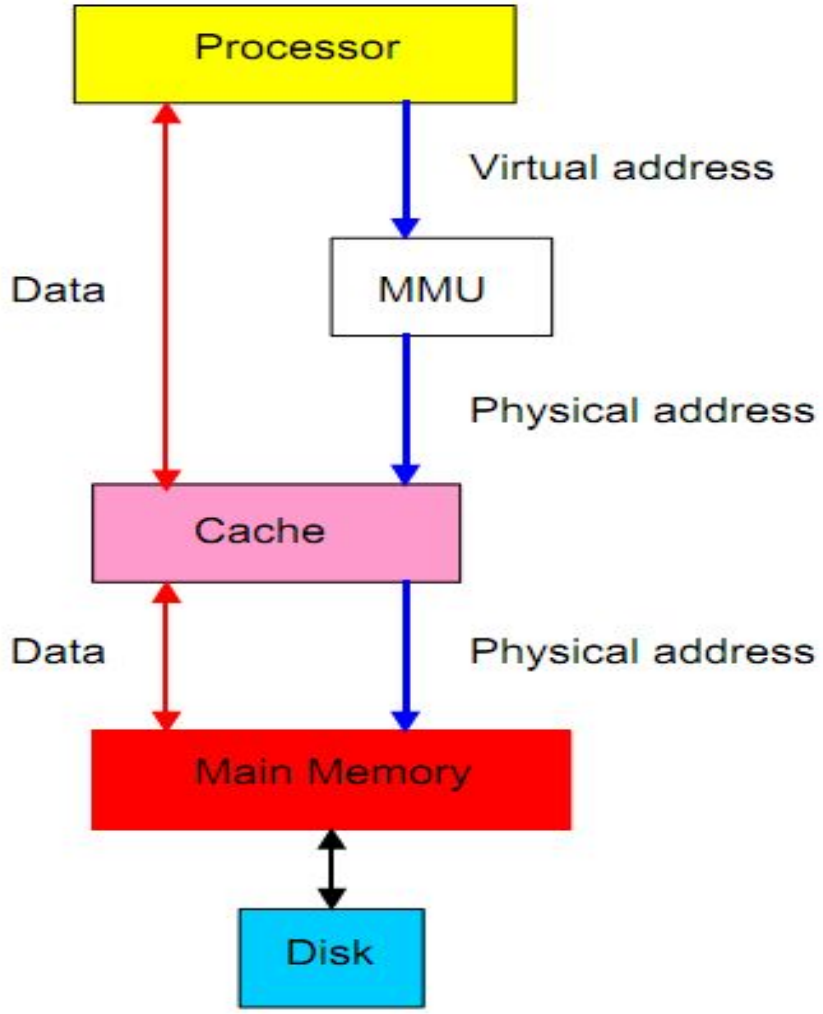
Module IV: Memory Management

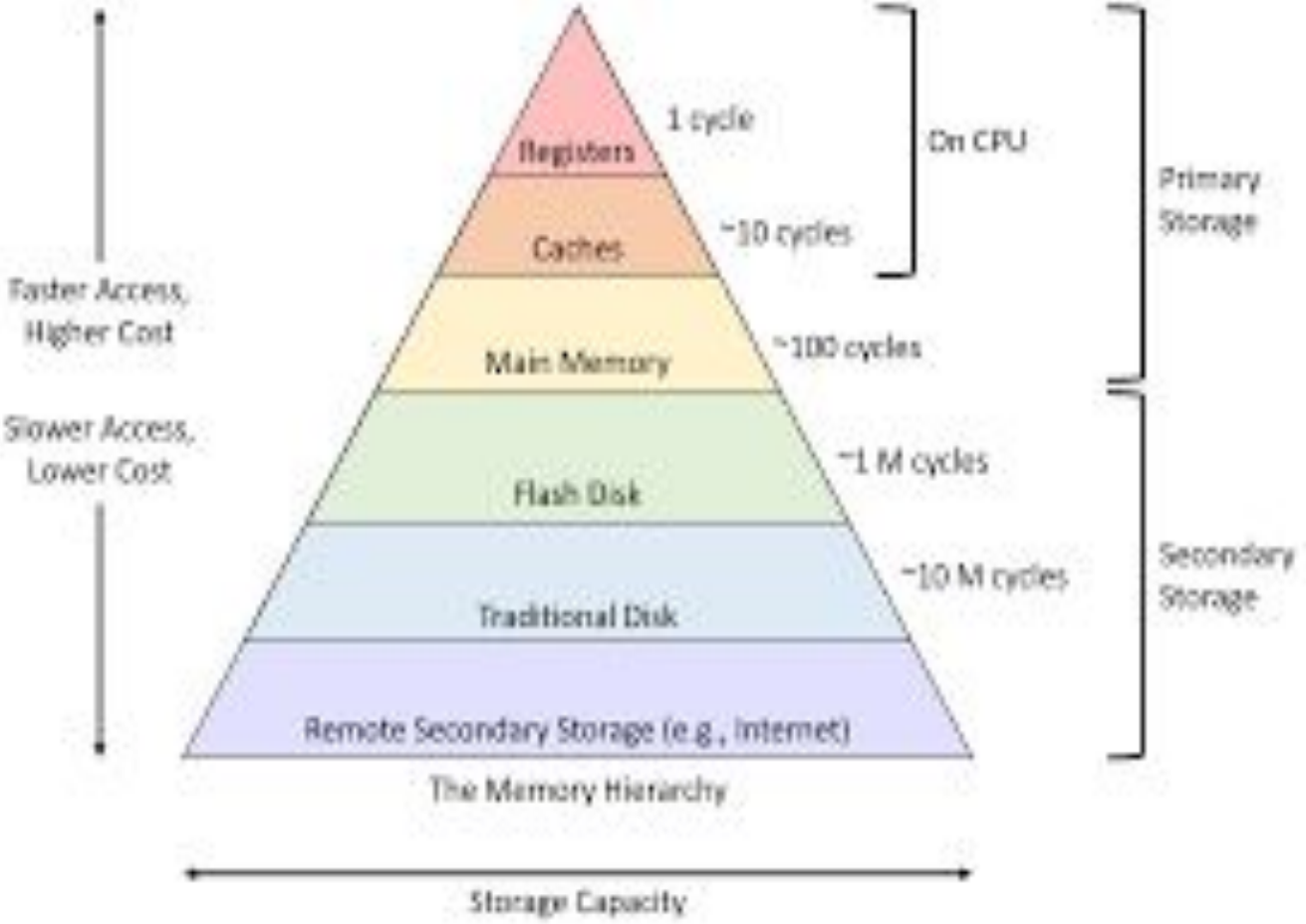
- Memory Management: Basics - Swapping -Memory Allocation (fixed partitions, variable partitions) Fragmentation - Paging - Segmentation - Virtual memory concepts – Demand paging - Page replacement algorithms (FIFO, Optimal, LRU) – Allocation of frames - Thrashing.

Memory Management Basics

- Don't have infinite RAM
- Do have a memory hierarchy-
  - Cache (fast)
  - Main(medium)
  - Disk(slow)
- Memory manager has the job of using this hierarchy to create an abstraction (illusion) of easily accessible memory



Memory Hierarchy



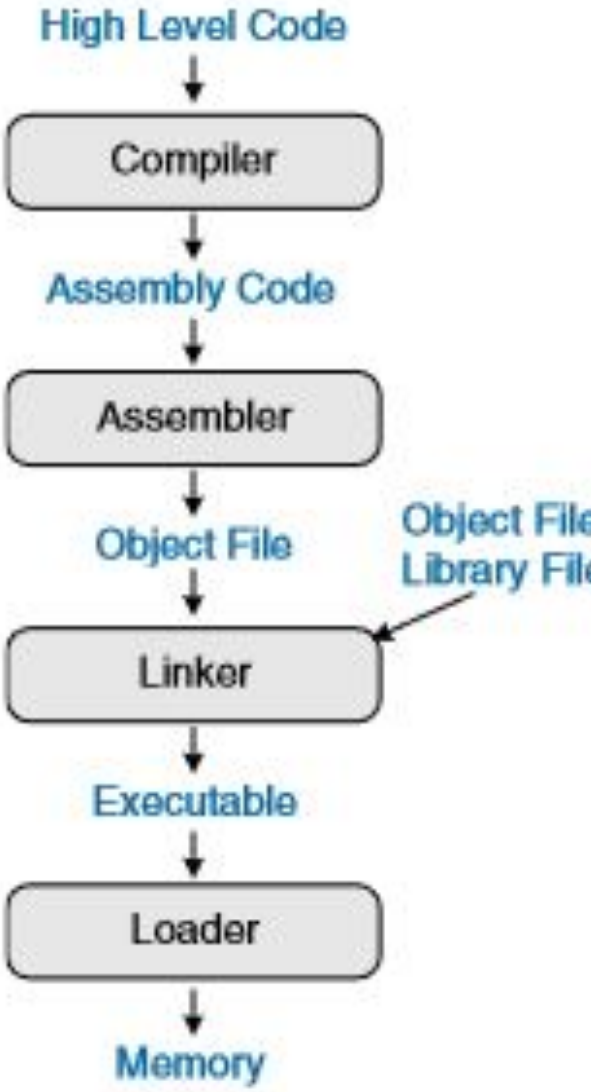
The Memory Hierarchy

Background

- Program must be brought into memory and placed within a process for it to be executed.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.

Steps before program execution

- Step-1: compilation
- Compiler translates high-level code into assembly language
- Step 2: Assembling
- Assembler turns assembly language code into an object file containing machine language code.
- Step 3: Linking
- Linker combines all of the object files into one machine language file called the executable.
- Step 4: Loader
- Loader loads the program into memory and starts execution.
- In practice, most compilers perform all three steps



Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time**: Compiler must generate *relocatable* code if memory location is not known at compile time and final binding is delayed until load time. Only reload user code if starting location changes.
- **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 (e.g., *base* and *limit registers*).

Dynamic Linking

- Linking postponed until execution time.
- A **dynamic linker** is part of OS that loads and links shared libraries needed by an executable when it is executed (at "run time"), by copying the content of libraries from persistent storage to RAM, filling jump tables and relocating pointers.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with address of routine, and executes routine.
- OS need to check if routine is in processes' memory address.

Overlays

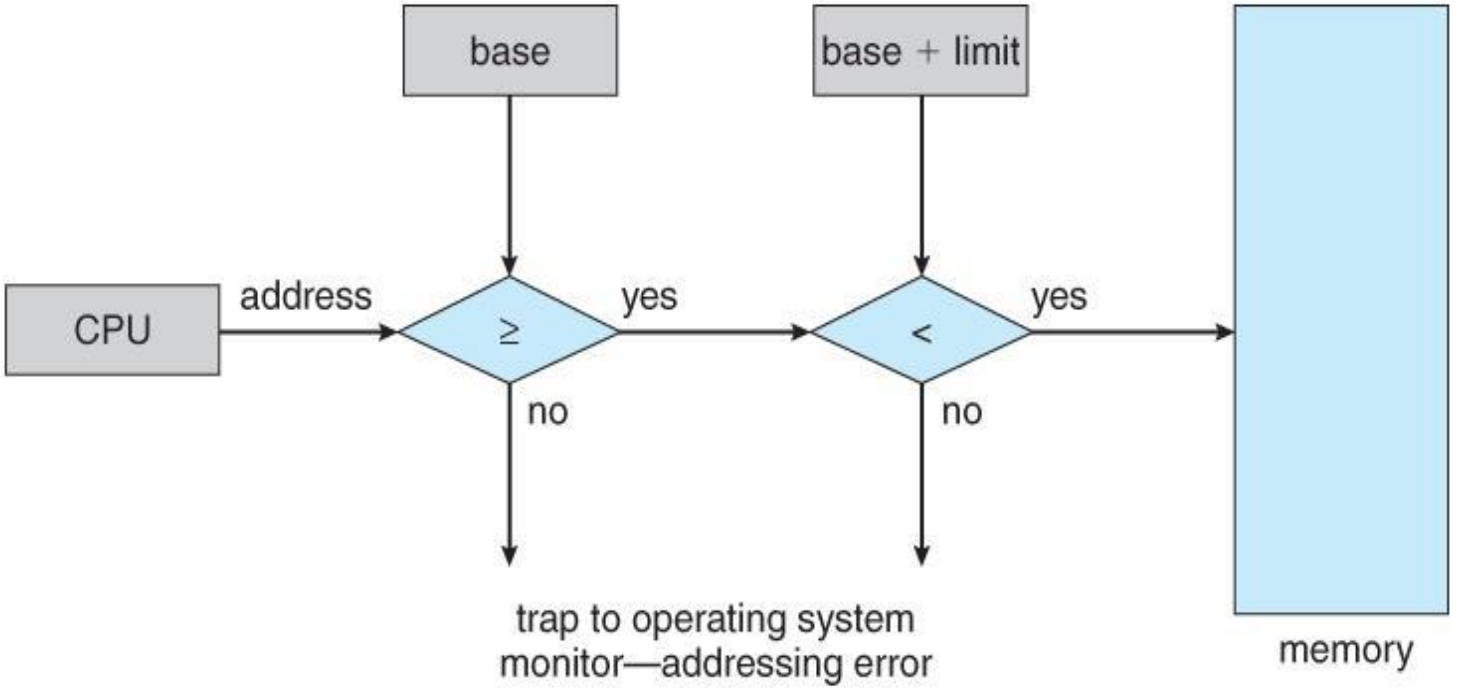
- Overlays-programmer breaks program into pieces which are swapped in by overlay manager
- Needed when process is larger than memory allocated to it.
- Idea of overlay is to keep in memory only those instructions and data that are needed at any given time.
- When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex.
- Define overlays in the program (divide program codes into

Memory Protection

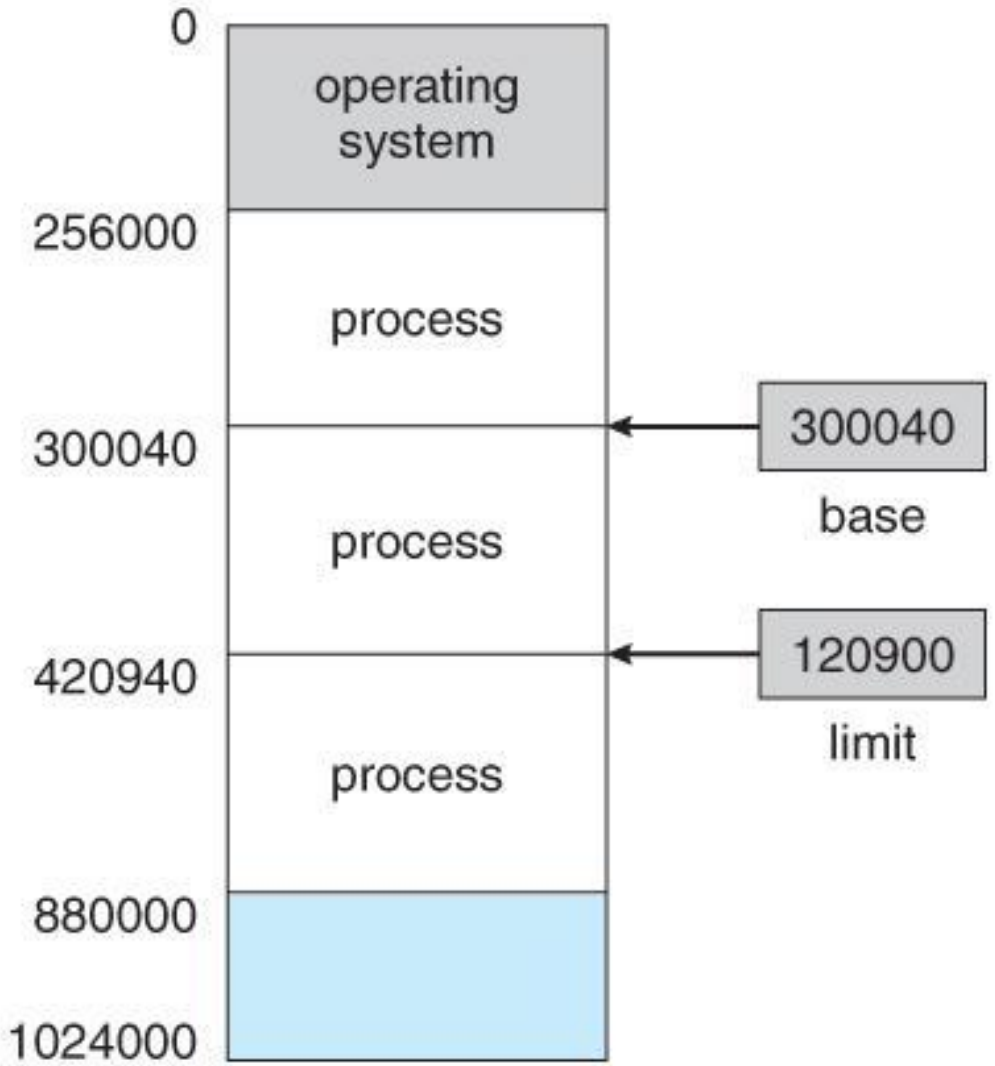
- We can prevent a process from accessing memory it does not own with a relocation register, together with a limit register
- This scheme is used to protect user processes from each other, and from changing operating-system code and data.
- Relocation register contains value of smallest physical address;
- limit register contains range of logical addresses – each logical address must be less than the limit register.
- The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped addressis sent to memory

Hardware support for relocation and limit registers

- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically.



Example


0
operating system
256000
process
300040
process
420940
process
880000
1024000
300040
base
120900
limit

Logical vs. Physical Address Space

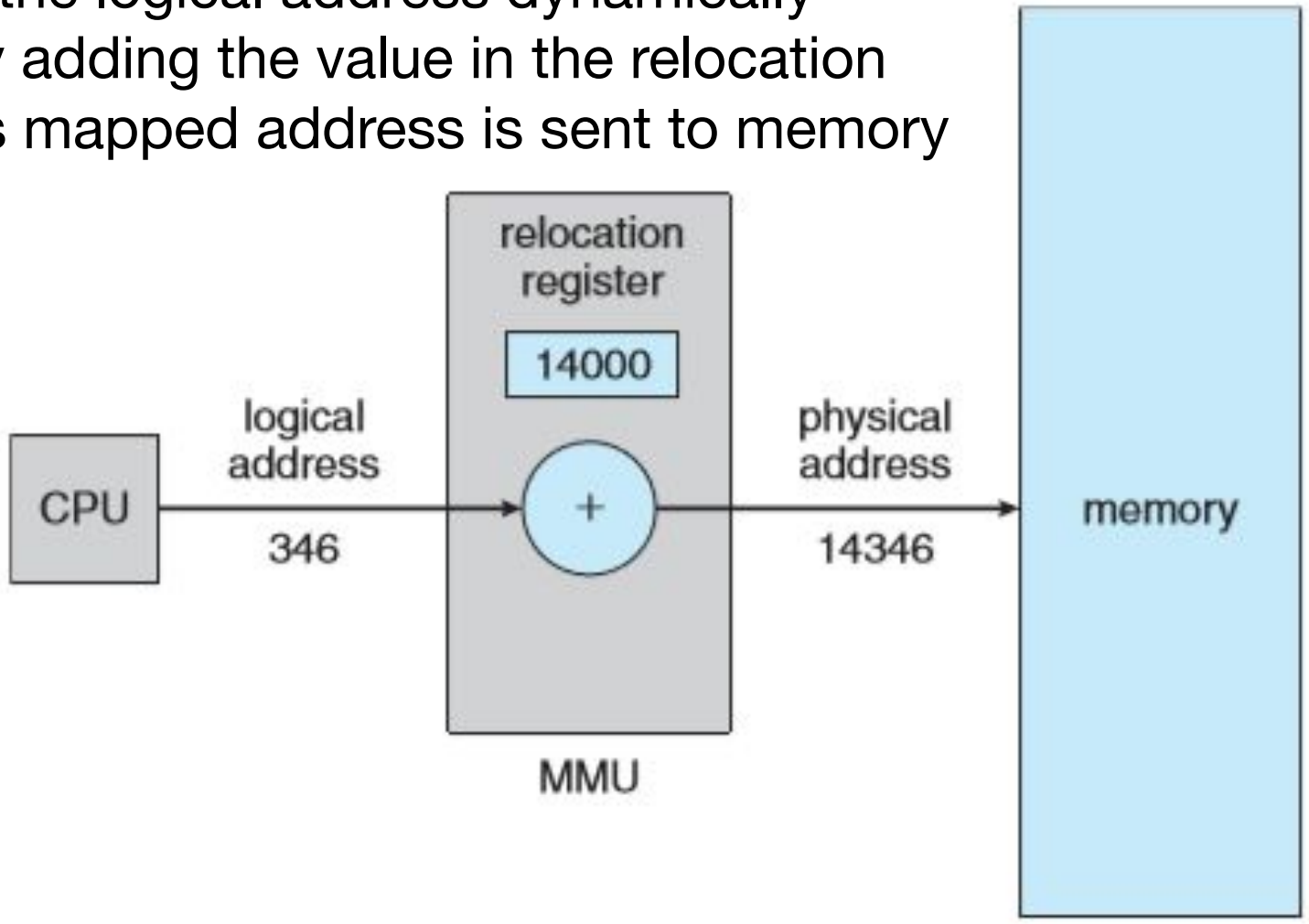
- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.
- Logical and physical 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.

Memory-Management Unit (MMU)

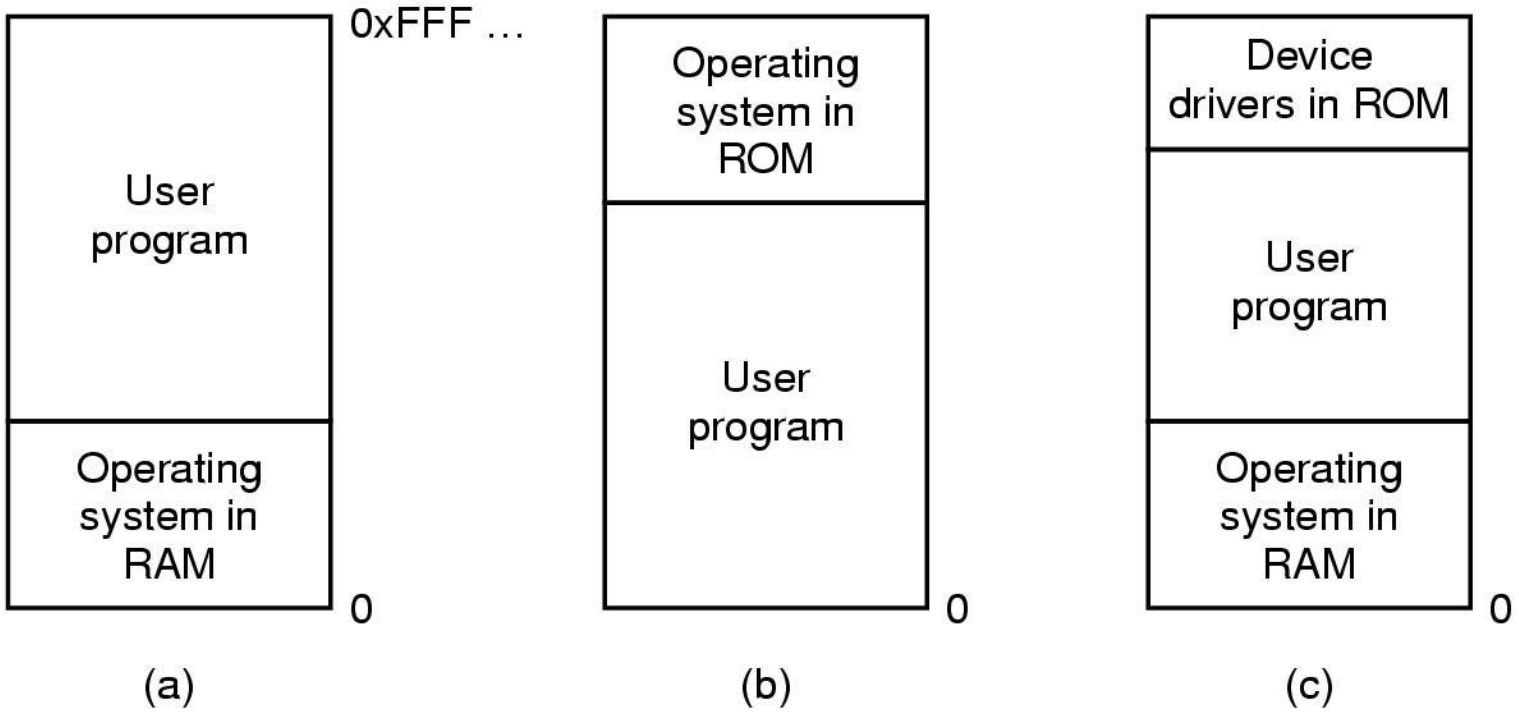
- Hardware device that maps virtual to physical address.
- MMU generates physical address from virtual address provided by the program
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
- The final address of a referenced memory address is determined at the time of reference (execution time binding).

Dynamic relocation using a relocation register

MMU maps the logical address dynamically (run-time) by adding the value in the relocation register. This mapped address is sent to memory



One program at a time in memory



OS reads program in from disk and it is executed

How to run more programs then fit in main memory at once

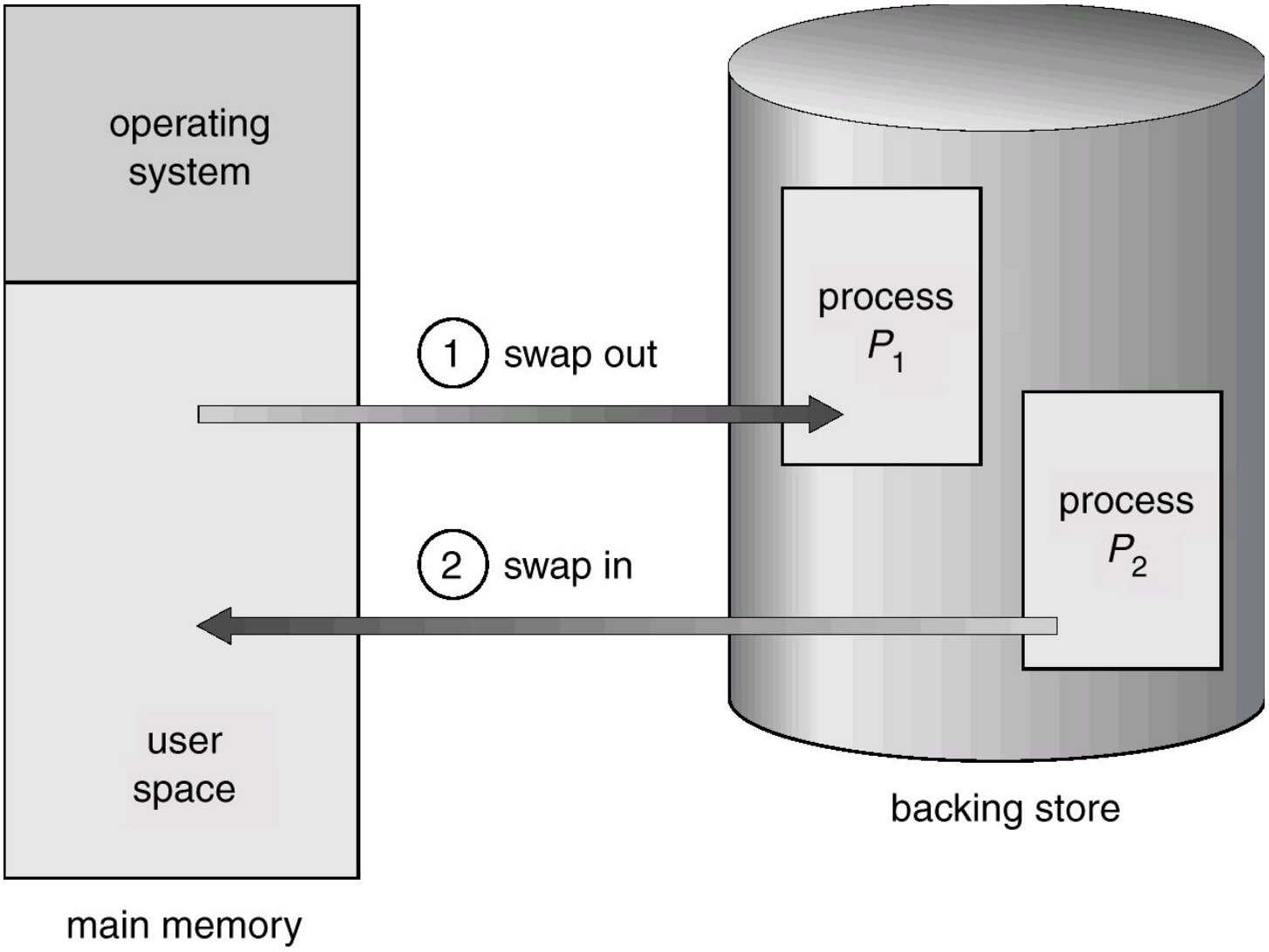
We usually want several user processes to reside in memory at the same time.

- Can't keep all processes in main memory
  - Too many (hundreds)
  - Too big (e.g. 200 MB program)
- Two approaches
  - Swap-bring program in and run it for awhile
  - Virtual memory-allow program to run even if only part of it is in main memory

Swapping

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

Schematic View of Swapping



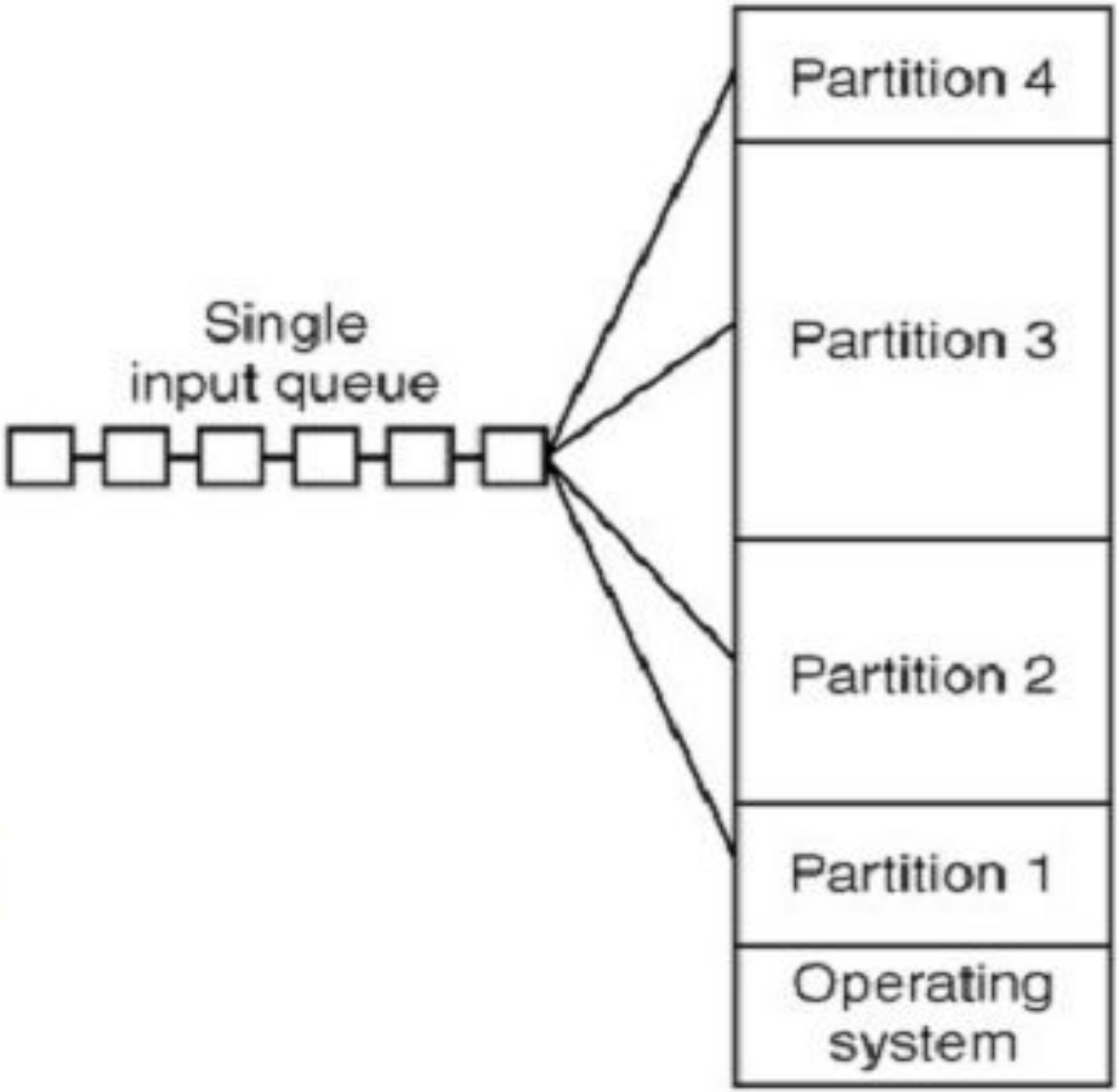
Memory Allocation : Multiple-partition

In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to section containing next process.

1. Fxed-sized partitions

- The simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process.
- Thus, the 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.

Fixed memory partitions with a single input queue



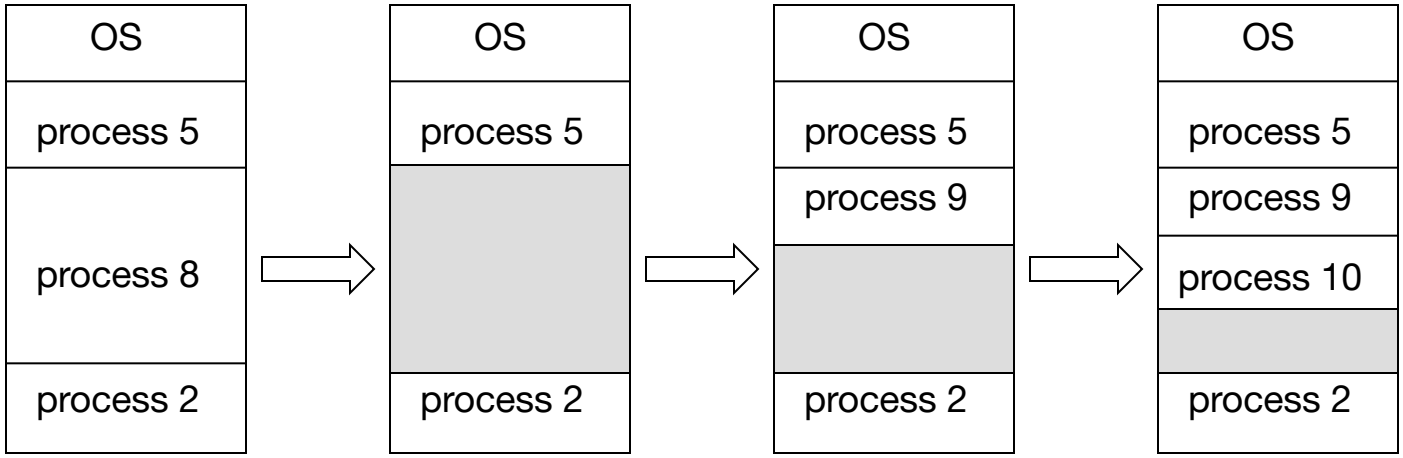
2. Variable-sized partitions

- In the variable-partition scheme, 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 one large block of available memory, a hole.
- As processes enter the system, they are put into an input queue.
- OS takes care of the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- After certain time memory blocks available comprise a set of holes of various sizes scattered throughout memory.
- When a process arrives, the system searches for a hole that is large enough for this process.

Contiguous Allocation (Cont.)

Multiple-partition allocation

- *Hole* – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

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. Produces the largest leftover hole.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

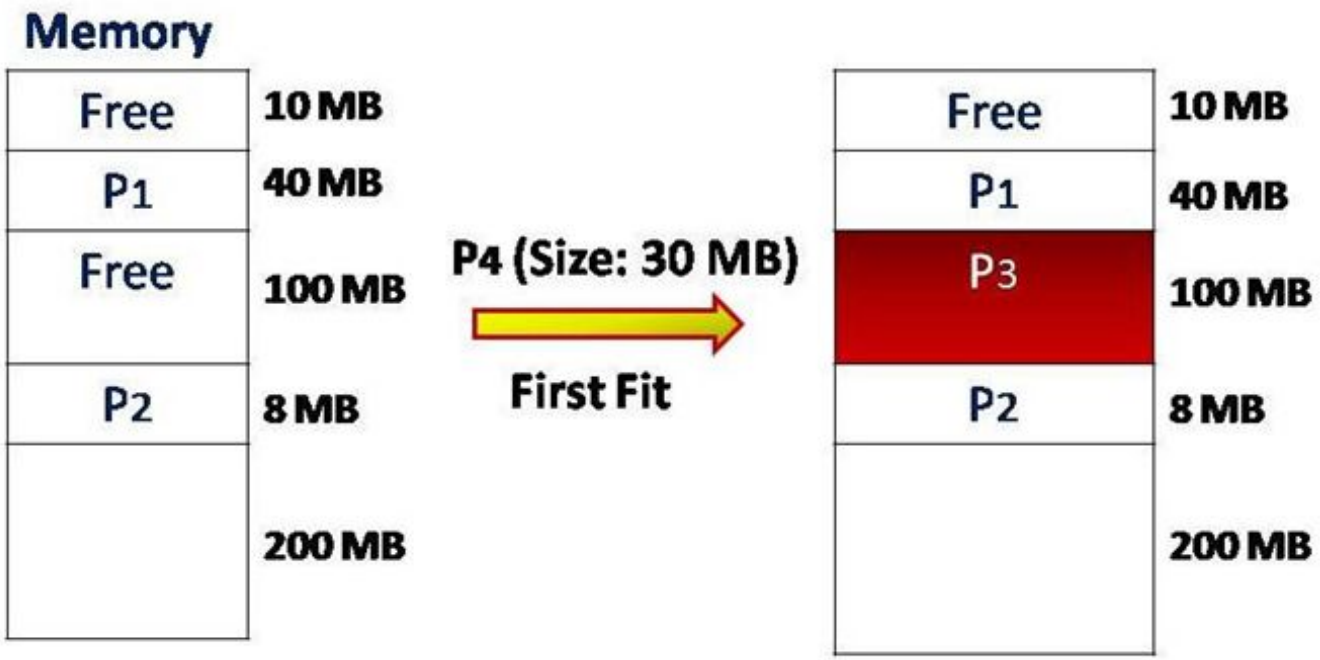
If most of the requests are of similar size, a worst fit policy tends to minimize external fragmentation.

Memory Allocation Policies

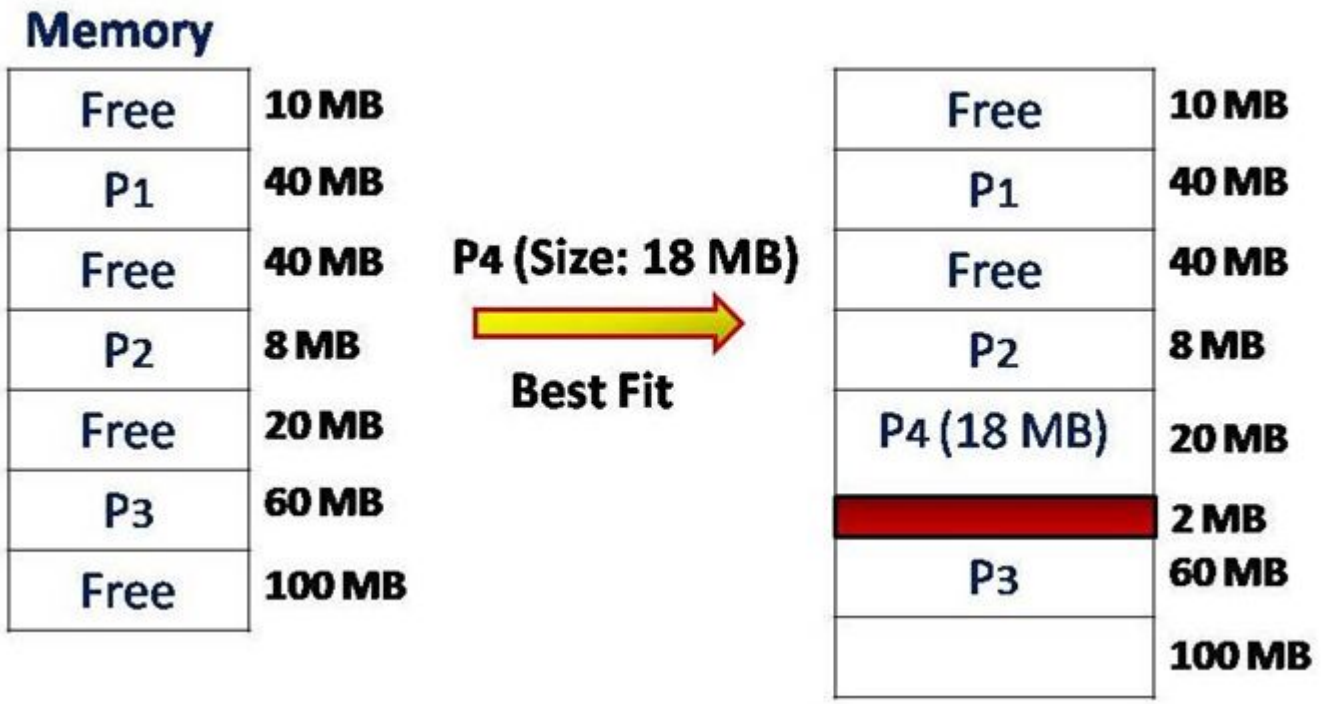
- **First Fit:** first fit allocation is that we begin searching the list and take the first block whose size is greater than or equal to the request size.
- **Best Fit**: This policy allocates the process to the smallest available free block of memory that is closest in size to the request. The best fit may result into a bad fragmentation, but in practice this is not commonly observed.
- **Worst Fit:** This policy allocates the process to the largest available free block of memory. This leads to elimination of all large blocks of memory, thus requests of processes for large memory cannot be met.
-

First fit

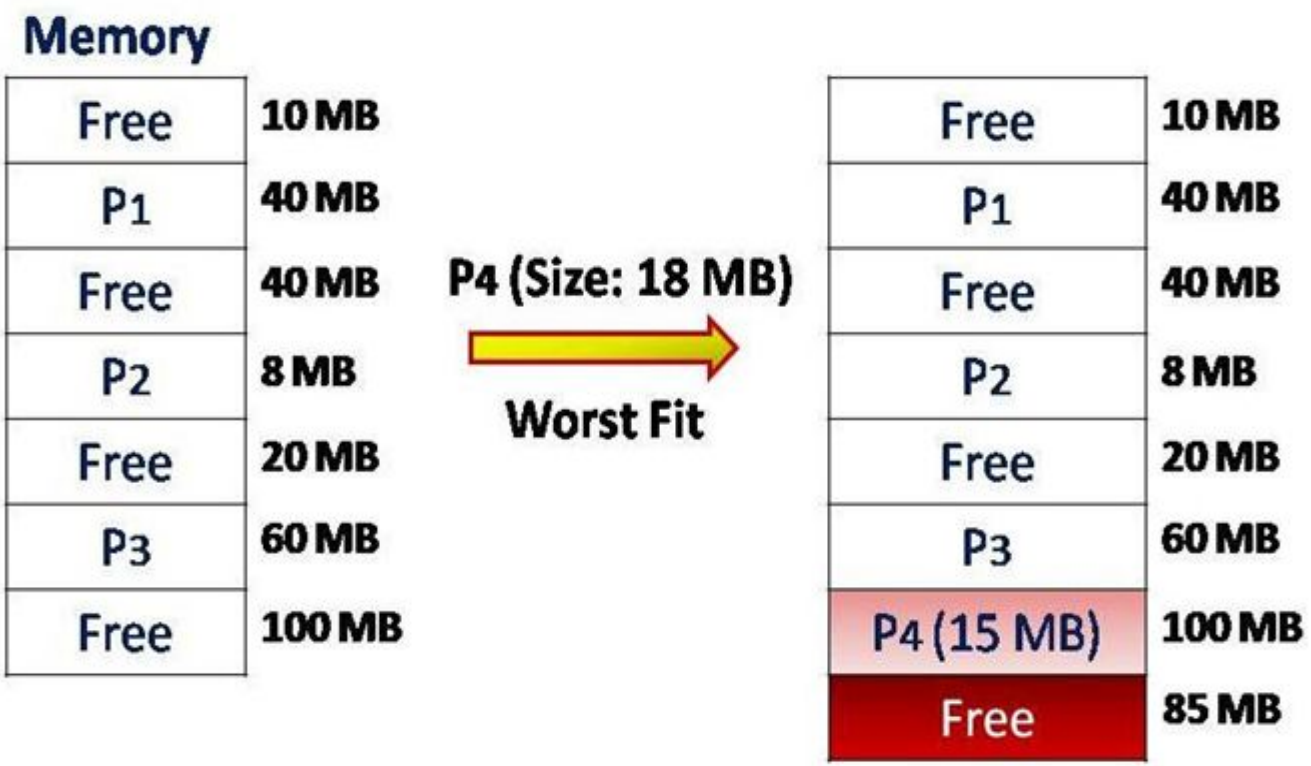
If the P3 size is 30MB then it fit the first location of the memory, first appropriate free location size is 100MB so, it is appropriate for P3



Best fit


Memory
Free
10 MB
P1
40 MB
Free
40 MB
P2
8 MB
Free
20 MB
P3
60 MB
Free
100 MB
P4 (Size: 18 MB)
Best Fit
Free
10 MB
P1
40 MB
Free
40 MB
P2
8 MB
P4 (18 MB)
20 MB
2 MB
P3
60 MB
100 MB

Worst fit


Memory
Free
10 MB
P1
40 MB
Free
40 MB
P2
8 MB
Free
20 MB
P3
60 MB
Free
100 MB
P4 (Size: 18 MB)
Worst Fit
Free
10 MB
P1
40 MB
Free
40 MB
P2
8 MB
Free
20 MB
P3
60 MB
P4 (15 MB)
100 MB
Free
85 MB

Problem

Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory

Fragmentation

- In contiguous memory allocation whenever the processes come into RAM, space is allocated to them.
- These spaces in RAM are divided either on the basis of fixed partitioning(the size of partitions are fixed before the process gets loaded into RAM) or dynamic partitioning (the size of the partition is decided at the run time according to the size of the process).
- As the process gets loaded and removed from the memory these spaces get broken into small pieces of memory that it can't be allocated to the coming processes. This problem is called fragmentation.

Fragmentation

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.
- For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste, or fragmentation.
- Fragmentation refers to the waste space in memory, which cannot be allocated to any process.

Two types: Internal, external

Fragmentation

- Fragmentation is the inability to reuse memory that is free
- External fragmentation occurs when enough free memory is available but isn't contiguous
  - Many small holes
- Internal fragmentation arises when a large enough block is allocated but it is bigger than needed
  - Blocks are usually split to prevent internal fragmentation

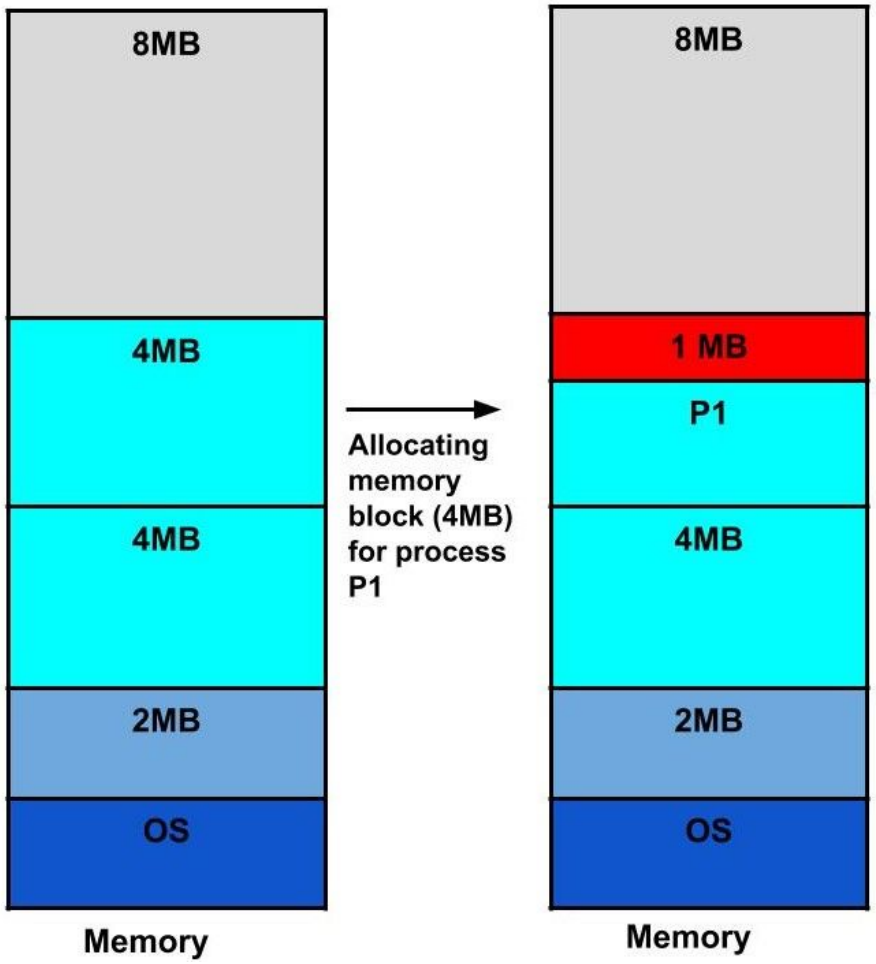
Internal fragmentation

- Internal fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Internal fragmentation—unused memory that is internal to a partition.
- Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation.

Example

- Example: Suppose there is fixed partitioning (i.e. the memory blocks are of fixed sizes) is used for memory allocation in RAM. These sizes are 2MB, 4MB, 4MB, 8MB. Some part of this RAM is occupied by the Operating System (OS).

- Now, suppose a process P1 of size 3MB comes and it gets memory block of size 4MB. So, the 1MB that is free in this block is wasted and this space can't be utilized for allocating memory to some other process. This is called internal fragmentation.

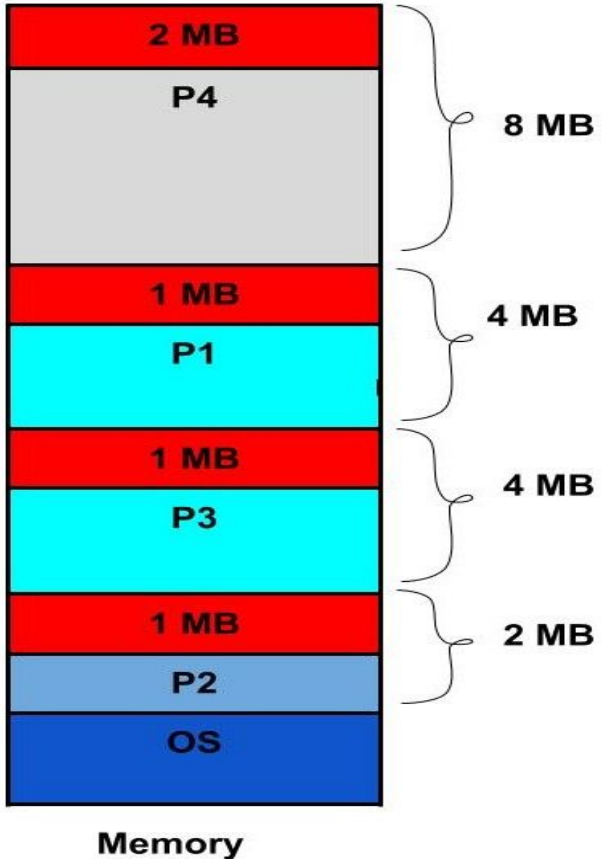

8MB
4MB
4MB
2MB
OS
Memory
Allocating memory block (4MB) for process P1
8MB
1 MB
P1
4MB
2MB
OS
Memory

How to remove internal fragmentation?

- This problem is occurring because we have fixed the sizes of the memory blocks.
- This problem can be removed if we use dynamic partitioning for allocating space to the process.
- In dynamic partitioning, the process is allocated only that much amount of space which is required by the process. So, there is no internal fragmentation.

External Fragmentation

- Total memory space exists to satisfy a request, but it is not contiguous. storage is fragmented into a large number of small holes.
- In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.
- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation


2 MB
P4
8 MB
1 MB
P1
4 MB
1 MB
P3
4 MB
1 MB
P2
2 MB
OS
Memory

How to remove external fragmentation?

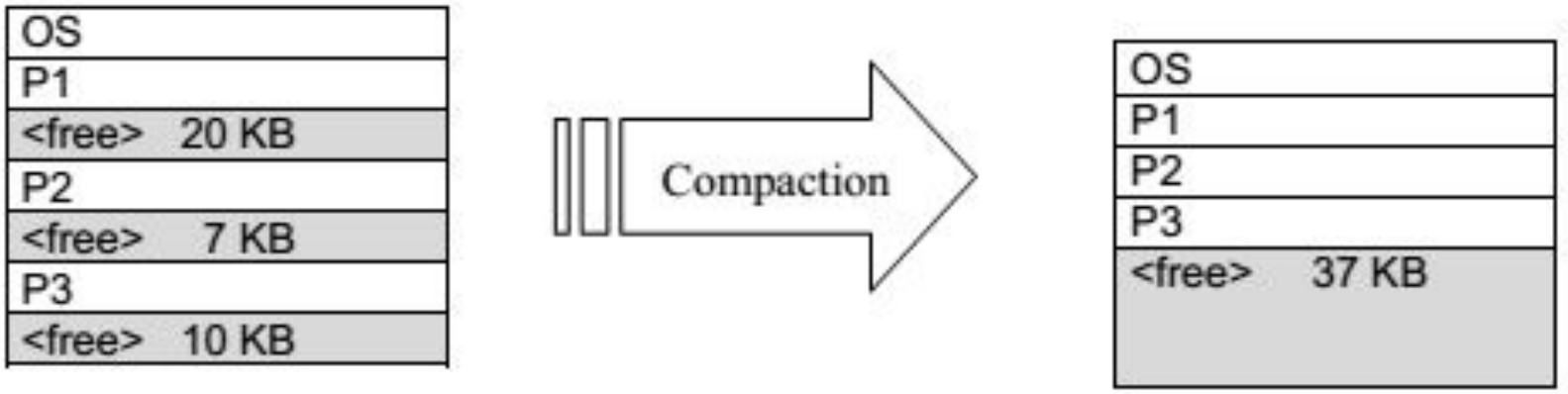
- This problem is occurring because we are allocating memory continuously to the processes. So, if we remove this condition external fragmentation can be reduced. This is what done in **paging** & **segmentation**(non-contiguous memory allocation techniques) where memory is allocated non-contiguously to the processes

- Another way to remove external fragmentation is **compaction**. When dynamic partitioning is used for memory allocation then external fragmentation can be reduced by merging all the free memory together in one large block. This technique is also called defragmentation. This larger block of memory is then used for allocating space according to the needs of the new processes.

Compaction

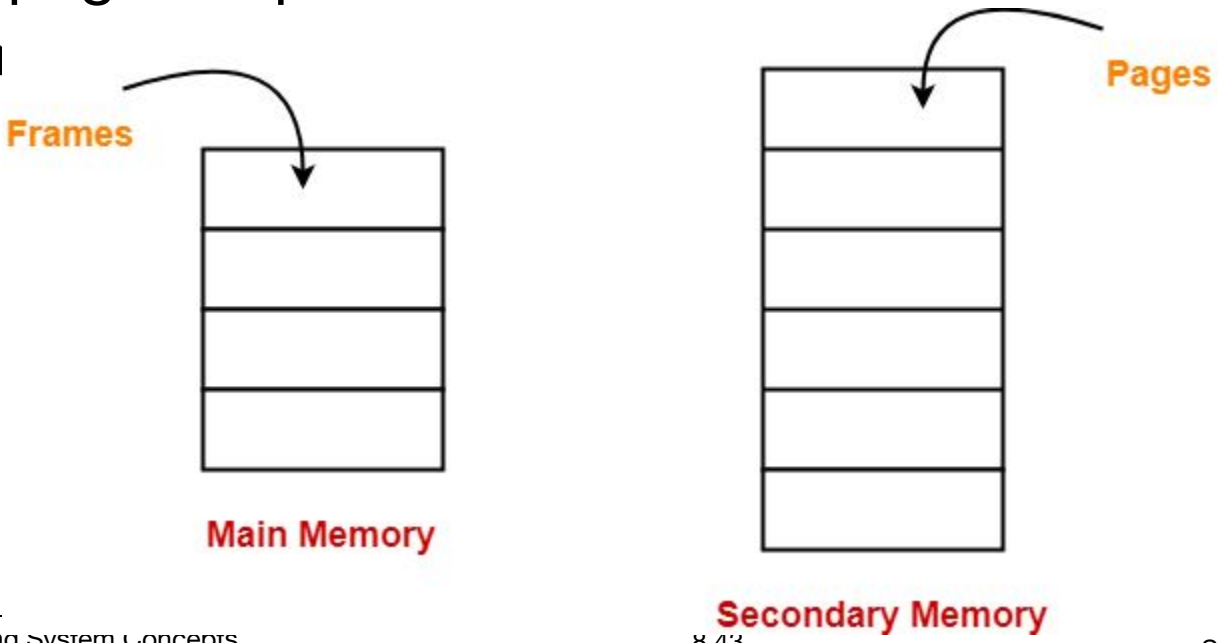
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
  - The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.
- Another possible solution is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever available.
- Two complementary techniques to this solution: segmentation and paging

Memory Compaction Example



Paging

- Paging is a fixed size partitioning scheme.
- Divide physical memory into fixed-sized blocks called frames
- Divide logical memory into blocks of same size called pages.
- The pages of process are stored in the frames of main men

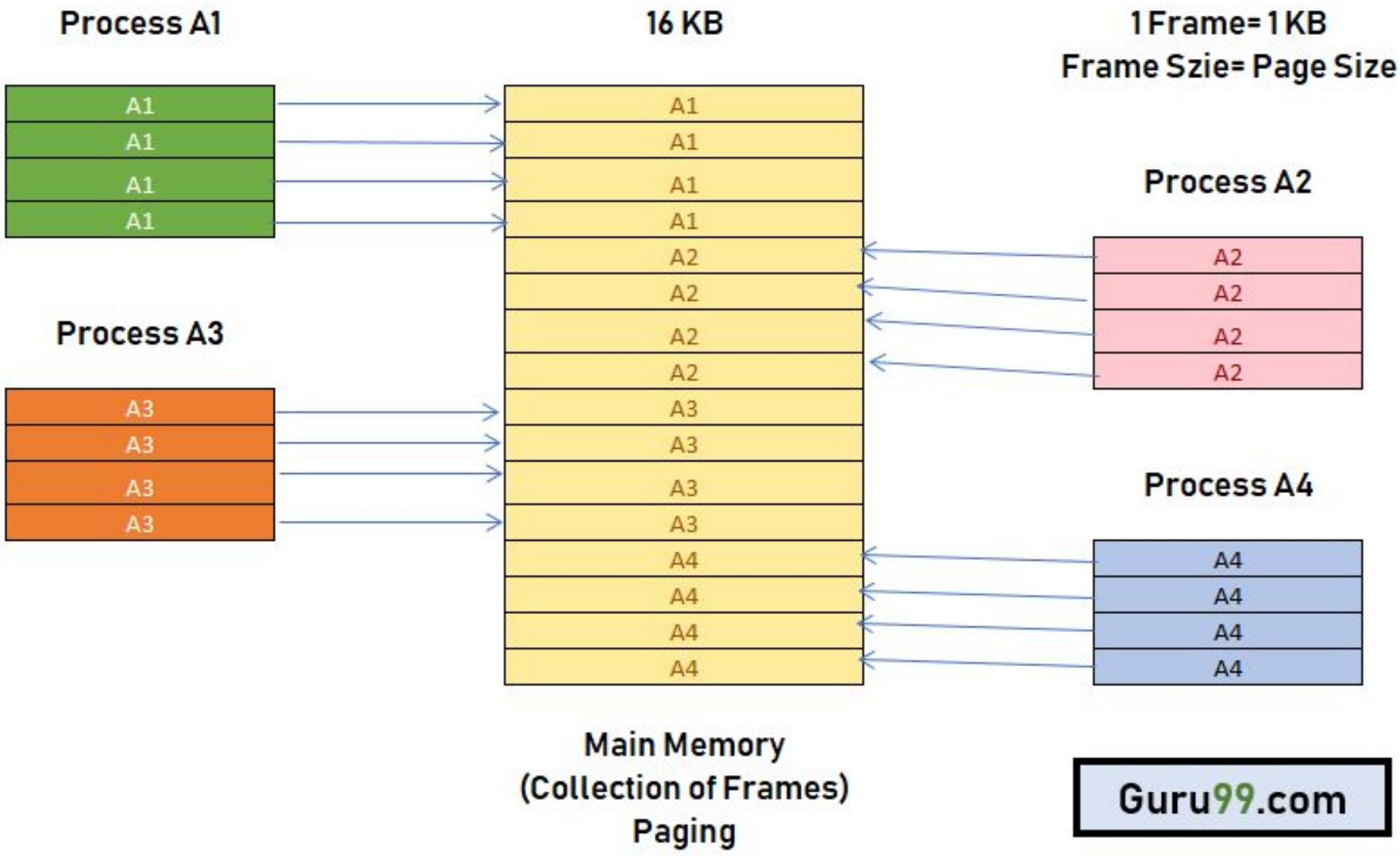


- Each process is divided into parts where size of each part is same as page size.
- The size of the last part may be less than the page size.
- Depending upon the availability, these pages may be stored in the main memory frames in a non-contiguous fashion.

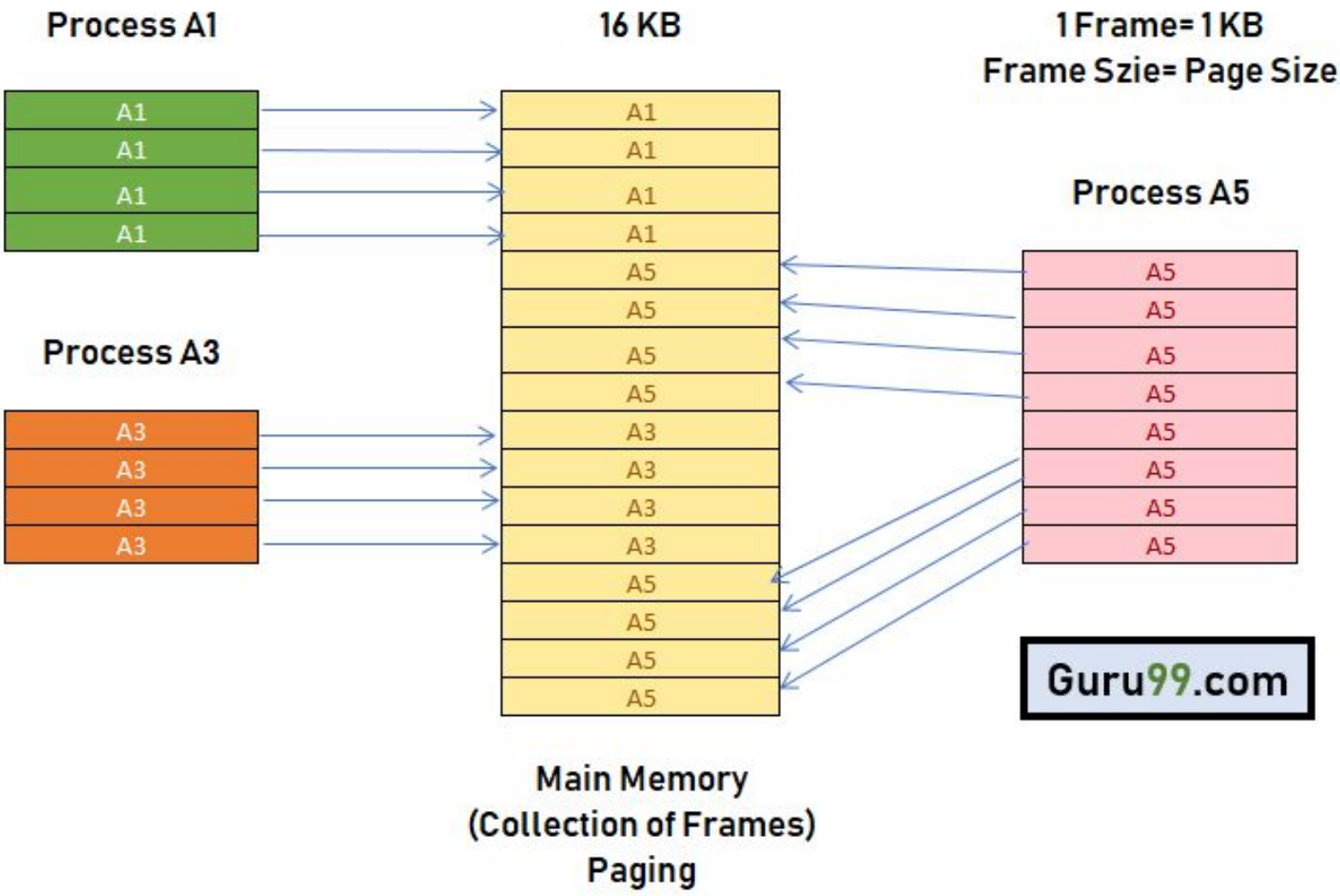
Advantages

- Paging avoids external fragmentation whereas segmentation does not.
- It allows to store parts of a single process in a non-contiguous fashion.

Paging Example



Paging Example (contd....)

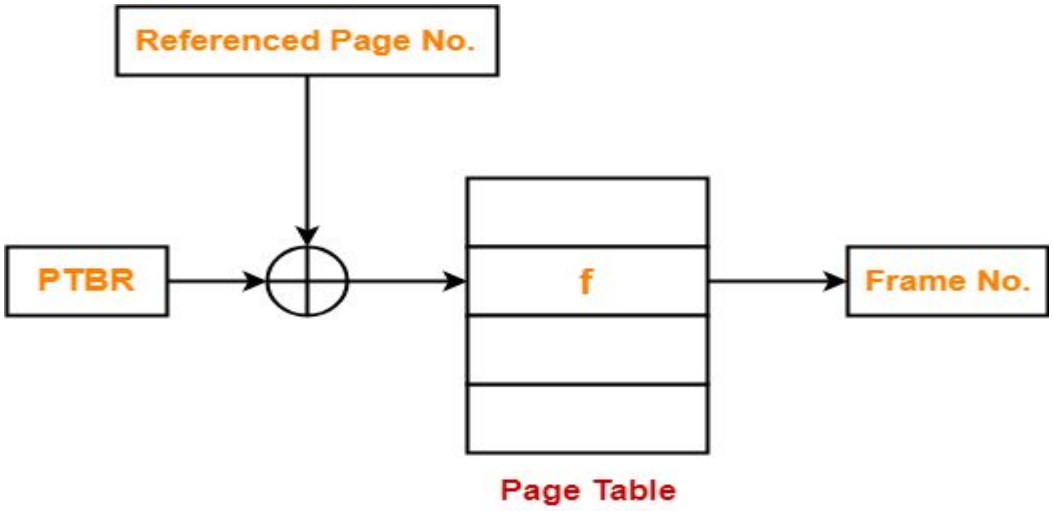


Page Table

A **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses.

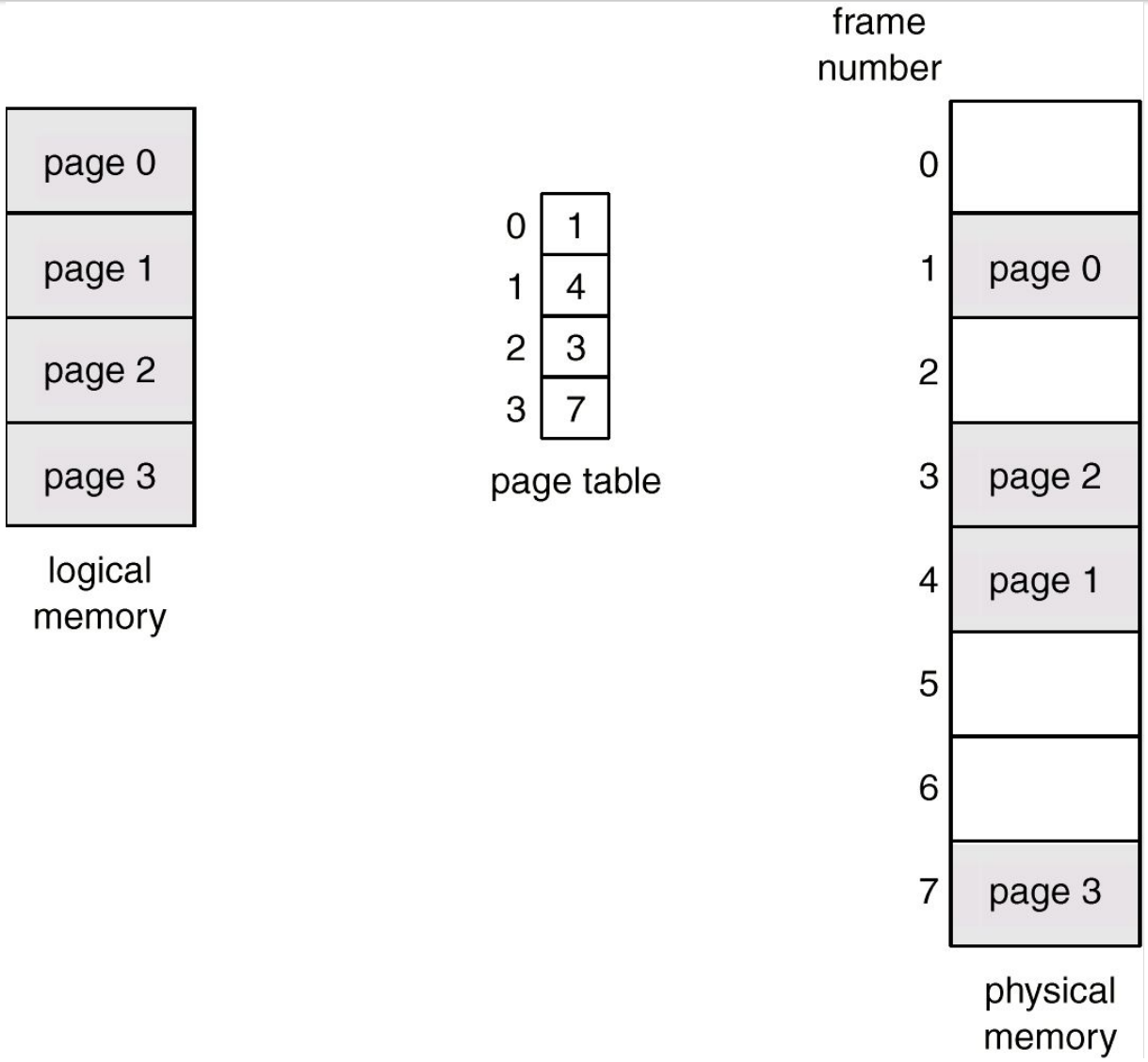
Number of entries in a page table = Number of pages in which the process is divided.

Each process has its own independent page table.



Obtaining Frame Number Using Page Table

Paging Example

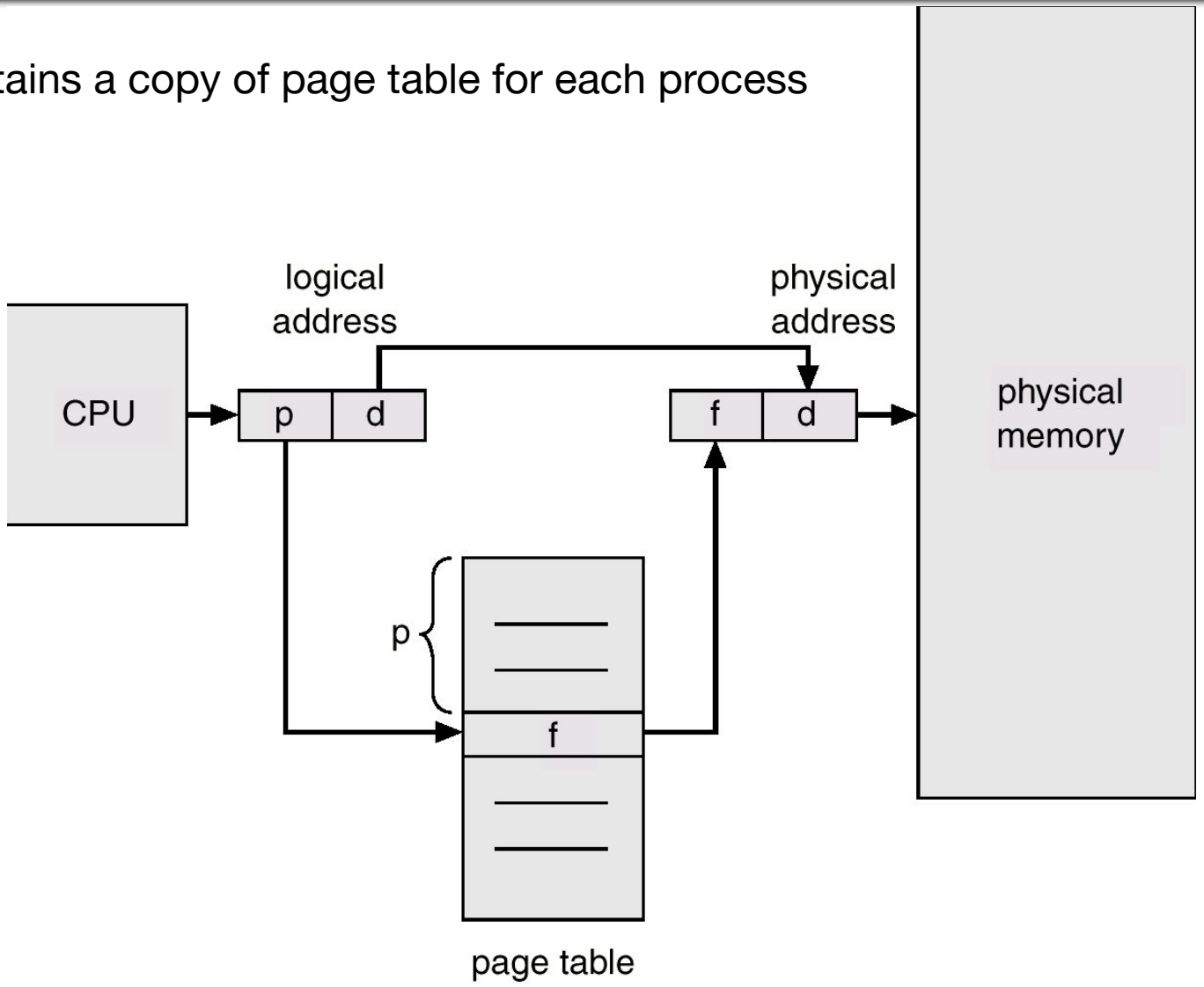


Address Translation Scheme

- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – specifies the specific word on the page that CPU wants to read.
- Page Table provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.
- Page offset combined with base address to define the physical memory address that is sent to the memory unit.
- Average Internal fragmentation in paging is one-half page

Address Translation Architecture

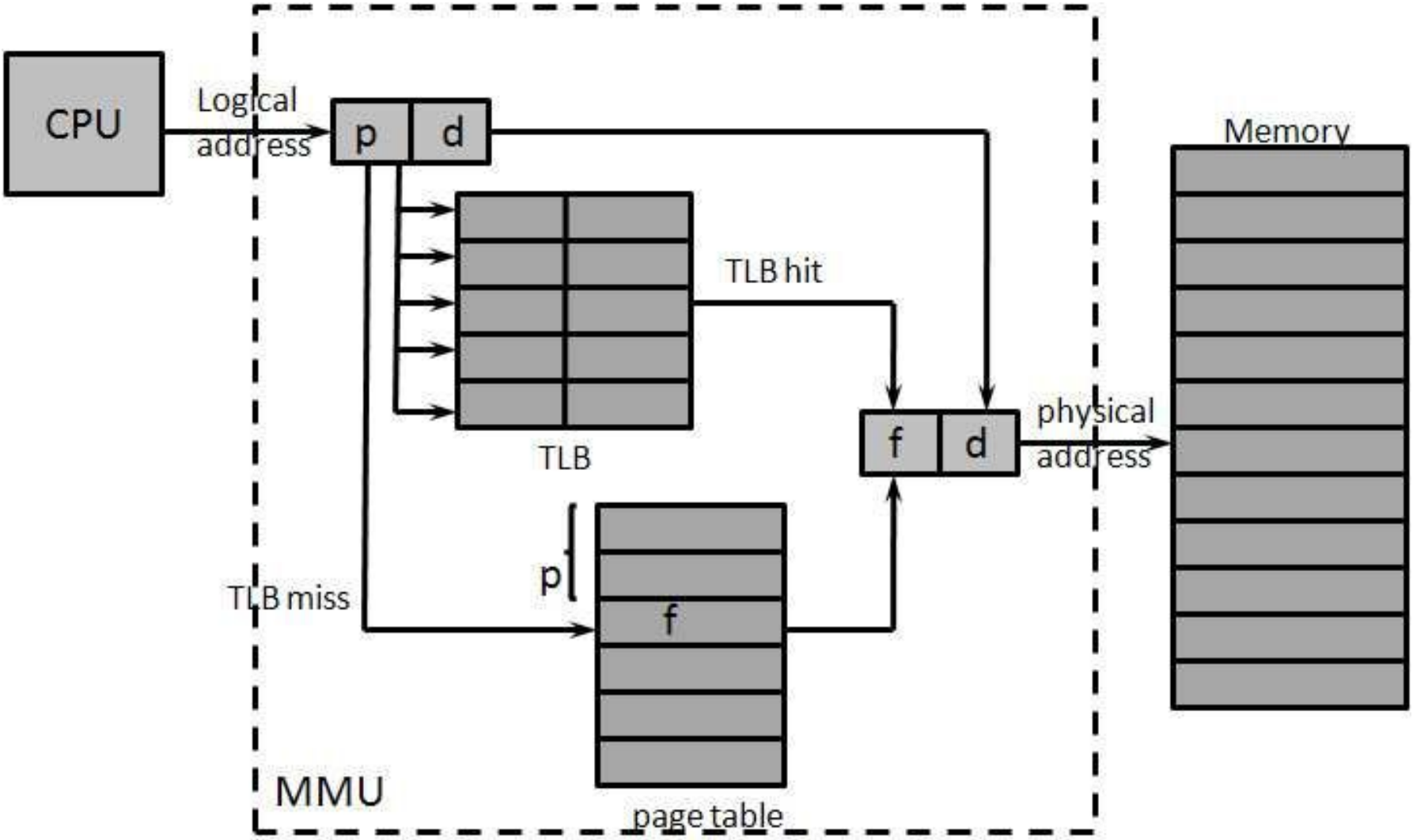
OS maintains a copy of page table for each process



Translation lookaside buffer or TLB.

- Page table information is used by the MMU for every read and write access, so ideally page table should be situated within the MMU.
- But due to its large size, it is impossible to include a complete page table on the processor chip. Therefore the page table is kept in the main memory.
- A copy of the small portion of the most recently used page table entries are cached in MMU to optimize address translation. This cache is commonly called a ***translation lookaside buffer*** or TLB.

Address translation using TLB

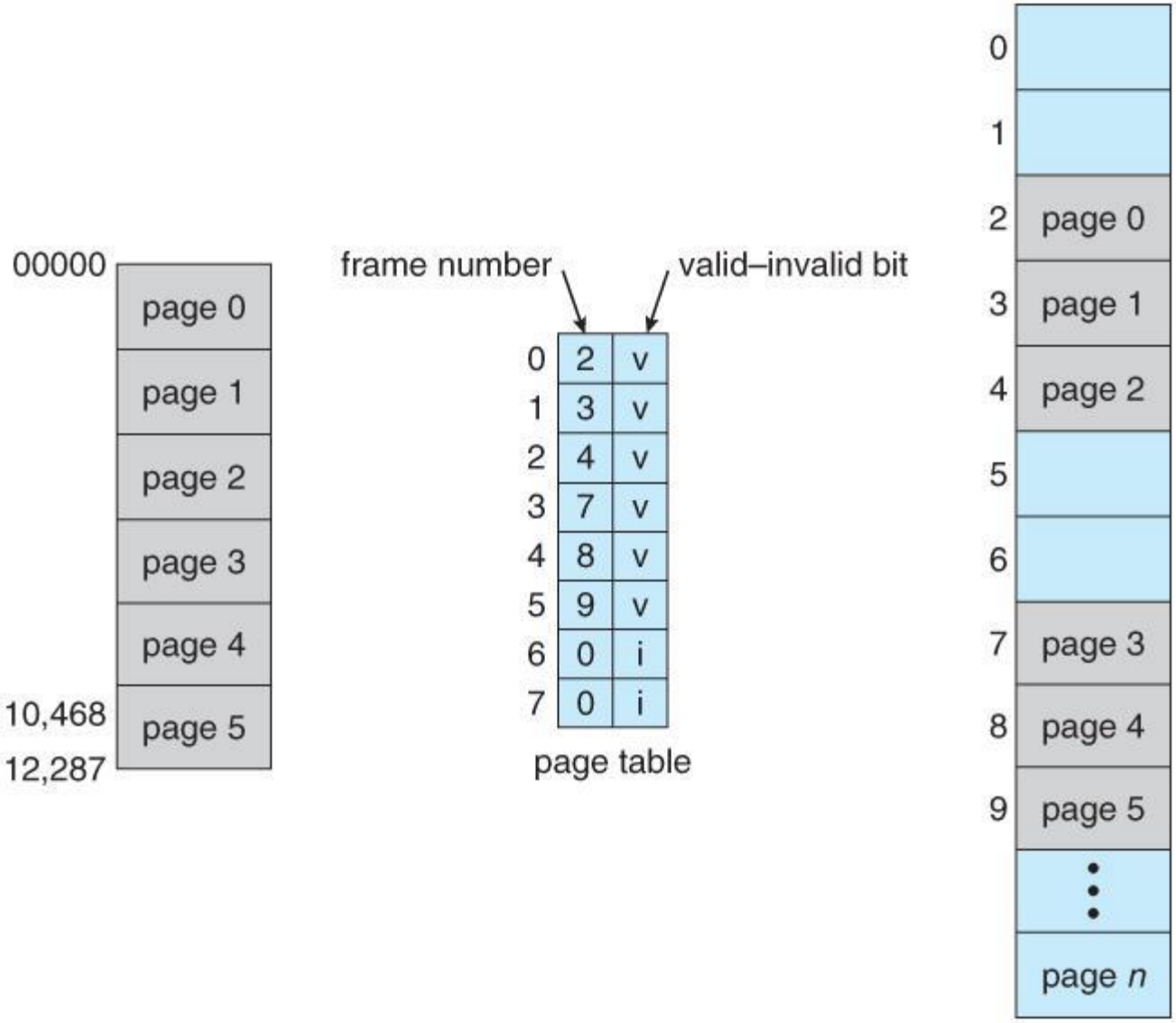


Address translation procedure using TLB

1. Given a virtual address, the MMU looks in the TLB, the physical address is obtained immediately.
2. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.
3. If requested page is not in main memory, a page fault occurred. The whole page must be brought from disk into the memory before access can proceed.
4. The OS then copies the requested page from the disk into the main memory.
5. Because a long delay occurs while the page transfer takes place, OS may suspend execution of the task that caused page fault and begin execution of another task whose pages are in main memory.

Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.



The disadvantages of paging are-

- It suffers from internal fragmentation.
- There is an overhead of maintaining a page table for each process.
- The time taken to fetch the instruction increases since now two memory accesses are required.

Logical Address in paging scheme

- If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order ***m–n*** bits of a logical address designate the page number, and the ***n*** low-order bits designate the page offset. Thus, the logical address is:

Example

1. Consider a logical address space of 32 pages with 1,024 words per page, mapped onto a physical memory of 16 frames.

Calculate:

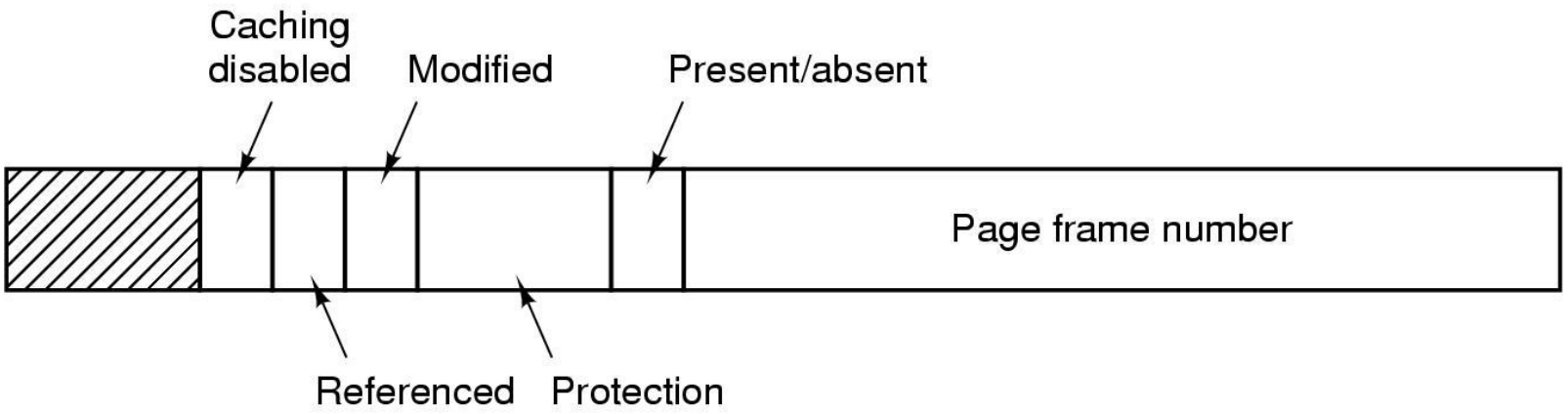
i) How many bits are required in the logical address?

ii) How many bits are required in the physical address?

2. Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a) 3085 b) 42095 c) 215201 d) 650000 e) 2000001

Structure of Page Table Entry



- Modified (dirty) bit: 1 means written to => have to write it to disk.
  0 means don't have to write to disk.
- Referenced bit: 1 means it was either read or written. Used to pick page to evict. Don't want to get rid of page which is being used.
- Present (1) / Absent (0) bit
- Protection bits: r, w, r/w

.

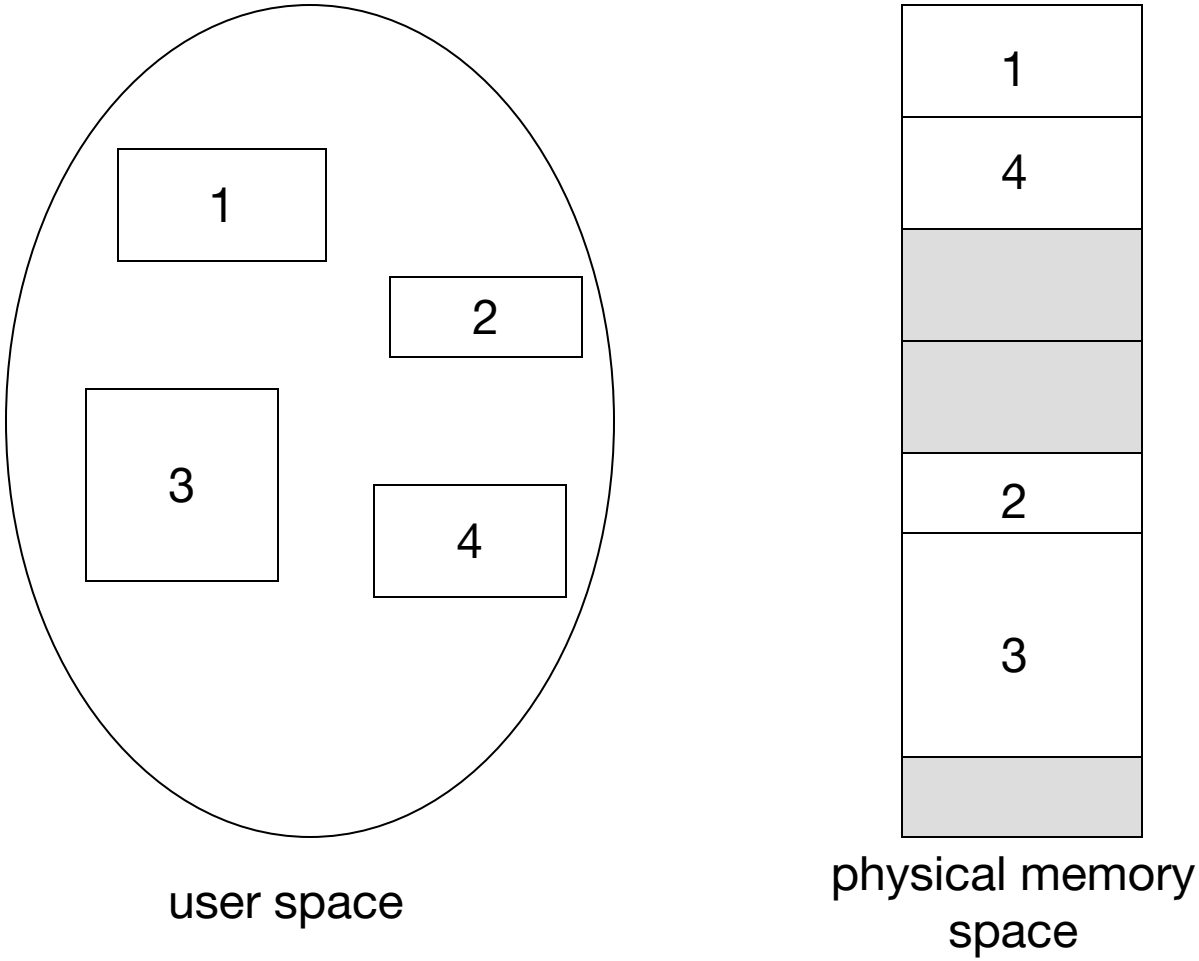
Implementation of Page Table

- Page table is kept in main memory.
- *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers (TLBs)*

Segmentation

- Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.
- Segments vary in length, and length of each is defined by its purpose in the program
- Address mapping is done by a segment table. Each entry in the segment table has a segment base and a segment limit.
- The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.
- Logical address consists of two parts: segment number, s, offset, 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, trap to OS (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Logical View of Segmentation



Memory segmentation

- Memory **segmentation** is the division of a **computer**'s primary memory into **segments** or sections.
- Memory addresses consist of a segment id and an offset within the segment.
- A hardware memory management unit (MMU) is responsible for translating the segment and offset into a physical address, and for performing checks to make sure the translation can be done and that the reference to that segment and offset is permitted.
- Each segment has a length and set of permissions (for example, *read*, *write*, *execute*) associated with it. A process is only allowed to make a reference into a segment if the type of reference is allowed by the permissions, and if the offset within the segment is within the range specified by the length of the segment. Otherwise, a hardware exception such as a segmentation fault is raised.

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

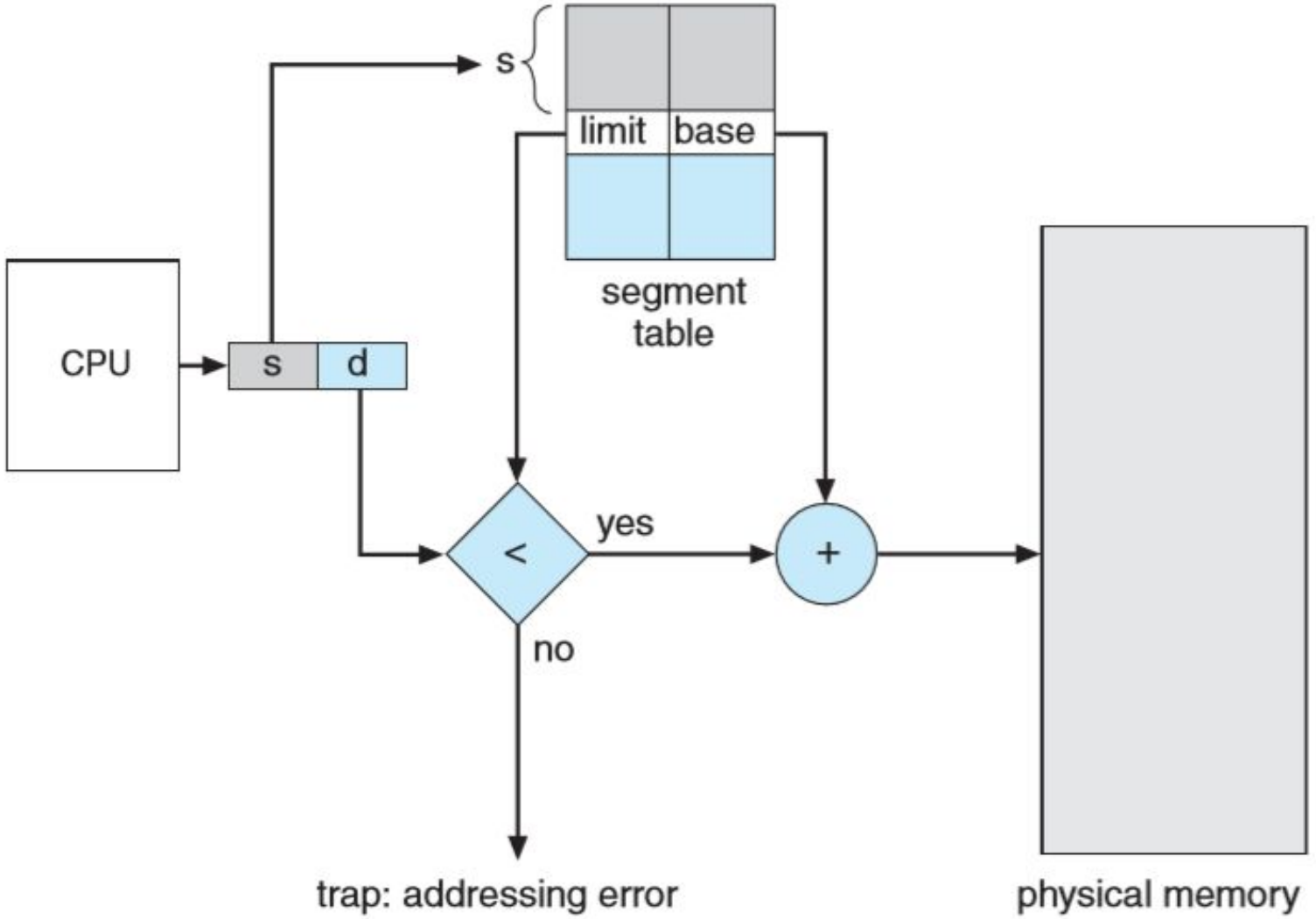
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.

- *Segment-table base register (STBR)* points to the segment table's location in memory.

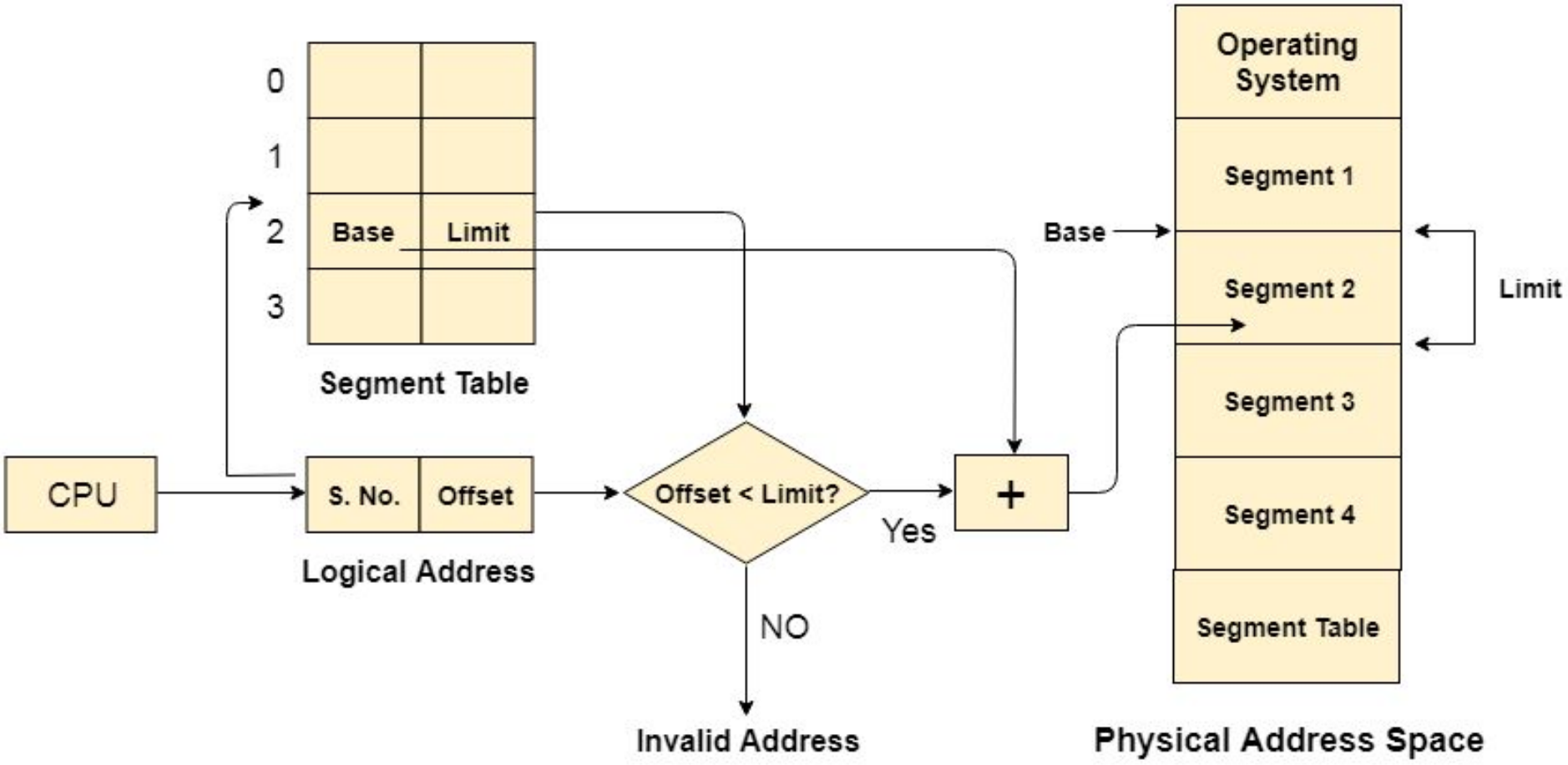
- *Segment-table length register (STLR)* indicates number of segments used by a program;

  segment number *s* is legal if *s* < STLR.

Segmentation hardware



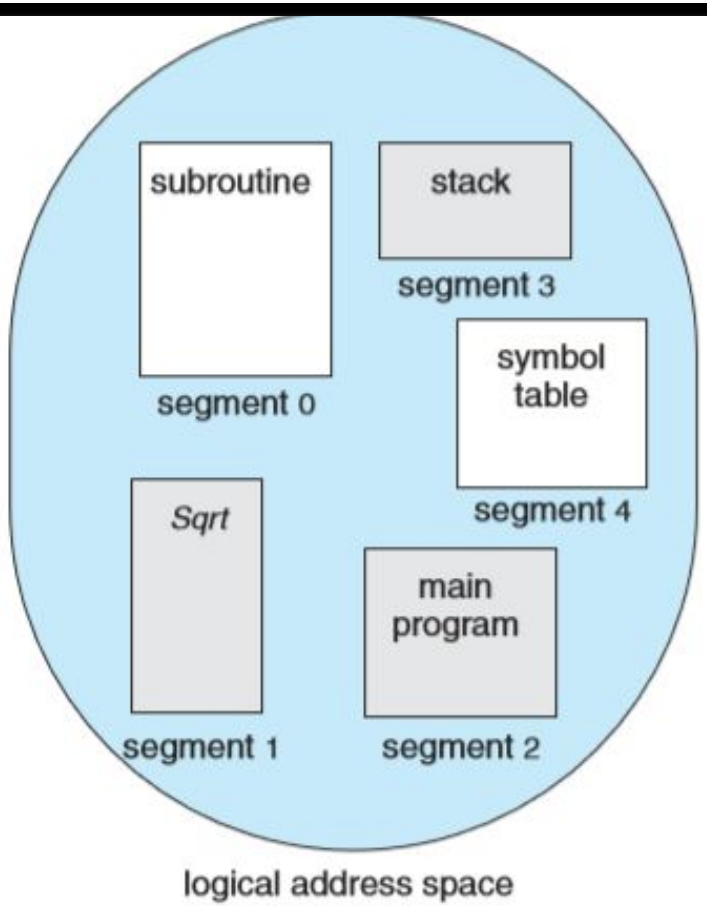
Translation of Logical address into physical address by segment table



Segmentation Architecture (Cont.)

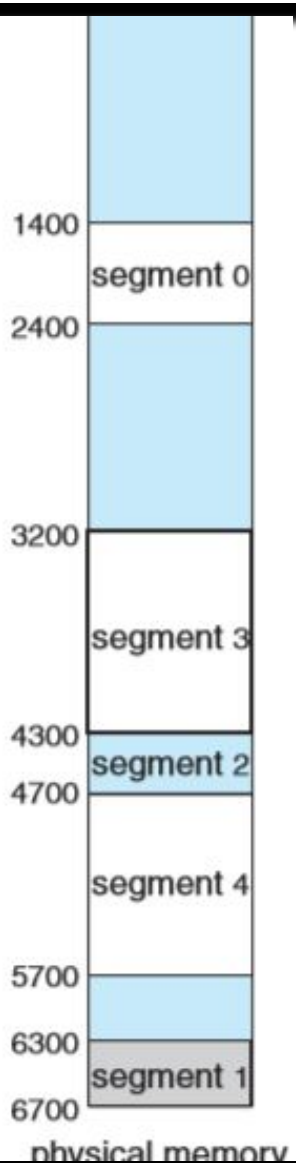
- Protection. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.
- A segmentation example is shown in the following diagram

Example of segmentation

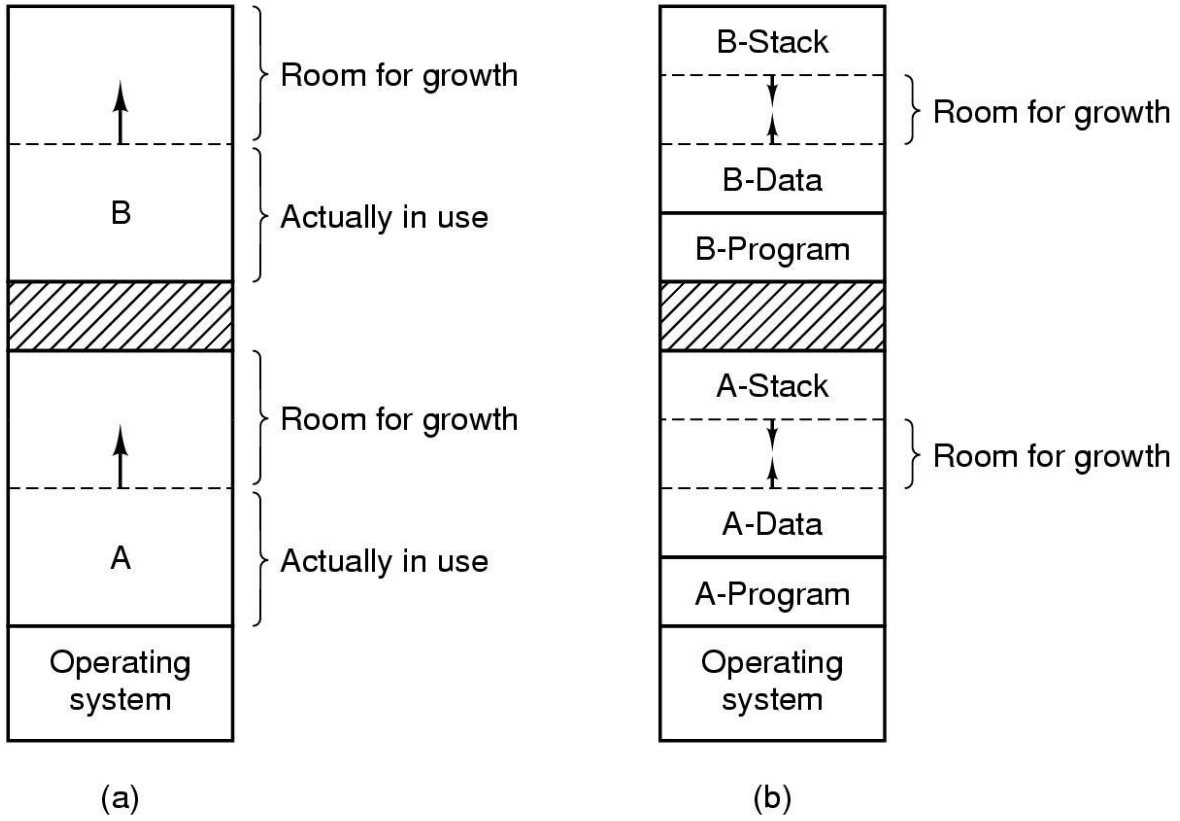


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



2 ways to allocate space for growth



(a) Just add extra space

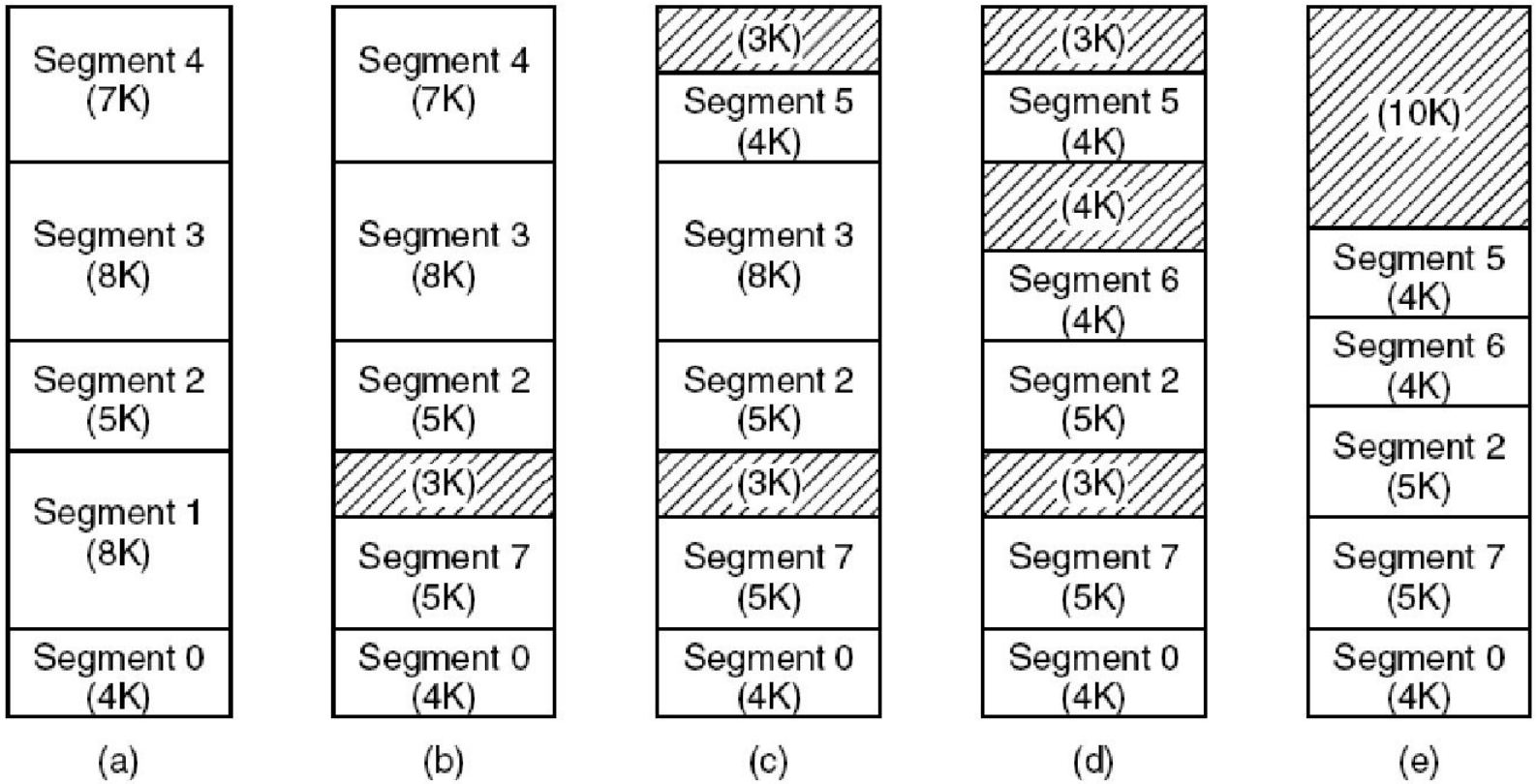
(b) Stack grows downwards, data grows upwards

Logical to physical address translation

Sample Problem

- A reference to byte 53 of segment 2 is mapped onto location 4300 (the base of segment 2) + 53 = 4300 +53=4353.
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to OS, as this segment is only 1,000 bytes long.

External fragmentation



(a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

Logical to physical address translation

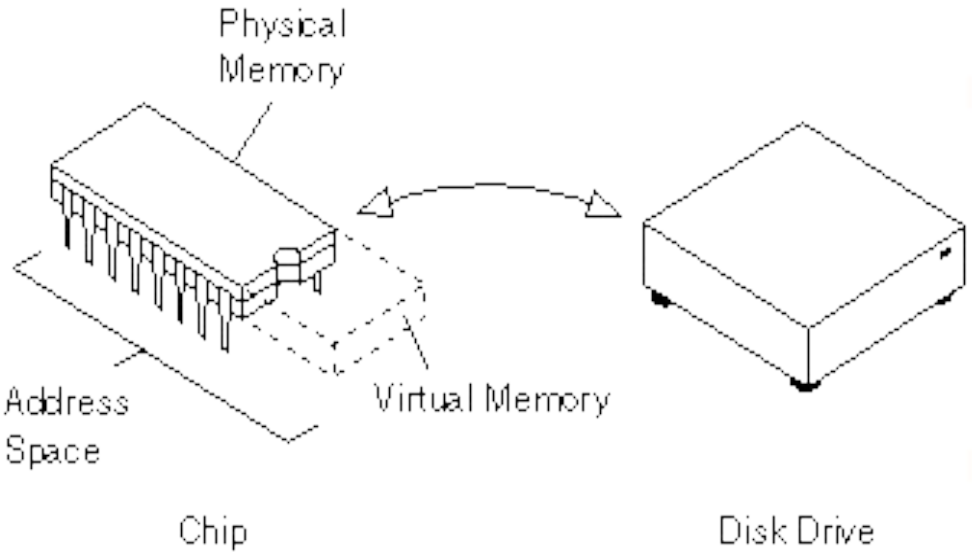
- Consider the following segment table:

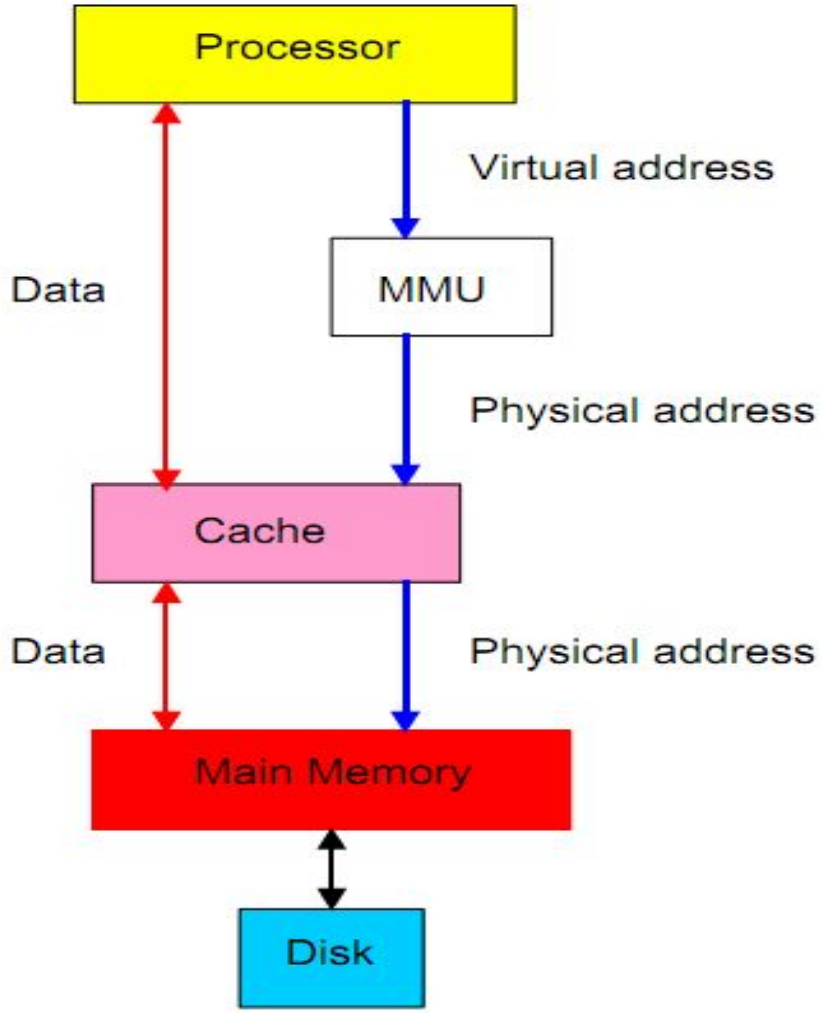
Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

- What are the physical addresses for the following logical addresses?

a) 0,430 b) 1,10 c) 2,500 d) 3,400 e) 4,112

Virtual memory organization





Virtual Memory

- Virtual memory is used to increase the apparent size of physical memory.
- Virtual memory combines computer's RAM with temporary space on your hard disk.
- Virtual memory is a feature of OS that enables a process to use a memory (RAM) address space that is independent of other processes running in the same system, and use a space that is larger than the actual amount of RAM present, temporarily transferring some contents from RAM to a disk, with little or no overhead.
- Virtual memory enables each process to act as if it has whole memory space to itself.
- Data may be stored in physical memory locations that have address different from those specified by the program.

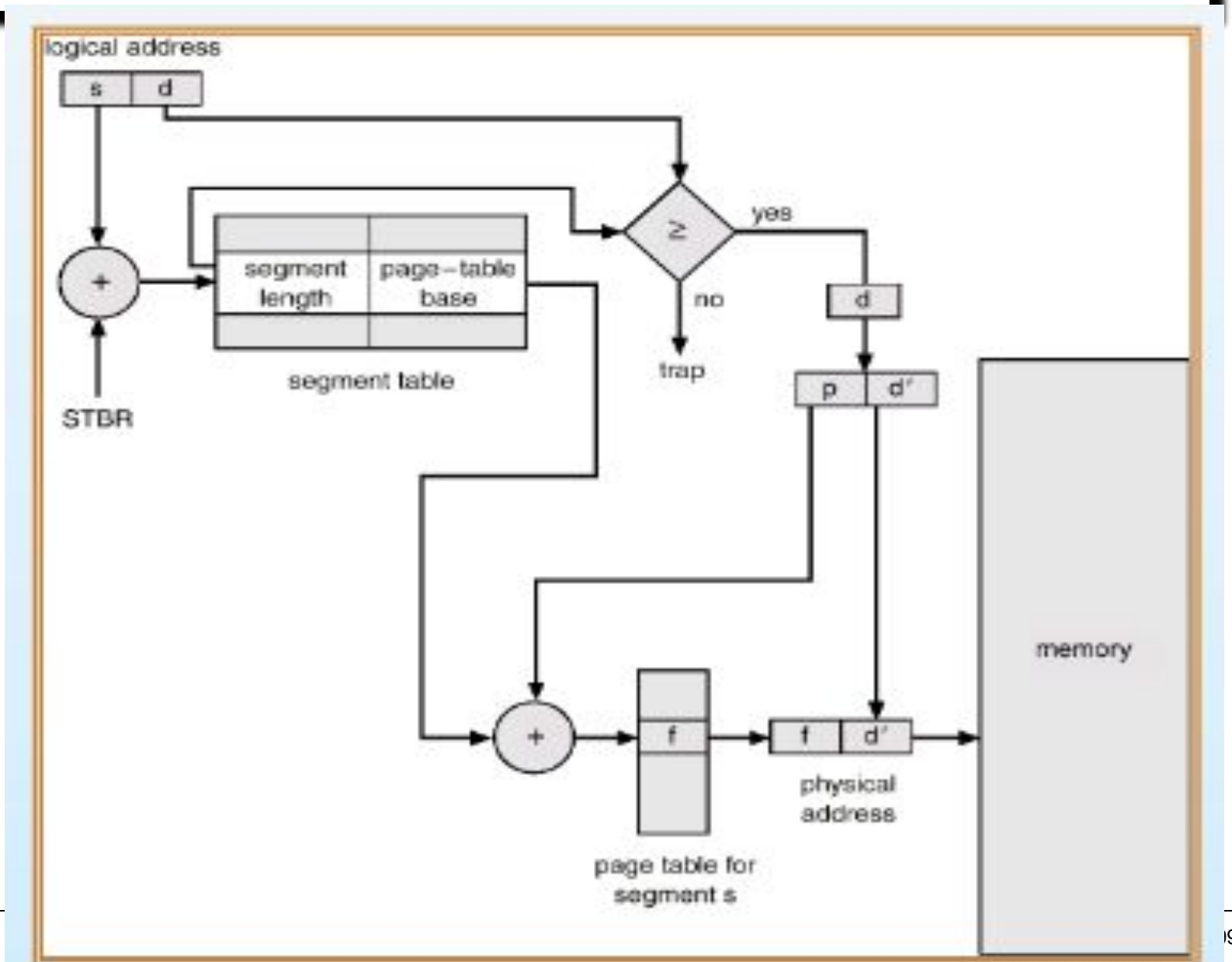
Virtual Memory

- Program's address space is broken up into fixed size pages
- Pages are mapped to physical memory
- If instruction refers to a page in memory, fine
- Otherwise OS gets the page, reads it in, and re-starts the instruction
- While page is being read in, another process gets the CPU

Segmentation with paging

- An implementation of virtual memory on a system using segmentation without paging requires that entire segments be swapped back and forth between main memory and secondary storage. When a segment is swapped in, the operating system has to allocate enough contiguous free memory to hold the entire segment.
- Often memory fragmentation results if there is not enough contiguous memory even though there may be enough in total.
- In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.
- Using segmentation with paging usually only moves individual pages back and forth between main memory and secondary storage, similar to a paged non-segmented system.
- Pages of the segment can be located anywhere in main memory and need not be contiguous. This usually results in a reduced amount of input/output between primary and secondary storage and reduced

Segmentation with Paging –Address Translation Scheme



Inverted Page Table

- A single global page table.
- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- The physical page number is not stored, since the index in the table corresponds to it
- 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

Inverted Page Table

- Virtual address in the system consists of a triple:

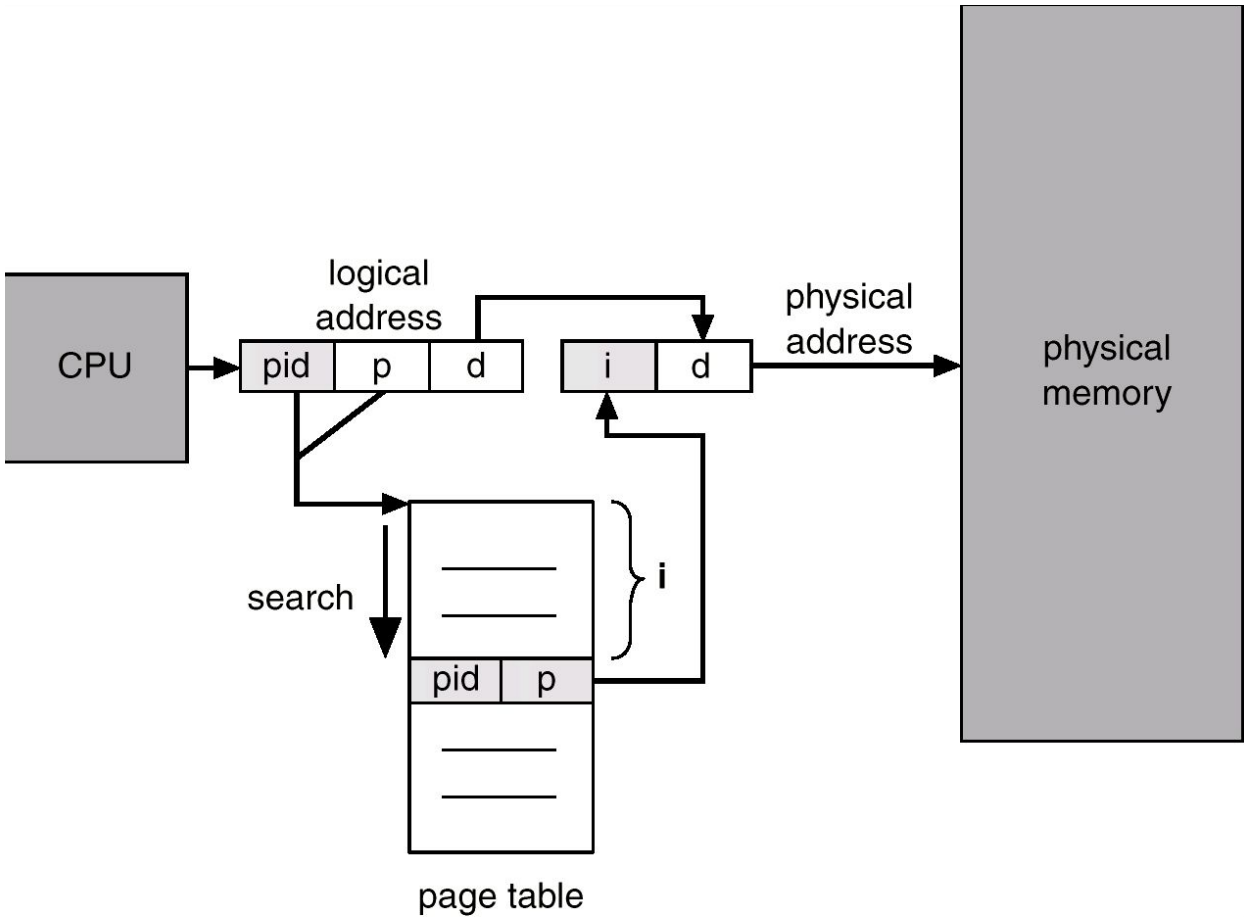
  <process-id, page-number, offset>.

- Each inverted page-table entry is a pair:<process-id, page-number> where the 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 has been attempted.

Inverted Page Table Architecture



Demand Paging

- Pages should only be brought into memory if the executing process demands them.
- This is often referred to as lazy evaluation as only those pages demanded by the process are swapped from secondary storage to main memory.
- To achieve this process a page table implementation is used. The page table maps logical memory to physical memory.
- The page table uses a bitwise operator to mark if a page is valid or invalid.
- A valid page is one that currently resides in main memory.
- An invalid page is one that currently resides in secondary memory.

Steps involved in demand paging

When a process tries to access a page, the following steps are generally followed:

- Attempt to access page.
- If page is valid (in memory) then continue processing instruction as normal.
- If page is invalid then a **page-fault trap** occurs.
- Schedule disk operation to read the desired page into main memory.
- Restart the instruction that was interrupted by the operating system trap.

Advantages

- Demand paging, as opposed to loading all pages immediately:
- Only loads pages that are demanded by the executing process.
- As there is more space in main memory, more processes can be loaded, reducing the context switching time, which utilizes large amounts of resources.
- Less loading latency occurs at program startup, as less information is accessed from secondary storage and less information is brought into main memory.

As main memory is expensive compared to secondary memory, this technique helps significantly reduce the bill of material (BOM) cost in smart phones.

Disadvantages

- Individual programs face extra latency when they access a page for first time.
- Low-cost, low-power embedded systems may not have a memory management unit that supports page replacement.
- Memory management with page replacement algorithms becomes slightly more complex.
- Thrashing which may occur due to repeated page faults.

Paging Vs Segmentation

Sno.	Paging	Segmentation
1	Block replacement easy Fixed-length blocks	Block replacement hard Variable-length blocks Need to find contiguous, variable-sized, unused part of main memory
2	Invisible to application programmer	Visible to application programmer.
3	No external fragmentation, But there is Internal Fragmentation unused portion of page.	No Internal Fragmentation, But there is external Fragmentation unused portion of main memory.
4	Units of code and date are broken into separate pages.	Keeps blocks of code or data as a single units.
5	paging is a physical unit invisible to the user's view and is of fixed size	segmentation is a logical unit visible to the user's program and id of arbitrary size
6	Paging maintains one address space.	Segmentation maintains multiple address spaces per process.

Page Replacement Algorithms

Introduction

- When a page fault occurs and no free page frames available, then OS has to choose a page to remove from memory to make room for the page that has to be brought in.
- The page replacement is done by swapping the required pages from backup storage to main memory and vice-versa.
- If the page to be removed has been modified while in memory, it must be rewritten to disk to bring disk copy up to date.
- If, the page has not been changed (e.g., it contains program text), disk copy is already up to date, so no rewrite is needed.
- The page to be read in just overwrites the page being evicted.

Page Replacement Algorithms

- If new page is brought in, need to chose a page to evict
- Don't want to evict heavily used pages
- Each operating system uses different page replacement algorithms.
- A page replacement algorithm is evaluated by running the particular algorithm on a string of memory references (references string) and compute the page faults.
- To select the particular algorithm, the algorithm with lowest page fault rate is considered.

Page Replacement Algorithms

- First-in, first-out page replacement
- Not recently used page replacement
- Least recently used page replacement
- Optimal page replacement algorithm
- Second chance page replacement
- Clock page replacement
- Working set page replacement
- WSClock page replacement

FIFO

- The oldest page in the physical memory is the one selected for replacement.
- Very simple to implement. - Keep a list
- On a page fault, the page at the head is removed and the new page added to the tail of the list

Issues

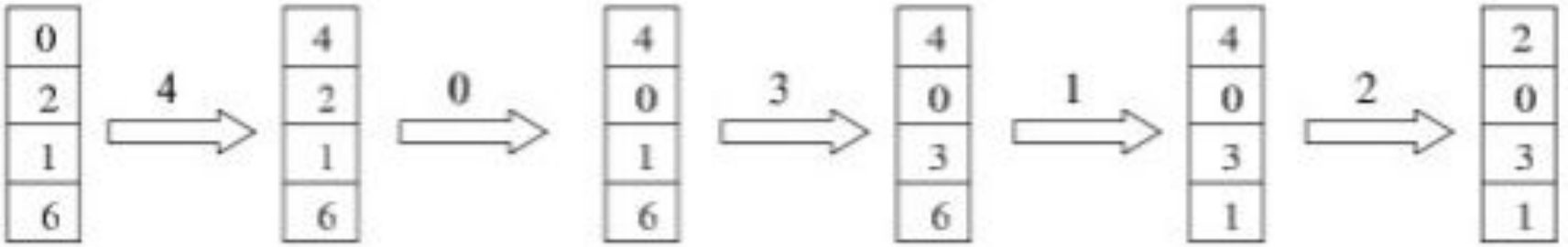
- Poor replacement policy
- Oldest might be most heavily used! No knowledge of use is included in FIFO.
- Doesn't consider the page usage.

FIFO Example with 4 memory frames

- Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

X X X X X X X X X

Compulsory Misses



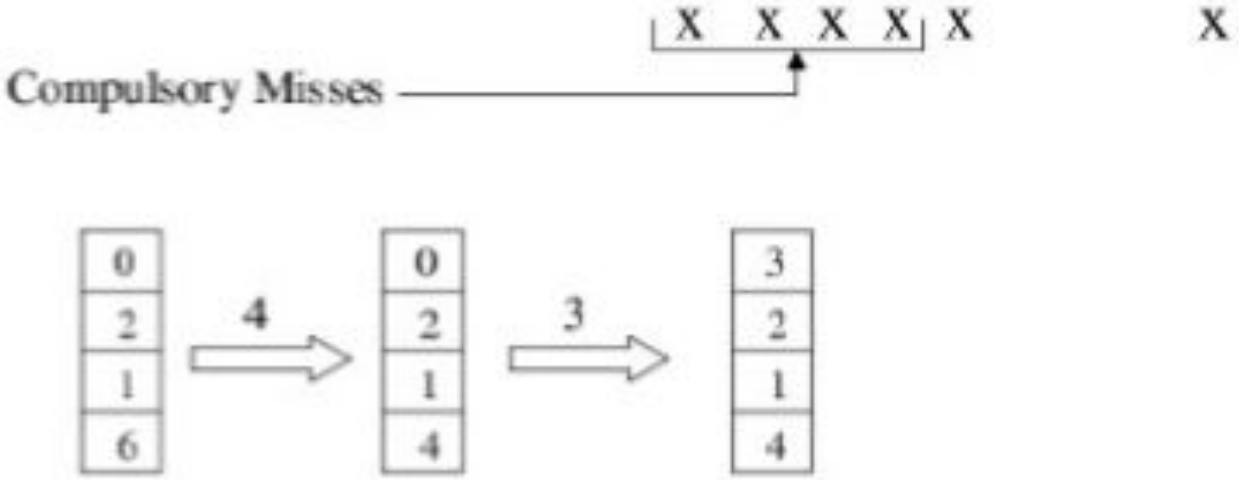
- Fault Rate = 9 / 12 = 0.75

Optimal Page Replacement

- Pages are replaced which would not be used for the longest duration of time in the future.
- Not possible unless know when pages will be referenced (crystal ball)
- Used as ideal reference algorithm
- Difficult to implement, because it requires future knowledge of the reference string.
- Mainly used for comparison studies
- The algorithm has lowest page fault rate of all algorithm

Optimal Page Replacement

- Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1



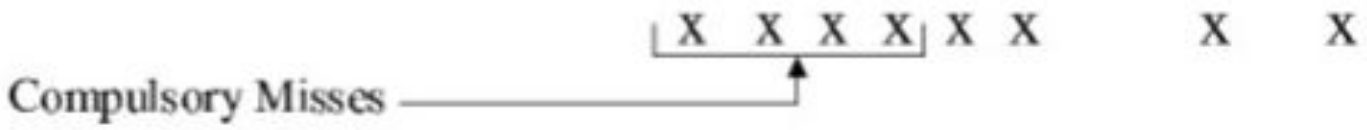
- Fault Rate = 6 / 12 = 0.50
- With the above reference string, this is the best we can hope to do

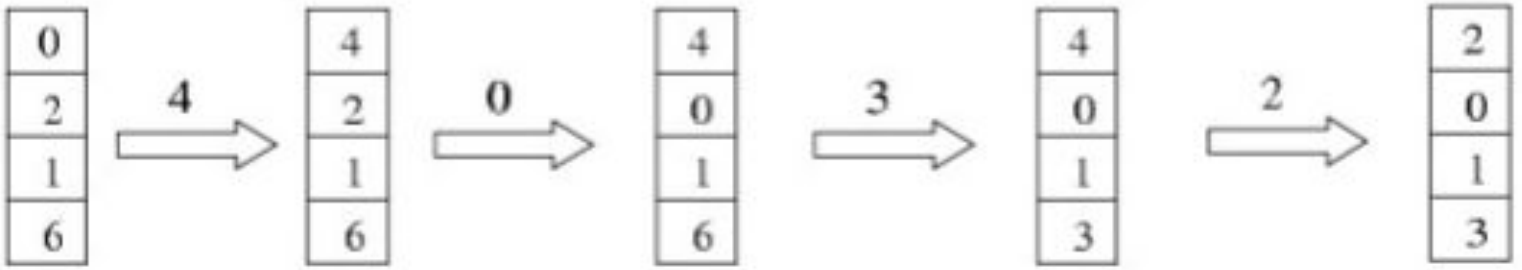
Least Recently Used: LRU

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- Approximate LRU by assuming that recent page usage approximates long term page usage
- Could associate counters with each page and examine them but this is expensive
- Maintain a linked list of all pages in memory with the MRU page at front and LRU page at the rear.
- The difficulty is that the list must be updated on every memory reference.

LRU Example

- Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1





- Fault Rate = 8 / 12 = 0.67

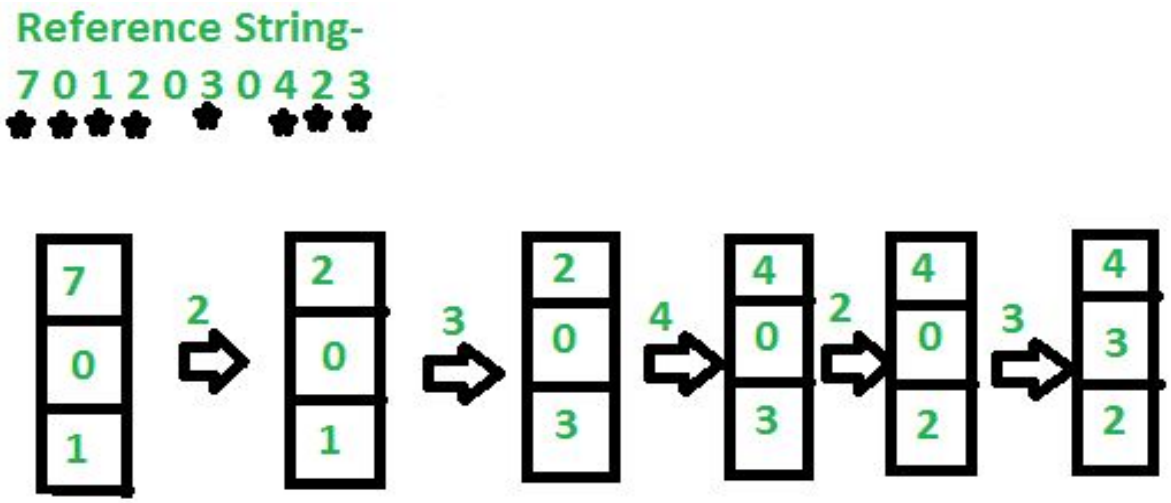
Not recently used (NRU)

- Two status bits R and M associated with each page
- R is set when a page is referenced; M is set when a page is modified.
- When a page fault occurs OS inspects all pages and divide them into following categories.
  - Class 0: not referenced, not modified
  - Class 1: not referenced, modified
  - Class 2: referenced, not modified
  - Class 3: referenced, modified
- Pick lowest priority page to evict (removes page at random with lowest class)

Not recently used (NRU): Example

From the given reference string NRU will remove a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use.

Example –



Given page reference string:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

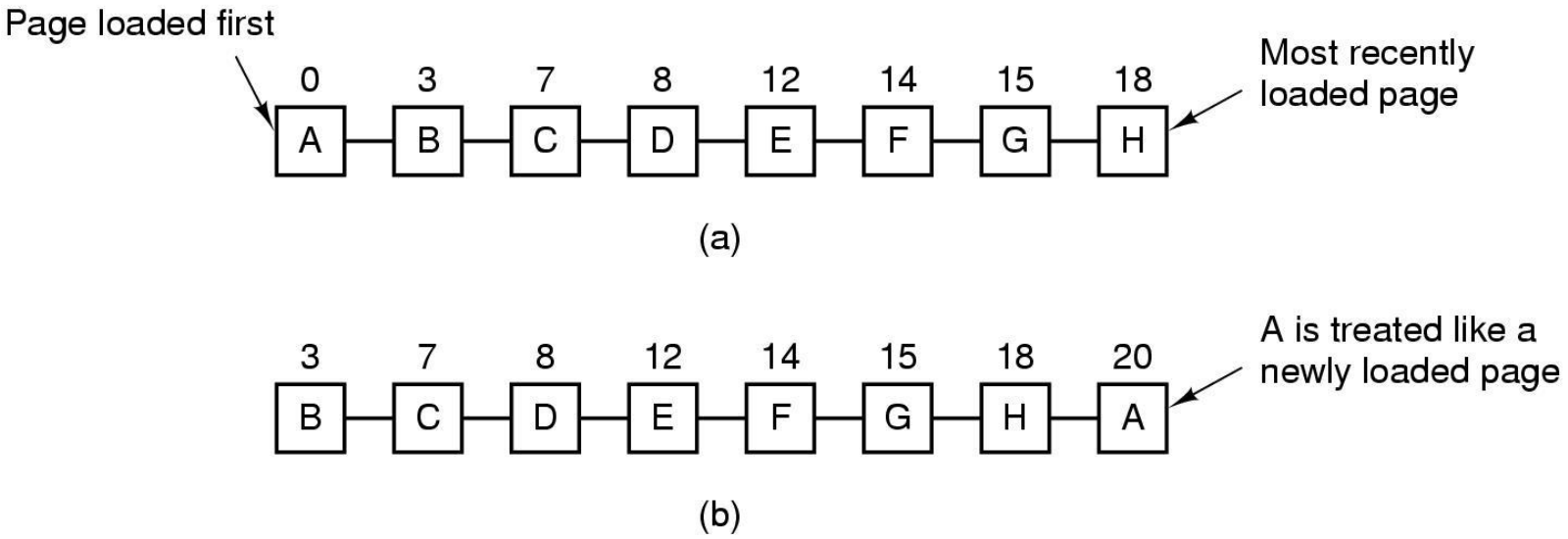
Compare the number of page faults for LRU,

FIFO and Optimal page replacement algorithm

second-chance page-replacement algorithm

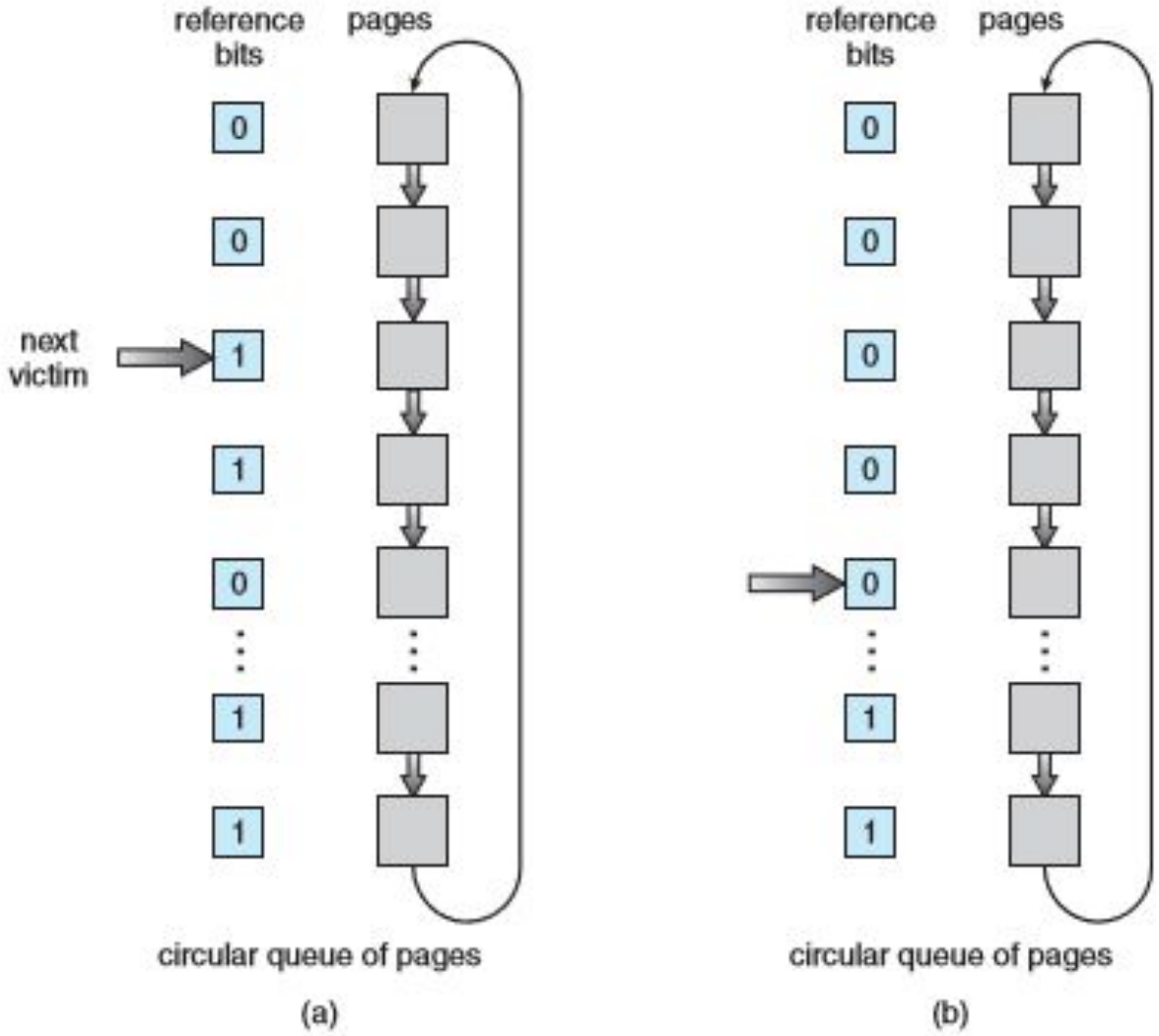
- The basic algorithm of second-chance replacement is a refinement of FIFO algorithm.
- It replaces page that is both oldest as well as unused instead of oldest page that may be heavily used.
- To keep track of page usage, it uses a reference bit (R) associated with each page and set when page is accessed.
- When a page has been selected, if its reference bit value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced

Second Chance Algorithm

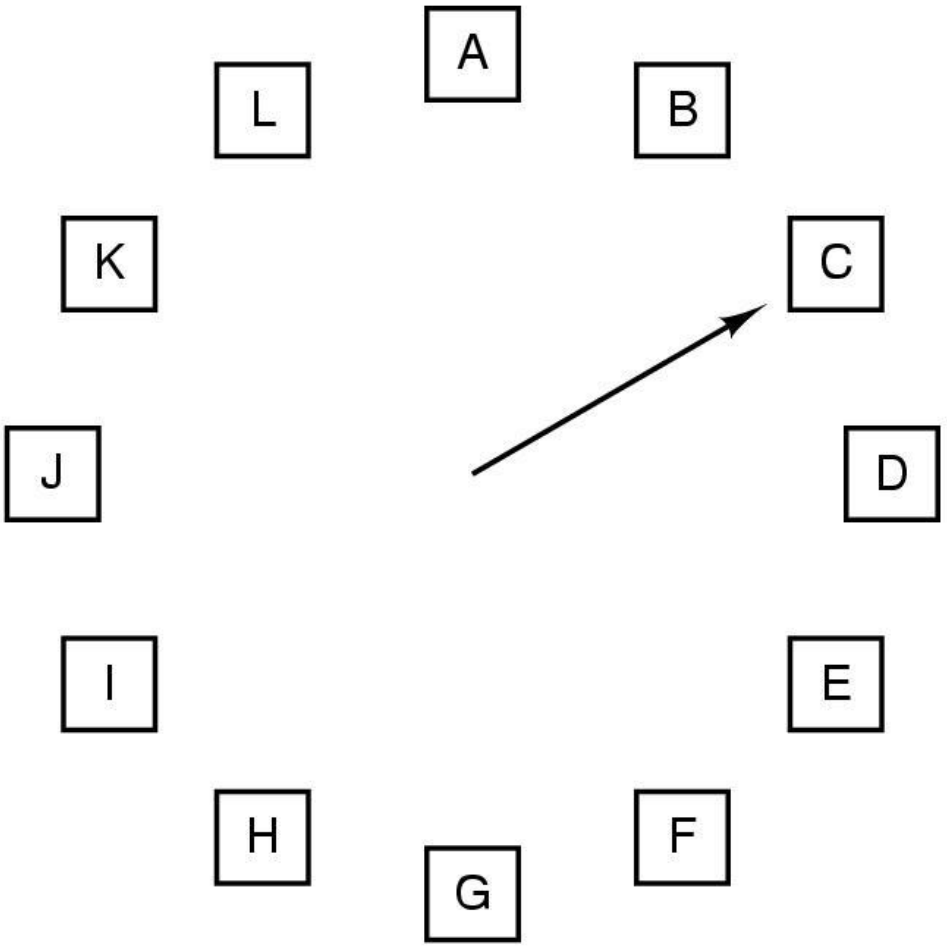


- Pages sorted in FIFO order by arrival time.
- Examine R bit. If zero, evict. If one, put page at end of list and R is set to zero.
- If change value of R bit frequently, might still evict a heavily used page

Second-chance (clock) page-replacement algorithm



Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

CLOCK

Clock is a more efficient version of FIFO than Second-chance because pages don't have to be constantly pushed to back of the list, but it performs same general function as Second-Chance.

The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location.

If R is 0, new page is put in place of the page "hand" points to, and the hand is advanced one position.

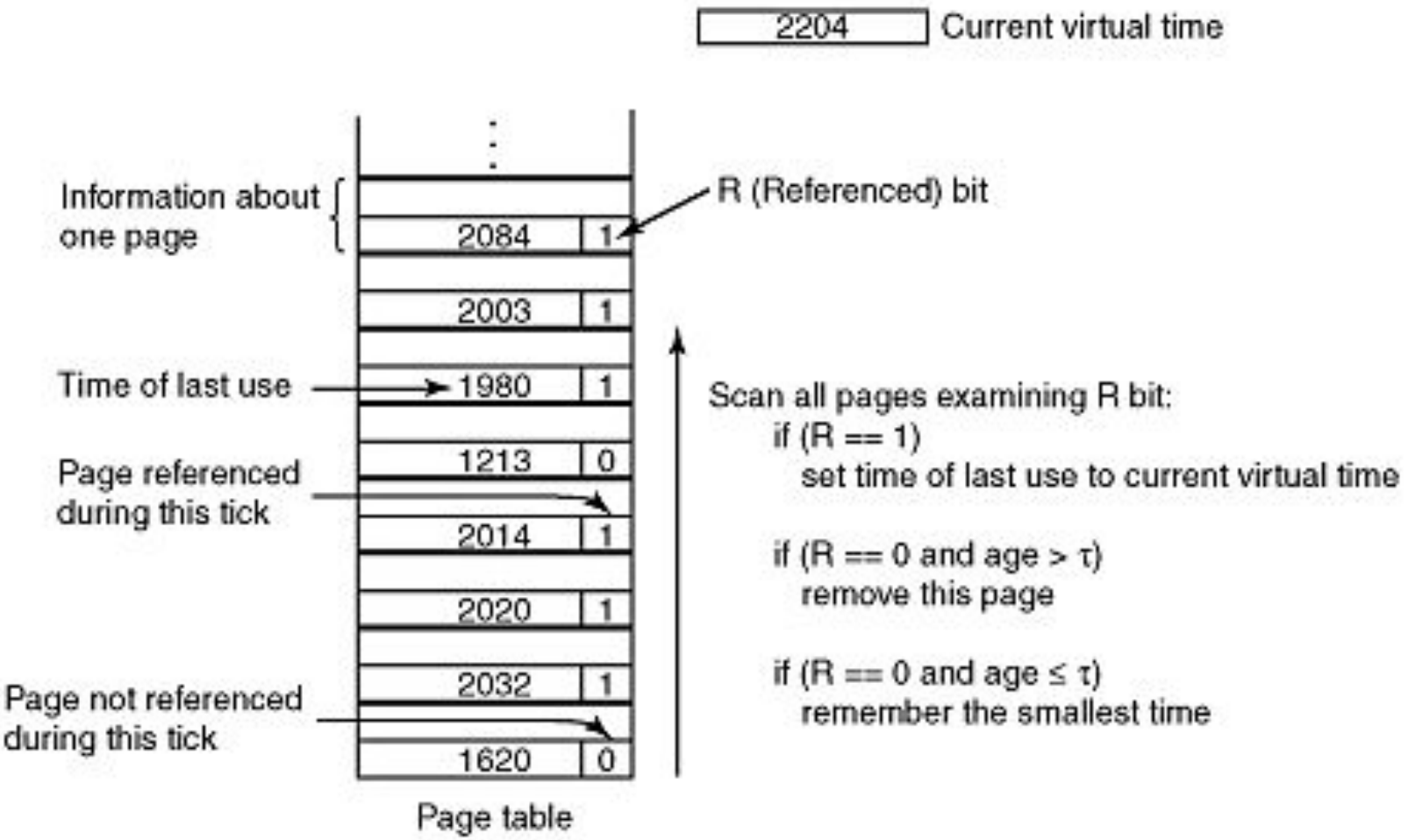
Otherwise, R bit is cleared, then clock hand is incremented and process is repeated until a page is replaced.

Thrashing

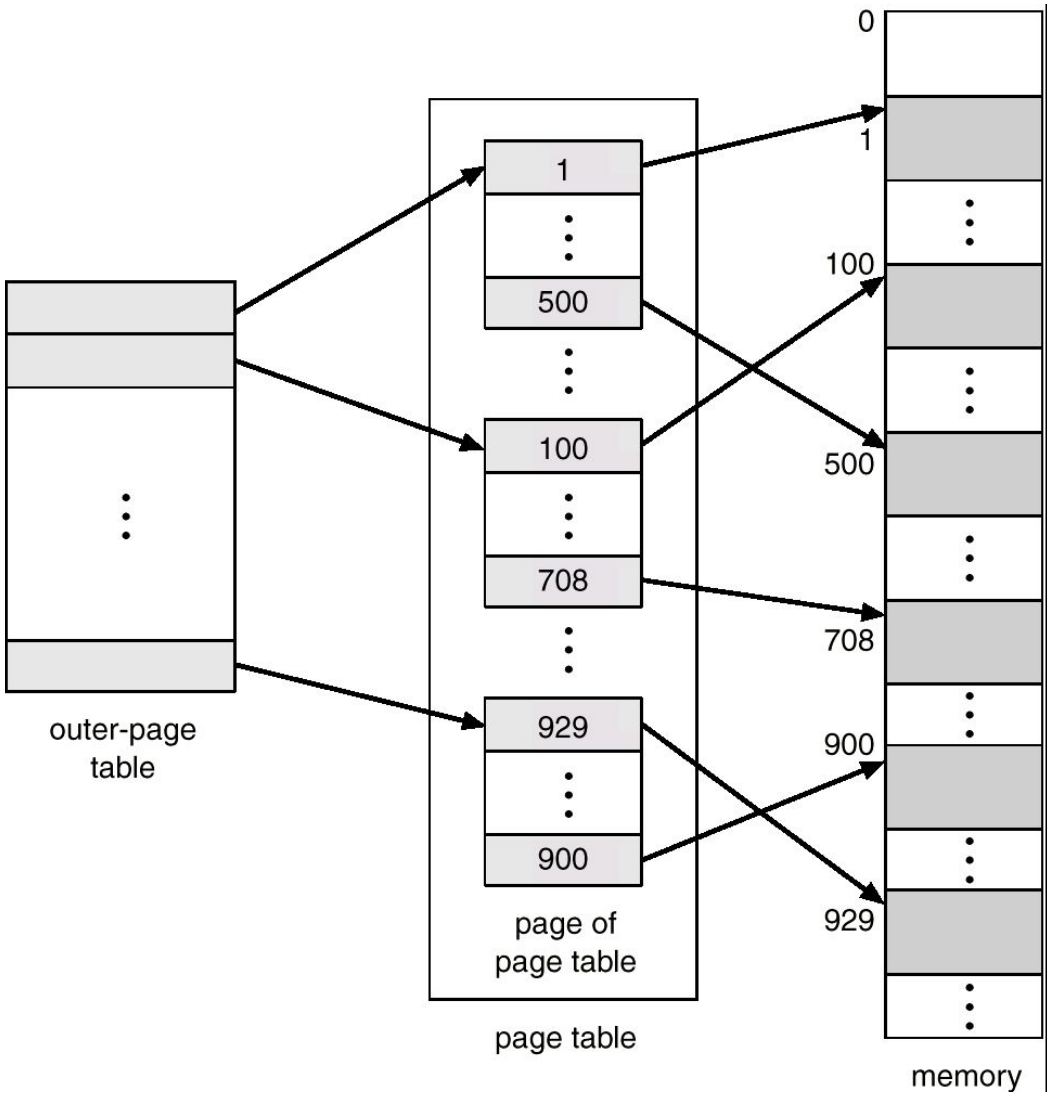
- Thrashing is excessive swapping of pages of data between memory and hard disk, causing application to respond slowly
- A process is thrashing if it is spending more time paging than executing.
- When each page in execution demands a page that is not currently in real memory (RAM) it places some pages on virtual memory and adjusts the required page on RAM.
- If the CPU is too busy in doing this task, thrashing occurs.
- 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 again, replacing pages that it must bring back in immediately.

Working Set Model

- The set of pages that a process is currently using is called its **working set**
- Demand paging-bring a process into memory by trying to execute first instruction and getting page fault. Continue until all pages that process needs to run are in memory (the working set)
- Try to make sure that working set is in memory before letting process run (pre-paging)
- Thrashing-memory is too small to contain working set, so page fault all of the time

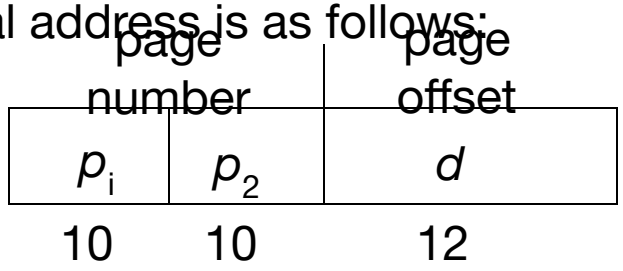

2204
Current virtual time
Information about one page
R (Referenced) bit
2084
1
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
1620
0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

Two-Level Page-Table Scheme



Two-Level Paging Example

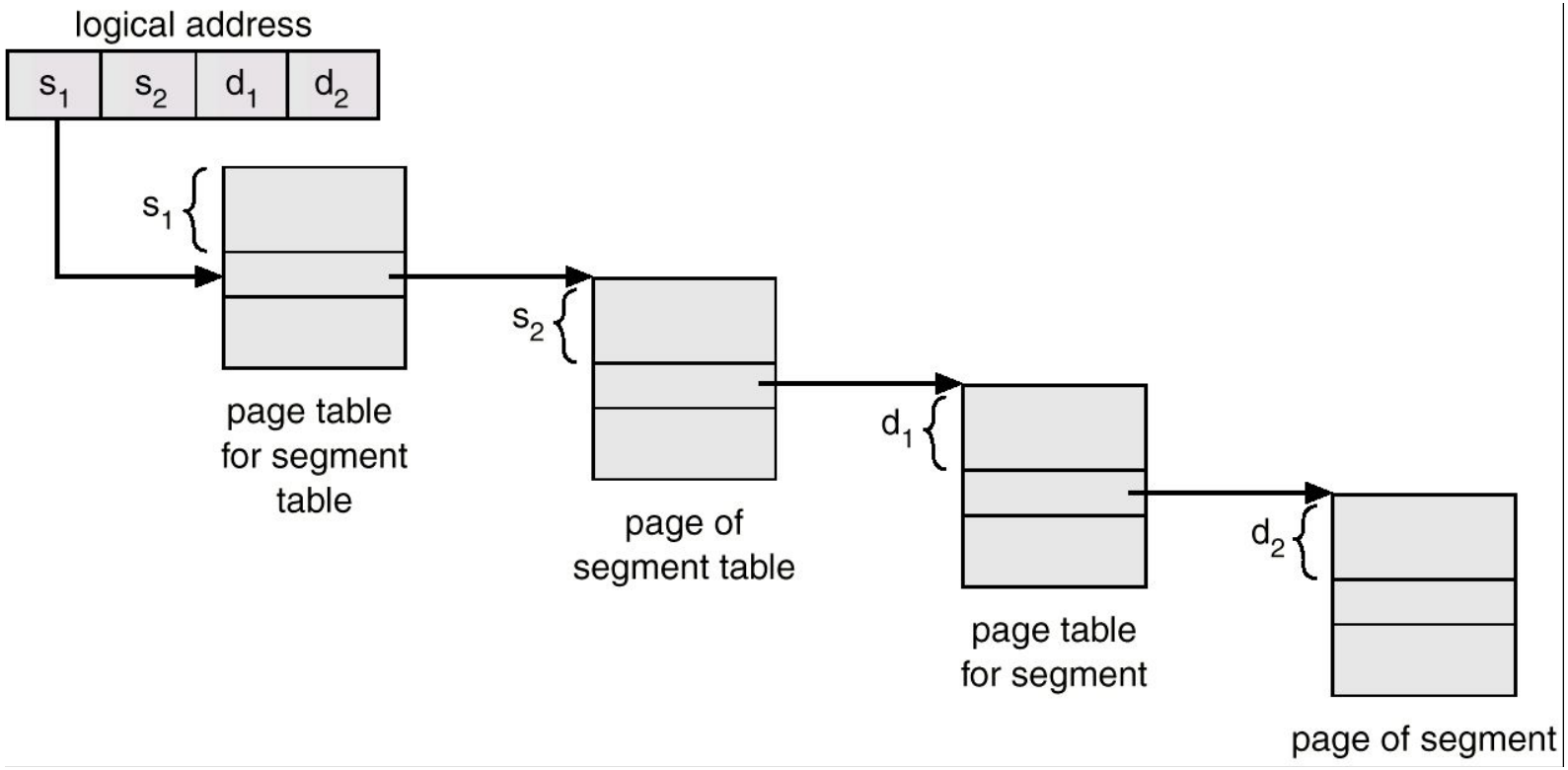
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:



where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

Address-Translation Scheme

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



Multilevel Paging and Performance

- Since each level is stored as a separate table in memory, covering a logical address to a physical one may take four memory accesses.
- Even though time needed for one memory access is quintupled, caching permits performance to remain reasonable.
- Cache hit rate of 98 percent yields:

  effective access time = 0.98 x 120 + 0.02 x 520

  = 128 nanoseconds.

  which is only a 28 percent slowdown in memory access time.

Bélády's anomaly

- In computer storage, Bélády's anomaly is the phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm.

- Generally, on increasing the number of frames to a process' virtual memory, its execution becomes faster as less number of page faults occur. Sometimes the reverse happens, i.e. more number of page faults occur when more frames are allocated to a process. This most unexpected result is termed as Belady's Anomaly.

Belady's Anomaly in FIFO –Example

- Assuming a system that has no pages loaded in the memory and uses the FIFO Page replacement algorithm. Consider the following reference string:
- 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **Case-1:** If the system has 3 frames, the given reference string on using FIFO page replacement algorithm yields a total of 9 page faults. The diagram below illustrates the

1	1	1	2	3	4	1	1	1	2	5	5
	2	2	3	4	1	2	2	2	5	3	3
		3	4	1	2	5	5	5	3	4	4
PF	PF	PF	PF	PF	PF	PF	X	X	PF	PF	X

Belady's Anomaly in FIFO –Example

- **Case-2:** If the system has 4 frames, the given reference string on using FIFO page replacement algorithm yields a total of 10 page faults. The diagram below illustrates the pattern of the page faults occurring in the example. 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	1	1	2	3	4	5	1	2
	2	2	2	2	2	3	4	5	1	2	3
		3	3	3	3	4	5	1	2	3	4
			4	4	4	5	1	2	3	4	5
PF	PF	PF	PF	X	X	PF	PF	PF	PF	PF	PF

- In this example on increasing the number of frames while using the FIFO page replacement algorithm, the number of **page faults increased** from 9 to 10.
- **Note –** It is not necessary that every string reference pattern cause Belady anomaly in FIFO but there are certain kind of string references