CS 33 Discussion 9

June 2, 2017

Agenda

- OpenMP Lab
- Virtual Memory
- I/O

Logistics

OpenMP Lab

Final Exam

OpenMP Lab

Objective -

Make the code fast.

OpenMP Lab

- Getting started:

– The openmplab.tgz file is available on CCLE

– Copy it to linux server

– Unzip the folder.

OpenMP Lab

To run the basic code:

```
-make seq
```

This creates an executable called seq

OpenMP Lab

Tools to be used:

1. gprof - to find performance of functions
2. OpenMP - uses thread parallelism to speed up code

OpenMP Lab

- To use gprof:
- Compile with GPROF=1
  – ex. `make seq GPROF=1`
- Execute the executable (this produces gmon.out)
  – ex. `./seq`
- Run gprof with the executable as the argument
  – ex. `gprof seq | less`
  – The command gprof seq normally prints out lots of text. The "`|less`" redirects the output as input into the "less" command, which allows you to read it more easily.
  – Quit "`less`" with q.

OpenMP Lab

The code performs filtering algorithm on a media file.

The main function is filter, found in **filter.c**. It calls six functions `func0` through `func5`.

Each function consists of a for-loop, and your job is to try to optimize the given code and extract parallelism from each function using OpenMP.

OpenMP Lab

`gprof` - profiling tool

When optimizing a large program, it is useful to profile it to determine which portions take up the largest portion of the execution time.

There is no point in optimizing code that only takes up a small fraction of the overall computations.

OpenMP Lab

● OpenMP is an extension to C/C++ (and Fortran) that allows for a simple and convenient way of launching threads to do work in parallel among multiple processors.
● You'll be specifying a bunch of pre-processor directives.
● Preprocessor directives are statements that are marked with “#”.
● Statements marked with these are instructions that tell the processor what to do:

- `#define min(a, b) (a < b ? a : b)`
- `#include <stdio.h>`
- `#don't become sentient`

OpenMP Lab

● The first step of compilation is the preprocessing step where the compiler will find all of the directives and follow the instructions.
● OpenMP exposes an interface for multi-threading that is almost purely based on these preprocessor instructions.
● Other than some helper functions defined in omp.h.

Ex.

- `omp_set_num_threads(int);`
- `omp_get_thread_num();`

OpenMP Lab

- Ex:

```
int main()
{
    int a = 0;
    printf(“%d\n”, a);
}
```

- Say you want to parallelize this so that 4 threads run this code.

OpenMP Lab

Ex:

```
int main()
{
    omp_set_num_threads(4);     //Use 4 threads
    #pragma omp parallel        //Define the following
        {                       //block as parallel code
            int a = 0;          //run by each thread
            printf("%d\n", a);
        }
}
```

OpenMP Lab

Ex:

```
int main()
{
    #pragma omp parallel
    {
        int a = 0;
        printf(“%d\n”, a);
    }
}
```

- If the number of threads is not specified, the default is used.

OpenMP Lab

● Upon hitting the block specified by the `#pragma omp` directive, the single thread of execution branches off into several parallel threads.

● When each thread reaches the end of the block:

```
#pragma omp parallel
{
...
} ← Here
```

● ...each thread will wait for all of the other threads before continuing. This is known as a “barrier”.

OpenMP Lab

- The “parallel” keyword means “launch threads in parallel”
- This is the basis how OpenMP works.
- Synchronization and control can also be done via these directives. Within a parallel block, you can also have other directives that specify things.

OpenMP Lab

- critical:

```
#pragma omp critical
{
    <CRITICAL SECTION>
}
```

Only one thread can enter the critical section at a time. Other threads must wait.

(what does this remind you of?)

OpenMP Lab

- atomic:

```
#pragma omp atomic
{
    tmp += 1;
}
```

– Only one thread can perform a particular read/write.

This only applies to single variable that is read/write.

OpenMP Lab

- An OpenMP critical section is completely general - it can surround any arbitrary block of code. You pay for that generality, however, by incurring significant overhead every time a thread enters and exits the critical section (on top of the inherent cost of serialization).
- (In addition, in OpenMP all unnamed critical sections are considered identical (if you prefer, there's only one lock for all unnamed critical sections), so that if one thread is in one [unnamed] critical section as above, no thread can enter any [unnamed] critical section. As you you might guess, you can get around this by using named critical sections).
- An atomic operation has much lower overhead. Where available, it takes advantage on the hardware providing (say) an atomic increment operation; in that case there's no lock/unlock needed on entering/exiting the line of code, it just does the atomic increment which the hardware tells you can't be interfered with.
- The upsides are that the overhead is much lower, and one thread being in an atomic operation doesn't block any (different) atomic operations about to happen. The downside is the restricted set of operations that atomic supports.

OpenMP Lab

- barrier:

– `#pragma omp barrier`

– All threads must wait here for all threads to complete before continuing.

OpenMP Lab

```
int main()
{
    int a = 0;
    #pragma omp parallel
    {
        a++;
        printf("%d\n", a);
    }
}
```

OpenMP Lab

● Data sharing:
– By default, a variable declared inside of a parallel block is private to each block.
– By default, a variable declared outside of a parallel block is shared among all of the threads of a parallel block that accesses the variable. In the previous example, “a” was the same “a” among all threads.
– If we want to share the variables in a way that's more graceful and predictable, we'll have to use critical or atomic but then, we'll essentially be doing code in serial.

OpenMP Lab

● A variable within a parallel block can be made private to each thread by using the “private” keyword

```
int main()
{
    int a = 0;
    #pragma omp parallel private(a)
    {
        a++;
        printf(“%d\n”, a);
    }
}
```

OpenMP Lab

- Warning: the variable will also be private relative to the original declaration.

```
int main()
{
    int a = 123;
    #pragma omp parallel private(a)
    {
        a++;
        printf("%d\n", a);
    }
}
```

- The “a” inside the block will not be initialized to 123. It will be uninitialized.

OpenMP Lab

● If you do want to make the private variable in the block initialized to the value specified in the original thread, use “firstprivate”.

```
int main()
{
    int a = 123;
    #pragma omp parallel firstprivate(a)
    {
        a++;
        printf(“%d\n”, a);
    }
}
```

● The “a” inside the block will now be initialized to 123.

OpenMP Lab

Make sure to use “make check” to confirm that the output.txt generated by the program is the same as correct.txt. If silent, the files are the same.

When you run make checkmem, that file will be analyzed to verify that all allocated memory was eventually freed.

OpenMP Lab

Extremely good tutorial for beginners to OpenMP:

http://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Other resources:

http://www.openmp.org/resources/tutorials-articles/

Virtual Memory

Memory

- We have covered caches and why it was necessary to have them.
- We need DRAM/Main Memory since we need some “large memory” in order to adequately contain our program's addressable space.
- Unfortunately, this large memory is too slow to be our primary source of satisfying memory accesses.

Memory

- We now have an adequate model for describing how a program is executed.
- Instruction addresses are fed into the processor and memory accesses generally must be satisfied by the cache.
- Memory addresses range from 32, 48, and 64 (?) bits. If we consider the 32 bit case, this implies that our memory must have a capacity of 2^32 bytes.

Memory

- That suggests that main memory must be at least 2^32 or approximately 4 GB.
- But that can't be right, computers can function just fine with less memory than that.
- For that matter even if you had 4GB of memory, how could you run multiple processes if each process expects to have 4 GB of personal memory space.
- What if a computer was running 100 processes?

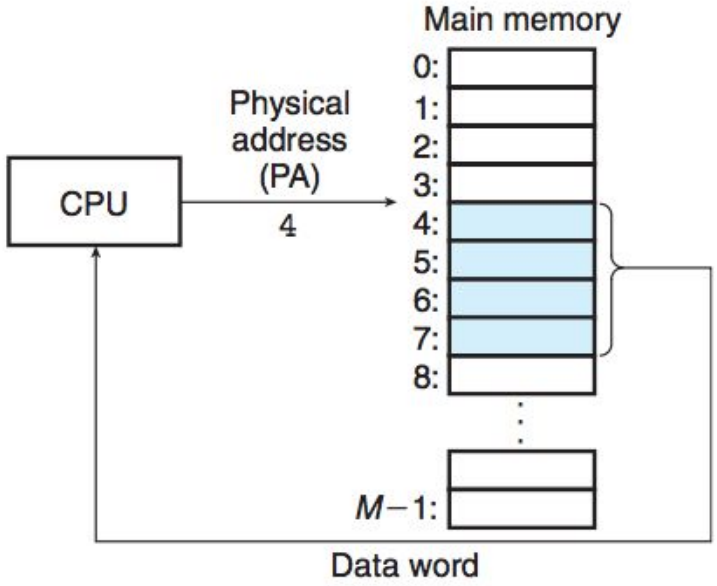
Virtual Memory: Basics

- Addresses that are known to a program are actually virtual addresses in a scheme known as virtual memory.
- Each process expects to be the sole owner of 2^32 bytes of memory, but it would be impractical to have actually that much DRAM
- Instead, we can simply allow each process to believe that they own the entire addressable space.

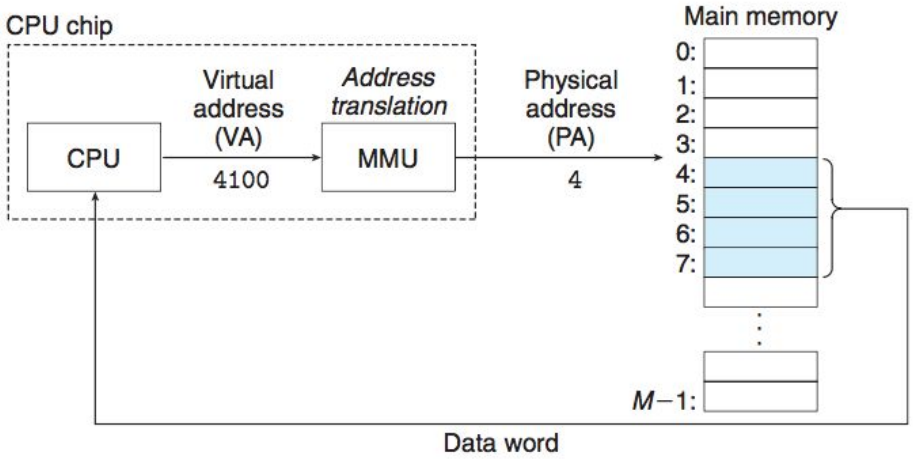
Virtual Memory: Basics

- In reality, the physical memory must be shared between all of the processes.
- The actual contents of each process' memory must actually be stored on the much larger (but much slower) disk.
- Main memory thus becomes a “cache” for the virtual memory on disk.

Physical Addressing vs Virtual Addressing



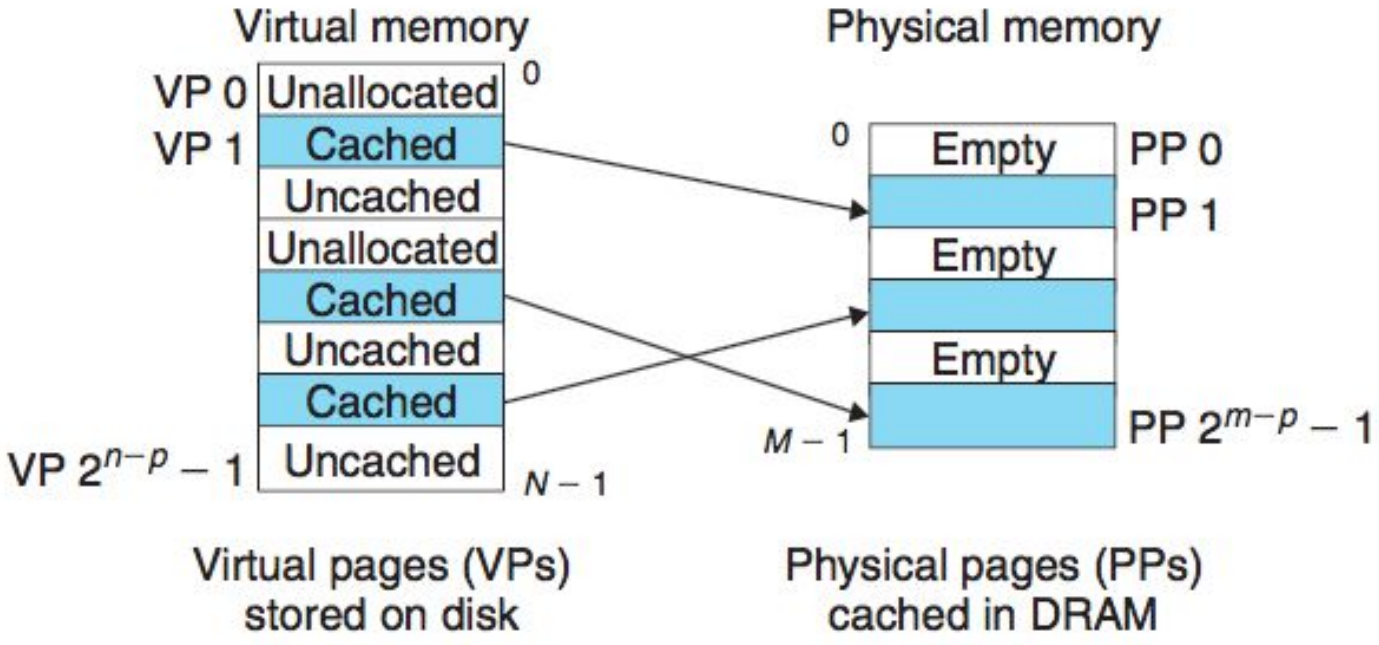
Physical Addressing



Virtual Addressing

Virtual Memory: Basics

- Memory (both virtual and physical) is split up into a number of “pages”.
- Each page is just a contiguous chunk of memory of a set size.
- When a process accesses memory, it uses a virtual address. If the physical memory contains the “page” that the virtual address belongs to, get the value from memory.
- If not (**page fault**), you'll have to find the page on disk, then copy it to physical memory.


Virtual memory
Physical memory
VP 0
VP 1
VP $2^{n-p}-1$
Unallocated
Cached
Uncached
Unallocated
Cached
Uncached
Cached
Uncached
0
$N-1$
0
$M-1$
Empty
Empty
Empty
PP 0
PP 1
PP $2^{m-p}-1$
Virtual pages (VPs)
stored on disk
Physical pages (PPs)
cached in DRAM

Virtual Memory: Basics

- Wait... so each process expects ~4 GB of memory which is stored on disk instead. That means if I have 100 processes, I need ~400 GB of disk space, just to store the memory spaces of all of the programs?

Virtual Memory: Basics

- Not all pages are actually “allocated”. This means they don't exist anywhere. This allows processes to only take up as much space as they need.
- For example, if pages are 4KB and the program has not defined any use for memory address range 0x01000000 to 0x01001000, then there's no need to have that page allocated in memory.

Virtual Memory: Basics

- If physical memory is a “cache” for the virtual memory on disk, where in physical memory is each physical page assigned?
- IE: Consider the page that spans virtual addresses:

  0x0000 → 0x1000
- Where in physical address should that belong to?

Virtual Memory: Basics

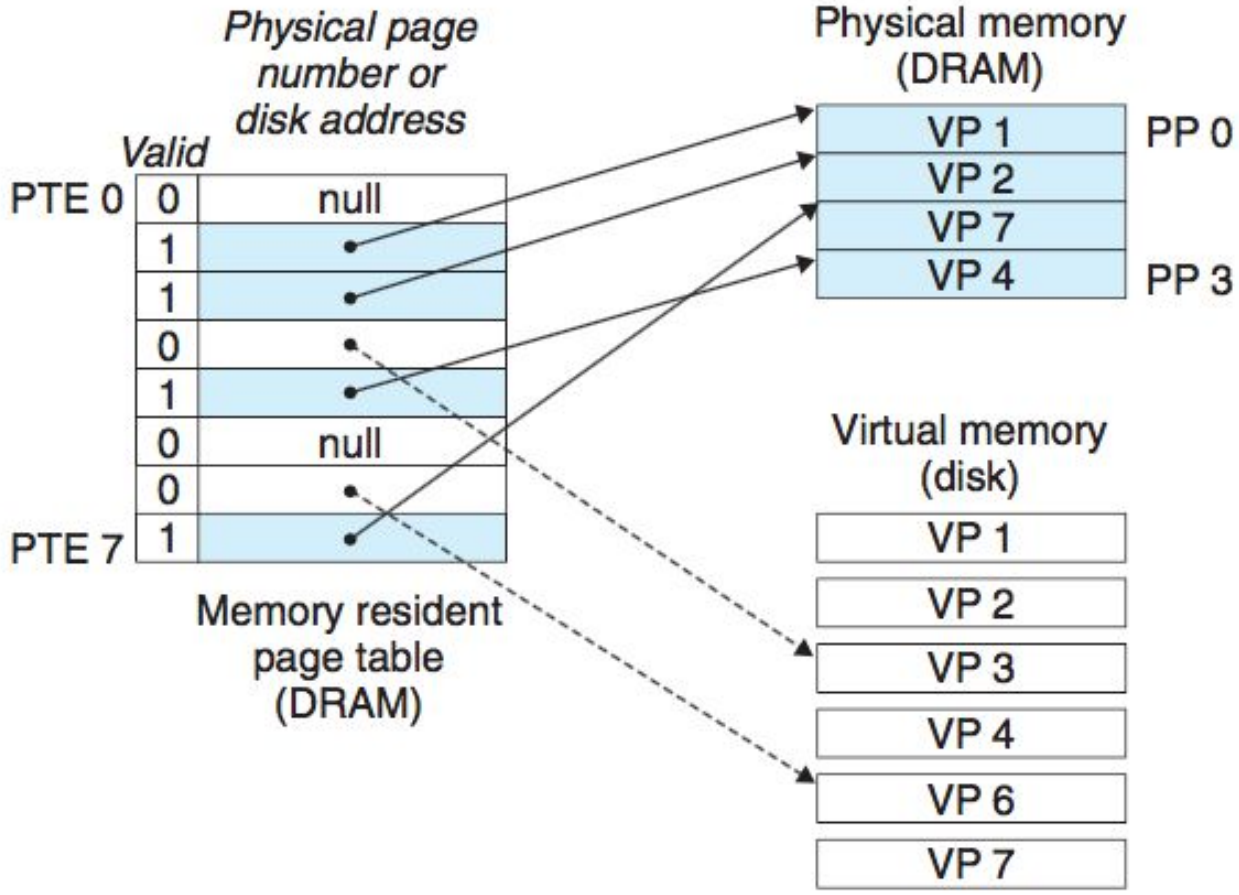
- Physical memory is like a “fully set associative” cache. This means that any page that belongs to virtual memory can be located anywhere in physical memory.
- As a result, a real view of physical memory would show that it's a big mess, interleaved with scattered pages from various processes.

Virtual Memory: Basics

- Like a cache, if main memory is full and you need to bring a different page from disk into main memory, some page is going to have to be a victim.
- Keep this in mind for later.
- If the virtual page can be anywhere in physical memory, how do we find it?
- It would be obnoxious to have to search every page in physical memory as we would with a set associative cache.

Virtual Memory: Page Table

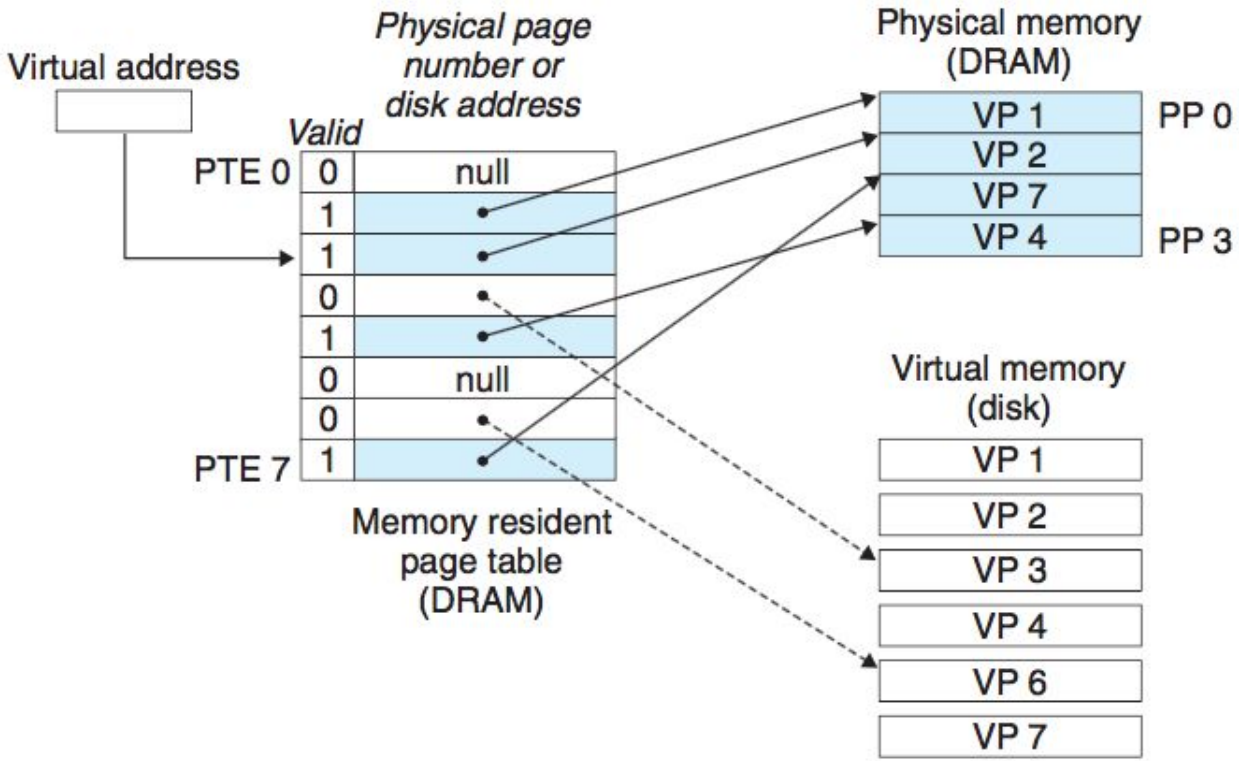
- When we need to look through main memory for a virtual page, we don't want to have to manually search every page.
- Instead, we maintain a Page Table which maps virtual addresses to physical addresses.
- We will use (part of) the virtual address to index into the table. Each entry in the page table will contain (part of) the physical address that it maps to.


Physical page number or disk address
Valid
PTE 0
PTE 7
0
1
1
0
1
0
0
1
null
null
Memory resident page table (DRAM)
Physical memory (DRAM)
VP 1
VP 2
VP 7
VP 4
PP 0
PP 3
Virtual memory (disk)
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

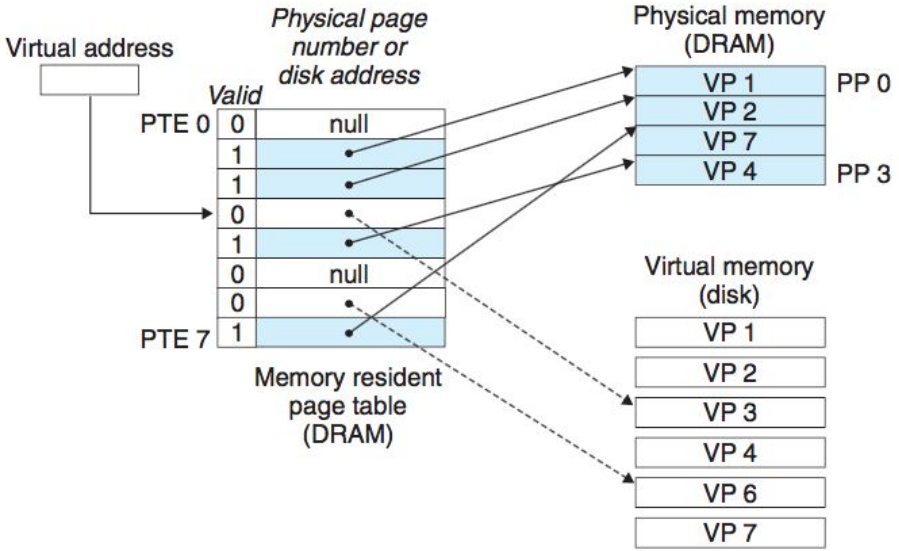
Virtual Memory: Page Table

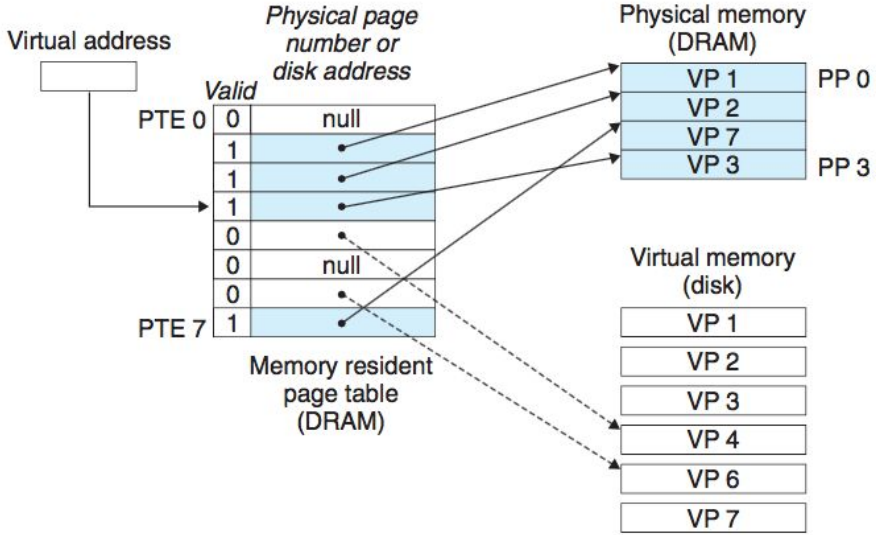
- Consider a page size of ~4 KB or 2^12 bytes. If the addressable space is 32 bits. How many pages could there possibly be in the virtual address space?
- 2^{32} bytes * 1/2^{12} pages/byte = $2^{32}/2^{12} = 2^{20}$ pages

Page Hit


Virtual address
Physical page number or disk address
Valid
PTE 0
0
null
1
1
0
1
0
null
0
PTE 7
1
Memory resident page table (DRAM)
Physical memory (DRAM)
VP 1
VP 2
VP 7
VP 4
PP 0
PP 3
Virtual memory (disk)
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Page Fault


Virtual address
Physical page number or disk address
Valid
PTE 0
PTE 7
0
1
1
0
1
0
0
1
null
null
Memory resident page table (DRAM)
Physical memory (DRAM)
VP 1
VP 2
VP 7
VP 4
PP 0
PP 3
Virtual memory (disk)
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7


Virtual address
Physical page number or disk address
Valid
PTE 0
PTE 7
0
1
1
1
0
0
0
1
null
null
Memory resident page table (DRAM)
Physical memory (DRAM)
VP 1
VP 2
VP 7
VP 3
PP 0
PP 3
Virtual memory (disk)
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Virtual Memory: Page Table

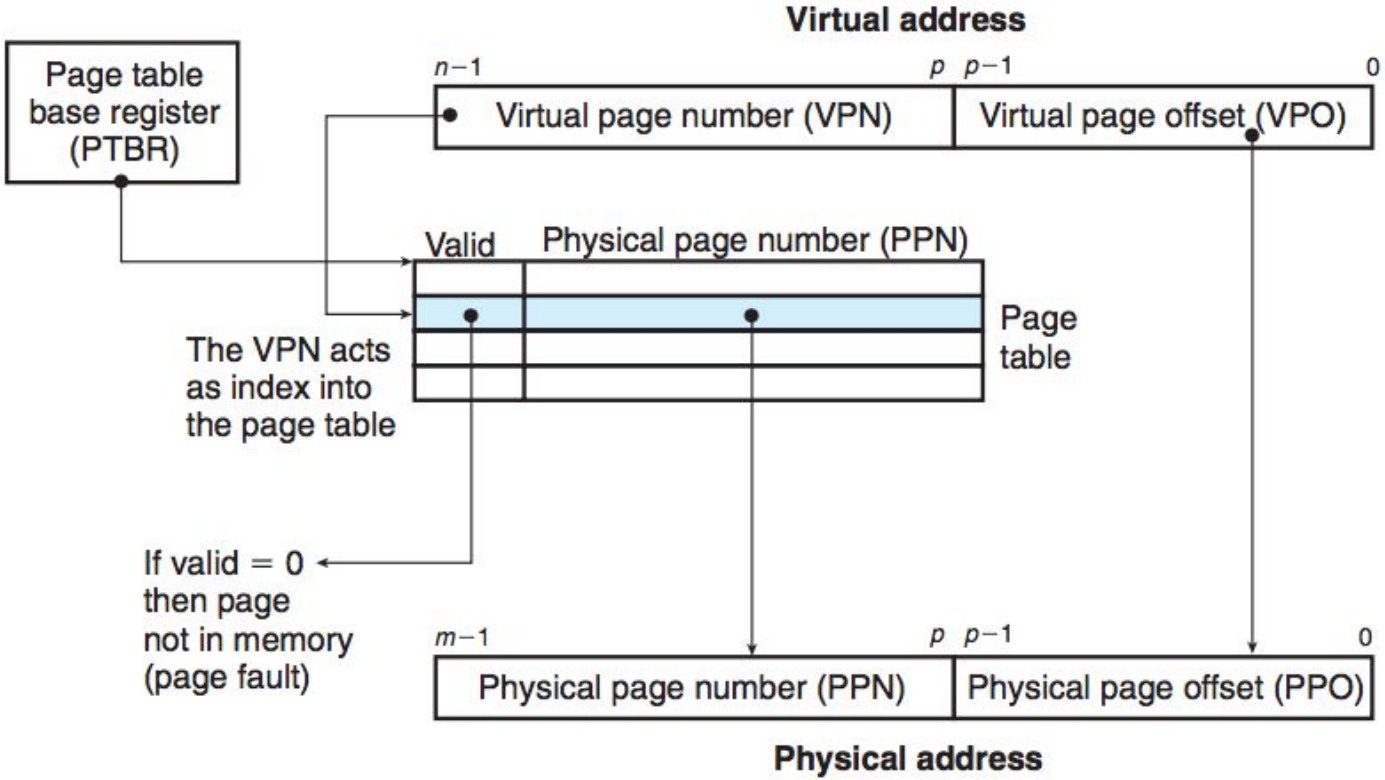
- Virtual Address decomposition.
- [VPN][VPO]
- VPN: Virtual Page Number
  - The index into the page table
- VPO: Virtual Page Offset
  - The byte offset into the page
- Index into the page table with the VPN. The value stored in the page table is the PPN or physical page number.

Virtual Memory: Page Table

- Virtual Address decomposition.
- [VPN][VPO]
- Physical Address decomposition.
- [PPN][PPO]

- The VPO and PPO are same since page size are consistent in both virtual and physical address space.

Virtual Memory: Page Table



Virtual Memory: Page Table

- Ex: Consider a 14-bit virtual address space and a 12-bit physical address space.
- Each page is 256 bytes.
- How is the Virtual Address split in order to index into the Page Table?

Virtual Memory: Page Table

- Ex.
- How is the Virtual Address split in order to index into the Page Table?
- Each page has 256 = 2^8 bytes. This means we need 8 bits to represent the page offset (VPO and PPO).
- This means the VPN is 6-bits and the PPN is 4-bits.

Virtual Memory: Page Table

- Ex.
- Suppose that you're given the following virtual address:

- 011001 00101000
- How many entries in the page table?

Virtual Memory: Page Table

- Ex.
- Suppose that you're given the following virtual address:

  011001 00101000
- There are 2^6 different pages in the memory space (2^6 = 2^14/ 2^8)
- This particular address will access the page table with index 25 (011001).

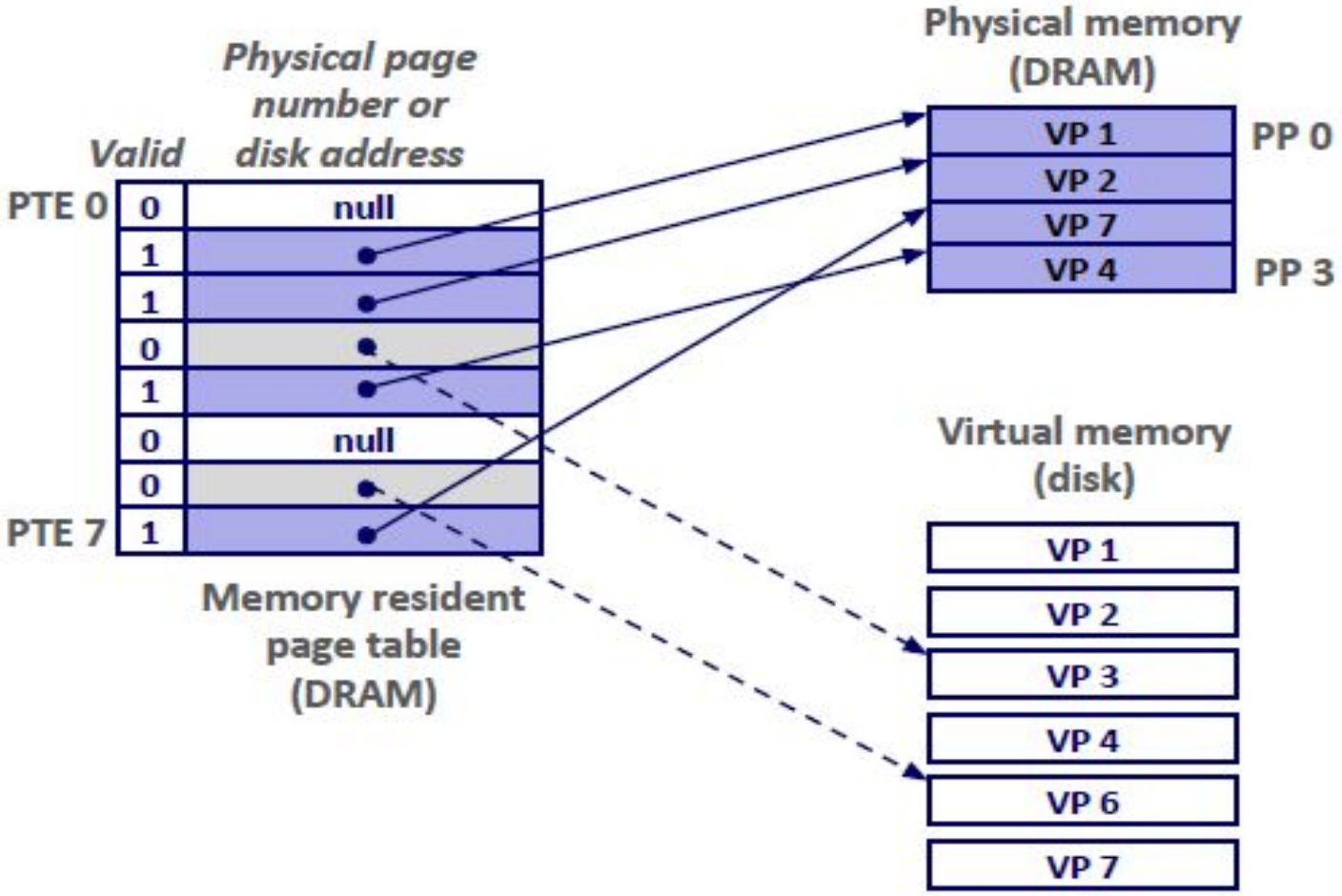
Virtual Memory: Page Table

- Ex.
- Say the entry at the page table entry 25 has the following as the PPN:
  - 0110.
- The reconstructed physical address is:
- [PPN]:[VPO] = 0110 00101000

Virtual Memory: Page Table

- Consider 8 bit virtual addresses with a page size of 2^5 bytes. This mean 5 bits for the virtual page offset and 3 bits for the virtual page number.
- There are 2^3 pages in this addressable space and thus, the there are 8 page table entries indexed from 0 (000) to 7 (111).
- Each page table entry would contain the mapping to the physical address (PPN).

Virtual Memory: Page Table



Virtual Memory: Page Table

Some notes:

- The Page Table is an entity stored in memory, it's not hardware.
- The physical address of the current process' page table is stored in a register (which is backed up in the context switch).
- **Each process has its own page table.**
- A page table entry will either indicate:
  - Virtual page is in memory at location [PPN : VPO]
  - or Virtual page is not in memory (valid bit = 0). Go to disk

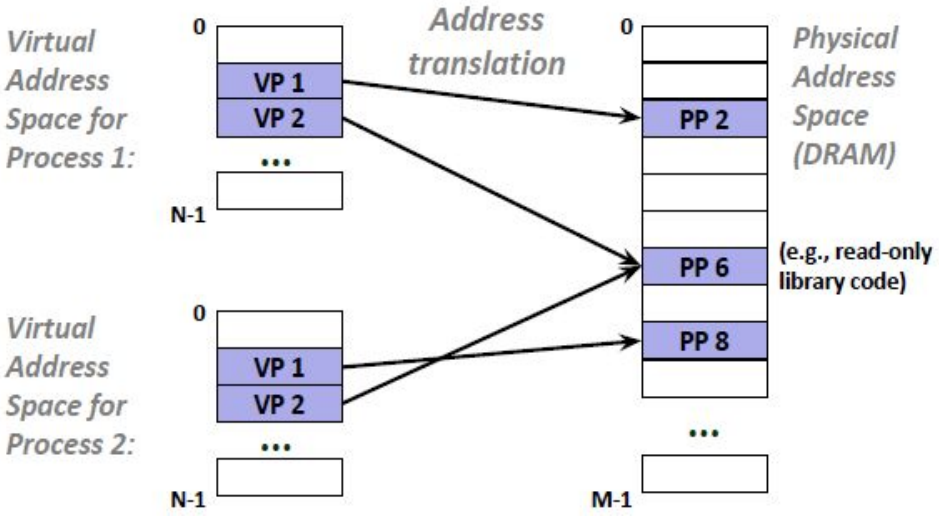
The page is either allocated in disk or unallocated.

Virtual Memory: Page Table

- More notes:
- The page table look-up process inherently provides one of the other great benefits of virtual memory: isolation between processes.
- Processes can't see the memory of any other process by virtue of the fact that all processes have the same virtual address space but different physical addresses.
- Ex. Process 1 has a variable at address 0x10. For process 2, address 0x10 refers to something completely different. (Remember??)

Virtual Memory: Page Table

- More notes:
- Virtual memory paging, however, also allows for simple explicit memory sharing.
- If you want two processes to share some value of memory, have their page tables point to the same physical address.



Virtual Memory: Page Table

- More notes:
- If a virtual page isn't in main memory, when the page is pulled in from disk, the page table of the process is updated.
- However, consider the case when we need to evict a page. The OS can choose to throw out any page belonging to any process.
- This means that evicting a page potentially means updating multiple page tables, whoever had a reference to that page.

Virtual Memory: Page Table

- The fact that the page table is stored in memory should be a red flag.
- For every memory access, we actually need to make two memory accesses?!
  - One to access page table to find where the memory is actually located.
  - One to actually access memory.

Well actually...

Virtual Memory: Page Table

- Because the page table is simply a data structure in memory, it can be cached just like anything else, sharing the instruction or data cache.
- How does this entire process look?

Virtual Memory: Page Table

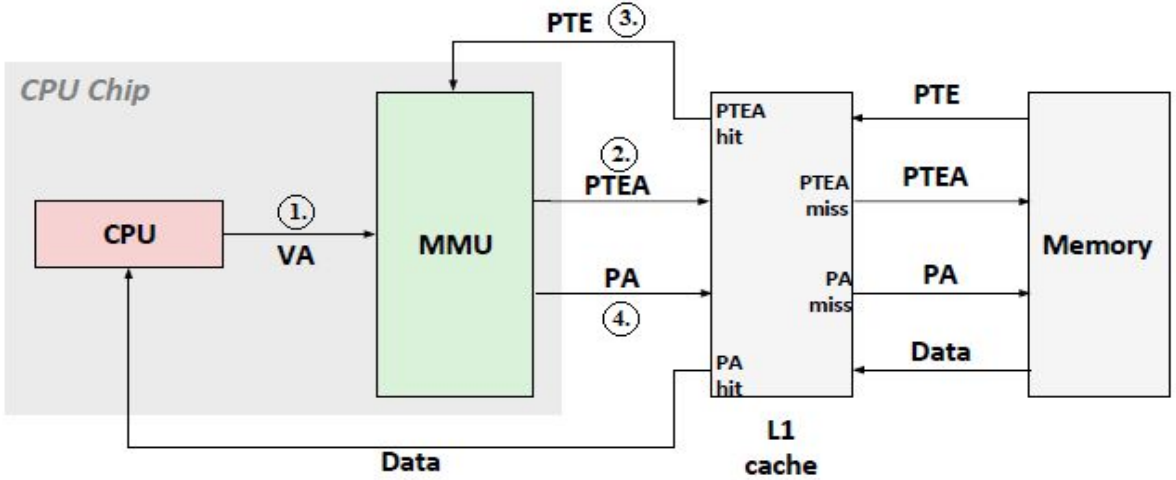
- An overview:
  - The CPU wants to access memory. It issues a memory address to the Memory Management Unit (MMU). This address is a *virtual address*.
  - The MMU takes the virtual address and using the address of the page table, attempts to find the Page Table Entry. This means issuing a request to memory.
  - If the MMU receives a valid PTE, then the Physical Address is extracted and the Physical Address is forwarded to the cache + memory etc.
  - If the MMU receives an invalid PTE, it means that the data that corresponds to the virtual address is not in RAM. An exception is thrown to get the data from disk into RAM.

Virtual Memory

- Index of abbreviations:
  - VA: Virtual Address
  - VPN: Virtual Page Number (ie the index into the page table)
  - PTE: Page Table Entry (the actual data within a page table entry)
  - PA: Physical Address
  - PTEA: Page Table Entry Address (the address of a particular page table entry)

Virtual Memory: Page Table

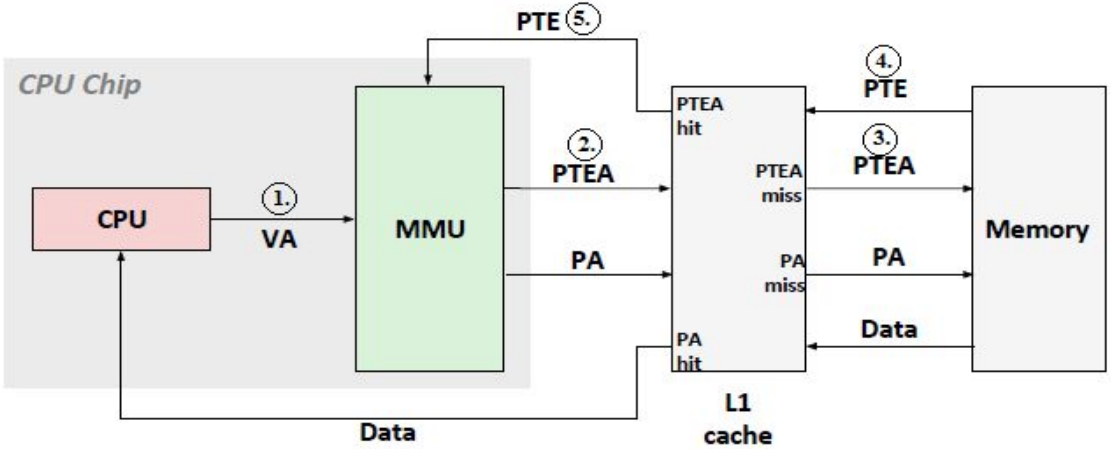
- Page Table Entry in cache



1. CPU issues virtual address to memory management unit.
2. MMU issues address of relevant page table entry to cache.
3. L1 cache return the page table entry to MMU.
4. MMU constructs physical address and sends it to cache

Virtual Memory: Page Table

- Page Table Entry **not** in cache



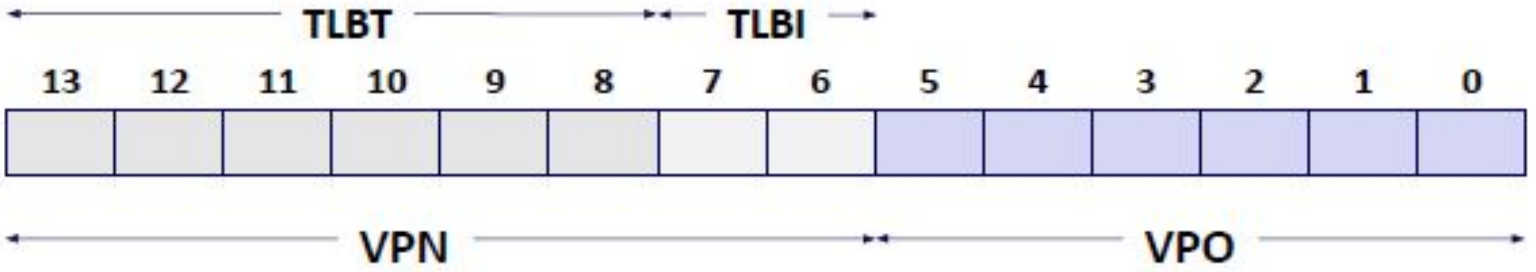
1. CPU issues virtual address to memory management unit.
2. MMU issues address of relevant page table entry to cache.
3. Page table entry not in cache. Cache passes page table entry address to memory.
4. Memory sends page table entry to cache. PTE is cached in... cache.
5. Cache sends page table entry to MMU... and so on.

Virtual Memory: TLB

- Despite being able to cache the page table, we really want to ensure that we don't have to go to RAM in order to get the page table entry.
- As a result, we would like a specialized cache that will only hold page table entries.
- Enter... the translation lookaside buffer.

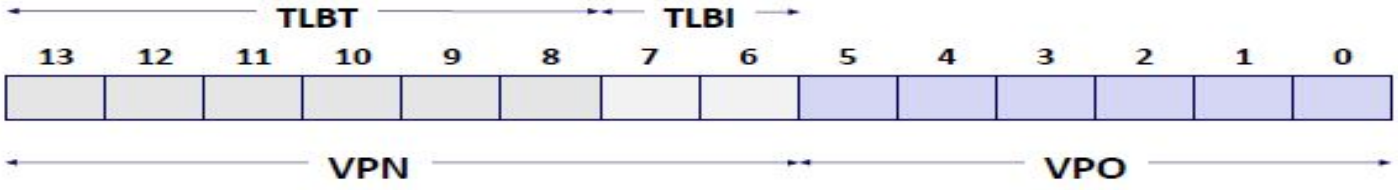
  Instead of having to go to the cache and then memory to get the page table, let's have a special cache for just the page table.

Virtual Memory: TLB



- Accessing the TLB is the same as accessing a cache; the (virtual) address is split into three components Tag (TLBT), Index (TLBI), and Offset (VPO).
- Instead of memory data, the TLB simply contains the page table entry corresponding to that virtual address.

Virtual Memory: TLB

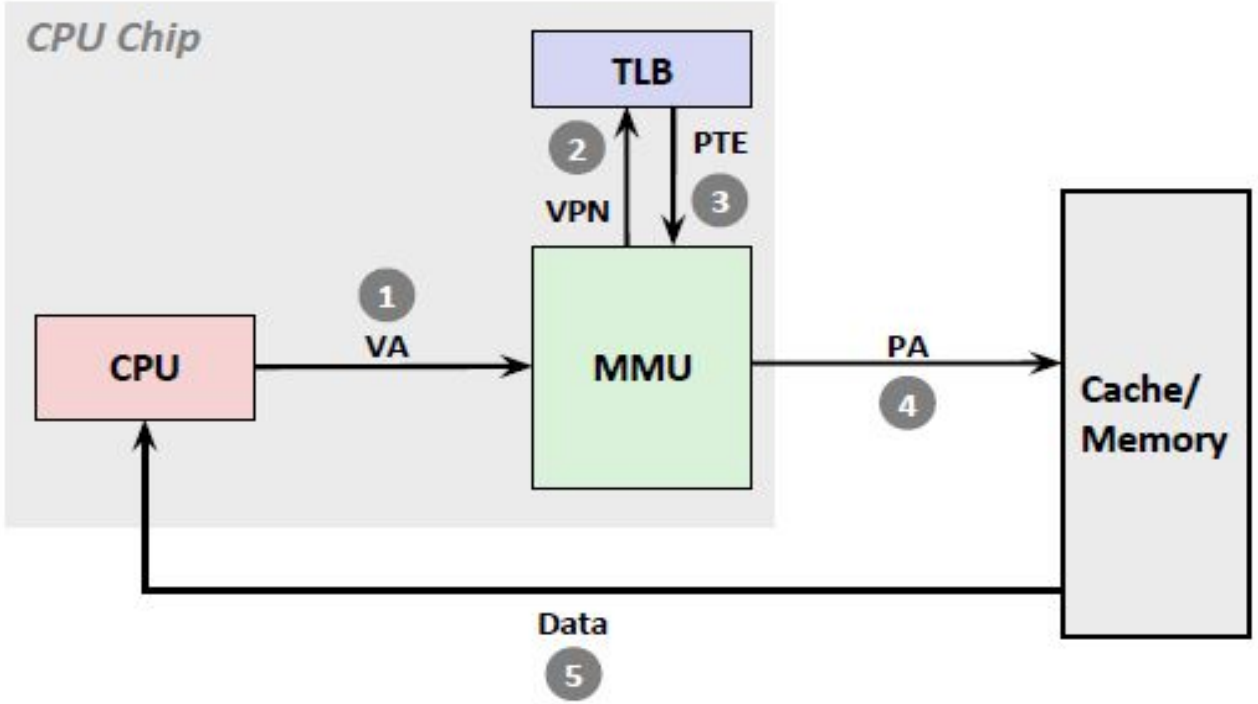


Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

- Each entry contains the tag, a valid bit, and a PPN.
- At a high level, how does this look?

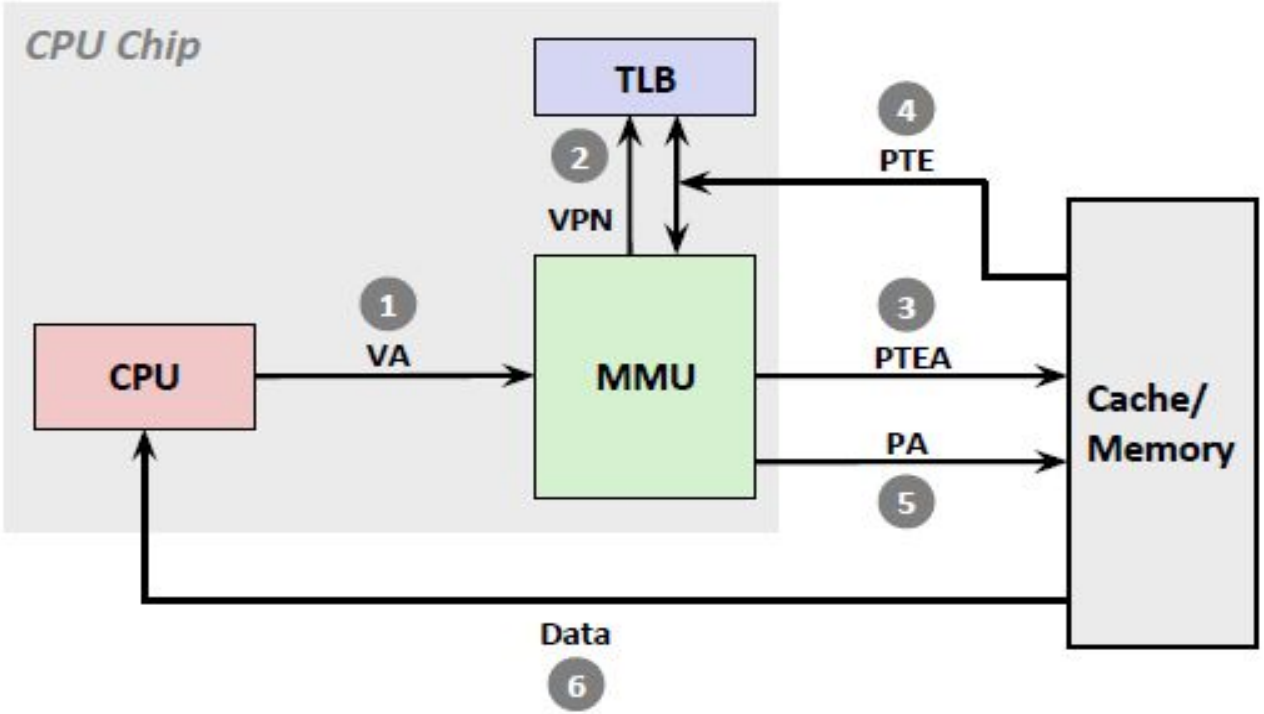
Virtual Memory: TLB

- TLB Hit (the page table is in the TLB)



Virtual Memory: TLB

- TLB Miss (the page table is not in the TLB)



Hierarchical Page Tables

- The page tables take up a pretty hefty chunk of memory. Additionally, the page table pretty much needs to be in
- memory at all times.
- However, the entire page table corresponds to the entire virtual addressable space. Most programs don't use up that much of the addressable space. As a result, most programs don't need to use most of the Page Table.

Hierarchical Page Tables

- The solution is to use a table structure that allows for only part of the table to be allocated at any given time.
- In the original case, the virtual address was split up as follows:

  [VPN][VPO]
- Now we split it up like this:

  [VPN1][VPN2][VPO]

Hierarchical Page Tables

[VPN1][VPN2][VPO]

This system requires a two-level page table hierarchy.

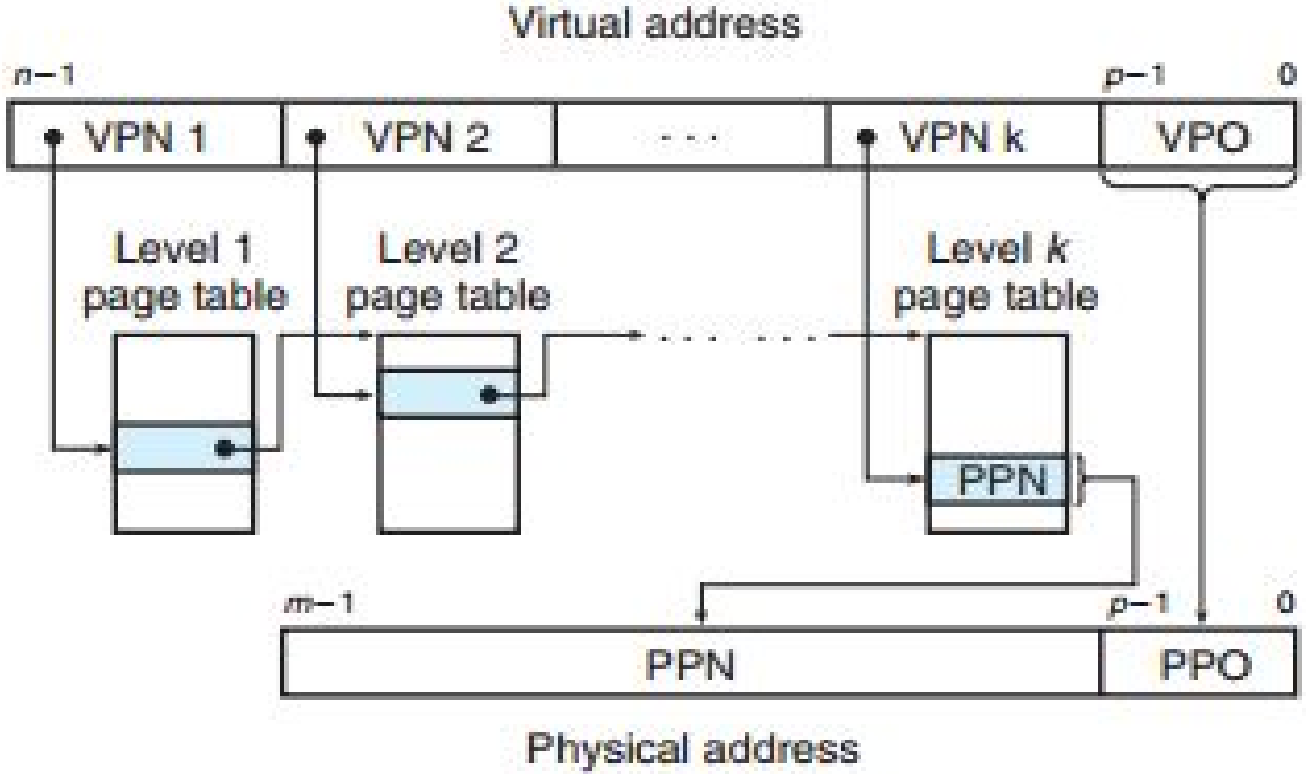
As with the traditional case, there is a register that points to the page table except now this register points to the first level page table.

As before, VPN1 is an index into the Level 1 Page Table.

Hierarchical Page Tables

- Now however, instead of the L1 Page Table containing the physical address that corresponds to virtual address, it contains a
- physical address to an L2 Page Table. VPN2 is used to index into the L2 Page Table.
- Now, the entry in the L2 Page Table will have the PPN.

Hierarchical Page Tables



Hierarchical Page Tables

Consider the following:

32-bit physical addresses

10-bit virtual addresses (yes, it's a weird example)

[3-bit VPN1][2-bit VPN2][5-bit VPO]

- L1 Page Table located at address 0x10. We are issued the address:
- 010 01 00100
- We consult the L1 Page Table.
- How many entries are in the L1 Page Table?

Hierarchical Page Tables

- Consider this example of an L1 page table.
- Because we index into the L1 page table with a 3-bit value, there are 2^3 = 8 entries.
  Each entry is 4 bytes long because they contain an address to another page table in physical memory.
  VPN1 = 010 = 2
- This means we're accessing the entry at address:

0x10 + 2 * 4

(base address of L1) (VPN1) (size of each table entry)

Addr	Entry
0x10	0x1230
0x14	0x1240
0x18	0x1250
0x1c	0x1260
0x20	0x1270
0x24	0x1280
0x28	0x1290
0x2c	0x12a0

Hierarchical Page Tables

- Thus, the L2 page table is at 0x1250. Now we use the VPN2. Because we index into the L2 page table with 2 bits, there are $2^2 = 4$ entries.
- VPN2 = 01 = 1
- L2 Page Table:

Addr	Entry
0x1250	-
0x1254	0xAB
0x1258	-
0x125c	0xCD

- The corresponding PPN is 0xAB

Hierarchical Page Tables

- How does this help the problem of page tables taking too much memory?
- In the non-hierarchical case, the 5-bit VPN would result in a 2^5 = 32 entry table.
- In this example, the L1 page table had addresses for each of its 8 entries, but perhaps only entry 2 is valid.

- This means, we only need to allocate space for one L2 page table to correspond to entry 2.

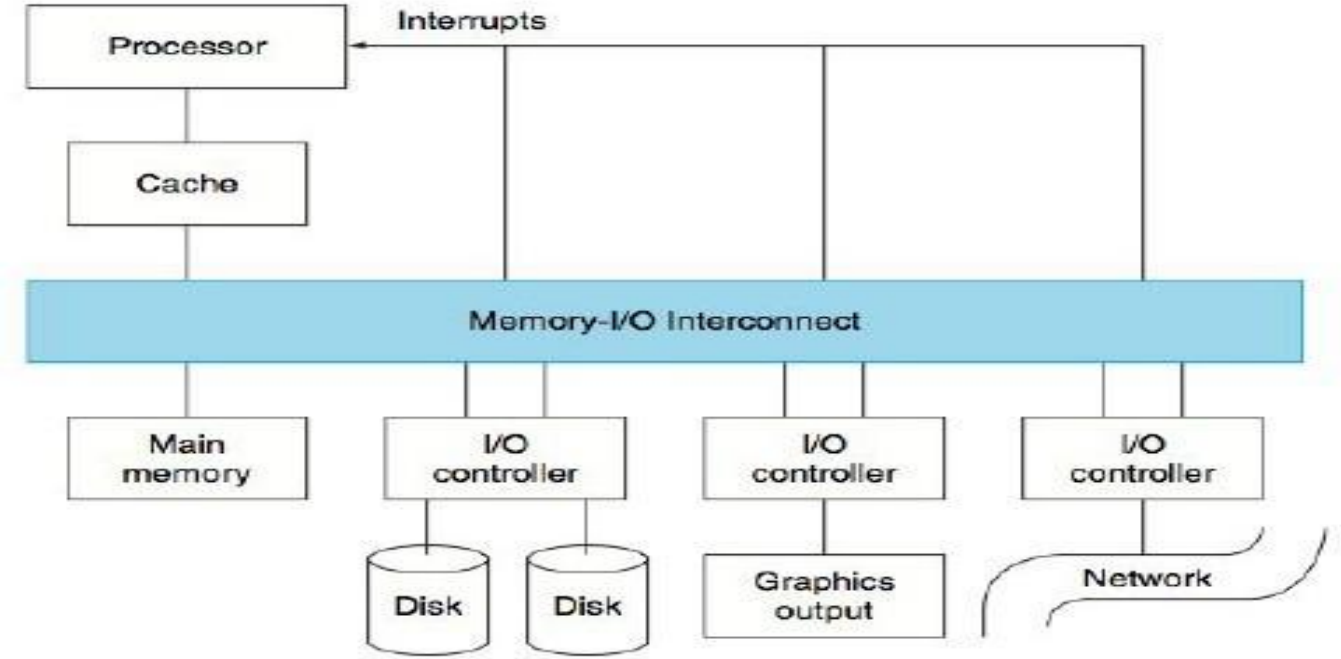
Hierarchical Page Tables

- This means in order to store this hierarchy, we only need 2^3 entries (1 L1 page table) + $1*2^2$ (1 L2 page table) = 12 entries total, instead 32.
- Note however, if every page table was allocated, we'd need $2^3 + 8*2^2 = 40$ entries.
- This only helps if we're not using up ALL of our page table entries... which is most of the time.
- Also note, the entries of the L1 Page Table will contain different data from the L2 Page table, which may mean different entry sizes.

I/O

The more hardware-ish viewpoint

- A hypothetical logical interconnection:



← Bus

I/O

- Things to note:
- Despite the variety of components (CPU, memory, devices), most things are linked together via a “bus”.
- A bus is essentially a collection of wires that can be used to transfer data.
- In this example the processor is not connected to the cache via the bus. Generally, we want at least the L1 cache to be as close to the processor as possible.

I/O

- Things to note:
- Devices are not directly connected to the bus. They are connected via an I/O controller.
- What are the trade-offs of having everything connected via a single bus?

I/O

- What are the trade-offs of having everything connected via a single bus?
- Pros:
  - Simple from a hardware standpoint. Requires few components/little space.
  - Transparency between devices. Each device can snoop on the bus and know exactly what's going on. This is a very nice property if you have multiple processors sharing memory via a bus.
- Cons?

I/O

- What are the trade-offs of having everything connected via a single bus?
- Cons:
  - You could potentially have many devices competing for a single resource. The bus becomes a huge bottleneck.
  - Bus arbitration logic. If a device wants to use a bus, it needs to fight the other devices for the right to use the bus. There is only one winner.

I/O

- Things to note:
- The wires of the bus fall into several categories and serve different purposes (data, address, control).
- Through the bus, each of the controllers can communicate with the other controllers (depending on the complexity of the controller)

I/O

- Computer architecture and interconnection is not standardized because different goals require different interconnections.
- There are, however, common features that you should be aware of so that you will be able to understand more practical/realistic arrangements.

Communication with I/O Devices

- At the processor, how are commands given to I/O devices?
- Memory Mapped I/O: Portions of the address space are assigned to represent an I/O device.
  - movl (0x1234), %eax
  - The address and operation are sent over the bus. The memory system recognizes that the address corresponds to an I/O device and doesn't respond.

Special I/O assembly instructions.

Communication with I/O Devices

- Now that the processor has issued a request to a device, how does the I/O device respond back to the processor?
  - Polling
  - Interrupt driven I/O Direct
  - Memory Access

Communication with I/O Devices

- Polling
  - When the I/O device has a result ready for the CPU, it indicates it or puts the result in a register.
  - When the CPU expects a result, it must manually go and check the registers to see if it has a delivery. If there is, the CPU goes and gets the result.
  - If the CPU must wait before continuing, it can issue a request and do nothing other than continually poll the register until a result is received.
  - What about if a CPU isn't explicitly waiting for something (like a mouse/keyboard action)?

Communication with I/O Devices

- Polling
  - What about if a CPU isn't explicitly waiting for something (like a mouse/keyboard action)?
  - The CPU will have to spend a lot of time checking the status register to make sure there isn't some result that it has to handle.
  - This is a waste of time.

Communication with I/O Devices

- Interrupt Driven:
  - When an I/O device has a result. It will modify a status and cause register and inform the CPU by explicitly sending an interrupt.

    Once the currently running instruction completes, the CPU will handle the exception and read the registers if necessary to decide how to handle the I/O.

Communication with I/O Devices

- Both interrupt-driven and polling simply inform the CPU of when it can fetch data and the CPU is ultimately responsible for getting the data.
- If the CPU is spending time fetching data from I/O devices, it's not running the program that it's supposed to be running. Program execution is halted.
- As a result, these methods work fine for small data transfers but don't work very well for large and slow data transfers (like reading from disk).

Communication with I/O Devices

- When the processor reads to disk, the processor will be copying from disk to RAM.
- What a bore.
- If only the CPU could spend it's time running the program and the “disk controller” could access memory... directly.

Communication with I/O Devices

- Direct Memory Access (DMA)
- Turns the I/O controllers into small CPUs, each of which can access the bus and do work such as transferring data.
    1. Processor issues request informing the DMA controller what to do.
    2. Processor goes back to doing it's job. Meanwhile the DMA controller arbitrates for the bus and attempts to do the data transfer to memory.
    3. When the DMA is done, it informs the CPU via an interrupt.

Communication with I/O Devices

- In this way, the processor can do the haughty processor work without having to sully itself with the petty matters of the proletariat (I/O devices).

What this means for you

- When it comes to applications, how can they make use of this I/O structure?

What this means for you

- Initiate the low level interactions by simply calling syscalls:
- open()
- read() write()
- close()
-

References

- Slides modified from DJ Kim, UT Wang and Shikhar Malhotra
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Thank You