**Applied Operating System**
**Study Guide**

| Module 3 |
| --- |
| MEMORY MANAGEMENT |

## SUBTOPIC 1: EARLY SYSTEM MEMORY MANAGEMENT TECHNIQUES

### What is Memory management?

**Memory management** - is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

- It keeps track of each and every memory location, regardless of either it is allocated to some process or it is free.
- It checks how much memory is to be allocated to processes.
- It decides which process will get memory at what time.

It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status

- **Address binding** is the process of mapping from one address space to another address space.
- Usually, a program resides on a disk as a binary executable file. The program must then be brought into main memory before the CPU can execute it.
- Depending on the memory management scheme, the process may be moved between disk and memory during its execution.
- The collection of processes on the disk that are waiting to be brought into memory for execution forms the **job queue** or **input queue.**

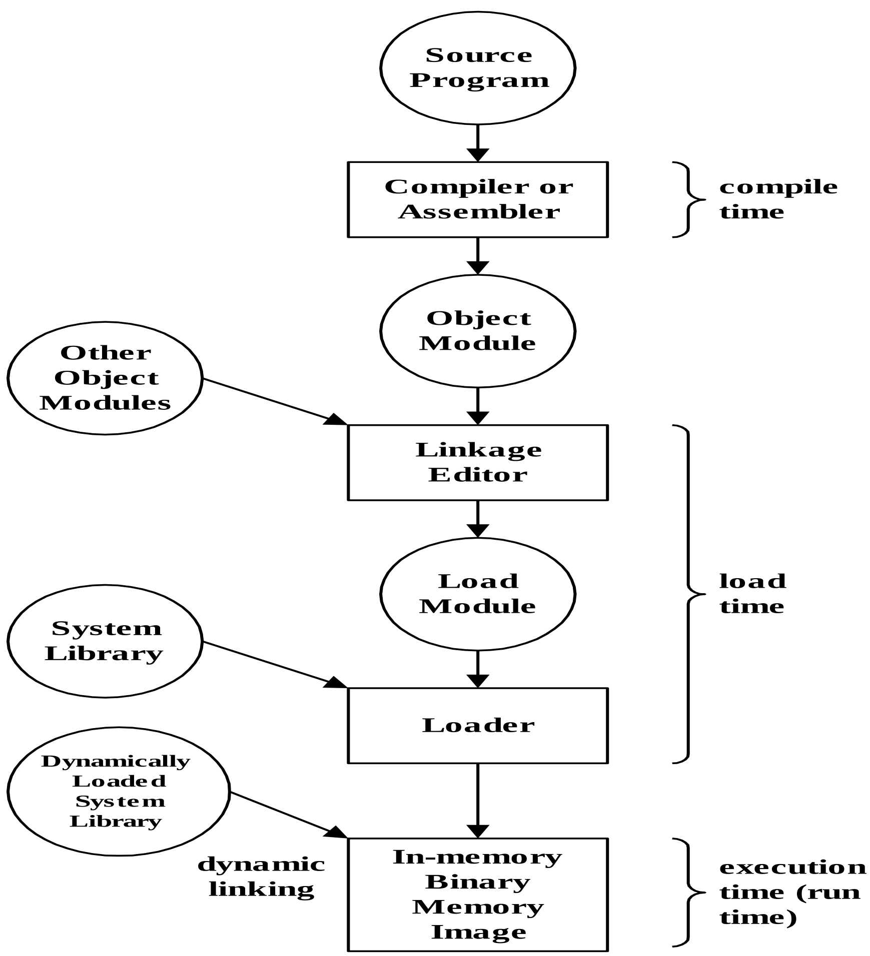
- **Compile Time Address Binding** – If you know that during compile time where process will reside in memory then absolute address is generated.
- **For example,** physical address is embedded to the executable of the program during compilation. Loading the executable as a process in memory is very fast. But if the generated address space is preoccupied by other process, then the program crashes and it becomes necessary to recompile the program to change the address space.

- **Load time** – If it is not known at the compile time where process will reside then relocatable address will be generated. Loader translates the relocatable address to absolute address. The base address of the process in main memory is added to all logical addresses by the loader to generate absolute address. In this, if the base address of the process changes then we need to reload the process again.

- **Execution time** - The instructions are in memory and are being processed by the CPU. Additional memory may be allocated and/or deallocated at this time. This is used if process can be moved from one memory to another during execution(dynamic linking- Linking that is done during load or run time). e.g. – Compaction.

- To obtain better memory-space utilization, **dynamic loading** is often used.
- With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format.
- Whenever a routine is called, the **relocatable linking loader** is called to load the desired routine into memory and to update the program's address tables to reflect this change.
- **Linking** is a method that helps OS to collect and merge various modules of code and data into a single executable file.
- The file can be loaded into memory and executed.
- OS can link system-level libraries into a program that combines the libraries at load time.
- In **Dynamic linking method**, libraries are linked at execution time, so program code size can remain small.
-

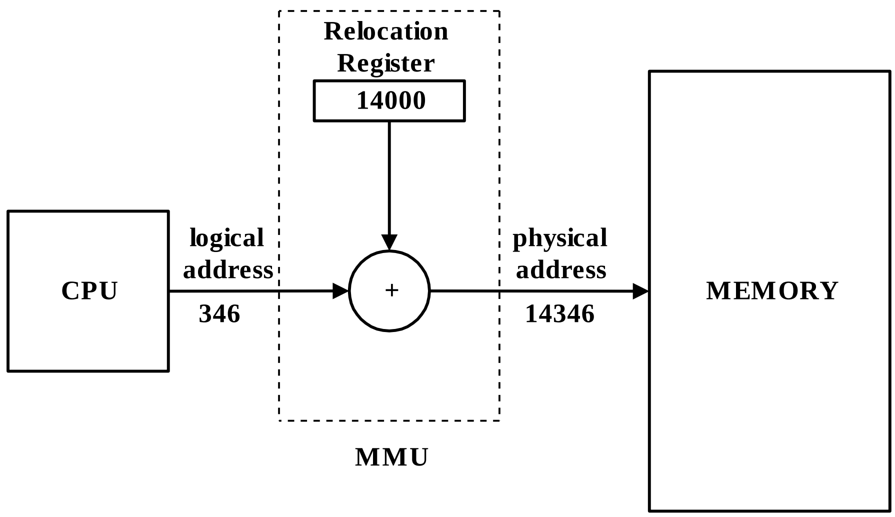
**MULTI-STEP PROCESSING OF A USER PROGRAM**



## LOGICAL AND PHYSICAL ADDRESS SPACE

- An address generated by the CPU is commonly referred to as a **logical address**.
- An address seen by the memory unit is commonly referred to as a **physical address**.
- The **compile-time** and **load-time** address binding schemes result in an environment where the **logical** and **physical addresses are *the same***.

- However, the **execution-time address-binding** scheme results in an environment where the **logical** and **physical addresses *differ***.
- The run-time mapping from logical to physical addresses is done by the **memory management unit (MMU)**, which is a hardware device.

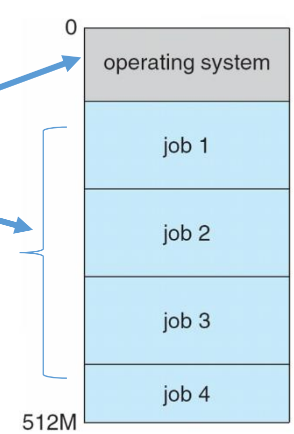



## EARLY SYSTEMS MEMORY ALLOCATION TECHNIQUES

**Memory allocation** is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes

Main memory usually has two partitions:

- Low Memory − Operating system resides in this memory.
- High Memory − User processes are held in high memory.

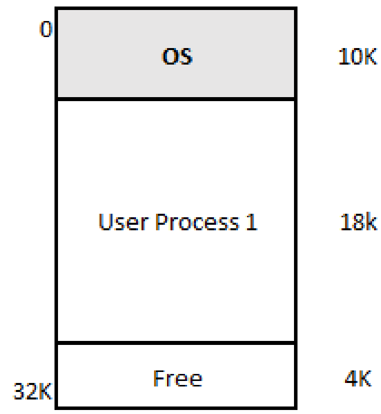


## MEMORY ALLOCATION TECHNIQUES

- **Single** Partition Allocation
- **Multiple** Partition Allocation
  - Fixed partitions (MFT)
  - Variable/Dynamic partitions (MVT)
  - Relocatable Variable/Dynamic partitions (MRVT)
- **Paging**
- **Segmentation**

## SINGLE PARTITION

- **Single** Partition Allocation set aside some memory for the OS and user program gets the rest.
- Use MMU to translate addresses by simple addition.
- Does not support multiprogramming.




## MULTIPLE FIXED PARTITIONS

**Fixed Partitions** is the oldest and simplest technique used to put more than one processes in the main memory.

In this partitioning, number of partitions (non-overlapping) in RAM are **fixed** but **size of each partition may or may not be same.**

## FIXED PARTITIONS

Since the size of a typical process is much smaller than the main memory, the operating system divides main memory into a number of partitions wherein **each partition may contain only one process**.

The **degree of multiprogramming** is bounded by the *number of partitions*.

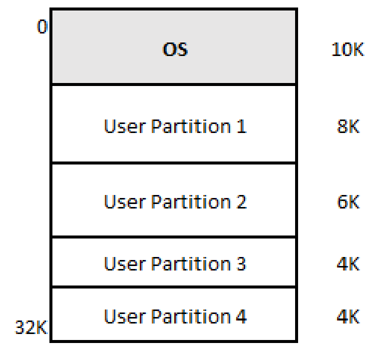
When a **partition is free**, the operating system **selects a process** from the **input queue** and **loads** it into the **free partition**.

When the process terminates, the partition becomes available for another process.

**Example:**

Assume a 32K main memory divided into the following partitions:

| | |
|---|---|
| **10K** | operating system |
| **8K** | user partition 1 |
| **6K** | user partition 2 |
| **4K** | user partition 3 |
| **4K** | user partition 4 |
| **32K** | |




- One flaw of the **FIRST-FIT** algorithm is that it forces other jobs (particularly those at the latter part of the queue to wait even though there are some free memory partitions).
- An alternative to this algorithm is the **BEST-FIT** algorithm. This algorithm allows small jobs to use a much larger memory partition if it is the only partition left. However, the algorithm still wastes some valuable memory space.

Other problems with Fixed Partitions:

1. What if a process requests for more memory?

**Possible Solutions:**

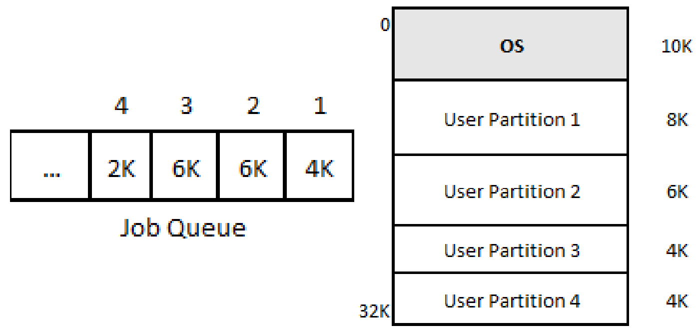
- kill the process
- return control to the user program with an “out of memory” message
- reswap the process to a bigger partition, if the system allows dynamic relocation

2. How does the system determine the sizes of the partitions?
3. Multiple Fixed Partition Technique (MFT) results in **internal** and **external** fragmentation which are both sources of memory waste.

The OS places jobs or process entering the memory in a job ***queue*** on a predetermined manner (such as **first-come first-served**).

There are two possible ways to assign or allocate jobs to the available memory partitions:

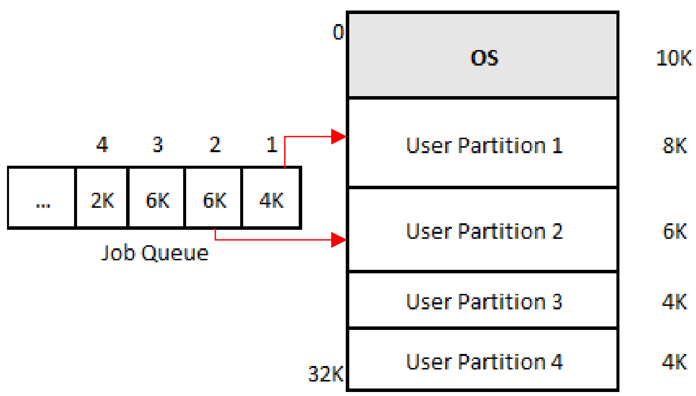
- **First-fit** allocation
- **Best-fit** allocation




**FIXED PARTITIONS – First Fit Allocation**

In **FIRST-FIT allocation**, first job claims the first available memory with space more than or equal to it's size.

The OS doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.




In **FIRST-FIT** memory allocation, below will be observed:

1. Assign **Job 1** (4K) to **User Partition 1** (8K) – since *Partition 1* is the **FIRST** encountered memory available that is large enough for *Job 1.*

2. Assign **Job 2** (6K) to **User Partition 2** (6K) - since *Partition 1* is already occupied by *Job 1*. Thus, *Partition 2* is the **FIRST** encountered memory available that is large enough for *Job 2*.

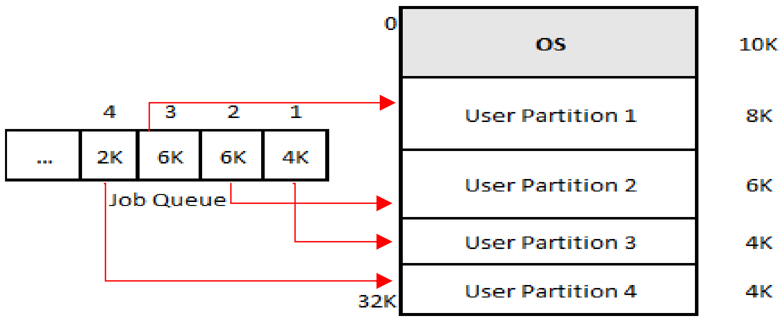
3. Job 3 (6K) **won't fit** to the rest of the available partitions (Partition 3 and 4). Thus, Job 3 should wait for its turn when there is a memory available that is large enough for Job 3.

4. Job 4 **cannot use** User Partition 3 since it will go ahead of Job 3 thus breaking the FCFS rule.  So it will also have to wait for its turn even though User Partition 3 is free.

**FIXED PARTITIONS – Best Fit Allocation**

In **BEST-FIT allocation**, keeps the free/busy list in order by size – smallest to largest.

In this method, the OS first searches the whole memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently.




In **BEST-FIT** memory allocation, below will be observed:

1. Assign **Job 1** (4K) to **User Partition 3** (4K) – since *Job 1* fits exactly to *Partition 3*.
2. Assign **Job 2** (6K) to **User Partition 2** (6K) - since *Job 2* fits exactly to *Partition 2*.
3. Assign **Job 3** (6K) to **User Partition 1** (8K) - since *Partition 1* is the closest-fitting free partition in the memory for *Job 3.*
4. Assign **Job 4** (2K) to **User Partition 4** (8K) - since *Partition 4* is the closest-fitting free partition in the memory for *Job 3.*

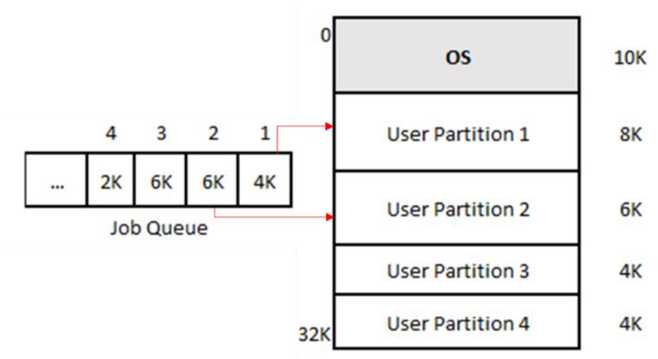
**FRAGMENTATION**

- As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as **Fragmentation**.
- Fragmentation is of **two types**:
  - Internal fragmentation
  - External fragmentation

**Internal fragmentation** occurs when a partition is too big for a process. The difference between the partition and the process is the amount of internal fragmentation.

**External fragmentation** occurs when a partition is available, but is too small for any waiting job.

**Example 1:**




Using **FIRST-FIT** algorithm, only ***Jobs 1*** and ***2*** can enter memory at ***Partitions 1*** and ***2***.

During this time:

**I.F.** = (8K - 4K) + (6K - 6K)

= 4K

**E.F.** = 4K + 4K

= 8K

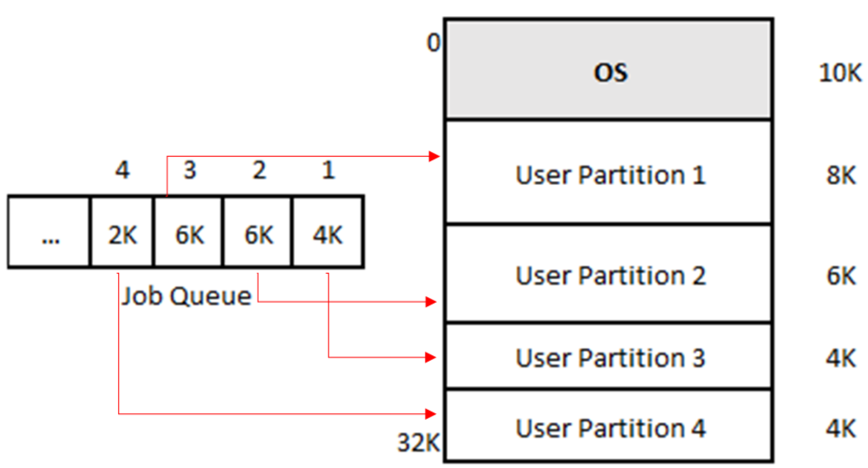
Therefore:

**Memory Utilization** = (total process size inside partitions / total memory available)

= (4K + 6K)/22K x 100

= 45.5%

**Example 2:**



Using **BEST-FIT** algorithm

During this time:

**I.F.** = (8K-6K)+(6K-6K)+(4K-4K)+(4K-2K)

= 4K

**E.F.** = 0

(since all partitions are containing processes)

Therefore:

**Memory Utilization** = (total process size inside partitions / total memory available)

= (6K + 6K + 4K + 2K)/22K x 100

= 81.81%

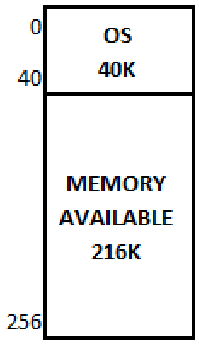
## VARIABLE PARTITIONS (MVT)

- In **Multiple Variable Partition Technique (MVT)**, the system allows the region sizes to vary dynamically. It is therefore possible to have a variable number of tasks in memory simultaneously.
- Initially, the OS views memory as one large block of available memory called a **hole**. When a job arrives and needs memory, the system searches for a hole large enough for this job. If one exists, the OS allocates only as much as is needed, keeping the rest available to satisfy future requests.
- Assume that the memory size is **256K** with the OS residing at the first **40K** memory locations.
- Assume further that the following jobs are in the job queue:
- The system again follows the **FCFS** algorithm in scheduling processes.

### Example:

Assume that all jobs arrived at the same time

| Job | Memory | Run Time |
|---|---|---|
| 1 | 60K | 10ms |
| 2 | 100K | 5ms |
| 3 | 30K | 20ms |
| 4 | 70K | 8ms |
| 5 | 50K | 15ms |



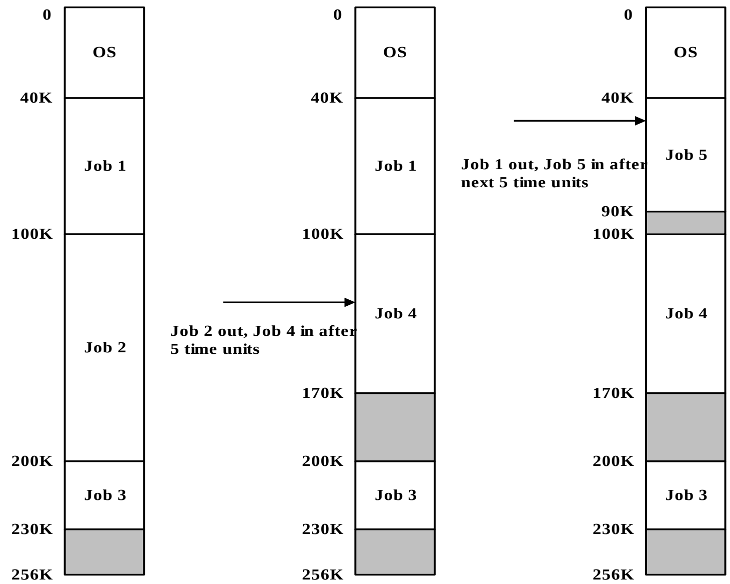




Table shows when a job has *started* and *ended*. It also shows the *waiting time (time started – arrival time)* and the *memory available* left after job allocation.

| Job | Memory | Run Time | TIME STARTED (ms) | TIME FINISHED (ms) | WAITING TIME (ms) | MEMORY AVAILABLE when job was allocated |
|---|---|---|---|---|---|---|
| 1 | 60K | 10ms | 0 | 10 | 0 | 156K |
| 2 | 100K | 5ms | 0 | 5 | 0 | 56K |
| 3 | 30K | 20ms | 0 | 20 | 0 | 26K |
| 4 | 70K | 8ms | 5 | 13 | 5 | 56K |
| 5 | 50K | 15ms | 10 | 25 | 10 | 66K |

Note that using **FIRST-FIT** algorithm, the last job finished at **25ms**.

This example illustrates several points about :

1. In general, there is at any time a set of holes, of various sizes, scattered throughout memory.
2. When a job arrives, the operating system searches this set for a hole large enough for the job (using the **first-fit**, **best-fit**, or **worst fit** algorithm).

**First Fit**

- Allocate the first hole that is large enough. This algorithm is generally faster and empty spaces tend to migrate toward higher memory. However, it tends to exhibit external fragmentation.

**Best Fit**

- Allocate the smallest hole that is large enough. This algorithm produces the smallest leftover hole. However, it may leave many holes that are too small to be useful.

**Worst Fit**

- Allocate the largest hole. This algorithm produces the largest leftover hole. However, it tends to scatter the unused portions over non-contiguous areas of memory.

- If the hole is too large for a job, the system splits it into two: the operating system gives one part to the arriving job and it returns the other the set of holes.
- When a job terminates, it releases its block of memory and the operating system returns it in the set of holes.
- If the new hole is adjacent to other holes, the system merges these adjacent holes to form one larger hole. This is also known as **coalescing**.
- **Internal fragmentation does not exist** in MVT but **external fragmentation is still a problem**. It is possible to have several holes with sizes that are too small for any pending job.
- The solution to this problem is **COMPACTION**. The goal is to shuffle the memory contents to **place all free memory together in one large block**.

**RELOCATABLE VARIABLE PARTITIONS (MRVT )**

**Example:**

- **Compaction** is possible only if relocation is *dynamic*, and is done at execution time.

**Given:**

| Job | Memory | Run Time |
|---|---|---|
| 1 | 60K | 10ms |
| 2 | 100K | 5ms |
| 3 | 30K | 20ms |
| 4 | 70K | 8ms |
| 5 | 50K | 15ms |

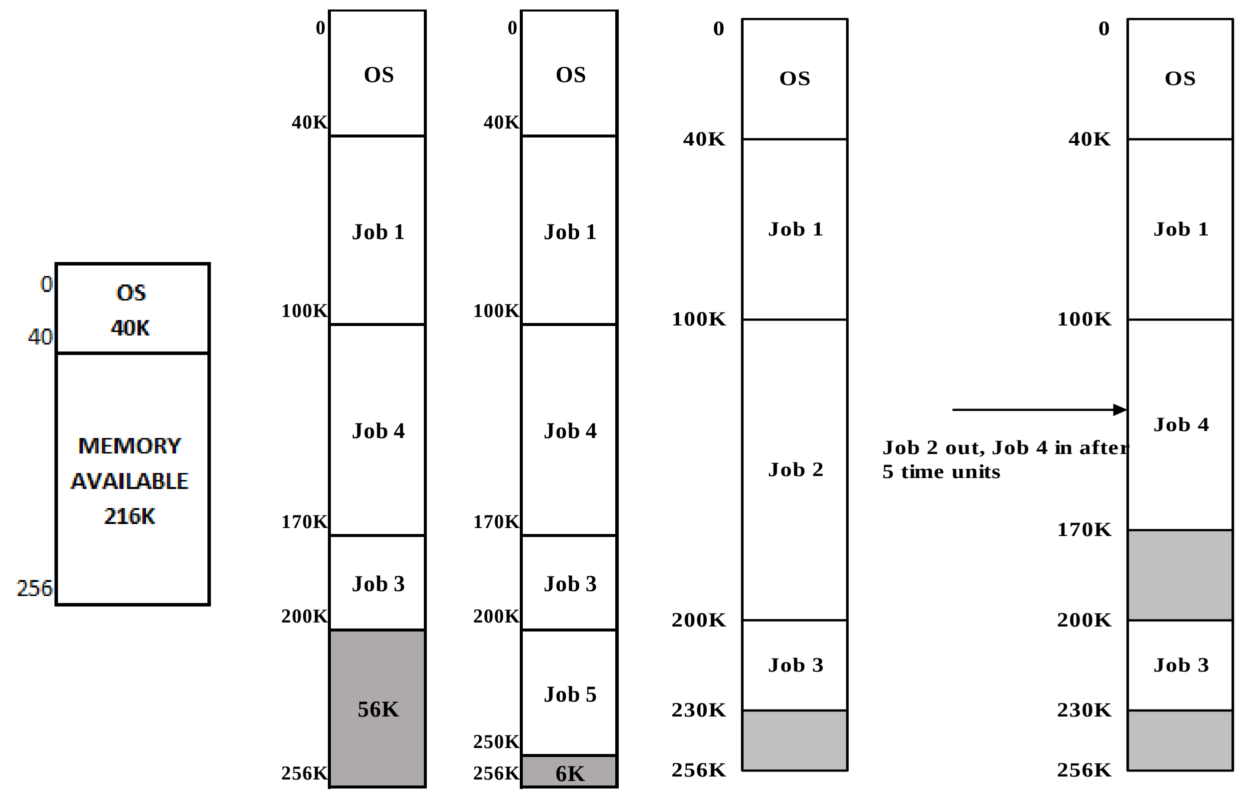



Table shows when a job has *started* and *ended*. It also shows the *waiting time (time started – arrival time)* and the *memory available* left after job allocation.

| Job | Memory | Run Time | TIME STARTED (ms) | TIME FINISHED (ms) | WAITING TIME (ms) | MEMORY AVAILABLE when job was allocated |
|---|---|---|---|---|---|---|
| 1 | 60K | 10ms | 0 | 10 | 0 | 156K |
| 2 | 100K | 5ms | 0 | 5 | 0 | 56K |
| 3 | 30K | 20ms | 0 | 20 | 0 | 26K |
| 4 | 70K | 8ms | 5 | 13 | 5 | 56K |
| 5 | 50K | 15ms | 5 | 20 | 5 | 6K |

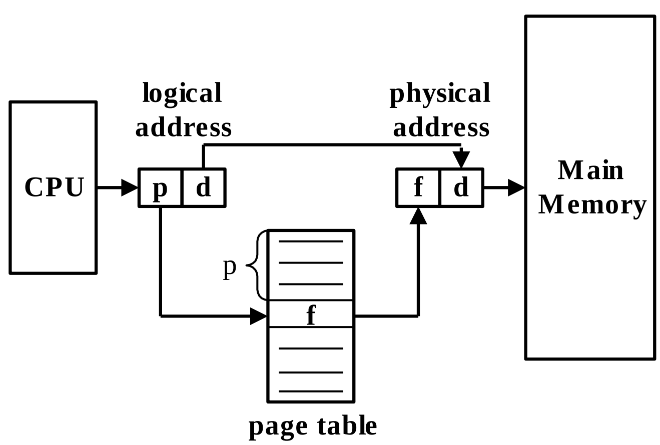
Note that using **BEST-FIT** algorithm, the last job finished at **20ms** which is **earlier** than **FIRST-FIT** algorithm thus making it able to use memory **efficiently**.

## SUBTOPIC 2: PAGING AND SEGMENTATION MEMORY ALLOCATION TECHNIQUES

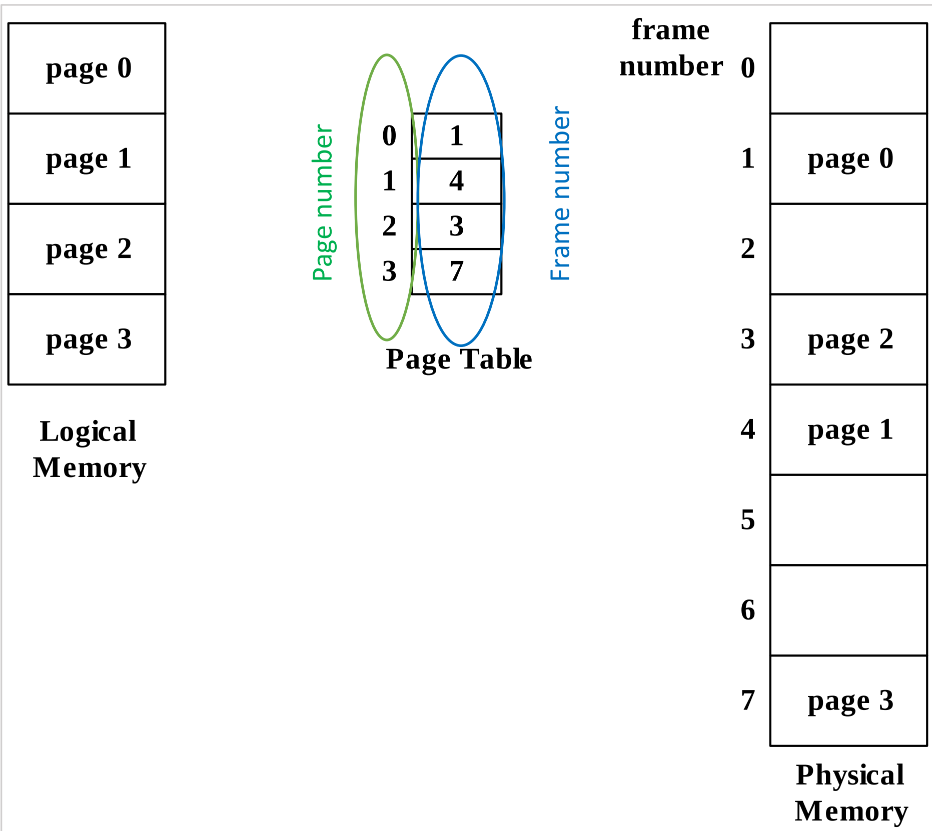
### PAGING

- **MVT** still suffers from **external fragmentation** when available memory is not contiguous, but fragmented into many scattered blocks.
- Aside from **compaction (MRVT)**, **paging** can **minimize external fragmentation**. Paging permits a program's memory to be non-contiguous, thus allowing the operating system to allocate a program physical memory whenever possible.
- **Memory paging** is a memory management technique for controlling how a computer or **virtual machine's (VM's) memory** resources are shared.
- This non-physical memory, which is called **virtual memory**, is actually **a section of a hard disk** that's set up to **emulate** the computer's RAM.
- The portion of the hard disk that acts as physical memory is called a **page file**.
- In paging, the OS divides **main memory** into **fixed-sized blocks** called ***frames***.
- The system also breaks a **process** into blocks called ***pages.***
- The **size of a memory frame** is **EQUAL** to the **size of a process page**.
- The pages of a process may reside in different frames in main memory.
- The OS translates this **logical address** into a **physical address** in main memory where the word actually resides. This translation process is possible through the use of a **page table**.
- Every address generated by the CPU is a *logical address*.
- A logical address has two parts:

1. The *page number* (*p*) indicates what page the word resides.
2. The *page offset* (*d*) selects the word within the page.



- The **page number** is used as an index into the page table.
- The page table contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.




- The page size (like the frame size) is defined by the hardware. The size of a page is typically **a power of 2** varying between 512 bytes and 16 MB per page, depending on the computer architecture.
- If the size of a logical address space is $2^m$, and a page size is $2^n$ addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the $n$ lower-order bits designate the page offset. Thus, the logical address is as follows:

| page number | page offset |
|---|---|
| $p$ | $d$ |
| $m - n$ | $n$ |

**Example**

Main Memory Size = 32 bytes

Process Size = 16 bytes

Page or Frame Size = 4 bytes

No. of Process Pages = 4 pages

No. of MM Frames = 8 frames

- **Physical Memory Address for <u>*32 bytes*</u>** *Memory Size*

  *(No. of frames =* ***8 frames;*** *frame size =* ***4 bytes*** *)*

Physical Address Format:

$A_4\,A_3\,A_2\quad A_1\;\;A_0$

***frame no.   page offset***

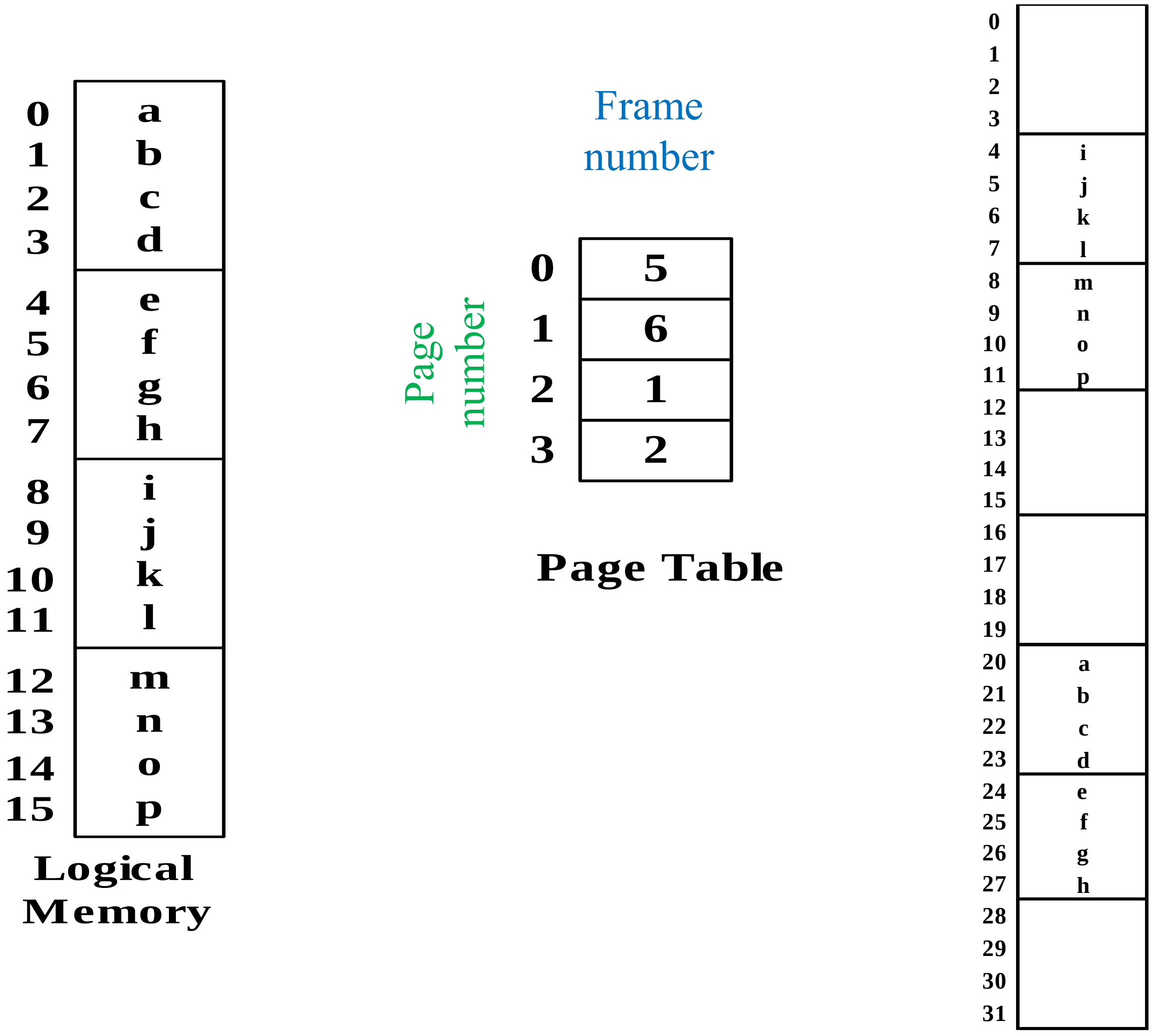
**Logical Memory Address for <u>*16 bytes*</u>** *Process Size*

*(No. of pages =* ***4 pages****; Page Size =* ***4 bytes****)*

Logical Address Format:

$A_3\;\;A_2\;\;A_1\;\;A_0$

***page no.  page offset***




- Logical address 0 **(a)** is page 0, offset 0.
- Indexing into the page table, it is seen that **page 0** is in **frame 5**. Thus, logical address 0 **(a)** maps to physical address **20**.
- **Physical address** = *frame no x page size + offset* = (5 x 4 + 0).
- Logical address 3 **(d)** is page 0, offset 3 maps to physical address **23** (5 x 4 + 3).
- Logical address 4 **(e)** is page 1, offset 0; according to the page table, **page 1** is mapped to **frame 6**.
- Logical address 4 **(e)** maps to physical address **24** (6 x 4 + 0).
- Logical address 13 **(n)** maps to physical address **9** (2 x 4 + 1).

| | |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

**Page Table**

| | |
|---|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |
| 7 | h |
| 8 | i |
| 9 | j |
| 10 | k |
| 11 | l |
| 12 | m |
| 13 | n |
| 14 | o |
| 15 | p |

**Logical Memory**

**Given:**

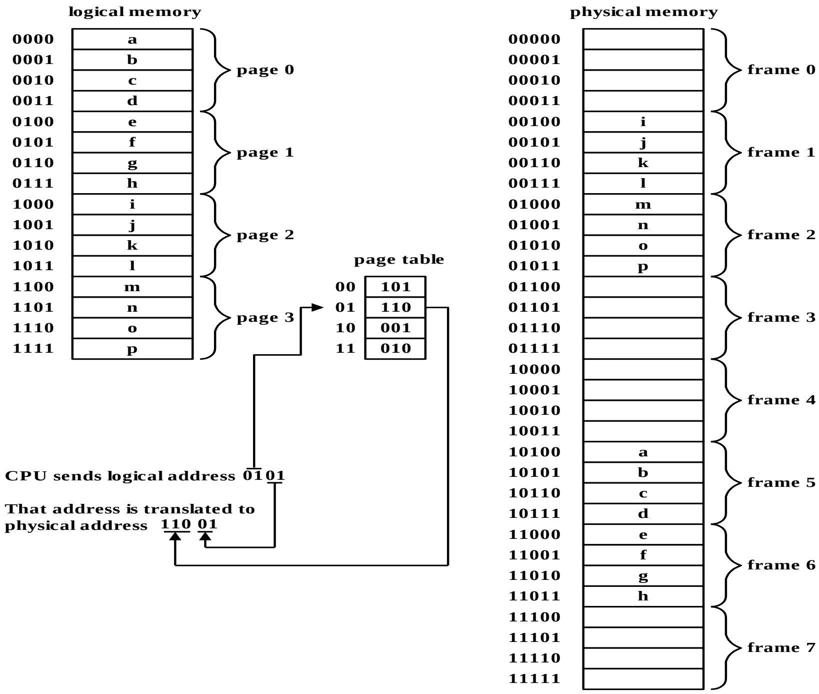
Main Memory Size = 32 bytes

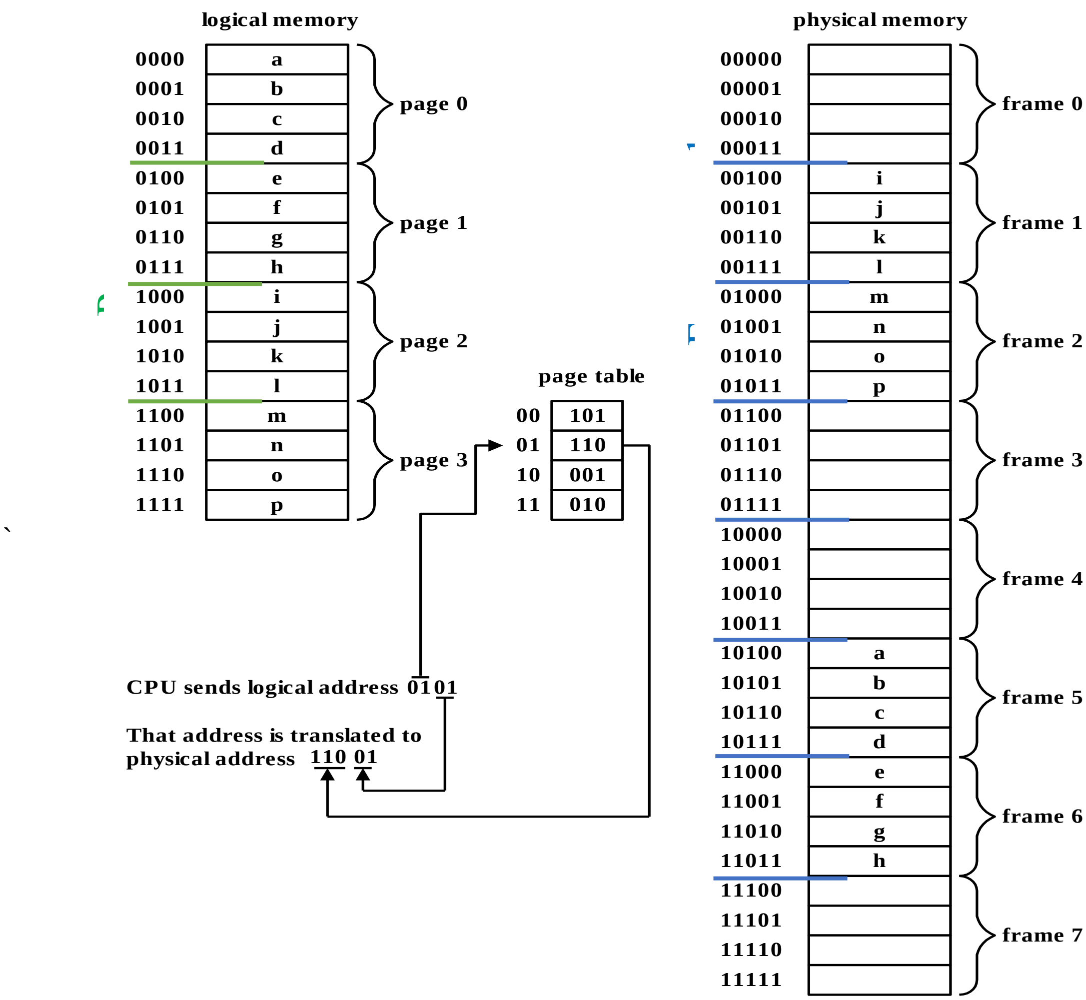
Process Size = 16 bytes

Page or Frame Size = 4 bytes

No. of Process Pages= 4 pages

No. of MM Frames = 8 frames


logical memory
physical memory
page table
page 0
page 1
page 2
page 3
frame 0
frame 1
frame 2
frame 3
frame 4
frame 5
frame 6
frame 7
CPU sends logical address 01 01
That address is translated to physical address 110 01


logical memory
physical memory
page table
page 0
page 1
page 2
page 3
frame 0
frame 1
frame 2
frame 3
frame 4
frame 5
frame 6
frame 7
CPU sends logical address 01 01
That address is translated to physical address 110 01

- There is **no external fragmentation** in paging since the operating system can allocate any free frame to a process that needs it.
- However, it is **possible to have internal fragmentation** if the memory requirements of a process do not happen to fall on page boundaries.
- In other words, the **last page may not completely fill up a frame**.
- **Example:** Page Size = 2,048 bytes

  Process Size = 72,766 bytes

No. of Pages = Process Size / Page Size = 72,766/2,048

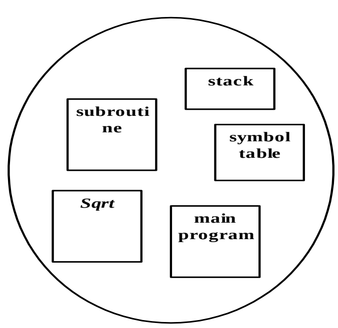
= **36 pages** *(35 pages plus 1,086 bytes)*

**Internal Fragmentation:** 2,048 - 1,086 = **962**

- In the worst case, a process would need *n* pages plus one byte. It would be allocated *n* + 1 frames, resulting in an internal fragmentation of almost an entire frame.

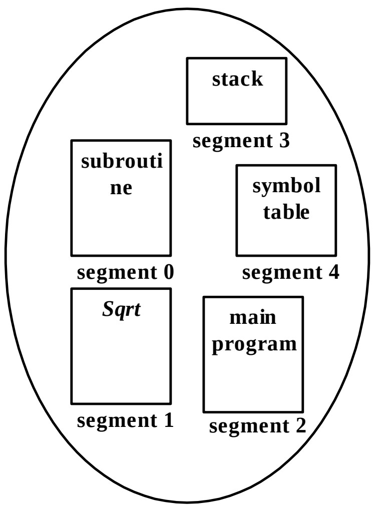
## SEGMENTATION

- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.
- When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
- Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
- A program segment contains the program's main function, utility functions, data structures, and so on.
- A logical address space is a *collection of segments*. Each segment has a name and a length. Addresses specify the name of the segment or its *base address* and the offset within the segment.




LOGICAL ADDRESS SPACE

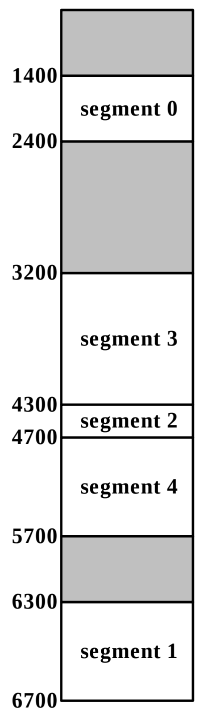
- The OS maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory.
- For each segment, the table stores the *starting address of the segment* ***(base)*** and *the length of the segment* ***(limit)***.




LOGICAL ADDRESS SPACE

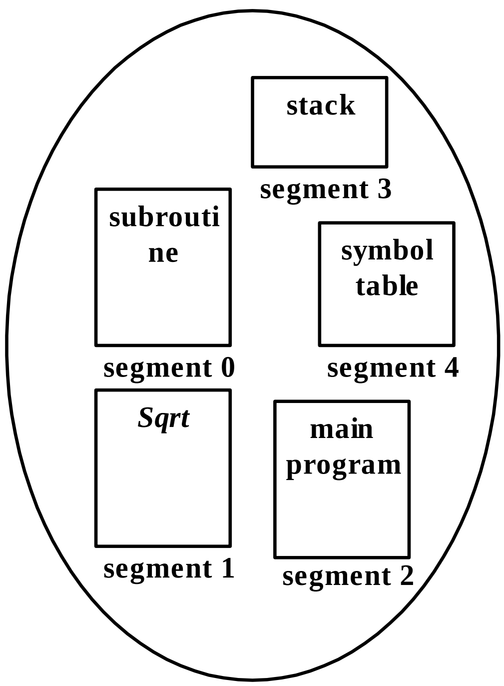
| | limit | base |
|---|---|---|
| 0 | 1000 | 1400 |
| 1 | 400 | 6300 |
| 2 | 400 | 4300 |
| 3 | 1100 | 3200 |
| 4 | 1000 | 4700 |

SEGMENT TABLE




PHYSICAL MEMORY

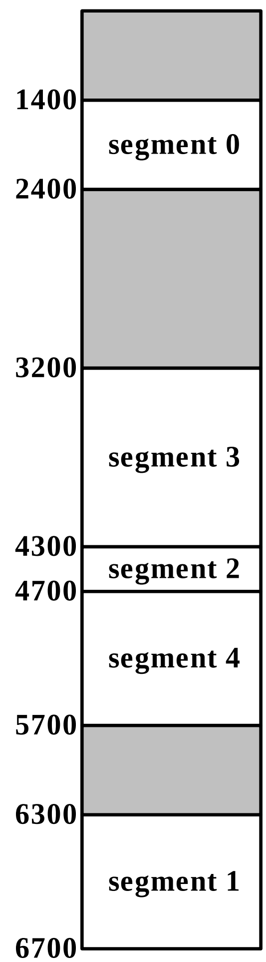
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to the operating system since this segment is only 1000 bytes long.




LOGICAL ADDRESS SPACE

| | limit | base |
|---|---|---|
| 0 | 1000 | 1400 |
| 1 | 400 | 6300 |
| 2 | 400 | 4300 |
| 3 | 1100 | 3200 |
| 4 | 1000 | 4700 |

SEGMENT TABLE




PHYSICAL MEMORY