# Course
# TÖL401G: Stýrikerfi /
# Operating Systems
# 9. Virtual-Memory Management

# Chapter Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the optimal, FIFO, LRU and Second-Chance page-replacement algorithms.
- Describe thrashing the working set of a process, and explain how it is related to program locality.
- Explain memory compression as alternative to paging out to storage devices
- Describe advanced applications of virtual memory, e.g. copy-on-write or demand loading.
- Explain management of kernel-internal memory.

# Contents

1. Introduction
2. Virtual Memory Using Demand Paging
3. Page Replacement Algorithms
4. Allocation of Frames,
5. Thrashing & Working-Set Models
6. Memory Compression Instead of Paging Out to Storage Device
7. Advanced Applications of Virtual Memory
8. Allocating Kernel Memory
9. Summary

Note for users of the Silberschatz et al. book: newer editions have the material ordered differently than on my slides. (In particular, memory mapped files are covered there in this chapter, while I will cover them in the next chapter.)

# 9.1 Introduction

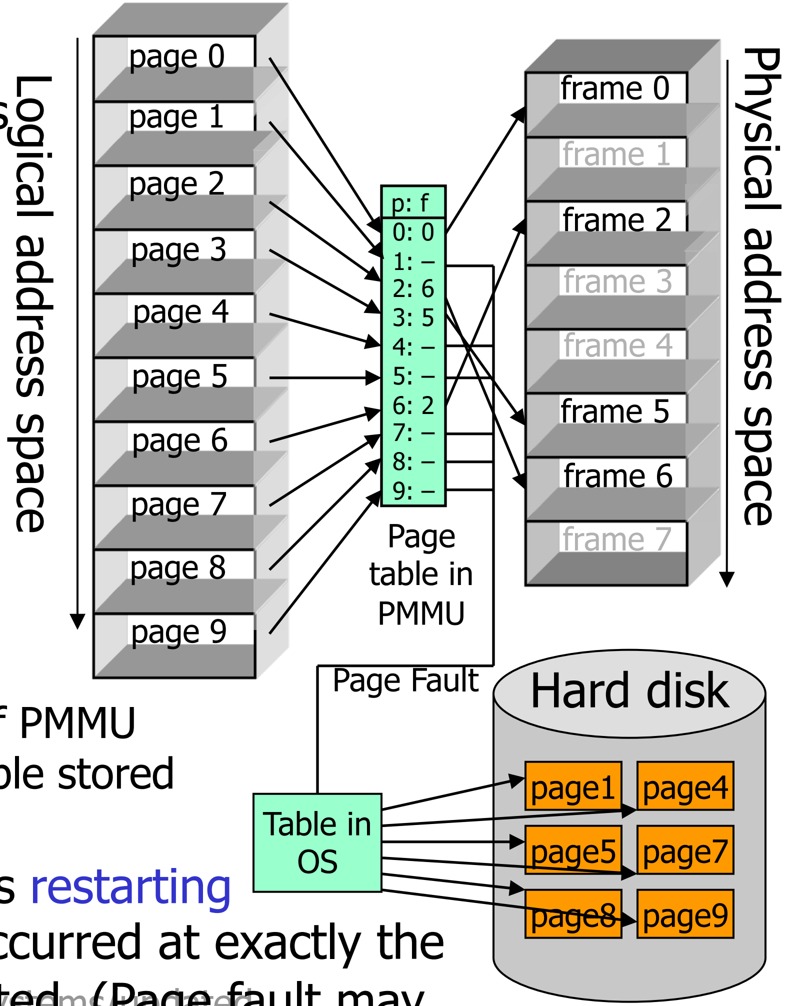
- Even though the previous chapter presented already some advanced usages of an MMU, it was still necessary that the complete logical address space used by a process is kept in physical memory while the process is executed.
  - E.g. swapping always swaps out and in the used logical address space of a process as a whole. It is not possible, that only a part of a process is swapped in or out.
- However, in practise, a process rarely accesses all of its logical address space or at least not all parts of the logical address space need not to be in physical memory at the same time.

# Virtual Memory

- Virtual memory: separation of logical memory from physical memory, i.e. logical address space does not need to map 1:1 on existing physical memory anymore.
  - Only part of a program needs to be in memory for execution.
    - Instead of swapping whole processes in and out, only parts need to be stored and reloaded from hard disk (less I/O ⇒ significant speed up).
    - More programs can be kept in physical memory (increased degree of multiprogramming) if only that part of each program’s logical address space is kept in physical memory that is actually currently needed.
  - Logical address space can be much larger than physical address space.
    - Programmers (& users) do not need to worry about the amount of physically available memory.
- Virtual memory typically implemented via demand paging.
  - Efficient: used by all of today's major operating systems.

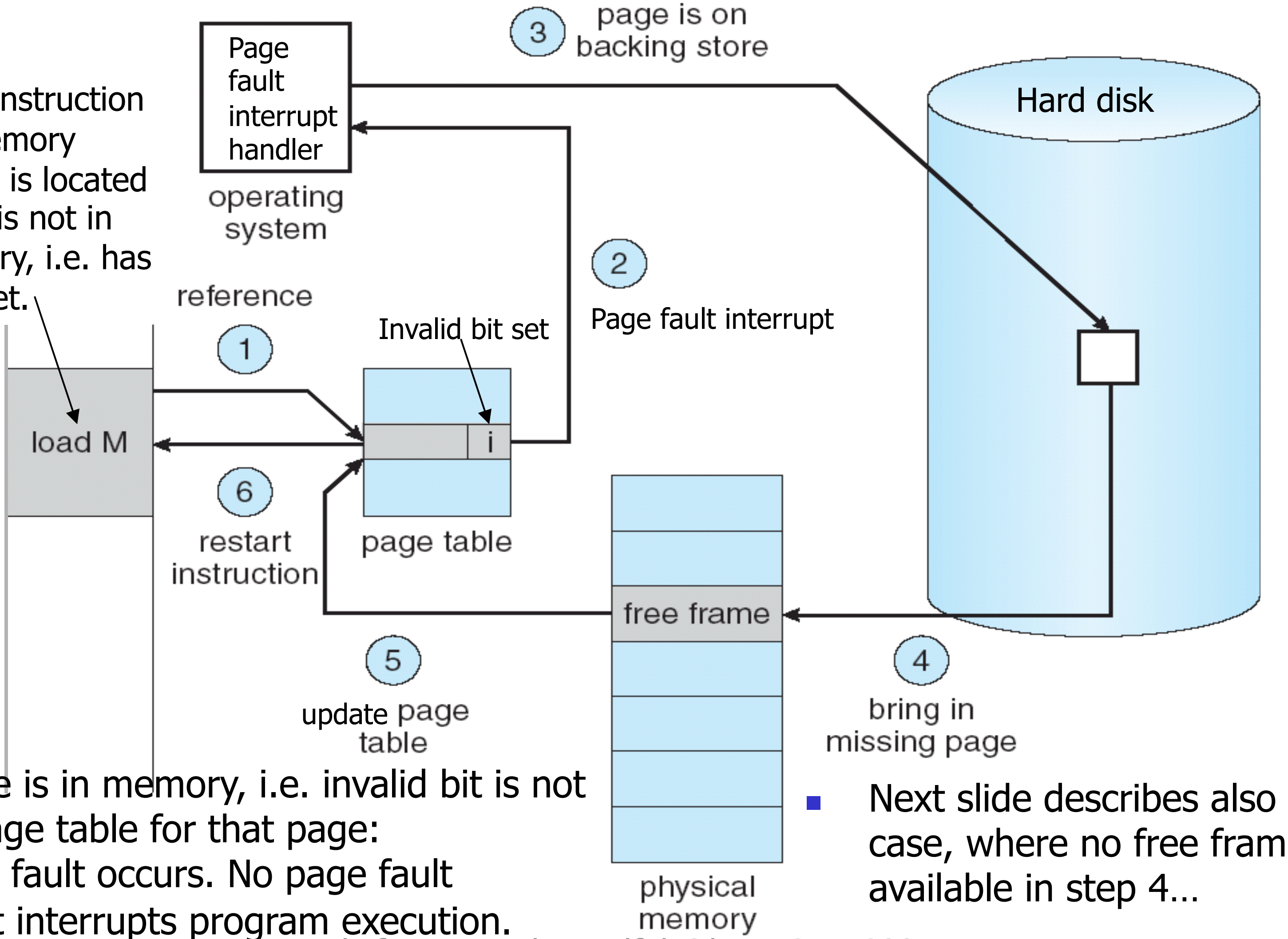
# 9.2 Virtual Memory Using Demand Paging

- Roughly comparable to idea of swapping; however, instead of whole processes, pages are used:
  - Bringing in/out single pages.
  - Do not bring in page unless it is accessed ("lazy paging").
    - Page currently not in physical memory will have in its page table entry the valid bit set to invalid: access to page triggers page Fault Interrupt that is serviced by OS and will call pager routine to bring in page.
  - Bring out a page only when physical memory is needed for another new page.
  - At each context switch, page table pointer of PMMU needs to be updated to point to the page table stored in the PCB of the particular process.
- Requires Paged MMU and CPU that supports restarting an instruction after a page fault interrupt occurred at exactly the same place and state where it was interrupted. (Page fault may occur at any memory access.) →Next slide.



# Procedure for Handling a Page Fault (Graphical description)

Machine code instruction referencing memory location M that is located in a page that is not in physical memory, i.e. has its invalid bit set.



- If a page is in memory, i.e. invalid bit is not set in page table for that page:
  No page fault occurs. No page fault interrupt interrupts program execution.
- Next slide describes also the case, where no free frame is available in step 4…

# Procedure for Handling a Page Fault (Textual description)

- Access to page that is not in physical memory (valid bit of page is set to "invalid"). ⇒ Page fault interrupt generated by PMMU.
- Page fault interrupt handler of OS checks internal table (stored in PCB of current process) to determine whether page is invalid because it
  - refers to non allocated memory (illegal access): ⇒ process has gone wild: terminate process.
  - refers to a page that is currently on hard disk ⇒ needs to be brought in:
    - If a free frame is available in the physical memory:
      read page from disk into free frame.
    - If no free frame is available in the physical memory :
      - Select a victim frame that will be replaced with contents of the new page.
      - Was victim frame modified due to a write access (modified bit of page table entry set)?
        - Yes: Save victim frame contents on disk.
      - Read required page from disk into victim frame.
    - Update page table: page is valid now, reset modified bit.
    - Inform scheduler that process is ready and interrupted instruction may be restarted.

# Performance of Demand Paging

- Effective Access Time (EAT), if page
  - is in memory: EAT := time of memory access cycle (e.g. 160ns).
    - (Assuming that page table entry is in TLB.)
  - is not in memory: EAT := service page fault interrupt (e.g. 100μs)

    + write page to disk (if modified) (e.g. 8ms)

    + read page in from disk (e.g. 8ms)

    + restart process (e.g. 100μs)

    + time of memory access cycle (e.g. 160ns)

    ($\Sigma \approx$ e.g. 16ms)

Note: SSDs are approx. 100-1000 times faster than harddisks: access time, e.g. 80μs, instead of 8ms. EAT would then be only 1000 times slower.

- EAT of page fault is about 100 000 times slower than EAT of a page hit.
  - In practise, the problem is not as severe, because while the process with the page fault is blocked for 16ms, other processes may be ready and execute during the disk access of the pager.
  - But still, good page replacement algorithms needed that take care that only those pages are removed from physical memory (and replaced by other pages) that are unlikely to be used.

# 9.3 Page Replacement Algorithms

- If no frames are free at a page fault, the OS needs to select a victim frame whose content will be replaced by the content of the requested page.
  - Page replacement algorithms desired that lead to as few page faults (=slow hard disk accesses) as possible.

- Note: The same algorithms are required in many other areas of computer science where caching is involved, e.g. caches in CPU hardware, caches in software, for storing values that have already been processes before:
  - Updating caches: which cache entry to replace after a cache miss?

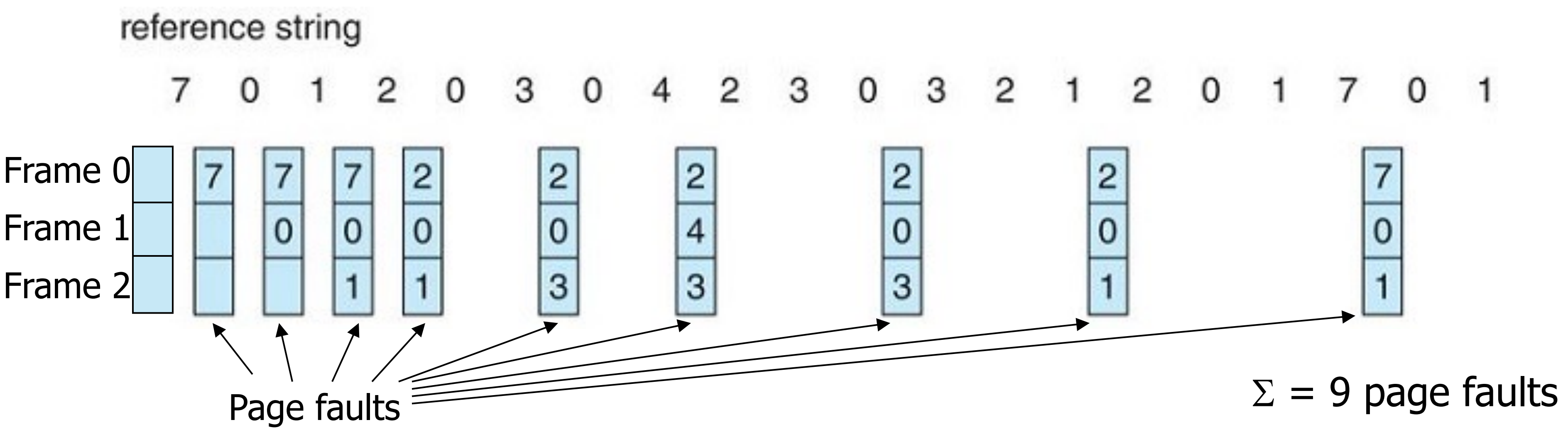
⇒ Page replacement algorithms relevant for areas outside of operating systems.

# Page Replacement Algorithms: Optimal Policy (OPT)

- Algorithm: Replace that frame containing the page that will not be used for the longest period of time in future.
- Guarantees the lowest possible page fault rate!
  - Unfortunately, not implementable: algorithm must be able to predict the future!
    - However, due to undecidability, it cannot be predicted which instructions will be executed in future.
  - Nevertheless, the optimal strategy can be used for comparing and evaluating other page replacement algorithms.
    - Comparison only reasonable when comparing performance for the same fixed string of page references and same number of frames.

# Page Replacement Algorithms: Optimal Policy (OPT)

- Example:
  - 3 frames (initially empty) with the following string of page references:



reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Page faults

Σ = 9 page faults

Applies to all page replacement algorithms: the frames are empty in the beginning and as long as empty frames are available, these are taken (and this counts as a page fault because the respective page is not in any of the frames).

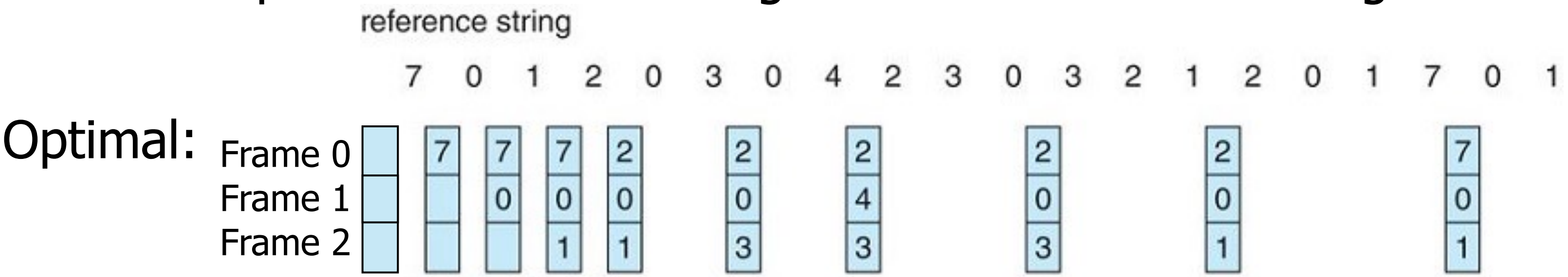
The actual page replacement algorithm is only applied if a page is accessed, but not in memory (i.e. due to a page fault interrupt. Then, a frame needs to be found for that page that is to be brought in.)

# Page Replacement Algorithms: First-in, First-out Policy (FIFO)

- As we cannot look into future as required for OPT, maybe start with a simple algorithm that does not require this: FIFO.
- Algorithm: When a page must be replaced, chose that frame containing the oldest page.
  - I.e. replace page that resides for longest time in the set of frames (i.e. page that is "oldest" in terms of residence in memory).
- Advantage:
  - Easy to implement using FIFO queue (size of queue = number of frames). Page at head of queue (=page that is oldest in queue) will be removed to make space for new page.
    - Well, in practise, to consider all the used frames being the FIFO queue would mean to move each time for each and every frame the content of that frame from to the next frame = copying gigabytes of RAM which would be very slow.
    - In practise, a circular buffer as used where frame contents stays in each frame, but rather a pointer to head of queue advances like a clock hand after each page fault.
- Disadvantage:
  - Even though a page is old, it might be the page that is most frequently used, i.e. in this case it would not be wise to replace that page…

# Page Replacement Algorithms: First-in, First-out Policy (FIFO)

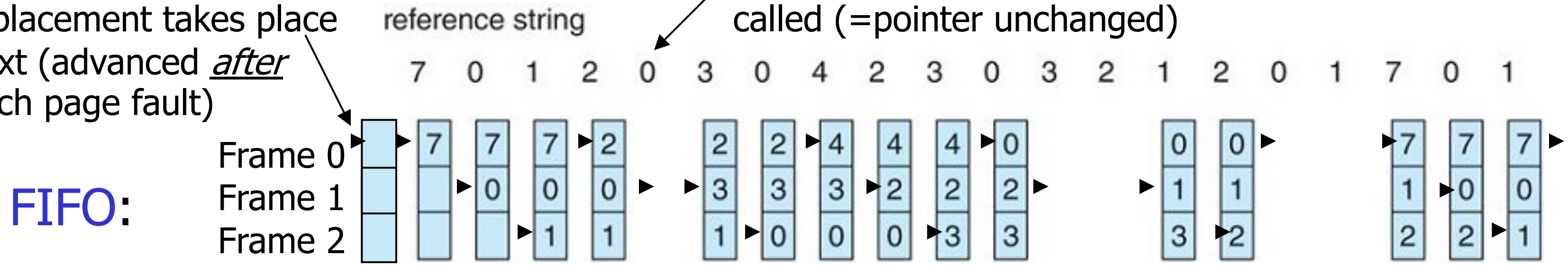
- Example:
  - Comparison with OPT using the same reference string as before.



Σ = 9 page faults

Pointer where next replacement takes place next (advanced *after* each page fault)

No page fault: algorithm not called (=pointer unchanged)



Σ = 15 page faults

Implemented using circular buffer (see also Clock algorithm): Pointer to head of queue advances like a clock hand *after* each page fault. I.e. pages stay in their frame instead of shifting them. If pointer reaches last frame, it starts again at first frame (like a clock).

# Page Replacement Algorithms: Least-Recently-Used Policy (LRU)

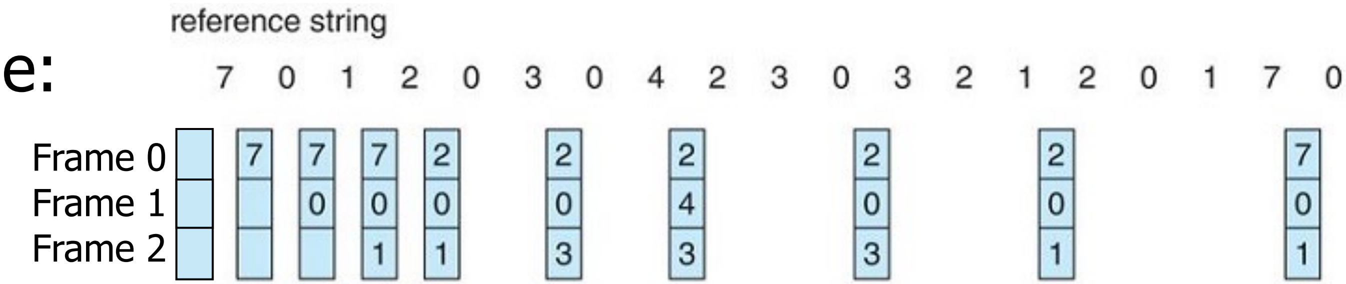
- FIFO is too simple as it does just look at the time when a page was brought in memory instead of considering the time when a page was recently used. We can try to approximate OPT based on the assumption that a page that has been recently used in the past will also be used soon in the future ("locality of execution"): LRU.

- Algorithm: Replace the page that has not been used for the longest period of time (i.e. page that is "oldest" in terms of being referenced).
  - At each reference to a page, the page needs to be timestamped to able to identify the page that is least recently used.
- Advantage:
  - Good approximation of OPT.
- Disadvantage:
  - No PMMU found in a modern CPU supports timestamping a page at each access! I.e. only implementable with special extra hardware.

# Page Replacement Algorithms: Least-Recently-Used Policy (LRU)

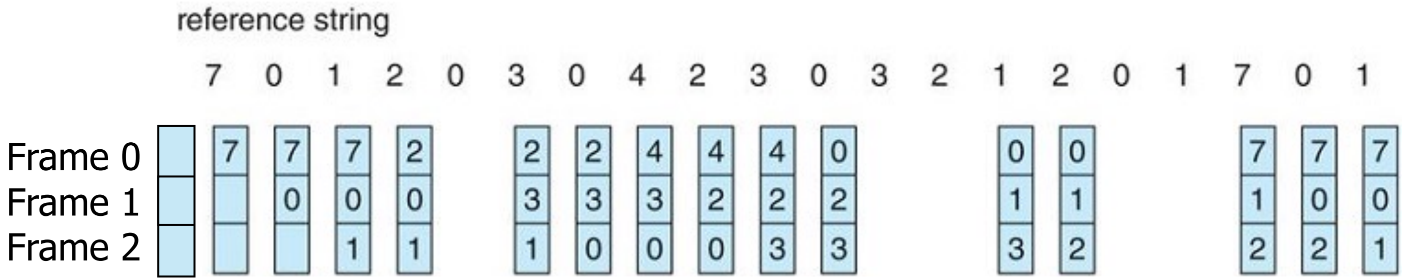
- Example:

Optimal:



Σ = 9 page faults

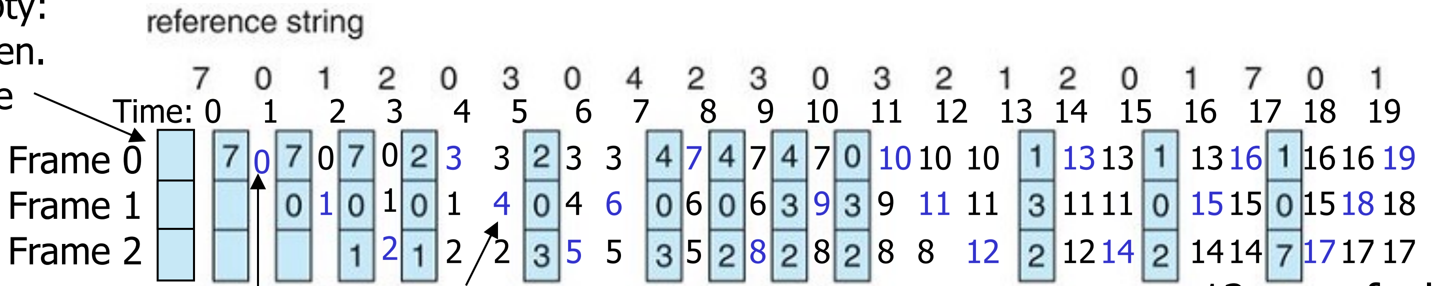
FIFO:
(Implemented using circular buffer)



Σ = 15 page faults

Initially, all frames empty: any frame can be chosen. Let's assume first frame chosen first.

LRU:



Σ = 12 page faults

Time stamp No page fault, but access updates time stamp

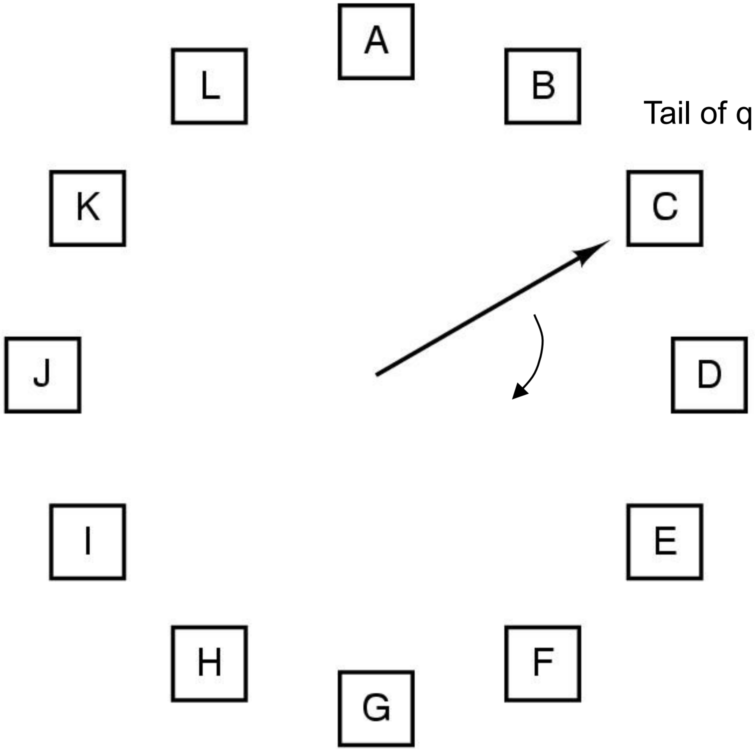
# Page Replacement Algorithms: Second-Chance Policy

- LRU seems to be better than FIFO; However, even though not impossible to implement, LRU requires additional hardware. Hence, let’s try a mixture of LRU and FIFO, where the oldest page in memory is only replaced if it has not been referenced: Second-Chance policy.

- Algorithm: Inspect (in a FIFO style) the page-table entry of the oldest page (in terms of residence in memory) in memory.
  - If the page table entry’s reference bit (→ch. 8: gets set at each read or write access to an address inside that page) is not set: replace this page.
    - (I.e. page is both old and not recently used.)
  - If the reference bit is set: move page from head of queue to tail of queue and reset reference bit of that page. Start over from new head of queue.
    - (I.e. even though page is old, it has been recently used: give it second chance by treating it as new page.)

# Page Replacement Algorithms: Second-Chance Policy Implemented Using Clock Algorithm

- As we have already seen in the example for FIFO, FIFO-style queues are typically implemented using a circular buffer. When representing a circular buffer graphically, it looks like a clock: ⇒ Second-Chance policy is often implemented (and referred to) as Clock algorithm.



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

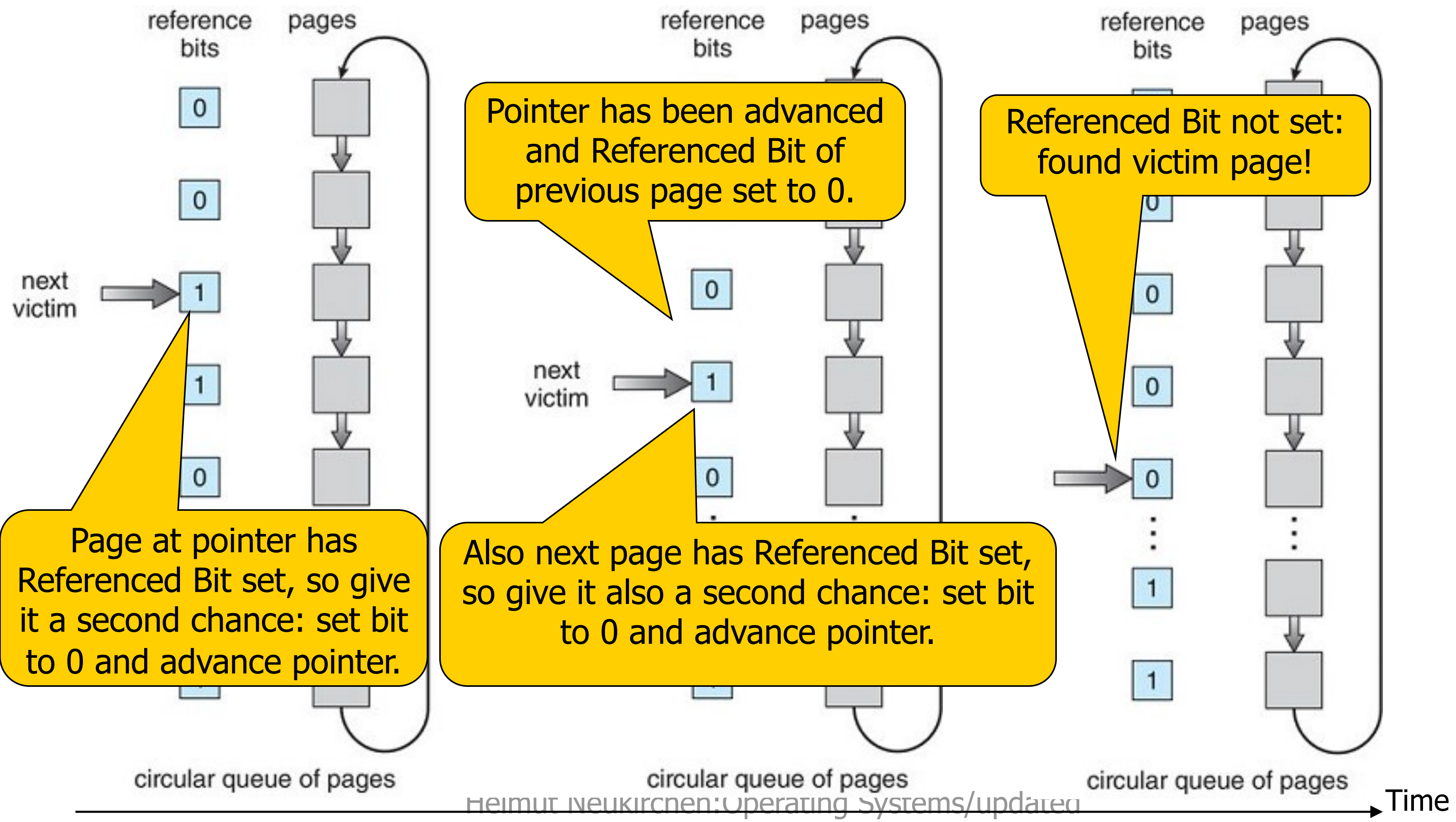
R = 0: Evict the page*

R = 1: Clear R and advance hand and start over.

* I.e. place new page into the frame of the evicted page. After that, advance hand (which then points to the oldest page – just like FIFO).

Note: hand is only advanced in case of page fault: only in this case, interrupt is raised and only then, routine that advances hand can get called by OS interrupt handler.

# Page Replacement Algorithms: Second-Chance Policy Implemented Using Clock Algorithm: Searching a page to evict



Page at pointer has Referenced Bit set, so give it a second chance: set bit to 0 and advance pointer.

Pointer has been advanced and Referenced Bit of previous page set to 0.

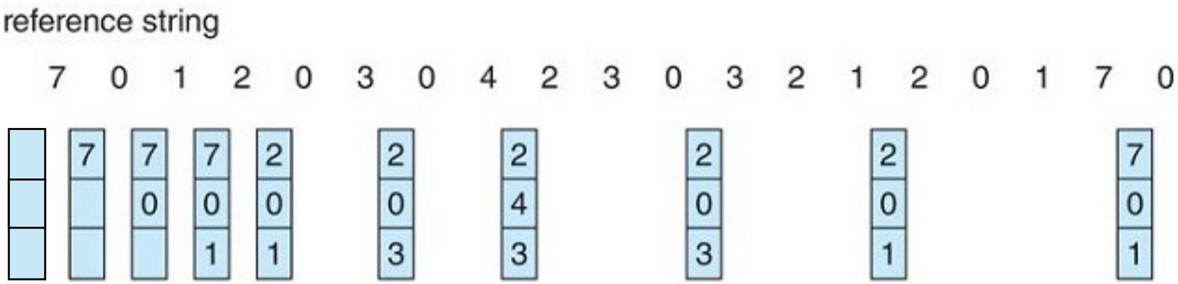
Also next page has Referenced Bit set, so give it also a second chance: set bit to 0 and advance pointer.

Referenced Bit not set: found victim page!

# Page Replacement Algorithms: Second-Chance/Clock

- Example:

Optimal:

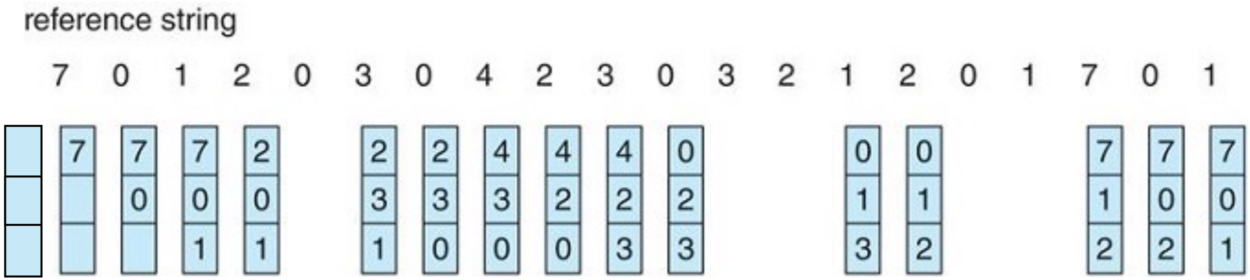


reference string

| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | 7 | 7 | 7 | 2 | | 2 | | 2 | | | 2 | | | 2 | | | | 7 | | |
| Frame 1 | | 0 | 0 | 0 | | 0 | | 4 | | | 0 | | | 0 | | | | 0 | | |
| Frame 2 | | | 1 | 1 | | 3 | | 3 | | | 3 | | | 1 | | | | 1 | | |

Σ = 9 page faults

FIFO:
(Implemented using circular buffer)



reference string

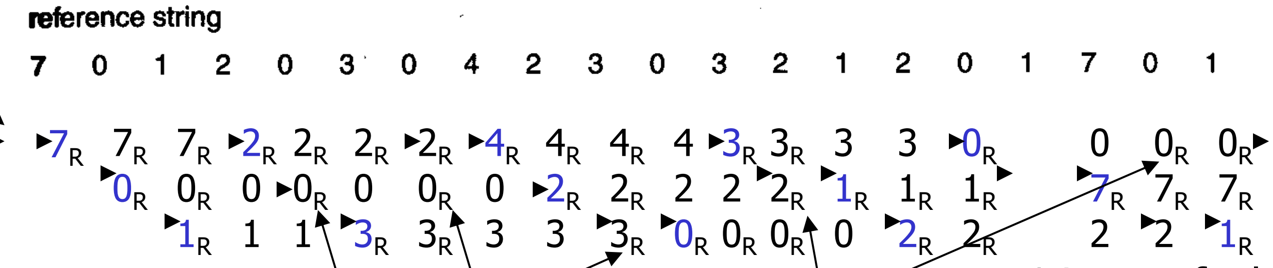
| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | 7 | 7 | 7 | 2 | | 2 | 2 | 4 | 4 | 4 | 0 | | | 0 | 0 | | | 7 | 7 | 7 |
| Frame 1 | | 0 | 0 | 0 | | 3 | 3 | 3 | 2 | 2 | 2 | | | 1 | 1 | | | 1 | 0 | 0 |
| Frame 2 | | | 1 | 1 | | 1 | 0 | 0 | 0 | 3 | 3 | | | 3 | 2 | | | 2 | 2 | 1 |

Σ = 15 page faults

Pointer where *next* replacement will take place

Second-Chance/Clock:

($_R$ means: R bit is set)



reference string

| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0▸ | ▸$7_R$ | $7_R$ | $7_R$ | ▸$2_R$ | $2_R$ | $2_R$ | ▸$2_R$ | ▸$4_R$ | $4_R$ | $4_R$ | 4 | ▸$3_R$ | $3_R$ | 3 | 3 | ▸$0_R$ | | 0 | $0_R$ | $0_R$▸ |
| Frame 1 | | ▸$0_R$ | $0_R$ | 0 | ▸$0_R$ | 0 | $0_R$ | 0 | ▸$2_R$ | $2_R$ | 2 | 2 | ▸$2_R$ | ▸$1_R$ | $1_R$ | $1_R$▸ | | ▸$7_R$ | $7_R$ | $7_R$ |
| Frame 2 | | | ▸$1_R$ | 1 | 1 | ▸$3_R$ | $3_R$ | 3 | 3 | ▸$3_R$ | ▸$0_R$ | $0_R$ | $0_R$ | 0 | ▸$2_R$ | $2_R$ | | 2 | ▸2 | ▸$1_R$ |

Σ = 14 page faults

No page fault, however R bit of page gets set again by PMMU due to reference.

# Page Replacement Algorithms: Second-Chance/Clock

- Advantage:
  - Easy to implement, in particular requires just a feature (reference bit that is set each time a page is either read or modified) that is provided by all modern PMMUs.
  - In fact, the Clock implementation of Second-Chance is used by all major operating systems (or some variants of it)!
- Disadvantage:
  - Just a rough approximation of LRU: Simply “referenced or not referenced” is used as least-recently used criterion. I.e. pages are divided into just two different classes (plus the age information due to the order in the FIFO queue).
- Note: Second-Chance degenerates to FIFO if all pages have their reference bit set.
  - At least, it terminates even in this case, because reference bits are step-by-step reset and finally, the algorithm will inspect the first page again which has now the reference bit reset and will thus be selected as victim.

# Page Replacement Algorithms: Enhanced Second-Chance Policy

- Let’s try an enhanced Second-Chance policy where at least four different classes of pages are distinguished. Sometimes, this Enhanced Second-Chance policy is also called: Not-Recently-Used (NRU).
- Algorithm: Use a classification of pages based on the reference bit and the modified bit of the page’s entry in the page table (R, M):
  1. (0, 0) not referenced, not modified: best page to replace!
  2. (0, 1) not referenced, but modified: not quite as good as page needs to be written to hard disk before replacement.
  3. (1, 0) referenced, but not modified: likely to be used in future again.
  4. (1, 1) referenced and modified: likely to be used in future again and page would need to be written to hard disk before replacement.
  - When having to select a page to replace, go in a clock-style through the circular FIFO buffer and look for the first page that is from class 1: if such a page is found, replace that page. If it is not found (i.e. all the pages of the buffer have been investigated without success), proceed with the next class and start over.
  - In addition, to prevent that after a while all pages have their referenced bit set, the OS does periodically reset the referenced bit of all page table entries. As a result, class 2 may occur (in general, class 2 would not be possible, because when modifying a page, the referenced bit would also be set by the PMMU).

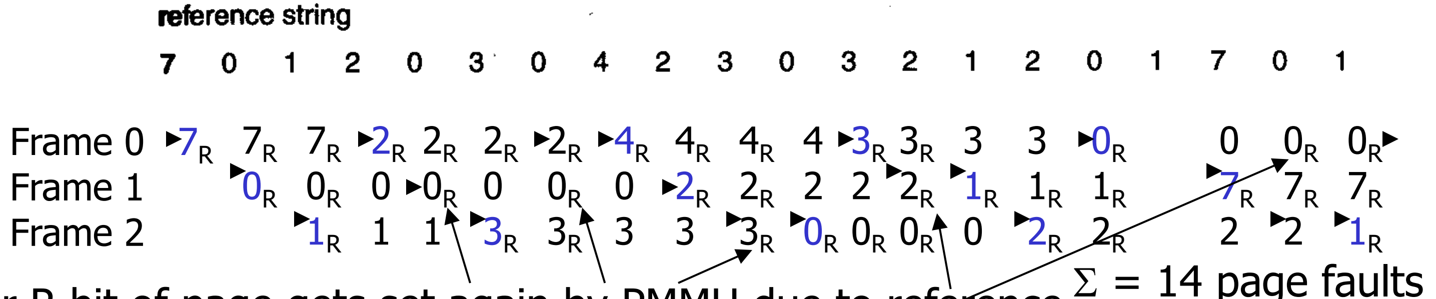
Advance hand like in Second-Chance/ Clock, i.e. while searching and after replacing a frame. But R bit gets not reset during search!

# Page Replacement Algorithms: Enhanced Second-Chance Policy

- Example:

Second-Chance/ Clock:

($_R$ means: R bit is set)



reference string

| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | ▶$7_R$ | $7_R$ | $7_R$ | ▶$2_R$ | $2_R$ | $2_R$ | ▶$2_R$ | ▶$4_R$ | $4_R$ | $4_R$ | 4 | ▶$3_R$ | $3_R$ | 3 | 3 | ▶$0_R$ | | 0 | $0_R$ | $0_R$▶ |
| Frame 1 | | ▶$0_R$ | $0_R$ | 0 | ▶$0_R$ | 0 | $0_R$ | 0 | ▶$2_R$ | $2_R$ | 2 | 2 | ▶$2_R$ | ▶$1_R$ | $1_R$ | $1_R$ | | ▶$7_R$ | $7_R$ | $7_R$ |
| Frame 2 | | | ▶$1_R$ | 1 | 1 | ▶$3_R$ | $3_R$ | 3 | 3 | ▶$3_R$ | ▶$0_R$ | $0_R$ | $0_R$ | 0 | ▶$2_R$ | $2_R$ | | 2 | ▶2 | ▶$1_R$ |

No page fault, however R bit of page gets set again by PMMU due to reference.

Σ = 14 page faults

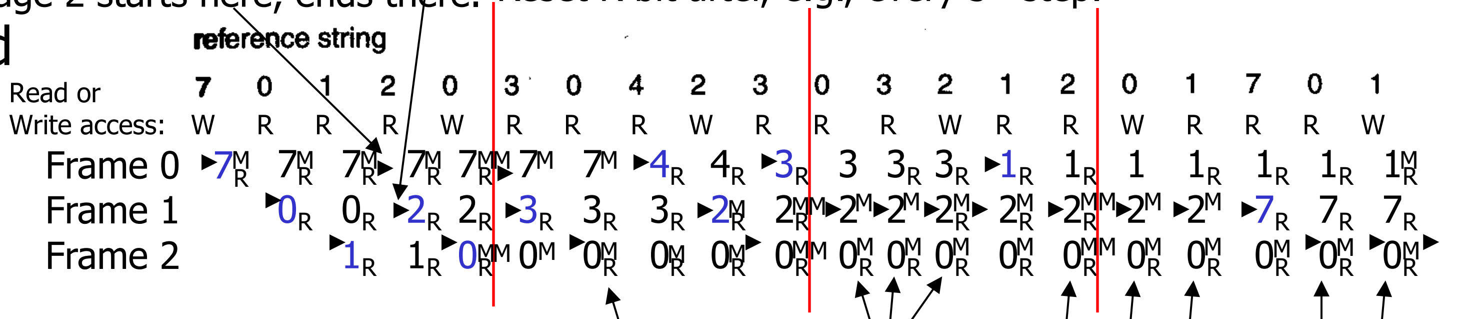
Search for frame for page 2 starts here, ends there.

Reset R bit after, e.g., every 5th step.

Enhanced Second-Chance

($_R$ means: R bit is set, $^M$ means: M bit is set in page table entry)

R,M
- (0,0)
- (0,1) M
- (1,0) R
- (1,1) RM



reference string

| | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Read or Write access: | W | R | R | R | W | R | R | R | W | R | R | R | W | R | R | W | R | R | R | W |
| Frame 0 | ▶$7^M_R$ | $7^M_R$ | $7^M_R$ | ▶$7^M_R$ | $7^{MM}_R$ | ▶$7^M$ | $7^M$ | ▶$4_R$ | $4_R$ | ▶$3_R$ | 3 | $3_R$ | $3_R$ | ▶$1_R$ | $1_R$ | 1 | $1_R$ | $1_R$ | $1_R$ | $1^M_R$ |
| Frame 1 | | ▶$0_R$ | $0_R$ | ▶$2_R$ | $2_R$ | ▶$3_R$ | $3_R$ | $3_R$ | ▶$2^M_R$ | $2^{MM}_R$ | ▶$2^M$ | ▶$2^M$ | ▶$2^M_R$ | ▶$2^M_R$ | ▶$2^{MM}_R$ | ▶$2^M$ | ▶$2^M$ | ▶$7_R$ | $7_R$ | $7_R$ |
| Frame 2 | | | ▶$1_R$ | $1_R$ | ▶$0^{MM}_R$ | $0^M$ | ▶$0^M_R$ | $0^M_R$ | $0^M_R$▶ | $0^{MM}_R$ | $0^M_R$ | $0^M_R$ | $0^M_R$ | $0^M_R$ | $0^{MM}_R$ | $0^M_R$ | $0^M_R$ | $0^M_R$ | ▶$0^M_R$ | ▶$0^M_R$▶ |

No page fault, however R bit of page gets set again by PMMU due to reference.

Σ = 11 page faults

# Page Replacement Algorithms: Enhanced Second-Chance Policy

- Advantage:
  - Having four different classes is already a better approximation of LRU than just the two of the non-enhanced Second-Chance policy.
  - Furthermore, it is implementable using just the features (reference bit and modified bit) that are provided by all modern PMMUs.

- Disadvantage:
  - Periodic resetting the reference bits of all page table entries may be too time consuming (in practise, page tables may be pretty huge).
    - This is the probably the reason why major OS use rather Second-Chance.
  - Distinguishing four different classes may still not be sufficient for approximating LRU as much as possible. (Improvement: next slide)

# Belady’s Anomaly

- Paradox: It may be the case that FIFO results in more page faults when you increase the number of available frames!
- Example:
  - 3 frames:

|  |  | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Youngest page |  | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 4 | 4 | 2 | 3 | 3 |  |
|  |  |  | 0 | 1 | 2 | 3 | 0 | 1 | 1 | 1 | 4 | 2 | 2 |  |
| Oldest page |  |  |  | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 1 | 4 | 4 |  |
|  |  | P | P | P | P | P | P | P |  |  | P | P |  | 9 Page faults |

(a)

  - 4 frames:

|  |  | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Youngest page |  | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |  |
|  |  |  | 0 | 1 | 2 | 2 | 2 | 3 | 4 | 0 | 1 | 2 | 3 |  |
| Oldest page |  |  |  | 0 | 1 | 1 | 1 | 2 | 3 | 4 | 0 | 1 | 2 |  |
|  |  |  |  |  | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 |  |
|  |  | P | P | P | P |  |  | P | P | P | P | P | P | 10 Page faults |

(b)

- So called stack algorithms do not suffer from Belady’s anomaly.
  - Stack algorithm: set of pages in memory for *n* frames is always a subset of the set of pages for *n+1* frames.
  - (OPT and LRU are stack algorithms.)

# 9.6 Memory Compression Instead of Paging Out to Storage Device

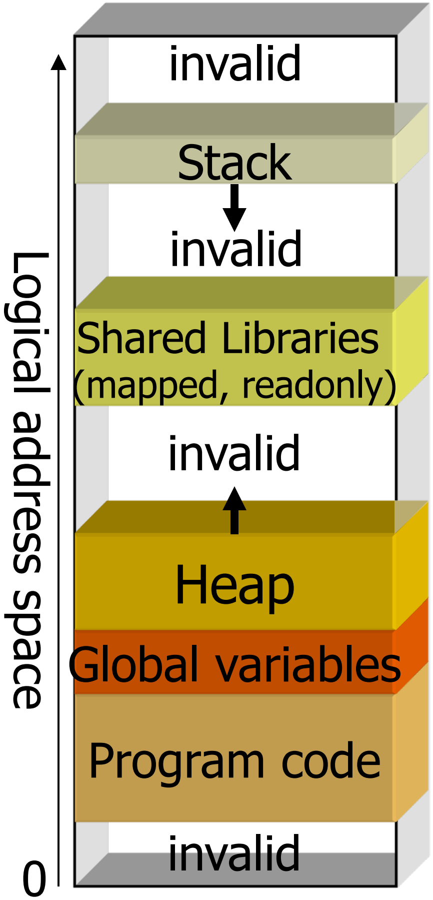
- Mobile devices typically do not page out pages to a storage device.
  - Still, the may run out of physical memory. Approaches to deal with:
  - Terminate processes (but allow a process to store it’s status, so that it can later be re-started).
  - Compress memory:
    - For those pages that would be candidates for paging out to storage device:
      - Compress contents of these pages using compression algorithm.
        - Similar to ZIP, for file compression. E.g. Microsoft's Xpress and Apple's WKdm: reduction to 30%-50% of original page size.
        - In average: 2-3 compressed pages fit into one frame, making thus 1-2 frames available.
- While compression needs CPU time, typically still faster than paging out to SSD storage.
  - Even non-mobile OSes use nowadays memory compression.
  - Only if memory compression is not enough, page out to storage device.

# 9.7 Advanced Applications of Virtual Memory: Demand Loading vs. Prepaging

- As the infrastructure of demand paging is anyway available, demand loading becomes possible:
  - When a process is started, do not load whole binary file containing instructions into memory, but just mark initially all pages as invalid.
    - At a page fault, load the according instructions for that page into memory.
  - Advantage: no unnecessary loading of instructions that might never get executed.
  - Disadvantage: resulting page faults lead to an overhead.
- Prepaging (just the opposite of demand loading):
  - Load all instructions into memory to avoid page faults.
  - Advantage: reduced number of page faults.
  - Disadvantage: unnecessary loading of instructions may occur.
- In practise, a compromise is used, i.e. the first *n* pages are prepaged and at a page fault, multiple consecutive pages (e.g. current working set size) are loaded.

# Advanced Applications of Virtual Memory: Growing Heap & Stack

- Without virtual memory, reserving the right amount of memory for stack and heap is difficult:
  - Too small: stack or heap overflow possible,
  - Too huge: memory is wasted.
- With virtual memory, we can just reserve the whole logical address space for a process and reserve a big amount of it for stack and heap.
  - ⇒ While sufficient space for stack and heap is reserved in the logical address space, use only as much physical frames as currently required.
    - As stack and heap increase, just more pages are actually used.
    - Overwriting shared libraries by stack or heap impossible as shared libraries serving as a sentinel or buffer in-between are read-only.
- Management of heap space:
  - While paging avoids external fragmentation, internal fragmentation may still occur within the heap of a process: when releasing allocated heap space, holes in the logical address space of the heap occur.



# Advanced Applications of Virtual Memory: Fork Using Copy-on-Write/Lazy-Copy

- Reminder: at fork, the child process gets an exact copy of the address space of the parent.
  - Side note: This becomes only possible using a programmable MMU:
    - The child’s copy will be located at a different physical address. However, the child will also get a copy of all the parent’s address references. These remain only valid, if a programmable MMU can be used to map the different physical addresses of the copy for the child to the same logical addresses that the parent process used.
- However, copying the physical memory of the parent is slow.

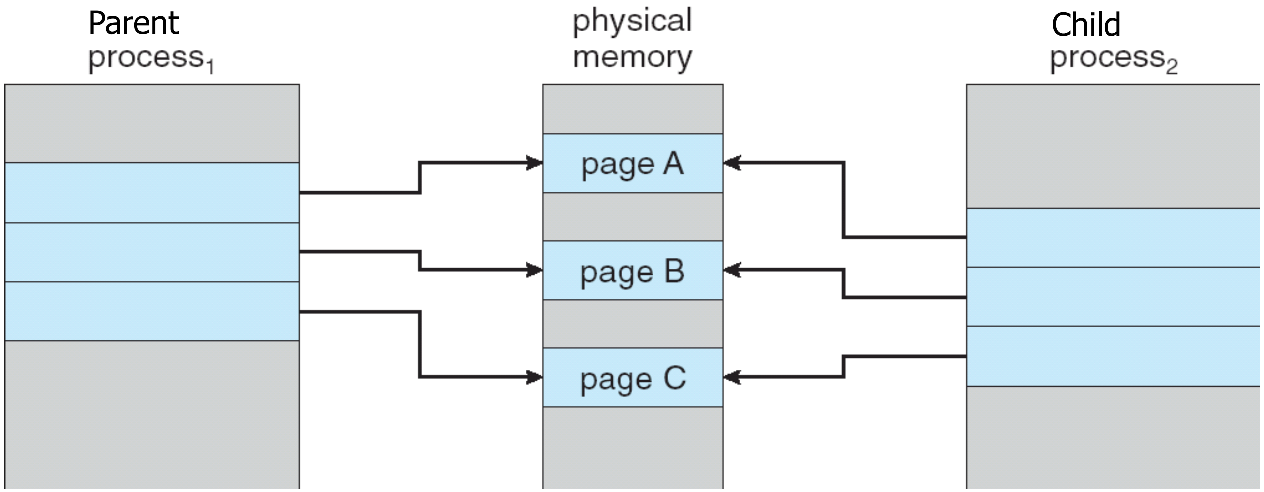
⇒ Faster: just map frames (instead of copying) of parent containing instructions and data into address space of child (=shared memory).

  - However, when parent or child modifies its address space, the copy of the other party must not be modified!

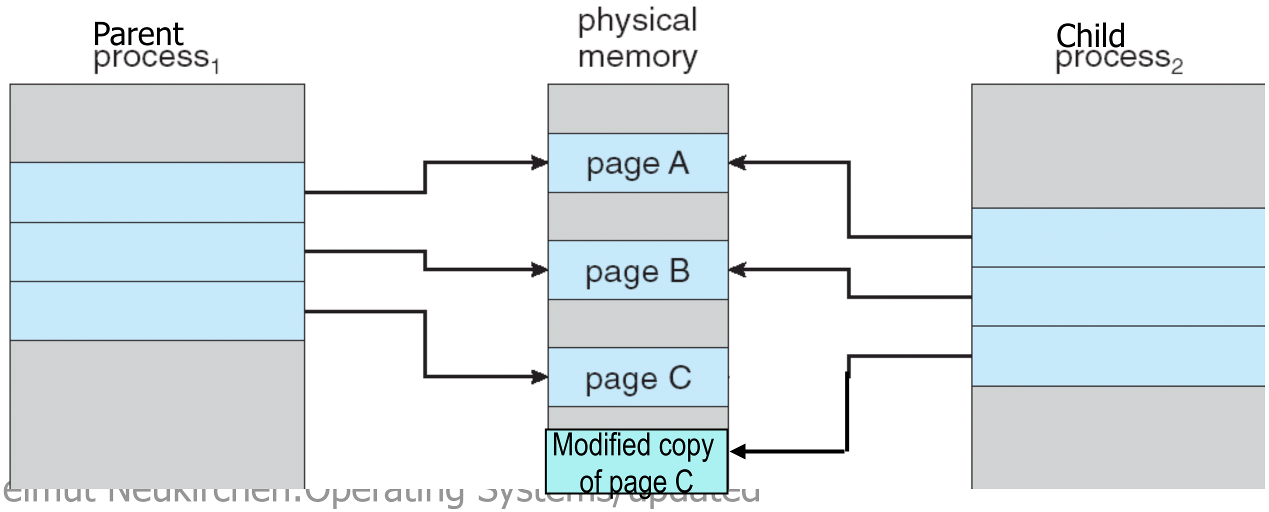
  ⇒ Mark shared pages as write-protected in page table entry: as soon as parent or child modify data, page fault interrupt occurs. Only then, just these frames are physically copied. (“Copy-on-Write”/“Lazy-Copy”)

# Advanced Applications of Virtual Memory: Fork Using Copy-on-Write/Lazy-Copy

- Example:
  - Initially, pages are shared between child and parent process.



  - After child (or parent) tries to modify page C, a copy of page C is created and the modifying CPU instruction is restarted.



# 9.9 Summary

- Virtual memory allows to execute processes whose logical address space is larger than physically available memory.
  - Allows to run extremely large processes and to increase degree of multiprogramming beyond physically available memory.
  - Typically implemented using demand paging.
    - Single pages are transferred between physical memory and hard disk.
      - Significantly faster than swapping of whole processes.
      - Still system may be busy doing disk transfers when thrashing occurs.
  - Different page replacement algorithms perform differently.
    - While the Optimal policy cannot be implemented and others involve too much overhead, Second-Chance is used in today's operating systems.
  - Instead of paging out to storage device: try to compress pages.
  - From speed point of view: have enough physical RAM to avoid paging!
  - Virtual memory enables other advanced applications, e.g. fast forking using copy-on-write/lazy-copy.
- Kernel memory managed separately: buddy system, slab allocation.