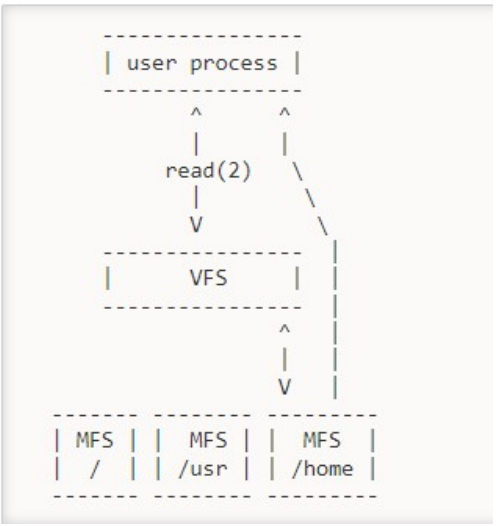
CS 551 Project 2

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

Submission from the **Team - 7**

1. Trace the Read system call in minix

In minix, the Virtual File System(VFS) implements the file system in coordination with one or more File Servers(FS). The File System reads the data on the disk and communicates with the VFS and the data is transmitted to the user. The role of the VFS is to handle most of the POSIX system calls that are supported by minix. read, write, etc. being some of the calls handled by the VFS for the OS. The VFS also coordinates with the other servers that are implemented on the os to smoothly handle the data transmission from the processes.



General overview of the read call handling

The brief description of the execution of the read system calls is described below.

1. The user process executes the read system call which is delivered to VFS.
2. VFS verifies the read is done on a valid (open) file and forwards the request to the FS responsible for the file system on which the file resides.
3. The FS reads the data, copies it directly to the user process, and replies to VFS it has executed the request.
4. Subsequently, VFS replies to the user process the operation is done and the user process continues to run.

Detailed description of the read call handling.

1. The user calling the read() function is essentially adhering to the POSIX standard and implements the wrapper function on top of the VFS read call to process it further.
2. The VFS Server first finds the flip object and the vnode specified by the file descriptor which is passed as a parameter to the read function. The read function implementations are in the file “minix/servers/vfs/read.c”
3. The VFS also checks whether the user process has the memory that it needs in order to perform the transfer.
4. Once the basic checks are done the VFS process will send a message to the FS process specified by the vnode.

5. The message sent by the VFS process is resolved by the relevant FS driver that is running for the mounted disk. All the various drivers and their relevant files are found in “minix/minix/fs”.
6. Finally in the “read.c” file in the relevant mounted file format, the “fs_readwrite()” function is invoked which performs the core read write operation in the minix os.
7. The fs_readwrite() function inturn finds the inode relevant to the file and reads the chunks from the disc and marks them to transfer the data to be read.
8. The VFS process receives the response message when the file is read without any errors and success value is returned to the user process.

2. Adding four system calls

The main minix repo has been accessed from the git.mini3.org, the latest development version available was **3.4.0** in the development of the project. Additionally, the documentation of the project was accessed and the system calls have been built accordingly. As the documentation available was from version **3.2.0** an additional modification of the files was required to implement the system calls.The detailed description of the individual files that have to be modified to implement the system calls are described in the **SYSTEM_CALLS.md** file which is attached with the project submission. A brief description of the process is discussed below.

1. **Implementation of a custom message**

A new message subtype has to be created to handle the message transferred between the processes when they are communicating to resolve the data in the message and the additional properties that have to be transferred. A message **mess_lsys_krn_sys_cs551** is implemented in the `minix/minix/include/minix/ipc.h`

2. **Implementation of the kernel call**

Implementation of a kernel call is quite complex as it requires modifying a lot of internal files which contain the core implementation of the minix os. Improper implementation of any of these files might result in the os not being able to compile and build. It is very important to keep a note of all the modifications that were done on the files to track and revert any improper changes in the kernel files. 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. 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_<*function_name*>”.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 `monitor.c`. The additional call numbers etc. are declared as required to support the message execution.

5. **Library implementation of the implemented 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.

Once the system calls are implemented and totally working, we can modify the core functions to increment the number of traps and messages.

Design for collecting the traps and messages

External variables are declared in the `minix/minix/kernel/glo.h` file to store the number of traps and messages that are collected during the execution of the tests.

```
/* CS 551 variables*/
EXTERN int trapcount;
EXTERN int msgcount;
```

Traps

A trap occurs when the kernel is trapped with an interrupt. The kernel as a whole is a singular system designed to support itself. The only entry point from the outside is a system call which is made, i.e., the kernel is trapped with an interrupt. Hence, we have decided the `void kernel_call()` method in the `` `minix/minix/kernel/system.c` `` file is the ideal place to set the incrementer and visualize the number of times the kernel is trapped. The trapcount is incremented when the kernel call is invoked every time.

```
/* CS 551 Project 2 logic*/
trapcount = (trapcount<0 || trapcount>(INT_MAX-1))? 0 : (trapcount+1);
```

Messages

All the calls in the minix system are passed as messages between the various processes and servers in the operating system. Most of the messages are processed by the `static int do_sync_ipc()` in the `minix/minix/kernel/proc.c` . The msgcount is incremented in the ipc call handler whenever the function is invoked.

```
/* CS 551 Project 2 logic*/
msgcount = (msgcount<0 || msgcount > (INT_MAX-1)) ? 0 : (msgcount+1);
```

Using the implemented Calls

With all the functions implemented and the kernel and the os compiled, we can call the four functions just by including the `#include <cs551.h>` header file in our c and cpp programs. The four functions implemented are as follows.

void inittrapcounter ()

This function resets the trapcount to zero in the kernel.

int trapcounter ()

This function returns the current number of traps that were encountered by the kernel.

void initmsgcounter ()

This function resets the msgcount to zero in the kernel.

int msgcounter ()
This function returns the current number of messages that were transmitted in the operating system.

3. Configuration of the Test bench

The minix operating systems wasinstalled 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 threads with the read and write system calls and the number of traps and the messages were noted and the data was processed to derive the understanding of the system calls. The code snippet run on the os was compiled by clang.

4. Test Results

The read and write system calls were subjected to the counting of the traps and the messages during the system call invocation. The example of the read system call being invoked after clearing the trap and message values. The values are then collected after the completion of the read call. The code for the read call is shown below.

```
void fileReadTask (char filePath[], bool output) {

    if(output) printf("reading file: %s \n", filePath);

    int f1 = open(filePath, O_RDONLY, 0644);
    if (f1 < 0) {
        if(output) printf("Error reading from file: %s : %d \n", filePath, f1);
        exit(1);
    }
    char *c = (char *) calloc((5*1024), sizeof(char));
    inittrapcounter();
    initmsgcounter();
    int sz = read(f1, c, (5*1024));
    if(output) printf("Read system call \nTraps: %d, Messages: %d \n", trapcounter(), msgcounter());
    if (close(f1) < 0) {
        printf("Error reading from file: %s \n", filePath);
        exit(1);
    }

    if(output) printf("File read successful: %s \n", filePath);

}
```

Similarly for the write system call as well the traps and the messages were collected for a file during the processing of the write system call.

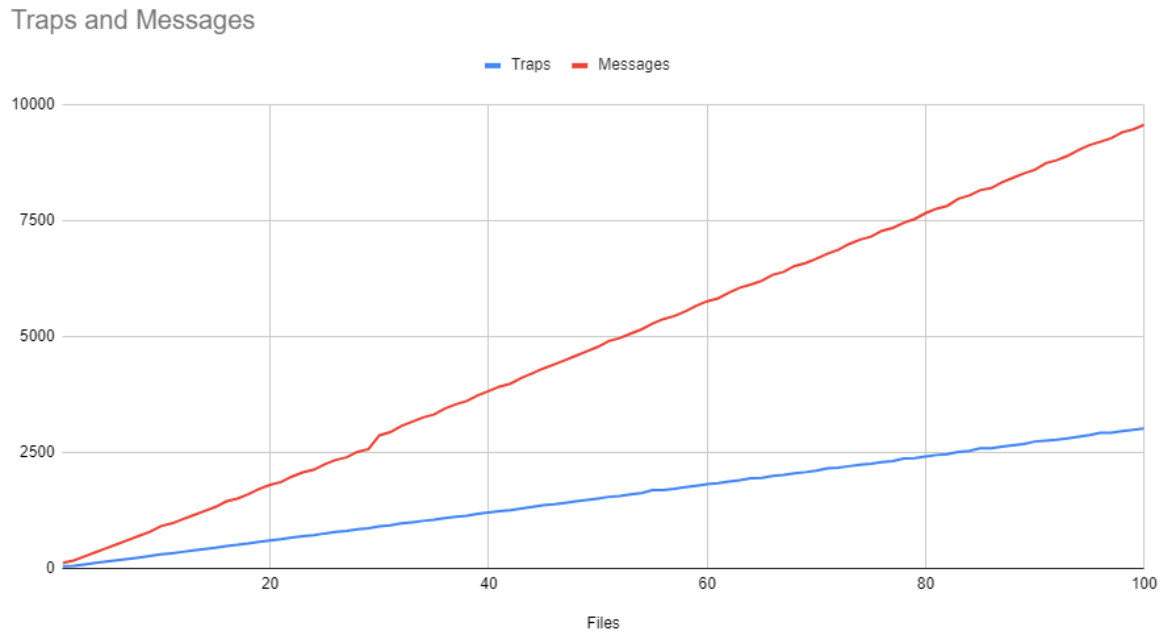
```
void fileWriteTask (char filePath[], bool output) {

    if(output) printf("writing file: %s \n", filePath);

    int f1 = open(filePath, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (f1 < 0) {
        if(output) printf("Error writing to file: %s : %d \n", filePath, f1);
        exit(1);
    }
    inittrapcounter();
    initmsgcounter();
    int sz = write(f1, "a ", (5*1024));
    if(output) printf("Write system call \nTraps: %d, Messages: %d \n", trapcounter(), msgcounter());
    if (close(f1) < 0) {
        printf("Error writing to file: %s \n", filePath);
        exit(1);
    }

    if(output) printf("File write successful: %s \n", filePath);

}
```

For a single read and write system call, the number of traps and messages that were collected are as follows.

```
minix#
minix# clang file_read_write.c
minix# ./a.out

writing file: ./5k_test/file0.txt
Write system call
Traps: 6, Messages: 24
File write successful: ./5k_test/file0.txt

reading file: ./5k_test/file0.txt
Read system call
Traps: 4, Messages: 12
File read successful: ./5k_test/file0.txt

minix# _
```

Note: We had run the tests with multiple files concurrently but as there was no new data to present the data has not been included in the report for brevity.

A total count on the number of traps and messages for all the system calls were run during the file read and write process and the data for 100 files have been collected and presented below.



The data collected from the same experiment during the 10 file run has been tabulated below.

Files	Traps	Messages
1	57	127
2	63	175
3	93	264
4	124	354
5	153	442

6	183	531
7	214	620
8	243	709
9	273	798
10	309	918

5. Inference from the tests

We have also run the test with multiple files but ideally the number of traps and the messages during the read and write system calls were constant. This shows the calls are standard and are similar while reading any number of files. Additionally, it is also noted that the traps and messages are more in number during the write system call rather than the read system call. There is further scope of analyzing the results and probing it further to resolve the messages that are being passed and the traps that are encountered in the process.

6. Improving the system read and write system call

As we have observed from the tests there are a lot of traps and messages being passed when the read or write system call is invoked. The graph is linear in the multiple file test runs which denote the file calls being called again and again with similar time intervals in order to perform the same task. This calls for some upgrades in the methodology of the system calls being implemented in the kernel level for performance enhancement. Two such methods are discussed below.

1. **Caching**
   Caching is one of the most useful methods in modern programming. Although a file system cache is implemented, we can also implement a layer of cache at the VFS server which increases the efficiency of the VFS as most of the user level operations are performed by it.
2. **Lookahead mechanism**
   We can also implement a look ahead mechanism which is generally implemented in the processors to track the most frequently used registers and simplify the wait time in the most frequently accessed system calls and improve the performance of the individual processes. This in turn gives us the ability to reduce the number of context switches and increase the throughput of the system.

We have just discussed a couple of methods which would be helpful in improving the efficiency of the minix system, further analysis of the kernel code would help us understand the relevance of the methods that are implemented currently.

7. Pitfalls and additional updates

Although our initial submission was in a cpp file run on the MINIX v**3.3.0**, we had to change our files and calls to support the v**3.4.0**. The additional issue that we had run into was the system calls were having linker errors while being compiled with **clang++**, especially the C++ files. We are currently looking at fixing the issue and making the libraries available for C++ files to work as well.