Counting-Based Page Replacement Algorithms

Counting-Based Algorithms

- Add a reference counter (in each row of the page table)
  - The HW has to increment the counter by 1 on each memory reference.
  - **Expensive hardware**
- Least Frequently Used
  - Victim selection: find the page with the **smallest** counter value
  - **Reasoning**: actively-used pages will get high count
  - **Problem**: pages with high count may become too sticky (hard to replace)
  - **Solution**: periodic decay (by OS) by right shift (div by 2) the counter value
- Most Frequently Used
  - Victim selection: find the page with the **largest** counter value
  - **Reasoning**: low count implies the page is just recently loaded, it may be needed again in the short future

More efficient swapping techniques

Overhead of Page Replacement

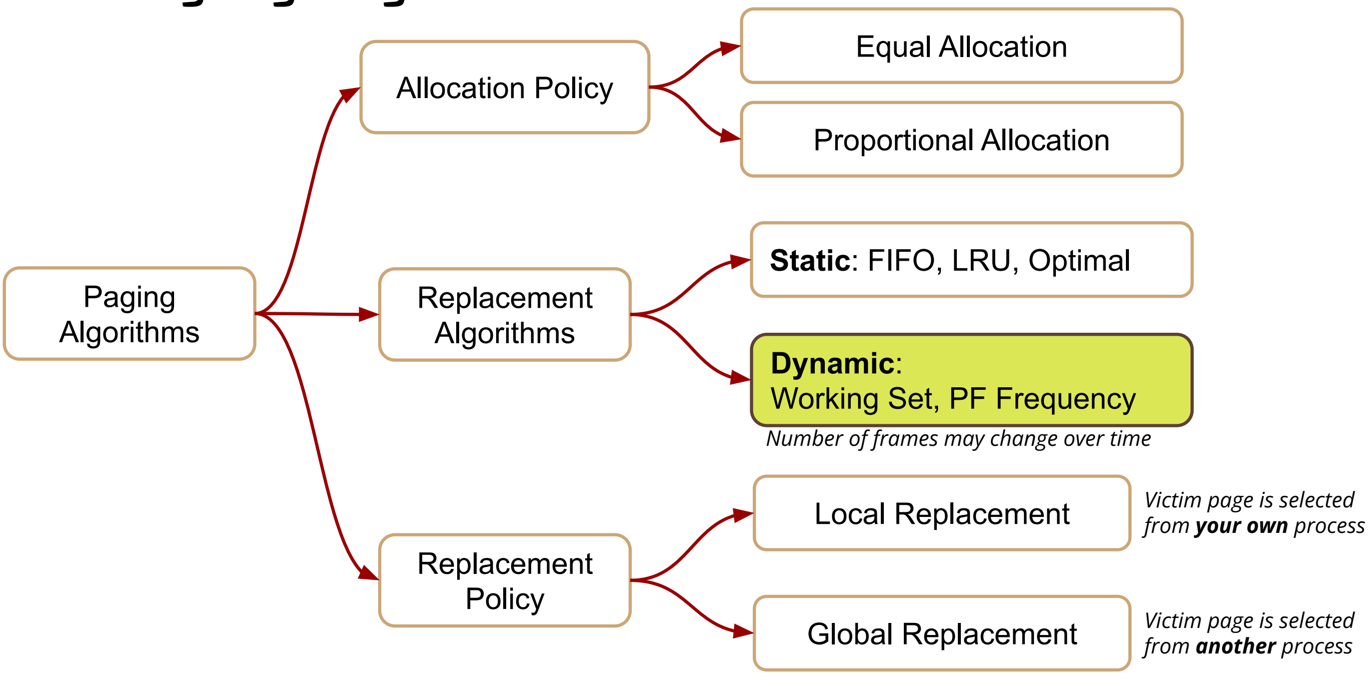
- Two I/O operations to/from the paging disk
  - Swap out: read the content of victim frame (RAM) to the paging disk
  - Swap in: load the requested frame from the paging disk to RAM
- Any techniques to avoid double I/O operations would improve the overall page response

Page Buffering Options

- **Lazy** swap out (versus immediate swap out)
  - Swap out are postponed (and perhaps performed in batch), only mark the frame as "*swapOutNeeded*"
  - The swapper maintains a **pool of these frames,**, swap in requests are handled immediately by restoring a frame from this pool
- Periodic Cleaning of "dirty" pages
  - When the paging disk/swap disk is idle write modified/dirty pages to the paging disk (thus making them "clean" again) in batch
- Tagged Pool of Free Frames
  - Tag each free frame with the **most recent page** that occupies the frame
  - When a swap in to bring in page Z is being serviced and the frame associated with Z has not been reused, then no actual I/O operation is needed to load the page

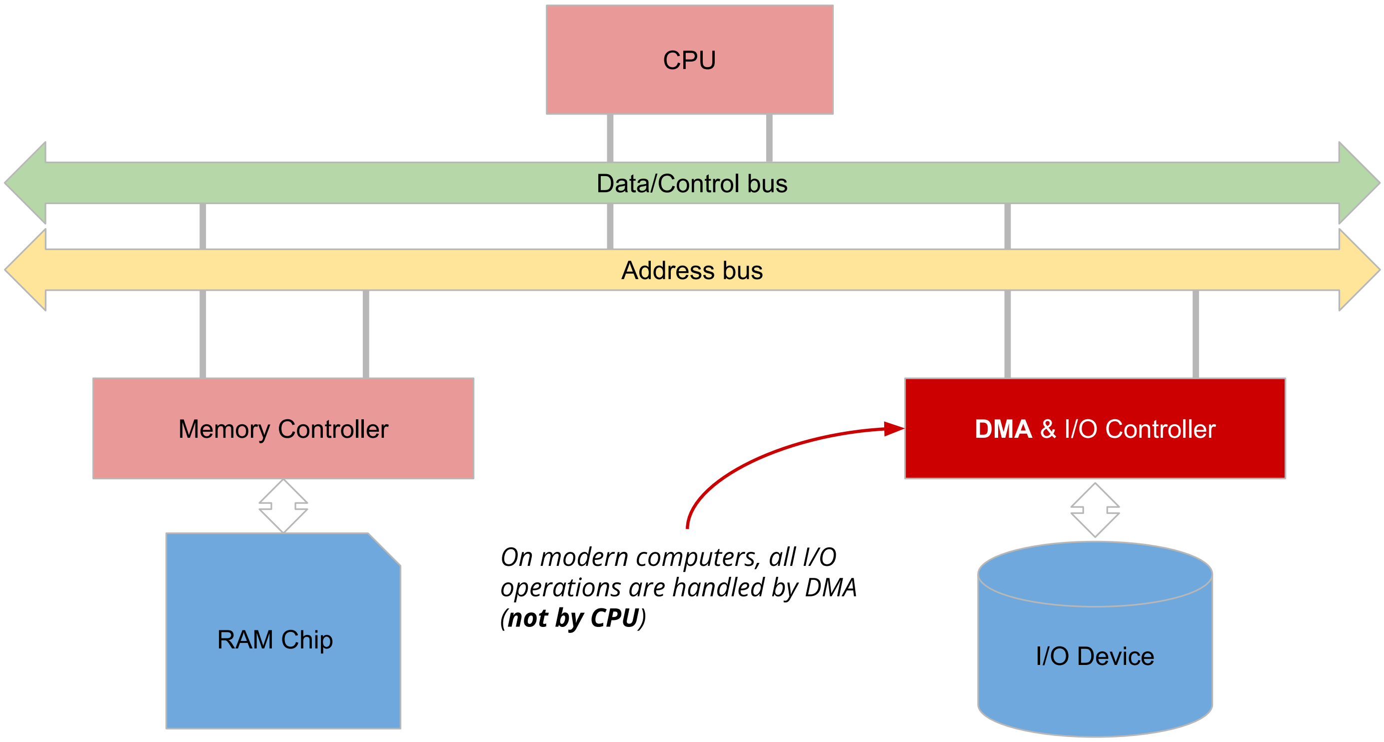
Thrashing & Dynamic Page Replacement

VM Paging Algorithms

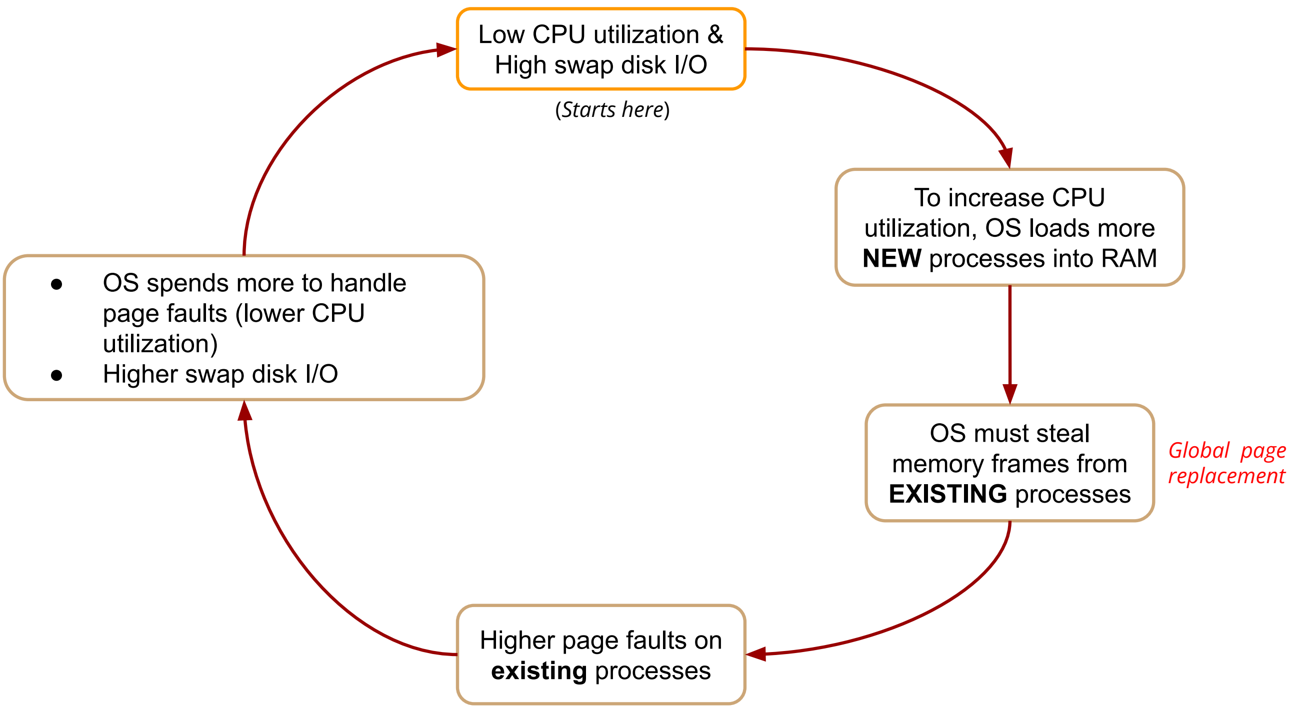


Thrashing

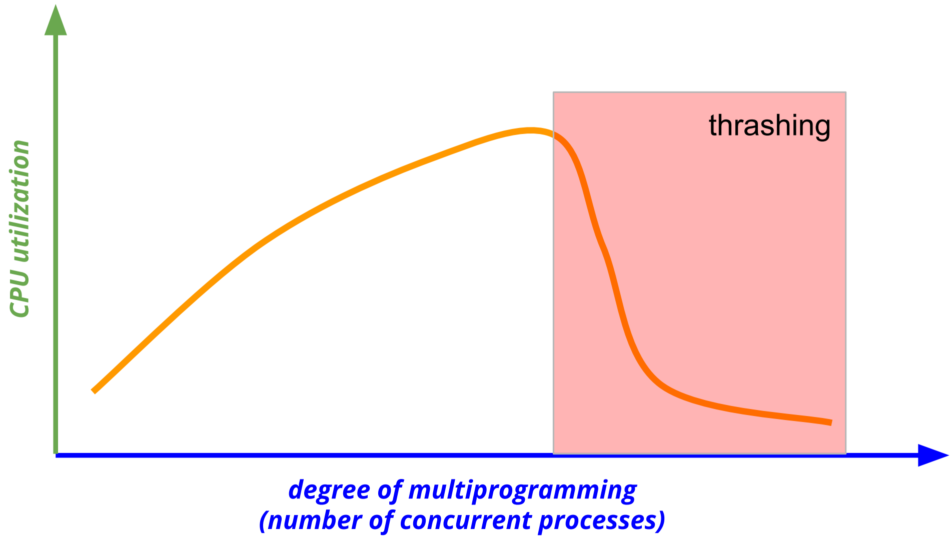
CPU, Memory, I/O Devices



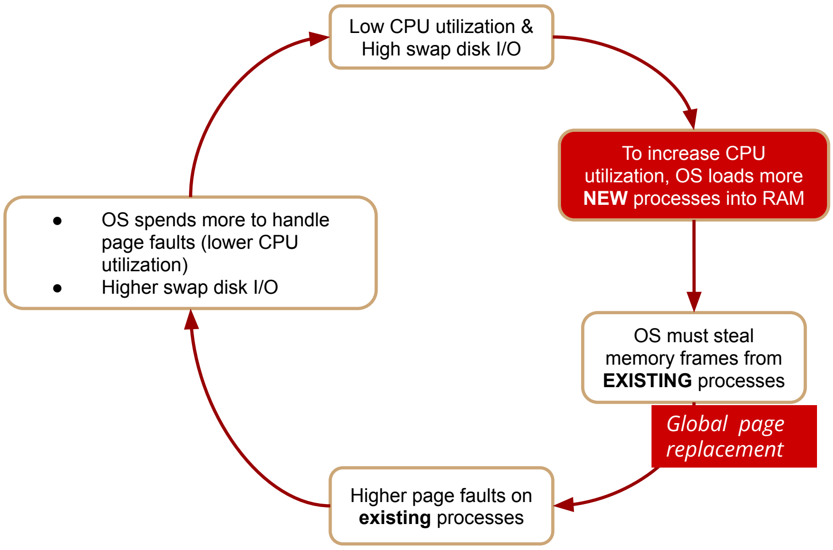
Thrashing: Circular Events



Thrashing

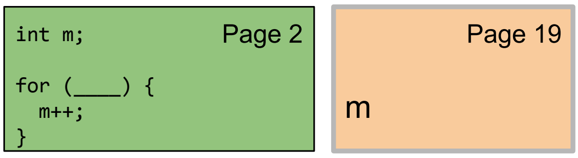


Thrashing: Problems and Solutions

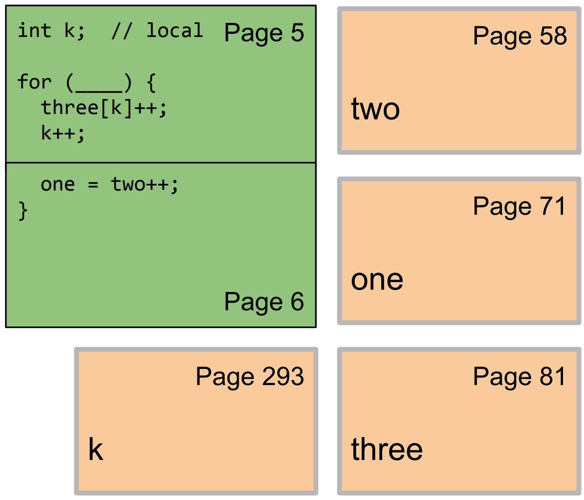


- Quick Solution
  - Don't bring in too many processes
  - Don't use GLOBAL replacement
- Better Solution: **adjust** the number of frames per process
  - *Add more*
  - *Take away some*

Dynamic Frame Allocation: Motivation

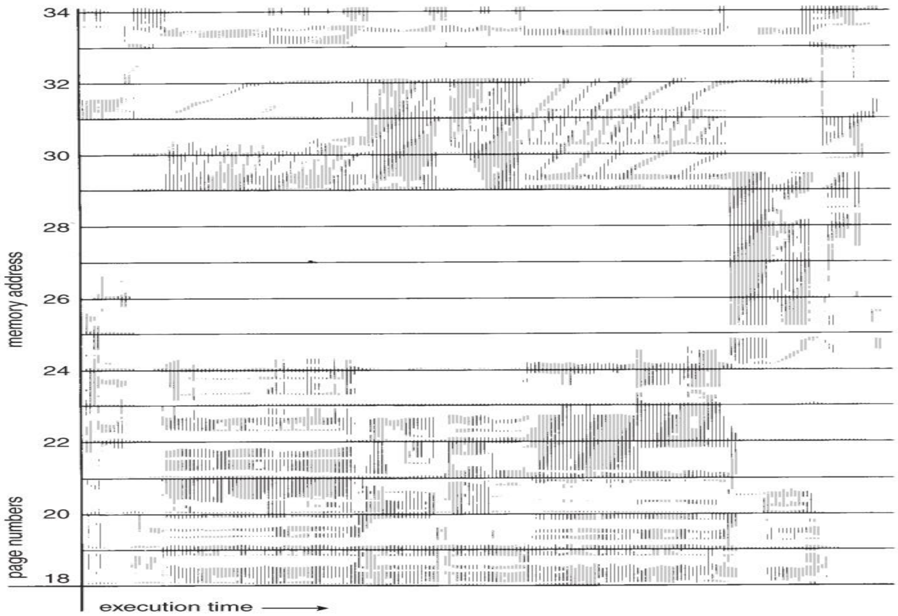


Requires ONLY two pages to run "comfortably"



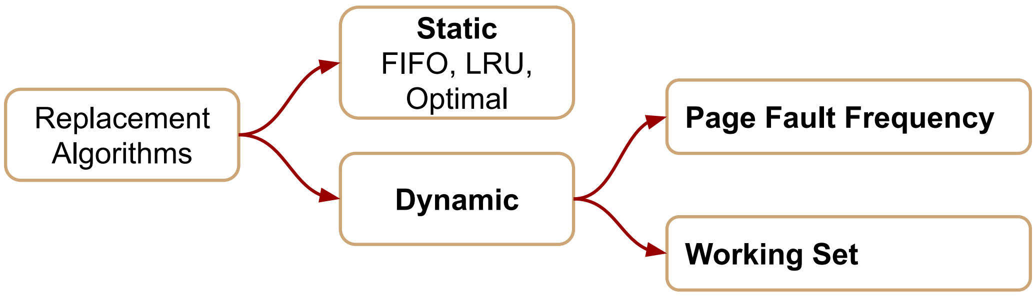
Requires SIX pages to run "comfortably"

Change of Locality: Effect on Page Faults?

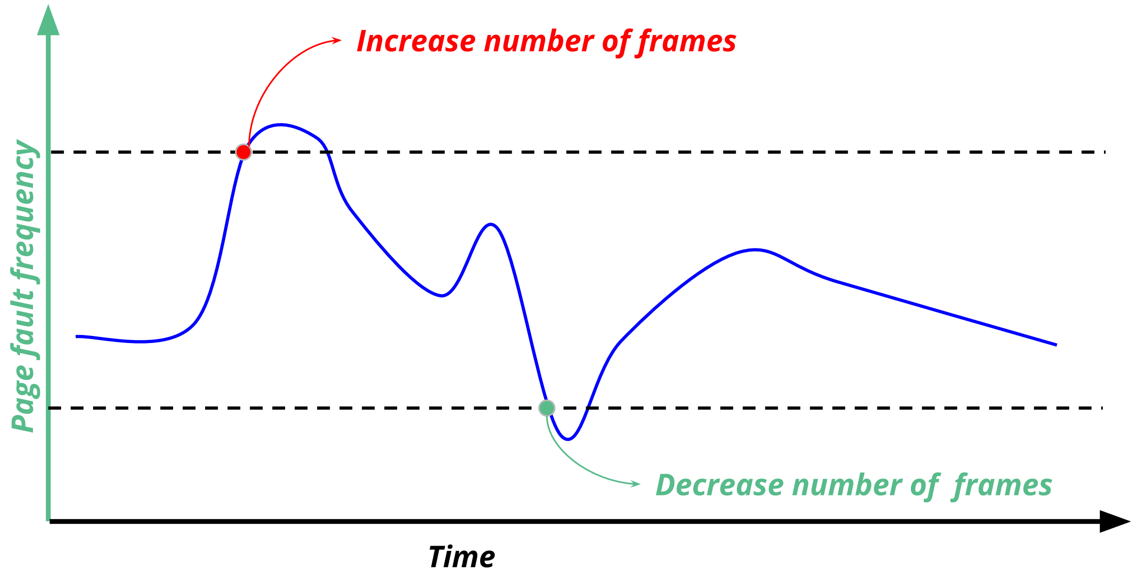


Number of PFs vs. execution time

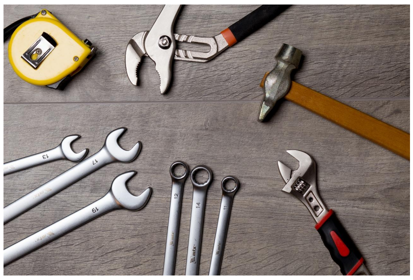
Dynamic Page Replacement Algorithms



Page Fault Frequency Algorithm

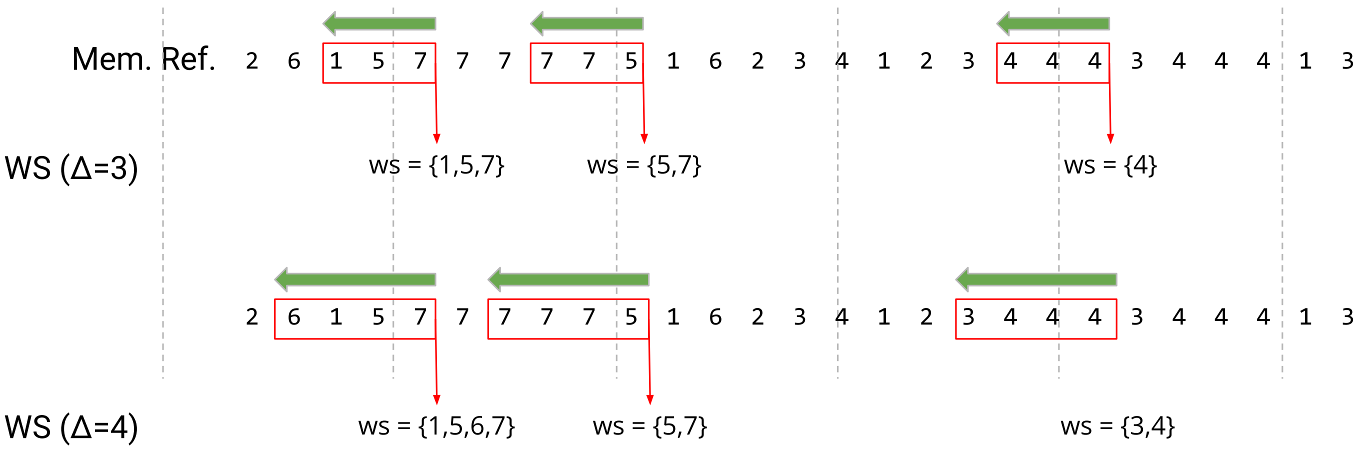


Working-Set Model



- Working-Set (of a process): is the ***current*** set of pages that must be resident in RAM for the process to work "happily"
- The WS model approaches the problem by monitoring the most recent set of pages used by a process
  - How recent? Window size Δ?
  - Too big = too costly and the window may include extra pages that may not be needed to make the process "happy"
  - Too small = incorrect estimate of Working Set

Working Set Examples



Working-Set Algorithm

- Algorithm performance depends on the window size Δ
  - Monitor the past Δ references of **each** process (working set)
- Page Replacement/Allocation Algorithm
  - **Add** new frames on page fault interrupts
  - **Remove** memory frames not in the WS
  - **Run** a process only if all of its WS pages are resident in RAM
  - **Suspend** some processes (swap out ALL their pages) if the **total** WS demands (of **ALL** runnable processes) exceeds the number of available frame

Working Set Algorithm Example (window size 3)

	7	0	1	2	0	3	0	4	2	2	2	3	2	1	2	0
	7	7	7	0	0	0	0	0	0	2	2	2	2	2	2	2
		0	0	1	1	3	3	3	2	4		3	3	3	1	1
			1	2	2	2		4	4					1		0
Working set (Δ=3)	{7}	{0,7}	{0,1,7}	{0,1,2}	{0,1,2}	{0,2,3}	{0,3}	{0,3,4}	{0,2,4}	{2,4}	{2}	{2,3}	{2,3}	{1,2,3}	{1,2}	{0,1,2}

WS Algorithm: Implementation

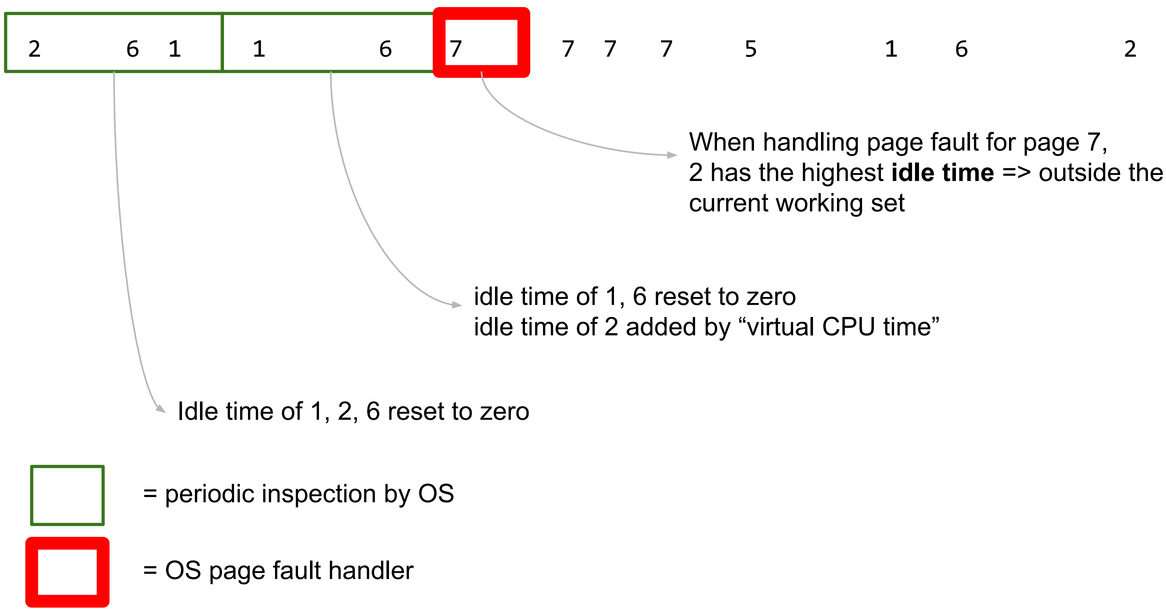
- HOW to monitor every memory reference (IMPOSSIBLE)
- Which process to **suspend** (when total demand > available RAM)
  - Smallest? Low-prio? Oldest?
  - **Ideally**: swap out a process so total demand <= available RAM

WS Approximation

- Borrow the idea from LRU approximation (k-bit history tag)
  - Use “total idle time” in place of history tag
- OS periodically inspects the ref-bit of each page (of every process)
  - If ref-bit is zero, add the amount of CPU time of the process to “idle time”
  - If ref-bit is one, reset “idle-time” to zero
  - Reset the ref-bit
- During PF handling
  - A page with large idle-time is outside the window Δ
  - A page with small idle-time is within the window Δ

Working Set: Idle time and Virtual CPU time

Reference String



Extra Benefits of Paging & Page Mapping

Memory-Mapped Files

- Background: page-fault handler **normally** loads missing pages from a **paging/swap disk**
- A one-bit flag can be used in the PTE to inform the OS to load the "*missing page*" from the **user file systems**
  - Fact: a disk block (of a file) can be mapped to a page (or pages) in RAM
  - A page fault during "memory read" cause the disk block to be loaded (from the file system to RAM)
  - A "memory write" does not necessarily imply an **immediate** physical write to the file system
- Linux
  - `mmap (void *mem_addr, ___, ___, ___, int file_des, ____)`

Memory-Mapped I/O

- Another example of memory-mapped "files"
- Goal: Reserved certain memory addresses to be used for I/O operations
- A feature that is usually provided by the CPU **hardware**
- How it works
  - I/O controllers (hardware) use **data and command registers** (I/O ports) for exchanging data/commands with the CPU or DMA
  - CPU splits the entire address space into "I/O addr space" and "MEM addr space"
  - References to address within the "I/O addr space" are routed to the appropriate I/O device(s)
- Linux: `/proc/iomem`