ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems for Automation M

10. Memory

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

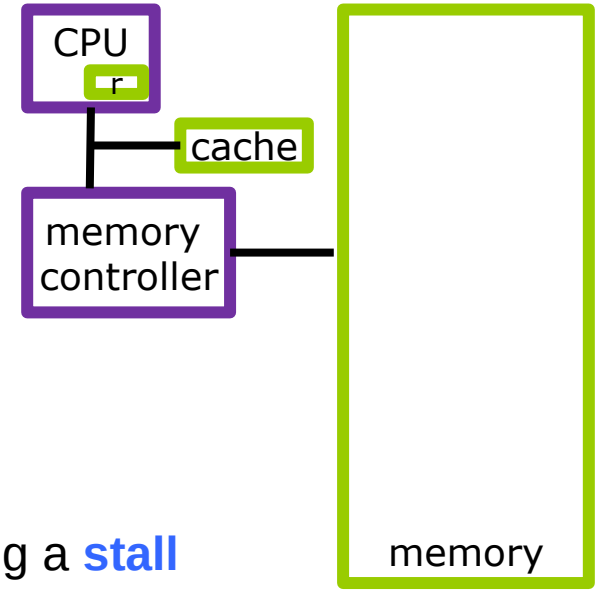
Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Chapters 8&9: Memory Management

- Background
- Contiguous Memory Allocation
- Segmentation
- Paging
- Virtual Memory

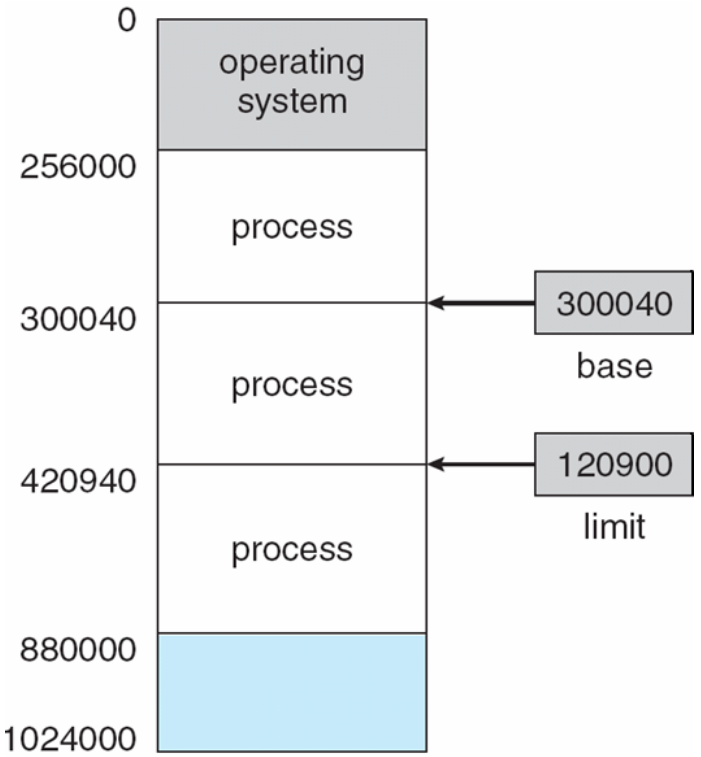
Memory Management

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory, cache and registers are the only storage the CPU can directly access
- There is difference in the access speed:
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
  - **Cache** sits between main memory and CPU registers
- **Protection** of memory required to ensure correct operation
- **Transparency** of the memory management from the point of view of the process

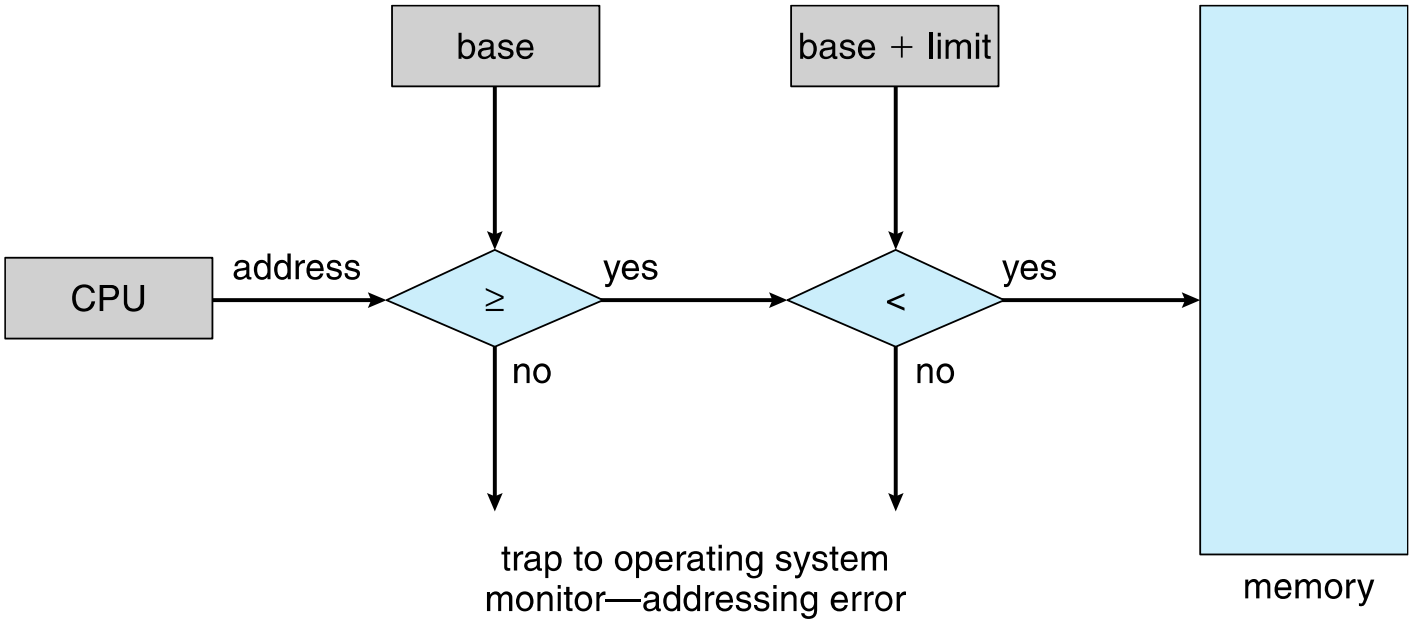


Base and Limit Registers

- Basic way to ensure **protection**:
  - divide the memory into segments
  - a pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that process

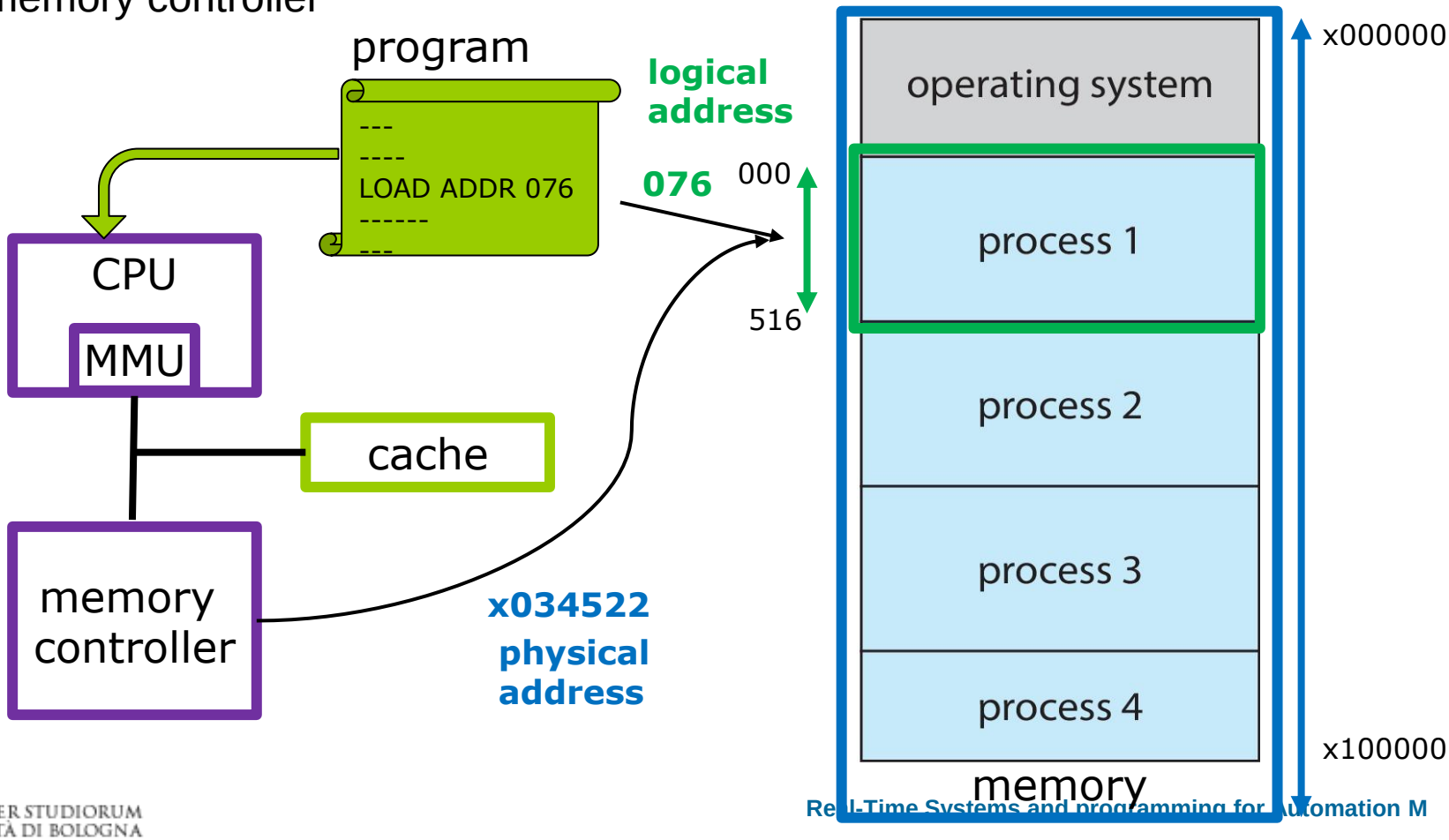


Hardware Address Protection with Base and Limit Registers



Transparency through Logical and Physical addresses

- Each process execution must be transparent to the other processes
- Introduce a difference between the addresses seen by the CPU (produced by process execution) and the addresses seen by the memory controller



Logical vs. Physical Address Space

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

- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

- Purpose: **transparency**!
  - If the process is removed from memory and then loaded again, the logical values will remain the same

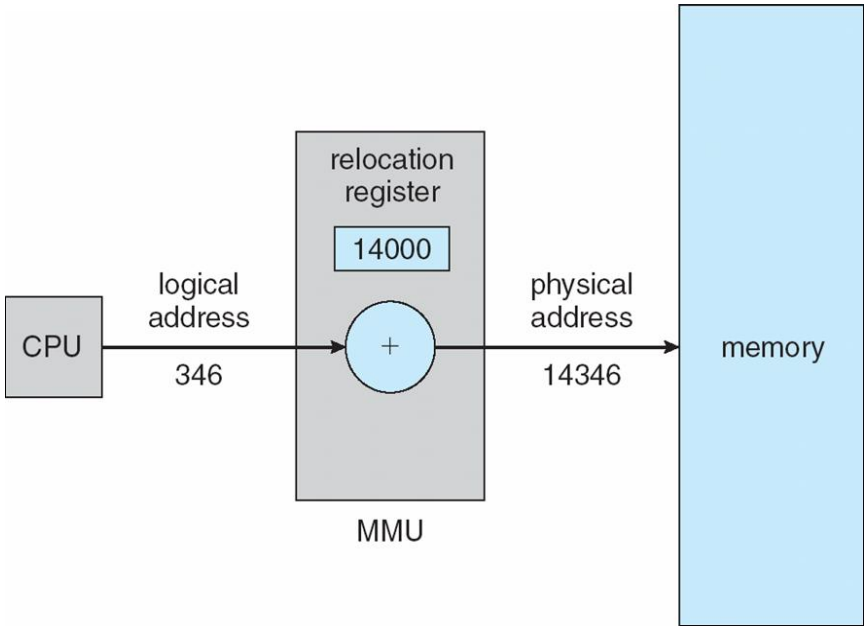
Address Binding

- Besides protection, the OS must isolate each process from the other
  - distinction between logical (or virtual) and physical addresses
  - address binding refers to the connection between logical and physical

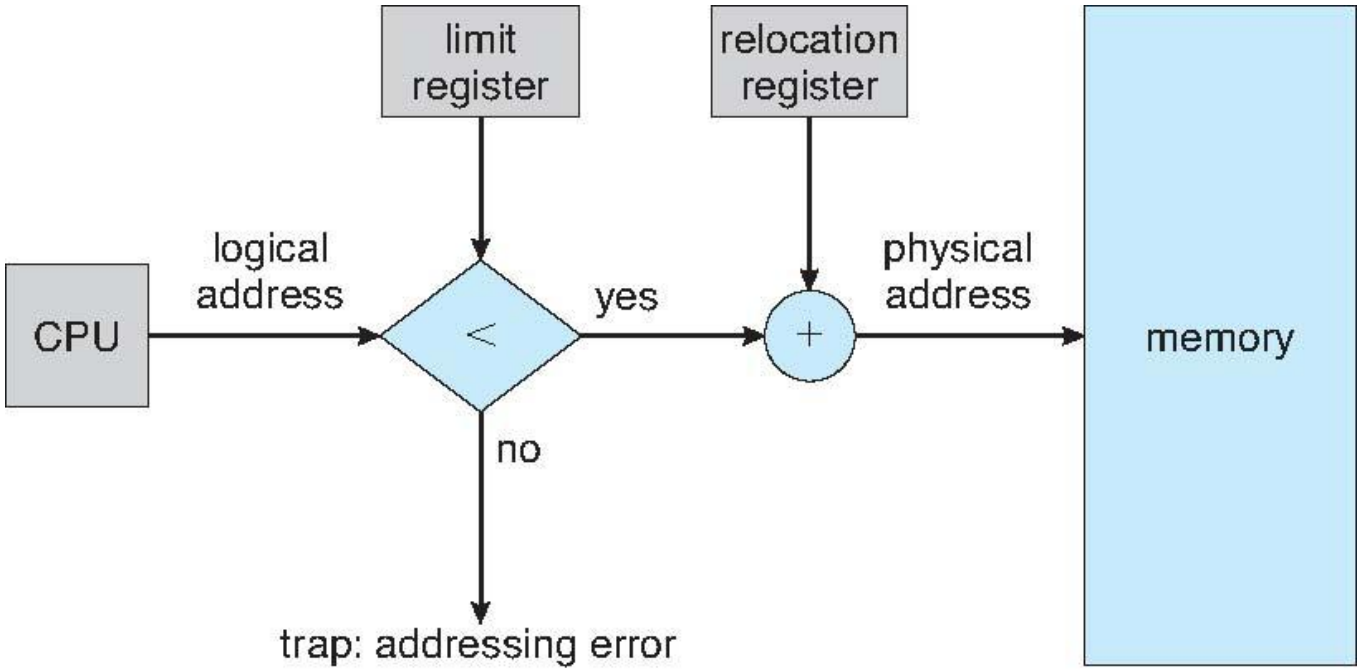
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
    - i.e., int x
  - Compiled code addresses **bind** to relocatable addresses
    - i.e. "14 bytes from beginning of this module"
  - Loader will bind relocatable addresses to absolute addresses
    - i.e. 74014 address known by physical memory
- **Each binding maps one address space to another**

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Many methods possible
- To start, consider simple scheme where the value in a **relocation register** is added to every address generated by a user process at the time it is sent to memory
  - relocation register contains the same as base register but used for bindings rather than protection purposes



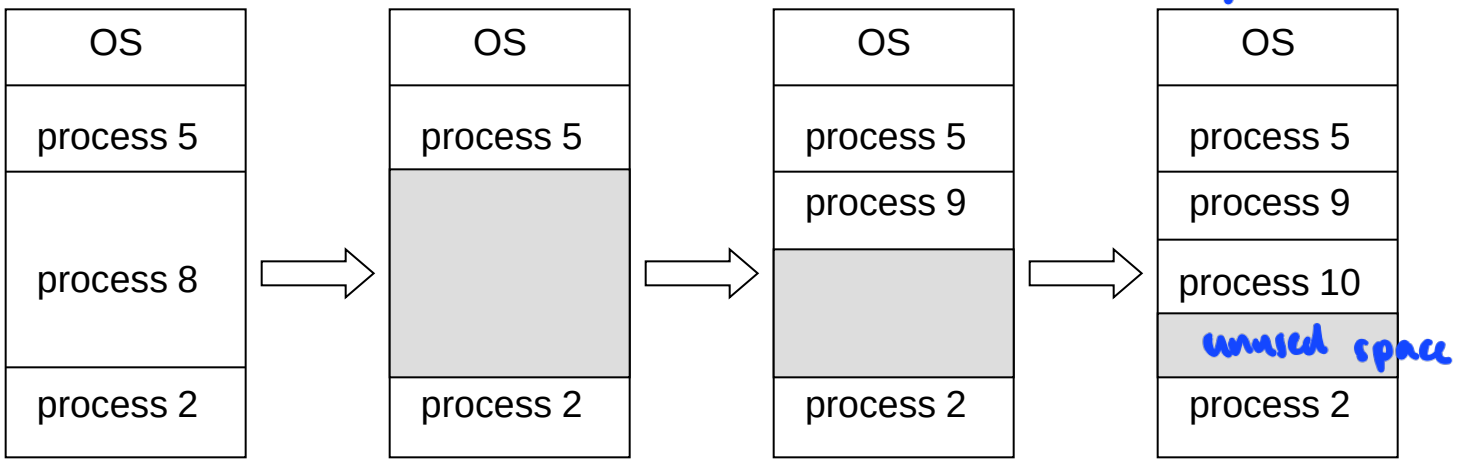
Hardware Support for Relocation and Limit Registers



works only in case of contiguous allocation

Contiguous Allocation

- Main memory usually divided into two **partitions**:
  - Resident operating system, usually held in low memory
  - User processes then held in high memory
- Contiguous alloc.: each process contained in **single contiguous section** of memory
- generates **holes** (block of available memory) of various size scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)
- Very simple management, but prone to memory **fragmentation**



Dynamic Storage-Allocation Problem

- How to satisfy a request of size ***n*** from a list of free holes?

saving time

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

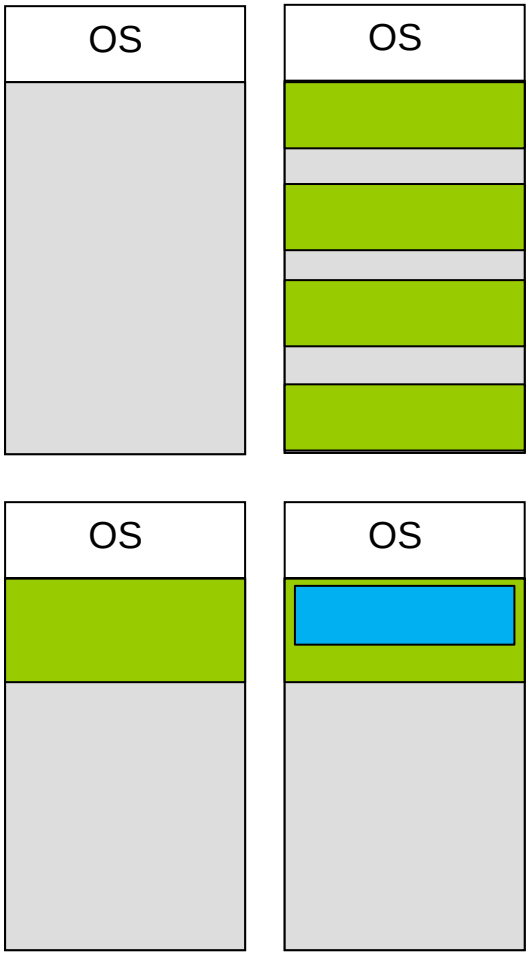
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

- However, there is always a certain degree of memory fragmentation

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- In the worst case, we could have a block of wasted memory between every two processes.
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

losing 50% mem. for each process



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

(heap and stack)

Fragmentation

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
- In general, **compaction is rather problematic**
  - Notice that disk has similar fragmentation problems
- Possible solution: noncontiguous allocation
  - Segmentation, paging

Segmentation

- Memory-management scheme that supports user view of memory
- Split the process memory image into segments, that can be allocated noncontigously, so that these smaller pieces can fit smaller holes in memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

    object

    local variables, global variables

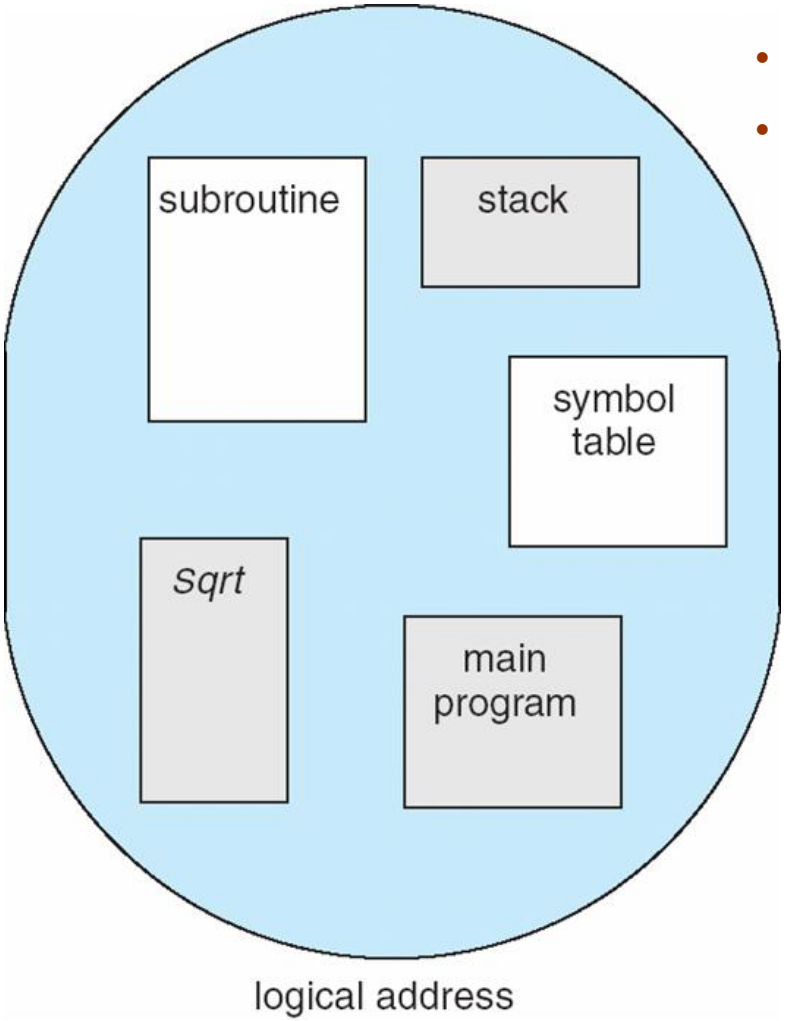
    common block

    stack

    symbol table

    arrays

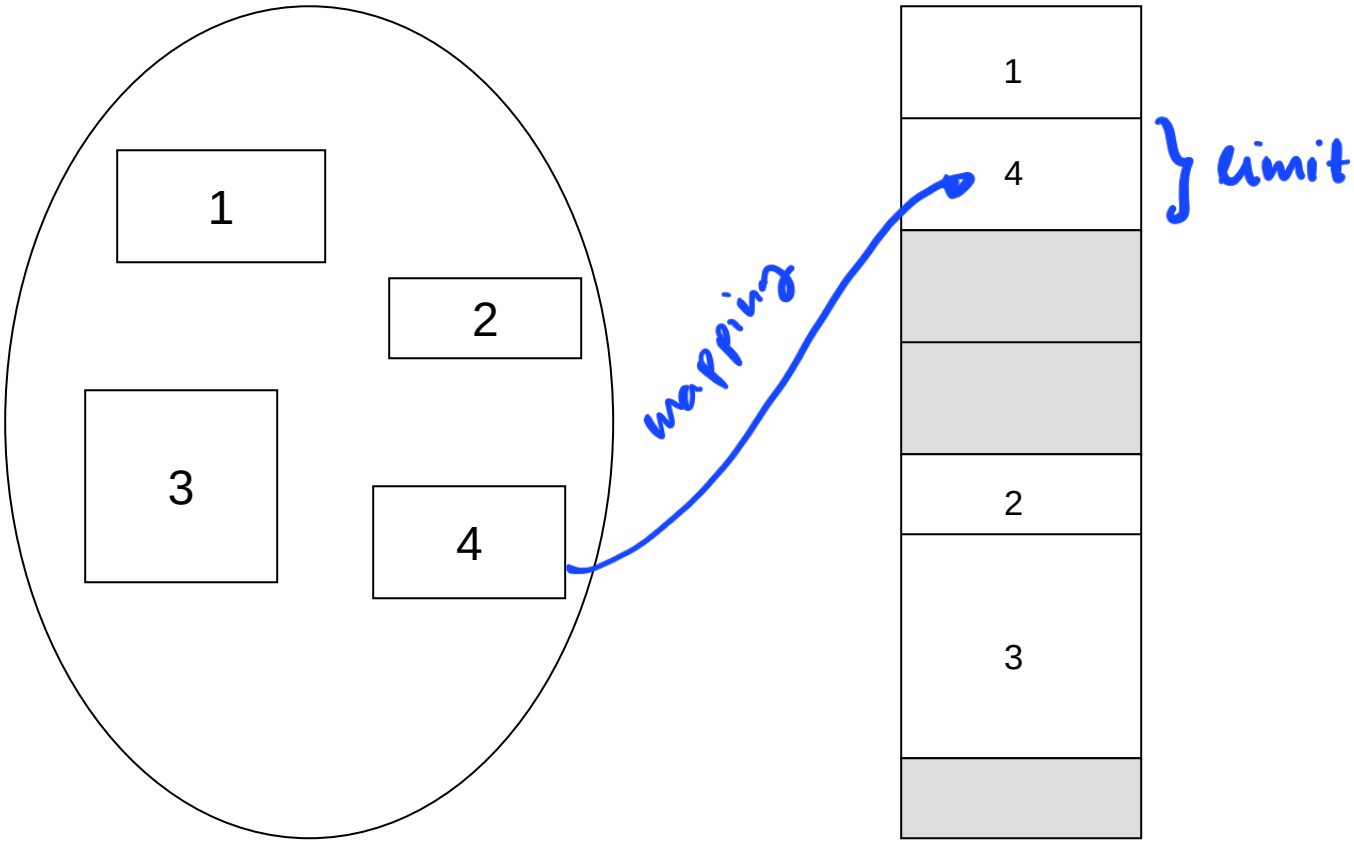
User's View of a Program



- Segments vary in length
- For efficiency, the logical split should respect **locality**
  - e.g., split code into pieces but keep together functions that call each others

Logical View of Segmentation

- Logical address space is a collection of segments



Segmentation Architecture

- Logical address consists of a two tuple: <segment-number, offset>
- **Segment table** – maps two-dimensional logical addresses into one-dim. physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside
  - **limit** – specifies the length of the segment

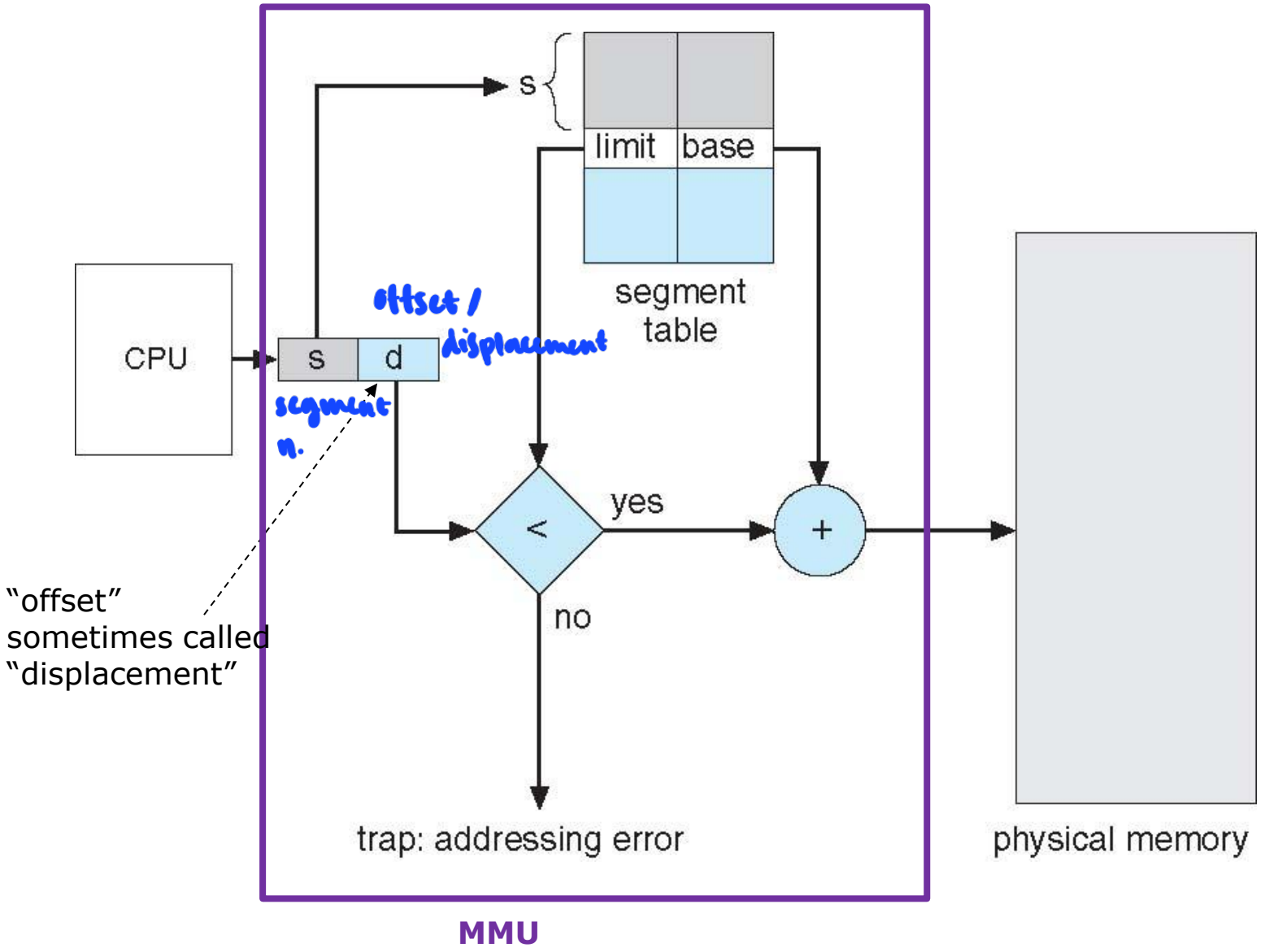
# S	Base	Limit

segment table

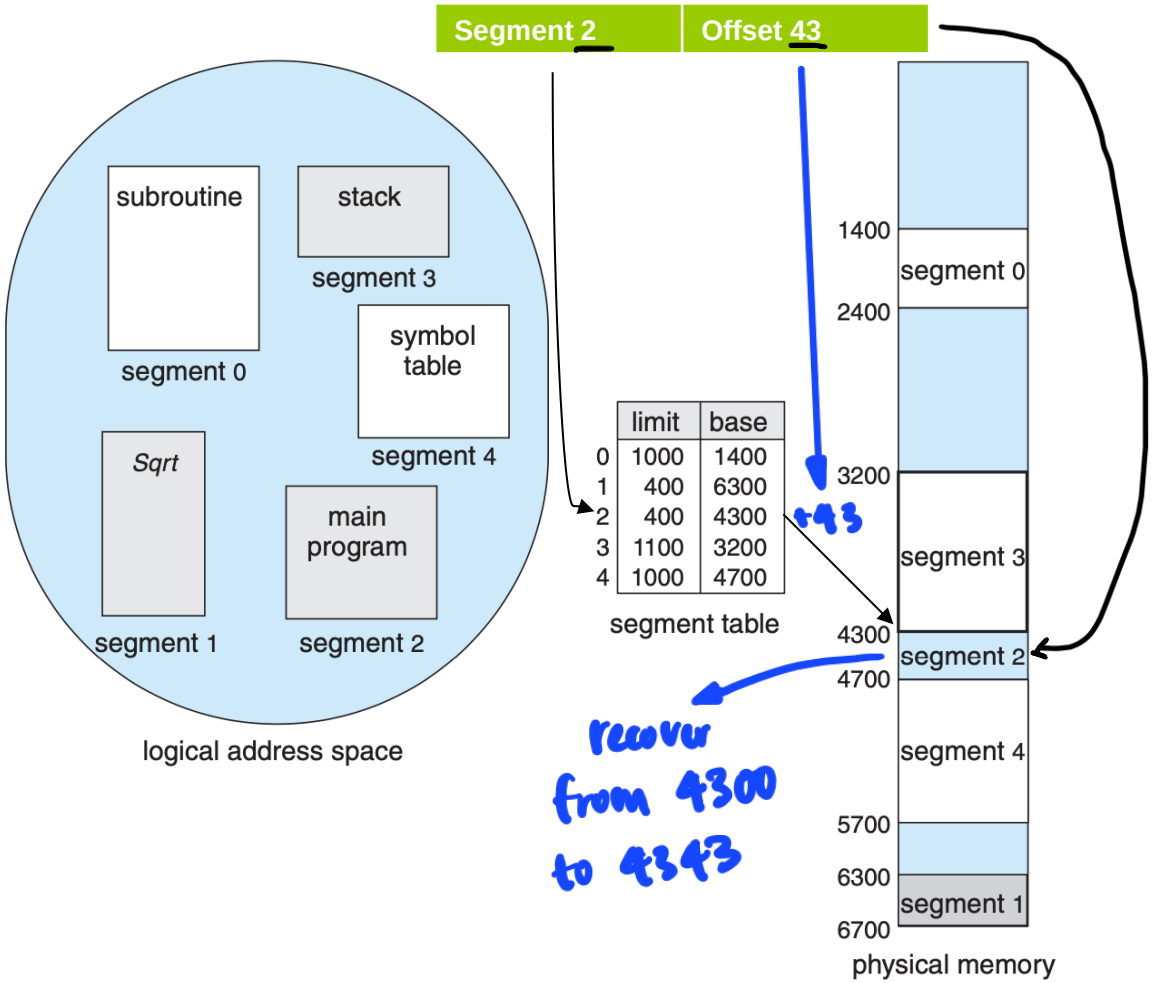
- Besides the space for storing the segment table we need:
  - **Segment-table base register (STBR)** points to the segment table's location in memory
  - **Segment-table length register (STLR)** indicates number of segments used by a program;
- Protection
  - segment number ***s*** is legal if ***s*** < **STLR**. Otherwise, **segmentation fault**
  - Also segmentation fault, if ***offset > limit***

outside dedicated segment

Segmentation Hardware



Segmentation example



Segmentation Advantages & Limitations

With respect to contiguous allocation:

- Smaller memory holes (even though does not solve fragmentation)
- Allows a finer-grain protection. Each segment table entry can be associated with:
  - validation bit = 0 ➔ illegal segment
  - read/write/execute privileges (e.g. code segment can be read and executed, but not modified)

we can set a segment with specific privileges

CONS

- Does not solve fragmentation

we need to keep updating the table

- To allocate a new process, need a *list of memory holes* (each entry keeps <base,limit>) and a research to find the first/best/worst one.

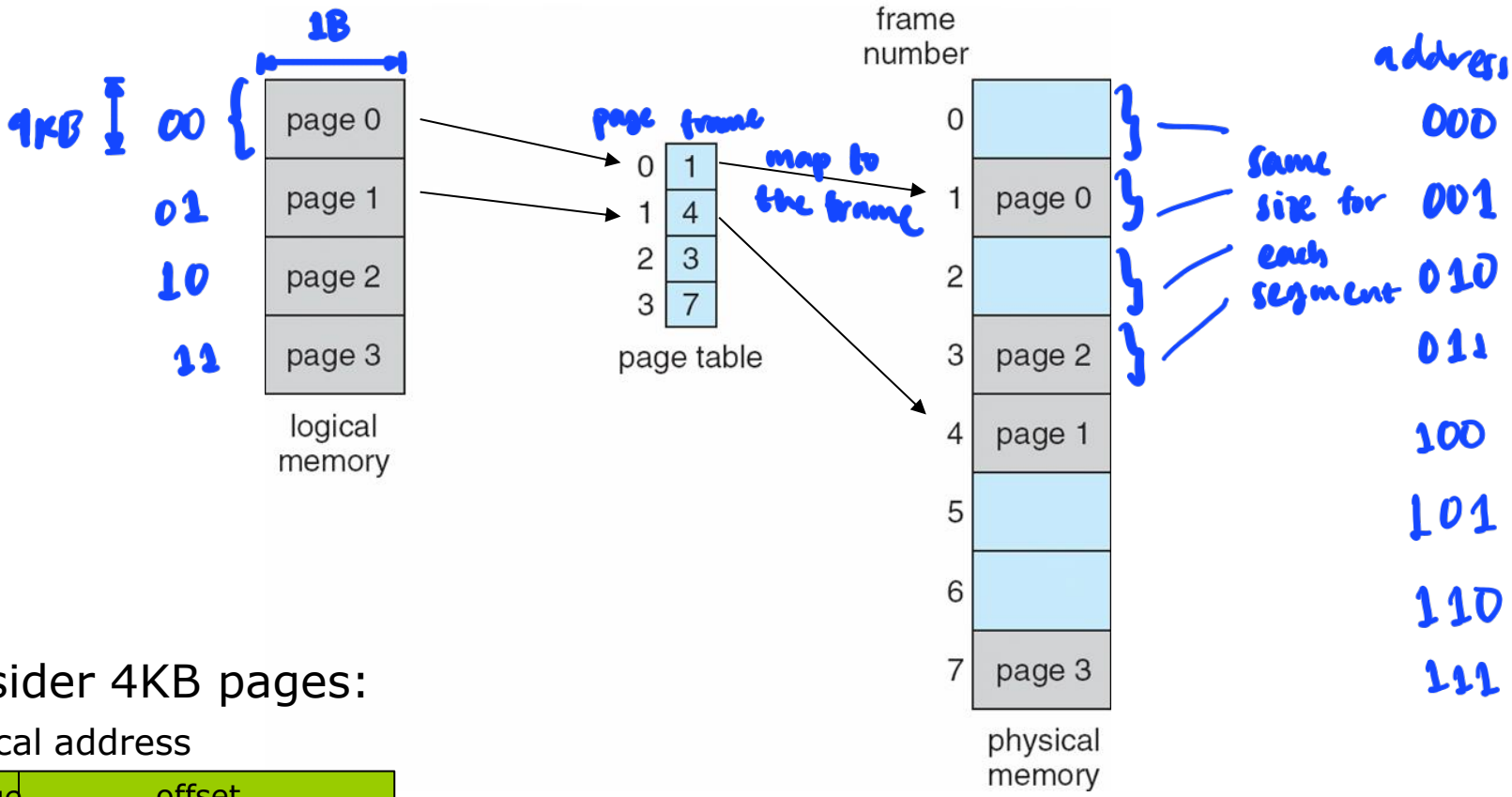
Paging

defined slots called pages and assign each process to avail pages.

- Divide physical memory into fixed-sized blocks called **frames**
  - size always a power of 2, from 512B to 16 MB
  - # of bits to address = log2 (size)
- Divide logical memory into blocks of same size called **pages**
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - No need to look for memory holes of the right size
- Keep track of all free frames, no need to keep track of their size
- To run a program of size ***N*** pages, just need to find *any* ***N*** free frames (any of them because they are all same size)
- Set up a **page table** for each process to translate logical to physical addresses
- Still have internal fragmentation (avg ½ page per process)

we only keep the hole (space) remaining in each page

Paging Model of Logical and Physical Memory



consider 4KB pages:

logical address

2bits 12bits 12 = log2 (4*1024)

physical address

3bits 12bits

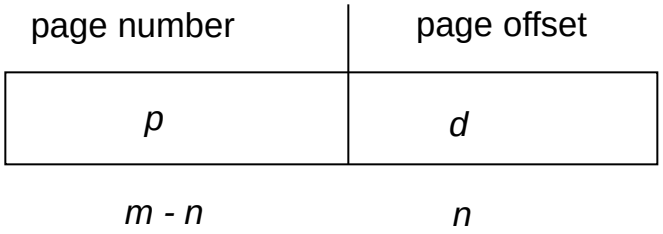
with a real-life RAM of dimension 16 GB = 2^{34}B

22bits 12bits

Address Translation Scheme

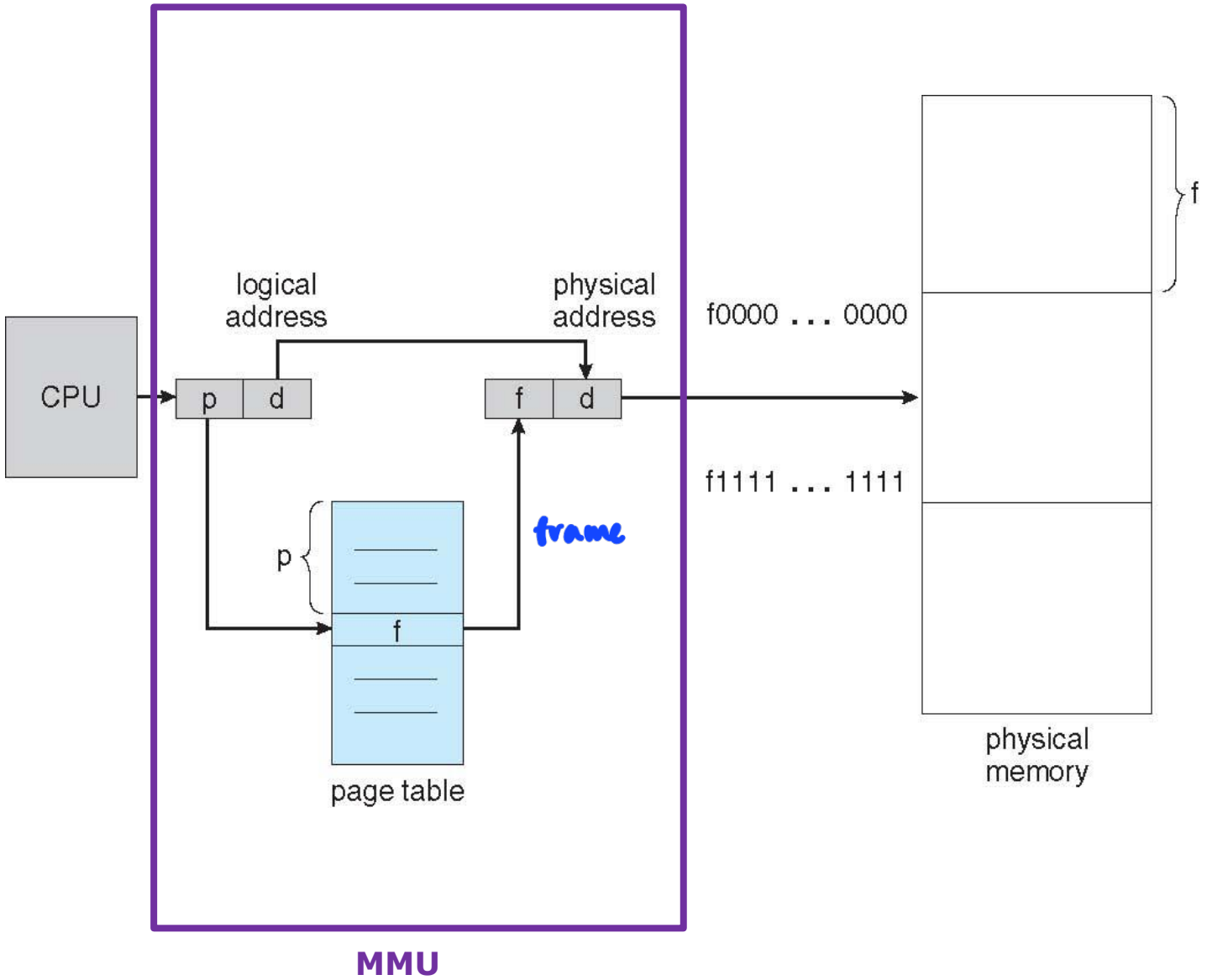
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

since we have fixed offset, we don't need to check the limit

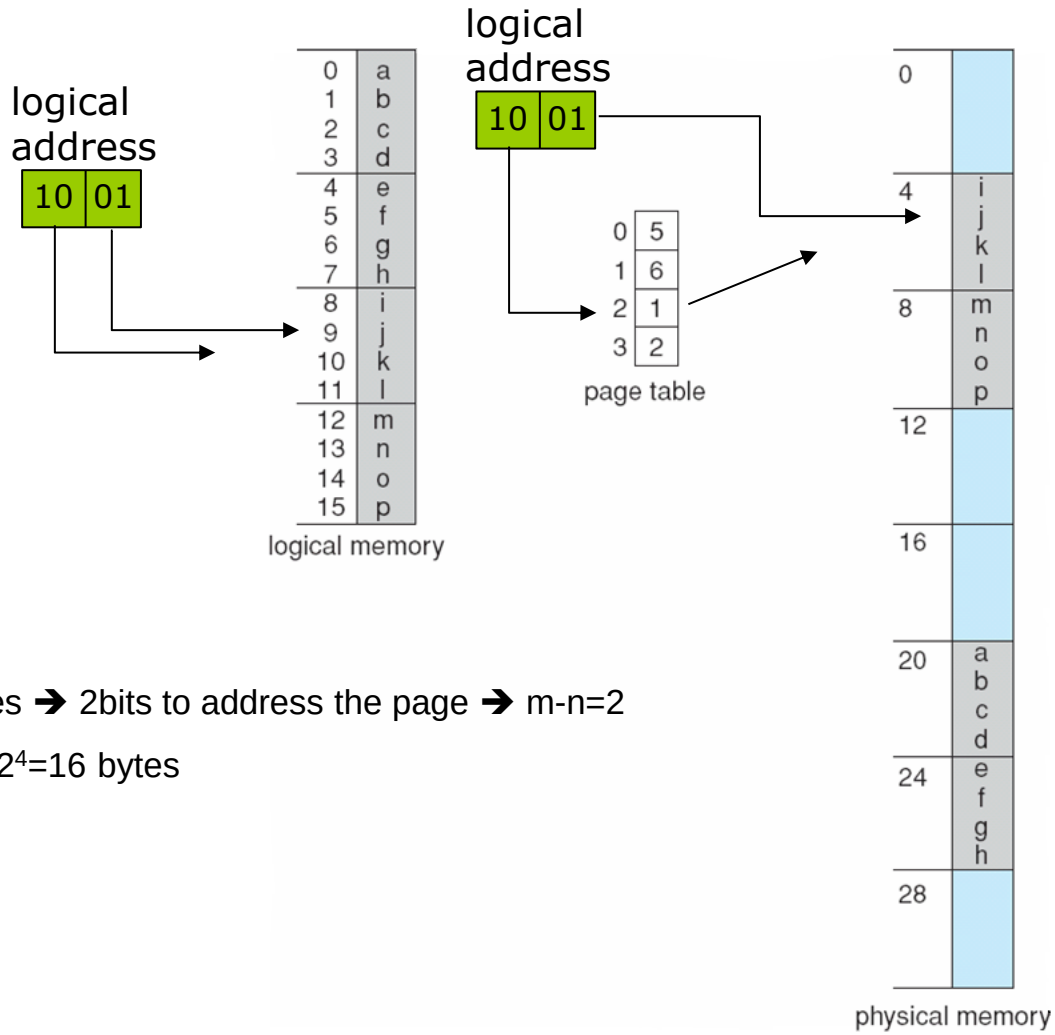


- For given logical address space 2^m and page size 2^n

Paging Hardware



Paging Example



4-byte pages ➔ *n*=2

process image needs 4 pages ➔ 2bits to address the page ➔ m-n=2

logical address space is $2^m=2^4=16$ bytes

physical address space is $2^5=32$ bytes

Paging (Cont.)

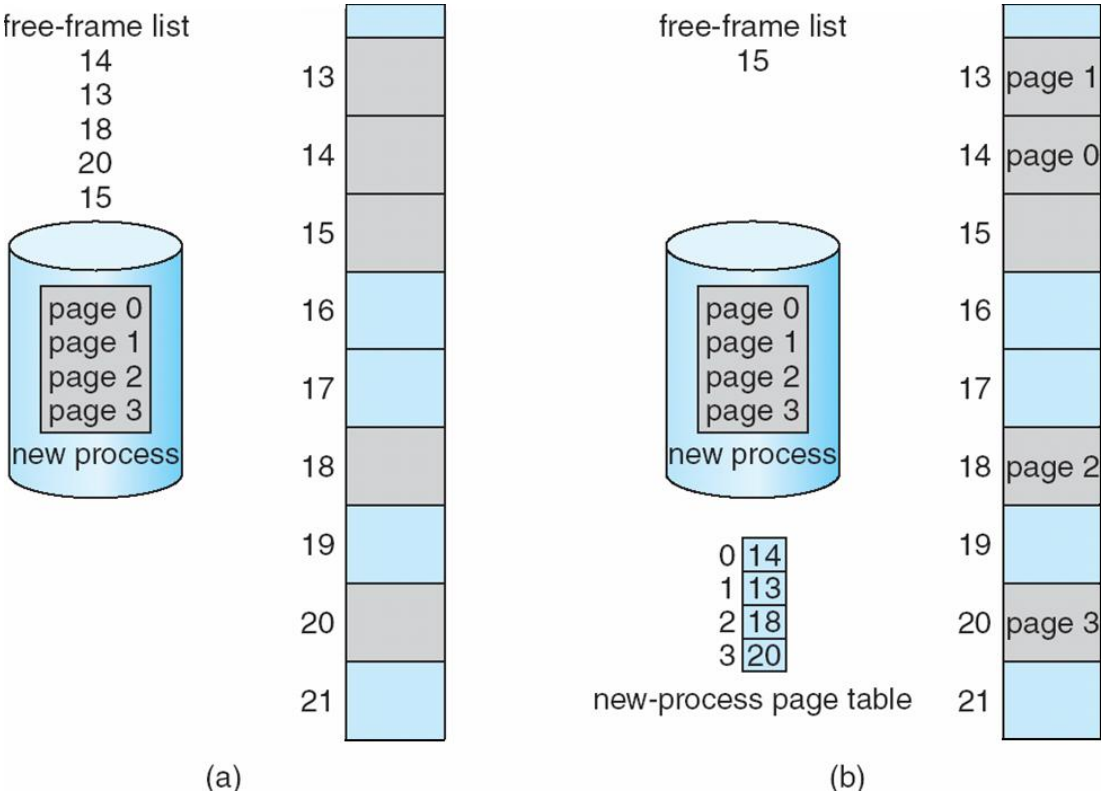
- Efficiency if:
  - frames are not too small.
    - Otherwise, each process image would be scattered in a multitude of noncontiguous frames
    - also, the page table becomes too big to be kept in memory
  - frames are not too big
    - otherwise, you have high internal fragmentation (on average internal fragmentation is restricted to half a page)

- Protection is obtained <u>by implementation</u>: a process can only access its own memory (it is impossible to map a logical address to a physical one outside the process memory image)

Free Frames (Frame Table)

How to find free space for a new process?

- no longer need list of memory holes and research of first/best/worst one

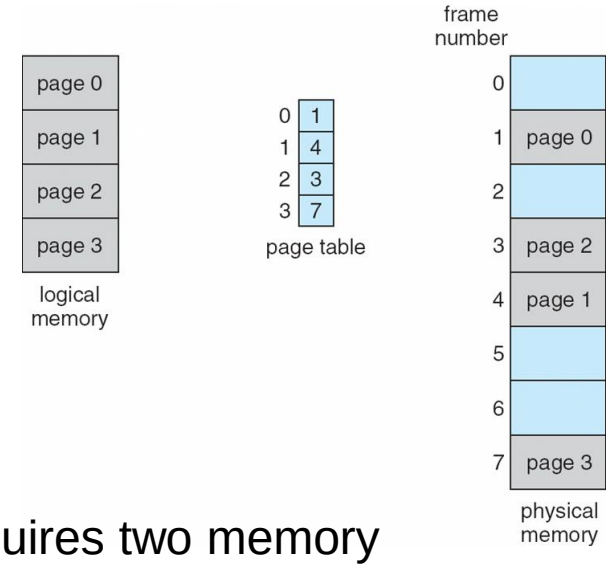


Before allocation

After allocation

Implementation of Page Table

- Page table of each active process is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table



- In this scheme every data/instruction access requires two memory accesses
  - One for the page table, and
  - one for the data / instruction

- No such problem before:
  - with contiguous allocation, a register is enough to keep the base register
  - with segmentation, need few registers (one for the base of each segment)

Virtual Memory

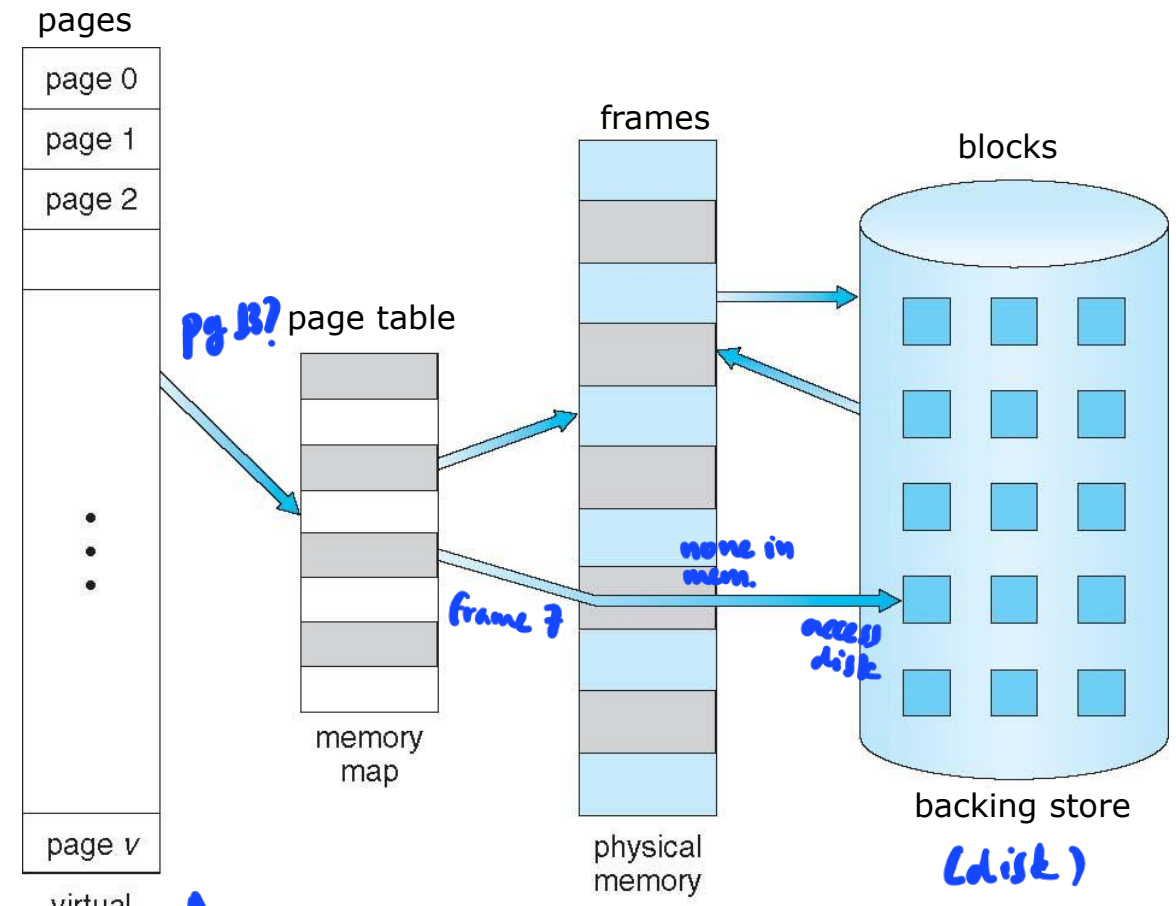
related to mem. fragmentation

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Same for data
- Entire program code and whole data not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program and data could be larger than physical memory

- **Virtual memory** – separation of user logical memory from physical memory

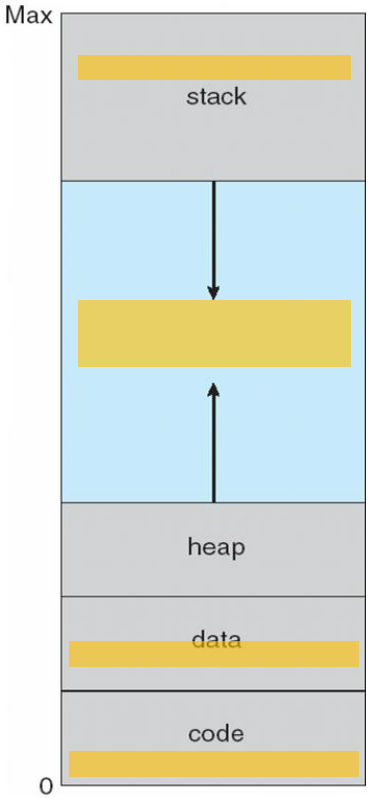
Memory Larger Than Physical Memory

- Only part of the program needs to be in memory for execution



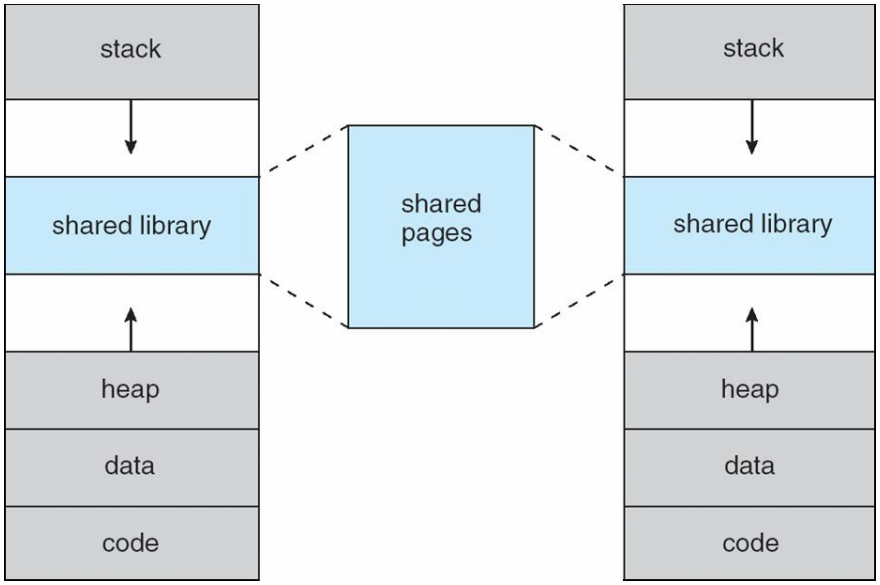
Virtual Address Space

- Virtual memory is useful because logical address space is usually **sparse**: relevant information is only in a small part of the logical address space.
- For example:
  - some free space is needed for stack and heap to grow
  - data may contain irrelevant information (matrix with 0-entries)
  - error code, unusual routines



Virtual Address Space

- Virtual memory also useful to enable shared system libraries



- Analogously, shared memory by mapping pages read-write into virtual address space

Virtual Memory

- **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
    - Easier to program (no need to care about the portion of memory a program will occupy)
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
    - More CPU utilization
- Implementation of virtual memory by **demand paging**

Demand Paging

- Bring a page into memory only when it is needed
- Whenever an instruction creates a reference to another page:
  - if it is in memory ➔ ok
  - if not ➔ bring it to memory
- Also called **lazy swapping**

we only put pages that ACTUALLY needed.