Virtual Memory Overview

Virtual address (VA): What your program uses

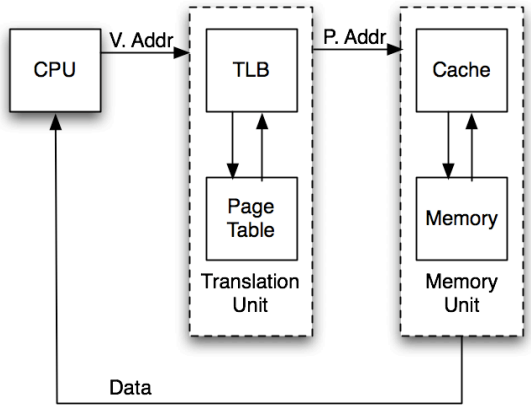
Virtual Page Number	Page Offset

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

Physical Page Number	Page Offset

e.g. With 4 KiB pages and byte addresses, 2^(page offset bits) = 4096, so page offset bits = 12

The Big Picture: Logical Flow
Translate VA to PA using the TLB and Pa
ge Table. Then use PA to access memory as the program intended.



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

The Page Table

Index = Virtual Page Number (not stored)	**Page Valid**	**Page Dirty**	**Permission Bits (read, write, ...)**	**Physical Page Number**
0				
1				
2				
...				
(Max virtual page number)				

Each stored row of the page table is called a **page table entry** (the grayed section is the first page table entry). The page table is stored *in memory*; the OS sets a register telling the hardware the address of the first entry of the page table. The processor updates the "page dirty" in the page table: "page dirty" bits are used by the OS to know whether updating a page on disk is necessary. Each process gets its own page table.

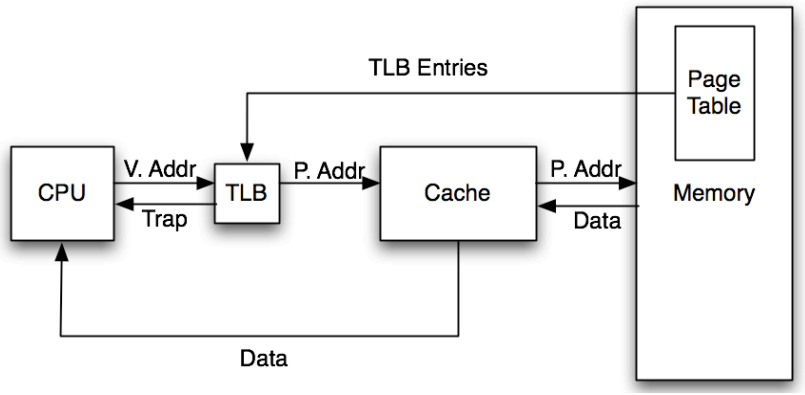
- **Protection Fault**--The page table entry for a virtual page has permission bits that prohibit the requested operation
- **Page Fault**--The page table entry for a virtual page has its valid bit set to false. The entry is not in memory.

The Translation Lookaside Buffer (TLB)

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 Entry Valid	Tag = Virtual Page Number	Page Table Entry		
		Page Dirty	Permission Bits	Physical Page Number
…	…	…	…	…

The Big Picture Revisited



Exercises

1) What are three specific benefits of using virtual memory?

Bridges memory and disk in memory hierarchy.
Simulates full address space for each process.
Enforces protection between processes.

2) What should happen to the TLB when a new value is loaded into the page table address register?

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

3) Fill in the following formulas below.

#Offset Bits = $\log_2$(page size in bytes)

#Virtual Address Bits = #(VPN Bits) + #(Page Offset bits)

#Physical Address Bits = #(PPN Bits) + #(Page Offset bits)

#Bits per row of PT = #(PPN Bits) + #(Extra bits [valid, dirty bits, etc.])

4) Fill this table out!

Virtual Address Bits	Physical Address Bits	Page Size	VPN Bits	PPN Bits	Bits per row of PT (4 extra bits)
32	32	16KiB	18	18	22
32	26	8KiB	19	13	17
36	32	32KiB	21	17	21
40	36	32KiB	25	21	25
64	40	64KiB	48	24	28

5) A processor has 16-bit addresses, 256 byte pages, 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.

Free physical pages: `0x17, 0x18, 0x19`

Access pattern:

Read	`0x11f0`
Write	`0x1301`
Write	`0x20ae`
Write	`0x2332`
Read	`0x20ff`
Write	`0x3415`

Initial TLB

VPN	PPN	Valid	Dirty	LRU
`0x01`	`0x11`	1	1	0
`0x00`	`0x00`	0	0	7
`0x10`	`0x13`	1	1	1

0x20	0x12	1	0	5
0x00	0x00	0	0	7
0x11	0x14	1	0	4
0xac	0x15	1	1	2
0xff	0x16	1	0	3

Read 0x11f0: hit, LRUs: 1,7,2,5,7,0,3,4
Write 0x1301: miss, map VPN 0x13 to PPN 0x17, valid and dirty, LRUs: 2,0,3,6,7,1,4,5
Write 0x20ae: hit, dirty, LRUs: 3,1,4,0,7,2,5,6
Write 0x2332: miss, map VPN 0x23 to PPN 0x18, valid and dirty, LRUs: 4,2,5,1,0,3,6,7
Read 0x20ff: hit, LRUs: 4,2,5,0,1,3,6,7
Write 0x3415: miss and replace last entry, map VPN 0x34 to 0x19, dirty, LRUs, 5,3,6,1,2,4,7,0

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

6) The specs for a MIPS machine's memory system that has one level of cache and virtual memory are:

- 1MiB of Physical Address Space
- 4GiB of Virtual Address Space
- 4KiB page size
- 16KiB 8-way set-associative write-through cache, LRU replacement

- 1KiB Cache Block Size
- 2-entry TLB, LRU replacement

The following code is run on the system, which has no other users and process switching turned off. Assume that the page table can hold 11 amounts of pages. **To make things "easier," pretend that the compiled binary for the following program does not require a page to be implemented for questions (a-g).** **`malloc`** should return address 0x100000 (you should be "block-aligned" and "page-aligned."

```
#define NUM_INTS 8192                          // This many ints...

int *A = (int *)malloc(NUM_INTS * sizeof(int));

int i, total = 0;

for(i = 0; i < NUM_INTS; i += 128) A[i] = i;

for(i = 0; i < NUM_INTS; i += 128) total += A[i];       // SPECIAL
```

a) What is the T:I:O bit breakup for the cache (assuming byte addressing)? T:_9_I:_1_O:_10_

b) What is the VPN : PO bit breakup for VM (assuming byte addressing)? _20_:_12_

c) What is the PPN : PO bit breakup for PM (assuming byte addressing)? _8_:_12_

d) How many page faults can occur in the **worst-case scenario** before the "SPECIAL" for loop?

Well, in the worst case scenario, the TLB is empty/flushed prior to the process and the page table is empty. Ignoring any possibility that the compiled binary code takes up any page, we know that our array takes up 8192 * 4 bytes = 2^{15} bytes, which is equivalent to

$2^3 * 2^{12}$ bytes, which is equal to 8 pages. Thus, we would have 8 page faults in the worst case.

For the following questions, only consider the line marked “SPECIAL”. However, the current state of the process should be whatever is expected after the prior `for` loop. Your answer can be a fraction:

e) Calculate the hit percentage for the cache

1/2 = 50%. Our cache is 2^{14}B 8-way set associative. We know that A is 8192 * 4 bytes = 2^{15} bytes, which is the equivalent to 2 cache sizes. We also know that we are block and page aligned. We read from A[i] once. If we look at accesses in the first block, we can see that there is a single miss. We then

move to i = 128, which is 512 bytes away and proceed to hit. Of these two memory accesses, there is one hit. This rate repeats for all blocks.

f) Calculate the hit percentage for the TLB

7/8 = 87.5%. We know that one page is 2^{12} Bytes. As seen in c), we are incrementing by 512 bytes, which means that there are $2^{12}/2^{9}=2^{3}$ page accesses per page. Of these accesses, we miss once to load in the page from disk to memory and then proceed to hit the other 7 times.

g) Calculate the page hit percentage for the page table

100% (look at the for loop before SPECIAL).

h) How would having the compiled binary for this program take up one page-sized amounts of data affect our problems for the "SPECIAL" loop? What about for part d?

We can potentially have one TLB page designated for code and the other for all memory accesses of our array (our TLB has 2 entries). For part d, we'd have one extra page fault however because our page table would not originally contain the page where our code is located.

i) What would happen to our TLB performance if **`malloc`** perhaps returned an address that was not "page aligned" and instead was for instance, 0x0FFFFC? What about our page fault count for part d?

Malloc would not have returned a page-aligned address. Our array is 8 page sizes long. If it was not page-aligned, we could potentially address it in such a way that it spanned 9 different pages in virtual memory, and in turn required three different mappings to physical memory. This could reduce our hit percentage for the TLB, and also increase our page fault count in part d.