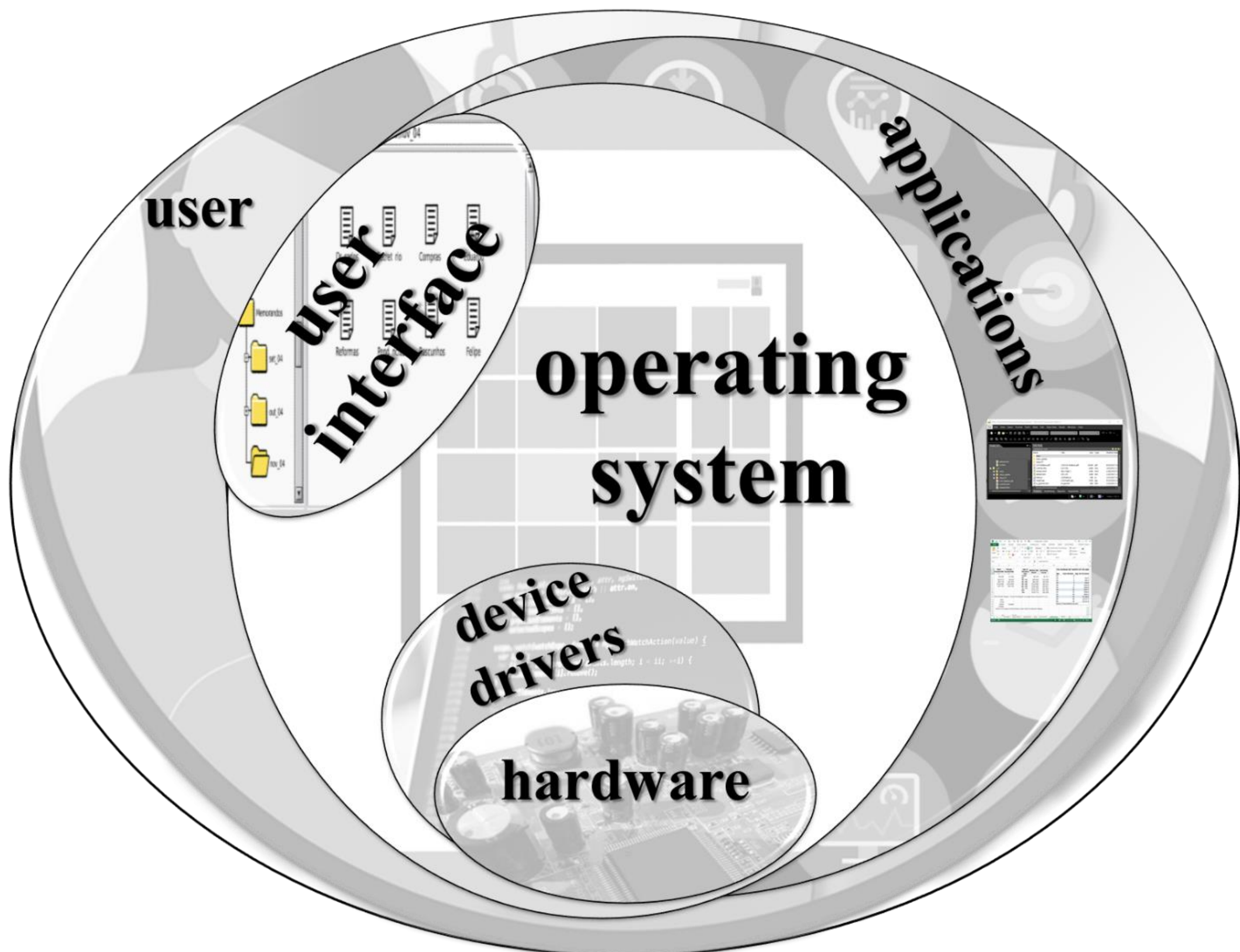
Operating Systems

The Operating System is the program that is in charge of everything on the computer. We don't want to have to deal with details of the hardware in order to tell the computer to run our programs and save our files, so the OS provides a buffer, dealing with the hardware so we don't have to. In fact, in most cases, the Operating system also deals with the hardware for our application software – programs deal with the OS instead of directly with the hardware.



The OS has three general jobs: managing all the resources of the computer, providing a user interface, and providing utilities.

Most OSs are very large programs, so we identify the *kernel* as the part of the OS that needs to run all the time for the computer to work, managing essential resources, while other parts of the OS, such as utilities and device drivers, can be loaded and unloaded as needed.

A *device driver* is a program that helps the OS deal with a particular piece of hardware. For example, when you add a new printer or graphics card to your system, you need to make sure you have the device driver for it, so the OS knows how to communicate with it[1].

Windows, MacOS, Android, and UNIX are examples of operating systems.

Utilities

We distinguish between *application software*, the kind of program that we use for its own sake, such as a word processor, web browser, or game, and *system software*, programs to help us manage the computer itself. The OS is system software, as are various utility programs that allow us to perform organizational or maintenance tasks on the computer.

The OS usually provides many utilities, for instance programs that run other programs (e.g. Start Menu) programs for organizing and browsing files (e.g. Windows Explorer) and programs for getting information about the computer's components (e.g. Device Manager).

Some utilities may not be provided by the OS, for instance not all OSs provide antivirus programs, recovery programs, or programs

---

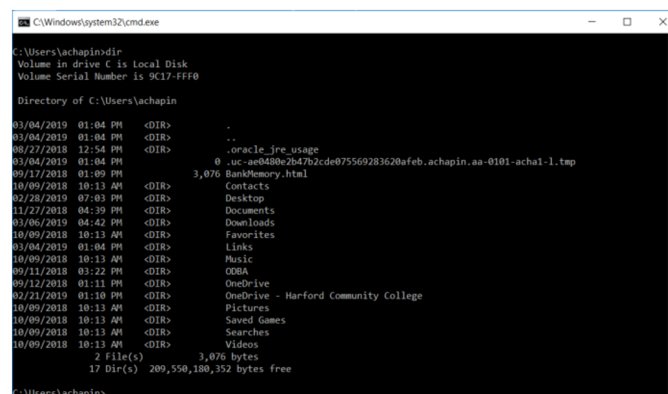
[1] Often the first advice when you're having trouble with a particular hardware device is that you check that you have an up to date device driver, or reinstall the device driver.

for re-naming many files at the same time. But in general, utilities have to work closely with the OS.

User Interface

The OS provides us with the *user interface* – the way the computer communicates with us and we communicate with it.

An operating system with a *command line* UI requires that the user type in commands to tell the computer what to do. If you do not know the right commands, you will not be able to use the system.

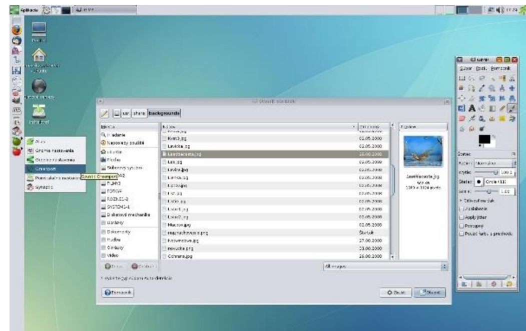


More recent OSs have replaced the command line with *GUIs – Graphical User Interfaces*, which use images. Most people find GUIs easier to use. However, if you do know the commands for a system that uses a command line, you may be able to do tasks, especially system maintenance tasks, much faster by typing one long command than by clicking many different parts of a GUI.

Almost all OSs now use *WIMP GUIs (Windows, Icons, Menus, Pointer)*. In a WIMP GUI programs are organized into boxes on the screen – Windows. Choices for actions and data are represented by small images – Icons. Lists of actions are provided, usually in a hierarchy that can be expanded – Menus.

To indicate what they want to do, users move a Pointer around the screen.

The Look and Feel of a UI is its visual style and the overall way the user interacts with it. For example, Windows and Linux have a somewhat different Look and Feel. However both are WIMP GUIS.



WIMP GUI has been the standard for OSs for a long time, because once people had learned it, they did not want to have to learn something else. This has begun to change because of the popularity of mobile devices. Mobile phones tend not to have enough screen space for multiple windows, and a pointer is not necessary on a touch screen, so WIMP GUIs are not well-suited to that format.

Kernel

The OS manages the resources of the computer. This means it must be running all the time. When the computer starts up, most of the job of the ROM is to see to it that the OS gets loaded from where it is stored on the hard drive into MM so that it can take charge. However, we don’t usually need all of the tools provided by the OS all the time, so only the kernel, the necessary part of the OS, is loaded. The kernel will then stay in MM for as long as the computer is running.

The OS loads other programs, and other parts of the OS, such as utilities, into MM, along with their data, and unloads them to make room for other things, but the kernel will stay.

Main Memory has finite space. Suppose we are designing an OS and decide that most of the OS should be in the kernel. Then the kernel will be large, and we will have considerably less space in MM for our other programs to run. If instead we only include the bare minimum in the kernel, we will have more space free in MM for other things.

On the other hand, anything included in the kernel is guaranteed to be loaded into MM by the time the OS is started, so any utilities that are in the kernel will start quickly when they are needed. If most of the OS is not in the kernel, each utility we use will take longer to start when we need it. (An OS with a particularly large kernel will take longer to start when we first turn on the computer, but then its contents will be faster to access.)

Managing the CPU: Multitasking

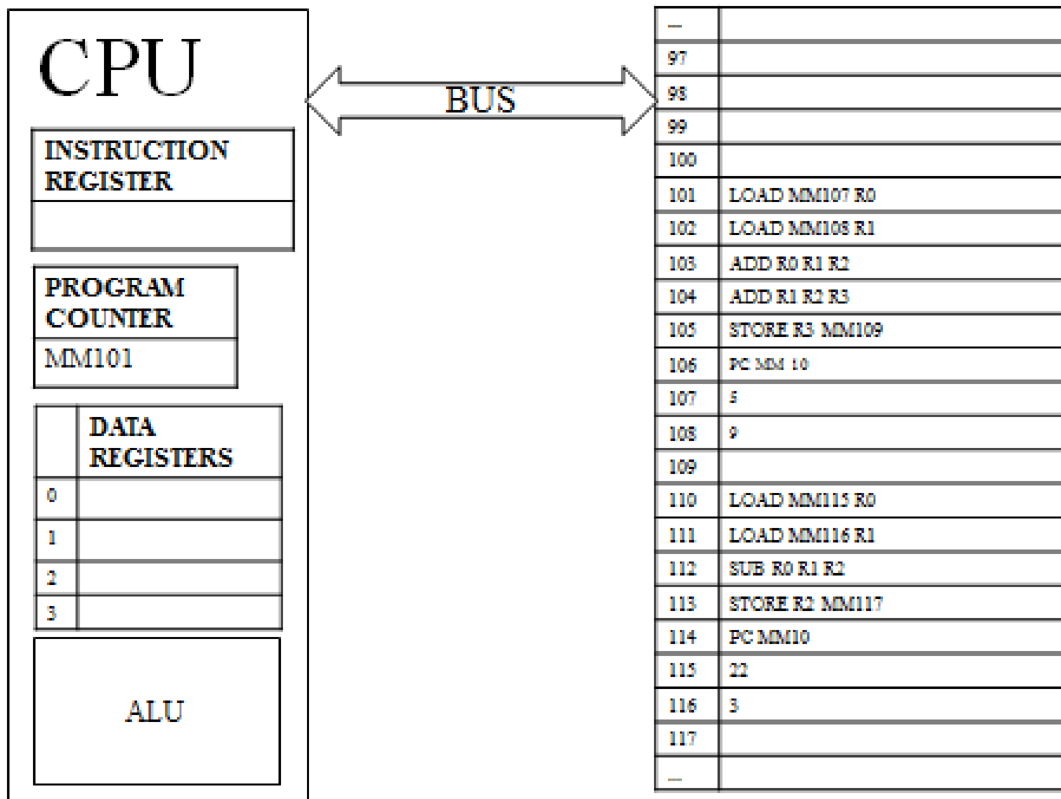
Note that the OS is running all the time, in addition to our other programs. Running more than one program at once is called *multitasking*. Remember that parallel processing allows us to run more than one instruction cycle at the same time on different CPUs, so we could run two programs at the same time by parallel processing if we have two CPUs. If we have four CPUs, we could run as many as four programs at the same time by parallel processing (remember that this includes the OS) but we frequently want to run many more programs than we have CPUs.

A *process* is a running program. In some cases we might run the same program more than once, in separate processes.

When there are fewer CPUs than processes to run at the same time, the computer cannot truly be running them at the same time. Instead, it fakes multitasking by *time slicing*: it switches back and forth between processes, doing a little of each at a time, so fast that they all *seem* to be running at the same time.

Multitasking by Time Slicing

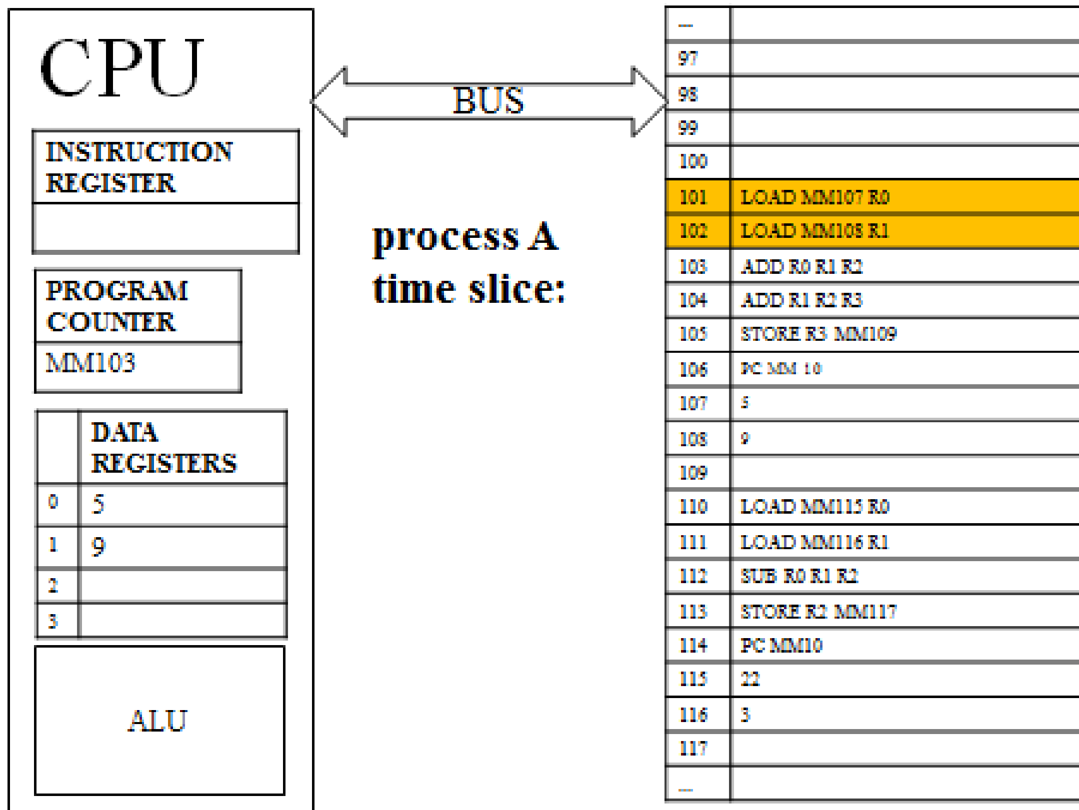
Suppose we are running two processes on one CPU. Call the one starting at MM101 Process A, and the one starting at MM110 Process B.



First Process A gets *a timeslice*: the OS sets the program counter to the address of the first instruction in Process A, and then the CPU executes a number[2] of instructions from Process A. These

[2] For real computers, which can do billions of instructions per second, this number may be very large. Our example is *tiny*.

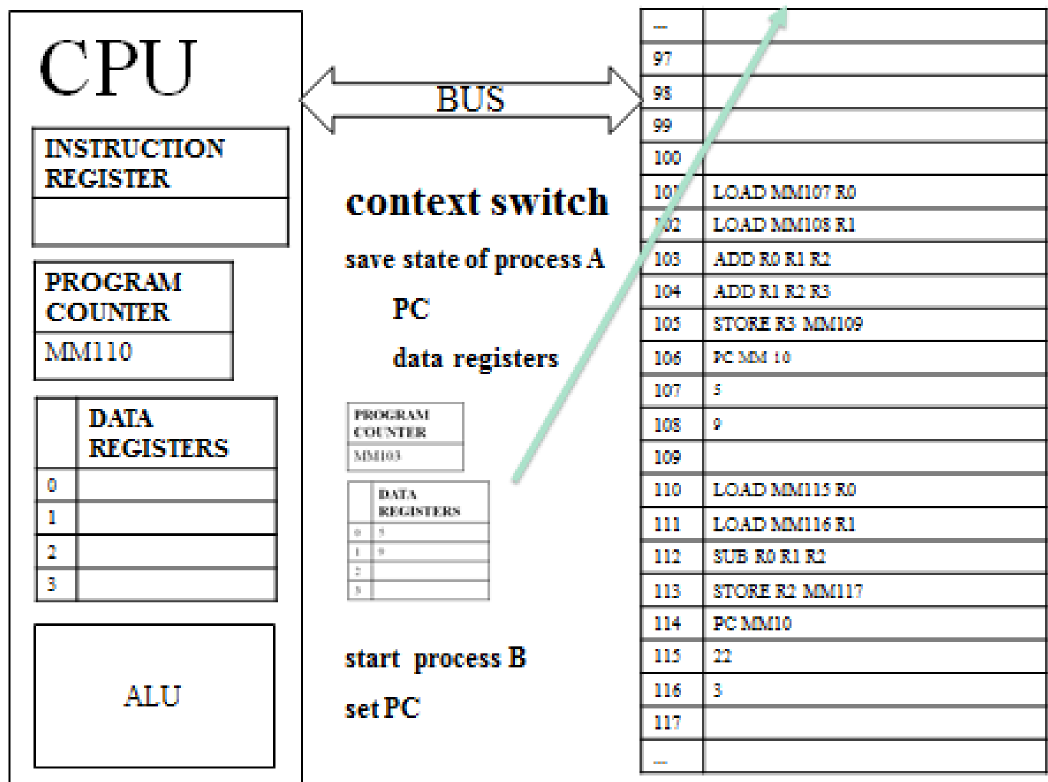
instructions may involve putting data into the data registers. It will certainly update the program counter at the end of each cycle.



Then Process A's time is up, and it is time for Process B to have a turn. When B gets its timeslice, it will need to use the program counter and probably the data registers. But we want to return to Process A later, and continue from where we left off, so we need to save anything currently in the data registers, and the current value of the program counter for Process A. Then, to start Process B, the program counter must be changed to the address of its first instruction.

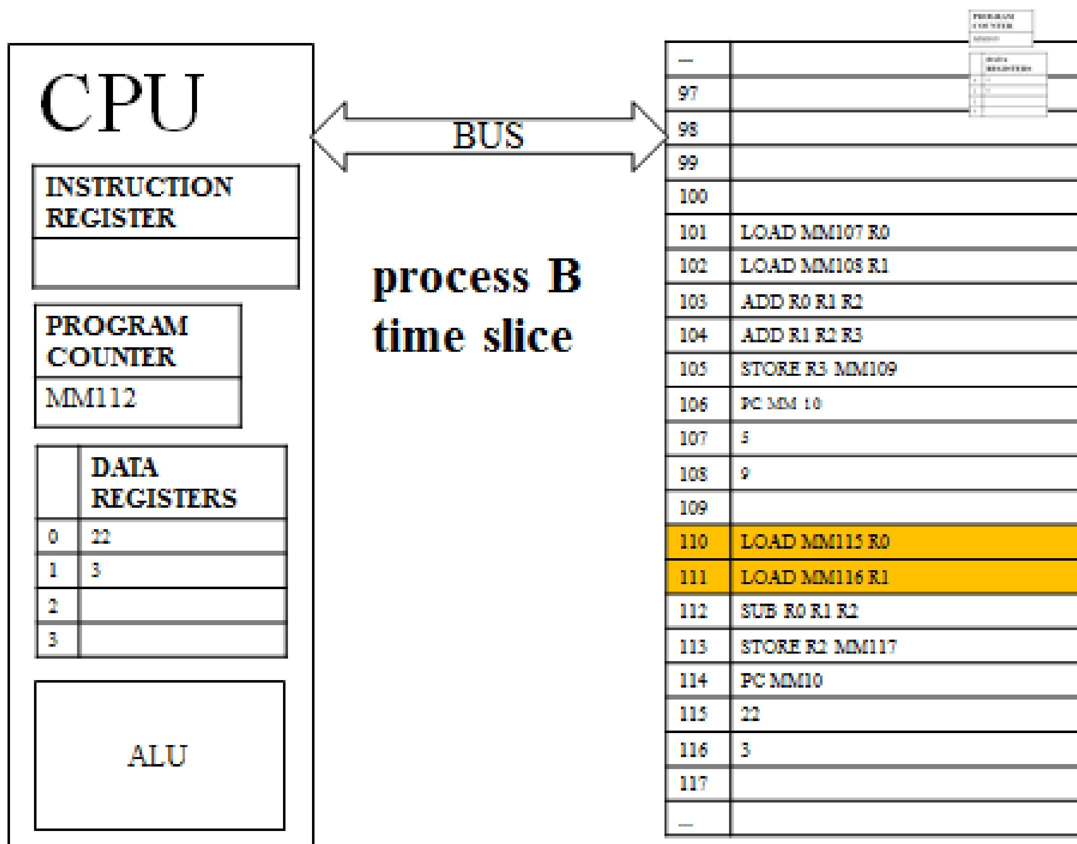
This business of saving our place from one process and getting ready to start another is called a *context switch*. The information we will need to continue Process A later, called its *state*, is stored

as data in Main Memory by the OS. The time spent on the context switch is called overhead.



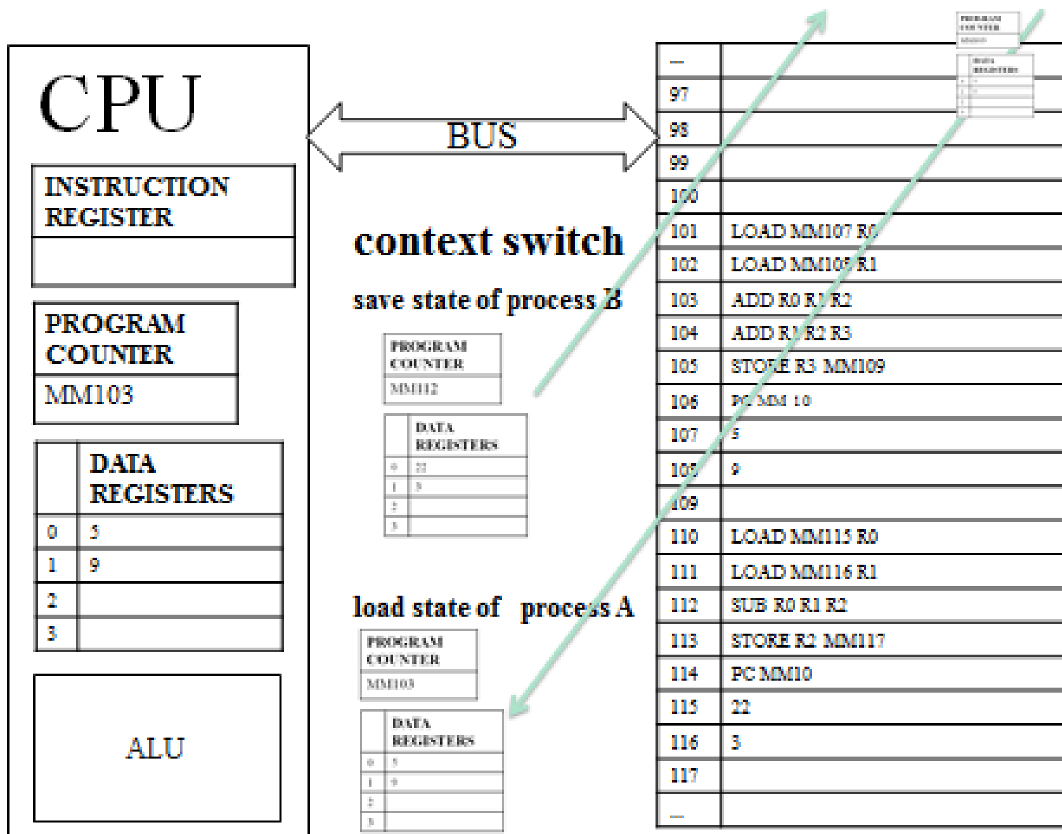
Now Process B gets a timeslice, so the CPU executes a number of instructions from Process B. Again these instructions may involve putting data into the data registers and will update the program counter.

Note that this ***does not*** have any effect visible to the user. This has ***nothing*** to do with the user switching programs. You will ***not*** see the windows switch on your screen. This is about what instructions are being executed in the CPU. Hundreds of timeslices may happen in a second with no sign of it on your screen (except that all your currently open programs keep running).

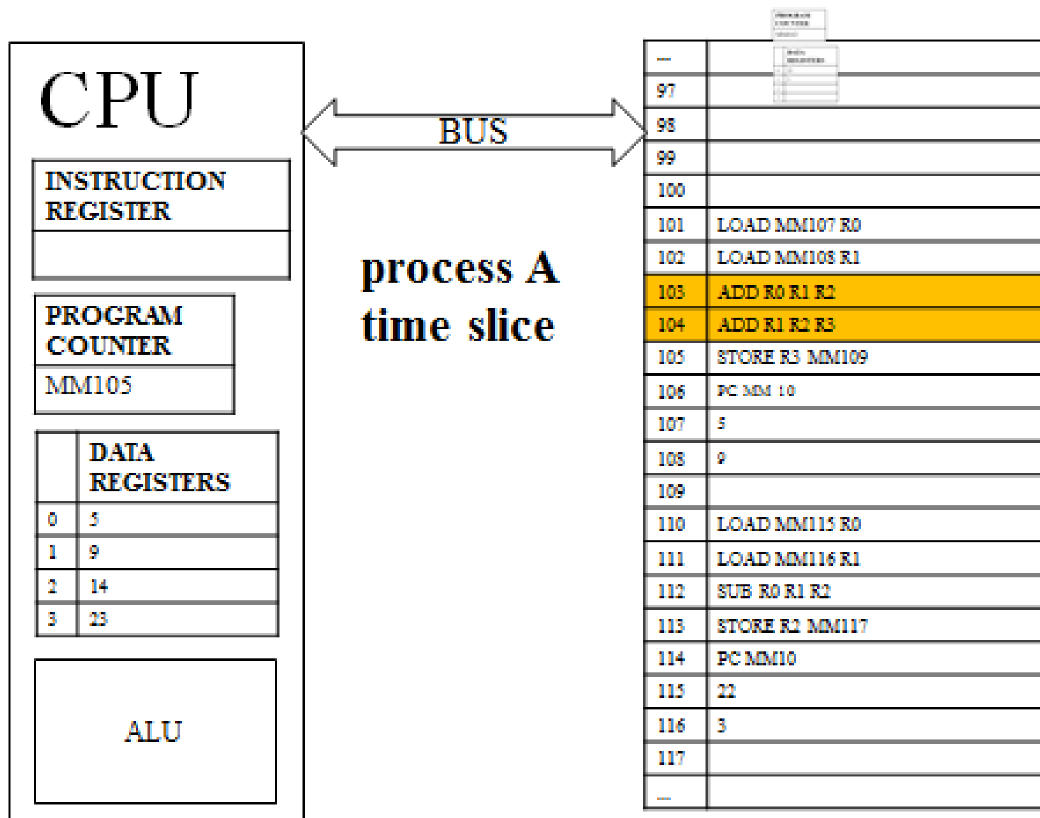


Now Process B's time is up and we need to go back to process A. During this context switch, we need to store the state (the program counter and data registers) for Process B to MM, and also get the values we previously stored for Process A and put them back in the CPU.

In general, a context switch will involve both storing the state of the current program to MM and loading the stored state of the next process from MM.



Once we have done this context switch, Process A can continue from where it left off, with no idea it was ever stopped.



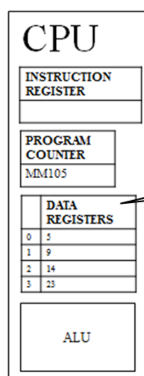
So Process A gets another timeslice, and a number of its instructions are executed.

We continue switching back and forth between these (so fast it looks like both run at once) until one of the programs is closed.

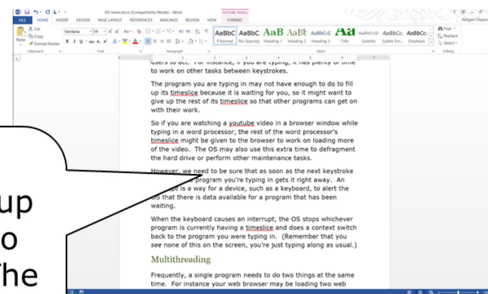
If there are more programs running at the same time, we would switch between them, for instance from Process A to Process B to Process C to Process D back to Process A, with a context switch between each. Instead of this simple rotation, many operating systems have more complex systems for deciding which process should get the next timeslice for the best results. One of the most important ways to adjust timeslicing is by adding interrupts.

Interrupts

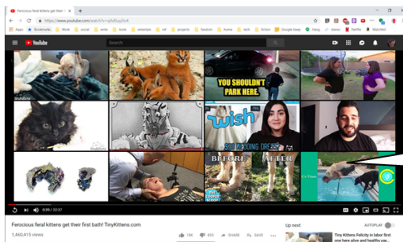
The computer is so fast that it spends a lot of time waiting for users to act. For instance, if you are typing, it has plenty of time to work on other tasks between keystrokes.



If you type 60 words per minute, I can do about a billion instructions between your keystrokes.



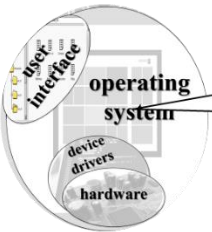
I've spellchecked your document and saved a backup and now I have nothing to do until your next keystroke. The rest of my timeslice is being wasted.



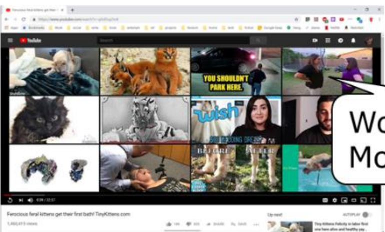
I've got plenty of kitty videos to load. Infinite kitties! …if only it was my turn.

The program you are typing in may not have enough to do to fill up its timeslice because it is waiting for you, so it might want to give up the rest of its timeslice so that other programs can get on with their work.

Since I’m waiting, I offer myself as tribute: I give up the rest of my timeslice



cool cool cool. Context switch to Youtube.



Woo hoo! Moar kitties!

I’ll let you know when I have something.

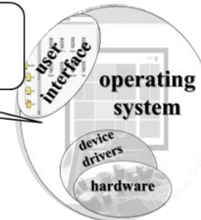
So if you are watching a youtube video in a browser window while typing in a word processor, the rest of the word processor’s timeslice might be given to the browser to work on loading more of the video. The OS may also use this extra time to defragment the hard drive or perform other maintenance tasks.

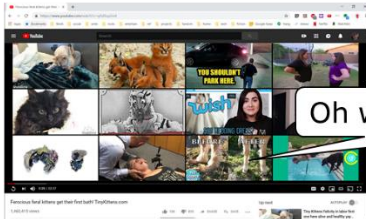
INTERRUPT!

Imma let you finish, but typing has happened!

Your keystroke is ‘m’

OK. Stop YouTube and context switch back to Word.

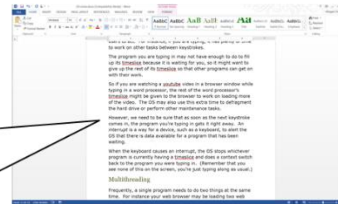




Oh well, brief but glorious.

Back in business! I will add ‘m’ to the document.

And spellcheck, grammar check, save backup, and… now I’m waiting again.



However, we need to be sure that as soon as the next keystroke comes in, the program you’re typing in gets it right away. An *interrupt* is a way for a device, such as a keyboard, to alert the OS that there is data available for a program that has been waiting.

When the keyboard causes an interrupt, the OS stops whichever program is currently having a timeslice and does a context switch back to the program you were typing in. (Remember that you *see* none of this on the screen, you’re just typing along as usual.)

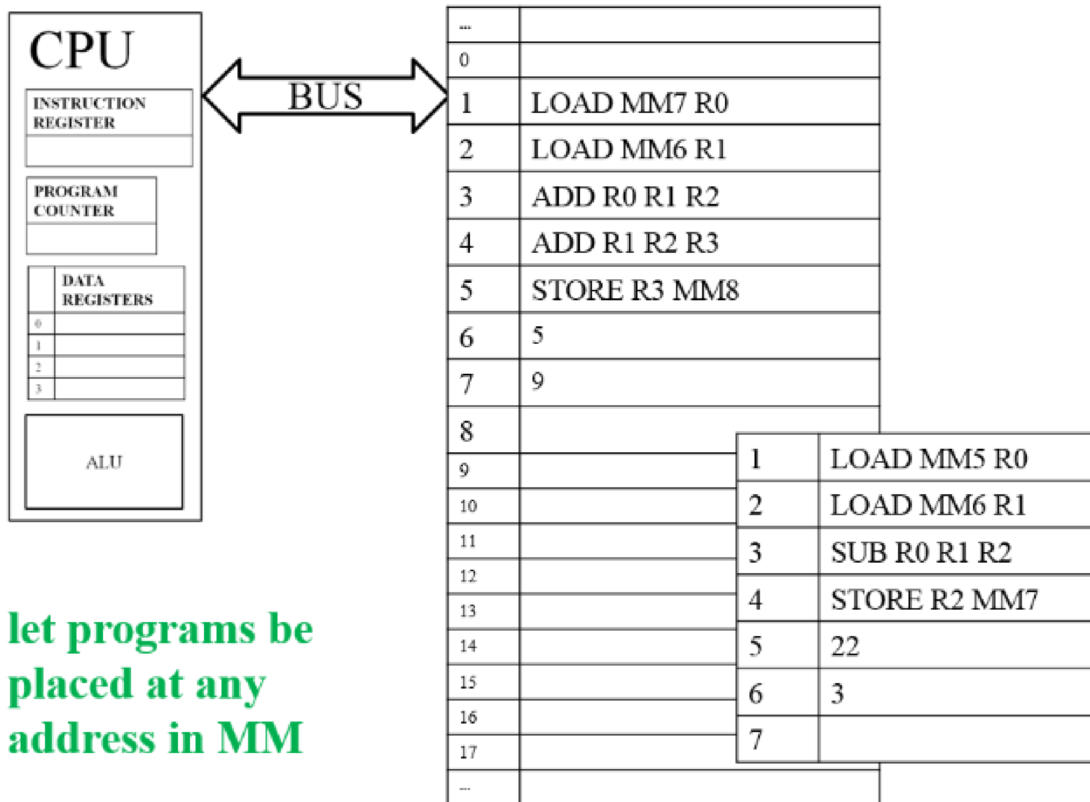
Multithreading

Frequently, a single program needs to do two things at the same time. For instance your web browser may be loading two web pages in other tabs while you read the current one, or your word processor may be saving and spell-checking your document while you type. A single program doing two things at the same time is

called *multithreading*. Like multitasking, this can be done with parallel processing, but mostly ends up being done by time slicing.

Managing Memory

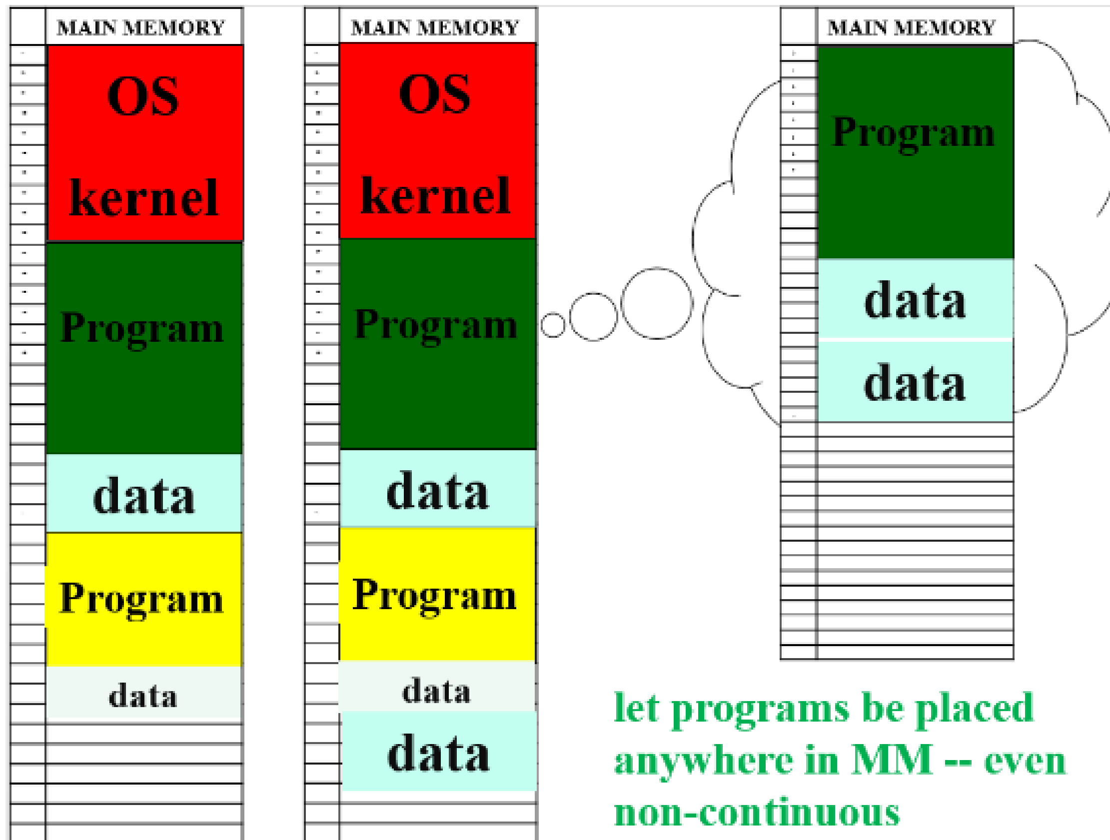
In addition to managing which program is running in the CPU at any time, the OS must manage the use of Main Memory. We have several problems in managing memory space that need to be dealt with.



Suppose we're running two programs in MM. We know that the OS loads them into MM after reading them from the hard drive, but suppose that both of them want to use the same addresses in MM. For example, suppose both programs have the instruction "STORE R0 MM6" – both were written assuming they can store

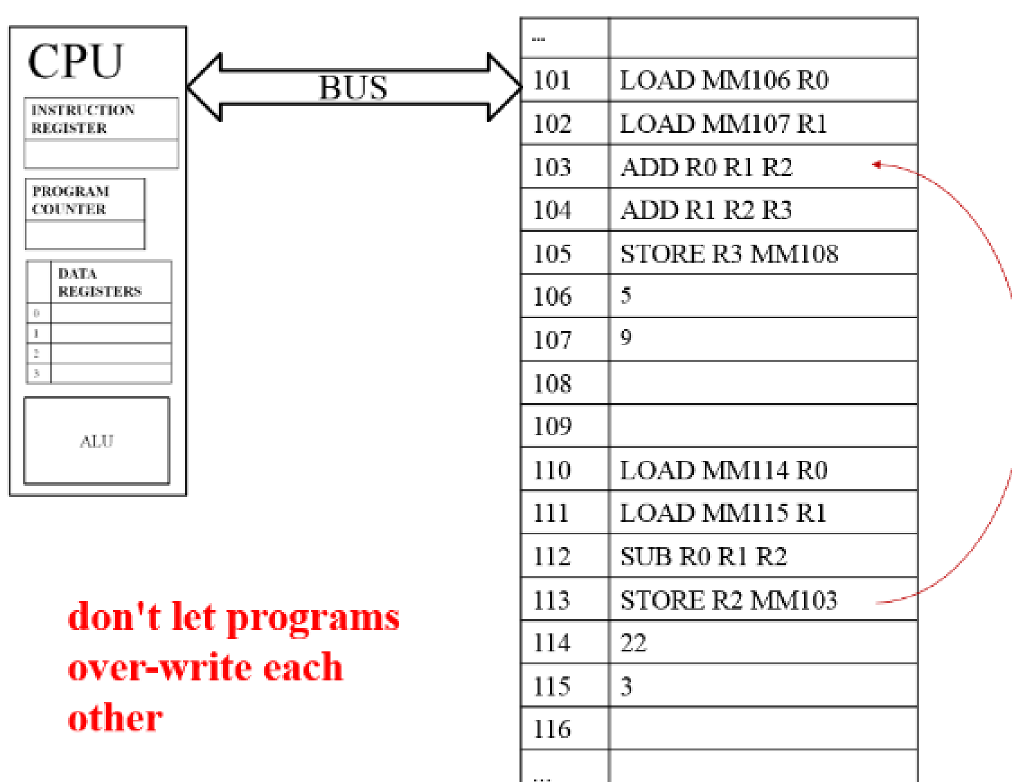
some data at Main Memory address 6. They can't both occupy the same memory locations, but we can’t just have the OS refuse to run those two programs at the same time either.

So one thing the OS needs to do is be able to place programs at any MM address and have them run, regardless of what MM addresses their instructions use.



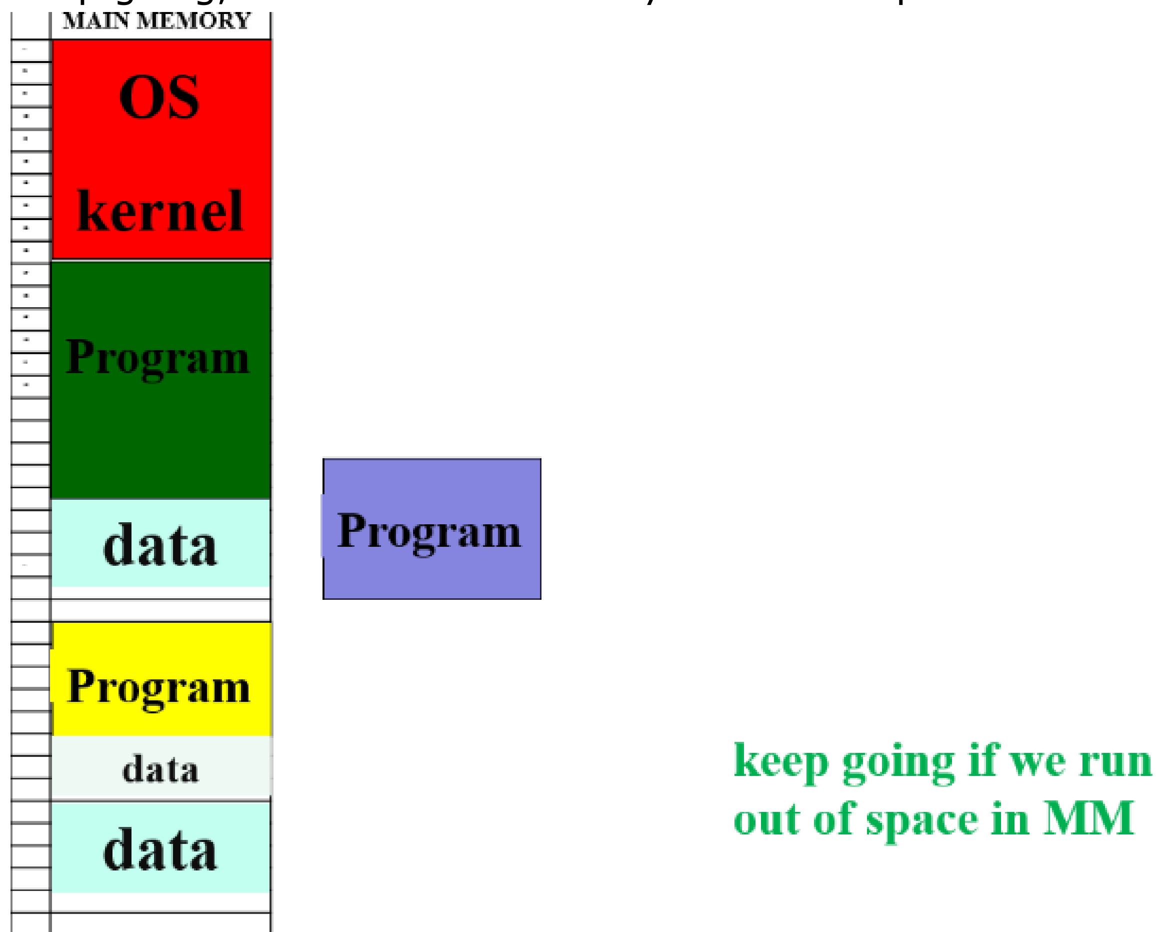
Suppose we have managed to load two programs and their data, but now one program needs more space for data – e.g. because we’ve written a longer document – but the next chunk of free space is below the other program. We don’t want to have to shift everything in MM around every time a program needs more space; that would slow all our programs down. So we need to be able to run programs anywhere in MM we have room for them, even in chunks of space that are not all together.

If two programs are running at the same time, each should be able to store data in MM, but they should not be able to store data in each others' space. If this were allowed, then at best, one program would put its data where the other had some instructions, and then the other program would stop with an error. More likely, one program would (maliciously or accidentally) replace some of the data of the other program, which would then continue to run, but with incorrect results. This could be disastrous – imagine if the data that got changed was your bank balance. So we need to ensure that programs cannot overwrite each other in MM.



Suppose we are running several programs (including the OS) and finally we simply fill up all the space in MM with programs and their data. But then the user starts one more program. We want to be able to open the program (or open more data files) and just

keep going, even if we are entirely out of MM space.



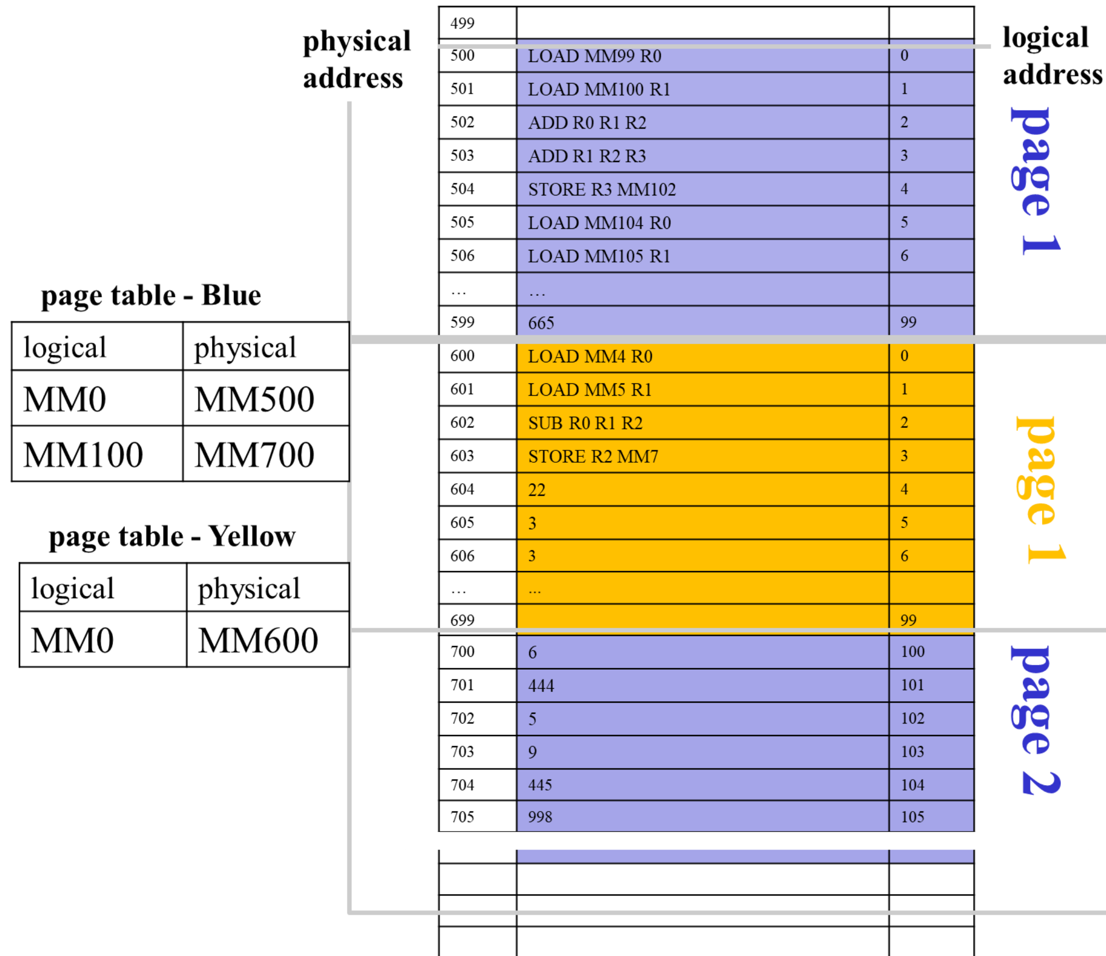
It turns out that the solution to all three problems is an approach to managing MM called Virtual Memory.

Virtual Memory

Virtual Memory is a system for keeping track of what is currently stored in Main Memory.

First, MM is divided into chunks called *pages*. Whenever a program needs space, it is assigned one or more pages in main memory. However, programs don't know which pages they will get each time they run. When a program is written, it can use *any* memory addresses in its instructions but programs are

usually written with the assumption that they will get the very first page, starting at MM0. VM allows the OS to let the programs think they are getting the memory location they are written for, even though they are actually being put in totally different pages.



The addresses used in the program are called *logical addresses*. The addresses of the actual pages the program happens to be in at the moment are the *physical addresses*.

The OS uses a page table – a table of pages and their contents – to keep track of which pages are used by each program. When

each instruction in a program is run, the logical addresses it uses are first translated to physical addresses using the page table[3].

Suppose each page is 100[4] addresses long. If an instruction uses address MM6, that is in the program's first page. If the page table says the program's first page starts at MM500, then logical address MM6 translates to physical address MM506. If another program whose first page starts at MM600 uses address MM6, then for that program it translates to MM606.

So, we have solved part of the first problem. The OS can now load any program into any pages in memory and run it, regardless of what addresses it uses.

In fact, we have solved the whole first problem. Suppose a program runs out of space; the OS then just assigns it the next free page, and translates based on which page the address falls in.

For instance, if a program uses address MM102, that must fall in the second page, so if the page table tells us that the second page starts at MM700, then MM102 translates to MM702. Now we can run programs in pages that are not next to each other. It wouldn't even matter if the second page came before the first. The programs are never aware that this translation is happening.

Our second problem was to stop programs from overwriting each other in Main Memory. It turns out that Virtual Memory can easily solve this as part of its system of translation.

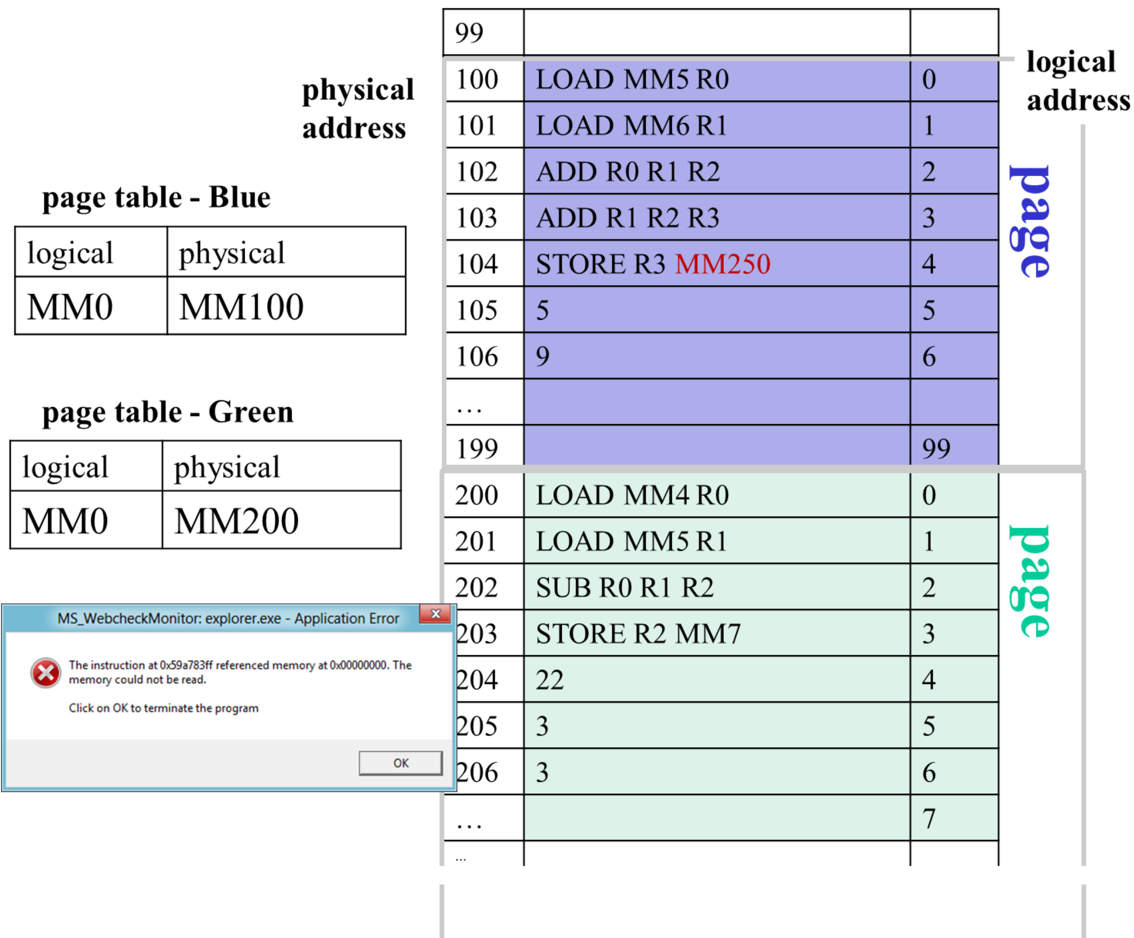
Suppose a program has two pages. Any address within those first two pages is in its own space, and doesn't affect any other

---

[3] Extra hardware is added to modern computers in order to be able to do this quickly enough.

[4] This was chosen for the example because it is a round number for humans. In the computer a round number in binary would be chosen, such as 512 or 1024.

program. But suppose it tries to use an address above 199. When faced with an address that can't be translated to a physical address within its allocated pages, Virtual Memory triggers an error.

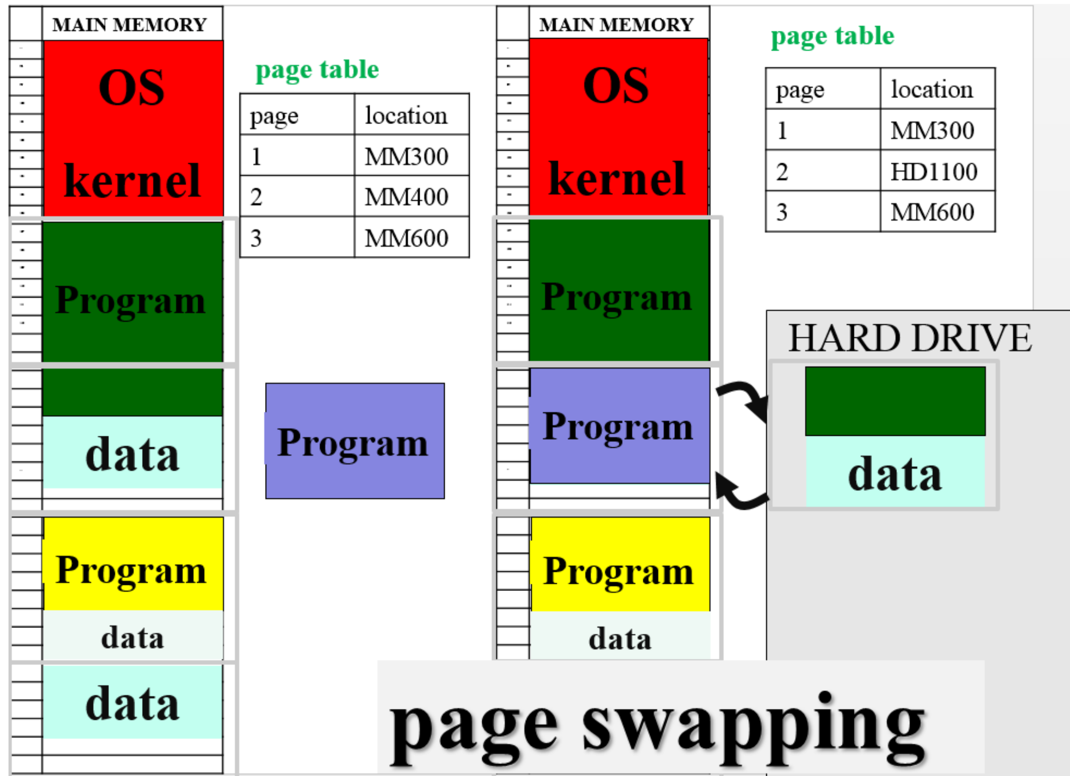


If the program actually needed more space for its data, it should have requested another page from the operating system.

Page Swapping

The third problem Virtual Memory will help us solve is how to continue opening more data and programs for current use even when we have run out of space in Main Memory. The solution will involve using the Hard Drive as an extension of Main Memory.

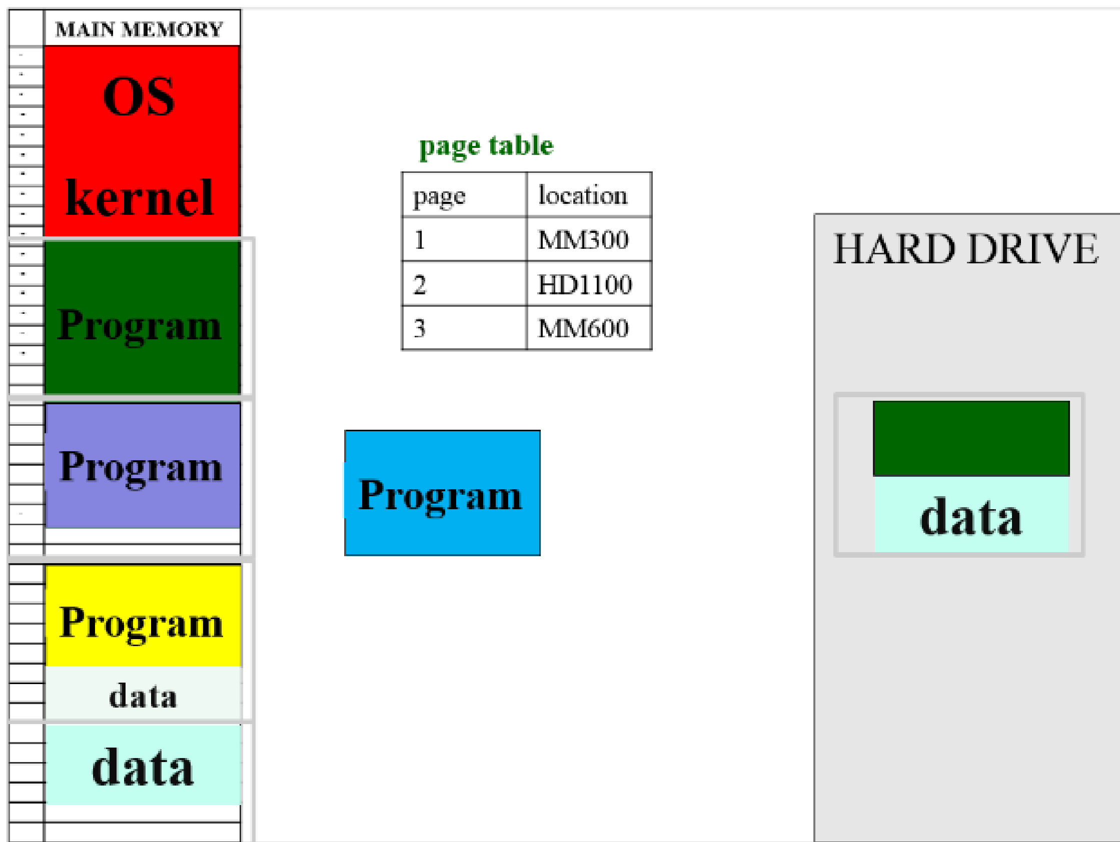
We will temporarily move pages out onto the hard drive and then bring them back as needed, using the page table to keep track. This is called *page swapping*.



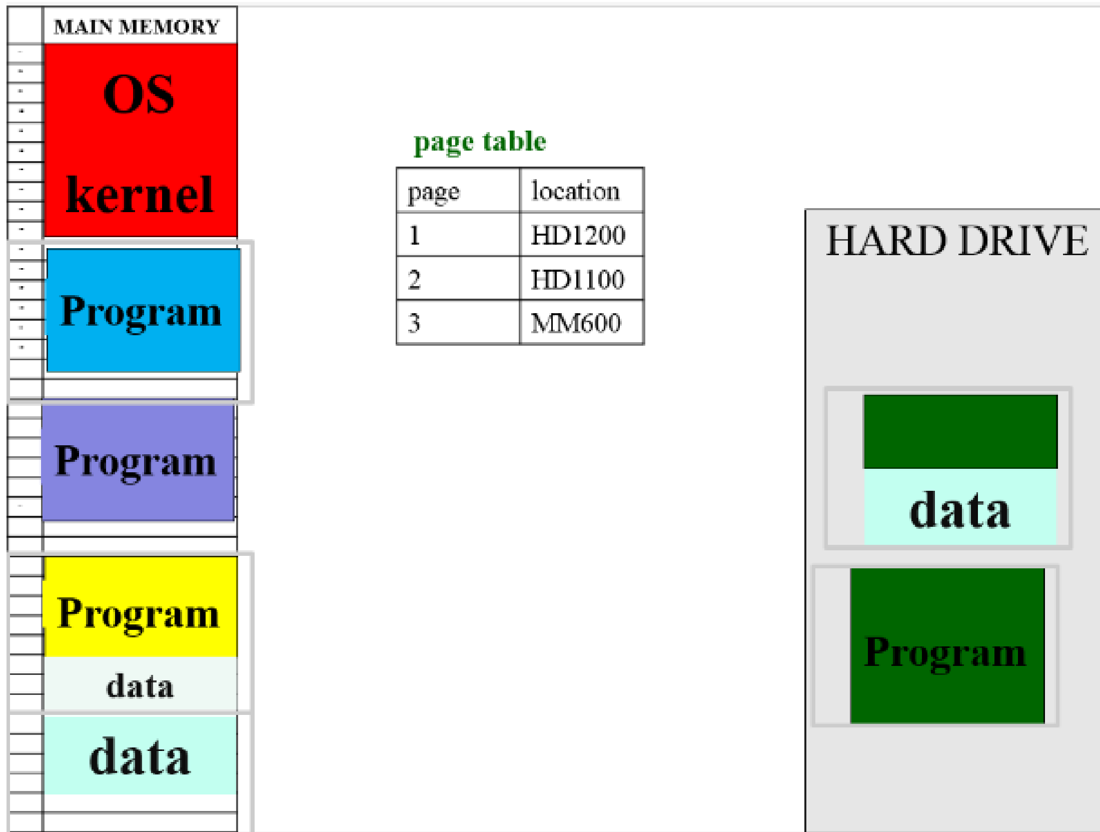
When we run out of space in MM but try to start another program or open another file[5], the OS identifies which page in MM is least recently used, because that page is likely to go unused for a while longer. The data and/or instructions on that page are temporarily copied out onto the hard drive and the page table is updated with a hard drive block address instead of a main memory address. Then that page in main memory is given to the new program.

---

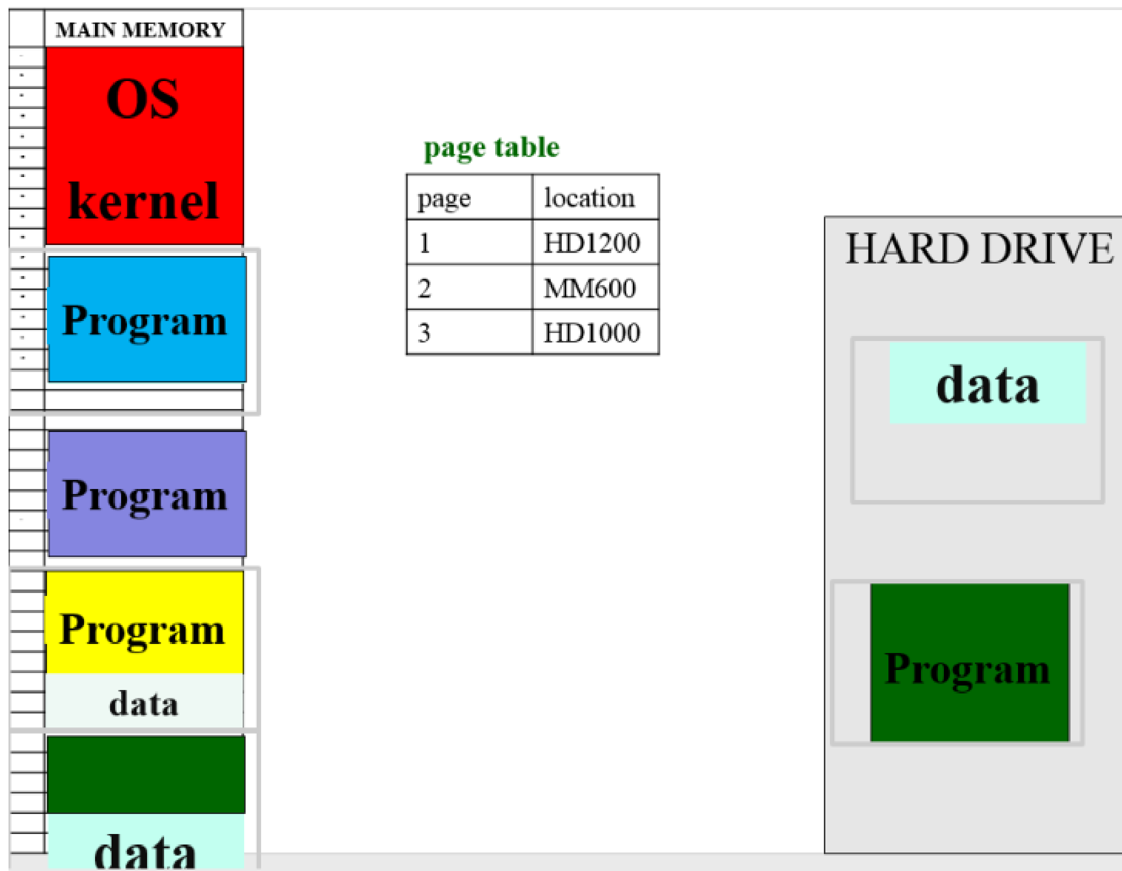
[5] Note that in fact, this could happen even if we only had one program open. If it was a large program or using very large data files (e.g. editing video) or we had a very small main memory, one program could fill up all the space MM has. Even an OS with a very big kernel might take up too much space for an old computer's MM. So we don't have to be multitasking to need page swapping.



Each time this happens, the OS is faking that the MM is growing one page larger, as it temporarily copy more pages out to the hard drive. Note that the user sees *none* of this. The program is *not* being closed. The program's data is *not* being saved. No windows are closed, or moved.



When next we need data or instructions from a page that is now on the hard drive, the OS discovers from the page table that the page isn't currently in MM. So the OS must go and read that page from the hard drive and bring it back into MM so it can be used.



page	location
1	HD1200
2	MM600
3	HD1000

This will probably require choosing another page to move out to the hard drive to make room. Note that there is no particular reason that the page should be swapped back to the same position it was in previously. We were translating the logical addresses to physical at the old location, we can just as easily translate at the new one. So we still use the least recently used page in MM to be the next move out to the HD to make room.

Note that this process for page swapping relies on having MM already divided into pages and having a page table to keep track of where the pages of a program are located, so page swapping relies on using virtual memory.

Because the hard drive is slower than main memory, there is one effect of page swapping the user might notice eventually. Every time we have to move a page to or from the hard drive, that will be slower than just dealing with data stored in MM. The more

pages currently on the HD, the slower our programs will seem. For this reason, when a computer is slow, people will often suggest adding more main memory space[6]. Notice, however, that what adding more main memory does is avoid page swapping, that is, avoid making it slower. Avoiding getting slower is *not* the same thing as getting faster; if you weren't filling up MM in the first place, adding more certainly won't speed up your computer.

The space set aside on the hard drive to use for page swapping is called the pagefile. If we have filled all the space in MM and all the space currently set aside for the pagefile, then the OS may finally alert the user and ask whether it can increase the size of the pagefile. Note that this decreases the space on the hard drive we can use to save other files. If we are out of space on the hard drive as well, then the OS must finally admit defeat and refuse to open anything else.

Unfortunately, the Windows message for increasing the amount of space for page swapping typically says something like "increase space for virtual memory." This has led a *lot* of people to be confused about what virtual memory actually means. Many people[7] say "virtual memory" when they in fact mean "page swapping (which relies on virtual memory)."

---

[6] They might say, "Your computer can't handle this huge program. You need to add more RAM."

[7] Sometimes including writers of textbooks. You should point and laugh at them.