Project 4

Margaret Connor

Spring 2019

Project

This project implements a page replacement mechanism and caching to improve performance. The assignment we are required to implement a buffer cache that stores frequently-accessed disk blocks in memory. During page replacement, we use the second-chance algorithm to maximize performance.

Cache Class Description

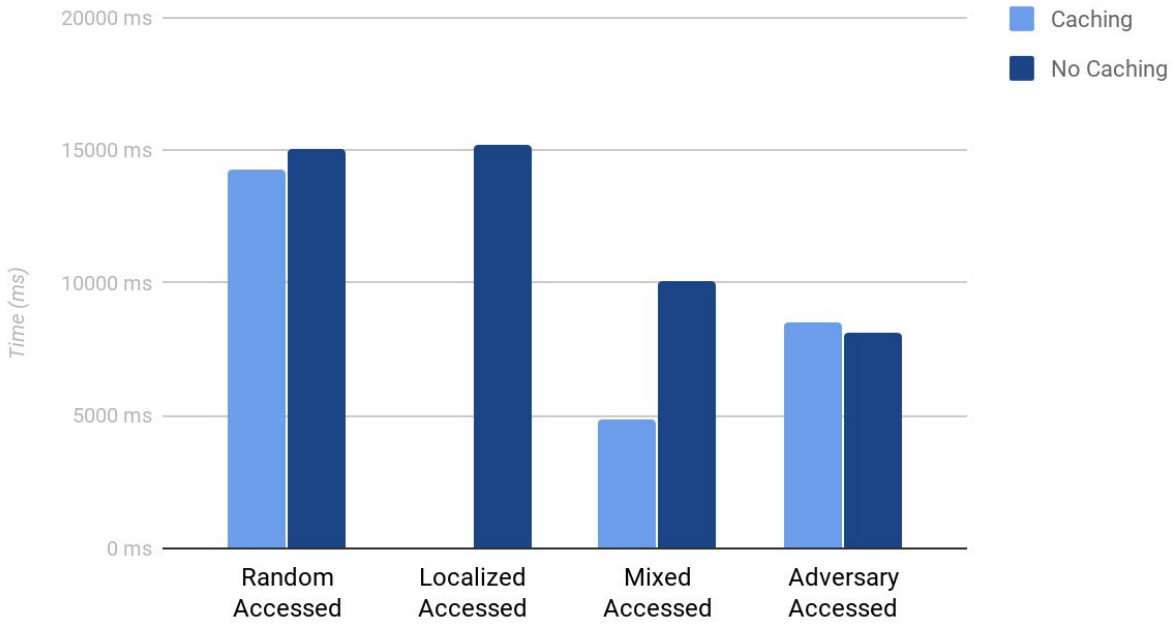
My Cache java class consists of an array of blocks and a queue of integers that acts like a clock. Each block is made up of an identification number, a byte array, a reference bit, and a dirty bit. The array of blocks and block sizes are initialized upon instantiation of the Cache object. The Cache can be read from, written to, flushed, and synched with the DISK. The block class provides assistance by allowing single blocks to be read from DISK, written to DISK, written from kernel to cache, and flushed.

When the Cache class encounters a page replacement scenario it will use the second-chance algorithm to pick a victim. The second-chance algorithm will choose a page that has not recently been referenced or (if all pages have been recently referenced) the page that has been in the cache for the longest time. This means that it acts as a FIFO second-chance cache.

Results Cache vs No Cache

In almost all instances using Caching improved the performance time of different types of memory access. The general reason why caching helps improve the performance of memory access is because it allows a small amount of memory to be stored in a fast access area. Using the second-chance algorithm allows us to keep a memory that has been recently used in the fast access area. In Figure 1-1 you can see the results of my testing.

Figure 1 - 1: Results of Test4



Random Access

This test read and write many blocks randomly across the disk. The performance for this model is only slightly better, this is likely because there is only a small chance that the same block is randomly selected twice. Even if a block is referenced multiple times it also has to be referenced close enough that the cycling does not make it a victim in between references. Still, the caching method improved the performance time.

Localized Access

Localized access had the best performance improvement across all the test. The localized access test read and write a small selection of blocks many times to get a high ratio of cache hits. This means that fewer page replacements had to occur if there were enough block in the cache to hold the small selection of blocks each block would only need to be paged in once. Localized access is the best case scenario for caching but is actually happen often in practice. User programs are likely to access the same data or the memory locations very close to the data previously accessed, this is called spatial locality.

Mixed Access

Mixed access testing makes 90% of the total disk operations localized accesses and 10% should be random accesses. This is a mix of the two previous testings and illustrates what performance looks like in more practical settings. With 90% of the disk operations being localized the process will run quickly until it hits one of the random access which is likely to cause a page replacement. The improvement is better than completely random but less than fully local.

Adversary Access

This testing is the worst case scenario for accessing memory in the DISK. Here the process never reads the same block from memory twice. Meaning that page replacement occurs at every step. In this case, the cache performed slower, likely because of the fact that pulling blocks to cache then giving it to the process has a slight overhead. It adds an extra step and performs as if caching was not implemented. But because of spatial locality, this is very unlikely from occurring.

Table 1 - 1: Results of Test4		
	Cache Enabled	Cache Disabled
Random Accessed	14274 ms	15091 ms
Localized Accessed	1 ms	15206 ms
Mixed Accessed	4823 ms	10101 ms
Adversary Accessed	8517 ms	8127 ms