Jessica Nguyen

CSS430

Cache.java

# SPECIFICATION/DESIGN ANALYSIS

## Private class

* 1. Entry: represents an individual Page object in the Kernel page table
  2. Private fields: int frameNumber -- Contains the disk block number of cached data

boolean reference -- Is set to true whenever this block is accessed

boolean dirty -- Is set to true whenever this block is written

## Private fields

* 1. Entry[] pageTable -- an array of Entry's represents pageTable
  2. Vector<byte[]> cache -- a vector of byte arrays that acts as the cache
  3. int blockSize -- a field that holds the block Size for the cache
  4. int victim -- int to keep track of current victim

## Private helper methods

* 1. findFreePage() -- finds and returns index of a free page in pageTable
  2. nextVictim() -- performs enhanced 2nd chance algorithm

postconditions: returns the index for the next victim that will be selected from the enhanced 2nd chance algorithm

* 1. writeBack() -- writes virtual memory from pageTable to physical memory on the disk

preconditions: victimEntry >= 0

postconditions: Writes back all dirty blocks to Disk

## Public methods

* 1. boolean read() -- reads into the buffer[ ] the contents of the cache block specified by blockId

preconditions: blockId >= 0, buffer is initiated to a valid byte array

postconditions: returns false if blockId < 0

returns true if succesfully reads contents of cache to buffer

* 1. boolean write() -- Writes the contents of buffer[ ] array to the cache block specified by blockId

preconditions: blockId >= 0, buffer is initiated to a valid byte array

postconditions: returns false if blockId < 0

returns true if succesfully writes contents to cache

* 1. void writeBack() -- writes virtual memory from pageTable to physical memory on the disk

preconditions: victimEntry >= 0

postconditions: Writes back all dirty blocks to Disk

* 1. void sync() -- Writes back all dirty blocks to Disk
  2. void flush() -- Writes back all dirty blocks to Disk and wipes all cached blocks

# PERFORMANCE SCREENSHOT

# PERFORMANCE ANALYSIS

1. Disabled vs Enabled cache
   1. In random access, disabled time is faster than enabled time. This could be explained by the randomness of data. And some portion of the memory access will miss the cache. For those times that the memory misses the cache, it has to reach inside the disk. Not having a cache makes the program look directly into the disk instead of spending time in the cache, making the time faster.
   2. In localized access, disabled time is much slower than enabled time. This is because in localized access, it gets a high ratio of cache hits. Thanks to the memory blocks being already there in the cache, we can return those blocks, rather than reaching farther into the hard drive, which takes more time. Without the cache, the process will take longer because we have to perform swapping, which produces overhead and takes longer
   3. In mixed access, disabled time is slower than enabled time. This is because there is a mix of high and low cache hits, so the presence of a cache there helps out a lot of times when a request does hit the cache. Having a cache there is better than none, because it helps reduce overhead in paging memory
   4. In adversary access, disabled time is faster than enabled time. This is because, in adversary testing, it generates disk accesses that do not make good use of the disk cache at all. The memory string purposely accesses non-present disk blocks to create cache misses. So not having a cache actually speeds up the process, because the program had to spend time looking into the cache, knowing the targetted memory won’t be there anyway. Not having a cache makes the program look directly into the disk instead of spending time in the cache, making the time faster.
2. Random access vs localized access
   1. Random access’s overall time is slower than localized access. This is because randomness can cause some memory requests to miss the cache, whereas localized access seeks to always hit the cache. When it misses the cache in random access, it has to reach inside the disk, which takes more time because of overhead
3. Localized access vs adversary access
   1. Localized access’s overall time is faster than adversary access. This is because localized access gets a higher ratio of cache hits whereas in adversary access, it accesses non-present disk blocks to create cache misses. When it misses the cache in random access, it has to reach inside the disk, which takes more time because of overhead
4. Overall results
   1. Overally in time, localized > (random == mixed) > adversary testing. Localized time is the fastest since it gets the most cache hits, therefore avoiding trips to the disk that required overhead time. Random and mixed testing receives a mixture of inputs, therefore it will depend on how “lucky” the cache is to have all the memory in place at the right time. Adversary testing is the slowest because it gets the least cache hits. Therefore, it spends a lot of time going to the disk, which generates overhead time.