Oversubscribing inotify on Embedded Platforms

By Donald Percivalle (CPE) and Scott Vanderlind (CSC)

Senior Project

California Polytechnic State University San Luis Obispo

Dr. Zachary Peterson

June 11, 2015

Abstract

For most computers running the popular Linux operating system, the integrated kernel component inotify provides adequate functionality for monitoring changes to files present on the filesystem. However, for certain embedded platforms where resources are very limited and filesystems are very populated (like network attached storage (NAS) devices), inotify may not have enough resources to provide watchers for every file. This results in applications missing change notifications for files they have watched. This paper explores methods for using inotify most effectively on embedded systems by leveraging more latent storage. Benefits of this include a reduction in dropped notifications in favor of an introduced delay on notifications for files that are less frequently changed.

Contents

1 Introduction

Western Digital produces lines of Network Attached Storage (NAS) devices that are built around an embedded Linux computer. This embedded system acts as the brains of the operation: It interfaces with the hard disk drives, manages the filesystem, and provides all the services that you'd expect in a NAS, like AFP, SMB, NFS, etc. NAS appliances that strictly provide filesystem services are pretty cut-and-dry, however, Western Digital also supports a suite of mobile and desktop applications to interface with these NAS devices that require additional services to be provided, like DLNA and iTunes library sharing and media "cloud" integration that makes media available through an iOS or Android app.

Targeted towards both consumers and businesses, Western Digital sells My Cloud devices with storage sizes ranging from 2TBs to 24TBs. It is intended to provide a seamless NAS experience where the end user can't tell if they are interfacing with local storage on their client machine, or on their My Cloud which could be across the Internet. Along with seamless reads and writes, Western Digital aims to provide an always-on media server environment for My Cloud customers. If a customer uploads a movie or an album of photos to their My Cloud, the goal is to have the iOS and Android apps, as well as any DLNA or media servers running locally on the My Cloud itself be notified of the new file as soon as possible. This goal requires a system to monitor the entire filesystem for new files.

1.1 Application

Historically, applications on Linux-like systems have tracked changes to files using a kernel API called inotify. Inotify provides applications with the means to subscribe to filesystem events at a folder level: Applications can be notified of files that have been created, modified, and deleted. This can be useful to many types of applications, especially those providing media sharing capabilities. For instance, when a media service is notified of new file placed into a watched directory, that service might examine the file for validity and make its contents available through some other abstracted API.

Western Digital's devices provide more advanced media sharing services that synchronize file state and maintain parity between multiple devices. In doing so, the services running depend heavily on knowledge about the state of the filesystem as provided by inotify.

1.2 Problem

The processes running on the embedded system in the My Cloud NAS depend on knowledge of the state of a large tree of directories. In the native implementation of inotify, a watch is created for each file or directory that needs to be monitored. When a directory is being watched, the

events that trigger a notification include a file creation, update, or delete within that directory. However, events that occur in subfolders of that watch do not trigger events. It is therefore necessary to create a watch for every folder in a tree should you want to subscribe to all file events.

The first hurdle is the hard limit of watches allowed to be allocated by each user on the system. By default this limit is set as 524,288, and therefore limits the number of files eligible for concurrent watching at 524,288. Theoretically, this limit is further lowered by redundant watches: Should two application processes desire to watch for changes in the same directory (for instance, DLNA and iTunes both watching a directory called `media`), they both may allocate (and expend) a watch for that file. In practice, this limit is easily raised by tweaking the contents of the file `/proc/sys/fs/inotify/max_user_watches`.

Let's consider the anatomy of an `inotify_watch` struct, one of which is created for each file and is registered to an `inotify_watcher` (a glorified file descriptor), the parent struct that aggregates a set of watches in the context of an application.

```
1  /* Excerpt from include/linux/inotify.h:L20 */
2  struct inotify_watch {
3      struct list_head        h_list; /* entry in inotify_handle's list */
4      struct list_head        i_list; /* entry in inode's list */
5      atomic_t                count;  /* reference count */
6      struct inotify_handle   *ih;    /* associated inotify handle */
7      struct inode            *inode; /* associated inode */
8      __s32                   wd;     /* watch descriptor */
9      __u32                   mask;   /* event mask for this watch */
10 };
```

Considering the structures that a single `inotify_watch` contains, allocating a single watch requires (on a 32-bit system) `540 bytes` of kernel memory. Unfortunately, the My Cloud EX2 device only has `500MB` of memory available to the entire system. Assuming the watch limit mentioned above has been raised, this allows for `940,740` simultaneous watches. This limit cannot be raised without adding more memory to the system, as kernel memory cannot be swapped. Worth noting is that this theoretical maximum does not give any allowances for other applications using memory.

1.3 Problem Statement

It is not feasible for multiple applications to simultaneously be running on low-power integrated systems while also monitoring sufficiently large file systems for changes (with 'large' meaning more than `940,740` folders).

2 Possible Solutions

A number of solutions have presented themselves. Let's examine them.

2.1 Modification of inotify

The most obvious solution is to modify how inotify works so that it is inherently recursive. The immediate benefit is that only a single watch would be required on the root folder of the file tree to be monitored. Unfortunately, this avenue has been explored and decided unfit for any system, let alone embedded ones:

> In many filesystems, including Unix-style ones, recursing down a directory hierarchy isn't a first class operation. [...] You list the contents of a directory and for each content that is itself a directory, you transcend into it, and repeat. [...]
>
> You could replicate this algorithm in the kernel [...] but the performance of the system would suffer. This is much too long-running an operation to perform inside the kernel, particularly if you were to do so while holding the appropriate locks. - Robert Love, inotify author[2]

2.2 Wholesale replacement of inotify

While inotify is bundled with Linux (including the distribution used on My Cloud devices), it is by no means the end-all solution to file monitoring across all platforms. Certainly commercial operating systems like Mac OS and Windows have developed respective analogs, as have other distributions of linux-like and unix-like operating systems. The barriers to porting another system from a fellow `POSIX` compliant OS are theoretically low.

However, no system exists that is sufficiently appealing. *BSD kernels (FreeBSD, OpenBSD, NetBSD, et al.) provide a module called `kqueue`, which has certain advantages over `inotify` but is similarly not recursive. A potential advangage is that kqueue can provide subscribers with notifications for all filesystem events, starting at the root directory instead of a specific folder. This is akin to drinking from a firehose while standing next to a water fountain, and leads to much application-level overhead just to filter through irrelevant events.

Other alternatives have similar enough limitations as to obviate the need to discuss them.

2.3 Development of user-space watch aggregator

The final option considered is a user-space watch aggregator and event sim-

ulator. This hybrid system depends on `inotify` for reporting file change events for as many files as possible, and uses manual scanning of file meta-data at set intervals to find the events that were missed by inotify. The biggest advantage of this approach is platform flexibility: More memory will of course allow for more watches, but less memory will not necessarily limit the number of files that can be tracked.

The biggest drawback of this approach is time sensitivity: Because a rescan is required to find changes in files not monitored by `inotify`, there is a considerable delay between the event occurring and the event being reported to the subscriber application. Still, this solution lends itself well to embedded NAS systems where RAM is expensive, disk space is cheap, and event response time is not critical.

2.4 Chosen Approach

The chosen approach is the third: Development of a user-space watch aggregator. The application produced will only serve as a proof-of-concept, but will surely provide important insight into the properties that need to be considered for future development.

2.5 Related Work

Relevant work has been done by Robert Love (inotify author) when contributing to the open-source project Beagle (a search and indexing tool) in 2004. The result of his investigations is an algorithm dubbed the Love-Trowbridge Recursive Directory Scanning Algorithm[1]. The purpose of this algorithm is to ensure that races do not occur when indexing the file system.

Love-Trowbridge specifies the order in which `inotify` setup tasks must be performed while visiting nodes in a file tree so that events occurring before a scan has completed are still handled. This generally amounts to creating event handlers for a directory after visiting it but before visiting any of its children.

Adherence to Love-Trowbridge in the developed approach may be beneficial, but not necessary, as the periodic scanning of directories will find and simulate any missed events.

3 Design

Let's investigate the design of a proof-of-concept system that implements a solution to the problem.

3.1 Constraints

When considering a design, there are several assumed constraints that are operated under. The aggregator must have:

- Low memory footprint
- Modular prioritization algorithm
- Large file tracking capacity

3.2 Implementation

To accomplish a large file tracking capacity, the aggregator can additionally collect watchers, which in turn collect watches. Ideally, the aggregator will transparently abstract away from the calling process the fact that several inotify watchers exist and will provide a unified namespace for watch descriptors. From the application's perspective, all events from all sub-directories of a given directory should be accessible from a single notification descriptor.

The aggregator will begin recursing into the directory tree and will use some algorithm to assign each directory a numerical priority. The system will exhaust all available `inotify` resources on the directories with the highest priority. When all available watches are consumed, the aggregator will collect all watches into a `fd_set` and will use that set in a call to the system function `select(2)`. The aggregator will return to the calling process a file descriptor to which all events from all the watchers will be written.

At the discretion of the calling process, the aggregator will periodically rescan and reallocate `inotify` watches utilizing the same scoring algorithm to effectively monitor the most important directories in the filesystem. As part of the rescan process, it will also detect and recreate missed events across the directory tree.

`Inotify` events report, among other things, the type of filesystem event, the filename the event occurred on, and a watch descriptor number. This watch descriptor is unique only in the context of the `inotify_watcher` instance to which the watch belongs to. Because the aggregator will register the maximum number of watchers and watches for each watcher, `inotify` events will no longer have unique watch descriptors. Watch descriptors are necessary to determine what directory an even occurred in.

The aggregator must replace the watch descriptor of all `inotify` events to a globally unique descriptor before passing the event to the calling process. Further, the aggregator must provide a mapping of the globally unique watch descriptors to directory paths in order for the calling process to resolve where an event took place in the filesystem. To accomplish this, the aggregator will record all watchers' unique descriptors (a file descriptor) and all of each watcher's watches' watch descriptors as they are registered. It will assign each unique (watcher, watch descriptor) pair a globally unique number. This number will be mapped to the directory path that that watch is registered to in a list of (global identifier, path) pairs. That list will be given to the calling process in order to do event location look ups. With these list of mappings, the aggregator will be able to translate the watcher-unique watch descriptor to the globally unique identifier of events read utilizing the `select(2)` call described above before passing the event to the calling process. Figure 1 displays this process of identifier translation and event passing.

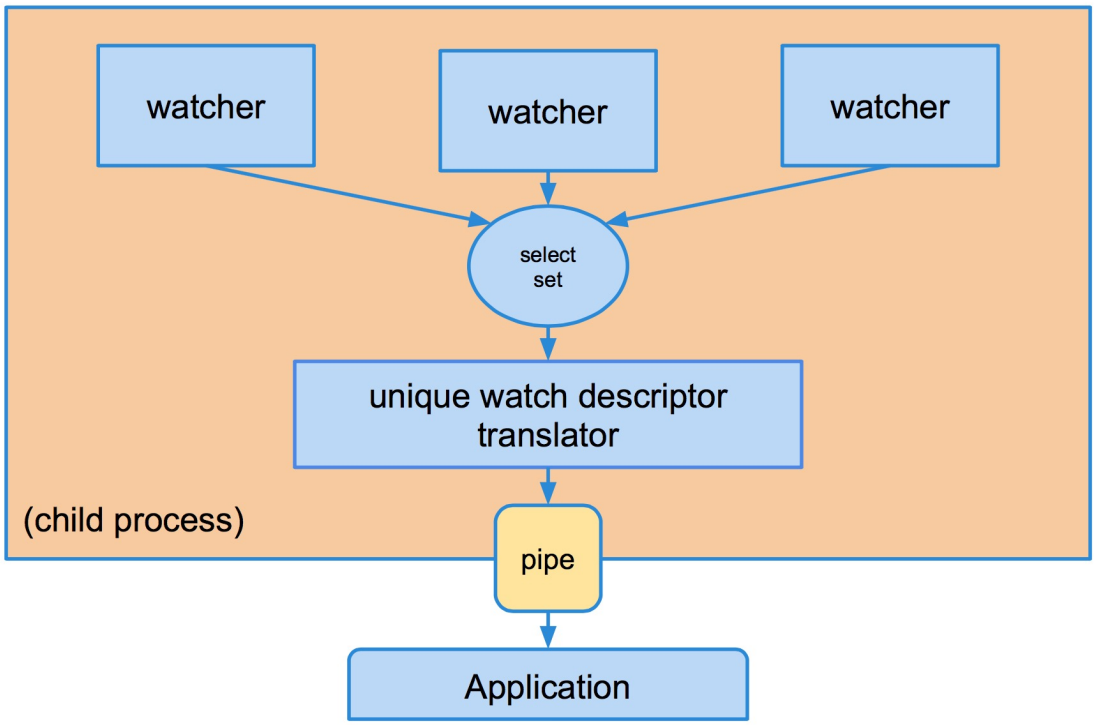

Figure 1: High level system design

4 Takeaways

4.1 Solution Viability

The described solution accomplished all of the requisite goals when implemented as a proof-of-concept. A reference application was devised to monitor a file tree, and system resources were artificially limited in order to characterize the behavior of the system. The results were favorable: Files that were watched had their events immediately forwarded to the monitor application. Files that did not make the cut had events simulated on their behalf when the next rescan occurred. The delay between the event occurring and the simulated event being created can be tuned by changing the interval of rescan, however it is important to consider that the rescan interval has a linearly inverse relationship with the amount of hard disk access operations generated by the application.

4.2 Future Work

Because the solution showed such promising results, there is much opportunity for future work. The user-land solution can only provide notifications to one client application, which is less than desirable

in systems like the My Cloud where many services require tracking the same files.

Ideally, the final solution will expose itself as an API to other applications. It will act as a middleware that allows applications to request the watches that they need without wasting any watches on files that are already watched.

A possible approach is to build the solution into a pair of kernel modules. One module would expose an API to applications wishing to subscribe to file events, and another module would provide the scoring algorithms to determine watch priority. This configuration would allow for changing scoring at any time by installing a different module.

Since this method of oversubscribing can operate with any number of watchers, it's possible to keep kernel memory usage within a reasonable amount while still enjoying the benefits of increased file monitoring capacity.

Additionally, it may be beneficial to explore incorporating functionality from the file monitoring API `fanotify`, which is also part of the Linux kernel. `fanotify` can, under certain circumstances, provide recursive monitoring of file systems. However, `fanotify` does not provide the same set of functionality as `inotify` and was therefore not considered as a wholesale replacement candidate[3]. That said, it could be incorporated in a way that would play to the strengths it does have.

5 Reference Implementation

Following is a proof-of-concept reference driver application and aggregator library. This application is intended to show how a real application would utilize the solution to oversubscribe the `inotify` system.

5.1 Driver Application

```
1  /**
2   * Test program for My Cloud inotify
3   * Recursively watches a directory for events.
4   */
5
6  #include <poll.h>
7  #include <signal.h>
8  #include <iostream>
9  #include "mycloud_inotify.hpp"
10
11 #define NUM_WATCHERS 128
12 #define NUM_WATCHES 524288
13
14 using namespace std;
15
16 void terminate(int sig){
17     cleanup_watchers();
18     exit(sig);
19 }
20
21 int main(int argc, char **argv){
22     char *buf = (char *)calloc(BUF_SIZE, sizeof(char));
23     int i, len, watcher;
24     struct pollfd fds[1];
25     struct inotify_event *event;
26     mycloud_inotify_t inotify_instance;
27
28     signal(SIGINT, terminate);
29     if (argc < 2){
30         printf("Usage: test <root dir>\n");
31         return 1;
32     }
33
34     mycloud_inotify_init(NUM_WATCHERS, NUM_WATCHES, argv[1],
35      inotify_instance);
36     watcher = inotify_instance.monitor;
37     fds[0].fd = watcher;
38     fds[0].events = POLLIN;
39     cout << "Monitor pid: " << inotify_instance.monitor_pid << '\n';
40
41     // Read from inotify fd
42     sleep(5);
43     re_crawl(NUM_WATCHERS, NUM_WATCHES, inotify_instance);
44     watcher = inotify_instance.monitor;
```

```
45      while(1){
46          i = 0;
47          memset(buf, 0, BUF_SIZE);
48
49          // Block until ready
50          poll(fds, 1, -1);
51          ioctl(watcher, FIONREAD, &len);
52          len = read(watcher, buf, len);
53          while (i < len){
54              event = (struct inotify_event*)(buf+i);
55              if (event->mask == IN_CREATE)
56                  cout << "[CREATE] in: "
57                   << inotify_instance.watch_descriptors[event->wd]
58                   << " on: " << event->name << '\n';
59              if (event->mask == IN_DELETE)
60                  cout << "[DELETE] in: "
61                   << inotify_instance.watch_descriptors[event->wd]
62                   << " on: " << event->name << '\n';
63              if (event->mask == IN_MODIFY)
64                  cout << "[MODIFY] in: "
65                   << inotify_instance.watch_descriptors[event->wd]
66                   << " on: " << event->name << '\n';
67
68              i += EVENT_SIZE + event->len;
69          }
70      }
71      return 0;
72  }
```

5.2 Watch Aggregation Module

```
1  /**
2   * mycloud_inotify.cpp
3   * My Cloud Inotify main functions.
4   * This file shall contain the functions necessary
5   * to initiazlie the My Cloud inotify to set up
6   * watchers.
7   *
8   * Donald Percivalle
9   * Scott Vanderlind
10  * Senior Project, 2015
11  */
12
13 #include <sys/inotify.h>
14 #include <sys/types.h>
15 #include <sys/stat.h>
16 #include <sys/select.h>
17 #include <sys/ioctl.h>
18 #include <limits.h>
19 #include <signal.h>
20 #include <fcntl.h>
21 #include <unistd.h>
22 #include <dirent.h>
23 #include <regex.h>
24 #include <stdio.h>
25 #include <string.h>
26 #include <errno.h>
27 #include <err.h>
28 #include <stdlib.h>
29 #include <iostream>
30 #include <string>
31 #include <set>
32 #include <map>
33 #include <vector>
34 #include <algorithm>
35 #include "mycloud_analysis.hpp"
36
37 using namespace std;
38
39 #define WATCH_FLAGS (IN_CREATE | IN_DELETE)
40 #define EVENT_CAP 100
41 #define EVENT_SIZE (sizeof(struct inotify_event))
42 #define BUF_SIZE (EVENT_CAP * (EVENT_SIZE + 50))
43 #define VAR_FILE "/var/MyCloudInotify"
44
```

```
45 // MyCloud Inotify instance descriptor struct
46 typedef struct mycloud_inotify {
47     int total_watches;
48     int num_dirs;
49     int monitor;
50     string root;
51     pid_t monitor_pid;
52     set <string> watched;
53     map <string, long> dir_map;
54     map <int, string> watch_descriptors;
55 } mycloud_inotify_t;
56
57 // Top-Level Initialization function
58 void mycloud_inotify_init(int max_watchers, int max_watches, string
       path,
59  mycloud_inotify_t &inotify);
60
61 // Directory crawling and scoring functions
62 int walk_recur_list(const char *dname, unsigned long crawl_time,
63  map<string, long> &list);
64 int get_dirs(const char *dname,  map<string, long> &dir_map);
65 void order_dirs(map<string, long> &list, vector<pair<string, long> > &
       ordered);
66 unsigned long get_last_crawl_time();
67 unsigned long update_crawl_time();
68
69 // Inotify registration and monitoring functions
70 int init_fd_watchers_set(fd_set *watchers, map<int,
71  map<int, int> > &watchers_watches);
72 int setup_watchers(int num_watchers, int num_watches,
73                    vector<pair<string, long> > &ordered,
74                    mycloud_inotify_t &inotify);
75 void catch_up(int write_to, unsigned long reference_time, string path,
76  int watch_descriptor);
77 void re_crawl(int max_watchers, int max_watches, mycloud_inotify_t &
       inotify);
78 void cleanup_watchers();
79 int init_fd_watchers_set(fd_set *watchers, map<int,
80  map<int, int> > &watchers_watches);
81 void monitor_watches(int write_to, map<int, map<int, int> > &
       watchers_watches);
82 pid_t write_monitor_pid(pid_t monitor);
83 pid_t get_monitor_pid();
```

```
1  /**
2   * mycloud_inotify.cpp
3   * My Cloud Inotify main functions.
4   * This file shall contain the functions necessary
5   * to initiazlie the My Cloud inotify to set up
6   * watchers.
7   *
8   * Donald Percivalle
9   * Scott Vanderlind
10  * Senior Project, 2015
11  */
12
13 #include "mycloud_inotify.hpp"
14
15 // Top-Level Initialization Functions
16 // max_watchers : the maximum allowed number of watchers.
17 // max_watches : the maximum allowed number of watches.
18 // path : the path to the root directory to watch.
19 // inotify : the struct holding the system state.
20 void mycloud_inotify_init(int max_watchers, int max_watches, string
       path,
21  mycloud_inotify_t &inotify){
22    vector <pair<string, long> > ordered;
23
24    // Initialize the state structure passed in.
25    inotify.root = path;
26    // Crawl the filesystem.
27    inotify.num_dirs = get_dirs(inotify.root.c_str(), inotify.dir_map);
28    // Sort the filesystem.
29    order_dirs(inotify.dir_map, ordered);
30
31    // Set up watchers for each directory, fork a child, and begin
          monitorning
32    // for changes.
33    if ((inotify.monitor
34     = setup_watchers(max_watchers, max_watches,ordered, inotify)) == 0)
           {
35       cout << "child returned\n";
36       exit(-1);
37    }
38    // Update the state with the pid of the child process.
39    inotify.monitor_pid = get_monitor_pid();
40 }
41
42 /**
```

```
43  * Recursively visit folders and have them scored.
44  * dname - The folder path to scan.
45  * crawl_time - THE LAST CRAWL TIME! DECEPTIVE!
46  * dir_map - the map inside where scanned directories should be
        recorded.
47  */
48 int walk_recur_list(const char *dname, unsigned long crawl_time, map<
        string,
49  long> &dir_map){
50    static int count = 0;
51    DIR *dir;
52    struct dirent *dent;
53    struct stat s;
54    char fn[FILENAME_MAX];
55    int len;
56    int score;
57
58    if (!(dir = opendir(dname)))
59       return 0;
60    if (!(dent = readdir(dir)))
61       return 0;
62
63
64    // Now do the recurse loop
65    do {
66       // If we're actually looking at a directory...
67       if ((dent->d_type = DT_DIR)) {
68          len = snprintf(fn, sizeof(fn)-1, "%s/%s", dname, dent->d_name)
              ;
69          fn[len] = 0;
70
71          // Ignore dotfiles and parent directories.
72          if (strcmp(dent->d_name, ".") == 0 || strcmp(dent->d_name, ".."
              ) == 0)
73             continue;
74          // Stat the dir.
75          if (stat(fn, &s) == 0){
76             if (S_ISDIR(s.st_mode)){
77                // If we have a directory, increment the count.
78                count++;
79                // Score it while we're here.
80                score = analyze_dir(fn, crawl_time);
81                dir_map[fn] = score;
82                // Recurse into directory
83                count += walk_recur_list(fn, crawl_time, dir_map);
84             }
```

```
85            }
86        }
87    } while ((dent = readdir(dir)));
88    // Close the door behind us.
89    closedir(dir);
90    // Return the number of dirs we've seen.
91    return count;
92 }
93
94 /**
95  * Crawl the filesystem and keep track of when it happened.
96  */
97 int get_dirs(const char *dname, map<string, long> &dir_map){
98    unsigned long crawl_time;
99    DIR *dir;
100   struct dirent *dent;
101   char fn[FILENAME_MAX];
102   int len;
103
104   crawl_time = get_last_crawl_time();
105   // Manually add root itself to the map
106   if (!(dir = opendir(dname)))
107      return 0;
108   if (!(dent = readdir(dir)))
109      return 0;
110   cout << "Adding manual\n";
111   len = snprintf(fn, sizeof(fn)-1, "%s", dname);
112   fn[len] = 0;
113   dir_map[fn] = analyze_dir(fn, crawl_time);
114
115   return walk_recur_list(dname, crawl_time, dir_map);
116 }
117
118 /**
119  * Sort the unordered list of directories into a vector.
120  */
121 void order_dirs(map<string, long> &list, vector<pair<string, long> > &
      ordered){
122   // Copy [dir, score] pairs into vector
123   for (auto itr = list.begin(); itr != list.end(); ++itr)
124      ordered.push_back(*itr);
125
126   // Sort by score.
127   sort(ordered.begin(), ordered.end(), [=](const pair<string, int> &a,
128                                            const pair<string, int
                                                > &b){
```

```
129         return a.second > b.second || (a.second == b.second && a.first >
                b.first);
130     });
131 }
132
133 /**
134  * Fetch the time of the last crawl.
135  */
136 unsigned long get_last_crawl_time(){
137     int fp;
138     unsigned long crawl_time;
139
140     crawl_time = 0;
141     if ((fp = open(VAR_FILE, O_RDONLY)) > 0){
142         // File exists, read time (should be first and only thing)
143         read(fp, &crawl_time, sizeof(unsigned long));
144         close(fp);
145     }
146
147     printf("Time: %lu\n", crawl_time);
148     return crawl_time;
149 }
150
151 /**
152  * Update the var file to include the current time.
153  */
154 unsigned long update_crawl_time(){
155     int fp;
156     unsigned long crawl_time;
157
158     crawl_time = 0;
159     // Open the file for writing.
160     if ((
161      fp = open(VAR_FILE, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IRGRP
             )) > 0){
162         // Get the time.
163         crawl_time = time(NULL);
164         // Make sure at least something was written.
165         if (write(fp, &crawl_time, sizeof(unsigned long)) <= 0){
166             perror("write call");
167             exit(-1);
168         }
169         // Close the file.
170         close(fp);
171     }
172     else {
```

```
173         perror("open call");
174         exit(-1);
175     }
176     // Return the time, it could come in handy to someone.
177     return crawl_time;
178 }
179
180 /**
181  * Returns the FD for a pipe that streams file change events
182  * from the provided map of watch descriptors.
183  */
184 int setup_watchers(int num_watchers, int num_watches,
185                    vector<pair<string, long> > &ordered,
186                    mycloud_inotify_t &inotify){
187     // Our special mapping of pseudo-fds to inotify fds.
188     // There are three ints here. The first is the watcher fd..
189     // The second is the original watch descriptor
190     // The third is *our* unique watch descriptor
191     map<int, map<int, int> > watcher_watches;
192     // The number of watches we have created.
193     int count_watches = 0;
194     // The number of watchers we have created within a watcher.
195     int count_watchers = 0;
196     // The total number of watches we have created.
197     int total_watches = 0;
198     // Helper vars, the current watcher we're filling and a temp fd
        holder.
199     int cur_watcher, temp_watch_d;
200     // Our pipe to the child process.
201     int pipefd[2];
202     // The PID of the child process.
203     pid_t child_pid;
204     // Time references
205     unsigned long this_crawl_time, last_crawl_time;
206
207     // Get last crawl time and update it
208     last_crawl_time = get_last_crawl_time();
209     this_crawl_time = update_crawl_time();
210
211     // Init pipefd as a pipe
212     pipe(pipefd);
213
214     // For each pair in the ordered vector, create a watch within a
        watcher.
215     for (auto itr = ordered.begin(); itr != ordered.end(); ++itr){
216         // If the number of watches per watch has filled up
```

```
217         // or this is the first watcher, init a new one.
218         if (count_watchers == 0 || count_watches >= num_watches){
219             // Increment our watcher count.
220             count_watchers++;
221             // Reset our watch count.
222             count_watches = 0;
223             // If we would create more watchers than your body has room
                    for,
224             // cut and run.
225             if (count_watchers > num_watchers){
226                 cout << "Reached watcher limit!\n";
227                 break;
228             }
229             cout << "Initiing new watcher----------\n";
230             // Create a new watcher.
231             cur_watcher = inotify_init();
232         }
233         // Increment the watch counts.
234         count_watches++;
235         total_watches++;
236
237         // These spoofed events should use our pseudo watch descriptor,
                which is
238         // total_watches.
239         if (inotify.watched.count(itr->first)) // Directory was already
                watched
240             catch_up(pipefd[1], this_crawl_time, itr->first, total_watches
                    );
241         else { // Directory hasn't been watched before
242             catch_up(pipefd[1], last_crawl_time, itr->first, total_watches
                    );
243             inotify.watched.insert(itr->first);
244         }
245
246         // Create a watch for the file path using the current watcher
247         temp_watch_d
248          = inotify_add_watch(cur_watcher, itr->first.c_str(), WATCHFLAGS
                );
249
250         // Map the inotify-provided file descriptor to a pseudo-
                descriptor
251         // since the inotify-provided FD is not bound to be unique.
252         watcher_watches[cur_watcher][temp_watch_d] = total_watches;
253         inotify.watch_descriptors[total_watches] = itr->first;
254
255         // Debug print
```

```
256         cout << itr->first << ' ' << itr->second << '\n';
257     }
258
259     // Fork a child process.
260     if ((child_pid = fork()) == 0){
261         // The child does not need to read from the pipe, only write.
262         // Close the read end of the pipe.
263         close(pipefd[0]);
264         // Begin monitoring the watchers we created above.
265         monitor_watches(pipefd[1], watcher_watches);
266     }
267     else if (child_pid < 0)
268         perror("fork call");
269     else {
270         // Write the PID of the child process to file so we can kill it
                l8r.
271         write_monitor_pid(child_pid);
272         // The parent does not need to write to the pipe, only read.
273         // Close the write end of the pipe.
274         close(pipefd[1]);
275         // Close inotify watcher fd(s) from parent's perspective
276         for (auto itr = watcher_watches.begin(); itr != watcher_watches.
                end(); ++itr)
277             close(itr->first);
278         return pipefd[0];
279     }
280
281     return 0;
282 }
283
284 /**
285  * Look in directory for files that would not have triggered events
286  * based on given reference_time, and spoof events for them.
287  */
288 void catch_up(int write_to, unsigned long reference_time, string path,
289  int watch_descriptor){
290     static struct inotify_event *event
291      = (inotify_event *)calloc(sizeof(inotify_event) + PATH_MAX, 1);
292     static char fn[PATH_MAX];
293     struct stat attrib;
294     struct dirent *ent;
295     DIR *dir;
296     int len;
297
298     // This is the first crawl, we don't want to spoof events yet
299     if (reference_time == 0)
```

```
300         return;
301
302     // Open directory, and crawl through files in it
303     if ((dir = opendir(path.c_str()))) != NULL){
304         while ((ent = readdir (dir)) != NULL) {
305             if (strcmp(ent->d_name, ".") == 0 || strcmp(ent->d_name, "..")
                    == 0)
306                 continue;
307             len = snprintf(fn, sizeof(fn)-1, "%s/%s", path.c_str(), ent->
                    d_name);
308             fn[len] = 0;
309             stat(fn, &attrib);
310             cout << fn << '\n';
311             if (S_ISREG(attrib.st_mode)
312              && (unsigned long)attrib.st_mtime > reference_time){
313                 event->wd = watch_descriptor;
314                 // The only event supported is MODIFY because unix only
                        tracks
315                 // access, modification, and change times.
316                 event->mask = IN_MODIFY;
317                 event->cookie = 0x00;
318                 event->len = strlen(fn);
319                 strcpy(event->name, fn);
320                 // Now write event out to pipe
321                 if (write(write_to, event, EVENT_SIZE + event->len) <= 0)
322                     perror("catch up: write call");
323             }
324         }
325         closedir (dir);
326     }
327 }
328
329 /**
330  * Re_crawl the file structure.  Kill off the child, then rescan for a
        list
331  * of directories.  Then sort again.  Then setup watchers.  Basically
        init
332  * again, but kill the child first.
333  */
334 void re_crawl(int max_watchers, int max_watches, mycloud_inotify_t &
        inotify){
335     // Kill child
336     cleanup_watchers();
337     mycloud_inotify_init(max_watchers, max_watches, inotify.root,
            inotify);
338 }
```

```
339
340
341 /**
342  * Kill the child process.
343  */
344 void cleanup_watchers(){
345     kill(get_monitor_pid(), SIGKILL);
346     cout << "killed child\n";
347 }
348
349 /**
350  * Insert all watches into a FD set.
351  */
352 int init_fd_watchers_set(fd_set *watchers,
353  map<int, map<int, int> > &watcher_watches){
354     int max_fd = 0;
355
356     // Clear out fd_set
357     FD_ZERO(watchers);
358
359     // Add descriptors to set and keep track of the max file descriptor.
360     for (auto itr = watcher_watches.begin(); itr != watcher_watches.end
        (); ++itr)
361     {
362         FD_SET(itr->first, watchers);
363         max_fd = (max_fd < itr->first) ? itr->first : max_fd;
364     }
365
366     return max_fd + 1;
367 }
368
369 /**
370  * Monitor all watches in the map and redirect their events to the pipe
371  * `write_to`.
372  */
373 void monitor_watches(int write_to, map<int, map<int, int> > &
    watcher_watches){
374     int num_descriptors, ready_descriptors, i, len;
375     fd_set *watchers = (fd_set *)malloc(sizeof(fd_set));
376     char *buf = (char *)calloc(BUF_SIZE, sizeof(char));
377     struct inotify_event *event;
378
379     while(1){
380         // Call fd_set for every watch and get the max descriptor.
381         num_descriptors = init_fd_watchers_set(watchers, watcher_watches)
            ;
```

```
382         // select() and wait around for fun times.
383         ready_descriptors = select(num_descriptors, watchers, NULL, NULL,
                NULL);
384         if (ready_descriptors < 0){
385             perror("select call");
386             exit(-1);
387         }
388         // Loop through watchers for event
389         for (auto &watch : watcher_watches){
390             memset(buf, 0, BUF_SIZE);
391             // If it is set
392             if (FD_ISSET(watch.first, watchers)){
393                 // Read in the events.
394                 ioctl(watch.first, FIONREAD, &len);
395                 if ((len = read(watch.first, buf, len)) < 0){
396                     perror("monitor: read call");
397                     continue;
398                 }
399                 // Digest all the pending events.
400                 i = 0;
401                 while (i < len){
402                     // Cast to a bona fide event.
403                     event = (struct inotify_event *)(buf + i);
404                     event->wd = watch.second[event->wd];
405                     if (write(write_to, buf, event->len + EVENT_SIZE) <= 0){
406                         perror("monitor: Write call");
407                     }
408                     // Get the next event.
409                     i += event->len + EVENT_SIZE;
410                 }
411             }
412         }
413     }
414 }
415
416 /**
417  * Write the pid of the monitor process.
418  */
419 pid_t write_monitor_pid(pid_t monitor){
420     int fp;
421     // Open the var file for writing.
422     if ((fp = open(VAR_FILE, O_WRONLY, S_IRUSR | S_IRGRP)) > 0){
423         // jump the file pointer over the last crawl time.
424         if (lseek(fp, sizeof(unsigned long), SEEK_SET) <= 0){
425             perror("lseek call");
426             kill(monitor, SIGKILL);
```

```
427             exit(-1);
428         }
429         // Write the pid
430         if (write(fp, &monitor, sizeof(pid_t)) <= 0){
431             perror("write call");
432             exit(-1);
433         }
434         // Close up shop.
435         close(fp);
436     }
437     else {
438         perror("open call");
439         exit(-1);
440     }
441     // Return the pid written.
442     return monitor;
443 }
444 
445 /**
446  * Fetch the PID of the monitor process.
447  */
448 pid_t get_monitor_pid(){
449     int fp;
450     pid_t monitor;
451     // Open the file for reading.
452     if ((fp = open(VAR_FILE, O_RDONLY)) > 0){
453         // Jump the file pointer over the last crawl time.
454         if (lseek(fp, sizeof(unsigned long), SEEK_SET) <= 0){
455             perror("monitor (get pid): lseek call");
456             exit(-1);
457         }
458         // Read the time.
459         if (read(fp, &monitor, sizeof(pid_t)) <= 0){
460             perror("monitor (get pid): read call");
461         }
462         close(fp);
463     }
464     // return what we just read.
465     return monitor;
466 }
```

5.3 Directory Analysis Module

```
1  /**
2   * mycloud_analysis.hpp
3   * My Cloud Inotify Analysis header
4   * This file shall contain the functions necessary
5   * to analyze directories
6   *
7   * Donald Percivalle
8   * Scott Vanderlind
9   * Senior Project, 2015
10  */
11
12 #include <sys/stat.h>
13 #include <sys/types.h>
14 #include <time.h>
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <string.h>
18
19 long analyze_dir(char *dir, unsigned long crawl_time);
```

```
1  /**
2   * mycloud_analysis.cpp
3   * My Cloud Inotify Analysis header
4   * This file shall contain the functions necessary
5   * to analyze directories
6   *
7   * Donald Percivalle
8   * Scott Vanderlind
9   * Senior Project, 2015
10  */
11
12 #include "mycloud_analysis.hpp"
13
14 long analyze_dir(char *dir, unsigned long crawl_time){
15     long score;
16     struct stat attrib;
17
18     stat(dir, &attrib);
19     score = attrib.st_mtime - crawl_time;
20
21     return score;
22 }
```

References

[1] R. Love, "on issues of recursive monitoring and races therein," October 2004. [Online]. Available: https://mail.gnome.org/archives/dashboard-hackers/2004-October/msg00022.html

Robert Love's proposal of the Love-Trowbridge Recursive Directory Scanning Algorithm

[2] ——, "Inotify monitoring of directories is not recursive. is there any specific reason for this design in linux kernel?" August 2012. [Online]. Available: http://www.quora.com/Inotify-monitoring-of-directories-is-not-recursive-Is-there-any-specific-reason-for-this-design-in-Li

Robert Love justifies why inotify recursion should be implemented in user space rather than kernel space.

[3] A. Morgado, "Filesystem monitoring in the linux kernel," November 2013. [Online]. Available: http://www.lanedo.com/filesystem-monitoring-linux-kernel/

A discussion of the capabilities of fanotify vs inotify vs dnotify