Chapter 7: Memory Management in Theory and x86

Introduction

It's well-know to us right now that one of 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 , 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 same as the physical view of the memory. In this view, the memory is a big bunch of contiguous bytes and each bytes 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 539kernel 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` [1], each page can be loaded exactly into one page frame. To load process `A` into the

[1] 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 [2], 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 [3]. 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,

[2] In x86, this logical memory address is known as *linear memory address* as we have discussed earlier in chapter

[3] In 32-bit x86 architecture, the length of memory address is `4 bytes = 32 bits`, that is, `2^32 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 and entry on another page table that contains the 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.