# Q1)

**Unit 4 – Memory Management**

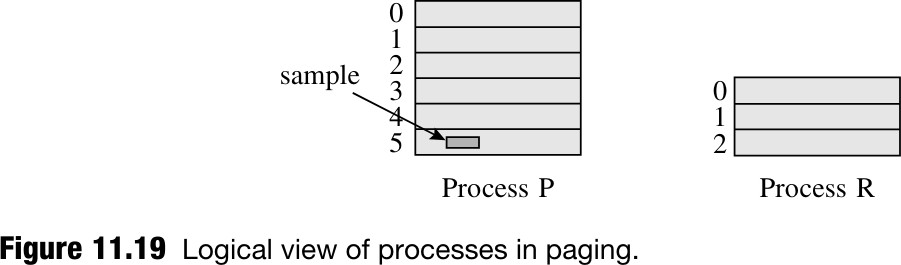
# Diagrams & Other points (Additional to Assignment)

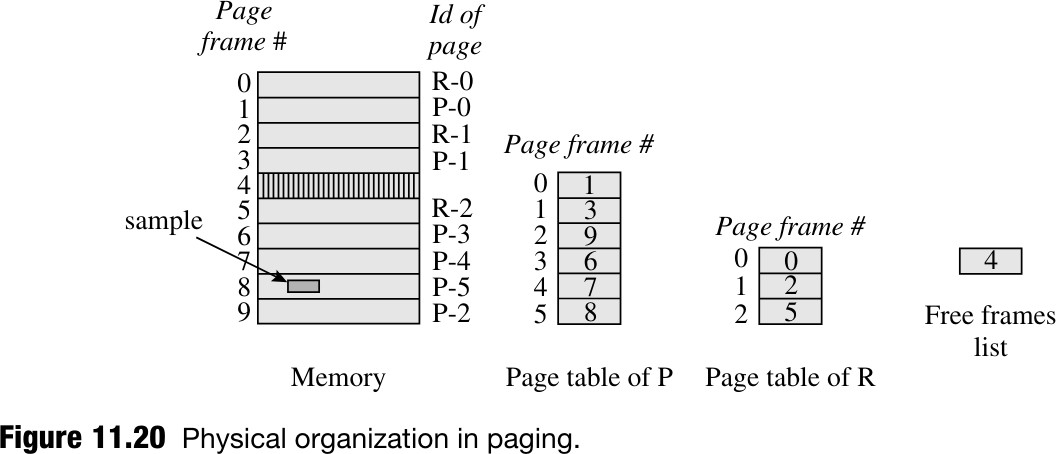
**8th Point in Difference**

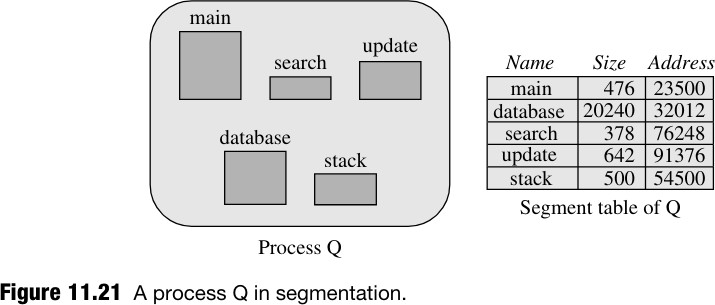
|  |  |
| --- | --- |
| **Contagious** | **Non-Contagious** |
| Swapping Unless the computer system provides a relocation register, a swapped-in process must be placed in its originally allocated area. | Components of a swapped-in process can be placed anywhere in memory |

# Q2)

**Paging –**







1. Segments in a program represent logical entities such as functions, data structures, or objects. Managing them as units allows for efficient loading into memory for

execution or sharing with other programs.

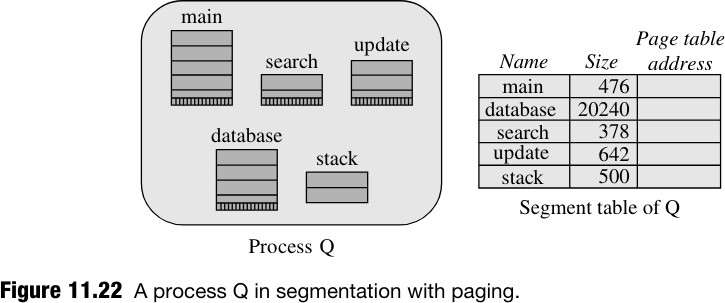
1. In the logical view, a process is composed of a collection of segments. However, in the physical view, segments of a process may exist in nonadjacent areas of memory.
2. A process named Q contains five logical entities: main, database, search, update, and stack. These are declared as segments during program coding, and each segment is

identified by a symbolic name.

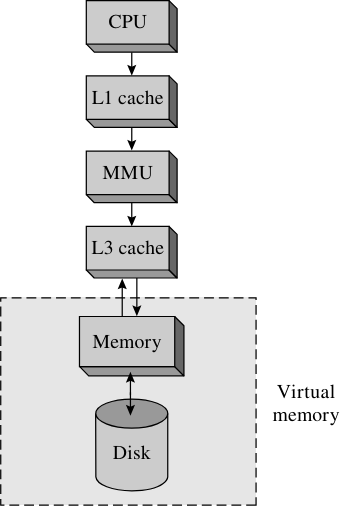
1. Each logical address in a process has the form (si, bi), where si is the segment ID and bi is the byte within the segment. This information is used by the compiler or

assembler to generate logical addresses during program translation.

1. The kernel constructs a segment table for each process, containing information about segment sizes and memory addresses allocated to them. The Memory Management Unit (MMU) uses this table for address translation during execution.
2. Address translation involves adding the byte offset (bi) to the start address of the segment (si) to calculate the effective memory address. This process ensures proper mapping of logical addresses to physical memory locations.
3. Memory allocation for each segment follows the contiguous memory allocation model. The kernel maintains a free list of memory areas and performs first-fit or best- fit allocation while loading a process. Upon termination, the memory areas allocated to its segments are returned to the free list.
4. External fragmentation can occur due to segments having different sizes. This means that although free memory exists, it may not be contiguous, leading to inefficient memory utilization.

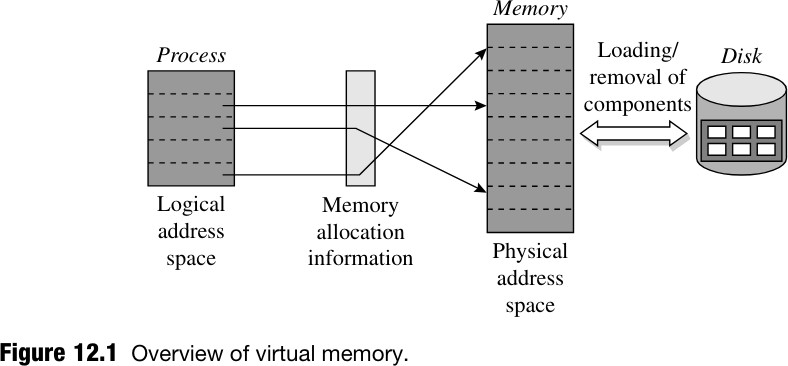


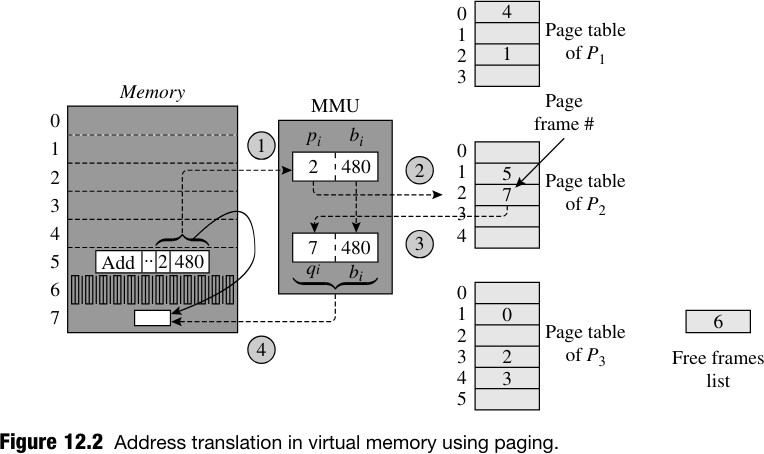
1. Each segment in a program is paged separately, with an integral number of pages allocated to each segment. This simplifies memory allocation, speeds it up, and avoids external fragmentation.
2. A page table is constructed for each segment, and the address of the page table is stored in the segment's entry in the segment table.
3. Address translation for a logical address (si, bi) is done in two stages.
   * In the first stage, the entry of si is located in the segment table, and the address of its page table is obtained.
   * In the second stage, the byte number bi is split into a pair (psi, bpi), where psi is the page number in segment si, and bpi is the byte number in page pi.
   * The effective address calculation is completed by obtaining the frame number of psi and concatenating it with bpi to obtain the effective address.
4. Each segment is paged independently, leading to internal fragmentation in the last page of each segment due to partial page utilization.
5. Each segment table entry now contains the address of the page table of the segment. The size field in a segment's entry is used to facilitate a bound check for memory protection.



# Q4)

**Virtual Memory –**





# Q6)

**Page Replacement Policies –**

# Optimal page replacement policy –

1. Optimal page replacement aims to minimize the total number of page faults

during the operation of a process. It selects pages for replacement in such a way that no other sequence of replacements could result in fewer page faults.

1. Achieving optimal page replacement would require considering all possible page replacement decisions at each page fault and selecting the best one based on future implications. However, this approach is infeasible in reality because the

virtual memory manager lacks knowledge of the future behavior of a process.

1. Despite the apparent complexity of optimal page replacement, Belady (1966) demonstrated that it is equivalent to a simple rule: replace the page whose next reference is farthest in the page reference string.

# Advantages:

1. FIFO is straightforward to implement and understand, making it suitable for systems with limited resources or simpler requirements.
2. It requires minimal computational overhead, as it only needs to maintain the order of pages in memory.
3. FIFO can suffer from Belady's anomaly, where increasing the number of frames can paradoxically lead to more page faults.
4. FIFO may replace active pages prematurely, leading to inefficient memory utilization, especially in scenarios with cache-like access patterns.

# First-in, first-out (FIFO) page replacement policy –

Ans in assignment is sufficient.

# Least recently used (LRU) page replacement policy –

1. The LRU (Least Recently Used) policy relies on the principle of locality of reference, which suggests that recently accessed pages are more likely to be accessed again in the near future.
2. At every page fault, the LRU policy replaces the page that has been least recently used. In other words, it selects the page that hasn't been accessed for the longest time.
3. Each page table entry records the time when the page was last referenced. This

information is initialized when the page is loaded into memory and updated every time the page is accessed.

1. When a page fault occurs, the system uses the recorded last reference times to locate the page that was least recently used. This page is identified as the one whose last reference occurred earlier than that of every other page in memory.
2. Once the LRU page is identified, it is replaced with the page that is required to satisfy the current page fault. This ensures that the page replaced is the one that has been unused for the longest duration, based on the recorded reference times.

# Advantages:

1. LRU tends to maximize memory utilization by retaining frequently accessed pages, reducing the overall number of page faults.
2. LRU is less susceptible to Belady's anomaly compared to FIFO, as it prioritizes retaining recently used pages in memory.

# Disadvantages:

1. Implementing LRU requires additional bookkeeping to track the order of page accesses, increasing computational overhead and implementation complexity.
2. Maintaining the order of page accesses in LRU can incur higher computational

overhead, especially in systems with a large number of pages or frequent accesses.

# Q7)

**Segmentation --- ans in Q2**

# Q8)

|  |  |
| --- | --- |
| **Static Memory Allocation** | **Dynamic Memory Allocation** |
| Memory allocated at compile/load time | Memory allocated dynamically during  program execution |
| Allocates fixed-size chunks determined  beforehand | Allows for variable-size allocation based on  runtime conditions |
| Requires known memory requirements in  advance | Provides flexibility for adjusting memory  sizes dynamically |
| Faster access times due to known memory  locations | Incurs runtime overhead for allocation and  deallocation |
| No runtime overhead associated with  allocation | May introduce complexities like memory  leaks or fragmentation |
| Lack of flexibility may lead to inefficient memory usage | Allows creation of variable-sized data structures, adapting to runtime  requirements |
| Memory management is less complex | More complex memory management due  to tracking and deallocation |
| Typically used for global/static variables and  fixed-size arrays | Used for dynamic data structures like linked  lists and dynamic arrays |