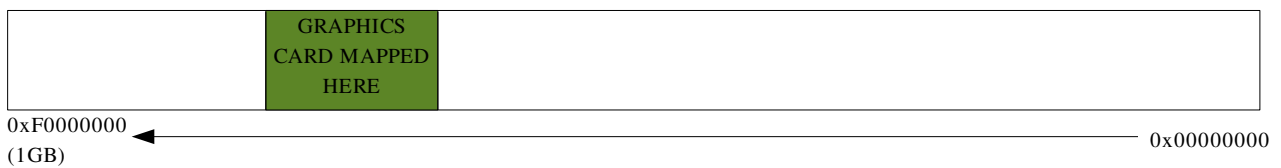


A Brief Overview of Virtual Memory

Virtual memory is vital in multi-tasking operating systems as it allows:

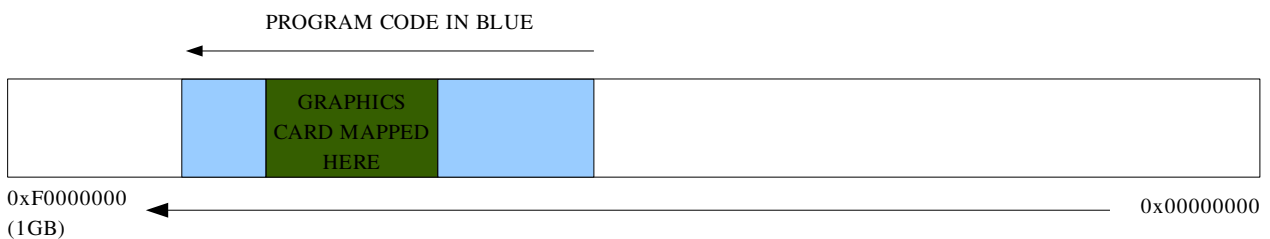
- Programs to use more memory than exists in a computer system.
- Protects running programs from memory holes in physical RAM (for example where we do memory mapped I/O).
- Reduces memory consumption as we can use the same memory (i.e. libc) in multiple programs at the same time.
- Provides the possibility for performance enhancements, copy-on-write etc.

If we assume a **physical address space** as follows:

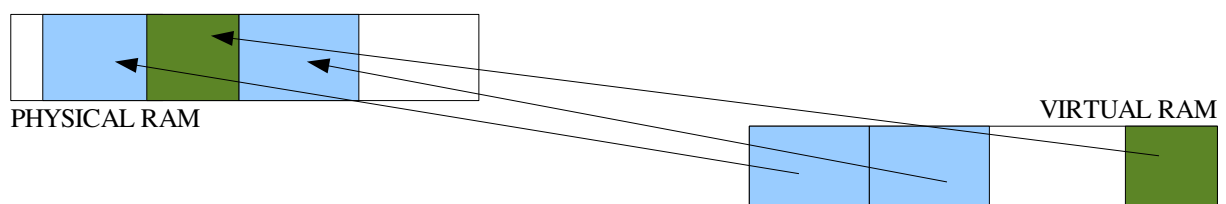


It is clear in this address space that we have 1GB of memory, and that a portion of that memory (say 256MB) is mapped for video memory. In other words, if we write to the memory where the video card is mapped we will be talking directly to the video card hardware, not RAM! This means that the amount of RAM we have available for programs is only 768MB. Further it should be noted that the video memory has not been mapped at the start or end of physical RAM. As such there is a memory 'hole'.

Given that programs execute using the fetch-execute cycle (i.e. take the current program counter value and load the instruction from RAM. Increment the program counter by 1 and load the next instruction. Continue this for ever!) it is clear that if we have a program that is in the middle of RAM, and split by the video memory (as shown in the following diagram) the CPU will start executing random data that is stored in the graphics card memory. This is clearly not desired.



So, what we want to be able to do is ignore the graphics card mapped memory, and this is where the first use of virtual memory appears. We can effectively create a 'fake' virtual address space, and map areas of the physical address space, as seen in the diagram below.



If a program were to execute using the virtual address space it is clear that there would be no hole. So, the question therefore remains – how is this implemented?

The CPU has a MMU or Memory Management Unit that is responsible for mapping virtual address spaces to physical addresses. On Intel/AMD x86 machines this can be done via two methods: paging and segmentation.

Segmentation

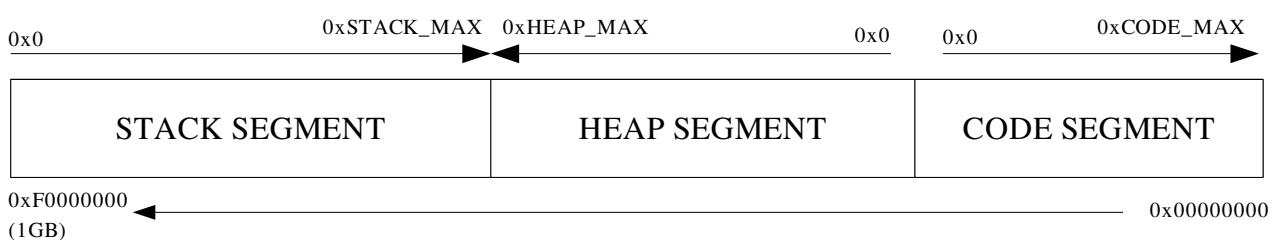
Segmentation allows us to split our address space (physical or virtual) into different segments. This means that for example, the code or text area of a program will be unable to interfere with the stack area. In addition, segmentation allows RAM to grow in different directions, meaning we can move the stack to the bottom end of RAM, and keep the code/text area of the program in the top half of the RAM. This further reduces the likelihood of a collision.

Occasionally we see errors such as 'Segmentation Fault' on UNIX machines. This happens when a stack or data segment overruns its boundaries – thus putting other data structures in danger of being overwritten or severely corrupted.

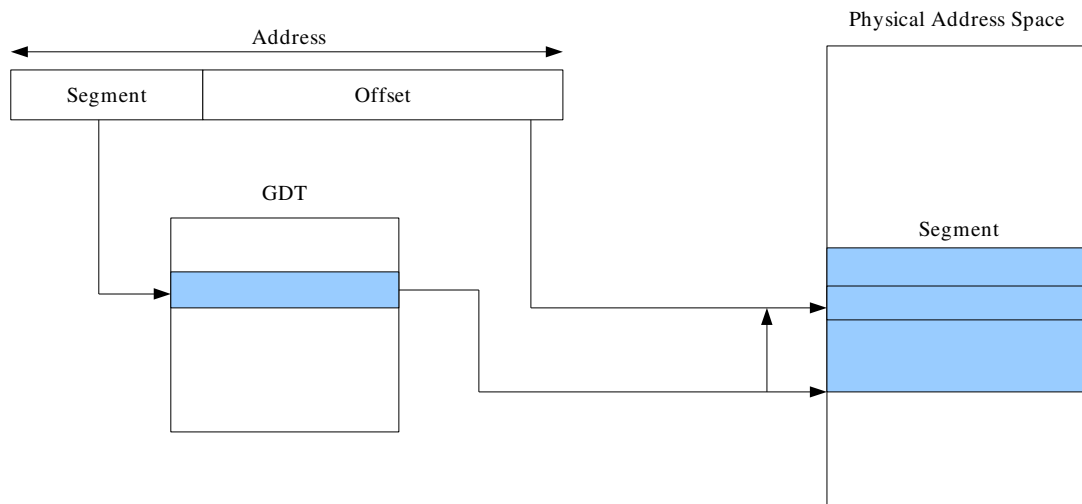
On x86 segmentation is supported by two tables held in memory (initially accessed via physical addressing, but it is possible to access via virtual addressing once paging is enabled) called the GDT (Global Descriptor Table) and the LDT (Local Descriptor Table).

The GDT is used specifically to define which segments of the physical address space are to be used for what. That is to say, is a segment to be used for data, stack, heap or text (code)? Further, it defines who has access to that segment, the Kernel, Drivers, Users or any combination of those. There are actually 4 possible levels of control over segments numbered 0 – 3. These are called 'rings'. They define which special instructions the CPU is allowed to execute. For example, in ring 0 every instruction is possible, however in ring 3 it is not possible to load in new GDT values or to execute instructions that change control registers. In addition, if a segment of memory is marked as ring 0 and the CPU is currently executing in ring 3. If a program tried to modify the memory in the segment marked as ring 0 an exception would be caused on the CPU. It is interesting to note that UNIX operating systems only use ring 0 and 3 for the kernel and user space respectively. I do not have any information regarding Windows, but it is interesting to note that the original Xbox which ran a modified version of Windows 2000 ran **everything** in ring 0 (kernel mode) – this is one of the reasons it was so easily hacked to be able to run Linux. Specifically in Linux/UNIX, device drivers and other low-level code which could run fine in ring 1 or 2 (but not ring 3) are usually executed in ring 0 – the kernel space.

In addition, the GDT also allows the specification as to which direction memory grows in a segment. This can be seen more clearly in the following diagram:



Notice that each segment has a start address of 0x0 and a maximum address (as defined by the size of the segment). This means that we now need a way of working out which address we mean when we get 0x0, and this is where the GDT starts to get properly used. The following diagram shows how selecting one GDT entry can have an impact on what physical (or virtual) address is actually reached.



Clearly if we changed to a different segment, but still had the same offset we would get a different physical address in memory.

The LDT works exactly as the GDT, but on a 'local' basis. That is, it is used in a virtual address space provided by paging, and not on the physical address space as with the GDT. The above diagram would be exactly the same for the LDT, except the 'physical address space' would be a 'virtual address space' which is provided via paging.

Paging

Segmentation alone is sufficient to implement a fully functional multi-process operating system. Every process could simply have its own segment. As such it would be able to start executing at address 0x00000000 from the base of the current segment selected. If a process ever reached the end of the segment an exception would be caused and the operating system could deal with it by increasing the size of the segment, killing the process, or various other possibilities. However, we will still be limited to the maximum amount of physical memory, and we still have the problem of memory holes.

Further segmentation does not solve the problem of having to load the same libraries in multiple times for each program. For example, assume you have two versions of the following program written in C, one that displays 'Hello World', the other 'Goodbye World'.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    printf("Hello World!");
    return EXIT_SUCCESS;
}
```

It is evident that this simple and small program is using functions that are provided elsewhere – specifically the `printf(...)` function, and the variable `'EXIT_SUCCESS'`. As it happens, `printf(...)` comes from the standard I/O library (`#include <stdio.h>`) and `EXIT_SUCCESS` comes from the standard library (`#include <stdlib.h>`).

Unfortunately these two libraries contain a vast amount of code, and as such in order for our program to run all that code must be copied from disk into memory at run time. If we simply used a segmentation model and were to run both programs at the same time we would end up having to read both libraries into memory twice. This is exceedingly wasteful of physical resources – both RAM, and time wasted reading the libraries in from disk twice!

Paging provides an answer to both of these issues with segmentation. It allows us to:

- Fake a full 4GB (32bit) address space for **every** program running. That is every process thinks it has a full 4GB of physical RAM to itself.
- Load in one copy of libraries (such as `libc`) and then share that between multiple-running programs.
- Protect running programs from memory holes in the physical address space.

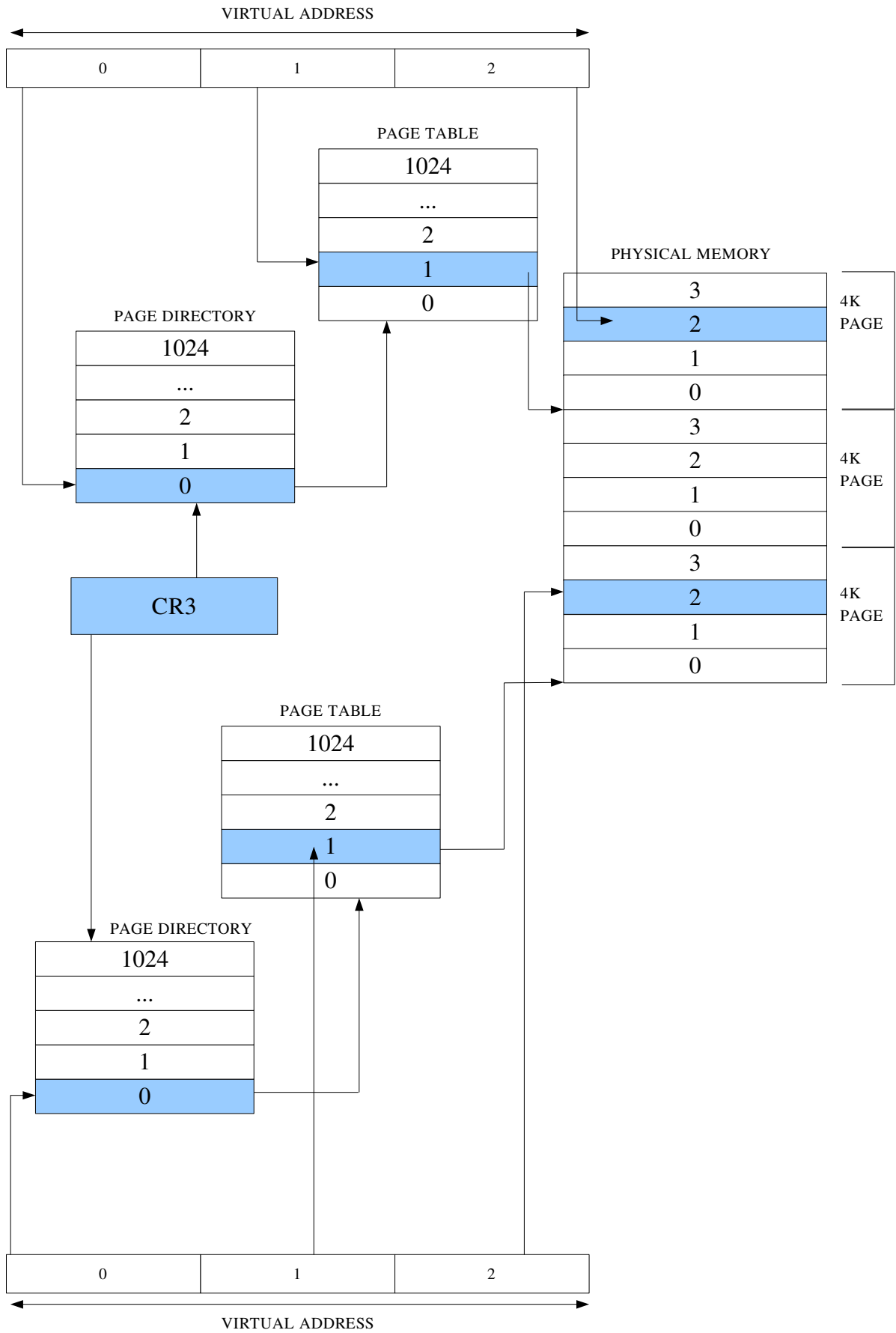
In addition, paging can also provide us with many of the protection capabilities that segmentation does. We can mark individual pages (a page being 4KB by default on x86, and 8KB default on SPARC processors) with a specific ring (again 0 – 3) – thus protecting against the execution or overwriting of code that does not belong to a specific process.

Paging provides us with the most benefits of the two methods of virtual memory management. However, it also can provide the most problems. Given that every program or process can effectively have a full 4GB address space, but there is a maximum of 4GB in total available in physical memory (in 32bit environments of-course) this means that if we have two programs we've got ourselves into a problem. We have promised both programs 4GB of RAM (a total of 8GB) but we actually only have a maximum of 4GB.

The solution is to only provide programs with the RAM as they need it. That is, tell a process: “here is a full address space, do with it as you please”. However, at the same time we only allocate 2 or 3 pages of memory – enough for the program to get going. Then we just hope that the programs never need the full 4GBs at the same time (there are solutions to this problem which are discussed later).

So, the question is – how is this trickery done? How do we fake a full address space? The solution – more tables! We've already talked a lot about the GDT and the LDT. These tables allow for effective segmentation. Paging requires another two tables. These tables are called the 'Page Directory (PD)' and the 'Page Table (PT)'. These, coupled with the use of a CPU register (specifically on x86 the register is CR3 – control register 3) we can map any virtual address in a processes own address space onto any physical address of our choosing.

This is best shown in diagram form. The diagram on the next page shows how we can have **two identical virtual addresses** that map onto two completely separate physical addresses – at the same time!



In the diagram the virtual address has been split up into three sections:

PAGE_DIRECTORY_ENTRY : PAGE_TABLE_ENTRY : OFFSET

The diagram shows that each of these represents a offset into one of the two tables, or into a page of memory.

When we do a task switch or context switch to change from one running program to another (this was discussed in the lecture and some sample assembly code was provided) we simply need to ensure that we change the value of CR3 to point to the relevant page directory. Every process can then have a single page directory and a page table for every entry in the page directory.

On x86, and in 32-bit mode, the page directory can have 1024 entries. Each page table can then have a further 1024 entries. The result of this is:

$$\begin{array}{rcccccc} \text{Page Directory Entries} & * & \text{Page Table Entries} & * & \text{Page Size} & & \\ 1024 & * & 1024 & * & 4\text{K} & = & 4\text{GB} \end{array}$$

What was not shown in the above diagram was how two virtual addresses could map to the same physical memory. However, given that **any virtual address** can map to **any physical address** it is clear that we could map two different virtual addresses to the same physical address – thus sharing resources such as memory mapped I/O devices, and libraries that are in use often – i.e. libc.

However, we do now start to see the problems that can arise if we don't have the specified page in physical memory, or if physical RAM is full, or if two different virtual address spaces want to use the same physical RAM for two **different** things.

This is where swapping comes into play.

Swapping

Swapping does exactly what it says on the tin. It is a software solution to the virtual memory problems. Swapping allows the software kernel or operating system to take a physical page in RAM and copy it to the hard disk. At this point the page on disk is said to be **swapped out**.

At this point, the kernel can then put a new page of memory in its place, or it can **swap in** a page that was previously swapped out. In other words, swapping allows other storage mediums to become an extension of physical memory. Pages that are not currently being used (the process is sleeping, dead but hasn't released its address space yet, or perhaps simply hasn't been used recently) can be swapped out to disk in favour of programs that are active and need more physical RAM.

This leads to the question – what if the page we want isn't currently in RAM, i.e. it has been swapped out?

If the processor gets a request to access a page that has been swapped out the page table entry will contain a flag which means that the page is not present. This will cause a **page fault** exception on the CPU. Exceptions on the CPU act almost exactly the same as interrupts, and as such the CPU

will enter kernel mode and call the software page fault handler. At this point, the page fault handler can copy in the required page to physical memory and update the flag to say that the page is now present.

At this point, a **rti (return from interrupt)** assembly command is issued and the CPU can continue executing the code (i.e. jumping to or reading/writing to/from the memory that was previously not there) without the user program ever knowing that the memory was not in physical RAM all the time.

It is clear (I hope) to everyone that swapping is both a good and bad thing. Yes, it allows us to use more RAM than we really have – but it is slow. It is fair to say that if your computer is swapping a lot then it is probably time to get some more RAM. You could conceivably see a considerable increase in performance of your computer without upgrading CPUs etc by simply increasing the RAM – up to a point. If your computer has to pull pages from disk, this can take a long time compared to accessing the page from main memory. As such, the less we need to swap to disk, the faster we can access our information, and so the faster we can execute our programs. However, if you find that swapping happens occasionally then this is **not a problem, and is actually expected**.