# Module 4

# Chapter 8: Memory-Management Strategies

## 8.5 Paging:

- Paging is a memory management scheme that allows processes to be stored in physical memory discontinuously.
- It eliminates problems with fragmentation by allocating memory in equal sized blocks known as **pages.**

### 8.5.1 Basic Method:

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called **frames,** and to divide a program's logical memory space into blocks of the same size called **pages.**
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.
- The hardware support for paging is illustrated in Figure8.7.

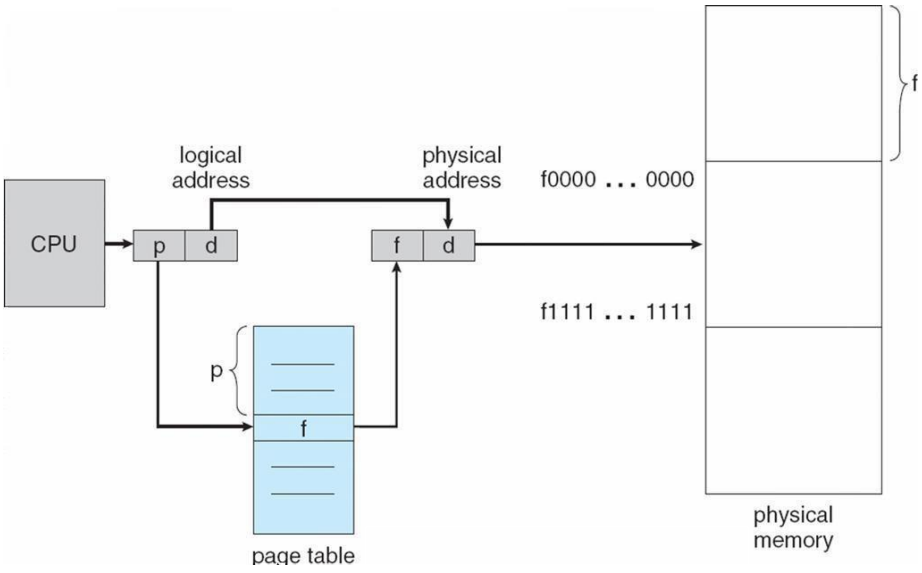



**Figure 8.7: Paging Hardware**

- Every **address** generated by the **CPU is divided into two parts**: a **Page number(p)** and a **Page offset (d)**.
- **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.
- The paging model of memory is shown in figure 8.8.

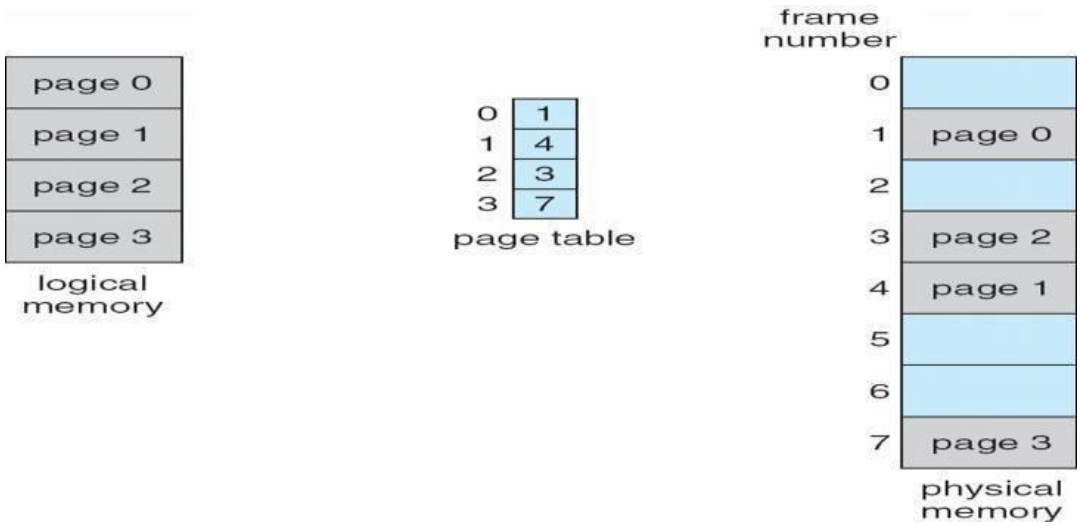
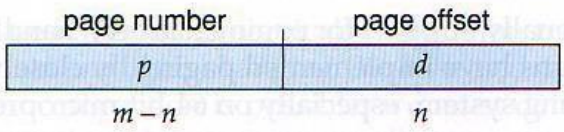



**Figure 8.8: Paging Model of Logical and Physical Memory**

- The page 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.
- The size of logical address space is $2^m$ and page size $2^n$ addressing units, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.
- The logical address is as follows:




where p is an index into the page table and d is the displacement within the page.

**Question: Consider the following example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory).**

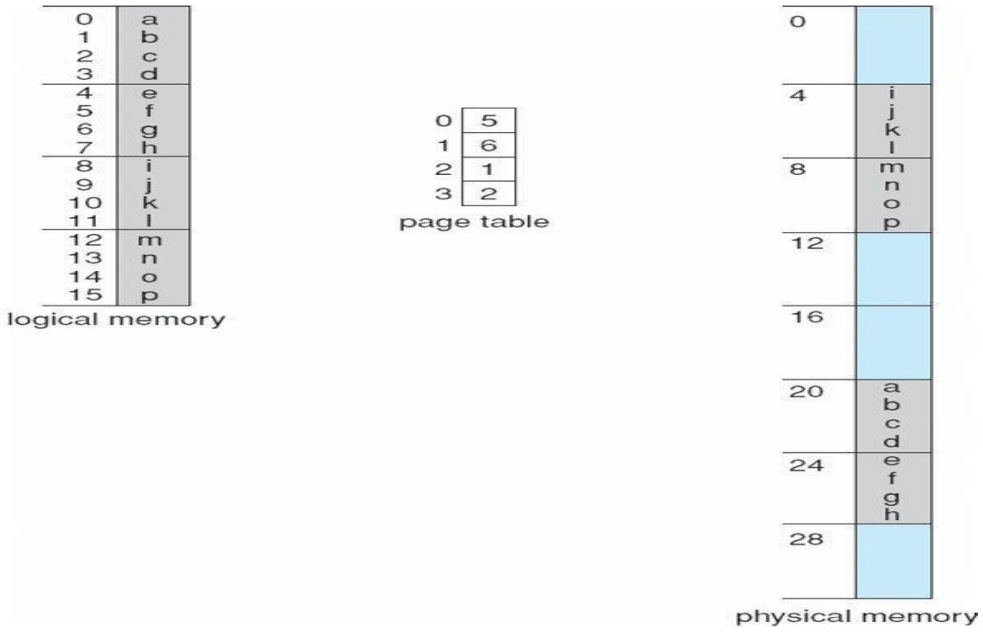



**Figure 8.9: Paging example for a 32-byte memory and 4-byte pages**

**Solution**: The 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 [= (2*4)+1].
- We may notice that, paging itself is a form of dynamic relocation.
- Every logical address is bound by the paging hardware to some physical address.
- Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

**Free frames:**

- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table.

- There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process.
- This increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running).

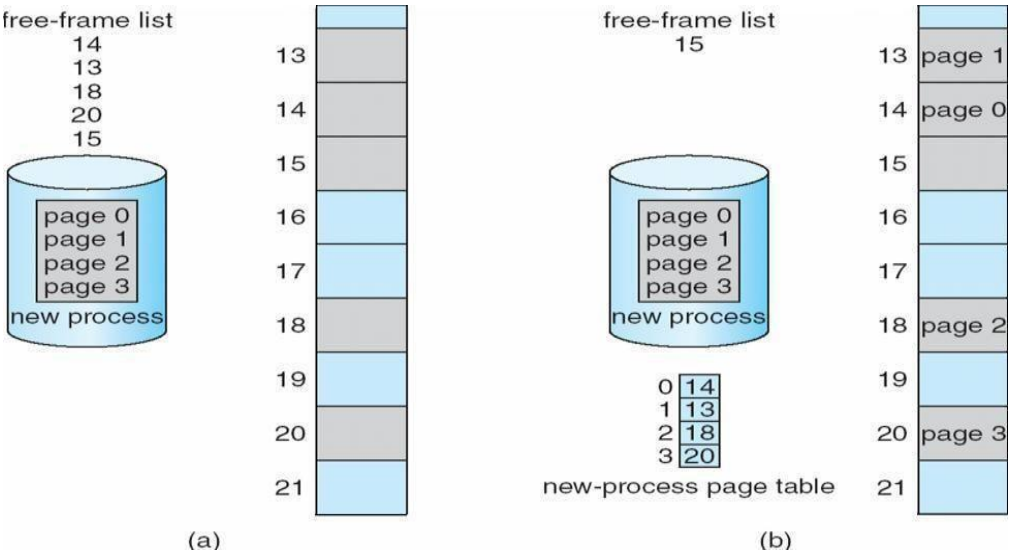



**Figure 8.10: Free Frames (a) before allocation and (b) after allocation**

### 8.5.2 Hardware Support:

- The hardware implementation of the page table can be done in several ways.
- In the simplest case, the page table is implemented as a set of **dedicated registers**.
- Here each register content is loaded, when the program is loaded into memory.
- For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process.
- An alternate option is to store the page table in main memory, and to use a single register called the **page-table base register, (PTBR )** to record the address of the page table is memory.
- Process switching is fast, because only the single register needs to be changed.

  Memory access is slow, because every memory access now requires two memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.

- The solution to this problem is to use a very special high-speed memory device called the **Translation look-aside buffer, TLB.**
- **The benefit of the TLB: (Important)**

➢ It can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.

➢ It is used as a cache device.

➢ Addresses are first checked against the TLB, and if the page is not there (a **TLB miss**), then the frame is looked up from main memory and the TLB is updated.

➢ If the TLB is full, then replacement strategies range from least-recently used, **LRU** to **random**.

➢ Some TLBs allow some entries to be wired down, which means that they cannot be removed from the TLB. Typically these would be kernel frames.

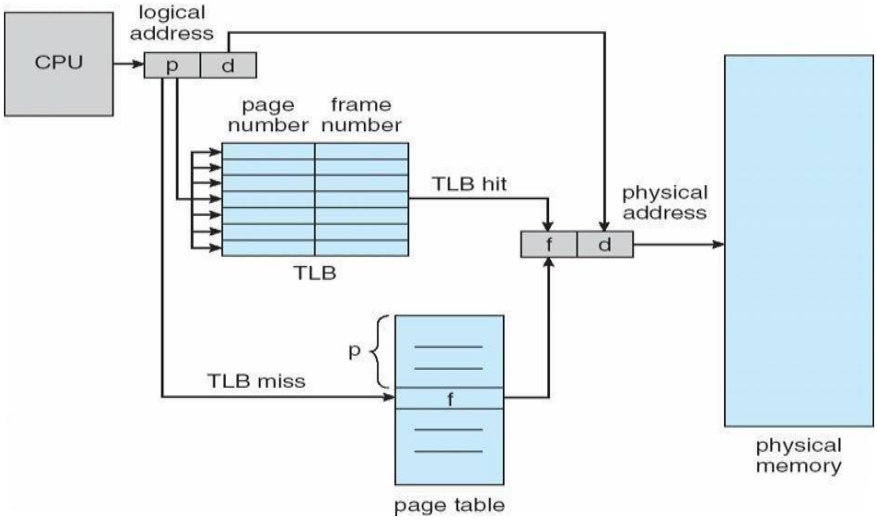



**Figure 8.11: Paging Hardware with TLB**

- Some TLBs store **address-space identifiers, ASIDs**, to keep track of which process "owns" a particular entry in the TLB.
- This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location.
- Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the **hit ratio.**
- **For example**, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB.

- So a TLB hit takes 120 nanoseconds total ( 20 to find the frame number and then another 100 to go get the data ), and a TLB miss takes 220 ( 20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.)
- So with an **80%** TLB hit ratio, the **average memory access time** would be:

  **0.80 * 120 + 0.20 * 220** = **140** nanoseconds

  for a **40%** slowdown to get the frame number.
- A **98%** hit rate would yield **122** nanoseconds average access time for a **22%** slowdown.

### 8.5.3 Protection:

- Memory protection in a paged environment is accomplished by protection bits associated with each frame.
- These bits are kept in the page table.
- Valid-invalid bit attached to each entry in the page table:
  1. **"valid"** indicates that the associated page is in the process's logical address space, and is thus a legal page.
  2. **"invalid"** indicates that the page is not in the process's logical address space.
- In figure 8.12, Addresses of the pages 0, 1, 2, 3, 4 and 5 are mapped using the page table as they are valid.
- Addresses of the pages 6 and 7 are invalid and cannot be mapped.
- Any attempt to access those pages will send a trap to the OS.

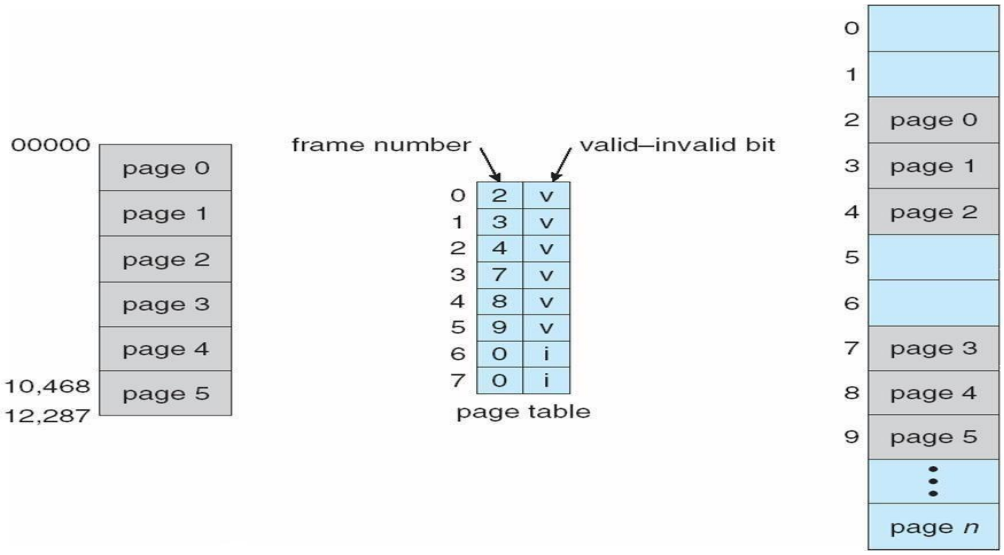



**Figure 8.12: Valid (v) or Invalid (i) bit in a page table.**

### 8.5.4 Shared Pages:

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames.
- This may be done with either **code or data**.
- If code is **reentrant code** (read-only files) that means that it does not write to or change the code in anyway.
- More importantly, it means the code can be shared by multiple processes, as long as each has their own copy of the data and registers, including the instruction register.
- For example in the figure 8.13, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.
- Some systems also implement shared memory in this fashion.

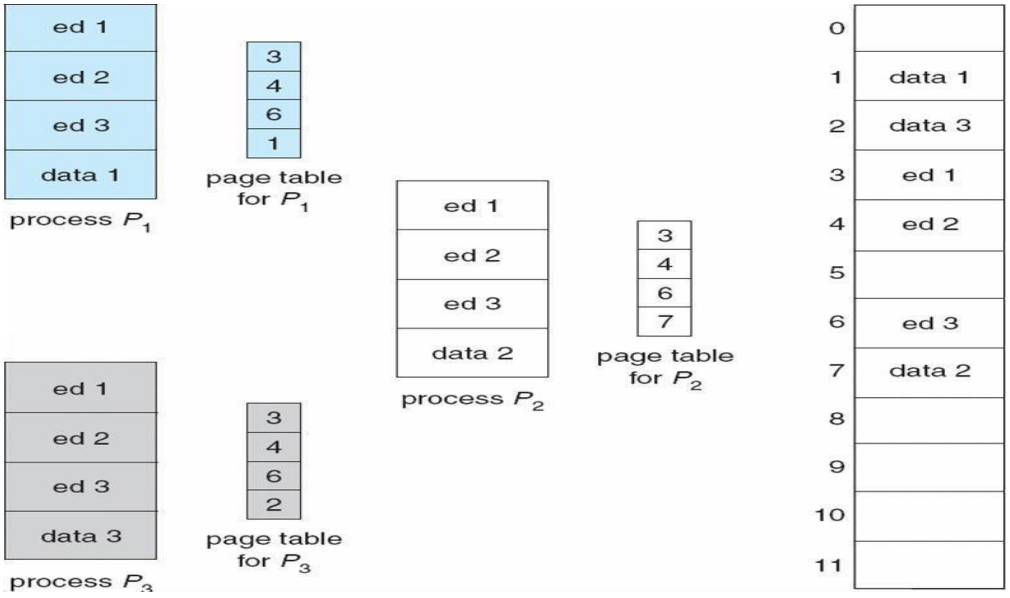



**Figure 8.13: Sharing of code in a paging environment.**

## 8.6 Structure of the Page Table:

### 8.6.1 Hierarchical Paging:

- This structure breaks up the logical address space into multiple page tables at different levels (tier).
- A simple technique is a two-level page table.
- Most modern computer systems support logical address spaces of 2^32 to2^64.

- **Two-Level Paging Example**
- A logical address (on **32-bit** machine with 1K page size) is divided into:
  - a **page number** consisting of **22 bits**
  - a **page offset** consisting of **10 bits**
- Since the **page table is paged**, the **page number is further divided into:**
  - a **12-bit page number**
  - a **10-bit page offset**

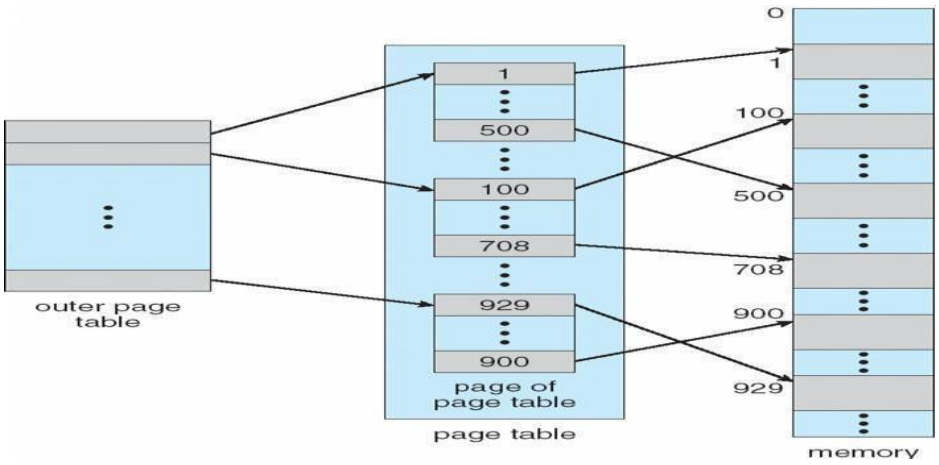
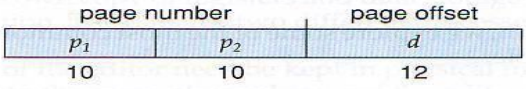



**Figure 8.14: Two-Level Page-Table Scheme**

- Thus, a logical address is as follows:




where p1 is an index into the outer page table, and p2 is the displacement within the page of the outer page table.

- The address translation method for this architecture is shown in **figure 8.15**, because address translation works from the outer page table inward, this scheme is also known as forward-mapped page table.

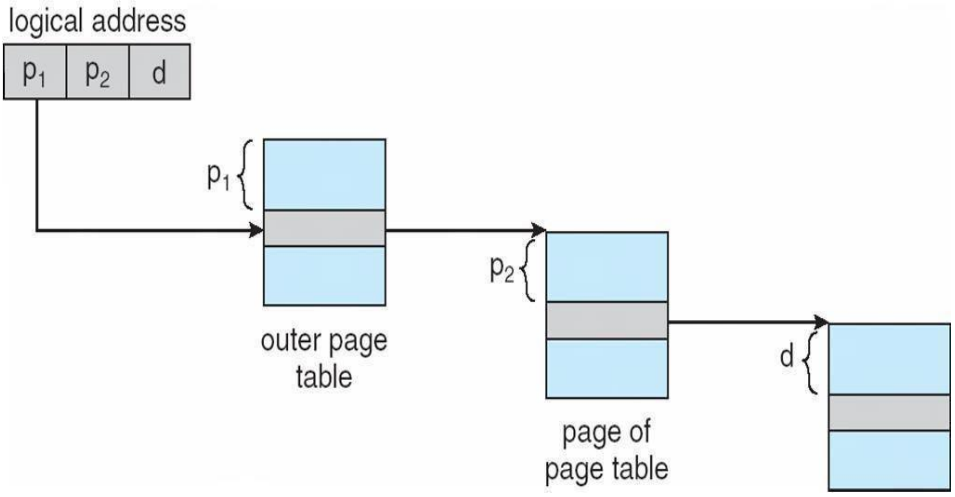
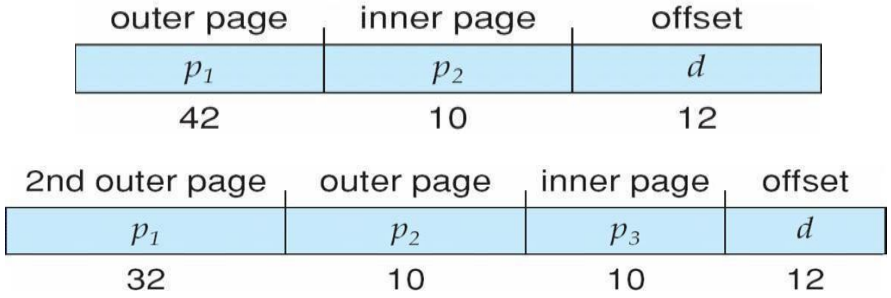



**Figure 8.15: Address-Translation for a two-level 32-bit paging architecture**

- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging.
- One could increase the paging level, but with 10-bit page tables it would take 7 levels of in direction, which would be prohibitively slow memory access.
- So some other approach must be used.




**Three-level Paging Scheme**

### 8.6.2 Hashed Page Tables:

- One common approach for handling address space larger than 32 bits is to use **hashed page table**, with **hash values** being the **virtual page number**.
- Each entry in hash table contains linked list of elements that hash to same virtual page number.
- Each element contains **3 fields**: **(1) virtual page number, (2) value of mapped page frame, (3) pointer to next element in the linked list.**

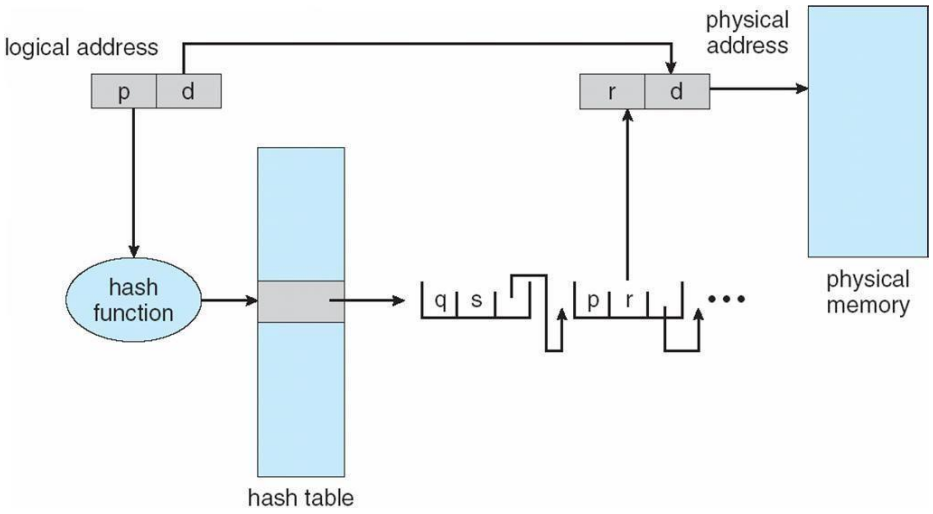



**Figure 8.16: Hashed Page Table**

- **Algorithm works as follows:** The virtual page number is hashed into a hash table.
- Virtual page numbers are compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (Field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for matching virtual page number.

### 8.6.3 Inverted Page Tables:

- Usually, each process has an associated page table. The page table has one entry for each page that the process is using.
- This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address.
- Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly.
- One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.
- To solve this problem, we can use an **inverted page table**.
- An inverted page table has one entry for each real page (or frame) of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page.
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

- Inverted page tables often require that an address-space identifier be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page.
- The 'id' of process running in each frame and its corresponding page number is stored in the page table.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.
- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.
- Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.
- Each inverted page-table entry is a pair

  **<process-id, page-number, offset>**

  where the process-id assumes the role of the address-space identifier.
- When a memory reference occurs, part of the virtual address, consisting of , is presented to the memory subsystem.
- The inverted page table is then searched for a match.
- If a match is found—say, at entry i—then the physical address is generated.
- If no match is found, then an illegal address access has been attempted.

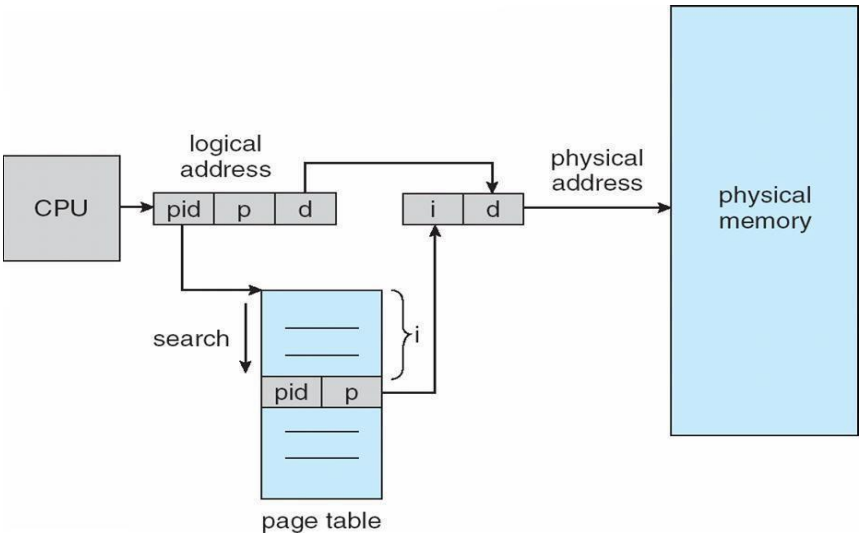



**Figure 8.17: Inverted Page Table**

# Chapter 9: Virtual Memory Management

- **Virtual memory** is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.

## 9.1 Background:

- **In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:**
  - Error handling code is not needed unless that specific error occurs, some of which are quite rare.
  - Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
  - Certain features of certain programs are rarely used.
- Even in those cases where the entire program is needed, it may not all be needed at the same time.
- **The ability to load only the portions of processes that are actually needed has several benefits:**
  - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
  - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
  - Less I/O is needed for swapping processes in and out of RAM, speeding things up.
- Thus, running a program that is most entirely in memory would benefit both the system and the user.
- **Virtual memory** is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- The figure below shows the general layout of virtual memory, which can be much larger than physical memory:

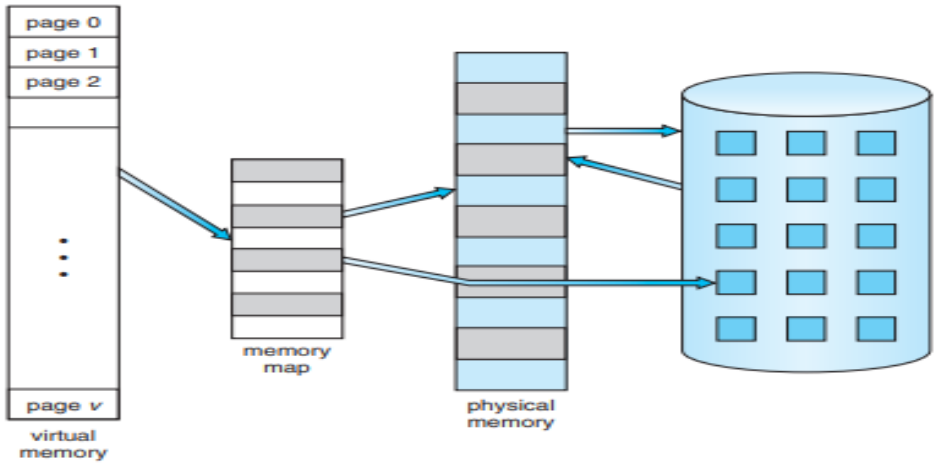



**Figure 9.1** Diagram showing virtual memory that is larger than physical memory.

- The **virtual address space** of a process refers to the logical (or virtual) view of how process is stored in the memory.
- This view is that a process begins at a certain logical address- say, address 0 – and exists in contiguous memory, as shown in figure 9.2
- The physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous.
- The address space shown in Figure 9.2 is **sparse** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

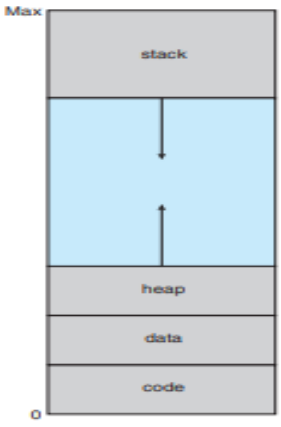



**Figure 9.2** Virtual address space.

- **Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits**:

1. System libraries can be shared by mapping them into the virtual address space of more than one process. (Fig 9.3)

2. Processes can also share virtual memory by mapping the same block of memory to more than one process.

3. Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

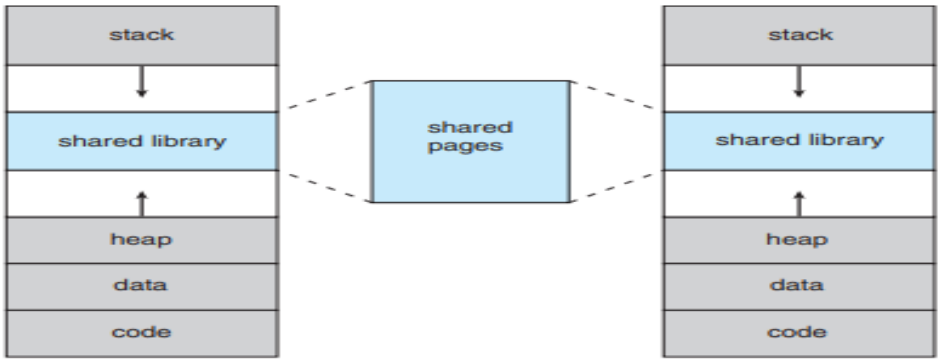



**Figure 9.3** Shared library using virtual memory.

## 9.2 Demand Paging:

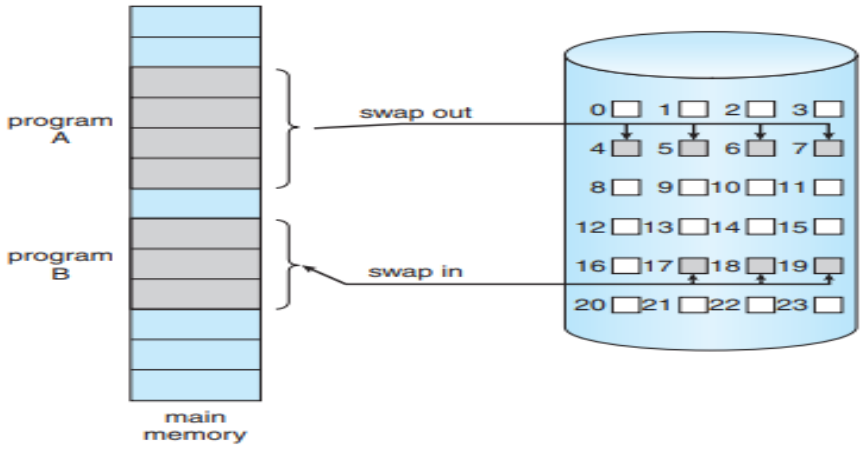



**Figure 9.4** Transfer of a paged memory to contiguous disk space.

- A demand paging is similar to a paging system with swapping (fig 9.4), where processes reside in secondary memory (usually disk).
- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, we use a **lazy swapper**.
- **A lazy swapper never swaps a page into memory unless that page will be needed**.
- A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process.
- We thus use pager, rather than swapper, in connection with demand paging.

### 9.2.1 Basic Concepts:

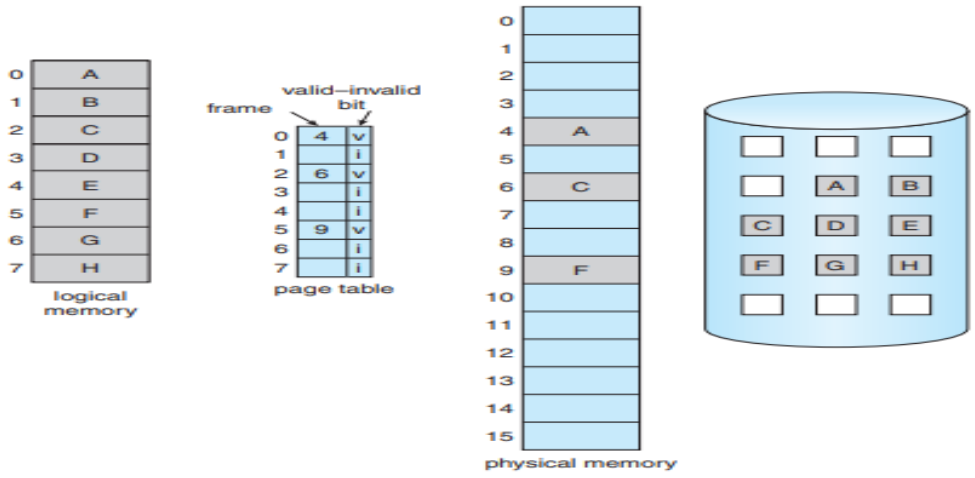



**Figure 9.5** Page table when some pages are not in main memory.

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages into memory.
- Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.
- Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme.
- When the bit is set to "**valid**", the associated page is both legal and in memory.
- If the bit is set to "**invalid**", the page either is not valid or is valid but is currently on the disk.
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. Pages loaded in memory are marked as valid. (Fig 9.5)
- If the process only accesses pages that are loaded in memory (**memory resident pages),** then the process runs exactly as if all the pages were loaded in to memory.
- Accessing a page marked invalid causes a **page-fault trap**. This trap is the result of the operating system's failure to bring the desired page into memory.

- **Page fault can be handled as following**: (fig 9.6)

**Important**

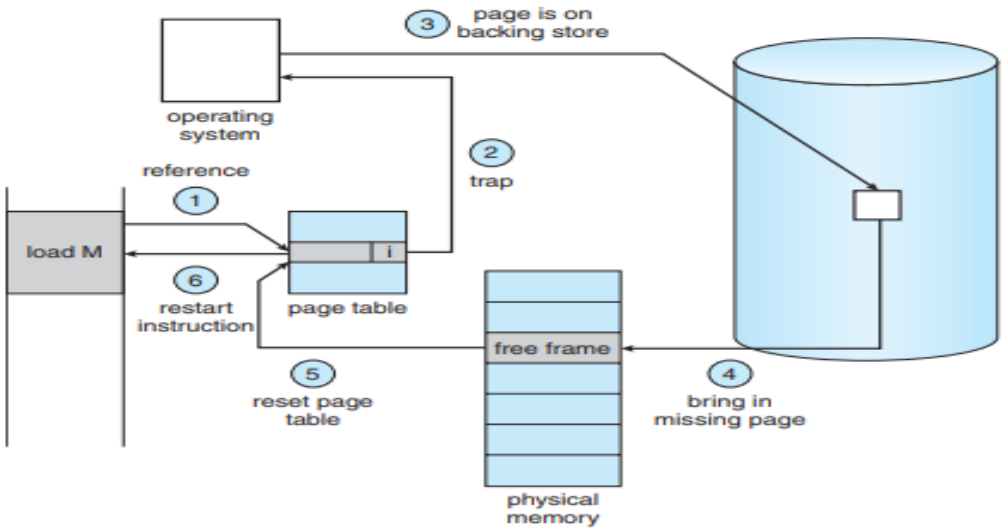



**Figure 9.6** Steps in handling a page fault.

1. We check an internal table for this process to determine whether the reference was a valid or 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 in the latter.
3. We find a free frame.
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 illegal address trap. The process can now access the page as though it had always been memory.

- In an extreme case, the program starts execution with zero pages in memory.
- Here NO pages are swapped in for a process until they are requested by page faults. This is known as **pure demand paging.**
- The hardware necessary to support demand paging is the same as for paging and swapping: **A page table and secondary memory.**
- **Page table:** This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
- **Secondary memory:** This memory holds those pages that are not present in main memory.
- The secondary memory is usually a high-speed disk. It is known as the swap device and the section of disk used for this purpose is known as **swap space**.

### 9.2.2 Performance of Demand Paging:

- Demand paging can significantly affect the performance of a computer system.
- **Effective access time** for demand-paged memory is computed using,

  **Effective access time = (1-p) * ma + p * page fault time**

- The memory –access time, denoted ma, ranges from 10 to 200 nanoseconds.
- As long as we have no page faults, the effective access time is equal to the memory access time.
- If a page fault occurs, we must read relevant page from disk and then access the desired work.
- Let p be the probability of a page fault ($0 \leq p \leq 1$).
- Suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access)
- With a page fault rate of p, ( on a scale from 0 to 1 ), the effective access time is now:
- Effective access time = p * time taken to access memory in page fault+ (1-p)* time taken to access memory

  = p * 8000000 + (1 - p ) * ( 200 )

  **= 200 + 7,999,800 * p**

- Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times.
- In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

### 9.3 Copy-on-Write:

- The idea behind a copy-on-write is that the pages of a parent process are shared by the child process, until one or the other of the processes changes the page.
- Only when a process changes any page content, that page is copied for the child.
- Only pages that can be modified need to be labeled as copy-on-write. Code segments can simply be shared.
- Some systems provide an alternative to the fork( ) system call called a **virtual memory fork, vfork( ).**
- In this case the parent is suspended, and the child uses the parent's memory pages.

- This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the exec( ) system call.

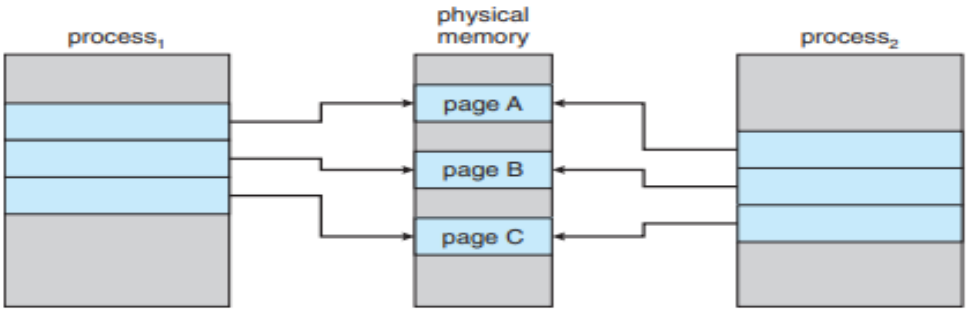



**Figure 9.7** Before process 1 modifies page C.

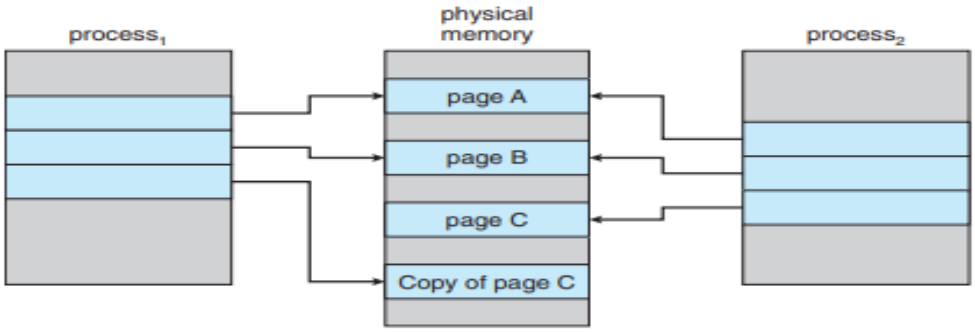



**Figure 9.8** After process 1 modifies page C.

### 9.4 Page Replacement:

- If some process suddenly decides to use more pages and there aren't any free frames available.
- Then there are several possible solutions to consider:

1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes.
2. Put the process requesting more pages into a wait queue until some free frames become available.
3. Swap some process out of memory completely, freeing up its page frames.
4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it.

- This is known as **page replacement**, and is the most common solution. There are many different algorithms for page replacement.

### 9.4.1 Basic Page Replacement: numericals

- **Page replacement takes the following approach.**

1. If no frame is free, we find one that is not currently being used and free it.
2. We can free a frame by writing its contents to swap space and changing the page table to indicate that the page is no longer in memory.(Fig 9.10)

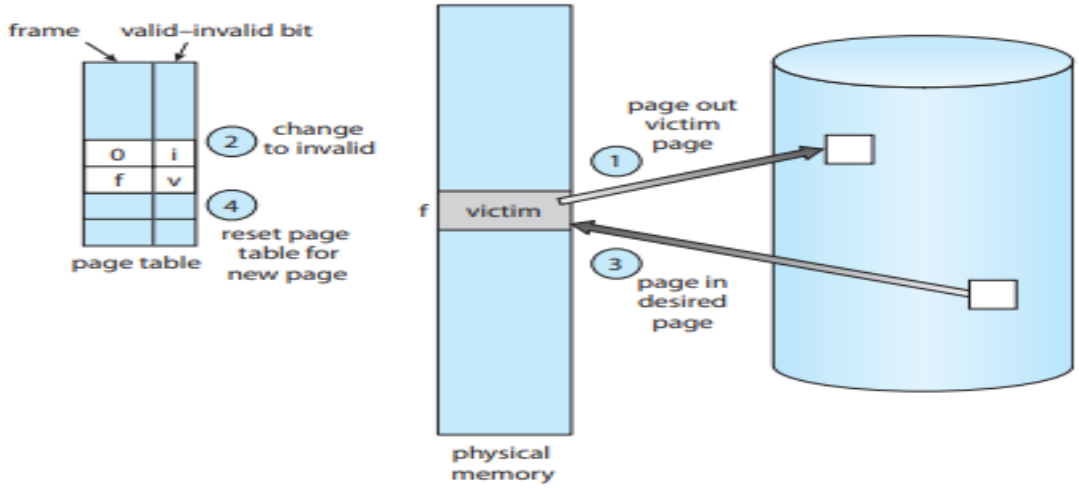



**Figure 9.10** Page replacement.

- **Now the page-fault handling must be modified to free up a frame if necessary, as follows:**

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   **b.** If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the **victim frame.**
   c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
3. Read in the desired page and store it in the frame. Change the entries in page table.
4. Restart the process that was waiting for this page.

- **There are two major requirements to implement a successful demand paging system.**

1. A **frame-allocation** algorithm: how many frames are allocated to each process.
2. **Page-replacement algorithm**: deals with how to select a page for replacement when there are no free frames available.

- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults.

- Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of page accesses known as a **reference string.**

### 9.4.2 FIFO Page Replacement:

- A simple and obvious page replacement strategy is **FIFO,** i.e. first-in-first-out.
- This algorithm associates with each page the time when that page was brought into memory. **When a page must be replaced, the oldest page is chosen.**
- Or a FIFO queue can be created to hold all pages in memory. As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.
- In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults.

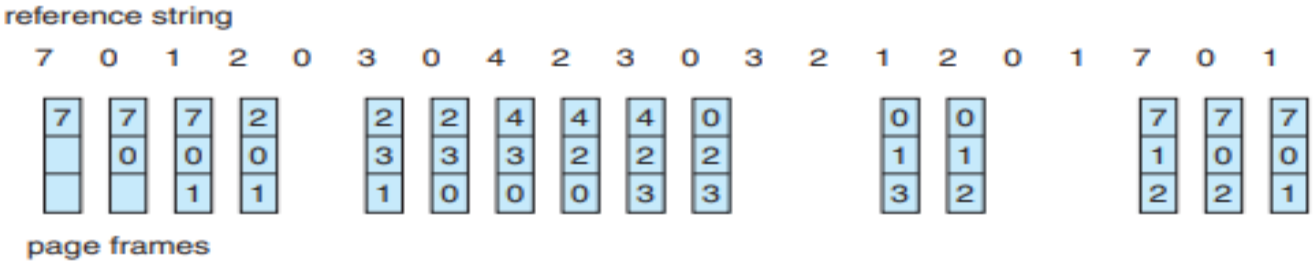



**Figure 9.12** FIFO page-replacement algorithm.

- Although FIFO is simple and easy to understand, it is not always optimal, or even efficient.
- **Belady's anomaly tells that for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases**.

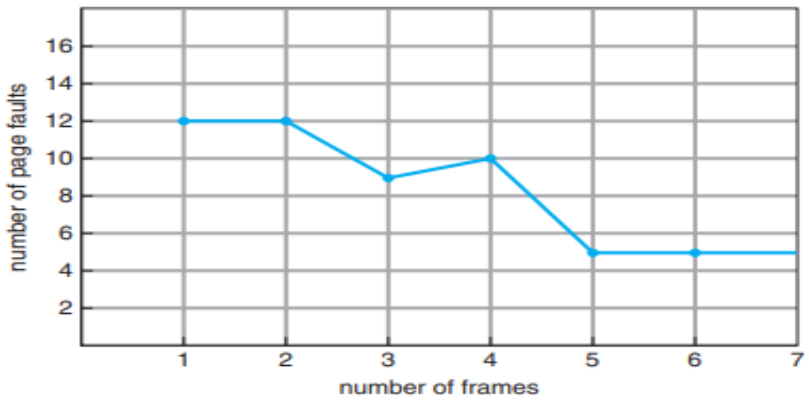



**Figure 9.13** Page-fault curve for FIFO replacement on a reference string.

### 9.4.3 Optimal Page Replacement:

- The discovery of Belady's anomaly lead to the search for an optimal page-replacement algorithm, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called OPT or MIN. This algorithm is **"Replace the page that will not be used for the longest time in the future."**
- The same reference string used for the FIFO example is used in the example below; here the minimum number of possible page faults is 9.

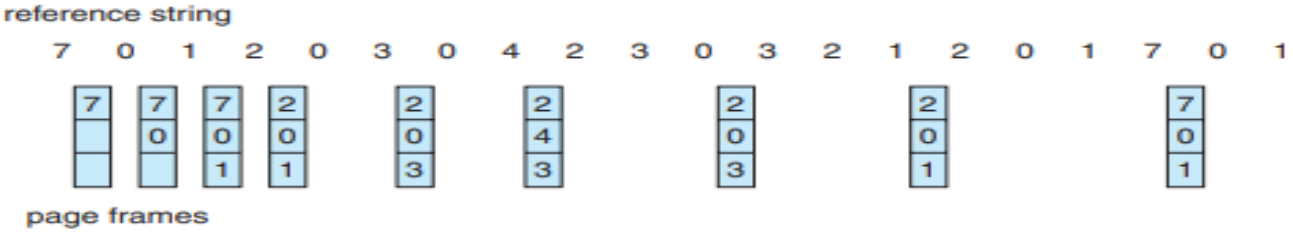



**Figure 9.14** Optimal page-replacement algorithm.

- Unfortunately OPT cannot be implemented in practice, because it requires the knowledge of future string, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.

### 9.4.4 LRU Page Replacement:

- The **LRU (Least Recently Used)** algorithm, predicts that **the page that has not been used in the longest time** is the one that will not be used again in the near future.
- Some view LRU as analogous to OPT, but here we look backwards in time instead of forwards.
- Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT).
- LRU is considered a good replacement policy, and is often used.

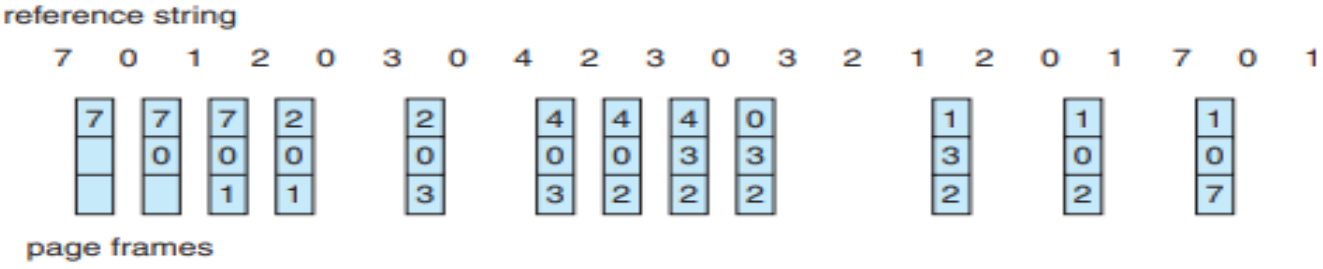



**Figure 9.15** LRU page-replacement algorithm.

- There are two simple approaches commonly used to implement this:

1. **Counters.** With each page-table entry a time-of-use field is associated. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.

- In this way, we always have the "time" of the last reference to each page.

- This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access.

2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top.

- The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called **stack algorithms,** which can never exhibit Belady's anomaly.

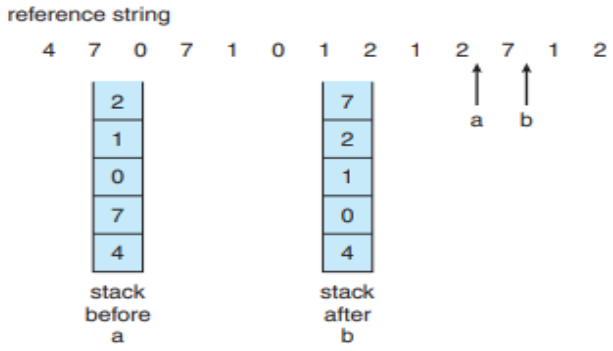



**Figure 9.16** Use of a stack to record the most recent page references.

## 9.6 Thrashing:

self study topic

- Thrashing is the state of a process where there is high paging activity.

- A process that is spending more time paging than executing is said to be **thrashing**.

### 9.6.1 Cause of Thrashing:

- When memory is filled up and processes starts spending lots of time waiting for their pages to page in, then CPU utilization decreases (Processes are not executed as they are waiting for some pages), causing the scheduler to add in even more processes and increase the degree of multiprogramming even more.

- Thrashing has occurred, and system throughput plunges.

- No work is getting done, because the processes are spending all their time paging.

- In the graph given, CPU utilization is plotted against the degree of multiprogramming.
- As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.
- Local page replacement policies can prevent thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue.

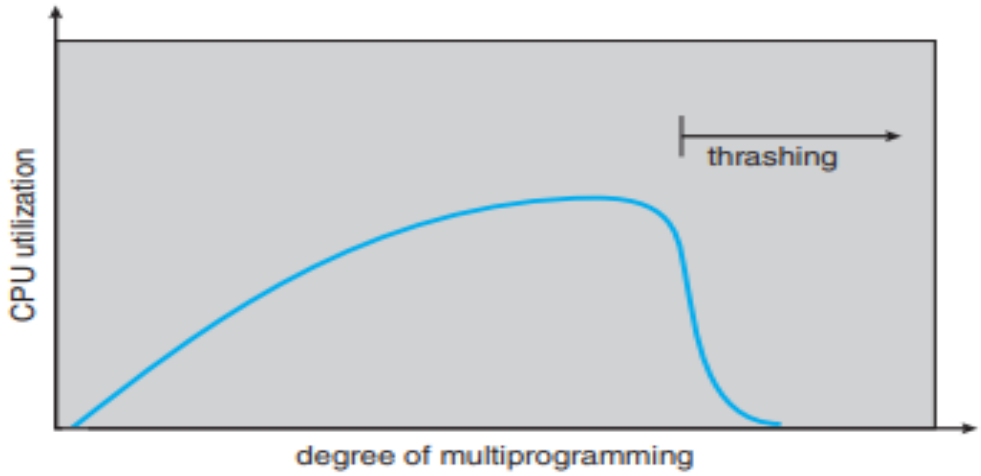



**Figure 9.18** Thrashing.

### 9.6.2 Working-Set Model:

- The **working set model** is based on the concept of locality, and defines a working set window, of length delta.
- Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure 9.20:

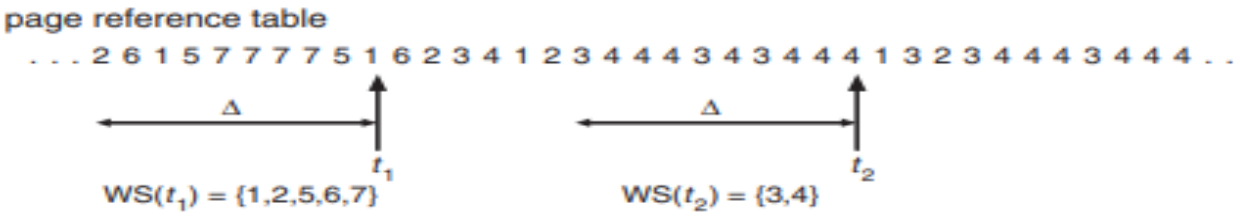



**Figure 9.20** Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand of frames, D, is the sum of the sizes of the working sets for all processes (D=WSSi).
- If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set.
- If D is significantly less than the currently available frames, then additional processes can be launched.

### 9.6.3 Page-Fault Frequency:

- When page-fault rate is too high, the process needs more frames and when it is too low, the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate.
- If the actual page-fault rate exceeds the upper limit, allocate the process another frame or suspend the process.
- If the page-fault rate falls below the lower limit, remove a frame from the process.
- Thus, we can directly measure and control the page-fault rate to prevent thrashing.

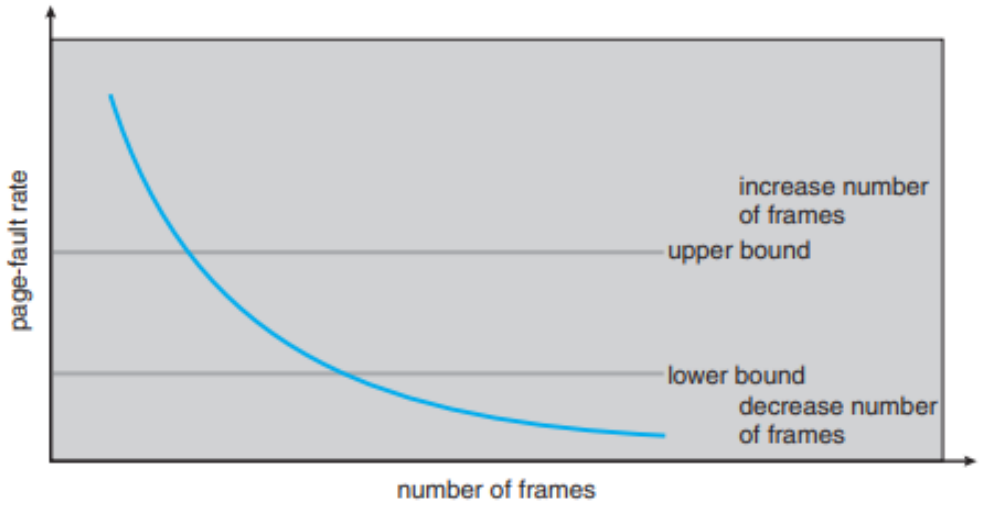



**Figure 9.21** Page-fault frequency.