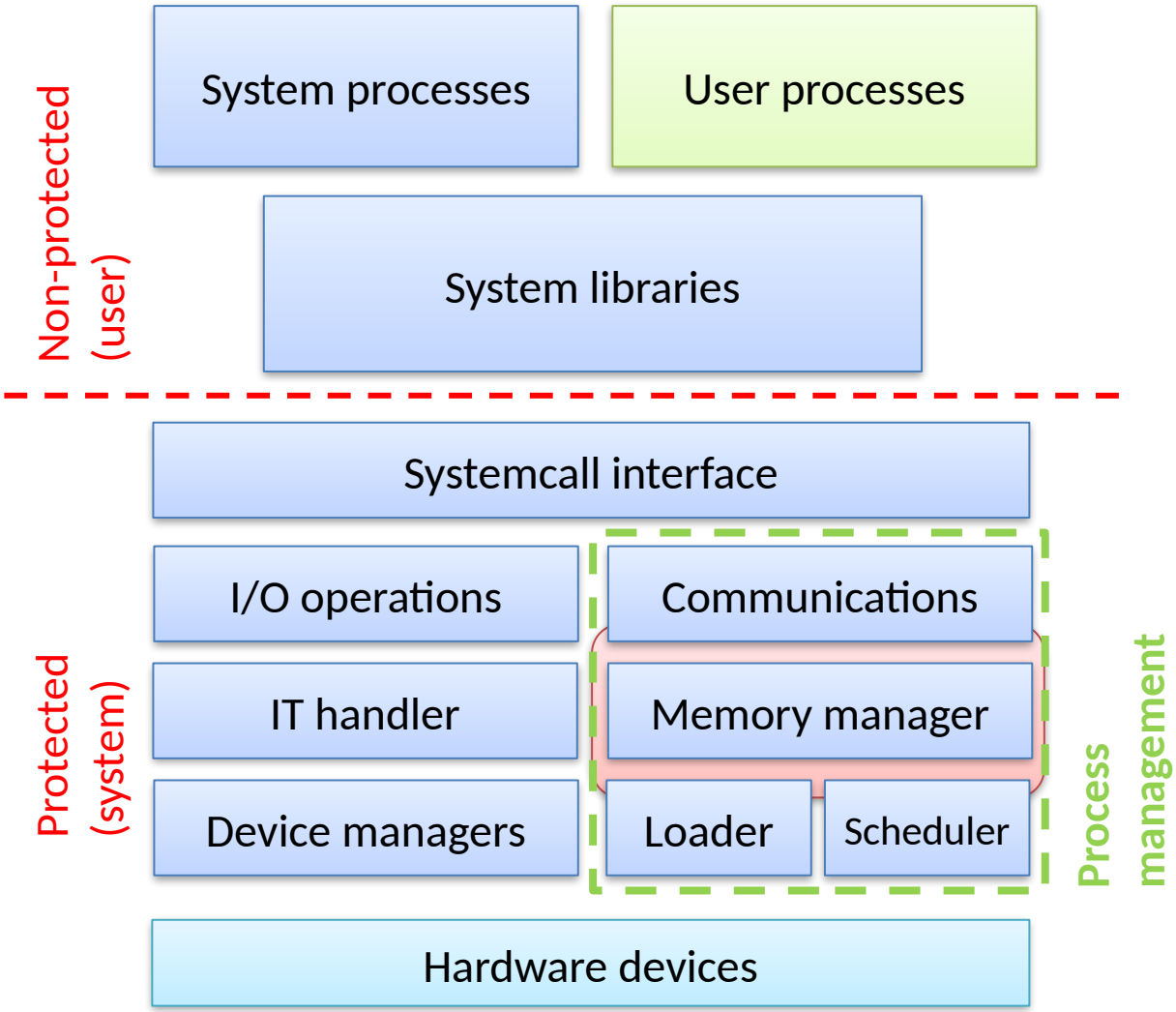
Operating Systems – Memory management

Lecturer: András Millinghoffer
milli@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Artificial Intelligence and Systems Engineering (MIT)

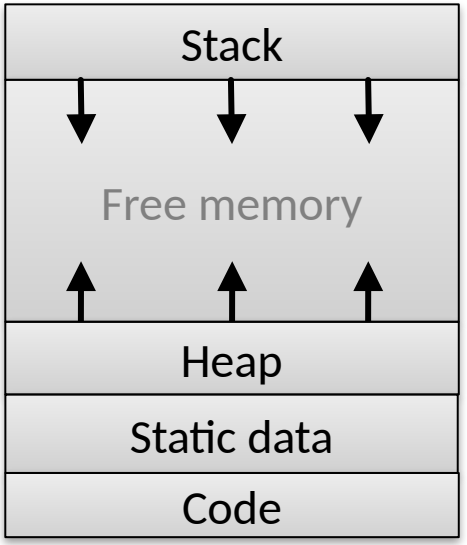
The slides of the latest lecture will be on the course page. (https://www.mit.bme.hu/eng/oktatas/targyak/vimiab00)

The main blocks of the OS and the kernel (recap)



Data structures of the tasks (recap)

- Activities performed by programs
  - Tasks have state and life-cycle
  - Tasks have own and administrative data structures
- **Program data** (in the task's memory range)
  - Code
  - Static allocated data
  - Stack: temporary storage, e.g. for function calls
  - Heap: runtime (dynamic) allocated memory space
- **Administrative data** (managed by the kernel)
  - Task (process, thread) descriptor
  - Unique ID (PID, TID)
  - State
  - Context of the task: the descriptor of the execution state
    - Program counter, CPU registers
    - Scheduling information
    - Memory management state (MMU state)
  - Owner and permissions
  - I/O state information



Separation of the tasks (abstract virtual machine concept)

- The ideal scenario: every task runs independent of each other
  - No effects on other tasks
  - It seems they running on a separate machine (resources)

- In the reality: not enough resources for each task
  - They have to share the resources (CPU, memory, etc.)
  - Goal: the task (and the user) don't notice this
  - The kernel provides an **abstract virtual machine** for the tasks (virtual CPU and memory)
  - A typical multi-programmed system
    - M processor (1<= M <= 8), N task (N > 10-100)
    - More task than processor (N >> M)
    - N abstract virtual machines have to be assigned to the physical resources
    - In a way that the tasks don't the existence of other tasks, but still sharing the common resources

- Complex activities require more than one task: this makes the situation more complex
  - Communication (IPC) and cooperation schemas have to be provided

Users' perspective

- End user
  - Does not care
  - Is there "enough"
- Administrator
  - How much is there (in use / free)
  - Can we increase it
  - What is the state of the swap space
- Programmer
  - How to allocate / free
  - How to handle virtual addresses
  - How much is available (can I allocate more than what is available / physically present)
  - Are there differences in the handling of different types
  - What errors can occur

Duties of memory management

- Storing and managing task's data structures in the RAM
  - Code and static data are loaded from the HDD
  - Dynamic data: heap and stack
- The kernel allocates some memory for its own data structures
  - Code and static data are loaded at the system boot
- Typically there isn't enough physical memory
  - Multiprogrammed systems: multiple tasks are loaded at the same time ␣ using memory
  - The OS try to provide memory for every task
  - ␣ the OS virtually increase the size of the physical memory: **virtual memory**
- The data structures of the tasks and the kernel should be protected
  - The physical memory sections are separated to ensure safe operation of the tasks (others can't corrupt their data)
  - Memory management provides the separation usually with HW support
  - The kernel manages the occurred errors: general protection fault, illegal addressing ␣ the faulty task will be stopped
- Supporting communication
  - Tasks may communicate with each other, or with the kernel
- Increasing efficiency
  - Shared memory ranges (e.g. code), avoiding unnecessary allocations

Typical task memory usage

- The code and data are in the same memory
  - Neumann architecture
- At the start the tasks don’t need all of their code and data
- Dynamic allocation during runtime
  - The tasks don’t mind the size of the physical memory
  - They can dynamically allocate a larger memory section
  - The allocated memory often not used entirely
- There are locality properties
  - Temporal locality: repeated operations on the same data
  - Spatial locality: operations on data near each other
  - Algorithmic locality: patters in the operation
- There can be never used code and data
  - The execution of the program code can different, influenced by external circumstances
  - Many functions are rarely used (exceptions, rare operations)
- The tasks may share some of their code and data

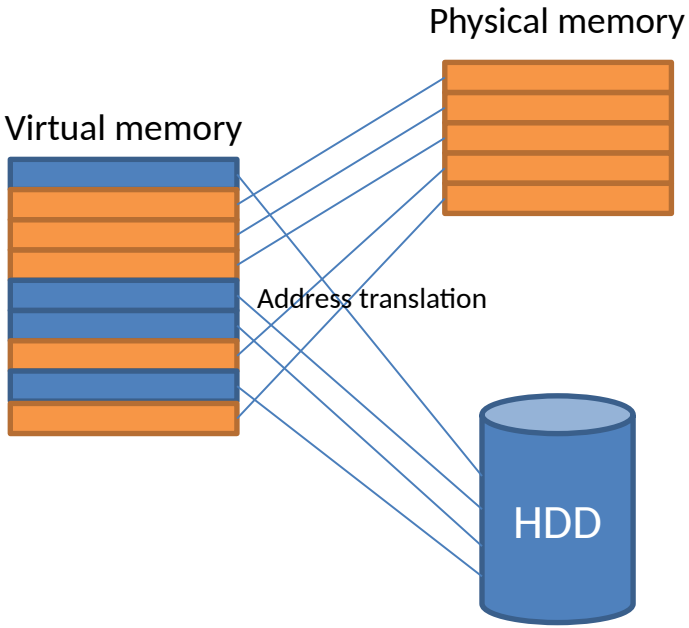
Virtual memory management

- Based on the typical usage and on the abstract virtual machine concept…
- The current OSs using **virtual memory management**
  - It provides a separated, contiguous virtual memory range
  - Manages the association between the virtual and physical memory sections
  - Providing memory for more tasks at the same time
  - Allocating only the necessary memory range for the tasks
  - It is possible to fulfill higher demands than the physical memory
  - The tasks may have shared ranges (read-only)
  - In the meantime the kernel provides separation and protection

- The basic methods of MM
  - Association of virtual and physical addresses: address translation
  - Separation of the tasks memory range with HW support: paging
  - Extending the (fast) physical memory size with (slow) HDD: swap

Address translation and paging

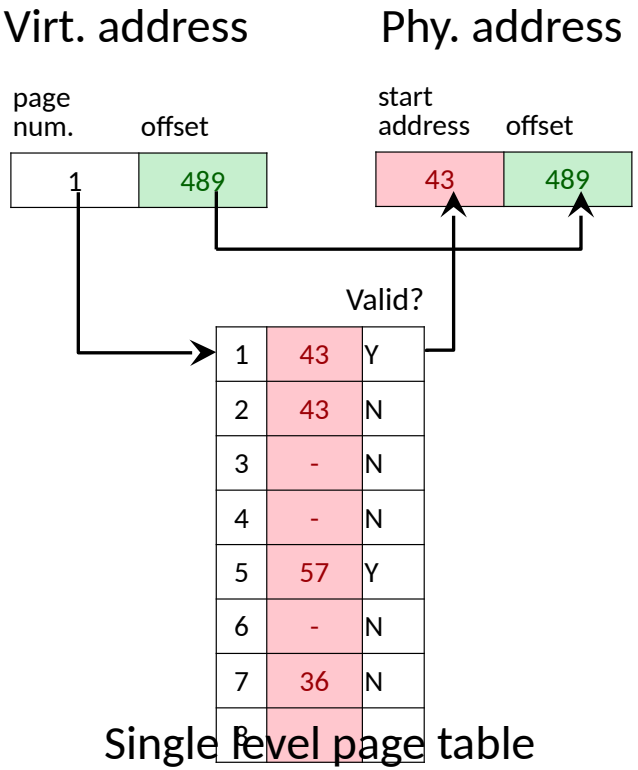
- The tasks of the Memory Management Unit (MMU)
  - Address translation: from CPU address to physical RAM and I/O addresses
- Address translation in virtual MM
  - The tasks reach the whole CPU memory range
    - This is the **virtual range**
      - E.g.: x86-64: 2^{48} = 256 terabyte
    - The physical memory is only a fragment of this
      - It may addressed with the **physical memory address range**
    - If there are no sufficient physical memory, the rest will be stored on the HDD



- The memory organized in pages
  - The virtual range is divided into equal size **pages**
  - The physical memory is also divided into same size **frames**
  - The pages stored on the HDD in **blocks**
  - Page table: association between physical frames, virtual pages and HDD blocks
    - Valid (=1, it's in the physical memory)
    - Dirty (=0, the data is the same as the stored version on the HDD)
    - Accessed (=1, recently used page)
  - Translation Lookaside Buffer (TLB): accelerating the address translation

Address translation and page table

- The common hardware uses hierarchical page table
  - The page table is also divided into pages
  - Relatively fast
  - Not stored entirely in the memory
  - On 64-bit systems it can be tree (4-6 levels)
  - It is used by ARM and x86 architectures also
- The steps of address translation
  - Subdividing addresses
    - Page number index
    - Offset
  - The physical frame is identified by the index
  - The offset is added to the index
- Address (task) separation
  - Usually separate page tables for tasks
    - The page table is part of the tasks context



Single level page table

Swap (or aka pagefile)

- It is used to extend the capacity of the phyisical memory
  - It is divivided into blocks
  - Code and date also can be stored in the blocks
  - CPU cannot access these data directly
    - First it has to loaded into the physical memory
  - Significantly slower than physical memory
- Swapping
  - If the stored data is requested by the CPU, it has to loaded back into the RAM
  - If we need more space in the RAM, some frames have to be moved to the swap
- Initial implementation of swapping
  - The developement started before the paging is used
  - The whole task's memory range was written to the HDD, to free physical memory
  - This caused the **fragmentation** of the RAM and the swap space
    - Variable size „holes" appeared in the ranges, hard to fill these without gaps
    - They try to manage this by dynamically reordering the memory spaces

Virtual memory management

- Based on paging and abstract virtual machine concept
  - The memory range of tasks are divided into pages (no/less fragmentation)
  - The pages are stored in the RAM or in the swap
  - The HW MMU is configured to support this behavior
  - The MMU interrupts has to be managed
- During task execution
  - The TLB and MMU translates the virtual addresses to physical addresses
  - The protection of the pages is done by the MMU HW
    - IT is generated, when an error occurs
- Managing ITs by MMU
  - Protection fault: the running task try to access and address outside its range
    - E.g.: bad pointer
    - The operation of this task should be terminated
  - Page fault: the requested page is not in the physical memory
    - Before the task continue its operation, the page has to be loaded back to RAM

Managing page faults

- The requested page is not valid (valid=0)
  - A page fault IT is generated by the MMU, the kernel's IT handler starts to run
  - It determines the source of the requested page
    - It is on the HDD swap space
    - Fill-on-demand
      - Zero-fill: e.g. dynamically allocated memory
      - Fill-from-text: for loading code or static data from the HDD
  - It starts loading the data to an empty physical frame
    - If there are no free frames, one of them has to swapped
    - If there is a free frame, the starting address of the frame is stored into the page table
    - This I/O op. may be long, so the task is entered into waiting state
  - The kernel returns from the IT handler (changes to another task)
- When the page load is done
  - The page table entry will be set to valid (valid=1)
  - The task will be ready-to-run
- When the task gets in running state
  - The operation is continues from the instruction which caused the page fault

Further tasks of the kernel's memory manager

- Providing free frames in the physical memory
  - It is a basic condition to solve page faults, but is also required for new allocations
  - This should be done in advance, when the system is lightly loaded ␣ freeing unused frames
  - If there are no free frames and a page fault happens ␣ page swap
  - There are multiple swapping strategies (see later)
- Administering
  - The pages of virtual memory – **page table**
  - The frames of the physical memory – **page frame data**
  - The swap space on the HDD – **disc block descriptor and swap map**
- Further tasks
  - Updating the page table with the MMU
  - Storing frames to the HDD which are cannot fit into the RAM
  - Loading requested pages from HDD
  - If required whole tasks may be swapped to the HDD

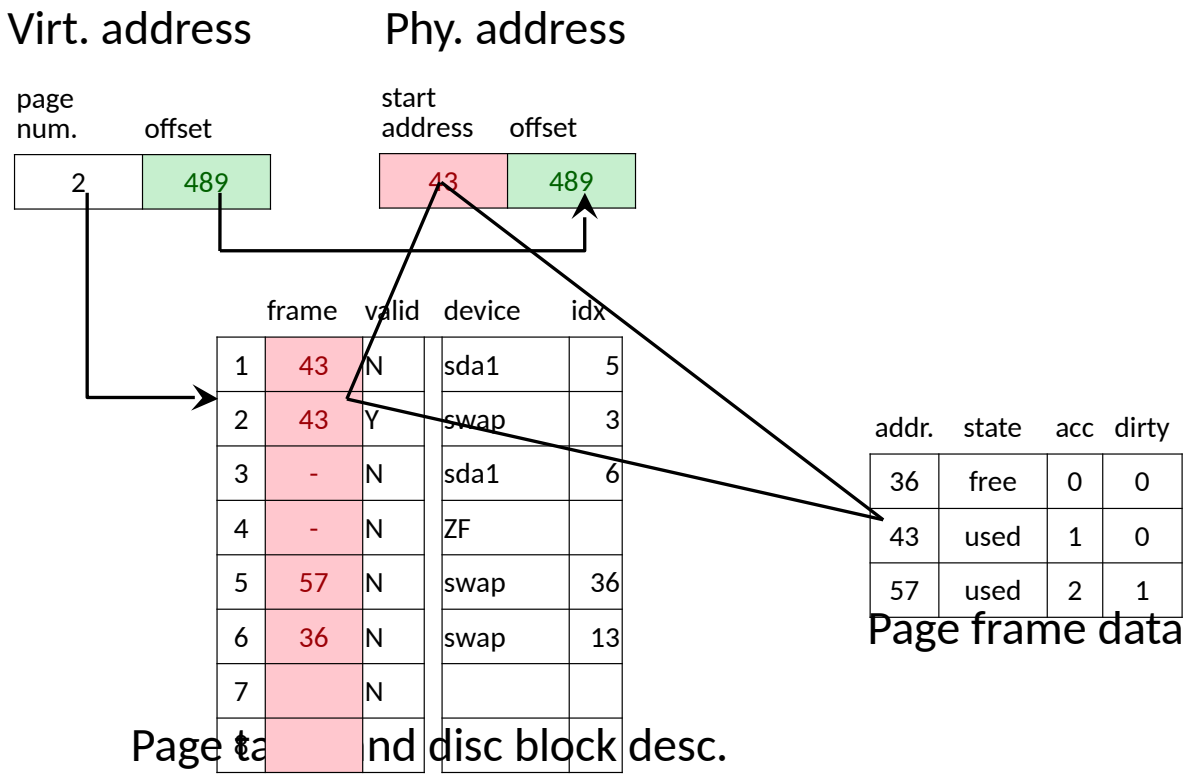
The data structures of virtual memory management

- **Page frame data** (pfdata) entry (kernel)
  - Every frame has one entry, indexed by the starting address of the frame
  - State: free, used, under DMA op., etc.
  - Reference counter: how many task uses this frame

- Kernels **page table entry** (part of task's context)
  - There fields used by the MMU HW
    - Index of the page
    - Frame identifier (where is (was) the page in the RAM)
    - Valid bit: =1 if the page is in the RAM
    - Dirty bit: =1 if the page is written since it is in the RAM
    - Accessed bit: =1 if the page is „recently" accessed
    - Read-only bit
  - There are fields which not managed by the MMU (HW dependent)
    - Page state: in RAM, on disk, fill-on-demand
    - Task ID, copy-on-write bit, permissions, etc.

- **Disk block descriptor** (kernel)
  - The disk ID: which file on which disk
  - Block index
  - Type: swap, fill-on-demand

References between date structures and address translation



Performance boosting techniques: fill-on-demand

- The tasks are allocating memory dynamically (e.g. malloc())
  - After the allocation the memory is uninitialized, the contents is undefined
  - Therefore a physical frame is not allocated, only a page table entry is generated
- Operation of a fill-on-demand entry
  - The kernel don't allocate a physical frame during malloc()
  - The new page table entry will be marked with fill-on-demand flag: fill-on-demand/zero-fill (ZF)
  - When the task tries to access the data first time
    - The MMU generates a page fault IT
    - The kernel's IT handler detects that a new frame has to be allocated from the free frames
    - Based on the flag, the frame will be filled with zeros or data from the disc (fill-from-text)
    - After the IT returns, the task can access the data
- With this technique the tasks can allocate memory efficiently
  - Only allocating resources when they are actually needed

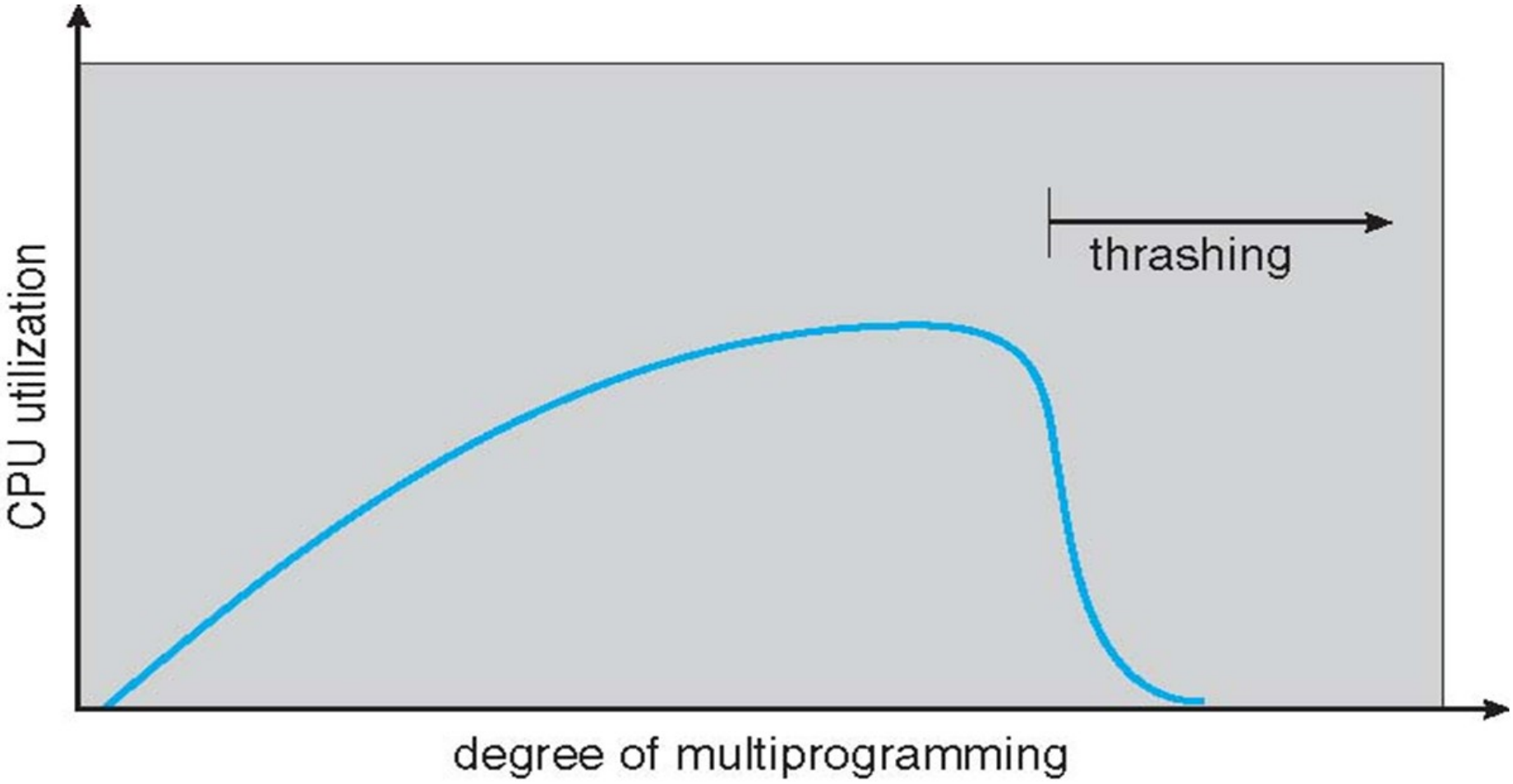
Performance boosting techniques: copy-on-write (COW)

- The number of used physical memory frames may be decreased with page sharing
  - More than one task can use the same frames
  - No problem when there are just read operations
  - In the case of writing, the frame will be duplicated and the writer task gets the new instance
- Memory management of the processes created by fork() system call
  - The fork() in UNIX systems is used to create a new process by duplicating the memory range of the caller task
    - The new process (child) is the exact copy of the parent
  - The child process inherits the parents page table entries, no new physical frame is allocated
    - The reference counter is increased in the page frame data
    - The read-only (RO) and copy-on-write (COW) flags are set
  - The two process are sharing the same frames and swap space
  - When one of the tasks tries to write one on the frames
    - Due to the RO bit, a HW IT is generated
    - The kernel's IT handler detects the RO and COW bits, so new frame will be allocated, and the data is copied
    - The RO and COW bits are cleared on both frames
    - Returns from the IT and the task can continue its write operation
  - Very efficient way to create new processes
- COW is a common technique in UNIX and Windows also

Which pages should be in the physical memory?

- A high page fault frequency (PPF) is disadvantageous
  - Managing page faults usually introduce
    - A high number of ITs ␣ high number of context changes
    - Additional I/O operations
  - The tasks execution is interrupted, waiting state, re-scheduling
    - A CPU intensive task may become I/O intensive, however the task don't execute any I/O operations
  - The overhead is getting higher ␣ system performance degradation
    - **Thrashing**: high PPF, severe performance degradation
    - Managing a page fault may introduce another page fault, more and more I/O operations, CPU utilization decreases
    - It may be managed by a medium-term scheduler, constraining the number of tasks
- If a task has many pages in the RAM
  - The number of page faults will be lower for this task
  - The other task will won't get enough physical memory
- If a task has few frames in the RAM
  - A high number of tasks may execute simultaneously
  - Each task will generate a high number of page faults ␣ slower operation

The emergence of trashing



Paging strategies

- Demand paging
  - Only runs when a page fault occurs, loads only the requested page
  - Simple, only the requested pages will be loaded to RAM
  - Every time a new page is requested, a PF will be generated
    - This can significantly slow tasks down (CPU intensive task will become I/O intensive)
    - E.g. iterated over a large data structure
    - Every PF will enter the task into waiting state ⎵ high number of context changes, inefficient I/O operations
- Anticipatory paging
  - It's try to figure out which pages will be requested
    - It is based on the locality properties of the tasks
    - And the tasks PF rate, higher PFF ⎵ more pages should be loaded to RAM
  - A good prediction will significantly lower the number of page faults
  - A „less good" prediction may load pages which are unnecessarily using the RAM at the moment
    - If there are enough RAM, it isn't a serious problem
  - Observing the global PFF, the number of active tasks can be determined
    - In order to avoid trashing, a number of active tasks should be decreased

Page replacement algorithms – introduction

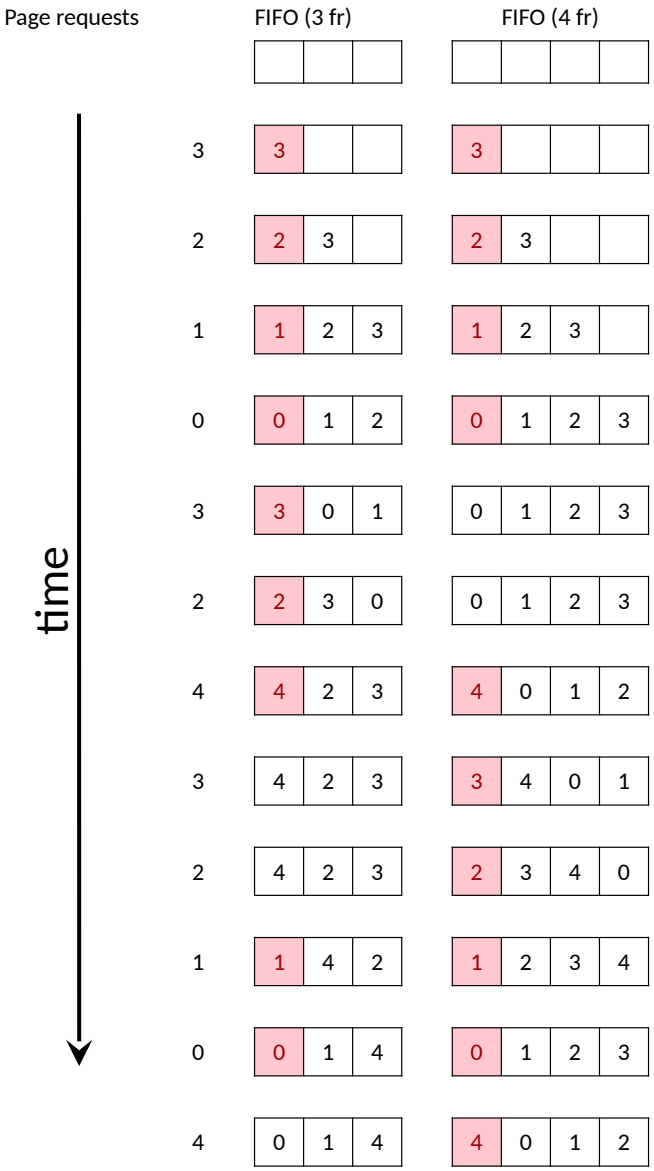
- A page has to be loaded, but there are no free frames
  - A used frame has to be picked, which will be moved to the swap space and mark as free
- The picking based on
  - Accessed bit: Is it used „recently”?
  - Dirty bit: Is the contents modified?
  - The allocation time of the frame
  - The latest access time of the frame
  - Reference counter: how many task uses this frame?
- The properties of page replacement algorithms
  - Ideal solution: seeing the future
    - It is similar to the estimation of the CPU burst of a task
    - In practice, the prediction is based on the behavior in the past
  - The frame to swap may be chosen from the actual task's (**local**) range or from the **global** range
  - Algorithms (details on following slides)
    - FIFO: the page loaded first will be replaced first
    - Second chance (SC): oldest and not accessed page
    - Least Recently Used (LRU): the oldest accessed page
    - Least Frequently Used (LFU): the most rarely used page
    - Not Recently Used (NRU): Non accessed and non modified page

FIFO page replacement and Bélády's anomaly

- Simple, low overhead, backward looking algorithm
- When a page is associated with a frame (loaded into the RAM) it will be queued in a FIFO queue
- If a free frame is needed
  - From the front of the queue a page table entry is picked
  - The associated frame will be written to the HDD
  - The frame will be freed and the new allocation can be made
- What happens if we increase the available frame count?
  - It is expected that PF rate will decrease
  - In practice this isn't always the case: from time to time the PF rate will increase when the available frame count is higher
  - This is called the **Bélády's anomaly** (László Bélády, IBM virtual memory management)

- Evaluating FIFO algorithm
  - Simple, simple implementation, low overhead
  - The estimation of the future demand is poor
  - It cannot differentiate between modified and unchanged pages, it may replace unchanged pages ⇏ unnecessary disk operations

Example of Bélády's anomaly



Number of PF when the FIFO size is 3: 9
Number of PF when the FIFO size is 4: 10

Second chance (SC) page replacement algorithm

- SC also uses FIFO data structure
  - When a page is associated with a frame it will be queued in a FIFO queue
- From the front of the queue the page table entry is picked
  - If (accessed_bit == 1) //MMU sets this bit
    - accessed_bit = 0;
    - The page table entry is moved back to the back of the queue
  - Else
    - The page is not used recently, so this will be replaced
- Evaluating SC algorithm
  - Simple, simple implementation, low overhead
  - The future demand estimation is better than pure FIFO
    - The accessed bit is showing the recent usage of a page
  - It still cannot differentiate between modified and unchanged pages, it may replace unchanged pages ⇨ unnecessary disk operations

Least Recently Used (LRU) page replacement algorithm

- The pages are ordered in a queue based on their last access time
  - There are multiple implementations based on the MMU HW
  - The page table entries are stored in an ordered chained list
  - There are also a reference counter for each page
  - The page with the smallest value will be replaced

- Evaluating LRU algorithm
  - Complex, high overhead, is should be implemented only with HW support
  - It gives a very good estimation of the future usage of the pages
  - Problem: a new page can be replaced with high priority
  - It still cannot differentiate between modified and unchanged pages, it may replace unchanged pages ⇨ unnecessary disk operations

Least Frequently Used (LFU) page replacement algorithm

- The simplified version of the LRU alg.
  - It can be implemented without HW support
  - The OS periodically checks that a page is used or not, if used a counter will be incremented
    - This not happens with every memory operation like in LRU
- Evaluating LFU algorithm
  - Medium overhead
  - It's a rather good estimation of the future uses of pages
  - Problem: a new page can be replaced with high priority
  - The counter can overflow (aging can handle this)
  - It still cannot differentiate between modified and unchanged pages, it may replace unchanged pages ⇒ unnecessary disk operations

Not Recently Used (NRU) page replacement algorithm

- The refined version of the SC alg.
  - Beside the accessed bit, the dirty bit is also taken into account
  - With this two bit, a „priority” is assigned to each page
    - acc=0, dirty = 0 ⇨ pri = 0
    - acc=0, dirty = 1 ⇨ pri = 1
    - acc=1, dirty = 0 ⇨ pri = 2
    - acc=1, dirty = 1 ⇨ pri = 3
  - If a frame is needed, the one with the smallest priority will be chosen

- Evaluating NRU algorithm
  - Low overhead, good utilization of the HW bits
  - Better estimation than SC
  - It can differentiate between the modified and unchanged pages

Page locking

- Multiple algorithms suffer from the same problem: the newly loaded pages
  - The newly loaded pages don't have „past", (e.g.: no counter values)
  - Therefore the future is hard to estimate
  - There is high chance this pages will be replaced
- Pages under I/O operation cannot be replaced
  - I/O operations using physical addresses
  - With DMA, these operation can be done without the CPU (in the background, simultaneously)
- **Page locking** can solve these problems
  - A special page lock bit is used
  - The locked pages cannot be replaced
  - The locking is maintained until the end of the I/O operations
  - With LRU and LFU algorithms the page is usually locked until it's first access

Providing free frames: page daemon task

- The page replacement usually happens at the „wrong” time
  - The replacement happens when there are no free frames, so it must happen
  - The free frames can be run out because the high system load ⇒ this isn’t an ideal time for page replacement (context change, I/O, etc.)
  - It would be better to do this when the system load is low
- Page daemon task (kswapd, Working Set Manager)
  - It runs periodically by the kernel
  - It tries to maintain the number of free frames between two thresholds
    - If the number drops below a minimum value, then additional frames will be freed up (e.g.: with NRU alg.), until the maximum value is reached
  - It may maintain the data structures for page replacement algorithms
    - Resetting accessed bit
    - Aging the counters
  - This kernel process may also perform the page replacement also
    - Despite the operation of this task, if the free frames still run out, the page replacement should be performed ASAP

Summary

- The concept of abstract virtual machine in multiprogrammed systems
  - The tasks get their own virtual memory range
- The tasks of the kernel's memory manager
  - Translating between virtual and physical addresses
  - Usually there are not enough physical memory ⇒ swapping
  - The memory is organized into pages/frames/blocks to avoid fragmentation
  - The virtual pages are assigned to RAM frames or disk blocks
  - The memory ranges of different tasks are protected
  - To improve efficiency some ranges can be shared
- Address translation
  - Works with HW support (MMU, TLB)
  - If the HW generates page fault IT, the SW page management steps in
- The kernel manages page faults (PF), it tries to avoid thrashing
  - There are different page replacement algorithms to free physical frames
  - It tries to decrease the PF rate with future page demand allocation