VIRTUAL MEMORY MANAGEMENT PART 2

CS124 – Operating Systems

Spring 2024, Lecture 18

Last Time: Page Replacement Policy

- Last time, began discussing page replacement policies
  - When the OS must allocate a frame but none are available, the page replacement policy chooses a page to evict from a frame
- A very simple replacement policy: FIFO
  - OS maintains a FIFO queue of all pages
  - When a new page is loaded, it is added to the end of the FIFO
  - When a page must be evicted, it is taken from the front of the FIFO
- Exhibits Belady's Anomaly:
  - Sometimes the page-fault rate goes <u>up</u>, as the number of frames in the system is increased
- Also introduced the optimal page replacement policy:
  - Evict the page that will be used furthest in the future
  - Produces the smallest number of page-faults possible
  - Problem: impossible to implement, unless we know the program's entire memory trace

Approximating the Optimal Policy

- Can’t really implement optimal page replacement policy
  - Just like with Shortest Job First (SJF) scheduling, simply cannot predict the future perfectly
- General approach: use the past to predict the future
- Gives rise to the **Least Recently Used** (LRU) policy
  - When a page must be evicted, always choose the one that was used furthest in the past
  - (LRU is basically OPT applied to the reference string in reverse)
- LRU does not exhibit Belady’s Anomaly
- As stated, LRU policy is still quite difficult to implement for virtual memory
  - Typically requires dedicated hardware to implement
  - Problem: must update the data needed to implement LRU on every memory access
  - (And, this data is usually stored in memory as well…)

Least Recently Used Policy

- Two general approaches for implementing LRU policy
- Option 1: Use a counter to record the last time each page is accessed
  - Update the counter on every instruction, or on every memory access
- Extend page-table entries to hold the value of the counter from when the memory was last accessed
  - The MMU must update this value on every page access
- When a page must be evicted:
  - Scan through all pages in memory to find the page with the oldest counter value
- Memory accesses incur additional accesses to update a page's counter-value
  - Can cache values in TLB entries to reduce writes to main memory
- Eviction requires O(N) scan to find the oldest page-counter value

Least Recently Used Policy (2)

- Option 2: Use a queue to track access order of all pages
- When a page is accessed, move it to back of the queue
  - Again, this must happen on every page access
- When a page must be evicted:
  - Page at front of the queue is the least recently used; evict that one!
- As unappealing as counter approach, but in different ways
- Choosing a page to evict is fast and easy…
  - Just pull the first element off the end of the queue
- …but the per-memory-access cost is significantly higher
  - Most accesses will incur linked-list manipulations, requiring multiple additional memory accesses per access
- In practice, LRU is too slow / difficult to implement for virtual memory ☹

Approximating the LRU Policy

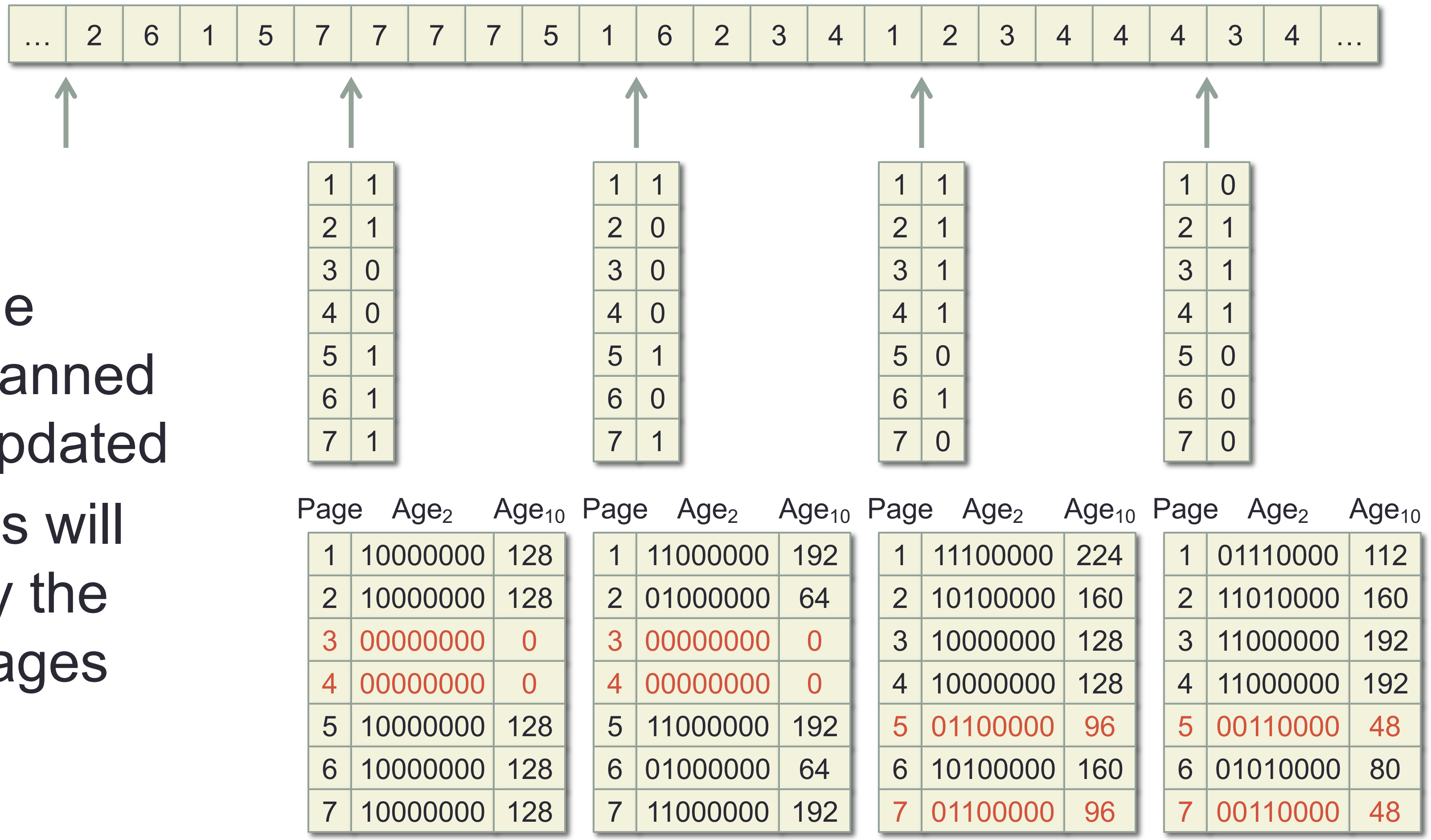
- Systems can implement a policy that approximates LRU
- MMUs usually maintain several bits in page table entries:
  - An “accessed” bit recording if the page was read or written
  - A “dirty” bit recording if the page was written
- Replacement policies can examine the “accessed” bit on regular interval, to see if a page was accessed “recently”
- Example: the **Not Frequently Used** policy
- Maintain a counter for each page in memory
- Periodically scan through all pages on a timer interrupt:
  - If a page’s “accessed” bit is set to 1, increment the page’s counter and clear the page’s “accessed” bit
- When a page must be evicted, choose the page with the lowest count

Approximating the LRU Policy (2)

- Not Frequently Used policy does poorly because it never forgets a page's history
  - e.g. if a page is accessed heavily in the early parts of a program's execution, then never again – it will be unlikely to be paged out
- A much better policy is called the **Aging** policy
- As before, the OS maintains a *b*-bit value for each page
- On a periodic timer-tick, the OS traverses all pages in memory:
  - Shift the page's value to the right by one bit, store the page's "accessed" bit as the new topmost bit, then clear "accessed" bit
- Pages with more recent accesses will have a larger value than pages with less recent accesses
- Evict the page(s) with the lowest value

The Aging Policy

- Example: a process’ memory reference string



…	2	6	1	5	7	7	7	7	5	1	6	2	3	4	1	2	3	4	4	4	3	4	…

- On each timer-tick, the page table table is scanned and age values are updated
- The lowest age values will *approximately* identify the least recently used pages

1	1
2	1
3	0
4	0
5	1
6	1
7	1

Page	Age_2	Age_{10}
1	10000000	128
2	10000000	128
3	00000000	0
4	00000000	0
5	10000000	128
6	10000000	128
7	10000000	128

1	1
2	0
3	0
4	0
5	1
6	0
7	1

Page	Age_2	Age_{10}
1	11000000	192
2	01000000	64
3	00000000	0
4	00000000	0
5	11000000	192
6	01000000	64
7	11000000	192

1	1
2	1
3	1
4	1
5	0
6	1
7	0

Page	Age_2	Age_{10}
1	11100000	224
2	10100000	160
3	10000000	128
4	10000000	128
5	01100000	96
6	10100000	160
7	01100000	96

1	0
2	1
3	1
4	1
5	0
6	0
7	0

Page	Age_2	Age_{10}
1	01110000	112
2	11010000	160
3	11000000	192
4	11000000	192
5	00110000	48
6	01010000	80
7	00110000	48

The Aging Policy (2)

- The main difference between aging policy and LRU is that aging has a *much* lower resolution on its “recency” info
- With aging policy, common to have multiple pages with the same age value
  - LRU policy would know exactly which page was accessed furthest in past, but aging policy treats them the same
- Similarly, if two pages have a value of 0:
  - LRU would know which one was accessed most recently, but aging views both as having not been accessed recently
- Nonetheless, aging policy generally performs very well with a relatively small number of bits, e.g. 8 or 16 bits per page

Other Policies Using the Accessed Bit

- Many other replacement policies that use “accessed” bit
- Example: make FIFO policy more intelligent
  - Original policy: always evict the page at the front of the FIFO
  - Tweak this policy to also use a page’s “accessed” bit
- When a page must be evicted:
  - Consider the page at the front of the FIFO
  - If the page’s “accessed” bit is 1, clear the “accessed” bit and then move the page back to the end of the FIFO
  - Otherwise, evict the page at the front of the FIFO
- Called the **Second-Chance replacement policy**
  - If a page has been accessed during its time in the FIFO, it is given a second chance

Second-Chance Replacement Policy

- Second-chance policy:
  - Consider the page at the front of the FIFO
  - If the page's "accessed" bit is 1, clear the "accessed" bit and then move the page back to the end of the FIFO
  - Otherwise, evict the page at the front of the FIFO
- What happens if <u>all</u> pages have their "accessed" bits set?
  - Pager will scan through all pages in the FIFO…
  - Every page's "accessed" bit will be cleared during this pass…
  - On second pass, pager will simply evict the first page in the FIFO

- Second-chance policy degenerates to FIFO replacement if all pages have been accessed since the last page-eviction

The Clock Replacement Policy

- The **Clock** replacement policy is a more efficient implementation of the second-chance policy
  - But, it implements the exact same policy
- Pages are maintained in a circular queue
- A “clock hand” points to the next page to be considered for eviction
- When a page must be evicted:
  - The page currently referenced by the clock hand is considered
  - If the page’s “accessed” bit is currently set, it is cleared and the clock hand is advanced
  - Otherwise, the page under the clock hand is evicted
- Clock is more efficient to implement than second-chance because it requires little to no linked-list manipulation

The Not Recently Used Policy

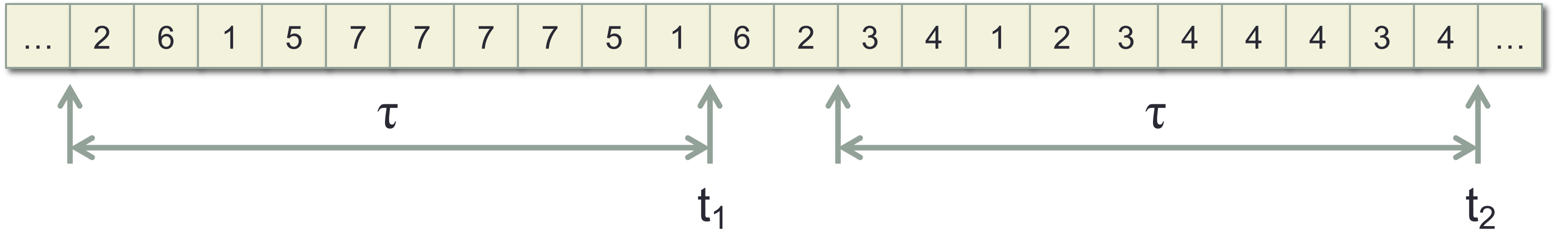
- The **Not Recently Used** policy is a very simple policy that relies on both the “accessed” and “dirty” bits
- A timer interrupt periodically scans through all pages in memory, clearing the “accessed” bit each page
- Pages are classified based on “accessed” and “dirty” bits:
  - Class 0: not accessed, not dirty
  - Class 1: not accessed, dirty
    - Occurs when a page has been written, but isn’t accessed again after the timer interrupt clears the page’s “accessed” bit.
  - Class 2: accessed, not dirty
  - Class 3: accessed, dirty
- When a page must be evicted, choose a page from the lowest numbered non-empty class
- Always prefers to keep pages that were recently accessed; of the not-accessed pages, prefers to avoid incurring I/O costs

A Working-Set Based Policy

- Another page replacement policy is based on the **working set** of a process
    - The set of pages the process is currently using for its computations
- As a program runs, its working set will change over time (i.e. as it goes through different phases of computation)
    - Pages will enter and leave the working set of each process
- Ideally, a page replacement policy should only evict pages that are outside of a process' current working set
    - If the policy evicts pages that are still in the current working set, this will increase the page-fault rate
- How do we approximate a process' working set?
- Can we create a policy that uses this information?

A Working-Set Based Policy (2)

- Example: a process’ memory reference string



…	2	6	1	5	7	7	7	7	5	1	6	2	3	4	1	2	3	4	4	4	3	4	…

τ τ

t_1 t_2

- Approximate the process’ working set at time t by looking at all the page access of the process from t – τ until t
  - τ is a tunable parameter specifying a window size (above, $\tau = 10$)
  - Want to choose τ large enough to completely capture the process’ working set, but not so large that it has pages outside working set
- At time t_1, the program has working set {1, 2, 5, 6, 7}
- At later time t_2, the program has working set {1, 2, 3, 4}
  - Ideally, policy will evict pages 5, 6, 7

The WSClock Policy

- The WSClock policy tries to take a process' working set into account when making paging decisions
  - Combines several policies (Clock, NRU), as well as attempting to identify the process' working set
- The system maintains a virtual clock for each process
  - e.g. the total time that the process has actually run on the CPU
  - Premise: If a given page has been accessed within τ of the current virtual time, it is still in the process' working set
- As with Clock policy, pages are kept in a circular queue
  - Each page also has a "time of last use" field that approximates when the page was actually used last
- On a periodic timer interrupt, all pages are examined:
  - If a page's "accessed" bit is 1, the page's "time of last use" value is set to the current virtual time, and the "accessed" bit is set to 0

The WSClock Policy (2)

- When a page must be evicted, the page under the clock hand is examined:
    - If the “accessed” bit is 1, page is clearly in the process’ working set. The “accessed” bit is cleared, and hand is advanced to next page.
- If the “accessed” bit is 0, the page may or may not be in the working set. So, examine “time of last use” value:
    - If time of last use is within τ of the current virtual time, the page is still in the current working set. Again, advance the clock hand.
- Otherwise, the page is outside the process’ working set
- Still two possibilities:
    - The page may be clean or dirty!
    - Want to avoid the I/O overhead of evicting a dirty page…

The WSClock Policy (3)

- WSClock cont. (found a page outside the working set…)
- If the page being considered is dirty, don’t want to evict it
  - Instead, schedule the page to be written back to disk, and continue looking for a clean page
- If the page being considered is clean, no cost for eviction!
  - Use the frame to load the new page

- This bias against evicting dirty pages is an aspect of NRU

The WSClock Policy (4)

- What if we traverse all pages while looking for a victim?

- Possibility 1: at least one write was scheduled…
  - Solution: Just keep traversing the list of pages until we find a clean page to evict. A scheduled write will eventually complete…

- Possibility 2: no writes were scheduled ☹
  - Implies that all pages are currently in the working set ☹
  - Solution: Just choose any clean page and evict it.
    Or, if there are no clean pages, just evict the current page.

Page Buffering

- Can enhance page replacement policies with **page buffering** techniques
- Very common for OSes to maintain a pool of “free page frames” available for use when page-faults occur
  - A faulting process will immediately have an available frame to use
  - Doesn’t have to wait for “page eviction” steps to take place (e.g. identify a page for eviction, possibly write back a dirty page, etc.)
- Pages are periodically reclaimed from active processes
  - This is no longer technically an “eviction”; rather, the page is now a *candidate* for eviction
- Reclaimed pages are added to an appropriate pool:
  - Clean pages are put into a free-frame pool for handling new faults
  - Dirty pages are added to a list of modified pages; these pages are written to disk when convenient, then added to the free-frame pool

Page Buffering (2)

- Free page-frame pools can be used to reverse bad decisions made by pagers
  - Until a free page frame is reused, it will still have its old contents…
- If a page fault occurs, and the faulting page is still in a free page-frame pool, simply pull it back out of the pool
  - Don’t need to actually load it from the disk in this situation
- Example: DEC VAX/VMS computer systems
  - Early VAX hardware didn’t implement the “accessed” bit correctly
  - The OS could tell if a page was dirty, but not if it was accessed…
  - VMS used FIFO replacement policy enhanced with page buffering
- If the FIFO policy reclaimed a page that was still in active use:
  - The process would eventually page-fault when accessing the page…
  - VMS checks the modified and free page-frame pools for the page; if still present, page is reinserted into the process’ address space

OS Emulation of Accessed/Dirty Bits

- Not all MMUs include support for “accessed” and “dirty” bits!
  - e.g. ARM processors with an MMU simply don’t have these bits
  - (some ARM CPUs don’t have an MMU)
- If an OS needs these bits for virtual memory management, it must emulate them using protections and page faults
- Example: Linux on ARM maintains two page-tables
  - The native-ARM page table doesn’t include “accessed” and “dirty” bits, but it can specify memory protections, e.g. read-only
  - The Linux kernel version of the page table does include these bits
- Linux virtual memory system can set pages to be read-only…
  - When protection fault occurs, then corresponding “dirty” bit can be set
- A similar process can be used for “accessed” bits:
  - Unmap the page; when it is accessed, it will generate a fault
  - In page-fault handler, remap the page and set the “accessed” bit to 1

Other Replacement Policies

- Many other interesting page replacement policies
  - OSes tend to have policies that are tuned in various ways
- Example: LRU-K policies
  - Examines the time of the Kth most recent access, not just the most recent access
  - (LRU == LRU-1)
- Very common to see LRU-2, which uses the time of the second-most-recent memory access
  - Prefers pages that have been accessed twice recently, over pages that have been accessed twice over a longer period of time
  - Combines both recency and frequency considerations in choosing a page to evict
- For certain program behaviors, LRU-2 outperforms LRU
  - e.g. LRU-2 is **scan-resistant** – it will quickly evict pages that are scanned through once, and then not accessed again

Adaptive Replacement Cache

- Example: Adaptive Replacement Cache (ARC) policy
  - Developed and patented by IBM
  - (This has dissuaded its adoption in open-source projects)
- Maintains two LRU queues:
- L_1 is LRU queue for pages accessed only once
  - L_1 captures recency information for the policy to use
- L_2 is LRU queue for pages accessed at least twice
  - L_2 captures frequency information for the policy to use
- Each LRU queue is divided into top and bottom regions
  - Only the top regions hold pages that are still in memory
  - Pages in the bottom regions have already been evicted, and are called **ghost entries**

Adaptive Replacement Cache (2)

- Ghost entries can be used to tune the cache's behavior
- When a page fault occurs:
  - If the page is still a ghost entry in either L_1 or L_2 queue, ARC can increase the size of either the L_1 or L_2 queue as needed
  - ARC can choose whether it should care more about recency or frequency of access in page-eviction decisions
- ARC generally performs <u>much</u> better than LRU
  - Can achieve greater hit rates than LRU with same cache size, or can achieve same hit rates as LRU with a much smaller cache

- Many other self-tuning cache algorithms now…
- Example: Clock with Adaptive Replacement (CAR)
  - Is self-tuning like ARC, and also generally outperforms LRU

Next Time

- Pintos virtual memory project – design guidance