Memory Management

CS3026 Operating Systems

Lecture 10

Page Replacement Policies

Page Replacement Algorithms

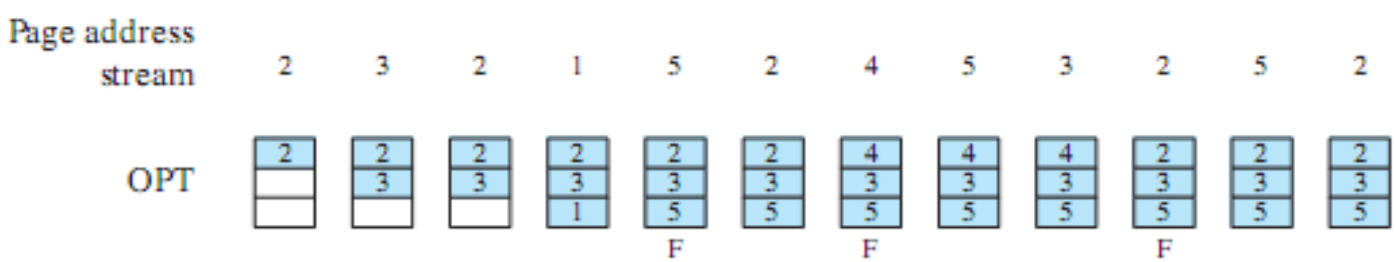
- The optimal policy (OPT) as a benchmark

- Algorithms used
  - Least recently used (LRU)
  - First-In-First-Out (FIFO)
  - Clock Algorithm

Optimal Policy OPT

- Assumption
  - 3 memory frames available
  - Process has 5 pages
  - When process is executed, the following sequence of page references occurs (called a page reference string):

Page reference string: 2 3 2 1 5 2 4 5 3 2 5 2



F= page fault occurring after the frame allocation is initially filled

OPT policy results in 3 replacement page faults (minus the initial ones to fill the empty frames, total page faults are 6)

LRU Least Recently Used Policy

Page 1 least recently used

Page 3 least recently used

Reference to page 5:

- **We replace page 3 in Frame 2**

Reference to pages 4:

- **We replace page 1 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	2					
Frame2		3	3	3	5	5	5					
Frame3				1	1	1	4					
Page fault					F		F					

Reference to page 3:

- **We replace page 2 in Frame 1**

Reference to page 2:

- **We replace page 4 in Frame 3**

Page ref.	2	3	2	1	5	2	4	5	3	2	5	2
Frame1	2	2	2	2	2	2	2	2	3	3	3	3
Frame2		3	3	3	5	5	5	5	5	5	5	5
Frame3				1	1	1	4	4	4	2	2	2
Page fault					F		F		F	F		

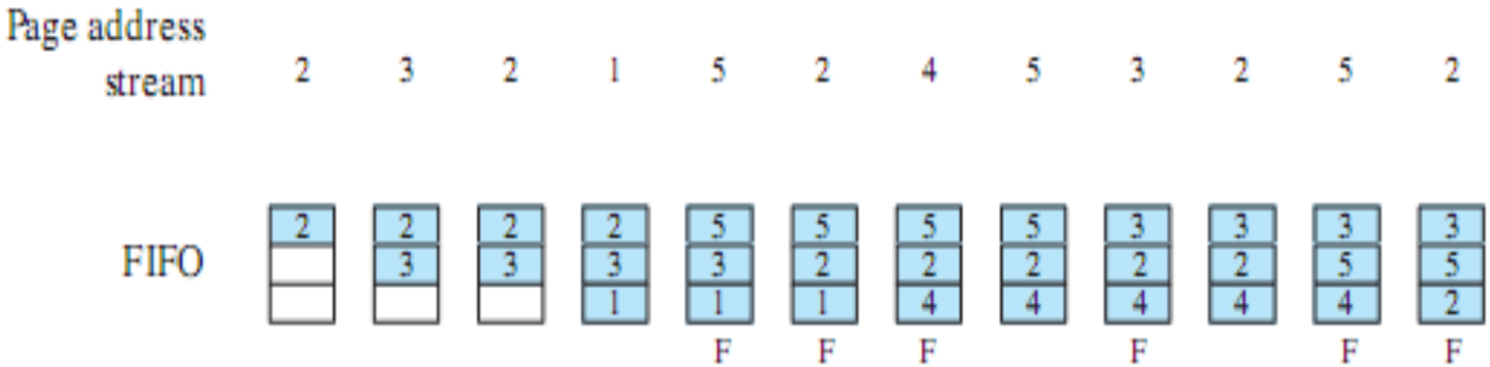
- With LRU, we replace page 2 with page 3, and immediately afterwards, we need page 2 again
- LRU is not able to detect this situation
- However, it is a strategy that comes close to OPT

FIFO and Second Chance Algorithm

First-In-First-Out (FIFO)

- First in First out
  - The first page that was loaded is also the first to be discarded (oldest page) – FIFO ordering of pages
- Problem
  - Oldest page may be the most frequently used
- Is the simplest replacement policy to implement
  - Treats memory frames allocated to a process as a circular buffer
  - Pages are replaced in a round-robin style
  - Strict ordering of pages due to age
- Does not indicate how heavily a page is actually used

FIFO Example

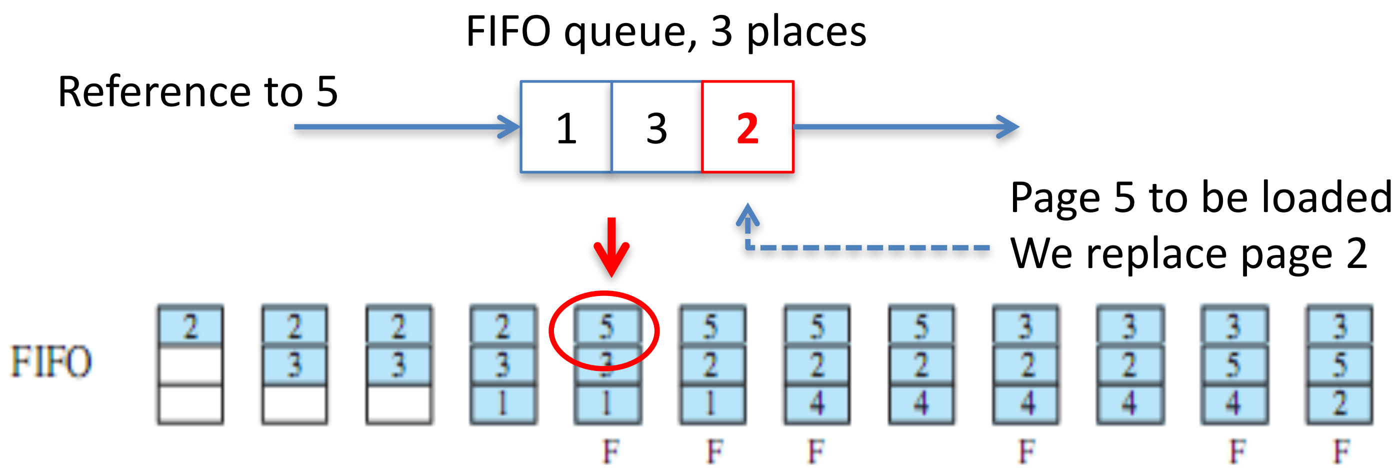


F= page fault occurring after the frame allocation is initially filled

- FIFO policy results in 6 replacement page faults (minus the initial ones to fill the empty frames, total page faults are 9)

FIFO Example

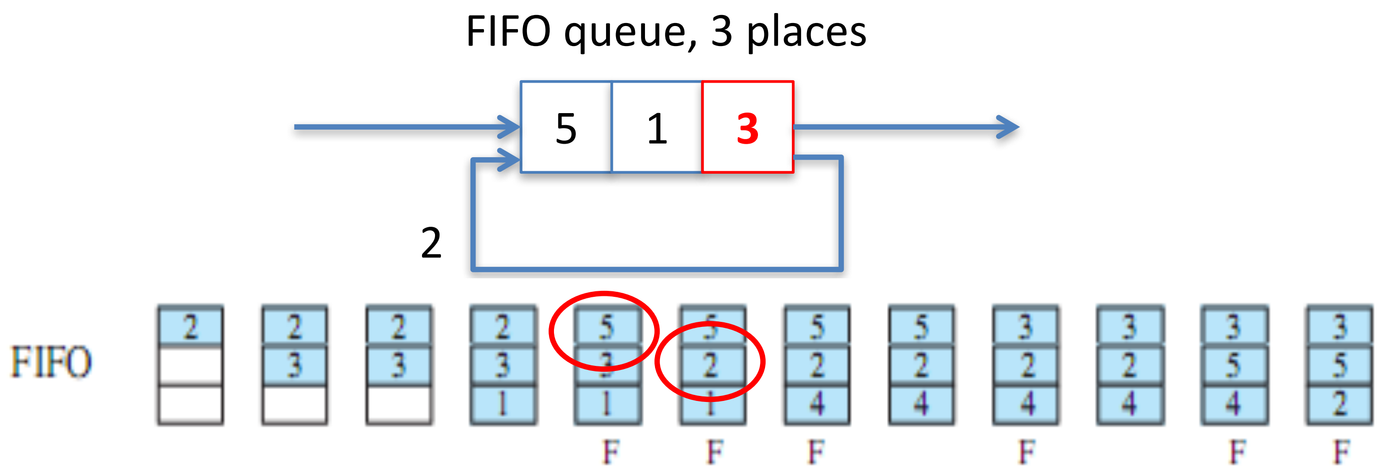
Page reference string: 2 3 2 1 5 2 4 5 3 2 5 2



- FIFO queue has 3 places (for 3 frames)
- References to pages 2, 3 and 1 will fill the available set of frames
- When page 5 is referenced, we replace the “oldest page”, which is 2
- We maintain a FIFO queue to record which page is the “oldest”

FIFO Example

Page reference string: 2 3 2 1 5 2 4 5 3 2 5 2

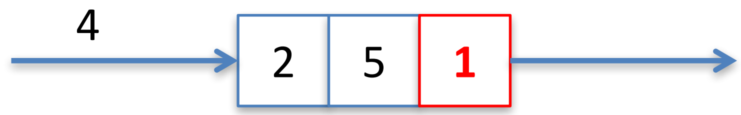


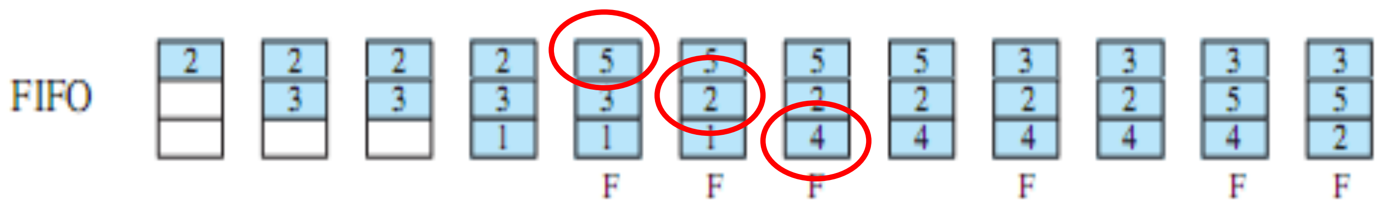
- FIFO queue has 3 places (for 3 frames)
- After page 5, page 2 is referenced again immediately, therefore it has to be brought back into memory
- We replace page 3

FIFO Example

Page reference string: 2 3 2 1 5 2 4 5 3 2 5 2

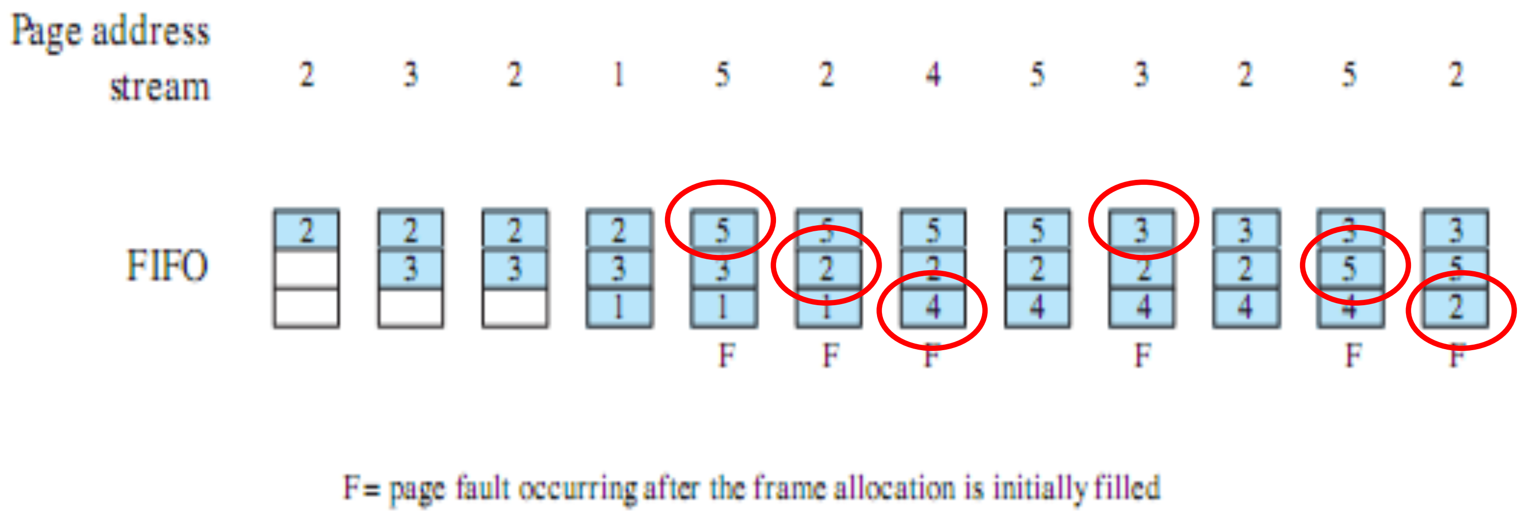
FIFO queue, 3 places





- FIFO queue has 3 places (for 3 frames)
- When queue is full, oldest page discarded, new page added
- Problem: actual demand or “access frequency” for pages is not considered, may lead to frequent reload of the same page

FIFO Example



- FIFO policy results in 6 replacement page faults (minus the initial ones to fill the empty frames, total page faults are 9)

FIFO Page Replacement

- Benefit of FIFO queue
  - Sequence in queue shows which page is the oldest
  - Simple to implement
- Page replacement
  - Newly loaded page added to the end of the queue, tail of FIFO queue
  - Oldest page is the first in line, is removed when replaced
  - Gives us a simple criteria which page to replace

FIFO Page Replacement

- Problem
  - FIFO only records sequence of arrival, not the “importance” of a page
- What if the “oldest” page is the most frequently used (the “most important” page)?
  - FIFO would discard this page and frequently load again – danger of “thrashing”

Second Chance Algorithm

Second Chance Algorithm

- Is a modification of the FIFO replacement algorithm that tries to avoid replacing recently used pages
  - Tries to account for the “importance” of a page
- Uses additional information
  - Each page has reference bit: records whether a process accessed page recently
- A page is given a second chance, if it was accessed recently:
  - will be moved to the tail of the FIFO queue and becomes the youngest page, not replaced
    - Page is “overlooked” and remains in physical memory
  - Reference bit is reset by this “second chance” action

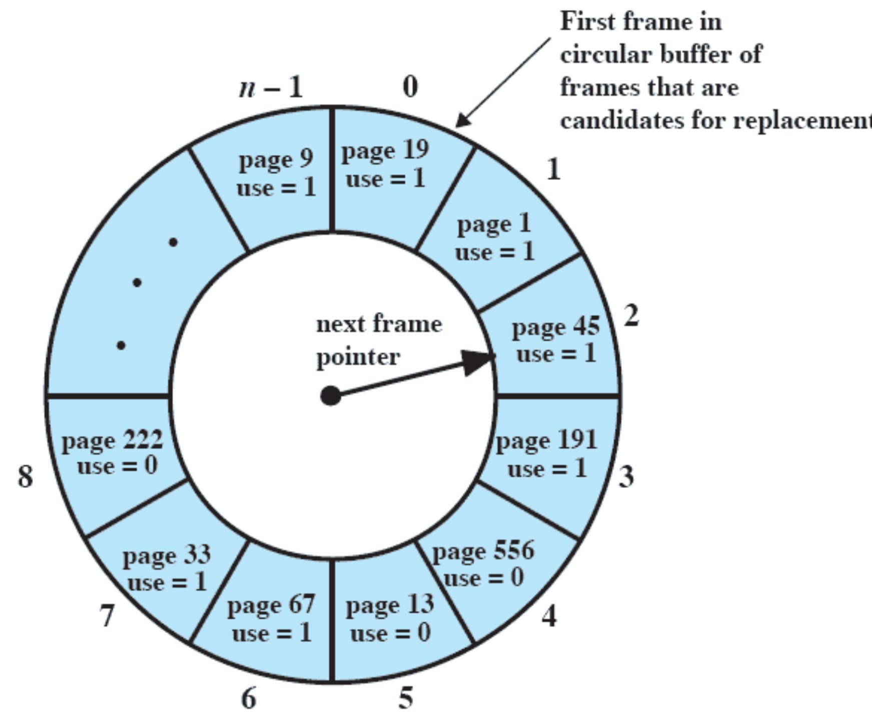
Second Chance Algorithm

- When oldest page selected for replacement, the reference bit is inspected
  - If value is 1: page was accessed recently, page gets a "second chance":
    - **Clear reference bit, set to 0**
    - Check next-oldest page in queue
  - If value is 0: replace page
- A page is given a second chance:
- Worst case:
  - All reference bits == 1, degenerates into pure FIFO

Clock Policy

Implementation of Second-Chance Algorithm

- The set of frames is considered as laid out like a circular buffer (FIFO queue)
  - Is considered a FIFO strategy
  - Can be circulated in a round-robin fashion
- Each frame has a “use” bit
  - When a page is first loaded (a page fault that leads to its load), the “use” bit of the frame is set to 1
- Use bit is “refreshed”
  - Each subsequent reference to this page again sets /overwrites this “use” bit to 1



(a) State of buffer just prior to a page replacement

Clock Algorithm

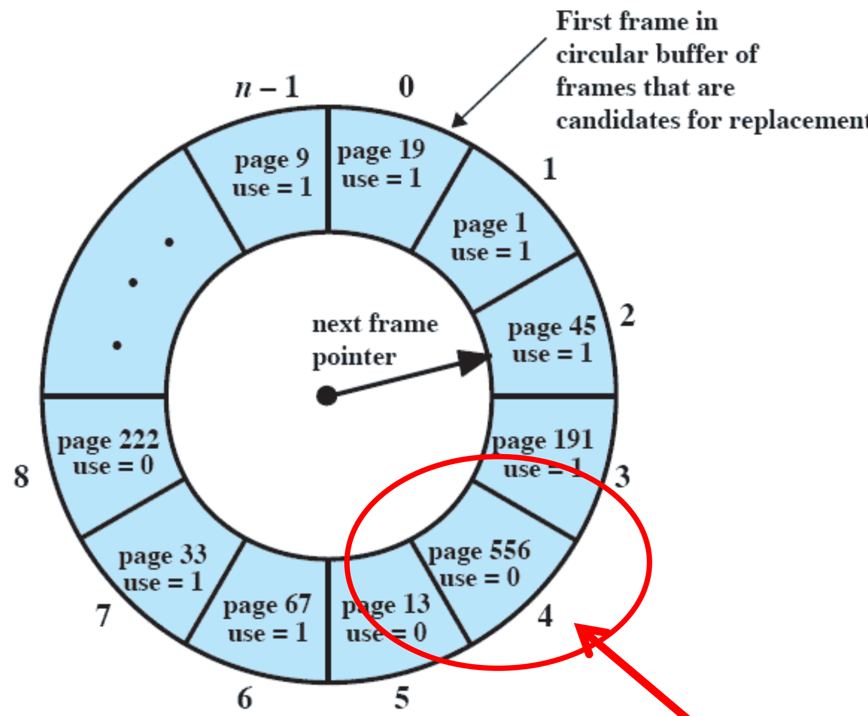
Looking for a page to be replaced

- Initially, all frames in circular buffer are empty, use bit == 0
- Frame pointer: is regarded the “clock hand”, is called the “next frame” pointer
- Procedure for replacing a page / finding unused frame
  - Find page to be replaced (or unused frame):
    - Progress “next frame” pointer to first frame with use bit == 0
    - During this progression: reset use bit of each visited frame entries to 0
    - Load page into the first found frame with use bit == 0, set use bit to 1
  - Move position pointer to next frame
- When the circular buffer is filled with pages, the “next frame” pointer has made one full circle
  - it will be back at the first page loaded
    - points to the “oldest” page

Clock Policy

Find the right Page to be replaced

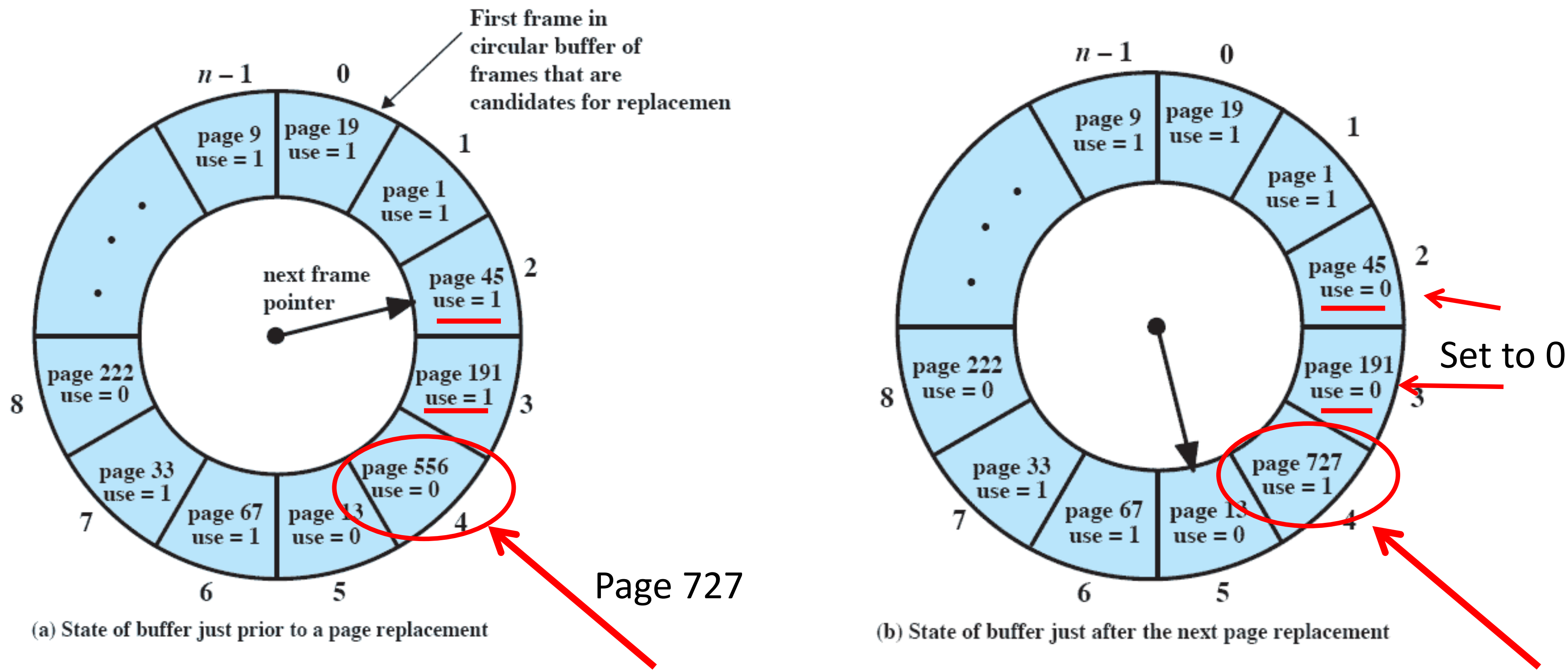
- Finding a page to replace
  - Advance “next frame” pointer from frame to frame, until a frame with “use” bit == 0 is found
  - As long as visited frames have “use” bit == 1, set “use” bit of visited frames to 0, give the pages situated there a ***second chance***
  - Load page into first frame found with “use” bit == 0, set “use” bit to 1
  - Page is replaced, pointer is advanced to *next* frame



(a) State of buffer just prior to a page replacement

***Load action*: set “use” bit = 1**
***Visit action*: reset “use” bit = 0**
Load only, if “use” bit == 0

Clock Policy
Before and after Page Replacement



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

Page-Buffering

- Avoid unnecessary read and write operations
- A replaced page is not lost, but assigned to one of two lists
  - Free page list
    - Add unmodified replaced page to this list
    - When page is needed again, its content is still held in memory, no I/O needed, can be reused immediately
  - Modified page list
    - Add replaced page that was modified, to this list
    - Pages are written to disk in clusters, saves I/O

Resident Set Management

How many Frames shall we allocate
to a process?

Resident Set

- Resident Set:

  The portions of of a process – the set of pages – that can be held in main memory at a particular time

- As long as the process makes memory references only to memory locations that are in the resident set, no page faults occur
- Resident set is restricted by the number of frames allocated to a process

Thrashing

- Thrashing is a situation where a process is repeatedly page faulting
  - The process does not have enough frames to support the set of pages needed for fully executing an instruction
  - It may have to replace a page that itself or another process may need again immediately
- A process is thrashing if it is spending more time on paging than execution

Thrashing

- Performance depends on a minimum number of frames allocated
  - If number of free frames decreases, page faults increase
  - Increase of page faults slows process execution
    - I/O operations
    - Instruction that leads to a page fault has to be restarted, effectively executing instructions twice
- We have to manage the size of the resident set to allocate the “right” amount of frames to a process

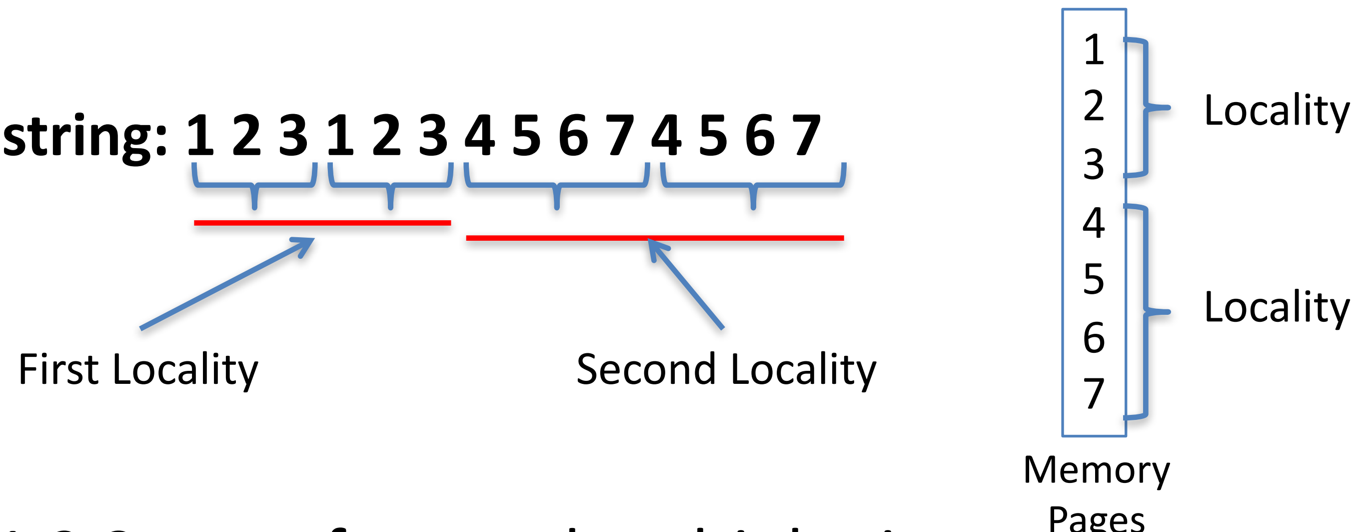
Principle of Locality

- Principle of Locality
  - A “locality” is a set of pages that are **actively** used together during a time period by a program
- A program may have **several different localities**
  - As a process executes, it moves from locality to locality
- If we want to prevent thrashing, we must provide a process with as many frames as needed to accommodate a “locality” – the “working set”
  - The resident set should be allowed to be of a size so that page faults are minimised

Localities

- A program may have **several different localities**
  - As a process executes, it moves from locality to locality
- Let’s consider the following page reference string:

Page reference string: 1 2 3 1 2 3 4 5 6 7 4 5 6 7

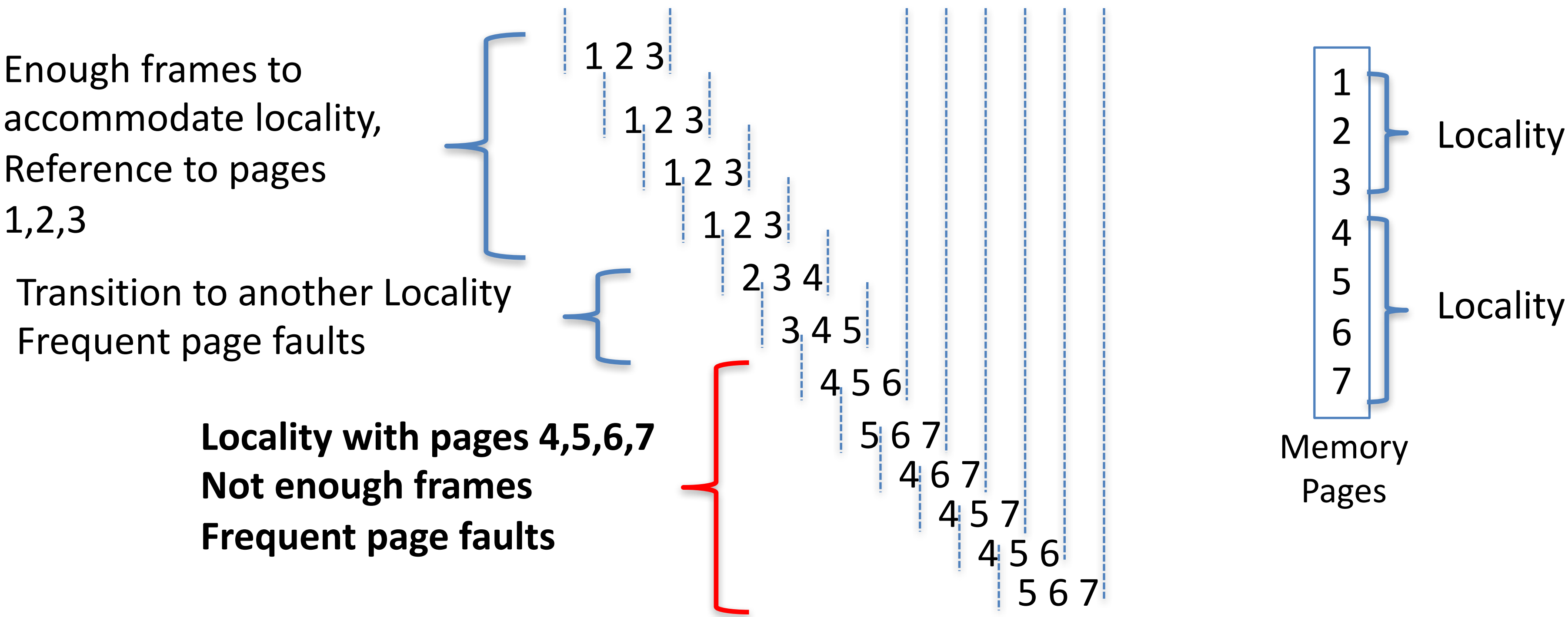


- First, pages 1,2,3 are referenced multiple times
- Then, the process switches to another locality, referencing pages 4,5,6,7

Accommodating the Locality

- Let’s assume that we allocate **3 frames** as the resident set:

Page reference string: 1 2 3 1 2 3 4 5 6 7 4 5 6 7



Working Set

- Working Set

  Set of pages a process is currently working with

- The working set of a process at current time ***t*** is the set of pages the process has been referencing for the past **Δ** time steps

- The *working set* can be larger than the *resident set* – some pages of the working set may have to be swapped in and out
- **A working set is an approximation of a program's locality**
- It is the set of pages actively used by a process and all these pages should be held in memory
- The size of this set gives us an estimate how many frames should be allocated to a process – how large the *resident set* should be

Tune the Observation Window

- Let's assume, we have 7 pages and the following reference history:

Look into the past

Page reference string: 1 2 3 1 2 3 4 5 6 7 4 5 6 7

- If we choose a working set window Δ = 3, then we only see part of the locality, which gives the impression that we only need 3 frames for pages 5, 6, 7
- If we choose a Δ = 10, we see part of the previous locality, which give the impression that we need 6 frames for pages 2, 3, 4, 5, 6, 7

Working Set Window

- We look into the past to estimate the future size of the "working set"
  - Set of pages that have been referenced in the recent past of a process execution
- How far to look into the past:
  - Define an observation window or "working set window" Δ
- Parameter Δ : number of most recent page reference observations
  - If Δ is too small, not all actively used pages (the current "locality") will be in the working set window
  - If Δ is too large, we may look back too far and see several different localities
    - Does not give us clear estimate how many frames we actually need for this process

Working Set

- The working set may change with each page reference
- How far shall we look into the past?
- Let's assume that we allocate **3 frames** to the process:

Δ ← t

Page reference string: 1 2 3 1 2 3 4 5 6 7 4 5 6 7

Enough frames to accommodate locality, Reference to pages 1,2,3

1 2 3
1 2 3
1 2 3
1 2 3

Transition to another Locality
Frequent page faults

2 3 4
3 4 5

Locality with pages 4,5,6,7
Not enough frames
Frequent page faults

4 5 6
5 6 7
4 6 7
4 5 7
4 5 6
5 6 7

1
2
3 — Locality
4
5
6
7 — Locality

Memory Pages

Working Set Strategy

- We need to ensure that we can keep the working set in memory:
  - Working set determines the minimum size of the resident set to avoid thrashing
  - If the working sets of all processes exceed total number of page frames, one or more processes have to be swapped out
- We can monitor the working set over time, provides insight in the required amount of page frames

Working Set Size WSS

- Working set size WSS_i
  - Size of the working set for process *i*
  - Depends on how many past observations Δ are taken into account
  - We assume that the pages accessed by process *i* in the observation window Δ are most likely to be accessed in the future as well
    - approximate the future working set
  - the number of pages observed determines the number of frames to be allocated
- Therefore, a process needs at least WSS_i frames to avoid thrashing
- Operating system monitors the working set of a process and allocates WSS_i frames to the process

Working Set

- What to do if the Resident set is too small
  - Danger of thrashing !
  - Consult the “working set”: Add more frames
  - If we run out of frames, suspend process and try to allocate again later
- What to do if the Resident Set is too large
  - There may be pages that are loaded into memory, but haven’t been referenced for a long time
  - Periodic discarding of pages from the resident set that are not in the working set

Working Set Strategy, Problem

- Problems
  - Page references have to be logged
  - Page references have to be ordered
  - Optimal time window size Δ is unknown at runtime and may vary
- Simplification
  - Monitor number of page faults per time unit per process, instead of monitoring the working set

Page-Fault Frequency

- Use page-fault frequency to control thrashing
  - Thrashing has a high page fault rate
- Idea
  - If page-fault rate is high, then process needs more frames
  - If page-fault rate is low, then process has too many frames
- We can establish upper and lower bounds on the desired page-fault rate, controls frame allocation
  - If actual page-fault rate exceed upper limit, process receives an extra frame
  - If actual page-fault rate falls below lower bound, process looses one frame
- If there are no more frames to allocate, a process exceeding the upper bound, will be suspended and its frames reallocated to other processes

Allocation Strategies

Frame Allocation to Processes

- Each process is allocated free memory
  - Each process has a Free-frame list
  - Demand-paging will allocate frames from this list due to page faults
  - When the free-frame list is exhausted, a page replacement algorithm will choose a page to be replaced
- With multiple processes: how many free frames should be allocated to each process?

Resident Set Management

	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.	•Not possible.
Variable Allocation	•The number of frames allocated to a process may be changed from time to time to maintain the working set of the process. •Page to be replaced is chosen from among the frames allocated to that process.	•Page to be replaced is chosen from all available frames in main memory; this causes the size of the resident set of processes to vary.

Resident Set Allocation

- Fixed-allocation
  – A process has a fixed pool of frames in main memory
  – If this pool is exhausted, a page fault will cause a page replacement
- Variable-allocation
  – Allow the pool of frames to vary in size over the lifetime of a process

Replacement Scope

- We can classify page replacement in terms of allocation behaviour
  - Global page replacement
    - A process selects a frame for replacement from a global list of frames (frames of all processes)
    - Competition with other processes: A page replacement will interfere with another process’ memory allocation
  - Local page replacement
    - A process can only select a frame for replacement from its own local list of frames

Replacement Scope

- Disadvantages
  - Global page replacement
    - Performance of a process depends on external circumstances, as other processes may replace its pages
    - Waste of performance
  - Local page replacement
    - Process is restricted to its own allocation of physical memory
    - Even if memory is not used by other processes, it is not available globally
    - Waste of resources
- Global replacement generally results in better system throughput and is the more commonly used method

Thrashing

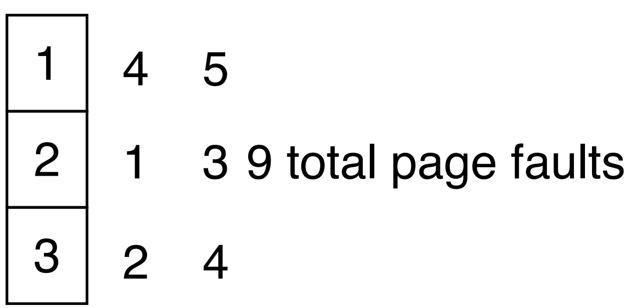
- Limit the effects of thrashing by using a local replacement algorithm
  - Processes can only be served from their allocated pool of free memory frames
  - Process cannot “steal” frames from other processes
- Problem
  - When process starts thrashing, it will occupy the paging device queue most of the time
  - This increases the time for other processes to be served by the paging device

Working Set

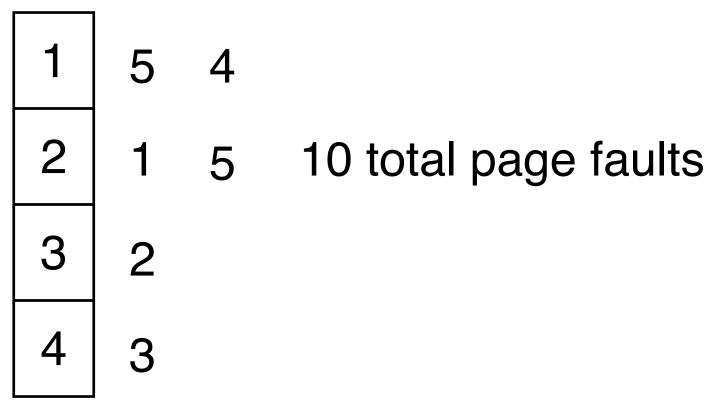
- How many processes can be started?
  - Operating system will start processes and allocate frames as needed by their working sets
  - Working sets can change in size and need for frames
  - If the sum of all needed frames exceeds the actually available number of frames, one of the processes is selected for suspension
    - Its pages are swapped out and its frames reallocated to other processes
  - The suspended process can be restarted later, when enough frames become available again

FIFO: Belady's Anomaly

- Problem of FIFO
  - Does not indicate how heavily a page is actually used
- Possible solution: increase number of physical frames
  - FIFO shows Belady's anomaly: the page-fault rate increases, when the number of frames is increased
- Example
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 frames vs 4 frames:



3 Frames, 5 pages



4 Frames, 5 pages