Module 4: Memory Management

The von Neumann principle for the design and operation of computers requires that a program has to be primary memory resident to execute. Also, a user requires to revisit his programs often during its evolution. However, due to the fact that primary memory is volatile, a user needs to store his program in some non-volatile store. All computers provide a non-volatile secondary memory available as an online storage. Programs and files may be disk resident and downloaded whenever their execution is required. Therefore, some form of memory management is needed at both primary and secondary memory levels.

Secondary memory may store program scripts, executable process images and data files. It may store applications, as well as, system programs. In fact, a good part of all OS, the system programs which provide services (the utilities for instance) are stored in the secondary memory. These are requisitioned as needed.

The main motivation for management of main memory comes from the support for multi-programming. Several executables processes reside in main memory at any given time. In other words, there are several programs using the main memory as their address space. Also, programs move into, and out of, the main memory as they terminate, or get suspended for some IO, or new executables are required to be loaded in main memory.

So, the OS has to have some strategy for main memory management. In this chapter we shall discuss the management issues and strategies for both main memory and secondary memory.

4.1 Main Memory Management

Let us begin by examining the issues that prompt the main memory management.

- **Allocation:** First of all the processes that are scheduled to run must be resident in the memory. These processes must be allocated space in main memory.
- **Swapping, fragmentation and compaction:** If a program is moved out or terminates, it creates a hole, (i.e. a contiguous unused area) in main memory. When a new process is to be moved in, it may be allocated one of the available holes. It is quite possible that main memory has far too many small holes at a certain time. In such a situation none of these holes is really large enough to be allocated to a new process that may be moving in. The main memory is too

fragmented. It is, therefore, essential to attempt compaction. Compaction means OS re-allocates the existing programs in contiguous regions and creates a large enough free area for allocation to a new process.

- **Garbage collection:** Some programs use dynamic data structures. These programs dynamically use and discard memory space. Technically, the deleted data items (from a dynamic data structure) release memory locations. However, in practice the OS does not collect such free space immediately for allocation. This is because that affects performance. Such areas, therefore, are called garbage. When such garbage exceeds a certain threshold, the OS would not have enough memory available for any further allocation. This entails compaction (or garbage collection), without severely affecting performance.
- **Protection:** With many programs residing in main memory it can happen that due to a programming error (or with malice) some process writes into data or instruction area of some other process. The OS ensures that each process accesses only to its own allocated area, i.e. each process is protected from other processes.
- **Virtual memory:** Often a processor sees a large logical storage space (a virtual storage space) though the actual main memory may not be that large. So some facility needs to be provided to translate a logical address available to a processor into a physical address to access the desired data or instruction.
- **IO support**: Most of the block-oriented devices are recognized as specialized files. Their buffers need to be managed within main memory alongside the other processes. The considerations stated above motivate the study of main memory management.

One of the important considerations in locating an executable program is that it should be possible to relocate it any where in the main memory. We shall dwell upon the concept of relocation next.

4.2 Memory Relocation Concept

Relocation is an important concept. To understand this concept we shall begin with a linear map (one-dimensional view) of main memory. If we know an address we can fetch its contents. So, a process residing in the main memory, we set the program counter to an absolute address of its first instruction and can initiate its run. Also, if we know the locations of data then we can fetch those too. All of this stipulates that we know the

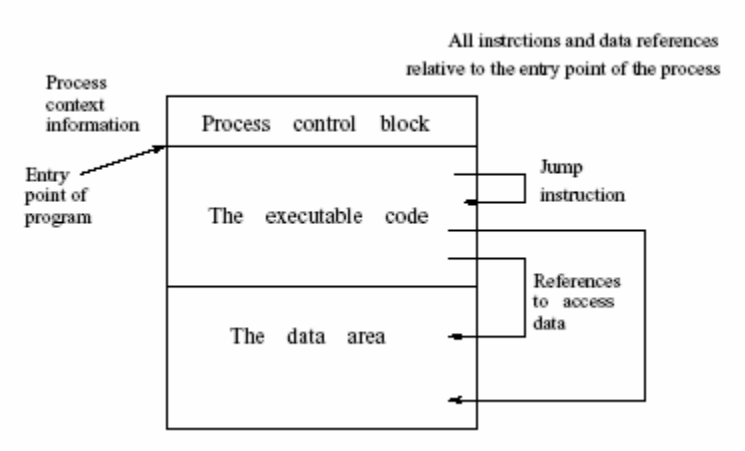

Figure 4.1: The relocation concept.

absolute addresses for a program, its data and process context etc. This means that we can load a process with only absolute addresses for instructions and data, only when those specific addresses are free in main memory. This would mean we loose flexibility with regard to loading a process. For instance, we cannot load a process, if some other process is currently occupying that area which is needed by this process. This may happen even though we may have enough space in the memory. To avoid such a catastrophe, processes are generated to be relocatable. In Figure 4.1 we see a process resident in main memory.

Initially, all the addresses in the process are relative to the start address. With this flexibility we can allocate any area in the memory to load this process. Its instruction, data, process context (process control block) and any other data structure required by the process can be accessed easily if the addresses are relative. This is most helpful when processes move in and out of main memory. Suppose a process created a hole on moving out. In case we use non-relocatable addresses, we have the following very severe problem.

When the process moves back in, that particular hole (or area) may not be available any longer. In case we can relocate, moving a process back in creates no problem. This is so because the process can be relocated in some other free area. We shall next examine the linking and loading of programs to understand the process of relocation better.

4.2.1 Compiler Generated Bindings

The advantage of relocation can also be seen in the light of binding of addresses to variables in a program. Suppose we have a program variable *x* in a program *P*. Suppose

the compiler allocated a fixed address to *x*. This address allocation by the compiler is called binding. If *x* is bound to a fixed location then we can execute program *P* only when *x* could be put in its allocated memory location. Otherwise, all address references to *x* will be incorrect.

If, however, the variable can be assigned a location relative to an assumed origin (or first address in program *P*) then, on relocating the program's origin anywhere in main memory, we will still be able to generate a proper relative address reference for *x* and execute the program. In fact, compilers generate relocatable code. In the next section we describe the linking, loading, and relocation of object code in greater detail.

4.3 Linking and Loading Concepts

In Figure 4.2 we depict the three stages of the way a HLL program gets processed.

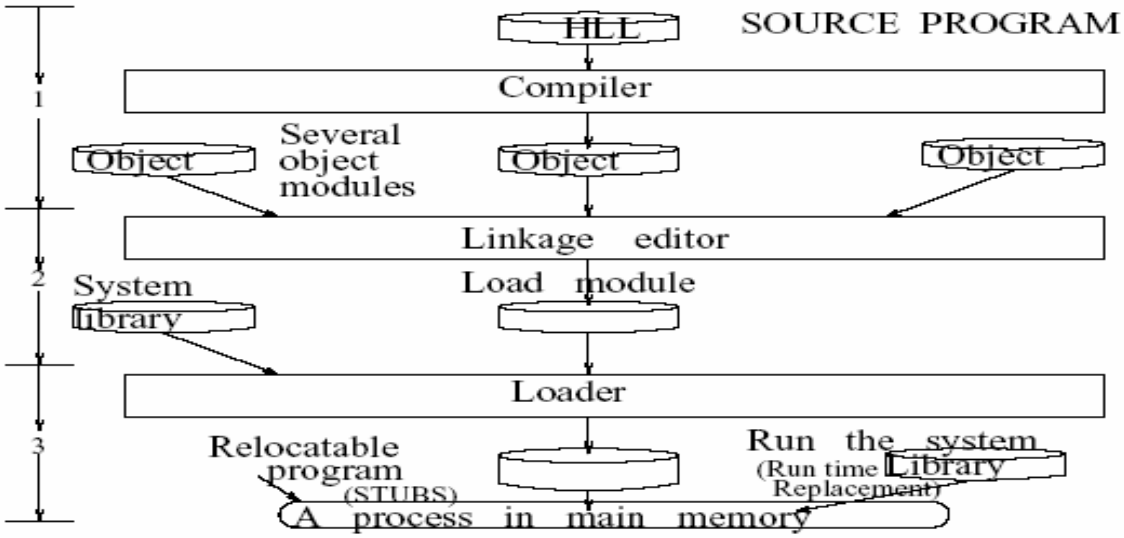

Figure 4.2: Linking and loading.

The three stages of the processing are:

- Stage 1: In the first stage the HLL source program is compiled and an object code is produced. Technically, depending upon the program, this object code may by itself be sufficient to generate a relocatable process. However many programs are compiled in parts, so this object code may have to link up with other object modules. At this stage the compiler may also insert stub at points where run time library modules may be linked.
- Stage 2: All those object modules which have sufficient linking information (generated by the compiler) for static linking are taken up for linking. The linking

editor generates a relocatable code. At this stage, however, we still do not replace the stubs placed by compilers for a run time library link up.

- Stage3: The final step is to arrange to make substitution for the stubs with run time library code which is a relocatable code.

When all the three stages are completed we have an executable. When this executable is resident in the main memory it is a runnable process.

Recall our brief discussion in the previous section about the binding of variables in a program. The compiler uses a symbol table to generate addresses. These addresses are not bound, i.e. these do not have absolute values but do have information on sizes of data. The binding produced at compile time is generally relative. Some OSs support a linking loader which translates the relative addresses to relocatable addresses. In any event, the relocatable process is finally formed as an output of a loader.

4.4 Process and Main Memory Management

Once processes have been created, the OS organizes their execution. This requires interaction between process management and main memory management. To understand this interaction better, we shall create a scenario requiring memory allocations. For the operating environment we assume the following:

- A uni-processor, multi-programming operation.
- A Unix like operating system environment.

With a Unix like OS, we can assume that main memory is partitioned in two parts. One part is for user processes and the other is for OS. We will assume that we have a main memory of 20 units (for instance it could be 2 or 20 or 200 MB). We show the requirements and time of arrival and processing requirements for 6 processes in Table 4.1.

	P1	P2	P3	P4	P5	P6
Time of arrival	0	0	0	0	10	15
Processing time required	8	5	20	12	10	5
Memory required	3 units	7 units	2 units	4 units	2 units	2 units

Table 4.1: The given data.

We shall assume that OS requires 6 units of space. To be able to compare various policies, we shall repeatedly use the data in Table 4.1 for every policy option. In the next section we discuss the first fit policy option.

With these requirements we can now trace the emerging scenario for the given data. We shall assume round robin allocation of processor time slots with no context switching

Time units	Programs in Main memory	Programs on disk	Holes with sizes	Figure 4.3	Comments
0	P1, P2, P3	P4	H1=2	(a)	P4 requires more space than H1
5	P1, P4, P3		H1=2; H2=3	(b)	P2 is finished P4 is loaded Hole H2 is created
8	P4, P3		H1=2; H2=3; H3=3	(c)	New hole created
10	P4, P3	P5			P5 arrives
10+	P5, P4, P3		H1=2; H2=3 H3=1	(d)	P5 is allocated P1's space
15	P5, P4, P3	P6	H1=2; H2=3; H3=1		P6 has arrived
15+	P5, P4, P6, P3		H1=2; H2=1; H3=1	(e)	P6 is allocated

Table 4.2: FCFS memory allocation.

over-heads. We shall trace the events as they occur giving reference to the corresponding part in Table 4.2. This table also shows a memory map as the processes move in and out of the main memory.

4.5 The First Fit Policy: Memory Allocation

In this example we make use of a policy called first fit memory allocation policy. The first fit policy suggests that we use the first available hole, which is large enough to accommodate an incoming process. In Figure 4.3, it is important to note that we are following first-come first-served (process management) and first fit (memory allocation) policies. The process index denotes its place in the queue. As per first-come first-served policies the queue order determines the order in which the processes are allocated areas.

In addition, as per first-fit policy allocation we scan the memory always from one end and find the first block of free space which is large enough to accommodate the incoming process.

In our example, initially, processes P1, P2, P3 and P4 are in the queue. The allocations for processes P1, P2, P3 are shown in 4.3(a). At time 5, process P2 terminates. So, process P4 is allocated in the hole created by process P2. This is shown at 4.3(b) in the figure. It still leaves a hole of size 3. Now on advancing time further we see that at time 8, process P1 terminates. This creates a hole of size 3 as shown at 4.3(c) in the figure.

This hole too is now available for allocation. We have 3 holes at this stage. Two of these 3 holes are of size 3 and one is of size 2. When process P5 arrives at time 10, we look for the first hole which can accommodate it. This is the one created by the departure of process P1. Using the first-fit argument this is the hole allocated to process P5 as shown in Figure 4.3(d). The final allocation status is shown in Figure 4.3. The first-fit allocation policy is very easy to implement and is fast in execution.

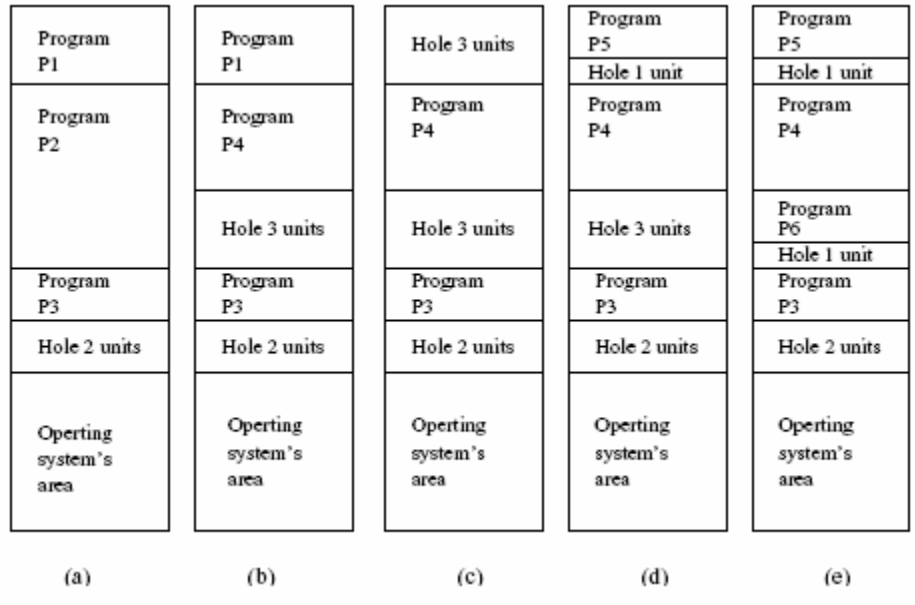

Figure 4.3: First-fit policy allocation.

4.6 The Best Fit Policy: Memory Allocation

The main criticism of first-fit policy is that it may leave many smaller holes. For instance, let us trace the allocation for process P5. It needs 2 units of space. At the time it moves into the main memory there is a hole with 2 units of space. But this is the last hole when we scan the main memory from the top (beginning). The first hole is 3 units. Using the first-fit policy process P5 is allocated this hole. So when we used this hole we also created a still smaller hole. Note that smaller holes are less useful for future allocations.

In the best-fit policy we scan the main memory for all the available holes. Once we have information about all the holes in the memory then we choose the one which is closest to the size of the requirement of the process. In our example we allocate the hole with size 2 as there is one available. Table 4.3 follows best-fit policy for the current example.

Also, as we did for the previous example, we shall again assume round-robin allocation of the processor time slots. With these considerations we can now trace the possible emerging scenario.

In Figure 4.4, we are following first-come first-served (process management) and best fit (memory allocation) policies. The process index denotes its place in the queue. Initially, processes P1, P2, P3 and P4 are in the queue. Processes P1, P2 and P3 are allocated as

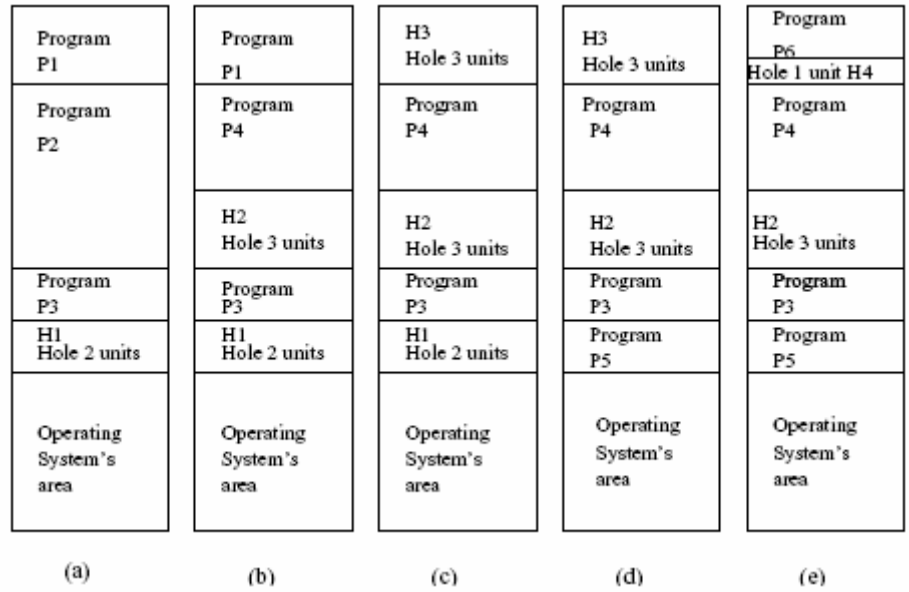

Figure 4.4: Best-fit policy allocation

shown in Figure 4.4(a). At time 5, P2 terminates and process P4 is allocated in the hole so created. This is shown in Figure 4.4(b). This is the best fit. It leaves a space of size 3 creating a new hole. At time 8, process P1 terminates. We now have 3 holes. Two of these holes are of size 3 and one is of size 2. When process P5 arrives at time 10, we look for a hole whose size is nearest to 2 and can accommodate P5. This is the last hole.

Clearly, the best-fit (and also the worst-fit) policy should be expected to be slow in execution. This is so because the implementation requires a time consuming scan of all of main memory. There is another method called the next-fit policy. In the next-fit method the search pointer does not start at the top (beginning), instead it begins from where it ended during the previous search. Like the first-fit policy it locates the next first-fit hole that can be used. Note that unlike the first-fit policy the next-fit policy can be expected to distribute small holes uniformly in the main memory. The first-fit policy would have a

tendency to create small holes towards the beginning of the main memory scan. Both first-fit and next-fit methods are very fast and easy to implement.

In conclusion, first-fit and next-fit are the fastest and seem to be the preferred methods. One of the important considerations in main memory management is: how should an OS allocate a chunk of main memory required by a process. One simple approach would be to somehow create partitions and then different processes could reside in different partitions. We shall next discuss how the main memory partitions may be created.

Time units	Programs in Main memory	Programs on disk	Holes with sizes	Figure 4.4	Comments
0	P1, P2, P3	P4	H1=2	(a)	P4 requires more space than H1
5	P1, P4, P3		H1=2; H2=3	(b)	P2 is finished P4 is loaded Hole H2 is created
8	P4, P3		H1=2; H2=3; H3=3	(c)	Creates a new hole
10	P4, P3	P5			P5 arrives
10+	P4, P3, P5		H2=3; H3=3	(d)	P5 is allocated the best fit hole
15	P4, P3, P5	P6	H2=3; H3=3		P6 arrives
15+	P6, P4, P3, P5		H2=3; H4=1	(e)	P6 takes the hole left by P1

Table 4.3: Best-fit policy memory allocation.

4.7 Fixed and Variable Partitions

In a fixed size partitioning of the main memory all partitions are of the same size. The memory resident processes can be assigned to any of these partitions. Fixed sized partitions are relatively simple to implement. However, there are two problems. This scheme is not easy to use when a program requires more space than the partition size. In this situation the programmer has to resort to overlays. Overlays involve moving data and program segments in and out of memory essentially reusing the area in main memory. The second problem has to do with internal fragmentation. No matter what the size of the process is, a fixed size of memory block is allocated as shown in Figure 4.5(a). So there will always be some space which will remain unutilized within the partition.

In a variable-sized partition, the memory is partitioned into partitions with different sizes. Processes are loaded into the size nearest to its requirements. It is easy to always ensure the best-fit. One may organize a queue for each size of the partition as shown in the Figure 4.5(b). With best-fit policy, variable partitions minimize internal fragmentation.

However, such an allocation may be quite slow in execution. This is so because a process may end up waiting (queued up) in the best-fit queue even while there is space available

elsewhere. For example, we may have several jobs queued up in a queue meant for jobs that require 1 unit of memory, even while no jobs are queued up for jobs that require say 4 units of memory.

Both fixed and dynamic partitions suffer from external fragmentation whenever there are partitions that have no process in it. One of techniques that have been used to keep both internal and external fragmentations low is dynamic partitioning. It is basically a variable partitioning with a variable number of partitions determined dynamically (i.e. at run time).

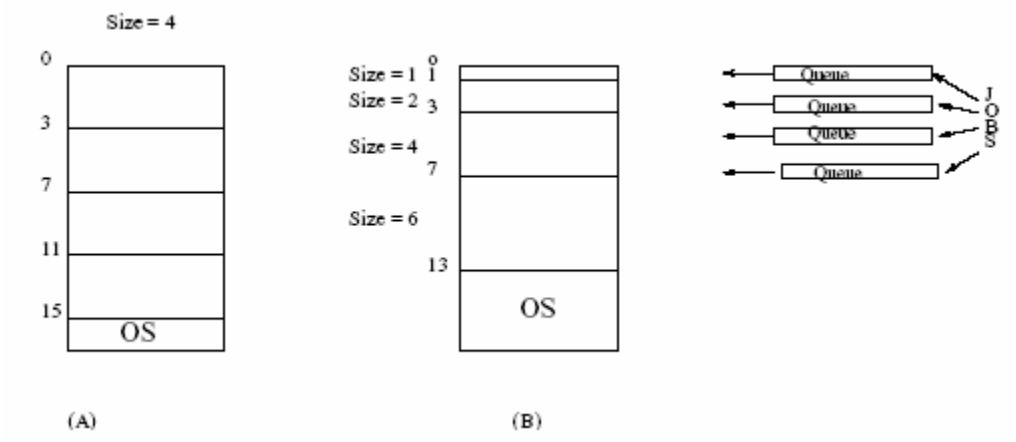

Figure 4.5: Fixed and variable sized partitions.

Such a scheme is difficult to implement. Another scheme which falls between the fixed and dynamic partitioning is a buddy system described next.

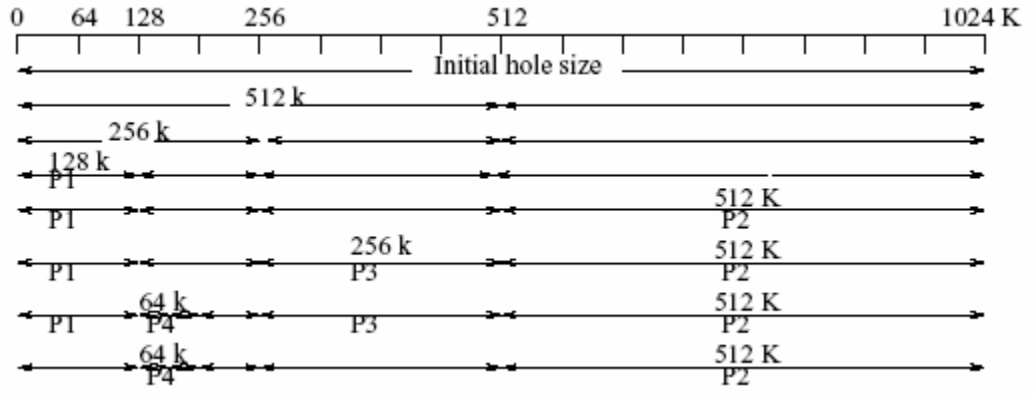

Figure 4.6: Buddy system allocation.

The Buddy system of partitioning: The buddy system of partitioning relies on the fact that space allocations can be conveniently handled in sizes of power of 2. There are two ways in which the buddy system allocates space. Suppose we have a hole which is the closest power of two. In that case, that hole is used for allocation. In case we do not have

that situation then we look for the next power of 2 hole size, split it in two equal halves and allocate one of these. Because we always split the holes in two equal sizes, the two are \buddies". Hence, the name buddy system. We shall illustrate allocation using a buddy system. We assume that initially we have a space of 1024 K. We also assume that processes arrive and are allocated following a time sequence as shown in figure 4.6.

With 1024 K or (1 M) storage space we split it into buddies of 512 K, splitting one of them to two 256 K buddies and so on till we get the right size. Also, we assume scan of memory from the beginning. We always use the first hole which accommodates the process. Otherwise, we split the next sized hole into buddies. Note that the buddy system begins search for a hole as if we had a fixed number of holes of variable sizes but turns into a dynamic partitioning scheme when we do not find the best-fit hole. The buddy system has the advantage that it minimizes the internal fragmentation. However, it is not popular because it is very slow. In Figure 4.6 we assume the requirements as (P1:80 K); (P2:312 K); (P3:164 K); (P4:38 K). These processes arrive in the order of their index and P1 and P3 finish at the same time.

4.8 Virtual Storage Space and Main Memory Partitions

Programming models assume the presence of main memory only. Therefore, ideally we would like to have an unlimited (infinite) main memory available. In fact, an unlimited main memory shall give us a Turing machine capability. However, in practice it is infeasible. So the next best thing is attempted. CPU designers support and generate a very large logical addressable space to support programming concerns. However, the directly addressable main memory is limited and is quite small in comparison to the logical addressable space. The actual size of main memory is referred as the physical memory. The logical addressable space is referred to as virtual memory. The notion of virtual memory is a bit of an illusion. The OS supports and makes this illusion possible. It does so by copying chunks of disk memory into the main memory as shown in Figure 4.7. In other words, the processor is fooled into believing that it is accessing a large addressable space. Hence, the name virtual storage space. The disk area may map to the virtual space requirements and even beyond.

Besides the obvious benefit that virtual memory offers a very large address space, there is one other major benefit derived from the use of virtual storage. We now can have many more main memory resident active processes. This can be explained as follows. During

much of the lifetime of its execution, a process operates on a small set of instructions within a certain neighborhood. The same applies for the data as well. In other words a process makes use of a very small memory area for doing most of the instructions and making references to the data. As explained in Section 4.9, this is primarily due to the locality of reference. So, technically, at any time we need a very small part of a process to really be memory resident. For a moment, let us suppose that this small part is only 1/10th of the process's overall requirements. Note in that case, for the same size of physical main memory, we can service 10 times as many memory resident programs. The next question then is how do we organize and allocate these small chunks of often required areas to be in memory. In fact, this is where paging and segmentation become important. In this context we need to understand some of the techniques of partitioning of main memory into pages or segments.

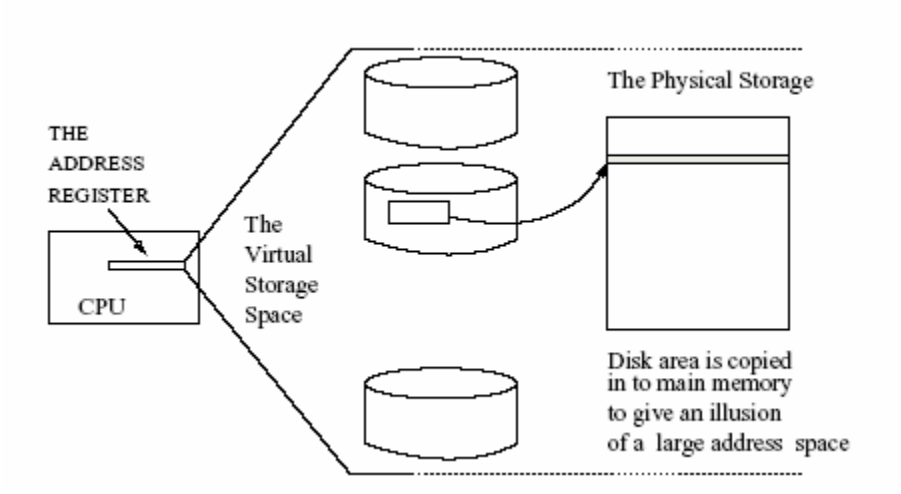

Figure 4.7: Virtual storage concept.

In addition, we need to understand virtual addressing concepts with paging and/or segmentation. We begin with some simple techniques of partitioning both these memories and management of processes.

4.9 Virtual Memory: Paging

In some sense, paging of virtual memory has an underlying mechanism which resembles reading of a book. When we read a book we only need to open only the current page to read. All the other pages are not visible to us. In the same manner, we can argue that even when we may have a large online main memory available, the processor only needs a small set of instructions to execute at any time. In fact, it often happens that for a brief

while, all the instructions which the processor needs to execute are within a small proximity of each other. That is like a page we are currently reading in a book. Clearly, this kind of situation happens quite frequently.

Essentially virtual memory is a large addressable space supported by address generating mechanisms in modern CPUs. Virtual address space is much larger than the physical main memory in a computer system. During its execution, a process mostly generates instruction and data references from within a small range. This is referred to as the locality of reference. Examples of locality of reference abound. For instance, we have locality of reference during execution of a *for* or *while* loop, or a call to a procedure. Even in a sequence of assignment statements, the references to instructions and data are usually within a very small range. Which means, during bursts of process execution, only small parts of all of the instruction and data space are needed, i.e. only these parts need be in the main memory. The remaining process, instructions and data, can be anywhere in the virtual space (i.e. it must remain accessible by CPU but not necessarily in main memory). If we are able to achieve that, then we can actually follow a schedule, in which we support a large address space and keep bringing in that part of process which is needed. This way we can comfortably support (a) multi-programming (b) a large logical addressable space giving enormous freedom to a programmer. Note, however, that this entails mapping of logical addresses into physical address space. Such a mapping assures that the instruction in sequence is fetched or the data required in computation is correctly used.

If this translation were to be done in software, it would be very slow. In fact, nowadays this address translation support is provided by hardware in CPUs. Paging is one of the popular memory management schemes to implement such virtual memory management schemes. OS software and the hardware address translation between them achieve this.

4.9.1 Mapping the Pages

Paging stipulates that main memory is partitioned into frames of sufficiently small sizes. Also, we require that the virtual space is divided into pages of the same size as the frames. This equality facilitates movement of a page from anywhere in the virtual space (on disks) to a frame anywhere in the physical memory. The capability to map “any page" to “any frame" gives a lot of flexibility of operation as shown in Figure 4.8

Division of main memory into frames is like fixed partitioning. So keeping the frame size small helps to keep the internal fragmentation small. Often, the page to frame movement is determined by a convenient size (usually a power of two) which disks also use for their own DMA data transfer. The usual frame size is 1024 bytes, though it is not unusual to have 4 K frame sizes as well. Paging supports multi-programming. In general there can be many processes in main memory, each with a different number of pages. To that extent, paging is like dynamic variable partitioning.

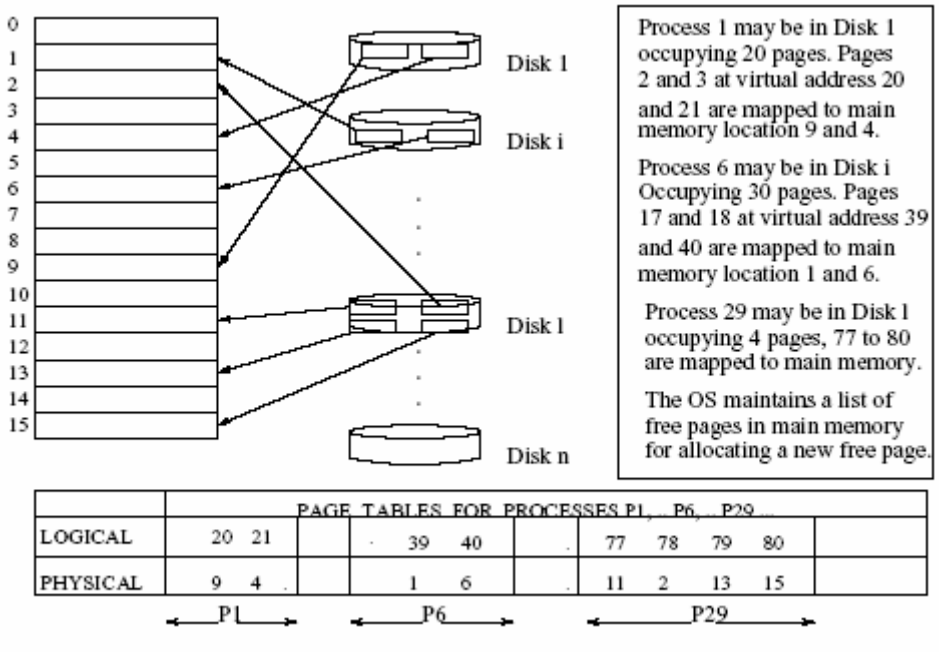

Figure 4.8: Paging implementation.

4.10 Paging: Implementation

Paging implementation requires CPU (HW) and OS (SW) support. In Figure 4.8, we assume presence of three active processes. These processes need to have their pages mapped to the main memory page frames. The OS maintains a page table for every process to translate its logical to physical addresses. The page table may itself be resident in main memory.

For a process, which is presently active, there are a number of pages that are in the main memory. This set of pages (being used by the process) forms its resident set. With the locality of reference generally observed, most of the time, the processes make reference within the resident set. We define the set of pages needed by a process at any time as the working set. The OS makes every effort to have the resident set to be the same as the

working set. However, it does happen (and happens quite often), that a page required for continuing the process is not in the resident set. This is called a page fault. In normal course of operation, though whenever a process makes virtual address reference, its page table is looked up to find if that page is in main memory. Often it is there. Let us now suppose that the page is not in main memory, i.e. a page fault has occurred. In that case, the OS accesses the required page on the disk and loads it in a free page frame. It then makes an entry for this page in process page table. Similarly, when a page is swapped out, the OS deletes its entry from the page table. Sometimes it may well happen that all the page frames in main memory are in use. If a process now needs a page which is not in main memory, then a page must be forced out to make way for the new page. This is done using a page replacement policy discussed next.

4.11 Paging: Replacement

Page replacement policies are based on the way the processes use page frames. In our example shown in Figure 4.9, process P29 has all its pages present in main memory. Process P6 does not have all its pages in main memory. If a page is present we record 1 against its entry. The OS also records if a page has been referenced to read or to write. In both these cases a reference is recorded. If a page frame is written into, then a modified bit is set. In our example frames 4, 9, 40, 77, 79 have been referenced and page frames 9 and 13 have been modified. Sometimes OS may also have some information about protection using *rwe* information. If a reference is made to a certain virtual address

Page tables for processes P1, .. P6, .. P29 ...

	P1	P6	P29
Logical	20 21	39 40	77 78 79 80
Physical	9 4	1 6	11 2 13 15
Present	0 1 1 0	0 0 1 1 0	1 1 1 1 1 1 1
Referenced	1 1	0 1	1 0 1 0
Modified	1 0	0 0	0 0 1 0
Protection	rw– rw–		r–– r––

Figure 4.9: Replacement policy.

and its corresponding page is not present in main memory, then we say a page fault has occurred. Typically, a page fault is followed by moving in a page. However, this may require that we move a page out to create a space for it. Usually this is done by using an appropriate page replacement policy to ensure that the throughput of a system does not

suffer. We shall later see how a page replacement policy can affect performance of a system.

4.11.1 Page Replacement Policy

Towards understanding page replacement policies we shall consider a simple example of a process P which gets an allocation of four pages to execute. Further, we assume that the OS collects some information (depicted in Figure 4.10) about the use of these pages as this process progresses in execution. Let us examine the information depicted in figure 4.10 in some detail to determine how this may help in evolving a page replacement policy. Note that we have the following information available about P.

1. The time of arrival of each page. We assume that the process began at some time with value of time unit 100. During its course of progression we now have pages that have been loaded at times 112, 117 119, and 120.
2. The time of last usage. This indicates when a certain page was last used. This entirely depends upon which part of the process P is being executed at any time.

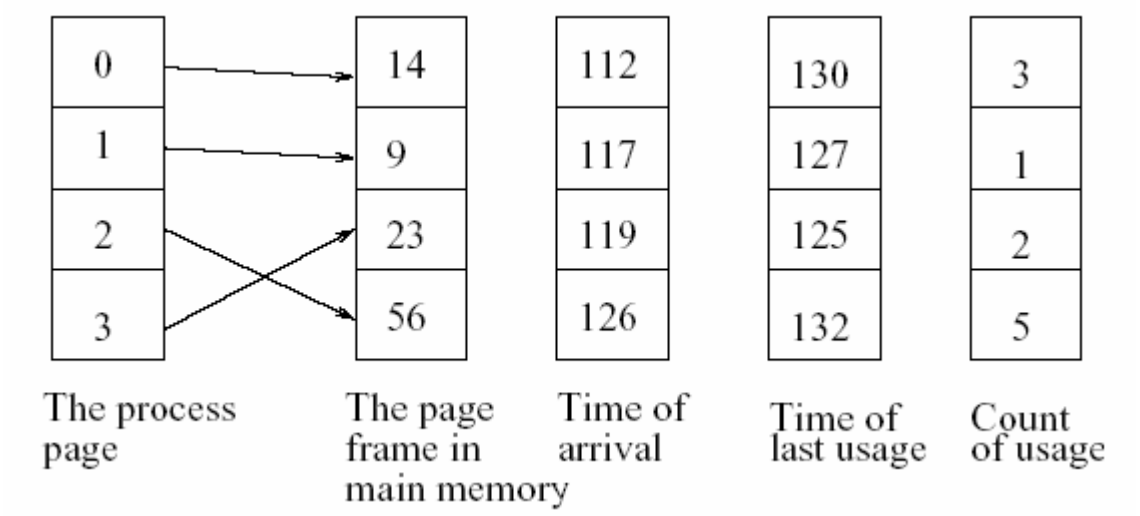

Figure 4.10: Information on page usage policy.

3. The frequency of use. We have also maintained the frequency of use over some fixed interval of time T in the immediate past. This clearly depends upon the nature of control flow in process P.

As an example we may say that page located at 23 which was installed at time 119, was last used at time unit 125 and over the time period T the process P made two references to it. Based on the above pieces of information if we now assume that at time unit 135 the process P experiences a page-fault, what should be done. Based on the choice of the

policy and the data collected for P, we shall be able to decide which page to swap out to bring in a new page.

FIFO policy: This policy simply removes pages in the order they arrived in the main memory. Using this policy we simply remove a page based on the time of its arrival in the memory. Clearly, use of this policy would suggest that we swap page located at 14 as it arrived in the memory earliest.

LRU policy: LRU expands to least recently used. This policy suggests that we re- move a page whose last usage is farthest from current time. Note that the current time is 135 and the least recently used page is the page located at 23. It was used last at time unit 125 and every other page is more recently used. So, page 23 is the least recently used page and so it should be swapped if LRU replacement policy is employed.

NFU policy: NFU expands to not frequently used. This policy suggests to use the criterion of the count of usage of page over the interval T. Note that process P has not made use of page located at 9. Other pages have a count of usage like 2, 3 or even 5 times. So the basic argument is that these pages may still be needed as compared to the page at 9. So page 9 should be swapped.

Let us briefly discuss the merits of choices that one is offered. FIFO is a very simple policy and it is relatively easy to implement. All it needs is the time of arrival. However, in following such a policy we may end up replacing a page frame that is referred often during the lifetime of a process. In other words, we should examine how useful a certain page is before we decide to replace it. LRU and NFU policies are certainly better in that regard but as is obvious we need to keep the information about the usage of the pages by the process. In following the not frequently used (NFU) and least recently used (LRU) page replacement policies, the OS needs to define *recency.* As we saw recency is defined as a fixed time interval proceeding the current time. With a definition of recency, we can implement the policy framework like least recently used (LRU). So one must choose a proper interval of time. Depending upon the nature of application environment and the work load a choice of duration of recency will give different throughput from the system. Also, this means that the OS must keep a tab on the pages which are being used and how often these are in use. It is often the case that the most recently used pages are likely to be the ones used again. On the whole one can sense that the LRU policy should be statistically better than FIFO.

A more advanced technique of page replacement policy may look-up the likely future references to pages. Such a policy frame would require use of some form of predictive techniques. In that case, one can prevent too many frequent replacements of pages which prevents thrashing as discussed in the subsection. 4.11.2.

Let us for now briefly pay our attention to page references resulting in a page hit and a page miss. When we find that a page frame reference is in the main memory then we have a page hit and when page fault occurs we say we have a page miss. As is obvious from the discussion, a poor choice of policy may result in lot of page misses. We should be able to determine how it influences the throughput of a system. Let us assume that we have a system with the following characteristics.

- Time to look-up page table: 10 time units.
- Time to look-up the information from a page frame (case of a page hit): 40 time units.
- Time to retrieve a page from disk and load it and finally access the page frame (case of a page miss): 190 time units.

Now let us consider the following two cases when we have 50% and 80% page hits. We shall compute the average time to access.

- Case 1: With 50% page hits the average access time is ((10+40) * 0:5) + (10+190) * 0:5 =125 time units.
- Case 2: With 80% page hits the average access time is (10+40) * 0:8) + (10+190) * 0:2 = 80 time units.

Clearly, the case 2 is better. The OS designers attempt to offer a page replacement policy which will try to minimize the page miss. Also, sometimes the system programmers have to tune an OS to achieve a high efficacy in performance by ensuring that page miss cases are within some tolerable limits. It is not unusual to be able to achieve over 90% page hits when the application profile is very well known.

There is one other concern that may arise with regard to page replacement. It may be that while a certain process is operative, some of the information may be often required. These may be definitions globally defined in a program, or some terminal related IO information in a monitoring program. If this kind of information is stored in certain pages then these have to be kept at all times during the lifetime of the process. Clearly, this requires that we have these pages identified. Some programming environments allow

directives like keep to specify such information to be available at all the time during the lifetime of the process. In Windows there is a keep function that allows one to specify which programs must be kept at all the time. The Windows environment essentially uses the keep function to load TSR (terminate and stay resident) programs to be loaded in the memory 1. Recall, earlier we made a reference to thrashing which arises from the overheads generated from frequent page replacement. We shall next study that.

4.11.2 Thrashing

Suppose there is a process with several pages in its resident set. However, the page replacement policy results in a situation such that two pages alternatively move in and out of the resident set. Note that because pages are moved between main memory and disk, this has an enormous overhead. This can adversely affect the throughput of a system. The drop in the level of system throughput resulting from frequent page replacement is called thrashing. Let us try to comprehend when and how it manifests. Statistically, on introducing paging we can hope to enhance multi-programming as well as locality of reference. The main consequence of this shall be enhanced processor utilization and hence, better throughput. Note that the page size influences the number of pages and hence it determines the number of resident sets we may support. With more programs in main memory or more pages of a program we hope for better locality of reference. This is seen to happen (at least initially) as more pages are available. This is because, we may have more effective locality of reference as well as multi-programming. However, when the page size becomes too small we may begin to witness more page-faults.

Incidentally, a virus writer may employ this to mount an attack. For instance, the keep facility may be used to have a periodic display of some kind on the victim's screen. More page-faults would result in more frequent disk IO. As disk IO happens more often the throughput would drop. The point when this begins to happen, we say thrashing has occurred. In other words, the basic advantage of higher throughput from a greater level of utilization of processor and more effective multi-programming does not accrue any more. When the advantage derived from locality of reference and multi-programming begins to vanish, we are at the point when thrashing manifests. This is shown in Figure 4.11.

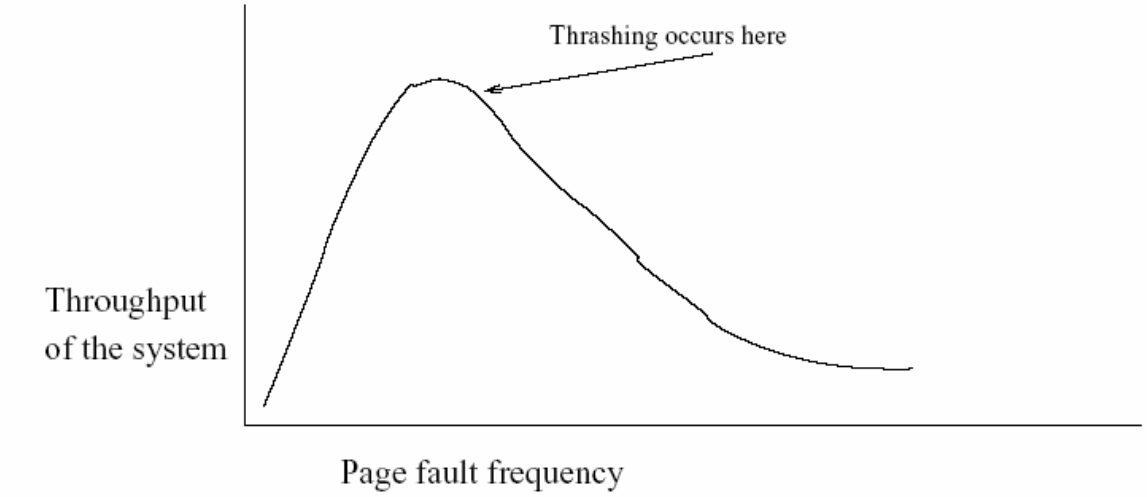

Figure 4.11: Thrashing on numerous page fault.

4.12 Paging: HW support

Recall we emphasized that we need HW within CPU to support paging. The CPU generates a logical address which must get translated to a physical address. In Figure 4.12 we indicate the basic address generation and translation.

Let us trace the sequence of steps in the generation of address.

- The process generates a logical address. This address is interpreted in two parts.
- The first part of the logical address identifies the virtual page.
- The second part of the logical address gives the offset within this page.
- The first part is used as an input to the page table to find out the following:
  - * Is the page in the main memory?
  - * What is the page frame number for this virtual page?
- The page frame number is the first part of the physical memory address.
- The offset is the second part of the correct physical memory location.

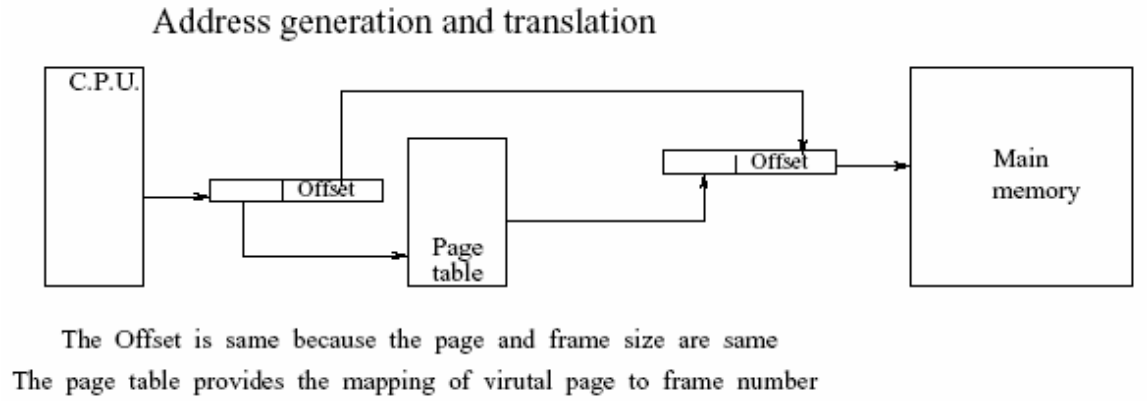

Figure 4.12: Hardware support for paging.

If the page is not in the physical memory, a page-fault is generated. This is treated as a trap. The trap then suspends the regular sequence of operations and fetches the required page from the disk into main memory.

We next discuss a relatively simple extension of the basic paging scheme with hardware support. This scheme results in considerable improvement in page frame access.

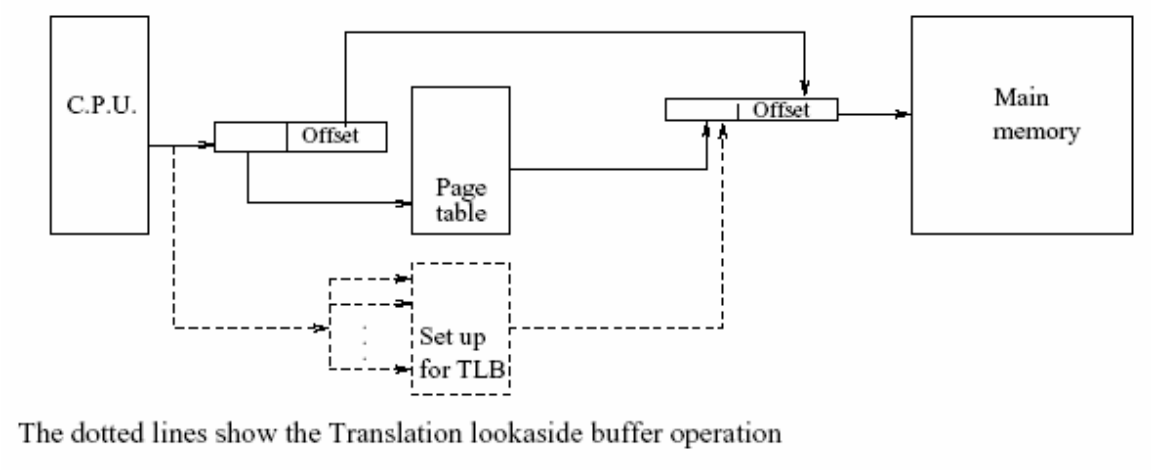

Figure 4.13: Paging with translation look-aside buffer.

4.12.1 The TLB scheme

The basic idea in the translation look-aside buffer access is quite simple. The scheme is very effective in improving the performance of page frame access. The scheme employs a cache buffer to keep copies of some of the page frames in a cache buffer. This buffer is also interrogated for the presence of page frame copy. Note that a cache buffer is implemented in a technology which is faster than the main memory technology. So, a retrieval from the cache buffer is faster than that from the main memory. The hardware signal which looks up the page table is also used to look up (with address translation) to check if the cache buffer on a side has the desired page. This nature of look-up explains why this scheme is called Translation Look-aside Buffer (TLB) scheme. The basic TLB buffering scheme is shown in Figure 4.13. Note that the figure replicates the usual hardware support for page table look-up. So, obviously the scheme cannot be worse than the usual page table look-up schemes. However, since a cache buffer is additionally maintained to keep some of the frequently accessed pages, one can expect to achieve an improvement in the access time required for those pages which obtain a page hit for presence in the buffer. Suppose we wish to access page frame p. The following three possibilities may arise:

1. Cache presence: There is a copy of the page frame p. In this case it is procured from the look-aside buffer which is the cache.
2. Page table presence: The cache does not have a copy of the page frame p, but page table access results in a page hit. The page is accessed from the main memory.
3. Not in page table: This is a case when the copy of the page frame is neither in the cache buffer nor does it have an entry in the page table. Clearly, this is a case of page-fault. It is handled exactly as the page-fault is normally handled.

Note that if a certain page frame copy is available in the cache then the cache look-up takes precedence and the page frame is fetched from the cache instead of fetching it from the main memory. This obviously saves time to access the page frame. In the case the page hit occurs for a page not in cache then the scheme ensures its access from the main memory. So it is at least as good as the standard paging scheme with a possibility of improvement whenever a page frame copy is in cache buffer.

4.12.2 Some Additional Points

Since page frames can be loaded anywhere in the main memory, we can say that paging mechanism supports dynamic relocation. Also, there are other schemes like multi-level page support systems which support page tables at multiple levels of hierarchy. In addition, there are methods to identify pages that may be shared amongst more than one process. Clearly, such shareable pages involve additional considerations to maintain consistency of data when multiple processes try to have read and write access. These are usually areas of research and beyond the scope of this book.

4.13 Segmentation

Like paging, segmentation is also a scheme which supports virtual memory concept. Segmentation can be best understood in the context of a program's storage requirements. One view could be that each part like its code segment, its stack requirements (of data, nested procedure calls), its different object modules, etc. has a contiguous space. This space would then define a process's space requirement as an integrated whole (or complete space). As a view, this is very uni-dimensional.

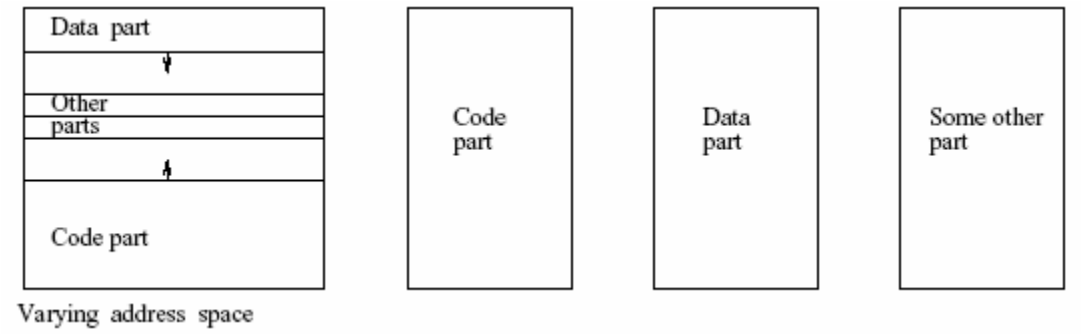

Figure 4.14: Segmentation scheme: A two dimensional view.

In using segmentation, one recognizes various segments such as the stack, object code, data area etc. Each segment has requirements that vary over time. For instance, stacks grow and shrink the memory requirements of object and data segments may change during the lifetime of the process. This may depend on which functions have been called and are currently active. It is, therefore, best to take a two-dimensional view of a process's memory requirement. In this view, each of the process segments has an opportunity to acquire a variable amount of space over time. This ensures that one area does not run into the space of any other segment. The basic scheme is shown in Figure 4.14. The implementation of segmentation is similar to paging, except that we now have segment table (in place of a page table) look-ups to identify addresses in each of the segments. HW supports a table look-up for a segment and an offset within that segment. We may now compare paging with segmentation.

- Paging offers the simplest mechanism to effect virtual addressing.
- While paging suffers from internal fragmentation, segmentation suffers from external fragmentation.
- One of the advantages segmentation clearly offers is separate compilation of each segment with a view to link up later. This has another advantage. A user may develop a code segment and share it amongst many applications. He generates the required links at the time of launching the application. However, note that this also places burden on the programmer to manage linking. To that extent paging offers greater transparency in usage.
- In paging, a process address space is linear. Hence, it is uni-dimensional. In a segment based scheme each procedure and data segment has its own virtual space mapping. Thus the segmentation assures a much greater degree of protection.

- In case a program's address space fluctuates considerably, paging may result in frequent page faults. Segmentation does not suffer from such problems.
- Paging partitions a program and data space uniformly and is, therefore, much simpler to manage. However, one cannot easily distinguish data space from program space in paging. Segmentation partitions process space requirements according to a logical division of the segments that make up the process. Generally, this simplifies protection.

Clearly, a clever scheme with advantages of both would be: segmentation with paging. In such a scheme each segment would have a descriptor with its pages identified. Such a scheme is shown in Figure 4.15. Note that we have to now use three sets of offsets. First, a segment offset helps to identify the set of pages. Next, within the corresponding page table (for the segment), we need to identify the exact page table. This is done by using the page table part of the virtual address. Once the exact page has been identified, the offset is used to obtain main memory address reference. The final address resolution is exactly as we saw in Section 4.9 where we first discussed paging.

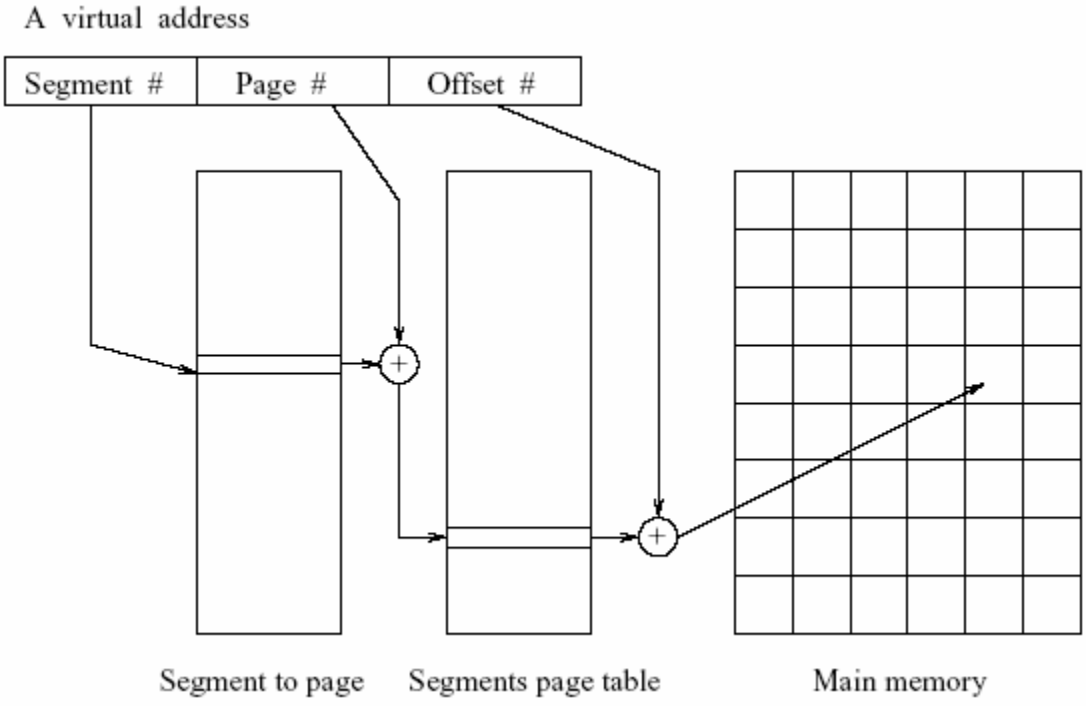

Figure 4.15: Segmentation with paging.

In practice, there are segments for the code(s), data, and stack. Each segment carries the *rwe* information as well. Usually the stack and data have read write permissions but no

execute permissions. Code rarely has write permission but would have a read and execute permission