Lecture 05: Memory Management — Historical Perspective

This lecture focuses on how the operating system manages memory. The lecture proceeds under the following captions:

- Introduction
- Overview of the Development of Memory Management
- Translation from Logical to Physical Address
- Swapping
- Multiple Partitions of Memory
- Paging
- Segmentation
- Segmented Paging
- Paged Segmentation
- Summary and Concluding Remarks

5.1 Introduction

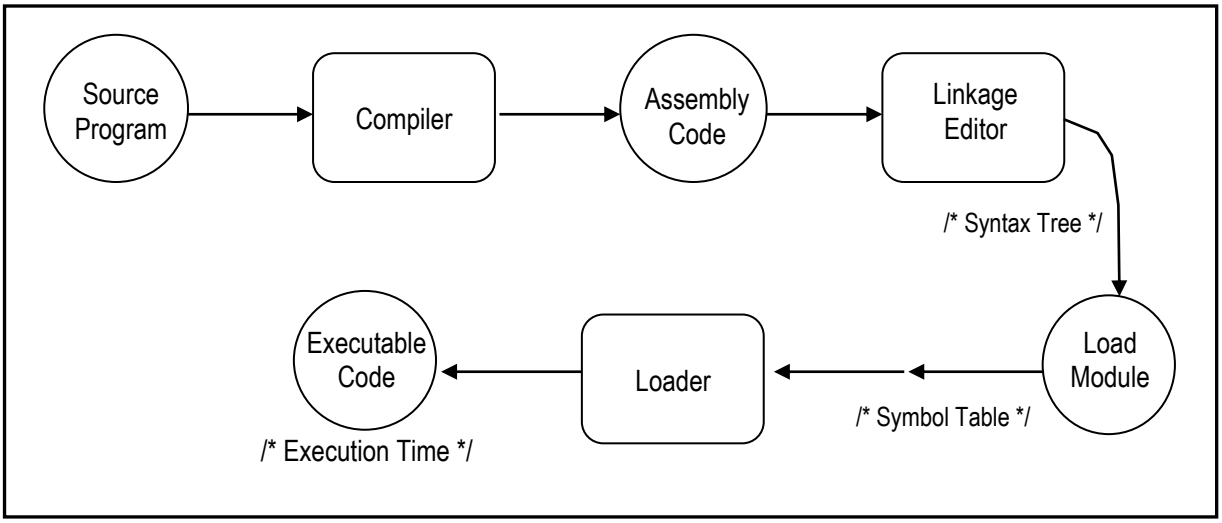
We have shown in the previous lecture how the resources of the CPU are shared among processes to achieve the objective of multiprogramming. In this lecture, we focus on how memory is shared to achieve the same objective.

Memory management is critical to the success of the OS, since both CPU and I/O peripherals must share memory. In memory management, we are interested in knowing how the OS allocates space for system programs and user programs.

To sharpen the focus, consider a HLL program which must be run on the computer system. Before this program can be executed, it must pass through various stages (as illustrated in figure 5.1).

Eventually, variables, files, fields, subroutines, instructions must be translated to memory locations. The OS must provide that critical mapping.

Figure 5.1: Illustrating the Process of Compilation and Execution of a HLL Program



5.2 Overview of the Development of Memory Management Systems

Memory management has been through several stages of development, as summarized in this section. The rest of the lecture focuses on some memory management strategies that have been used in the past. They are discussed because they form the basis for more contemporary strategies. The next lecture discusses virtual memory (VM) – the contemporary approach.

5.2.1 Bare Machine

Early systems had very primitive memory management methods. The user had control over the over the entire memory space.

Advantages of this approach include:
- Simplicity
- Low cost as no additional hardware/software is required
- Maximum flexibility to the user

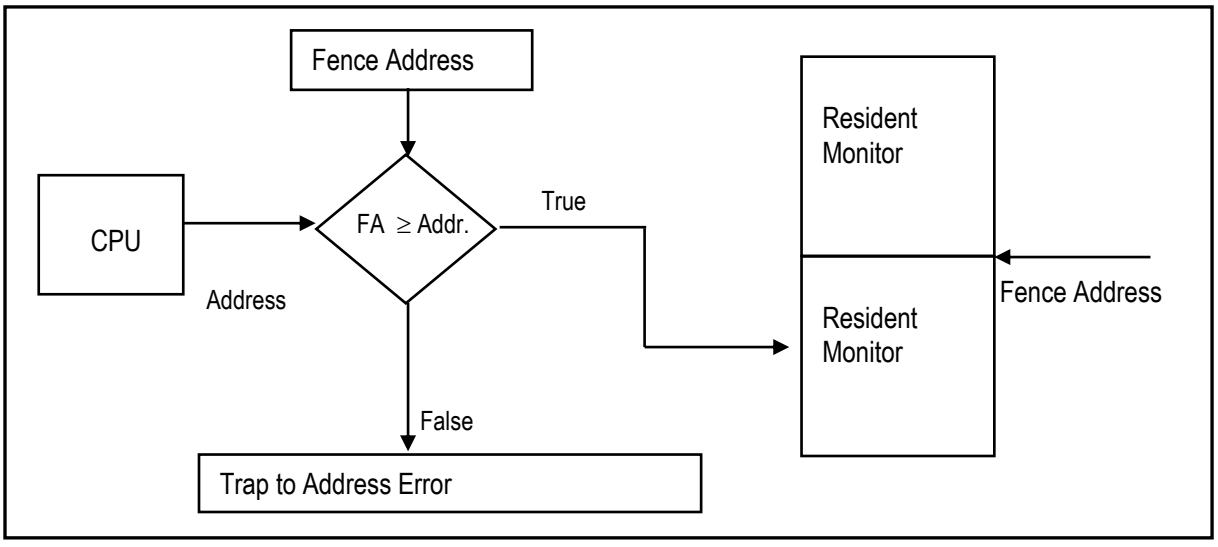
Disadvantages include:
- No services provided
- Only highly technical people could use the machine
- The approach could only be used on dedicated systems

5.2.2 Protection Hardware for Resident Monitor

With the introduction of the *resident monitor*, a boundary (fence) address is permanently held by a dedicated fence register.

All addresses are checked against the fence address to ensure that the users operate in user (*and not monitor*) area.

Figure 5.2: Illustrating Hardware Protection for Resident Monitor



5.2.2 Protection Hardware for Resident Monitor (continued)

The fence register could be loaded by a privileged instruction (in monitor mode).

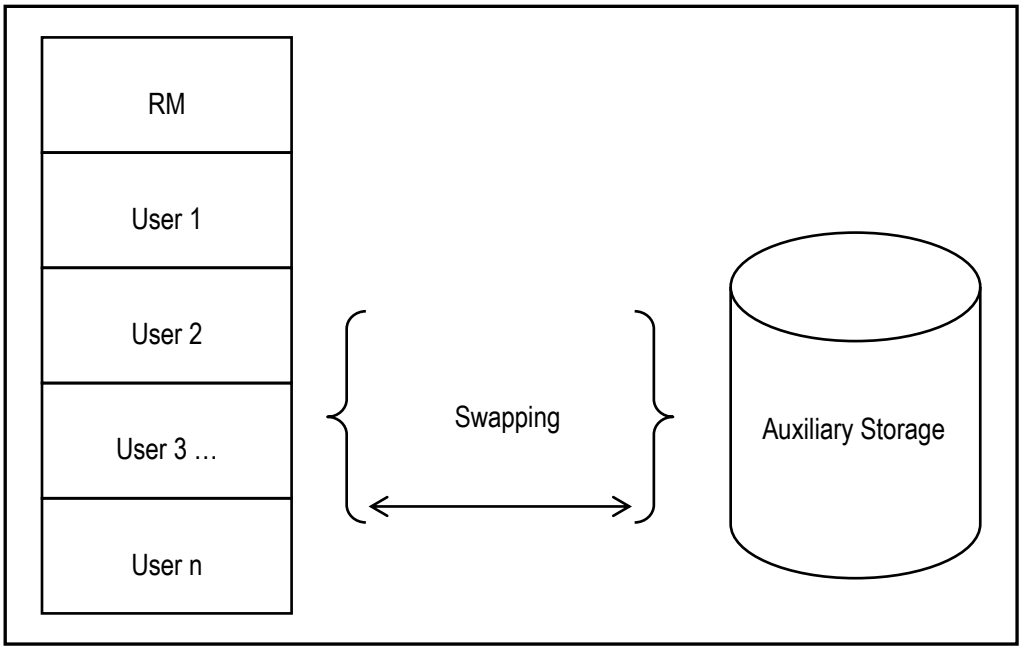
Advantages of the approach:
- An improvement on bare machine strategy
- The space for the resident monitor could grow or shrink with time and the fence address change in accordance
- The OS could run in monitor mode

Disadvantages of the approach:
- The objective of multiprogramming was still a problem
- Limited services provided by the OS
- Fence address must be static during execution of a program

5.2.3 Partitioned Memory

In this approach, memory is partitioned into user compartments, with a partition reserved for the OS. Each partition supports a contiguous memory allocation to programs in execution. If the allocated space is full, swapping allows for some (lower priority) programs to be swapped out until later.

Figure 5.3: Illustrating Partitioned Memory



5.2.3 Partitioned Memory (continued)

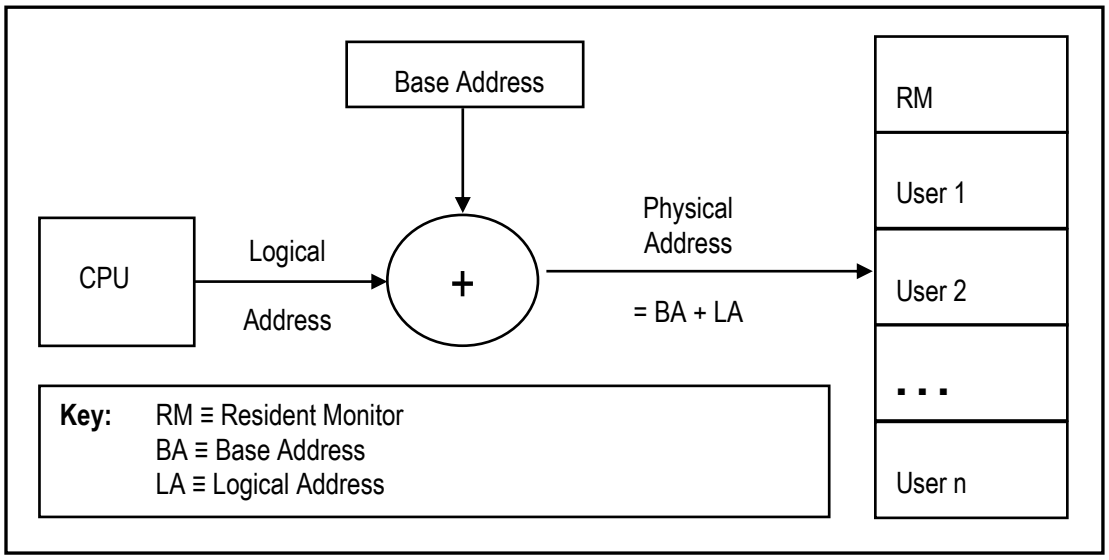
Advantages:
- Multiple processes could be facilitated without interference from each other
- The OS could still run in monitor mode

Disadvantages:
- It was possible to have a high occurrence of space wasting or swapping. The former affected efficiency; the latter affected performance.
- Fence addresses must be static during the execution of a program.

5.2.4 Relocation

This strategy improved on the previous one by implementing the fence register as a relocation base register. The value in the base register is added to the compile time address.

Figure 5.4: Illustrating the use of a Base Address



The gain here is that the fence (*base*) register may change while the program is in execution. The drawback is increased execution time due to address translation.

5.2.5 Paging and Segmentation

In paging, the program is split into pages each of equal size; in segmentation, the program is split into multiple segments, which could be of variable length.

The pages/segments of a program do not have to be contiguous. However, all pages/segments must be loaded before execution.

Advantage: This strategy ensures both efficiency and performance of the OS.

5.2.6 Virtual Memory

Virtual memory represents the culmination of the efforts of previous approaches. It is used in conjunction with paging and/or segmentation: Only currently important pages/segments of the program are retained in memory. Swapping transfers pages/segments in or out, depending on the current situation and need.

The OS uses some strategy to determine which pages/segments to swap out.

5.3 Protection

Processes must be protected from each other's memory boundaries. To achieve this, the CPU must be prevented from accessing a memory location outside of the (logical & physical) boundary of the executing process.

Address checking may be via hardware or software, the former being more efficient.

The PCB of each process is kept in registers in the CPU, or a special area of memory, reserved for this purpose.

Boundary registers must hold address limits of competing processes. Two strategies are possible:
- Lower and upper boundary
- Base and process length

5.4 Translation from Logical to Physical Address

Contemporary systems tend to implement virtual memory (to be discussed in the next lecture), which is essentially an enhancement of paging or segmentation. For these approaches, translating from logical address to physical address is an imperative. Three alternatives to address translation have been forwarded:

- Compile time
- Compile, link and load time (*sometimes combined*)
- Execution time

Compile Time Translation: The job (program) must be compiled each time it is run. This is quite expensive. Re-compilation is also necessary when the fence register changes. Additionally, the fence register must be static during program execution.

Link/Load Time Translation: The program must be swapped back to the identical space that it was first put; the fence address must be static during program execution.

Execution Time Translation: The relocation strategy previously outlined is employed:

- The fence register may change during execution
- The program can be compiled ahead of execution time
- The program can be swapped to different spaces each time it is required
- The PCB would contain additional information
  - Base Register & Program Length or
  - Boundary Registers

5.5 Swapping

As mentioned in the previous lecture, swapping involves the transfer of an object from auxiliary storage to main memory or vice-versa.

A swap-out may be required due to limitation of memory space or to service a higher priority job, or both. Swapping necessitates housekeeping procedures every time a process is swapped in or out, due to the context switch. A high degree of swapping could negatively affect the performance of the system.

Example 1:

A 20 Kb program in a system where transfer rate is 50,000 bits/sec and average latency of 8 microseconds (ms): To transfer a 20 Kb program takes:

$$\frac{20 * 10^3 * 8 \text{ bits}}{50 * 10^3 \text{ bits/sec}} + 8 \text{ ms}$$

$$= 3.2 \text{ sec.} + .008 \text{ sec} = 3.208 \text{ sec.}$$

To swap a program in or out takes 3.208 seconds; both in and out takes approximately 6.416 seconds.

5.6 Multiple Partitions of Memory

In order to facilitate multi programming, memory is partitioned into multiple partitions for the various processes which must run concurrently. Two memory management schemes were popular:

- Multiple Contiguous Fixed Partition (MFP)
- Multiple Contiguous Variable Partition (MVP)

MFP is implemented by the *multiprogramming with fixed number of tasks* (MFT) strategy. MVP is implemented by *the multiprogramming with variable number of tasks* (MVT) strategy. We will discuss both strategies next.

5.6.1 Multiprogramming with Fixed number of Tasks

In *multiprogramming with a fixed number of tasks* (MFT), all jobs entering the system are put in a job queue. The scheduler allocates memory to jobs based on the size of the job. Once loaded, the job can then compete for CPU time.

Of particular importance is how memory is allocated to jobs. Following is an account of how this is typically done, with illustrations in figures 5.5 and 5.6:

- Classify jobs according to their memory requirement (user may be required to specify memory requirements, or the system may determine it)
- Keep all jobs in one queue. The scheduler selects the next job to run and waits on large enough memory space to fit it. Allocation may be based on *best-fit*, or *first-fit*.

Best-Fit starts with the largest job and fits it in the smallest available space, until all possible allocations are made. It could also process the jobs in FIFO order, fitting each in the smallest space available. First-fit processes the jobs in FIFO order, fitting each in the first available space, until no more allocation is possible.

Figure 5.5: Illustrating MFT with Separate Queues based on Job Size

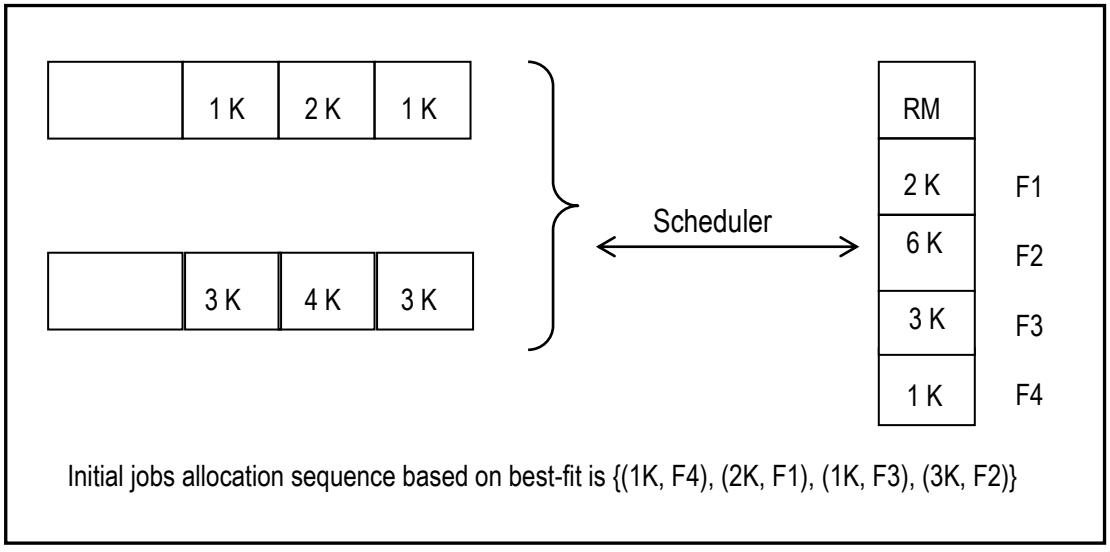
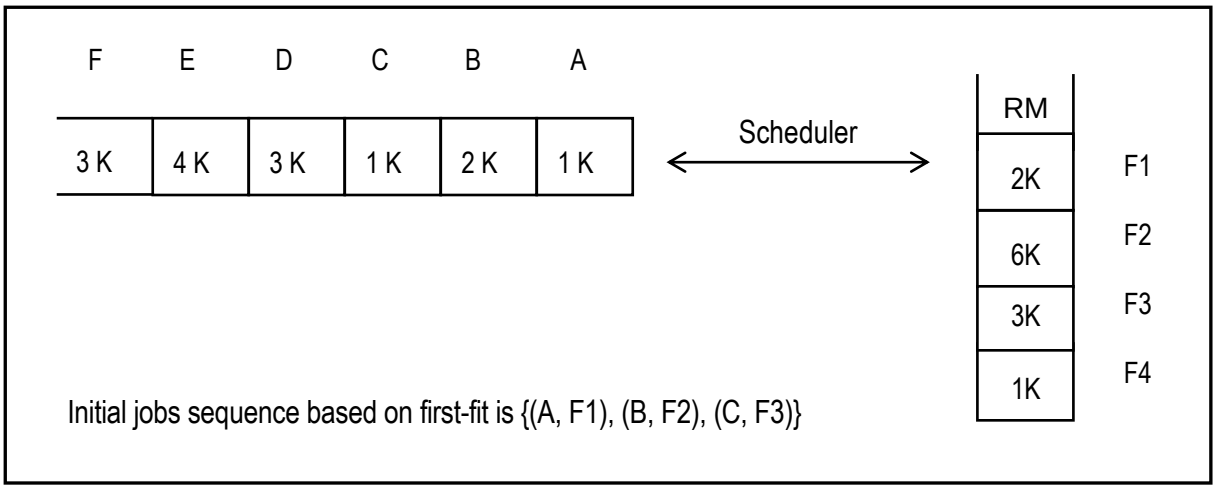


Figure 5.6: Illustrating MFT with One Queue



Advantages of MFT:
- Simple
- A fixed number of jobs at a time and this is known
- No frequent switching of boundary registers
- Fences are fixed
- The OS is catered for a number of PCB's
- The strategy can be used along with swapping.

Disadvantages of MFT:
- Job scheduling is an expensive overhead
- Fragmentation can be high:
  - Internal fragmentation occurs when space is allocated but not used
  - External fragmentation occurs when space not allocated and not used
- There could be a job that is too large to run in the system (for any of the partitions)

5.6.2 Multiprogramming with Variable number of Tasks

Multiprogramming with variable number of tasks (MVT), which was successfully implemented on UNIVAC 1108 machines, allows partition sizes to vary dynamically. As many jobs as can fit in memory are loaded.

Initially, memory is one large hole. As jobs arrive, just enough space for each job is allocated, thus gradually decreasing the size of the hole.

When a job terminates, its space is released as a hole. Adjacent holes are merged into a simple (larger) hole. Holes are allocated based on *best-fit*, *first-fit* or *worst-fit*. First-fit processes the jobs FIFO, trying to fit each job in the first available hole. Best-fit starts with the largest job and fits it in the smallest available hole. This continues until there are no jobs that can fit. Worst-fit starts with the smallest job and fits it in the largest available hole. This continues until there are no remaining jobs that can fit.

5.6.2 Multiprogramming with Variable number of Tasks (continued)

As an example, consider space of 216K and five jobs as shown in figure 5.7. Assuming first-fit allocation, figure 5.8 shows how the jobs might be allocated.

Figure 5.7: MVT Scheduling Example

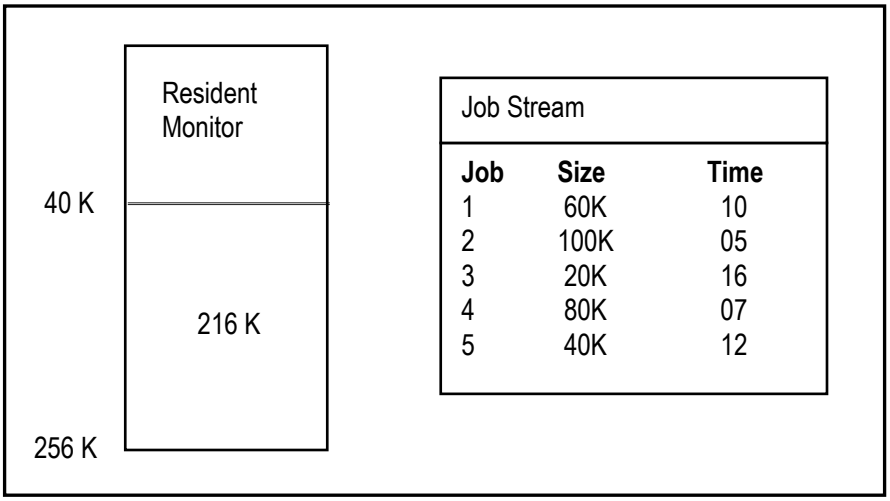
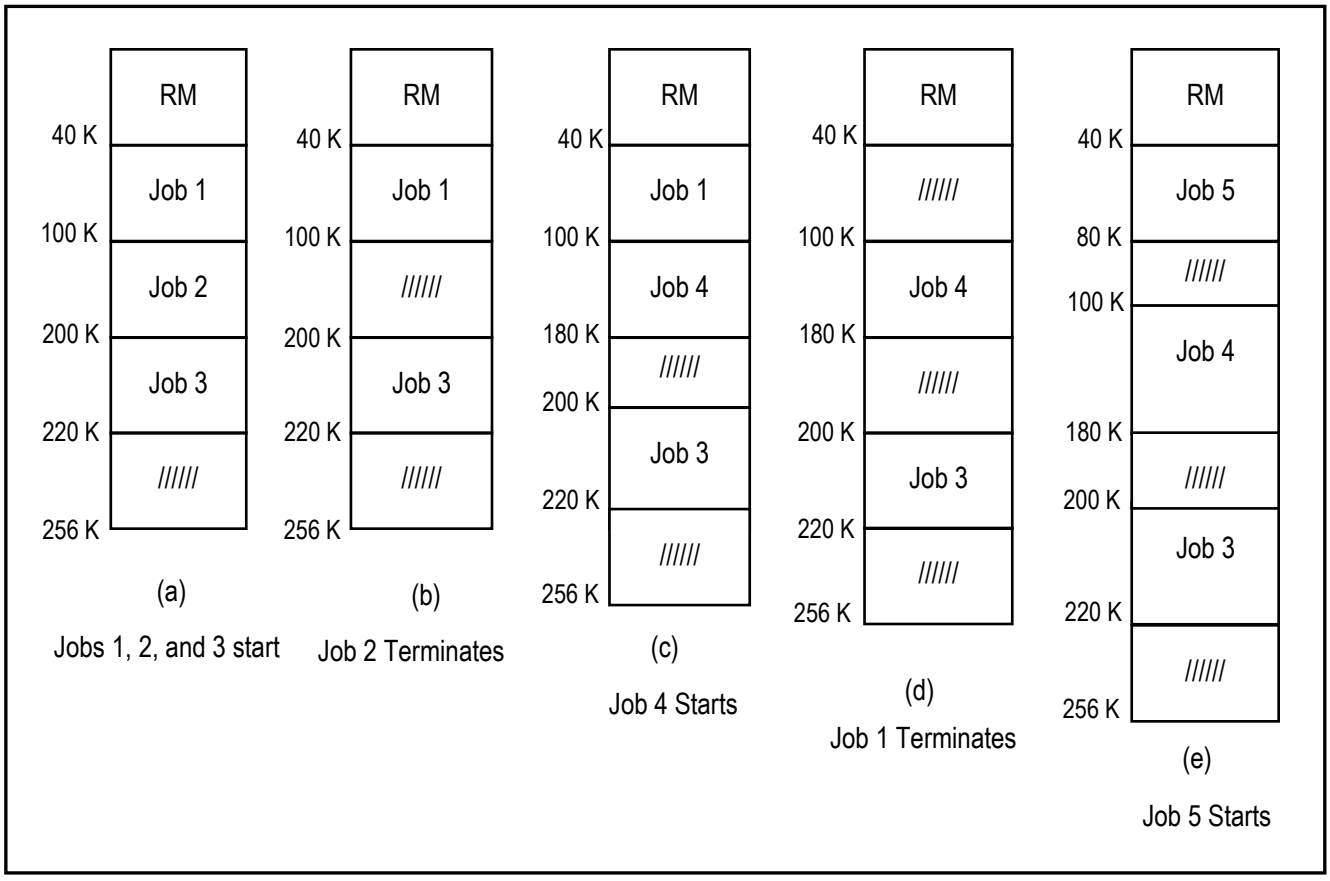


Figure 5.8: MVT Illustrating Memory Allocation for the Job Stream in Figure 5.7



5.6.2 Multiprogramming with Variable number of Tasks (continued)

Advantages of MVT:
- Significant reduction of internal fragmentation (to be confined to the last block occupied by a job).
- More efficient use of memory space.
- The strategy can be used along with swapping.
- With dynamic relocation, a job can be rolled to a different location than originally occupied.

Drawbacks of MVT:
- External fragmentation is still possible, though its likelihood is also reduced. To offset this, compaction gathers all scattered holes into one large hole. Another strategy is to hold waiting jobs for a time quantum, and allow jobs in memory to clear up; then continue.
- Starvation can occur if a large job is at the head of the queue. *Skipping* circumvents this problem.

Differences between MFT and MVT:
- MFT caters for a fixed number of jobs while MVT caters for a variable number of jobs.
- MFT uses fixed fences while MVT uses variable fences.
- Fragmentation is a major problem with MFT; with MVT, internal fragmentation is significantly reduced and external fragmentation can be corrected.

5.7 Paging

Paging (which can be considered as an enhancement of MFT) is a memory management strategy that allows a program's memory to be non-contiguous, thereby eliminating external fragmentation and reducing internal fragmentation to within a page.

5.7.1 Overview of Paging

Memory is split up into frames (typically 512 bytes or 1024 bytes or some multiple of 512 bytes). The program is split up into pages also (so that the page size is equal to the frame size) and loaded page by page — wherever the pages can fit.

The operating system maintains a page table of each job (in the job's PCB), outlining where the pages may be found (figure 5.9 illustrates).

In general, if page size is P, then a logical address, L, produces a page number, p, and offset/displacement, d, given by

```
p = L div P
d = L mod P
```

5.7.1 Overview of Paging (continued)

Figure 5.9a: Illustrating Paging

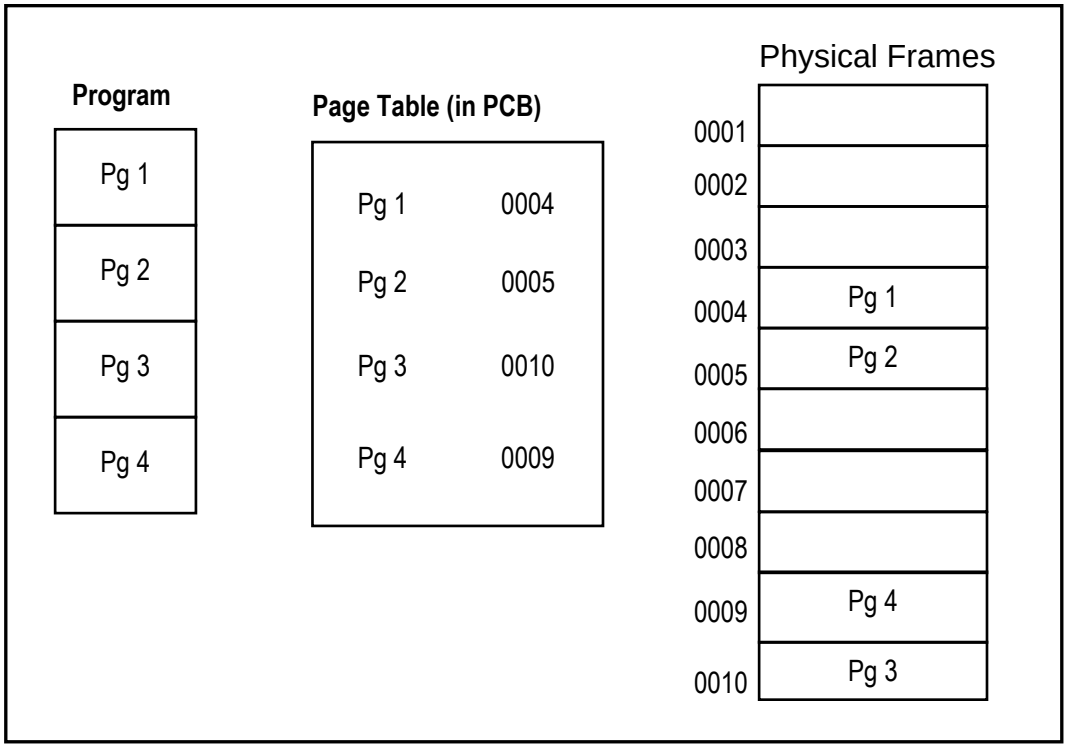
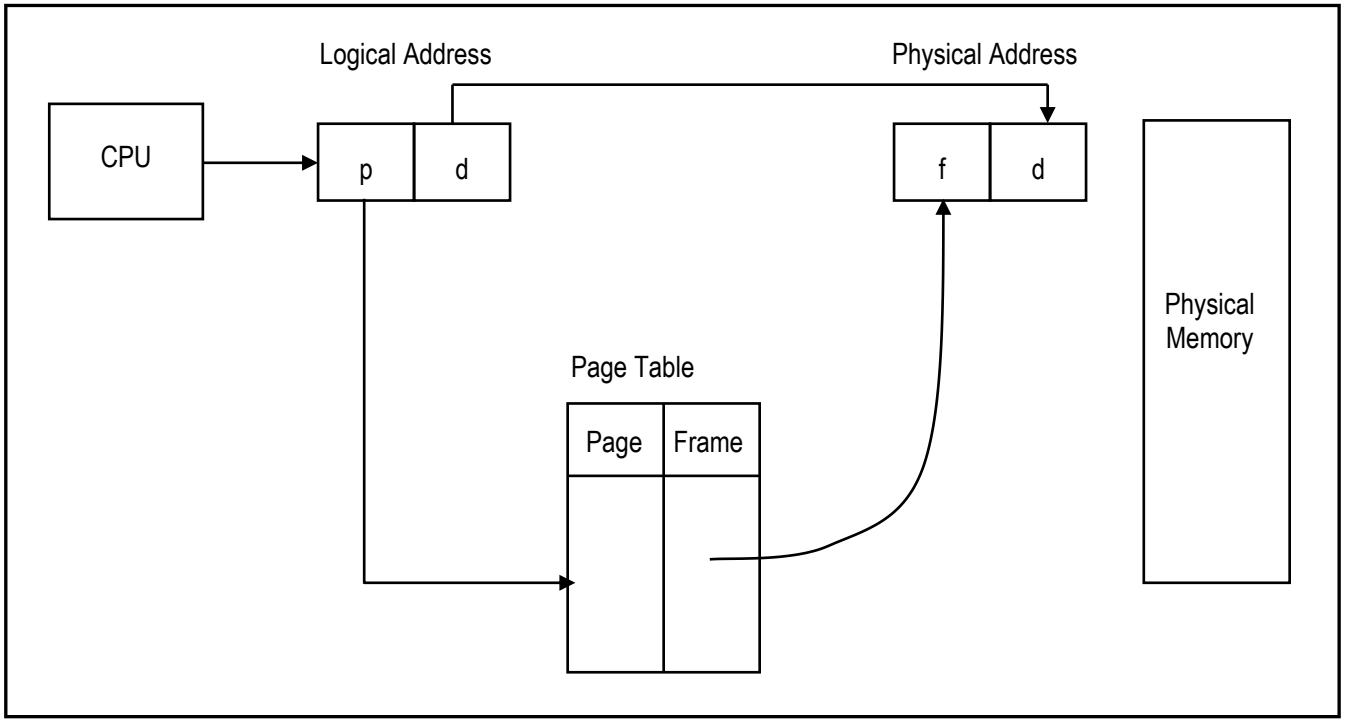


Figure 5.9b: Illustrating the Paging Hardware



5.7.1 Overview of Paging (continued)

Example 2:

If there are 4 pages 8 bytes each, i.e. P = 8, then
Address space is 32 bytes (i.e. 8 * 4)

Logical Address: 0, 1, ... 31

4 pages may be represented by 2 bits; 8 bytes may be represented by 3 bits; we therefore have a 5-bit address as follows:

Logical (L)	Page Number (p)	Displacement (d)
00	00	000
01	00	001
02	00	010
03	00	011
04	00	100
05	00	101
06	00	110
07	00	111
08	01	000
09	01	001
…		
15	01	111
16	10	000
17	10	001
…		
23	10	111
24	11	000
25	11	001
…		
31	11	111

The displacement cycle repeats within each page.

Example 3:

If a computer is referred to by an 8 bit page, and an 8 bit offset, then address space is as follows:

8-bit page $\Rightarrow$ 256 pages
8-bit offset $\Rightarrow$ 256 bytes per page

Total memory space = (256 * 256) bytes

5.7.1 Overview of Paging (continued)

Please note, the logical address space needn't match the physical address space. For instance, an n-bit number can be mapped to a (n+2) bit frame number. Here, the memory space available is 4 times more than any user can access. Thus at least 4 users can comfortably reside in memory simultaneously. Multiprocessing ensures that all memory is used.

Job scheduling is done in the following way: When a job arrives to be executed, the scheduler examines its size, and determines whether enough frames exist to hold the pages of the job. If enough frames exist, the job is loaded and the page table updated.

Paging eliminates external fragmentation (except for jobs that are too large to be loaded) and reduces internal fragmentation to less than a page.

5.7.2 Page Table

Each job has its own page table which is stored in the PCB. When the job is swapped out, the page table is detached from the PCB; when the job is swapped in, the page table is reconstructed. We therefore have a sort of dynamic relocation.

Where will the page table be stored? The page table must be stored so that address translation is as fast as possible since CPU speed depends on address translation speed. We will examine three alternatives:

Option 1: Store Page Table in a set of Fast Registers

This strategy works well for small and medium size systems but is impractical for large systems. Typically from 8 to 256 registers would be needed for the page table. In a multiprogramming environment, this was far too expensive.

Example 4:

The approach was used on the following systems:

- XDS-940 which had 8 pages of 2048 words each and 8 page table registers
- Nova 3/D had 32 pages of 1024 words each with 32 page table registers
- The Sigma 7 had an 8-bit page number. requiring 256 page table registers

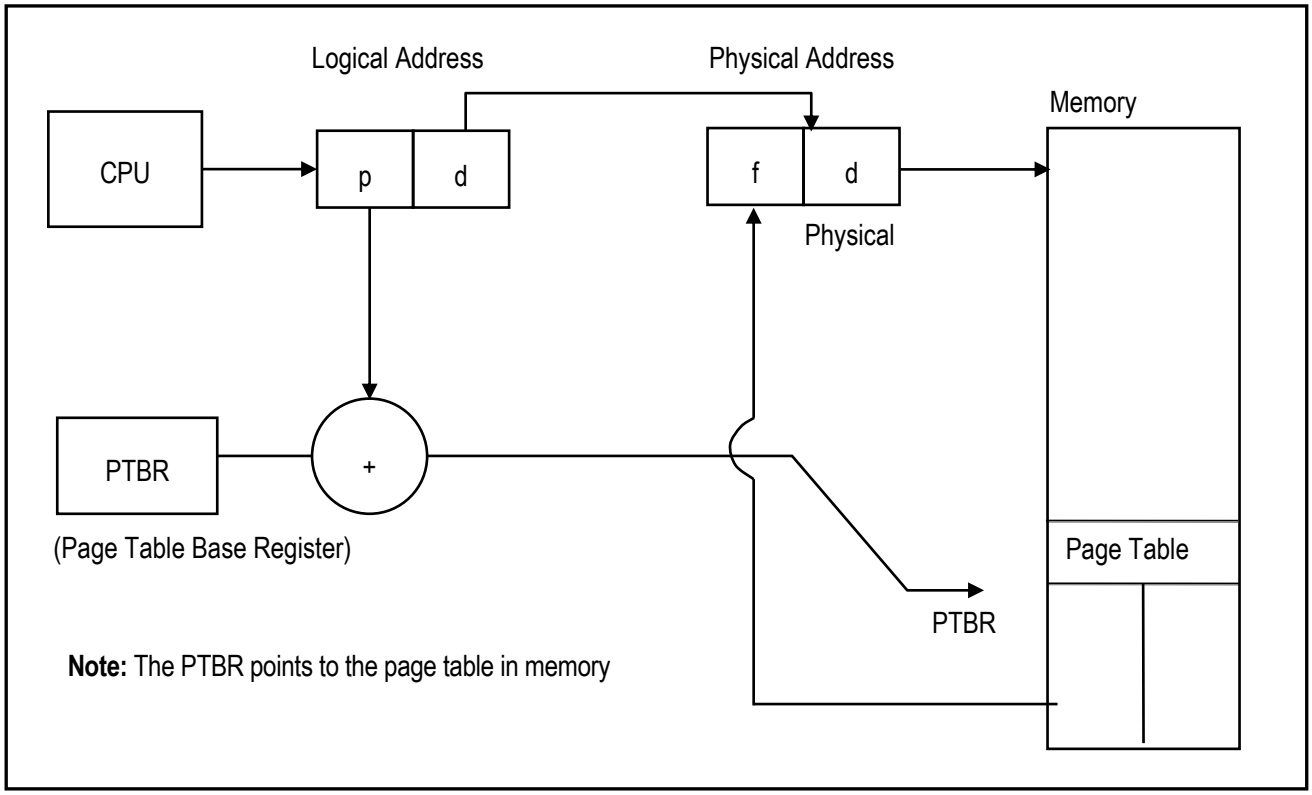
The DEC-10 had 512 pages, IBM 370 had 4096 pages (12-bit page and 12- bit offset) Multics had 16,777,216 pages. For these, fast registers for the page table was impractical.

Option 2: Allocating a Section of Memory for the Page Table

This approach was implemented for DEC-10, Multics, and IBM 370. It suffered from one major setback: Reference is made to memory twice for each instruction (see figure 5.10). This slows down processing speed significantly.

5.7.2 Page Table (continued)

Figure 5.10: Illustrating the use of Memory to store the Page Table



Option 3: Put Some Page Table Entries in Registers and Others in Memory

This approach is based on the *locality principle*, which observes that a program tends to refer to a local area in memory at a time. The pages being referenced are kept in fast registers; others in memory.

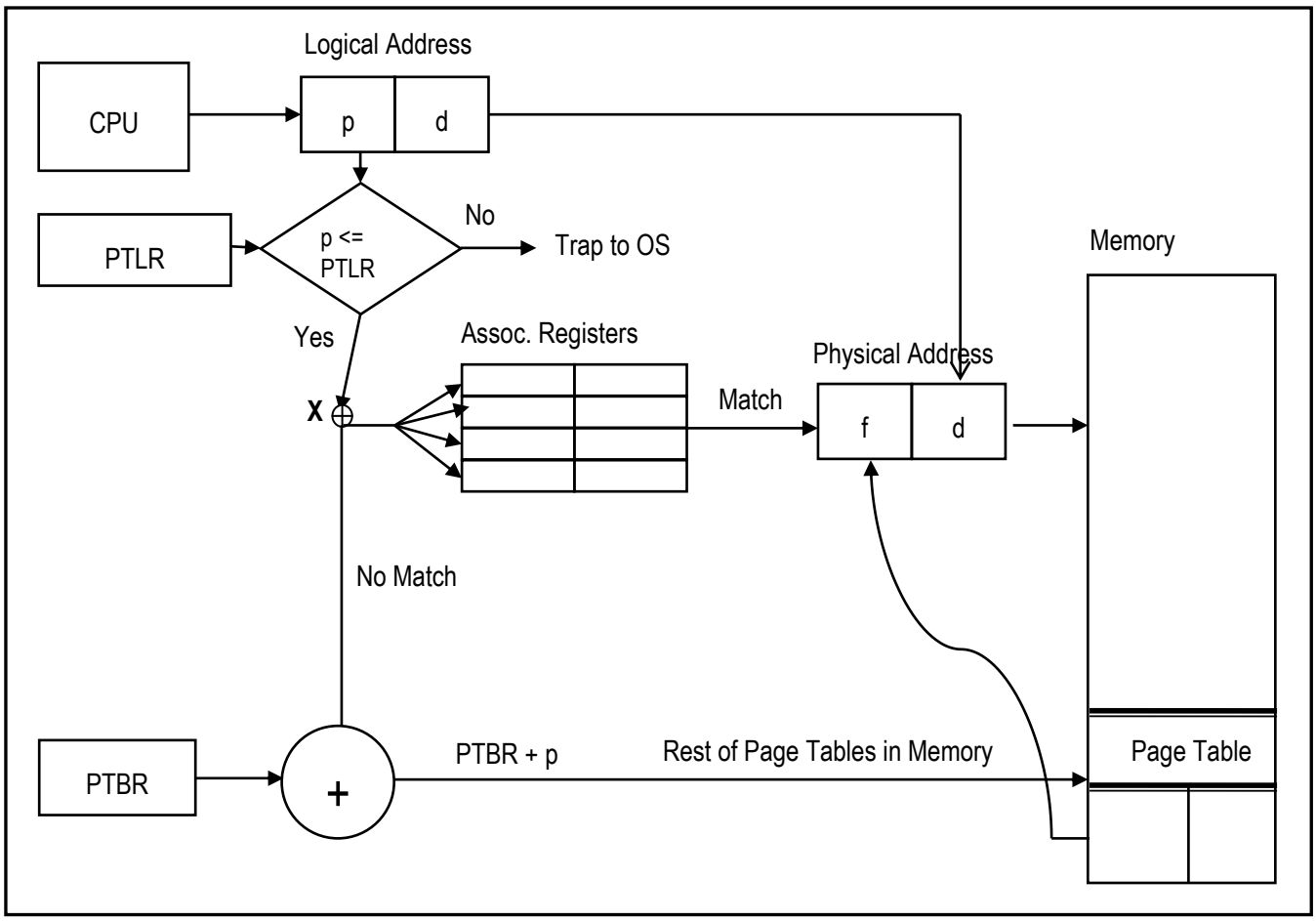
For each logical address, the OS first checks if the address is in the fast associative registers. If it's not there, memory is referenced. [Search of items in the associative registers is simultaneous — the address is checked against all entries simultaneously].

Eventually, frequently referenced pages will be in the fast registers (associative registers). Cache memory may be used instead of, or with associative registers, to further boost performance. In modern systems, this is usually done.

Figure 5.11 illustrates how using associative registers helps with memory management. At point X on the figure, the associative registers are checked for a matching page. If found, the page's frame number is retrieved, and recombined with the displacement to yield the desired physical address. If a match is not found in the associative registers, then the rest of the page that resides in memory, is accessed to obtain the desired frame number for that page. The page table is found by combining the contents of the page table base register (PTBR) with the desired page number.

5.7.2 Page Table (continued)

Figure 5.11: Illustrating use of Associative Registers for the Page Table



The *hit ratio* (*i*.e. the number of times a match is found in the associative registers) depends on the amount of registers used. With 8 – 16 registers, a hit ratio of 80% – 90% can be obtained.

With this arrangement, the access time of the system is significantly improved. To illustrate, suppose that it takes 50 ns to access an associative register, and 750 ns to access memory. Then it takes 800 ns to access an instruction whose page is in an associative register, and 1550 ns to access an instruction whose page is not in an associative register (since we must first check the associated registers).

The *hit ration* is the percentage of times that a required page is likely to be found in an associative register. A hit ratio above 75% is considered as good. Referring to the previously mentioned illustration, for 80% hit ratio, *mean access time = .80 (800) + .20 (1550) = 950 ns.*

There is no way of knowing how many associative registers to use. Usually, a simulation is done and the most suitable number is chosen.

5.7.2 Page Table (continued)

How are associative registers filled? This is a difficult question to which different answers apply. One approach is as follows:

- Place an arbitrary number of frames (pages) in the associative registers. Count occurrences of references to pages in memory.

> (If reference to page in memory) > (Some base reference)
> then replace it with one of the associative register pages.

- Strategies for page replacement include:
  - Oldest entry (FIFO)
  - Least Frequently Used (LFU)
  - Least Recently Used (LRU)

  FIFO is hardly used because it seems obviously unreasonable. LFU suffers from the problem that some pages could be frequently used at the start of the program and thereby build up a high score. These pages will qualify to stay even though they may not be currently required.
  For LRU, some timer is applied to pages in the associated registers — each entry has a time stamp. The entry with the smallest time is swapped out.

5.7.3 Address Checking and Cross Linking with Programs

How does the operating system ensure that a program does not access an address not in its page table?

- A *page table length register* (PTLR) is used to contain the number of the highest page of the program.
- Each page number is checked against the contents of the PTLR. If the requested page is greater, a trap to the operating system is made (review figure 5.11).

Pages can have read/write (R/W) protection flags. Typically, a single bit is used for this. Finally, to link programs and therefore reference pages outside of the page table, a validity bit is added to the page table so that "foreign" pages may be referenced. A summary of the entries is shown figure 5.12.

Figure 5.12: Revised Page-table Entries

Page	Frame	Validity Bit	R/W

5.7.4 Benefits and Challenges of Paging

Paging introduces a number of benefits to memory management:
- Significant reduction of external fragmentation.
- Reduction of internal fragmentation to bytes within a frame.
- Increased multiprogramming.
- It is not necessary for users to estimate size of jobs.
- It is possible to have reentrant code (*i*.e. code where multiple users share the same code, but different memory locations are updated with user data e.g. a word processor).

But like most good things, paging also introduces additional overheads for the operating system. The two main ones are:
- Address translation;
- Maintenance of the page table within the PCB.

When the benefits are considered, these challenges are a small price to pay.

5.8 Segmentation

Perhaps the best way to introduce segmentation is to note its difference from paging: In paging, the program is split into pages of equal size based on the physical frame size. In segmentation, the program is split into logical segments — procedures/functions are held together; sets of data (e.g. arrays and linked lists) may appear as segments. Segmentation occurs at compilation time. The segments are of variable size.

5.8.1 Overview of Segmentation

Program segments are put in memory gaps large enough to hold them. The concept is similar to MVT, but in MVT the program is held together whereas here, it is split into multiple segments.

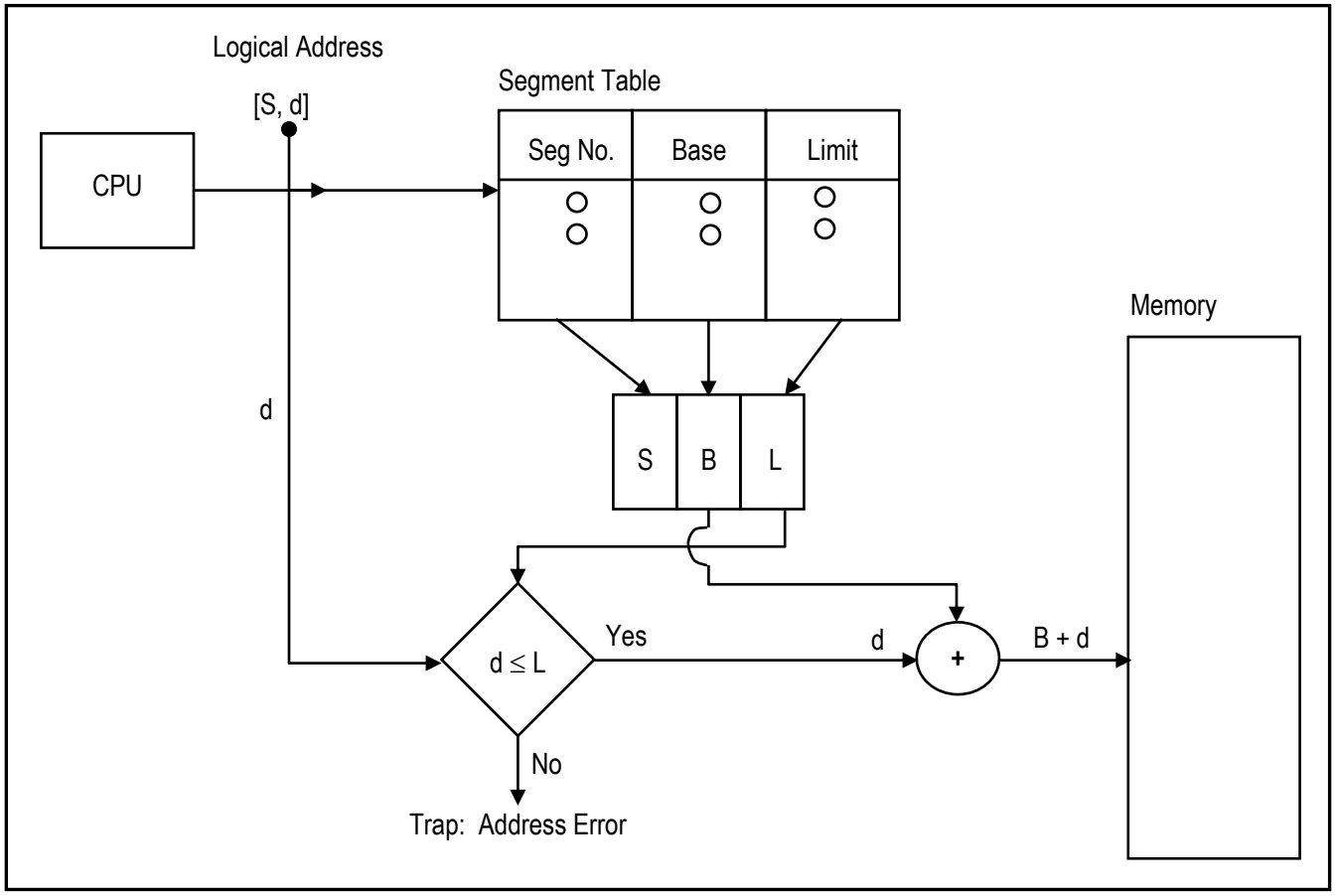
Each segment has a length. An address is specified by the segment-name and an offset within the segment. For simplicity, segments are numbered rather than named. An address is therefore specified by segment number and offset.

The segment table provides the mapping between (logical) segments and physical memory. A logical address consists of a segment, S and a displacement (offset) d. We can therefore represent an address as [S, d].

Each segment of the segment table has a base and a limit: [B, L]. Each offset, d, must lie between 0 and the segment limit, L. Thus, *Physical Address* = B + d, as illustrated in figure 5.13.

5.8.1 Overview of Segmentation (continued)

Figure 5.13: Illustrating Segmentation



5.8.2 Storing the Segment Table

The alternatives and consequences for storing the segment table are similar to those for paging:

- Use of registers
- Use main memory — segment table base register (STBR) instead of PTBR and a segment table length register (STLR) instead of a page table length register (PTLR)
- Use of associative registers
- Could use cache to improve performance

Example 5:

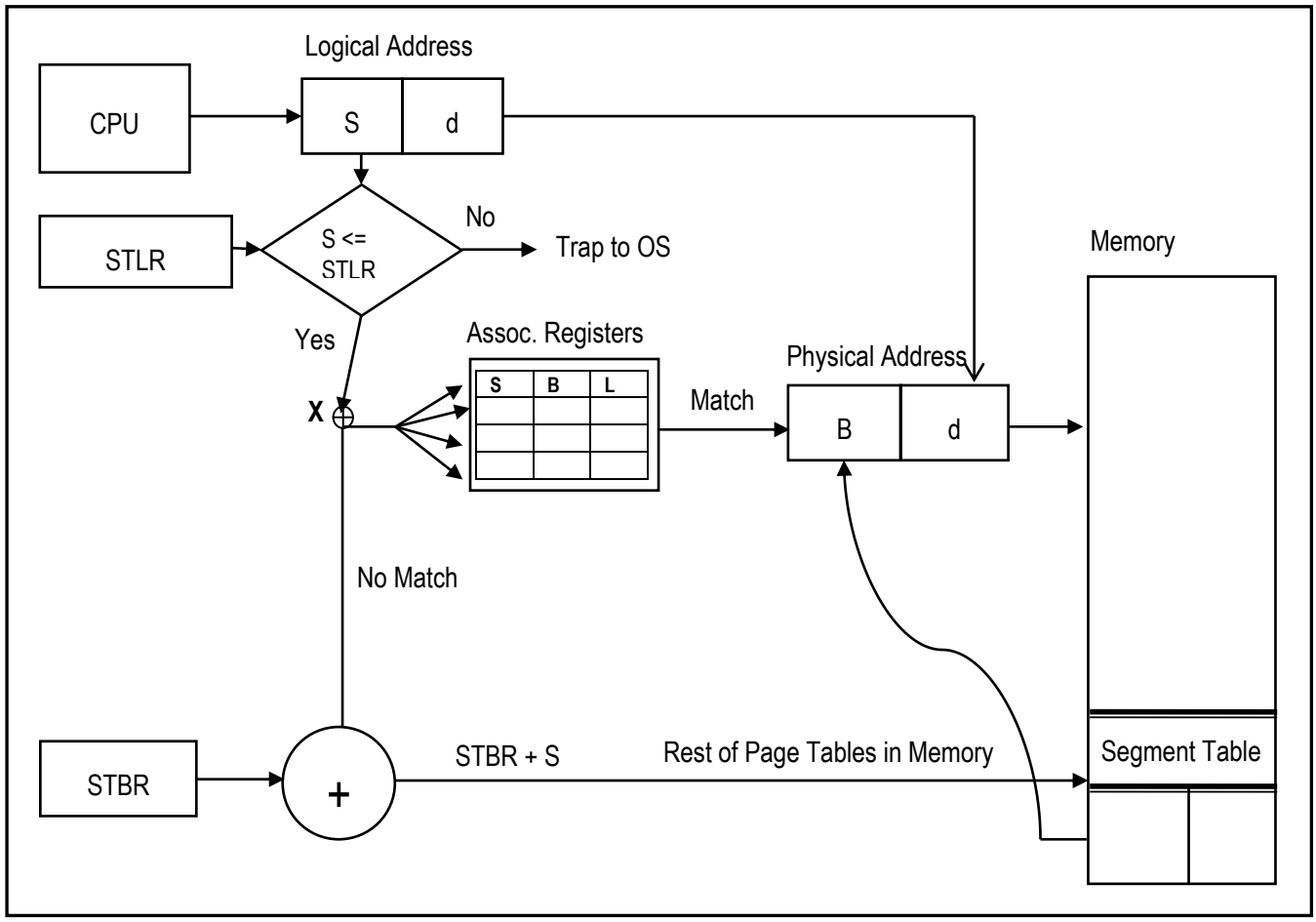
The PDP-11/45 by DEC used 8 segment registers. A 16-bit address is formed from a 3-bit segment number and a 13-bit offset. With this arrangement, each segment could be up to 2^{13} bytes i.e. 8 KB.

The GE 645 used for Multics allowed up to 256 K segments of up to 64 K words each. For this the segment table was stored in memory.

5.8.2 Storing the Segment Table (continued)

Figure 5.14 is similar to figure 5.11 except that the one case applies to paging and the other to segmentation. When a logical address is encountered, the associative registers are checked for a match. If a match is found, the physical address is obtained by combining the segment's base address with the displacement. If a match is not found, then the STBR is combined with the segment number to yield the address of the segment table, which is then accessed for the base address; this base address is combined with the displacement to yield the required physical address.

Figure 5.14: Illustrating use of Associative Registers for the Segment Table



As an exercise, attempt to illustrate diagrammatically, segmentation with segment table stored exclusively in memory (use figure 5.10 as a guide).

5.8.3 Segment Size, Protection and Cross Linking Programs

If the whole program is made a segment, the system reduces to MVT. If on the other hand, segments are too small, the segment table lengthens and access slows down. The segment size must therefore be carefully chosen. This can be challenging.

The ratio (memory used for segment) / (memory used for segment table) must not be too low since this results in inefficiency.

Address checking and protection is done in a manner similar to paging:
- A segment table length register (STLR) is used to contain the number of the highest segment of the program.
- Each segment number is checked against the STLR. If Segment# is greater than the value in the STLR, a trap to the operating system is made.

Segments can have read/write (R/W) protection flags. Typically, a single bit is used for this. Finally, as for paging, reference can be made to segments outside of the segment table, by making use of the validity bit. A summary of the entries is shown in figure 5.15.

Figure 5.15: Revised Segment-table Entries

Segment	[Base, Length]	Validity Bit	R/W

5.8.4 Benefits and Challenges of Segmentation

Segmentation has the following benefits:
- Reduction of external fragmentation
- Reduction of internal fragmentation to negligible amounts (can be considered eliminated)
- Use of the locality principle can result in improved performance in the OS
- Facilitation of reentrant code
- Increased multiprogramming
- It is not necessary for users to estimate the size of jobs

Overheads of segmentation include:
- Address translation
- Maintenance of segment table within PCB
- External fragmentation as in MVT
- Determining an appropriate segment size

5.9 Segmented Paging

The idea of segmented paging is to segment the page table (i.e. split page table into segments) and thus avoid the overhead of space wasting in the page table. Effectively, pages are put into segments.

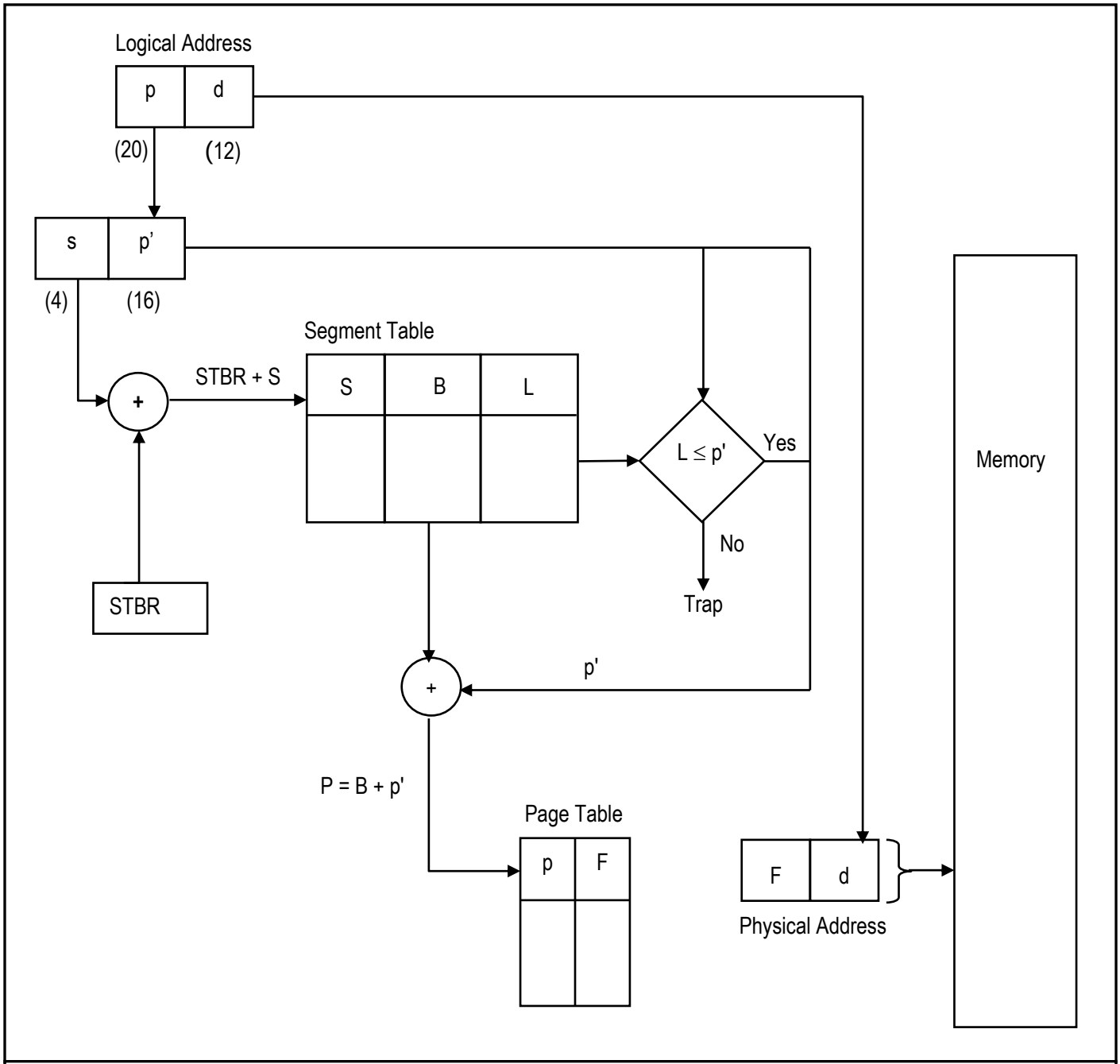
The objective is to retain the benefits of either approach, while minimizing the drawbacks (primarily of segmentation).

Example 6 — the IBM 370 (earlier IBM 360/67):

- The early system had a 24-bit address field
  - 12-bit page & 12-bit offset
  - This allowed for 4096 pages
  - Each page table entry spanned two bytes (12-bits plus validity bit);
  - Each page table occupied 4096 * 2 bytes i.e. 8 KB
- It was desired to expand the logical address to 32 bits — 20-bit page and 12-bit offset. This required a page table of 1,048,576 entries, or 2,097,152 bytes i.e. 2 MB.
- Not all programs would require so much space (consider a 2 KB file with a 2 MB page table).
- To circumvent this, the page table was segmented — upper 4 bits used as segment number to select one of 16 segment table entries.
- Each segment could be up to 268,435,456 bytes in length i.e. segment has 2^{16} pages each of 4096 bytes. Figure 5.16 illustrates.

5.9 Segmented Paging (continued)

Figure 5.16: Illustrating Segmented Paging (on the IBM 370)



Note:

1. For small programs the high order bits are all zero and only one (of four) segments is used.
2. Each segment points to page table entries.
3. The page table points to frames of memory.
4. Associative registers could be used to speed up processing.

5.10 Paged Segmentation

In paged segmentation, each segment has its own page table i.e. each segment is paged, thus eliminating external fragmentation.

Segment table entries contain the base address of a page table for this segment. The segment offset is broken into a page number and a page offset.

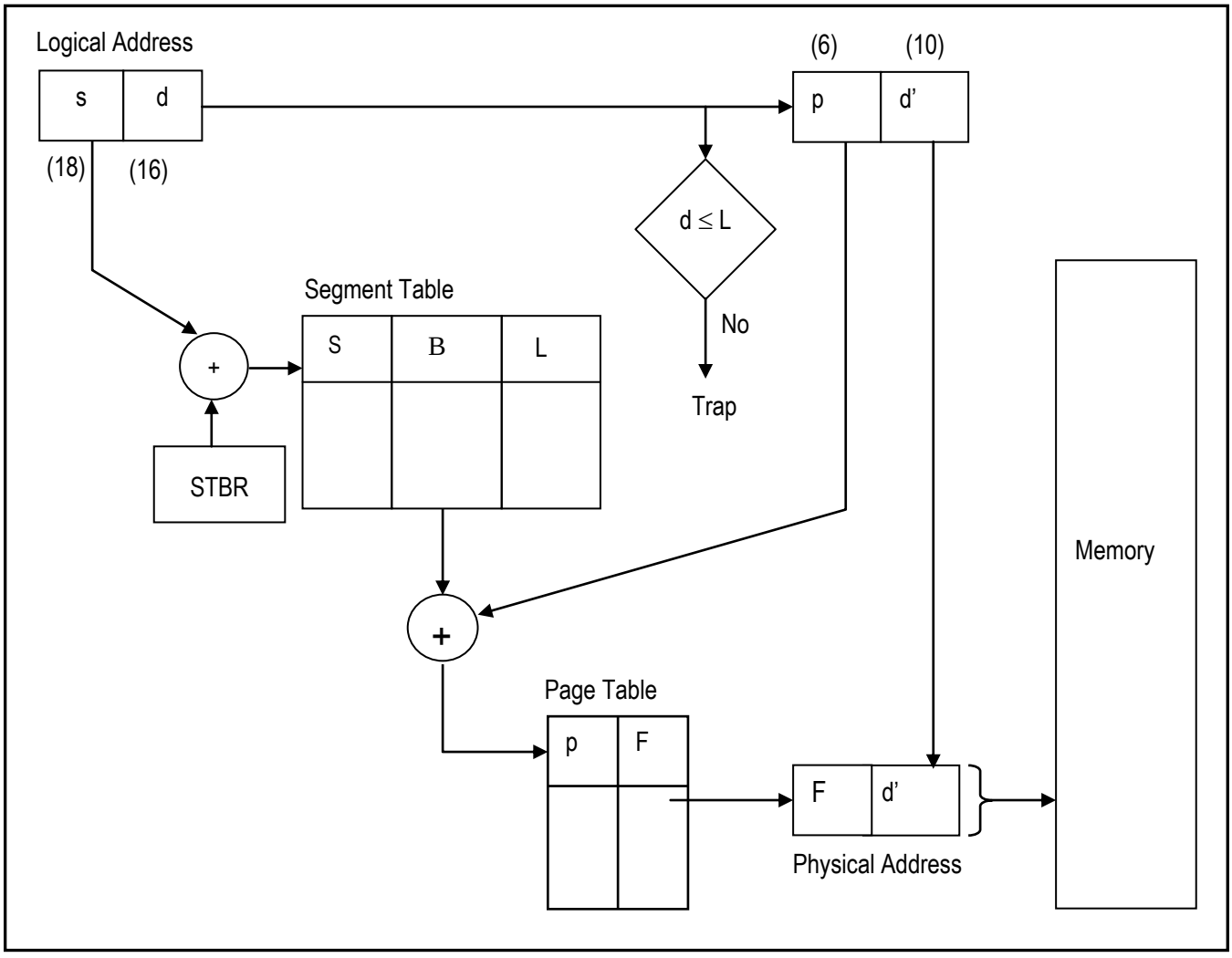
Example 7 — The Multics (GE 645) System:

Had a 18-bit segment and a 16-bit segment offset.

Thus each segment could be up to 64 K words. External fragmentation and long search time were two nagging problems.

The solution was paged segmentation as illustrated in figure 5.17.

Figure 5.17: Illustrating Paged Segmentation (on the Multics System)



5.10 Paged Segmentation (continued)

Note:

1. With this particular implementation, each segment has 64 pages each of 1K words. There can be 262,144 segments.
2. External fragmentation is eliminated, but internal fragmentation may occur on the last page of each program.
3. A more complex algorithm is required to take care of address translation.
4. Associated registers may be used.

5.11 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- Early computer systems provided negligible memory management services. In order to use the system, one had to be technically inclined to interfacing with the bare machine.
- Next came the concept of the resident monitor. The system memory was divided into two partitions — a resident monitor area where the operating system would run, and a user area for application programs. The partitioning was implemented by a fixed fence register.
- Then came the concept of relocation, which entertained the idea of variable fence registers.
- Memory management by partitioning was further improved to MFT and MVT. In MFT, memory was partitioned into a fixed number of partitions. Jobs were allocated primarily via best-fit and first-fit strategies. MVT was an improvement over traditional techniques, but was plagued by internal and external fragmentation.
- In MVT, memory was partitioned into a variable number of partitions. Jobs were allocated via best-fit, first-fit, and worst-fit strategies. MVT led to a significant reduction of internal fragmentation to insignificant levels. However, external fragmentation was still a problem.
- One significant problem with the early memory management strategies was fragmentation, which led to poor operating system performance. Another problem was limitation in the level of multiprogramming that could be provided. This was primarily due to the fact that an executing job had to occupy contiguous memory locations, and the entire job had to be loaded.
- Then came the introduction of paging. In paging, memory is partitioned in page frames (each typically a multiple of 512 bytes). The program to be executed is broken up into pages, each page being the size of a page frame. Pages are loaded into available memory frames (which do not need to be contiguous), and a page table is maintained. Paging significantly reduced external fragmentation, and virtually eliminated internal fragmentation (only the last page of an executing job was subject to internal fragmentation).
- The main challenge associated with paging is where to store the page table. Three strategies have been explored: storing the page table in registers only; storing it in memory; and having a compromise that involves the use of associative registers for currently required pages, and memory for the remaining portion of the page table. The latter approach has led to the development of cache.
- Segmentation was introduced as an alternative to paging; the intent was to reduce the level of external fragmentation. In segmentation, the program is split into variable-length components and loaded into available memory segments. A segment table is used to keep track of the loaded program segments.

5.11 Summary and Concluding Remarks (continued)

- As in paging, the main challenge was where to store the segment table. The approaches employed were similar to those of paging. Additionally, segmentation turned out to be a bit more challenging to program, since the operating had to deal with variable-length segments.
- Two hybrid strategies that were explored were paged segmentation, and segmented paging. In paged segmentation, each segment has its own page table, thus avoiding a wasteful oversized segment table. In segmented paging, the page table is segmented to avoid dealing with a wasteful oversized page table.

Each of these historical strategies was useful in contributing to the development of contemporary memory management strategies. However, one problem that persisted in all of these strategies is that an entire program had to be in memory in order for it to run. Contemporary memory management strategies have been able to solve this problem. These strategies all come under the umbrella of *virtual memory*, which is the focus of the next lecture.

5.12 References and/or Recommended Readings

[Bacon & Harris 2003] Bacon, Jean & Tim Harris. 2003. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley. See Chapter 5.

[Nutt 2004] Nutt, Gary. 2004. *Operating Systems: A Modern Perspective* 3rd ed. Boston: Addison-Wesley. See Chapter 11.

[Silberschatz 2012] Silberschatz, Abraham, Peter B. Galvin, & Greg Gagne. 2012. *Operating Systems Concepts*, 9th Ed. Update. New York: John Wiley & Sons. See Chapter 8.

[Stallings 2005] Stallings, William. 2005. *Operating Systems* 5th ed. Upper Saddle River, New Jersey: Prentice Hall. See Chapter 7.

[Tanenbaum & Wodhull 1997] Tanenbaum, Andrew S., & Albert S. Woodhull. 1997. *Operating Systems: Design and Implementation* 2nd ed. Upper Saddle River, New Jersey: Prentice Hall. See Chapter 4.