Unit-4
Memory Organization

- Memory Hierarchy
- Main Memory
- Associative Memory
- Cache Memory: Cache Mapping techniques
- Virtual Memory

Memory Hierarchy

- Memory unit is essential component of digital computer since it is needed for storing programs and data.
- Memory unit that communicates directly with CPU is called Main memory.
- Devices that provide backup storage is called auxiliary memory.
- Only programs and data currently needed by processor reside in the main memory.
- All other information is stored in auxiliary memory and transferred to main memory whenever needed.

Table 4.1 Key Characteristics of Computer Memory Systems

Location
- Internal (e.g. processor registers, main memory, cache)
- External (e.g. optical disks, magnetic disks, tapes)

Capacity
- Number of words
- Number of bytes

Unit of Transfer
- Word
- Block

Access Method
- Sequential
- Direct
- Random
- Associative

Performance
- Access time
- Cycle time
- Transfer rate

Physical Type
- Semiconductor
- Magnetic
- Optical
- Magneto-optical

Physical Characteristics
- Volatile/nonvolatile
- Erasable/nonerasable

Organization
- Memory modules

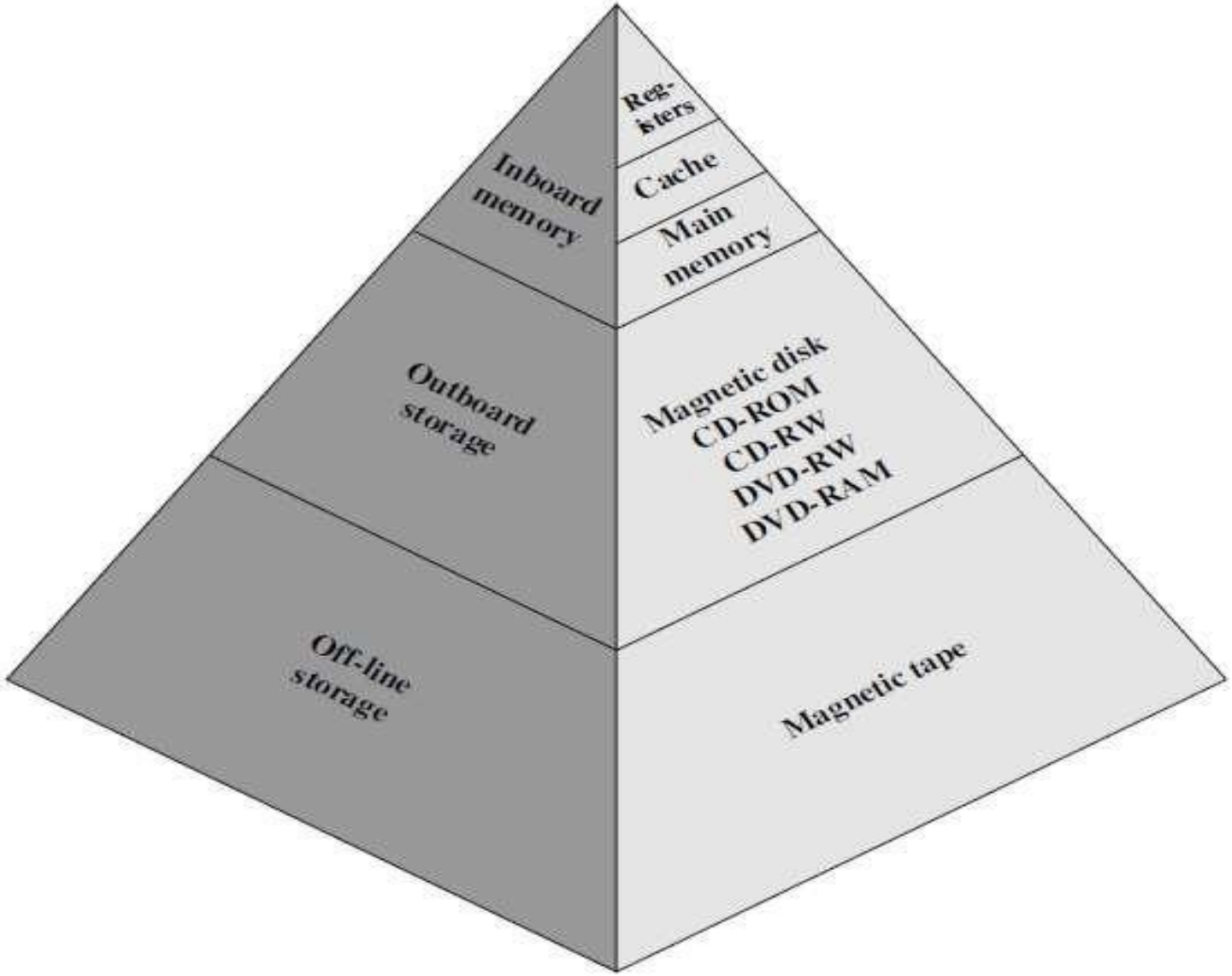


Figure 4.1 The Memory Hierarchy

Memory Hierarchy

- Memory hierarchy system consist of all storage devices from auxiliary memory to main memory to cache memory
- As one goes down in the hierarchy:
  - Capacity increases.
  - Cost per bit decreases.
  - Access time increases.
  - Frequency of access by the processor decreases.

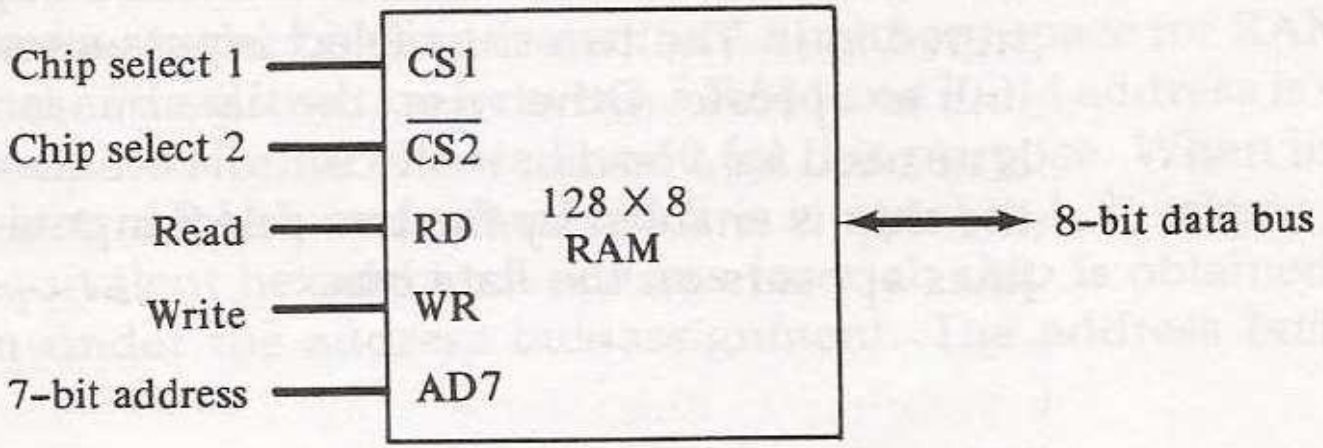
Main Memory

- It is the memory used to store programs and data during the computer operation.
- The principal technology is based on semiconductor integrated circuits.
- It consists of RAM and ROM chips.
- RAM chips are available in two form static and dynamic.

Main Memory

SRAM	DRAM
Uses capacitor for storing information	Uses Flip flop
More cells per unit area due to smaller cell size. PD is less	Needs more space for same capacity. PD is more
Cheaper and smaller in size	Expensive and bigger in size
Slower	Faster
Requires refresh circuit	No need
Used in main memory	Used in cache

Figure Typical RAM chip.



(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

ROM Memory

- ROM is uses random access method.
- It is used for storing programs that are permanent and the tables of constants that do not change.
- ROM store program called bootstrap loader whose function is to start the computer software when the power is turned on.
- When the power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader.

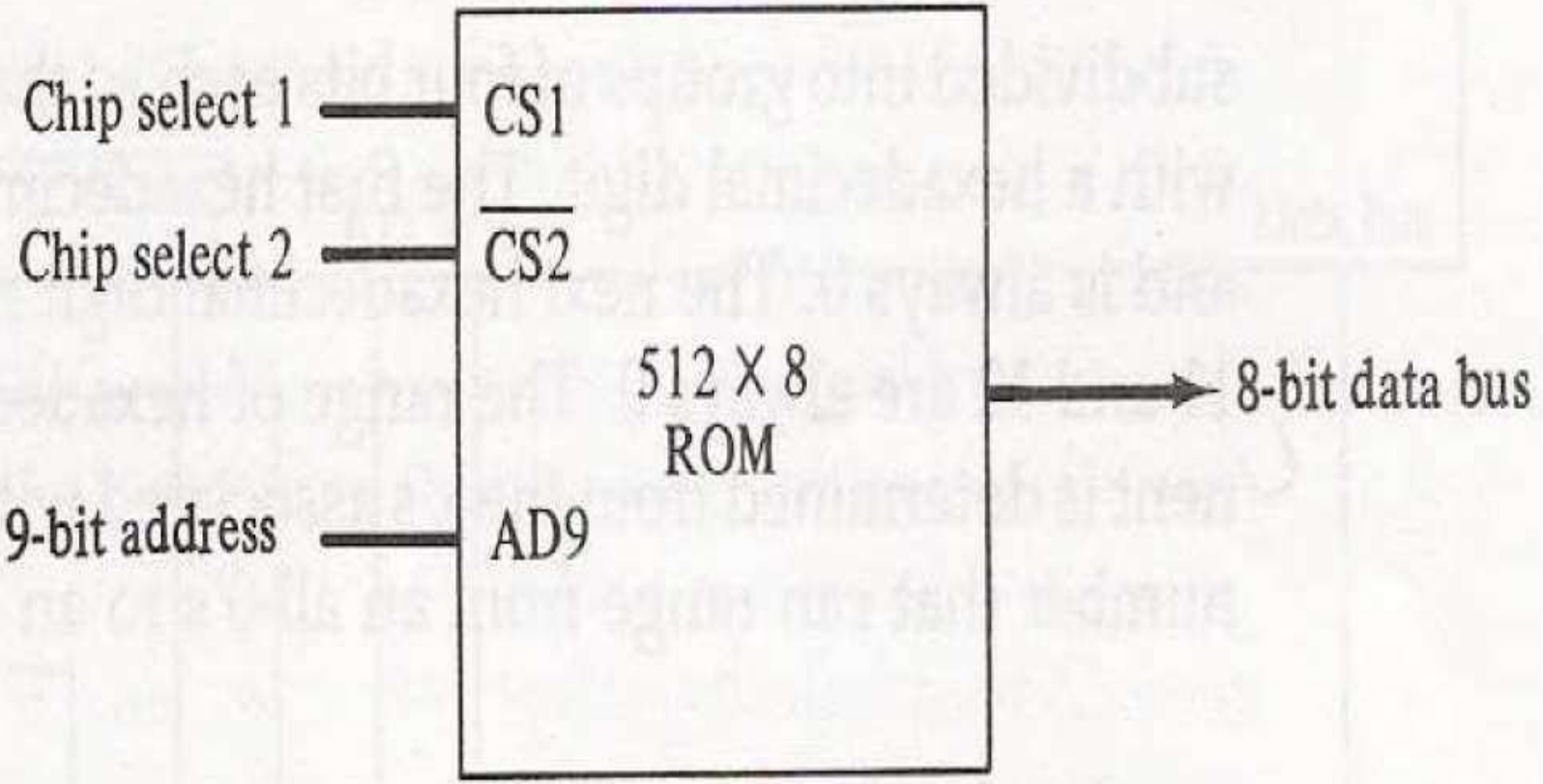


Figure Typical ROM chip.

ROM Memory

- For the same size chip it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM,
- For this reason the diagram specifies 512 byte ROM and 128 bytes RAM.

Memory address Map

- Designer must specify the size and the type (RAM or ROM) of memory to be used for particular application.
- The addressing of the memory is then established by means of table called memory address map that specifies the memory addresses assign to each chip.
- Let us consider an example in which computer needs 512 bytes of RAM and ROM as well and we have to use the chips of size 128 bytes for RAM and 512 bytes for ROM.

TABLE Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus 10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

Recap of Last Lecture

- Memory Hierarchy
- Main Memory
- Organization of Main Memory
- Memory Address Map

Memory Connection with CPU

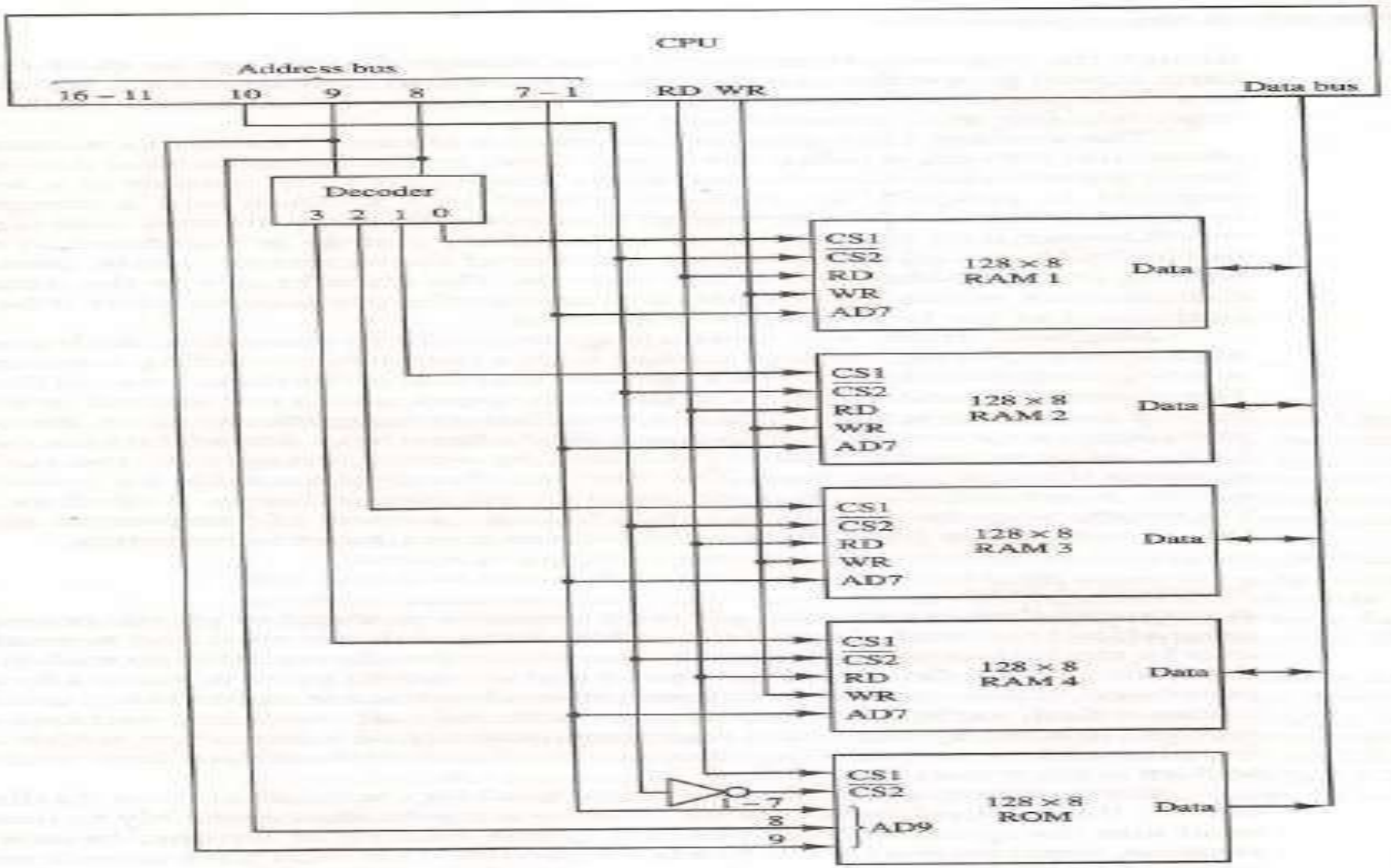


Figure 12-4 Memory connection to the CPU.

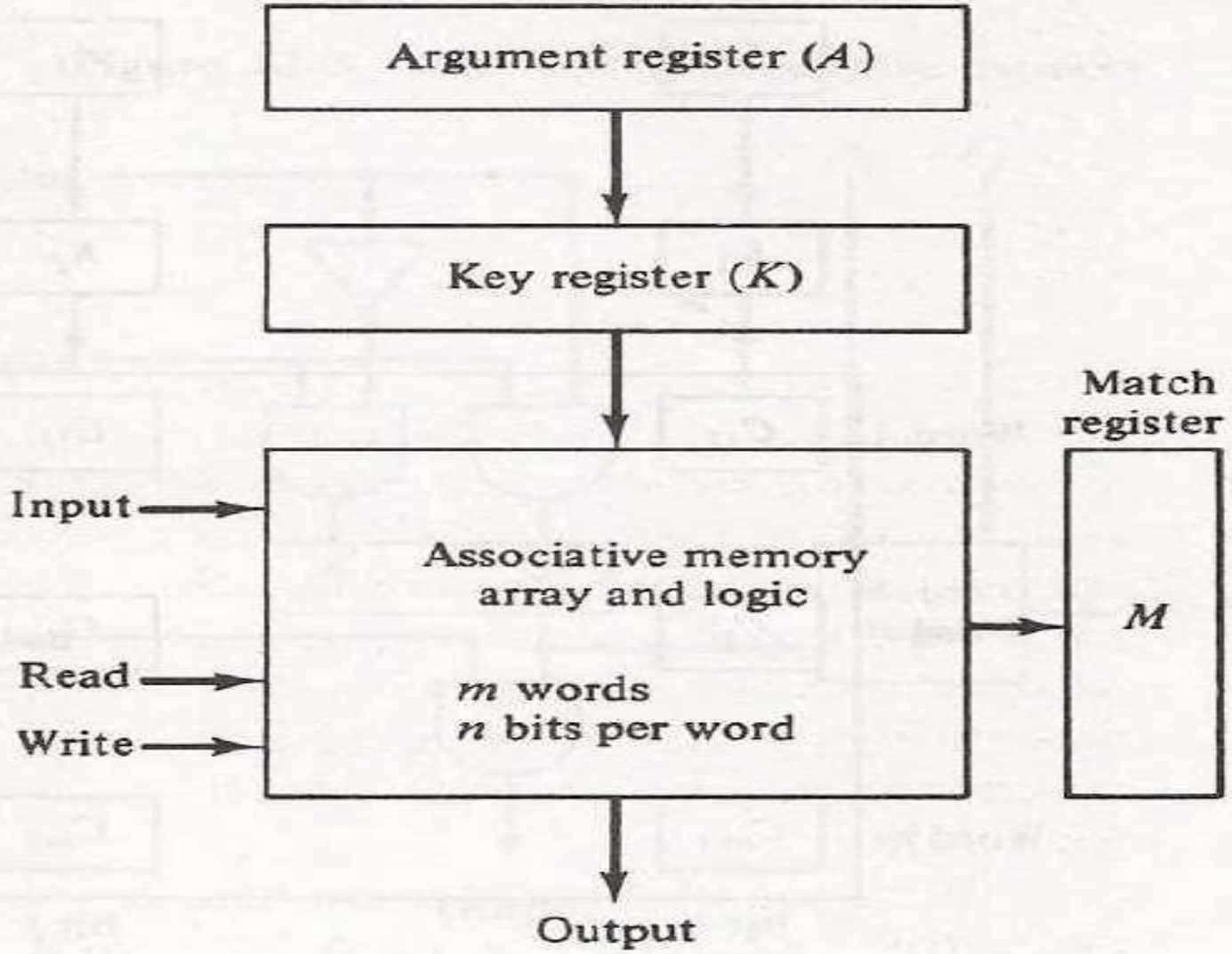
Associative Memory

- To search a particular data in memory, data is read from certain address and compared if the match is not found, content of the next address is accessed and compared.
- This goes on until required data is found. The number of access depend on the location of data and efficiency of searching algorithm.
- The searching time can be reduced if data is searched on the basis of content.

Associative Memory

- A memory unit accessed by content is called associative memory or content addressable memory (CAM)
- This type of memory is accessed simultaneously and in parallel on the basis of data content.
- Memory is capable of finding empty unused location to store the word.
- These are used in the application where search time is very critical and must be very short.

Figure Block diagram of associative memory.



Associative Memory

- It consists memory array of m words with n bits per words
- Argument register A and key register K have n bits one for each bit of word.
- Match register has m bits, one for each memory word.
- Each word in memory is compared in parallel with the content of the A register. For the word that match corresponding bit in the match register is set.

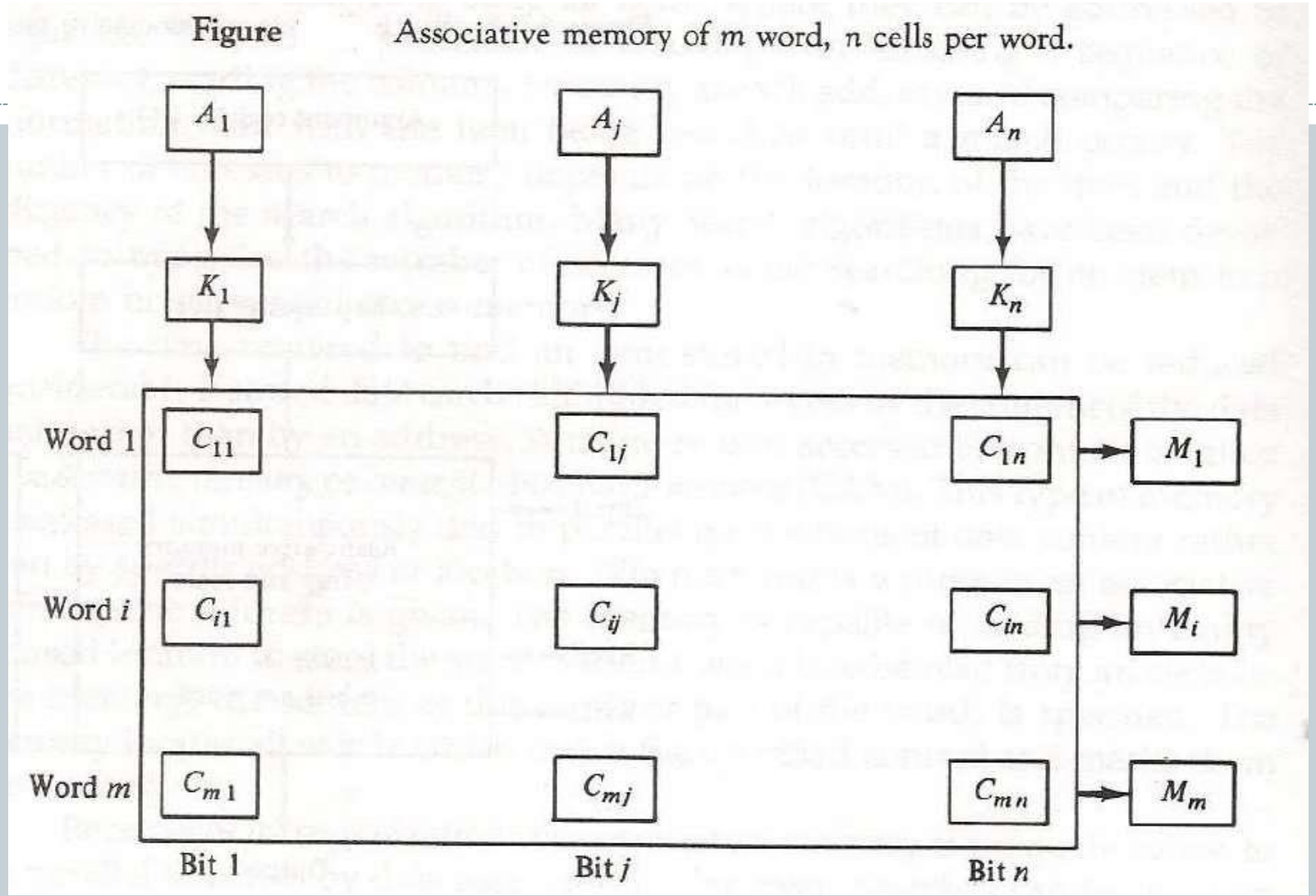
Associative Memory

- Key register provide the mask for choosing the particular field in A register.
- The entire content of A register is compared if key register content all 1.
- Otherwise only bit that have 1 in key register are compared.
- If the compared data is matched corresponding bits in the match register are set.
- Reading is accomplished by sequential access in memory for those words whose bit are set.

Example for Associative Memory

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Figure Associative memory of m word, n cells per word.



Match Logic

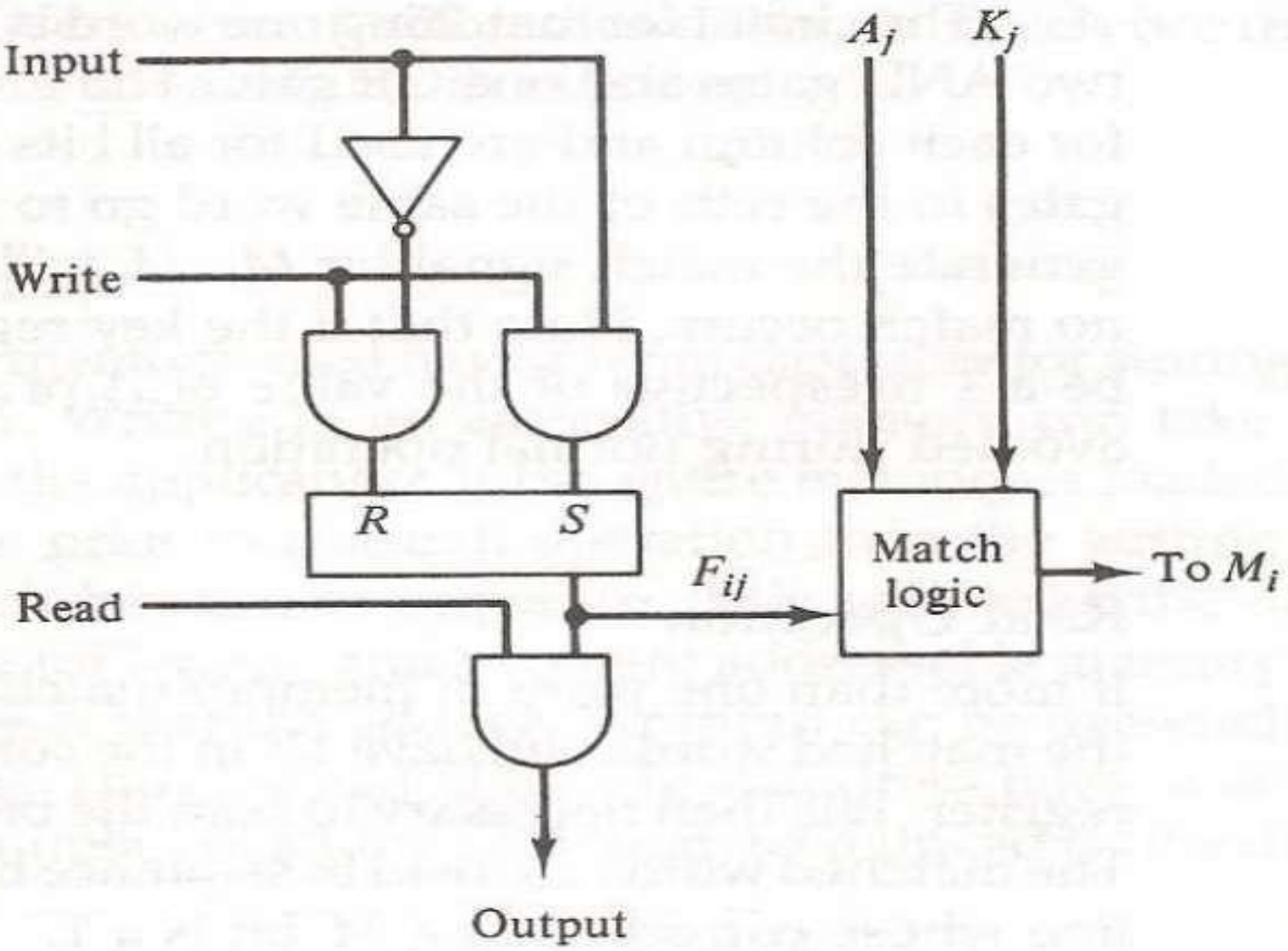
- Let us neglect the key register and compare the content of argument register with memory content.
- Word i is equal to argument in A if $A_j = F_{ij}$ for j=1,2,3,4……..n
- The equality of two bits is expressed as

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

- x_j =1 if bits are equal and 0 otherwise.

$$M_i = x_1 x_2 x_3 \cdots x_n$$

Figure One cell of associative memory.



Match Logic

- Let us include key register. If K_j=0 then there is no need to compare A_j and F_{ij}.
- Only when K_j=1, comparison is needed.
- This achieved by ORing each term with K_j.

$$M_i = (x_1 + K_1')(x_2 + K_2')(x_3 + K_3') \cdots (x_n + K_n')$$

If we substitute the original definition of x_j, the Boolean function above can be expressed as follows:

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

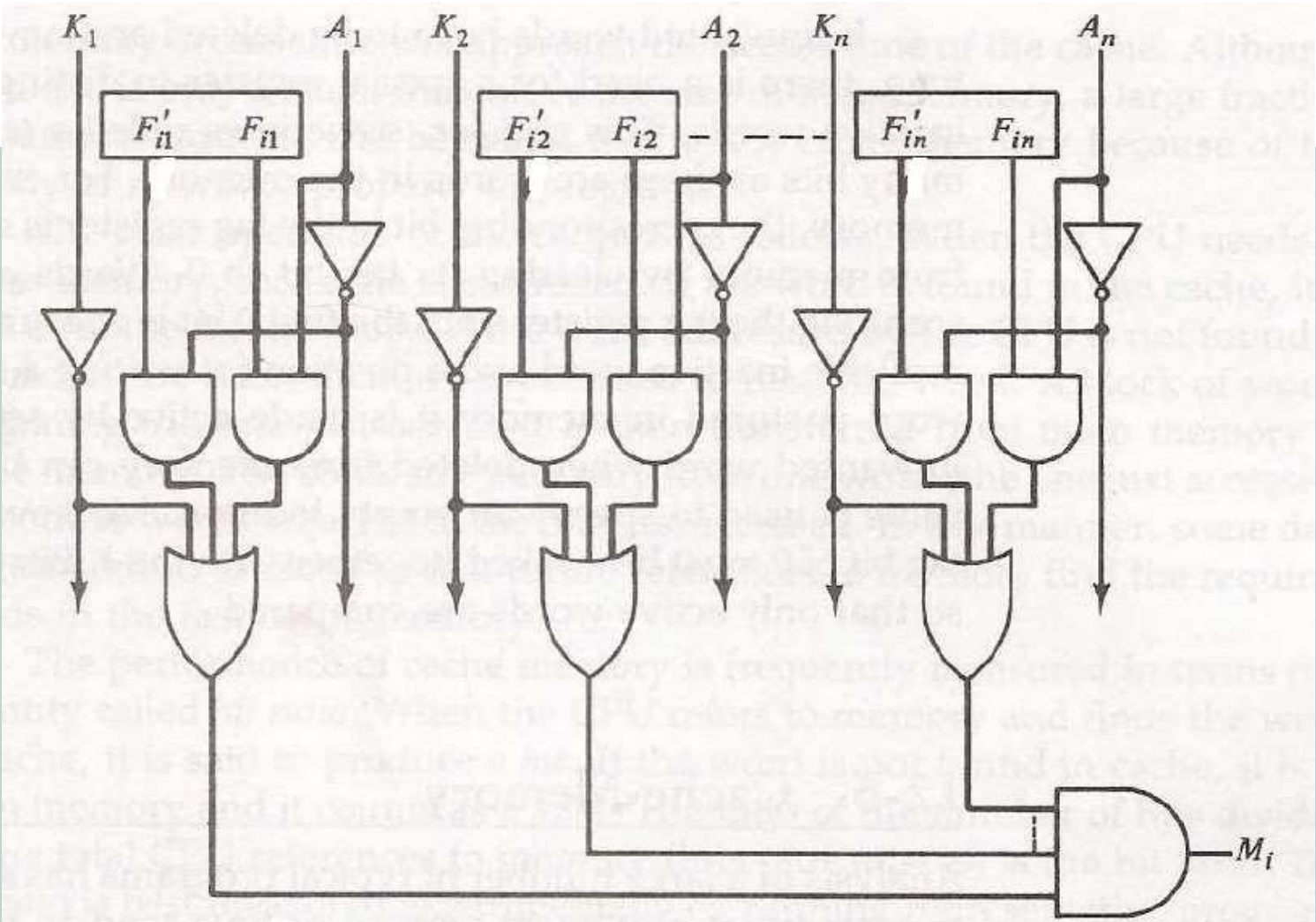


Figure Match logic for one word of associative memory.

Read Operation

- If more than one word match with the content, all the matched words will have 1‘s in the corresponding bit position in match register.
- Matched words are then read in sequence by applying a read signal to each word line.
- In most application, the associative memory stores a table with no two identical items under a given key.

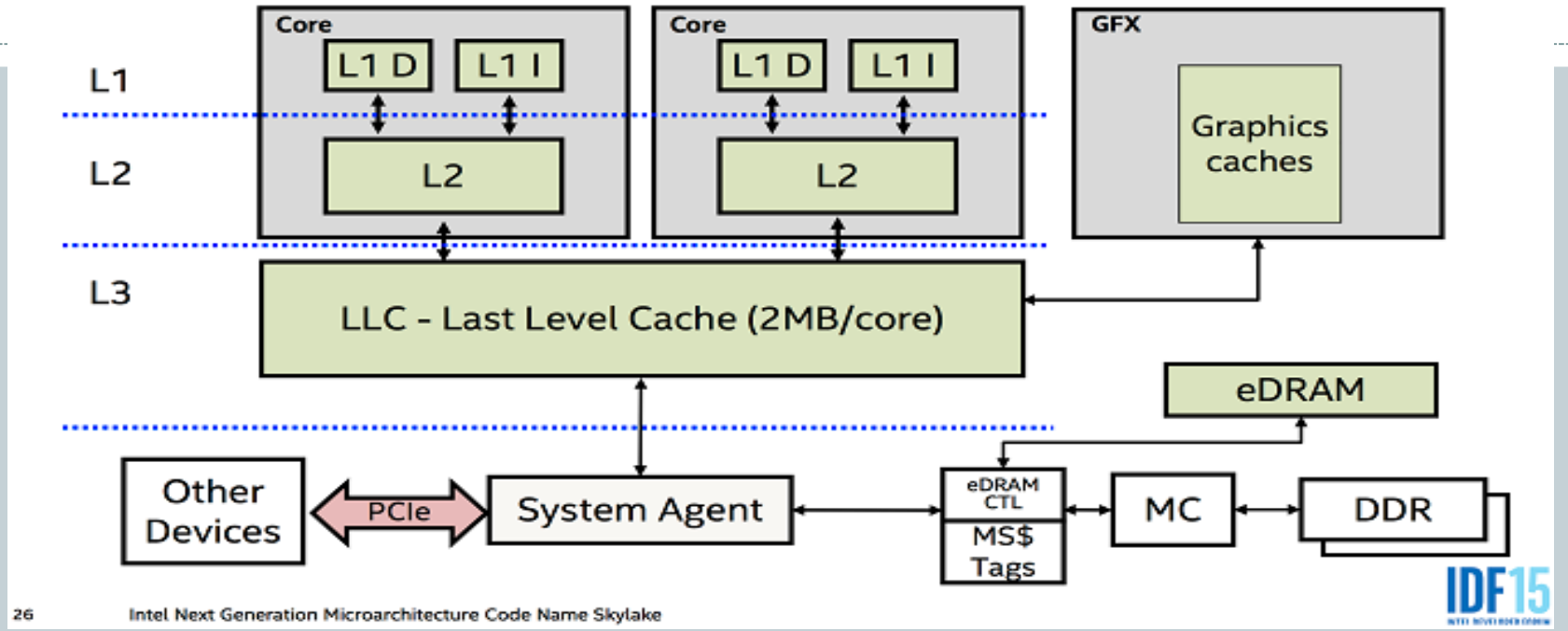
Write Operation

- If the entire memory is loaded with new information at once prior to search operation then writing can be done by addressing each location in sequence.
- Tag register contain as many bits as there are words in memory.
- It contain 1 for active word and 0 for inactive word.
- If the word is to be inserted, tag register is scanned until 0 is found and word is written at that position and bit is change to 1.

Cache Memory

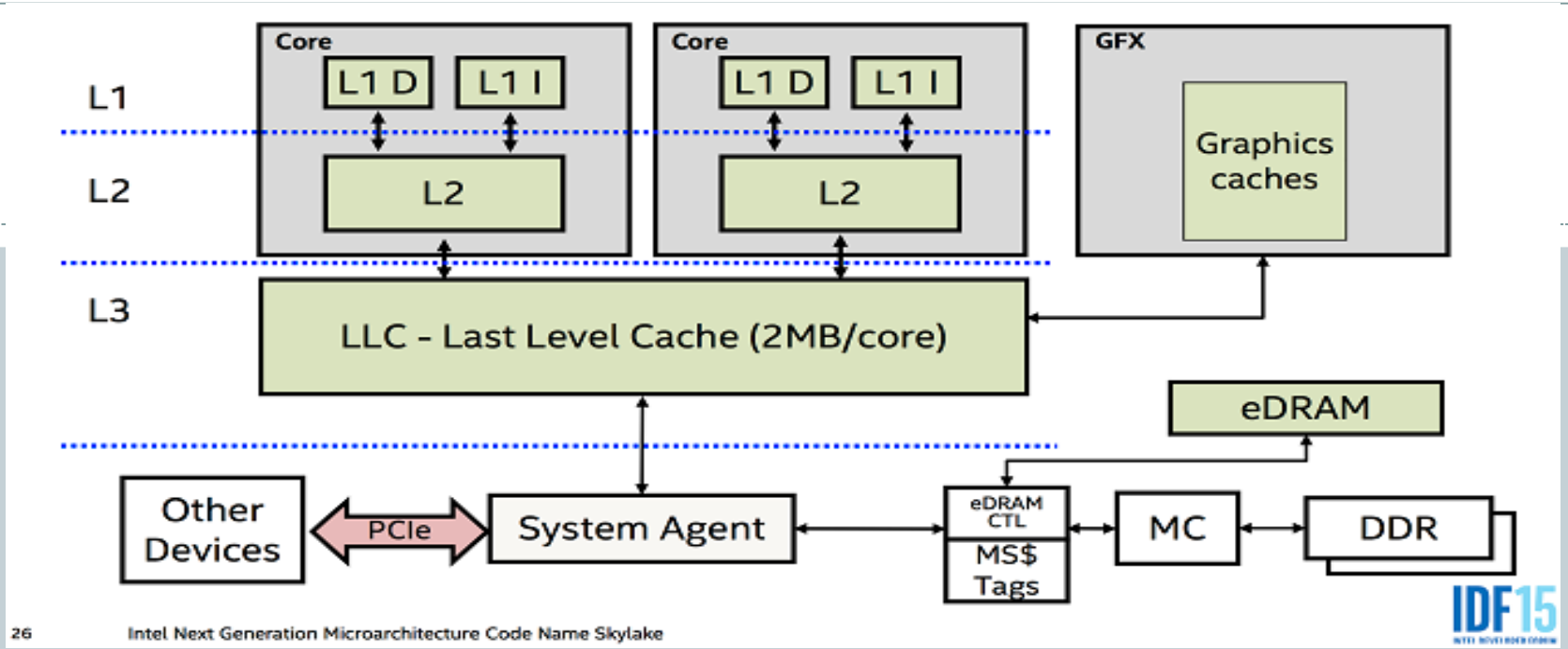
- With the analysis of large number of typical programs, shows that reference to memory at any given interval of time tend to be confined within a few localized area in the memory. This is known as locality of reference.
- A typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently.
- If the active portion of program and data are placed in fast memory, then average execution time of the program can be reduced. Such fast memory is called cache memory.
- It is placed in between the main memory and the CPU.

CPU cache is divided into three main 'Levels', L1, L2, and L3. The hierarchy here is again according to the speed, and thus, the size of the cache.



L1 (Level 1) cache is the fastest memory that is present in a computer system. In terms of priority of access, L1 cache has the data the CPU is most likely to need while completing a certain task. As far as the size goes, the L1 cache typically goes up to 256KB. However, some really powerful CPUs are now taking it close to 1MB. Some server chipsets (like Intel's top-end Xeon CPUs) now have somewhere between 1-2MB of L1 cache.

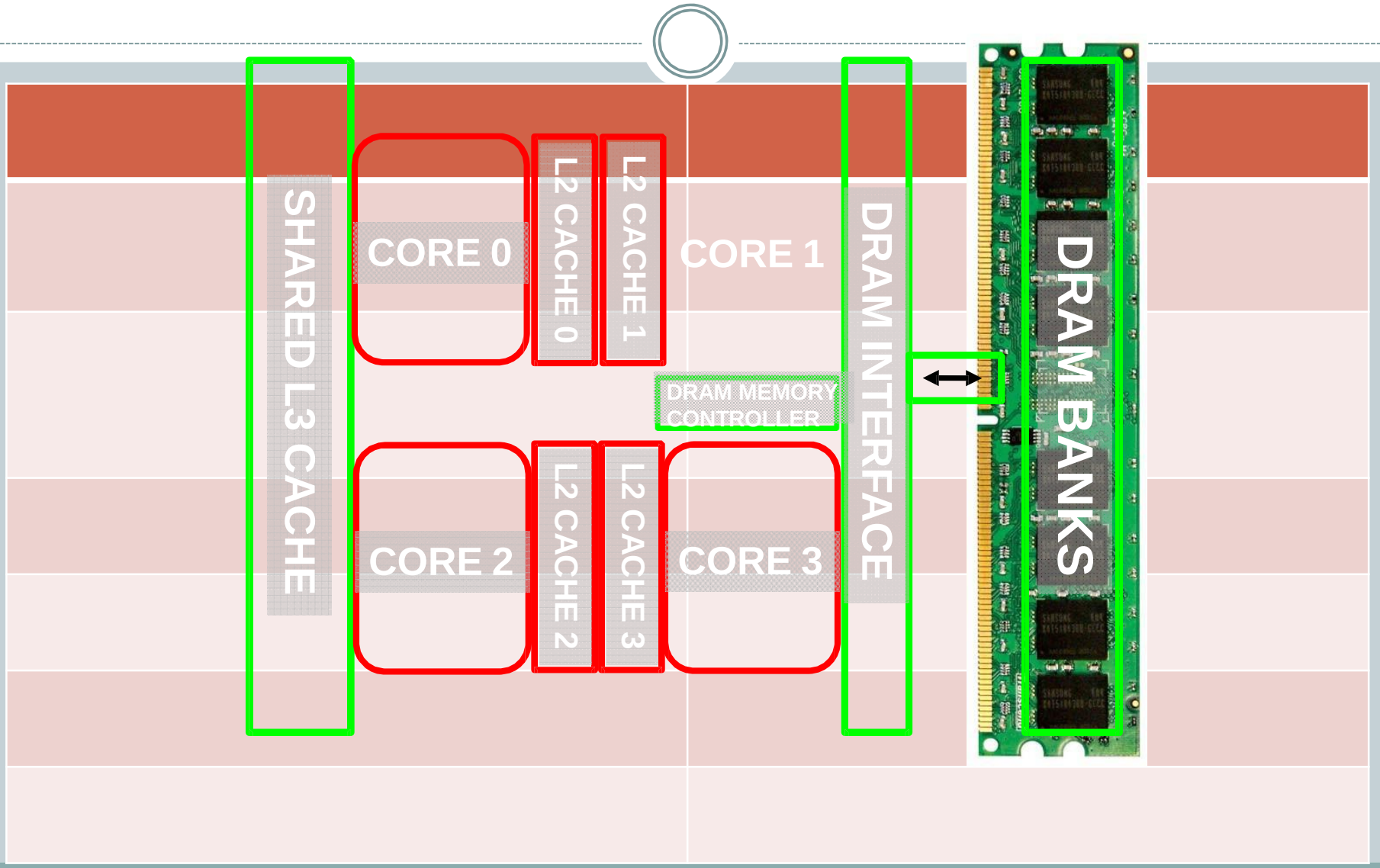
L1 cache is also usually split two ways, into the instruction cache and the data cache. The instruction cache deals with the information about the operation that the CPU has to perform, while the data cache holds the data on which the operation is to be performed.



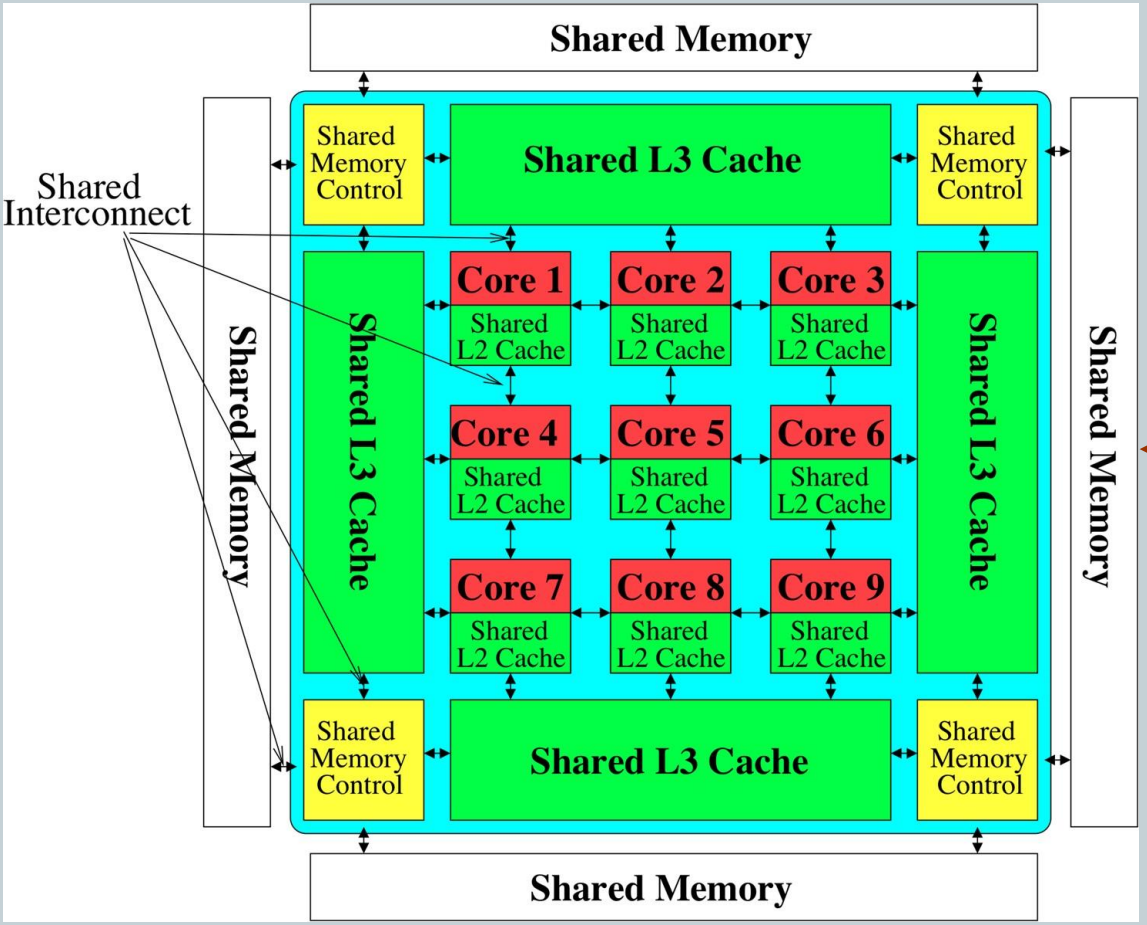
L2 (Level 2) cache is slower than L1 cache, but bigger in size. Its size typically varies between 256KB to 8MB, although the newer, powerful CPUs tend to go past that. L2 cache holds data that is likely to be accessed by the CPU next. In most modern CPUs, the L1 and L2 caches are present on the CPU cores themselves, with each core getting its own cache.

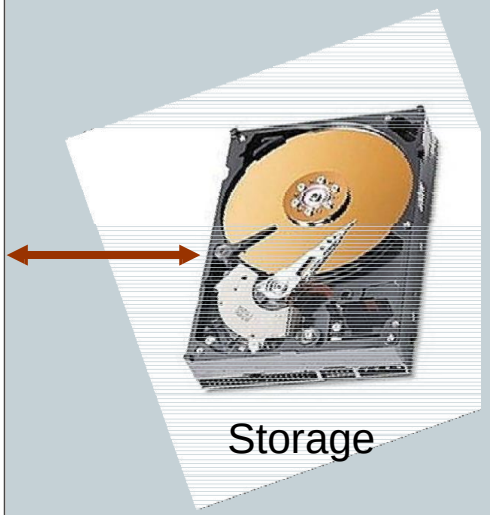
L3 (Level 3) cache is the largest cache memory unit, and also the slowest one. It can range between 4MB to upwards of 50MB. Modern CPUs have dedicated space on the CPU die for the L3 cache, and it takes up a large chunk of the space.

Top View of Memory Organization in Computer System

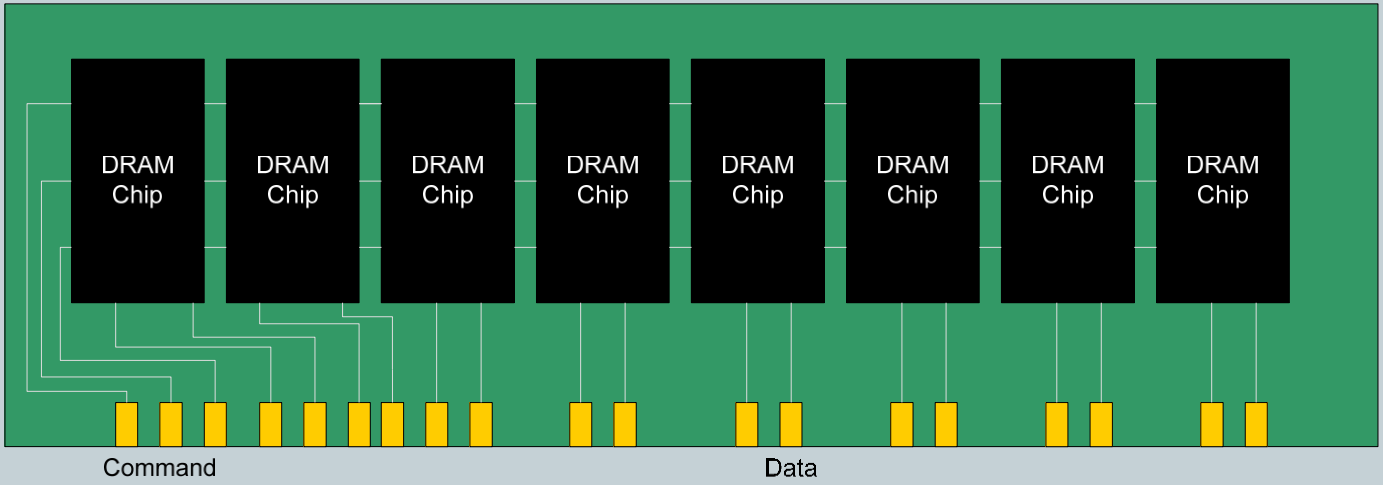


Memory System: A *Shared Resource* View

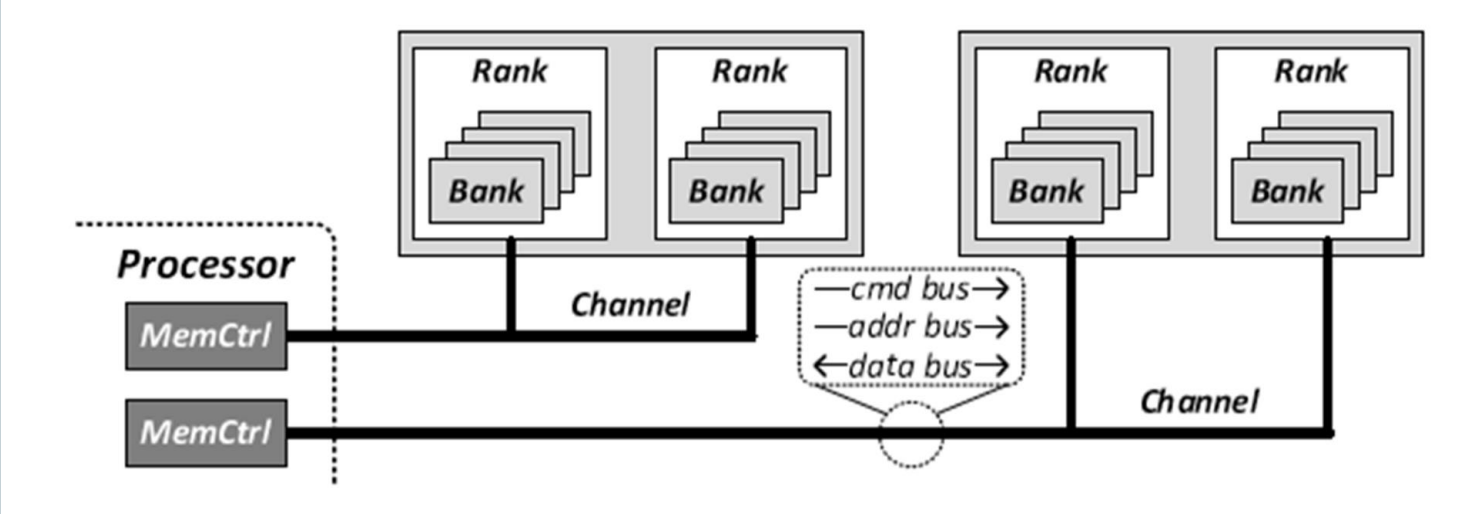




A 64-bit Wide DRAM



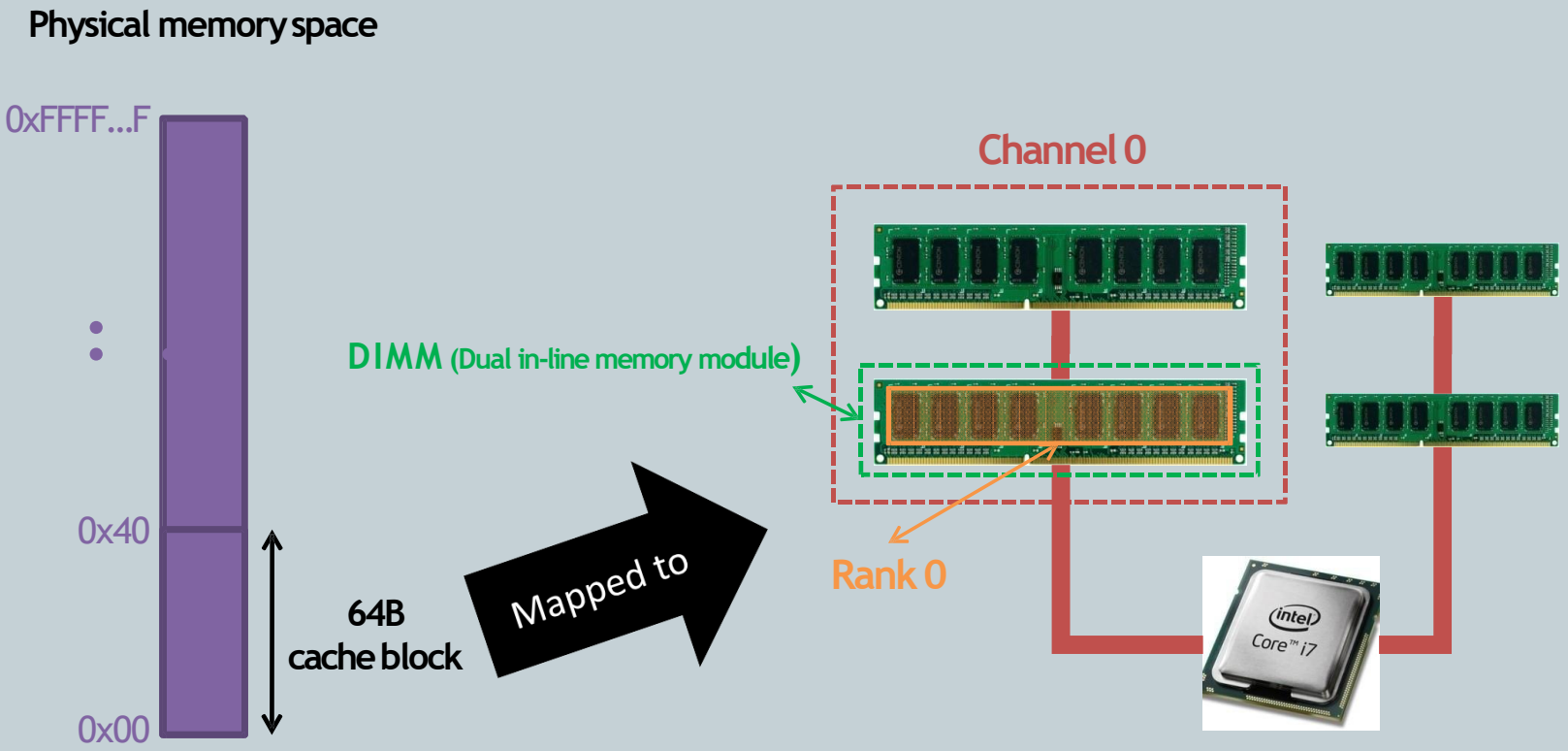
Generalized Memory Structure



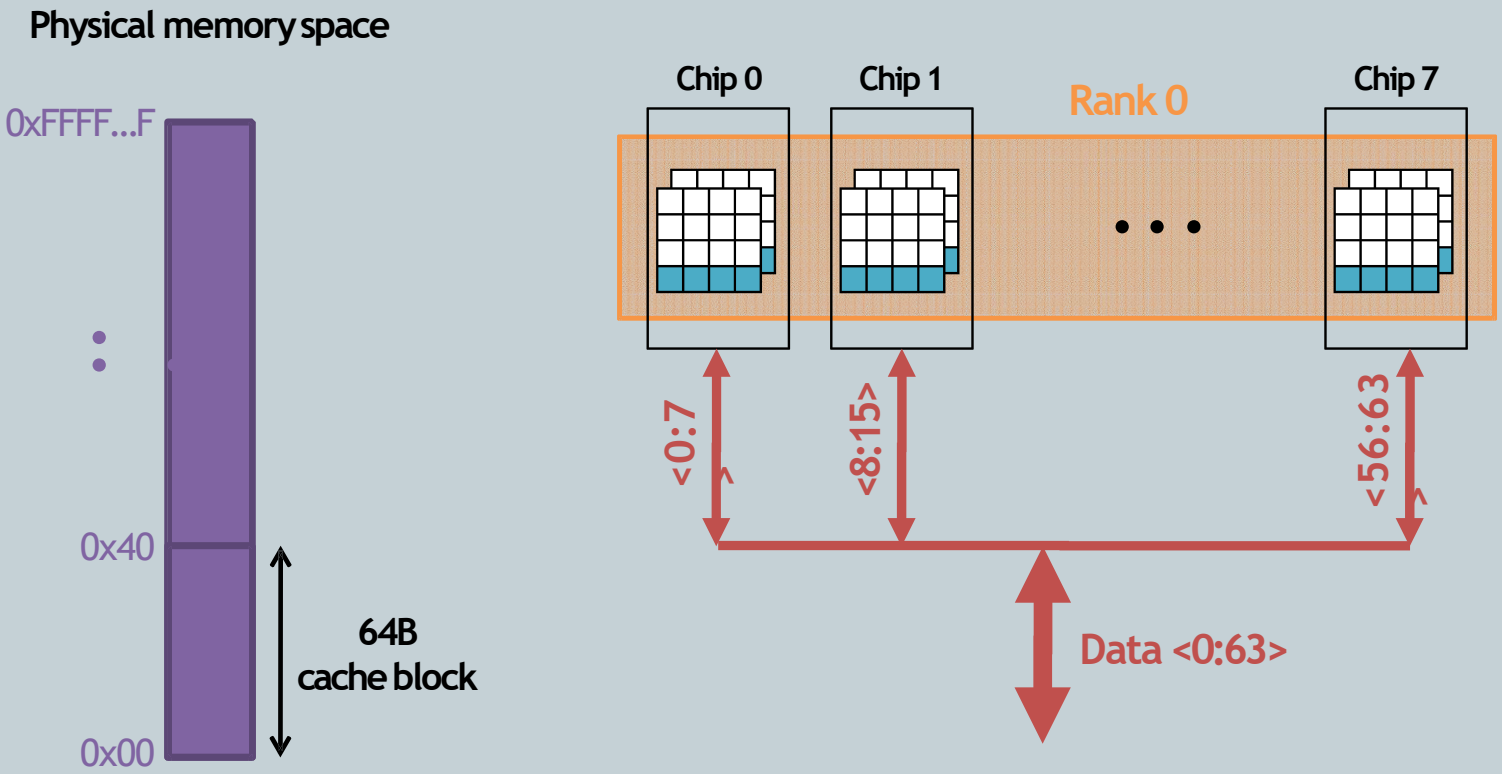
Basic Operation of Cache Memory

- When the CPU need to access the memory, it first search in cache. If word is found, it is read.
- If the word is not found, it is read from main memory and a block of data is transferred from main memory to cache which contain the current word.
- If the word is found in cache, it is said hit. If the word is not found, it is called miss.
- Performance of cache is measured in terms of hit ratio which is the ratio of total hit to total memory access by CPU.

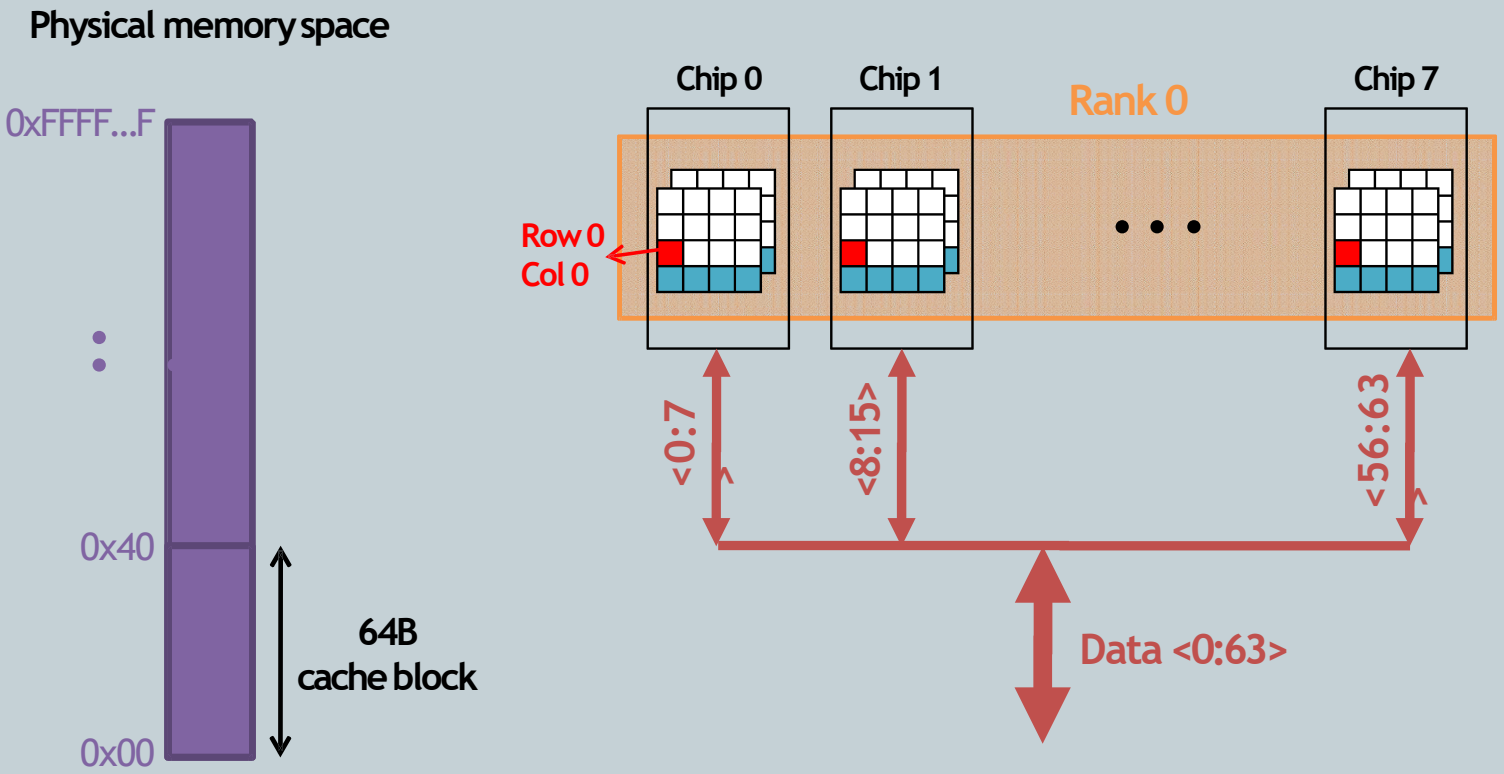
Example: Transferring a cache block



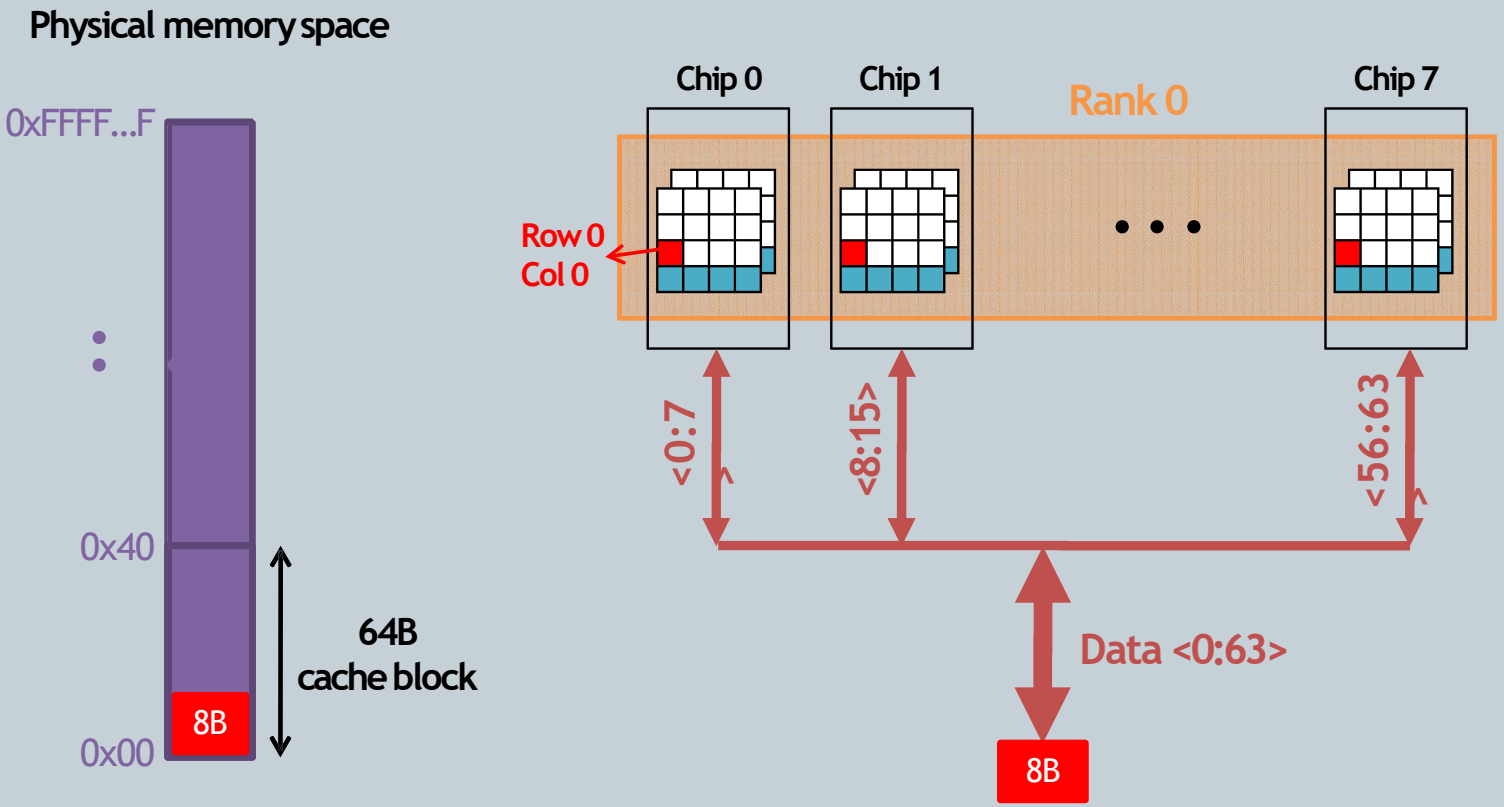
Example: Transferring a cache block



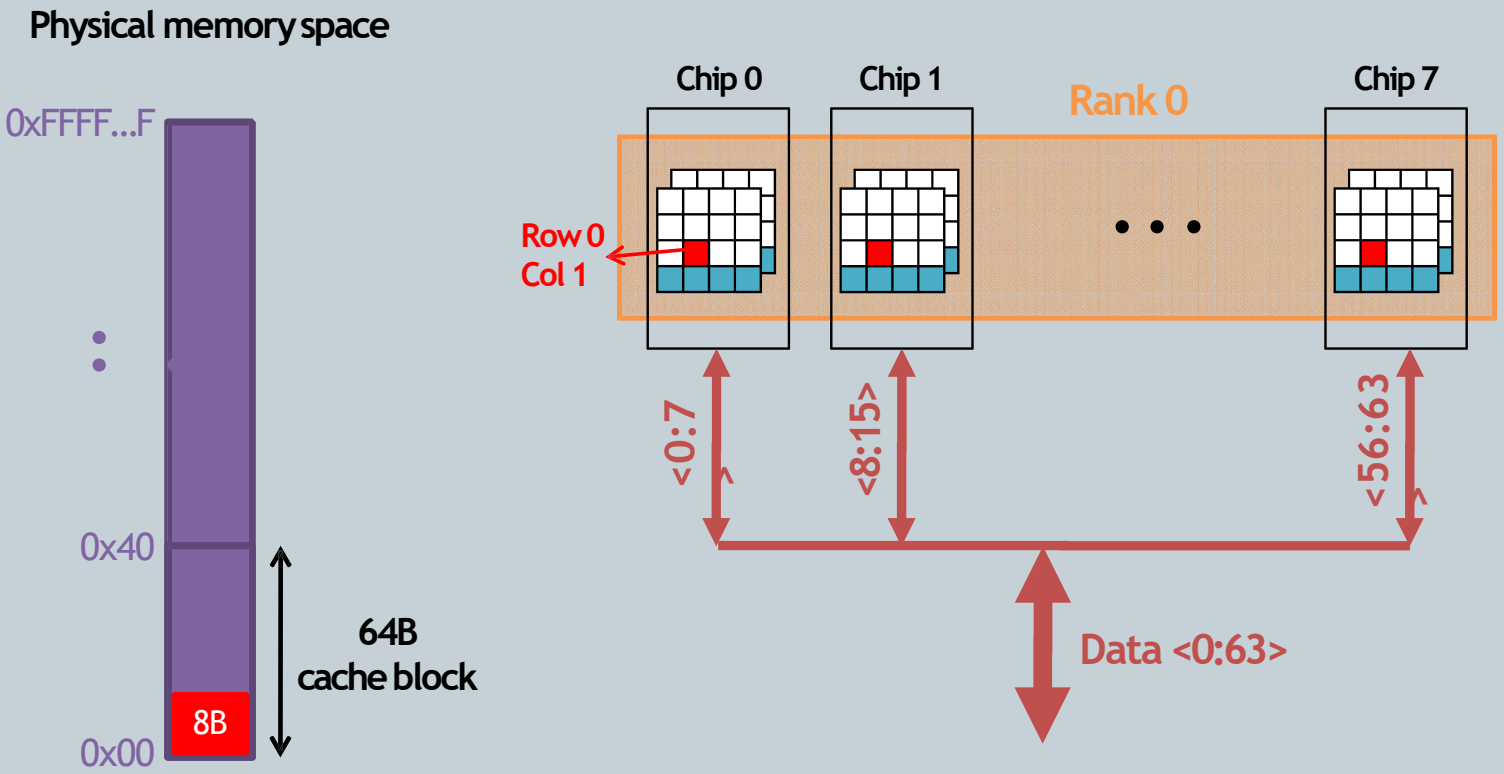
Example: Transferring a cacheblock



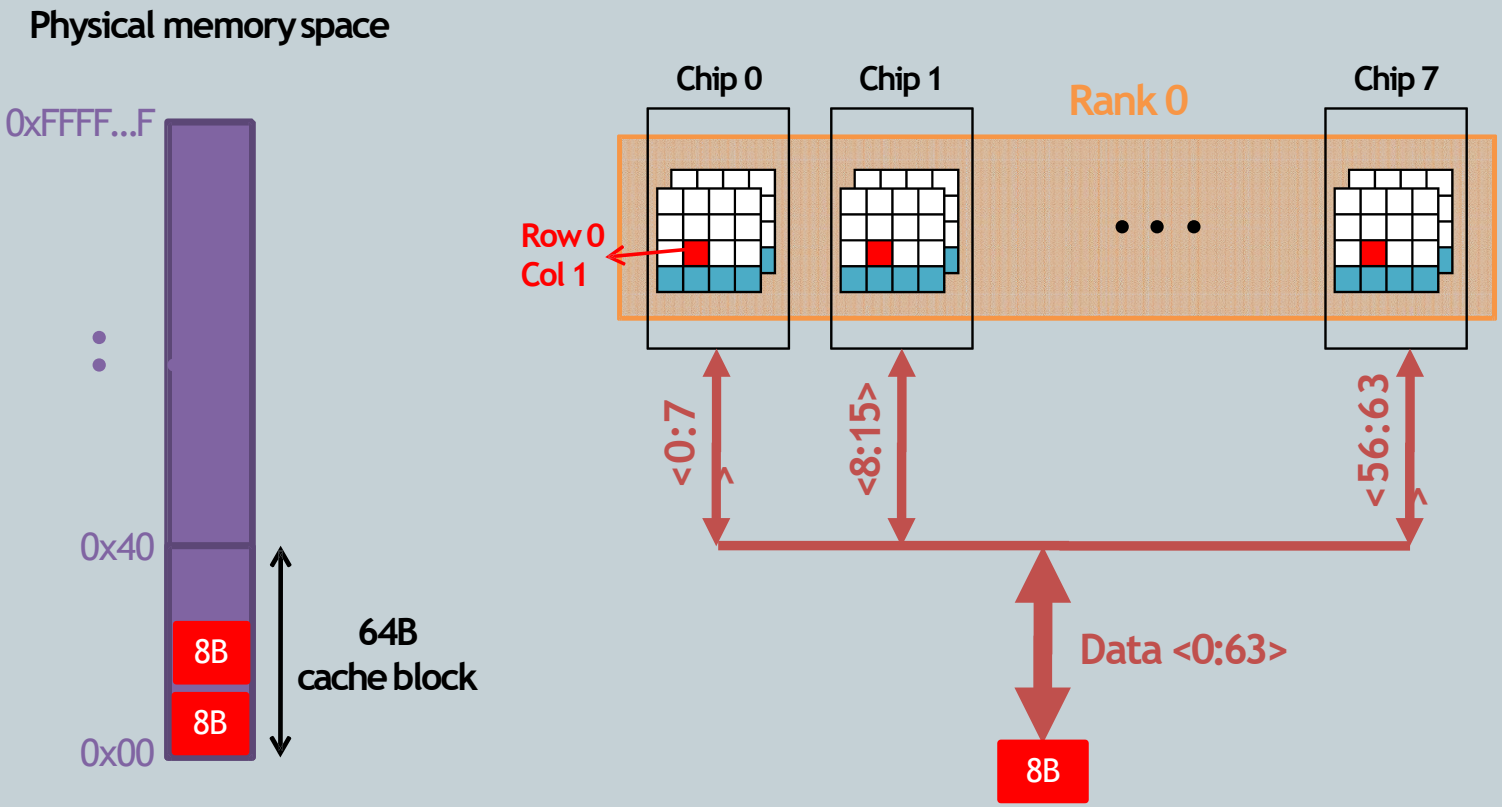
Example: Transferring a cache block



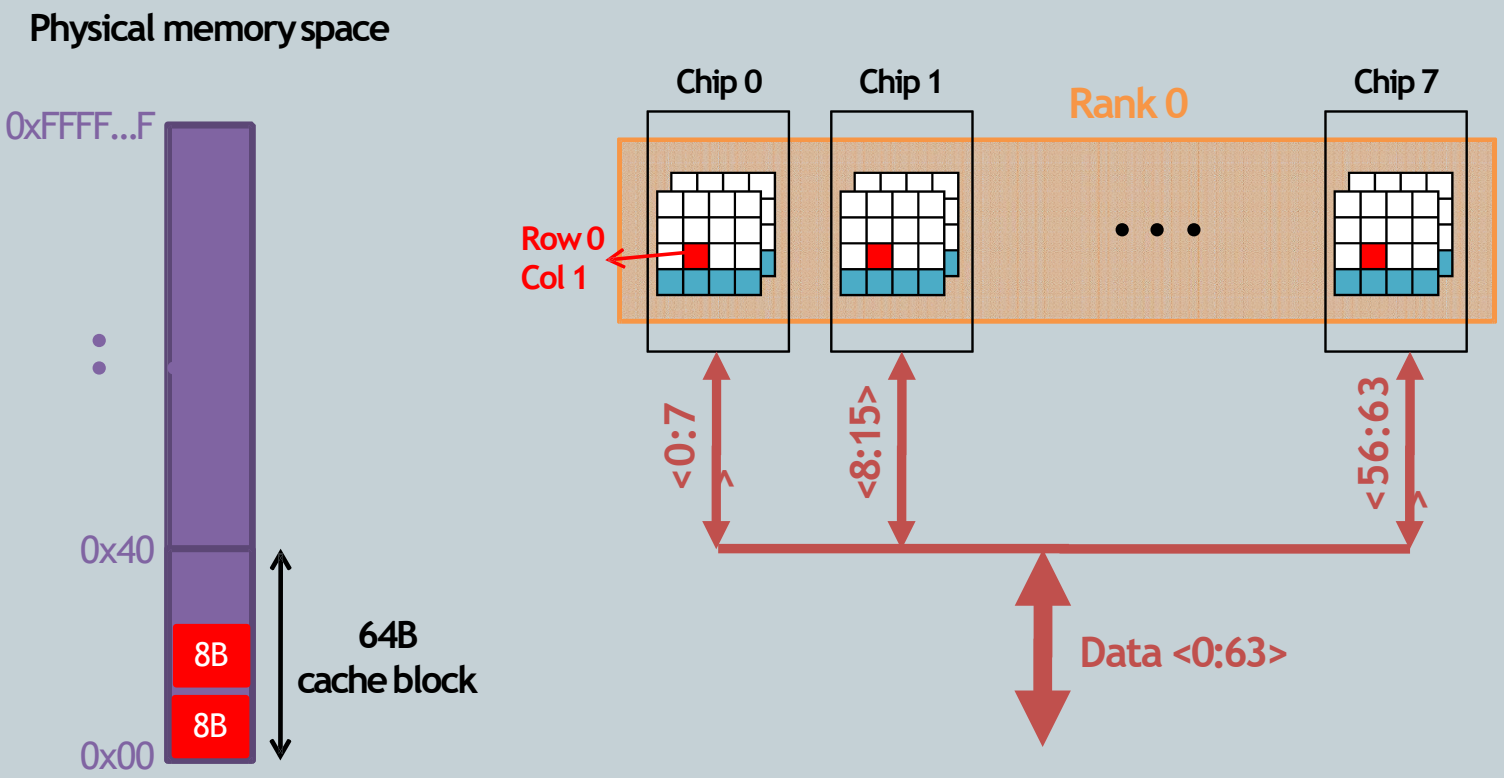
Example: Transferring a cache block



Example: Transferring a cache block



Example: Transferring a cache block



A 64B cache block takes 8 I/O cycles to transfer.

During the process, 8 columns are read sequentially.

Mapping Techniques

- The transformation of data from main memory to cache is known as mapping process. Three types of mapping procedures are:
  - Associative Mapping
  - Direct Mapping
  - Set-Associative Mapping

Cache Memory Organisation

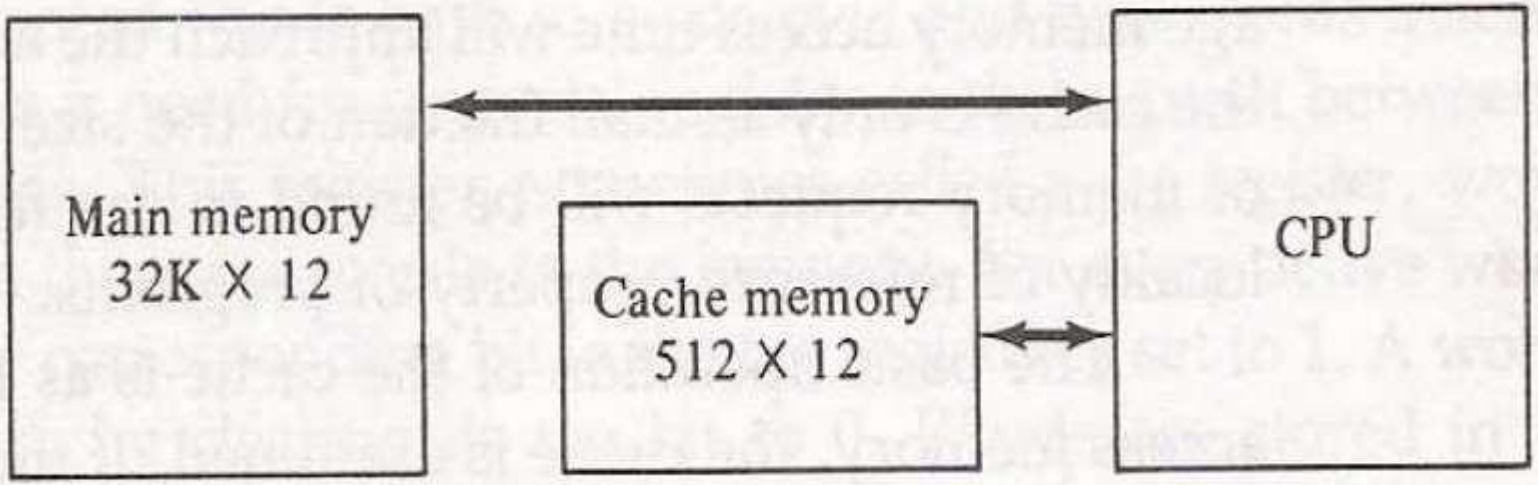


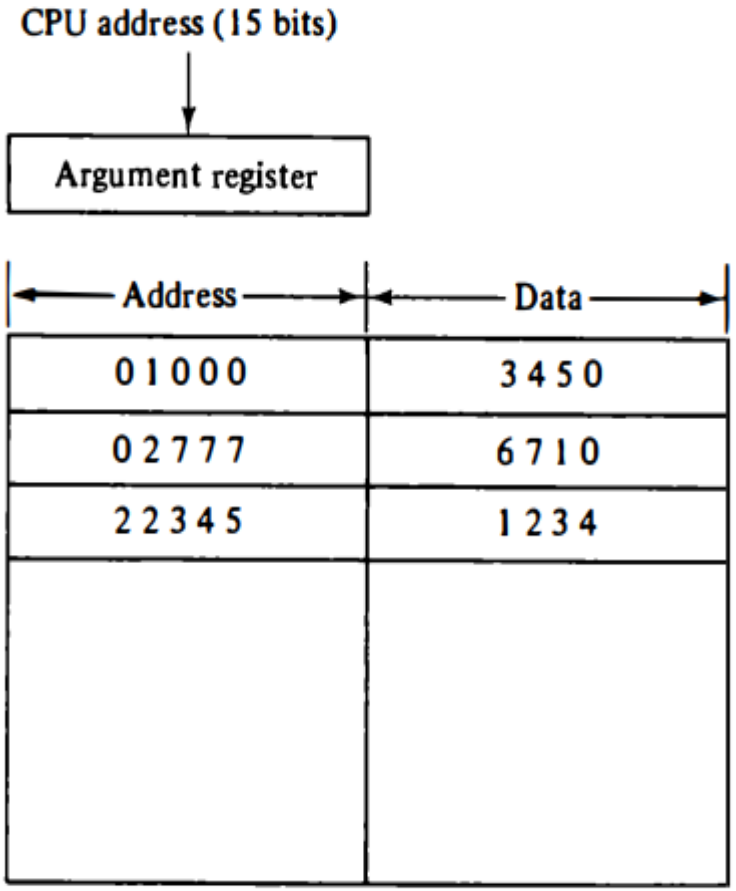
Figure Example of cache memory.

Associative Mapping

- Fastest and most flexible cache organization uses associative memory.
- It stores both address and content of memory word.
- Address is placed in argument register and memory is searched for matching address.
- If address is found corresponding data is read.
- If address is not found, it is read from main memory and transferred to cache.

Associative Mapping

Figure 11 Associative mapping cache (all numbers in octal).



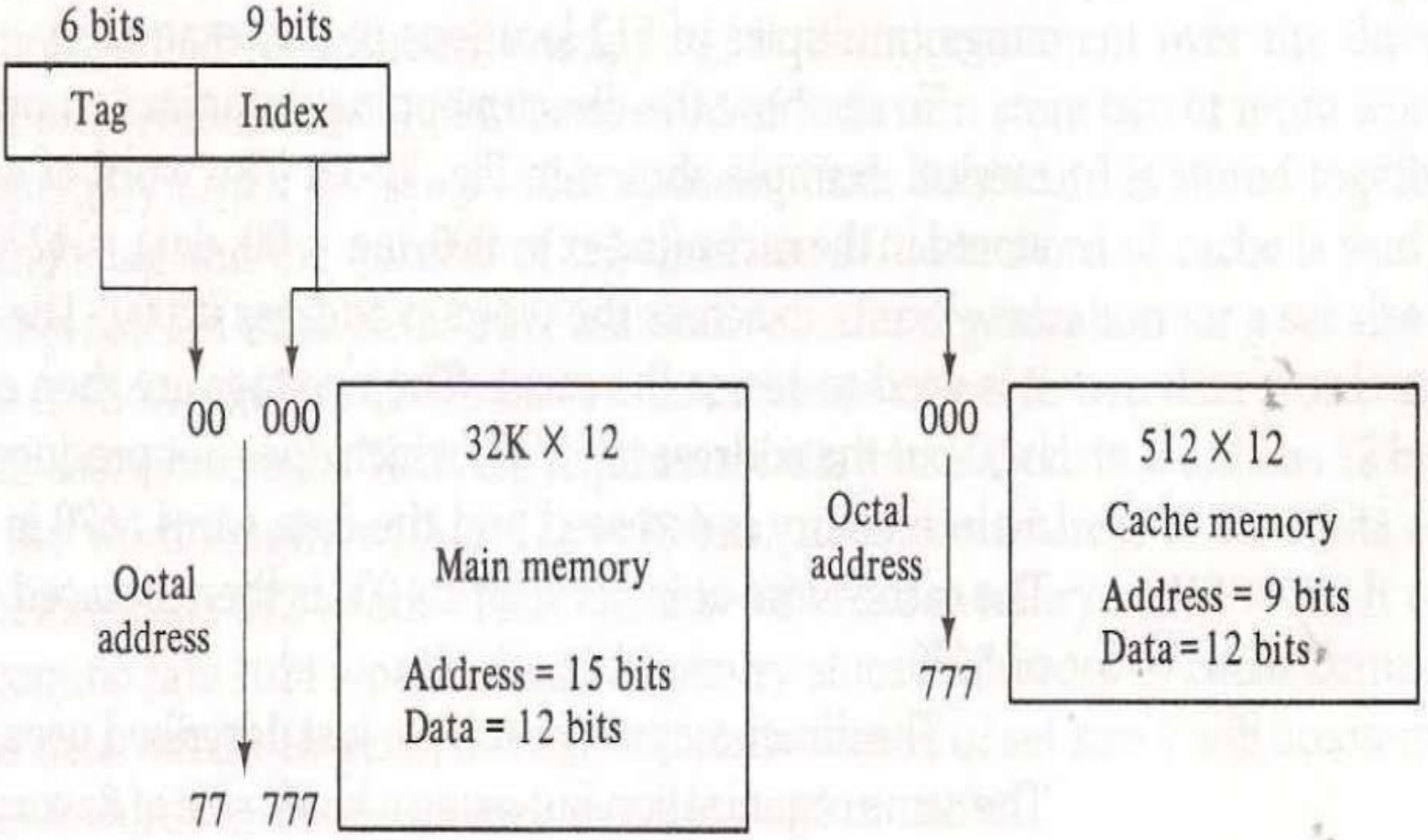
Associative Mapping

- If the cache is full, an address- word pair must be displaced.
- Various algorithm are used to determine which pair to displace. Some of them are FIFO(First In First Out), LRU(Least Recently Used) etc.

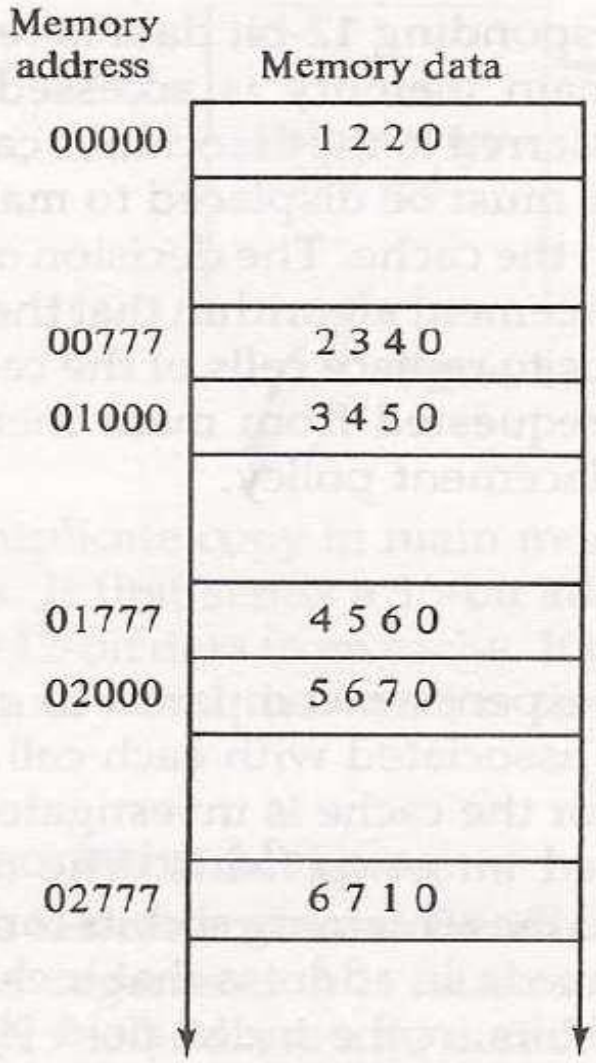
Direct Memory

- CPU address is divided into two fields tag and index.
- Index field is required to access cache memory and total address is used to access main memory.
- If there are 2^k words in cache and 2^n words in main memory, then n bit memory address is divided into two parts. k bits for index field and n-k bits for tag field.

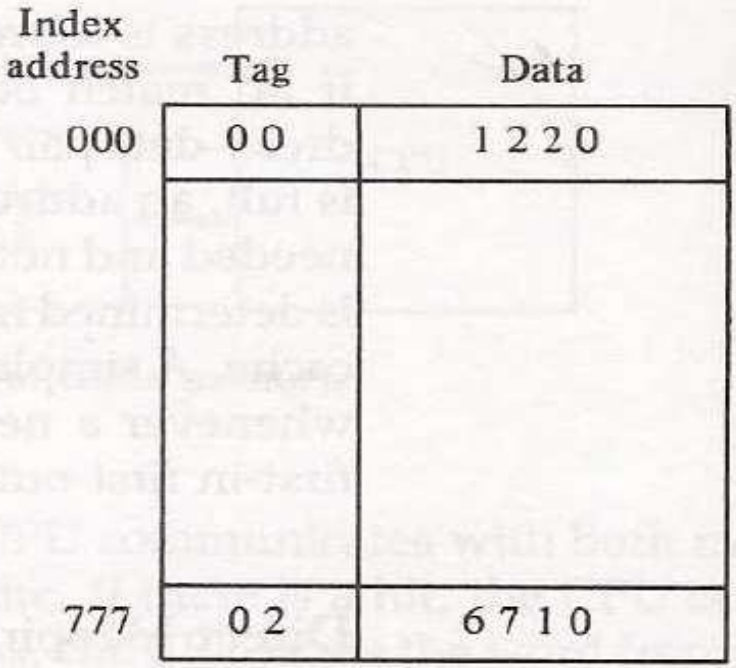
Figure Addressing relationships between main and cache memories.



Direct Mapping Cache Organization



(a) Main memory



(b) Cache memory

Direct Mapping Cache Organization

- When CPU generates memory request, index field is used to access the cache.
- Tag field of the CPU address is compared with the tag in the word read. If the tag match, there is hit.
- If the tag does not match, word is read from main memory and updated in cache.
- This example use the block size of 1.
- The same organization can be implemented for block size 8.

Direct Mapping Cache Organization

- The index field is divided into two parts: block field and word field.
- In 512 word cache there are 64 blocks of 8 words each(64*8=512).
- Block is specified with 6 bit field and word within block with 3 bit field.
- Every time miss occur, entire block of 8 word is transferred from main memory to cache.

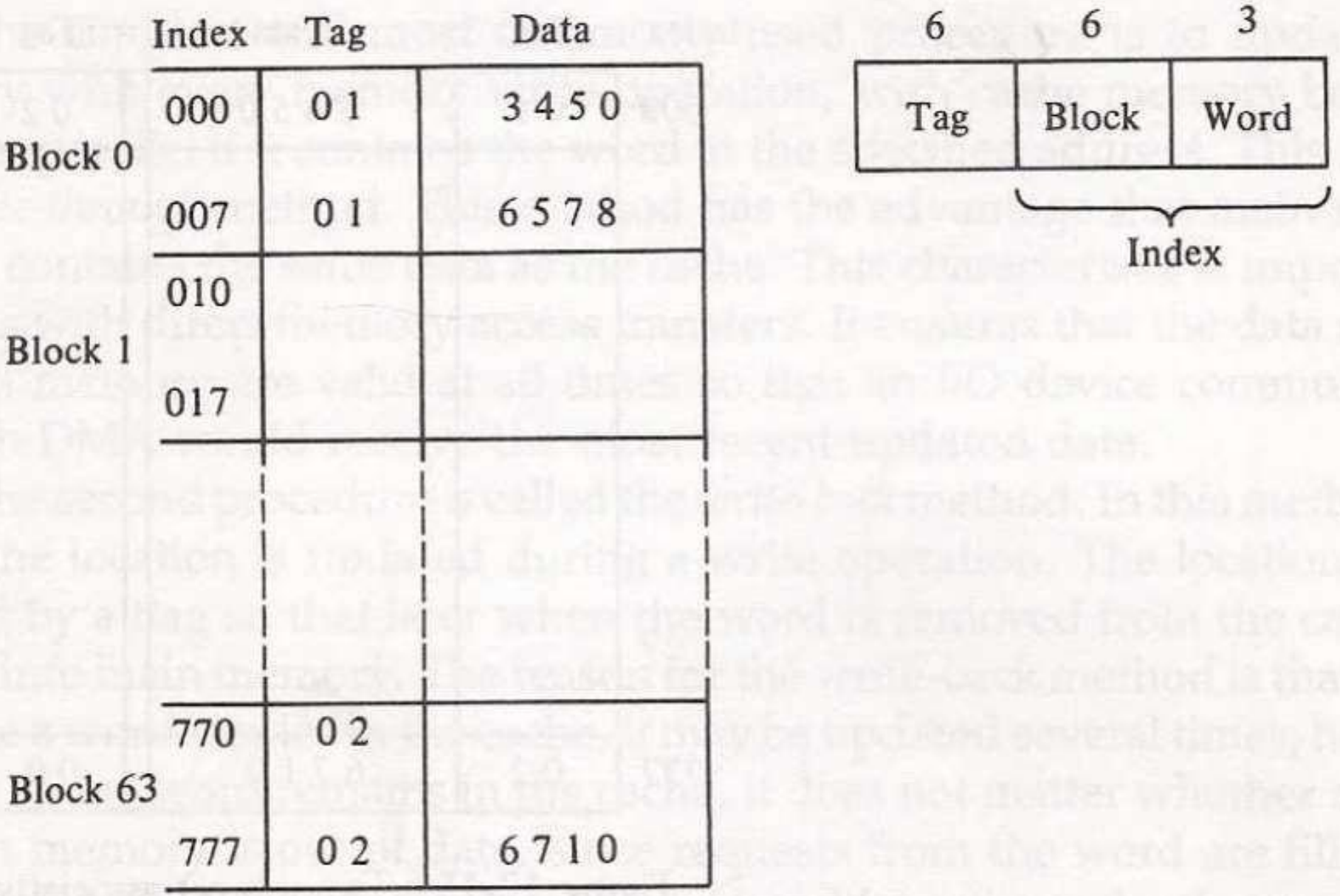


Figure Direct mapping cache with block size of 8 words.

Set-Associative Mapping

- In direct mapping two words with same index in their address but different tag values can't reside simultaneously in memory.
- In this mapping, each data word is stored together with its tag and number of tag-data items in one word of the cache is said to form set.
- In general, a set associative cache of set size k will accommodate k words of main memory in each word of cache.

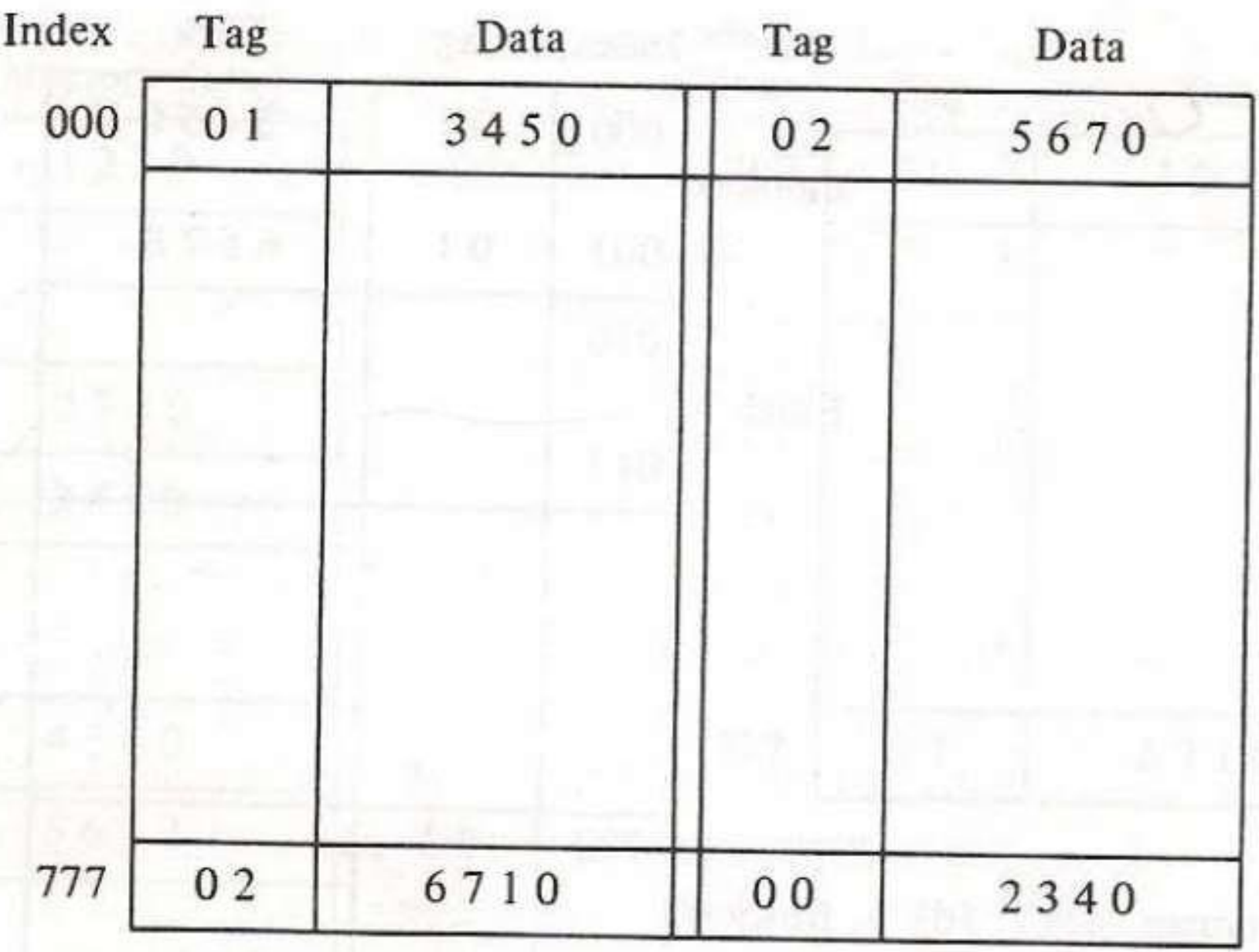


Figure Two-way set-associative mapping cache.

Set-Associative Mapping

- When a miss occur and the set is full, one of the tag data item is replaced with new value using various algorithm.

Set- Associative Mapping

- If the cache is full, an address- word pair must be displaced.
- Various algorithm are used to determine which pair to displace. Some of them are FIFO(First In First Out), LRU(Least Recently Used) etc.

Virtual Memory

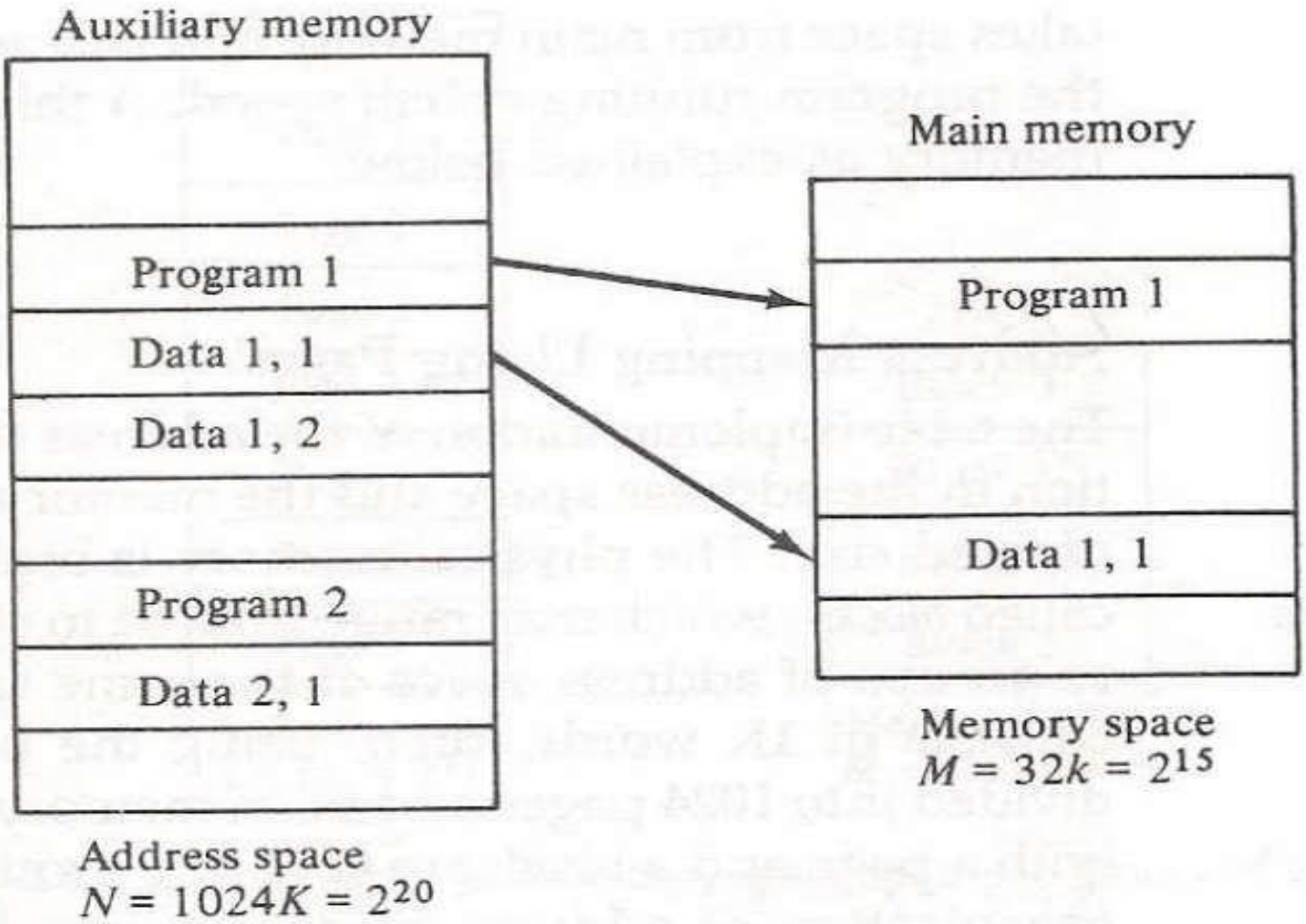


Figure Relation between address and memory space in a virtual memory system.

Virtual Memory

- Virtual memory is a concept used in computer that permits the user to construct a program as though large memory space is available equal to auxiliary memory.
- It gives the illusion that computer has large memory even though computer has relatively small main memory.
- It has mechanism that convert generated address into correct main memory address.

Address Space and Memory Space

- An address used by the programmer is called virtual address and set of such addresses is called address space.
- An address in main memory is called physical address. The set of such locations is called memory space.

Virtual Memory

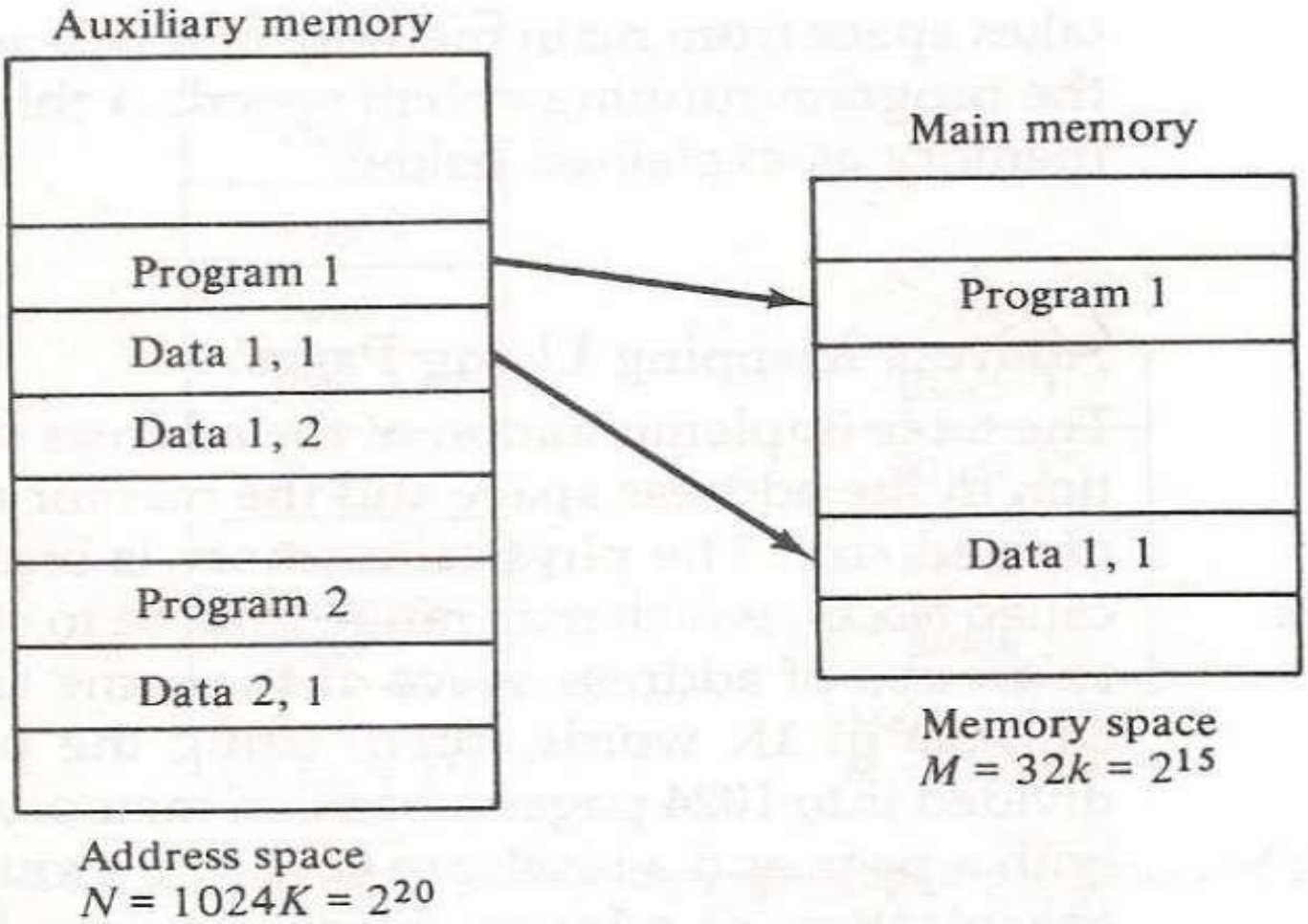
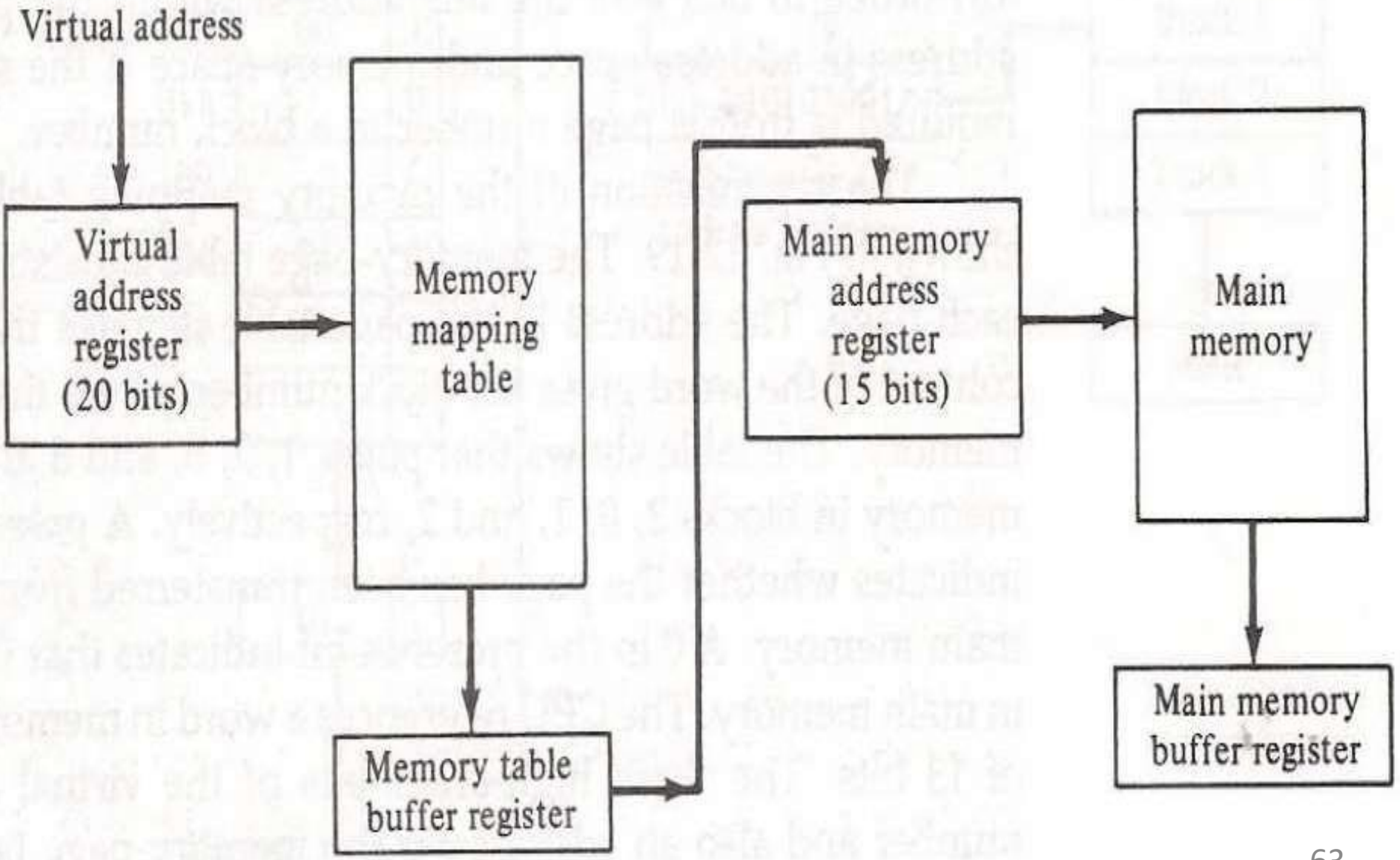


Figure Relation between address and memory space in a virtual memory system.

Figure Memory table for mapping a virtual address.



Address Mapping Using Pages

- The main memory is broken down into groups of equal size called blocks.
- Term pages refers to groups of address space of same size.
- Although page and block are of equal size, page refer to organization of address space and block represent the organization of memory space.
- The term page frame is sometimes used to denote block.

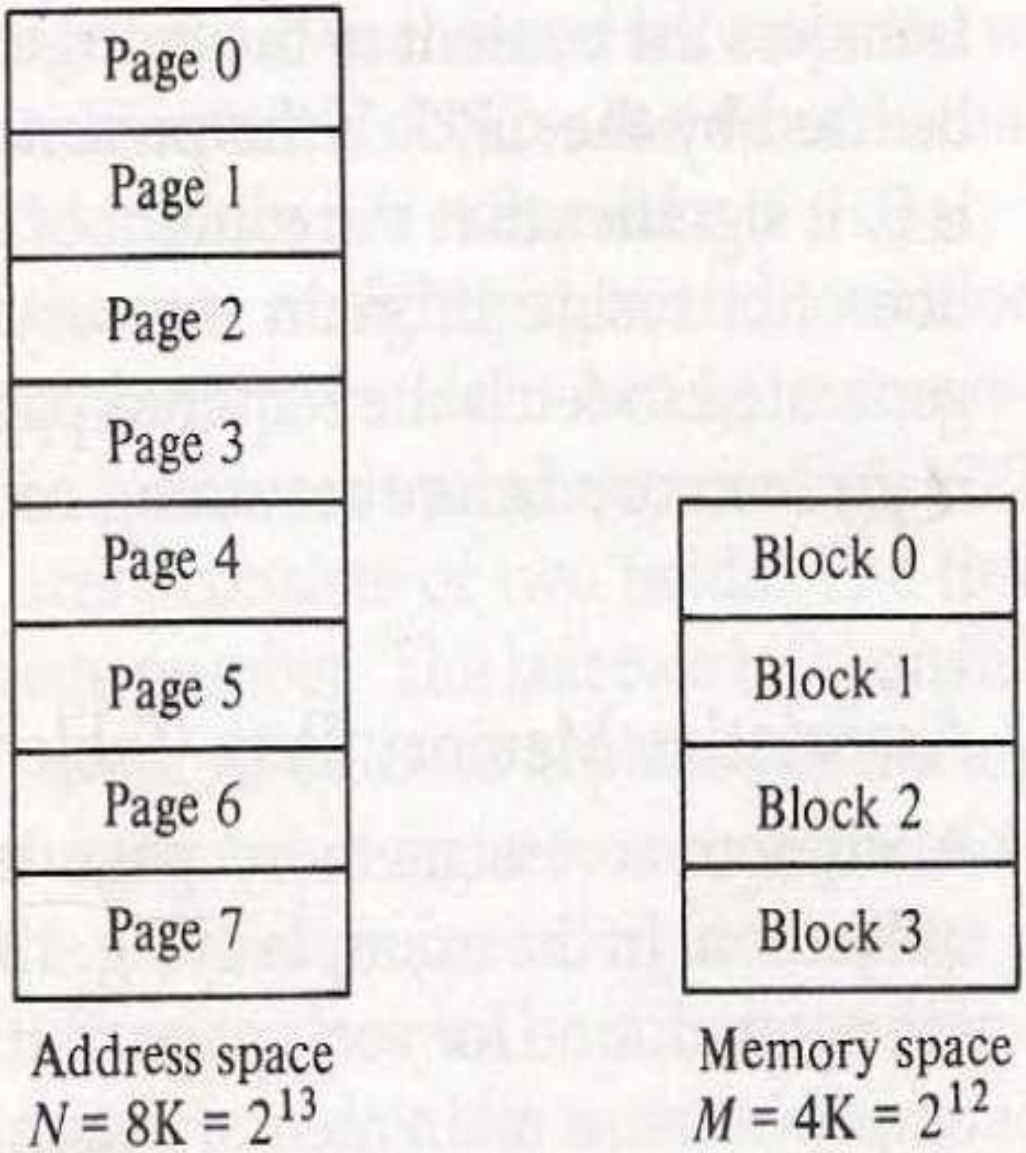


Figure Address space and memory space split into groups of 1K words.

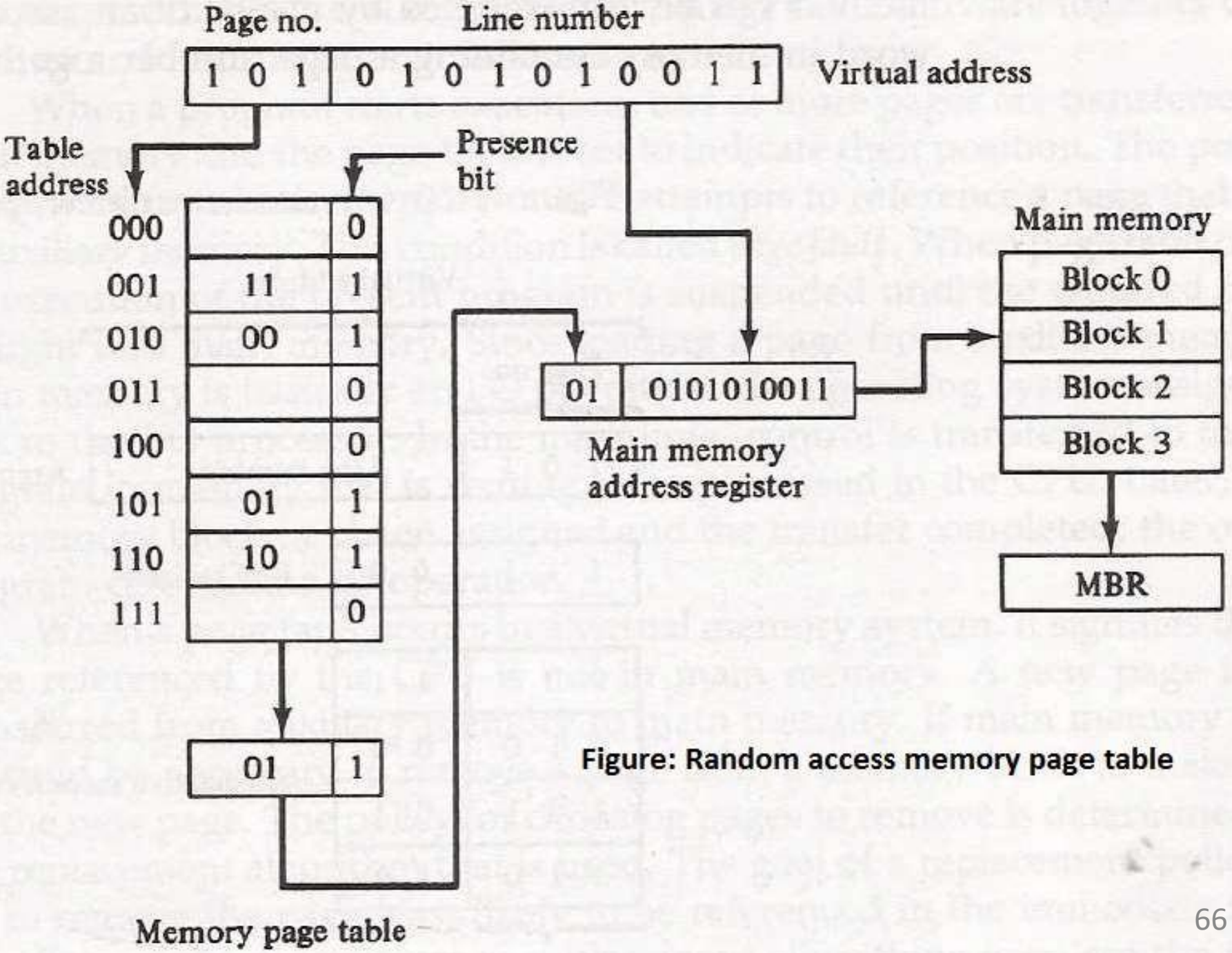
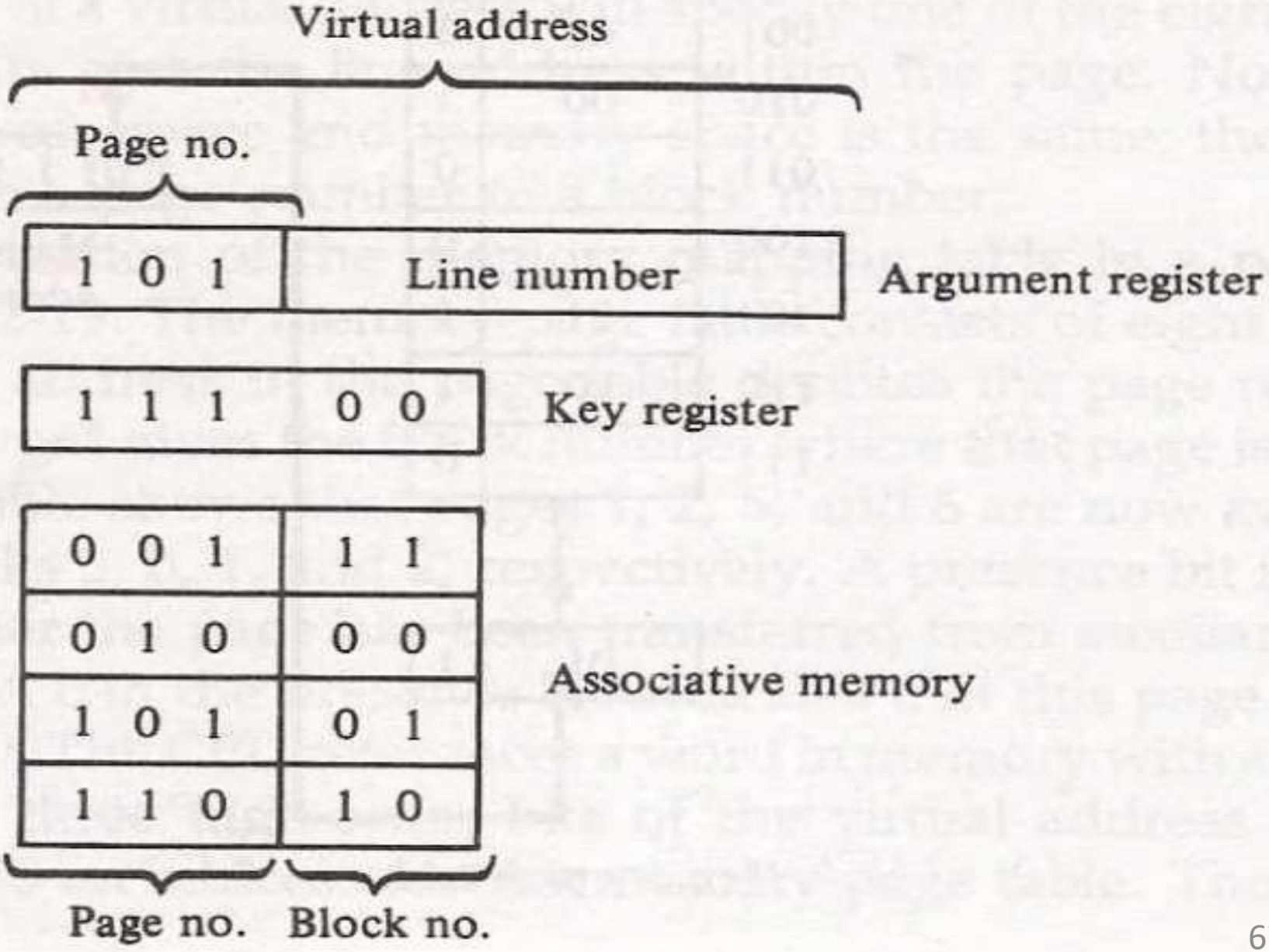


Figure: Random access memory page table

Figure An associative memory page table.



Page Replacement

- The program is executed from main memory until page required is not available.
- If page is not available, this condition is called page fault. When it occurs, present program is suspended until the page required is brought into main memory.
- If main memory is full, pages to remove is determined from the replacement algorithm used.

Writing into Cache

- Writing into cache can be done in two ways:
  - Write through
  - Write Back
- In write through, whenever write operation is performed in cache memory, main memory is also updated in parallel with the cache.
- In write back, only cache is updated and marked by the flag. When the word is removed from cache, flag is checked if it is set the corresponding address in main memory is updated.

Cache Initialization

- When power is turned on, cache contain invalid data indicated by valid bit value 0.
- Valid bit of word is set whenever the word is read from main memory and updated in cache.
- If valid bit is 0, new word automatically replace the invalid data.

Background

Virtual memory – separation of user logical memory from physical memory.

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation

Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

Memory Management

Introduction

Memory refers to storage needed by the kernel, the other components of the operating system and the user programs. In a multi-processing, multi-user system, the structure of the memory is quite complex. Efficient memory management is very critical for good performance of the entire system. In this discussion we will study memory management policies, techniques and their implementations.

Memory management requirements

Relocation: Branch addresses and data references within a program memory space (user address space) have to be translated into references in the memory range a program is loaded into.

Protection: Each process should be protected against unwanted (unauthorized) interference by other processes, whether accidental or intentional. Fortunately, mechanisms that support relocation also form the base for satisfying protection requirements.

Memory management requirements (contd.)

Sharing : Allow several processes to access the same portion of main memory : very common in many applications. Ex. many server-threads executing the same service routine.

Logical organization : allow separate compilation and run-time resolution of references. To provide different access privileges (RWX). To allow sharing. Ex: segmentation.

…requirements(contd.)

Physical organization: Memory hierarchy or level of memory. Organization of each of these levels and movement and address translation among the various levels.

Overhead : should be low. System should be spending not much time compared execution time, on the memory management techniques.

Memory management techniques

Fixed partitioning: Main memory statically divided into fixed-sized partitions: could be equal-sized or unequal-sized. Simple to implement. Inefficient use of memory and results in internal-fragmentation.

Dynamic partitioning : Partitions are dynamically created. Compaction needed to counter external fragmentation. Inefficient use of processor.

Simple paging: Both main memory and process space are divided into number of equal-sized frames. A process may in non-contiguous main memory pages.

Memory management techniques

Simple segmentation : To accommodate dynamically growing partitions: Compiler tables, for example. No fragmentation, but needs compaction.

Virtual memory with paging: Same as simple paging but the pages currently needed are in the main memory. Known as demand paging.

Virtual memory with segmentation: Same as simple segmentation but only those segments needed are in the main memory.

Segmented-paged virtual memory

Basic memory operations: Relocation

A process in the memory includes instructions plus data. Instruction contain memory references: Addresses of data items, addresses of instructions.
These are *logical addresses*: relative addresses are examples of this. These are addresses which are expressed with reference to some known point, usually the beginning of the program.
Physical addresses are absolute addresses in the memory.
Relative addressing or position independence helps easy relocation of programs.

Basic memory operations: Linking and loading

The function of a linker is to take as input a collection of object modules and produce a load module that consists of an integrated set of program and data modules to be passed to the loader. It also resolves all the external symbolic references in the load module (linkage).

The nature of the address linkage will depend on the types of load module created and the time of linkage: static, load-time dynamic linking, run-time dynamic linking.

Dynamic linking: Deferring the linkage of external references until load-module is created: load module contains unresolved references to other programs.

Virtual memory

Consider a typical, large application:

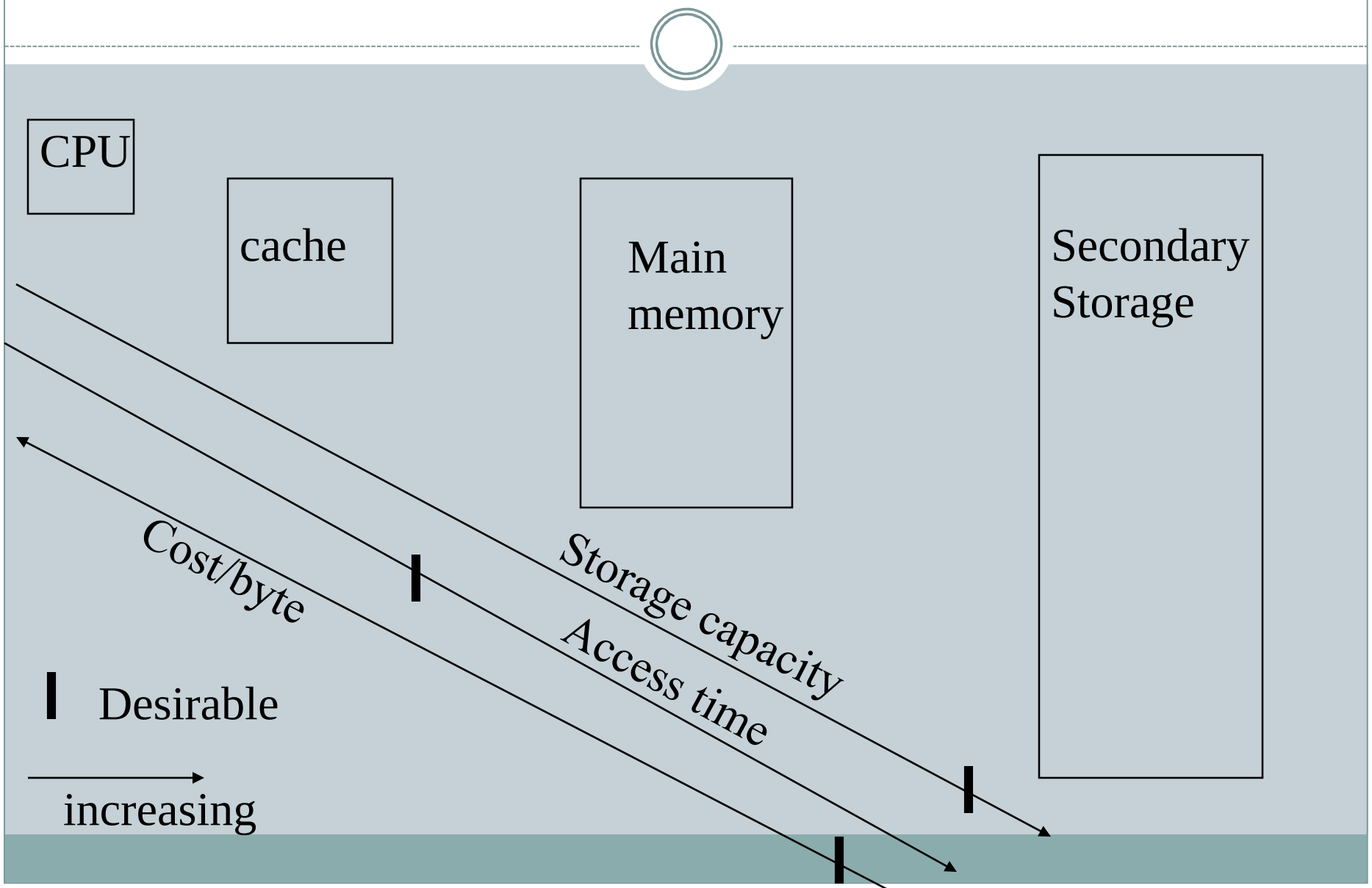
- There are many components that are mutually exclusive. Example: A unique function selected dependent on user choice.
- Error routines and exception handlers are very rarely used.
- Most programs exhibit a slowly changing **locality of reference**.
- There are two types of locality: **spatial and temporal**.

Locality

Temporal locality: Addresses that are referenced at some time Ts will be accessed in the near future (Ts + delta_time) with high probability. Example : Execution in a loop.

Spatial locality: Items whose addresses are near one another tend to be referenced close together in time. Example: Accessing array elements.
How can we exploit this characteristics of programs? Keep only the current locality in the main memory. Need not keep the entire program in the main memory.

Space and Time



Demand paging

Main memory (physical address space) as well as user address space (virtual address space) are logically partitioned into equal chunks known as pages. Main memory pages (sometimes known as frames) and virtual memory pages are of the same size.

Virtual address (VA) is viewed as a pair (virtual page number, offset within the page). Example: Consider a virtual space of 16K , with 2K page size and an address 3045. What the virtual page number and offset corresponding to this VA?

Virtual Page Number and Offset

3045 / 2048 = 1

3045 % 2048 = 3045 - 2048 = 997

VP# = 1

Offset within page = 1007

Page Size is always a power of 2? Why?

Page Size Criteria

Consider the binary value of address 3045 :

1011 1110 0101

for 16K address space the address will be 14 bits.

Rewrite:

00 1011 1110 0101

A 2K address space will have offset range 0 -2047 (11 bits)

00 1	011 1110 0101

Page# Offset within page

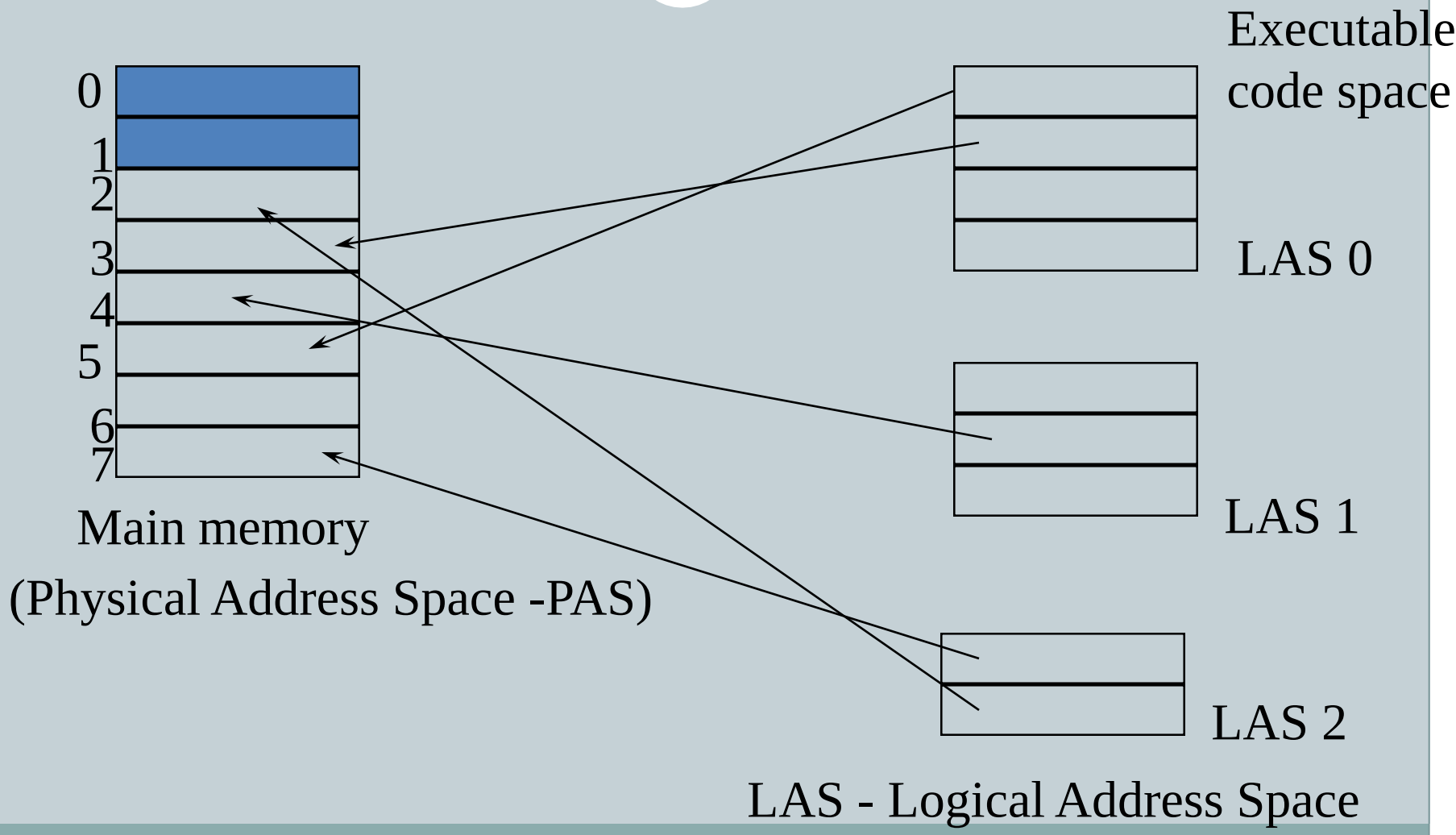
Demand paging (contd.)

There is only one physical address space but as many virtual address spaces as the number of processes in the system. At any time physical memory may contain pages from many process address space.

Pages are brought into the main memory when needed and “rolled out” depending on a page replacement policy.

Consider a 8K main (physical) memory and three virtual address spaces of 2K, 3K and 4K each. Page size of 1K. The status of the memory mapping at some time is as shown.

Demand Paging (contd.)



Issues in demand paging

How to keep track of which logical page goes where in the main memory? More specifically, what are the data structures needed?

- Page table, one per logical address space.

How to translate logical address into physical address and when?

- Address translation algorithm applied every time a memory reference is needed.

How to avoid repeated translations?

- After all most programs exhibit good locality. “cache recent translations”

Issues in demand paging (contd.)

What if main memory is full and your process demands a new page? What is the policy for page replacement? LRU, MRU, FIFO, random?

Do we need to roll out every page that goes into main memory? No, only the ones that are modified. How to keep track of this info and such other memory management information? In the page table as special bits.

Page table

One page table per logical address space.
There is one entry per logical page. Logical page number is used as the index to access the corresponding page table entry.
Page table entry format:
Present bit, Modify bit, Other control bits, Physical page number

Address translation

Goal: To translate a logical address LA to physical address PA.

1. LA = (Logical Page Number, Offset within page)
   Logical Page number LPN = LA DIV pagesize
   Offset = LA MOD pagesize
2. If Pagetable(LPN).Present step 3
   else **PageFault** to Operating system.
3. Obtain Physical Page Number (PPN)
   PPN = Pagetable(LPN).Physical page number.
4. Compute Physical address:
   PA = PPN *Pagesize + Offset.

Example

Exercise 8.1: Page size : 1024 bytes.

Page table

Virtual_page#	Valid bit	Page_frame#
0	1	4
1	1	7
2	0	-
3	1	2
4	0	-
5	1	0

PA needed for 1052, 2221, 5499

Page fault handler

When the requested page is not in the main memory a page fault occurs.
This is an interrupt to the OS.
Page fault handler:
1. If there is empty page in the main memory , roll in the required logical page, update page table. Return to address translation step #3.
2. Else, apply a replacement policy to choose a main memory page to roll out. Roll out the page, if modified, else overwrite the page with new page. Update page table, return to address translation step #3.

Replacement policies

FIFO: first-in first-out.
LRU: Least Recently used.
NRU: Not recently used.
Clock-based.
Belady's optimal min. (theoretical).

Translation look-aside buffer

A special cache for page table (translation) entries.
Cache functions the same way as main memory cache. Contains those entries that have been recently accessed.

When an address translation is needed lookup TLB.
If there is a miss then do the complete translation, update TLB, and use the translated address.
If there is a hit in TLB, then use the readily available translation. No need to spend time on translation.

Resident Set Management

Usually an allocation policy gives a process certain number of main memory pages within which to execute.
The number of pages allocated is also known as the resident set (of pages).
Two policies for resident set allocation: fixed and variable.
When a new process is loaded into the memory, allocate a certain number of page frames on the basis of application type, or other criteria. Prepaging or demand paging is used to fill up the pages.
When a page fault occurs select a page for replacement.

Resident Set Management (contd.)

Replacement Scope: In selecting a page to replace,

- a **local replacement policy** chooses among only the resident pages of the process that generated the page fault.
- a global replacement policy considers all pages in the main memory to be candidates for replacement.

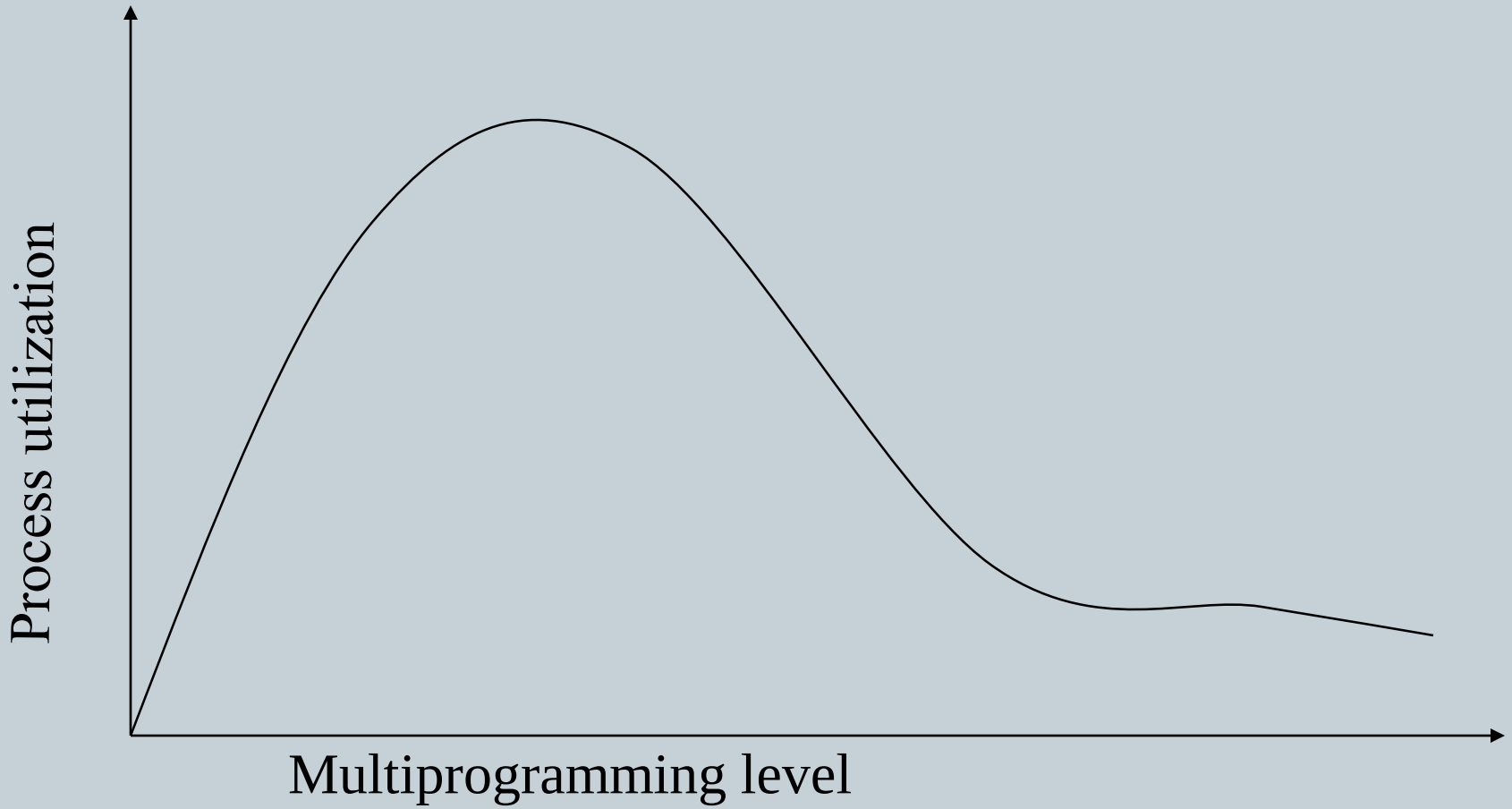
In case of variable allocation, from time to time evaluate the allocation provided to a process, increase or decrease to improve overall performance.

Load control

Multiprogramming level is determined by the number of processes resident in main memory. Load control policy is critical in effective memory management.

- Too few may result in inefficient resource use,
- Too many may result in inadequate resident set size resulting in frequent faulting.
- Spending more time servicing page faults than actual processing is called “thrashing”

Load Control Graph



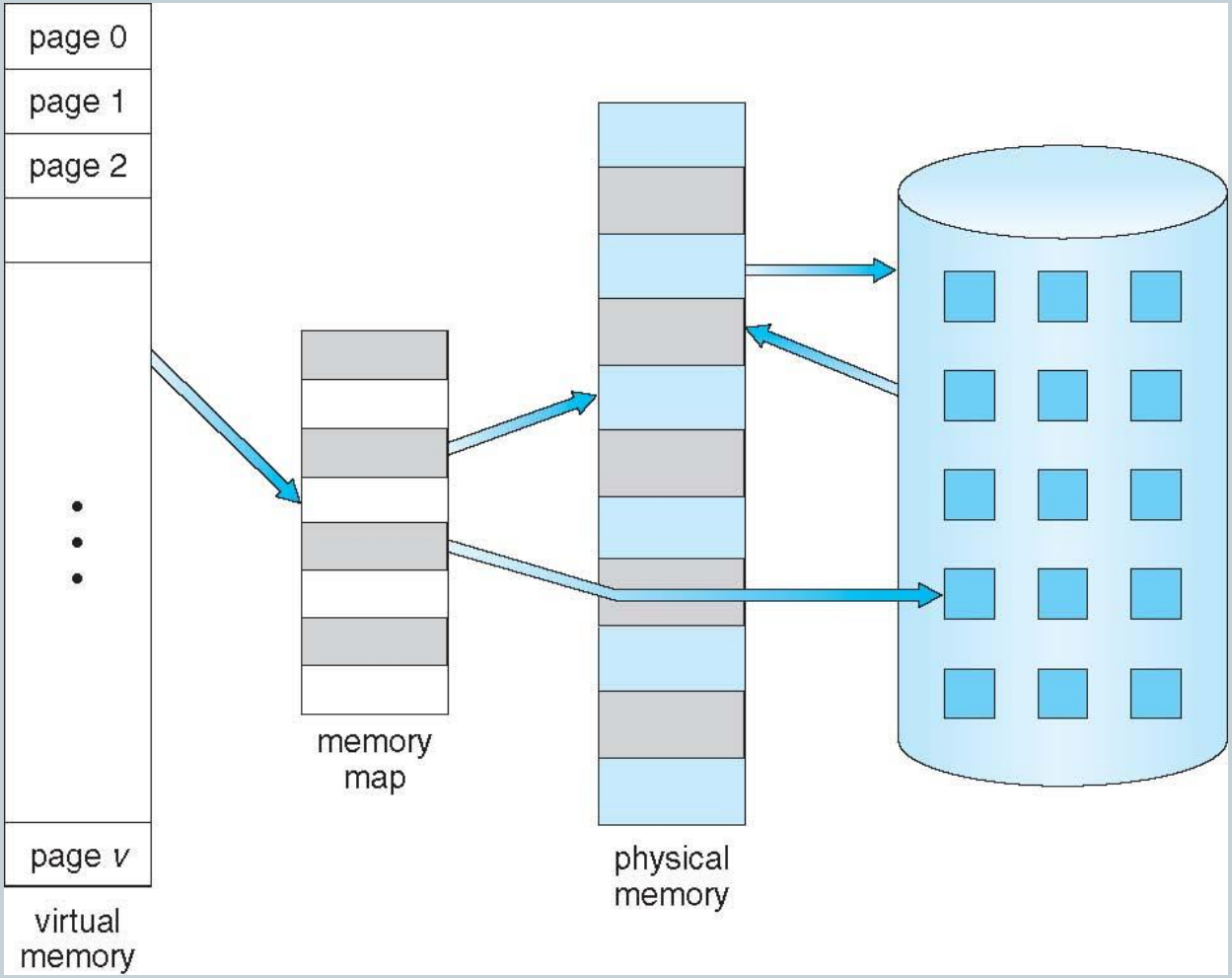
Load control (contd.)

Processor utilization increases with the level of multiprogramming up to to a certain level beyond which system starts “thrashing”.

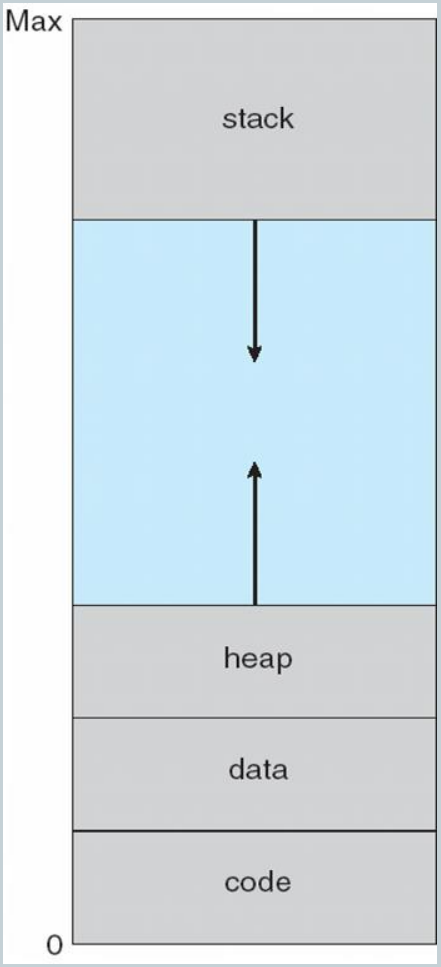
When this happens, allow only those processes whose resident set are large enough are allowed to execute.

You may need to suspend certain processes to accomplish this: You could use any of the six criteria (p.359,360) to decide the process to suspend.

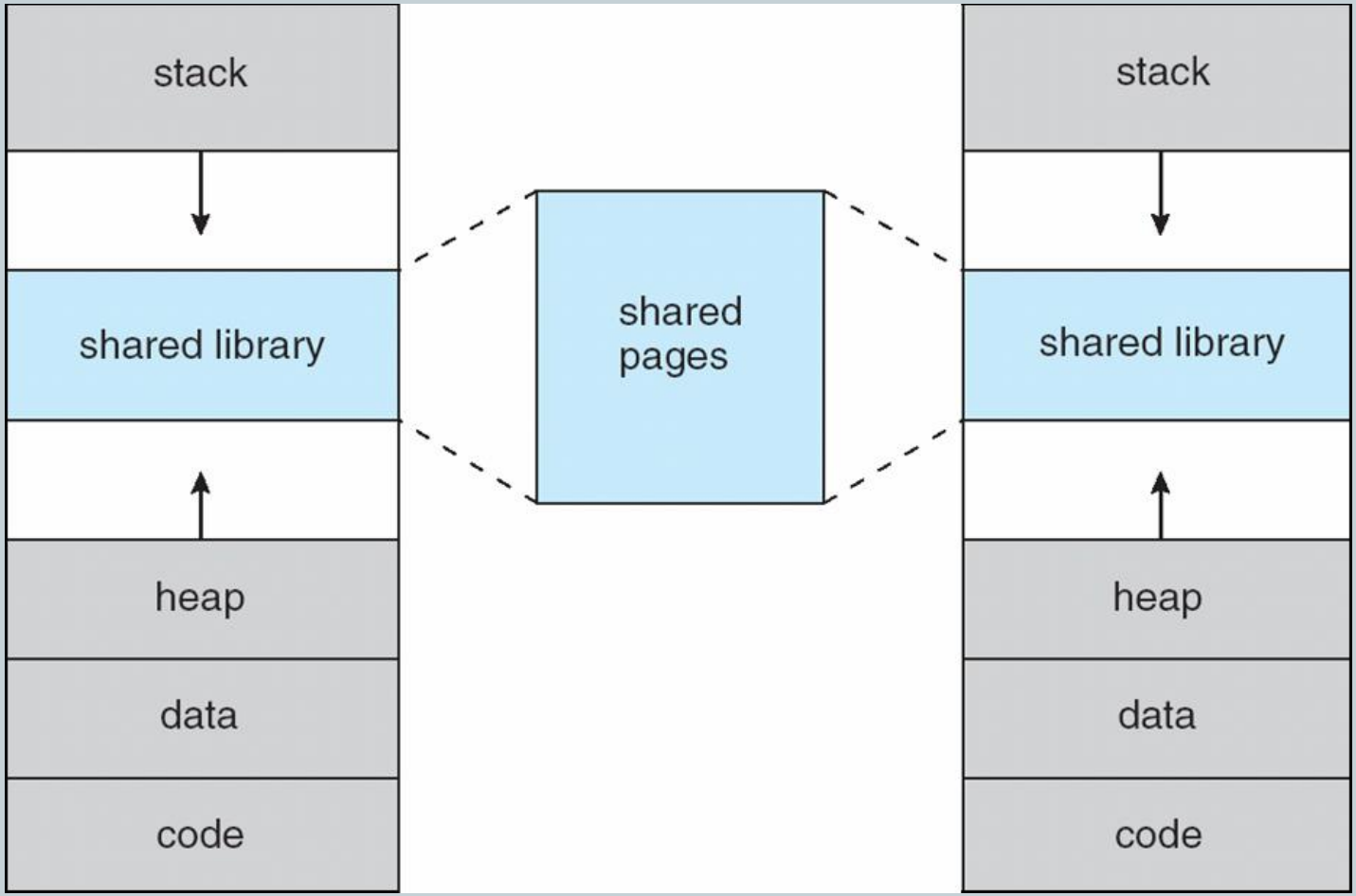
More about Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory



Demand Paging

Bring a page into memory only when it is needed

- Less I/O needed
- Less memory needed
- Faster response
- More users

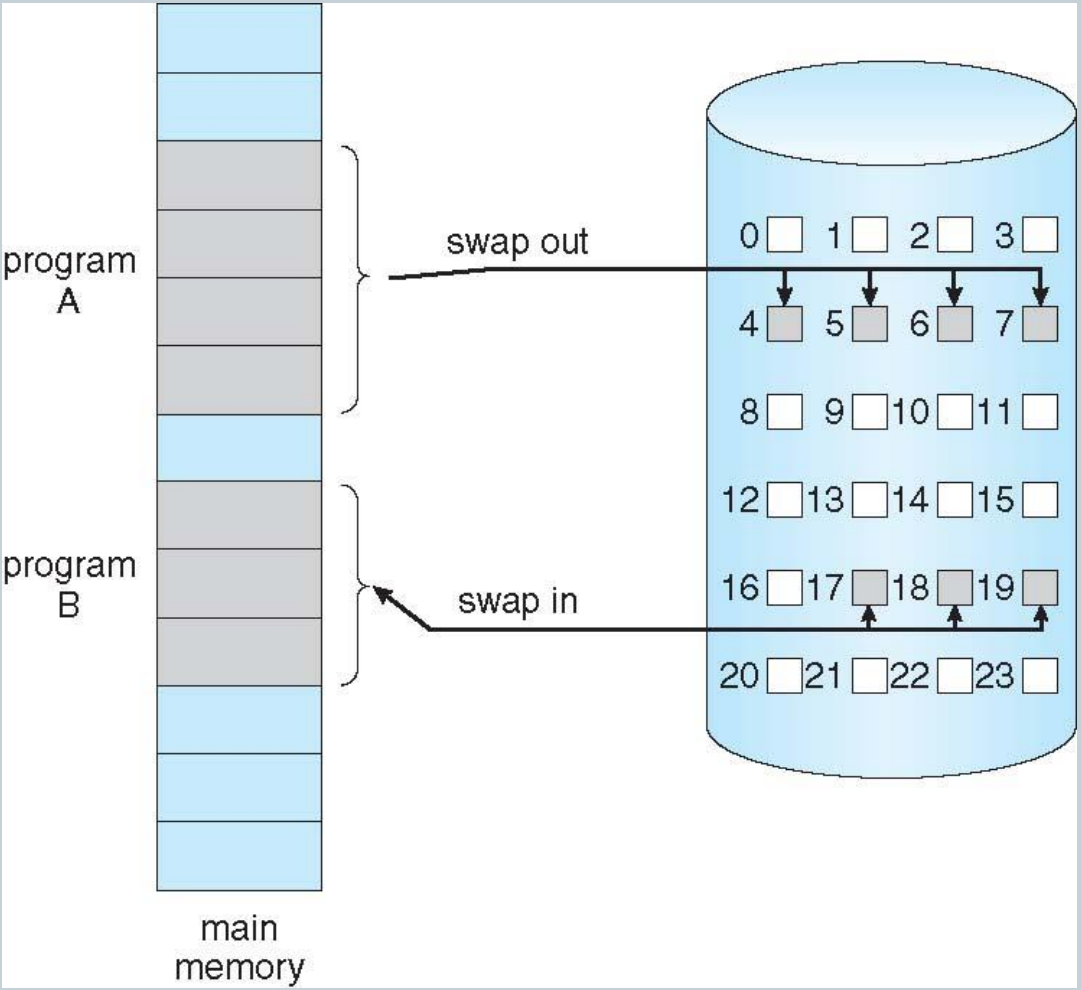
Page is needed ⇒ reference to it

- invalid reference ⇒ abort
- not-in-memory ⇒ bring to memory

Lazy swapper – never swaps a page into memory unless page will be needed

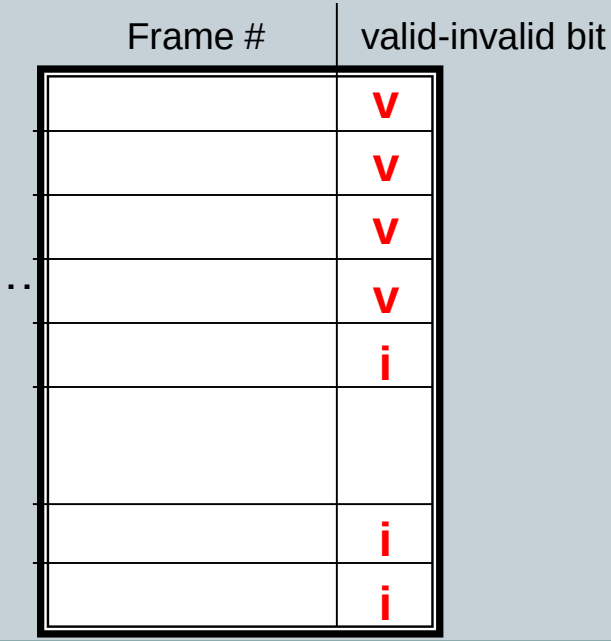
- Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space

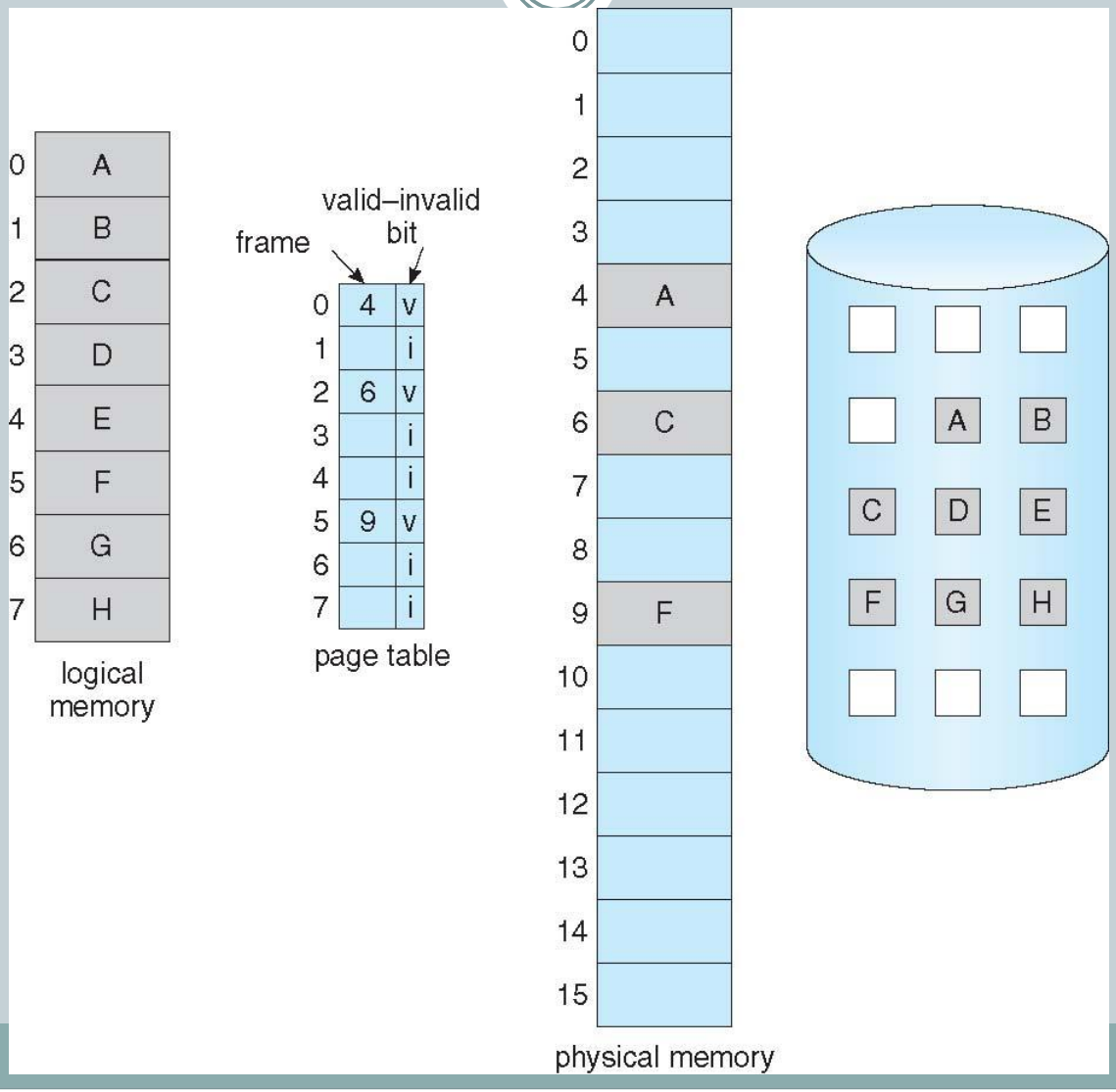


Valid-Invalid Bit

With each page table entry a valid–invalid bit is associated
(**v** ⇒ in-memory, **i** ⇒ not-in-memory)
Initially valid–invalid bit is set to **i** on all entries
Example of a page table snapshot:



Page Table When Some Pages Are Not in Main Memory



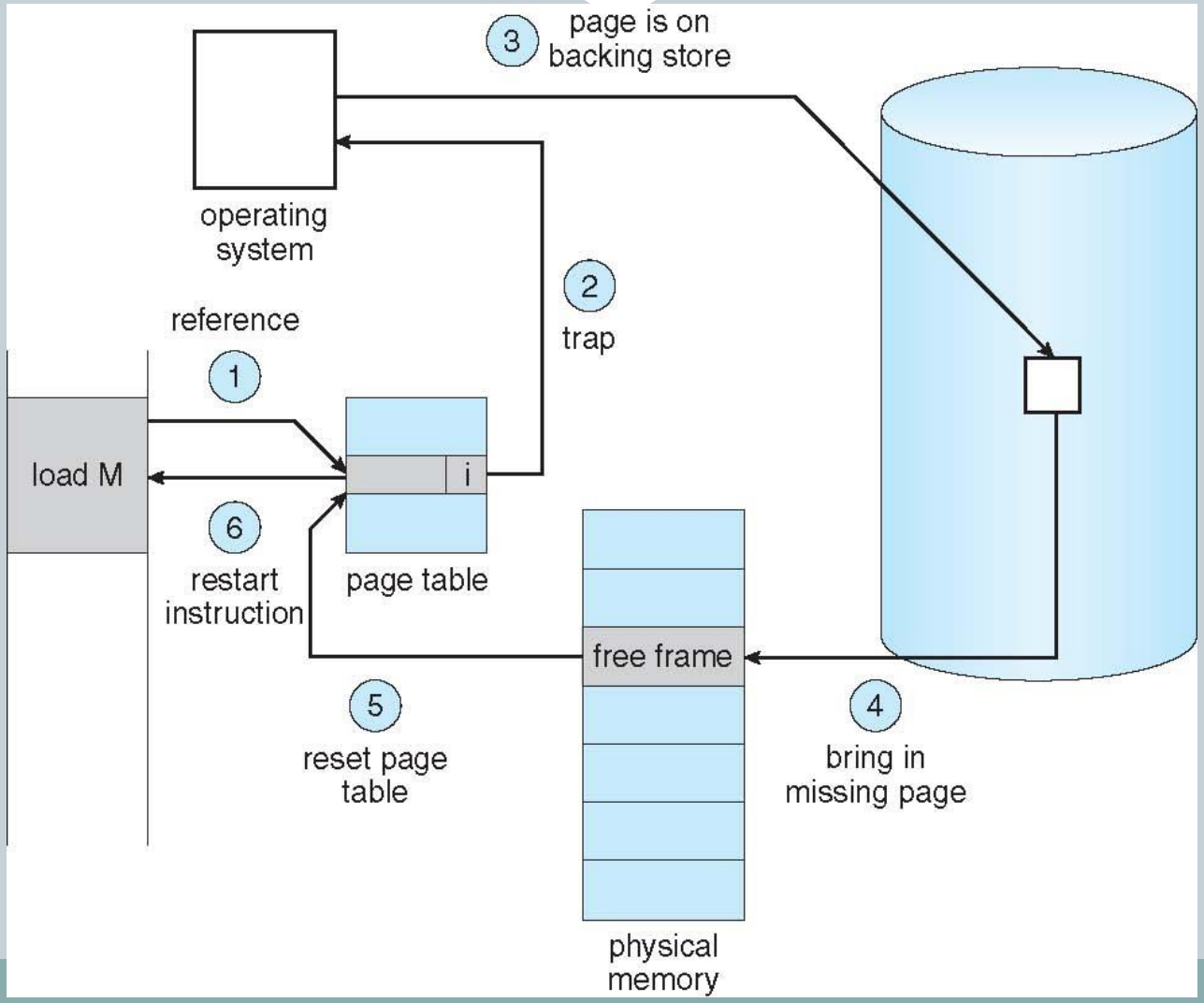
Page Fault

If there is a reference to a page, first reference to that page will trap to operating system:

page fault

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

Steps in Handling a Page Fault



Performance of Demand Paging

Page Fault Rate $0 \le p \le 1.0$

if $p = 0$ no page faults

if $p = 1$, every reference is a fault

Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = {} & (1 - p) \times \text{memory access} \\ & + p \,(\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart overhead} \\ & \,) \end{aligned}$$

Demand Paging Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

EAT = (1 – p) x 200 + p (8 milliseconds)

= 200 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800 nanosecs

If one access out of 1,000 causes a page fault, then

EAT = 200 + 0.001 x 7,999,800 = 200 + 7999.8

EAT = 8199.8 nanoseconds or about 8.2 microseconds

This is a slowdown by a factor of over 40!! (8200 / 200 = 41)

Process Creation

Virtual memory allows other benefits during process creation:

- Copy-on-Write

- Memory-Mapped Files (later)

Copy-on-Write

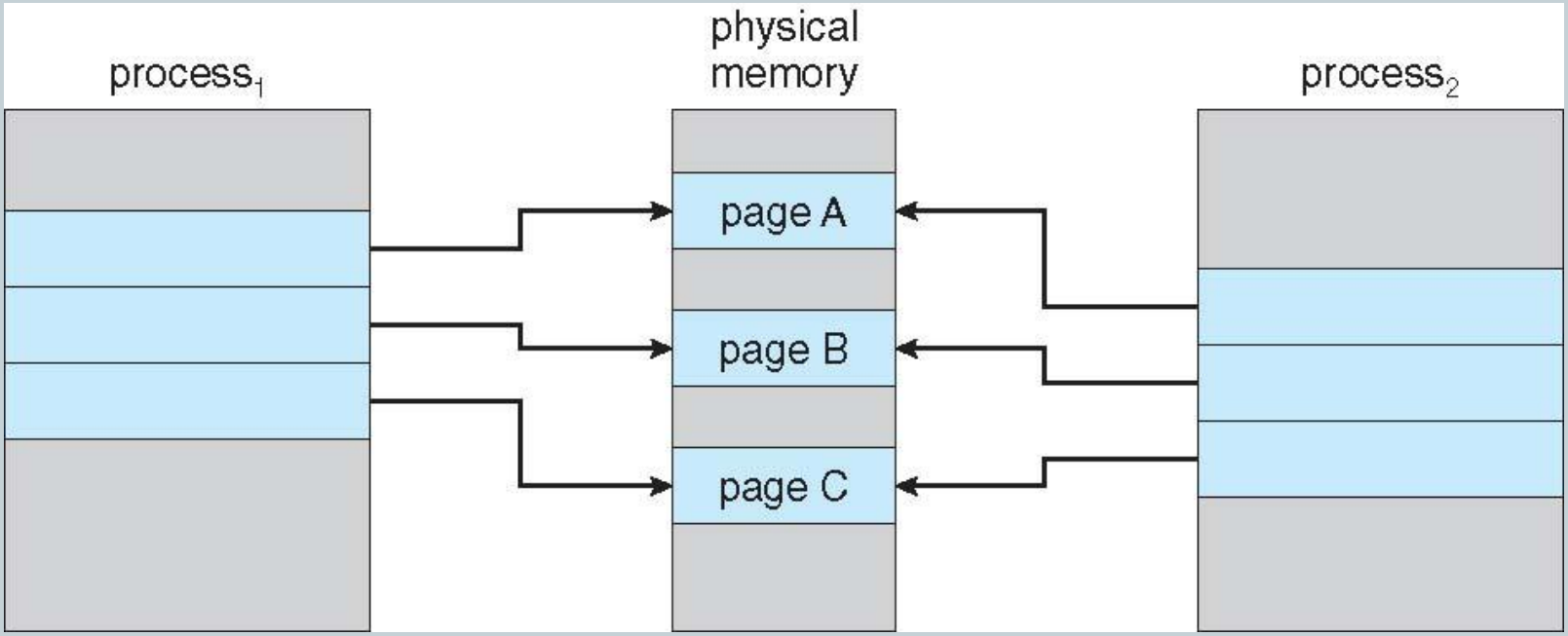
Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

If either process modifies a shared page, only then is the page copied

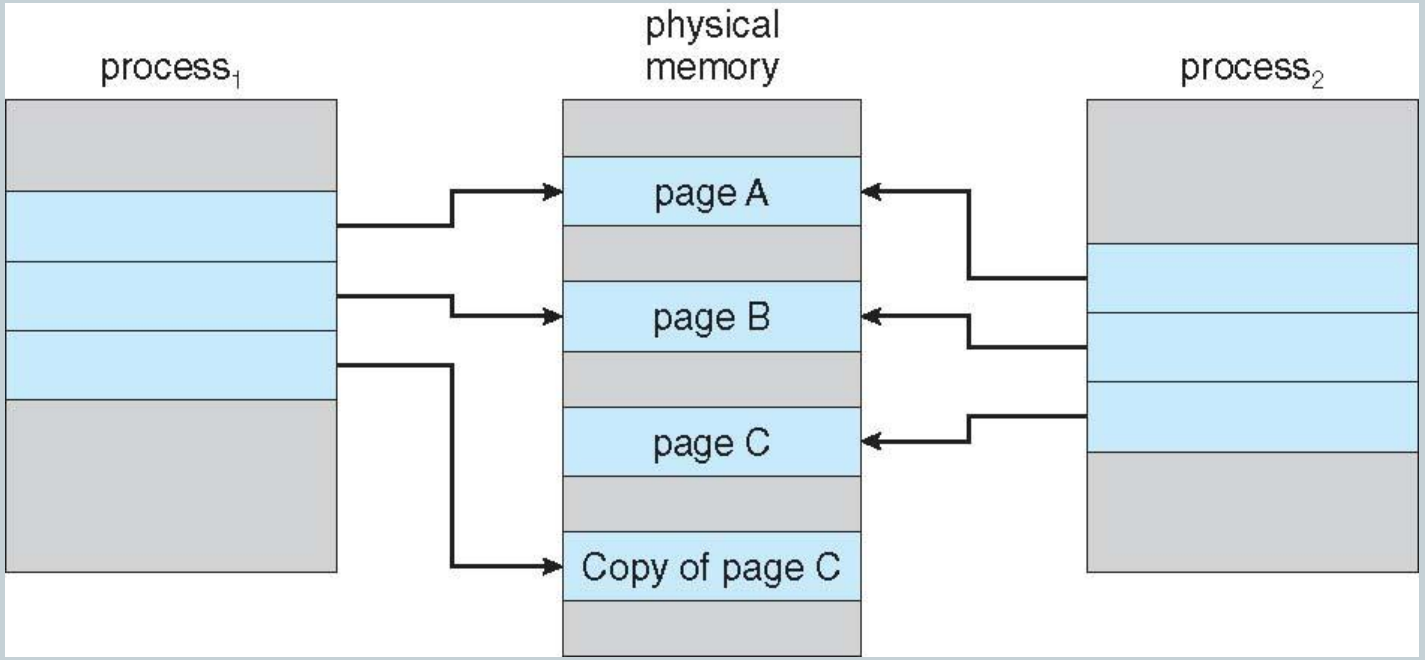
COW allows more efficient process creation as only modified pages are copied

Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What happens if there is no free frame?

Page replacement – find some page in memory, but not really in use, swap it out

- algorithm
- performance – want an algorithm which will result in minimum number of page faults

Same page may be brought into memory several times

Paging Revisited

If a page is not in physical memory

- find the page on disk
- find a free frame
- bring the page into memory

What if there is no free frame in memory?

Page Replacement

Basic idea:

- if there is a free page in memory, use it
- if not, select a *victim* frame
- write the victim out to disk
- read the desired page into the now free frame
- update page tables
- restart the process

Page Replacement

Main objective of a good replacement algorithm is to achieve a low *page fault rate*

- insure that heavily used pages stay in memory
- the replaced page should not be needed for some time

Secondary objective is to reduce latency of a page fault

- efficient code
- replace pages that do not need to be written out

Page Replacement

Reference String

Reference string is the sequence of pages being referenced

If user has the following sequence of addresses

123, 215, 600, 1234, 76, 96

If the page size is 100, then the reference string is

1, 2, 6, 12, 0, 0

First-In, First-Out (FIFO)

The oldest page in physical memory is the one selected for replacement

Very simple to implement

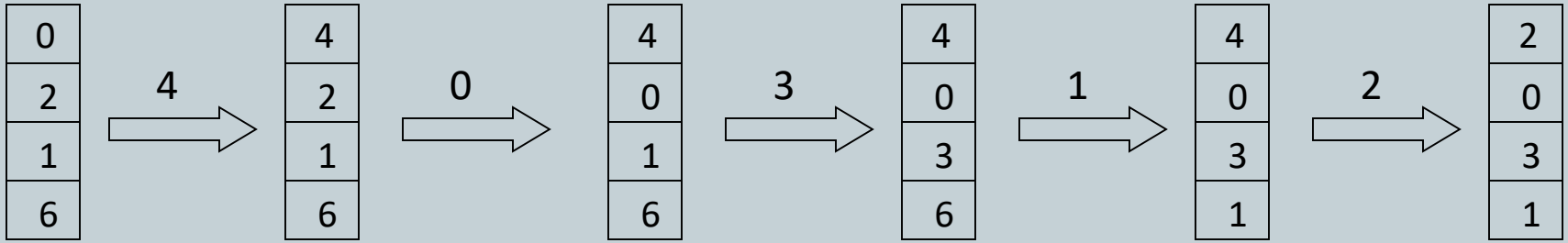
- keep a list
  - ❖ victims are chosen from the tail
  - ❖ new pages in are placed at the head

FIFO

- 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

FIFO Issues

Poor replacement policy

Evicts the oldest page in the system

- usually a heavily used variable should be around for a long time
- FIFO replaces the oldest page - perhaps the one with the heavily used variable

FIFO does not consider page usage

Optimal Page Replacement

Often called Balady's Min

Basic idea

- replace the page that will not be referenced for the longest time

This gives the lowest possible fault rate

Impossible to implement

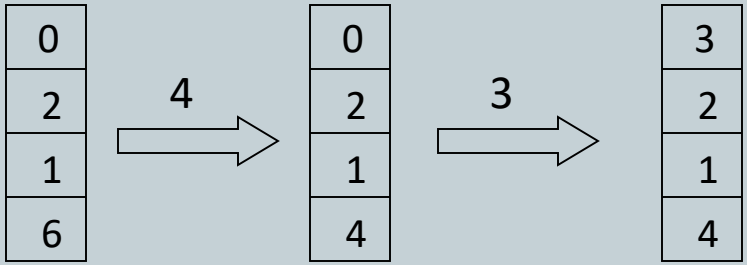
Does provide a good measure for other techniques

Optimal Page Replacement

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

x x x x x x

Compulsory Misses



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

Least Recently Used (LRU)

Basic idea

replace the page in memory that has not been accessed for the longest time

Optimal policy looking back in time

as opposed to forward in time

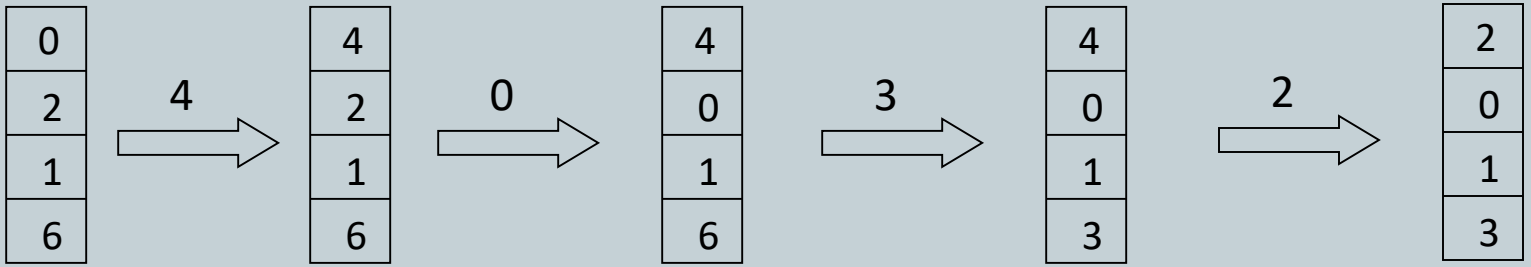
fortunately, programs tend to follow similar behavior

LRU

- 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

Compulsory Misses



- Fault Rate = 8 / 12 = 0.67

LRU Issues

How to keep track of last page access?

- requires special hardware support

2 major solutions

- counters
  - hardware clock “ticks” on every memory reference
  - the page referenced is marked with this “time”
  - the page with the smallest “time” value is replaced
- stack
  - keep a stack of references
  - on every reference to a page, move it to top of stack
  - page at bottom of stack is next one to be replaced

LRU Issues

Both techniques just listed require additional hardware

- remember, memory reference are very common
- impractical to invoke software on every memory reference

LRU is not used very often

Instead, we will try to approximate LRU

Replacement Hardware Support

Most system will simply provide a *reference bit* in PT for each page
On a reference to a page, this bit is set to 1
This bit can be cleared by the OS
This simple hardware has lead to a variety of algorithms to approximate LRU

Sampled LRU

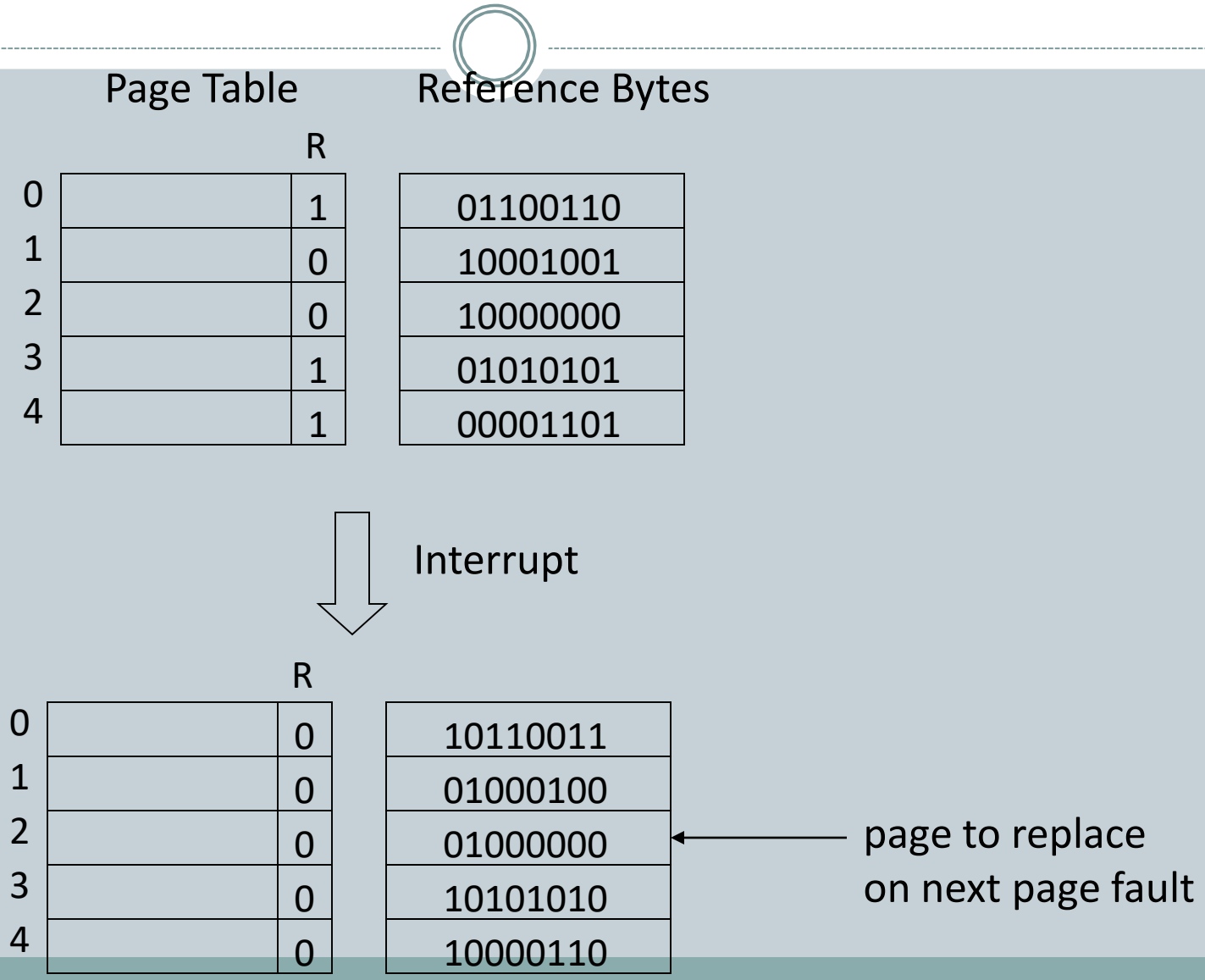
Keep a *reference byte* for each page

At set time intervals, take an interrupt and get the OS involved

- OS reads the reference bit for each page
- reference bit is *stuffed* into the beginning byte for page
- all the reference bits are then cleared

On page fault, replace the page with the smallest reference byte

Sampled LRU


Page Table
Reference Bytes
R
0
1
2
3
4
1
0
0
1
1
01100110
10001001
10000000
01010101
00001101
Interrupt
R
0
1
2
3
4
0
0
0
0
0
10110011
01000100
01000000
10101010
10000110
page to replace
on next page fault

Clock Algorithm (Second Chance)

On page fault, search through pages

If a page's reference bit is set to 1

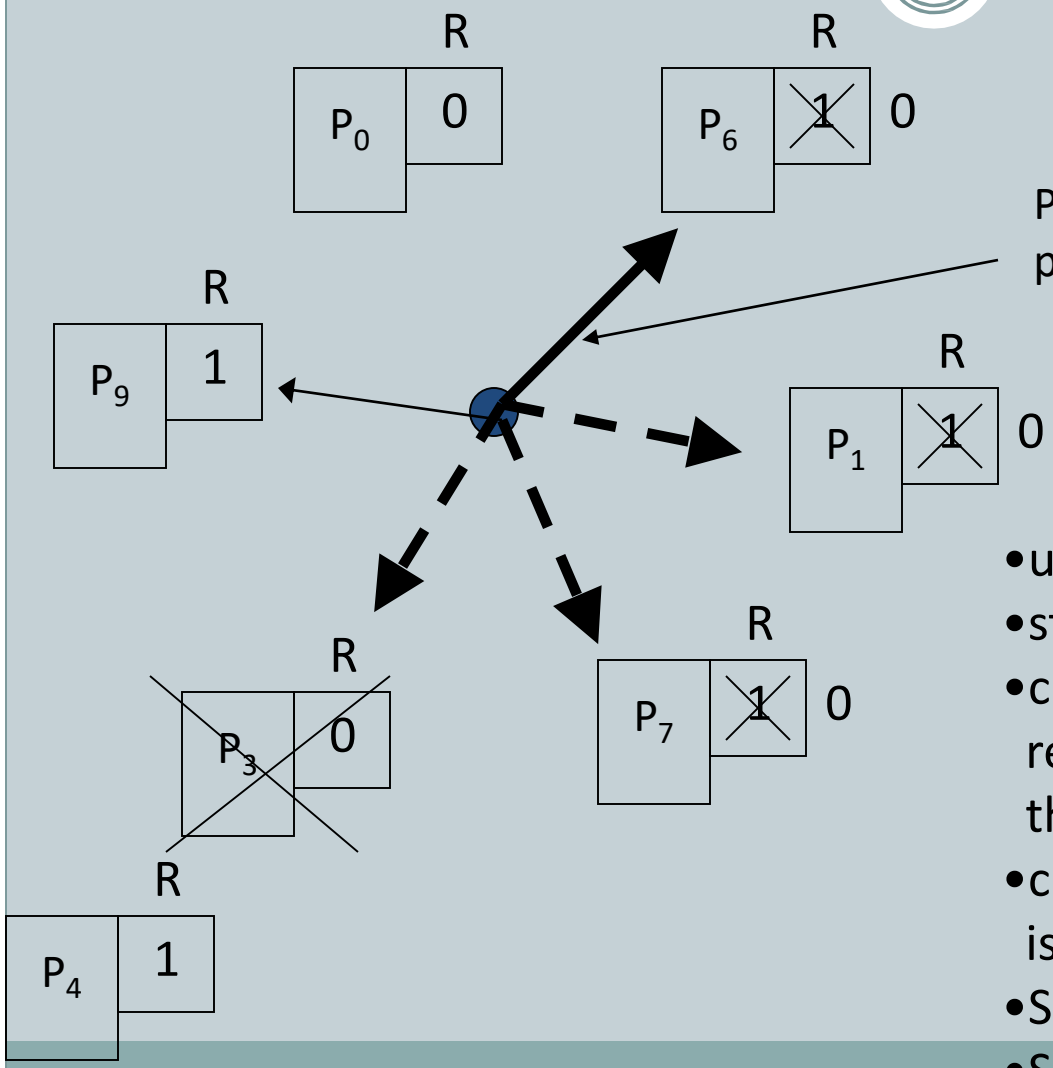
set its reference bit to zero and skip it (give it a second chance)

If a page's reference bit is set to 0

select this page for replacement

Always start the search from where the last search left off

Clock Algorithm



- user refs P_4 - not currently paged
- start at P_6
- check P_6, P_1, P_7 and set their reference bits to zero (give them a second chance)
- check P_3 and notice its ref bit is 0
- Select P_3 for replacement
- Set pointer to P_9 for next search

Dirty Pages

If a page has been written to, it is *dirty*

Before a dirty page can be replaced it must be written to disk

A *clean* page does not need to be written to disk

the copy on disk is already up-to-date

We would rather replace an old, clean page than and old, dirty page

Modified Clock Algorithm

Very similar to Clock Algorithm

Instead of 2 states (ref'd and not ref'd) we will have 4 states

- (0, 0) - not referenced clean
- (0, 1) - not referenced dirty
- (1, 0) - referenced but clean
- (1, 1) - referenced and dirty

Order of preference for replacement goes in the order listed above

Modified Clock Algorithm

Add a second bit to PT - *dirty bit*
Hardware sets this bit on write to a page
OS can clear this bit
Now just do clock algorithm and look for best page to replace
This method may require multiple passes through the list

Page Buffering

It is expensive to wait for a dirty page to be written out

To get process started quickly, always keep a pool of free frames (buffers)

On a page fault

- select a page to replace
- write new page into a frame in the free pool
- mark page table
- restart the process
- now write the dirty page out to disk
- place frame holding replaced page in the free pool