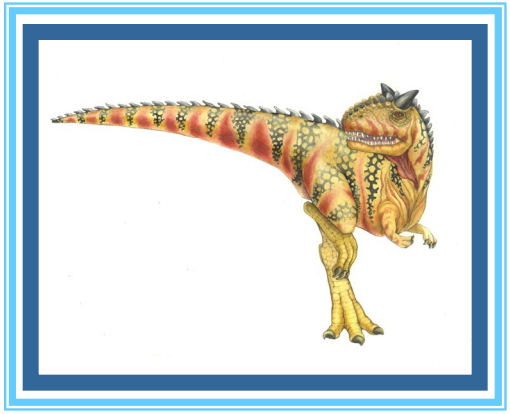
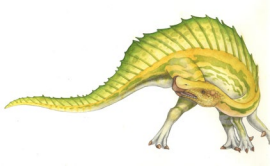
Chapter 10: Virtual Memory



Introduction

- **Virtual Memory (VM)** is a technique that allows the execution of **processes** that are not entirely in **main memory**.
- One significant advantage of this VM scheme is that programs or applications can be larger than physical memory (main memory).
- **Virtual memory** abstracts **main memory** into a vast, uniform array of storage, separating ***logical memory*** as viewed by the programmer from ***physical memory***.
  - VM frees programmers from the concerns of memory-storage limitations.



Introduction

- **Virtual memory** also allows processes to share files and libraries, and to implement shared memory.
  - It provides an efficient mechanism for process creation.
- **Virtual memory** is not easy to implement; it may substantially decrease performance if it is used carelessly.
- The **memory-management algorithms** outlined in Chapter 9 require that *the instructions being executed must be in physical (main) memory*.
- One of the approaches to meet this is to place the **entire logical address** space in **physical memory**.
- **Dynamic linking** can help to ease this restriction.
  - **Dynamic linking** is a process where an OS loads and binds shared libraries (DLLs, .so files) to an **executable at runtime**.

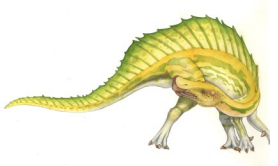
Background

- The requirement that instructions must be in **physical memory** to be executed seems both necessary, but it is also crucial when *the size of a program exceeds the size of physical memory*.
- For instance, consider the following:
  - ***Arrays***, ***lists***, and ***tables*** are often allocated more memory than they need.
  - An array may be declared **100 by 100** elements, even though it is seldom larger than **10 by 10** elements.
- The ability to execute a program that is only partially allocated in memory would confer many benefits:
  - *A program would no longer be constrained by the amount of physical memory that is available*.

Background

- *Users would be able to write programs that are significantly larger than the available memory space for a process.*
- *Because each program could take less physical memory, more programs could be run at the same time, with an increase in **CPU utilization** and **throughput**.*
- *Less I/O operations would be needed to load or swap portions of programs into memory, with the logical memory, so each program would run faster.*

- Thus, running a program that is not entirely in memory would benefit both the system and its users.



Background

- **Virtual memory** involves the separation of **logical memory** as perceived from **physical memory**.
- This separation allows an extremely **large virtual memory** to be provided for programmers when only a **smaller physical memory** is available (**Figure 10.1**).
- Through the **Virtual memory,** a programmer no longer needs to worry about the size of the **main memory space**.

Virtual Memory That is Larger Than Physical Memory

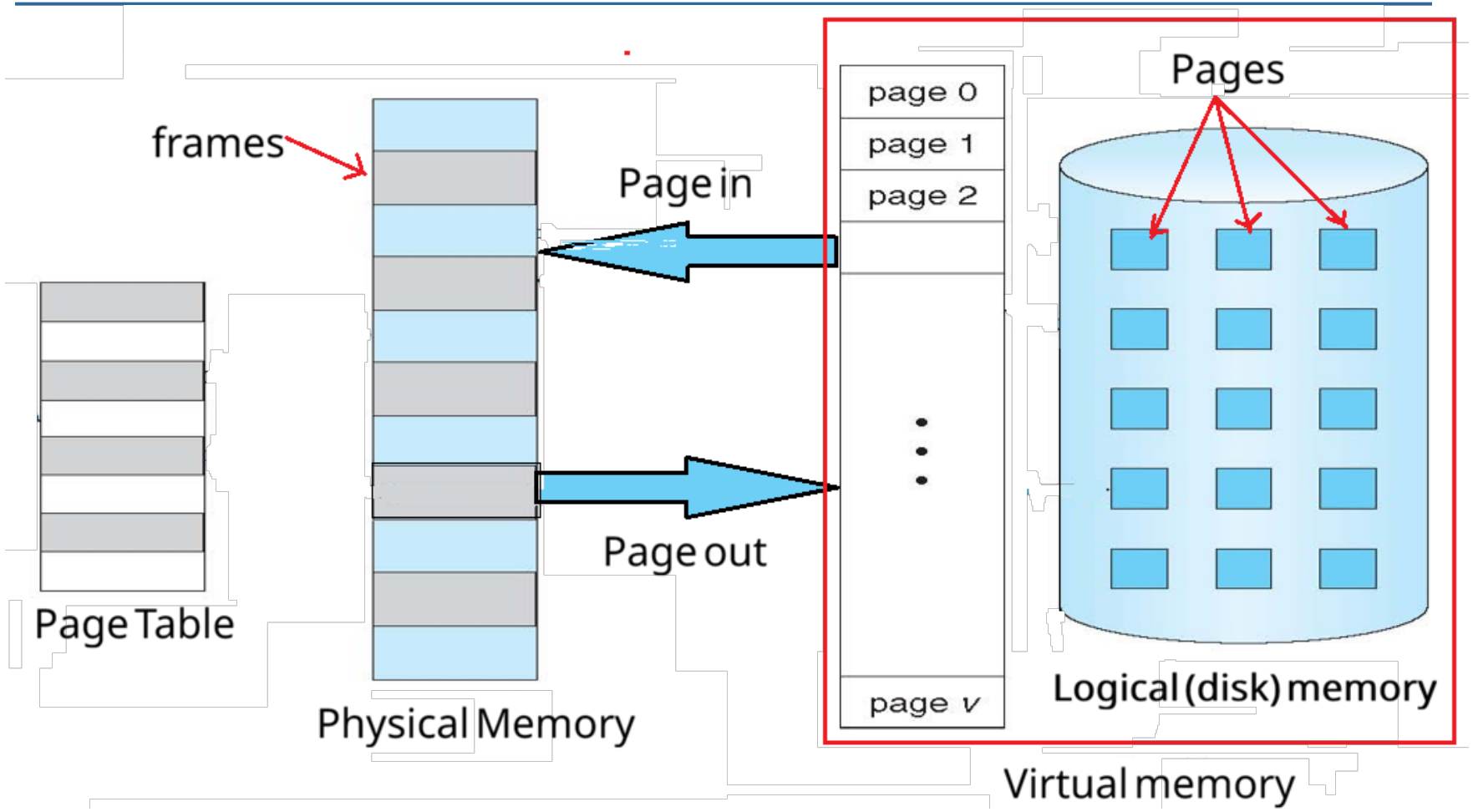


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

Background

- **Figure 10.2 shows** that the **heap** grows upward in memory as it is used for **dynamic memory allocation**.
  - Similarly, we allow for the **stack** to grow downward in memory through successive **function calls**.
  - The large blank space (or hole) between the **heap** and the **stack** is part of the **virtual address space** but will require actual physical pages only if the heap or stack grows.
- **Virtual address spaces** that include **holes** are known as ***sparse address spaces***.
  - Using a ***sparse address space*** is beneficial because **holes** can be filled as **stack or heap segments** grow, or when **dynamically linked libraries (or other shared objects) are dynamically linked during program execution (pages shared to this hole, see Figure 10.3).**

Background

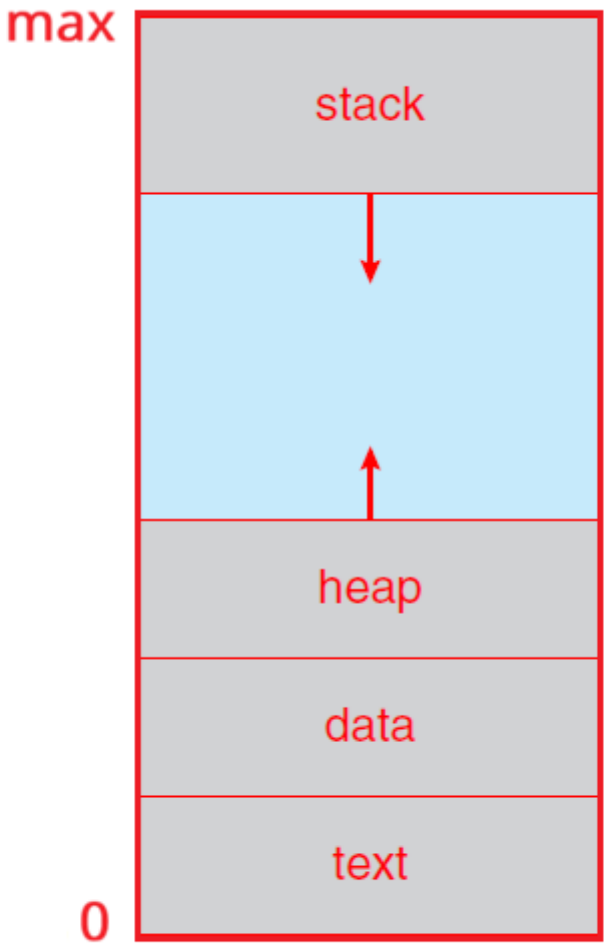


Figure 10.2 Virtual address space of a process in memory.

Background

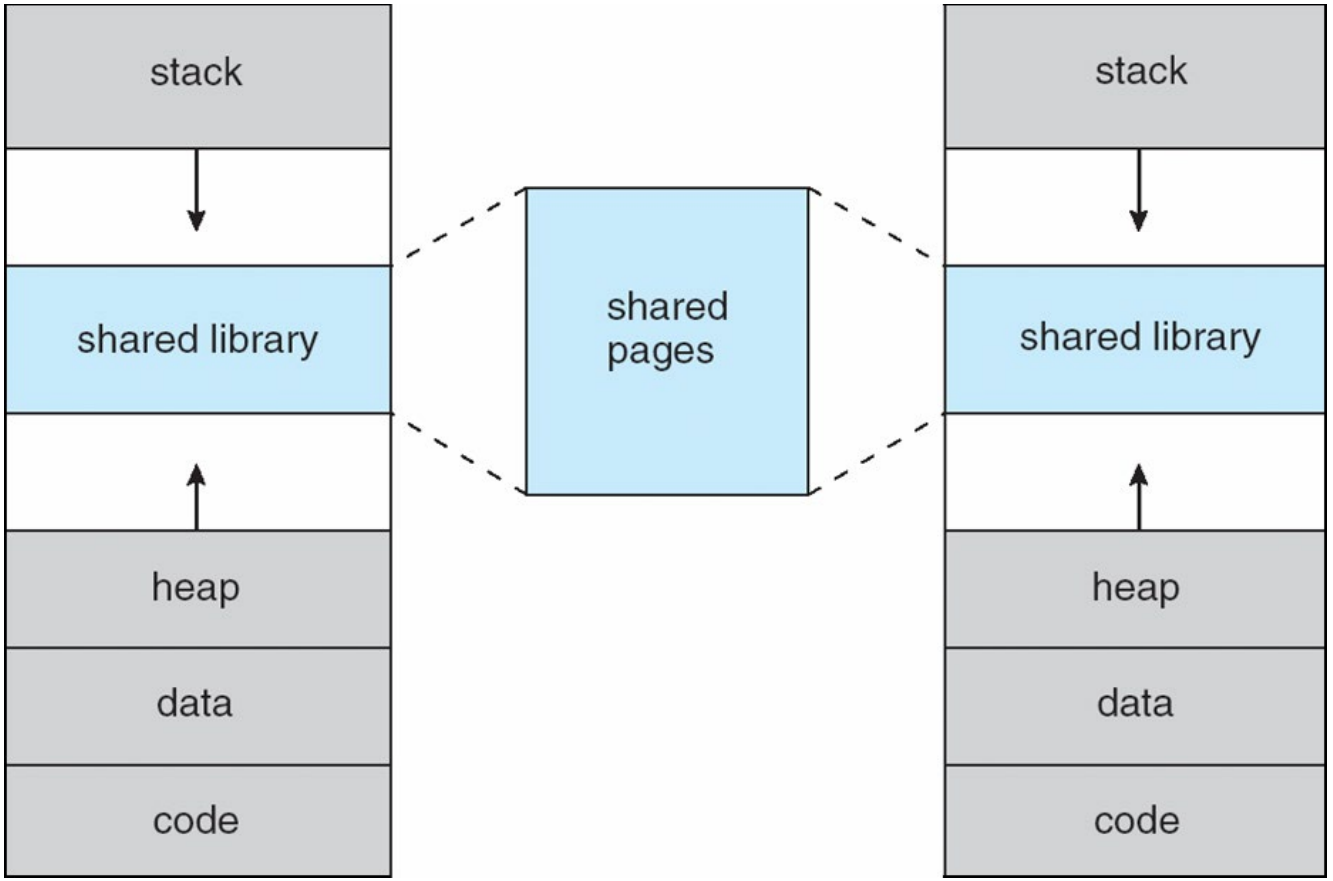


Figure 10.3 Shared pages to fill holes during dynamic allocation.

Background

- The **virtual address space of a process** refers to the **logical (or virtual)** view of how a process is stored in memory as **pages**.
- Recall from Chapter 9 that **physical memory is organized in page frames and that the pages of a process assigned to frames may not be contiguous**.
- **Virtual address space** $\Rightarrow$ **logical view** of how an executable file of a process is stored in the logical (disk) memory space.
- **Physical memory (Main Memory) is** organized in **page frames**
- It is up to the **memory management unit (MMU) to map pages to physical frames in memory.**

Background

- **Virtual memory** access implemented via:
  - **Demand paging**
  - **Demand segmentation**

Demand Paging

- ***How might an executable program be loaded from secondary storage into memory?***
- **One option** is to load the entire program (the full .exe file) into physical memory at program execution time.
  - However, to execute this program, the entire **.exe file** does not need to be in main memory.
- **An alternative strategy** is to load pages only as they are needed. This technique, known as **demand paging**, is commonly used in virtual memory systems.
  - With **demand-paged virtual memory**, pages are loaded only when they are ***demanded*** during program execution.
  - *Pages that are never accessed are thus never loaded into physical memory*.

Demand Paging

- The general concept behind **demand paging** is to load a page in memory only when it is needed.
- As a result, while a process is executing, some pages will be in memory, and some will be in logical (disk) memory.
- The **valid–invalid bit** of the corresponding **page table** can be used for determining the **currently loaded pages** in the memory frames.
- If the bit is set to "**invalid**," the page either is **not valid** (that is, not in the logical address space of the process) or is valid but is currently in secondary storage.
- This situation is depicted in **Figure 10.4**.

Page Table When Some Pages Are Not in Main Memory

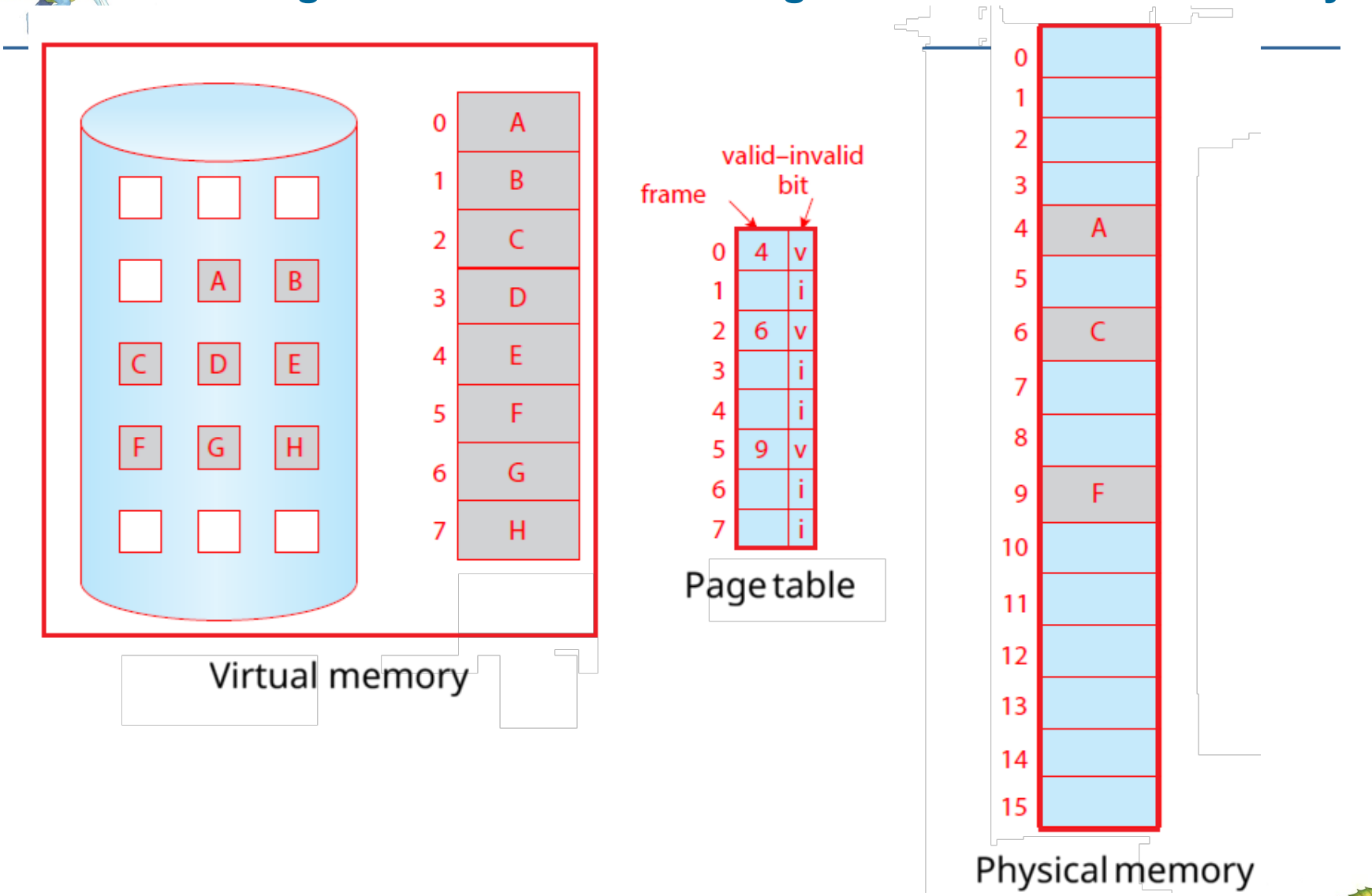
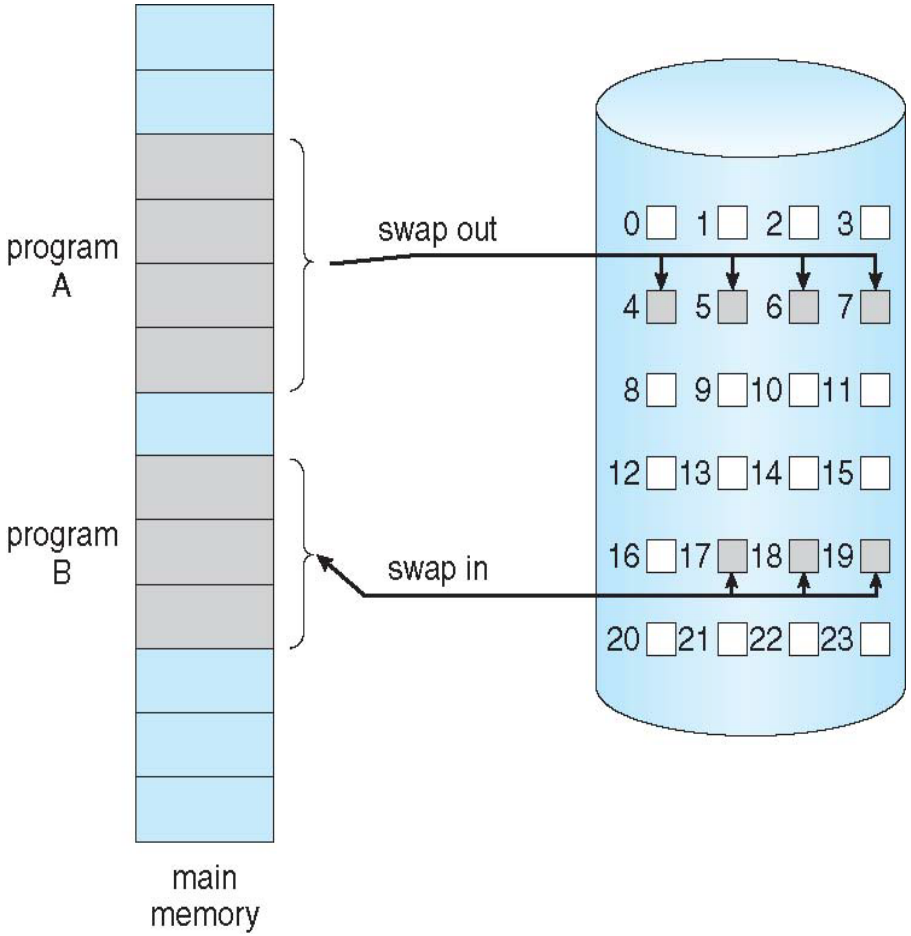


Figure 10.4 Page table when some pages are not in memory frames.

Demand Paging

- **Bring a page into memory only when it is needed**
  - Less memory needed, Faster response, and More process execution
- **Page is needed ⇒ as a reference to it**
  - **Not-in-memory (invalid) ⇒ bring to memory frames from VM**



Demand Paging

- **What happens if the page that is needed for execution is not brought into the memory frames?**
- Access to a page marked **invalid** causes a **page fault**.
- During the **page fault**, the paging hardware, in translating the virtual address through the page table, will notice that **the invalid bit** is set, causing a **trap** to the OS.
- This **trap** is the result of the operating system's failure to bring the desired page into memory.
- The **procedure for handling this page fault** is called the **page fault-handler routine (PFHR)** of the OS, and is shown in **Figure 10.5.**

Steps in Page Fault-Handler Routine

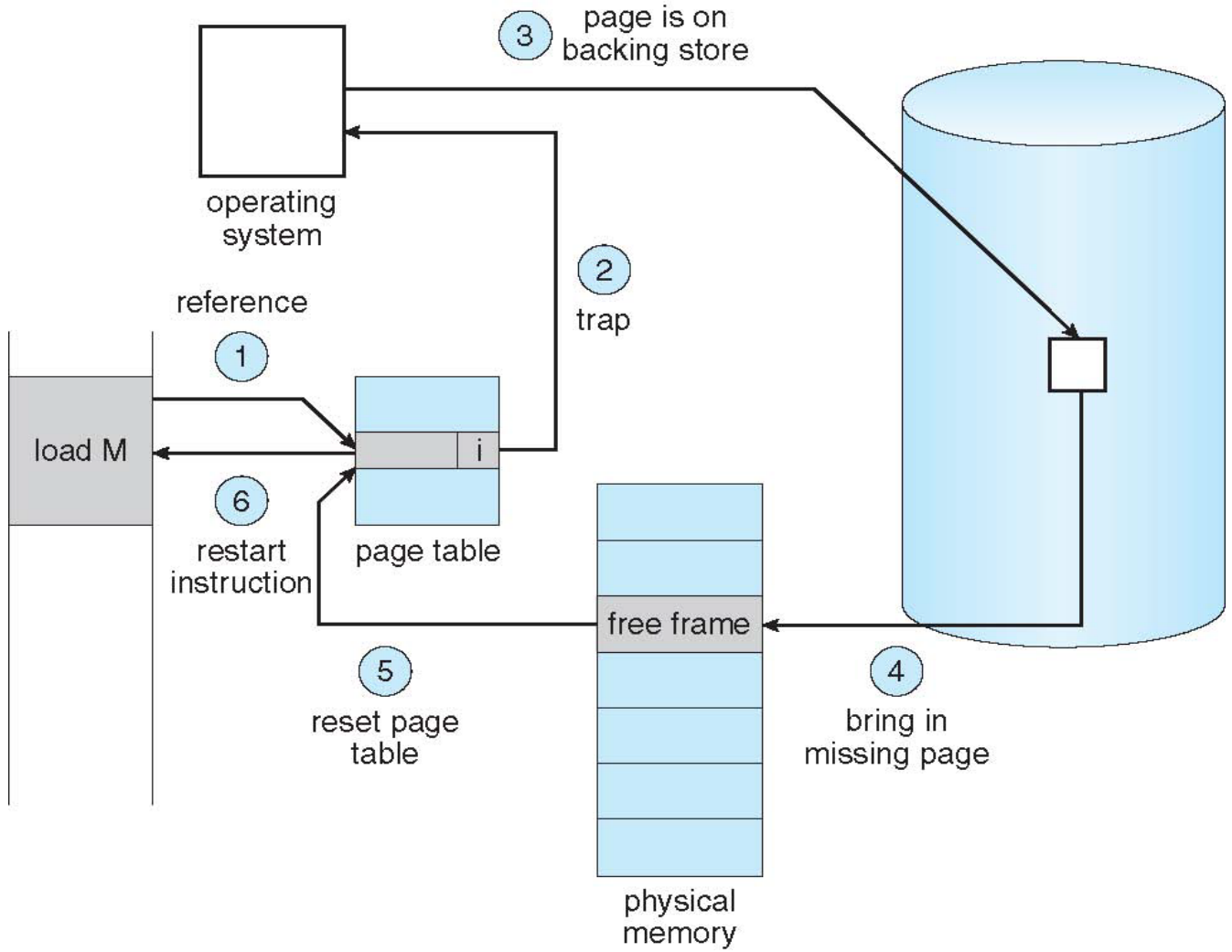


Figure 10.5 Steps in handling a page fault.

Steps in Handling a Page Fault

- **The procedure for handling this page fault is straightforward (Figure 9.6):**
  1. First, check the **page table** for this process to determine whether the reference was a **valid or an invalid** memory access.
  2. If the reference was **invalid**, we terminate the process. If it was **valid** but has not yet brought in that page from disk into **MM**.
  3. Find a **free frame** in memory for the new page *(if no free frame space, then evict one of the pages from the frames – called **page replacement***).
  4. Then allocate the missed pages into the selected frame by swapping the page from logical (disk) memory.
  5. After the page is allocated into the frame, then update the **page table** to indicate that the new page is now in memory frame.
  6. Then restart the execution by accessing the page using the virtual address. This is the OS's **PFHR**.

Performance of Demand Paging

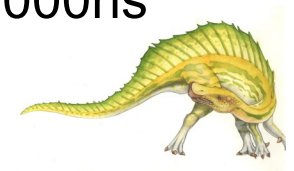
- How to compute the ***effective memory access time*** (***EMAT***) for a demand-paged memory?
  - For most computer systems, the ***memory-access time*** ranges from **10 to 200 nanoseconds**.
- ***If there is no page fault, then the EMAT is equal to the memory access time*** (***MAT***).
- Let ***p*** be the **probability of a page fault** (**0 ≤ *p* ≤ 1**). We would expect ***p*** to be close to **zero**—that is, we would expect to have only a few page faults. The **EMAT** is then:

$$\textbf{EMAT} = (1 - p) \times \textit{MAT} + (p \times \textit{page fault time})$$

- With an average page-fault service time of **8 ms** and a memory access time of **200 ns**, the ***EMAT*** in ns is $(1 - p) \times (200) + p\ (8\ \text{ms})$

$$= (1 - p) \times 200 + p \times 8{,}000{,}000 \quad \{1\text{ms} = 1000{,}000\text{ns}$$

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



What Happens if There is no Free Frame?

- **Page replacement** – find some page in memory frame, but not really in use, then swap it out
  - **Algorithm** – terminate? swap out? replace the page?
  - **Performance** – want an algorithm which will result in minimum number of **page faults**
- *Same page may be brought into memory several times*

Page Replacement

- Consider that the system memory is not used only for holding all of the pages from the user program but also it act as buffers for I/O operation.
- This can increase the strain on memory-placement algorithms.
- During a program execution, deciding how much **memory to allocate to I/O** and **how much to program pages** is a significant challenge of the **OS**.
- Some systems allocate a **fixed percentage** of memory for **I/O buffers**, whereas others allocate all system memory for both **user processes** and **I/O operations**.

Page Replacement

- **Over-allocation** of memory manifests itself as follows:
  - Assume that a user process causes a **page fault**. The OS determines where the **desired page** resides on **disk**.
  - In case, there are ***no free frames*** on the MM for the **missing page**, then OS should free one of the memory frames for the new page (**Figure 10.9**).
- The **OS has several options** at this point. **It could terminate one of the user processes**:
  - However, **demand paging** is the OS's attempt to improve the computer system's *utilization* and *throughput*.
  - **paging should be logically transparent to the user**.
- **So this option is not the best choice.** The OS could instead **swap out** a process, **freeing all its frames** and reducing multiprogramming.

Need For Page Replacement

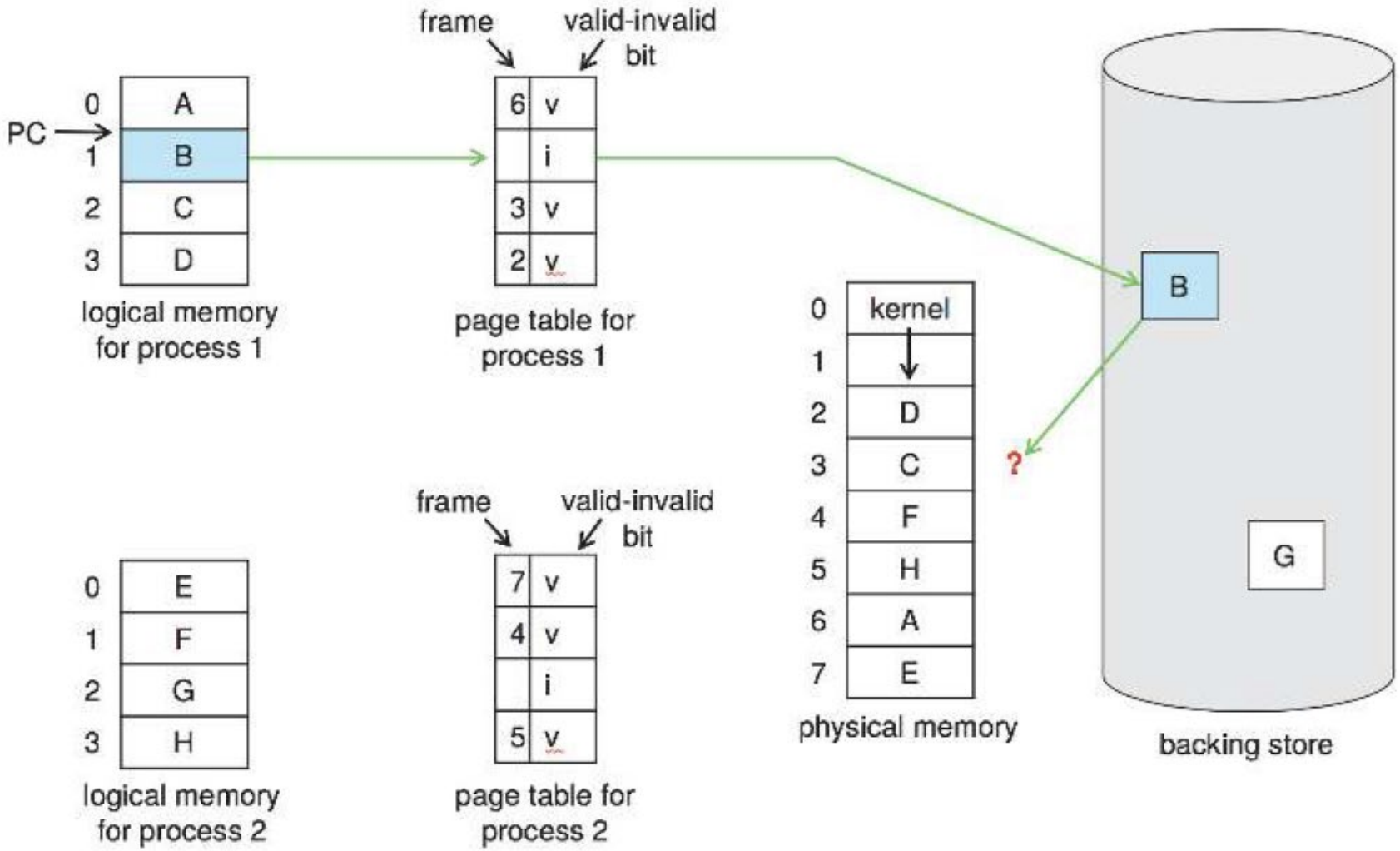


Figure 10.9 Need for page replacement.

Basic Page Replacement

- **Page replacement from the MM (main memory) takes the following steps**:
  - If **no frame is free**, find one of the frames that is **not currently being used for a long time** and free it.
  - Free a frame by moving it to the disk space and updating the page table (and all other tables) to indicate that the page is no longer in memory (**Figure 10.10**).
  - Then use the freed frame to hold the missing page for which the process is needed.

Basic Page Replacement

the page-fault service routine to include page replacement (Figure 10.10):

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.
   c. Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

Page Replacement

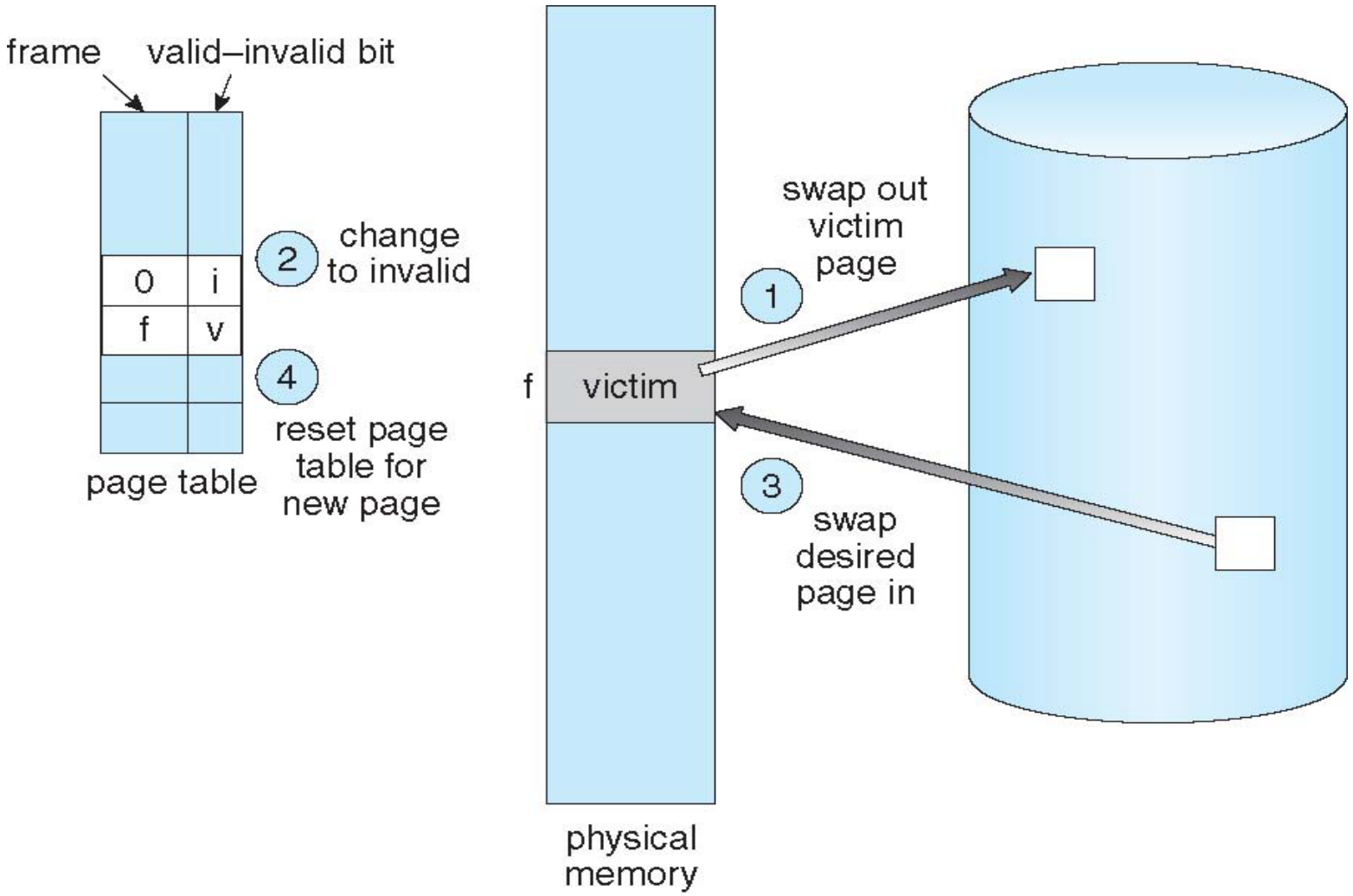
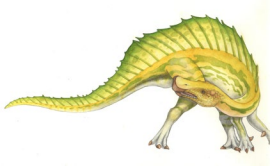


Figure 10.10 Page replacement.

Basic Page Replacement

- **Page replacement** is described as follow**:**
  1. Find the location of the desired page on the disk.
  2. Find a free frame:
     - **a**. If there is a free frame, use it.
     - **b**. If there is no free frame, use a **page-replacement algorithm** to select a **victim frame** (*the frame which is going to be replaced from a MM frame is called the* ***victim frame***).
     - **c**. Write the **victim frame** to the disk (**swap-out**); update the page table accordingly.
  3. Read the desired page into the newly freed frame
  4. Continue the user process from where the page fault occurred.



Page-replacement Algorithms

- There are many different **page-replacement** algorithms.
  - Every OS probably has its own page replacement scheme.
- **How do we select a particular replacement algorithm?**
  - In general, we want the one with the **lowest** page-fault rate.
- We **evaluate a page replacement algorithm** by running it on a **particular string** of **memory references** and computing the number of **page faults**.
  - The string of memory references is called a **reference string**.

Page-replacement Algorithms

- **Page replacement** is basic to **demand paging**.
- It completes the separation between **logical memory** (Disk) and **physical memory** (MM).
- With this mechanism, an **enormous virtual memory** can be provided for programmers on a **smaller physical memory**.
- **With no demand paging**, user addresses (*logical addresses*) are mapped into physical addresses:
  - That is, all the pages of a process still must be in physical memory!.
- **With demand paging**, the size of the logical address space (program size) is no longer constrained by physical memory size.

Page-replacement Algorithms

- For example, assume a user process has **twenty pages**, the system can execute it in **ten frames** simply by using **demand paging** and using a **replacement algorithm** to **find a free frame** whenever necessary:
  - If a **page** that has been **modified** (**updated**) is to be replaced, its contents are copied to the disk.
  - A **later reference** to that **modified page** will cause **a page fault** (because the modified page is moved to disk).
    - At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

Page-replacement Algorithms

- There are many page replacement algorithms available but this topic focuses only the three of them:
  - **Optimal**
  - **Least recently used (LRU)**
  - **First-in-first-out (FIFO)**

Page-replacement Algorithm - Optimal

- The **optimal** page replacement policy ***selects the page for which the time to the next reference is the longest*** (*if any resident page in the frame is not going to be used next or a resident page is very far for the next reference, then replace that from frame with the newly arrived page*).
- The **Figure 8.15** gives an example of the optimal policy. The example assumes a fixed frame allocation for this process of **three frames**.
- The execution of the process requires reference to **five distinct pages**.
- The page **reference string** is 2 3 2 1 5 2 4 5 3 2 5 2 (means that the first page referenced is 2, the second page is referenced is 3, the third page referenced is 2, and so on).
- The optimal policy produces **three page faults** after the frame allocation has been filled (see **Figure 8.15**).
  - *This policy is **very complex to implement**, because it would require the OS to have perfect knowledge of future page addresses*.

Page-replacement Algorithm -Optimal

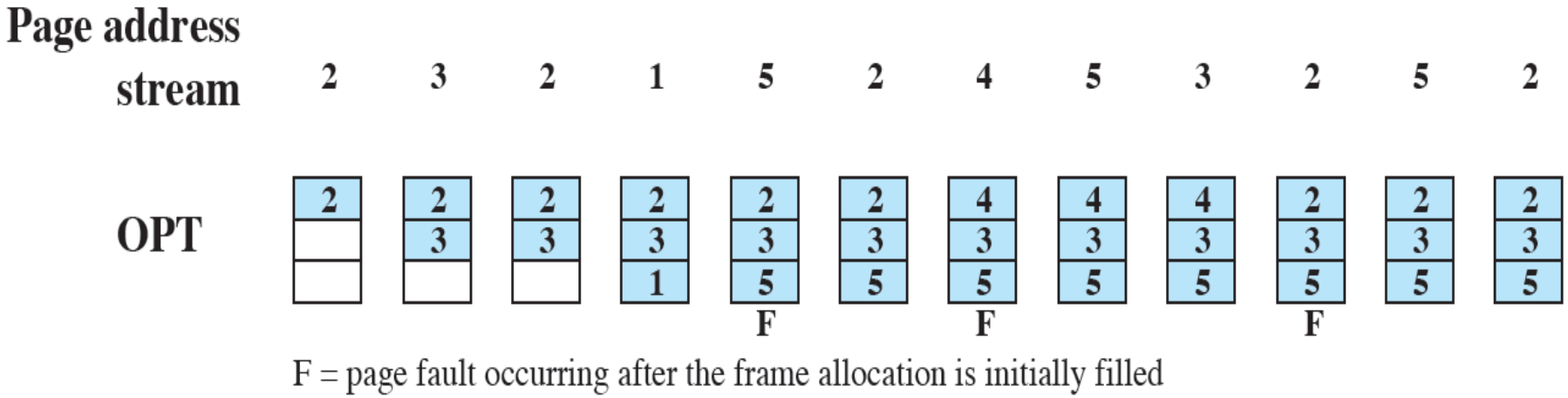


Figure 8.15 gives an example of the optimal policy

Page-replacement Algorithm -LRU

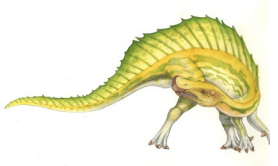
- The **least recently used (LRU) policy** *replaces the page in memory that has not been referenced for longest time*.
- The LRU policy does nearly as well as the optimal policy.
- To tag each page with the time of its **last reference**; this would have to be done at each memory reference, both instruction and data.
- ***The problem with this approach is the complexity in implementation.***
- Even if, the HW would support such scheme, the overhead would be tremendous.
- **Figure 8.15** shows an example of the LRU policy, using the same page **reference string** as for the optimal policy example. In this case, there are **four page faults**.

Page-replacement Algorithm -LRU

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		

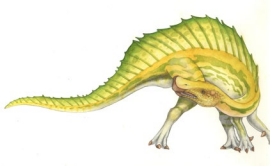
F = page fault occurring after the frame allocation is initially filled

Figure 8.15 shows an example of the LRU policy



Page-replacement Algorithm -FIFO

- The logic behind **FIFO policy** is *replacing the page that has been in memory longest.*
  - The FIFO policy treats the page frames allocated to a process as a **circular buffer** and pages are removed in **round-robin** style.
- All that is required is **a pointer** that circles through the page frames of the process.
- This is therefore one of the **simplest page replacement policies** to implement.
- A page fetched into memory a long time ago may have now fallen out of use. This reasoning will often be wrong. Pages will be repeatedly **paged-in** and **page-out** by the FIFO algorithm.
- **Figure 8.15** shows the FIFO policy results **six page faults**.
  - Note that LRU recognizes that **pages 2 and 5** are referenced more frequently than other pages, whereas FIFO does not.

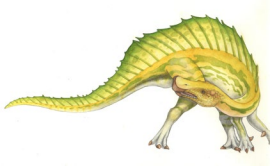


Page-replacement Algorithm -FIFO

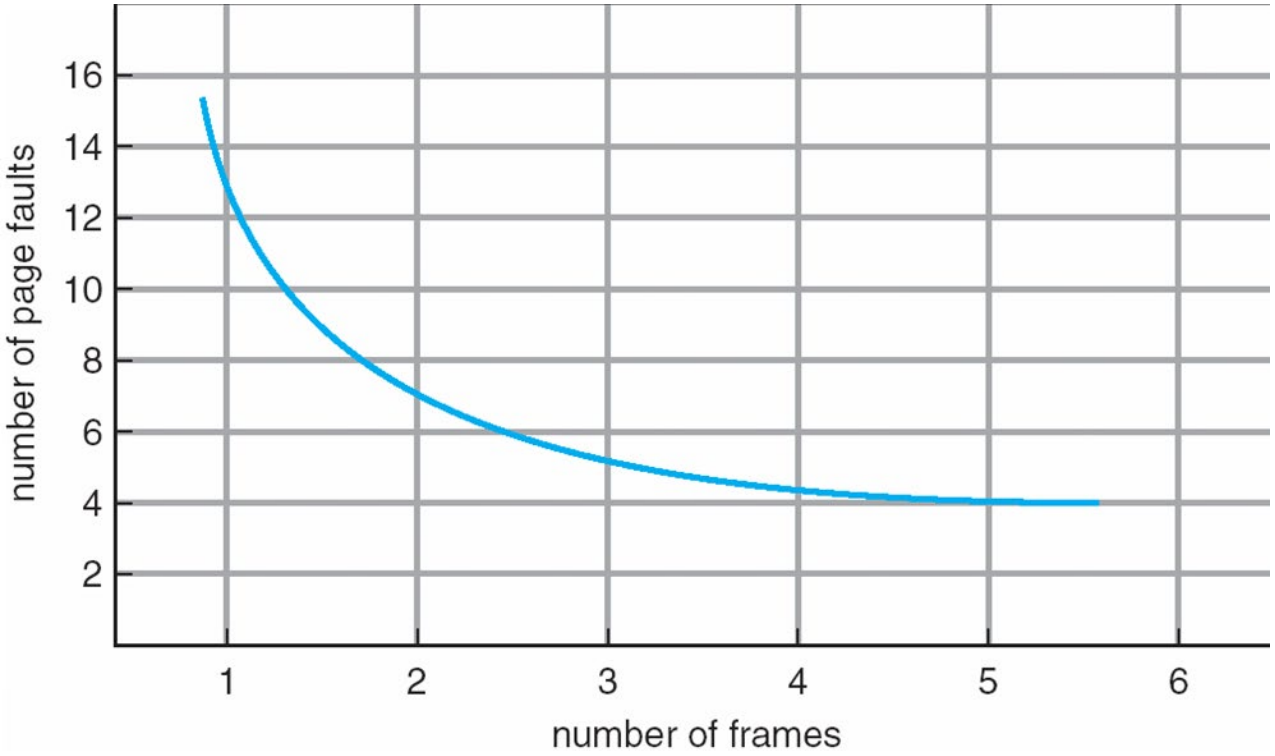
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F

F = page fault occurring after the frame allocation is initially filled

Figure 8.15 shows the FIFO policy results six page faults



Graph of Page Faults Versus The Number of Frames



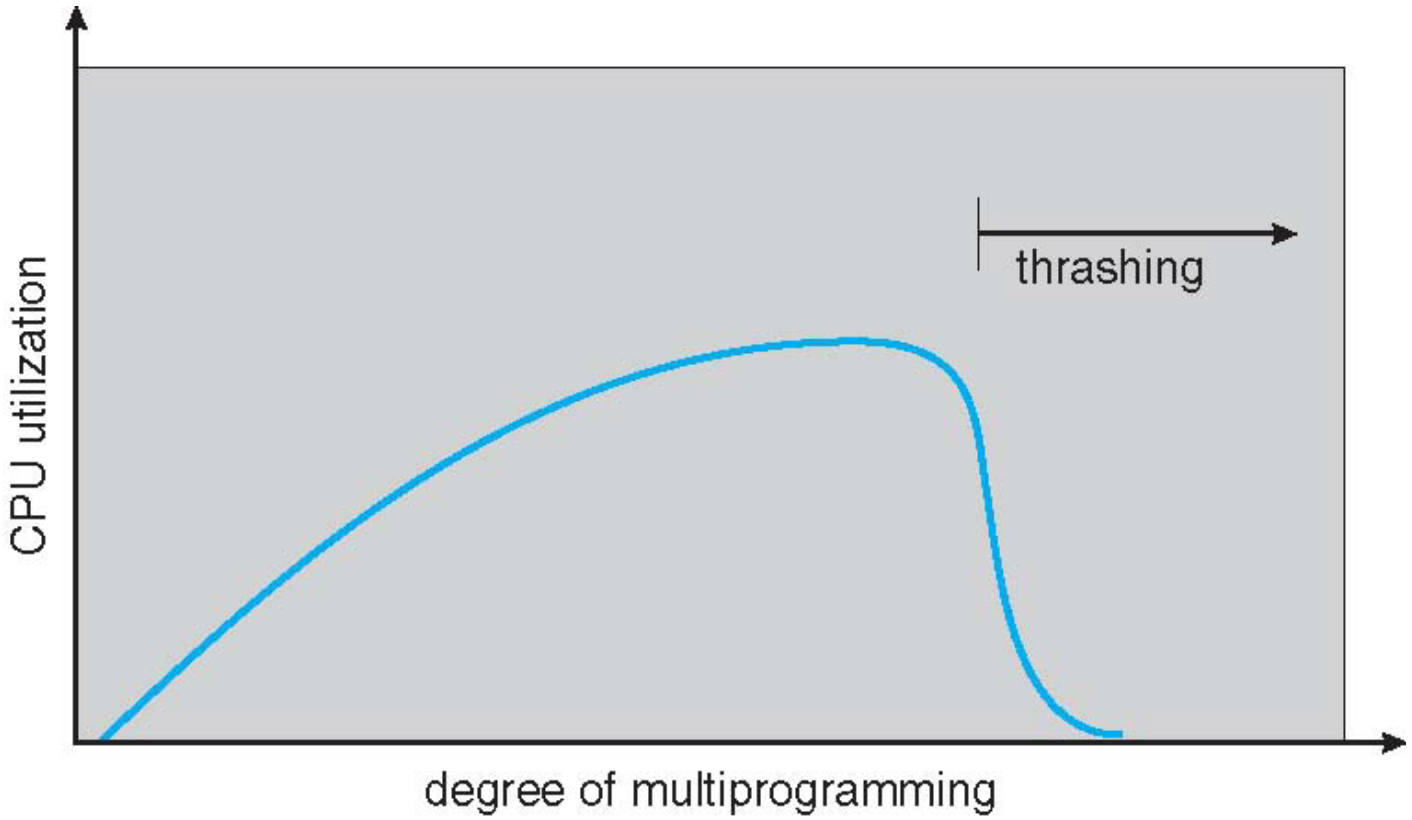
Thrashing

Thrashing

Consider what occurs if a process does not have "enough" frames—that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing. As you might expect, thrashing results in severe performance problems.

Thrashing (Cont.)



Other Issues – Page Size

- Sometimes OS designers have a choice about **page size**
  - Especially if running on custom-built CPU
- **Page size** selection must take into the following consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Page size is always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)

Other Issues – TLB Reach

- **TLB Reach** - The amount of memory accessible from the TLB
  - **TLB Reach = (TLB Size) X (Page Size)**
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- **Increase the Page Size**
  - This may lead to an **increase in fragmentation** as not all applications require a large page size
- **Provide Multiple Page Sizes**
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

Let's look at a contrived but informative example. Assume that pages are 128 words in size. Consider a C program whose function is to initialize to 0 each element of a 128-by-128 array. The following code is typical:

```
int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

Notice that the array is stored row major; that is, the array is stored `data[0][0]`, `data[0][1]`, ···, `data[0][127]`, `data[1][0]`, `data[1][1]`, ···, `data[127][127]`. For pages of 128 words, each row takes one page. Thus, the preceding code zeros one word in each page, then another word in each page, and so on. If the operating system allocates fewer than 128 frames to the entire program, then its execution will result in $128 \times 128 = 16{,}384$ page faults.

Program Structure

In contrast, suppose we change the code to

```
int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

This code zeros all the words on one page before starting the next page, reducing the number of page faults to 128.