Virtual Memory

The model of MIPS Memory that we have been working with is as follows. There is your MIPS program, including various functions and data used by this program, and there are some kernel programs e.g. that may handle exceptions. This model makes sense from the perspective of one person writing one program. However, as you know, many programs may be running on a given computer at any time. For example, when you run your Windows-based laptop, you may have several programs going simultaneously: a web browser, Word, Excel, etc. You may even have multiple copies of each of these programs going. The operating system also has many programs running at any time. (If you are running LINUX, and you type `ps -e`, you will get a list of a large number of programs that are running at the current time.)

A program that is running is called a *process*. When multiple processes are running simultaneously, they must share the computer's memory somehow. There are two issues that immediately hit us in the face. First, since there are many processes, the total size of all processes may be greater than 2^{32} bytes. Second, different processes may (and generally do) have overlapping addresses. (An extreme example is that multiple copies of a program may be running.) Moreever, at any time, various registers (PC, ALU registers `$0-$31`, ..., pipeline registers, etc) all contain values for just one process. This introduces a problem: how can multiple processes exist simultaneously when these processes share the same address space (2^{32} bytes) ?

In order for multiple processes to share an address space and to share the processor, one needs a trick called *virtual memory*. The basic idea is that the program addresses (also called *virtual addresses* are translated into physical memory addresses, in such a way that the physical memory addresses are non-overlapping. This translation is hidden from the user. It is done by the operating system (kernel).

Besides allowing multiple processes to share the processor, virtual memory allow independence between the size and layout of the program address space (2^{32} bits), and the size and layout of *physical address* space. From the point of view of memory hardware, there is nothing magical about 32 bits for (MIPS) program addresses. You know that when you buy a computer, you can choose the amount of RAM (e.g. 2 GB or 8 GB), and you can choose the size of your hard disk (200 GB vs. 1 TB). These choices are just a matter of how much money you want to spend. They are not crucial limitations to the processor you use. This distinction between the size of a program address space (2^{32} in MIPS) and size of physical memory may seem strange at first, but it will make sense gradually in the coming lectures.

Physical Memory

MIPS program memory consists of 2^{32} bytes. How much is that?

- 2^{10} is about 1 KB (kilobyte - thousand)
- 2^{20} is about 1 MB (megabyte - million)
- 2^{30} is about 1 GB (gigabyte - billion).
- 2^{40} is about 1 TB (terabyte - trillion).
- 2^{50} is about 1 PB (petabyte - quadrillion)

How does that compare to the size of memories that you read about when buying a personal computer or tablet, or flash drive, etc.

Disk memory

Consider some examples of *disk memories*:

- floppy disk holds about 1.4 MB (not used any more), magnetic
- CD (less than 1 GB), optical
- DVD e.g. 10 GB, optical
- hard disk drive (HDD) on personal computer e.g. 500 Gbytes, magnetic

The access time for disk memories is quite slow. The memory is a mechanical disk which must spin. Data is read (or written) when the appropriate part of the disk physically passes over the read (or write) head. When we talk about a spinning disk, you should realize that this is very slow relative to the time scales of the processor's clock. A typical processor these days has a clock speed of 1 GHz (clock pulses occuring every 10^{-9} seconds). If you want read a word from memory (as a step in the `lw` instruction, for example) and the word you are looking for is on one of the above disks, then you won't be able to do so in one clock cycle. Rather it will typically take millions of clock cycles for the disk to spin around until the word is physically aligned with the read (or write) head. If the processor has to wait this long every time a lw or sw is executed, it will be very inefficient. Another solution is needed – and that's what we'll be talking about in the next several lectures.

Disk memories are typically called "external memory" (even though the hard disk on your desktop or laptop is inside the case of the computer). To avoid these long memory access times in these computers, the computer has an internal memory that is much faster than the hard disk. We'll discuss these below.

Flash memory – SSD (solid state drive)

Another kind of external memory is flash memory. The memory cards that you put in your digital cameras or mobile phones, or the USB drives that you keep in your pocket are made of flash memory.

Many tablet computers such the Apple IPad use flash memory instead of a hard disk (HDD) for their bulk storage. Flash is much faster to access than disk. Typical flash access times are 10^{-4} ms, although these numbers will vary. Flash has the advantage that it has no moving parts, i.e. it is semiconductor based. So you can shake a tablet and it will work fine.

Hard disk memory and flash serve the same purpose. They are *non-volatile* memories. When you turn off the power, they maintain their values. They can therefore be used for storing the operating system software and all your applications.

RAM

The "main memory" inside your computer is called RAM. RAM stands for "Random access memory". There is nothing random about it though, in the sense of probability. Rather, it just means

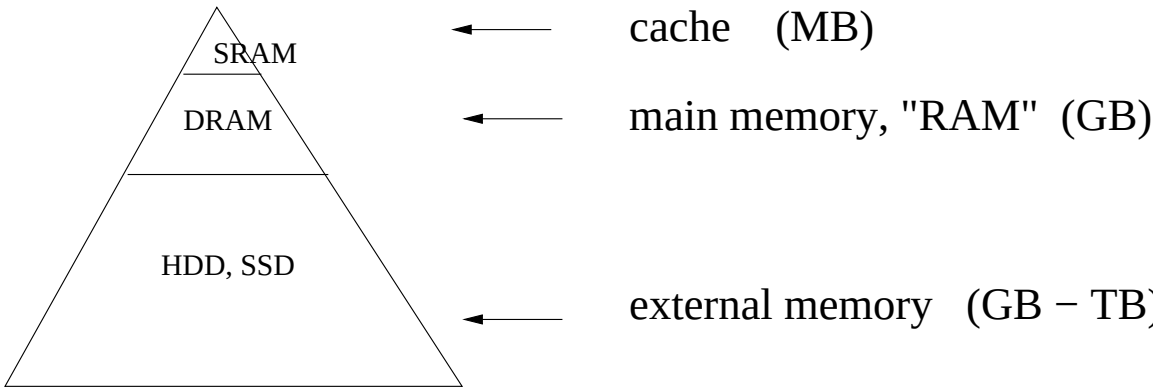
that you can access any byte at any time – with access time independent of the address. Note this uniform access time property distinguishes RAM from disk memory. (Note that flash memory, which is a relatively recent invention, also allows uniform access time.)

Physically, there are two kinds of RAM:[1] these are generally called SRAM (static RAM) and DRAM (dynamic RAM). SRAM is faster but more expensive than DRAM. These two types of RAM serve very different roles in the computer, as we will see in the coming lectures. SRAM is used for the cache, and DRAM is used for "RAM." (The terminology is confusing here, so I'll say it again. When one says "RAM", one means "main memory" and the technology used is DRAM. When one says "cache", one is referring to SRAM.)

Memory Hierarchy

You would like to have all your programs running in the fastest memory (SRAM), so that any item of data or any instructions could be accessed in one clock cycle. (In the data path lectures, we were *assuming* that instruction fetch and (data) Memory access took only one clock cycle. However, because faster memory is more expensive, it is just not feasible to have everything all instructions and data sitting in the fastest memory. As a result, the typical laptop or desktop has a relatively little SRAM (say 1 MB), much more DRAM (say a few GB), and much much more hard disk space (hundreds of GB).

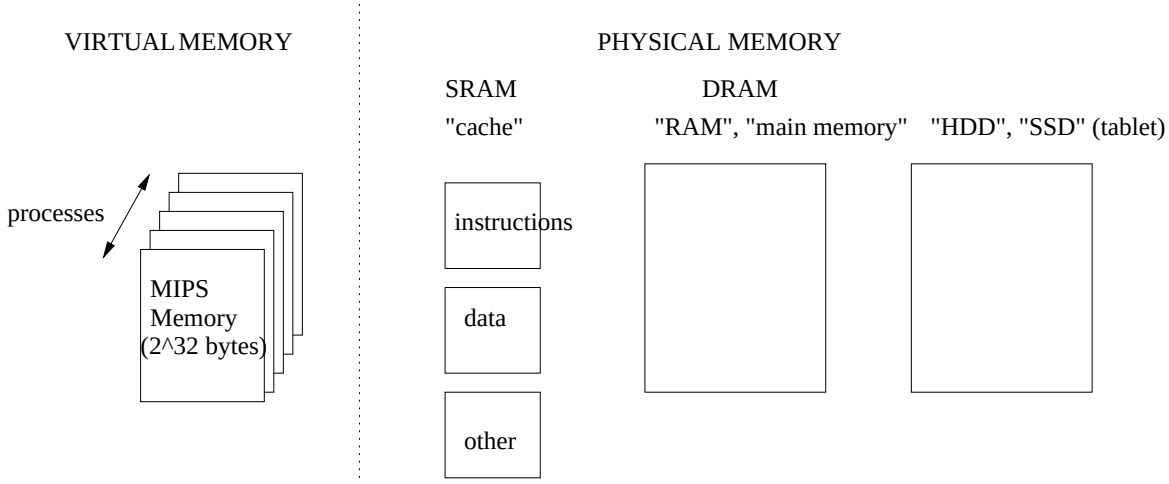
It is common to imagine a "memory hierarchy" as follows. This figure captures two concepts: the first is the order of access: you try to access data and instructions by looking at the top level first, and then if the data or instructions are not there (in SRAM), you go to the next level (main memory, DRAM) and if what you are looking for is not there either, then you proceed down to the next level (HDD or SSD). The second concept is that the sizes of the memories grows with each level.



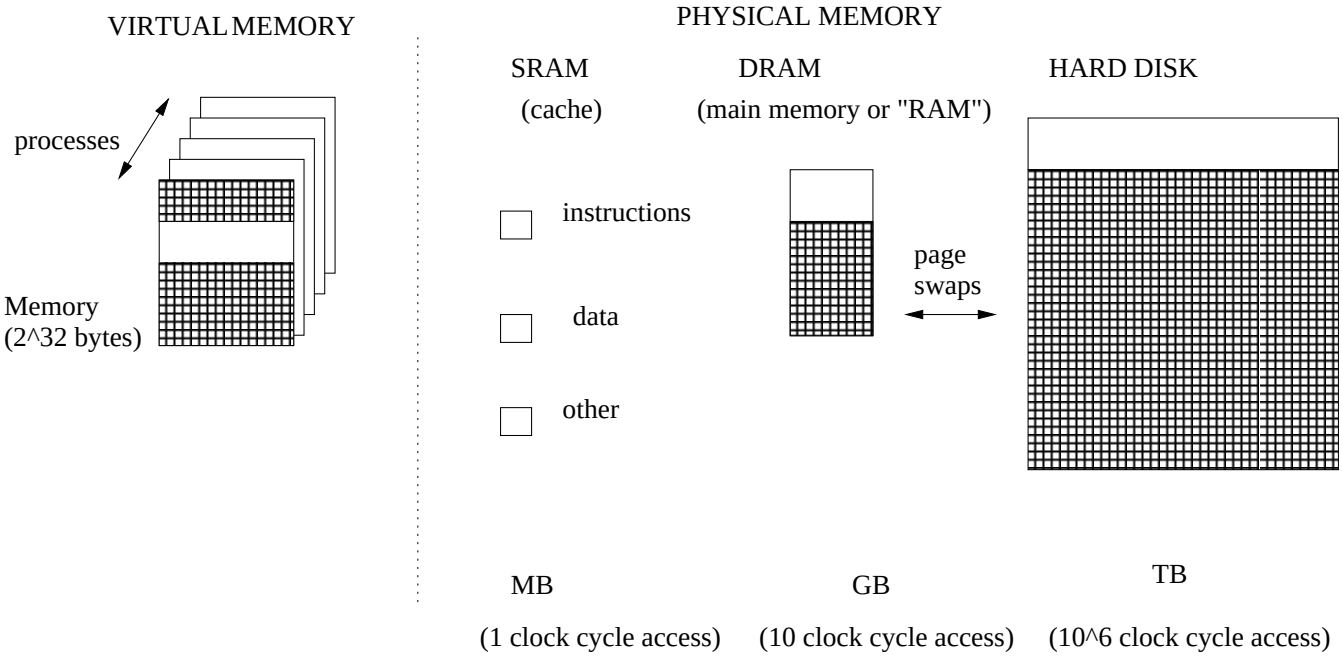
[1]The underlying technologies are *not* our concern in this course. You can just pretend both SRAM and DRAM are made out of transistors and other electronics and leave it at that. In fact, there are many subtypes of both SRAM and DRAM and these are changing remarkably fast. If you wish to learn the details, you'll need to read up on your own.

Address translation: virtual to physical (page tables)

In the rest of this lecture, we will begin to relate two notions of memory. The first is the programmer/software notion: each program assumes a Memory with 2^{32} bytes. The second is the hardware notion: memory must be physically embodied in something.



In order for programs to run on the actual processor using real memory, program addresses (also called *virtual addresses*) must be translated into physical addresses. The figure below contains the main elements of what we will discuss.



How can we map virtual addresses to physical addresses? We do this by partitioning virtual address space and the physical address space into equal sized chunks, called *pages*. For example,

each page might be 4 KB. (This is an arbitrary number.) The address of a specific byte *within a page* would then be 12 bits. We say that this 12 bit address is the *page offset* of that byte.

For example, a 32 bit MIPS (virtual) address would be split into two pieces. The lower order piece (bits 0-11) is the page offset. The high order piece (bits 12-31) is a virtual page number. Each virtual page number of a process corresponds to some physical page number in physical memory. We'll say this page is either in RAM or on the hard disk (though more generally it could also be on a CD, etc.)

The computer needs to translate (or map) a virtual page number to a physical page number. This translation (or mapping) is stored in a *page table*. Each process has its own page table.

Here's how it works for our 4 KB page example. Take the high order 20 bits (bits 12-31) which we are calling the virtual page number, and use these as an index (address) into the page table. Each entry of the page table holds a *physical page number*. We are calling it a table, but to make it concrete you can think of it as an array, whose index variable is the virtual page number.

Each page table entry also contains a *valid bit*. If the valid bit is 1, then the address stored in that page table entry is a physical page number in main memory (RAM). Suppose main memory consists of 1 GB (30 bit addresses). Then there are $2^{18} = 2^{30}/2^{12}$ pages in main memory. The address of a particular byte in main memory would thus be 18 high order bits for the physical page number, concatenated with the 12 bit offset from the virtual address. It is the upper 18 bits of address that would be stored in the "physical page address" field of the page table.

If the valid bit is 0, then the physical address has a page number that is on the hard disk. There are more pages on the hard disk than in main memory. For example, if the hard disk has 1 TB (2^{40}) then it has 2^{28} pages (since each page is 2^{12} bytes).

Each entry in the page table may have bits that specify other administrative information about that page and the process which the kernel may need. e.g. whether the page is read or write protected (user A vs. user B vs. kernel), how long its been since that page was in main memory (assuming valid is 1, when the page was last accessed, etc).

virtual page number (up to 2^{20} addresses/entries)	physical page address (depends on size of phys. mem.)	valid bit	R/W	etc
0				
1				
2				
3				
⋮				

The page table of each process doesn't need to have 2^{20} entries, since a process may only use a small amount of memory. The kernel typically represents the page table entries using a more clever data[2] structure to keep the page tables as small as possible.

[**Modified March 23**]
There are different ways to organize the page table in main memory. one way is to store it in a part of main memory that is reserved for the kernel. In this scheme, the part of memory storing page tables is not itself partioned into pages. Why do this? The purpose of paging is to give us flexibilty in where data and instructions go, namely either in main memory or on the hard disk. But page

[2]for example, hash tables which you may have learned about in COMP 250

tables tables must be accessed frequently – we don't want them every to be on the hard disk! If we keep them in main memory, then they can be accessed quickly. And if they are kept in main memory, then there is less of a need to use paging. By keeping the each page table in a non-paged region of main memory, we can use a *fixed* translation from virtual (kernel) address where page tables are stored to the physical kernel address where page tables are stored. **In lecture 19, I will present an alternative way of organizing page tables, which used paged memory.**

Page faults and swapping

When a program tries to access a word (either an instruction or data), but the valid bit of the entry of that page in the page table is 0, then the page is on the hard disk. Here we say that a *page fault* occurs. When a page fault occurs, a kernel program arranges that the desired page be copied from the hard disk into main memory. Often this requires first removing some page from main memory (to make space). This copying is called a *page swap*.

The kernel program that handles page faults is an example of an *exception handler*. When a user program is running and page fault occurs, the program branches to the kernel. It may branch directly to the address of the *page fault handler* (a kernel program that handles page faults). Or it may first branch to a general exception handler which then analyzes what the exception is and then branches to the particular handler for that exception. Either way, it is ultimately the page fault handler that updates the page tables and administers the copy of data from main memory to and from hard disk. (More details will come in upcoming lectures.)

[ASIDE: as processes are created and terminated, so are their page tables. Each process has a process ID, and so the kernel needs also to have a Process ID table, which keeps track of the address in memory of the page table for that process.]

Finally, when you think of 'where the page tables are', you can think either of a virtual addresses (program addresses) or physical addresses. The type of address space that you think and talk about depends on what you are trying to say.

Next lecture: the cache

If you understood the descriptions above, then you will realize that each virtual Memory access (whether an instruction fetch, or a data access e.g. `lw` or a `sw` in MIPS) requires two physical memory accesses. The first accesses the page table so that the virtual page number can be translated into a physical page number. The second accesses the word itself using its physical address, either in RAM or on disk. Even if the address is indeed in main memory (valid bit of page is 1), we still have a big problem. Each main memory (DRAM) access requires many clock cycles (say 10). Thus, every instruction would seem to require many clock cycles - cycles to get the translation and cycles to fetch/load or store the word. This can't be the way things are done (and its not).

The solution to the problem is to use a small very fast memory (SRAM) to hold frequently used instructions and data, as well as frequently used page table entries (translations). Check out the last slides of this lecture for a sketch of how this is done. Next lecture, I will go into more details.