Virtual Memory and Paging (4)

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

Paging Policies

- At this point, we have all the mechanisms but we need to define the policies, namely
  - The Page Replacement Policy: how to pick victims?
  - The Frame Allocation Policy: how many frames to each process?
- The main goal: Minimize page faults
- Contrast with the CPU though:
  - CPU Scheduling
    - The CPU is so fast that the decisions have to be made very quickly
    - Therefore, algorithms need to be simple
  - Memory Scheduling
    - The disk is so slow that it is worth spending some time to make a decision
    - Avoiding a few more page faults can have a large impact on performance
    - More sophisticated algorithms may be worthwhile
    - As usual the OS works with imperfect/partial information (e.g., no knowledge of the future, no knowledge of what jobs will do)

Page Replace Policy

- Let’s define the Page Replacement Problem
- Problem Input
  - A set of page references
  - A number of available frames allocated to the process
- Problem Objective: Minimize the number of page faults

- This is a computational difficult problem (as usual)
- Let’s look at examples and how 3 standard algorithms would work on them...

Optimal Page Replacement

- Of course we all want optimal algorithms for everything
- If we have perfect knowledge of the future, we can make optimal page replacement decisions
- Not feasible in practice, but useful to have an upper bound on how well we could do in an ideal scenario
  - If I have an algorithm that in practice is 1% worse than the optimal unfeasible algorithm, I can say that the algorithm is “very good”

- Optimal algorithm: evict the page that will not come in use for the longest time (assuming I know the future)
  - Think about it, it makes sense...
- Let’s go through an example with the following page reference sequence:
  7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- Assuming that the process is allocated **3 frames only**

Example: Optimal Algorithm

References	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame #0	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
Frame #1		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
Frame #2			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Page faults	x	x	x	x		x		x			x			x				x		

- We have a total of 9 page faults - this is the best we can do
- Let's now look at a simple algorithm that does not assume we know the future (because we don't)

Example: FIFO Page Replacement

- FIFO: Kick out the oldest page brought to memory

References	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame #0	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
Frame #1		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
Frame #2			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Page faults	x	x	x	x		x	x	x	x	x	x			x	x			x	x	x

- We have a total of 15 page faults
- The problem with FIFO is that an old page may be used all the time
- So it is likely better to keep track of when a page was last used
- This leads us to our 3rd algorithm...

Example: LRU Page Replacement

- LRU: Kick out the least recently used/accessed page

References	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame #0	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
Frame #1		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
Frame #2			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
Page faults	x	x	x	x		x		x	x	x	x			x		x		x		

- We have a total of 12 page faults
- LRU is generally considered a “good” algorithm
- Question: How to keep track of the last time of use for each frame?

How to Implement LRU?

- Use counters?
  - Augment each page table entry with a “time of use” field
  - Increment a “clock” counter each time a memory access is performed
  - Update the “time of use” field with the clock value
  - When eviction is necessary search for the minimum “time of use” field: it is the victim frame
  - High-overhead
- Use a stack?
  - A frame is moved to the top of the stack after it is referenced
  - Requires a bunch of pointers shuffling
  - But the victim is always at the bottom of the stack

- The usual bad news is that...

Help from the Hardware?

- If the hardware does not provide any dedicated component, overhead to do anything other than FIFO is too expensive :(
- OSes do not implement LRU page replacement

- But the hardware usually provides a reference bit
  - Associated to each entry in each page table entry, and initially set to 0
  - Set to 1 by the hardware when the page is referenced
  - Settable to 0 by the OS
- Can be used to make (somewhat) enlightened decisions

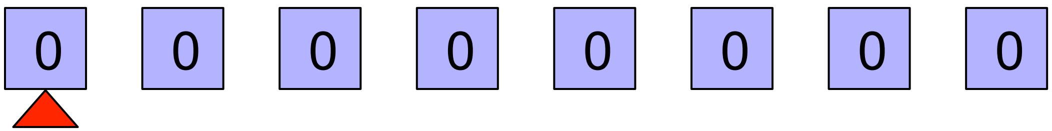
- One can do approximate LRU using the reference bit

Approximating LRU: The Clock Algorithm

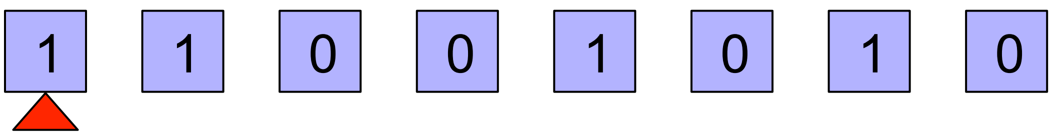
- What OSes do: The Clock Algorithm
- Key idea: use one reference bit per frame
- Whenever a page is referenced by the program, set its frame’s reference bit to 1
- When a page in a frame needs to be evicted:
  - If the reference bit is 1, set it to 0, and move the queue head to the next item in the queue
  - If the reference bit is 0, evict the page in that frame
- A page in a frame that keeps on being referenced is never evicted (its reference bit is always 1)

Clock Algorithm (8-frame Example)

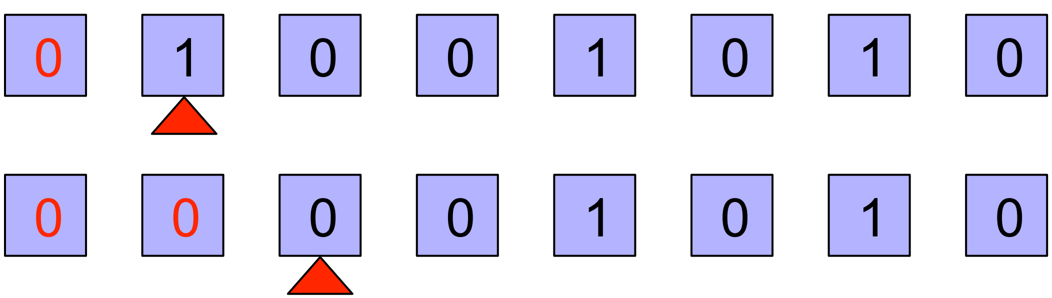
- Initially all reference bit are set to 0 and the head of the queue is (say) positioned on the first bit (the one for the first frame)



- As time goes on, frames are referenced by processes, so that some reference bits are set to 1... For example:

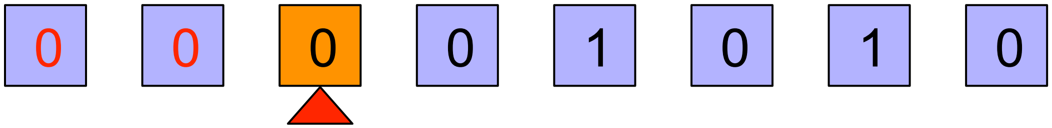


- Now a page fault happens and we have to find a victim
- While we “see” a 1 under the head, we set it to 0 and move the head to the right…

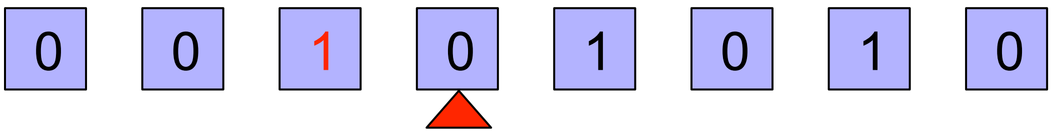


- We now see a zero: that’s our victim frame (frame 2 in this example)

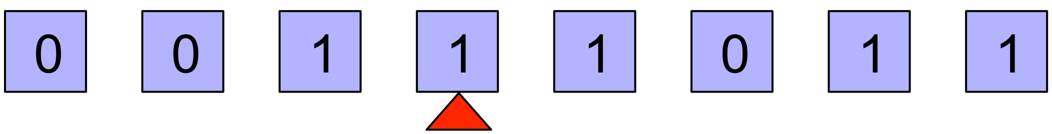
Clock Algorithm (8-frame Example)



- The victim frame is evicted and a new page is loaded and referenced, updating the reference bit. The pointer advances



- Before the next page faults, more frames have been references and more reference bits have been updated…



- Say now we have a page fault? Which will be the next victim?

Frame 5 (the 6th frame)

(The first frame with a 0 reference bit when moving the head to the right)

Approximate LRU works!

- Make sure you read OSTEP 22.6-22.8, which talks about page replacement and shows simulation results like this one
  - This one is with some locality: 80% of references go to 20% of pages
- Take-away: The Clock algorithm is really close to LRU
- It's a good approximation of it!

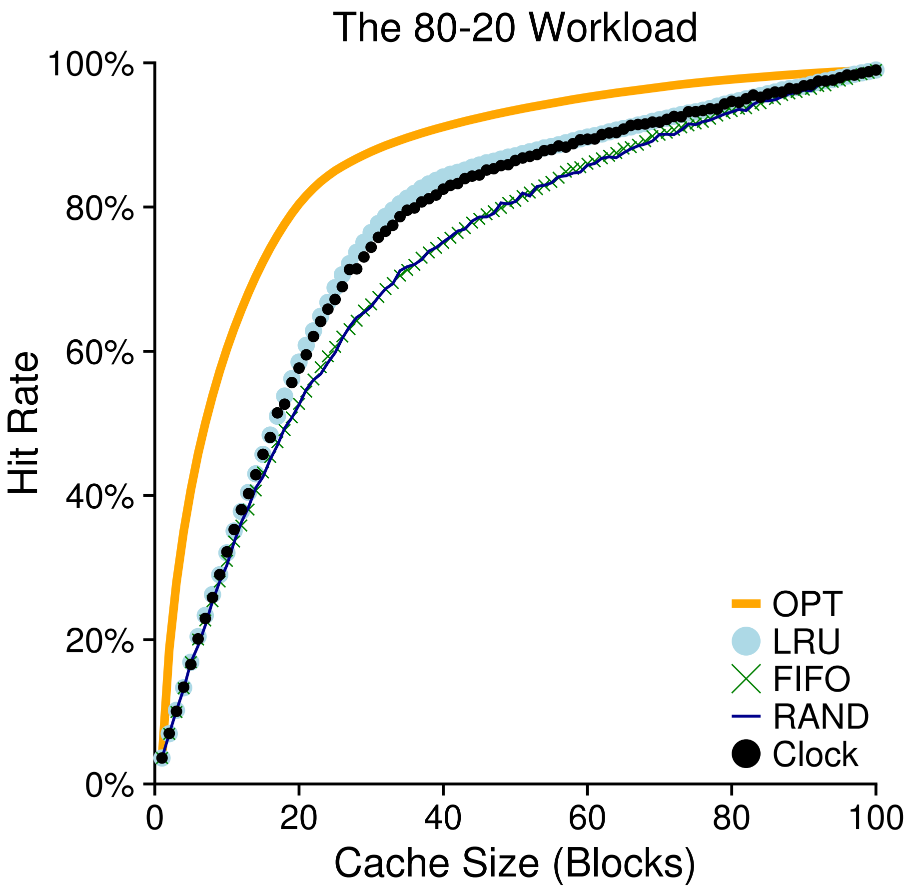


Figure 22.9: **The 80-20 Workload With Clock**

Global/Local Replacement

- Local Replacement: Victim among the process pages
  - Limits the number of frames per process
- Global Replacement: Any victim can be selected
  - Good for high-priority processes
  - Performance of one process depends on other processes
- Global is generally used: simple and increases system throughput
- So yes, your process could lose pages because my process is page-faulting!
  - It’s a jungle out there

Frame Allocation Algorithms

- The Frame Allocation Problem: How many frames should be given to a process?
- Maximum number of frames: The physical memory
  - But making one process happy is not going to please the other processes…

Frame Allocation Policies

- Fair Allocation: m frames, n processes: Give each process m/n frames
- Proportional Allocation: if s_i is the size of process i, and $S = \Sigma_i s_i$ is the total size, give $s_i / S \times m$ frames to process i
- Priority allocation: tweak the above with priorities

- Current OSes implement variations on these themes

Thrashing

- Phenomenon observed on systems with a global page replacement policy and a high-level of multi-programming (many processes) using the whole memory (e.g., a server)

- A process needs more frames, and so its page-fault rate increases
- It takes frames away from other processes, increasing their page-fault rates
- These processes are moved from the ready queue to the waiting one (since they are waiting for the disk)
- The CPU utilization decreases
- Which is good for the CPU scheduler: It can start new processes!
- The first thing these new processes do is page fault, and they are sent to the waiting queue right away
- At this point: No work gets done because each process is waiting for pages

- This is called thrashing
- Note the paradox: To increase the CPU utilization the multi-programming level must be reduced
  - The CPU scheduler is blind to memory issues :(

Thrashing Prevention

- Working Set Strategy:
  - Observe the pages referenced by each process (called the working set)
  - When the sum of the sizes of all working sets gets greater than the number of memory frames, swap out an entire process and reclaim its frames
  - Hence no thrashing (but one very unhappy process)
- Page-Fault Frequency Strategy:
  - Monitor the page-fault rate for each process
  - If the rate is above some (fixed) upper bound, give the process another frame
  - If the rate is below some (fixed) lower bound, take a frame from the process
  - If a process requests a new frame but none is available: swap it out

- “Thrashing” and “swapping” are often use interchangeably. Formally though thrashing is the problem and swapping is the solution.

Conclusion

- An address space is a bunch of non-contiguous pages (but virtualized as a big slab)
- Process Address Spaces can only be partially in memory
- Main issues:
  - Page Replacement Policy
  - Frame Allocation Policy
- Thrashing is bad

- There was A LOT of content in these 4 sets of lecture notes (and we skipped many details!)
  - OSes do exactly what we described conceptually, but use many tricks
  - In particular, to make sure page tables are not too large!