Virtual Memory I [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 first one.

1 Introduction

Let us look at essential questions and terminology of our topic.

1.1 Core Questions

- What is virtual memory?
  - How can RAM be (de-) allocated flexibly under multitasking?
  - How does the OS keep track for each process what data resides where in RAM?
  - How does the OS manage the exchange of data between secondary storage and RAM under multitasking?

(Based on Chapter 6 of (Hailperin 2019))

This set of presentations addresses the management of virtual memory by the OS. In particular, we will see how to allocate the hardware resource RAM flexibly under multitasking, which requires the OS to keep track for each process what data resides where. In addition, we look at the management of data transfers between secondary storage and RAM.

1.2 Learning Objectives

- Explain mechanisms and uses for virtual memory
  - Including principle of locality and page fault handling
  - Including paging, swapping, and thrashing
- Perform address translation with page tables
- Apply page replacement with FIFO, LRU, Clock

Take some time to think about the learning objectives specified here.

1.3 Previously on OS . . .

Recall some previously presented concepts.

1.3.1 Retrieval Practice

- How are processes and threads related?
- What happens when an interrupt is triggered?

Answer the questions listed here.

1.3.2 Recall: RAM in Hack

Take this quiz.

[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.

1.4 Big Picture

- Each process with own **virtual address space**
  - Virtual address space and RAM split into equal-sized chunks
    - **Pages** in virtual address space, **frames** in RAM
- Physical memory split among all processes

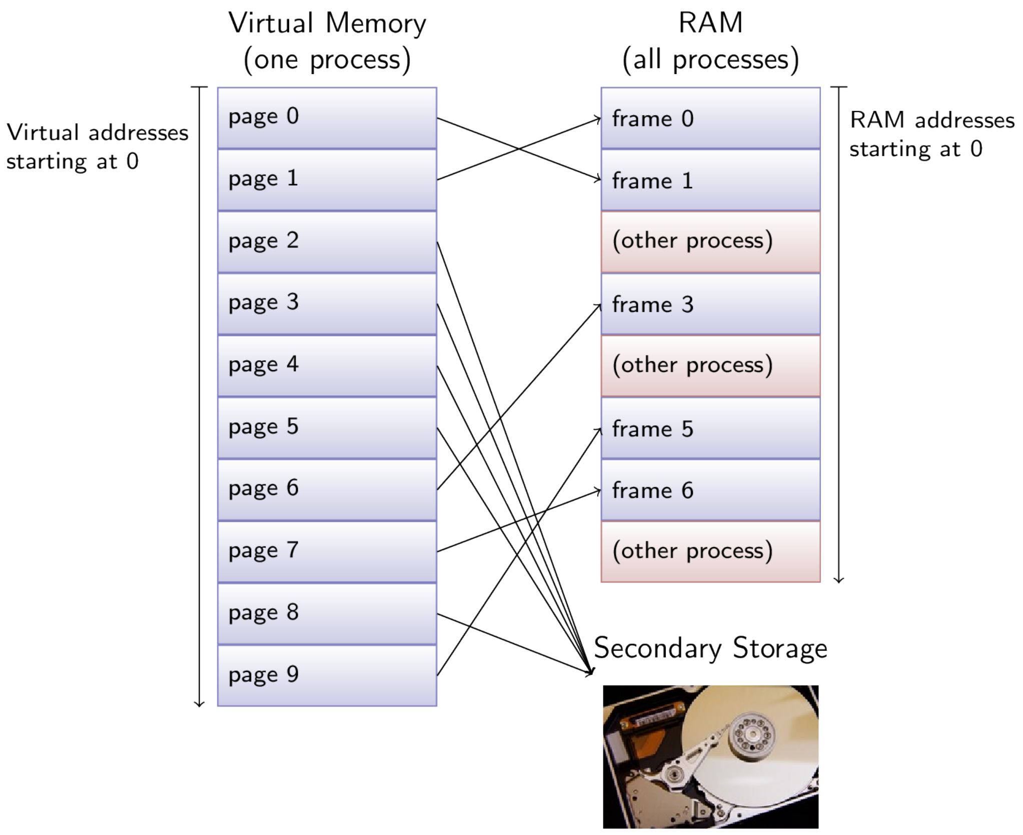

Figure 1: Big picture for virtual memory

The key idea of virtual memory management is to provide a layer of abstraction that hides allocation of the shared hardware resource RAM to individual processes. Thus, processes (and their threads) do not need to care or know whether, or where, their data structures reside in RAM.

Physical memory consists of RAM and secondary storage devices, such as solid-state disks or hard disks. Typically, the OS uses dedicated portions of secondary storage as so-called *swap areas* or *paging areas* to enlarge physical memory beyond the size of RAM. Again, processes do not need to know about this fact, which is handled by the OS in the background.

Each process has its own individual virtual address space, starting at address 0, consisting of equal-sized blocks called *pages* (e.g., 4 KiB in size each). Each of those pages may, or may not, be present in RAM. RAM in turn is split into *frames* (of the size of pages).

Here you see a sample process with a virtual address space consisting of 10 pages (numbered 0 to 9, implying that the virtual address space has a size of 10 * 4 KiB = 40 KiB), while RAM consists of 8 frames (numbered 0 to 7, implying that RAM has a size of 8 * 4 KiB = 32 KiB).

In the figure, arrows indicate which pages of the sample process are located in RAM, and where. For example, page 0 is located in frame 1, while page 3 is located on disk.

Besides, frames 2, 4, and 7 are not allocated to the process under consideration. Importantly, address spaces of different processes are isolated from each other, and threads of a process can only access frames of their process. Access to frames of other processes is not possible (unless such frames are explicitly shared, under the control of the OS).

- OS maintains page table per process

Valid	Frame#
1	1
1	0
0	X
0	X
0	X
0	X
1	3
1	6
0	X
1	5

The OS loads pages into frames and maintains a so-called *page table* per process. Such tables indicate where pages of virtual address spaces are located in physical memory. E.g., here a sample page table for the information visualized with arrows is shown. It contains one row per page of the process' virtual address space, i.e., 10 rows. The rows of the table are ordered by page number. Therefore, page numbers do not need to be included in page tables explicitly. E.g., the first row, which is row 0, indicates where page 0 is located, namely in frame 1. Afterwards, the table indicates that page 1 is located in frame 0.

Page tables also contain a number of control bits, among which only the "Valid" bit is shown here. If that bit is 1, the page is located in RAM under the given frame number. Otherwise, the page is not located in RAM; hence, the frame number does not matter and is visualized as "X". Thus, pages 2, 3, 4, and 5 are currently not located in RAM. Then, page 6 is located in frame 3 etc.

Other control bits may specify whether a page is dirty, read-only, or executable: A page is "dirty" when it contains changes compared to a clean version on disk.

If a page is marked read-only, write accesses to that page trigger an exception, which is handled by the OS.

If a page is executable, the processor is allowed to execute instructions inside that page. (Otherwise, an exception is triggered.)

Notice that neighboring pages of the virtual address space may be allocated in arbitrary order in physical memory. As processes and threads just use virtual addresses, they do not need to know about such details of physical memory.

Code of threads just uses virtual addresses within machine instructions. For RAM access, those virtual addresses need to be translated into physical addresses, which are used on the address bus. Page tables are a major tool for that translation process. In this presentation, you will see details about that translation.

1.4.1 Drawing for Page Tables

Warning! External figure **not** included: "The page table" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

This drawing illustrates several aspects of the big picture for paging. Importantly, the illustration stresses that a single virtual address means different things for different processes, as each process has its own address space. The page table specifies for each process how to interpret, or translate, its virtual addresses into RAM addresses.

Consequently, when switching between threads of different processes, the OS needs to tell the CPU which page table to use.

As a side note, regarding the final part of the illustration, when a thread tries to access a memory location that lies outsides its own address space, a so-called segmentation fault is triggered. The OS then typically terminates the thread and its process, and you may see "segmentation fault" as error message on the command line.

1.4.2 Big Picture of VM

Before you continue, learn the vocabulary introduced in the big picture.

1.5 Different Learning Styles

- The bullet point style may be exceptionally challenging for this presentation
- You may prefer this short text (PDF version)
  - It provides an alternative view on
    - Topics of Introduction
    - Topics of section Paging
- Besides, Chapter 6 of (Hailperin 2019) is about virtual memory

Parts of the topics presented subsequently are also explained in a short document. Please check out that document if you find the slides in this presentation to be confusing.

Agenda

- Part 1
  - Introduction

The topic of virtual memory is presented in two parts, of which the first one introduces paging and address translation. Afterwards, we see how to bring data from disk to RAM and how to replace RAM contents once all frames of RAM are full.

2 Paging and Address Translation

Let us see what is involved in virtual addresses and their translation to physical ones.

2.1 MMU and TLB

- Address translation by **Memory Management Unit**
  - Latency: Access of page table before RAM access

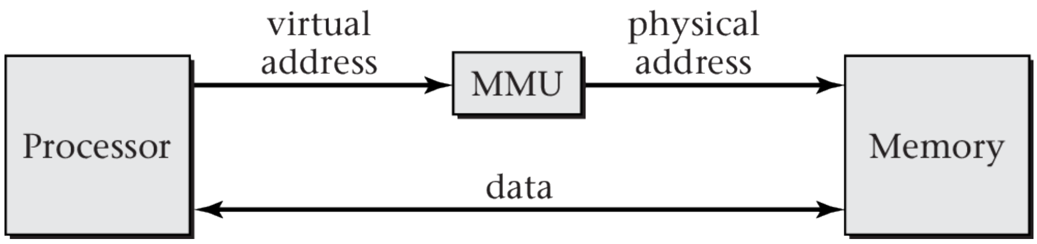

Figure 2: "Figure 6.4 of (Hailperin 2019)" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

When the CPU executes machine instructions, only virtual addresses occur in those instructions, which need to be translated into physical RAM addresses to be used on the address bus. A piece of hardware called memory management unit, MMU for short, performs that translation, before resulting physical addresses are used to access RAM contents.
The MMU uses page tables for that translation, which takes some time, adding to the latency for RAM accesses.

- Improvement: Caching
  - Special cache, **Translation Lookaside Buffer** (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 invalidate TLB $\rightarrow$ Overhead
    - (Beyond class: (Hailperin 2019) explains address space identifiers)

To reduce the latency of address translation, the so-called translation lookaside buffer, TLB for short, is used as fast cache by the MMU. The TLB contains pairs of recently accessed page numbers with their frame numbers. When a virtual address needs to be translated to a physical one, the MMU first searches for the page number in the TLB. Only if no pair for that page number is found, the page table needs to be accessed.

As each process has its own address space, context switches between threads of different processes require the use of new page tables. Thus, it may be necessary to invalidate all TLB contents upon such a context switch, followed by slow page table accesses for subsequent address translations. Thus, additional latency may arise, adding to the overhead of context switches.

Beyond class topics, our text book explains how address space identifiers may be added to TLB entries, which allows keeping TLB entries upon context switches.

2.2 Page Fault Handler

- Pages may or may not be present in RAM
  - Access of virtual address whose page is in RAM is called **page hit**
    - (Access = CPU executes machine instruction referring to that address)
  - Otherwise, **page miss**

As explained already, each page of the address space for the current thread may, or may not, be present in RAM. If the CPU executes a machine instruction that refers to an address which is present in some page in RAM, we say that a page hit occurs. In that case, the MMU translates the virtual address into a physical one, and the CPU executes the machine instruction as usual. Otherwise, i.e., if the page containing the address is not present in RAM, we say that a page miss occurs.

- Upon page miss, a **page fault** is triggered
  - Special type of interrupt
  - **Page fault handler** of OS responsible for disk transfers and page table updates
    - OS blocks corresponding thread and manages transfer of page to RAM
    - (Thread runnable after transfer complete)

In case of a page miss, the MMU generates a special type of interrupt, called page fault. Thus, the interrupt handler of the OS takes over and initiates a transfer of the page contents from disk to RAM. It also blocks the corresponding thread for the duration of that transfer, and it updates the page table once the transfer is complete.

2.3 Use of Page Table

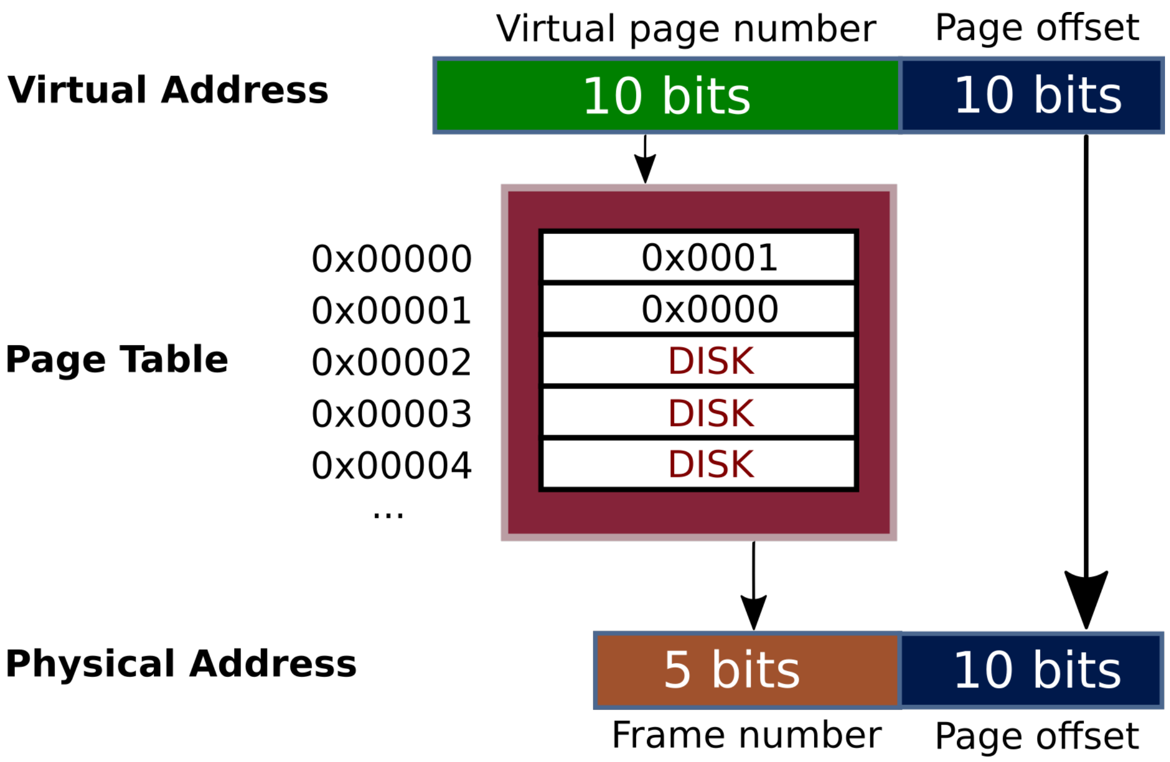

Figure 3: "Translation of hierarchical address with lookup in page table" by Max Lütkemeyer and Jens Lechtenbörger under CC BY-SA 4.0; from GitLab

Consider the page table here, which shows the beginning of the page table of the big picture (omitting the valid bit): Page 0 is contained in frame 1, page 1 in frame 0.

- Virtual `address` interpreted as **hierarchical** object
  - **Page number**, determined by most significant bits of `address`
    - Below, 10 bits; simple (unrealistic) example
  - **Offset**, remaining bits of `address` = byte number within its page
    - Below, also 10 bits; typically, 12 bits in practice

For paging, we consider virtual addresses as hierarchical objects, with two levels: On the first level, some bits enumerate pages. On the second level, the remaining bits enumerate bytes within those pages. For the sake of this simplistic example, we suppose that virtual addresses have a size of 20 bits, while physical addresses only have a size of 15 bits.

It is quite common that virtual address spaces are larger than the size of physical RAM. Indeed, recall from the big picture that virtual address spaces also cover areas of secondary storage.

Moreover, recall that pages and frames have the same size. For this example, the size is supposed to be determined by 10 bits. Thus, pages and frames share the same size of 2^{10} B, i.e., 1 KiB.

In practice, 4 KiB is a typical size for pages and frames, and addresses are much larger than 20 bits. E.g., with 32 bits, we can address up to 2^{32} B, which is 4 GiB. Even "small" devices such as smartphones may have more RAM than that, requiring more address bits… As a side note, modern processors support different, and much larger page sizes, e.g., as documented under a hyperlink in the notes for Linux.

Coming back to the sample numbers used here, we see that the first 10 bits making up the page number are used as index into the page table, where the frame number for the page is found.

The remaining 10 bits are used as stable offset into pages and frames as illustrated next.

2.3.1 Offset as Pointer into Range

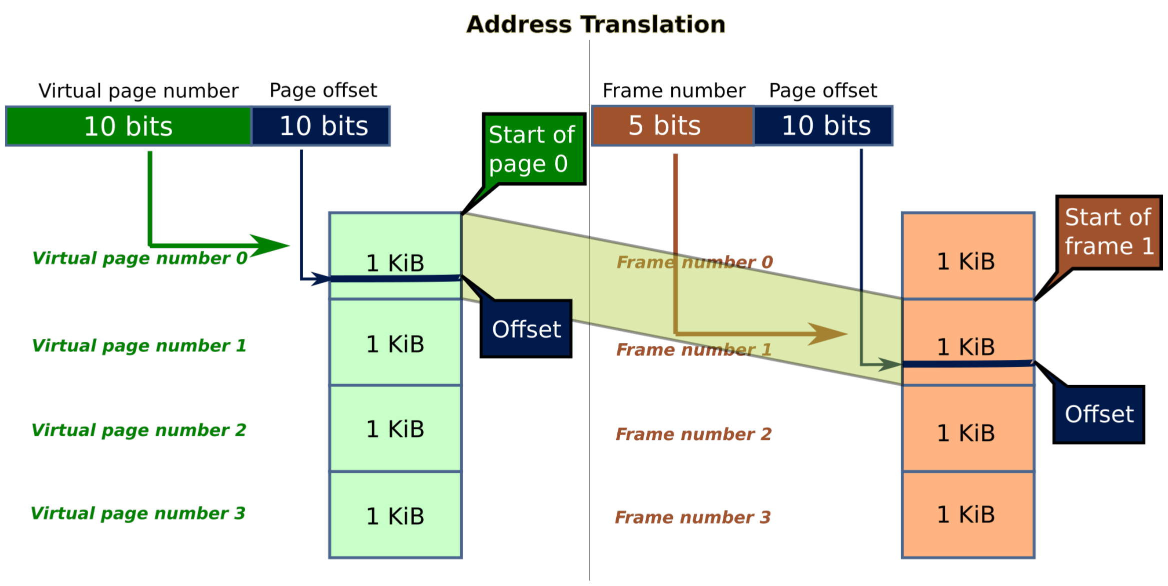

Figure 4: "Address translation with offset in covered address range" by Max Lütkemeyer and Jens Lechtenbörger under CC BY-SA 4.0; from GitLab

For a different view on the hierarchical nature of virtual addresses, let us continue the previous scenario of virtual addresses of 20 bits, to be translated to physical addresses of 15 bits, with a page size of 1 KiB.

Out of the $2^{10} = 1024$ possible pages and $2^5 = 32$ possible frames, only the first four of each type are shown.

As before, suppose that page 0 is located in frame 1, as recorded in the page table. Thus, for translation of addresses falling into page 0, the 0 encoded in the first 10 bits of the virtual address is replaced by a 1 encoded in the first 5 bits of the physical address. Importantly, the 10 offset bits do **not** change under address translation.

Note how, given 10 bits for the offset, each page and each frame cover a range of 1024 addresses. The offset identifies a single byte in that range.

Subsequent slides provide sample calculations for address translation.

2.3.2 Address Translation Example (1/3)

- Task: Translate virtual address to physical address
  - Subtask: Translate bits for page number to bits for frame number

    Let us translate a virtual address to a physical address. As explained already, this involves a lookup of the frame number for the page number containing the virtual address under consideration.
  - Suppose
    - Pages and frames have a size of 1 KiB (= 1024 B)
    - 15-bit physical addresses for RAM locations
    - 20-bit virtual addresses, as on previous slides

  Let us use the sample sizes and numbers of bits shown here.
- First, **derive** following pieces of information

  Importantly, from the information given previously, we can infer or derive the pieces of information shown subsequently.
  - Size of physical address space: 2^{15} B = 32 KiB

    From the numbers of bits used for physical addresses, we can compute the size of the physical address space.
  - Size of virtual address space: 2^{20} B = 1024 KiB = 1 MiB

    From the numbers of bits used for virtual addresses, we can compute the size of the virtual address space.

- 10 bits are used for offsets (as 2^{10} B = 1024 B)
  - Remaining 5 physical bits enumerate $2^5 = 32$ frames
  - Remaining 10 virtual bits enumerate $2^{10} = 1024$ pages

Given the number of bits used for the offset, we can compute the size of pages and frames. Given the numbers of bits not used for the offset, we can compute the numbers of frames and pages.

Note that the numbers fit together: The number of frames multiplied by the size of frames equals the size of the physical address space. Similarly, the number of pages multiplied by the size of pages equals the size of the virtual address space.

2.3.3 Address Translation Example (2/3)

- Hierarchical interpretation of addresses
  - 20-bit virtual address: 10 bits for page number 10 bits for offset
  - 15-bit physical address: 5 bits for frame number 10 bits for offset
- Task: Translate virtual address 42

  The hierarchical interpretation of addresses is emphasized with different colors for the bits of page numbers, frame numbers, and offset.

  Let us translate the virtual address 42 to a physical address.

  - 42 = 0000000000 0000101010
    - Page number = 0000000000 = 0
    - Offset = 0000101010 = 42

    First, translate the address to binary. Note that in this case all bits for the page number are 0.

    Here, the offset is 42 in binary, with leading 0s.
  - Based on page table: Page 0 is located in frame 1
    - In general, address translation exchanges page number with frame number
      - Here, 0 with 1

    In general, address translation exchanges the page number with its frame number. In the page table we see that page 0 is located in frame 1. Thus, the bits encoding the page number 0 have to be replaced with bits encoding 1.
  - Thus, 42 is located in frame 1
    - Physical address 00001 0000101010 = 1066 (= 1024 + 42)

    The resulting physical address is shown here, both in binary and in decimal.

2.3.4 Address Translation Example (3/3)

- Based on page table
  - Page 6 is located in frame 3
- Page 6 contains addresses between 6*1024 = 6144 and 6*1024+1023 = 7167
  - Consider virtual address 7042
    - 7042 = 0000000110 1110000010
      - Page number = 0000000110 = 6
      - Offset = 1110000010 = 898
    - Replace page number 6 with frame number 3
    - 7042 is located in frame 3
      - Physical address 00011 1110000010 = 3970 (= 3*1024 + 898)

Similarly to the previous slide, here you see the translation of a different virtual address. Please verify its steps.

2.4 Self-Study Tasks

Answer the following questions in Learnweb.

Suppose that 32-bit virtual addresses with 4 KiB pages are used.

- How many bits are necessary to number all bytes within pages?
- How many pages does the address space contain? How many bits are necessary to enumerate them?
- Where within a 32-bit virtual address can you “see” the page number?

Take a break for self-study questions shown here.

Bibliography

Hailperin, Max. 2019. *Operating Systems and Middleware – Supporting Controlled Interaction.* revised edition 1.3.1. `https://github.com/Max-Hailperin/Operating-Systems-and-Middleware--Supporting-Controlled-Interaction`.

The bibliography contains references used in this presentation.

License Information