BCA
Fourth Semester
" Operating System"

Types of Memory:

Primary Memory (eg. RAM)
Holds data and programs used by a process that is executing
Only type of memory that a CPU deals with
Secondary Memory (eg. hard disk)
Non-volatile memory used to store data when a process is not executing.

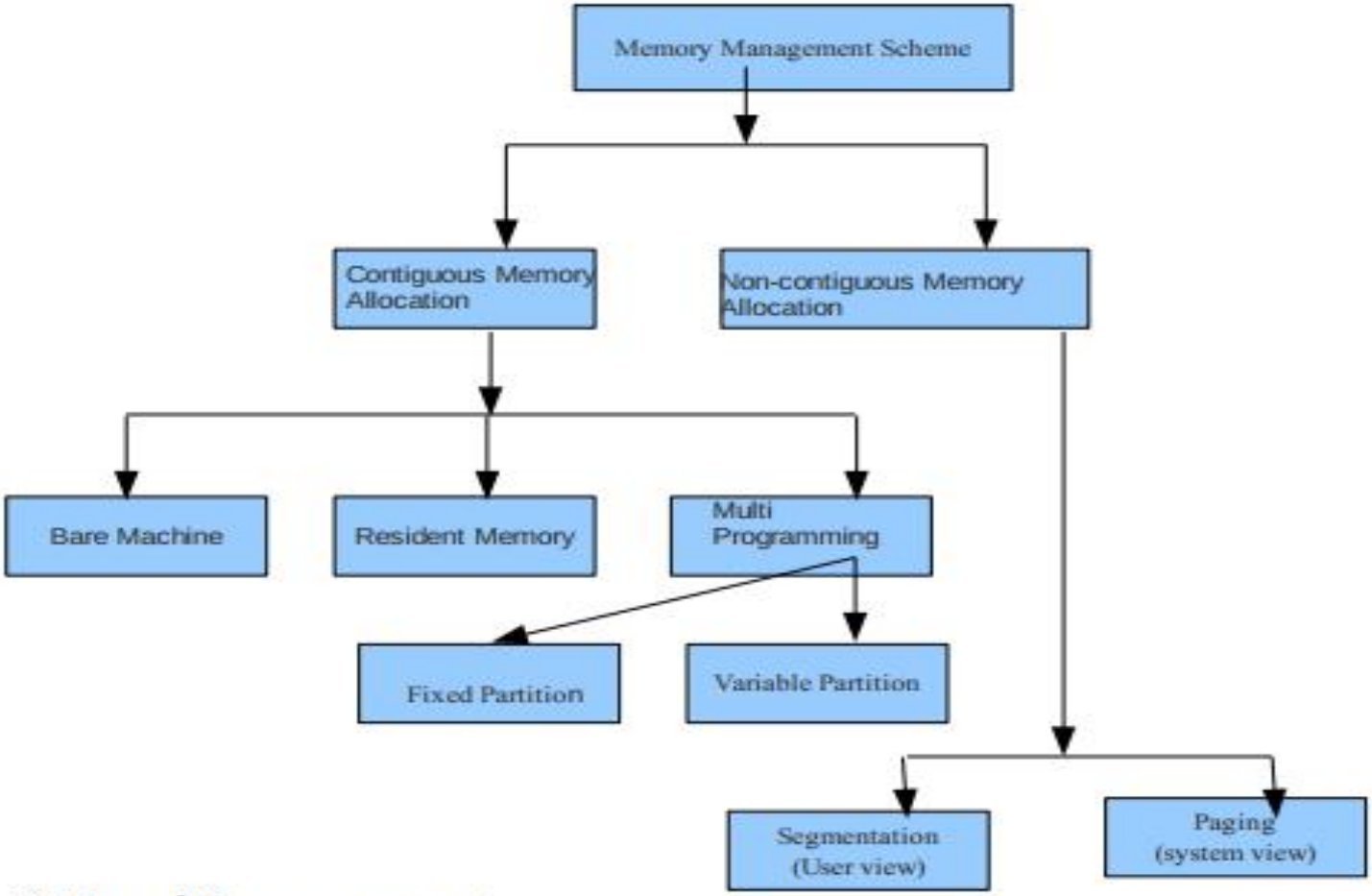


Fig:Types of Memory management

Memory Management

Reading: Chapter 8 of Textbook

The program and data of a process must be in main memory for the process to execute.

How to keep the track of processes currently being executed?
Which processes to load when memory space is available?
How to load the processes that are larger than main memory?
How do processes share the main memory?

OS component that is responsible for handling these issues is a *memory manager*.

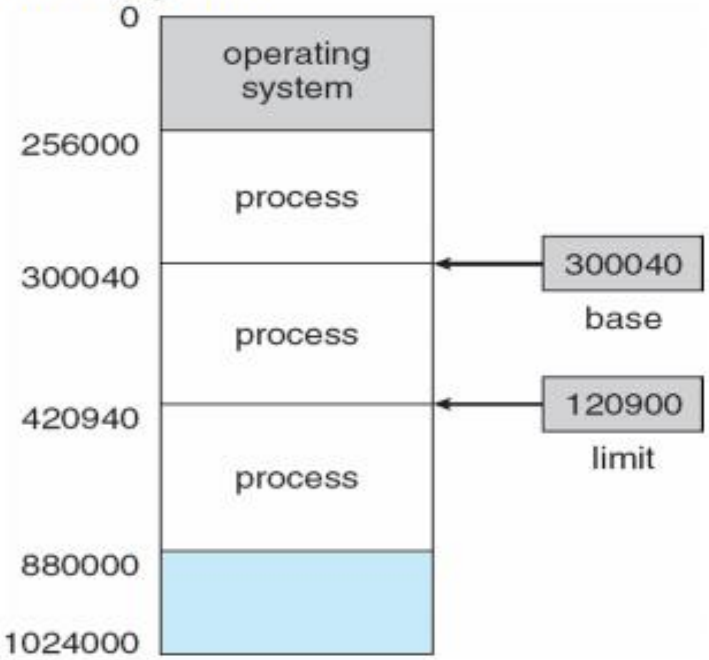
Memory Management Requirements

Relocation

- Programmer does not know where the program will be placed in memory when it is executed
- While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
- Memory addresses must be translated in the code to actual physical memory address.

Hardware used - Base register, limit register.

Base register: Holds smallest legal physical memory address.
Limit register: contain the size of range.
Example:
logical address = 300
if Base = 1500
physical address = 1500 + 300 = 1800.



Memory Management Requirements

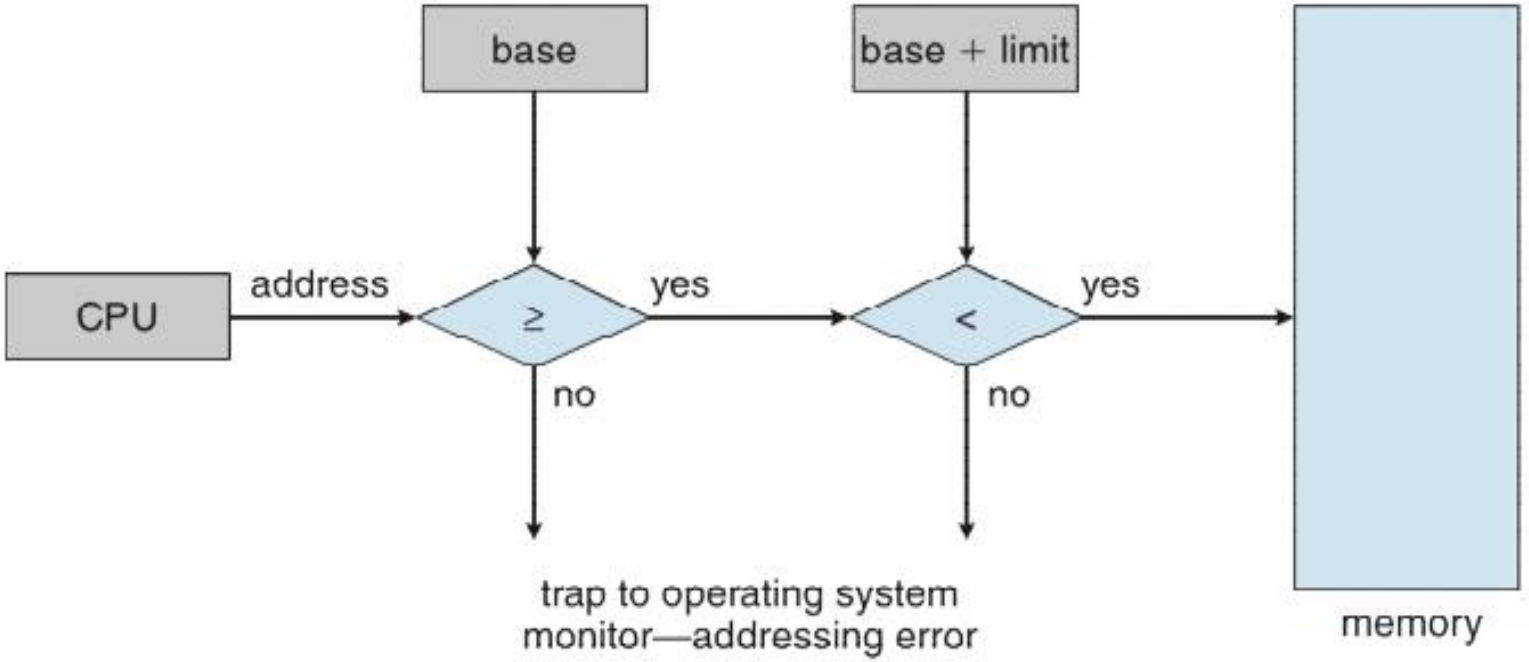
Protection

- Prevent processes from interfering with the OS or other processes.
- Processes should not be able to reference memory locations in another process without permission.
- Impossible to check absolute addresses in programs since the program could be relocated.
- Often integrated with relocation.

Sharing

- Allow several processes to access the same portion of memory (allow to share data).
- Better to allow each process (person) access to the same copy of the program rather than have their own separate copy.

Address Protection with Base and Limit Registers



CPU compare every address generated in user mode with the base registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error

Logical vs. Physical Memory Address Space

Logical Address: The address generated by CPU. Also known as virtual address.

Physical Address: Actual address loaded into the memory address register.

The set of all logical addresses generated by a program is a logical address space; the set of all physical addresses corresponding to these logical addresses is a physical address space.

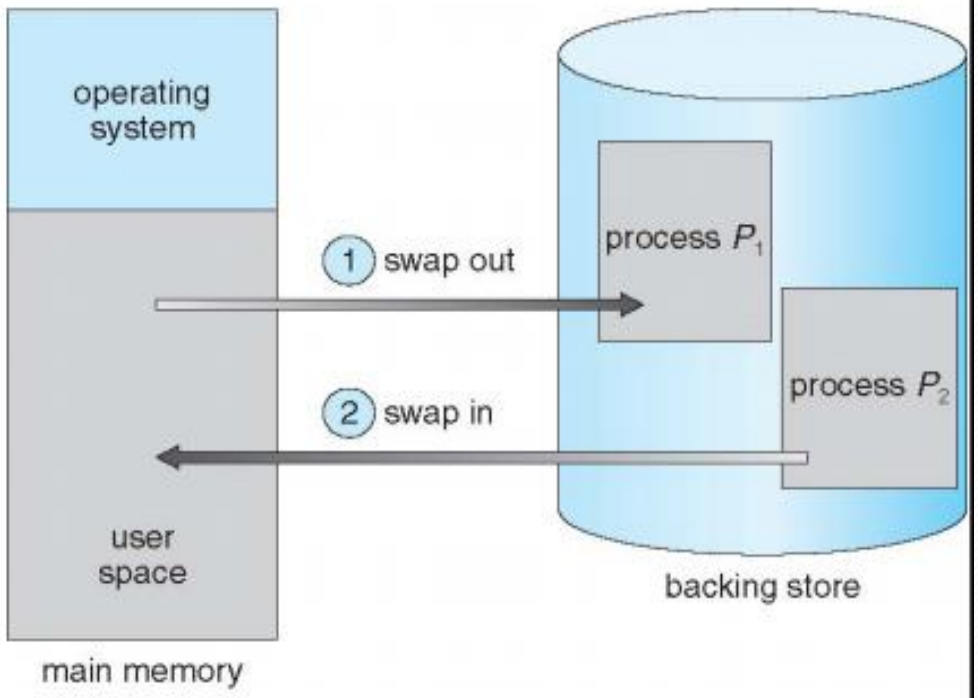
Mapping from logical address to physical address is *relocation*.
Two registers: *Base (relocation)* and *Limit* are used in mapping.

Hardware device that maps virtual to physical address at run time is *Memory Management Unit (MMU)*

Swapping

In multiprogramming environment, some times there is no enough memory to hold all processes that is being executed.
How to handle this problem? – *swapping*.
A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
both the memory and CPU.

In multiprogramming processes will be swapped in and out many times before its completion.
In some swapping systems, one processes occupies the main memory at once; when process can no longer to continue it relinquishes



Memory Allocation

How to allocate the space for a process as much as they need?
Two Strategies:

Fixed size partitions and variable size partitions

Fixed Size Partitions:

Multiple Partitions are created to allow multiple user processes to resident in memory simultaneously
Partition table stores the partition information for a process.
When job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Science the partition are fixed, any space in partition not used by a process is lost.
Degree of multiprogramming is bound by number of partitions

Variable Partitions

When processes arrive, they are given as much storage as they need.

Memory Allocation

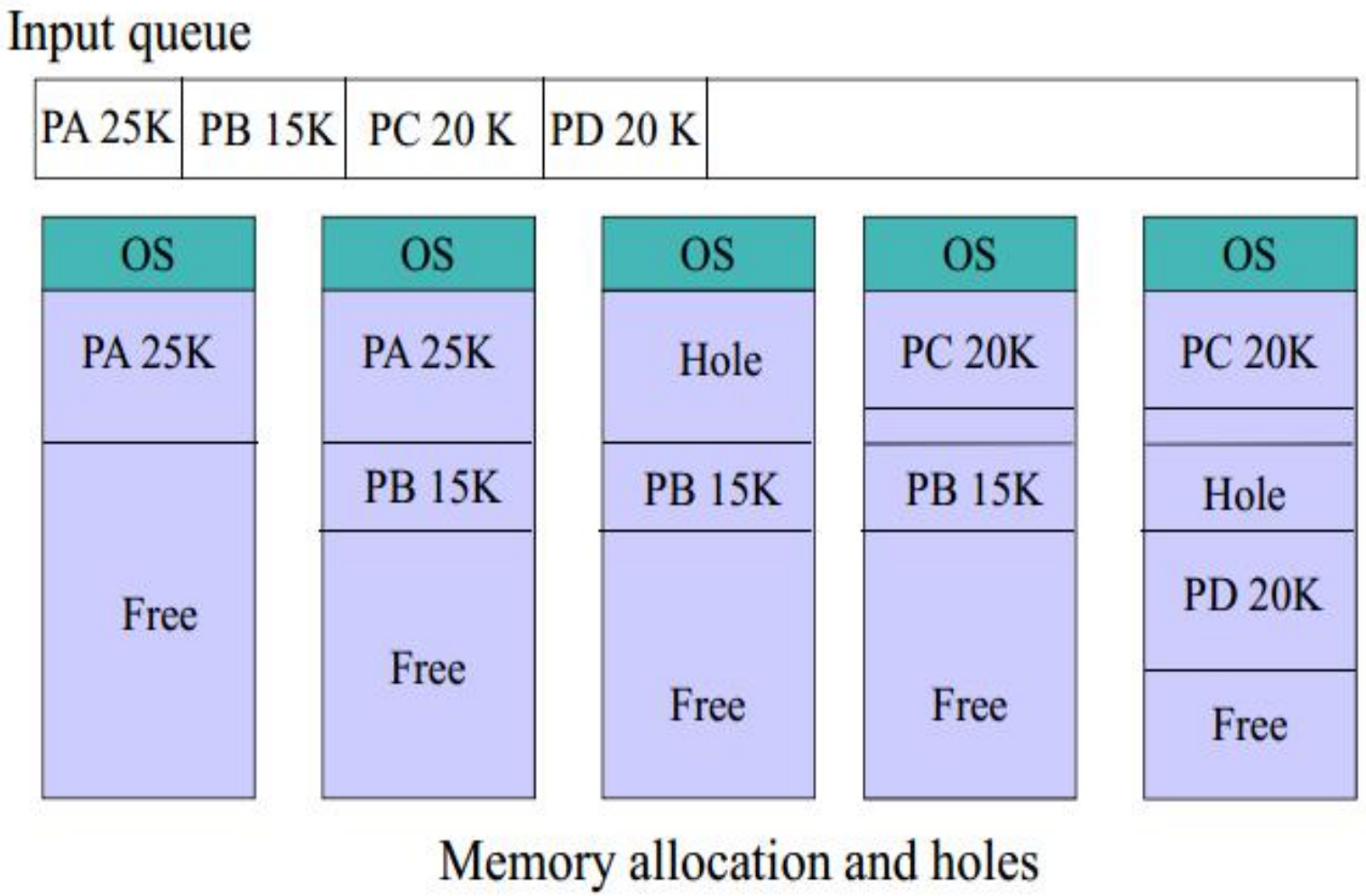
When processes finish, they leave holes in main memory; OS fills these holes with another processes from the input queue of processes.

How to satisfy a request of size *n* from a list of the free holes (Partition Selection Problem)?

Situation: Multiple memory holes are large enough to contain a process, OS must select which hole the process will be loaded into.

There are many solutions to this problem – *Partition Selection Algorithms or Placement Algorithms*

Memory Allocation



Memory allocation and holes

Partition Selection Algorithms

First Fit

Allocate the first hole that is big enough. It stop the searching as soon as it find a free hole that is large enough. The hole is then broken up into two pieces, one for the process and one for unused memory.

Advantage: It is a fast algorithm because it search as little as possible.

Problem: not good in terms of storage utilization.

Best Fit

Allocate the smallest hole that is big enough. It search the entire list, and takes the smallest hole that is big enough. Rather than breaking up a big hole that might be needed later, it finds a hole that is close to the actual size.

Advantage: More storage utilization than first fit but not always.

Problem: Slower than first fit because it requires to search whole list at every time. Creates tiny hole that may not be used.

Worst Fit

Allocate the largest hole. It search the entire list, and takes the largest hole. Rather than creating tiny hole it produces the largest leftover hole, which may be more useful.

Advantage: Some time it has more storage utilization than first fit and best fit.

Problem: not good for both performance and utilization.

Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

Answer:

a. First-fit:
b. 212K is put in 500K partition
c. 417K is put in 600K partition
d. 112K is put in 288K partition (new partition 288K = 500K - 212K)
e. 426K must wait
f. Best-fit:
g. 212K is put in 300K partition
h. 417K is put in 500K partition
i. 112K is put in 200K partition
j. 426K is put in 600K partition
k. Worst-fit:
l. 212K is put in 600K partition
m. 417K is put in 500K partition
n. 112K is put in 388K partition
o. 426K must wait

In this example, Best-fit turns out to be the best.

Memory Fragmentation

When holes given to other process they may again partitioned, the remaining holes gets smaller eventually becoming too small to hold new processes – waste of memory occur – memory fragmentation.

External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous. Fist fit and Best Fit suffers from external fragmentation. External fragmentation can be reduced by memory **compaction**

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, 0.5 *N* blocks lost to fragmentation 1/3 may be unusable! -> **50-percent rule**

Memory Compaction

By moving processes in memory, the memory holes can be collected into a single section of unallocated space – memory compaction.

Problems:

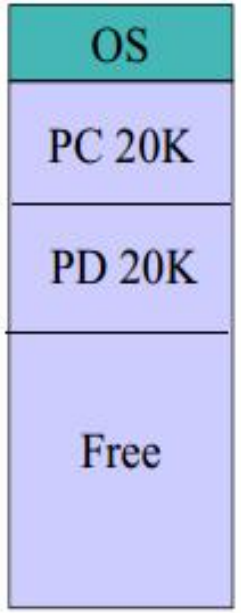
Not possible in absolute translation. Compaction is possible only if relocation is dynamic, and is done at execution time

If it possible, it is highly expensive – it requires a lots of CPU time;

Eg: On a 256-MB machine that can copy 4 bytes in 40 nsec, it takes about 2.7 sec to compact all memory.

It stops every thing when compaction – *not good solution*

Alternate solutions: Allows the noncontiguous allocations e.g Paging, segmentation *(wait!)*



Memory compaction

Paging

The logical address space (process) is divided up into fixed sized blocks called *pages* and the corresponding same size block in main memory is called *frames*
When a process is to be executed, its pages are loaded into any available memory frames from the backing store.

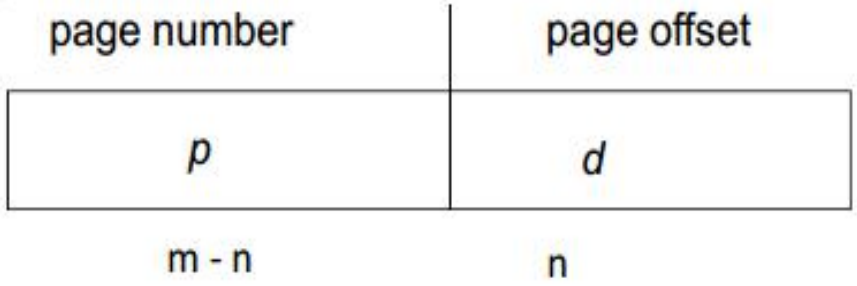
The size of the pages is determined by the hardware, normally from 512 bytes to 16MB (in power of 2).
Paging permits the physical address space of process to be noncontiguous.
Traditionally, support for paging has been handled by hardware, but the recent design have implemented by closely integrating the hardware and OS.
Keep track of all free frames
To run a program of size N pages, need to find N free frames and load program
Set up a *page table* to translate logical to physical addresses
Still have Internal fragmentation

Address Translation

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) - combined with base address to define the physical memory address that is sent to the memory unit.

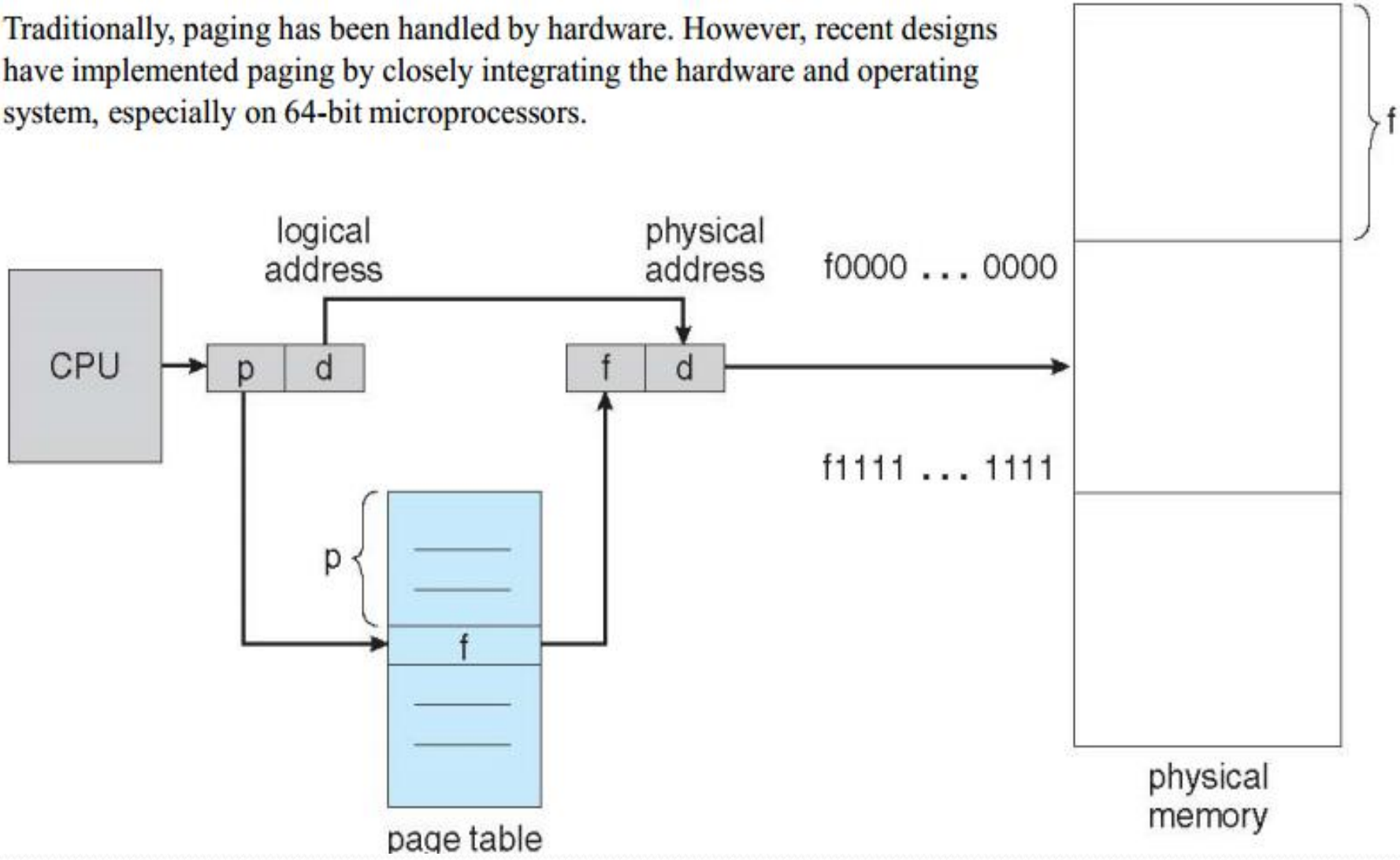


If the size of the logical address space is 2^m and page size is 2^n, then the high-order m - n bits of logical address designate page number, and n lower order bits designate the page offset.

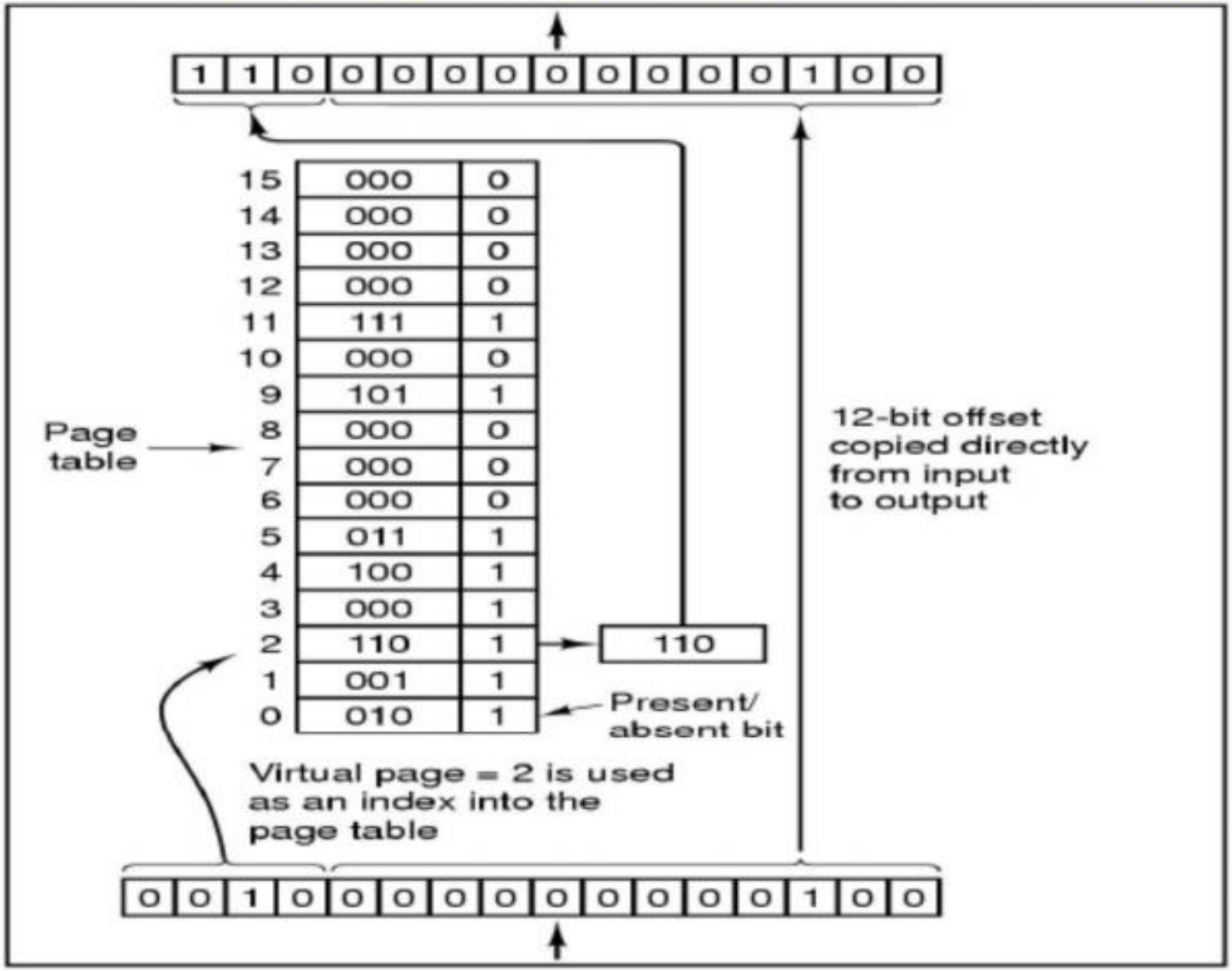
Present/absent bit keeps the track of which pages are physically present in memory.

Address Translation

Traditionally, paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.



Address Translation Example



Paging Fragmentation

No external fragmentation: any free frame can be allocated to a process that needs it. However, it may have some internal fragmentation

Calculating internal fragmentation

Page size = 2,048 bytes, Process size = 72,766 bytes,
No of pages = 35 pages + 1,086 bytes = 36 frames
Internal fragmentation = 2,048 - 1,086 = 962 bytes

In the worst case, a process would need *n* pages plus 1 byte. It would be allocated *n* + 1 frames, resulting in internal fragmentation of almost an entire frame.

Page Tables

A page table is allocated for each process, stores the number of frames allocated for each page.

The purpose of the page table is to map virtual pages into pages frames. This function of page table can be represented in mathematical notation as:

$$page\_frame = page\_table(page\_number)$$

The virtual page number is used as an index into the page table to find the corresponding page frame.

Page Tables Issues

Page table is kept in main memory

Efficiency of mapping.

If a particular instruction is being mapped, the table lookup time should be very small than its total mapping time to avoid becoming CPU idle.
What would be the performance, if such a large table have to load at every mapping.

Size of page table

Most modern computers support a large virtual-address space (2^32 to 2^64). If the page size is 4KB, a 32-bit address space has 1 million pages. With 1 million pages, the page table must have 1 million entries.

think about 64-bit address space!

How to handle these issues?

Hardware Support: TLB

How to speed up address translation?

Locality: Most processes use large number of reference to small number of pages.

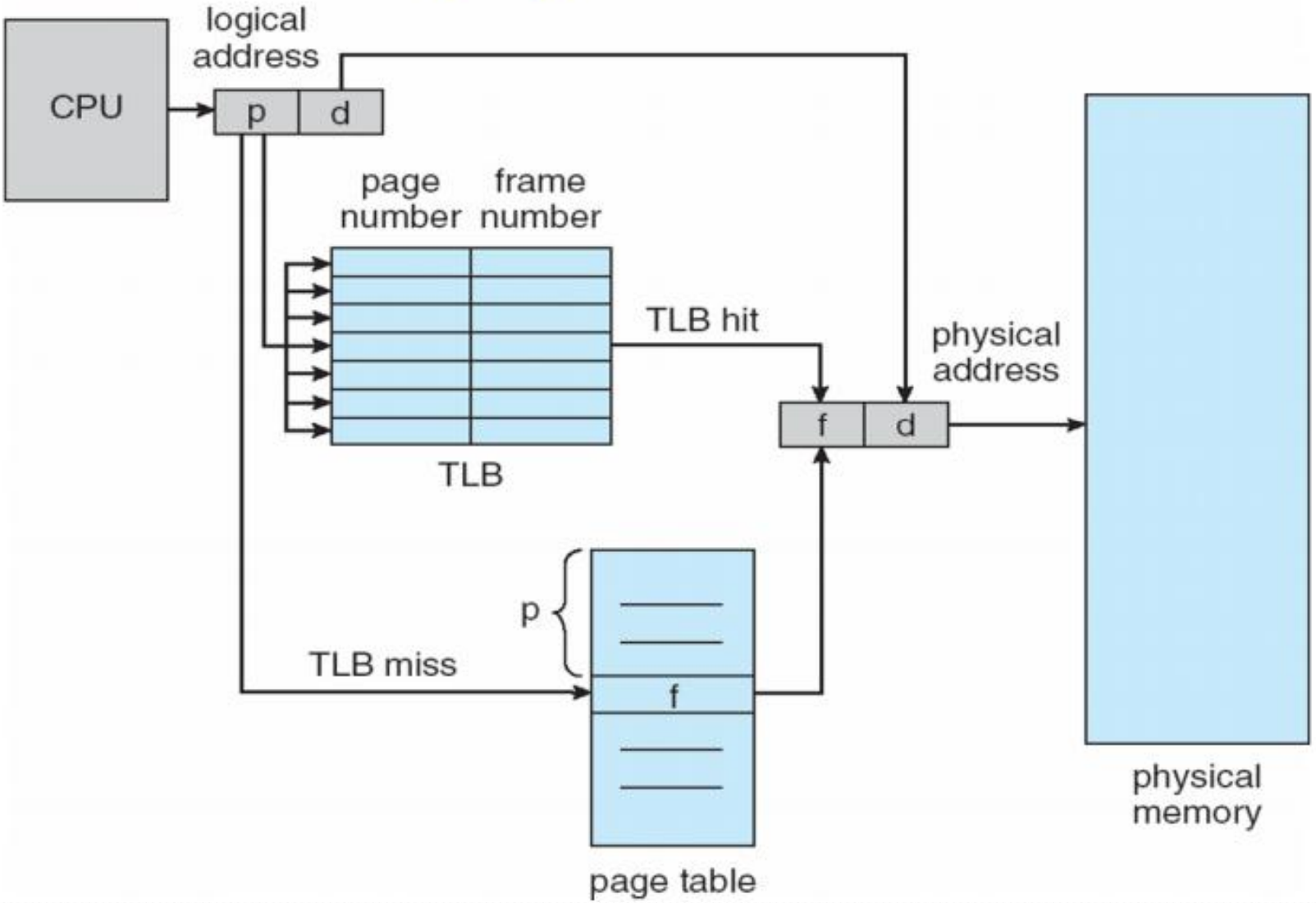
The small, fast, lookup hardware cache, called translation look-aside buffers (TLBs) or associative memory is used to overcome this problem, by mapping logical address to physical address without page tables.

The TLB is associative, high-speed, memory; each entry consists information about virtual page-number and physical page frame number.
Typical sizes: only 64 to 1024 entries.

When logical address is generated by CPU, its page number is presented to the TLB; if page number is found (TLB hit), its frame number is immediately available, the whole task would be very fast because it compare all TLB entries simultaneously.

If the page number is not in TLB (TLB miss), a memory reference to the page table must be made. It then replace one entry of TLB with the page table entry just referenced. Thus in next time, for that page, TLB hit will found.

Paging with TLB



Multilevel Page Tables

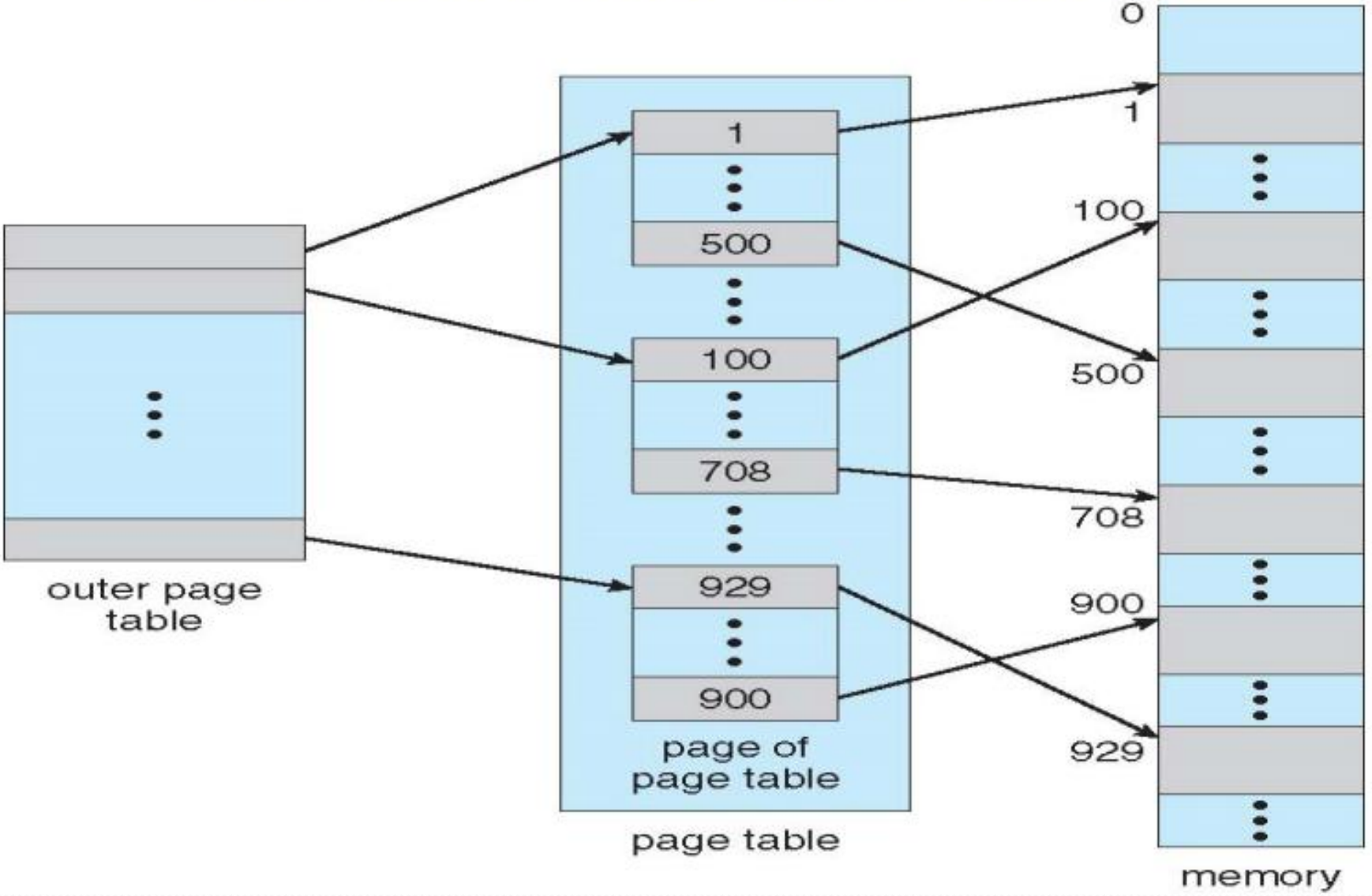
To get around the problem of having to store huge page tables in memory all the time, many computers use the multilevel page table in which the page table itself is also paged.

Break up the logical address space into multiple page tables

Pentium II -2 level, 32-bit Motorola -4 level, 32 -bit SPARC-3 level etc.
For 64-bit architectures, multilevel page table are generally considered inappropriate. For example, the 64-bit UltraSPARC would require seven level of paging - increase accessing complexity.

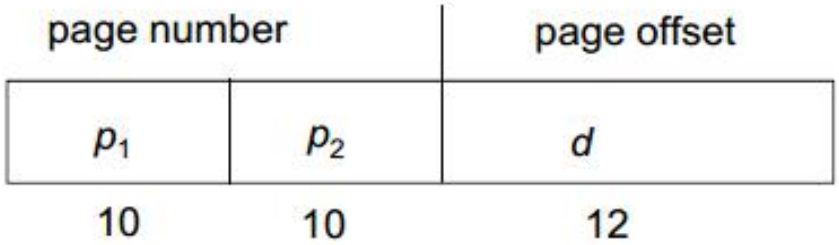
A simple technique is a two-level page table

Two-Level Page-Table Scheme

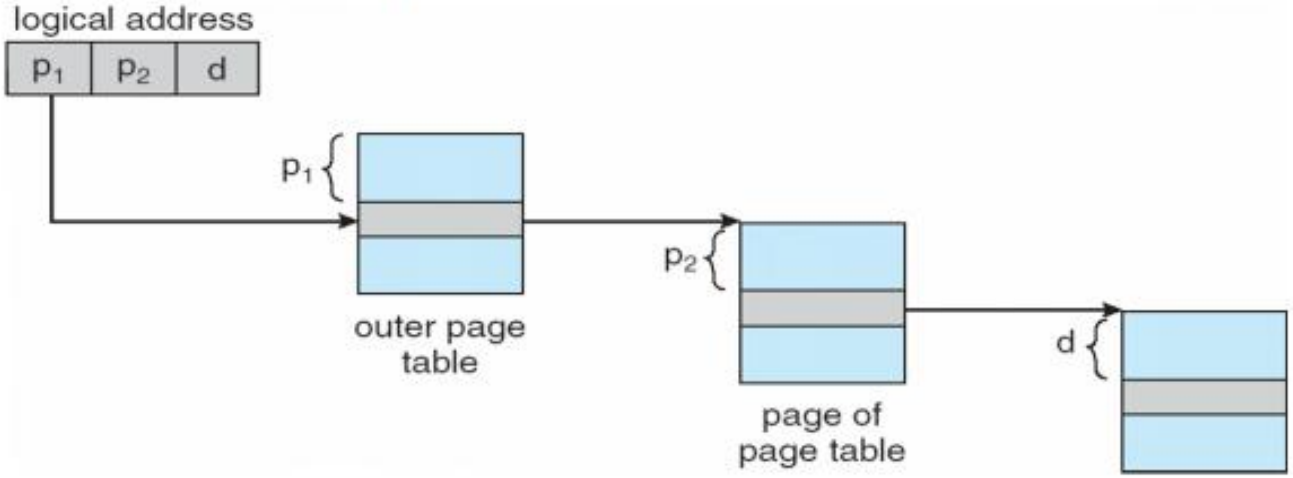


Two-Level Paging Example

A 32-bit virtual address space with a page size of 4 KB, the virtual address space is partitioned into a 10-bit P1 field, a 10-bit P2 field, and a 12-bit offset field.



The top level have 1024 entries, corresponding to P1. At mapping, it first extracts the P1 and uses this value as an index into the top level page table. Each of these entries have again 1024 entries, the resulting address of top-level yields the address or page frame number of second-level page table.



Hashed Page Tables

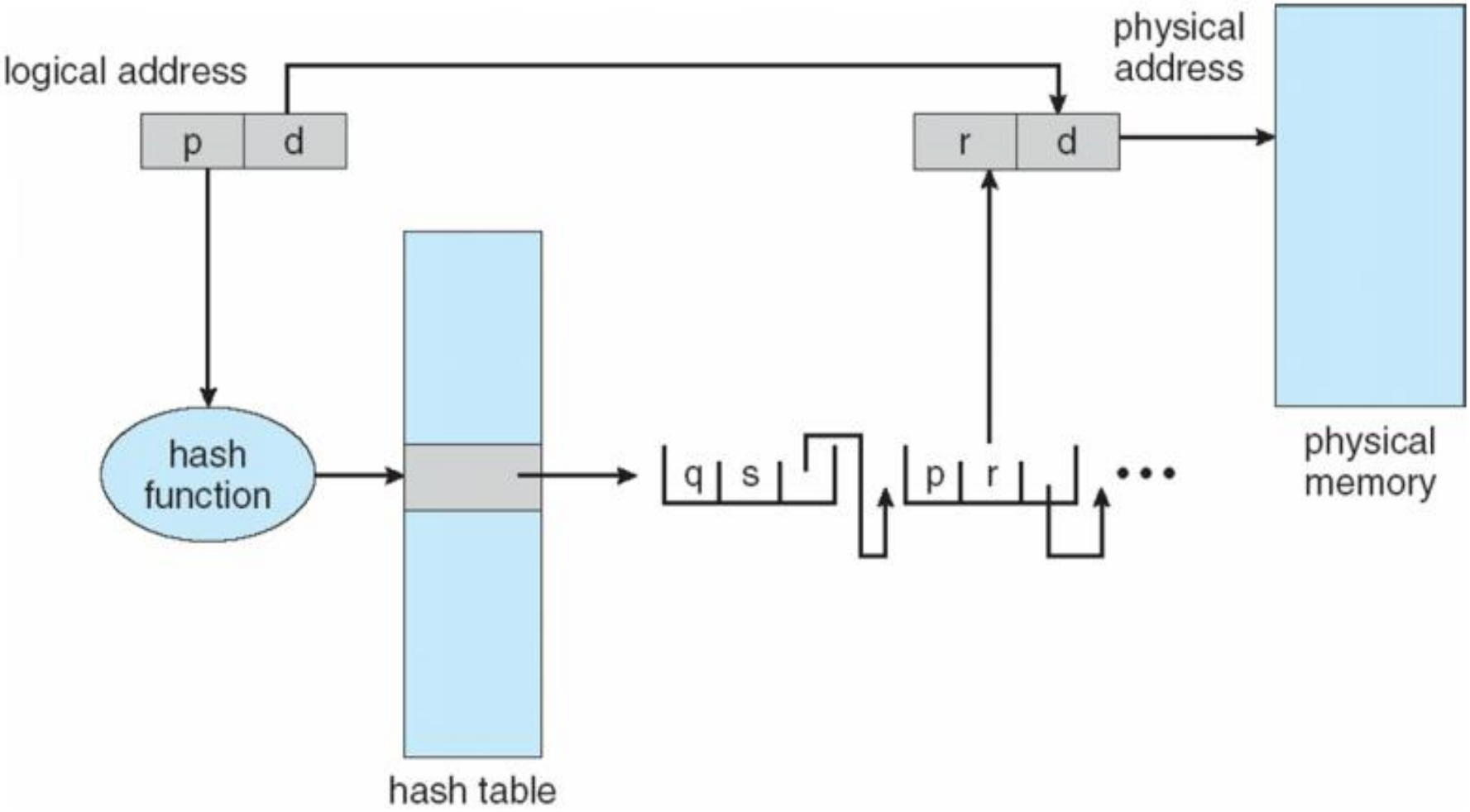
A common approach for handling address space larger than 32-bit.

The hash value is 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: *virtual-page-number, value of mapped page frame, and a pointer to the next element.*

The virtual address is hashed into the hash table, if there is match the corresponding page frame is used, if not, subsequent entries in the linked list are searched.

Hashed Page Tables



Inverted Page Tables

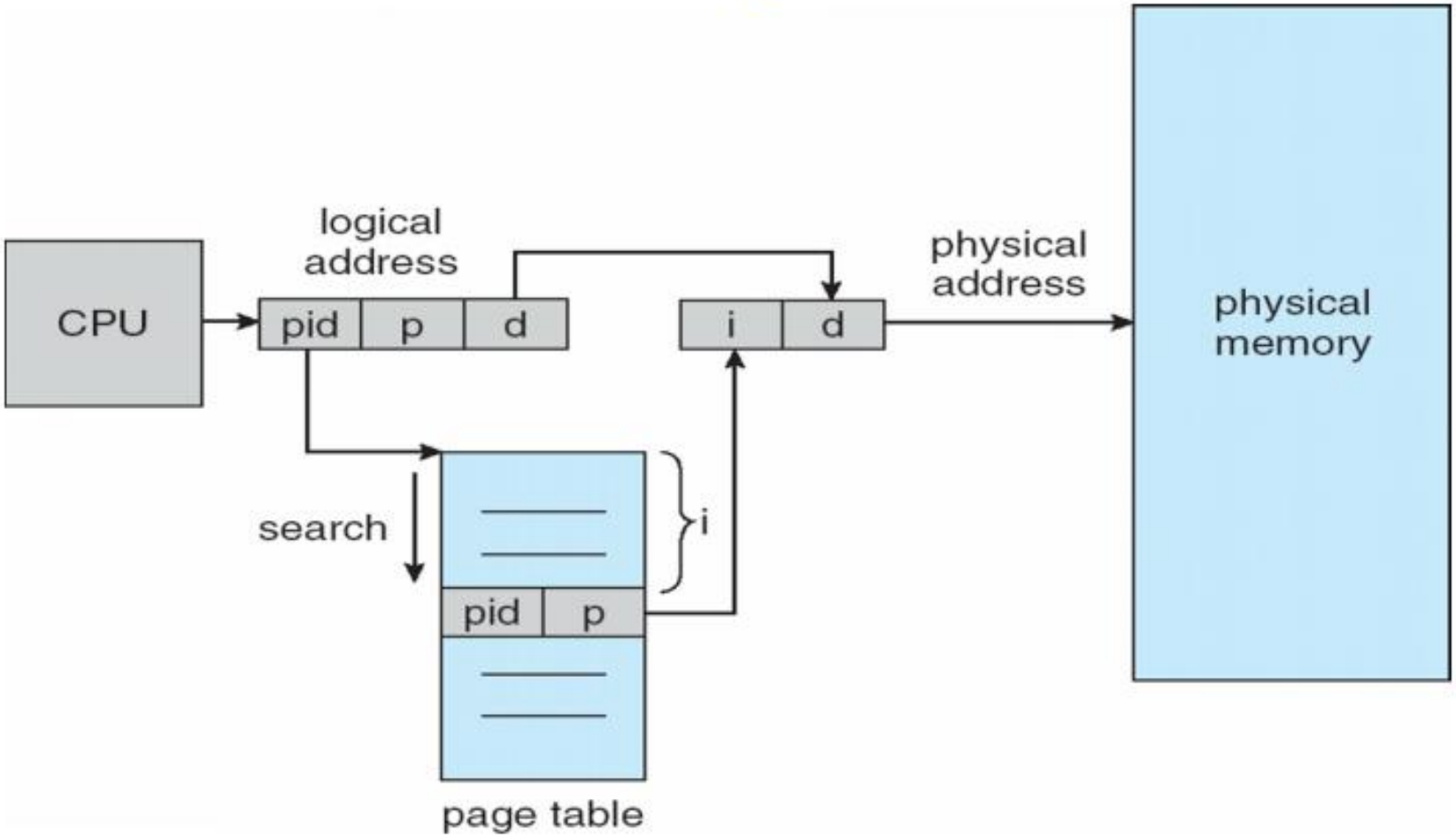
A common approach for handling address space larger than 32-bit.
One entry per page frame, rather than one entry per page in earlier tables. (Ex: 256MB RAM with 4KB page requires only 65,536 entries in table)

Entry consists of the virtual address of the page stored in that physical memory location, with information about the process that owns that page.

Virtual address consists three fields: *[process-id, page-number, offset]*. The inverted page table entry is determined by *[process-id, page-number]*. The page table is search for the match, say at entry *i* the match is found, then the physical address *[i, offset]* is generated.

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

Inverted Page Tables

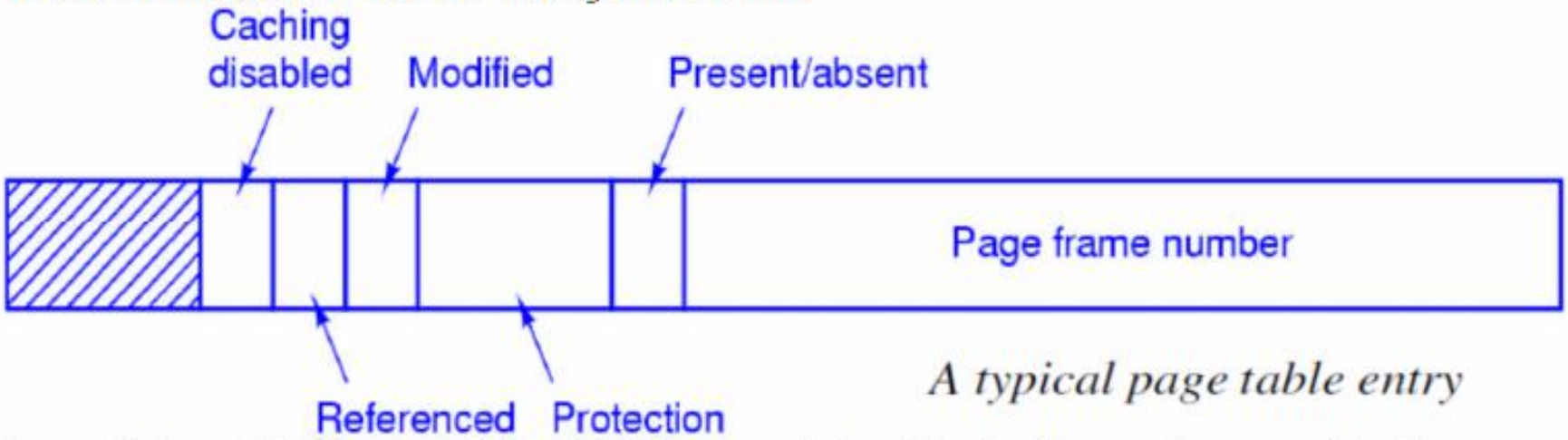


Advantage: Decreases the memory size to store the page table.

Problem: It must search entire table in every memory reference, not just on page faults.

Page Table Structure

The exact layout of page table entry is highly machine dependent, but more common structure for 32-bit system is as:



A typical page table entry

Present/absent bit: If present/absent bit is present, the virtual addresses is mapped to the corresponding physical address. If present/absent is absent the trap is occur called page fault.
Protection bit: Tells what kinds of access are permitted read, write or read only.
Modified bit (dirty bit): Identifies the changed status of the page since last access; if it is modified then it must be rewritten back to the disk.
Referenced bit: set whenever a page is referenced; used in page replacement.
Caching disabled: used for that system where the mapping into device register rather than memory.
Frame number: The goal is to locate this value.

Advantages/Disadvantages of Paging

Advantages:

Fast to allocate and free:

Allocate: keep free list of free pages, grab first page in the list.

Free: Add pages to free list.

Easy to swap-out memory to disk.

Frame size matches disk page size.

Swap-out only necessary pages.

Easy to swap-in back from disk.

Disadvantages:

Additional memory reference.

Page table are kept in memory.

Internal fragmentation: process size does not match allocation size.

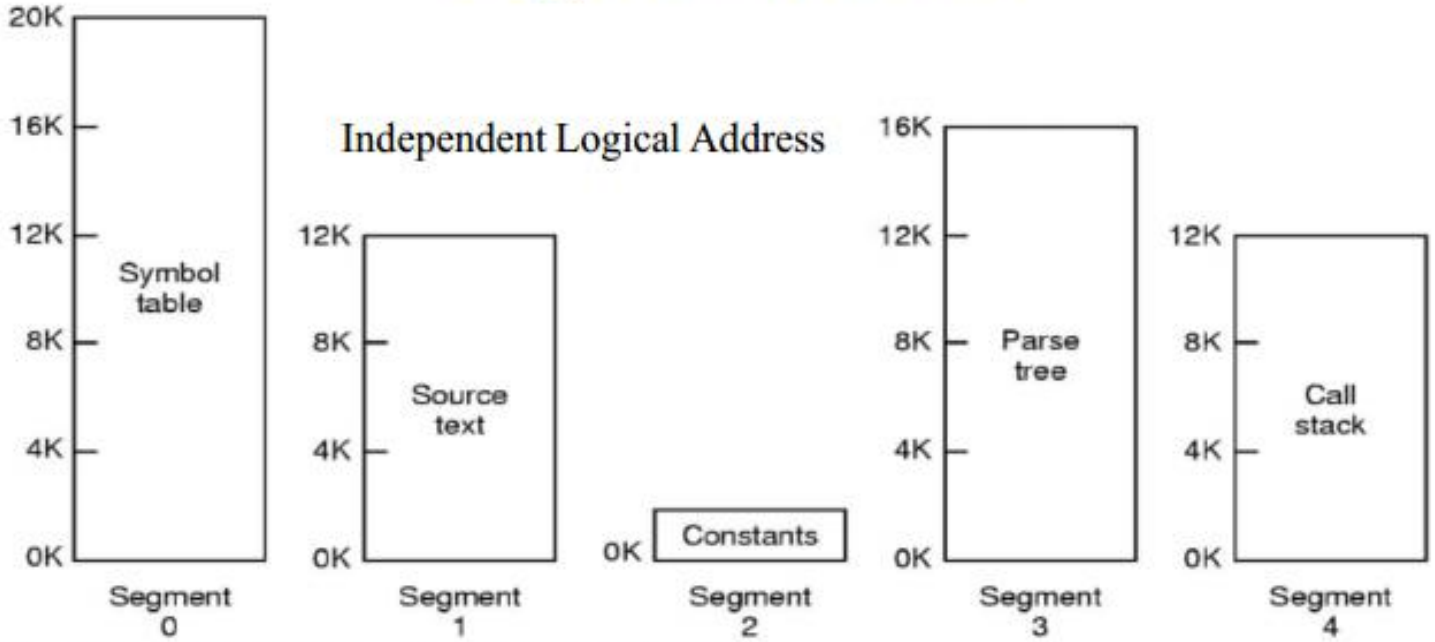
Segmentation

What happens if program increase their size in their execution?
How to manage expanding and contracting tables?
How to protect only data from the program?
How to share data to other program or functions?

The general solution of these issues is to provide the machine with many completely independent address spaces, called segments.

Memory management that support variable partitioning and mechanisms with freedom of contiguous memory requirement restriction.
The independent block of the program is a segment such as: main program, procedures, functions, methods, objects, local variables, global variables, common blocks, stacks, symbol table, arrays.
The responsibility for dividing the program into segments lies with user (or compiler).

Segmentation



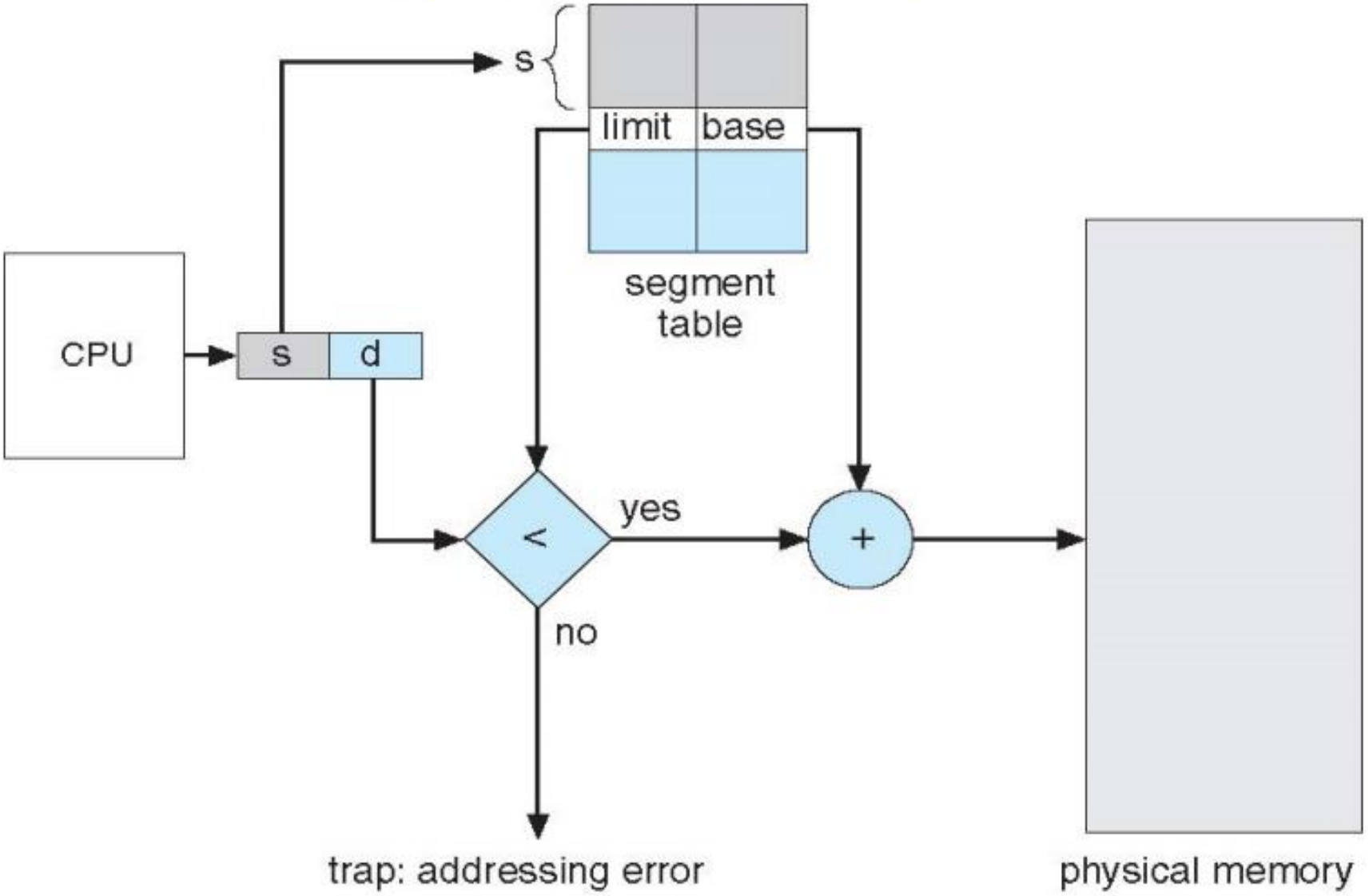
Different segments have its own name and size.

The different segment can grow or shrink independently, with out effecting the others; so the size of segment changed during execution.

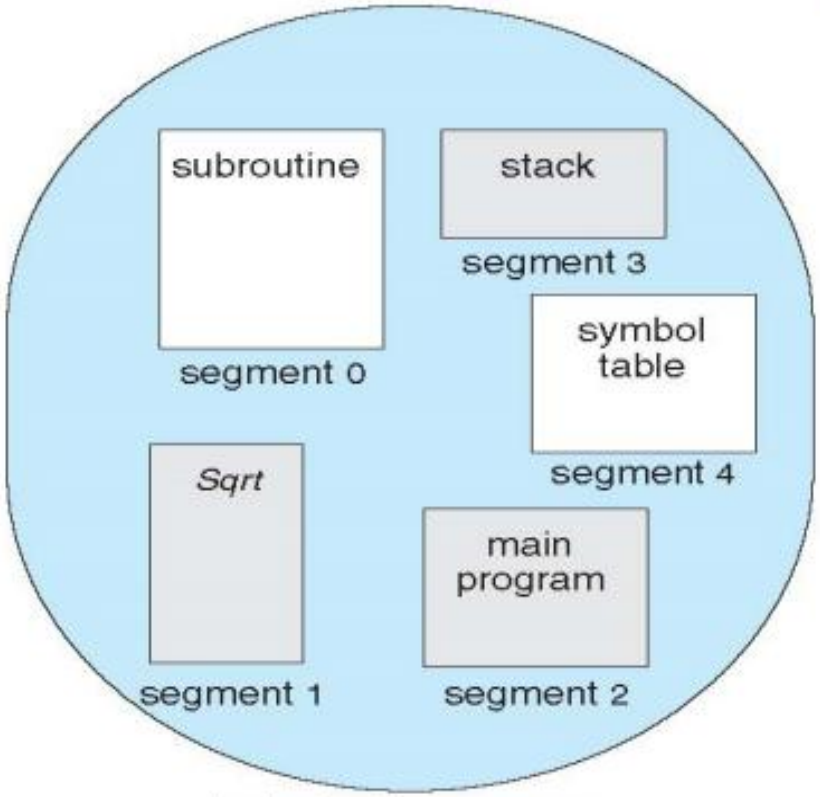
For the simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by segment name. Thus the logical address consist: segment number and offset.

The segment table (like page table but each entry consist limit and base register value) is used to map the logical address to physical address.

Segmentation Hardware

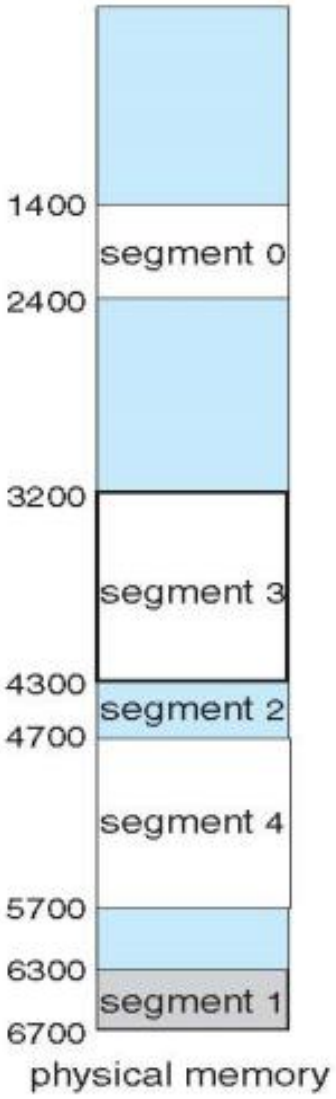


Example of Segmentation



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

segment table

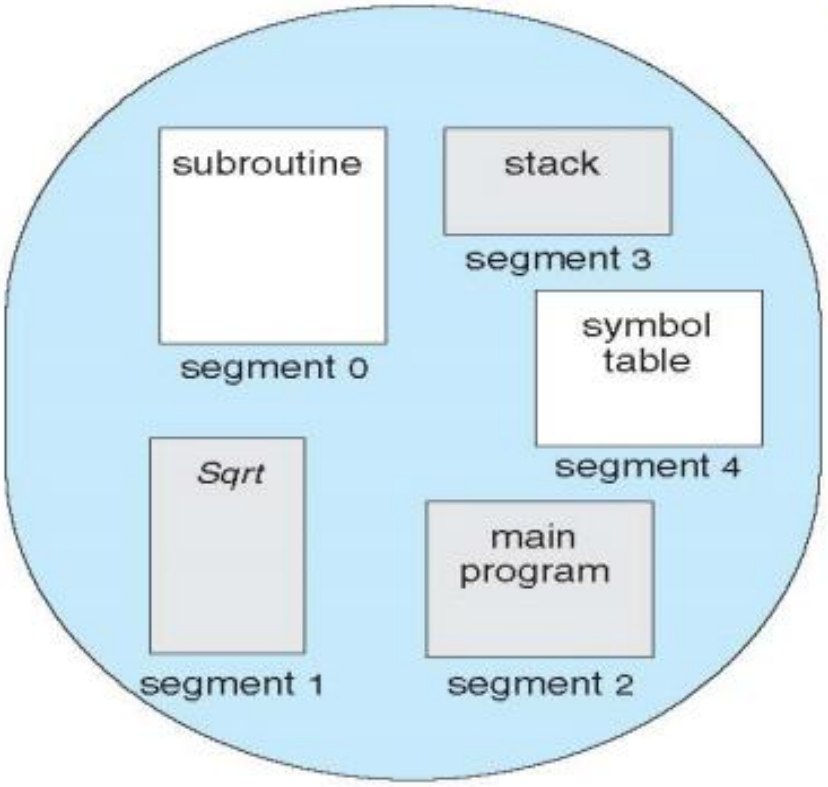


The segment number used as index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If not ,trap occur, if it is legal it is added to the segment base to produce the address in the physical memory.

Paging vs. Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

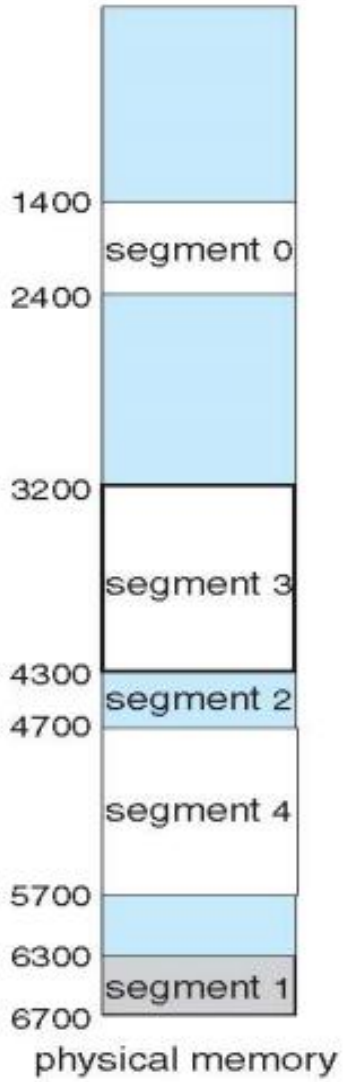
Example of Segmentation



logical address space

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

segment table



physical memory

The segment number used as index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If not ,trap occur, if it is legal it is added to the segment base to produce the address in the physical memory.

Segmentation with Paging

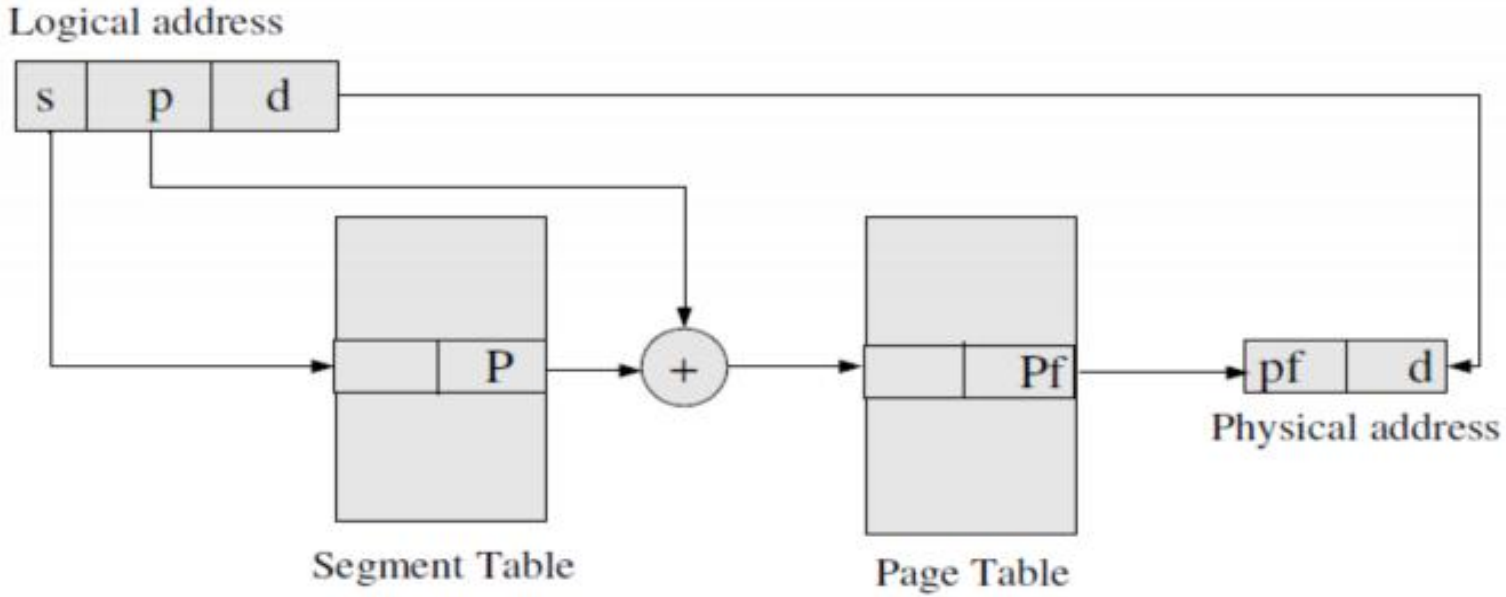
What happen when segment are larger than main memory?

Segmentation can be combined with paging to provide the efficiency of paging with the protection and sharing capabilities of segmentation.

As with simple segmentation, the logical address specifies the segment number and the offset within the segment.

When paging is added, the segment offset is further divided into a page number and page offset.

The segment table entry contains the address of the segment's page table.



Segmentation with Paging Examples

The Intel Pentium:

The Intel Pentium 80386 and later architecture uses segmentation with paging memory management.

The maximum number of segments per process is 16K, and each segment can be large as 4 GB. The page size is 4K. It use two-level paging scheme.

Multics:

It has 256K independent segments, and each up to 64K. The page size is 1K or small.

Virtual Memory Management

Reading: Chapter 9 of Textbook

Virtual memory is a concept that is associated with ability to address a memory space much larger than that the available physical memory.

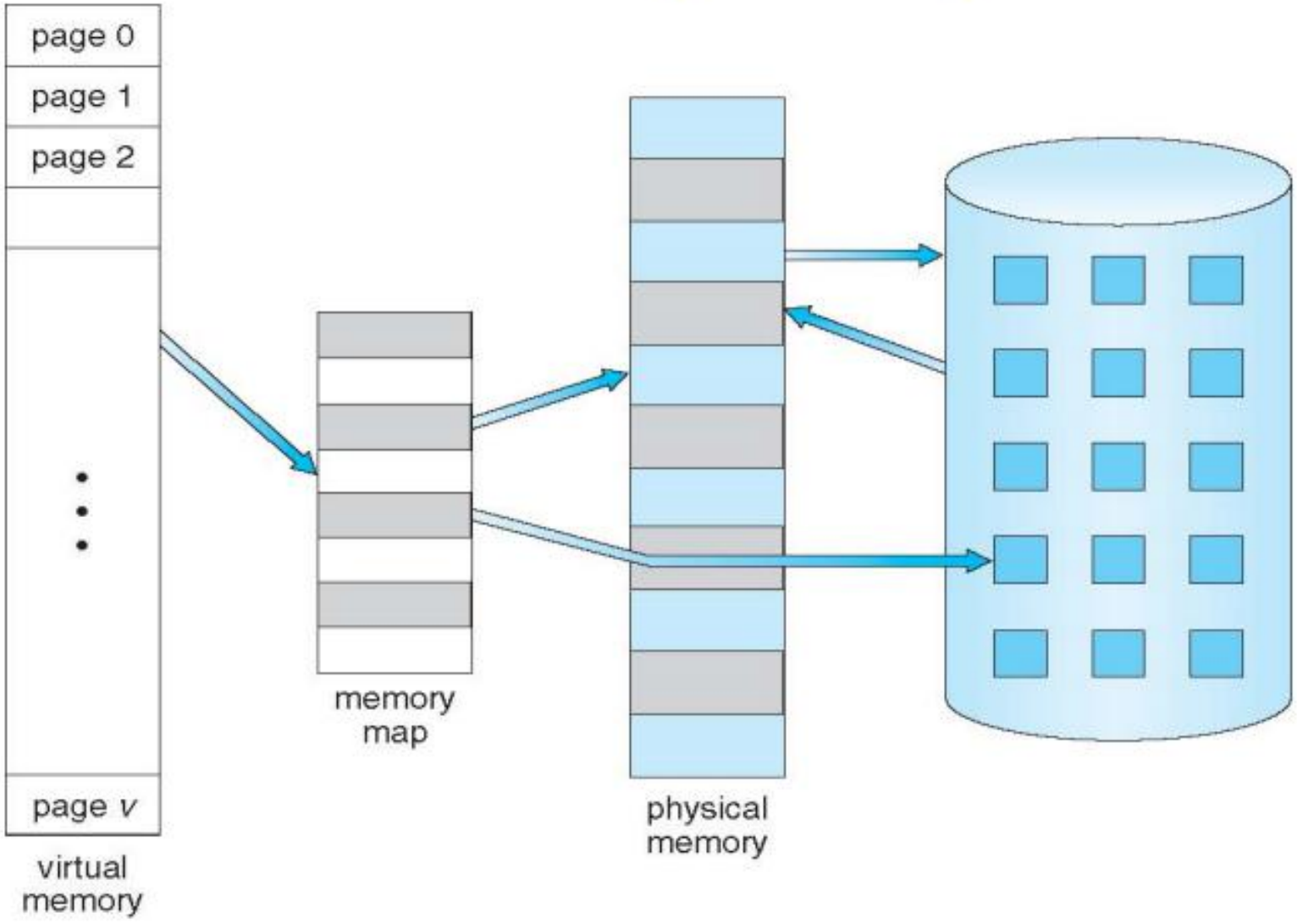
Fact: entire program would not be required at once for execution.

The basic idea behind the virtual memory is that the combined size of the of the program, data, and stack may exceed the amount of physical memory available for it. The OS keeps those part of the program currently in use in main memory, and the rest on the disk.

Virtual storage is not a new concept, this concept was devised by Fotheringham, 1961 and used in Atlas computer system.

But the common use in OS is the recent concept, all microprocessor now support virtual memory.

Virtual Memory Management



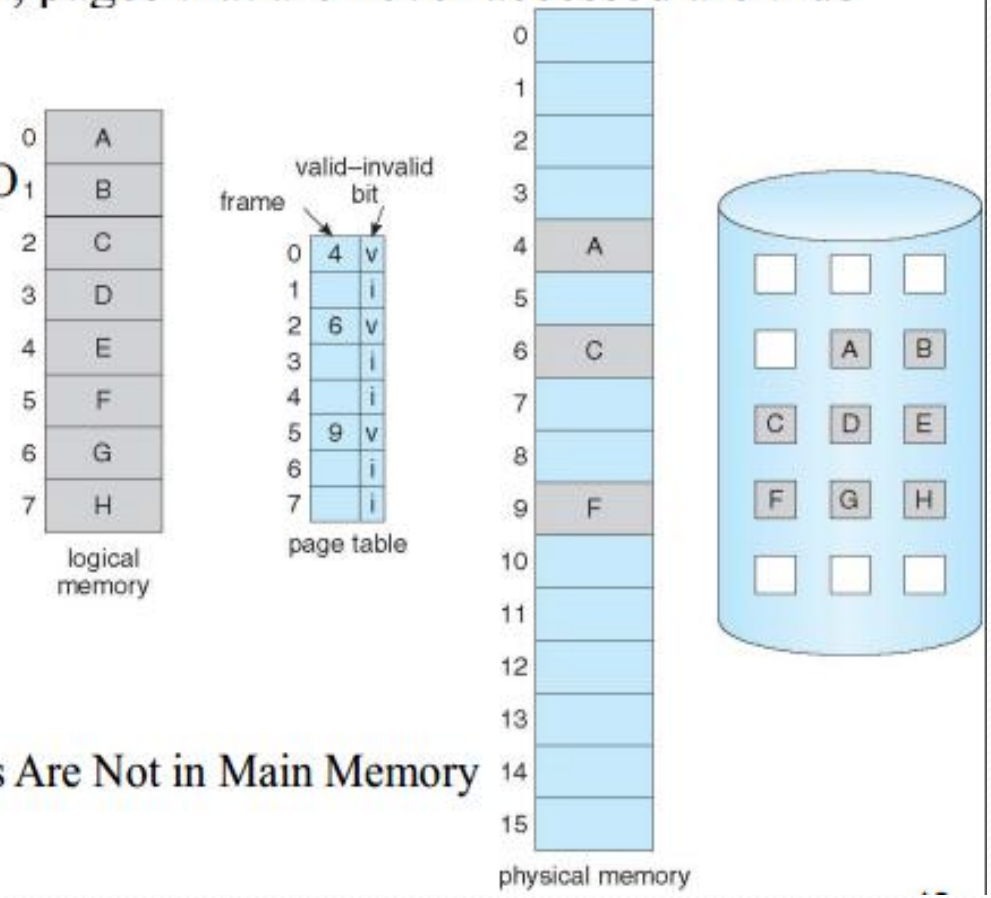
Demand Paging

Pages requested by CPU – Demand pages

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.

Demand paging strategy:

- Less I/O needed, no unnecessary I/O
- Less memory needed
- Faster response
- More users



Page Table When Some Pages Are Not in Main Memory

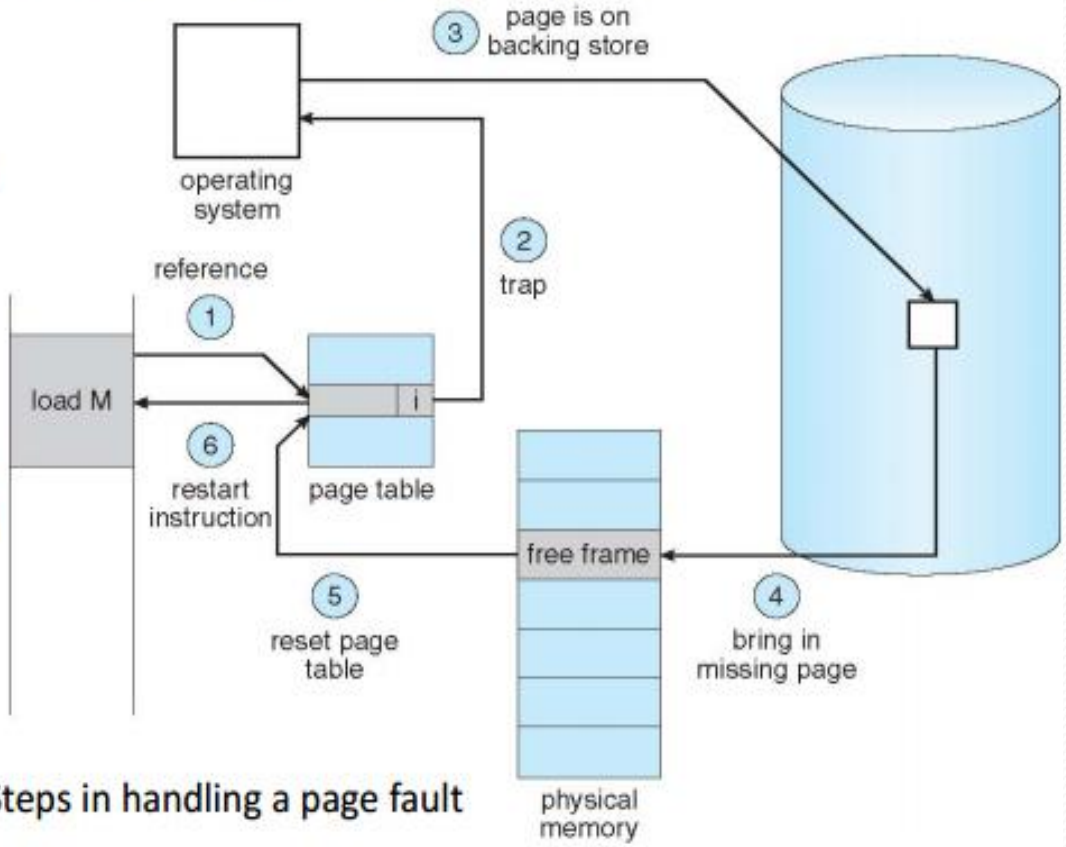
Page Fault

What happens if the process tries to access a page that was not brought into memory?

Access to a page marked invalid causes a *Page Fault*. (1)

The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.

2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Get empty frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory. Set validation bit = **v**
6. Restart the instruction that caused the page fault



Steps in handling a page fault

Page Replacement

What Happens if There is no Free Frame?

The OS has to choose a page to remove from memory to make the room for the page that has to be brought in.

Which one page to be removed ?

What happen if the page that required next, is removed?

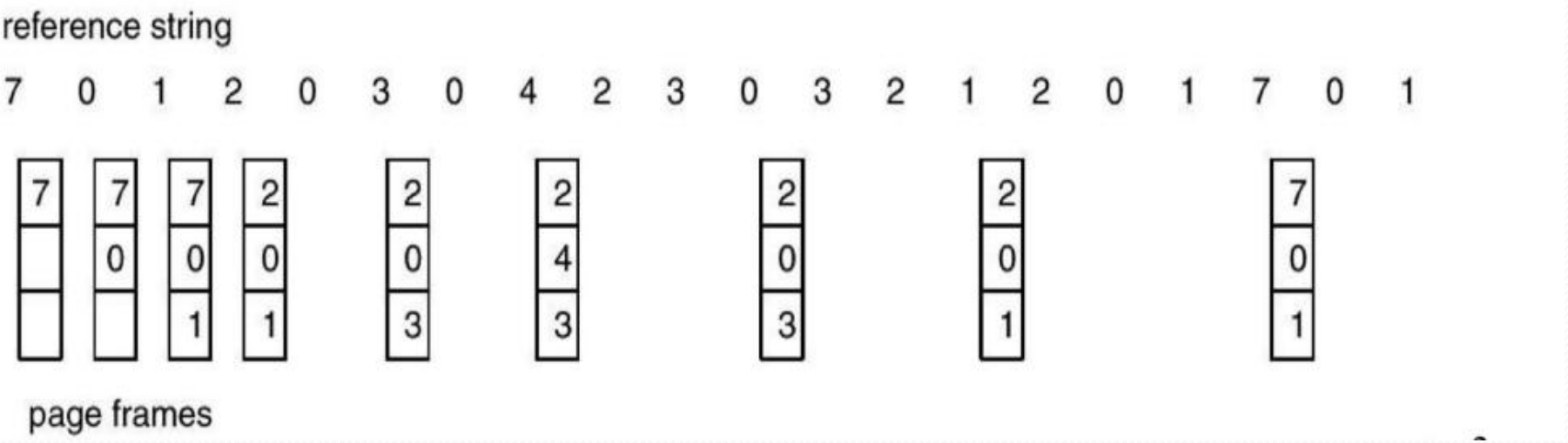
Principle of Optimality: To obtain optimal performance the page to replace is one that will not be used for the furthest time in the future.

Optimal Page Replacement (OPR)

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

Each page can be labeled with number of instructions that will be executed before that page is first referenced.

Ex: For 3 - page frames and 8 pages system the optimal page replacement is as:



Optimal Page Replacement (OPR)

Advantages:

An optimal page-replacement algorithm; it guarantees the lowest possible page fault rate.

Problems:

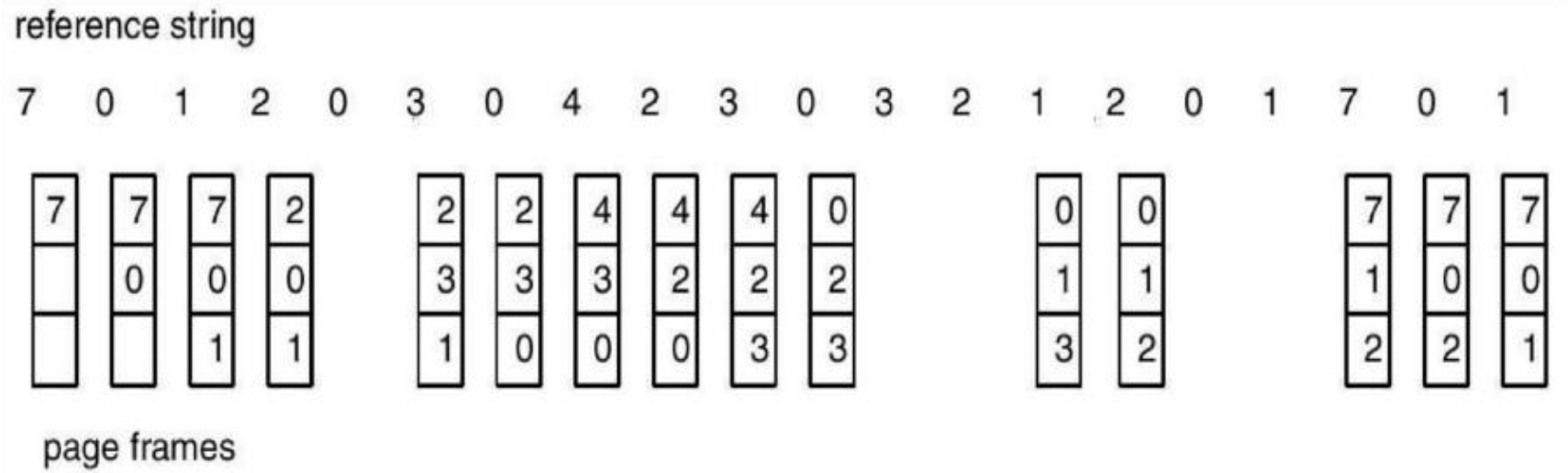
Unrealizable, at the time of the page fault, the OS has no way of knowing when each of the pages will be referenced next.

This is not used in practical system.

First-In-First-Out (FIFO)

This associates with each page the time when that page was brought into the memory. The page with highest time is chosen to replace.

This can also be implemented by using queue of all pages in memory.



First-In-First-Out (FIFO)

Advantages:

Easy to understand and program.

Distributes fair chance to all.

Problems:

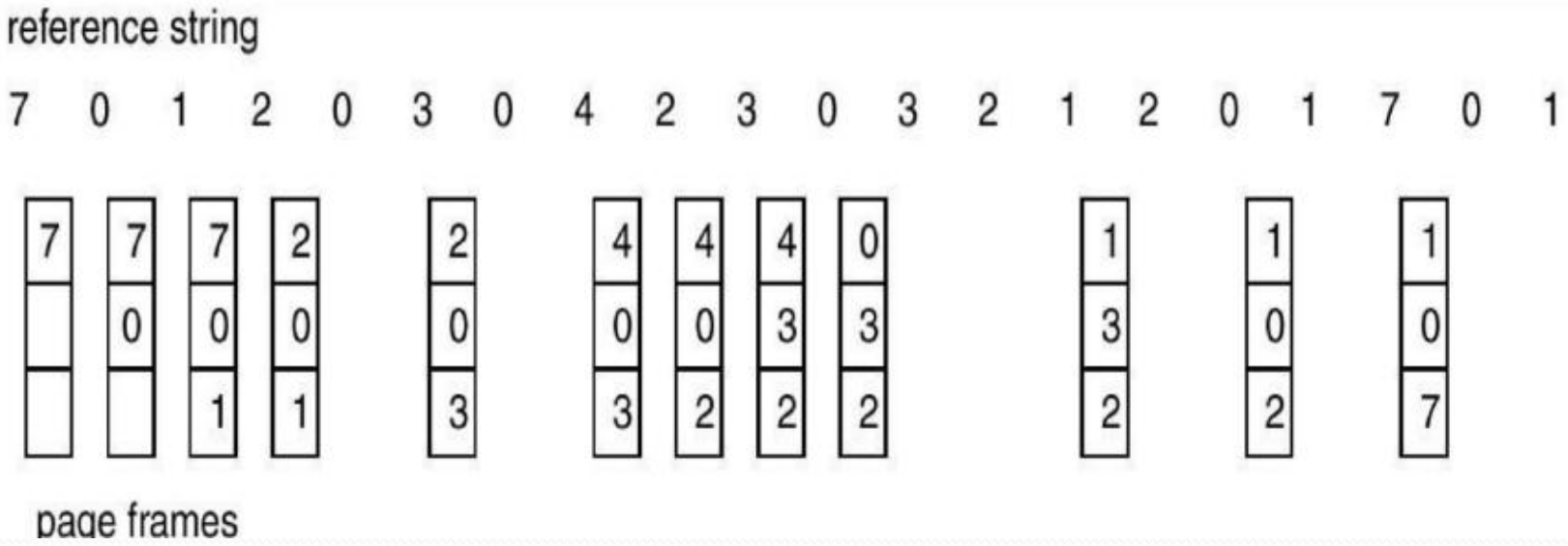
FIFO is likely to replace heavily (or constantly) used pages and they are still needed for further processing.

Least Recently Used (LRU)

Recent past is a good indicator of the near future.

When a page fault occurs, throw out the page that has been unused for longest time.

It maintain a linked list of all pages in memory with the most recently used page at the front and least recently used page at the rare. The list must be updated on every memory reference.



Least Recently Used (LRU)

Advantages:

Excellent, efficient is close to the optimal algorithm. Problems:

Difficult to implement exactly.

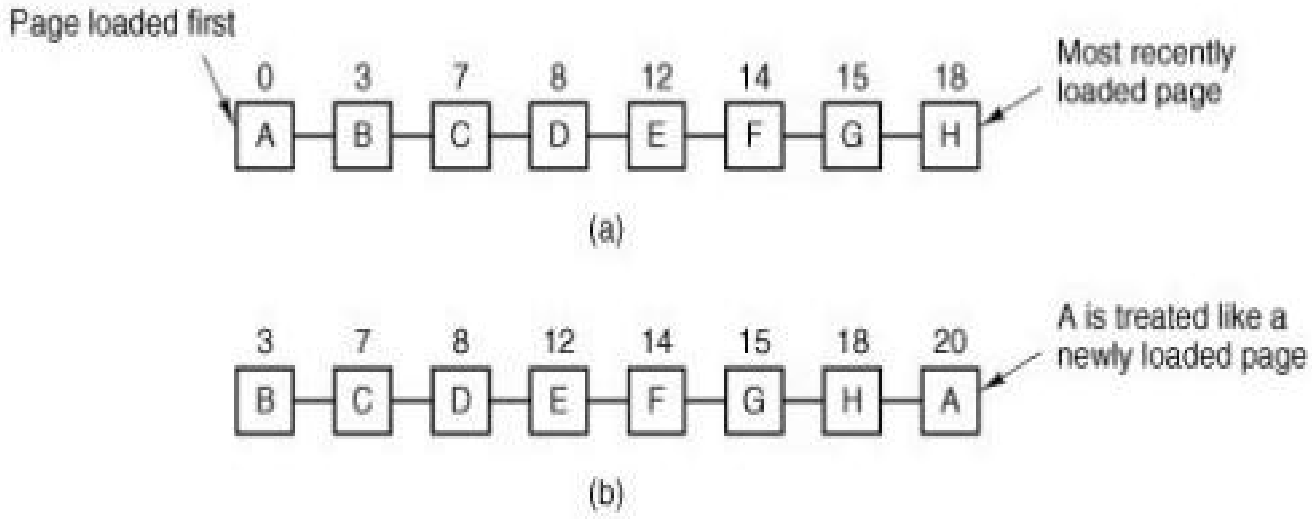
How to find good heuristic?

If it is implemented as linked list, updating list in every reference is not a way making system fast!

The Alternate implementation is by hardware primitivs, it requires a time-of-use field in page table and a logical clock or counter in the CPU.

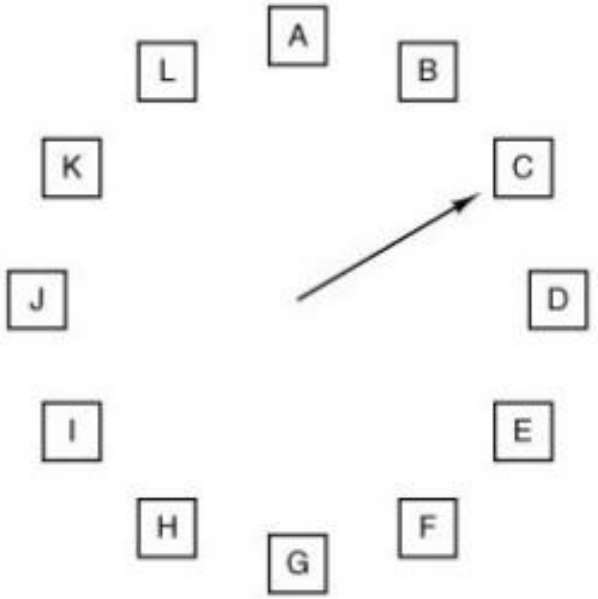
The Second Chance Page Replacement Algorithm:

A simple modification to FIFO that avoids the problem of heavily used page. It inspects the R bit If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.



The Clock Page Replacement Algorithm

keep all the page frames on a circular list in the form of a clock, as shown in Fig. A hand points to the oldest page.



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

When a page fault occurs, the page being pointed to by the hand is inspected. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0

END