IOS103

OPERATING SYSTEM MEMORY MANAGEMENT

Objectives

At the end of the course, the student should be able to:

- *Define memory management;*
- *Discuss the concept of address binding;*
- *Define logical and physical address space;*
- *Discuss swapping, multiple partitions, paging and segmentation.*

Memory Management

- The sharing of the CPU by several processes requires that the operating system keeps several processes (including the OS itself) in main memory at the same time.

- The operating system should therefore have algorithms for facilitating the sharing of main memory among these processes (***memory management***).

Memory Management

The Concept of Address Binding

- 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***.

Memory Management

The Concept of Address Binding

- The normal procedure is to select one of the processes in the input queue and to load the process into memory.

- A user process may reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process does not need to be 00000.

Memory Management

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.

Memory Management

Logical and Physical Address Space

- The run-time mapping from logical to physical addresses is done by the ***memory management unit*** (MMU), which is a hardware device.

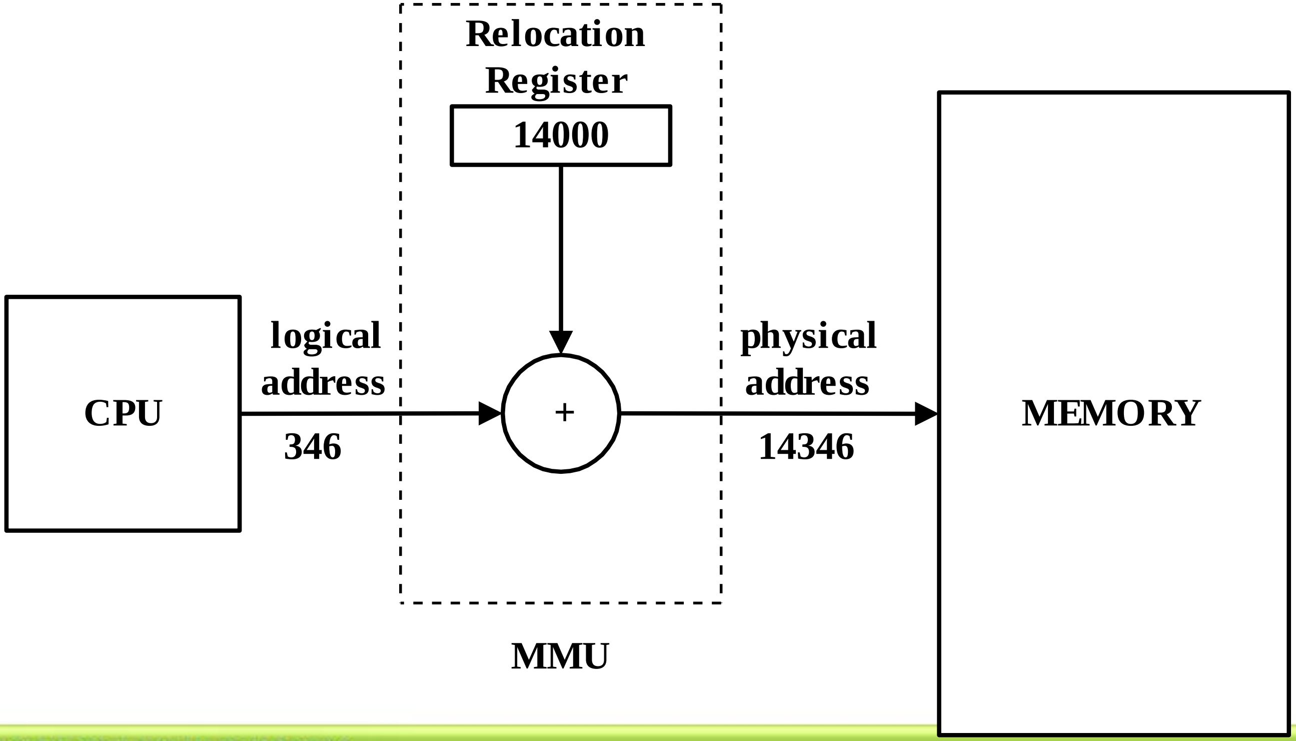
Memory Management

Logical and Physical Address Space

- The hardware support necessary for this scheme is similar to the ones discussed earlier. The base register is now the ***relocation register***. The value in the relocation register is added to every address generated by the user process at the time it is sent to memory.

Memory Management

For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.



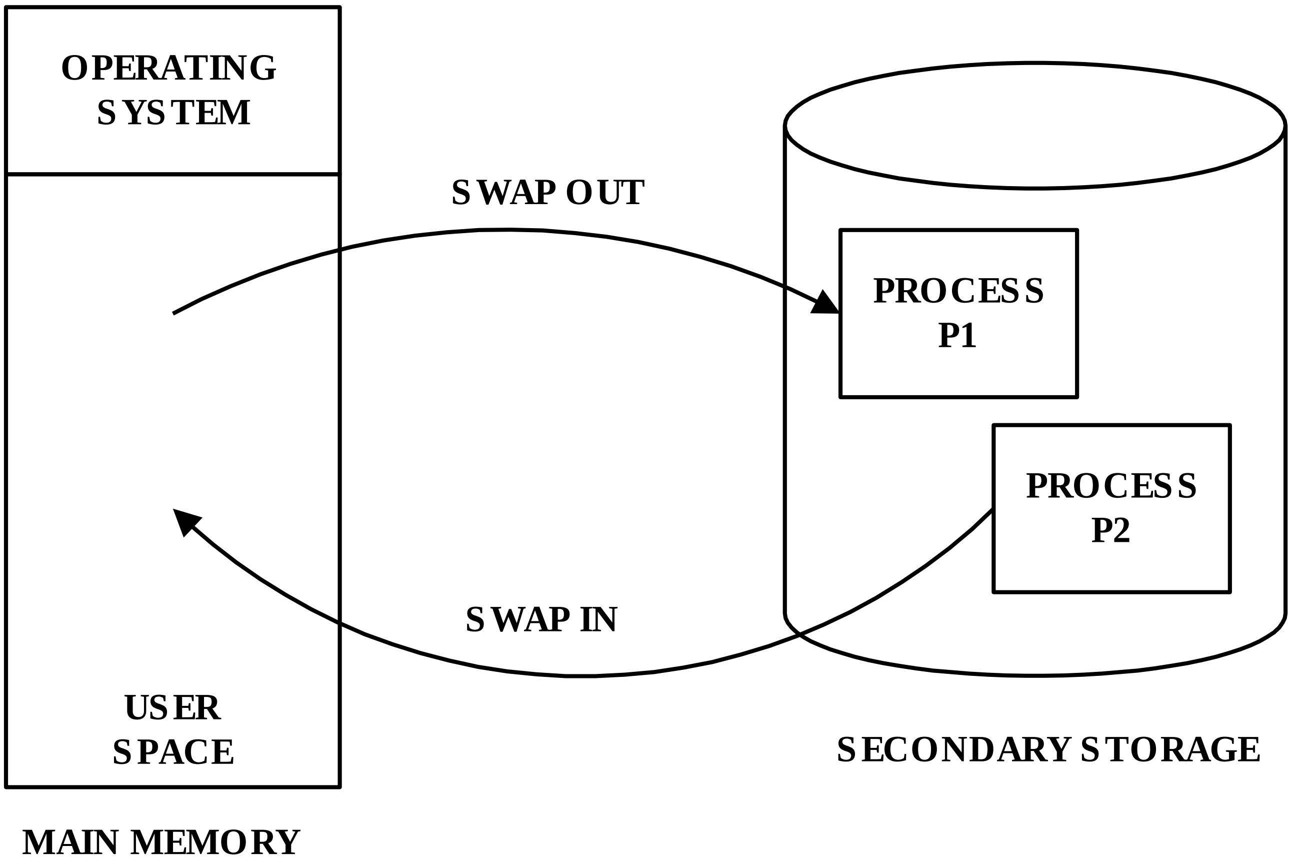
Memory Management

SWAPPING

- To facilitate multiprogramming, the OS can temporarily *swap* a process out of memory to a fast secondary storage (fixed disk) and then brought back into memory for continued execution.

Memory Management

SWAPPING



Memory Management

SWAPPING

- For example, assume a multiprogrammed environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed.

Memory Management

SWAPPING

- Take note that the quantum must be sufficiently large that reasonable amounts of computing are done between swaps.

- The context-switch time in swapping is fairly high.

Memory Management

Example:

Size of User Process	=	1 MB
	=	1,048,576 bytes
Transfer Rate of Secondary Storage	=	5 MB/sec
	=	5,242,880 bytes/sec

The actual transfer rate of the 1 MB process to or from memory takes:

1,048,576 / 5,242,880 = 200 ms

Memory Management

SWAPPING

- Assuming that no head seeks are necessary and an average latency of 8 ms, the swap time tales 208 ms. Since it is necessary to swap out and swap in, the total swap time is then about 416 ms.

Memory Management

SWAPPING

- For efficient CPU utilization, the execution time for each process must be relative long relative to the swap time. Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 416 ms.
- Swapping is constrained by other factors as well. A process to be swapped out must be completely idle.

Memory Management

MULTIPLE PARTITIONS

- In an actual multiprogrammed environment, many different processes reside in memory, and the CPU switches rapidly back and forth among these processes.

- Recall that the collection of processes on a disk that are waiting to be brought into memory for execution form the input or job queue.

Memory Management

MULTIPLE PARTITIONS

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

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

Memory Management

MULTIPLE 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.

Memory Management

MULTIPLE PARTITIONS

- There are two major memory management schemes possible in handling multiple partitions:

1. ***Multiple Contiguous Fixed Partition Allocation***

Example:

MFT Technique (***M***ultiprogramming with a ***F***ixed number of ***T***asks) originally used by the IBM OS/360 operating system.

Memory Management

MULTIPLE PARTITIONS

- There are two major memory management schemes possible in handling multiple partitions:

2. ***Multiple Contiguous Variable Partition Allocation***

Example:

MVT Technique (***M***ultiprogramming with a ***V***ariable number of ***T***asks)

Memory Management

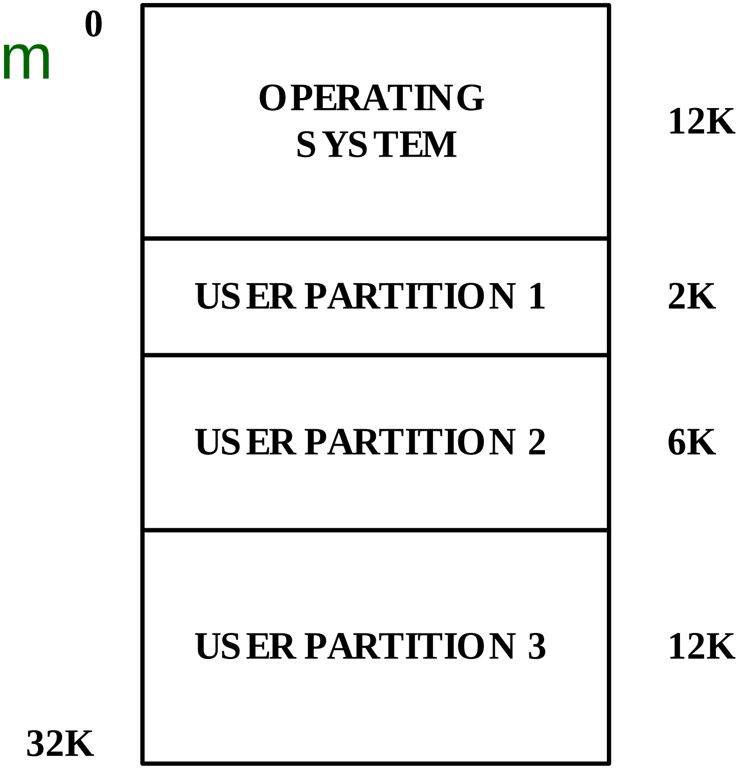
MULTIPLE PARTITIONS

- Fixed Regions (MFT)
- In MFT, the region sizes are fixed, and do not change as the system runs.
- As jobs enter the system, they are put into a job queue. The job scheduler takes into account the memory requirements of each job and the available regions in determining which jobs are allocated memory.

Memory Management

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

12K for the operating system
2K for very small processes
6K for average processes
12K for large jobs



0

OPERATING SYSTEM 12K

USER PARTITION 1 2K

USER PARTITION 2 6K

USER PARTITION 3 12K

32K

Memory Management

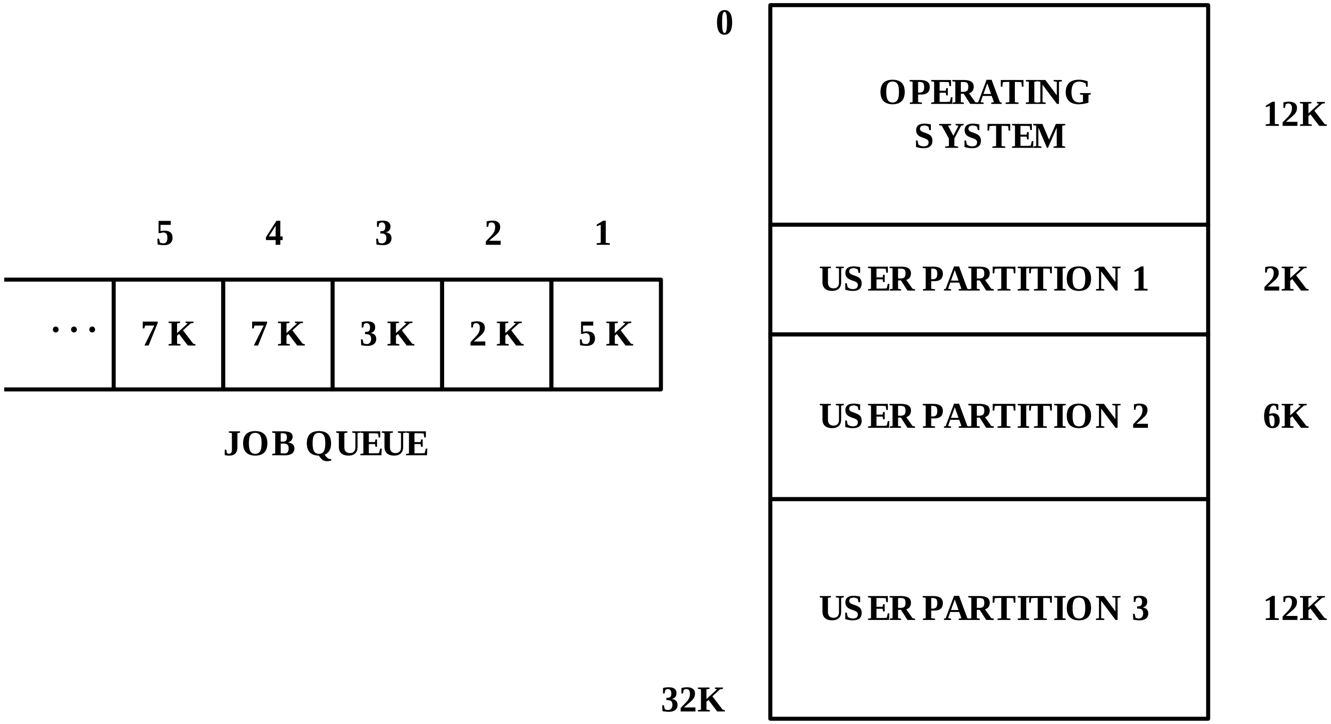
MULTIPLE PARTITIONS

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

- The job scheduler then selects a job to place in memory depending on the memory available.

Memory Management

Example:



5	4	3	2	1
. . . 7 K	7 K	3 K	2 K	5 K

JOB QUEUE

0	OPERATING SYSTEM	12K
	USER PARTITION 1	2K
	USER PARTITION 2	6K
	USER PARTITION 3	12K
32K		

Memory Management

A typical memory management algorithm would:

1. Assign Job 1 to User Partition 2
2. Assign Job 2 to User Partition 1
3. Job 3 (3K) needs User Partition 2 (6K) since it is too small for User Partition 3 (12K). Since Job 2 is still using this partition, Job 3 should wait for its turn.
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.

This algorithm is known as the ***best-fit only*** algorithm.

Memory Management

MULTIPLE PARTITIONS

- One flaw of the best-fit only 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 available*** 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.

Memory Management

MULTIPLE PARTITIONS

- Another option is to allow jobs that are near the rear of the queue to go ahead of other jobs that cannot proceed due to any mismatch in size. However, this will break the FCFS rule.

Memory Management

MULTIPLE PARTITIONS

Other problems with MFT:

•1. What if a process requests for more memory?

Possible Solutions:

A] kill the process

B] return control to the user program with an “out of memory” message

C] reswap the process to a bigger partition, if the system allows dynamic relocation

Memory Management

MULTIPLE PARTITIONS

- 2. How does the system determine the sizes of the partitions?

- 3. MFT results in ***internal*** and ***external fragmentation*** which are both sources of memory waste.

Memory Management

MULTIPLE PARTITIONS

- Internal fragmentation occurs when a process requiring *m* memory locations reside in a partition with *n* memory locations where *m* < *n*. The difference between *n* and *m* (*n* - *m*) is the amount of internal fragmentation.

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

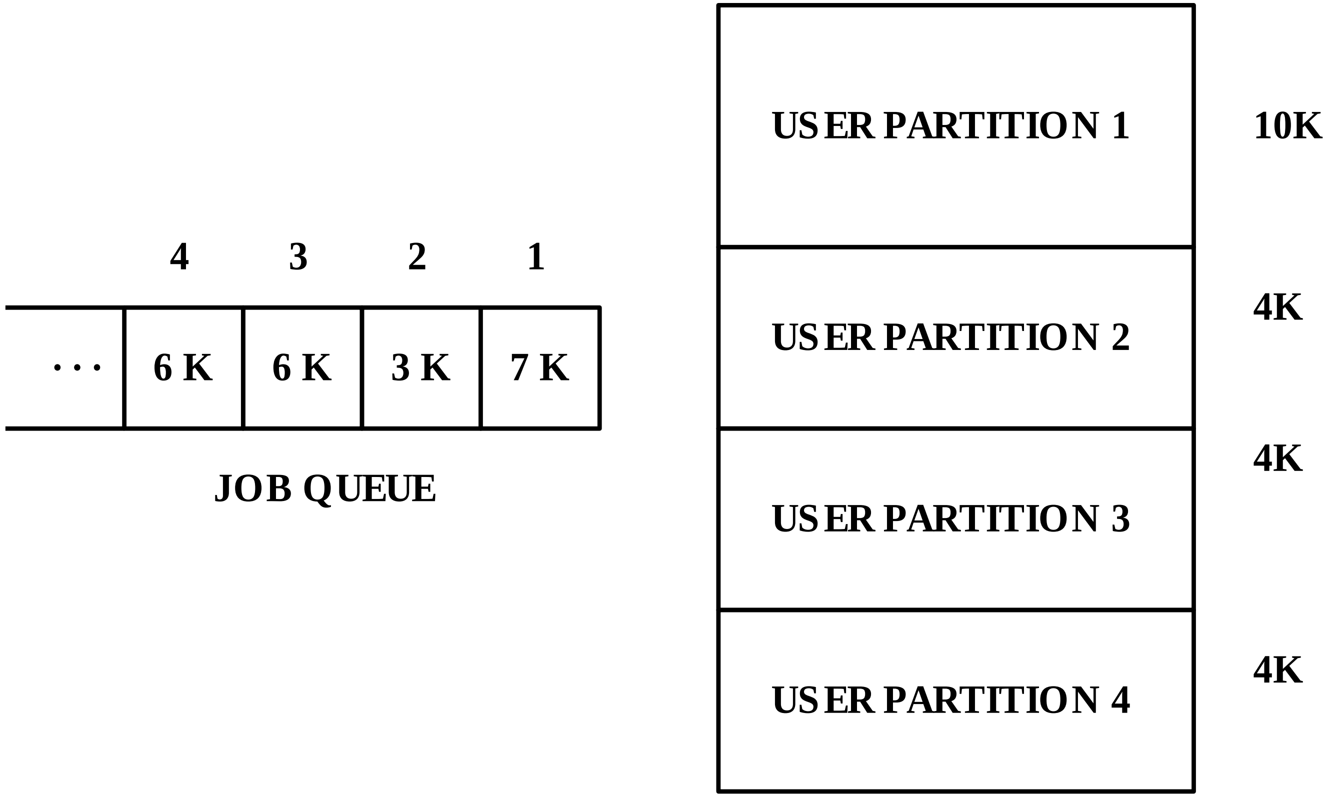
Memory Management

MULTIPLE PARTITIONS

- Partition size selection affects internal and external fragmentation since if a partition is too big for a process, then internal fragmentation results. If the partition is too small, then external fragmentation occurs. Unfortunately, with a dynamic set of jobs to run, there is probably no one right partition for memory.

Memory Management

Example:



	4	3	2	1
. . .	6 K	6 K	3 K	7 K

JOB QUEUE

USER PARTITION 1	10K
USER PARTITION 2	4K
USER PARTITION 3	4K
USER PARTITION 4	4K

Memory Management

MULTIPLE PARTITIONS

- Only Jobs 1 and 2 can enter memory (at partitions 1 and 2). During this time:

  I.F. = (10 K - 7 K) + (4 K - 3 K)
  = 4 K
  E.F. = 8 K

- Therefore:

  Memory Utilization = 10/22 x 100
  = 45.5%

- What if the system partitions memory as 10:8:4 or 7:3:6:6?

Memory Management

Variable Partitions (MVT)

- In 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 operating system 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.

Memory Management

Example:

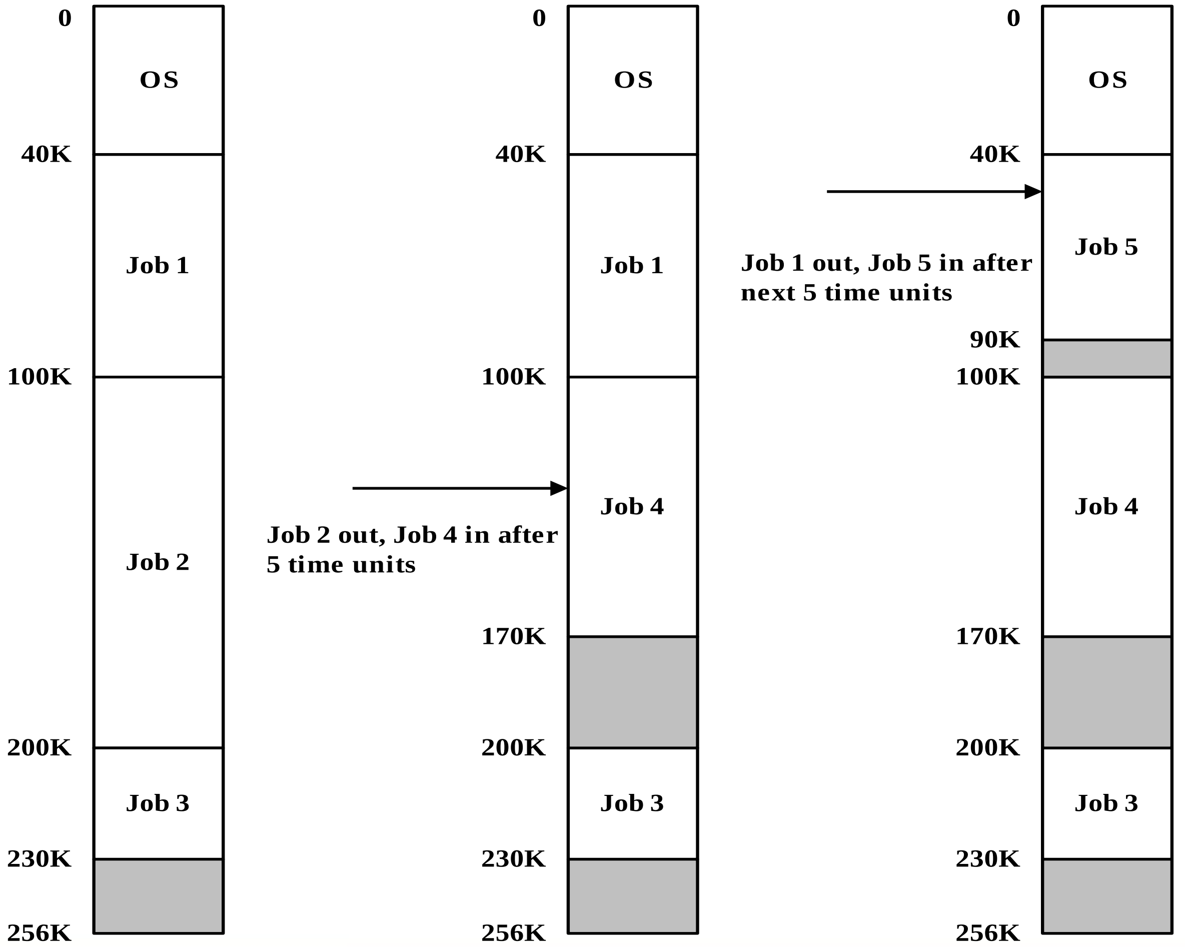
Assume that memory has 256 K locations with the operating system residing at the first 40 K locations. Assume further that the following jobs are in the job queue:

JOB	MEMORY	COMPUTE TIME
1	60K	10 units
2	100K	5 units
3	30K	20 units
4	70K	8 units
5	50K	15 units

The system again follows the FCFS algorithm in scheduling processes.

Memory Management

Example Memory Allocation and Job Scheduling for MVT



Memory Management

Variable Partitions (MVT)

- This example illustrates several points about MVT:
- 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).

Memory Management

Variable Partitions (MVT)

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.

Memory Management

Variable Partitions (MVT)

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.

Memory Management

Variable Partitions (MVT)

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.

Memory Management

Variable Partitions (MVT)

- 3. 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.
- 4. When a job terminates, it releases its block of memory and the operating system returns it in the set of holes.
- 5. If the new hole is adjacent to other holes, the system merges these adjacent holes to form one larger hole.

Memory Management

Variable Partitions (MVT)

- It is important for the operating system to keep track of the unused parts of user memory or holes by maintaining a linked list. A node in this list will have the following fields:

  1. the base address of the hole
  2. the size of the hole
  3. a pointer to the next node in the list

Memory Management

Variable Partitions (MVT)

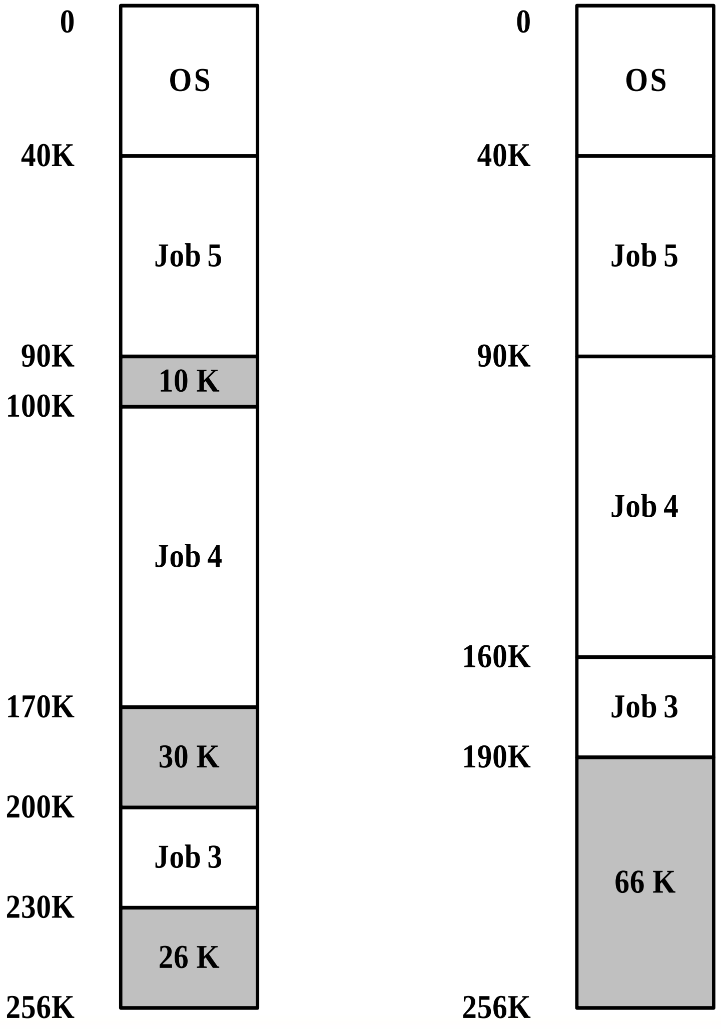
- Internal fragmentation does not exist in MVT but external fragmentation is still a problem. It is possible to have several holes with sizes that are too small for any pending job.

- 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.

Memory Management

Example:

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



Memory Management

PAGING

- MVT still suffers from external fragmentation when available memory is not contiguous, but fragmented into many scattered blocks.

- Aside from compaction, ***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 Management

PAGING

- In paging, the operating system divides main memory into fixed-sized blocks called ***frames***. The system also breaks a process into blocks called ***pages*** where 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.

Memory Management

PAGING

- 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.

Memory Management

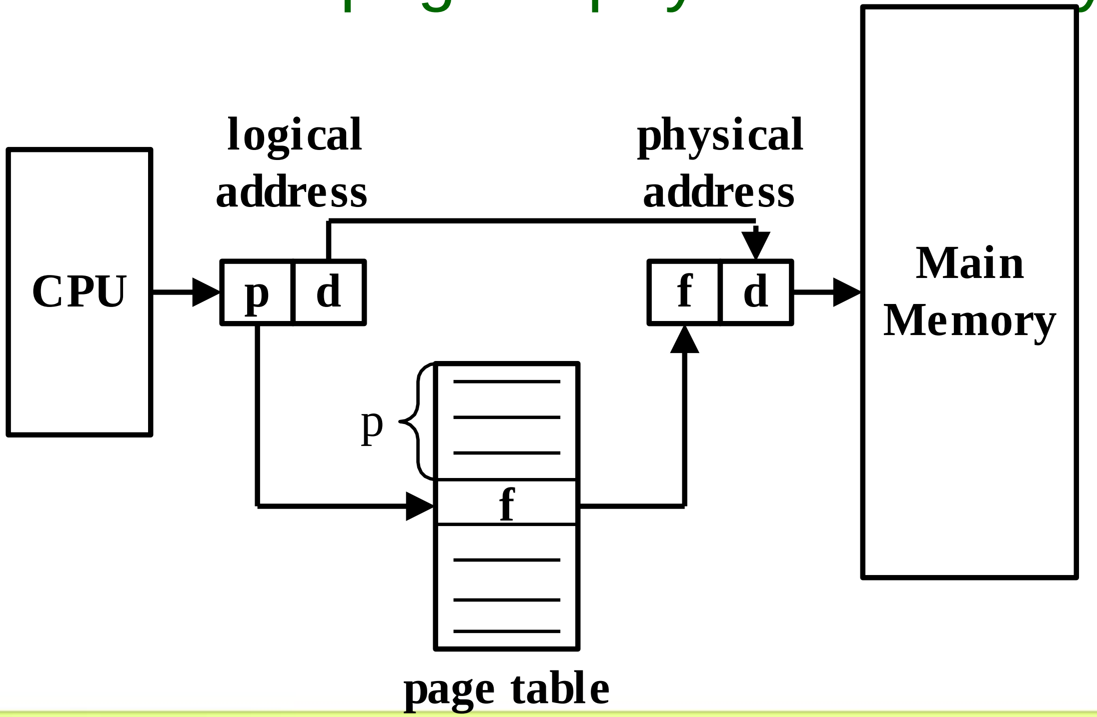
PAGING

- The operating system 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***.

Memory Management

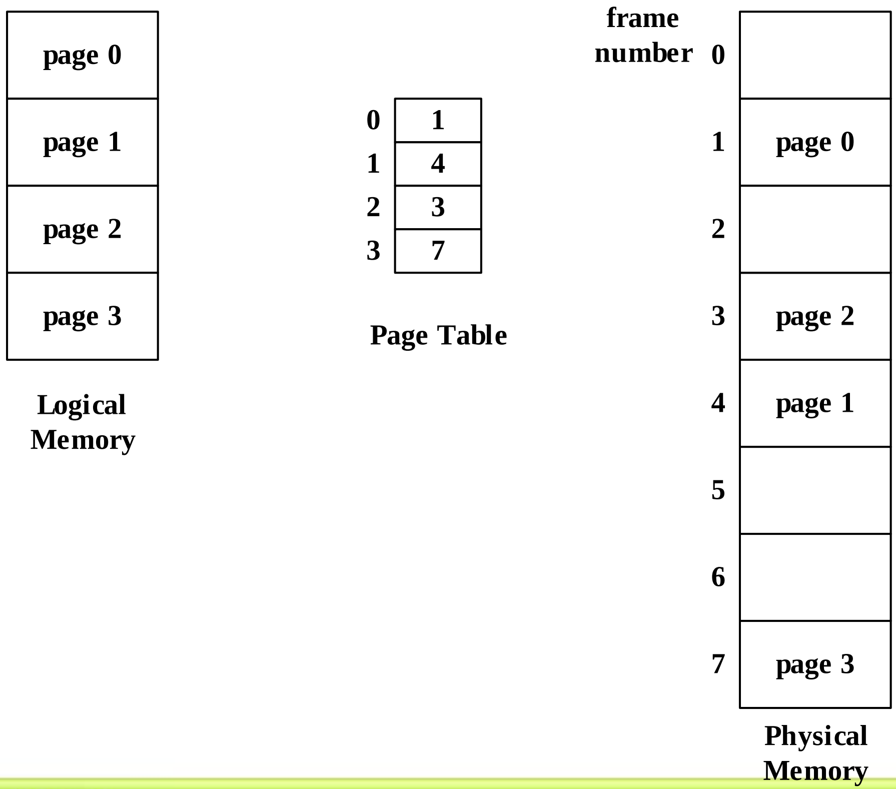
PAGING

- 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.



Memory Management

- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



page 0

page 1

page 2

page 3

Logical Memory

0	1
1	4
2	3
3	7

Page Table

frame number	
0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

Physical Memory

Memory Management

- 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*

Memory Management

- 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

Memory Management

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

0	5
1	6
2	1
3	2

Page Table

0	
1	
2	
3	
4	i
5	j
6	k
7	l
8	m
9	n
10	o
11	p
12	
13	
14	
15	
16	
17	
18	
19	
20	a
21	b
22	c
23	d
24	e
25	f
26	g
27	h
28	
29	
30	
32	

Physical Memory

Considerations:

1. Get the logical address to get the page number
2. Frame No. x no. of bytes + offset

Memory Management

PAGING

- Logical address 0 is page 0, offset 0. Indexing into the page table, it is seen that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (5 x 4 + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (5 x 4 + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus logical address 4 maps to physical address 24 (6 x 4 + 0). Logical address 13 maps to physical address 9.

Memory Management

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

Memory Management

Example:

Logical Address Format:

$$\underbrace{A_3 \quad A_2}_{\text{page number}} \quad \underbrace{A_1 \quad A_0}_{\text{page offset}}$$

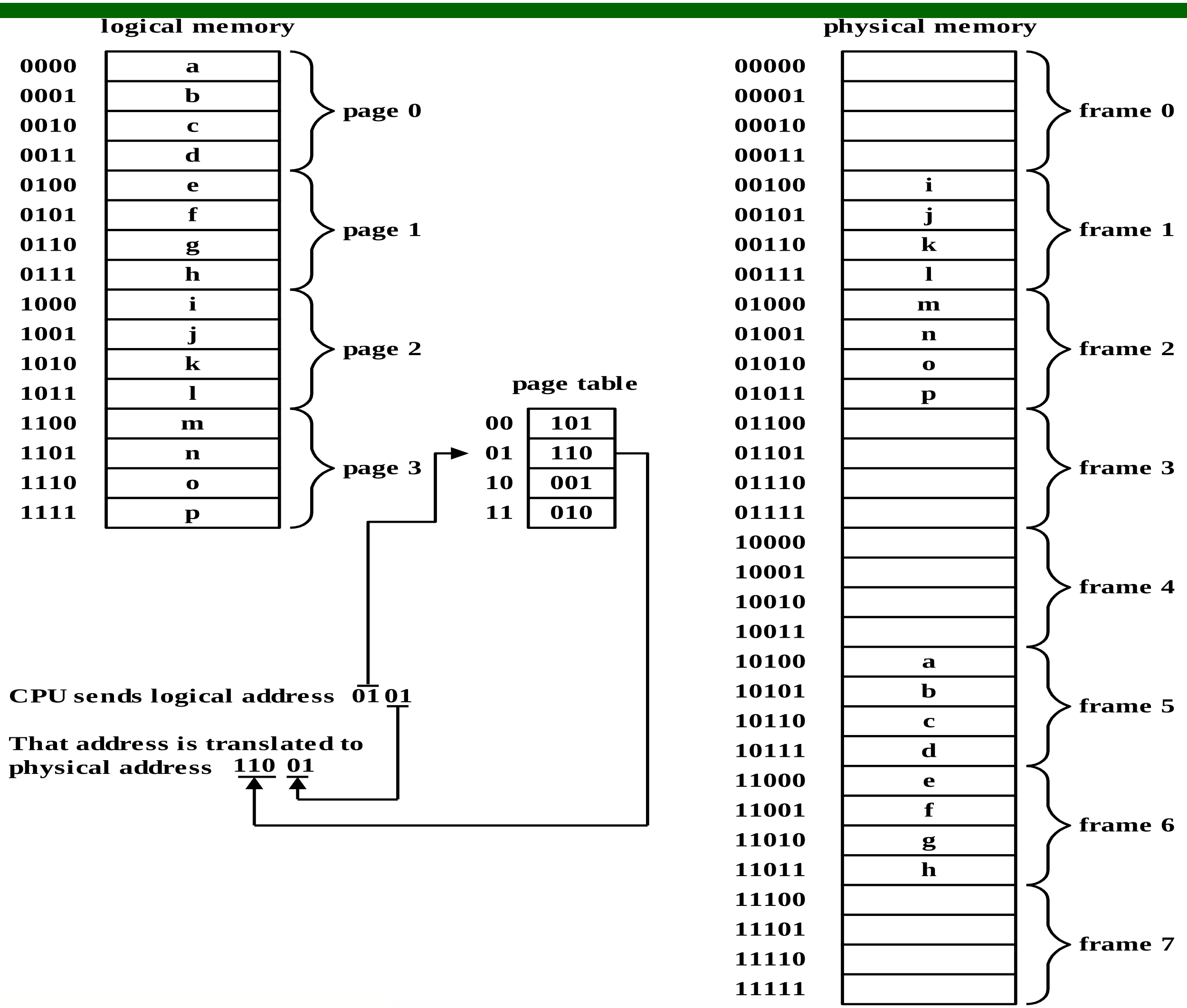
Memory Management

Example:

Physical Address Format:

$$\underbrace{A_4\ A_3\ A_2}_{\text{frame number}}\ \underbrace{A_1\ A_0}_{\text{frame offset}}$$

Memory Management



logical memory

0000	a	page 0
0001	b	
0010	c	
0011	d	
0100	e	page 1
0101	f	
0110	g	
0111	h	
1000	i	page 2
1001	j	
1010	k	
1011	l	
1100	m	page 3
1101	n	
1110	o	
1111	p	

page table

00	101
01	110
10	001
11	010

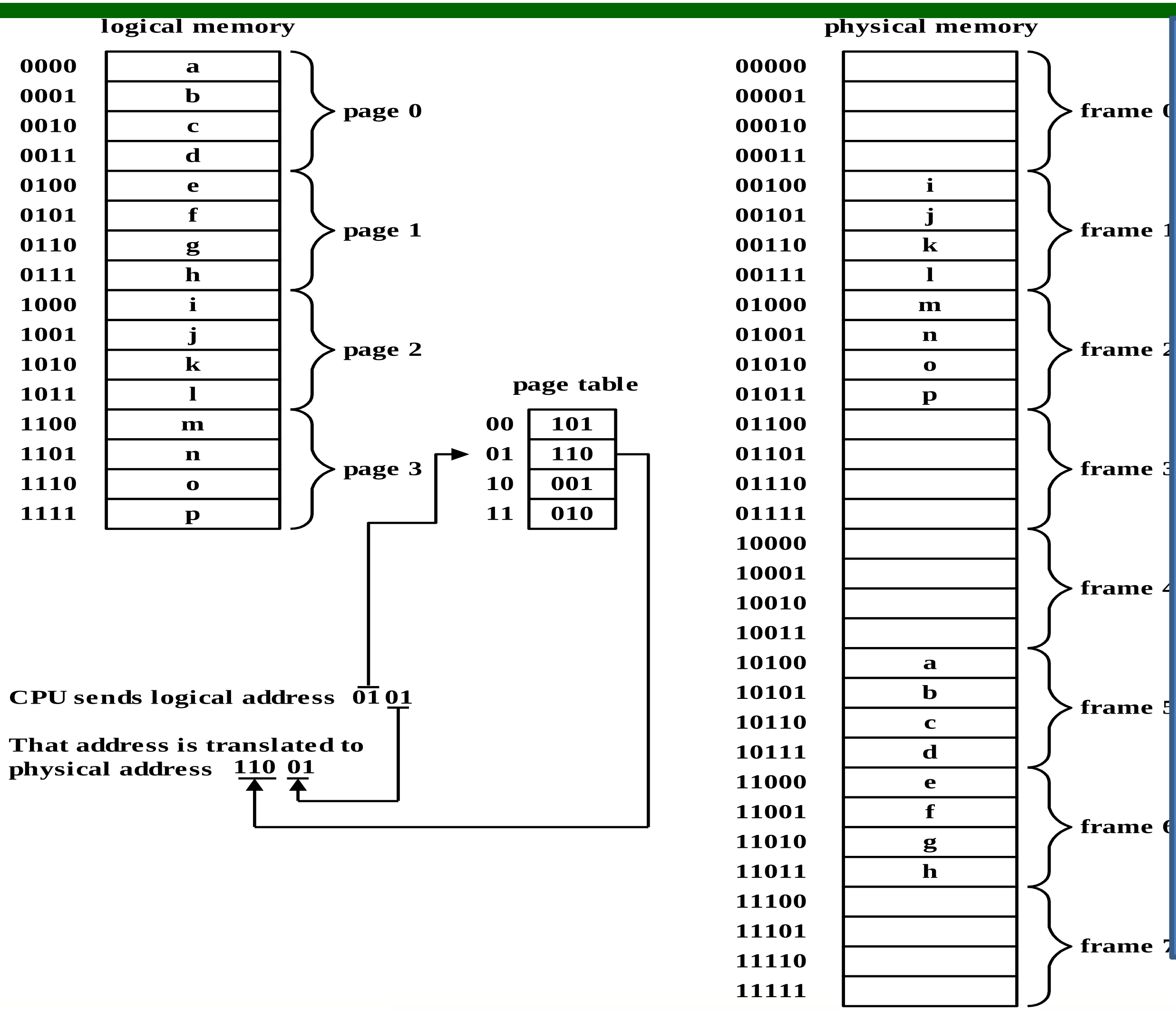
physical memory

00000		frame 0
00001		
00010		
00011		
00100	i	frame 1
00101	j	
00110	k	
00111	l	
01000	m	frame 2
01001	n	
01010	o	
01011	p	
01100		frame 3
01101		
01110		
01111		
10000		frame 4
10001		
10010		
10011		
10100	a	frame 5
10101	b	
10110	c	
10111	d	
11000	e	frame 6
11001	f	
11010	g	
11011	h	
11100		frame 7
11101		
11110		
11111		

CPU sends logical address 01 01

That address is translated to
physical address 110 01

Memory Management



logical memory

0000	a	page 0
0001	b	
0010	c	
0011	d	
0100	e	page 1
0101	f	
0110	g	
0111	h	
1000	i	page 2
1001	j	
1010	k	
1011	l	
1100	m	page 3
1101	n	
1110	o	
1111	p	

page table

00	101
01	110
10	001
11	010

physical memory

00000		frame (
00001		
00010		
00011		
00100	i	frame 1
00101	j	
00110	k	
00111	l	
01000	m	frame 2
01001	n	
01010	o	
01011	p	
01100		frame 3
01101		
01110		
01111		
10000		frame 4
10001		
10010		
10011		
10100	a	frame 5
10101	b	
10110	c	
10111	d	
11000	e	frame 6
11001	f	
11010	g	
11011	h	
11100		frame 7
11101		
11110		
11111		

CPU sends logical address 01 01

That address is translated to
physical address 110 01

Memory Management

PAGING

- 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.

Memory Management

Example:

Page Size	= 2,048 bytes
Process Size	= 72,766 bytes
No. of Pages	= 36 pages (35 pages plus 1,086 bytes)

Internal Fragmentation is 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.

Memory Management

PAGING

- If process size is independent of page size, it is expected that internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the number of data being transferred is larger.

Memory Management

PAGING

- Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the program counter) in the PCB. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

Memory Management

- The options in implementing page tables are:

1. ***Page Table Registers***

   In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with high-speed logic to make page-address translation efficient. The advantage of using registers in implementing page tables is fast mapping. Its main disadvantage is that it becomes expensive for large logical address spaces (too many pages).

Memory Management

2. ***Page Table in Main Memory***

The page table is kept in memory and a ***Page Table Base Register*** (PTBR) points to the page table. The advantage of this approach is that changing page tables requires changing only this register, substantially reducing context switch time. However, two memory accesses are needed to access a word.

Memory Management

3. ***Associative Registers***

The standard solution is to use a special, small, fast-lookup hardware cache, variously called ***Associative Registers*** or ***Translation Look-aside Buffers*** (***TLB***).

Memory Management

- The associative registers contain only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to a set of associative registers that contain page numbers and their corresponding frame numbers. If the page number is found in the associative registers, its frame number is immediately available and is used to access memory.

Memory Management

- If the page number is not in the associative registers, a memory reference to the page table must be made. When the frame number is obtained, it can be used to access memory (as desired). In addition, the page number and frame number is added to the associative registers, so that they can be found quickly on the next reference.

Memory Management

- Another advantage in paging is that processes can share pages therefore reducing overall memory consumption.

  Example:

  Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150K of code and 50K of data space, then the system would need 200K x 40 = 8,000K to support the 40 users.

  However, if the text editor code is ***reentrant*** (pure code that is non-self-modifying), then all 40 users can share this code. The total memory consumption is therefore 150K + 50K x 40 = 2,150K only.

Memory Management

PAGING

- It is important to remember that in order to share a program, it has to be reentrant, which implies that it never changes during execution.

Memory Management

SEGMENTATION

- Because of paging, there are now two ways of viewing memory. These are the ***user's view*** (logical memory) and the ***actual physical memory***. There is a necessity of mapping logical addresses into physical addresses.

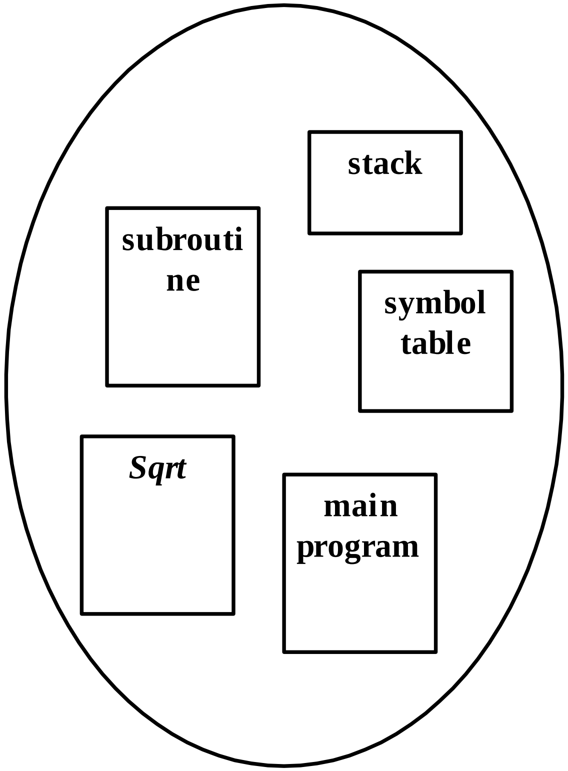
Memory Management

SEGMENTATION

- Logical Memory
  - A user or programmer views memory as a collection of variable-sized segments, with no necessary ordering among the segments.

Memory Management

- Therefore, a program is simply a set of subroutines, procedures, functions, or modules.



LOGICAL ADDRESS SPACE

Memory Management

SEGMENTATION

- Each of these segments is of variable-length; the size is intrinsically defined by the purpose of the segment in the program. The user is not concerned whether a particular segment is stored before or after another segment. The OS identifies elements within a segment by their offset from the beginning of the segment.

Memory Management

SEGMENTATION

Example:

The Intel 8086/88 processor has four segments:

1. The Code Segment
2. The Data Segment
3. The Stack Segment
4. The Extra Segment

Memory Management

SEGMENTATION

- ***Segmentation*** is the memory-management scheme that supports this user's view of memory. 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.

Memory Management

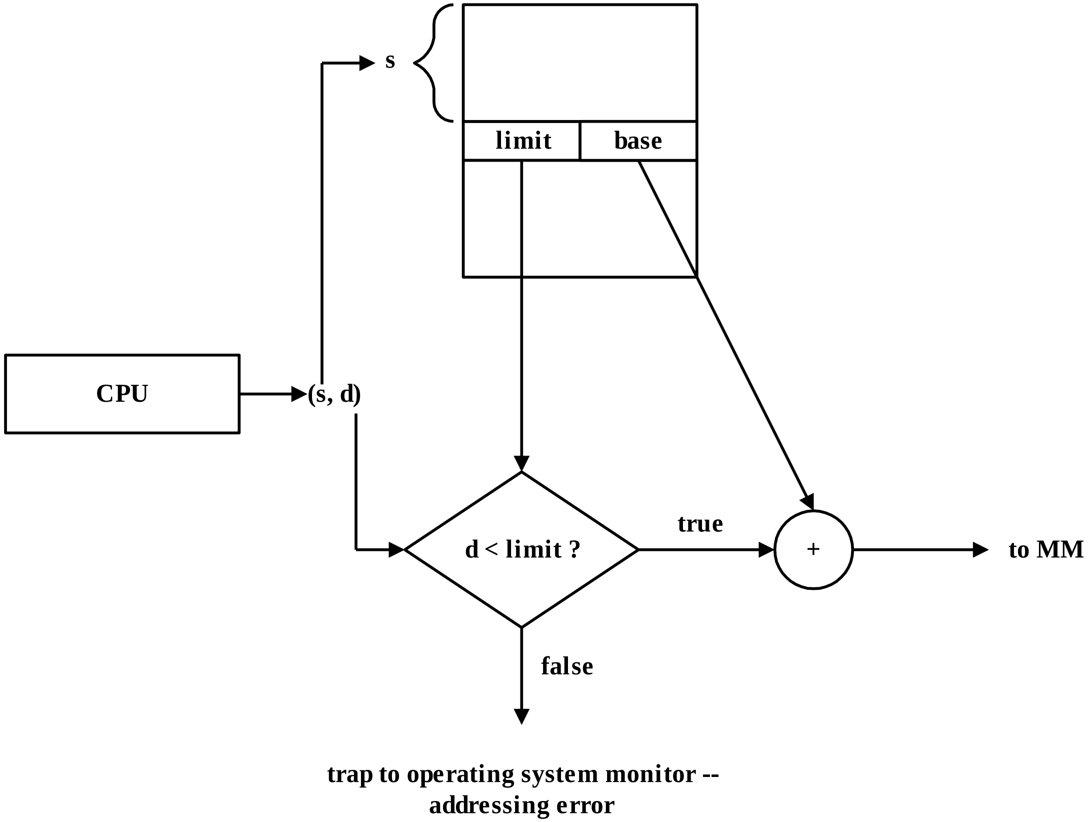
SEGMENTATION

Example:

To access an instruction in the Code Segment of the 8086/88 processor, a program must specify the base address (the CS register) and the offset within the segment (the IP register).

Memory Management

The mapping of logical address into physical address is possible through the use of a ***segment table***.



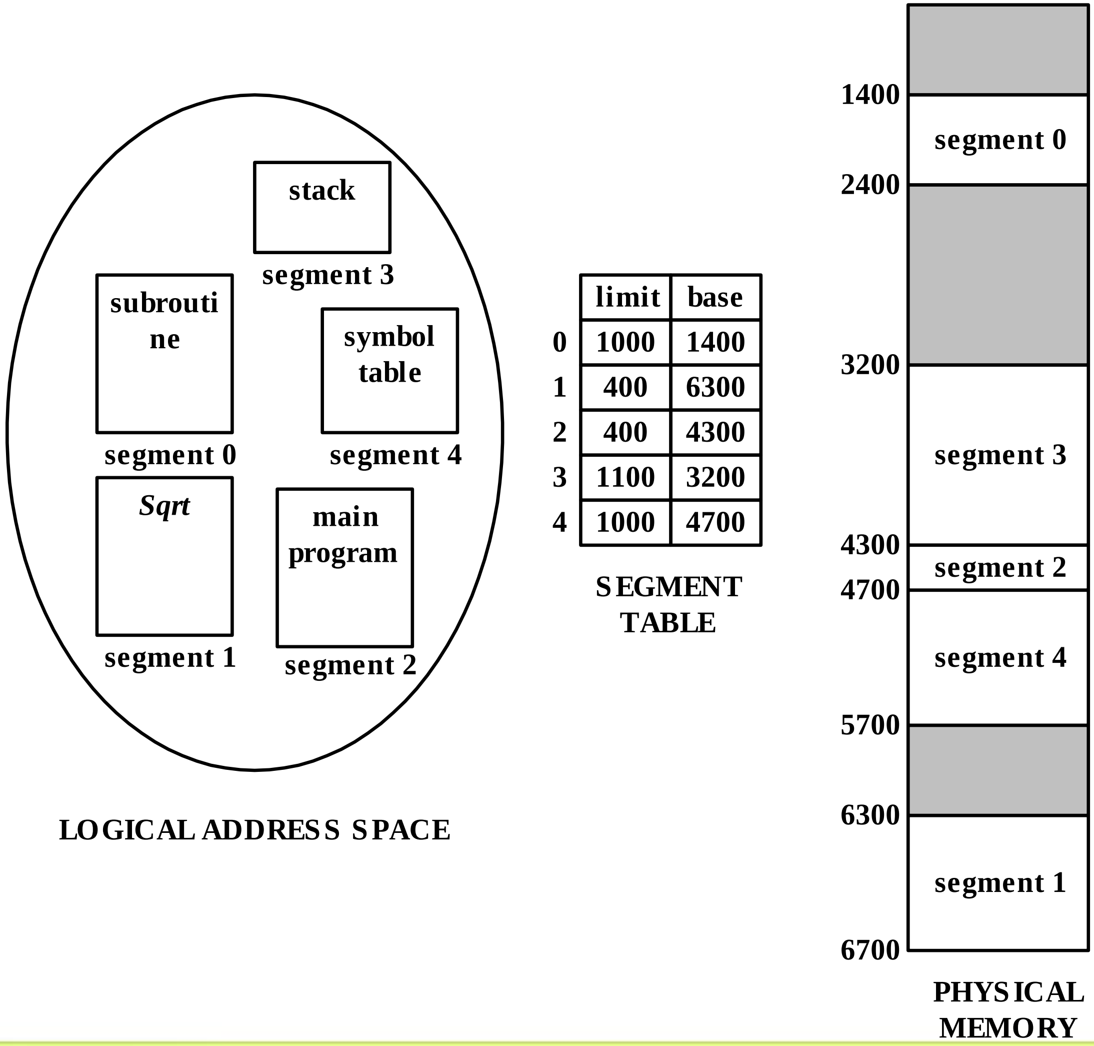
Memory Management

SEGMENTATION

- A logical address consists of two parts: a segment number *s*, and an offset into the segment, *d*. The segment number is an index into the segment table. Each entry of the segment table has a segment *base* and a segment *limit*. The offset *d* must be between 0 and *limit*.

Memory Management

Example:



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

SEGMENT TABLE

Memory Management

SEGMENTATION

- 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.