Program 4

Sunday, November 22, 2015 2:25 PM

Virtual Memory and Paging

1. Purpose

This assignment focuses on page replacement mechanisms and the performance improvements achieved by implementing a buffer cache that stores frequently-accessed disk blocks in memory.

In this assignment you will implement the ***enhanced second-chance*** algorithm as described in the OS textbook - Chapter 9.4. You will then measure its performance when running various test cases, and consider the merits and demerits of the implementation.

2. Introduction to Disk Caching

Caching data from slower external main memory to the faster internal CPU memory takes advantage of both spatial and temporal ***locality*** of data reference. 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 and is a key reason why an operating system reads a block or page of data from the disk rather than reading just the few bytes of data the program has accessed. User programs also tend to access the same data again in a very short period of time which in turn means that the least-recently used blocks or pages are unlikely to be used and could be *victims* for page replacement.

To accelerate disk access in *ThreadOS*, you will implement a cache that stores frequently-accessed disk blocks into main memory. Therefore, subsequent access to the same disk block can quickly read the data cached in the memory. However, when the disk cache is full and another block needs to be cached, the OS must select a victim to replace. You will employ the *enhanced second-chance algorithm* to choose the block to replace. If the block to be replaced has been updated while in the cache it must be written back to disk; otherwise, it can just be overwritten with the new block.

The following instructions summarize how to read, write, cache and replace a block of data:

1. Block-read algorithm
Scan all the cache entries (sequential search is good enough for this assignment). If the corresponding entry has been found in the cache, read the contents from the cache. Otherwise, find a free block entry in the cache and read the data from the disk to this cache block. If the cache is full and there is not a free cache block, find a victim using the *enhanced second-chance algorithm*. If this victim is dirty, you must first write back its contents to the disk and thereafter read new data from the disk into this cache block. Don't forget to set the reference bit.

2. Block-write algorithm
Scan the cache for the appropriate block (again, sequential search is adequate). If the corresponding entry has been found in the cache, write new data to this cache block. Otherwise, find a free block entry in the cache and write the data to this cache block. You do not have to write this data through to the disk device. Mark its cache entry as dirty. If you cannot find any free cache block, then find a victim using the *enhanced second-chance algorithm*. If this victim is dirty, you must first write back its contents to the disk and thereafter write new data into this cache block. Don't forget to set the reference bit.

3. Block-replacement algorithm
Follow the *enhanced second-chance algorithm* described in the textbook pages 415-416.

Code availability

Option 1 Linux Lab: (Preferred option)

Get a fresh copy of ThreadOS from the normal location. The complete ThreadOS source code and .class files can be found in the UW1-320 Linux machines
`/usr/apps/CSS430/ThreadOS/`

Use this *kernel_org.java* and rename it as *Kernel.java*:

Kernel_org

Option 2 Right here (not recommended, only in emergency)

In case you have issues accessing the lab, you can use the following, but it is highly recommended you use the files from the linux lab instead:

For those having issues accessing that folder, there you can download this file. **Use *kernel_org.java* and rename it as *Kernel.java*.**	P4

3. Statement of Work

Part 1: Implementation

Requirements

Design a disk cache based on the enhanced second-chance algorithm and implement it in *Cache.java*. A skeleton file of Cache.java is provided for you.

The specification for the disk cache is:

methods	**descriptions**
`Cache( int blockSize, int cacheBlocks )`	The constructor allocates a *cacheBlocks* number of cache blocks in memory. Each cache Block should contain a *blockSize*-byte of data.
`boolean read( int blockId, byte buffer[ ] )`	1. Scans the page table for `blockId` to determine whether the data is in memory a. If it *is* in memory: reads into the *buffer[ ]* the contents of the cache block specified by `blockId` b. If it is *not* in memory: reads the corresponding disk block from the ThreadOS disk (Disk.java) and load it into the main memory and add it to the page table (i.e., is in cache). 2. Return true if no errors occur (See below). Note: The ThreadOS disk contains 1000 blocks. You should only return false in the case of an out of bounds blockId.
`boolean write( int blockId, byte buffer[ ] )`	1. Writes the contents of *buffer[ ]* array to the cache block specified by *blockId* a. If the block specified by `blockId` is *not* in cache, find a free cache block and writes the *buffer [ ]* contents to it. b. If the block is *not* in cache *and* there are no free blocks, you'll have to find a victim for replacement. (Note that these are the relevant options for write). 2. Return true unless there is an error (Specified below). Method does not read from or write to ThreadOS disk. Note: The ThreadOS disk contains 1000 blocks. You should only return false in the case of an out of bounds `blockId`.
`void sync( )`	1. Writes back all dirty blocks to *Disk.java* 2. Forces *Disk.java* to write back all contents to the *DISK* file. Still maintains clean block copies in *Cache.java*. Must be called when shutting down *ThreadOS*.
`void flush()`	1. Writes back all dirty blocks to *Disk.java* 2. Forces *Disk.java* to write back all contents to the *DISK* file. 3. Wipes all cached blocks. Should be called when you choose to run a different test case that doesn't include the cached data from the previous test.

Kernel_org.java uses your *Cache.java* in its *interrupt( )* method, so that you don't have to modify the kernel.

1. **BOOT:** instantiates a *Cache.java* object, (say *cache*) with 10 cache blocks.
2. **CREAD:** is called from the *SysLib.cread( )* library call. It invokes *cache.read( )*.

3. **CWRITE:** is called from the *SysLib.cwrite( )* library call. It invokes *cache.write( )*.
4. **CSYNC:** is called from the *SysLib.csync( )* library call. It invokes *cache.sync( )*.
5. **CFLUSH:** is called from the *SysLib.flush( )* library call. It invokes *cache.flush( )*.

NOTE: To read from the Disk you can use SysLib.rawread(), and to write use SysLib.rawwrite() (hint: look at Kernel.java)

Each cache block must have the following entry items:

Entry items	descriptions
block frame number	• Contains the disk block number of cached data. • if this entry does not have valid block information, the disk block number should be set to -1 • Recall, the disk in ThreadOS contains *1000* blocks grouped into 10 tracks.
reference bit	• Is set to 1 or true whenever this block is accessed. • Reset it to 0 or false by the second-chance algorithm when searching a next victim.
dirty bit	• Is set to 1 or true whenever this block is written. • Reset it to 0 or false when this block is written back to the disk.

***Note, ThreadOS' disk contains 1000 blocks. Assume there is no out of bounds block accesses, i.e., take into account the aforementioned number of blocks.**

Part 2: Performance Test

Write a user-level test program called *Test4.java* that conducts disk validity tests and measures the performance. *Test4.java* should perform the following four tests:

1. **Random accesses:**
   read and write many blocks randomly across the disk. Verify the correctness of your disk cache.
2. **Localized accesses:**
   read and write a small selection of blocks many times to get a high ratio of cache hits.
3. **Mixed accesses:**
   90% of the total disk operations should be localized accesses and 10% should be random accesses.
4. **Adversary accesses:**
   generate disk accesses that do not make good use of the disk cache at all, i.e., purposely accessing blocks to create cache misses.

Test4.java should first call *flush* to clear the cache before executing. In this way the cache will not have valid blocks from previous applications or tests. It should receive the following two arguments and perform a different test according to a combination of those arguments.

1. **The 1st argument:** directs that a test will use the disk cache or not. *Test4.java* uses `SysLib.rawread`, `SysLib.rawwrite`, and `SysLib.sync` system calls if the argument specifies **disabled**, i.e., **don't** use disk cache. Otherwise, if the argument specifies **enabled**, it should use instead `SysLib.cread`, `SysLib.cwrite`, and `SysLib.csync` system calls.
2. **The 2nd argument:** directs one of the above four performance tests.

Example:

```
$ java Boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test4 [enabled|disabled] [1-4]
```

For each test, your *Test4.java* should display the average read time and average write time. Compare the performance and consider the results when you run each of the above four test cases with and without the disk cache.

4. What to Turn In

Submit the following code (according to Canvas assignment):

1. Part 1:

   - Cache.java
2. Part 2:
   - Test4.java
   - Performance results that must test:
     - random accesses with cache disabled
     - random accesses with cache enabled
     - localized accesses with cache disabled
     - localized accesses with cache enabled
     - mixed accesses with cache disabled
     - mixed accesses with cache enabled
     - adversary accesses with cache disabled
     - adversary accesses with cache enabled
3. Report:
   - Specification/design explanation on Cache.java
   - Performance consideration on random accesses
   - Performance consideration on localized accesses
   - Performance consideration on mixed accesses
   - Performance consideration on adversary accesses

Part 1 and Part 2 should be zipped into a ZIP file.   Part 3 should be submitted separately (outside the ZIP).

5. FAQ

This website could answer your questions: click here before emailing the professor :-)