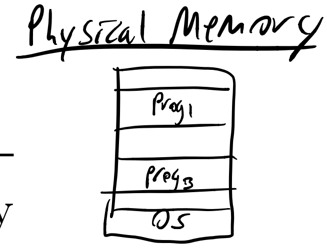
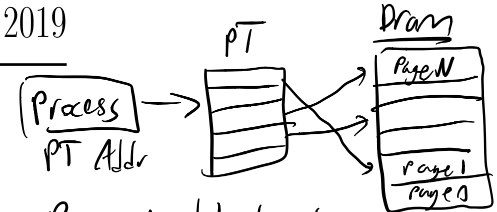
CS 61C Fall 2019

Virtual Memory

Discussion 10: November 4, 2019





Note size (VA) ≠ size (PA)
But the offset must (not always) be the same

Page table to DRAM may have no set ordering!

1 Addressing

Virtual Address (VA) What your program uses

Virtual Page Number (VPN)	Page Offset

translate — copy bits!

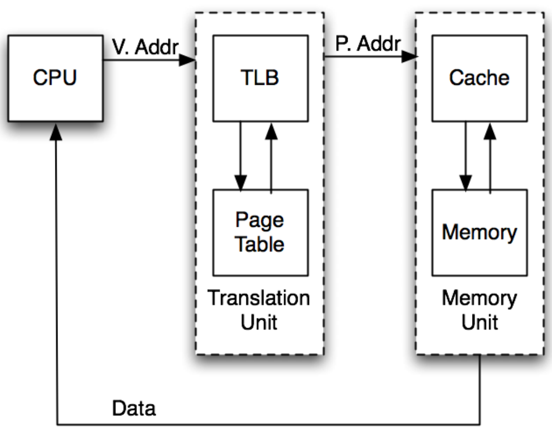
Physical Address (PA) What actually determines where in memory to go

Physical Page Number (PPN)	Page Offset

KiB 4
$2^{10} \cdot 2^2 = 2^{12} \to \log_2(2^{12}) = 12$ bit offset

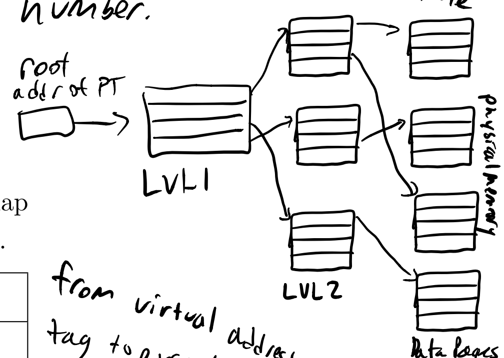
For example, with 4 KiB pages and byte addresses, there are 12 page offset bits since 4 KiB $= 2^{12}B = 4096B$.

Each process has its own page table



We also conserve space by making heirarchial pagetables. It is a tree where a page table level points to another page table level till it gets to the page number.

For a 32 bit machine:



from virtual address tag to physical address tag

Pages

A chunk of memory or disk with a set size. Addresses in the same virtual page map to addresses in the same physical page. The page table determines the mapping.

Valid	Dirty	Permission Bits	PPN
— *Page entry (VPN: 0)* —			
— *Page entry (VPN: 1)* —			

A lot of this is done by hardware & not the OS. This is up to the ISA of the language of the machine.

Each stored row of the page table is called a **page table entry**. There are $2^{\text{VPN bits}}$ such entries in a page table. Say you have a VPN of 5 and you want to use the page table to find what physical page it maps to; you'll check the 5th (0-indexed) page table entry. If the valid bit is 1, then that means that the entry is valid (in other words, the physical page corresponding to that virtual page is in main memory as opposed to being only on disk) and therefore you can get the PPN from the entry and access that physical page in main memory. The page table is stored in memory: the OS sets a register (the Page Table Base Register) telling the hardware the address of the first entry of the page table. If you write to a page in memory, the processor updates the "dirty" bit in the page table entry corresponding to that page, which lets the OS know that updating that page on disk is necessary (remember: main memory contains a subset of what's on disk). This is a similar concept as having a dirty bit for each cache block in a write-back cache, which we covered in lecture and in Lab 9. Each process gets its own illusion of full memory to work with, and therefore its own page table.

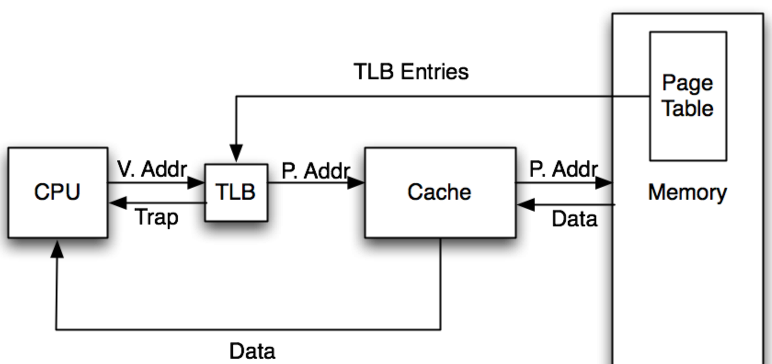
Protection Fault The page table entry for a virtual page has permission bits that prohibit the requested operation. This is how a segmentation fault occurs.

Page Fault The page table entry for a virtual page has its valid bit set to false. This means that the entry is not in memory, so we pull it from disk, add the page to memory (evicting another page if necessary), and add the mapping to the page table *and the TLB*. ← This is if it is not in DRAM. May be on disk. If so, evict a page in DRAM & Allocate the space & Move the page from disk to DRAM. If not valid, allocate new page in DRAM & evict if no room

Translation Lookaside Buffer

A cache for the page table. Each block is a single page table entry. If an entry is not in the TLB, it's a TLB miss. Assuming fully associative:

TLB Valid	Tag (VPN)	Page Table Entry		
		Page Dirty	Permission Bits	PPN
— *TLB entry* —				
— *TLB entry* —				



To access some memory location, we get the virtual page number (VPN) from the virtual address (VA) and first try to translate the VPN to a physical page number (PPN) using the translation lookaside buffer (TLB). If the TLB doesn't contain the desired VPN, we check if the page table contains it (remember: the TLB is a subset of the page table!). If the page table doesn't contain an entry for the VPN, then this is a page fault; memory doesn't contain the corresponding physical page! This means we need to fetch the physical page from disk and put it into memory, update the page table entry, and load the entry into the TLB, Then, we use the physical page and the offset of the physical address in the page to access memory as the program intended.

1.1 What are three specific benefits of using virtual memory?

(well really the full address space)

- Illusion of infinite memory (bridges memory and disk in memory hierarchy).

- Simulates full address space for each process so that the linker/loader dont need to know about other programs.

Isolation!

- Enforces protection between processes and even within a process (e.g. read-only pages set up by the OS).

1.2 What should happen to the TLB when a new value is loaded into the page table address register? → Invalidate TLB Entries

The valid bits of the TLB should all be set to 0. The page table entries in the TLB corresponded to the old process/page table, so none of them are valid once the page table address register points to a different page table

1.3 A processor has 16-bit addresses (2^4), 256 byte pages (2^8), and an 8-entry fully associative TLB with LRU replacement (the LRU field is 3 bits and encodes the order in which pages were accessed, 0 being the most recent). At some time instant, the TLB for the current process is the initial state given in the table below. Assume that all current page table entries are in the initial TLB. Assume also that all pages can be read from and written to. Fill in the final state of the TLB according to the access pattern below.

Question: How many bits does the TLB need to store?

Answer: VPN + PPN + valid + dirty + LRU

$8 + 8 + 1 + 1 + 3 = \boxed{21}$

8 entries so $\log_2(8)$ bits needed

2 4 6

Free Physical Pages ~~0x17~~, ~~0x18~~, ~~0x19~~

Access Pattern

1. 0x11|f0 (**Read**)
2. 0x13|01 (**Write**)
3. 0x20|ae (**Write**)
4. 0x23|32 (**Write**)
5. 0x20|ff (**Read**)
6. 0x34|15 (**Write**)

Page = 256 = 2^8 so $\log_2(2^8) = 8$ bit offset

Address Space Size − Page Offset = Tag

16 − 8 = 8 bit tag

Initial TLB

	VPN	PPN	Valid	Dirty	LRU	1	2	3	4	5	6
	0x01	0x11	1	1	0	1	2	3	4	4	5
2	~~0x00~~ 0x13	~~0x00~~ 0x17	~~0~~ 1	~~0~~ 1	7	7	0	1	2	2	3
	0x10	0x13	1	1	1	2	3	4	5	5	6
5 3	0x20	0x12	1	~~0~~ 1	5	5	6	0	1	0	1
4	~~0x00~~ 0x23	~~0x00~~ 0x18	~~0~~ 1	~~0~~ 1	7	7	7	7	0	1	2
1	0x11	0x14	1	0	4	0	1	2	3	3	4
	0xac	0x15	1	1	2	3	4	5	6	6	7
6	~~0xff~~ 0x34	~~0xff~~ 0x19	1	~~0~~ 1	3	4	5	6	7	7	0

Final TLB

VPN	PPN	Valid	Dirty	LRU
0x01	0x11	1	1	5
0x13	0x17	1	1	3
0x10	0x13	1	1	6
0x20	0x12	1	1	1
0x23	0x18	1	1	2
0x11	0x14	1	0	4
0xac	0x15	1	1	7
0x34	0x19	1	1	0

On miss, we use LRU Replacement to put the new page in.

steps:
1) Determine Vaddr Tag (VPN)
2) Check TLB for VPN
3) If exist check valid
↳ Hit
↳ else miss and load new page
4) Translate Vaddr (Virtual Page Number) to Paddr (Physical page number).

1. 0x11|f0 (**Read**): hit, LRUs: 1, 7, 2, 5, 7, 0, 3, 4
2. 0x13|01 (**Write**): miss, map VPN 0x13 to PPN 0x17, valid and dirty, LRUs: 2, 0, 3, 6, 7, 1, 4, 5

   because that is a free Physical Page

   got to validate it.

   dirty b/c it was a Write
3. 0x20|ae (**Write**): hit, dirty, LRUs: 3, 1, 4, 0, 7, 2, 5, 6
4. 0x23|32 (**Write**): miss, map VPN 0x23 to PPN 0x18, valid and dirty, LRUs: 4, 2, 5, 1, 0, 3, 6, 7

   was a write
5. 0x20|ff (**Read**): hit, LRUs: 4, 2, 5, 0, 1, 3, 6, 7
6. 0x34|15 (**Write**): miss and replace last entry, map VPN 0x34 to 0x19, dirty, LRUs, 5, 3, 6, 1, 2, 4, 7, 0

   not in TLB & TLB full so evict & write to mem the LRU

   since LRU is not dirty, we do not need to write the page back to disk.