Higher Certificate in Science in Computing – BN002

Operating Systems (Client)

Lecture 6: Memory Management 2

Dr. Kevin Farrell

Institute of Technology Blanchardstown
Institiúid Teicneolaíochta Baile Bhlainséir

Table of Contents

List of Figures

License

Feedback

Constructive comments and suggestions on this document are welcome. Please direct them to:
kevin.farrell@itb.ie

Acknowledgements

Thanks in particular go to the Dr. Anthony Keane, whose Lectures this material was partly based on.

Modifications and updates

Version	Date	Description of Change
1.0	1st June 2005	First published edition.
1.1	10th August 2005	Minor formatting changes
1.2	12th October 2005	Major formatting changes

1. Overview

1.1. Lecture Summary

In this Lecture, we continue our discussion of memory management. In the last Lecture, all of the methods discussed involved loading each process, in its entirety, into primary memory in order for it to be executed. This has considerable drawbacks. The newer methods discussed in that Lecture—simple paging and simple segmentation—can be adopted so that only a portion of each process is required to be loaded into primary memory. This leads us to the concept of virtual memory, whereby secondary memory (usually hard disk storage) is used to store the remainder of the process' pages. However, since secondary storage devices are electro-mechanical in nature, they are several orders of magnitude slower than purely electronic primary memory. We explain, using the working set model, how, despite these speed differences, virtual memory still allows for a system to operate at acceptable speeds on account of the principle of locality of reference. A number of other aspects of virtual memory-based systems are also discussed, such as thrashing, page-fault limiting algorithms, page-replacement algorithms and paged/segmented systems.

1.2. Learning Outcomes

After successfully completing this Lecture you should be able to:

- Distinguish between real and virtual memory.
- Calculate the virtual address space of a particular system based on your knowledge of processors.
- Explain the mechanics of virtual memory based on paging
- Describe the working set model, and how it allows for a virtual memory-based system to operate at acceptable speeds.
- Compare and contrast different page-replacement algorithms.

- Describe the paged/segmented system.

1.3. Reading

Material for this Lecture was gathered from a number of difference sources. The following texts are *essential* reading:

- “Operating Systems incorporating Windows and UNIX”, Colin Ritchie.
- “Operating System Concepts”, Silberschatz, Galvin & Gagne, John Wiley & Sons, (6th Edition, 2003).
- “Operating Systems”, William Stallings, Prentice Hall, (4th Edition, 2000)

1.4. Suggested Time Management

This Lecture should take between 2 and 4 hours of your study time:

- Lecture Content: 1 hours
- Further reading: 2 hours

1.5. Typographical Conventions

Throughout this Operating Systems Module, I have tried to use uniform typographical conventions; the aim being to improve readability, and lend understanding:

Key/Technical Terms:	**<u>Bold/Underline</u>**	for eg: **<u>Multiprogramming</u>**
Emphasis:	*Italics*	
Command names:	**`Courier/Bold`**	for eg: **`ls -l`**
Filenames/Paths:	`Courier`	for eg: `/home/kevin/cv.doc`

2. Introduction

The last Lecture dealt with **physical memory** (also called **primary memory** or **real memory**[1]); the actual addresses in RAM of the computer. Recall the following main points from that Lecture:

- A process, divided into separate pages or segments, can be loaded into separate page frames in main memory.
- Memory address references can be translated dynamically; i.e. at run time.
- It was suggested that only a portion of a process needs to be in memory in order for it to be executed, **but we did not discuss the implementation of this!**

The implications of this later point are:

- More processes can be sustained simultaneously implying greater CPU activity.
- Process can be larger than available **physical memory** (i.e. larger than amount of RAM).
- Increase in system complexity.

This lecture will deal with the means of running a process, *without* having to load all of the process into physical memory. That is, we will examine how the **virtual memory manager** operates to allow **secondary memory** (i.e. a portion of the hard-disk drive) to be used as an extension of physical memory. Figure 2.1 shows a schematic where several processes are active in a system, even though the total physical memory space is considerably less than the total space occupied by all of the processes. This is achieved by loading only a proportion of each process's pages into physical memory.

1 We will use the terms physical memory, real memory and primary memory interchangeably throughout this Lecture.

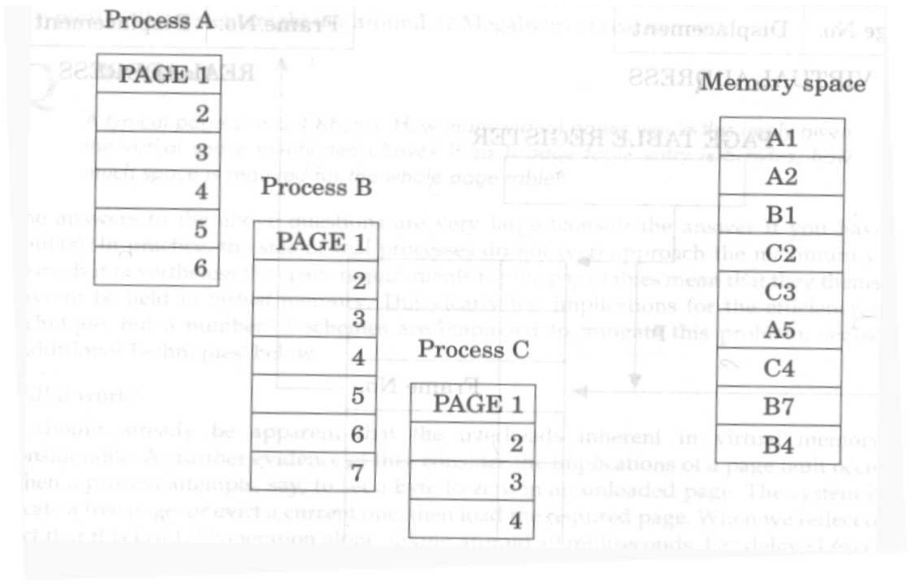


Figure 2.1: Virtual Memory allows several processes to be active in the system, even with a limited amount of physical memory (indicated by the words "Memory space").

3. Virtual and Real Address Space

The maximum amount of **virtual address space** is defined by the computer's processor; for example, a Pentium-IV has a 32-bit address bus, giving 4 GB of addressable memory (i.e. 2^{32} bytes = 4 GB). **Real address space** is limited to the amount of primary (physical) memory (i.e. RAM). It is unlikely (at least, for desktop PCs) that the amount of RAM would be as large as 4 GB. Usually, it is a lot less. The additional secondary memory, which can be used for virtual memory, is equal to the size of the virtual address space less the amount of RAM. The size of physical memory and the amount of active processes affect the performance of the system; in general, the more physical memory a system possesses, the greater the performance.

4. Mechanics of Virtual Memory Based on Paging

A new process must load at least *one* page into physical memory. If a virtual address is referenced that is not in the page, then a **page fault** is generated which interrupts the process and requires the system to load up the next page with the referenced address. This is called **demand paging**. When the required page is loaded, execution recommences. By a series of page faults occurring, a subset of pages of a process will gradually accumulate in real memory, this subset is called the **resident set** of the process. Loading each page requires a mapping between virtual memory and physical memory. This is shown schematically in Figure 4.1.

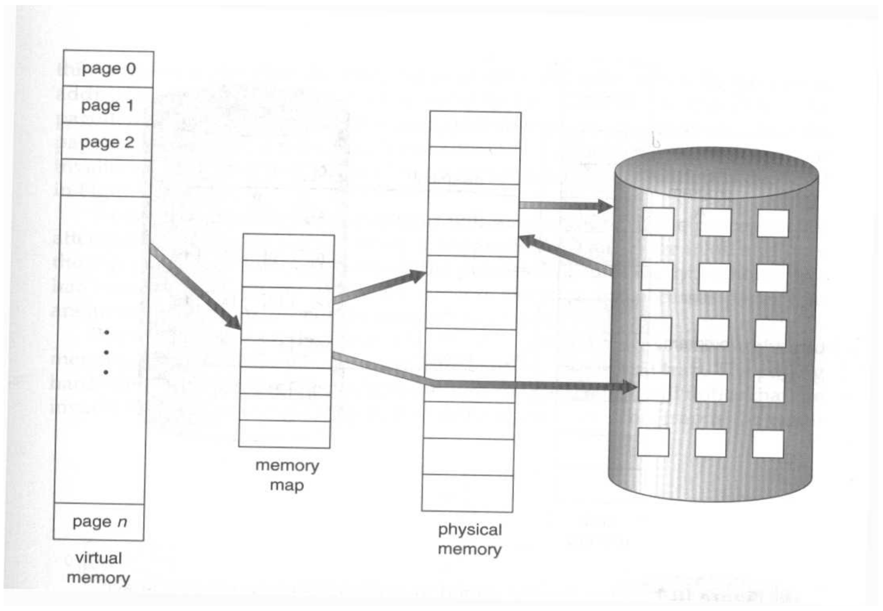


Figure 4.1: This figure shows that the logical representation of virtual memory is the same as real memory; i.e. It is represented as a single column with many rows. A memory map table is used to map between virtual and physical memory. Virtual memory is normally a swap file or swap partition on a hard disk drive (cylinder), and is larger than physical memory.

4.1. Page Replacement

In modern operating systems, there are generally many processes competing for the limited physical memory space. If a page fault occurs, the operating system often has to remove a currently loaded page in order to accommodate the new page. This procedure is called **page replacement**. There are a number of different algorithms, which can be used to determine which page should be removed from the real memory space. These are discussed later. Figure 4.2 below shows the transfer of pages from physical (real) memory to virtual memory on disk (indicated by the cylinder).

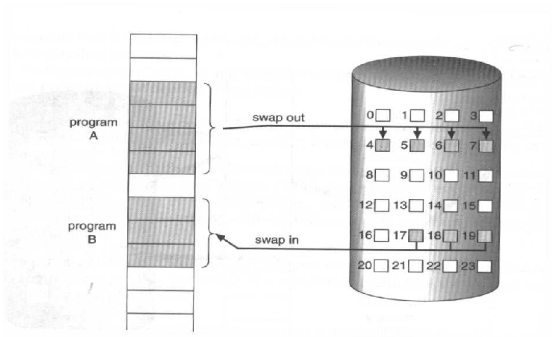


Figure 4.2: This figure shows the transfer of a pages from main (real) memory to virtual memory on disk (indicated by the cylinder).

4.2. Address Translation

As with simple paging discussed in the last lecture, a process loaded into virtual memory will not necessarily be loaded into the exact same memory addresses in real memory. This is particularly the case when virtual memory is larger than physical memory, which it usually is. Therefore, virtual memory addresses need to be **translated** (**relocated** or **converted**) to real addresses. This is implemented in a similar fashion to simple paging—that is, by using a **page table**—but includes an extra table called the **page table register**.

Since the page table of a process is of variable length, depending on the size of the process, we cannot expect to hold it in CPU registers. Instead, it must be in main memory to be accessed. When a particular process is running, a register, called the **page table register** holds the starting address of the page table for that process. This will contain the address in physical memory of the **page table** for the currently active process.

Since virtual memory can be very large, a very large number of processes can be active in the system (loaded into virtual memory). Since each process requires a page table, this would mean that there would be a very large number of page tables. However, unlike simple paging, storing all of these page tables in *real* memory is not feasible since we would run out of real memory just to store them, and would have no memory left for the actual pages of the processes. Instead, the page tables of all the processes are stored in virtual memory. This means that pages tables are subject to paging just as other pages are! When a process is running, at least part of its page table must be in main memory, including the page table entry of the currently executing page. For more details describing various schemes for managing a system with large numbers of pages, see William Stallings book.

The page table contains:

- real memory page translation
- control bits (Present, P (1,0); Modified, M (1,0))

The P bit indicates whether corresponding virtual page is in real memory. If the page is accessed by the process and P=0, then a page fault occurs.
The M bit indicates if a page has been modified (dirty) i.e. contents changed.

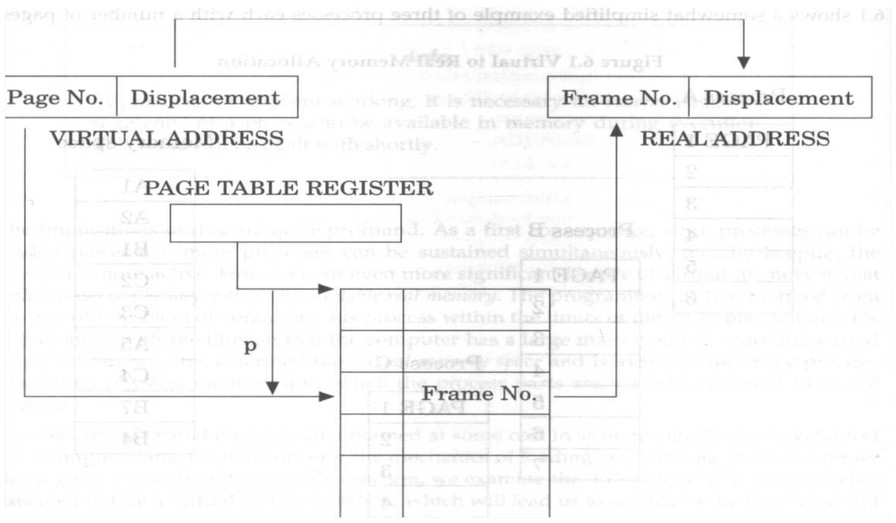


Figure 4.3: Virtual to real address translation is similar to simple paging. A page table register is used to locate the page table in virtual memory. Also, "modified" and "present" bits are used for housekeeping purposes by the memory manager.

4.3. Why Virtual Memory Works: The Working Set Model

Virtual memory introduces considerable overheads on the system when page faults are executed. However, between page faults, the system tends only to require address references in loaded pages, thus allowing the method to work at acceptable speeds. This phenomenon, called the **locality of reference**, ties in nicely with modular programming. Of course, the more pages of a process that are loaded into page frames in memory, the fewer the number of page faults that will occur (see Figure 4.4 below).

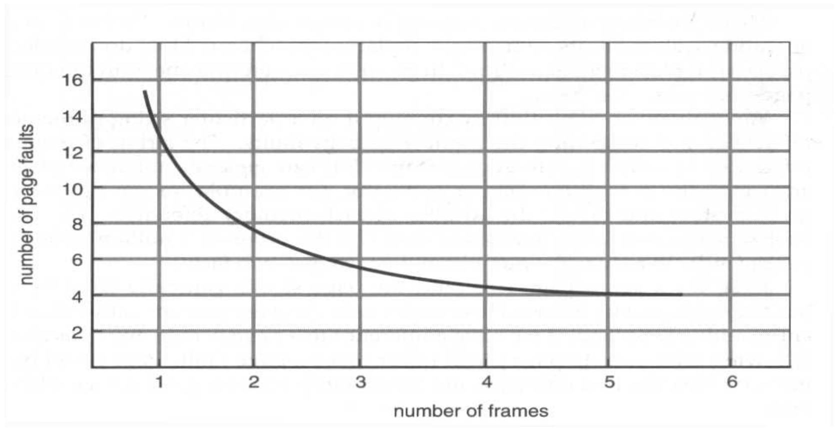


Figure 4.4: As the number of pages loaded into page frames in real memory increases, the number of page faults decreases.

The implication of locality of reference is that address references generated by a process tend to cluster within a narrow range, which are likely to be contained in a few pages. Consequently, the needs of a process over a period of time can be satisfied by a relatively small number of resident pages, which tends to reduce the number of page faults occurring. The locality principle leads to the concept of the **working set model**. The working set is the set of pages accessed by a process over a specific time interval called the **working set window**. For example, if the window is one millisecond, the working set is likely to be 1 to 2 pages, while if the window exceeds process time, the **working set**—the set of pages of the process accessed during the period of the working set window—will probably be the entire process. This would include pages loaded into real memory plus those residing in virtual memory.

Ideally the working set should be equal the resident set so that no page faults occur, and yet the process occupies minimal page frames. This is difficult to achieve in practice. Figure 4.5 shows the results of principle of locality in operation. Over time, the working set size increases and decreases. There are periods of stability, where pages are not loaded into memory, and the working set size is approximately constant. This is followed by a transient period, where many of the pages in real memory are unloaded, and new ones loaded, which results in another stable period. During the periods of stability, few page faults occur, allowing the system to execute at an acceptable speed.

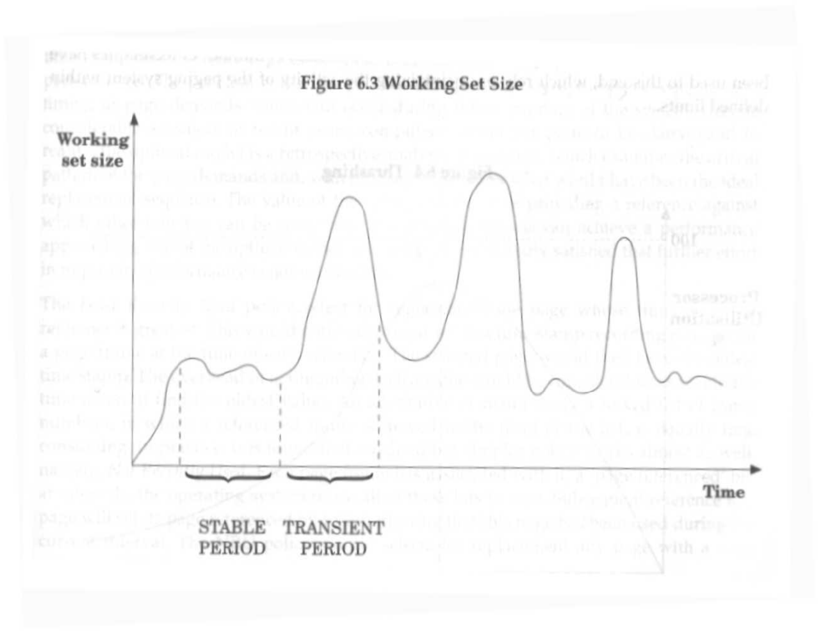


Figure 4.5: This figure shows the results of the principle of locality in operation.

4.4. Thrashing

Thrashing is when the processor spends most of its time swapping pages and doing little productive work. This occurs if too many processes are loaded, restricting the size of their resident sets. There is an optimum number of processes in a system, where the processor is utilised most efficiently. This is shown in Figure 4.6.

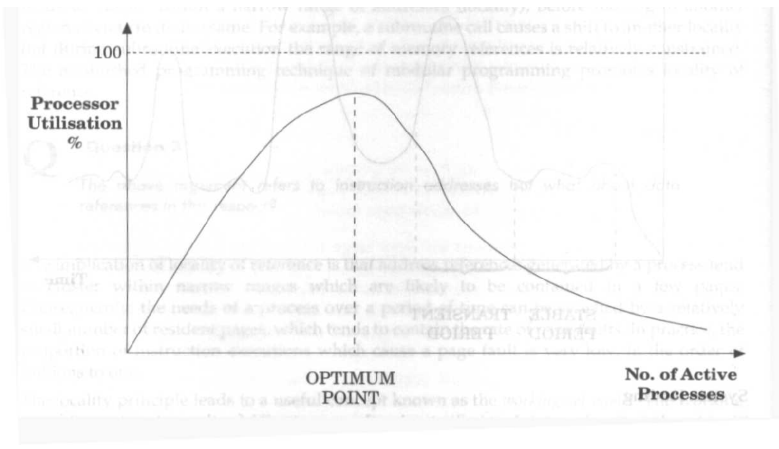
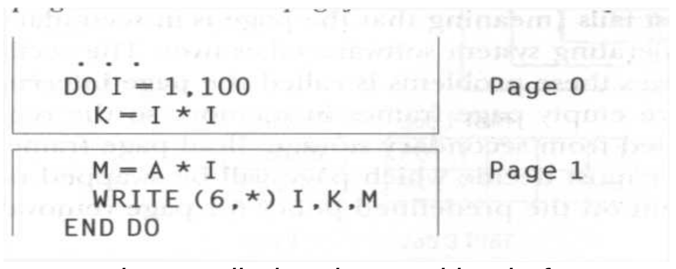


Figure 4.6: Too many processes can result in thrashing. This leads to poor processor utlilisation. There is an optimum number of processes in any system, where the processor is used most efficiently.

Below is an example of demand paging, which results in a page swap each time the loop is executed, and results in thrashing. If only a single page frame is available this program will have one page fault each time the loop is executed. If the working set consisted of the two pages, then no page faults occur.



```
  . . .
  DO I = 1,100                  Page 0
    K = I * I

    M = A * I                   Page 1
    WRITE (6,*) I,K,M
  END DO
```

Figure 4.7: Poorly compiled code, resulting in frequent demand paging. This results in a page swap each time the loop is executed, and leads to thrashing.

In practice, it is difficult to determine the working set size for a process without a counter-productive amount of measurement. The specific problem is how to prevent thrashing. Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate. We can establish upper and lower bounds on the desired page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate that process to another frame, (i.e. increase the size of the resident set); if the page-fault rate falls below the lower limit, we remove a frame from that process, (i.e. reduce the processes resident set size). Thus we can directly measure and control the page-fault rate to prevent thrashing. The algorithm, which accomplishes this is called **page-fault frequency** (PFF). Figure 4.8 shows the page-fault rate versus number of pages-frames used for a process. The horizontal lines indicate the upper and lower bounds of acceptability for page-fault rates.

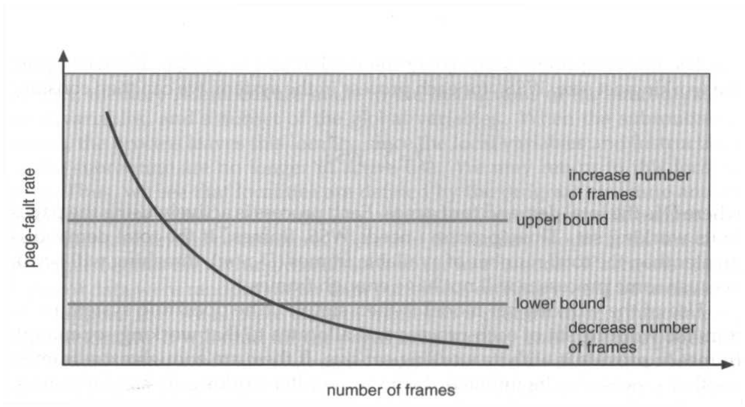


Figure 4.8: Page-Fault frequency imposes upper and lower bounds on the page-fault in order to control thrashing.

4.5. Page Replacement Policies

The operating system must control the number of processes to prevent the onset of thrashing and try to run the processor at maximum utilisation. Also to be fair to all processes, the resident sets should reflect the size of their process. A number of algorithms exist that try to achieve these ends; so-called **page-replacement** algorithms. We summarise four of the common ones as follows:

- **Least Recently Used (LRU):** Selects for replacement the page with the *greatest time since last reference* value. Requires overhead of time stamp on each page.
- **Not Recently Used (NRU):** At intervals, the OS resets all *page reference* bits to zero. If the page is subsequently referenced, the bit changes to 1, indicating that this page has been used in the current interval. The NRU policy simply selects for replacement any page with a page referenced bit equal to zero.
- **First In, First Out (FIFO):** The FIFO method selects for removal the page which has been resident in memory for the longest time since such a page is likely to be of no longer use. Simple algorithm to implement. Uses simple queuing method. Drawback is the loss of a heavily used page results in many page faults.
- **Clock:** This is a variation of FIFO. This requires the addition of another "used" bit in the queue entry, and the queue is now viewed as a circular list, with a pointer which moves round the circle, effectively indicating the "head" of the queue. When a page is first loaded its used bit is set to zero. When the page frame is subsequently referenced its used bit is set to 1. When a replacement is required, the pointer moves forward (from its previous position) round the list until a page frame with a zero used bit is found. Entries encountered with used bits of 1 have them set to zero. If all the entries are set to 1 in the first instance, the pointer will return to its starting point, which was initially zeroed. This is similar to the FIFO method except that a page frame which was recently used (used bit = 1) will be passed over in the first instance.

4.6. Hardware Solutions to Improve Performance

There are a number of hardware-based techniques, which can improve the performance of virtual paged systems. These are described below.

4.6.1. Translation Look-aside Buffer (TLB)

The page tables are held in memory and a memory reference by a process has to incur additional memory access to obtain the required page table entry. To reduce the effect of this overhead on the system performance, most systems use a form of hardware memory cache for page table entries called **Translation Look-aside Buffer** (TLB). The buffer holds a number of recently used page table entries, and is searched first by the translation process before it looks in the page table stored in primary memory. See Figure 4.9. Finding the page table entry in the TLB speeds up the translation procedure; that is, the procedure of determining to which page-frame of physical memory, a particular virtual page corresponds. Of course, it does not speed up accessing the memory address, once that translation procedure has completed. Recall that the memory address consists of the sum of the page frame number and displacement. If the page table entry is found in the TLB, we say that a **TLB hit** occurs; otherwise, we say that a **TLB miss** occurs. Figure 4.9 shows the address translation process incorporating TLB.

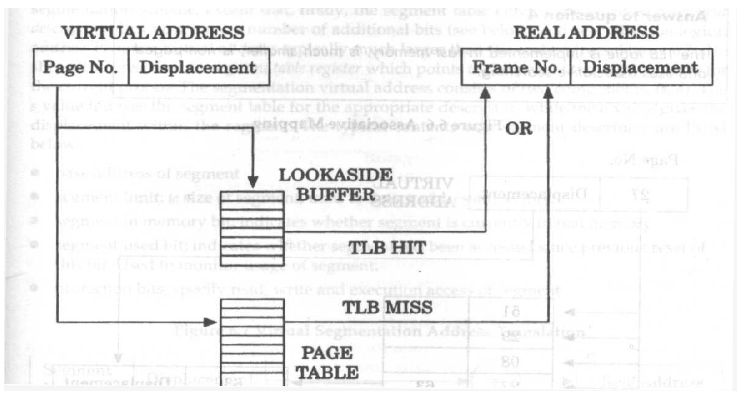


Figure 4.9: Translation Lookaside Buffer (TLB).

The main drawback of TLB is that the buffer has no index, and therefore needs to be searched serially.

4.6.2. Associative Mapping

The method of **associative mapping** greatly increases the speed of searching for an item within an array of stored items. This hardware system compares a search value with every entry simultaneously within the **associative store** (**associate look-aside buffer** or **associative registers**).This is shown schematically in Figure 4.10.

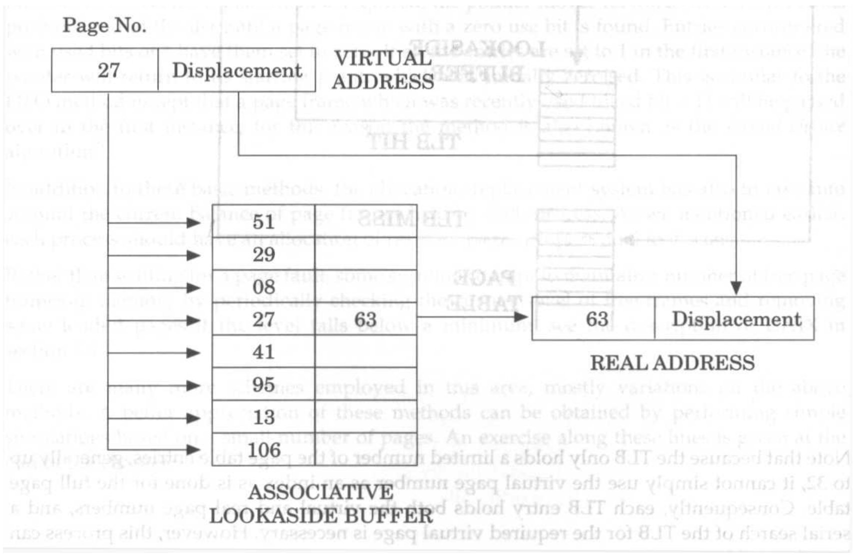


Figure 4.10: Associative Mapping.

Effective Access Time

We can quantify the improvement that the use of associative registers and TLB provide, by calculating the **Effective Access Time** ***(***EAT***)***. This is best explained by using examples.

Example 1: *Without* the use of associative registers or TLB, how long does a paged memory reference of a word take, If a memory reference takes **200 ns**?

Answer:

1. Because the page table itself resides in memory, it takes 200 ns to access that page table to determine (from reading the table) the location of the word in memory (the translation procedure).
2. It takes a further 200 ns to access the word in memory.

The total time for the reference of the word in memory is the sum of these two values = 400 ns.

Example 2: If we add associative registers, and **75%** of all page-table references are found in the associative registers, what is the effective memory access time? (Assume that finding a page-table entry in the associative registers takes **zero** time, if the entry is there).

Answer:

1. For 75% of the time, the location of the word in memory can be read by referencing the page-table in the associative register. We are told that accessing the register itself takes zero time, so reading the contents of the page table takes zero time. However, subsequently accessing the word in memory takes 200 ns. This gives a contribution to the **Effective Access Time** (EAT) of:
   - 75% x (0 + 200) ns = 0.75 x 200 ns = 150 ns
2. For the rest of the time (25% of the time), the page-table is not in the associative register, and thus requires a memory reference to access the page-table itself. Once the memory location of the word is determined from the table, accessing the word, takes a further 200 ns. This gives a contribution to the Effective Access Time of:
   - 25% x (200 + 200) ns = 0.25 x 400 ns = 100 ns
3. Therefore, the total Effective access time is:
   - ***EAT*** = 150 ns + 100 ns = 250 ns

We can see from these two examples that the time to access the word in memory is considerably reduced by using associative registers.

5. Virtual Segmented Systems

In a **Virtual Segmented System** (VSS) scheme, the segments of a process are loaded independently, and hence may be stored in any available physical memory positions or not be in physical memory at all. The advantages of VSS are:

- uses dynamic memory
- sharing of code/data between processes
- logical structure is reflected in the physical structure which reinforces the locality principle.

5.1. Segmented Address Translation Process

The segmented address translation procedure is similar to that for simple segmentation described in the previous Lecture. The procedure works as follows:

- **segment table register** points[2] to **segment table** which consists of entries called **segment descriptors**.
- logical address is virtual address (i.e. larger than real memory)
- segmentation virtual address has two components, (s,d) where:
  - ➢ s indexes the segment table and
  - ➢ d is the displacement within segment.
- Typical contents of each segment descriptor are:
  - ➢ base address of segment
  - ➢ segment limit (size): used to detect address errors
  - ➢ segment in memory bit: different values depending on whether in real memory or in virtual memory.
  - ➢ segment used bit: indicates segment used since previous reset.
  - ➢ protection bits: access of segment is read, write or execute.

As can be seen, the methodology also has some similarities to the translation procedure for the virtual paging system described earlier in this Lecture.

2 Note: in the C/C++ programming language, a “pointer” to a variable is the address of that variable. Since operating systems are written predominately in these languages, the programming language terminology has naturally wound its way into OS terminology.

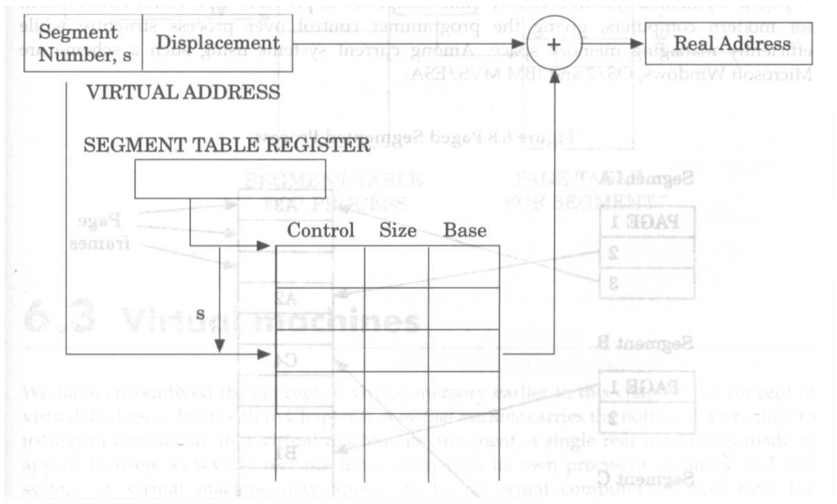


Figure 5.1: Virtual Segmented Address Translation.

6. Paged/Segmented System

One of the significant problems associated with segmented systems is the placement of variable sized segments in physical memory. Over time, as segments are unloaded and others loaded, memory becomes fragmented. One solution to this problem is the **paged/segmented system**.

A combined paged/segmented system appears to the programmer to be simply a segmented system; however the segments defined are subdivided at the physical level into a whole number of fixed length pages. This means that a segment smaller than a page will use the whole page, and typically one page within a segment will be partially unfilled. This small amount of “wasted” space in partially unfilled pages is internal fragmentation. However, the placement of each segment in physical memory now becomes much simpler. By loading pages of each segment into equal sized page-frames of physical memory there is no external fragmentation, and consequently, there is no requirement to manage external fragments – see Figure 6.1 below.

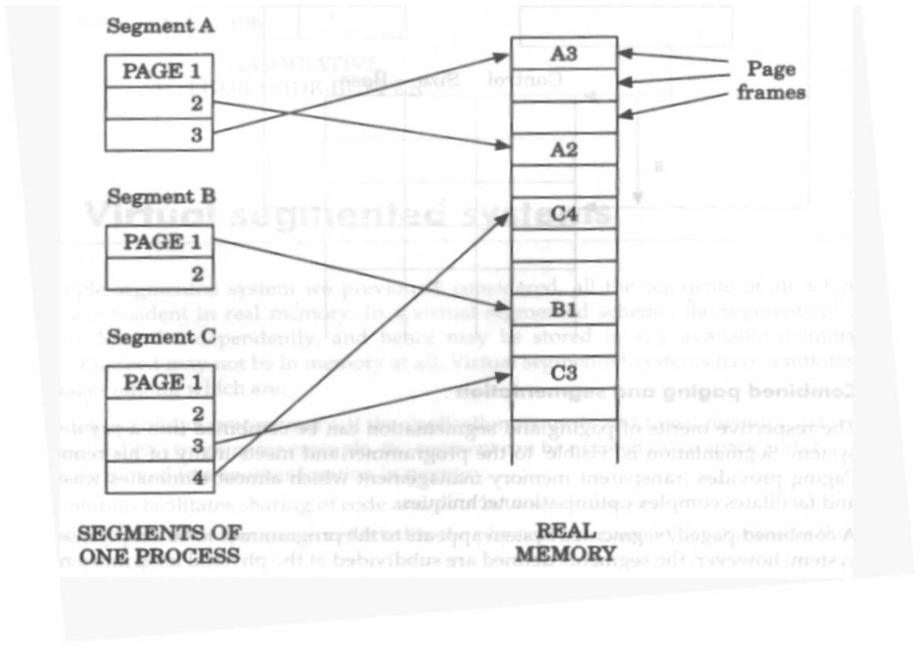


Figure 6.1: Logical memory arrangement in a Paged-Segmented System. Segments, which consist of whole numbers of pages, are loaded into page-frames in memory on a page-by-page basis.

6.1. Address Translation

Address translation in a paged/segmented system contains the translation procedures of the individual paging and segmentation systems. A memory reference is specified by a *three* element value such as (s,p,d):

- s = segment number
- p = page within segment
- d = displacement within page

where,

- Segment number, s, is used as index to segment table.
- Segment descriptor contains pointer to page table for that segment
- Page number, p, is used as index page table.
- Page table converts virtual page number to physical page number.

- The entity, d, is displacement within this physical page frame.

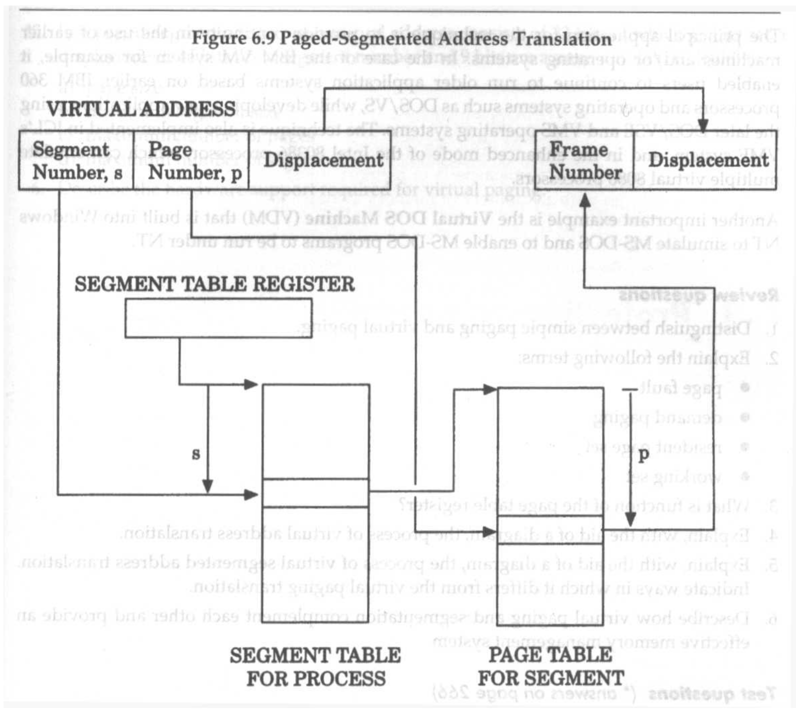


Figure 6.2: Address translation in a page-segmented system.

A paged/segmented system is clearly quite complex, but provided a powerful environment for modern computers giving the programmer control over process structure while efficiently managing memory space. Among current systems using such a scheme are Microsoft Windows, OS/2 and IBM MVS/ESA.

7. Protection and Sharing

It is essential that each process running in a system does not interfere with the code/data of any other process, either by accident or by deliberate intrusion. However, it is also desirable in some circumstances that process code/data is shared by two or more processes.

In the fixed and *v*ariable partition systems of memory allocation, the simplest mechanism for detection of page faults is the use of a limit register. This would contain the upper limit of the address space of the currently active process. If the generated address exceeds the limit, an address violation interrupt is triggered.

In paging systems, the displacement value can not exceed its limit of the page size, however the logical address could exceed the actual process size (real or virtual) which would produce an invalid page number. This can be trapped by use of a hardware register that holds the maximum page number for the current process. While it is possible for two or more processes to share real memory pages, this is rarely attempted, since the content of pages is generally unknown. One of the merits of the paging system is that it is largely transparent to the programmer.

It is an advantage of segmentation systems that sharing and protection can be readily implemented. Since segments are logical entries, holding programmer defined procedures and data objects, precise control of access and sharing is possible. Sharing is convenient in systems where several users require the same software such as text editor and compiler for example. These can be loaded as a shareable segment and accessed by each online user. Many systems, such as windowing environments use shared libraries that contain numerous routines necessary for commonly required functions, such as *window management*.

8. Windows Memory Management

MS-Windows 3.x had three modes of operation:

- Real mode – 8086/8088 compatible
- Standard mode – extended memory

- Enhanced mode – virtual memory
- Application programmers interface (API) to Win3.x memory system is called Win16API.
- Windows uses DLLs (Dynamic Link Libraries) to conserve memory space. DLLs are executable program files, contain shareable code linked with application code at run time. Device drivers are implemented as DLLs.

Windows 95/98/NT use 32-bit memory addressing which gives 4GB of logical address space. In addition, it

- It uses Win32API.
- 4 GB of virtual memory is divided into 2GB for system, 2GB for processes.
- System memory is broken into three sections:
  - Non-paged – virtual addressing
  - Paged
  - Physical addressing – kernel code/data

WinNT implements virtual memory using demand paging with clustering. Clustering handles page faults by bringing in the required page and multiple pages around that page. When the process is first created, it is assigned a working set minimum and maximum. The working set minimum will guarantee a certain amount of pages in memory and if sufficient memory is available, a process may be assigned as many pages as its work set maximum. The VMM maintains a list of free page frames. Associated with this list is a threshold value that is used to indicate whether there is sufficient free memory available or not. If a page fault occurs for a process that is below its working set maximum, the VMM allocates a page from this list of free pages. If a process is at its working maximum and it incurs a page fault, it must select a page for replacement using a local FIFO page replacement policy. When the amount of free memory falls below the threshold, the VMM uses a tactic known as automatic working set trimming to restore the value above the threshold.

9. UNIX Memory Management

UNIX memory management can be summarised as follows:

- UNIX process has three segments:
  - Text – code
  - Data – initialised and uninitialised
  - Stack segments – dynamic data
- Several processes can share one segment.
- Early UNIX used swapping.
- Current UNIX use virtual paging with page size 4K.
- Uses minimum 25, and maximum 40 limits on free frames.
- Page fault rate is monitored
  - In normal circumstances, pages are removed using the clock algorithm.
  - If page fault rate is excessive, pages are removed using LRU algorithm.

10. Exercises

1. What are the steps involved in handling a page fault?
2. A typical page size is 4 KB. How many virtual pages would this imply for a virtual memory space of 4 GB? If each page table entry is 5 bytes, how much space is required just to hold the whole page table?
3. Normal page table addressing and TLB translation both access a table to obtain a frame number. Why is TLB translation so much faster?

4. Without the use of Associative Registers or TLB, how long does a paged memory reference of a word take, If a memory reference takes **150 ns**? If we add associative registers, and **70%** of all page-table references are found in the associative registers, what is the effective memory access time? Assume that finding a page-table entry in the associative registers takes **10ns**, if the entry is there.
5. In a paged/segmented system, a virtual address consists of 32-bits, of which 12-bits are a displacement, 11-bits are a segment number and 9-bits are a page number. Calculate (a) page size (b) maximum segment size and (c) maximum number of pages.