Memory Management

Memory Management

Learnt : So Far

- We showed how the CPU can be shared by a set of processes.
- As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users.
- To realize this increase in performance, however, we must keep several processes in memory; that is, we must share memory.

The mernory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).

Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Segmentation

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

Background

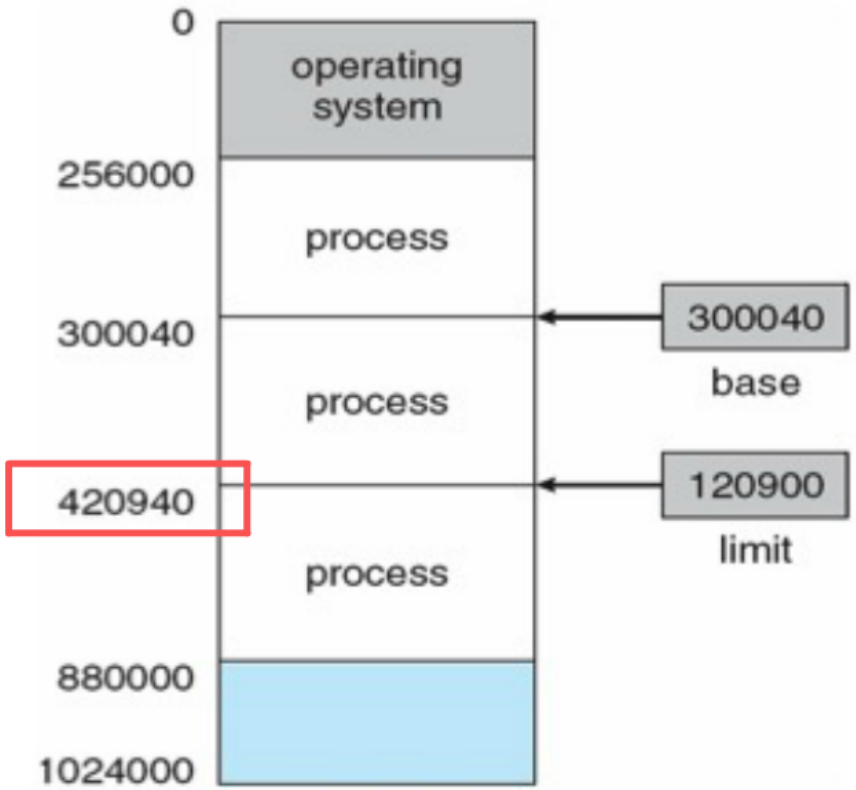
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
- Each process has a separate memory space

  - determine the range of legal addresses that the process may access

  - ensure that the process access only these addresses.

Base and Limit Registers

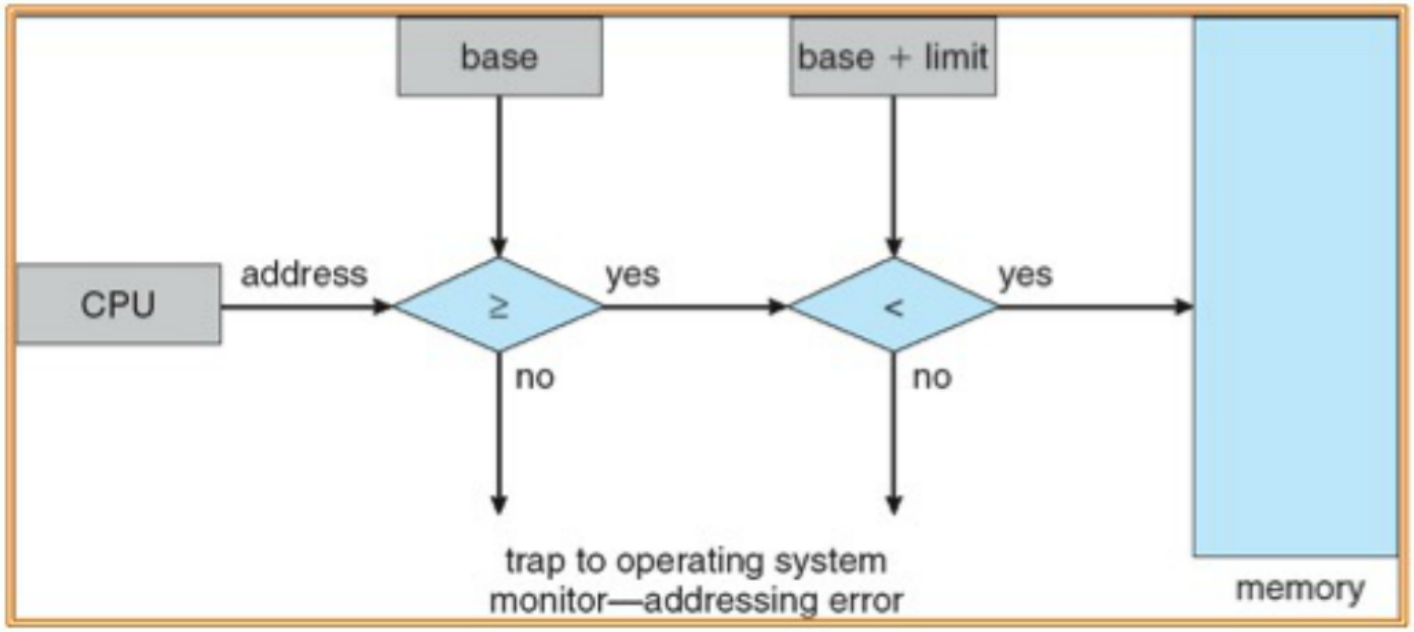
- A pair of **base** and **limit** registers define the logical address space
- **Base register:** holds the smallest legal physical memory address.
- **Limit register:** specifies the size of the range.



HW address protection with base and limit registers

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error



Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.

Binding of Instructions and Data to Memory

Address Binding

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

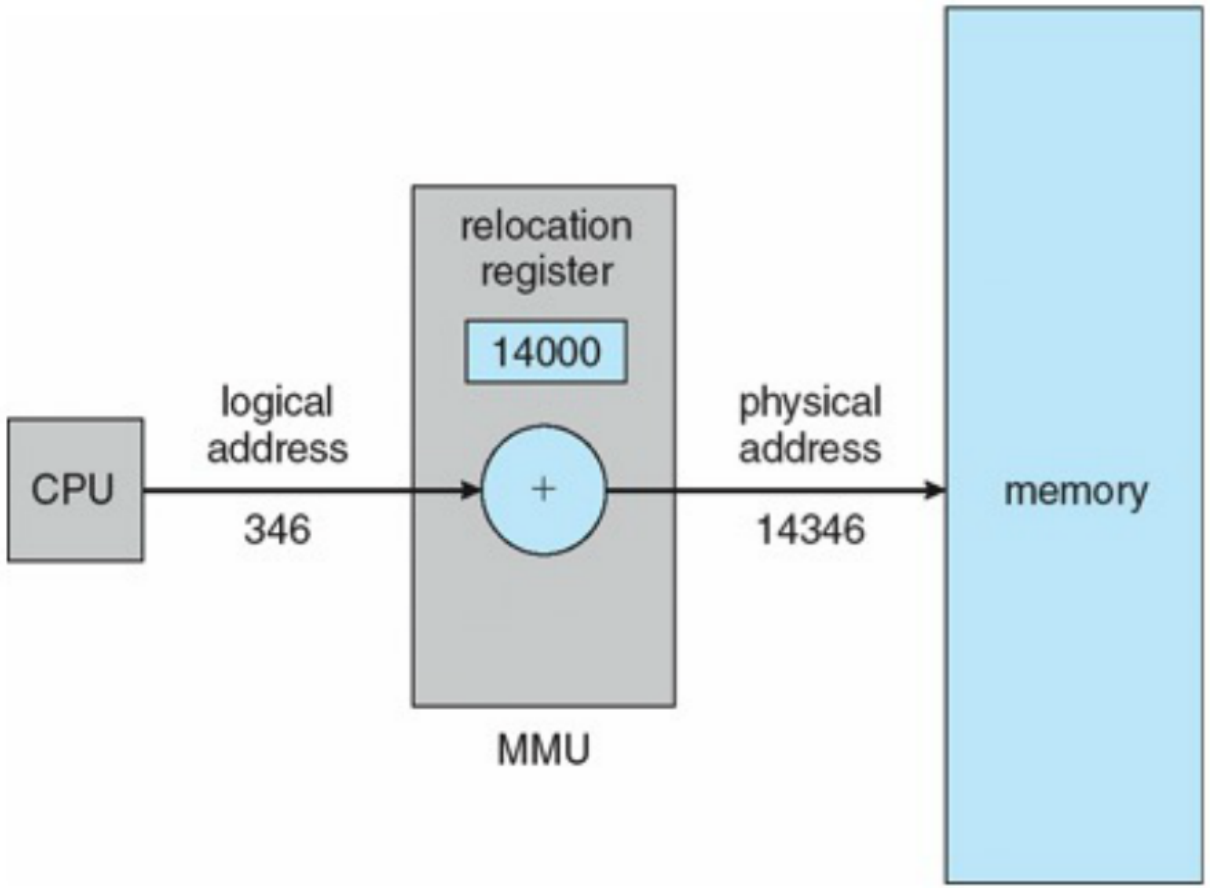
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register

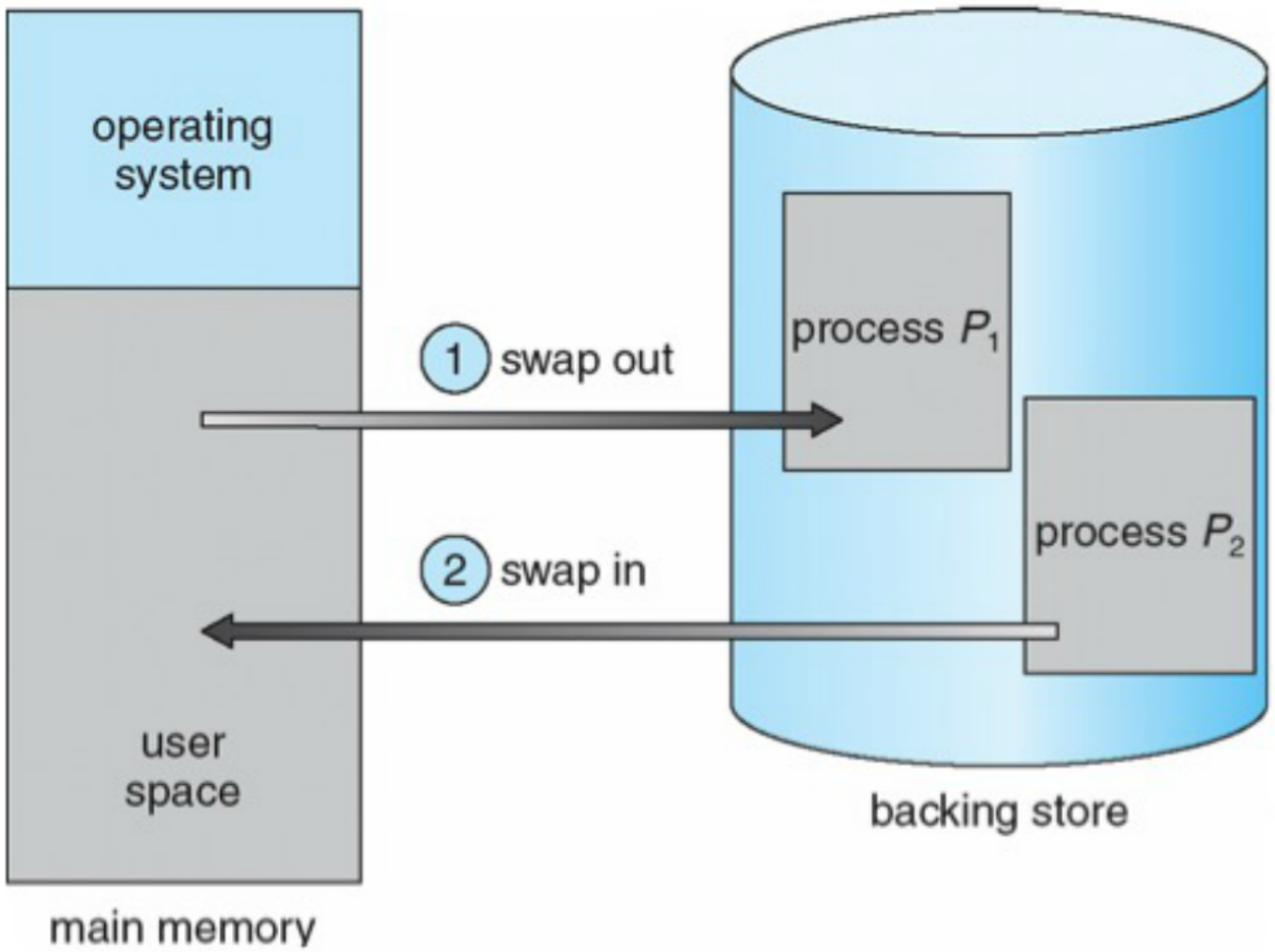


Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding.

Schematic View of Swapping



Swap Time - Calculate!

To get an idea of the context-switch time, let us assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50MB per second. The actual transfer of the 100-MB process to or from main memory takes

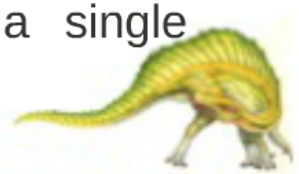
100MB/50MB per second= 2 seconds.

Assuming an average latency of 8 milliseconds, the swap time is 2008 milliseconds. Since we must both swap out and swap in, the total swap time is about 4016 milliseconds.

Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory.

In. contiguous memory allocation, each process is contained in a single contiguous section of memory.



Partitioning Schemes

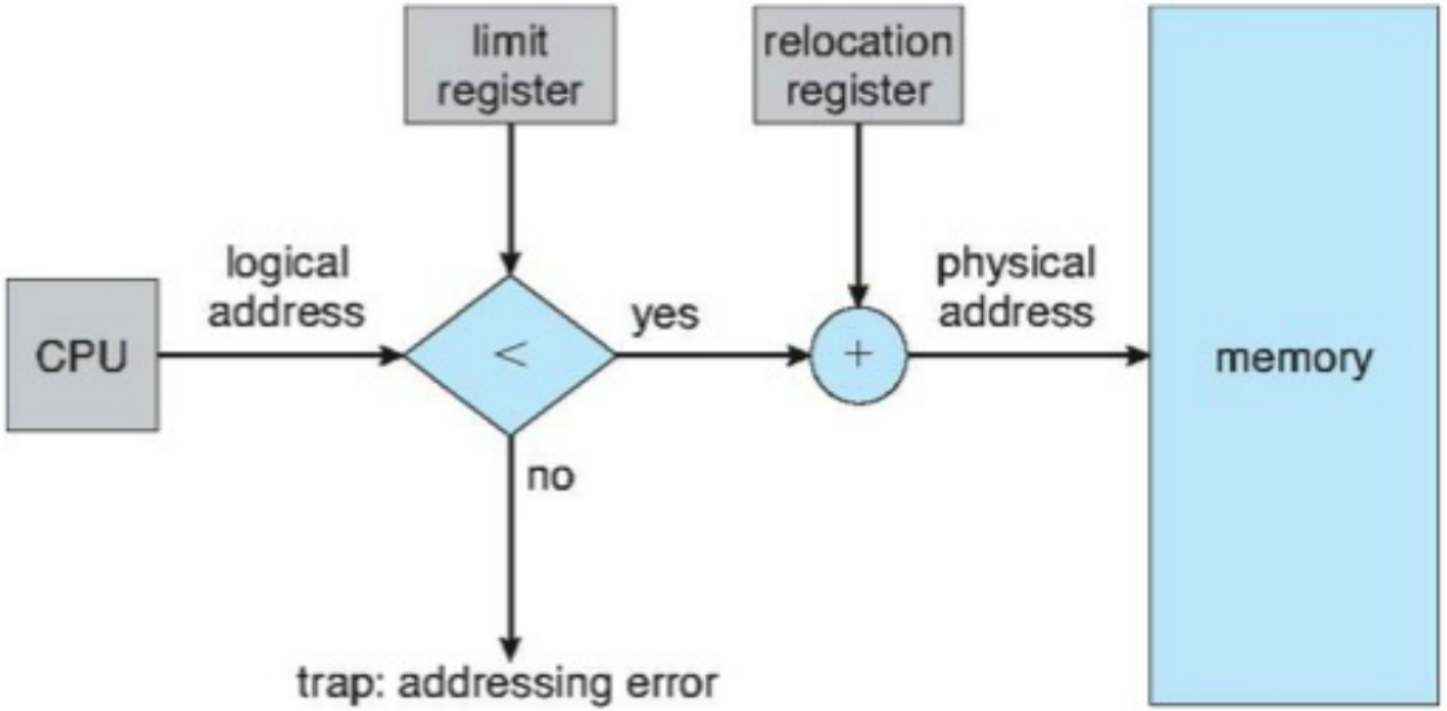
One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**. Each partition may contain exactly one process.

In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory a **hole**.

Dynamic Memory Allocation: In generat as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

Paging

Hardware Support for Relocation and Limit Registers



Dynamic Storage-Allocation Problem

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

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

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

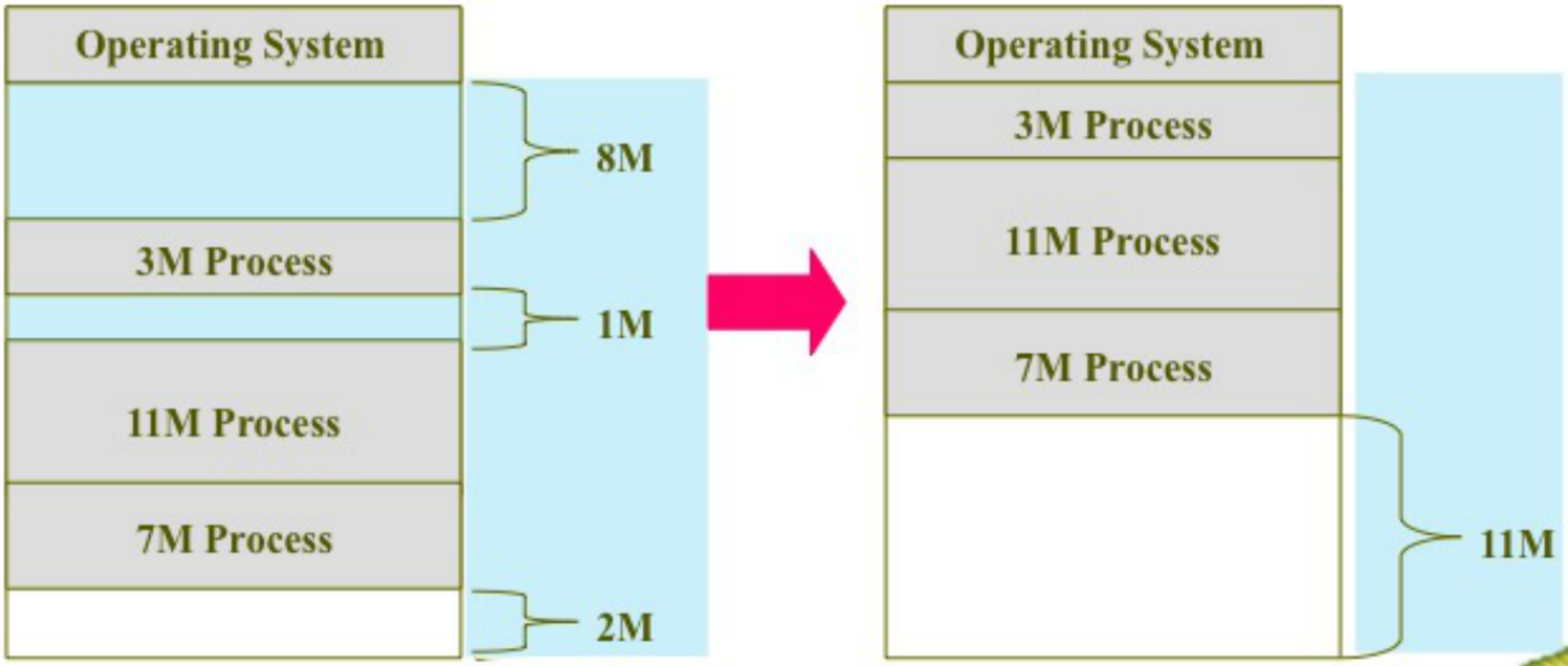
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- 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

External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes.

Compaction

- Eliminate holes by moving processes
  - Copy operation is expensive

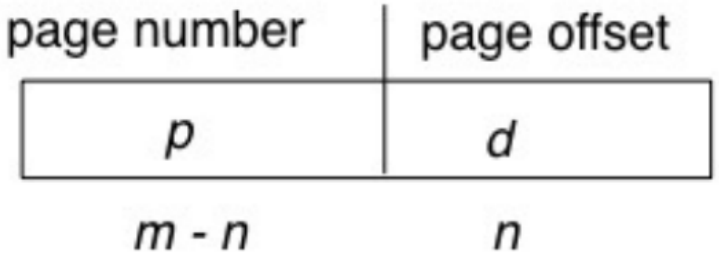


Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

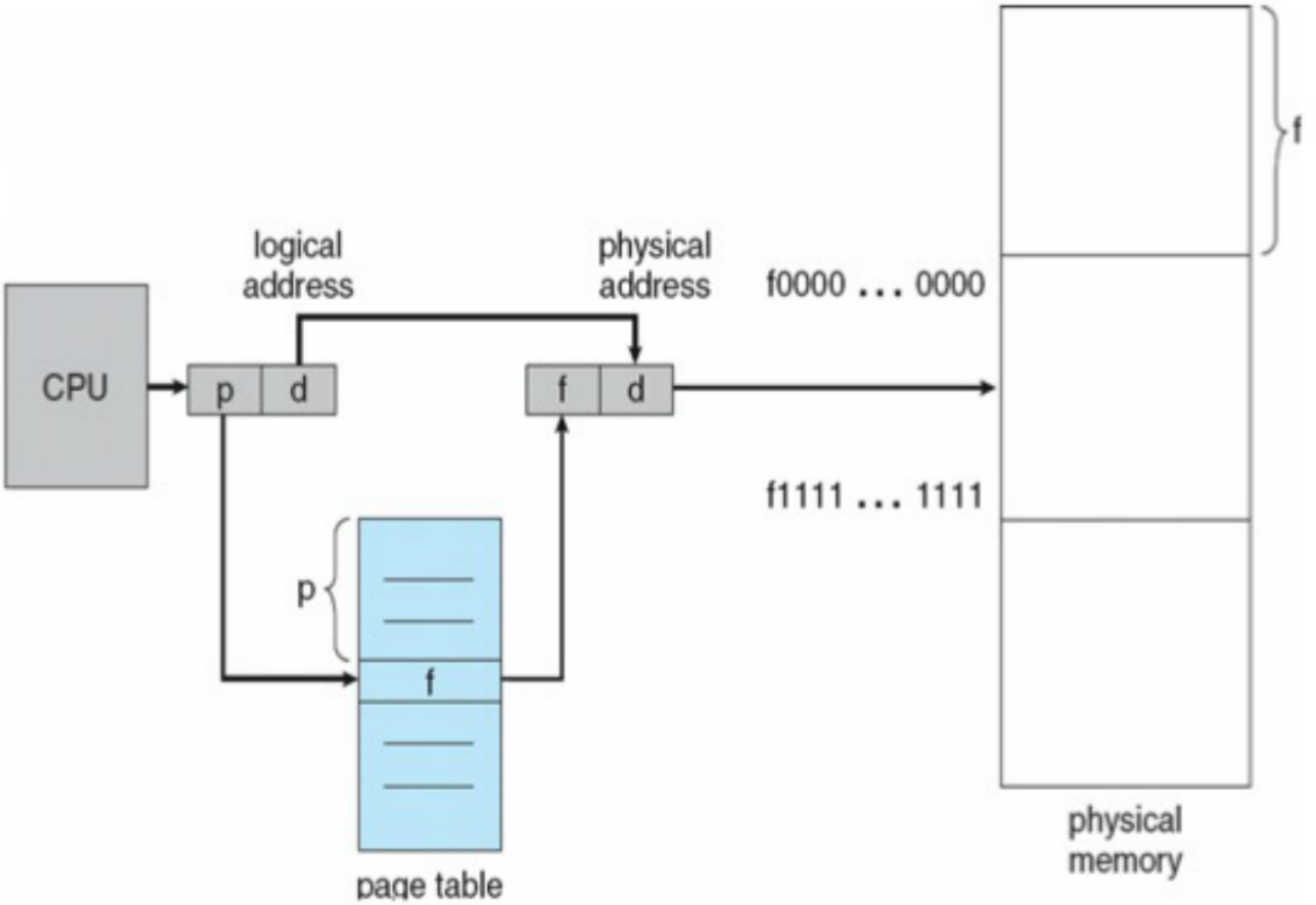
Address Translation Scheme

- 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

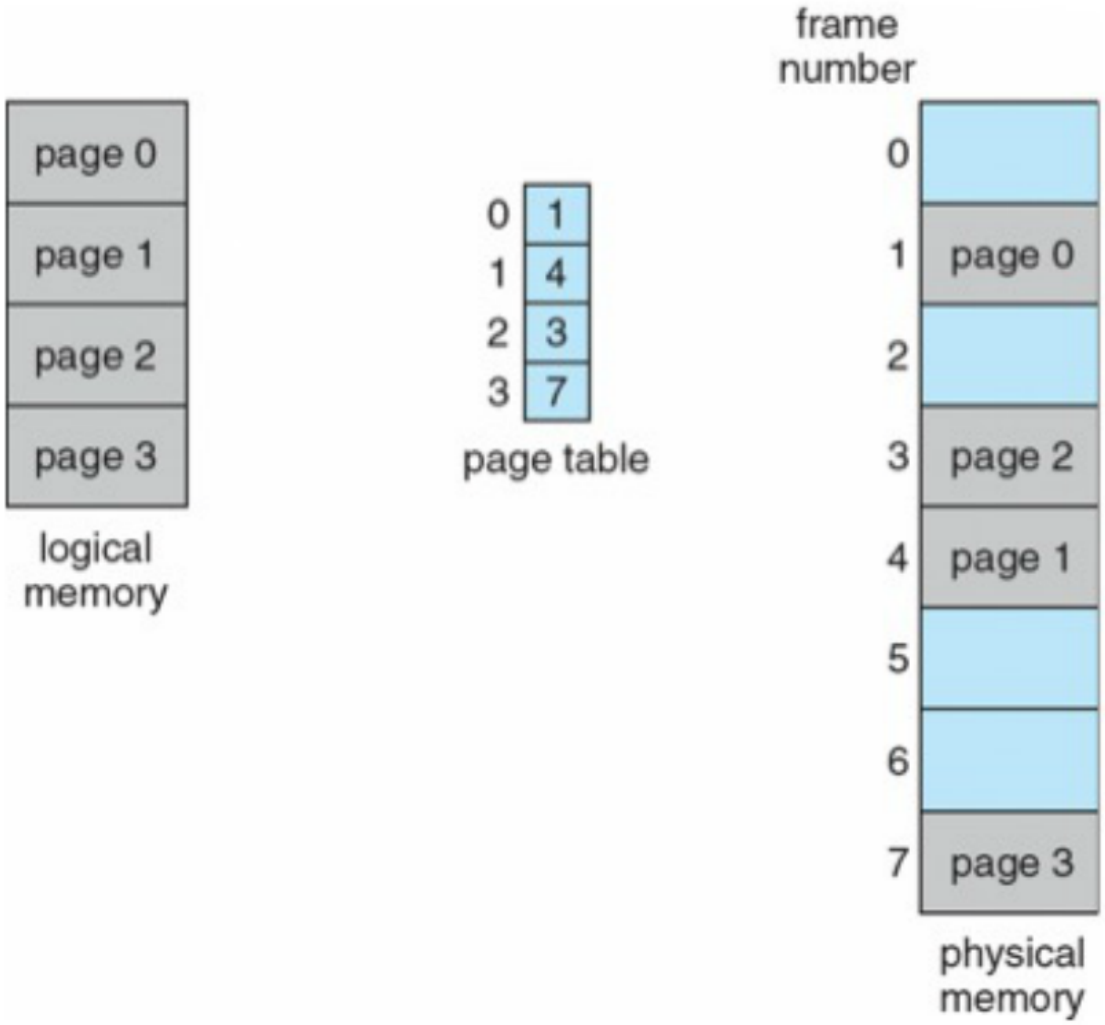


  - For given logical address space 2^m *and page size* 2^n

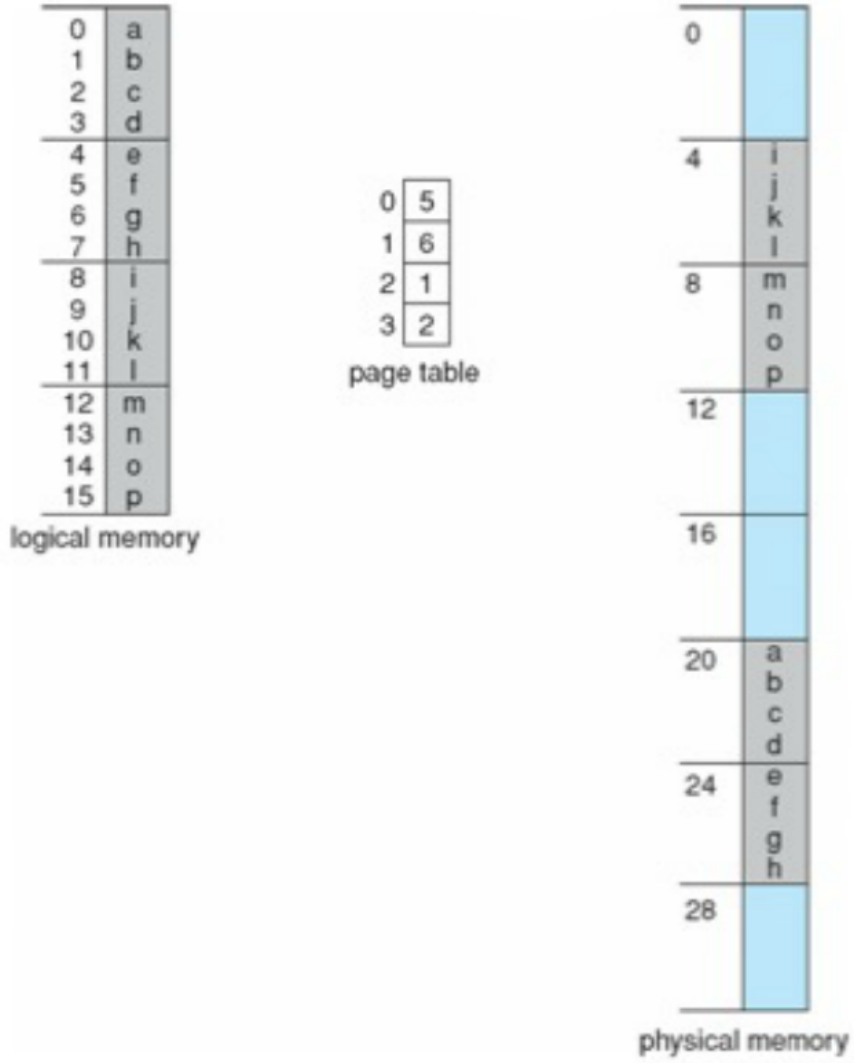
Paging Hardware



Paging Model of Logical and Physical Memory

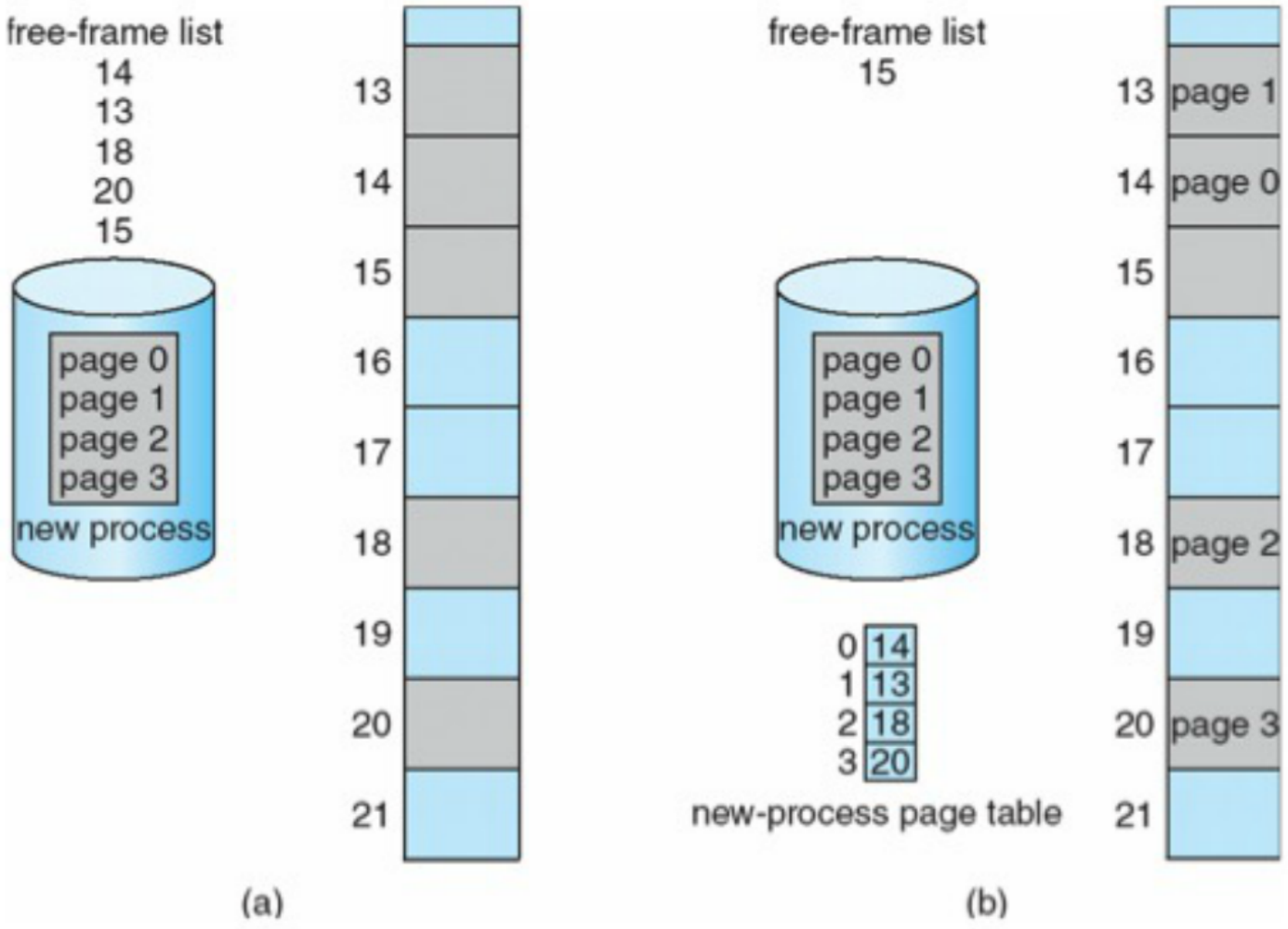


Paging Example



32-byte memory and 4-byte pages

Free Frames



Before allocation

After allocation

End of 22nd Sept

Paging

- Paging is a memory-management scheme that permits the physical address space a process to be non-contiguous.
- Paging avoids external fragmentation and the need for compaction.
- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem.
- Paging in its various forms is used in most operating systems

Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

Implementation of Page Table

Each operating system has its own methods for storing page tables. Most allocate a page table for each process.

Option 1

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient.

Option 2

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). Rather, the page table is kept in main memory, and a **Page-Table-Base-Register (PTBR)** points to the page table.

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction. **frame: page: offset**
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)** : **Virtual Memory**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

It provides us with the frame number, which is combined with the page offset to produce the actual address. With this scheme memory access is slowed by a factor of 2.

Associative Memory

- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

- If p is in associative register, get frame # out
- Otherwise get frame # from page table in memory

TLB miss and Hit ratio

- **TLB miss**:
  - If the page number is not in the TLB, a memory reference to the page table must be made
- **Hit ratio**:
  - percentage of times that a page number is found in the TLB.
- For example:
  - Assume TLB search takes 20ns; memory access takes 100ns
  - TLB hit → 1 memory access; TLB miss → 2 memory accesses

Effective Access Time (EAT)

- If Hit ratio = 80%
  - EAT = (20 + 100) * 0.8 + (20 + 200) ∗ 0.2 = 140ns
- If Hit ratio = 98%
  - EAT = (20 + 100) * 0.98 + (20 + 200) ∗ 0.02 = 122ns

Calculate: If we have 1GB (1000000 KB) memory space and typical page size is 100 KB what is the size of the TLB we have to have?

10K X 2 X 2= 40K

Issues with TLB

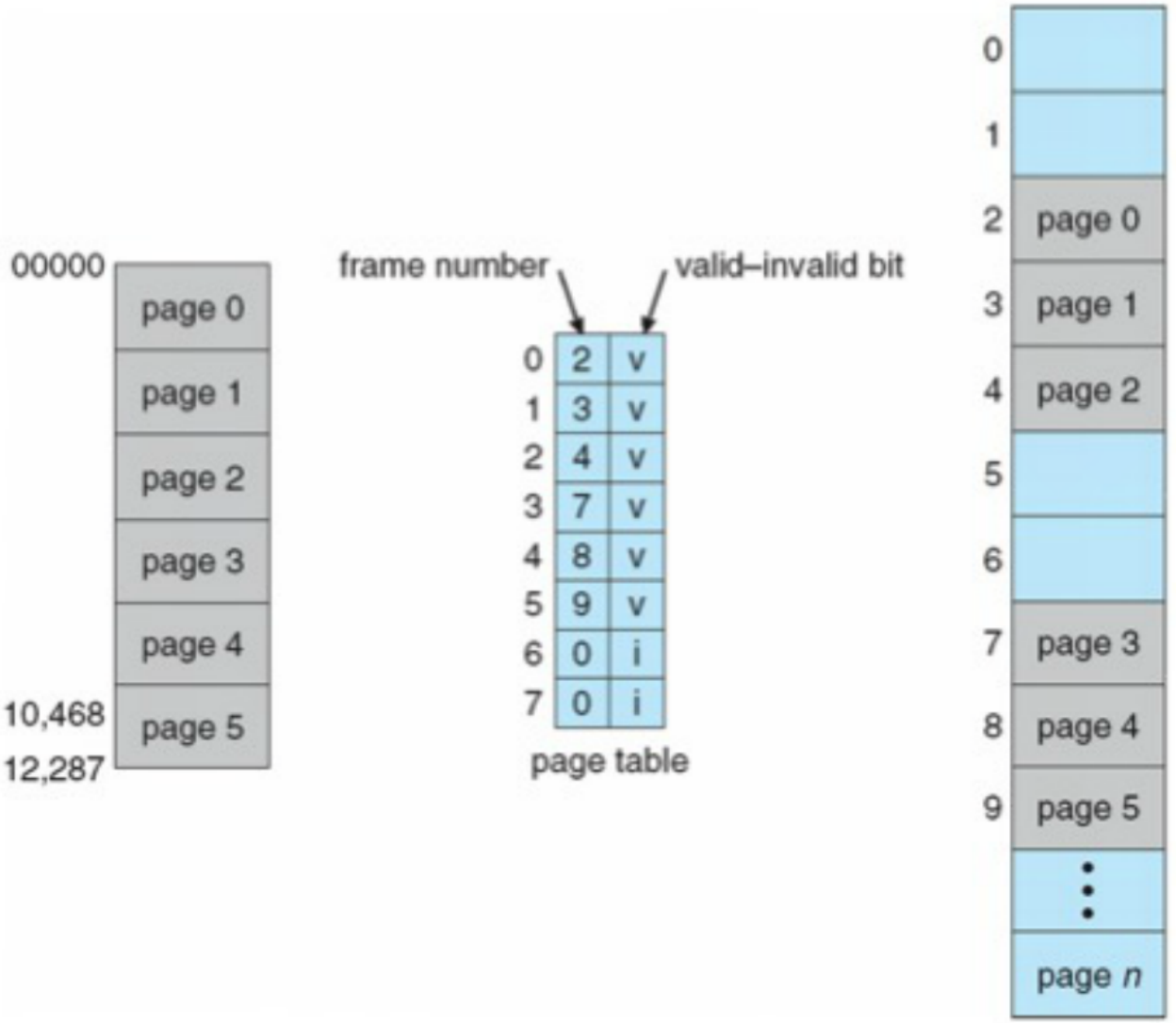
If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random.

Some TLBs store Address-Space-Identifiers (ASIDs) in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.

Memory Protection

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

Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages: Example

- Consider a system that supports 40 users, each of whom executes a text editor.
- If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- Main code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.
- Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user.
- The total space required is now 2,150 KB instead of 8,000 KB-a significant savings.

end of 29th Sept

Segmentation

- Memory-management scheme that supports user view of 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

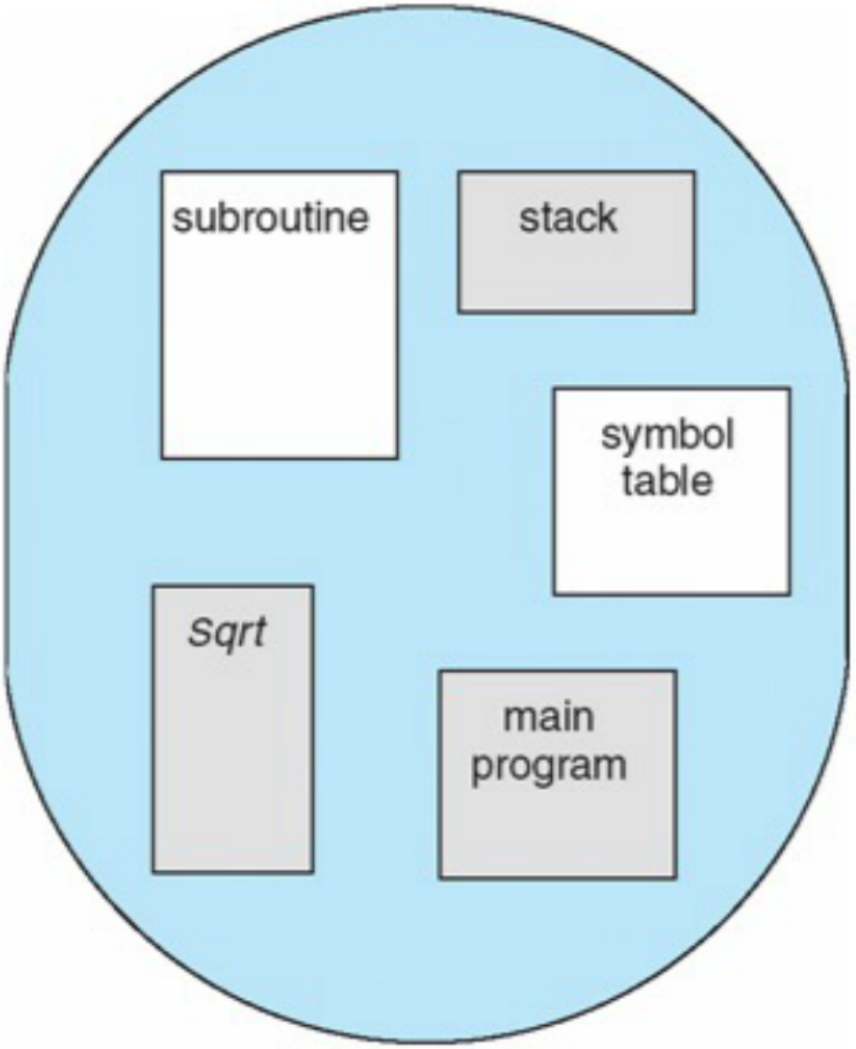
    common block

    stack

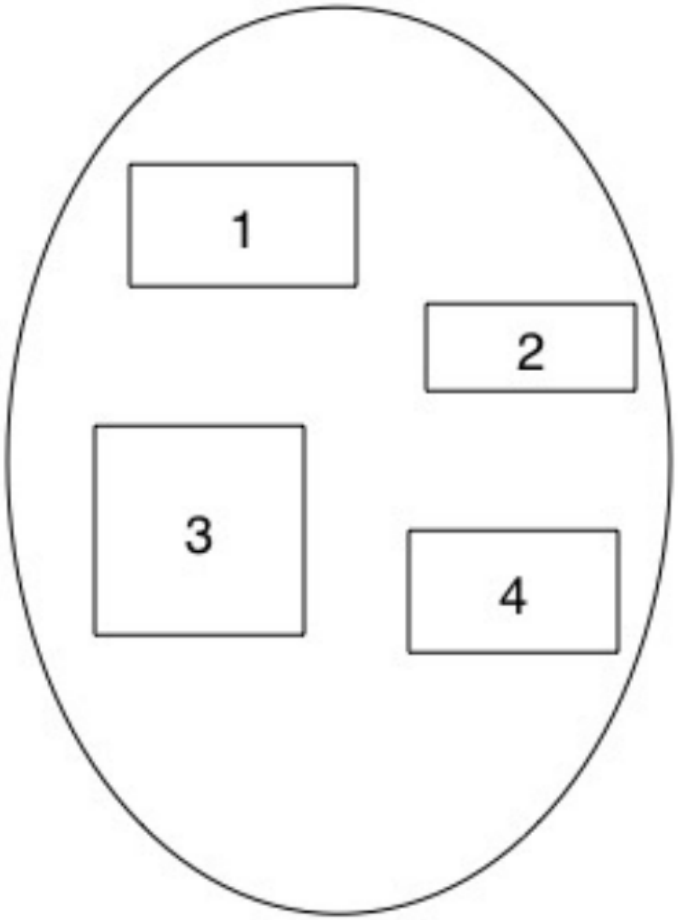
    symbol table

    arrays

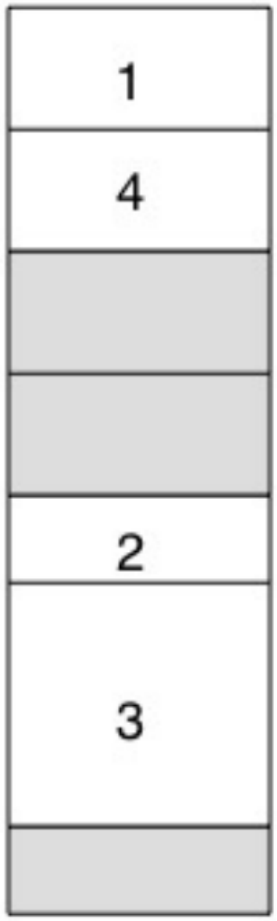
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

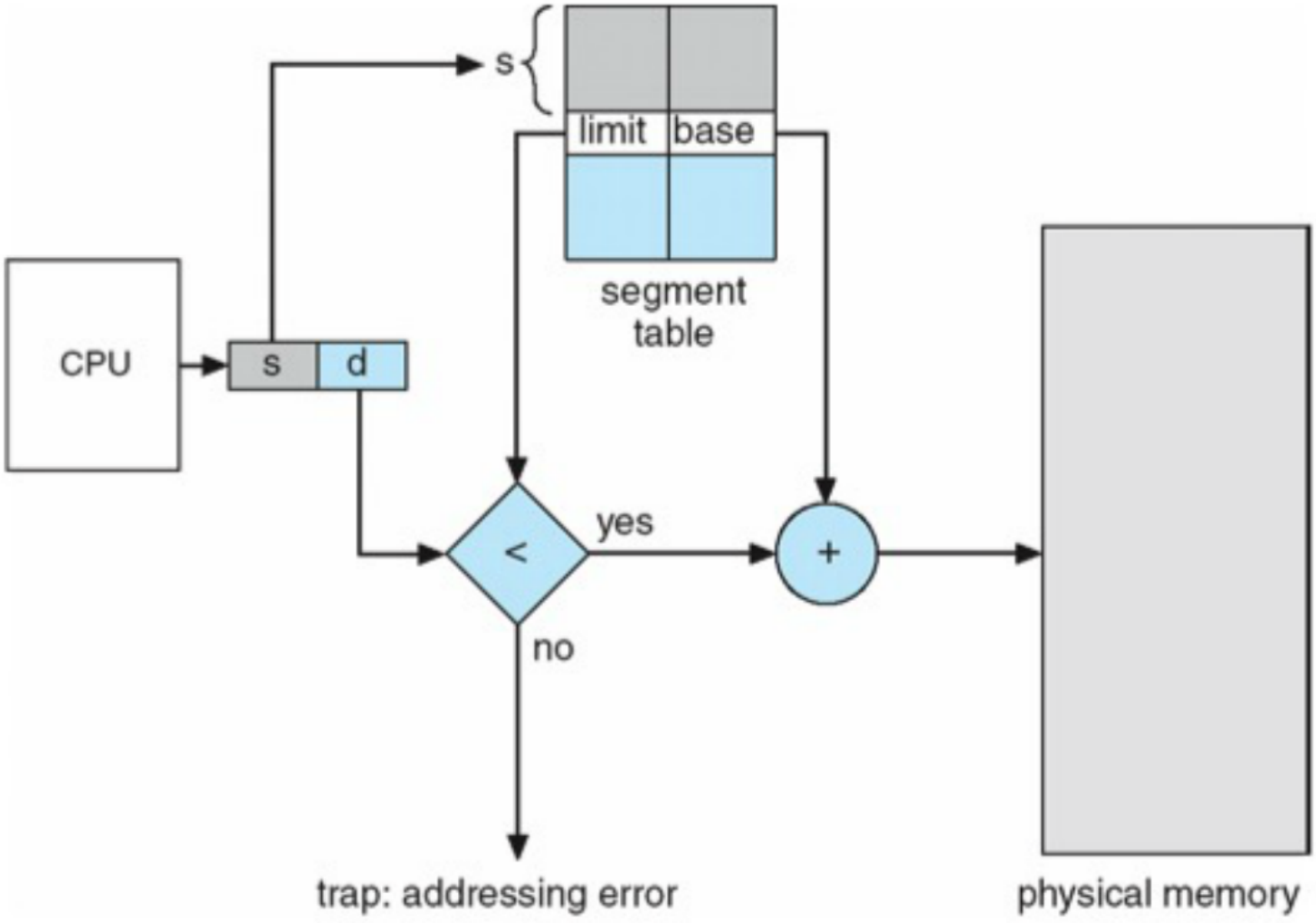
- Logical address consists of a two tuple:

  <segment-number, offset>,

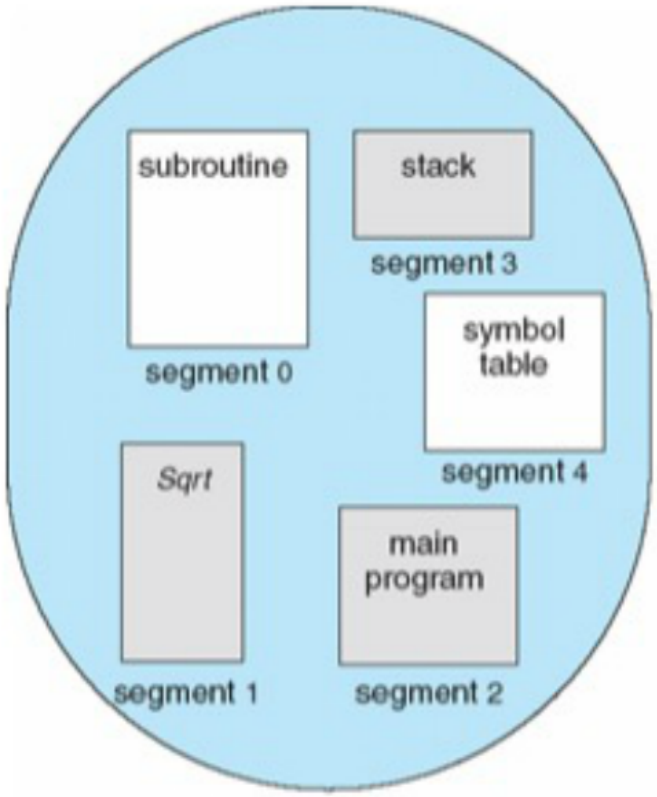
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

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

Segmentation Hardware

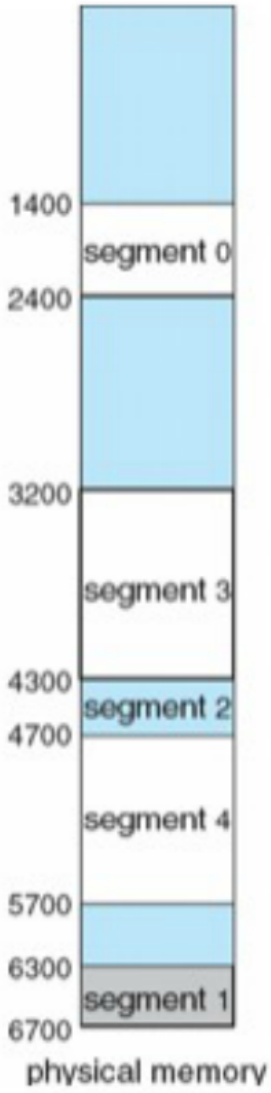


Example of Segmentation



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

segment table



Segmentation Architecture (Cont.)

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

Sharing of Segments