Virtual Memory

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Virtual Memory

In this chapter, we will:

1. Define what virtual memory is, and explain what issues it solves.
2. Define and illustrate the methods of Demand Paging, Copy-on-Write, and Page Replacement.
3. Introduce strategies for allocating frames, and analyze the issue of thrashing.
4. Find out how to map files onto memory, and what benefits this approach brings.

Virtual Memory

The following video defines and illustrates the concept of virtual memory in a nutshell:

https://www.youtube.com/watch?v=qlH4-oHnBb8

Virtual Memory

Any questions?

Virtual Memory

Suppose that you wrote a program that needs to use more memory space than main memory can offer. If the operating system attempts to fully store the process in memory, the process will crash due to lack of memory.

A solution to this situation would be to treat the program as virtual memory.

Virtual memory is the logical view of the program, and can be much larger than the existing physical memory. With virtual memory, we can choose to store only the necessary process pages in main memory, while storing the unused ones on disk until we need them.

When a page that is currently stored on disk is needed to be used, the operating system will copy some currently loaded page from memory back to disk (to provide enough space for the new page) and load the needed page to memory.

Virtual Memory

Why is it possible to load some page to memory while keeping others on disk?

1. Programs often contain code for handling errors (e.g., using `try-catch` blocks) that occur rarely. During the execution of a program, if such a rare error doesn't occur, it is not necessary to keep the error-handling instructions in memory until the error happens.
2. When an array (or another data structure that serves like a container) is created, many times a very large size is allocated for the array, just in case the worst case occurs for a certain algorithm. As long as the worst case doesn't occur, we can store only the used-up potion of the array in memory.
3. Many functions/methods in a program are rarely used, so it is not necessary to keep the instructions of such functions in memory until the program calls the function(s).

In other words, there are many instances in which we could choose to keep a certain process page on disk rather than loading it to memory.

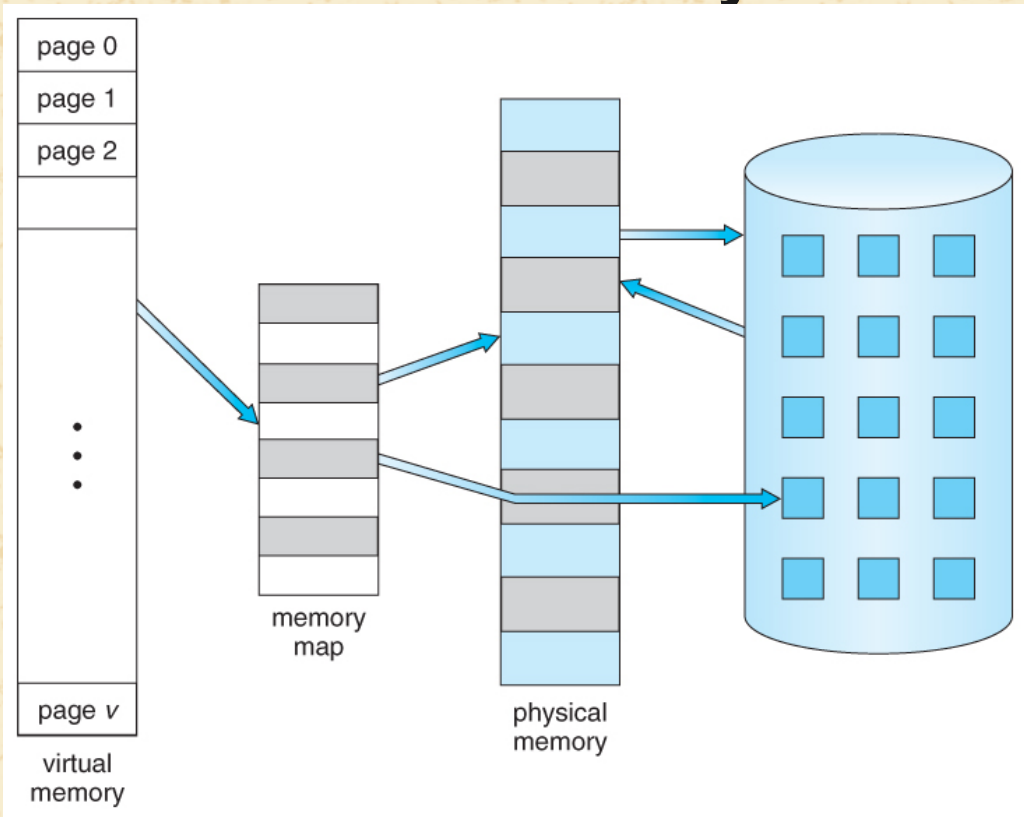
Virtual Memory

Benefits of loading only a portion of a process's pages into memory:

1. Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer. Loading all the page will crash the program, so if we decide to load only some pages at a time, the program could successfully execute.
2. Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
3. Less time is needed for swapping processes in and out of RAM during context switches.

The figure on the following slide shows the general layout of virtual memory.

Virtual Memory



General layout of virtual memory. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging

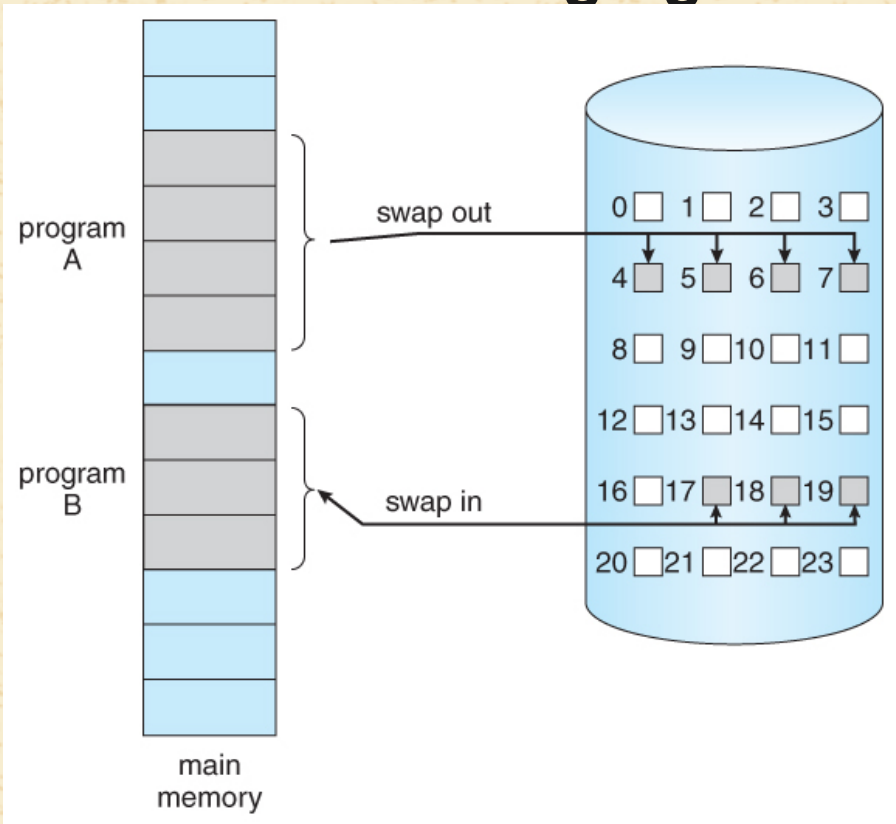
When a process is loaded to memory, but its pages are not loaded all at once (but only on demand) is called **Demand Paging**. That is, only when the process needs the data/code on a specific page, this page will be copied from disk to memory.

We call the mechanism behind such paging a **lazy pager** (or **lazy swapper**.)

To tell what pages are currently loaded to memory and which ones aren't, we add another column to the page table called the **invalid bit**. If a page is loaded onto a main memory frame, the bit will be 1 (valid); if it isn't loaded, the bit will be 0 (invalid). The OS will use this column to learn which pages are present in memory at the current moment (= **memory resident** pages).

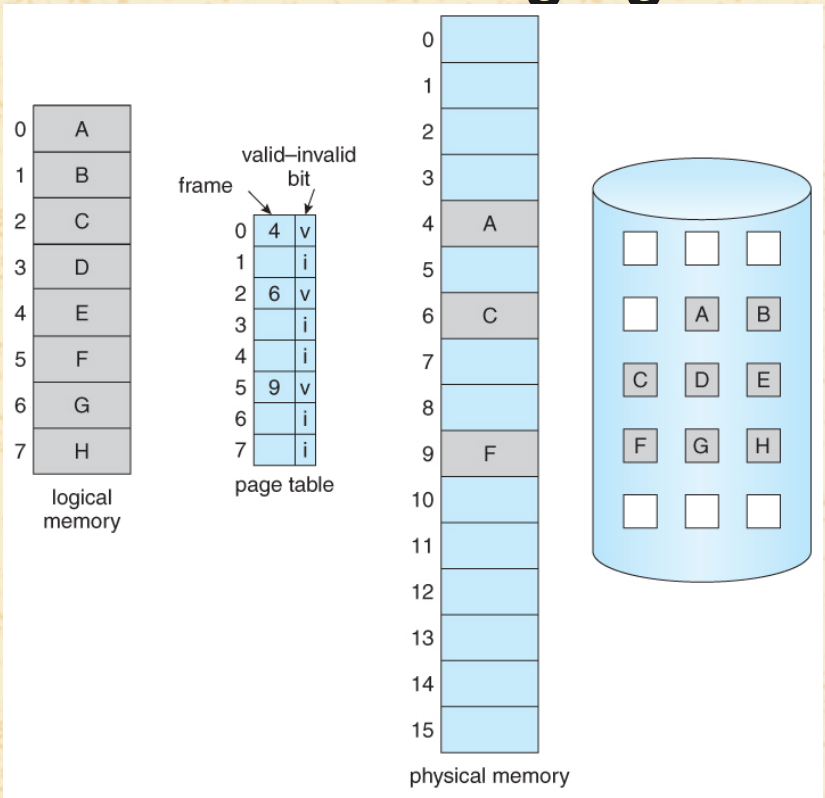
In case a process needs to use a page that is not currently in memory, a **page fault** trap (= software interrupt) will happen. To handle this interrupt, the OS will copy the content of the missing page from disk to memory.

Demand Paging



Swapping pages in and out. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging



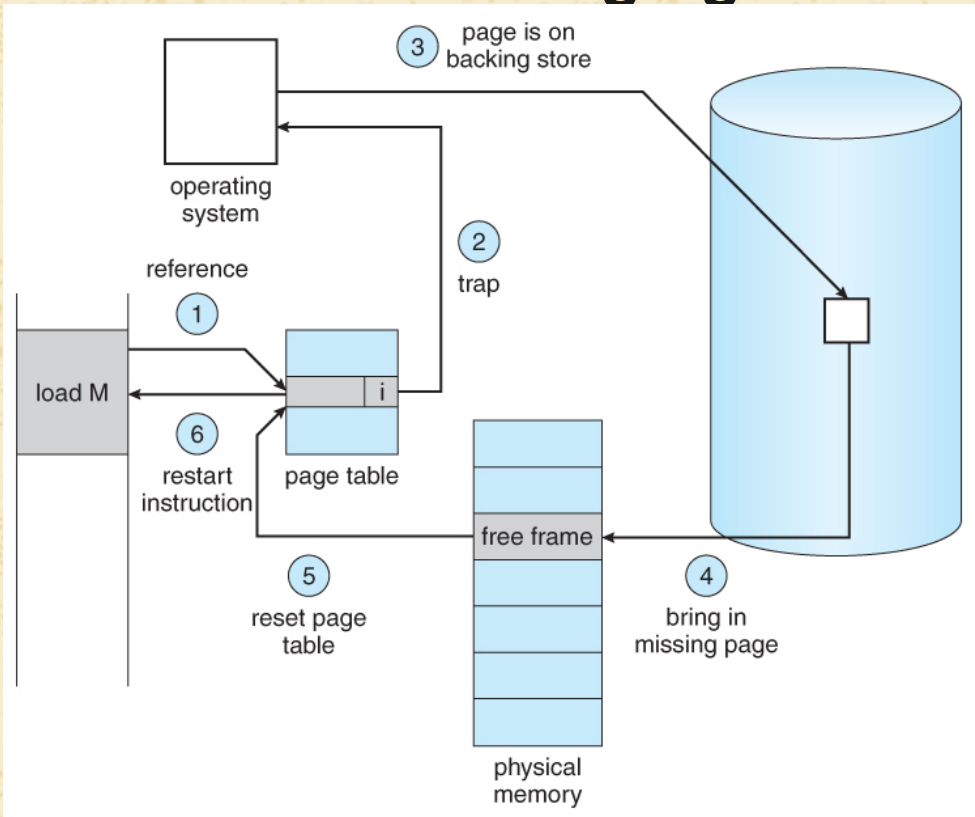
Some pages are out of memory. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging

An OS handles a page fault trap as follows:

1. The memory address requested is first checked, to make sure it was a valid memory request (that is, that this address doesn't belong to another process.)
2. If the reference was invalid, the process is terminated. Otherwise, the page must be **paged in** (= brought from disk to memory.)
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk, during which the process is blocking (= waiting).
5. When the copy from disk finishes, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted *from the beginning*, and this process will continue executing on the CPU.

Demand Paging



Steps in handling a page fault. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Demand Paging

A page fault incurs slowdown in the activity of the OS. We can find the new **average memory access time** that takes such a slowdown into account. To calculate it, we need to know (1) the normal average memory access time (which we calculated in Topic 7, slide 36), (2) the time it takes to handle a page fault, and (3) the **page fault rate** on a scale from 0 to 1 (the fraction of memory accesses in which page faults happen).

Examples:

- If the normal average memory access time is 200 ns, the page fault handling time is 8000000 ns, and the page fault rate is 0.001, the updated average memory access time would be 0.001 * 8000000 + (1 - 0.001) * 200 = 8000 + 199.8 = 8199.8 ns (about 8.2 microseconds), which is about 40 times greater than 200 ns!
- If the normal average memory access time is 50 ns, the page fault handling time is 8000000 ns, and the page fault rate is 0.002, the updated average memory access time would be 0.002 * 8000000 + (1 - 0.002) * 50 = 16,000 + 49.9 = 16,049.9 ns (about 16 microseconds), which is about 320 times greater than 50 ns!

Copy-on-Write

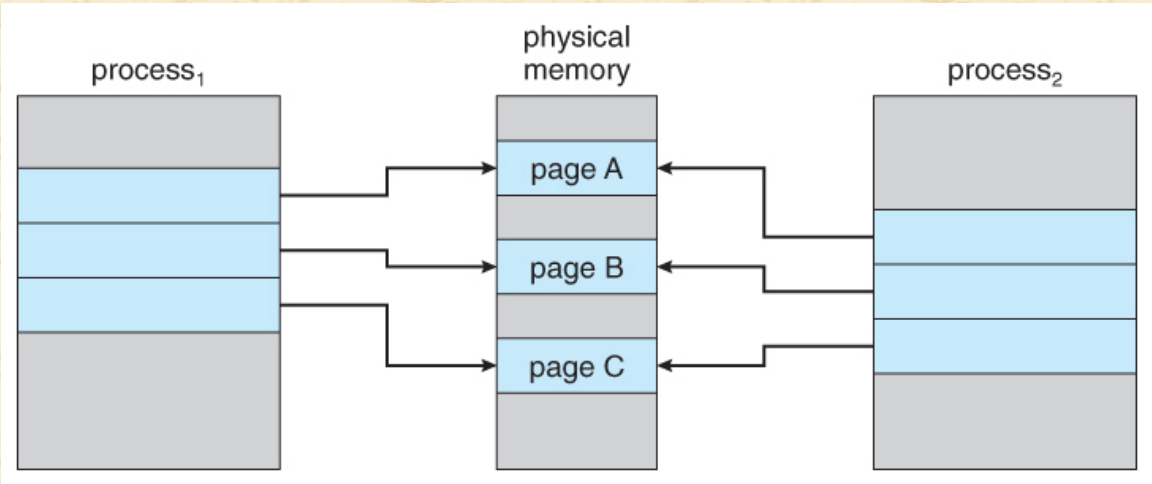
Suppose that a process created a child process using the `fork()` system call. As we remember from slide 22 of Topic 4 on processes, this will cause a new process to be created in memory, and its memory chunk will be *identical* to this of the parent.

Having learned the tool of virtual memory, we realize that we can eliminate the full copy of the parent's memory section to the child's memory section. We could decide to only copy those pages in the child process that *differ* (after a change was made) from those of the parent. This technique is called **Copy-on-Write**.

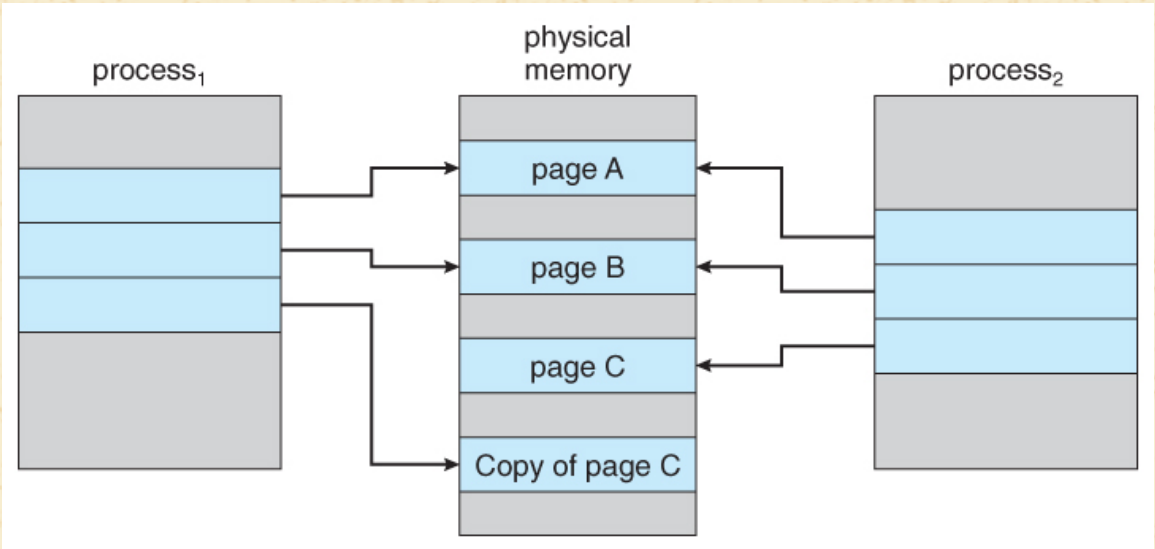
Since in many cases, we call the `exec()` system call right after `fork()`, this technique is actually very useful (since the pages of the child process won't get to change prior to calling `exec()`.) Recall that `exec()` will replace the memory section that was copied (or not copied) from the parent by the original code, data, stack, etc. of the child process.

Copy-on-Write

Suppose we have 2 processes, Process 1 (the parent) and Process 2 (the child of Process 1). When one of the pages of Process 1 changes, only then we create a new page copy. Otherwise, both processes will point to the same pages.



Before process 1 modifies page C. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.



After process 1 modifies page C. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Copy-on-Write

Any questions?

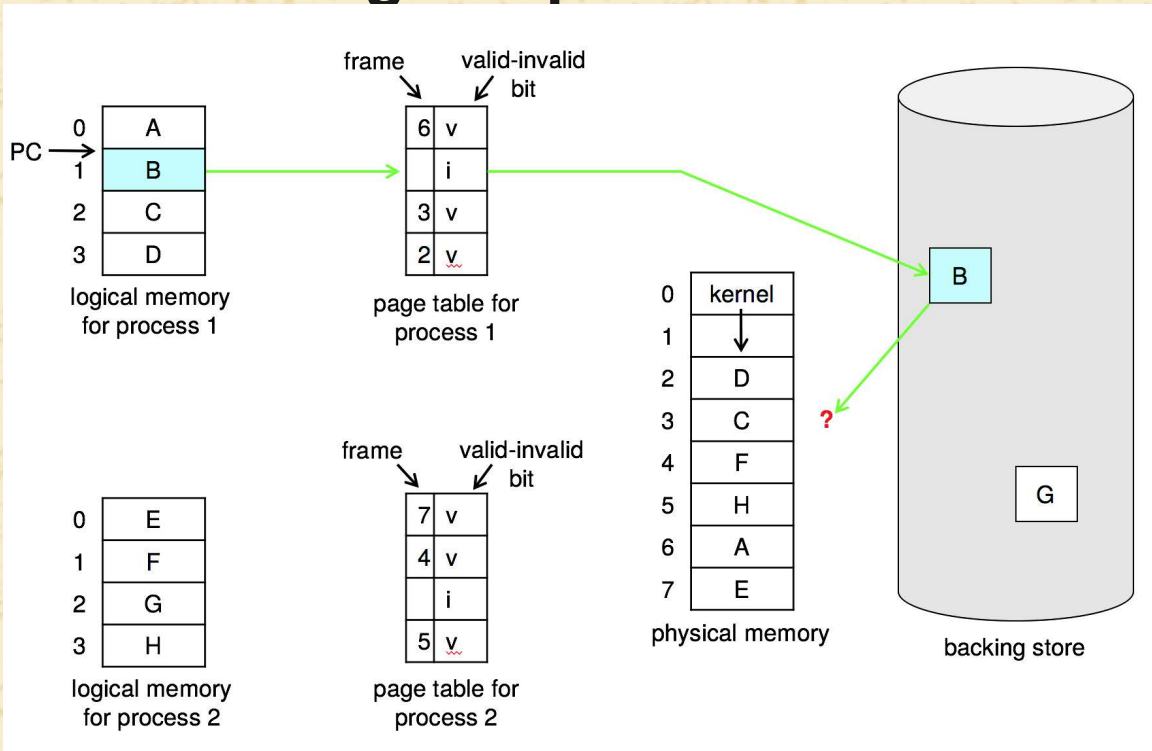
Page Replacement

Suppose that a process needs to load a page from disk into memory *but there are **no** free frames available.*

The operating system will do either of the following to allow the page to load:

1. A considerable part of memory is used to store I/O buffers (= arrays with I/O data.) Freeing some buffers, if they aren't currently needed, would free space for more frames.
2. Put the process into a waiting state until other processes go into waiting or terminate so that enough frames become available.
3. Swap some process out of memory completely, freeing up its page frames.
4. Find some page in memory that isn't being used right now, and swap that page only out to disk (without swapping the entire process,) freeing up a frame that can be allocated to the process requesting it. This is known as **page replacement** and is the most commonly used solution. There are many different algorithms to implement it.

Page Replacement



A need for page replacement. Taken from Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, 10th Edition", Chapter 10 (Virtual Memory).

Page Replacement

The procedure of page replacement goes as follows:

1. The operating system finds the location on disk of the page that is to be loaded.
2. The operating system searches for 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 an existing frame to be replaced, known as the **victim frame**.
        i. Copy the data of the victim frame to disk in order to free the frame up.
        ii. Change all related page tables to indicate that this page is no longer in memory.
3. Read in the desired page from disk and store it in the memory frame that was freed. Adjust all related page and frame tables to indicate the change.
4. Resume the execution of the process that was waiting for this page.

Page Replacement

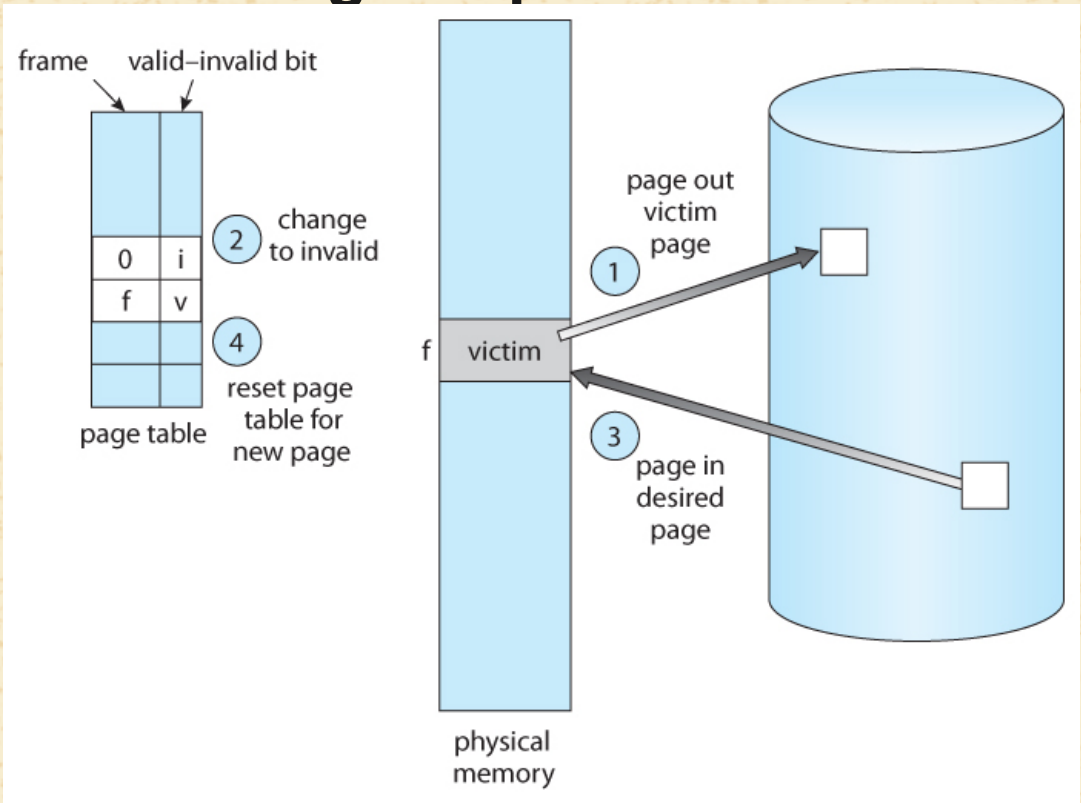


Illustration of page replacement. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement

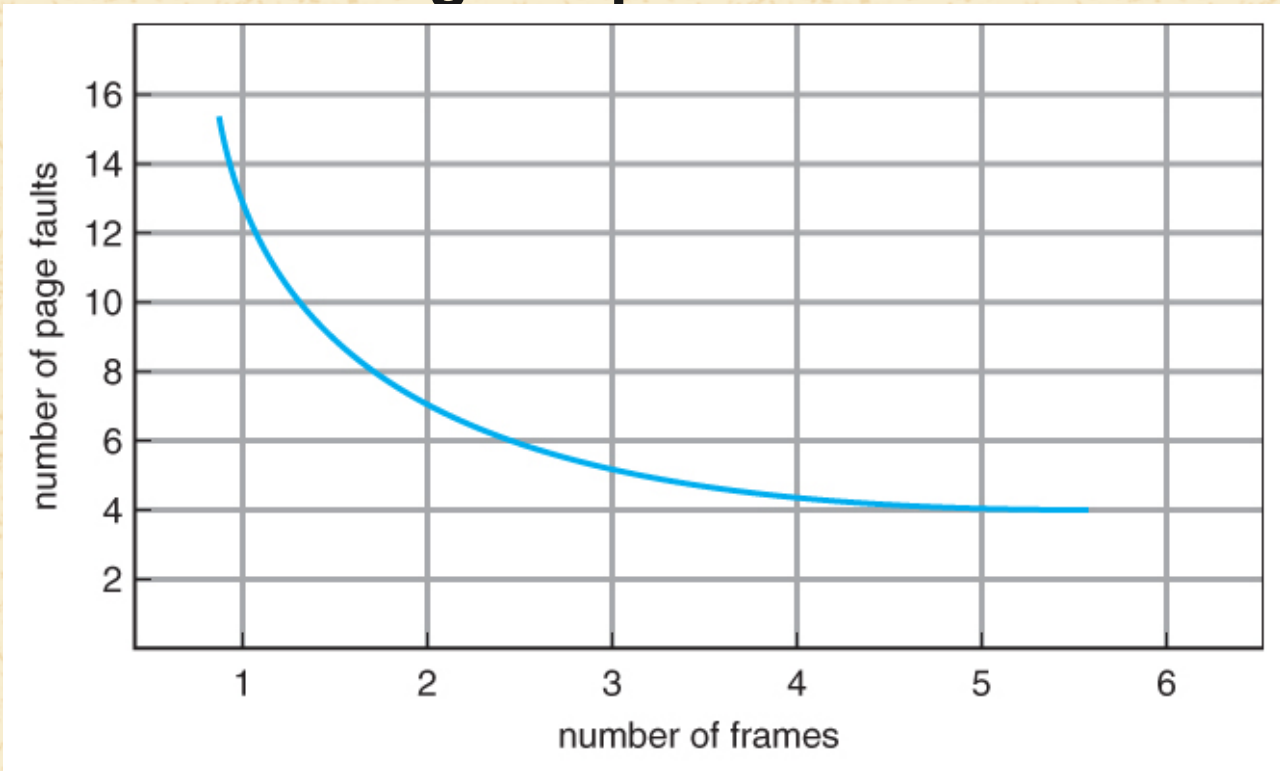
Note that the copying of the victim page from the frame back to disk doubles the extra time we spent during page faults (in addition to the time it takes to copy the page we want to bring from disk to memory.)

- As such, most algorithms will try evicting a page that hasn't been modified since the time it was brought from disk. This will spare the operating system from the need to copy the page to the disk (since it didn't change at all!)
- To know which pages changed or didn't change, the OS can store a bit per each page called **modify bit** or **dirty bit**. If the bit is `1`, this means that a change was made to some data on this page. Otherwise, the bit will be `0`.

A goal of the algorithms we'll see is not only to make page replacement less time consuming but to reduce the number of occurring page faults. Working with more frames should reduce page faults, too, as the next slide diagram shows.

After we discuss each algorithm, we'll demonstrate how it works by running it on a sample sequence of pages that a process might request to load. We call this sequence the **reference string**.

Page Replacement

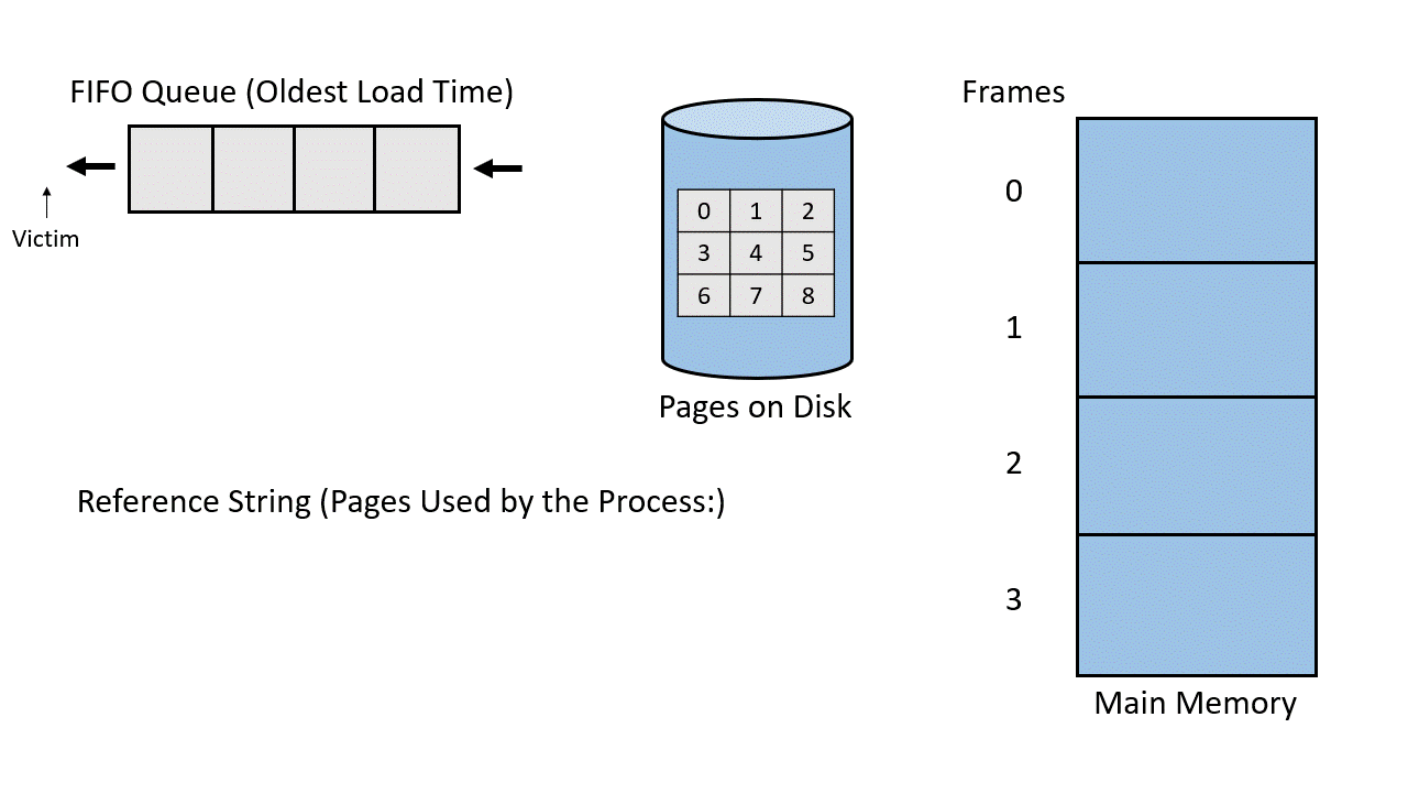


Number of page faults versus number of frames. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement Algorithms

1. **FIFO Page Replacement**.
   - We maintain a queue of pages based on the order in which they were loaded to memory.
   - As new pages are brought into memory, they are added to the tail of a queue, and the page at the head of the queue (the page sitting in memory for the longest time) is the next victim.
   - The victim page will be removed from memory (copied back to disk,) and the page that the process is requesting will be copied from disk into memory.
   - This algorithm is simple, but not efficient, as it will result in a large number of page replacements.
   - **Belady's anomaly** is a phenomenon in which an increase in the number of frames leads to an increase in the number of page faults (which is counterintuitive.) FIFO Page Replacement exhibits this phenomenon, as we shall see in a future slide.

Page Replacement Algorithms

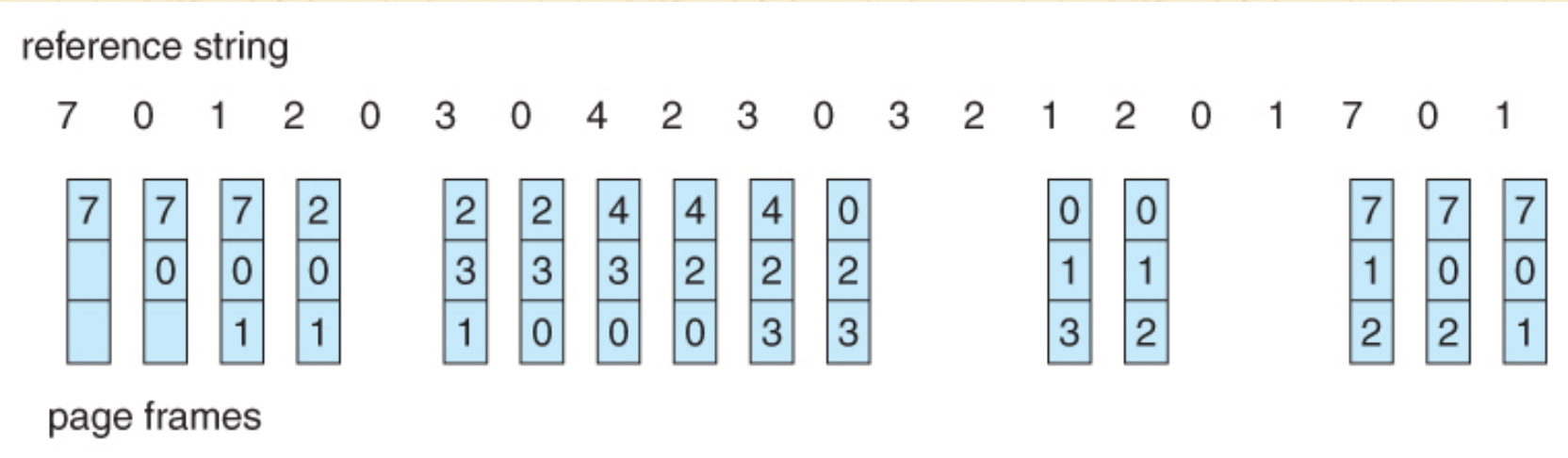


Example of FIFO Page Replacement with 4 frames. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Page Replacement Algorithms

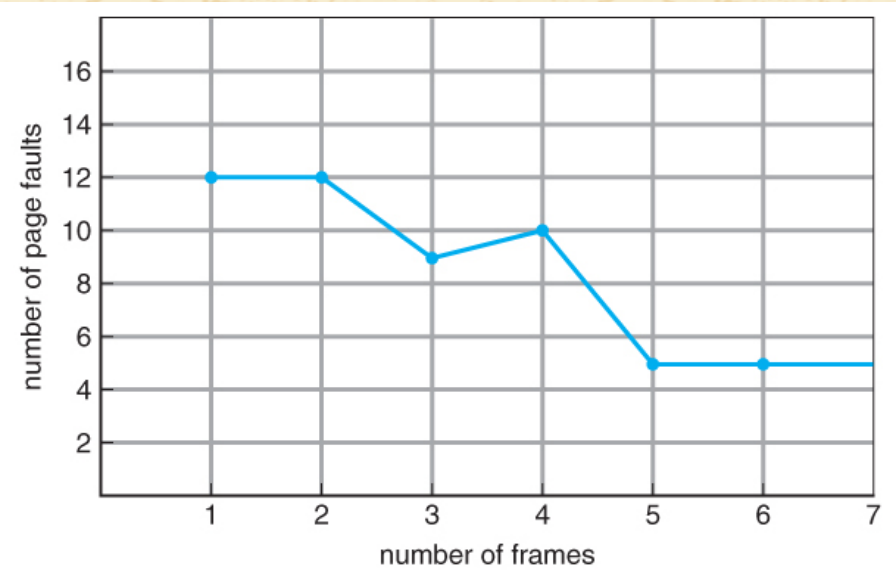
Here is another example of FIFO, this time with 3 available frames:



FIFO page-replacement algorithm. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement Algorithms

If the reference string is: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 and we use FIFO, we see an example of Belady's anomaly if we were to switch from using 3 frames to 4 frames, as there will be an increase in the page faults:

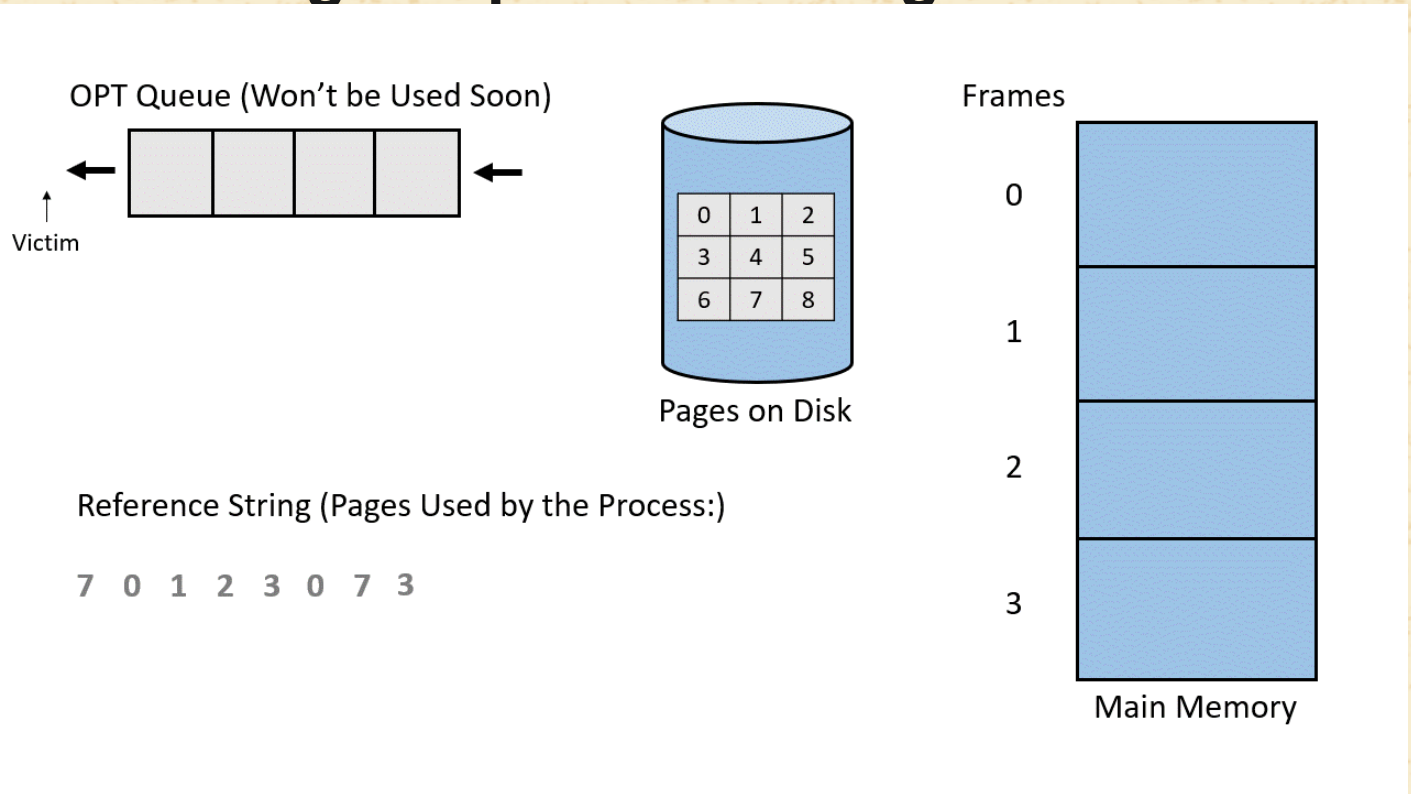


Page-fault curve for FIFO replacement on a reference string. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement Algorithms

2. **Optimal Page Replacement**.
   - It is called **OPT** or **MIN** for short.
   - Such an algorithm will yield the lowest possible number of page faults and won't suffer from Belady's anomaly.
   - It turns out that such an algorithm exists: it will simply replace the page that will not be used for the longest time in the future.
   - The only problem in implementing such an algorithm is that we simple can't see the future!
   - In practice, most page-replacement algorithms try to approximate Optimal Page Replacement by predicting (estimating) in one fashion or another what page will not be used for the longest period of time.
   - This algorithm doesn't suffer from Belady's anomaly.

Page Replacement Algorithms

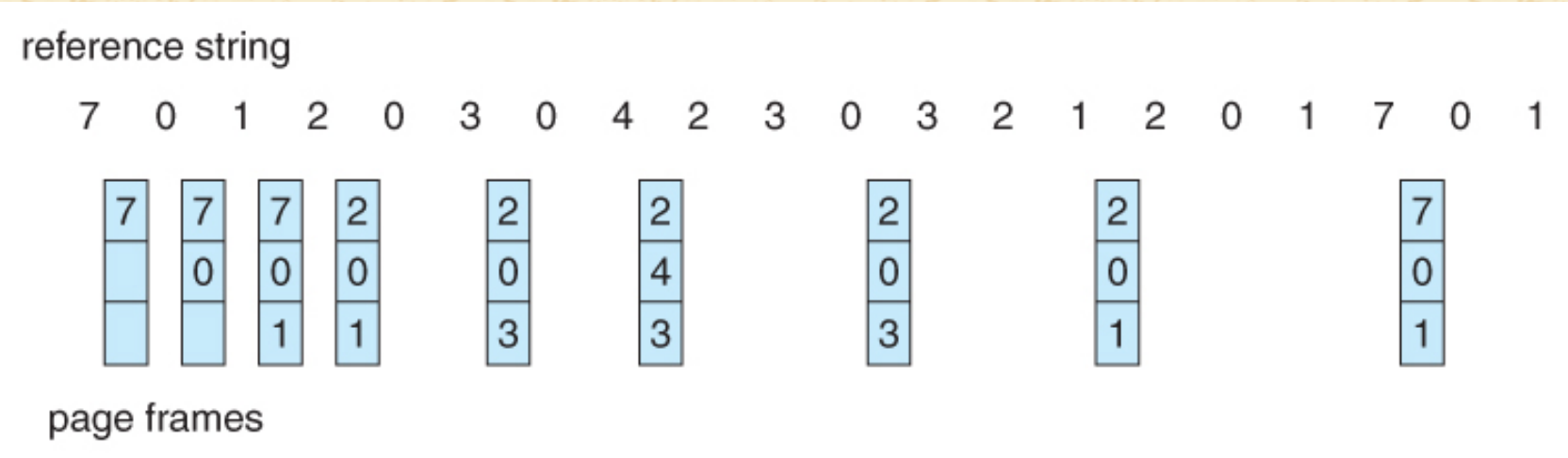


Example of OPT Page Replacement with 4 frames. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Page Replacement Algorithms

Here is another example of OPT with 3 available frames. Notice the considerable reduction in the number of page faults:

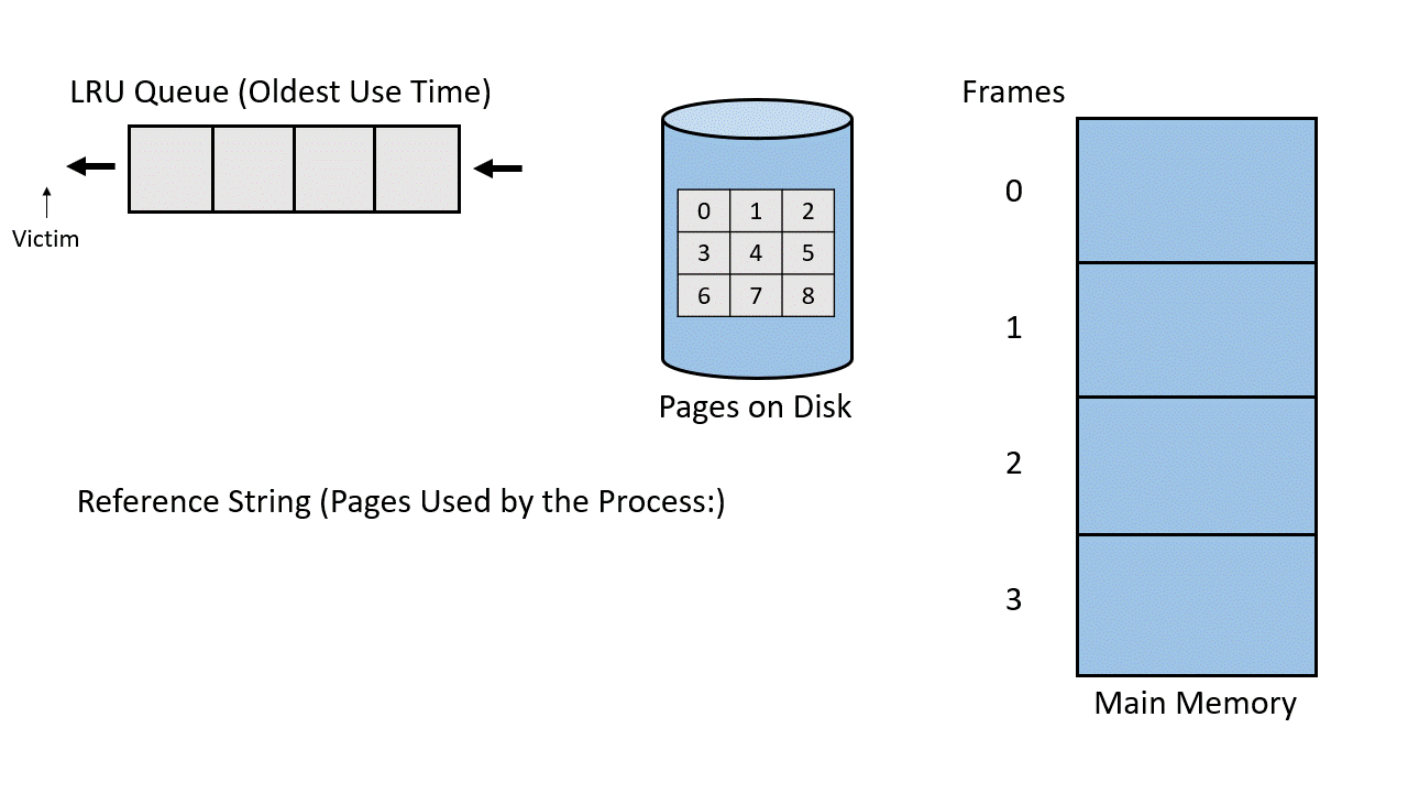


Optimal page-replacement algorithm. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement Algorithms

3. **LRU (Least Recently Used) Page Replacement**.
   - Here, we will replace the page that wasn't used by the process for the longest time duration.
   - Note the distinction between FIFO and LRU: The former looks at the oldest **load** time, and the latter looks at the oldest **use** time.
   - LRU is considered a good replacement policy, and is often used.
   - To implement the algorithm, we use either **counters** (1 counter per page) that tell how much time passed since a page was used, or use a **stack** to pull a page from the middle of the stack and place it on the top when the page is used, and replace those pages located at the bottom of the stack.
   - LRU doesn't suffer from Belady's anomaly as well.

Page Replacement Algorithms

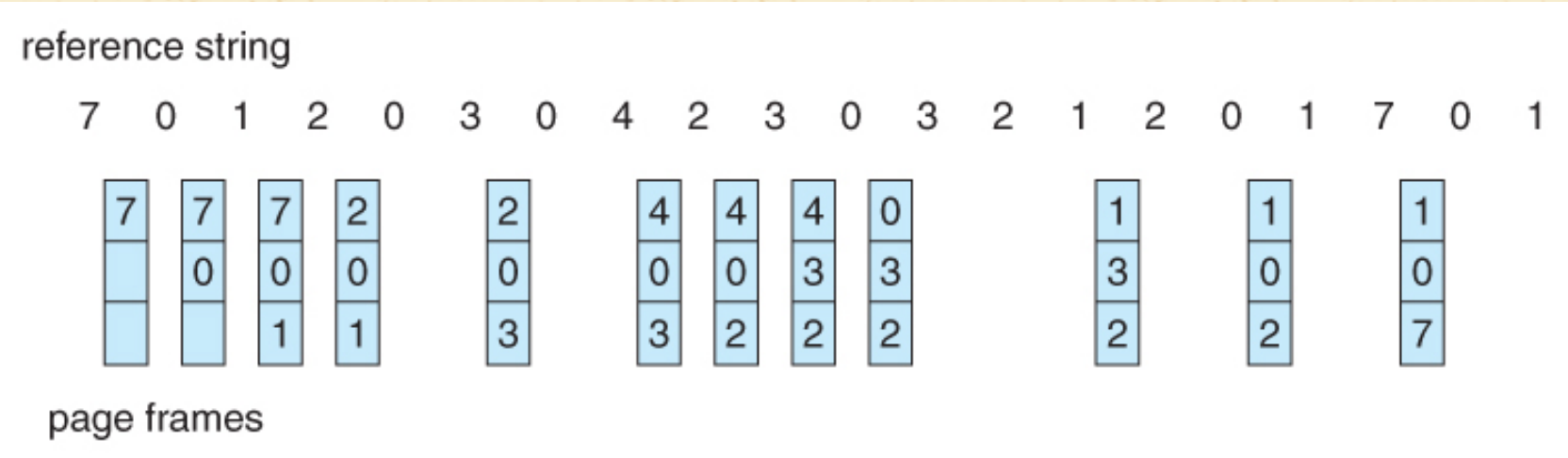


Example of LRU Page Replacement with 4 frames. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Page Replacement Algorithms

Here is another example of LRU with 3 available frames. There are fewer page faults than in FIFO:



LRU page-replacement algorithm. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

Page Replacement Algorithms

Any questions?

Frame Allocation

How does the OS decide how many frames to initially give to each process?

This depends on the **frame allocation** policy of the particular operating system.

- The smallest number of frames that the OS must allocate to a process dependents on system architecture.
  - Usually, this number will be the largest number of pages that could be accessed by a single instruction.
  - When an instruction uses indirect access (pointer to a variable), several pages might be accessed.
- Algorithms that the OS could use to decide how to allocate frames are:
  1. **Equal Allocation**: If there are *m* frames available and *n* processes to share them, each process gets *m* / *n* frames, and the leftovers are kept in a free-frame buffer pool.
  2. **Proportional Allocation**: Allocate the frames proportionally to the size (or prioroty) of the process, relative to the total size of all processes. So if the size of process *p* is S_p, and *S* is the sum of all S_p, then the allocation for process *p* is: ($m * S_p / S$) frames.

Thrashing

A process is **thrashing** when it is spending more time on page faults (and page replacements) than executing its code.

Cause: historically, whenever the OS notices that the CPU utilization is low (the CPU is not executing code,) it runs more processes. The problem is that when memory is filled up and processes start spending more time waiting for their pages to be loaded from disk, then CPU utilization would decrease (since the processes are waiting,) causing the scheduler to add in even more processes, which makes thrashing grow.

To prevent thrashing, we check how many frames the process needs "right now" using one of the methods below:

1. **Locality Model**: It is typical for processes to access pages located in the same general area of memory (= **locality**), so the number of needed frames could become the number of frames in the current locality.
2. **Working-Set Model**: Take note of (observe) the most recently accessed pages for some time, and make the number of frames equal to the amount of these accessed pages.

Memory-Mapped Files

Suppose that your program is accessing a file (either reading its content or adding content to it.) Each time your program accesses this file, the OS will need to access a location on disk, which takes considerable time than accessing main memory.

To shorten these access times, the OS can **memory-map** the file: it will copy the content of the file (or a part of it) to memory, so the program will access the file as a regular variable in memory (specifically, an array of characters.)

When a process writes (= makes changes) to a memory-mapped file, these changes won't be written to disk right away.

- To force the OS to back up all the recent changes to disk, the program needs to call the `sync()` system call.
- Changes to a memory-mapped file are flushed to disk also when the program closes the file via the `close()` system call. This is why it is so important to include in your programs a function call to close a file after it was used.

Some systems (e.g., Linux) provide special system calls to memory map files and use direct disk access otherwise.

Memory-Mapped Files

Any questions?

Virtual Memory: Sources

Bell, John T. "Virtual Memory." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html

Chapter 6: Virtual Memory.

Wienand, Ian (2019). Computer Science from the Bottom Up. This work is licensed under the Creative Commons Attribution-ShareAlike3.0 Unported License. URL: https://www.bottomupcs.com/csbu.pdf