5/1/24 More Eviction

When to evict?

evict on the critical path of memory allocation

→ when we have to, when there's none left available

→ when free frames fall under a threshold → can be done as a background task, always have frames to allocate when needed

Interaction with swap

→ evict a frame ⇒ allocate a page sized block from the swap partition, note the swap location for future access

→ access an evicted page ⇒ find its swap loc from vm metadata, read content back to memory

→ upon a process exit, free its swapped pages

Eviction Policies: What page/frame to evict?

-> FIFO

-> pick the page that's brought in first (longest time in memory)

-> a queue of frames in order of allocation

-> doesn't care about access patterns

Page accessed in this order : A, B, C, D, A, B, E, A, B, C, D, E

FIFO 3 frames

Frame 1	A	A	A	D	D	D	E	E	E	E	E	E
Frame 2	-	B	B	B	A	A	A	A	A	C	C	C
Frame 3	-	-	C	C	C	B	B	B	B	B	D	D
	x	x	x	x	x	x	x			x	x	

(9 PFs)

Belady's Anomaly

More frames may cause more page faults with FIFO policy

FIFO 4 frames

Frame 1	A	A	A	A	A	A	E	E	E	E	D	D
Frame 2	-	B	B	B	B	B	B	A	A	A	A	E
Frame 3	-	-	C	C	C	C	C	C	B	B	B	B
Frame 4	-	-	-	D	D	D	D	D	D	C	C	C
	x	x	x	x			x	x	x	x	x	x

(10 PFs)

→ Optimal Algorithm (minimum # of page faults)

→ assume we know all memory accesses (including future ones)

→ evict the frame accessed furtherest in the future.

→ Least Recently Used (LRU)

→ use the past to predict the future (like MLFQ)

→ evict page accessed furtherest in the past

→ Belady's anomaly ???

No! N+1 frames contains N+1 recently used pages, a superset of N frames LRU

How to implement LRU?

uses page access as a counter

→ class attempt: tracks # of page accesses since a page is last accessed

→ needs a counter for every mapped page in the system.
lots of data to update on every page access ☹

→ needs to search through all counters to find least recently used page. ☹

→ common sw attempt: tracks a LRU queue

→ moves accessed page to the end on each page access ☹ software intervention is slow

→ evicts from the front of queue (fast!) ☺

→ hw-based attempt: hardware timestamp → read on page table walk or in TLB anyway

→ hw writes the current timestamp to pte on every access. ☺

→ must scan through all ptes to find the LRU page ☹

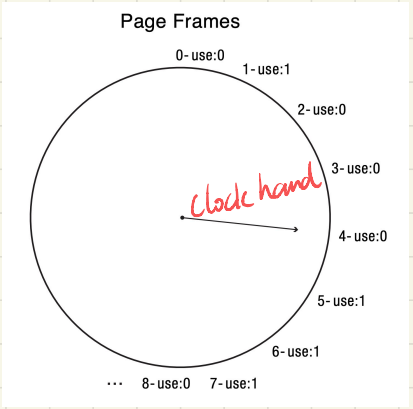
→ Clock (Approximates LRU)

→ hw updates accessed bit of the pte when a page is accessed.

63	62–52	51–M	M-1–12	11–8	7	6	5	4	3	2	1	0	
XD	Prot. Key[4]	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored														0	PTE: not present

★ clock only cares about access info.

→ goes through frames starting at clock hand → stateful (moves after every eviction run)
→ points to a frame




```
for each frame starting at the clock hand:
    if ( pte accessed bit == 0 ) {
        evict page;
        move clockhand forward;
        return;
    } else {
        clear pte accessed bit;
        move clockhand forward; // keep searching
    }
```

→ Cost of Eviction

→ evicts a code page ⇒ no need to evict, already on disk (ELF)

→ cleans stack page ⇒ no need to write out if nothing has been written.

☆ much cheaper to evict a page that doesn't need to be swapped.

→ Second Chance / Enhanced Clock

→ takes cost of eviction into account, uses both accessed & dirty bit

```
if (accessed bit == 1) {
        clear accessed bit; move clockhand;
} else if (dirty bit == 1) {
        clear dirty bit; add to list of dirty pages; move clockhand;
} else {   // both bits are 0
        evict page; move clockhand; return;
}
```