Scalability in Ext2 File System

Guoliang Jin, Yancan Huang
Computer Science Department
University of Wisconsin-Madison
{aliang, hyc}@cs.wisc.edu

Abstract

Scalability is an important performance issue in modern File System. In this paper, we present our case study where we designed a simple benchmark and measured the performance of create and open operations on ext2. In order to explain an interesting result from the case study, we traced into the code and explained what actually leads to that result. Finally we added a new create mode that accelerates the create process a lot.

1. Introduction to Scalability

Scalability is an important performance issue in modern File System. Paper [1] describes the architecture and design of a scalable file system, called XFS. In that paper, the author summarized the scalability issues in a file system and designed mechanisms for the following issues respectively.

1.1. To Support Large File Systems

A scalable file system should be able to manage huge storage, while most of the current file systems are limited to either a few gigabytes or a few terabytes in size. These limitations stem from the use of data structures that don't scale, for example the bitmap, or some other linear data structures on disk.

1.2. To Support Large Files

A scalable file system should be able to manage files each of which has a few terabytes in size. Most of the current file systems use the block mapping scheme proposed in FFS [2], which does not work with variable length extents, and therefore cannot support very large files in practice. Entries in the FFS block map point to individual blocks in the file, and up to three levels of indirect blocks can be used to track blocks throughout the file. This scheme requires that all entries in the map point to extents of the same size. This is because it does not store the offset of each entry in the map with the entry, and thus forces each entry to be in a fixed location in the tree so that it can be found.

Another problem is that the mechanisms in many other file systems for allocating large, contiguous files do not scale well. Most file systems use linear bitmap structures for tracking free and allocated blocks in the file system. Finding large regions of contiguous space in such bitmaps in large file systems is not efficient.

1.3. To Support Large Directories

A scalable file system should be able to support directories with more than a few thousand entries which has not been addressed by most Unix-like systems. While some file systems at least speed up searching for entries within a directory block via hashing, most file systems use directory structures which require a linear scan of the directory blocks in searching for a particular file. The lookup and update performance of these un-indexed formats degrades linearly with the size of the directory.

1.4. To Support Large Number of Files

A scalable file system should be able to manage large number of files. While most file systems could theoretically provide support for this, in practice they do not. The reason is that the number of inodes allocated in these file systems is fixed at the time the file system is created. Choosing a very large number of inodes up front wastes the space allocated to those inodes when they are not actually used afterwards. The real number of files that will reside in a file system is rarely known at the time the file system is created. As a result, being forced to choose makes the management of large file systems more difficult than it should be.

In summary, there are several scalability problems within existing file systems. In our project, we focused on the last two scalability issues of ext2, that is, to support **large directories** and **large number of files**.

The rest of the paper is organized as following: in section 2, we briefly describe the Ext2 file system; in section 3 we propose the methodology of our project including the environmental preparation and benchmark for our experiments; section 4 shows the experiment results; section 5 tracks the kernel code to give explanations for the interesting findings in the previous section; finally we conclude in section 6 and propose our plan for the future work in section 7.

2. Ext2 File System

Unix-like operating systems use several types of file systems. Although the files of all such file systems have a common subset of attributes required by a few POSIX APIs such as stat(), each file system is implemented in a different way. The first versions of Linux were based on the Unix-like MINIX file system. As Linux matured, the Extended File System (ext FS) was introduced; it included several significant extensions, but offered unsatisfactory performance. The Second Extended File System (ext2) was introduced in 1994; besides including several new features, it is quite efficient and robust and is, together with its offspring ext3, the most widely used Linux file system. In the following subsections, we will describe the disk data structures and disk managing methods in ext2 file system. [3]

2.1 Disk Data Structures

The first block in each Ext2 partition is reserved for the partition boot sector and is never managed by the Ext2 file system. The rest of the Ext2 partition is split into block groups, each of which has the layout shown in Figure 1 as follows.

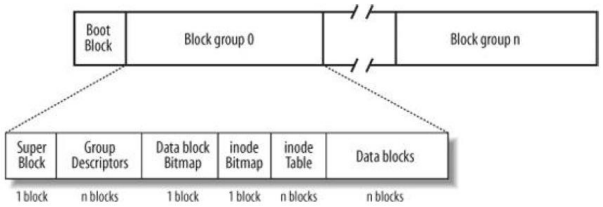

Figure 1: Layouts of an Ext2 partition and an Ext2 block group

As the figure shows, some data structures must fit in exactly one block, while others may require more than one block. All the block groups in the file system have the same size and are stored sequentially, thus the kernel can derive the location of a block group in a disk simply from its integer index.

Block groups reduce file fragmentation, because the kernel tries to keep the data blocks belonging to a file in the same block group, if possible. Each block in a block group contains one of the following pieces of information:

- A copy of the file system's superblock
- A copy of the group of block group descriptors
- A data block bitmap
- An inode bitmap
- A table of inodes
- A chunk of data that belongs to a file; i.e., data blocks

If a block does not contain any meaningful information, it is said to be free.

3. Methodology

In this section, we will discuss the methodology in this project, including the experiment environment, output techniques, measuring techniques and our benchmark.

3.1. Environmental Preparation

We conducted our experiments under User Mode Linux[4][6]. To have a clean version of ext2, we added two new file systems called ext2s and ext2k to the kernel. The ext2s is intended to modify the code to make ext2 more scalable while the ext2k is used to print out the function call path for the operations in the benchmark. Except for that, ext2s and ext2k use the original code of ext2.

As the on-disk and underlying structures of ext2s and ext2k are the same as ext2, they can be formatted using the tool designed for ext2 but mounted with file system type of ext2s or ext2k accordingly.

In the UML, ext2 and ext2s use 1GB virtual disk. Originally we used 100MB, but it turned out to be too small due to the limitation of the inode number the 100MB disk can provide.

3.2. Output Techniques

In ext2, we want to obtain the whole path of function calls in order to get a taste of what is going on during

such a benchmark workload. Therefore we modified the ext2k file system where we added printk at the beginning and end of each method which shows a nice picture of the function call path.

A main part of the output issue is about the performance data. We are going to create about 120K files, which means we need to output a lot of data in a good format to accelerate the analysis rather than spending a lot time on extracting data from a large ugly log file.

It is clear that we should use our log file so that the data can be formatted decently. But the problem is that in some methods we cannot use this approach to write our own log. In our method we call to filp_open directly, but some of the data we want comes from the functions called from directly or indirectly from filp_open. In these functions we have to use printk to avoid this circular dependence. The problem with printk is not limited to the difficulty to extract the useful data from a long log. Another problem is that the log may be disrupted by some operation outside the benchmark. The log may also lose some data as the lines of performance data we got from the system is always less than the actual number of file created or opened.

3.3. Measuring Techniques

Firstly our project aims to measure performance, so we need to obtain the time cost of each method called by our benchmark. In our benchmark, we use gettimeofday() function to get real time in seconds.

After that, we want to figure out the reason for one interesting result, so we need to find out a way to measure the time in the kernel. We use the RDTSC instruction to get the timestamp counter (TSC) of the current point. The following is the code to get current value of TSC in c:

```
long long c;
__asm__ __volatile__ ("rdtsc" : "=A" (c));
```

There is no logging overhead issue in our project. In the benchmark, we put the get time function before and after create or open and no overhead is added to create or open itself. For the TSC data we collected to find which sub-function A called by function B takes the most time of B, we measure the time for the whole B in B and the time for A in B. So we are not trying to figure out the exact percentage one function takes in the whole create or open process; instead, we just try to find which sub-function leads the whole process.

3.4. Benchmark

The benchmark we used in our project in simple. In one empty 1GB ext2/ext2s disk, we create the largest possible number of files in one directory called test. Besides the test directory, there are one file for benchmark source code and one file for binary code.

For the create process, the benchmark keeps creating files with different names until it meets the limitation of the number of inodes that the 1GB disk can provide. For the open process, the benchmark keeps opening files with the same name sequence used in the create process. We use the measuring techniques described to measure the time for creating or open each file.

4. Result of the benchmark

In this section we show a result, obtained at the beginning, which our project wants to explain.

Figure 2 shows the time for the create process. From the graph we can see that as the number of files in the directory increases, the time for the benchmark to create a new file also increases. Although the graph does not show a single line, we mainly focus the bottom of the graph which we view as a near-linear line.

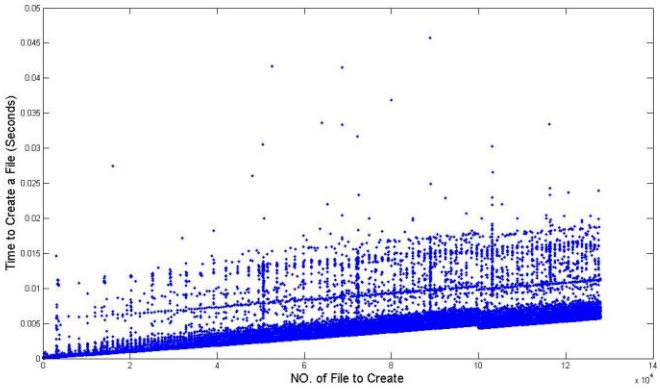

Figure 2: The time to for the create process

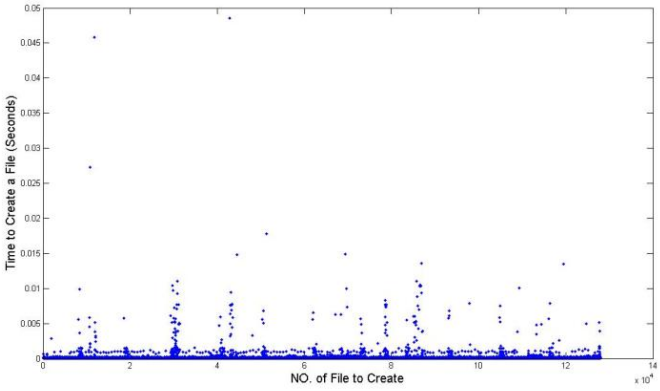

Figure 3: The time to for the open process

Figure 3 shows the time for the open process. The graph is quite flat implying that most open operations take the same amount of time.

However the create will finally go to the same function as open: sys_open. So we are going to look into the code of sys_open to explain what actually leads to the difference between these two graphs.

5. Code track

In this section, we track down how the kernel works when user want to create a new file in the ext2 file system, and measure the time to find out where the time goes. All codes shown in this section are within kernel version 2.6.24. In this section, we only describe the code path for **successful file create and open**.

5.1. sys_open

The function sys_creat for creating a new file is in file fs/open.c. The code shows sys_creat(pathname, mode) is identical to sys_open (pathname, O_CREAT | O_WRONLY | O_TRUNC, mode) [5], so we track down to function sys_open.

In sys_open, there are two function calls:

```
sys_open {
        do_sys_open
        prevent_tail_call
}
```

The measured time for prevent_tail_call is almost constantly 43 cycles using RDTSC. It shows that most time is spent on do_sys_open.

In do_sys_open, there are six function calls:

```
do_sys_open {
        getname
        get_unused_fd_flags
        do_filp_open
        fsnotify_open
        fd_install
        putname
}
```

We used RDTSC to measure the time for do_sys_open and do_filp_open. Figure 4 shows the percentage do_filp_open takes in do_sys_open. In the beginning, it just takes about 25%, but as the number in the directory increases, the percentage finally goes to near 1. So the increased time to create a new file goes to do_filp_open.

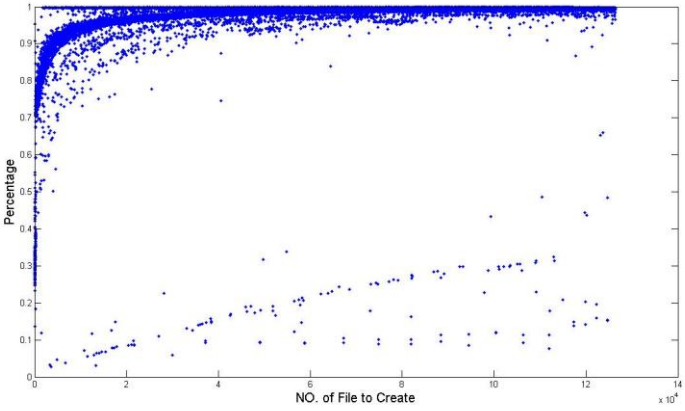

Figure 4: percentage of do_filp_open/ do_sys_open

In do_filp_open, there are two function calls:

```
do_filp_open {
        open_namei
        nameidata_to_filp
}
```

We used RDTSC to measure the time for do_filp_open and open_namei. Figure 5 shows the percentage open_namei takes in do_filp_open. The value is always close to 1 and it clearly shows that most time is spent on open_namei.

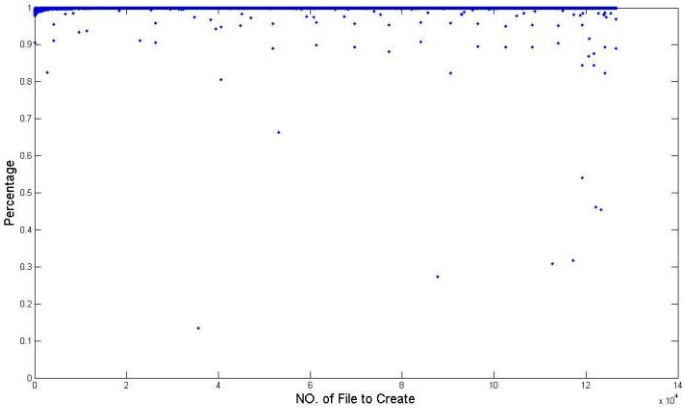

Figure 5: percentage of open_namei / do_filp_open

5.2. When open_namei meets file create

From the measured data, the time for open_namei increases while creating new files. So now we trace into the code to find out what does open_namei do for file create.

In open_namei, there are four main function calls when it creates a new file:

```
open_namei {
        ……
        path_lookup_create
        mutex_lock
        lookup_hash
        open_namei_create
        ……
}
```

We measure the time for open_namei, lookup_hash and open_namei_create. Figure 6 and 7 show the percentage lookup_hash and open_namei_create take in open_namei, and the averages are about 45% and 54% respectively.

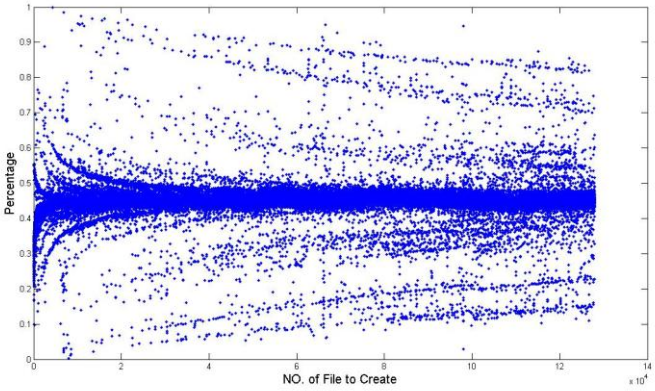

Figure 6: percentage of lookup_hash /open_namei

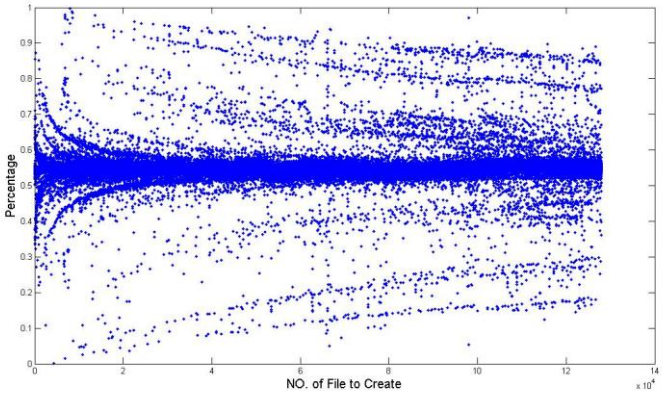

Figure 7: percentage of open_namei_create /open_namei

In lookup_hash, there are two main function calls:

```
lookup_hash {
        permission
        __lookup_hash
}
```

The measured time for permission is around 200 cycles using RDTSC. So most time of lookup_hash is spent on __lookup_hash.

In __lookup_hash, there are three main function calls: cached_lookup, d_alloc, and inode->i_op->lookup. The structure of the code is the following:

```
__lookup_hash {
        cached_lookup   // always fail
        struct dentry *new = d_alloc
        dentry = inode->i_op->lookup
        if (!dentry)        // always true
                dentry = new
        return dentry
}
```

Figure 8 show the percentage inode->i_op->lookup takes in the __lookup_hash, and the value is close to 1. It clearly shows that most time of __lookup_hash is spent on inode->i_op->lookup.

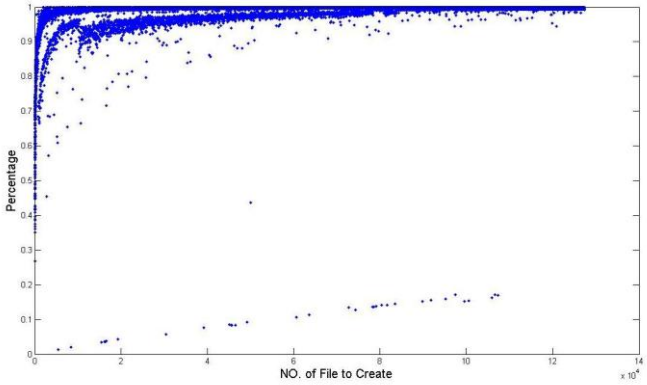

Figure 8: percentage of inode->i_op->lookup /__lookup_hash

In our benchmark, cached_lookup will fail as the file we want to create doesn't exist. inode->i_op->lookup will goes to specific implementation for each file system. In our case, it is ext2_lookup. Basically, this lookup also fails to find a directory entry with the name of the new file we want to create. So dentry always gets new, which is a new allocated directory entry. We examine this in detail in next subsection.

As inode->i_op->lookup will goes to ext2 implementation, and we want finish our trip in VFS first, so now let's take a look again at where we are now in open_namei:

```
open_namei {
        ……
        path_lookup_create
        mutex_lock
        lookup_hash
        open_namei_create
        ……
}
```

lookup_hash finally returns a directory entry for the new file the benchmark wants to create, then the code path goes to open_namei_create.

In open_namei_create, there are two main function calls:

```
open_namei_create {
        vfs_create
        may_open
}
```

We measure the time for open_namei_create and vfs_create. Figure 9 show the percentage vfs_create takes in open_namei_create. It shows that most time of open_namei_create is spent on vfs_create**.**

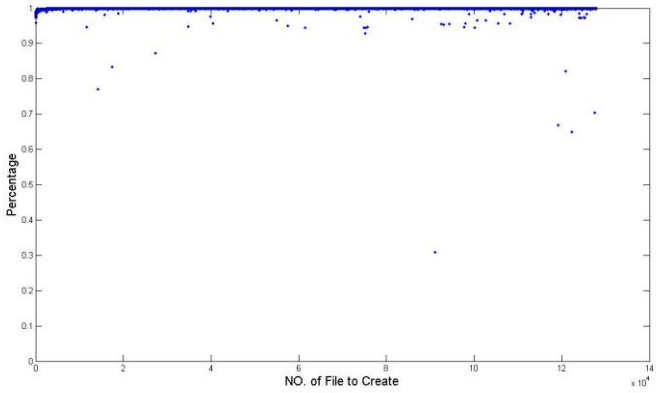

Figure 9: percentage of vfs_create / open_namei_create

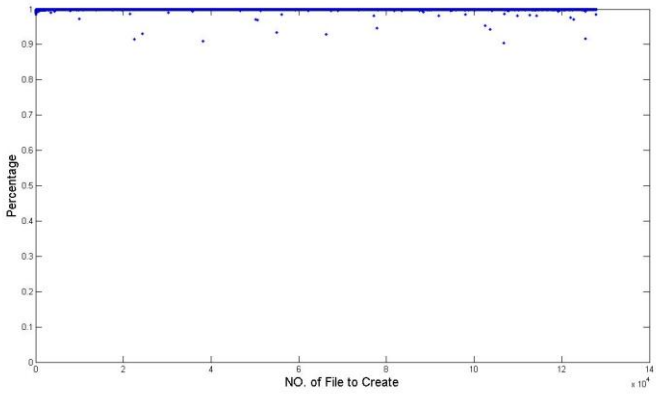

Figure 10: percentage of dir->i_op->create / vfs_create

In **vfs_create**, there are four main function calls:

```
vfs_create {
        may_create
        security_inode_create
        dir->i_op->create
        fsnotify_create
}
```

Figure 10 shows the percentage dir->i_op->create takes in the open_namei_create. It shows that most time of open_namei_create is spent on dir->i_op->create**.**

dir->i_op->create will goes to ext2 implementation, after that open_namei will return and finally the new file will be created.

5.3. ext2_lookup and ext2_creat

In this subsection, we are going to trace to the ext2 code. Unlike the previous subsection where we measured a lot of data to find out where most of the time goes, we use another way to proof our result. As we are now in ext2 where we can change the code, we add a lot of printks to find out the difference between the function call paths of the first file create and the a late file create. After that, we change the code and add a new create mode for our benchmark. Then we measure the performance data which confirms our conclusion.

The differences we found are: in the first phrase before ext2_create is called, the occurrence of pattern where one ext2_get_page and ext2_last_byte followed by multiple ext2_matchs and ext2_next_entrys increases; in the second phrase after ext2_create is called but before it returns, the occurrence of pattern where one ext2_get_page and ext2_last_byte followed by multiple ext2_matchs and ext2_rec_len_from_disks increases. This clearly shows the two level nested loop code structure.

In ext2_lookup, there are three main function calls:

```
ext2_lookup {
        ext2_inode_by_name
        iget
        d_splice_alias
}
```

In ext2_inode_by_name, there is one main function call:

```
ext2_inode_by_name {
        ext2_find_entry
}
```

The following is the main structure of ext2_find_entry:

```
ext2_find_entry {
        do {
                ext2_get_page
                ext2_last_byte

                while () {
                        ext2_match
```

```
                                        ext2_next_entry
                                    }
                                } while ()
                            }
```

This piece of code goes through the dentry memory structure of the inode of the parent directory of the new file. ext2_find_entry has a two level nested loop structure: the outer loop get one page of dentry from the inode mapping region and the inner loop goes through all these dentries. If one entry with a same name of the new file is found, the code jumps out and the found dentry will be returned. However, in our benchmark this function will go through all dentry and finally find that no existing dentry has the the same name of the new file we want create.

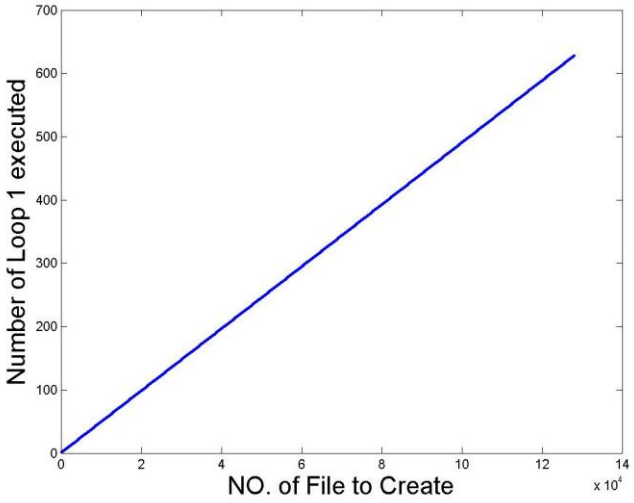

Figure 11: outer loop of ext2_find_entry and ext2_add_link

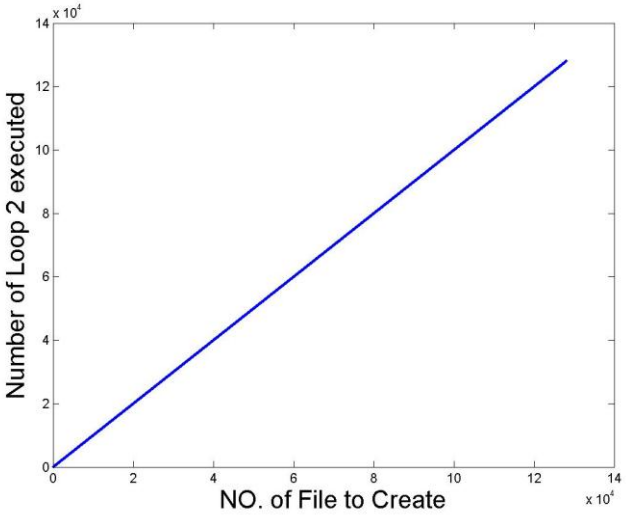

Figure 12: inner loop of ext2_find_entry and ext2_add_link

The measured TSC for ext2_get_page, ext2_last_byte, ext2_match and ext2_next_entry are almost constant, so the increased time comes from the increased number of dentries need to be checked. Figure 11 and 12 shows how many calls to ext2_get_page and ext2_match are executed while creating one new file.

The function in figure 11 is y = (int)x/204 + 1, and in figure 12 it is y = x. From the relationship, we see that there are about 204 dentries in on page.

In ext2_create, there are three main function calls:

```
ext2_create {
        ext2_new_inode
        mark_inode_dirty
        ext2_add_nondir
}
```

In ext2_add_nondir, there are two main function calls:

```
ext2_add_nondir {
        ext2_add_link
        d_instantiate
}
```

The following is the main structure of ext2_add_link:

```
ext2_add_link {
        for () {
                ext2_get_page
                ext2_last_byte
                while () {
                        ext2_match
                        ext2_rec_len_from_disk
                }
        }
        __ext2_write_begin
        ext2_commit_chunk
}
```

The structure of ext2_add_link is quite similar to ext2_find_entry.

This piece of code also goes through the dentry memory structure of the inode of the parent directory of the new file. It has a two level nested loop structure: the outer loop get one page of dentry from the inode mapping region and the inner loop goes through all these dentries. If one entry with a same name of the new file is found, this function fails. These two pieces of code are different: in ext2_add_link the for loop starts from page 0 and once it finds a empty dentry, the following code will use that dentry for the new file.

The increased time comes from the increased number of dentries need to be checked. Figure 11 and 12 shows how many calls to ext2_get_page and ext2_match are executed while creating one new file. As the two figures for ext2_find_entry and ext2_add_link are the same, we just show one for each.

5.4. A new mode S_CR736

In order to verify our conclusion in subsection 5.3, we changed the ext2 code.

As in our experiment, we know there won't be name conflict in the test directory, so when the sys_create has the mode S_CR736 in it, the modified create process will skip the call to ext2_lookup, and in ext2_add_link, we add a global valuable to indicate the previous page number a new inode linked to, and start from that page number so that we call ext2_get_page at most for two times. We also commented the ext2_match out as there won't be name conflict. Figure 13 shows the performance when the benchmark uses this mode.

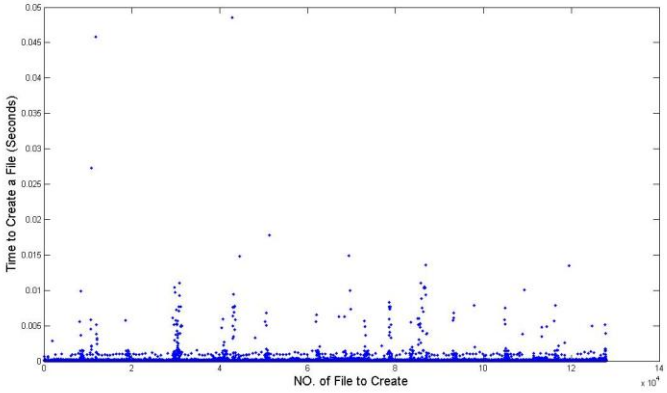

Figure 13: the time of create process using new mode

From the result, we can see that the graph for also changes to a flat one. Even better, the time to create a new file is slightly smaller than minimum of the times from the benchmark which does not use the new mode.

5.5. Code trace of open

Now we know why the time for create increases linearly as the number of files in the directory increases. So let's revisit sys_open to see why the graph for open process is flat.

```
sys_open {
        do_sys_open {
                getname
                get_unused_fd_flags
                do_filp_open {
                        open_namei
                        inameidata_to_filp
                }
                fsnotify_open
                fd_install
                putname
        }
        prevent_tail_call
  }
```

The time for create process increases and the increased part comes from open_namei, while the time of for open process is constant, so the difference comes from the function call open_namei.

When open_namei meets open, the code will go to path_lookup_open. Following shows the structure of path_lookup_open:

```
  path_lookup_open {
        __path_lookup_intent_open {
                get_empty_filp
                do_path_lookup {
                        path_walk {
                                link_path_walk {
                                        ……
                                }
                        }
                }
        }
  }
```

As the time for open process is almost constant, so we did not measure any time for particular functions.

The following is the structure of link_path_walk:

```
        link_path_walk {
                __link_path_open
                if (fail) {
                        dget
                        mntgrt
                        __link_path_open
                }
        }
```

As our benchmark opens file in the order they are created, so for most files the __link_path_open can be satisfied in the dcache which has build-in hash mechanism, and most of them do not need to go to the real but expensive lookup requests. This explains why the graph for open process is flat.

6. Conclusion

Under the current implementation of ext2, when user creates a lot of files in one directory, then the time to create each file will increase as the number of the files in the directory increases. The time increased is mostly spent on scanning through the directory entry structure of the directory. About a little less than half of the time is spent on ext2_lookup to make sure that there won't be name conflicts, so when the user calls create operation ext2 will go through all the file names in that directory. About a little more than half of the time is spent on ext2_add_link to find an empty directory entry that the new file can link to. In this process, ext2 still performs the check process to make sure there will not be name conflicts. The ext2_add_link always starts from page 0 so that it will not miss any name conflicts.

We added a new mode of create to verify that ext2_lookup and ext2_add_link are the two bottlenecks where the increasing time comes from. For the ext2_lookup, we do not even call to this method from VFS. In the new mode, the dentry for the new file will always get a new allocated one. In ext2_add_link, we used a global page number as a hint and start the empty directory entry search from that page, and the name check is commented out. The performance data we got for the benchmark using the new mode confirmed that we got the real reason why the time for create increases.

The time for create increases, however, if the benchmark opens each file in the order they are created, then the graph of time to open each file will be flat implying that most of them just take the same amount of time. The graph for create with the new mode is quite the same.

Another scalability issue in ext2 is that its inode number is determined when the disk is formatted, so in our benchmark we can create up to about 120K files in the test directory.

7. Future Work

From our study we know that the directory entry structure of ext2 is linear while the scalable file system XFS uses B-tree to manage this structure in memory, so applying B-tree in ext2 will make it more scalable. The new added mode has very good performance, but it may not be safe to be adopted into productive environment. We may study more on the directory entry structure and try to find a general optimization to make the ext2 create graph flat. Maybe we can start from the proved successful B-tree structure.

In addition, the benchmark we used in this project is too simple and maybe a little naïve. Most of this limitation comes from that we open files in the order they created. The reason why ext2 open graph is that most of the directory entry lookup requests are satisfied in the dcache, so benchmark that forces the real lookup will give a better view of the open scalability of ext2 file system. Certainly, a benchmark that can do this is still not enough, but it could be another good start.

Acknowledgement

Thanks to Prof. Remzi Arpaci-Dusseau for his guide through the whole project.

References

1. Adam Sweeney, etc. Scalability in the XFS File System. Proceedings of the USENIX 1996 Annual Technical Conference.
2. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, A Fast File System for UNIX, ACM Transactions on Computer Systems, August 1984, pages 181-197.
3. Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 3rd Edition. O'Reilly, Nov. 2005.
4. Dike Jeff. User Mode Linux. Prentice Hall, Apr. 2006.
5. W. Richard Stevens, Stephen A. Rago. Advanced Programming in the UNIX Environment: Second Edition. Addison Wesley Professional, Jun. 2005.
6. The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/

Appendix-all the routines that get called during create

```
sys_create
    sys_open
        do_sys_open
        getname
        get_unused_fd_flags
        do_filp_open
            open_namei
                path_lookup_create
                lookup_hash
                    permission
                    __lookup_hash
                        cached_lookup
                        d_alloc
                        inode->i_op->lookup          // goes to ext2_lookup
                            ext2_inode_by_name
                                ext2_find_entry
                                    do
                                        ext2_get_page
                                        ext2_last_byte
                                        while ()
                                            ext2_match
                                            ext2_next_entry
                                    while ()
                            iget
                            d_splice_alias
                open_namei_create
                    vfs_create
                        may_create
                        security_inode_create
                        dir->i_op->create            // goes to ext2_create
                            ext2_new_inode
                            mark_inode_dirty
                            ext2_add_nondir
                                ext2_add_link
                                    for ()
                                        ext2_get_page
                                        ext2_last_byte
                                        while ()
                                            ext2_match
                                            ext2_rec_len_from_disk
                                    __ext2_write_begin
                                    ext2_commit_chunk
                                d_instantiate
                        fsnotify_create
                    may_open
            inameidata_to_filp
        fsnotify_open
        fd_install
        putname
    prevent_tail_call
```