1878

Western

UNIVERSITY · CANADA

Chapter 10 – Virtual Memory

Spring 2023

Overview

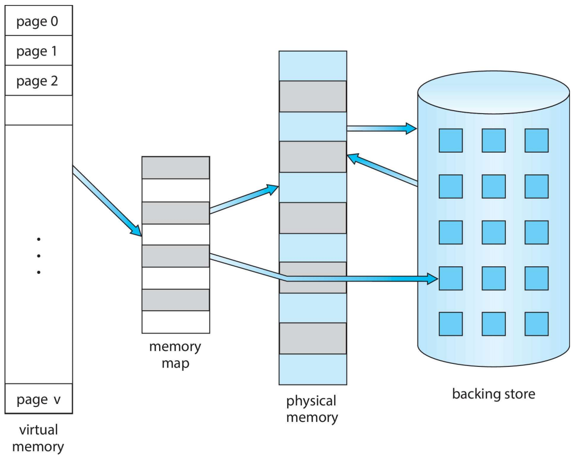
- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Other Considerations

Background

- Programs need to be in memory to execute, but not the whole program
  - Some routines are rarely (or never) used. E.g. Error routines
  - Only a portion of a large data structure might be used at any time
- Ideally, we would like to load only the portion of a program that is needed
  - We are no longer constrained by the limits of physical memory
  - Fewer programs in memory means more programs can execute

Background

- **Virtual Memory** – Separation of user logical memory from physical memory
- Chapter 9 covered logical (pages) and physical addresses (frames) but assumed the number of pages and frames were equal.
- If a system uses more pages than frames, it is using virtual memory

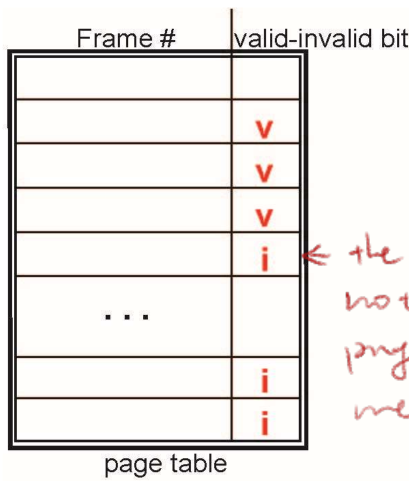


Demand Paging

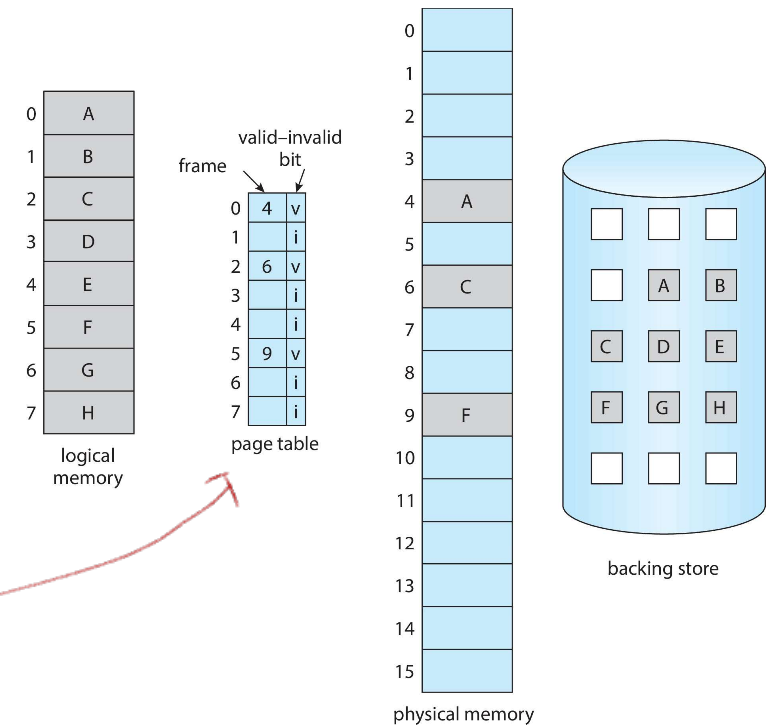
- Bring a page into memory only when it is needed ("demanded")
  - Less I/O and no unnecessary I/O
  - Less memory used
  - Faster response time
- Similar to swapping except exactly which pages are brought in is intentional
  - The MMU requires new functionality
    - Use the valid/invalid bit approach: Valid (**memory-resident**) means the page is in memory and legal. Invalid means the page is either illegal or legal but not in memory yet

Demand Paging

- Initially, every entry in the page table is set to invalid
- If a requested page is invalid, generate a **page fault**



the OS would not load this page into memory.



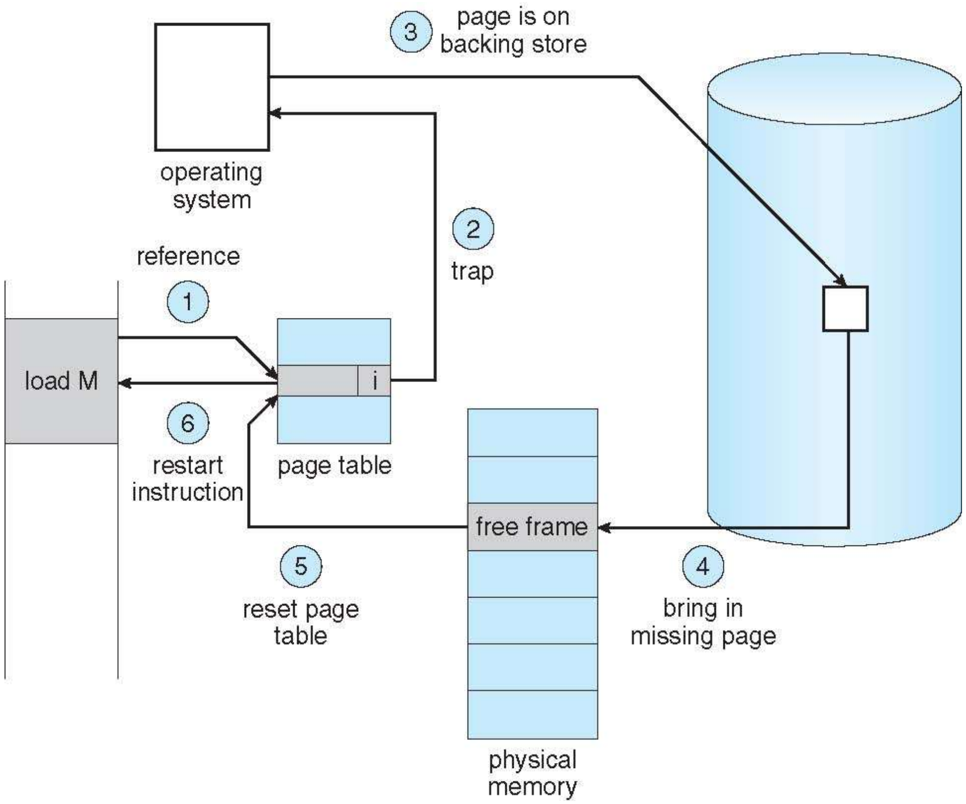
Demand Paging

- 1. If there is a reference to a page, first reference to that page will trap to operating system (Page fault)
- 2. Operating system looks at another table to decide:
  - Invalid reference → abort
  - Just not in memory
- 3. Find free frame

Demand Paging

- 4. Swap page into frame via scheduled disk operation
- 5. Reset tables to indicate page now in memory. Set validation bit = v
- 6. Restart the instruction that caused the page fault

Demand Paging



Demand Paging

- If all frames begin as invalid, the page table will eventually populate with valid pages
- In practice, this means a lot of page faults initially but eventually the process will settle

Demand Paging

- Free-Frame list
  - When a page needs to be brought into memory, the operating system needs to know where to put it
  - Operating systems maintain a list of free-frames
  - For security reasons, a frame is "zeroed-out" prior to being assigned

    set to zer
  - Which frame to choose will be discussed later

head → 7 → 97 → 15 → 126 ··· → 75

Demand Paging

- Performance
  - Page faults need to be kept to a minimum
  - Reading a page from disk can take considerable time
  - Calculate the **effective access time** similar to how we calculate the effective memory-access time
    - If $0 \leq p \leq 1$ where p is the probability of a page fault

$$\text{Effective Access Time} = (1 - p) \bullet AccessTime + (p) \bullet PageFaultTime$$

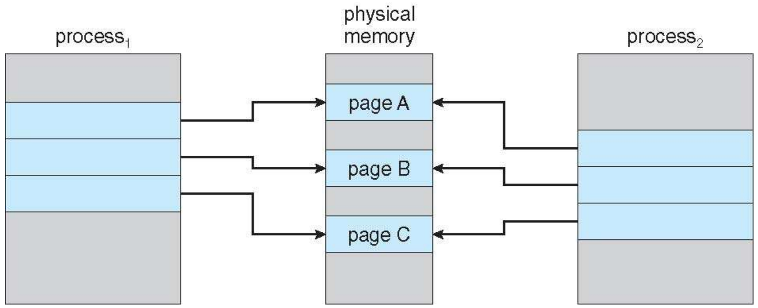
Demand Paging

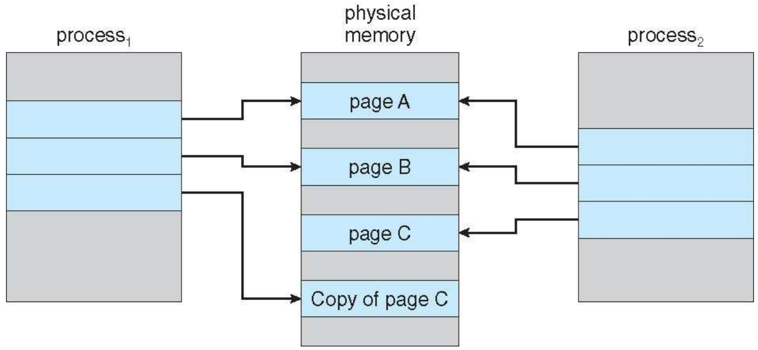
at the beginning, all page faults are invalid.

- Performance
  - Suppose a HDD has a page fault time of ~8 milliseconds
  - Suppose memory access time is 200 nanoseconds
  - Suppose a page fault in 1/1000 accesses
  - Then effective access time is 8.2 microseconds instead of 200 nanoseconds ?
- Using efficient secondary storage (like SSDs) structured to manage swap space can help
- Mobile devices typically do not use swap space

Demand Paging

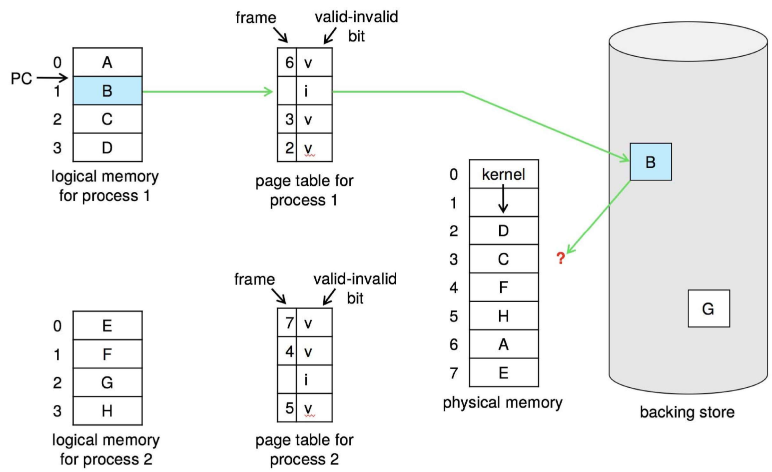
- Another way to improve performance of demand paging is to employ **copy-on-write**
- We learned the `fork()` system call copies the parent process space to the child including all pages
  - In fact, for efficiency, the pages are shared until there is an update
  - Most child processes simply call `exec()`. Not copying pages makes a lot of sense





Page Replacement

- Demand paging, as it has been discussed, ensures that a page fault occurs for a page exactly once: The first time the page is used
- Eventually, the system will run out of free frames. Some pages that are valid but no longer needed will need to be **replaced**



Page Replacement

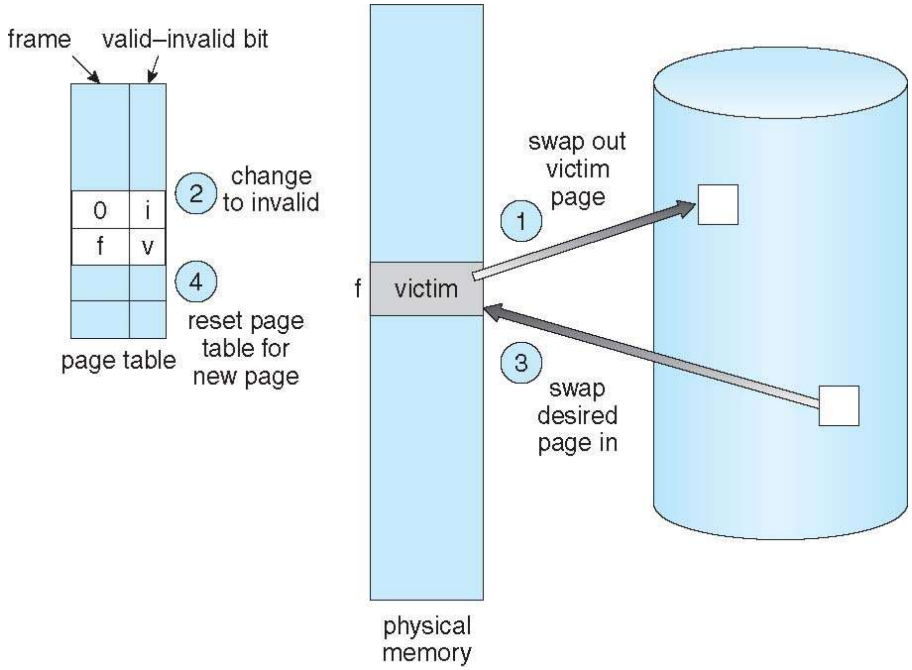
- 1. Find the location of the desired page on disk
- 2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a victim frame
  - Write victim frame to disk; update the page and frame tables
- 3. Bring the desired page into the (newly) free frame; update the page and frame tables
- 4. Continue the process by restarting the instruction that caused the trap

Page Replacement

- This involves a page-out and a page-in for every page fault
  - We can eliminate the need for a page-out by maintaining a **modify bit** (**dirty bit**)
    - If any byte in this page has been changed, set the modify bit
    - If the modify bit is set, then a page-out is required
    - If the modify bit is not set, then the page can be discarded

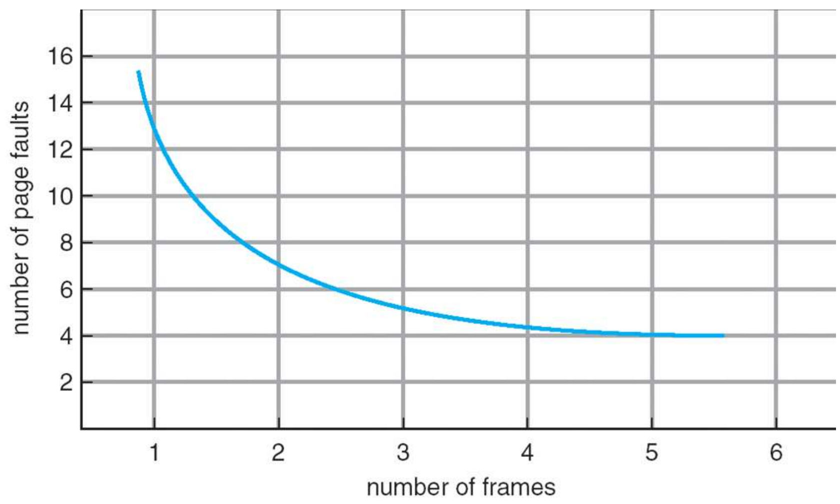
Page Replacement

- How do we select a victim frame?
  - Page replacement algorithms
    - First-in First-out (FIFO)
    - (Optimal Page Replacement)
    - Least Recently Used (LRU)
    - Counting algorithms
      - Least Frequently Used (LFU)
      - Most Frequently Used (MFU)



Page Replacement

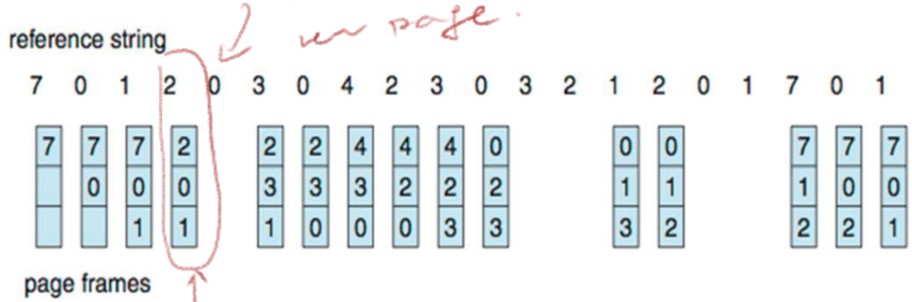
- Using a string of page numbers and calculate the number of page faults for a given number of frames
  - E.g.
    - 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
    - 3 frames
- We should expect More frames == Fewer faults



Page Replacement

- FIFO
  - 15 page faults

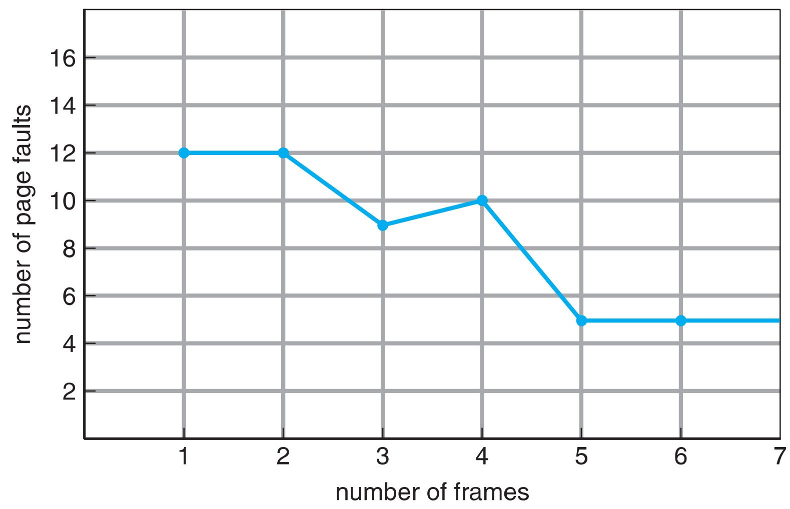
page 0 already in memory, no need to generate a new page.



pick page 7 as victim since FIFO

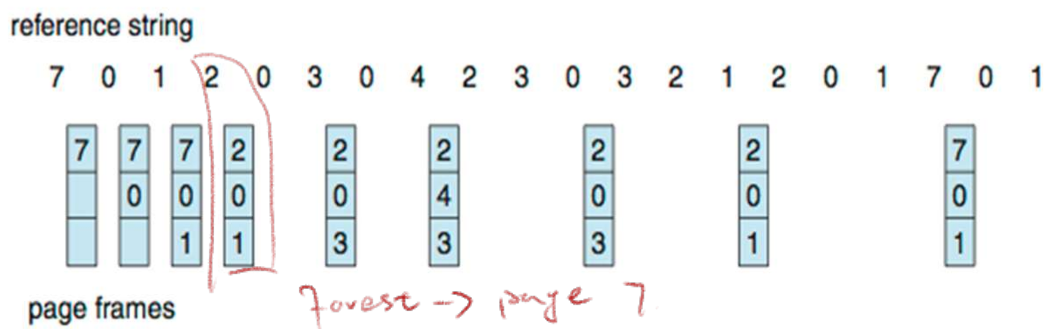
Page Replacement

- FIFO can sometimes produce poor or unexpected results
- For example, here is a graph for 1,2,3,4,1,2,5,1,2,3,4,5
  - Sometimes simply adding more memory (frames) can increase page faults
  - This is known as **Belady's anomaly**



Page Replacement

- Optimal Page Replacement
  - We can derive the theoretical optimal replacement algorithm by replacing the page that will not be used for the longest period of time
  - Just like the Shortest-Job First algorithm, this requires perfect knowledge of future values. It is used as a theoretical target to measure against
  - E.g. 9 page faults

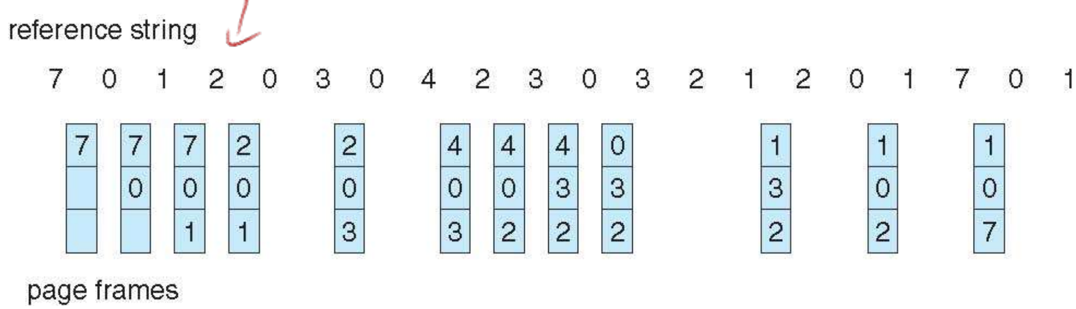


what is the difference with FIFO?

Page Replacement

- Least Recently Used
  - Use past knowledge to expire old entries
  - E.g. 12 page faults
  - Better than FIFO
  - Approaching Optimal

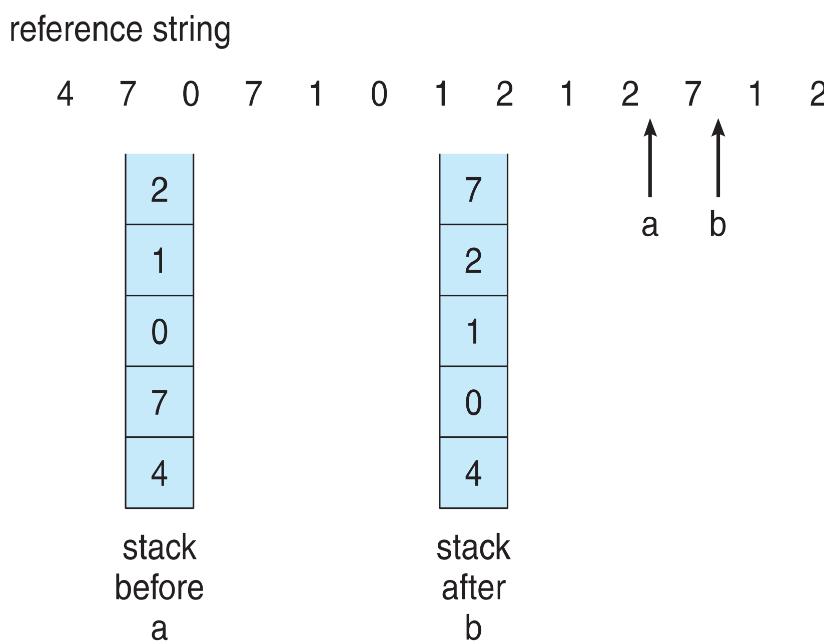
7 is the least recently used.



Page Replacement

- Least Recently Used
  - How to determine which frame was least recently used?
    - Counter
      - Update the counter whenever it is used
      - When a page needs to be replaced, search all entries for the oldest entry
    - Stack (with modifications)
      - Move the frame to the top of the stack when it is used
      - When a page needs to be replaced, use the page at the bottom of the stack

Page Replacement

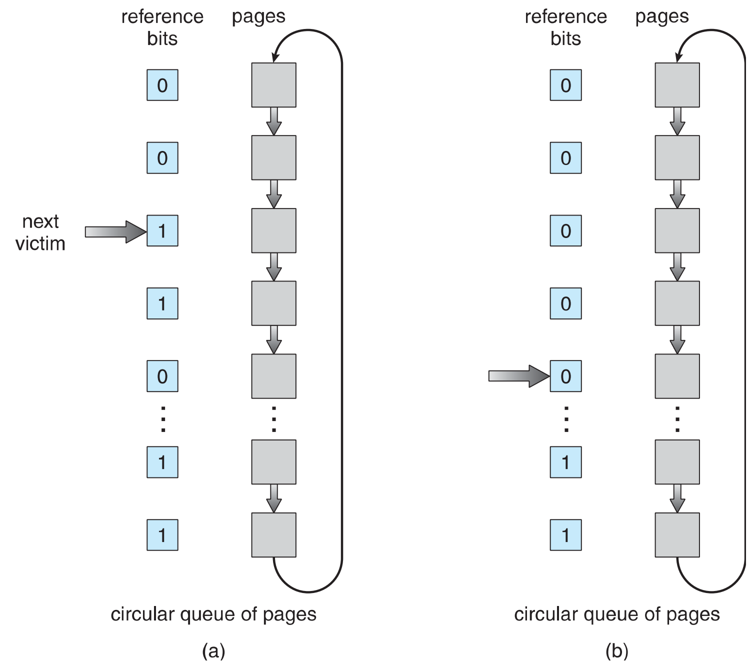


Page Replacement

- Least Recently Used
  - It might not be ideal to be so judicious about replacing pages
  - We can introduce a bit of tolerance to the replacement algorithm by introducing a **reference bit**. Set the bit to 1 when it is used
    - This could be
      - a simple 0/1 flag
      - something larger like an 8-bit number. Set the highest order bit when it is used
  - This is a **Second-Chance Algorithm**

Page Replacement

- Least Recently Used
  - If the page is a candidate for a victim page
    - check the reference bit first
    - If it is >0, set the bit to 0 (or shift the bits right)
    - Else, the page is 0 (or 0000 0000)
      - This is the victim page



diff with least recent used ?

Page Replacement

- Counting Algorithms. Not commonly used, but described for illustration purposes
  - Requires a counter for each page
  - Least Frequently Used (LFU) – Replace the page with the smallest count
    - An actively used page should have a high count and therefore should not get replaced. However, it might have a high count at program startup and never used again. This algorithm could just slowly decrease the count of unused pages over time
  - Most Frequently Used (MFU) – Replace the page with the highest count
    - The page with the smallest count definitely should not be replaced since it is likely going to be used soon

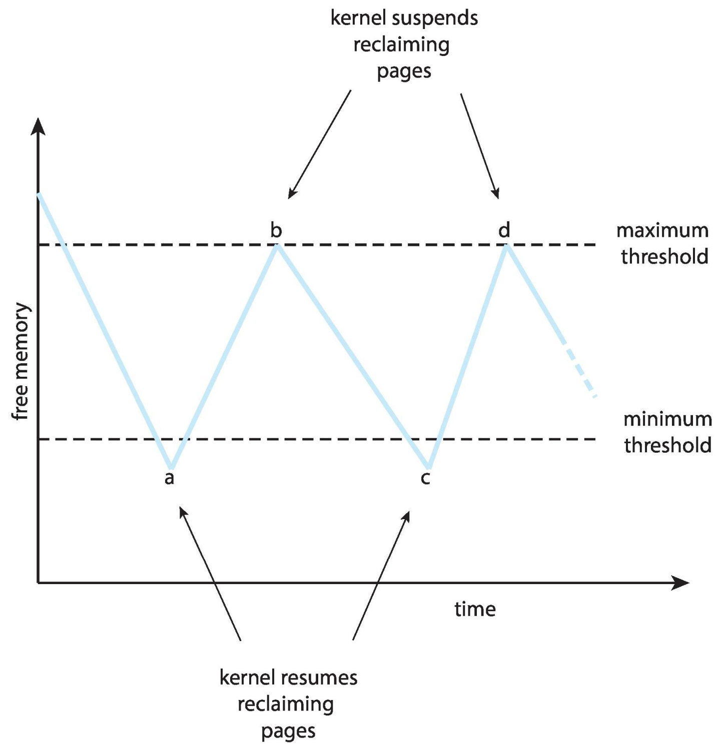
Page Replacement

- All these algorithms have operating system guessing about future page access
- Some applications have better knowledge (e.g. Databases, Data Warehouses, …)
  - These applications usually manage memory for their specific workload better than the operating system defaults
  - Operating systems can give these applications direct access to the disk, bypassing all the operating system memory management and file-system structures

Allocation of Frames

- As frames are pulled from the free-frame list, eventually we will run out and need to employ some page replacement strategies
- Since a page fault hinders performance, a page fault and forcing a page-out/page-in hinders performance even more
- Ideally, the operating system should always have a set of free-frames ready for use
- Instead of allowing the free-frame list to drop to 0, periodically search for frames that can be freed

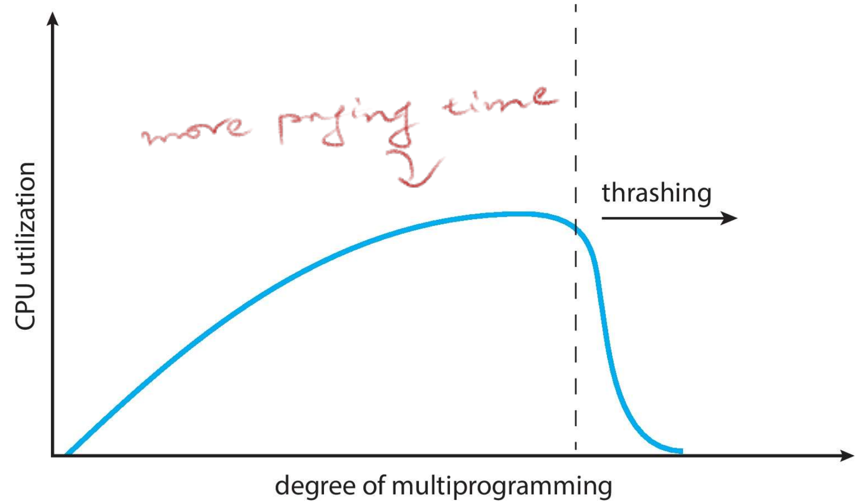
Allocation of Frames



Thrashing

- If a process needs a page in memory, it may replace a page it will need very soon
- Then a request for that page will page fault causing a recent page to be replaced
- Then a request for that recent page will page fault causing a recent page to be replaced
- If the system has too many active processes, processes may start "stealing" pages from other active processes
    - CPU utilization will drop so the operating system will think it should accept more processes creating more page faults
- **Thrashing** – When the amount of time spent paging exceeds the amount of time spent executing real work

Thrashing



- Possible Solutions
  - Force processes to use their own set of pages so they do not steal from others. Good estimates for how many pages a process will need is required.
  - Page-in the page requested <u>and</u> other pages with the same **locality** (e.g. all the instructions in the same program structure or same data structure)

Other Considerations

- More on locality and program/data structure
  - Consider the following code
    - ```
      int i,j;
      int[128][128] data;
      for (j=0; j < 128; j++){
        for (i=0; i < 128; i++){
          data[i][j] = 0;
        }
      }
      ```
  - Since C holds multidimensional arrays in row-major order, if a page can hold one row, this code could generate 128x128 (16,384) page faults due to demand paging. One for every update.
      ```

Other Considerations

- More on locality and program/data structure
  - Compare with the following code
    - ```
      int i,j;
      int[128][128] data;
      for (i=0; i < 128; i++){
        for (j=0; j < 128; j++){
          data[i][j] = 0;
        }
      }
      ```
  - This will zero each element in the page before requesting the next one. This will cause 128 page faults (~0.8% of 16384). Good programming can reduce page faults.
      ```

Other Considerations

- More on locality and program/data structure
  - Stacks have good locality since access is always at the top
  - Hash tables do not have good locality since it scatters references all over the place
  - The compiler and loader can separate read-only code from data. More read-only pages means no need to page-out.
  - The loader can avoid placing routines across page boundaries. It can combine routines that call each other together on the same page.
  - (Of course, locality is only one factor to consider. Programmers must also consider search speed, memory references, algorithmic efficiency, etc.)

Other Considerations

- Prepaging
  - In contrast to demand paging, **prepaging** some or all of a program can be used to avoid page faults at program startup, decreasing startup time.
  - Prepaging part of a program can be difficult because it is difficult to know what pages will be required
  - Prepaging an input file is easier since files are usually read sequentially.
    - Linux provides `readahead()` to prefetch a file into memory, so all subsequent reads are done in memory instead of to disk

Other Considerations

- Page Locking (Pinning)
  - The operating system may tell a disk drive which memory address to write to
    - Meanwhile, the process that asked for the data is in an I/O waiting queue
    - Another process needs a frame, but the page replacement algorithm recommends replacing the frame being written to!
    - The operating system can set a **lock bit** to ensure the page is "off-limits"
  - For efficiency, the operating system may set the lock bit for common internal processes (e.g. the memory management module itself)
  - Some privileged user processes (e.g. Databases) may also request the lock bit

1878

Western

UNIVERSITY · CANADA