Virtual Memory

The model of MIPS Memory that we have been working with is as follows. There are your MIPS programs, including main and various functions that are called by main and by each other, and data used by this program, and there are some kernel functions 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. Two related issues arise. First, different processes use the same 32 bit program address space. (An extreme example is that multiple copies of a program may be running!) Second, at any time, various registers (PC, registers `$0-$31`, ..., pipeline registers, etc) all contain values for just one process. We will deal with the second issue in a few weeks. For now, let's consider how processes share the same Memory address space.

We first need to distinguish 32 bit program addresses from physical addresses. The 32 bit program address space is called *virtual memory* and the program addresses themselves are called *virtual addresses*. These addresses must be translated into physical memory addresses. This translation is hidden from the MIPS programmer. As we will see, it is done by the a combination of hardware and software, namely the operating system i.e. kernel.

Besides allowing multiple processes to share the processor, virtual memory allows independence between the size 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.

An analogy for this idea of virtual versus physical addresses is telephone numbers. My office telephone is 514-398-3740. Do you think that the person in the office next to me should have telephone number 514-398-3741 ? Of course not. Or do you think that phone numbers 514-399-xxxx should be close to McGill? No, of course not. You know that telephone numbers are arbitary, i.e. there is some arbitrary mapping between phone numbers and physical places. The same is true for the relation between program addresses and physical addresses.

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, and the technology they use:

- 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. 1 TB magnetic

Because disk memories have moving parts, their access time 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

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. Tablet computers use flash memory (called a solid state drive SSD) instead of a hard disk (HDD) for their bulk storage and indeed many laptops are using SSD's now too. Flash is much faster to access than disk. Typical flash access times are 10^{-4} ms, although this will vary a lot between technologies and it can be much faster than that.

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

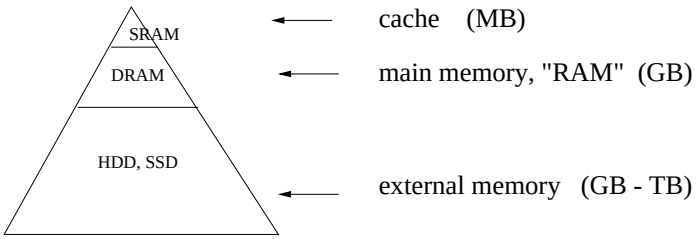
The "main memory" inside your computer is called RAM. RAM stands for "Random access memory". There is nothing random about it though, in the sense of probability. Rather, it just means that you can access any byte at any time – with access time independent of the address. Note this uniform access time property distinguishes RAM from disk memory. (Flash memory also allows uniform access time. However, flash memory is non-volatile, whereas RAM is volatile.)

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 repeat myself. When one says "RAM", one means "main memory" and the technology used is DRAM. When one says "cache", one is referring to SRAM.)
[ASIDE: after the lecture, I was asked by a student how sleeping and hibernation work, with respect to these different types of memory. Putting your computer to sleep keeps the data and program state in volatile memory, and the battery is used to provide enough power to maintain the values in memory. Hibernation is quite different. The program states are stored on disk (non-volatile). You can unplug your computer and let the battery drain out and the states will be saved. When you start the computer again, the program states are read off disk and the programs can continue.]

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. e.g. 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. The typical laptop or desktop has a relatively little SRAM (say 1 MB), much more DRAM (say a few GB), and much much more disk space (hundreds of GB).

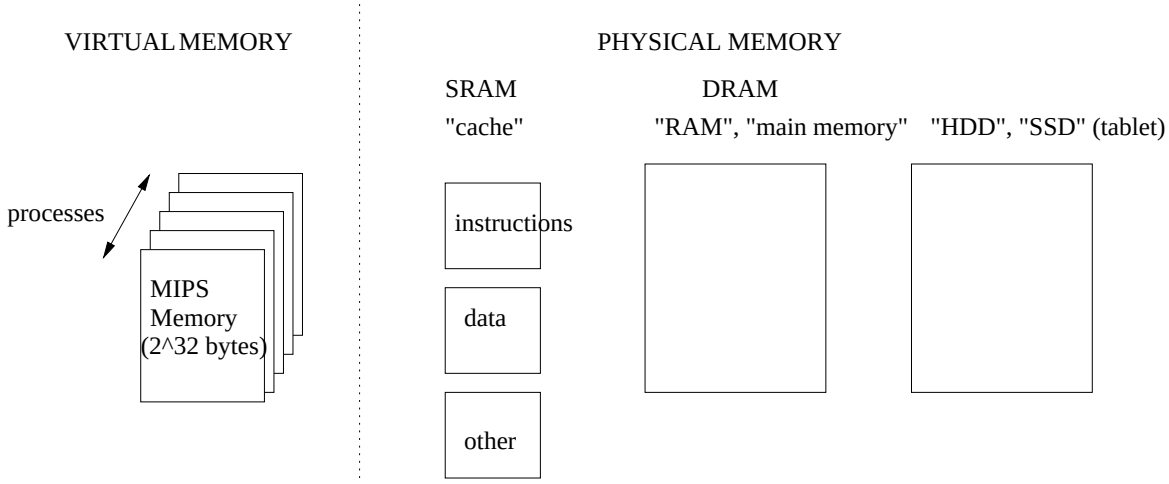
It is common to imagine a "memory hierarchy" as follows. This figure captures two concepts: the first is the order of access: you try to access data and instructions by looking at the top level first, and then if the data or instructions are not there (in SRAM), you go to the next level (main memory, 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, and the cost per bite decreases.



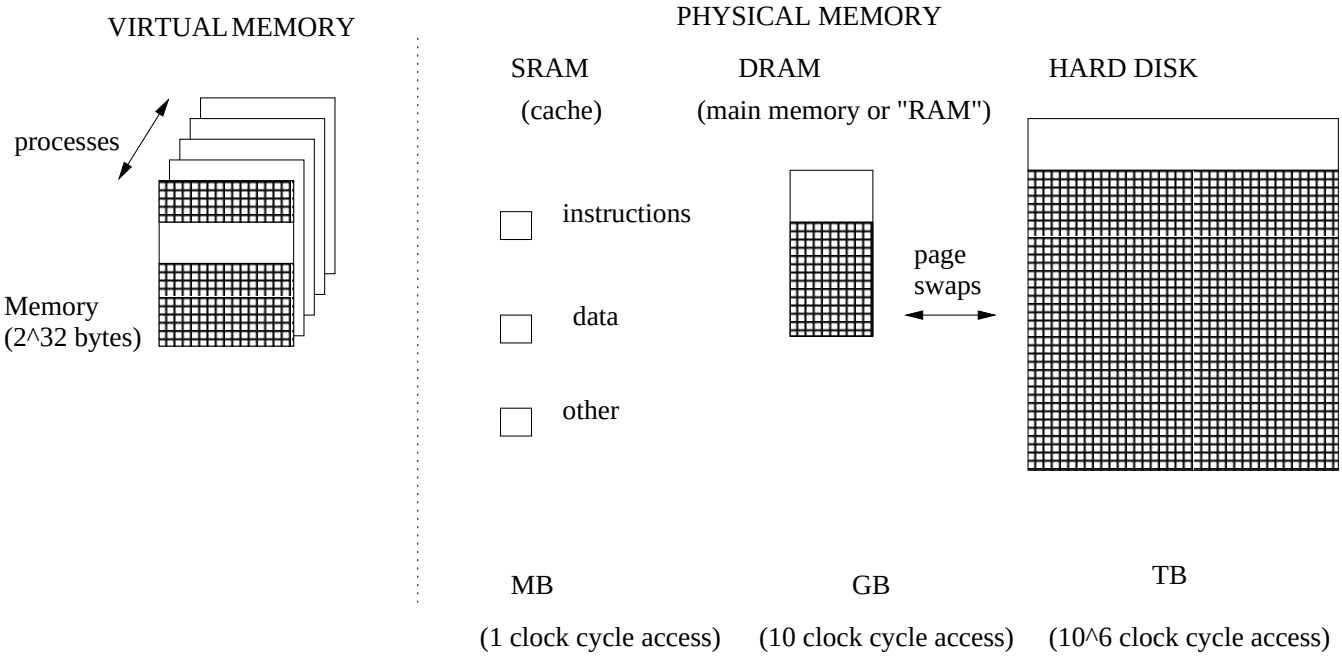
[1] The underlying technologies are *not* our concern in this course. You can just pretend both SRAM and DRAM are made out of transistors and other electronics and leave it at that. In fact, there are many subtypes of both SRAM and DRAM and these are changing remarkably fast.

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.



As discussed earlier today, in order for programs to run on the actual processor using real memory, program/virtual addresses must be translated into physical addresses. This is done by partitioning virtual address space and the physical address space into equal sized regions, called *pages*.



Let's work with an example in which each page is 2^{12} bytes i.e. 4 KB. 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. Thus, the 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 called the *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.

The computer needs to translate (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, since the mapping from virtual address to physical address is defined separately for each process.

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 page table entry holds a physical page number in main memory (RAM) and if the valid bit is 0 then the page table entry holds a physical page number on the hard disk.

Suppose main memory consists of 1 GB (30 bit addresses). Then there are $2^{18} = 2^{30}/2^{12}$ pages in main memory, assuming all of main memory (RAM) is paged. 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 $2^{28} = 2^{40}/2^{12}$ pages.

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				
⋮				

[ASIDE: 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.]

Where are the page tables ? First, note that page tables exist both in the virtual address space (program addresses) as well as in physical address space. Page tables are constructed and maintained by kernel programs, and so the kernel programs must be able to address them. Of

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

course, they also must exist physically somewhere, and so we need to be able to talk about physical address too.

For simplicity, you can think of page tables as *not* themselves lying in a part of Memory that is paged. (The reason this simplifies things, is that if page tables were themselves broken into pages, then you would need to think about a page table for the page table, ... which you probably don't want to think about right now.) In fact, often page tables are kept in paged memory, but let's keep it simple and ignore that.

Page faults and page swaps

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 gets copied from the hard disk into main memory. The kernel must choose an available page in main memory, and if there is no available page then it must first removeg some page from main memory in order to make space. This copying of a page from disk to main memory and from main memory to disk is called a *page swap*. We'll discuss page swaps again later in the course.

The way page faults are done from a software point of view is as follows. 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. The page fault handler then updates the page tables and administers the copy of data from main memory to and from hard disk.