VM Page Replacement

Emin Gun Sirer

All Paging Schemes Depend on **Locality**

- Processes tend to reference pages in localized patterns
  - Temporal locality
    - locations referenced recently likely to be referenced again
  - Spatial locality
    - locations near recently referenced locations are likely to be referenced soon.
- Goal of a paging system is
  - stay out of the way when there is plenty of available memory
  - don’t bring the system to its knees when there is not.

Demand Paging

- Demand Paging refers to a technique where program pages are loaded from disk into memory as they are referenced.
- Each reference to a page not previously touched causes a page fault.
- The fault occurs because the reference found a page table entry with its valid bit off.
- As a result of the page fault, the OS allocates a new page frame and reads the faulted page from the disk.
- When the I/O completes, the OS fills in the PTE, sets its valid bit, and restarts the faulting process.

Paging

- Demand paging
  - don’t load page until absolutely necessary
  - commonly used in most systems
  - doing things one at a time can be slower than batching them.
- Prepaging
  - anticipate fault before it happens
  - overlap fetch with computation
  - hard to predict the future
  - some simple schemes (hints from programmer or program behavior) can work.
    - `vm_advise`
    - larger “virtual” page size
    - sequential pre-paging from mapped files
High Level

- Imagine that when a program starts, it has:
  - no pages in memory
  - a page table with all valid bits off
- The first instruction to be executed faults, loading the first page.
- Instructions fault until the program has enough pages to execute for a while.
- It continues until the next page fault.
- Faults are expensive, so once the program is running they should not occur frequently, assuming the program is “well behaved” (has good locality).

Page Replacement

- When a fault occurs, the OS loads the faulted page from disk into a page of memory.
- At some point, the process has used all of the page frames it is allowed to use.
- When this happens, the OS must replace a page for each page faulted in. That is, it must select a page to throw out of primary memory to make room.
- How it does this is determined by the *page replacement algorithm*.
- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.

Finding the Best Page

- A good property
  - if you put more memory on the machine, then your page fault rate will go down.
  - Increasing the size of the resource pool helps everyone.
- The best page to toss out is the one you’ll never need again
  - that way, no faults.
- Never is a long time, so picking the one closest to “never” is the next best thing.
  - Replacing the page that won’t be used for the longest period of time absolutely minimizes the number of page faults.
  - Example:

Optimal Algorithm

- The optimal algorithm, called Belady’s algorithm, has the lowest fault rate for any reference string.
- Basic idea: replace the page that will not be used for the longest time in the future.
- Basic problem: phone calls to psychics are expensive.
- Basic use: gives us an idea of how well any implementable algorithm is doing relative to the best possible algorithm.
  - compare the fault rate of any proposed algorithm to Optimal
  - If Optimal does not do much better, then your proposed algorithm is pretty good.
  - If your proposed algorithm doesn’t do much better than Random, go home.
**Evaluating Replacement Policies**

Effective Access Time: \((1-p)T_m + pT_d\)
- \(T_m\) = time to access main memory
- \(T_d\) = time to fault

Execution time = (roughly) \#memory ref * E.A.T.

<table>
<thead>
<tr>
<th>Few Frames</th>
<th># of Physical Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>In here, forget it.</td>
</tr>
<tr>
<td>Opt</td>
<td>In here, you can expect to have some effect.</td>
</tr>
<tr>
<td>FIFO</td>
<td>Both is this range. It doesn’t matter so much what you do.</td>
</tr>
</tbody>
</table>

**FIFO**

- FIFO is an obvious algorithm and simple to implement.
- Basic idea: maintain a list or queue of pages in the order in which they were paged into memory.
- On replacement, remove the one brought in the longest time ago.
- Why might it work?
  - Maybe the one brought in the longest ago is one we’re not using now.
- Why it might not work?
  - Maybe it’s not.
  - We have no real information to tell us if it’s being used or not.
- FIFO suffers from “Belady’s anomaly”
  - the fault rate might actually increase when the algorithm is given more memory – a bad property.

**An Example of Optimal and FIFO in Action**

Reference stream is A B C A B D A D B C

**OPTIMAL**

- 5 Faults
  - toss A or D

**FIFO**

- 7 Faults
  - toss C

**Least Recently Used (LRU)**

- Basic idea: we can’t look into the future, but let’s look at past experience to make a good guess.
- LRU: on replacement, remove the page that has not been used for the longest time in the past.
- Implementation: to really implement this, we would need to time stamp every reference, or maintain a stack that’s updated on every reference. This would be too costly.
- So, we can’t implement this exactly, but we can try to approximate it.
  - why is an approximate solution totally acceptable?
Using the Reference Bit

- Various LRU approximations use the PTE reference bit.
  - keep a counter for each page
  - at regular intervals, do:
    - for every page:
      - if ref bit = 0, increment its counter
      - if ref bit = 1, zero its counter
    - zero the reference bit
  - the counter will thus contain the number of intervals since the last reference to the page.
  - the page with the largest counter will be least recently used one.
- If we don’t have a reference bit, we can simulate it using the VALID bit and taking a few extra faults.
  - therefore want impact when there is plenty of memory to be low.

LRU Clock (Not Recently Used)

- Basic idea is to reflect the passage of time in the actual data structures and sweeping method.
  - Arrange all of physical pages in a big circle (a clock).
  - A clock hand is used to select a good LRU candidate:
    - sweep through the pages in circular order like a clock.
    - if the ref bit is off, it’s a good page.
    - else, turn the ref bit off and try next page.
  - Arm moves quickly when pages are needed.
  - Low overhead when plenty of memory
  - If memory is big, “accuracy” of information degrades.
    - add in additional hands

Fixed Space Vs. Variable Space

- In a multiprogramming system, we need a way to allocate memory to the competing processes.
  - Question is: how to determine how much memory to give to each process?
- In a fixed-space algorithm each process is given a limit of pages it can use; when it reaches its limit, it starts replacing new faults with its own pages. This is called local replacement.
  - some processes may do well while others suffer.
- In variable-spaced algorithms, each process can grow or shrink dynamically, displacing other process’ pages. This is global replacement.
  - one process can ruin it for the rest.

Working Set Model

- Peter Denning defined the working set of a program as a way to model the dynamic locality of a program in execution.
- Definition:
  \[ WS(t,w) = \{ \text{pages i s.t. i was referenced in the interval } (t,t-w) \} \]
  - \( t \) is a time, \( w \) is the working set window, a backward looking interval, measured in references.
- So, a page is in the WS only if it was referenced in the last \( w \) references.
Working Set Size

- The working set size is the number of pages in the working set; i.e., the number of pages touched in the interval \([t, t-w]\).
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - so, within that period of time, you will have a larger working set size.
- For some parameter \(w\), we could keep the working sets of each process in memory.
- Don’t run process unless working set is in memory.

Page Fault Frequency

- PFF is a variable space algorithm that uses a more ad hoc approach.
- Basic idea:
  - monitor the fault rate for each process
  - if the fault rate is above a high threshold, give it more memory
  - should work fine
  - if it doesn’t always
    - if the rate is below a low threshold, take away memory
    - should work more
    - but doesn’t always
  - hard to tell between changes in locality and changes in size of working set.

What do you do to pages?

- If the page is dirty, you have to write it out to disk.
  - record the disk block number for the page in the PTE.
- If the page is clean, you don’t have to do anything.
  - just overwrite the page with new data
  - make sure you know where the old copy of the page came from
- Want to avoid THRASHING
  - when a paging algorithm breaks down
  - most of the OS time spent in ferrying pages to and from disk
  - no time spent doing useful work
  - the system is OVERCOMMITTED
    - no idea what pages should be resident in order to run effectively
- Solutions include:
  - SWAP
  - Buy more memory