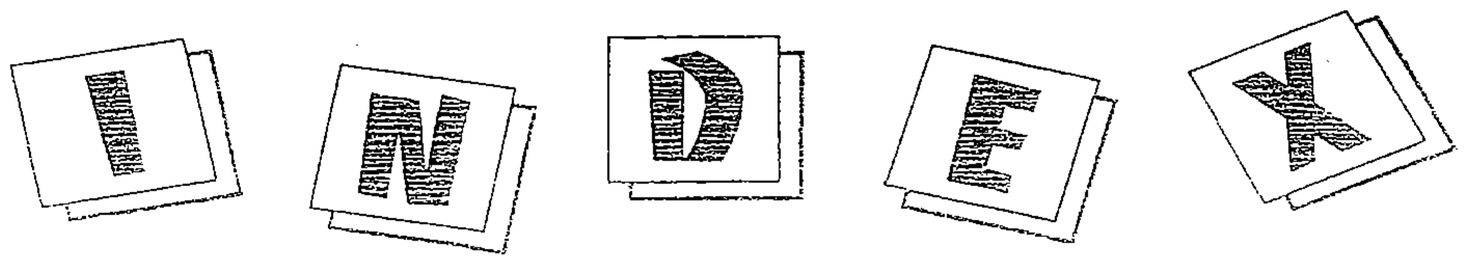


NAME: KIRAN M. STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: _____

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		10CS53 SEMESTER - V		
		OPERATING SYSTEMS		
		PART - B		
UNIT-5		MEMORY MANAGEMENT		
CHAPTER-8		MEMORY MANAGEMENT STRATEGIES	1-29	
CHAPTER-9		VIRTUAL-MEMORY MANAGEMENT	29-47	
UNIT-6		FILE SYSTEM, IMPLEMENTATION OF FILE SYSTEM		
CHAPTER-10		FILE SYSTEM	48-65	
CHAPTER-11		IMPLEMENTING FILE SYSTEMS	65-81	

UNIT-5

MEMORY MANAGEMENT

Chapter 8: Memory Management Strategies

Objectives: * To provide a detailed description of various ways of organizing memory hardware.

* To discuss various memory-management techniques, including paging and segmentation.
* To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging.

⓪ Background

→ Program must be brought (from disk) into memory and placed within a process for it to be run.

→ Main memory and registers are only storage CPU can access directly.

→ Memory unit only sees a stream of addresses + read requests, (or) address + data and write requests

→ Register access in one CPU clock (or less).

→ Main memory can take many cycles.

→ Cache sits between main memory and CPU registers

→ Protection of memory required to ensure correct operation.

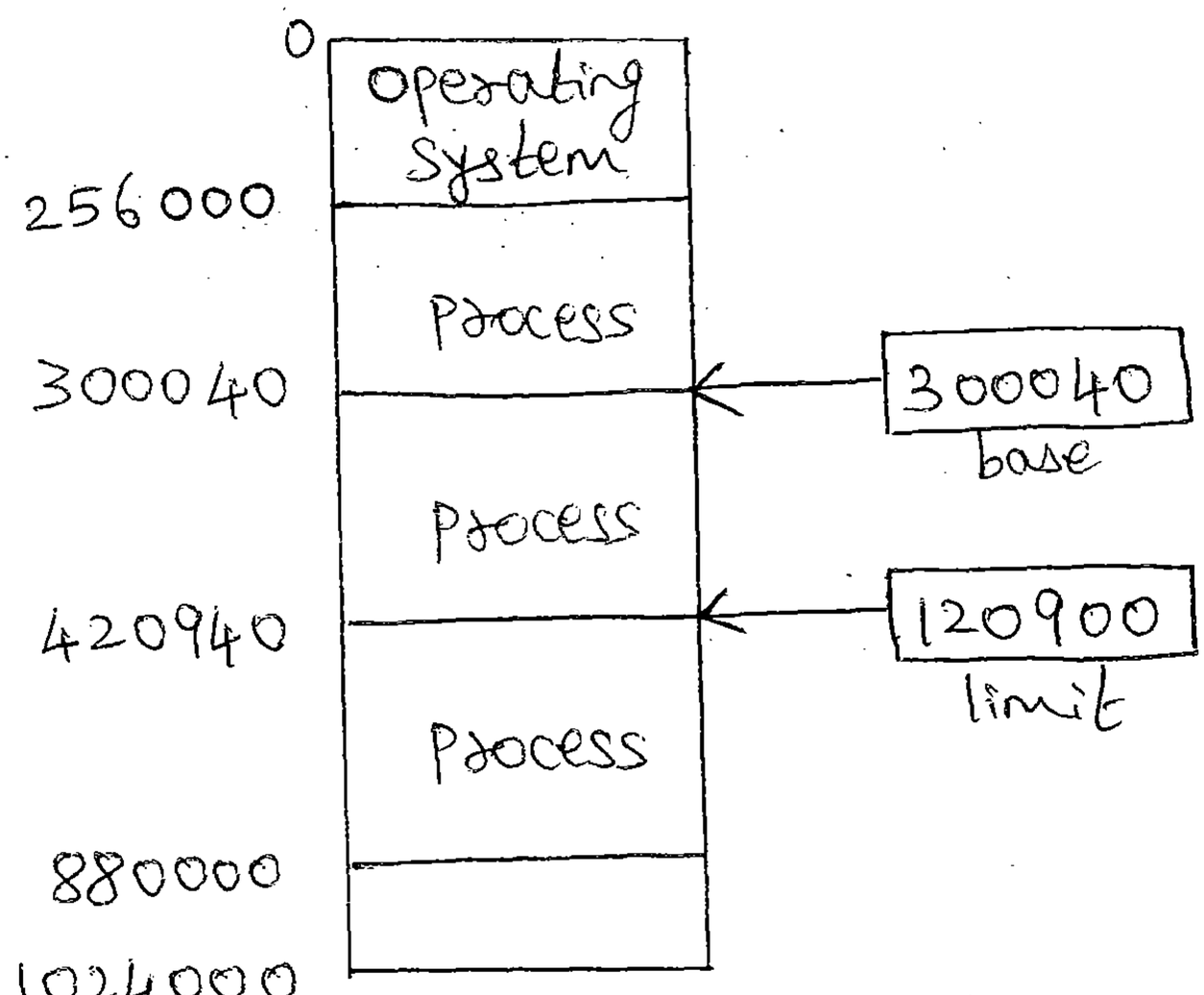
Basic Hardware

→ The base register holds the smallest legal physical memory address.

→ The limit register specifies the size of the range.

→ A pair of base and limit registers define the logical address space.

→ for example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939.



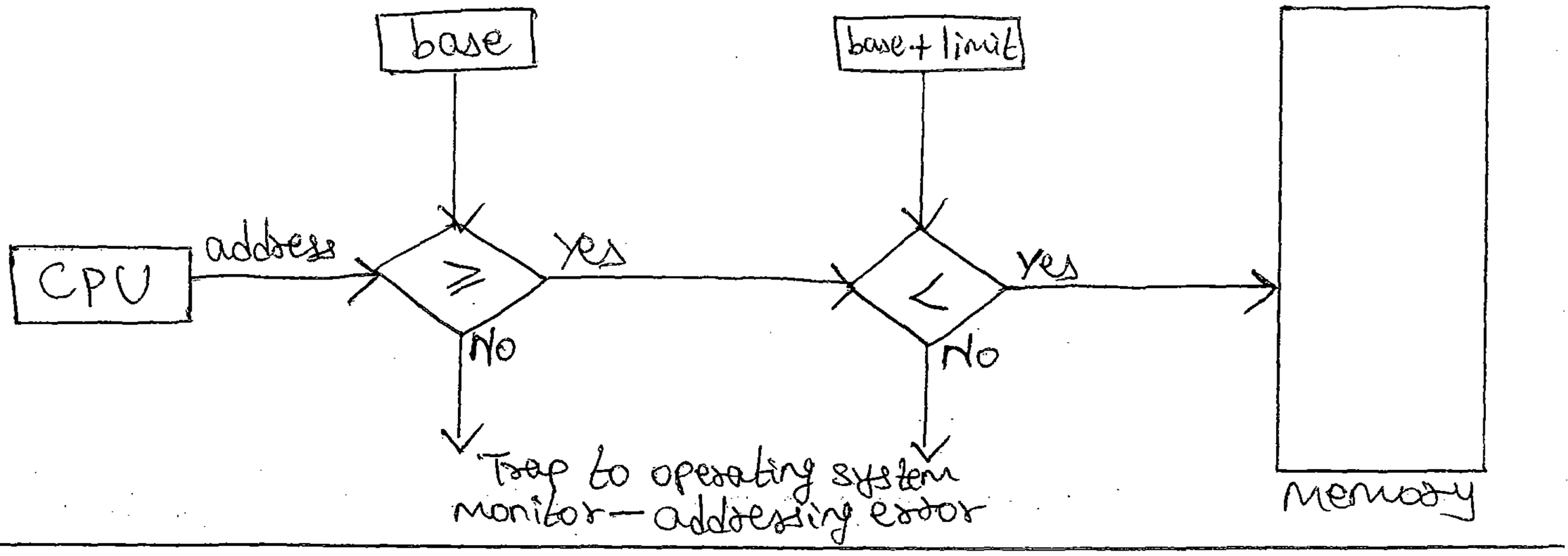
A base and a limit register define a logical address space

→ Protection of memory space is accomplished by having the CPU hardware, compare every address generated in user mode with the registers.

→ The base and limit registers can be loaded only by the OS, which uses a special privileged instruction.

→ The OS, executing in kernel mode, is given unrestricted access to both OS memory and users' memory.

→ Hardware address protection with base and limit registers



Address Binding

→ Usually, a 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.

→ The address space of the computer starts at 00000, the first address of the user process need not be 0000

→ This approach affects the addresses that the user progra can use.

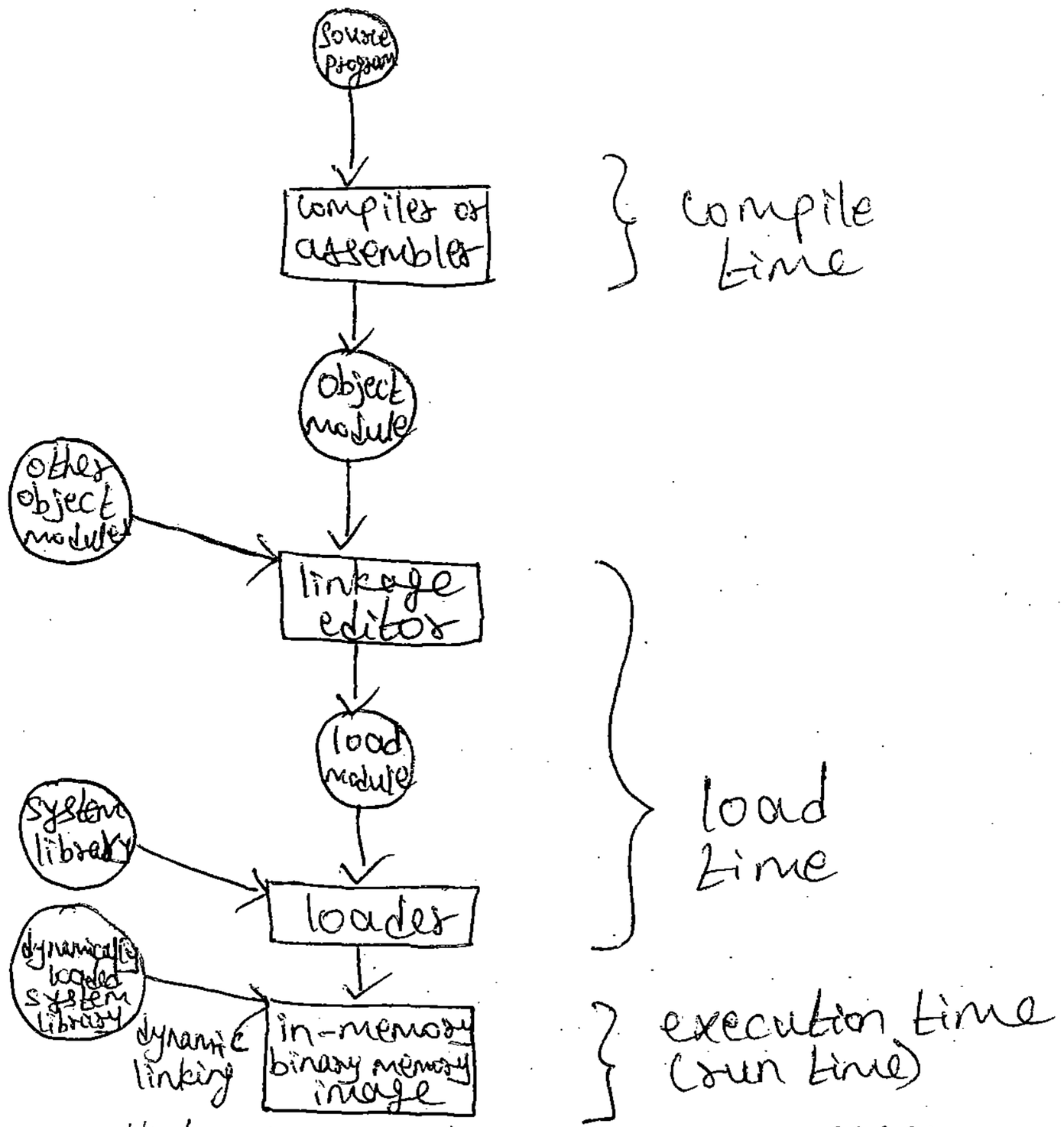
→ Addresses in the source program are generally symbolic (such as count).

→ A compiler will typically bind these symbolic addresses to relocatable addresses.

  * i.e., 14 bytes from the beginning of this module.

→ The linkage editor (or) loader will bind relocatable addresses to absolute addresses. (such as 74014).

→ Each binding maps one address space to another.



multistep processing of a user program

→ The binding of instructions and data to memory addresses can be done at any step along the way:

* Compile Time - If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

* Load Time - Must generate relocatable code if memory location is not known at compile time.

* 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.
(Ex:- base and limit registers).

Logical versus physical Address space.

→ An address generated by the CPU is commonly referred to as a logical address.

→ An address seen by the memory unit - that is, the one loaded into the memory-address registes of the memory is commonly refered to as a physical address.

→ logical address also refessed to as virtual address.

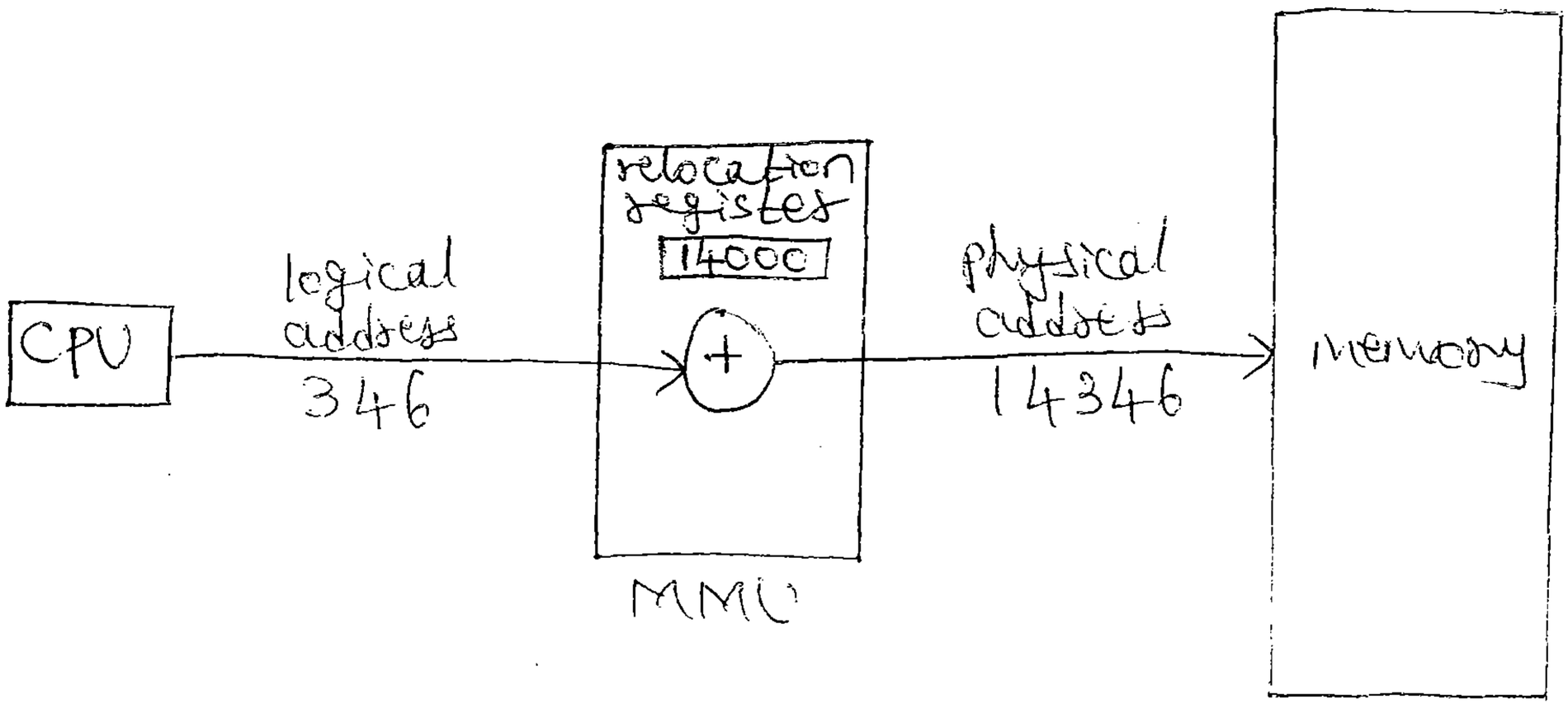
→ logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

→ Logical address space is the set of all logical addresse generated by a program.

→ Physical address space is the set of all physical addresse generated by a program.

→ The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-managemen unit (MMU).

→ Dynamic relocation using a relocation register



→ Consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

* Base registes now called relocation registes
* MS-DOS on Intel 80x86 wed 4 relocation registes.

→ The user program deals with logical addresses; it never sees the real physical addresses.

* Execution-time binding occurs when reference is made to location in memory.
* logical address bound to physical addresses.

Dynamic Loading

→ For a process to be executed it should be loaded into the physical memory. The size of the process is limited to the size of the physical memory.

→ Dynamic loading is used to obtain better memory utilization

→ Here, a routine is not loaded until it is called.

→ Whenever a routine is called, the calling routine first checks whether the called routine is already loaded (or) not.

→ If it is not loaded it cause the loader to load the desired program into the memory and updates the program's address tables to indicate the change and control is passed to newly loaded routine.

→ The advantage of dynamic loading is that an unused routine is never loaded.

→ Does not require special support from the OS.

→ OS can help by providing library routines to implement dynamic loading

Dynamic Linking and Shared Libraries.

→ static Linking - system language libraries and program code combined by the loader into the binary program image.

→ Dynamic Linking - Linking postponed until execution time.

→ small piece of code, stub used to locate the appropriate memory-resident library routine.

→ stub replaces itself with the address of the routine, and executes the routine.

→ OS checks if routine is in processes' memory address.

→ Dynamic linking is particularly useful for libraries

→ system also known as shared libraries.

→ Consides applicability to patching system libraries.

* Versioning may be needed.

Swapping

→ A process must be in memory to be executed.

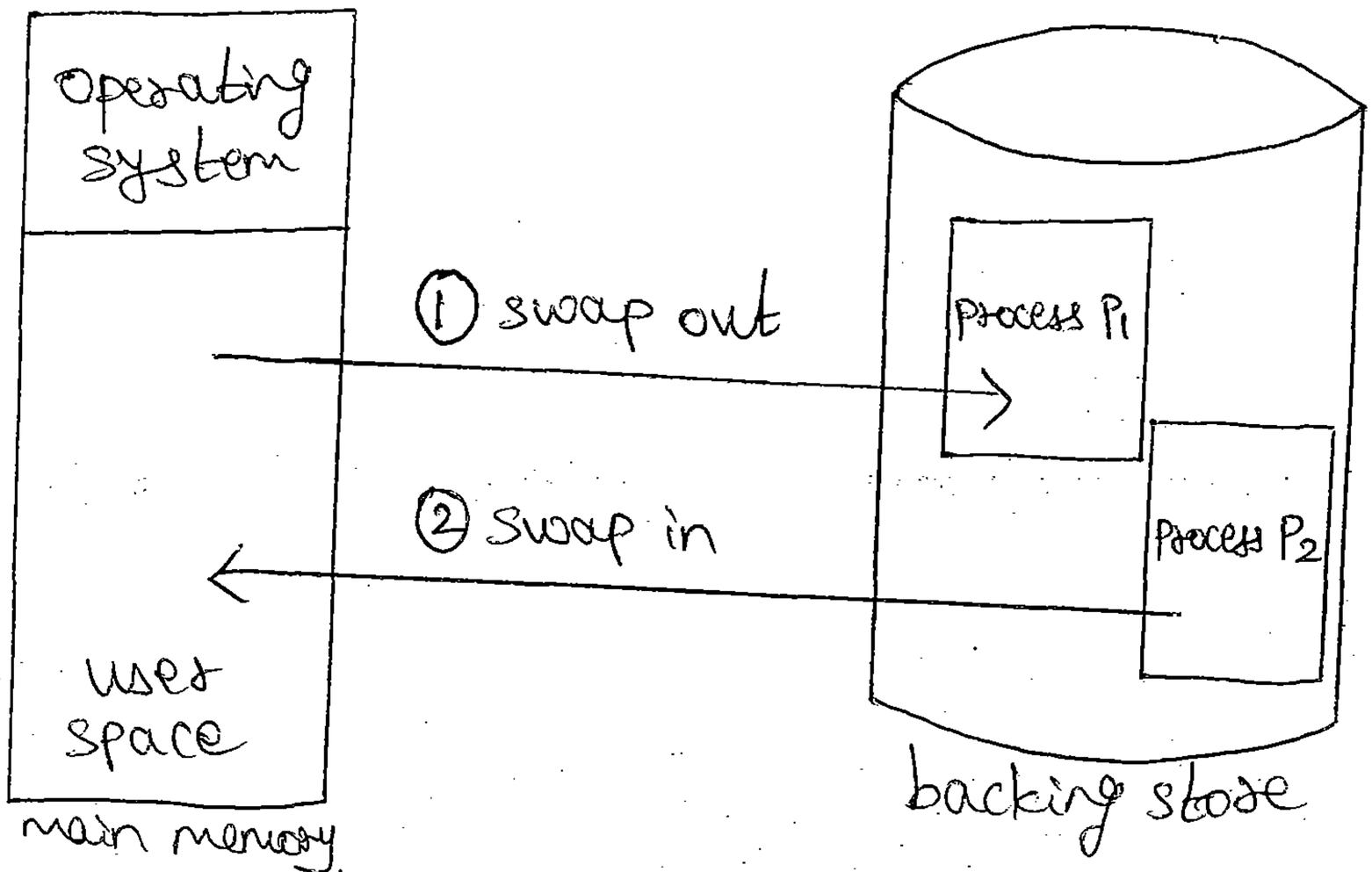
→ A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for Continued execution.

* Total physical memory space of processes can exceed physical memory.

→ Backing store - fast disk large enough to accomodate copies of all memory images for all ~~memory~~ users; ~~images~~ 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

→ Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.



swapping of two processes using a disk as a backing store

→ System maintains a ready queue of ready-to-run processes which have memory images on disk.

→ Does the swapped out process need to swap back in to same physical addresses?

* Depends on address binding method
  - ⇒ Plus consider pending I/O to/from process memory space.

→ Modified versions of swapping are found on many systems. (i.e, UNIX, Linux and Windows).

* Swapping normally disabled.
* Started if more than threshold amount of memory allocated.
* Disabled again once memory demand reduced below threshold.

→ Context Switch Time including Swapping.

* If next processes to be put on CPU is not in memory need to swap out a process and swap in target process
* Context switch time can be very high.
* 100MB process swapping to hard disk with transfer rate of 50 MB per second [Actual transfer of 100 MB process to (or) from main memory takes
  - » plus disk latency of 8ms. 100MB/50MB/second = 2 seconds.
  - » swap out time of 2008ms.
  - » plus swap in of same sized process.
  - » Total context switch swapping component time of 4016 ms (> 4 seconds).
* Can reduce if reduce size of memory swapped - by knowing how much memory really being used.
  - » system calls to inform OS of memory use via request memory and release memory.

---

⑪ Contiguous Memory Allocation.

→ The main memory must accommodate both the OS and the various user processes.

→ therefore, need to allocate main memory in the most efficient way possible

→ Main memory usually divided into two partitions:
  - * resident OS, usually held in low memory with interrupt vector.
  - * User processes then held in high memory.

→ Here, each process contained in a single contiguous section of memory.

Memory Mapping and Protection

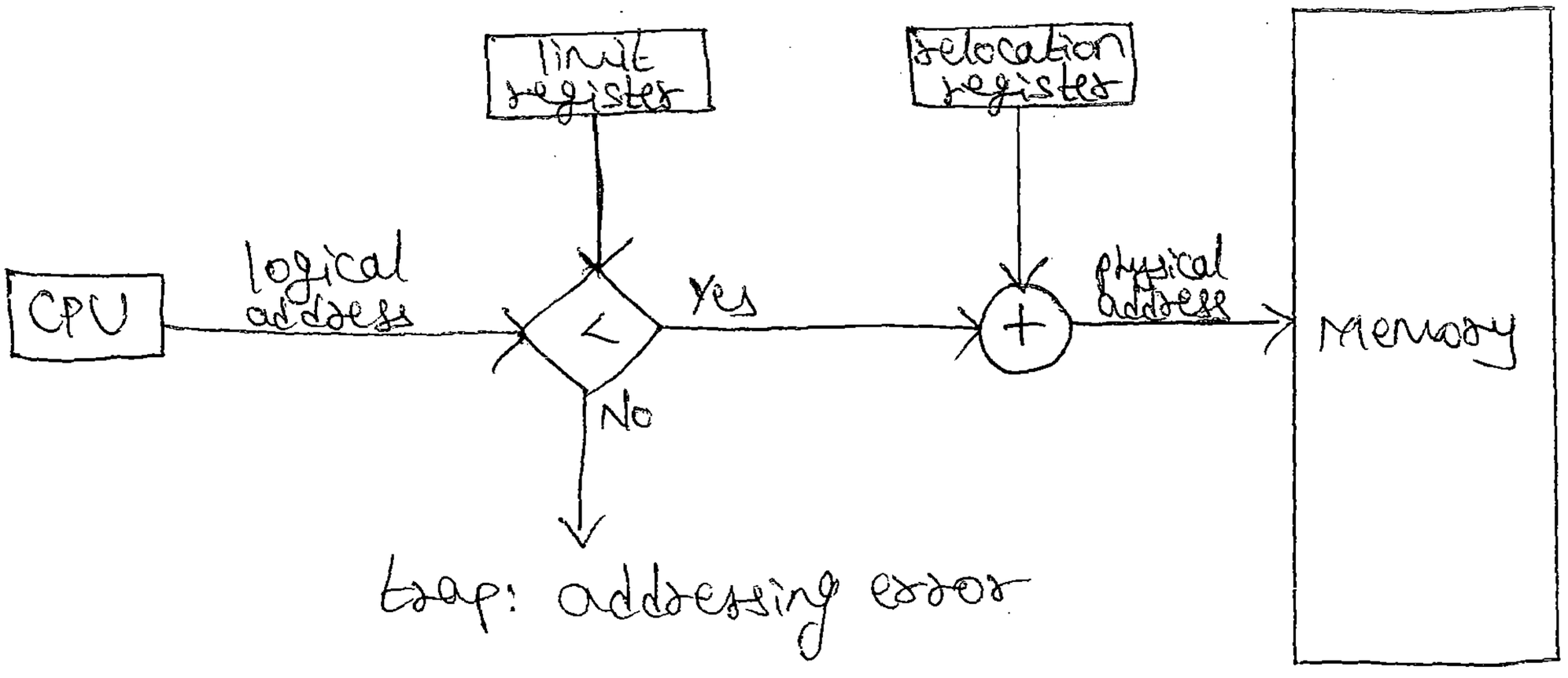
→ Relocation registers used to protect user processes from each other, and from changing OS code and data.

* The relocation register contains the value of the smallest physical address.

* The limit register contains the range of logical addresses (For ex:- relocation = 100040 & limit = 74600), each logical address must be less than the limit register.

* MMU maps logical address dynamically by adding the value in the relocation register.

* Can then allow actions such as kernel code being transient and kernel changing size



Hardware support for relocation and limit registers

Memory Allocation

→ One of the simplest methods for allocating memory is to divide memory into several fixed-size partitions.

→ Each partition may contain exactly one process.

→ Degree of multiprogramming is bound by the number of partitions.

→ In 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. — This method was originally used by the IBM OS/360 operating system (called MFT).

→ In variable-partition scheme, the OS 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 as one large block of available memory, a hole. Holes of various size are scattered throughout memory.

→ when a process arrives, it is allocated memory from a hole that is large enough to accommodate it.

→ Process exiting frees its partition, adjacent free partitions combined.

→ OS maintains information about:
  a) Allocated partitions.
  b) free partitions (hole)

→ Dynamic storage-allocation problem

* which concerns, how to satisfy a request of size 'n' from a list of free holes?

  » First-fit: Allocate the first hole that is big enough.
  » Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size.
    — Produces the smallest leftover hole.

» Worst-fit: Allocate the largest hole; must also search entire list, unless it is sorted by size.
- produces the largest leftover hole.

Fragmentation

→ Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.

→ External Fragmentation - Total memory space exists to satisfy a request, but it is not contiguous.

→ Internal Fragmentation - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

→ First fit analysis reveals that given 'N' blocks allocated, 0.5N blocks lost to fragmentation.

* i.e, one-third of memory may be unusable. This property is known as the 50-percent rule.

→ One solution to the problem of external fragmentation is compaction.

* shuffle memory contents to place all free memory together in one large block.

* Compaction is possible only if relocation is dynamic, and is done at execution time.

* I/O problem
» Latch job in memory while it is involved in I/O.
» Do I/O only into OS buffers.

→ Two complementary techniques - Paging and segmentation.

Paging

→ Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguou

→ Paging avoids external fragmentation and need for compactio

---

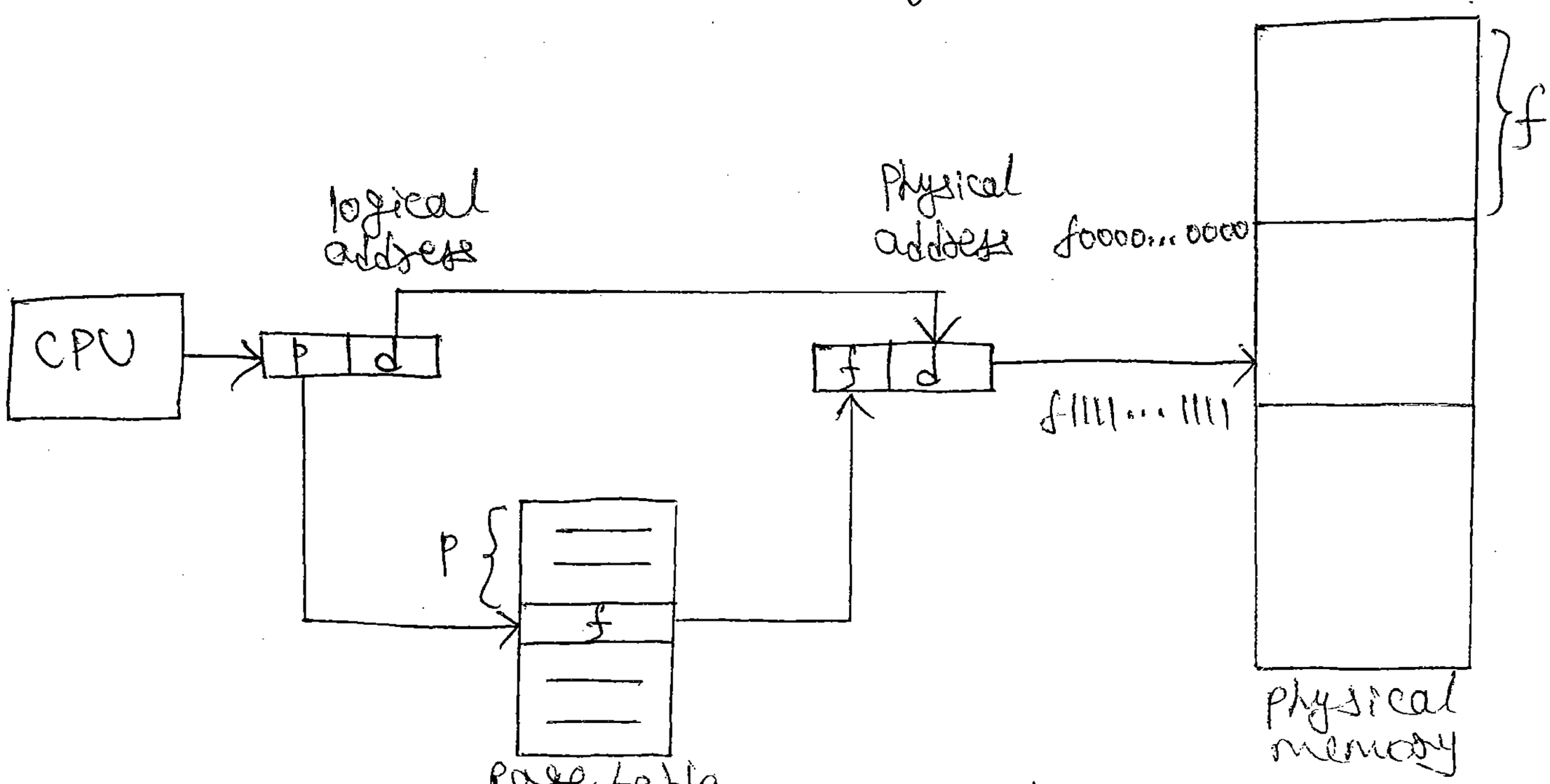
Basic Method

→ The basic method for implementing paging involves breaking physical memory into fixed-size blocks called frames.

→ Breaking logical memory into blocks of the same size called pages.

→ 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 below:



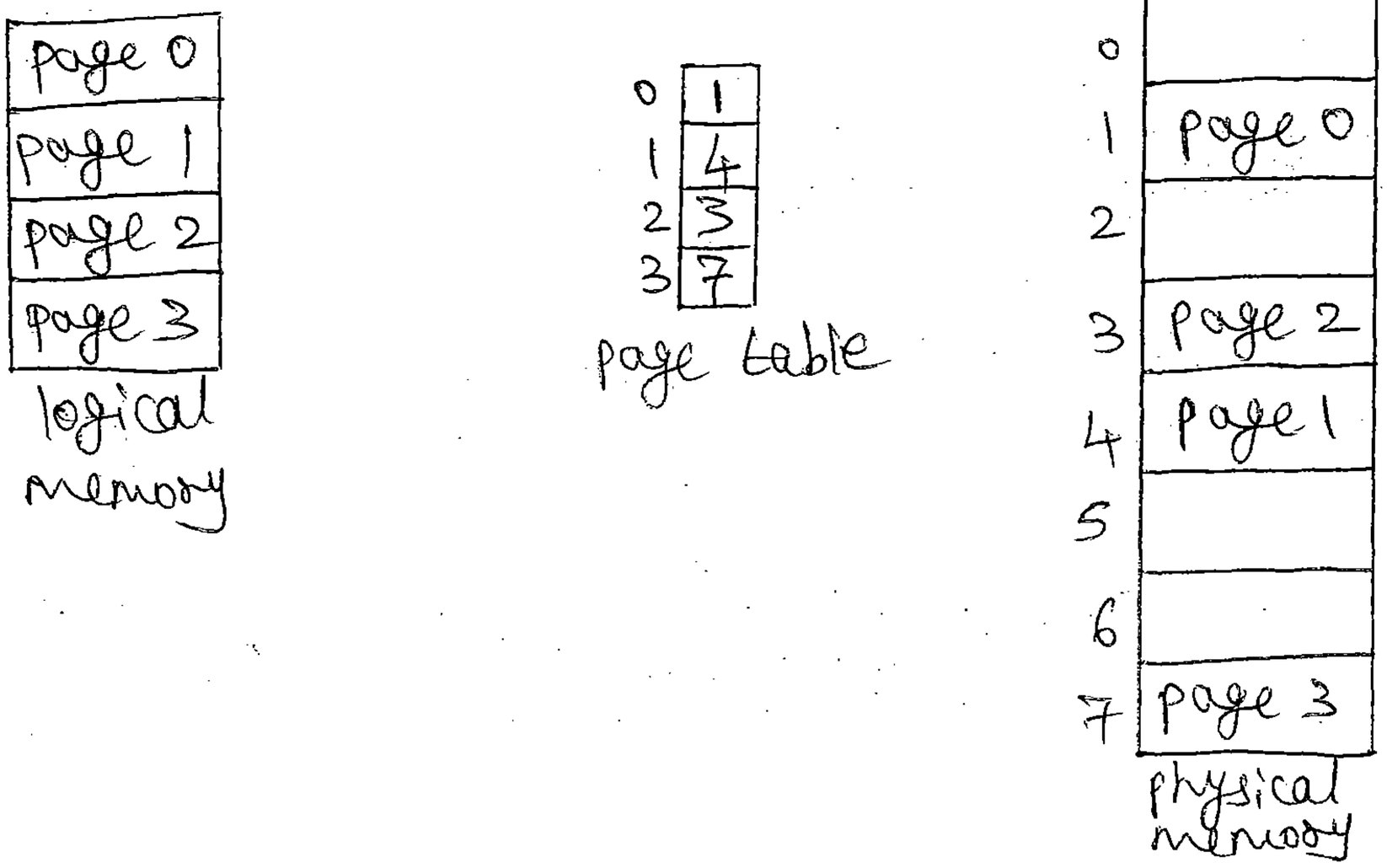
paging Hardware

→ Every address generated by the CPU is divided into two parts:

(i) Page number (p) : Used as an index into a page table which contains base address of each page in physical memo

(ii) Page offset (d) - combined with base address to define the physical memory address that is sent to the memory unit.

→ Paging Model of logical and physical memory

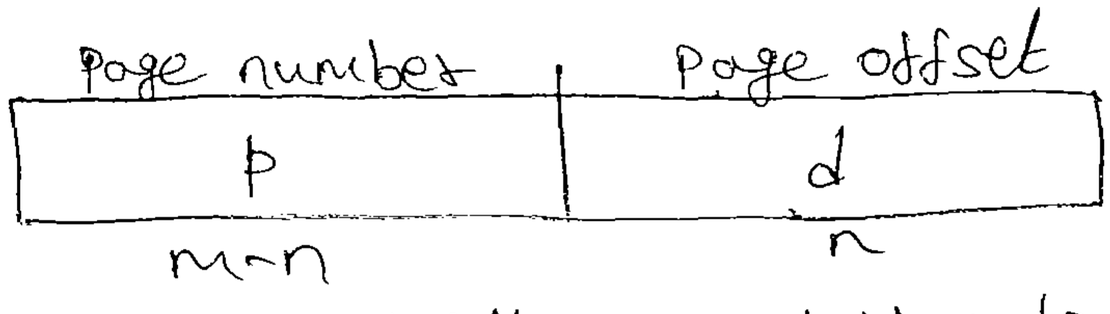


→ The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.

→ The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

→ If the size of the logical address space is 2^m and a page size is 2^n addressing units (bytes/words), then the high-order 'm-n' bits of a logical address designate the page number.

* the 'n' low-order bits designate the page offset.
* The logical address is as follows:



where p = an index into the page table ; d = displacement within

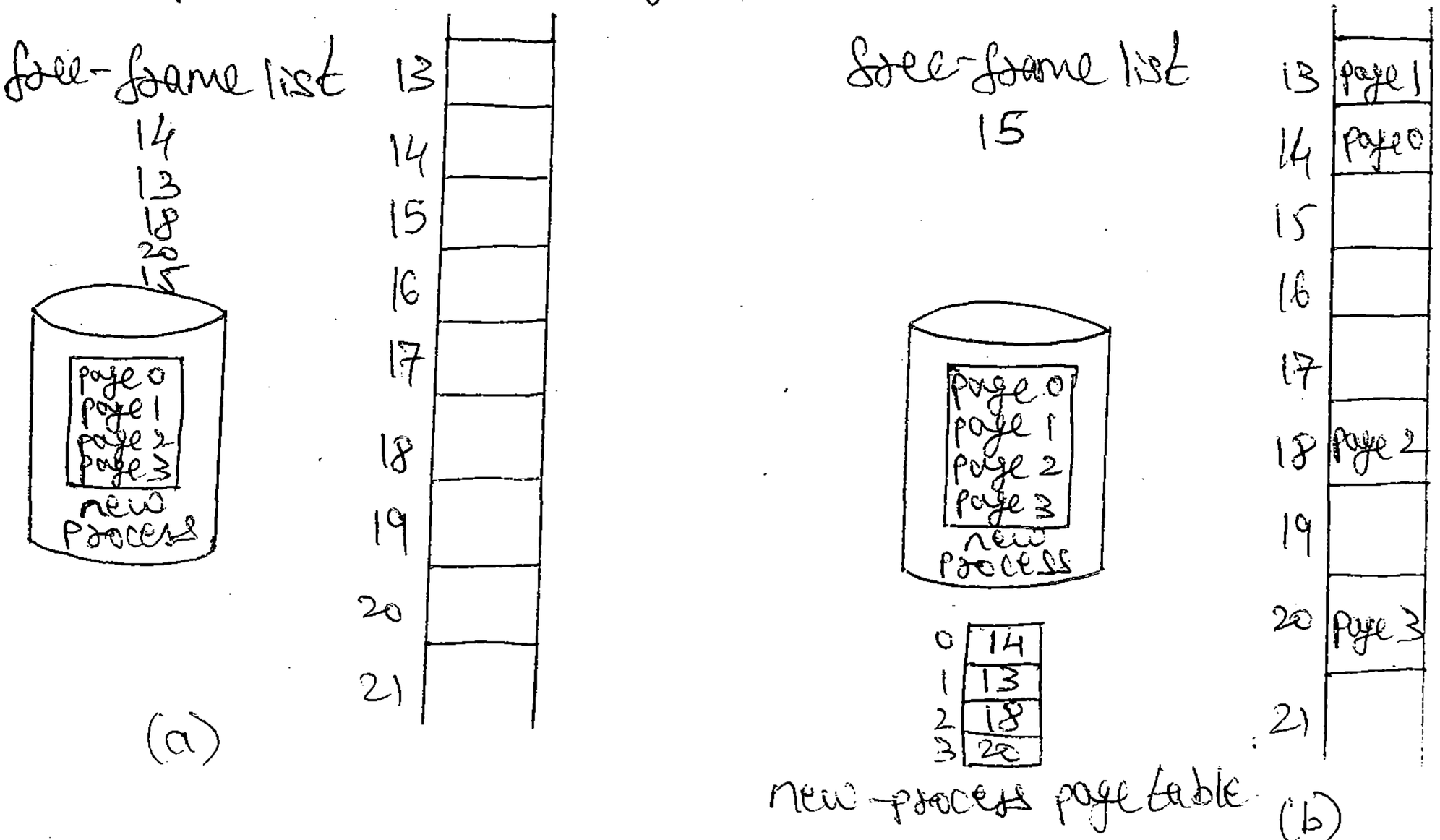
→ Calculating internal fragmentation

* 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.
* On average fragmentation = 1/2 frame size.
* Each page table entry takes memory to track.
* Page sizes growing over time.
  - » Solaris supports two page sizes: 8 KB & 4 MB.

→ When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame.

→ If the process requires 'n' pages, at least 'n' frames must be available in memory. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.

→ The next page is loaded into another frame, and its frame number is put into the page table and so on.



Free frames (a) before allocation and (b) after allocation

→ An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory.

→ Os managing physical memory, must be aware of the allocation details of physical memory; which frames are allocated, which frames are available, how many total frames are these and so on.

— This information is generally kept in a data structure called a Frame Table.

→ The frame table has one entry for each physical page frame, indicating whether the latter is free (or) allocated and, if it is allocated, to which page of which process (or) processes.

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.

→ These registers should be built with very high-speed logic to make the paging-address translation efficient.

→ The page table is kept in main memory and a page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

→ The problem with this approach is, the time required to access a user memory location.

→ The standard solution to this problem is to use a special, small, fast-lookup hardware cache, called a Translation

→ The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.

→ When the associative memory is presented with an item, the item is compared with all keys simultaneously.

If the item is found, the corresponding value field is returned. The search is fast, the hardware, however is expensive.

→ The number of entries in a TLB is small, often numbering between 64 and 1,024.

→ The TLB is used with page tables in the following way:

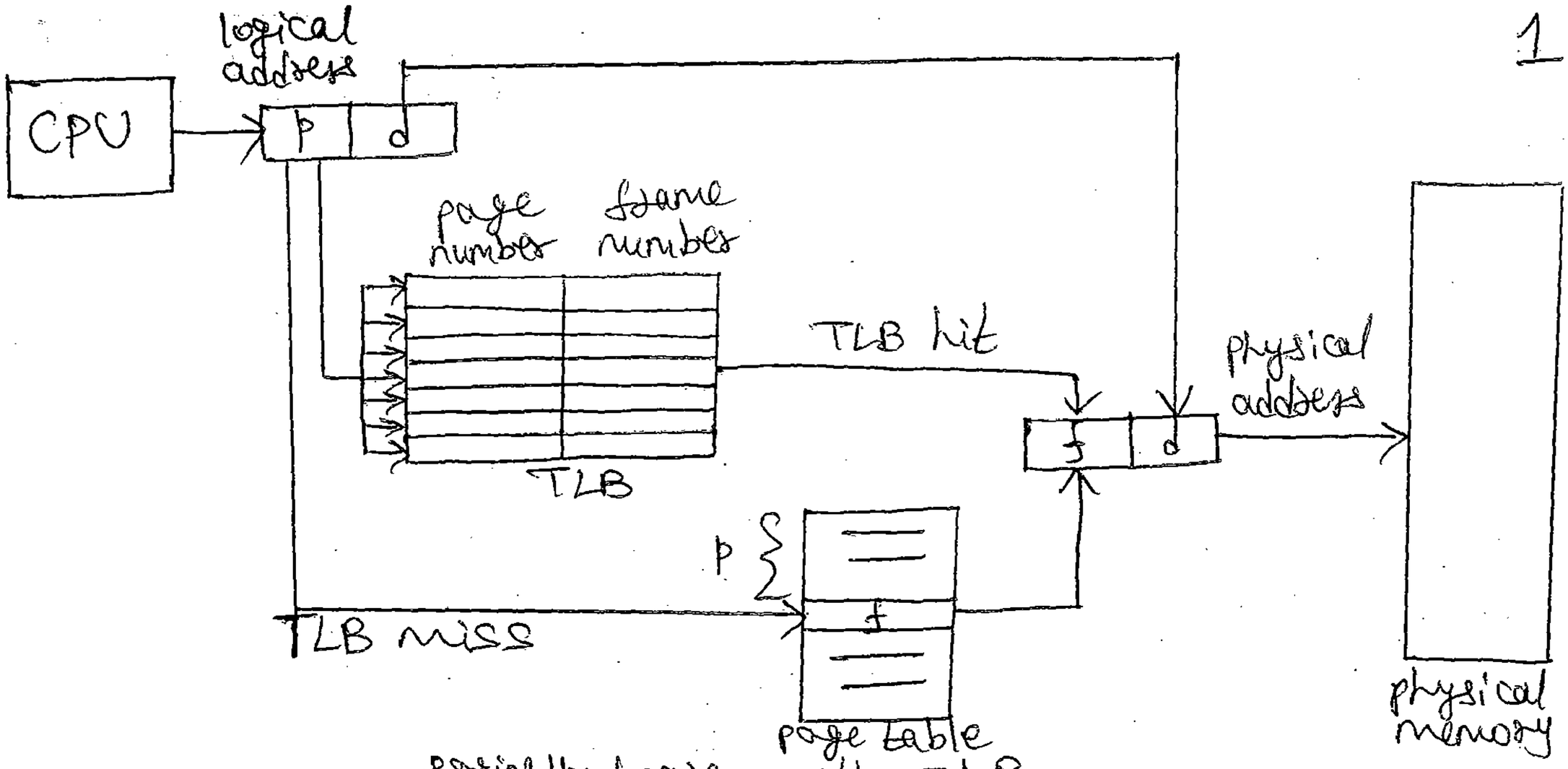
* The TLB contains only a few of the page-table entries
* When a logical address is generated by the CPU, its page number is presented to the TLB.
* If the page number is found, its frame number is immediately available and is used to access memory.
* The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

→ If the page number is not in TLB (known as a TLB miss) a memory reference to the page table must be made.

→ When the frame number is obtained, we can use it to access memory.

→ If the TLB is already full of entries, the OS must select one for replacement. Replacement policies range from least recently used (LRU) to random.

→ Some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.



Paging Hardware with TLB

→ Some TLBs store address-space identifiers (ASIDs) in each TLB entry.

→ An ASID uniquely identifies each process and is used to provide address-space protection for that process.

→ If the TLB does not support separate ASIDs, then every time a new page table is selected, the TLB must be flushed (or erased) to ensure that the next executing process does not use the wrong translation information.

→ The percentage of times that a particular page number is found in the TLB is called the hit ratio.

→ An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.

→ To find the effective memory-access time, we weight each case by its probability:

effective access time = 0.80 x 120 + 0.20 x 220
= 140 nanoseconds.

→ For 98 percent hit ratio,

effective access time = 0.98 x 120 + 0.02 x 220
= 122 nanoseconds.

protection

→ Memory protection in a paged environment is accomplished by protection bits associated with each frame.

→ One additional bit is generally attached to each entry in the page table: a valid-invalid bit.

→ when this bit is set to "valid", the associated page is in the process's logical address space and is thus a legal (or valid) page.

→ when the bit is set to "invalid", the page is not in the process's logical address space.

→ some systems provide hardware, in the form of a page-table length register (PTLR), to indicate the size of the page table.

---

shared pages.

→ An advantage of paging is the possibility of sharing common code.

→ This consideration is particularly important in a time-sharing environment.

→ Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 kB of code and 50 kB of data space, we need 8,000 kB to support the 40 users.

→ If the code is reentrant code (or pure code), however, it can be shared.

→ Reentrant code is non-self-modifying code; it never changes during execution.

→ Thus, two (or) more processes can execute the same code at the same time.

⑩ Structure of the page table

Hierarchical paging

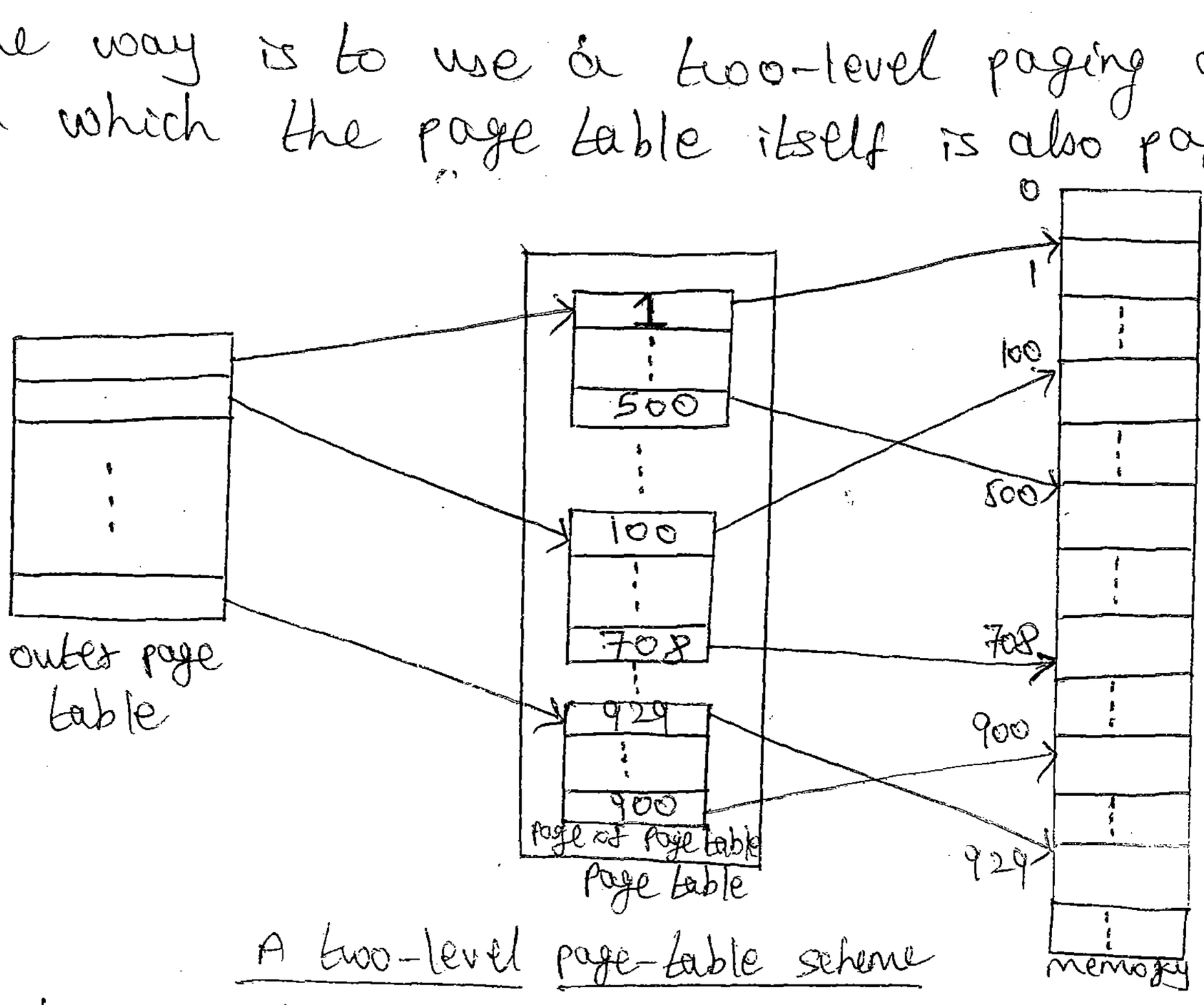
→ Most modern computer systems support a large logical address space (2^{32} to 2^{64}).

→ With this, page table itself becomes excessively large.

→ For example, consider a system with 32-bit logical address space. If the page size in such a system is 4KB (2^{12}), then a page table may consist of upto 1 million entries ($2^{32}/2^{12}$).

- Assuming that each entry consists of 4 bytes, each process may need upto 4MB of physical address space for the page table alone.

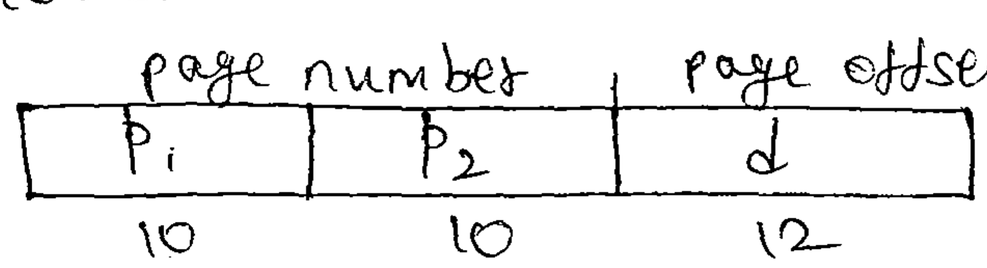
→ One way is to use a two-level paging algorithm; in which the page table itself is also paged.



A two-level page-table scheme

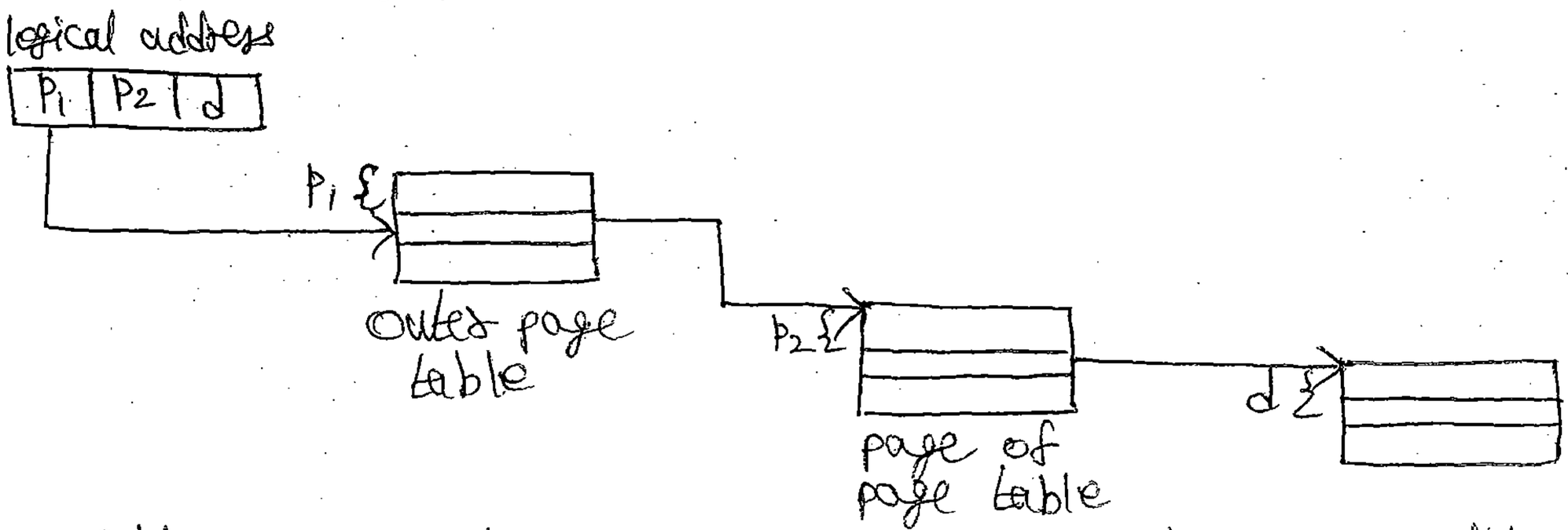
→ w.r.t 32-bit machine with a page size of 4KB. A logical address is divided into a page number consisting of 20 bits & a page offset consisting of 12 bits.

→ The page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where p_1 = An index into the outer page table.
p_2 = displacement within the page of the outer page table.

→ The address-translation method for this architecture is shown below:

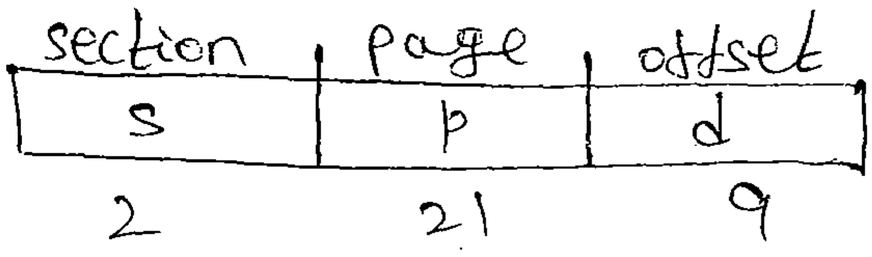


Address translation for a two-level 32-bit paging architecture

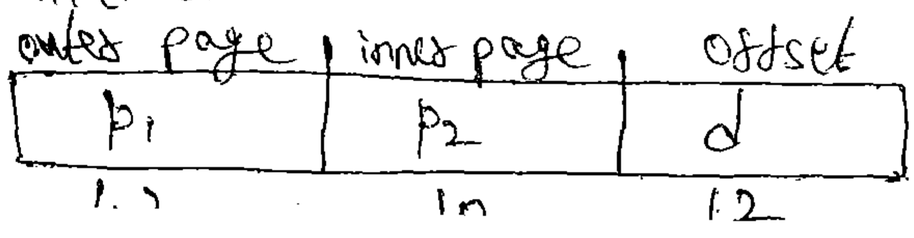
→ Address translation works from the outer page table inward, this scheme is also known as forward-mapped page table.

→ The VAX architecture also supports a variation of two-level paging. The VAX is a 32-bit machine with a page size of 512 bytes.

⇒ An address on the VAX architecture is as follows:



⇒ For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. The addresses look like this:



→ The outer page table consists of 2^{42} entries, or 2^{44} bytes.

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

→ The outer page table is still 2^{34} bytes in size.

→ The SPARC architecture (with 32-bit addressing) supports a three-level paging scheme.

→ 32-bit Motorola 68030 architecture supports four-level paging scheme.

→ 64-bit UltraSPARC would require seven levels of paging.

Hashed page Tables

→ A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.

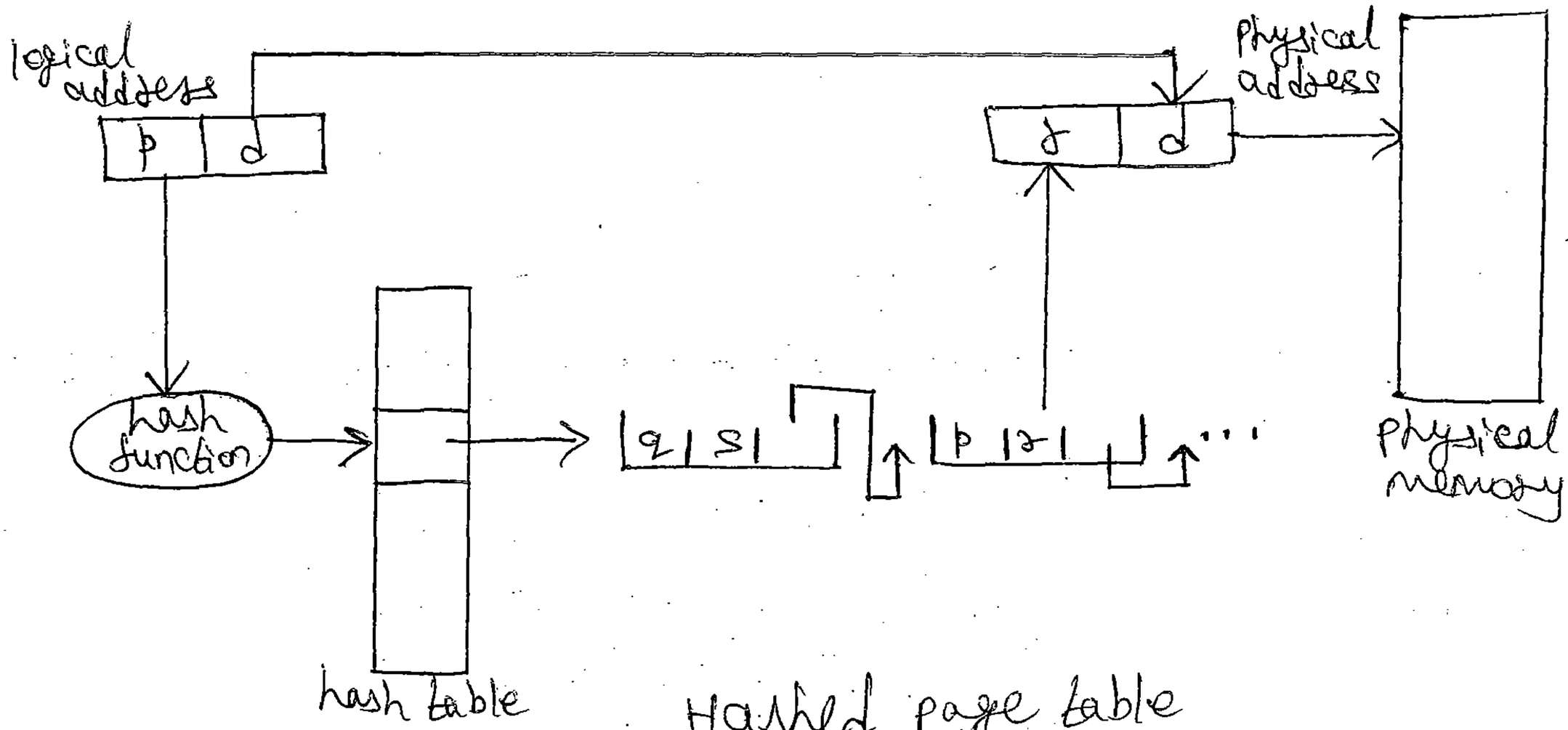
→ Each entry in the hash table contains a linked list of elements that hash to the same location.

→ Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

→ The Algorithm works as follows:

* The virtual page number in the virtual address is hashed into the hash table
* The virtual page number is 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 a matching virtual page number.
* The scheme is shown below.



Hashed page table

→ A variation of this scheme that is favorable for 64-bit address spaces has been proposed.
→ This variation uses clustered page tables, which are similar to hashed page tables except that each entry in the hash table refer to several pages (such as 16) rather than a single page.
→ Therefore, a single page-table entry can store the mappings for multiple physical-page frames.
→ clustered page tables are particularly useful for sparse address space, where memory references are non-contiguous and scattered throughout the address space.

Inverted page Tables

→ 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 process reference pages through the pages' virtual addresses.

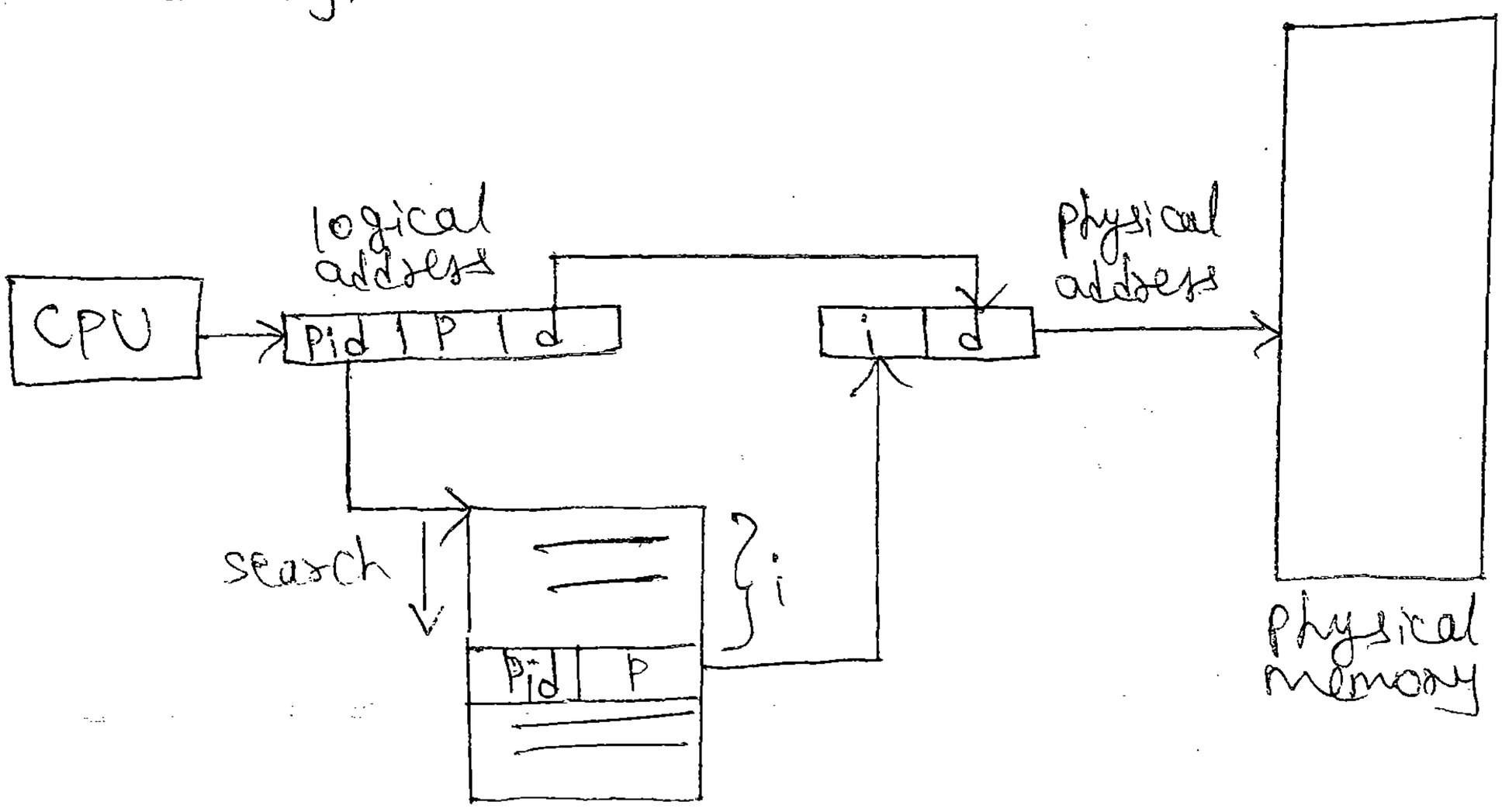
→ The OS must then translate this reference into a physical memory address.

→ 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.

* It 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 that page.
* Thus, only one page table is in the system, and it has only one entry for each page of physical memory.



Inverted page table

→ To illustrate this method, we describe a simplified version of the inverted page table used in the IBM RT.

* Each virtual address in the system consists of a triple:
  <process-id, page-number, offset>.
* Each inverted page-table entry is a pair:
  <process-id, page-number>.
  where, process-id assumes the role of the address-space identifier.
* when a memory reference occurs, part of the virtual address, consisting of <process-id, page-number> 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 <i, offset> is generated.
  >> If no match is found, then an illegal address ~~access~~ access has been attempted.

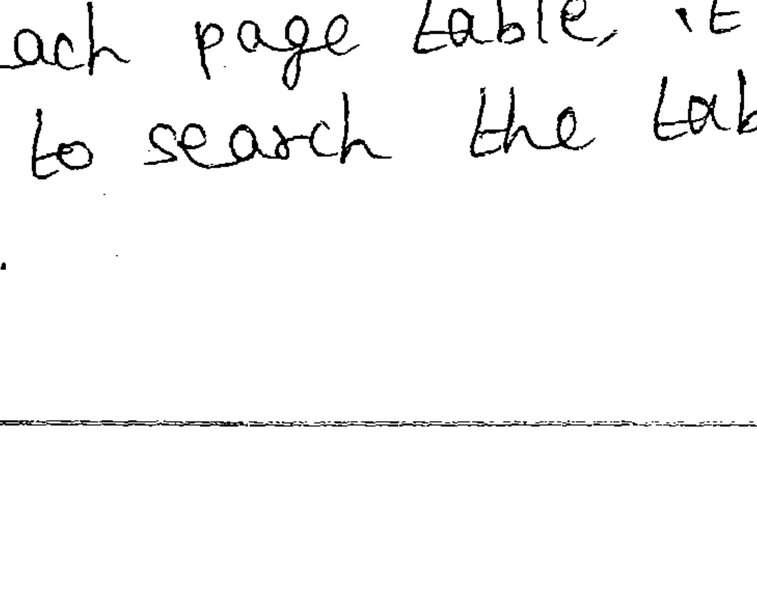
→ Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs.

---

(11) segmentation

Basic Method

→ Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments.



logical address

→ Segmentation is a memory-management scheme that supports this user view of memory.

→ A logical address space is a collection of segments.

→ Each segment has a name and a length

→ the addresses specify both the segment name and the offset within the segment.

→ Therefore, user specifies each address by two quantities: a segment name and an offset.

→ for simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.

→ Thus, a logical address consists of a two tuple:

<segment-number, offset>.

→ Usually, the user program is compiled and the compiler automatically constructs segments reflecting the input program.

* A 'C' compiler might create separate segments for the following:
  (i) The code
  (ii) Global variables
  (iii) The heap, from which memory is allocated
  (iv) The stacks used by each thread.
  (v) The standard 'C' library.

→ Libraries that are linked in during compile time might be assigned separate segments.

→ The loader would take all these segments and assign them segment numbers.

Hardware

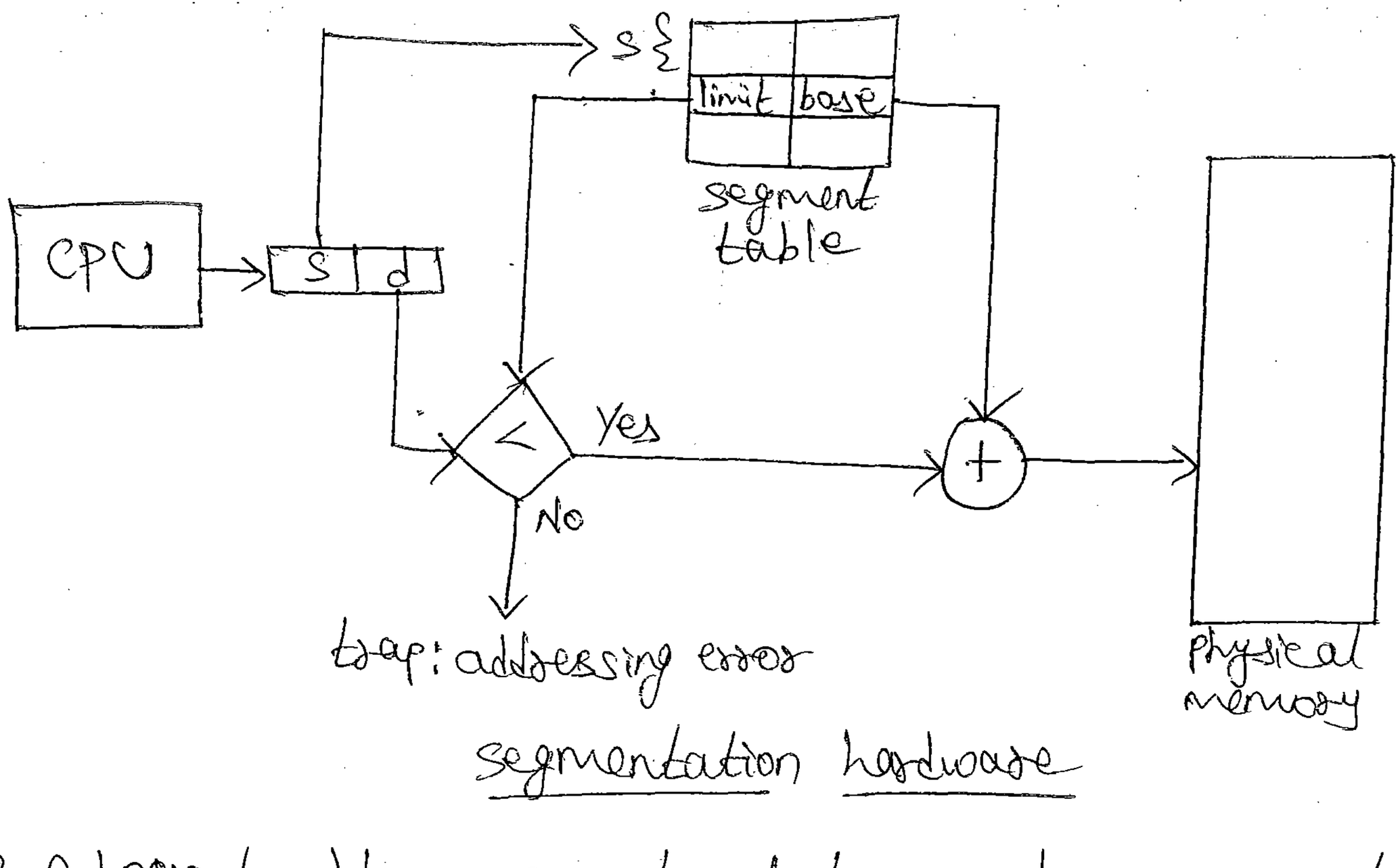
→ we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.

* This mapping is effected by a <u>segment table</u>

→ Each entry in the segment table has a <u>segment base</u> and a <u>segment limit</u>.

* The segment base contains the starting physical address where the segment resides in memory.
* The segment limit specifies the length of the segment.

→ The use of a segment table is illustrated below:



segmentation hardware

→ A logical address consists of two parts: a segment number, 's' and an offset into that segment, 'd'.

→ The segment number is used as an index to the segment table.

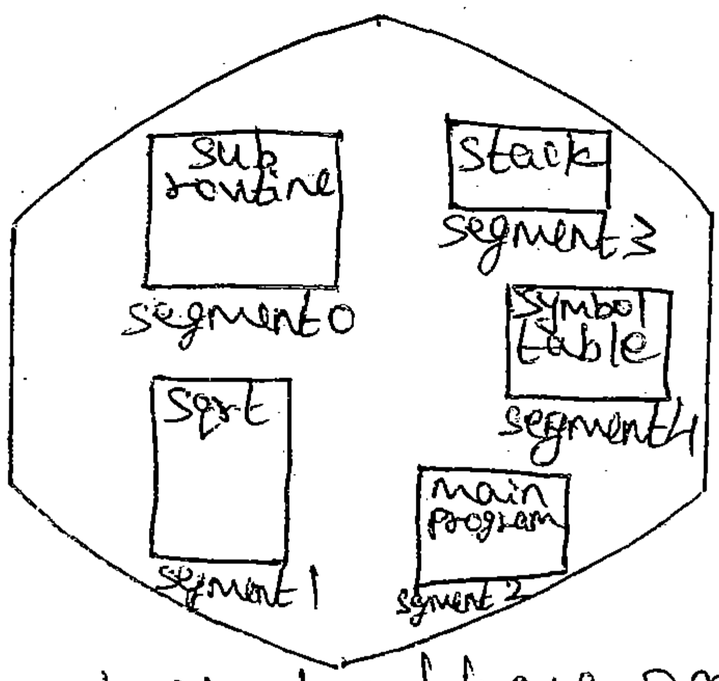
→ The offset 'd' of the logical address must be between 0 and the segment limit.

→ If it is not, we trap to the OS.

→ When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

→ The segment table is thus essentially an array of base-limit register pairs.

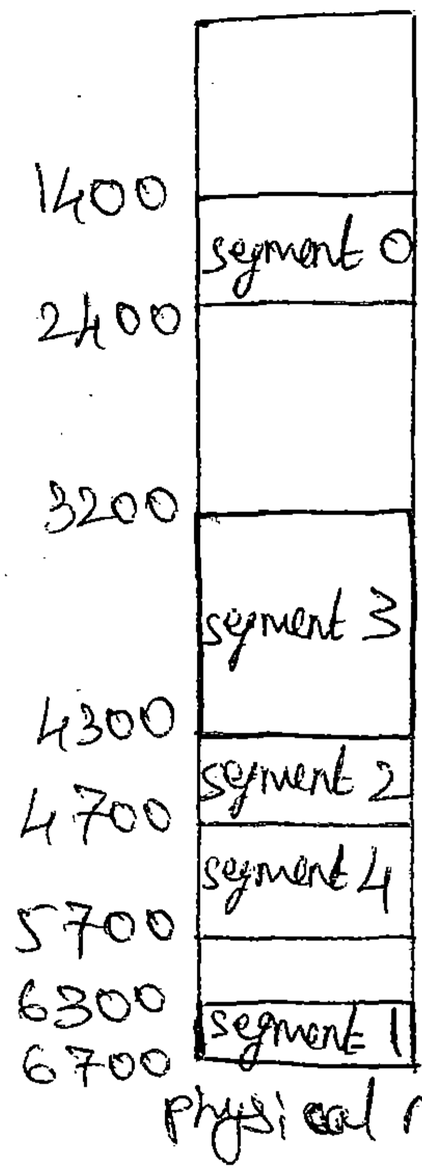
→ As an example, consider the situation shown below:



logical address space

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



physical memory

Example of segmentation

→ Five segments numbered from 0 through 4.

→ The segments are stored in physical memory.

→ The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).

→ For example, segment 2 is 400 bytes long and begins at location 4300.

→ Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.

→ A reference to segment 3, byte 1222 of segment 0 would result in a trap to the OS, as this segment is only 1,000 bytes long.

Chapter 9: Virtual-Memory Management

Objectives: * To describe the benefits of a virtual memory system.
* To explain the concepts of demand paging, page-replacement algorithms and allocation of page frames.
* To discuss the principles of working-set model.

⓪ Background

→ Virtual memory involves the separation of logical memory as perceived by users from physical memory.

→ This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

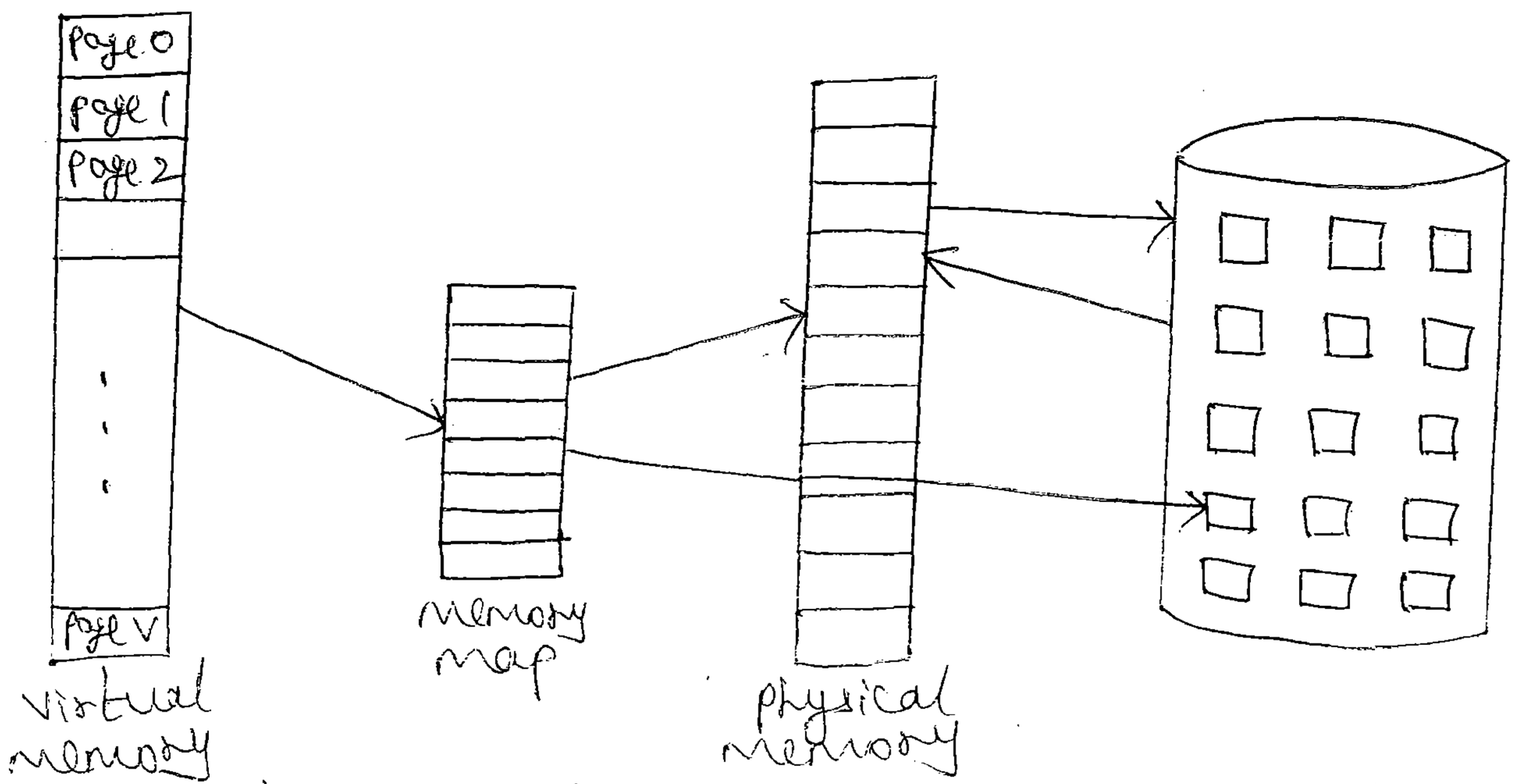
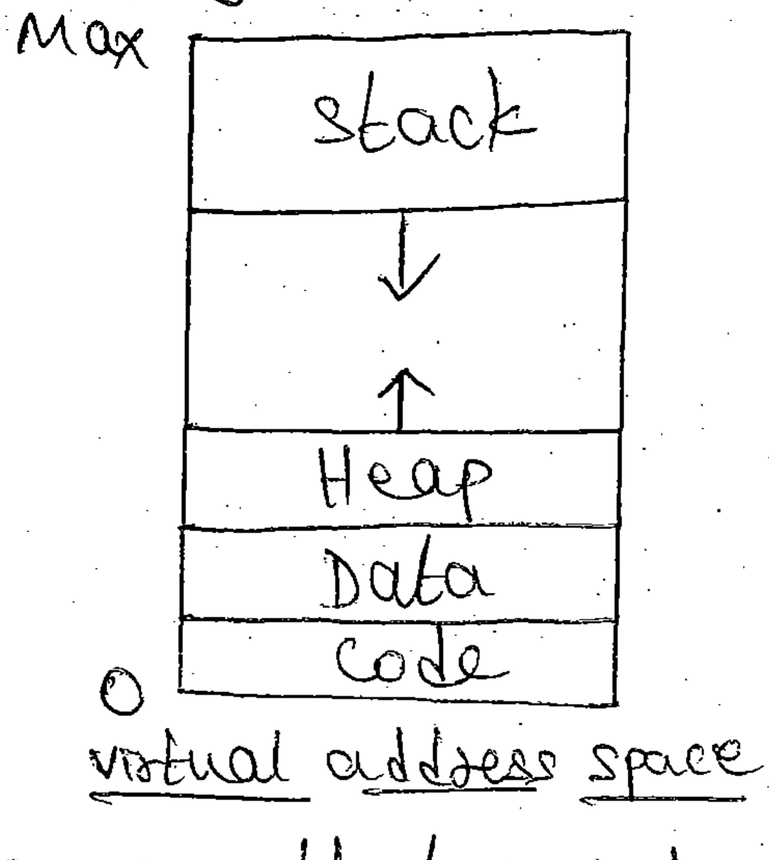


Diagram showing virtual memory that is larger than physical memory.

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

→ The <u>virtual address space</u> of a process refers to the logical (or virtual) view of how a process is stored in memory.



<u>virtual address space</u>

→ Virtual address spaces that include <u>holes</u> are known as <u>sparse</u> address spaces.

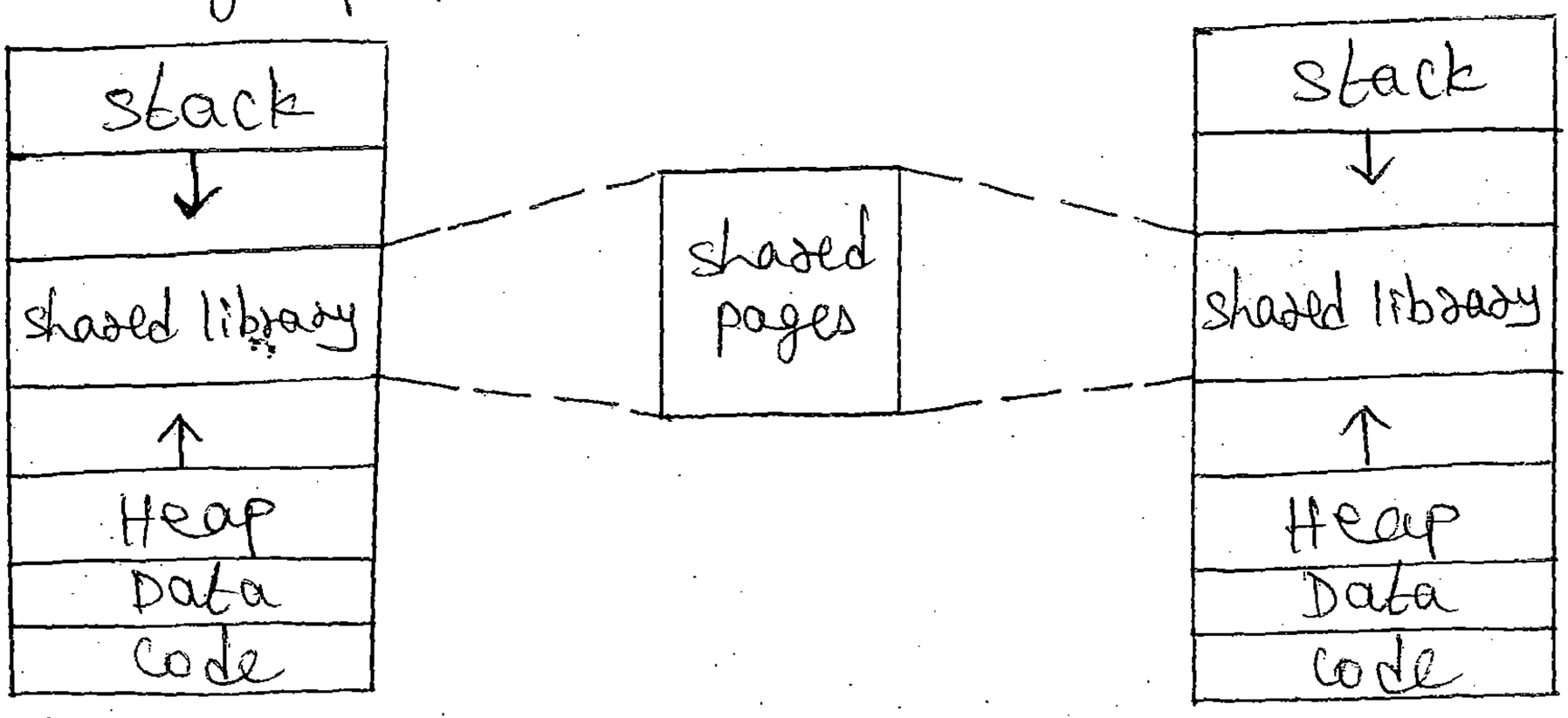
→ Using a sparse address space is beneficial because the holes can be filled as the stack (or) heap segments grow (or) if we wish to dynamically link libraries during program execution.

→ Virtual memory also allows files and memory to be shared by two (or) more processes through page sharing.

→ This leads to following benefits:

* System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
* Virtual memory enables processes to share memory.

* Allows one process to create a region of memory that it can share with another process.
* Virtual memory can allow pages to be shared during process creation with the fork() system call, thus speeding up process creation.



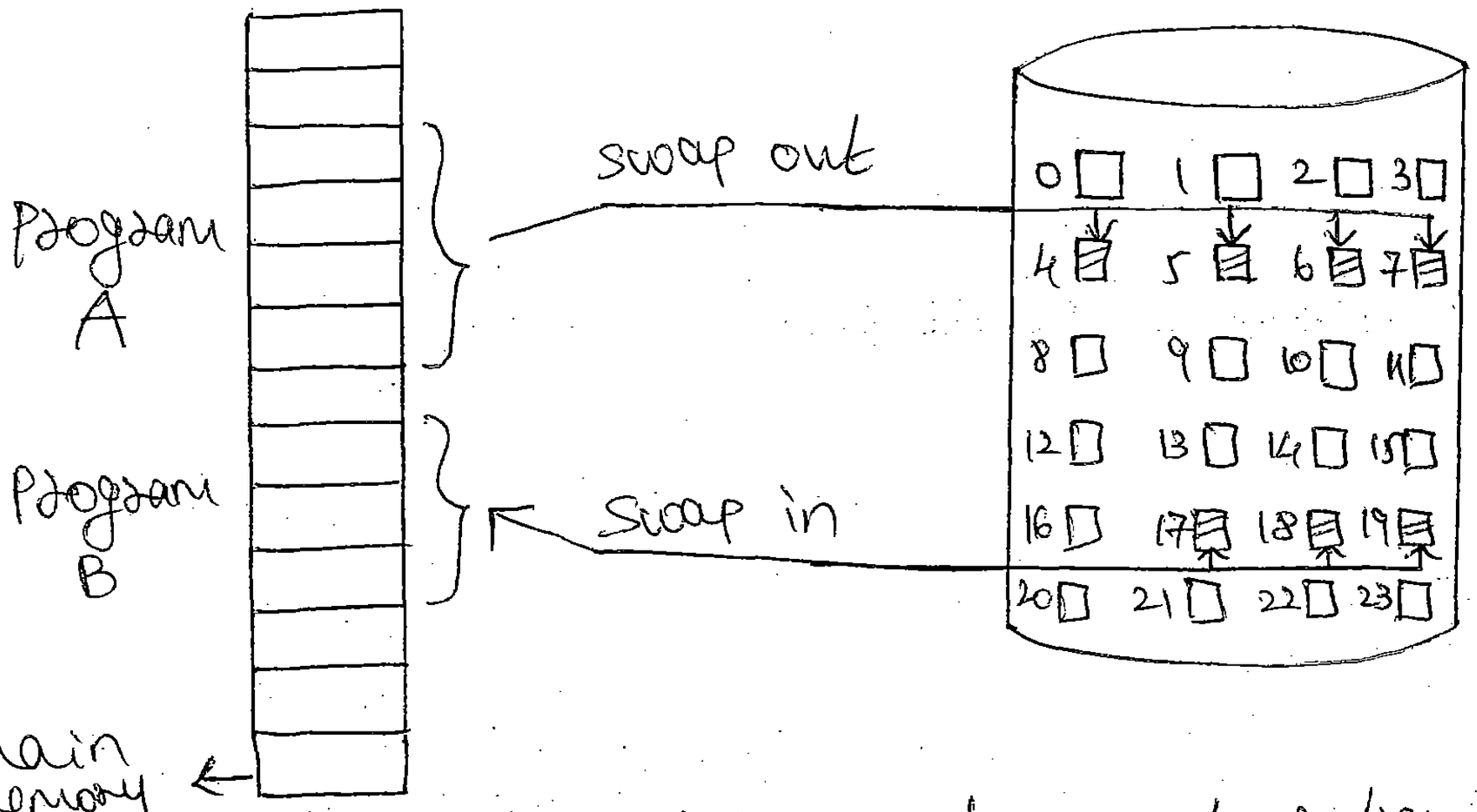
shared library using virtual memory

---

⑪ Demand Paging

→ Consider a program that starts with a list of available options from which the user is to select.
→ Loading the entire program into memory results in loading the executable code for all options, regardless of whether an option is ultimately selected by the user (or) not.
→ An alternative strategy is to initially load pages only as they are needed.
→ This technique is called as Demand Paging.
  - It is commonly used in virtual memory systems.
→ With demand-paged virtual memory, pages are only loaded when they are demanded during program execution.
→ pages that are never accessed are thus never loaded into physical memory.

→ A demand-paging system is similar to a paging system with swapping, where processes reside in secondary memory (usually a disk).



Transfer of a paged memory to contiguous disk space

→ When we want to execute a process, we swap it into memory.

→ Rather than swapping the entire process into memory, however, we use a lazy swapper.

— A lazy swapper never swaps a page into memory unless that page will be needed.

— Using the term swapper is technically incorrect.

→ A swapper manipulates entire processes, whereas a pager is concerned with the individual page of a process.

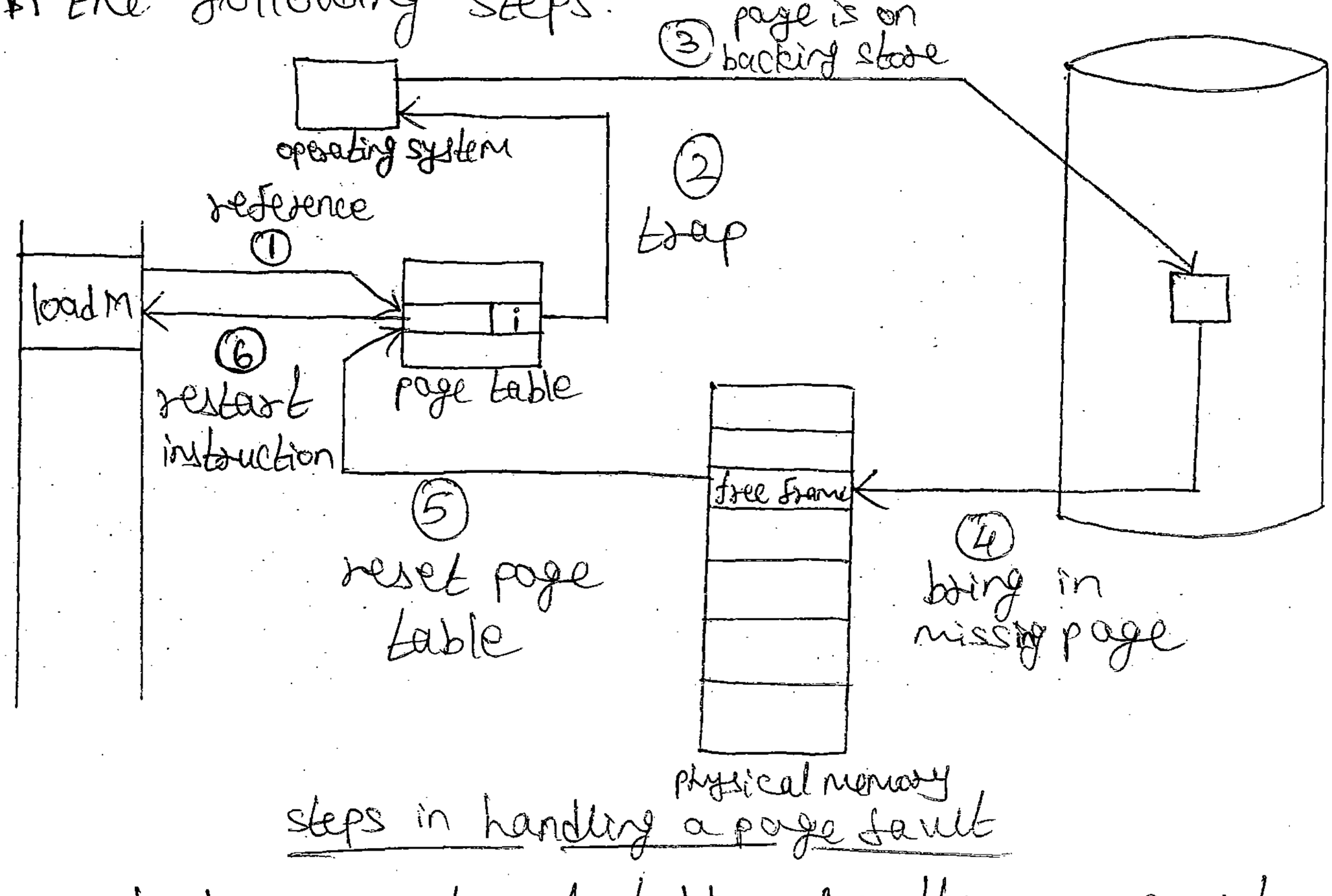
→ Thus, we use 'pager', rather than 'swapper' in connection with Demand paging.

Basic concepts

→ While the process executes and accesses pages that are memory resident, execution proceeds normally.

→ Access to a page marked invalid causes a page-fault trap.

→ The procedure for handling page fault is shown in the following steps:



steps in handling a page fault

1) we check an internal table 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.
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.

→ Pure Demand paging: Never bring a page into memory until it is required.

→ Programs tend to have locality of Reference, which results in reasonable performance from demand paging.

→ The hardware to support demand paging is the same as the hardware for paging and swapping:

* 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 swap device and the section of disk used for this purpose is known as swap space.

→ A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault.

→ A page fault may occur at any memory reference.

→ As a worst-case example, consider a three-address instruction such as ADD the content of A to B, placing the result in C. These steps are to execute the instruction:

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 store in C, we will have to get the desired page, bring it in, correct the page table, and restart the instruction.

→ The major difficulty arises when one instruction may modify several different locations.

Performance of Demand Paging

→ Demand paging can significantly affect the performance of a computer system.

→ Let 'p' be the probability of a page fault ($0 \leq p \leq 1$). we ~~sh~~ould expect 'p' to be close to zero – i.e., we could expect to have only a few page faults. The <u>effective access time</u> is then:

"effective access time = $(1-p) \times ma + p \times$ page fault time"

where, ma = memory-access time.

→ To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1) Trap to the OS.
2) save the uses registers and process state.
3) Determine that the interrupt/signal was a page fault.
4) check that the page reference was legal and 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 (CPU scheduling, optional).

7) Receive an interrupt from the disk I/O subsystem.
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 desired page is now in 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 three major components of the page-fault service time:

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

→ If we take an average page-fault service time of 8ms and a memory-access time of 200ns, then the effective access time in ns is:

$$\begin{aligned} \text{effective access time} &= (1-p) \times (200) + p\,(8\text{ ms}) \\ &= (1-p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p. \end{aligned}$$

→ We see here, the effective access time is directly proportional to the page-fault rate.

⑪ Copy-on-Write

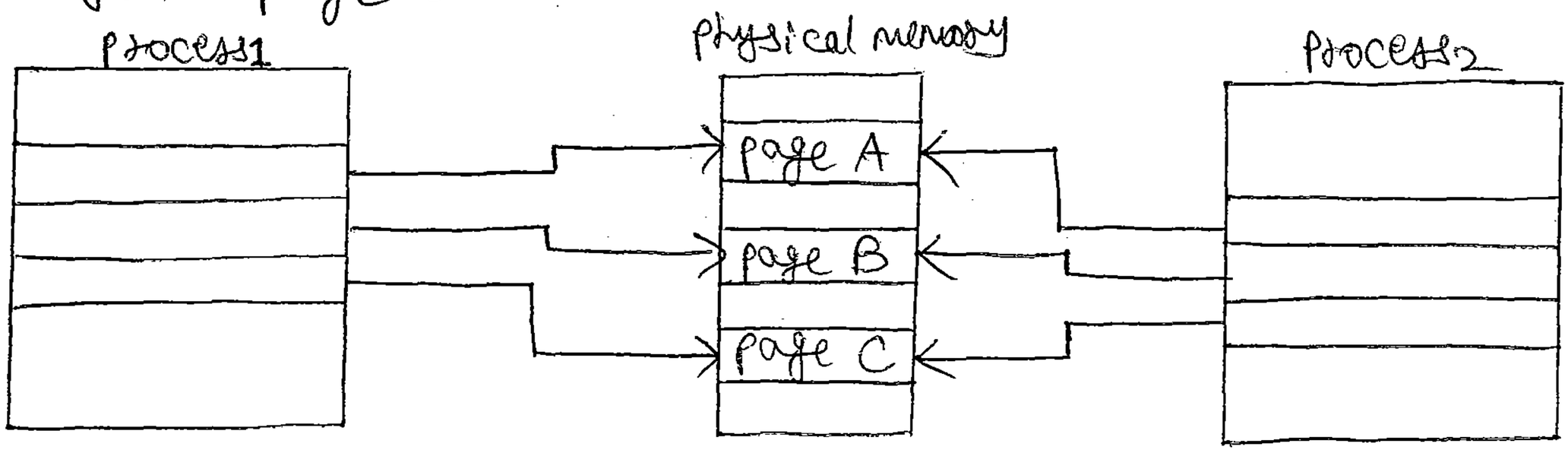
→ Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.

→ However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary.

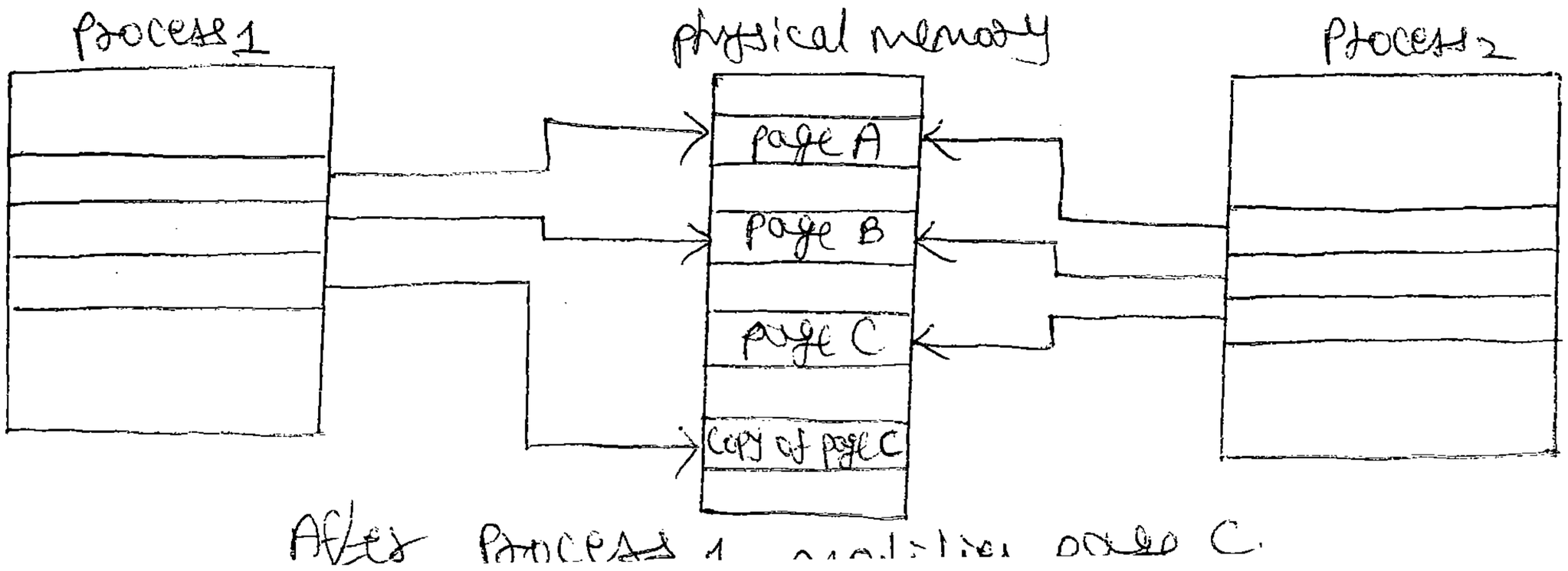
→ Alternatively, we can use a technique known as: copy-on-write, which works by allowing the parent and child processes initially to share the same pages.

- These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

→ Copy-on-write is illustrated, which show the contents of the physical memory before and after process 1 modifies page C.



Before process 1 modifies page C.



After process 1 modifies page C.

→ Many operating systems provide a pool of free pages for the requests. these free pages are typically allocated when the stack/heap for a process must expand (or) when these are copy-on-write pages to be managed.

→ OSs typically allocate these pages using a technique known as zero-fill-on-demand.

→ zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.

→ several versions of UNIX also provide a variation of the fork() system call - vfork() : virtual memory fork.

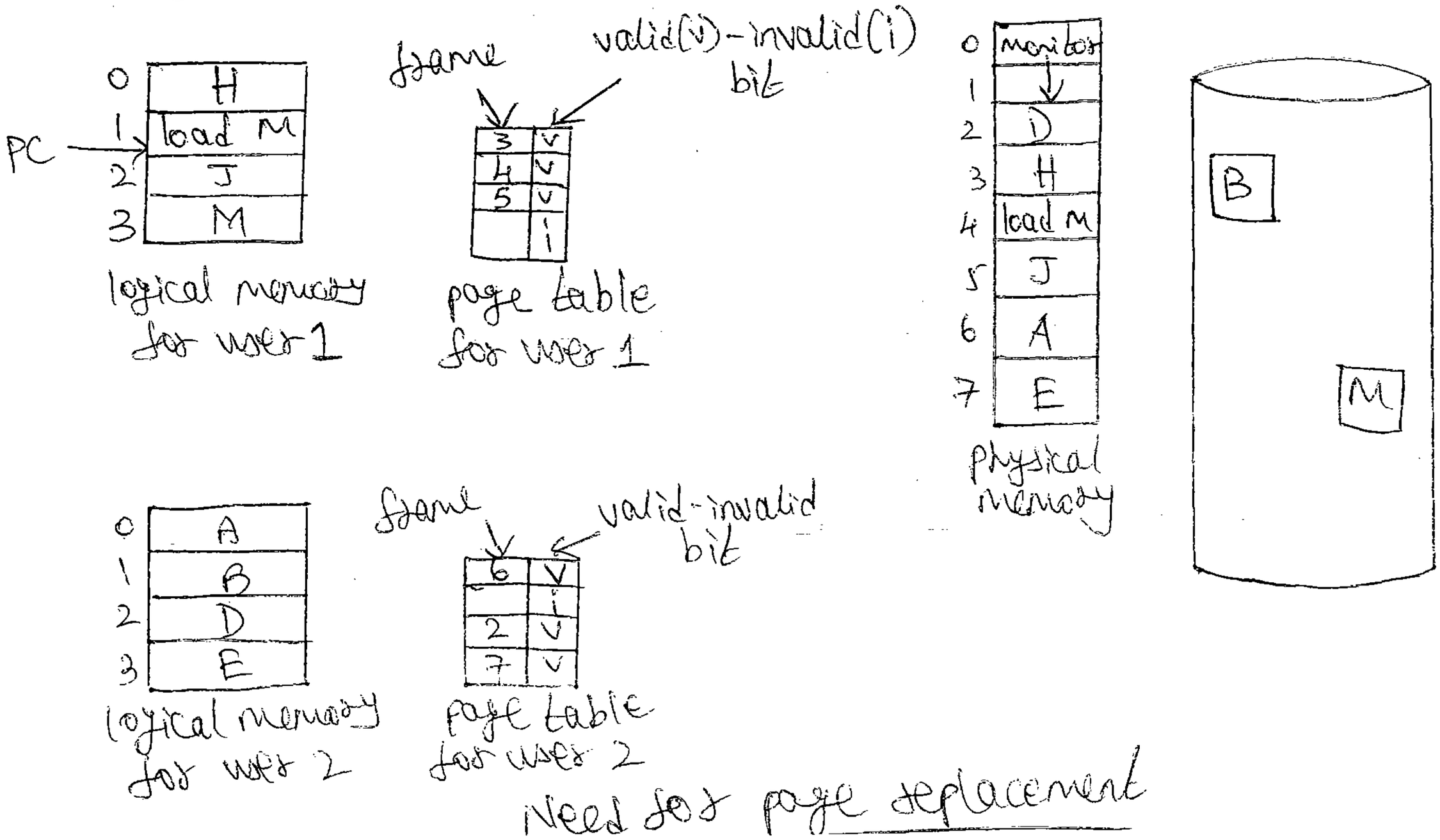
---

⑩ Page Replacement.

→ If we increase our degree of multiprogramming, we are over-allocating memory.

→ Over-allocation of memory manifests as follows by itself.

* while a user process is executing, a page fault occurs.
* the OS determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use.



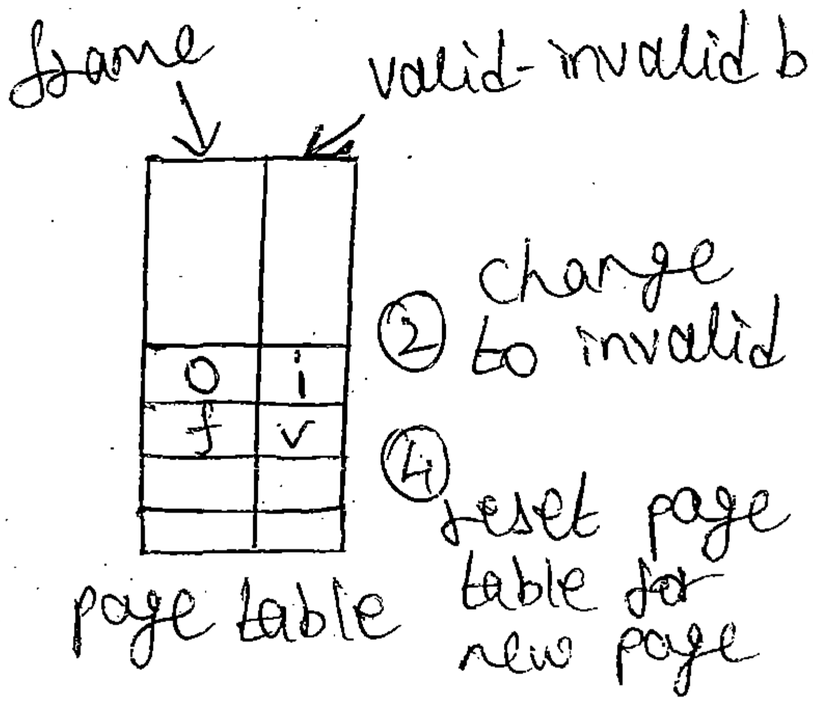
Need for page replacement

→ Most common solution: Page Replacement.

Basic page Replacement

→ Page replacement takes the following approach: If no frame is free, we find one that is not currently being used and free it.

- It can be done by writing its contents to swap space and changing the page table, to indicate that the page is no longer in memory.



Page Replacement

→ we modify the page-fault service routine to include page replacement:

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 a victim frame.
   (c) write the victim frame to the disk; change the page and frame tables accordingly.
3) Read the desired page into the newly freed frame; change the page and frame tables.
4) Restart the user process.

→ we can reduce the overhead by using a modify bit (or dirty bit)

→ Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory.

— with this mechanism, an enormous virtual memory can be provided for programmes on a smaller physical memory.

→ we must solve two major problems to implement demand paging: we must develop a frame-allocation algorithm and a page-replacement algorithm

→ we evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.

— The string of memory references is called a Reference string

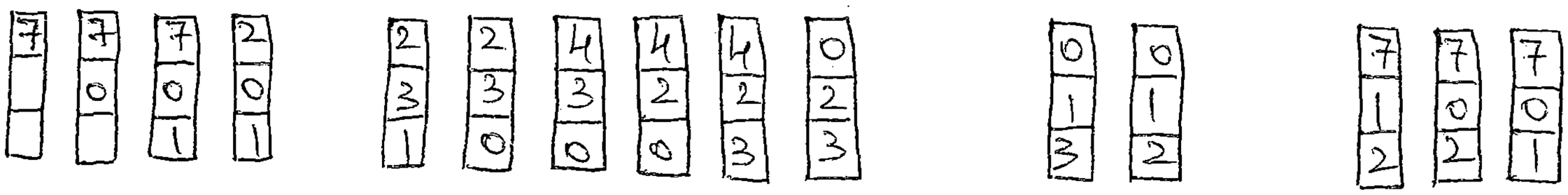
---

FIFO page Replacement

→ The simplest page-replacement algorithm is First-In, First-Out (FIFO) algorithm.

Reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



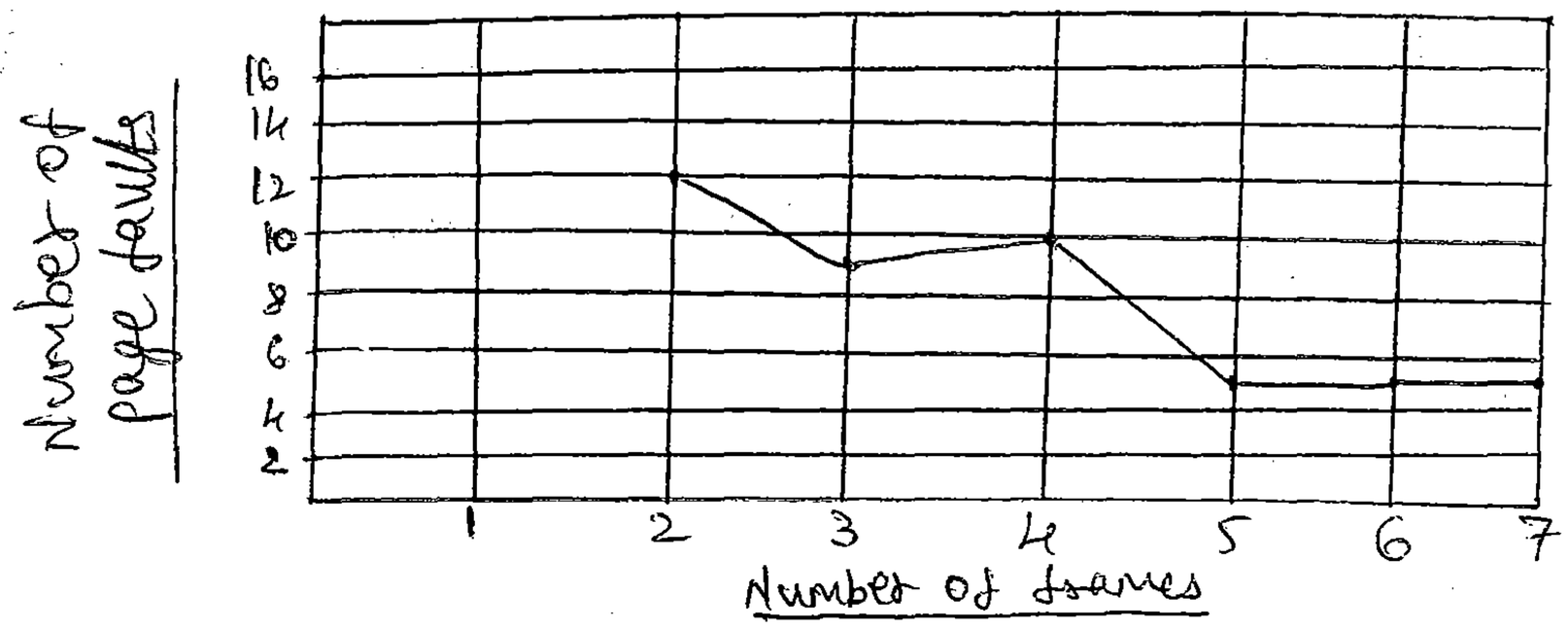
page frames

page faults = 15

→ To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

→ The graph shows the curve of page faults for this above reference string versus the number of available frames.



→ Notice that the number of faults for four (4) frames is ten (10) is greater than the number of faults for three (3) frames is nine (9). This most unexpected result is known as Belady's Anomaly.

- For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

Optimal Page Replacement

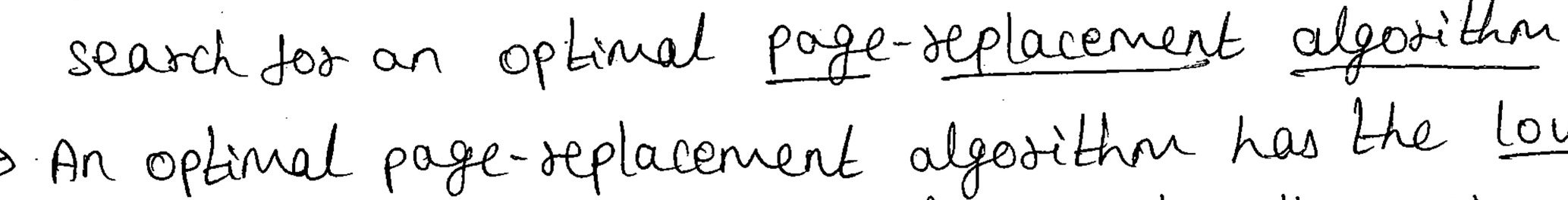
→ One result of the discovery of Belady's Anomaly was the search for an optimal page-replacement algorithm.

→ An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's Anomaly.

"Replace the page that will not be used for the longest period of time."

Reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

page faults = 9

→ Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (Similar to SJF-scheduling Algorithm)

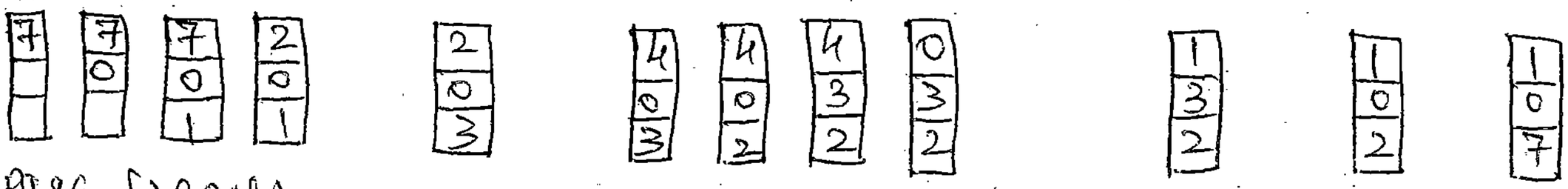
LRU Page Replacement

→ If we use the recent past as an approximation of the near future, then we can replace the page that "has not been used" for the longest period of time.

— This approach is the Least-Recently-Used (LRU) algorithm.

Reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



Page frames

Page faults = 12

→ The LRU policy is often used as a page-replacement algorithm and is considered to be good.

→ The major problem is how to implement LRU replacement.

→ It requires substantial hardware assistance

→ Two implementations are feasible:

1) Counters - In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock (or) counter. The clock is incremented for every memory reference.

2) Stack - Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever the page is referenced, it is removed from the stack and put on the top.

→ Like optimal replacement, LRU replacement does not suffer from Belady's Anomaly. Both belong to a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's Anomaly.

LRU - Approximation Page Replacement

→ Some systems provide no hardware support and other page-replacement algorithms (such as FIFO algorithm) must be used.

→ many systems provide some help, however, in the form of a reference bit - for a page is set by hardware whenever that page is referenced.

→ Reference bits are associated with each entry in the page tabl

→ Additional-Reference-Bits Algorithm.

* we can gain additional ordering information by recording the reference bits at regular intervals.
* A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.
* The number of bits of history can be varied, of course, and is selected to make the updating as fast as possible.
* In extreme case, the number can be reduced to zero, leaving only the reference bit itself.
  - This algorithm is called the second-chance-page-replacement algorithm.

→ Second-chance Algorithm [clock Algorithm]

* One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is to be replaced next.
* Once a victim page is found, the page is replaced and the new page is inserted in the circular queue in that position.

→ Enhanced second-chance Algorithm

* we Enhance by considering the reference bit & the modify bit.
* with these two bits, we have following four possible classes:
  1) (0,0) neither recently used nor modified - best page to replace.

3) (0,1) not recently used but modified - not quite as good, because the page will need to be written out before replacement.

3) (1,0) recently used but clean - probably will be used again soon.

4) (1,1) recently used and modified - probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.

Counting-Based Page Replacement

→ The least frequently used (LFU) page-replacement algorithm
- requires that the page with the smallest count be replaced.

→ The most frequently used (MFU) page-replacement algorithm
- is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

---

Allocation of Frames

Minimum number of Frames

→ One reason for allocating at least a minimum number of frames involves performance.

→ The minimum number of frames is defined by the computer architecture.

→ The maximum number of frames is defined by the amount of available physical memory.

Allocation Algorithms.

→ The easiest way to split 'm' frames among 'n' processes is to give everyone an equal share, 'm/n' frames.

→ For example, if there are 93 frames and 5 processes, each process will get 18 frames. The leftover 3 frames can be used as a free-frame buffer pool. This scheme is called Equal Allocation.

→ Proportional Allocation - we allocate available memory to each process according to its size.

* Let the size of the virtual memory for process 'pi' be 'si' and define:

$$S = \sum s_i.$$

* Then, if the total number of available frames is 'm', we allocate 'ai' frames to process 'pi', where 'ai' is approximately:

$$a_i = s_i/S \times m.$$

* for proportional allocation, we would split 62 frames between 2 processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since:

$$10/137 \times 62 \approx 4 \text{ and}$$

$$127/137 \times 62 \approx 57.$$

Global versus Local Allocation

→ Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process.

→ Local replacement requires that each process select from only its own set of allocated frames.

⓪ Thrashing

→ If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.

→ At this point, it must replace some page.

→ However, since all its pages are in active use, it must replace a page that will be needed again right away.

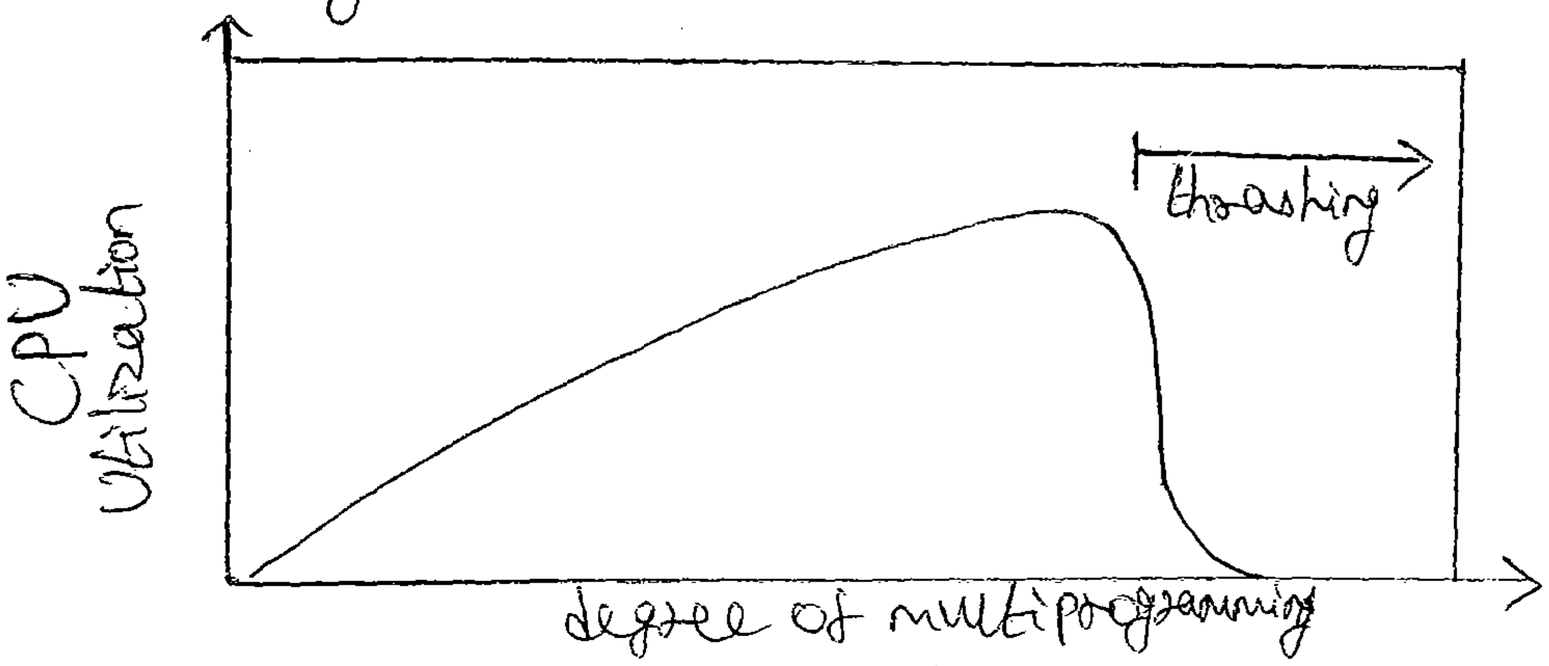
→ Consequently, it quickly faults again and again, and so on, replacing pages that it must bring back in immediately.

— This high paging activity is called <u>Thrashing</u>.

→ A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

→ Thrashing results in severe performance problems.



<u>Thrashing</u>

→ Here, CPU utilization is plotted against the degree of multiprogramming.

→ As the degree of multi-programming 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.

→ we can limit the effects of thrashing by using local replacement algorithm (or priority replacement algorithm).

→ The locality model states that, as a process executes, it moves from locality to locality.
- A locality is a set of pages that are actively used together.

---

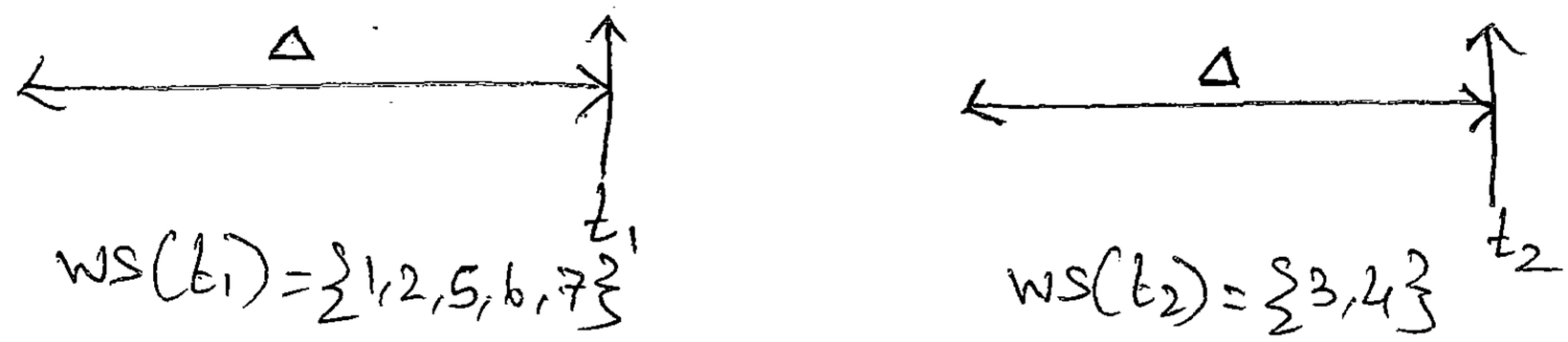
working-set Model

→ The working-set model is based on the assumption of locality

→ This model uses a parameter, 'Δ' to define the working-set window. The idea is to examine the most recent Δ page references.

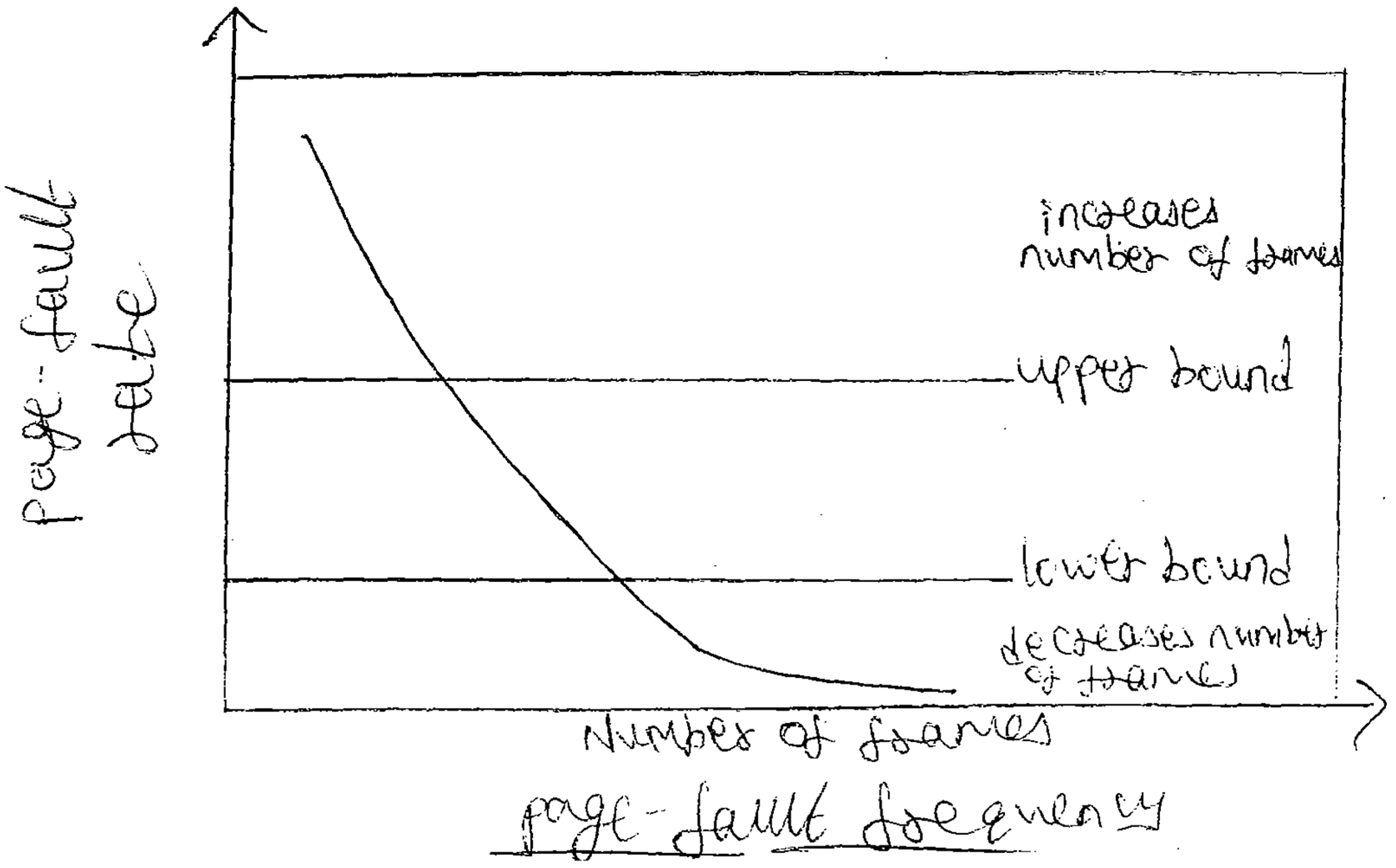
page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



working-set model

Page-Fault Frequency



page-fault frequency