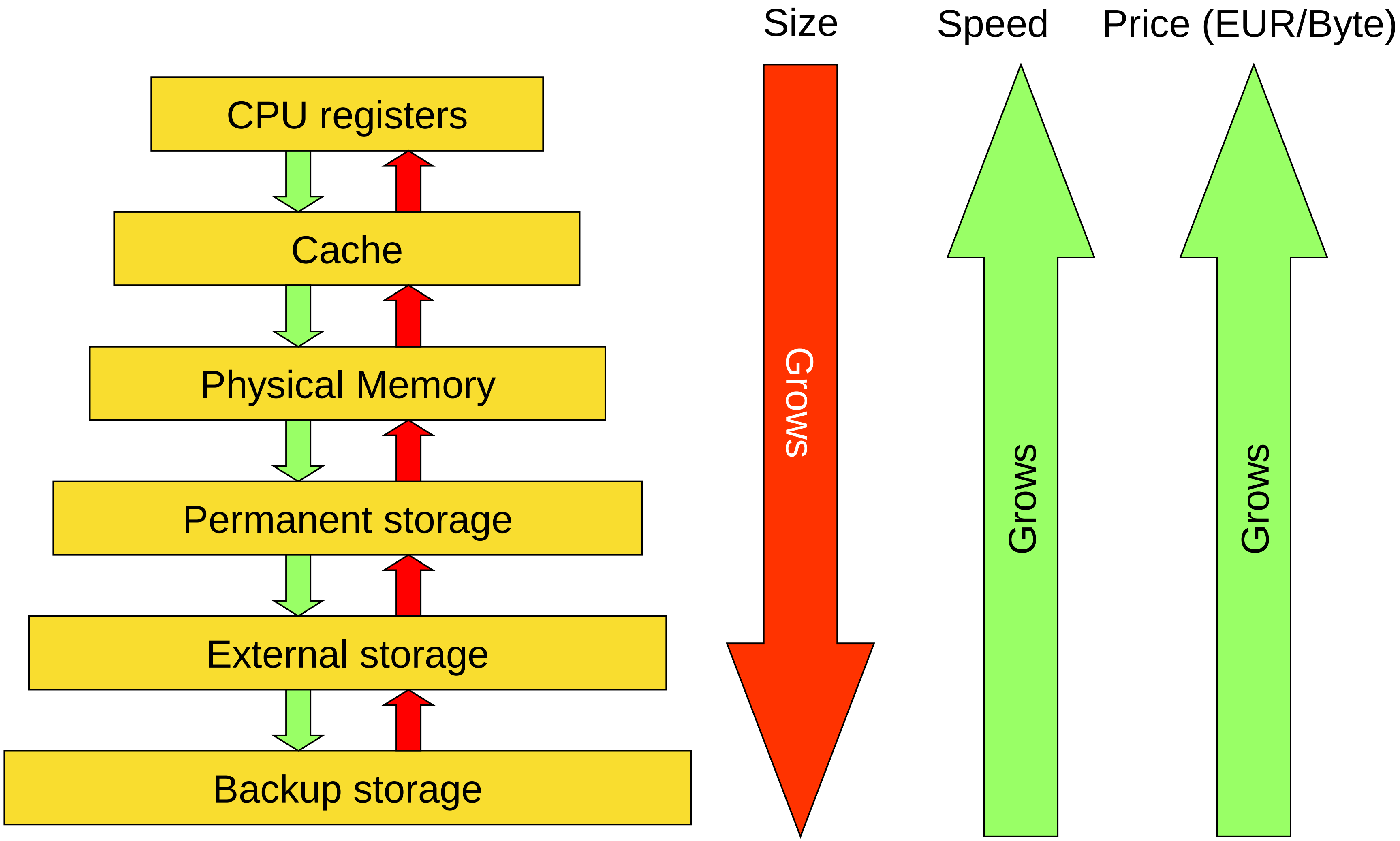
Handling of Memory

Tamás Kovácsházy, Phd

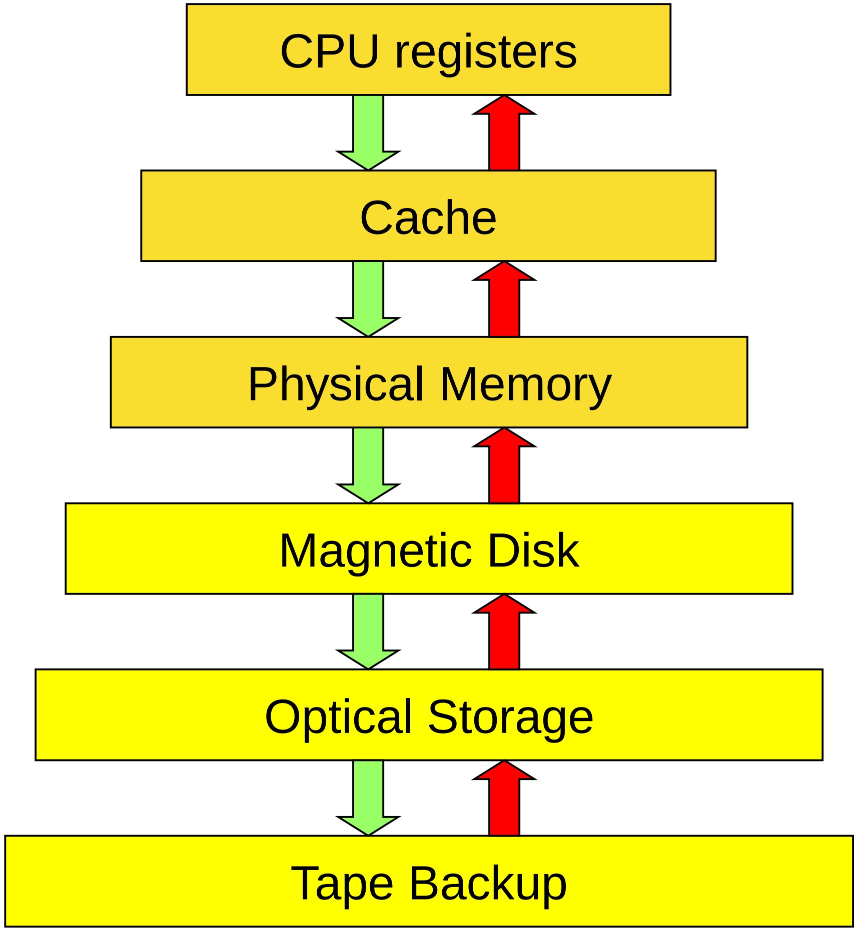
16th topic
Handling of Memory

Méréstechnika és Információs Rendszerek Tanszék

Storage hierarchy today



Storage hierarchy some years ago


CPU registers

Cache

Physical Memory

Magnetic Disk

Optical Storage

Tape Backup

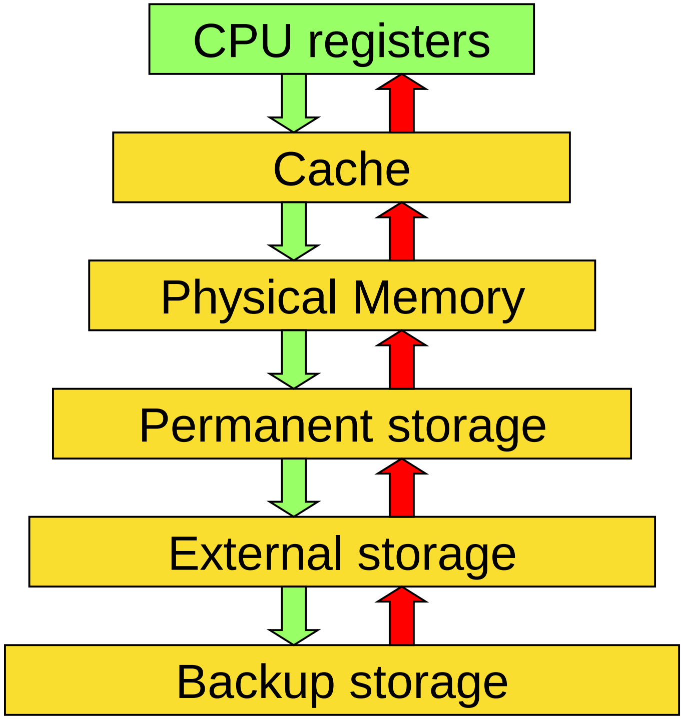
Flash memory (Pendrive, SSD) changed the picture!

Disappearing due to the Internet, portable HDD, pendrive!

Disappearing due to HDD backup!

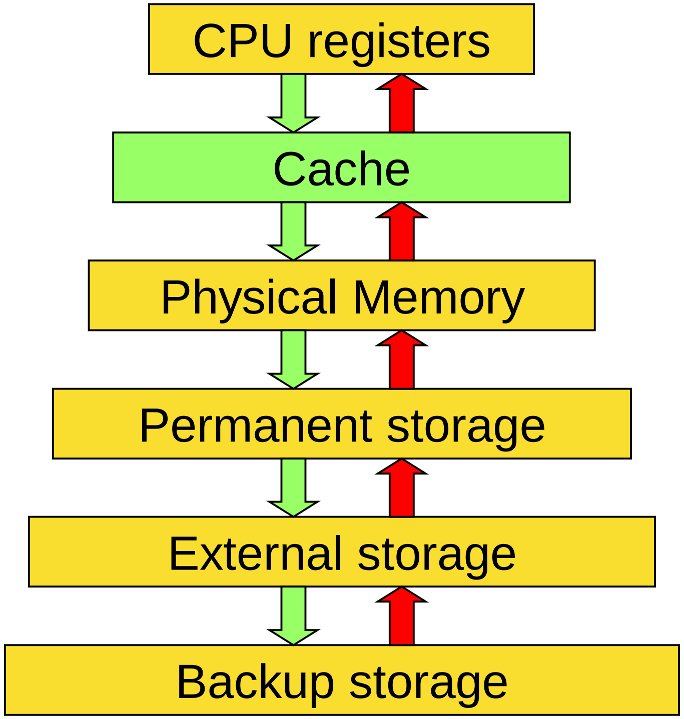
CPU registers

- Type: D flip-flop
  - 10-100 machine words
    - Small number of the on the x86 architecture while others may have more
  - Special purpose for some of them (PC, segment register., stb.).
- Speed:
  - During a single instruction may be accessed multiple times
    - E.g. ADD instruction (read one of the operands then write the result)
- Price: Hard to quantify
- Volatile memory (forgets the content if power supply is lost)



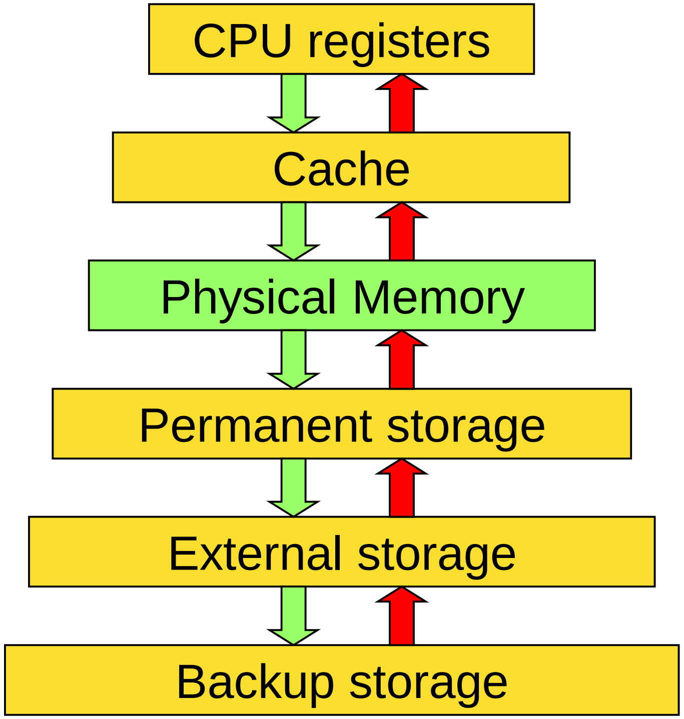
Cache

- Type: Static RAM (SRAM)
  - Multiple levels
  - 1st level: 64-128 Kbyte (I+D).
  - 2nd level: 1-8 Mbyte.
  - 3rd level: 4-32 Mbyte (if there is any).
- Speed:
  - Bandwidth: n*10 Gbyte/s.
  - Delay:
    - 1st level cache: one or some clock cycles
    - 2nd and 3rd level: 10 – n*10 clock cycles
- Price
  - High, SRAM need lot of space on the chip
- Volatile memory (forgets the content if power supply is lost)



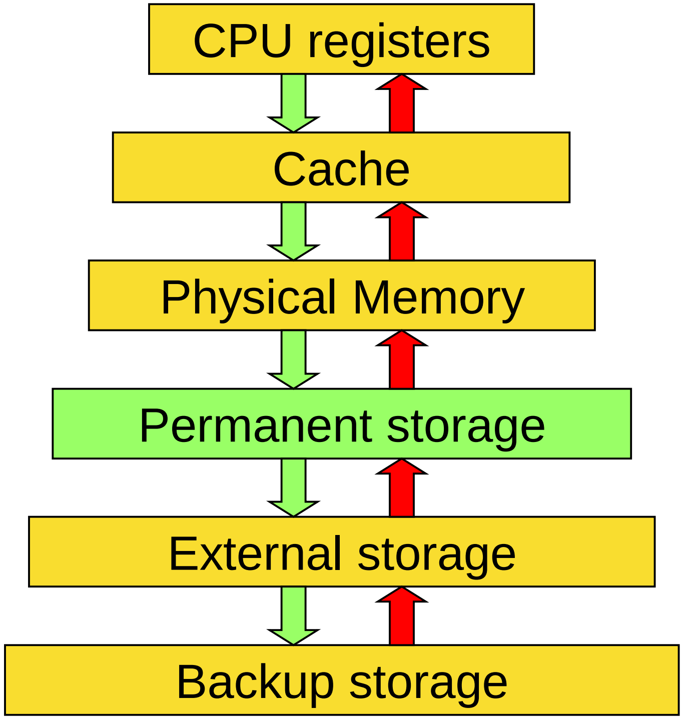
Physical (operative) memory

- Type: Dynamic RAM (DRAM)
- Size: n*10 Mbyte – n*100 Gbyte
- Speed:
  - Memory wall
  - Sequential and random access
    - Random access is slower
  - Max.: n*1 Gbyte/s – 10 Gbyte/s.
  - Delay: n*10 ns (worst case)
- Price:
  - Depends on market and availability, large fluctuations
- Must be refreshed periodically (today done by an on-chip controller)
- Volatile memory (forgets the content if power supply is lost)



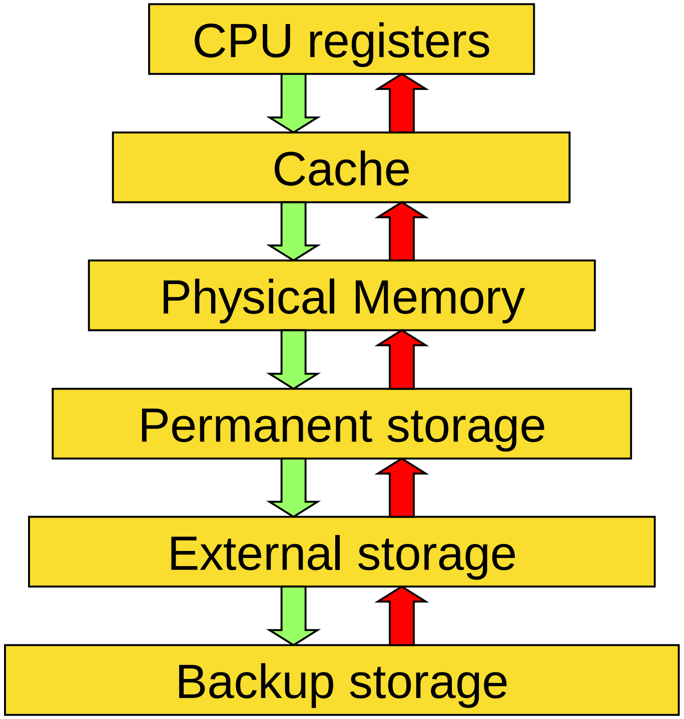
Permanent storage (HDD, Flash) 1.

- Types:
  - HDD (magnetic disk)
  - Flash memory (pendrive, SSD, etc.).
  - Size:n*1 Mbyte (Flash) – n*100 Tbyte.
- File based access
  - Content is seen as files from the operating system
  - Block based access on low HW level
- Speed:
  - Sequential and random access
    - Random access is slower for HDD
  - Read and write is different
  - Max. speed: n*10 Mbyte/s (low price pendrive) –
    n*1 Gbyte/s (RAID array).
  - Delay: n*10 ns (Flash) – kb. 10 ms HDD



Permanent storage (HDD, Flash) 2.

- Price: Strong size, technology, speed dependence
  - Cheap HDD is in the 70-80 EUR range even now
  - Storage array may cost millions of EUR
- Non-volatile, but very sensitive, can fail:
  - The data cannot be read back, i.e., it is lost
  - HDD: MTBF
  - Flash: wear
    - Limited number of writes can be endured by a flash block



Topics addressed

- Handling of physical memory
  - The notion of physical, logical and virtual memory
  - Mapping in between them and the management of them
- Handling of the permanent storage
  - Through virtual memory it is interconnected with the physical memory
  - How the OS handles the permanent storage, how it is organized, partitioned, how file systems and files are handled, etc.
  - RAID, NAS, SAN, etc.
- External storage and backup storage are not addressed in the class
  - Today, they are more and more handled just as permanent storage

Memory

- The CPU uses it for:
  - For loading instructions from it
  - Reading and writing data
- Sequence of memory operations
  - Reads and writes in a given sequence
  - As it is defined by the running program
- Process support must be provided (separation in memory).
- Let’s neglect the Cache!
  - It only influences the speed of operation assuming that the system is cache coherent

Logical and physical address

- Logical address:
  - The CPU generates it while it executes code in the process
  - Logical address space: All logical addresses of a process
- Physical address:
  - The address of a memory element as it appears on the memory bus driven by the memory controller
  - Physical address space: All the physical addresses of a process or the whole OS
- If they are different:
  - Sometimes we refer to it as virtual address
  - In this case the mapping between them is implemented by the MMU

Address binding 1.

- When the mapping between logical and physical addresses happen?
- Compile/link time binding
  - It is known in advance to where the code will be loaded
  - Absolute addressing
  - The program has physical addresses in it
  - It is typically used in embedded systems firmwares, BIOS/EFI and OS kernel may use it, DOS *.com programs use it also
- Load time binding
  - Relocatable code
  - The mapping is done while loading
  - The process uses physical addresses

Execution time address binding 2.

- Mapping can change while running
  - The processes may be moved to another location in the physical address space while running
  - Special HW must do the binding (MMU).
    - Execution speed must not be reduced due to it
  - The process is not allowed to see the actual physical addresses used by it directly
    - It always uses logical/virtual addresses
- Using logical addresses bound to physical addresses is a fundamental concept in modern operating systems

Dynamic loading

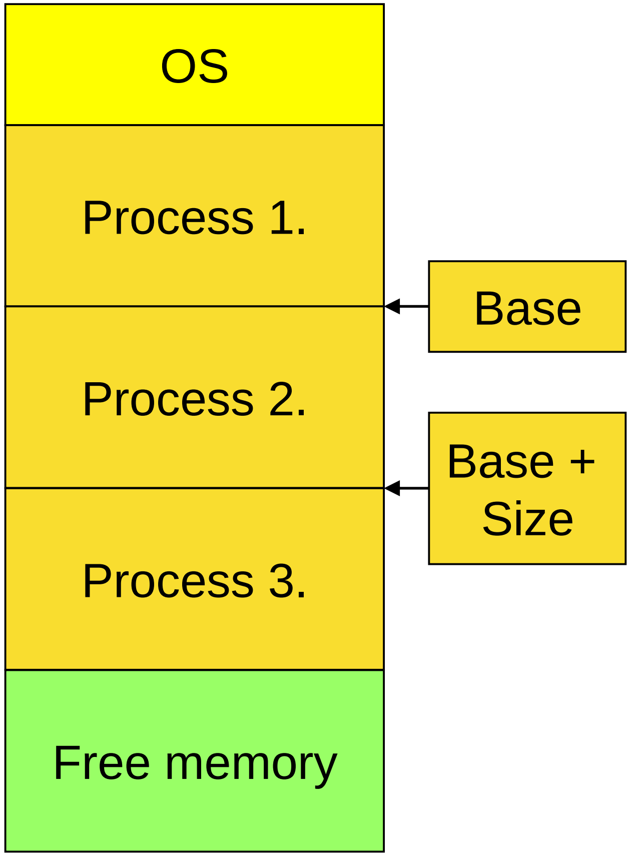
- Certain functions are not loaded into memory if the process does not use them
  - The program can be started faster (it is not necessary to copy everything in memory)
  - Less memory is used
  - Dynamic loading is implemented by the program itself
    - The operating system does not know about that, nor it supports it

Dynamic linking

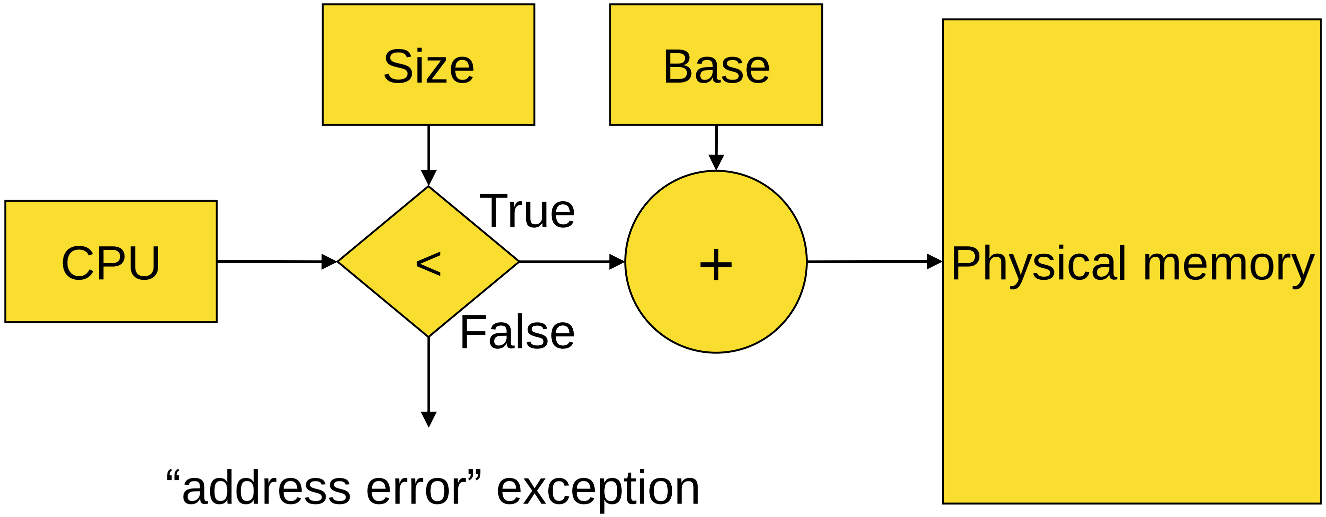
- Dynamic loading with operating system support
  - Dynamically linked/loaded library (Windows *.dll).
  - Shared Object (UNIX/Linux *.so).
  - Program libraries loaded using dynamic linking can be used by multiple processes (code sharing)
- Implementation:
  - The program itself contains only a stub of the program library
  - The stub finds and loads the whole program library if necessary using the services of the OS
  - The same program library may be present in the system with multiple versions (same name, multiple locations)
    - Which one is loaded?
    - UNIX: LD_LIBRARY_PATH environment variable defines it
    - Windows: It is version dependent how the DLLs are found

First solution (variable sized partitions)

- The process are located in a continuously addressed, static, predefined size physical address region
  - It starts from the Base address
  - It has a given size
  - The scheme is also called Base-relative addressing
- The process must fit into it during its lifetime
- Not an efficient solution, but it is a good first iteration



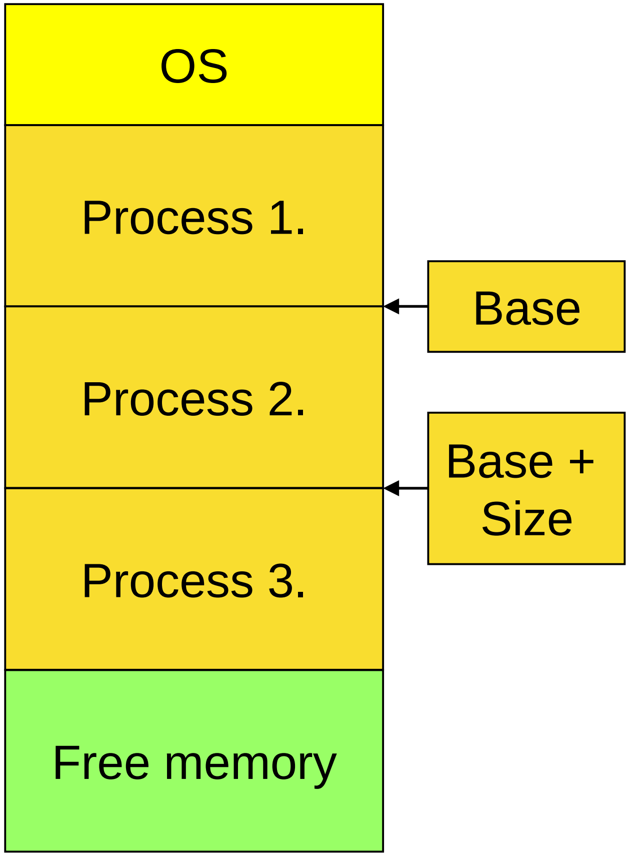
Implementation



- Required for implementation:
  - Two protected (available only from the OS kernel) register to store the Base address and the Size for the partition per process
  - One comparator and one adder
- If the process generate address out of the available memory it is signaled by an exception to the CPU
  - This exception is handled by the CPU in kernel mode

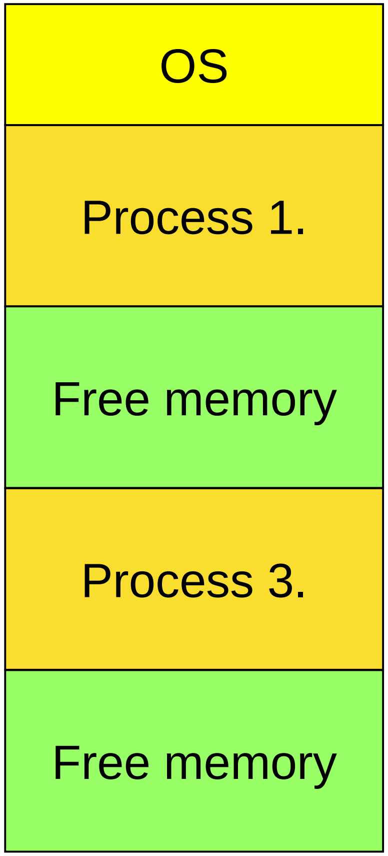
Problem 1.

- External fragmentation:
  - Process 2. is terminated



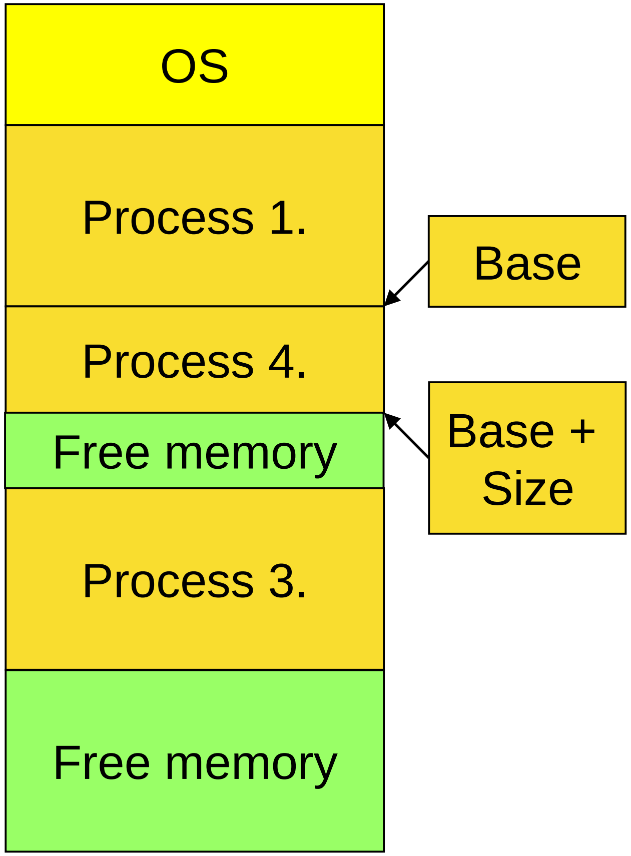
Problem 1.

- External fragmentation:
  - Process 2. is terminated
  - Its address space is freed



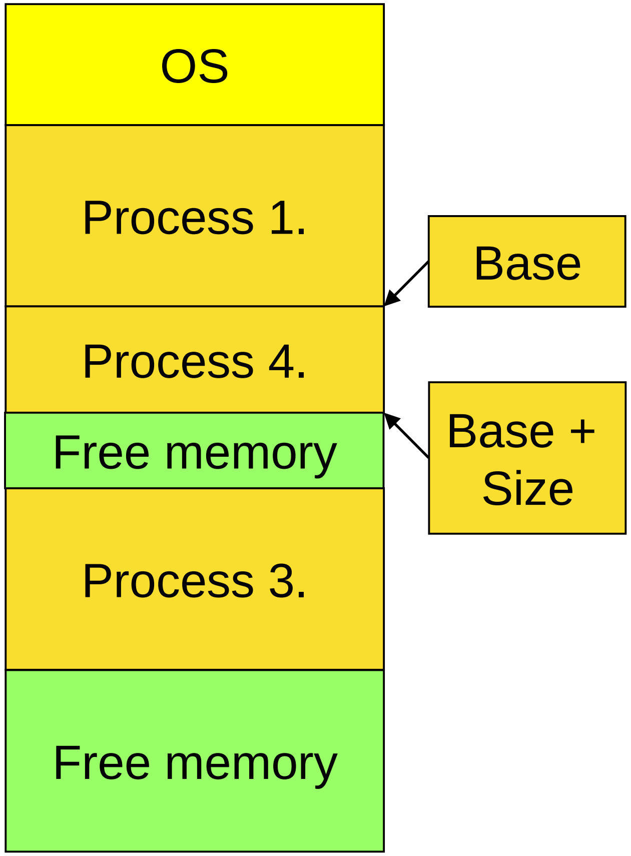
Problem 1.

- External fragmentation:
  - Process 2. is terminated
  - Its address space is freed
  - A smaller sized Process 4. is loaded into its former address space



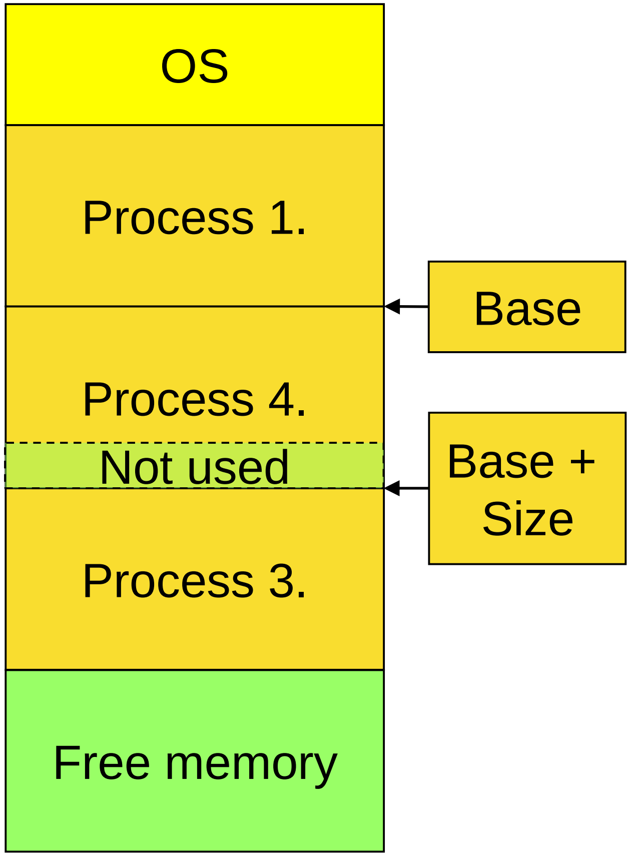
Problems 1.

- External fragmentation:
  - Process 2. is termineted
  - Its address space is freed
  - A smaller sized Process 4. is loaded into its former address space
  - The memory between Process 4. and Process 3. cannot be used due to its minimal size
    - No processes can be loaded into it
    - This memory is practically lost for the OS



Problems 2.

- Internal fragmentation:
  - Same as external fragmentation, but…
  - The small memory between Process 4. and Process 3. is not handled as free
    - It is given to Process 4.
    - It does not worth the resources used for accounting
  - The address space reserved by the process is not necessarily used by it



Problems 3.

- The memory requirement of running programs (processes) changes dynamically, it is very hard to specify an upper limit
- Running programs have the following properties:
  - A small part of the program code uses a small part of the data for very long periods of time typically (locality).
  - Certain parts of the program are never executed
    - E.g., research has shown that 90% of users never use 90% of functionalities of MS Office
    - Dynamic linking allows us only to load the really necessary parts of the code, and only in one instance for all programs using that code

Allocation strategies for new requests

- First fit:
  - It starts at the beginning of the storage and allocates the first applicable free area
- Next fit:
  - It starts to find applicable are at the free space left from the last allocation, otherwise it operates as the first fit
- Best fit:
  - From all the free areas the smallest applicable one is selected
  - The smallest external fragmentation is the aim
- Worst fit:
  - From all the free areas the largest applicable one is selected
  - The assumption is that we may be able to use the large remaining space later

Evaluation of allocation strategies

- They influence external fragmentation
- The First/Next/Best fit algorithms can allocate the 33% of the whole memory typically based n statistical analyses, i.e., half of the used memory cannot be allocated
- The worst fit algorithm is ever worse! 50% of the whole memory cannot be allocated
- First and the Next fit algorithms require less resources, the others require searching through the whole list
    - Assuming a simple list of free entries, not a more sophisticated data structure
- Example: Similar tasks could be found in the mid-term

Example

Introduce/explain the operation of the following allocation strategies for variable size partitions:

a, first fit

b, next fit

c, best fit

d, worst fit

by evaluating them for the following case:

Free areas: 23K, 64K, 10K, 80K, 12K, 50K és 40K

Request: 65K, 21K, 48K, 13K, 62K

Show how memory is allocated and specify if it is possible or not to allocate all the request!

Solution for the first fit algorithm

- Requests: 65K, 21K, 48K, 13K, 62K
- Free areas in the iterations:

0. 23K, 64K, 10K, 80K, 12K, 50K és 40K (65K req.)
1. 23K, 64K, 10K, 15K, 12K, 50K és 40K (21K req.)
2. 2K, 64K, 10K, 15K, 12K, 50K és 40K (48K req.)
3. 2K, 16K, 10K, 15K, 12K, 50K és 40K (13K req.)
4. 2K, 3K, 10K, 15K, 12K, 50K és 40K (62K req.)

It is not possible to allocate memory for the last request…

Solution for the worst fit algorithm

- Requests: 65K, 21K, 48K, 13K, 62K
- Free areas in the iterations:

0. 23K, 64K, 10K, 80K, 12K, 50K és 40K (65K req.)
1. 23K, 64K, 10K, 15K, 12K, 50K és 40K (21K req.)
2. 23K, 64K, 10K, 15K, 12K, 50K és 40K (48K req.)
3. 23K, 64K, 10K, 15K, 12K, 2K és 40K (13K req.)
4. 23K, 64K, 10K, 2K, 12K, 2K és 40K (62K req.)
5. 23K, 2K, 10K, 2K, 12K, 2K és 40K

It is possible to allocate the memory for all the requests…

Garbage collection

- Compaction, Garbage collection
- Solves the problem of external fragementation
  - Rearranges the allocation of memory for processes for continuous allocation
  - Creates a large, continuous free memory area
- Running it requires lot of resources
  - All base address registers must be reset to a new value
  - Physical memory must be copied from the old location to the new one

Better solution to the problem

- Physical memory management must be implemented more efficiently than that is possible with variable sized partitions
  - It was exceptionally critical with early computers (1960-1970) having physical memory in the range of n*10 kByte
  - But even with the current memory sizes (n*1Gbyte) the problem is here to stay (we want to do more)
- Swapping
- Segmentation (not directly a solution, but programmers like it to have)
- Paging

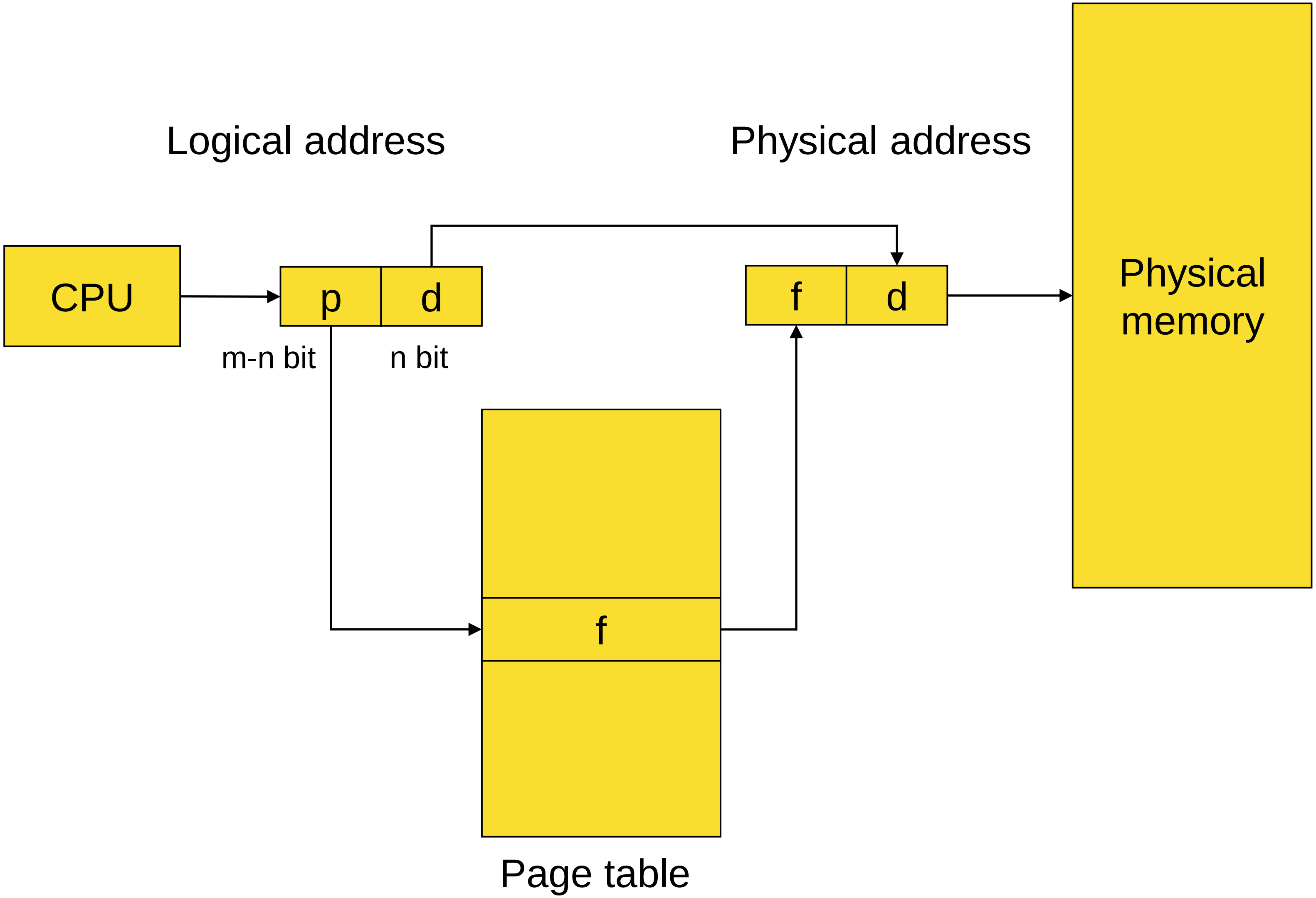
Swapping

- The whole process can be written to permanent storage (it is a memory content only):
  - It is possible if the process is not in running state and if there is no ongoing I/O operation on its memory are (DMA)
    - If I/O is done in OS buffers, the process itself can be swapped out
  - If run-time address binding is used
    - The process can be loaded to another memory area
  - Context switch associated with swapping operations is very time consuming (permanent storage is very slow compared to memory).
    - Whole processes must be written to or read from the permanent storage!

Paging

- Physical address space of the process can no none continuous (practically, it is non continuous)
- Physical memory is split into frames
- Logical memory is split into pages
- Page and frame sizes are identical
- All logical addresses are split into two parts
  1. Page number, p
  2. Page offset, d
- Page number is used to index the page table
  - A page table entry stores the base address of the physical memory frame (f) and some additional data (flag bits with various meaning)
- The frame table stores the list of free frames

Address translation with paging



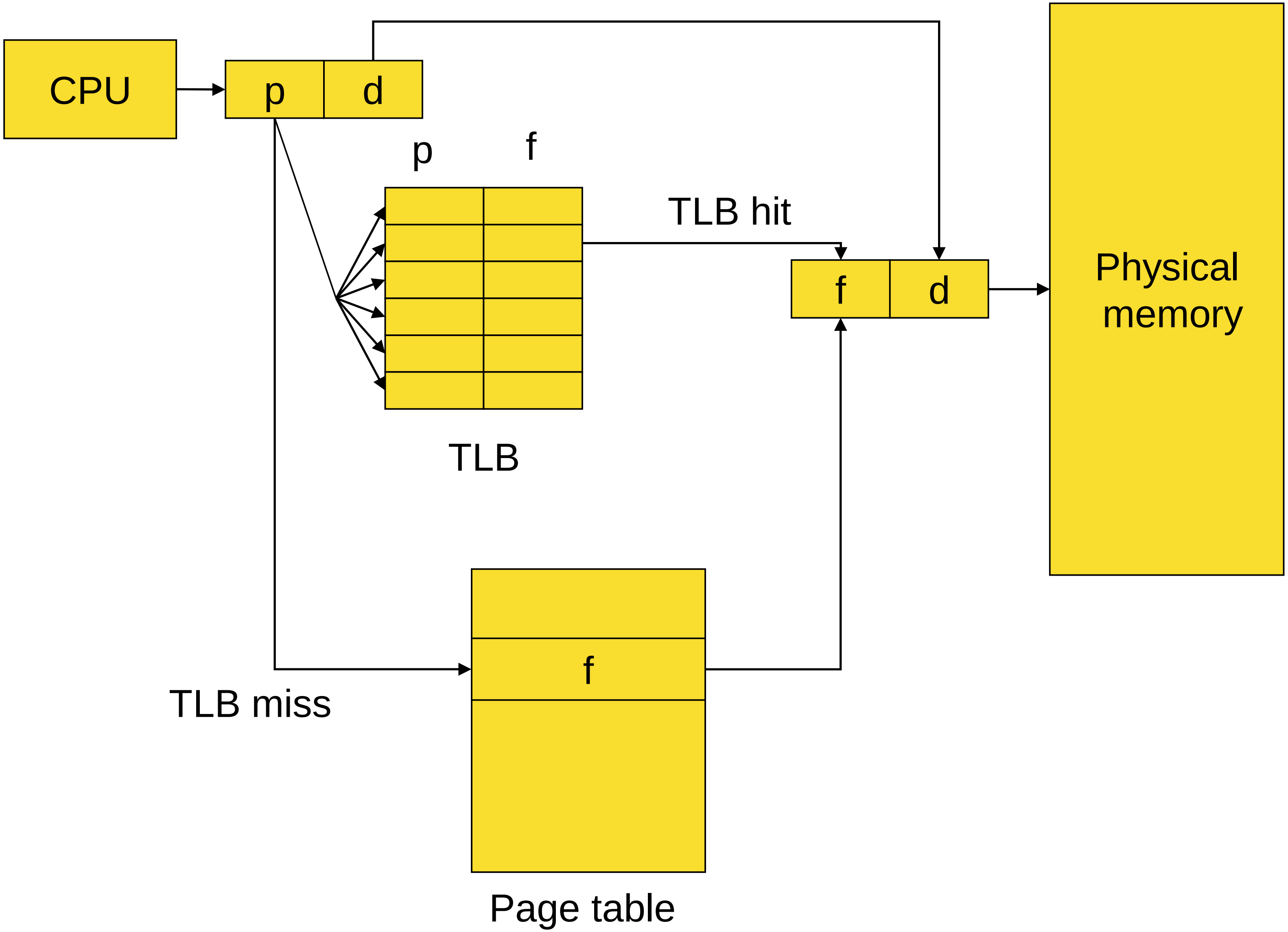
Evaluation of paging

- Mapping is done by hardware
- Process separation is implemented
  - Logical address space seen by processes are totally different from the actual physical address space used by the processes
  - The operating system handles the page and frame table in kernel mode, processes cannot access it in user mode
  - The process cannot have access to the memory area of other processes because it access memory through the page table…
- No external fragmentation
- Internal fragmentation is ½ page in average (small page size optimizes memory allocation efficiency)
- In modern computers large amount of memory is typical (large page size is better):
  - Simplified administration
  - Smaller page and frame table may be used

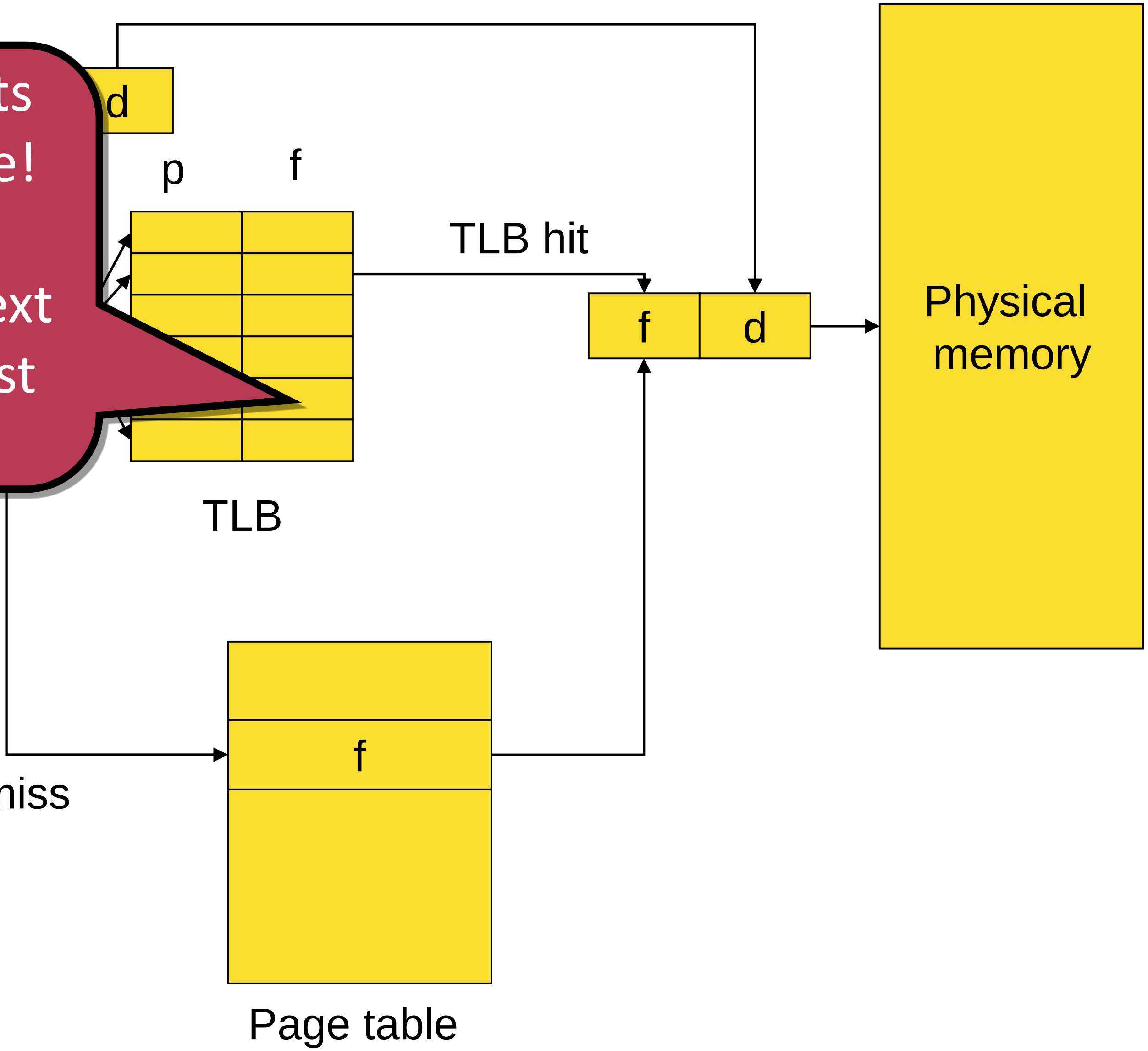
Searching in the page table

- The page and frame table can be quite large in modern operating systems:
  - In case of 4Kbyte page size and 32 bit addresses there are 2^{20} entry in the page table, that can be stored in a 4MByte memory (just the page table)
  - Several page sizes may be supported in a system
- Solutions (from the subject Computer architectures)
  - Hierarchical paging
  - Hashed page table
  - Inverted page table
- Search in the page table is slow:
  - Typically 2 times more than direct memory access (page table lookup then memory access)
  - Translation look-aside buffer (TLB) is used to speed up the process (a kind of associative memory)

Translation look-aside buffer (TLB)



Translation look-aside buffer (TLB)



TLB miss results in a TLB update!

In case of context switch TLB must be cleared…

TLB effectiveness

- The real question is the hit ratio?
  - It depends on the size of the TLB, TLB operational details, and on the executed code
    - How many pages are used and in which order they are used
  - Typically the hit ratio is in the range of 80-90%
- Effect of TLB on computer performance: AMD Phenom (2007)
  - There is a minor error in TLB, the BIOS allows to switch it of, differences:
    - 20% performance difference in the whole benchmark, 14% aapplication benchmarks
    - Memory benchmarks may show 50% difference!
    - http://techreport.com/articles.x/13741/4

Additional information in the page table

- Additional information:
  - Valid/invalid bit
    - Is the page in physical memory (page is mapped to a frame)?
    - If not, it must be brought in.
  - Read/read-write bit, execute bit, etc.
    - Specifies the allowed memory operation on the page
    - E.g. No Execute : No code execution from a page storing data (buffer overflow)
  - Referenced/Used bit
    - If a memory address in the page is used this bit is set by the MMU
- The HW (MMU) sets/checks the bits, and if the memory is not used according the values of bits the MMU generates exceptions, that are handled by the OS
- The OS works based on this bits, and sometimes also sets them

Sharing pages

- Processes are separated in memory form each other/
- However, for performance reasons this separation is limited
- In an OS controlled way memory may be shared:
  - The OS controls all aspects of sharing
- Applications
  - Shared code pages (read only, e.g. DLL/SO).
  - Copy on Write (COW):
    - In case of processes in parent-child relation
    - The two processes first use a fully shared physical memory
    - If any of the processes tries to write the page, during the first write, a process specific copy is created for the child, after that they can use their own specific (different) page
  - Writable shared pages
    - UNIX System V Shared memory.
- HW (MMU) must be present with the necessary capabilities to support these performance enhancements

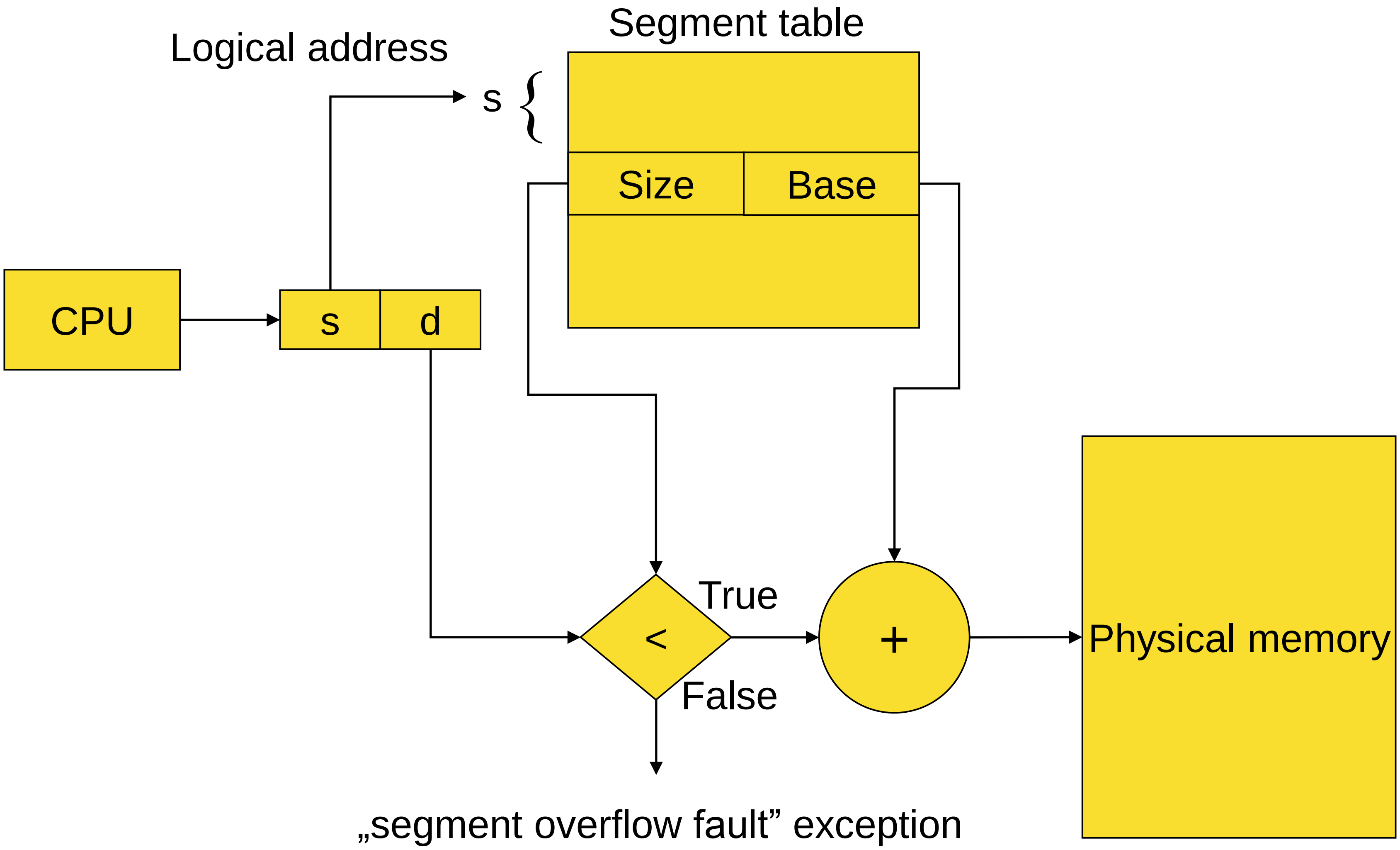
Segmentation 1.

- Programmers like to see memory:
  - Data, code, program, stack, heap, etc. memory areas are clearly identified
  - The program is located in a non-continuous memory in the view of the programmer
  - Violations of these limits are handled by the OS
- Segmentation implements this expectation
  - The logical address space is split into segments
  - Programmers specify a segment (segment name/ID) and a segment offset inside that to identify memory locations
    - In case of paging the programmer specify an address and that is split by hardware!
    - Here the programmer specifies two parts, and these two parts are fused together into a logical address!

Segmentation 2.

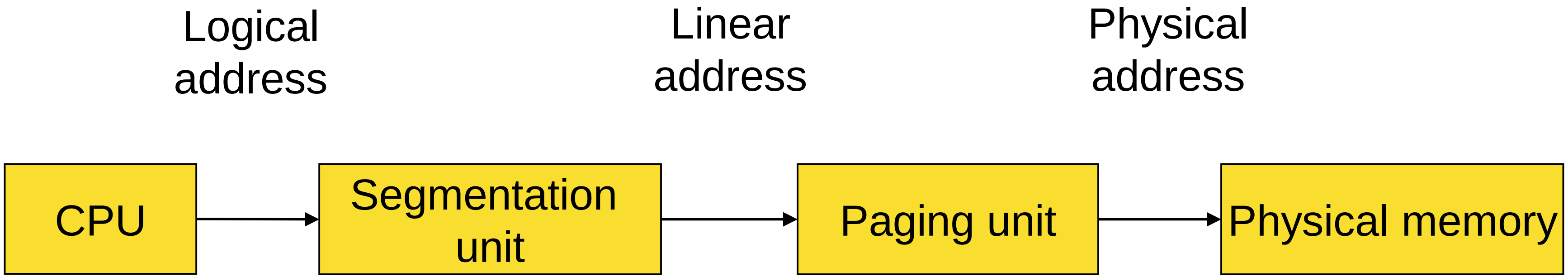
- Segment size must be stored!
  - If the address specified is outside the segment a “segment overflow fault” exception is generated by the MMU
- Segments can be stored continuously, not internal fragmentation happens (or at least it can be achieved)
- Segments are set up by the compiler and linker, and the program specifies it for the loader, which sets up the memory space of the program according that specification

Implementation



Segmentation and paging

- Some HW supports both (such as the x86 architecture)
- In the book it is introduced in detail (we do not deal with it due to time restrictions)
- Segmentation is used sometimes on OSs supporting x86 hardware
  - E.g. Linux uses 6 segments: kernel code/data, user code/date, Task state, default local descriptor table
- On the other hand, paging is a fundamental requirement
  - x86 HW (32 bit) uses 4KByte pages (2 level page table) or 4Mbyte pages (1 level page table)
  - Linux has a 3 level page table, the 2nd level is not used on x86 hardware



Virtual memory

- Earlier we talked about virtual addresses (when logical address $\neq$ physical address
  - Virtual memory is a different thing (much more) while it is uses virtual addresses
- Sometimes it is assumed to be the same as swapping
  - It is uses features like swapping, but quite different than that
- Practically, virtual memory is a new memory management technology built on the foundation of earlier solutions

Observations

- Executing processes in physical memory:
  - Seems to be a required and reasonable solution
  - There are serious consequences of the solution

1. It is not required to load the whole process to memory to execute it, most cases it is sufficient to have the neighborhood of the PC (Program counter) and necessary data in memory (the locality)
2. Large parts of the code of processes are never or rarely executed (error handling, software/feature bloat)
3. It is not necessary to load the whole program to start it
4. Some code can be shared among processes, and sometimes it is even reasonable to share data (e.g. they use the same code and data).
5. Some code and data in processes are not used again or used again after long times (and therefore not needed in physical memory), while other processes lack sufficient amount of physical memory

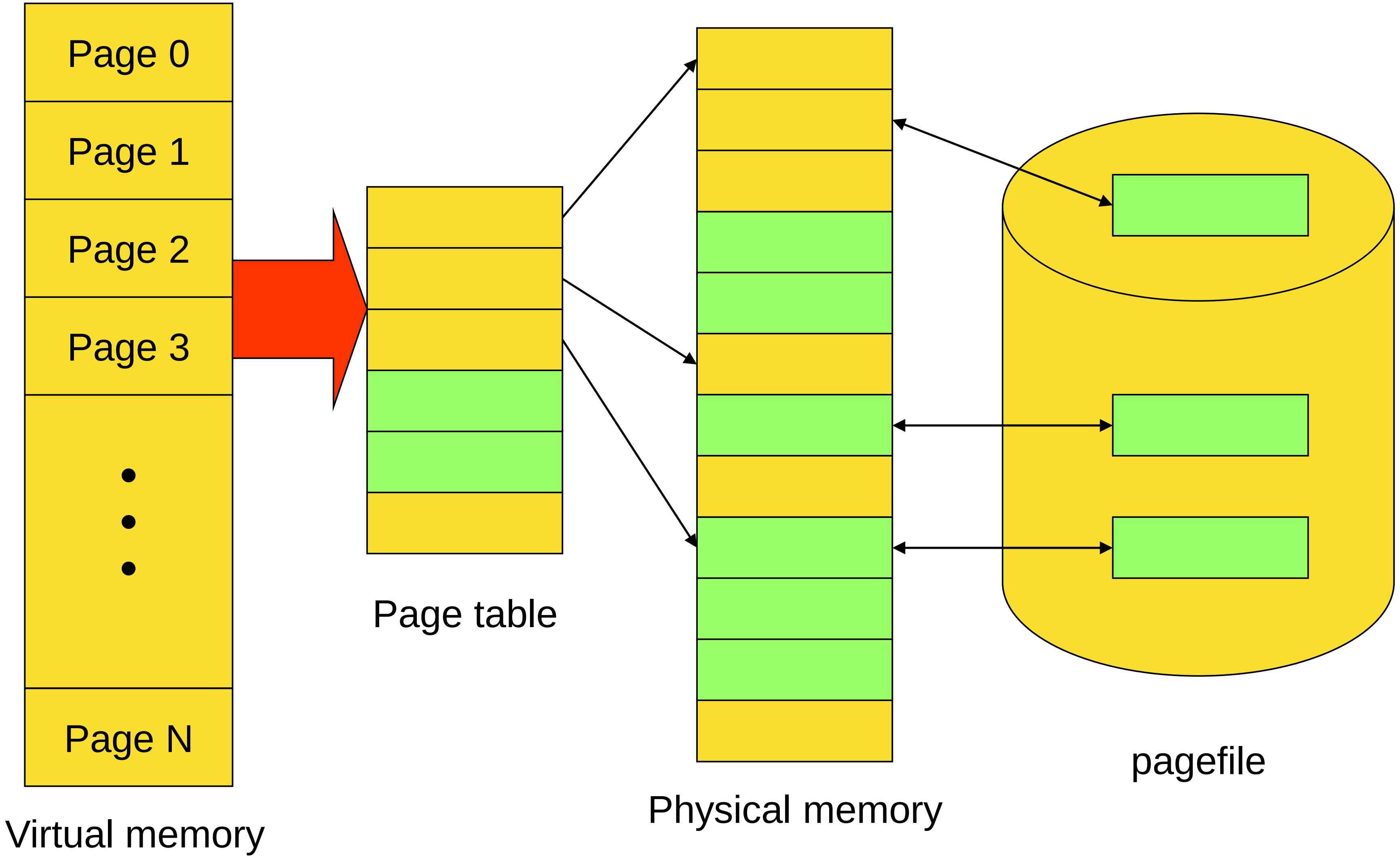
Requirements

- It would be nice to run programs with bigger memory requirements than the available physical memory
  - Virtual memory, the programmer does not need to deal with the available memory, it is always there (at least virtually)
  - There are architectural limits (32 bit architecture, 4Gbyte)
  - There are consequences: Complexity and speed
- The OS assigns only the necessary physical memory to processes, therefore, more processes can run in parallel (kept in physical memory)
- Programs load only the minimal part code and data into physical memory while starting, therefore, they can start up faster
- They can share code, data, and resources in an efficient way in memory

Virtual memory

- The foundation is paging
  - The continuous virtual address space is mapped by a table (memory map, page table)
- The mapping is not directly to physical memory
  - The virtual memory is partially mapped to physical memory
  - The virtual memory is partially mapped to a special area of the permanent storage (HDD, SSD)
    - Pagefile (Windows) or swap partition/file (UNIX/Linux)
    - The UNIX/Linux name is selected for archaic reasons
- The process sees a continuous, large virtual memory space, but that space is sparsely populated (large parts are not mapped at all to anything) with physical memory or permanent storage locations

Virtual memory in a figure

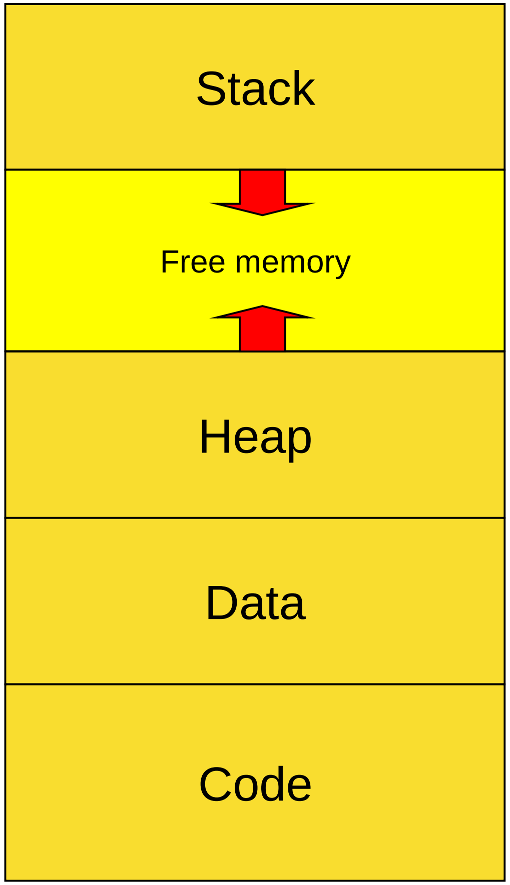


Other properties

- Processes may share physical memory areas with read and even write access rights
  - Access rights must be also stored in the page table
  - These physical memory areas are paged into the virtual address space of multiple processes (on different virtual addresses)
- Keeping record of modifications (modified/dirty bit):
  - All page has this bit maintained by the MMU in the page table
    - At load it is set to False, in case of a write to the page it is set to True
    - The system knows if a page has been changed (need to be written back to its copy stored on the permanent storage)
- Keeping record of use (referenced/used bit):
  - OS clear it periodically or in case of some events
  - MMU sets it in case of use (read or write on the page)

Consequences

- The process sees a continuous, large memory in which it can run
- In reality the program runs in a sparse address space
  - A large part of the virtual memory is not mapped at all
  - If it is required the system puts physical memory behind the referenced virtual memory on demand



Process in memory

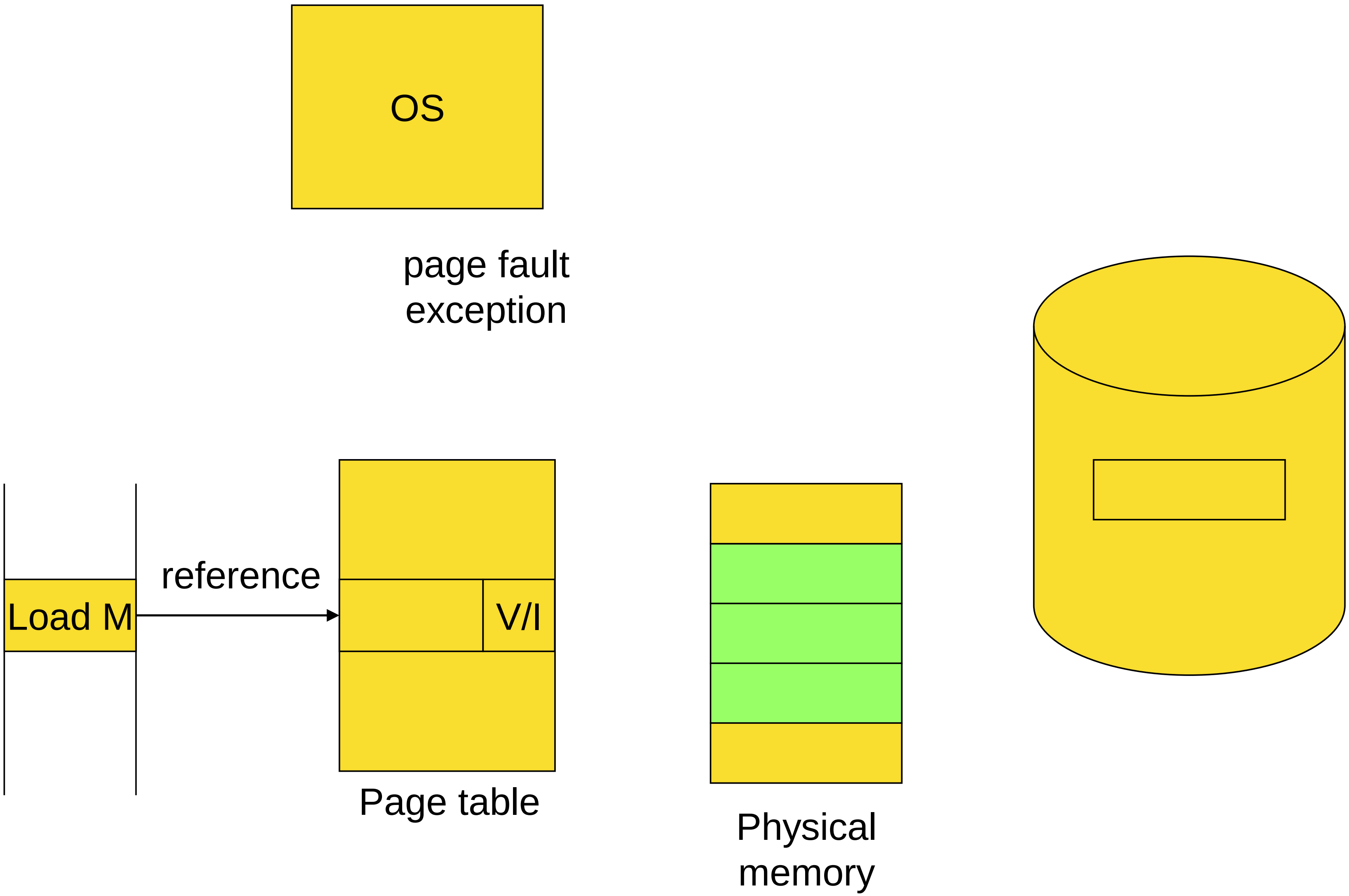
Operation

- If the referenced virtual memory page is in physical memory, the instruction can be executed
- What if not (valid/invalid bit)?
  - A page fault exception is generated by the MMU
    - The page is not in physical memory, i.e., physical memory must be put “behind” the virtual one
    - It is not an error, but the part of the regular operation!
  - The operating system handles the situation, possibilities:
    - It is written to the page file, must be brought in
    - Never loaded, must be brought in from its original location (e.g. a file in the filesystem)
    - Question: To where, especially interesting question if the physical memory is full?
  - The OS return control back to the process, the process sees nothing from it
  - While physical memory is assigned the process waits passively (in the middle of an instruction)

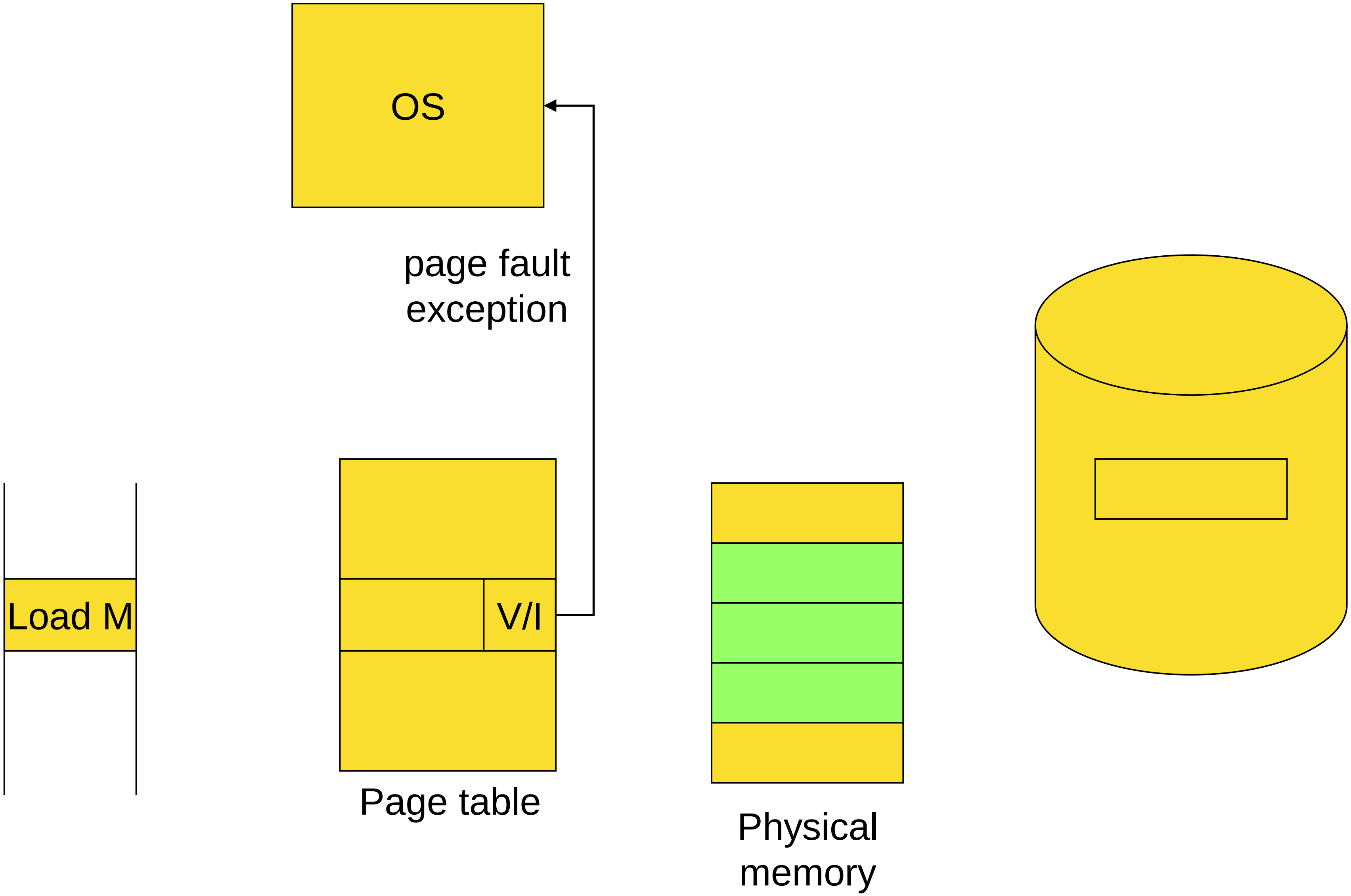
Fetch strategies

- Demand paging:
  - The algorithm starts only in case of a page fault, and loads only the page into physical memory required to continue operation
  - Only the used pages are in physical memory
  - Referencing a page not in physical memory results in long waiting times (it must be brought in)
- Anticipatory paging:
  - Looks into the future (estimation), the OS tries to find out which of the pages are to be used, and brings into physical memory also those pages
  - Requirement: free resources (CPU, HDD, physical memory)
  - If the preloaded pages are really used (successful estimation) the number of page faults decreases
  - If not, we use resources without any gain (even performance can degrade due to this)

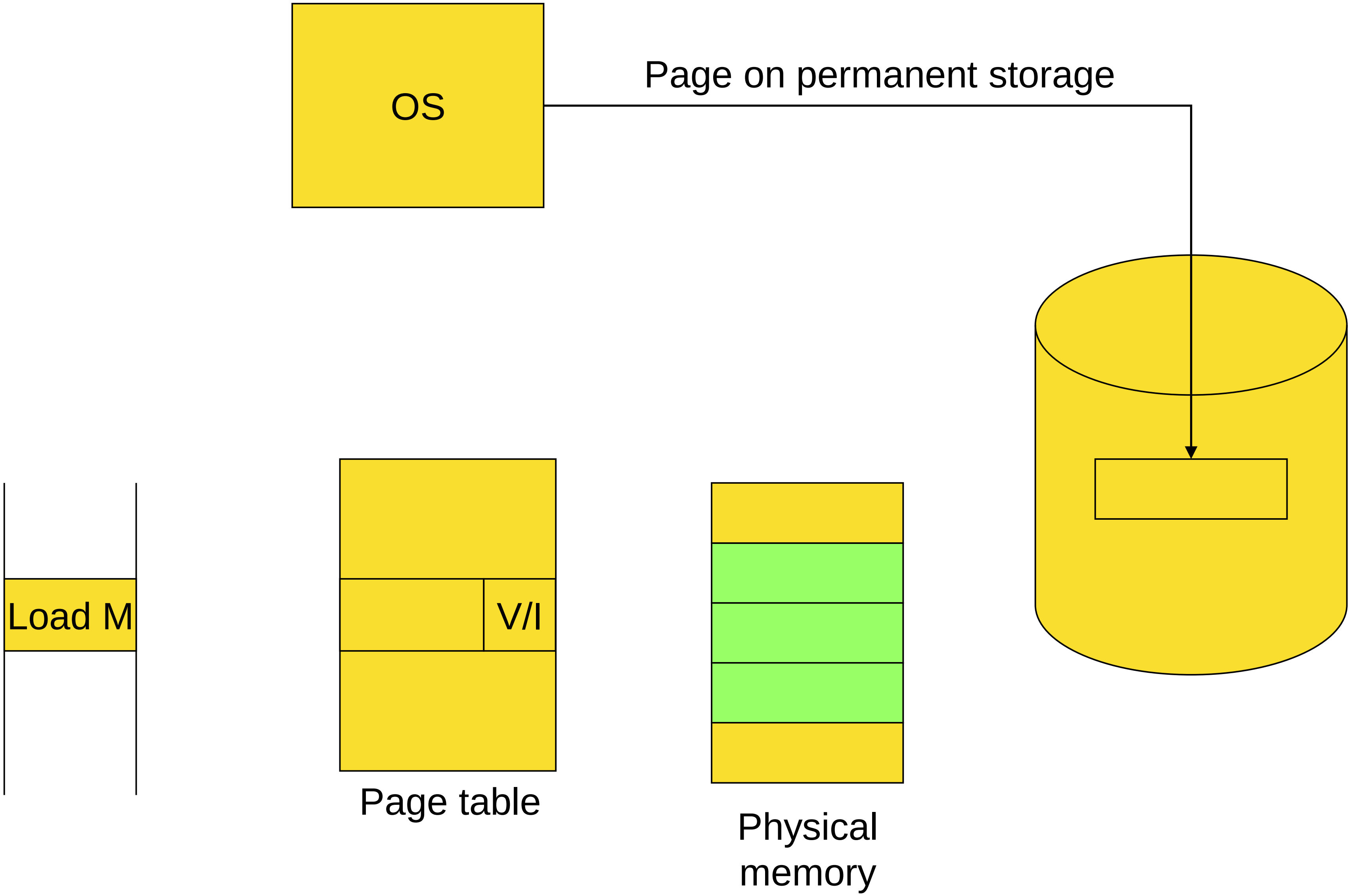
Operation in case of a page fault 1.



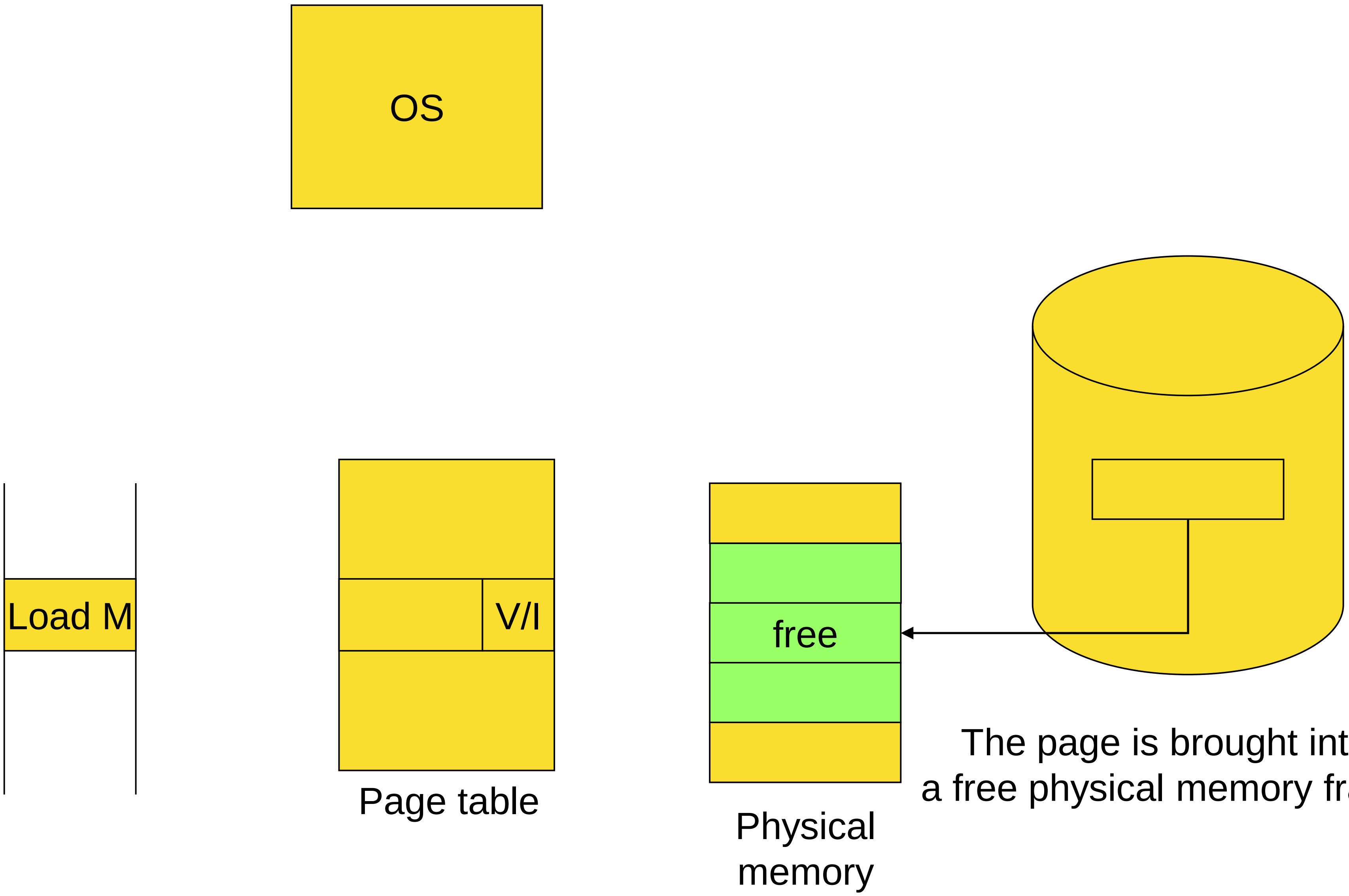
Operation in case of a page fault 2.



Operation in case of a page fault 3.

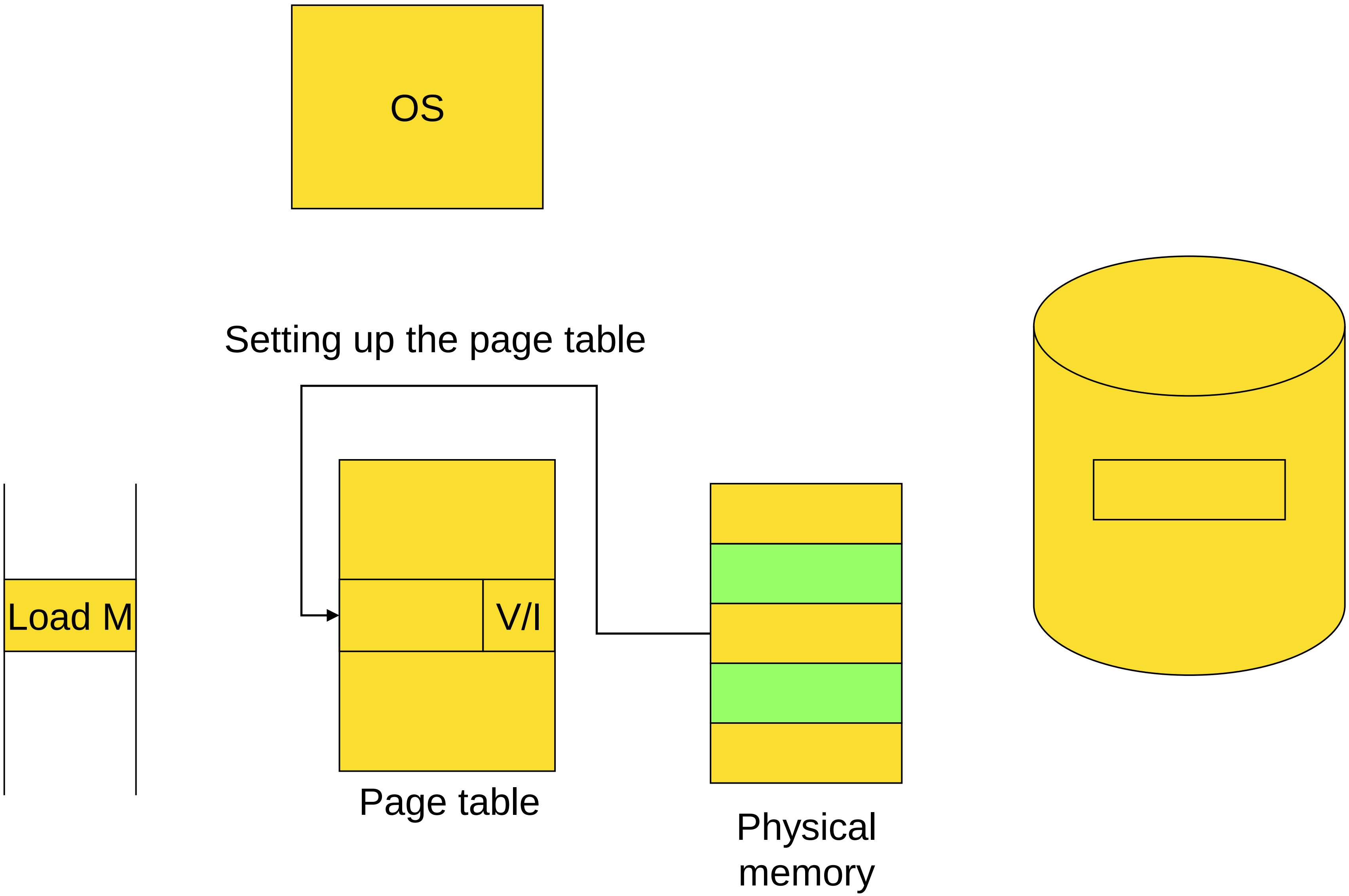


Operation in case of a page fault 4.

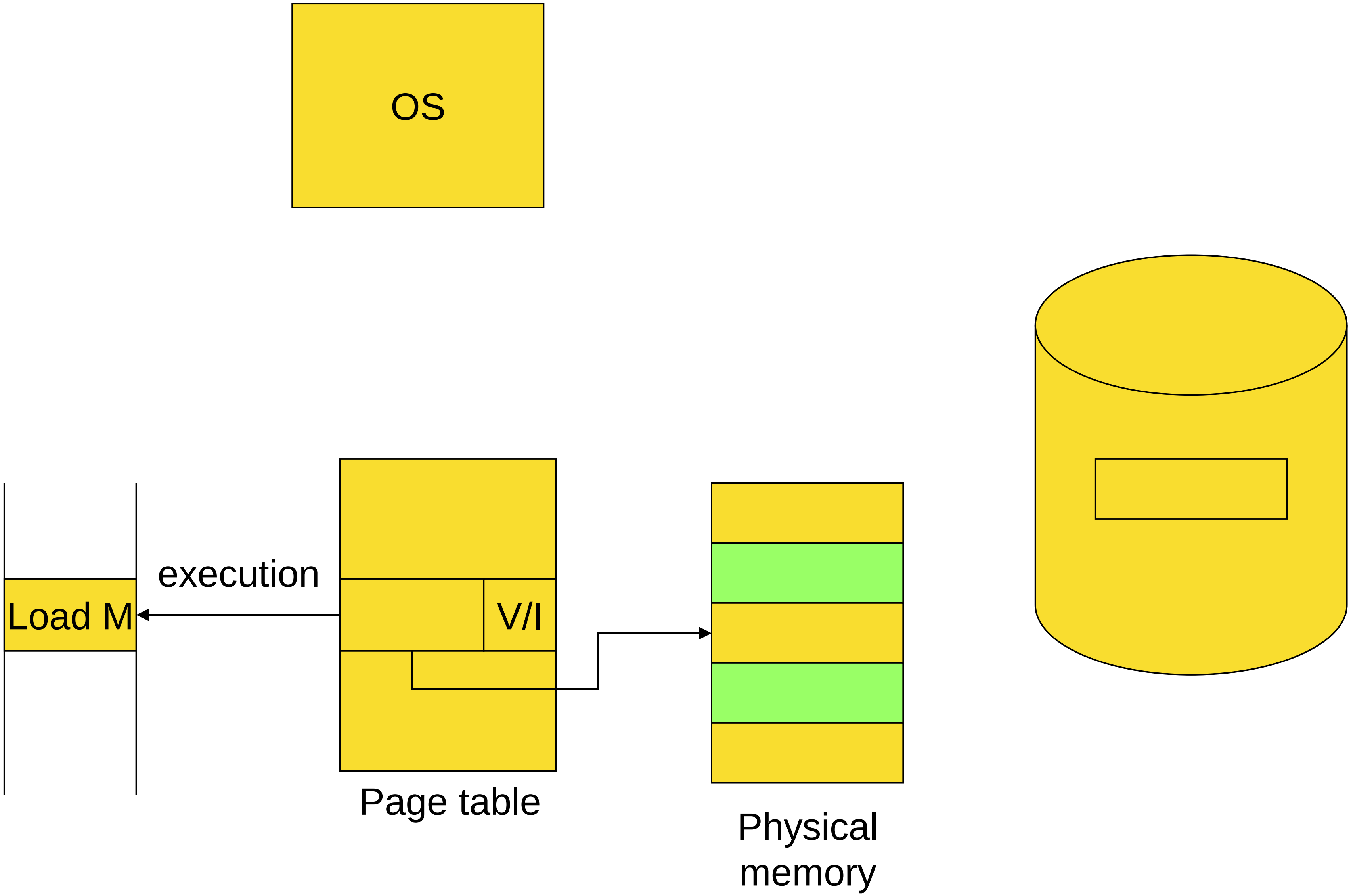


The page is brought into
a free physical memory frame

Operation in case of a page fault 5.



Operation in case of a page fault 6.



Page replacement strategies

- The physical memory is full and a page fault happens
  - We need to select a physical memory frame to be written to the permanent storage
    - Victim selection
  - To this place we will load the data to be loaded into physical memory
- Algorithms:
  - Optimal algorithm
  - Oldest page (FIFO)
  - Second chance (SC)
  - Least recently used (LRU)
  - Least frequently used (LFU)
  - Not recently used (NRU)

Optimal algorithm

- Looks into the future, and knows witch of the pages are going to be used
    - Cannot be realized (Who knows the future?), but a good reference point as a performance target
    - In the knowledge of the actually referenced pages one can compute what the optimal algorithm would have done

FIFO

- References to pages in physical memory are organized in a FIFO, and the oldest one (brought in first) is replaced
  - Very simple, makes decision based on the past, looks into the past
  - It may replace frequently used pages
    - It does not take into account the actual usage
    - It takes into account of the time the page is brought in
  - Bélády anomaly (László Bélády, a former BME student, who left the country in 1956, and worked for IBM):
    - If the number of pages assigned to a process is increased the number of page faults may also increase
    - Anomaly: It does not work as expected
    - The number of page faults should decrease, but not in this case

Second chance, SC

- It operates like the FIFO, replacing the page brought in the earliest time but only if that page has not been referenced (it uses the referenced/used bit).
  - The reference bit is set to TRUE by the MMU if the page is used (referenced in an instruction in any way)!
  - The SC algorithm sets the reference bit to FALSE if encounters with a page for which this bit is set to TRUE and gives the page a second chance to stay in physical memory:
    - This is why it is named to second chance
    - After that the page is put to the end of the FIFO as a newly loaded page
    - Otherwise it would en up in an infinite cycle
    - Then it tries to replace the next page in the FIFO
  - It shows a higher complexity than the FIFO, but generally it is a simple algorithm compared to the other alternatives
  - It looks back into the past, and makes decisions based on the time of presence in the physical memory, and usage also
    - It is somewhat capable of taking into the locality of the program

Least Recently Used, LRU

- A good solution:
  - Complex, but can provide performance close to the performance of the optimal algorithm
    - It takes locality into account
  - It also looks back
  - Various implementations exist:
    - Counter based: For all pages there exists a “last used counter”
    - Linked list: The page used last is put to the end of list
    - Two-dimensional array: NxN matrix, where N is the number of pages
  - Most cases the LRU algorithm is approximated only

Least Frequently Used, LFU

- The pages frequently used in the recent past are used again due to the locality of the program
  - The pages rarely used in the recent past are not likely to be used again
  - The value of the Referenced (R) bit is added to a page specific counter, and the value of R is erased
  - The page with the smaller counter value is replaced
  - The algorithm does not forget anything...
  - The algorithm will replace the newly loaded pages because the their counter has 0 or very low value
    - Pages newly loaded into physical memory must be frozen into the physical memory for a while

Not Recently Used, NRU

- Both the Referenced (R) and modified (M) bits are taken into account during the algorithm
- R can be erased, M must be kept
- Priority is assigned to pages based on the value of the bits R and M
  - Priority 0: R=0, M=0 (lowest priority)
  - Priority 1: R=0, M=1
  - Priority 2: R=1, M=0
  - Priority 3: R=1, M=1 (highest priority)
- Always selects a page from the lowest priority group if there is a page in that priority group

Aspects of page replacement

- Global page replacement: The whole physical memory can be a victim
- Local page replacement: Only the physical memory of the process can be replaced
- Locking page into physical memory (lock bit):
  - Why?
  - I/O operations reference the page
    - In I/O physical addresses are used!
  - The newly loaded pages in case of the LFU algorithm may be replaced immediately, due to the fact they have much lower use count than other pages

Example

- Let’s introduce ourselves to the algorithms by examples
- Similar tasks can be found in the exam or in the mid-term (if this part is presented before the mid-term)

Example

A system using demand paging there makes 3 or 4 physical memory frames available for a process. The process references the following pages while running:

0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4, 0, 1

- How many pages faults happen for the following algorithms, if the physical memory pages at the beginning are empty?
  - FIFO algorithm with 3 or 4 physical memory frames
  - Least Recently Used (LRU) algorithm with 3 or 4 physical memory frames
  - Second chance (SC) algorithm with 3 or 4 physical memory frames
- Explain the results!

FIFO with 3 physical memory frames

Pages referenced		0	1	2	3	0	1	4	0	1	2	3	4	0	1
Results		0	1	2	3	0	1	4	4	4	2	3	3	0	1
			0	1	2	3	0	1	1	1	4	2	2	3	0
				0	1	2	3	0	0	0	1	4	4	2	3
Page faults		y	y	y	y	y	y	y			y	y		y	y

FIFO with 4 physical memory frames

Pages referenced		0	1	2	3	0	1	4	0	1	2	3	4	0	1
Results		0	1	2	3	3	3	4	0	1	2	3	4	0	1
			0	1	2	2	2	3	4	0	1	2	3	4	0
				0	1	1	1	2	3	4	0	1	2	3	4
					0	0	0	1	2	3	4	0	1	2	3
Page faults		y	y	y	y			y	y	y	y	y	y	y	y

LRU with 3 physical memory frames

Pages referenced	0	1	2	3	0	1	4	0	1	2	3	4	0	1
Results	0 1	0 2	0 3	3 1	3 2	3 3	4 1	4 2	4 3	2 1	2 2	2 3	0 1	0 2
		1 1	1 2	1 3	0 1	0 2	0 3	0 1	0 2	0 3	3 1	3 2	3 3	1 1
			2 1	2 2	2 3	1 1	1 2	1 3	1 1	1 2	1 3	4 1	4 2	4 3
Page faults	y	y	y	y	y	y	y			y	y	y	y	y

LRU with 4 physical memory frames

Pages referenced	0	1	2	3	0	1	4	0	1	2	3	4	0	1
Results	0 1	0 2	0 3	0 4	0 1	0 2	0 3	0 1	0 2	0 3	0 4	4 1	4 2	4 3
		1 1	1 2	1 3	1 4	1 1	1 2	1 3	1 1	1 2	1 3	1 4	0 1	0 2
			2 1	2 2	2 3	2 4	4 1	4 2	4 3	4 4	3 1	3 2	3 3	3 4
				3 1	3 2	3 3	3 4	3 5	3 6	2 1	2 2	2 3	2 4	1 1
Page faults	y	y	y	y			y			y	y	y	y	y

SC with 3 physical memory frames

Pages referenced	0	1	2	3	0	1	4	0	1	2	3	4	0	1
Results	0 y	1 y	2 y	3 y	0 y	1 y	4 y	4 y	4 y	2 y	3 y	3 y	0 y	1 y
		0 y	1 y	2 n	3 y	0 y	1 n	1 n	1 y	4 n	2 y	2 y	3 n	0 y
			0 y	1 n	2 n	3 y	0 n	0 y	0 y	1 n	4 n	4 y	2 n	3 n
Page faults	y	y	y	y	y	y	y			y	y		y	y

Demand paging, it brings in the page and uses it!

SC with 4 physical memory frames

Pages referenced	0	1	2	3	0	1	4	0	1	2	3	4	0	1
Results	0 y	1 y	2 y	3 y	3 y	3 y	4 y	0 y	1 y	2 y	3 y	4 y	0 y	1 y
		0 y	1 y	2 y	2 y	2 y	3 n	4 y	0 y	1 y	2 n	3 y	4 y	0 y
			0 y	1 y	1 y	1 y	2 n	3 n	4 y	0 y	1 n	2 n	3 y	4 y
				0 y	0 y	0 y	1 n	2 n	3 n	4 y	0 n	1 n	2 n	3 y
Page faults	y	y	y	y			y	y	y	y	y	y	y	y

Evaluating the results

- FIFO and SC for 3 or 4 physical memory frames:
  - It is worse for 4 frames than for 3 frames
  - Bélády anomaly
- LRU for 3 frames are worse than the FIFO or SC for 3 frames:
  - The list of referenced pages has larger locality than the available frames
- For 4 frames the LRU is the best
- The algorithms can be evaluated based on a full statistical analyses
  - This numbers are made up to show the typical behavior

Performance of virtual memory 1.

- The performance of the physical memory
  - n*1 Gbyte/s throughput
  - n*10 ns delay
  - If it is cached, it is even faster
- The performance of the permanent storage
  - Typically the throughput is in the range of 100 Mbyte/s, but in case of random access and large number of parallel users it can be significantly slower
  - 10 ms worst case access time (head movement).
  - In case of Flash storage deleting and writing content is slow and problematic
    - It may be damaged fast due to the extensive number of write cycles
- The permanent storage is magnitudes slower

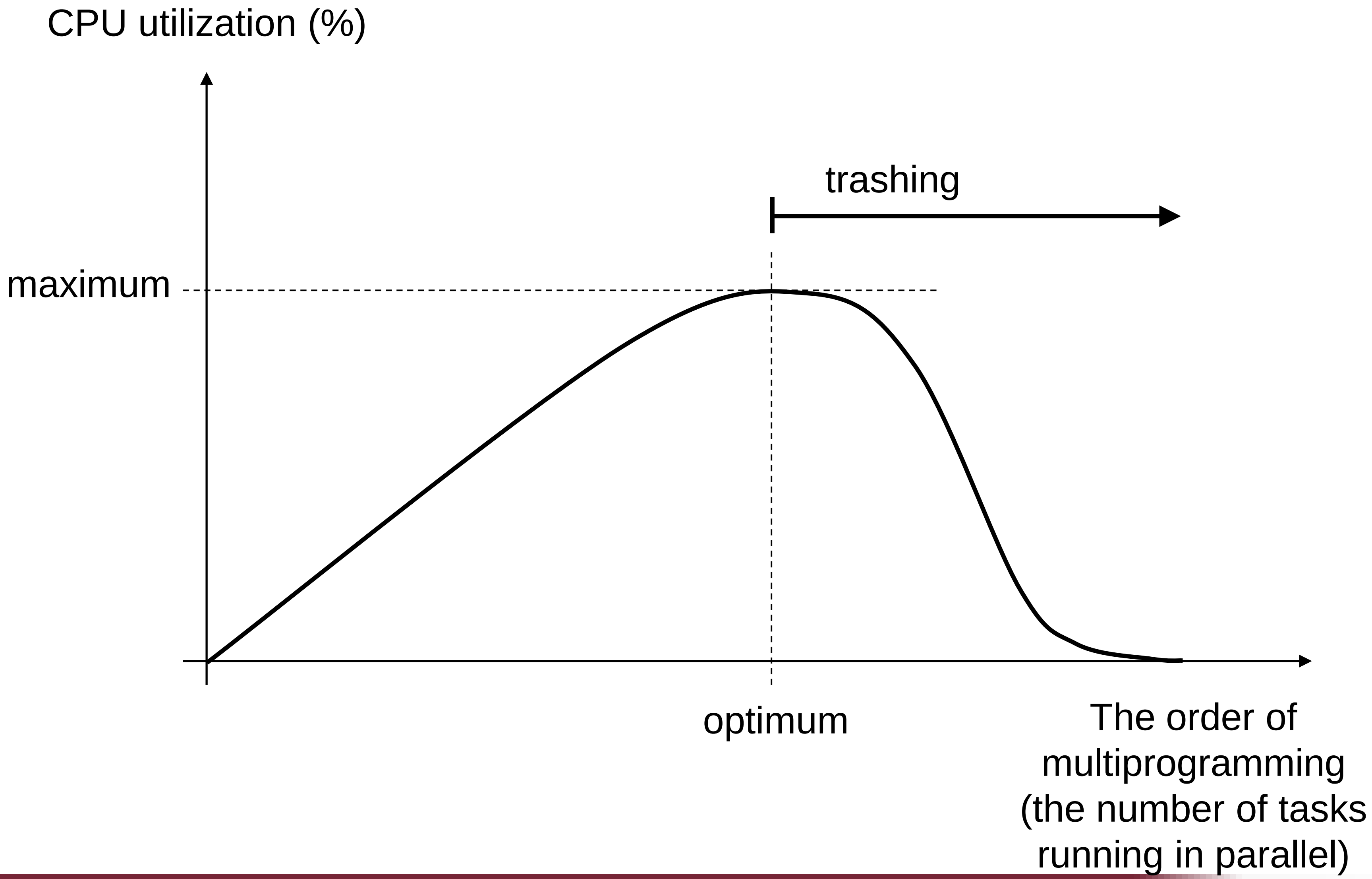
Performance of virtual memory 2.

- If a page is not in physical memory:
  - It is slower by several order of magnitude to load it from the permanent storage than from physical memory, i.e., if it happens regularly the system will slow down drastically (without virtual memory it could not run at all, though)
- The achievable speed is primarily determined by the frequency of page faults
  - Frequent page faults can reduce the performance of the system drastically...
  - Successful anticipatory paging can increase the performance also!

Thrashing

- How we can determine the number of physical memory frames assigned to a process?
  - Too few: Large number of page faults(trashing).
  - Too many: Other processes cannot get enough physical memory
- DEF: The performance reduction caused by frequent page faults is called trashing
  - While handling a page fault a new page fault appers
  - Read request due to handling page faults are queued in the queue of the permanent storage, and the queue grows continuously
  - The CPU waits for handling page faults
  - If there is a long term scheduler it will term the situation as an I/O intensive scenario, and therefore it allows new processes to enter the system...
    - The situation even get worth in this case!

Trashing (figure)



Avoiding trashing

- Aim: Low page fault frequency (PFF).
- While handling a page fault no new page fault should appear:
  - The wait queue of the permanent storage device may be decreased...
- Local page replacement strategy:
  - The processes cannot take away the physical memory frames from each other
  - The problem cannot spread to other processes
  - The problem is reduced, but not solved
- How many physical memory frames are required by a process to operate efficiently?

Locality

- Statistics: In a given time period processes use a small portion of their virtual address space
  - Temporal
  - Spatial
- Trashing:
  - Allocating the required of physical memory frames
    - No trashing
    - Some page faults if the locality changes
  - Smaller number of frames: trashing

Working-set

- Based on locality
- The set of pages of a process that is used/referenced by the process in a given time frame (working-set windows)
- Based on the working-set the size of the working-set can be determined
- The required physical memory size (D) can be also computed in the knowledge of the working-set of the running processes

$$D = \sum WSS_i$$

Application of working-set

- The OS measures the WSS for all running processes
  - If there exists free physical memory frames:
    - Frame request can be fulfilled from the free set of frames
    - New processes are allowed to enter the system (there is free memory)
  - If there exists no free physical memory frames:
    - A victim process must be selected
    - The victim must be suspend, i.e., it is a real swap out
    - The physical memory frames of the victim can be used as free frames for the other processes
- Trashing can be avoided by keeping the level of multiprogramming at the optimum
- In practice it requires lot of resources (complex):
  - A simpler solution must be found…

PFF based optimization

- Trashing: PFF is high for the process
- PFF can be measured in a relatively simple way
  - Low PFF: the process has to many physical memory frames
  - High PFF: the process has to few physical memory frames
- Low and high limits must be specified somehow
  - If PFF is over the high limit: the process gets a physical memory frame
  - If PFF is under the low limit: a physical memory frame is taken away from the process
    - Only if there exists no free physical memory frames in the system
    - If there is no free physical memory frames at all: a process may be suspended

Embedded systems

- Memory handling may be drastically different in embedded systems, but we do not go into the details, some facts:
  - A large portion of CPUs built into embedded system have no MMU (or the MMU is not configured):
    - Paging, virtual memory based operation, process separation, etc. cannot be used due to it
  - If there is an MMU and that is configured:
    - Limited or not writable permanent storage makes using a page file impossible
    - Virtual memory cannot be used in real-time systems
      - In case of a page fault how can we give an upper bound on execution time?
    - Primarily aims of using an MMU in real-time or safety critical embedded systems are process separation and/or virtualization