(CC-311) Operating System

Lecture: 12 & 13

Professor: Syed Mustaghees Abbas

Memory Management

Review

- Contiguous Memory Allocation (*fixed-sized partitions, MVT*)
- Best Fit, Worst Fit, First Fit
- Fragmentation (Internal and External)
- Paging
- Hardware Support (Set of dedicated registers, PTBR, TLB)

Protection in Paging

- Memory protection in a paged environment is accomplished by protection bits associated with each frame.

- Normally, these bits are kept in the page table.

- A bit defines a page to be read-write or read-only.

- Every reference to memory is 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).

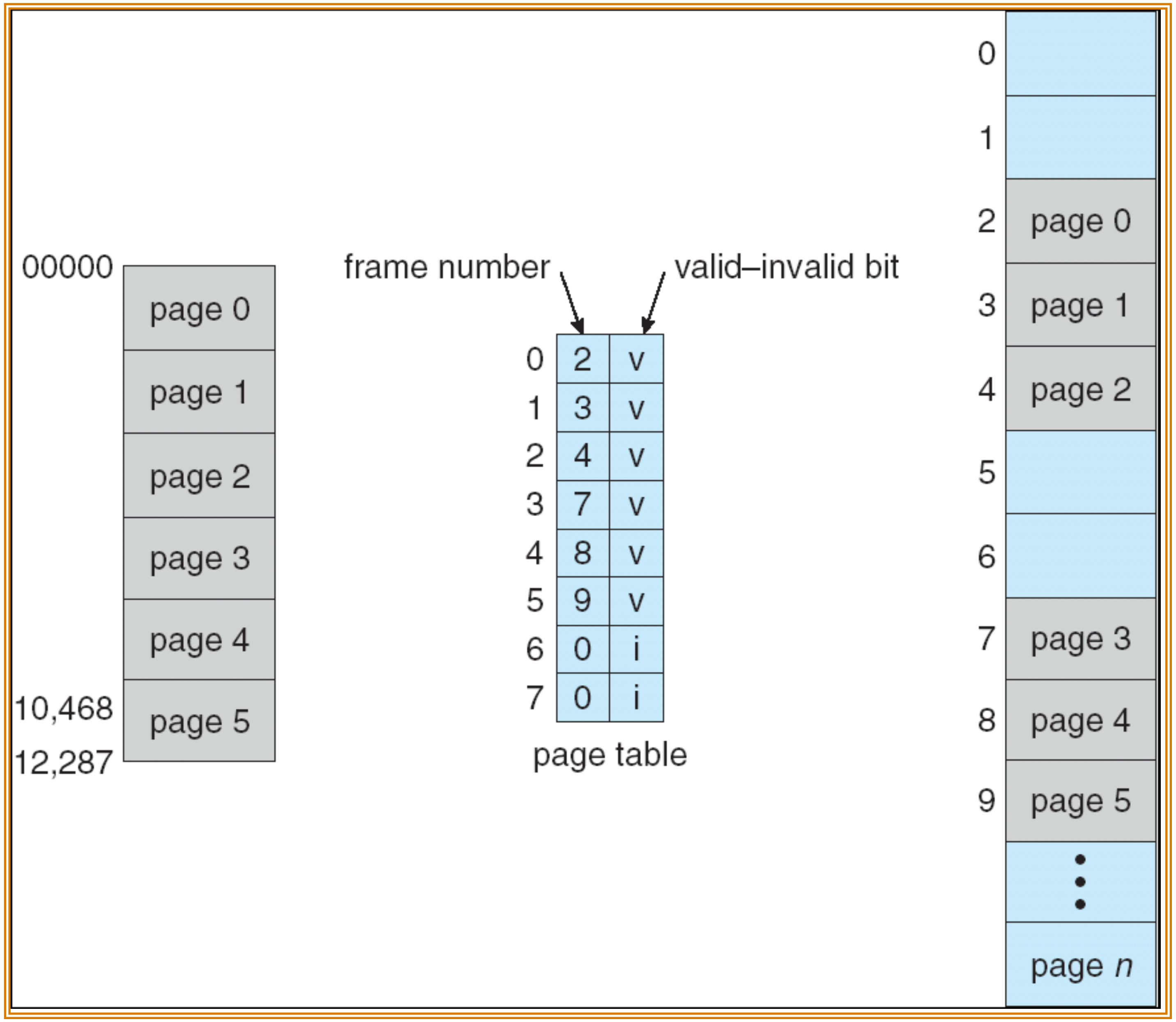
Protection in Paging(cntd)

- A **valid-invalid bit** is generally attached to each entry in the page table.
- 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.

Protection in Paging(cntd)

- In a system with a 14-bit address space (0 to 16383), if we have a program that should use only addresses 0 to 10468.
- Given a page size of 2 KB, 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).

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



Problem

- This scheme has a problem that although the program extends to 10468, accesses to addresses up to 12287 are valid.
- Only the addresses from 12288 to 16383 are invalid.
- This *problem* is a result of the 2-KB page size and reflects the internal fragmentation of paging.

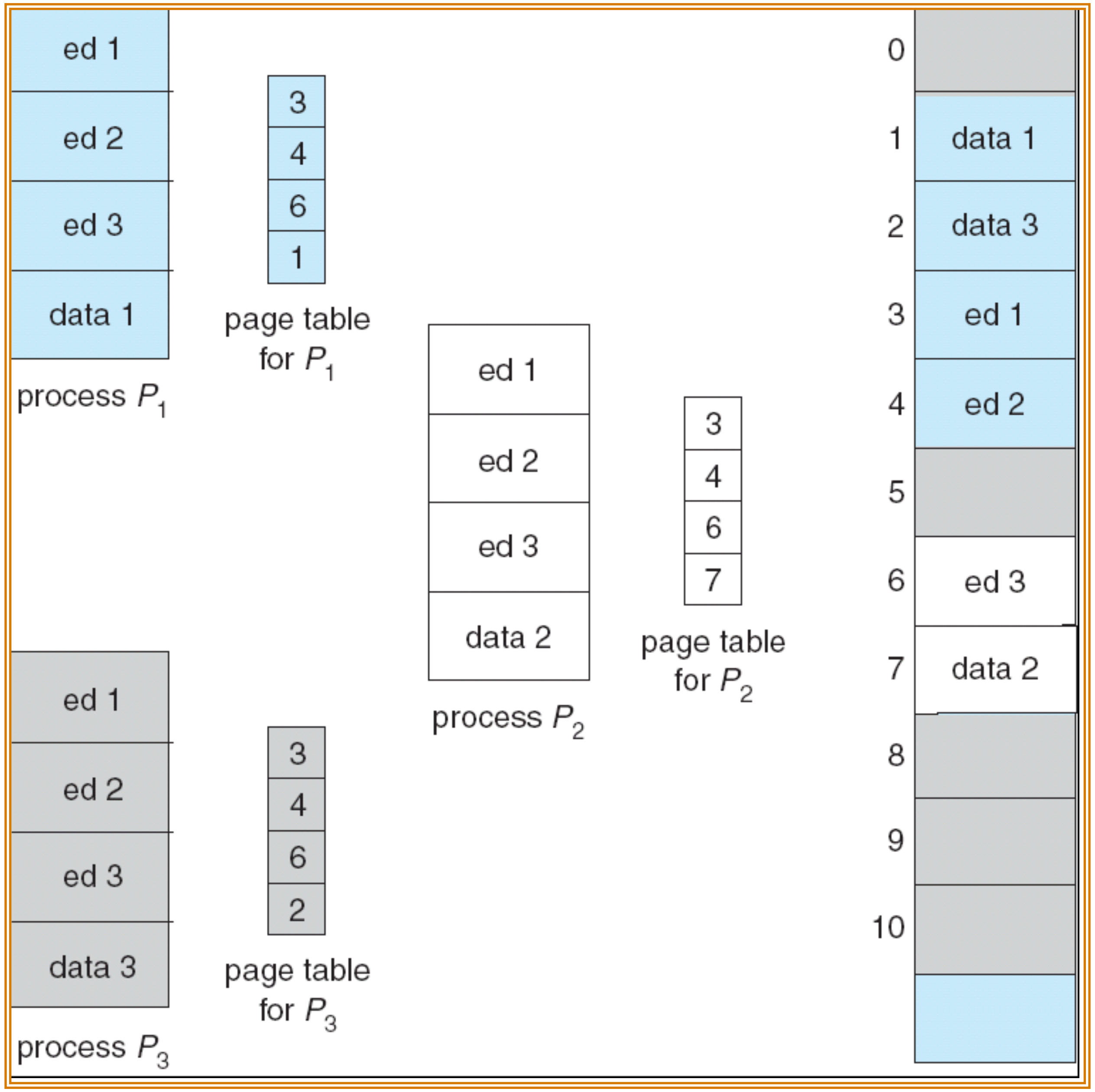
Shared Pages

- An advantage of paging is the possibility of sharing *common code*.
- Consider a system with 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.
- If the code is **reentrant code (or pure code)**, however, it can be shared. *Reentrant code* is non-self-modifying code; it never changes during execution.
- Here we see a three-page editor—each page 50 KB in size being shared among three processes. Each process has its own data page.

Shared Pages(cntd)

- 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.
- 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.
- Now 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.

Shared Pages(cntd)



Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Most modern computer systems support a large logical address space (2^{32} to 2^{64}). Hence 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 up to 1 million entries ($2^{32}/2^{12}$).

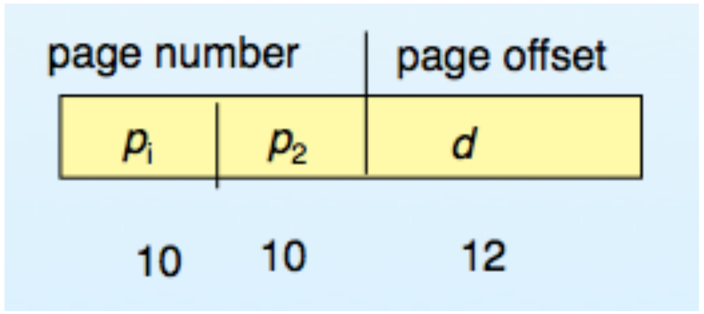
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- Clearly, we would not want to allocate the page table *contiguously* in main memory.

- One simple solution to this problem is to divide the page table into smaller pieces.

- A simple technique is a two-level page table

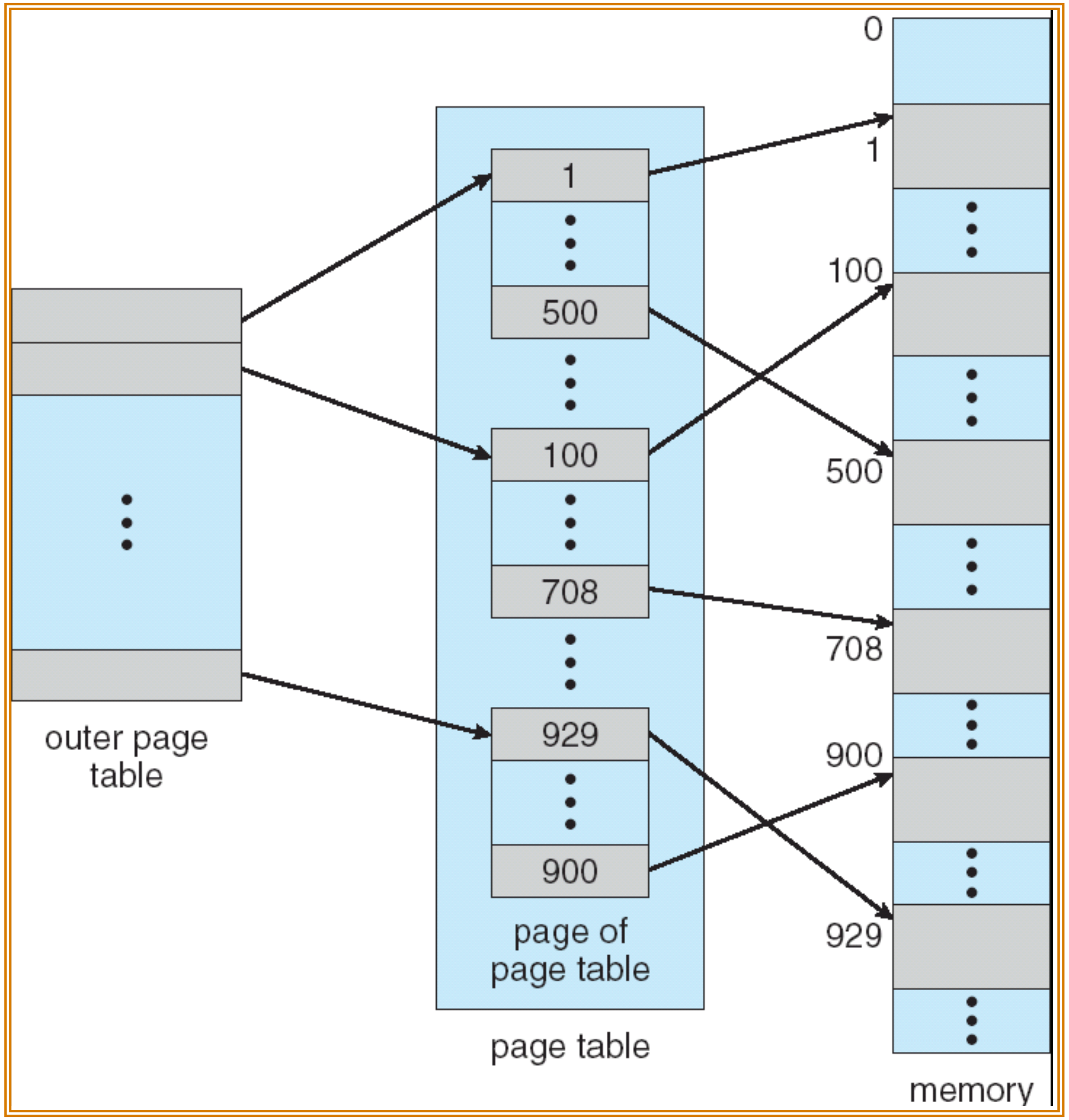
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

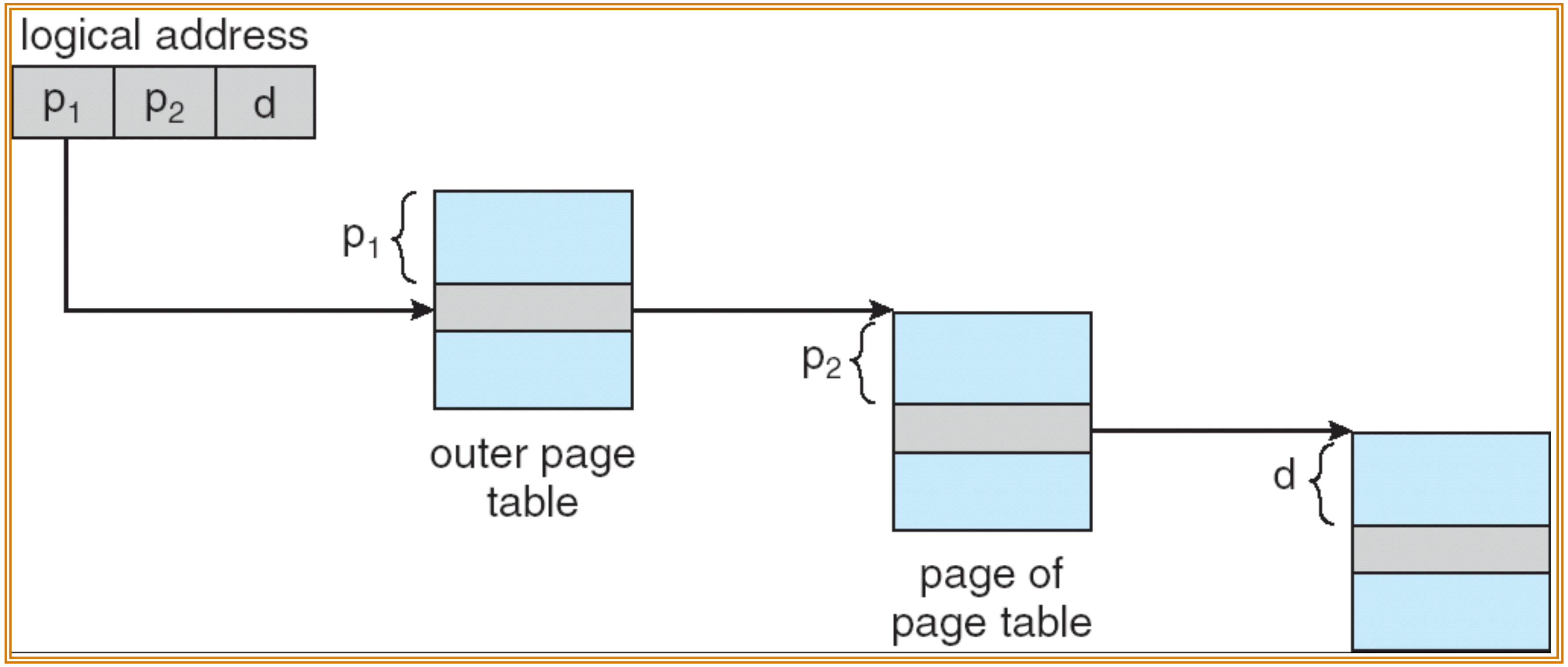


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Paging Example

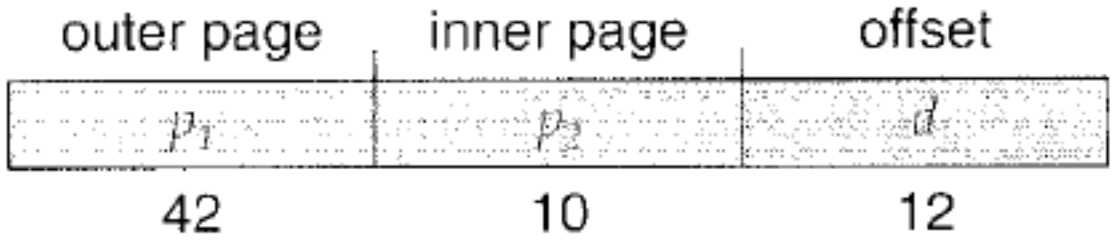


Address-Translation Scheme



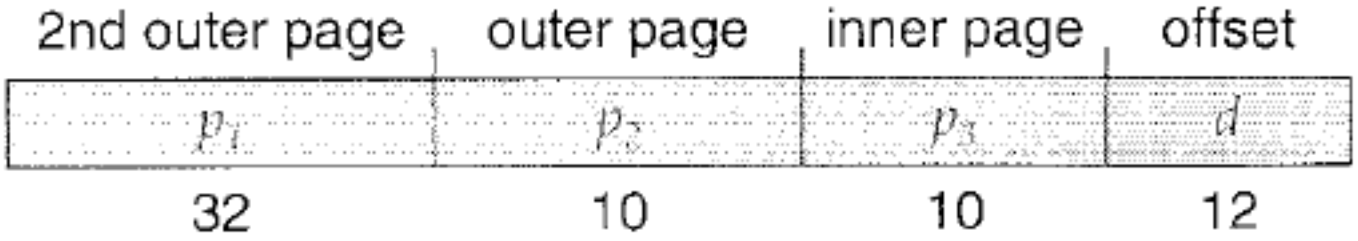
Hierarchical Page Tables(cntd)

- This scheme is also known as a **forward-mapped page table** because address translation works from the outer page table inward.
- 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 (without hierarchical paging).
- 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.



Hierarchical Page Tables(cntd)

- The outer page table consists of 2^{42} entries. Still too much entries.
- The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.



- The outer page table still has 2^{32} entries. The next step would, be a four-level paging scheme, where the second level outer page table itself is also paged.

- For 64-bit architectures, hierarchical page tables are generally considered inappropriate due to *prohibitive number of memory accesses*—to translate each logical address.

Hashed Page Tables

- A common 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 (to handle collisions).

- 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.

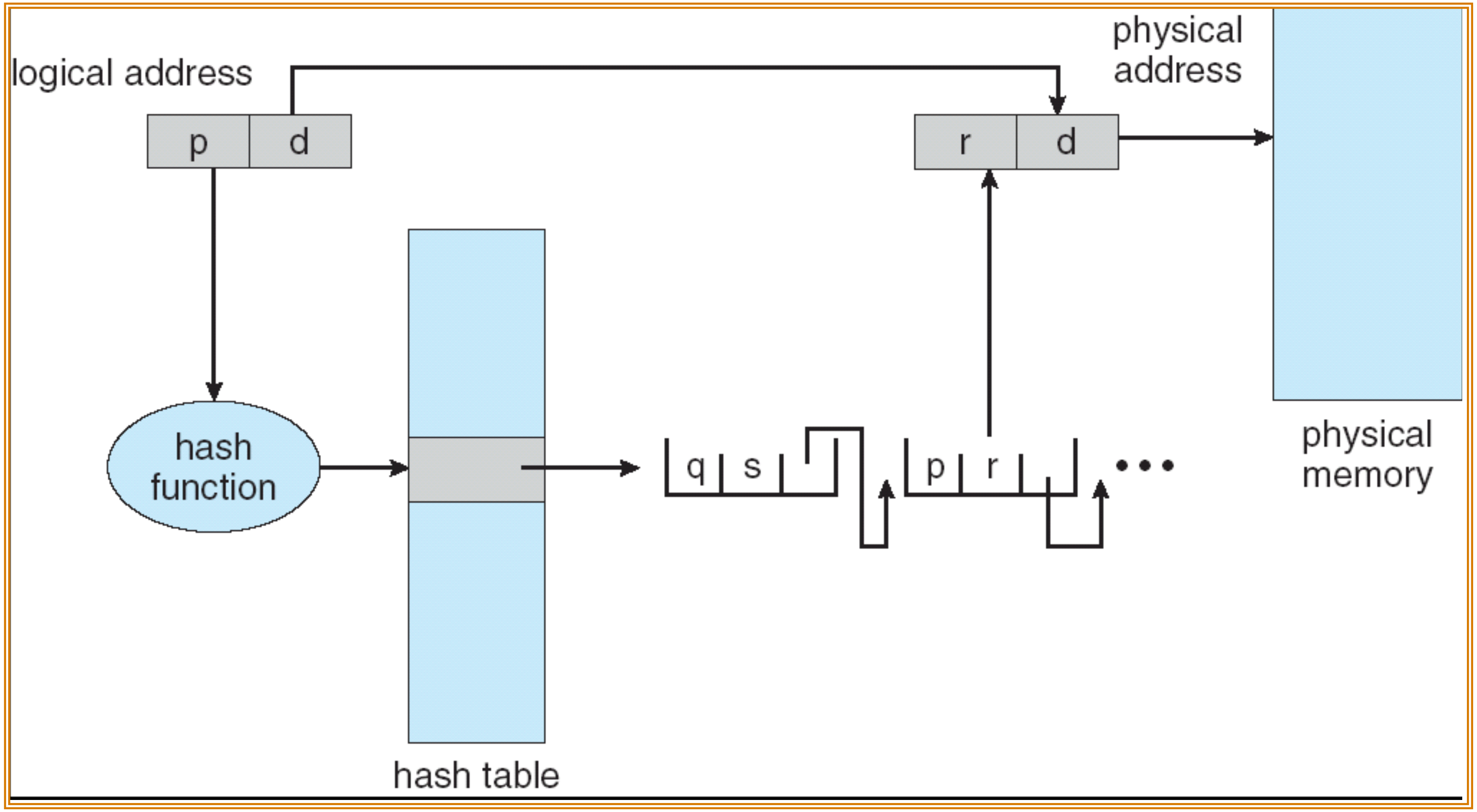
- The algorithm works as follows:
  1. The virtual page number in the virtual address is hashed into the hash table.
  2. The virtual page number is compared with field 1 in the first element in the linked list.

Hashed Page Tables(cntd)

- If there is a match, the corresponding page frame (field 2) 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.

Hashed Page Tables(cntd)



Inverted Page Tables

- Usually, each process has an associated page table. 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, we can use an inverted page table.
- An inverted page table has one entry for each frame of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Thus, only one page table is in the system, and it has only one entry for each frame of physical memory.

Inverted Page Tables (cntd)

- Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

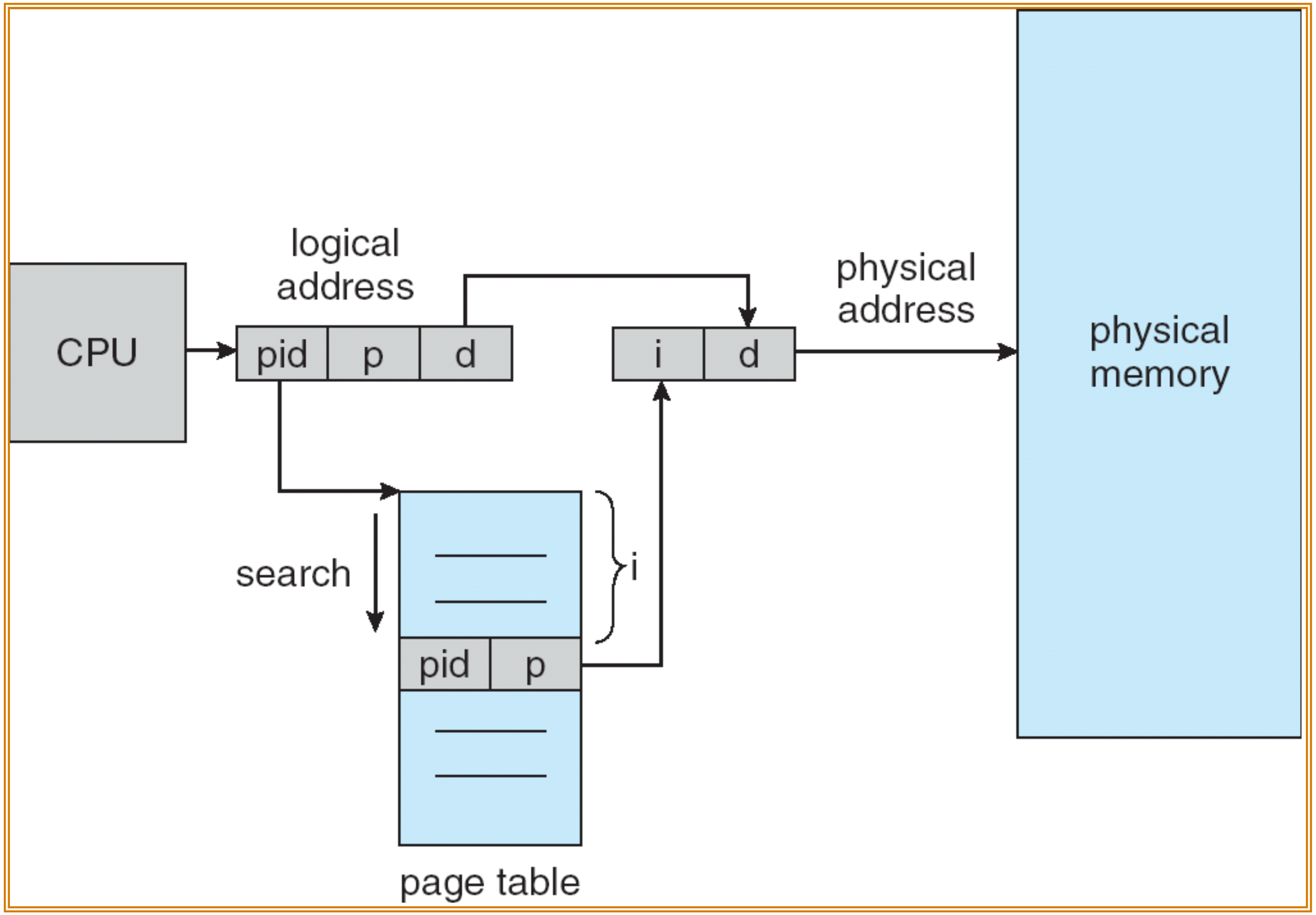
- This scheme increases the amount of time needed to search the table when a page reference occurs.

- The whole table might need to be searched for a match.

- To alleviate this problem, we use a hash table to limit the search to one—or at most a few—page-table entries.

- Systems that use inverted page tables have difficulty implementing shared memory.
-
- Shared memory is usually implemented as multiple virtual addresses that are mapped to one physical address. This standard method cannot be used with inverted page tables.

Inverted Page Table Architecture


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table