CS 551 Project 3

Sri Sai Ranga Malisetti (**A20523038**)
Poorna Venkat Neelakantam(**A205177752**)
Prabhu Avula (**A20522815**)

Submission from the **Team - 7**

1. Part 1: Implementation of the three system calls

Semantics of the system call implementation

The initial part of the project involves the implementation of three system calls which in theory could improve the performance of the read and write system calls in the minix file system. The calls to be implemented are listed below.

a. nicerTo()

The nicerTo method is designed to change the priority of a process whose process id is passed as an argument in the function. Although the schedule (**sched**) server already contains the **do_nice**() method available, it only works in the user space of the process. To move further and give a process the highest priority, the kernel must execute the schedule process itself in the kernel space. Hence, this system call is implemented to be traversed all the way to the kernel to update the priority of the process to **TASK_Q.** The implementation of this system call was similar to the system calls which were implemented in Project 2. A more detailed description of the implementation of the nicerTo system call can be found in the **NicerTo.md**, which is attached with the project submission. A brief description of the implementation is discussed below.

1. Implementation of the noxfer_message

To handle the data passed in the message when the process manager (**pm)** server is invoked from the user function and the passage of data from the pm server to the kernel, two messages with the process id as the payload and endpoints are required respectively. Two such messages are implemented **mess_lc_pm_nicerto** and **mess_lsys_krn_sys_nicerto** in the `minix/minix/include/minix/ipc.h` file.

2. Implementation of the kernel call

Implementation of the kernel call requires modifying the internal files in the `minix/minix/kernel` folder. This folder along with the implementation of the **sys_*** wrapper will enable us to call the sys function and invoke the kernel call. The declarations for the kernel calls are implemented in the `minix/minix/kernel/system.h` the implementation of the system calls is written in the `minix/minix/kernel/system/` folder. The main logic for the kernel call is located in the **do_nicerto.c** file. There are a lot of Makefile modifications to support the compilation of the included files.

3. Implementation of the system call wrapper

The kernel call needs to be implemented with certain privileges, and needs to be called using the _kernelcall function to be executed. These functions are called from the sys call wrapper which are declared in the `minix/minix/include/minix/syslib.h` which are later implemented in the `minix/minix/lib/libsys/` folder under the names

sys_nicerto.c file. There are a lot of Make file and lib files that have to be updated to handle the new system call function.

4. **Implementing the system call in the process manager server**

The system call can not be executed on its own in minix, hence a server has to send the message to execute the system call from the process manager service. The declaration of the prototypes of the functions to implement the system calls are defined in `minix/minix/servers/pm/proto.h`. The implementations of the declared functions are written in a file in the `pm` folder in a file called `improve.c`. The additional call numbers etc. are declared as required to support the message execution.

5. **Library implementation of the implemented system calls**

The system calls that we have implemented require a POSIX standard library implementation. The prototype is declared in the `minix/include/unistd.h` to support along with the function implementations in a newly created header file `minix/include/cs551.h`. The _sys_call method is called along with the required parameters and the equivalent return types and error handling.

6. **Unit testing of the implemented function**

The nicerto() method is defined to take the process id as an argument and give it the highest priority available. The syscall in the process manager can be invoked along with the process id as given below.

```
message m;
memset(&m, 0, sizeof(m));
m.m_lc_pm_nicerto.pid = getpid();
int i = _syscall(PM_PROC_NR, PM_NICERTO, &m);
```

The _syscall method returns **OK**(0) for success, by which we can identify the successful implementation of the system call.

b. moreCache()

The moreCache method is designed to change the number of buffers assigned to a block cache in the Minix File System. The factor by which the number of buffers are to be increased is passed as an argument in the function. The call is made from the Virtual File System (**vfs**) server which is executed by the minix file system (**mfs**). The implementation of the moreCache method is different from the nicerTo method as the modification of the cache involves changing the **fs driver** along with the modification of the minix file system server which maintains the file system operations in minix. A more detailed description of the implementation of the moreCache system call can be found in the **MoreCache.md**, which is attached with the project submission. A brief description of the implementation is discussed below.

1. **Implementation of the noxfer_message**

To handle the data passed in the message when the virtual file system (**vfs)** server is invoked from the user function and the passage of data from the vfs server to the mfs server through the request pipeline, two messages with the new buffer factor as the payload are required. Two such messages are implemented **mess_vfs_fs_morecache** and **mess_lc_vfs_morecache** in the `minix/minix/include/minix/ipc.h` file.

2. Implementation of the system call in mfs server

The Minix File System is implemented as an additional layer on the file system driver inheriting the methods. All the requests from the virtual file system are routed through the pipeline, where the request is received by the mfs server running and executes them. We also have to declare functions in the libfsdriver to process the incoming request from the vfs. Hence, we declare a new request `REQ_MORECACHE` in the `minix/include/minix/vfsif.h` file. The driver function to process the request is implemented in the `minix/lib/libfsdriver/call.c` file.

The system call from the virtual file system through the fsdriver reaches the minix file system to be executed. The declaration of the function in `minix/fs/mfs/proto.h` and the implementation of the function in `minix/fs/mfs/misc.c` completes the call which was made by the user to the minix file system. The **fs_morecache** contains the implementation of the desired more cache method.

Additionally, a new method was introduced in the **libminixfs** to get the number of buffers that are currently assigned and used in the disk block cache. The declaration is written in the `minix/include/minix/libminixfs.h` and the implementation of the method in the `minix/lib/libminixfs/cache.c` file.

3. Implementation of the system call in vfs server

The vfs server contains the method available for the user to access which in turn invokes the methods in the mfs server through the request pipeline. The declaration of the prototypes of the functions to implement the system calls are defined in `minix/minix/servers/vfs/proto.h`. The implementations of the declared functions are written in a file in the `vfs` folder in a file called `cache_improve.c`. The **req_morecache** function has the implementation in the `minix/minix/servers/vfs/request.c` file to forward the request to the mfs server. The additional call numbers etc. are declared as required to support the message execution.

4. Library implementation of the implemented system calls

The system calls that we have implemented require a POSIX standard library implementation. This step is completely similar to the library implementation for the nicerTo function.

5. Unit testing of the implemented function

The morecache() method is defined to take the new buffer factor as an argument and increase the **num_bufs** in the disk cache. The syscall in the virtual file system can be invoked along with the process id as given below.

```
message m;
memset(&m, 0, sizeof(m));
m.m_lc_vfs_morecache.new_buf_factor = 2;
int j = _syscall(VFS_PROC_NR, VFS_MORECACHE, &m);
```

The _syscall method returns **OK**(0) for success, by which we can identify the successful implementation of the system call.

c. moreZone()

The moreZone method is designed to change the number of blocks per zone in the Minix File System. The additional blocks by which the number of blocks per zone are to be increased is passed as an argument in the function. The call is made from the Virtual File System (**vfs**) server which is executed by the minix file system (**mfs**). The implementation of the moreZone method is completely similar to the moreCache method. It follows the same

steps of modification of the mfs which involves changing the **fs driver** along with the modification of the minix file system server which maintains the file system operations in minix. A more detailed description of the implementation of the moreZone system call can be found in the **MoreZone.md**, which is attached with the project submission.

Design philosophy of the calls

The semantics of the system calls deal with the methodology or the logical pipeline built for the user to invoke certain system calls or the kernel calls. The design philosophy of the system calls is the main function of the system call that it was defined for. The brief information about the design of the implementation of the system calls.

a. nicerTo()

The core design of the nicerTo method is implemented in the **do_nicerto.c** in the kernel of the minix os. The method receives the endpoint of the process whose priority has to be increased from the process manager server. The function evaluates the proc from the given endpoint and calls the **sched_proc** with the **TASK_Q** priority which is the max priority that can be assigned to the process in the minix os.

b. moreCache()

The core design of the moreCache method is implemented in the `minix/fs/mfs/misc.c` in the minix file system. The method receives the factor by which the **nr_bufs** must be increased from the virtual file system. The function gets the current **nr_bufs** that are being used in the cache and calls the **lmfs_buf_pool** with the multiplication result of the **nr_bufs** and the factor which was sent by the vfs server.

c. moreZone()

The core design of the moreCache method was also to be implemented in the minix file system. The blocks per zone value which was defined by the **s_log_zone_size** which is the `log2 of blocks/zone` according to the comments in the minix source code. Additionally, it was found that increasing the number of blocks per zone has been abandoned by the developers in the minix os and any changes to the value might result in the function returning an **`EINVAL`**. Below is the snippet in the code mentioning the same in the source code.

```
/* Zones consisting of multiple blocks are longer supported, so fail as
early
* as possible. There is still a lot of code cleanup to do here, though.
*/
if (sp->s_log_zone_size != 0) {
   printf("MFS: block and zone sizes are different\n");
   return EINVAL;
}
```

Unit tests of the calls

The **syscall_test.c** is a simple fork method which calls the implemented system calls. The test results of the running of the call is shown below.

a. nicerTo()

The screenshot below shows the output from the nicerTo() with the process id passed.

```
minix# clang syscall_test.c
minix# ./a.out
I am the child with pid 646
Calling sys nicerto from pm service
Making a kernel call next
sys_nicerto: 98490
do_nicerto: reached: 98490
Child process is exiting
Parent process is exiting
minix# _
```

b. moreCache()

The screenshot below shows the output from the moreCache() with the increase factor passed as 2.

```
minix# clang syscall_test.c
minix# ./a.out
I am the child with pid 652
Calling sys morecache from vfs service
req_morecache: invoked
fsdriver_morecache: request init
fs_morecache: function invoked
fs_morecache: new_bufs: 2430 -> 4860
fsdriver_morecache: reqeust complete : ret: 0
Child process is exiting
Parent process is exiting
minix#
```

2. Part 2: Optimization of the file handling

Test bench configuration

The minix operating system was installed on a virtual machine with 2 cores and 2 threads along with 4GB of ram. It is the developmental version, **3.4.0** and the file read write tests were rerun with multiple processes with the read and write system calls along with the use of the implemented optimization techniques. The code snippet run on the os was compiled by clang.

Motivations for caching

The file was run initially to create a file and to perform the read and write operations. It took around **0.5s** to run as shown in the image below.

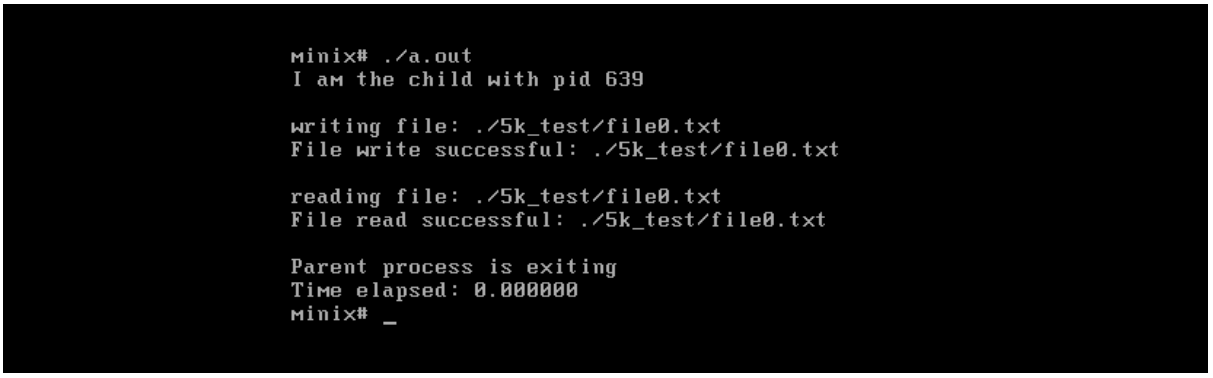
```
minix# clang file_read_write.c
minix# ./a.out
I am the child with pid 637

writing file: ./5k_test/file0.txt
File write successful: ./5k_test/file0.txt

reading file: ./5k_test/file0.txt
File read successful: ./5k_test/file0.txt

Parent process is exiting
Time elapsed: 0.050000
minix# _
```

But when the same file was run for the second time we noticed that the time taken was less than the range we were trying to capture, which resulted in it showing **0s**.



This above example shows the importance of caching that is implemented in the minix file system and improving it might result in good performance increase and efficiency of the operating system.

Methods to improve performance

There are various theoretical optimization techniques that were experimented earlier. Although they offer some improvements in the performance, many were discarded due to some issues. We have experimented with two such optimization techniques. Both the optimization techniques are discussed in detail below.

a. nicerTo()

The nice method of optimization is essentially increasing the priority of the process to make the process in focus have the highest priority in the runtime. This method however ignores the case that it only affects the scheduling algorithm, and if the other components are slower might not result in improved performance and in some cases might lower the performance as well. Changing of the priority might be of the highest improvement in performance for IO tasks.

b. moreCache()

Increase in the cache however heavily depends on the hit ratio of the cache. This performance was visible in minix as files opened for the second time were accessed much quickly. Hence, adding more buffers to the cache might result in faster performance in the cache.

Additional methods such as multithreading in the minix file system were considered but it depends on the architecture of the operating system and the message passing etc. in minix makes it harder to find zones of optimization. There is a lot of fluctuation in the operating system during intensive tasks.

The test file was **optimization_test.c** file attached with the submission which was used to test the proposed methods.

3. Test Results

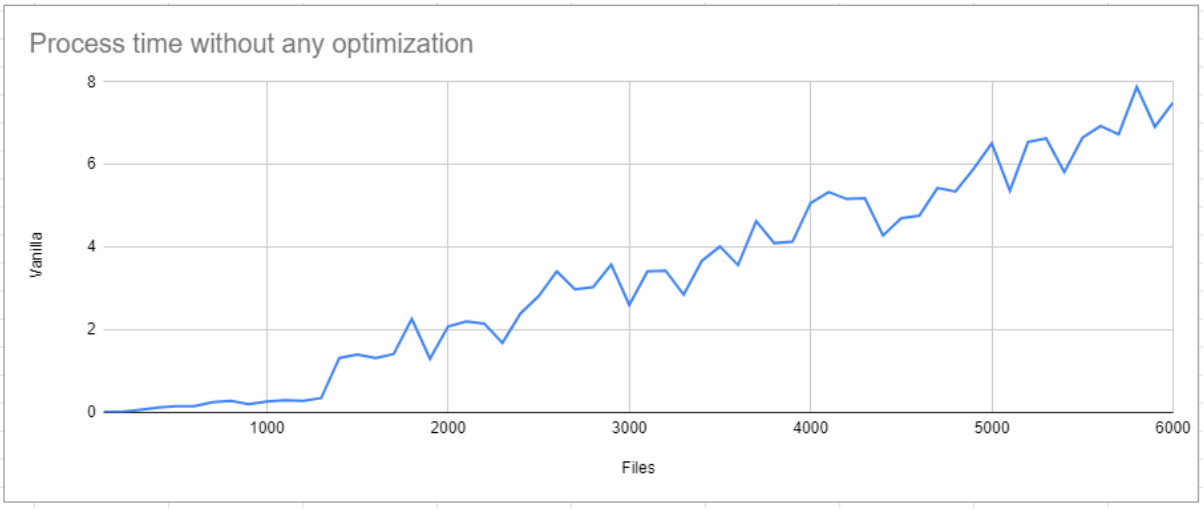
Our test results were in the magnitude of 100 as the difference between the times in files during the order of 10's and 1's was in the order of nanoseconds and the C was not capable of resolving the difference in the near nanoseconds. Hence, we had to record data in the order of milliseconds.

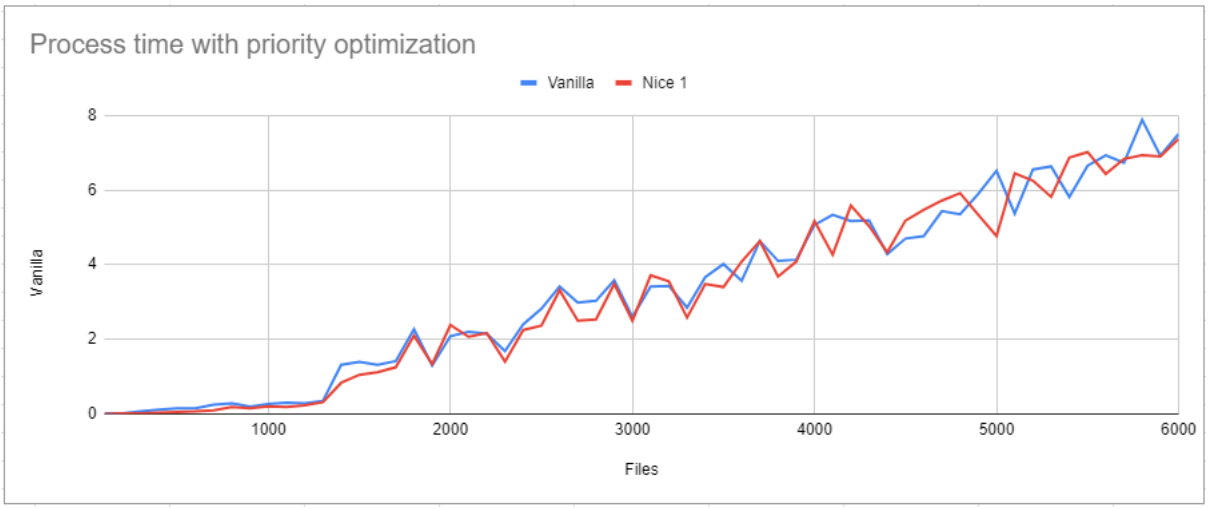
The table below shows the time in milliseconds.
It shows the subset of the test results for a range of 100 to 1500 files run on **Minix3.4**

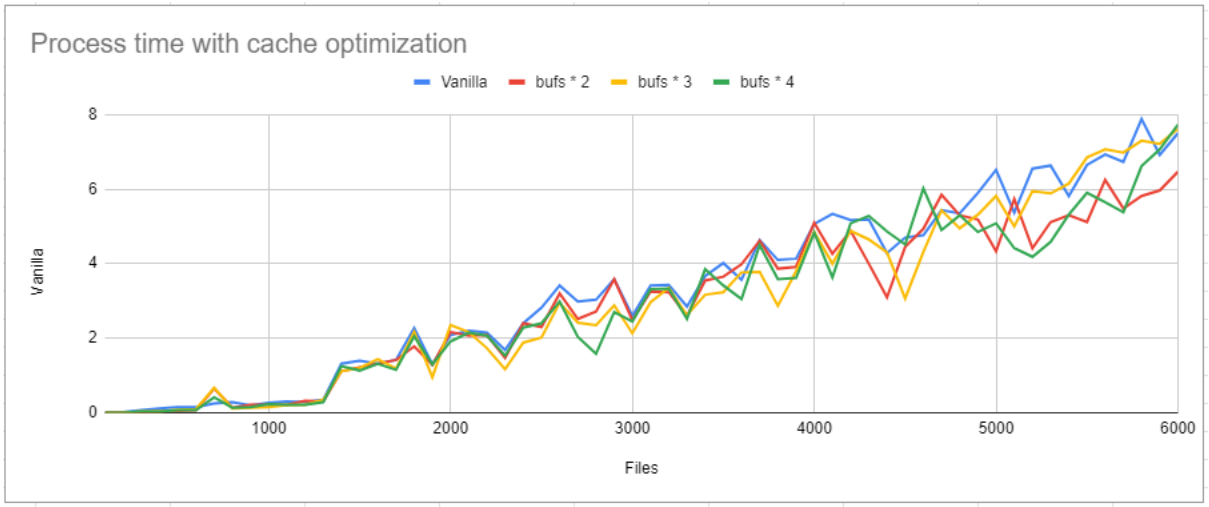
Files	Vanilla	Nice 1	Nice 2	bufs * 2	bufs * 3	bufs * 4	buf * 4 and nice
100	0.016675	0	0	0	0	0	0
200	0.016675	0.01665	0	0	0	0	0
300	0.06665	0.016675	0.01665	0.01665	0.033325	0.033325	0.016675
400	0.116675	0.033325	0.03335	0.03335	0.033325	0.03335	0.033325
500	0.15	0.05	0.05	0.05	0.08335	0.06665	0.05
600	0.15	0.066675	0.083325	0.06665	0.1	0.066675	0.066675
700	0.25	0.1	0.75	0.65	0.66665	0.416675	0.1
800	0.283325	0.183325	0.1	0.13335	0.116675	0.133325	0.1
900	0.2	0.15	0.133325	0.21665	0.133325	0.15	0.133325
1000	0.266675	0.2	0.216675	0.216675	0.15	0.233325	0.18335
1100	0.3	0.18335	0.25	0.216675	0.2	0.216675	0.25
1200	0.283325	0.233325	0.233325	0.31665	0.216675	0.216675	0.2
1300	0.35	0.316675	0.266675	0.3	0.35	0.283325	0.233325
1400	1.316675	0.833325	1.133325	1.116675	1.116675	1.25	1
1500	1.4	1.05	1.2	1.216675	1.2	1.133325	1.166675

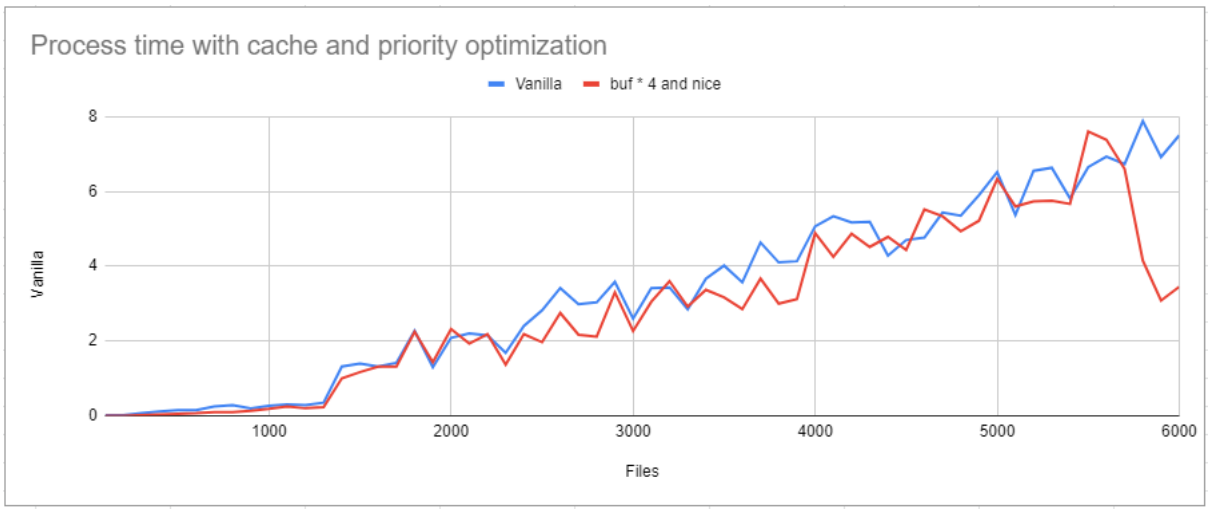
Header	Description
Files	The number of files on which the test was run
Vanilla	The initial test without any changes in the priority or cache
Nice 1	The nicerTo() call invoked in the process. Test 1
Nice 2	The nicerTo() call invoked in the process. Test 2
bufs * 2	Multiplied the current number of buffers by 2
bufs * 3	Multiplied the current number of buffers by 3
bufs * 4	Multiplied the current number of buffers by 4
buf * 4 and nice	Multiplied the current number of buffers by 5 and the nicerTo() invoked on the process

The collected data has been plotted and shown below









There is a lot of fluctuation in the data that has been collected and the degree of randomness must be associated with the difference in the access time in the cache. Additionally, there was a dip that was observed in the cache and priority optimization at 6000 files.

4. Inference from the Tests

The test results were predictable along with a little ambiguity as well.There was a lot of noise in the data collected but overall we observed that there was a steady increase in the time as there was an increase in the files that were being read and written. Additional inference from each of the optimizations that were implemented is discussed below.

a. nicerTo()

There was an initial dip in the time for the file operations, it was eventually caught up with the vanilla implementation and had no much performance difference to offer. The randomness as it is also observed without any optimization rules out the possibility that the priority optimization might have introduced any new randomness in the data collected during the test.

b. moreCache()

The increment in the number of blocks might not have introduced any visible improvement as the resolution of the data that was collected was high. The difference between the time was not much observed as the cache optimization by just increasing the number of buffers per block was not much noticeable.

5. Scope of improvement

Although it is already known that the data resolution for spawning a new process was much higher when we conducted Project 1. Hence, we were able to observe time differences in the order of seconds. Whereas now we are forking the process using the system call and generating a new child process to calculate the number of files that were read and written in one cycle.

A better approach of calculating the number of traps and messages passed as done in Project 2 might produce better results to observe the difference in the performance of the various calls. Further analyzing the traps and messages count along with the implementation of the analysis of the number of clock cycles might deduce better data for future implementation. The additional data that was collected during the experiments was not included in the report for brevity.