# CSE2431 – Lecture Topic 5
# Virtual Memory (part 4)
# Swaping


# Virtual Memory (pt 4) Swaping

Instructor: Luan Duong, Ph.D.

CSE 2431: Introduction to Operating Systems

**Reading: Chapter 21-23 in Required Textbook**

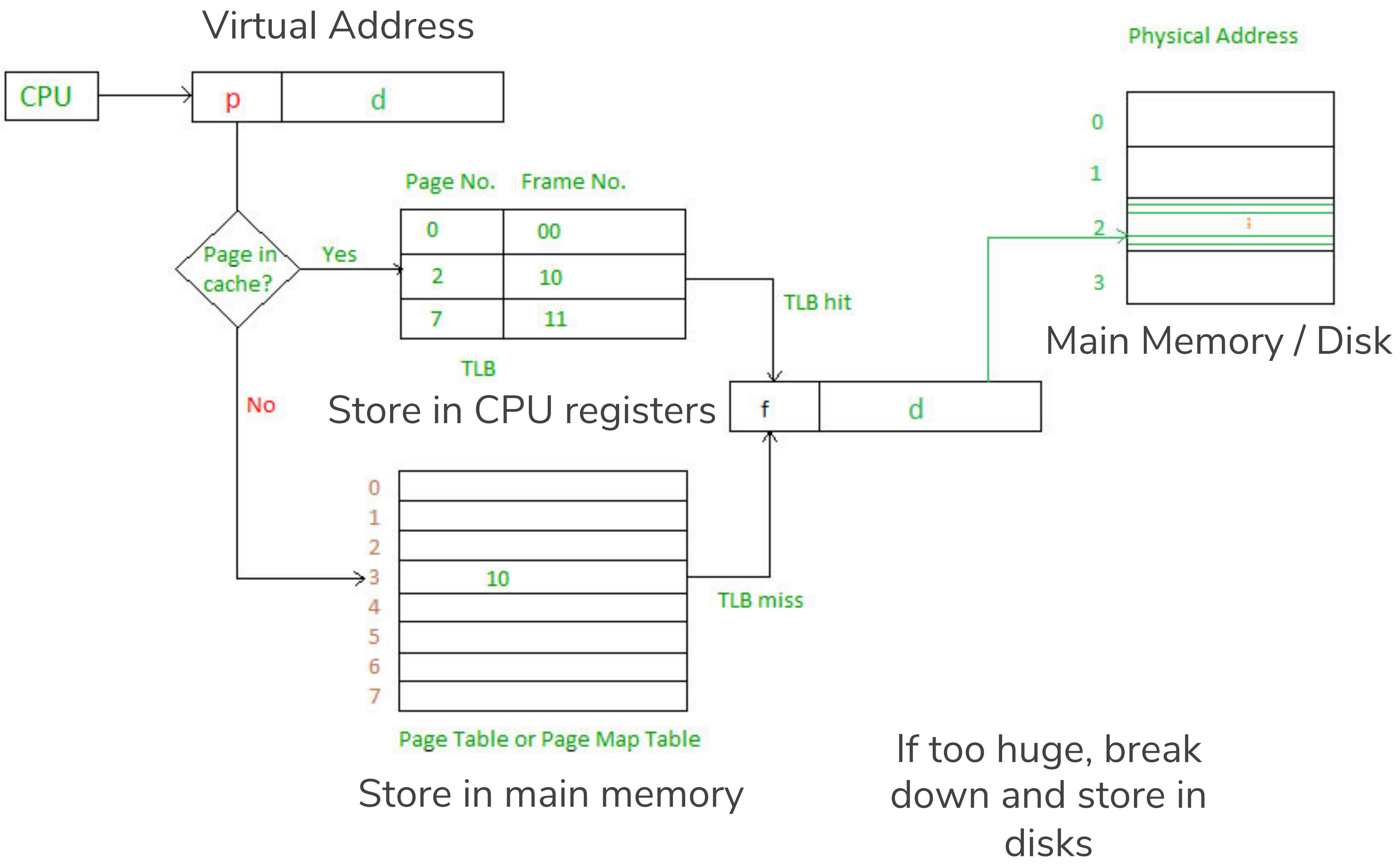
Lecture slides and materials adopted and referred from previously taught course by Dr. Yang Wang

# Previous lecture

- Paging (Part 3)
- Swaping (Part 4)

# Review on TLB and Page Table

- <u>Scenario 1:</u> VPN **is in TLB**. Compute PA and access memory
- <u>Scenario 2:</u> VPN **is not in TLB** but is **present**. Put **it in TLB** and retry (will go to Scenario 1)
- <u>Scenario 3:</u> VPN **is not in TLB** and **not present**. **Raise a page fault** and retry (will go to Scenario 2).



# Outline: Virtual Memory (Part 4)

- Memory overview (part 1)
- Virtual memory procedures (part 2)
  - Dynamic relocation
- Virtual memory procedure (part 3)
  - Paging
- Virtual memory procedure (part 4)
  - Swaping

# Motivation

- In Dynamic Relocation, we still assume all address **spaces fit in physical memory**.
  - This assumption limits the memory space of a process.
  - This assumption limits the number of processes we can run concurrently.
- Can we relax this assumption?
- Opportunities:
  - A process **may not use** all its virtual address space (i.e. many pages are invalid): we already exploit this opportunity in paging.
  - Some processes run infrequently.
  - Part of the code or data of a process may be accessed infrequently.

# Basic idea of Swapping

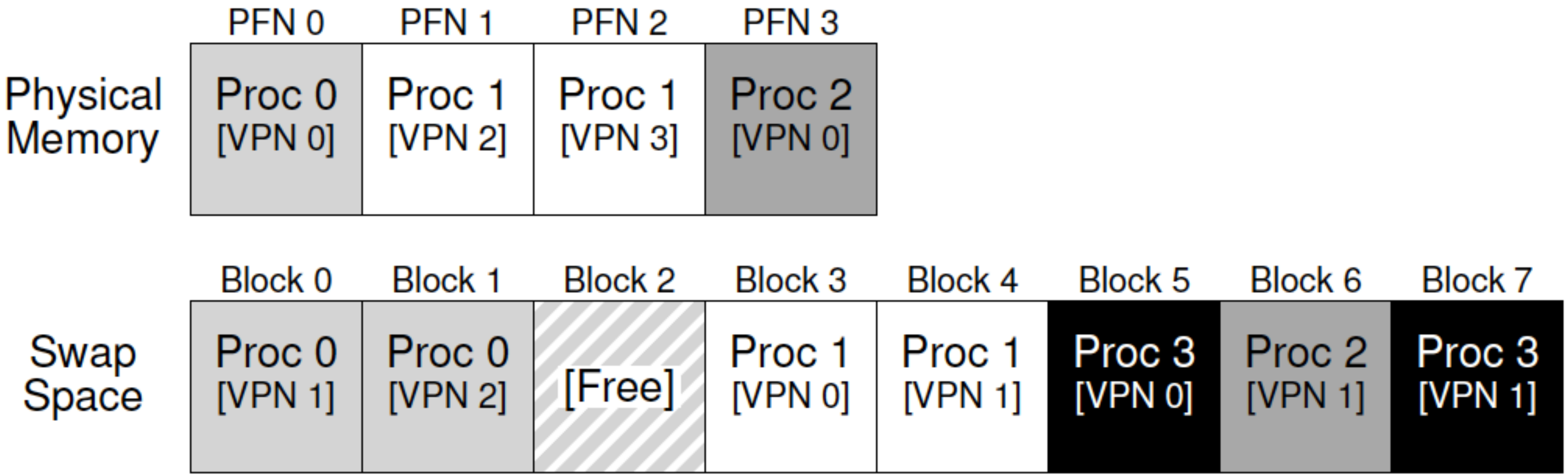
- **Swap out** **infrequently** accessed pages to disks
- **Swap in** pages from disks when they are accessed
- This is like caching
  - Cache stores the frequently accessed data of memory.
  - Memory stores the frequently accessed data of disk.
  - They are transparent to the users.

# Swap space

- An OS needs to **reserve** some space on disks for **swapping**
  - An administrator can set the size of the swap space.

- For code, the file that contains the program can also serve as **swap space**

# Example of Swap Space



# Swap space implementation

- The OS needs to keep track of whether a page is in memory or not.

- An OS keeps a Present Bit for each page in the page table:
  - If Present Bit = true, page is in memory
  - If Present Bit = false, page is on disk

- Compare Valid Bit and Present Bit
  - Valid Bit: whether the page has been allocated
  - Present Bit: whether the allocated page is in memory or on disk

# Do you remember Address translation?

Given a virtual address (VA), calculate page number (VPN)

1. Search VPN in TLB
2. If not found, search VPN in page table
3. Check whether the entry is valid
4. If valid, find the frame number (PFN) in the entry and calculate physical address (PA)

# Incorporating Swapping

- Given a virtual address (VA), calculate page number (VPN)
- Search VPN in TLB
- If not found, search VPN in page table
- Check whether the entry is valid
- If valid, check whether the page is present.
- If present, find the frame number in the entry and calculate physical address
- If not present, raise a page fault.

# Handling Page Fault

- First, it is **NOT** a **real “fault”**. It just means page is not in memory.

- Page Fault is usually handled by OS
  - OS needs to register a page fault handler to CPU

- The page fault handler reads the page from disk into memory
  - OS needs to store disk location of a page in the page table
  - Page fault handler sets Present Bit to true after completion
  - CPU then retries the instruction
  - All are transparent to the process

# What to do if memory is full?

- It may happen that the OS has no free frame to swap in a page
- In this case, the OS needs to first swap out a page to swap space
- Which page to swap out?
  - We will learn later.

# Full Workflow of a memory access

- Scenario 1: VPN **is in TLB**. Compute PA and access memory
- Scenario 2: VPN **is not in TLB** but is **present**. Put **it in TLB** and retry (will go to Scenario 1)
- Scenario 3: VPN **is not in TLB** and **not present**. **Raise a page fault** and retry (will go to Scenario 2).

# Real Replacement Strategy

- It's bad to swap out a page when there is actually no free space
  - Because a memory access has to wait for two disk I/Os
- OS swaps out pages in the background
  - OS swaps out pages periodically (every few seconds).
  - OS also swaps out pages if the number of free frames is under a threshold.
  - Swapping a number of pages together allows optimizations.

# Swapping: Terminology

- **Reference string:** the memory reference sequence generated by a program.
- **Paging:** moving pages from (to) disk
- **Optimal:** the best (theoretical) strategy
- **Eviction:** throwing something out
- **Pollution:** bringing in useless pages or cache lines
- **Dirty/Clean page**: If a page is dirty, its content has not been written to the disk.

# Swapping: Policies

- Let's get back to the question: which page to swap out when memory is full?

- As usual, we first need a **criteria** to decide which policy is better
  - Physical memory can be viewed as a cache for all memory pages
  - When a memory accesses a page in physical memory, it is a cache hit
  - When a page is not in physical memory, it is a cache miss
  - Average memory access time = hit ratio * memory latency + miss ratio * disk latency (hit ratio + miss ratio = 1)
  - A good policy should have a high hit ratio.

# Swapping: Policies

- **Principle of Optimality:** Replace the page that won't be used again the farthest time in the future.
- **FIFO (First In, First Out):** Replace the page that's been in main memory the longest
- **LRU (Least Recently Used):** Replace the page that hasn't been used for the longest time
  - **LFU (Least Frequently Used):** Replace the page that's used the least often (approximates LRU)
  - **NRU (Not Recently Used):** Replace the page that isn't used recently (approximates LRU)
- **Random page replacement:** Choose a page randomly
- **Working Set:** Keep in memory pages that the process is actively using.

# Reality: Latencies

- **Memory latency** is hundreds of nanoseconds ($10^{-9}$ seconds)
- **Disk latency** is tens of milliseconds ($10^{-3}$ seconds)
- So disk is **super slow** compared to memory
- We'd better make hit ratio high.

# The Optimal Replacement Policy (MIN)

- Is there a policy that can always minimize number of misses?

- Yes. Swap out the page that will be accessed furthest in the future.
  - It can be proved that this policy always results in fewest-possible misses.

- But ……
  - This policy requires an OS to have the ability to foresee the future.
  - This is impossible in general.

- Although not practical, MIN usually serves as a base line of comparison

# Example of MIN

- Physical memory can hold 3 pages.

Cold start misses: accesses 0, 1, 2

Capacity misses: accesses 3, 2

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 2 | 0, 1, 3 |
| 0 | Hit | | 0, 1, 3 |
| 3 | Hit | | 0, 1, 3 |
| 1 | Hit | | 0, 1, 3 |
| 2 | Miss | 3 | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |

Hit ratio = 6/11

# Another Example of MIN (Optimality)

- 12 references, 7 faults

| Page Refs | 3 Page Frames | | | |
|---|---|---|---|---|
| | Fault? | Page Contents | | |
| – | – | – | – | – |
| A | Yes | A | – | – |
| B | Yes | A | B | – |
| C | Yes | A | B | C |
| D | Yes | A | B | D |
| A | No | A | B | D |
| B | No | A | B | D |
| E | Yes | A | B | E |
| A | No | A | B | E |
| B | No | A | B | E |
| C | Yes | C | B | E |
| D | Yes | C | D | E |
| E | No | C | D | E |

time

# First-in, First-out (FIFO)

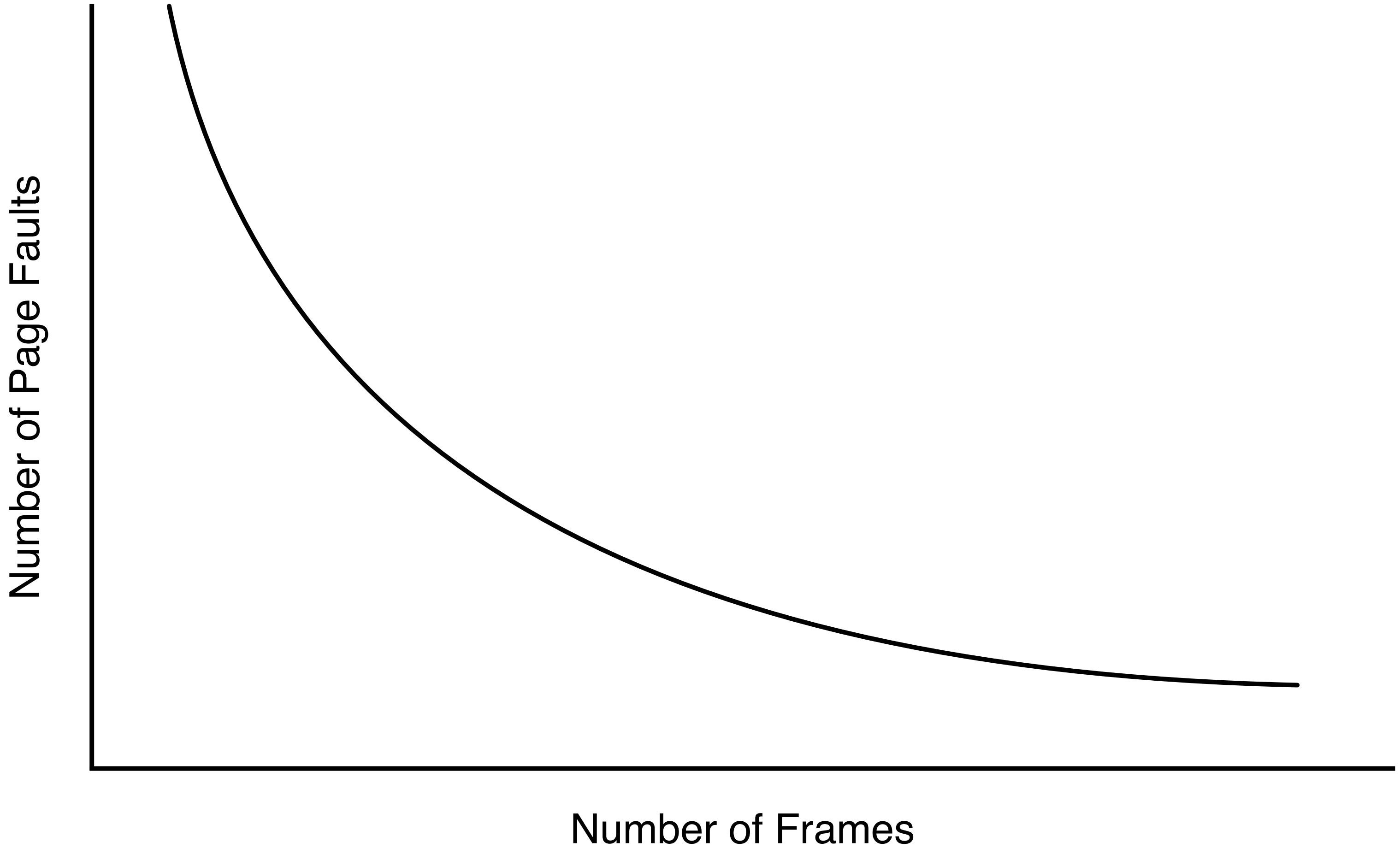
- Swap the page that was first swapped in
- 12 references, 9 faults

| Page Refs | 3 Page Frames | | | |
|---|---|---|---|---|
| | Fault? | Page Contents | | |
| – | – | – | – | – |
| A | Yes | A | – | – |
| B | Yes | A | B | – |
| C | Yes | A | B | C |
| D | Yes | D | B | C |
| A | Yes | D | A | C |
| B | Yes | D | A | B |
| E | Yes | E | A | B |
| A | No | E | A | B |
| B | No | E | A | B |
| C | Yes | E | C | B |
| D | Yes | E | C | D |
| E | No | E | C | D |

time ↓

# First-in, First-out (FIFO)

- Expecting paging behaviour with increasing number of page frames



# First-in, First-out: Belady's Anomaly (FIFO)

- FIFO, 4 physical pages
- 12 references, 10 faults
- As the number of page frames increase, so does the fault rate

| Page Refs | 4 Page Frames | | | | |
|---|---|---|---|---|---|
| | Fault? | Page Contents | | | |
| – | – | – | – | – | – |
| A | Yes | A | – | – | – |
| B | Yes | A | B | – | – |
| C | Yes | A | B | C | – |
| D | Yes | A | B | C | D |
| A | No | A | B | C | D |
| B | No | A | B | C | D |
| E | Yes | E | B | C | D |
| A | Yes | E | A | C | D |
| B | Yes | E | A | B | D |
| C | Yes | E | A | B | C |
| D | Yes | D | A | B | C |
| E | Yes | D | E | B | C |

time

# First-in, First-out (FIFO): Belady's Anomaly

- Belady's anomaly: increasing cache size may decrease hit ratio
  - E.g. 0,1,2,3,0,1,4,0,1,2,3,4 (cache size 3 and 4)

| Access | Hit/Miss? | Evict | Resulting Cache State | |
|---|---|---|---|---|
| 0 | Miss | | First-in→ | 0 |
| 1 | Miss | | First-in→ | 0, 1 |
| 2 | Miss | | First-in→ | 0, 1, 2 |
| 0 | Hit | | First-in→ | 0, 1, 2 |
| 1 | Hit | | First-in→ | 0, 1, 2 |
| 3 | Miss | 0 | First-in→ | 1, 2, 3 |
| 0 | Miss | 1 | First-in→ | 2, 3, 0 |
| 3 | Hit | | First-in→ | 2, 3, 0 |
| 1 | Miss | 2 | First-in→ | 3, 0, 1 |
| 2 | Miss | 3 | First-in→ | 0, 1, 2 |
| 1 | Hit | | First-in→ | 0, 1, 2 |

Hit ratio = 4/11

# LFU and LRU

- LFU: pages that are frequently accessed may be accessed again
  - Swap out the page that is least frequently used

- LRU: pages that are recently accessed may be accessed again
  - Swap out the page that is least recently used

# Example of LRU

Hit ratio = 6/11

| Access | Hit/Miss? | Evict | Resulting Cache State | |
|---|---|---|---|---|
| 0 | Miss | | LRU→ | 0 |
| 1 | Miss | | LRU→ | 0, 1 |
| 2 | Miss | | LRU→ | 0, 1, 2 |
| 0 | Hit | | LRU→ | 1, 2, 0 |
| 1 | Hit | | LRU→ | 2, 0, 1 |
| 3 | Miss | 2 | LRU→ | 0, 1, 3 |
| 0 | Hit | | LRU→ | 1, 3, 0 |
| 3 | Hit | | LRU→ | 1, 0, 3 |
| 1 | Hit | | LRU→ | 0, 3, 1 |
| 2 | Miss | 0 | LRU→ | 3, 1, 2 |
| 1 | Hit | | LRU→ | 3, 2, 1 |

# LRU = No Anomalies!

- LRU: 4 physical pages
- 12 references, 8 faults
- Anomalies cannot occur. Why?

| Page Refs | 4 Page Frames | | | | |
|---|---|---|---|---|---|
| | Fault? | Page Contents | | | |
| – | – | – | – | – | – |
| A | Yes | A | – | – | – |
| B | Yes | A | B | – | – |
| C | Yes | A | B | C | – |
| D | Yes | A | B | C | D |
| A | No | A | B | C | D |
| B | No | A | B | C | D |
| E | Yes | A | B | E | D |
| A | No | A | B | E | D |
| B | No | A | B | E | D |
| C | Yes | A | B | E | C |
| D | Yes | A | B | D | C |
| E | Yes | E | B | D | C |

# LRU Issues:

- How to track "**recently**"?
  - **Use time**
    - Record time of reference with page **table entry**
    - Use counter **as clock**
    - Search for **smallest** time
  - **Use stack**
    - Remove reference of page from stack (**linked list**)
    - Push it on top of stack
- Both approaches require large processing overhead, more space, and hardware support

# Random Swap

- Swap out a **random** page

- Randomness has its advantage
  - There does not exist a case that is particularly bad for Random.

- Random swap is like a “no-bad no-good guy”. It depends.

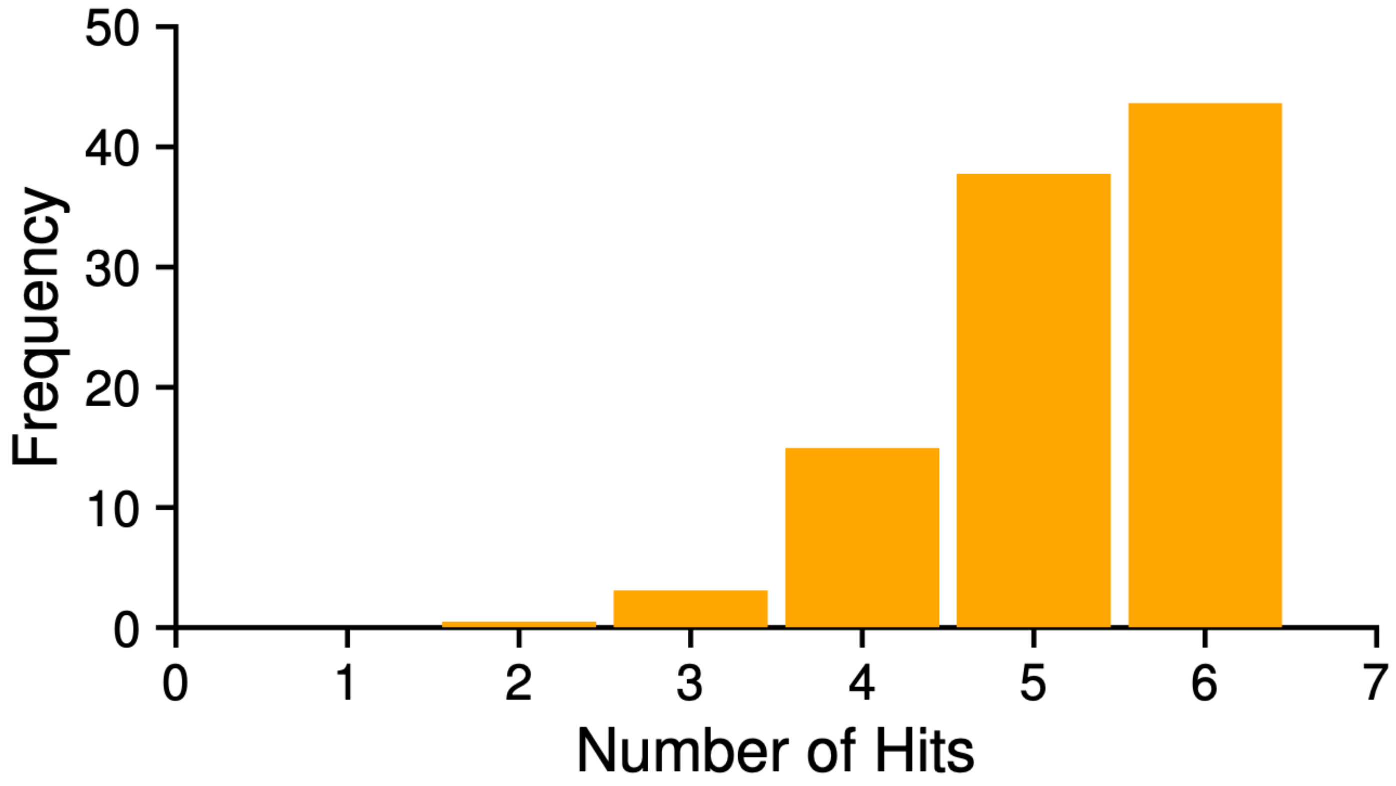
# Random Swap: Example

- Swap out a **random** page
- Hit ratio: 5/11
- Note: This is just one case of Random Swap. How Random does depend on the luck of the draw.

| Access | Hit/Miss? | Evict | Resulting Cache State |
|---|---|---|---|
| 0 | Miss | | 0 |
| 1 | Miss | | 0, 1 |
| 2 | Miss | | 0, 1, 2 |
| 0 | Hit | | 0, 1, 2 |
| 1 | Hit | | 0, 1, 2 |
| 3 | Miss | 0 | 1, 2, 3 |
| 0 | Miss | 1 | 2, 3, 0 |
| 3 | Hit | | 2, 3, 0 |
| 1 | Miss | 3 | 2, 0, 1 |
| 2 | Hit | | 2, 0, 1 |
| 1 | Hit | | 2, 0, 1 |

# Random Swap: Example

- Random Performance Over 10,000 Trials



# Which policy to choose?

- MIN is proved to be the best, but is impractical
- Random may be good, but may also be bad (no guarantee)
- LRU is **good** in general (read Section 22.6 yourself)
  - FIFO: does not consider locality
  - LFU: if a page is hot for a while and then gets cold, it's hard to get rid of it.
  - This is not a theorem. You can find cases where FIFO or LFU is better than LRU.
- Many other algorithms have been developed.
  - E.g. ARC from IBM, MQ from UIUC, .....
  - A paper published in 2023 argues optimized FIFO can outperform LRU.

# Implementing LRU

- Solution 1:
  - Keep a **timestamp** for each page
  - **Update** the timestamp when **the page is accessed**
  - **Swap out** the page with the **smallest timestamp**

- This approach is **direct**, but **expensive**
  - **Updating and swap out operations are both O(logN) (using a heap)**

# Implementing LRU

- Solution 2: **queue**
  - Maintain **a queue for all pages**
  - When a page is accessed, **move it to the tail of the queue**
  - **Swap out** the page **at the head** of the queue
- This approach is fine. It is indeed used in many systems.
  - Both **updating** and **swapping** out operations are **O(1)**
  - Actual implementation is a little bit tricky but can be done. Think about it yourself first.
  - **Space overhead** is non-negligible: need a double-linked list to implement the queue. Need to add two pointers per page.

# Implementing LRU

- Solution 3: **approximating LRU**
  - Many variations:

- Simplest one:
  - Maintain a bit for each page
  - Set bit to true when the page is accessed
  - When swapping out, search from the current page
    - If bit is true, set it to false
    - If bit is false, swap it out

# NRU: Not Recently Used (LRU approx.)

- **<u>Page Classes (*R*, *M*):</u>**
  - (0, 0): Neither referenced nor modified
  - (0, 1): Not referenced (recently) but modified
  - (1, 0): Referenced but unmodified
  - (1, 1): Referenced and modified
- **<u>Algorithm</u>**
  - Select a page from lowest class
  - If conflict, use random or FIFO.
  - More details:
    - Hardware sets up *R* and *M* bits for each memory access
    - OS periodically clears *R* bit

# NRU: Not Recently Used (LRU approx.)

- NFU (Not Frequently Used): Evict a page not frequently used
- LRU: evict a page that is least recently used.
- NFU implementation: **simpler** than LRU
  - A software counter is kept per page
  - The **R bit (`0/1`)** is added into the counter periodically
    - Directly added (never forget history)
    - Right-shift the counter one bit and add the R bit to the leftmost (aging)
  - `00110011` would be accessed more frequently than `00010111`
  - The page whose counter has the lowest number is the least frequently used.
  - The value may not be unique; use FIFO to resolve conflicts

# Second Chance (Clock)

- Only **one reference** bit in the page table entry
  - `0` initially
  - `1` when a page is referenced
- Pages are kept in FIFO order
- Choose "victim" to evict
  - Select the head of linked list. If page has reference bit set, reset it, put it back to tail of list, and process the next page.
  - Keep processing until reach page with zero reference bit and page that one out.
- Clock algorithm is a variant of second chance (with *circular list*)
- Read more about NRU here: https://www.geeksforgeeks.org/not-recently-used-nru-page-replacement-algorithm/

# Dirty pages

- A page is **dirty** if its content has not been written to disk
  - When swapping out a dirty page, an **OS needs to write it to disk first**
  - Recall that OS periodically writes dirty pages to disks

- It is better to **swap out a clean page** comparing to swapping out a dirty page

# When to swap in a page

- Demand paging: swap in a page when it is accessed
- Prefetching: swap in a page and its **following pages**
  - A following page has a high likelihood to be accessed (space locality).
  - Reading several consecutive pages together is more efficient than reading them separately (we will learn why when we discuss disks).

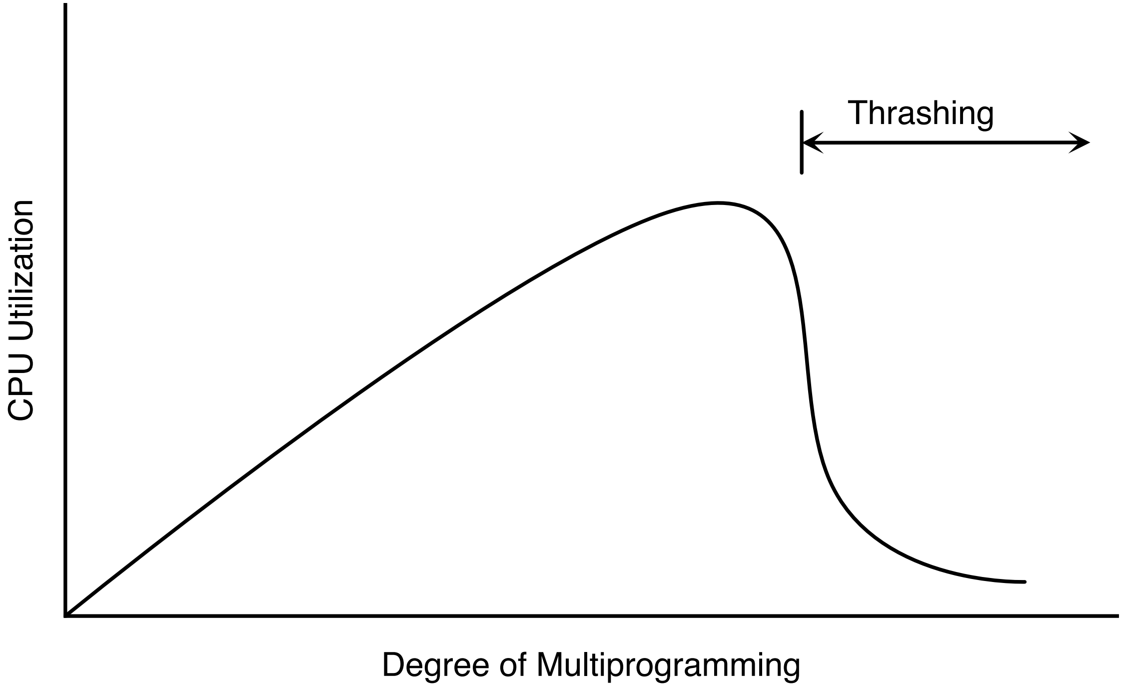
# Thrashing

- If the number of frequently used pages is larger than the number of physical frames, then frequent page faults will happen
  - Remember a disk access is $10^5$ times slower than a memory access.
  - Frequent page faults means a significant slowdown of your system.
  - Frequently used pages are sometimes called "working set".

- Swapping allows processes to have address spaces larger than physical memory
  - It is helpful if your working set is smaller than physical memory.
  - It is not designed to run applications whose working set is larger than physical memory.

# Thrashing and CPU Utilization

- As page fault rate goes up, processes get suspended on page-out queues for the disk
- System may try to optimize performance by starting new jobs
- But starting new jobs reduces the number of page frames available to each process, increasing the page fault requests
- Hence, system throughput plunges



# Frame Allocation: Minimum

- How are page frames allocated to individual processes' virtual memories in a multi-programmed environment?
- Simple case: allocate minimum number of frames per process
  - Most instructions require two operands
  - Include extra page for paging out, one for paging in
  - Moves, indirection instructions may need more pages

# Frame Allocation: Equal

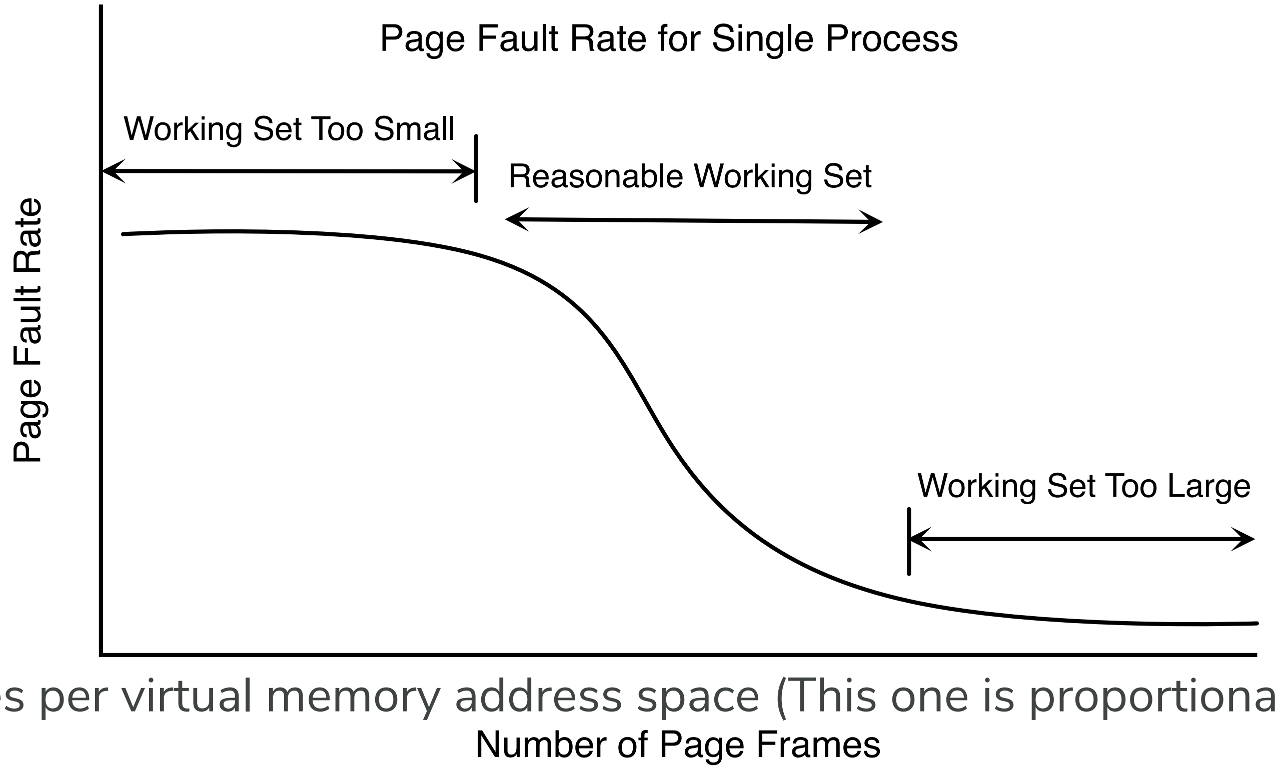
- Allocate an equal number of frames per job
  - But jobs use memory unequally
  - High-priority jobs have same number of page frames as low-priority jobs
  - Degree of multiprogramming might vary

# Proportional Allocation

- Allocate number of frames per job proportional to job size
- Challenge: how do you determine job size
  - Running command parameters?
  - Dynamic estimation?

# Page Fault Rate Curve

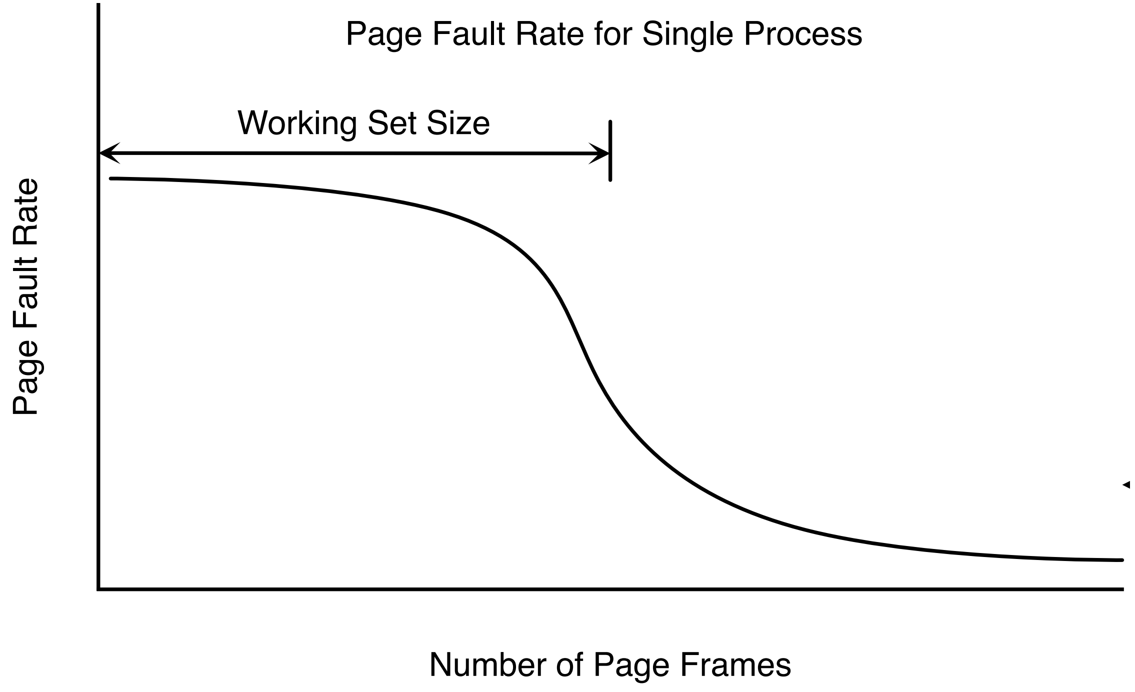
As the number of page frames per virtual memory address space **increases**, the page fault rate **decreases**




# page frames per virtual memory address space (This one is proportional to the working set)

# Working Set

- Working set model assumes locality
- ***Principle of locality:* programs cluster access to data and text *temporally***
- As the number of page frames exceeds a threshold, page fault rate drops dramatically



# Working Set in Action

- Algorithm
  - If number of free page frames **exceeds** working set of a suspended process, then **activate** the process and map in all its working set
  - If **working set size** of some process **increases** and there are no page frames free, suspend the process and release all its pages
- Moving window over reference string used for determination
- Keep track of working set

# Working Set Example

- Sliding window size: Δ
- 12 References, 8 Faults

| Page Refs | D = 4 Page Frames | | | | |
|---|---|---|---|---|---|
| | Fault? | Page Contents | | | |
| – | – | – | – | – | – |
| A | Yes | A | – | – | – |
| B | Yes | A | B | – | – |
| C | Yes | A | B | C | – |
| D | Yes | A | B | C | D |
| A | No | A | B | C | D |
| B | No | A | B | C | D |
| E | Yes | A | B | D | E |
| A | No | A | B | E | – |
| B | No | A | B | E | – |
| C | Yes | A | B | C | E |
| D | Yes | A | B | C | D |
| E | Yes | A | B | C | E |

time

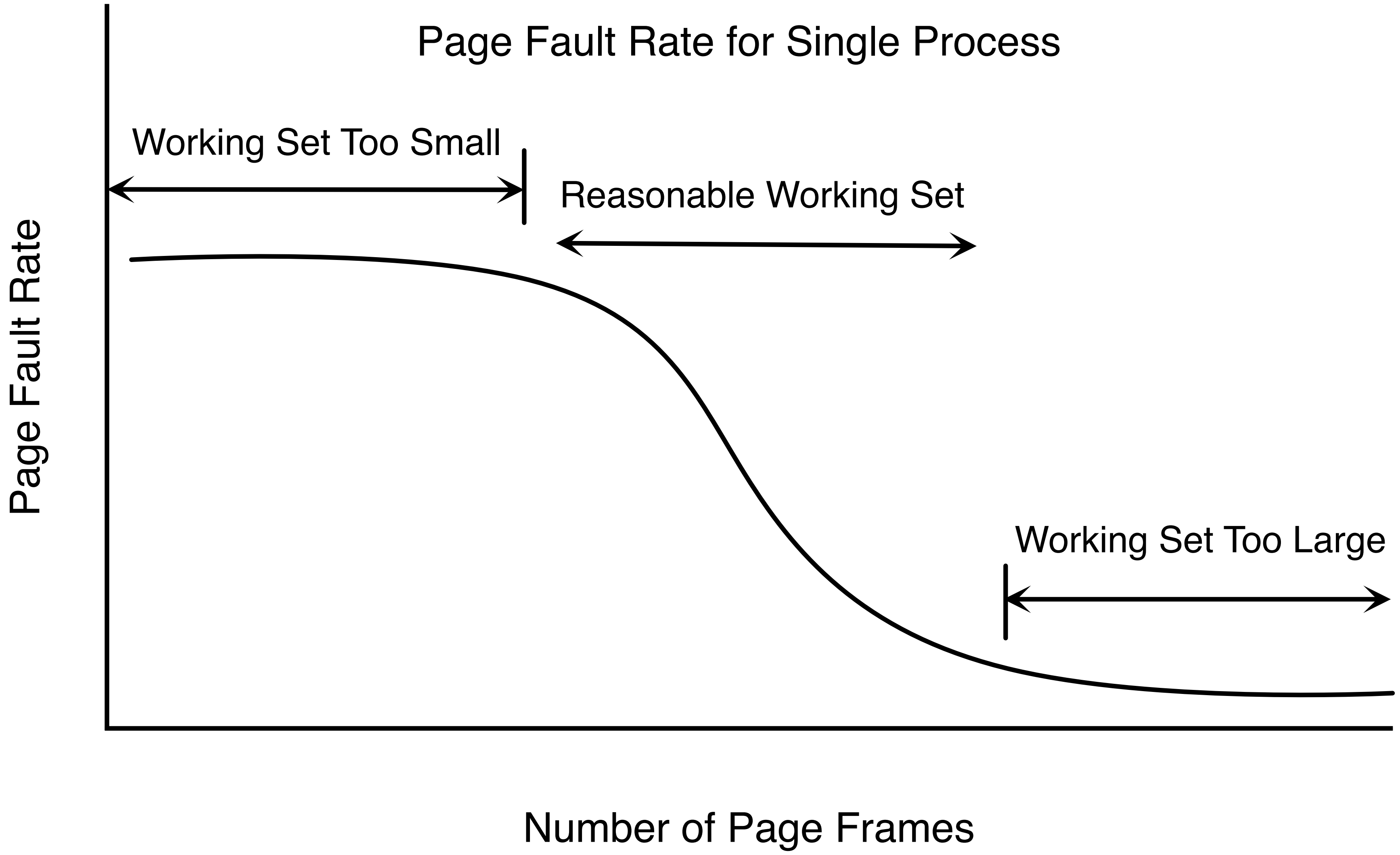
# Working Set Solution

- Approximate working set model using **timer**, **reference bit**, and **age**.
- Set timer to interrupt periodically to clear reference bit
- Once fault happens, remove pages that have not been referenced and old enough ($> \tau$) $\Longrightarrow$ outside of working set

# Page Fault Frequency Version of Working Set (1)

- Assume that if the working set is correct there will not be many page faults.
- If page fault rate increases beyond assumed knee of curve, then increase number of page frames available to process.
- If page fault rate decreases below foot of knee of curve, then decrease number of page frames available to process.

# Page Fault Frequency Version of Working Set (2)



# Page Fault Frequency Version of Working Set (1)

- Carr and Hennessy (1981); used very widely (Linux)
- Circular list as in the clock algorithm (initially empty list)
  - As pages are loaded into memory, pages are added to the list
  - Each entry contains Time of last use plus reference, dirty bits.
- Algorithm:
  - At each page fault, examine page pointed to by clock hand. Repeat the following:
    1. If reference bit is `1`, then page has been referenced during current tick, so it's in working set (poor candidate for eviction). Clear bit, update *Time of last use*
    2. If reference bit is `0`, then check if it's in working set window (*i.e.*, if *Current Time – Time of last use* < *Working set window size time*).
    3. If page is not in the working set and the page is clean, then replace it.
    4. If page is not in working set and page is dirty, request write to disk, go to next page in circular list

# Page Size Again

- **Small** pages
  - Rationale:
    - **Locality of reference tends** to be small (256)
    - Less fragmentation
  - Problem: Require large page tables
- **Large** pages
  - Rationale:
    - **Small page table**
    - I/O transfers have high seek time, so better to transfer more data per seek
  - Problem: Internal fragmentation

# A real virtual memory system

- Reference to “Case Study: VAX/VMS”

# Summary of Virtual Memory

- It provides a nice abstraction, which makes programming easy
  - A process has contiguous address space
  - The process has full control of this address space (not shared with others)
  - An address space can be even larger than physical memory space
- CPU and OS collaborate together to realize it with complex mechanisms and policies
  - Page table, TLB, page fault, ……
- Such mechanisms do have a great impact on performance
  - Your program will be more efficient if it has better data locality