Virtual Memory

Frédéric Haziza <daz@it.uu.se>

Department of Computer Systems
Uppsala University

Spring 2007

Background

Paging & Segmentation

Keep many processes in memory simultaneously
to allow multiprogramming

Virtual Memory

Not the whole process in main memory

Background

Requirement: Executed instruction must be in MEM

A solution: Place the whole logical address space in MEM

(Help: dynamic loading)

Problem: limits programs to size of physical MEM

But programs not entirely needed.

(Ex: error routines, array allocated too large)

Even if entirely needed, maybe not at the same time...

So....

Allow programs to be partially in memory

Advantages

1. Prg can be bigger than phys MEM.
   Prg are written for large virtual mem space
2. Each prg takes less phys MEM
   $\Rightarrow$ more prg in MEM
   $\Rightarrow$ more competition for CPU scheduling
   $\Rightarrow$ increase CPU utilization and throughput.
   (But no increase in response time or turnaround time)
3. less IO needed to swap in or out
   ($\Rightarrow$ faster for each user prg)

Benefits both USER and SYSTEM.

Outline

Common solution: Demand Paging

If we don’t have all the pages at once:

- How to allocate pages?
- Whom to replace, in case MEM is “full”
- Any problems?

Demand Paging

Main method: Lazy swapping

Bring the pages in, only when needed.

- Begin with the page containing the start PC
- Page already in memory? Use it
- Page referenced but not in memory? Bring it into memory
- etc...

Implementation

Idea

Use the valid/invalid bit to distinguish between pages in MEM and pages on disk.

Valid: both

- Legal
- In memory

Invalid: either

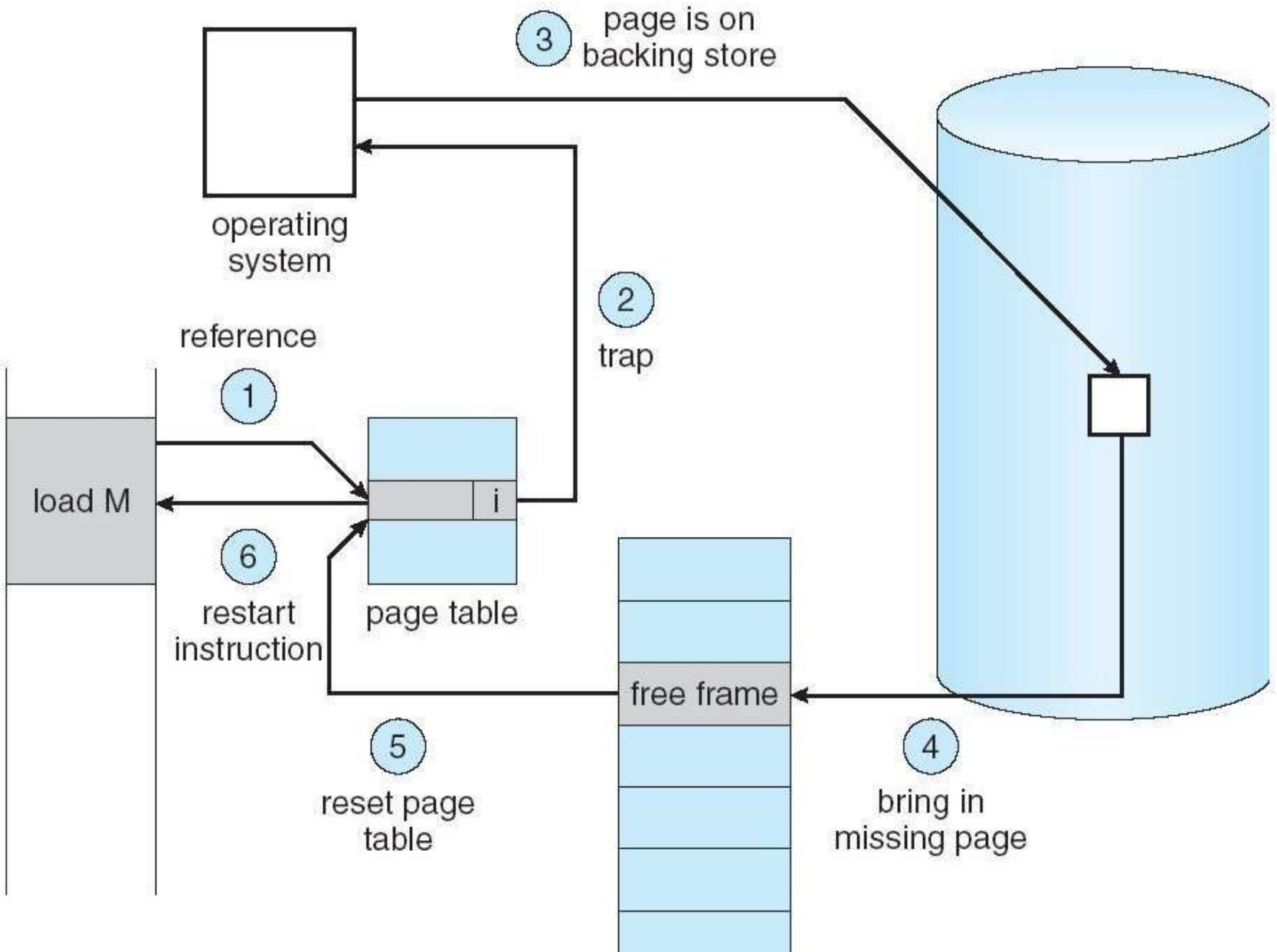
- Not Legal (not in the logical address space of that process)
- Valid but currently not in memory

Page-Fault

Access to invalid pages causes a page-fault trap

- check if the page exists (else give addressing error)
- find a free frame for the page:
  if none, page out some other page
  $\Rightarrow$ page replacement
- read the page from disk (move to device queue)
- when page read, update page table
- when process scheduled, restart the instruction

Page-Fault


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page

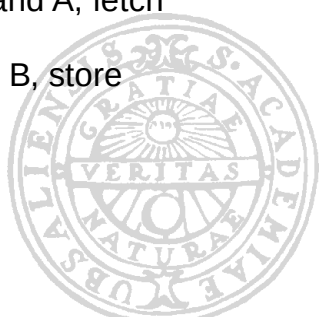
Performance

Restart the instruction... Problem?

Example ($C = A + B$)

1. Fetch and decode the instruction (ADD)
2. Fetch A (load mem location in register)
3. Fetch B
4. Add A and B ($ADD\ R_1, R_2, R_3$)
5. Store the sum in C

Fetch instruction, decode it, fetch operand A,*...oops...page-fault...*fetch instruction, decode it, fetch operand A, fetch operand B,*...oops...page-fault...*fetch instruction, decode it, fetch operand A, fetch operand B, add A and B, store result in C,*...oops...page-fault...*fetch instruction, decode it...



Effective access time

Slow? Fast?

Effective access time

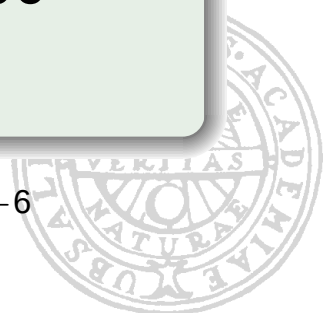
$= p * \text{page-fault-time} + (1 - p) * \text{mem-access-time}$

Example

200ns to access memory. 8ms to service a page-fault.

$$\begin{aligned} \textit{effective access time} &= p * (8ms)(1 - p) * (200ns) \\ &= p * 8'000'000 + (1 - p) * 200 \\ &= 7'999'800 * p + 200 \end{aligned}$$

Max 10% overhead? $220 > 200 + 7'999'800 * p \quad \Rightarrow p < 2.5 * 10^{-6}$



Effective access time

Effective access time

$= p * \text{page-fault-time} + (1 - p) * \text{mem-access-time}$

Minimize p.

Minimize page-fault-time ?

Swap space faster than *file space* (Bigger blocks, no file lookups, no indirect allocation)

- copy whole program file to swap space at start
- bring from file space, but update back pages in swap space

2 major problems to issue

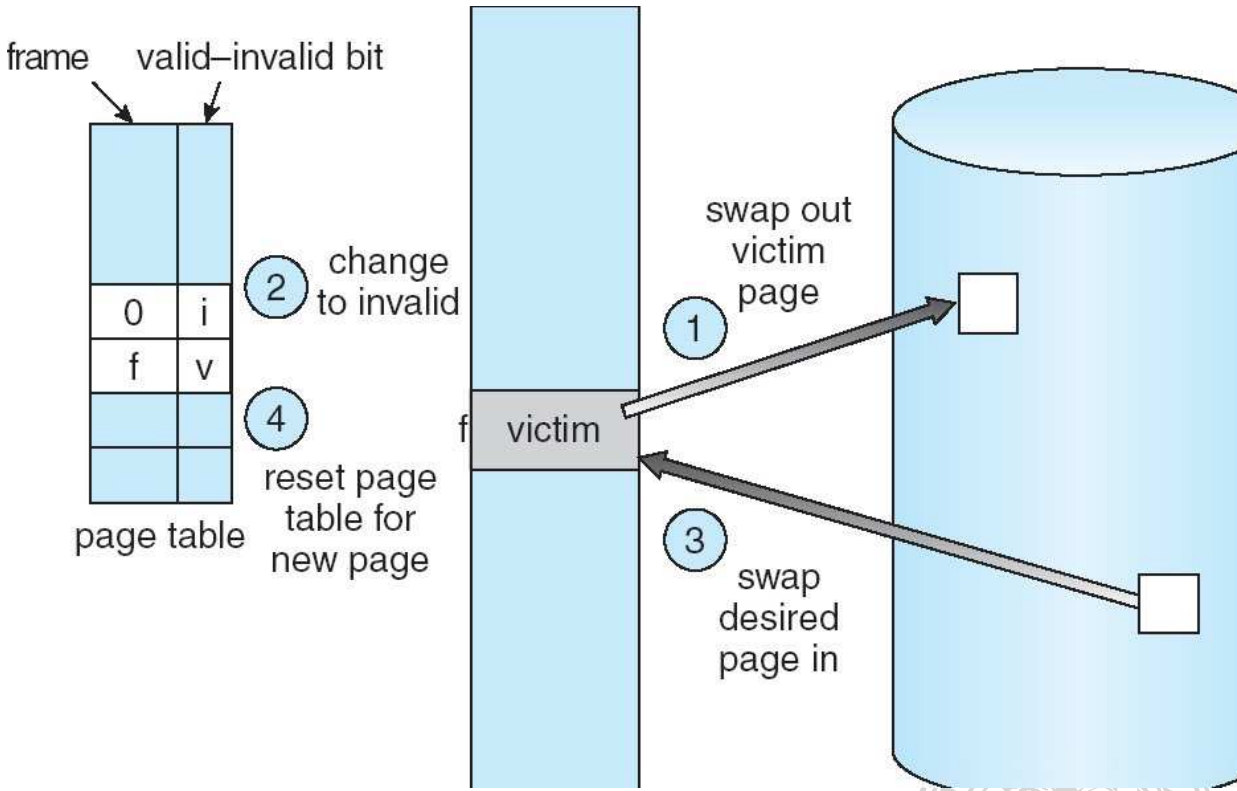
- Page replacement (Who's the victim?)
- Frame allocation (How many frames per process?)

Page Replacement

At a page-fault, if no frame is free:

1. Find a victim
2. Save the victim to swap space (*)
3. Read the new page to the freed frame
4. Restart instruction

(*) Can skip using a modify bit (or dirty bit) (in page table)



Page Replacement Algorithms - Evaluation

Goal

Keep the page-fault rate low

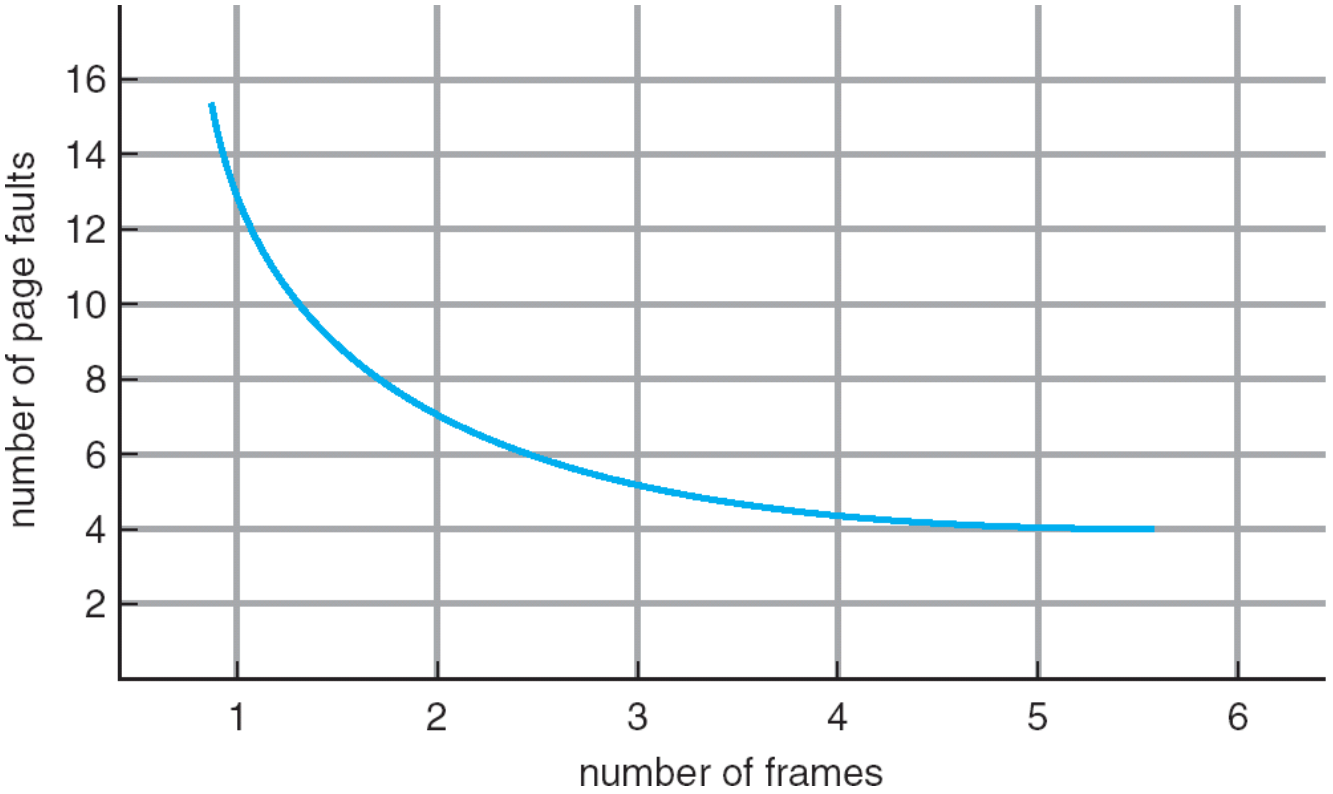
Evaluation with reference strings
For a given page size, just consider the frame numbers.

Example

0100,0423,0101,0612,0102,0103,0104,0101,0611,0102,0103,
0104,0101,0610,0102,0103,0104,0101,0609,0102,0105

at 100 bytes per page, reduces to
1,4,1,6,1,6,1,6,1,6,1

No surprise



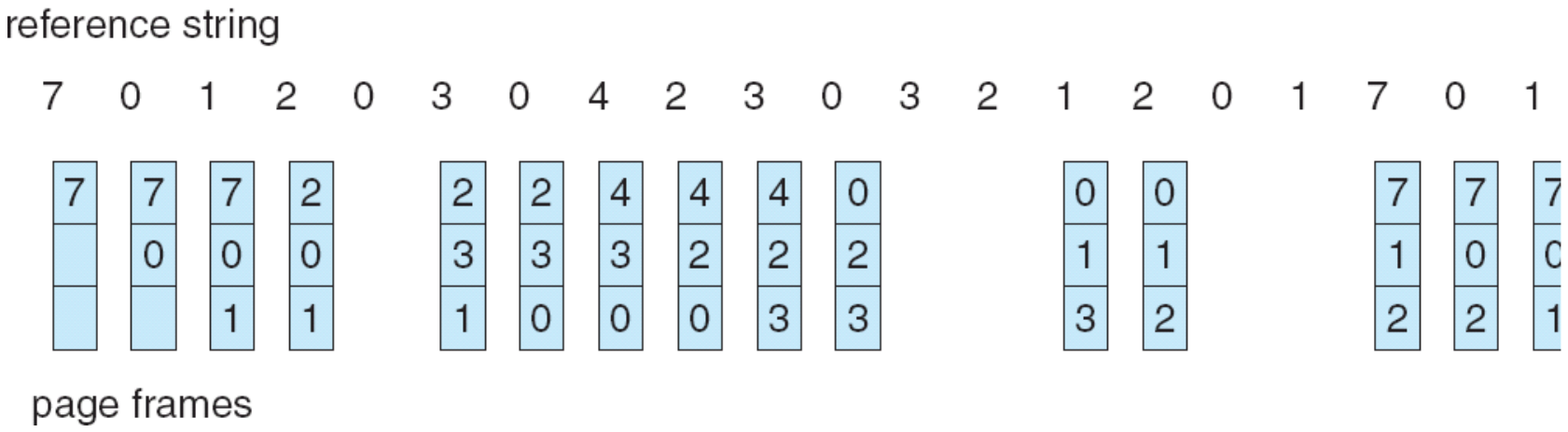
Keyword: LOCALITY

Page Replacement Algorithms

- FIFO replacement
- Optimal algorithm (OPT or MIN)
- LRU: Least-Recently Used
- LRU approximations

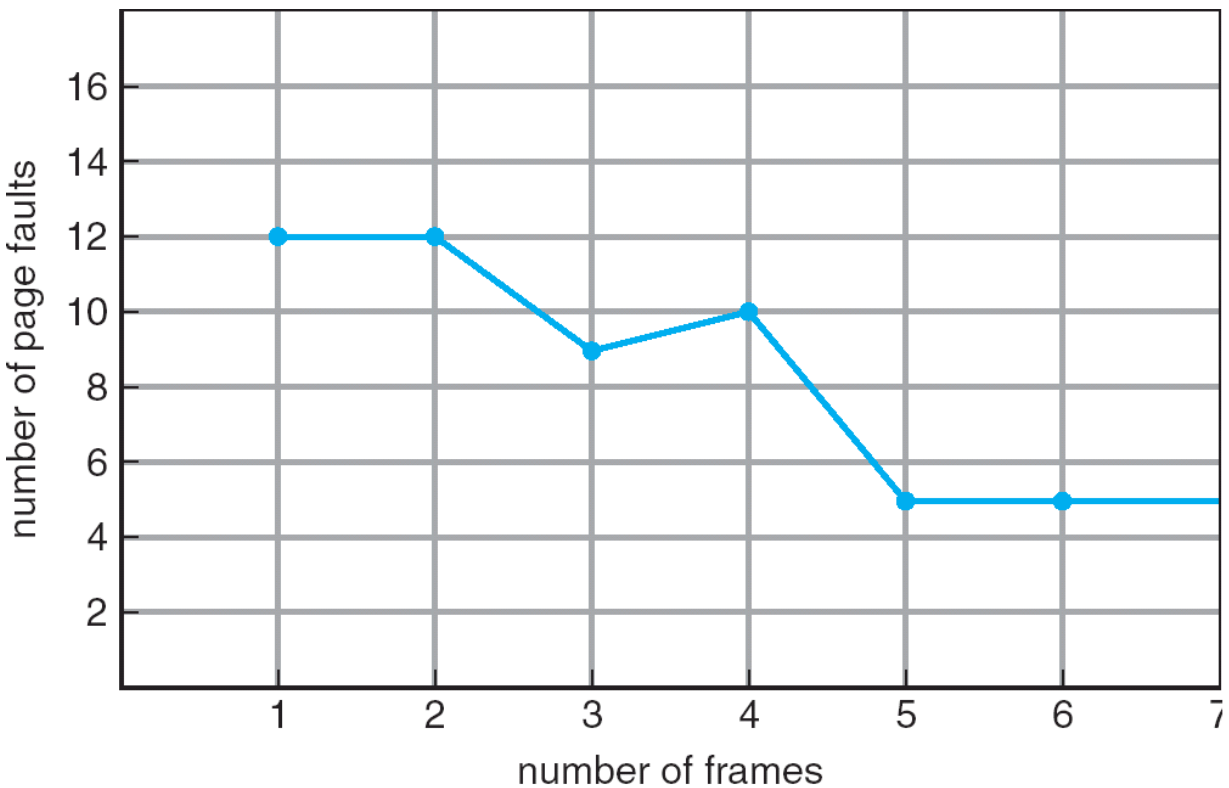
reference string
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

FIFO Page Replacement



Belady's anomaly

1,2,3,4,1,2,5,1,2,3,4,5



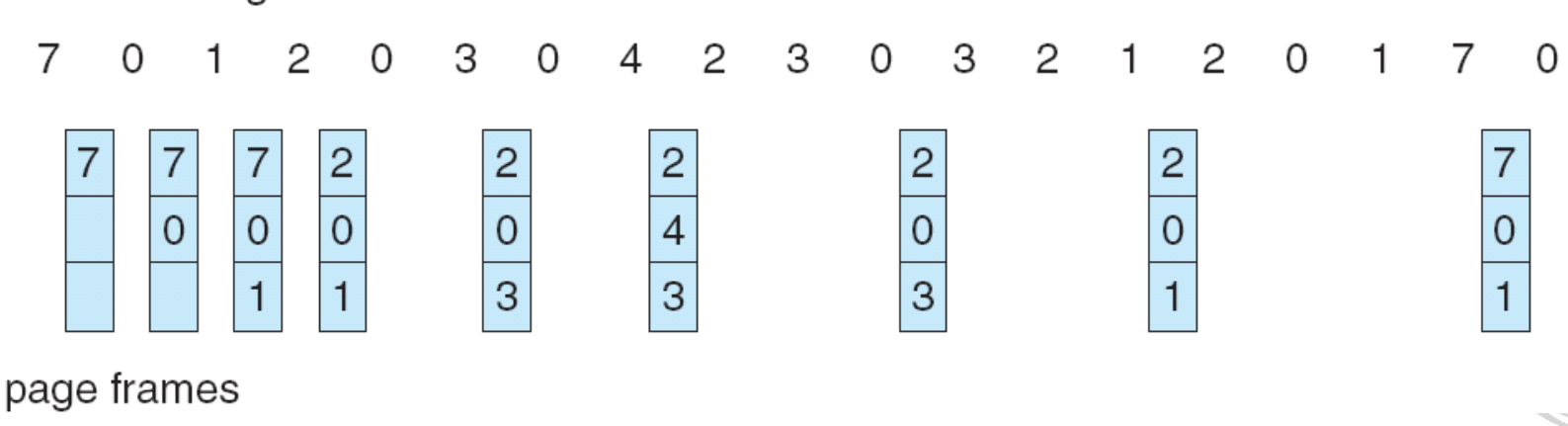
Optimal Page Replacement Algorithm

- Lowest page-fault rate
- No Belady's anomaly

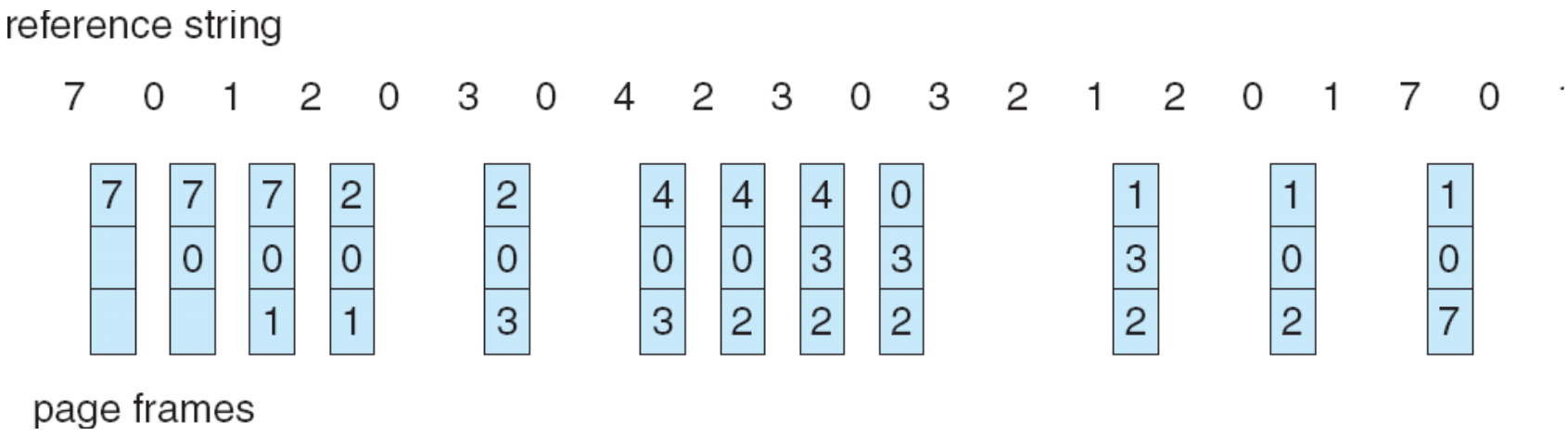
But difficult to implement

(because need for future knowledge of reference string. Cf STF)

Comparison: New algo not optimal but good enough: within 12.3% at worst and 4.7% on average



LRU: Least Recently Used



LRU: Least Recently Used - *How?*

Keep track of when frames are used: hardware support

- Clock/Counter: each frame's *clock* field is updated when the page is referenced. The *clock* gets incremented at each memory reference.
  Problems:
  - clock overflow
  - search page table for least recently used
  - write to page table on each reference ...
- Stack: when a frame is referenced, put it on the top of the stack. Replace the frame at the bottom of the stack (use double-linked list).
  No searching for victim, but update of links necessary.

Every memory references....$\Rightarrow$ Done in TLB

LRU: Least Recently Used - Approximation

- Reference bit(s)
- Second-Chance algorithm
- Reference counting

LRU-Approximation: Reference bit(s)

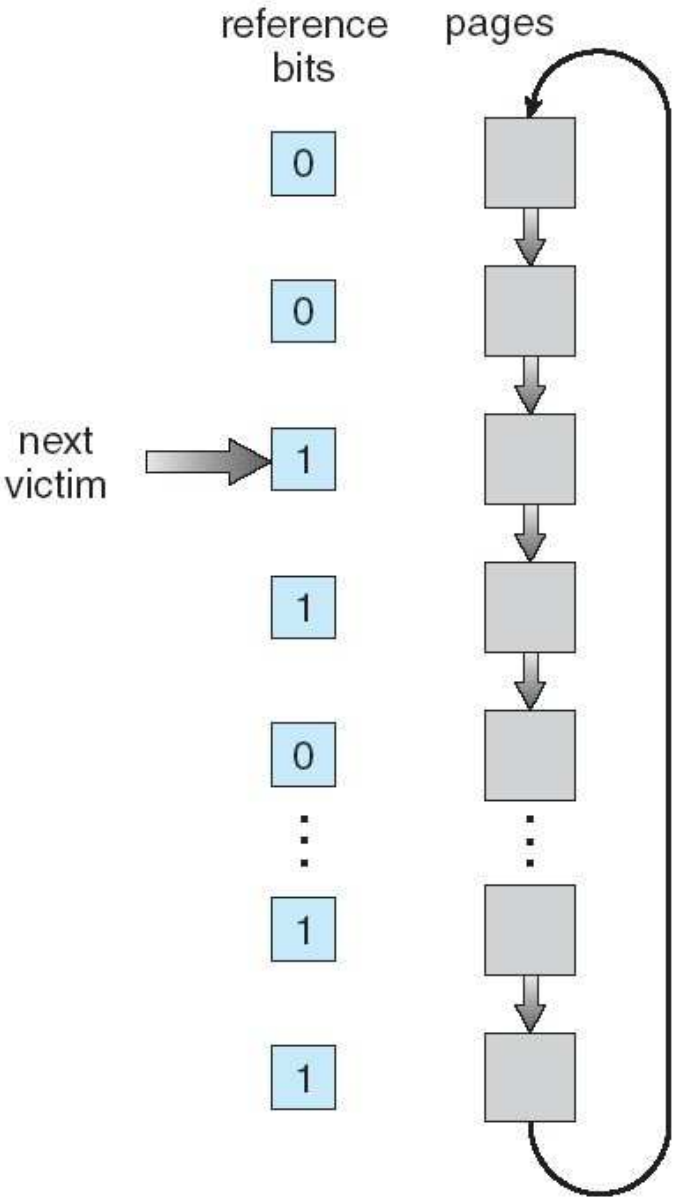
One single bit:

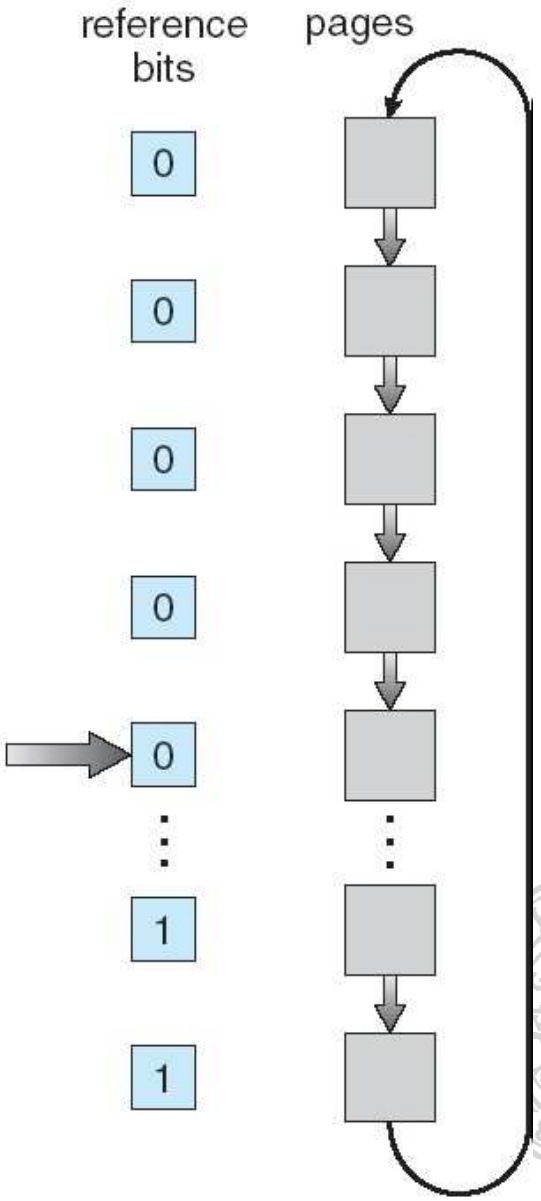
1. set the reference bit when page is referenced
2. clear the bits periodically
3. select the first non-marked (not used frequently) as victim.

More reference bits

1. shift right all bits periodically
2. set the most significant when referenced (approximates a clock)
3. select the lowest value as victim.

Second-Chance Algorithm (FIFO + One reference bit)





Reference counting

Keep track of the number of references (not *when* the reference was made)

- Least Frequently Used (LFU)
  An active page is referenced often: High count.
- Most Frequently Used (MFU)
  The smallest counter has just arrived and will be referenced again soon.

Neither is a good approximation of OPT,
and both are expensive.

Decreasing Page-Fault handling time

Goal

Low page-fault rate

- Keep a pool of free frames, page in to one while paging out the victim: wait for the I/O "in parallel".
- Keep track of which pages are in the free frames, and if one needs to be paged in, no I/O needed.
- If paging disk is idle, write out dirty frames in background (and mark them "clean"), so it's already done when needed.

How many frames?

How to divide frames between processes? How many per processes? Equal? Not equal?

Max? $\Rightarrow$ total number of available frames
Min? $\Rightarrow$ seek performance $\Rightarrow$ Low page-fault rate

Restart the instruction after page-fault

Minimum number of frames

Need enough frames to hold the different pages that an instruction can reference, including indirect addressing, etc...

$\Rightarrow$ Depends on the computer architecture

Principles

- Equal share: $\frac{\#\text{frames}}{\#\text{processes}}$
  but different processes have different needs (not equal sizes)
- Proportional: use relative size: $\frac{s_i}{\sum s_i} * \#\text{frames}$
  but consider process priorities too!

Allocation - How to choose?

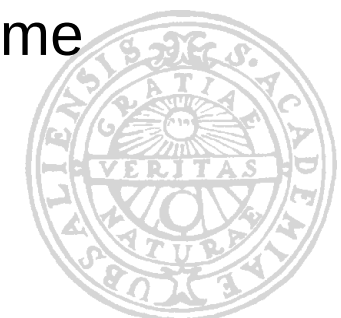
From whom to choose?

- Global replacement: replace any page
  (including those allocated to other processes)
- Local replacement: only replace pages of the process itself

Global replacement: page-fault behavior of one process

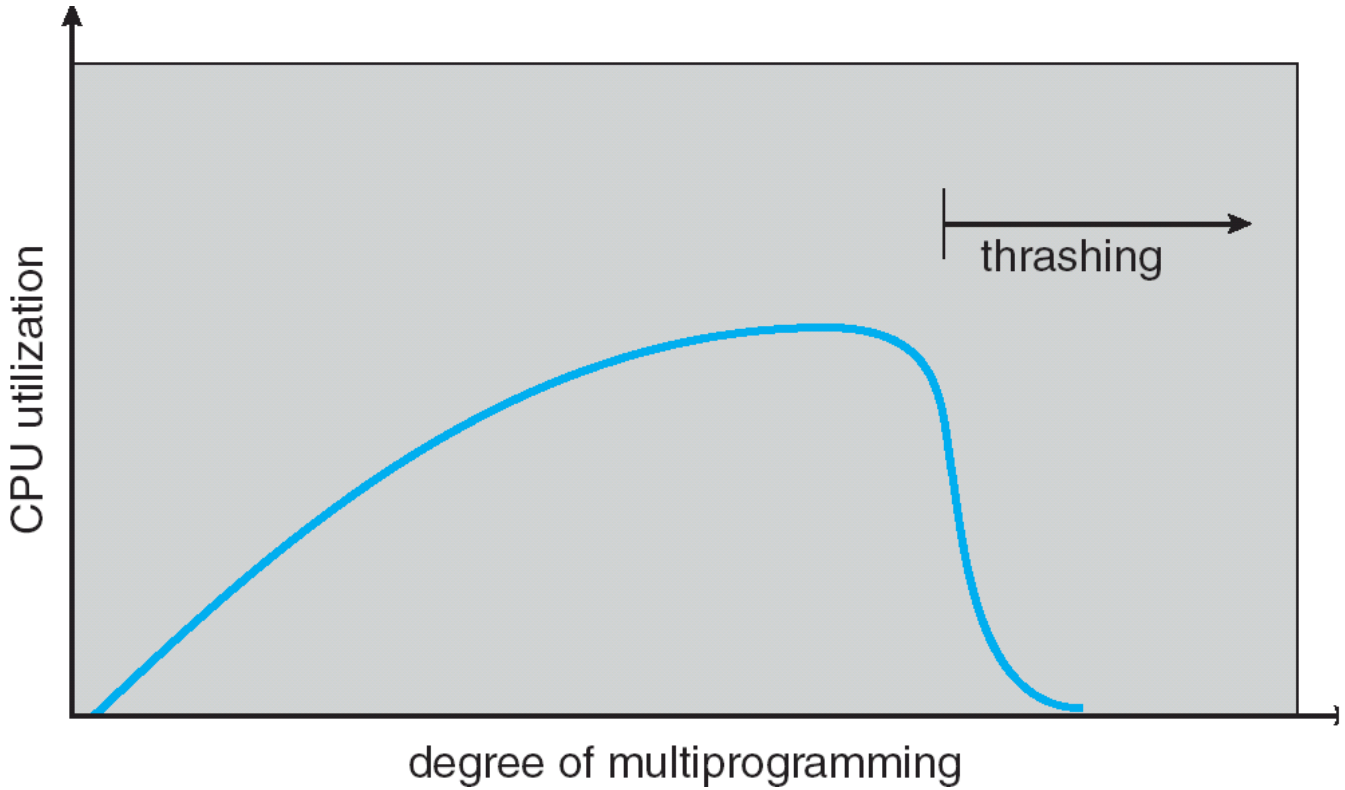
- affects all the others (by "stealing" frames from others)
- is affected by other processes too (symmetrically)

Local replacement: more “fair”, but may lead to lower frame utilization ($\Rightarrow$ lower CPU utilization)



Thrashing

More time is spent paging (=handling page faults) than executing programs



Thrashing - Brighter sky

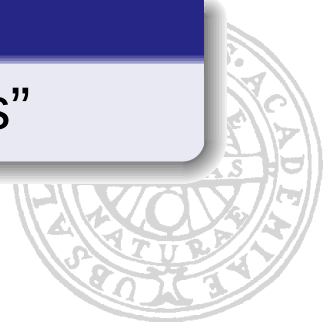
Local replacement helps by *not* spreading thrashing from one process to the others

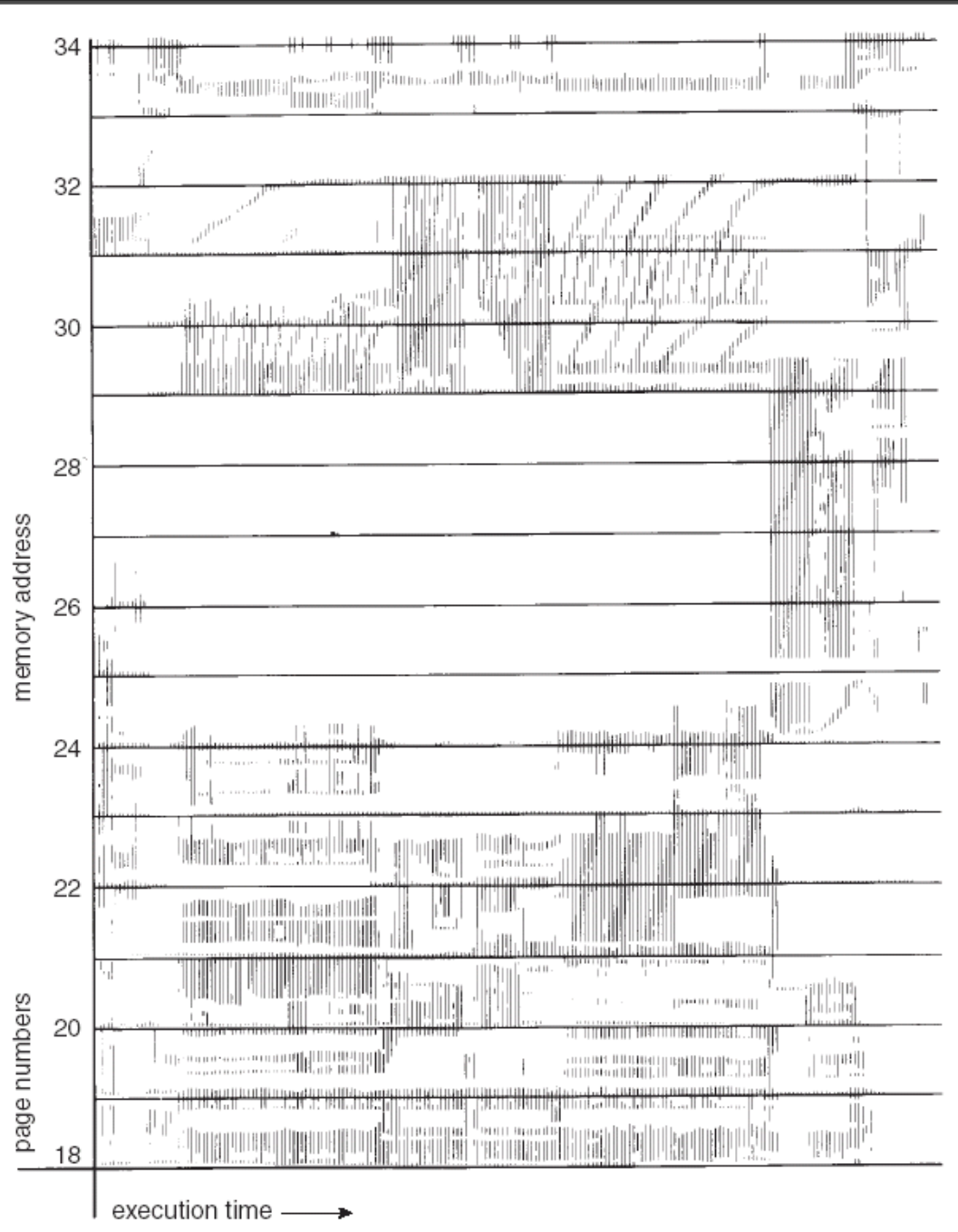
but

the thrashing process increases the load on the paging disk
$\Rightarrow$ longer access times for other processes.

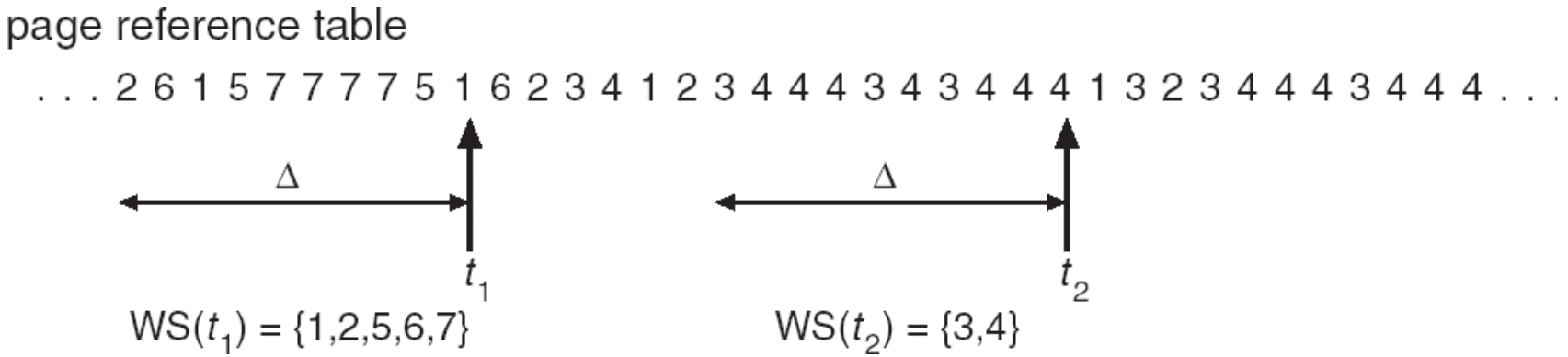
Preventing thrashing

Must provide a process with as many frames as it “needs”




34
32
30
28
26
24
22
20
18
memory address
page numbers
execution time

Working-set model



Page-fault frequency