

MEMORY MANAGEMENT: SWAPPING POLICIES

Kai Mast

CS 537

Fall 2022

SWAPPING MOTIVATION

OS Goal: Allow execution even when not enough physical memory available

- Single process with very large address space
- Multiple processes whose total address spaces are large

Process execution should be independent of the amount of physical memory

- Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does swapping work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)

SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk (HDD or SSD)

- Slower, cheaper backing store than memory

Process can run even when not all its pages are loaded into main memory

- OS and hardware cooperate to make large disk seem like memory
- Same behavior (correctness) as if all of address space in main memory

Requirements:

- OS must have a **mechanism** to identify location of each page in address space in memory or on disk
- OS must have a **policy** for determining which pages live in memory and which on disk

SWAPPING MECHANISMS

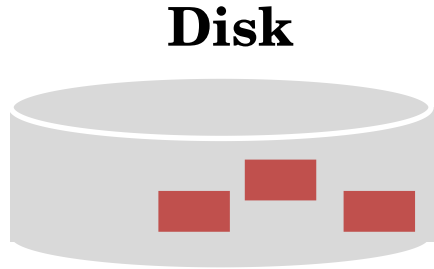
Each page in virtual address space maps to one of three locations:

- Physical main memory: Small capacity, fast performance, expensive cost
- Disk (backing store): Large capacity, slow performance, inexpensive
- Nothing (error): Infinite size, infinitely fast, no cost

Extend page tables with extra bits: **present** and **dirty**

- Permissions (r/w), valid, present, dirty in every PTE
- Page in memory: present bit set
- Page on disk: present bit cleared
 - PTE points to block on disk
 - Causes trap into OS when page is referenced (page fault)
- Page in memory, but in-memory contents don't match those on disk: dirty bit set

PRESENT BIT



Phys. Memory



PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
28	1	rw-	0
4	1	rw-	1

Where does each page reside?

What if we access VPN 0x2?

VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual page number

- if TLB hit, address translation is done; page in physical memory

Else // TLB miss...

- Hardware or OS walk page tables
- If PTE designates page is present, then page is in physical memory
load TLB with VPN -> PPN mapping (replace some other mapping)

Else // Page fault

- Trap into OS (not handled directly by hardware)
- OS selects **victim** page in memory to replace
 - Write victim page out to disk if modified
(if dirty bit was set; clear it)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set (dirty bit is cleared)
- Process continues execution

INTERACTION WITH OS SCHEDULER?

Page faults are very expensive; read page from disk (milliseconds)
(Much different than TLB miss!)

When a process has a page fault, what state is process moved to?

- BLOCKED

When OS finishes reading page into physical memory and sets present bit?

- READY

PAGE SWAPPING POLICIES

(Book Chapter 22)

SWAPPING POLICIES

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions to make

- Page selection

When should a page (or pages) on disk be **brought into** memory?

- Page replacement

Which resident page (or pages) in memory should be **evicted** to disk?

1) PAGE SELECTION

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (**oracle**) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- Problems: **Mispredictions** – needed v/s not-needed

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
- Example: `madvise()` in Unix

2) PAGE REPLACEMENT

Which page in main memory should be selected as **victim**?

- If modified (dirty bit set), write out victim page to disk
- If victim page is not modified (clean), just discard

Optimal (OPT): Replace page not will not be used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages:
 - Requires that OS predict the future
 - Not practical, but good for comparison

PAGE REPLACEMENT

First-in First-Out (FIFO): Replace page that has been in memory the longest

- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement
- Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in the past

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
 - Harder to implement, must track which pages have been accessed
 - Does not handle all workloads well

PAGE REPLACEMENT EXAMPLE

Page reference string (access pattern): ABCABDADBCB

Three pages of
physical memory

Miss count:

Optimal 5

FIFO 7

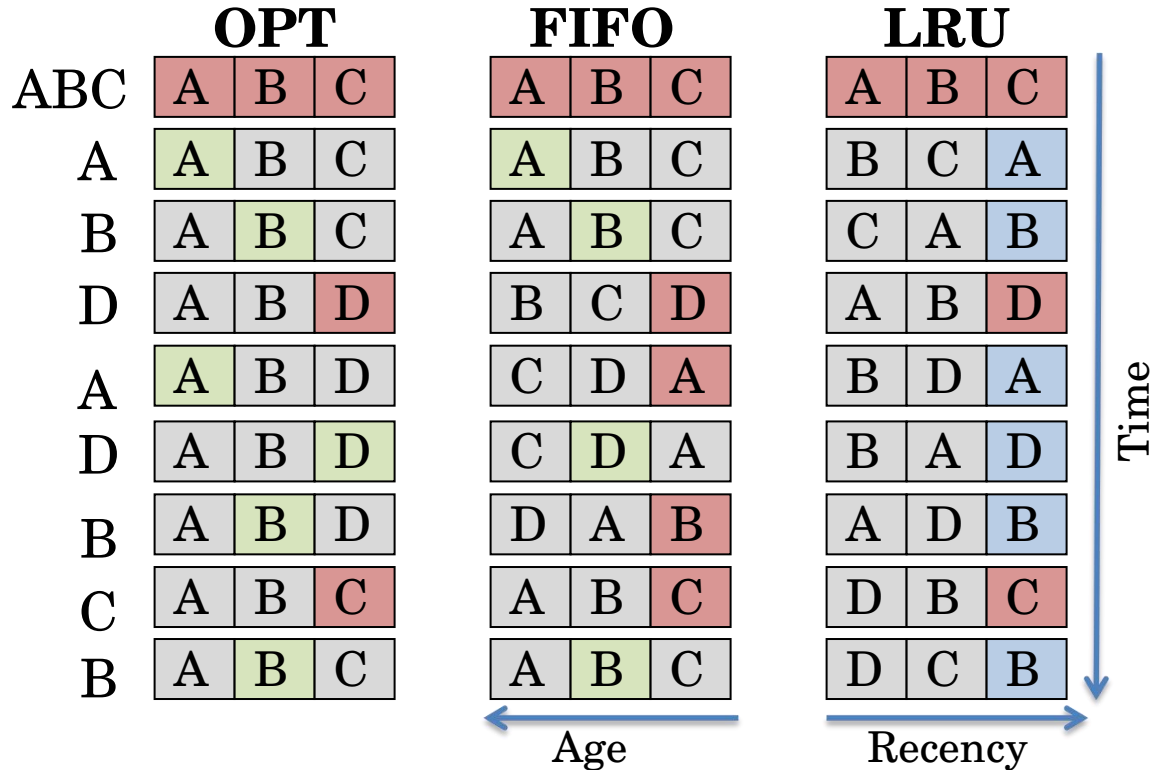
LRU 5

Legend

Miss/Insertion

Hit

Hit+Reorder



PAGE REPLACEMENT COMPARISON

Add more physical memory, what happens to performance?

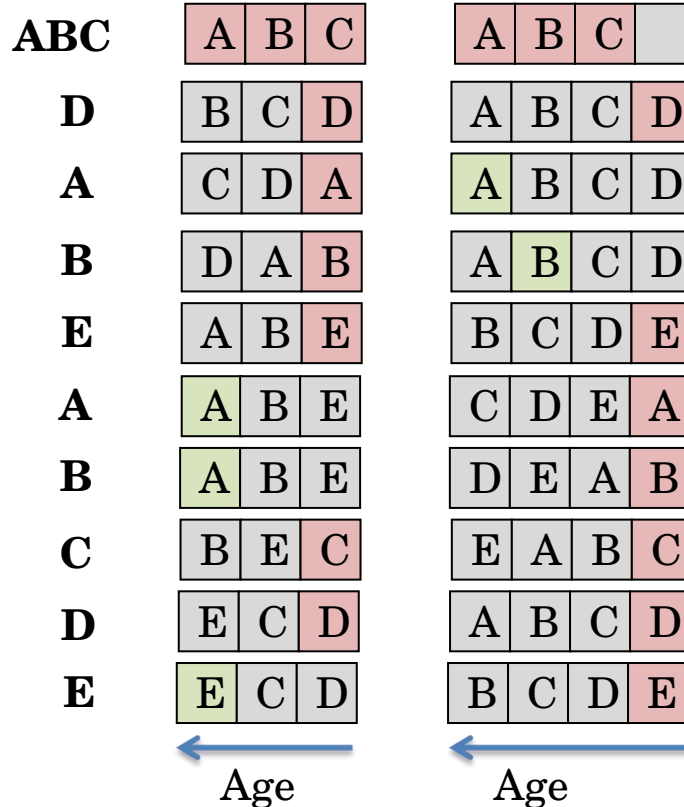
LRU, OPT:

- Guaranteed to have fewer (or same number of) page faults
- Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
- Stack property: smaller cache always subset of bigger (useful for hardware caches...)

FIFO:

- Usually have fewer page faults
- Belady's anomaly: May actually have **more** page faults!

BELADY'S ANOMALY FOR FIFO



Page reference string:
ABCDABEABCDE

Miss count:

3 Pages: 9 misses

4 Pages: 10 misses

Legend

 Miss/Insertion
 Hit

IMPLEMENTING LRU (CONCEPTUALLY)

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement

In practice, systems do not implement perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very oldest

CLOCK ALGORITHM

Hardware

- Keep **use (or reference) bit** for each page frame
- When page is referenced: set use bit

Operating System

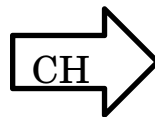
- Page replacement: Look for page with **use bit cleared** (has not been referenced for awhile)
- Implementation:
 - Keep pointer to last examined page frame
 - Traverse pages in circular buffer
 - Clear use bits **as we are searching**
 - Stop when find page with already cleared **use bit, replace this page**

CLOCK: FINDING A PAGE TO EVICT

Need to find victim frame: Which one will the OS pick?

Assume:

- Tiny physical memory: five frames
- We want to load a new page in memory
- Program does not modify use bits (unrealistic)



PFN	Use
0x0	1
0x1	1
0x2	0
0x3	1
0x4	1


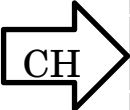

Which would be the first page to evict? 0x2

Which would be the next page to evict? 0x1

Remember:

- Algorithm clears use bits and advance clock hand (CH)
- Only frames with use bit set to 0 can be evicted

CLOCK: FINDING A PAGE TO EVICT

Initial State				After 1st Replacement				After 2nd Replacement			
	PFN	Use	Data		PFN	Use	Data		PFN	Use	Data
	0x0	1	0xfaf		0x0	1	0xfaf		0x0	0	0xfaf
	0x1	1	0xbab		0x1	0	0xbab		0x1	1	0xb37
	0x2	0	0xf00		0x2	1	0x123		0x2	1	0x123
	0x3	1	0xba7		0x3	1	0xba7		0x3	0	0xba7
	0x4	1	0x010		0x4	1	0x010		0x4	0	0x010

Legend

- Use bit cleared
- Replacement

Note: A running program would update use bits between replacements

PROBLEMS WITH LRU-BASED REPLACEMENT

From previous lecture (TLBs): LRU perform poorly when working set is greater than available resources

LRU does not consider **frequency** of accesses

- Is a page accessed **once** in the past equal to one accessed many times?
- Common workload problem:
 - Scanning one large data region flushes memory

Solution: Least Frequently Used (LFU) policy

- Trade-Off: We need to store more data

MEMORY THRASHING

Working Set: Memory locations that must be cached for reasonable hit rate (for individual process)

Balance Set: Collection of working sets of all active processes

Thrashing:

- When a system's memory cache is too small for the current balance set
- System spends all of its time paging in and paging out

What can you do if system is thrashing?

- Do **not schedule** some processes
- **Terminate** some processes (e.g., Out-of-Memory Killer in Linux)

WHERE DO PAGES GO ON DISK?

Kernel organizes regions of virtual memory as “areas” or segments according to how they are swapped

- Data that gets swapped to the same file all goes to a segment
- Multiple memory areas can get swapped to the same file in different places, or to anywhere in a “swap file” or “swap partition”

Challenge: How to find place to swap a page in the swap file?

- Swap map

[Not on midterm/final]

SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation

- Contiguous memory: base, bounds, segmentation
- Using fixed sized pages with pages and linear page tables

Challenges with paging

- Extra memory references: Avoid with TLB
- Page table size: Avoid with multi-level paging

Larger virtual address spaces:

- Paging/swapping mechanisms (present, use, dirty bits)
- Policies for selection and eviction (LRU, Clock)