Operating Systems (Client)

Lecture 5 – Memory Management 1

Higher Cert. in Science in Computing (IT) – BN002

Bachelor of Science in Computing (IT) – BN013

Bachelor of Science (Hons) in Computing – BN104

Dr. Kevin Farrell

itb
Institute of Technology
Blanchardstown
Institiúid Teicneolaíochta
Baile Bhlainséir

Contents

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	17th October 2005	Minor formatting changes
1.3	17th October 2007	New figure showing division of memory address into Page number and Displacement in Simple Paging. The allocation of bits to (p, d) now corresponds with other figures in the document.

1. Overview

1.1. Lecture Summary

In this Lecture, we discuss a number of methods for allocating real memory to processes. After a brief introduction, we review some historical methods, which are no longer used in popular contemporary operating systems. However, they do provide insight into how to establish the newer methods. These newer methods, simple paging and simple segmentation are then discussed in the latter half of the Lecture. All of the methods discussed involve loading each process in its entirety into memory in order for it to be executed. In the next Lecture, we will address the possibilities of loading only a portion of each process into memory.

1.2. Learning Outcomes

After successfully completing this Lecture you should be able to:

- Describe a number of historical memory allocation methods, and explain their drawbacks.
- Describe the methods of simple paging and simple segmentation, and their relative merits.
- Consider changes to simple segmentation, which would make it more efficient.

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 3 and 5 hours of your study time:

- Lecture Content: 2 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: **Bold/Underline** for eg: **Multiprogramming**

Emphasis: *Italics*

Command names:	**`Courier/Bold`**	for eg: **`ls -l`**
Filenames/Paths:	`Courier`	for eg: `/home/kevin/cv.doc`

2. Introduction

The management of main memory is critical to the computer system. Historically, the performance of the system has been directly dependent on *two* things:

1. how much memory is available
2. how memory is optimised while jobs are being processed.

Main memory not only holds the program code and the data operated on by the code, but it also holds the operating system code and data, other system programs, the system BIOS and device drivers, and so on.

Historically, a number of different memory management techniques have been used, (and improved upon) in the operating system. The principal goals of the operating system's memory management are:

- to provide the memory space to enable several processes to be executed at the same time.
- to provide a satisfactory level of performance for the system users.
- to protect each programs resources.
- to share (if desired) memory space between processes.
- to make the addressing of memory space as transparent as possible for the programmer.

2.1. Some Terminology

Logical address is the address space as perceived by the programmer. All programs, once compiled into machine code, start at address 0. But, of course, one cannot load every program into memory starting at address 0.

Physical address is the address space as used by OS. A conversion mechanism allows the logical addresses of a program to be translated into physical addresses.

Process loading is the transfer of program code from secondary storage to main memory.

In some systems, a **swapping** of processes occurs where the process loading is matched by a process being transferred from main memory to secondary storage at the same time. Swapping systems do not usually employ the standard file management facilities because they would be too slow. Instead, an area of hard disk is dedicated to the swapping activity and special software manages the data transfer at high speed.

Example 1:

Q. How long will it take to transfer a 100KB process by swapping? Assume the *data transfer rate* is 1 MB per sec, and the *access time* is 10 milliseconds (ms).

A. Transfer time = 100K/1000K = 1/10 = 0.1 s or 100 ms

Access time = 10 ms therefore total transfer time is 100+10=110 ms.

Since the swap involves a two-way movement (equivalent size process must be removed from memory), this value is doubled to 220 ms.

Example 2:

Q. What process states would be preferred for swapping and why?

A. If a process is BLOCKED waiting on an I/O operation, it would be a suitable candidate for swapping since it is occupying but not utilising the memory space.

We will now look at a number of memory management techniques in order of increasing complexity. It will be seen that each method “solves” some shortcomings of the previous method and reflects the way in which the systems evolved.

3. Memory Allocation Methods Discussed Here

We divide our discussion up into *two* parts in this Lecture. First, we examine historical memory allocation methods. These methods are much simpler than modern (contemporary) methods, and facilitate understanding of memory allocation issues. Once we have discovered the problems associated with these historical allocation methods, we discuss some memory allocation methods, which are used in contemporary operating systems.

Below, we list the names (and their variants) of the different methods discussed.

3.1. Historical Methods

The historical methods, which we will discuss are as follows:

- Single-User Process System
- Fixed Partition Memory
- Variable Partition Memory
  - coalescing of gaps
  - storage placement policies
    - best fit
    - first fit
- Variable Partition Allocation with Compaction

3.2. Contemporary Methods

The contemporary methods, which we will then discuss are:

- Simple Paging
- Simple Segmentation

4. Single-User Process System

The first memory allocation systems worked by loading each program to be processed *entirely* into memory and allocated as much **contiguous space** in memory as it needed. If a program was too large to fit in memory, it couldn't be executed.

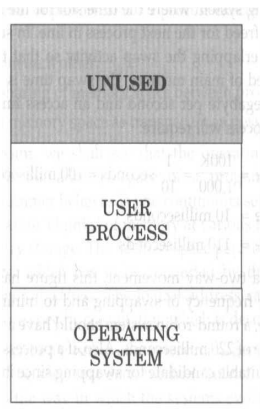

Figure 4.1: Single-User Process Memory Allocation System. A process has been allocated memory in this picture.

4.1. Algorithm to load a job in a single-user process system

1. Store first memory location of program into **base register**
2. Set **program counter** equal to address of first memory location
3. Load instructions of program
4. Increment program counter by number of bytes in instructions
5. Has the last instruction been reached?
   - if yes, then stop loading program.
   - if no, then continue with step 6.
6. Is program counter greater than memory size?
   - If yes, then stop loading
   - If no, then go to step 3

Note that the Single-User Process System of memory management doesn't support multiprogramming. It can handle only one job at a time.

5. Fixed Partition Memory Allocation

The first attempt to allow for multiprogramming was to create **fixed partitions** within the main memory – one partition for each job (see Figure 5.1). The partition sizes were decided when the system was powered-on and remained static until shut-down. The number of partitions in a practical system would depend on the amount of memory available and the sizes of processes to be run. Fixed partitions introduced the concept of ***memory space protection***. Once a partition was assigned to a process no other process could enter it.

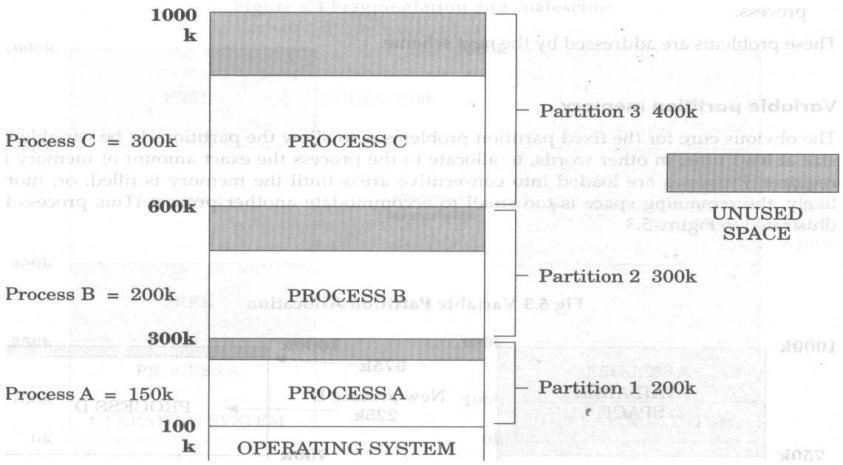

Figure 5.1: Fixed Partition Memory Allocation. Several user processes (A, B and C) are allocated to fixed partitions in memory in this picture. Note the **internal fragmentation**.

This partition scheme is more flexible than the single-user scheme because it allows several programs to be in memory at the same time. However, it still requires that the *entire program* be stored contiguously and in memory form the beginning to the end of execution.

In the next subsection, we give a sample pseudo-code for this algorithm, which could be used to find a suitable partition for a process needing loading.

5.1. Algorithm to load a job in a fixed partition

1. Determine job's requested memory size
2. If job_size > size_largest_partition
3. then reject the job
   - print appropriate message to operator
   - go to step 1 to handle next job in queue
4. else set counter to 1
5. Do while counter <= number_of_partitions_in_memory
6. If job_size > memory_partition_size(counter)
   - then counter = counter + 1
7. else if memory_partition_status(counter) = "free"
   - then load job into memory_partition(counter)
   - change memory_partition_status(counter) to "busy"
   - go to step 1
8. else counter = counter + 1
9. end do
10. No partition available at this time, put job in waiting queue
11. Go to step 1

5.2. Major drawbacks to the fixed partition memory allocation scheme:

There are two major drawbacks associated with the fixed-partition memory allocation scheme:

- **internal fragmentation**: the *partial* usage of fixed partitions by processes and leads to unused space in memory.
- fixed partitions can prevent processes from running if all the partitions are smaller than the process.

6. Variable Partition Memory Allocation

The weaknesses of the static partition approach can be solved if we allow the partitions to be *variable in size* at load time, i.e. to allocate to the process the exact amount of memory it requires. Processes are loaded into consecutive areas until the memory is filled, or the remaining free space is too small to hold another process.

When a process terminates, the space it occupies is freed and becomes available for loading a new process. Free spaces appear as “gaps” between active memory areas. A flaw in this approach is that a new process may be too big to fit any of the gaps. The distribution of free memory in this fashion is called **external fragmentation**. “*External*” refers to space outside any process allocation. As more processes finished, the gaps adjacent to a terminating process will coalesce making one bigger gap.

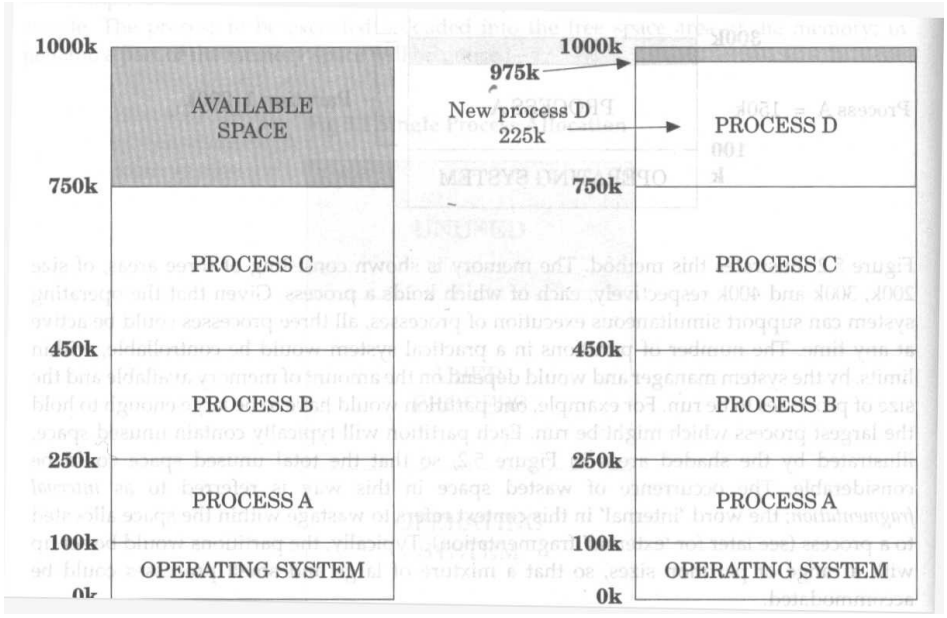

Figure 6.1: Variable Partition Memory Allocation. Several processes (A, B, C and D) are allocated partitions in memory in this picture.

Before going on to the next allocation scheme, we will now examine how the OS keeps track of the *free sections of memory*. There are a number of storage placement policies, principally **first-fit** and **best-fit*:***

- **First-fit*:*** an incoming process is placed in the first available gap that can accommodate it.
- **Best-fit*:*** an incoming process is placed in a gap in which it fits best.

Sample pseudo-code for each of these algorithms is given on the following pages.

6.1. Algorithm for First-Fit Policy

1. Set counter to 1
2. Do while counter <= number of blocks in memory
3. If job_size > memory_size(counter)
    - then counter = counter + 1
4. else
    - load job into memory_size(counter)
    - adjust free/busy memory lists
    - go to step 4
    - End do
5. Put job in waiting queue
6. Go fetch next job

6.2. Algorithm for Best-Fit Policy

1. Initialise memory_block(0) = 99999
2. Compute initial_memory_waste = memory_block(0) – job_size
3. Initialise subscript = 0
4. Set counter to 1
5. Do while counter <= number of blocks in memory
    - If job_size > memory_size(counter) then
        - counter = counter + 1
    - Else
        - memory_waste = memory_size(counter) – job_size
        - If memory_waste then < initial_memory_waste
            - initial_memory_waste = memory_waste
            - subscript = counter
            - counter = counter + 1
6. End do
7. If subscript = 0 then
    - put job in waiting queue
8. Else
    - load job into memory_size(subscript)
    - adjust free/busy memory lists
9. Go fetch next job

One problem with the Best-Fit algorithm is that the *entire table* must be searched before the allocation can be made because the memory blocks are physically stored in sequence according to their location in memory. This is expensive; i.e. there is a large overhead associated with this table-search.

6.3. Overheads

Any memory management scheme will incur some operating overhead; in the case of the variable partition scheme, the system must keep track of the posi-

tion and size of each gap, taking account of the effect of coalescing. A common method is to use a linked list, which contains pointers to the start of each gap and the gap size. Each allocation method would access such a list in its own way. The First-fit scheme would simply select the first item on the list of sufficient size, while the Best-fit would need to scan the full list of sufficient size before deciding.

7. Variable Partition Allocation with Compaction

The external fragmentation problem can be tackled by physically moving resident processes about the memory in order to close up the gaps and hence bring the free space into a simple large block. This process is known as **compaction** – see Figure 7.1.

The drawbacks are the need to suspend all processes while the reshuffle takes place, and the overhead of updating the process context information. Such an activity would not be feasible in a time critical system and would be a major overhead in any system. There is also the problem of *when, and how often* to do the compaction of memory.

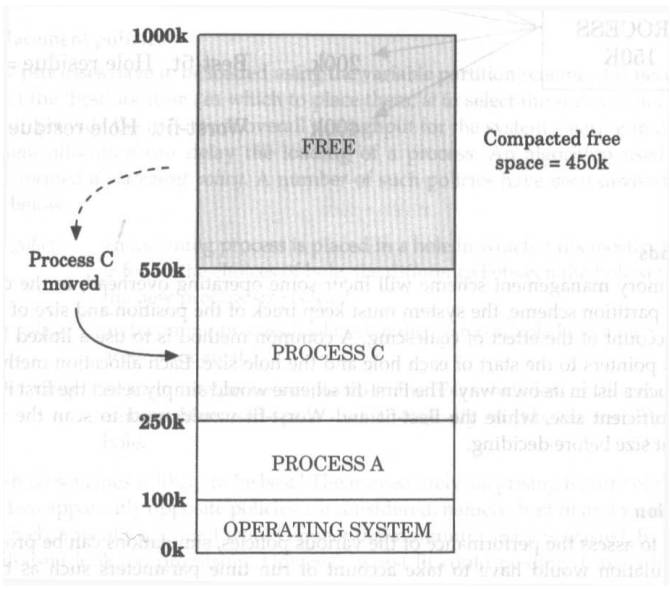

Figure 7.1: Variable Partition Memory Allocation with Compaction. This picture shows compaction of memory, whereby Process C is moved to a location adjacent to Process A; thereby freeing up a large block of memory.

8. Review of Historical Methods

Let us review the historical methods:

- Seeking a Memory Management system to enhance throughput of the system
- Problem with all methods discussed so far is the unwanted creation of gaps in memory
- Even the best solution has unacceptable overheads

9. Simple Paging

Previously, we looked at memory allocation methods that require the storage of the entire program in main memory in contiguous locations. We have discussed how this leads to problems such as fragmentation and/or overheads associated with relocation of those fragments.

A more sophisticated memory allocation method called **paging** removes the restriction of storing programs contiguously and this eliminates the requirement that the entire program resides in memory during its execution. In a paged system, each process is divided into a number of fixed size chunks called **pages**, typically 4K long. The memory space is also viewed as a set of **page frames** of the same size. The loading process now involves transferring each process page to some memory page frame.

Figure 9.1 shows a schematic with a number of processes waiting to be loaded into memory. If process B terminates, as shown in Figure 9.2, it leaves a gap between A and C. Two new processes D and E are loaded. D needs 3 pages and E needs 4 pages. These are easily accommodated using the new system of paging and there is no loss of space. This approach leads to better space utilisation and increases throughput. Processes F, G and H lead to further separation of gaps, and this raises the question of how does the system track these changes?

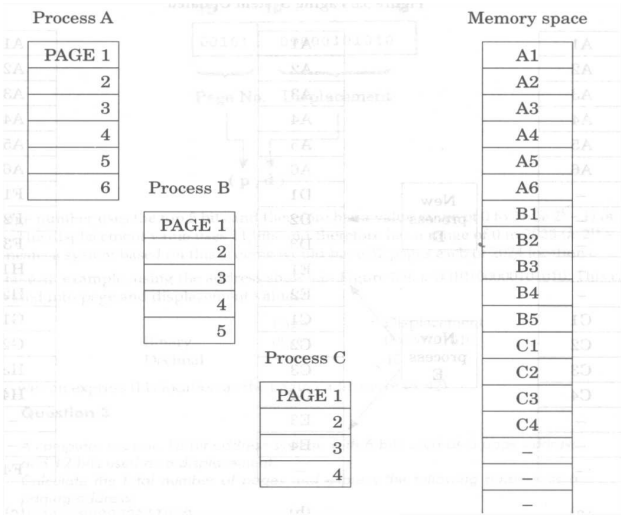

*Figure 9.1: Simple Paging Memory Allocation. Note the allocation of **pages** of each process (A, B, C) from disk to **page frames** in memory.*

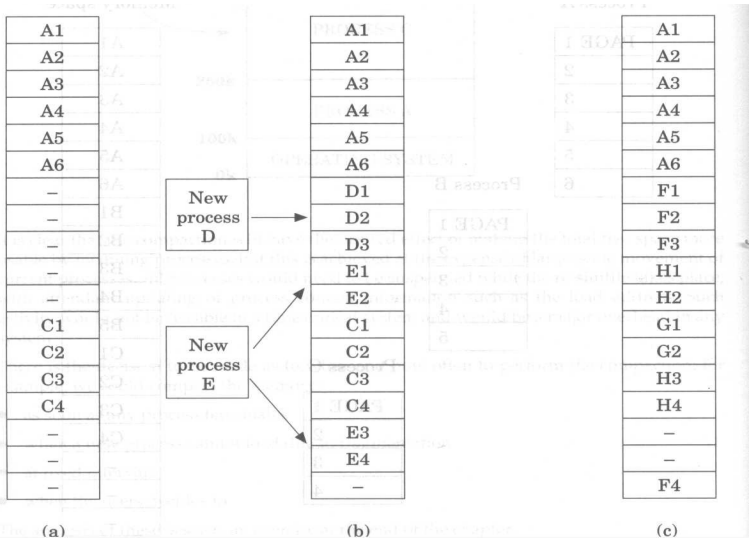

*Figure 9.2: Simple Paging Memory Allocation. The **pages** of each process (A, B, C, etc.) are allocated from disk to **page frames** in memory. Note that pages of any one process need not be in a contiguous block of memory.*

9.1. Implementation of Simple Paging

In a paging system, the memory location has an address of the form (p,d) where

- p = number of the page containing the location,
- d = displacement (offset) of the location from the start of the page

These parameters are derived from the actual memory address by subdividing the latter into two portions, representing the respective page and displacement values.

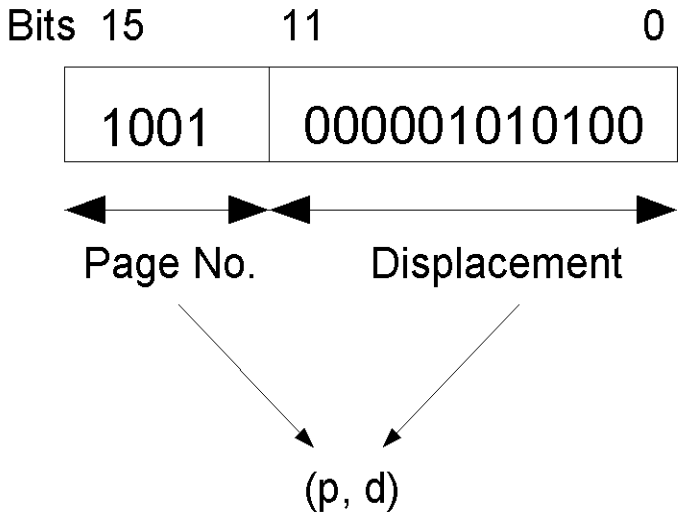

Figure 9.3: Simple paging is implemented by using a portion of the memory address as a Page Number and the other portion as a Displacement in that page.

The page number in our example use 4 bits giving a range of 0 → 15 (2^4-1) or 16 pages and the displacement uses 12 bits giving range 0 → 4095 (2^{12}-1) or 4096 locations. Therefore, a system based on this scheme has 16 pages each of 4096 bytes in size, since each location within a page is 1 byte in size. (Recall each address is 1 byte in size, and an address is obtained from combining one page and one displacement).

9.1.1. Conversion of a Logical Address to a Real Address

To keep track of the paging, the memory manager keeps a **page table**. This has one entry for each possible **process page number** and contains the corresponding **memory page frame number**.

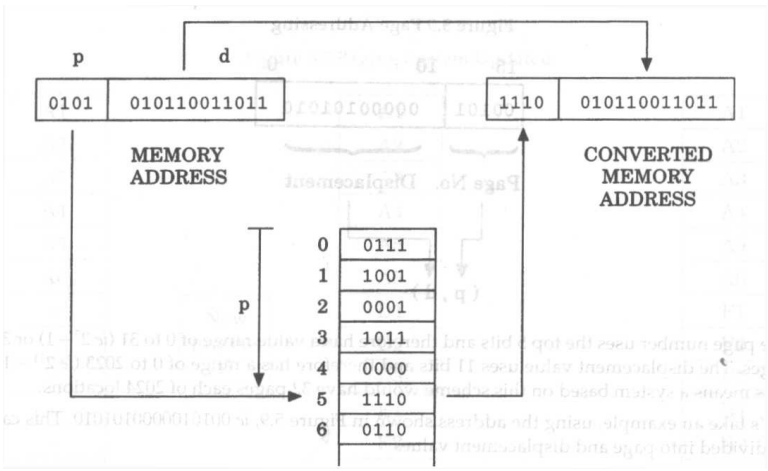

Figure 9.4: Conversion of a **logical address** *of a process into a* ***real address*** *in a* **page frame** *in memory is achieved by using a look-up table, called the Page Table. Note that all of the displacements in the page frame are the same as the displacements in the page.*

The relocation of a page is achieved using the page table. When a **process page** is positioned in some memory page frame, the **page number parameter** of the paging address changes but the displacement remains the same. Relocation reduces simply to converting a *process page number to a memory page frame number.* Note that the process page number is used to *index* the page table that in effect is an array of memory page frame numbers.

9.2. Example of Paged Memory Allocation

The example in Figure 9.5 below shows how the memory manager keeps track of a program that is four pages long. To simplify the arithmetic we will set the page size at 100 lines (or bytes). Job 1 is 350 lines (or bytes) long and is READY for execution. With Job 1 in memory, the maximum available space for other processes is 700 lines.

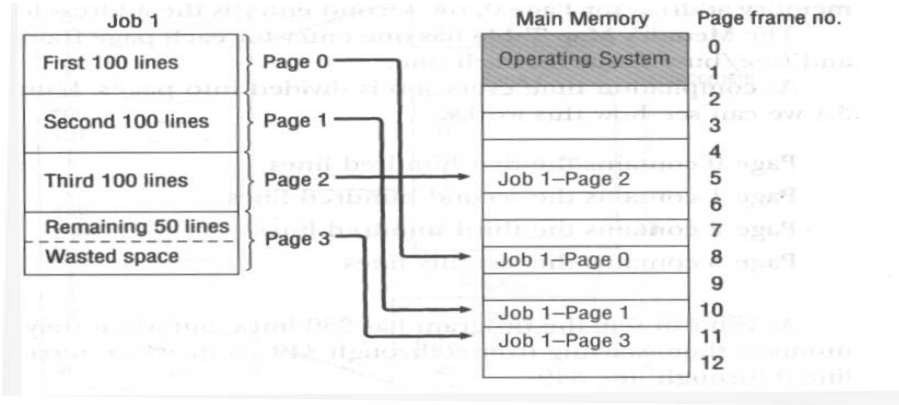

Figure 9.5: Programs too long to fit on a single page are split into equal-sized pages than can be stored in free page frames. In the picture above, each page frame can hold 100 lines. This job (process) is 350 lines long, and is divided among four page frames. This leaves internal fragmentation in the last page frame.

9.2.1. Keeping Track of Memory Allocation for Processes

The memory manager uses tables to keep track of pages. There are *three* tables:

- **Job Table** (JT)
- **Page Map Table** (PMT)

- **Memory Map Table** (MMT)

The **Job Table** (JT) contains *two* entries for each active job (process):

1. size
2. memory location of **Page Map Table** (PMT))

Each active job has its own PMT containing:

1. page number (automatic)
2. page frame memory address.

The MMT has one entry for each page frame listing the location and free/busy status for each job. At compilation time, every job is divided into pages. In the case of our example (see Figure 9.6 below):

- Page 0 contains the first 100 lines
- Page 1 contains the next 100 lines
- Page 2 contains the next 100 lines
- Page 3 contains the last 50 lines

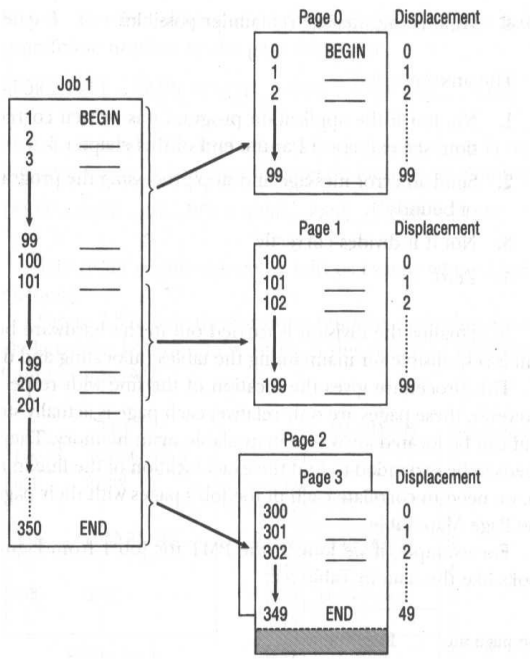

Figure 9.6: At compilation time, every job is divided into pages. The displacement (offset) of a line in a page (i.e. how far away a line is from the beginning of its page) is the number used to locate that line within its page frame, after the page has been loaded into a page frame in memory.

The system stores the program lines from 0 to 349. The displacement (offset) of a line (that is, how far away a line is from the beginning of its page) is the factor used to locate that line within its page frame. It's a relative factor. The first line of each page has a displacement zero, the second line has a displacement of one, and so onto the last line (or byte), which has a displacement of 99.

For example, lines 1, 100, 200 and 300 are the first lines for pages 0, 1, 2 and 3 respectively, so each has a displacement of zero. If the OS needs to access line 214, it would first go to page 2 and then to line 14, (i.e. the 15th line). So, once the operating system finds the correct page, it can access a line using the line's relative position within its page; i.e. its displacement.

The OS uses an algorithm to calculate the page and displacement. To find the address of a given program line, the line number is divided by the page size, keeping the remainder as an integer. The resulting quotient is the page number, and the remainder is the displacement with that page.

For example: Let the page size = 100 lines (or bytes). The page number and displacement of line 214 is obtained as follows:

214/100 = Quotient 2 with Remainder 14

=> We have Page 2 and Displacement 14 within that page.

10. Simple Segmentation

Segmentation is an alternative method of dividing a process by using *variable length* chunks called **segments**. Segmentation is similar in some ways to the variable partition method of allocation described earlier in the Lecture, except that a process can be loaded in several different portions, e.g. Segments. Because these can be independently positioned in the memory, it can provide more efficient utilisation of free space areas. Segments can be of any length, up to a maximum value determined by the design of the system. Segments correspond to *logical divisions* of the process and are defined *explicitly* by the programmer. Typically, the segments defined by the programmer would reflect the modular structure of the process, e.g. data in one segment, code in another segment. Figure 10.1 shows an example of three processes, each of which are separated into segments of various sizes, and subsequently loaded into memory.

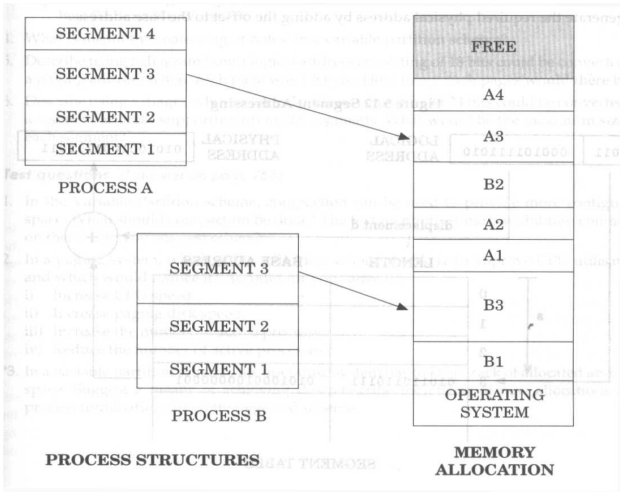

Figure 10.1: Separation of processes into segments, and their subsequent allocation into memory.

10.1. Implementation of Simple Segmentation

In a segmented system, the memory location has an address of the form (s,d) where

s = the **segment reference**: number of the segment containing the location

d = **displacement** (offset) of the location from the **base** (start) of the segment.

These parameters are derived from the memory address by subdividing the latter into two portions, representing the respective segment reference and displacement values.

The segment reference is used by the OS to index a **process segment table** whose entries specify the **base address** and **segment size**.

10.2. Conversion of a Logical Address to a Real Address

The mechanism for conversion of a **logical address** into a **real address** reference is slightly more complicated than for simple paging. It requires the following steps by the memory manager:

- Extract the segment number and the displacement from the logical address
- Use the segment number to index the **segment table**, to obtain the **segment base address** and the **segment length** (Remember: segments are of varying lengths). The ***segment base address*** is the (real) address in memory of the *beginning* (base) of the segment.
- Check that the displacement (also called the offset) is not greater than the segment length; if it is, an invalid address is signalled, since that displacement would then refer to some location beyond the end of the segment, which is impossible!
- Generate the required physical (real) address by adding the displacement to the base address.

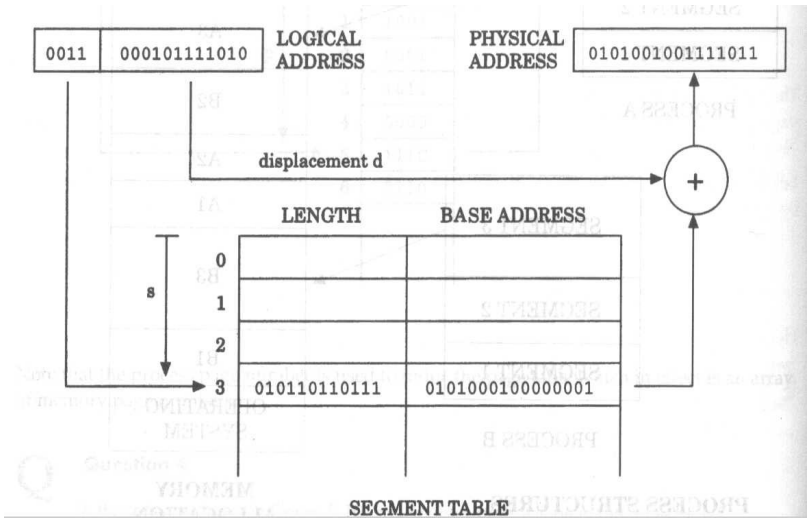

Figure 10.2: Conversion of a logical segment address of a process to a real memory address is achieved by using a look-up table, called the Segment Table. An additional check is required to verify that the displacement is within the bounds of the segment length.

11. Exercises

1. A variable partition memory system has at some point in time the following gaps sizes in the given order: 20K, 15K, 40K, 60K and 25K. A new process is loaded of size 25K. Which gap would be filled using first-fit and best-fit policies? Give reasons for your answer.

2. In Figure 9.4, how many entries will be in the page table? Show your calculations.

3. In relation to Figure 9.6, answer the following questions:

   Calculate the page number and displacements for the following lines in the process: 136, 300, 56 and 350.

   Could the OS access a page number outside the last page of a program?

   If it did, what should the OS do?

   Could the OS obtain a remainder of more than 99? What is the smallest remainder possible?

4. An Operating System uses a 32-bit address system. Each 32-bit address is subdivided by allocating 19 bits to the **page number** and the remaining 13 bits to the **displacement**. What is the maximum number of pages available in such a system?
5. Explain the allocation problems associated with simple segmentation. How might these problems be addressed?