csci 3411: Operating Systems

Virtual Memory

Gabriel Parmer

Slides adapted from Silberschatz and West

Safety, Liability, and Software

- Consumer protection from engineering products
  - I'm not talking about skynet...
- Consumer protection from software?
- Fundamentally different?

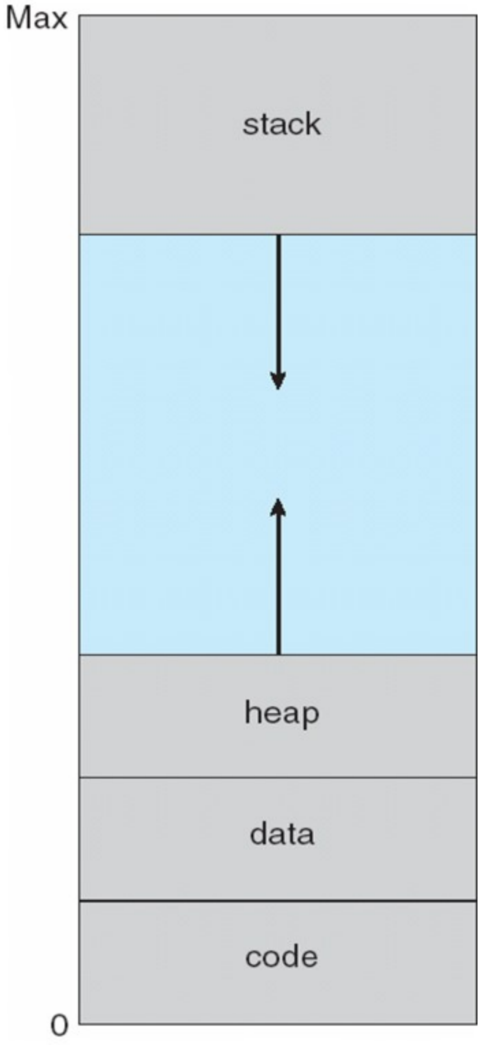
Virtual Memory

> *All problems in computer science can be solved by another level of indirection*
> *– Butler Lampson*

- *Indirection* – don't access the *thing* directly, ask something where you can access the *thing*
- OS/Hardware provide virtual address spaces
  - Separation of application's view of memory, and actual memory
  - Map virtual addresses to physical addresses
- Page tables provide this indirection
  - “Where can I find the [physical] memory for this [virtual] memory access?”

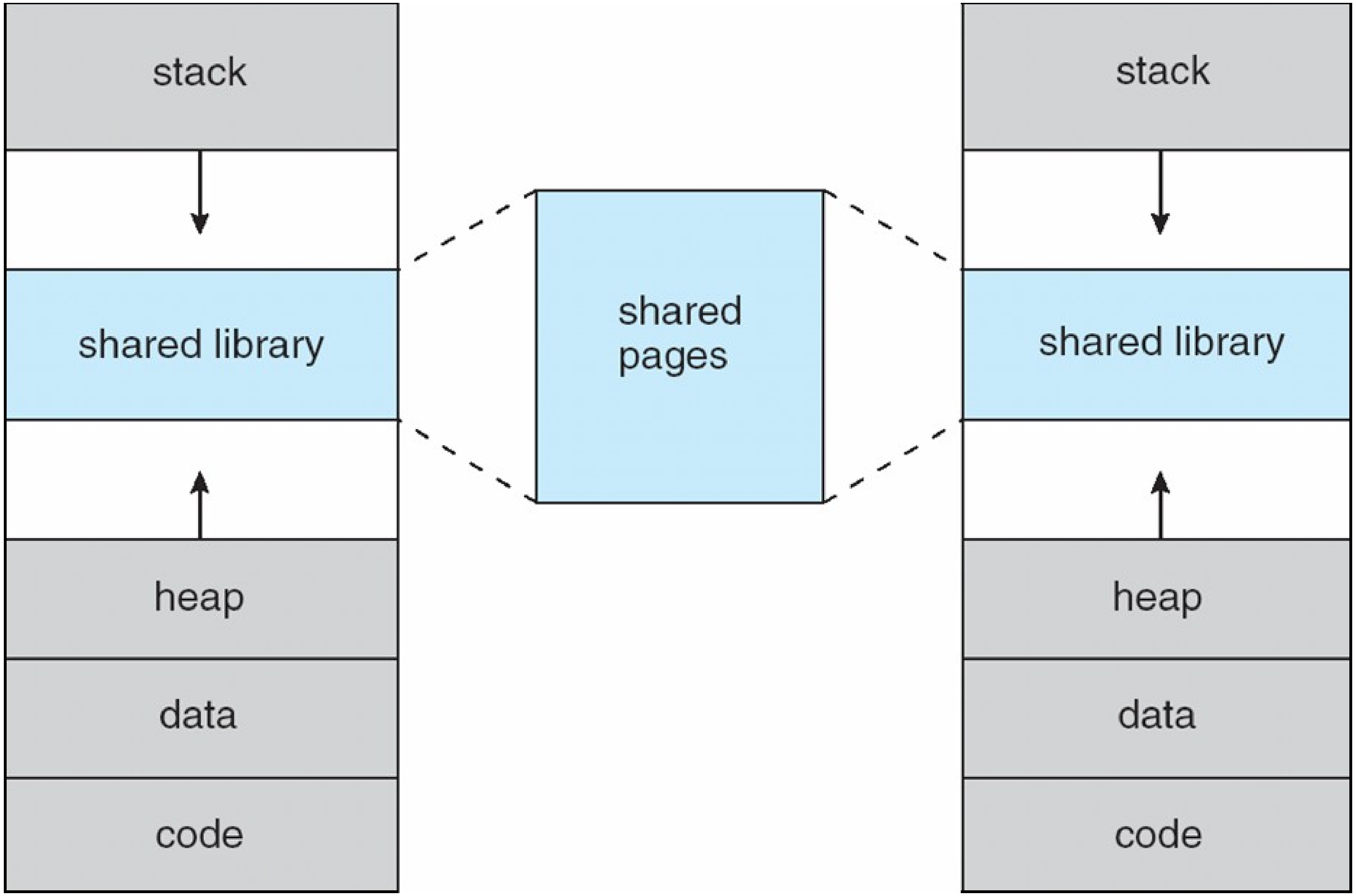
- Benefits of a virtual ↔ physical address separation

Process Virtual Address Space (VAS)



- Illusion of resource usage monopoly
  - VAS abstraction
- *Protection*
  - Fault isolation
    - Because humans mess up
  - Security

Shared Memory



- Firefox:
  - Virtual Memory used: 754MB
  - Shared Pages: 38MB (that's about 40MB saved!)

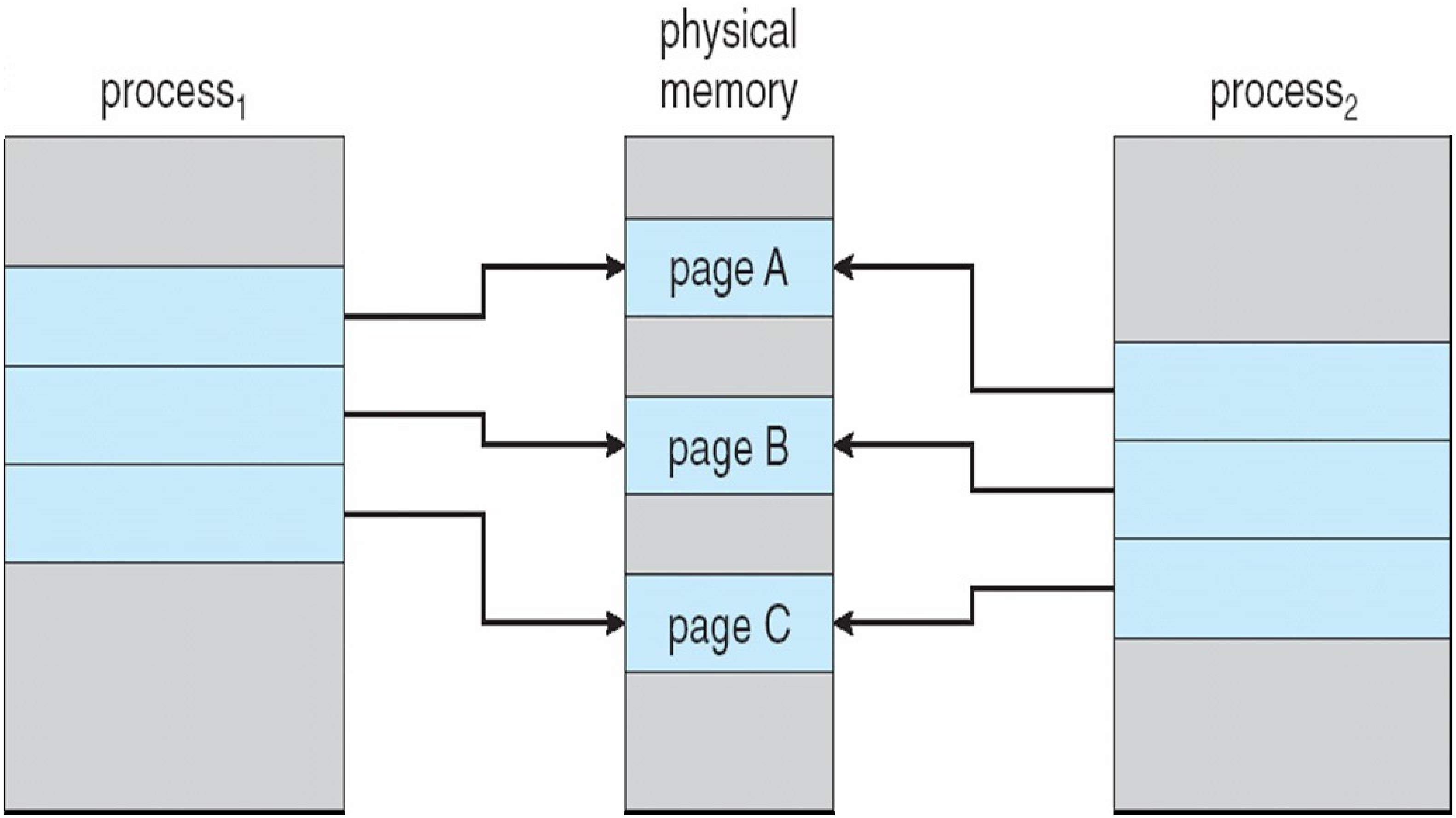
Process Creation: *fork()*

- Remember:
  - Process is an executing *program*
  - *fork()* system call creates a copy of a process, and resumes execution in both child and parent
- How is this implemented?
- Opportunities for optimization?

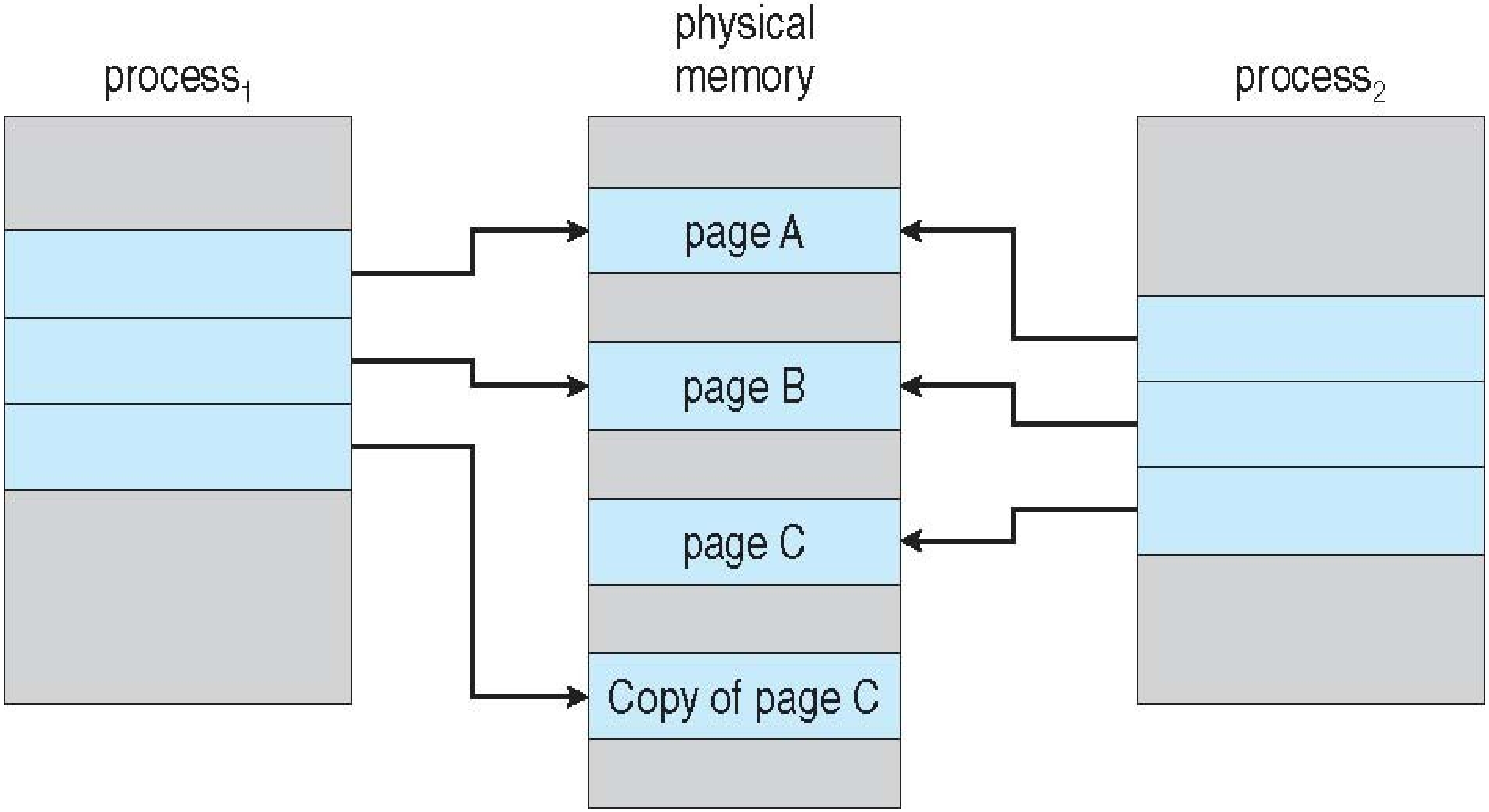
Process Creation II

- *fork()* implementation options
  1) Copy all memory for a process, create a new page-table, start child process
  2) Don't copy *any* memory upon *fork*, instead
     1) Ensure that memory cannot be modified
     2) Copy memory *lazily* only when the process modifies (writes to) it
        - Child and parent still effectively have *copies* of address space

COW: Before P1 Modifies Page C



COW II


process$_1$
physical memory
process$_2$
page A
page B
page C
Copy of page C

Holy COW!

- Use page table support for read-only access on individual pages
  - Bits in page table for read, write, execute, valid/invalid
    - If a read-only page is written to, trap to kernel
  - Mark all pages in both parent's and child's page-tables as read-only
  - When memory write is made, copy only pages being written to *lazily*
    - Copy-On-Write (COW)
- *fork()* is now faster! Or is it???
- *In which cases is fork() faster? Slower?*
  - *Does it hurt, or help interactivity?*
  - *Common fork use cases...*

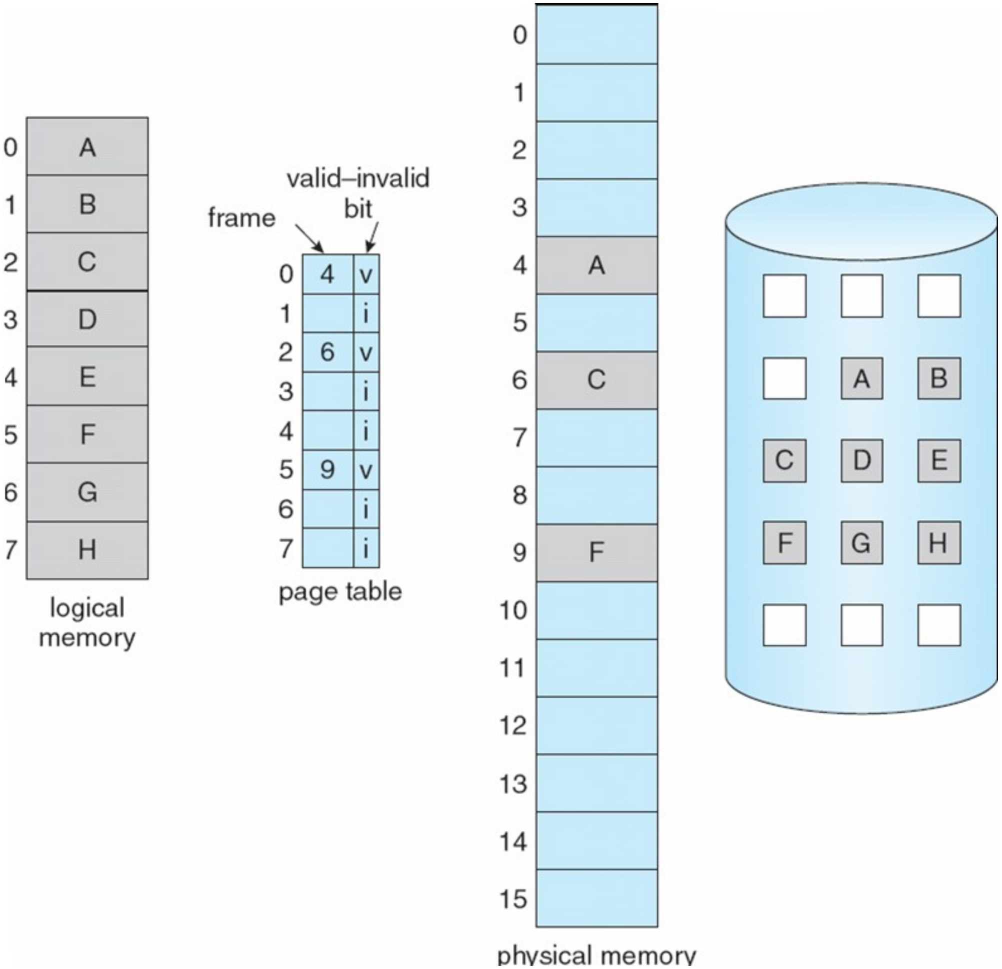
Program Execution: *exec()*

- Remember
  - *exec()* will stop execution in the current process and begin executing a program on disk
- How is this implemented?
- Room for optimization?
  - Hint:
    - Firefox
      - virtual memory used: 754MB
      - memory resident (backed by frames): 410MB

Demand Paging

- *exec()*: Must load a program from disk into memory
- Options
  1) Load program all at once
     1) Pull all of program from disk into memory
     2) Load program into virtual memory of process
  2) Demand paging
     1) Create an initially empty virtual address space
        - Page table entries are marked *invalid*
     2) As faults occur, load program from disk into virtual memory *on demand*
     - Benefit: load only that memory of program needed *now*
     - Speed up program loading/interactivity (less mem/I/O)

Demand Paging III



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
  - EAT = $(1 – p)$ x memory access

    \+ p (page fault overhead

    \+ swap page in

    \+ restart overhead)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

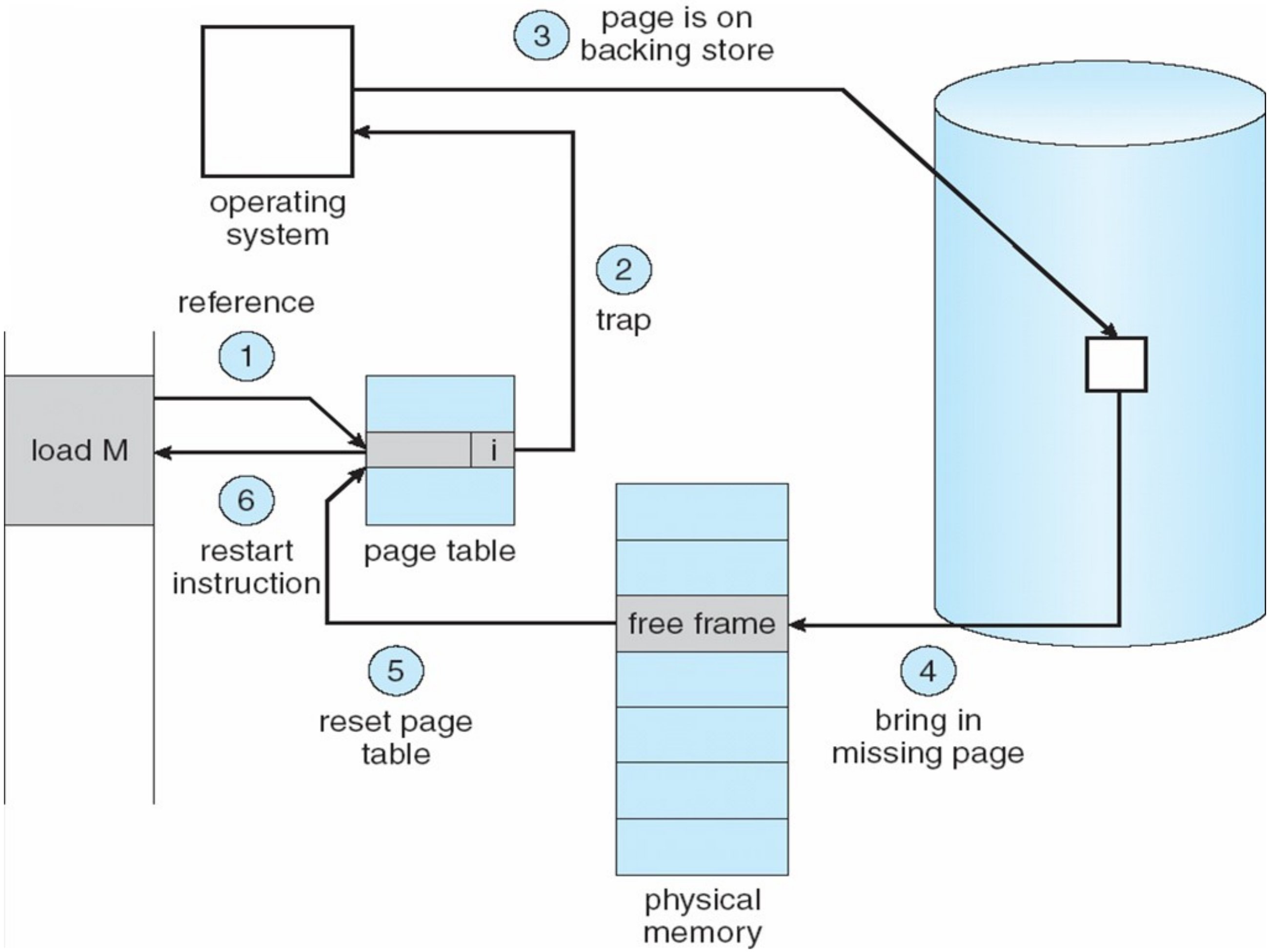
- EAT = $(1 - p) \times 200 + p$ (8 milliseconds)

  $= (1 - p) \times 200 + p \times 8{,}000{,}000$

  $= 200 + p \times 7{,}999{,}800$

- If one access out of 1,000 causes a page fault, then
  - EAT = 8.2 microseconds
  - This is a slowdown by a factor of 40

Handling a Page Fault



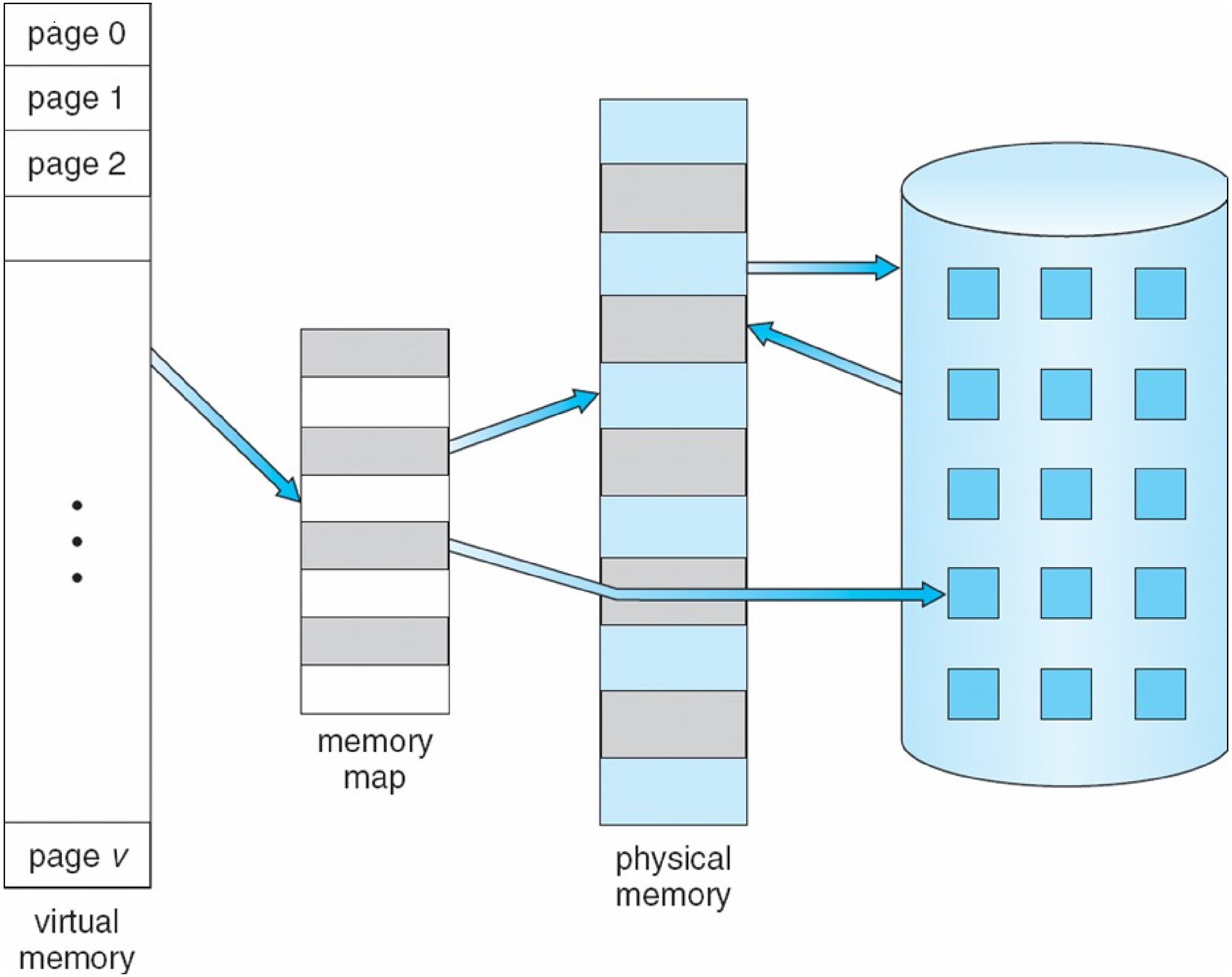
Virtual Memory

- Lets use this to do something really clever!
  - Storage hierarchy: remember, we want GBs of storage, all as fast to access as registers
    - We want to make memory look as large as disk, and as fast as registers
- Virtual Address space can be larger than system memory
  - Not all memory in the process is *resident* (backed by real memory)
- Only memory in *use* by a program must actually be in physical main memory
  - *Where can we put the memory not currently in use by a process?*

Low Memory Situations

- General System Goal: high resource utilization
  - Requires *multiprogramming/concurrency*
    - Increases memory usage
- What happens if we want to allocate memory and there is none?
  - Normal memory allocation request for a process
  - Page reference requires allocation due to
    - COW
    - Demand paging
- *Can we do something here, or do we just need to kill off a process?*

Disk: Part of the Storage Hierarchy



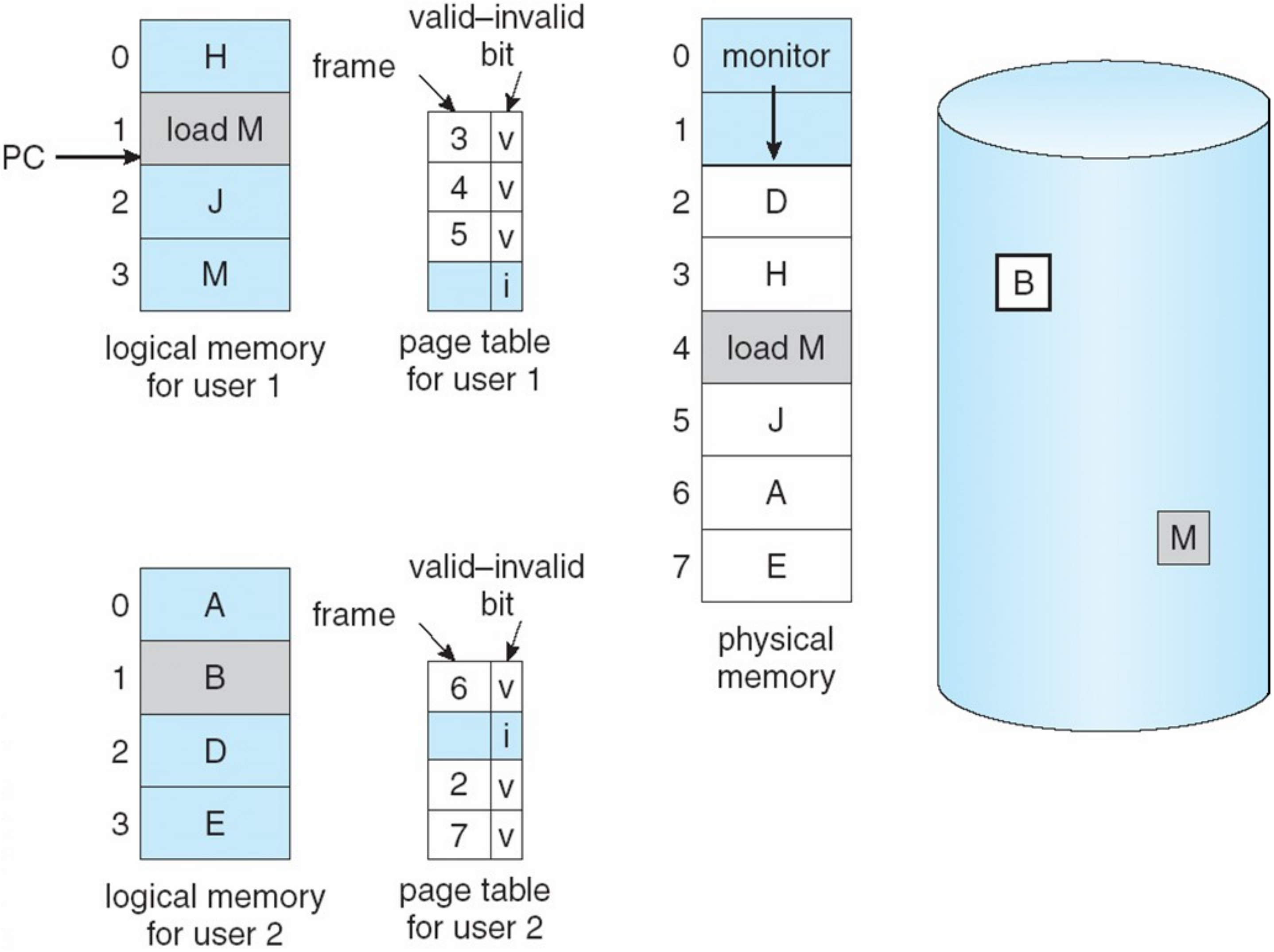
Page Replacement

- Use memory as a cache for disk

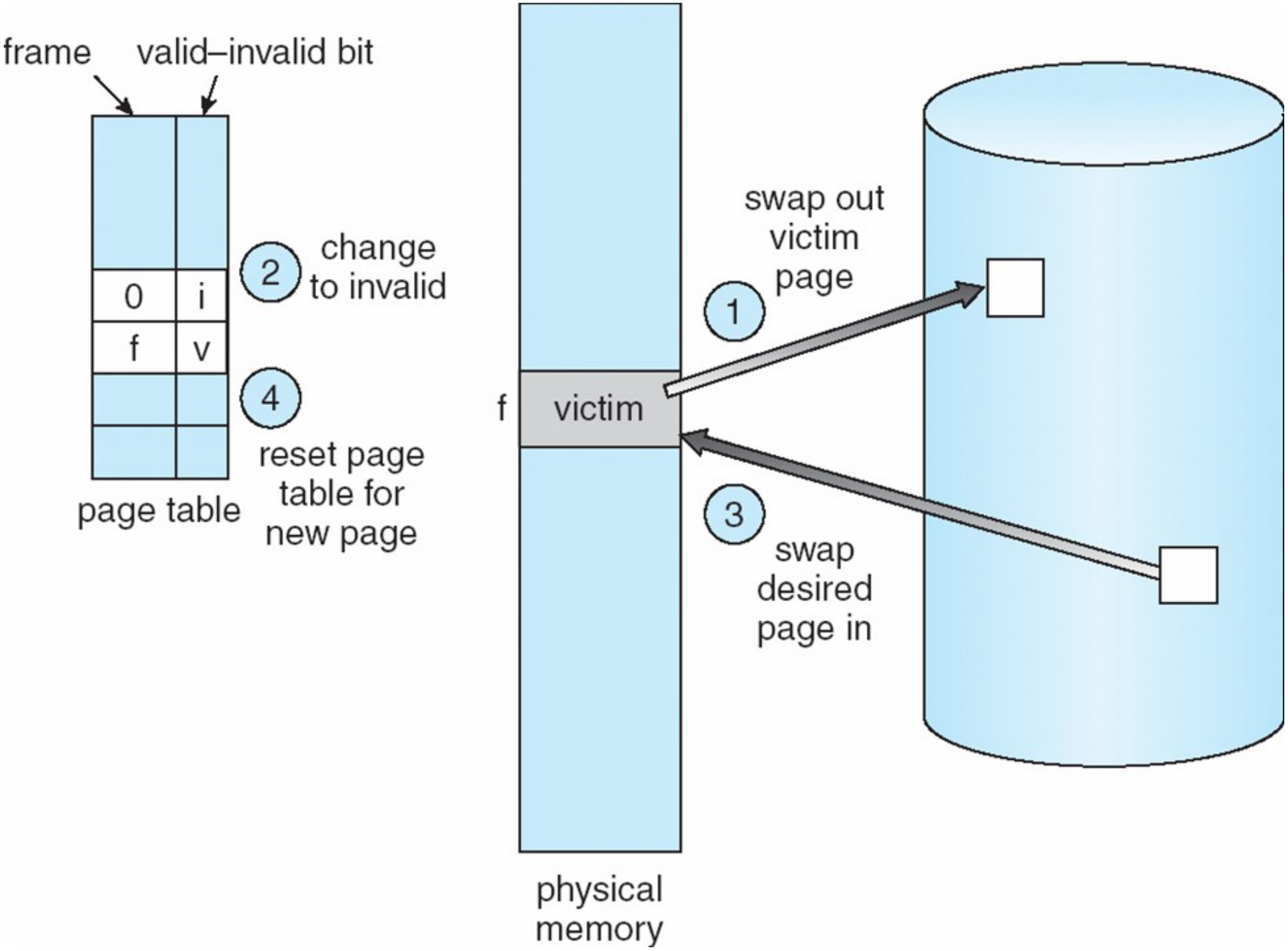
- Page replacement
  - Find a *victim* frame in memory
  - *Swap* it out – transfer it to disk to free that memory for other uses

- Swapping is the act of moving active memory back and forth from disk
  - Also called *paging* in page-based systems

Page Replacement II



Page Replacement III



How do we Choose a Victim Frame?

- Going to disk is *expensive*
  - Want to swap as infrequently as possible
- Find frame that is least likely to be referenced in the near future
- Optimization: consider frames that already exists on disk, and haven't been modified in RAM!
  - How is this more efficient?
  - Page tables include *modified* bit

- Algorithms for finding victim frame