OS09: Virtual Memory II *

Based on Chapter 6 of [Hai19]

Jens Lechtenbörger

Computer Structures and Operating Systems 2023

1 Introduction

1.1 OS Plan

- OS Overview (Wk 20)
- OS Introduction (Wk 21)
- Interrupts and I/O (Wk 21)
- Threads (Wk 23)
- Thread Scheduling (Wk 24)
- Mutual Exclusion (MX) (Wk 25)
- MX in Java (Wk 25)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- Processes (Wk 27)
- Security (Wk 28)

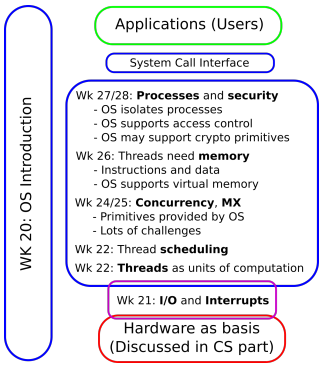

Figure 1: OS course plan, summer 2022

1.2 Today's Core Questions

- How can the size of page tables be reduced?
- How can address translation be sped up?
- How does the OS allocate frames to processes?

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

1.3 Learning Objectives

- Explain paging, swapping, and thrashing
- Discuss differences of different types of page tables
- Explain role of TLB in address translation
- Apply page replacement with FIFO, LRU, Clock

1.4 Retrieval Practice

1.4.1 Recall: Hash Tables

- Hash table = data structure with search in O(1) on average
  - Taught in Data Structures and Algorithms
- What are hash collisions, buckets, chaining?

1.4.2 Previously on OS ...

- What is a virtual address, how is it related to page tables?
  - What piece of hardware is responsible for address translation?
- How large are page tables? How many exist?
- What happens upon page misses?
- What is demand loading?
- The size of page tables poses a challenge.

1.4.3 Selected Questions

Table of Contents

2 Multilevel Page Tables

2.1 Core Idea

- So far: Virtual address is hierarchical object consisting of page number and offset
- Now **multilevel** page tables: Interpret page table as tree with fixed depth, i.e., a fixed number of multiple levels
  - (Visualizations on next two slides)
  - For n levels, **split** page number into n smaller parts
    - **Two-level** for 32 bits: Split 20 bits into **two** parts with 10 bits each
  - To traverse page table (tree), use one part on each level
- Aside: On 64-bit machines, Linux introduced 5-level tables as default on 2019-09-16

2.2 Two-Level Page Table

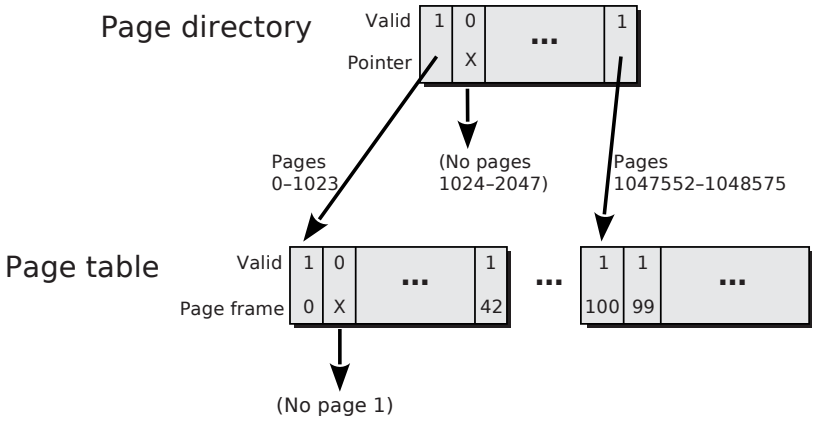

Figure 2: "IA-32 two-level page table" by Jens Lechtenbörger under CC BY-SA 4.0; Frame numbers and valid bits added to and third layer removed from Figure 6.13 of [Hai17] by Max Hailperin under CC BY-SA 3.0. Source at GitLab.

Note: *Page table* contains entries of an ordinary page table. Previously, valid bit and page frame numbers were shown in columns; here, they are shown in rows.

This figure shows a two-level page table as used in Intel's 32-bit processor architecture IA-32. The entry point to this two-level page table is called page directory and can point to 1024 chunks of the page table, each of which can point to 1024 page frames. Note that with 1024 entries of 4 B each, the page directory as well as chunks of the page table fit exactly into pages and frames of 4 KiB. The leftmost pointer leading from the leftmost chunk of the page table points to the frame holding page 0. Each entry can also be marked invalid, indicated by an X in this diagram. For example, the second entry in the first chunk of the page table is invalid, showing that no frame holds page 1. The same principle applies at the page directory level as well; in this example, no frames hold pages 1024-2047, so the second page directory entry is marked invalid.

2.2.1 Two-Level Address Translation

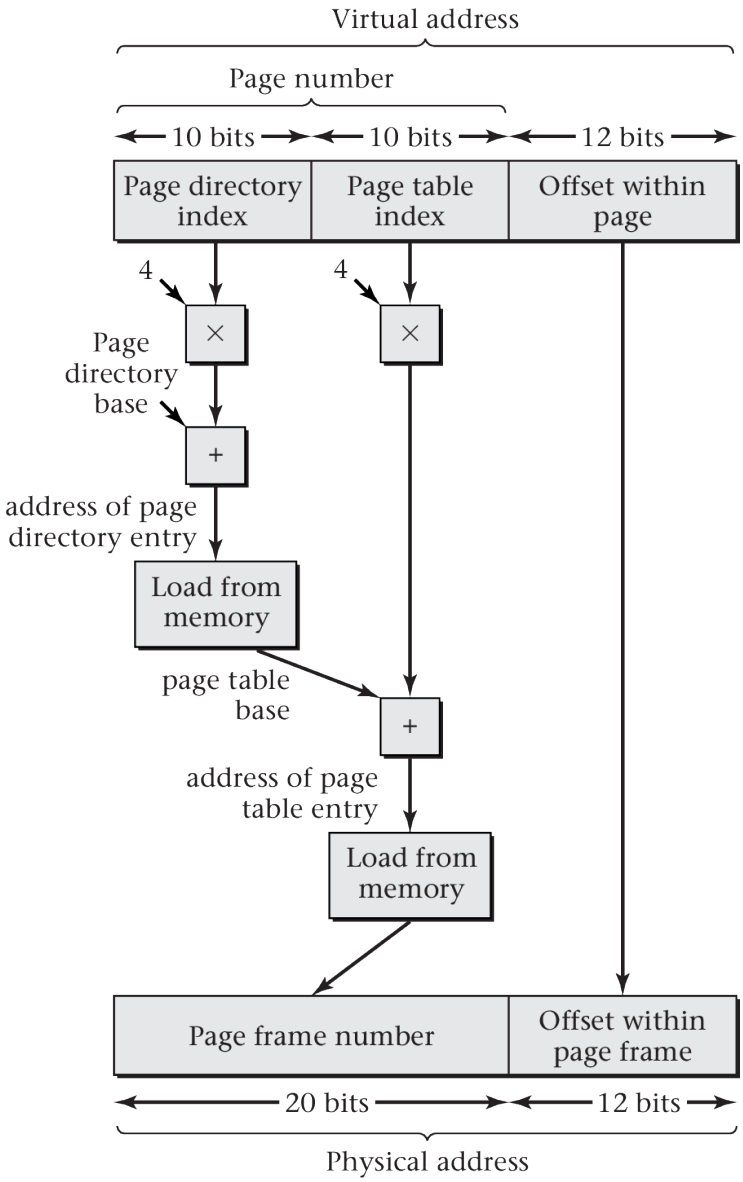

Figure 3: "Figure 6.14 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

This diagram shows the core of IA-32 paged address mapping. As explained previously, virtual addresses are understood as hierarchical objects which are divided into a 20-bit page number and 12-bit offset within the page; the latter 12 bits are left unchanged by the translation process. Due to the two-level nature of the page table, the 20-bit page number is subdivided into a 10-bit page directory index and a 10-bit page table index. Each index is multiplied by 4, the number of bytes in each entry, and then added to the base physical address of the corresponding data structure, producing a physical memory address from which the entry is loaded. The base address of the page directory comes from a special register, whereas the

base address of the page table comes from the page directory entry.

3 Inverted Page Tables and Hardware Support

3.1 Inverted Page Tables

- Recall: Page tables can be huge, **per process**
- Key insights to reduce amount of memory:
  - Number of frames is (usually) **much** smaller than aggregate number of pages
  - Thus, let us record information **per frame**, not per page and process
    - (For each frame, what page of what process is currently contained?)
- Obtain frame for page via **hashing** of page number
  - PowerPC, UltraSPARC, IA-64

3.1.1 Example

- Simplistic example, 4 frames, hashing via `modulo 4`

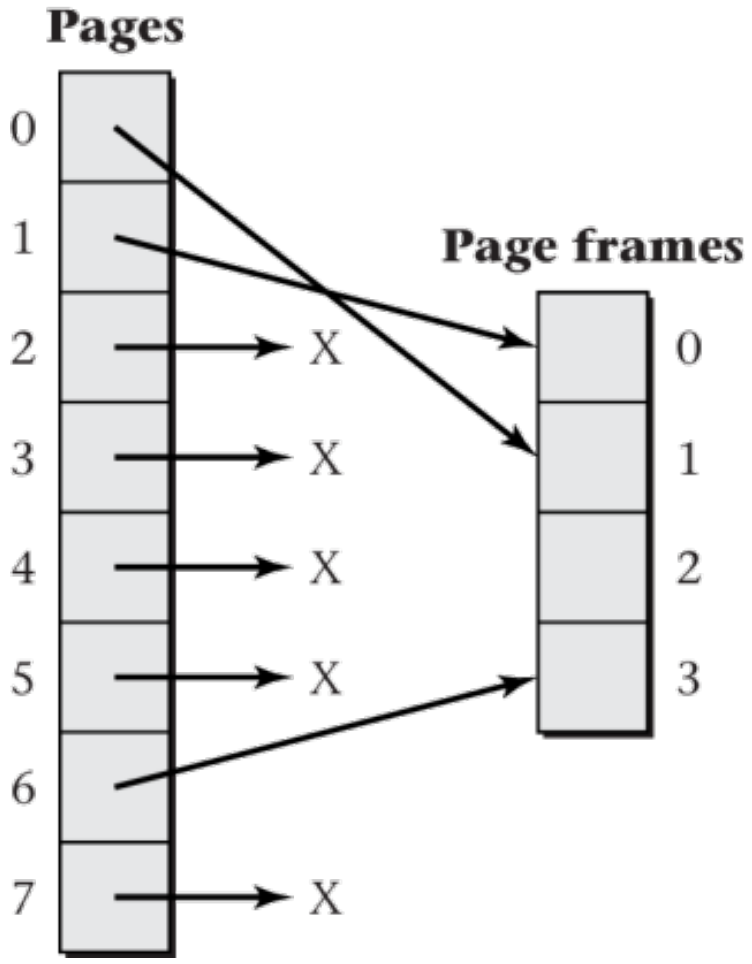

Figure 4: "Figure 6.10 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

- (Inverted page table below based on Fig. 6.15 of [Hai19]; represents main memory situation shown to the right)
- E.g., page 0: 0 mod 4 = 0; thus look into row 0, find that page 0 is contained in frame 1

Valid	Page	Process	Frame
1	0	42	1
1	1	42	0
1	6	42	3
0	X	X	X

Consider the simplified and simplistic inverted page table shown here capturing the memory situation of the process shown to the right, which is called process 42. Note that in reality, RAM would contain pages of several processes.

Here just 4 frames of RAM are available, and hashing of page number n is computed as n *modulo* 4.

When, for example, an instruction executed by the CPU on behalf of process 42 touches a virtual address located in page 0, hashing is used to compute $0 \bmod 4 = 0$, which indicates

that the first table entry needs to be accessed (as counting starts from 0). This entry shows that page 0 is located in frame 1, and the physical address can be built as usual.

As a side remark, if you read elsewhere about inverted page tables please note that you may find a slightly different scheme where the frame number is not included in table entries: If the table contains exactly one entry per frame of RAM, frame numbers can be omitted and instead entry number n would indicate the contents of frame number n. E.g., entry 2 here would not contain a frame number but directly indicate that frame 2 contains page 6 of process 42.

3.1.2 Observations

- **Constant** table size
  - Proportional to main memory size
  - Independent of number of processes
    - One entry per frame is sufficient
- Entries are large
  - Page numbers included (hash collisions)
  - Process IDs included (hash collisions)
  - Pointers for overflow handling necessary (not shown above)
  - If there is one entry per frame, the frame number does not need to be included (implicit as entry's number)
- Side note: Efficient use in practice is hard
  - See comments by Linus Torvalds if you are interested

3.2 Hardware Support for Address Translations

- Lots of architectures support page tables in hardware
  - Multilevel and/or inverted page tables
  - **Page table walker** does translation in hardware
    - Architecture specifies page table structure
      - For multilevel page tables, special register stores start address of page directory
- Special cache to reduce translation latency, the TLB (next slide)

3.2.1 Translation Lookaside Buffer (TLB)

- Access to virtual address may require **several memory accesses** → Overhead
  - Access to page table (one per level)
  - Access to data
- Improvement: Caching
  - Special cache, called **TLB**, for page table entries

        * Recently used translations of page numbers to frame numbers
    - MMU searches in TLB first to build physical address
        * Note: Search for **page**, not entire virtual address
        * If not found (TLB miss): Page table access
    - Note: Context switch may require TLB flush → Overhead
        * Reduced when entries have address space identifier (ASID)
            · See [Hai19] if you are interested in details

4 Policies

4.1 Terminology

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

4.1.1 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
- Or (e.g., mainframe OSs)
    - 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
    - Swap in later, make process/threads runnable again
    - (Not considered subsequently)

4.1.2 Thrashing

- **Permanent** swapping without progress
    - Another type of livelock
    - Time wasted with **overhead** of swapping and context switching
- Typical situation: no 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
- Reason: Too many processes/threads
    - Mainframe OSs may swap out entire processes then
        * Control so-called **multiprogramming level** (MPL)
            · Enforce upper bound on number of **active** processes
    - Desktop OSs let users deal with this

4.2 Fetch Policy

- General question: When to bring pages into RAM?
- Popular alternatives
    - **Demand paging** (contrast with demand loading)
        * Only load page upon **page fault**
        * Efficient use of RAM at cost of lots of page faults
    - **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)

4.2.1 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
- OS and program start
    - OSs may **monitor page faults**, record and use them upon next start to pre-load necessary data
        * Linux with readahead system call
        * Windows with Prefetching and SuperFetch

4.3 Replacement Policy

- What frame to re-use when a page fault occurs while all frames are full?
- Recall goal: Keep working sets in RAM
- Local vs global replacement
  - Local: Replace within frames of same process
    - When to in- or decrease resident set size?
  - Global: Replace among all frames

4.3.1 Sample Replacement Policies

- OPT: Hypothetical **optimal** replacement
  - Replace page that has its next use furthest in the future
    - Needs knowledge about future, which is unrealistic
- FIFO: **First In, First Out** replacement
  - Replace oldest page first
    - Independent of number/distribution of references
- LRU: **Least Recently Used** replacement
  - Replace page that has gone the longest without being accessed
    - Based on principle of locality, upcoming access unlikely
- **Clock** (second chance)
  - Replace "unused" page
    - Use 1 bit to keep track of "recent" use

4.3.2 Replacement Examples

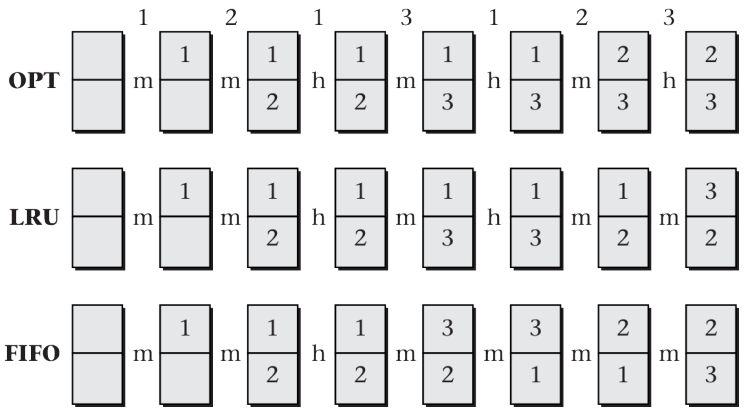

Figure 5: "Figure 6.19 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

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. The numbers within the boxes indicate which page is stored in each frame. The numbers across the top are the page reference sequence, and the letters h and m indicate hits and misses. In this example, LRU performs better than FIFO, in that it has one more hit. OPT performs even better, with three hits.

4.3.3 Clock (Second Chance)

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

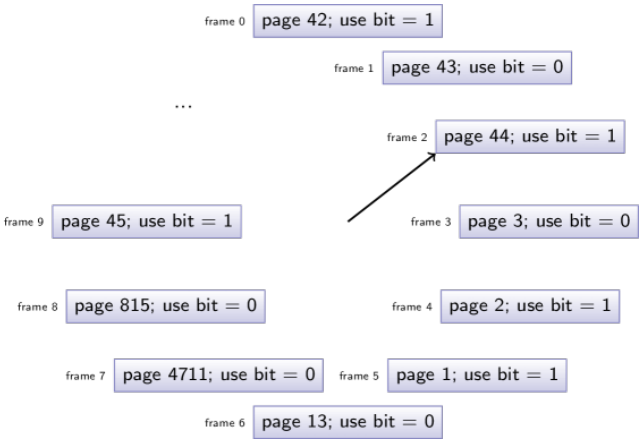

Figure 6: Clock algorithm for page replacement

  - (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

4.3.4 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 immediately set 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

4.3.5 Clock (Second Chance): Algorithm

- If **page hit**
  - Set use-bit to 1

  - Keep pointer unchanged

- If **page fault**
  - 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**)

4.3.6 Clock (Second Chance): Animation

- Consider reference of page 7 in previous situation
  - All frames full, page 7 not present in RAM
  - Page fault **Warning!** Figure omitted as gif format **not** supported in LaTeX: "Animation of Clock algorithm for page replacement"
    (See HTML presentation instead.)
    * 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

4.3.7 Clock: Different Animation

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

Warning! Figure omitted as gif format **not** supported in LaTeX: ""Page replacement example with Clock algorithm" by Christoph Ilse under CC0 1.0; from GitLab"
(See HTML presentation instead.)

4.3.8 More Replacement Examples

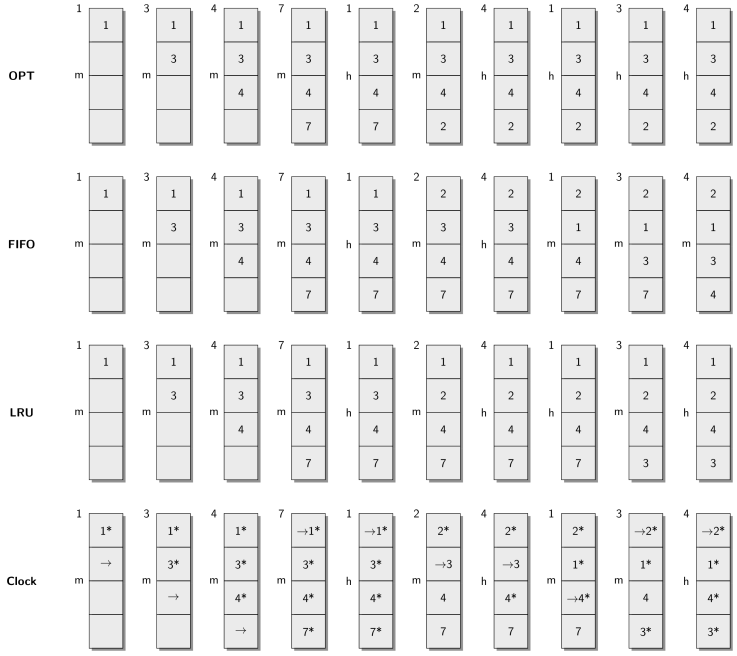

Figure 7: Example for page replacement with OPT, LRU, FIFO, Clock

The layout of this diagram mirrors the one of Fig. 6.19 but is extended to four frames. For Clock, demand paging is assumed; the arrow shows the pointer position, and "*" indicates a use-bit of 1.

Let's see how Clock works. Consider the 6th page reference, which is supposed to bring page 2 into RAM under the situation where

- all frames are full,
- all use-bits are 1,
- 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.

4.4 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.

5 Conclusions

5.1 Summary

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

Bibliography

[Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3, 2017. URL: https://gustavus.edu/mcs/max/os-book/.

[Hai19] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1, 2019. URL: https://gustavus.edu/mcs/max/os-book/.

License Information