Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T14

Beyond Physical Memory
Swapping Mechanisms

Referência principal
Ch.21 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 12 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

How can the OS make use of a larger, slower device to transparently provide the illusion of a large virtual address space?

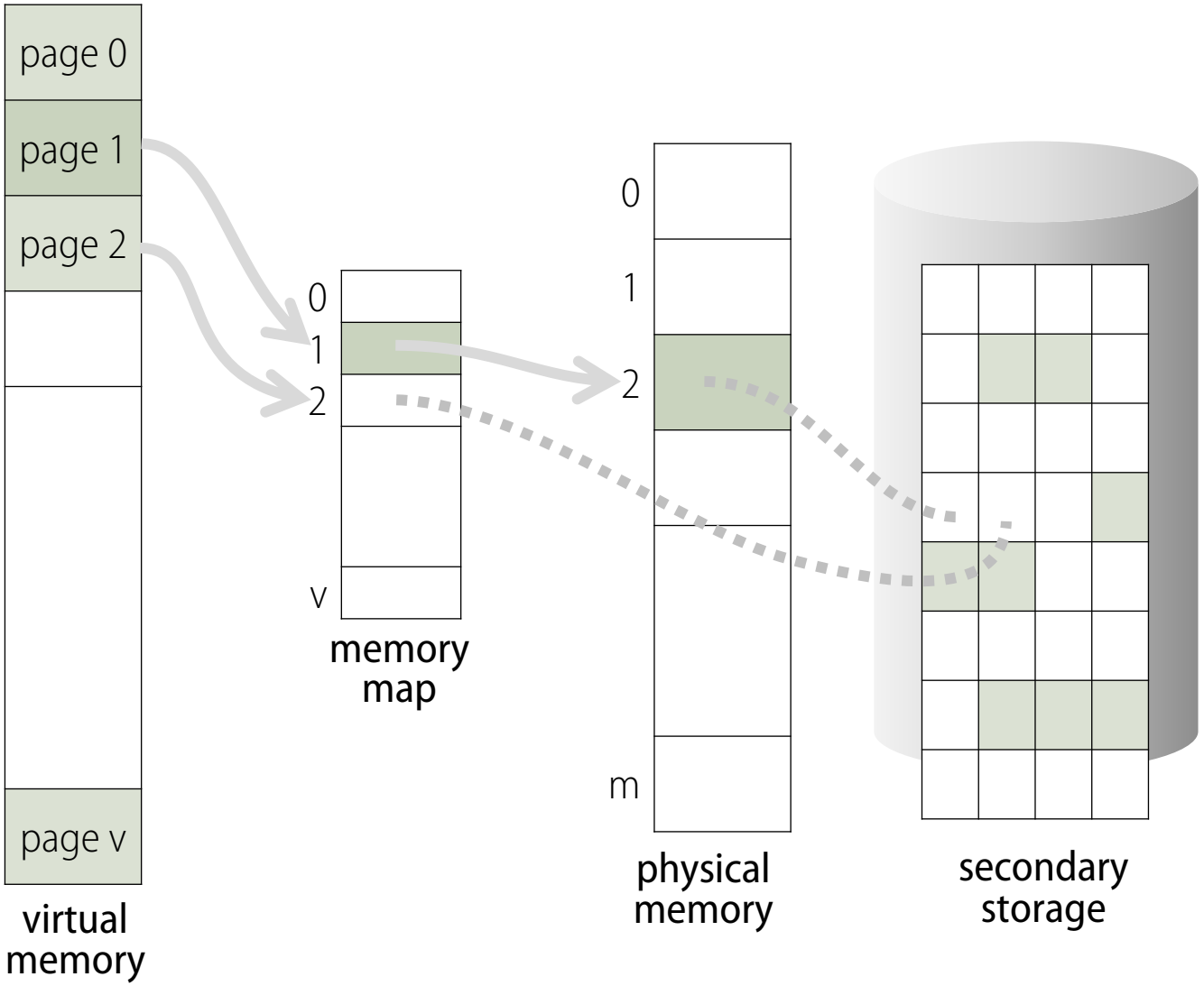
Background

- The requirement that instructions must be in physical memory to be executed may limit the size of a program to the size of main memory.
- In many cases, parts of the program are not needed or, at least, not at the same time, e.g.
  - Code to handle unusual error conditions.
  - Data structures that are dimensioned in excess of actual need.
  - Rarely used options and features.

Background

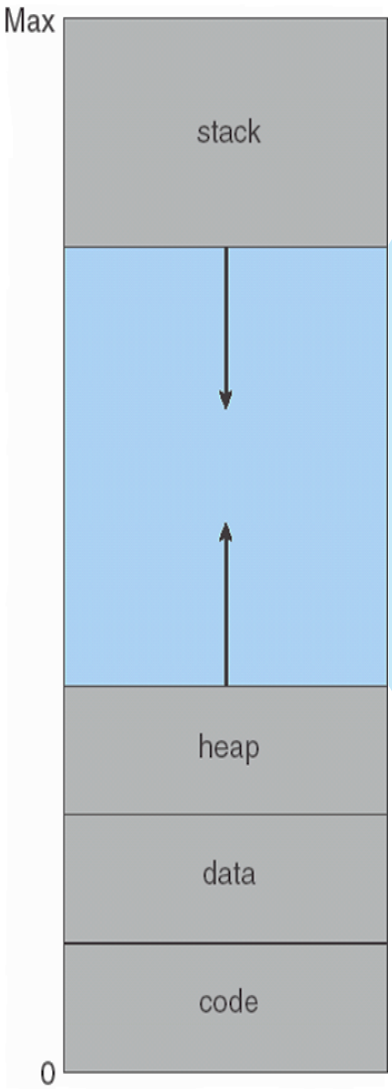
- The ability to execute a program which is only partially loaded in memory would have many benefits
  - The size of the program would no longer be constrained by the physical memory that is available.
  - Since programs would require less memory, the degree of multiprogramming could be increased.
  - Loading or swapping user programs into memory would require less I/O.
- Being able to run a program that is not entirely in memory would benefit both user and system.

Virtual Memory Can Be Much Larger Than Physical Memory



Background

- Virtual memory involves the separation of logical memory (perceived by users) from physical memory (available in the target machine).

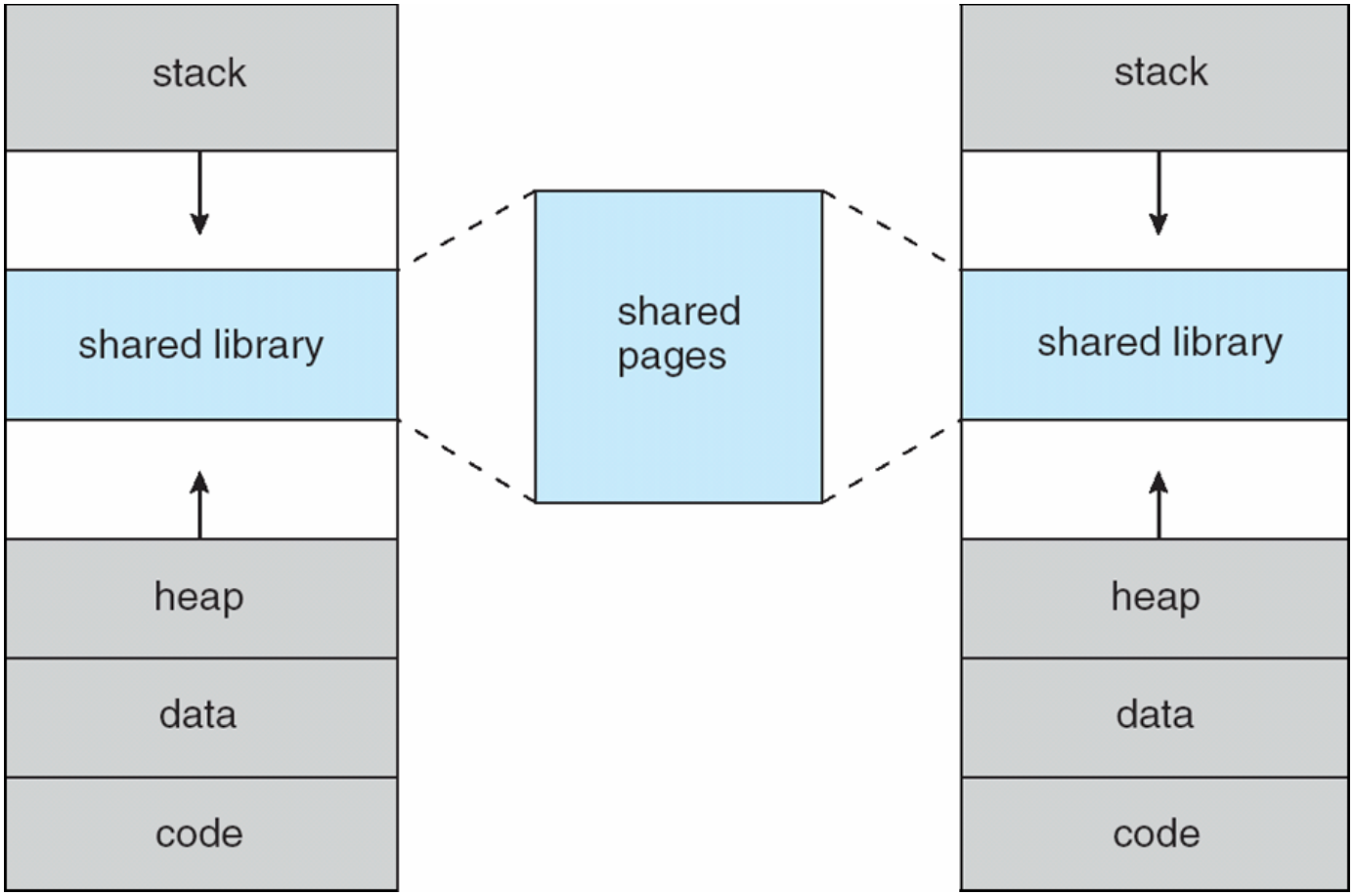


The space between the heap and the stack is part of the virtual address space but only will be mapped onto actual physical pages if the heap or the stack grows.

Background

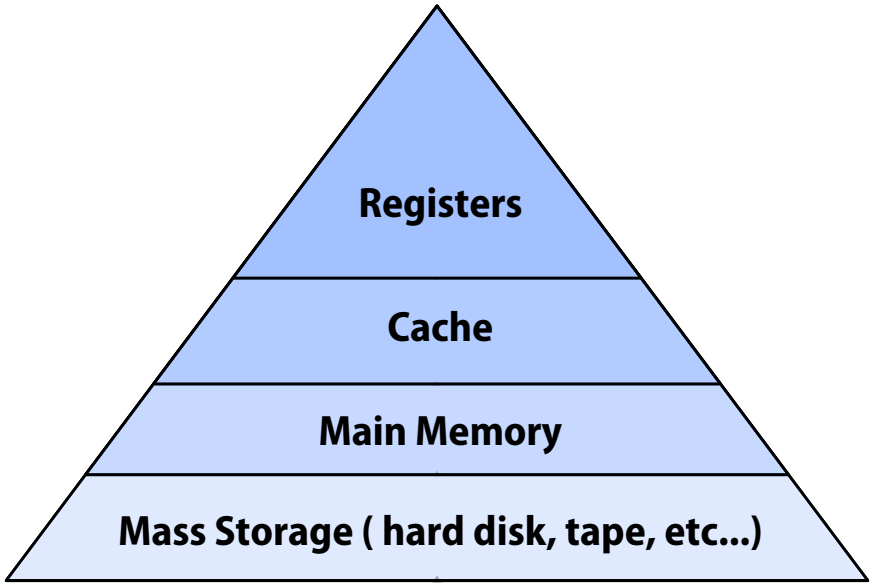
- Additional benefits of virtual memory are
  - System libraries can be shared by several processes through mapping of the shared object onto a virtual address space.
  - Processes can share part of their virtual address spaces, by mapping them onto actually shared memory pages.
  - Parent and child processes may share pages, thus speeding up process creation.

Several processes can share system libraries by mapping them onto a virtual address space.



Beyond Physical Memory: Mechanisms

- Supporting many concurrent-running large address spaces requires an additional level in the memory hierarchy.
  - The OS needs a place to stash away portions of an address space that currently aren't in great demand.
  - In modern systems, this role is usually served by a hard disk drive



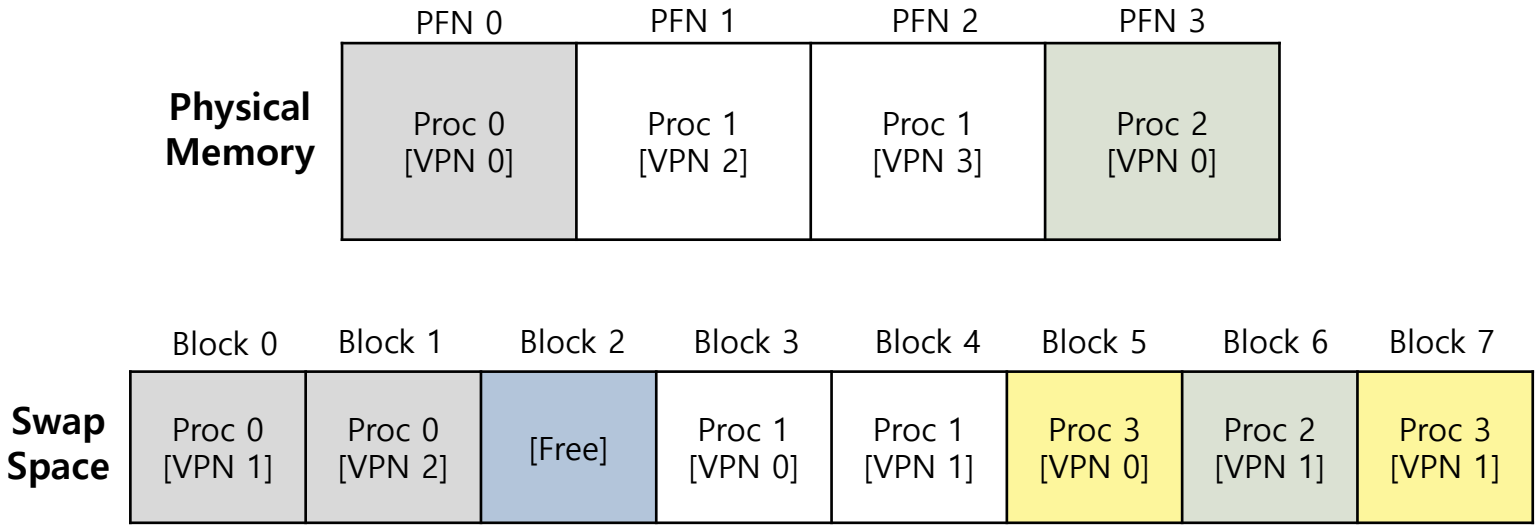
Memory Hierarchy in modern systems

Providing a single large address for a process

- We always need to arrange for the code or data to be in memory before calling a function or accessing data.
  - In older systems this was done via **memory overlays**, managed by the programmer.
- Going beyond just a single process…
  - The addition of **swap space** allows the OS to support the illusion of a large virtual memory for multiple concurrently-running processes.

Swap Space

- Reserved space on the disk for moving pages back and forth.
- The OS reads from and writes to the swap space in page-sized units.
- The OS remembers what pages of which processes have been swapped to a certain place on the disk.

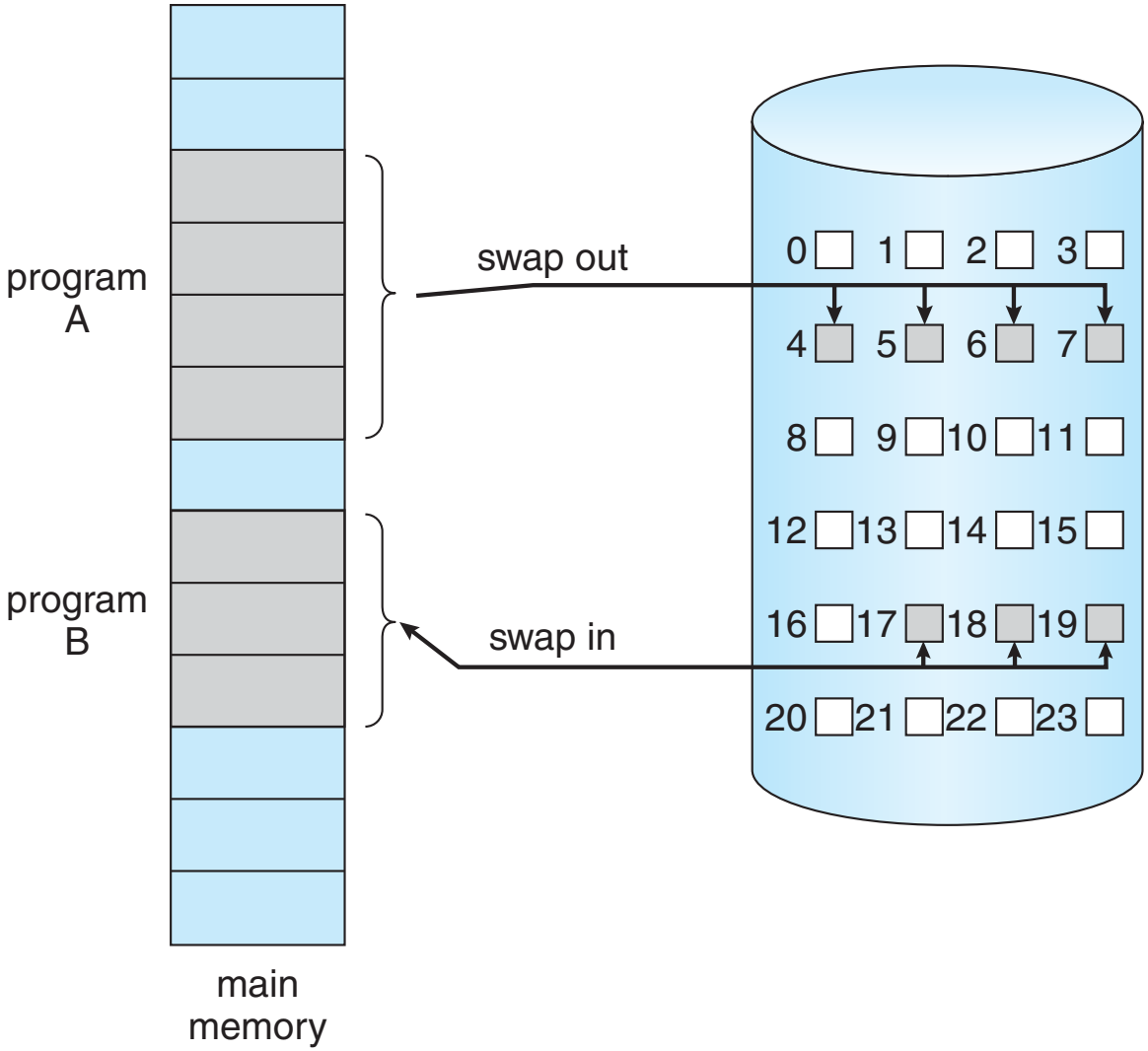


Physical Memory and Swap Space

Demand Paging

- **Demand paging** means bringing a page into memory only when it is needed.
  - This means that pages that are never accessed will never be loaded into physical memory.
- The model adopts a **lazy swapper**, which never swaps a page into memory unless it is needed.
  - A swapper that deals with pages is called a **pager**.

Transferring entire programs between main memory and contiguous disk spaces



When Replacements Really Occur

- OS waits until memory is entirely full, and only then replaces a page to make room for some other page.
  - This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.

- Swap Daemon, Page Daemon
  - When there are fewer than **LW pages** available, a background thread that is responsible for freeing memory runs.
  - The thread evicts pages until there are **HW pages** available.

The presence of a page in memory is indicated by a Valid-Invalid bit in the Page Table

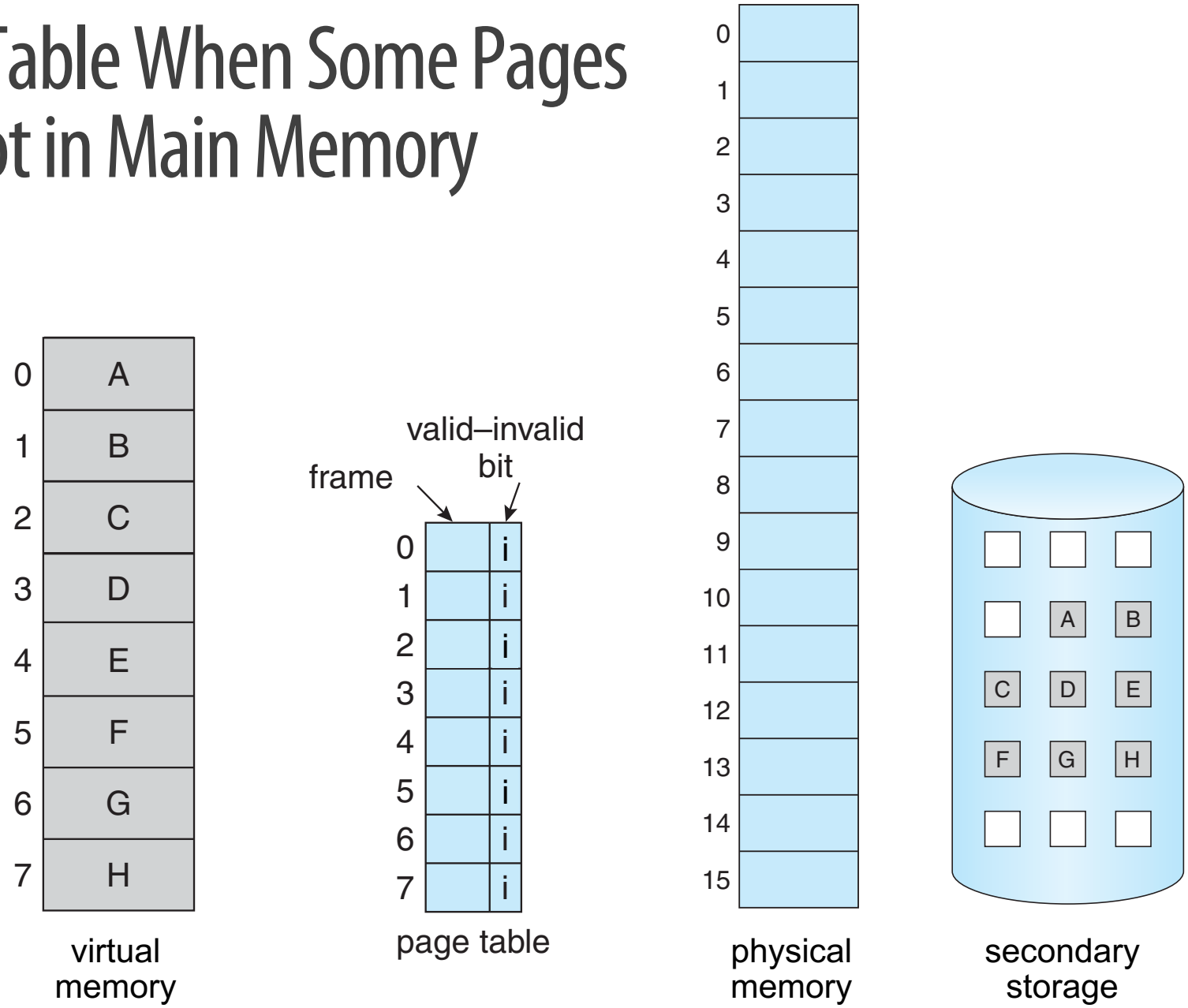
- A valid–invalid bit is associated with each page table entry:
  - **v** ⇒ legal and in-memory
  - **i** ⇒ illegal or not-in-memory
- Initially valid–invalid bit is set to **i** on all entries
- During a legal address translation, if valid–invalid bit in page table entry is **i** then there is a page fault

- A snapshot of the Page Table of a 4-page program

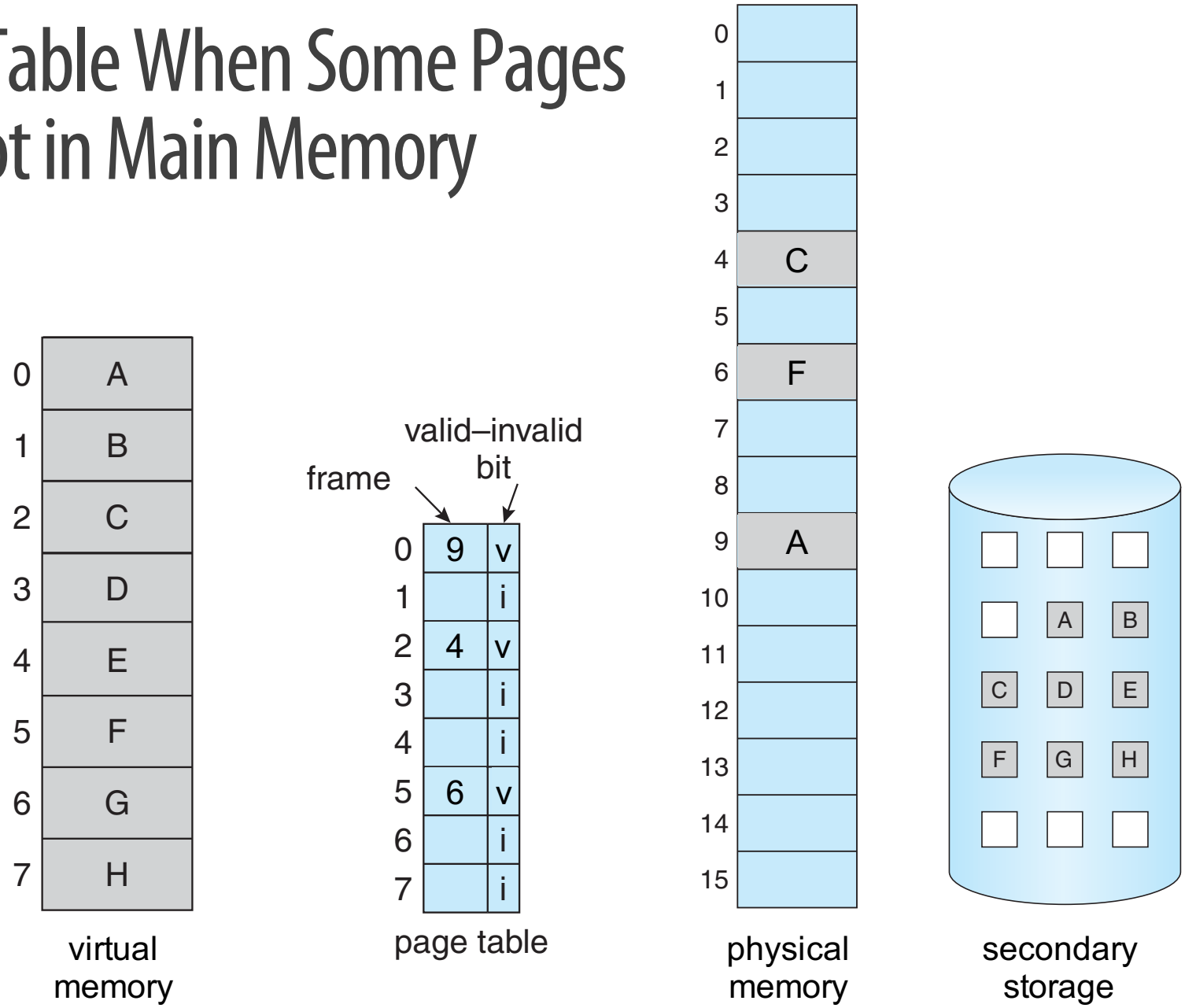
Page Table

	Frame #	Valid-invalid bit
0		v
1		v
2		i
3		v
4		i
…		…
		i
		i

Page Table When Some Pages Are Not in Main Memory



Page Table When Some Pages Are Not in Main Memory



Page Fault

- A reference to an address in a page which is not in memory will generate a **page fault** trap to the OS.
- The OS checks an internal table to decide
  - If the reference is invalid, abort the process.
  - If the page is legal but not in memory
    1. Find a free frame
    2. Swap page into frame
    3. Modify Page Table and internal table to indicate that the page is now in memory
    4. Restart the instruction that caused the page fault

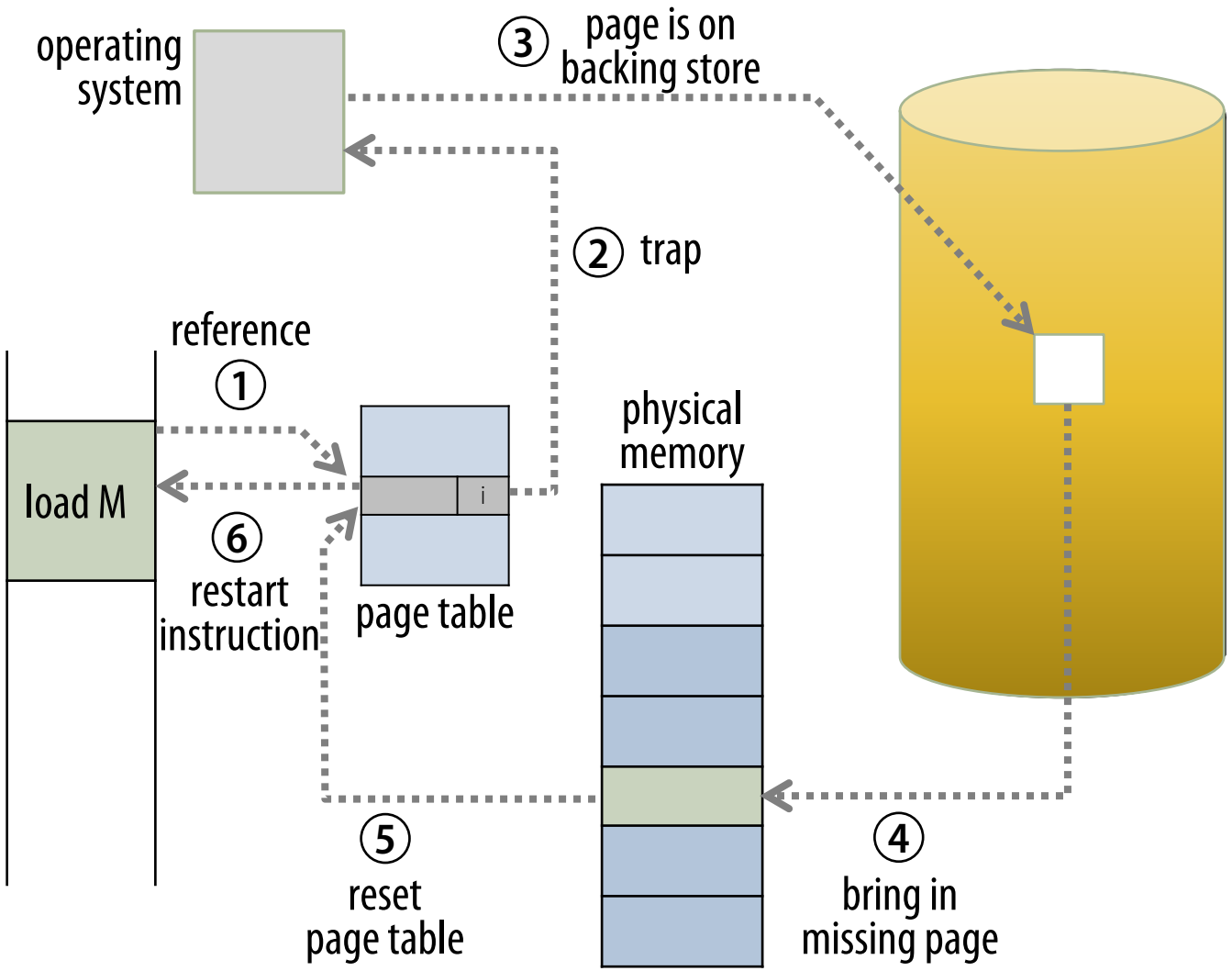
The Page Fault

- The act of accessing a page that is not in physical memory in usually called a **page fault**.
- Upon a page fault, a particular piece of the OS code – a **page-fault handler** – runs and must service the page fault.
  - If a page is not present and has been swapped to disk, the page fault handler must swap it back into memory in order to service the page fault.

What If Memory Is Full ?

- The OS would like to swap out pages to make room for the new pages it needs to bring in.
- The criteria used to pick a page to kick out, or replace, is known as the **page-replacement policy**.
  - Kicking out the wrong page can exact a great cost on program performance.
  - Making the wrong decision can cause a program to run at disk-like speeds instead of memory-like speeds.
  - In current technology, that means a program could run 10,000 or 100,000 times slower.

Page Fault Control Flow



Page Fault Control Flow – Hardware

```
1    VPN = (VirtualAddress & VPN_MASK) >> SHIFT;
2    (Success, TlbEntry) = TLB_Lookup(VPN);
3    if (Success == True) // TLB Hit
4        if (CanAccess(TlbEntry.ProtectBits) == True) {
5            Offset = VirtualAddress & OFFSET_MASK;
6            PhysAddr = (TlbEntry.PFN << SHIFT) | Offset;
7            Register = AccessMemory(PhysAddr);
8        } else {
9            RaiseException(PROTECTION_FAULT);
10       }
11   else { // TLB Miss
```

Page Fault Control Flow – Hardware

```
11  else { // TLB Miss
12      PTEAddr = PTBR + (VPN * sizeof(PTE));
13      PTE = AccessMemory(PTEAddr);
14      if (PTE.Valid == False) {
15          RaiseException(SEGMENTATION_FAULT);
16      } else {
17          if (CanAccess(PTE.ProtectBits) == False) {
18              RaiseException(PROTECTION_FAULT);
19          } else if (PTE.Present == True) {
20              // assuming hardware-managed TLB
21              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits);
22              RetryInstruction();
23          } else if (PTE.Present == False) {
24              RaiseException(PAGE_FAULT);
25          }
26      }
27  }
```

Page Fault Control Flow – Software

- The OS must find a physical frame for the soon-be-swapped-in page to reside within.
- If there is no such page, wait for the replacement algorithm to run and kick some pages out of memory, before reading from the disk, updating the PTE and retrying the instruction that raised the page fault.

```
1    PFN = FindFreePhysicalPage();
2    if (PFN == -1)                  // no free page found
3        PFN = EvictPage();              // run replacement algorithm
4    DiskRead(PTE.DiskAddr, pfn);    // sleep (waiting for I/O)
5    PTE.present = True;             // update page table with present
6    PTE.PFN = PFN;                  // bit and translation (PFN)
7    RetryInstruction();             // retry instruction
```

Demand Paging Requirements

- A crucial requirement is the ability to restart any instruction after a page fault has been serviced.
- In some situations this is easier said than done:
  - Block move

  - Auto increment/decrement location

Performance of Demand Paging (1/2)

- Demand paging can significantly affect the performance of a computer system.
- Let's compute the **Effective Access Time (EAT)** for a demand-paged memory.
- Let p $(0 \leq p \leq 1)$ be the **Page Fault Rate**, i.e. the probability of a page fault in any memory access.
  - If $p = 0$, no page faults
  - If $p = 1$, every reference is a fault

Performance of Demand Paging (2/2)

- $EAT = (1 - p) \times memory\ access\ time$

$$+ p \times (\ page\ fault\ interrupt\ service\ overhead + swap\ page\ out\ overhead + swap\ page\ in\ overhead + process\ restart\ overhead\ )$$

Page fault service time

Example of Demand Paging Overhead

- Memory access time $= 200ns$
- Average page-fault service time $= 8ms = 8{\times}10^6 ns$
- $EAT = (1 - p){\times}200 + p{\times}(8.000.000)$
  $= 200 + 7.999.800{\times}p$
- If one out of 1.000 accesses causes a page fault
  - $EAT = 8.200ns = 8{,}2\mu s$
  - This means a slowdown by a factor of 40!!
- To limit degradation to 10%, less than about one in 400.000 accesses can lead to page faults.

Can the OS reach such a demanding goal?