# * Chapter 07 : Virtual Memory Management *

- Background
- Demand paging
- Page replacement

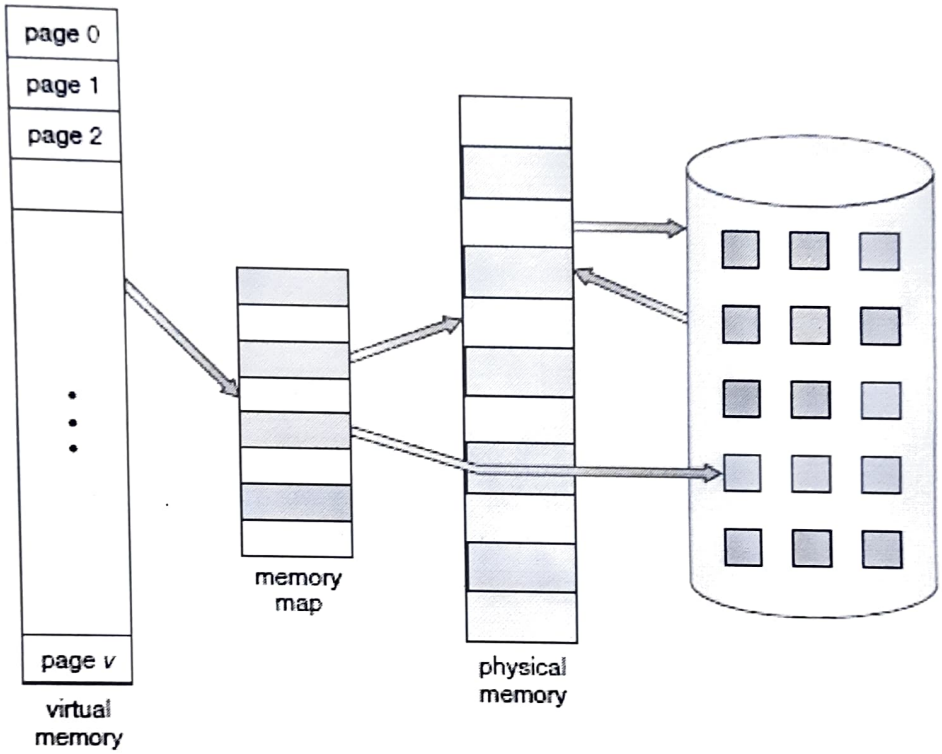
## Background

When a code is executing in memory, the entire program is rarely used. There usually are error codes, unusual routines, large data structures etc. Also, the entire program code is not needed at once.

The ability to execute a program that is only partially in memory would confer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available.
- More programs could be run at same time, with a increase in CPU Utilization & throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

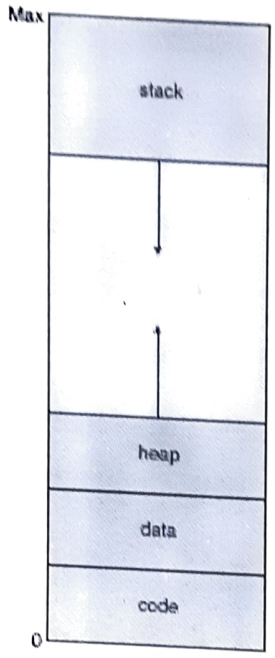
Virtual memory involves the separation of logical memory as perceived by users from physical space memory. This separation allows an extremely large virtual memory to be provided for programmers when only a small physical memory is available.




Virtual memory makes task of programming much easier as the programmer no longer needs to worry about the amount of physical memory available.

Programmer can concentrate instead on the problem to be programmed, with no concerns of memory limits.

The Virtual address space of a process refers to logical view of how a process is stored in memory.
Typically, this view is that a process begins at a certain logical address, say 0 and exists in contiguous memory.

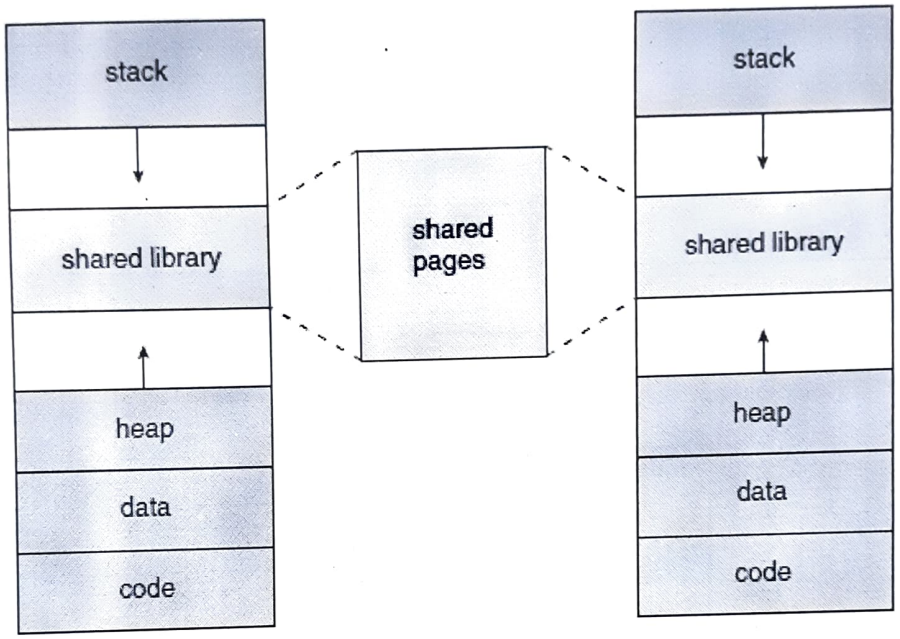



MMU maps logical pages to physical page frames in memory.

We allow heap to grow in one direction as it is used for dynamic memory allocation & Stack grows in other direction, through successive function calls.

Virtual address spaces that include holes are known as sparse address spaces.
Sparse address spaces can be utilized by stack or heap.

System libraries can be shared by several processes through mapping of the shared object into a virtual address space. The actual pages on physical space are shared by all the processes.




Similar processes can share memory.

Pages can be shared during process creation with fork() system call, thus speeding up process creation.

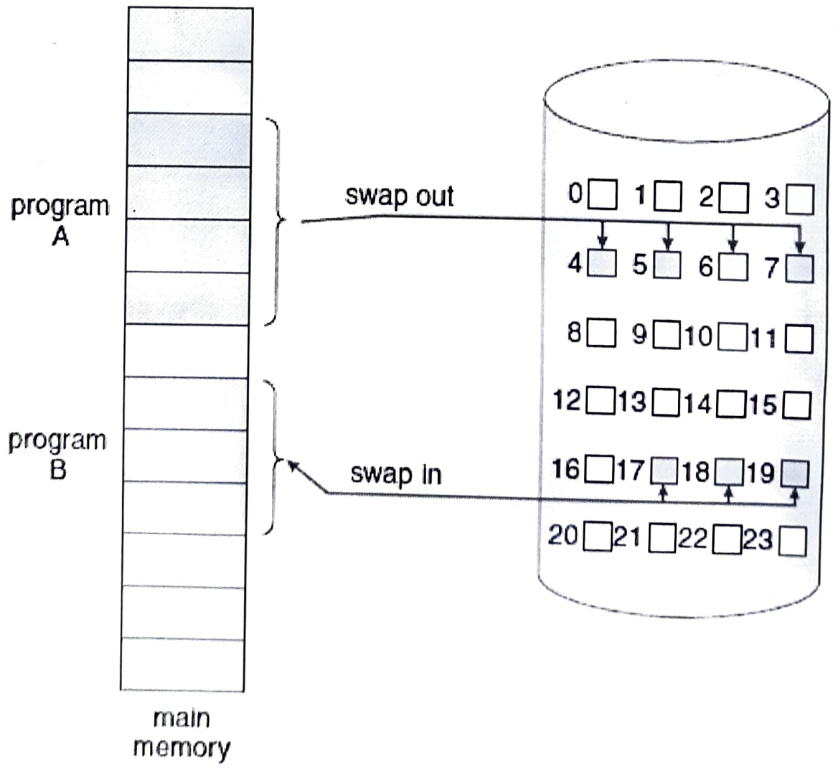
Note:

Typically, a library is mapped read-only into the space of each process that is linked with it.

# Demand Paging

Demand Paging is commonly used in virtual memory systems. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory.

A Lazy Swapper never swaps a page into memory unless that page will be needed. A "pager" is a right word than swapper as swapper manipulates entire process, where as in this scenario, we are talking about individual pages of a process.

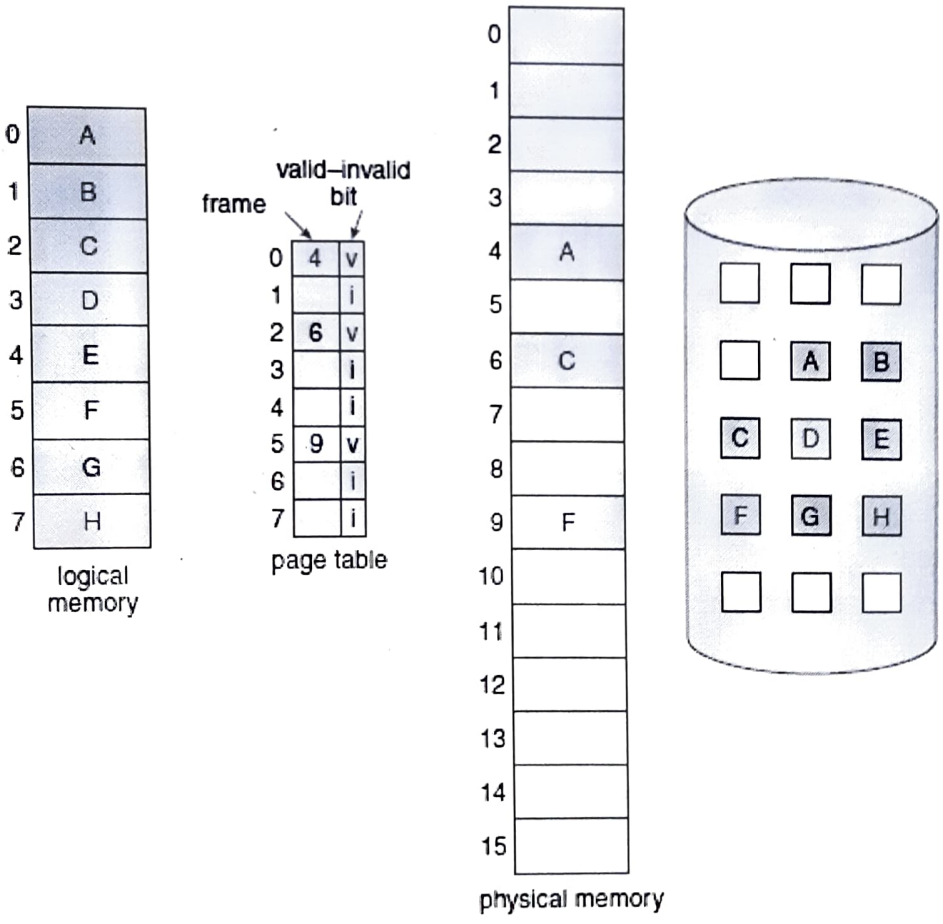


The pager guesses the pages for swap-out and swap-in.

The hardware support involves using the valid-invalid bit scheme.

Valid - the associated page is legal and in memory.

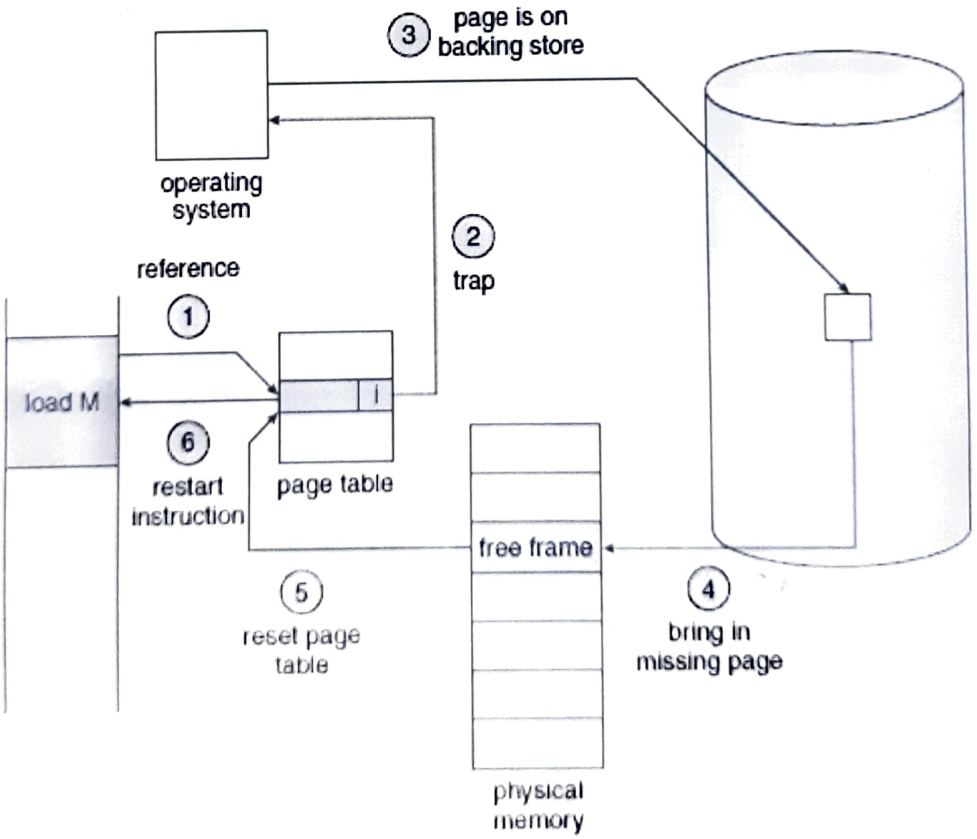
Invalid - either the page is not in the logical address space of the process or is valid but currently on the disk.




The process will run exactly as though we had brought all pages in we guess & have pages that are only needed. While the process executes and accesses pages that are

memory resident, execution proceeds normally.

Access to a page marked invalid cause a page fault. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the Operating System. This trap is the result of the OS's failure to bring the desired page into memory.




↳ Steps in handling a Page Fault

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with no pages in memory. Then pages are brought as necessary.
Pure Demand Paging - never bring a page into memory until it is required.

Programs tend to have a locality of reference, which results in resonable performance from demand paging.

tendency of a processor to access the same set of memory locations repetitively over a short period of time.

Hardware Support
Page Table - valid/invalid bits
Secondary memory - holds pages not present in main memory.

A crucial requirement for demand Paging is the ability to restart any instruction after a page fault. If a page fault occurs while we are fetching an Operand, we must fetch and decode the instruction again and then fetch the operand.

Consider the below worst case scenario:

ADD A & B & place result in C

1. Fetch and decode the instruction (ADD)
2. Fetch A
3. Fetch B
4. Add A and B
5. Store the sum in C

If we fault when we try to save C, (as C is in a page not currently in memory), we will have to get the desired page, bring it in, correct the page table, and restart the instruction.

Pagefault for fetch is rather much less work.

However, major difficulty arises when one instruction may modify several different locations.

Solutions
- ↳ make sure all relevant pages are in memory
- ↳ Use temporary registers to hold the values of overwritten locations

## Performance

Let p be the probability of a page fault ($0 \leq p \leq 1$) & we expect p to be close to zero. (minimal page faults)

effective access time (EAT) = $(1-p) * ma + p *$ page fault time

ma → memory access time (usually ranges from 100 to 200 ns)

To compute EAT, we need to know how much time is needed to service a page fault.

A page fault causes: (worst case)

1. Trap to OS
2. Save user registers & process state
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal & determine the location of the page on the disk

5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced
   b. Wait for the device seek and/or latency time
   c. Begin the transfer of the page to a free frame

6. While waiting, allocate the CPU to some other user (optional)

7. Receive an interrupt from the disk I/O subsystem (I/O completed)

8. Save the registers and process state for the other user (if step 6 is executed)

9. Determine that the interrupt was from the disk

10. Correct the page table and other tables to show that the disabled page is now in the memory

11. Wait for the CPU to be allocated to this process again

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

In any case, we are faced with 3 major components of the page-fault service time.

1. Service the page-fault interrupt
2. Read in the page
3. Restart the process

The first and third tasks can be reduced with careful coding to several hundred instructions.

With an average page fault service time of 8ms & memory access time of 200ns,

$$EAT = (1-p) * 200 + p * 8{,}000{,}000$$

$$= 200p + 7{,}999{,}800p$$

EAT & page-fault rate

If one access out of 1000 causes a page fault, EAT is 8.2 microseconds. - The computer will be slowed down by a factor of 40 because of demand paging.

If we want performance degradation to be less than 10%., we need to keep A as

$$220 > 200 + 7{,}999{,}800p$$
$$20 > 7{,}999{,}800p$$
$$p < 0.0000025$$

i.e; we need to allow fewer than one memory access out of 399,990 to page-fault. ∴ It is is important to keep page-fault rate low in a demand-paging system.

An additional aspect of demand paging is the handling and overall use of swap space.