Computer and Operating Systems (COS)

Lecture 12

Overview of the contents

- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing

Virtual Memory Management

Different memory management strategies are discussed in the previous lecture. All these strategies have the same goal:

To allow as many processes as possible to be in memory at the same time for multiprogramming.

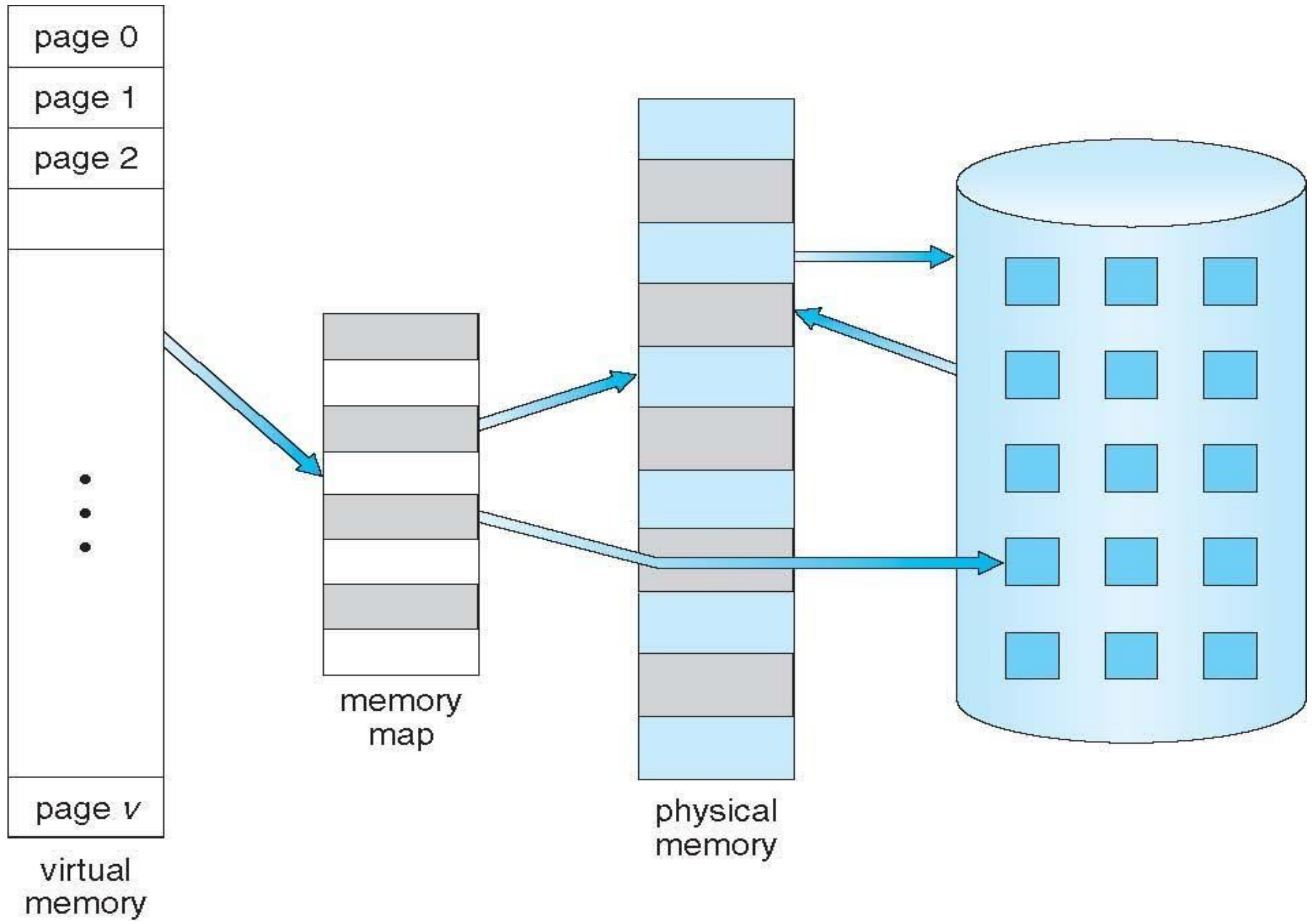
These strategies tend to require that an **entire** process is loaded in memory before it can execute.

Now we will take a closer look at a technique, **Virtual memory**, that allows the execution of processes that are **not completely** in memory.

This means that: processes can be larger than the physical memory!

This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files and libraries, and to implement shared memory.

The basic idea of Virtual Memory



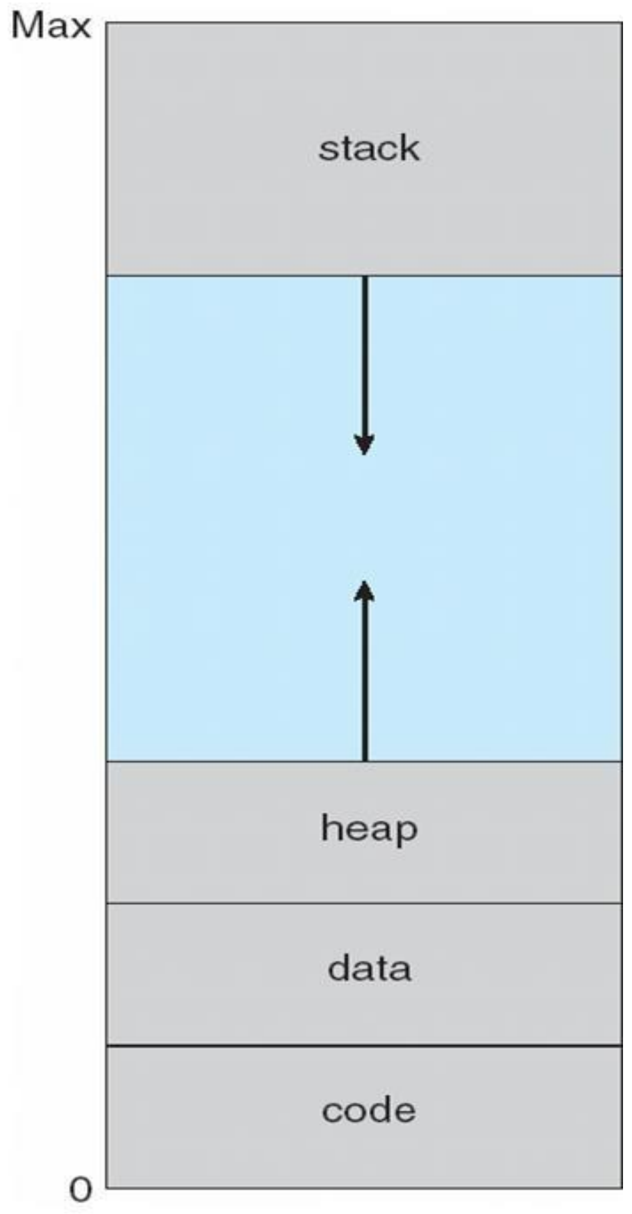
The virtual memory viewed by the programmer is contiguous, separate and larger than the physical memory.

Virtual Address Space

The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory, i.e., a process begins at address 0 and ends at address max in contiguous virtual memory.

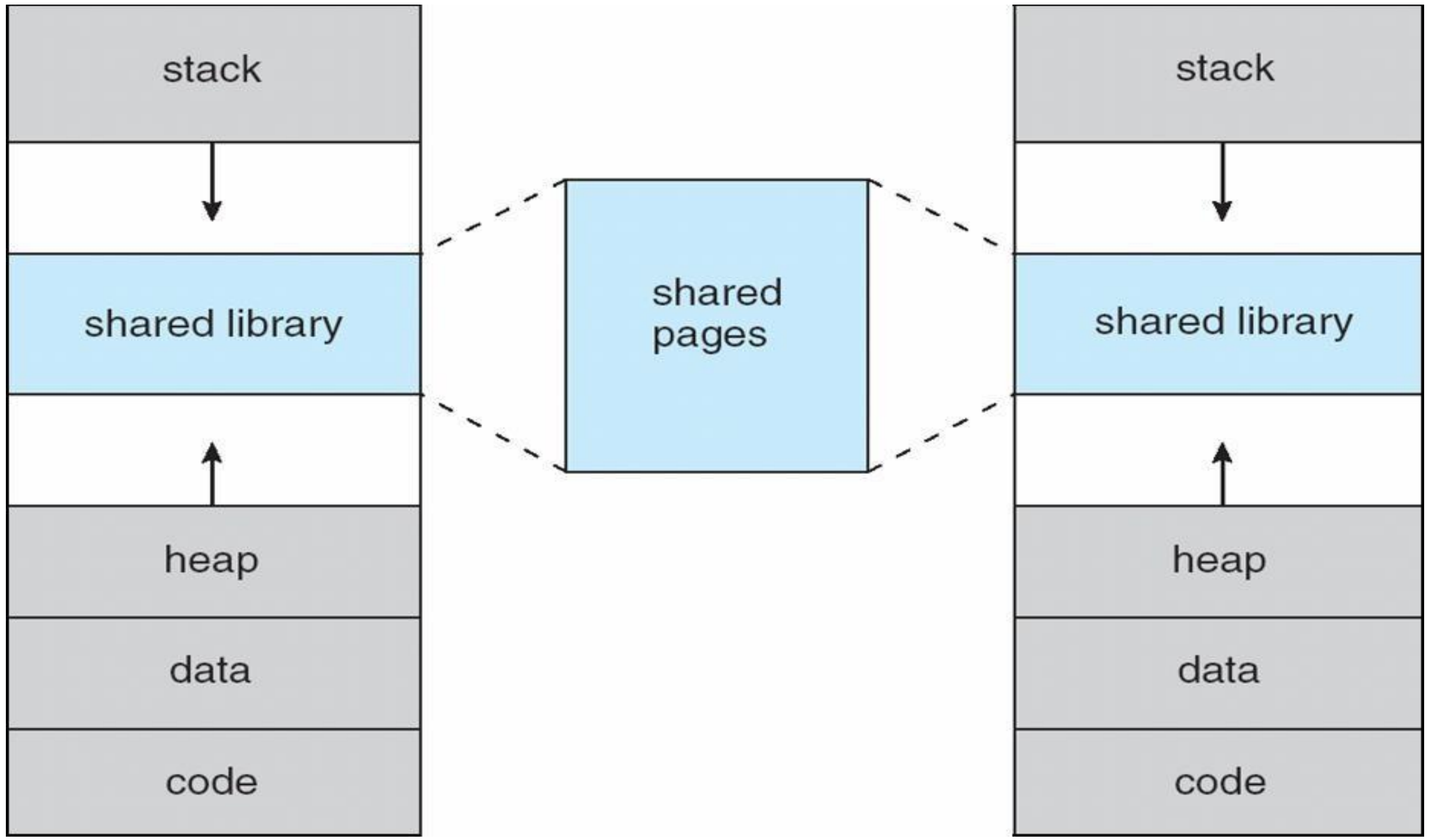
The gap between Stack and Heap is part of the virtual address space. but will require actual physical page frames only:

- When the stack or heap grow.
- Or when libraries or other shared objects are dynamically linked during the program execution.



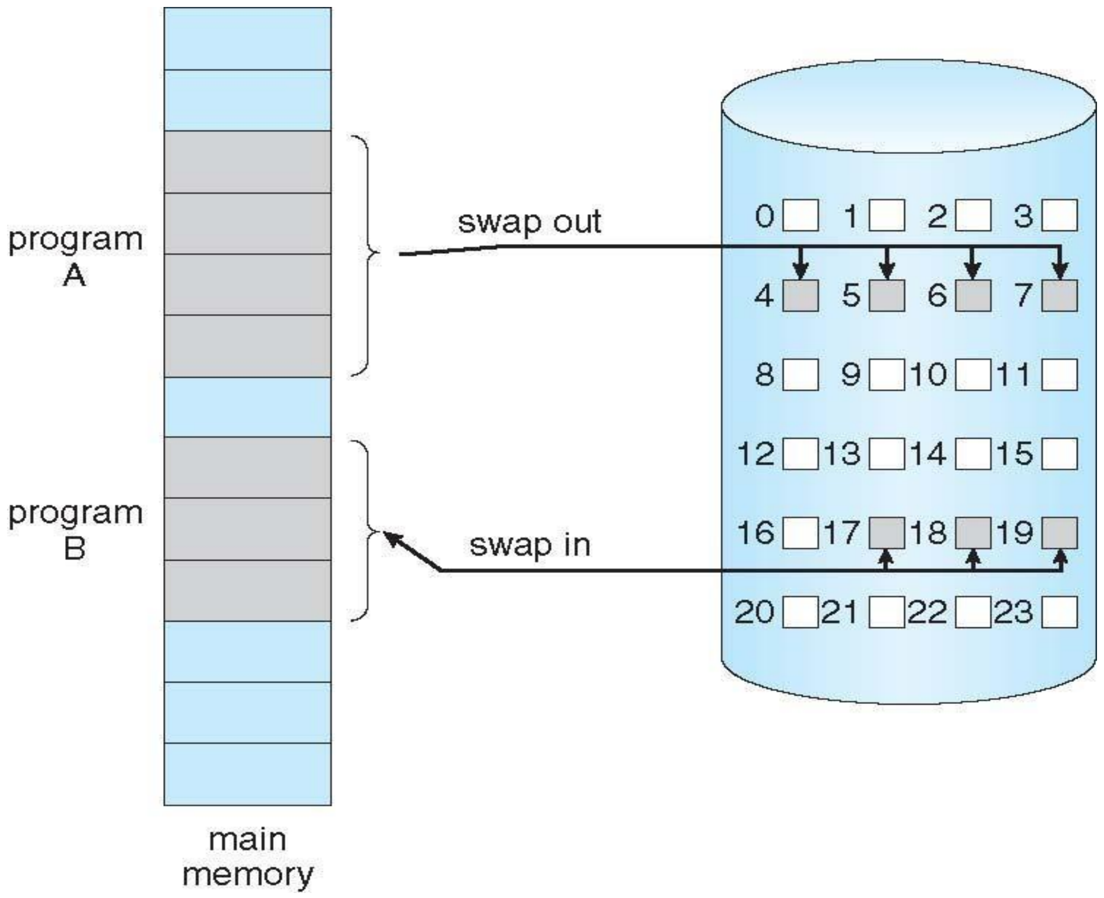
Virtual Address Space

Example: shared libraries



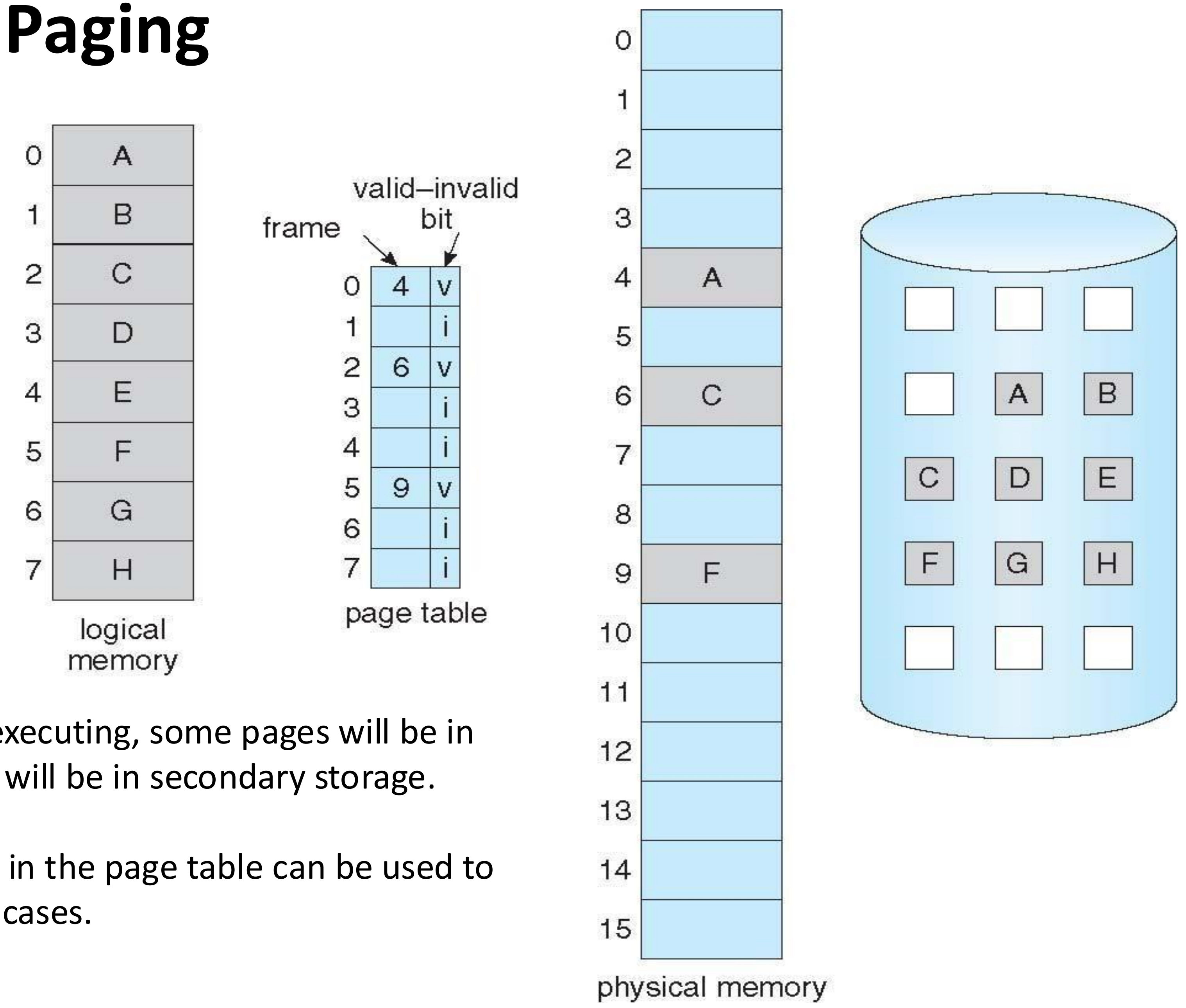
Demand Paging

Similar to the swapping with paging technique introduced before. Here is the process on the disk's swap space. But unlike the swapping technique, the individual pages of the process are loaded into memory when they are needed.



Demand Paging

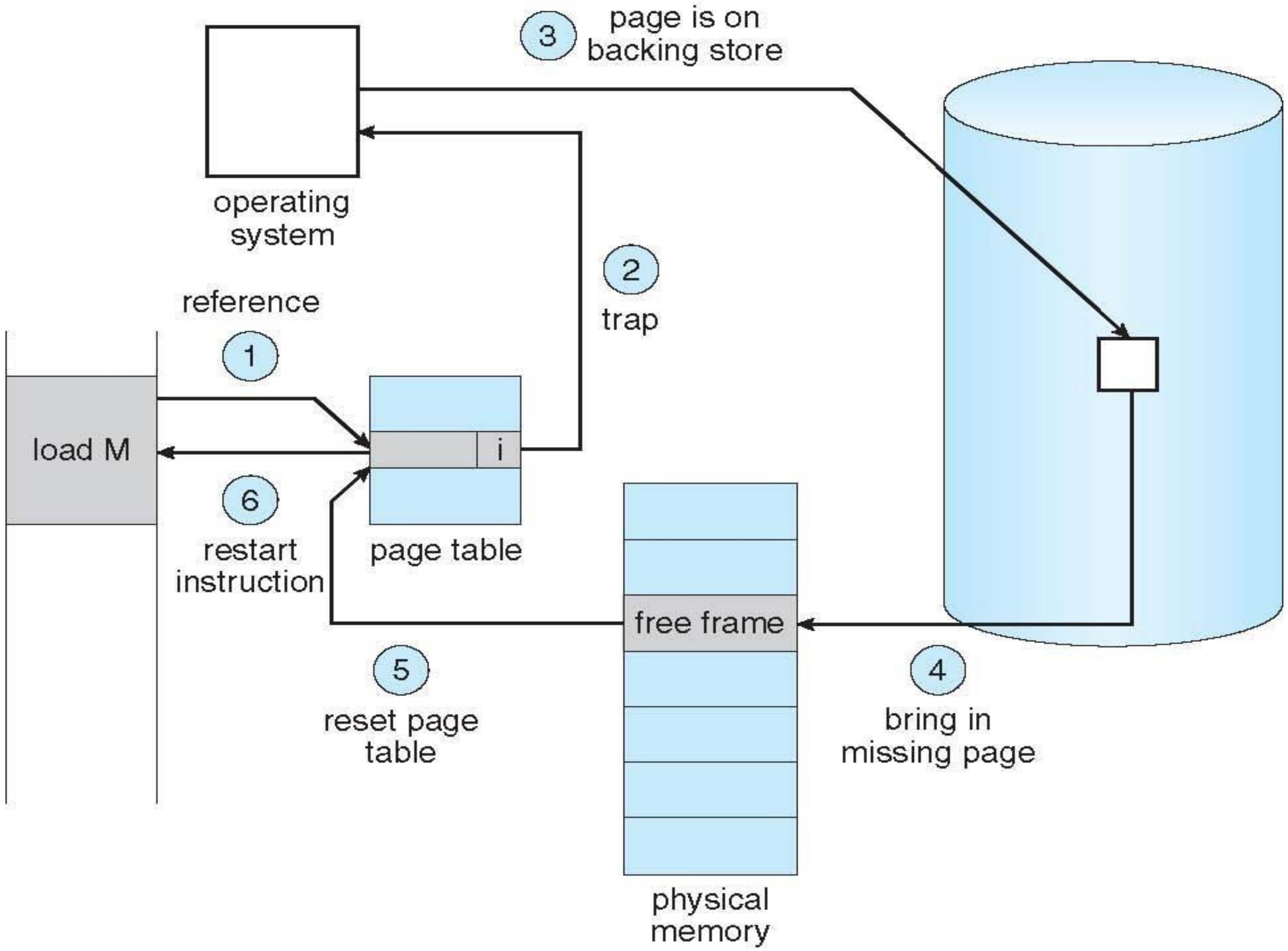
Basic idea



When a process is executing, some pages will be in memory, and some will be in secondary storage.

The valid-invalid bit in the page table can be used to distinguish the two cases.

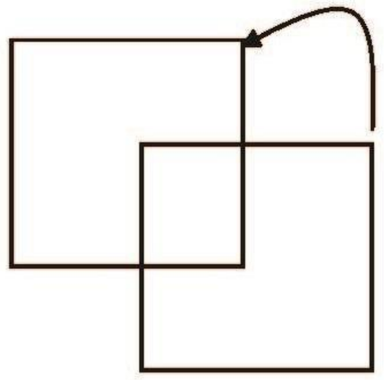
Demand Paging – Page fault handling



Process accessing to a page marked invalid will causes a **page fault.**

Restarting Instruction

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible.



In most cases, this requirement is easy to meet. The major difficulty arises when one instruction may modify several different locations, e.g., Block move.

- Solution 1: the microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen at this step, before anything is modified.
- Solution 2: uses temporary registers to hold the values of overwritten locations. If there is a page fault, all the old values are written back into memory before the trap occurs.

Demand Paging – Page fault handling

Performance

Efficient Access Time (EAT)

$$EAT = (1 - p) \times ma + p \times \text{page fault time}$$

Where:

p = page fault rate

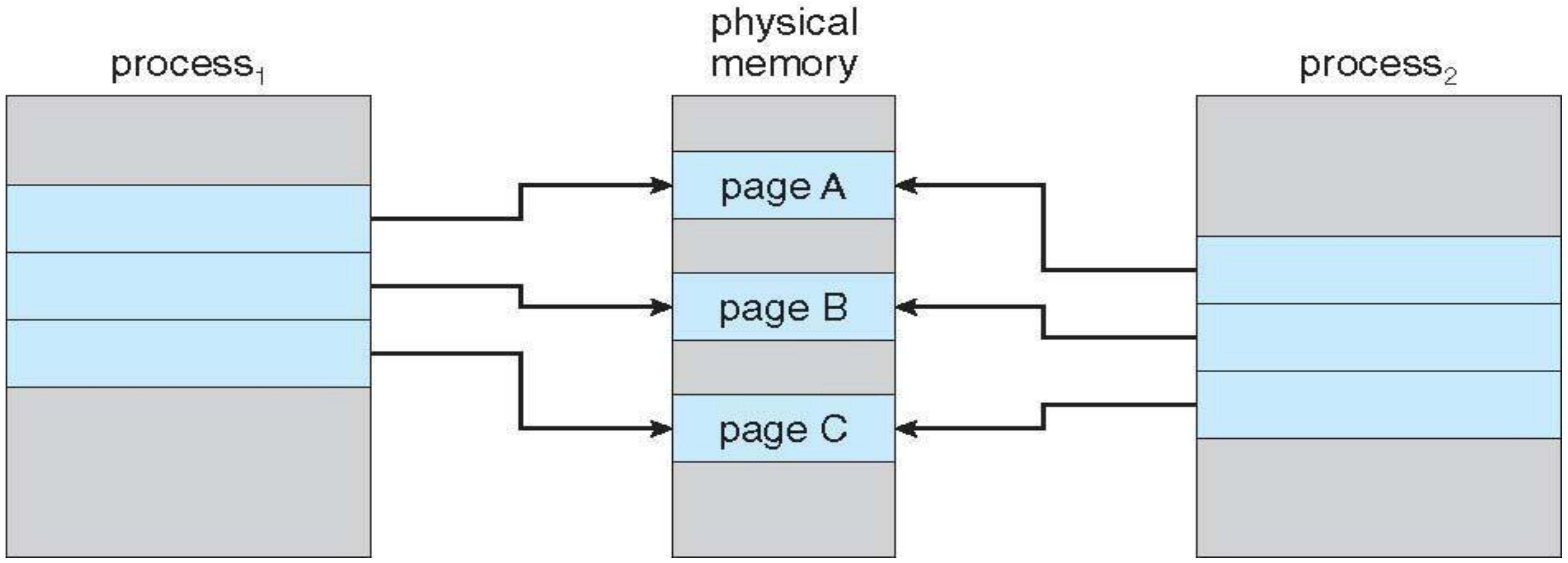
ma = memory access time (possibly with TLB etc.)

page fault time = trap to the operating system (context switch) identify page fault
+ locate the missing page on disk
+ allocate a free frame in memory (free-frame list)
+ schedule a transfer from disk to frame (wait in a queue)
+ receive an interrupt signal from I/O subsystem (I/O completed)
+ put the process in the ready queue (waiting time)
+ update page table (valid / invalid bit)
+ context switch (process gets CPU time)
+ restart the instruction that failed before

The effective access time is directly proportional to the page-fault rate. If one access out of 1,000 causes a page fault, the computer will be slowed down by a factor of 40 due to the demand paging! <u>If we want performance degradation to be less than 10 percent, that we may only allow to have 1 page fault per. 399,990 references to the memory.</u>

Copy-on-Write

A technique to speed up the creation of a child process

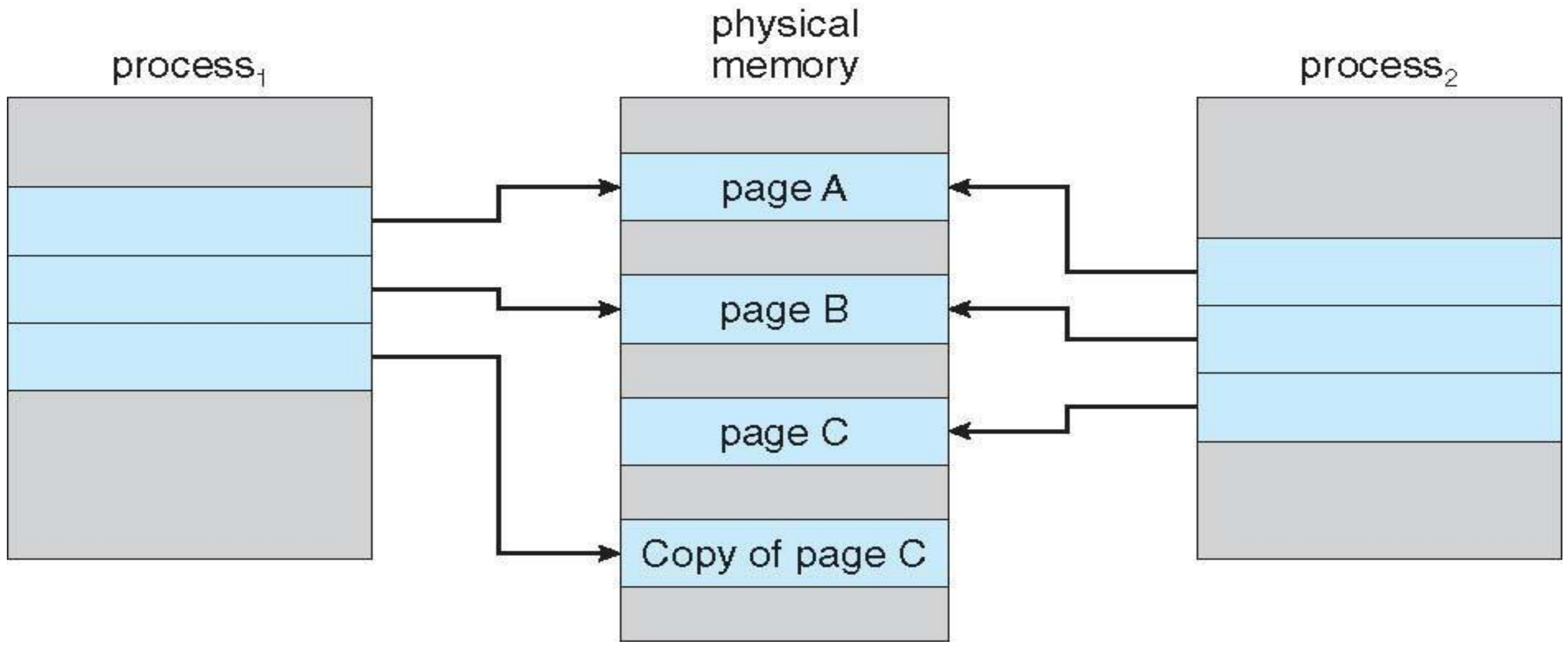


The fork() system call creates a child process that is a duplicate of its parent. Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.

Considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, a technique, **copy-on-write**, can be used to allow the parent and child processes initially to share the same pages.

Copy-on-Write

A technique to speed up the creation of a child process

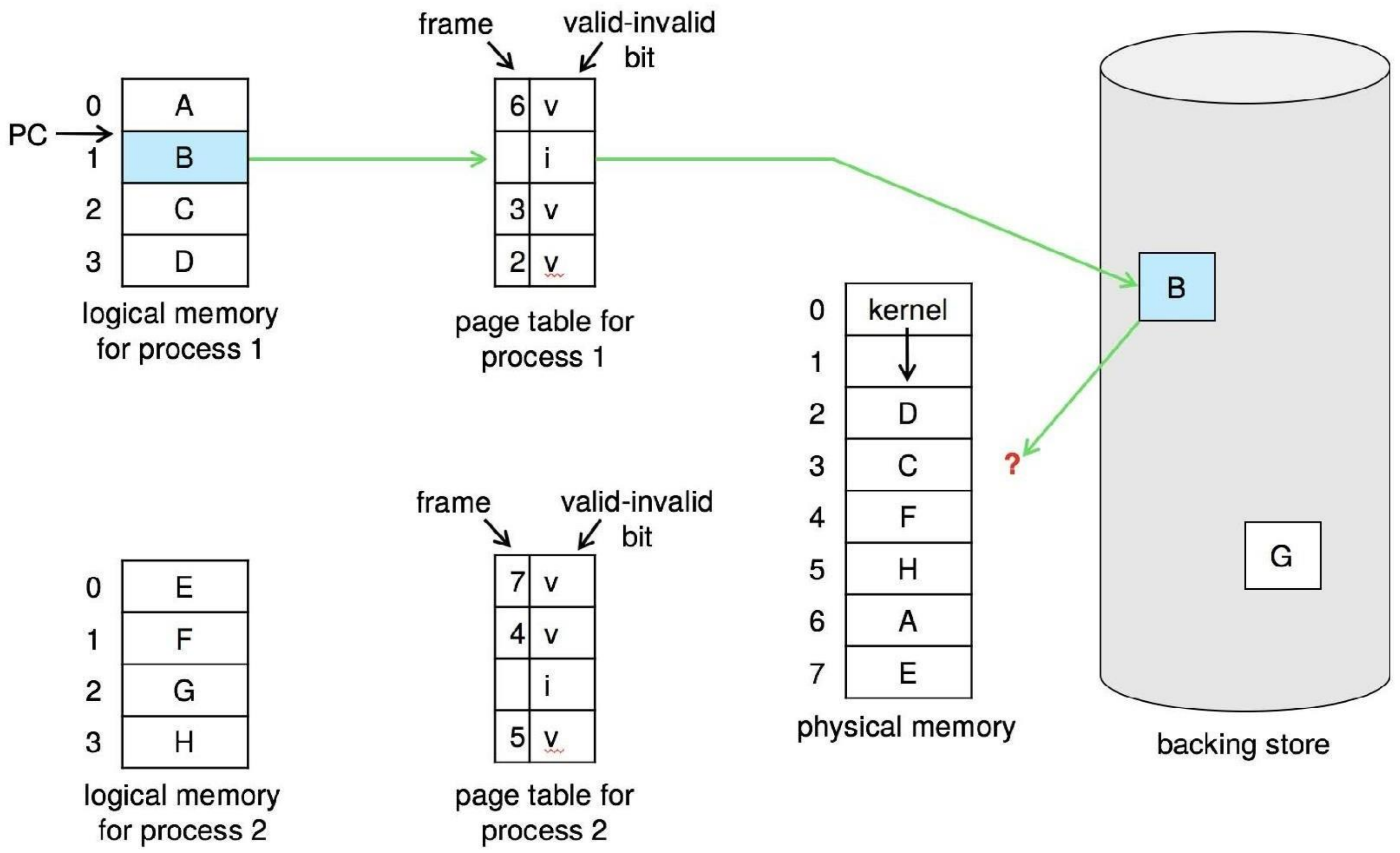


These shared pages are marked as **copy-on-write pages**, meaning that if either process writes to a shared page, a copy of the shared page is created.

Only the pages that are modified by either process are copied, and all unmodified pages can be shared by the parent and child processes.

Page replacement

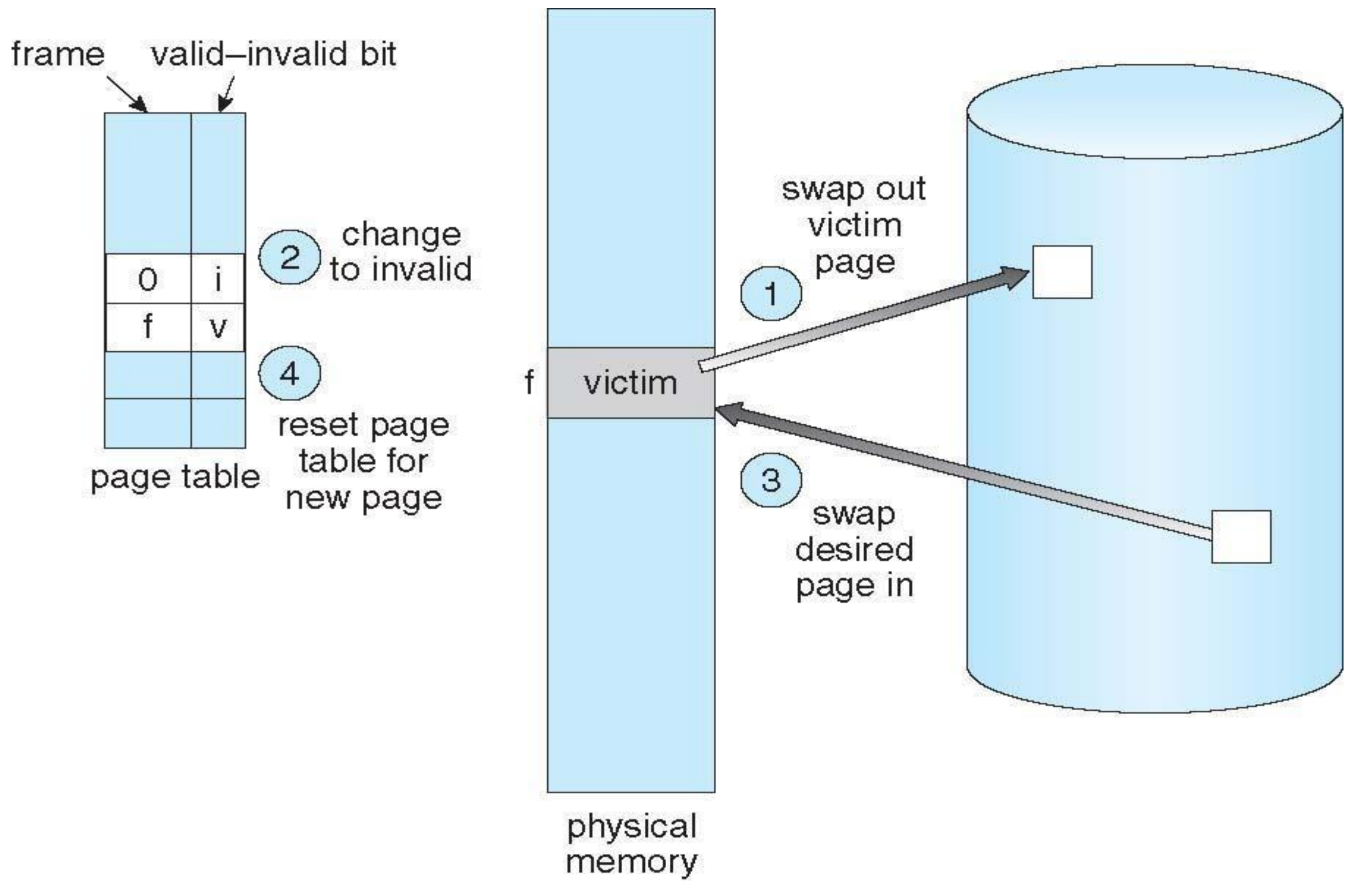
So far, we assume that there are always free frames for demand paging



But that is not always the case, and what do we do then?

Page replacement is an important technique to solve this issue.

Page replacement



(1) Find a **<u>victim (a frame not currently being used)</u>**, (swap the page out on the disk if it is modified)
(2) Correct the valid / invalid bit for the page that changed victim to invalid
(3) Swap in the page that caused the page fault.
(4) Update the valid / invalid bit and continue the process from where the page fault occurred.

Page replacement algorithms

We must solve two major problems to implement demand paging:
a frame-allocation algorithm and **a page-replacement algorithm**.

To select a page replacement algorithm, it is evaluated 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**.

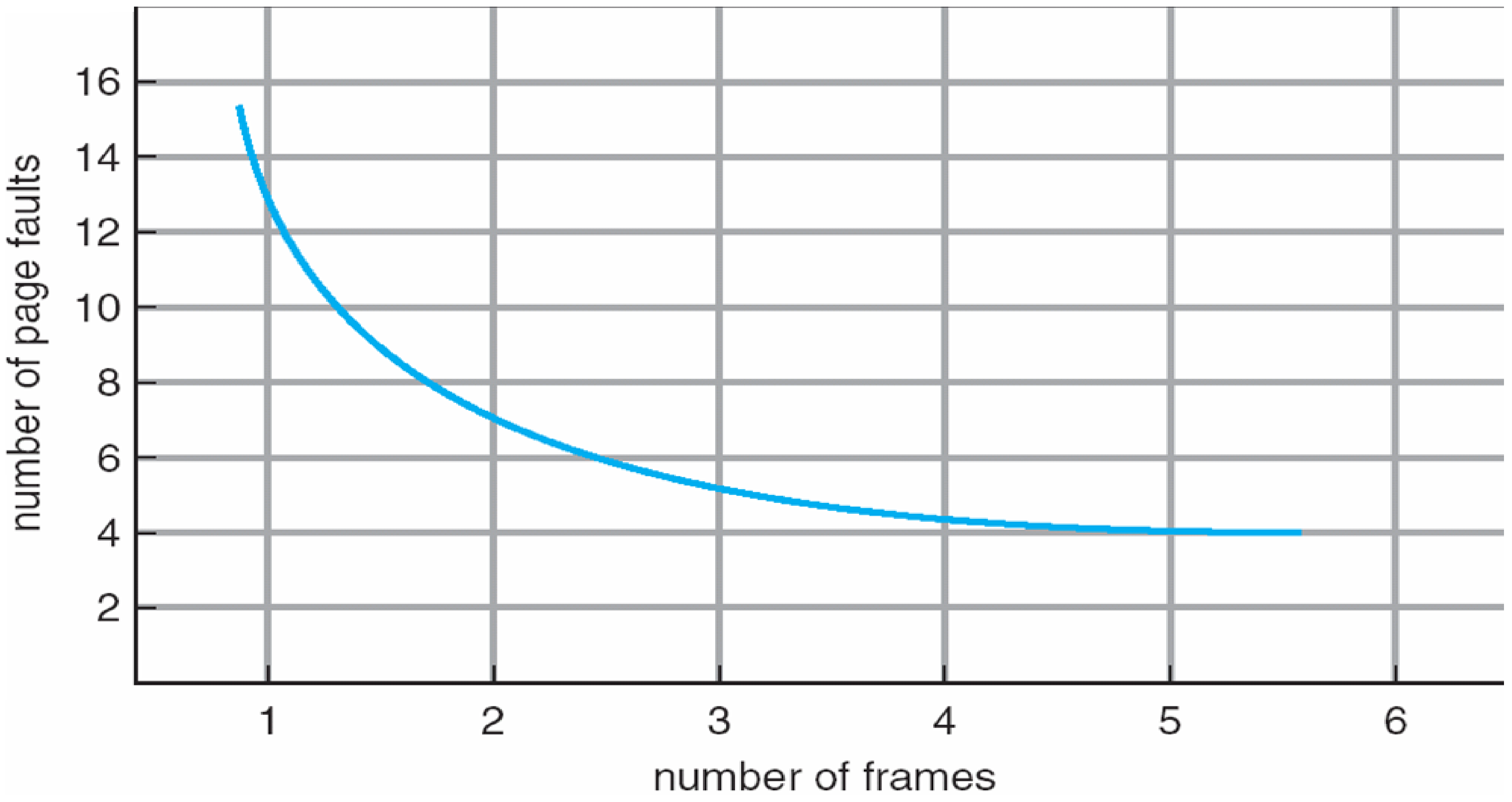
For instance, an address sequence is recorded as follows

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

can be reduced to the following reference string (E.g.: assume that a page is 100 bytes for convenience).

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Page replacement algorithms



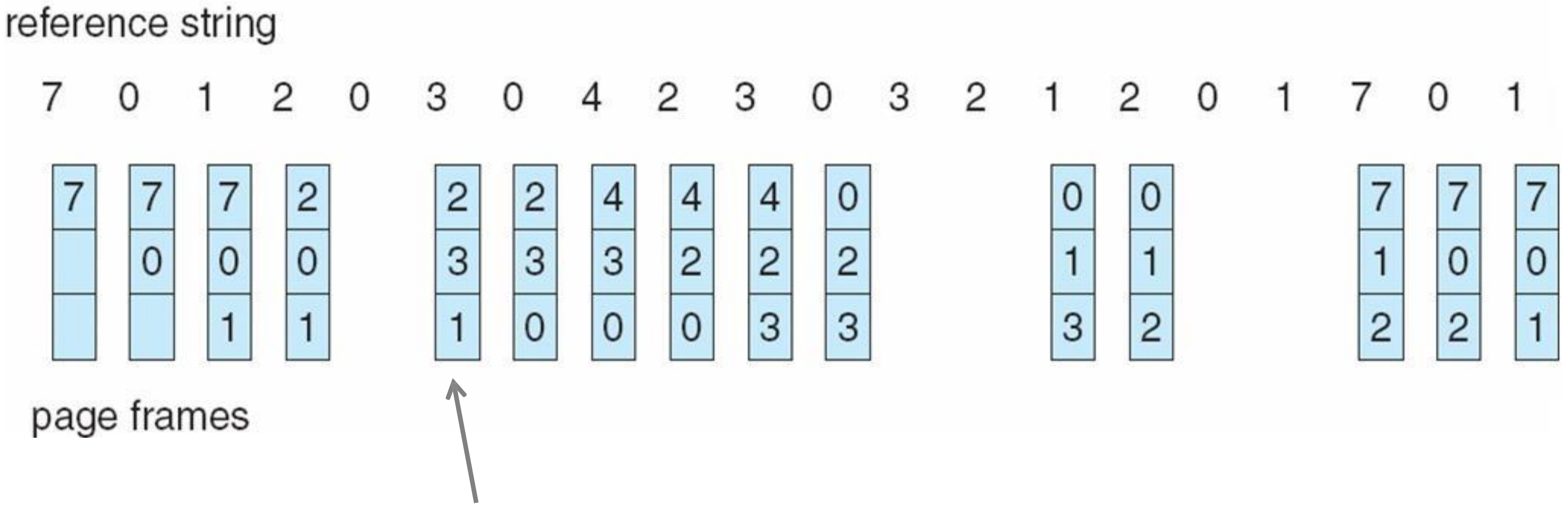
To determine the number of page faults for a particular reference string and page-replacement algorithm, we need to know the number of page frames available.

As the number of frames available increases, the number of page faults decreases.

Page replacement algorithms

First-in first-out (FIFO) algorithm

A FIFO replacement algorithm creates a FIFO queue to hold all the pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.



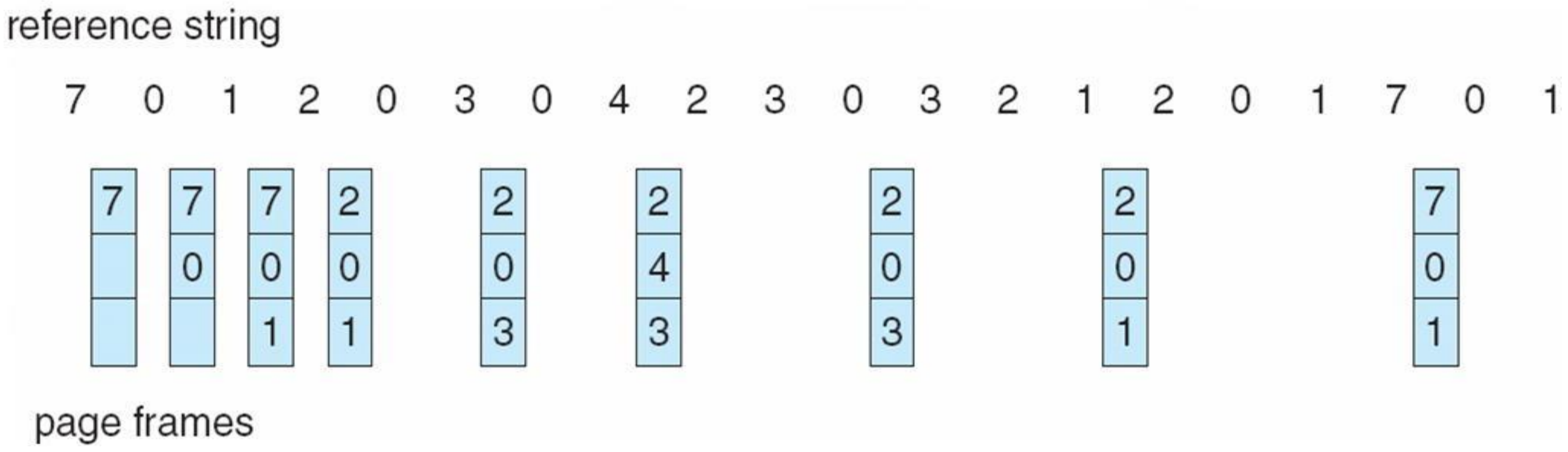
Every time there is a box, we have a page fault

Problem: if we select for replacement a page that is in active use, after we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. A bad replacement choice increases the page-fault rate and slows process execution.

Page replacement algorithms

Optimal page-replacement algorithm

The algorithm **replaces the page that will not be used for the longest period of time**.
It has the **lowest page-fault rate** of all algorithms has been called OPT or MIN.



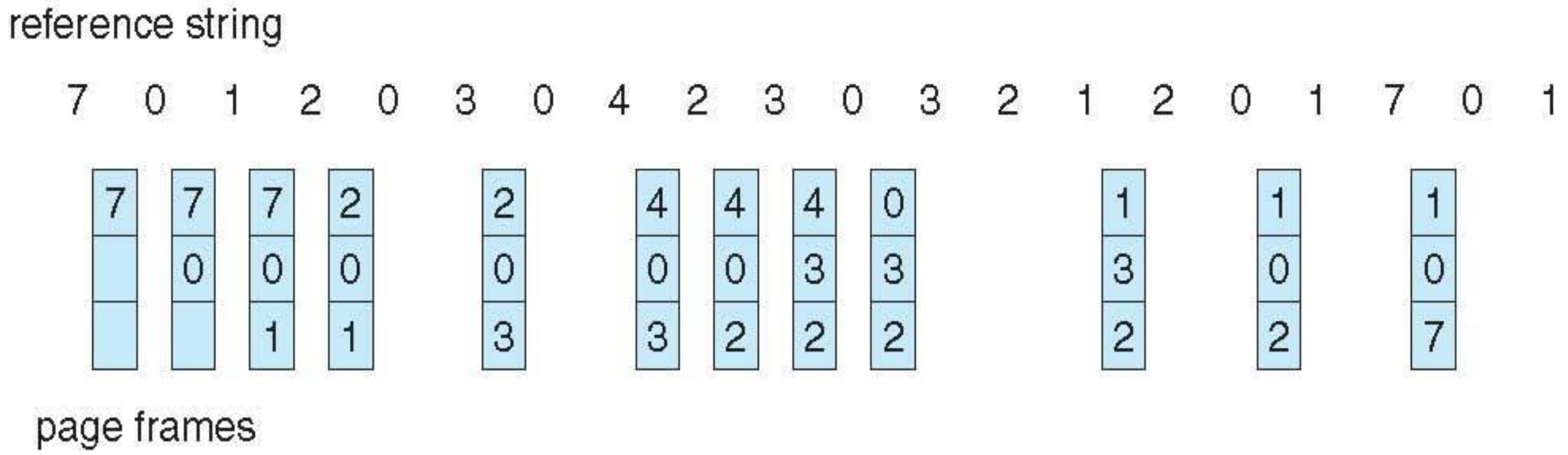
On our sample reference string, the optimal page-replacement algorithm would yield 9-page faults, and is better than a FIFO algorithm, which results in 15-page faults.

Problem: the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

Page replacement algorithms

Least Recently Used algorithm (LRU)

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. If we use the recent past as an approximation of the near future, then we can **replace the page that has not been used for the longest period of time**. This approach is the **least recently used (LRU) algorithm**.

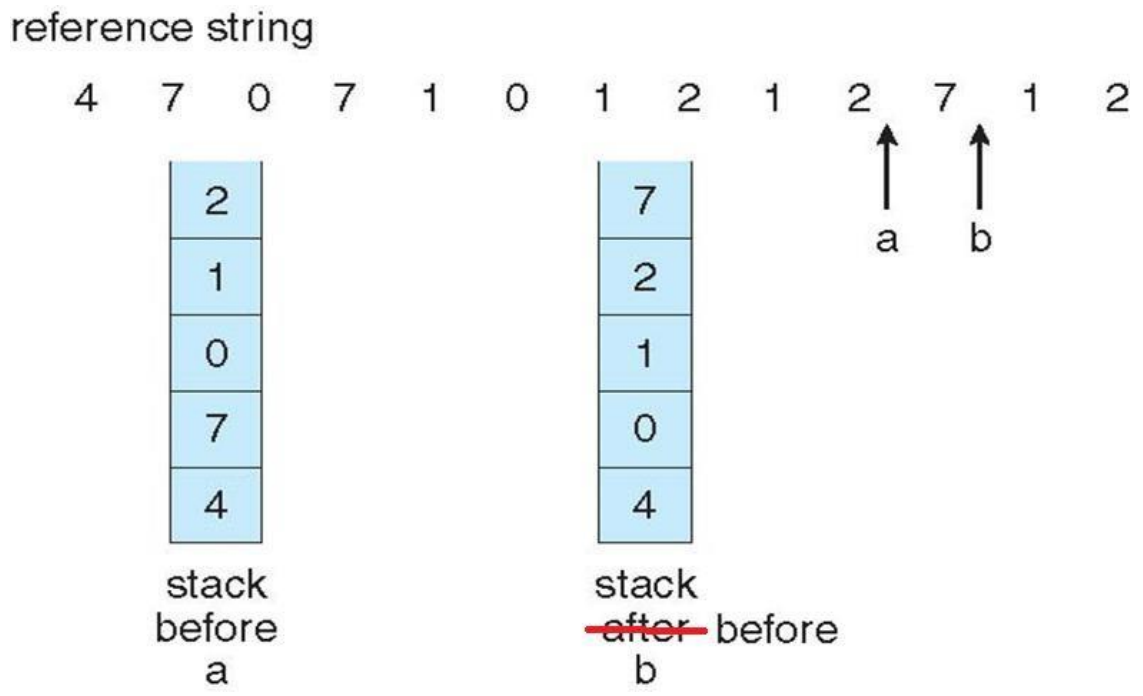


LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

Page replacement algorithms

Least Recently Used (LRU) – Stack-based

Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom.



This can be implemented by using a doubly linked list with a head pointer and a tail pointer. No search for a replacement. The tail pointer points to the bottom of the stack, which is the LRU page.

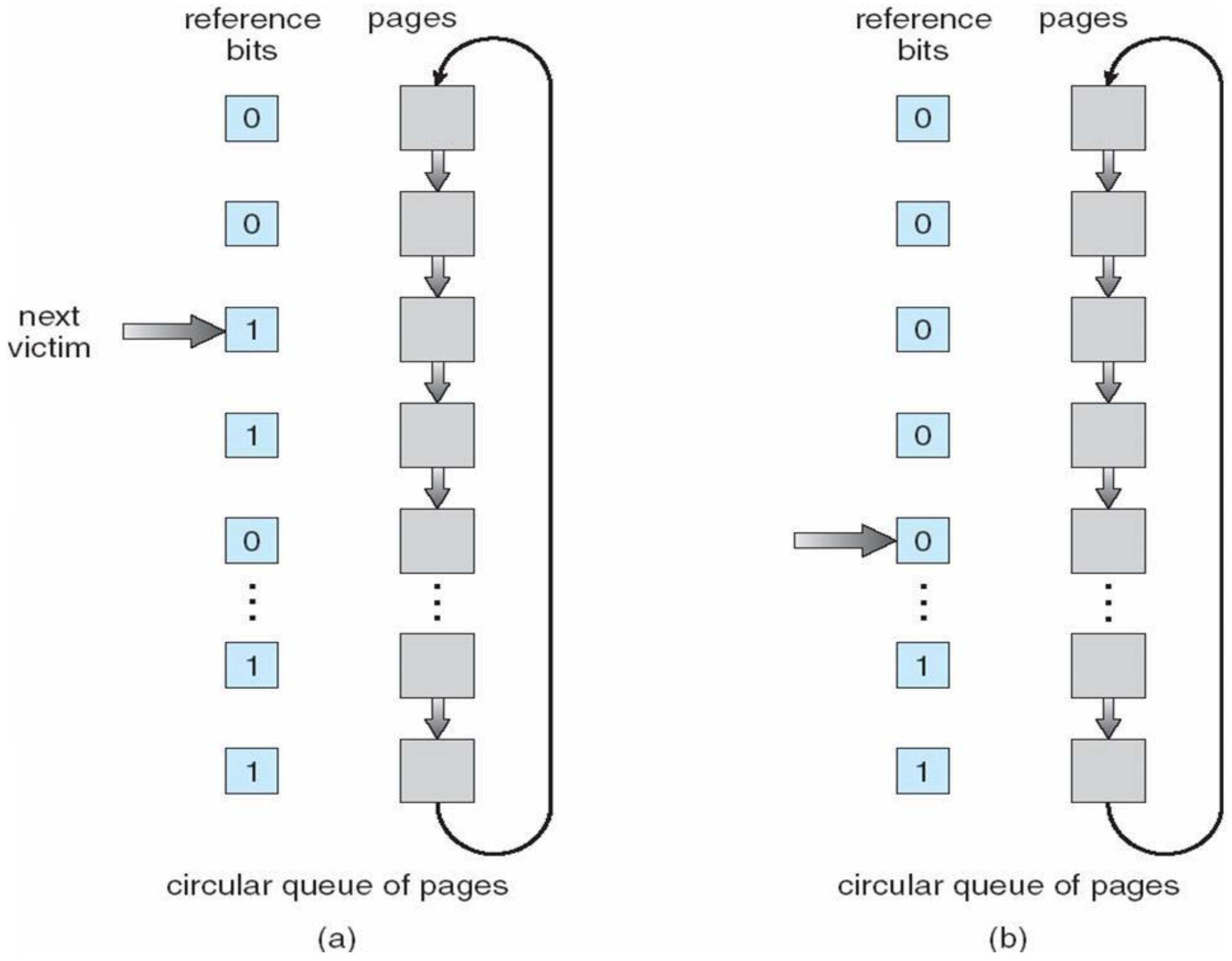
Page replacement algorithms

LRU-approximation page replacement – Second-Chance (clock) algorithm

Many systems provide some help in the form of a reference bit. The reference bit for a page is set to 1 by the hardware whenever that page is referenced.

If the value is 0, we proceed to replace this page. But if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared to 0.

Implementation: Circular queue



Page replacement algorithms

LRU-approximation page replacement – Enhanced Second-Chance Algorithm

In addition to the reference bit, a so-called **modify** (or **dirty**) **bit** is introduced to reflect if a page is modified or not. For a page, which is not modified, we need not write the memory page to storage since it is already in storage.

There will be 4 possible classes for an ordered pair in the decision process:

	Reference	Modify	
1	**0**	**0**	**not used or modified - best candidate for replacement**
2	**0**	**1**	**not used but modified - must be written back before replacement**
3	**1**	**0**	**used but not modified - will probably be used again soon**
4	**1**	**1**	**used and modified - will probably be used again soon and also written back first**

The algorithm replace the first page encountered in the lowest numbered class.
The major difference is the preference for replacing clean pages in order to reduce the number of I/Os required.

Page replacement algorithms

Counting-based Page Replacement

Some other algorithms use a **counter** of the number of references that have been made to each page to select the page to be replaced.

Least Frequently Used (LFU):
Here the page with the smallest count is replaced. Based on the philosophy that actively used page should have a large reference count.

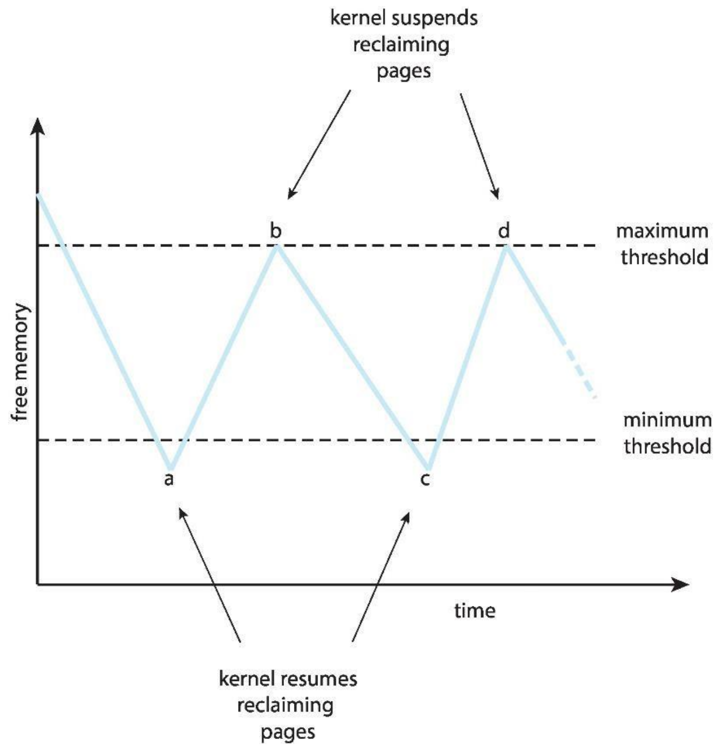
Most Frequently Used (MFU):
Here the page with the largest count is replaced.
This algorithm is based on an argument that the page with the smallest count value was probably just brought into memory and therefore need to be used shortly.

Neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Page-Buffering Algorithms

Systems commonly have a pool of “free frames”

- When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.
- The system may periodically review processes and reclaim pages to the pool.
- This action will often be done when the number of pages in the pool is below a minimum threshold.



The system may have a list of modified pages

- Whenever the paging device is idle, a modified page is selected and is written to secondary storage. Its modify bit is then reset, and the page can then be replaced later.

The system can keep track of the content of pages in its free frame pool

- When a page fault occurs, we first check whether the desired page is in the free-frame pool. If yes, then the old page in the pool can be reused again without the need to pick up from secondary storage.

Allocation of frames

Besides the page replacement, the other important issue in the implementation of demand paging is **frame allocation**, i.e., How do we allocate a fixed amount of free memory (frames) among the various processes?

Here are 2 main schemes when frames are to be allocated

- *Equal (or fixed)* allocation
- *Proportional* allocation

There is also a requirement for the minimum number of frames a process is assigned

- This is defined by the computer architecture (while the maximum number is defined by the amount of available physical memory).

Allocation of frames

Equal allocation - example

If we have a system with 128 frames with a 1-KB frame size

the operating system occupies 35 KB i.e., 35 frames

93 frames for user processes

If there are 5 processes, then they get <u>18 frames</u> each. (5 x 18 = 90 frames) the leftover 3 pages are placed in the free-frame buffer pool.

Since the processes can have very different sizes, so this may <u>not</u> be a fair distribution of resources.

Allocation of frames

Proportional allocation - example

Assume we have m = 62 available free frames for user processes

if there are 2 processes, one of 10 KB and the other one of 127 KB and they got 31 each according to the equal allocation

- then the small process would not be able to utilize all its assigned frames.
- and the big process would get unnecessarily many page faults.

Assume s_i is the virtual memory of process p_i, and define S:

$$S = \sum s_i$$

The number of frames allocated to pi is $a_i = \frac{s_i}{S} \cdot m$

$$a_1 = \frac{10}{137} \cdot 62 = 4.53 \approx 4$$

$$a_2 = \frac{127}{137} \cdot 62 = 57.47 \approx 57$$

Allocation of frames

Local and Global allocation

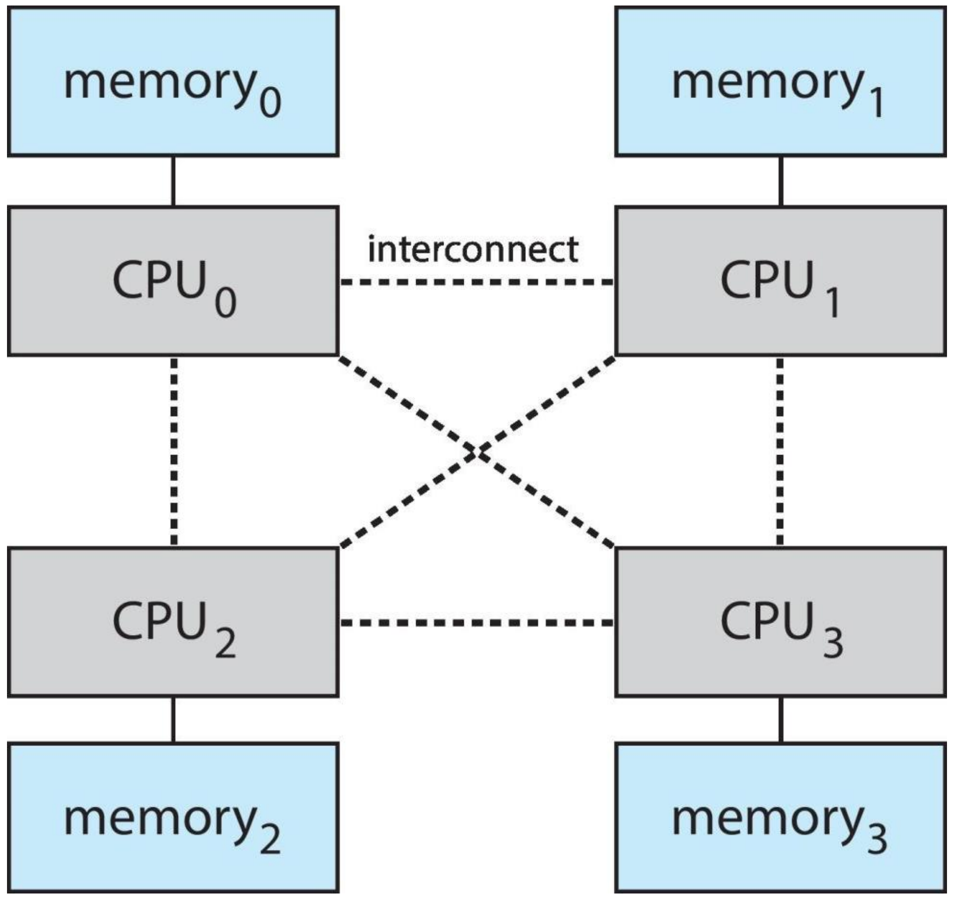
Another important factor is how frames are allocated to processes when there are no more available frames in page replacement.

There are 2 options for multiple processes competing for frames.

- *Local allocation* - page replacement occurs within the process's own frames
  - more consistent performance for each process without influence of different external circumstances (the other processes)
  - possibly poorer utilization of memory

- *Global allocation* - page replacement occurs among the set of all frames including those from all lower-priority processes
  - execution time for processes can vary greatly due to the "theft"
  - but better system throughput, so therefore most commonly used.

Non-Uniform Memory Access (NUMA)

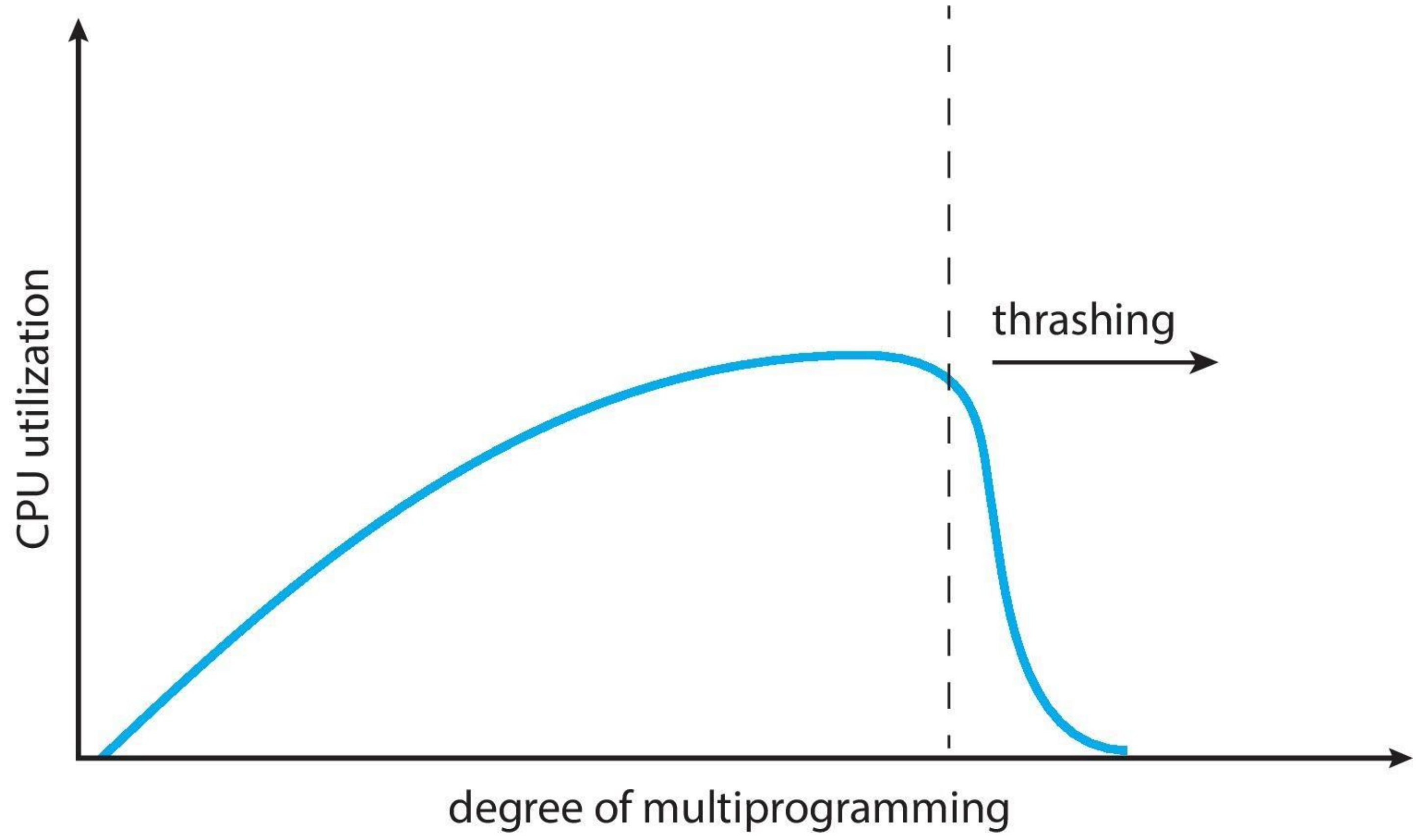


Until now, we have assumed that all main memory is created equal. However, on **non-uniform memory access (NUMA)** systems with multiple CPUs, a given CPU can access local memory section faster than it can access other CPUs' memory sections.

The OS must take into account the CPU affiliation of processes.

Thrashing

When a process spends almost all its time on page replacement. This high paging activity is called <u>**thrashing**</u>.



Processes leave the ready queue for CPU and queue up for the paging device

Thrashing

The effects of thrashing can be limited by using a **local replacement algorithm**.
As mentioned earlier, local replacement requires that each process select from only its own set of allocated frames. Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

To prevent thrashing, we must provide a process with as many frames as it needs. One strategy is to utilize the **locality model** to figure out the current need of a process. Because all processes have locality (e.g., 90% of the time within 10% of the code)

- **processes move from one locality to another**
- **localities may have overlap**

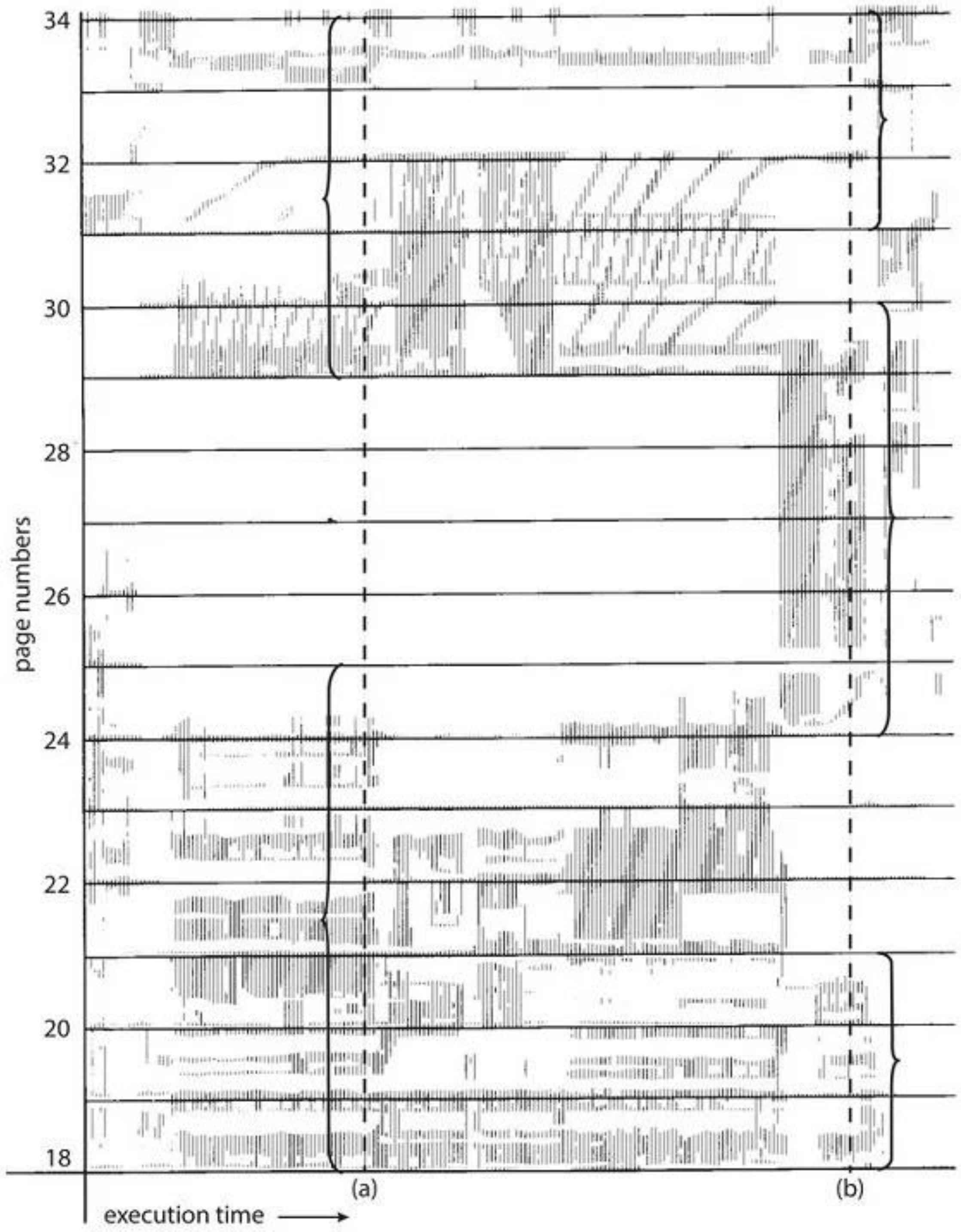
Locality

An example

Locality at time (a) is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}

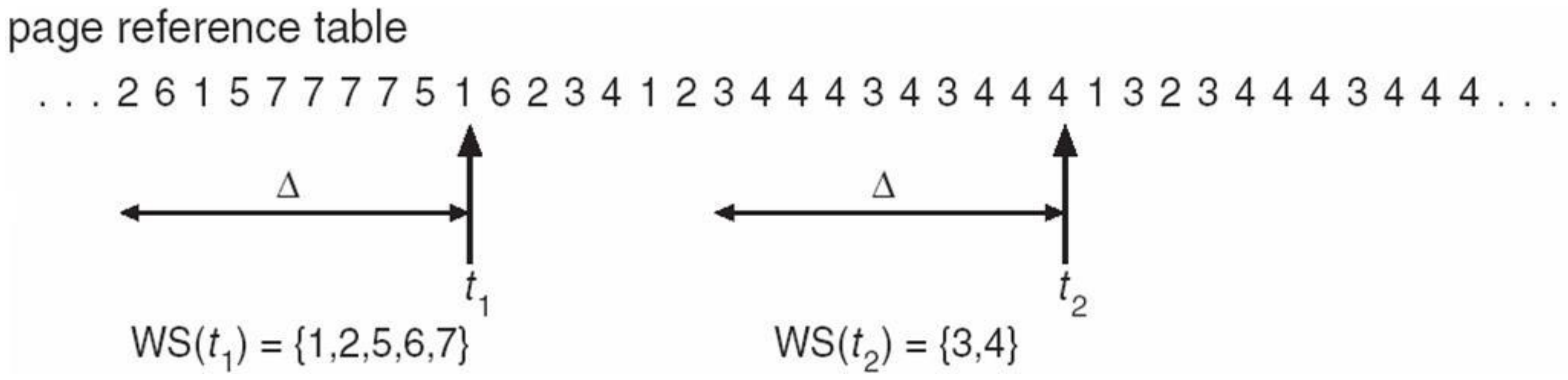
Locality at time (b) changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}

Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.



Working-set model

The **working-set model** is based on the assumption of locality. This model uses a parameter, Δ, to define a **working-set window**. Δ is the number of the most recent page references. The set of pages in the working-set window is the **working set** .

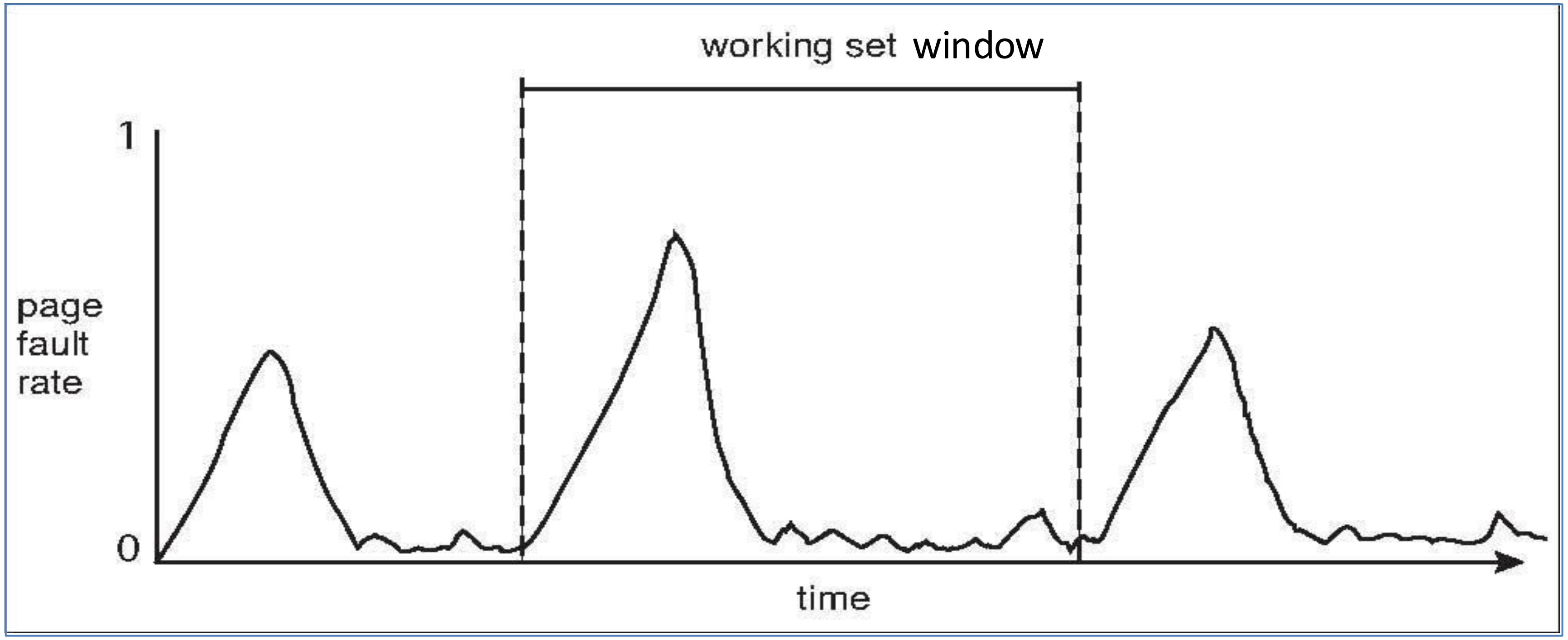


A process performs well if it has enough pages that cover its working set. This working set can vary over time! Assume WSSi is the working-set size for each process

$$\sum \boldsymbol{WSs_i} > \ \textbf{\textit{physical memory}} \rightarrow \textbf{\textit{thrashing}}$$

If the sum of the working-set sizes (total demand for frames) exceeds the amount of physical memory, then thrashing occurs.

Working-set model

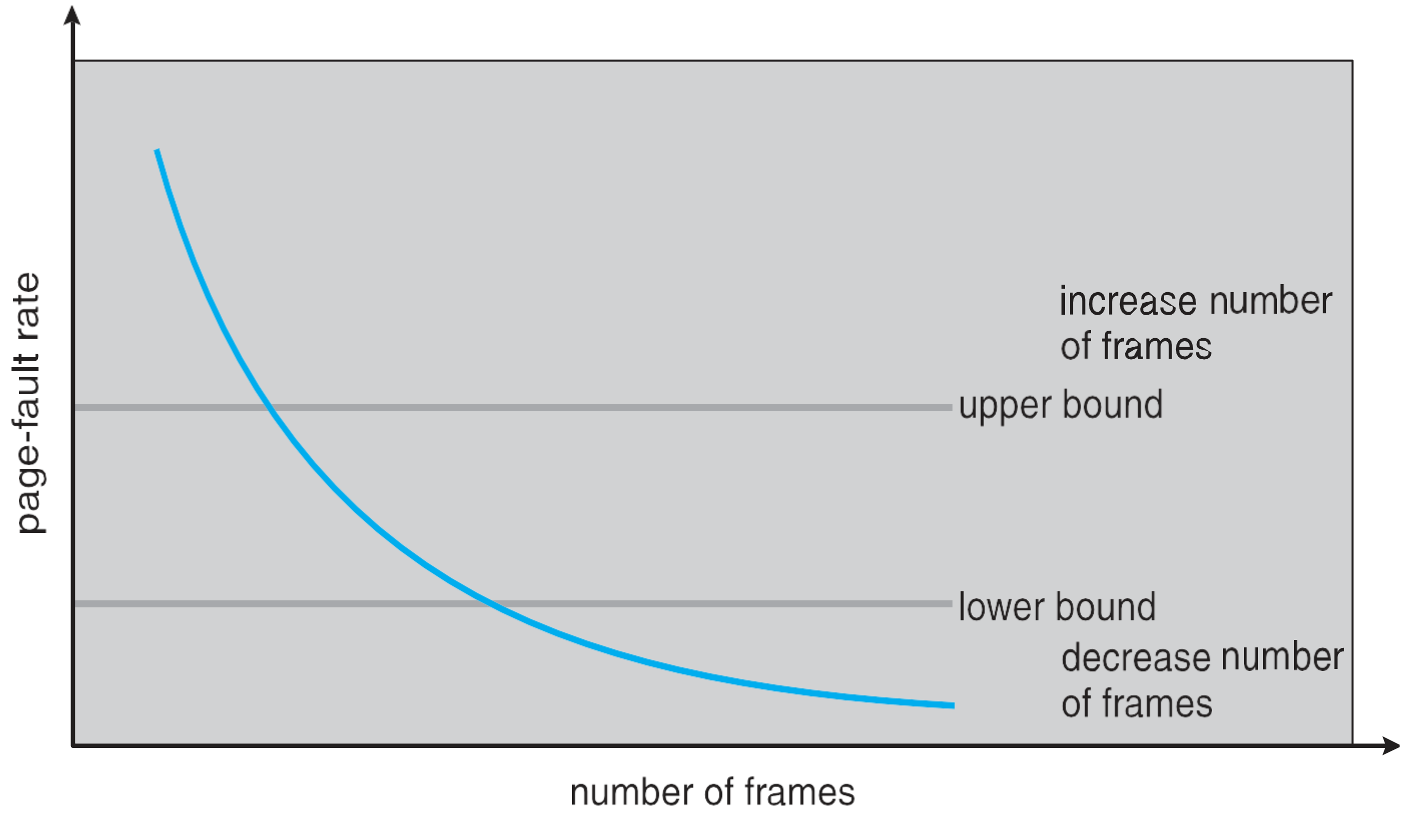


A peak in the page-fault rate occurs when we begin demand-paging a new locality.

However, once the working set of this new locality is in memory, the page-fault rate falls.

The span of time between the start of one peak and the start of the next peak represents the transition from one working set to another.

Page-Fault Frequency



- If the page-fault rate falls below a lower bound: the OS reclaim frames from the process to the free frame list.
- If the page-fault rate exceeds an upper bound: the OS allocate frames to the process from the free frame list.

However, the best solution to thrashing and high page fault rates is always

More memory!