50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Virtual Memory

Natalie Agus (Fall 2018)

1 Memory Contents

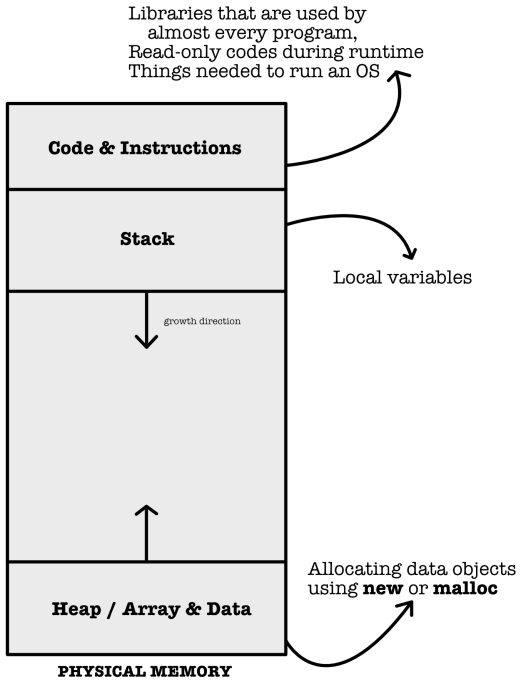

Figure 1

Our physical memory (RAM) contains things shown in Figure 1. Stack and heap can grow during runtime (stack mainly due to recursion and local variables). A physical memory alone definitely is not enough to hold all of our computer data. We need to store the majority of our data in the disk

2 Disk

Disk is cheap, and it gives a big address space, and is **non volatile**, meaning that it retains stored data when unplugged from the power source. Eventually, we will combine DRAM (cache), memory (SRAM), and disk to give the illusion of a computer running at a cache speed with disk-size address space. In other words, we can think of the RAM as the ”cache” to disk.

3 Pages

In a disk, the address space is huge (there's too many memory addresses), so we typically divide them into **pages**,

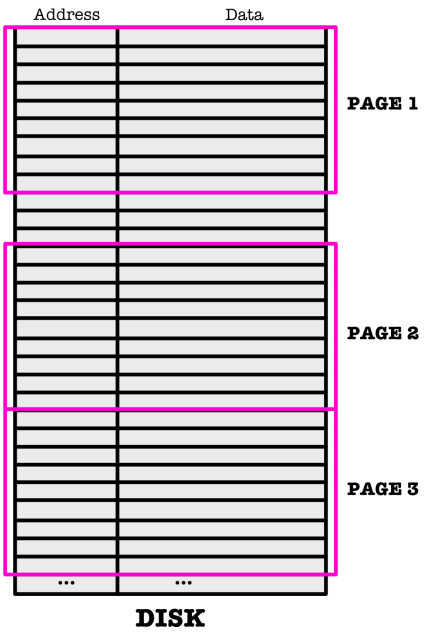

Figure 2

Page: collection of contiguous memory addresses. Good for locality of reference.
So whenever we fetch data from the disk to the RAM, we fetch them by pages. When we want to look for a particular memory location, we now look for their **page number (PPN)** first (i.e: the higher m bits of their physical address), and then their **page offset (PO)** (i.e.: the lower p bits of their physical address).

4 Virtual Memory

Before you can understand the rest of this notes, you need to know that programs (codes) are only brought into RAM when it is running. The majority of your pro-

grams that are not running stays on disk.

A computer can run many programs at a time, and you know that there's so many programs that can share a single RAM. The RAM is also filled with other things, such as as data, kernel space, stack, libraries, etc. For convenience sake, programs "*do not necessarily know*" the existence of another programs and everything else that lives in the RAM. Therefore they don't have to keep track of which addresses in RAM is full or free to use. Only the Operating System (OS) knows the physical location of each and every running program and installed program in the computer. This provides some layer of abstraction, as the OS is the only program that needs to care about memory management and the rest of the programs in the computer can run as per normal. So for example, each program's PCs can start from address 0x0000 onwards within their own program. This way we can say that each program has their own "**virtual memory**". Virtual memory allows the system to give every process its own memory space isolated from other processes.

However since we only have one RAM, in that RAM space, each program's (virtual) PC is mapped to different physical addresses, depending on where the instructions for each program codes are actually stored in real life. This mapping is done by the **memory management unit**. Below are the definitions of virtual address and physical address:

Virtual Address (VA): addresses generated by programs as it runs in the CPU
Physical address (PA): real address wiring in the memory chips

The pagetable contains **all possible combination of virtual address of a program**, but not all VA has corresponding PA at a time in the RAM (it may be in the disk). The mapping between VA to PA is done by the MMU (memory management unit):

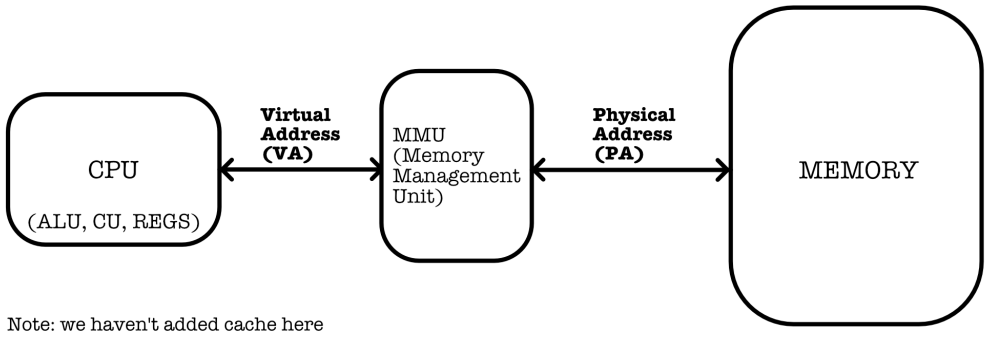

Figure 3

5 Disk Swap Space

As you open more and more program and let it run, we can eventually *run out of RAM space*, due to growing stack, or even running codes itself. We need some unused area on disk **called the swap space** to serve as an extension of the RAM. In other words, **virtual memory spans from physical addresses in the RAM and disk addresses in the swap space**. It uses a portion of the empty space on your disk to temporarily hold the data that would otherwise be held in your RAM. Programs do *not* know that their actual RAM is divided in two different places, within the program itself, it may be running in contiguous virtual addresses.

6 Demand Paging

This section explains how the system in Figure 3 works. This entire procedure is called *demand paging*, which is what computers do when it asks for a certain data / instruction from the Memory.

Demand paging main idea:

1. In demand paging, the main idea is that **at first** (when the program has just started) **the entire virtual memory is placed on the swap space of the disk**
2. Pages will only be brought up to RAM if the program code asks for it.

More detailed steps:

1. Everything is initially on disk (when a computer just started up), the disk contains all your installed programs (executables [1]), data, etc.
2. So initially the RAM is empty and we have a *nicely organized* [2] *array of pages on disk*
3. When we click a program executable open to begin its execution, **the OS prepares the entire virtual memory space for this program on disk**, in particular, on disk's swap space, so all of its VA corresponds to address on disk. Here in the VM, all instructions necessary for this program to run, stack space, and data space are prepared nicely by the OS [3]. .

[1] Executables are useful for the OS to set up the context for the program (provide it memory, load its data, set up a stack for it), the OS needs to read a program's executable file and needs to know a few things about the program such as the data which the program expects to use, size of that data, the initial values stored in that data region, the list of opcodes that make up the program (also called the text region of a process), their size etc.

[2] Organized means we (the OS) **know how to get them**: the disk address of everything in the computer through the OS file directory or the necessary information of a program contained within its executable file.

[3] Note that a program code and data **is not entirely loaded** to VM when the program is started. Only a small subset, essentially its entry point (elf table [4], main function, initial stack) is loaded, and everything else is loaded later. This type of virtual memory management by the OS, i.e: how to access data on disk or allocate VM space is out of the scope of this subject. We assume that the OS is capable getting necessary data from disk for running programs.

4. After you opened the program, the first line of code of the program can now be executed. This will result in **page-fault** exception because all its virtual addresses aren't resident (means in RAM) yet

5. The OS will then handle this "missing" page and start getting the necessary pages from the disk swap space and put it into the RAM. Loading the instructions of this program to the RAM means that this turns some virtual addresses into *resident*, hence updating the pagetable.

    - Know that each program has its own virtual memory (its like giving the illusion that each program has independent RAM all for itself)
    - So for example, VA of each program can start from 0x00000000 onwards but it actually points to different physical addresses on disk.

6. Many page faults will occur as the program begins its execution **until most of the working set of pages are in physical memory** (not the entire program, as some programs can be way larger than RAM, e.g: your video games)

7. In other words, OS bring only necessary pages that are going to be executed into RAM as program runs

8. This eventually fills up the RAM, and we update the pagetable accordingly since now some VPN corresponds to a PPN (resident).

9. If a new page P is asked but RAM runs out of space, we need to "swap" some old (LRU/FIFO, depends on the replacement algorithm) pages in RAM to disk,

10. In particular, we place these 'old' pages in RAM to a special section of disk area called the 'swap area'. This free up some space in the RAM, so that we can write the new page P in from disk to RAM

11. This disk swap area serves as an extension to RAM, for running programs only

12. Step 4-14 continues each time more and more programs are opened and run

13. Finally when you close a program, the OS saves all your data back on disk (not the disk swap space, but the determined storage area of that program), and free up your virtual memory (both on RAM and disk swap space)

7 Page-Map Design

The MMU maps the higher v bits of VA (called the VPN) to a corresponding PPN. The PPN is the m higher order bits of the PA[5]. **1 entry in the pagetable is needed for every virtual page, so the total number of entries in the pagetable is** 2^v. There are two extra bits D, and R in the MMU (pagetable / tlb):

1. **R: Residence Bit**:

[5] due to locality of reference

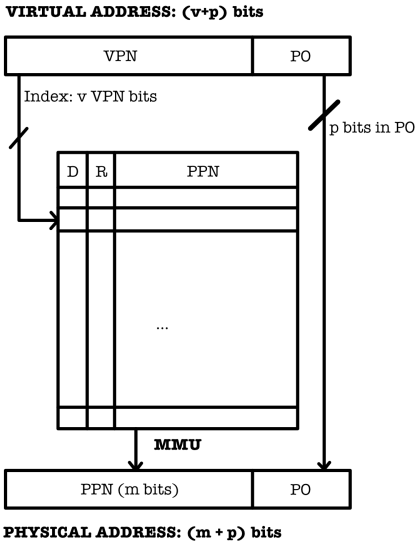

Figure 4

(a) If $R == 1$, then the data is in the RAM

(b) If $R == 0$, then it is a **page-fault** exception (no PA in RAM, because data not in RAM). Either get it from Disk or it hasn't been allocated at all.

2. **D: Dirty Bit**: if $D == 1$, then the data has to be written to the Disk before it is removed from the RAM Depending on the replacement strategy, the TLB and pagetable can contain the LRU column too if LRU replacement strategy is used. The number of LRU bits needed per item is b bits, since the number of entries in the pagetable is 2^b.

8 The Virtual Memory Summary

Each program has to refer to the MMU to find out the corresponding PA for some of their VA. The VA address space is much larger than the address space of PA in RAM [6]. Therefore if a page-fault is met, one has to (1) find a page to replace from the RAM, (2) write it to disk if it is dirty, (3) fetch the requested page from disk, (4) write it to RAM, and (5) update the MMU. With this in place, the program had the "illusion" that it has a huge address space with average speed of a RAM. This is called **having a virtual memory**.

8.1 Pseudocode of swapping pages

```
Given VPN and PO,
        if(R[VPN] == 0),
                PageFault(VPN) /* Go to page fault handler */
```

[6] Whenever we say PA, it refers to the address in RAM, and not Disk

```
        else return (PPN[VPN] << p) | PO /* Concatenate PPN with PO */

If page-fault, /* this is done in software level, by OS */
         /* Make space in RAM */
         find LRU page i
         if dirty,
                write to disk, then set R[i] = 0
         otherwise, set R[i] = 0

         /* Fetch data from disk */
         Then PPN[VPN] = PPN[i] /* set the PPN[VPN] to be
                                   the allocated cell in RAM */
         ReadPage(DiscAddr[VPN], PPN[i]) /* get from disk and write
                                            it into memory */
         R[VPN] = 1
         D[VPN] = 0
```

9 MMU Details

9.1 Page-Map Arithmetic

The physical representation of how PPN and PO points to a specific address in the page within memory is shown in the figure below.

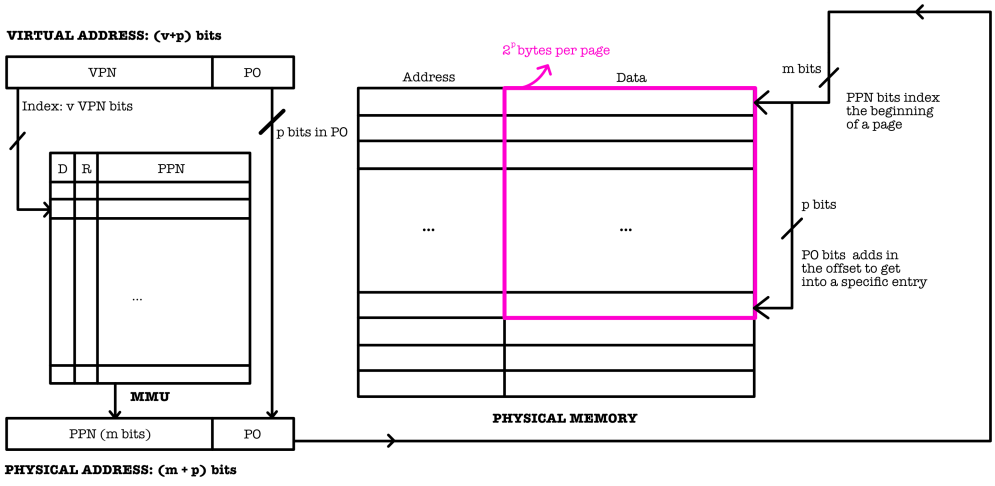

Figure 5

With byte addressing, we have

1. 2^{v+p} bytes of VM
2. 2^{m+p} bytes of actual memory

3. $(m+2)2^v$ bits in MMU Pagetable (2^v rows, each row having $m+2$ bits[7])
4. There are 2^p bytes per page

9.2 MMU Pagetable Location

The MMU Pagetable is actually in the RAM itself, because it has to accommodate every VA and therefore its size is quite large. It is expensive to implement it as SRAM. A portion of the RAM is dedicated to store the MMU Pagetable. The page-table pointer (**PGTBL Pointer**) points to the first entry of the MMU Pagetable section in the RAM.

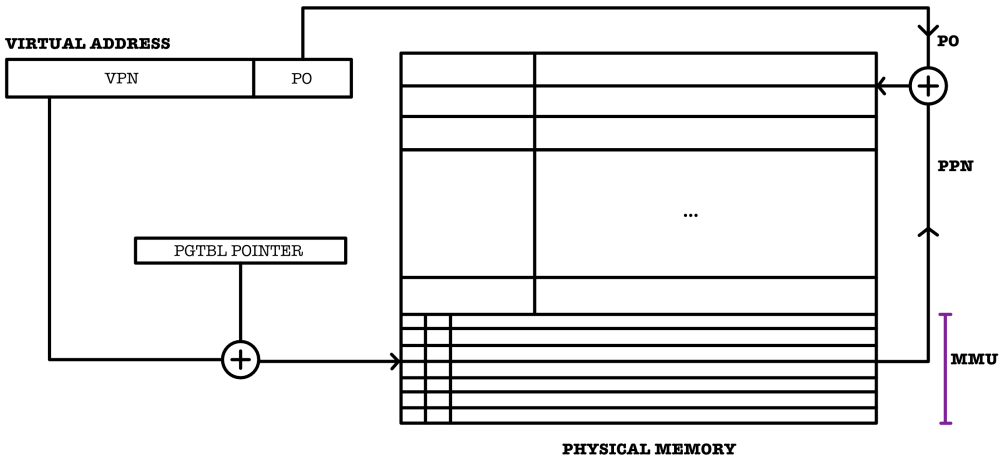

Figure 6

However, this causes us to **access the RAM twice**:

1. Look up Pagetable to get the PA
2. Access the RAM again to get the data with that PA

While it is cheap to put the MMU Pagetable within the RAM, it is too slow. However if we implement the entire MMU Pagetable using SRAM, it is too expensive. The solution is to have a ”cache” for MMU Pagetable, it is called the **Translation Lookaside Buffer** (TLB).

9.3 TLB

The TLB is serves as a cache to the Pagetable. It is a small, FA cache for mapping certain VPN to PPN. There is locality of reference in address memory reference patterns, therefore there is **super locality** in reference to page-map (hit-rate > 99%).

[7]The 2 bits for D and R bits

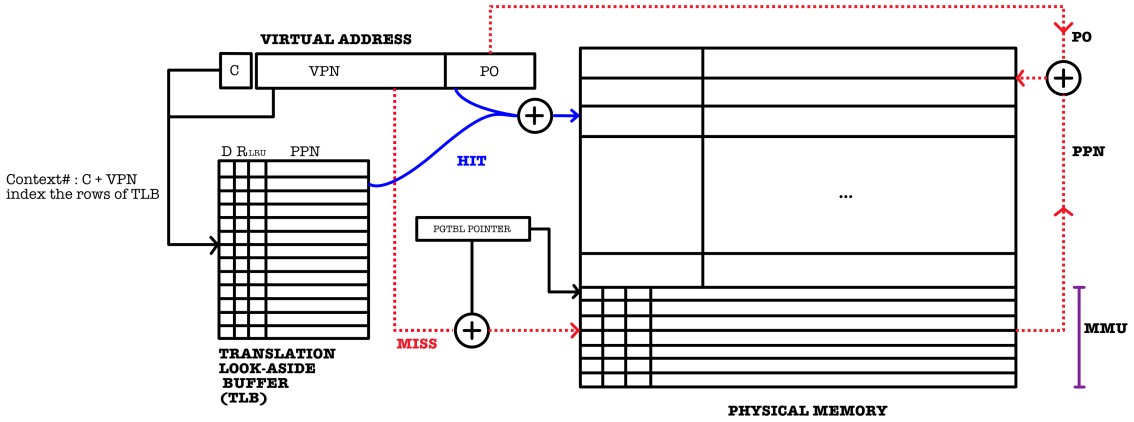

Figure 7

10 Virtual Memory Benefit: Context Switching

In Figure 7, there is an extra field for the VA called C (context). A context is a mapping of VA to PA locations as dictated by the Pagetable. Take a look at the figure below:

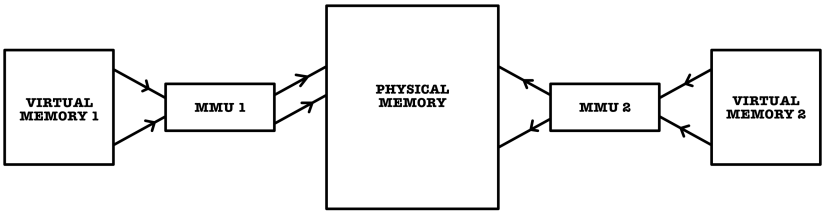

Figure 8

1. Several programs may be loaded to the main memory simultaneously
2. Each program PC starts from 0
3. Each program thinks that they have the entire machine to themselves
4. They can be written as if it has access to all memory without considering other programs
5. The machine does **context switching** to run them ”simultaneously” (i.e: they appear that they run in parallel, but they actually rapidly take turns to run)

10.1 Context Switching

Each program is assigned to a context, with its own Pagetable that translate its specific VA to PA. Context switching between programs is done by adding a register to hold the index of current context, and hence we do not have to ”flush” the TLB whenever we change context. As shown in Figure 7, the context # *C* and VPN together serve as the index of the rows of the TLB.

11 Using Cache with Virtual Memory

So far we have learned about MMU: consisted of PageTable in the RAM and also a ”cache” to it called the TLB. Given a VA, we refer to TLB (or pageTable if its a miss) to get the PA and then access the RAM. We now combine the concept with *cache* that we learned last week. Recall that a cache stores the memory address and the data straight away (not pages), so there is no need to access the RAM. There are two possible options on where to connect the cache, **before** or **after** the MMU:

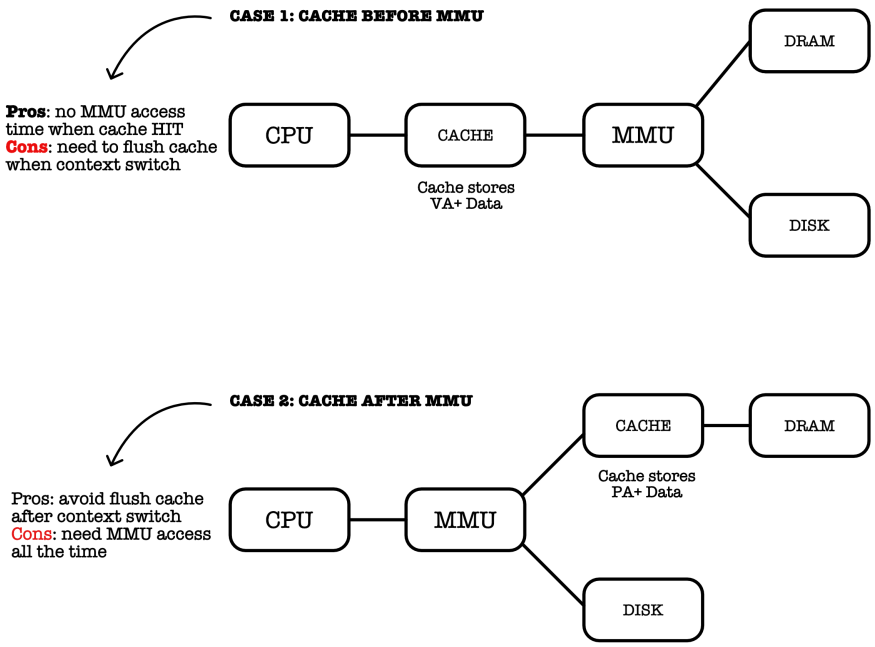

Figure 9

11.1 The best of both worlds

The indexing of cache lines uses PO, while the indexing of the pagetable in MMU uses VPN.

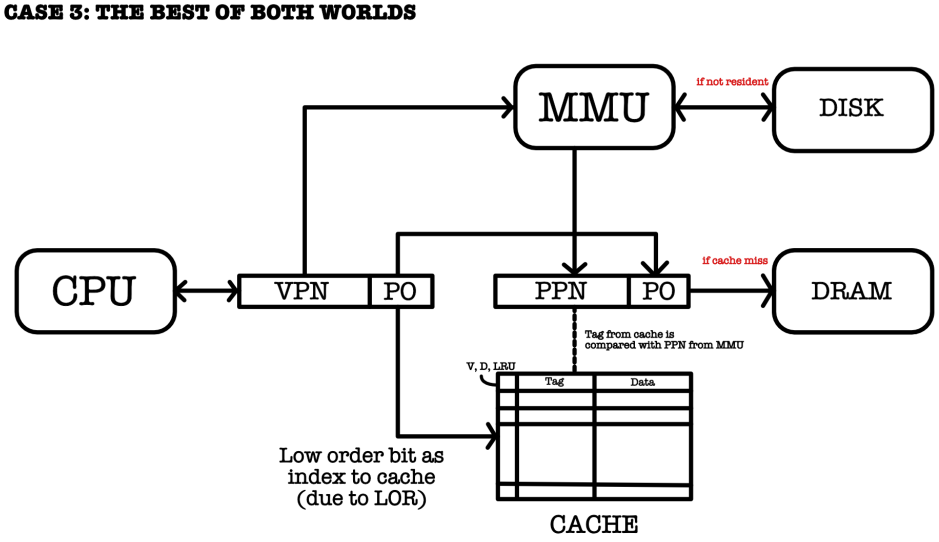

Figure 10

11.1.1 The Cache: DM or NW

Each cache line stores a single data (not pages) and the address of each data. Therefore the index of the tag in the cache is set to be the PO of the VA, due to locality of reference [8]. The content of the tag field in cache is still the PPN. The PA from cache is compared with PPN from MMU + PO. **MMU access and cache access can be done in parallel**.

11.1.2 The MMU

If cache miss, then the machine can immediately fetch the data from either memory or disk, and update the MMU and cache at the same time.

[8] If higher order bit is used then you will end up with address contention for DM cache