Cache Review → Virtual Memory → Advanced Caches

Remember the difference between a "Block" and a "Page".
- A "Block" is a small chunk of memory that fits within a Cache line.
- A "Page" is a larger chunk of memory that is used to virtualize a Process' view of memory from the actual, Physical Memory.

Virtual Memory

Topics:
- Virtual → Physical Addr Translation
- Page Tables
- Page Table Sizes
- Multi-Level Page Tables
- TLB
- Addr Decomposition for Page Tables
  - Virtual Page #
  - Page Offset

This lesson discusses the difference between virtual and physical memory. Virtual memory is used by programs and must be translated to physical memory. Since the translation can be slow, a Translation Look-Aside Buffer (TLB) is used.

Why Virtual Memory

Hardware view of memory - the memory is a block of memory with a specific address allocated to each location.

Programmer's View of memory - memory is large arrays with many more addresses than the actual memory. Each program has its own view of the memory.

Virtual memory reconciles the programmer's view of memory with the hardware memory.

Processor's View of Memory

The processor sees the physical memory (actual memory).

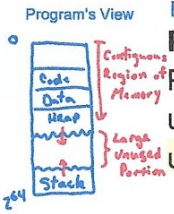
The actual amount of memory is sometimes < 4 GB

Although the address is 64 bits, there is never this much memory (16 Exabytes/process)

Basically what this is saying is that even though a process may think it can access memory with a 64 bit address, we don't actually have 2^64 bytes of Main Memory available to us (2^64 = 16 Exabytes)

The amount of Physical Memory that we have is usually less (by a lot) than what programs can access.

From the Processor's perspective of Memory, there is a 1:1 relationship between an address and an actual Byte/Word in memory. From this perspective, an address ALWAYS goes to the same physical location.



Program's View of Memory

Programs sees the stack and the heap, with a large gap between the two. The gap is never fully used. The programs see much more memory than the actual memory, so virtual memory is used.

Separate programs can also have different virtual memory addresses that actually map to the same physical memory, so they're effectively sharing that memory. How do we reconcile this?

Mapping Virtual → Physical Memory

Physical memory is divided into 4k Byte frames, the virtual memory is divided into 4k Byte pages.

The operating system uses page tables to map the pages to frames. If programs use the same physical memories, then the pages are mapped to the same page.

"Pages" are to "Frames" like how "Memory Blocks" are to "Cache Lines". Note that Pages are MUCH bigger (ex: 4 KB) than Memory Blocks (ex: 32 B). They serve different purposes!

Where is the Missing Memory

Some of the virtual memory pages are actually stored on a hard disk, rather than in memory.

Processor Cannot Directly Access Pages Stored on Disk, only Main Memory. In order to be used by the Processor, the pages on disk must be brought into Main Memory first.

Virtual to Physical Translation

The virtual address is separated into the virtual page number (the MSB) and the page offset (LSB).

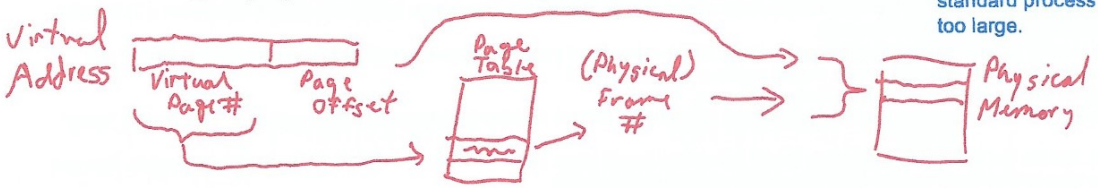
The Virtual Page Number portion of the Virtual Address is used to index into the Page Table for the Process. The Page Table entry will store the Physical Frame Number that the Processor can use.

The virtual page is translated (using a page table) to the physical frame number. This physical frame number is combined with the page offset to determine the physical memory.

Size of Flat Page Table

Flat page tables = one entry for every page in the table .

Each process needs its own Page Table, but a table for a standard process could need up to 2^20 (1 million) entries. This is too large.



Some of these pages are never accessed because of the large unused chunk of a process's memory that is UNUSED and left for stack/heap growth

Since some of these pages are never accessed, this table is unnecessarily large.

(Flat) Page table size = (Virtual Memory / Page Size) * Size of Entry

Of Page Table Entries

Remember that EACH PROCESS gets its own Page Table too!

Multi-Level Page Tables

Multi-level page tables are used for large virtual memories.

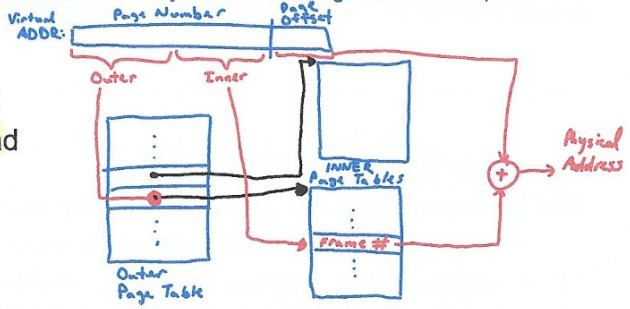
Multi-level page tables avoid having page entries for the unused virtual memory addresses.

Multi-Level Page Table Structure

The page number is partitioned into inner and outer page numbers, leading to a hierarchical page table.

When calculating the size of a Multi-Level Page Table, first figure out how many actual Pages are used by the process first, and then work backwards starting with the innermost Page Table level. # of Page Tables Used = # of Innermost Page Table Entries Needed. Then from that #, figure out how many Innermost Page TABLES are needed to hold that many of Innermost Page Table Entries, etc.

Choosing the Page Size



Larger Pages:

mean smaller page tables -which is good

mean internal fragmentation - which is bad

Smaller Pages

large page tables - bad

A compromise between the large and small pages means a page size that is a few KBytes to a few MBytes.

The Page Tables are not stored on the Processor Chip (too large and too many {1 per process} to fit). Instead, they're stored in Main Memory.

Memory Access Time with V→ P Translation

Additional time must be allowed for the virtual to physical address translation.

The extra steps required for translation are:

-Computer the physical address of the page table entries (fast)

-read the page table entry

-compute the physical address (fast)

For Multi-Level Page Tables, we'd have to do this multiple times

The translation can be longer than the memory access time.

Translation Look-Aside Buffer (TLB)

For a Multi-Level Page Table, only the final translation (aka the Physical Frame #) is stored in the TLB, not any of the intermediate steps. Makes it fast.

To speed up the translation, a special, small cache is used, called the TLB.

If we have a TLB Miss, then we need to pay the penalty to go to Main Memory (potentially multiple times in a Multi-Layer Page Table) to perform the V->P translation, and then store the final results in the TLB.

For a TLB miss the operating system (Software TLB Miss Handling) or the processor updates the TLB (Hardware TLB Miss Handling).

TLB Organization

Remember, each entry in a TLB is a mapping from a Virtual Page Number to a Physical Page. So each entry technically "covers" a full Page of Memory. So if a TLB has 64 entries, then it can cache the translations of 64 Pages. We usually want enough entries so that the size of our "memory coverage" (# of entries or "Page Coverage" times Page Size) is at least as big as the total size of our Cache, if not more.

A TLB is similar to a cache.

TLB tends to be Fully or Highly Associative

TLB Size = 64 to 512 entries (if more entries are needed, use a two level TLB)

Remember the Memory access process (*** for "Physically Indexed-Physically Tagged" (PIPT) Caches - explained in the "Advanced Caches" Lectures ***):

1) Start with the Virtual Address of a Process

2) Convert the Virtual Address to a Physical Address. Split the Virtual Address into the Virtual Page Number and the Page Offset.

2.1) Take the Virtual Page # of the Virtual Address and use it to do a lookup in the TLB. If it hits, then we get the Physical Frame Number so continue to Step 3. Otherwise, a TLB Miss requires the Processor to proceed with the V -> P Translation process (detailed in Step 2.2).

2.2) For each Level in the Page Table (if any) we need to go to Main Memory and do lookups until we find the innermost page table where we will find the corresponding Page Table Entry for our specific Virtual Address. This will give us the Physical Frame Number, which we can continue to use in Step 3.

3) Take the Physical Frame Number and append the Page Offset to get the final Physical Memory Address.

4) Look up the Physical Memory Address in the Cache first, which means splitting the Physical Memory Address into a Block Number (which can be further split as a "Tag" and an "Index" section) and a Block Offset.

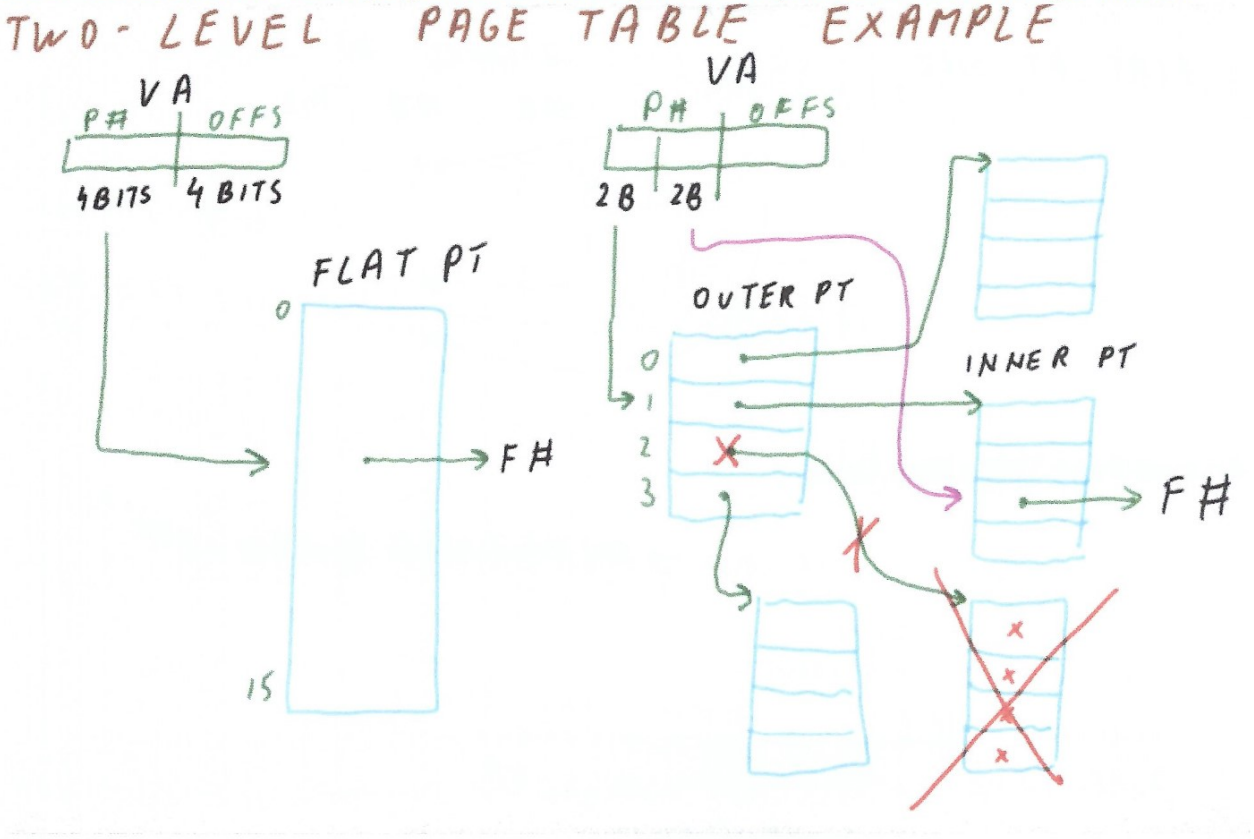
5) Using the Index (if any- depends on the type of Cache), find the Set that the Block belongs to and search the Cache Lines within that Set for a matching Tag. If there's a match, then pull the data from that Cache Line. Otherwise there's a Cache Miss and we need to go to Main Memory for that Physical Address.

Additional Virtual Memory Notes

Remember the Memory access process:

1. Start with the Virtual Address of a Process
2. Convert the Virtual Address to a Physical Address. Split the Virtual Address into the Virtual Page Number and the Page Offset.
    a. Take the Virtual Page # of the Virtual Address and use it to do a lookup in the TLB. If it hits, then we get the Physical Frame Number so continue to Step 3. Otherwise, a TLB Miss requires the Processor to proceed with the V -> P Translation process (detailed in Step 2.b).
    b. For each Level in the Page Table (if any) we need to go to Main Memory and do lookups until we find the innermost page table where we will find the corresponding Page Table Entry for our specific Virtual Address. This will give us the Physical Frame Number, which we can continue to use in Step 3.
3. Take the Physical Frame Number and append the Page Offset to get the final Physical Memory Address.
4. Look up the Physical Memory Address in the Cache first, which means splitting the Physical Memory Address into a Block Number (which can be further split as a "Tag" and an "Index" section) and a Block Offset.
5. Using the Index (if any- depends on the type of Cache), find the Set that the Block belongs to and search the Cache Lines within that Set for a matching Tag. If there's a match, then pull the data from that Cache Line. Otherwise there's a Cache Miss and we need to go to Main Memory for that Physical Address.

How does a multi-level page table save space compared to a Flat Page Table?

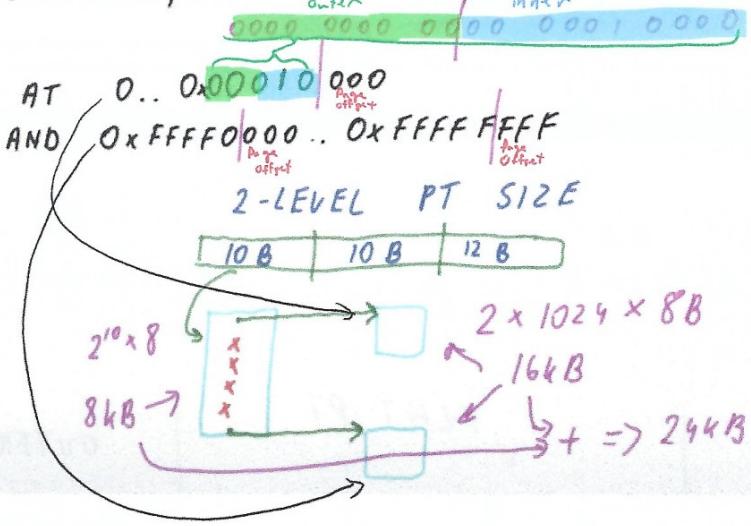


The summary is that if we had ALL the inner page tables that we needed, their combined size would add up to be the same size as the corresponding Flat Page Table. However, since a large amount of the Virtual Memory address space is unused by a process, we can actually decide to NOT provide Inner Page Tables for those portions of the Virtual Address Space. We can't do that with a Flat Page Table. This allows us to save considerable space!

So you can think of a 2-Level Page Table as a Flat Page Table that has been partitioned so that we can cut out sections of the unused Virtual Address Space.

TWO-LEVEL PAGE-TABLE SIZE

- 32-BIT ADDRESS SPACE
- 4 kB PAGE
- 1024-ENTRY OUTER PAGE TABLE, 1024-ENTRY INNER TABLES
- PT ENTRY 8 BYTES
- PROGRAM USES VM AT 0.. 0x00010000 AND 0xFFFF0000 .. 0xFFFFFFFF



FLAT PT SIZE

$\frac{2^{32}}{2^{12}} = 2^{20}$

8 MB

4-Level Page Table Size Quiz:

4-LEVEL PAGE TABLE QUIZ

- 64-BIT VIRTUAL ADDR SPACE → Total Virtual Address Space Size = 2^{64}
- 64kB PAGE SIZE
- 8B PAGE TABLE ENTRY
- PROGRAM USES ONLY ADDRESSES 0.. 4GB

A FLAT PAGE TABLE SIZE IS $2^{[51]}$

B 4-LEVEL PAGE TABLE (PAGE # SPLIT EQUALLY) USES _____ kB

A) Flat Page Table Size = (# of Page Table Entries)(Size of each Page Table Entry)

# of Page Table Entries = $\frac{\text{Total Size of Virtual Address Space}}{\text{Page Size}} = \frac{2^{64}\,B}{64\,KB} = \frac{2^{64}\,B}{2^6 \cdot 2^{10}\,B} = \frac{2^{64}\,B}{2^{16}\,B} = 2^{48}$

Flat Page Table Size = $(2^{48})(8B) = (2^{48})(2^3\,B) = 2^{51}\,B$

size

B) • Virtual Address is 64 Bits long, which ones of those are the Page Offset & which ones are used for the Page # (split equally by 4)?

• We know the page size, so we can find out how many bits we need to index into that page size

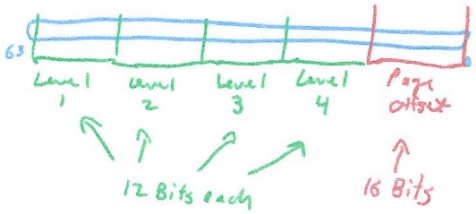
$2^X = 64KB = 2^6 \cdot 2^{10} = 2^{16} \rightarrow X = 16$ Bits

So the Least Significant 16 Bits of the 64 Bit Virtual Address are needed for the Page Offset

64 - 16 = 48 Bits for the Page #.

Since we have 4 Levels divided equally, each level uses:

48/4 = 12 Bits



Each table can have 2^{12} entries

The programs only uses addresses 0 - 4GB in its Virtual Address Space.

• So how many pages does it need to cover this range?

$4GB = 2^{32}$ Bytes ÷ each Page is 64 KB ($2^6 \cdot 2^{10} = 2^{16}$ B)

so it needs $\frac{2^{32}}{2^{16}} = 2^{16}$ Pages

• Starting @ our innermost page table level (Level 4), how many pages can a Level 4 page table reference? Since each table can have 2^{12} entries, that means that a single Level 4 Page Table can reference up to 2^{12} pages. That's not enough!

• If each Level 4 Page Table can reference up to 2^{12} pages, then how many Level 4 Page Tables do we need in order to be able to reference 2^{16} pages?

$\frac{2^{16} \text{ Pages needed}}{2^{12} \text{ Pages per Level 4 Table}} = 2^4 = 16$ Level 4 Tables needed

◦ This means we need 16 entries in a Level 3 Page Table in order to reference each of these 16 Level 4 Tables, which we can all fit in a single Level 3 Page Table (which, again, holds up to 2^{12} entries).

• We'd also need 1 Level 2 page table (we'd only use one of its 2^{12} entries to reference the single Level 3 page table).

• And we'd need 1 Level 1 page table (again, using only one of its 2^{12} entries to reference the single Level 2 page table).

Total # of Page Tables needed? : 1 Level 1 + 1 Level 2 + 1 Level 3 + 16 Level 4 = 19 Page Tables

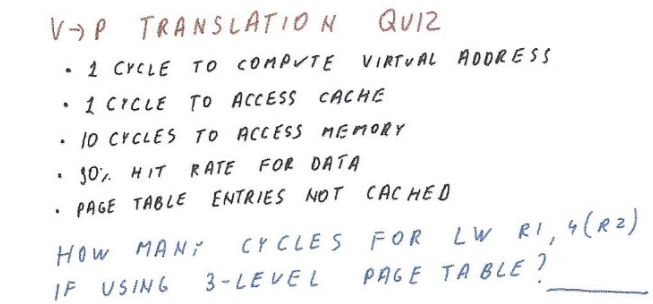
Total size of all Page Tables? $= (19 \text{ Page Tables})\left(\frac{2^{12} \text{ Entries}}{1 \text{ Page Table}}\right)\left(\frac{8 \text{ B}}{\text{Entry}}\right) = 19(2^{12})(2^3)\text{ B} = 19(2^{15})$

$= 19(2^5)(2^{10})$

$= 19(2^5)$ KB

$= 608$ KB

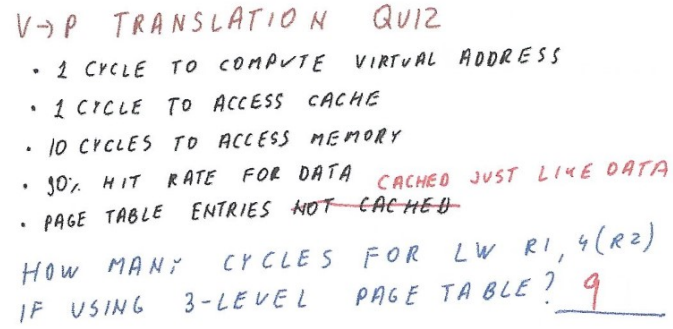
V->P Translation Quiz



What're the steps that occur for this 1 instruction?
1) Calculate the Virtual Address (1 cycle)
2) Convert the Virtual Address to a Physical Address. What does this entail? We need to access Memory 3 times for each Page Table lookup. Since theres 3 Levels, that means that we do 3 Look ups to Memory. (3 * 10 cycles)
3) Once we have the Physical Address, we first check the cache to see if the data at that Physical Memory location is cached (1 Cycle)
4~) If the data isn't in the cache (10% chance), then we need to access main memory again (10 cycles). That means 10% of the time, we're paying the penalty of 10 cycles, so on average, each memory access will cost us 10*10% = 1 cycle.

As a result, it will take (1+30+1+1) = 33 cycles

V->P Translation Quiz 2



What're the steps that occur for this 1 instruction?
1) Calculate the Virtual Address (1 cycle)
2) Convert the Virtual Address to a Physical Address. What does this entail? We need to access Memory 3 times for each Page Table lookup. Since theres 3 Levels, that means that we do 3 Look ups to the CACHE (3 * 1 = 3 Cycles). 10% of the time, each of these 3 Cache Lookup will be a Miss and result in a lookup from Memory (3 * 10% * 10 = 3 Cycles)
3) Once we have the Physical Address, we first check the cache to see if the data at that Physical Memory location is cached (1 Cycle)
4~) If the data isn't in the cache (10% chance), then we need to access main memory again (10 cycles). That means 10% of the time, we're paying the penalty of 10 cycles, so on average, each memory access will cost us 10*10% = 1 cycle.

As a result, it will take (1+3+3+1+1) = 9 cycles

$$1 + 3 \times (1 + 0.1 \times 10) + 1 + 0.1 \times 10$$

TLB PERFORMANCE Q1/12

- 1 MB ARRAY, READ ONE BYTE AT A TIME FROM START TO END, DO THIS 10 TIMES
- NO OTHER MEM ACCESSED
- 4kB PAGE, 128-ENTRY L1 TLB, 1024-ENTRY L2 TLB
- TLBS INITIALLY EMPTY, ARRAY PAGE-ALIGNED, DIRECT-MAPPED TLBS
- L1 TLB 10483200 HITS, 2560 MISSES
- L2 TLB 2304 HITS, 256 MISSES

$1\ MB = 2^{20}\ B$

How many pages is this?

$$\frac{2^{20}\ B}{4kB} = \frac{2^{20}\ B}{2^2 \cdot 2^{10} B} = \frac{2^{20} \cancel{B}}{2^{12} \cancel{B}} = 2^8 = 256 \text{ pages}$$

So if the TLBs are initially empty, the first access to each page will cause a miss, but all subsequent accesses will be a hit. Since each page is 4kB & we access memory 1 Byte @ a time, for each page we will have 1 miss + (4kB − 1) hits.

On the first loop:
- L1: 256 (4kB − 1) Hits, 256 Misses
- L2: (No hits) 256 Misses

On loops 2–10:
- L1: 256 Misses, 256 (4kB − 1) Hits
- L2: 256 Hits, 0 Misses

L1 Hits: $10[256(4kB-1)] = 10[2^8(2^2 \cdot 2^{10} - 1)] = 10[2^{20} - 2^8] = 10483200$

L1 Misses: $10(256) = 2560$

L2 Hits: $0 + 9(256) = 2304$

L2 Misses: $256 + 9(0) = 256$