Chapter 7 Memory Management Techniques
(Revision number 20)

Let us review what we have seen so far. On the hardware side, we have looked at the instruction-set of the processor, interrupts, and designing a processor. On the software side, we have seen how to use the processor as a resource and schedule it to run different programs. The software entities we have familiarized ourselves with include the *compiler* and *linker* that live above the operating system. We have also familiarized ourselves the *loader* and *process scheduler* that are part of the operating system.

By now, we hope we have de-mystified some of the magic of what is inside "a box." In this chapter, we continue unraveling the box by looking at another important component of the computer system, namely, the memory.

More than any other subsystem, memory system brings out the strong inter-relationship between hardware and system software. Quite often, it is almost impossible to describe some software aspect of the memory system without mentioning the hardware support. We cover the memory system in three chapters including this one. This chapter focuses on different strategies for memory management by the operating system along with the necessary architectural support. In Chapter 8, we delve into the finer details of page-based memory system, in particular, page replacement policies. Finally, in Chapter 9, we discuss memory hierarchy, in particular, cache memories and main or physical memory.

7.1 Functionalities provided by a memory manager

Let us understand what we mean by memory management. As a point of differentiation, we would like to disambiguate memory management as we mean and discuss in this textbook from the concept of "automatic memory management" that is used by programming languages such as Java and C#. The runtime systems of such languages automatically free up memory not currently used by the program. *Garbage Collection (GC)* is another term used to describe this functionality. Some of the technical issues with GC are similar to those handled by the memory management component of an operating system. Discussion of the similarities and differences between the two are outside the scope of this textbook. The interested reader is referred to other sources to learn more about GC[1].

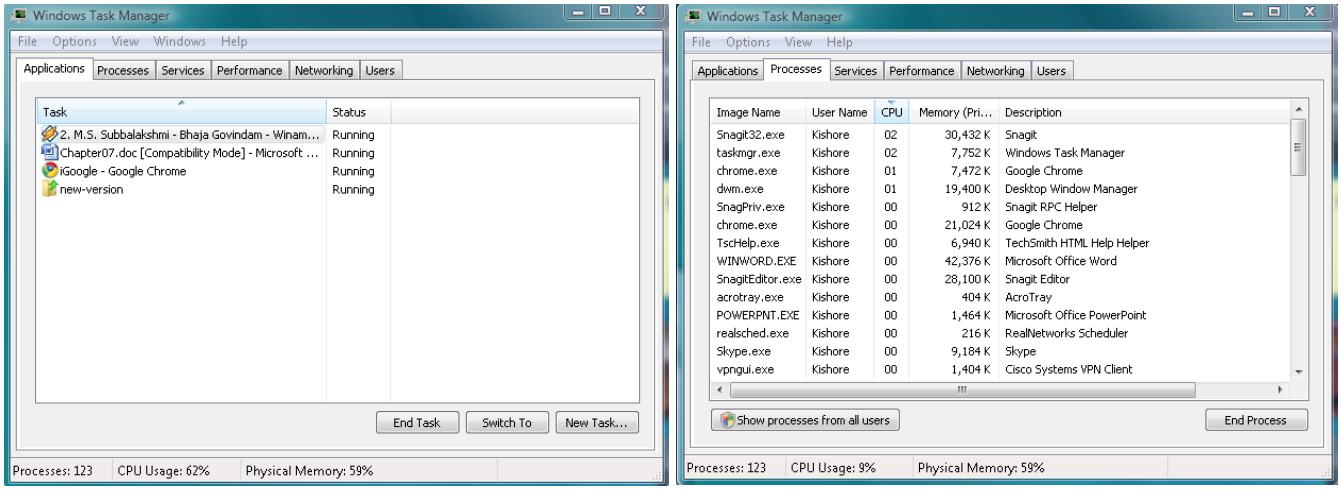
In this textbook, we focus on how the operating system manages memory. Just like the processor, memory is a precious resource and the operating system ensures the best use of this resource. Memory management is an operating system entity that provides the following functionalities.

1. Improved resource utilization

Since the memory is a precious resource, it is best to allocate it on demand. The analogy

[1] Richard Jones, Garbage Collection Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons.

is the use of office space. Each faculty member in the department may ask for a certain amount of space for his/her students and lab. The chair of the department may not give all the space requested but do it incrementally as the faculty member's group grows. Similarly, even though the memory footprint of the program includes the heap (Figure 6.1 in Chapter 6) there is no need to allocate this to a program at startup. It is sufficient to make the allocation if and when the program dynamically requests it. If a faculty member's group shrinks, he/she does not need all the space allocated any more. The chair of the department would reclaim that space and reallocate it to someone else who needs it more. In the same manner, if a process is not actively using memory allocated to it, then perhaps it is best to release it from that process and use it for some of other process that needs it. Both these ideas, incremental allocation and dynamic reallocation, will lead to improved resource utilization of memory. The criticality of using this resource judiciously is best understood by showing a simple visual example. The left half of the following figure shows a screenshot of the task manager listing the actual applications running on a laptop. The right half of the same figure shows a partial listing of the actual processes running on the laptop. There are 4 applications running but there are 123 processes! The purpose for showing this screenshot is to get the point across that in addition to the user processes, the operating system and other utilities spawn a number of background processes (often called *daemon* processes). Thus, the demand placed on the memory system by the collection of processes running at any point of time on the computer is quite intense. The laptop shown in this example has 2 GB of memory out of which 58% is in use despite the fact that only 4 applications are running at this point of time.



2. Independence and Protection

We mentioned that several processes are co-resident in memory at any point of time. A program may have a bug causing it to run amok writing into areas of memory that is not part of its footprint. It would be prudent to protect a buggy process from itself and from other processes. Further, in these days of computer viruses and worms, a malicious program could be intentionally trying to corrupt the memories of other genuine programs. Therefore, the memory manager provides *independence* for each process, and *memory protection* from one another. Once again using the space analogy, you probably can

relate to the need for independence and protection if you grew up in a home with siblings. You either had or wished you had a room for yourself that you can lock when you want to keep the siblings (and the parents) out!

3. Liberation from resource limitations

When a faculty member is hired, the chair of the department would promise as much space as he/she wants, perhaps even larger than the total space available in the department. This gives the faculty member to think "big" in terms of establishing and growing his/her research agenda and the student base. In a similar manner, it is ideal for the programmer not to worry about the amount of physical memory while developing his/her program. Imagine a faculty member has 10 students but has lab space only for five. The students may have to work different shifts sharing the same office space. In the same vein, imagine having to write a multi-player video game program, and your manager says you have a total memory space of 10Kbytes. You, as the programmer, will have to think of identifying data structures you will not need at the same time and use the same memory space for storing them (in essence, overlaying these data structures in memory). The program can get ugly if you have to resort to such methods. To address this problem, the memory manager and the architecture work together to devise mechanisms that gives the programmer an illusion of a large memory. The actual physical memory may be much smaller than what the programmer is given an illusion of.

4. Sharing of memory by concurrent processes

You may want to keep your siblings out most of the time but there are times when you want to be able to let them come into your room, perhaps to play a video game. Similarly, while memory protection among processes is necessary, sometimes processes may want to *share* memory either implicitly or explicitly. For example, you may have multiple browser windows open on your desktop. Each of them may be accessing a different web page, but if all of them are running the same browser application then they could share the code. This is an example of implicit sharing among processes. Copying and pasting an image from a presentation program into a word processor is an example of explicit data sharing. The memory manager facilitates memory sharing when needed among processes.

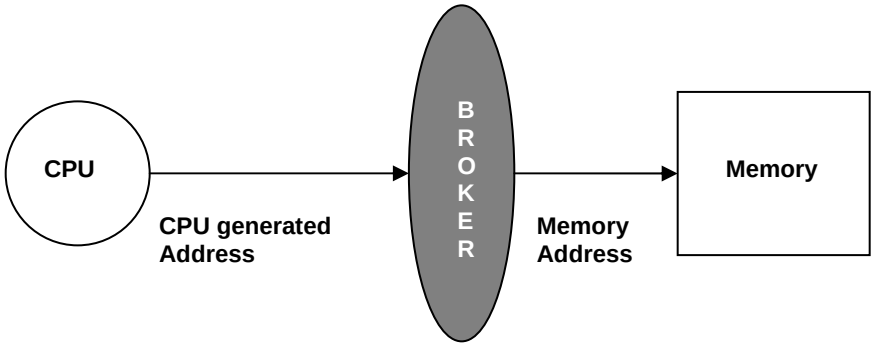


Figure 7.1: A Generic Schematic of a Memory manager

All of the above functionalities come at a price. In particular, we are introducing a *broker* between the CPU and the memory as shown in Figure 7.1. The broker is a piece

of hardware that provides the mechanisms needed for implementing the memory management policies. In principle, it maps a CPU generated (logical) memory address to the *real* memory address for accessing the memory. The sophistication of the broker depends on the functionalities provided by the memory manager.

So far, we have not introduced such functionality in LC-2200. However, as we get ambitious and let LC-2200 take over the world from game players to high-performance computers, such functionalities become indispensable.

The overall goals of a good memory manager are three-fold:
1. Require minimal hardware support
2. Keep the impact on memory accesses low
3. Keep the memory management overhead low (for allocation and de-allocation of memory)

7.2 Simple Schemes for Memory Management

In this section, let us consider some simple schemes for memory management and the corresponding hardware support. This section is written in the spirit of shared discovery to understand how to accomplish the goals enumerated above, while at the same time meeting the functionalities presented in the previous section. The first two schemes and the associated hardware support (fence register and bounds registers) are for illustration purposes only. We do not know of any machine architecture that ever used such schemes. The third scheme (base and limit registers) was widely used in a number of architectures including CDC 6600 (the very first supercomputer) and IBM 360 series. All three have limitations in terms of meeting the functionalities identified in the previous section, and thus set the stage for the sophisticated memory management schemes, namely, paging and segmentation, found in modern architectures.

1. Separation of user and kernel

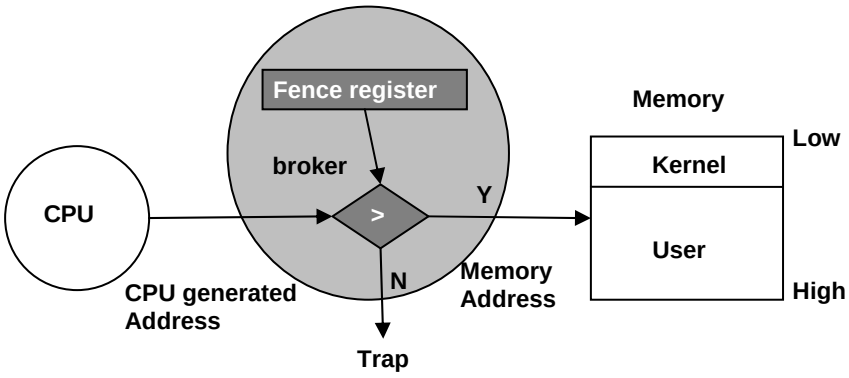


Figure 7.2: Fence Register

Let us consider a very simple memory management scheme. As we have seen before, the operating system and the user programs share the total available memory space. The space used by the operating system is the *kernel space*, and the space occupied by the user programs is the *user space*. As a first order of approximation, we wish to ensure that

there is a boundary between these two spaces, so that a user program does not straddle into the memory space of the operating system. Figure 7.2 shows a simple hardware scheme to accomplish this separation. The shaded area in the figure corresponds to the hardware portion of the work done by the broker in Figure 7.1. The scheme relies on three architectural elements: a *mode* bit that signifies whether the program is in user or kernel mode; a *privileged instruction* that allows flipping this bit; and a *fence* register. As you can imagine, the name for this register comes from the analogy of physical fences one might build for property protection. The memory manager sets the fence register when the user program is scheduled. The hardware validates the processor generated memory address by checking it against this fence register. This simple hardware scheme gives memory protection between the user program and the kernel. For example, let us assume that the fence register is set to 10000. This means that the kernel is occupying the memory region 0 through 10000. In user mode, if the CPU generates any address over 10000 then the hardware considers it a valid user program address. Anything less than or equal to 10000 is a kernel address and generates an access violation trap. The CPU has to be in kernel mode to access the kernel memory region. To understand how the CPU gets into the kernel mode, recall that in Chapter 4 we introduced the notion of *traps*, a synchronous program discontinuity usually caused by program wishing to make a system call (such as reading a file). Such traps result in the processor automatically entering the kernel mode as part of implementation of the trap instruction. Thus, the CPU implicitly gets into kernel mode on system calls and is now able to address the memory region reserved for the kernel. Once the operating system completes the system call, it can explicitly return to the user mode by using the architecture provided privileged instruction. It is a privileged instruction since its use is limited to kernel mode. Any attempt to execute this instruction in user mode will result in an *exception*, another synchronous program discontinuity (which we introduced in Chapter 4 as well) caused by an illegal action by a user program. As you may have already guessed, writing to the fence register is a privileged instruction as well.

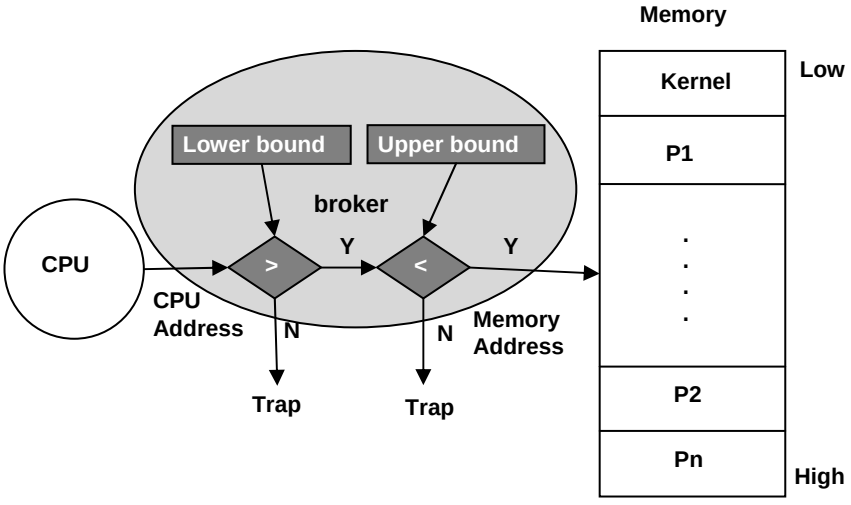


Figure 7.3: Bounds Registers

2. Static relocation

As previously discussed, we would like several user programs to be co-resident in memory at the same time. Therefore, the memory manager should protect the co-resident processes from one another. Figure 7.3 shows a hardware scheme to accomplish this protection. Once again, the shaded area in the figure corresponds to the hardware portion of the work done by the broker in Figure 7.1.

Static relocation refers to the memory bounds for a process being set at the time of linking the program and creating an executable file. Once the executable is created, the memory addresses cannot be changed during the execution of the program. To support memory protection for processes, the architecture provides two registers: *upper bound* and *lower bound*. The *bounds* registers are part of the process control block (PCB, which we introduced in Chapter 6). Writing to the bounds registers is a privileged instruction provided by the architecture. The memory manager sets the values from the PCB into the bounds registers at the time of dispatching the process (please refer to Chapter 6 on scheduling a process on the CPU). At link time, the linker allocates a particular region in memory for a particular process[2]. The loader fixes the lower and upper bound register values for this process at load time and never changes them for the life of the program. Let us assume that the linker has assigned P1 the address range 10001 to 14000. In this case, the scheduler will set the lower bound and upper bound registers to 10000 and 14001, respectively. Thus while P1 is executing, if the CPU generates addresses in the range 10001 and 14000, the hardware permits them as valid memory accesses. Anything outside this range will result in an access violation trap.

A concept that we alluded to in Chapter 6 is *swapping*. "Swapping out" a process refers to the act of moving an inactive process (for example, if it is waiting on I/O completion) from the memory to the disk. The memory manager does this act to put the memory occupied by the inactive process to good use by assigning it to other active processes. Similarly, once a process becomes active again (for example, if its I/O request is complete) the memory manager would "swap in" the process from the disk to the memory (after making sure that the required memory is allocated to the process being swapped in).

Let us consider swapping a process in from the disk with static relocation support. Due to the fixed bounds, the memory manager brings a swapped out process back to memory into exactly the same spot as before. If that memory space is in use by a different process then the swapped out process cannot be brought back into memory just yet, which is the main limitation with static relocation.

In reality, compilers generate code assuming some well-known address where the program will reside in memory[3]. Thus, if the operating system cannot somehow rectify this assumption a process is essentially *non-relocatable*. We define a process as non-

[2] Note that modern operating systems use dynamic linking, which allows deferring this decision of binding the addresses to a process until the time of loading the process into memory. Such a dynamic linker could take the current usage of memory into account to assign the bounds for a new process.

[3] In most modern compilers, a program starts at address 0 and goes to some system-specified maximum address.

relocatable if the addresses in the program cannot be changed either during loading into memory or during execution. We define static relocation as the technique of locating a process at *load time* in a different region of memory than originally intended at compile time. That is, the addresses used in the program are bound (i.e., fixed) at the time the program is loaded into memory and do not change during execution. IBM used a version of this kind of static relocation in their early models of mainframes (in the early 1960's). At the time of loading a process into memory, the loader will look at unused space in memory and make a decision as to where to locate the new process. It will then "fix" up all the addresses in the executable so that the program will work correctly in its new home. For example, let us say the original program occupied addresses 0 to 1000. The loader decides to locate this program between addresses 15000 and 16000. In this case, the loader will add 15000 to each address it finds in the executable as it loads it into memory. As you can imagine, this is a very cumbersome and arduous process. The loader has an intimate knowledge of the layout of the executable so that it can tell the difference between constant values and addresses to do this kind of fix up.

3. Dynamic relocation

Static relocation constrains memory management and leads to poor memory utilization. This is because once created an executable occupies a fixed spot in memory. Two completely different programs that happen to have the same or overlapping memory bounds cannot co-exist in memory simultaneously even if there are other regions of memory currently unoccupied. This is similar to two kids wanting to play with the same toy though there are plenty of other toys to play with! This is not a desirable situation. *Dynamic relocation* refers to the ability to place an executable into any region of memory that can accommodate the memory needs of the process. Let us understand how this differs from static relocation. With dynamic relocation, the memory address generated by a program can be changed during the *execution* of the program. What this means is that, at the time of loading a program into memory, the operating system can decide where to place the program based on the current usage of memory. You might think that this is what a dynamic linker lets us do in the previous discussion on static relocation. However, the difference is, even if the process is swapped out, when it is later brought in, it need not come to the same spot it occupied previously. Succinctly put, with static relocation, addresses generated by the program are fixed during execution while they can be changed during execution with dynamic relocation.

Now, we need to figure out the architectural support for dynamic relocation. Let us try a slightly different hardware scheme as shown in Figure 7.4. As before, the shaded area in the figure corresponds to the hardware portion of the work done by the broker in Figure 7.1. The architecture provides two registers: *base* and *limit*. A CPU generated address is *always* shifted by the value in the base register. Since this shift necessarily happens during the execution of the program, such an architectural enhancement meets the criterion for dynamic relocation. As in the case of static relocation, these two registers are part of the PCB of every process. Every time a process is brought into memory (at either load time or swap time), the loader assigns the values for the base and bound registers for the process. The memory manager records these loader assigned values into the corresponding fields of the PCB for that process. Similar to the bounds registers for

static relocation, writing to the base and limit registers is a privileged instruction supported by the architecture. When the memory manager dispatches a particular process, it sets the values for the base and limit registers from the PCB for that process. Let us assume that P1's memory footprint is 4000. If the loader assigns the address range 10001 to 14000 for P1, then the memory manager sets the base register to 10001 and the limit register 14001 in the PCB for P1. Thus when P1 is executing, any CPU generated address is automatically shifted up by 10000 by the hardware. So long as the shifted address is less than the value set in the limit register, the hardware permits it as a valid memory access. Anything that is beyond the limit results in an access violation trap. The reader should feel convinced that dynamic relocation will result in better memory utilization than static relocation.

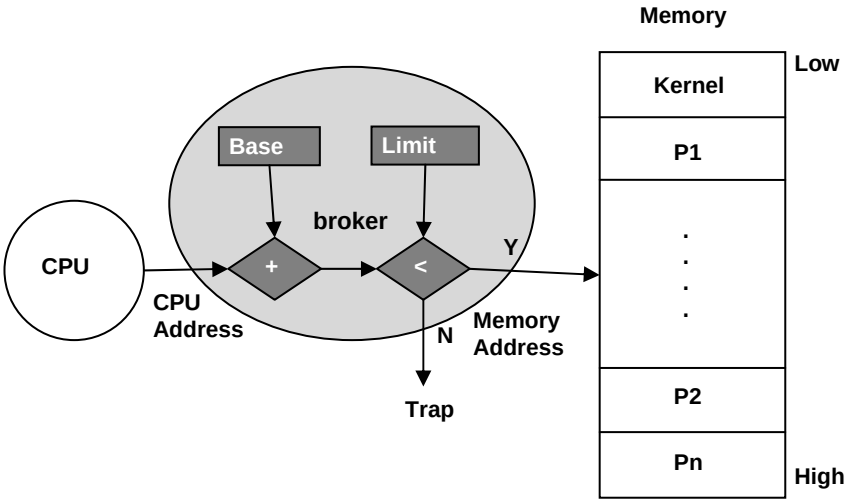


Figure 7.4: Base and Limit Registers

The architectural enhancements we presented for both static and dynamic relocation required two additional registers in the processor. Let us review these two schemes with respect to hardware implementation. Compare the datapath elements that you would need for Figures 7.3 and 7.4. For Figure 7.3, you need two comparison operations for bounds checking. For Figure 7.4, you need an addition followed by a comparison operation. Thus, in both schemes you will need two arithmetic operations to generate the memory address from the CPU address. Therefore, both schemes come out even in terms of hardware complexity and delays. However, the advantage one gets in terms of memory utilization is enormous with the base and limit register scheme. This is an example of how a little bit of human ingenuity can lead to enormous gains with little or no added cost.

7.3 Memory Allocation Schemes

We will assume using the *base plus limit register* scheme as the hardware support available to the memory manager and discuss some policies for memory allocation. In each case, we will identify the data structures needed to carry out the memory management.

7.3.1 Fixed Size Partitions

In this policy, the memory manager would divide the memory into fixed size partitions. Let us understand the data structure needed by the memory manager. Figure 7.5 shows a plausible data structure in the form of an allocation table kept in kernel space. In effect, the memory manager manages the portion of the memory that is available for use by user programs using this data structure. For this allocation policy, the table contains three fields as shown in Figure 7.5. The *occupied bit* signifies whether the partition is in use or not. The bit is 1 if the partition has been allocated; 0 otherwise. When a process requests memory (either at load time or during execution), it is given one of the fixed partitions that is equal to or greater than the current request. For example, if the memory manager has partitions of 1KB, 5KB and 8KB sizes, and if a process P1 requests a 6KB memory chunk, then it is given the 8KB partition. The memory manager sets the corresponding bit in the table to 1; and resets the bit when the process returns the chunk. Upon allocating P1's request for 6KB the allocation table looks as shown in Figure 7.5a. Note that there is wasted space of 2KB within this 8KB partition. Unfortunately, it is not possible to grant a request for a 2KB memory chunk from another process using this wasted space. This is because the allocation table maintains summary information based on fixed size partitions. This phenomenon, called *internal fragmentation*, refers to the wasted space internal to fixed size partitions that leads to poor memory utilization. In general, internal fragmentation is the difference between the granularity of memory allocation and the actual request for memory.

Internal fragmentation = Size of Fixed partition - Actual memory request (1)

Example 1:

A memory manager allocates memory in fixed chunks of 4 K bytes. What is the maximum internal fragmentation possible?

Answer:

The smallest request for memory from a process is 1 byte. The memory manager allocates 4 K bytes to satisfy this request.

So the maximum internal fragmentation possible

= 4K bytes - 1

= 4096 – 1 = 4095 bytes.

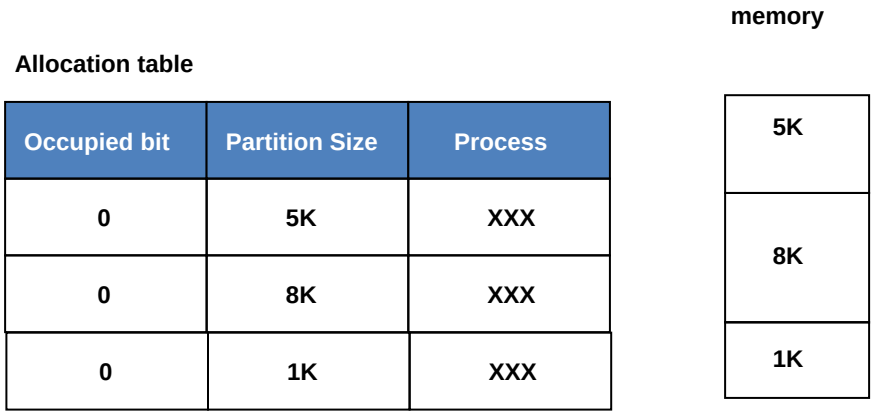


Figure 7.5: Allocation Table for fixed size partitions

Suppose there is another memory allocation request for 6KB while P1 has the 8KB partition. Once again, this request also cannot be satisfied. Even though cumulatively (between the 1KB and 5KB partitions) 6KB memory space is available, it is not possible to satisfy this new request since the two partitions are not contiguous (and a process's request is for a contiguous region of memory). This phenomenon, called *external fragmentation*, also results in poor memory utilization. In general, external fragmentation is the sum of all the non-contiguous memory chunks available to the memory system.

$$\text{External Fragmentation} = \sum \text{All non-contiguous memory partitions} \quad (2)$$

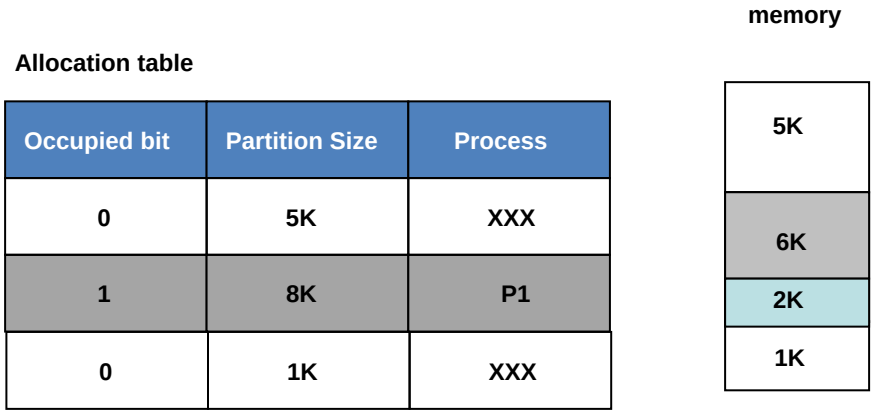


Figure 7.5a: Allocation Table after P1's request satisfied

7.3.2 Variable Size Partitions

To overcome the problem of internal fragmentation, we will discuss a memory manager that allocates variable sized partitions commensurate with the memory requests. Let us assume that 13KB is the total amount of memory that is available to the memory manager. Instead of having a static allocation table as in the previous scheme, the memory manager dynamically builds the table on the fly. Figure 7.6 shows the initial state of the allocation table before any allocation.

Figure 7.6: Allocation Table for Variable Size Partitions

Figure 7.6a shows the table after the memory manager grants a series of memory requests. Note that the manager has 2KB of free space left after the satisfying the requests of P1, P2, and P3.

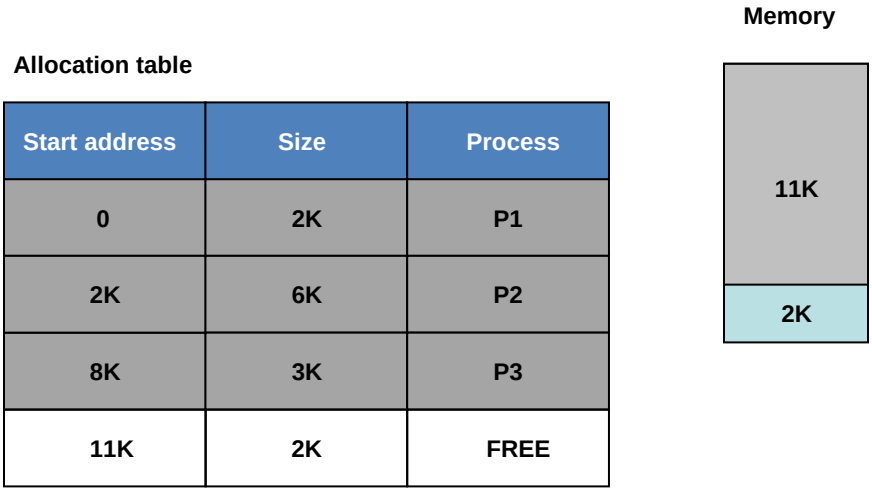


Figure 7.6a: State of the Allocation Table after a series of memory requests from P1 (2KB); P2 (6KB); and P3 (3KB)

Figure 7.6b shows the state upon P1's completion. The 2KB partition occupied by P1 is marked FREE as well.

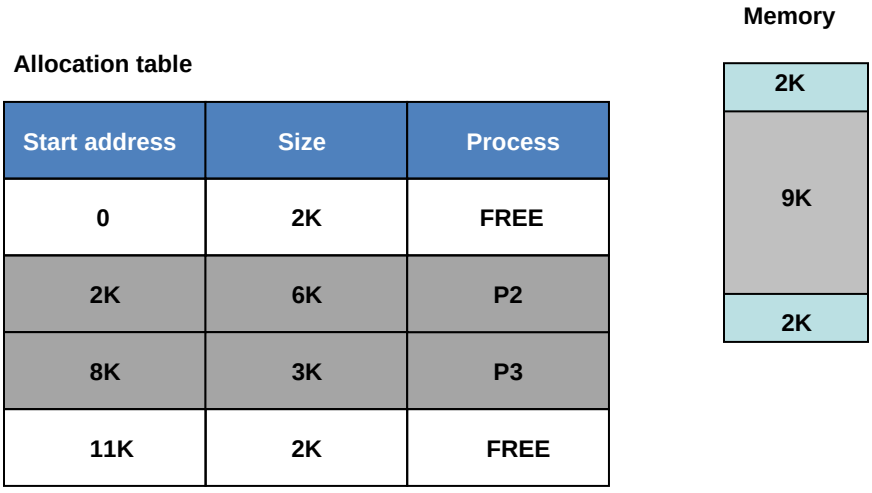


Figure 7.6b: State of allocation table after P1's completion

Suppose there is a new request for a 4KB chunk of memory from a new process P4. Unfortunately, this request cannot be satisfied since the space requested by P4 is contiguous in nature but the available space is fragmented as shown in Figure 7.6b. Therefore, variable size partition, while solving the internal fragmentation problem does not solve the external fragmentation problem.

As processes complete, there will be *holes* of available space in memory created. The allocation table records these available spaces. If adjacent entries in the allocation table free up, the manager will be able to coalesce them into a larger chunk as shown in the before-after Figures 7.6c and 7.6d, respectively.

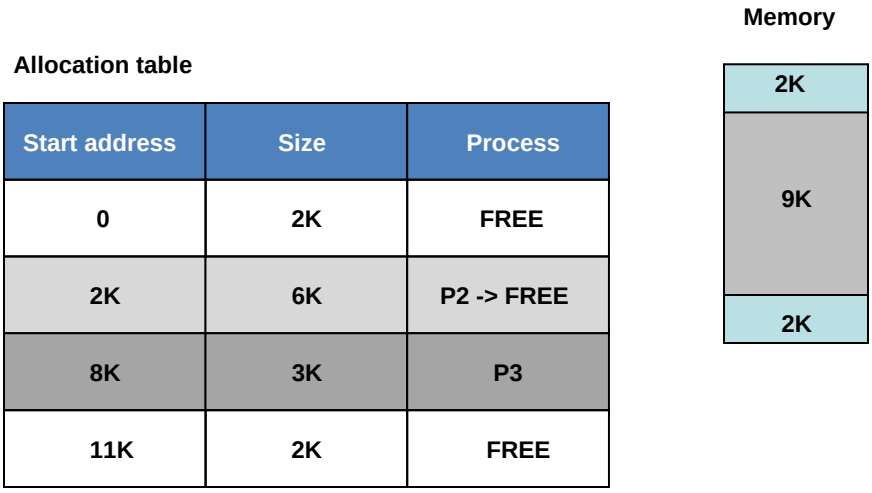


Figure 7.6c: Allocation Table before P2 releases memory

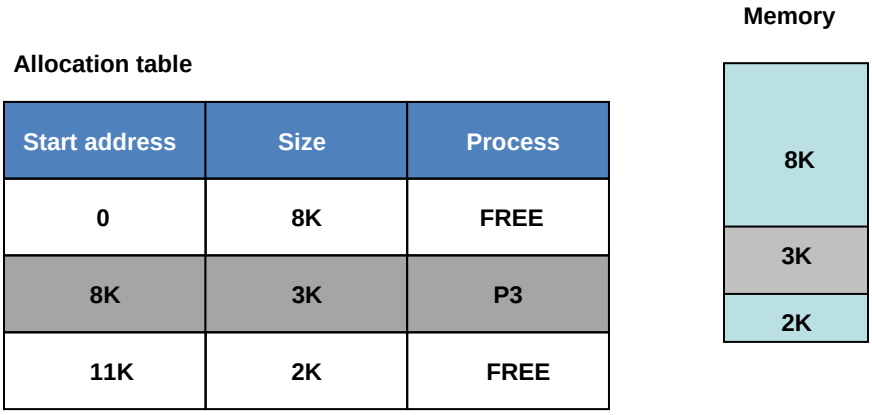


Figure 7.6d: Allocation Table after P2 releases memory (coalesced)

Example 2:
What is the maximum external fragmentation represented by each of the Figures 7.6b, and d?

Answer:
In Figure 7.6b, 2 chunks of 2KB each are available but non contiguous.
External fragmentation = 4K bytes
In Figure 7.6d, two chunks of 8KB and 2KB are available but not contiguous.
External fragmentation = 10K bytes

Upon a new request for space, the memory manager has several options on how to make the allocation. Here are two possibilities:

1. Best fit
The manager looks through the allocation table to find the best fit for the new request. For example, with reference to Figure 7.6d, if the request is for 1KB then the manager will make the allocation by splitting the 2KB free space rather than the 8KB free space.

2. First fit
The manager will allocate the request by finding the first free slot available in the allocation table that can satisfy the new request. For example, with reference to Figure 7.6d, the memory manager will satisfy the same 1KB request by splitting the 8KB free space.

The choice of the allocation algorithm has tradeoffs. The time complexity of the best-fit algorithm is high when the table size is large. However, the best-fit algorithm will lead to better memory utilization since there will be less external fragmentation.

7.3.3 Compaction

The memory manager resorts to a technique called *compaction* when the level of external fragmentation goes beyond tolerable limits. For example, referring to Figure 7.6d, the memory manager may relocate the memory for P3 to start from address 0, thus creating a

contiguous space of 10KB as shown in Figure 7.6e. Compaction is an expensive operation since all the embedded addresses in P3's allocation have to be adjusted to preserve the semantics. This is not only expensive but also virtually impossible in most architectures. Remember that these early schemes for memory management date back to the 60's. It is precisely for allowing dynamic relocation that IBM 360 introduced the concept of a base register[4] (not unlike the scheme shown in Figure 7.4). OS/360 would dynamically relocate a program at load time. However, even with this scheme, once a process is loaded in memory, compaction would require quite a bit of gyrations such as stopping the execution of the processes to do the relocation. Further, the cost of compaction goes up with the number of processes requiring such relocation. For this reason, even in architectures where it is feasible to do memory compaction, memory managers perform compaction rarely. It is usual to combine compaction with swapping; i.e., the memory manage will relocate the process as it is swapped back into memory.

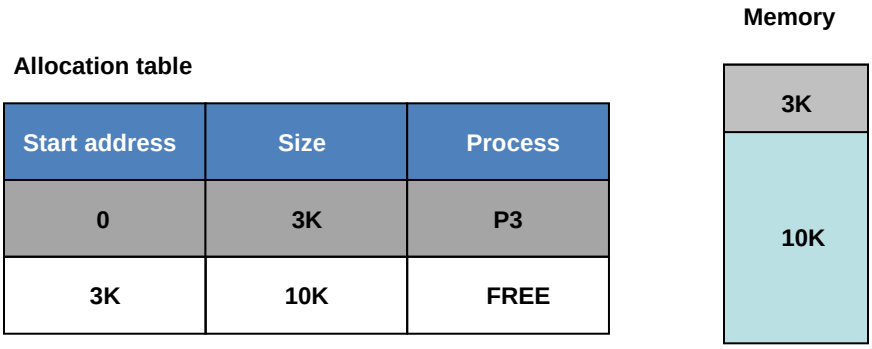


Figure 7.6e: Memory compacted to create a larger contiguous space

7.4 Paged Virtual Memory

As the size of the memory keeps growing, external fragmentation becomes a very acute problem. We need to solve this problem.

Let us go to basics. The user's view of the program is a contiguous footprint in memory. The hardware support for memory management that we have discussed until now, at best relocates the program to a different range of addresses from that originally present in the user's view. Basically, we need to get around this inherent assumption of contiguous memory present in the user's view. The concept of *virtual memory* helps get around this assumption. *Paging* is a vehicle for implementing this concept.

The basic idea is to let the user preserve his view of contiguous memory for his program, which makes program development easy. The broker (in Figure 7.1) breaks up this contiguous view into equal *logical* entities called *pages*. Similarly, the physical memory consists of *page frames*, which we will refer to simply as physical frames. Both the logical page and the physical frame are of the same fixed size, called *pagesize*. A physical frame *houses* a logical page.

[4] Source: Please see http://www.research.ibm.com/journal/rd/441/amdahl.pdf for the original paper by Gene Amdahl on IBM 360.

Let us consider an analogy. The professor likes to get familiar with all the students in his class. He teaches a large class. To help him in his quest to get familiar with all the students, he uses the following ruse. He collects the photos of his students in the class. He has an empty picture frame in his office (Figure 7.7-(a)). When a student comes to visit him during his office hours, he puts up the picture of the student in the frame (Figure 7.7-(b)). When the next student comes to visit, he puts up the picture of that student in the frame (Figure 7.7-(c)). The professor does not have a unique picture frame for each student, but simply re-uses the same frame for the different students. He does not need a unique frame for each student either since he sees him or her one at a time during his office hours anyhow.

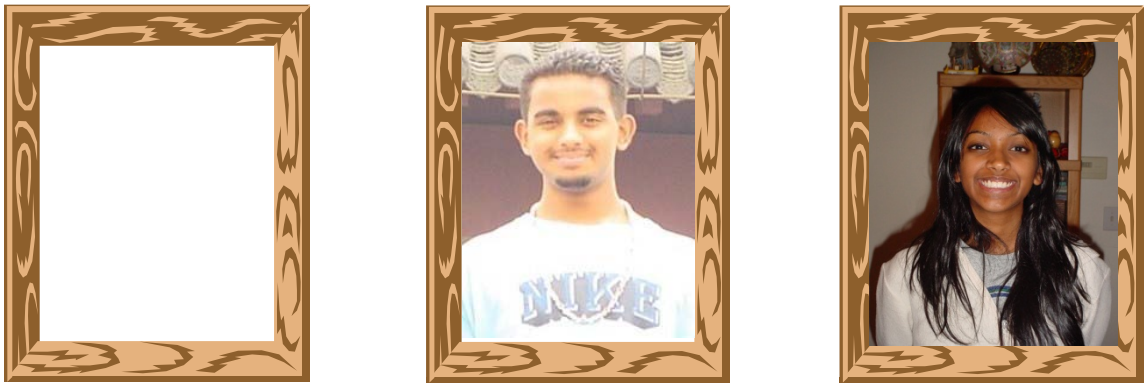


Figure 7.7: Picture Frame Analogy

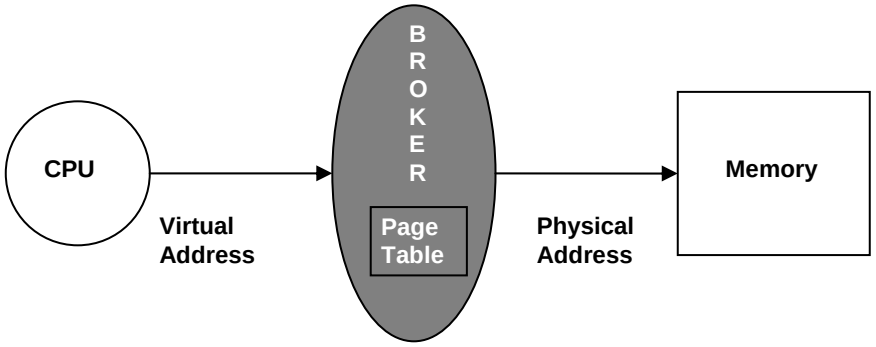


Figure 7.8: Page Table

Allocation of physical memory divided into page frames bears a lot of similarity to this simple analogy. The picture frame can house any picture. Similarly, a given physical frame can be used to house any logical page. The broker maintains a *mapping* between a user's *logical page* and the physical memory's *physical frame*. As one would expect it is the responsibility of the memory manager to create the mapping for each program. An entity called the *page table* holds the mapping of logical pages to physical frames. The page table effectively decouples the user's view of memory from the physical

organization. For this reason, we refer to the user's view as *virtual memory* and to the logical pages as *virtual pages*, respectively. The CPU generates *virtual addresses* corresponding to the user's view. The broker *translates* this virtual address to a physical address by looking up the page table (shown in Figure 7.8). Since we have decoupled the user's view from the physical organization, the relative sizes of the virtual memory and physical memory do not matter. For example, it is perfectly reasonable to have a user's view of the virtual memory to be much larger than the actual physical memory. In fact, this is the common situation in most memory systems today. Essentially, the larger virtual memory removes any resource restriction arising out of limited physical memory, and gives the illusion of a much larger memory to user programs.

The broker is required to maintain the contiguous memory assumption of the user only to addresses within a page. The distinct pages need not be contiguous in physical memory. Figure 7.9 shows a program with four virtual pages mapped using the paging technique to four physical frames. Note that the paging technique circumvents external fragmentation. However, there can be internal fragmentation. Since the frame size is fixed, any request for memory that only partially fills a frame will lead to internal fragmentation.

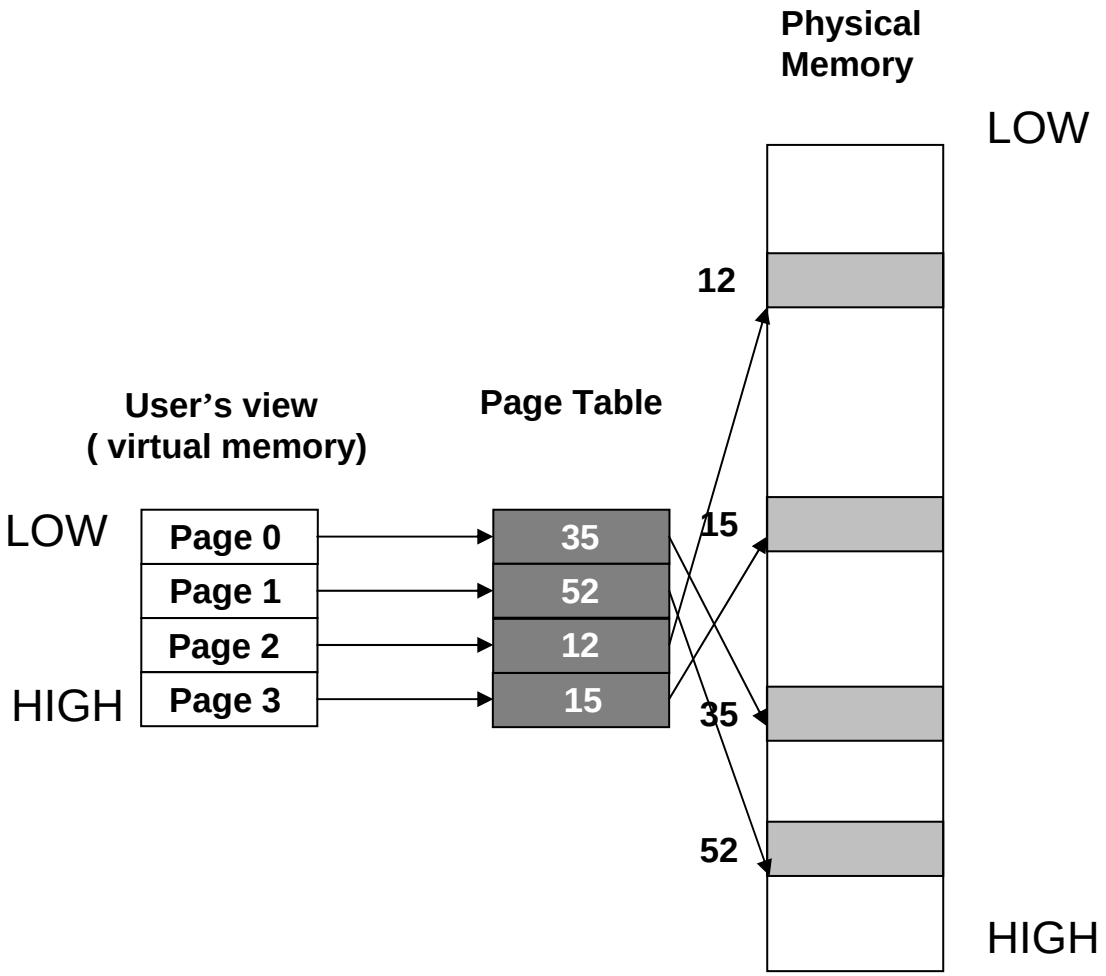


Figure 7.9: Breaking the user's contiguous view of virtual memory

7.4.1 Page Table

Let us drill a little deeper into the paging concept. Given that a virtual page (or a physical frame) is of fixed size, and that addresses within a page are contiguous, we can view the virtual address generated by the CPU as consisting of two things: *virtual page number (VPN)* and *offset* within the page. For the sake of exposition, we will assume that the page size is an integral power of two. The hardware first breaks up the virtual address into these two pieces. This is a straightforward process. Remember that all the locations within a given page are contiguous. Therefore, the offset portion of the virtual address must come from the low order bits. The number of bits needed for the offset is directly discernible from the page size. For example, if the page size is 8 Kbytes then there are 2^{13} distinct bytes in that page; so, we need 13 bits to address each byte uniquely, which will be the size of the offset. The remaining high order bits of the virtual address form the virtual page number. In general, if the page size is N then $\log_2 N$ low order bits of the virtual address form the page offset.

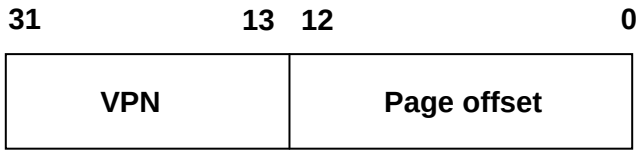
Example 3:

Consider a memory system with a 32-bit virtual address. Assume that the pagesize is 8K Bytes. Show the layout of the virtual address into VPN and page offset.

Answer:

Each page has 8K bytes. We need 13 bits to address each byte within this page uniquely. Since the bytes are contiguous within a page, these 13 bits make up the least significant bits of the virtual address (i.e., bit positions 0 through 12).

The remaining 19 high order bits of the virtual address (i.e., bit positions 13 through 31) signify the virtual page number (VPN).



The translation of a virtual address to a physical address essentially consists of looking up the page table to find the *physical frame number (PFN)* corresponding to the virtual page number (VPN) in the virtual address. Figure 7.10 shows this translation process. The shaded area in Figure 7.10 is the hardware portion of the work done by the broker. The hardware looks up the page table to translate the virtual to physical address. Let us investigate where to place the page table. Since the hardware has to look it up on every memory access to perform the translation, it seems fair to think that it should be part of the CPU datapath. Let us explore the feasibility of this idea. We need an entry for every VPN. In Example 3 above, we need 2^{19} entries in the page table. Therefore, it is infeasible to implement the page table as a hardware device in the datapath of the processor. Moreover, there is not just one page table in the system. To provide memory protection every process needs its own page table.

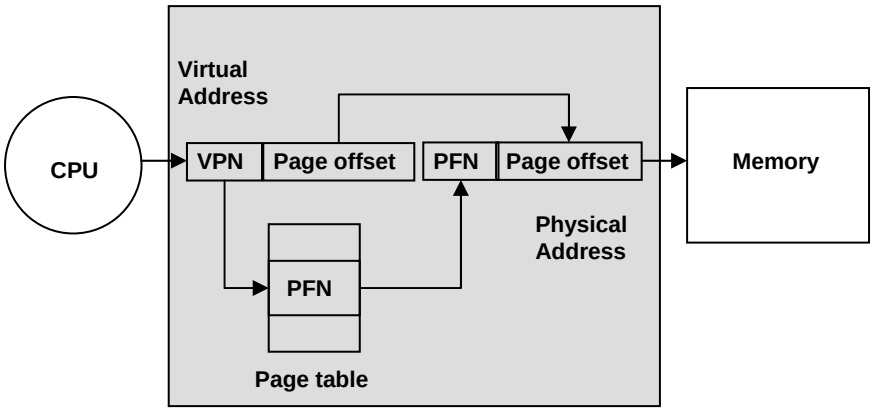


Figure 7.10: Address Translation

Therefore, the page table resides in memory, one per process as shown in Figure 7.11. The CPU needs to know the location of the page table in memory to carry out address translation. For this purpose, we add a new register to the CPU datapath, *Page Table Base Register (PTBR)*, which contains the base address of the page table for the currently running process. The PTBR is part of the process control block. At context switch time, this register value is loaded from the PCB of the newly dispatched process.

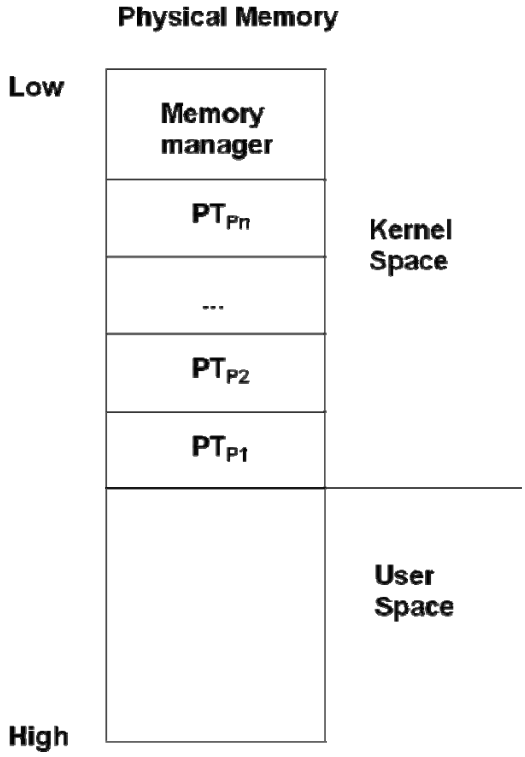
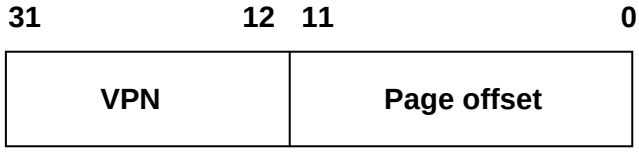


Figure 7.11: Page Tables in Physical Memory

Example 4:
Consider a memory system with 32-bit virtual addresses and 24-bit physical memory addresses. Assume that the pagesize is 4K Bytes. (a) Show the layout of the virtual and physical addresses. (b) How big is the page table? How many page frames are there in this memory system?

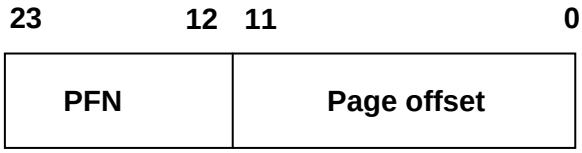
Answer:
a)
For a pagesize of 4Kbytes, the lower 12-bits of the 32-bit virtual address signify the page offset, and the remaining high order bits (20-bits) signify the VPN.

Layout of the virtual address:



Since the physical address is 24 bits, the high order bits 12 bits (24-12) of the physical address forms the PFN.

Layout of the physical address:



(b)

Number of page table entries = $2^{(\text{Number of bits in VPN})} = 2^{20}$
Assuming each entry is 1 word of 32 bits (4 bytes),
The size of the page table = 4 x 2^{20} bytes = 4 Mbytes

Number of page frames = $2^{(\text{Number of bits in PFN})} = 2^{12} = 4096$

7.4.2 Hardware for Paging

The hardware for paging is straightforward. We add a new register, PTBR, to the datapath. On every memory access, the CPU computes the address of the *page table entry* (PTE) that corresponds to the VPN in the virtual address using the contents of the PTBR. The PFN fetched from this entry concatenated with the page offset gives the

physical address. This is the translation process for fetching either the instruction or the data in the FETCH and MEM stages of the pipelined processor, respectively. The new hardware added to the processor to handle paging is surprisingly minimal for the enormous flexibility achieved in memory management. Let us review the overhead for memory accesses with paging. In essence, the hardware makes two trips to the memory for each access: first to retrieve the PFN and second to retrieve the memory content (instruction or data). This seems grossly inefficient and untenable from the point of view of sustaining a high performance processor pipeline. Fortunately, it is possible to mitigate this inefficiency significantly to make paging actually viable. The trick lies in remembering the recent address translations (i.e., VPN to PFN mappings) in a small hardware table inside the processor for future use, since we are most likely to access many memory locations in the same physical page. The processor first consults this table, called a *translation lookaside buffer (TLB)*. Only on not finding this mapping does it retrieve the PFN from physical memory. More details on TLB will be forthcoming in the next chapter (please see Section 8.5).

7.4.3 Page Table Set up

The memory manager sets up the page table on a process startup. In this sense, the page table does double duty. The hardware uses it for doing the address translation. It is also a data structure under the control of the memory manager. Upon setting up the page table, the memory manager records the PTBR value for this process in the associated PCB. Figure 7.12 shows the process control block with a new field for the PTBR.

```
typedef struct control_block_type {
        enum state_type state;
        address PC;
        int reg_file[NUMREGS];
        struct control_block *next_pcb;
        int priority;
        address PTBR;
        ….
        ….
} control_block;
```

Figure 7.12: PCB with PTBR field

7.4.4 Relative sizes of virtual and physical memories

As motivated in this section, the whole purpose of virtual memory is to liberate the programmer from the limitations of available physical memory. Naturally, this leads us to think that virtual memory should always be larger than physical memory. This is a perfectly logical way to think about virtual memory. Having said that, let us investigate if it makes sense to have physical memory that is larger than the virtual memory. We can immediately see that it would not help an individual program since the address space available to the program is constrained by the size of the virtual memory. For example,

with a 32-bit virtual address space, a given program has access to 4 Gbytes of memory. Even if the system has more physical memory than 4 Gbytes, an individual program cannot have a memory footprint larger than 4 Gbytes. However, a larger physical memory can come in handy for the operating system to have more resident processes that are huge memory hogs. This is the reason for Intel architecture's *Physical Address Extension (PAE)* feature, which extends the physical address from 32-bits to 36-bits. As a result, this feature allows the system to have up to 64 Gbytes of physical memory, and the operating system can map (using the page table) a given process's 4 Gbytes virtual address space to live in different portions of the 64 Gbytes physical memory[5].

One could argue that with the technological advances that allow us to have larger than 32-bit physical addresses, the processor architecture should support a larger virtual address space as well. One would be right, and in fact, vendors including Intel have already come out with 64-bit architectures. The reason why Intel offers the PAE feature is simply to allow larger physical memories on 32-bit platforms that are still in use for supporting legacy applications.

7.5 Segmented Virtual Memory

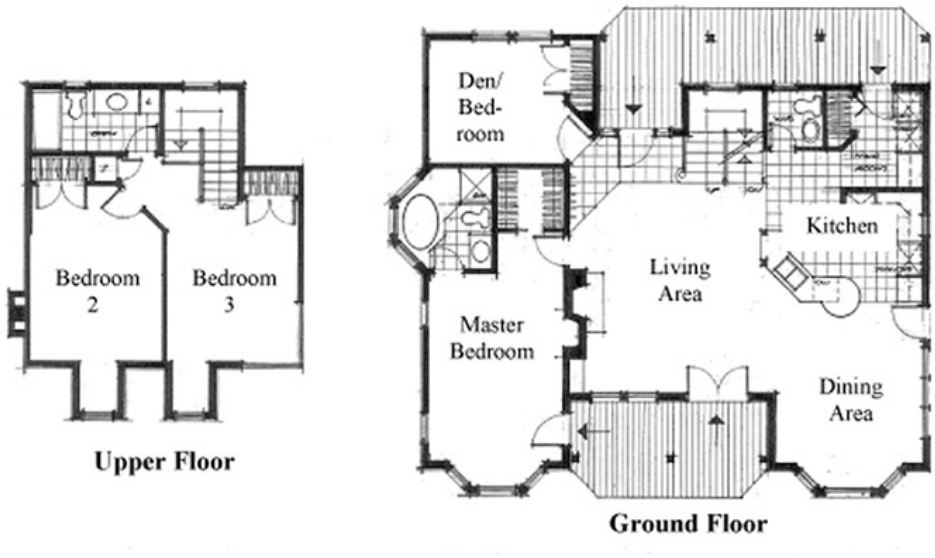


Figure 7.13: Floor plan for a house[6]

Let us consider an analogy. Figure 7.13 shows a floor plan for a house. You may have a living room, a family room, a dining room, perhaps a study room, and one or more bedrooms. In other words, we have first logically organized the space in the house into functional units. We may then decide the amount of actual physical space we would like to allocate given the total space budget for the house. Thus, we may decide to have a

[5] The interested reader is referred to, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1."
[6] Source: http://www.edenlanehomes.com/images/design/Typical-Floorplan.jpg

guest bedroom that is slightly larger than say a kid's bedroom. We may have breakfast area that is cozy compared to the formal dining room, and so on. There are several advantages to this functional organization of the space. If you have visitors, you don't have to rearrange anything in your house. You can simply give them the guest bedroom. If you decide to bring a friend for sleepover, you may simply share your bedroom with that friend without disturbing the other members of your family. Let us apply this analogy to program development.

In the previous chapter (see Section 6.2), we defined the address space of a process as the space occupied by the memory footprint of the program. In the previous section, we stressed the need to preserve the contiguous view of the memory footprint of a user's program. We will go a little further and underscore the importance of having the address space always start at 0 to some maximum value, so that it is convenient from the point of view of code generation by the compiler. Virtual memory helps with that view.

Let us investigate if one address space is enough for a program. The analogy of the space planning in the house is useful here. We logically partitioned the space in the house into the different rooms based on their intended use. These spaces are independent of each other and well protected from one another (doors, locks, etc.). This ensures that no one can invade the privacy of the other occupants without prior warning. At some level, constructing a program is similar to designing a house. Although we end up with a single program (a.out as it is called in Unix terminology), the source code has a logical structure. There are distinct data structures and procedures organized to provide specific functionality. If it is a group project, you may even develop your program as a team with different team members contributing different functionalities of the total program. One can visualize the number of software engineers who would have participated in the development of a complex program such as MS Word at Microsoft. Therefore, the availability of multiple address spaces would help better organize the program logically. Especially, with object-oriented programming the utility of having multiple address spaces cannot be over-emphasized.

Let us drill a little deeper and understand how multiple address spaces would help the developer. Even for the basic memory footprint (please see Figure 6.1 in Chapter 6), we could have each of the distinct portions of the memory footprint, namely, code, global data, heap, and stack in distinct address spaces. From a software engineering perspective, this organization gives us the ability to associate properties with these address spaces (for example, the code section is read only, etc.). Further, the ability to associate such properties with individual address spaces will be a great boon for program debugging.

Use of multiple address spaces becomes even more compelling in large-scale program development. Let us say, we are writing an application program for video-based surveillance. It may have several components as shown in Figure 7.14. One can see the similarity between a program composed of multiple components each in its dedicated address space and the floor plan of Figure 7.13.

This is a complex enough application that there may be several of us working on this project together. The team members using well-defined interfaces may independently develop each of the boxes shown in Figure 7.14. It would greatly help both the development as well as debugging to have each of these components in separate address spaces with the appropriate levels of protection and sharing among the components (similar to having doors and locks in the house floor plan). Further, it would be easy to maintain such an application. You don't move into a hotel if you wanted to retile just your kitchen. Similarly, you could rewrite specific functional modules of this application without affecting others.

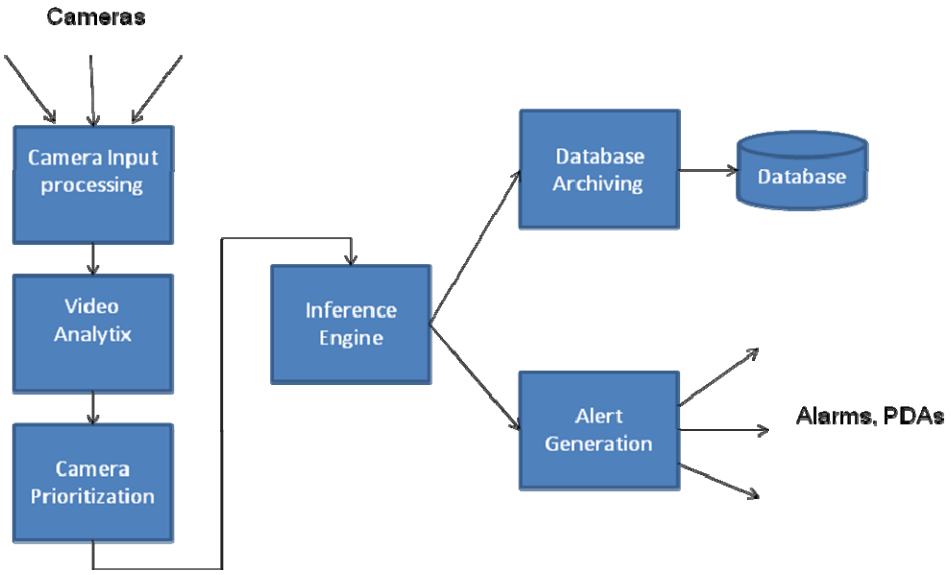


Figure 7.14: An Example Application

Segmentation is a technique for meeting the above vision. As is always the case with any system level mechanism, this technique is also a partnership between the operating system and the architecture.

The user's view of memory is not a single linear address space; but it is composed of several distinct address spaces. Each such address space is called a *segment*. A segment has two attributes:

- Unique segment number
- Size of the segment

Each segment starts at address 0 and goes up to (Segment size – 1). CPU generates addresses that have two parts as shown below in Figure 7.15.

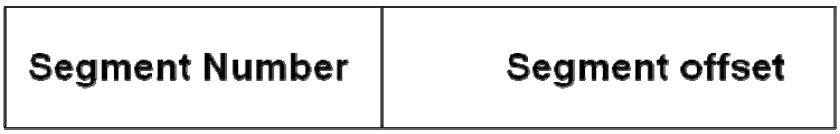


Figure 7.15: A segmented address

As with paging, the broker that sits in the middle between the CPU and memory converts this address into a physical address by looking up the segment table (Figure 7.16). As in the case of paging, it is the operating system that sets up the segment table for the currently running process.

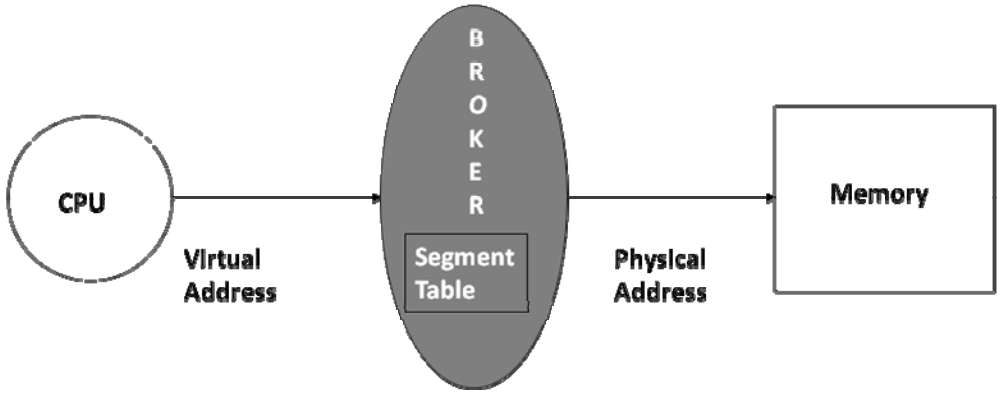


Figure 7.16: Segment Table

Now, you are probably wondering that but for the name change from "Page" to "Segment" there appears to be not much difference between paging and segmentation. Before, we delve into the differences between the two, let us return to the example application in Figure 7.14. With segmentation, we could arrange the application as shown in Figure 7.17. Notice that each functional component is in its own segment, and the segments are individually sized depending on the functionality of that component. Figure 7.18 shows these segments housed in the physical memory.

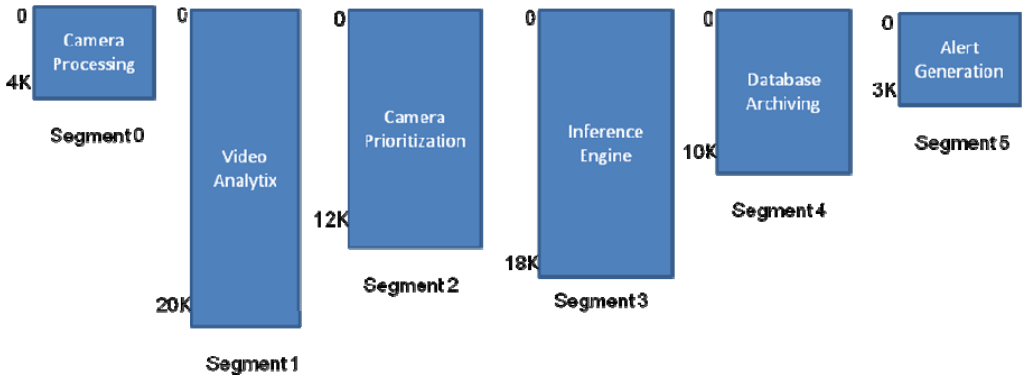


Figure 7.17: Example Application Organized into Segments

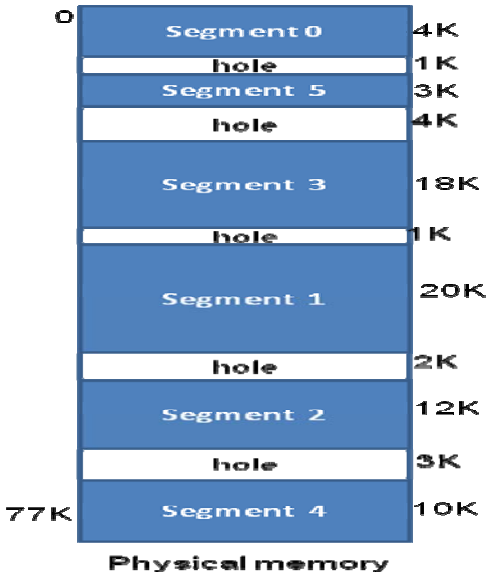


Figure 7.18: Segments of example application in Figure 7.17 mapped into physical memory

---

Example:

A program has 10KB code space and 3KB global data space. It needs 5KB of heap space and 3KB of stack space. The compiler allocates individual segments for each of the above components of the program. The allocation of space in physical memory is as follows:

Code start address 1000
Global data start address 14000
Heap space start address 20000
Stack space start address 30000

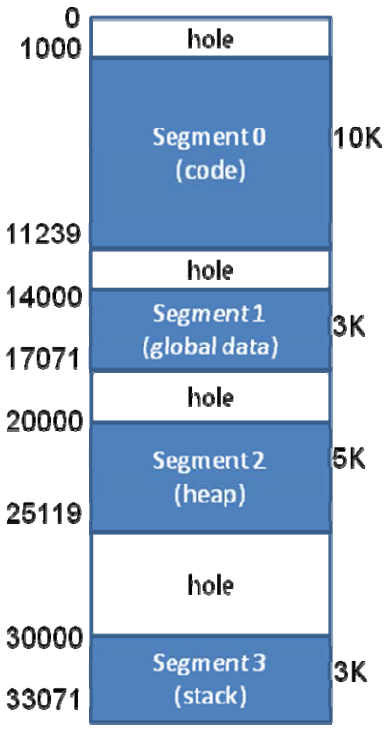
(a) Show the segment table for this program.

Answer:

Segment Number	Start address	Size
0	1000	10KB
1	14000	3KB
2	20000	5KB
3	30000	3KB

(b) Assuming byte-addressed memory, show the memory layout pictorially.

Answer:



(c) What is the physical memory address corresponding to the virtual address:

0	**299**

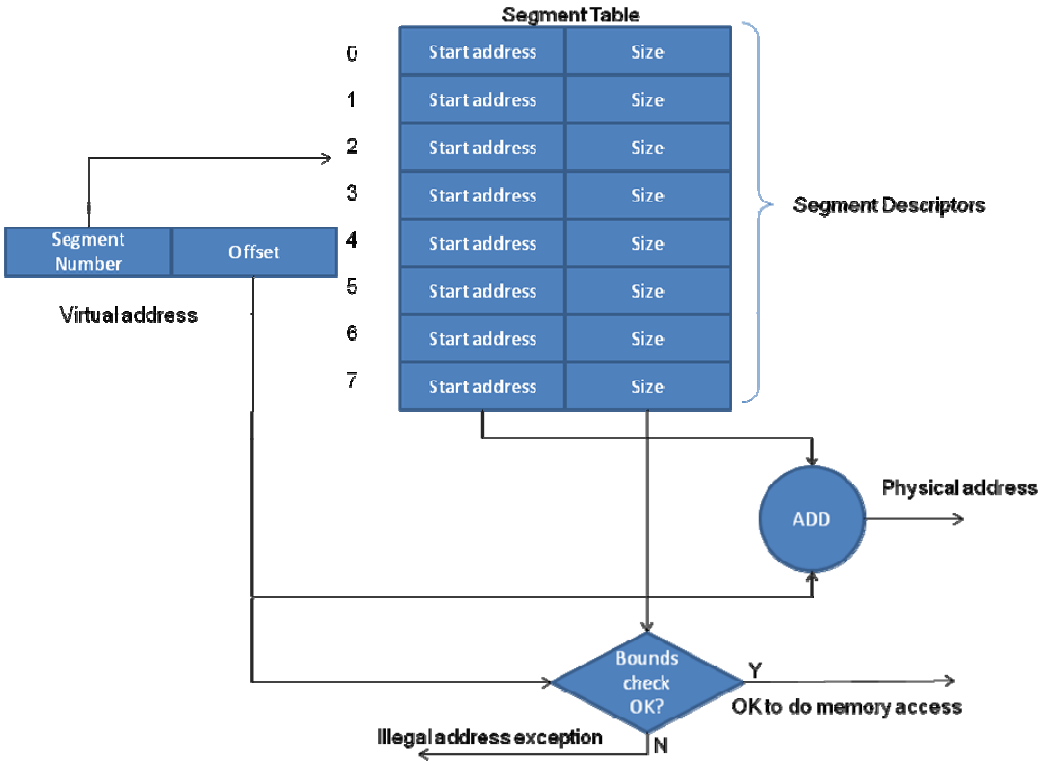
Answer:

1) The offset 299 is within the size of segment 0 (10K).
2) The physical memory address
= Start address of segment 0 + offset
= 1000+299
= **1299**

7.5.1 Hardware for Segmentation

The hardware needed for supporting segmentation is fairly simple. The segment table is a data structure similar to the page table. Each entry in this table called a *segment descriptor*. The segment descriptor gives the start address for a segment and the size of the segment. Each process has its own segment table allocated by the operating system at the time of process creation. Similar to paging, this scheme also requires a special register in the CPU called *Segment Table Base Register (STBR)*. The hardware uses this

register and the segment table to do address translation during the execution of the process (see Figure 7.19). The hardware performs a bounds check first to ensure that the



offset provided is within the size limit of the segment before allowing the memory access to proceed.

Figure 7.19: Address translation with segmentation

Segmentation should remind the reader of the memory allocation scheme from Section 7.3.2, variable-size partition. It suffers from the same problem as in variable size partition, namely, external fragmentation. This can be seen in Figure 7.18.

7.6 Paging versus Segmentation

Now, we are ready to understand the difference between paging and segmentation. Both are techniques for implementing virtual memory but differ greatly in details. We summarize the similarities and differences between the two approaches in Table 7.1.

Attribute	Paging	Segmentation
User shielded from size limitation of physical memory	Yes	Yes
Relationship to physical memory	Physical memory may be less than or greater than virtual memory	Physical memory may be less than or greater than virtual memory
Address spaces per process	One	Several
Visibility to the user	User unaware of paging; user is given an illusion of a single linear address space	User aware of multiple address spaces each starting at address 0
Software engineering	No obvious benefit	Allows organization of the program components into individual segments at user discretion; enables modular design; increases maintainability
Program debugging	No obvious benefit	Aided by the modular design
Sharing and protection	User has no direct control; operating system can facilitate sharing and protection of pages across address spaces but this has no meaning from the user's perspective	User has direct control of orchestrating the sharing and protection of individual segments; especially useful for object-oriented programming and development of large software
Size of page/segment	Fixed by the architecture	Variable chosen by the user for each individual segment
Internal fragmentation	Internal fragmentation possible for the portion of a page that is not used by the address space	None
External fragmentation	None	External fragmentation possible since the variable sized segments have to be allocated in the available physical memory thus creating holes (see Figure 7.18)

Table 7.1: Comparison of Paging and Segmentation

When you look at the table able, you would conclude that segmentation has a lot going for it. Therefore, it is tempting to conclude that architectures should be using segmentation as the vehicle for implementing virtual memory. Unfortunately, the last row of the table is the real killer, namely, external fragmentation. There are other considerations as well. For example, with paging the virtual address that the CPU generates occupies one memory word. However, with segmentation, it may be necessary to have two memory words to specify a virtual address. This is because, we may want a segment to be as big as the total available address space to allow for maximum flexibility. This would mean we would need one memory word for identifying the segment number and the second to identify the offset within the segment. Another serious system level consideration has to do with balancing the system as a whole. Let us elaborate what we mean by this statement. It turns out that the hunger for memory of applications and system software keeps growing incessantly. The increase in memory footprint of desktop publishing applications and web browsers from release to release is an attestation to this growing appetite. The reason of course is the desire to offer increased functionality to the end user. The reality is that we will never be able to afford the physical memory to satisfy our appetite for memory. Therefore, necessarily virtual memory will be far larger than physical memory. Pages or segments have to be brought in "on demand" from the hard drive into the physical memory. Virtual memory extends the memory system from the physical memory to the disk. Thus, the transfer of data from the disk to the memory system on demand has to be efficient for the whole system to work efficiently. This is what we mean by balancing the system as a whole. Since the page-size is a system attribute it is easier to optimize the system as a whole with paging. Since the user has control over defining the size of segments it is more difficult to optimize the system as a whole.

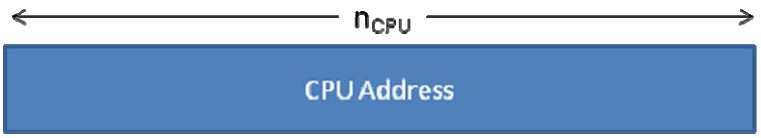
For these reasons, true segmentation as described in this section is not a viable way to implement virtual memory. One way to solve the external fragmentation problem is as we described in Section 7.3.3, by using memory compaction. However, we observed the difficulties in practically implementing compaction in that section as well. Therefore, a better approach is to have a combined technique, namely, *paged-segmentation*. The user is presented with a segmented view as described in this section. Under the covers, the operating system and the hardware use paging as described in the previous section to eliminate the ill effects of external fragmentation.

Description of such paged-segmentation techniques is beyond the scope of this textbook. We do present a historical perspective of paging and segmentation as well as a commercial example of paged-segmentation approach using Intel Pentium architecture as a case study in Section 7.8.

7.6.1 Interpreting the CPU generated address

The processor generates a simple linear address to address the memory.

CPU generated address:



The number of bits in the CPU generated address depends on the memory addressing capability of the processor (usually tied to the word-width of the processor and the smallest granularity of access to memory operands). For example, for a 64-bit processor with byte-addressed memory $n_{CPU} = 64$.

The number of bits in the physical address depends on the actual size of the physical memory.

With byte-addressed memory,

$$n_{phy} = \log_2(\text{size of physical memory in bytes}) \qquad (3)$$

For example, for a physical memory size of 1 Gbyte, $n_{phy} = 30$ bits.

The exact interpretation of the CPU generated linear address as a virtual address depends on the memory system architecture (i.e., paging versus segmentation). Correspondingly, the computation of the physical address also changes. Table 7.2 summarizes the salient equations pertaining to paged and segmented memory systems.

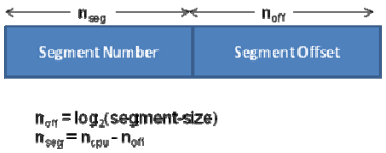
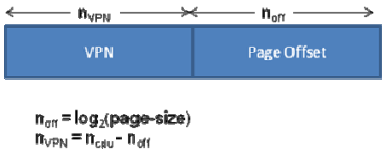
Memory System	Virtual Address Computation	Physical Address Computation	Size of Tables
Segmentation		Segment Start address = Segment-Table [Segment-Number] Physical address = Segment Start Address + Segment Offset	Segment table size = $2^{n_{seg}}$ entries
Paging		PFN = Page-Table[VPN] Physical address: 	Page table size = $2^{n_{VPN}}$ entries

Table 7.2: Address Computations in Paged and Segmented Memory Systems

7.7 Summary

The importance of the memory system cannot be over-emphasized. The performance of the system as a whole crucially depends on the efficiency of the memory system. The

interplay between the hardware and the software of a memory system makes the study of memory systems fascinating. Thus far, we have reviewed several different memory management schemes and the hardware requirement for those schemes. In the beginning of the chapter, we identified four criteria for memory management: improved resource utilization, independence and protection of process' memory spaces, liberation from memory resource limitation, and sharing of memory by concurrent processes. These are all important criteria from the point of view of the efficiency of the operating system for managing memory, which is a precious resource. From the discussion on segmentation, we will add another equally important criterion for memory management, namely, *facilitating good software engineering practices*. This criterion determines whether the memory management scheme, in addition to meeting the system level criteria also helps in the development of software that is flexible, easy to maintain, and evolve. Let us summarize these schemes with respect to these memory management criteria. Table 7.3 gives such a qualitative comparison of these memory management schemes.

Memory Management Criterion	User/ Kernel Separation	Fixed Partition	Variable-sized Partition	Paged Virtual Memory	Segmented Virtual Memory	Paged-segmented Virtual Memory
Improved resource utilization	No	Internal fragmentation bounded by partition size; External fragmentation	External fragmentation	Internal fragmentation bounded by page size	External fragmentation	Internal fragmentation bounded by page size
Independence and protection	No	Yes	Yes	Yes	Yes	Yes
Liberation from resource limitation	No	No	No	Yes	Yes	Yes
Sharing by concurrent processes	No	No	No	Yes	Yes	Yes
Facilitates good software engineering practice	No	No	No	No	Yes	Yes

Table 7.3: Qualitative Comparison of memory management schemes

Only virtual memory with paged-segmentation meets all the criteria. It is instructive to review which one of these schemes are relevant to this day with respect to the state-of-the-art in memory management. Table 7.4 summarizes the memory management

schemes covered in this chapter along with the required hardware support, and their applicability to modern operating systems.

Scheme	Hardware Support	Still in Use?
User/Kernel Separation	Fence register	No
Fixed Partition	Bounds registers	Not in any production operating system
Variable-sized Partition	Base and limit registers	Not in any production operating system
Paged Virtual Memory	Page table and page table base register	Yes, in most modern operating system
Segmented Virtual Memory	Segment table, and segment table base register	Segmentation in this pure form not supported in any commercially popular processors
Paged-segmented Virtual Memory	Combination of the hardware for paging and segmentation	Yes, most modern operating systems based on Intel x86 use this scheme[7]

Table 7.4: Summary of Memory management schemes

7.8 Historical Perspective

Circa 1965, IBM introduced System/360 series of mainframes. The architecture provided base and limit registers thus paving the way for memory management that supports dynamic relocation. Any general-purpose register could be used as the base register. The compiler would designate a specific register as the base register so that any program written in high-level language could be dynamically relocated by the operating system OS/360. However, there is a slight problem. Programmers could find out which register was being used as the base register and could use this knowledge to "stash" away the base register value for assembly code that they want to insert into the high-level language program[8]. The upshot of doing this is that the program can no longer be relocated once it starts executing, since programmers may have hard coded addresses in the program. You may wonder why they would want to do such a thing. Well, it is not unusual for programmers who are after getting every ounce of performance from a system to lace in assembly code into a high-level language program. Some of us are guilty of that even in this day and age! Due to these reasons, dynamic relocation never really worked as well as IBM would have wanted it to work in OS/360. The main reason was the fact that the address shifting in the architecture used a general-purpose register that was visible to the programmer.

Circa 1970, IBM introduced virtual memory in their System/370 series of mainframes[9]. The fundamental way in which System/370 differed from System/360 was the

[7] It should be noted that Intel's segmentation is quite different from the pure form of segmentation presented in this chapter. Please Section 7.8.2 for a discussion of Intel's paged-segmentation scheme.
[8] Personal communication with James R. Goodman, University of Wisconsin-Madison.
[9] For an authoritative paper that describes System/360 and System/370 Architectures, please refer to:

architectural support for dynamic address translation, which eliminated the above problem with System/360. System/370 represents IBM's first offering of the virtual memory concept. Successive models of the System/370 series of machines refined the virtual memory model with expanded addressing capabilities. The architecture used paged virtual memory.

In this context, it is worth noting a difference in the use of the terminologies "static" and "dynamic", depending on operating system or architecture orientation. Earlier (see Section 7.2), we defined what we mean by static and dynamic relocation from the operating system standpoint. With this definition, one would say a program is dynamically relocatable if the hardware provides support for changing the virtual to physical mapping of addresses at execution time. By this definition, the base and limit register method of IBM 360 supports dynamic relocation.

Architects look at a much finer grain of individual memory accesses during program execution and use the term address "translation". An architecture supports dynamic address translation if the mapping of virtual to physical can be altered at any time during the execution of the program. By this definition, the base and limit register method in IBM 360 supports only static address translation, while paging supports dynamic address translation. Operating system definition of "static vs. dynamic" relocation is at the level of the whole program while the architecture definition of "static vs. dynamic" address translation is at the level of individual memory accesses.

Even prior to IBM's entry into the virtual memory scene, Burroughs Corporation[10] introduced segmented virtual memory in their B5000 line of machines. Another company, General Electric, in partnership with the MULTICS project at MIT introduced paged-segmentation in their GE 600 line of machines[11] in the mid 60's. IBM quickly sealed the mainframe war of the 60's and 70's with the introduction of VM/370 operating system to support virtual memory and the continuous refinement of paged virtual memory in their System/370 line of machines. This evolution has continued to this day and one could see the connection to these early machines even in the IBM z series of mainframes[12] to support enterprise level applications.

7.8.1 MULTICS

Some academic computing projects have a profound impact on the evolution of the field for a very long time. MULTICS project at MIT was one such. The project had its had hey days in the 60's and one could very easily see that much of computer systems as we know it (Unix, Linux, Paging, Segmentation, Security, Protection, etc.) had their birth in this project. In some sense, the operating systems concepts introduced in the MULTICS project were way ahead of their time and the processor architectures of that time were not geared to support the advanced concepts of memory protection advocated by MULTICS.

---

http://www.research.ibm.com/journal/rd/255/ibmrd2505D.pdf

[10] Source: http://en.wikipedia.org/wiki/Burroughs_large_systems

[11] Source: http://en.wikipedia.org/wiki/GE_645

[12] Source: http://www-03.ibm.com/systems/z/

The MULTICS project is an excellent example of the basic tenet of this textbook, namely, the connectedness of system software and machine architecture.

MULTICS introduced the concept of paged segmentation[13]. Figure 7.20 depicts the scheme implemented in MULTICS.

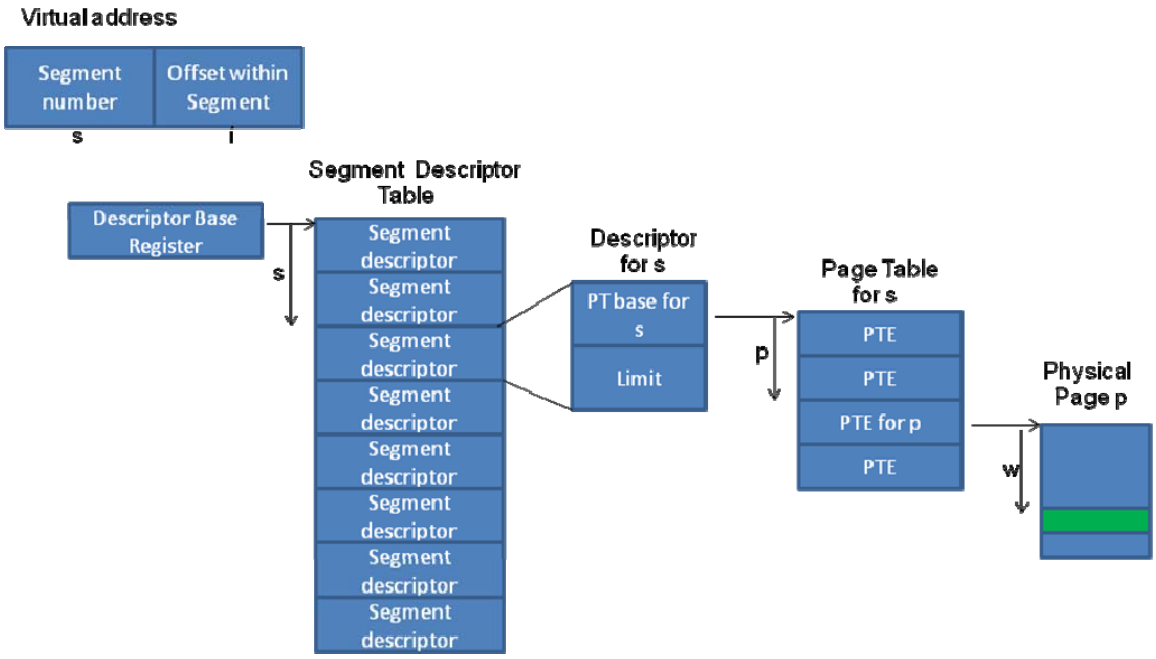


Figure 7.20: Address Translation in MULTICS

The 36-bit virtual address generated by the CPU consists of two parts: an 18-bit segment number (s) and an 18-bit offset (i) within that segment. Each segment can be arbitrary in size bounded by the maximum segment size of $2^{18} - 1$. To avoid external fragmentation, a segment consists of pages (in MULTICS the page-size is 1024 words, with 36-bits per word). Each segment has its own page table. Dynamic address translation in MULTICS is a two-step process:

- Locate the segment descriptor corresponding to the segment number: Hardware does this lookup by adding the segment number to the base address of the segment table contained in a CPU register called Descriptor Base Register.
- The segment descriptor contains the base address for the page table for that segment. Using the segment offset in the virtual address and the page-size, hardware computes the specific page table entry corresponding to the virtual address. The physical address is obtained by concatenating the physical page number with the page offset (which is nothing but segment offset *mod* page-size).

If s is the segment number and i is the offset within the segment specified in the virtual address then, the specific memory word we are looking for is at page offset w from the beginning of the p^{th} page of the segment, where:

[13] See the original MULTICS paper: http://www.multicians.org/multics-vm.html

$$w = i \bmod 1024$$
$$p = (i - w) / 1024$$

Figure 7.20 shows this address translation.

7.8.2 Intel's Memory Architecture

Intel Pentium line of processors also uses paged-segmentation. However, its organization is a lot more involved than the straightforward scheme of MULTICS. One of the realities of academic projects versus an industrial product is the fact that the latter has to worry about backward compatibility with its line of processors. Backward compatibility essentially means that a new processor that is a successor to an earlier one has to be able to run, unchanged, the code that used to run on its predecessors. This is a tall order and constrains the designers of a new processor significantly. The current memory architecture of Intel Pentium evolved from the early editions of Intel's x86 architectures such as the 80286; consequently, it has vestiges of the segmentation scheme found in such older processors coupled with the aesthetic need for providing large numbers of virtual address spaces for software development.

We have intentionally simplified the discussion in this section. As a first order of approximation, a virtual address is a segment selector plus an offset (see Figure 7.21). Intel's architecture divides the total segment space into two halves: *system* and *user*. The system segments are common to all processes, while the user segments are unique to each process. As you may have guessed, the system segments would be used by the operating system since it is common to all the user processes. Correspondingly, there is one descriptor table common to all processes called the *Global Descriptor Table (GDT)* and one unique to each process called the *Local Descriptor Table (LDT)*. The segment selector in Intel is similar to the segment number in MULTICS with one difference. A bit in the segment selector identifies whether the segment being named by the virtual address is a system or a user segment.

As in MULTICS, the segment descriptor for the selected segment contains the details for translating the offset specified in the virtual address to a physical address. The difference is that there is a choice of using simple segmentation without any paging (to be compatible with earlier line of processors) or use paged-segmentation. In the former case, the address translation will proceed as we described in Section 7.5 (i.e., without the need for any page table). This is shown in Figure 7.21 (the choice of GDT versus LDT for the translation is determined by the U/S bit in the selector). The effective address computed is the physical address. In the case of paged-segmentation, the base address contained in the descriptor is the address of the page table base corresponding to this segment number. The rest of the address translation will proceed as we described in Section 7.8.1 for MULTICS, going through the page table for the selected segment (see Figure 7.20). A control bit in a global control register decides the choice of pure segmentation versus paged-segmentation.

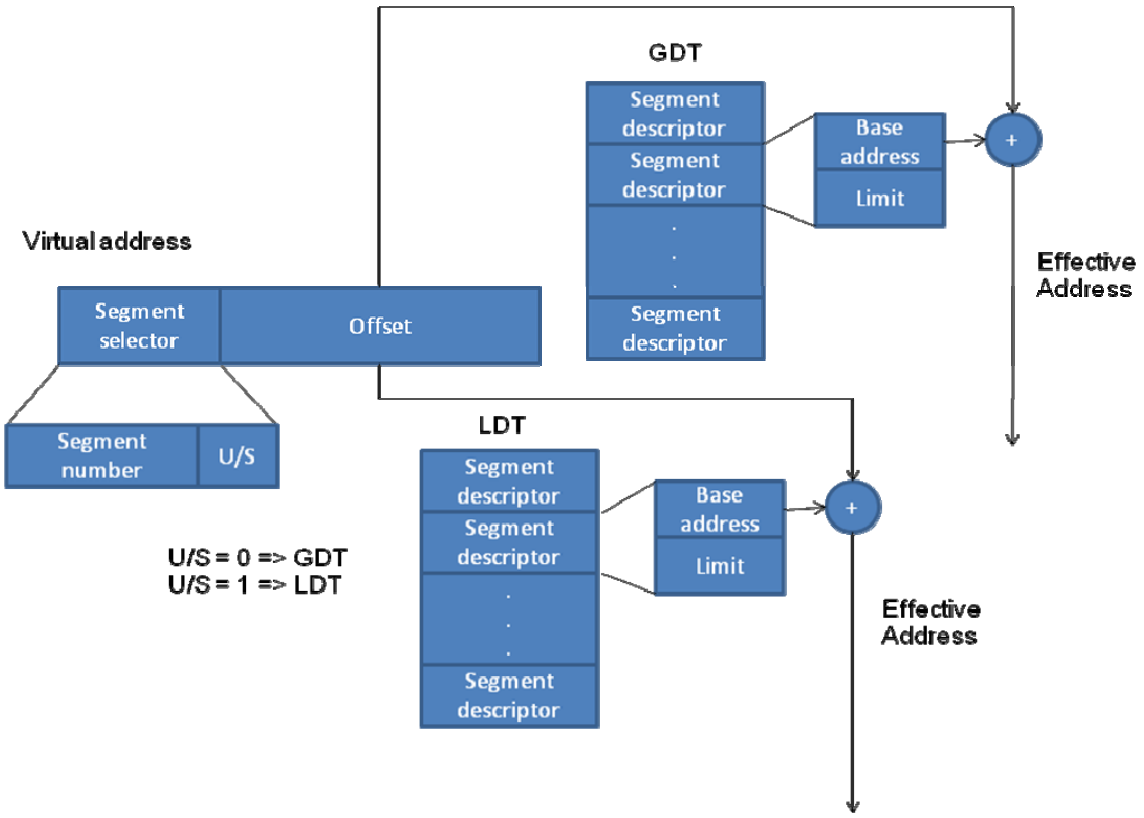


Figure 7.21: Address Translation in Intel Pentium with pure Segmentation

Please refer to Intel's System Programming Guide[14] if you would like to learn more about Intel's virtual memory architecture.

7.9 Review Questions

1. What are the main goals of memory management?

2. Argue for or against the statement: Given that memory is cheap and we can have lots of it, there is no need for memory management anymore.

3. Compare and contrast internal and external fragmentation.

4. A memory manager allocates memory in fixed size chunks of 2048 bytes. The current allocation is as follows:

   P1 1200 bytes
   P2 2047 bytes
   P3 1300 bytes
   P4 1 byte

[14] **Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide:** http://www.intel.com/design/processor/manuals/253668.pdf

What is the total amount memory wasted with the above allocation due to internal fragmentation?

5. Answer True/False with justification: Memory compaction is usually used with fixed size partition memory allocation scheme.

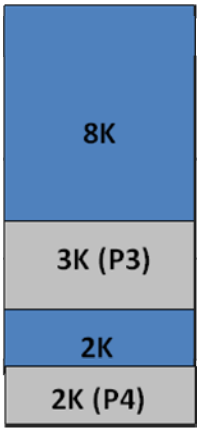
6. Answer True/False with justification: There is no special advantage for the base and limit register solution over the bounds register solution for memory management.

7. Assume an architecture that uses base and limit register for memory management. The memory manager uses variable sized partition allocation. The current memory allocation is as shown below.

Allocation table

Start address	Size	Process
0	8K	FREE
8K	3K	P3
11K	2K	FREE
13K	2K	P4

Memory



There is a new memory request for 9 Kbytes. Can this allocation be satisfied? If not why not? What is the amount of external fragmentation represented by the above figure?

8. What is the relationship between page size and frame size in a paged memory system?

9. In terms of hardware resources needed (number of new processor registers and additional circuitry to generate the physical memory address given a CPU generated address) compare base and limit register solution with a paged virtual memory solution.

10. How is a paged virtual memory system able to eliminate external fragmentation?

11. Derive a formula for the maximum internal fragmentation with a page size of *p* in a paged virtual memory system.

12. Consider a system where the virtual addresses are 20 bits and the page size is 1 KB. How many entries are in the page table?

13. Consider a system where the physical addresses are 24 bits and the page size is 8 KB. What is the maximum number of physical frames possible?

14. Distinguish between paged virtual memory and segmented virtual memory.

15. Given the following segment table:

Segment Number	Start address	Size
0	3000	3KB
1	15000	3KB
2	25000	5KB
3	40000	8KB

What is the physical address that corresponds to the virtual address shown below?

1	**399**

16. Compare the hardware resources needed (number of new processor registers and additional circuitry to generate the physical memory address given a CPU generated address) for paged and segmented virtual memory systems.