OS08: Virtual Memory I *

Based on Chapter 6 of [Hai19]

Jens Lechtenbörger

Computer Structures and Operating Systems 2022

1 Introduction

1.1 OS Plan

- OS Overview (Wk 20)
- OS Introduction (Wk 20)
- Interrupts and I/O (Wk 21)
- Threads (Wk 22)
- Thread Scheduling (Wk 22)
- Mutual Exclusion (MX) (Wk 24)
- 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)

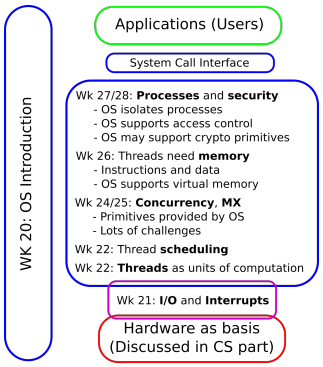

Figure 1: OS course plan, summer 2022

1.2 Today's Core Questions

- 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?

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

1.3 Learning Objectives

- Explain mechanisms and uses for virtual memory
  - Including principle of locality and page fault handling
  - Including access of data on disk
- Explain and perform address translation with page tables

1.4 Previously on OS ...

1.4.1 Retrieval Practice

- How are processes and threads related?
- What happens when an interrupt is triggered (e.g., a page fault)?

1.4.2 Recall: RAM in Hack

1.5 Big Picture

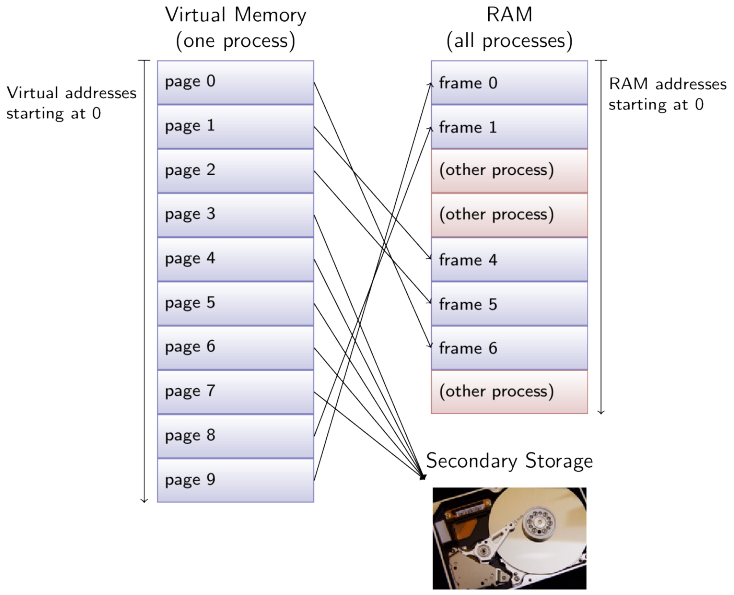

Figure 2: Big picture for virtual memory

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 SSDs or HDDs. 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 need neither care nor know about this fact, which is handled by 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). The OS loads pages into frames and keeps track what pages of virtual address spaces are located where in physical memory.

Here you see a 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). For example, page 0 is located in frame 6, while page 3 is located on disk, and frames 2, 3, and 7 are not allocated to the process under consideration.

Notice that neighboring pages in 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, and it is the OS's task to locate the corresponding physical addresses in RAM or to bring data from secondary storage to RAM in the first place.

1.6 Different Learning Styles

- The bullet point style may be particularly challenging for this presentation
- You may prefer this 5-page introduction
  - It provides an alternative view on
    - Topics of Introduction and Main Concepts
    - Topics of section Paging
  - After working through that text, you may want to jump directly to the corresponding JiTT tasks to check your understanding
    - Afterwards, come back here to look at the slides, in particular work through section Uses for Virtual Memory (not covered in the text)
- Besides, Chapter 6 of [Hai19] is about virtual memory

Table of Contents

2 Main Concepts

2.1 Modern Computers

- RAM is **byte**-addressed (1 byte = 8 bits)
  - Each `address` selects a byte (not a word as in Hack)
    - (Machine instructions typically operate on words (= multiple bytes), though)
- **Physical** vs **virtual** addresses
  - Physical: Addresses used on memory bus
    - Hack `address`
  - Virtual: Addresses used by threads and CPU
    - Do not exist in Hack

2.2 Virtual Addresses

- Additional layer of **abstraction** provided by OS
  - Programmers do not need to worry about physical memory locations at all
  - Pieces of data (and instructions) are identified by virtual addresses
    - At different points in time, the same piece of data (identified by its virtual address) may reside at different locations in RAM (identified by different physical addresses) or may not be present in RAM at all
- OS keeps track of **(virtual) address spaces**: What (virtual address) is located where (physical address)
  - Supported by hardware, **memory management unit** (**MMU**)
    - Translation of virtual into physical addresses (see next slide)

2.2.1 Memory Management Unit

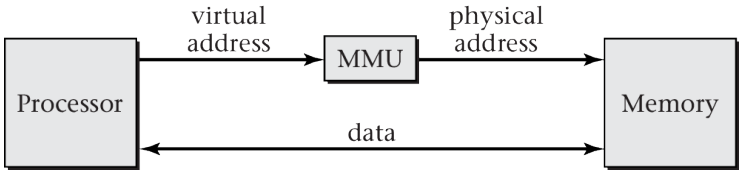

Figure 3: "Figure 6.4 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

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) performs that translation, before resulting physical addresses are used on the memory's address bus to access RAM contents, i.e., data.

As explained in detail later on, the OS manages data structures called page tables to keep track of what virtual addresses correspond to what physical addresses, and the MMU uses those page tables during address translation. Also, as discussed in the next presentation but not shown here, the MMU uses a special cache called translation lookaside buffer (TLB) to speed up address translation.

2.3 Processes

- OS manages running programs via **processes**
  - More details in upcoming presentation
- For now: **Process** $\approx$ group of threads that share a virtual address space
  - Each process has its **own** address space
    - Starting at virtual address 0, mapped per process to RAM by the OS, e.g.:
      - Virtual address 0 of process P1 located at physical address 0

        - Virtual address 0 of process P2 located at physical address 16384
        - Virtual address 0 of process P3 not located in RAM at all
      - Processes may **share** data (with OS permission), e.g.:
        - `BoundedBuffer` located at RAM address 42
        - Identified by virtual address 42 in P1, by 4138 in P3
  - Address space of process is **shared by its threads**
    - E.g., for all threads of P2, virtual address 0 is associated with physical address 16384

2.4 Pages and Page Tables

- Mapping between virtual and physical addresses does **not** happen at level of bytes
  - Instead, larger **blocks** of memory, say 4 KiB
    - Blocks of virtual memory are called **pages**
    - Blocks of physical memory (RAM) are called **(page) frames**
- OS manages a **page table** per process
  - One entry per page
    - In what frame is page located (if present in RAM)
    - Additional information: Is page read-only, executable, or modified (from an on-disk version)?

2.4.1 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**
- 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)

2.5 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.)

3 Uses for Virtual Memory

3.1 Private Storage

- Each process has its own address space, **isolated** from others
  - **Autonomous use** of virtual addresses
    - Recall: Virtual address 0 used differently in every process
  - Underlying **data protected** from accidental and malicious modifications by other processes
    - OS allocates frames exclusively to processes (leading to disjoint portions of RAM for different processes)
    - Unless frames are explicitly shared between processes
      - Next slide
- Processes may partition address space
  - Read-only region holding machine instructions, called **text**
  - Writable region(s) holding rest (data, stack, heap)

3.2 Controlled Sharing

- OS may map limited portion of RAM into multiple address spaces
  - Multiple page tables contain entries for the **same frames** then
    - See `smem` demo later on
- Shared code
  - If same program runs multiple times, processes can share text
  - If multiple programs use same libraries (libXYZ.so under GNU/Linux, DLLs under Windows), processes can share them

3.2.1 Copy-On-Write Drawing

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

3.2.2 Copy-On-Write (COW)

- Technique to create a copy of data for second process
  - Data may or may not be modified subsequently
- Pages **not** copied initially, but marked as **read-only** with access by second process
  - Entries in page tables of both processes point to original frames
  - Fast, no data is copied

- If process tries to **write** read-only data, MMU triggers interrupt
  - Handler of OS **copies** corresponding frames, which then become writable
    - **Copy** only takes place **on write**
  - Afterwards, write operation on (now) writable data

3.3 Flexible Memory Allocation

- Allocation of RAM does not need to be contiguous
  - Large portions of RAM can be allocated via individual frames
    - Which may or may not be contiguous
    - See next slide or big picture
  - The virtual address space can be contiguous, though

3.3.1 Non-Contiguous Allocation

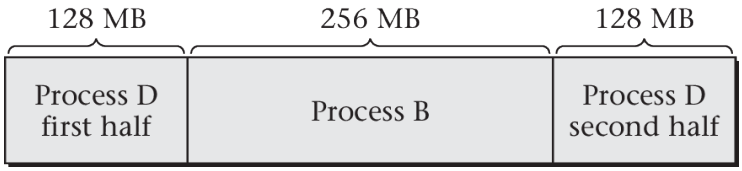

Figure 4: "Figure 6.9 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

3.4 Persistence

- Data kept persistently in files on secondary storage
- When process opens file, file can be **mapped** into virtual address space
  - Initially without loading data into RAM
    - See `page 3` 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**
  - 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

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, 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.

While read accesses just return the requested data, write accesses trigger 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. Afterwards, the file's pages are marked as clean and read-only again.

3.5 Demand-Driven Program Loading

- Start of program is **special case** of previous slide
  - Map executable file into virtual memory
  - Jump to first instruction
    - Page faults automatically trigger loading of necessary pages
    - No need to load entire program upon start
      - Faster than loading everything at once
      - Reduced memory requirements

3.5.1 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
- **Principle of locality**
  - Memory references typically close to each other
  - Few pages sufficient for some interval
- **Working set**: Necessary pages for some interval
  - Aim: Keep working set in resident set
    - Replacement policies in next presentation

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

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 without the principle of locality caching, i.e., 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; neither the form of caching seen here, where RAM serves as cache for disk data, nor CPU caches for RAM data.

The so-called working set (for some given time interval) of a thread T is that set of pages which allows T 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 is contains the working set (and ideally not much else). Page replacement policies, to be discussed in the next presentation, work towards that goal.

4 Paging

4.1 Major Ideas

- Virtual address spaces split into **pages**, RAM into **frames**
  - Page is **unit of transfer** by OS
    - Between RAM and secondary storage (e.g., disks)
  - Each virtual `address` can be interpreted in two ways
    1. Integer number (`address` as binary number, as in Hack)
    2. Hierarchical object consisting of page number and offset
       - **Page number**, determined by most significant bits of `address`
       - **Offset**, remaining bits of `address` = byte number within its page
         - (Detailed example follows)
- **Page tables** keep track of RAM locations for pages
  - If CPU uses virtual address whose page is not present in RAM, the Page fault handler takes over

4.2 Sample Memory Allocation

- Sample allocation of frames to some process

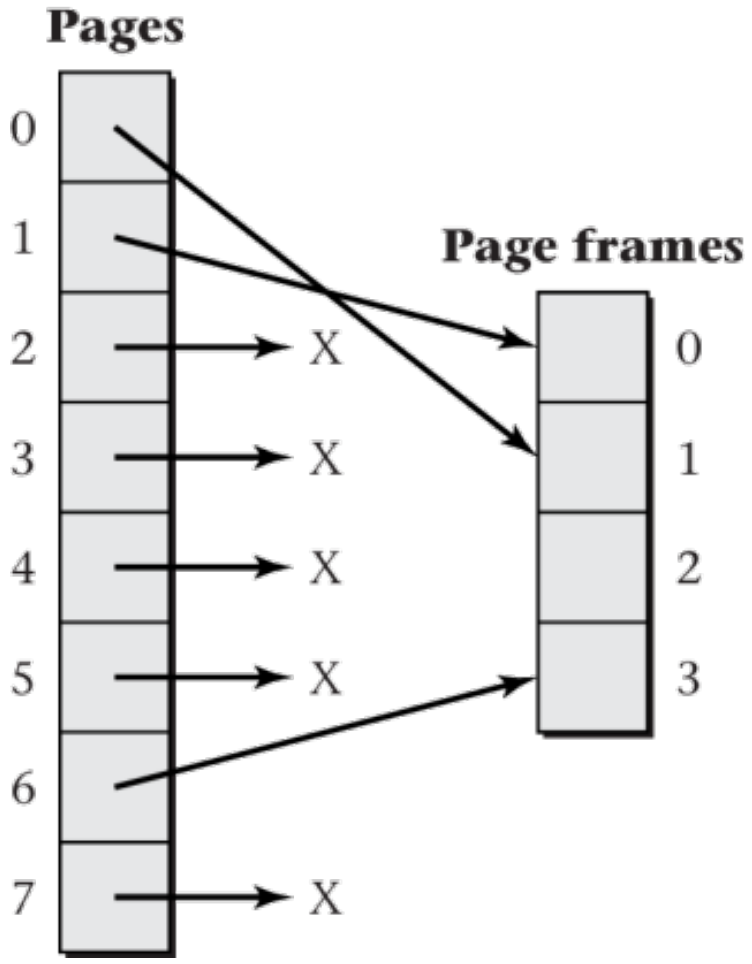

Figure 5: "Figure 6.10 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

Several subsequent slides will refer to this example, which shows a main memory situation with just four frames of main memory. Clearly, that is an unrealistically small example, but it is sufficient to demonstrate the main points. Here, a process with a virtual address space of 8 pages is shown, some of which are allocated to frames as indicated by arrows. Note that neighboring pages can (a) be mapped to frames in arbitrary order or (b) not be mapped to RAM at all. The Xs indicate that no frame is assigned to hold pages 2-5 or page 7. Frame 2 is unused.

4.3 Page Tables

- Page Table = Data structure managed by OS
  - **Per process**
- Table contains one entry per page of virtual address space
  - Each entry contains
    - Frame number for page in RAM (if present in RAM)

    - Control bits (not standardized, e.g., valid, read-only, dirty, executable)
    - Note: Page tables do not contain page numbers as they are implicitly given by row numbers (starting from 0)
  - Note on following sample page table
    - "0" as valid bit indicates that page is not present in RAM, so value under "Frame#" does not matter and is shown as "X"

4.3.1 Sample Page Table

- Consider previously shown RAM allocation (Fig. 6.10)

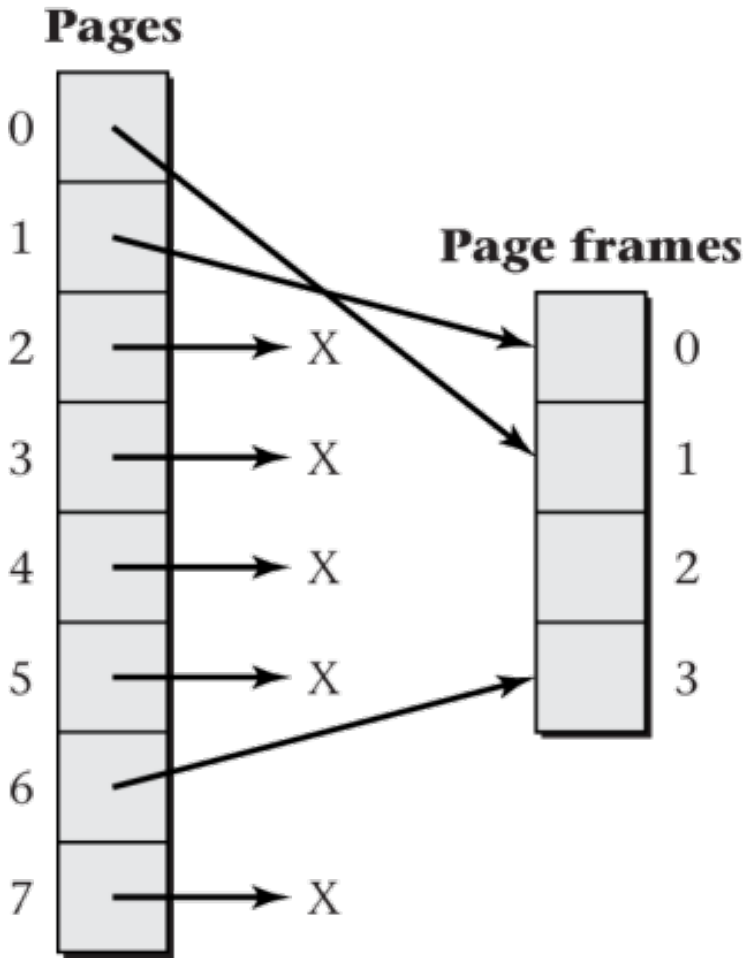

Figure 6: "Figure 6.10 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

  - Page table for that situation (Fig. 6.11)

Valid	Frame#
1	1
1	0
0	X
0	X
0	X
0	X
1	3
0	X

4.3.2 Address Translation Example (1/3)

- Task: Translate virtual address to physical address
  - Subtask: Translate bits for page number to bits for frame number
- Suppose
  - Pages and frames have a size of 1 KiB (= 1024 B)
  - 15-bit addresses, as in Hack
- Consequently
  - Size of address space: 2^{15} B = 32 KiB
  - 10 bits are used for offsets (as 2^{10} B = 1024 B)
  - The remaining 5 bits enumerate 2^5 = 32 pages

4.3.3 Address Translation Example (2/3)

- Hierarchical interpretation of 15-bit addresses
  - Virtual address: 5 bits for page number 10 bits for offset
  - Physical address: 5 bits for frame number 10 bits for offset
- Based on page table
  - Page 0 is located in frame 1
- Page 0 contains virtual addresses between 0 and 1023, frame 1 physical addresses between 1024 and 2047
  - Consider virtual address 42
    - 42 = 00000 0000101010
      - Page number = 00000 = 0
      - Offset = 0000101010 = 42
    - 42 is located at physical address 00001 0000101010 = 1066 (= 1024 + 42)

4.3.4 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 = 00110 1110000010
      - Page number = 00110 = 6
      - Offset = 1110000010 = 898
    - In general, address translation exchanges page number with frame number
      - Here, 6 with 3
    - 7042 is located at physical address 00011 1110000010 = 3970 (= 3*1024 + 898)

4.4 JiTT Tasks

4.4.1 JiTT: Address Translation

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?

4.4.2 JiTT: A quiz

4.5 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
      - Page table itself needs 2^{10} pages/frames! **Per process**!
  - Much worse for 64-bit addresses
- Solutions to reduce amount of RAM for page tables
  - Multilevel (or **hierarchical**) page tables (2 or more levels)
    - Tree-like structure, efficiently representing large unused areas
    - Root, called **page directory**
      - 4 KiB with 2^{10} entries for page tables
      - Each entry represents 4 MiB of address space

  - Inverted page tables in next presentation

While the sample pages tables shown so far may seem simple to manage, pages tables can be huge in practice. As page tables are used to locate data in RAM, a naïve implementation might require the page tables themselves to be located in RAM in the first place. Let's see how large page tables might get.

With 32-bit addresses, you see a calculation on this slide, showing that the page table for every process requires up to 4 MiB of RAM. Note that those 4 MiB are pure OS overhead, unusable for applications. So, after you booted your system half a GB of RAM may already be gone.

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 2^{39} B $= 2^9$ GiB $= 512$ GiB. In words: A single page table might occupy 512 GiB of RAM, quite likely more than you've got.

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, in case of 32-bit addresses and two levels (as on subsequent slides) each entry represents 1024 pages with a total size of 4 MiB. If such a 4 MiB region is not used at all, no data needs to be allocated in lower levels of the tree like structure. Details are presented on subsequent slides.

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 virtual address space. You will see how address translation works with inverted page tables on later slides. Right now, you may want to think about that yourself. Starting again from a page number for which the corresponding frame number is necessary, how do you locate the appropriate entry in the inverted page table? Clearly, a linear search is too slow.

5 Multilevel Page Tables

5.1 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 depth n, split page number into n smaller parts
    - Two-level: 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 uses 5-level tables by default since 2019-09-16

5.2 Two-Level Page Table

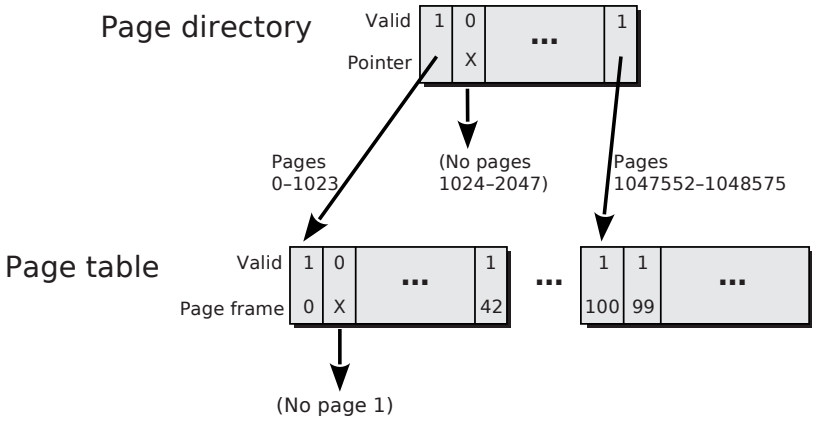

Figure 7: "IA-32 two-level page table" by Jens Lechtenbörger under CC BY-SA 4.0; Frame numbers and valid bits added to and third layer removed from Figure 6.13 of [Hai17] by Max Hailperin under CC BY-SA 3.0. Source at GitLab.

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.

5.2.1 Two-Level Address Translation

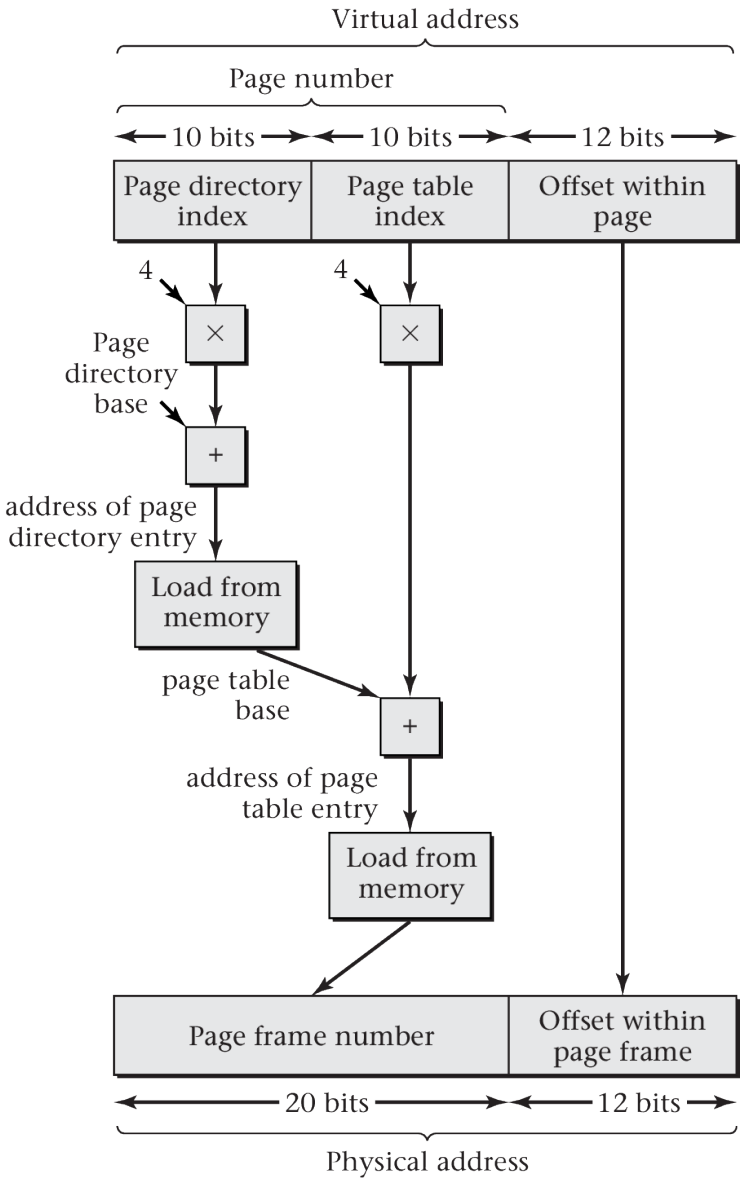

Figure 8: "Figure 6.14 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

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.

5.3 JiTT: Questions, Feedback, Survey

1. This slide serves as reminder that I am happy to obtain and provide feedback for course topics and organization. If **contents** of presentations are confusing, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (maybe on GitLab). Please use the session's shared document or MoodleOverflow. Most questions turn out to be of general interest; please do not hesitate to ask and answer where others can benefit. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

2. Please take some minutes to answer this anonymous survey in Learnweb on your thoughts related to Just-in-Time Teaching (JiTT) and the presentation format for Operating Systems.

6 Looking at Memory

6.1 Linux Kernel: `/proc/<pid>/`

- `/proc` is a pseudo-filesystem
  - See `https://man7.org/linux/man-pages/man5/proc.5.html`
    - (Specific to Linux kernel; incomplete or missing elsewhere)
  - "Pseudo": Look and feel of any other filesystem
    - Sub-directories and files
    - However, files are no "real" files but meta-data
  - Interface to internal **kernel data structures**
    - One sub-directory per process ID
    - OS identifies process by integer number
    - Here and elsewhere, `<pid>` is meant as **placeholder** for such a number

6.1.1 Video about `/proc`

The video shows some aspects of the `/proc` filesystem related to memory management, which are described in more abstract form on subsequent slides.

6.1.2 Drawing about `/proc`

Warning! External figure **not** included: "/proc" © 2018 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

6.1.3 Drawing about `man` pages

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

6.2 Linux Kernel Memory Interface

- Memory allocation (and much more) visible under `/proc/<pid>`
- E.g.:
  - `/proc/<pid>/pagemap`: One 64-bit value per virtual page
    - Mapping to RAM or swap area
  - `/proc/<pid>/maps`: Mapped memory regions
  - `/proc/<pid>/smaps`: Memory usage for mapped regions
- Notice: Memory regions include **shared** libraries that are used by lots of processes

6.3 GNU/Linux Reporting: `smem`

- User space tool to read `smaps` files: `smem`
  - See `https://linoxide.com/memory-usage-reporting-smem/`
- Terminology
  - **Virtual set size** (VSS): Size of virtual address space
  - **Resident set size** (RSS): Allocated main memory
    - Standard notion, yet overestimates memory usage as lots of memory is shared between processes
      - Shared memory is added to the RSS of every sharing process
  - **Unique set size** (USS): memory allocated exclusively to process
    - That much would be returned upon process' termination
  - **Proportional set size** (PSS): USS plus "fair share" of shared pages
    - If page shared by 5 processes, each gets a fifth of a page added to its PSS

6.3.1 Sample `smem` Output

```
$ smem -c "pid command uss pss rss vss" -P "bash|xinit|emacs"
  PID Command                          USS      PSS      RSS      VSS
  765 /usr/bin/xinit /etc/X11/Xse      220      285     2084    15952
 1390 /bin/bash -c libreoffice5.3      240      510     2936    13188
  826 /bin/bash /usr/bin/qubes-se      256      524     3008    13204
  750 -su -c /usr/bin/xinit /etc/      316      587     3368    21636
 1251 bash                            4864     5136     7900    26024
 2288 /usr/bin/python /usr/bin/sm     5272     6035     9432    24688
 1145 emacs                          90876    93224   106568   662768
```

6.3.2 Sample `smem` Graph

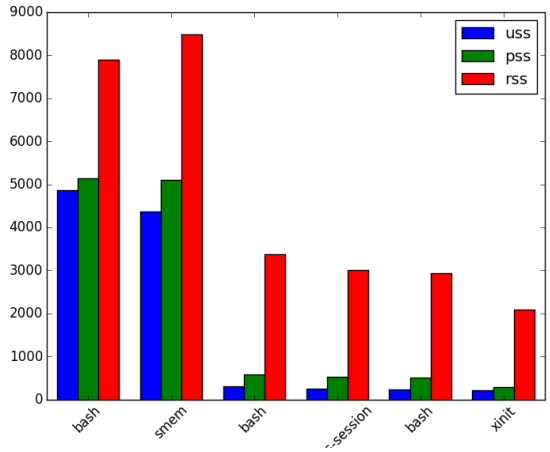

Figure 9: `smem --bar pid -c "uss pss rss" -P "bash|xinit"` ("Screenshot of smem" under CC0 1.0; from GitLab)

7 Conclusions

7.1 Summary

- Virtual memory provides abstraction over RAM and secondary storage
  - Paging as fundamental mechanism
    - Isolation of processes
    - Stable virtual addresses, translated at runtime
- Page tables managed by OS
  - Address translation at runtime
  - Hardware support via MMU with TLB
  - Multilevel page tables represent unallocated regions in compact form

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