Pouria Ghadimi
CSS430
Professor Dimpsey

Program 4: Paging

Report

Part 1's Algorithm and design:

My Cache is following the enhanced Second Chance Algorithm (eSCA). eSCA works as follows:

(0,0) Victim page to remove and replace
(0,1) Not the ideal victim, we have to write back buffer then it's a good victim
(1,0) Set it to (0,0): give it a second chance
(1,1) Set it to (0,1)

To implement this I created a small weight class called cachBlock which includes 4 elements: frameNumber, refBit, dirtyBit and Buffer. frameNumber is the address for the cacheBlock refBit and dirtyBit are to indicate last read and write in the cache block and buffer is the placeholder for the data.

First important execution inside this class is the initialization of the page table and filling it with initialized cacheBlocks. Then in order to make my read and write methods more efficient I created findVictim() and findFreePage() functions which respectively search for next victim to be removed and free page in the pageTable to fill with data coming from disk. Victim function has a search that uses a clock iterator that moves and stores the last position of the clock and free page function just searches through the page table to find a page with frame number equals to negative one (-1).

Inside the read and write functions first of all I checked for a hit so I had to iterate through the entire page table if it wasn't hit then I looked for a free page and if there is no free page at all, it is time to find a victim. If the victim is dirty write back all the data into disk and continue the operation.

For last two functions, sync and flush, all I had to do was to use for loops, write back and set all the necessary data to initial values.

Performance consideration on random accesses

After running the random accesses test on the created cache file, as it is visible in the results having the caching enabled is slightly better than no caching. This small different in the result is because of number of the operations that uses the cache the difference will be more if the amount of data changes.

```
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
shell[1]% Test4 enabled 1
Test4
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=1)
Test random accesses (cache enabled): 14637
shell[2]% Test4 disabled 1
Test4
threadOS: a new thread (thread=Thread[Thread-9,2,main] tid=3 pid=1)
```

Performance consideration on localized accesses

After localized accesses test because of the size of data difference between duration of cache enabled and disabled is more. it’s about 2 to 3 seconds. Which shows that cache enable algorithm works faster.

```
shell[3]% Test4 enabled 2
Test4
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=1)
Test localized accesses (cache enabled): 83837
shell[4]% Test4 disabled 2
Test4
threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=1)
Test localized accesses (cache disabled): 81700
```

Performance consideration on mixed accesses

Since in the mixed accesses has only 10 percent of random accesses after running the test, we can say that the cache enabled algorithm is way faster than disabled cache.

```
shell[1]% Test4 enabled 3
Test4
threadOS: a new thread (thread=Thread[Thread-11,2,main] tid=4 pid=3)
Test mixed accesses (cache enabled): 4498
shell[2]% Test4 disabled 3
Test4
threadOS: a new thread (thread=Thread[Thread-13,2,main] tid=5 pid=3)
Test mixed accesses (cache disabled): 9900
```

Performance consideration on adversary accesses

results in this test segment shows that the cache enabled has worse performance which is actually accurate because adversary test is not making good use of cache at all.

```
-->l Test4 enabled 4
l Test4 enabled 4
threadOS: a new thread (thread=Thread[Thread-5,2,main] tid=1 pid=0)
Test adversary accesses (cache enabled): 8486
-->l Test4 disabled 4
l Test4 disabled 4
threadOS: a new thread (thread=Thread[Thread-7,2,main] tid=2 pid=0)
Test adversary accesses (cache disabled): 8151
```