**CSS 430 OPERATING SYSTEMS**

**Mariana Chagoyan**

**Program4 7**

**Report 4 Part 2:**

**Team Member(Kevin Parker)**

* **Random Accesses with Cache Disable**d

l Test4 disabled 1

Average Time Write: 40msec Average Time Read40 msec

l Test4 disabled 1

Average Time Write: 40msec Average Time Read40 msec

l Test4 disabled 1

Average Time Write: 40msec Average Time Read40 msec

* **Random Accesses with Cache Enabled**

l Test4 enabled 1

Average Time Write: 40msec Average Time Read44 msec

l Test4 enabled 1

Average Time Write: 36msec Average Time Read40 msec

l Test4 enabled 1

Average Time Write: 38msec Average Time Read42 msec

* **Localized Accesses with Cache Disabled**

l Test4 disabled 2

Average Time Write: 233msec Average Time Read233 msec

l Test4 disabled 2

Average Time Write: 233msec Average Time Read233 msec

l Test4 disabled 2

Category test: localize Accesses Disk CacheDisable

Average Time Write: 232msec Average Time Read231 msec

* **Localized Accesses with Cache Enabled**

l Test4 enabled 2

Average Time Write: 0msec Average Time Read117 msec

l Test4 enabled 2

Average Time Write: 0msec Average Time Read117 msec

l Test3 enabled 2

Average Time Write: 0msec Average Time Read117 msec

* **Mixed Accesses with Cache Disabled**

l Test4 disabled 3

Category test: Mixed Accesses Disk CacheDisable

Average Time Write: 28msec Average Time Read28 msec

l Test4 disabled 3

threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=0)

Category test: Mixed Accesses Disk CacheDisable

Average Time Write: 27msec Average Time Read27 msec

l Tet4 disabled 3

Category test: Mixed Accesses Disk CacheDisable

Average Time Write: 29msec Average Time Read29 msec

* **Mixed Accesses with Cache Enabled**

l Test4 enabled 3

Category test: Mixed Accesses Disk CacheEnable

Average Time Write: 13msec Average Time Read19 msec

l Test4 enabled 3

Category test: Mixed Accesses Disk CacheEnable

Average Time Write: 8msec Average Time Read13 msec

l Test4 enabled 3

Category test: Mixed Accesses Disk CacheEnable

Average Time Write: 10msec Average Time Read15 msec

* **Adversary Accesses with Cache Disabled**

l Test4 disabled 4

Category test: Adversary Accesses Disk CacheDisable

Average Time Write: 40msec Average Time Read40 msec

l Test4 disabled 4

Category test: Adversary Accesses Disk CacheDisable

Average Time Write: 40msec Average Time Read40 msec

l Test4 disabled 4

Category test: Adversary Accesses Disk CacheDisable

Average Time Write: 40msec Average Time Read40 msec

* **Adversary Accesses with Cache Enabled**

l Test4 enabled 4

Category test: Adversary Accesses Disk CacheEnable

Average Time Write: 28msec Average Time Read42 msec

l Test4 enabled 4

Category test: Adversary Accesses Disk CacheEnable

Average Time Write: 28msec Average Time Read42 msec

l Test4 enabled 4

Category test: Adversary Accesses Disk CacheEnable

Average Time Write: 28msec Average Time Read42 msec

**3. Report:**

**Specification/design explanation on Cache.java**

Cache class implements a buffer cache that stores frequently accessed disk blocks in memory for the ThreadOS using the enhanced second chance algorithm for page victim selection and replacement.

The cache is structured as an array of entries as private defined as a sub-class named Entry and it tracks the reference and dirty bits, and the actual byte data for the cache entry.

**Entry Subclass Constructor**

Takes an integer to define the size of the page and initializes the following variables:

A byte array, cacheBlockData, an integer blockFrameNumber and Boolean variables for referenceBit and dirtyBit.

**Cache Default Constructor**Takes two integer values, the block size and number of blocks, to initialize a new cache  
Creates and initializes a new page table object of the given size, store the size in internal variable  
Initialize the size of a frame, next victim and it will start out at end (beginning) of table.

**findFreePage**

It’s a private helper function to find the next free page if there is any by using helper method findPage.

By looping through the page table and checking if the next block is empty. If it’s found, its value is returned but if it’s not found, it returns -1.

**findPage**

It’s a private helper function to find a given page by the blockFrameNumber. if the page is found, it return its location in the page table. If not, loops through the cache updating the status of a cache block and get passed a cache block ID, the frame number, and what to set the reference bit to update the frame number and   
sets the reference bit to the passed Boolean value.

**nextVictim**

It’s a private helper function to find the next victim free page and it is used if it did not find any free pages.  
Implements the enhanced second change algorithm. It starts at the beginning of the cache and looks for an entry. Moves the victim entry to the next position. It loops through beginning at end of table and checks if the reference bit is toggled. If it’s not set, this will be the next victim. Otherwise, toggle it to 0 and check the next.

Then writeBack method gets called which checks the victim’s dirty bit and decides if it needs to be written to disk or not. If the dirty bit is 1, we write to disk, if not, its free to replace it.

**writeBack**

it’s a private helper function to write data from a cache block to disk. Checks if the cache entry's dirty bit is set, and it is not empty. If it meets these criteria, uses syslib.rawwrite to write it and then checks if entry is dirty, and not empty. Then writes it to disk, and updates the dirty bit.

**read**

it’s a public cache method that enables reading a given block ID into a buffer, but first it validates the block ID, and then see if its in the cache to read. If it’s in the cache, we see if we have any empty cache blocks to read it to. If not that, then we use the enhanced second change algorithm to pick a victim and then read. It also makes sure it has a valid block ID (non-negative) and check if its already in the cache. if it is (by being valid) read it and return true, and next looks for an empty block to read to read from disk to cache and then reads to buffer from cache. Otherwise, must find a victim, replace, and then read.

**write**

It’s a public cache method that enables writing a buffer to a given block ID. First it makes sure the block ID is valid, and then it finds where it can write. If it’s already in the cache, updates it. If it isn’t, it finds if there is any empty blocks and writes to them. If not, then uses the enhanced second chance algorithm and it finds a victim to replace. In the meantime, it makes sure that has a valid block ID (non-negative) and checks if it’s already in the cache, it copies the buffer to that cache block and update the next try to find an empty block. Otherwise, it must find a victim to save, and then write the data.

**sync**

It’s a public cache method that syncs the cache. It goes through each entry and writes to disk, if it’s dirty,  
loops through the page table and writes to disk if we need to.

**flush**

It’s a public cache method that flush the cache. It goes through each entry and write to disk if needed and then clear it. Then it loops through the cache and writes to disk if necessary to update that block to reflect it is empty, clear reference bit and then finally syncs.

**copyIntoCacheBlock**

it’s a private helper function to read data from the buffer into the selected cache block. Uses arraycopy to copy the data, and then it make sure the dirty bit and reference bits are set appropriately. It copies from buffer to cache and sets the dirty bit. Then updates the cache entry to the block we reference and that it has been referenced.

**readCacheBlock**

It’s a private helper function to read data from a cache block to the buffer. It also updates the reference bit to be set copy from cache to buffer as well as the cache entry to the block we've referenced and that it has been referenced.

**Specification/design explanation on Test4.java**

A user level testing program that conducts disk validity tests and measures the performance  
Performs four tests: Random Accesses, Localized Accesses, Mixed Accesses and Adversary Accesses.

Test4 receives 2 command line arguments and performs a different test according to a combination of those arguments.

The first argument is enabled/disable, which directs a read/write test with caching ability turned on or not test by using SysLib.rawread, SysLib.rawwrite and SysLib.sync system calls. if the argument specifies disable meaning no disk cache. Otherwise, enable specifies using SysLib.cread, SysLib.cwrite, and SysLib.csync sytem calls. The second argument directs one of the four performances tests represented with numbers 1, 2, 3 or 4 and each test includes a read/write time in miliseconds.

**Performance Consideration Comparisons Results For All Category Tests**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Category  Test | Not Using  Disk Cache  (Disabled)  WAT(3) | Not Using  Disk Cache  (Disabled)  RAT(3) | Using  Disk Cache  (Enabled)  WAT(3) | Using  Disk Cache  (Enabled)  RAT(3) | Difference  msec  WAT(3) | Difference  in  msec  RAT(3) |
| Random Accesses | 40 msec | 40 msec | 38 msec | 42 msec | 2 msec | -2 msec |
| Localized Accesses | 233 msec | 233 msec | 0 msec | 117 msec | 233 msec | 116 msec |
| Mixed Accesses | 28 msec | 28 msec | 10.33 msec | 15.66 msec | 17.67 msec | 12.34 msec |
| Adversary Accesses | 40 msec | 40 msec | 28 msec | 34 msec | 12 msec | 6 msec |

* **Performance Consideration on Random Accesses**

Enabling Disk Cache in Random Accesses performed a bit faster than disabling Disk Cache in both Read and Write, but there is not that much difference between the speeds. In 200 reads and 200 writes events random accesses chances of hitting a cache or not did not impact much on performance.

* **Performance Consideration on Localized Accesses**

Enabling Disk Cache in Localized Accesses performed more efficiently with much faster speed than disabling Disk Cache in both Read and Write: Testing Enable Cache outputs for write/read results of 0 msec/117 msec. Values in memory seemed as if they never went to disk at all.

* **Performance Consideration on Mixed Accesses**

Enabling Disk Cache in Mixed Accesses performed a lot faster than disabling Disk Cache in both Read and Write: Difference Read and Write Average time were 17.67 msec/ 12.34 msec. The mixed test performed simulates that there will be a cache hit of 90% and a miss of 10% of the time. As a result shows, when the cache is enabled average write/read time of 10.33/15.66 msec compared to 28/28 msec. Showing similar improvements for both reading and writing because it is missing 10% of hitting and 90% better hitting the cache of the time causing that same percentage as overhead.

* **Performance Consideration on Adversary Accesses**

Enabling Disk Cache in Adversary Accesses performed faster than disabling Disk Cache outputting and average difference for write/read 12 msec./6 msec . In this test case, adversary test accesses generates an adversary array blocks in groups by 100, so modulo 11 then it multiplies by 90 to get units that are separated by 100 block units apart, but within bounds still. However in some previous test running cases there was not much difference which probably tells me that either this adversary test needs to be modified to create more adversary accesses but in other was adversary enough.