Computer and Operating Systems (COS)

Lecture 11

Overview of the contents

- **Contiguous memory allocation**
- **Paging**
- **Structure of the page table**
- **Swapping**

Memory management

When many processes are executed in the same system, they access the memory to:

- **Fetch instructions - in connection with program execution**
- **Save data - as a result of calculations**
- **Retrieve data - which must be included in new calculations or something else…**

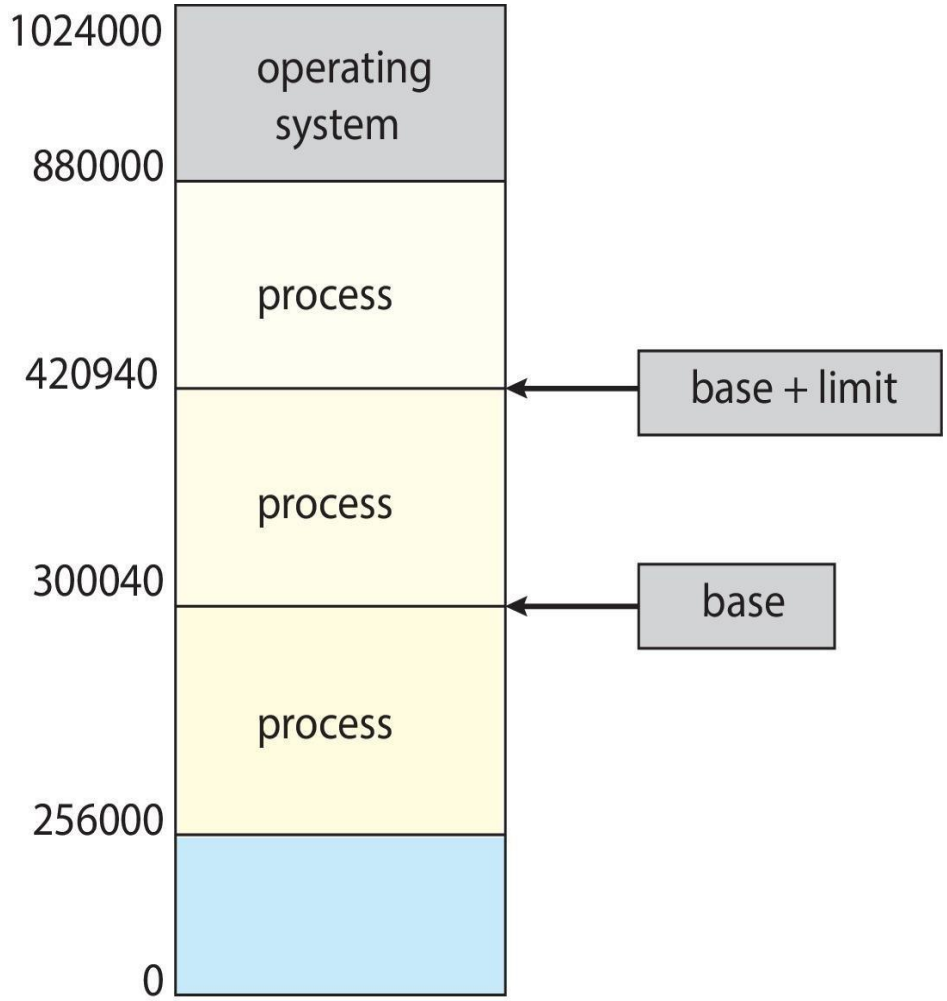
Which means processes must share memory. We will learn here memory management techniques.

Basic hardware

We must protect the operating system from access by user processes, as well as protect user processes from one another. This protection must be provided by the hardware.

Each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.

To separate memory spaces, The **base register** holds the smallest legal physical memory address while the **limit register** specifies the size of the range.

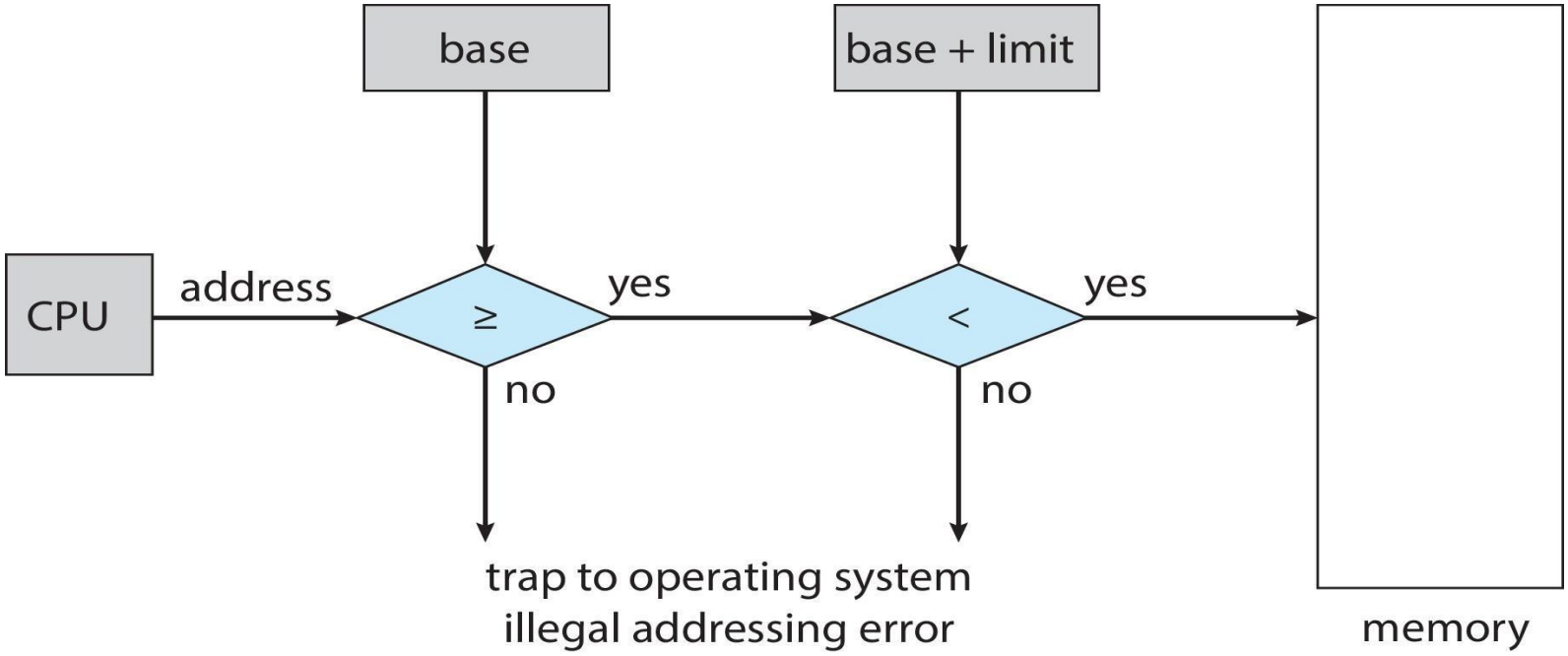


Basic hardware

Protection of memory space is accomplished by having the CPU hardware compare every generated address 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.

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.



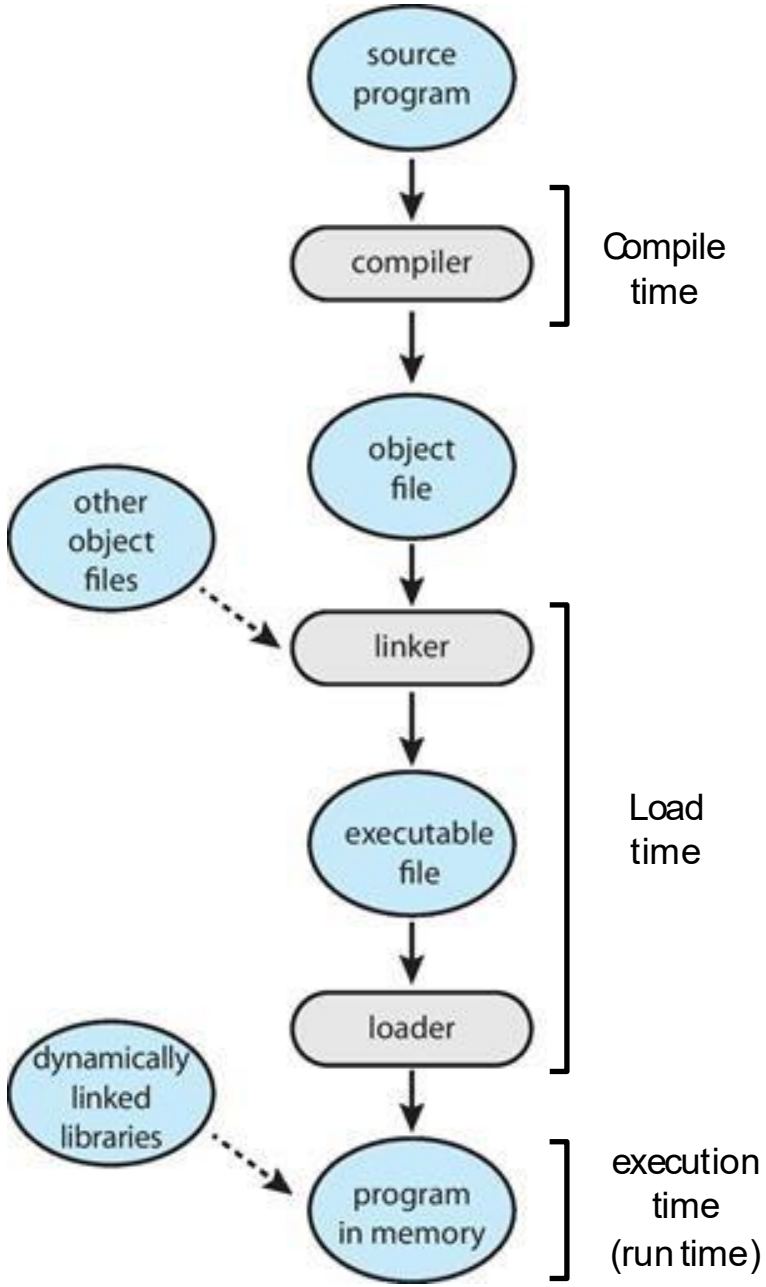
Address binding

To run, the program must be brought into memory and placed within the context of a process.

Most systems allow a user process to reside in any part of the physical memory.

The binding of processes' instructions and data to memory addresses can be done at any step along the way:

- ***Compile time***: If you know at compile time where the process will reside in memory, then absolute code can be generated.
- ***Load time***: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. i.e., final binding is delayed until load time.
- ***Execution time***: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

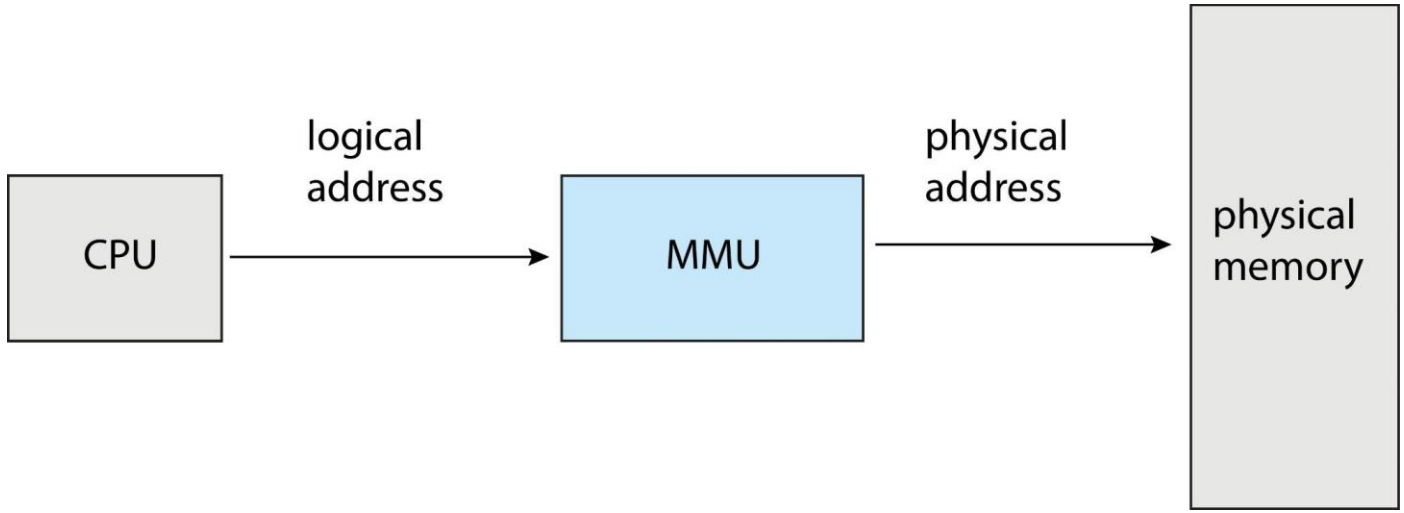


Logical / Physical address space

Logical address: An address generated by the CPU.
Physical address: An address loaded into the memory-address register of the memory.
The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**.

Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in different logical and physical addresses.

The run-time mapping from logical to physical addresses is done by a hardware device called the **memory-management unit (MMU).**

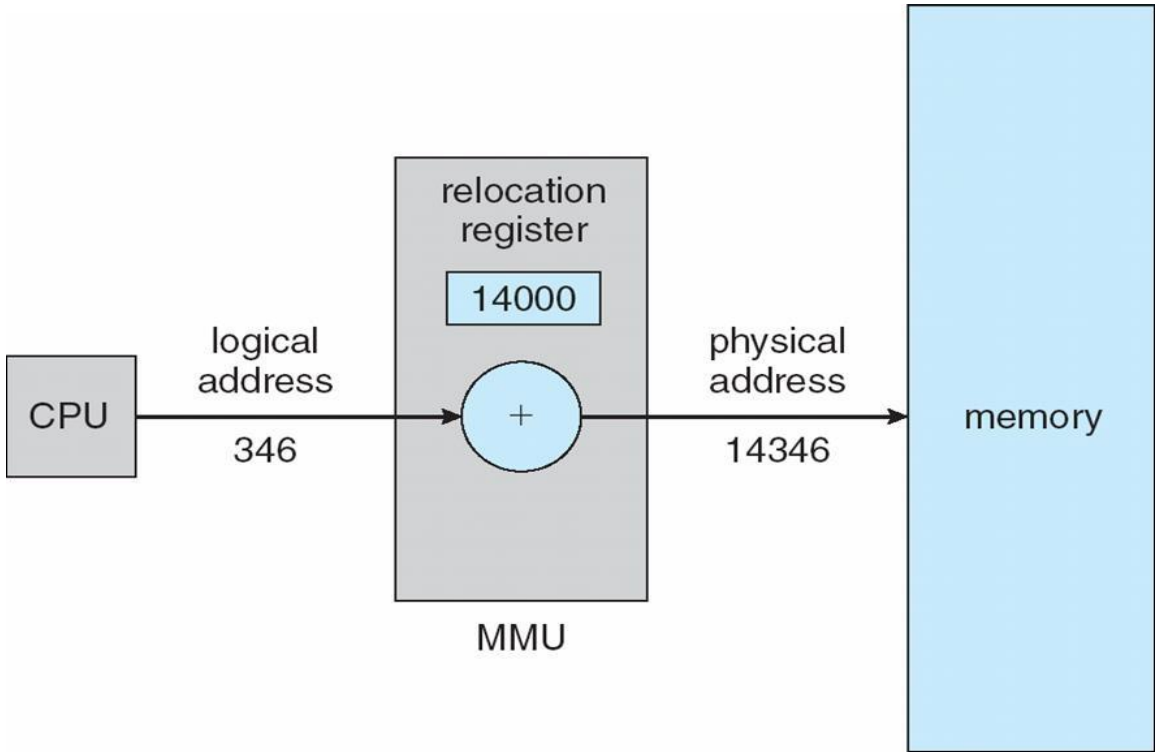


Logical / Physical address space: example

MMU scheme is a generalization of the base-register scheme described before. The value in the **relocation register** (base register) is added to every address generated by a user process at the time the address is sent to memory.

The user program never accesses the real physical addresses, only generate logical addresses, and thinks that the process runs in memory locations from 0 to max. However, these logical addresses must be mapped to physical addresses ranging from R+0 to R+max.

The memory-mapping hardware separates the logical address space from the physical one.



Memory management - optimization strategies

Dynamic Loadning

Unlike before, where the entire program and all data of a process, should be in the physical memory when it was to be executed.

To obtain better memory-space utilization, we can use **dynamic loading**. a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The basic idea here is that not all functionality in a program should be used every time. As a result, a program will take up less space, and there may be more processes in the memory at the same time – i.e., better utilization.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method.

Memory management - optimization strategies

Dynamic Linking

Dynamically linked libraries (DLLs) (a.k.a. shared libraries) are system libraries that are linked to user programs when the programs are run. Some operating systems support only static linking, in which system libraries are treated like any other object module and are combined by the loader into the binary program image.

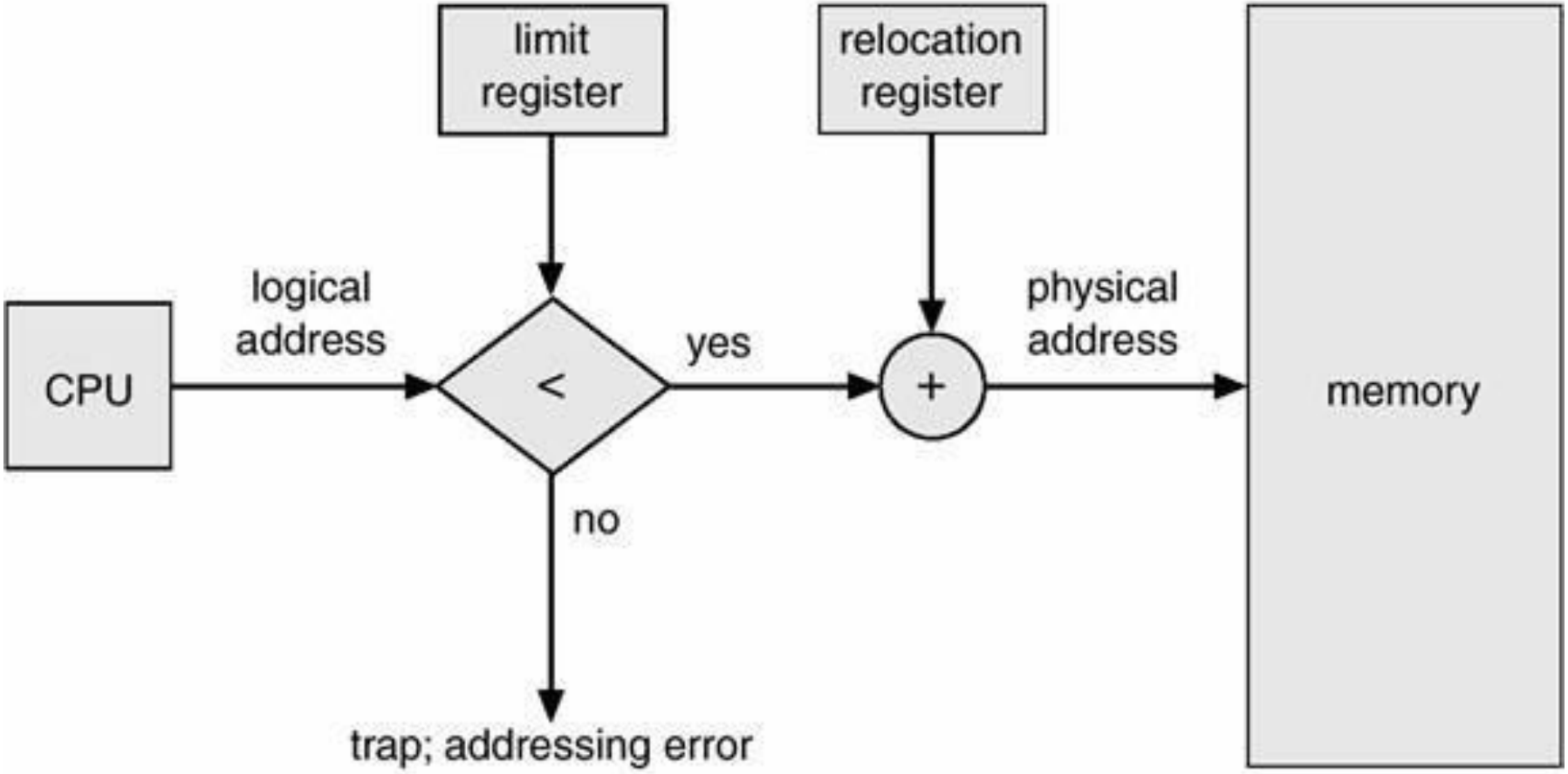
Dynamic linking, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time.

- Without this facility, each program on a system must include a copy of its (language) library in the executable image. This requirement not only increases the size of an executable image but also may waste main memory.
- A second advantage of DLLs is that these libraries can be shared among multiple processes, so that only one instance of the DLL in main memory.

Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in a memory space that can allow multiple processes to access.

Memory management

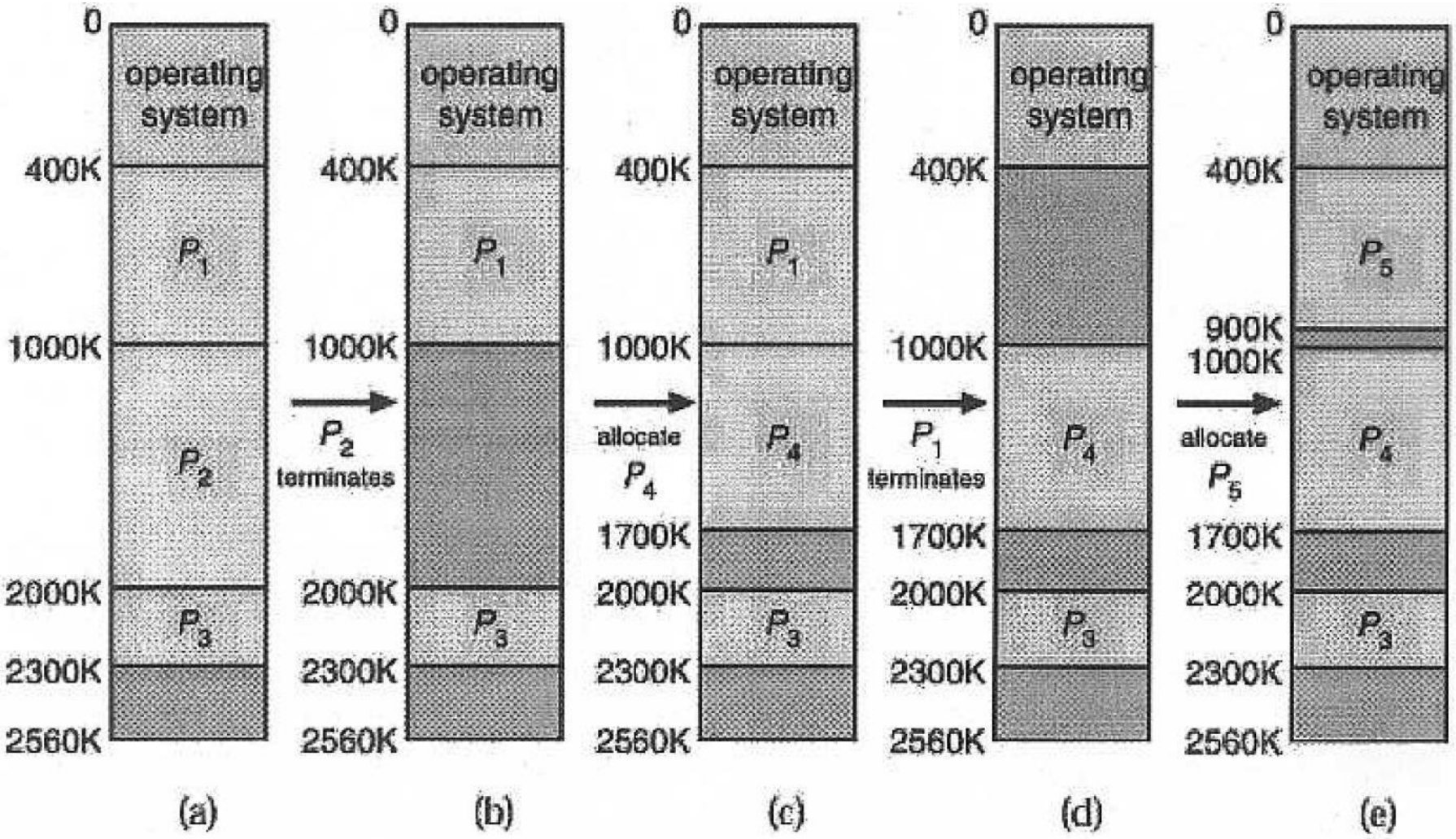
Contiguous Memory Allocation - Memory Protection



When the CPU scheduler selects a process for execution, the dispatcher loads the **relocation and limit registers** with the correct values as part of the context switch.
we can protect both the operating system and the other users' programs and data from being modified by this running process by checking against these registers.

Memory management

Contiguous Memory Allocation



Memory management

Contiguous Memory Allocation

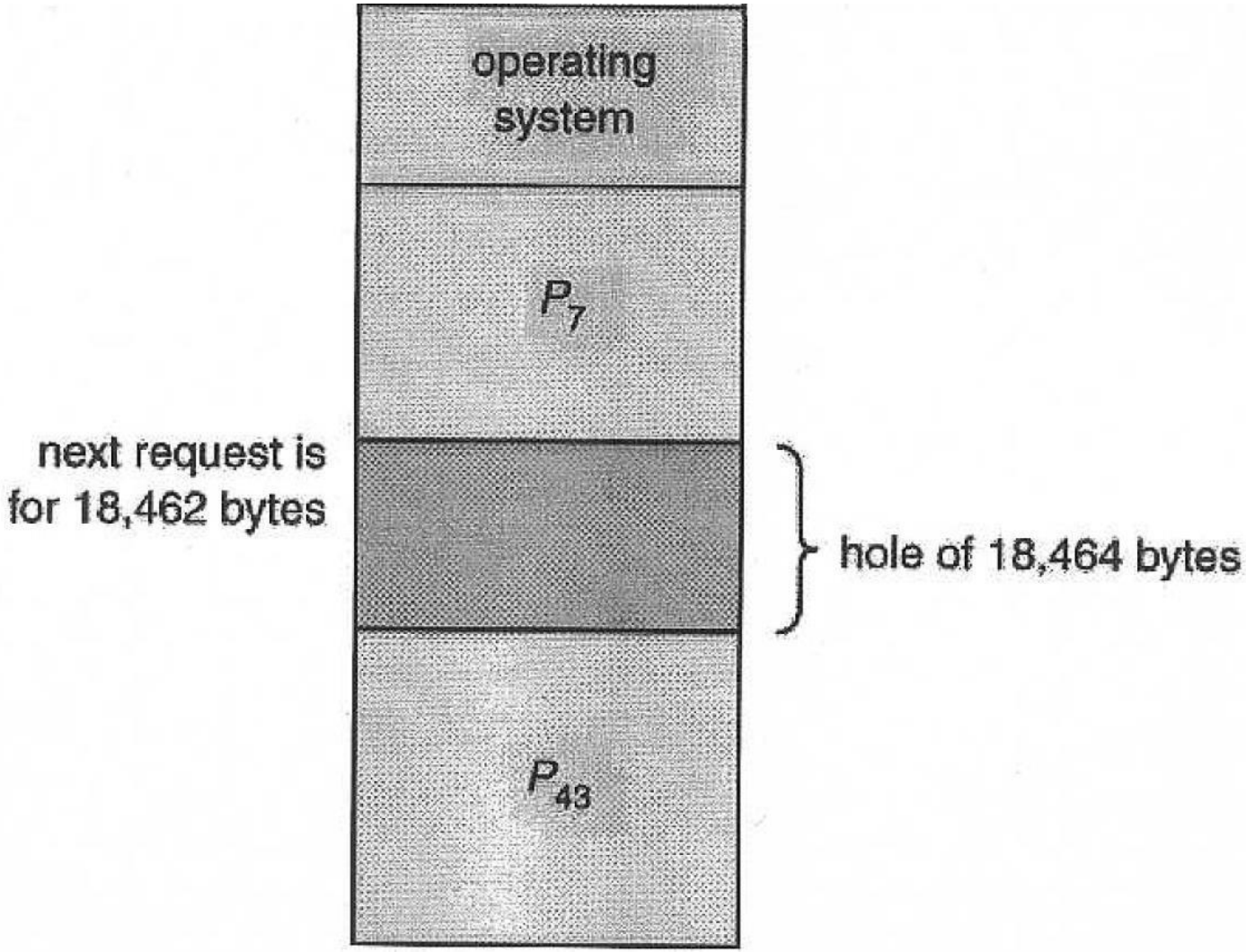
Available memory blocks comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. The following 3 strategies can be used:

- **First-fit**: Allocates the first hole in the set of holes that is large enough for the process
- **Best-fit**: Allocates the smallest hole in the set of holes that is large enough for the process
  - the entire set must be searched (unless it is sorted by size)
  - the strategy leaves the smallest hole in the memory (utilization)
- **Worst-fit**: Allocates the largest hole in the list
  - the entire set must be searched (unless it is sorted by size)
  - the strategy leaves the largest hole in the memory (which may be large enough and can be used for other processes.

First-fit and **best-fit** are better than **worst-fit** in terms of decreasing time and memory utilization

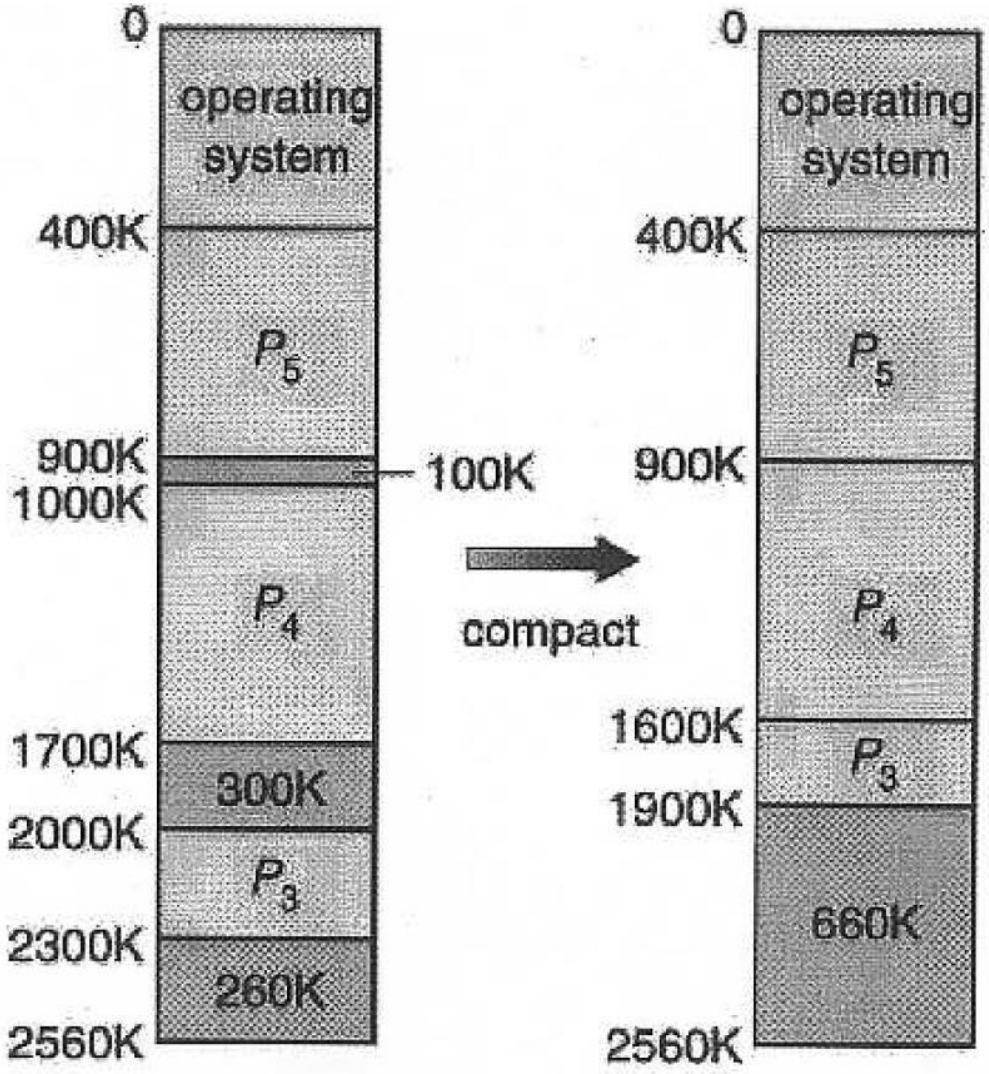
Memory management

Contiguous Memory Allocation - Fragmentation



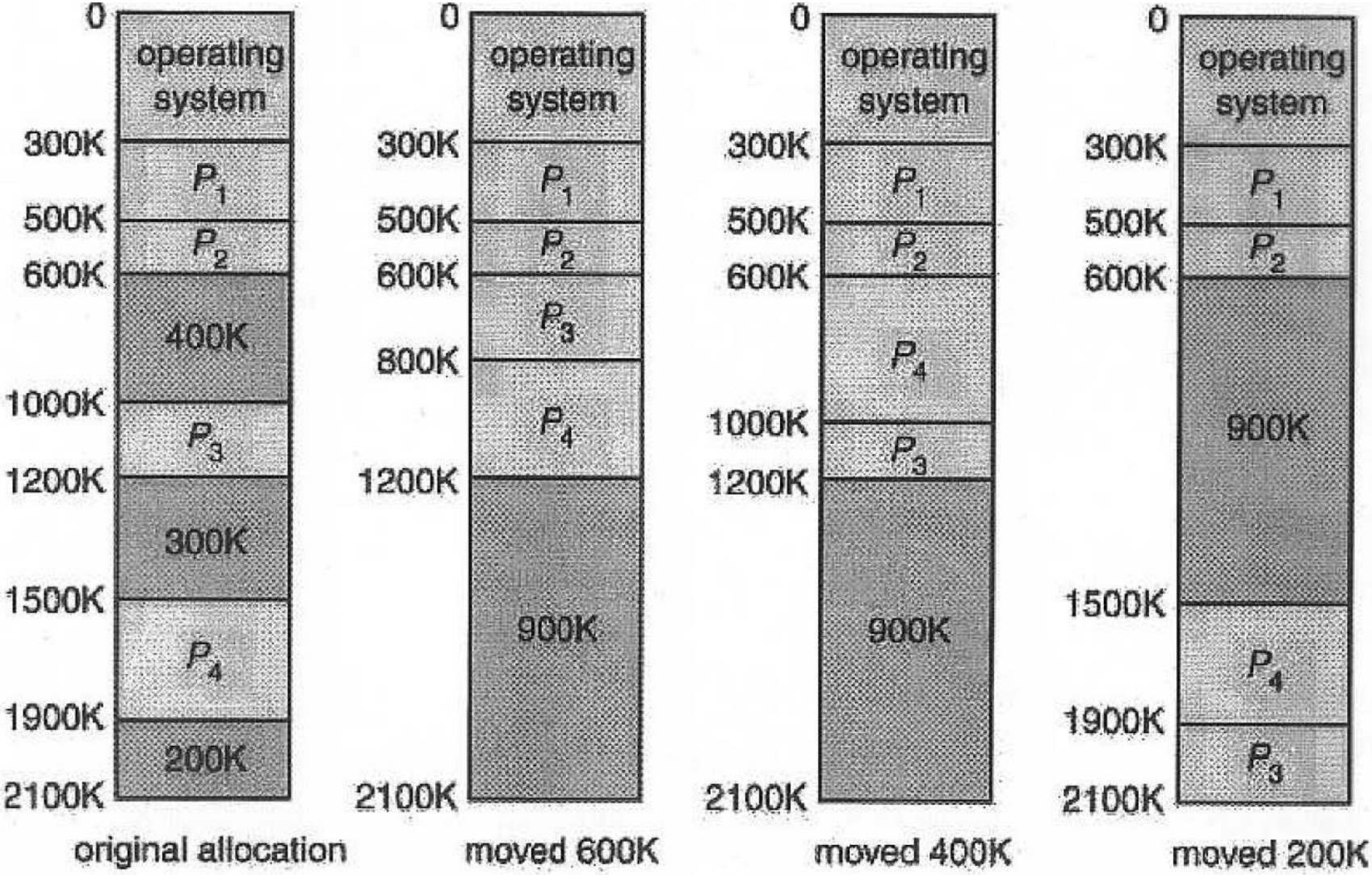
Memory management

Contiguous Memory Allocation - Compaction



Memory management

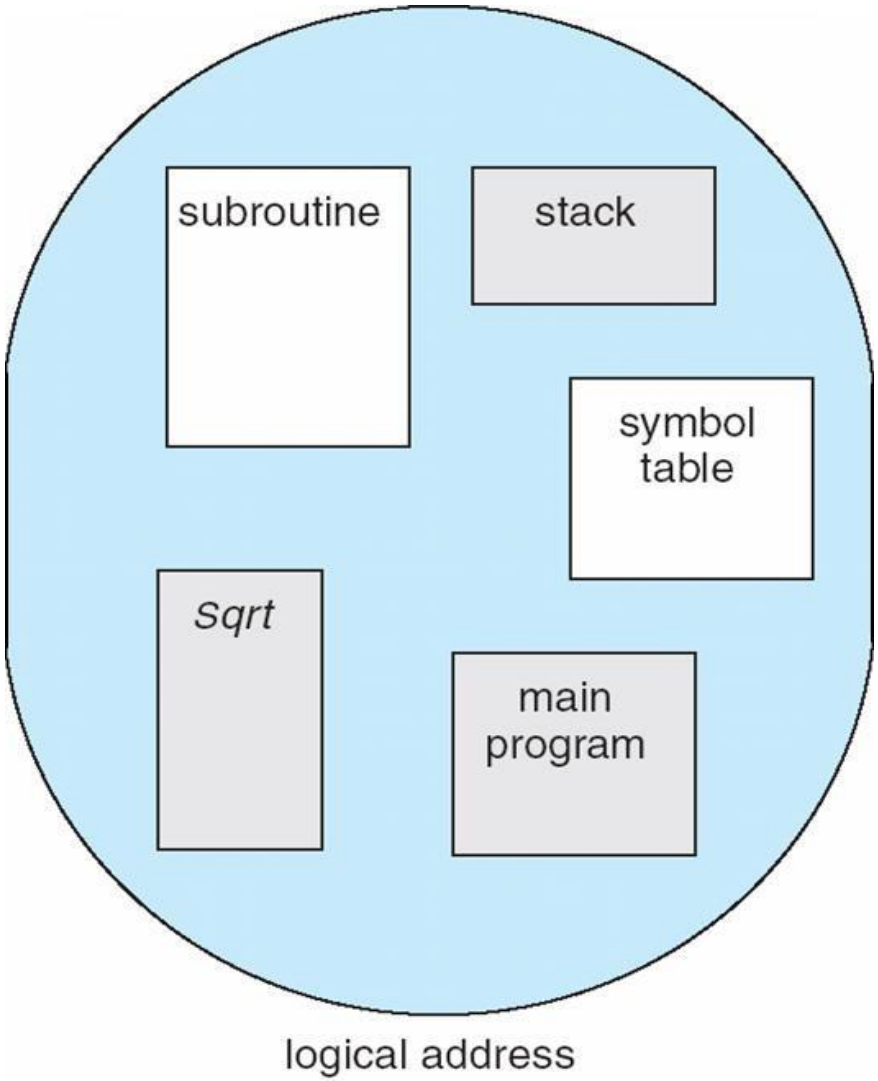
Contiguous Memory Allocation - Compaction



Segmentation

A memory management model that supports the user's view of memory

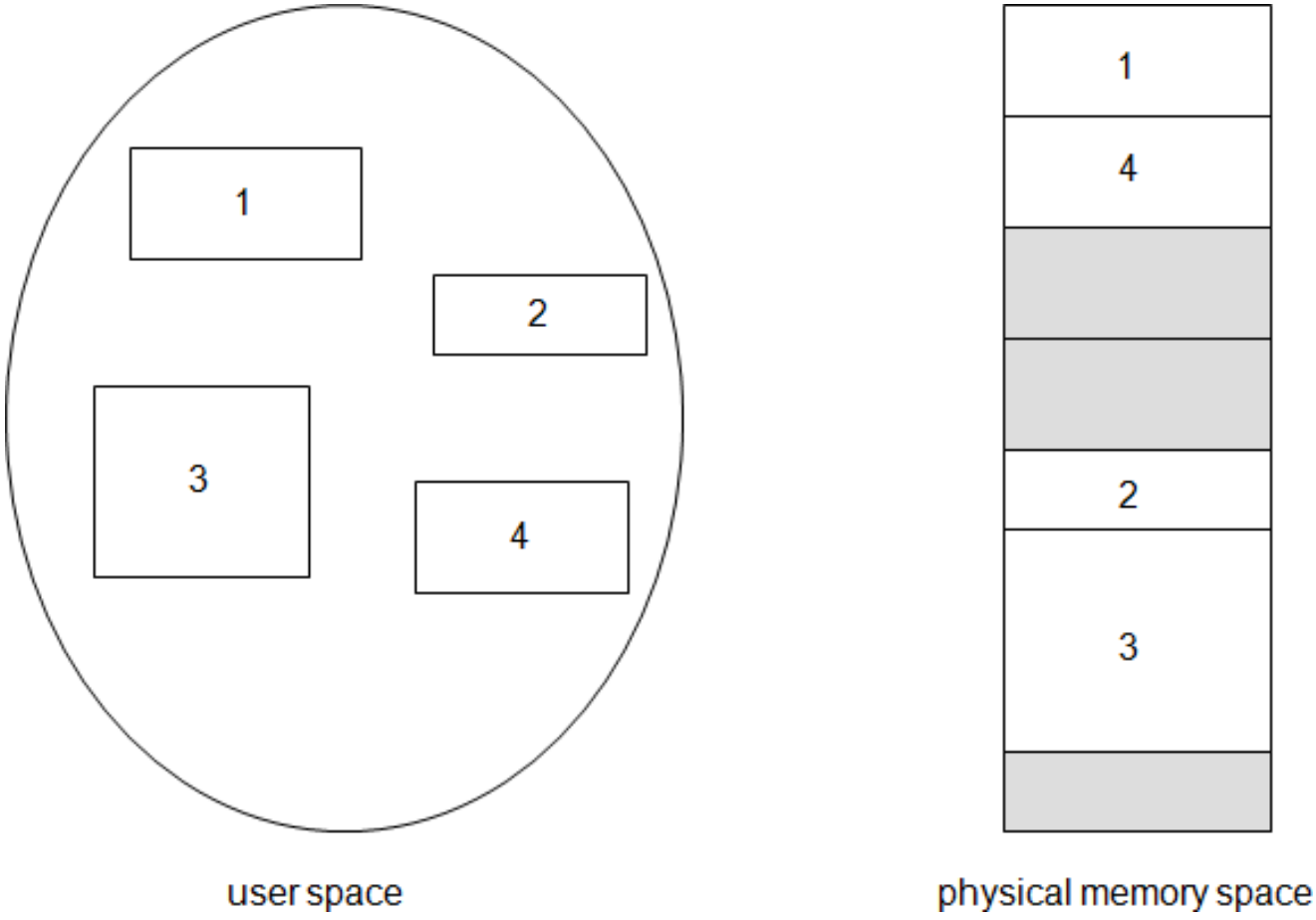
A program is a collection of segments



Segmentation

A memory management model that supports the user's view of memory

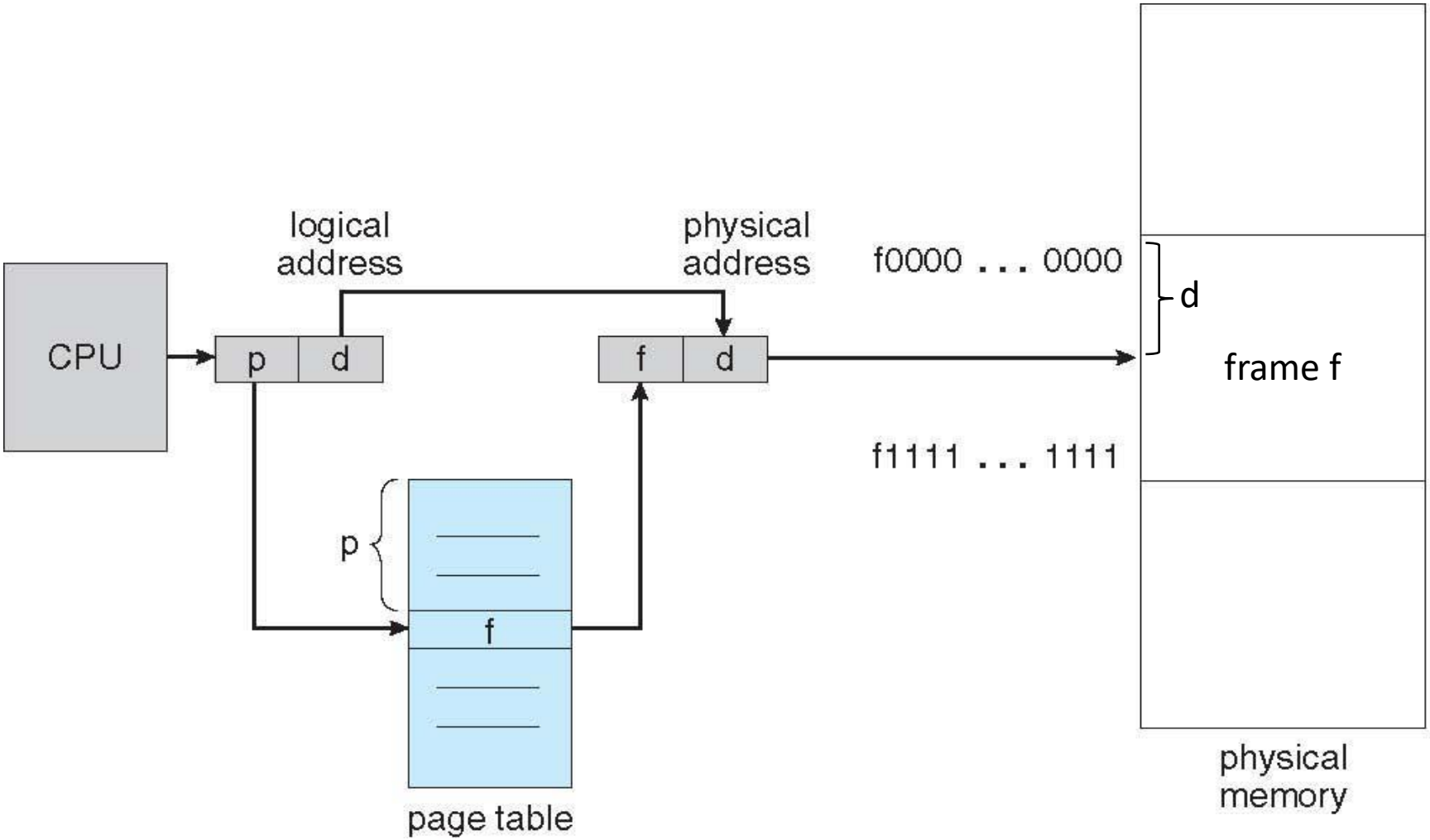
Logical view of the process and physical organization



Paging

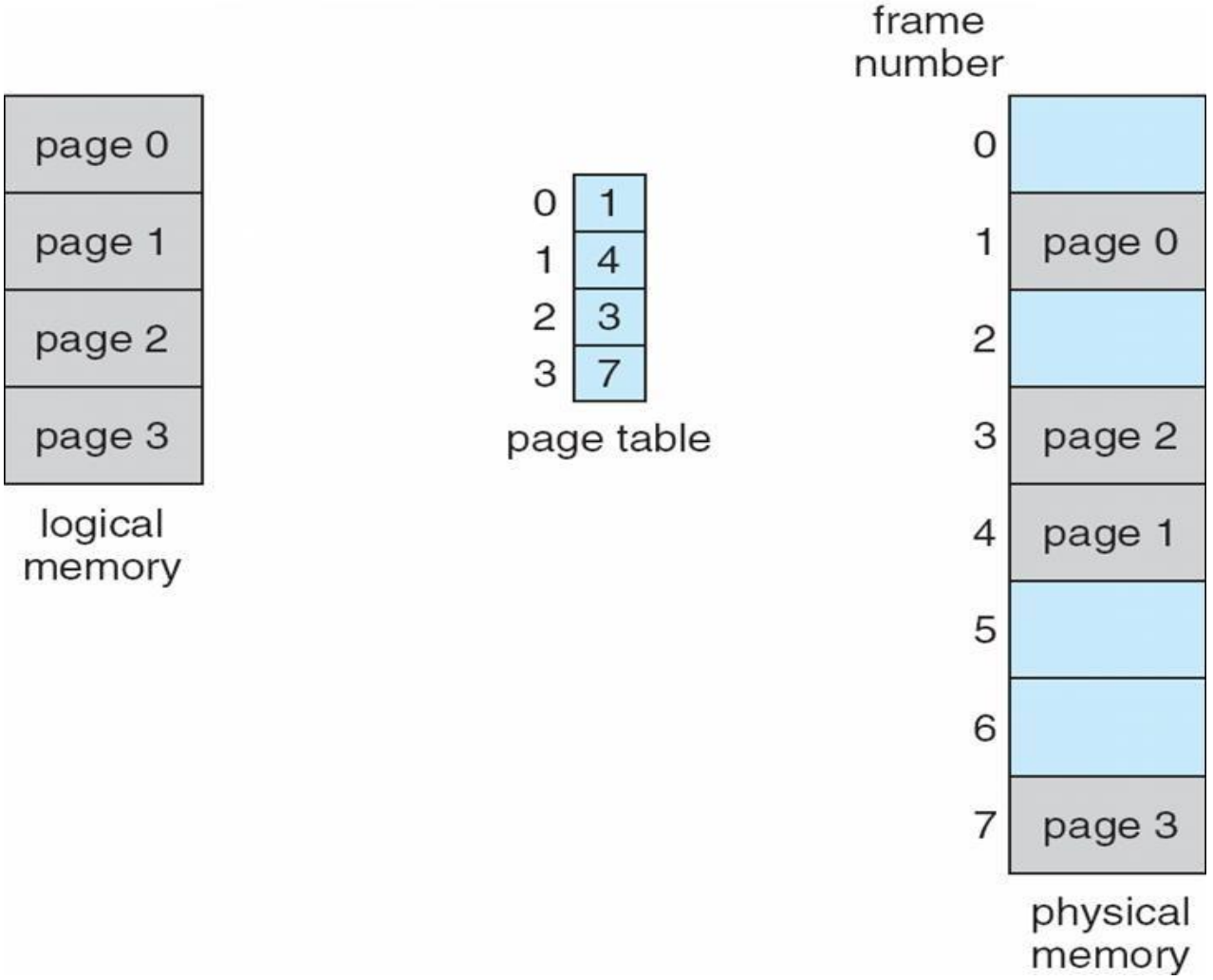
Hardware support

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



Paging

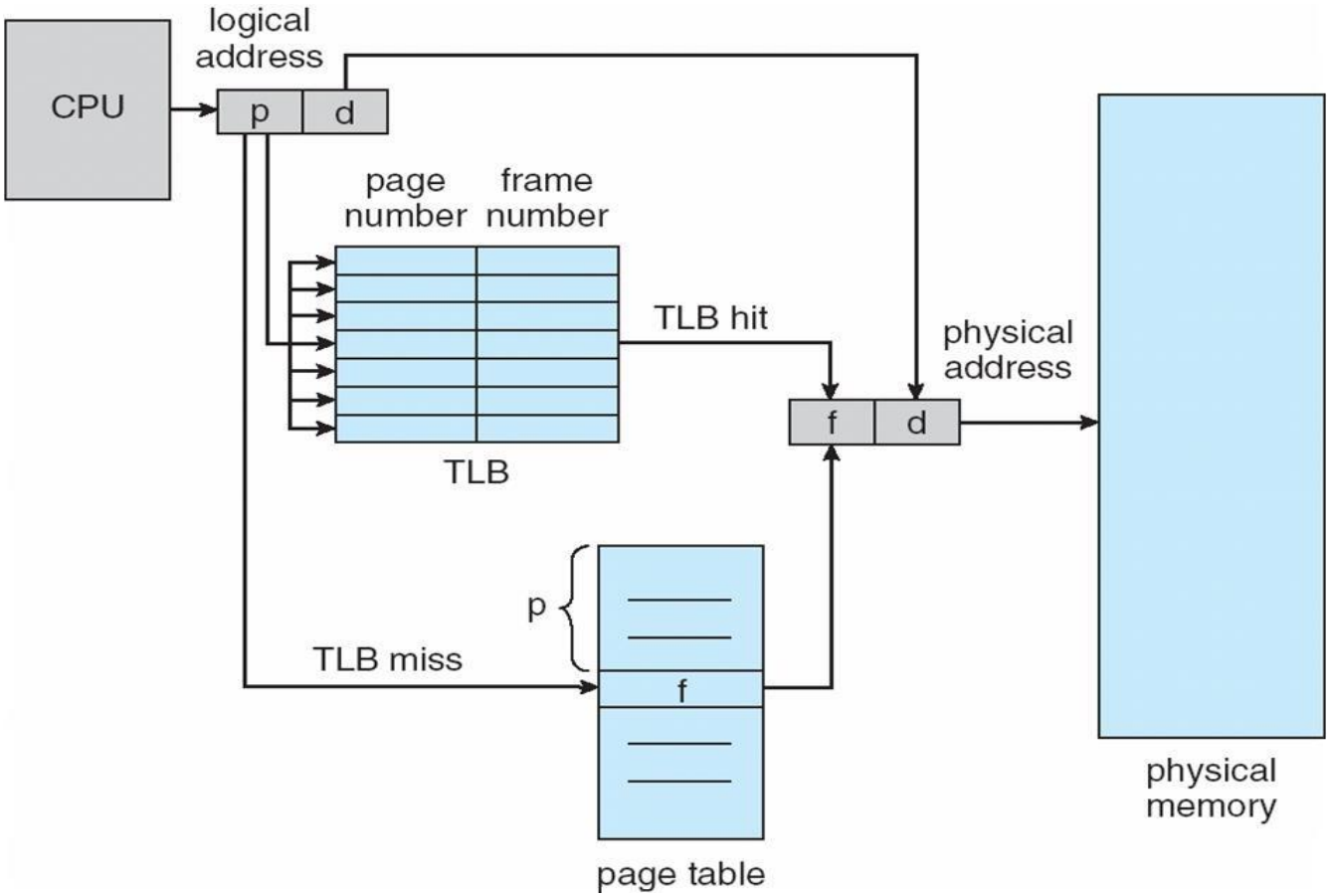
Logical and physical memory



Note - no external fragmentation (internal fragmentation only)

Paging

Reducing access time using a TLB



TLB sizes between 32 and 1024 entries

E.g., - Intel i7 CPU: instruction TLB 128 entries, and Data TLB 64 entries

Paging

Reducing access time using a TLB example

- Storage in TLB (Fully Associative Lookup) = t_{TLB} ns
  - Will often be < 20% of memory access time
- Hit ratio = α
  - Hit ratio – How often do we find the page number in the TLB.
    Hit ratio depends, among other things, on the number of entries in the TLB

HIT: Lookup in TLB→Page no. exist→ memory accessed
MISS: Lookup in TLB→Page no. does not exist → Page table accessed → memory accessed

Effective Access Time (EAT)

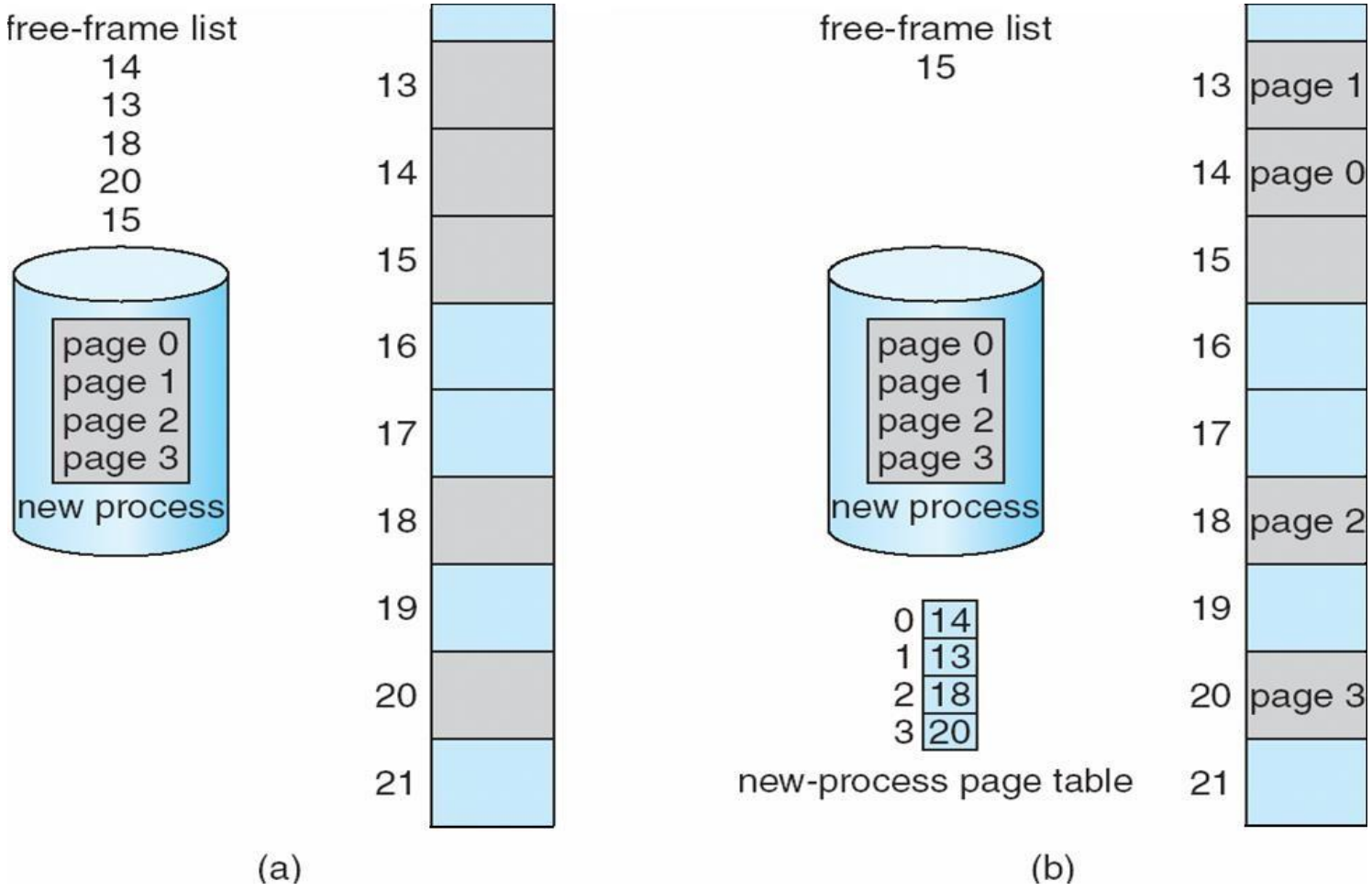
$$EAT = (t_{TLB} + t_{mem})\ \alpha + (t_{TLB} + t_{mem} + t_{mem})(1 - \alpha)$$

Example: $\alpha = 80\%$, $t_{TLB} = 20$ns for TLB lookup, $t_{mem} = 100$ns for memory access $EAT = (100 + 20) \cdot 0{,}80 + (100 + 100 + 20)(1 - 0{,}80) =$ **140ns**

Realistic example: $\alpha = 99\%$, $t_{TLB} = 20$ns for TLB lookup, $t_{mem} = 100$ns for memory access $EAT = (100 + 20) \cdot 0{,}99 + (100 + 100 + 20)(1 - 0{,}99) =$ **121ns**

Paging

The operating system keeps track of the available frames in a free-frame list

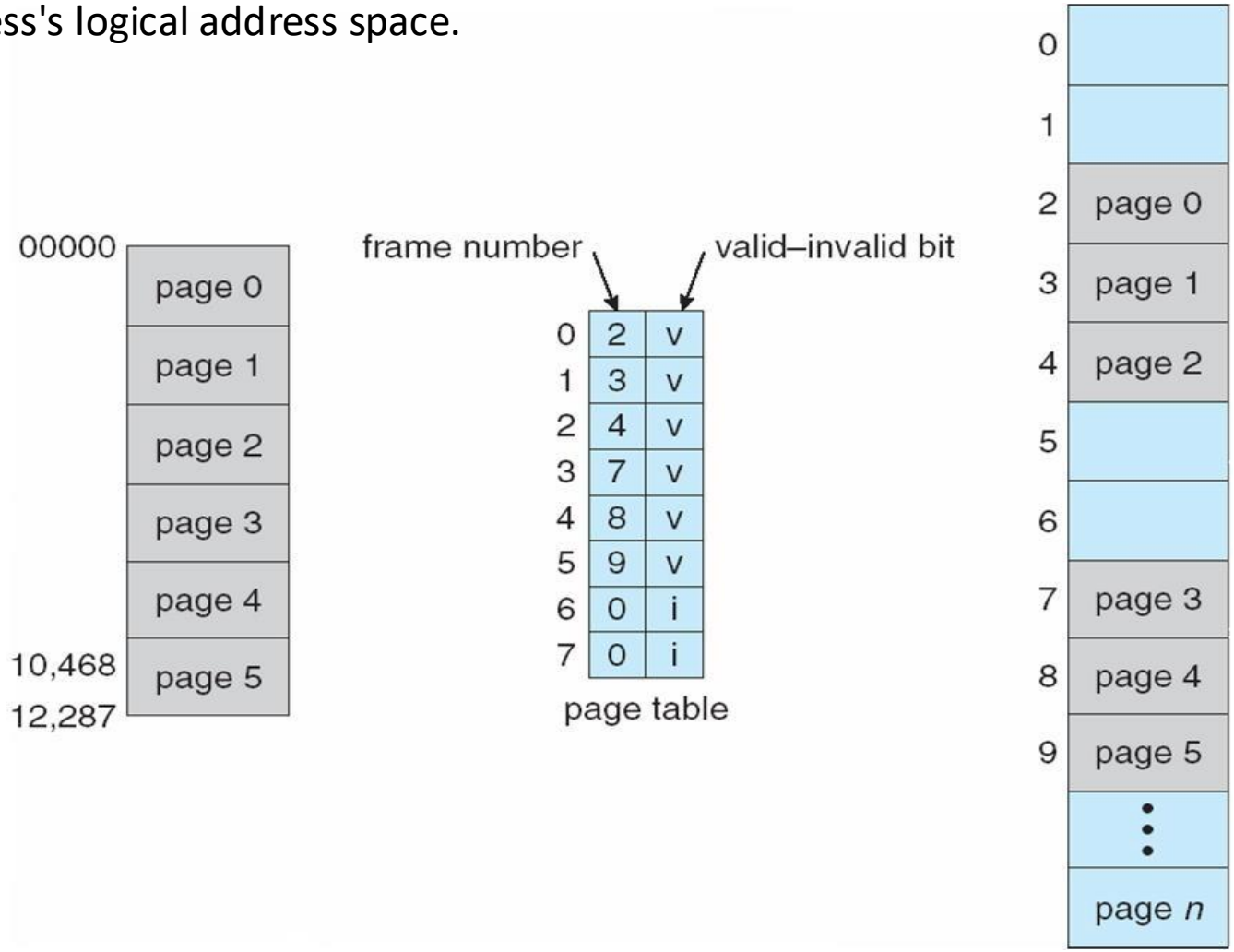


It is faster to assign the processes frames from this free-frame list than to first have to find them when they are to be used

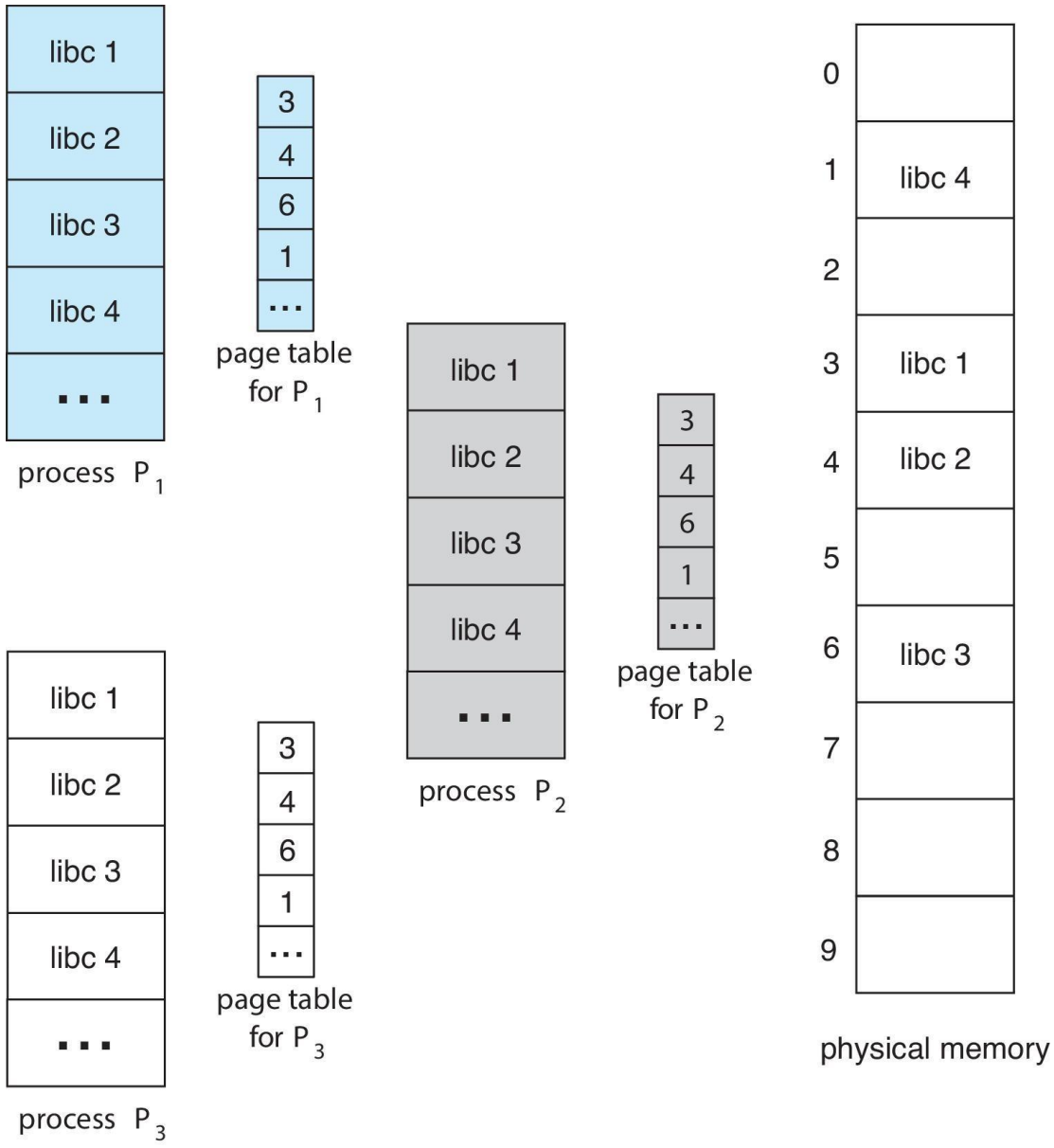
Protection

e.g., read only, read-write or execute only

One additional bit is generally attached to each entry in the page table: a valid–invalid bit. When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space.

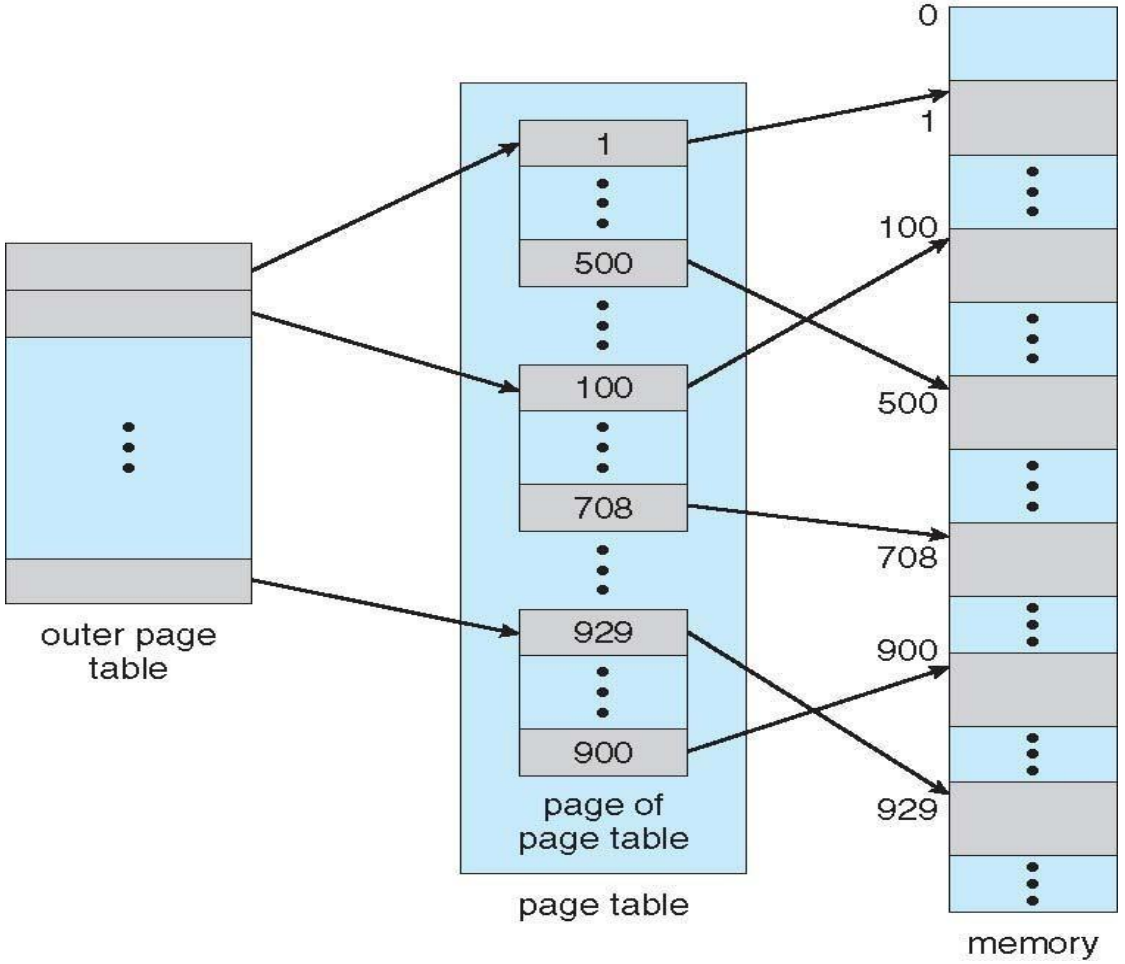


Shared Pages



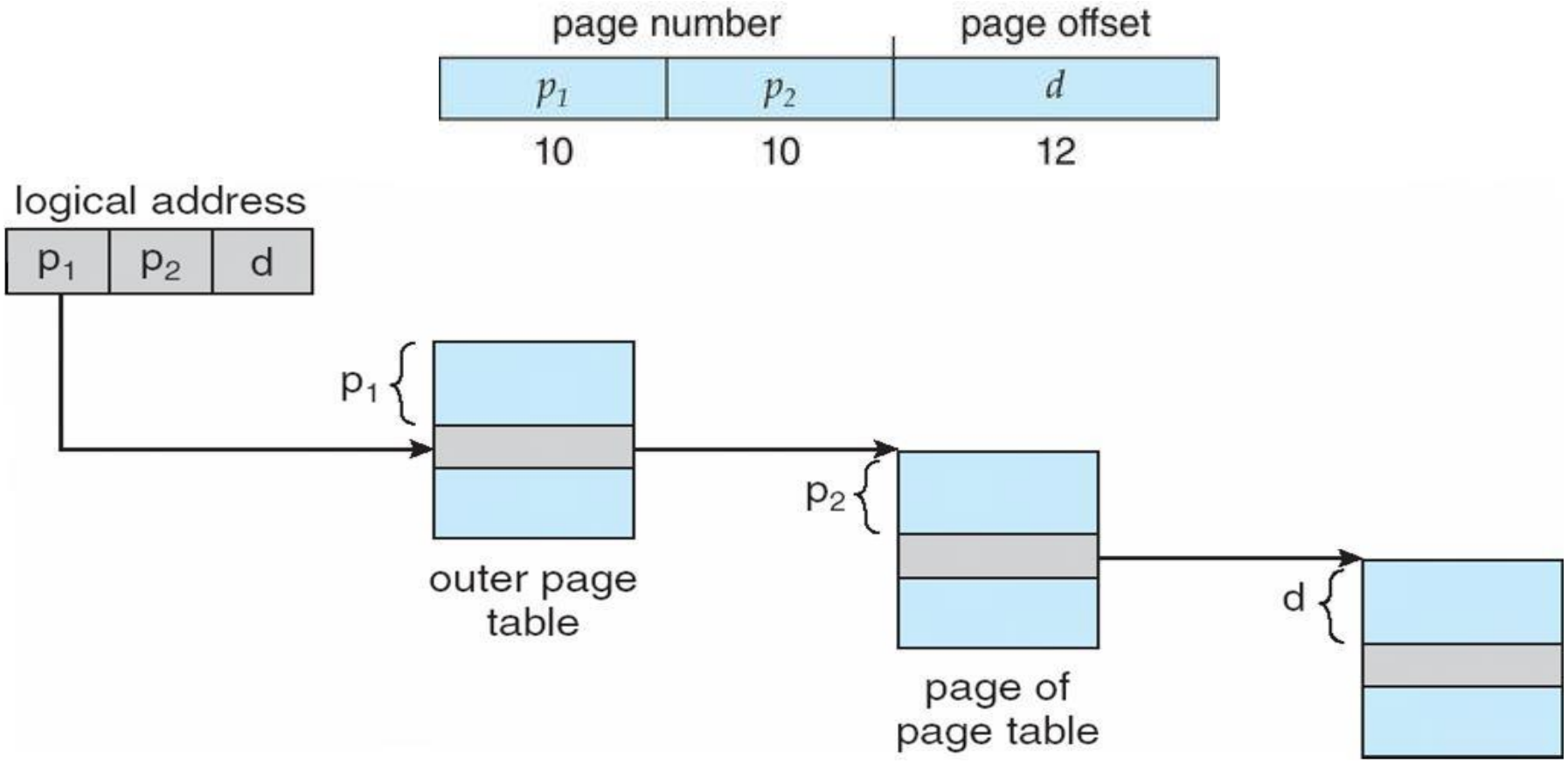
Remember that the code in the memory must be reentrant and that each process has its own data

Hierarchical Page Table structure



E.g., a 32-bit address space and a page size of 4 Kb gives over 1 million entries. Here the solution can be several levels in a page table, so the whole table can be divided into small pieces.

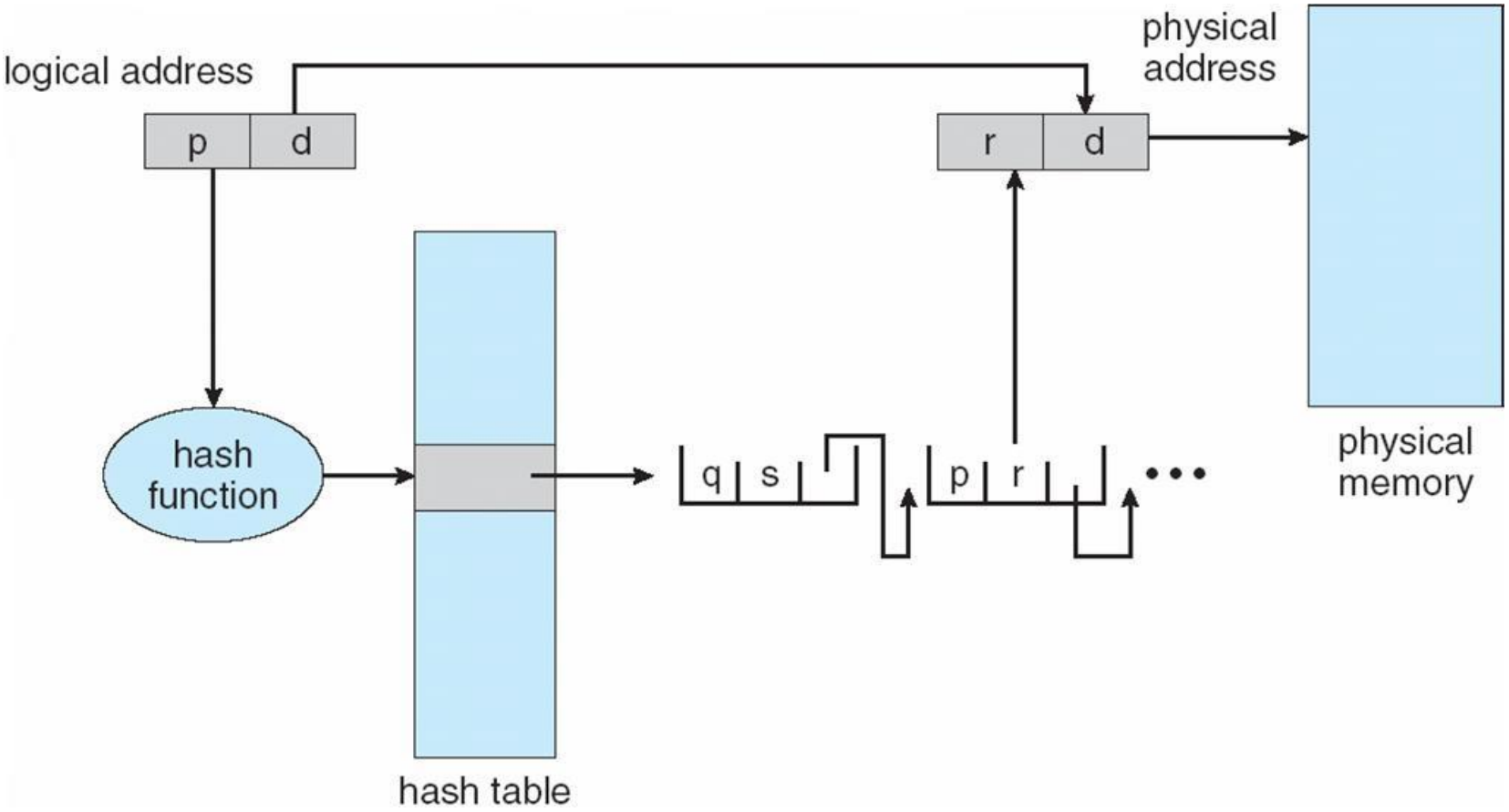
Hierarchical Page Table



The advantage: does not have to have such a large amount of page table related data in memory.

The cost for this is longer access times (which can be partially remedied with TLBs)

Hashed Page Table

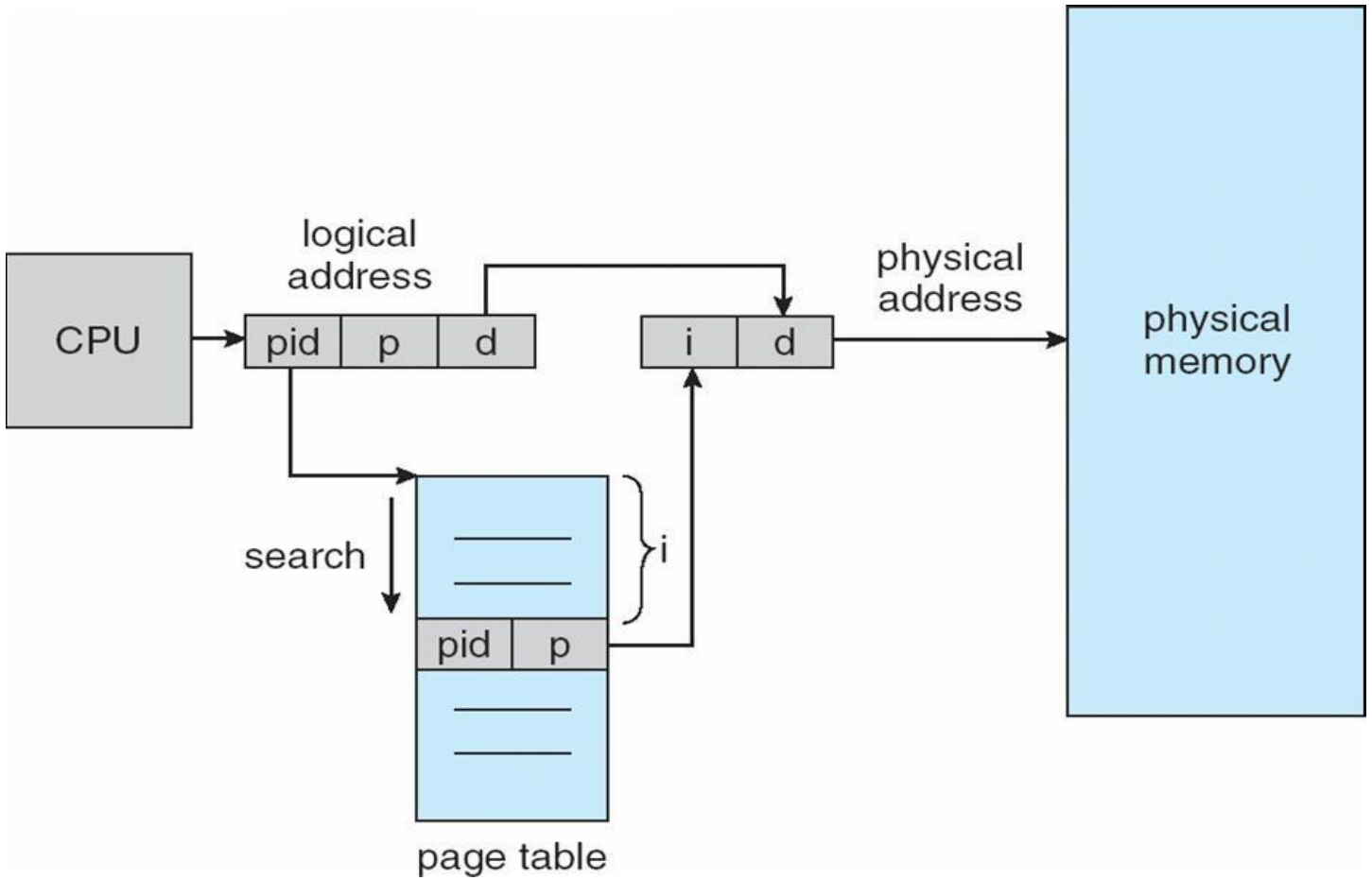


Often used for address spaces larger than 32-bit. If the address space becomes very large, clustered page table is used.

e.g.: $f_{hash}(p) \rightarrow$ entry q (a linked list): p is compared to the first place in the list elements: when p is found, r is used

(the list is there because multiple $f_{hash}(x)$ can lead to the same entry)

Inverted Page table

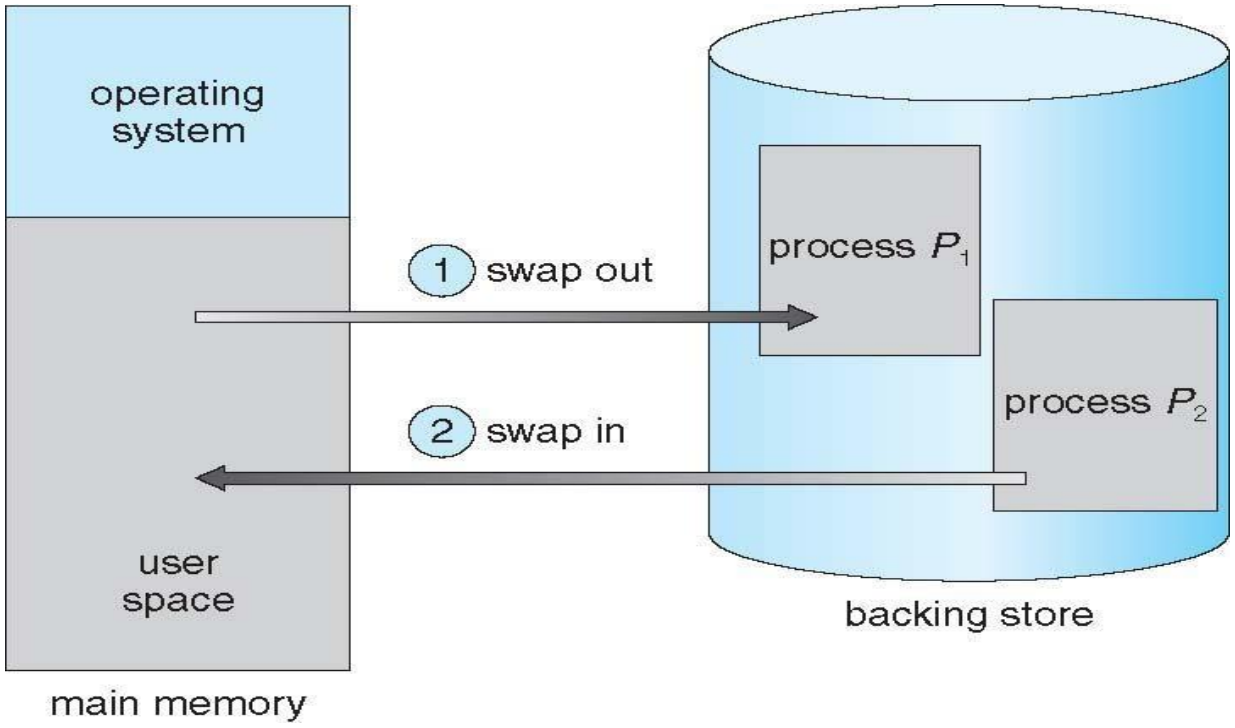


only one page table in the system, one entry for each page of phsical memory.

Reduces administration but loses the ability to share pages.

Memory management - optimization strategies

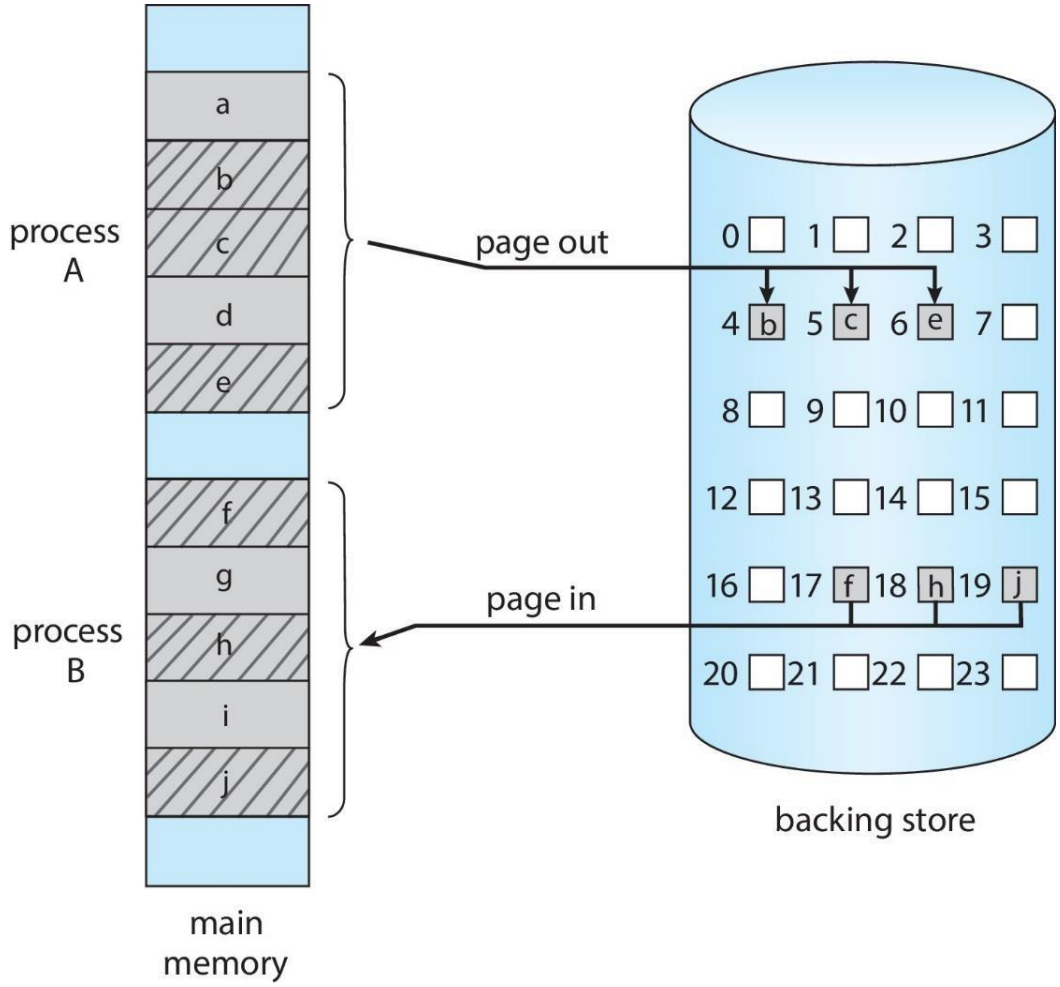
Swapping



Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Memory management - optimization strategies

Swapping with paging



Here only a subset of the process pages is swapped in/out. Those that are not swapped here could be pages that contain I/O buffers.