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.

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 [4], 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.

[4] 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.

Paging in x86

In x86, unlike segmentation which is enabled by default, paging is disabled by default, also, paging is not available on real mode, in 32-bit environment it can only be used in protected-mode [5]. If paging is intended to be used, the kernel should switch to protected-mode first, then, enables paging through a special register in x86 known as `CR0` which is one of *control registers* of x86 architecture . The last bit of `CR0` is the one that decides if paging is enabled, when is value is `1`, or disabled when its value is `0`. There are three *paging modes* in x86 a kernelist can chooses from, the difference between these three modes is basically related to the sizes of memory addresses and the available sizes of a page. These modes are *32-bit paging*, *PAE paging* [6] and *4-level paging* which is available for `64-bit` environment only. Beside to the last bit `CR0`, there are another two bits that can be used to decided the current paging mode. The first one is known as *PAE bit* which is the fifth bit of the control register `CR4`, when the value of this bit is `1` that means PEA mode is enabled, while `0` mean otherwise. The second bit is known is `LME` in a register known as `IA32_EFER`, setting value of this register to `1` makes the processor to switch from the protected-mode (`32-bit` environment) to the long-mode (`64-bit` environment) and when the value of `PAE` bit is `1`, then 4-level mode will be enabled . In our discussions, we are going to focus on `32-bit` paging mode which is the most basic one that is available for `32-bit`

[5] In 64-bit architecture paging is available in both protected-mode and long-mode.

[6] PAE stands for "Physical Address Extension"

environment. In this mode, there are two available sizes for a page 4KB and 4MB, also, 4GB of memory is addressable in this mode.

Linear Memory Address

Previously, we have discussed a part of the translation process of memory addresses in x86. To sum what we have already discussed up, any memory address that is generated by an executing code in x86 is known as a logical address, which is not the real memory address that contains the required data. This logical address need to be translated to get the real address. The first step of this translation process is to use segment descriptors to translate a logical address to a linear address by using the mechanism that we have already mentioned in chapter . When paging is disabled, the resulted linear address will be the physical (real) address that can be sent to the main memory to get the required data. On the other hand, when paging is enabled, the linear address needs a further translation step to obtain the physical memory address by using paging mechanism. To be able to perform this step of translation, a page table is used with the parts that compose the linear address.

Figure shows the structure of a linear address and its parts. As you can see, the size of a linear address is 32-bit which is divided into three parts. The bits 22 to 31 represent a *page directory* entry, the bits 21 to 12 represent a page table entry and the bits 0 to 11 represent an offset that contains the required data within a page. For example, assume a linear address which is composed of the following: page directory entry x, page table entry y and offset z. That means that this linear address needs to read the offset z from a page that is represented by the entry y in the page table, and this page table is represented by the page directory entry x. As you can see here, unlike our previous discussion of page table, the one which is implemented in x86 is a two-level page table, the first level is known as *page directory* which is used to point to the second level which is a page table and each page table, as we know, points to a page frame. The reason of using multi-level page tables is to save some memory since the size of page tables tend to have relatively large sizes in modern architecture and given that each process needs its own page table, then, its better to use multi-level page table which allows us to load just the needed parts of a page table (in a way similar to paging) instead of loading the whole page table into the memory.

Page Directory

The page directory in x86 can hold up to 1024 entries. Each entry points to a page table and each one of those page tables can hold up to 1024 entries which represent a process's pages. In other words, we can say that, for each process, there are more than one page table, and each one of those page tables is loaded in a different part of the main memory and the page directory of the process helps us in locating the page tables of a process.

As we have mentioned before, the page directory is the first level of x86's page table and each process has its own page directory. How the processor can find the current page directory, that is, the page directory of the current process? This can be done by using the register `CR3` which stores the base physical memory address of the current page directory. The first part of a linear address is an offset within the page directory, when an addition operation is performed between the first part of a linear address and the value on `CR3` the result is the base memory address of the entry that contains a page table that contains the page the has the required data.

The Structure of a Page Directory Entry

The size of an entry in the page directory is `4` bytes (`32` bits) and its structure is shown in the figure . The bits from `12` to `31` contain the physical memory address of the page table that this entry represent. As mentioned earlier, the reason of using a multi-level page table is to save some space in the main memory, and that's can be performed in a way same as demand paging. Not all page tables that a page directory points to are loaded into the main memory, instead, only the needed page tables, the rest are stored in a secondary storage until they are needed then they should be loaded. To be able to implement this mechanism, we need some place to store a flag that tells us whether the page table in question is loaded into the main memory or not, and that's exactly the job of bit `0` of a page directory entry, this bit is known as *present bit*, when its value is `1` that means the page table exists in the main memory, while the value `0` means otherwise. When an executing code tries to read a content from a page frame that its page table is not in the memory, the processor generates a page fault that tells the kernel to load this page table because it is needed right now.

When we have discussed segment descriptors, we have witnessed some bits that aim to provide additional protection for a segment. Paging in x86 also has bits that help in providing additional protection. Bit `1` in a page directory entry decides whether the page table that the entry points to is read-only when its value is `0` or if its writable `1` . Bit `2` decides whether the page table that this entry points to is restricted to privilege level code, that is, the code that runs on privilege level `0`, `1` and `2` when its value is `0` or that the page table is also accessible by a non-privileged code, that is, the code that runs on privilege level `3`.

Generally in computing, *caching* is a well-known technique. When caching is employed in a system, some data are fetched from a source and stored in a place faster to reach if compared to the source, these stored data are known as *cache*. The goal of caching is to make a frequently accessed data faster to obtain. Think of you web browser as an example of caching, when use visit a page [7] in a website, the web browser fetches the images of that page from the source (the server of the website) and stores it in your own machine's storage device which

[7] Please do not confuse a web page with a process page in this example.

is definitely too much faster to access if compared to a web server, when you visit the same website later, and the web browser encounters an image to be shown, it searches if its cached, if so, this is known as *cache hit*, the image will be obtained from your storage device instead of the web server, if the image is not cached, this is known as *cache miss*, the image will be obtained from the web server to be shown and cached. The processor is not an exception, it also uses cache to make things faster. As you may noticed, the entries of page directories and page tables needed to be frequently accessed, in the code of software a lot of memory accesses happen and with each memory access by the executing code both page directory and pages tables need to be accessed. With this huge number of accesses to page table and given the fact that the main memory is too much slower than the processor, then some caching is needed, and that exactly what is done in x86, a part of the page directory and page tables are cached in an internal, small and fast memory inside the processor , each time an entry of page table of directory is needed this memory is checked first, if the needed entry is on it, that is, we got a cache hit, then it will be used. In x86 paging, caching is controllable, say that for some reason, you wish to disable caching for a given entry, that can be done with bit `4` in a page directory entry. When the value of this bit is `1`, then the page table that is represented by this entry will not be cached by the processor, but the value `0` in this bit means otherwise. Unlike web browsers, the cache of a page table can be written to, for example, assume that page table `x` has been cached after using it in the first time and there is a page in this page table, call it `y`, that isn't loaded into the memory. We decided to load the page `y` which means present bit of the entry that represents this page should be changed in the page table `x`. To make things faster, instead of writing the changes to the page table `x` in the main memory (the source), these changes will be written to the cache which makes a difference between the cached data and the source data. This inconsistency between the two places of the data should be resolved somehow, the obvious thing to do is to write these changes later on also on the source. In caching context, the timing of writing the changes to the source is known as *write policy* and there are two well-known policies, the first one is known as *write-through*, in this policy, once the new data is written on the cache and the source at same time, once the writing process finishes on the cache we don't need to wait the writing process to the source to complete , the second policy is known as *write-back*, in which the writing process is performed only on the cache, while writing the changes on the source is performed later, when, for example, we decide to clear the cache. Bit `4` of the page directory entry decides which write policy will be used for the cached data, the value `1` means write-through policy will be used, while the value `0` means write-back policy will be used.

As in segment descriptors, when a page table which is referred by a given page directory entry is accessed, there is a bit in the directory entry known as *access bit* which is the fifth bit in the entry. The processor sets the value `1` automatically when the page table is accessed. Setting the value to `0` for any reason is the responsibility of the kernel.

We have said earlier that 32-bit paging in x86 provides us with two possible options for the size of a page, either 4KB page or 4MB page. The bit 7 in a page directory entry decides the size of the pages, when its value is 0 then the page size will be 4KB while the value 1 means that the page size is 4MB. There is a major difference between the two options. When the size of the page is 4MB, the page table will be a normal one-level page table, which means that the page directory will not refer to a page table anymore, but it is going to refer to a page frame. When the size of the page is 4KB, the two-level hierarchy will be employed. That makes sense, the number of entries that are needed to represent 4KB pages are way more than the number of entries that are needed to represent 4MB pages. However, in our discussion, we have focused (and will focus) on the case of 4KB pages. Finally, the bits 6, 8, 9, 10 and11 in the page directory entry are ignored.

Page Table

In 4KB pages environment, a page table is referred to by an entry in the page directory. As mentioned earlier, each page table can hold 1024 entries. After finding the base memory address of the page table in question by consulting the page directory, this base memory address will be used with the second part of the linear address to figure out which page table entry should the processor consult to reach the required data in the physical memory. Of course, the most important information that a page table entry stores is the base physical memory address of the page frame, this memory address will be used with the third part of the linear address (offset) to get the final physical memory address. The entry of a page table is exactly same as the entry of a page directory, its size is 4 bytes. Though, there are some simple differences, the first difference is bit 7, which was used to decide the page size, is ignored in the entry of a page table. The second difference is in bit 6, which was ignored in the entry of page directory. In our previous discussion on virtual memory we know that at some point of time, a victim frame may be chosen. This frame is removed from the main memory to free up some space for another page that we need to load from the disk. When the victim frame is removed from the main memory, its content should be written to the disk since it content may have been changed while it was loaded into the memory. Writing the content of the victim frame to the disk and loading the new page also from disk, given that the disk is really too slow compared to the processor, is going to cause some performance penalty. To make the matter a little bit better, we should write the content of the victim frame only if there is a real change in its content compared to the version which is already stored in the disk. If the victim frame version which is on the main memory and the version on the disk are identical, there is no need to waste valuable resource on writing the same content on this disk. A bit known as *dirty bit* is used to indicate whether the content of the memory has been changed and has differences with the disk version (when the value of the bit 1) or the two versions are identical (value 0). Bit 6 of a page table entry is the dirty bit of a given page.