Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T15

Beyond Physical Memory
Swapping Policies

Referência principal
Ch.22 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 17 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

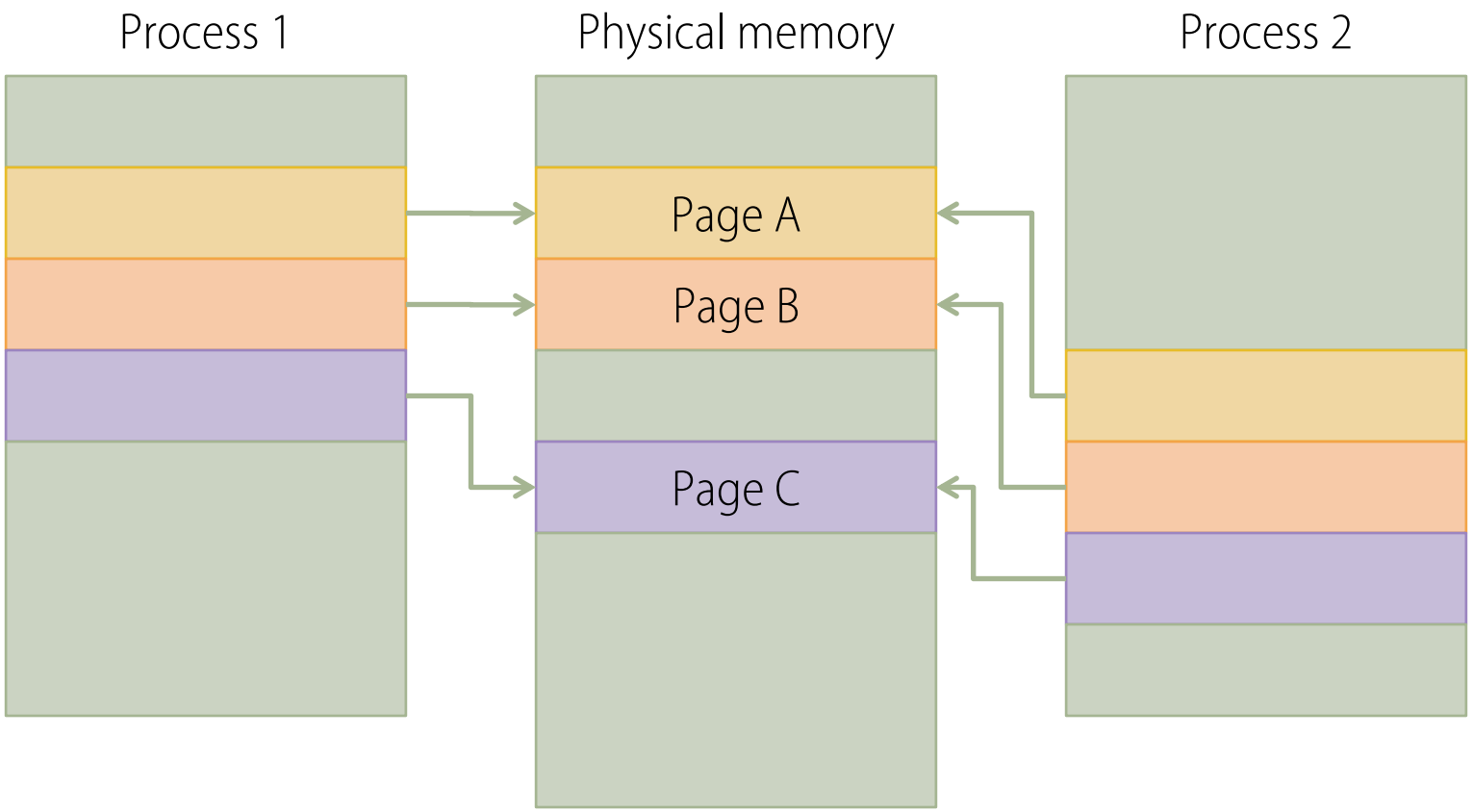
Beyond Physical Memory: Policies

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

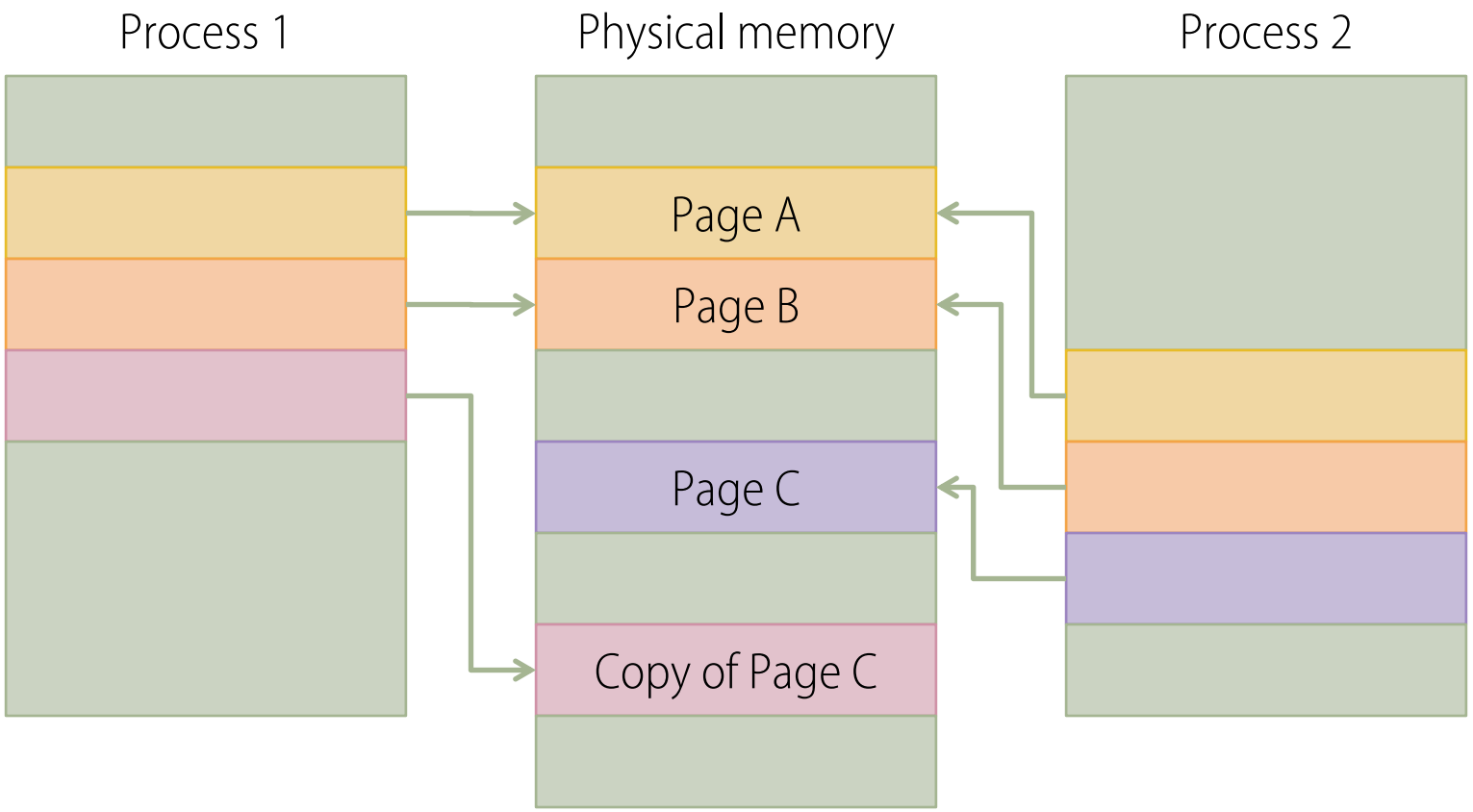
Process Creation: Copy-on-Write

- **Copy-on-write** provides for rapid process creation and minimizes the number of new pages that must be allocated to a new process.
  - Parent and child processes initially share same pages in memory.
  - When either process modifies a shared page, that page is copied.
- COW allows more efficient process creation as only modified pages are copied.
- Free pages are usually allocated by the OS from a **pool** of **zero-filled on-demand pages**.

Before Process 1 Modifies Page C



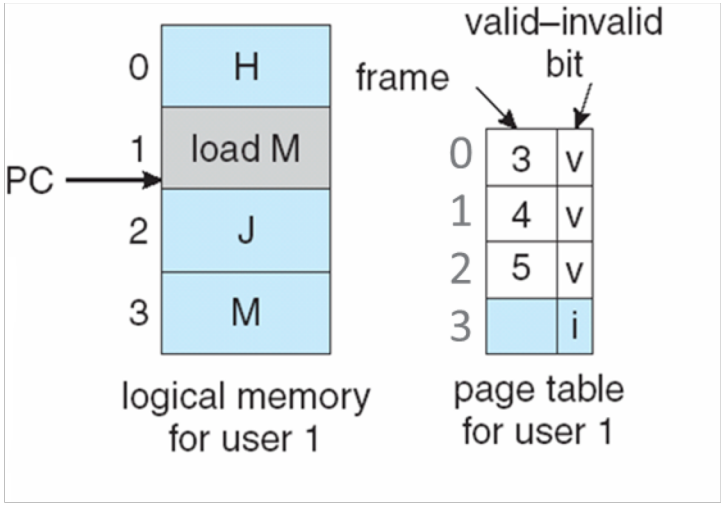
After Process 1 Modifies Page C

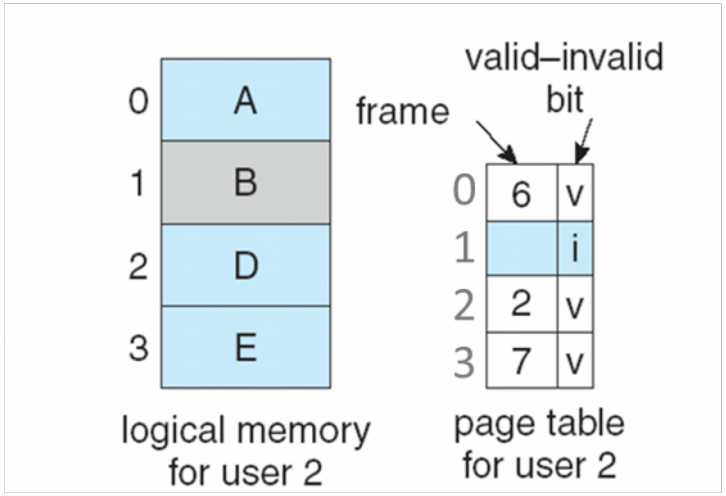


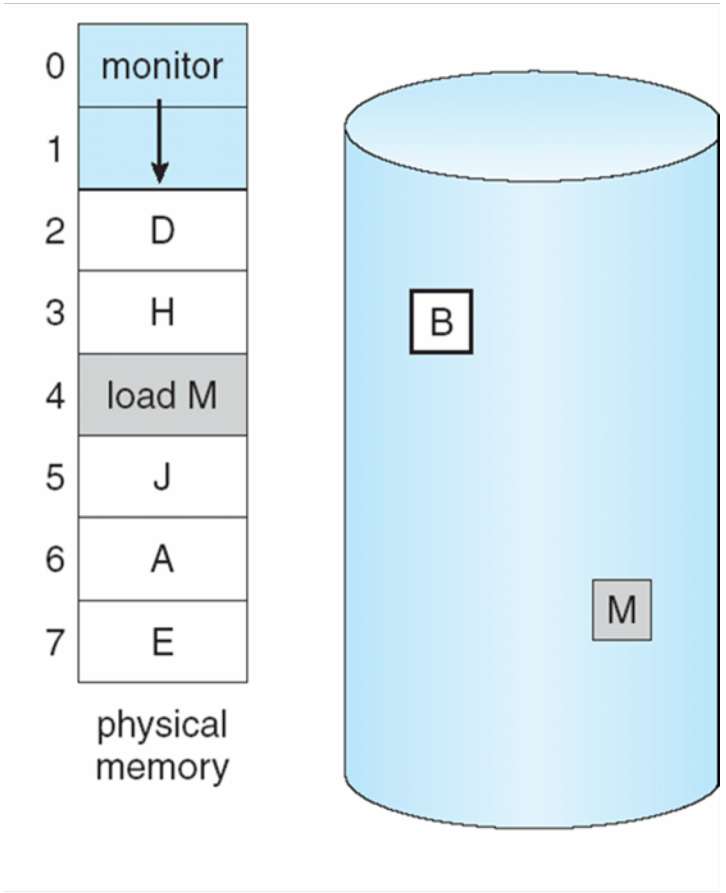
Page Replacement

- When the degree of multiprogramming is increased, OS usually **over-allocates** memory.
- What happens when a page fault occurs and there is no free frame?
  - Find some page in memory, but not really in use, and swap it out.
- Page replacement
  - An algorithm is wanted which will result in minimum number of page faults.
  - The same page may be brought into memory several times.

Over-allocation and Need For Page Replacement







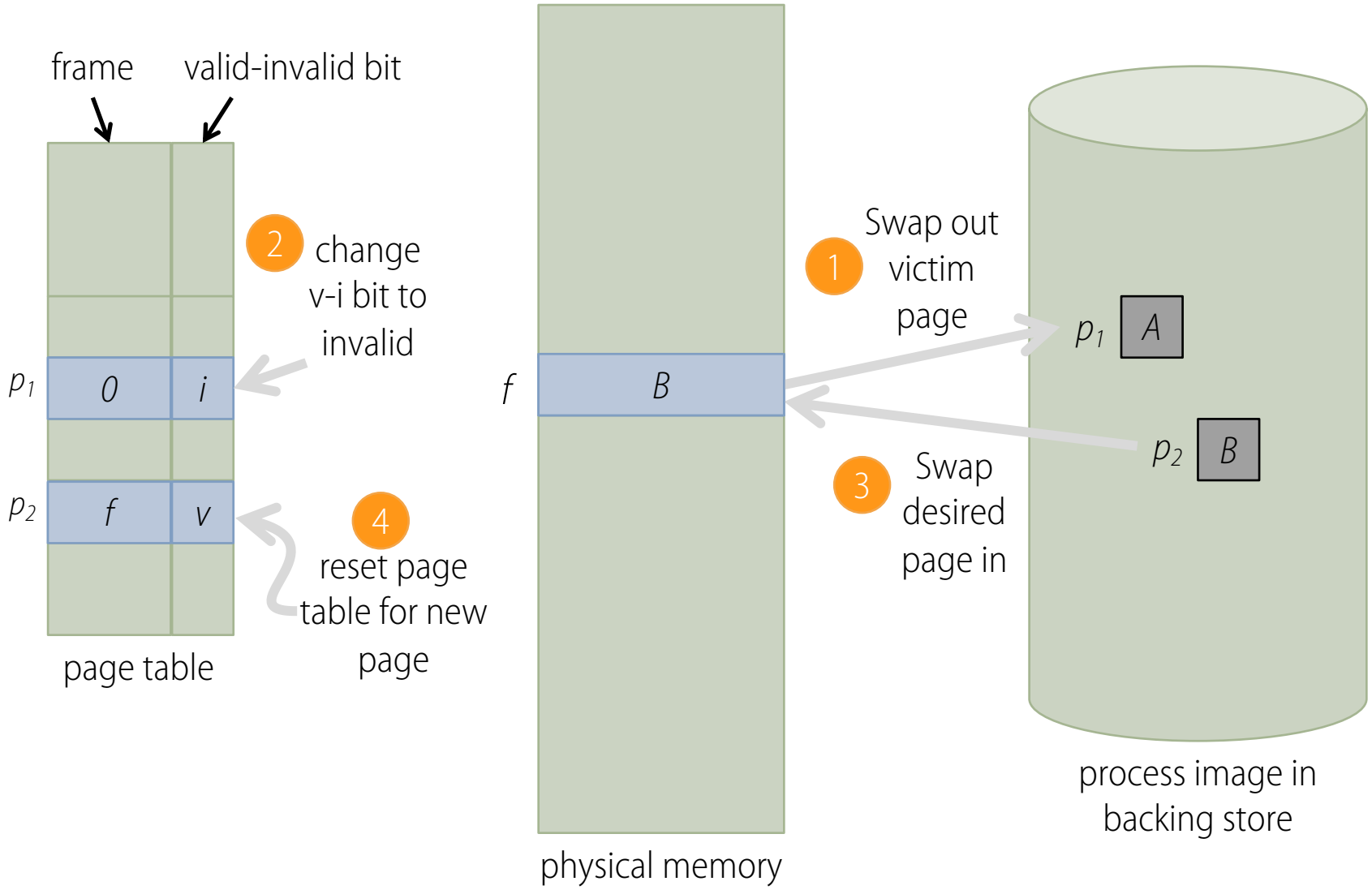
Page Replacement

- Deal with over-allocation of memory by modifying page-fault service routine to include page replacement.
- Page replacement completes separation between logical memory and physical memory
  - A large virtual memory can be provided on a much smaller physical memory.
- When replacing pages in memory a **modify bit** is used to reduce overhead of page transfers.
  - Only modified pages are written to disk.

Basic Page Replacement

1. Find the location of the desired page on disk.
2. 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**.
   - If the victim frame's modify bit is set, write it to disk; update the page and frame tables.
3. Bring the desired page into the free frame; update the page and frame tables.
4. Restart the process.

Page Replacement



Cache Management

- The goal in picking a replacement policy for this cache is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the *average memory access time (AMAT)*.

$$AMAT = (P_{hit} * T_M) + (P_{miss} * T_D)$$

Argument	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{hit}	The probability of finding the data item in the cache (a hit)
P_{miss}	The probability of not finding the data in the cache (a miss)

The Optimal Replacement Policy…

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed *furthest in the future*…

    … which would result in the fewest-possible cache misses.

- Since we cannot predict the future…

  … it serves only as a comparison point, to know how close we are to perfection.

Page Replacement Algorithms

- In general, we want the lowest page-fault rate.
- A page replacement algorithm can be evaluated by
  - running it on a particular string of memory references (**reference string**) and
  - computing the number of page faults on that string.

Creating Reference Strings

- An initial reference string can be created artificially or by tracing a given system and recording the address of each memory reference.
- Since we are interested only in page faults, the length of the reference string can be reduced if we take into account that
  - For each reference, only the page number needs to be considered.
  - If there is a reference to page *p*, any immediately following references to *p* can be dropped, since they will never cause a page fault.

Example: Creating a Reference String

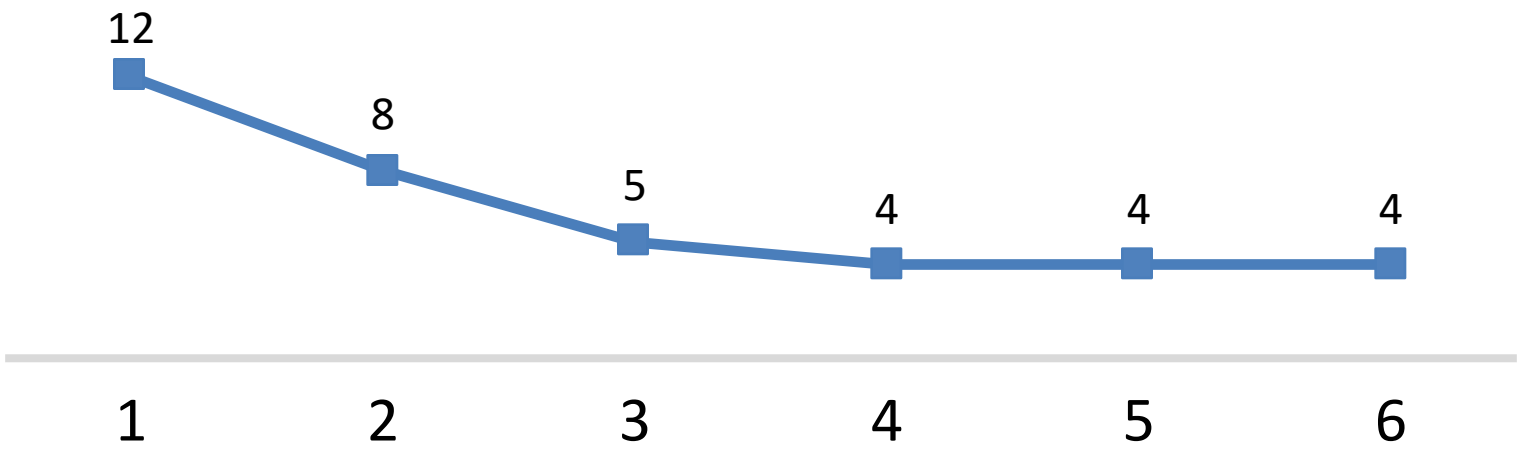
- Assume
  - A 12-bit paged virtual memory with eight 512B-pages.
  - A process trace which produced the following reference string:
    - 1000, 4032, 1001, 6012, 1002, 1003, 1004, 1001, 5010, 1002, 1003, 1004, 1001, 6010, 1002, 1003, 1004, 1001, 6007, 1002, 1005, 5010
- To analyze the page faults, the string can be reduced to
  - 1, 4, 1, 6, 1, 1, 1, 1, 5, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1, 5
- And, then, further reduced to
  - 1, 4, 1, 6, 1, 5, 1, 6, 1, 6, 1, 5

Number of Page Faults vs Number of Frames

- Assume
  - A first-in first-out replacement policy
  - Reference string: 1, 4, 1, 6, 1, 5, 1, 6, 1, 6, 1, 5

For number of frames = 2

1	4	6	1	5	6	**1**	***5***



Classical Page Replacement Algorithms

- First-In First-Out
- Optimal
- Least-Recently-Used
- Least-Recently-Used Approximation
- Counting-Based

- To illustrate the algorithms we will use a memory with 3 frames and the reference string
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

FIFO Replacement

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

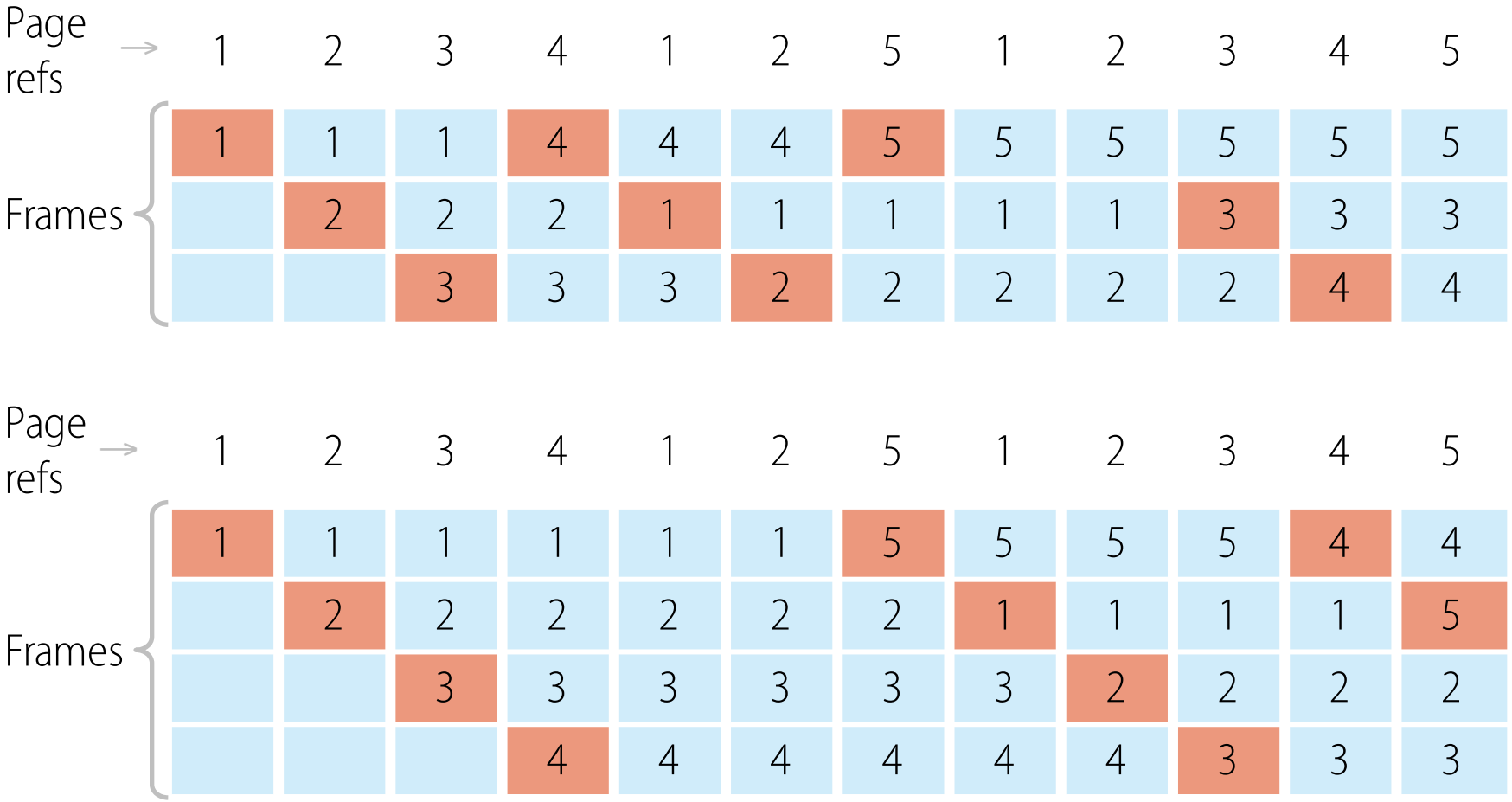
page frames

FIFO Replacement: Belady's Anomaly

Page refs →	1	2	3	4	1	2	5	1	2	3	4	5
Frames	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4

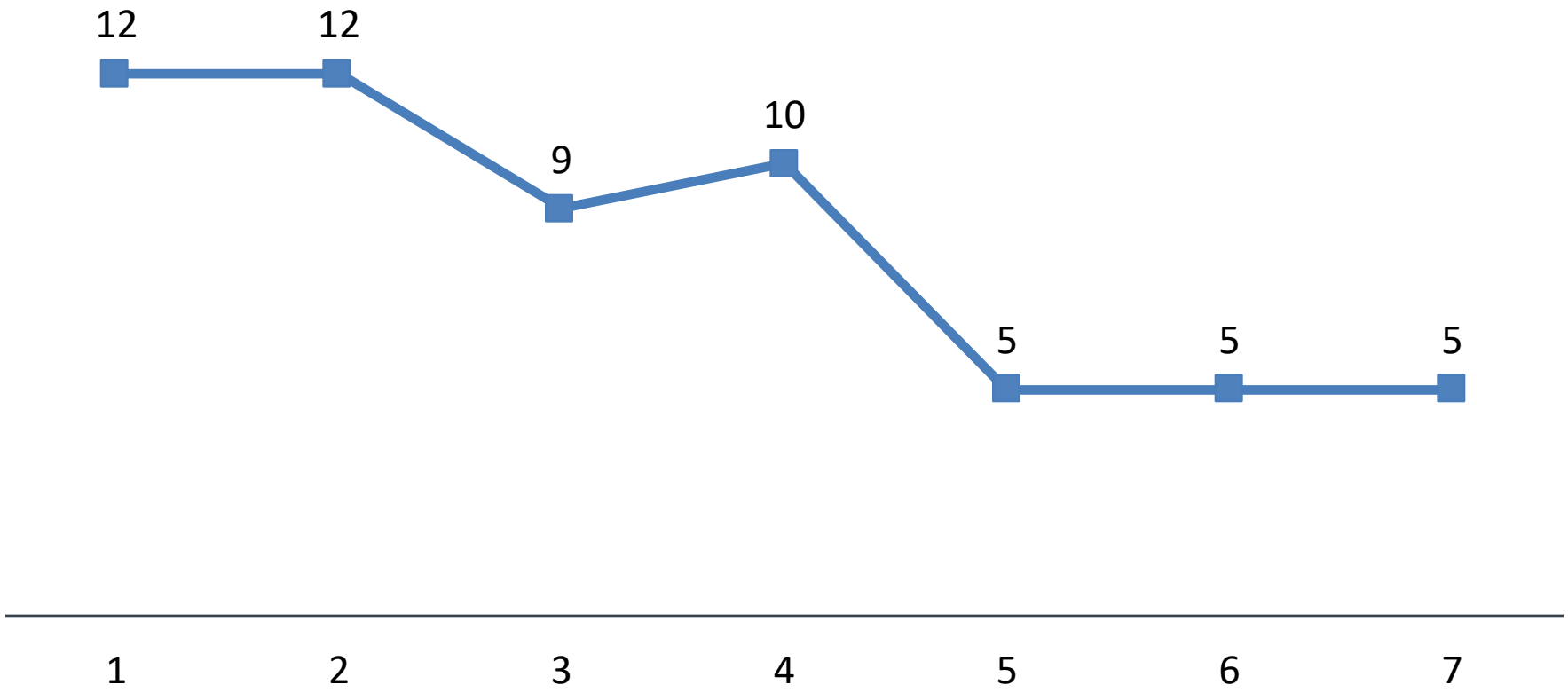
Page refs →	1	2	3	4	1	2	5	1	2	3	4	5
Frames	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3

FIFO Replacement: Belady's Anomaly



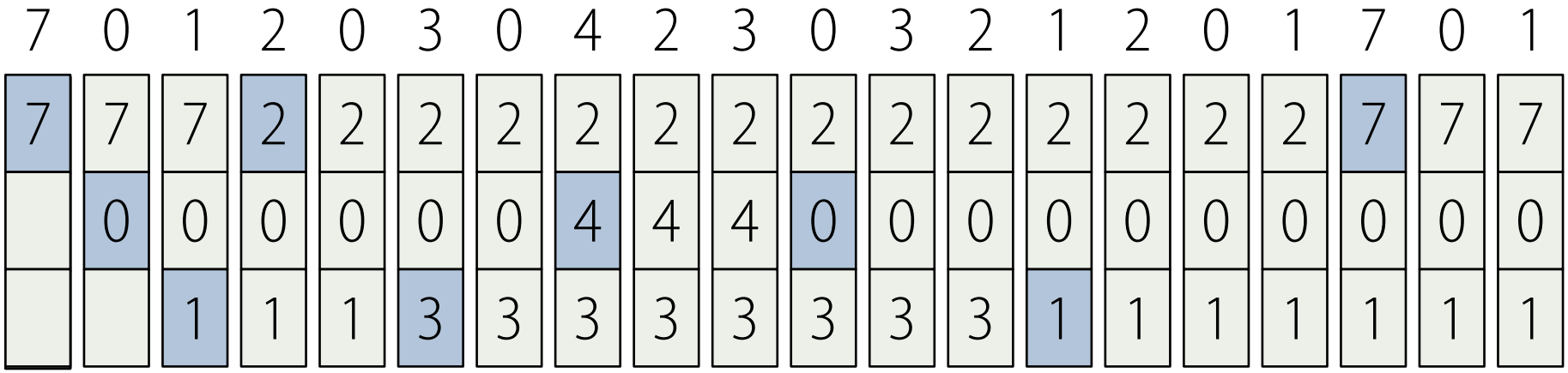
Graph of Page Faults vs Number of Frames

- Assuming the following reference string
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



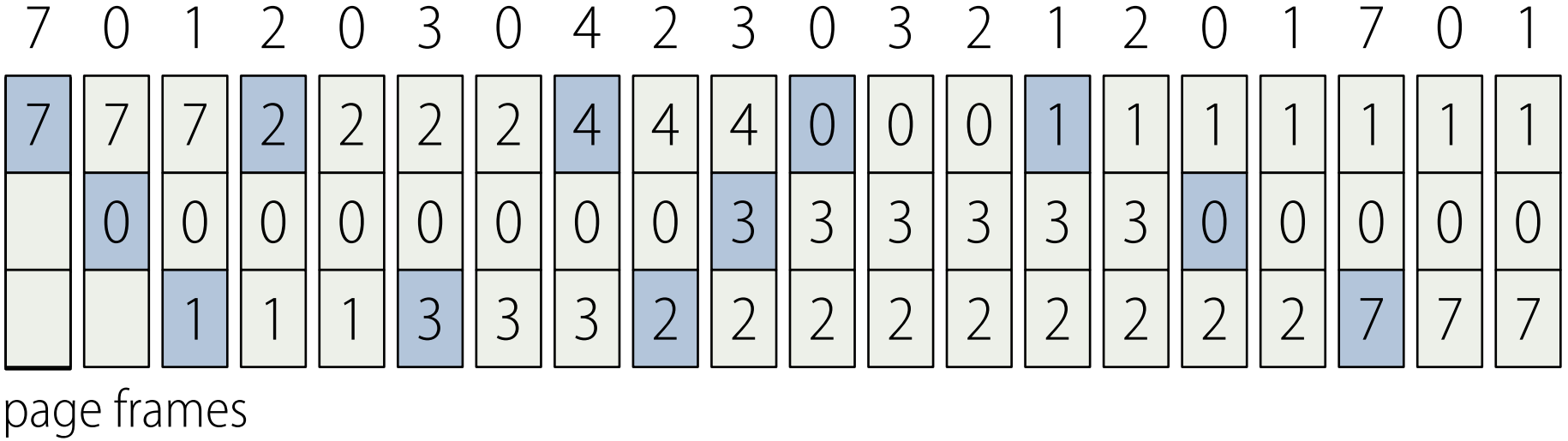
Optimal Algorithm

- Replace the page that will not be used for longest period of time



- What is the purpose of this since we do not know the future?
  - It is useful for measuring how well our algorithms perform.

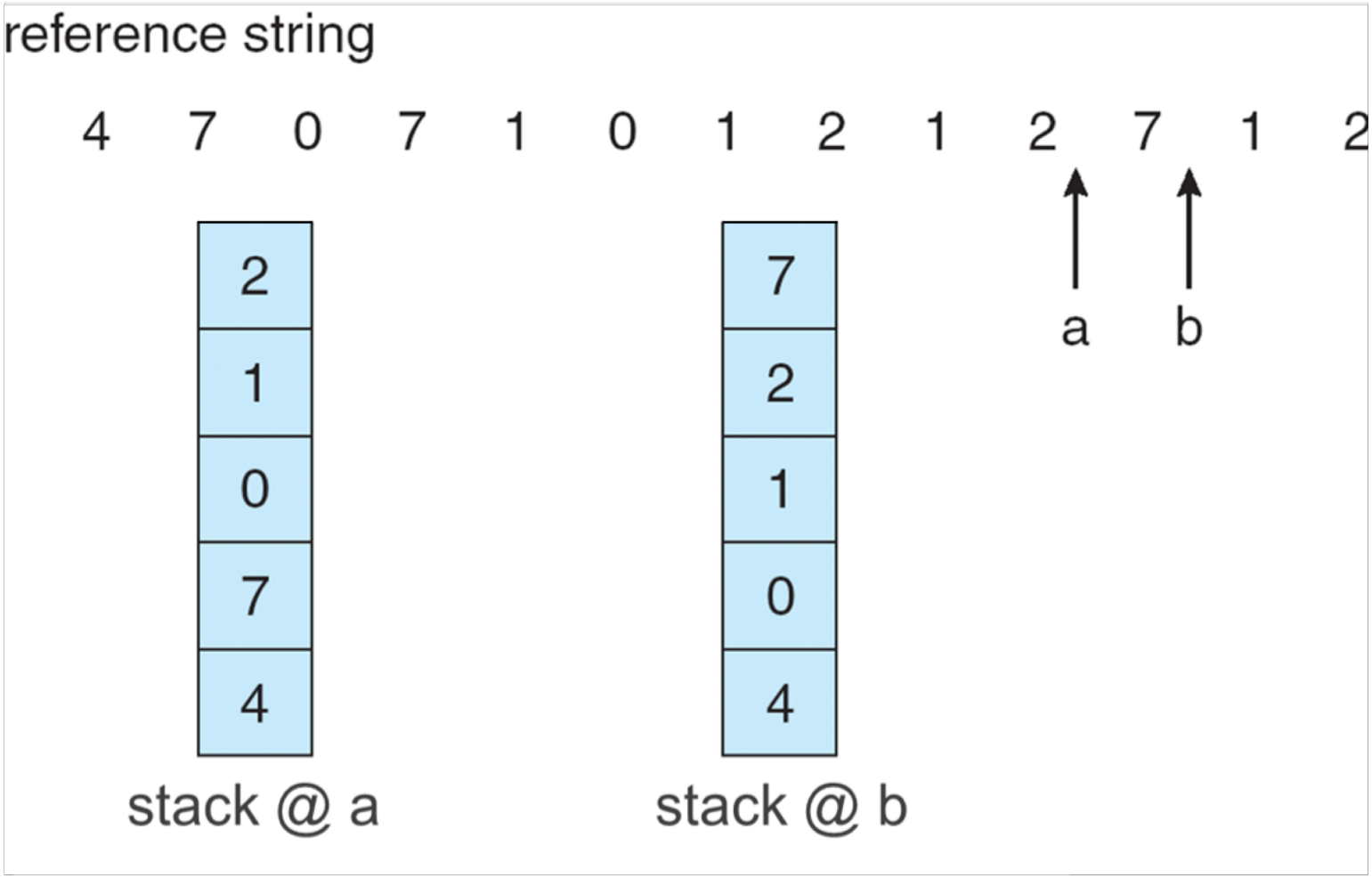
Least Recently Used (LRU) Algorithm



Least Recently Used (LRU) Algorithm

- Counter implementation
  - Every page entry has a counter; every time page is referenced, copy the clock into the counter
  - When a page needs to be swapped in, look at the counters to determine which page will be swapped out
- Stack implementation
  - Keep a stack of page numbers in a doubly linked list.
  - When a page is referenced
    - move it to the top
    - 6 pointers must be updated
  - No search for replacement page

Use of a stack to record the most recent page references in LRU



LRU Approximation Algorithms

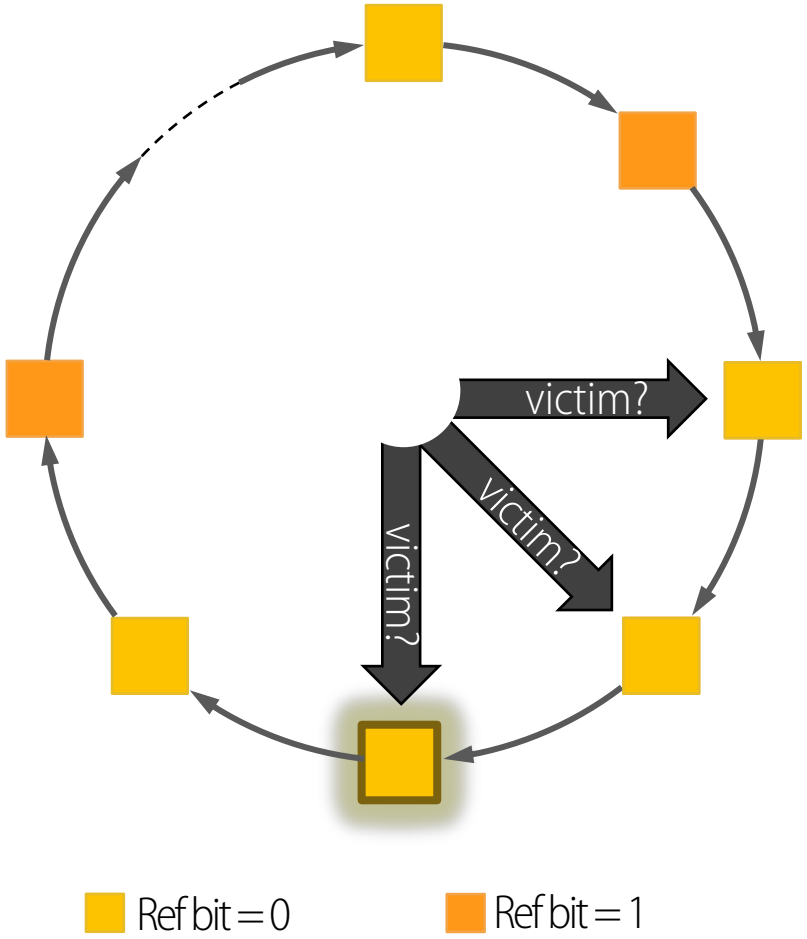
- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on every memory reference.
  - Add a little bit of hardware support.
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced, set bit to 1
  - Replace a page whose reference bit is 0 (if one exists)
    - However, the order of the references is not known.

LRU Approximation Algorithms

- Additional reference bits
  - Uses reference bit plus a one byte register for each page in a table in memory.
  - At regular intervals a timer interrupt transfers control to the OS.
  - For each page, the OS shifts the register right by one bit and copies the reference bit into the high order bit of the register.
  - When needed, interpret the registers as unsigned 8-bit integers and replace the page with the lowest one.
    - If more than one page have the lowest value, we can either swap out all of them or use FIFO.

LRU Approximation Algorithms

- Second Chance (Clock)
  - Uses reference bit
  - Clock replacement
    - If page to be replaced (in clock order) has reference bit = 1
      1. `set reference bit to 0`
      2. `leave page in memory`
      3. `try to replace the next page (in clock order)`



LRU Approximation Algorithms

- Enhanced Second Chance
  - Uses reference bit and modify bit as an ordered pair.
  - These two bits generate four possible cases
    - (0, 0) = neither recently used nor modified
    - (0, 1) = not recently used but modified
    - (1, 0) = recently used but not modified
    - (1, 1) = recently used and modified
  - Execution pattern is the same as in the clock algorithm.
  - We replace the first page encountered in the lowest non-empty class.

Counting-Based Algorithms

- Keep a counter of the number of references that have been made to each page

- LFU (Least-Frequently Used) Algorithm
  - Replaces page with smallest count

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

Additional Procedures

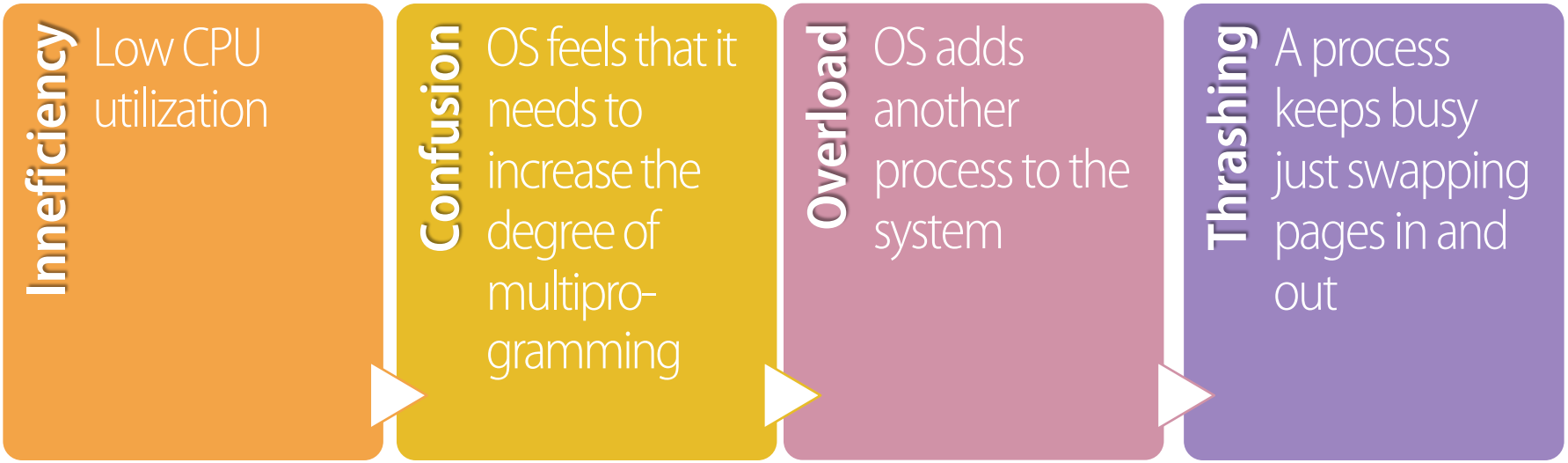
- Keep a pool of free frames
  - A victim frame is chosen using any method
  - The desired page is read into a frame taken from the pool without waiting for the victim to be written out.
  - When the victim has been written out, its frame is added to the free frames pool.
- Pool of free frames with “memory”
  - Keep a pool of free frames but remember which page was in each frame.
  - If the page is required again, use it directly.

Additional Procedures

- Keep a list of modified pages
  - Whenever the paging device is idle, write out one page from the list and reset its modify bit.

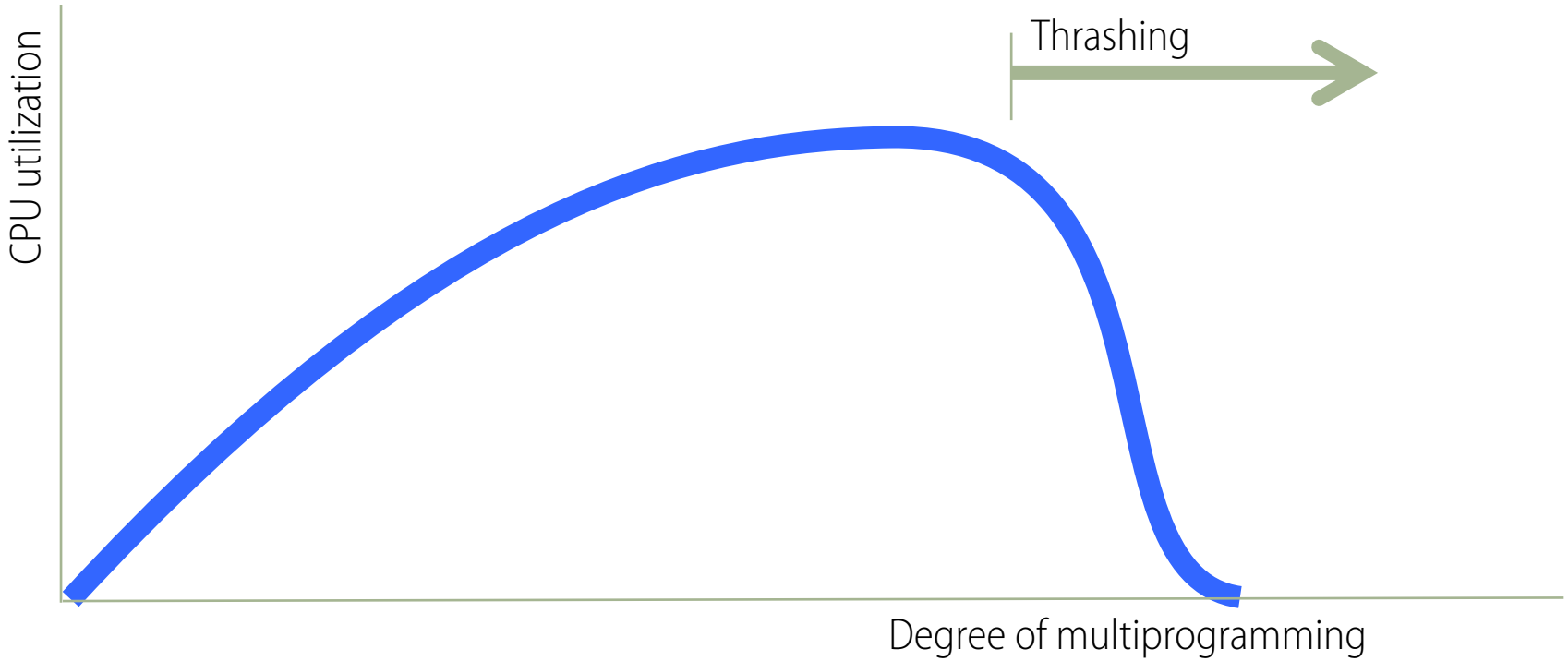
Thrashing

- When a process does not have "enough" pages the page-fault rate can become very high.
- This may lead to



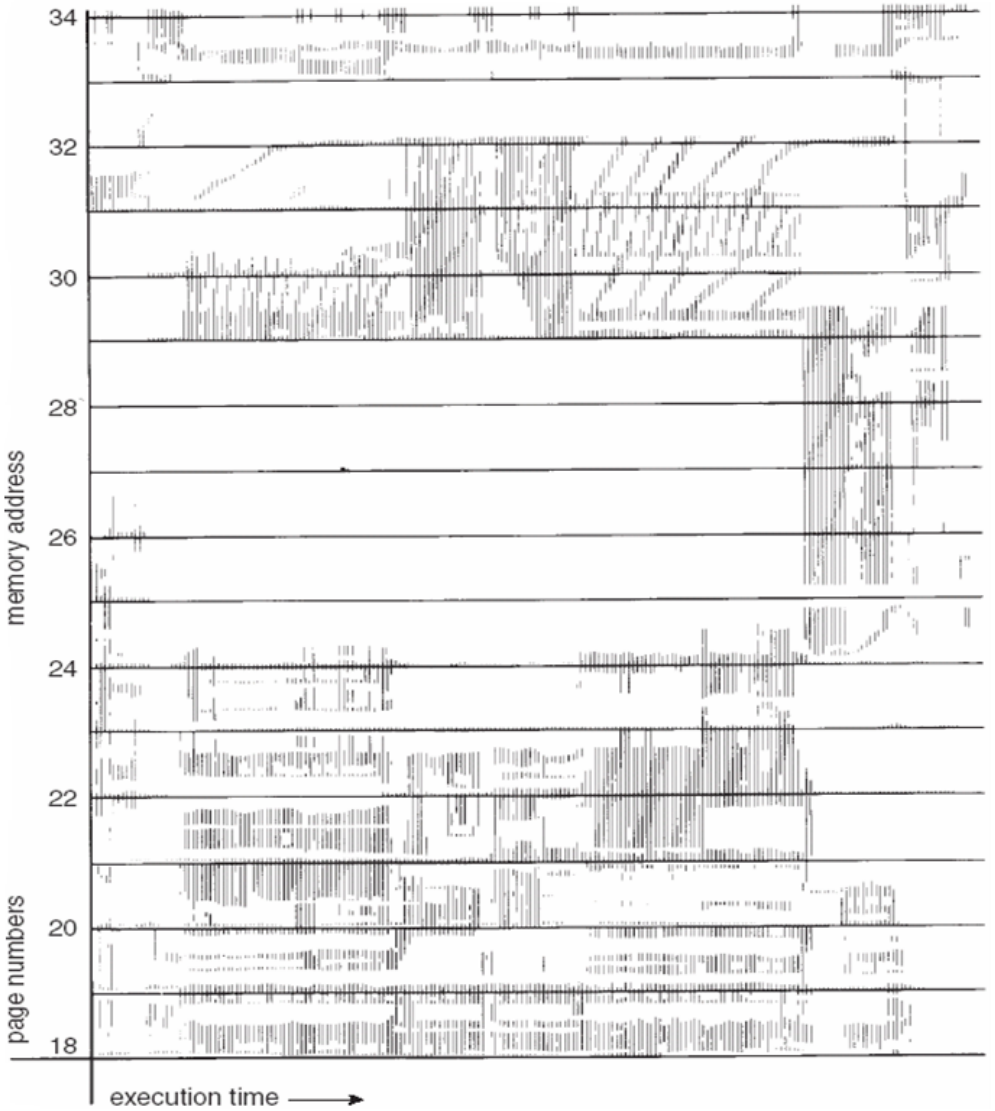
Thrashing

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.
  - Decide not to run a subset of processes.
  - Reduced set of processes working sets fit in memory.



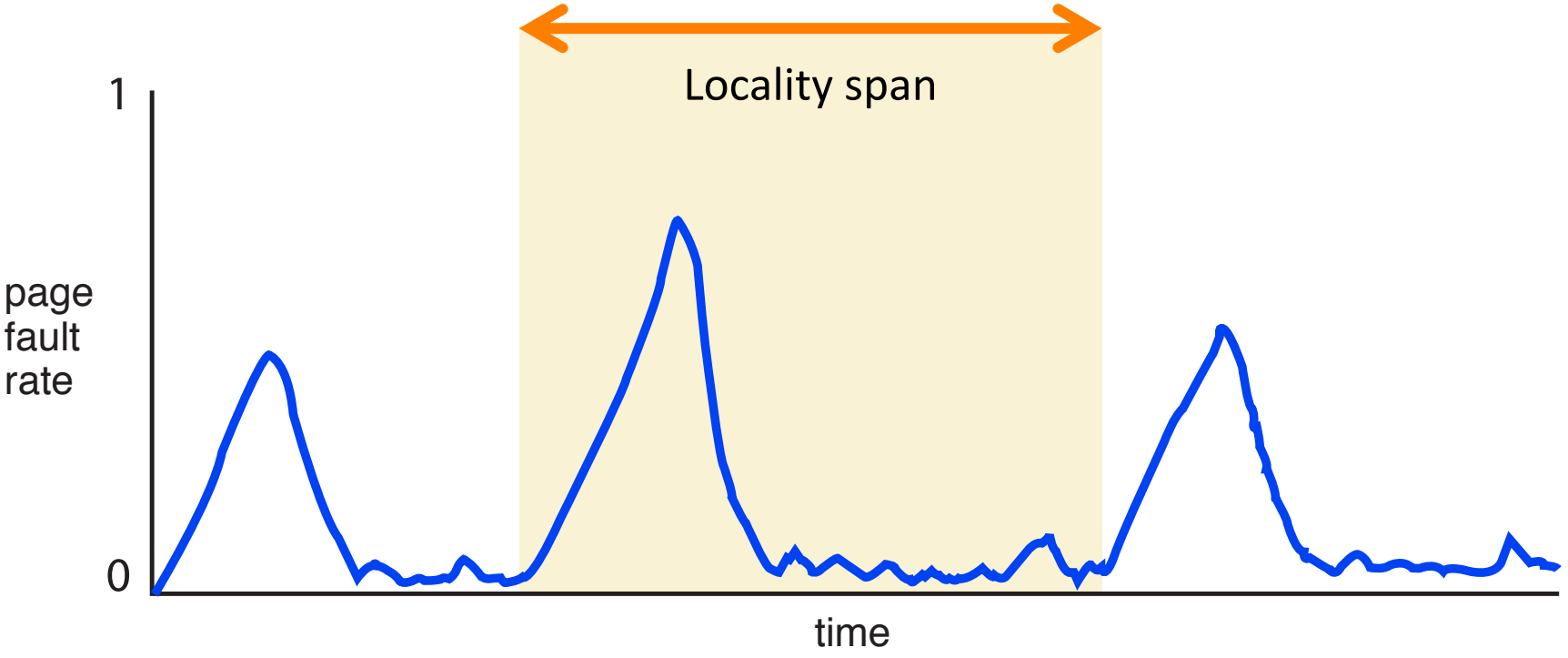
Demand Paging and Thrashing

- Why does demand paging work?
- Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - Σ (size of localities) > total memory size



Localities and Page Fault Rates

- While a process executes within a locality its page fault rate is low.



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10.000 instructions

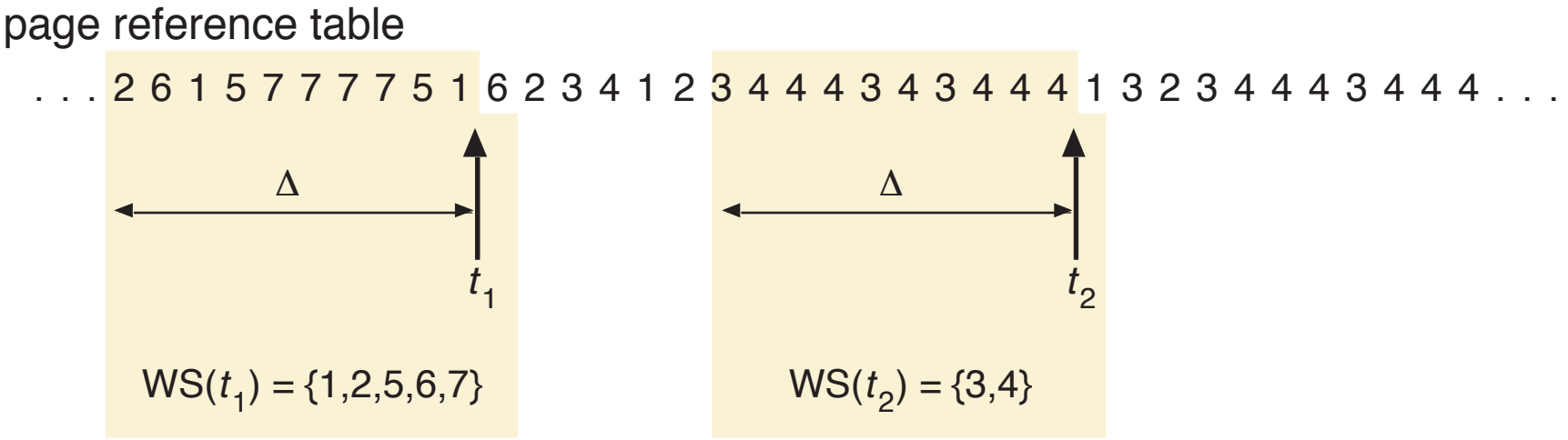
- WSS_i (working set size of process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ is too small it will not encompass entire locality
  - if Δ is too large it may encompass several localities
  - if $\Delta = \infty$ it will encompass entire program

Working-Set Model

- Total demand for frames
  - $D = \Sigma\ WSS_i$
- Total number of available frames = m
- If $D > m$, thrashing will occur because some processes will not have enough frames.
- Policy
  - OS monitors the WS of each process and allocates enough frames to each.
  - If there are extra frames, another process can be started.
  - If $D > m$, one of the processes will be suspended.

Working-set model

- The difficulty with the model is keeping track of the working-set because its window is a moving one.
- Example:

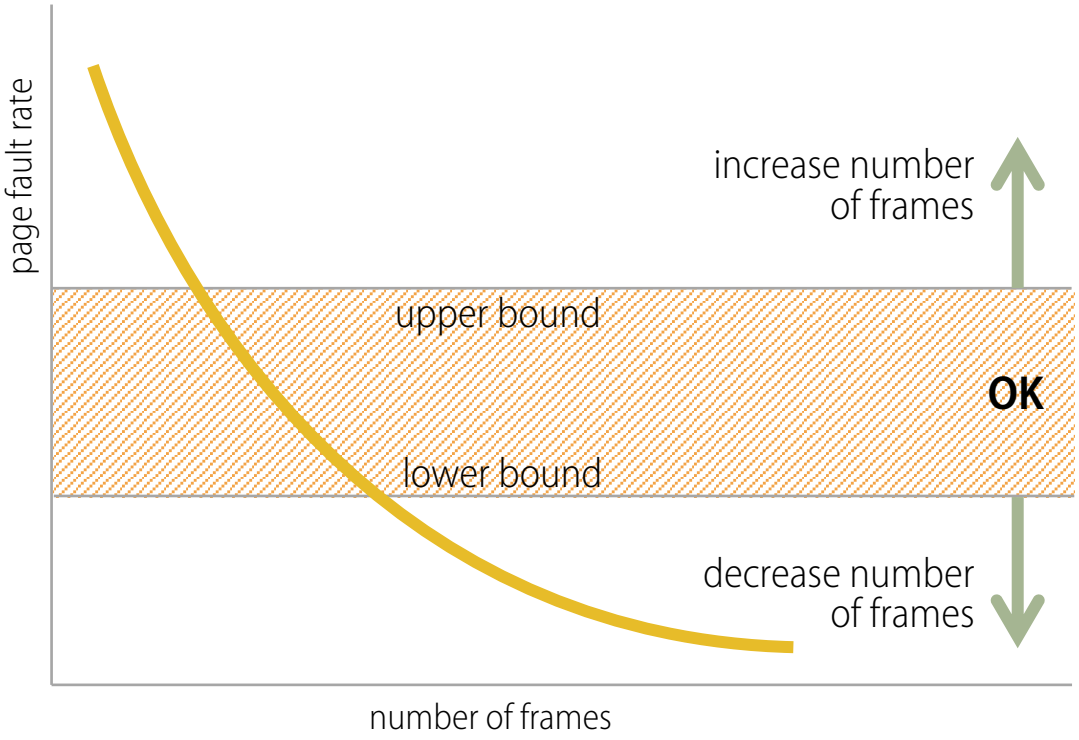


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5,000 time units
  - Keep in memory 2 history bits for each page
  - Whenever a timer interrupt occurs, for each page, shift the history bits left, add the reference bit and set it to 0.
  - If (history bits) $\neq 0$, the page is considered in the WS.
- Why is this not completely accurate?
  - The uncertainty can be reduced by increasing the number of history bits and the frequency of interrupts.

Page-Fault Frequency Scheme

- An alternative to the working-set scheme.
- Establish "acceptable" page-fault rates
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Allocation of Frames

- System architecture defines a *minimum* number of frames per process in order to guarantee that it can execute e.g.
  - Instructions which operate on blocks of memory
  - Indirect addressing
- Two major frame allocation schemes
  - Fixed allocation
    - Equal allocation
    - Proportional allocation
  - Priority allocation

Fixed Allocation

- Equal allocation
  - Give to each process an equal share of the number of frames available.
- Proportional allocation
  - Allocate available frames proportionally to the size of each process.
  - A process' share cannot be lesser than the minimum number of frames per process prescribed by the architecture.

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 or
  - Select for replacement a frame from a lower-priority process.

Global vs. Local Allocation

- Global replacement
  - Replacement frame is selected from the set of all frames.
  - One process can get a frame from another.
  - Processes cannot control their own page-fault rate.
- Local replacement
  - Each process gets frames from only its own set of allocated frames.
  - Process may be hindered by not having access to other, less used, frames.

Other issues: Non-Uniform Memory Access

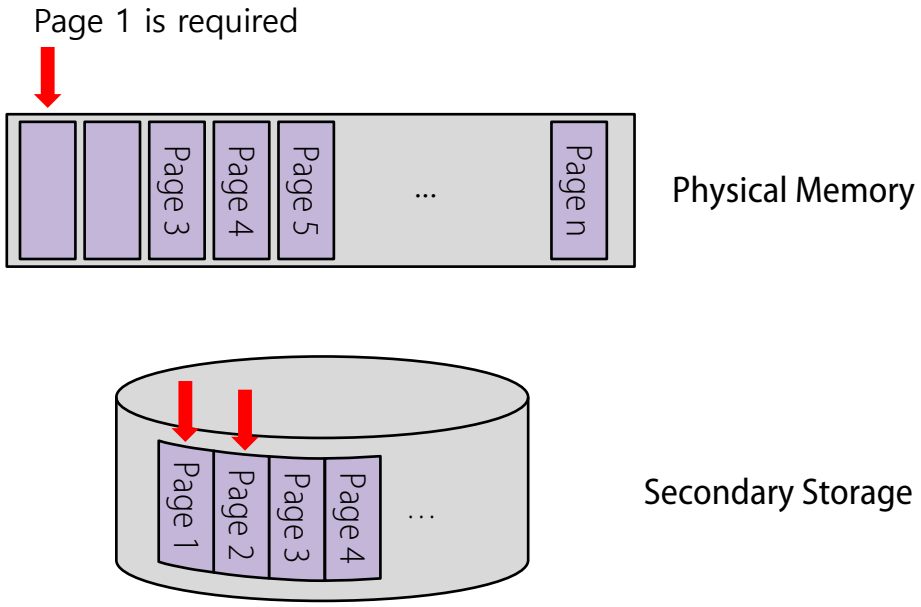
- In systems with multiple CPUs a given CPU can access some regions of main memory faster than it can access others.
- Systems in which memory acccess times vary significantly are known as NUMA (non-uniform memory access) systems.
- In such systems, frame allocation algorithms must take into account the "distance" between the frame and the CPU.
- This is more complicated when threads are used, because these may be scheduled to CPUs in many different system boards.

Other Issues: Prepaging

- Prepaging is intended to reduce the large number of page faults that occurs at process startup.
- Prepage all or some of the pages a process will need, before they are referenced.
  - But if prepaged pages remain unused, I/O and memory were wasted.
- Assume s pages are prepaged, of which α are used.
  - Is the cost of $s \times \alpha$ avoided page faults greater or lesser than the cost of prepaging $s \times (1 - \alpha)$ unnecessary pages?
  - If lesser, prepaging loses.

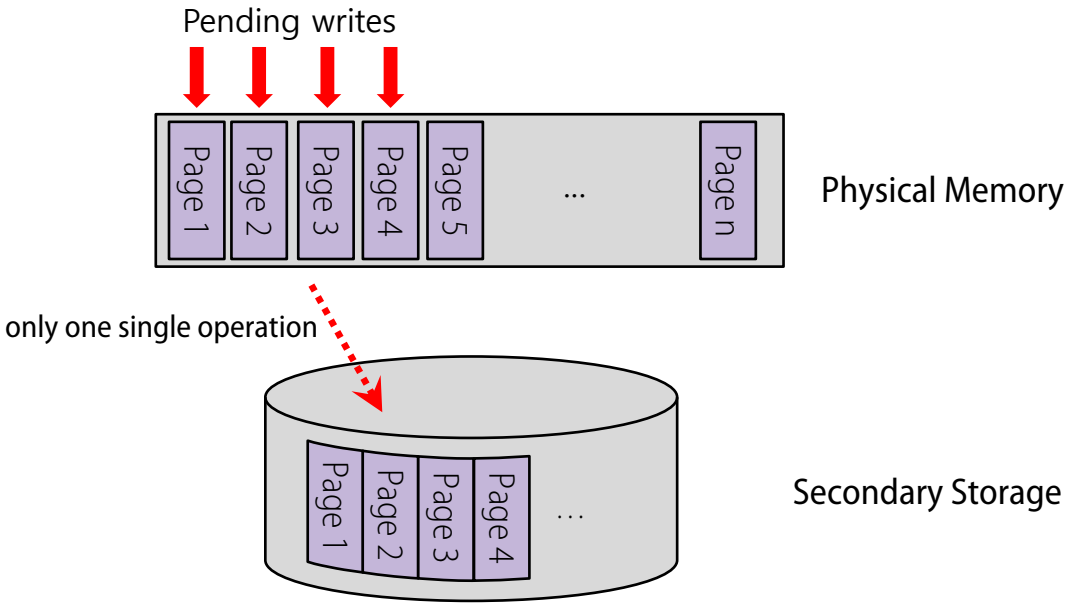
Prepaging

- The OS guesses that a page is about to be used, and thus brings it in ahead of time.
  - For example, Page 2 is likely to be accessed soon and thus should also be brought into memory.



Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
  - A single large write is more efficient than many small ones.



Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
  - A single large write is more efficient than many small ones.

Considering Dirty Pages

- The hardware includes a modified bit (aka dirty bit)
  - Page has been modified and is thus dirty, it must be written back to disk to evict it.
  - Page has not been modified, the eviction is free.

Other Issues: Page Size

- At the time of OS design little can be done about page size. *Why?*
- During the design of a new machine, page size selection must take into consideration
  - Fragmentation
  - Table size
  - I/O overhead
  - Locality

Other Issues: TLB Reach

- We call **TLB Reach** the amount of memory accessible from the TLB
  - $TLB\ Reach\ =\ (TLB\ Size) \times (Page\ Size)$
- Ideally, the working set of each process should be accessible from the TLB.
  - Otherwise there will be a high degree of page faults.
- To overcome this issue we may
  - Increase the Page Size
    - This may lead to an increase in fragmentation as not all applications require a large page size.
  - Provide Multiple Page Sizes
    - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other Issues: Program Structure

- `int[128,128] data;`
- Each row is stored in one 512B page
- Program 1

```
for (i = 0; i <128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

- Program 2

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 i.e. 16.384 page faults

Other Issues: I/O interlock

- I/O Interlock
  - Pages must sometimes be locked into memory.
  - E.g. I/O pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.