

"640 Kilobyte ought to be enough for anybody."

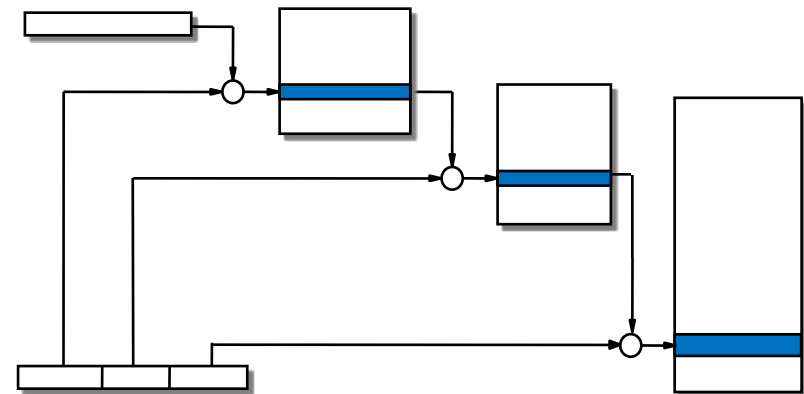
-- *Bill Gates, 1981*

"Wir haben so viel Speicher, den müssen wir gar nicht managen."

-- *Abraham Söyler, 2018*

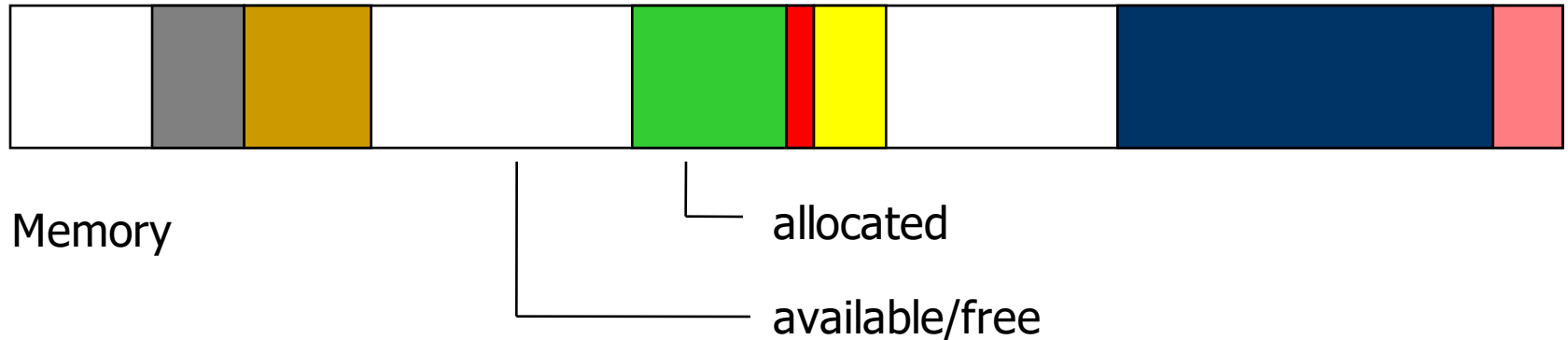
Chapter 7

Memory Management



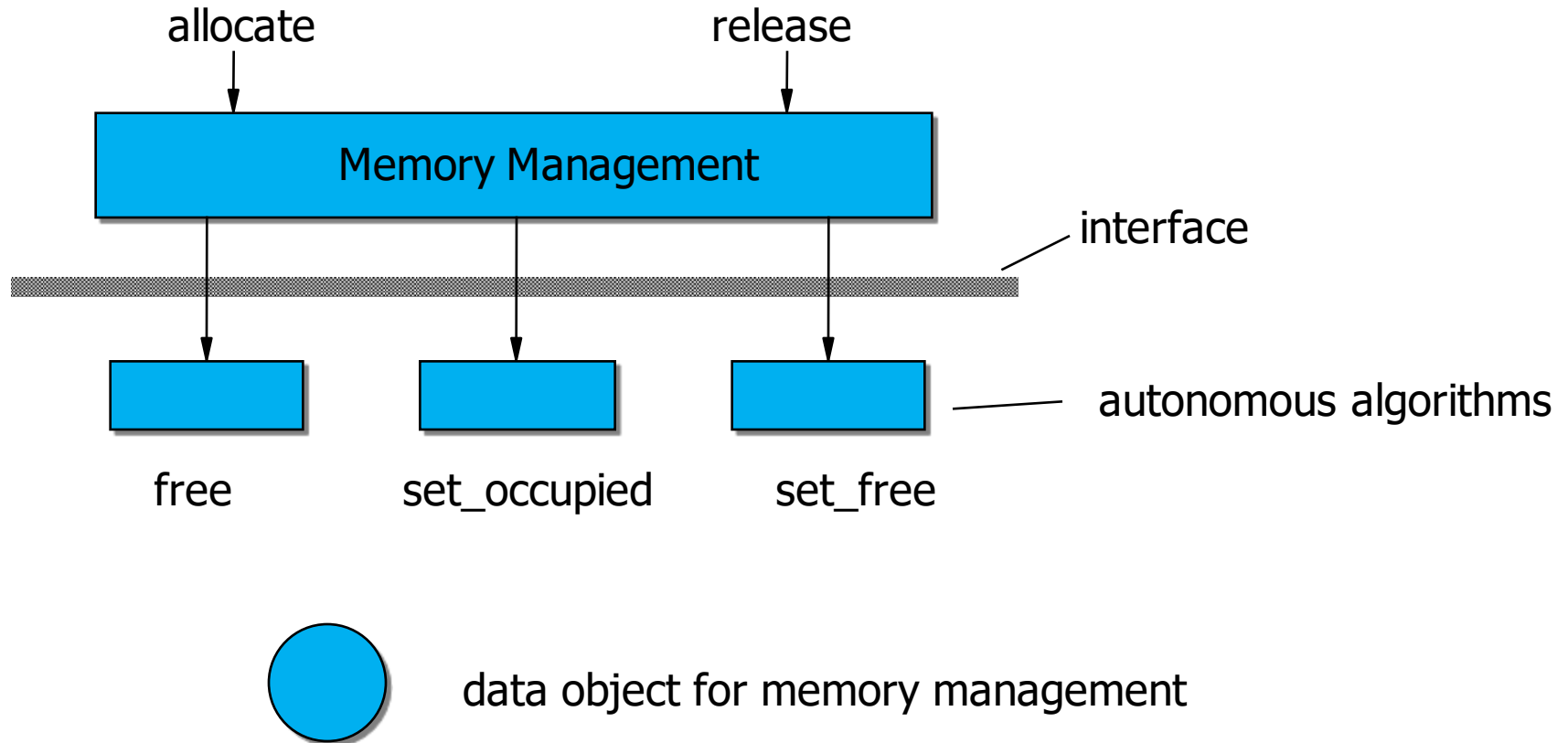
7.1 Allocation strategies

- Problem



- Selection of memory sections/pieces
- Efficiency of algorithms
- Memory usage
- Problem conditions
- Application area: (real) Main Memory (and Swap Space)

Structure of Memory Management



- Memory management strategies can be distinguished based on:
 - Sequence of operation
 - Size of pieces
 - Representation of allocation
 - Fragmentation
 - Allocation strategies (with free pieces)
 - (Re-)integration

Sequence of operation

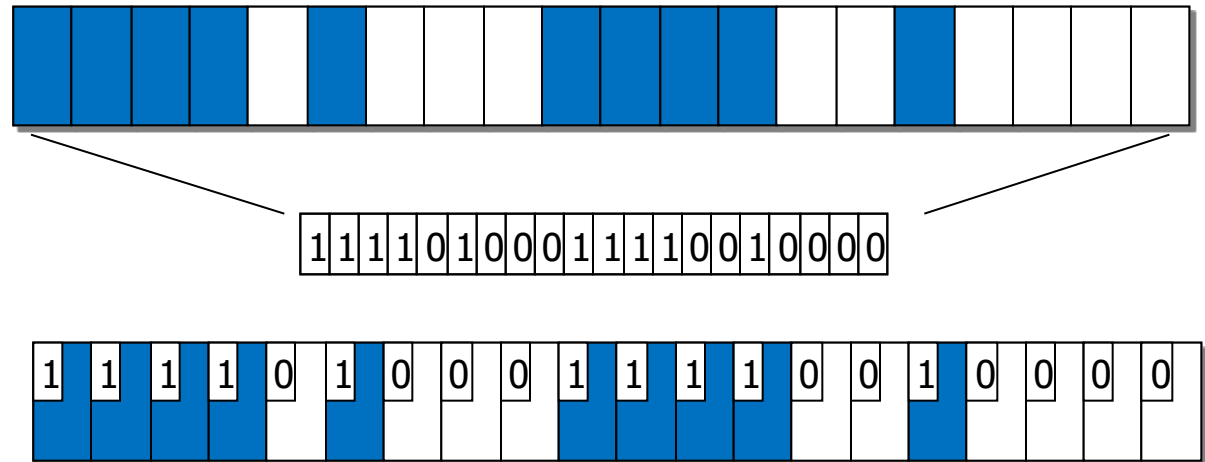
- Allocation and release
 - in same order
 - Queuing approaches, FIFO = First In First Out
 - in reverse order
 - Batch approaches, LIFO = Last In First Out
 - in arbitrary order
 - General approach

Size of pieces

- Constant size
 - $NUM = 1$ (unit size)
- Multiple of constant size
 - $NUM = k$ (unit size)
- Given size of partitions
 - $NUM = k_1, k_2, k_3, \dots$
- Arbitrary size
 - $NUM = x$

Representation of allocation

- How?
 - Vector
 - Table
- Where?
 - Separated
 - Integrated

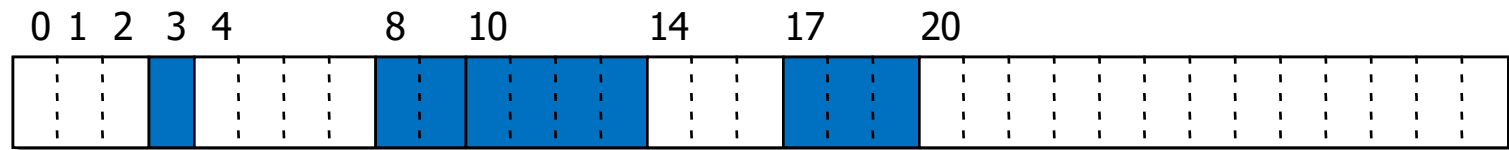


Representation by vector separated and integrated

- Example
 - Main Memory 128 Mbyte (2^{27} Byte)
 - Unit size 512 Byte (2^9 Byte)
 - Sum 262144 Units (2^{18})
 - Representation with 8192 words with 32 Bit

Representation of allocation

- Representation by table
 - Separated representation
 - Holding information about allocation in table
 - Sorting by address and/or length



Sorted by address

Address	Length
0	3
4	4
14	3
20	13

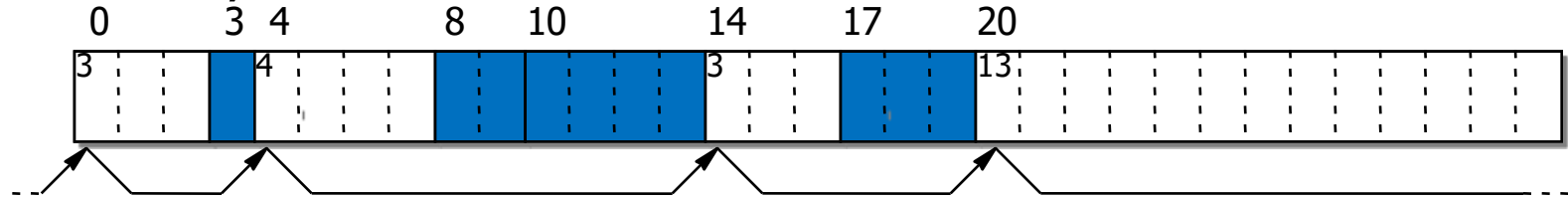
Sorted by length

Length	Address
3	14
3	0
4	4
13	20

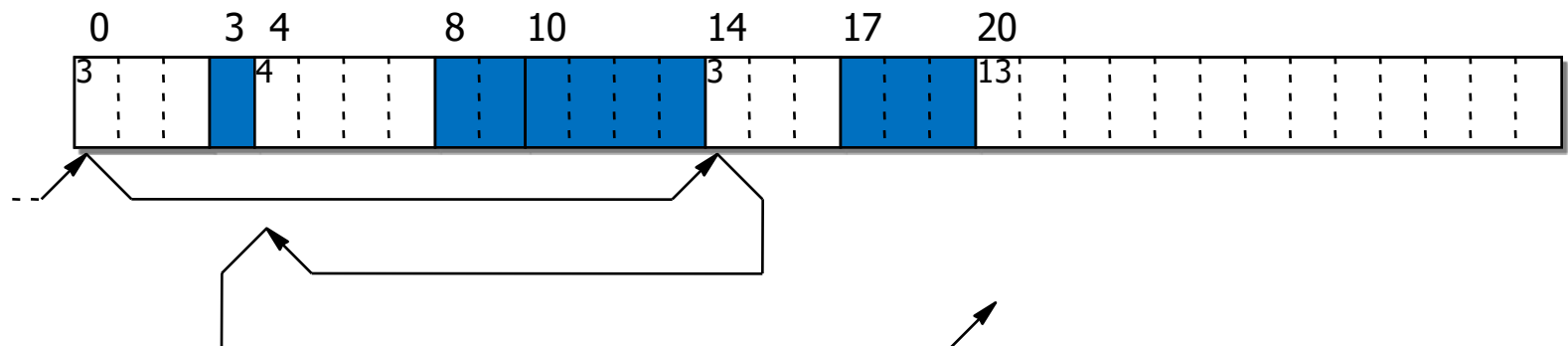
Representation of allocation

- Integrated representation (by table)
 - Pieces identify itself, specify length and provide pointer to next element of free list.

Sorted by address



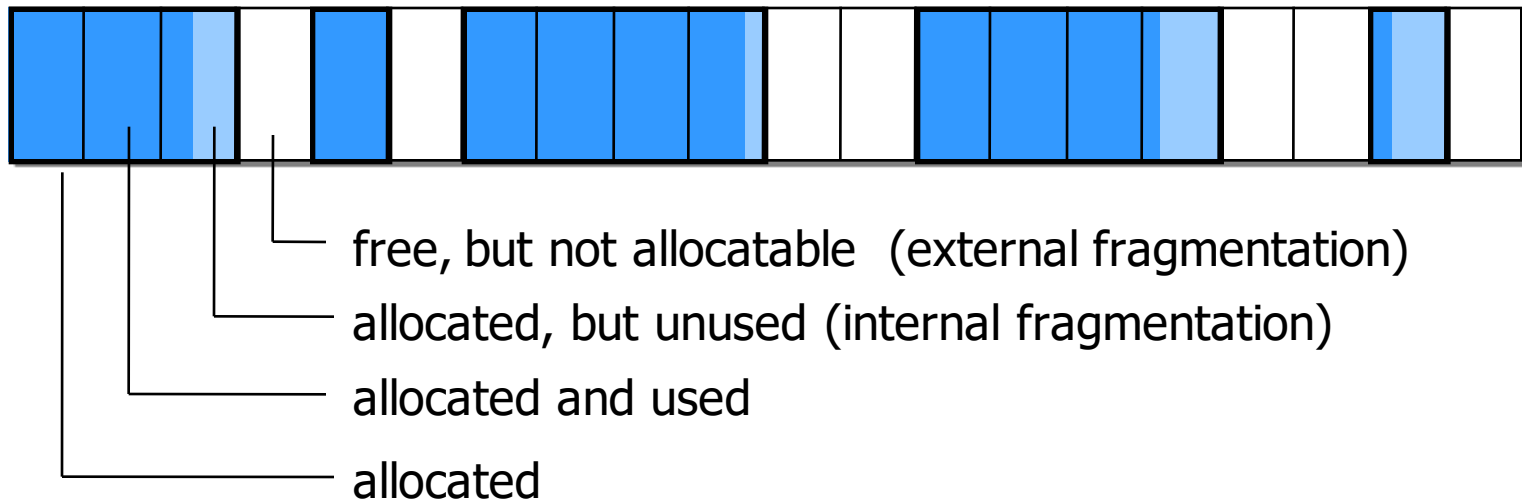
Sorted by length



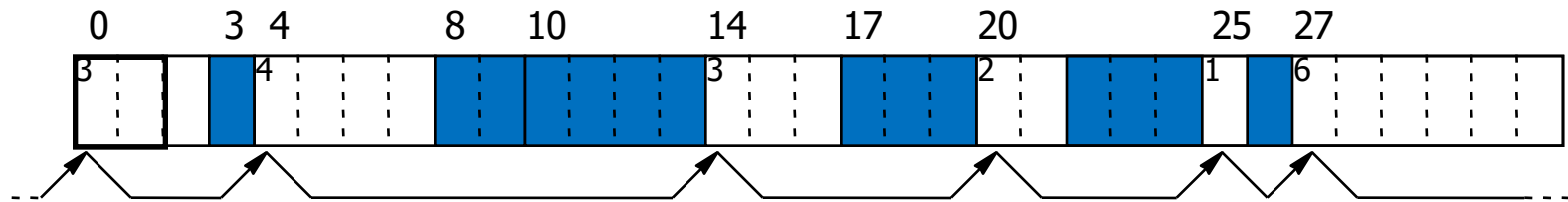
Fragmentation

- Usually memory is allocated for multiple of units.
- Requests therefore are rounded up to the next multiple of units.
- This come with unused parts of the allocated memory.
- The unused piece of memory is called **internal fragmentation f_{int}** .
- Due to the dynamic of allocation and release of pieces it may happen the overall amount of free memory can satisfy a request, but because of the layout of all of the pieces of free memory is cannot be fulfilled.
- So free memory is created, which is not suitable to be used for requests.
- This is called **external fragmentation f_{ext}** .

Fragmentation



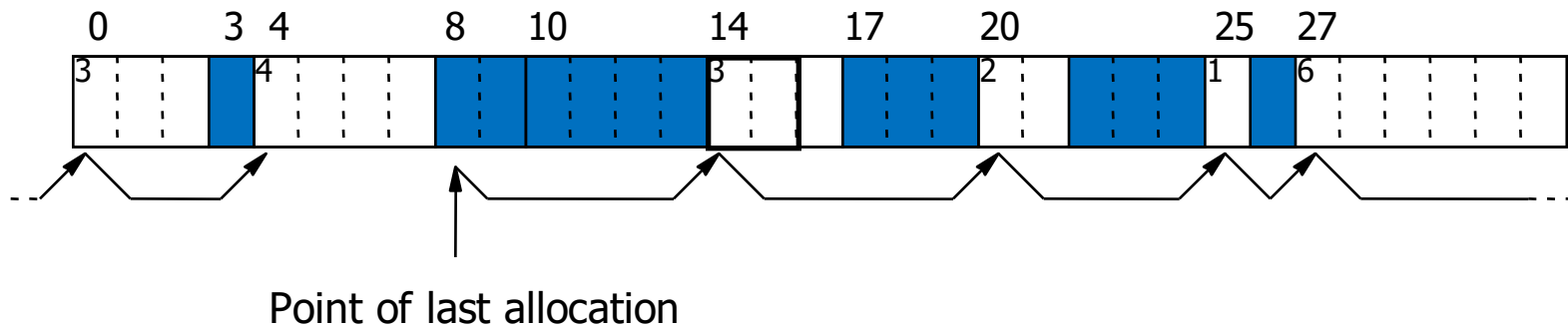
- First Fit strategy



- Search the free list from start.
- Take the first piece of free memory satisfying the request.
- Properties
 - Low search effort (in case of almost empty memory space).
 - External fragmentation
 - Concentration of allocated memory at the begin of the memory space
 - Increased search effort in loaded situations

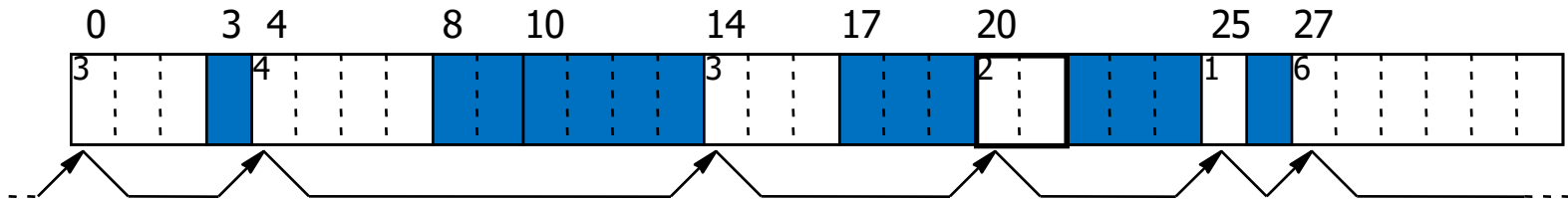
Allocation strategies

- Next Fit strategy, Rotating First Fit strategy



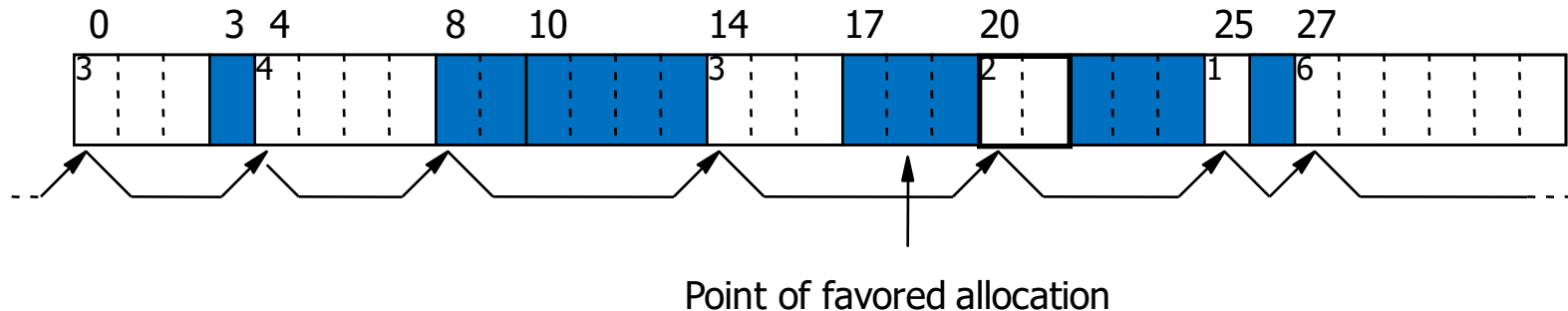
- Cyclic search of list.
- Search start at the point of last allocation.
- Properties
 - Like First Fit, but without concentration at the begin of the memory space
 - Therefore slightly reduced search effort (memory space not empty).

- Best Fit strategy



- Allocation of the smallest piece of memory satisfying the request.
- Properties
 - If sorted by address the whole free list has to be searched.
 - List should be sorted by size of piece of free memory.
 - Usually reduced external fragmentation, because requests for small amount of memory may be served without derogation of larger pieces.
 - But produces very small pieces of free memory unsuitable for any request (external fragmentation).

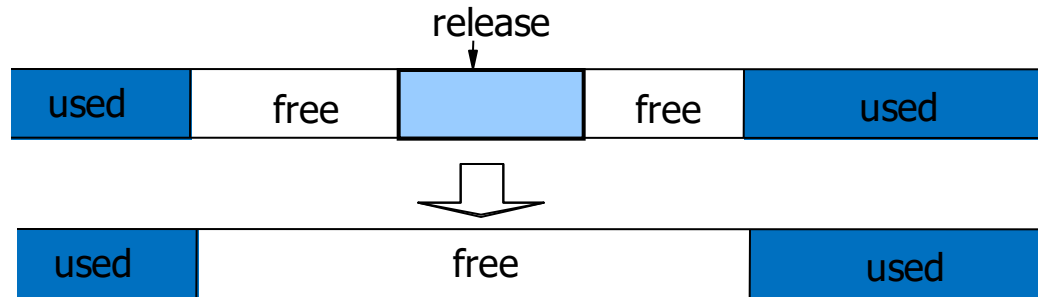
- Nearest Fit strategy



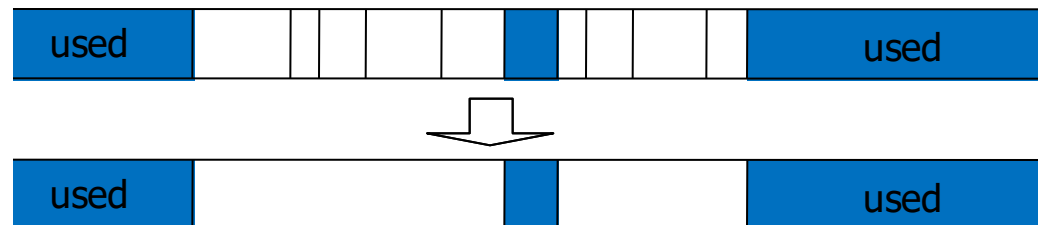
- A favored address is provided.
- Search with First Fit from the point of favored allocation.
- Properties
 - In case of disc space minimizing the movement of disc arm. Especially if the sequence of access is known, the movement of the disc arm can be optimized.
 - File directory information can be located in the middle of a cylinder.
 - In case of expansion of files the blocks to be allocated should be located in the neighborhood.

Reintegration

- Instantly after release



- Delayed aggregation



- Ring buffer



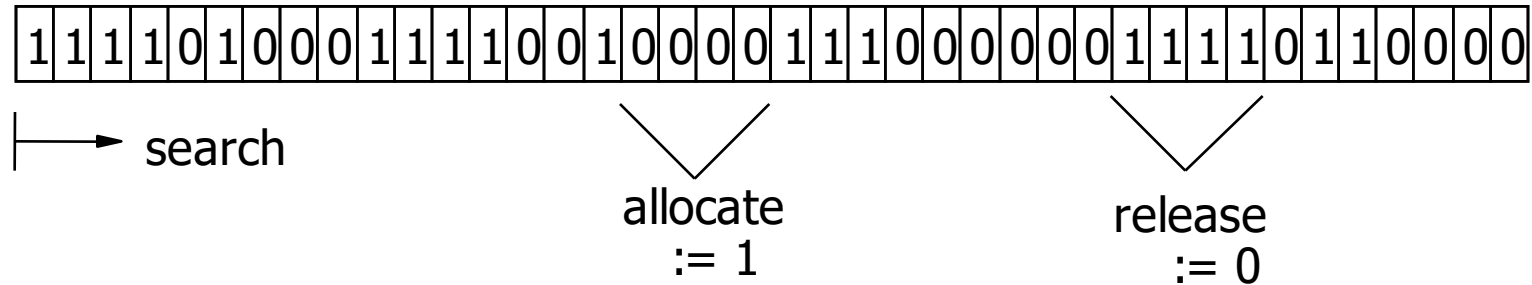
- Allocation and release in same direction (FIFO)
- Fix length of pieces
- No search needed
- No external fragmentation
- Automatic and immediate reintegration

- Stack



- Allocation and release in inverse direction (LIFO)
- Arbitrary length of pieces
- No search needed
- Little external fragmentation
- Automatic and immediate reintegration

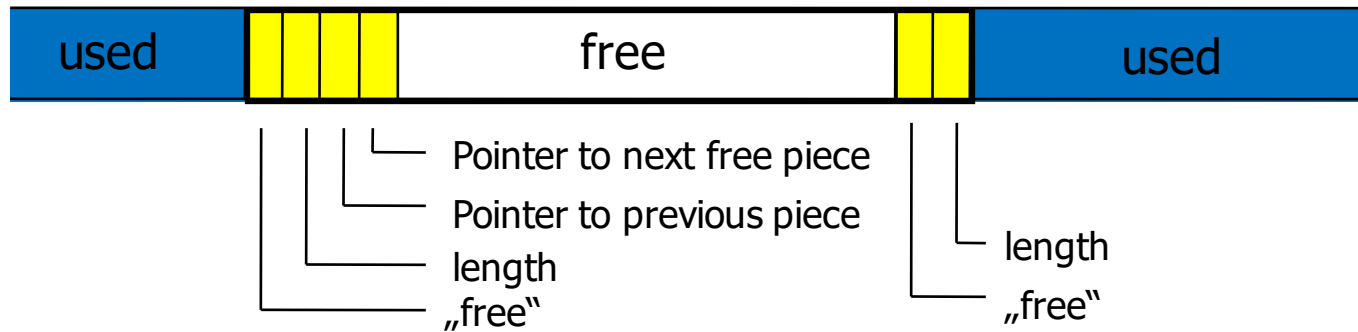
- Vector based approach



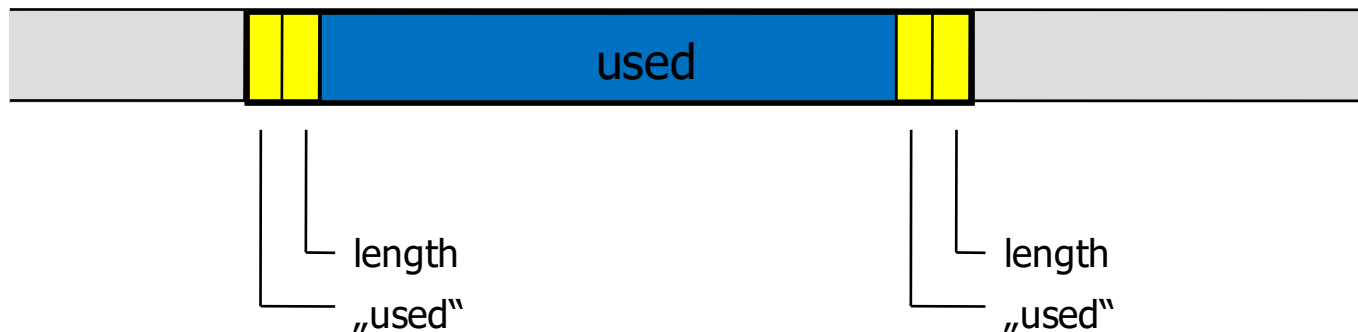
- Allocation and release in arbitrary direction
- Fixed length with $k * \text{unit size}$
- Search for first fitting piece
- Internal and external fragmentation
- Automatic and immediate reintegration

- Boundary tag system

free piece



used piece



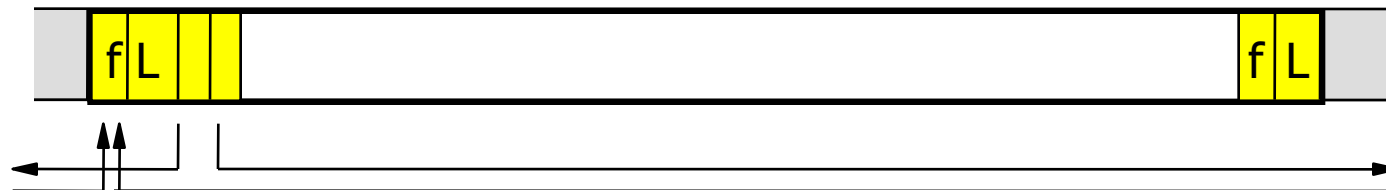
- Label for pieces
- Sorted list by size (length)

Boundary tag system

after release

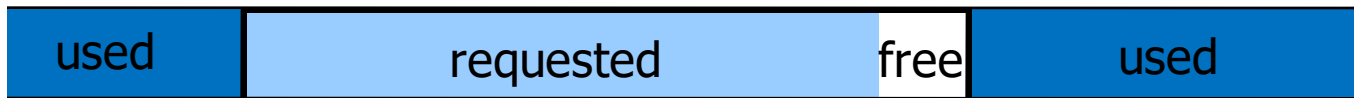


after reintegration

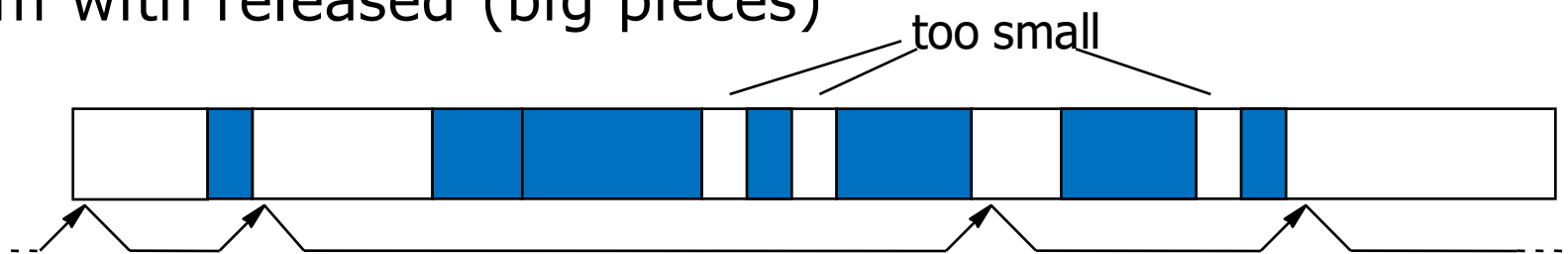


- Properties
 - Operation in arbitrary order
 - Allocation of pieces with arbitrary size (length)
 - Integration of management and representation of pieces
 - Doubly linked list sorted by size of pieces
 - Best Fit search strategy
 - External fragmentation
 - Explicit immediate reintegration using length field to check with neighboring pieces
 - Immediate integration into linked list

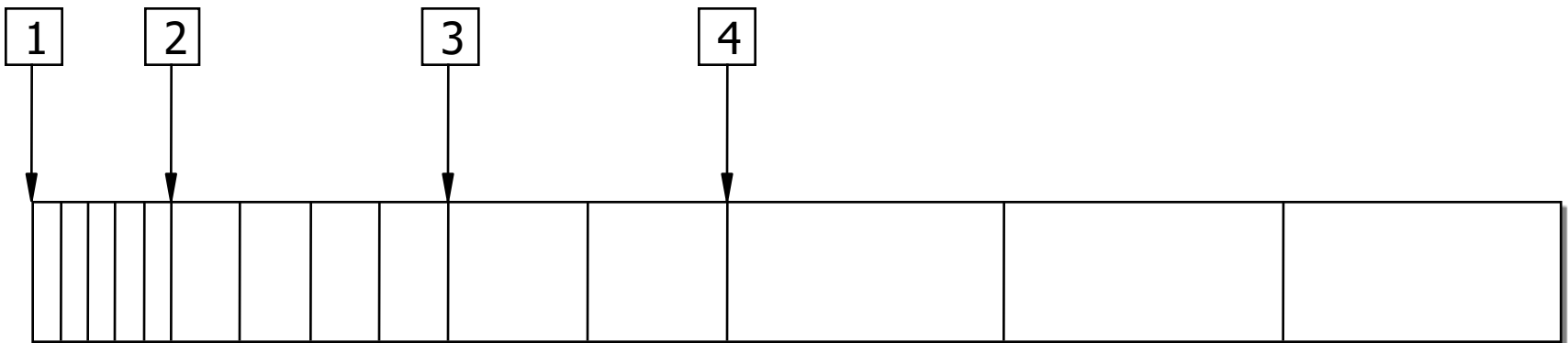
- Reduction of management efforts based on small pieces
- Merge requested piece and small piece (transform external fragmentation into internal fragmentation)



- Avoid integration of small pieces into free list, but merge them with released (big pieces)

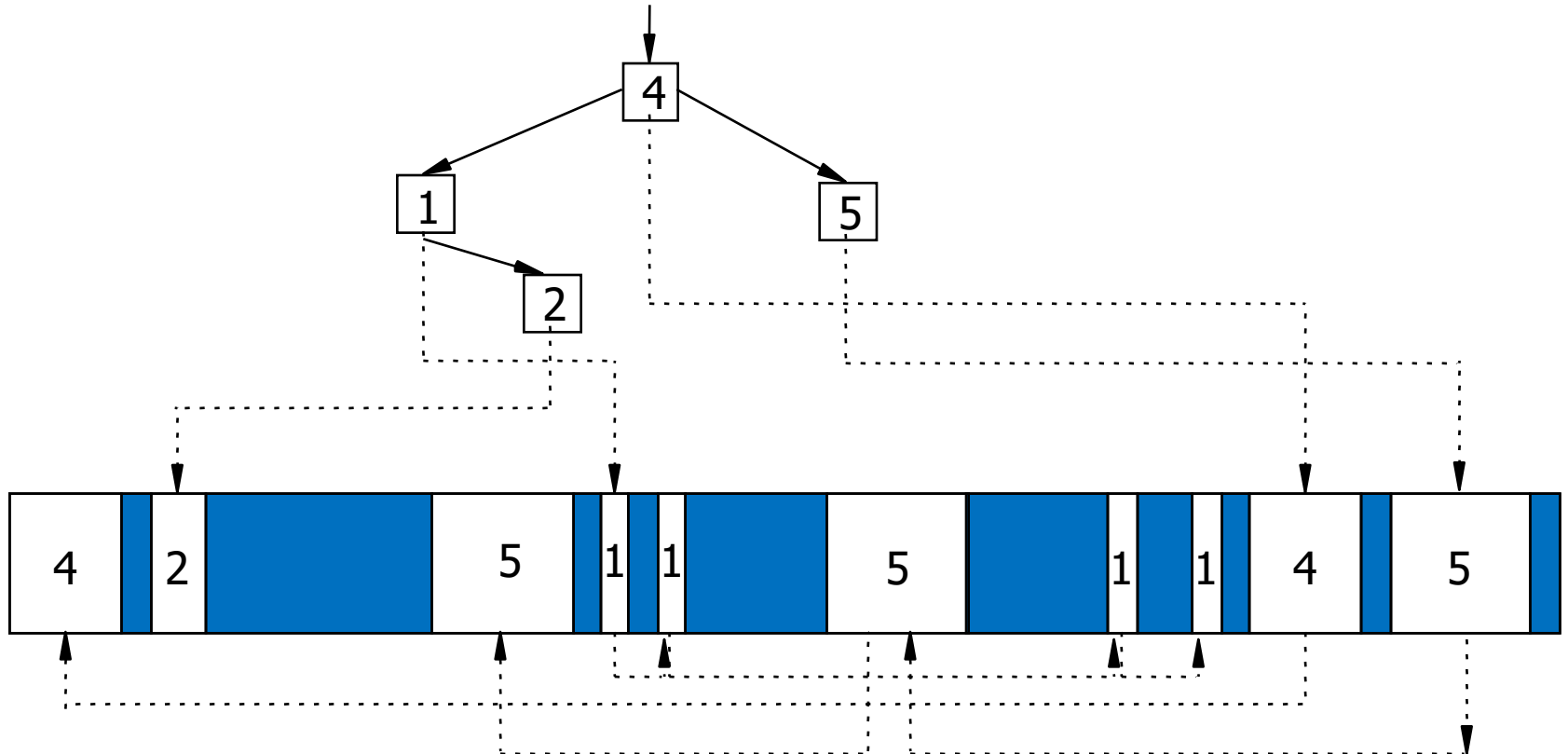


- Cost of search on arbitrary order of allocation and release – $O(n)$
- Reduce search costs
 - Tailored pieces
 - Given size (length) of pieces
 - Provide number of (statistically) frequently used pieces



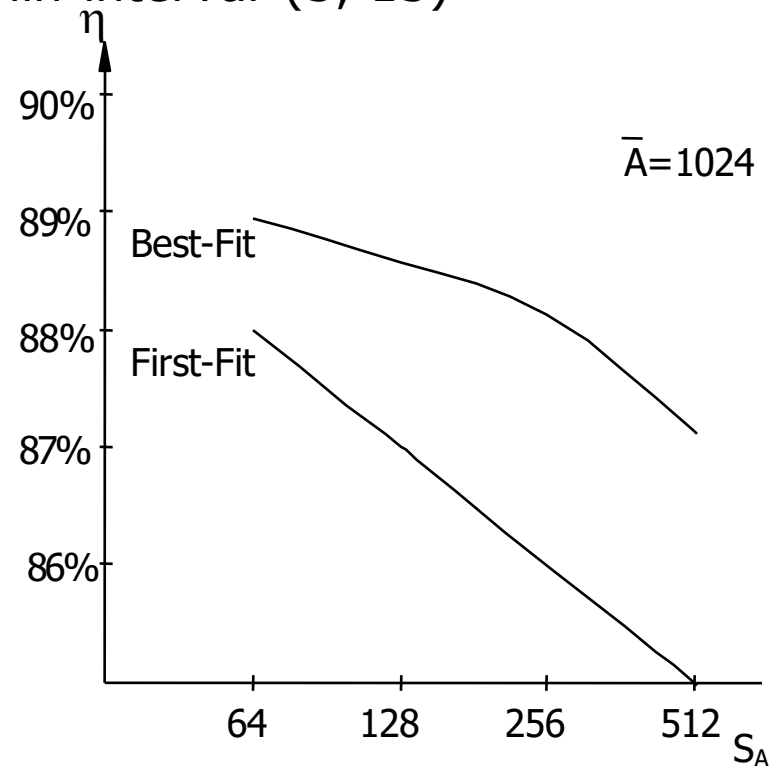
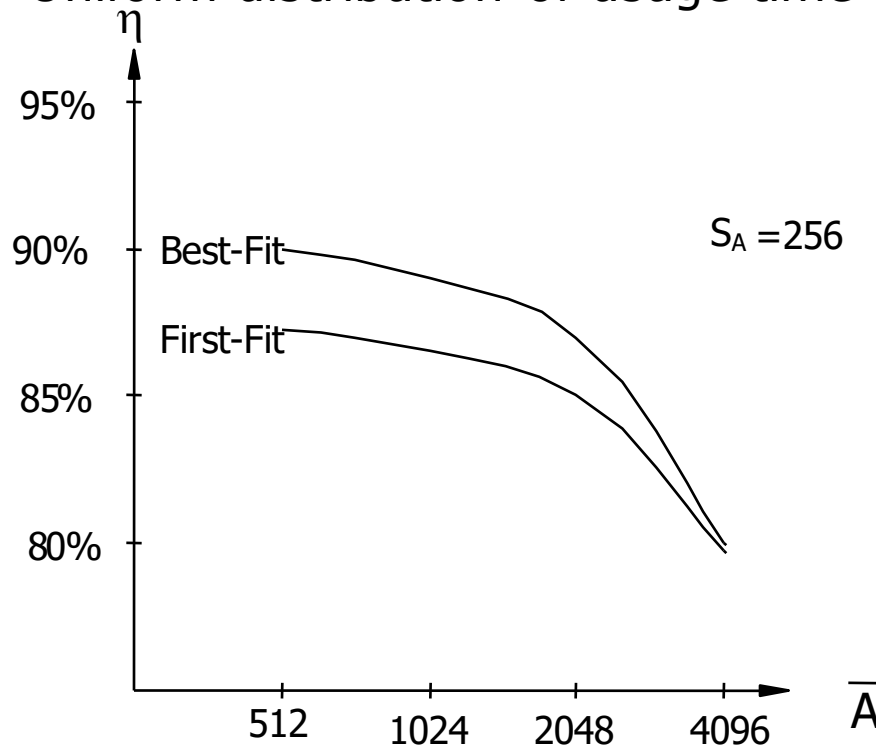
Reduction of search costs

- Example: access by binary tree



Memory usage

- Simulation with 32 K units
- Uniform distribution of requests with mean value \bar{A} and standard deviation S_A
- Uniform distribution of usage time within interval (5, 15)

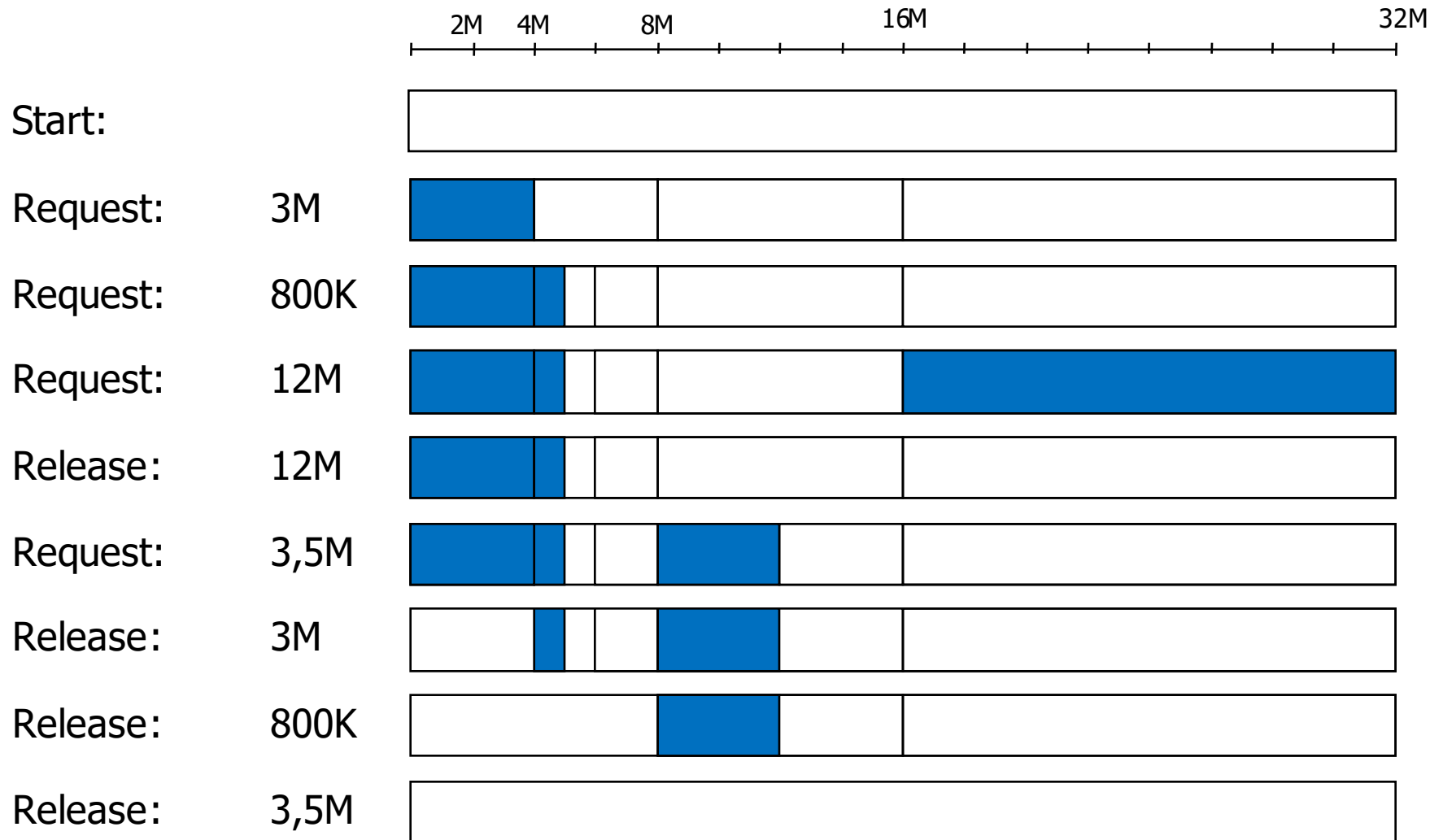


- External fragmentation is increasing with size and variation of request

Buddy system

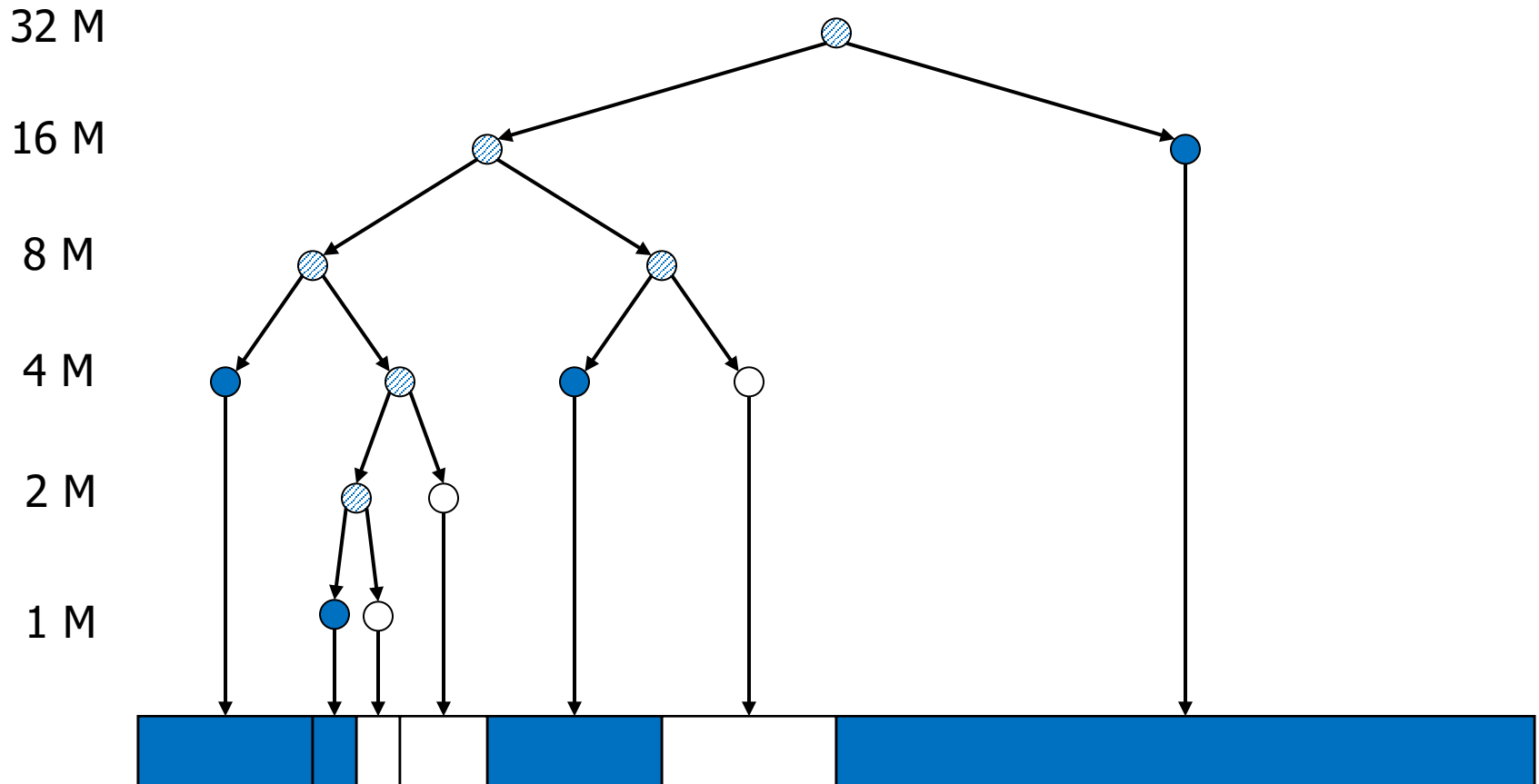
- Memory is separated in $2^{k_{\max}}$ units
- Smaller pieces are created by (continuously) performed bisection of bigger pieces
- Pieces split in one action can be joined by release
- Properties
 - Allocation and release in arbitrary order
 - Allocation of pieces with unit size of $2^0, 2^1, 2^2, \dots, 2^k$
 - Separated representation
 - Limited search costs
 - Internal and external fragmentation
 - Explicit reintegration

Buddy system



Buddy system

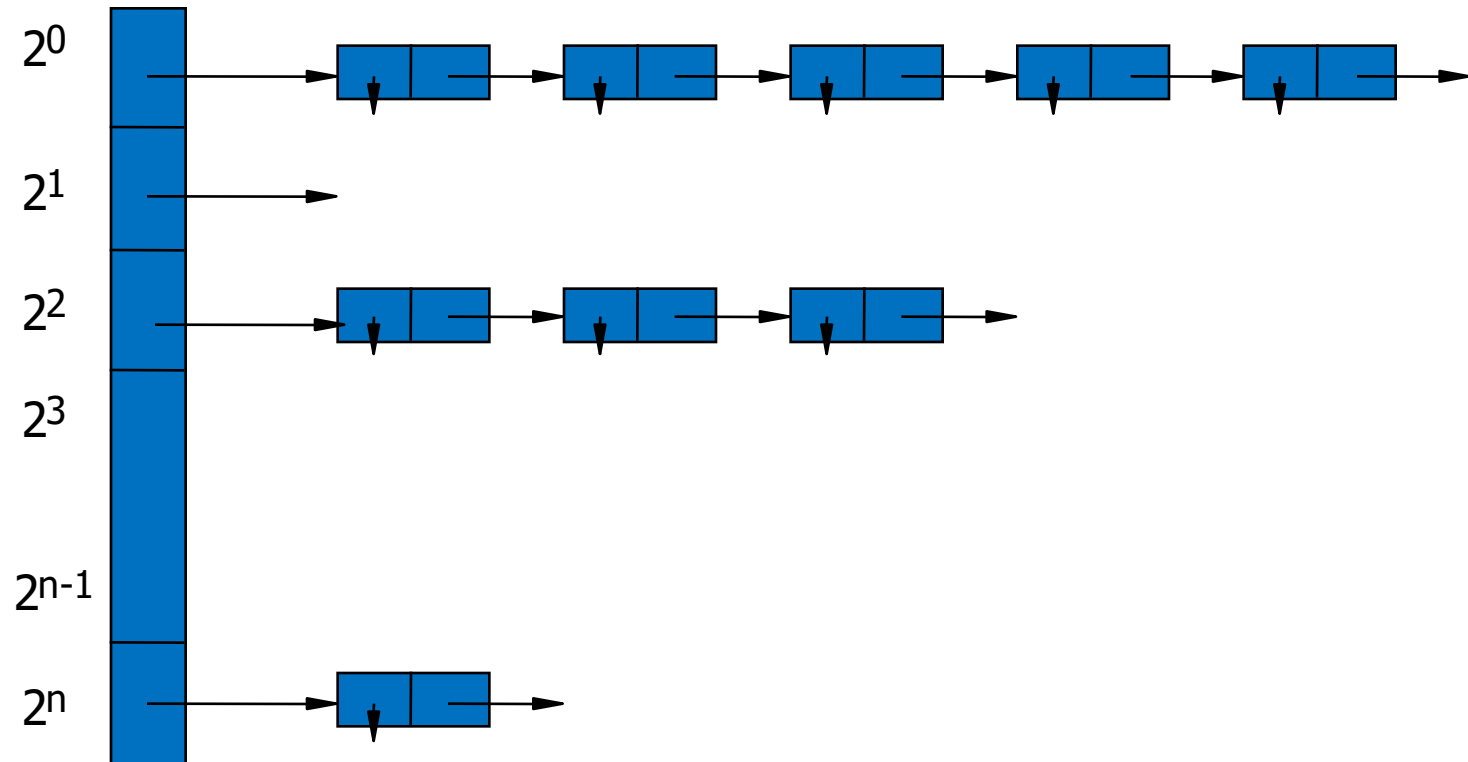
- Representations as tree



Buddies have the same parent node.

Data structures of a Buddy system

- With separated representation



- Array of heads of free lists for pieces with same size

- Handling of requests
 - Check for next value with power of two
 - Take first entry of list
 - In case of empty list (recursive):
 - Take first entry of next list with bigger pieces
 - Cut piece in half
 - Insert second half into list of the original size
 - Take remaining piece to satisfy the request
- Handling release
 - Determine buddy of the piece to be released
 - If buddy is used, insert piece into list
 - In case buddy is free: join both (piece and buddy)
 - Insert emerged piece into the next list

Buddy system – internal fragmentation

- Requests of size a : 1 2 3 4 5 6 7 8 9 10 ...
- Size of allocated pieces $b(a)$: 1 2 4 4 8 8 8 8 16 16 ...
- p_a – probability request is of size a
- $b(a)$ – size of allocated piece for request of size a

- Def.: **Internal fragmentation** ratio between the expected value of the number of unused pieces and the expected value of the number of allocated pieces:

$$\frac{\sum_{a=1}^{a_{\max}} p_a (b(a) - a)}{\sum_{a=1}^{a_{\max}} p_a b(a)}$$

- With $S_b := \sum_{a=1}^{a_{\max}} p_a b(a)$ and $S_a := \sum_{a=1}^{a_{\max}} p_a a$ as the expected values of the size of the allocated piece b or of the size requested respectively the internal fragmentation is $1 - S_a/S_b$.

Buddy system – internal fragmentation

- To determine the internal fragmentation an assumption about the distribution of the requests is needed.
- To simplify matters we assume sizes of request are uniform distributed over the interval $[1, 2^n]$. So every size of request have the same probability $p_a = 2^{-n}$.
- Approximately the average size requested is

$$S_a = \frac{1}{2^n} \sum_{i=1}^{2^n} i = \frac{1}{2^n} \frac{2^n(2^n + 1)}{2} = 2^{n-1} + \frac{1}{2} \approx 2^{n-1}$$

- Keeping in mind the size of the allocated pieces is based on the next value with power of two:

$$\begin{aligned} S_b &= \frac{1}{2^n} \left(1 + 2 + 4 + 4 + 8 + 8 + 8 + 8 + \dots + \underbrace{2^n + \dots + 2^n}_{2^{n-1} \text{ times}} \right) \\ &= \frac{1}{2^n} (1 + 1 \cdot 2 + 2 \cdot 4 + 4 \cdot 8 + \dots + 2^{n-1} 2^n) \\ &= \frac{1}{2^n} \left(1 + 2 \sum_{i=0}^{n-1} 2^{2i} \right) = \frac{1}{2^n} \left(1 + 2 \frac{2^{2n} - 1}{2^2 - 1} \right) \\ &= \frac{1}{2^n} \frac{2^{2n+1} + 1}{3} \approx \frac{2^{n+1}}{3} \end{aligned}$$

- Therefore the ratio $S_a / S_b \approx 3 \cdot 2^{n-1} / 2^{n+1} = 3/4$
so the allocated pieces are used by $3/4$ and the internal fragmentation is 25%.

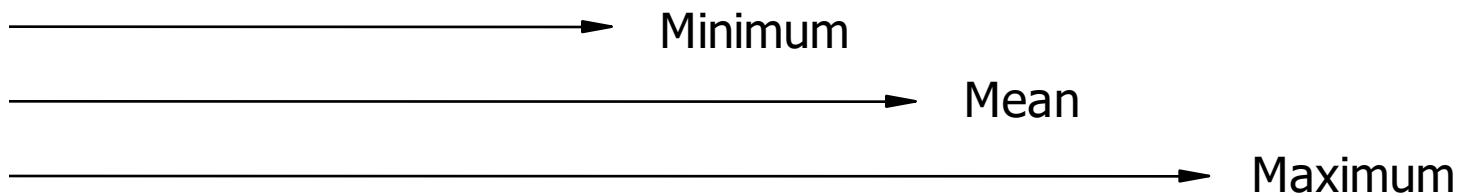
Buddy system

- Fast operation with $O(1)$
- Adaption to distribution of requests
- Only limited number of split and join operations after transient oscillation.
- Amount of internal fragmentation fairly large.



25% int. fragmentation

Requests with uniform distribution:



7.2 Address Translation

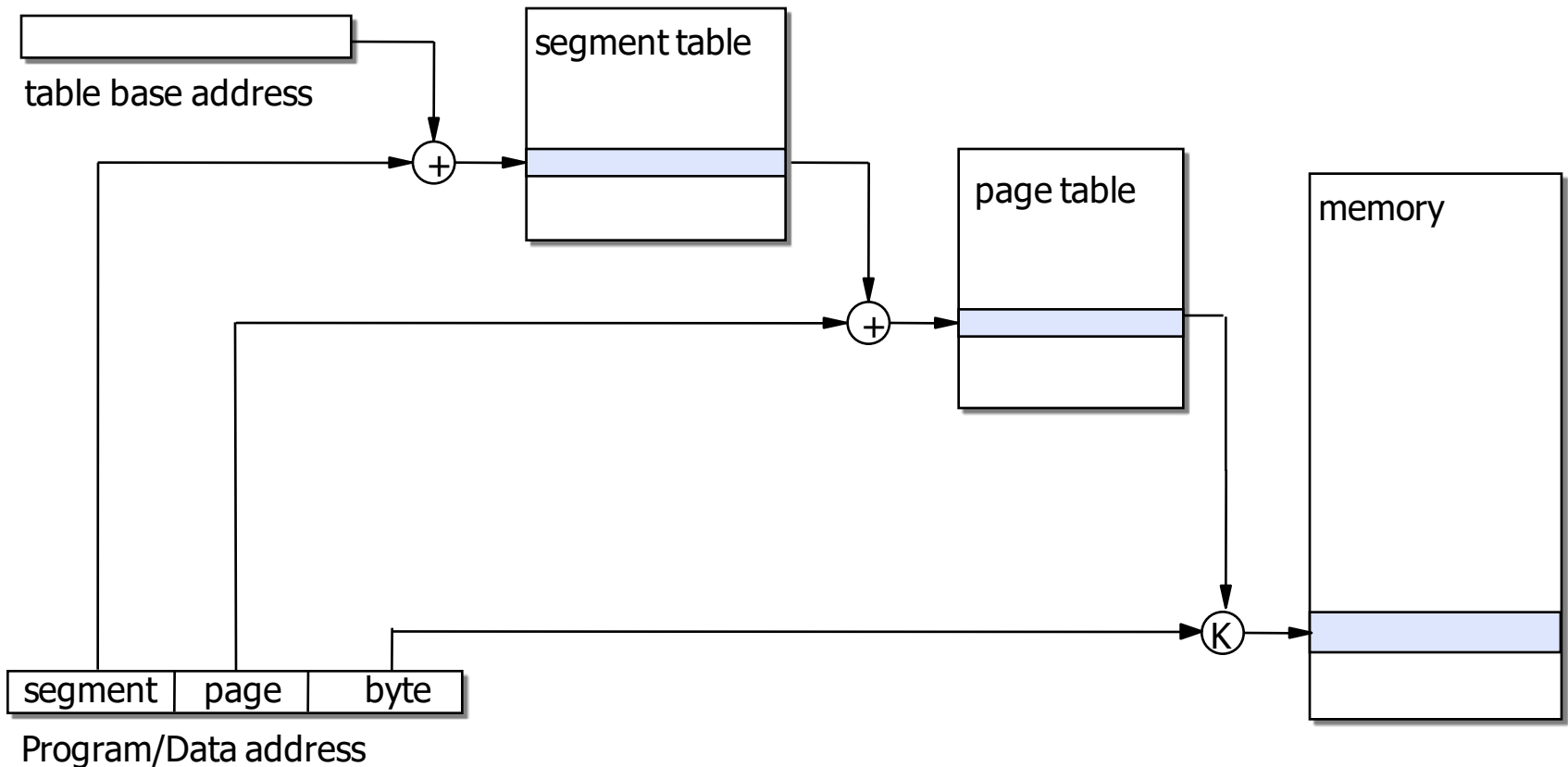
- An address space is a contiguous set of addresses.
- It holds all necessary instructions and data structures needed to execute a program.
- Parts of the address space may be undefined. Access to undefined parts of the address space leads to an error.
- We distinguish:
 - Logical address space, program address space (from the view of the thread/program)
 - Physical address space (defined by the width of the address bus)
- For higher efficiency and security, logical address spaces are decomposed into segments (of different size) which in turn are cut into pages (equal size)

Address Spaces: Examples

- Address spaces of, e.g., 64 bit machines are not always as expected:
- Linux: `cat /proc/cpuinfo`
Here: only small snippets from some example machines
- Intel, mobile CPU, 2007
model name : Intel(R) Core(TM)2 Duo CPU L7700 @ 1.80GHz
address sizes : 36 bits physical, 48 bits virtual
- Intel, desktop CPU, 2011
model name : Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
address sizes : 36 bits physical, 48 bits virtual
- Intel, entry server CPU, 2009
model name : Intel(R) Xeon(R) CPU X3470 @ 2.93GHz
address sizes : 36 bits physical, 48 bits virtual
- Intel, server CPU, 2009
model name : Intel(R) Xeon(R) CPU X5570 @ 2.93GHz
address sizes : 40 bits physical, 48 bits virtual
- AMD, desktop CPU, 2008
model name : AMD Athlon(tm) 64 X2 Dual Core Processor 5600+
address sizes : 40 bits physical, 48 bits virtual
- AMD, desktop CPU, 2011
model name : AMD FX(tm)-6100 Six-Core Processor
address sizes : 48 bits physical, 48 bits virtual
- AMD, server CPU, 2009
model name : Six-Core AMD Opteron(tm) Processor 8435
address sizes : 48 bits physical, 48 bits virtual

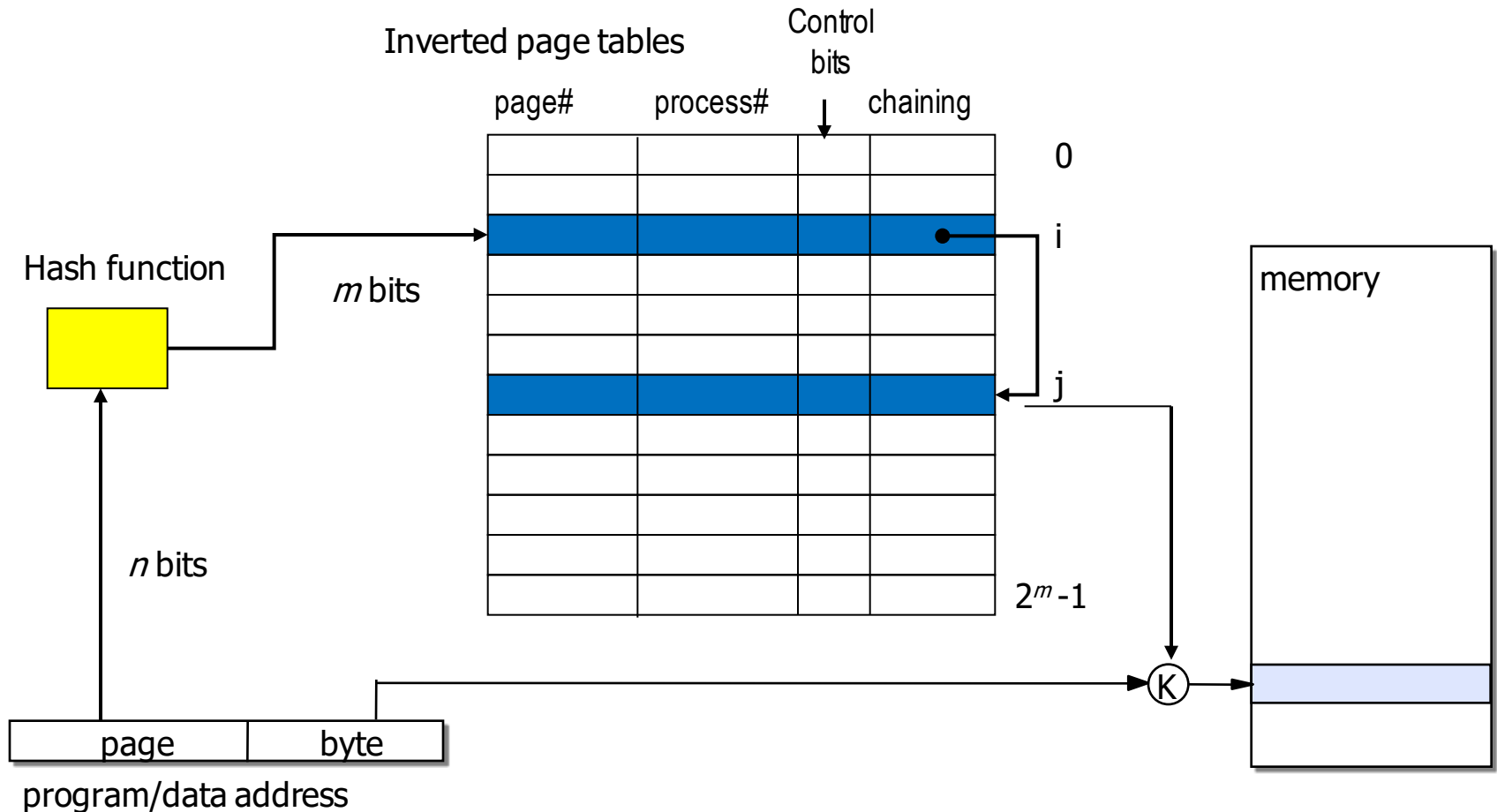
Two-stage hierarchical address translation

- Each segment consists of a variable number of pages.



Inverted page table

- While the Intel I32 processors or ARM processors support multistage segment / page tables, PowerPC and UltraSPARC-processors use **inverted page tables**.



Page Table: Theoretical Example

- 32 Bit addresses
- 4 GB logical address space
- 64 MB RAM (physical)
- Pages of 1 KB
- One page table for the whole logical address space

address (32 Bit)	
page address (22 Bit)	offset in page (10 Bit)

- Offset (inside pages): 10 Bit ($2^{10} = 1 \text{ KB}$)
- Page addresses: 22 Bit ($32-10$)
- Number of entries in page table: $2^{22} = 4\text{M}$
- Size of an entry: 16 Bit = 2 Byte
($64 \text{ MB} = 2^{26} \text{ B} = 2^{16} \text{ frames}$)
- Size of page table (ignoring management information such as dirty bits etc. and ignoring alignment): 8 MB ($4\text{M} \times 2 \text{ Byte}$)

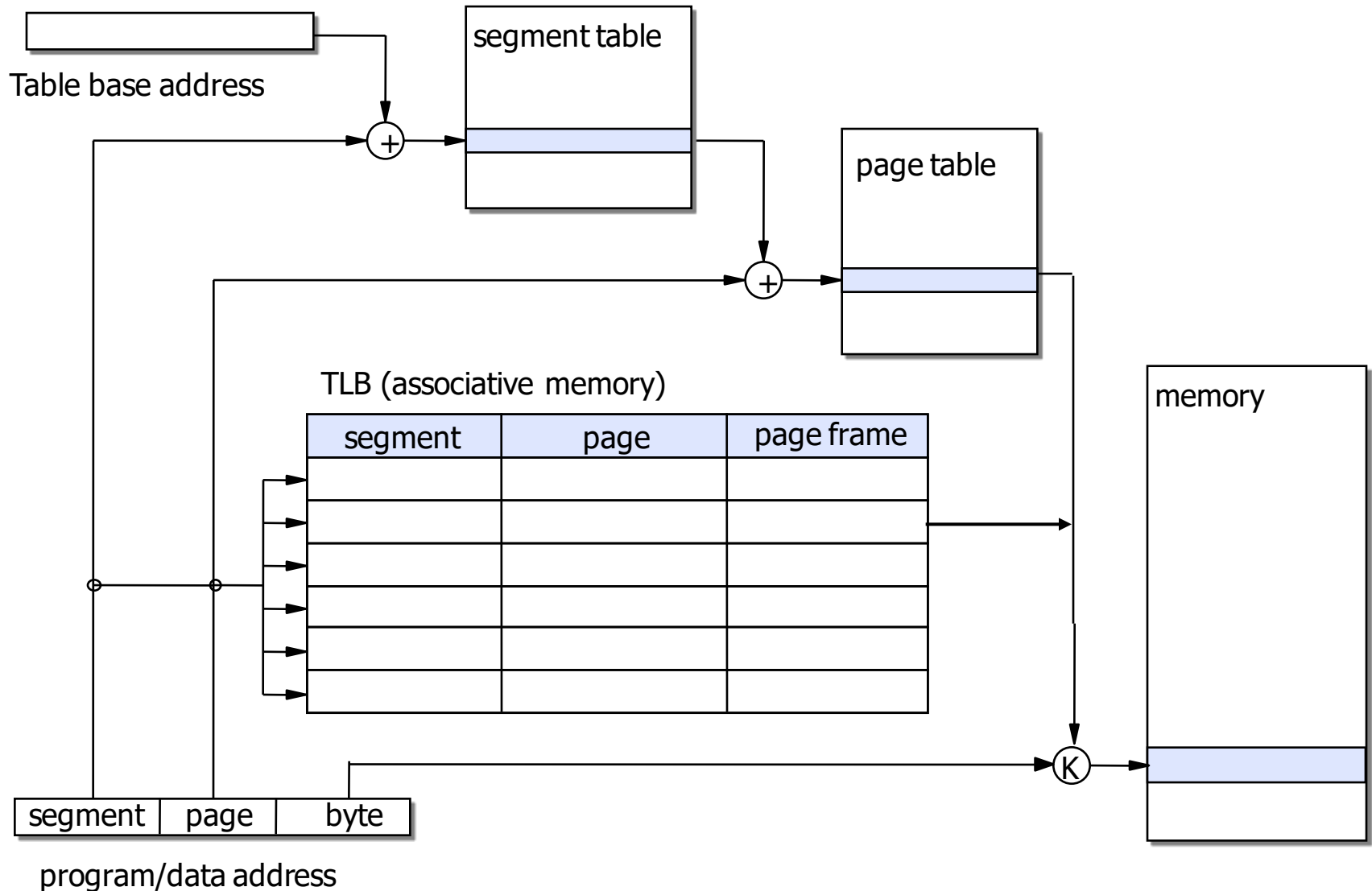
Inverted Page Table: Theoretical Example

- 32 Bit addresses
- 4 GB logical address space
- 64 MB RAM (physical)
- Pages of 1 KB
- One page table for the whole logical address space
- Offset (inside pages): 10 Bit ($2^{10} = 1 \text{ KB}$)
- Number of frames: $65536 = 2^{16}$
- Frame addresses: 16 Bit
- Number of entries in inverted page table: $2^{16} = 64\text{K}$
- Size of an entry: 22 Bit = 2.75 Byte
(page addresses are 22 bit (32-10))
- Size of page table (ignoring management information such as dirty bits etc. and ignoring alignment): 176 KB ($64\text{K} \times 2.75 \text{ Byte}$)
- But: Search is much more complicated (e.g. hash function)!

Problem:

- Segment and page tables are so large that they have to be kept in main memory.
- To build an effective main memory address, we first need to get the page and/or segment address.
- For each address (instruction or data) we need at least two accesses to main memory.
- Thus, the processing speed is reduced by a factor of 2.
- To prevent that, the currently used parts of the segment/page tables are stored in a fast set of registers. (TLB = Translation Lookaside Buffer, part of MMU)
- The TLB is an associative memory, i.e. a table in which the entry to be found is being searched simultaneously in all lines of the table.
- It is used as a sort of cache for page/segment tables.
- Usually, the search can be performed in one processor cycle.

Two stage hierarchical address translation with associative register

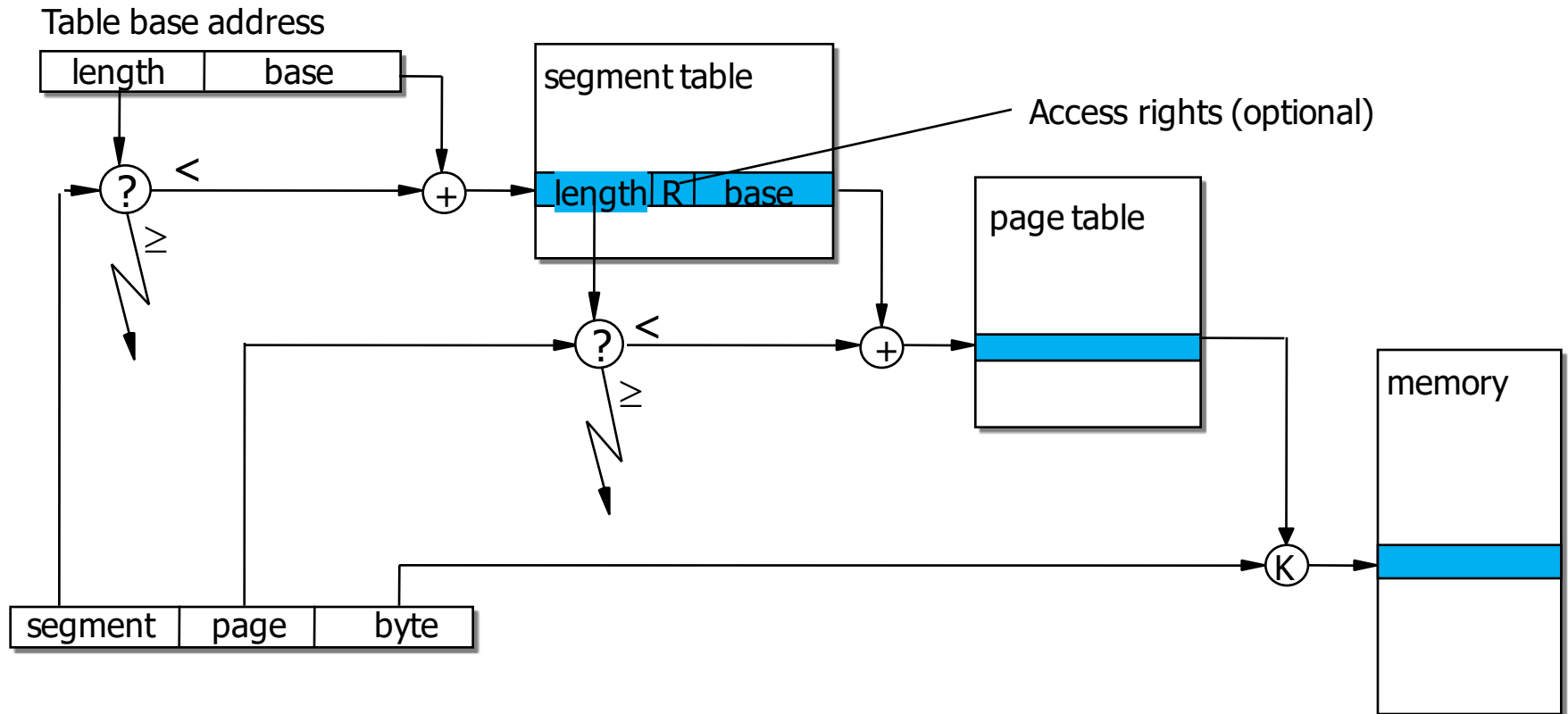


Typical properties of a TLB

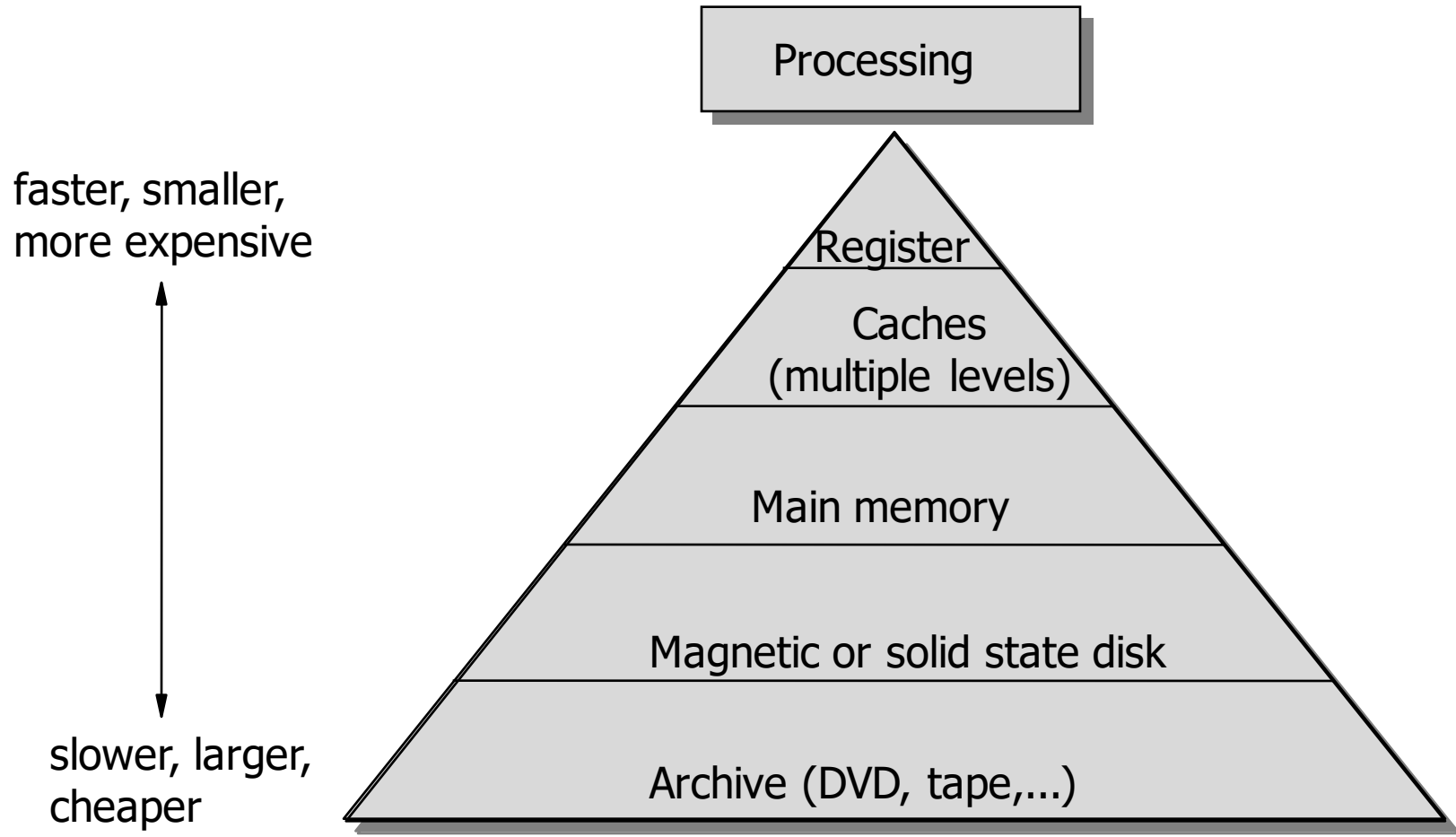
- Line width : 4-8 bytes: Logical page/segment-no., page frame no., Management bits
- Time for address translation:
 - hit: ≤ 1 processor cycle
 - miss: 10 - 200 processor cycles (depending on memory speed)
- Hit rate: 99.0% - 99.99%
- TLB-size: 32 – 1024 lines (entries)

Memory protection for hierarchical address translation

- Table base register and segment table entries are complemented by a length field indicating the appropriate amount of memory.
- Exceeding the length triggers an interrupt (segmentation fault).
- It is possible to differentiate between read and write access and/or different processor modes.

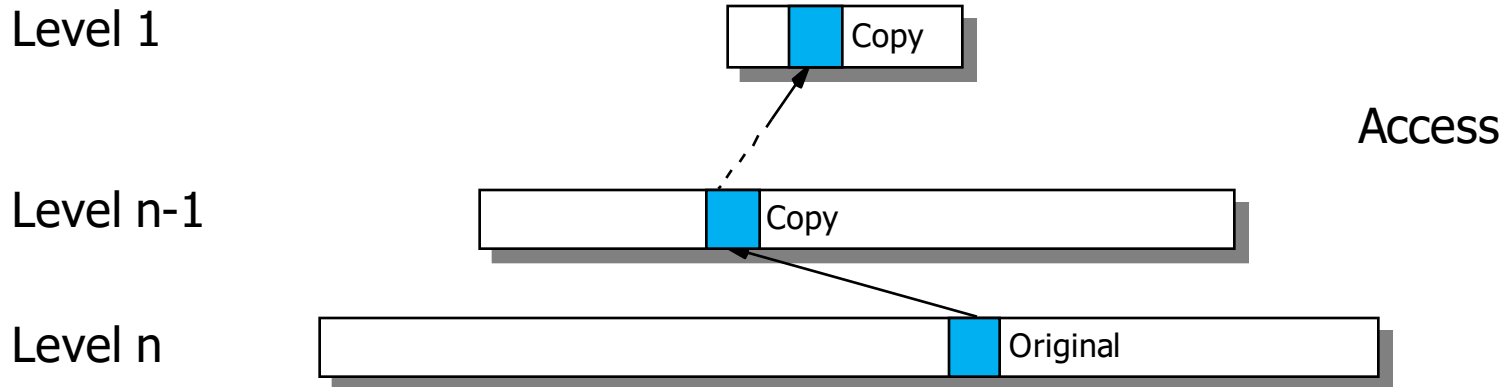


7.3 Memory Hierarchy and Locality

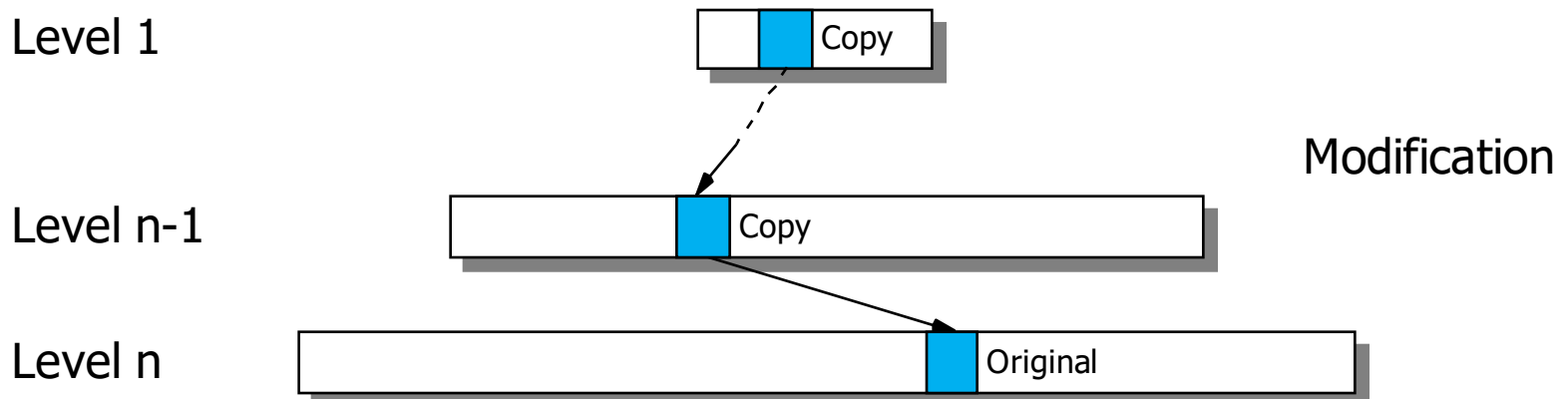


Operation of Memory Hierarchy

- Copies of the data object will be generated at time of (first) access, so it seems that the data object is “climbing up”.



- After modification of the data object changes will be propagated (step-by-step, delayed) downwards.



The memory hierarchy is based on the

Principle of locality

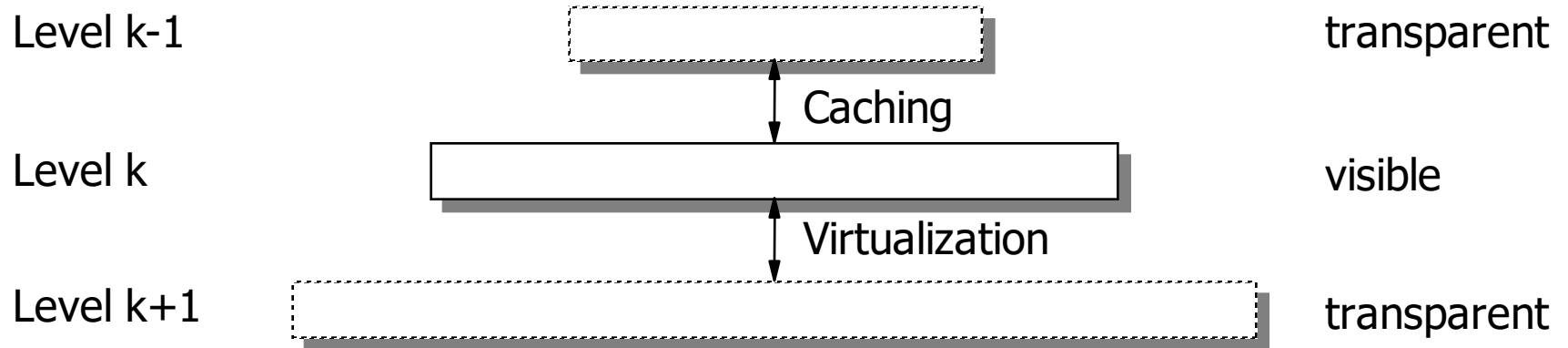
- A program limits its accesses within a small time interval Δt to only a small subset of its address space A .
 - Spatial locality: when a program accesses an address a , then another access to a nearby address is very likely.
 - Temporal locality: When a program accesses an address a , then a repeated access to the same address within short time is very likely.

Why ?

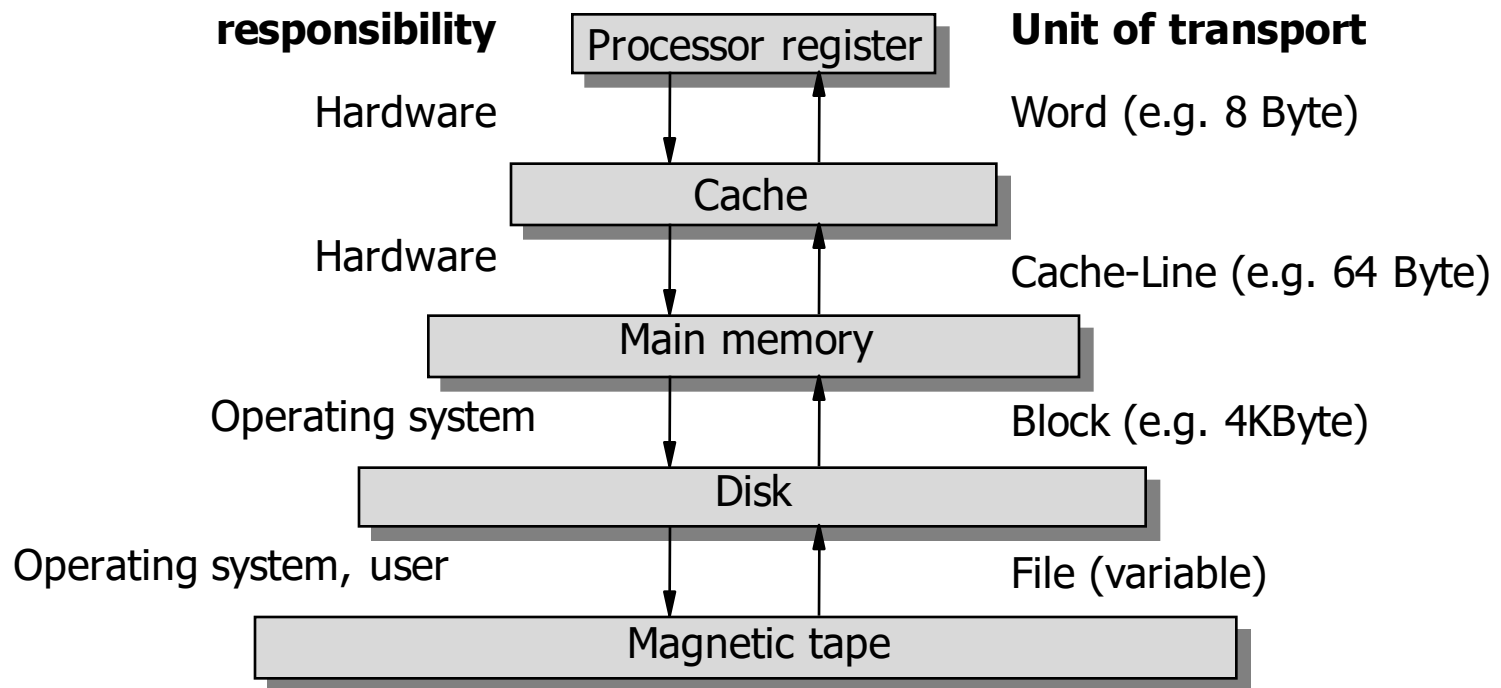
- Mostly, instructions are executed sequentially.
- Programs spend the most time in loops.
- Some parts of the program are executed only in exceptional cases.
- Many arrays are only partially filled.
- 90/10-Rule: A thread spends 90% of its time in 10% of its address space.

- Goal
 - Hold data needed on highest level as possible.
- Problem
 - Capacity is shrinking on the way up.
- Questions
 - How it can be known what data object is accessed next?
Knowledge about the program behavior
 - Who is responsible for data transport between levels?
User/Programmer, Compiler, OS, Hardware
 - What is the size of the data objects feasible for transportation?
Bytes, Words, Blocks, Files
 - Is there an automatic mechanism for transportation between levels?
 - Is there an acceleration of the data access (Caching) or enlargement of capacity (Virtualization)?

Caching vs. Virtualization



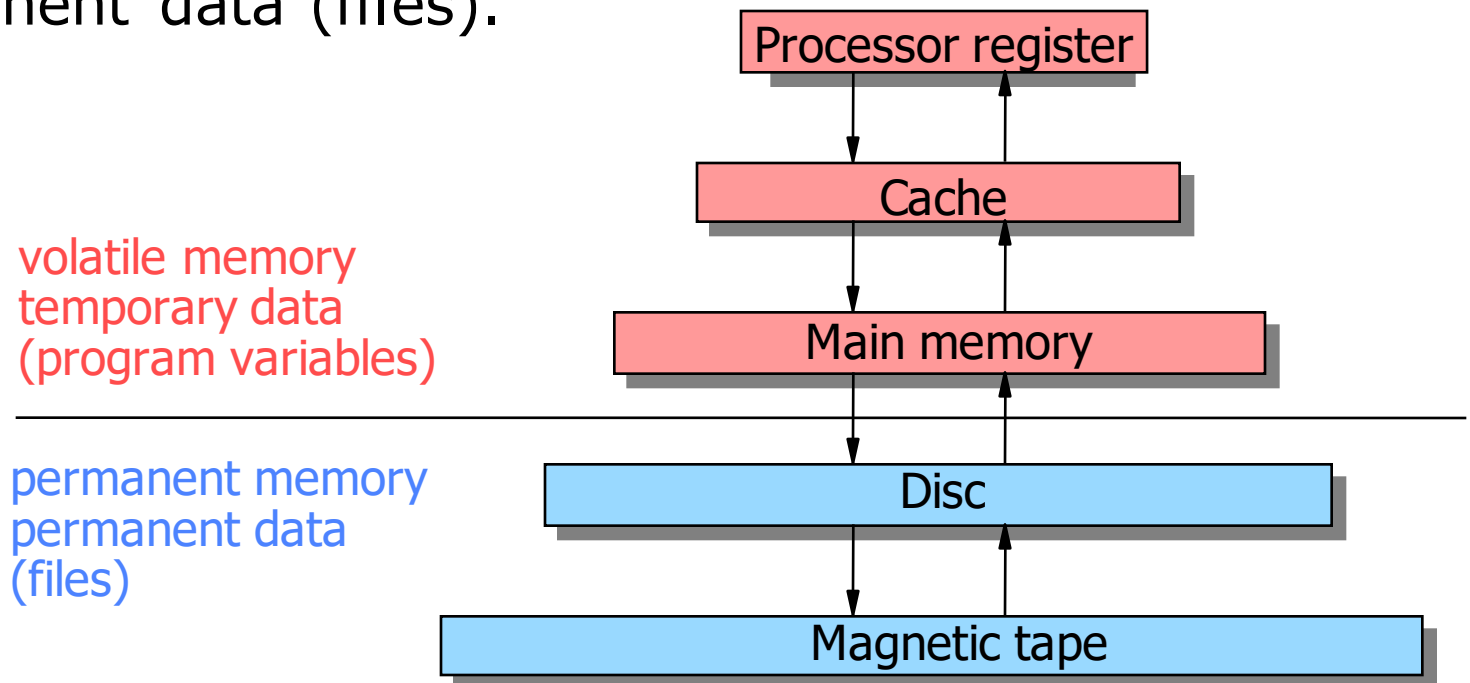
- Usually not all of the levels are recognizable for the programmer or user – some are hidden or transparent.
- So the user has the impression to access Level k only.
- In case of Caching the access is performed on Level k-1 and Level k is visible.
- In case of Virtualization the access is performed on Level k+1, but the user has the impression to access Level k.
- Caching is used to accelerate the data access, Virtualization is used to enlarge the capacity.



- During the runtime of the program the transport of data and instructions between main memory, cache, and processor is done by the hardware (transparent to software).
- Accesses to the disk are performed by the operating system.
- Writing files to and reading from archive memory can be done either explicitly by the user or automatically by the operating system (file system).

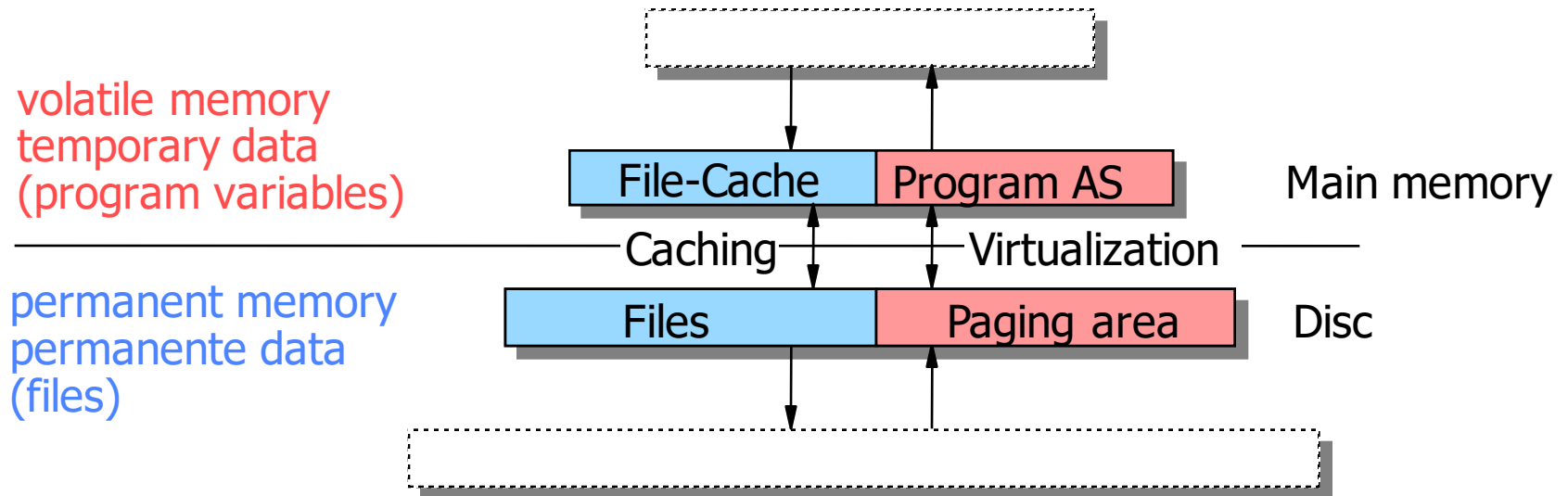
Volatile vs. permanent Memory

- Due to the used media the memory on higher levels usually is implemented as volatile memory. So the data stored within this memory is lost after power cutoff.
- Therefore higher levels are used to hold temporary data (program variables), while the other levels hold permanent data (files).



Volatile vs. permanent Memory

- Using Caching and Virtualization led to weaken the difference between Main memory (for address spaces only) and Disc space (for files only).



7.4 Virtual Memory

- Due to the principle of locality, only those parts of the address space that is currently in use by the program, needs to be present in the physical memory.
- The pages needed are loaded only when addressed (*demand paging*).
- The copy-out and copy-in operations of the pages can be automated (by some hardware support).
- For the user / programmer all these activities are transparent.
- Programmer has the impression that main memory is available in (almost) unlimited size.
- But this unlimited memory is only *virtually* existent.

Requirements for efficient operation:

- Noncontiguous allocation (page tables)
 - Pages are the units of transfer.
- Automatic detection of missing pages
 - Access to missing page triggers interrupt.
 - Loading of page from disk is initiated as part of the interrupt handling.

Involved components (Data structures)

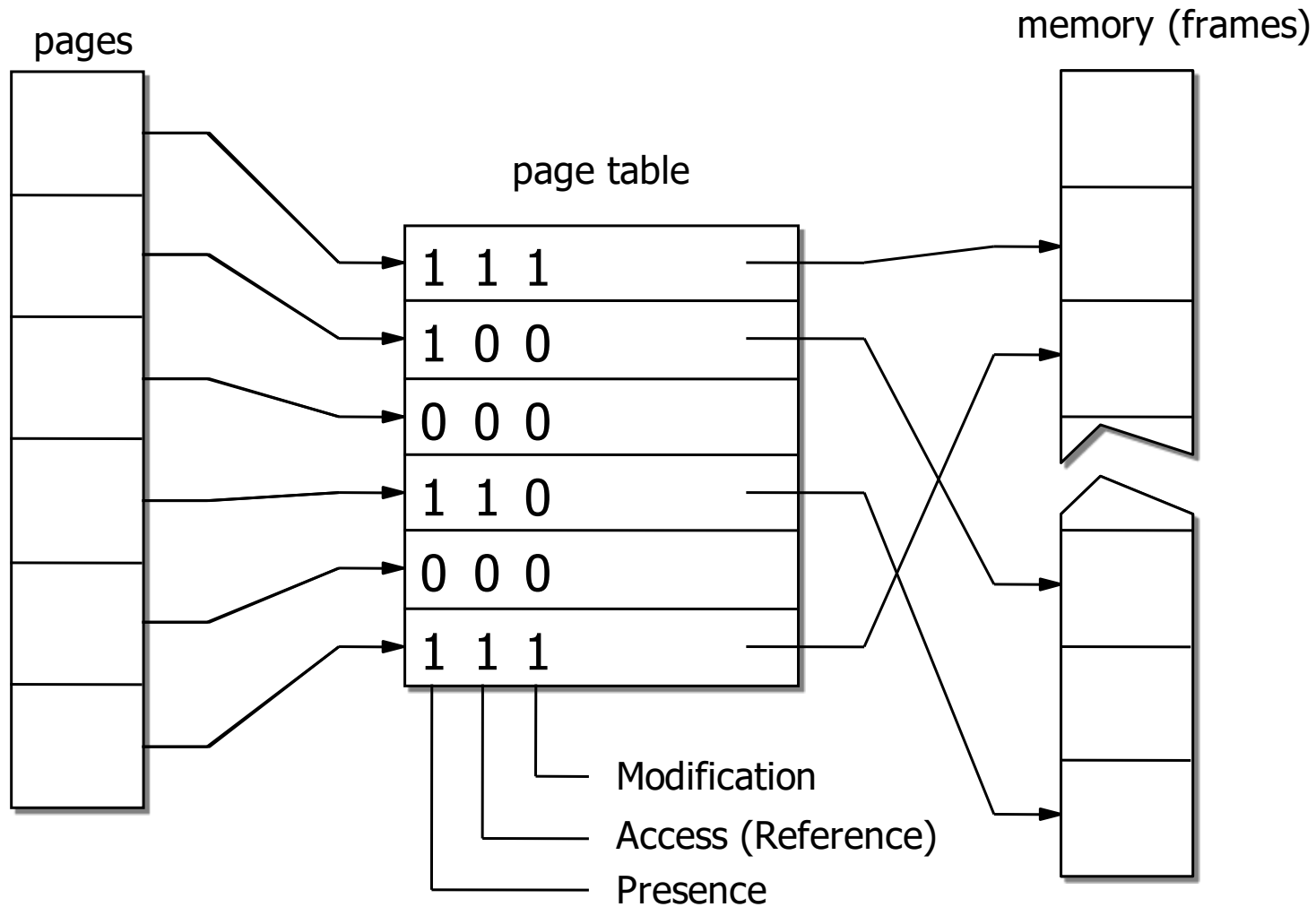
- Page table
 - Function: address transformation
 - Content: for each page
 - usage and presence information
 - physical address (page frame number)
- Page frame table, inverted page table
 - Function: memory management
 - Content: for each page frame
 - state (free / occupied)
 - owner
 - occupying page
- Swap area (paging area)
 - Function: areas of storage to store the pages that are swapped out
 - Usually mass storage such as magnetic or solid state disks
 - Seldom network devices

Page table for virtual memory

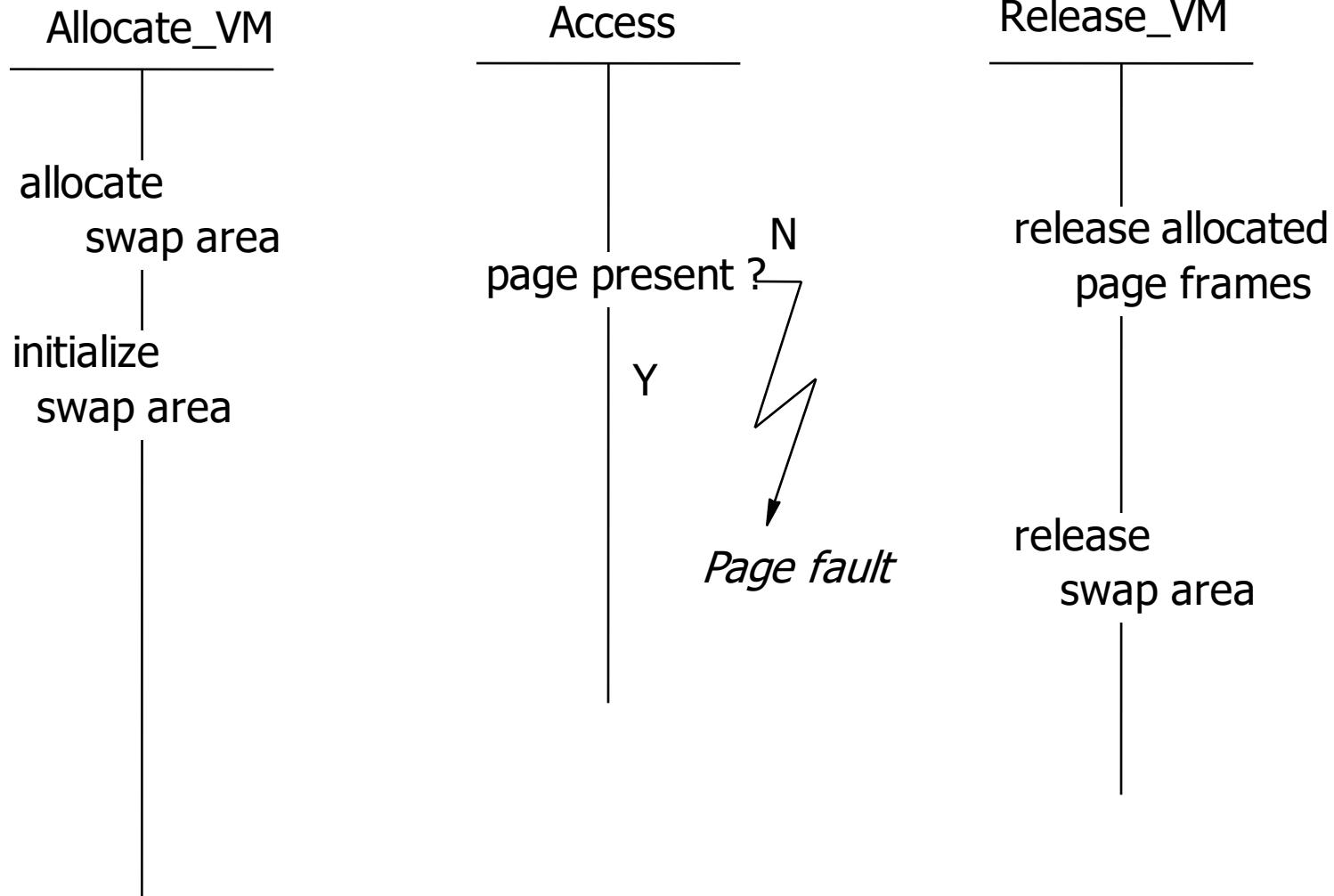
In addition to the physical address, each entry provides information whether

- the page is present in main memory:
 - presence bit, valid bit
- the page has been accessed:
 - reference bit
- the page has been modified (write access):
 - modification bit, dirty bit

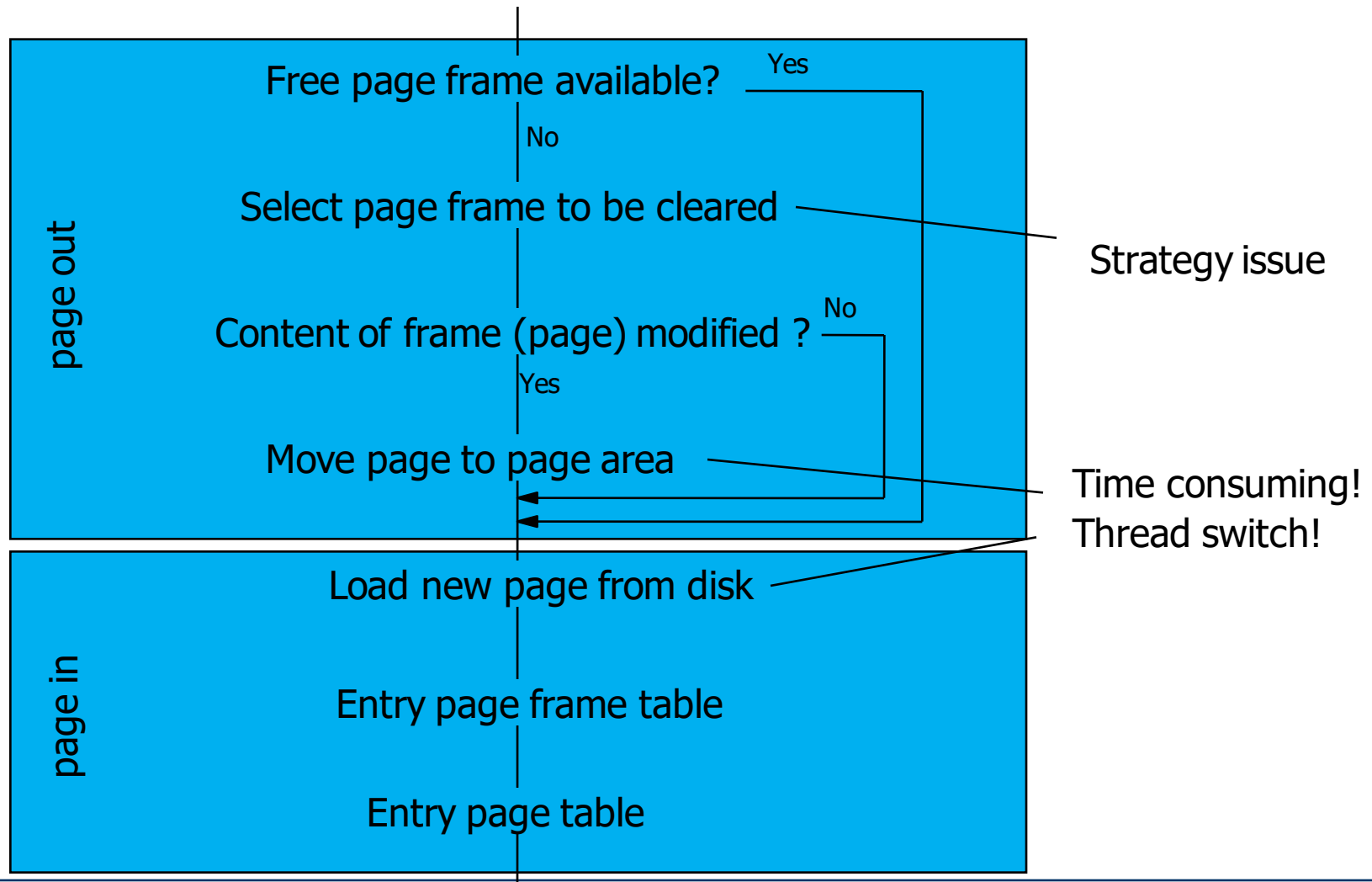
Page table for virtual memory



Tasks for virtual memory management

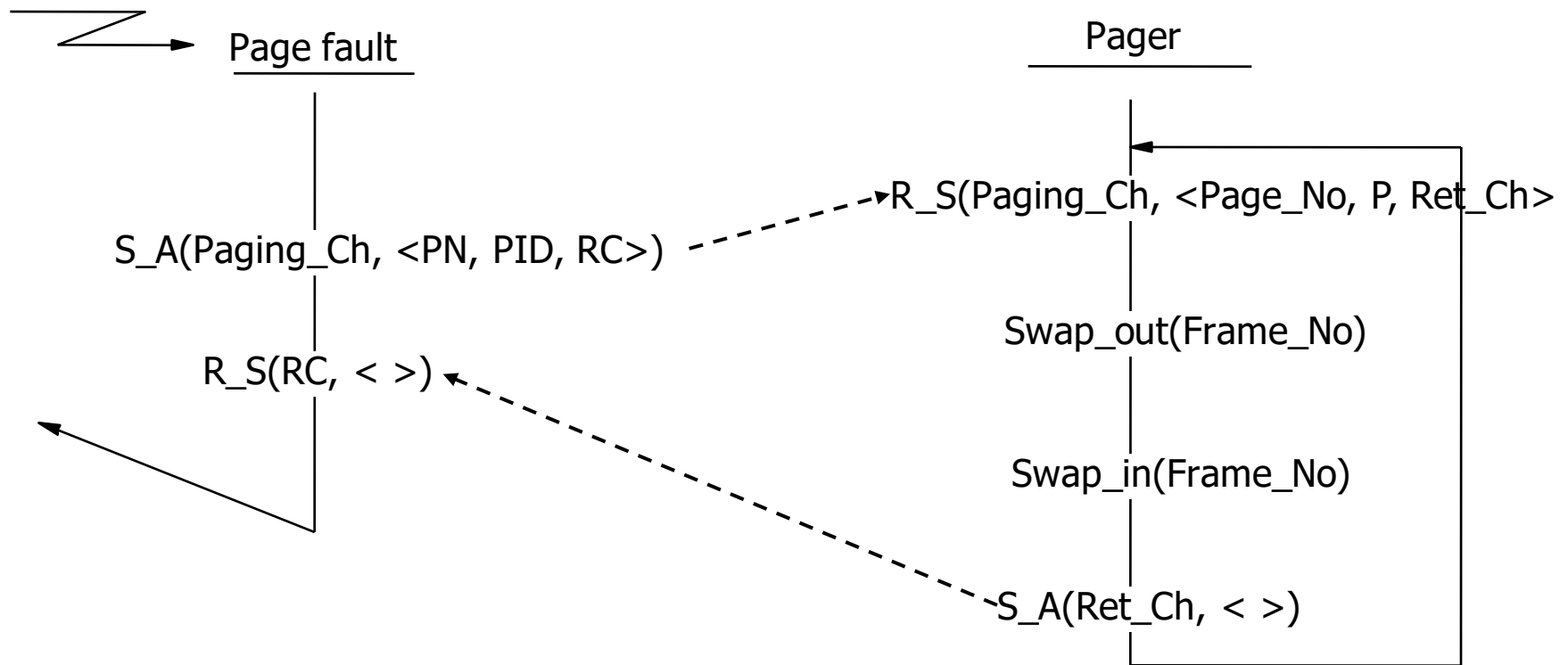


Page fault

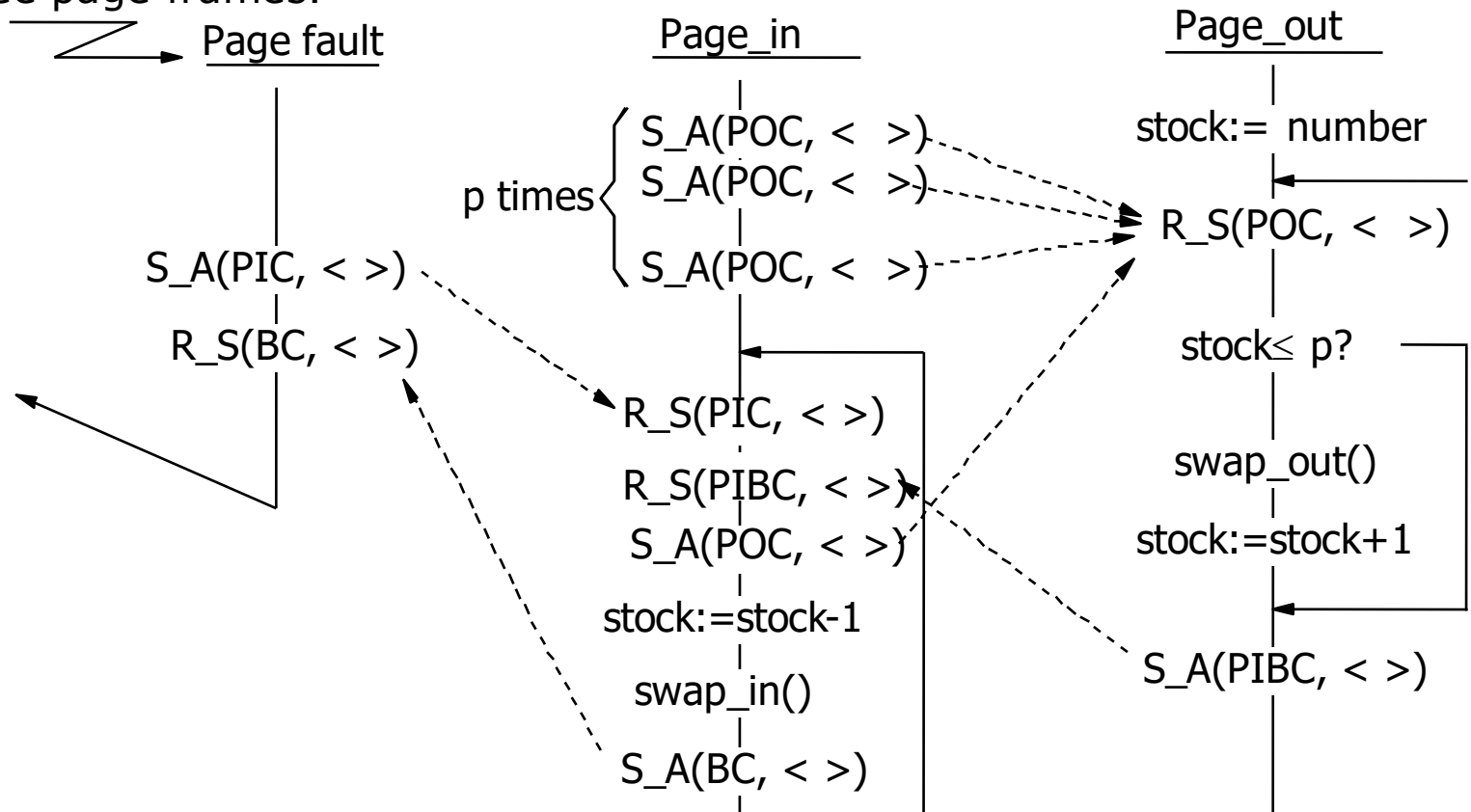


Parallelization of paging

- Paging is a time-critical component, we therefore try to speed it up by parallelization.



- Since page faults often occur in bulks, it is recommend to have some amount of free page frames available to avoid costly page-out operations when time is tight.
- To that purpose we parallelize by applying the buffering principle to get a stock of free page frames.



7.5 Page replacement strategies

A small calculation:

- Let p_{pf} be the probability of a page fault, t_m the memory access time and t_{pf} the time needed for handling a page fault.
- Then we obtain as effective memory access time t_{eff} in the virtual memory:

$$t_{eff} := (1 - p_{pf}) \cdot t_m + p_{pf} \cdot t_{pf}$$

- Using roughly realistic numbers, e.g. $t_m = 20$ nsec and $t_{pf} = 20$ msec

$$\begin{aligned} t_{eff} &= (1 - p_{pf}) \cdot 20 + p_{pf} \cdot 20.000.000 \\ &= 20 + 19.999.980 \cdot p_{pf} \end{aligned}$$

- With a page fault probability of $p_{pf} = 0.001$, we get an effective access time of 20 μ sec, i.e. a slow-down by a factor of 1000!
- Even with a value of $p_{pf} = 10^{-6}$ the effective access time doubles.
- Thus it is of utmost importance to keep the number of page faults extremely low.

- The page fault rate strongly depends on which pages are kept in real memory and which are stored on disk.
- Selection strategy:
When a page fault occurs and no page frame is free, which page frame should be emptied?
- Differentiation
 - Local selection strategy:
We clear a page frame of that process that caused the page fault.
 - Global selection strategy:
An arbitrary page frame (belonging to other processes) is cleared.

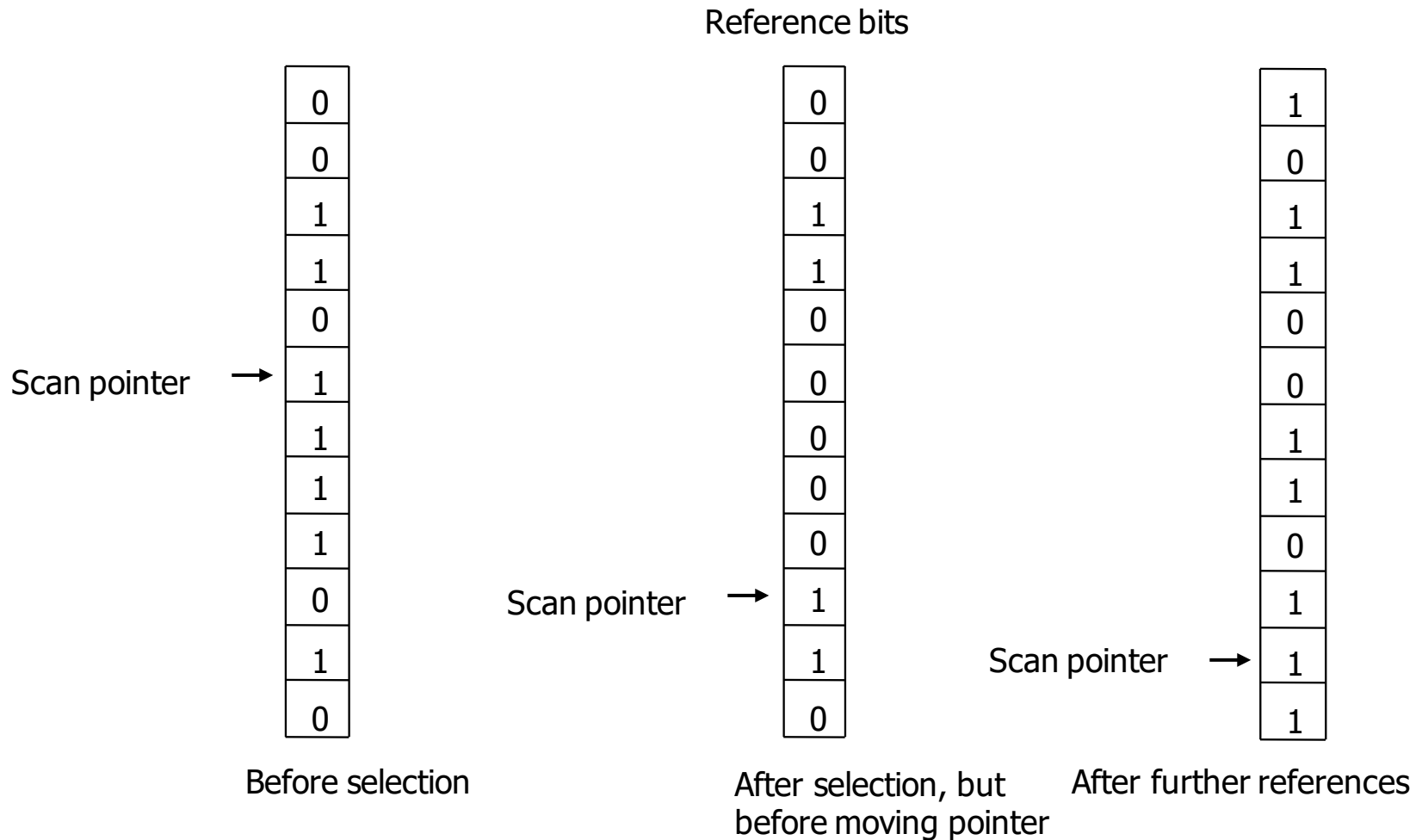
- **FIFO** (First-In-First-Out)
The page that is longest in memory is swapped out.
- **LFU** (Least Frequently Used)
The page that has been least frequently referenced is swapped out.
- **LRU** (Least Recently Used)
The page not been referenced for the longest period is swapped out.
- **RNU** (Recently Not Used)
The page not been referenced within some specified time period is swapped out.

Second-Chance-Algorithm (Clock-Algorithm)

The Clock- Algorithm is smarter, since it resets the reference bits not all at once but only smaller subsets:

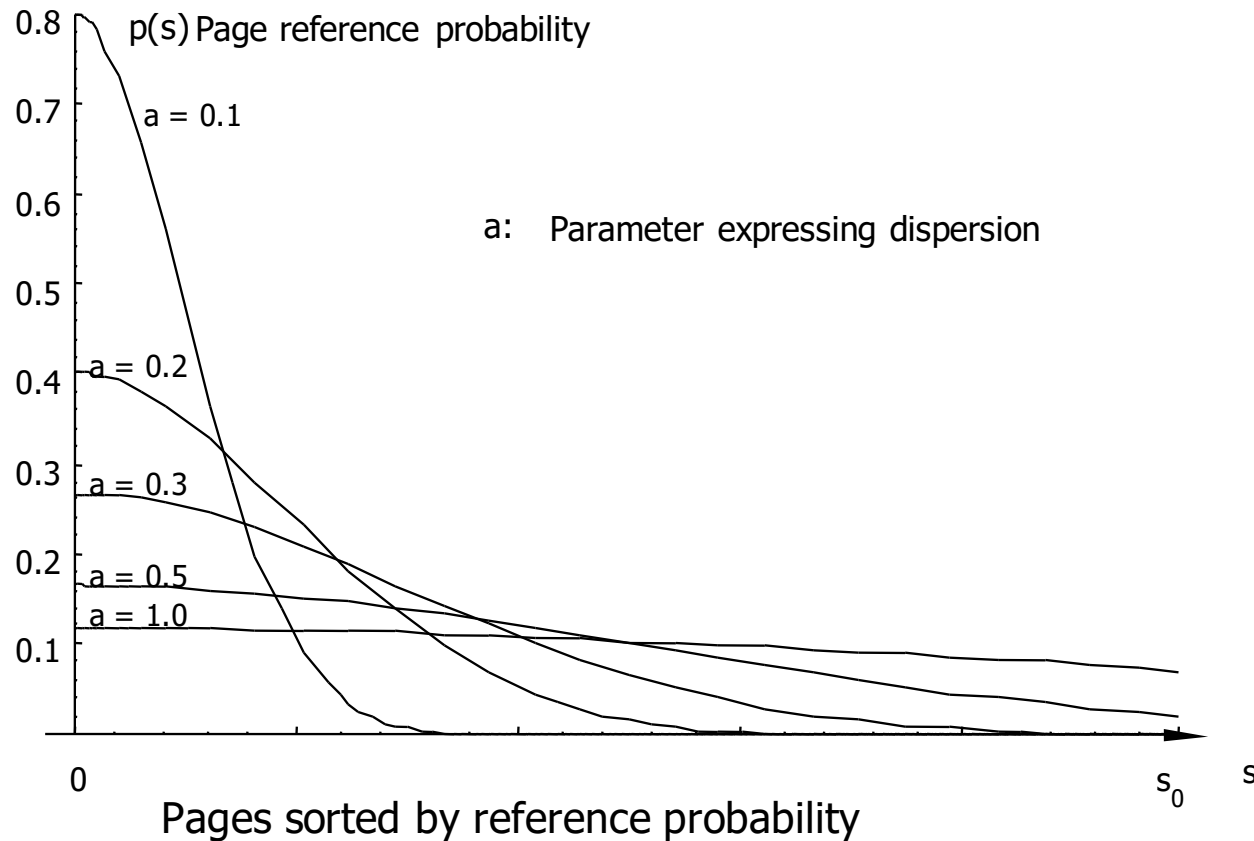
- The vector of reference bits is scanned cyclically.
- For searching the next candidate for swap out, the next page is selected which has a reference bit of 0.
- During this linear search all visited reference bits are reset to 0.
- They have – until the scan pointer revisits the page again during the next cycle – a second chance to be referenced and stay in memory.
- That means that the selected page has the property that it has not been referenced during the last scan cycle.

Second-Chance-Algorithm



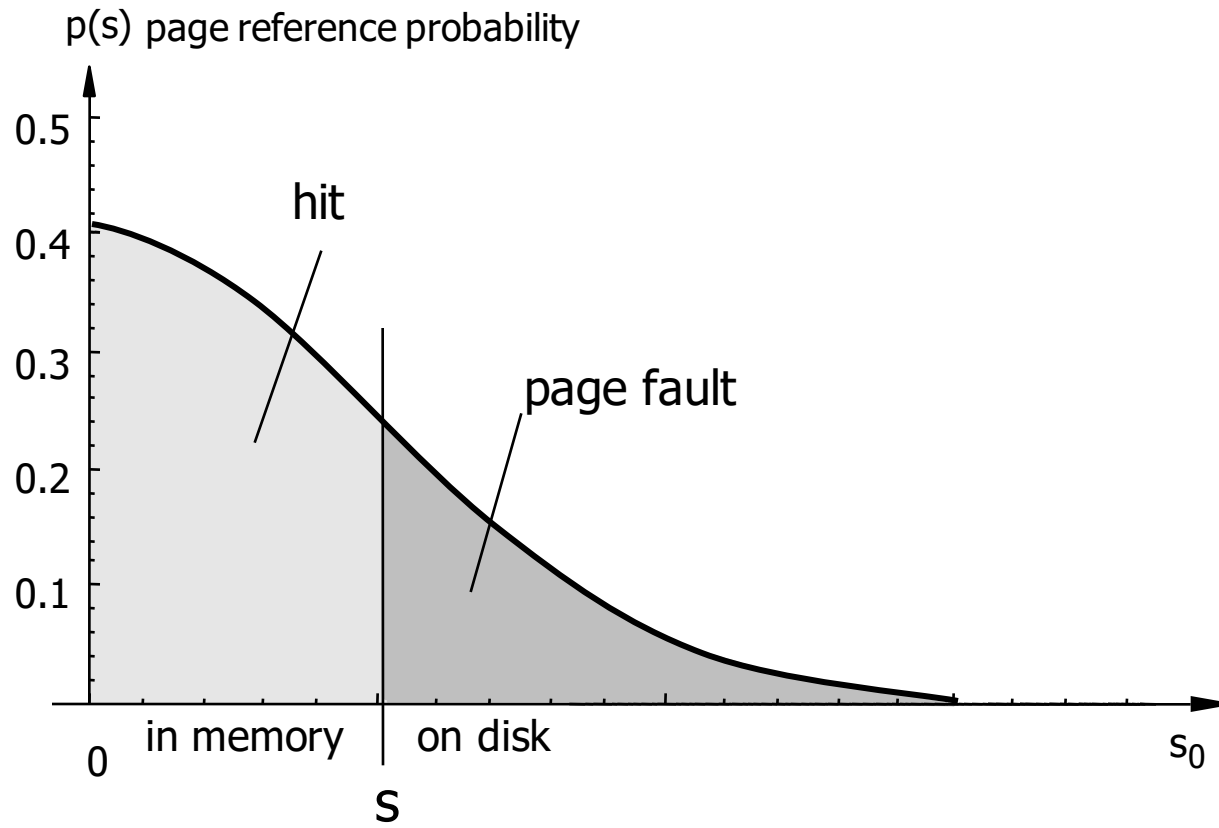
7.6 Performance aspects of virtual memory

- The virtual memory works the better, the higher the programs' locality.
- Locality is good, if few pages are referenced with high probability, and many pages with low probability (small a).



7.6.1 Modeling paging

- In memory should only be those pages that are referenced with high probability.



Pages sorted by reference probability

Modeling paging

- Let be
 - s number of available frames
 - s_0 size of address space
- The s most frequently referenced pages are assumed to be in memory (i.e., in the s available frames).

Then we have:

- Hit probability

$$p_{hit}(s) = \int_0^s p(z) dz$$

- Page fault probability

$$p_{pf}(s) = 1 - p_{hit}(s)$$

Normalized to size of address space:

- Memory offer

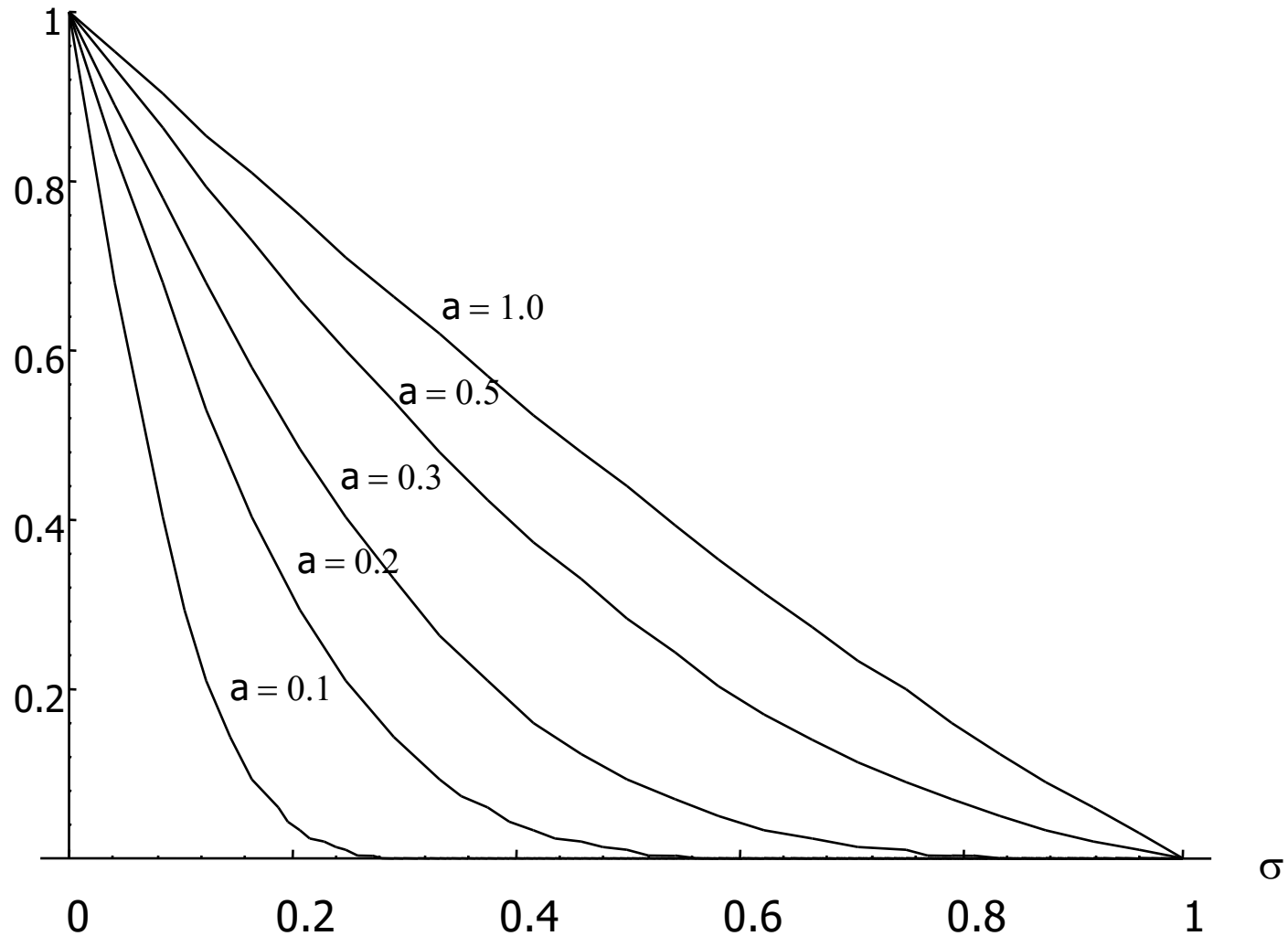
$$\sigma = s/s_0$$

- Normalized page fault probability

$$p'_{pf}(\sigma) = p_{pf}(s)$$

Dependence of page fault probability on memory offer

$p'_{pf}(\sigma)$ page fault probability

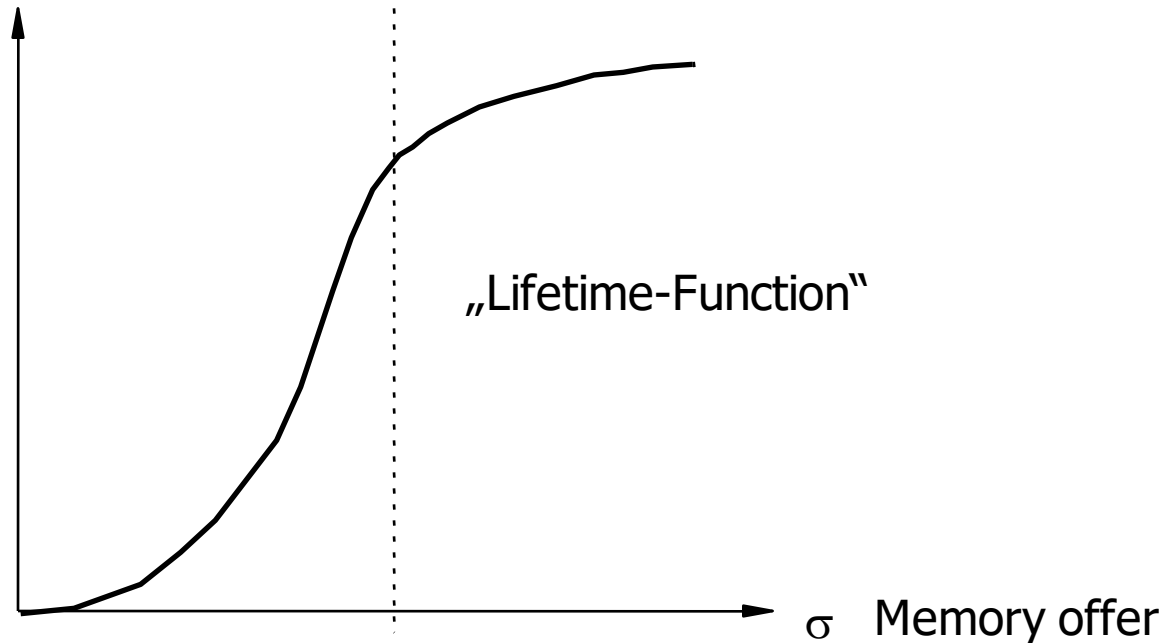


- K number of memory frames
- n number of programs in memory
(*Multiprogramming Level, MPL*)
- s_0 size of program address space (in pages)
- t_s time between two page faults
- t_T page transfer time
- s memory offer in pages
- $\sigma = s/s_0$ memory offer normalized to program address space

- For n identical programs the following holds:

$$s(n) = K / n \quad \text{or} \quad \sigma(n) = K / (n s_0), \text{ resp.}$$

t_s time between two page faults



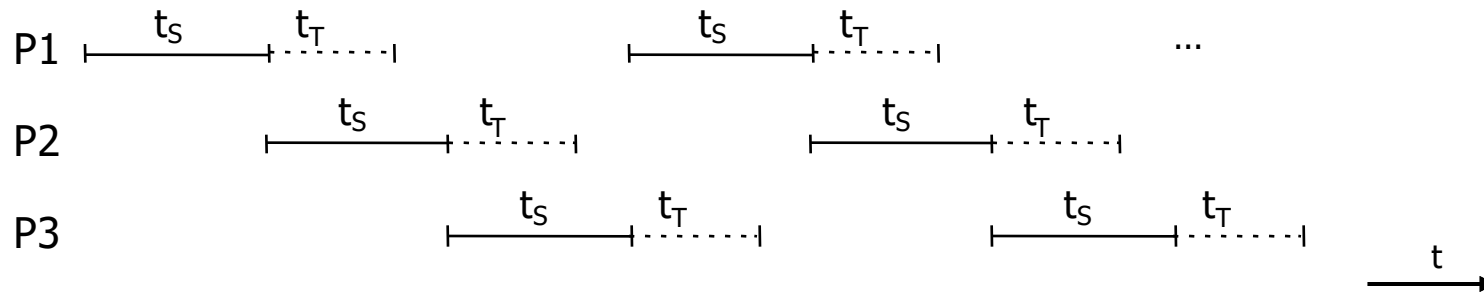
- The time between two page faults depends on the amount of available memory
- Left of the „knee“ the function can be approximated by a parabola:

$$t_s \approx a \cdot \sigma^2 = a \cdot \left(\frac{K}{n \cdot s_0} \right)^2$$

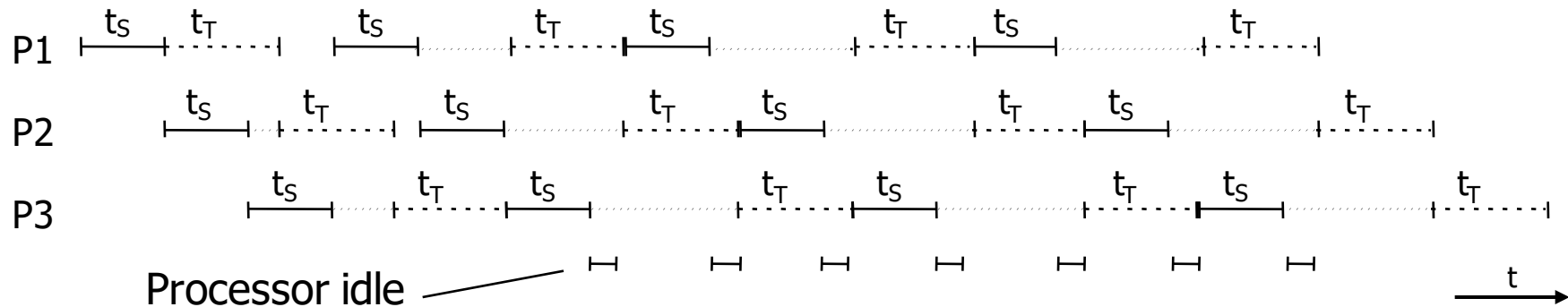
i.e. t_s decreases with growing n

Interleaving of compute and page transfer phases

Case: $t_T < t_S$



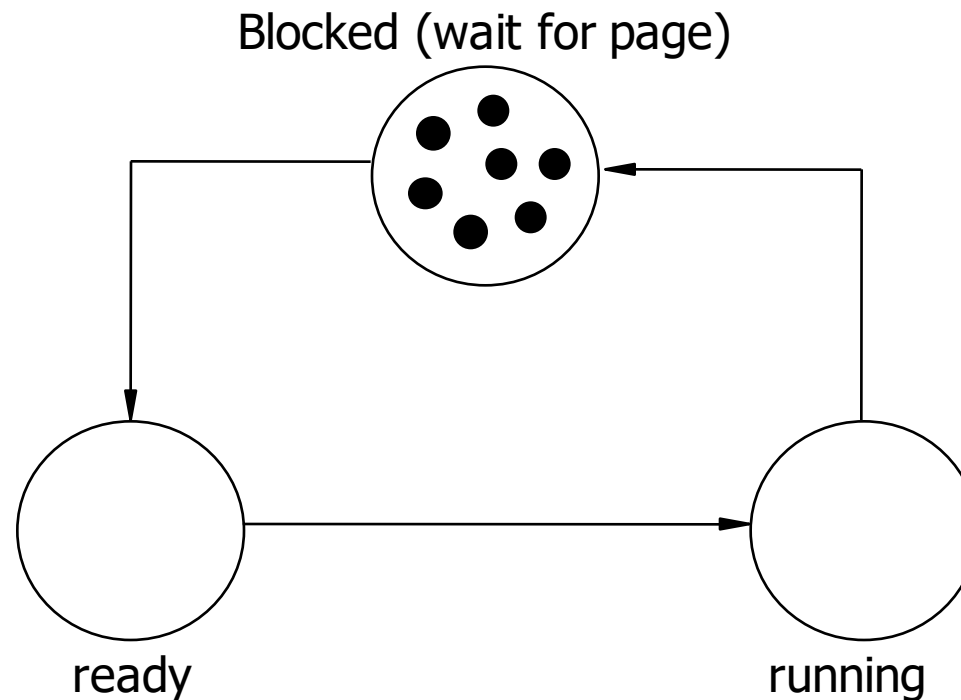
Case: $t_T > t_S$



- In the second case, we experience phases where the processor is idle since all processes wait for their pages to be swapped in.

7.6.2 Thrashing Effect

- The system is completely occupied with paging and cannot perform regular useful work.



Goal: High processor utilization



Many programs executed simultaneously



High multiprogramming degree n



Low memory space s per process



Short time between successive page faults



Congestion at paging device (disk)

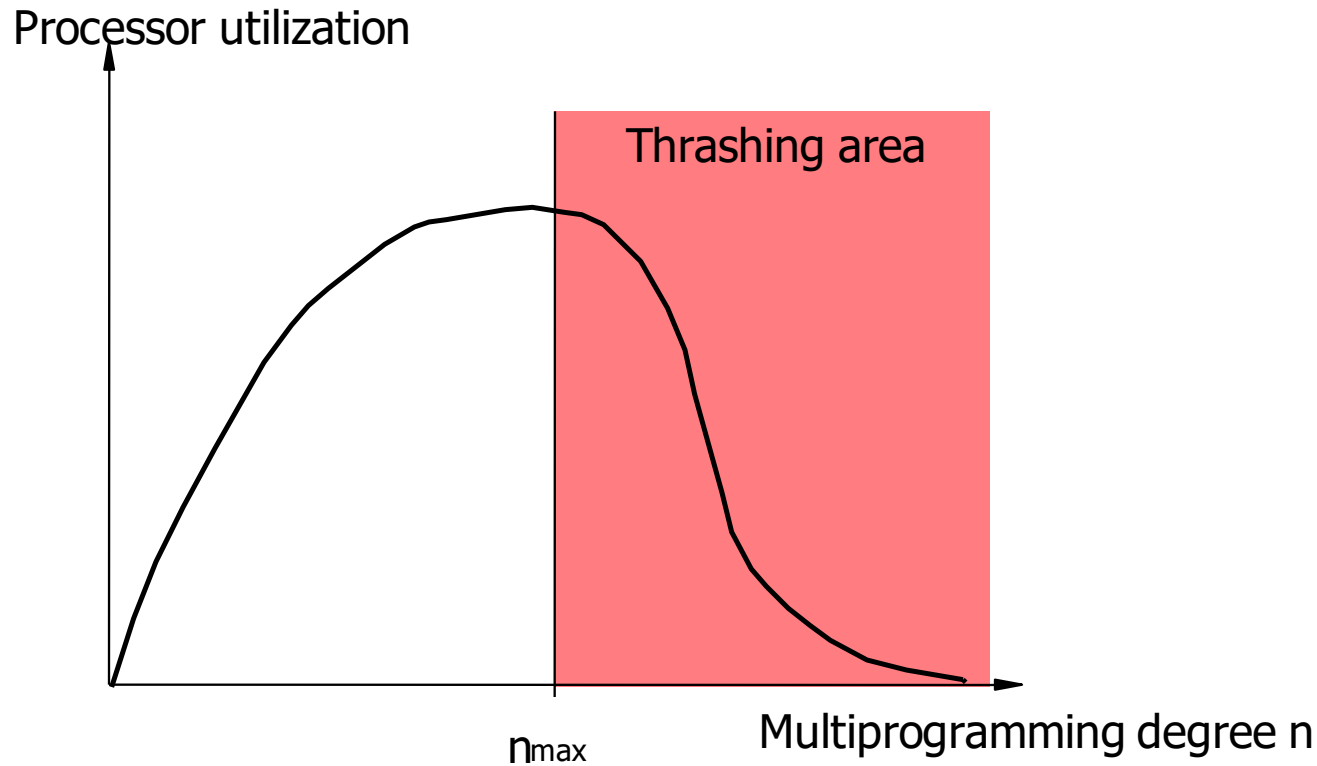


Almost all processes blocked



Result: poor processor utilization

Thrashing Curve



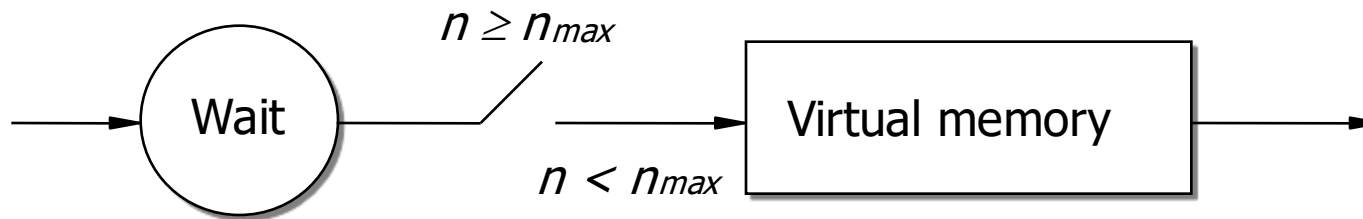
- We have to take care that the system does not enter the overload region.

Overload phenomena

- Thrashing is a special variant of an overload phenomenon that can be found in many areas (not only in computer science) and always leads to a performance collapse.
- Examples:
 - Computer networks too many packets
 - Telephone networks: too many calls
 - Database systems: too many transactions
 - Road traffic too many cars
 - Parallel computing too many processors
- Reason:

Overhead for coordination grows overlinearly.

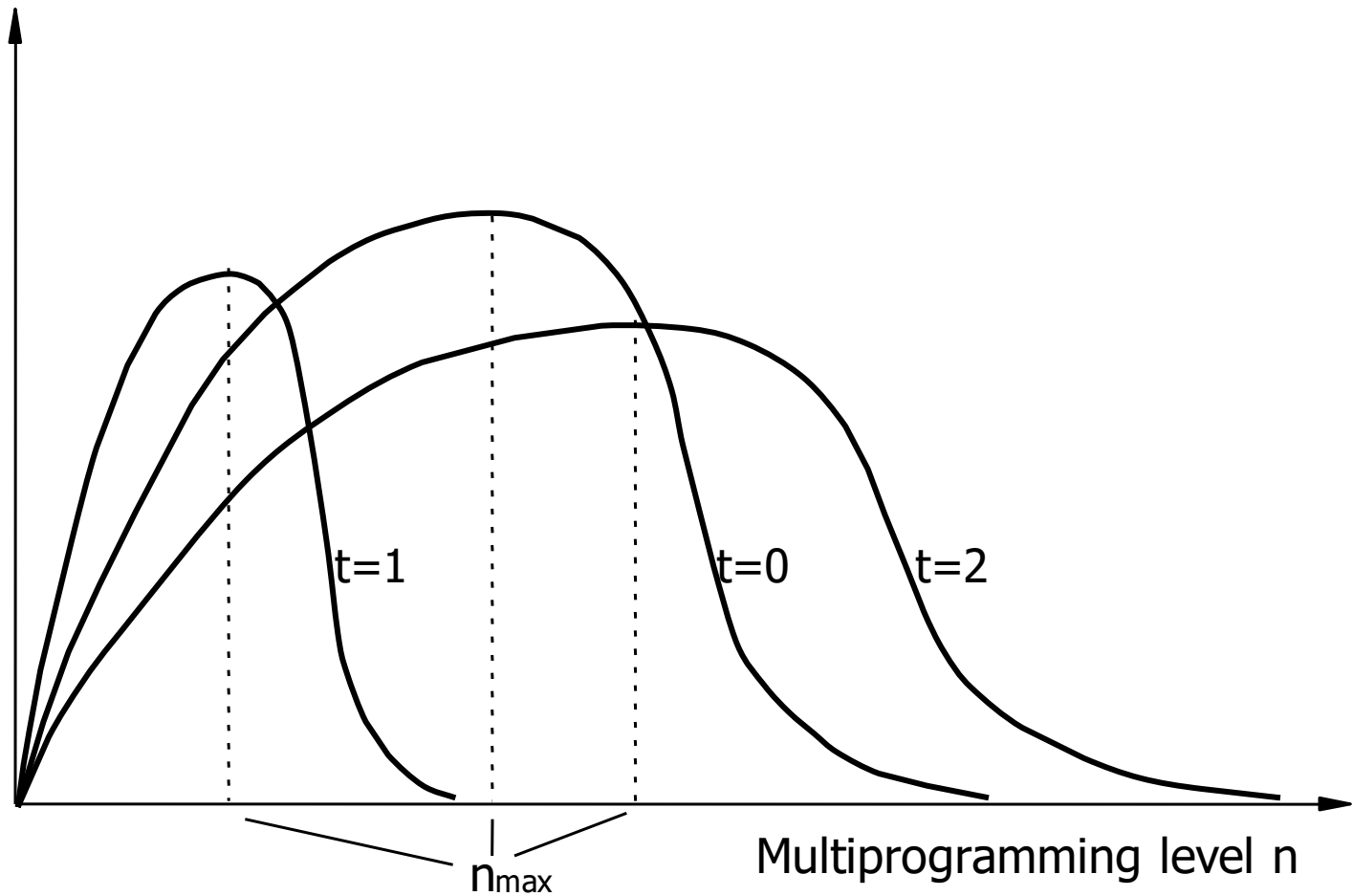
- To prevent the thrashing effect, the multiprogramming level must be limited.



- Problem: How to find an optimal n_{max} ?
- Difficulty:
Program behavior changes over time:
 - Individual program behavior changes
 - Combination of program set in memory changes (multiprogramming mix)

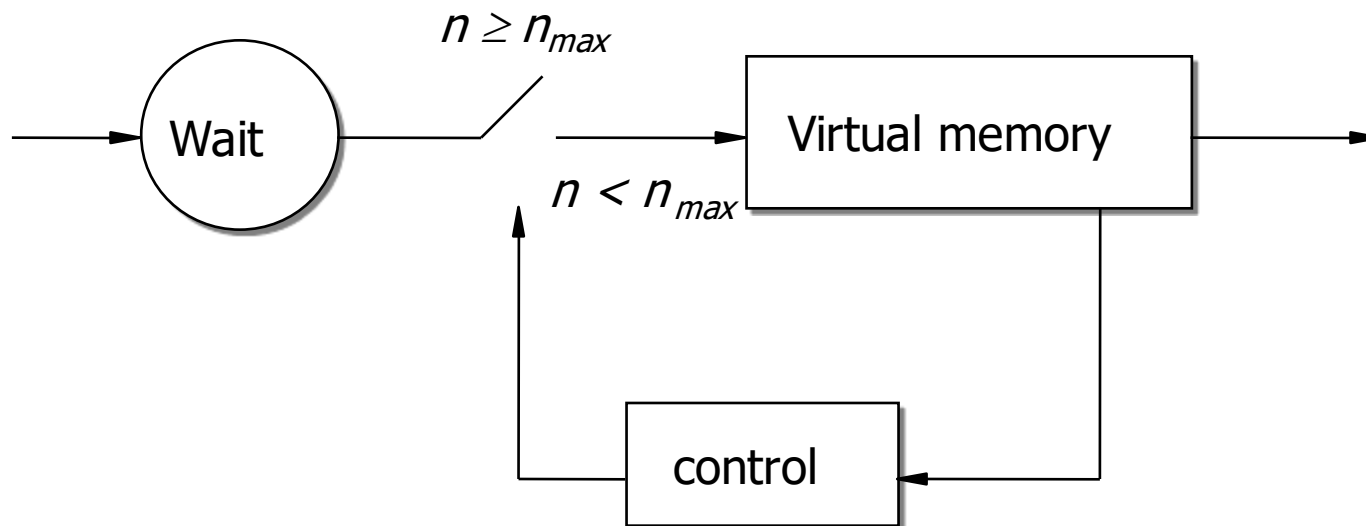
Thrashing curve dynamics

Processor utilization



Thrashing prevention

- The optimal n_{max} turns out in operation and has to be adopted dynamically.
- Thrashing prevention is therefore done by feedback control.



Thrashing prevention

Two strategic approaches:

- *Indirect* or *local* strategy

For each process i a reasonable number of frames s_i is determined dynamically.

The maximum multiprogramming level can be found indirectly:

$$n_{max} := \max \left\{ n \mid \sum_{i=1}^n s_i \leq K \right\}$$

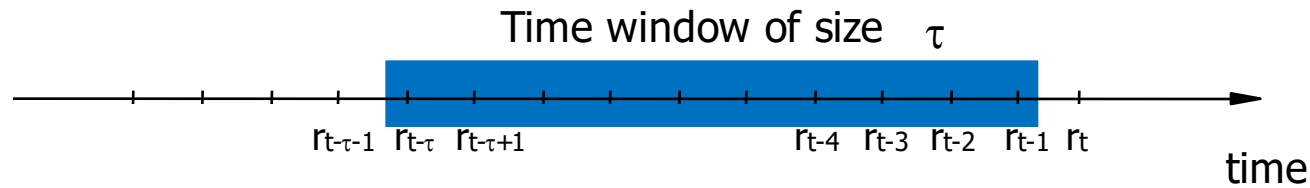
- *Direct* or *global* strategy

The measurement of the global paging activity leads to the calculation of an optimal n_{max} .

7.6.3 Local control of paging activity

- **The Working-Set Model**

- The Working-Set of a program i is defined as the set of pages that have been referenced within the last τ time units.

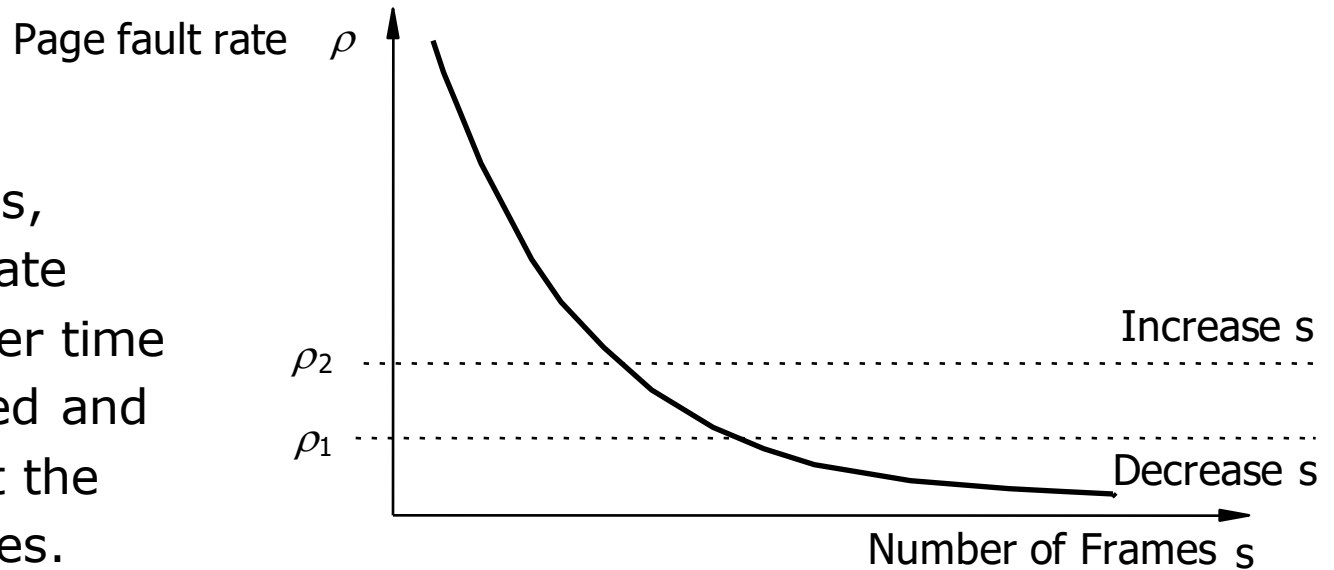


- With suitable choice of t the size of the working set $w_i(t, \tau) := |W_i(t, \tau)|$ indicates the number of page frames that the process needs for efficient work.
- $w_i(t, \tau)$ is estimated using the reference information for each process.
- A new process x is loaded into memory, only if

$$w_x \leq K - \sum_{i=1}^n w_i(t, \tau)$$

- It is the goal of the algorithm that all processes can accommodate their particular working sets in the memory.

- For each process, the page fault rate (#page faults per time unit) is measured and serves to adjust the number of frames.

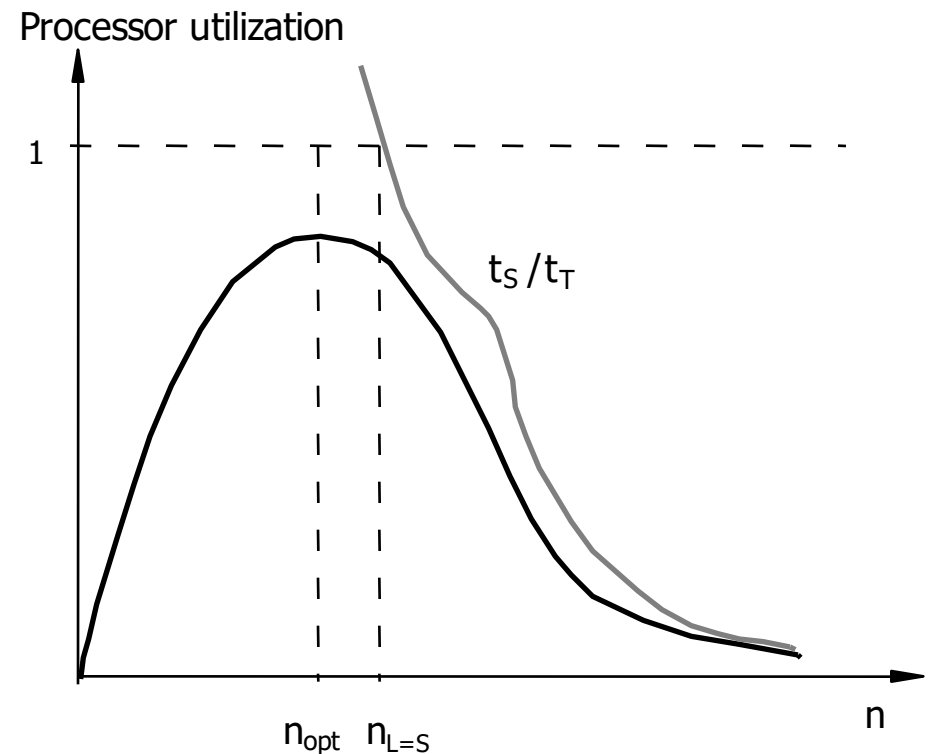


- Control mechanism: $\rho < \rho_1 : s =: s - 1$
 $\rho > \rho_2 : s =: s + 1$
- The multiprogramming level can be calculated indirectly as with the Working-Set algorithm.

The criterion of the interpagefault time (L=S-criterion)

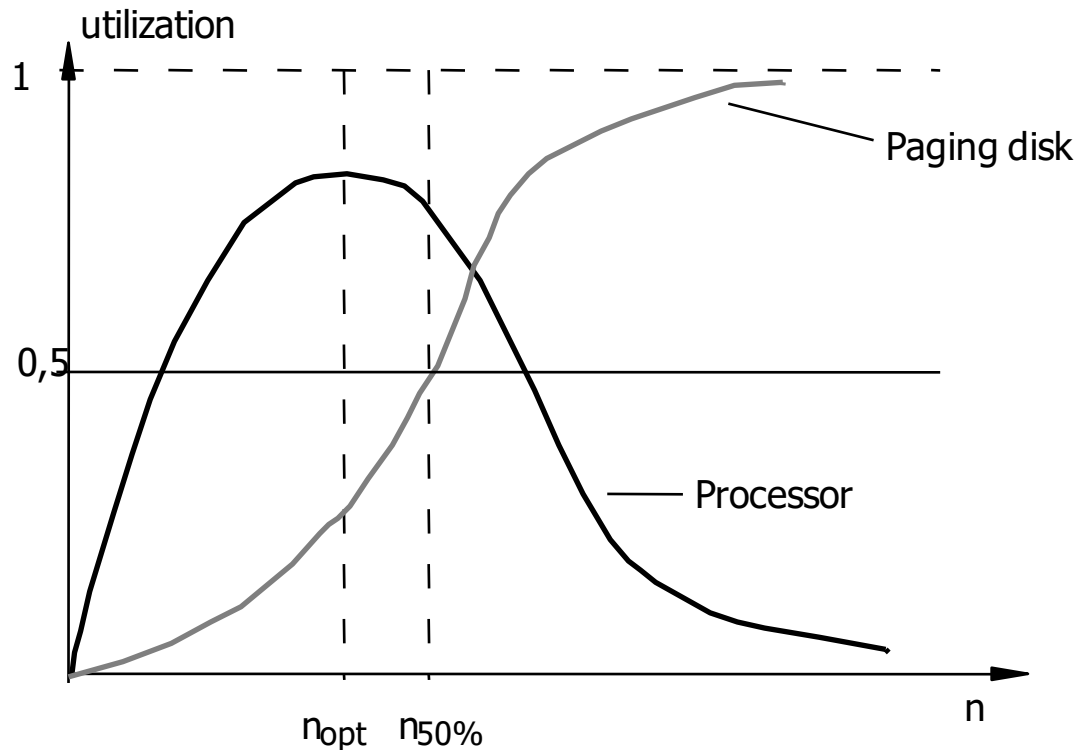
- The time between two page faults t_s (or L resp.) should be roughly the same as the page transfer time t_T (or S , resp.).

The resulting operation point is in most cases too far at the right which can be taken into account in the control laws.



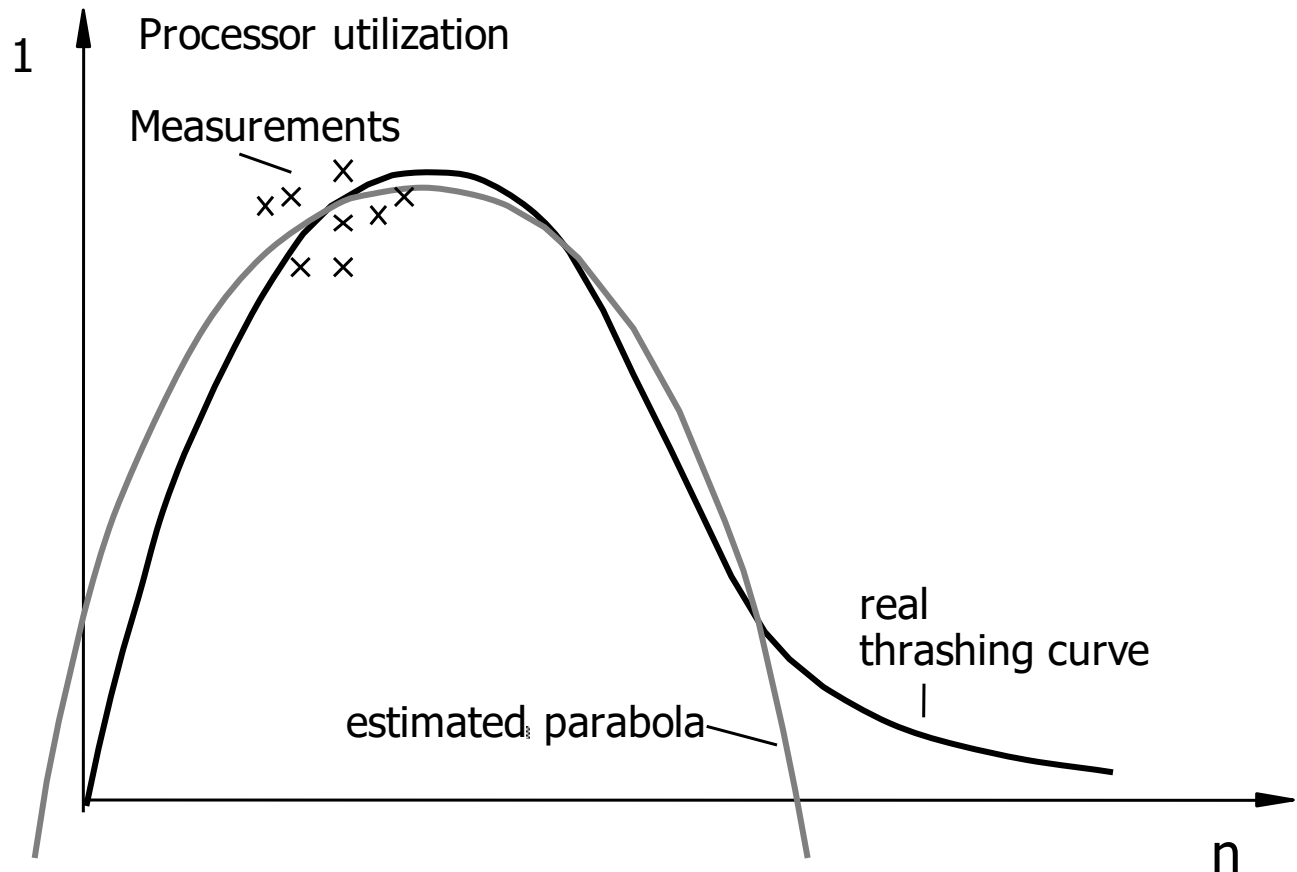
The 50%-Rule

- Thrashing happens, when many processes are blocked due to paging, i.e. when the mean queue length at the paging device is larger than 1.
- According to queuing theory this corresponds to a device utilization of $> 50\%$.
- New processes are loaded to main memory only if the utilization of the disk (measured over a longer time period) is below 50% .



Parabola approximation

- The thrashing curve can be approximated by a parabola in the region of the maximum.



Parabola approximation

- Approximation formula $\eta = a_0 + a_1n + a_2n^2$
- The coefficients a_0, a_1, a_2 are dynamically estimated based on measurements (η_t, n_t) .
- The apex n^* of the parabola can be calculated and used as upper bound of the multiprogramming level.
- If the estimation results in parabola that opens upward, the apex (extreme point) cannot be used as optimum.
- In this case the first derivative indicates the slope, i.e. whether we are left or right of the optimum.
- The current upper bound can then be incremented or decremented.

Swapping

- Early Unix systems did not have a virtual memory.
- The main memory had been managed as a resource with preemption, i.e. processes and their address spaces were swapped to disk, if
 - No space for process generation (fork) was available,
 - A dynamic memory request could not be satisfied.
- The process to be swapped out was chosen according to the following criteria:
 - State – blocked processes were favored for swapping out
 - Priority and residence time in memory
- Priority and time since its last swap-in are added.
- The process with the highest value is swapped out.
- Management of memory and swap area is done using a separate list-based mechanism with First-Fit.

- Today's Unix Systems all provide virtual memory (demand paging).
- When a page fault occurs the missing page is loaded into an empty page frame.
- A special server process (*page-daemon*) has to take care that a sufficient number of empty frames (*lotsfree*) is always available.
- If there are too few empty frames available, the page daemon starts to flush pages to disk.
- For that, a *global* Second-Chance-Algorithm is used.
- The different Unix-systems use different variants.
- To prevent thrashing, Unix uses swapping, i.e. entire processes (address spaces) are swapped out.

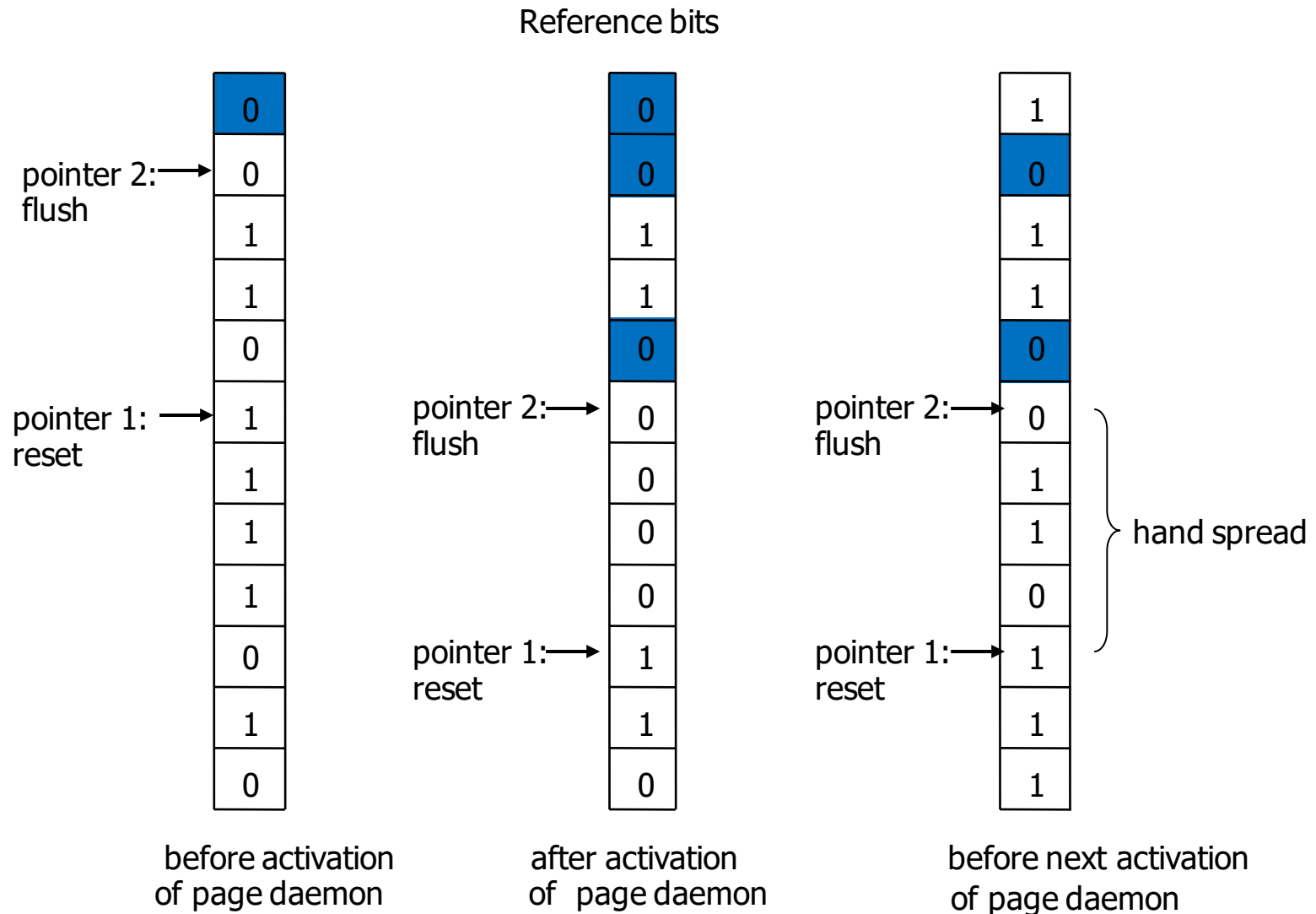
- **AT&T System V:**

- Original Second-chance-Algorithm
- Instead of *lotsfree* two parameters *min* and *max* are used
 - Activation, if current no. of frame $< min$
 - stop, if current no. of frame $> max$

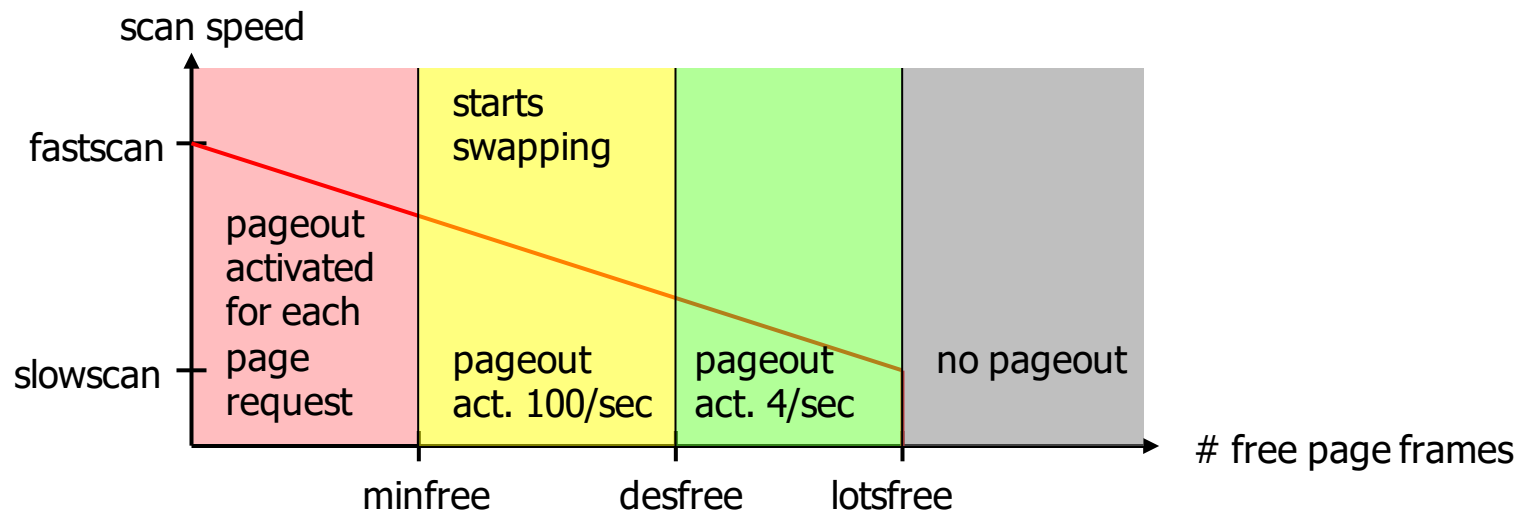
- **4.3BSD:**

- Modified Second-chance-Algorithm (Two-Hand-Clock-Algorithm)
- Parameter *lotsfree*
 - Activation, if current no. of frame $< lotsfree$
 - stop, if current no. of frame $> lotsfree$

Modified "Second Chance"-Algorithm (Two-Hand-Clock-Algorithm, Unix 4.2BSD)



- Solaris (Sun Microsystems/Oracle) also uses the 2-Hand-Clock (page out) with the following parameters
 - hand spread: difference between the two hands (# Frames)
 - scan speed: speed of frame scanning (slow: 100 frames/sec, fast: 8192 frames/sec)
 - lotsfree: amount at which paging sets in (e.g. 1/64 of total number of frames)
 - desfree: desirable amount of empty frames
 - minfree: minimal amount of empty frames



7.7.2 Memory management in Windows

In contrast to Unix, Windows uses a *local paging strategy* :

- If, as a consequence of a page fault, a page needs to be swapped out, always a page of that process which caused the page fault is chosen.
- Not only the missing page, but also some more of the „neighborhood“ of that page is swapped in (*clustering*).
- The set of all currently loaded pages of a process is called working set.

- The paging strategy depends on the hardware:
 - *FIFO* (modified) for Alpha processors and Intel multiprocessor systems
 - *Clock* for Intel monoproductors
- The size of the working sets is initialized by default values (Min and Max).
- On demand Working Sets can grow beyond the maximum (*Working set expansion*) and shrink again (*Working set trimming*).
- Both are dependent on the page fault rate and on the number of free frames.
- For the OS itself also a working set mechanism is used.

- Stallings, W.: *Operating Systems*, 5th ed., Prentice Hall, 2005, Chapter 7 (7.1+7.2) , Chapter 8
- Tanenbaum, A.: *Moderne Betriebssysteme*, 2.Aufl., Hanser, 1995, Kapitel 3, Abschnitt 3.1+3.2
- Knuth, D.E.: *The Art of Computer Programming*, Vol. 1, 3rd ed., 1997, pp. 435-451
- Peterson, J.; Norman T. A.:
Buddy systems. CACM 20, 6 (June 1977, pp. 421-431
- Shore, J.E.: *Anomalous behavior of the fifty-percent rule in dynamic memory allocation*. CACM 20, 11 (Nov. 1977) pp. 812 - 820
- Denning, P.J.: *Working Sets Past and Present*, IEEE TOSE, Vol 6, (Jan. 1980) pp. 64-84.

- Heiss, H.-U.: *Verhinderung von Überlast in Rechensystemen*, Springer (Informatik-Fachberichte), 1988
- Heiss, H.-U.: *Overload Effects and their Prevention*. Performance Evaluation Vol.12, No.4 (July 1991), S. 219-235.
- Markatos, E.: [*Visualizing Working Sets*](#), ACM Operating Systems Review 31,4 (1997), pp.3-11
- Megiddo, N.; Modha, D.S.: *Outperforming LRU with an Adaptive Replacement Cache Algorithm*, IEEE Computer, April 2004