Swapping

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2024

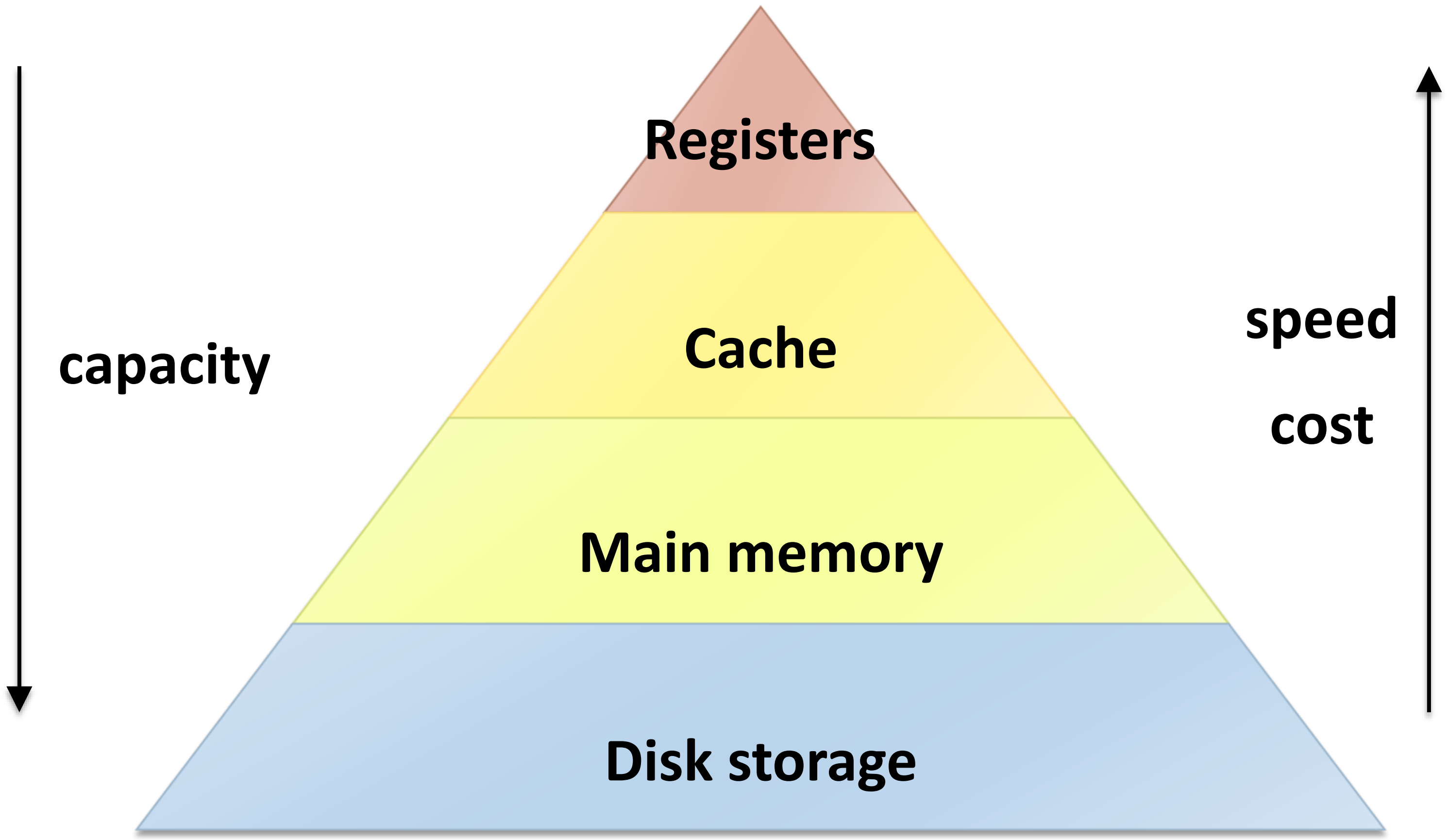
Swapping

- Support processes when not enough physical memory
  - User program should be independent of the amount of physical memory
  - Single process with very large address space
  - Multiple processes with combined address spaces

- Consider physical memory as a ________ for disks
  - Leverage locality of reference within processes
  - Process only uses small amount of address space at a moment
  - Only small amount of address space must be resident in physical memory
  - Store the rest of them to disk

Memory Hierarchy

- Each layer acts as "backing store" for layer above



How to Swap

- ____________
  - Programmers manually move pieces of code or data in and out of memory as they were needed
  - No special support needed from OS
- Process-level swapping
  - A process is swapped temporarily out of memory to a backing store
  - It's brought back into memory later for continued execution
- Page-level swapping
  - Swap pages out of memory to a backing store (*swap-out* or *page-out*)
  - Swap pages into memory from the backing store (*swap-in* or *page-in*)

Where to Swap

- Swap space
  - Disk space reserved for moving pages back and forth
  - The size of the swap space determines the maximum number of memory pages that can be in use
  - Block size is same as the page size
  - Can be a dedicated partition or a file in the file system

	PFN 0	PFN 1	PFN 2	PFN 3
Physical Memory	PID 0 (VPN 0)	PID 1 (VPN 1)	PID 1 (VPN 2)	PID 2 (VPN 0)

	Blk 0	Blk 1	Blk 2	Blk 3	Blk 4	Blk 5	Blk 6	Blk 7
Swap Space	PID 0 (VPN 1)	PID 0 (VPN 2)	Free	PID 1 (VPN 0)	PID 1 (VPN 1)	PID 3 (VPN 0)	PID 2 (VPN 1)	PID 3 (VPN 1)

When to Swap

- Proactively based on thresholds
  - OS wants to keep a small portion of memory free
  - Two threshold values: *HW* (high watermark) and *LW* (low watermark)
  - A background thread called swap daemon (or page daemon) is responsible for freeing memory (e.g., `kswapd` in Linux)

  - If (# free pages < *LW*), the swap daemon starts to evict pages from physical memory
  - If (# free pages > *HW*), the swap daemon goes to sleep

  - What if the allocation speed is faster than reclamation speed?

What to Swap

- What happens to each type of page frame on low memory?
  - Kernel code → **Not swapped**
  - Kernel data → **??**
  - Page tables for user processes → **Not swapped**
  - Kernel stack for user processes → **??**
  - User code pages → **Dropped**
  - User data pages → **??**
  - User heap/stack pages → **Swapped**
  - Files mmap'ed to user processes → **??**
  - Page cache pages → **Dropped or go to file system**
- Page replacement policy chooses the pages to evict

Page Replacement

- Which page in physical memory should be selected as a victim?
  - Write out the victim page to disk if modified (dirty bit set)
  - If the victim page is clean, just discard
    - The original version is either in the file system or in the swap space
  - Why not use direct-mapped or set-associative design similar to CPU caches?

- Goal: minimize the page fault rate (miss rate)
  - The miss penalty (cost of disk access) is so high (> x100,000)
  - A tiny miss rate quickly dominates the overall AMAT (Average Memory Access Time)

OPT (or MIN)

- Belady's optimal replacement policy (1966)
  - Replace the page that will not be used for the longest time in the future
  - Shows the lowest fault rate for any page reference stream
  - Problem: have to predict the future
  - Not practical, but good for comparison

Reference:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
	1	1	1	1	1	1	1	1	1	3	3	3
PF rate = 7 / 12		2	2	2	2	2	2	2	2	2	4	4
			3	4	4	4	5	5	5	5	5	5
	Miss	**Miss**	**Miss**	**Miss**	**Hit**	**Hit**	**Miss**	**Hit**	**Hit**	**Miss**	**Miss**	**Hit**

FIFO

- First-In First-Out
  - Replace the page that has been in memory the longest
  - Why might this be good?
    - Maybe, the one brought in the longest ago is not being used
  - Why might this be bad?
    - Maybe, it's not the case
    - Some pages may always be needed
  - Obvious and simple to implement
  - Fair: all pages receive equal residency
  - FIFO suffers from "Belady's anomaly"
    - The fault rate might increase when the algorithm is given more memory

FIFO: Belady's Anomaly

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
PF rate = 9 / 12	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
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
PF rate = 10 / 12	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
	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

LRU

- Least Recently Used
  - Replace the page that has not been used for the longest time in the past
  - Use past to predict the future
    - cf. OPT wants to look at the future
  - With locality, LRU approximates OPT
  - "Stack" algorithm: does not suffer from Belady's anomaly
  - Harder to implement: must track which pages have been accessed
  - Does not consider the frequency of page accesses
  - Does not handle all workloads well

Stack Property

- Stack algorithms
  - Policies that guarantee increasing memory size does not increase the number of page faults (e.g., OPT, LRU, etc.)
  - Any page in memory with *m* frames is also in memory with *m+1* frames

Reference:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Stack distance:	∞	∞	∞	∞	4	4	∞	3	3	5	5	5
	1	2	3	4	1	2	5	1	2	3	4	5
PF rate = 10 / 12		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2
							3	3	3	4	5	1
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss

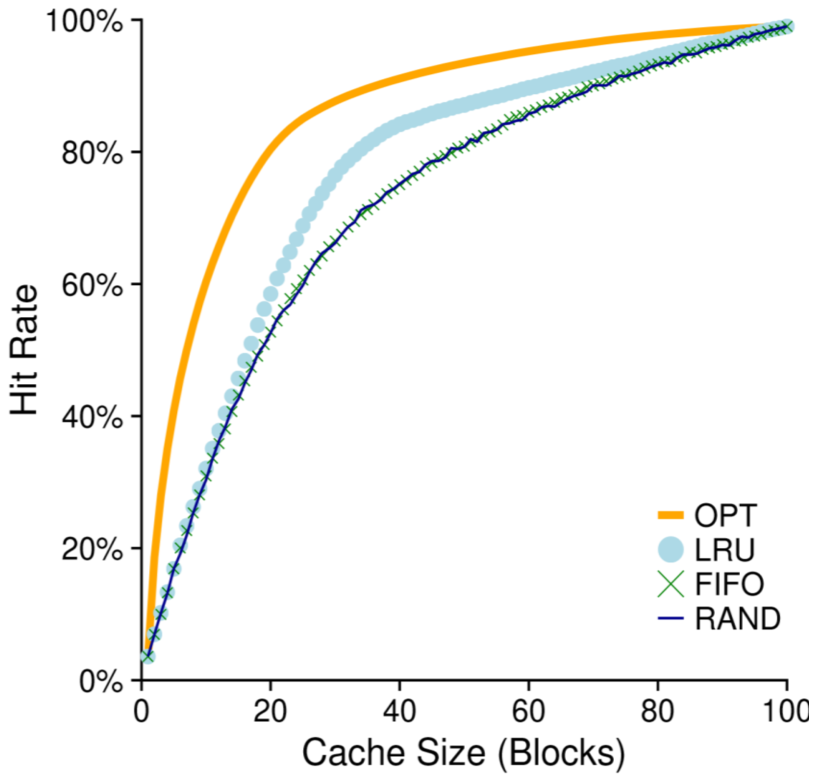
RANDOM

- Another simple policy
  - Simply picks a random page to replace under memory pressure
  - Simple to implement: no bookkeeping needed
  - Performance depends on the luck of the draw
  - Outperforms FIFO and LRU for certain workloads

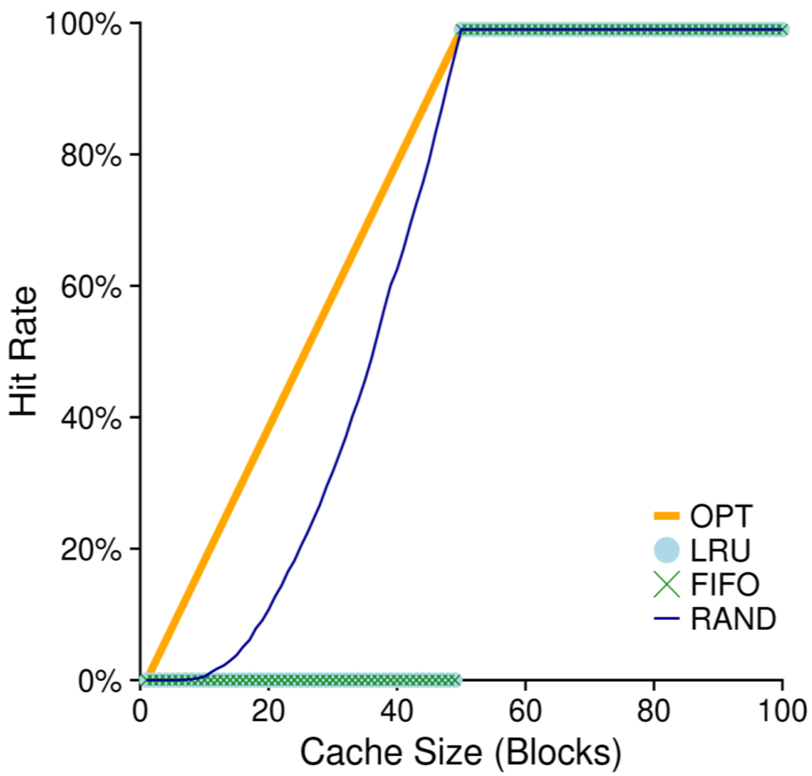
Reference:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
	1	1	1	1	1	1	1	1	1	1	1	5
PF rate = 8 / 12		2	2	2	2	2	2	2	2	3	3	3
			3	4	4	4	5	5	5	5	4	4
	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Hit	Hit	Miss	Miss	Miss

Comparisons

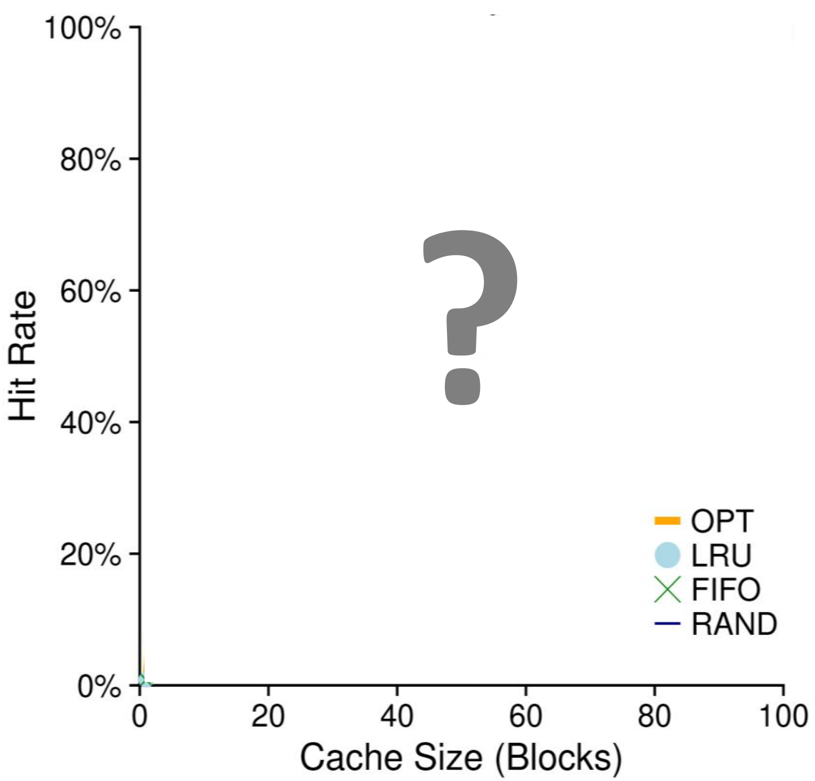
The 80-20 Workload



The Looping Workload (50 blocks)



The Random Workload

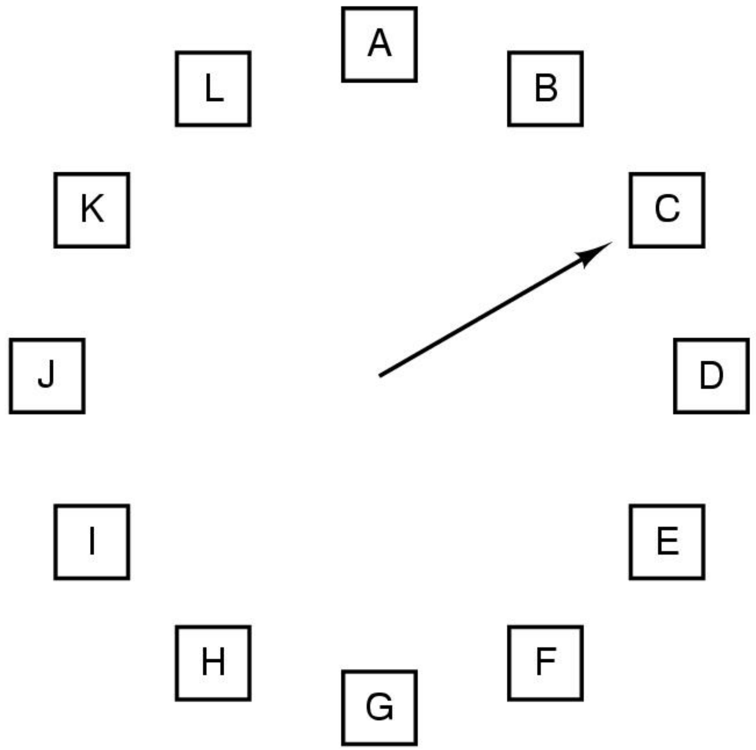


Implementing LRU

- Software approach
  - OS maintains ordered list of page frames by reference time
  - When page is referenced: move page to the front of the list
  - When need victim: pick the page in the back of the list
  - Slow on memory reference, fast on replacement
- Hardware approach
  - Associate timestamp register with each page frame
  - When page is referenced: store system clock in register
  - When need victim: scan through registers to find oldest clock
  - Fast on memory reference, slow on replacement (especially as the size of memory grows)

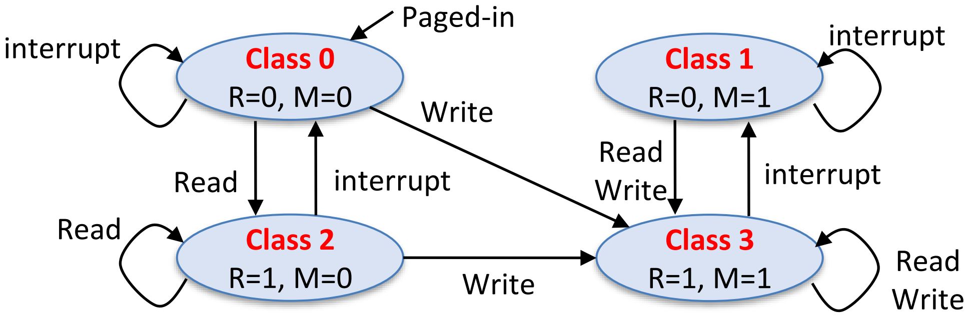
CLOCK

- An LRU approximation algorithm
  - Uses R (Reference) bit in each PTE
  - Arranges all of physical page frames in a big circle
  - A clock hand is used to select a victim
    - When a page fault occurs, the page the hand is pointing to is inspected
    - If (R == 1), turn it off and go to next page (second chance)
    - If (R == 0), evict the page
    - Arm moves quickly when pages are needed
  - If memory is large, "accuracy" of information degrades



Clock Extensions

- Clustering: Replace multiple pages at once
  - Expensive to run replacement algorithm
  - A single large write is more efficient than many small ones
- Use M (modify) bit to give preference to dirty pages
  - More expensive to replace dirty pages
  - Replace pages that have R bit and M bit cleared
- Add software counter for each page frame
  - Better ability to differentiate across pages
  - Increment software counter if R bit is 0
  - Smaller counter value means the page accessed more recently
  - Replace pages when counter exceeds some specified limit



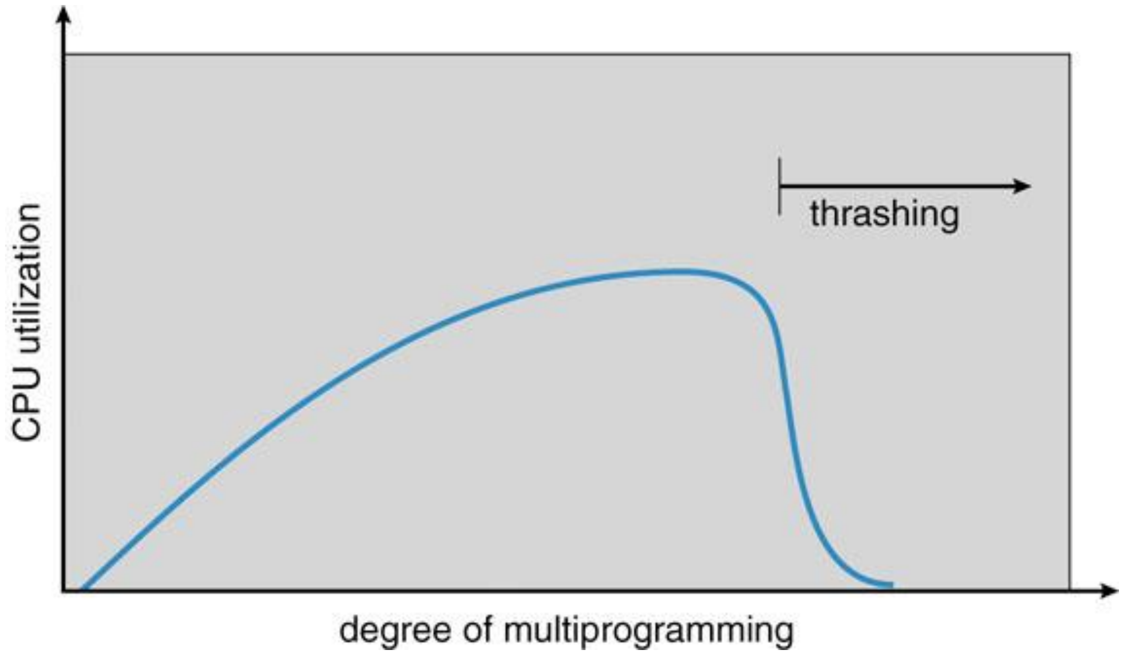
Physical Memory Allocation Polices

- Fixed-space allocation policy
  - Each process is given a limit of page frames it can use
  - When it reaches its limit, it replaces from its own page frames
  - Local replacement: some processes may do well, others suffer

- Variable-space allocation policy
  - Processes' set of pages grows and shrinks dynamically
  - Global replacement: one process can ruin it for the rest
  - Used in Linux

Thrashing

- What happens when physical memory is not enough to hold all the "working sets" of processes
  - Working set: a set of pages that a process is using actively
  - Most of the time is spent by an OS paging data back and forth from disk
  - Possible solutions:
    - Kill processes
    - Buy more memory



- Android's LMK (Low Memory Killer)

Summary

- VM mechanisms
  - Physical and virtual addressing
  - Partitioning, segmentation, paging
  - Page table management, TLBs, etc.
- VM policies
  - Page replacement policy, page allocation policy
- VM optimizations
  - Demand paging, copy-on-write (space)
  - Multi-level page tables (space)
  - Efficient translation using TLBs (time)
  - Page replacement policy (time)