Operating System Concepts

Lecture 26: Page Table Structure & Demand Paging

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Dealing with large page tables
  - hierarchical
  - hashed
  - inverted
- Demand paged virtual memory

Paging in modern computer systems

memory structures for paging can get huge using straight-forward methods for modern systems that support a larger address space (2^{32} to 2^{64})

- consider a 32-bit logical address space
  - page size of 4KB (2^{12})
  - page table would have 1 million entries (2^{32} / 2^{12})

page no.	offset
20 bits	12 bits

- if each entry is 4 bytes then we need 4MB of physical address space for storing the page table alone
  - it is larger than the page size and we don't want to allocate that contiguously in main memory
  - in this case, we cannot use more than 10 bits for the page number if we want to fit the entire page table into one page

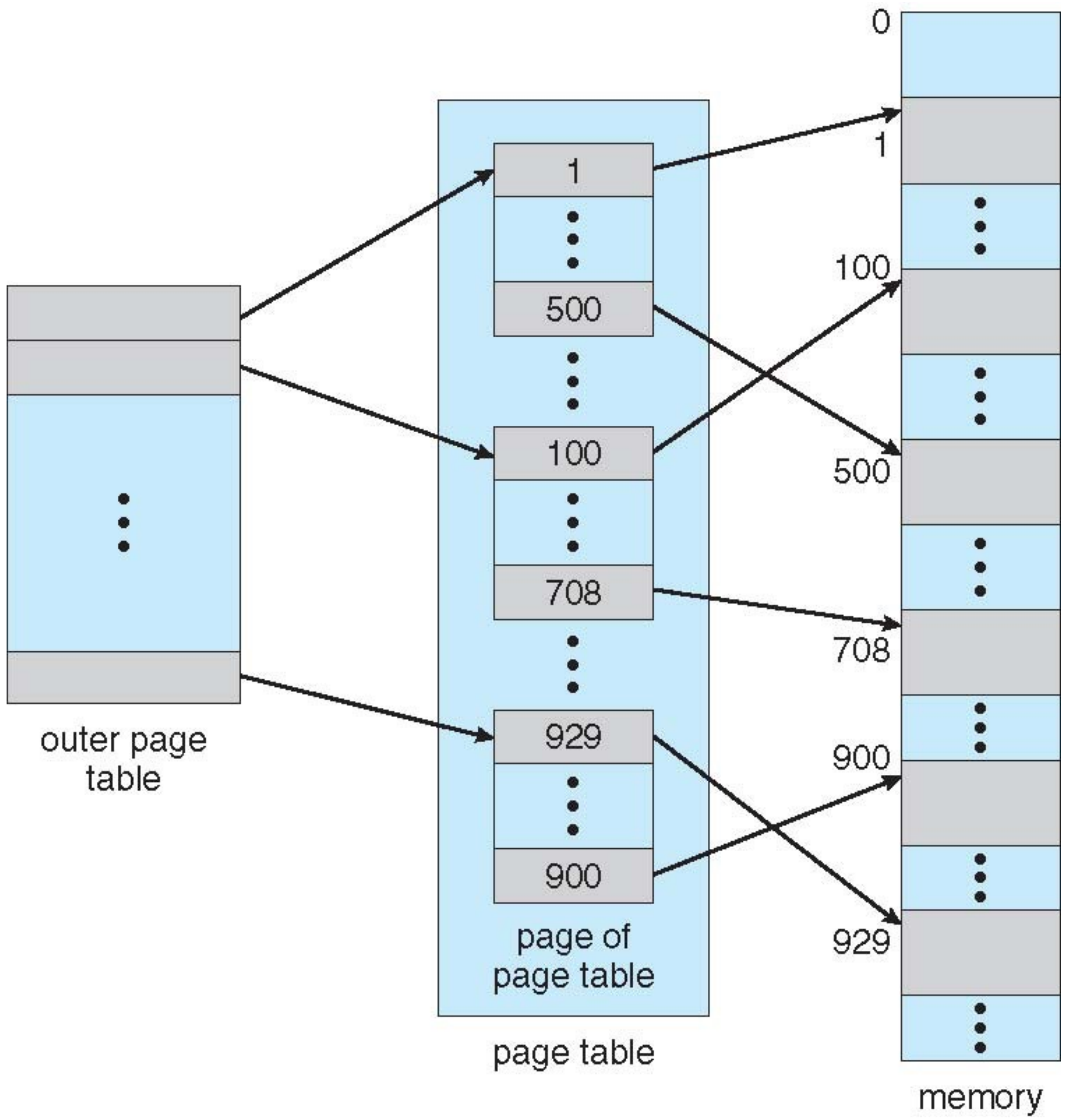
?	page no.	offset
10 bits	10 bits	12 bits

Structure of the page table

- a simple solution is to divide the page table into smaller units
    - hierarchical Paging
    - hashed Page Tables
    - inverted Page Tables

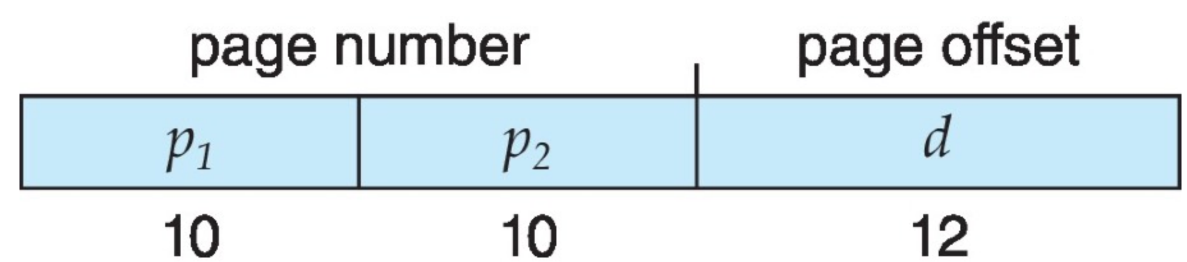
Hierarchical page table

- break up the logical address space into multiple page tables
- a simple technique is a two-level page table
  - we page the page table
  - this requires two lookups per memory reference



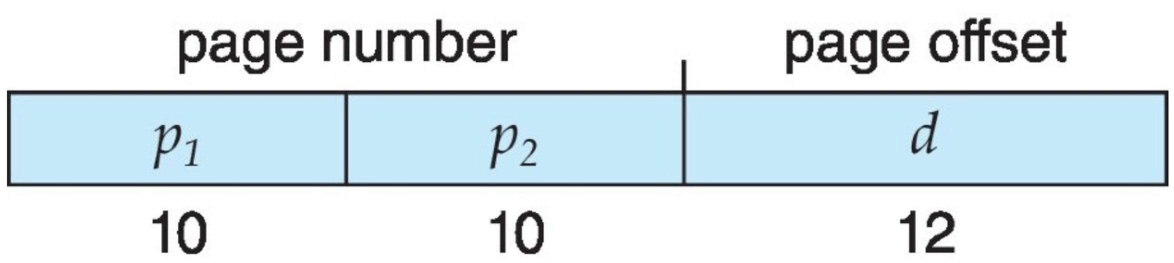
Hierarchical page table: example

- consider a 32-bit logical address space with the page size of 4KB (2^{12})
  - since the page table is paged, the page number is further divided into:
    - a 10-bit page number
    - a 12-bit page offset

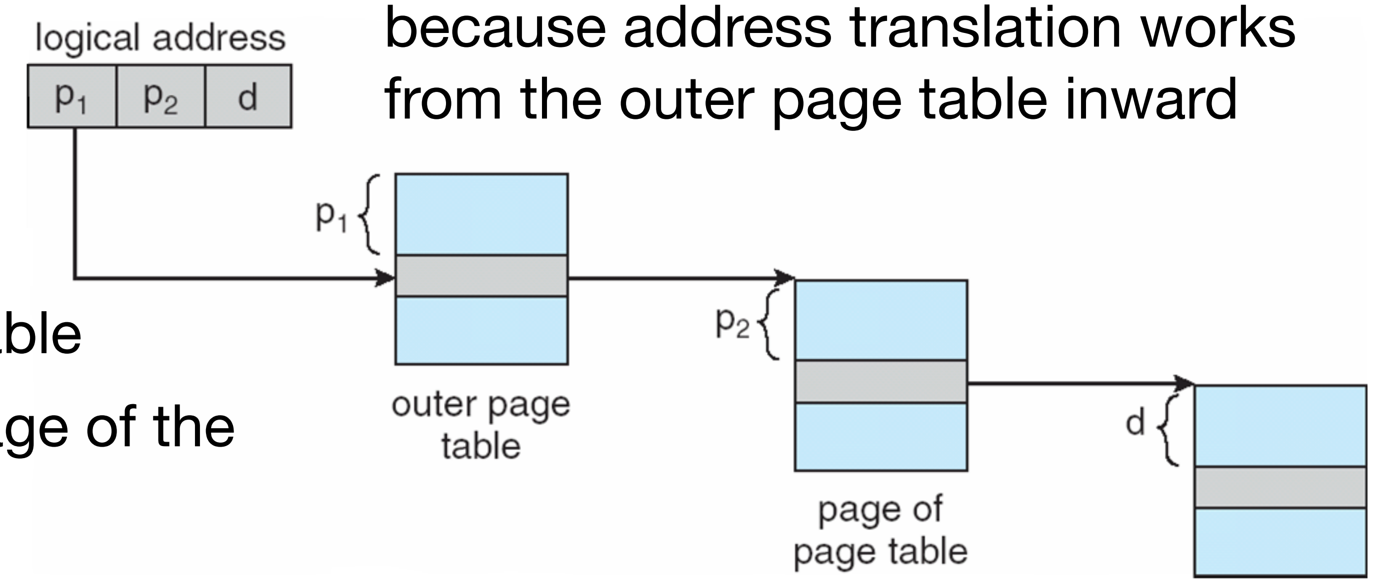


Hierarchical page table: example

- consider a 32-bit logical address space with the page size of 4KB (2^{12})
  - since the page table is paged, the page number is further divided into:
    - a 10-bit page number
    - a 12-bit page offset



known as forward-mapped page table because address translation works from the outer page table inward



p_1 is an index into the outer page table

p_2 is the displacement within the page of the inner page table

d is the page offset

What about a 64-bit logical address space?

?	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

What about a 64-bit logical address space?

?	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

should we page the outer page too (three-level paging scheme)? but the 2nd outer page table will be 2^{34} bits (16 GB) in size and can't fit in one page

2nd outer page no.	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

What about a 64-bit logical address space?

?	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

should we page the outer page too (three-level paging scheme)? but the 2nd outer page table will be 2^{34} bits (16 GB) in size and can't fit in one page

2nd outer page no.	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

this requires a prohibitive number of memory accesses! e.g., 5 levels of tables

*n*th outer page no.	...	2nd outer page no.	outer page no.	page no.	offset
<10 bits		10 bits	10 bits	10 bits	12 bits

What about a 64-bit logical address space?

?	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

should we page the outer page too (three-level paging scheme)? but the 2nd outer page table will be 2^{34} bits (16 GB) in size and can't fit in one page

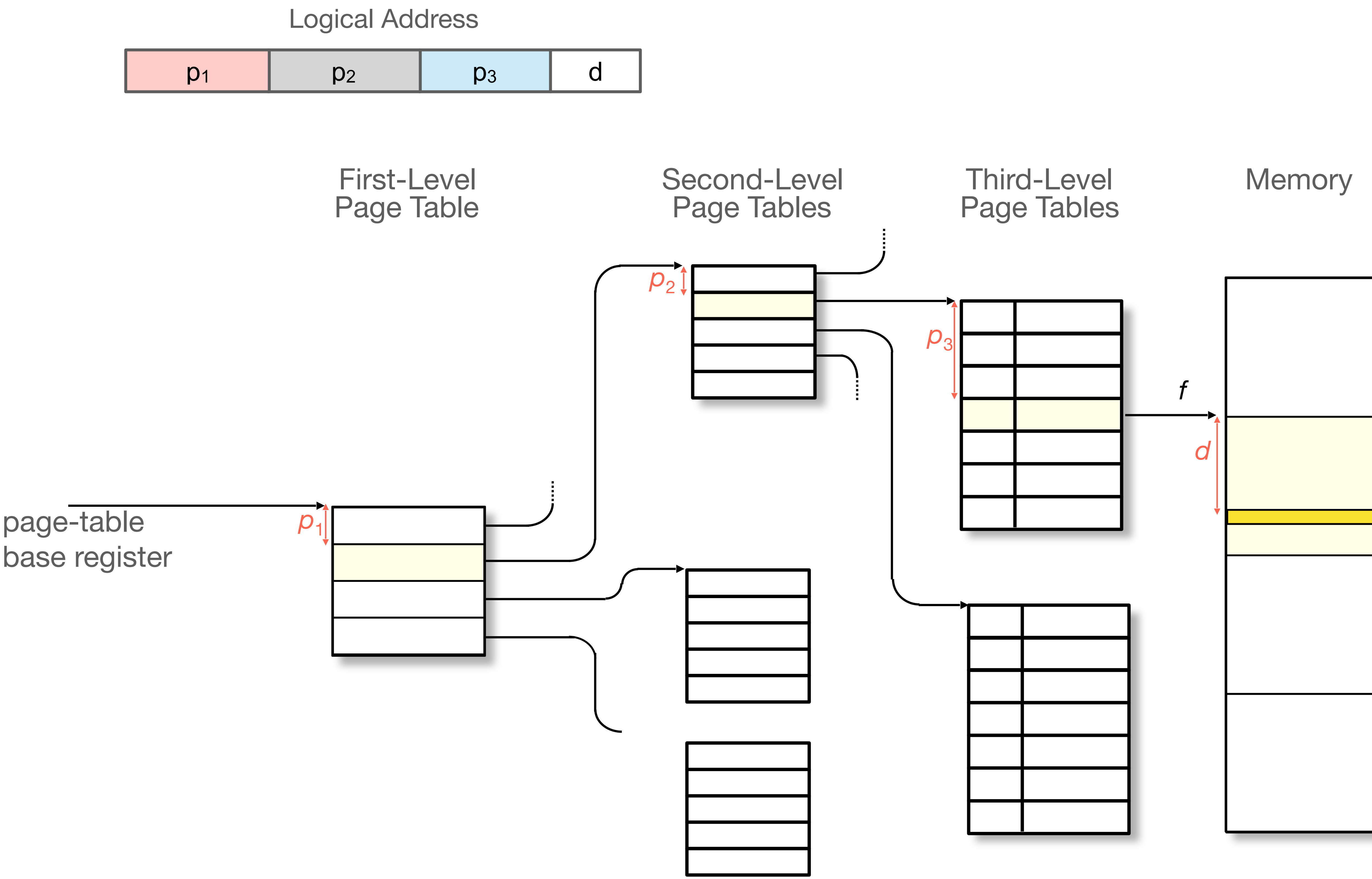
2nd outer page no.	outer page no.	page no.	offset
32 bits	10 bits	10 bits	12 bits

this requires a prohibitive number of memory accesses! e.g., 5 levels of tables

*n*th outer page no.	...	2nd outer page no.	outer page no.	page no.	offset
<10 bits		10 bits	10 bits	10 bits	12 bits

thus, for 64-bit architectures, hierarchical page tables are generally considered inappropriate

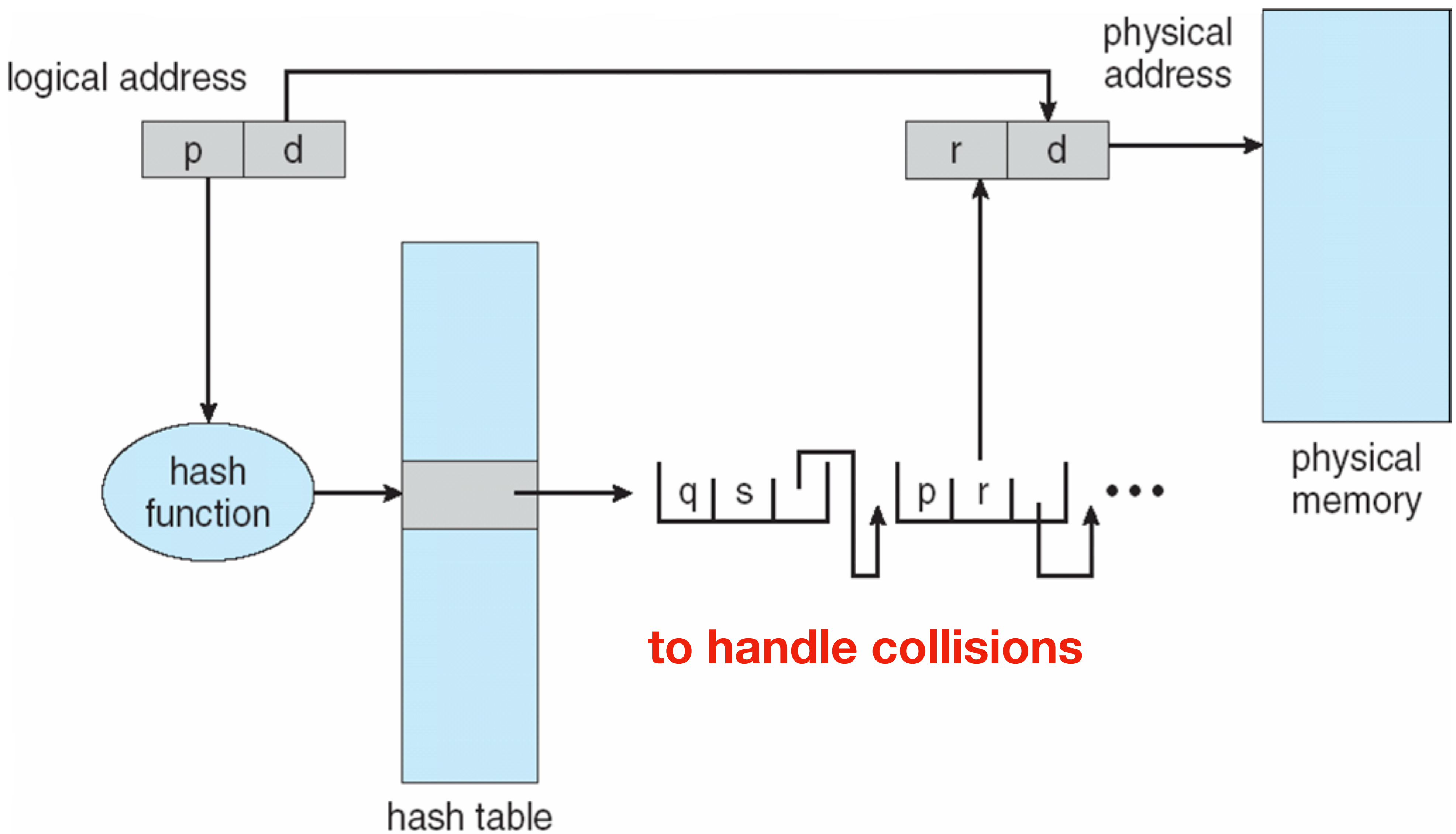
Multilevel address translation



Hashed page table

- common in address spaces > 32 bits
  - the virtual page number is hashed into a page table, so page *i* is placed in slot *H(i) where H* is an agreed-upon hash function
- each slot in a chain of elements hashing to the same location
  - each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- virtual page numbers are compared in this chain searching for a match
  - if a match is found, the corresponding physical frame is extracted

Hashed page table

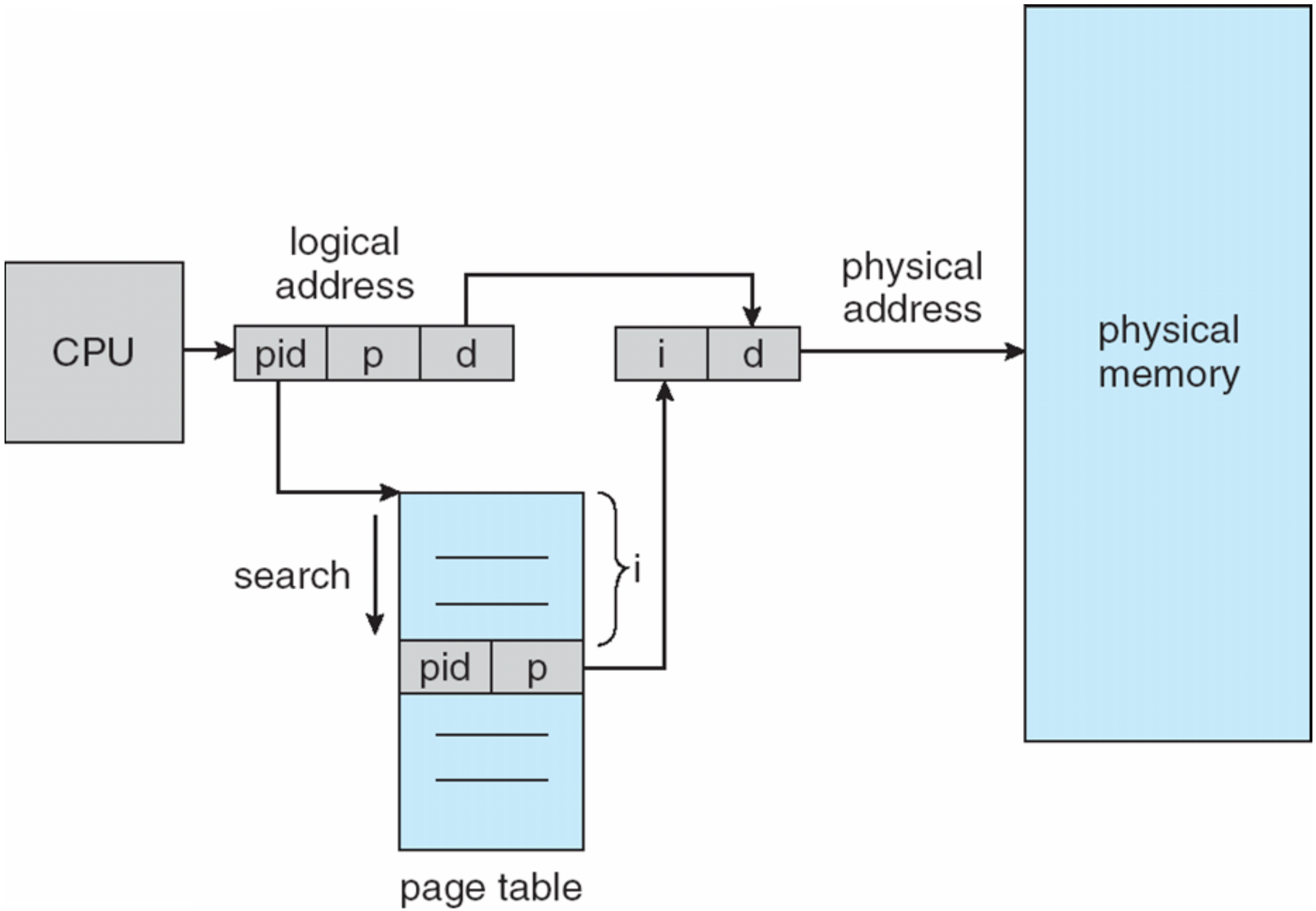


to handle collisions

Inverted page table

- rather than each process having a page table and keeping track of all possible logical pages, track all physical pages (frames) in one page table
  - the inverted page table has one entry for each frame of memory (sorted by physical address)
  - each page table entry consists of the virtual address of the page stored in that real memory location and an **address-space identifier** (e.g., which process owns that page)
- it **decreases** memory needed to store each page table, but **increases** time 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
- sharing cannot be used with inverted page table
  - because one physical page cannot have two or more shared virtual addresses

Inverted page table



Hierarchical versus hashed/inverted page table

- page table has one entry per virtual page
- hashed/inverted page table has one entry per page frame (i.e., physical frame)

VIRTUAL MEMORY

the illusion of an infinite virtual memory!

Virtual memory

- **fact**: the entire address space of a process is rarely used (e.g., unusual routines, large data structures) and we can run a process even if all its pages are not in memory
  - so why to load all pages into memory before running a process?

Virtual memory

- **fact**: the entire address space of a process is rarely used (e.g., unusual routines, large data structures) and we can run a process even if all its pages are not in memory
  - so why to load all pages into memory before running a process?
- **basic idea**: only portions of a process's virtual address space (working set) are mapped at any point in time; the remainder is **in multiple files on disk** (in file system or in swap space)
  - code pages are stored in a file on disk (e.g., `a.out`)
  - data, stack, and heap pages are also stored in a special file that is not visible to users and exists only when the program is executing
  - some of these pages are loaded in the main memory; hence, the main memory acts as a cache for the disk
  - the page table indicates if the page is on disk or memory using the **valid/invalid** bit

Virtual memory

- **fact**: the entire address space of a process is rarely used (e.g., unusual routines, large data structures) and we can run a process even if all its pages are not in memory
  - so why to load all pages into memory before running a process?
- **basic idea**: only portions of a process's virtual address space (working set) are mapped at any point in time; the remainder is **in multiple files on disk** (in file system or in swap space)
  - code pages are stored in a file on disk (e.g., `a.out`)
  - data, stack, and heap pages are also stored in a special file that is not visible to users and exists only when the program is executing
  - some of these pages are loaded in the main memory; hence, the main memory acts as a cache for the disk
  - the page table indicates if the page is on disk or memory using the **valid/invalid** bit
- **consequences**: (a) the virtual address space is no longer constrained by the size of the physical address space; (b) more processes can fit in memory to run concurrently

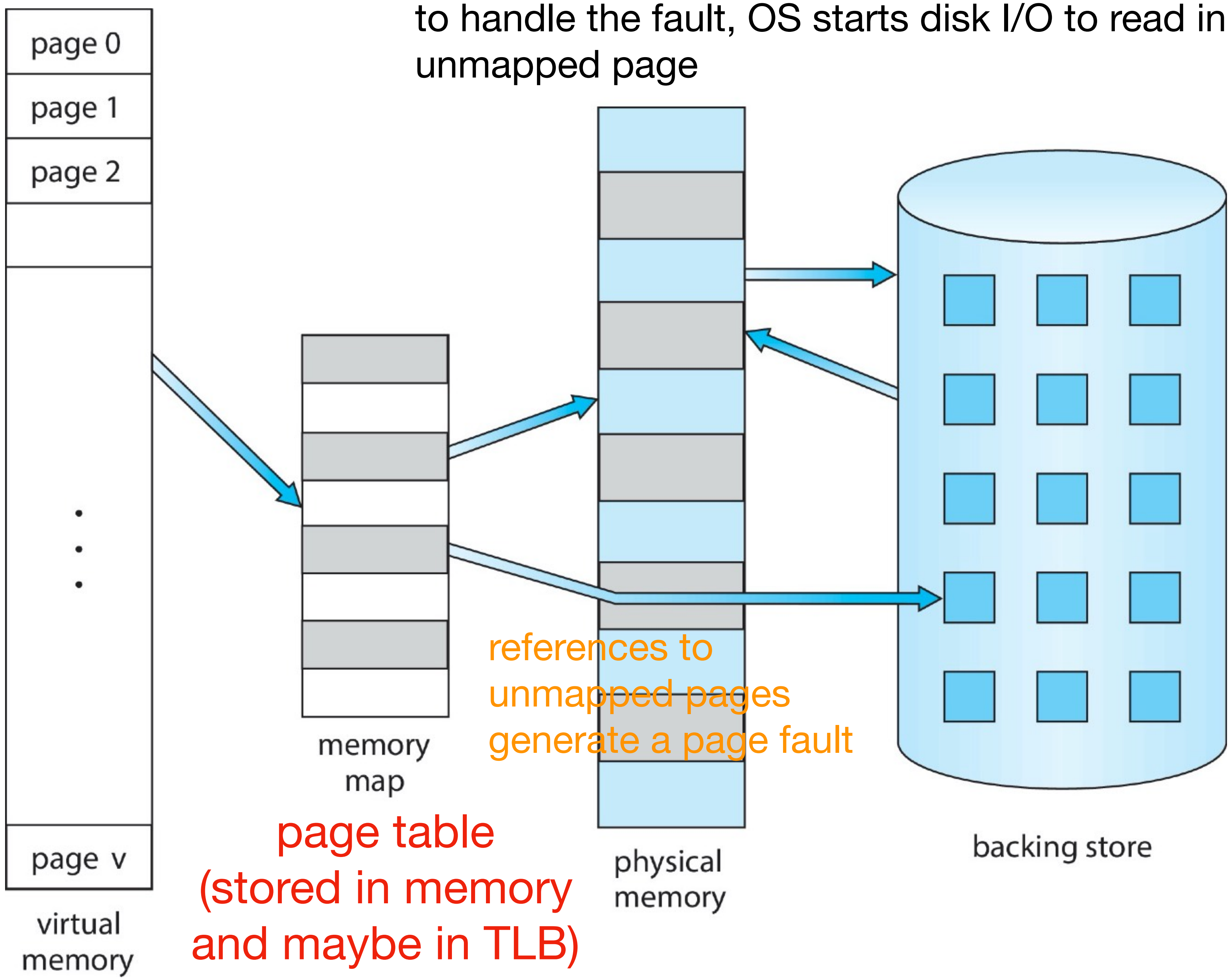
Virtual memory

- **fact**: the entire address space of a process is rarely used (e.g., unusual routines, large data structures) and we can run a process even if all its pages are not in memory
  - so why to load all pages into memory before running a process?
- **basic idea**: only portions of a process's virtual address space (working set) are mapped at any point in time; the remainder is **in multiple files on disk** (in file system or in swap space)
  - code pages are stored in a file on disk (e.g., `a.out`)
  - data, stack, and heap pages are also stored in a special file that is not visible to users and exists only when the program is executing
  - some of these pages are loaded in the main memory; hence, the main memory acts as a cache for the disk
  - the page table indicates if the page is on disk or memory using the **valid/invalid** bit
- **consequences**: (a) the virtual address space is no longer constrained by the size of the physical address space; (b) more processes can fit in memory to run concurrently
- **caveat**: memory accesses must be for pages that are in memory for the vast majority of time, otherwise the effective memory access time will approach that of the disk

What happens if a program references a page that is not memory resident?

a page fault occurs when the page table is accessed and the entry for the page does not have the valid bit set

to handle the fault, OS starts disk I/O to read in the unmapped page



When to load a page into memory?

- load all pages of a process at its start time
  - disadvantages: unnecessary I/O for brining all pages into memory, and wasting a lot of memory
  - the virtual address space must be no larger than the physical memory (it’s not necessarily true)

When to load a page into memory?

- load all pages of a process at its start time
  - disadvantages: unnecessary I/O for brining all pages into memory, and wasting a lot of memory
  - the virtual address space must be no larger than the physical memory (it's not necessarily true)
- leave it to the programmer to decide when to load and evict pages
  - is difficult to do and error-prone

When to load a page into memory?

- load all pages of a process at its start time
  - disadvantages: unnecessary I/O for brining all pages into memory, and wasting a lot of memory
  - the virtual address space must be no larger than the physical memory (it's not necessarily true)
- leave it to the programmer to decide when to load and evict pages
  - is difficult to do and error-prone
- request paging: process tells OS before it needs a page, and when it is through with a page

When to load a page into memory?

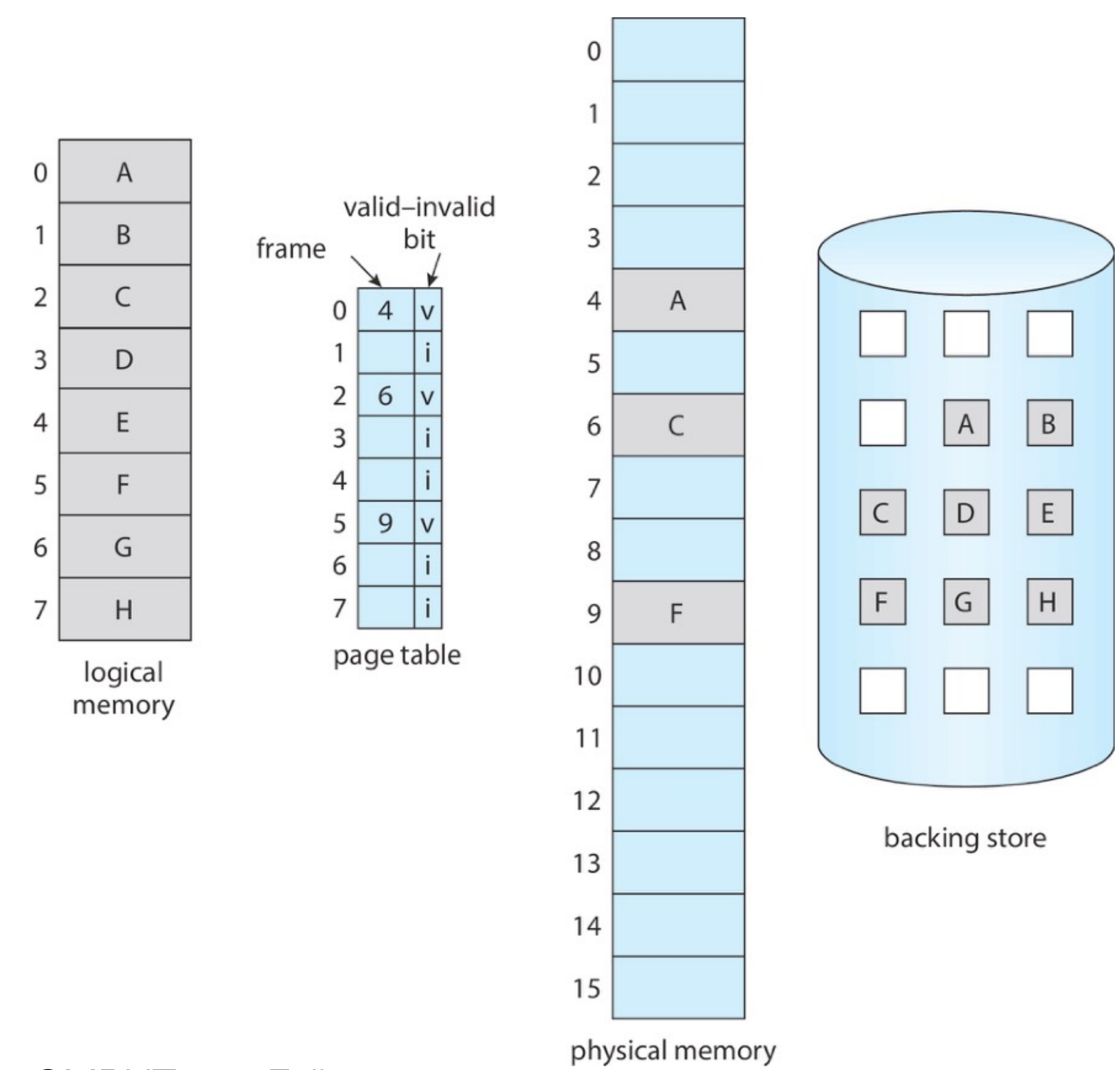
- load all pages of a process at its start time
  - disadvantages: unnecessary I/O for brining all pages into memory, and wasting a lot of memory
  - the virtual address space must be no larger than the physical memory (it’s not necessarily true)
- leave it to the programmer to decide when to load and evict pages
  - is difficult to do and error-prone
- request paging: process tells OS before it needs a page, and when it is through with a page
- **demand paging**: OS loads a page the first time it is referenced (transparent to the process)
  - may need to remove a page from memory to make room for the new page (**will discuss this later**)
  - process must give up the CPU while the page is being loaded (transitions to the waiting state)

When to load a page into memory?

- load all pages of a process at its start time
  - disadvantages: unnecessary I/O for brining all pages into memory, and wasting a lot of memory
  - the virtual address space must be no larger than the physical memory (it's not necessarily true)
- leave it to the programmer to decide when to load and evict pages
  - is difficult to do and error-prone
- request paging: process tells OS before it needs a page, and when it is through with a page
- **demand paging**: OS loads a page the first time it is referenced (transparent to the process)
  - may need to remove a page from memory to make room for the new page (**will discuss this later**)
  - process must give up the CPU while the page is being loaded (transitions to the waiting state)
- pre-paging: OS guesses in advance which pages the process will need and pre-loads them into memory
  - allows more overlap of CPU and I/O if the OS guesses correctly, otherwise it causes a page fault
  - errors may result in removing useful pages
  - difficult to get right due to branches in code

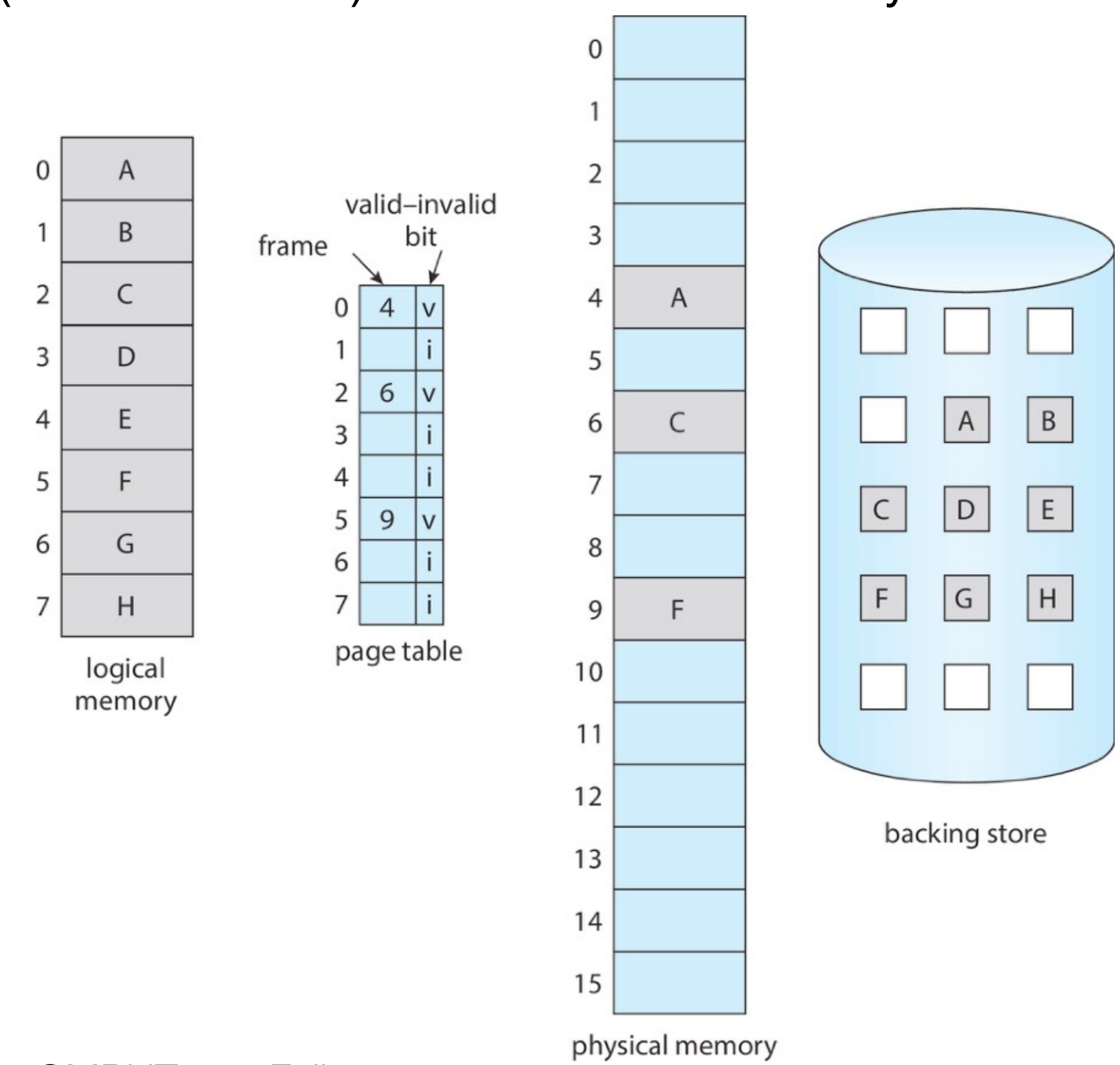
Implementing demand paging

- a copy of the entire program must be stored on disk



Implementing demand paging

- a copy of the entire program must be stored on disk
- the valid bit in the page table indicates that the page is in memory
  - ‘v’ indicates that the page is in-memory
  - ‘i’ indicates that it is not-in-memory (initiate disk I/O) or is an invalid memory reference (abort the process in this case)



What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out

free-frame list: a pool of free frames

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out
- while I/O is being done, OS context switches to another process
- OS gets interrupt when I/O completes (i.e., the page is read from disk and loaded into memory)
  - saves the registers and state of the other process

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out
- while I/O is being done, OS context switches to another process
- OS gets interrupt when I/O completes (i.e., the page is read from disk and loaded into memory)
  - saves the registers and state of the other process
- OS updates the page table entry

What happens on a page fault?

- instruction faults on a page whose valid bit is not set
- page fault causes a trap to kernel
  - saves the registers and state of the faulting process
  - checks if the memory reference was legal and determines the location of the page on the disk (requires a more complex page table)
- OS puts the faulting process on the wait queue of disk and starts reading the unmapped page from disk to a frame
  - requires selecting a free frame (or a page to replace using a page replacement algorithm) and zeroing it out
- while I/O is being done, OS context switches to another process
- OS gets interrupt when I/O completes (i.e., the page is read from disk and loaded into memory)
  - saves the registers and state of the other process
- OS updates the page table entry
- OS puts the faulting process back on the ready queue
  - resumes the interrupted instruction when CPU is allocated to this process again

Handling a page fault