Assignment 2: Unix v6 File Systems, Take II

Kudos to Mendel Rosenblum for this assignment and the vast majority of the handout.

The purpose of this assignment is to give you hands-on experience with performance optimization of an existing system—in this case, the read-only file system you created for Assignment 1. To ensure that you're familar with the file system design and implementation, you're to revisit that system and a second user program implemented in terms of it with the intent of making the file system itself and the user program as fast as possible. By doing so, you'll be exposed to the types of optimizations that systems programmers are constantly working to find in order to make systems and software that layers on top of its services as fast as possible.

The assignment will also introduce you to some of the performance tools and techniques that are typical of what most modern operating systems support. You will perform an iterative approach to performance optimization where you study a working but poorly performing system, decide what the promising performance optimizations are, implement then and measure the performance gains, and repeat as necessary.

In some cases, you may find an algorithm can be replaced with an asymptotically tighter one. In other cases, you might employ caching and/or memoization to minimize the number of high-latency disk reads. The end goal is to minimize CPU user time, CPU system time, and the number of I/O disk operations. You'll reach that goal by leveraging your understanding of caching and optimization techniques from CS107 and your understanding of the file system implementation you worked with for Assignment 1. This is also an opportunity to work with code you've written in the past and deal with any bad decisions that didn't matter then but matter now.

Due Date: Wednesday, October 19th, 2016 at 11:59 p.m.

Getting started

All coding should be done on the `myth` cluster machines, as that's where we'll be testing all `assign2` submissions. You should clone the master mercurial repository we've set up for you by typing:

```
myth22> hg clone /usr/class/cs110/repos/assign2/$USER assign2
```

Doing so will create an `assign2` directory within your own file space, and you can descend into that `assign2` directory and code there.

There will be an placeholder `assign1` subdirectory inside your `assign2` directory. Copy the .h and .c files contributing to your solution to Assignment 1 into the `assign1` subdirectory, overwriting the existing files in place so that everything compiles. Make sure you only copy the `.h` and `.c` files (e.g. `cp ../../assign1/*.[hc]`), since you'll otherwise copy some hidden files that mercurial uses and get error messages about nested repos when you try to locally commit changes using `hg commit`.

If you didn't get Assignment 1 working, you have to do that before you can start on Assignment 2. (We're planning on turning around feedback on Assignment 1 by this coming Monday, but you should already know if your Assignment 1 submission is broken; we're happy to spend some time in office hours helping you get your Assignment 1 submission working if you need us).

Building

`cd` to code directory and type `make <makeopt>` to build, where `<makeopt>` is one of: `debug`, `valgrind`, `gprof`, `perf`, or `opt` (if you omit the `<makeopt>`, then it defaults to `opt`). Since this assignment is about performance timing, evaluation runs should be done with `opt` builds, which enables aggressive compiler optimizations. Since compiler optimizations can interfere with some of the debugging and measurement tools, we provide other build options that work well with the tools. Building with `make debug`, for instance, should be used when debugging with `gdb`.

We also provide builds customized for the `valgrind`, `perf`, and `gprof` tools. Note that when you want to build a different target, you must type `make clean` first to remove the previous build. For example to switch from a `debug` build to an `opt` build type the command:

```
myth22> make clean && make opt
```

Testing

Type:

```
myth22> ./disksearch <options> diskimagePath
```

One example:

```
myth22> ln -s /usr/class/cs110/samples/assign2/testdisks/simple.dat simple.dat
myth22> ln -s /usr/class/cs110/samples/assign2/testdisks/simple.img simple.img
myth22> ./disksearch -f simple.dat simple.img
```

This will query `simple.img` for the words in `simple.dat`.

`<options>` can be:

- `-c <n>` to set the cache size in KB
- `-l <n>` to set the disk latency (default is 8000s)
- `-w <word>` to specify a query word on the command line
- `-f` <**file**> to specify a file containing query words

`-d <flags>` generates debug output

We provide a diagnostic test script that will run the `disksearch` program on the several disk images to make sure your Assignment 1 code is in working order. To run the test script, type:

```
myth22> ./diagnostic_test.sh
```

If this script reports a failure, you will need to fix your Assignment 1 bugs before you continue. If problems persist, any one of the CS110 TAs or I can help you get your code working. You should continue to run `diagnostic_test.sh` periodically as you implant various optimizations. This'll ensure those optimizations haven't broken existing functionality.

The Existing Disk Image Search Implementation

The software we give you as part of the assignment builds on top of your Assignment 1 submission to implement a **keyword search**. The code works by walking the file system tree to discover all the files on disk, scanning the files to find words, and entering said words into a hash-table-backed index. For each word, we store the absolute pathnames of the files containing the word and the offsets where the word occurs. Once the index is built, the code looks up a series of words and lists all of the files that contain them. (Words are considered to be strings of upper or lower case characters of length 64 or less).

The test harness we provide you takes as input a Version 6 Unix disk image (as with Assignment 1) and a file of words to look up. The existing software keeps track of many counters and timings of things that you can use.

Assignment Details

The code we give you is fully functional and acts as a specification of what the system should do. Unless otherwise specified, the code with your modifications should have the same output as the original code we provide for all inputs (and not just for the tests and the disk images we provide). The starter code is slow enough that you will not want to wait for it to finish the longer tests. As you speed up your implementation you will be able to run more of the tests. Fully optimized, the largest test will run for several minutes.

Big note: you are free to change any file in your `assign2` repo, including code you wrote for Assignment 1 (in the `assign1` sub-repo). The only exception: you may not make any changes to `disksim.c`, since that's what we use to simulate timing.

This assignment is much more open-ended than the first one. As part of the assignment, you should **document** your performance optimizations in a text file (`README.txt`), and this `README.txt` file should be submitted with your assignment. This text file should take the form of a log where each entry corresponds to a particular performance optimization you tried. Each entry must contain:

- A detailed description of the optimization.

- The analysis you performed and any measurements you made to suggest the optimization might work well.
- Back-of-the-envelope calculations you did to predict how much the optimization should help. If there's no obvious back-of-the-envelope calculation, then you must say so.
- How you measured whether the optimization was effective or not.

The format implies that you need to implement optimizations serially, with performance measurements before and after each optimization. You should have at least 3 entries demonstrating performance gains. **These entries must detail the three most beneficial optimizations you implemented.**

Your solution must:

- be functionally identical to the original program—specifically, your changes should only affect program performance and should not alter semantics or program output. **This goes for all program inputs, not just the ones we provide you with. Restated, you can't change the program to be faster on just the four disk images we expose when it would certainly break functionality for other test disks.**
- not alter the program's output when run with the `-q` option, so that our automatic grading scripts continue to work. It will also allow both the course staff and yourself to see if your optimization broke something.
- not adversely affect the speed of the query processing phase. Assume that in steady state, the system will run queries for much longer than the index-building phase so doing something like getting rid of the index would not be an acceptable solution.
- not break the disk latency simulation model imposed in this assignment. In other words, you may not modify `disksim.c`.

If you decide to implement a generic cache as one of your optimizations, in addition to the above, your solution must limit the amount of memory used. If you decide to allocate memory to hold file data, the total amount of the memory must be less than that specified via the **-c** option. The memory usage of your optimizations should not blow up if you are given a very big file system, a very big file, or a large numbers of files. Note that the index itself might get big, but the memory used by your optimizations shouldn't. If you decide to allocate memory to hold file data, the total amount of the memory must be less than is specified via the **-c** option. The code we give you allocates the specified amount of memory in a single chunk (see the `cachemem` module). All disk data cached by your optimizations must use this memory chunk. Any non-disk data (e.g. metadata for your cache) need not be allocated from this chunk and may be allocated by calling `malloc` or extending existing data structures.

Exception: A small number of blocks (less than 10) of disk data per file may be stored outside the `cachemem` chunk while the file is actively being processed (e.g., while it is open). The spirit of the `cachemem` size limitation is to force you to use some form of replacement policy (as you learned in CS107 or some other systems course) if you decide to implement a cache. If you do, we're not going to go digging around counting the number of individual blocks that are saved elsewhere to see if they total slightly more than 10.

The grading will be done on one configuration: with a 1MB cache (`-c 1024`). Your `README.txt` file should report your performance for this configuration for optimizations that use a cache. For the `vlarge` disk, you should **also** report the performance with 16KB (`-c 16`), 256KB (`-c 256`), and 512KB (`-c 512`) cache sizes.

There are 5 disks provided: `simple`, `medium`, `large`, `vlarge`, and `manyfiles`. You are expected to get the system performing well on the `simple`, `medium`, `large` and `vlarge` disks. The `simple` disk is provided to help you get started, but it is a trivial test case, and `manyfiles` is in place for those who want to shoot for a very small amount of extra credit (about 10%). We are most interested in what happens on the `medium`, `large` and `vlarge` disks. You will earn points on a sliding scale as follows:

`medium`:

- 70% 24 seconds (shoot for 3,000 I/Os and a CPU time that adds up to 24 seconds for 70% of the points)
- 80% 18 seconds (2,500 I/Os)
- 90% 12 seconds (1,500 I/Os)
- 100% 6 seconds (750 I/Os)

`large`:

- 70% 120 seconds (15,000 I/Os)
- 80% 80 seconds (10,000 I/Os)
- 90% 40 seconds (5,000 I/Os)
- 100% 20 seconds (2,500 I/Os)

`vlarge`:

- 70% 8,000 seconds (1,000,000 I/Os)
- 80% 800 seconds (100,000 I/Os)
- 90% 400 seconds (50,000 I/Os)
- 100% 240 seconds (30,000 I/Os)

For example, if your assignment runs in 13 seconds on the `medium` image, you'll receive 80% of the points for that case.

Given the lack of dedicated machines for performance runs and the multi-hour runtimes of some of the disks, we expect you to run with a zero-latency disk (`-l 0`) and use the number of I/O operations and total CPU time (system time + user time) to estimate your performance. For grading purposes, we will run your code with a zero-latency-disk flag and we will use the estimate of (CPU time + 8ms * number of I/Os). The table above provides just the I/O counts we will use in computing the grading thresholds. So to be clear, if your application executes in less than 240 seconds worth of CPU and simulated disk access time on the `vlarge` image, you would received 100% of the points for that one test.

For your reference, below are the baseline I/O counts (using our own solution for Assignment 1). You may want to make sure your numbers align with or are better than ours for your standalone Assignment 1 solution (at least for `simple` and `medium` disks) before you begin implementing optimizations.

- `simple`: 5 seconds (583 I/Os)
- `medium`: 33,000 seconds (4,020,055 I/Os)
- `large`: 102,088 seconds (12,761,023 I/Os)
- `vlarge`: 190,000,000 seconds (23,590,667,468 I/Os)

What We Provide

The code provided with the assignment is located in the files `diskimg.[ch]`, `disksim.[ch]`, `disksearch.c`, `fileops.[ch]`, `index.[ch]`, `pathstore.[ch]`, and `scan.[ch]`. You are welcome to add your own files and modules if you'd like, provided you update the `Makefile` and you add any new `.h` and `.c` files to your mercurial repository. The modules we give you include:

- `fileops`: This module implements an API similar to the Unix application as seen by user applications (e.g. `open`, `close`, `read`). You can view this as the file system layer described in Section 2.5.11 of the Saltzer textbook. It implements access to the files on disk using your routines from Assignment 1.
- `index` and `pathstore`: These files implement the index that maps words to where they occur in files. The `index` module uses a hash table to store locations consisting of an absolute pathname and file offset associated with words. The `pathstore` module stores absolute pathname strings for the index. Aside from simply storing strings in memory, the module also implements a check and prevents identical pathnames from being loaded into the index.
- `scan`: This module contains the code that walks file system tree. It traverses the file system on the disk image, scanning the files it finds for words and then entering the words into the index.
- `diskimg`: We provide you with a **new** version of the `diskimg` disk access module, which introduces disk latency (see next item).
- `disksim`: This module implements a simple model of the delays of a physical disk. These delays are controlled via the `-l` flag used on the command line. (And no, you may not disable

the disk simulator code as an optimization). This is the one module you **may not change**—you may change all other `.h` and `.c` files.

- `debug`: This provides a small number of self-explanatory debugging routines.
- `cachemem`: The `cachemem` module allocates the memory that any caching-based optimizations must use. It exports two global variables:

- `cacheMemPtr` - A pointer to a chunk of memory
- `cacheMemSizeInKB` - Size of the chunk of memory in kilobytes

- `disksearch.c`: This is the main driver program that initializes everything, builds the index, and manages the queries.

Hints and Suggestions

Our own solution to the program relies on three optimizations: a general sector cache that relies on a 4-way associative cache fill and eviction policy, caching optimizations specific to the fileops module, and caching optimizations specific to the pathstore and index modules. You're free to do any optimizations your way, provided you don't change the overall system's behavior (not only for the provided disk images, but any other well-formed disk images we could substitute in). However, I recommend looking into the three optimizations that I used for our own solution.

You should start the assignment by understanding how the existing software works. Look through the code and figure out what is being executed. You will need the big-picture view of the system to select the right optimization to prioritize.

You should understand what is on the various test disks provided. Your Assignment 1 program is useful for this purpose. Some worthwhile optimizations will appear to have no effect on the running time of some disks. Moreover, an awareness of the unique properties of each disk will help you put the performance tool results in context, and suggest different optimizations.

Look for big improvements at first. You already understand a lot of the code, since much of it is your file system from Assignment 1. You just need to figure out how they are being called. *Note that getting rid of most of the calls to a function will give you a bigger win than simply optimizing a function. Don't waste time optimizing a function, only to end up removing the calls to it.*

The real world and this assignment care about the amount of memory your optimizations use. It should be the case if we were to give you an infinitely large disk with an infinite number of files that, excluding the index structure, your solution would use a finite amount of space. Clearly the index structure along with any small amounts of data your optimizations might add to the index structures (e.g., a field in a `struct`) would use infinite space. Any other memory that scales with the disk size must be bounded by placing it in the `cachemem` module memory.

There are several tools you can use to figure out where the time is going. They include:

- The debug module has a DPRINTF macro for debugging with printfs. You can enable existing DFPRINTFs or add your own to help you understand the program flow.
- If you make valgrind, you can use valgrind and callgrind to see where execution time is being spent. valgrind and callgrind were covered in extensively CS107.
- If you make perf, you can use perf to see where time is being spent. In the past, students have preferred to use perf over gprof, because the call graphs have been easier to generate and interpret.
- If you make gprof, you can use gprof to see where execution time is being spent. In my experience, gprof is more difficult to use than valgrind and perf, but if you already know gprof then you might benefit from using it as well.
- The debug module also exports the Debug_GetTimeInMicrosecs function that allows you to time things. For example:

```
int64_t startTime = Debug_GetTimeInMicrosecs();

// some block of code you are interested in

int64_t endTime = Debug_GetTimeInMicrosecs();

printf("Time is %lld microsecs\n", endTime - startTime);
```

Evaluation

We will grade your assignment based on the test results, code quality, and your design document (which should be typed up over the course of your work and placed in README.txt).

Grading:

- Code test results: 40 points (10 points for medium, 10 points for large, 20 points for vlarge)
- Code quality: several metrics, graded on the bucket system (solid, minor-problems, major-problems, etc).
- Design document: 30 points

We will not be testing this particular assignment against valgrind.

Submitting your work

Once you're done, you should hg commit all of your work as you normally would and then run the famous submission script by typing in /usr/class/cs110/tools/submit. There's also a sanity-check for this assignment, but sanitycheck only passes if you meet or exceed the 100% benchmark for all three diskimages. If you don't pass sanitycheck, you can still submit your work, but you'll need to circumvent the sanitycheck to do so.

Running perf

To run **perf**, first recompile your code with the appropriate flags:

```
myth22> make clean && make perf
```

Now run your program within **perf** by typing in the following

```
myth22> perf record -g ./disksearch -b simple.img
```

The **-g** flag instructs **perf** to collect call graph information about where the processor is spending its time.

Once the call graph has been collected, you can view it by typing in:

```
myth22> perf report --stdio
```

Provided everything works properly, **perf report** will present an ASCII version of the call graph that you can scroll through to find out what functions are calling each other and what percentage of time is spent where. Here's an example of what **perf report --stdio** outputs:

```
# Events:  3K cycles
#
# Overhead    Command    Shared Object                    Symbol
# ........  ..........  .................  ........................
#
    92.82% disksearch [vdso]            [.] 0x7ffff8125794
           |
           --- 0x7ffff812570c
               0x7ffff81257e5
               0x7ffff8125a1b
               Debug_GetTimeInMicrosecs
               SimulateDiskLatency
               disksim_readsector
               |
               |--50.92%-- inode_iget
               |           |
               |           |--41.52%-- Fileops_getchar
               |           |           |
               |            |           |--84.08%-- Fileops_read
               |            |           |          Scan_TreeAndIndex
               |            |           |          main
               |            |           | __libc_start_main
               |            |           |
               |            |           --15.92%-- Scan_File
               |            |                      Scan_TreeAndIndex
               |            |                      Scan_TreeAndIndex
               |            |                      main
               |            |                      __libc_start_main
               |            |
               |            |--41.47%-- file_getblock
               |            |           |
               |            |           |--92.57%-- Fileops_getchar
               |            |           |           |
```

How does one read this? You can see that 50.92% of the time spent in `disksim_readsector` comes as a result of calls to `inode_iget`, 41.52% of the time spent in those `inode_iget` calls comes as a result of calls to `Fileops_getchar`, and 84.08% of the time spent in those `Fileops_getchar` calls comes as a result of calls to `Fileops_read` (called from `Scan_TreeAndIndex`, called from `main`) and the remaining 15.92% of the time spent in those `Fileops_getchar` calls comes via calls to `Scan_File` (also called from `TreeAndIndex`, recursively called from itself, called from `main`).

Some students prefer to walk through the inverted call graph by passing the `-G` flag to `perf report`, as with this:

```
myth22> perf report --stdio -G
```

Here's an example of what the inverted call graph looks like:

```
    # Events:  3K cycles
    #
    # Overhead    Command     Shared Object                   Symbol
    # ........  ..........  .................  ........................
    #
        92.82% disksearch [vdso]            [.] 0x7ffff8125794
               |
               -- __libc_start_main
                  main
                  |
                  |--99.59%-- Scan_TreeAndIndex
                  |           |
                  |           |--52.11%-- Fileops_read
                  |           |           Fileops_getchar
                  |           |           |
                  |           |           |--65.96%-- file_getblock
                  |           |           |           |
                  |           |           |           |--50.27%-- inode_iget
```

99.59% of the time spent in `main` is actually spent within `Scan_TreeAndIndex`. Of the time spent within `Scan_TreeAndIndex`, 52.11% is spent within `Fileops_read`. Of the time spent in `Fileops_read`, 100% is spent within `Fileops_getchar`, and so forth.