Course
TÖL401G: Stýrikerfi / Operating Systems
8. Memory Management Strategies

Chapter Objectives

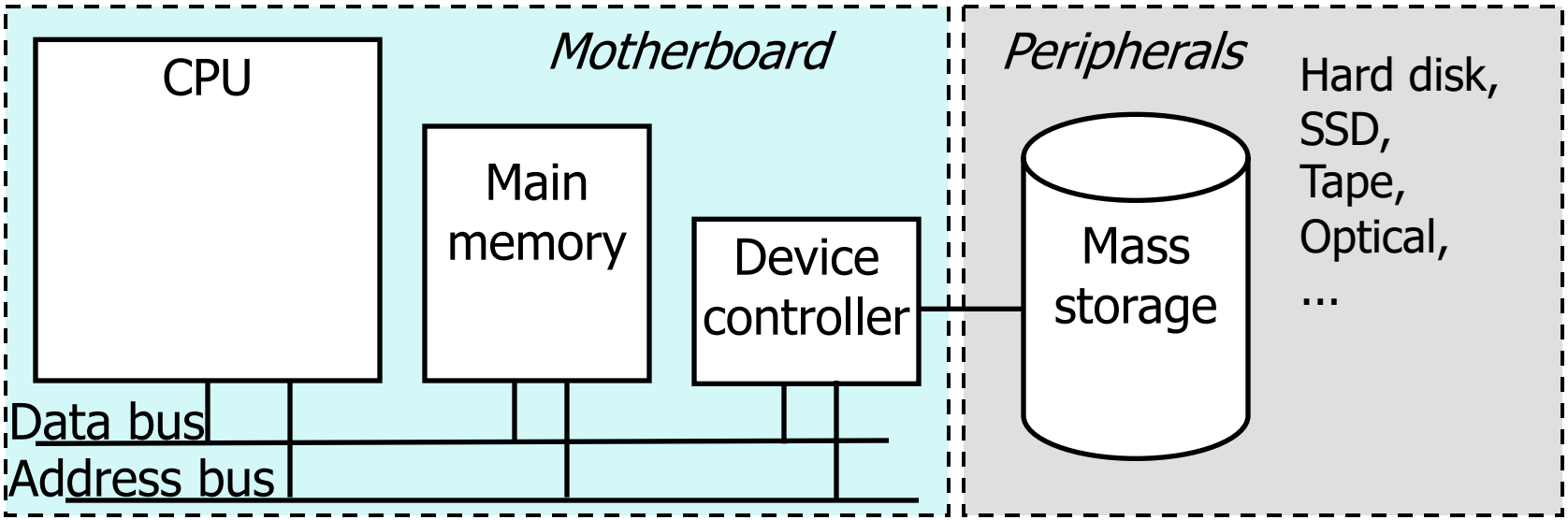
- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in translating addresses.
- Explain memory organisation and address binding when loading processes and how they relate to different types of MMUs.
- Explain Swapping.
- Apply first-, next-, best-, and worst-fit strategies for allocating memory contiguously.
- Explain the distinction between internal and external fragmentation.
- Understand Paged MMUs (PMMUs) with page tables and translation lookaside buffers for speeding up lookups.
- Translate logical to physical addresses in a paging system using PMMU.
- Describe hierarchical paging/multi-level paging.
- Describe applications of paging: memory protection, circumvent external fragmentation, shared memory.

Contents

Note for users of the Silberschatz et al. book: in the beginning, my slides or more extensive than in the book (whereas towards the end, I omit some topics in comparison to the book).

8.1 Introduction

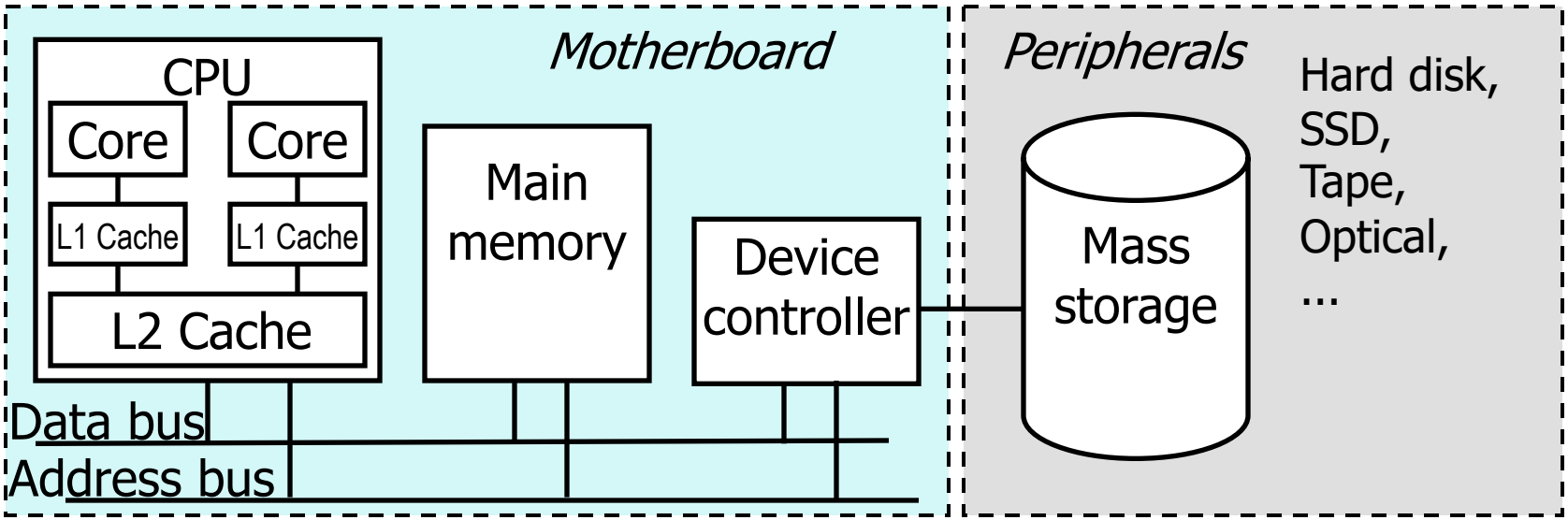
- Different types of memory in a multi-core CPU computer:



- CPU can only access locations (& execute instructions) in main memory (& read, store, and process data in CPU registers).
  - Mass storage only indirectly accessible via device controller.
  - Program instructions must be loaded from mass storage to main memory.

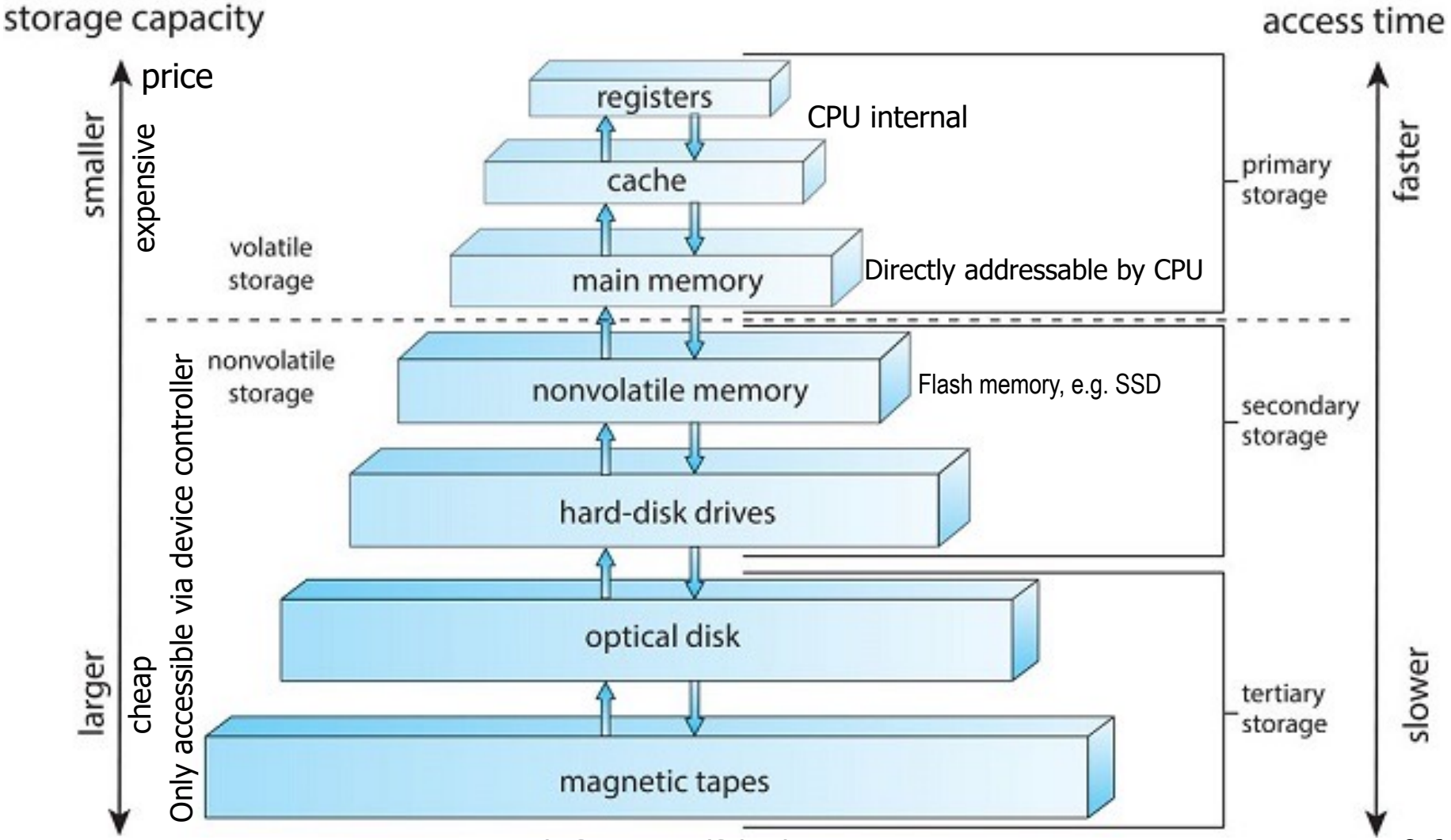
CPU Caches

- Access to main memory is slow (in comparison to CPU registers).



- Cache stores values that have been read or written before.
  - Fast cache RAM (reduces memory stalls), but limited in size.
- Multi-level caching (note: sometimes even Level 3 caches used):
  - Level 1 (L1) cache uses faster (& more expensive) RAM: small size.
  - Level 2 (L2) cache uses slower (& less expensive) RAM: larger size.

Reminder Chapter 1: Storage-Device Hierarchy



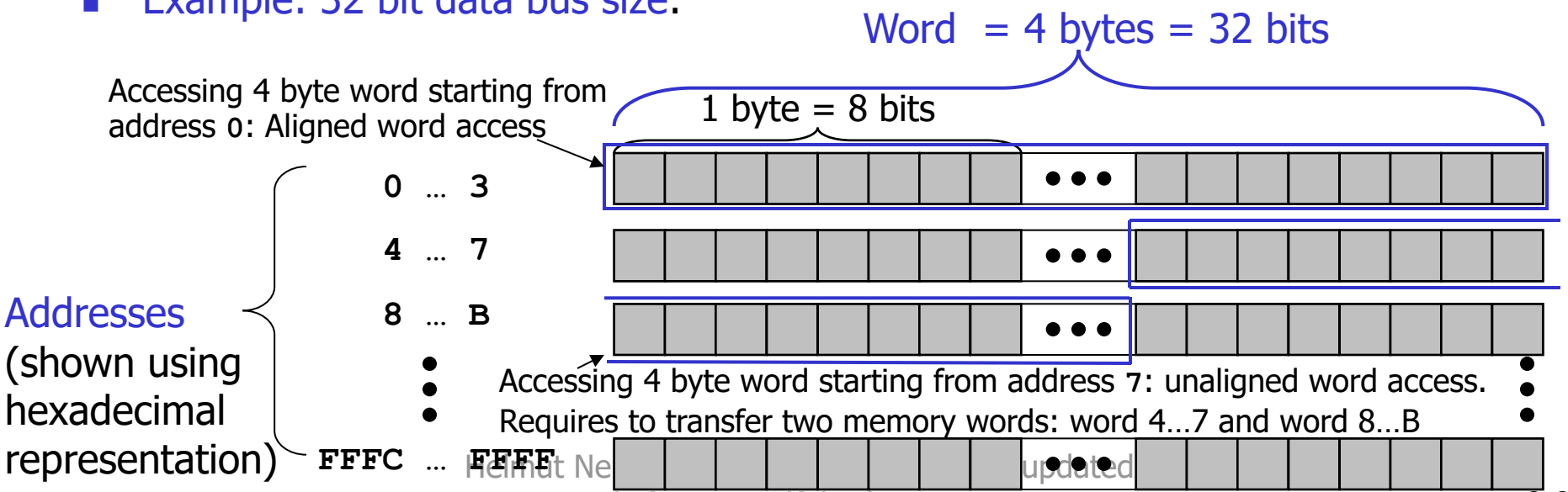
Reminder Chapter 1:
Performance of Various Levels of Storage

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- In the past, cost was the main driver that prevented to use larger main memory.
- Then, the maximum memory addressable by a 32 Bit CPU became a restriction:
  - 32 Bit CPU uses 32 Bit to address memory: maximum memory addressable with 32 Bit: 4GB
- Today's 64 Bit CPUs allows to address larger memory.

Addressing Main Memory: Data

- Data bus size (e.g. 32 bit "words" or 64 bit "words") determines how many bits are transferred during a memory access cycle.
  - RAM address to be accessed is put on the address bus by the CPU,
  - Then, data bus is used to transfer words between RAM and CPU.
    - Even if CPU wants to process only single bits or bytes, always the whole word is transferred.
  - If a word shall be accessed that is not aligned (i.e. crosses boundary of two physical words in memory), two memory access cycles are required. (⇒Aligned access faster.)
- Example: 32 bit data bus size:



Addressing Main Memory: Addresses

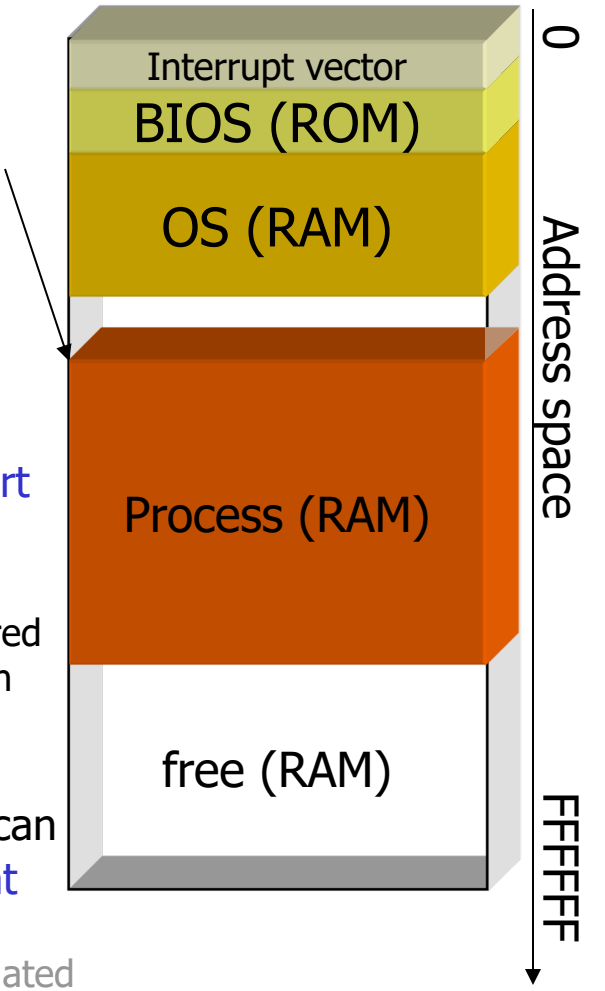
- Address bus size and CPU address register size restrict memory size.
  - Size (in bits) of CPU address registers restricts maximum number of bytes addressable by programs (logical addresses).
    - Machine code uses CPU address register to address and access main memory.
    - (Program counter (PC) is a special address register to address the instruction in main memory to be executed.)
  - Size of address bus, i.e. number of CPU pins and number of address bus tracks found on motherboard, restricts size of accessible physical memory (physical addresses).
    - It makes no sense to plug-in more RAM in your computer than the address bus can address.
    - As memory is always accessed word-wise using word-aligned memory addresses, the address bus lacks the lowest significant bits.
      - E.g. with 64 bit words=8 Byte words, the lowest 3 Bits lacking on the address bus. (e.g. bytes 0..7 are read as one word over the databus).
- ⇒It is possible to logically address more locations than physically accessible: in chapter 9 on virtual memory, we will learn more.

Addressing Main Memory: Addresses: 32 bit / 64 bit examples

- Example: 32 bit system:
  - 32 bit CPU address registers: allow to address 4GB.
  - 32 bit data bus: allows to plug-in 4GB of RAM.
- Example: 64 bit system:
  - 64 bit CPU address registers: allow to address 16 exabytes (=16.8 million TB).
  - As today, no-one needs 16 exabytes of RAM modules in a single computer, today's 64 bit CPUs only have – depending on CPU– a 36, 40 or 44 bit address bus : allows to access at least 64 GB of RAM (if motherboard allows to plug in as much).

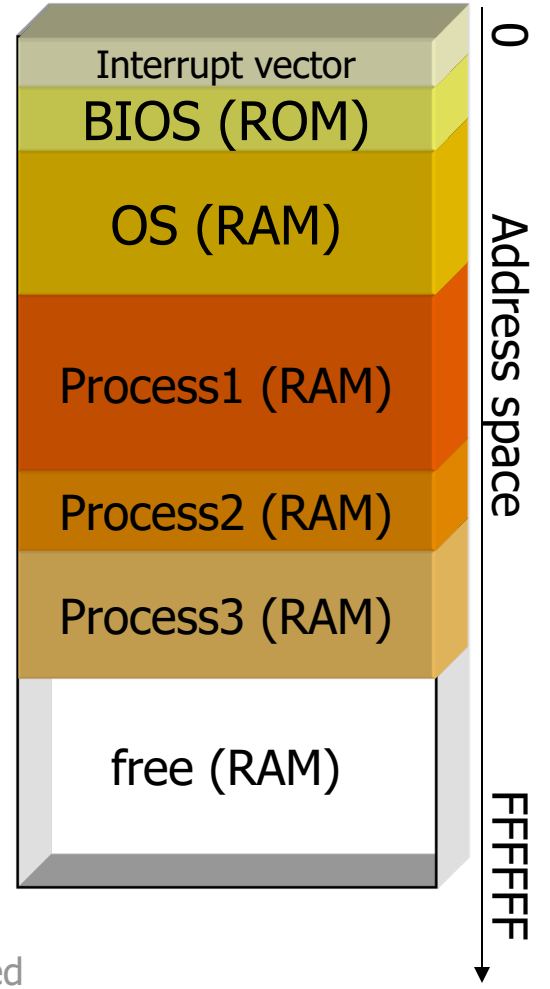
Organisation of Memory: Simple Monoprogramming

- Monoprogramming (e.g. MS-DOS):
  - BIOS in ROM, OS in RAM,
  - OS loads & executes only one process at a time.
  - ⇒ (Apart from BIOS & OS) there are only the instructions of the current process found in the address space:
    - Program code is created for running at a fixed start address in memory (absolute code).
      - E.g.: All programs are loaded at the start address 1000. If a program wants to access some data stored at the 8th byte of the program, the programmer can rely on that this byte will be at address 1008. (I.e. "binding" of address occurs at compile time.)
    - No memory protection: the whole address space can be accessed by the current process: process might overwrite OS and thus crash the system.



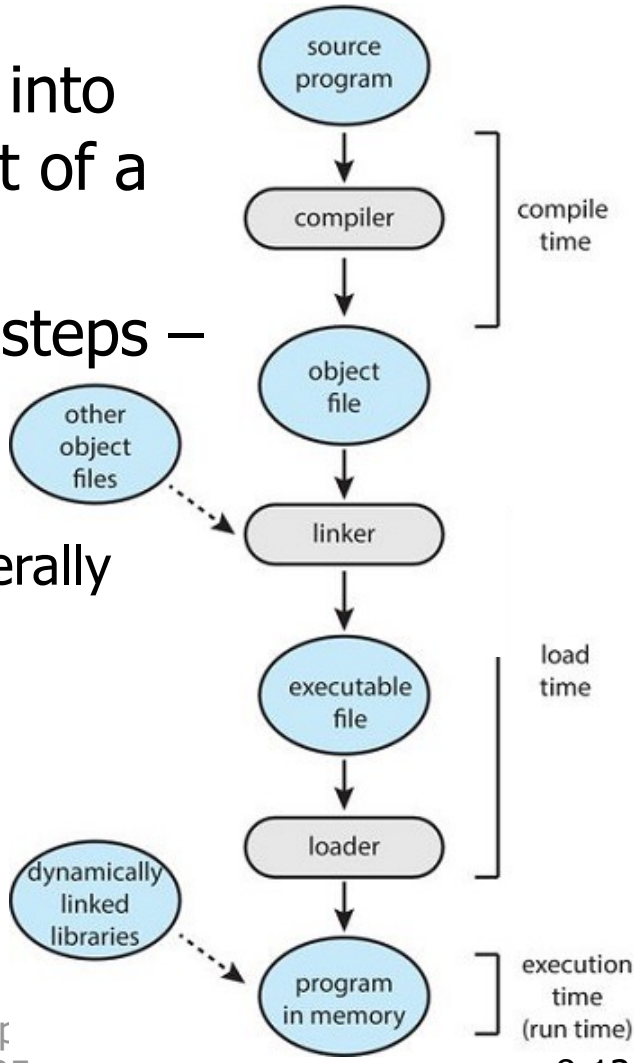
Organisation of Memory: Simple Multiprogramming

- Simple multiprogramming, i.e. still shared address space (e.g. Windows 3.11, 95, 98, ME):
  - BIOS in ROM, OS in RAM,
  - Multiple processes sharing the available RAM.
    - Start address is different for each process.
    - A program cannot rely anymore at which absolute address it is executing. (It rather depends on which start address it has been loaded to by the OS.)
    - Program code must be relocatable: use only machine code instructions based on relative addressing (if available) or adjust all absolute addresses during loading (load-time binding). (More on relocation on next slides.)
    - No memory protection: the whole address space can be accessed by the current process: process might overwrite OS and other processes and thus crash these.



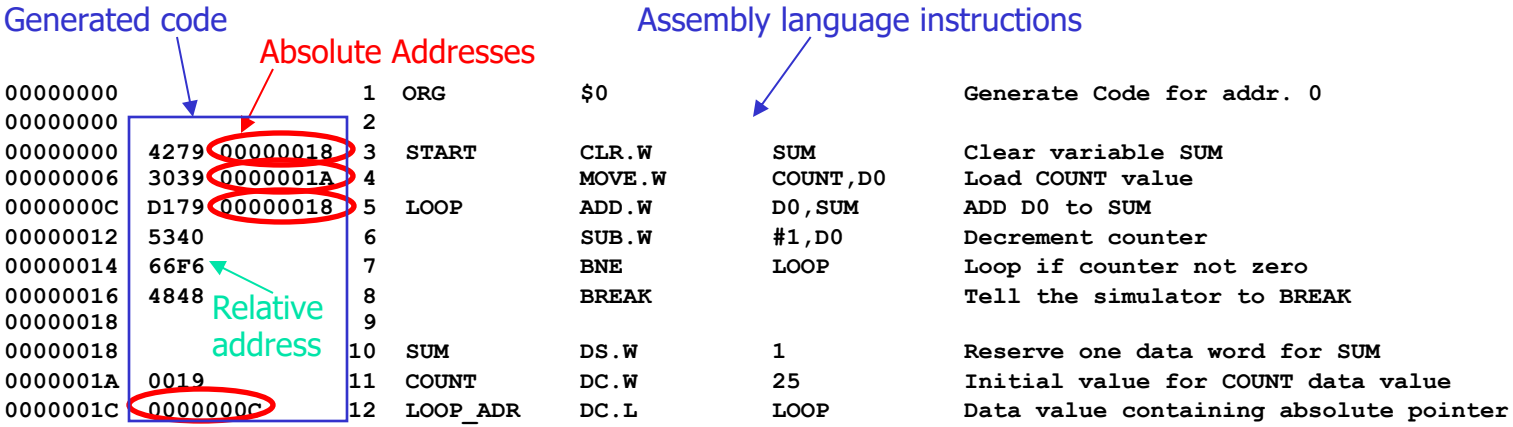
Excursion: Address Binding

- To run, the program must be brought into memory and placed within the context of a process.
- A user program goes through several steps – some of which may be optional – before being executed.
  - Addresses in the source program are generally symbolic (such as a variable **`count`**).
  - A compiler typically binds these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning").
  - The loader in turn binds the relocatable addresses to absolute addresses (such as 74014).



Excursion: Relocatable Code (Load-time Binding)

- Program code is created by compiler/assembler for start address 0 and a relocation table is added to the program file.
- Relocation table records each absolute address found in the program file:




```
00000000                1  ORG       $0                    Generate Code for addr. 0
00000000                2
00000000  4279 00000018 3  START     CLR.W     SUM         Clear variable SUM
00000006  3039 0000001A 4            MOVE.W    COUNT,D0    Load COUNT value
0000000C  D179 00000018 5  LOOP      ADD.W     D0,SUM      ADD D0 to SUM
00000012  5340          6            SUB.W     #1,D0       Decrement counter
00000014  66F6          7            BNE       LOOP        Loop if counter not zero
00000016  4848          8            BREAK                 Tell the simulator to BREAK
00000018                9
00000018               10  SUM       DS.W      1           Reserve one data word for SUM
0000001A  0019         11  COUNT     DC.W      25          Initial value for COUNT data value
0000001C  0000000C     12  LOOP_ADR  DC.L      LOOP        Data value containing absolute pointer
```

⇒ Relocation table contains address entries: 2, 8, E, 1C

- When loading a program, the OS program loader chooses a free memory location, loads the program code into that memory area, and adds the starting address of the chosen memory location to all locations containing absolute addresses listed in the relocation table.

Excursion: Static Linking (Compile-time Binding)

- In addition to its own instructions, a program typically calls instructions of a library.
  - E.g. the implementation of the C function `printf(…)` is contained in the C standard library (stdlib or libc).
- In the simplest case, the machine code instructions of all libraries needed by a program are linked to the program at compile time of that program (static linking).
  - Blows up size of executable program file on disk.
  - If multiple programs are in memory at the same time and these programs use the same library (which is likely for e.g. the C standard lib), memory is wasted, because all the programs have their own copy of the same library.
  - If an update of a library is published, the programs that have already been linked, do still use the old library.
    - Programs need to be re-linked using the new libraries.
    - (Re-linking requires the intermediate `*.o` object files of the program. I.e. re-linking not possible if just the executable program file is available.)

Excursion: Dynamic Linking / Dynamic-link Libraries (DLL)

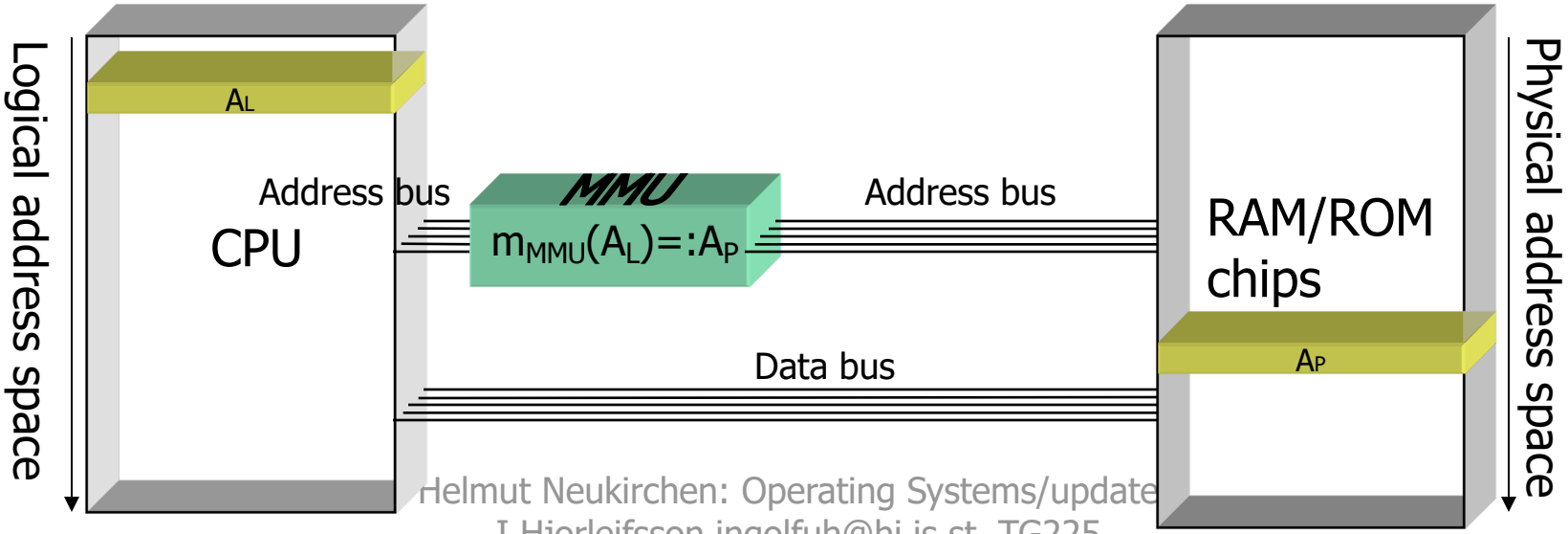
- Dynamic linking of libraries avoids the problems of static libraries:
- Linking occurs just at execution time:
  - At compile time (to be precise: at link time), only a small piece of code (a stub) that represents the library is linked to the program.
  - When a call to a library function is made by a program, the stub is called:
    - Stub asks OS whether library is already in memory: if not, OS loads it from a special library location from the file system (POSIX: /usr/lib) into a free memory location (to enable a library to run at any location, it must be relocatable).
    - Once library is in memory, stub locates memory location of the desired function in the dynamic library (using a symbol table) and calls that function.
    - To speed up further look-ups of the same library function, the memory location returned by the first lookup may be used next time to jump directly to the memory location of the desired function.

  ⇒ Size of executable program file is reduced (library exists only once in the file system); programs do not need to be re-linked to use a new library.

  ⇒ By sharing loaded libraries between all the running processes (shared library), also the main memory footprint can be reduced.

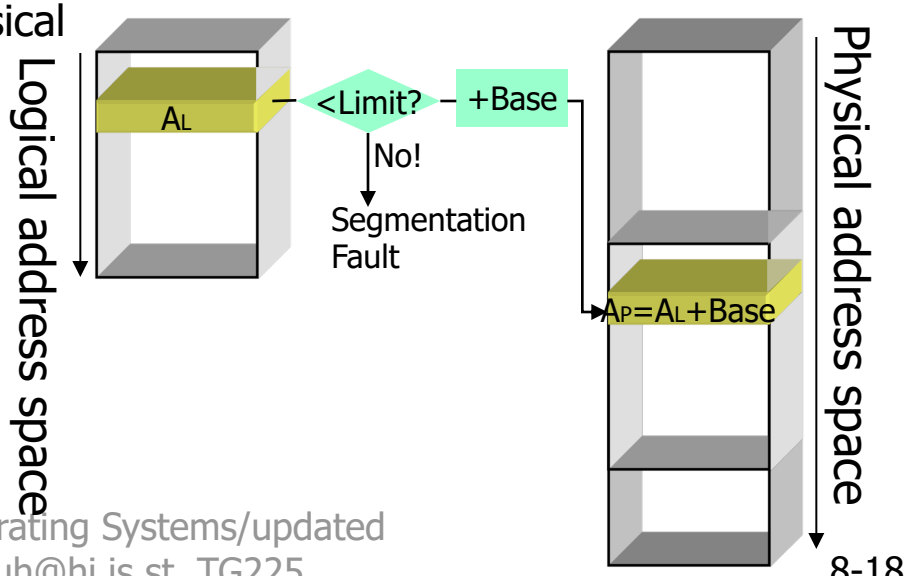
Organisation of Memory: MMUs & Logical and Physical Addresses

- A programmable Memory Management Unit (MMU) may be used to translate CPU addresses into addresses of the physical main memory:
  - MMU may be part of the CPU (e.g. in today's PCs) or a separate chip, or it may be completely absent (e.g. on cheap embedded systems).
  - Mapping from logical to physical addresses: m_{MMU}: $A_L \rightarrow A_P$
    - physical address (A_P): real address in physical memory (view of MMU).
    - logical address (virtual address) (A_L): address used by a program (view of CPU).



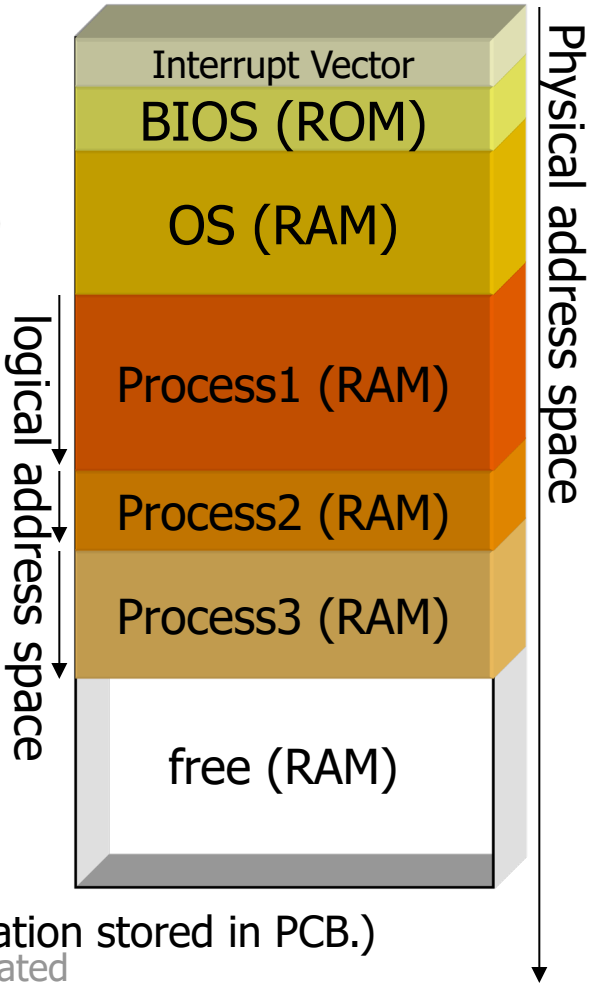
Organisation of Memory: Basic MMU Hardware

- MMU translates every logical address accessed by the CPU to a physical address.
- Most basic type of a programmable MMU: Segmenting MMU (obsolete, but Intel CPUs still support it).
  - Can be programmed using two registers:
    - Base register: physical address of logical address 0,
    - Limit register: maximum allowed logical address.
    - Segmenting MMU generates physical address by adding base register to logical address as long as it is within the limit: $A_{P:}=A_L+Base$
    - If logical address is outside limit: Segmentation Fault interrupt is raised by MMU.
      - OS interrupt handler then typically terminates process that has gone wild.



Organisation of Memory: MMU-supported Multiprogramming

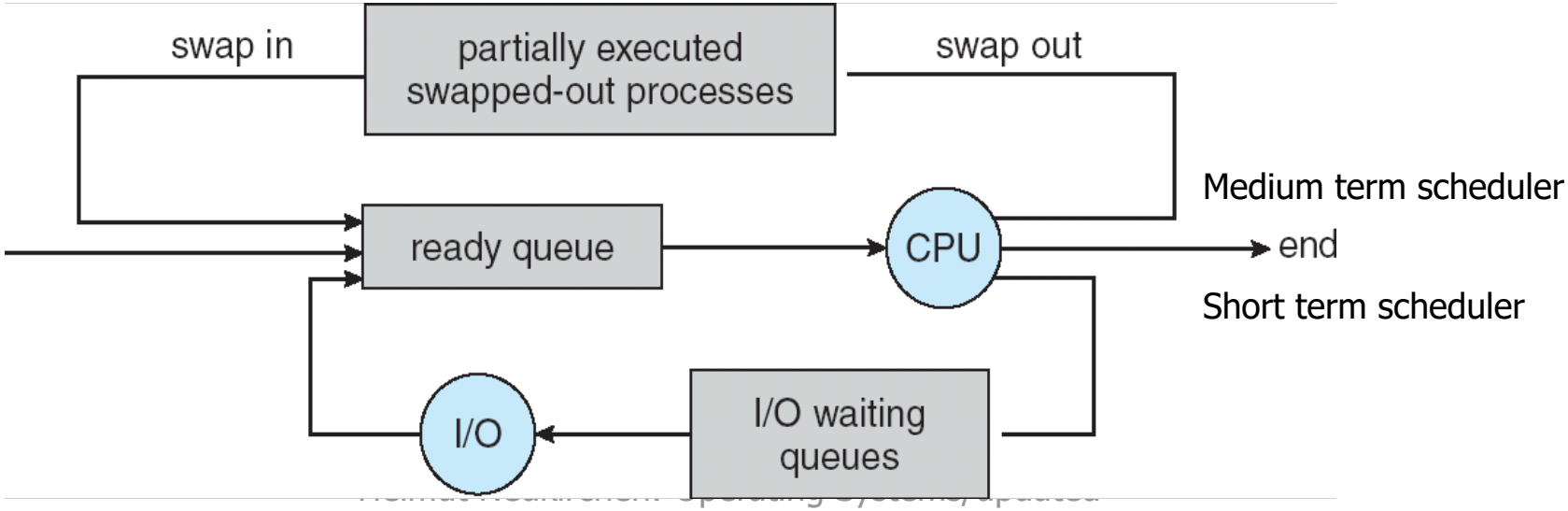
- Multiple processes, each having its own logical address space.
  (e.g. POSIX, MS Windows since Windows NT)
  - MMU is reprogrammed at each context switch to give each process its own logical address space.
    - No relocation required, as each process has it's own address space (each starting with logical address 0); hence all programs can be compiled for the same absolute memory location.
    - Memory protection: memory of other processes or of the OS not accessible. (When trying to access logical memory that is not allocated to current process: Segmentation Fault interrupt generated by MMU.) At each context switch, MMU is reprogrammed according to physical memory owned by current process!



(Address space information stored in PCB.)

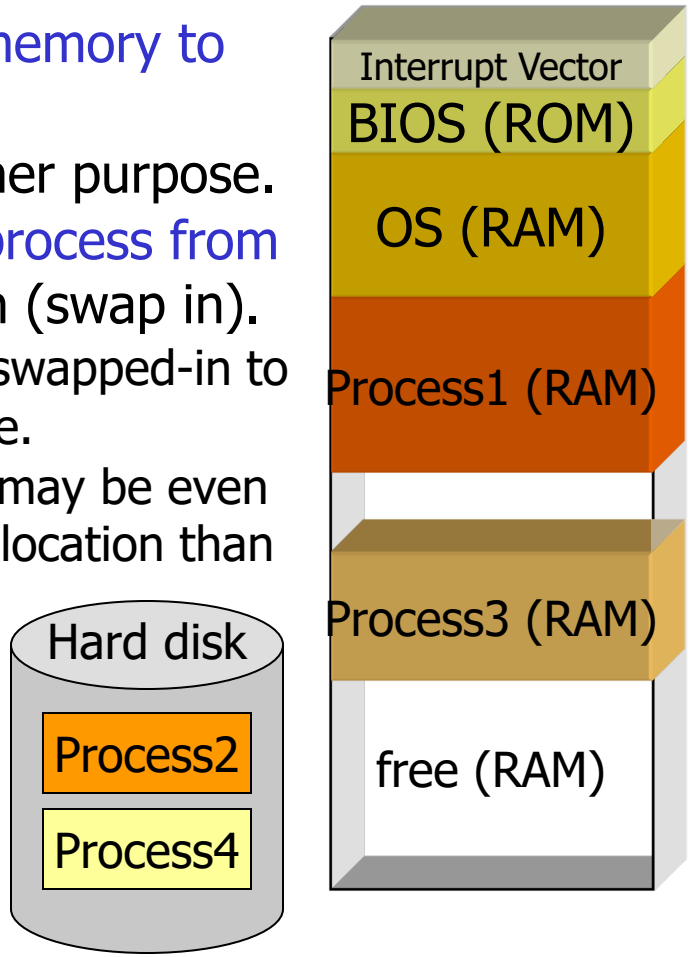
8.2 Swapping: Reminder Chapter 3 Medium Term Scheduling

- Sometimes, degree of multiprogramming needs to be temporarily reduced.
  - E.g. memory is filled up by processes and now, one of these processes requires further memory. The only solution is to (temporarily) remove one of the processes from memory.
  - A medium term scheduler may be used to identify processes that should not be considered by short-term scheduler for some time:
  - Processes are removed from main memory to hard disk (swapped out) and later-on, e.g. if another process terminates, put into main memory again (swapped in).



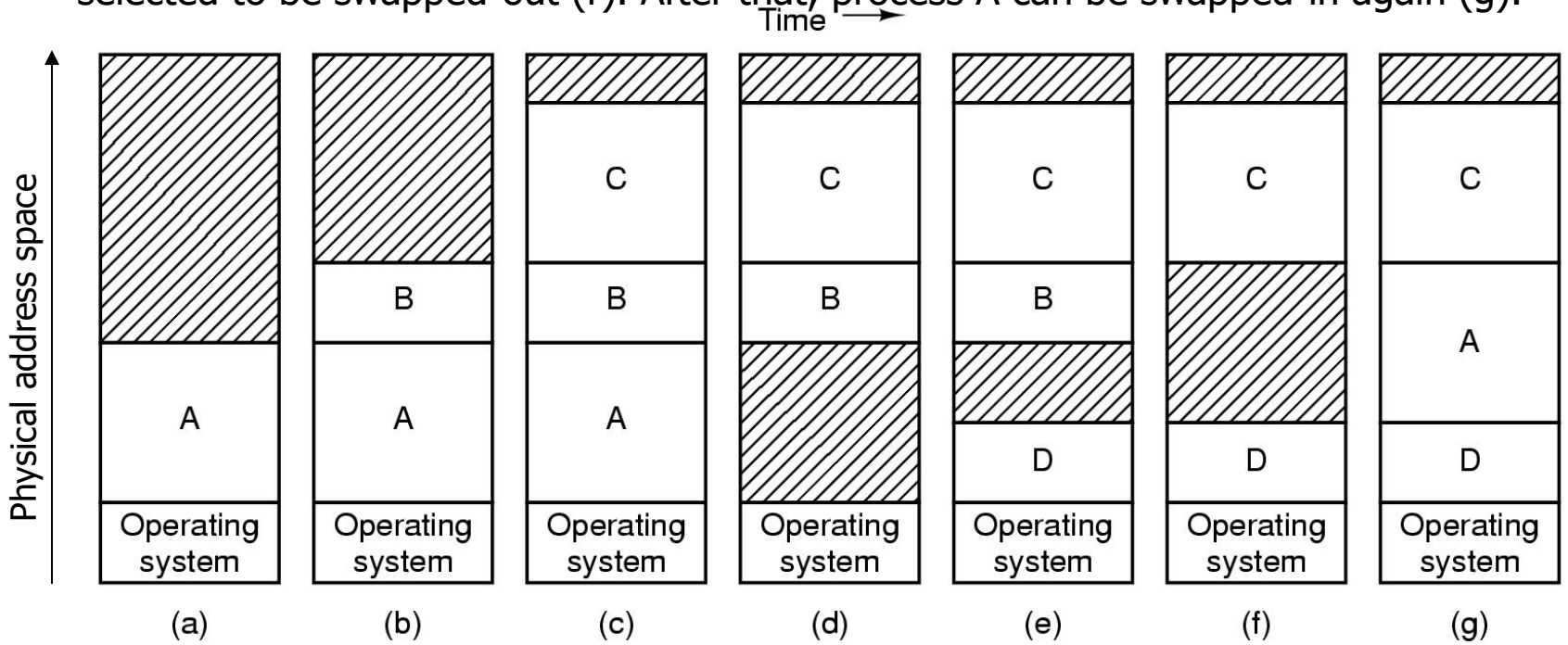
Swapping

- Transfer a process as a whole from main memory to hard disk (swap out).
- Use the gained main memory for some other purpose.
- Later, re-transfer the swapped-out whole process from hard disk to memory to continue execution (swap in).
  - If no MMU is available: processes must be swapped-in to exactly the same memory location as before.
  - If segmenting MMU is available: processes may be even swapped-in to a different physical memory location than before:
    - By reprogramming the base register, the MMU adapts the logical addresses to the new physical addresses, hence a process may just continue execution using the unchanged logical addresses.
- Today, swapping is not used very much, as it is inefficient to swap out whole processes. (Instead: Paging →8.4)



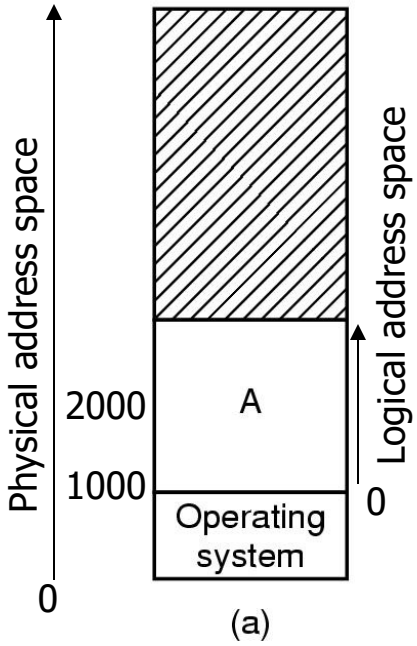
Swapping: Example using MMU

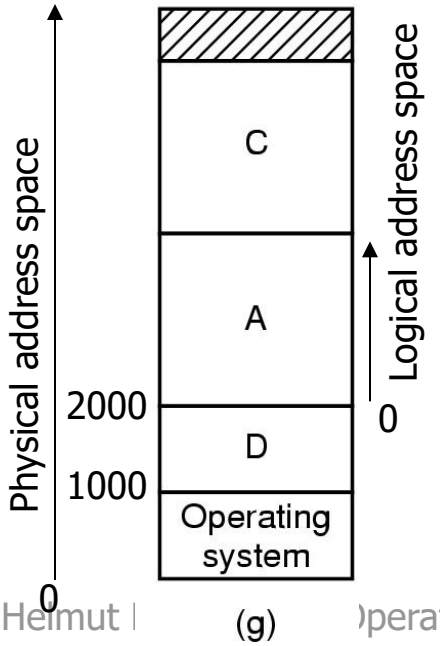
- First (a), process A is started. Then (b), process B is started and (c) process C is started.
- Next, process D shall be started. However, no sufficient space is left in the physical memory, hence e.g. process A is selected to get swapped-out (d). After that, process D can be started (e).
- Next, process A shall be run again. To make space for process A, e.g. process B is selected to be swapped-out (f). After that, process A can be swapped-in again (g).



Swapping: Example using MMU Logical vs. physical addresses

- Assume, before swapping out (a), logical address 0 of A was at physical address 1000.
- After swapping in again (g), logical address 0 of process A might then be at physical address 2000.
- A program knows only about logical addresses, hence the MMU needs to achieve that in step (a) logical address 0 is mapped to physical address 1000, but in step (g) logical address 0 is mapped to physical address 2000.





- In case of a segmenting MMU:
- Step (a): Base=1000
- Step (g): Base=2000
- Every time a memory location is accessed by the CPU, the MMU does the address translation.

8.3 Contiguous-Memory Allocation

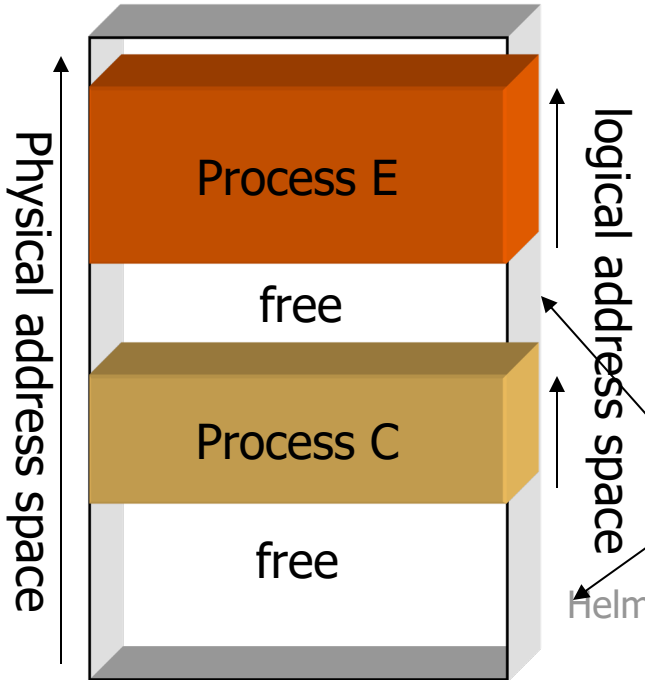
- As we have seen on the previous slides, there may be "holes" in the memory, i.e. free memory blocks next to allocated memory blocks.
  - As instructions of a program need to be in consecutive memory locations (the compiler keeps the code as compact as possible, i.e. consecutive instructions), it is not possible to gather multiple scattered smaller free memory locations to satisfy a request for a larger block of memory.
    - (At least, not with a segmenting MMU. – We will later-on learn more about paged MMUs that are more flexible.)

Contiguous-Memory Allocation and Release

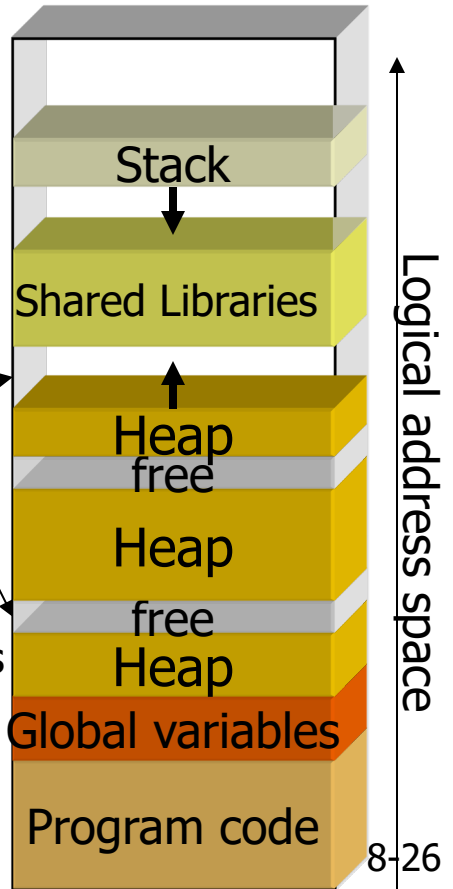
- OS manages:
  - 1) Memory for whole processes,
  - 2) Memory within a process.
- Situations where OS has to allocate free contiguous memory holes:
  - Physical memory (simple multi-programming only): When a new process is started or when a swapped-out process is swapped in.
  - Logical memory: During run-time of a process: process may dynamically request memory (e.g. in Java or C++ using **`new`**, in C using the **`malloc`** C library call) from the "heap" that has been reserved for the process.
- Situations when allocated memory is released:
  - Physical memory: Processes termination. When a process is swapped out.
  - Logical memory: Release of heap memory (C++: **`delete`**, C/Unix: **`free`**).

Contiguous-Memory Allocation Visualised

- Programs need to be loaded to contiguous logical memory locations.
  - At least with no MMU or with segmenting MMU this means: contiguous physical memory locations.

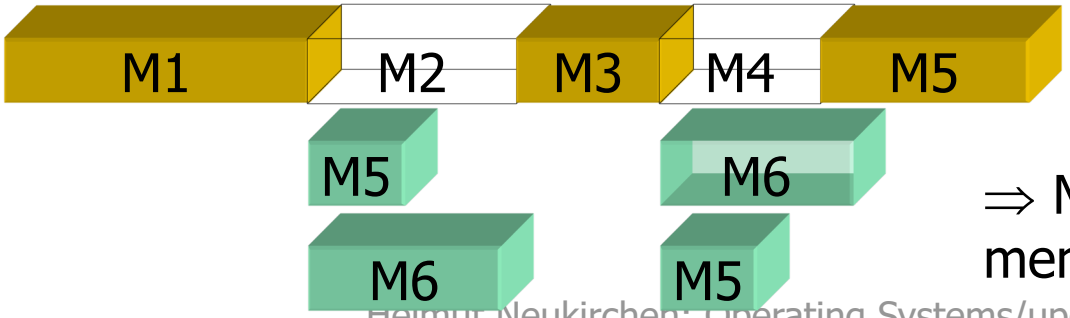


- Process may during runtime request further memory from the heap of their logical address space.
  - After a couple of requesting and releasing memory from the heap, the heap may look like a Swiss cheese: a lot of holes.
  - From these holes, a contiguous logical memory area will be needed to satisfy the next request of contiguous memory from the heap.



Problem of Contiguous-Memory Allocation

- When multiple holes are available to satisfy a memory allocation request, OS has to decide on which hole to use.
- Example:
  - Memory locations M2 and M4 have been released.
  - Now, requests for M5 and M6 are pending: which hole to chose for M5, which one for M6? ("Dynamic storage-allocation problem")
    - If we would put M5 in the hole of M2, this would satisfy M5. However, M6 cannot be satisfied anymore!
    - Putting M5 into hole of M4 and M6 into M2 would work, though.



⇒ Management of dynamic memory allocation required!

Problem of Fragmentation

- Typically, memory allocation requests do not fill exactly a memory hole: some part of the hole is used for the allocation request, the remainder becomes a smaller hole again:

⇒ As a result, memory contains after some time many small holes:

- External fragmentation: Free memory that is, however, fragmented into many non-contiguous holes.
  - Even if the sum of the holes would satisfy a memory request, the memory is non-contiguous and can thus not satisfy the request for contiguous memory. (Solution if MMU is available to relocate whole processes in physical memory at run-time: shift them so that one large block free remains (compaction).)
- Internal fragmentation: Free memory within the memory allocated to a process, that cannot be used to satisfy requests of other processes.
  - The MMU enforces memory protection, hence internally fragmented memory cannot be accessed by (and thus not allocated to) other processes.

⇒ Dynamic memory allocation strategies should minimise fragmentation.

Dynamic Memory Allocation Strategies

- Many strategies possible, e.g.:
  - Best Fit: Allocate the smallest hole that is big enough.
    - Produces the smallest leftover holes: sounds nice in theory. However, practise shows that the resulting leftover holes are so small that they can typically not be used to satisfy any future requests (fragmentation).
  - Worst Fit: Allocate the largest hole.
    - Produces the biggest leftover holes: sounds like a remedy for the problem of Best Fit. However, practise shows that huge holes are quickly degraded into smaller ones that cannot satisfy average sized future requests (fragmentation).
  - First Fit: Always search from the beginning of the free holes list to allocate the first hole that is big enough.
    - Small holes tend to accumulate at the beginning of the free holes list, making the search for bigger holes farther and farther (=slower) each time.
  - Next Fit: Like First Fit, however search for memory hole starts where last request has been satisfied.
    - (I.e. if hole used to satisfy last request has not been completely occupied by last request, that resulting smaller hole is used as starting point for next request.)
    - Eliminates the problem of First Fit.

Dynamic Memory Allocation Strategies: Example

- Holes of free memory are available in the following order: 10 KB, 5 KB, 14 KB, 9 KB, 33 KB and 25 KB. Now, contiguous-memory m_i of size m1=13 KB, m2=6 KB, m3=2 KB, m4=17 KB and m5=18 KB is requested in consecutive order.

Holes:	10 KB	5KB	14 KB	9 KB	33 KB	25 KB
First Fit	m2=6 KB		m1=13 KB		m4=17 KB	m5=18 KB
	m3=2 KB		1 KB		16 KB	7 KB
	2 KB					
Next Fit			m1=13 KB	m2=6 KB	m4=17 KB	m5=18 KB
			1 KB	m3=2 KB	16 KB	7 KB
				1 KB		
Best Fit			m1=13 KB	m2=6 KB	m5=18 KB	m4=17 KB
			1 KB	m3=2 KB	15 KB	8 KB
				1 KB		
Worst Fit					m1=13 KB	m2=6 KB
					m3=2 KB	m4=17 KB
					m5=18 KB	2 KB

8.4 Paging

- While the basic segmenting MMU supports relocation of processes and memory protection, only one contiguous block of physical addresses can be mapped by a segmenting MMU to a process.
  - ⇒ Requests for process memory must be satisfied using contiguous physical memory.
  - ⇒ Only whole processes (not parts of it) may be swapped out and in.

⇒ Segmenting MMUs have been superseded by Paged MMUs.

Pages and Frames

- Instead of mapping contiguous blocks of logical and physical memory, Paged MMUs divide the logical memory into multiple fixed-sized pages that can be mapped onto physical frames of the same size.
  - While the memory within each page and frame must be contiguous, the consecutive pages of a process may be mapped on frames scattered in the physical memory.
    - ⇒No external fragmentation.
    - However still internal fragmentation due to fixed size of pages/frames.
  - ⇒Even parts of a process may be stored to and retrieved from a hard disk.

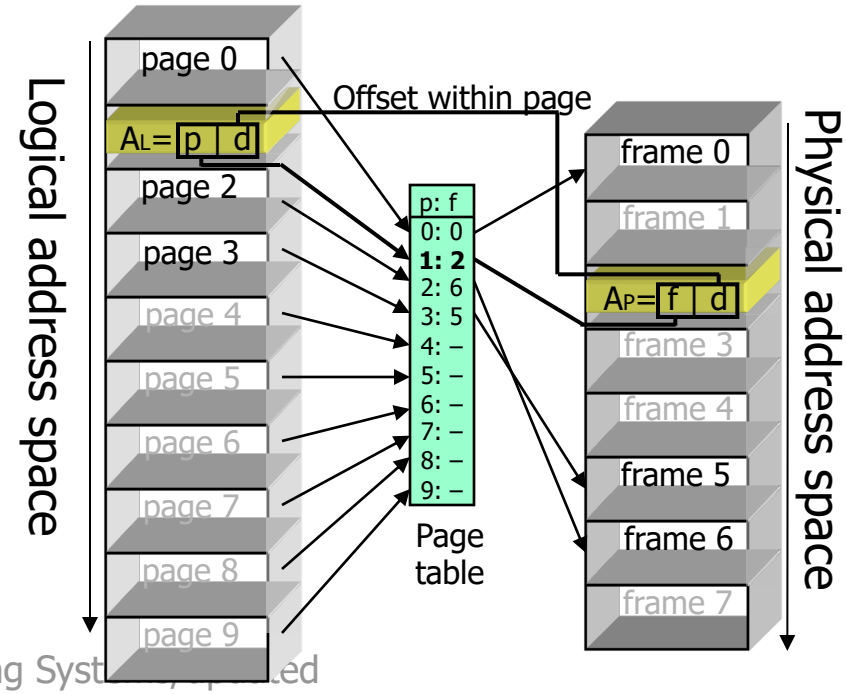
Advanced MMU Hardware: Paged MMUs

- More flexible: Paged MMU (PMMU).
  - Logical address space is divided into pages of fixed size.
  - Pages are mapped onto frames in the physical memory using a page table.
  - Pages/frames have same size that is a power of 2, i.e. page size=2^n.
  - Logical address A_L (having m bits) generated by CPU is divided into:

m-n bits	n bits
Page number *p*	Page offset *d*

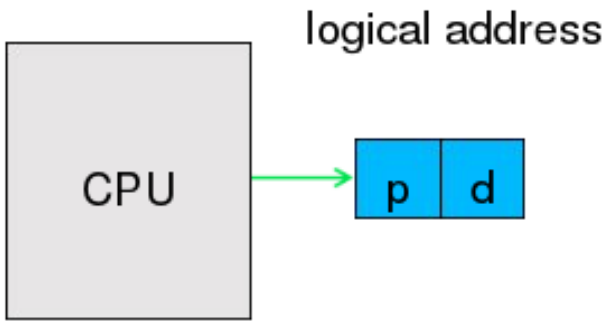
  - By looking up the frame *f* onto which page *p* is mapped according to the page table, the resulting physical address A_P (m bits) is:

m-n bits	n bits
frame number *f*	Page offset *d*



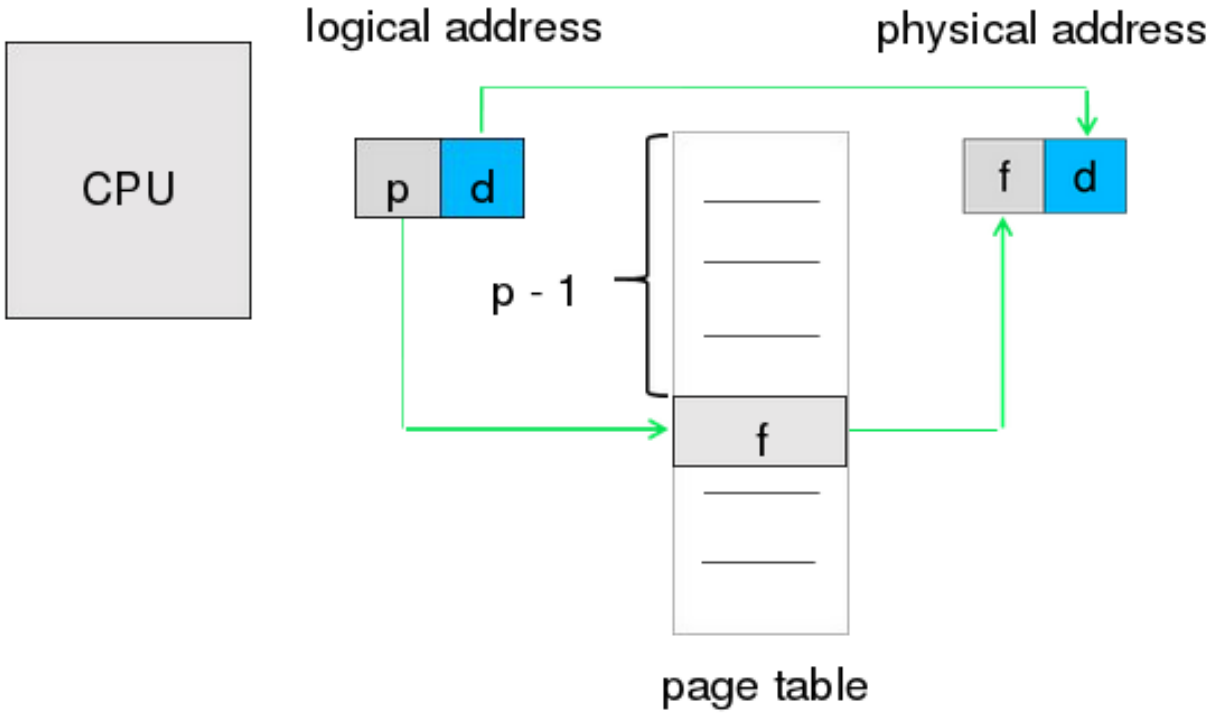
PMMU Address Translation Step 1

1. the logical address generated by the CPU is sent to the MMU where it is divided into a page number (p) and an offset (d)
   – the number of bits in each part depends on the page size



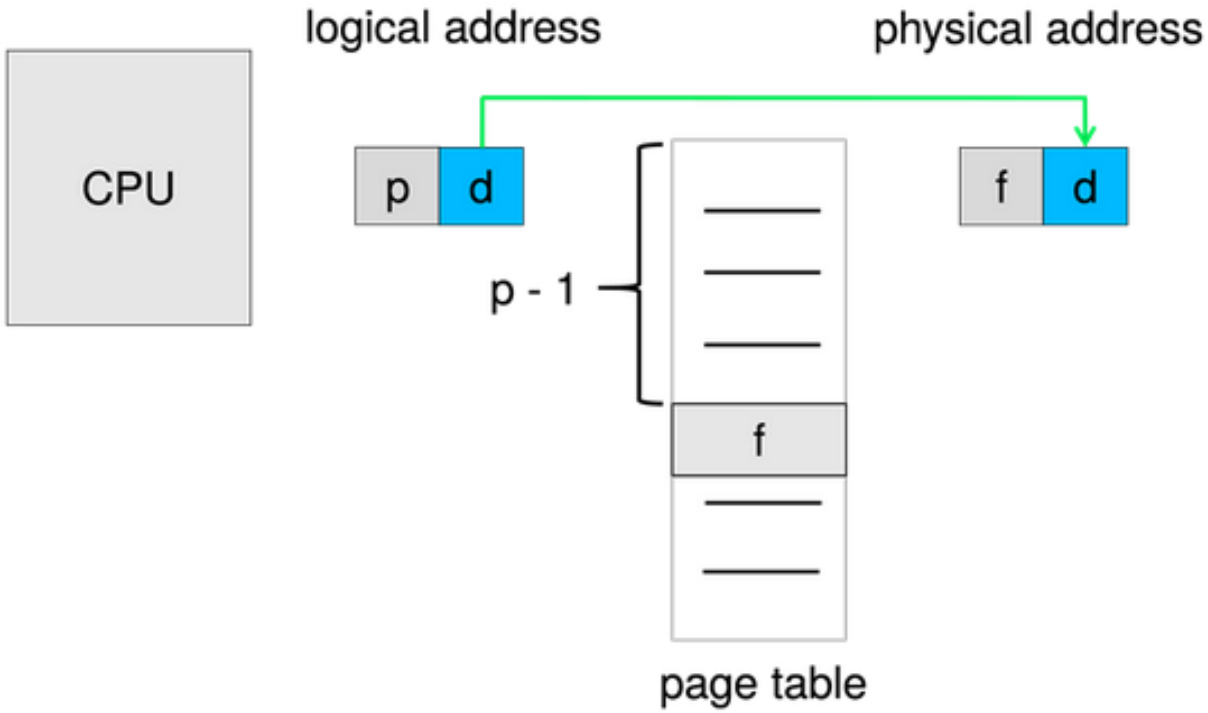
PMMU Address Translation Step 2

2. the page number is used as an index into the page table
   – the entry in the page table at that index is the number of the frame of physical memory containing the page
   – that frame number is the first part of the physical memory address



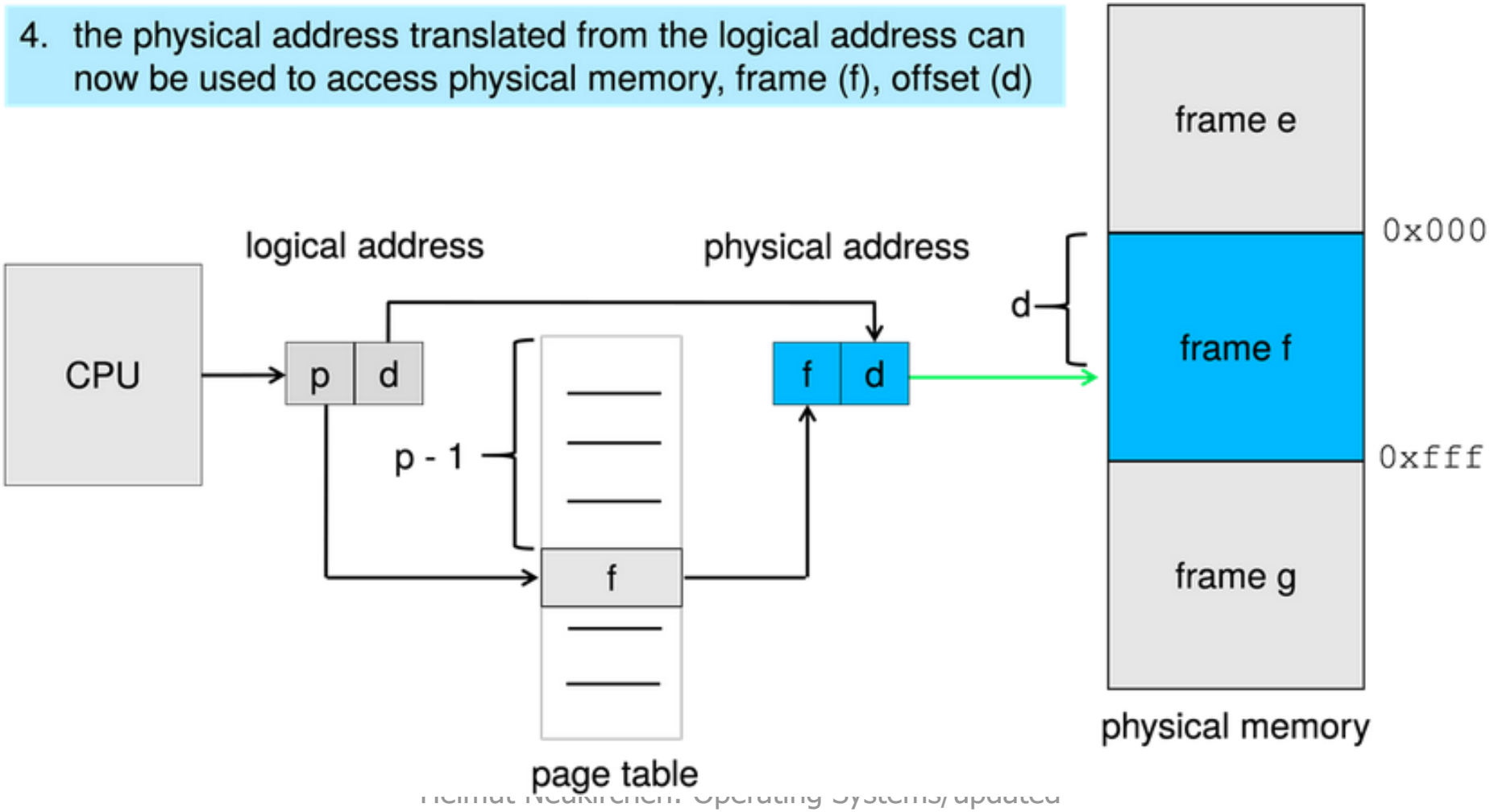
PMMU Address Translation Step 3

3. the offset (d) within the page is the same as the offset within the frame, so it is retained, together with the frame number forming the physical address



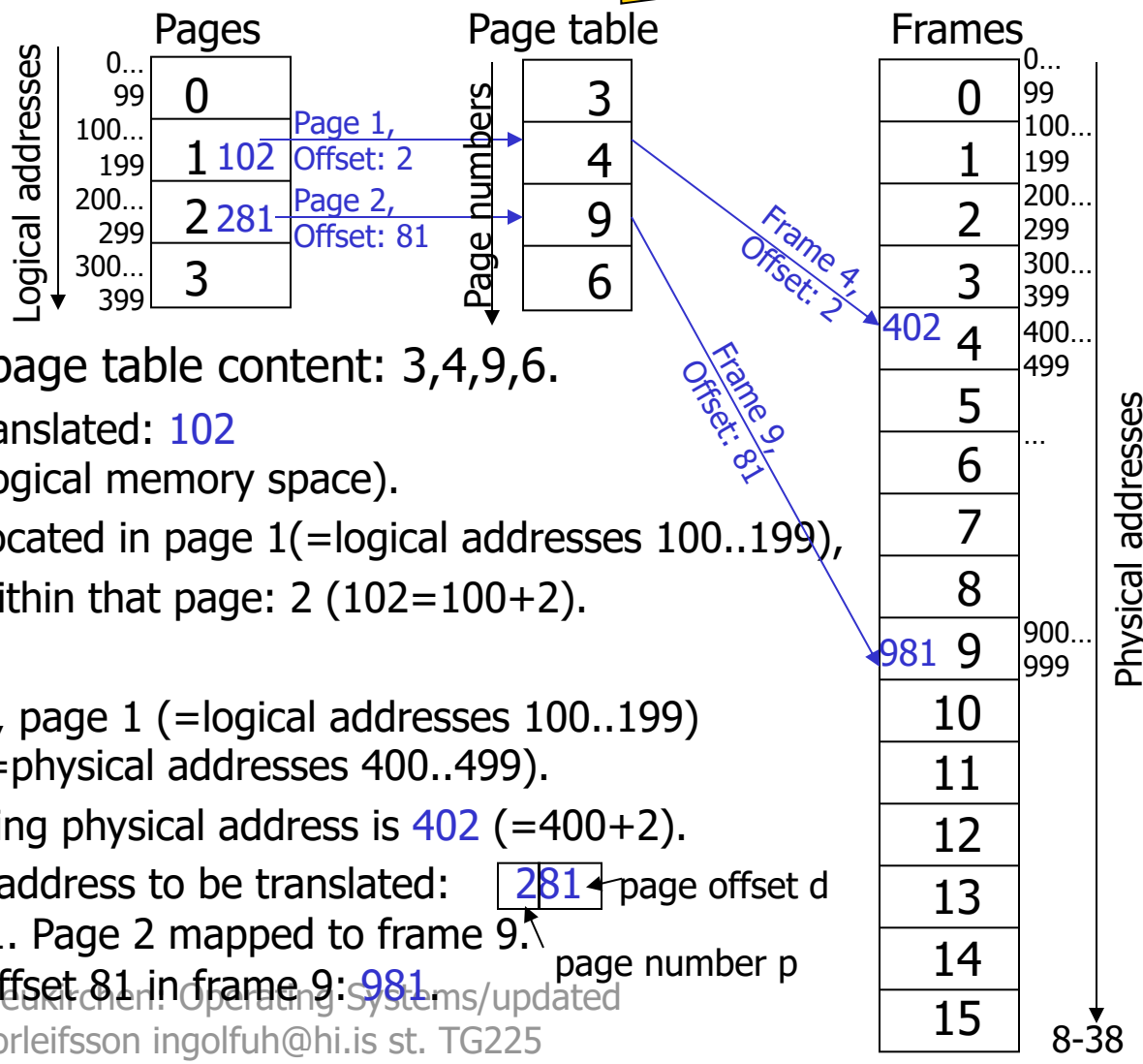
PMMU Address Translation Step 4

4. the physical address translated from the logical address can now be used to access physical memory, frame (f), offset (d)



Example address translation by PMMU

Each process has it's own page table (stored in PCB; PMMU's pointer to page table location is re-programmed as part of context switch).



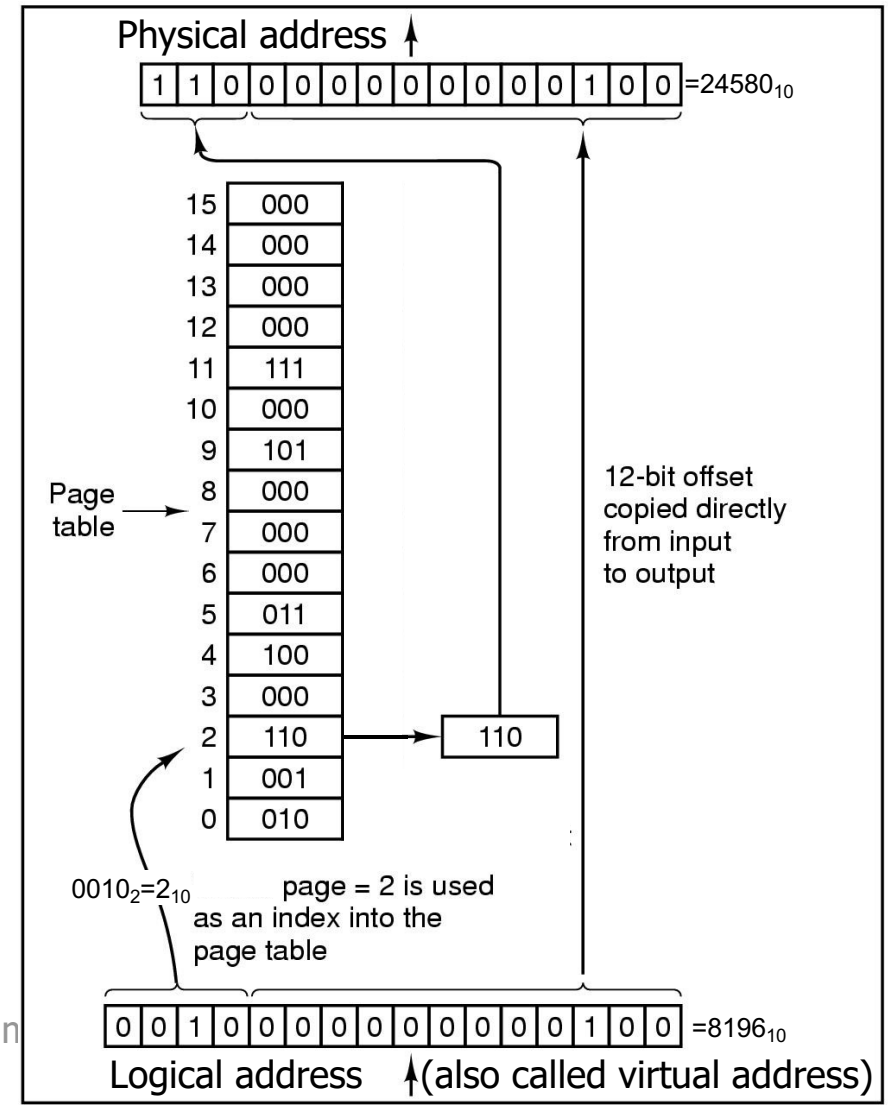
- For simplicity, assume page sizes that are a power of 10 (not 2 as in real hardware).
- Example: page size =frame size=100 bytes, page table content: 3,4,9,6.
  - Logical address to be translated: 102 (=byte number 102 in logical memory space).
  - Logical address 102 is located in page 1(=logical addresses 100..199),
  - Offset of address 102 within that page: 2 (102=100+2).

page number p → 1 | 02 ← page offset d

  - According to page table, page 1 (=logical addresses 100..199) is mapped to frame 4 (=physical addresses 400..499).
  - With offset 2, the resulting physical address is 402 (=400+2).
  - Other example: Logical address to be translated: 281 ← page offset d
    281 → page 2, offset 81. Page 2 mapped to frame 9.
    → Physical address of offset 81 in frame 9: 981. ← page number p

Advanced MMU hardware: Inner life of a Paged MMU

- Upper *m-n* bits (=page number *p*) of logical address used as index into page table.
- Entry of page table (=frame number *f*) replaces upper *m-b* bits of address.
- Lower *n* bits (=page offset *d*) of address copied without modification.
- Resulting address is used as physical address.



Note:
In this example, logical address space is bigger than physical address space! (Whereas on slide 8-34, logical address A_L and physical address A_P had the same number of bits.)

PMMU: Page Tables

- Today's PMMUs support page sizes between 256 B and 2 GB.
  - Most operating systems use 4 KB or 8 KB page size (Solaris even 4 MB).
- Structure of a page table entry: 4 Byte on a 32 bit system (assuming a 32 Bit address space and 256 Byte page size):

Page was accessed (MMU will automatically set it at read or write access.)

Page has been modified (MMU will set it automatically if CPU writes to that page.)

Page may only be read

Page is allowed to be accessed (would be set to invalid for memory protection or to indicate that no corresponding frame is in physical memory). Access will result in Page Fault interrupt that is serviced by OS interrupt handler.

- On 64 bit systems, the physical address space is bigger and thus more than 2^{24} frames may exist.
  - Pages table entry is 8 Byte allowing more bits for the frame numbers.

PMMU: Translation Lookaside Buffer TLB

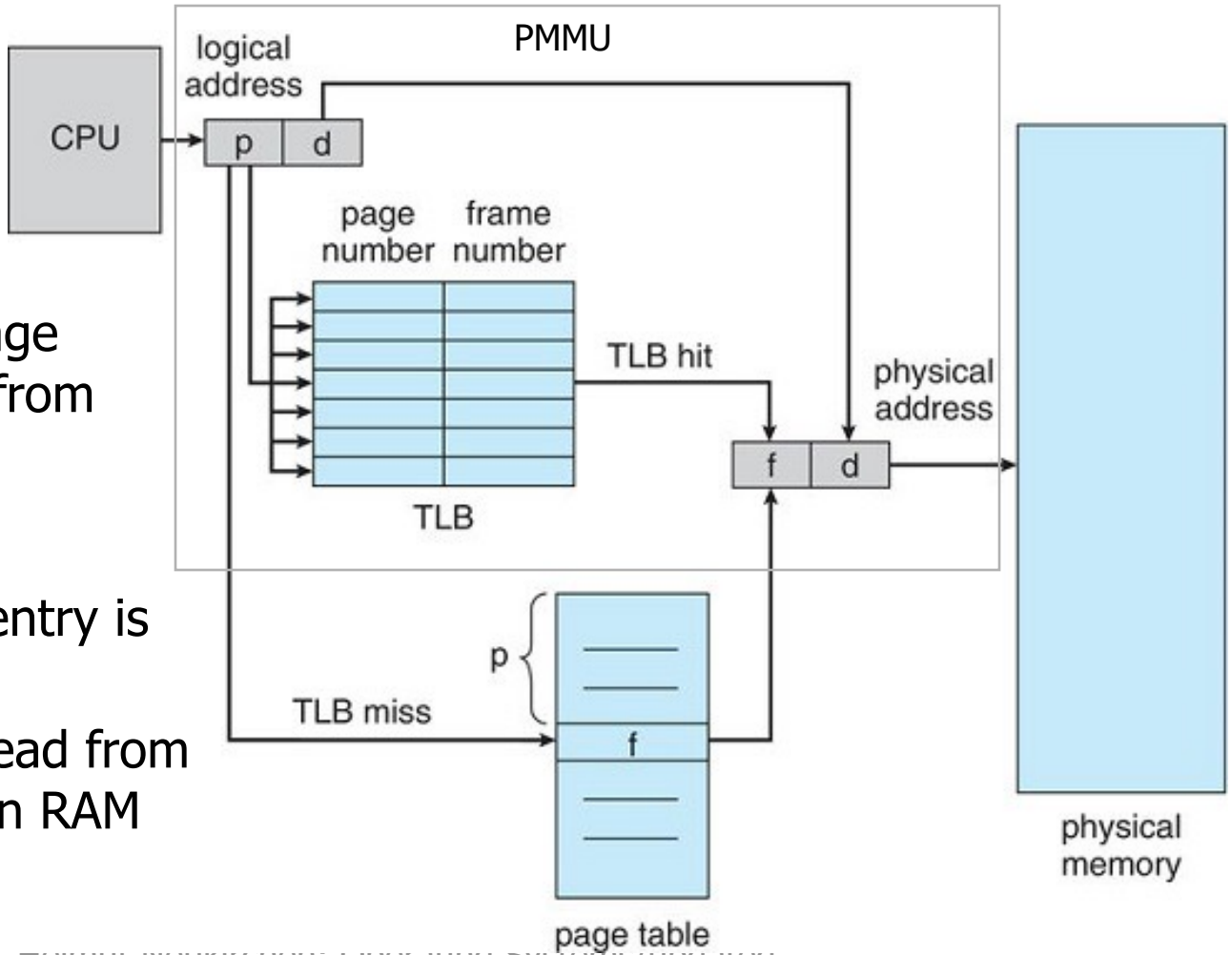
- Page table can be huge and thus cannot be kept in PMMU-internal registers, but rather in the RAM.
  - The MMU is anyway in charge of managing the RAM, so no problem to access RAM in order to read page table.
- Memory access slowed down by factor of 2 due to access to page table:
  1. Access to page table that is located in RAM,
  2. Access to final physical address.

$\Rightarrow$ Cache for page table entries (Translation Lookaside Buffer TLB):

- If page table entry is in TLB: only 1 RAM access needed (+ fast TLB access).
  - Memory access with PMMU and TLB almost as fast as without any PMMU.
- Problem: Context switch.
  - Each process has its own page with completely different page table entries.
  - TLB/PMMU need to keep page table entries from each process apart.
    - Each process has an address-space identifier that is stored in each TLB entry.

PMMU with TLB

- TLB hit:
  - Page table entry is in TLB.
  - Can read page table entry from fast TLB.
- TLB miss:
  - Page table entry is not in TLB.
  - Additional read from page table in RAM needed.



PMMU: Size of Page Tables

- Example: 4KB ($=2^{12}$B) page size, 32 Bit logical address space:
  - Address spaces consists of $2^{32}/2^{12}=2^{20}\approx 1$ million pages. Hence, page table needs 1 million entries, too.
  - Each page table entry is 4 Byte $\Rightarrow \approx$4 MB for page table!
  - A typical translation lookaside buffer (TLB) has place for caching between 64 and 1536 entries.

  $\Rightarrow$ Only a small fraction of the page table would be cacheable by the PMMU! (Even worse with 64 Bit logical address space.)

Multiply by 2^{32} for a 64 Bit system!

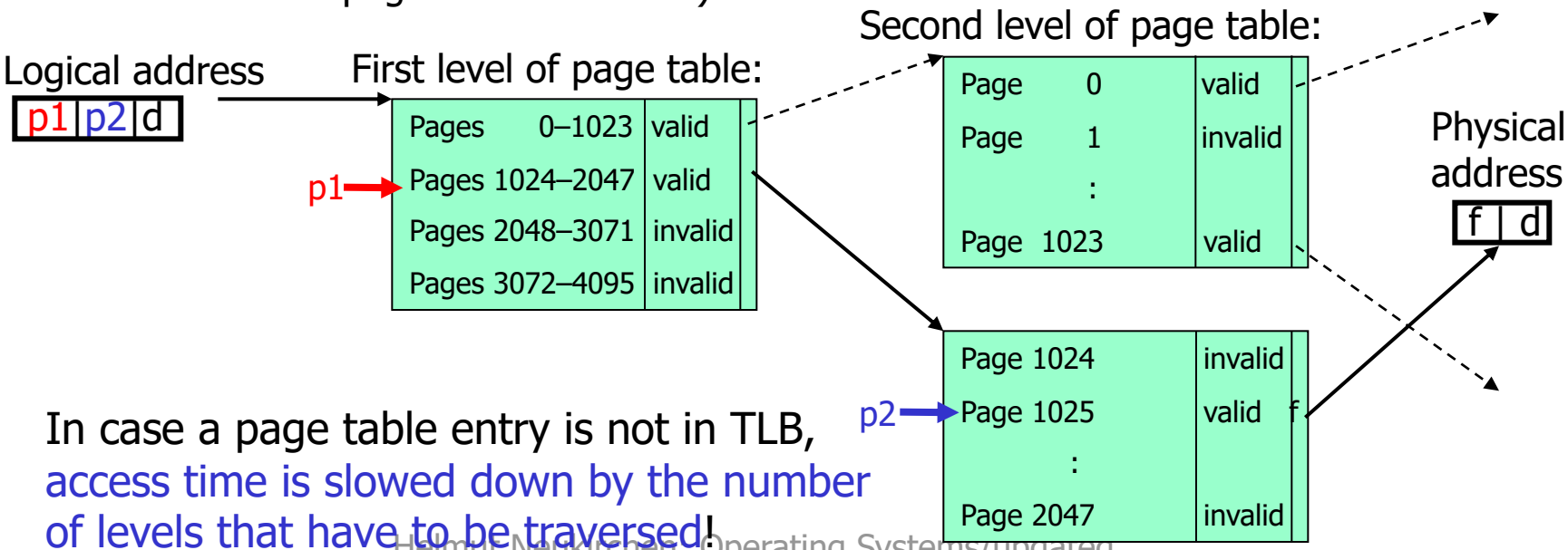
- Luckily, the logical address space is almost never used completely; instead, there are many contiguous unused areas for which the valid bits of their corresponding page table entries would be set invalid.

  $\Rightarrow$ Divide the page table into sections representing a contiguous set of pages: represent a section that contains only invalid pages by only one page table entry that indicates that all pages represented by this entry are invalid.

  $\Rightarrow$ Multi-level page tables ($\rightarrow$next slide).

Hierarchical Paging: Multi-level Page Tables

- Example: Two-level page table.
  - Frame number of a first level page table entry points to a frame containing the second level page table for all the pages represented by the first level page table entry or
  - First level page table entry is set to invalid to indicate that all the pages represented by the first level page table entry are invalid (in this case, no frame containing second level page table is needed).



- In case a page table entry is not in TLB, access time is slowed down by the number of levels that have to be traversed!

Paging on 64 bit systems

- With modern 64 bit CPUs, it is even more likely that huge areas of the logical address space are unused (=set to invalid in the page tables entries).
- Typically, at least 4 levels multi-level page tables used by 64 bit OSes (and supported by PMMUs of 64 bit CPUs).
  - May lead to a 4 times slowdown of access times if page table entry is not in TLB (or more if more than 4 levels are used).
  - Trends for PMMUs in 64 bit non-Intel CPUs:
    - Hashed page tables, Inverted page tables.
      - Avoid huge page tables and slowdown due multi-level lookup times.
      - Hashing techniques used instead.
      - Details not covered here.

Applications of Paging: Solve Fragmentation, Memory Protection

- A PMMU can be used to implement:
  - Getting rid of external fragmentation:
    - A free frame can be allocated to the contiguous logical address space of any process. (However, internal fragmentation due to fixed page size remains.)
  - Memory protection using write-protect & (in)valid flag of each page:
    - Invalid: To indicate memory that belongs to another process or is not available at all,
    - Write protect: Read-only memory,
    - Any other (="normal") page: Read and writable.
    - Switch PMMU page table pointer at each context switch.
      - Page table of each process ("address space") stores process' PCB.
  - Virtual memory: see chapter 9…
  - Shared memory: see next slides…

Applications of Paging: Shared Memory

- Shared memory=Map pages of different processes to the same frame:
  - Shared libraries:
    - Dynamic link library is loaded into physical memory only once and mapped (with read-only protection) into logical address space of all processes. ⇒ Reduces physical memory footprint of processes using this library.
  - Starting the same program multiple times:
    - OS detects that a program file has already been loaded to physical memory, hence frames containing program's instructions are mapped (with read-only protection) into logical address space of new processes.
  - Process communication using shared memory (→chapter 3):
    - Map shared memory frame into logical address space of all processes that want to share memory.
    - Reminder from slide 3-32: POSIX system calls for shared memory, e.g.:
      - `int shmget (long key, int size, int flag)`: create a new shared memory area (=frame) or retrieve shared memory handle (will be the return value) using a `key` (on which applications have to agree on).
      - `void* shmat (int id, char* addr, int flag)`: map shared memory area to page with `addr` using handle `id` to identify previously created area/frame.

8.5 Summary

- Different memory-management hardware (no MMU, segmenting MMU, paged MMU) support different memory organisation (from monoprogramming to advanced multiprogramming).
  - The resulting memory-management strategies can be compared based on:
  - Required hardware support: PMMU allows more sophisticated strategies than segmenting MMUs that are still better than no MMU.
  - Performance: MMUs add a performance penalty to each memory access. While segmenting MMUs are fast, PMMUs are slow; however, a TLB reduces this problem.
  - Fragmentation: Fixed-size allocation units (pages) suffer from internal fragmentation, one variable-size allocation unit (segmenting) suffers from external fragmentation.
  - Relocation: MMUs support relocating code, thus enabling compaction to remove external fragmentation (not an issue with PMMU).
  - Swapping: Whole processes are copied to hard disk and back to allow running more processes than can be fit into memory at one time.
  - Sharing: Sharing of pages between processes allows to fit more processes into memory, thus increasing the multiprogramming degree.
  - Protection: PMMUs support pages to be invalid, read-only, read-write; thus, enabling memory protection and sophisticated strategies, e.g. copy-on-write (→next chapter).