EE516 : Project 1

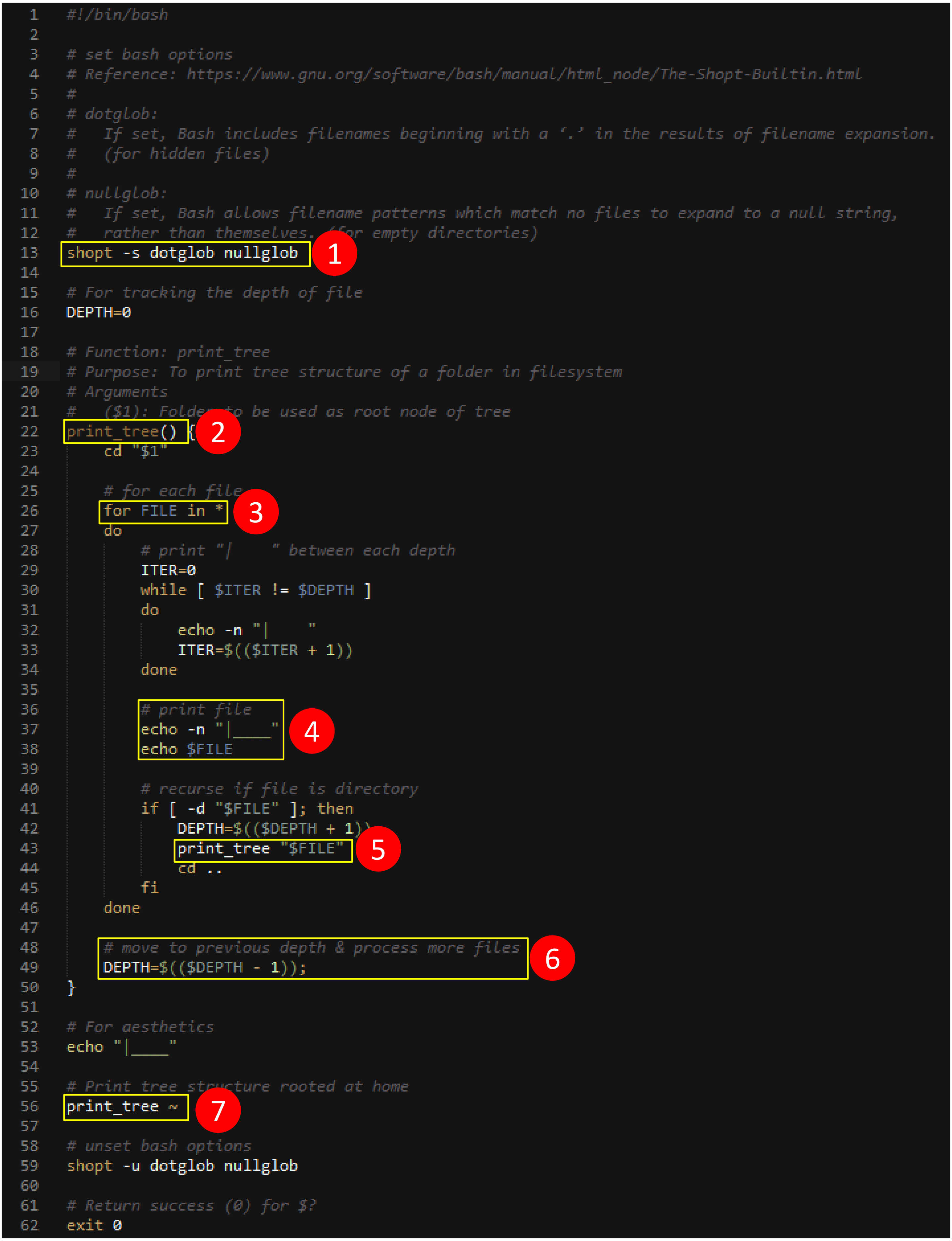
Gaurav Kalra

(Student ID: 2016 45 93)

gvkalra@kaist.ac.kr

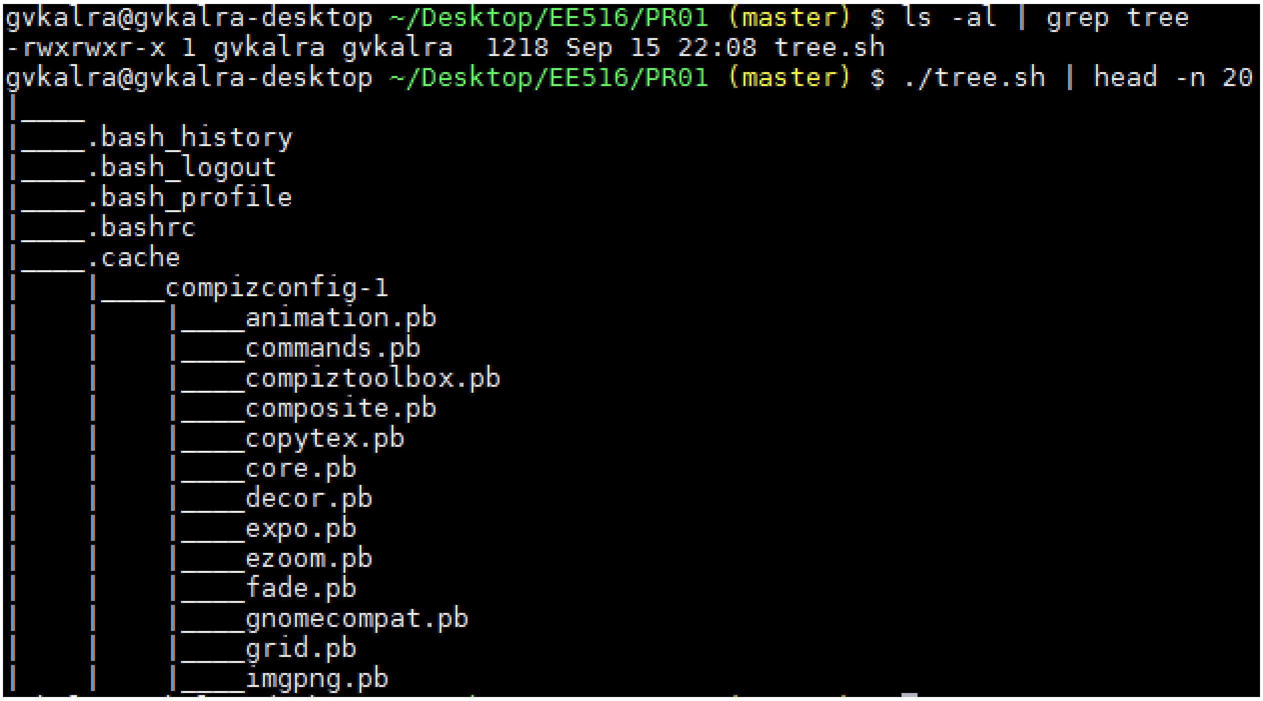
Task 1: Tree Script

Write a bash shell script that shows tree structure of the directories and files included in your own home directory.



```bash
#!/bin/bash

# set bash options
# Reference: https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html
#
# dotglob:
#   If set, Bash includes filenames beginning with a ‘.’ in the results of filename expansion.
#   (for hidden files)
#
# nullglob:
#   If set, Bash allows filename patterns which match no files to expand to a null string,
#   rather than themselves. (for empty directories)
shopt -s dotglob nullglob

# For tracking the depth of file
DEPTH=0

# Function: print_tree
# Purpose: To print tree structure of a folder in filesystem
# Arguments
#   ($1): Folder to be used as root node of tree
print_tree() {
    cd "$1"

    # for each file
    for FILE in *
    do
        # print "|     " between each depth
        ITER=0
        while [ $ITER != $DEPTH ]
        do
            echo -n "|     "
            ITER=$(($ITER + 1))
        done

        # print file
        echo -n "|____"
        echo $FILE

        # recurse if file is directory
        if [ -d "$FILE" ]; then
            DEPTH=$(($DEPTH + 1))
            print_tree "$FILE"
            cd ..
        fi
    done

    # move to previous depth & process more files
    DEPTH=$(($DEPTH - 1));
}

# For aesthetics
echo "|____"

# Print tree structure rooted at home
print_tree ~

# unset bash options
shopt -u dotglob nullglob

# Return success (0) for $?
exit 0
```

Figure 1: Source code of Tree Script with Key Points highlighted



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ls -al | grep tree
-rwxrwxr-x 1 gvkalra gvkalra  1218 Sep 15 22:08 tree.sh
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ./tree.sh | head -n 20
|____
|____.bash_history
|____.bash_logout
|____.bash_profile
|____.bashrc
|____.cache
|    |____compizconfig-1
|    |    |____animation.pb
|    |    |____commands.pb
|    |    |____compiztoolbox.pb
|    |    |____composite.pb
|    |    |____copytex.pb
|    |    |____core.pb
|    |    |____decor.pb
|    |    |____expo.pb
|    |    |____ezoom.pb
|    |    |____fade.pb
|    |    |____gnomecompat.pb
|    |    |____grid.pb
|    |    |____imgpng.pb
```

Figure 2: Sample output of executing Tree Script stripped to 20 lines

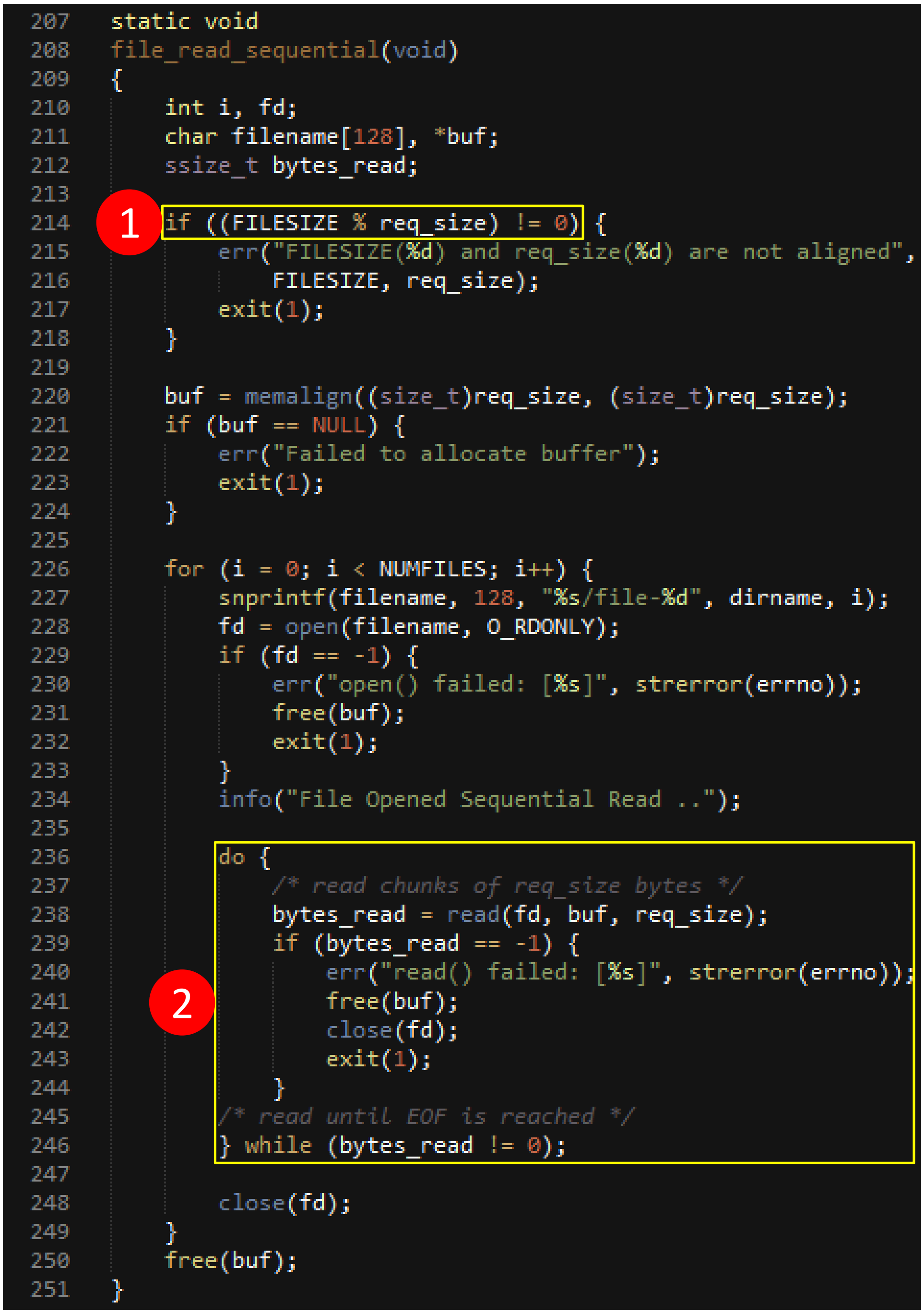
The core of *tree.sh* is implemented as a recursive function *print_tree()*, which keeps track of the depth of a sub-directory using *DEPTH* variable.

Bash options *dotglob* and *nullglob* are necessary to be set for enabling *for FILE in ** loop iterate on hidden files & reporting contents of an empty directory as null string respectively.

It is also important to double quote *$1* and *$FILE* since the filenames may contain spaces, in which case *print_tree()* will treat one filename as multiple arguments.

Task 2: Simple C Program

Implement Sequential Read, Random Write, Random Read for File System Benchmarking

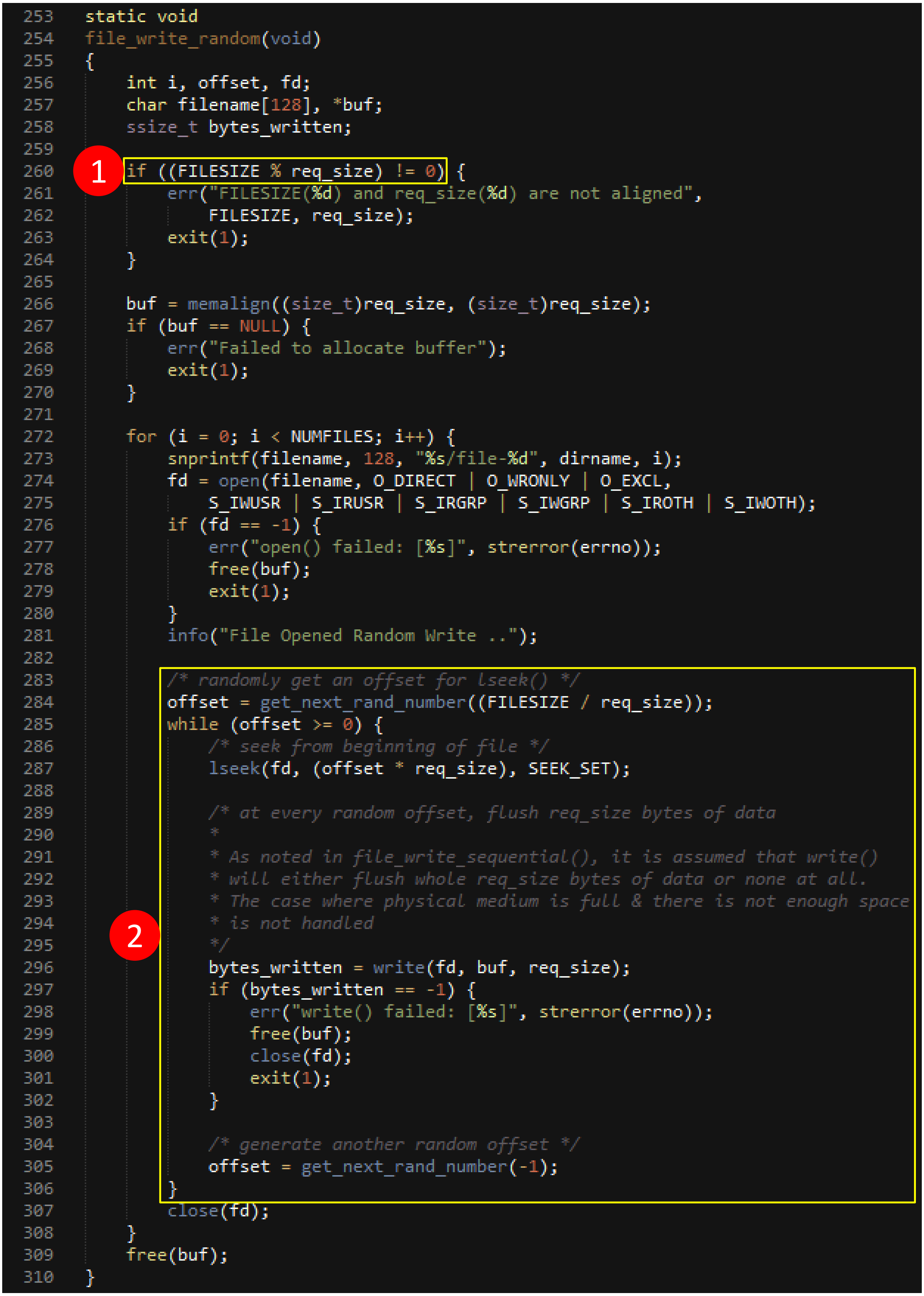


```c
static void
file_read_sequential(void)
{
    int i, fd;
    char filename[128], *buf;
    ssize_t bytes_read;

    if ((FILESIZE % req_size) != 0) {
        err("FILESIZE(%d) and req_size(%d) are not aligned",
            FILESIZE, req_size);
        exit(1);
    }

    buf = memalign((size_t)req_size, (size_t)req_size);
    if (buf == NULL) {
        err("Failed to allocate buffer");
        exit(1);
    }

    for (i = 0; i < NUMFILES; i++) {
        snprintf(filename, 128, "%s/file-%d", dirname, i);
        fd = open(filename, O_RDONLY);
        if (fd == -1) {
            err("open() failed: [%s]", strerror(errno));
            free(buf);
            exit(1);
        }
        info("File Opened Sequential Read ..");

        do {
            /* read chunks of req_size bytes */
            bytes_read = read(fd, buf, req_size);
            if (bytes_read == -1) {
                err("read() failed: [%s]", strerror(errno));
                free(buf);
                close(fd);
                exit(1);
            }
        /* read until EOF is reached */
        } while (bytes_read != 0);

        close(fd);
    }
    free(buf);
}
```

Figure 3: Source code of Sequential Read with Key Points highlighted

The check for *FILESIZE % req_size* to be zero is added as a sanity measure for invalid inputs.

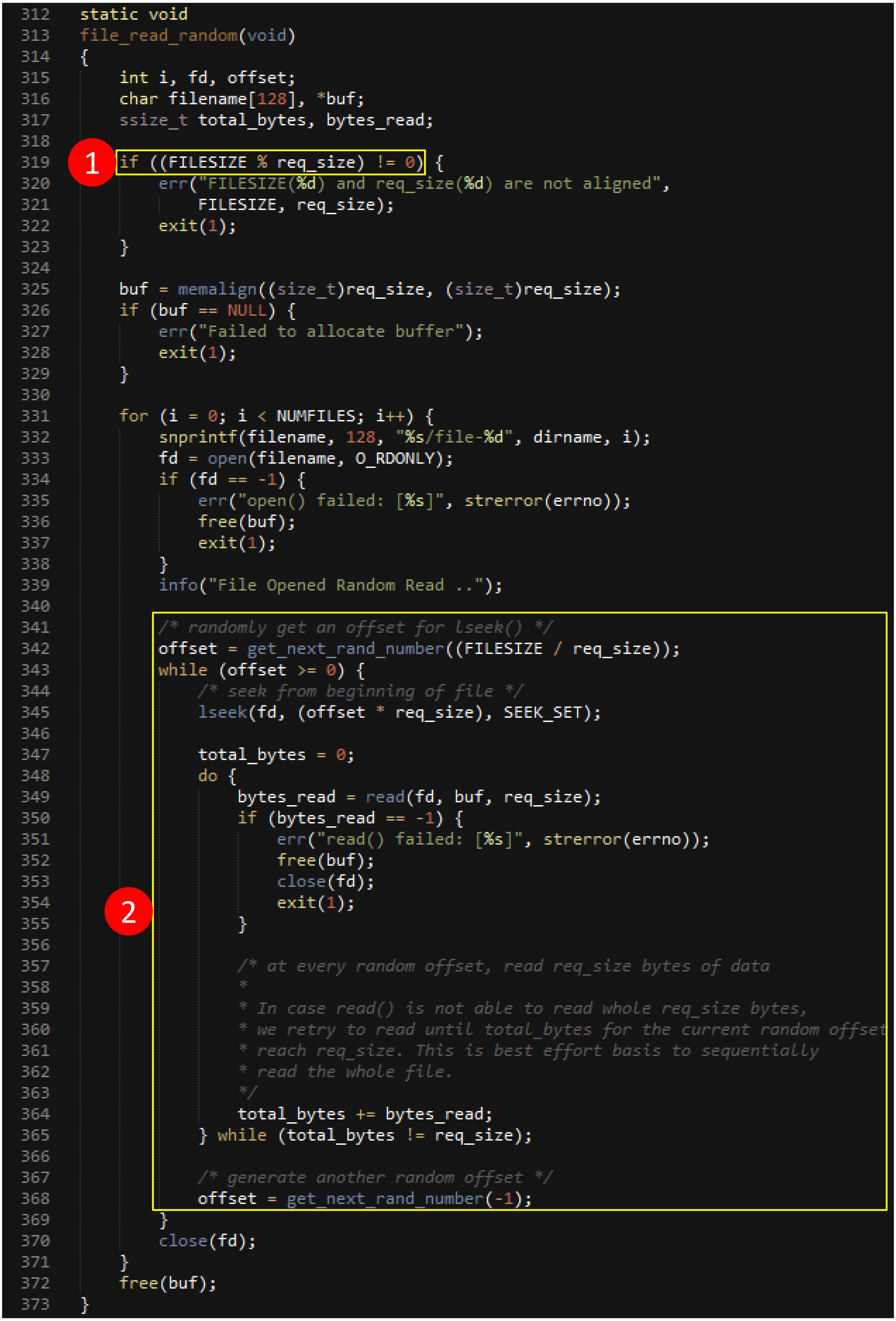
For sequential *read()*, I try to read *req_size* bytes of data in a while loop until EOF is reached.



```c
static void
file_write_random(void)
{
    int i, offset, fd;
    char filename[128], *buf;
    ssize_t bytes_written;

    if ((FILESIZE % req_size) != 0) {
        err("FILESIZE(%d) and req_size(%d) are not aligned",
            FILESIZE, req_size);
        exit(1);
    }

    buf = memalign((size_t)req_size, (size_t)req_size);
    if (buf == NULL) {
        err("Failed to allocate buffer");
        exit(1);
    }

    for (i = 0; i < NUMFILES; i++) {
        snprintf(filename, 128, "%s/file-%d", dirname, i);
        fd = open(filename, O_DIRECT | O_WRONLY | O_EXCL,
            S_IWUSR | S_IRUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
        if (fd == -1) {
            err("open() failed: [%s]", strerror(errno));
            free(buf);
            exit(1);
        }
        info("File Opened Random Write ..");

        /* randomly get an offset for lseek() */
        offset = get_next_rand_number((FILESIZE / req_size));
        while (offset >= 0) {
            /* seek from beginning of file */
            lseek(fd, (offset * req_size), SEEK_SET);

            /* at every random offset, flush req_size bytes of data
            *
            * As noted in file_write_sequential(), it is assumed that write()
            * will either flush whole req_size bytes of data or none at all.
            * The case where physical medium is full & there is not enough space
            * is not handled
            */
            bytes_written = write(fd, buf, req_size);
            if (bytes_written == -1) {
                err("write() failed: [%s]", strerror(errno));
                free(buf);
                close(fd);
                exit(1);
            }

            /* generate another random offset */
            offset = get_next_rand_number(-1);
        }
        close(fd);
    }
    free(buf);
}
```

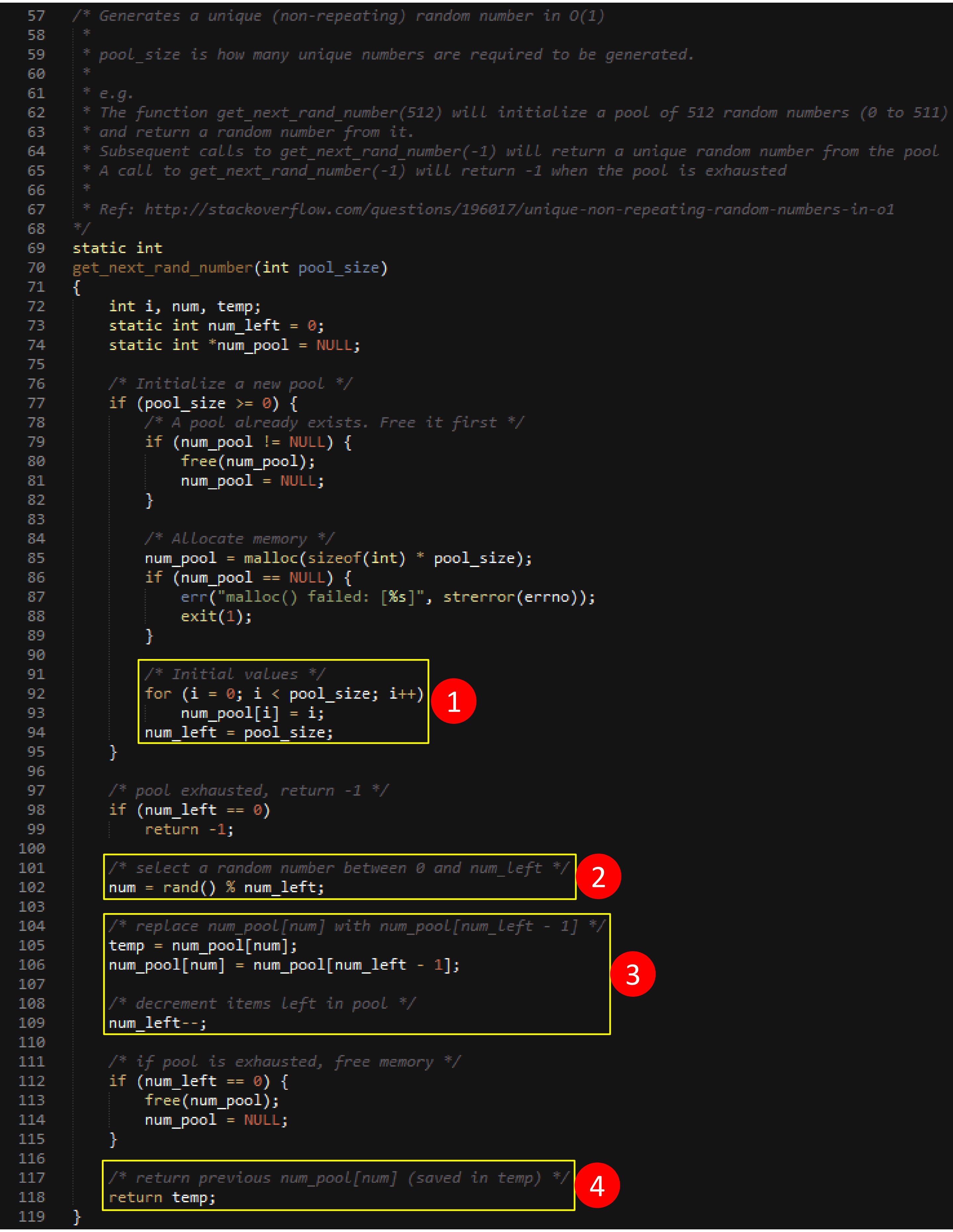
Figure 4: Source code of Random Write with Key Points highlighted

For random *write()*, I generate a unique random number, *lseek()* to the random offset & try to write *req_size* bytes of data until random number pool is exhausted (size of file becomes *FILESIZE*).



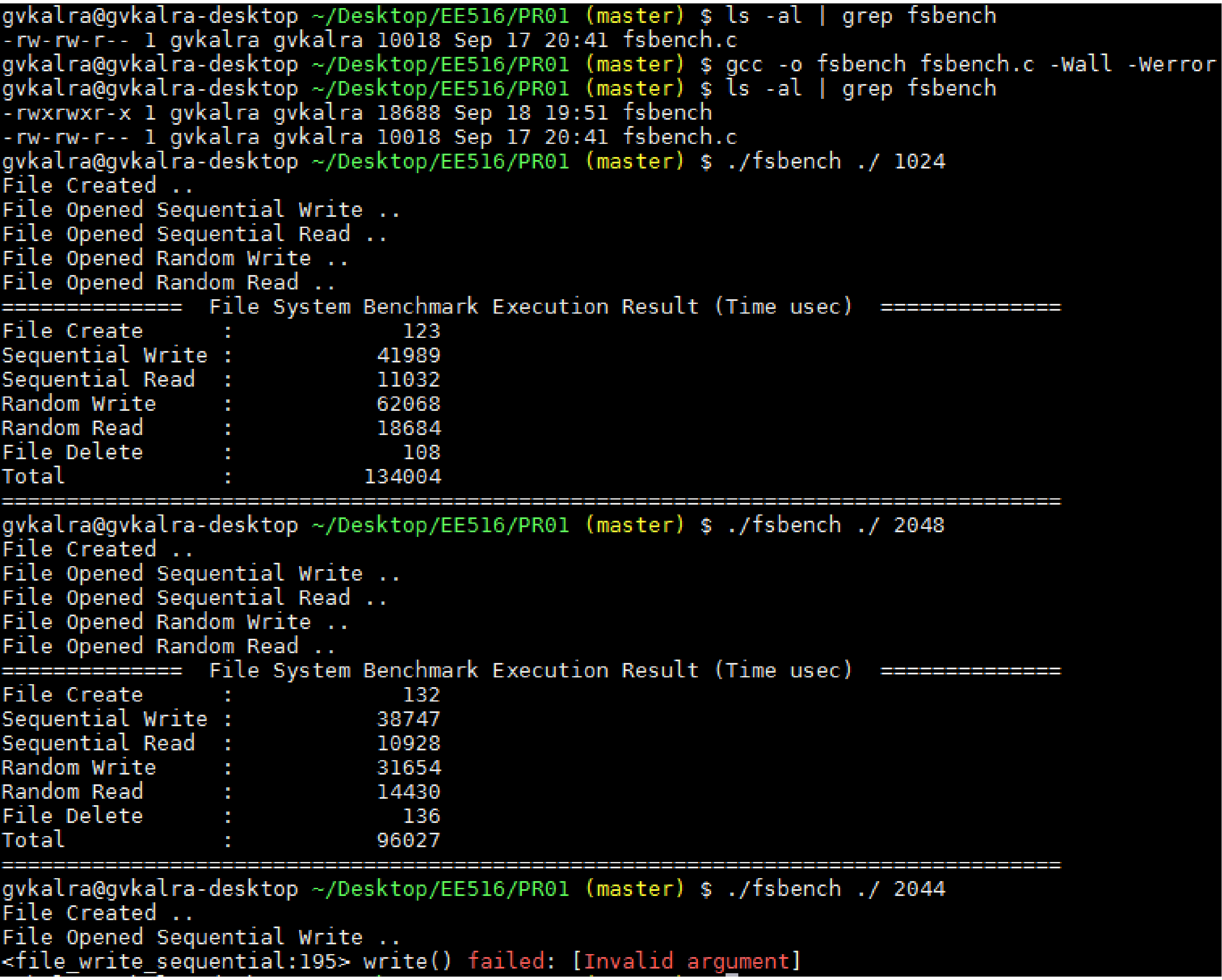
```c
static void
file_read_random(void)
{
    int i, fd, offset;
    char filename[128], *buf;
    ssize_t total_bytes, bytes_read;

    if ((FILESIZE % req_size) != 0) {
        err("FILESIZE(%d) and req_size(%d) are not aligned",
            FILESIZE, req_size);
        exit(1);
    }

    buf = memalign((size_t)req_size, (size_t)req_size);
    if (buf == NULL) {
        err("Failed to allocate buffer");
        exit(1);
    }

    for (i = 0; i < NUMFILES; i++) {
        snprintf(filename, 128, "%s/file-%d", dirname, i);
        fd = open(filename, O_RDONLY);
        if (fd == -1) {
            err("open() failed: [%s]", strerror(errno));
            free(buf);
            exit(1);
        }
        info("File Opened Random Read ..");

        /* randomly get an offset for lseek() */
        offset = get_next_rand_number((FILESIZE / req_size));
        while (offset >= 0) {
            /* seek from beginning of file */
            lseek(fd, (offset * req_size), SEEK_SET);

            total_bytes = 0;
            do {
                bytes_read = read(fd, buf, req_size);
                if (bytes_read == -1) {
                    err("read() failed: [%s]", strerror(errno));
                    free(buf);
                    close(fd);
                    exit(1);
                }

                /* at every random offset, read req_size bytes of data
                 *
                 * In case read() is not able to read whole req_size bytes,
                 * we retry to read until total_bytes for the current random offset
                 * reach req_size. This is best effort basis to sequentially
                 * read the whole file.
                 */
                total_bytes += bytes_read;
            } while (total_bytes != req_size);

            /* generate another random offset */
            offset = get_next_rand_number(-1);
        }
        close(fd);
    }
    free(buf);
}
```

Figure 5: Source code of Random Read with Key Points highlighted



```c
/* Generates a unique (non-repeating) random number in O(1)
 *
 * pool_size is how many unique numbers are required to be generated.
 *
 * e.g.
 * The function get_next_rand_number(512) will initialize a pool of 512 random numbers (0 to 511)
 * and return a random number from it.
 * Subsequent calls to get_next_rand_number(-1) will return a unique random number from the pool
 * A call to get_next_rand_number(-1) will return -1 when the pool is exhausted
 *
 * Ref: http://stackoverflow.com/questions/196017/unique-non-repeating-random-numbers-in-o1
*/
static int
get_next_rand_number(int pool_size)
{
    int i, num, temp;
    static int num_left = 0;
    static int *num_pool = NULL;

    /* Initialize a new pool */
    if (pool_size >= 0) {
        /* A pool already exists. Free it first */
        if (num_pool != NULL) {
            free(num_pool);
            num_pool = NULL;
        }

        /* Allocate memory */
        num_pool = malloc(sizeof(int) * pool_size);
        if (num_pool == NULL) {
            err("malloc() failed: [%s]", strerror(errno));
            exit(1);
        }

        /* Initial values */
        for (i = 0; i < pool_size; i++)
            num_pool[i] = i;
        num_left = pool_size;
    }

    /* pool exhausted, return -1 */
    if (num_left == 0)
        return -1;

    /* select a random number between 0 and num_left */
    num = rand() % num_left;

    /* replace num_pool[num] with num_pool[num_left - 1] */
    temp = num_pool[num];
    num_pool[num] = num_pool[num_left - 1];

    /* decrement items left in pool */
    num_left--;

    /* if pool is exhausted, free memory */
    if (num_left == 0) {
        free(num_pool);
        num_pool = NULL;
    }

    /* return previous num_pool[num] (saved in temp) */
    return temp;
}
```

Figure 6: Random Number Generator with Key Points highlighted

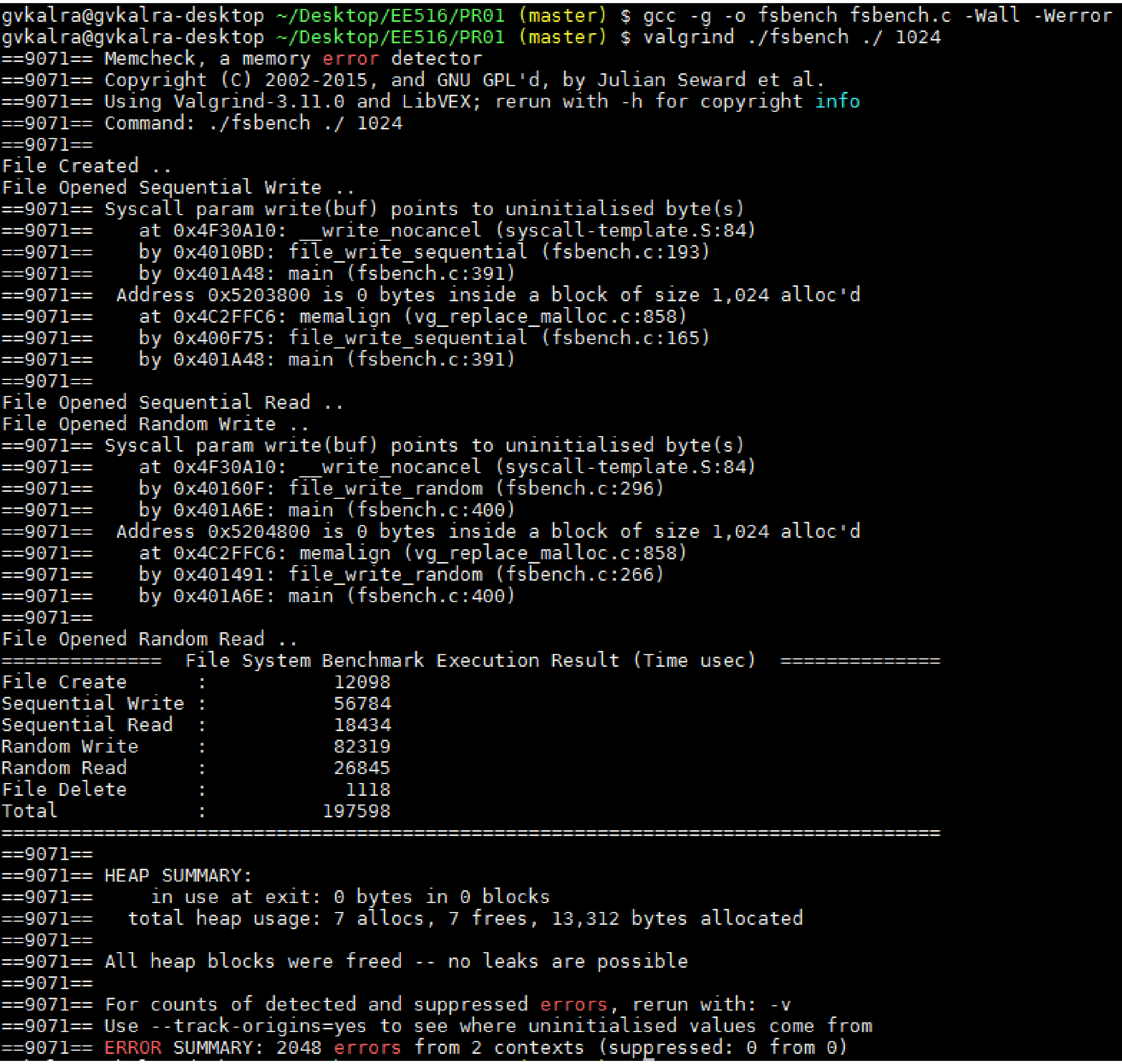


```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ls -al | grep fsbench
-rw-rw-r-- 1 gvkalra gvkalra 10018 Sep 17 20:41 fsbench.c
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ gcc -o fsbench fsbench.c -Wall -Werror
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ls -al | grep fsbench
-rwxrwxr-x 1 gvkalra gvkalra 18688 Sep 18 19:51 fsbench
-rw-rw-r-- 1 gvkalra gvkalra 10018 Sep 17 20:41 fsbench.c
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ./fsbench ./ 1024
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
File Opened Random Write ..
File Opened Random Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :           123
Sequential Write  :         41989
Sequential Read   :         11032
Random Write      :         62068
Random Read       :         18684
File Delete       :           108
Total             :        134004
==================================================================================
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ./fsbench ./ 2048
File Created ..
File Opened Sequential Write ..
File Opened Sequential Read ..
File Opened Random Write ..
File Opened Random Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create       :           132
Sequential Write  :         38747
Sequential Read   :         10928
Random Write      :         31654
Random Read       :         14430
File Delete       :           136
Total             :         96027
==================================================================================
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ ./fsbench ./ 2044
File Created ..
File Opened Sequential Write ..
<file_write_sequential:195> write() failed: [Invalid argument]
```

Figure 7: Output of compiling & executing File System benchmark

Please notice that *./fsbench ./ 2044* fails because current file offset, buffer, bytes to be written are not aligned in *file_write_sequential()*. Subsequent executions of fsbench will as well fail until “*rm -f file-**” is executed in the current directory. This is a known & easy to fix bug. However, it is outside the purview of current assignment.

Notice that there are no warnings (source is compiled with *-Wall -Werror* flags).



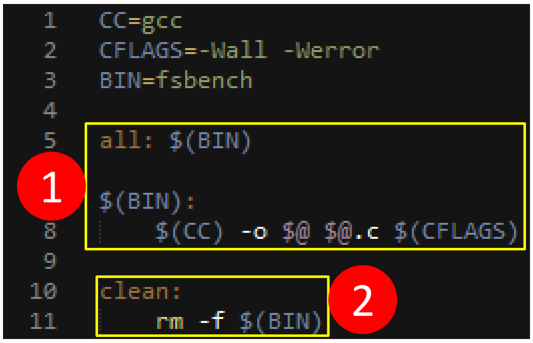
```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ gcc -g -o fsbench fsbench.c -Wall -Werror
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ valgrind ./fsbench ./ 1024
==9071== Memcheck, a memory error detector
==9071== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==9071== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==9071== Command: ./fsbench ./ 1024
==9071==
File Created ..
File Opened Sequential Write ..
==9071== Syscall param write(buf) points to uninitialised byte(s)
==9071==    at 0x4F30A10: __write_nocancel (syscall-template.S:84)
==9071==    by 0x4010BD: file_write_sequential (fsbench.c:193)
==9071==    by 0x401A48: main (fsbench.c:391)
==9071==  Address 0x5203800 is 0 bytes inside a block of size 1,024 alloc'd
==9071==    at 0x4C2FFC6: memalign (vg_replace_malloc.c:858)
==9071==    by 0x400F75: file_write_sequential (fsbench.c:165)
==9071==    by 0x401A48: main (fsbench.c:391)
==9071==
File Opened Sequential Read ..
File Opened Random Write ..
==9071== Syscall param write(buf) points to uninitialised byte(s)
==9071==    at 0x4F30A10: __write_nocancel (syscall-template.S:84)
==9071==    by 0x40160F: file_write_random (fsbench.c:296)
==9071==    by 0x401A6E: main (fsbench.c:400)
==9071==  Address 0x5204800 is 0 bytes inside a block of size 1,024 alloc'd
==9071==    at 0x4C2FFC6: memalign (vg_replace_malloc.c:858)
==9071==    by 0x401491: file_write_random (fsbench.c:266)
==9071==    by 0x401A6E: main (fsbench.c:400)
==9071==
File Opened Random Read ..
==============  File System Benchmark Execution Result (Time usec)  ==============
File Create      :           12098
Sequential Write :           56784
Sequential Read  :           18434
Random Write     :           82319
Random Read      :           26845
File Delete      :            1118
Total            :          197598
==================================================================================
==9071==
==9071== HEAP SUMMARY:
==9071==     in use at exit: 0 bytes in 0 blocks
==9071==   total heap usage: 7 allocs, 7 frees, 13,312 bytes allocated
==9071==
==9071== All heap blocks were freed -- no leaks are possible
==9071==
==9071== For counts of detected and suppressed errors, rerun with: -v
==9071== Use --track-origins=yes to see where uninitialised values come from
==9071== ERROR SUMMARY: 2048 errors from 2 contexts (suppressed: 0 from 0)
```

Figure 8: Output of compiling & executing File System benchmark with valgrind for checking heap memory leaks

Please notice “HEAP SUMMARY” which says “All heap blocks were freed – no leaks are possible”

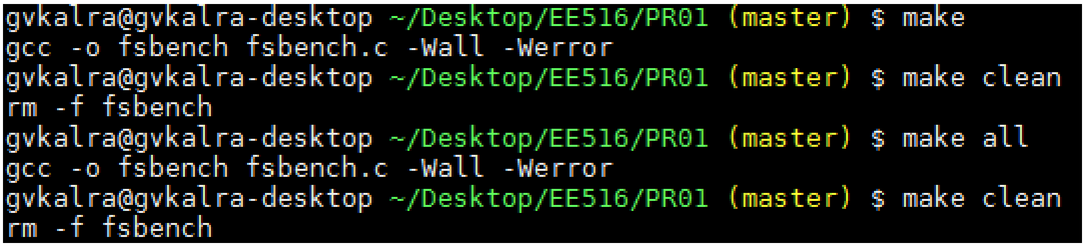
Task 3: Makefile

Write a Makefile for the source code of Task 2



```
CC=gcc
CFLAGS=-Wall -Werror
BIN=fsbench

all: $(BIN)

$(BIN):
	$(CC) -o $@ $@.c $(CFLAGS)

clean:
	rm -f $(BIN)
```

Figure 9: Makefile of Task 2 with Key Points highlighted



```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ make
gcc -o fsbench fsbench.c -Wall -Werror
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ make clean
rm -f fsbench
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ make all
gcc -o fsbench fsbench.c -Wall -Werror
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR01 (master) $ make clean
rm -f fsbench
```

Figure 10: Sample output of Makefile targets