

## Chapter 7: Memory Management in Theory and x86

### Introduction

It's well-known to us right now that one of the most important aspects in modern operating systems is to protect the memory in a way that doesn't allow a process to access or write to the memory of another process, furthermore, the memory of the kernel should be protected from the running processes, that is, they should be prevented from accessing directly the memory of the kernel or writing to the memory of the kernel. When we use the term *memory of the kernel* or *memory of the process* here we mean the region of the main memory that is being used by the kernel or the process and all of its data or code is stored in this region of the memory. In chapter 3, we have presented the distinction between the logical view and physical view of the memory and one of the logical views of the memory has been presented on the same chapter, this logical view was segmented-memory model. We have seen how the hardware has employed the protection techniques to provide memory protection and protect the segments from each other. In the same chapter, we have presented another logical view of the memory, it is flat-memory model, which is exactly the same as the physical view of the memory. In this view, the memory is a big bunch of contiguous bytes and each byte has its unique address that can be used to refer to this byte in order to read it or to write to it. We know that modern operating systems use the flat-memory model and based on that we decided to use this model on x86 kernel instead of the segmented-memory model. Deciding which model to use is the job of the kernelist. However, unlike segmentation, when we introduced the flat-memory model, we haven't shown how the memory can be protected, in this chapter we present one of the methods that can be used to implement memory protection in with flat-memory model. This technique is known as *paging*, it is a well-known technique that is used widely by modern operating systems and it has a hardware support in x86 architecture.

### Paging in Theory

In paging, the memory of the process (before being loaded to the physical memory) is divided into a number of fixed size blocks known as *pages*, in the same manner, the physical memory is divided into blocks with the same fixed size, these blocks of physical memory are known as *page frames*. Figure shows an example of pages and page frames, as you can see in the figure, process A is divided into  $n$  pages and the main memory is divided into  $n$  page frames. Because both page and page frame have the same size, for example 4KB<sup>1</sup>, each page can be loaded exactly into one page frame. To load process A into the

---

<sup>1</sup>That is, each page is of size 4KB and each page frame is of size 4KB,

memory, each of its pages should be loaded into a page frame. A page can be loaded into any page frame, for example, let's assume we are loading page 0 of process A and the first free page frame that we found is page frame 30, then, the page 0 is loaded into page frame 30. Of course, the pages of more than one process can be loaded into the page frames.

A data structure known as *page table* is used to maintain these information about the mapping between pages and their corresponding page frame. Each process has its own page table, in our example of process A, the information that tells the processor that page 0 can be found in page frame 30 is stored in process A's page table. In paging, any memory address generated by the process to read or write some data to the memory will be a logical memory address <sup>2</sup>, that is, a not physical memory address, it has a meaning for the process itself, but for the processor it should be translated to the corresponding physical memory address. The page table of the process is used to perform this translation. Basically, every generated logical memory address of a process running in a paging-enabled environment is composed of the following parts: the page number and the offset. For example, assume a system employs paging with length of 2 bytes of memory addresses <sup>3</sup>. In this hypothetical system, the format of logical address is the following, the first byte represents the page table and the second by represents the offset. Process B is a process runs in the system, assume that it performed an instruction to read data from the following memory address 0001 0050h, this is a logical memory address that needs to be translated to the physical address to be able to get the required content. Based on the format of the logical memory addresses in this system, the first byte of the generated memory address which is 0001h represents the page, that means that the required data is stored in page 0001h = 1d of the process B, but where exactly? According the the generated address, it is on the offset 0050h = 80d of that page.

To perform the translation and getting the physical memory address, we need to know in which page frame the page 1 of process B is loaded. To answer this question the page table of process B should be consulted. For each pages, there is an entry in the page table that contains necessary information, and of course one of those information is the page frame that this page is stored on. It could be the page frame number or the base memory address of the page frame, it doesn't matter since we can get the base memory address of the page frame by knowing its number and the size of page frames. After getting the base memory address, we can combine it with the required offset to get the physical memory address of the data in question. The hardware that is responsible for the process of memory address translation is known as *memory management unit* (MMU) .

Sometimes, the page table is divided into more than one level. For example,

---

<sup>2</sup>In x86, this logical memory address is known as *linear memory address* as we have discussed earlier in chapter

<sup>3</sup>In 32-bit x86 architecture, the length of memory address is 4 bytes = 32 bits, that is, 2<sup>32</sup> bytes = 4GB are addressable. An example of a memory address in this environment is FFFFFFFFh which is of length 4 bytes and refers to the last byte of the memory.

in two-level page table, the entries of the main page table refers to an entry on another page table that contains the base address of the page frame, x86 architecture uses this design, so we are going to see it on details later on. The reason of using such design is the large size of page tables for a large main memory. As you know, the page table is a data structure that should reside in the main memory itself, and for each page there is an entry in the page table, in x86 for example, the size of this entry is 8 bytes. Furthermore, the size a page tend to be small, 4KB is a real example of page size. So, if 4GB is needed to be presented by a page table with 8 bytes of entry size, then 8MB is needed for this page table which is not a small size for a data structure needed for each process in the system.

It should be clear by now how paging provides memory protection. Any memory address that is generated by the process will be translated to the physical memory by the hardware, there is no way for the process to access the data of any other process since it knows nothing about the physical memory and how to reach it. Assume process C that runs on the same hypothetical system that we have described above, in the memory location that's represented by the physical memory address 00A1 039Bh there is some important data which is stored by the kernel and process C wishes to read it. If process C tries the normal way to read from the memory address 00A1 039Bh the MMU of the system is going to consider it as a logical memory address, so, the page table of process C is used to identify in which page frame that page 00A1h of process C is stored. As you can see, the process knows nothing about the outside world and cannot gain this knowledge, it thinks it is the only process in the memory, and any memory address it generates belongs to itself.

## Virtual Memory

In multitasking system, beside the need of memory protection, also, the main memory should be utilized as much as we can. In such environment, multiple processes should reside in the main memory and at some point of time the main memory will become full and the kernel will not be able to create any new process before stopping a currently running process to use its space in the main memory. There are many situations where the current processes are occupying a space from the main memory but doesn't really use this space, that wastes this space since it can be used to load a process that really needs this space. An example of these situations is when the software is idle, that is, doing nothing but waiting for some external action (e.g. a button click), in this case the only active code of this software that should be there in the main memory is the code that makes the software wait for an event. Furthermore, modern software tend to be too large, there are a lot of routines in a code of modern software that might not be called at all during executing that software, also, loading the code of those routines into the main memory wastes the occupied space, the routines will be there in the memory, taking some space that can be used for more useful

purposes and they never been called.

Virtual memory is a memory management technique that can be used to utilize the main memory. You might noticed in modern operating systems, you can open any number of software in a given time and you never get a message from the operating system that tells you that there is no enough space in the main memory although the software that you are running need a large space of memory<sup>4</sup>, how can that be achieved? Well, by using virtual memory which depends on paging that we have discussed earlier. Regarding to paging, we may ask ourselves an important question, should all pages be loaded into the memory? In fact, not really. As we have said, the binary code of the software may have a lot of routines that may not be called, so, the pages that contain these routines should not be loaded into the memory since they will not be used, instead, this space can be used for another pages that should really be on the memory. To realize that, we the software is loaded for the first time, only the page the contains the entry code of the software (e.g. `main` function in C) is loaded into the memory, not any other page of that software. When some instruction in the entry code tries to read data or call a routine that doesn't exist on the loaded page, then, the needed page will be loaded into the main memory and that piece which was not there can be used after this loading, that is, any page of the process will not be loaded into a free page frame unless it's really needed, otherwise, it will be waiting on the disk, this is known as *demand paging*.

By employing demand paging, virtual memory save a lot of memory space. Furthermore, virtual memory uses the disk for two things, first, to store the pages that are no demanded yet, they should be there so anytime one of them is needed, it can be loaded from the disk to the main memory. Second, the disk is used to implement an operation known as *swapping*. Even with demand paging, at some point of time, the main memory will become full, in this situation, when a page is needed to be loaded the kernel that implements virtual memory should load it, even if the memory is full! "How?" You may ask, the answer is by using the swapping operation, one of page frame should be chosen to be removed from the main memory, this frame in this case is known as *victim frame*, the content of this frame is written into the disk, it is being *swapped out*, and its place is used for the new page that should be loaded. The swapped out page is not in the main memory anymore, so, when it is needed again, it should be reloaded from the disk to the main memory. The problem of which victim frame to choose is known as *page replacement* problem, that is, when there is no free page frame and a new page should be loaded, which page frame should we make free to be able to load the new page. Of course, there are many page replacement algorithms out there, one of them is *first-in first-out* in which the page frame that was the first one to be loaded among the current page frames is chosen as a victim frame. Another well-known algorithm is *least recently used* (LRU), in this algorithm, everytime the page is accessed, the time of access is stored, when a victim frame is needed, then it will be the oldest one that has been accessed.

---

<sup>4</sup>Modern web browsers are obvious example.

The page table can be used to store a bunch of information that are useful for virtual memory. First, a page table usually has a flag known as *present* which indicates whether the page is presented in the main memory or not. By using this flag, the processor can tell if the page that the process tries to access is loaded into the memory or not, if it is loaded, then a normal access operation is performed, but when the present flag indicates that this page is not in the memory, what should be done? For sure, the page should be loaded from the disk to the memory. Usually, the processor itself doesn't perform this loading operation, instead, it generates an interrupt known as *page fault* and makes the kernel deal with it. A page fault tells the kernel that one of the processes tried to access an not-loaded page, so it needs to be loaded. As you can see, page faults help in implementing demand paging, anytime a page needs to be loaded into the memory then a page fault will be generated.

With this mechanism that virtual memory uses to manage the memory, we can make a process to own a size of memory that is not even available on the system. For example, in x86 architecture with systems that employ virtual memory, each process thinks that it owns **4GB** of main memory, even if the system has only **2GB** of RAM. This is possible due to demand paging and page replacements. Of course, a large size being available for the process, makes it easier for the programmers to write their code.