside Buffer (TLB)
  - Small, fully-associative hardware cache of recently used translations
  - Part of the MMU

### TLBs

Translate **virtual page #s** into PTEs (not physical addr)

- Can be done in a single machine cycle
- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
- TLBs exploit locality
  - Processes only use a handful of pages at a time
    - 16-48 entries/pages (64-192K)
    - Only need those pages to be "mapped"
  - Hit rates are therefore very important

### Pentium Address Translation



## Managing TLBs

- Address translations for most instructions are handled using the TLB
  - >99% of translations, but there are misses (TLB miss)...
- Who places translations into the TLB (loads the TLB)?
  - Hardware (Memory Management Unit)
    - Knows where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)
  - Software loaded TLB (OS)
    - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
    - Must be fast (but still 20-200 cycles)
    - CPU ISA has instructions for manipulating TLB
    - Tables can be in any format convenient for OS (flexible)

CSC69115



3/6/2007

## Software-Managed TLB

- Can define partial page table for parts of virtual address space that are used

```

struct region {
    vaddr_t vbase;
    int len;
    pte_t *table;
}

struct addrspace {
    array of region
}

int
as_define_region(vaddr_t vbase, int len,...)
{
    struct region *r = (struct region *)
        kmalloc(sizeof(struct region));
    r->vbase = vbase;
    r->len = len;
    int nentries = DIVROUNDUP(len, PAGE_SIZE);
    r->table = (pte_t *)kmalloc(nentries *
        sizeof(pte_t));
    add r to array of regions in addrspace
}
  
```

CSC69115



3/6/2007

## Managing TLBs (2)

- OS ensures that TLB and page tables are consistent
  - When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why? What is one way to fix it?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple

CSC69115



3/6/2007

## Paging Limitations- Space

- Memory required for page table can be large (space overhead)
  - Need one PTE per page
  - 32 bit virtual address space w/ 4K pages =  $2^{20}$  PTEs
  - 4 bytes/PTE = 4MB/page table
    - Half that (2MB) if only user part needs to be mapped
  - 25 processes = 100MB just for page tables!
    - And modern processors have 64-bit address spaces  $\rightarrow$  16 petabytes for page table!
  - Solution 1: Hierarchical page tables
  - Solution 2: Hashed page tables
  - Solution 3: Inverted page tables

CSC69115



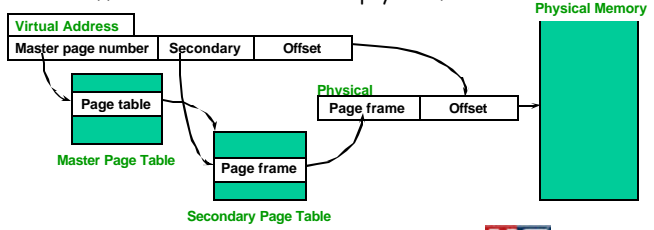
3/6/2007

### Managing Page Tables

- How can we reduce space overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- How do we only map what is being used?
  - Can dynamically extend page table...
  - Does not work if address space is sparse (internal fragmentation)
  - Can create per-region tables, if software-managed TLB
- Use another level of indirection

### Two-Level Page Tables

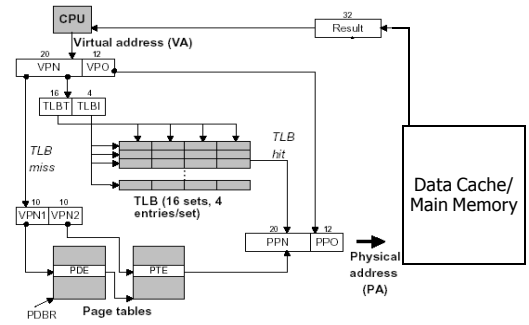
- Virtual addresses (VAs) have three parts:
- Master page number, secondary page number, and offset
  - Master page table maps VAs to secondary page table
  - Secondary page table maps page number to physical frame
  - Offset selects address within physical frame



### 2-Level Paging Example

- 32-bit virtual address space
  - 4K pages, 4 bytes/PTE
- How many bits in offset?
  - 4K = 12 bits, leaves 20 bits
- Want master/secondary page tables in 1 page frame each:
  - 4K/4 bytes = 1K entries. How many bits?
  - Master (1K) = 10, offset = 12, inner = 32 - 10 - 12 = 10 bits
- Note: this is why 4K is such a common page size on 32-bit architectures!

### Pentium Address Translation Redux



### 64-bit Address Spaces

- Suppose we just extended the hierarchical page tables with more levels
  - 4K pages → 52 bits for page numbers
  - Maximum 1024 entries per level → 6 levels
    - Too much overhead
  - 16K pages → 48 bits for page numbers
  - Maximum 4096 entries per level → 4 levels
    - Better, but still a lot
  - Instead, use hashed page tables
    - Page number hashes to page table entry
    - Linked list is searched for match

### Inverted Page Tables

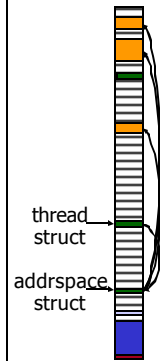
- Keep one table with an entry for each physical page frame
- Entries record which virtual page # is stored in that frame
  - Need to record process id as well
- Less space, but lookups are slower
  - References use virtual addresses, table is indexed by physical addresses
  - Use hashing (again!) to reduce the search time

### MIPS R2000 Virtual Memory Space

0xffffffff	KSEG2
0xc0000000	
0xbfffffff	KSEG1
0xa0000000	
0x9fffffff	KSEG0
0x80000000	
0x7fffffff	KUSEG
0x00000000	

- MMU defines 4 distinct regions with different properties:
  - KUSEG for user-addresses. Translated using paging, cacheable
  - KSEG0 for direct-mapped, cacheable kernel addresses
    - Translation to/from physical simply subtracts/adds 0x80000000 to V.A.
  - KSEG1 like KSEG0 but no caching
  - KSEG2 for kernel addresses that are translated using paging

### Physical Memory Space



Color Scheme:  
 Exception vector  
 Operating system code  
 Boot Stack (light blue)  
 OS Data (from kmalloc)  
 User Data

Operations:  
 Create thread → allocate kernel memory  
 Create address space → allocate kernel memory  
 Initialize address space → allocate user memory

## Addressing Page Tables

Where do we store page tables (which address space)?

- Physical memory
  - Easy to address, no translation required (or very simple translation, like KSEG0 in MIPS R2000)
  - allocated page tables consume memory for lifetime of VAS
- Virtual memory (OS virtual address space, KSEG2)
  - Cold (unused) page table pages can be paged out to disk
  - But, addressing page tables requires translation
  - How do we stop recursion?
  - Do not page the outer page table (called **wiring**)
- If we're going to page the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (fault, interrupt handlers)

CSC69115



3/6/2007

## Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- TLB's hide the cost for frequently-used pages

CSC69115



3/6/2007

## Paged Virtual Memory

- We've mentioned before that pages can be moved between memory and disk
  - This process is called **demand paging**
- OS uses main memory as a page cache of all the data allocated by processes in the system
  - Initially, pages are allocated from memory
  - When memory fills up, allocating a page in memory requires some other page to be evicted from memory
    - This is why physical memory pages are called "frames"
  - Evicted pages go to disk (where? the swap file)
  - The movement of pages between memory and disk is done by the OS, and is transparent to the application
- OS must keep track of use of each physical frame

CSC69115



3/6/2007

## Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
  2. When a process accesses the page, the invalid PTE will cause a trap (**page fault**)
  3. The trap will run the OS page fault handler
  4. Handler uses the invalid PTE to locate page in swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process
- But where does it put it? Have to evict something else
  - OS usually keeps a pool of free pages around so that allocations do not always cause evictions

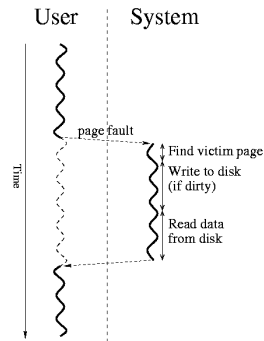
CSC69115



3/6/2007

## Early VM System (Atlas)

- Inverted page table (entry per physical page, records what virtual page is stored there)
  - Only 2048 entries, stored in registers, searched in parallel
- Missing pages fetched on demand from drum into core
  - Victim also selected on demand



CSC69115



3/6/2007

## Advanced Functionality

- Now we're going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on Write
  - Mapped files

CSC69115



3/6/2007

## Sharing

- Private virtual address spaces protect applications from each other
  - Usually exactly what we want
- But this makes it difficult to share data (have to copy)
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
- We can use **shared memory** to allow processes to share data using direct memory references
  - Both processes see updates to the shared memory segment
    - Process B can immediately read an update by process A
  - How are we going to coordinate access to shared data?

CSC69115



3/6/2007

## Sharing (2)

- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when page becomes invalid
- Can map shared memory at same or different virtual addresses in each process' address space
  - Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid (Why?)
  - Same: Less flexible, but shared pointers are valid (Why?)
- What happens if a pointer inside the shared segment references an address outside the segment?

CSC69115



3/6/2007

## Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create **shared mappings** of parent pages in child virtual address space
  - Shared pages are protected as read-only in child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
  - How does this help fork()? (Implemented as Unix vfork())

CSC69H15



3/6/2007

## Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of "open, read into buffer, operate on buffer, ..."
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address  $base + N$  refers to offset  $N$  in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    - Another use of the dirty bit in PTE

CSC69H15



3/6/2007

## Mapped Files (2)

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  - Virtual address space not backed by "real" files also called **Anonymous VM**
- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying
- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)

CSC69H15



3/6/2007

## Windows XP Virtual Memory

- 4KB page size on IA32 processors
  - 8 kB on the IA64
- 4GB virtual address space, upper 2 GB used by XP in kernel mode
- Multi-level page table
  - Page directory contains 1024 page directory entries (PDE) of size 4 bytes
  - PDEs point to page tables containing 1024 page table entries (PTEs) of size 4 bytes
- Page frames are tracked using a "page frame database" with one entry per page of physical memory; entry points to PTE which points to frame

CSC69H15



3/6/2007

## Summary

### Paging mechanisms:

- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)
  - Demand paged virtual memory (space)
- Recap address translation
- Advanced Functionality
  - Sharing memory
  - Copy on Write
  - Mapped files

Next time: Paging policies, Read Chapter 4.4-4.6

CSC6915



3/6/2007

## EXTRA SLIDES

The following slides were not covered in lecture. They review the entire address translation process, which you are expected to understand, from top to bottom.

CSC6915



3/6/2007

## Address Translation Redux

- We started this topic with the high-level problem of translating virtual addresses into physical address
- We've covered all of the pieces
  - Virtual and physical addresses
  - Virtual pages and physical page frames
  - Page tables and page table entries (PTEs), protection
  - TLBs
  - Demand paging
- Now let's put it together, bottom to top

CSC6915



3/6/2007

## The Common Case

- Situation: Process is executing on the CPU, and it issues a **read** to an address
  - What kind of address is it? Virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the **page number** of the address
  2. Common case is that the page number matches, returning a **page table entry (PTE)** for the mapping for this address
  3. TLB validates that the **PTE protection** allows reads
  4. PTE specifies which **physical frame** holds the page
  5. MMU combines physical frame & offset into a **physical address**
  6. MMU reads from that physical addr, returns value to CPU
- Note: **This is all done by the hardware**

CSC6915



3/6/2007

## TLB Misses

- At this point, two other things can happen
  1. TLB does not have a PTE mapping this virtual address
  2. PTE exists, but memory access violates PTE protection bits
- We'll consider each in turn

CSC69115



3/6/2007

## Reloading the TLB

- If the TLB does not have mapping, two possibilities:
  - 1. MMU loads PTE from page table in memory
    - Hardware managed TLB, OS not involved in this step
    - OS has already set up the page tables so that the hardware can access it directly
  - 2. Trap to the OS
    - Software managed TLB, OS intervenes at this point
    - OS does lookup in page table, loads PTE into TLB
    - OS returns from exception, TLB continues
- most machines will only support one method or the other
- At this point, there is a PTE for the address in the TLB

CSC69115



3/6/2007

## TLB Misses (2)

Note that:

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
  - Assuming page tables are in OS virtual address space
  - Not a problem if tables are in physical memory
  - Yes, this is a complicated situation
- When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
    - No fault occurs, hardware reads the memory address
  - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
    - Becomes a page fault...

CSC69115



3/6/2007

## Page Faults

- PTE can indicate a protection fault
  - Read/write/execute - operation not permitted on page
  - Invalid - virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
  - R/W/E - OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
  - Invalid
    - Virtual page not allocated in address space
      - OS sends fault to process (e.g., segmentation fault)
    - Page not in physical memory
      - OS allocates frame and reads it in

CSC69115



3/6/2007