ELEC 377 – Operating Systems

Week 6 – Class 3

Last Class

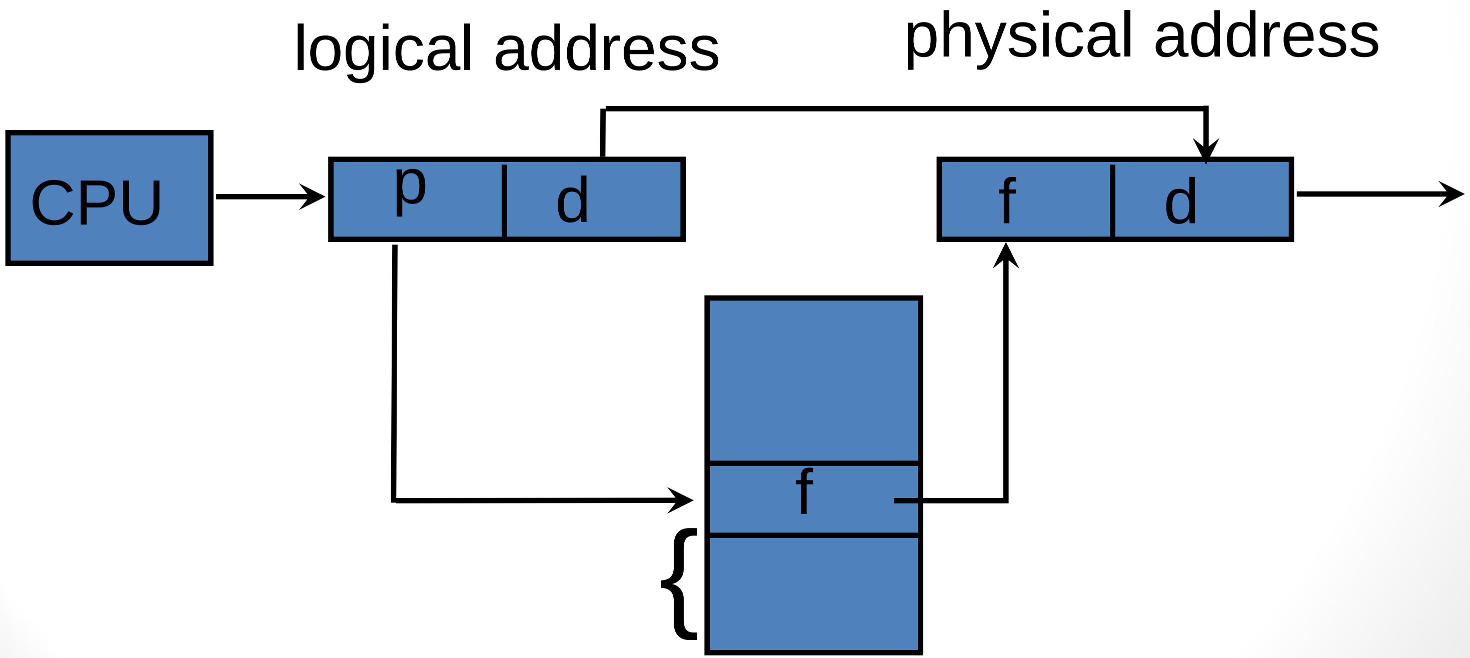
- Memory Management
- ◊ Memory Paging
- ◊ Paging Structure

Today

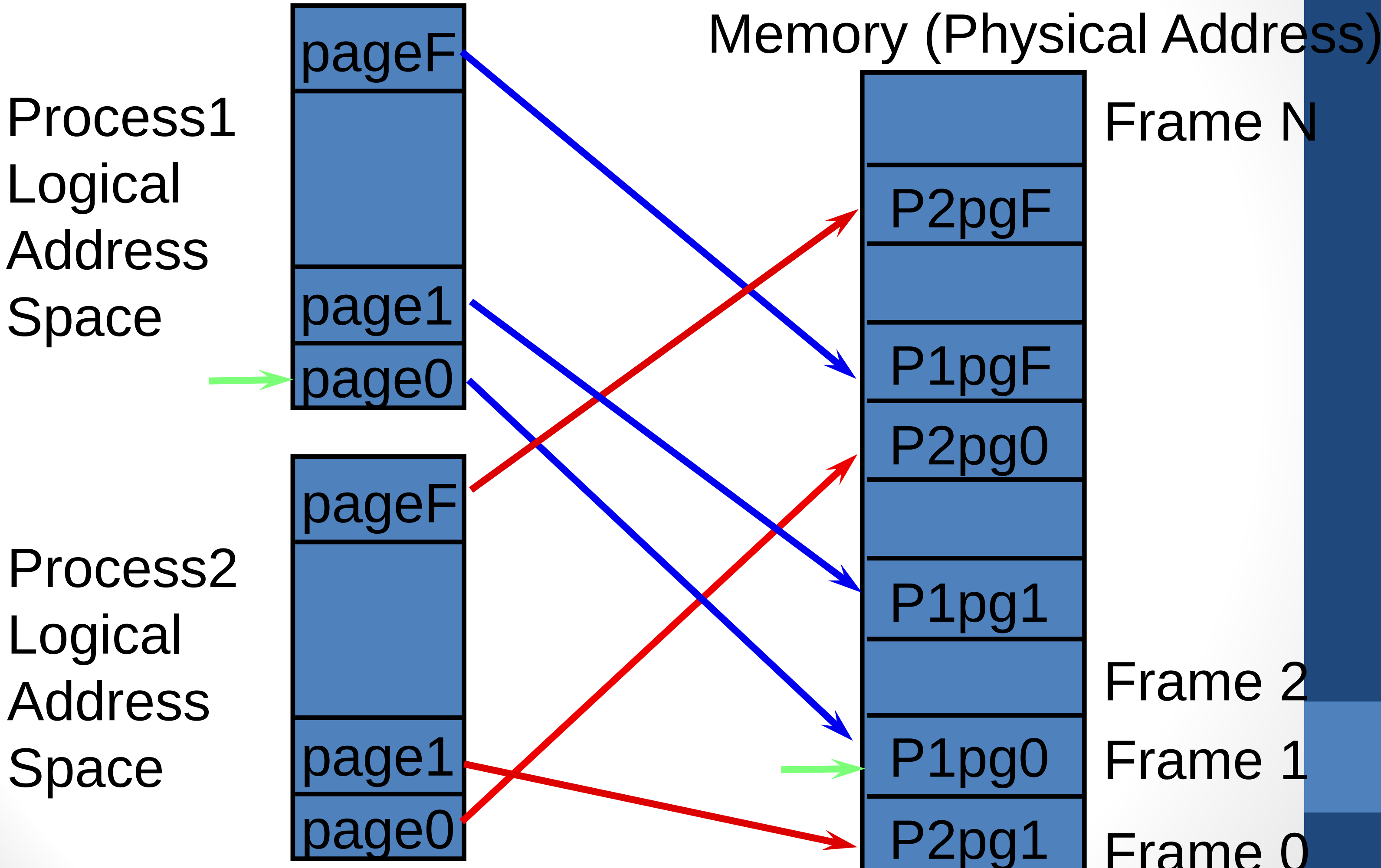
- Paging Sizes
- Virtual Memory
- ◊ Concept
- ◊ Demand Paging

Page Table

- address generated by CPU is divided into two parts
  - ◊ page number(p) - index into page table
  - ◊ page offset(d) – location within the page



Memory is mapped by Page Tables



Page Tables

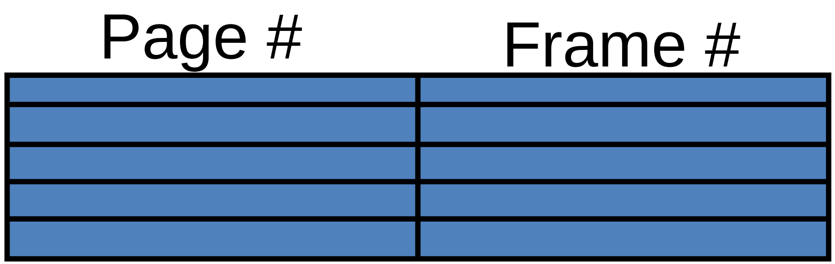
- Each process can have its own logical address space
    - ◊ exception: Mac OS 7 had a single page table for all processes (no memory protection between processes)
- Thus each process has its own page table
- Thus each process has its own mapping to physical memory

Paging

- **frame table** - keeps track of allocated and free frames
- I/O operations have to know memory layout
- page table has to be switched during context switch
  - increase in context switch time
- page tables are large, and usually kept in main memory - page table base register, length register
  - ◊ Extra memory traffic
  - ◊ Must first use page number to find frame number, then access instruction or data in memory
  - ◊ Simple linear table adds one memory access for each data access.

Paging - TLB

- Want to minimize extra memory traffic of page tables
- Small cache inside of MMU
  - ◊ TLB - translation look-aside buffer
  - ◊ associative memory



TLB

- If address not in TLB, called a *miss*
  - ◊ Requires one extra memory access cycle in linear page model (one for table, one for memory)
  - ◊ new address added to TLB
  - ◊ performance very sensitive to hit-ratio
- Some entries in TLB are not modifiable (kernel addresses)
- Some TLBs support multiple processes by adding process IDs to the TLB. This allows more than one process in TLB at a time.
  - ◊ otherwise TLB must be flushed on each context switch

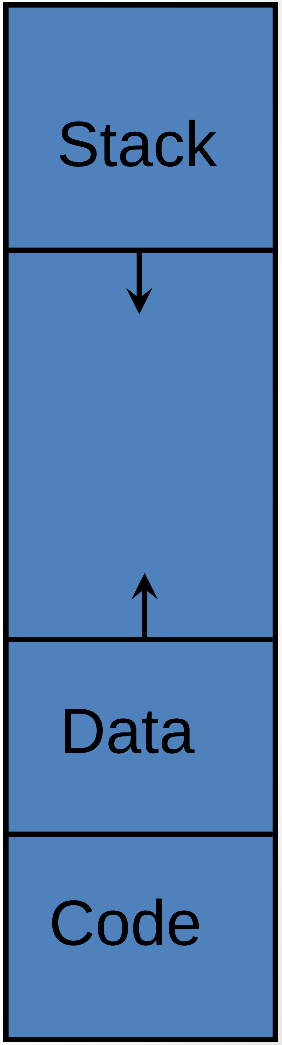
Memory Protection

- Since memory is no longer contiguous, base + limit not sufficient for protection
  - ◊ could still be used on logical address side
- memory belonging to other processes not in page table and thus not visible!!
- OS may be in page table for quick access
- add valid-invalid bit to page table. Process can only access valid pages

Page Table Structure

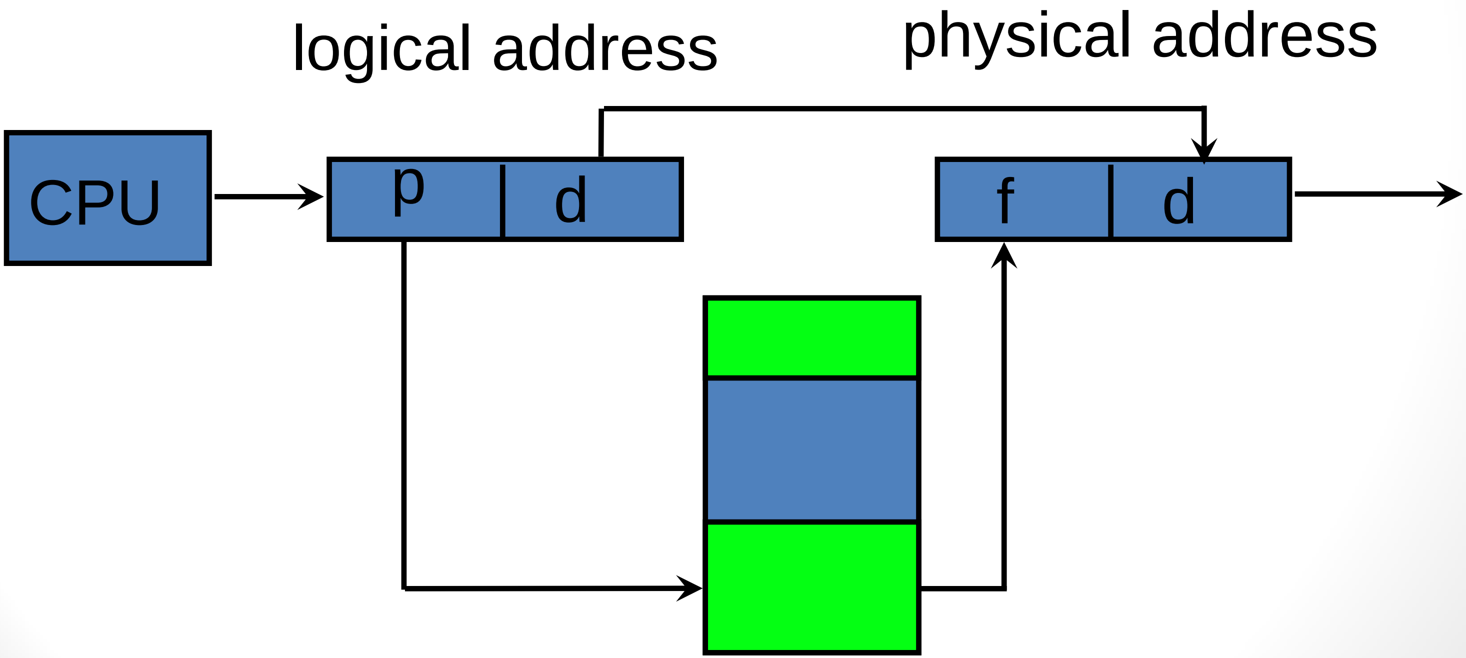
- The page table can become very large.
  ◊ 32 bit address space, 12 bit page (4K) give 4 Megabyte page table (allocated contiguously)
  ◊ larger than some programs!!

- Want to reduce resources required by page tables
  - ◊ break up page table so not contiguous
  - ◊ reduce size of page table
  - ◊ not all process space may be valid!!



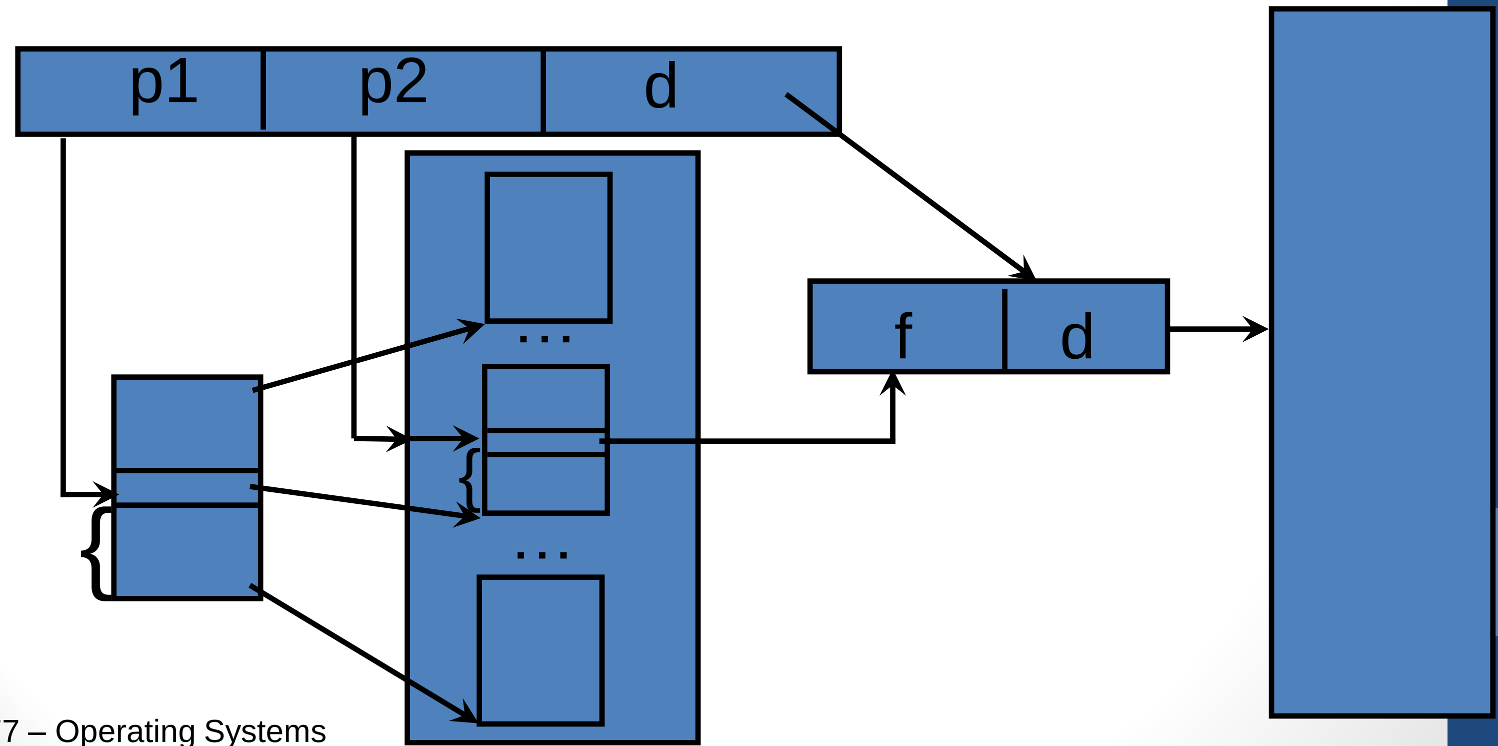
Page Table

- Middle of table is not used (Does not contain references to physical memory frames)
- page the page table?



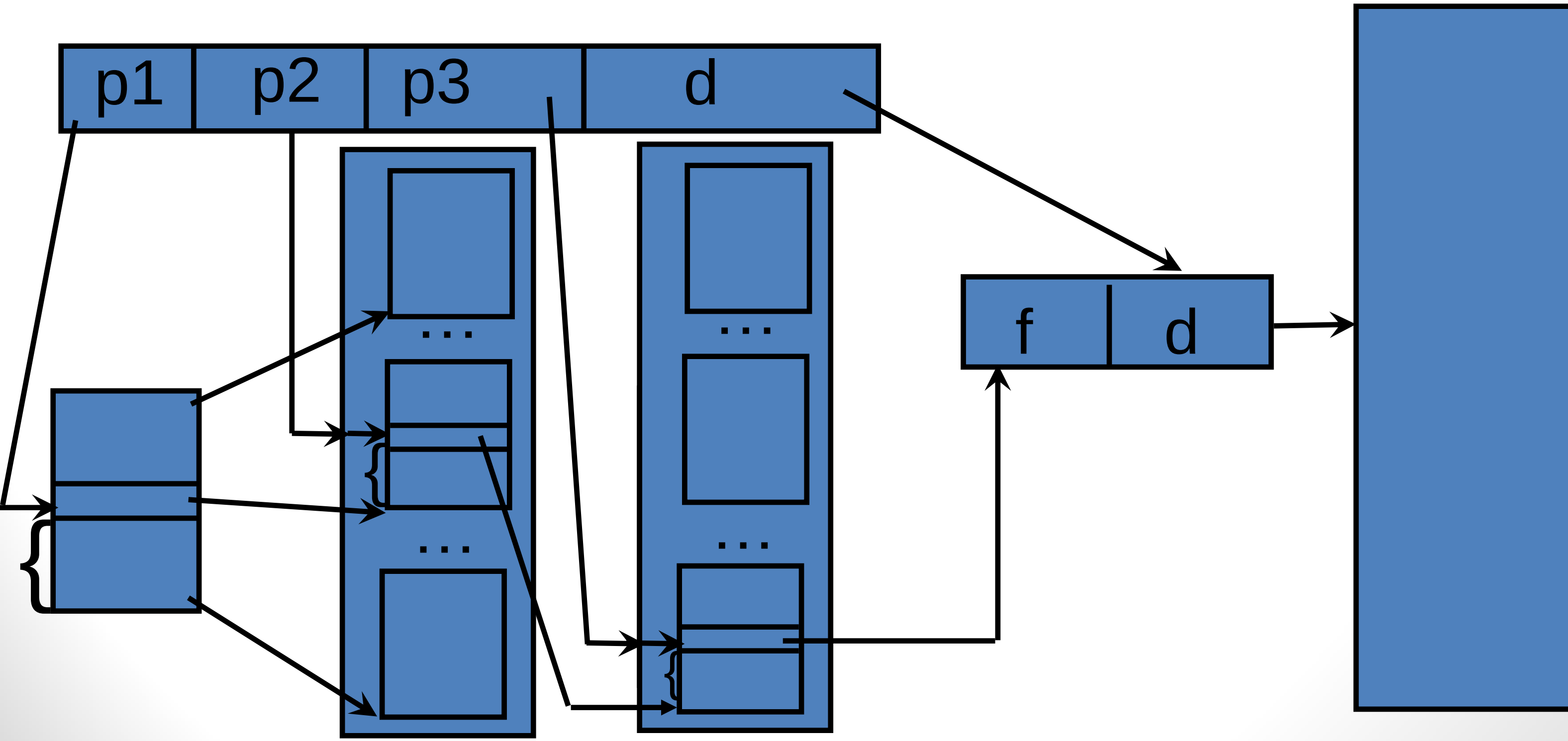
Hierarchical Tables

- Multiple Tables (forward-mapped page table)
  - ◊ page number is split into one or more sections
  - ◊ First table gives address of next table ending at physical memory *(*(*(T1+p1) + p2)+d)



Hierarchical Tables

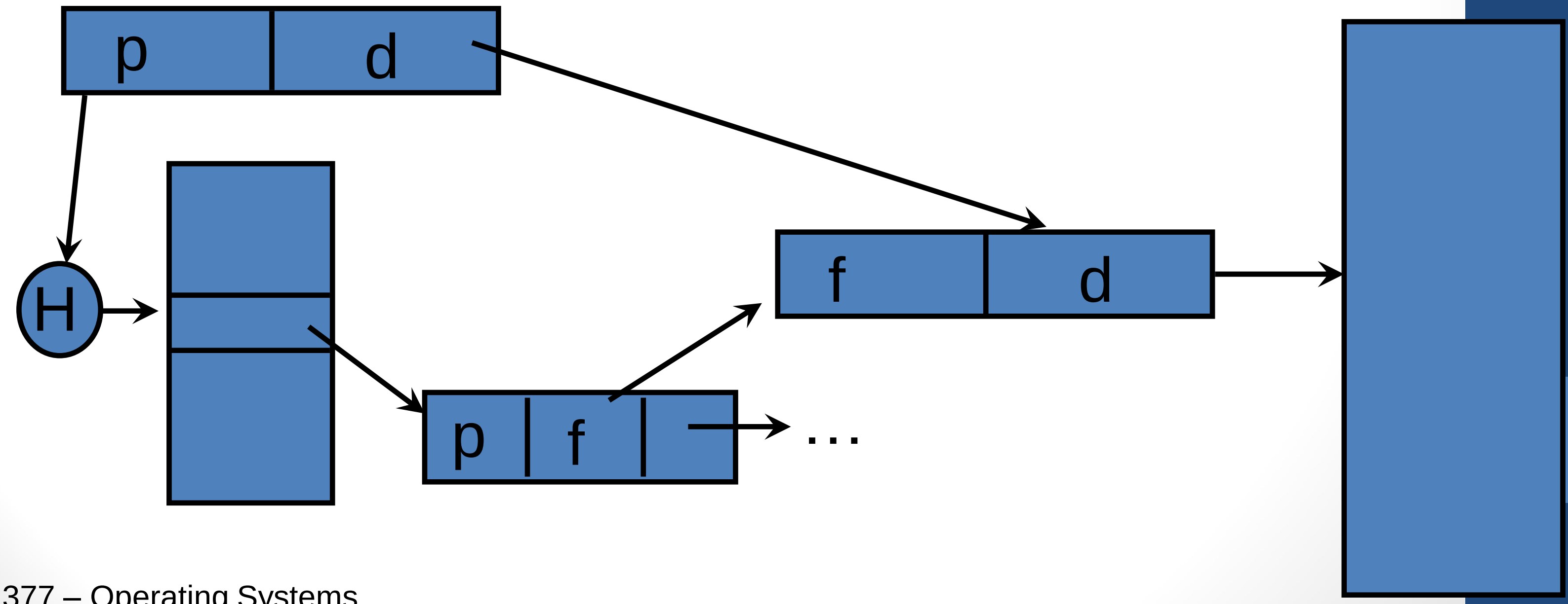
- Can be more than two levels (Pentium has 4 levels)
- Cost of miss(assume each lookup is one access)
  - one extra access for each extra level of table



Hashed Page Tables

- Store page entries in a hash table, hashed by page number
- Hash table has collisions (more than one page number might hash to a given location)

◊ Store as a linked list (variable extra cost!!)



Inverted Pages

- previous schemes have separate table(s) per process
- Just as the TLB has a combined representation, so do page tables
- tables have empty space
- Inverted tables have one entry for each *frame*
  - ◊ Stores process number and page number corresponding to the frame
  - ◊ Search table for <pid,page#>
  - ◊ hash table can shorten search time

- Need to extend to handle shared memory

Page Structures - Summary

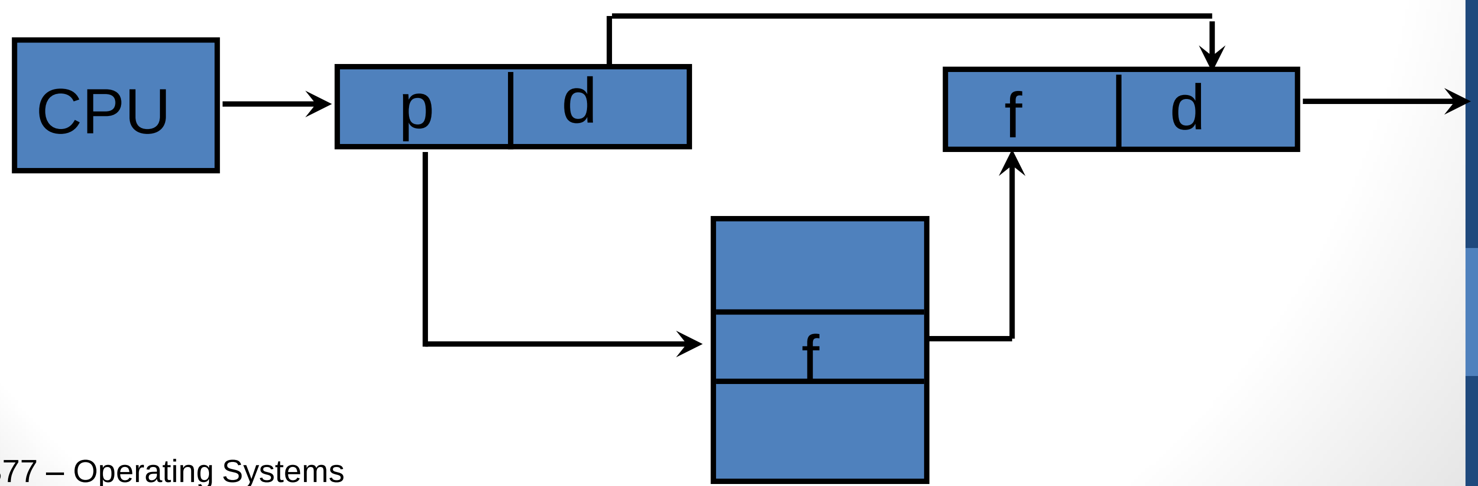
- Three ways of reducing the memory requirements of the page tables
- *All* of them increase the cost of converting a logical address to a physical address
  - ◊ TLB absorbs much of the cost
  - ◊ Increase the cost of a TLB miss
  - ◊ Effectiveness of TLB is more important

Demand Paging

- All pages of the process are swapped out
- swap each page in as it is needed
  - ◊ lazy swapper
  - ◊ if page is never referenced -> never loaded!!
  - ◊ pager vs swapper
- Hardware needs to know which pages are in memory

Logical vs Physical Address Size

- Pages and Frames must be the same size
- ◊ |d| same in both physical and logical spaces
- But there may be more frames than pages
- ◊ or more pages than frames
- not necessary that |p| = |f|



Logical vs Physical Address Size

- d = 12, logical = 32, physical = 40

◊ page/frame size = 4K

◊ #pages = | 20 | = 1024K pages = 1M pages

  - page table size (1 word per entry) = 4M

◊ #frames = | 28 | = 256M frames

- d = 10, logical = 32, physical = 24

◊ page/frame size = 1K

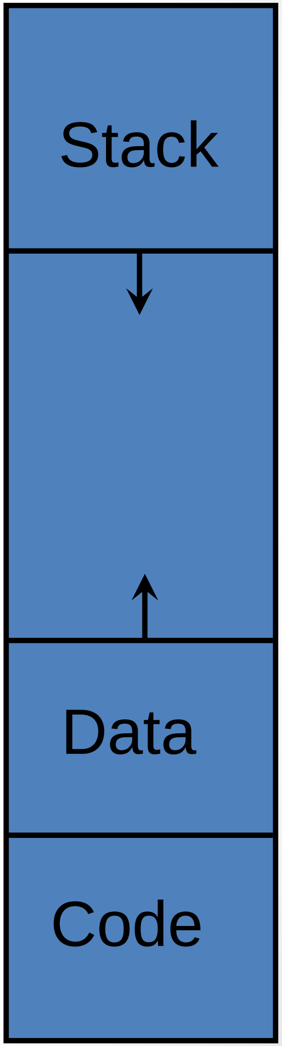
◊ #pages = |22| = 4M pages

◊ #frames = |14| = 16K frames

  - total physical memory = 16M

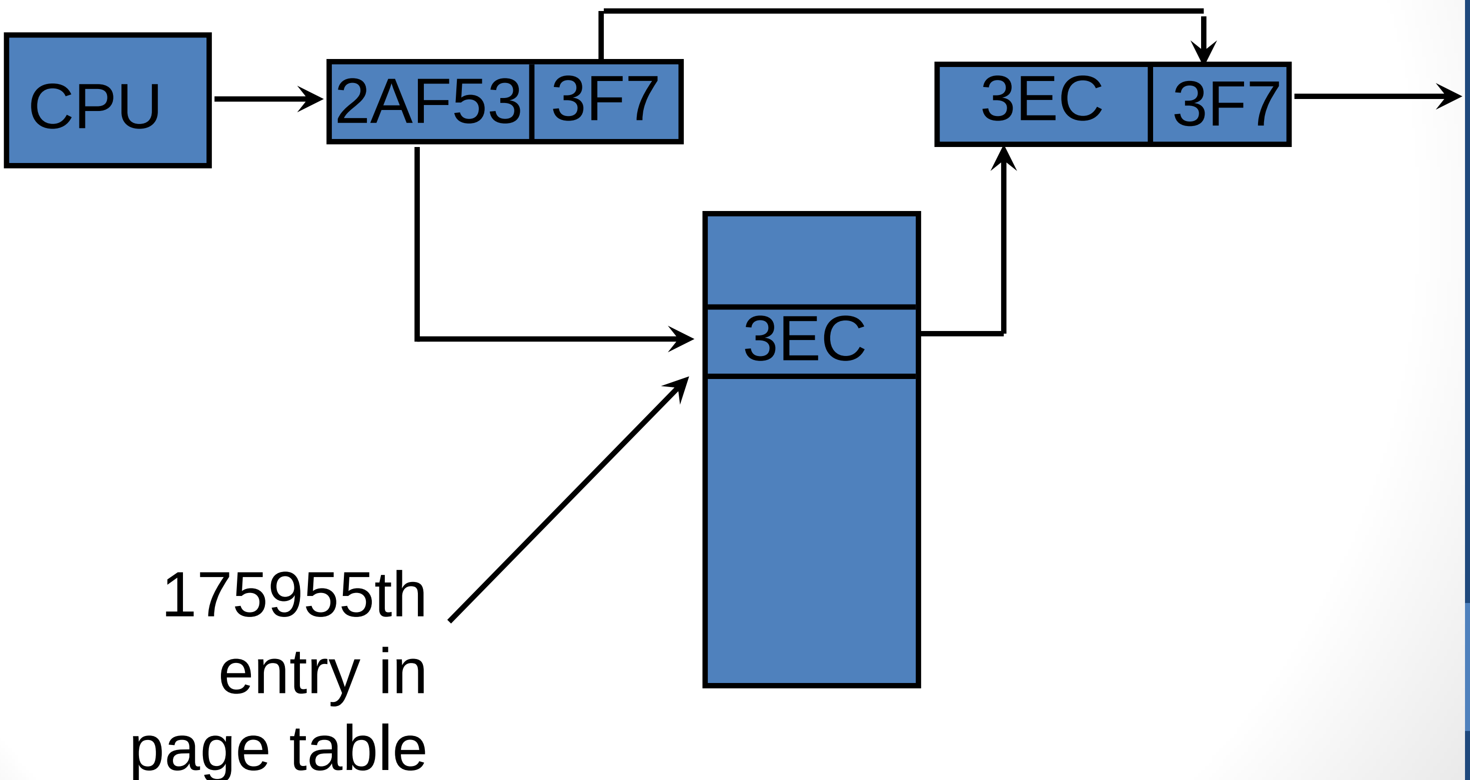
Logical vs Physical Address Size

- how can the physical address space be less than the logical address space?
- ◊ Not all of the logical address space is used
- ◊ 32 bits = 4Gb
- ◊ A process simple process may only have 10 pages (not including shared libraries)
  = 80K
- ◊ middle is not mapped



Example Translation

Logical = 32, Physical = 24, Page Size = 4K



Hierarchical Tables Example

- Program with 222k of Code and Data, 30K of Stack
  - ◊ 32 bit address space, 1k pages, p1 is 12 bits
    - how much space is taken by the page tables??
    - assume 4 bytes for each entry of p1 table and 4 bytes for each entry of each p2 table.

p2 = 10 bits

how many pages of code and data? -> 222 pages

how many pages for stack? -> 30 pages

How many p2 tables? -> 2 tables

how many p1 tables? -> 1 table

size of p1 table = $2^{12} * 4 = 2^{14}$

size of each p2 table = $2^{10} * 4 = 2^{12}$

Total table space = $2^{14}+2^{12}+2^{12} = 2^{14}+2^{13}$=16k+8k=24k

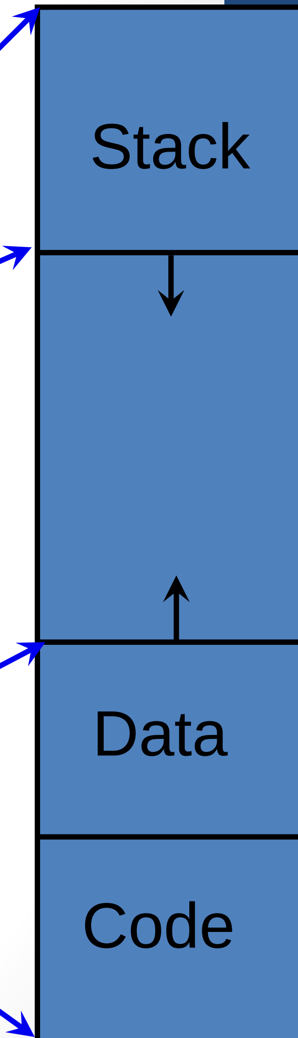
Page Table Example

Stack - 30 K

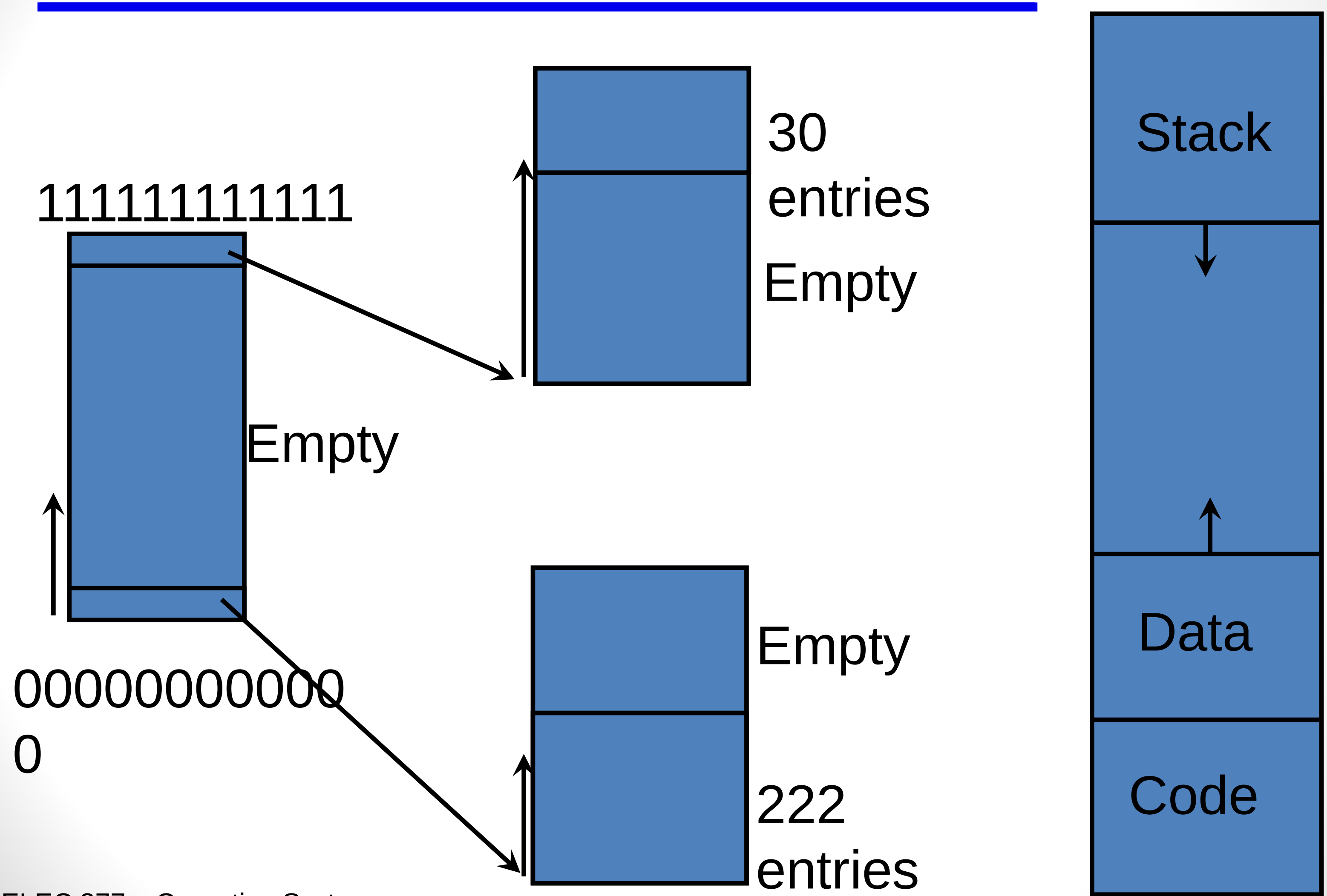
11111111111*1111111111*1111111111
11111111111*1111100010*0000000000

Code+Data 222K

00000000000*0011011101*1111111111
00000000000*0000000000*0000000000



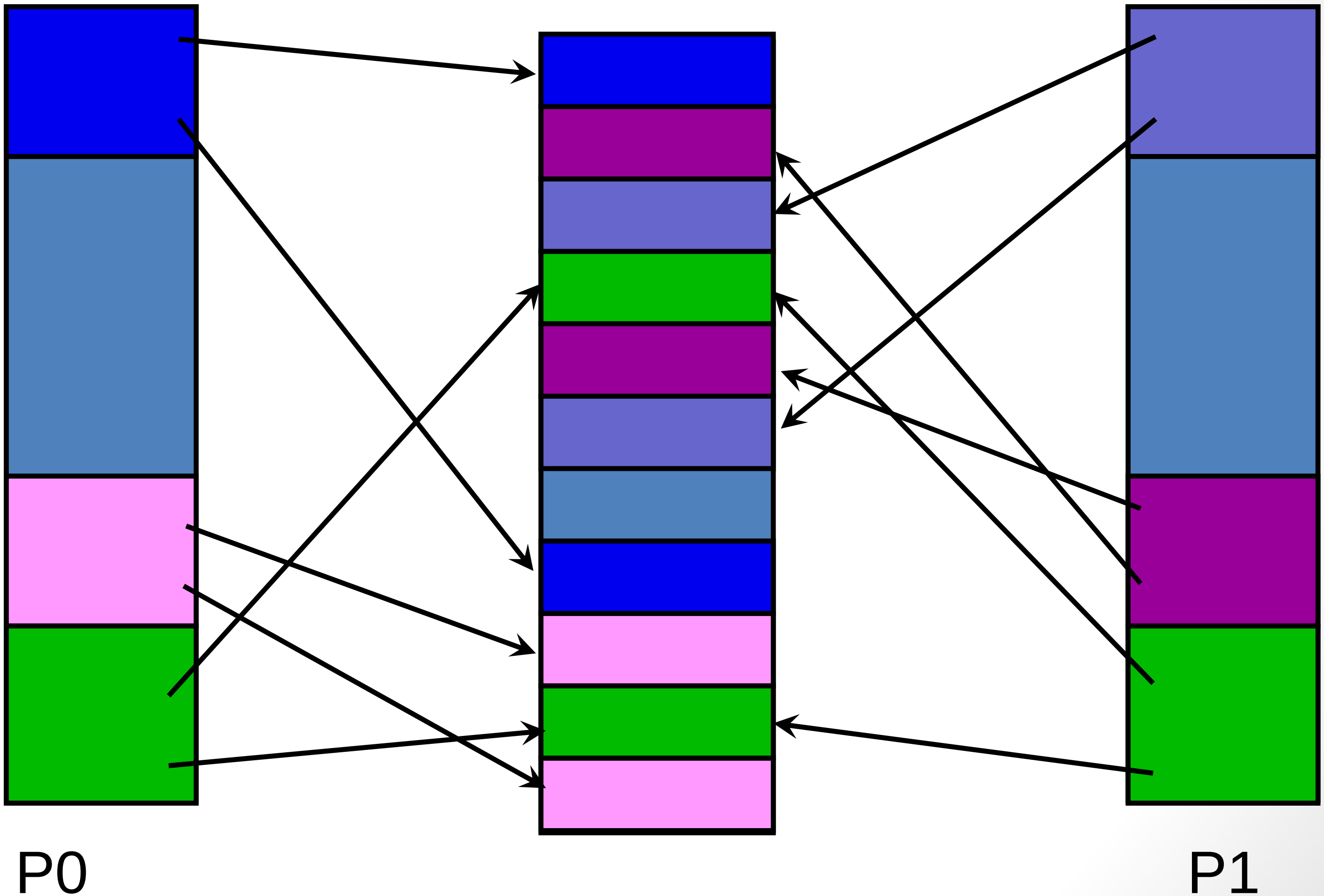
Page Table Example



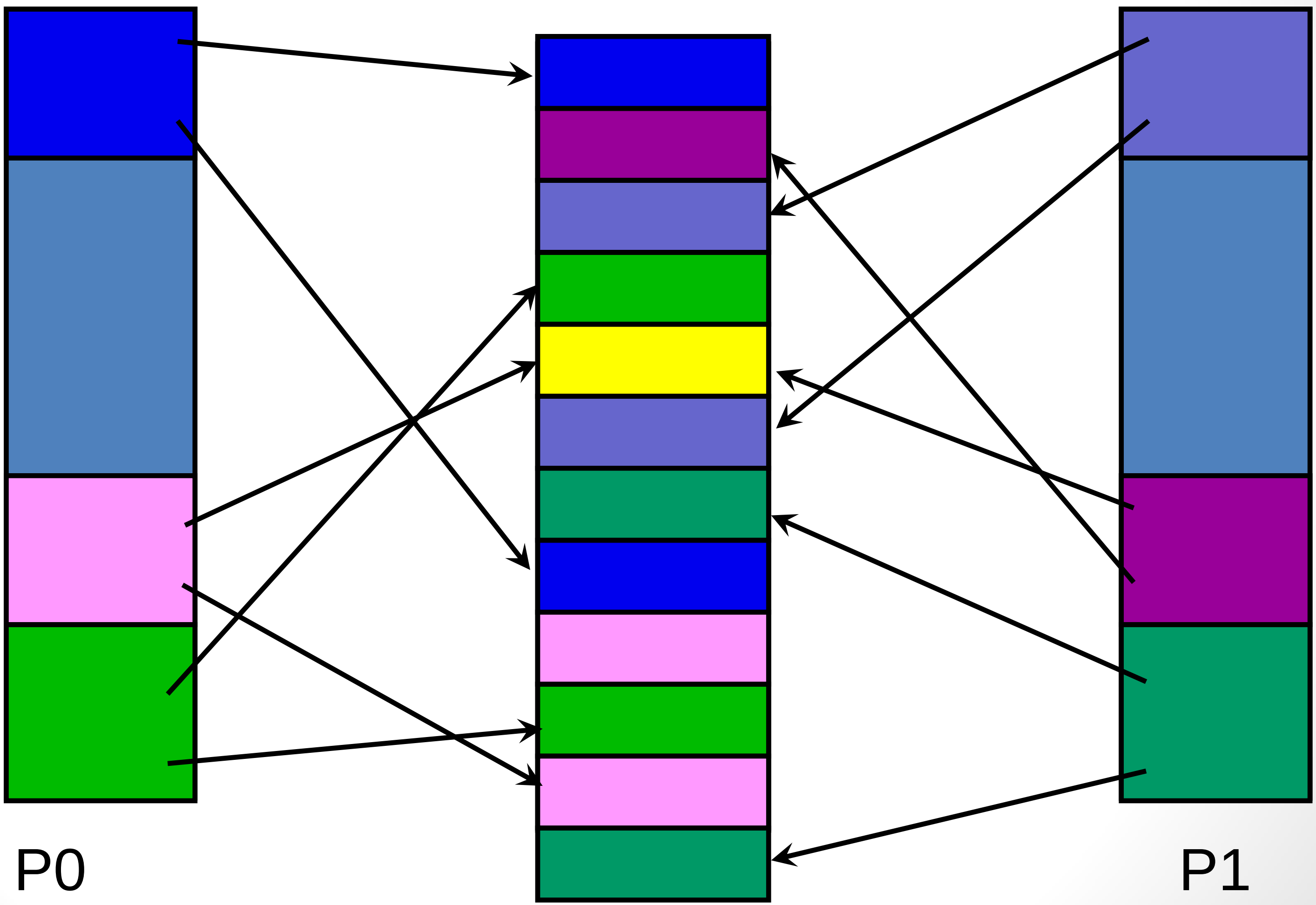
Sharing Pages

- Shared Libraries
- Multiple invocations of a given program (e.g. shell, editor)
- Contiguous memory allocation makes sharing difficult
- Shared code must be reentrant
  - ◊ Must not modify itself
  - ◊ Must also be position independent
- Most data is not shared
  - however IPC Shared Segments (lab3) now easy!!
- Page table entries for shared code and data in each process point to the common frames
- Page table entries for private data point to different frames

Sharing Pages-Two Procs same Prog



Sharing Pages -Two Progs, 1 Shared



Today

- Paging Sizes
- Virtual Memory <<<<<<<

◊ Concept

◊ Demand Paging

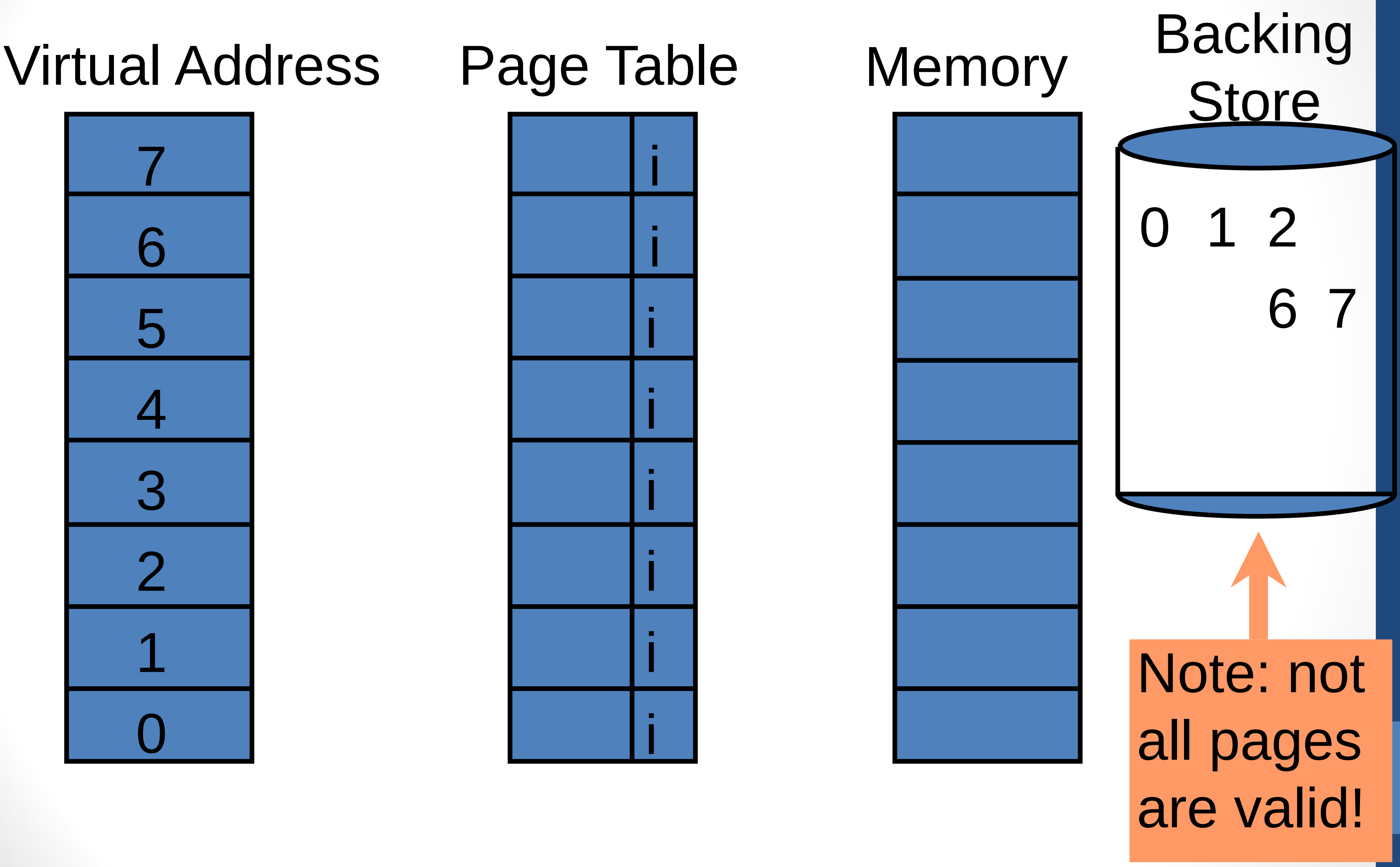
Virtual Memory

- Separation of Logical Address from Physical Address
- Each process has it's own virtual address space
  - ◊ Thinks it has the machine to itself
- Not all of the program need be in memory at one time
  - ◊ dynamic loading, overlays
  - ◊ only map the needed pages to frames in memory
  - ◊ store unused pages on the disk (similar to swap)
  - ◊ more efficient to start a new process
- Logical address space can be bigger than physical address space!!
  - ◊ process can more effectively share available memory resources

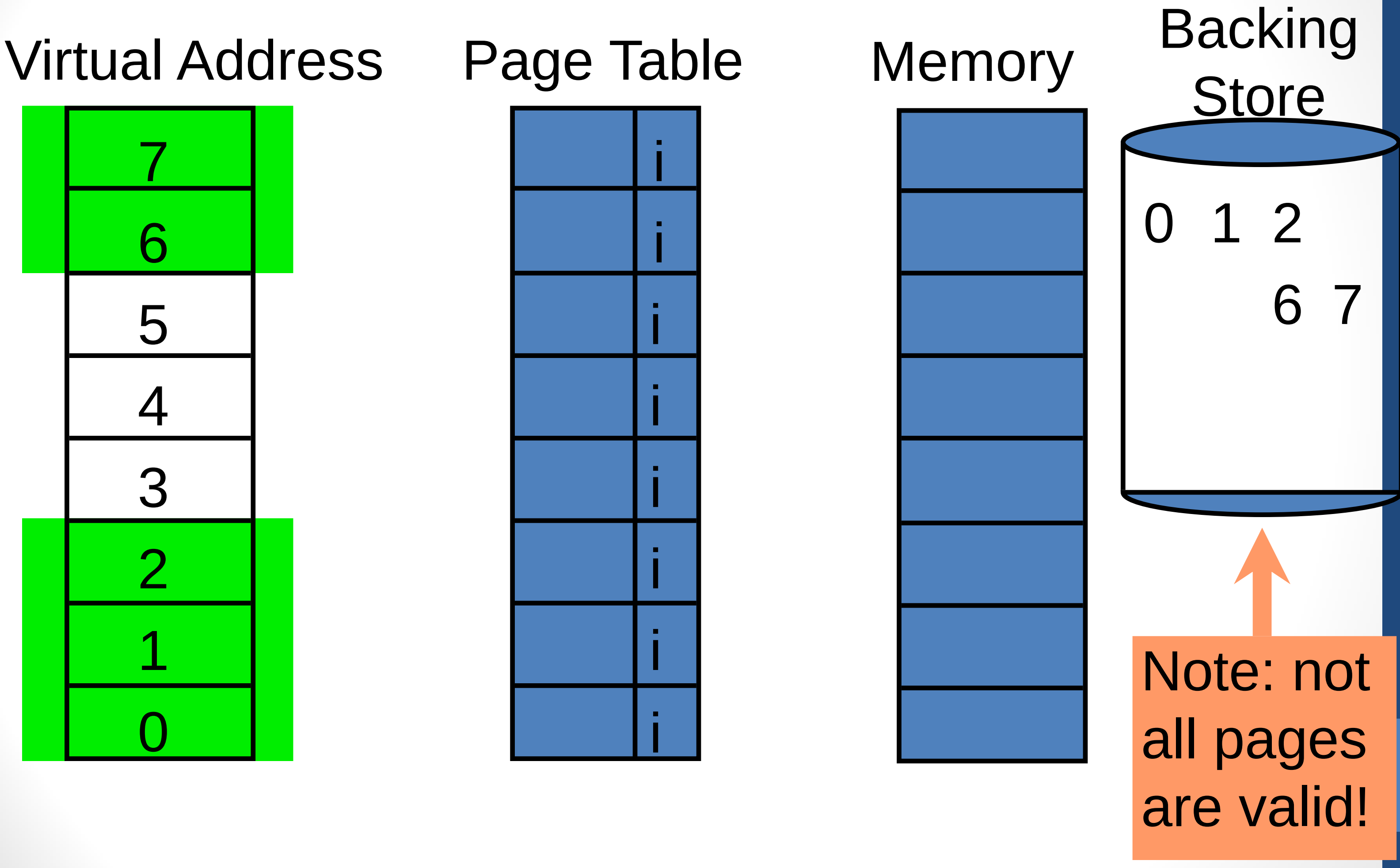
Demand Paging

- All pages of the process are swapped out
- swap each page in as it is needed
  - ◊ lazy swapper
  - ◊ if page is never referenced -> never loaded!!
  - ◊ pager vs swapper
  - ◊ *locality of reference*
- Hardware needs to know which pages are in memory
  - ◊ invalid bit -> now means not in memory
  - ◊ when a process starts, all pages are invalid
  - ◊ generate a page-fault when the page is not in memory.

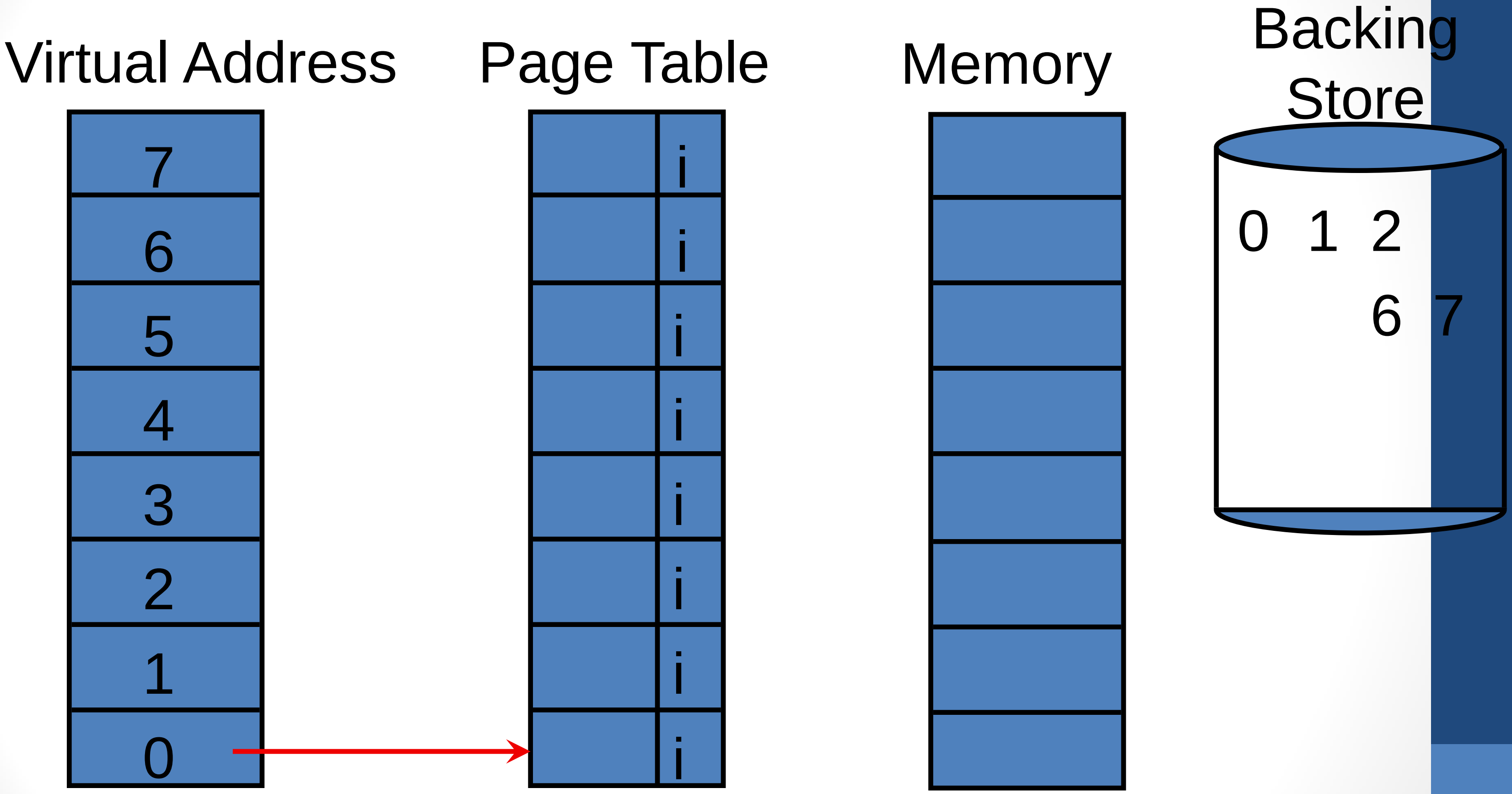
Demand Paging



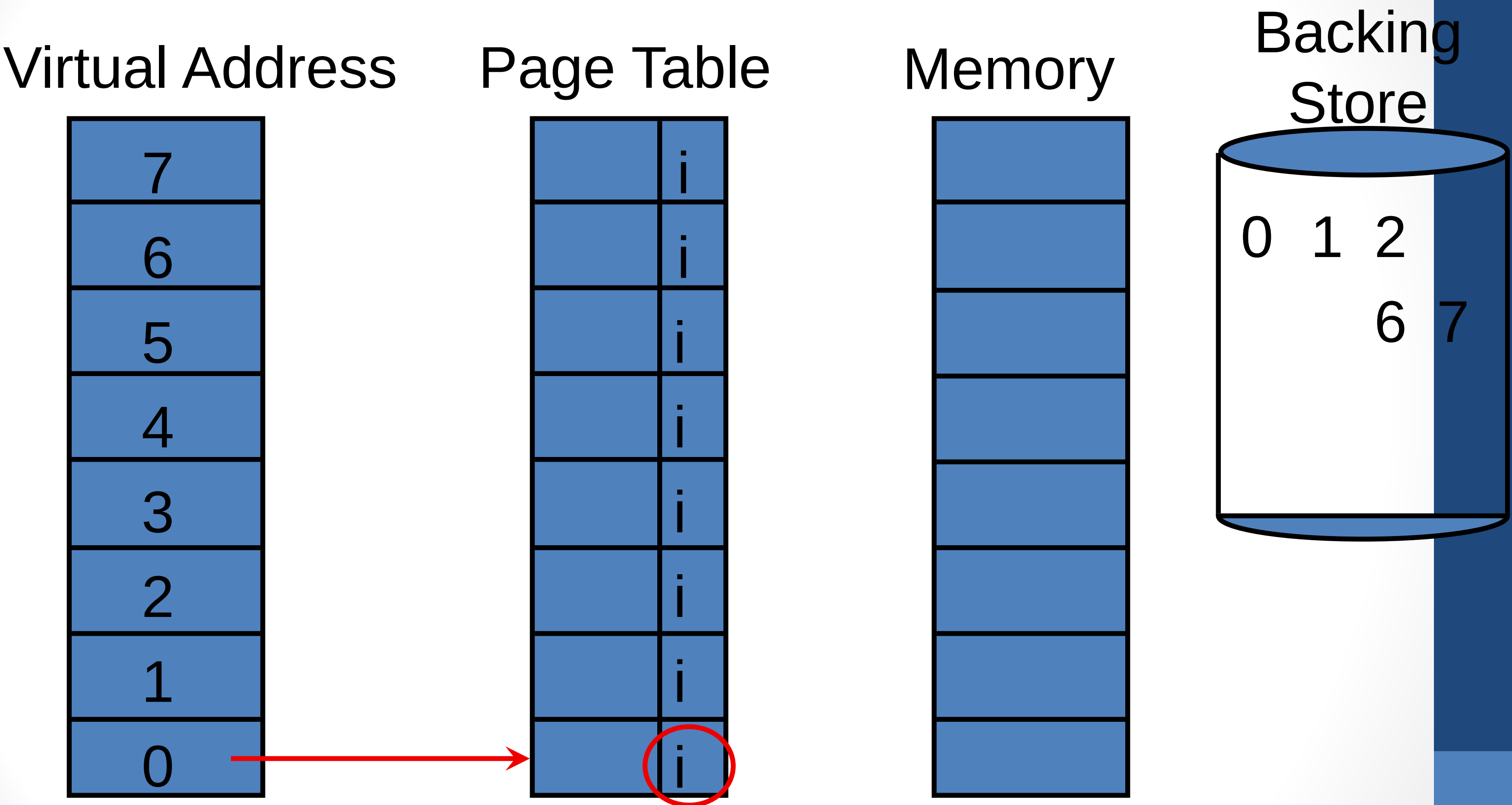
Demand Paging



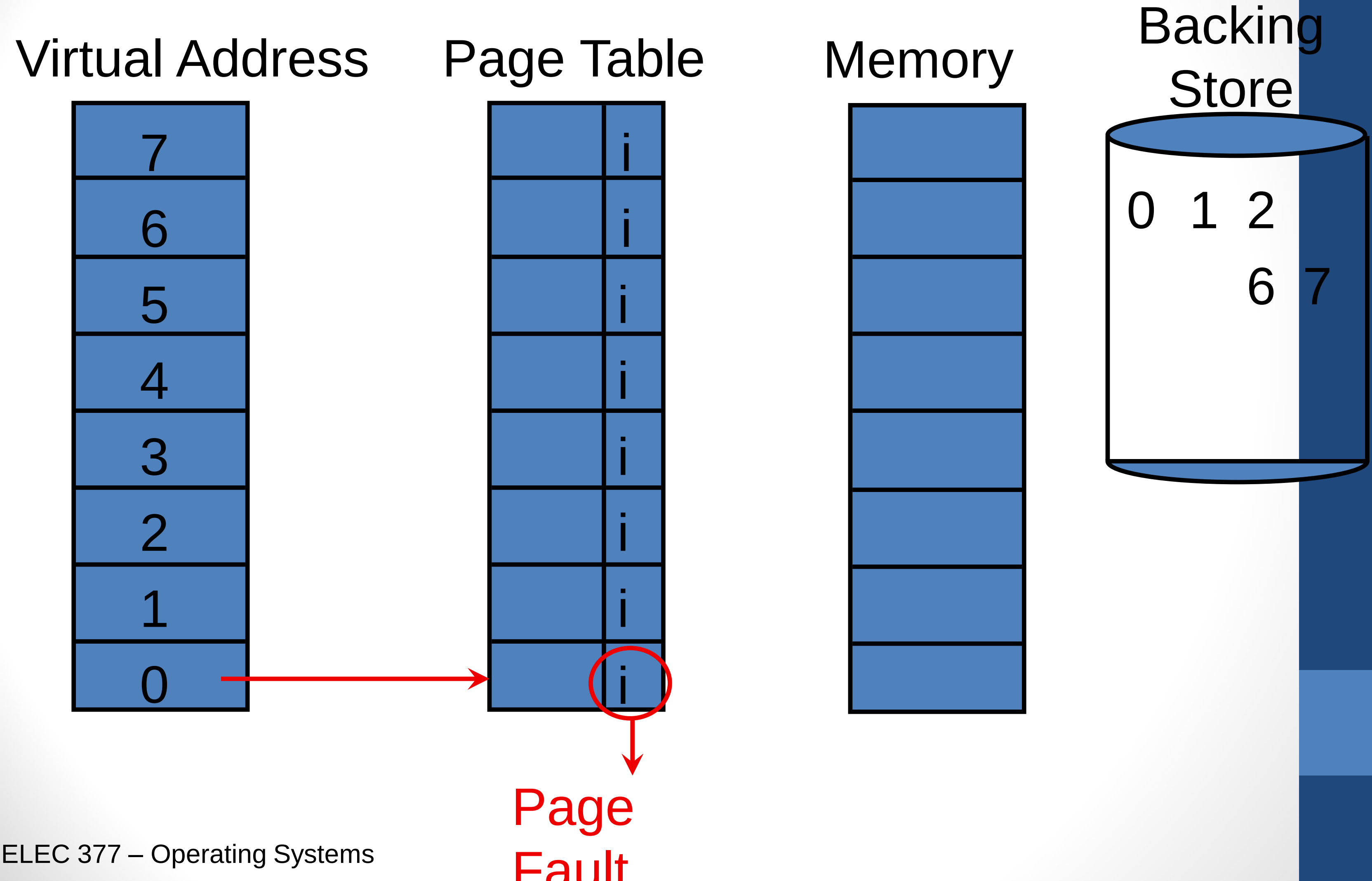
Demand Paging



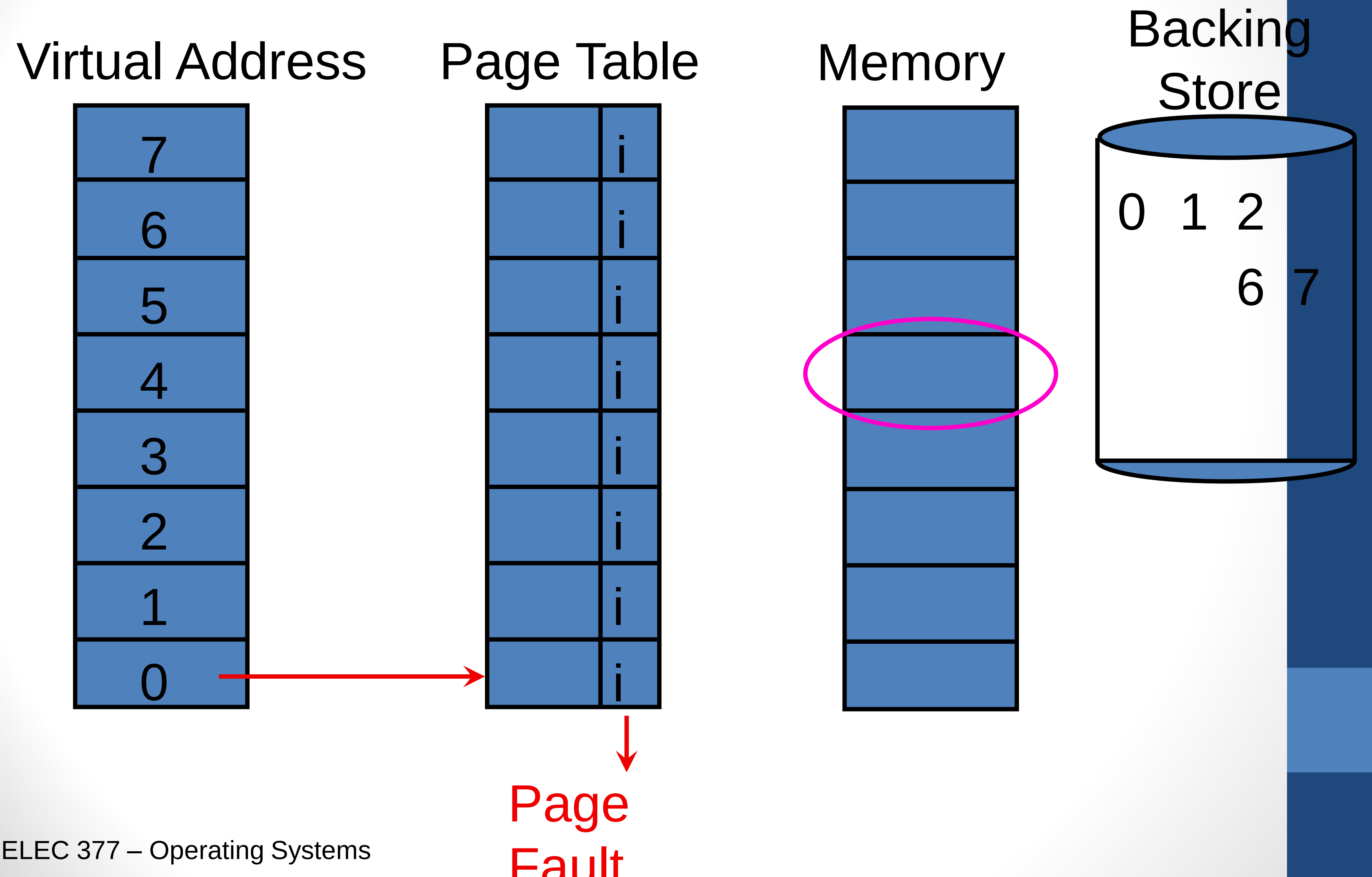
Demand Paging



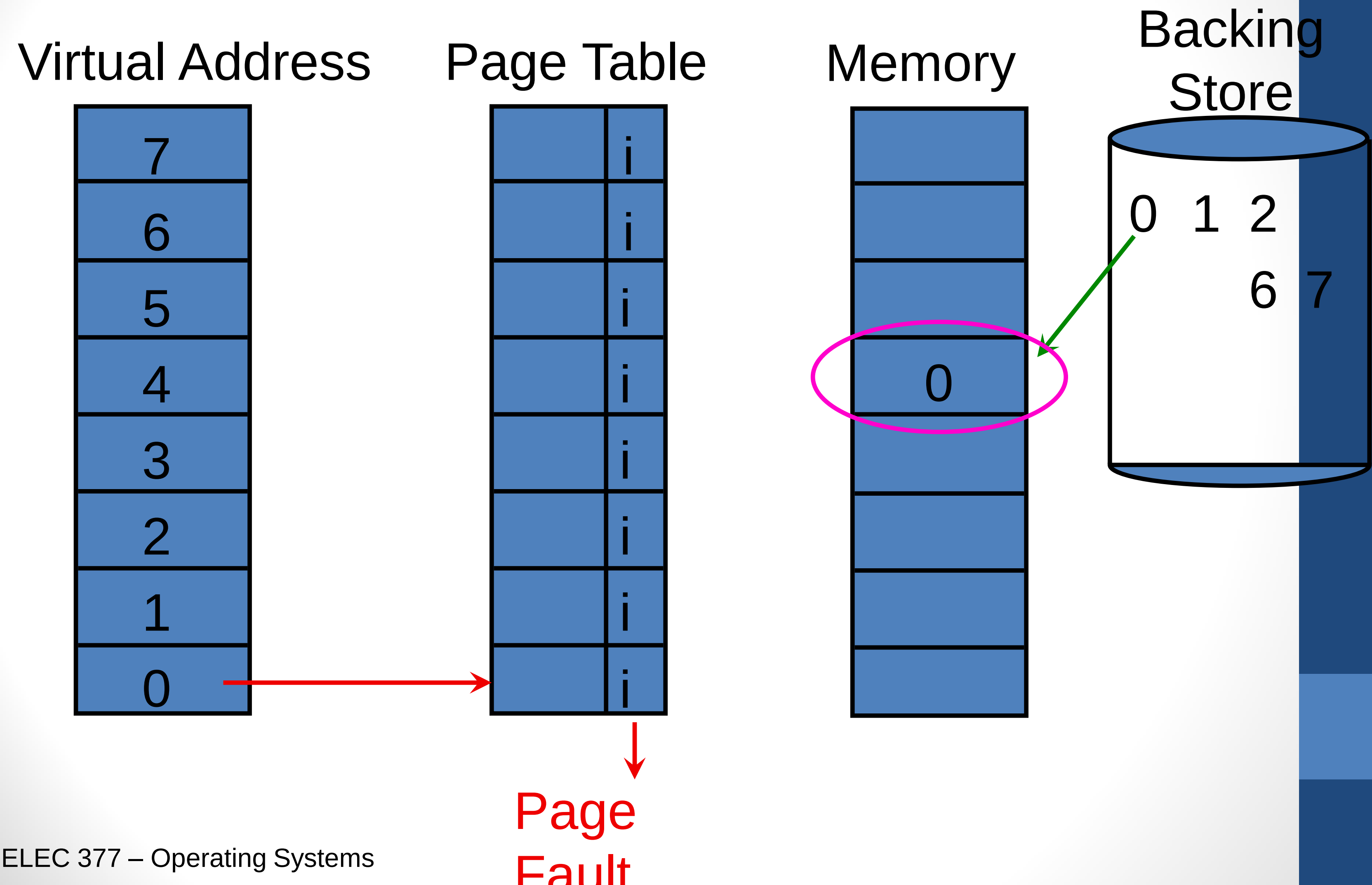
Demand Paging



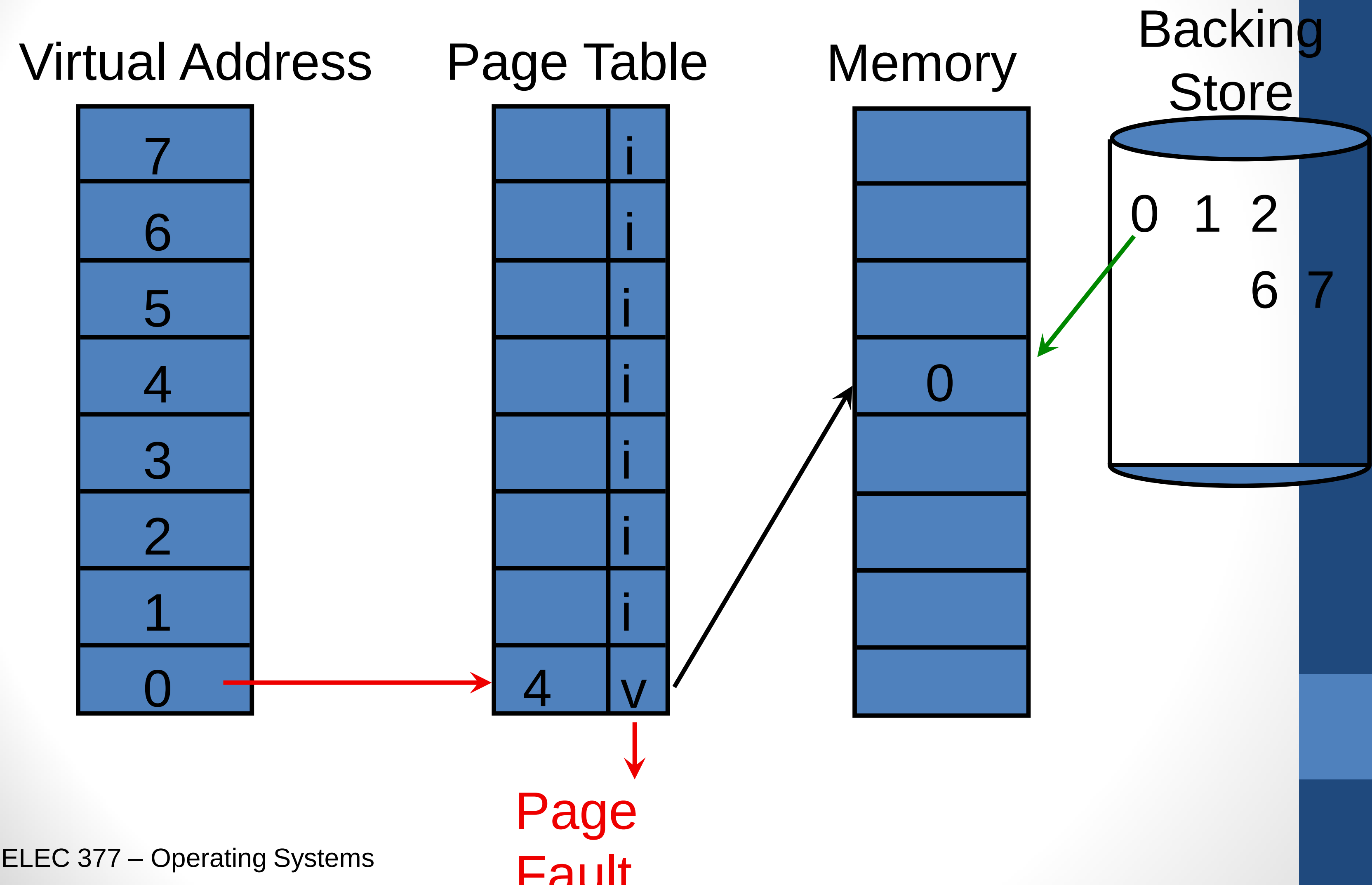
Demand Paging



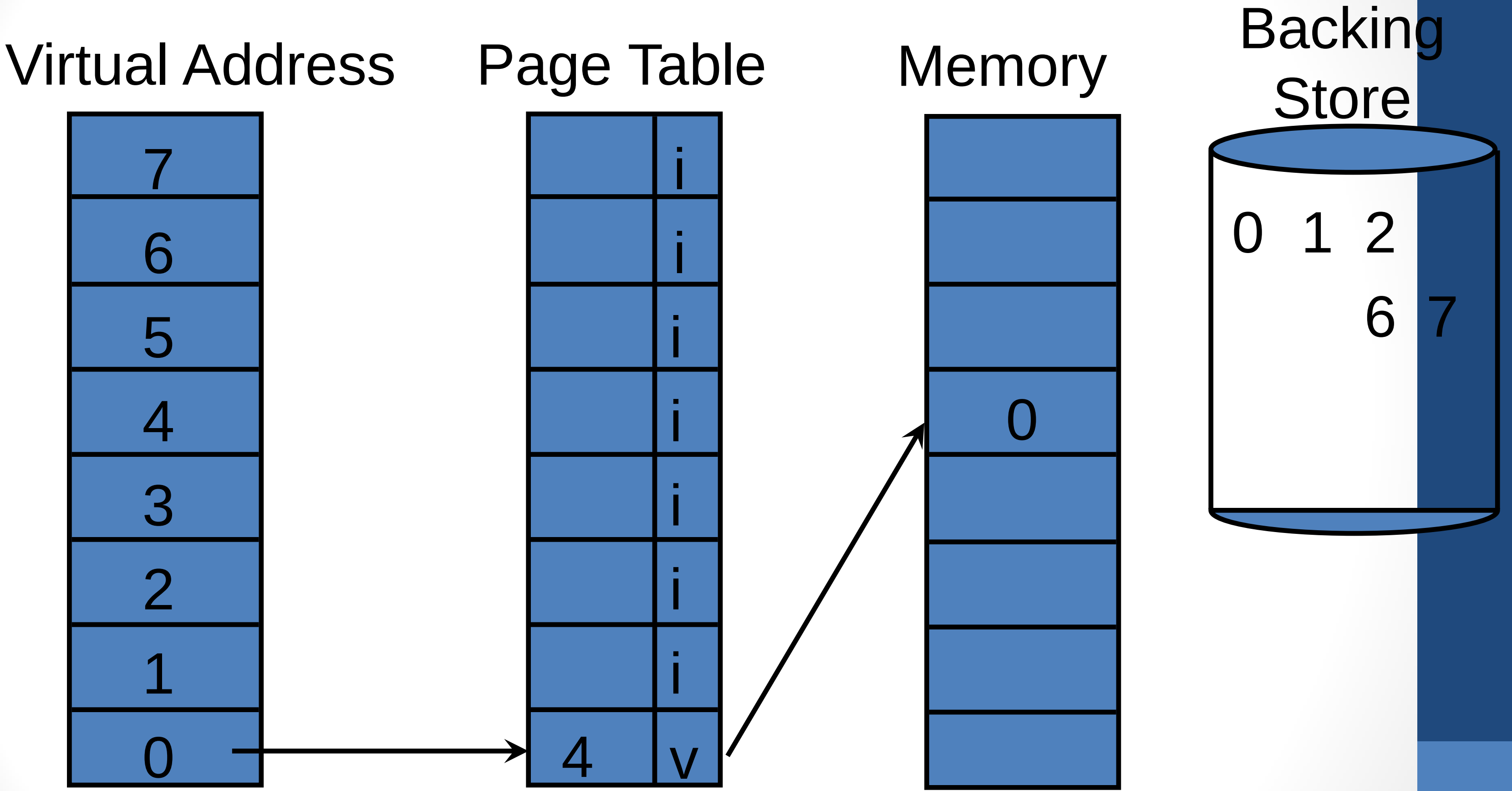
Demand Paging



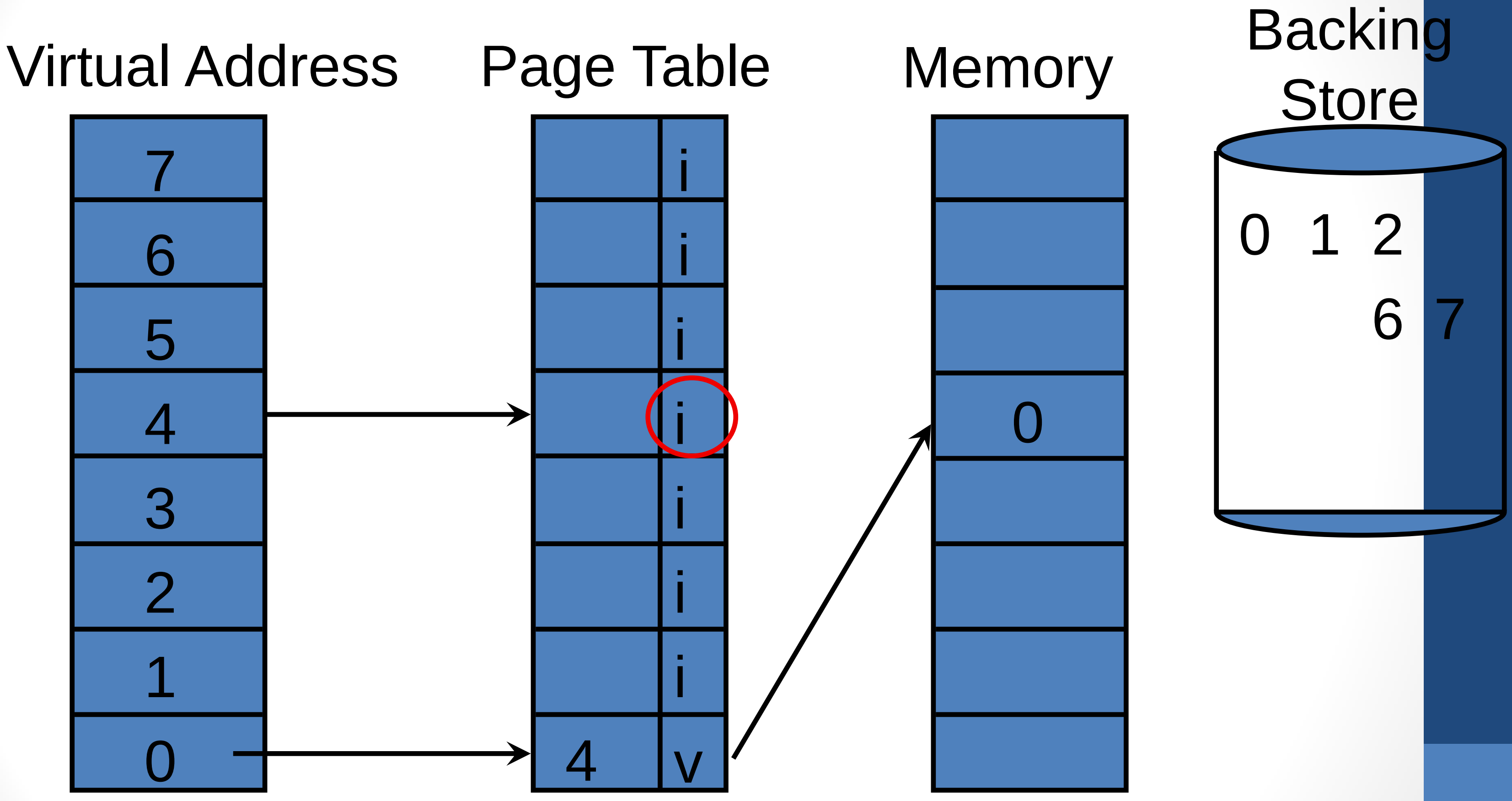
Demand Paging



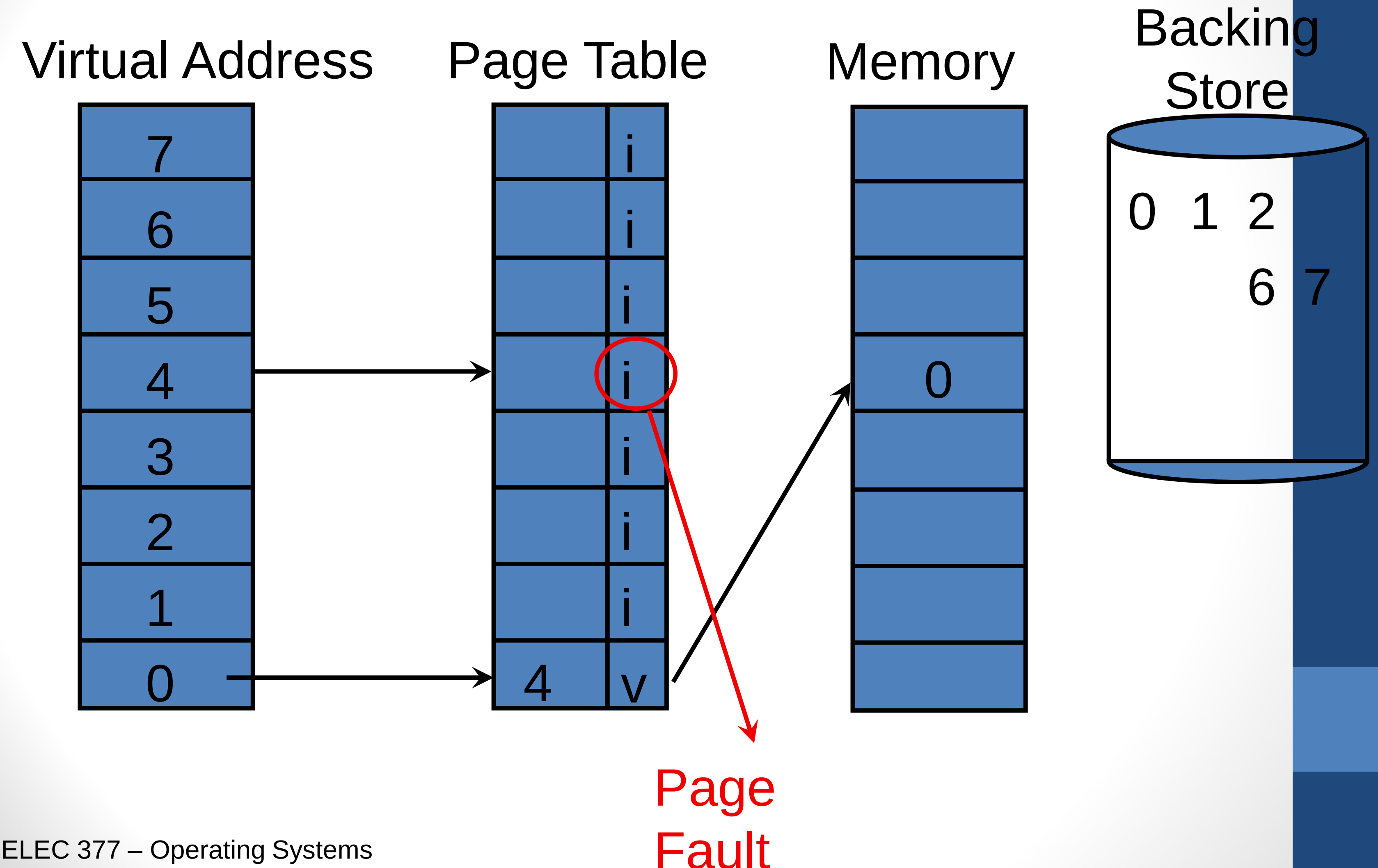
Demand Paging



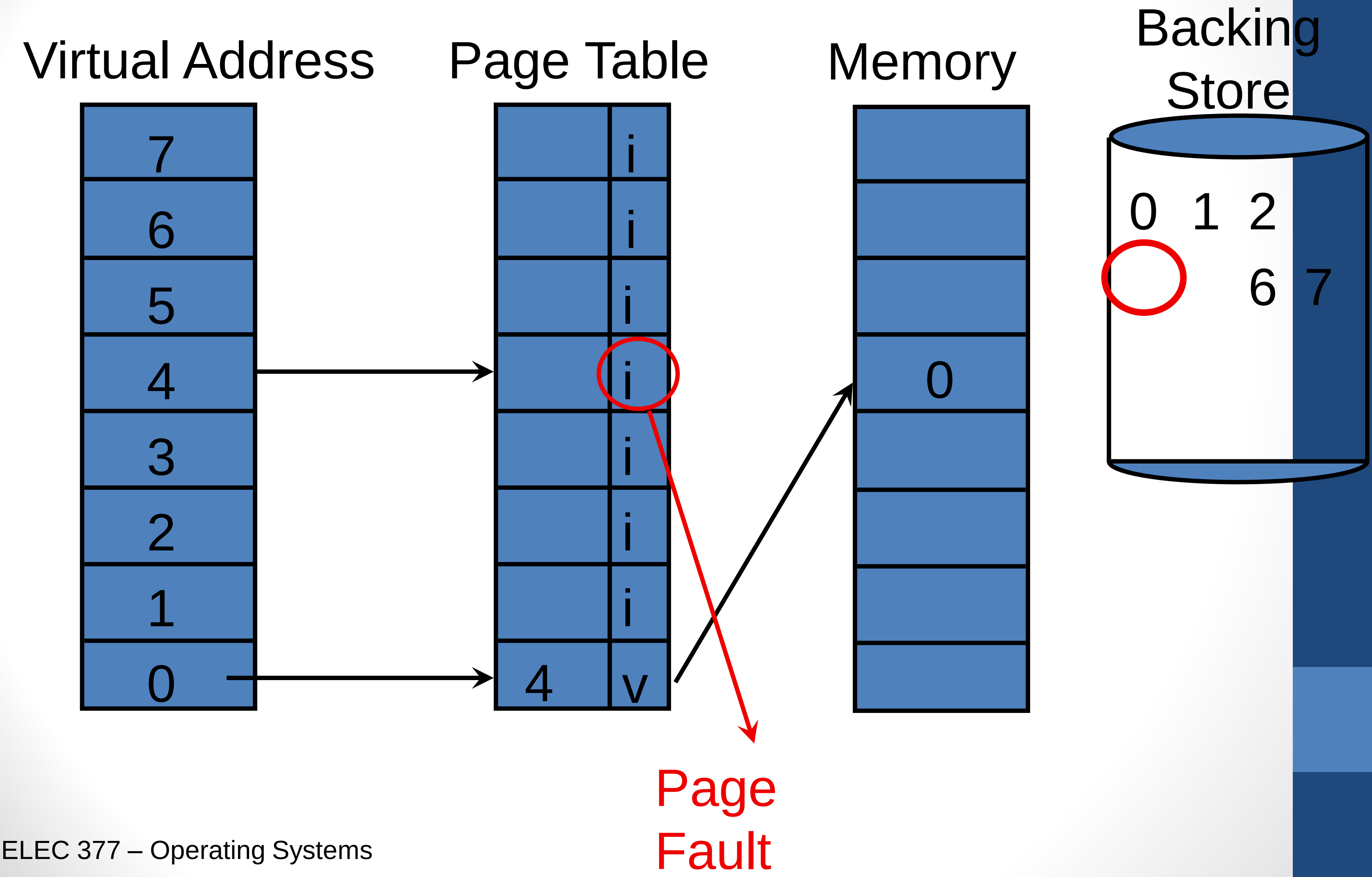
Demand Paging



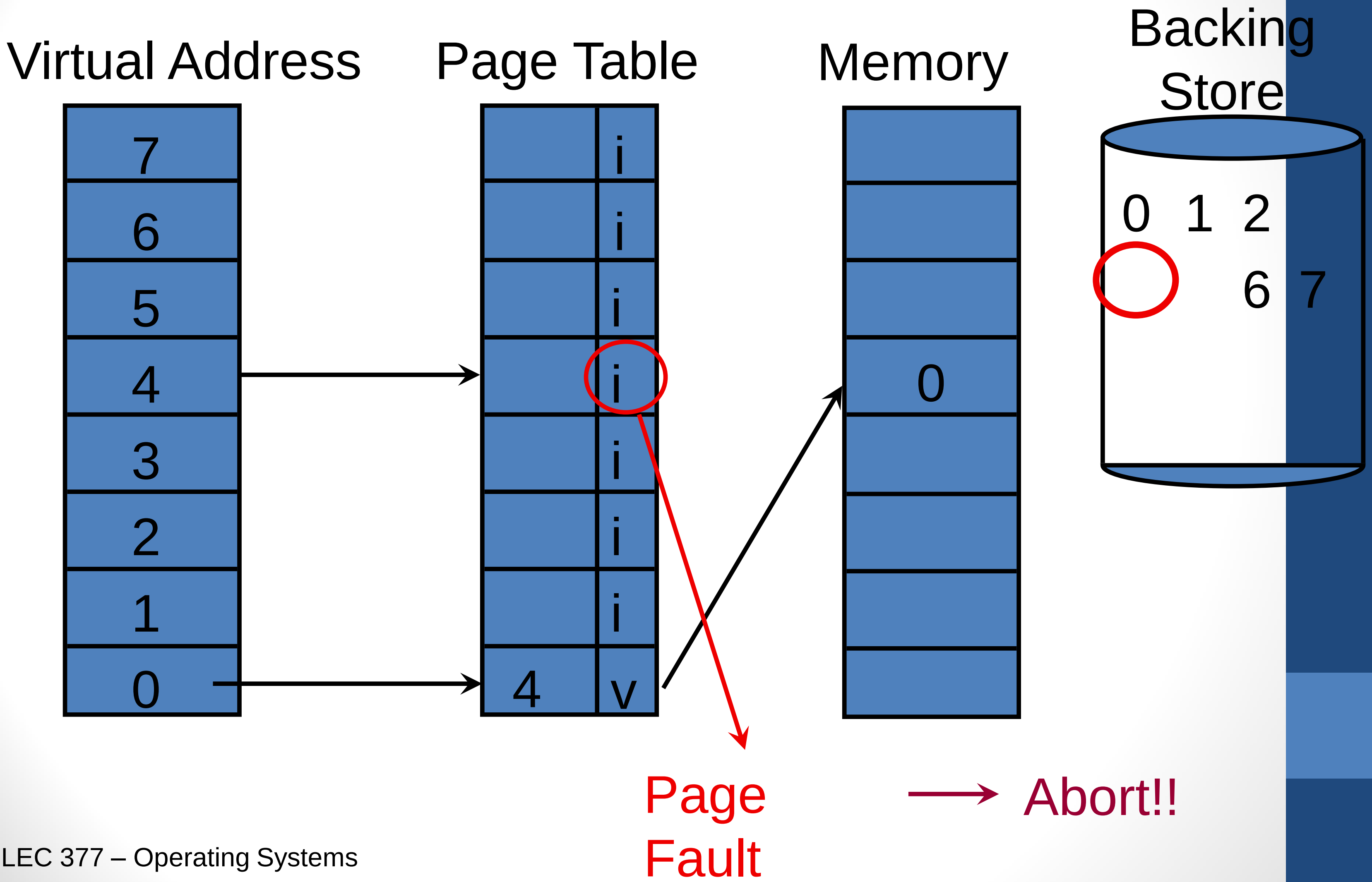
Demand Paging



Demand Paging



Demand Paging



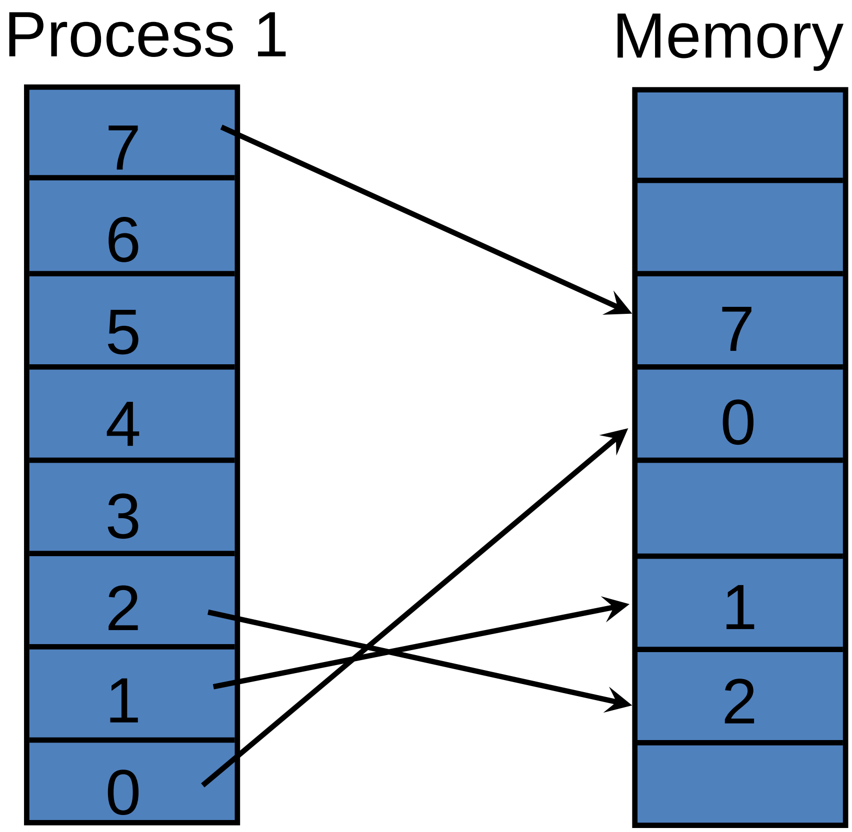
Demand Paging

- Pure demand paging (no pages to start)
- In practice, we know that at least the first code page will be used
- Pages that are in memory are called *memory resident*
- Restarting the instruction may cause more page faults
  - ◊ some instructions can access a lot of memory
    - – multiple indirection addressing modes
    - – memory copy (string move instruction on x86)
  - ◊ text talks about restarting instructions
    - – undoing side effects of instructions
  - ◊ some CPUs can suspend the instruction
    - – save internal registers on stack

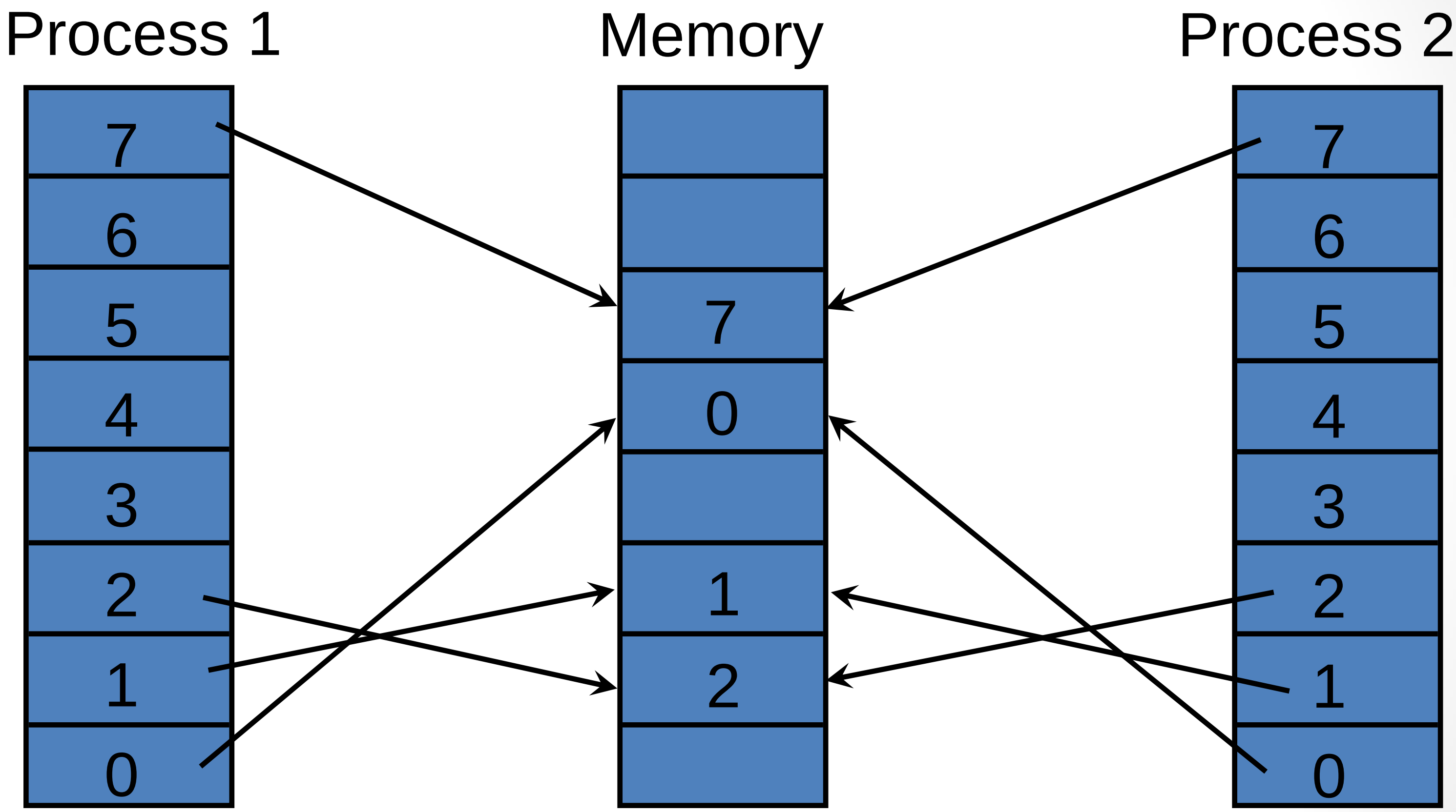
Copy on Write

- Unix fork()
  - ◊ duplicate child process
    - duplicate code
    - duplicate data
    - duplicate stack
- We already talked about sharing code pages, and separate data and stack pages
- On a fork(), we could share all pages
  - ◊ only copy a page when it is modified
  - ◊ requires support from hardware (MMU)
  - ◊ more efficient use of memory
  - ◊ don’t waste cpu cycles copying memory

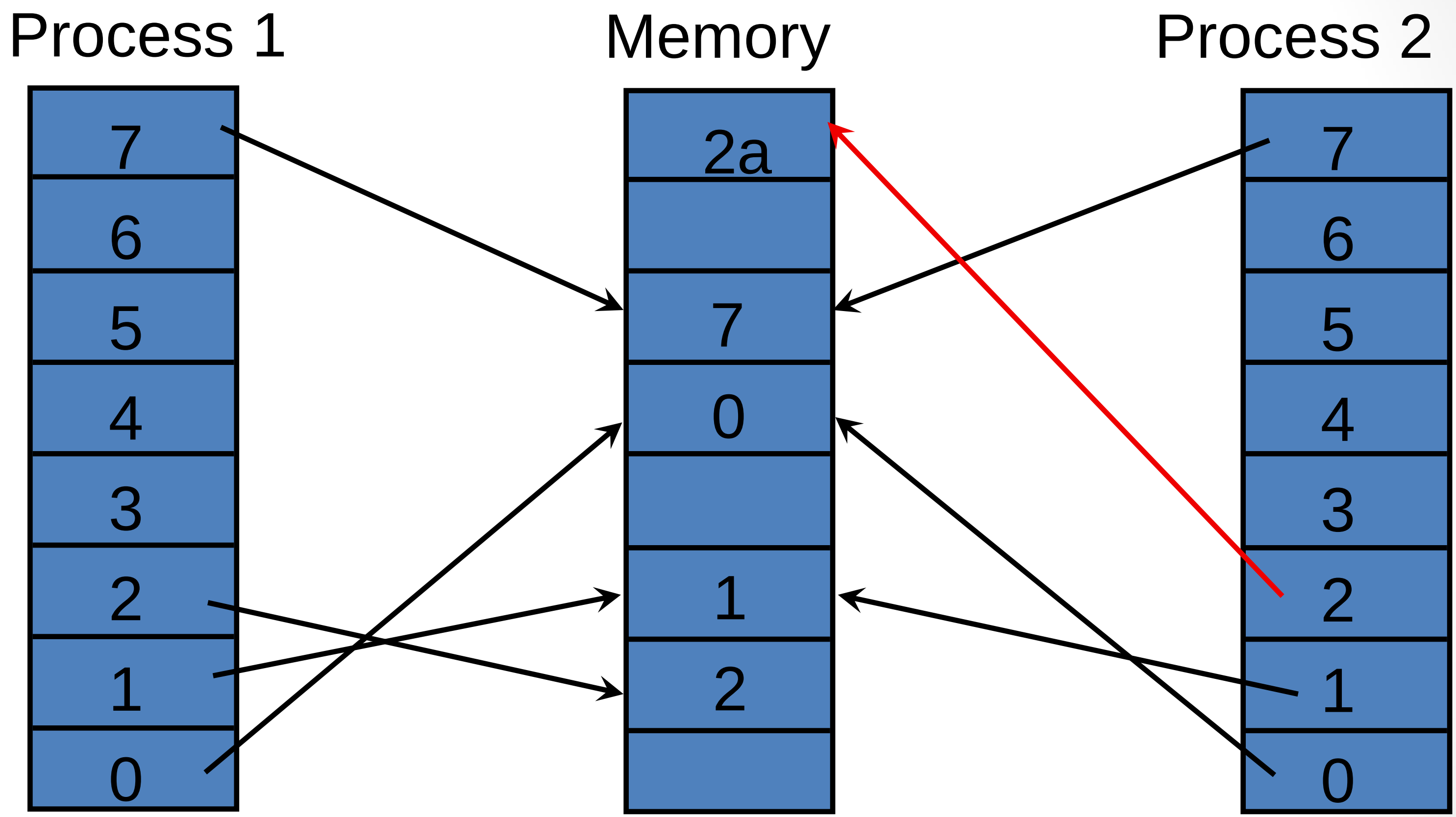
Copy on Write



Copy on Write – after fork()



Copy on Write – after P2 changes 2



Memory Mapped Files

- Traditional I/O
  - ◊ library accumulates output until a buffer is filled and then calls O/S to write the buffer to the file
  - ◊ input is read a buffer at a time from O/S and then library gives it in smaller amounts
- Paging allows a different approach
- Take unmapped pages of the process space and map them to the blocks of the file.
  - ◊ page fault brings the block into memory
  - ◊ when the file is closed, write all modified blocks
  - ◊ random access to data file!
  - ◊ files can be shared with write access!

Page Replacement

- What happens when we run out of memory?
  - ◊ running more processes than memory (*over allocation*)
  - ◊ no spare frames
  - ◊ select some other frame in physical memory
  - ◊ write its contents to disk
  - ◊ invalidate the MMU registers that point to the frame
  - ◊ reuse the frame
- Two transfers (write old contents, read new contents)

Page Replacement

- dirty bit?
  - ◊ add another flag to the page table
  - ◊ indicates that the page has been changed (dirty)
  - ◊ only write dirty pages (otherwise matches copy on the disk)

- Code pages are mapped from the program executable
  - ◊ since code doesn't change (reentrant), never have to write code pages.
  - ◊ Backing store only saves data and stack pages

Page Replacement Algorithms

- Similar to scheduling algorithms
  - ◊ want to minimize page faults

- FIFO
  - ◊ Not particularly good
  - ◊ Belady's Anomaly
    –more memory more page faults

- Optimal
  - ◊ similar to Shortest Job First scheduling algorithm
  - ◊ page that will not be used for the longest time
  - ◊ future knowledge

Page Replacement Algorithms

- LRU - Least recently used
  - ◊ past behaviour predicts future behaviour
  - ◊ page referenced longest ago gets replaced
    - – hardware support(page counters, stack)
- Approximation
  - ◊ reference bits (history of page references)
  - ◊ second chance algorithm (FIFO with 1 ref bit)
- Alternatives
  - ◊ include modified bit
    - – prefer clean pages to dirty pages
    - – not as important as recently used reference bit, breaks ties