Memory Management

Main Memory

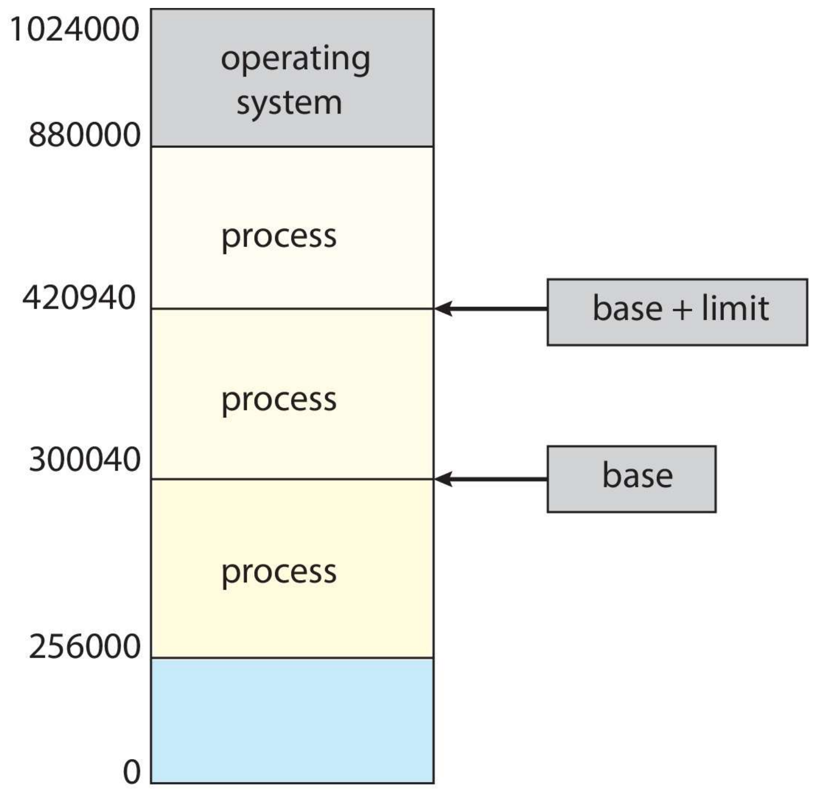
Background

- Memory is central to the operation of a modern computer system
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and the registers built into each processing core are the only general-purpose storage that the CPU can access directly

- There are machine instructions that take memory addresses as arguments, but none that take disk addresses
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices
- If the data are not in memory, they must be moved there before the CPU can operate on them

Protection

- Need to ensure that a process can access only those addresses in it address 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
- This protection is given by using two registers, usually a base and a limit, as illustrated in Figure in the next slide

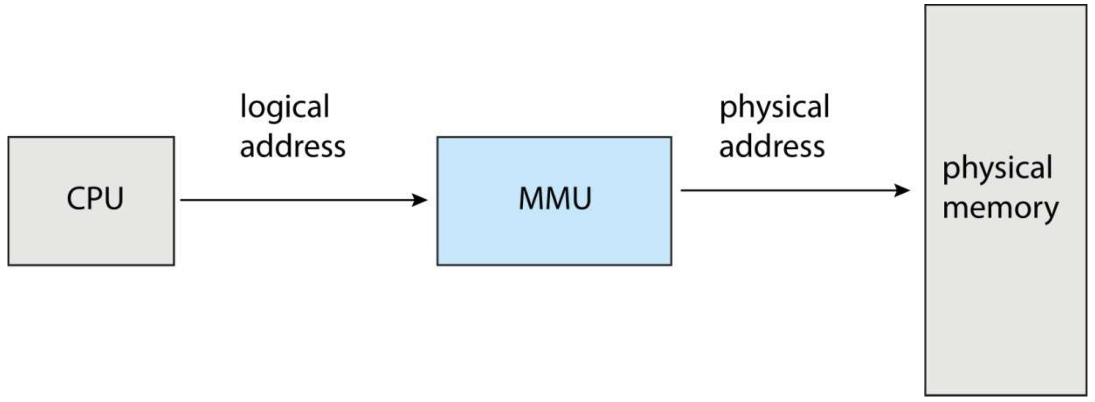

1024000
operating system
880000
process
420940
base + limit
process
300040
base
process
256000
0

- The base register holds the smallest legal physical memory address
- The limit register specifies the size of the range
  - For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a logical address
  - Also known as virtual address
- An address seen by the memory unit that is, the one loaded into the memory-address register of the memory is commonly referred to as a physical address.
- The set of all logical addresses generated by a program is a logical address space

- Physical address space is the set of all physical addresses generated by a program
- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU)



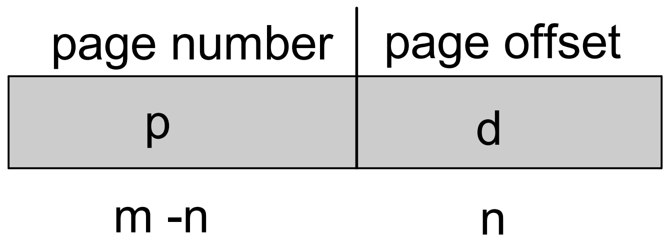
Paging

- Previous to paging, the physical address space of a process to was contiguous
- Continuous memory allocation had demerits
  - Fragmentation
  - To overcome fragmentation, there was a need for compaction
- Paging avoids these disadvantages and offers other advantages
- Paging memory management strategy is used in all the modern operating systems

Basic Method

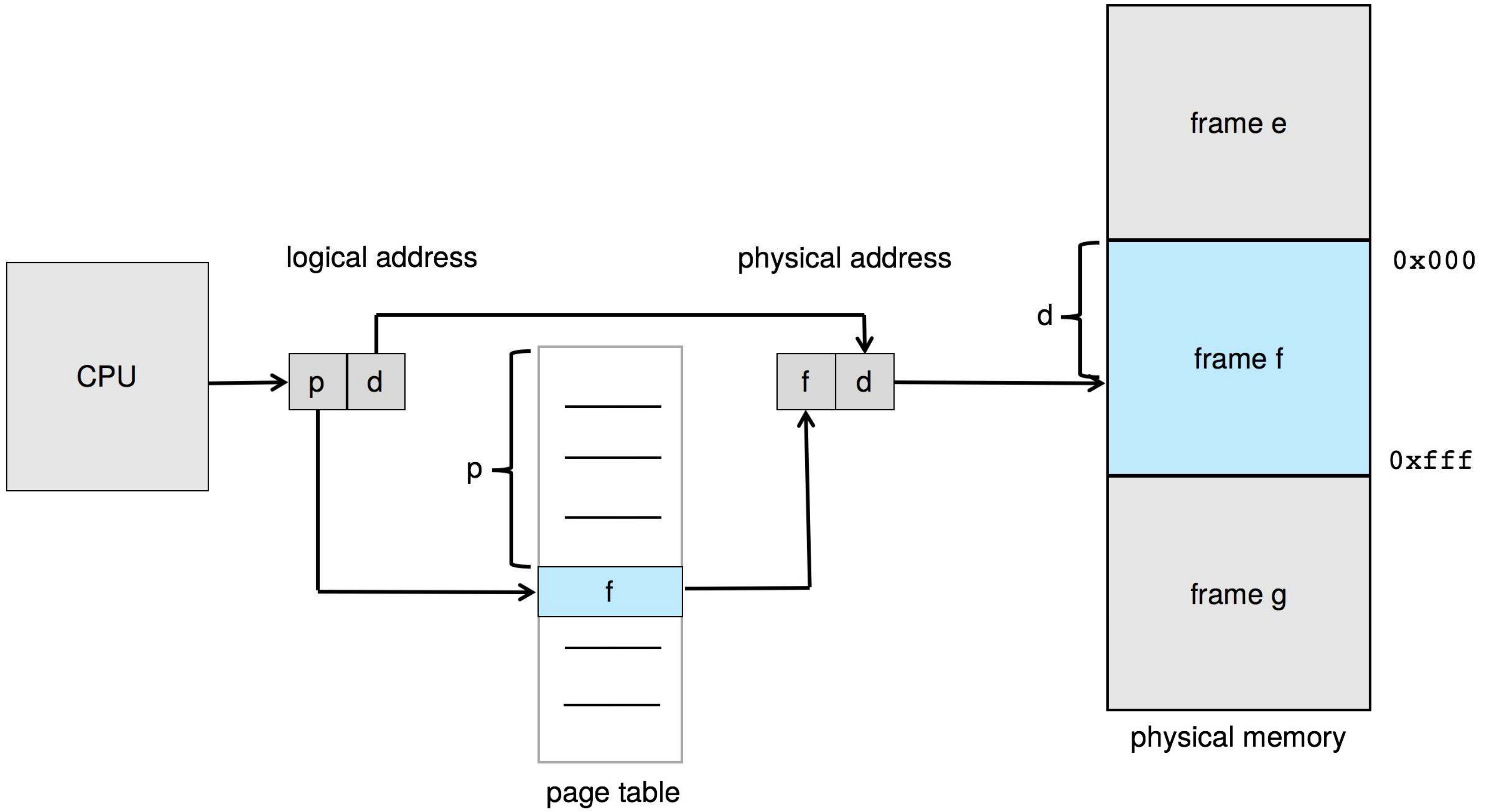
- The basic method involves
  - breaking physical memory into fixed-sized blocks called ***frames***
  - breaking logical memory into blocks of the same size called ***pages***
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store)
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames

- Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d):



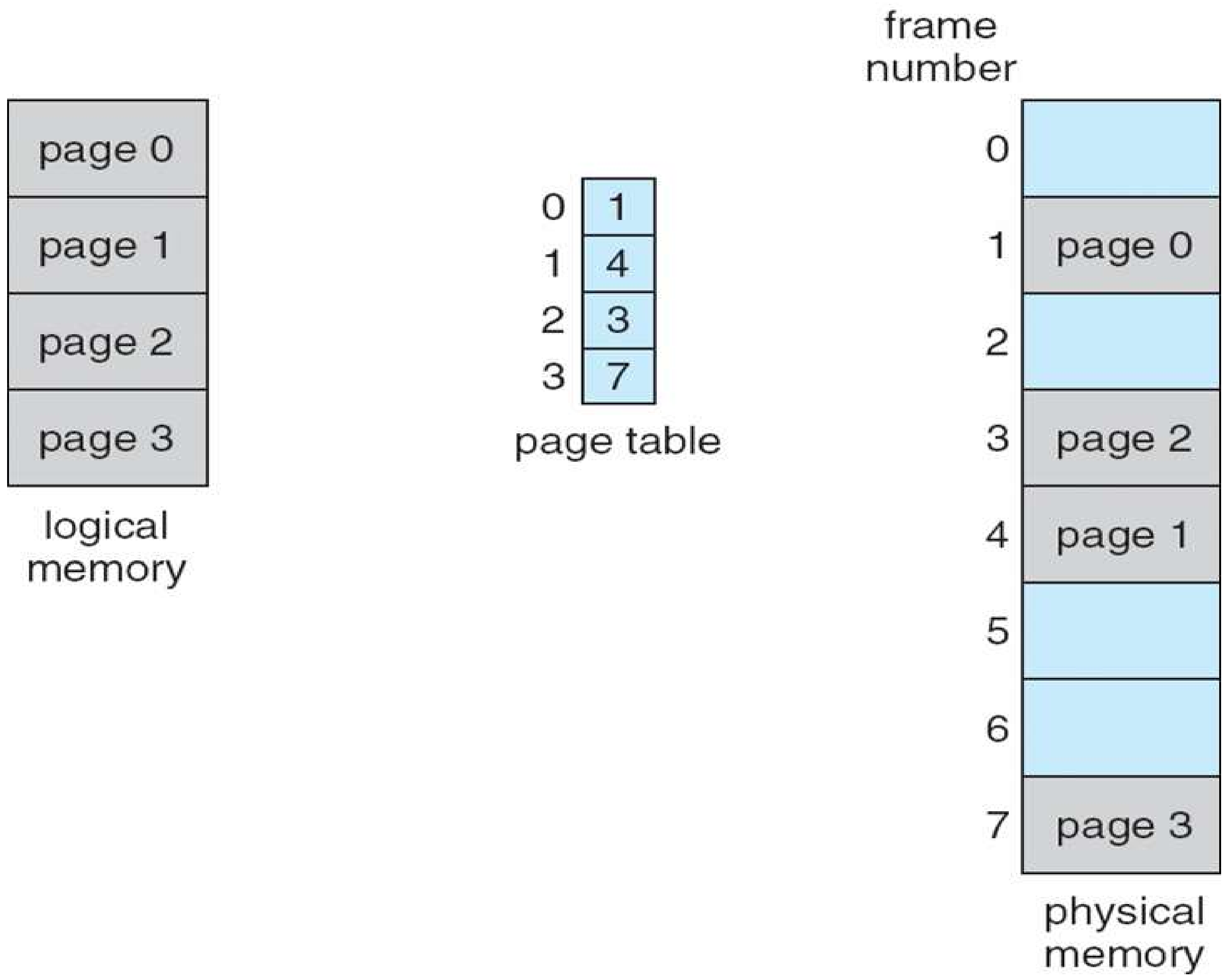
- The page number is used as an index into a per-process page table
- This is illustrated in the figure in the next slide

Paging Hardware



- The page table contains
  - the base address of each frame in physical memory, and
  - the offset is the location in the frame being referenced
- Thus, the base address of the frame is combined with the page offset to define the physical memory address
- The paging model of memory is shown in Figure in the next slide

Paging model of logical and physical memory

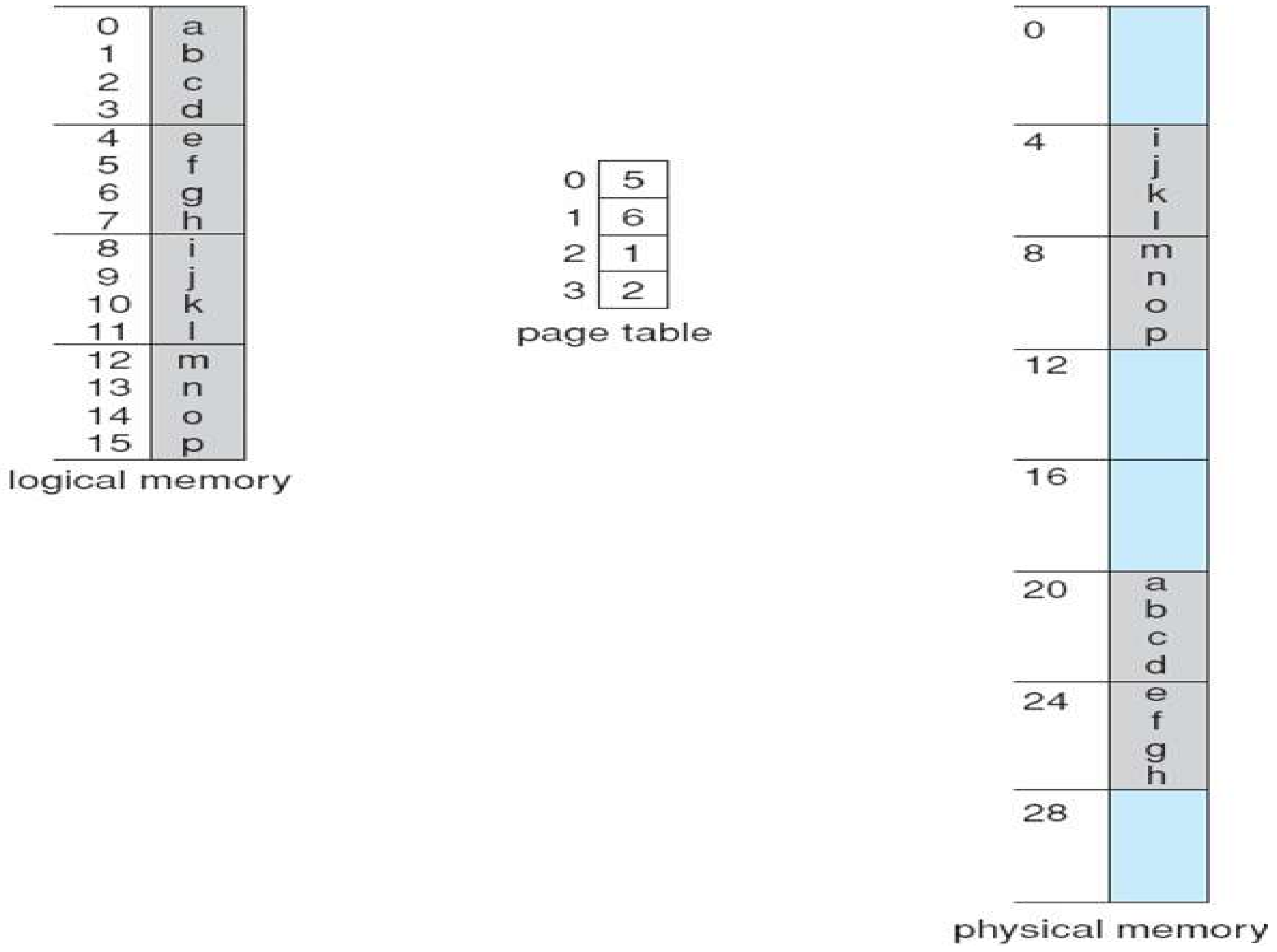


- The steps taken by the MMU to translate a logical address generated by the CPU to a physical address:
    1. Extract the page number p and use it as an index into the page table
    2. Extract the corresponding frame number f from the page table
    3. Replace the page number p in the logical address with the frame number f
- As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address

- The page size (like the frame size) is defined by the hardware
- The size of a page is a power of 2, typically varying between 4 KB and 1 GB per page, depending on the computer architecture
- If the size of the logical address space is 2^m,and a page size is 2^n bytes,
  - then the high-order m−n bits of a logical address designate the page number, and the n low-order bits designate the page offset
  - Diagram in slide 11

Example

- Consider the memory in the figure below:

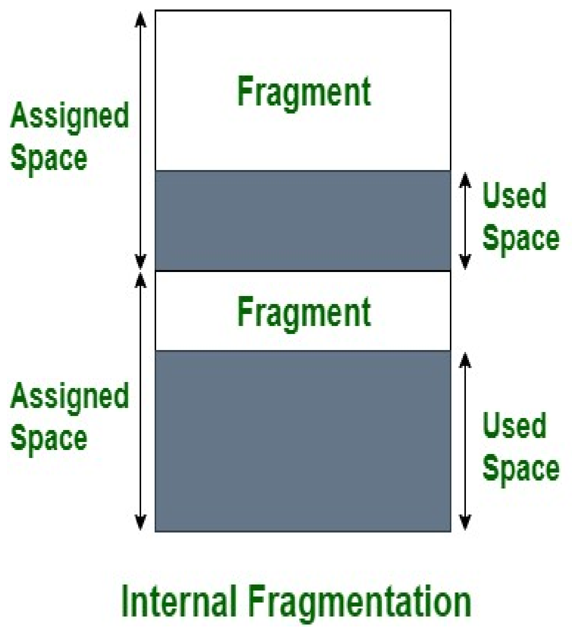


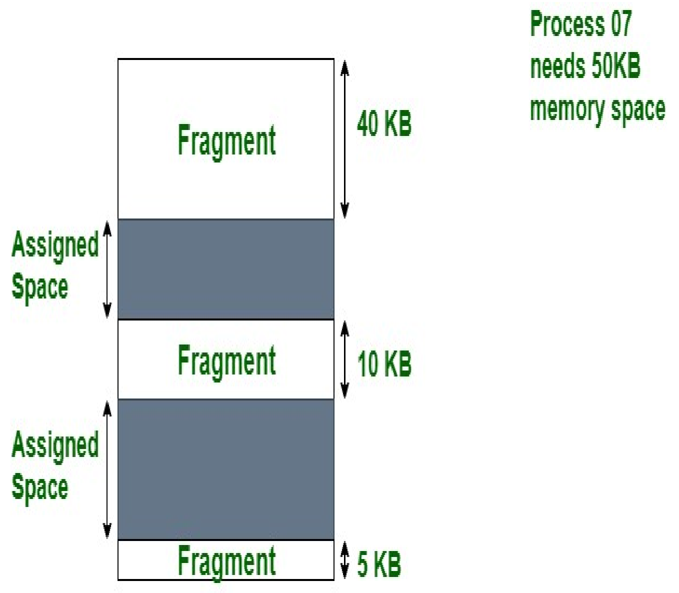
- Here, in the logical address, n = 2 and m = 4
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), the mapping of programmer's view of memory with the physical memory is shown
- Logical address 0 is page 0, offset 0
- Indexing into the page table, we find that page 0 is in frame 5
- Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]

- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 ×4) + 3]
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6
  - Thus, logical address 4 maps to physical address 24 [= (6 ×4) + 0]
- Logical address 13 maps to physical address 9
- When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it

- However, we may have some internal fragmentation
- In fragmentation there are two types
  - External Fragmentation
  - Internal fragmentation
- External fragmentation
  - 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

- Internal fragmentation
  - With this approach, the memory allocated to a process may be slightly larger than the requested memory
  - The difference between these two numbers is internal fragmentation i.e., unused memory that is internal to a partition



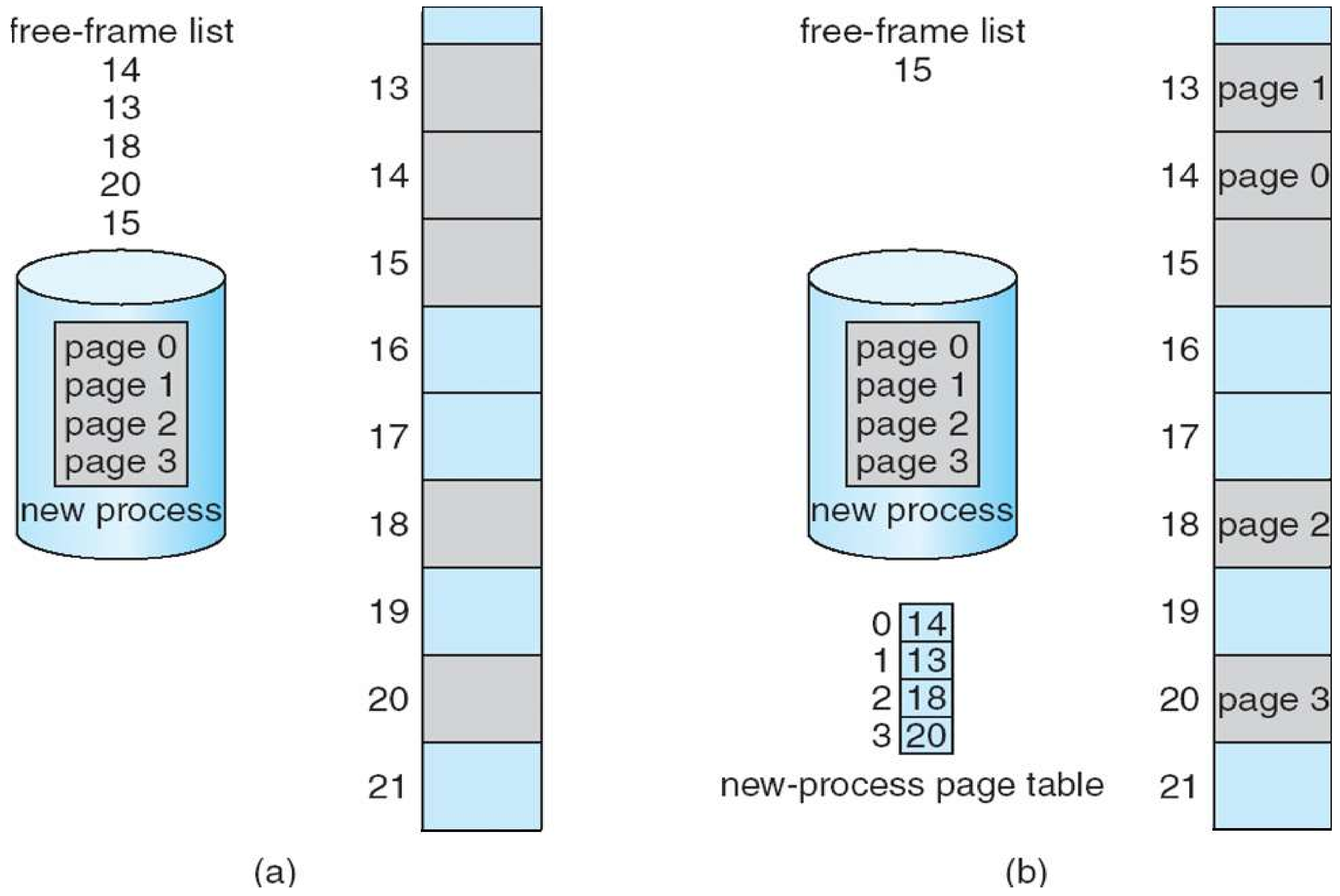


External fragmentation

- Example f or internal fragmentation in paging
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- In the worst case, a process would need n pages plus 1 byte It would be allocated n+1 frames, resulting in internal fragmentation of almost an entire frame

- Free frames
  - When a process arrives in the system to be executed, its size, expressed in pages, is examined
  - Each page of the process needs one frame
  - Thus, if the process requires n pages, at least n frames must be available in memory
  - If n frames are available, they are allocated to this arriving process
  - The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process
  - The next page is loaded into another frame, its frame number is put into the page table, and so on

Free frames



free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Implementation of Page Table

- In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient
- However, this approach increases context-switch time, as each one of these registers must be exchanged during a context switch
- The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).

- Most contemporary CPUs, however, support much larger page tables (for example,2^{20} entries)
- For these machines, the use of fast registers to implement the page table is not feasible
- Rather, the page table is kept in main memory, and a page-table base register (PTBR) points to the page table
- Changing page tables requires changing only this one register, substantially reducing context-switch time

- Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times
- Suppose we want to access location i
  - We must first index into the page table, using the value in the PTBR offset by the page number for i
  - This task requires one memory access
  - It provides us with the frame number, which is combined with the page offset to produce the actual address
  - We can then access the desired place in memory

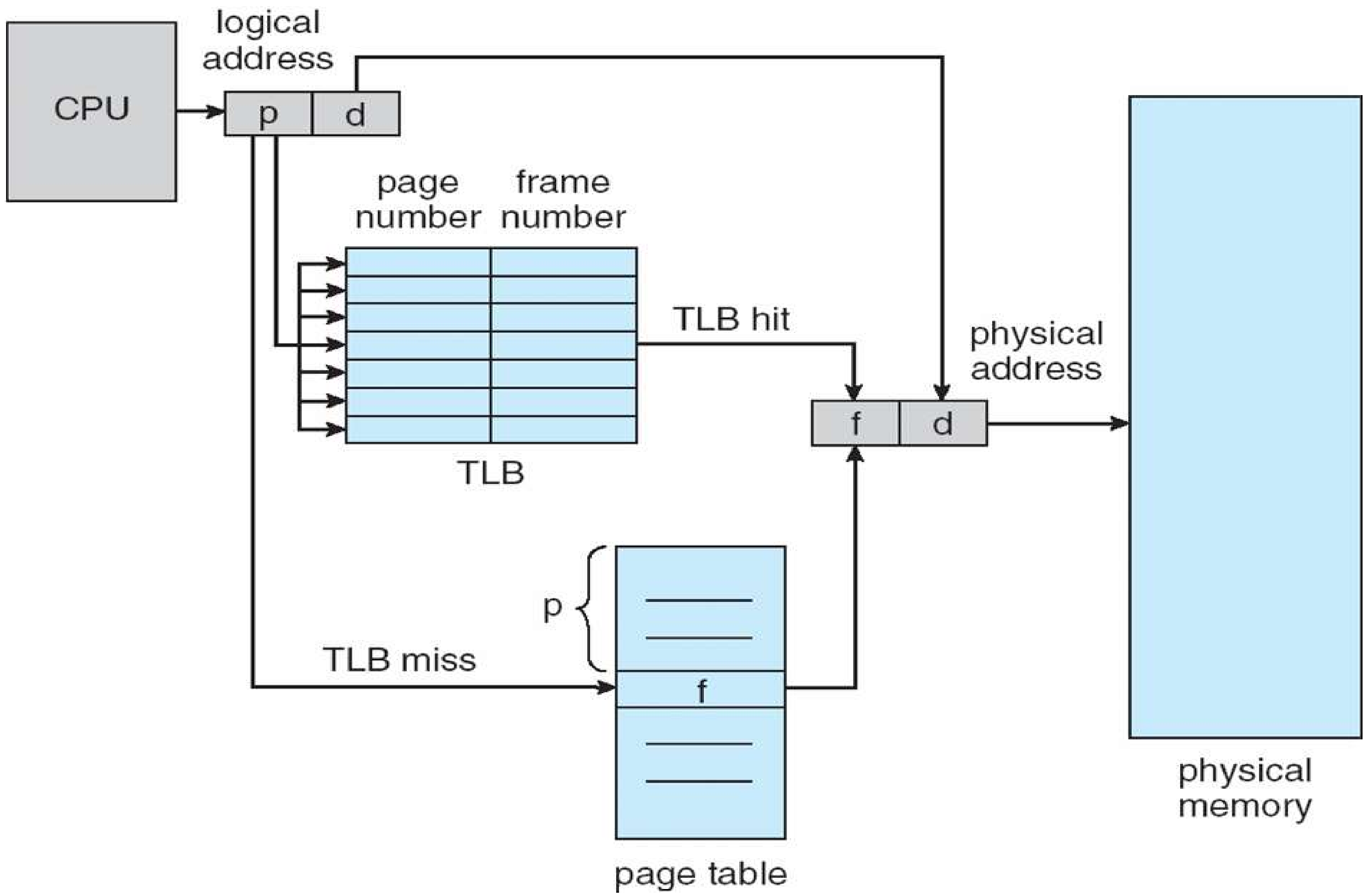
- With this scheme, two memory accesses are needed to access data(one for the page-table entry and one for the actual data)
- Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances
- The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a translation look-aside buffer (TLB)

Translation Look-Aside Buffer

- The TLB is associative, high-speed memory
- Each entry in the TLB consists of two parts:
  - a key (or tag) and
  - a value
- When the associative memory is presented with an item, the item is compared with all keys simultaneously
- If the item is found, the corresponding value field is returned
- The search is fast

- The TLB is used with page tables in the following way
  - The TLB contains only a few of the page-table entries
  - When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB
  - If the page number is found, its frame number is immediately available and is used to access memory
  - If the page number is not in the TLB (known as a TLB miss) then memory reference to the page table must be made
  - When the frame number is obtained, we can use it to access memory
  - In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference

Paging Hardware With TLB



- If the TLB is already full of entries, an existing entry must be selected for replacement
- Replacement policies range from least recently used (LRU) through round-robin to random
- Furthermore, some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB
- Typically, TLB entries for key kernel code are wired down

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry
  - uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- The percentage of times that the page number of interest is found in the TLB is called the hit ratio
- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time

- If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB
- If we fail to find the page number in the TLB then
  - we must first access memory for the page table and frame number (10 nanoseconds) and
  - then access the desired byte in memory (10nanoseconds),for a total of 20 nanoseconds

- To find the effective memory-access time, we weight the case by its probability:
- effective access time = 0.80×10 + 0.20×20 = 12 nanoseconds
- For a 99-percent hit ratio
  - effective access time = 0.99×10 + 0.01×20 = 10.1 nanoseconds
- CPUs today may provide multiple levels of TLBs
- A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier

Questions

- Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames
    a. How many bits are required in the logical address?
    b. How many bits are required in the physical address?
- Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
    a. 3085
    b. 42095

Answer

- Logical address space = 256 pages with 4KB page size
  - Representing in powers of 2, 256 pages=2^8
  - Each page size=4KB=4096=2^{12}
  - Logical address space requires 2^8 X 2^{12} =2^{20}=20 bits
- Physical address=64 frames with 4KB frame size
  - Representing in powers of 2, 64 frames =2^6
  - Each frame size=4KB=4096=2^{12}
  - Physical address space requires 2^6 X 2^{12} =2^{18}=18 bits

Answer

- Logical address space is divided into
  - Page number and offset
- Page size= 1KB=1024 bytes= 2^{10} bytes=10 bits
- Consider a 32- bit logical address space, then page number =22 bits and offset =10 bits
- Logical address (decimal)=3085
- Logical address (binary)(32 bits) = 00000000000000000000110000001101
- Page number (binary)(22 bits)=00000000000000000011

- Offset (binary) (10 bits)-0000001101
- Page Number in decimal- 3
- Offset in decimal- 13

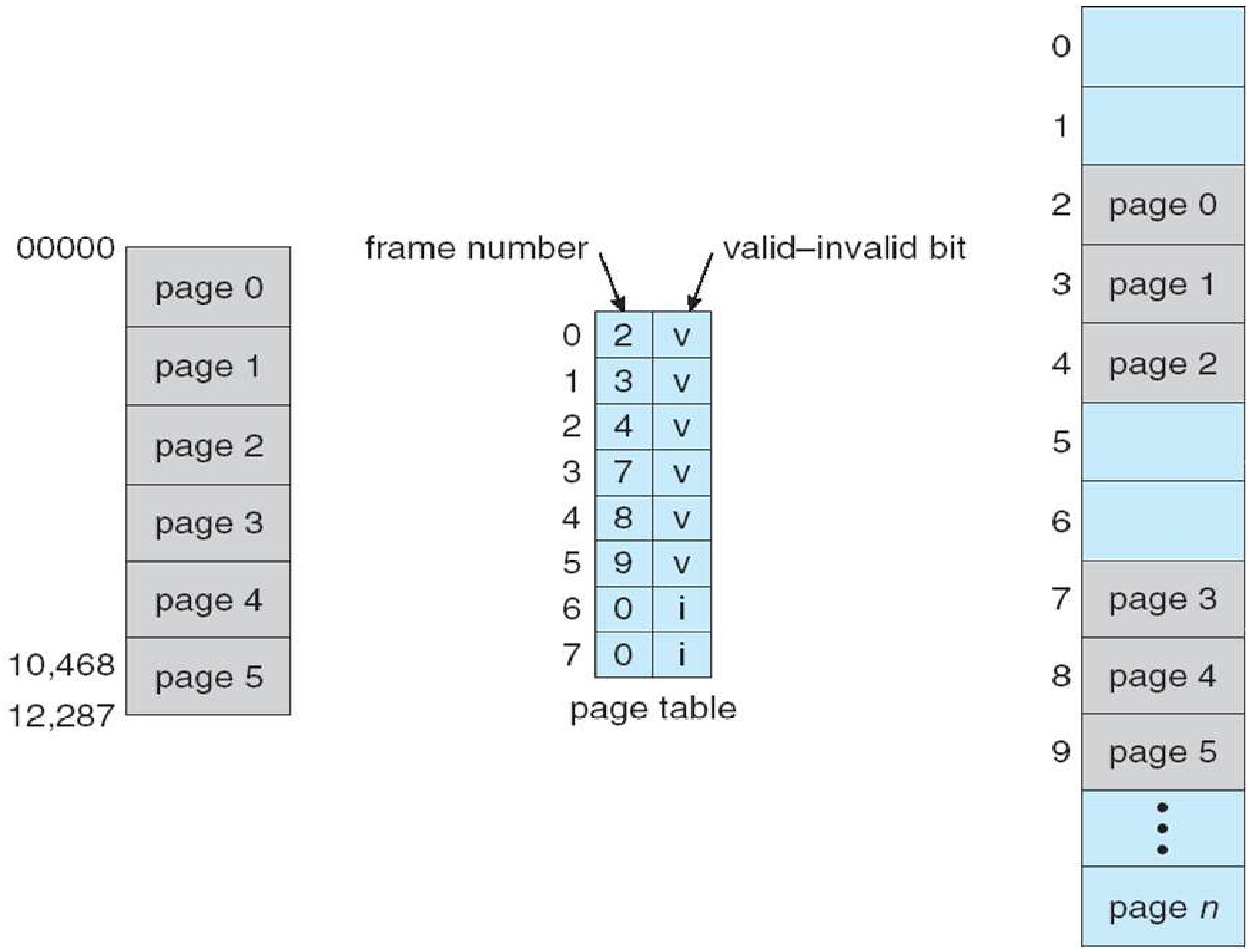
Protection

- Memory protection in a paged environment is accomplished by protection bits associated with each frame
- Normally, these bits are kept in the page table
- One bit can define a page to be read–write or read-only
- Every reference to memory goes through the page table to find the correct frame number

- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page
- An attempt to write to a read-only page causes a hardware trap to the operating system or memory-protection violation
- 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
- Illegal addresses are trapped by use of the valid–invalid bit
- The operating system sets this bit for each page to allow or disallow access to the page

- Example
  - Suppose that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468
  - Given a page size of 2KB, we have the situation shown in Figure in the next slide
  - Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table
  - Any attempt to generate an address in pages 6 or 7, however, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system(invalid page reference)


00000
page 0
page 1
page 2
page 3
page 4
page 5
10,468
12,287
frame number
valid–invalid bit
0 2 v
1 3 v
2 4 v
3 7 v
4 8 v
5 9 v
6 0 i
7 0 i
page table
0
1
2 page 0
3 page 1
4 page 2
5
6
7 page 3
8 page 4
9 page 5
page n

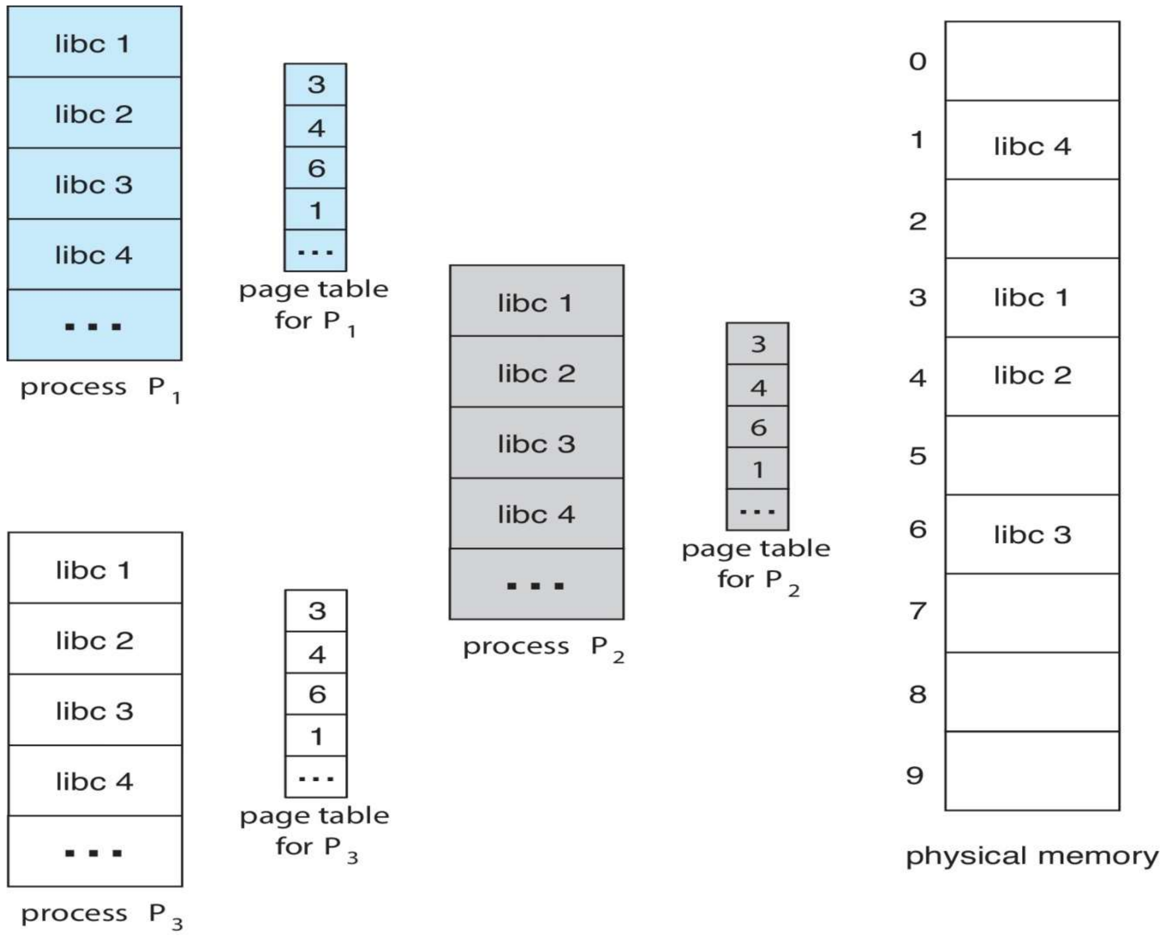
- Notice that this scheme has created a problem
- Because the program extends only to address 10468, any reference beyond that address is illegal
- However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid
- Only the addresses from 12288 to 16383 are invalid

Shared Pages

- An advantage of paging is the possibility of sharing common code, a consideration that is particularly important in an environment with multiple processes
- Consider the standard C library, which provides a portion of the system call interface for many versions of UNIX and Linux
- On a typical Linux system, most user processes require the standard C library libc
- One option is to have each process load its own copy of libc into its address space

- If a system has 40 user processes, and the libc library is 2 MB, this would require 80 MB of memory
- If the code is reentrant code, however, it can be shared, as shown in Figure in the next slide
- Here, three processes are sharing the pages for the standard C library libc
- Reentrant code is non-self-modifying code: it never changes during execution

Shared Pages



- Thus, two or more processes can execute the same code at the same time
- Each process has its own copy of registers and data storage to hold the data for the process’s execution
- The data for two different processes will, of course, be different
- Only one copy of the standard C library need be kept in physical memory

- The page table for each user process maps onto the same physical copy of libc
- Thus, to support 40 processes, only one copy of the library is needed and the total space now required is 2 MB instead of 80 MB
- In addition to run-time libraries such as libc, other heavily used programs can also be shared—compilers, window systems, database systems, and so on

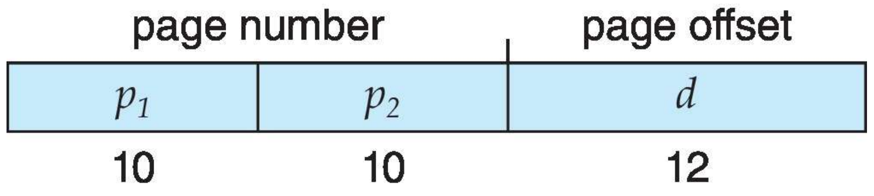
Structure of the Page Table

- The most common techniques for structuring the page table
  - Hierarchical paging
  - Hashed page table
  - inverted page table

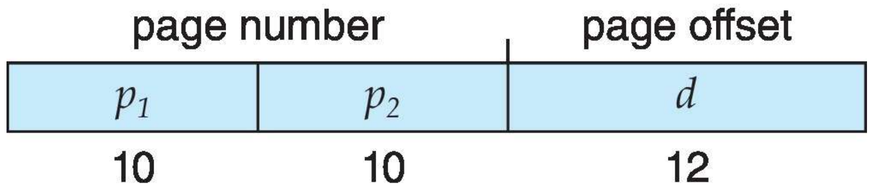
Hierarchical Paging

- Most modern computer systems support a large logical address space (2^{32} to 2^{64})
- In such an environment, the page table itself becomes excessively large
- For example, consider a system with a 32-bit logical address space
- If the page size in such a system is 4 KB (2^{12}), then a page table may consist of over 1 million entries ($2^{20} = 2^{32}/2^{12}$)

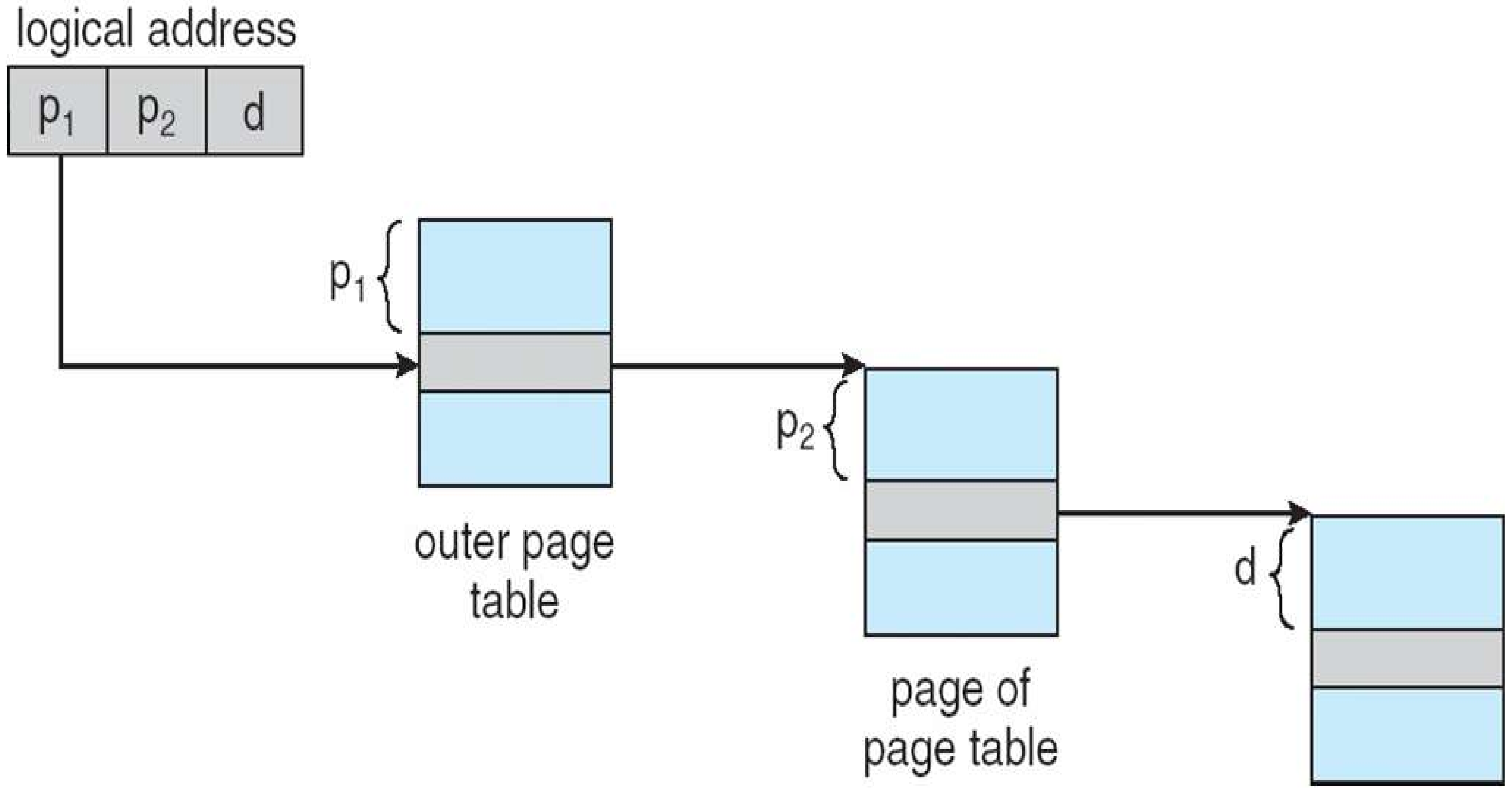
- Assuming that each entry consists of 4 bytes, each process may need upto 4 MB of physical address space for the page table alone
- One simple solution to this problem is to divide the page table into smaller pieces
- One way is to use a two-level paging algorithm, in which the page table itself is also paged (see the figure below)



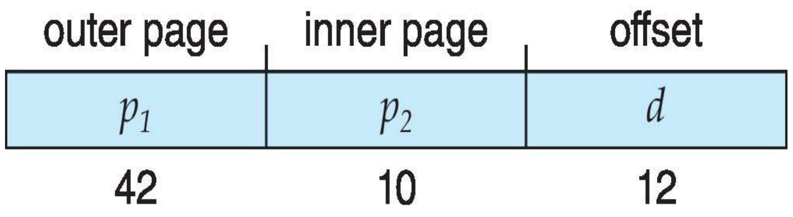
- For example, consider again the system with a 32-bit logical address space and a page size of 4 KB
- A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits
- Because page table is paged ,the page number is further divided into a10-bit page number and a 10-bit page offset
- Thus, a logical address is as follows:



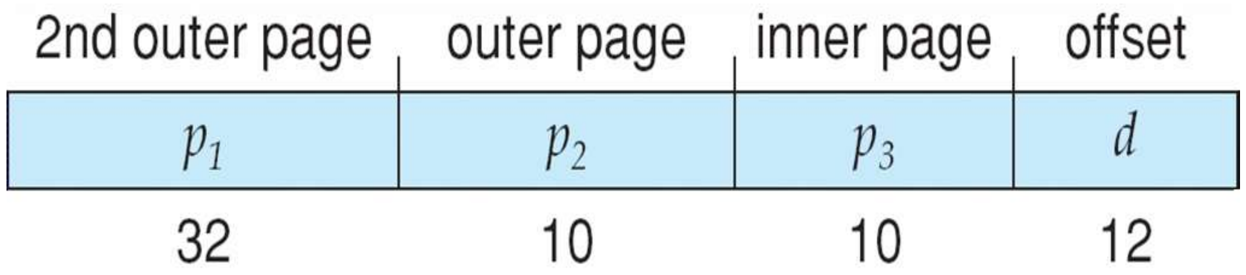
- where p_1 is an index into the outer page table
- p_2 is the displacement with in the page of the inner page table
- The address-translation method for this architecture is shown in Figure next slide
- Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table


logical address
p_1
p_2
d
p_1
outer page table
p_2
page of page table
d

- For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate
- Suppose that the page size in such a system is 4 KB (2^{12})
- In this case, the page table consists of up to 2^{52} entries
- If we use a two-level paging scheme, then the inner page tables can conveniently be one page long ,or contain 2^{10} 4-byte entries
- The addresses look like this:



- The outer page table consists of 2^{42} entries, or 2^{44} bytes
- The outer page table can be divided in various ways
- For example, we can page the outer page table, giving us a three-level paging scheme
- Suppose that the outer page table is made up of standard-size pages(2^{10} entries, or 2^{12} bytes)



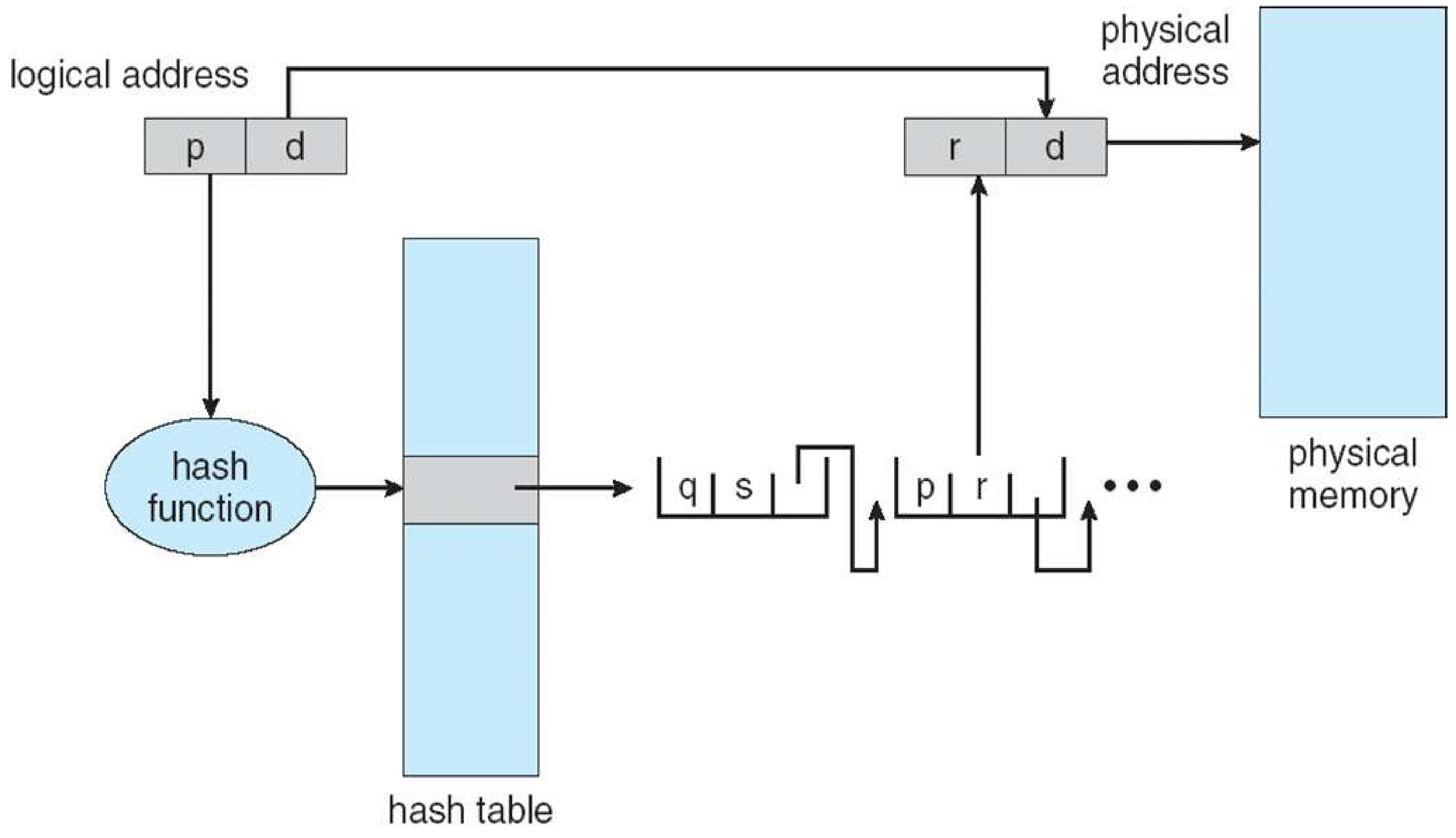
- The outer page table is still 2^{34} bytes (16 GB) in size
- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth
- The 64-bit UltraSPARC would require seven levels of paging which is a prohibitive number of memory accesses to translate each logical address

Hashed Page Tables

- One approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number
- Each entry in the hash table contains a linked list of elements that hash to the same location
- Each element consists of three fields:
  1. The virtual page number,
  2. The value of the mapped page frame,
  3. A pointer to the next element in the linked list

- The algorithm works as follows:
  - The virtual page number in the virtual address is hashed into the hash table
  - The virtual page number is compared with field 1 in the first element in the linked list
  - If there is a match, the corresponding page frame (field2) is used to form the desired physical address
  - If there is no match, subsequent entries in the linked list are searched for a matching virtual page number
- This scheme is shown in Figure in the next slide

Hashed Page Table



- A variation of this scheme that is useful for 64-bit address spaces has been proposed
- This variation uses clustered page tables, which are similar to hashed page tables
  - except that each entry in the hash table refers to several pages (such as 16) rather than a single page
  - Therefore, a single page-table entry can store the mappings for multiple physical-page frames
  - Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space

Inverted Page Tables

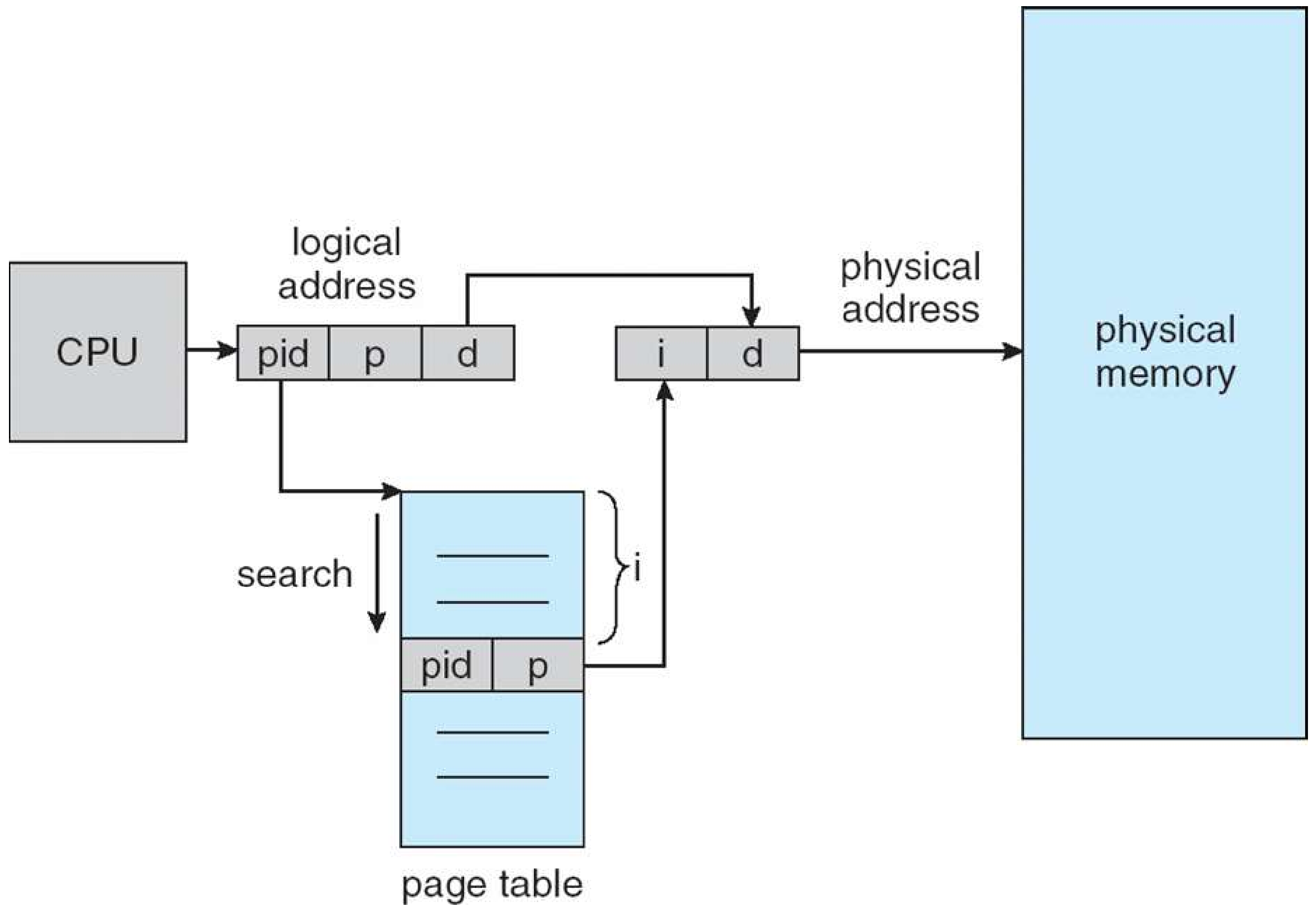
- Drawbacks of page table
  - The page table has one entry for each page that the process is using
  - Each page table may consist of millions of entries
  - These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used
- To solve this problem, an inverted page table is used
- An inverted page table has one entry for each real page (or frame) of memory

- Only one page table is in the system, and it has only one entry for each page of physical me
- To illustrate this method, a simplified version of the inverted page table used in the IBM RT is described
- For the IBM RT, each virtual address in the system consists of a triple:
  - <process-id, page-number, offset>
- Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier

- When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, is presented to the memory subsystem
- The inverted page table is then searched for a match
- If a match is found at entry i then the physical address

  <i, offset> is generated
- If no match is found, then an illegal address access has been attempted

Inverted Page Table



Question

- The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?

a) A conventional, single-level page table

b) An inverted page table

c) What is the maximum amount of physical memory in the BTV operating system?

Answer

a. Virtual address space =21 bits=2^{21}

Page size=2KB=2048 bytes=2^{11}

Number of entries =2^{21} / 2^{11} = 2^{10} =1024

b. Physical address= 16 bits = 2^{16}

Page size=2KB=2048 bytes=2^{11}

In inverted page table, the page table will have entries equal to the size of physical memory

Therefore, number of entries= 2^{16} / 2^{11} = 2^{5} = 32

c. Maximum amount of physical memory in the BTV operating system

Physical address= 16 bits = 2^{16} = 65536 or 64KB

- Decreases the amount of memory needed to store each page table
- Increases the amount of time needed to search the table when a page reference occurs
- Use hash table to limit the search to one or at most a few page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

Swapping

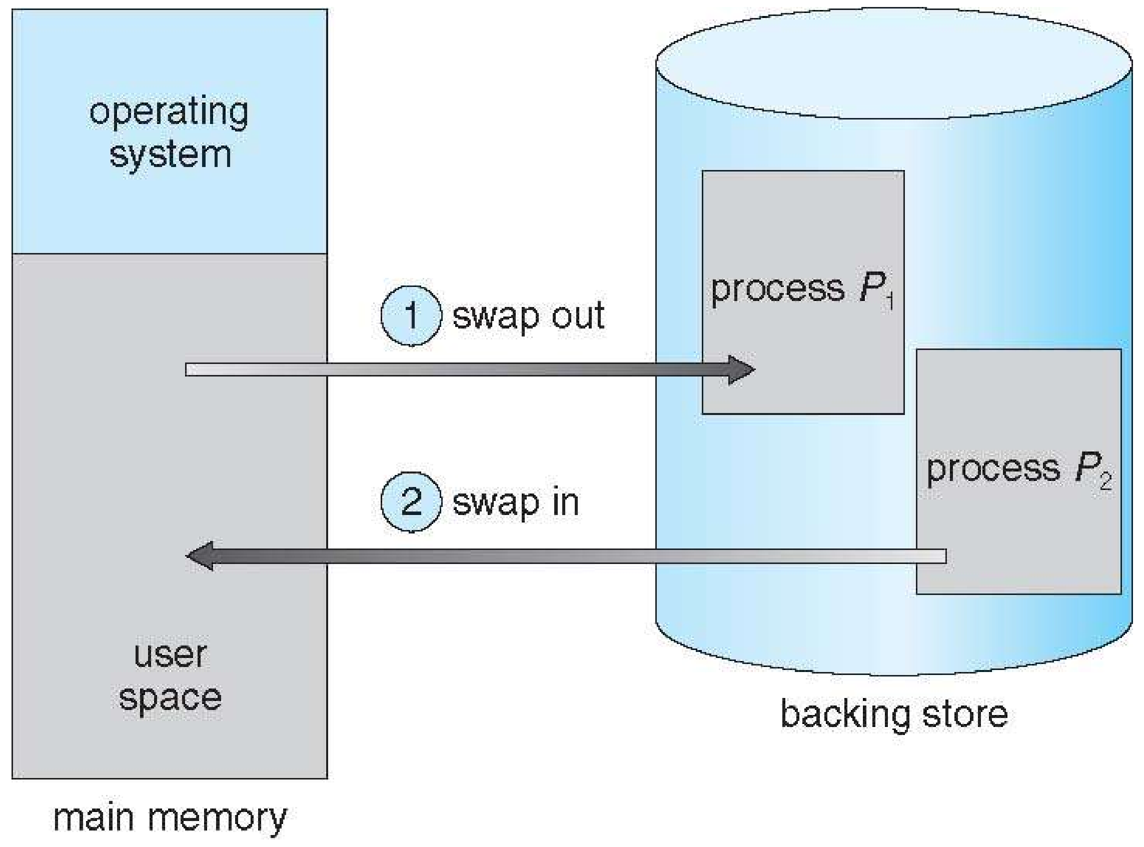
- Process instructions and the data they operate on must be in memory to be executed
- However, a process, or a portion of a process,
  - can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution
- Swapping makes it possible for
  - the total physical address space of all processes to exceed the real physical memory of the system thus increasing the degree of multiprogramming in a system

Standard Swapping

- Standard swapping involves moving entire processes between main memory and a backing store
- The backing store is commonly fast secondary storage
  - It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and
  - It must provide direct access to these memory images
- When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store

- For a multithreaded process, all per-thread data structures must be swapped as well
- The operating system must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back in to memory
- The advantage of standard swapping is that
  - It allows physical memory to be oversubscribed, so that the system can accommodate more processes than there is actual physical memory to store them

Standard Swapping



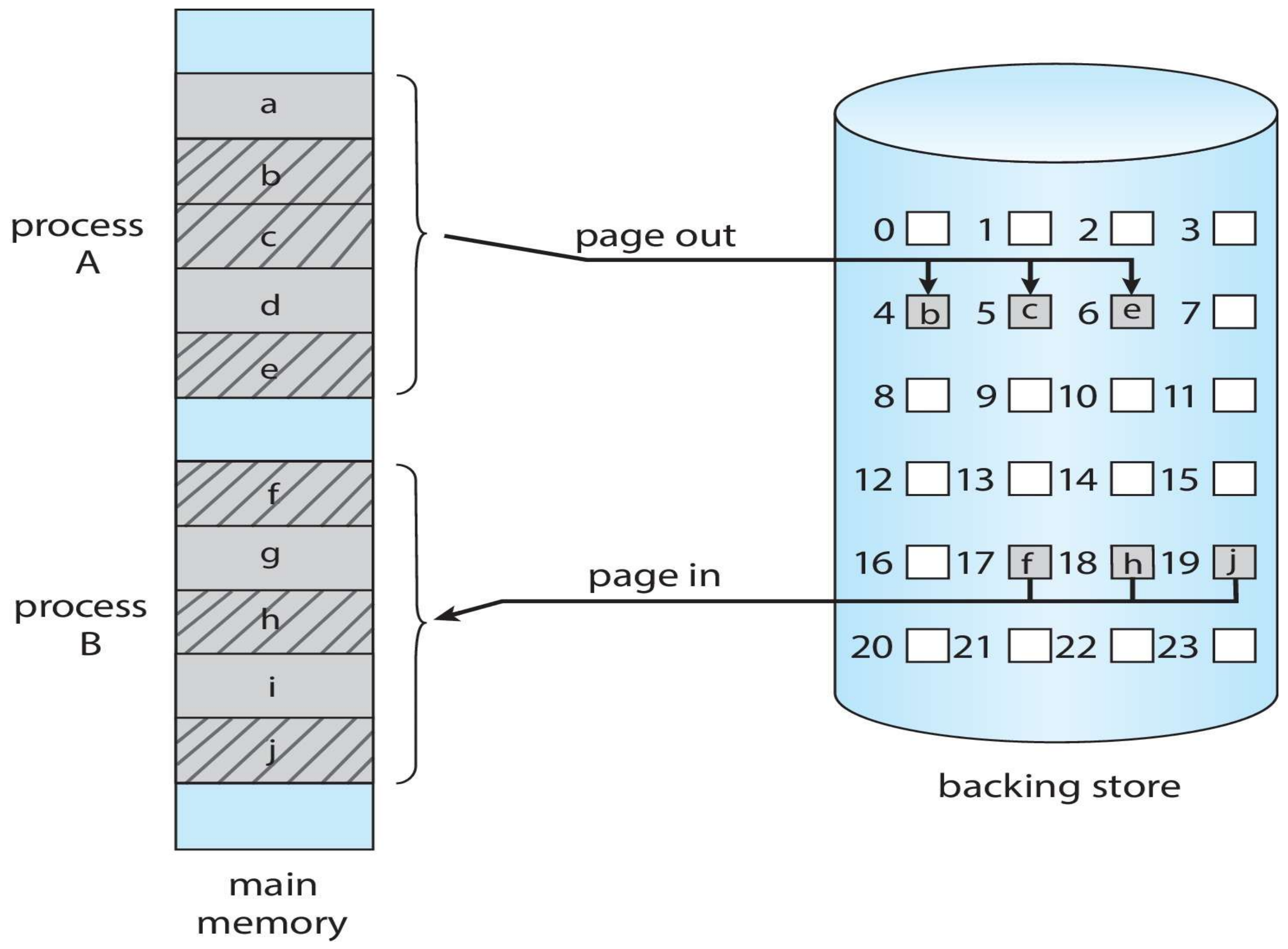
- Idle or mostly idle processes are good candidates for swapping
- Any memory that has been allocated to these inactive processes can then be dedicated to active processes
- If an inactive process that has been swapped out becomes active once again, it must then be swapped back in

Swapping with Paging

- Standard swapping was used in traditional UNIX systems, but it is generally no longer used in contemporary operating systems
  - because the amount of time required to move entire processes between memory and the backing store is prohibitive
- Most systems, including Linux and Windows, now use a variation of swapping in which pages of a process rather than an entire process can be swapped

- This strategy still allows physical memory to be over subscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping
- In fact, the term swapping now generally refers to standard swapping, and paging refers to swapping with paging
- A page out operation moves a page from memory to the backing store
- The reverse process is known as a pagein

Swapping with paging



a
b
c
d
e
f
g
h
i
j
process A
process B
main memory
page out
page in
0 1 2 3
4 b 5 c 6 e 7
8 9 10 11
12 13 14 15
16 17 f 18 h 19 j
20 21 22 23
backing store