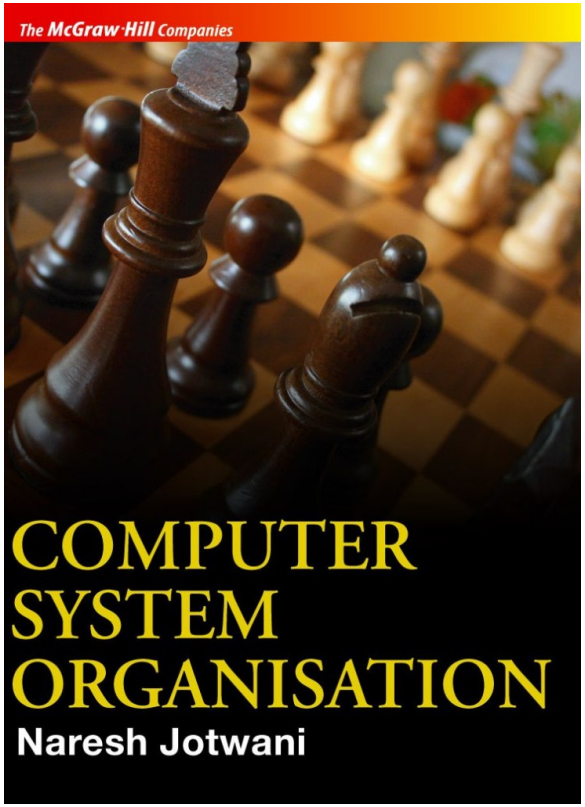
Mc Graw Hill Education

COMPUTER SYSTEM ORGANISATION

Naresh Jotwani

PowerPoint Slides



CHAPTER 9

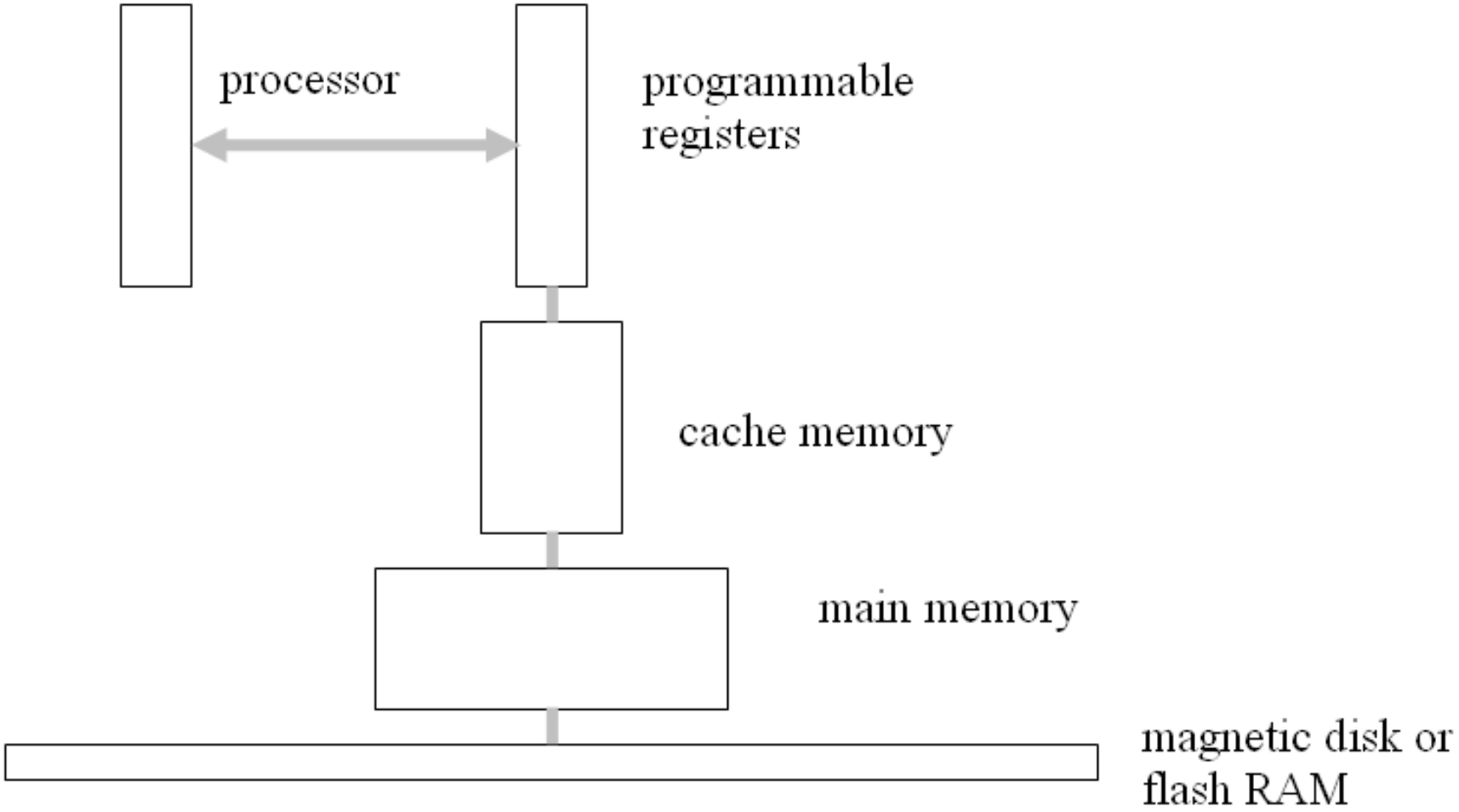
MEMORY ORGANIZATION

Memory hierarchy:

- In a computer system, there may be four distinct types of memory:
  - Registers in the processor
  - Cache memory
  - Main memory
  - Secondary storage

- The next figure depicts this using rectangles at each level of hierarchy
  - Width of the rectangles depicts memory capacity
  - Height of the rectangles depicts memory speed
  - Processor is shown as a separate rectangle on the left, interacting with processor registers
  - Heights of these two rectangles are equal, since processor registers, functional units and busses operate at comparable speeds

Memory hierarchy in a computer system :



- System designers aim for the following goals:

1. Make instructions and data available to running programs at the speed of faster memory, and
2. Make available in faster memory the instructions and data which are *currently in use*.

This strategy reduces the impact on program execution speed of accesses to slower memories

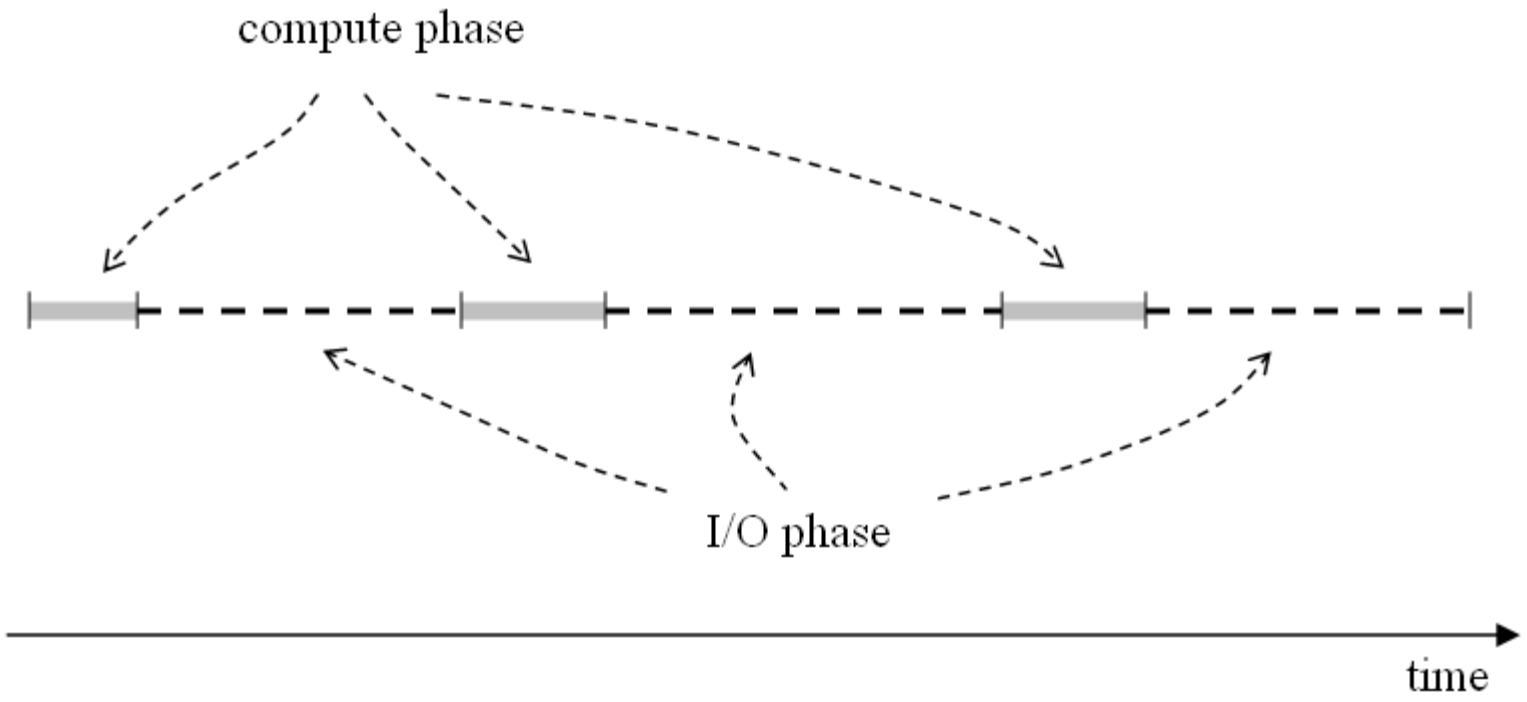
- Two important techniques which are used to extract higher performance from a computer system are: *multiprogramming* and *virtual memory*.

Multiprogramming:

- There is a major speed mismatch between *processor* and *input/output (I/O) devices.*
- Behaviour of a running program can be visualized as an alternating sequence of compute and I/O phases.

  [shown in the next slide]
- Assume that we are working with a *sequential* program.

Compute and I/O phases of a running program:



- Due to this speed mismatch, a program may spend a large fraction of its time in performing I/O operations.
- i.e. the dashed lines of *I/O phase* may be much longer than the thick lines of *compute phase*.
- But, in a sequential program, the next compute phase cannot begin until the ongoing I/O operation is over.
- Therefore the processor must remain idle - if there is no other program which can run on it.
- But low processor utilization is unacceptable.

- To increase processor utilization, the solution is:
  - Hold multiple independent programs in memory. If one program is in I/O phase, another program in *its* compute phase would run on the processor.
  - i.e. If the processor is not executing the instructions of one program, it is executing those of another program.
  - The nett effect is that processor utilization is higher, i.e. the system completes more work per unit time.
- The number of programs *n* which the system holds in memory is known as its *degree of multiprogramming*.

Example

- Say a typical program spends 10% of its time on the processor, and the remaining 90% in I/O operations.
- i.e. The probability p that such a program is performing an I/O operation at a given time is 0.9. If only one such program is running on the system, then processor utilization is 10%.
- Suppose 8 such programs run on the system. Then the processor will be idle only if all eight programs are performing I/O operations at a given time.
- The programs are independent of each other. So the probability that all 8 are performing I/O operations at a given time is $(0.9)^8 = 0.43$.
- This is the probability that the processor is idle – and therefore the processor utilization is now (1-0.43)x100 = 57%.
- Thus processor utilization has increased from 10% to 57%, with degree of multiprogramming $n = 8$.

- In fact the overall system *resource utilization* improves as a result of multiprogramming.

- For a computer system under multi-programming, a performance *trade-off* is involved between:

  – (a) utilizing system resources to the full, and
  – (b) providing acceptable running time to each program running on the system.

Partitioned memory:

- For multiple programs to reside in main memory and run independently:
  - (a) Each running program must be allocated the amount of memory required for its instructions and data, and
  - (b) No running program must make references to instructions or data outside of the main memory allocated to it.

- Solution: Divide main memory into partitions for the *n* programs – plus one other partition for the operating system.

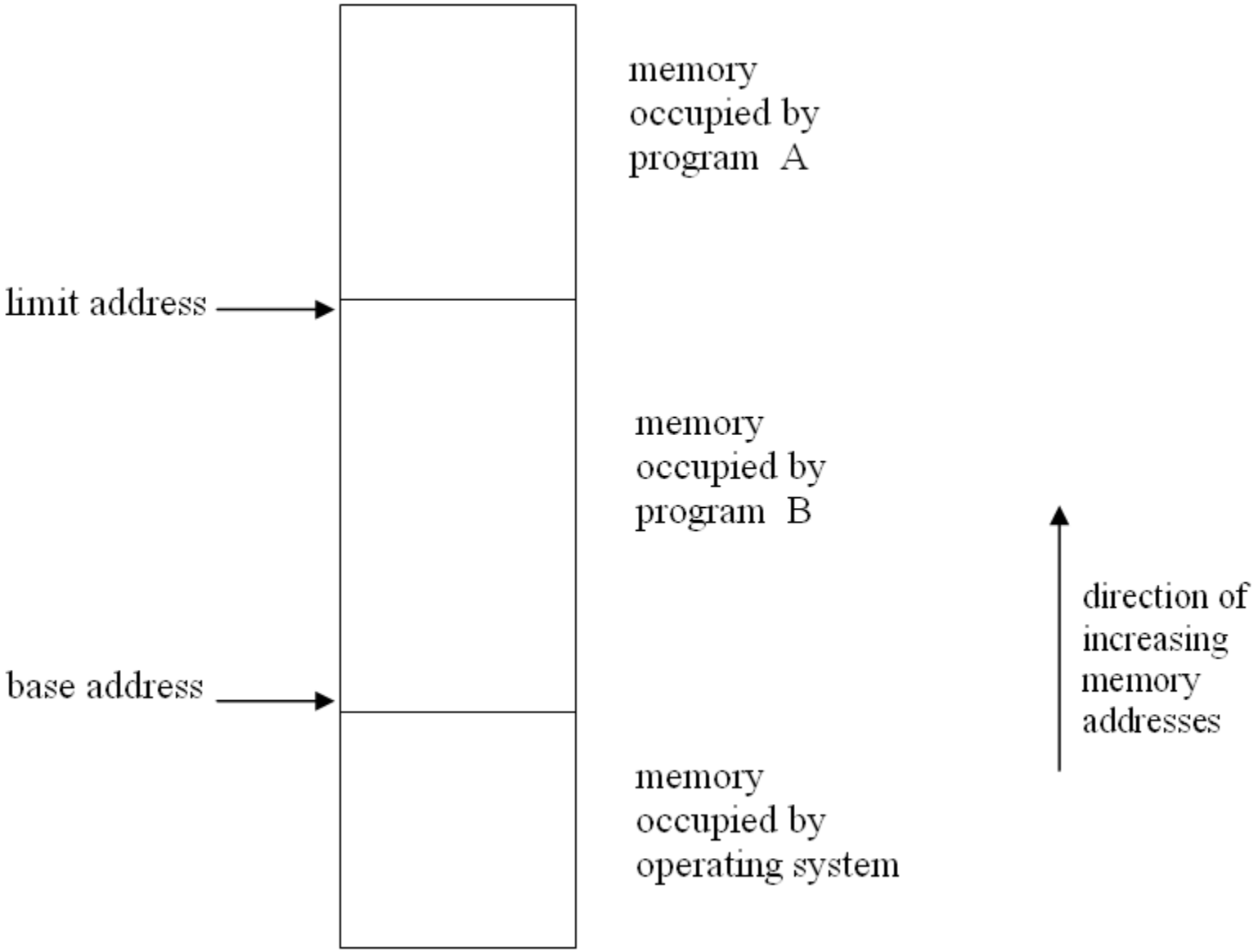
- The main memory is thus said to be *partitioned*

  – Next figure depicts partitioned main memory, shared between two programs and the operating system.

  – Here the degree of multiprogramming *n* is 2.

  – Each partition has a *start address* and an *end address* in physical memory (also termed as *base address* and *limit address* respectively).

Sharing of main memory by OS and two programs:



- For memory protection, each running program must be provided its own *logical memory space*.

- The logical memory space 'seen' by a running program is called its *logical address space*.
  - It comprises of all the memory addresses which a program can use for instructions as well as data.
  - Each such address used within a program is known as a *logical address*.

- As against this, the memory space spanned by physical memory addresses is called the *physical address space* of a system

- Consider the machine instruction: **`LOAD 2000, R1`**
- Memory address 2000 here is a *logical* address.
- Suppose this program occupies physical memory starting at *physical* memory address 40000.
- The location 2000 above must be seen as *relative* to the physical starting address of the program.
- Therefore the reference to *logical* address 2000 will be *mapped* to physical address 40000 + 2000 = 42000.
- Thus:

physical address = base address + logical address

- To protect other partitions from invalid access, we need to ensure: *physical address* $\leq$ *limit address*

- Thus this mapping from logical address to physical address involves one addition, and an extra comparison for memory protection.

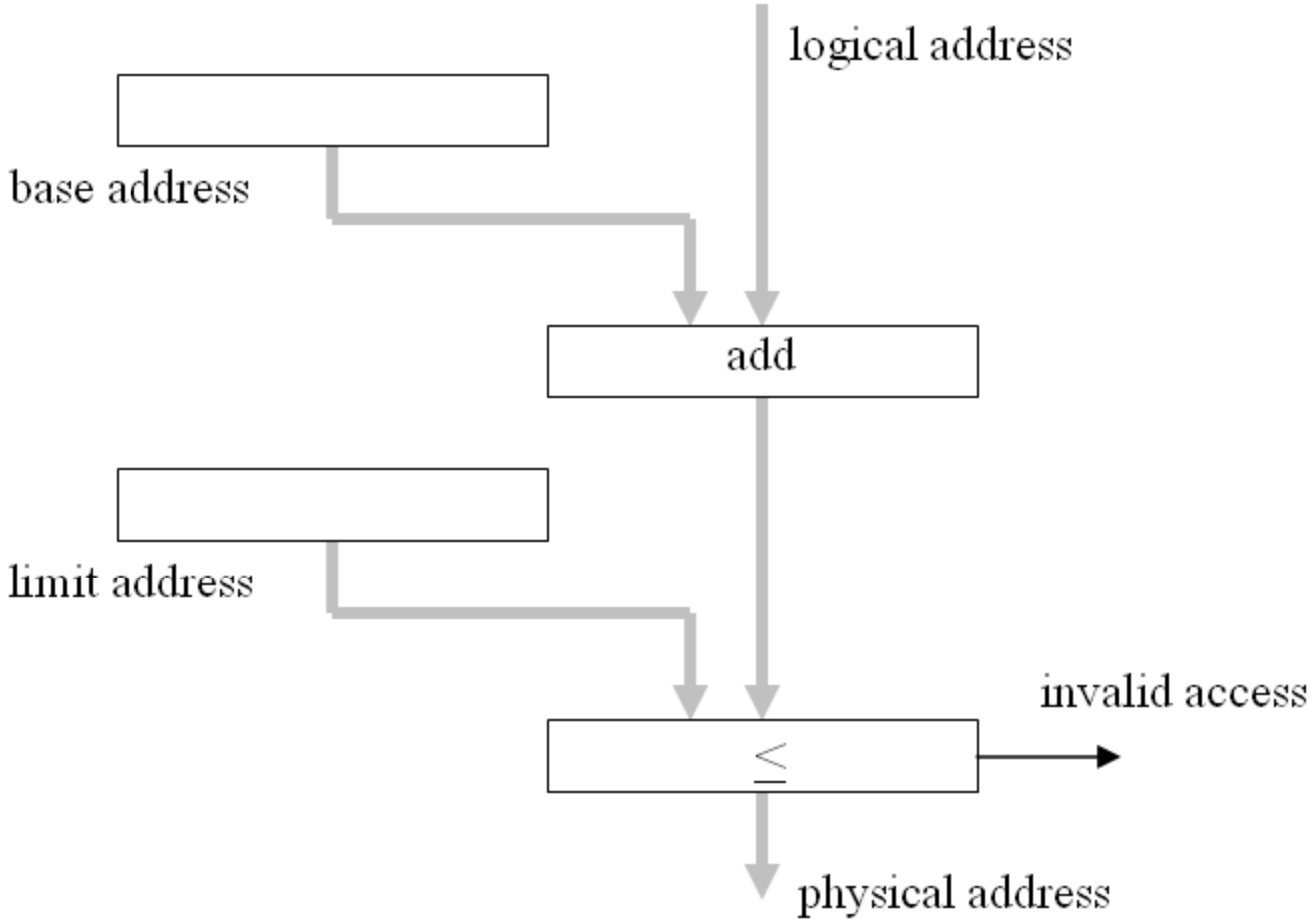
- The next figure shows the hardware for implementing this mapping.

  (i) The *address* register provides the logical address.

  (ii) The physical address is carried via address lines to the memory sub-system.

- Thus, for partitioned memory, the *memory management* hardware shown is added on to the processor. Hardware raises *invalid access* exception if running program attempts memory access outside its partition.

Mapping from logical to physical address, for partitioned memory:



- For a running program, the operating system ensures that the *base* and *limit* registers in the processor are loaded with the correct values.

- To do this, *privileged instructions* are required, which only the operating system can execute.

- Any processor supporting multiprogramming must provide such instructions.

- The program status word (PSW) must have *mode* bit(s) indicating whether the processor is running in *privileged mode* (a.k.a. *supervisor mode*) or *user mode*.

Non-contiguous memory allocation:

- In a system with multiprogramming, the number of programs running on the system changes with time, as do their memory requirements.
- As one program terminates, the memory partition occupied by it becomes available for another program.
- After such a multiprogramming system runs for some time, the partitioned main memory of the system becomes *fragmented*.

- A *fragment* is an unused block of memory which cannot be utilized for running a program, either because:

  (a) the fragment is too small for a running program, or

  (b) the fragment is within a larger memory partition allocated to a program

- These two types of fragmentation – *external fragmentation* and *internal fragmentation*, resp. – lead to poor memory utilization.

- The root cause of such fragmentation is in the basic assumption that:

  – Program requiring amount of memory P must be loaded in a *contiguous* memory partition of size $S \geq P$.

- To get out of this limitation, we can try the following:

  (i) Allocate main memory to programs in multiple units of fixed size – called *pages* – making this unit size independent of program size

  (ii) Allocate to any running program as many pages as it requires – but not necessarily in a single contiguous block

- Typical *page size* in computer systems may be in the range from 1 kilobyte ($=2^{10}$ bytes) to 4 kilobytes ($=2^{12}$ bytes).

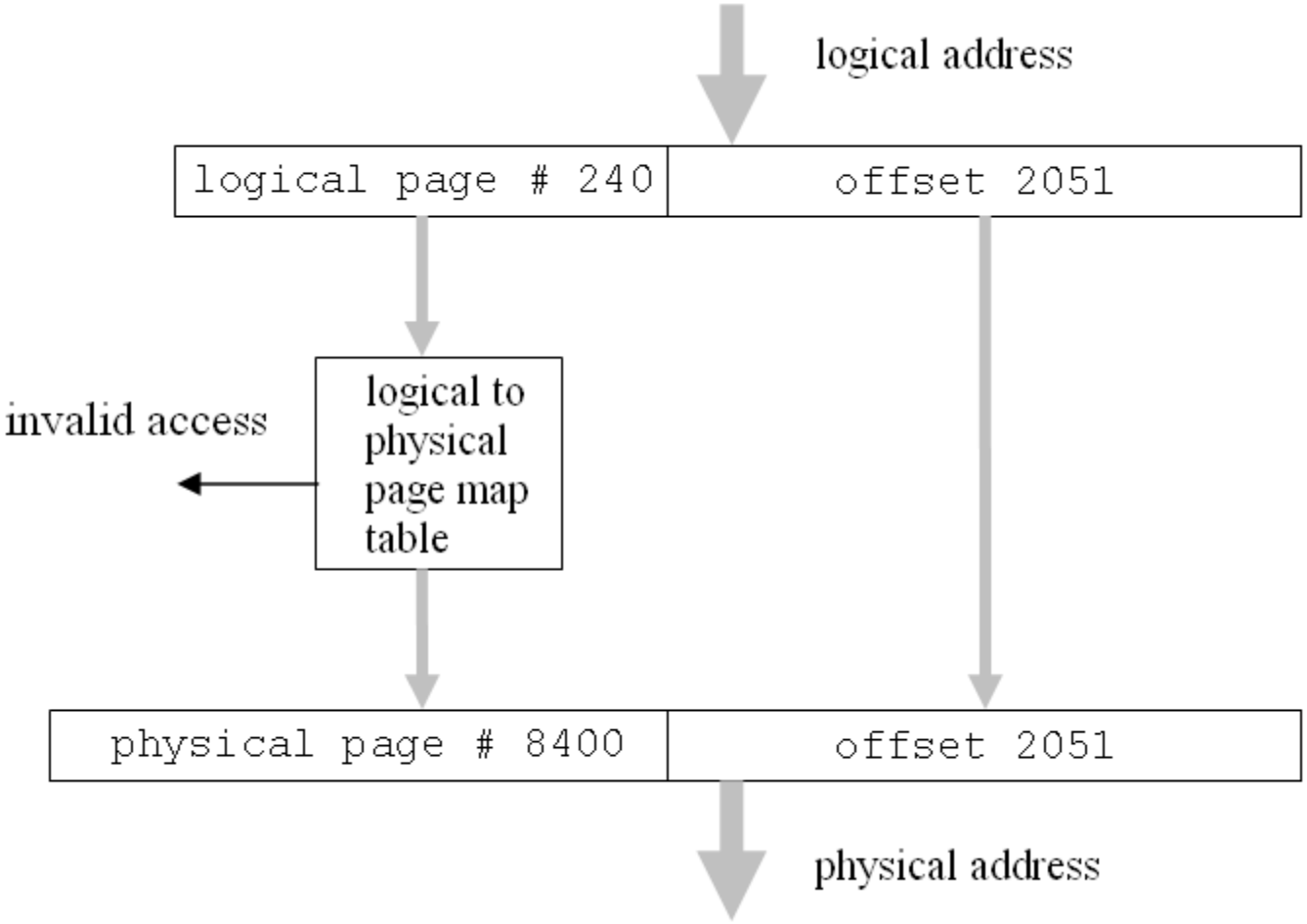
- When memory is thus allocated, a memory address gets divided between a *page number* and an *offset* within page.

- Logical to physical address translation for such a system is illustrated in the next slide.

- Logical page number gets mapped to the corresponding physical page number, while the offset within page is carried forward unchanged.
  - The mapping from logical to physical page number is implemented in hardware using a fast memory element in the form of a *page map table*.

- The logical page number serves as the *index* into the page map table.

Logical to physical address translation:



Page map table entry also stores memory protection bits for each page, according to which *read*, *write,* or *execute* access is granted to the running program for this page.

- With this type of mapping, it is not necessary that consecutive logical pages of a running program get mapped into consecutive physical pages in main memory.
- Physical pages – also known as *page frames* – can be allocated to a program anywhere in physical memory.
- Thus the physical memory allocated to a running program becomes *non-contiguous*.
- This type of memory allocation solves the problem of fragmentation to a great extent.

- Thus we have seen two different ways in which multiprogramming can be supported.

- With contiguous memory allocation, base and limit address values of each running program must be maintained. These must be loaded into base and limit registers when the program begins – or resumes – execution.

- With non-contiguous memory allocation, the page map information for each running program must be maintained. This must be loaded into the page map table when the program begins – or resumes – execution.

Principle of Virtual Memory

- Basic idea: Over time periods of the order of tens of milliseconds or longer, only the currently referenced portions of the logical address space of a running program are loaded in main memory.
  - When reference is made to a logical address which is not available in main memory, the portion containing it is brought in from secondary storage.
  - The entire logical address space of a running program is maintained only in secondary storage.
- This provides a computer system with *virtual memory*, since it allows the total extent of logical address space of a running program to exceed the amount of main memory allocated to it.

- System performs additional work in transferring portions of running programs between secondary storage and main memory.
  - Since main memory is better utilized, a higher degree of multiprogramming is made possible.
- Large programs consist of many commands, options and operating modes. Loading of an entire program into main memory may take a few seconds, making the system appear slower in responsiveness.
- With a virtual memory system, only the initially required features of the program are loaded into main memory. Other parts of the program are loaded into main memory as required, thus reducing the demand on main memory, and making the program more responsive.

Virtual Memory with Paging:

- To implement virtual memory, the following questions must be addressed:

  (1) How do we define ‘portions of a running program’?

  (2) How does the system determine which portions are ‘currently in use’?

  (3) What are the implications of this design on the logical to physical address translation scheme?

- One option: Virtual memory designed on the basis of fixed size pages and non-contiguous memory allocation.
- For any program, we load into memory the set of *logical* pages currently in use.
- The set of logical pages of a running program currently in use is known as the *working set* of the program.
- Of course the working set of a program changes with time.
- If a running program refers a logical page not available in main memory, such a ‘missing page’ event generates a hardware exception known as a *page fault*.

- Logical pages required by a program are brought in 'on demand' – i.e. when the program generates a page fault on a required page. This mechanism is known as *demand paging*.

- Another mechanism is also required to free up physical pages in memory – i.e. page frames – to make room for incoming pages demanded by a running program.

- This *page replacement algorithm* is based on removing from main memory either

  (a) a page which is not among the recently used pages, or

  (b) a page which has not been frequently used in the recent past.

- Question (3) listed above pertains to the translation required from logical to physical address. For this, the page map table mechanism is used.

- If a logical page is available in main memory, the page map table provides its physical address, as seen earlier.

- If a logical page is not available in main memory – i.e. in case of *page fault* – the page map table provides its address in secondary storage.

- The operating system then arranges to bring the required page into main memory, after freeing up a page frame for it by using the page replacement algorithm.

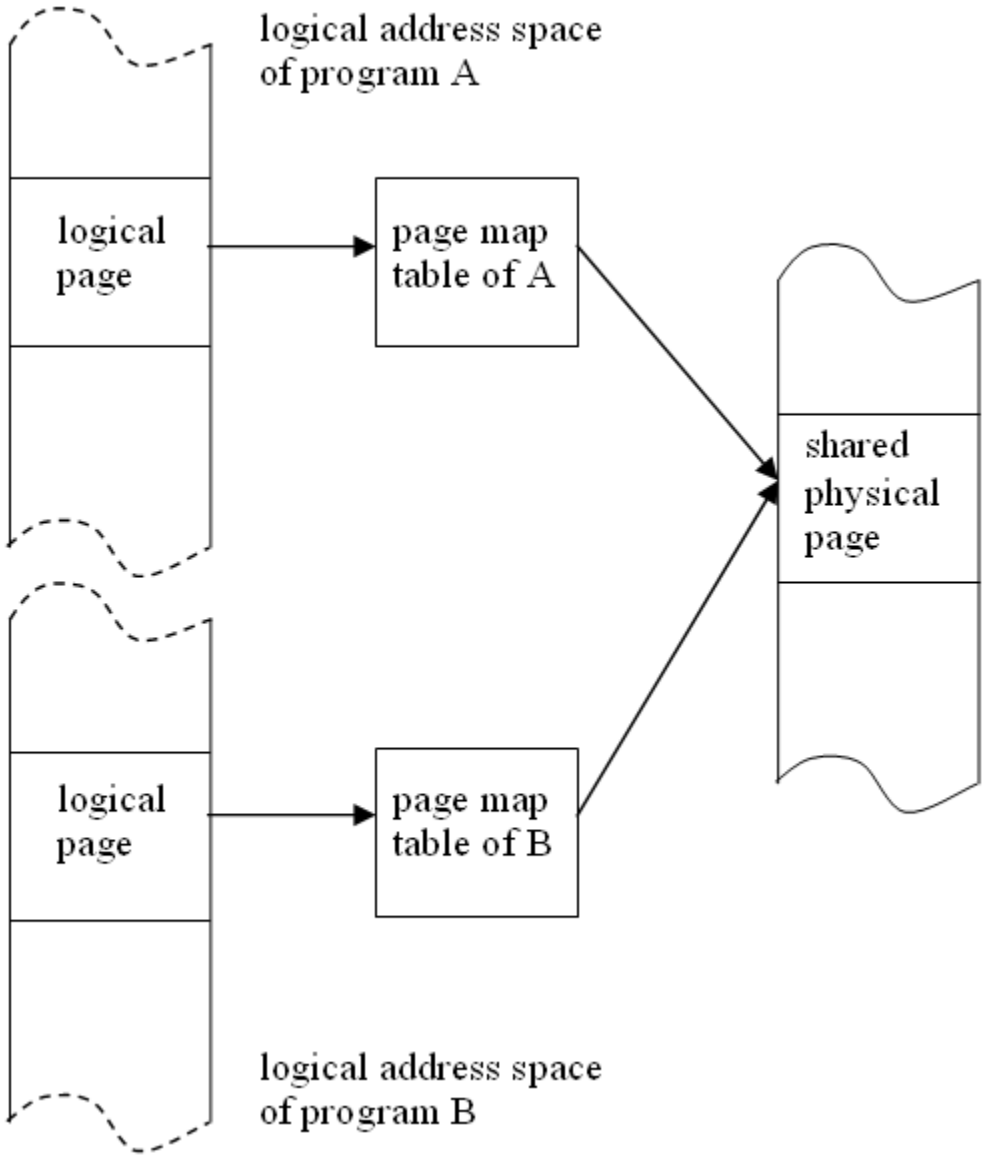
- Servicing of page faults is an additional overhead on the system. But as long as their frequency is not too high, the overall system throughput improves, due to higher degree of multiprogramming achieved.

- Under multiprogramming, two or more running programs may need to *share* a part of their logical address and/ or data space.

- In a scheme based on fixed size pages – with or without virtual memory – such sharing can be provided in units of pages.

- When a page is shared between multiple programs, that physical page appears in the page map table of each of the programs, as shown below.

Sharing of physical page between two running programs:



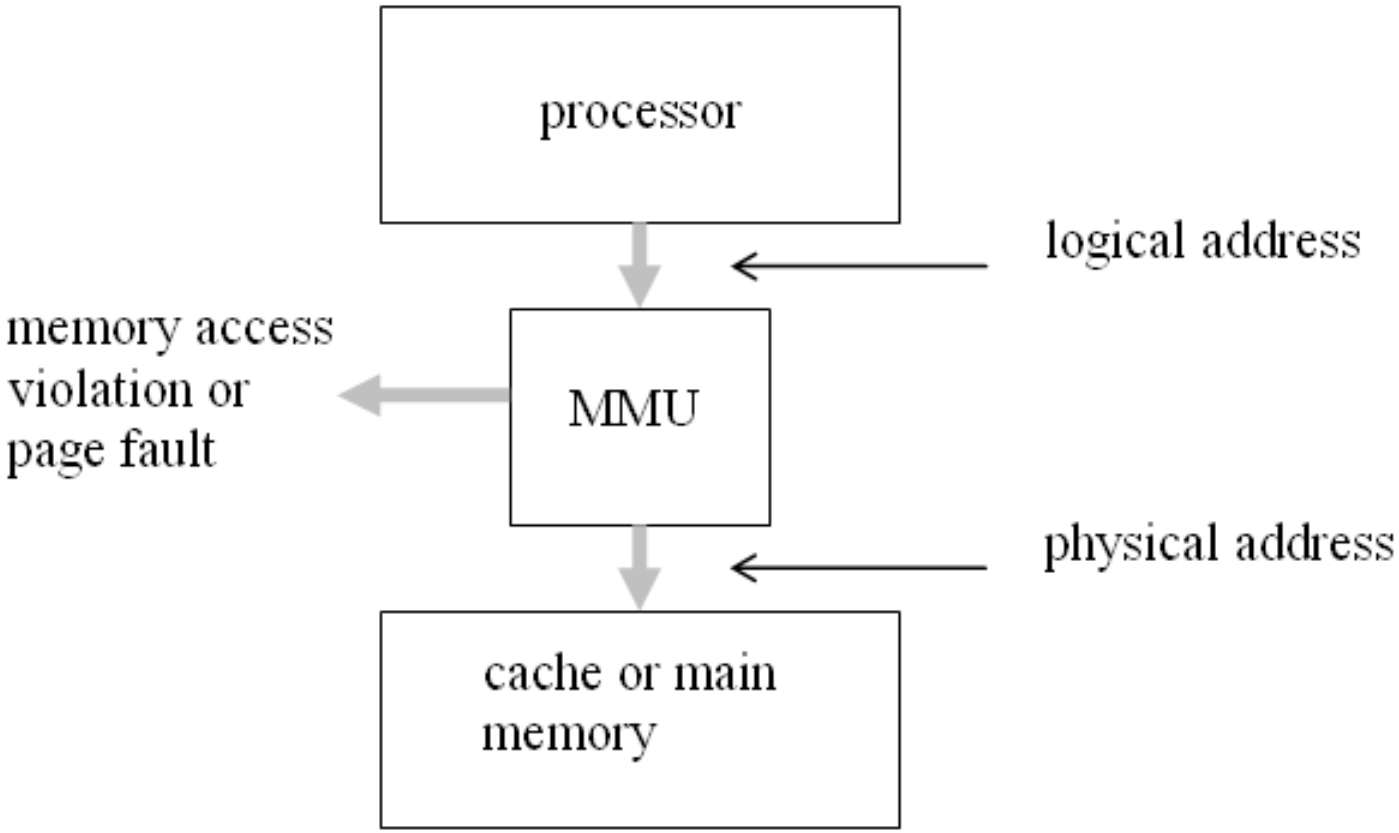
- Thus the mechanism of paging can support:
  (i) virtual memory,
  (ii) sharing of pages, and
  (iii) page-wise access protection, i.e. *read*, *write* and/or *execute* access per page.

- A page fault, and also any form of access violation, is detected by the page map hardware, for appropriate action to be taken by the operating system.

- This hardware is the *memory management unit*, which is closely associated with the processor.

- [See next slide.]

Role of memory management unit:



Segmented logical address space:

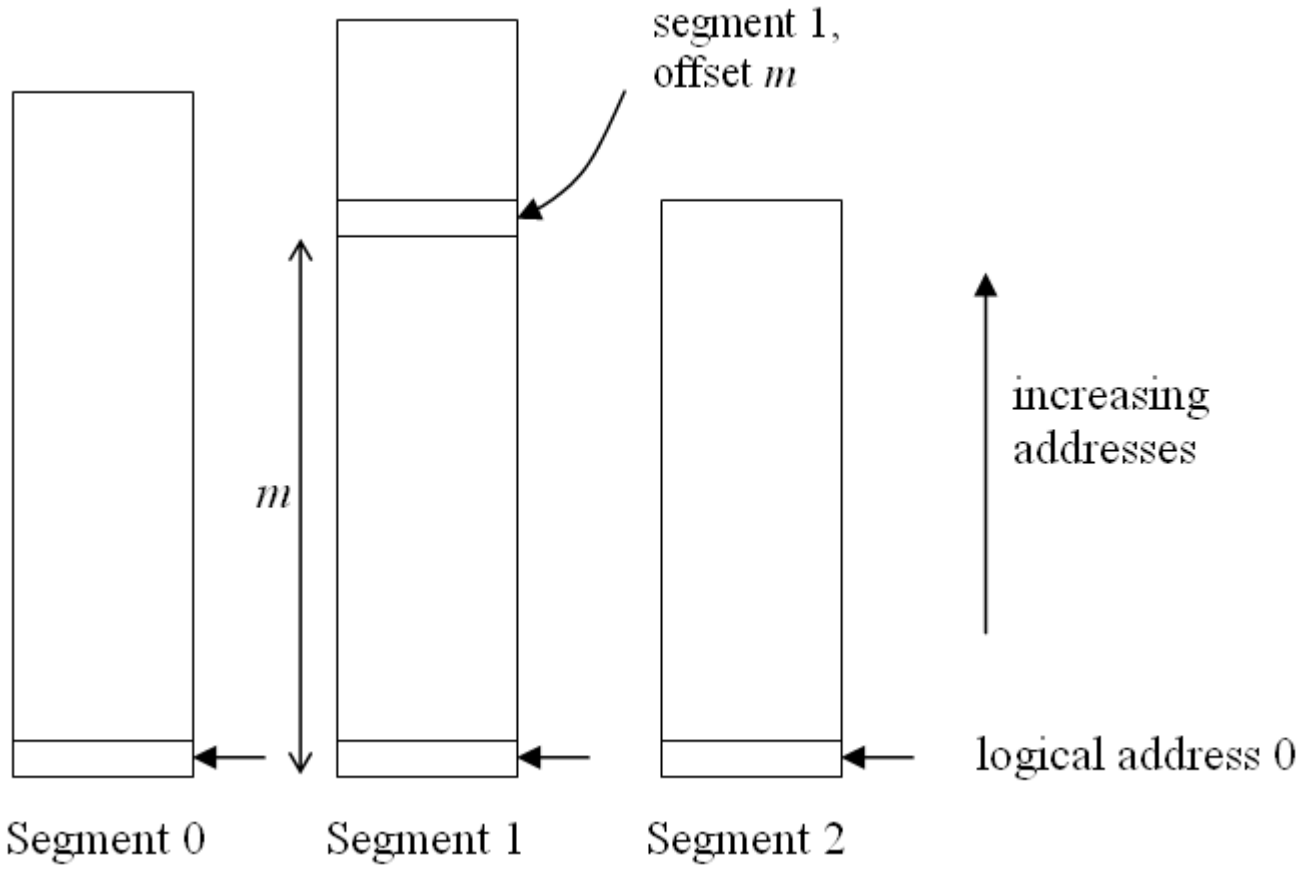
- Large applications are often developed in the form of several independently running programs.

- Running programs interact with each other by
  (a) sharing code and/or data, and/or
  (b) sending and receiving *messages*.

- For such applications, logical address space of a running program can be provided as several independent *segments* – each being a linear sequence of bytes.

- See the next figure. The logical address space of a single program has three *segments*: 0, 1 and 2.

Segmented logical address space of a single program:



Each segment is a linear sequence of bytes, from 0 to N_k-1, where k is the segment number, i.e. here k = 0, 1, 2. Size of segment number k is N_k.

- The logical address of any memory location is now split into two parts:

  `segment number : offset within segment`

- This is shown in the previous slide, for a memory location at offset *m* within segment 1.

Example:

- A processor with segmented memory has logical address made up of 8-bit segment number and 24-bit offset within segment.
  - Size of the two-part logical address is 32 bits.
  - The maximum number of segments a running program can address is $2^8 = 256$, with each segment limited to 2^{24} bytes = 16 megabytes.

- Each memory reference made in the program – for load/store, conditional or unconditional jump, or function call – should specify the 32-bit address.

- But provision is made in the instruction set for memory references which do not require a separate segment number. This is done by providing within the processor special *segment registers* to hold the starting addresses of commonly used segments.

- Suppose *code segment register* holds the starting address of the segment which has program code. Then *intra-segment* jumps and function calls do not need to specify the 8-bit segment number.

- Similar arrangements can be made for references to *data* segment and the *stack* segment.

- Techniques of paging and virtual memory can be combined with segmented logical address space.
- For this, each segment must be provided with its own page map table.
- There are applications in which two or more programs need to *share* a segment. This can be achieved if each program sharing the segment uses the page map table of that segment.
- Memory protection can also be provided on segment-wise basis by providing appropriate *read*, *write* and *execute* flags per segment.

Associative memory

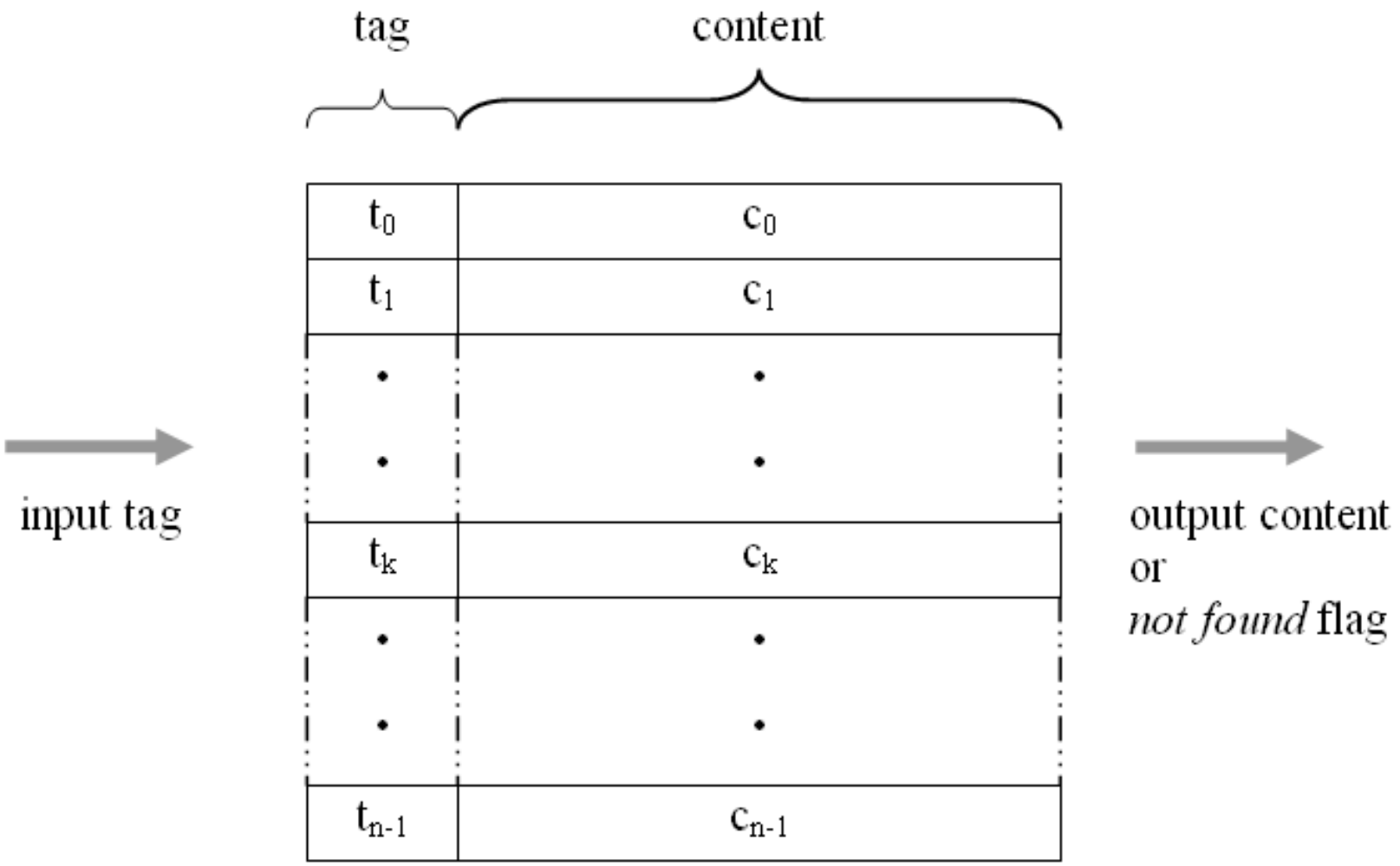
- Main memory is usually *byte addressable*, and often it is designed so that multiple bytes – 2, 4 or 8 – can be read or written in one memory cycle.

- The twin concepts of (i) *memory address*, and (ii) *data* stored at the address are central to main memory organization.

- But certain specialized functions require *content addressable* – a.k.a. *associative* – memory elements. Data in such a memory element is located by a part of the content stored in the memory element itself.

- [This principle is illustrated in the next figure.]

Principle of associative memory:



- Figure shows an associative memory having *n* locations. Data stored in each location is made up of two parts: *tag* t_k and *content* c_k, for $0 \leq k \leq n\text{-}1$.

- To locate a data item in such an associative memory, it is supplied with the *tag value* being sought.

- This value is 'matched in parallel' with all the tag values stored. If a match is found, the corresponding content is read out and placed on memory output; otherwise, a hardware flag *not found* is raised.

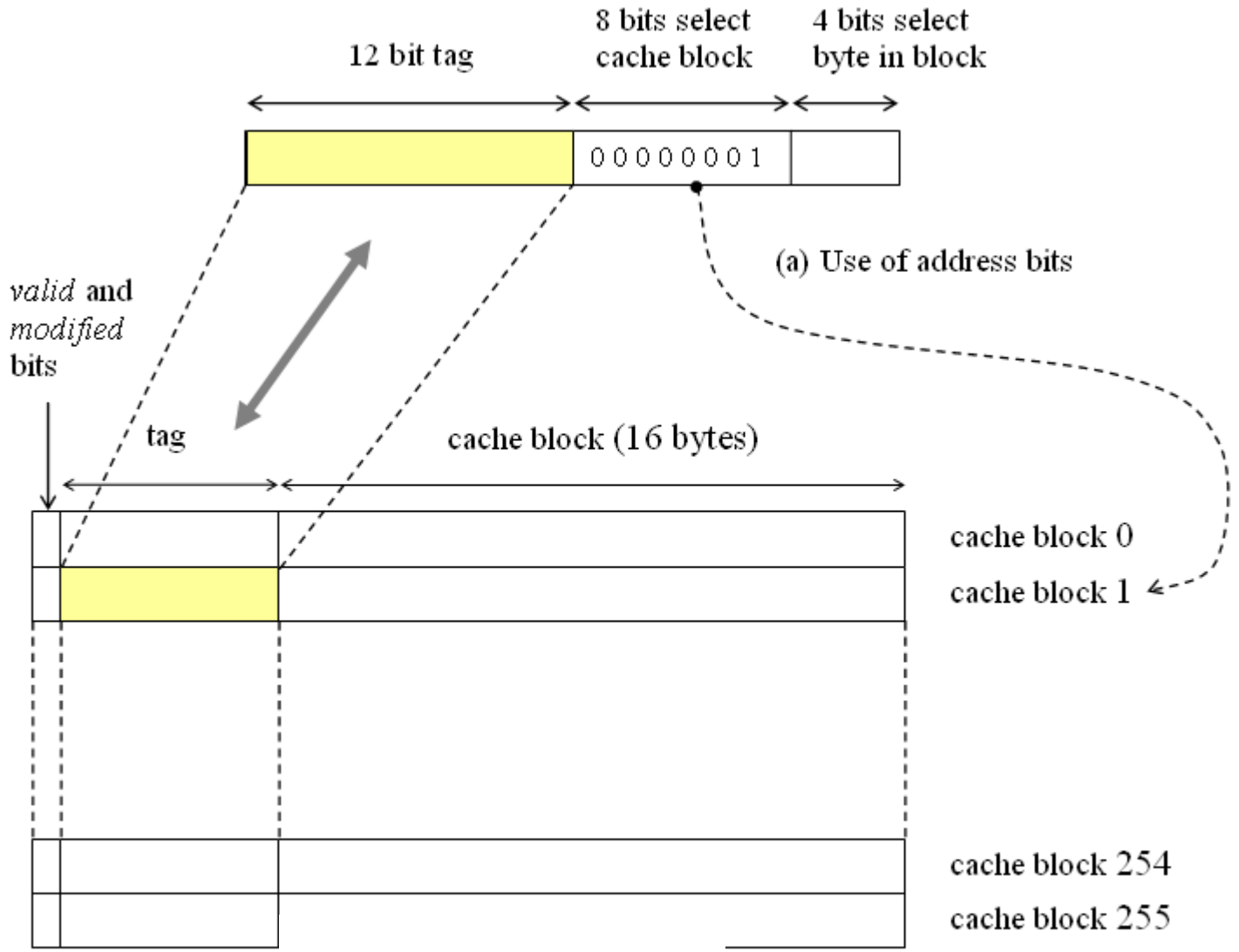
- With RAM, the search would require on average $n/2$ memory accesses and comparisons. With associative memory, it is done in a single clock cycle, since the comparisons are made in parallel.

Cache memory organization

- Running program exhibits both *temporal locality* and *spatial locality* in memory references.
- To take advantage of spatial locality, cache block size is made a multiple of word size. When a byte or word is brought into cache, its neighbouring bytes and words are also bought in.
- When the processor makes an access to a byte or word in main memory, the required byte or word is sought in cache. For this, memory address of the accessed byte or word is *mapped* to a location in cache memory.
- The simplest way to map a memory address to a location in cache is known as *direct mapping*.

Example of direct mapped cache:

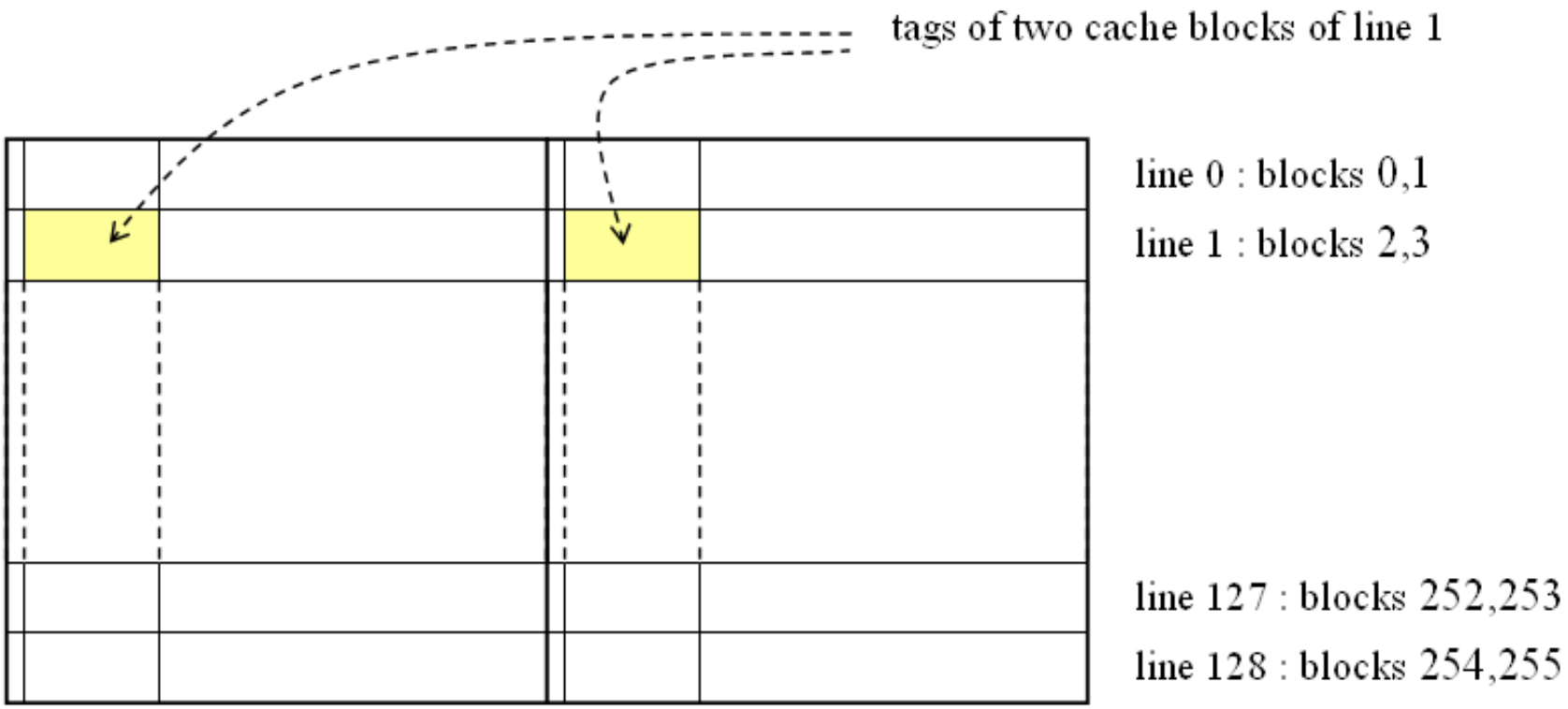
4 kbyte cache, 24 bit address, 16 byte cache block



(b) Organization of cache

- But, with a large address size, the number of memory blocks mapped to the same cache block will be large.
- When a program makes references to two or more memory blocks mapping to the same cache block, repeated cache misses are generated.
- *Set-associative mapping* of cache blocks is designed to overcome this weakness – but at the expense of a more complex design.
- In a set-associative cache, typically two or four candidate cache blocks are provided – any one of which a memory block can occupy.

Two-way set-associative cache:

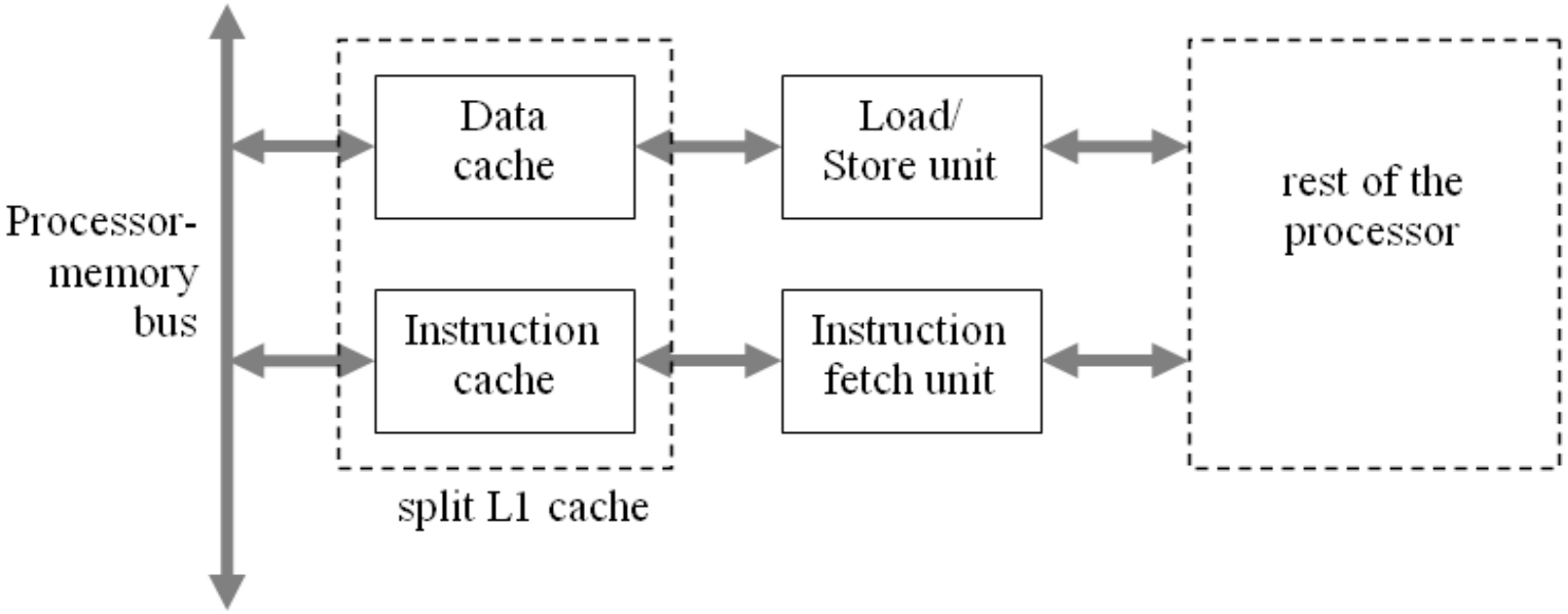


- *Four-way set associative* cache has four candidate cache blocks in every cache line; any one of these is occupied by a memory block which maps to the line.

- In the above example, this implies 64 cache lines, 6 bits of logical address to select a line, and the higher order 14 logical address bits serving as tag.

- When a write operation takes place to a byte or word in cache, when does the main memory copy of the byte or word get written?
  - One option: Initiate main memory write at the same time as the cache byte or word is written → *write-through* cache.
  - Second option: Write the cache block back to main memory only when the block is removed from cache → *write-back* or *deferred write* cache.

Use of split L1 cache



• For faster access to instructions and data, pipelined processors are provided with separate instruction and data caches at L1 level. This results in a smaller number of pipeline stalls.

• Note: Since the program does not modify the machine instructions being executed, I-cache content does not need to be written back to main memory.

Summary

- Memory hierarchy
- Technique of multiprogramming
- Contiguous versus non-contiguous memory allocation; use of fixed size memory pages
- Virtual memory system
- Segmentation
- Associative memory
- Cache memory organization; split L1 cache for instructions and data