Efficient Tracking of Modified Files

Himanshu Jindal and Piyush Kansal
Stony Brook University
Adviser: Dr. Erez Zadok, Stony Brook University

Abstract

Backup systems need an efficient method to find the last modified files in a directory of user's choice, so that they can be backed up immediately. The existing implementations, either scans the directory recursively or consumes more memory by creating a per file data structure which is bigger in size (as compared to our implementation).

We have designed and built a system, which does this tracking efficiently (in terms of both time and memory). It works because of our *simple* yet *robust* design and a much smaller memory footprint. As per the experiments, our system needs *92% lesser memory* and a *slightly higher execution time* as compared to Inotify, which we used to benchmark our results. Inotify is present in Linux mainstream code.

1 Introduction

Backup systems like Dropbox and SugarSync backs-up the files on Clouds. It is the responsibility of user to define which directory he wants to backup. Once it is defined, all the files belonging to that directory should be backed up periodically or instantly. To figure out which files should be backed-up, backup systems need to keep a track of newly created and modified files. File modifications can be anything, which changes the data in a file, its timestamp or owner etc. So, it is quite important for backup systems to keep this track in the most efficient way, i.e. without consuming much CPU cycles and memory.

There are two ways in which existing implementations solve this problem:

1. **Depth First Search (DFS)**: do a (DFS) on the user defined directory and check the timestamp of each file and figure out whether it is latest or was modified in the last couple of minutes or hours. If yes, they mark it for back-up and thus continue their search deep down the directory hierarchy. The problem here is that DFS is costly and execution time increases with deeper directory hierarchies

2. **Per-File Data Structure:** maintain the information about modified files in a per-file data structure. So, as soon as a file is created or modified in directory, a new data structure is created and appended to a master list of data structure. This approach is used in Inotify. However, here are the couple of problems with Inotify:

1. The per-file data structure size is bigger and thus consumes more memory [2]
2. Inotify cannot track the files present in the user defined directory recursively. So, a separate watch has to be added on all the child directories [1]
3. For large files, it does not keep track of the exact chunks that were changed. So, if a file is 4GB in size and user changes just 1 byte at location 2GB, then it reports that the file got modified and not the location or chunk. So, in this case, the backup system would end backing up the whole 4GB file, which is unnecessary waste of computing resources
4. If multiple changes are made to the same file, then Inotify generates a separate event for all these modifications using a separate 272 bytes data structure for each event. Thus, the memory requirement increases manifold even for single file [3]

The system, which we have designed, solves these problems as follows:

1. Does not perform DFS to figure out the modified files
2. Maintain a per-file data structure, which is smaller in size as compared to one used by Inotify
3. User needs to set the watch only on a parent directory and not on all of its child directories
4. Keeps track of all the file chunks which were changed, so that only the required sector from disk can be read and backed-up instead of backing up the whole file again. This can be very helpful in minimizing number of disk reads and will thus be efficient for both backup system and computing resources
5. Multiple changes on the same file are traced using a constant amount of memory

The rest of this report is organized as follows: section 2 describes the design of the system, section 3 evaluates the performance of the system, section 4 describes the future changes and section 5 lists down the references.

2 Design

Our 5 design goals in decreasing priority were:

Computation resources friendly:

1. **Memory efficient:** Track all modified files and directories using minimum amount of memory
2. **Tracking exact file chunk:** To avoid backing up the whole file when only a few chunks are changed and doing it using a constant amount of memory
3. **Time efficient:** Track all modified files and directories in minimum possible time

Important Features:

4. **Recursive tracking:** User should set watch only on parent directory and then all of its child directories should be tracked recursively
5. **Multiple directories tracking:** To allow user to track multiple directories

Our implementation involved changes at both User and Kernel level.

User Level

We have implemented a user level program, uWatch, using which a user can carry out the following operations:

1. **Set watch:** takes a directory name as input and add watch on it. User can set watch on multiple directories
2. **Get watched directories:** gets the count and list of all the directories being watched
3. **Get modified files:** takes a directory as input and gets the count and list of files modified since last flush
4. **Flush:** using flush, we tried to simulate a scenario where backup system backs up all the modified files till a time t', so this information is no more required to be stored in data structure and should be flushed
5. **Remove watch:** removes the directory from watch

Kernel Level

We have implemented two system calls, watch() and get_watch() along with a Linux Kernel Module(LKM), kWatch.

1. **watch():** this system call is made when a user sets, flushes or removes a watch on a directory or gets the count of modified files or watched directories
2. **get_watch():** this system call is made when a user wants to get the list of all the parent directories being watched or all the modified files in a watched directory
3. **kWatch:** When this LKM is inserted into the kernel, it initializes the required data structures (DS) and intercepts following system calls:

sys_utimensat(), sys_utimes(), sys_open(), sys_write(), sys_unlink(), sys_truncate(), sys_ftruncate(), sys_rename(), sys_chmod(), sys_fchmod(), sys_chown(), sys_fchown(), sys_mkdir(), sys_rmdir()

The DS we have used for tracking the changes, are quite simple and efficient. We have created two data DS:

(a) **head_node**: maintains the information about directories under watch and contains following fields:

```
struct head_node {
    unsigned long inode;
    struct data_node* data;
    struct head_node* next;
    int num_data;
};
```

inode contains the inode number of the directory on which watch is added by user. *data* is a pointer to files modified inside watched directory. *next* is a pointer to other watched directories. *num_data* is count of modified files inside a watched directory.

(b) **data_node:** maintains the modified files inside the directory being watched

```
struct data_node {
    unsigned long inode;
    int bMap[4];
    struct data_node* next;
};
```

inode contains the inode number of the modified file or directory inside the watched directory. *bMap* is a 128-bit bitmap to indicate modified chunk number, ownership/mode change etc. *next* is a pointer to other watched directories. The 128-bits of *bMap* are used as follows:

Bit	Used For
0	FILE_MODIFY_BIT
1	FILE_RENAME_BIT
2	FILE_DELETE_BIT
3	FILE_CREATE_BIT
4	FILE_OWNER_BIT
5	FILE_MODE_BIT
6	FILE_TIME_BIT
7	FILE_MMAP_BIT
8-126	Chunk number
127	Rest of the chunk

Table 1: Bitmap utilization

As mentioned in Table 1, bits 8-126 are used to denote which chunk in a file was changed. We have taken the chunk size as 16KB. This value has been chosen so that it is multiple of magnetic disc sector size and page size for better efficiency. Besides, this value has been kept less considering that generally user tends to backup small files more rather than files, which go to GB in size. So, using a modest chunk size of 16KB, we are able to track a file up to (126-8+1)*16KB=1904KB in size. If the file size if bigger than this, then the last 127th bit is used to track any changes made in a file after 1904KB.

The first step in our implementation is that the LKM, *kWatch* is inserted into kernel. Then user sets a watch on his directory using *uWatch*. It triggers the *watch()* system call which then creates a new *head_node* structure in memory.

User now creates, modifies or deletes some files/folders present in this directory and since their parent directory is already being watched, a new *data_node* structure is created for all of these files/folders. The address of very first *data_node* structure is set into *head_node* structure and then rest *data_node* structures are linked to first *data_node* structure. Besides, appropriate bit is set in the bitmap *data_node.bitMap* to denote the type of change.

If user tries to add another directory to watch, then another *head_node* structure is created and appended next to previous *head_node* structure already present in memory. And this is the way this data structure grows and eventually looks like following:

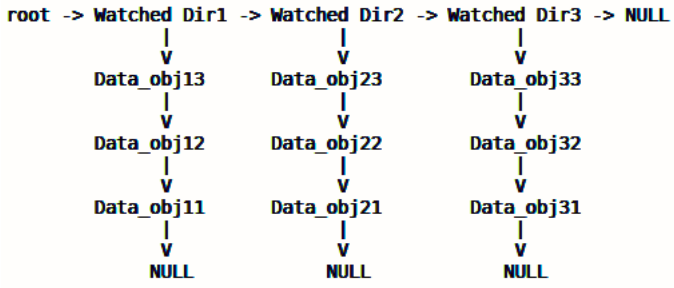

Figure 1: In memory data structure

Users can now use the *uWatch* to do following:

1. Get the list of all modified files in a directory: it first gets the number of files modified inside that directory and then calls *get_watch()* to get the list of inode numbers and bitmaps of all the modified files. Backup systems can now use this information to take further action.

2. Flush: any of the directories being watched using *uWatch* which triggers w*atch()* with an appropriate parameter to flush the complete chain under that directory and thus frees up the memory.

3. Get the list of watched directories: calls *get_watch()* with an appropriate parameter.

4. Remove watch: from a directory which calls *watch()* with an appropriate parameter

Besides, we have taken care of following scenarios:

1. If user adds a watch on directory "/a/b/" and then tries to add another watch on "/a/b/c" then an error is reported to user. This is because we are already taking care of all the child directories recursively
2. If a user tries to add a watch on directory which does not belong to him or he is not the owner, then again an error will be reported. So, this prevents any malicious user to eavesdrop on someone else files

This implementation is efficient because of following reasons:

1. It consumes only *16 bytes* for each head_node and *24 bytes* for each data_node
2. The directory being watched can be found in O(n), where n is the number of directories being watched
3. All of the last modified files in a given watched directory can also be found in O(n)
4. The design is pretty simple and robust

3 Evaluation

We have evaluated the performance of our implementation with Inotify [4, 5] on execution time and memory consumption. We used Linux 3.2.2 on a 32-bit machine for evaluation.

We divided the evaluation in two parts: (1) checks for file creation, file modification by writing data into it and file deletion. The functionality for chunk size is also taken care of in this part itself, (2) checks for file ownership change, mode change, and file rename. For both of these parts, we created test programs which takes number of test files to create as input from user and then run our implementation and Inotify to compare the results. These test programs create one directory and then create input number of files in that directory.

Following table shows the results we got:

Number of Test Files	**Execution Time in ms (kWatch)**	**Memory Footprint in KB (kWatch)**	**Execution Time in ms (Inotify)**	**Memory Footprint in KB (Inotify)**
100	0.006	2.4	0.004	27.2
500	0.024	12	0.018	136
1000	0.061	24	0.045	272

Table 1: Performance Evaluation

Following is how we will explain our test results:

1. Execution Time (kWatch): The execution time is slightly higher than that of Inotify because each time a file is modified, we check if any of its parent directory is under watch. And it is done every time a change is made to the file. So, this takes time. We have thought of a solution to this problem and are explained in section 4
2. Memory Footprint (kWatch): For 1000 files, we will be creating 1 *head_node* (16 Bytes) and 1000 *data_node* (24 Bytes), so total consumption will be 24*1000 + 16 = 24016 Bytes or 24 KB. Note that using constant amount of memory, we are able to track all 3 events i.e. file create, modify and delete. Similar calculation can be done for other test runs
3. Execution Time (Inotify): The execution is lesser than *kWatch* because Inotify makes just one check if a parent directory is under watched or not (as already explained above, Inotify requires one to set separate watches on all the sub-directories)

4. Memory Footprint (Inotify): For 1000 files and 3 kinds of events i.e. file create, modify and delete, Inotify creates 3000 *inotify_events* (272 Bytes) and 1 *inotify_watch* (40 Bytes), so total consumption will be 272*1000 + 40 = 272040 Bytes or 272KB. Similar calculation can be done for other test runs [2]

4 Future Work

We have thought of following ways by which this project can be expanded further:

1. Current implementation supports watching directories for only one user at a time. So, introducing a user id field in head_node can extend it further.
2. Current implementation does not taken into account the file system identifier. So, if two files exist on two different file systems with same inode number, out of which one file is already being watched, then current implementation would not allow setting the watch again. So, adding a file system identifier field in the head_node data structure can further extend the system.
3. Due to the simple design of the DS, we can easily save the complete DS to a file, during module unload and then read it back again, during module load. This can be helpful in scenarios, when for some reason, the backup system goes down, and then all modified files keep accumulating, causing the size of this DS to become considerably huge. At that moment, this complete data structure can be flushed to a file thus preventing any loss of information
4. If user adds a watch on directory "/a/b/" and then tries to add another watch on "/a" then an error should be reported to user wherein he is first supposed to remove the watch for "/a/b" and only then add it to "/a"
5. To reduce the execution time further, we can use a fixed size cache which will store the inode number of recently accessed files along with the inode number of their parent watched directory. So, once a file entry will be made in the cache, we will be able to figure out in constant time whether if it is supposed to watch and thus will give us better time efficiency.

As we see, most of the shortcomings (1 and 2) can be met by adding just one more integer in the data structure. Other problems (3 and 4)can be met by adding small routines in the kernel. The system can be made even more efficient by adding a sort of module cache (5). Hence, our LKM is extensible and any other required feature can be added too at a later stage.

5 References

[1] http://en.wikipedia.org/wiki/Inotify#Disadvantages
[2] http://kerneltrap.org/node/3847
[3] http://www.ibm.com/developerworks/linux/library/l-inotify/?ca=drs-#N1012A
[4] http://www.developertutorials.com/tutorials/linux/monitor-linux-inotify-050531-1133/
[5] http://www.linuxjournal.com/article/8478