Virtual Memory II [1][2]

IT Systems, Summer Term 2026

Dr. Matthes Elstermann

The topic of virtual memory is explained in two presentations, of which this is the second one.

1 Introduction

Let us start with a quiz, followed by new terminology.

1.1 Retrieval Practice

Take this quiz.

1.2 Terminology

- To page = to load a page of data into RAM
  - Managed by OS
- Paging may cause swapping and may lead to thrashing as discussed next
- Paging policies, to be discussed afterwards, aim to reduce both phenomena

In the context of OSs, to page means to load a page of data into RAM. This paging is managed by the OS.
If all frames are full, and a new page should be brought to RAM, the OS needs to swap the content of some frame: The current contents are written to disk, and the new page is loaded into that frame. If this swapping happens too frequently, a phenomenon called thrashing may occur.
With paging policies, the OS aims to reduce swapping and thrashing.

1.2.1 Swapping

- Under-specified term
  - Either (desktop OSs)
    - Usual paging in case of page faults
      - Page replacement: Swap one page out of frame to disk, another one in
      - Discussed subsequently

Swapping means different things in different contexts. In the case of desktop OSs, swapping is just the usual process of page replacement if all frames of RAM hold some data and a page fault occurs. In that case, the OS has to find a frame into which to load the new page. It selects some frame with a replacement policy, swaps out, i.e., saves its contents to disk and loads the new page into that frame.

  - Or (e.g., mainframe OSs, not considered subsequently)
    - Swap out **entire process** (all of its pages and all of its threads)
      - New state for its threads: swapped/suspended
      - No thread can run as nothing resides in RAM
      - Control so-called **multiprogramming level** (upper bound on number of **active** processes)
    - Swap in later, make process/threads runnable again

Beyond class goals, in the case of mainframe OSs, entire processes can be swapped out. In that case, all pages of a process (and its threads) are swapped out to disk, and the state of the threads is changed to swapped or suspended. In that state, threads are not considered by the scheduler any longer.
This mechanism can be used by the OS if either "too many" processes are active or if "too many" page faults occur to control the so-called multiprogramming level, i.e., the number of active processes. Once some active processes finish, the OS makes threads of suspended processes runnable again (and either swaps in some pages or does so later on in response to page faults).

[1] This PDF document is an inferior version of an OER in HTML format; free/libre Org mode source repository.
[2] Material created by Jens Lechtenbörger; see end of document for license information.

1.2.2 Thrashing

- **Permanent** swapping without progress
  - Another type of livelock
  - Time wasted with **overhead** of swapping and context switching
- Situation: Too many processes/threads, too few free frames
  - Page faults are **frequent**
    - OS **blocks** thread, performs **page replacement** via swapping
    - After context switch to different thread, again page fault
    - More swapping

If too little RAM is available to satisfy the needs of processes and threads, swapping may become so frequent that a livelock called thrashing can occur. In that case, running threads frequently produce page faults, effectively asking for more RAM, leading to blocking, context switches, and page replacement with disk I/O. The pages swapped out to satisfy the demands of one thread may in fact be needed for another thread, which then also produces a page fault, leading to more swapping.

Desktop OSs usually allow users to start as many processes and threads as they like, potentially leading to thrashing and unresponsive systems. If you are interested, see here beyond class topics for an "early" out-of-memory approach under Linux.

Agenda

In part 1, you saw basic terminology regarding virtual address spaces and the translation of virtual addresses to physical ones.

In part 2, we now look at policies to decide how and when to load pages and how to replace them when a page fault occurs and all frames are full. Conclusions end this topic.

2 Fetching Pages

Let us see how to load, or to fetch, pages to RAM.

2.1 Files and Virtual Memory

- Data kept persistently in files on secondary storage
- When (thread of) process opens file, file can be **mapped** into virtual address space

  Typical OSs offer file systems for the persistent storage of data on disks, where persistent means that (in contrast to RAM) such data remains safely in place even if the machine is powered down. Different OSs offer different system calls for file access, and this slide focuses on a technique called memory-mapped files. Here, the file is simply mapped into the virtual address space of the process containing the thread, which invokes the system call. "Mapping" means that afterwards the file's bytes are available starting at a virtual address returned by the system call.

  - Initially without loading data into RAM
    - E.g., page 2 in big picture
  - Page accesses in that file trigger **page faults**
    - Handled by OS by loading those pages into RAM
      - Marked read-only and **clean**

    Initially, no data needs to be loaded into RAM at all. If the thread now tries to access a byte belonging to the file, a page fault occurs, and the thread gets blocked. The page fault handler then triggers the transfer of the corresponding block of disk data to RAM (using metadata about the file system for address calculations). The completion of that transfer is indicated by an interrupt, in response to which the page table is updated, and the corresponding page is marked as read-only and clean, where clean indicates that the page is identical to the copy stored on disk. Also, the thread accessing the file is made runnable and can access its data.

- Upon write, MMU triggers interrupt, OS makes page writable and remembers it as **dirty** (changed from **clean**)
  - Typically with MMU hardware support via **dirty bit** in page table
  - Dirty = to be written to secondary storage at some point in time
    - After being written, marked as clean and read-only

  While reading just returns the requested data, writing triggers another interrupt as the page is marked read-only. Now, the interrupt handler marks the page as writable and dirty. Being writable implies that further write accesses succeed without further interrupts, and being dirty indicates that the version in RAM now differs from the version on disk. Thus, when a thread requests to write data back to the file, dirty pages need to be written to disk. After writing, the file's pages are marked as clean and read-only again.

2.2 Fetch Policy

- General question: When to bring pages into RAM?

- Popular alternatives

  A fetch policy determines when to bring pages into RAM. Two popular alternatives are explained on this slide.

  - **Demand paging**
    - Only load page upon **page fault**
    - Efficient use of RAM at cost of lots of page faults

    With demand paging, the OS only loads pages in response to page faults. This policy is simple to implement and makes efficient use of RAM, as only those pages are loaded which are really needed. However, this policy may lead to lots of page faults, each of which incurs some overhead. E.g., when starting a web browser, it may need several hundreds of megabytes of data and instructions.

  - **Prepaging**
    - Bring several pages into RAM, anticipate **future use**
    - If future use guessed correctly, fewer page faults result
      - Also, loading a random hard disk page into RAM involves rotational delay
      - Such delays are reduced when neighboring pages are read in one operation
      - (Even for SSDs, multiple random I/O operations are slower than a single sequential I/O operation of the same size as each operation comes with overhead)

    With prepaging, the OS predicts future needs for pages, and loads multiple pages at once. Then, fewer page faults arise.

    In addition, each individual I/O operation comes with some overhead, making it more efficient to load a range of neighboring pages at once.

2.2.1 Prepaging ideas

- **Clustered paging**, read around

  - Do not read just one page but a cluster of neighboring pages
    - Can be turned on or off in system calls

Given the overhead of loading individual pages mentioned on the previous slide, system calls may enable so-called clustered paging. In that case, the OS requests not only a single, necessary page but a cluster of neighboring pages.

- OS and program start

  - OSs may **monitor page faults**, record and use them upon next start to preload necessary data
    - Linux with readahead system call
    - Windows with Prefetching and SuperFetch

In addition to or alternatively to clustered paging, the OS may also monitor the page faults occurring when the OS starts and when programs start. At the next start, the OS can then load necessary pages before page faults occur. Here you see names under which this technique is used in Linux and Windows.

2.2.2 Working Set

- OS loads part of program into main memory
  - **Resident set**: Pages currently in main memory
  - At least current instruction (and required data) necessary in main memory

    As discussed so far, typically not all pages of a process are located in RAM. Those that are located in RAM comprise the so-called resident set. For von Neumann machines at least the currently executing instruction and its required data need to be present in RAM, and demand paging is a technique to provide that data on the fly.

- **Principle of locality**
  - Memory references typically close to each other
  - Few pages sufficient for some interval

  As data is transferred in pages, one can hope that a newly loaded page does not only contain one useful instruction or one useful byte of data but lots of them. Indeed, if you think of a typical program it is reasonable to expect that the program counter is often just incremented or changed by small amounts, e.g., in case of sequential statements, loops, or local function calls. Similarly, references to data also often touch neighboring locations in short sequence, e.g., in case of arrays or objects. This reasoning is known as principle of locality, which implies that frequently only few pages in RAM are sufficient to allow prolonged progress for a thread without page faults.

  Please take a moment to convince yourself that the principle of locality is necessary for effective caching. In other words, the transfer of some set of data from a large and slow storage area to a smaller and faster storage area, would not be effective without locality of accesses.

- **Working set**: Necessary pages for some interval
  - Aim: Keep working set in resident set
    - Replacement policies on subsequent slides

The so-called working set (for some given time interval) of a given thread is that set of pages which allows the thread to execute without page faults throughout the interval. Clearly, once in a while new pages are added to the working set, while other pages are removed since their contents are not necessary any longer. Note that the working set is a hypothetical construct, whose precise shape and evolution is unknown to the OS. However, the goal of memory management is to manage the resident set in such a way that it contains the working set (and ideally not much else). Page replacement policies, to be discussed subsequently, work towards that goal.

2.2.3 Beyond Learning Objectives: Datacenter Memory

- Main memory management at Meta: (Maruf et al. 2023)
  - Modern memory is organized in **tiers** with different characteristics (e.g., cost, size, bandwidth, latency)
    - E.g.: DRAM, NVM, low-power DRAM
    - Accessible via CXL
  - Estimate page **temperature** as criterion for transparent page placement (TPP) in specific tier
    - Page is **hot** if reuse is likely within 2 minutes, **warm** for reuse within 10 minutes, **cold** otherwise
    - Idea: Move pages between faster and slower tiers based on temperature
      - Sample hardware counters (e.g., cache misses) to estimate temperature
    - Integrated into Linux kernel

Beyond class topics, this slide cites a paper related to working set management. That paper describes datacenter memory management at Meta, supported by enhancements of the Linux kernel. Importantly, modern main memory can be organized in tiers with different characteristics, in particular, regarding speed and cost, which leads to the question of which pages to keep in what tier. Here, so-called page temperatures are derived as estimators for upcoming future use. Then, pages are moved between tiers based on their temperature, to keep hot, heavily used pages in fast memory, while colder pages are assigned to slower tiers.

3 Replacing Pages

Page replacement policies determine what frame to use when a page fault occurs, while all frames are full.

3.1 Sample Replacement Policies

We consider the basic page replacement policies shown on this slide.

- OPT: Hypothetical **optimal** replacement
  - Replace page that has its next use furthest in the future
    - Needs knowledge about future, which is unrealistic

OPT is a hypothetical replacement policy, which is the optimal policy that avoids page misses as much as possible, but requires knowledge about the future.

It simply replaces the page that has its next use furthest in the future (maybe never).

We can use this policy as benchmark to compare its amount of page misses against those of other policies.

- FIFO: **First In, First Out** replacement
  - Replace oldest page first
    - Independent of number/distribution of page references

With first-in, first-out replacement, the oldest page is replaced. While this policy is easy to implement, it usually leads to many page misses as it does not take the use or importance of pages into account.

- LRU: **Least Recently Used** replacement
  - Replace page that has gone the longest without being accessed
    - Based on principle of locality, upcoming access unlikely

Least recently used is a popular policy replacing the page that has gone the longest without being accessed.

Based on the principle of locality, upcoming accesses for that page are less likely than for other pages.

- **Clock** (second chance)
  - Replace “unused” page
    - Use 1 bit to keep track of “recent” use

The clock policy keeps track of recent use of pages with 1 bit of information and replaces a page without recent use.

3.2 Replacement Examples

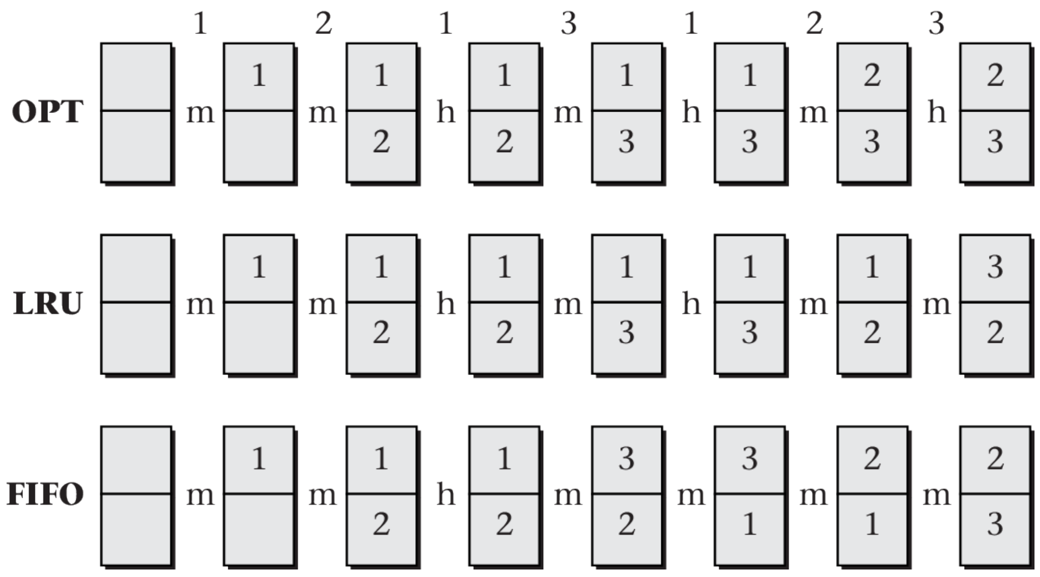


Figure 1: “Figure 6.19 of (Hailperin 2019)” by Max Hailperin under CC BY-SA 3.0; converted from GitHub

In this comparison of the OPT, LRU, and FIFO replacement policies, each pair of boxes represents the two frames available on an unrealistically small system. Numbers inside boxes are page numbers, showing which page is present in what frame.

The numbers across the top are the page reference sequence, which indicates what pages are accessed, or referenced, in what sequence. Here, page 1, then 2, then 1 again, then 3 etc.

The letters h and m indicate page hits and misses.

In this example, LRU performs better than FIFO, in that it has one more page hit. OPT performs best, with three hits.

Regarding OPT, note how all frames are full when page 3 is referenced for the first time. In this situation, either page 1 or page 2 needs to be replaced. As page 1 will be referenced sooner, it is kept, and page 2 is replaced with page 3.

Regarding LRU, the OS keeps track of page references, conceptually in a queue, where the replacement candidate is located at the front, and upon reference a page moves to the end. E.g., when page 3 is referenced for the first time, page 2 is located at the front and is therefore replaced, leaving page 1 at the front. Then, when page 1 is referenced, it moves to the back, which is why page 3 is replaced by page 2 afterwards.

Regarding FIFO, the oldest page is replaced. E.g., when page 3 is referenced for the first time, page 1 is the oldest page and is replaced. Afterwards, page 2 is the oldest page, to be replaced at the next page miss.

3.3 Clock (Second Chance)

- Frames arranged in cycle, **pointer** to next frame

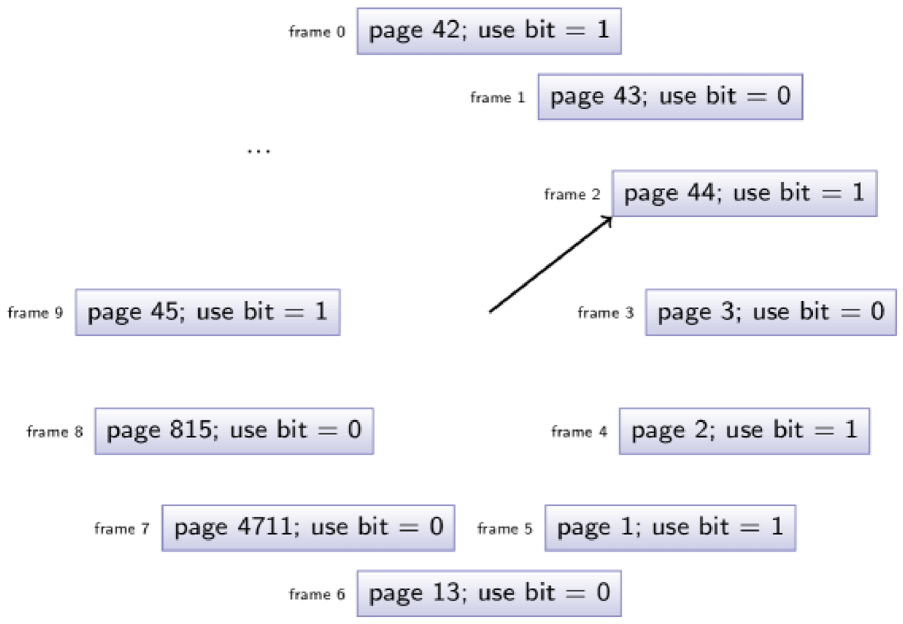


Figure 2: Clock algorithm for page replacement

  - (Naming: Pointer as hand of **clock**)
  - Pointer “wraps around” from “last” frame to “first” one
- **Use-bit** per frame
  - Set to 1 when page referenced/used

For Clock, frames are arranged circularly, like numbers on an analog wall clock, with a hand pointing to the next replacement candidate. Each frame has a use-bit, which is set to 1 when a page is referenced.

3.3.1 Beware of the Use Bit

- Use-bit may be part of hardware support
- Use-bit set to 0 when page swapped in
- Under demand paging, use-bit changed to 1 due to reference
  - Following examples assume that page is referenced for use
  - Thus, use-bit is 1 for new pages
- Under prepaging, use-bit may stay 0

The use-bit may be supported in hardware. In any case, when a page is swapped in, the use-bit is set to 0. Once the page is referenced, its use-bit is set to 1.

Note that under demand paging the use-bit is changed to 1, as pages are only loaded to be referenced. Also, for examples with pen and paper, new pages are swapped in because they are referenced. Therefore, subsequently, the use-bit is 1 for new pages.

Under prepaging, the use-bit might stay 0 for longer periods of time.

3.3.2 Clock (Second Chance): Algorithm

- If **page hit**
  - Set use-bit to 1
  - Keep pointer unchanged

  The Clock algorithm is specified here. Note that Clock is a replacement algorithm. Thus, it does not need to do much in case of page hits as no replacement is necessary. Thus, only the use-bit for the frame of the referenced page is set to 1.

- If **page miss**
  - Check frame at pointer
  - If free, use immediately, advance pointer
  - Otherwise
    - If use-bit is 0, then **replace**; advance pointer
    - If use-bit is 1, reset bit to 0, advance pointer, repeat (Go to “Check frame at pointer”)
    - (Naming: In contrast to FIFO, page gets a **second chance**)

  In case of a page miss, a new page needs to be loaded. To determine the frame, the algorithm checks the frame at the pointer. If that frame is free, it is used, and the pointer is advanced.

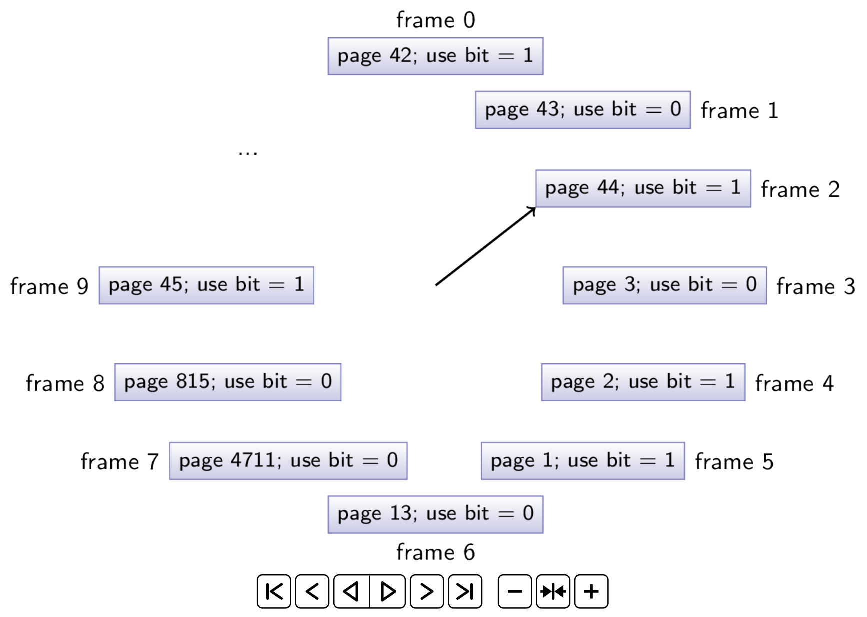
  If that frame is not free, the use-bit is checked: If the use-bit is 0, then this frame is replaced, and the pointer is advanced.

  If the use-bit is 1, it is reset to 0, the pointer is advanced, and the next frame is checked in the same manner until a frame with a use-bit of 0 is found.

  As an aside, the Clock algorithm is also called Second Chance, as the use-bit offers each referenced page a second chance compared to FIFO: While FIFO immediately replaces the oldest page, with Clock a referenced page gets a chance not to be replaced.

3.3.3 Clock (Second Chance): Animation

- Consider reference of page 7 in previous situation
  - All frames full, page 7 not present in RAM
  - Page miss



  - Frame at pointer is 2, page 44 has use bit of 1
    - Reset use bit to 0, advance pointer to frame 3
  - Frame at pointer is 3, page 3 has use bit of 0
    - Replace page 3 with 7, set use bit to 1 due to reference
    - Advance pointer to frame 4

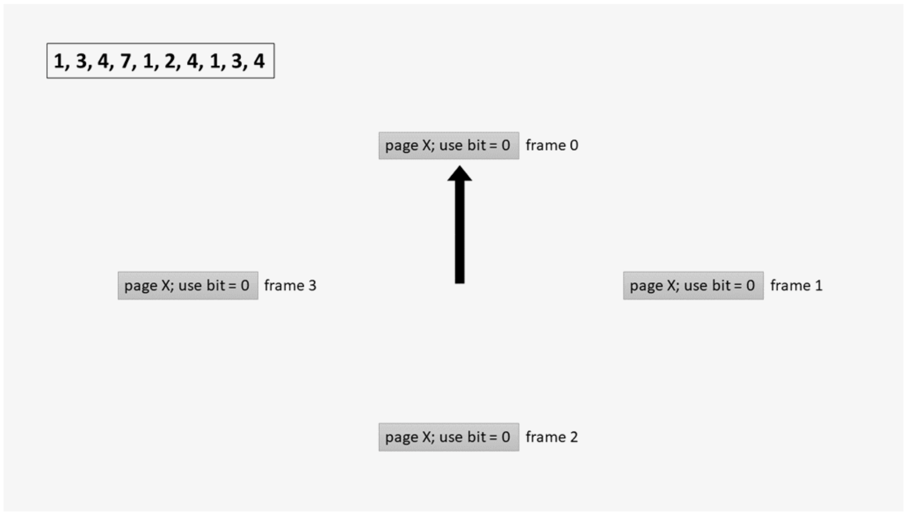
This animation shows the sequence of events in response of the reference of page 7 in the previous situation, namely a page miss when all frames are full:

The candidate frame at the pointer is 2. It contains page 44 and has a use-bit of 1. Thus, the use-bit is reset to 0, and the pointer is advanced to frame 3. In other words, page 44 used its second chance and stays in RAM.

Frame 3 contains page 3 with a use-bit of 0. Thus, page 3 is replaced with page 7. As page 7 is referenced, its use-bit is set to 1, and the pointer is advanced to frame 4 as next candidate frame for replacement.

3.3.4 Clock: Different Animation

- Situation
  - Four frames of main memory, initially empty
  - Page references: 1, 3, 4, 7, 1, 2, 4, 1, 3, 4



This animation was contributed by a student. It shows how Clock operates on four frames for the given page references.

3.4 More Replacement Examples

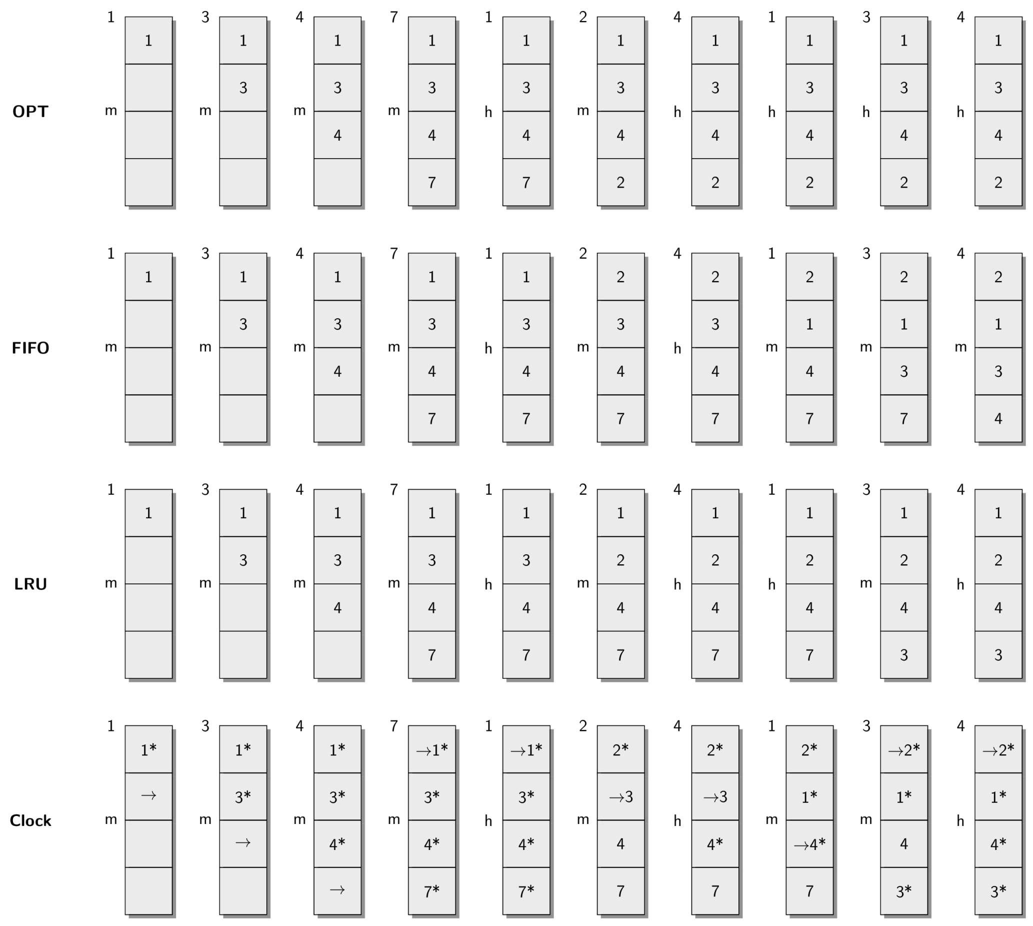


Figure 3: Example for page replacement with OPT, LRU, FIFO, Clock

The layout of this diagram mirrors an earlier one, but is extended to four frames. For Clock, demand paging is assumed; the arrow shows the pointer position, and a star indicates a use-bit of 1.

Let us see how Clock works. Consider the sixth page reference, which is supposed to bring page 2 into RAM under the situation where all frames are full, all use-bits are 1, and the pointer is at frame 0, where page 1 has a use bit of 1.

Following Clock's steps, the use-bit of page 1 is reset to 0, and the pointer is advanced to frame 1. Page 3 in frame 1 has a use-bit of 1, which is reset to 0, and the pointer is advanced. That way all use-bits are reset, before the pointer points to page 1 in frame 0 again. This time, the use-bit is 0, hence the contents of frame 0 are replaced with page 2, and the pointer is advanced once more. As we consider demand paging, an access into page 2 occurs, which sets the use-bit to 1.

3.5 Self-Study Task

- Perform the following task in Learnweb.

  Apply the page replacement algorithms OPT, FIFO, LRU, and Clock (Second Chance) for four frames of main memory to the following stream of page references under demand paging: 1, 3, 4, 7, 1, 2, 4, 1, 3, 4 Verify your results against the previous slide and raise any questions that you may have.

Perform page replacement as instructed here.

4 Conclusions

Let us conclude.

4.1 Challenge: Page Table Sizes

- E.g., 32-bit addresses with page size of 4 KiB (2^{12} B)
  - Virtual address space consists of up to 2^{32} B = 4 GiB = 2^{20} pages
    - Every page with entry in page table
    - If 4 bytes per entry, then **4 MiB** (2^{22} B) per page table

      Beyond learning objectives, let us briefly think about the size of page tables. As stated here, a virtual address space for 32 bit addresses consists of about a million pages, each of which needs an entry in the page table. Assuming only 4 bytes per entry, the page table has a size of 4 MiB, which requires about 1000 frames itself.
      - Page table itself needs 2^{10} pages/frames! **Per process**!

        Also recall that a page table is necessary per process! Thus, a considerable amount of RAM is allocated to page tables, as overhead for virtual memory management.
  - Much worse for 64-bit addresses
    - E.g., 48 bits, 4 KiB pages, 8 B per entry: 2^{36} * 8 B = 512 GiB

    Although this result is already pretty bad, for 64-bit systems the situation is much worse, even if current PC processors do not use all 64 bits for addressing. Suppose 48 bits are used for virtual addresses, again with 4 KiB pages. Then, 2^{36} pages may exist per process, now maybe with 8 B per entry in the page table, leading to 512 GiB for a single page table. Quite likely, this is more than your total amount of RAM.

- Two approaches to reduce page table sizes
  1. Multilevel (or **hierarchical**) page tables
     - Tree-like structure, efficiently representing large unused areas
     - Root, called **page directory**
       - Entries cover larger address space portions
  2. Inverted page tables

  Solutions to reduce the amount of RAM for page tables fall into two classes, namely multilevel page tables and inverted page tables.

  The key idea of multilevel page tables is that large portions of the theoretically possible virtual address space remain unused, and such unused portions do not need to be represented in the page table. To efficiently represent smaller (used) and larger (unused) portions, the page table is represented and traversed as a tree-like structure with multiple levels. The root of that tree-like structure is always located in RAM and is called page directory. Each entry in that page directory represents a large portion of the address space. If such a portion is not used at all, no data needs to be allocated in lower levels of the tree-like structure.

  The key idea of inverted page tables is that RAM is limited and typically smaller than the virtual address space. Instead of storing each allocated frame per page as discussed so far, with inverted page tables one entry exists per frame of RAM, recording what page of what process is currently located in that frame (if any).

  Note that only one such inverted page table needs to be maintained, whereas page tables exist per process.

  Also note that the number of entries of the inverted table is determined by the number of frames in RAM, instead of the (potentially much larger) number of pages of the virtual address space.

  Address translation with inverted page tables then makes use of hashing to locate the appropriate entry for a given page number.

4.2 SIEVE for Caching

- Page replacement policies also work for caching
  - E.g., for web contents or key-value stores
  - LRU is highly popular
- (Zhang et al. 2024) presents SIEVE as variant of Clock for caches
  - Web page with visualization
    - Also with use-bit ("Visited") and hand/pointer
    - Does not replace candidate, instead:
      - Remove candidate from cache
      - Insert new contents at head of queue
  - Superior performance demonstrated
  - Adopted in several systems and cache libraries

The details of this page are beyond learning objectives.

Students sometimes ask why they should learn about simple page replacement algorithms, which they perceive to be outdated.

First, the algorithms presented here are not outdated. They are highly popular in practice, sometimes only with minor modifications.

Second, and somewhat surprisingly, in 2024 the paper cited here was published. Here, the authors propose a minor modification of Clock, called SIEVE, to be used for cache management. Notably, caching is a highly relevant topic given its performance improvements not only in the memory hierarchy of individual computers, but also for large-scale distributed systems, to be revisited in the context of cloud computing.

Briefly, SIEVE is quite similar to clock with a use-bit and a pointer into a queue. However, it does not replace a candidate. Instead, it removes the candidate from the cache and then inserts new contents at the head of the queue.

The authors report a superior performance for SIEVE compared to a variety of competitors.

In the meantime, SIEVE has been adopted in several systems and cache libraries. Check out the web page hyperlinked on the slide if you are interested in details.

4.3 Summary

- Virtual memory provides abstraction over RAM and secondary storage
  - Isolation of processes
  - Paging as fundamental mechanism for flexibility
- Page tables managed by OS
  - Hardware support via MMU with TLB
  - Management of "necessary" pages is complex
    - Tasks include prepaging and page replacement

Virtual memory is a crucial feature in modern operating systems, which provides an abstraction layer over both RAM and secondary storage, such as hard disks or solid-state drives. Based on this abstraction, the OS isolates different processes from each other: They are unable to access or modify each other's code and data structures.

In addition, this abstraction enables programs to work with large amounts of data, while only utilizing a fraction of their physical memory resources at any given time. Also, treating pages as individual units of memory management offers considerable flexibility when allocating RAM to different processes.

To manage address translation from virtual addresses to physical addresses based on page tables efficiently, the OS relies on hardware support from specialized components like the MMU and the TLB.

However, managing the working sets of necessary pages for all processes is a complex task, which involves techniques such as prepaging and page replacement.

Bibliography

Maruf, Hasan Al, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. "Tpp: Transparent Page Placement for Cxl-Enabled Tiered-Memory." In *Proceedings of the 28th Acm International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 742–55. Asplos 2023. https://doi.org/10.1145/3582016.3582063.

Zhang, Yazhuo, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. 2024. "SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches." In *21St Usenix Symposium on Networked Systems Design and Implementation (Nsdi 24)*, 1229–46.

The bibliography contains references used in this presentation.

License Information