Assignment 6

> 12011702 张镇涛

Q1

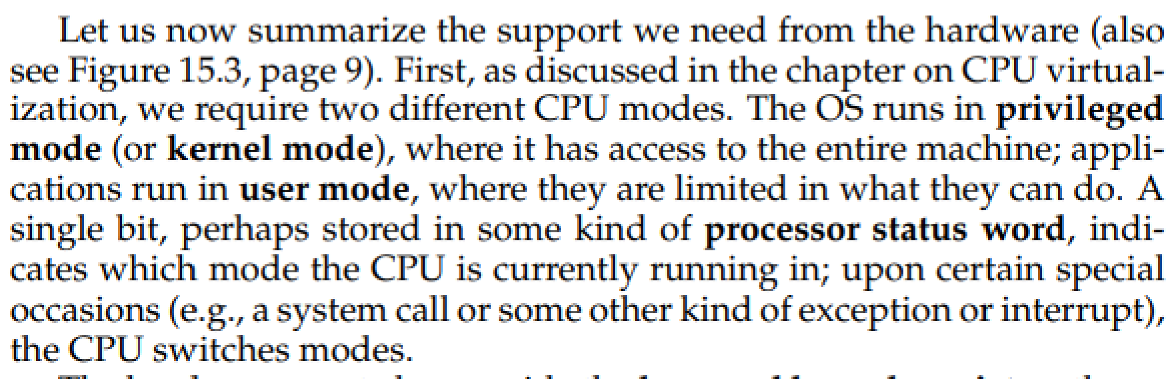
Hardware provides transformations for each memory access (e.g., an instruction fetch, load, or store), changing the virtual address provided by the instruction to a physical address.

OS must get involved at key points to set up the hardware so that the correct translations take place.

Their responsibilities are listed as follows:

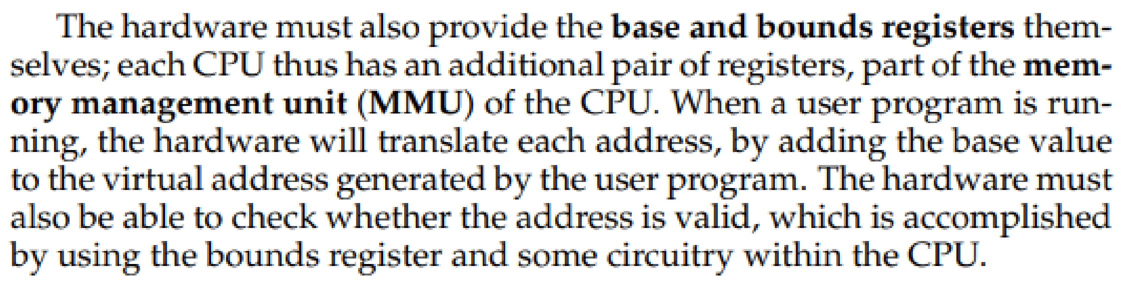
Hardware

1. Hardware provides two CPU modes: kernel mode and user mode. OS is in kernel mode, and user application is in the user mode. Upon certain occasions, CPU switches mode.



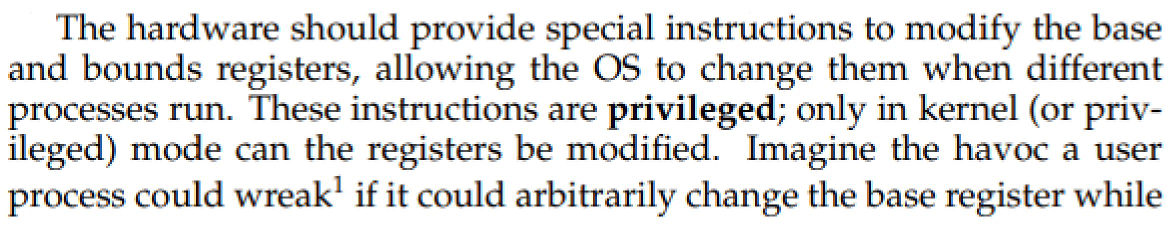
Let us now summarize the support we need from the hardware (also see Figure 15.3, page 9). First, as discussed in the chapter on CPU virtualization, we require two different CPU modes. The OS runs in **privileged mode** (or **kernel mode**), where it has access to the entire machine; applications run in **user mode**, where they are limited in what they can do. A single bit, perhaps stored in some kind of **processor status word**, indicates which mode the CPU is currently running in; upon certain special occasions (e.g., a system call or some other kind of exception or interrupt), the CPU switches modes.

2. Hardware provide base and bound register and it will translate each address upon execution of user program. It will also check validity of address.



The hardware must also provide the **base and bounds registers** themselves; each CPU thus has an additional pair of registers, part of the **memory management unit** (**MMU**) of the CPU. When a user program is running, the hardware will translate each address, by adding the base value to the virtual address generated by the user program. The hardware must also be able to check whether the address is valid, which is accomplished by using the bounds register and some circuitry within the CPU.

3. Hardware provides special privileged instructions to modify base and bound register which allows OS to change them when different processes run.



The hardware should provide special instructions to modify the base and bounds registers, allowing the OS to change them when different processes run. These instructions are **privileged**; only in kernel (or privileged) mode can the registers be modified. Imagine the havoc a user process could wreak[1] if it could arbitrarily change the base register while

4. CPU is able to generate exceptions when illegal memory access request occurs and it will stop execution, arranging exception handler on OS to run.

Finally, the CPU must be able to generate **exceptions** in situations where a user program tries to access memory illegally (with an address that is "out of bounds"); in this case, the CPU should stop executing the user program and arrange for the OS "out-of-bounds" **exception handler** to run. The OS handler can then figure out how to react, in this case likely terminating the process. Similarly, if a user program tries to change the values of the (privileged) base and bounds registers, the CPU should raise an exception and run the "tried to execute a privileged operation while in user mode" handler. The CPU also must provide a method to inform it of the location of these handlers; a few more privileged instructions are thus needed.

Operating System

1. OS takes action on process creation, finding space for its address space in memory. It maintains a data structure to find room for new address space.

First, the OS must take action when a process is created, finding space for its address space in memory. Fortunately, given our assumptions that each address space is (a) smaller than the size of physical memory and (b) the same size, this is quite easy for the OS; it can simply view physical memory as an array of slots, and track whether each one is free or in use. When a new process is created, the OS will have to search a data structure (often called a **free list**) to find room for the new address space and then mark it used. With variable-sized address spaces, life is more complicated, but we will leave that concern for future chapters.

2. OS reclaiming all of the terminated process's memory for use in other processes or OS. It puts this memory back on the free list and clean up related data structures.

Second, the OS must do some work when a process is terminated (i.e., when it exits gracefully, or is forcefully killed because it misbehaved), reclaiming all of its memory for use in other processes or the OS. Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.

3. OS needs to perform save and restoration of base-and-bounds pair when it switch between processes.

Third, the OS must also perform a few additional steps when a context switch occurs. There is only one base and bounds register pair on each CPU, after all, and their values differ for each running program, as each program is loaded at a different physical address in memory. Thus, the OS must *save and restore* the base-and-bounds pair when it switches between processes. Specifically, when the OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the **process structure** or **process control block** (PCB). Similarly, when the OS resumes a running process (or runs it the first time), it must set the values of the base and bounds on the CPU to the correct values for this process.

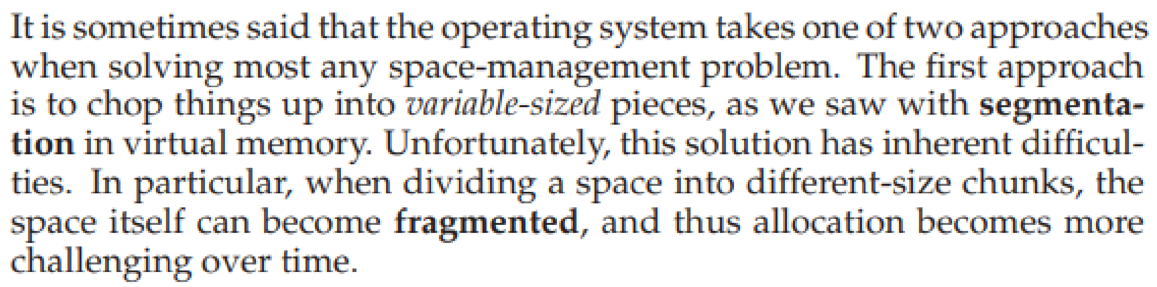
Q2

Comparisons will be given in following aspects:

<u>Size of chunks</u>

Segmentation

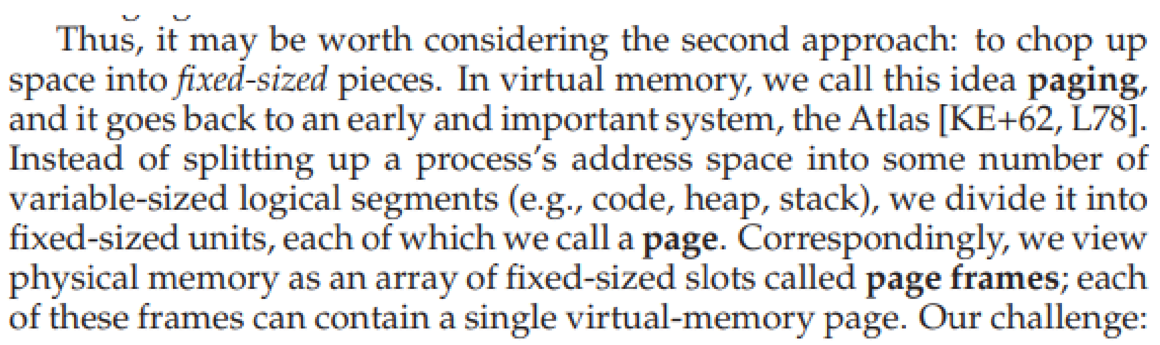
For segmentation, the size of chunk is **variable-sized**.



It is sometimes said that the operating system takes one of two approaches when solving most any space-management problem. The first approach is to chop things up into *variable-sized* pieces, as we saw with **segmentation** in virtual memory. Unfortunately, this solution has inherent difficulties. In particular, when dividing a space into different-size chunks, the space itself can become **fragmented**, and thus allocation becomes more challenging over time.

Paging

For paging, however, the size of chunk is **fixed**.

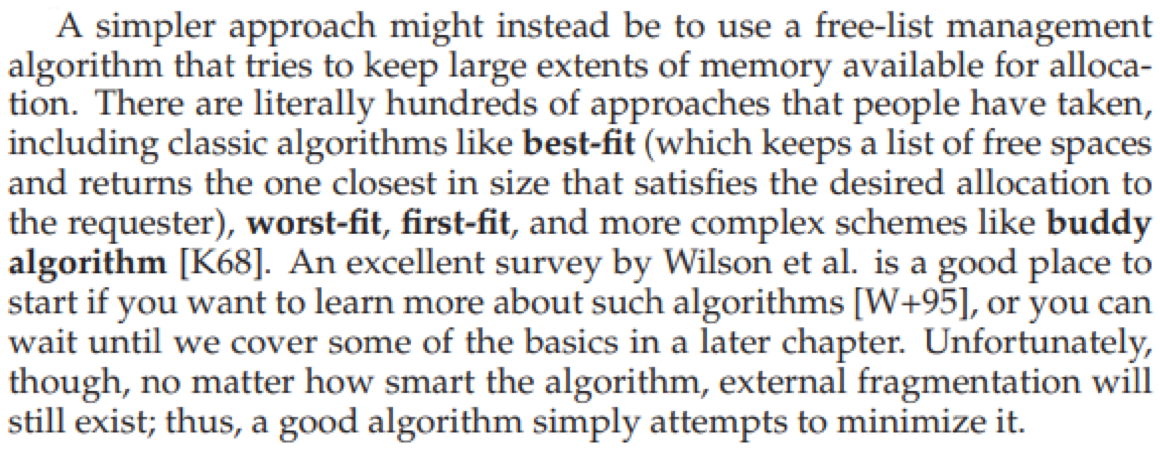


Thus, it may be worth considering the second approach: to chop up space into *fixed-sized* pieces. In virtual memory, we call this idea **paging**, and it goes back to an early and important system, the Atlas [KE+62, L78]. Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a **page**. Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**; each of these frames can contain a single virtual-memory page. Our challenge:

<u>Management of free space</u>

Segmentation

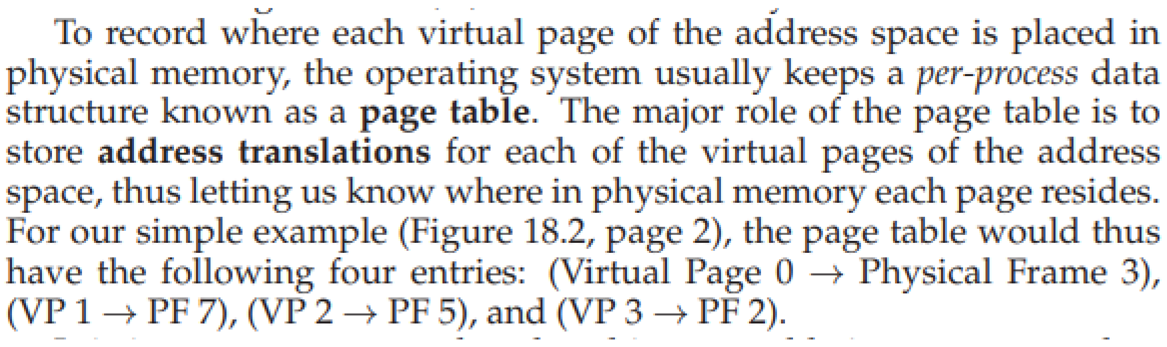
For segmentation, it uses a **free-list management algorithm**. It includes classic algorithm like best-fit, worst-fit, forst-fit, etc.



A simpler approach might instead be to use a free-list management algorithm that tries to keep large extents of memory available for allocation. There are literally hundreds of approaches that people have taken, including classic algorithms like **best-fit** (which keeps a list of free spaces and returns the one closest in size that satisfies the desired allocation to the requester), **worst-fit**, **first-fit**, and more complex schemes like **buddy algorithm** [K68]. An excellent survey by Wilson et al. is a good place to start if you want to learn more about such algorithms [W+95], or you can wait until we cover some of the basics in a later chapter. Unfortunately, though, no matter how smart the algorithm, external fragmentation will still exist; thus, a good algorithm simply attempts to minimize it.

Paging

For paging, it uses a per-process data structure called page table. It stores address translation for each of virtual pages of address space.



To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a *per-process* data structure known as a **page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page resides. For our simple example (Figure 18.2, page 2), the page table would thus have the following four entries: (Virtual Page 0 $\rightarrow$ Physical Frame 3), (VP 1 $\rightarrow$ PF 7), (VP 2 $\rightarrow$ PF 5), and (VP 3 $\rightarrow$ PF 2).

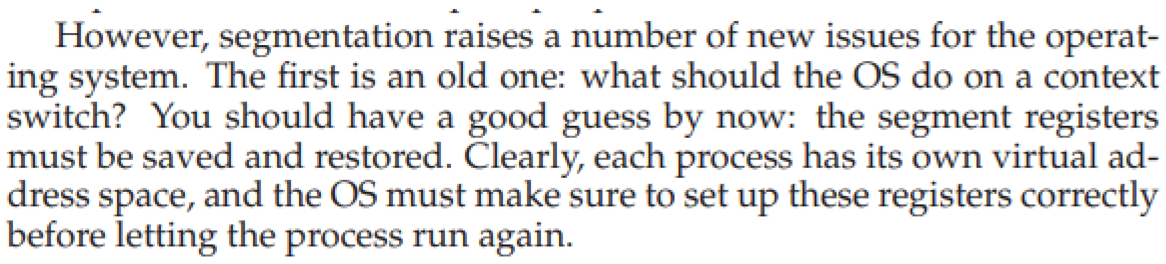
Context Switch Overhead

Segmentation

Context switch overhead is relatively high.

OS needs to:

1. save and restore segment registers correctly.
2. setup process's own virtual address space .
3. start the process to be switched.

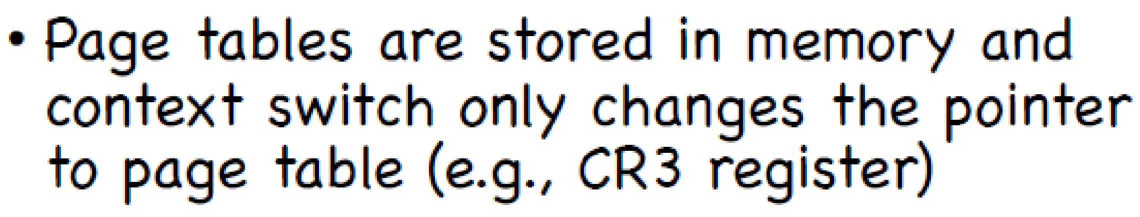


However, segmentation raises a number of new issues for the operating system. The first is an old one: what should the OS do on a context switch? You should have a good guess by now: the segment registers must be saved and restored. Clearly, each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again.

Paging

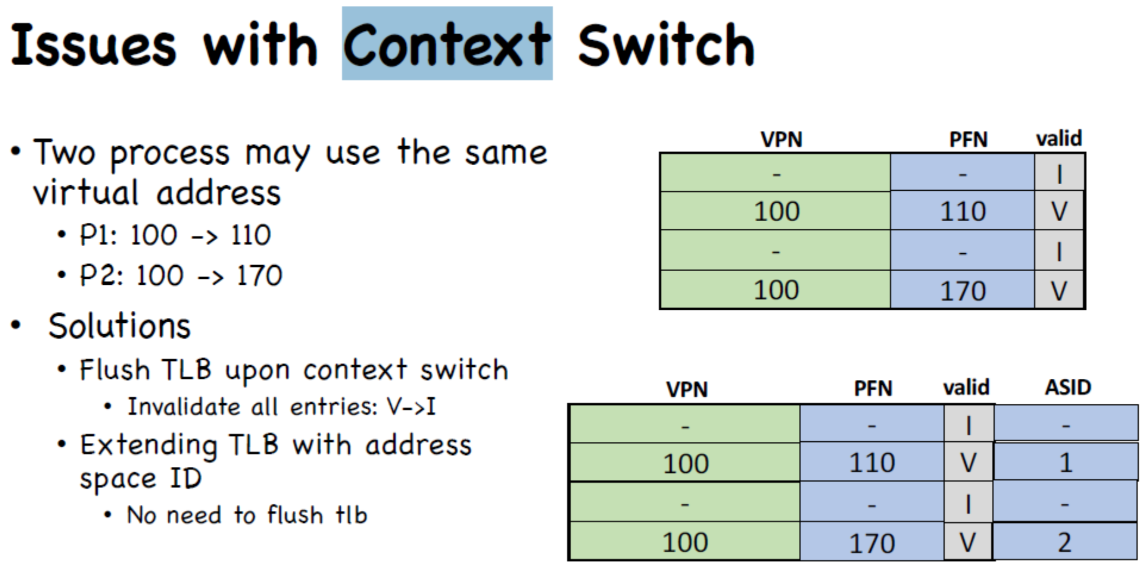
Context switch overhead is relatively low.

Based on lecture slides, for paging, upon context switch, it only needs to change the pointer to page table.



- Page tables are stored in memory and context switch only changes the pointer to page table (e.g., CR3 register)

Also, TLB needs to be updated based on strategy used. It may invalidate all entries or extend with address space ID to identify same virtual address of different processes.


Issues with Context Switch

- Two process may use the same virtual address
  - P1: 100 -> 110
  - P2: 100 -> 170
- Solutions
  - Flush TLB upon context switch
    - Invalidate all entries: V->I
  - Extending TLB with address space ID
    - No need to flush tlb

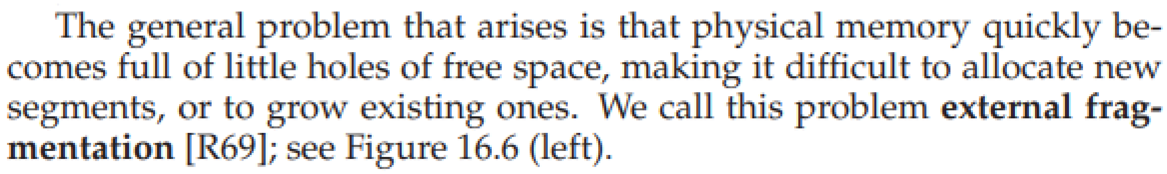
VPN	PFN	valid
-	-	I
100	110	V
-	-	I
100	170	V

VPN	PFN	valid	ASID
-	-	I	-
100	110	V	1
-	-	I	-
100	170	V	2

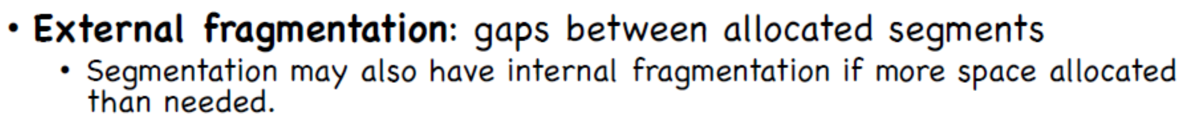
fragmentation

Segmentation

For segmentation, external fragmentation problem is easy to happen.

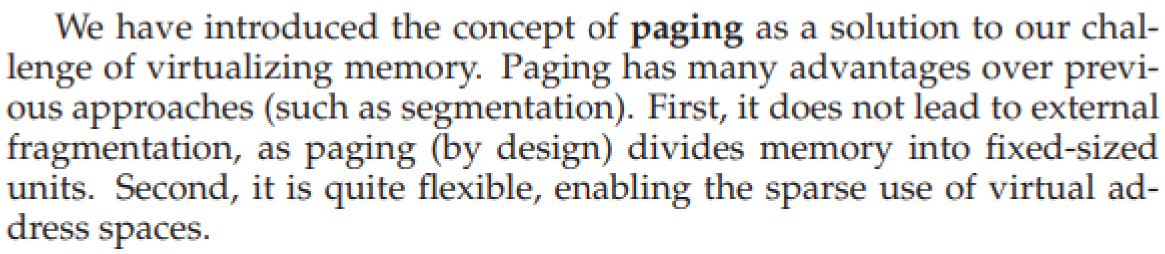

The general problem that arises is that physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones. We call this problem **external fragmentation** [R69]; see Figure 16.6 (left).

And also based on lecture slides, internal fragmentation may exists as well.


- **External fragmentation**: gaps between allocated segments
  - Segmentation may also have internal fragmentation if more space allocated than needed.

Paging

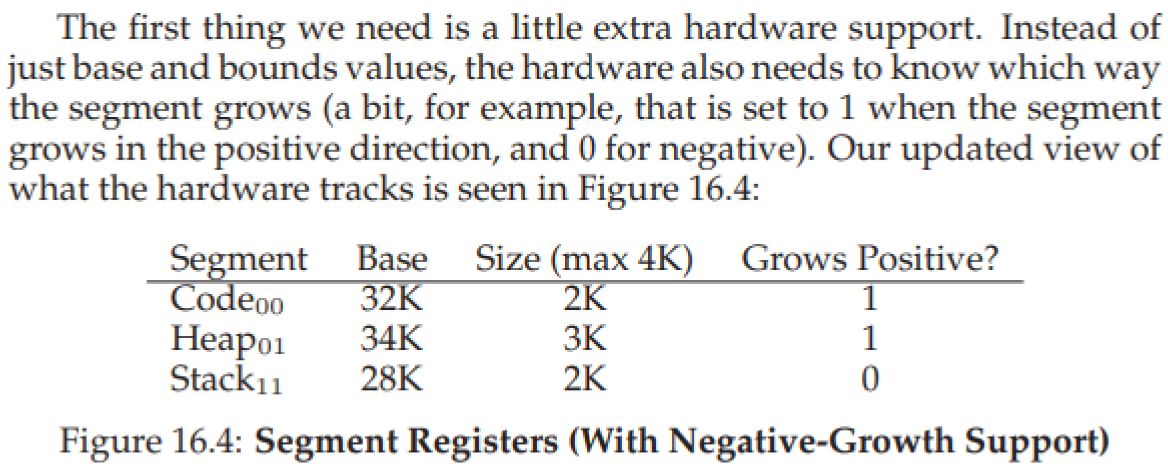
Paging divides memory into fixed-sized units so that it won't lead to external fragmentation.


We have introduced the concept of **paging** as a solution to our challenge of virtualizing memory. Paging has many advantages over previous approaches (such as segmentation). First, it does not lead to external fragmentation, as paging (by design) divides memory into fixed-sized units. Second, it is quite flexible, enabling the sparse use of virtual address spaces.

Status bits and protection bits

Segmentation

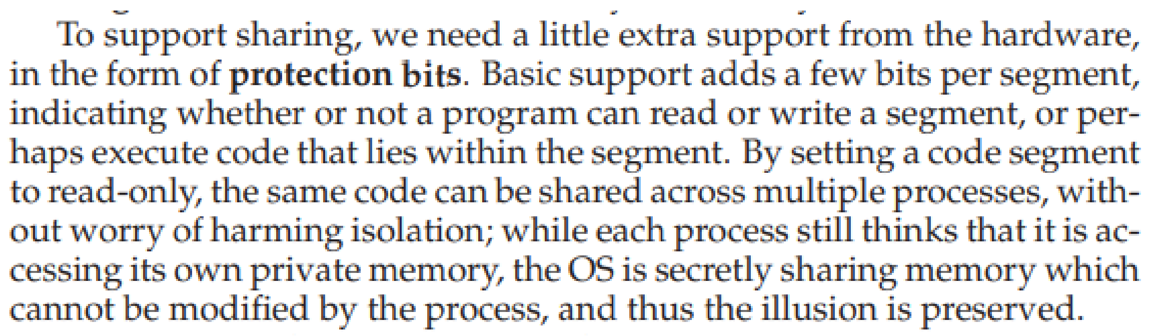
Hardware needs to maintain **a bit to represent which way the segment grows**.


The first thing we need is a little extra hardware support. Instead of just base and bounds values, the hardware also needs to know which way the segment grows (a bit, for example, that is set to 1 when the segment grows in the positive direction, and 0 for negative). Our updated view of what the hardware tracks is seen in Figure 16.4:

Segment	Base	Size (max 4K)	Grows Positive?
Code$_{00}$	32K	2K	1
Heap$_{01}$	34K	3K	1
Stack$_{11}$	28K	2K	0

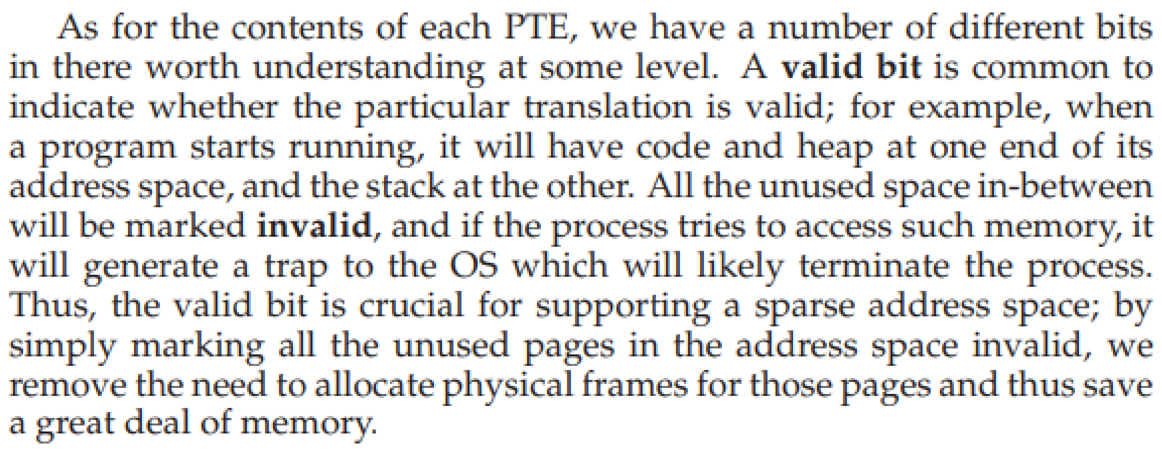
Figure 16.4: **Segment Registers (With Negative-Growth Support)**

Also, several **protection bits** is needed to deal with sharing. It indicates whether or not a program can read or write a segment, or perhaps execute code that lie within the segment.

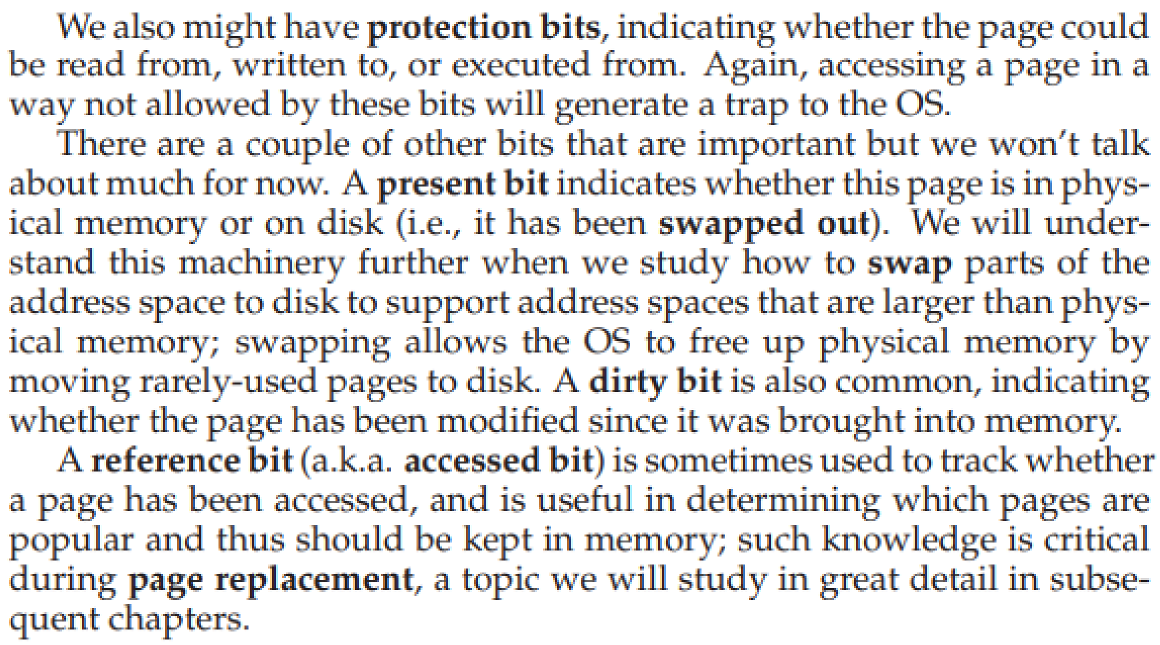

To support sharing, we need a little extra support from the hardware, in the form of **protection bits**. Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment. By setting a code segment to read-only, the same code can be shared across multiple processes, without worry of harming isolation; while each process still thinks that it is accessing its own private memory, the OS is secretly sharing memory which cannot be modified by the process, and thus the illusion is preserved.

Paging

In each page table entry, it includes a **valid bit**. When a process tries to access a invalid address, it will generate a trap to OS which will likely to terminate the process.


As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is valid; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process. Thus, the valid bit is crucial for supporting a sparse address space; by simply marking all the unused pages in the address space invalid, we remove the need to allocate physical frames for those pages and thus save a great deal of memory.

Also, PTEs also have protection bits (whether the page could be read from, written to, etc.), present bit (whether the page is in physical memory or on disk), dirty bit (whether the page has been modified since it was brought to memory), reference bit (track whethet a page has been accessed), which serves for different functionality.

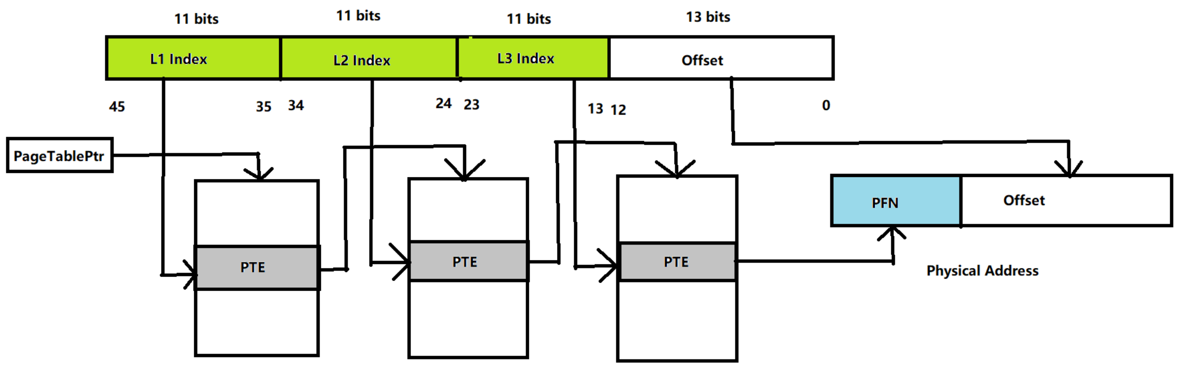


We also might have **protection bits**, indicating whether the page could be read from, written to, or executed from. Again, accessing a page in a way not allowed by these bits will generate a trap to the OS.

There are a couple of other bits that are important but we won't talk about much for now. A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**). We will understand this machinery further when we study how to **swap** parts of the address space to disk to support address spaces that are larger than physical memory; swapping allows the OS to free up physical memory by moving rarely-used pages to disk. A **dirty bit** is also common, indicating whether the page has been modified since it was brought into memory.

A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed, and is useful in determining which pages are popular and thus should be kept in memory; such knowledge is critical during **page replacement**, a topic we will study in great detail in subsequent chapters.

Q3



Page size = 8KB = 2^{13} bytes => Offset = 13 bits

And also, since page table entry size is 4 bytes. we can obtain that # Page entry = $2^{13}/4 = 2^{11}$.

Therefore, page can accommodate 2^{11} entries, which requires 11 binary digits.

33 can be divided to three 11 digits, so that **3 levels of page tables** would be required to map the entire virtual address space.

Q4

(a)

Page size= 2^{12} bytes= 4KB.

maximum number of page table entry = 2^{20}. And size of a page table entry is 4bytes, so maximum page table size=$2^{20} \times 4 = 4$MB.

(b)

0xC302C302 = 1100001100 0000101100 001100000010

So as we can see the **1-st level page number** is 1100001100, which is **780** in decimal.

And **offset** is 001100000010, which is **770** in decimal.

0xEC6666AB = 1110110001 1001100110 011010101011

So as we can see the **2-nd level page number** is 1001100110, which is **614** in decimal.

And **offset** is 011010101011, which is **1707** in decimal.