OPERATING SYSTEMS MODULE 4

SHAMEELA SULAIMAN
COLLEGE OF ENGINEERING, ADOOR

SYLLABUS

- Memory Management:
    - Concept of address spaces
    - Swapping
    - Contiguous memory allocation,
        - fixed and variable partitions,
    - Segmentation,
    - Paging.
    - Virtual memory,
    - Demand paging,
    - Page replacement algorithms.

MEMORY MANAGEMENT

- Memory is central to the operation of a modern computer system.
- Memory consists of a large array of bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.
- A typical instruction-execution cycle, for example, first fetches an instruction from memory.
- The instruction is then decoded and may cause operands to be fetched from memory.
- After the instruction has been executed on the operands, results may be stored back in memory.

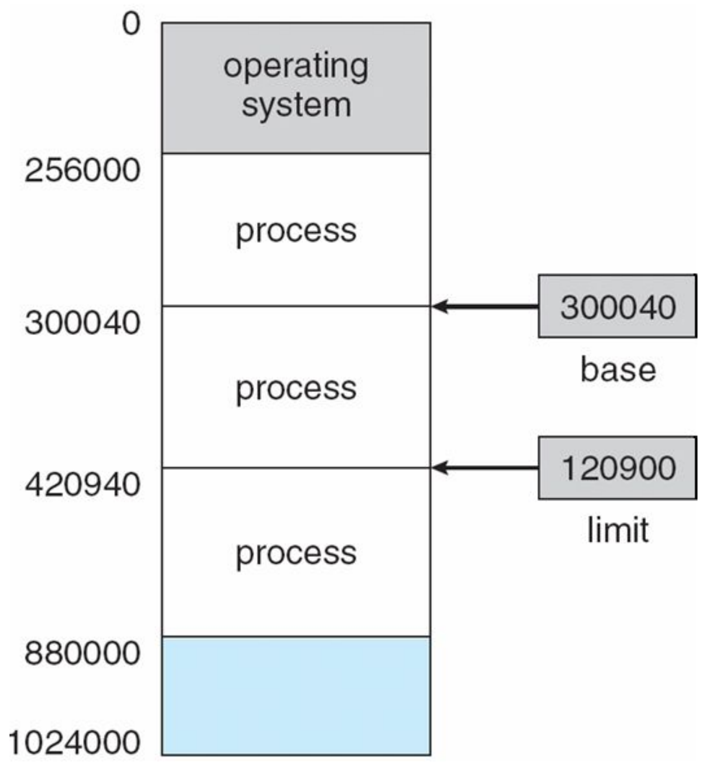
MEMORY MANAGEMENT

- Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.
- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock.

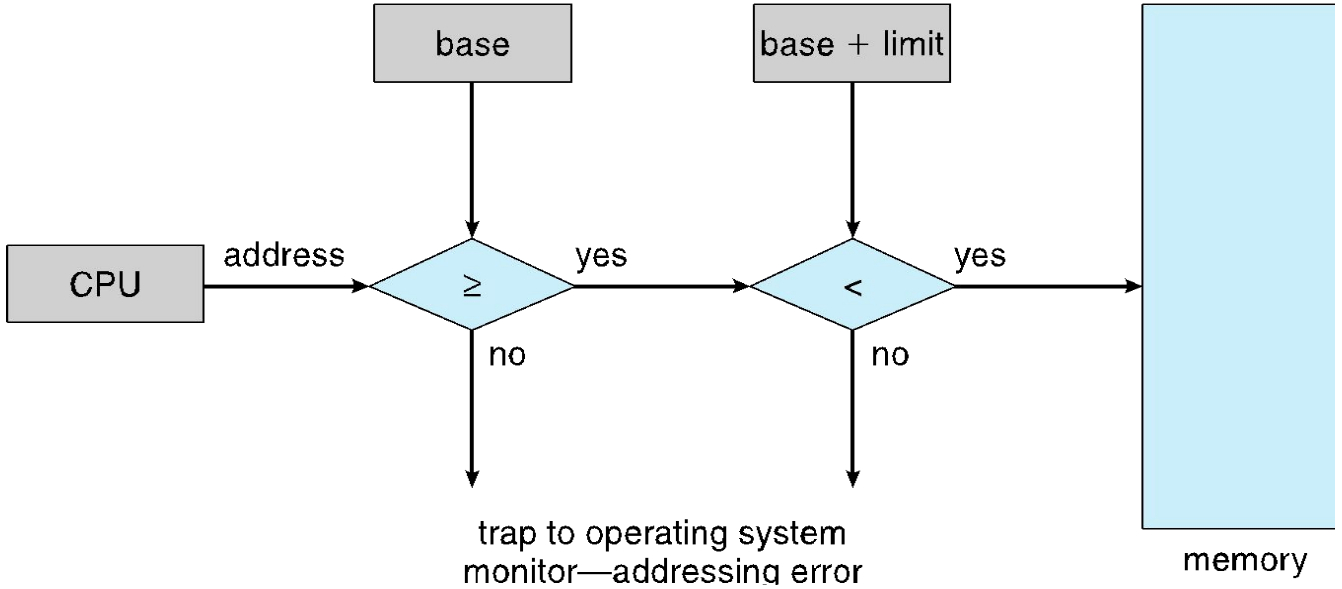
MEMORY MANAGEMENT

- Multiple processes loaded in memory for concurrent execution.
- Each process has a separate memory space.
- Separate per-process memory space protects the processes from each other
- We can provide this protection by using two registers,
- The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.
-

Base & Limit Registers



H/W address protection



ADDRESS BINDING

- Program resides on a disk as a binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- The processes on the disk that are waiting to be brought into memory for execution form the **input queue.**
- Addresses may be represented in different ways during these steps.
- Addresses in the source program are generally symbolic.
- A compiler typically binds these symbolic addresses to relocatable addresses
- The linkage editor or loader in turn binds the relocatable addresses to absolute addresses
- **Each binding is a mapping from one address space to another.**

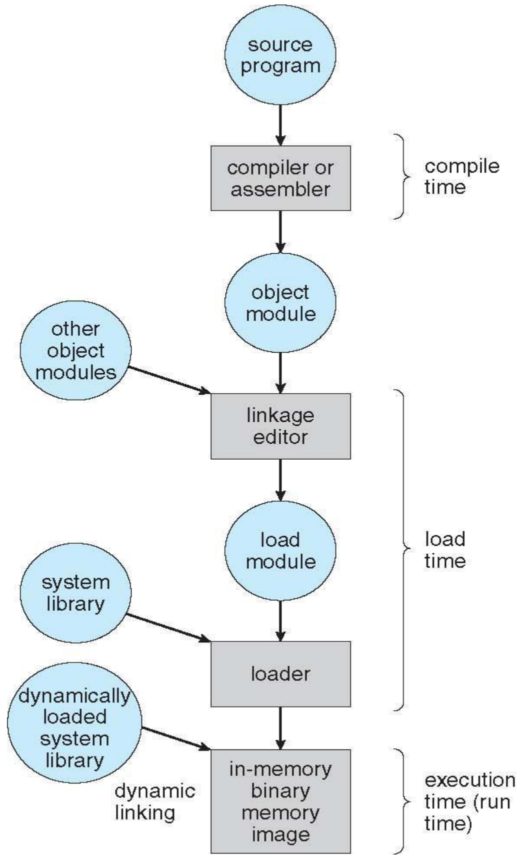
ADDRESS BINDING

- The binding of instructions and data to memory addresses can be done at any step along the way:
- **Compile time**:- If you know at compile time where the process will reside in memory, then absolute code can be generated.
- For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.
-

ADDRESS BINDING

- **Load time.**:- If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
- Need hardware support for address maps.

MULTI LEVEL PROCESSING OF A USER PROGRAM



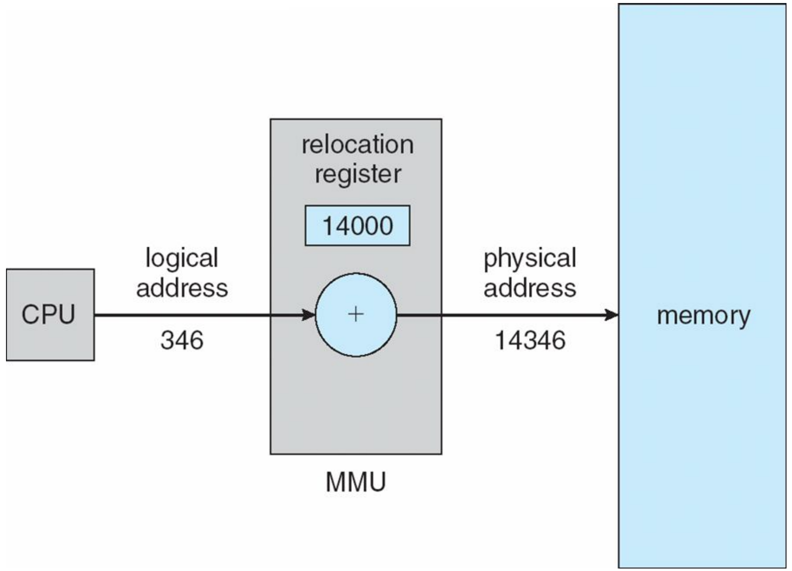
Physical Vs Logical Address Space

- An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit—that is, the one loaded into the memory-address register of the memory—is commonly referred to as a physical address.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- However, the execution-time address binding scheme results in differing logical and physical addresses.
- The set of all logical addresses generated by a program is a logical address space.
- The set of all physical addresses corresponding to these logical addresses is a physical address space.
- In the execution-time address-binding scheme, the logical and physical address spaces differ.

MEMORY MANAGEMENT UNIT MMU

- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).
- The base register is now called a relocation register.
- The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory
- 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.
- The user program never sees the real physical addresses.
- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- However, these logical addresses must be mapped to physical addresses before they are used.

MMU


relocation
register
14000
logical
address
CPU
+
346
physical
address
14346
memory
MMU

Dynamic loading

- With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, 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.
- Then control is passed to the newly loaded routine

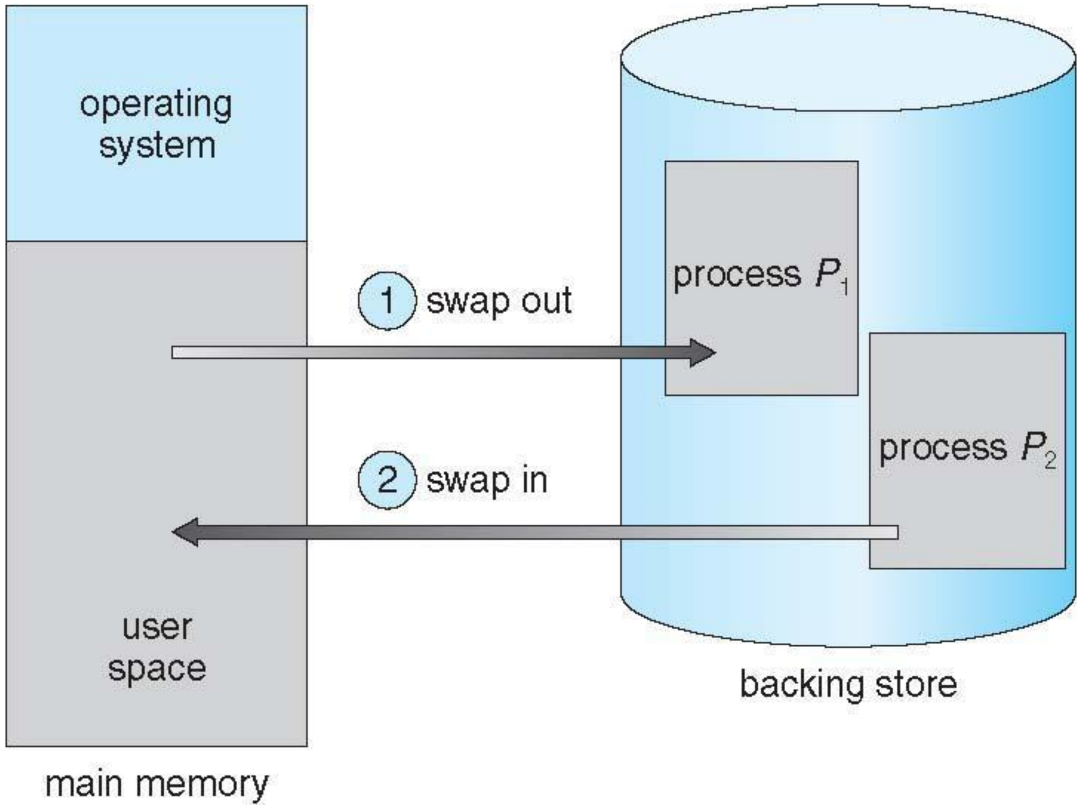
Dynamic Loading

- The advantage of dynamic loading is that a routine is loaded only when it is needed.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
- In this case, although the total program size may be large, the portion that is used may be much smaller.
- Dynamically linked libraries are system libraries that are linked to user programs when the programs are run
- Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.
- Dynamic linking, in contrast, is similar to dynamic loading.
- Here, though, linking, rather than loading, is postponed until execution time.
- This feature is usually used with system libraries, such as language subroutine libraries.

SWAPPING

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
- Total physical memory space of processes can exceed physical memory.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk

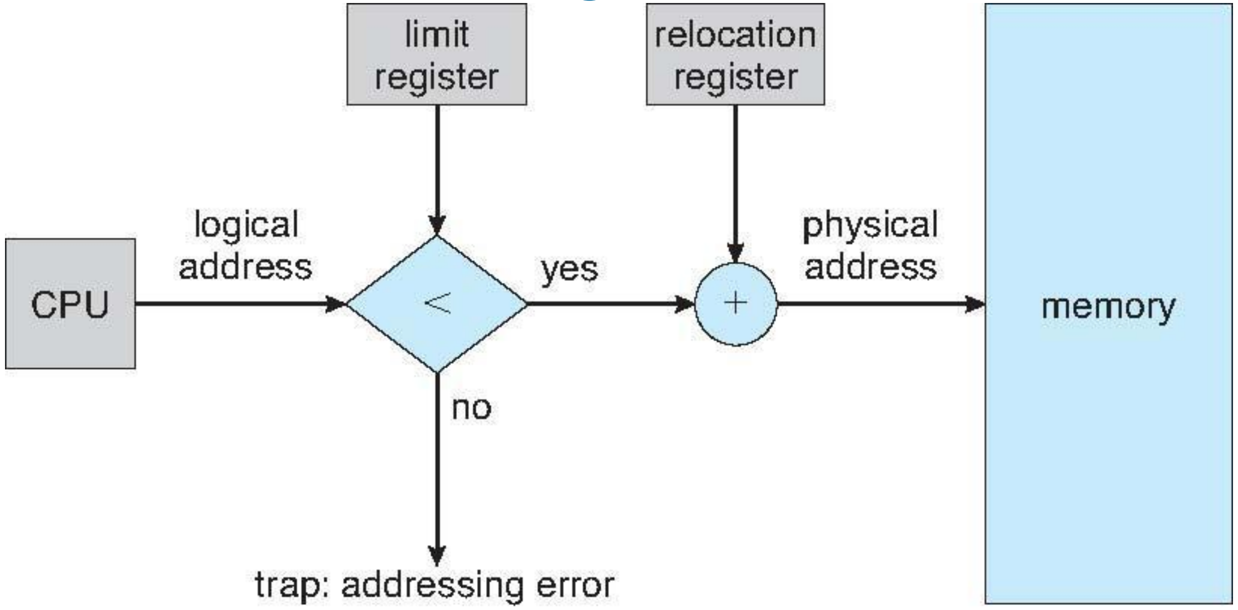
SWAPPING


operating system
1 swap out
process P_1
process P_2
2 swap in
user space
backing store
main memory

Contiguous memory allocation

- The main memory must accommodate both the operating system and the various user processes.
- We need to allocate main memory in the most efficient way possible.
- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.
- We can place the operating system in either low memory or high memory.
- In contiguous memory allocation, each process is contained in a single section of
- memory that is contiguous to the section containing the next process.

Hardware Support for Relocation and Limit Registers



FIXED SIZED PARTITION

- Divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process

VARIABLE PARTITION

- In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
- Memory contains a set of holes of various sizes.

ALLOCATION (FOR FURTHER READING)

- As processes enter the system, they are put into an input queue.
- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time.
- When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- At any given time, then, we have a list of available block sizes and an input queue.
- The operating system can order the input queue according to a scheduling algorithm.
- Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process.
- The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

HOW TO ALLOCATE PROCESS TO A PARTITION

- The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.
- First fit.
  - Allocate the first hole that is big enough.
  - Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.
  - We can stop searching as soon as we find a free hole that is large enough.

HOW TO ALLOCATE PROCESS TO A PARTITION

- Best fit.
  - Allocate the smallest hole that is big enough.
  - We must search the entire list, unless the list is ordered by size.
  - This strategy produces the smallest leftover hole.
- Worst fit.
  - Allocate the largest hole.
  - Again, we must search the entire list, unless it is sorted by size.
  - This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

EXTERNAL FRAGMENTATION

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe.
- In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes

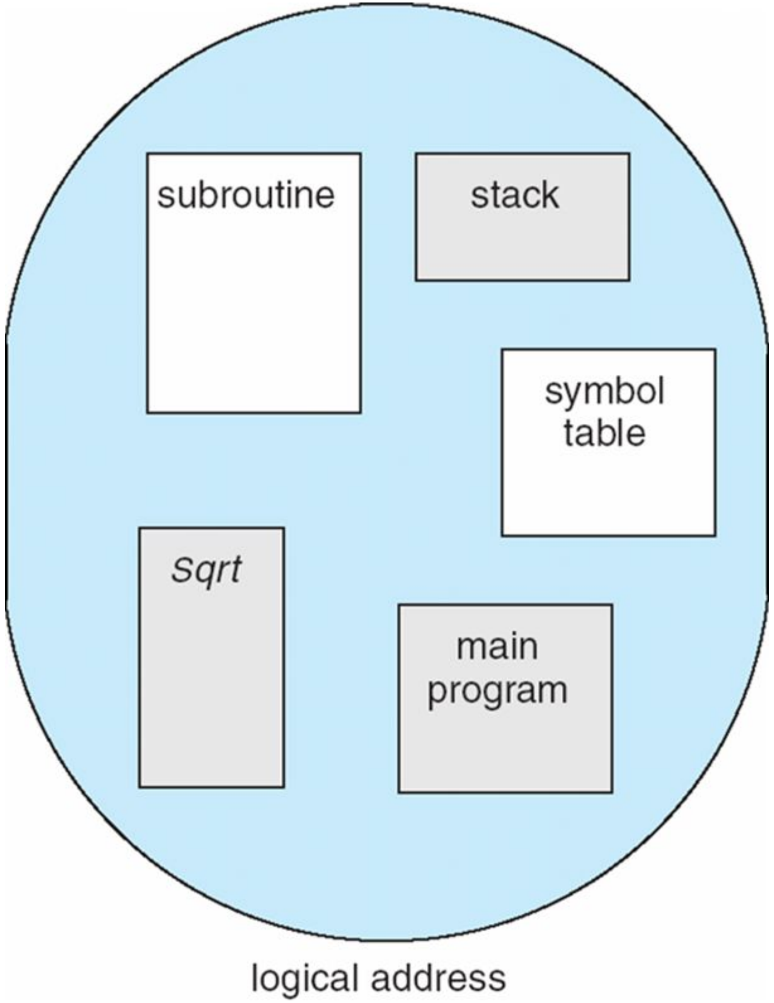
INTERNAL FRAGMENTATION

- If you break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- The difference between these two numbers is internal fragmentation —unused memory that is internal to a partition.

SEGMENTATION

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
- A segment is a logical unit such as:
- procedure main program
- function
- method
- object
- local variables, global variables
- common block
- stack
- symbol table
- arrays

USERS VIEW OF PROGRAM



LOGICAL VIEW OF SEGMENTATION

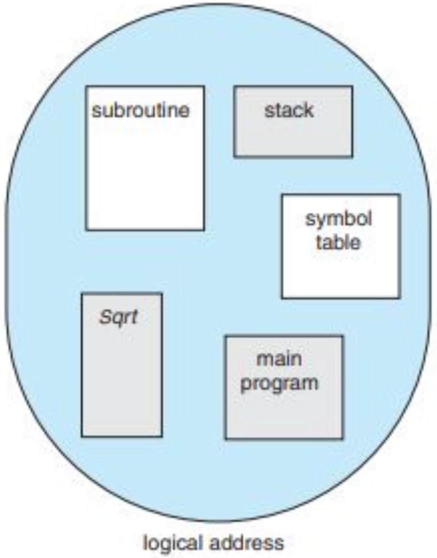


Figure 8.7 Programmer's view of a program.

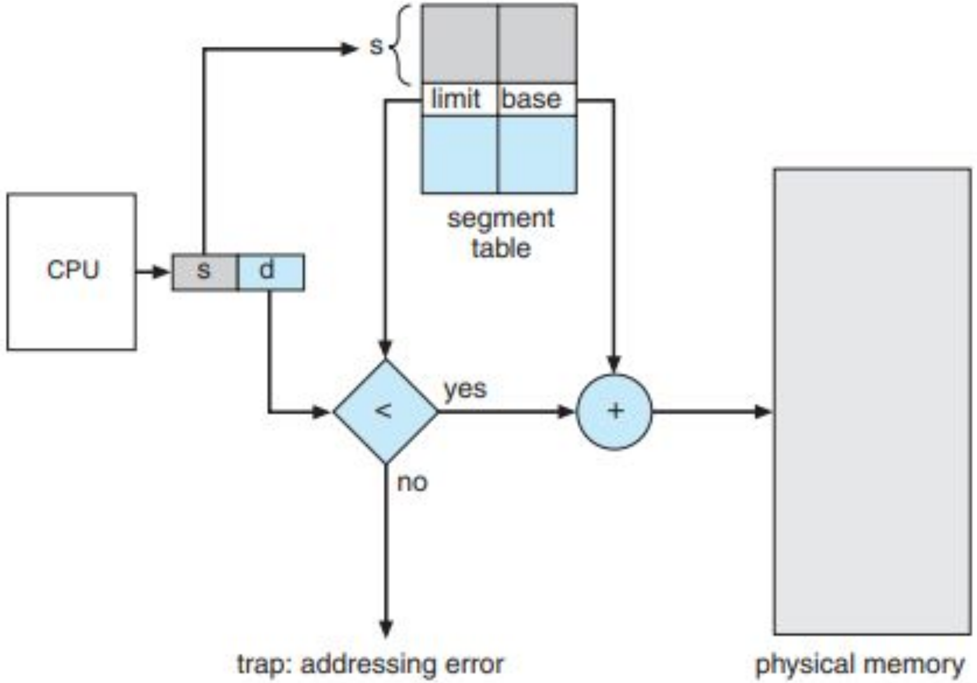


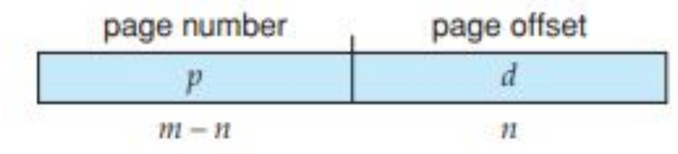
Figure 8.8 Segmentation hardware.

PAGING

- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical address space 2^m and page size 2^n



Paging Hardware

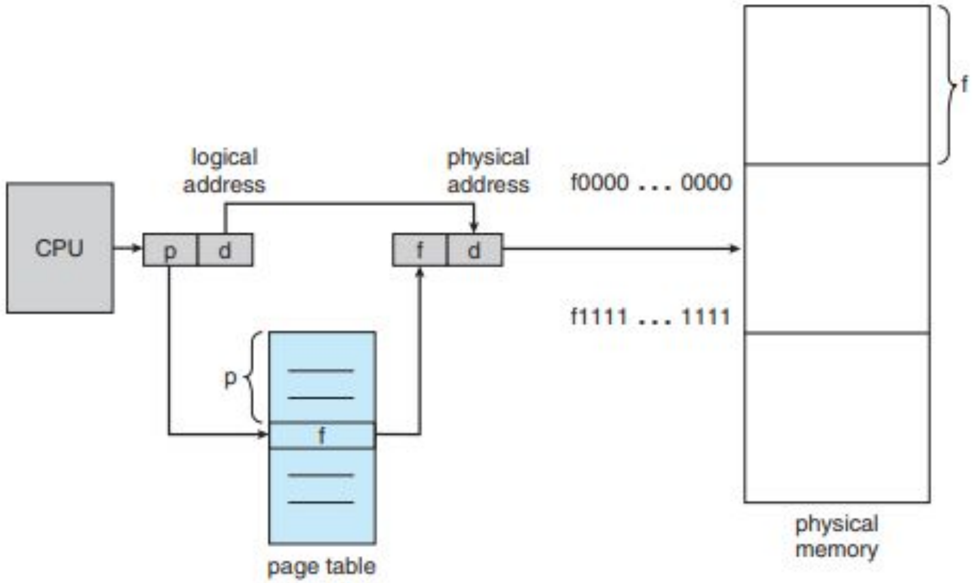


Figure 8.10 Paging hardware.

- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d).
- The page number is used as an index into a 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.
- If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order m – n bits of a logical address designate the page number, and the n low-order bits designate the page offset.

PAGING MODEL OF LOGICAL & PHYSICAL MEMORY

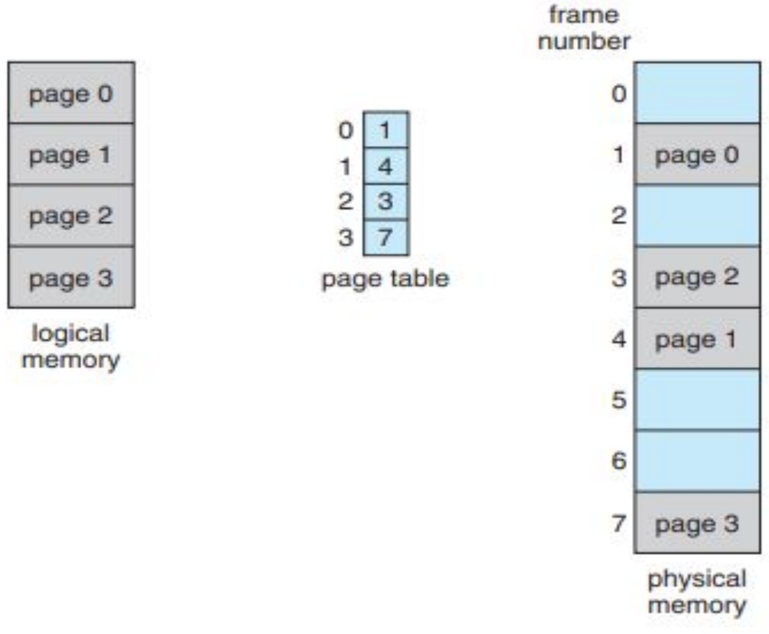


Figure 8.11 Paging model of logical and physical memory.

EXAMPLE

- Consider the memory shown: Here, in the logical address, n= 2 and m = 4.
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory.
- Logical address 0 is page 0, offset 0.
- Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 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 × 4) + 0].
- Logical address 13 maps to physical address 9.

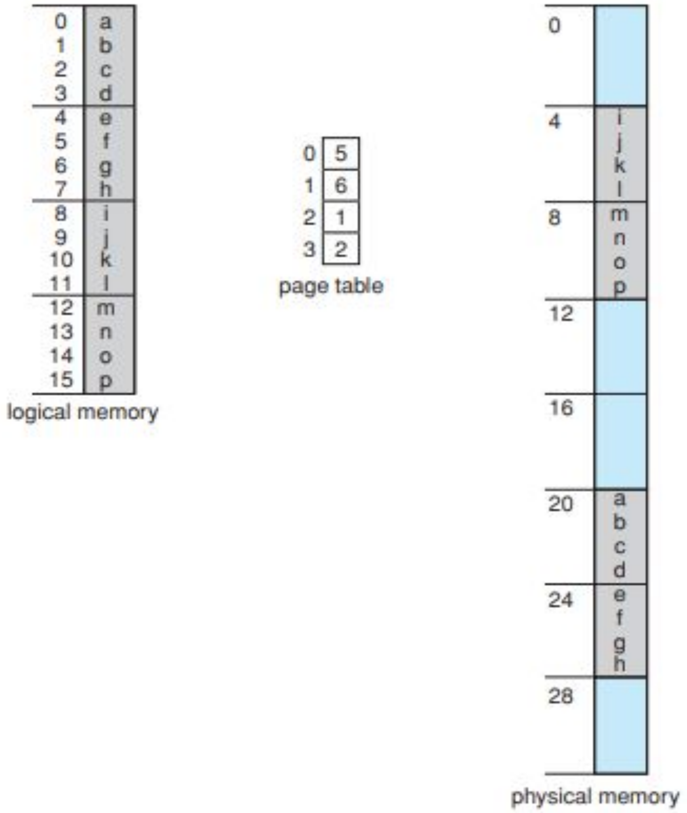


Figure 8.12 Paging example for a 32-byte memory with 4-byte pa

Internal fragmentation

- Calculating internal fragmentation
- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
-