COP 4600 – Summer 2012

Introduction To Operating Systems

Memory Management – Part 4

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2012

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Operating System Software

- Algorithms employed for various aspects of the memory management unit.

Fetch Policy Demand Pre-paging	**Resident Set Management** Resident set size Fixed Variable Replacement Scope Global Local
Placement Policy	**Cleaning Policy** Demand Pre-cleaning
Replacement Policy Basic Algorithms Optimal LRU FIFO Clock Page Buffering	**Load Control** Degree of Multi-programming

Design Considerations – More Details

- The primary issue is one of performance: we need to minimize the rate at which page faults occur due to the overhead involved with handling a page fault.
  - At a minimum the overhead includes deciding which resident page or pages to replace, and the I/O of exchanging the pages. Also the OS must schedule another process to run during the page I/O, causing a process switch.
- Accordingly, we'd like to arrange matters so that, during the time that a process is executing, the probability of referencing a word on a missing page is minimized.
- There will be no overall best policy that will cover all occasions.

Design Considerations – More Details

- The task of memory management in a paging environment is fiendishly complex.
- Furthermore, the performance of any particular set of policies depends on main memory size, the relative speed of main and secondary memory, the size and number of processes competing for resources, and the execution behavior of individual programs.
- Smaller OS designers should pick a set of policies that will provide "good" behavior over a wide range of conditions. Larger OS should be equipped with monitoring and control tools to tune the OS based on site conditions.

Fetch Policy

- The fetch policy determines when a page should be brought into main memory.
- The two common alternatives are demand paging and prepaging.
- With demand paging, a page is brought into memory only when a reference is made to a location on that page.
- With prepaging, pages other than the one demanded by a page fault are brought into main memory. This technique attempts to take advantage of secondary memory characteristics. For example, if the pages of a process are stored contiguously in secondary memory, then it is more efficient to bring in a number of contiguous pages at one time as opposed to bringing them in one at a time over an extended period.

Placement Policy

- The placement policy determines where in main memory (real memory) a process piece is to reside.
  - Recall that in a pure segmentation system, the placement policy algorithms of best-fit, first-fit, and next-fit.
- In most modern systems, the placement policy is not a huge concern due to the flexibility of the address translation hardware and main memory access hardware since any page-frame can be accessed with equal efficiency.
- However, in a NUMA (Non-Uniform Memory Access) multiprocessor, the distributed shared memory of the machine can be shared by any processor on the machine, but the time for accessing a particular physical location will vary with the distance between the processor and the memory module. Thus, performance depends heavily on the placement policy.

Resident Set Management and Replacement Policy

- The replacement policy involves several interrelated concepts.
  - How many page frames are to be allocated to each active process.
  - Whether the set of pages to be considered for replacement should be limited to those of the process that caused the page fault or encompass all the page frames in main memory.
  - Among the set of pages to be considered, which particular page should be selected for replacement.
- The first two concepts are resident set management, and the third concept is the replacement policy.
- Resident set management deals with frame allocation and replacement scope.
- Replacement policy deals with the selection of the page to replace.

Resident Set Management

- Deciding how many page frames to allocate to a process is dictated by the following concepts:
  - The smaller the amount of memory allocated to a single process, the more processes that can reside in the main memory at any one time, hence a higher degree of multiprogramming. This increases the probability that the OS will find at least one ready process at any given time and thus reduce the time lost to swapping.
  - If a relatively small number of pages of a process are in main memory, then despite the principle of locality, the rate of page faults will be rather high.
  - Beyond a certain size, additional allocation of main memory to a certain process will have no noticeable effect on the page fault rate for that process because of the principle of locality.

Resident Set Management

- These concepts lead to two different policies that are to be found in most modern operating systems.
- A fixed allocation policy gives a process a fixed (static) number of page frames within which it must execute.
  - The number of page frames allocated to a process does not change during its lifetime.
- A variable allocation policy gives a process a variable (dynamic) number of page frames within which it must execute.
  - A process that incurs a high fault rate will have its frame allocation increased at the expense of a process whose fault rate is much lower.

Replacement Scope

- The scope of a replacement strategy can be categorized as local or global.
- A local replacement policy chooses only among the resident pages of the process that generated the page fault in selecting a page to replace.
- A global replacement policy considers all unlocked pages in main memory in selecting the page to replace, regardless of which process owns a particular page.
- The frame allocation schemes combined with the replacement scope policies give rise to the various possibilities shown in the table on page 12.

Frame Locking

- One restriction on the replacement strategy is that some of the frames in main memory may be locked.
- When a frame is locked, the page currently in that frame many not be replaced.
- Much of the kernel of the OS is held on locked frames, as well as key OS control structures. In addition, I/O buffers, and other time-critical areas may be locked in main memory frames.

Allocation Policy And Replacement Scope

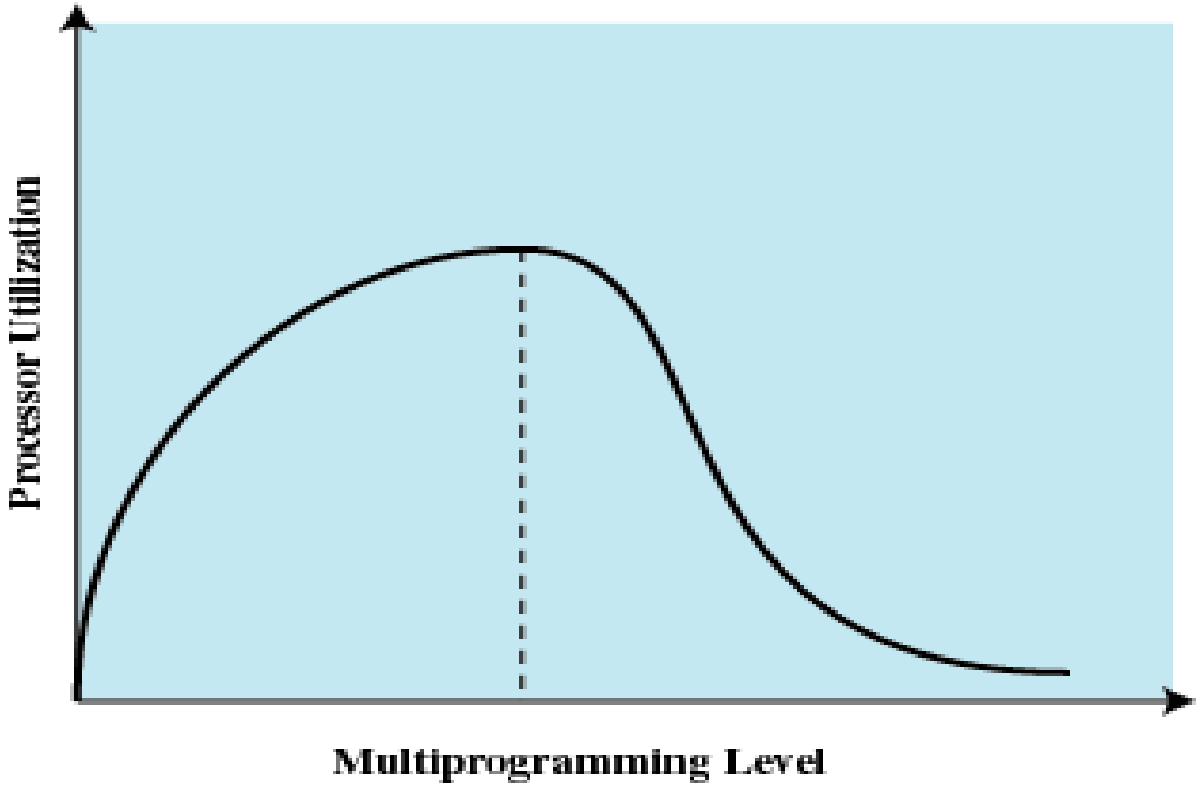
	Local Replacement	Global Replacement
Fixed Allocation	Number of frames allocated to a process is fixed. Page to be replaced is chosen from among the frames allocated to that process only.	Not possible
Global Allocation	The number of frames allocated to a process may vary during its lifetime (working set). Page to be replaced is chosen from among the frames allocate to that process only.	Page to be replaced is chosen from all available frames in main memory, this may cause the size of the resident set of processes to vary.

Cleaning Policy

- The cleaning policy is just the opposite of the fetch policy; its concerned with determining when a modified page should be written to secondary memory.
- Two common alternatives are demand cleaning and precleaning.
- Demand cleaning is when a page is written to secondary memory only when it has been selected for replacement.
- Precleaning will write a modified page to secondary memory before their page frames are needed so that writes can be batched.

Load Control

- Load control is concerned with determining the number of processes that can be resident in main memory, i.e., the degree of multiprogramming.



Basic Page Replacement Policies

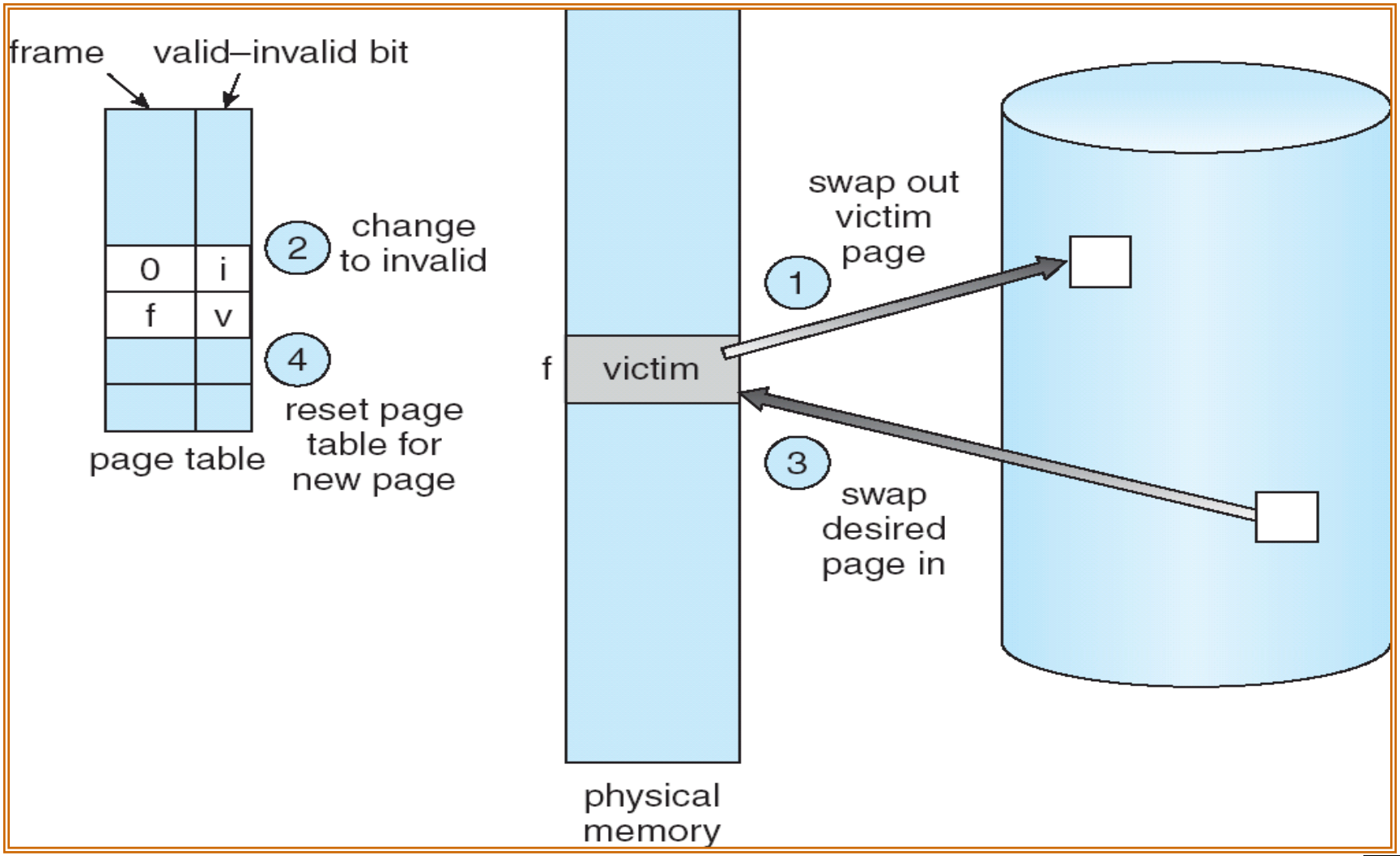
1. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

2. Bring the desired page into the (newly) freed frame; update the page and frame tables

3. Find the location of the desired page on disk

4. Restart the process

Page Replacement



Page Replacement Algorithms

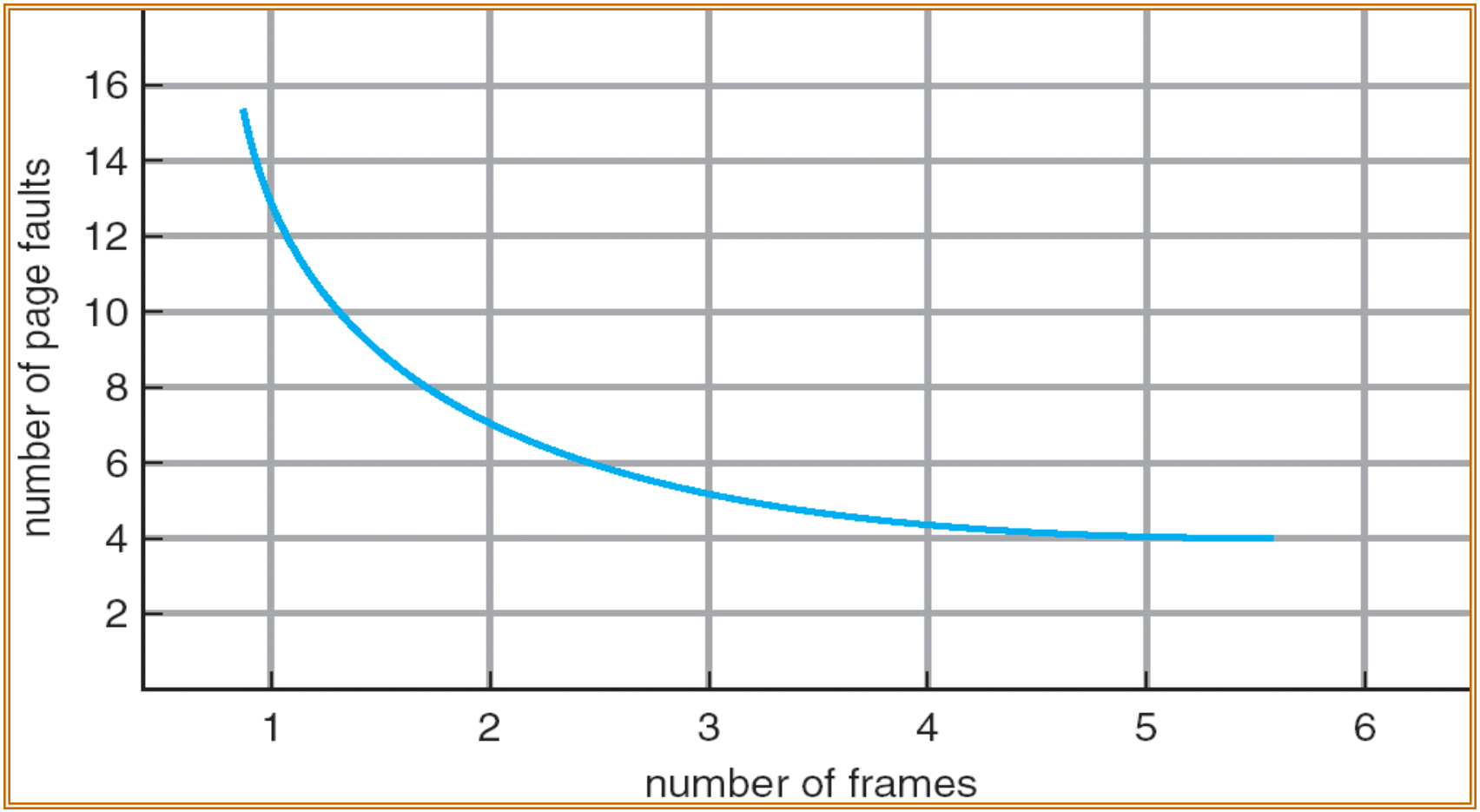
- Want lowest page-fault rate possible.

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

- In many of the examples that follow, the reference string is:

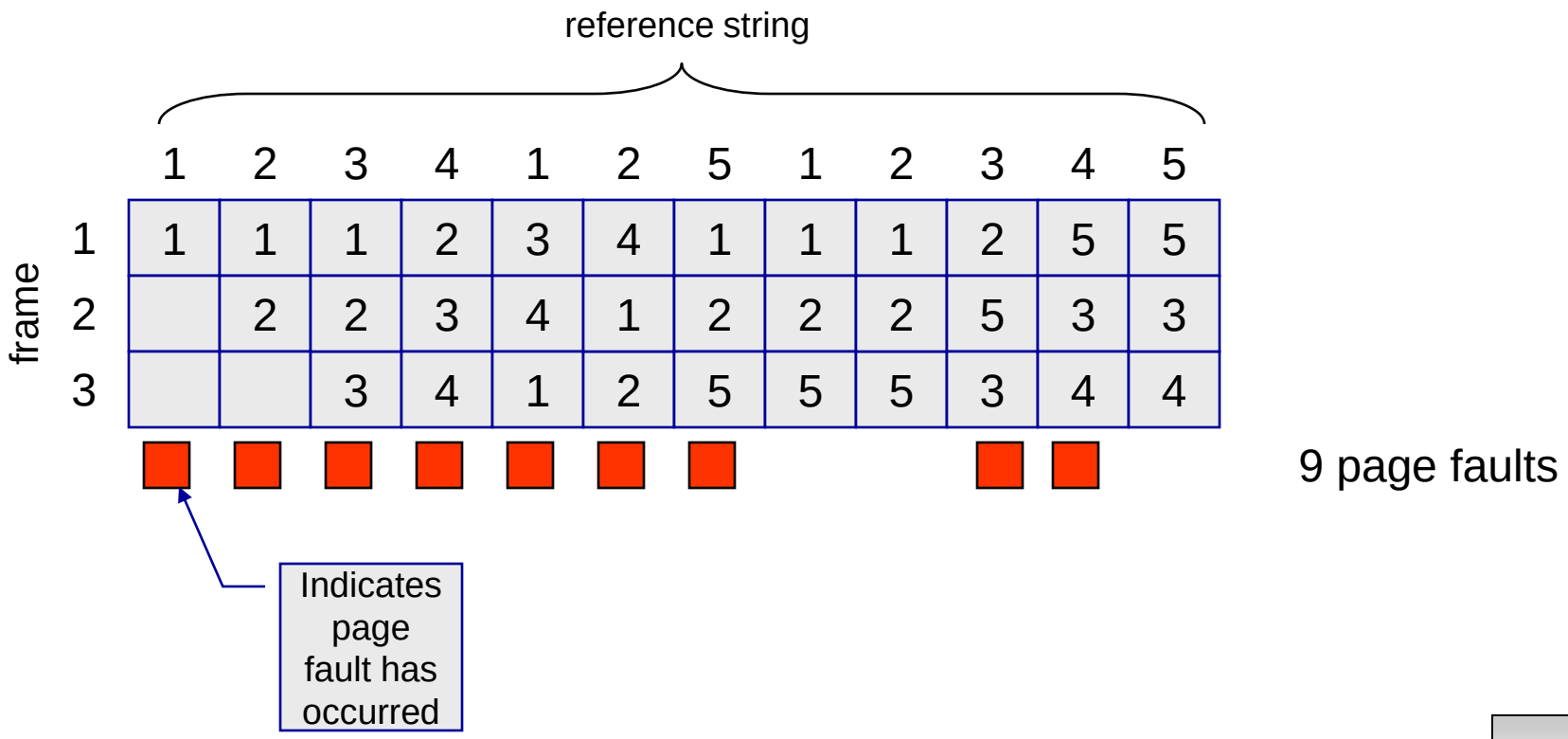
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



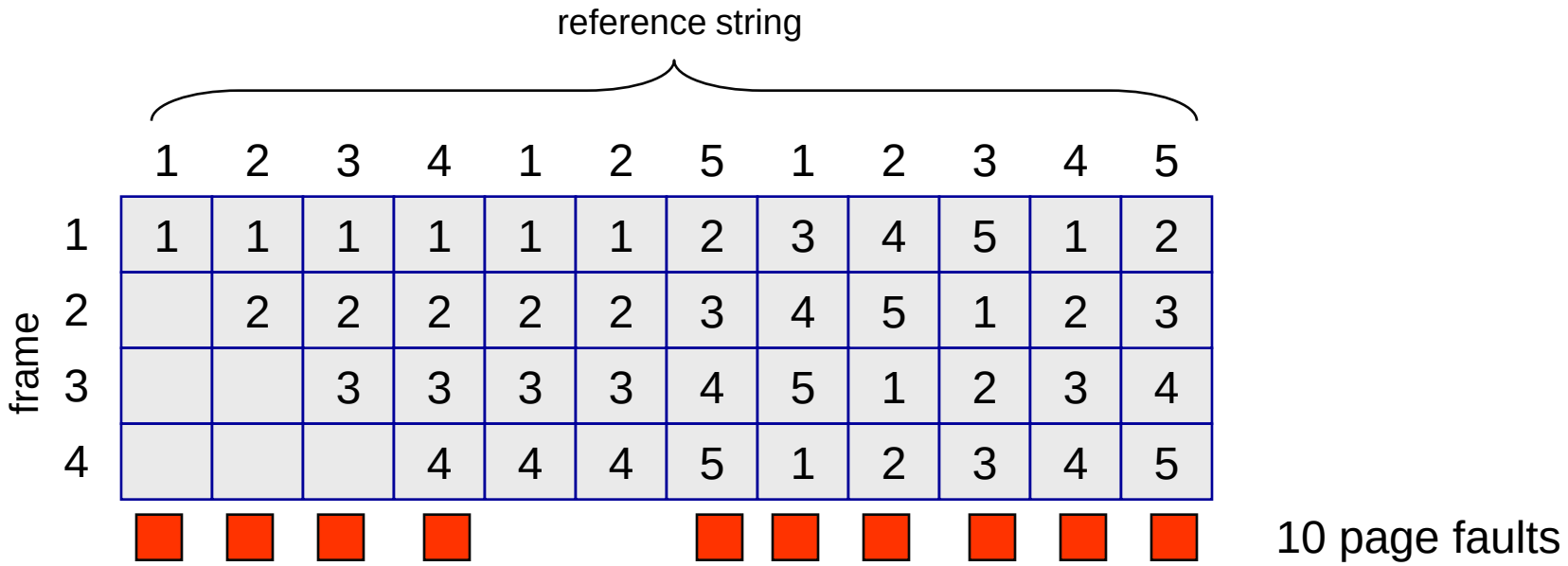
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



First-In-First-Out

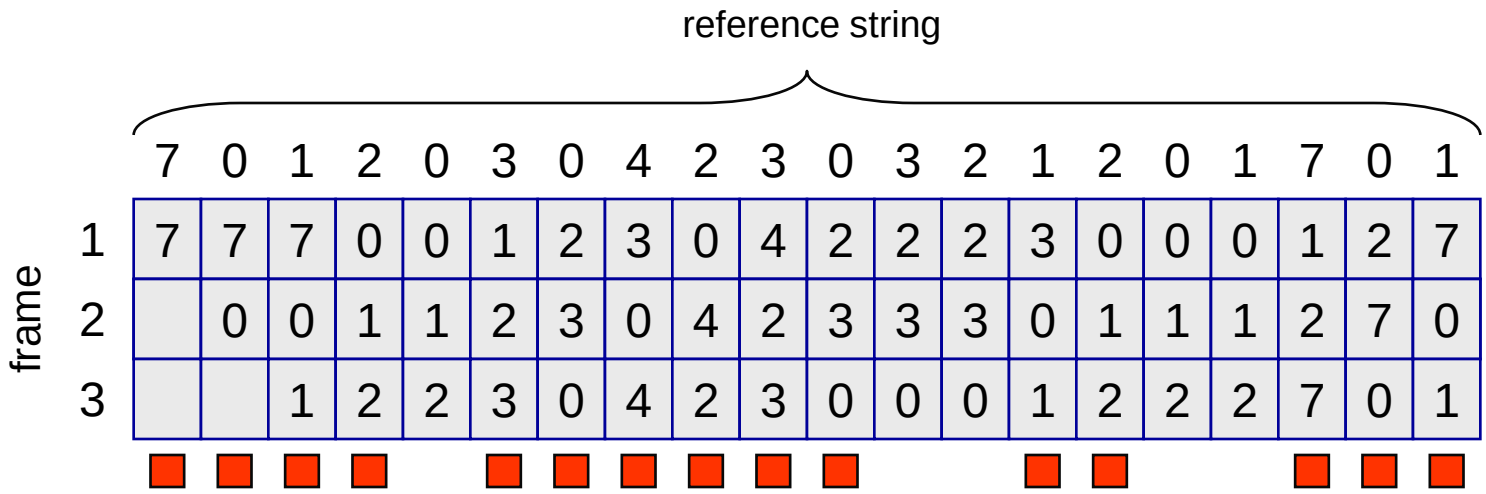
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 frames (4 pages can be in memory at a time per process)



– Belady's Anomaly: more frames ⇒ more page faults

First-In-First-Out – Another Example

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)

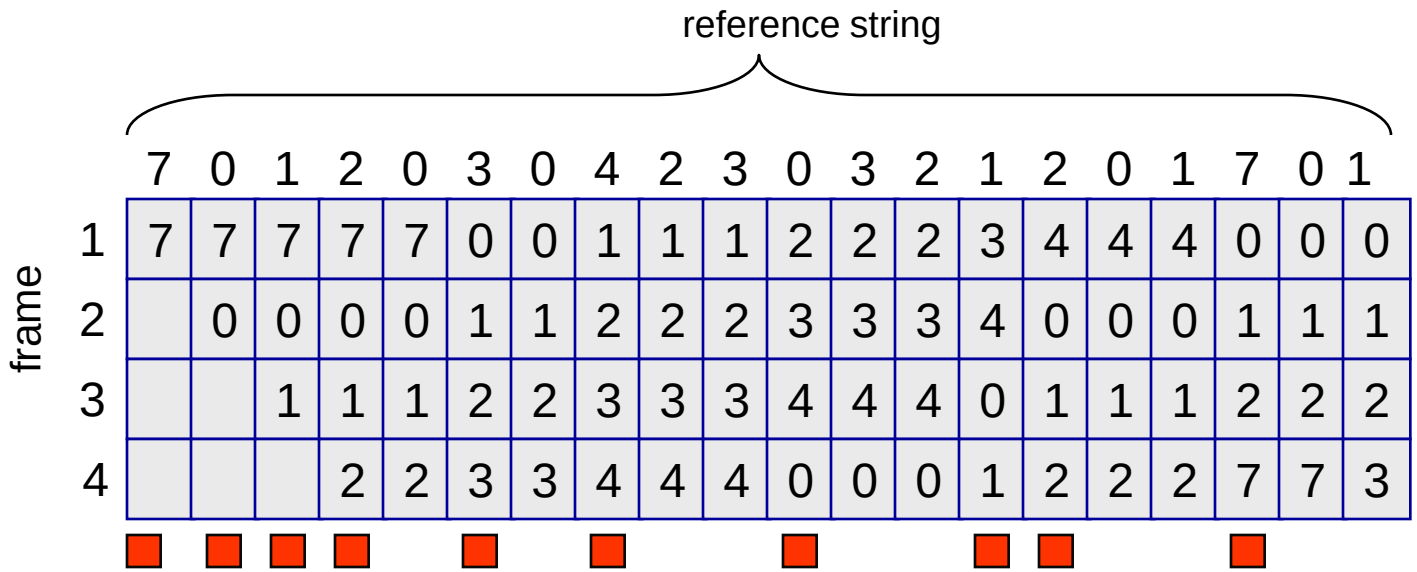


15 page faults

- Does this reference string exhibit Belady's Anomaly?

First-In-First-Out – Another Example

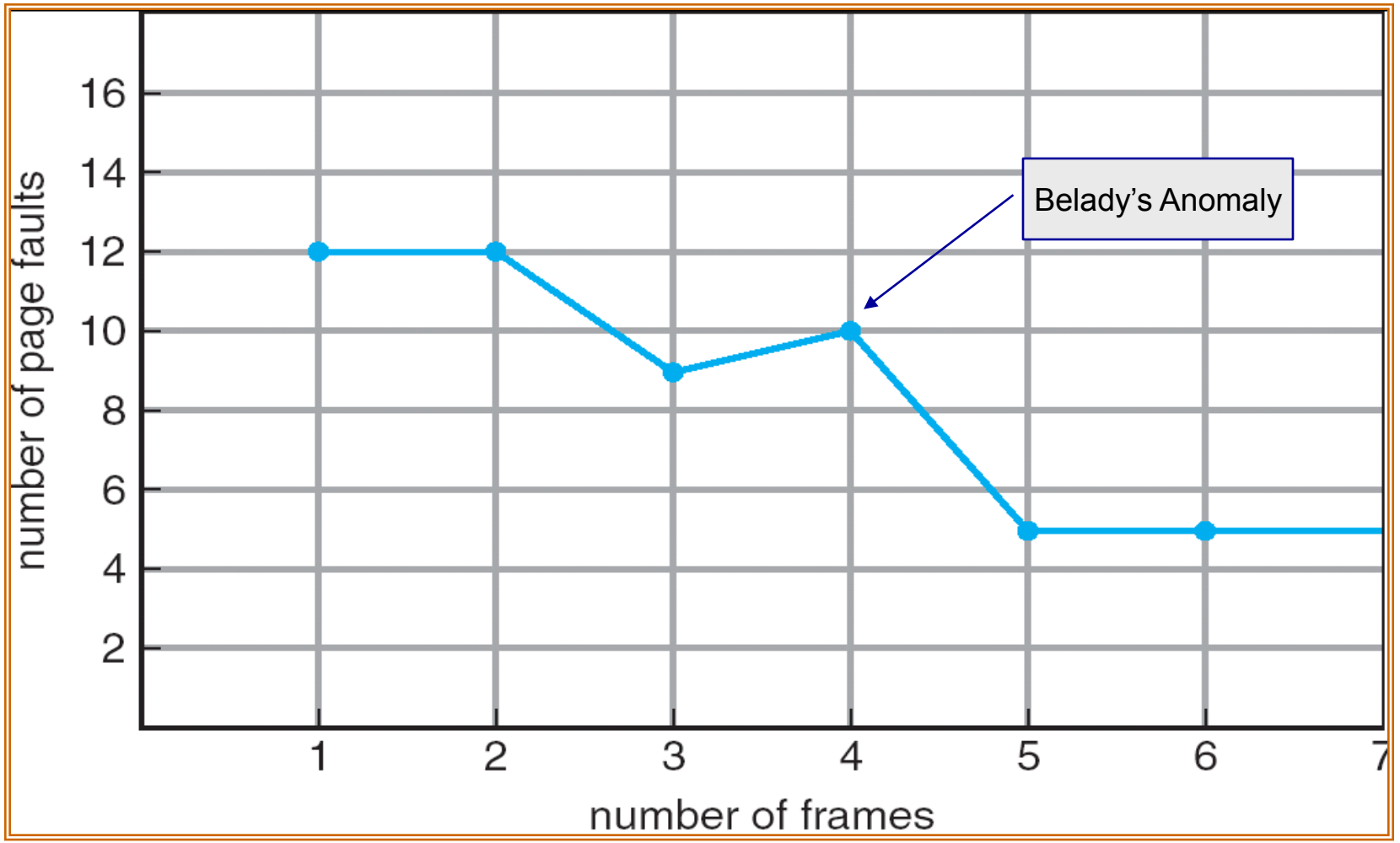
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 4 frames (4 pages can be in memory at a time per process)



10 page faults

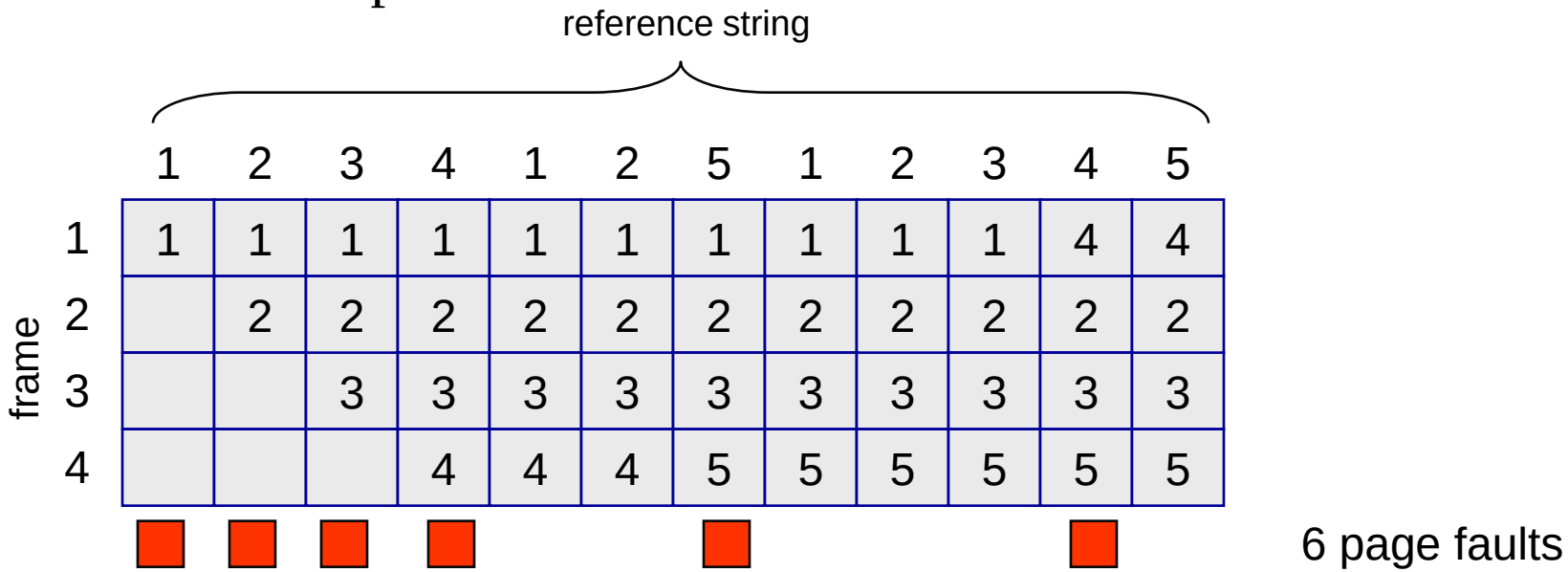
- Answer: No, at least not with 3 and 4 pages.

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

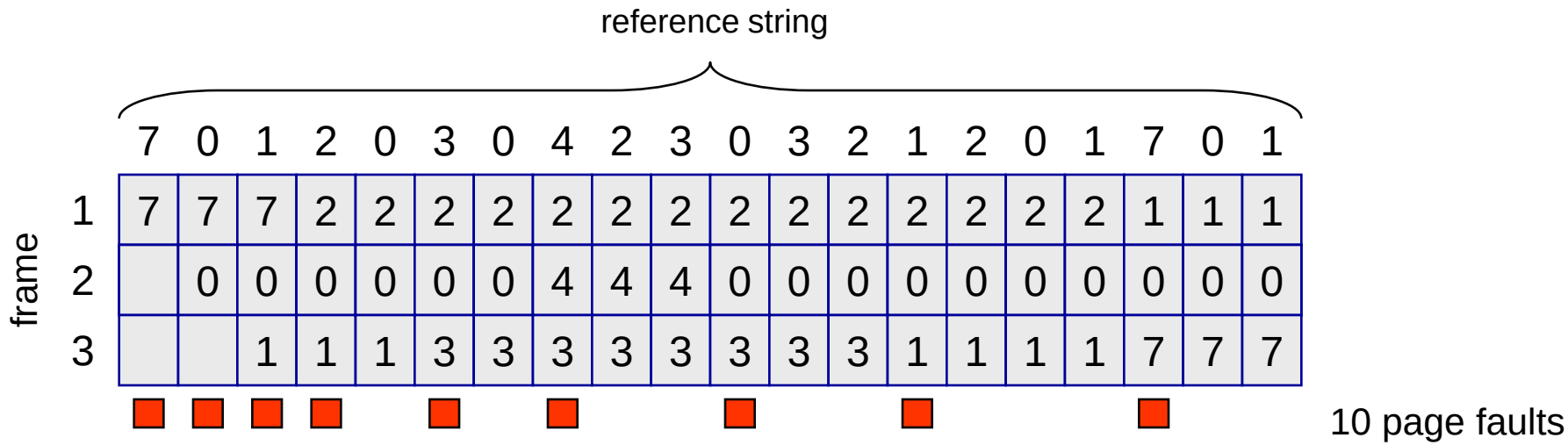


How do you know this?

- Used for measuring how well your algorithm performs

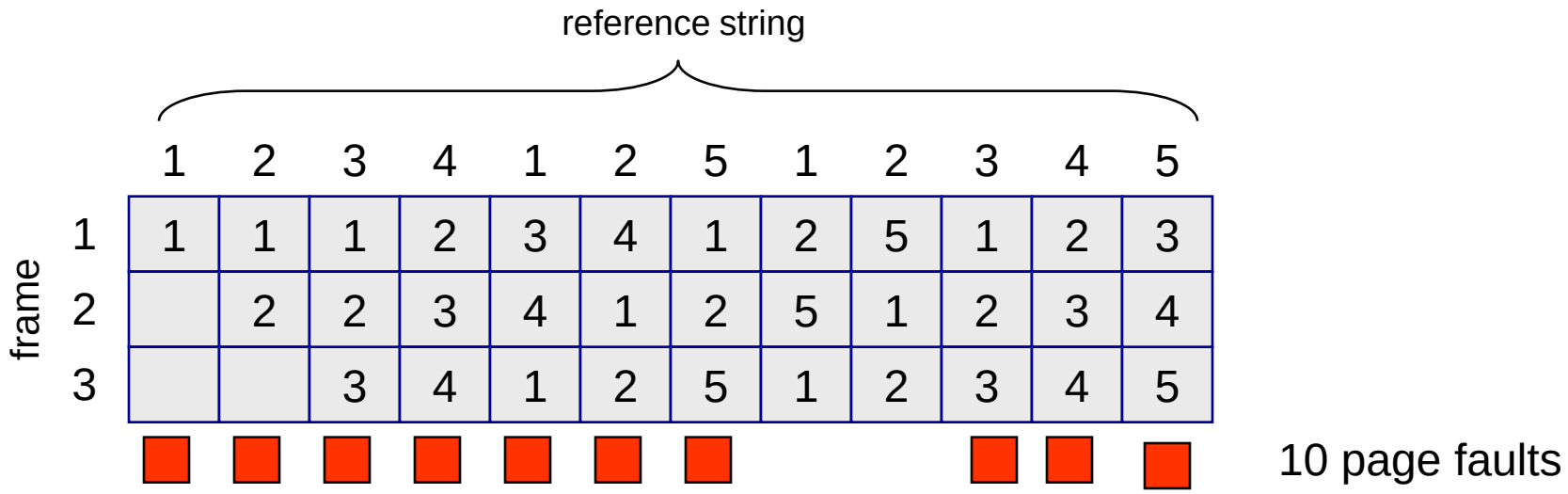
Optimal Algorithm – Another Example

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



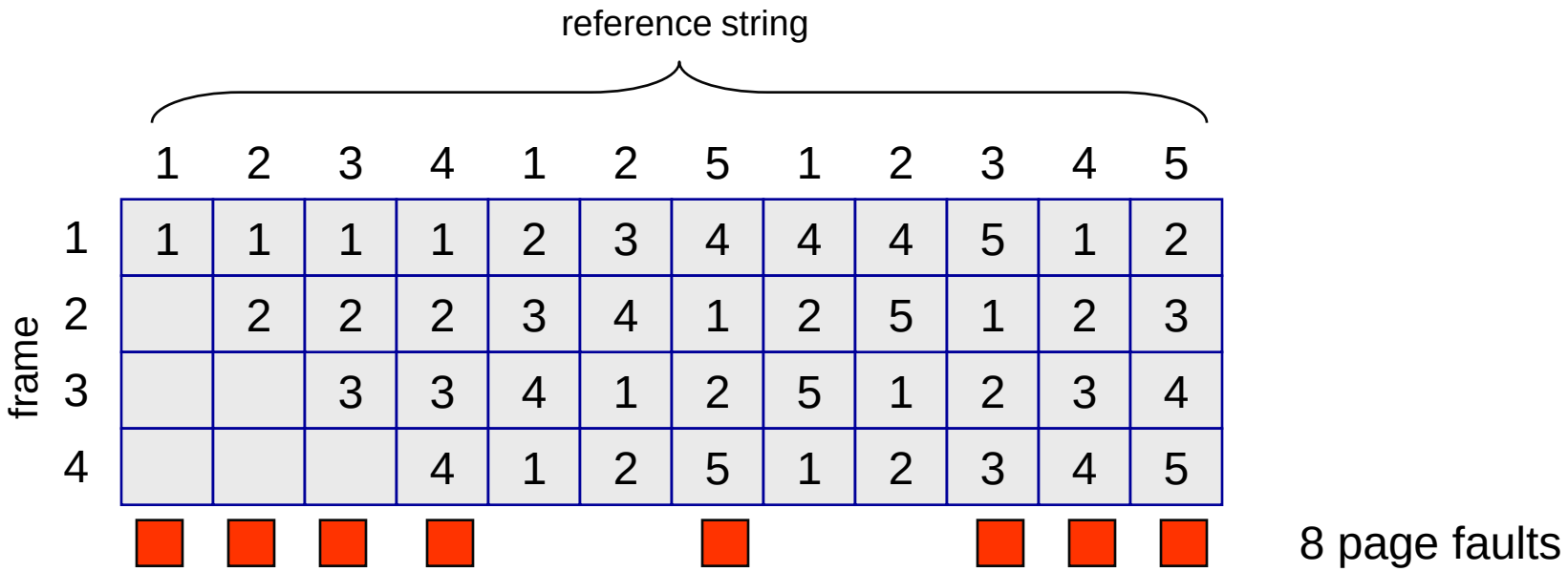
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



LRU Algorithm

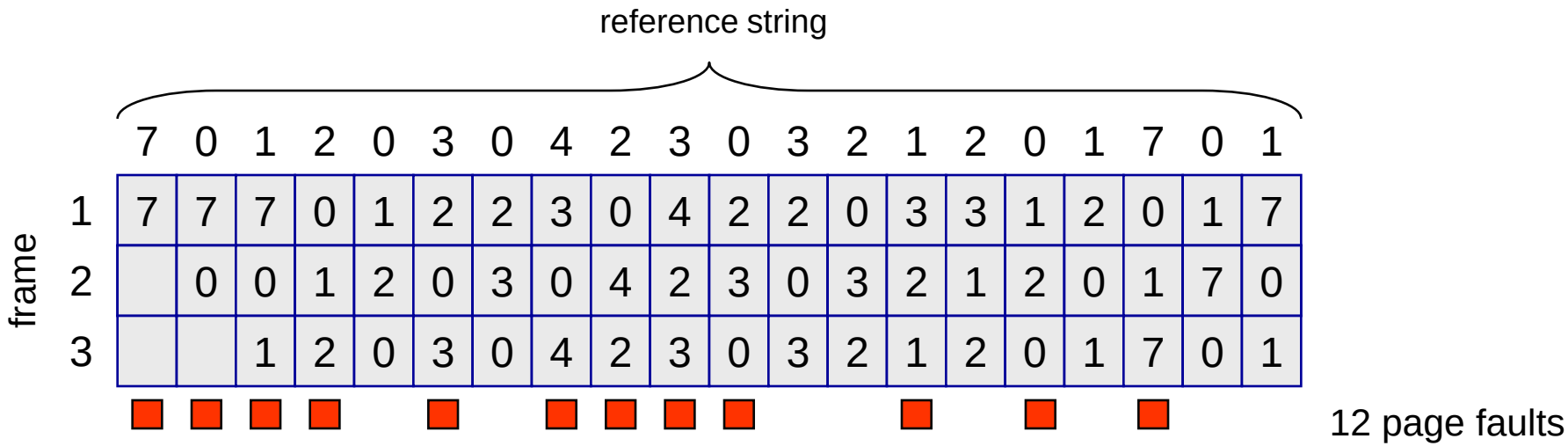
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 frames (4 pages can be in memory at a time per process)



8 page faults

LRU – Another Example

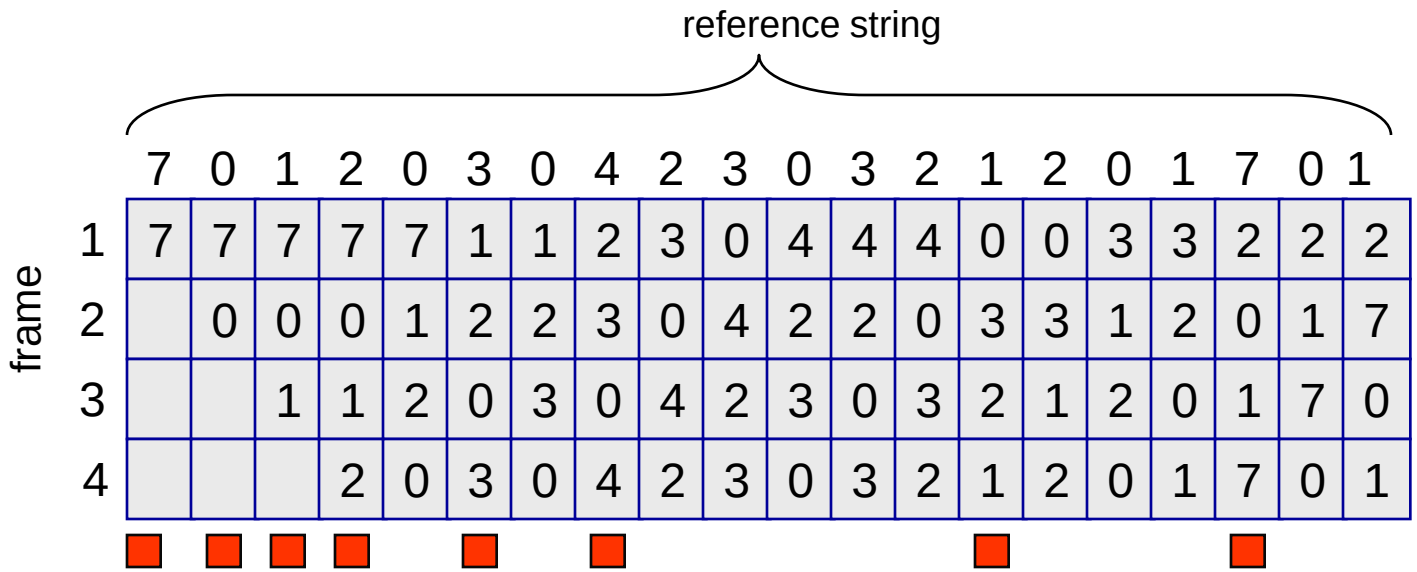
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



- Does this reference string exhibit Belady's Anomaly?

LRU – Another Example

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 4 frames (4 pages can be in memory at a time per process)

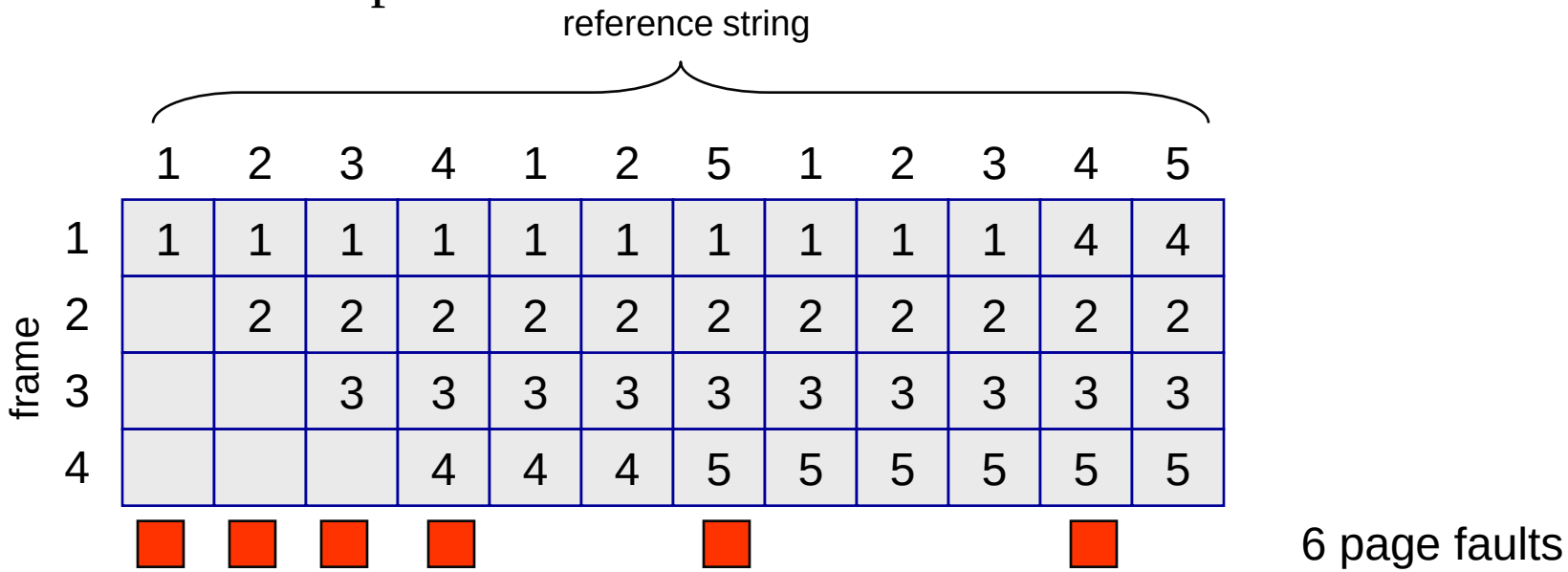


8 page faults

- Answer: No, at least not with 3 and 4 pages.

Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

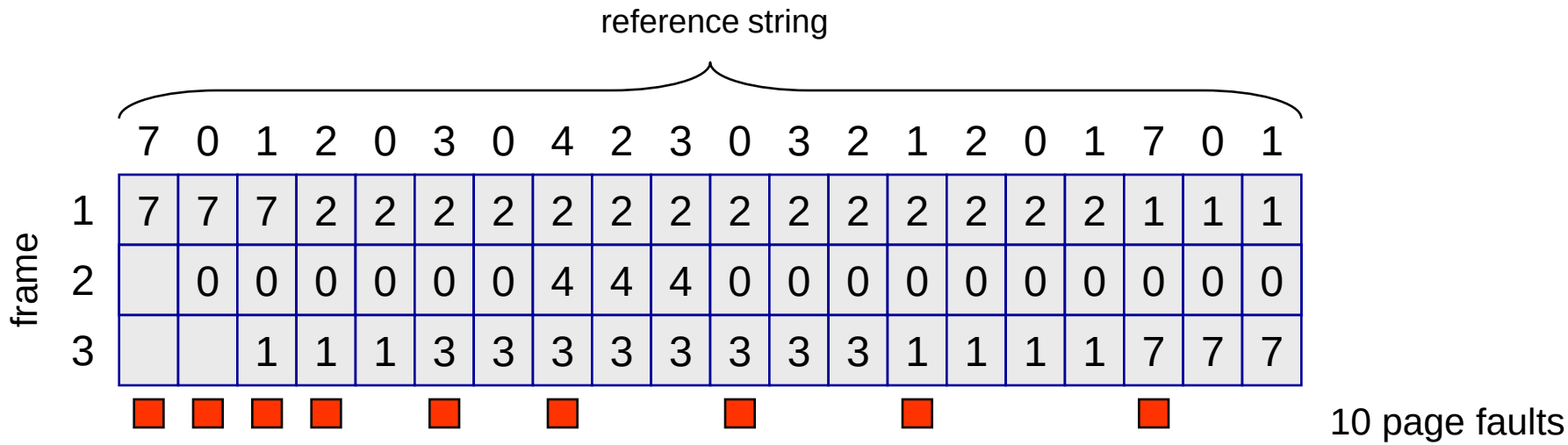


How do you know this?

- Used for measuring how well your algorithm performs

Optimal Algorithm – Another Example

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)

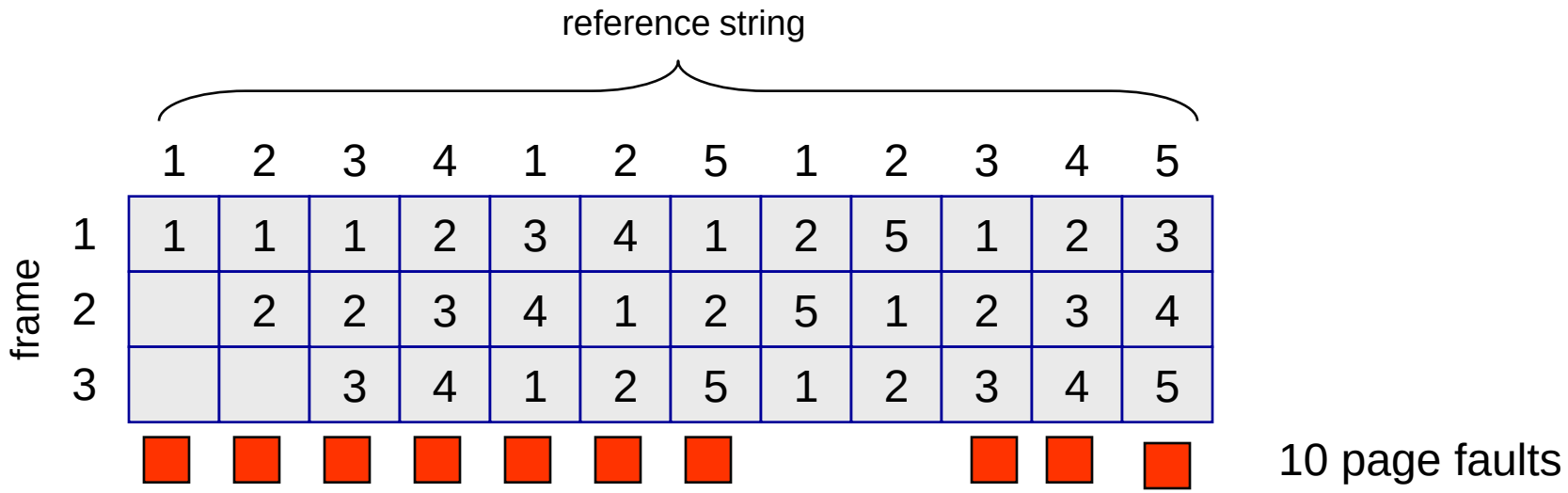


LRU Algorithm (cont.)

- While the optimal algorithm is not feasible for implementation, an approximation of it is possible.
- The main difference between the FIFO and Optimal algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the Optimal algorithm uses the time when a page is to be used.
- If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time.
- The Least Recently Used (LRU) algorithm is an approximation of the optimal algorithm.

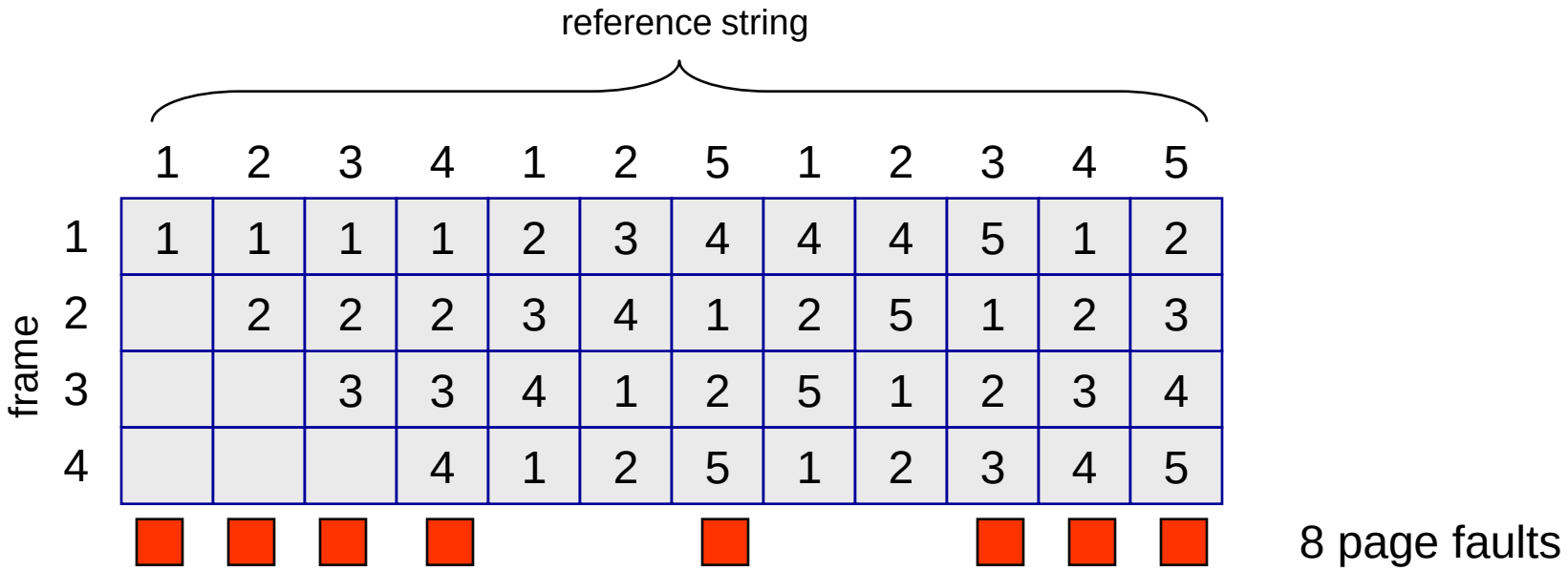
LRU Example

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



Another LRU Example

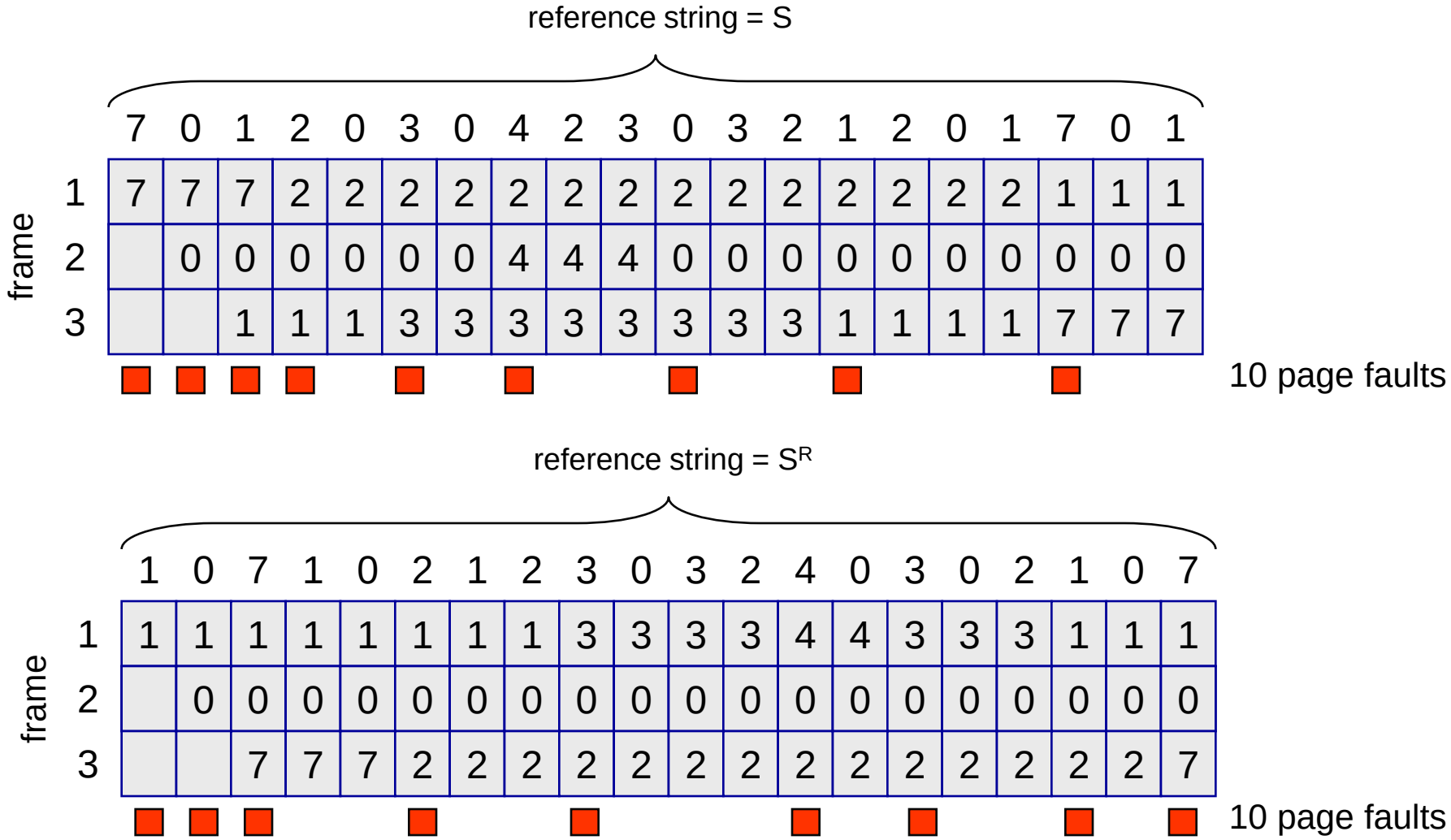
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 4 frames (4 pages can be in memory at a time per process)



LRU Algorithm (cont.)

- The LRU algorithm can be viewed as the optimal page replacement looking backward in time rather than forward.
- Strangely, if we let S^R be the reverse of the reference string S, then the page fault rate for the Optimal algorithm on S is the same as the page-fault rate for the Optimal algorithm on S^R.
- Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on S^R.
- The examples on the next two pages illustrate this phenomenon.

Optimal Algorithm On S and S^R



LRU Algorithm On S and S^R

reference string = S

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

12 page faults

reference string = S^R

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

12 page faults

LRU Algorithm (cont.)

- The LRU algorithm is often used as a page replacement protocol and is considered to be a reasonably good technique.
- The major problem is how to implement LRU replacement.
- An LRU page-replacement algorithm may require substantial hardware assistance.
- The problem is to determine an order for the frames as defined by their last time of use.
- Two implementations are feasible:

1. Counters
2. Stack

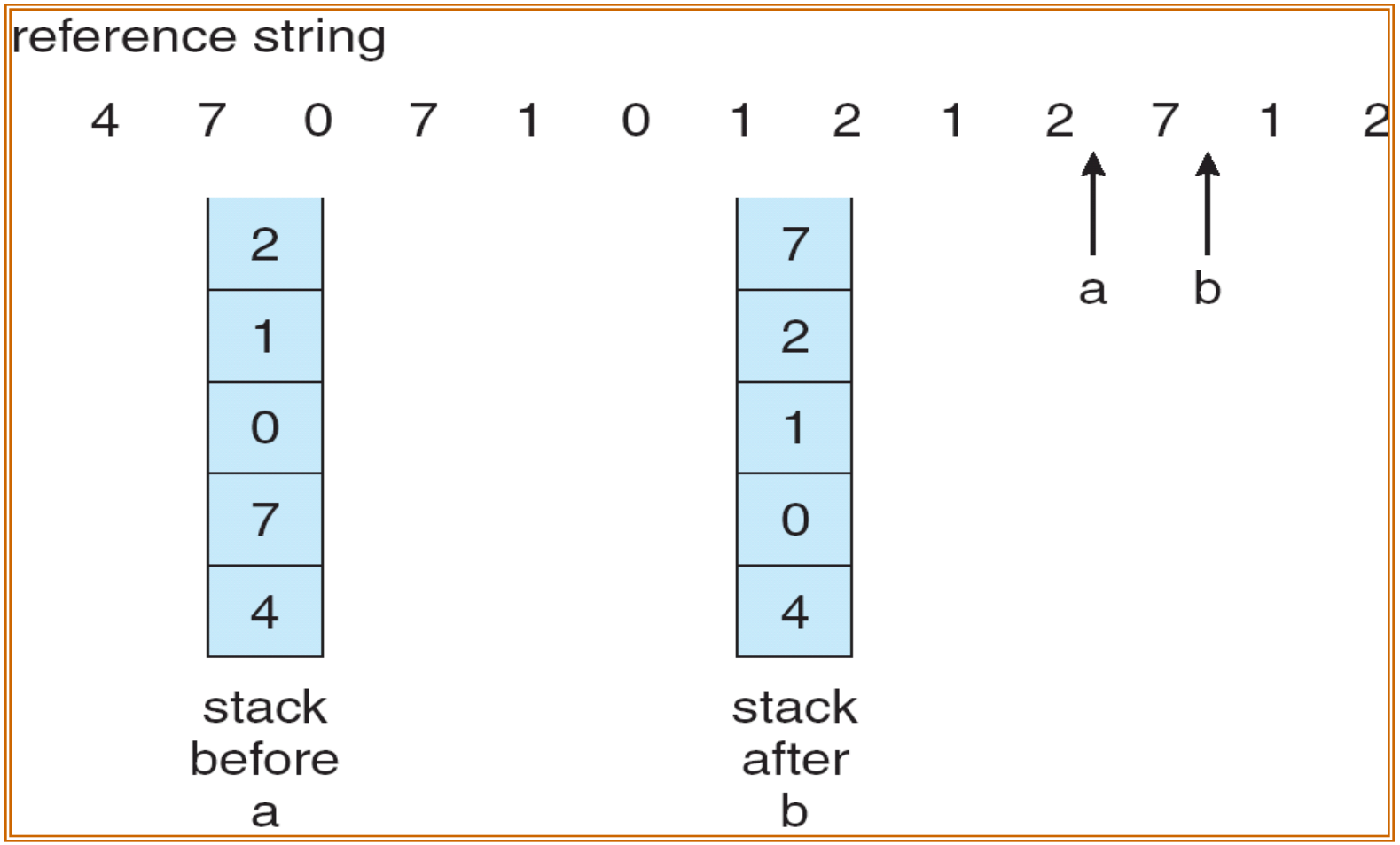
LRU Algorithm – Counter Implementation

- In the simplest case, each page-table entry has an associated time-of-use field and the CPU must include a logical clock or counter.
- The clock is incremented for every memory reference.
- Whenever a reference to a page is made, the content of the clock register are copied into the time-of-use field in the page-table entry for that page. This records the "time" of the last reference to that page.
- The page which is selected as a "victim" is the page with the smallest time value (the oldest page).
- This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access. The times must also be maintained when page tables are changed due to CPU scheduling.

LRU Algorithm – Stack Implementation

- Another approach to implementing LRU replacement is to keep a stack of page numbers.
- Whenever a page is referenced, it is removed from the stack and put on the top. In this manner, the most recently used page is always on the top of the stack and the least recently used page is always on the bottom.
- Since entries are removed from the middle of the stack, the stack is typically implemented as a doubly linked list with head and tail pointers.
- In this fashion, removing a page and putting it on the top of the stack requires changing six pointer values in the worst case.
- Each update is more expensive (a bit) but there is no search required to find the LRU page.

Use Of A Stack to Record The Most Recent Page References



More On Belady's Anomaly

- Did you notice in the examples for the Optimal algorithm and the LRU algorithm that increasing the number of page frames allocated to a process did not increase the number of page faults as was the case with the FIFO algorithm?
- It turns out that the Optimal algorithm and the LRU algorithm belong to a class of algorithms known as stack algorithms.
- A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n* frames is always a subset of the pages that would be in memory with *n+1* page frames.
- Consider the LRU algorithm. The set of pages that would be in memory with an *n* frame allocation would be the *n* most recently used pages. If the number of page frames is increased, then these *n* pages will still be the most recently referenced and so would still be in memory.
- Stack algorithms do not suffer from Belady's anomaly.

LRU Approximation Algorithms

- Unfortunately, few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support and must rely on page replacement algorithms such as FIFO.
- Many systems however, do provide some support in the form of a reference bit.
- The reference bit for a page is set by the hardware whenever that page is referenced (either by a read or a write to any byte in the page).
- Reference bits are associated with each entry in the page table.
- Initially, all bits are cleared (set to 0) by the OS. As a user process executed, the bit associated with each page referenced is set (to 1) by the hardware. After some period of time, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of their use.
- This information is the basis for many page-replacement algorithms that approximate the behavior of the LRU algorithm.

Additional-Reference-Bits Algorithm

- Additional information about the order in which pages are referenced can be obtained by recording the reference bits at regular intervals.
- A single byte (8-bits) is maintained for each page in a table in memory.
- At regular intervals (say 100 msec), a timer interrupt transfers control to the OS. The OS shifts the reference bit for each page into the high-order bit of the its byte, shifting the other bits in the byte, one bit to the right and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register for a page contains 00000000, then that page has not been referenced for eight time periods.

Additional-Reference-Bits Algorithm (cont.)

- Similarly, a page that has been referenced at least once during each of the last eight time periods would have a history register value of 11111111.
- A page with a history register value of 11000100 has been used more recently than a page with a register value of 01110111.
- If the history register values are interpreted as unsigned integers, the page with the lowest number is the LRU page.
- Notice, that the numbers may not be unique, so it is possible to page out all of the pages with the smallest value, or use the FIFO method to choose amongst them.
- The number of bits of history can be varied and is typically selected, depending on the hardware available, to make the updating as fast as possible.

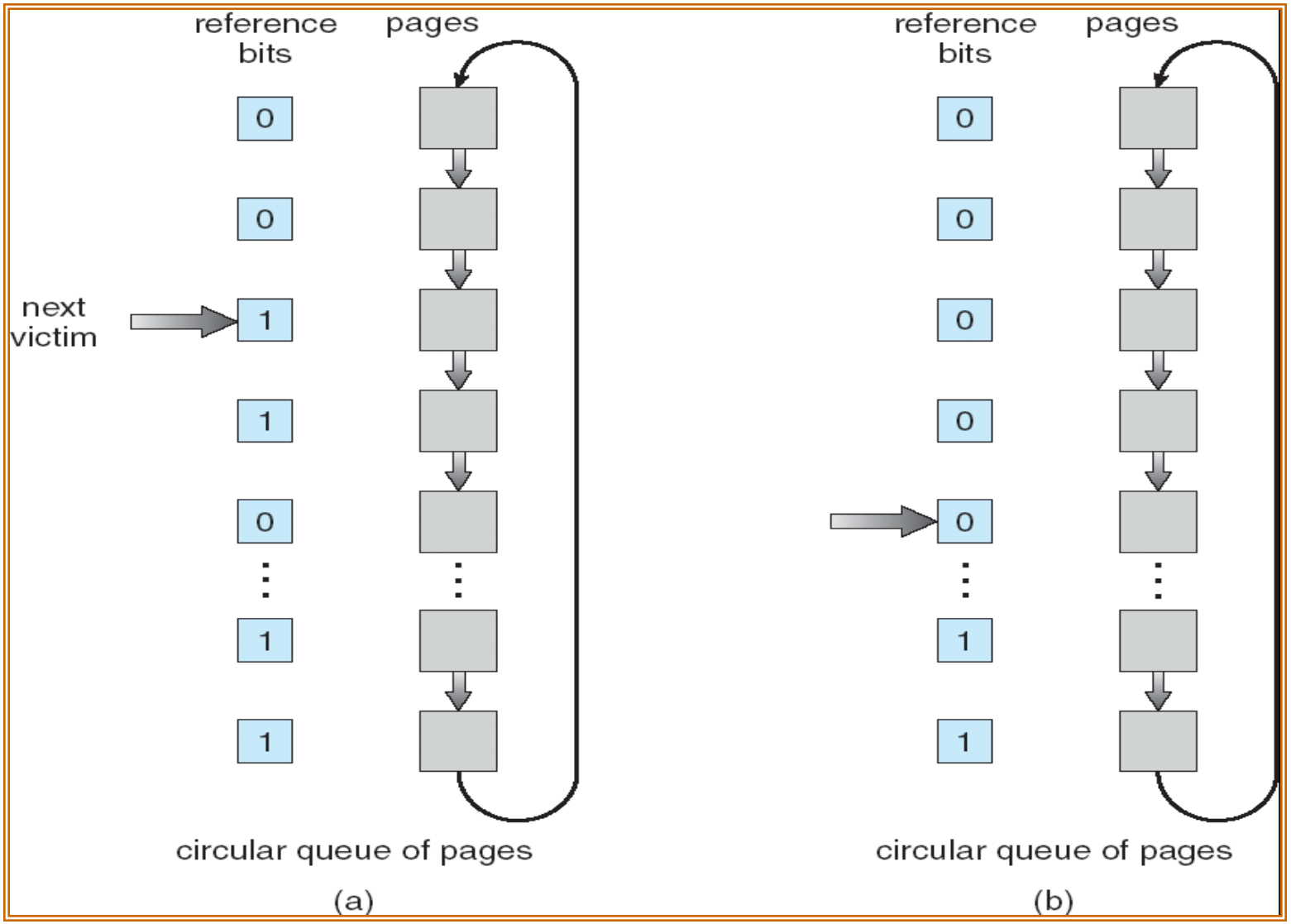
Second-Chance Algorithm

- In the extreme case, the number of bits in the history register is reduced to 0 (i.e., there is no history register, only the reference bit on the page).
- In this case, the page-replacement algorithm is called the second-chance page-replacement algorithm.
- The basic algorithm of second-chance replacement is FIFO, however, when a page has been selected, its reference bit is checked. If the value is 0, the page is replaced; but if the value is 1, we give the page a second chance and move on to select the next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances).
- Additionally, if a page is used often enough to keep its reference bit set to 1, it will never be replaced.

Clock Algorithm

- One way to implement the second-chance algorithm is via a circular queue. This implementation is referred to as the clock algorithm.
- A pointer (i.e., a hand on the clock) indicates which page is to be replaced next.
- When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears all the reference bits.
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.
- The figure on the next page illustrates this implementation.
- Notice that, in the worst case, when all bits are set, the pointer cycles through the entire queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacements.
- Second-chance degenerates to FIFO if all the replacement bits are set.

Clock Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- The second-chance algorithm can be enhanced by considering the reference bit and the modify bit (the bit used to indicate whether any bit on a page has been modified) as an ordered pair.
- With these two bits, there are four possible classes that can be defined:
  1. (0,0) neither recently used nor modified, the best page to replace.
  2. (0,1) not recently used but modified, not quite as good because the page will need to be written out before it can be replaced.
  3. (1,0) recently used but clean – probably will be used again soon.
  4. (1,1) recently used and modified – probably will be used again soon, and the page will need to be written to disk before it can be replaced. Worst case as a victim.
- Each page is in one of these four classes.

Enhanced Second-Chance Algorithm (cont.)

- When page replacement is required, the same basic scheme as the clock algorithm is utilized; but instead of examining whether the page to which the pointer is pointing has the reference bit set to 1, a check is made to determine the class to which that page belongs.
- The page to replace is the first page encountered in the lowest nonempty class.
- Notice that the circular queue may require several scans before a page to replace can be found.
- The major difference between this algorithm and the simpler clock algorithm is that in this case preference is given to those pages that have been modified in order to reduce the number of I/O operations that will be required.

LRU Approximation Algorithms
Counter-Based Algorithms

- Keep a counter of the number of references that have been made to each page.
- Least Frequently Used (LFU) Algorithm: replaces the page with the smallest counter value. The reason for this selection is that an actively used page should have a large reference count.
  - A problem arises however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a high reference count and remains in memory even though it is no longer needed.
  - One solution is to shift the bits in the counter to the right by 1 bit at regular intervals, forming an exponentially decaying average use count.

LRU Approximation Algorithms
Counter-Based Algorithms (cont.)

- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

- As you might expect, neither LFU or MFU replacement is very common. The implementation of these algorithms is expensive, and they do not approximate the Optimal replacement algorithm well.

Allocation Of Frames

- In addition to selecting a victim for page replacement, we must also consider the allocation of frames to processes.
- We saw that with the FIFO page-replacement algorithm that the number of page faults may actually increase for an increase in frame allocation. Although stack algorithms do not suffer from Belady's anomaly, the performance of processes running under these types of page replacement protocols are certainly impacted by the number of frames allocated to the process.
- If for example, we have 100 free frames and two processes, how many frames does each process get?

Allocation Of Frames (cont.)

- Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system would have 128 frames.
- Suppose the OS requires 35 frames, leaving 93 frames for the user process.
- Under pure demand paging, all 93 frames would initially be put on the free frame list. When a user process begins execution, it would generate a sequence of page faults.
- The first 93 faults would all get free frames from the free frame list.
- When the free frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94^{th}, and so on.
- When the process terminates, the 93 frames would be returned to the free frame list.

Allocation Of Frames (cont.)

- There are many variations on the simple strategy outlined on the previous page.
  - We could require that the OS allocate all its buffer and table space from the free-frame list, when not used by the OS, it can be used to support user paging.
  - We could try to maintain a three page frame reserve on the free frame list at all times. Thus, whenever a process page faults, there is always a free frame available to page into. While paging is occurring, a replacement can be selected, which is then written back to the disk as the user process continues to execute.
- Other variants are possible, but the basic strategy is clear: the user process is allocated any free frame.

Minimum Number Of Frames

- Strategies for the allocation of frames are constrained in a variety of ways.
- It is not possible, for example, to allocate more than the total number of available frames (unless there is page sharing).
- A minimum number of frames must also be allocated to each process.
- The obvious reason that a minimum number of frames must be allocated is performance.
  - As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
- Another reason is that when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, the process must have enough frames to hold all the different pages that any single instruction can reference.

Minimum Number Of Frames (cont.)

- An example is the IBM 370 MVC (move characters) instruction. This instruction takes 6 bytes and can straddle two pages. The block of characters to move and the area to which it is to be moved can each also straddle two pages. This situation would require six frames.
- The worst case scenario occurs when the MVC instruction is itself the operand of an EXECUTE instruction that straddles a page boundary; in this case, two additional frames are required, bringing the total to eight.
- Whereas the minimum number of frames is determined by the computer architecture (through its instruction set), the maximum number of defined by the amount of physical memory. In between these two values, we are left with a significant choice in frame allocation.

Frame Allocation Algorithms

- The easiest way to split *m* frames among *n* processes it to give each process an equal share, *m/n* frames. This scheme is known as equal allocation.
  - For example, if there are 93 frames and 5 processes, each process will get 93/5 = 18.6 = 18 frames. The leftover 3 frames can be used as a free frame pool.
- An alternative is to recognize that various processes will need differing amounts of memory.
  - For example, consider a system with 1 KB frames. A process of 10 KB and a second process of 127 KB are the only two processes running with 62 free frames. It makes no sense to give 31 frames to the 10 KB process which in the worst case will need only 10 frames, which will waste the other 21 frames, which could have been allocated to the 127 KB process.
- This alternative is known as proportional allocation.

Proportional Allocation

- Using proportional allocation we have the following:

$s_i = \text{size of virtual memory for process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

- Example

$m = 64$

$s_i = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

Note: With either equal or proportional allocation, the allocation may vary depending on the degree of multiprogramming. If the multiprogramming level is increased, each process will lose some frames to meet the allocation for the new process. Similarly, if the degree of multiprogramming is decreased, the frames that were previously allocated to the departed process can be spread over the remaining processes.

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement.
- With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories:
  - Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another.
  - Local replacement – each process selects from only its own set of allocated frames.
- For example, consider an allocation scheme where we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

Global vs. Local Allocation (cont.)

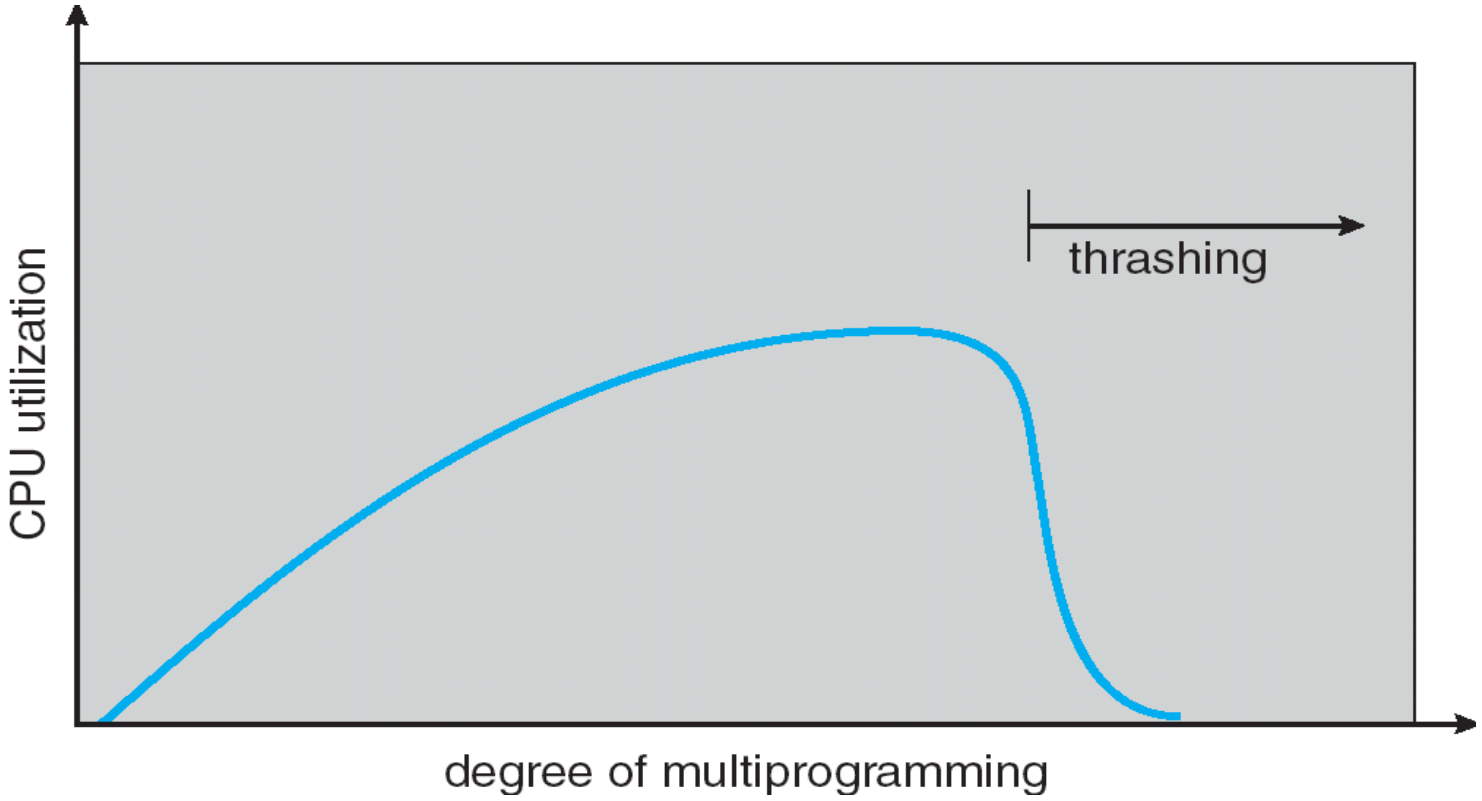
- With a local replacement strategy, the number of frames allocated to a process does not change (unless the degree of multiprogramming changes).
- With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it (assuming that other processes did not choose its frames for replacement).
- One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes. Therefore, the same process may perform quite differently for different executions because of totally external circumstances.
- Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process.
- Local replacement might hinder a process, however, by not making available to it other, less used pages of memory.
- Thus, global replacement generally results in greater system throughput and is therefore the more common method.

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high.
- This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- Thrashing ≡ a process is busy swapping pages in and out without accomplishing any real activity.

Thrashing (cont.)

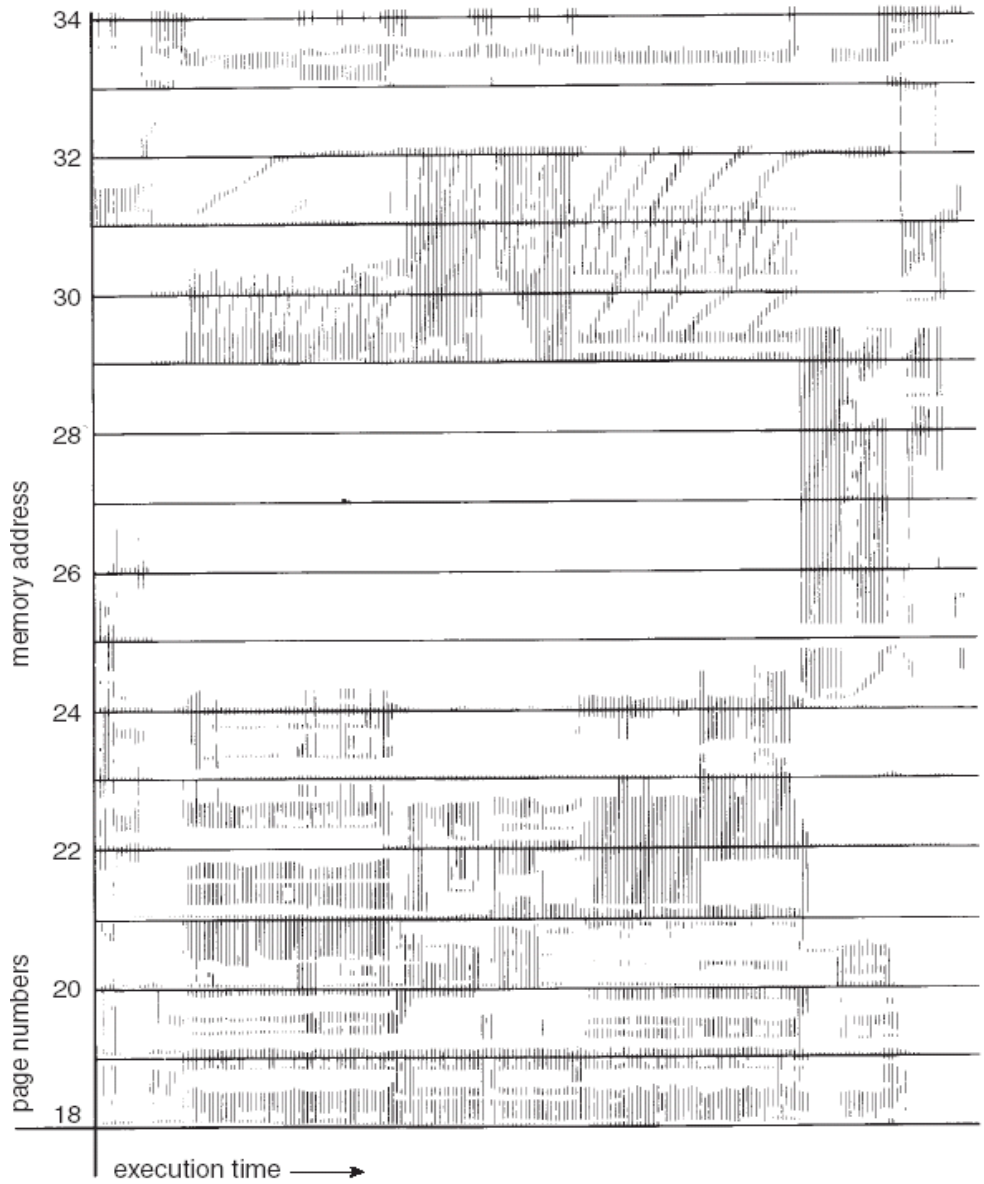


Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > total memory size

Locality In A Memory-Reference Pattern

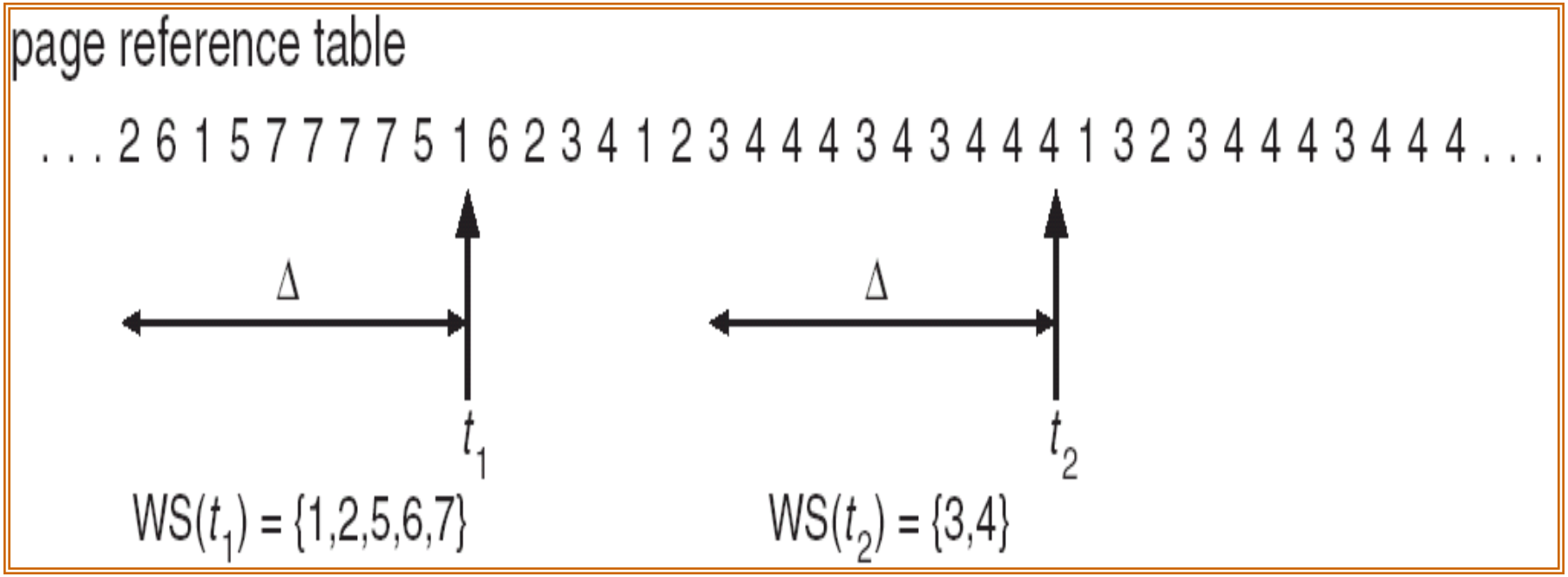


Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions.
- WSS_i (working set size of process P_i) =
  total number of pages referenced in the most recent Δ

  (time variant)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend one of the processes

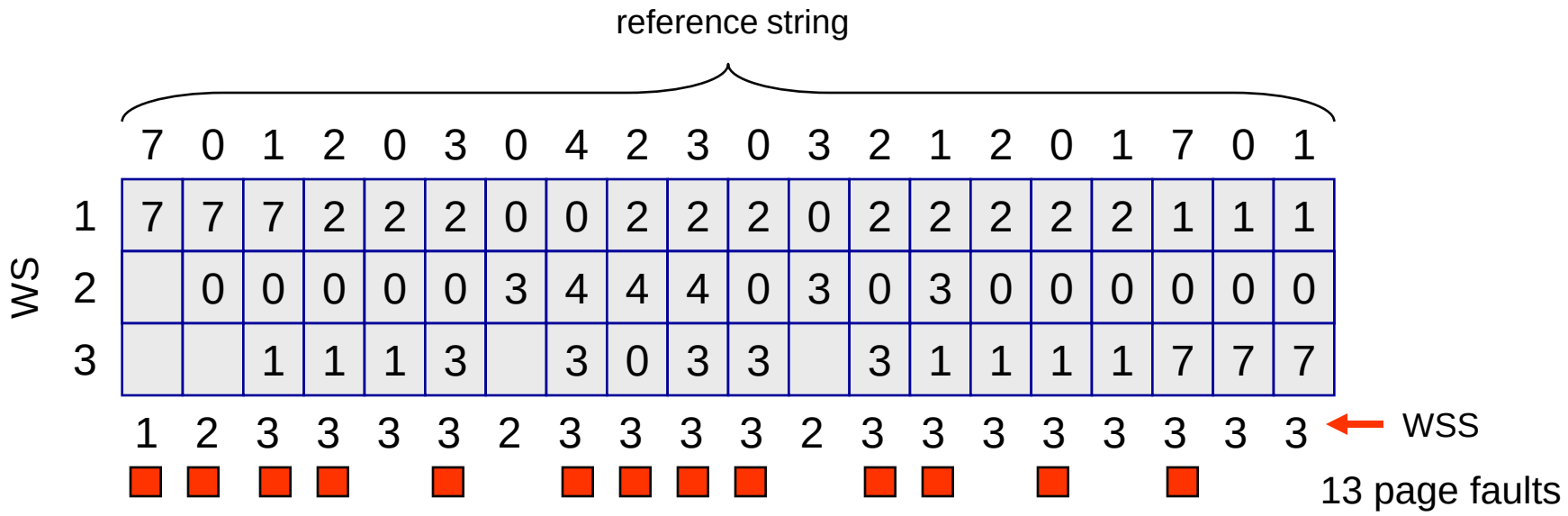
Working-set model Example



Assume Δ = 10

Working Set Model – Another Example

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- $\Delta = 3$



Working-Set Model (cont.)

- Once Δ has been selected, using the working set model is simple.
- The OS monitors the working set of each process and allocates to that working set enough page frames to provide it with its working set size.
- If there are enough extra frames, another process can be initiated.
- If the sum of the working set sizes increases, exceeding the total number of available frames, the OS will select a process to suspend. The suspended process's pages are swapped out, and its frames are reallocated to other processes. The suspended process will be restarted later.
- The working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible, thus optimizing CPU utilization.

Keeping Track of the Working Set

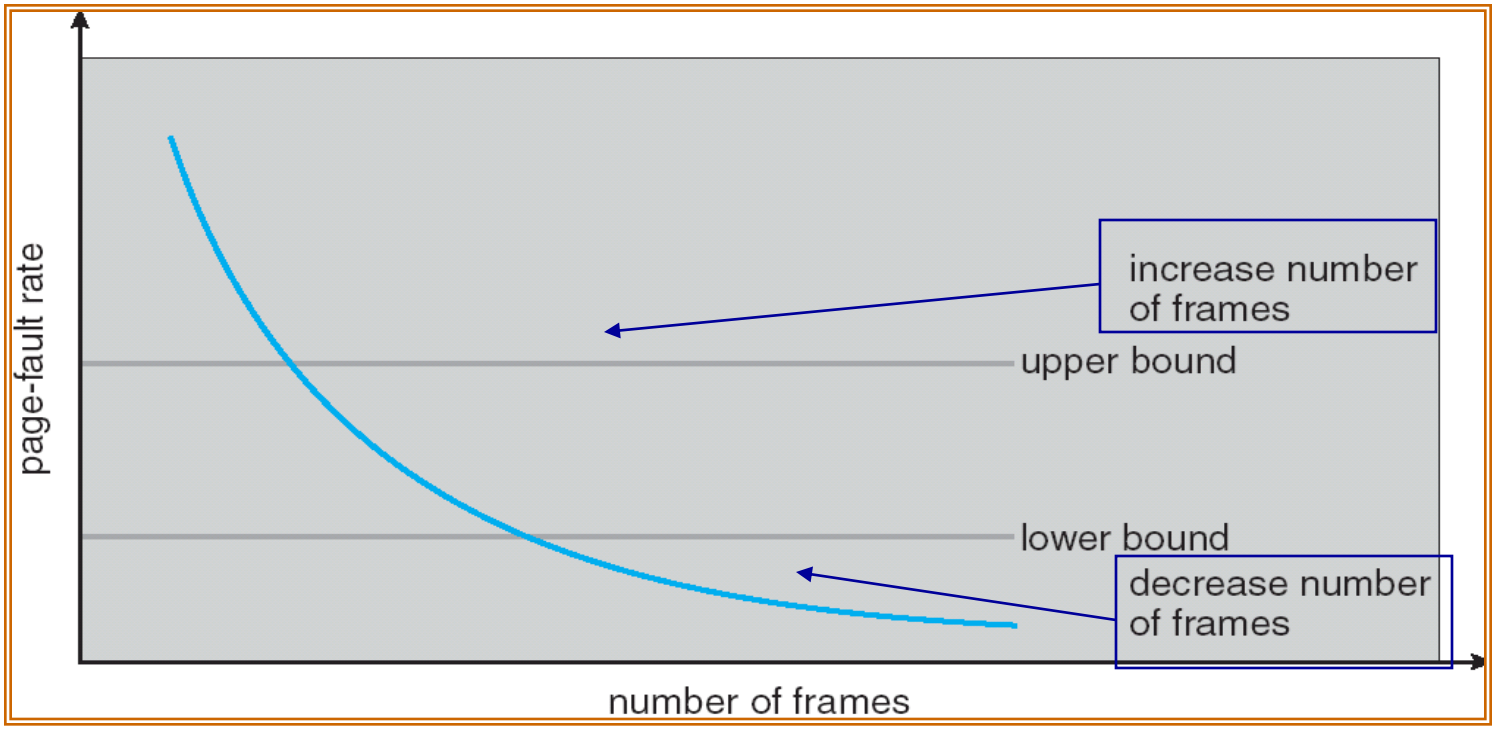
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate? Because you can't tell where, within an interval of 5000, a reference occurred.
- Improvement = 10 bits and interrupt every 1000 time units. The disadvantage to this approach is higher cost to service more frequent interrupts.

Page-Fault Frequency Scheme

- While the working set model is successful, and knowledge of the working set can be useful for prepaging (more later), it is a clumsy mechanism for controlling thrashing.
- A strategy that uses the page-fault frequency (PFF) is a more direct approach for controlling thrashing.
- Since thrashing exhibits a very high page fault rate, we need to control the page fault rate.
  - Too high a page fault rate implies that a process needs more page frames.
  - Too low a page fault rate implies that a process may have more page frames than it needs.
- Establish upper and lower bounds on the page fault rate.

Page-Fault Frequency Scheme

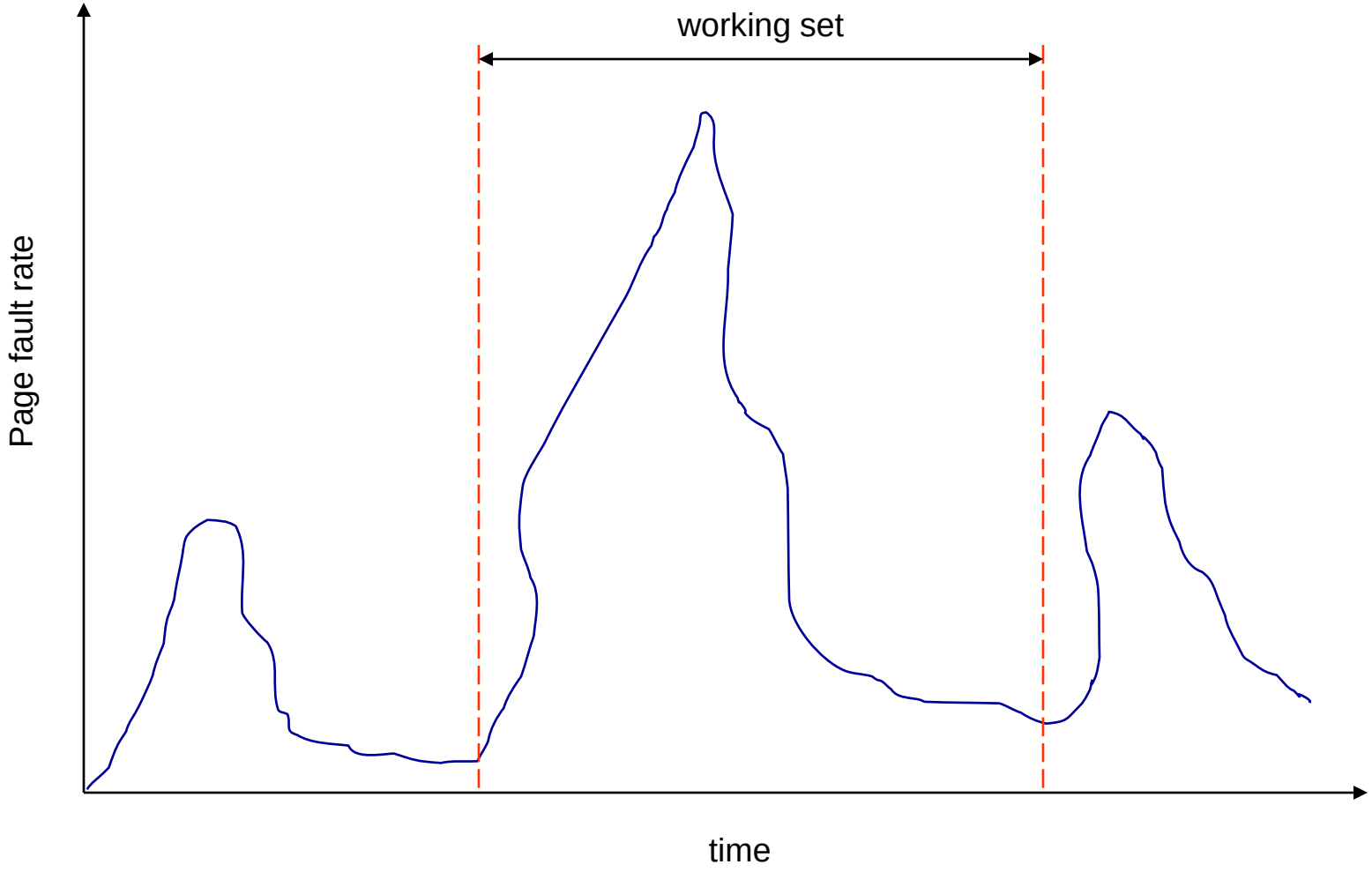
- Establish an "acceptable" page-fault rate
  - If the actual page fault rate is too low, process loses frames.
  - If the actual page fault rate is too high, process gains frames



Working Sets and Page Fault Rates

- There is a direct relationship between the working set of a process and its page fault rate.
- As shown in the example on page 75, typically the working set of a process changes over time as references to code and data sections move from one locality to another.
- Assuming that the process is not thrashing (i.e., it has a sufficient frame allocation), the page fault rate of the process will transition between peaks and valleys over time.
- This general behavior is illustrated on the next page.

Working Sets and Page Fault Rates (cont.)



Working Sets and Page Fault Rates (cont.)

- A peak in the page fault rate occurs when demand paging begins in a new locality.
- Once, the working set of the new locality is in memory, the page fault rate falls.
- When the process moves to a new working set, the page fault rate rises towards a peak once again, returning to a lower rate once the new working set is in memory.
- The span of time between the start of one peak and the start of the next peak illustrates the transition from one locality to another (one working set to another).

Process Suspension

- If the degree of multiprogramming is to be reduced, one or more of the currently resident processes must be suspended (swapped out).
- There are six commonly used choices:

1. The lowest priority process – this implements a scheduling policy decision and is unrelated to performance issues.
2. The faulting process – the idea here is that there is a greater probability that the faulting process does not have its working set resident, and performance would suffer the least by suspending it. In addition, this choice would have an immediate payoff because it blocks a process that is about to be blocked anyway and eliminates the overhead of a page replacement and I/O operation.

Process Suspension

3. The last process activated – this is the process that is least likely to have its working set resident.

4. The process with the smallest resident set – this will require the least future effort to reload the process. However, it penalizes processes with strong locality.

5. The largest process – this obtains the most free frames in an overcommitted memory, making additional deactivation in the near future unlikely.

6. The process with the largest remaining execution window – in most process scheduling schemes, a process may run for only a certain quantum of time before being interrupted and placed at the end of the ready queue. This approximates a shortest processing time first scheduling discipline.