

Note: for test 4, I accidently didn’t change the int to char for the value at map[6009]. It has been corrected in the code attached and in my code.

Explanations for 4&5:

The output from experiment 4 was expected since the stat(2) system call uses the file descriptor. In this scenario, the file size did not change. The kernel allocated two pages for our manipulation (through mmap) which doesn’t affect the original file size.

The output from experiment 5 was that the value of the byte that I wrote. This is because when we lseek length + 16, there is still more space left before the end of the second page (I did 6009 for length, so less than 2 pages but more than 1 page). This would set more zeros until length + 16. And when reading from the allocated space + 1, we would see the value that we declared.

Explanation for 6:

When the process tries to go to the second page given the initial length was allocated to less than the size of a page, the kernel cannot satisfy the demand because the two pages were not allocated within the Page Global Directory. In this scenario, the kernel will give an error on the return value of mmap.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <time.h>

#include <signal.h>

void sig\_handler(int sig){

fprintf(stderr, "Signal %s received!\n", strsignal(sig));

exit(sig);

}

void error\_message(const char \*message, const char \*error\_val){

if(!error\_val)

fprintf(stderr, "%s\n", message);

else

fprintf(stderr, "%s Error value: %s.\n", message, error\_val);

}

int create\_fd(int length){

int fd;

char buf[1];

if((fd = open("file.txt", O\_RDWR|O\_CREAT|O\_TRUNC, 0666)) < 0){

error\_message("ERROR: could not open properly", strerror(errno));

exit(255);

}

srand(time(NULL));

for(int i = 0; i < length; i++){

if((rand() % 36) <= 10){

sprintf(buf, "%i", rand()%10);

if(write(fd, buf, 1) < 0){

error\_message("ERROR: could not write properly", strerror(errno));

exit(255);

}

}

else{

buf[0] = rand()%26 + 'A';

if(write(fd, buf, 1) < 0){

error\_message("ERROR: could not write properly", strerror(errno));

exit(255);

}

}

}

return fd;

}

void map\_close\_error(int length, int fd, char \* map){

if(munmap(map, length) < 0){

close(fd);

error\_message("ERROR: could not munmap file from memory.", strerror(errno));

exit(255);

}

if(close(fd) < 0){

error\_message("ERROR: could not close file descriptor.", strerror(errno));

exit(255);

}

}

int test\_1(){

char \*map;

size\_t length = 4096;

printf("Executing Test #1 (write to r/o mmap): \n");

int fd = create\_fd(length);

if((map = mmap(NULL, length, PROT\_READ, MAP\_SHARED, fd, 0)) == MAP\_FAILED){

error\_message("ERROR: could not mmap file from memory.", strerror(errno));

exit(255);

}

printf("map[3] = \'%c\'\nwriting a \'%c\'\n", map[3], map[4]);

map[3] = map[4];

printf("After write, map[3] returned \'%c\'\n", map[3]);

map\_close\_error(length, fd, map);

exit(0);

}

int test\_23(int flag){

char \*map;

size\_t length = 4096;

char buf[1];

char \*out\_flag;

int private, offset = 6;

if(flag == MAP\_SHARED){

printf("Executing Test #2 (writing to MAP\_SHARED): \n");

out\_flag = "MAP\_SHARED";

}

else if(flag == MAP\_PRIVATE){

printf("Executing Test #3 (writing to MAP\_PRIVATE): \n");

out\_flag = "MAP\_PRIVATE";

}

else

error\_message("ERROR: invalid input for 2 or 3.", strerror(errno));

int fd = create\_fd(length);

if((map = mmap(NULL, length, PROT\_WRITE, flag, fd, 0)) == MAP\_FAILED){

error\_message("ERROR: could not mmap file from memory.", strerror(errno));

exit(255);

}

printf("map[%d] = \'%c\'\nwriting a new value: J\n", offset, map[offset]);

map[offset] = 'J';

if(lseek(fd, offset, SEEK\_SET) < 0)

error\_message("ERROR: could not lseek properly", strerror(errno));

if(read(fd, buf, 1) < 0)

error\_message("ERROR: could not read properly", strerror(errno));

if(buf[0] == 'J')

private = 0;

else

private = 1;

printf("Updated map[%d] with \'%c\' for %s. Map %s private.\n", offset, map[offset], out\_flag, private ? "is": "is not");

map\_close\_error(length, fd, map);

private ? exit(0) : exit(1);

}

int test\_4(){

char \* map;

size\_t length = 6009;

struct stat sb;

printf("Executing Test #4 (writing beyond the edge): \n");

int fd = create\_fd(length);

if(fstat(fd, &sb) < 0){

error\_message("ERROR: could not fstat properly.", strerror(errno));

exit(255);

}

int size\_1 = sb.st\_size;

if((map = mmap(NULL, length, PROT\_WRITE, MAP\_SHARED, fd, 0)) == MAP\_FAILED){

error\_message("ERROR: could not mmap file from memory.", strerror(errno));

exit(255);

}

printf("map[%d]'s original size is %d, writing a new value at the next byte: D\n", map[length-1], size\_1);

map[length] = 'D';

if(fstat(fd, &sb) < 0){

error\_message("ERROR: could not fstat properly.", strerror(errno));

exit(255);

}

int size\_2 = sb.st\_size;

printf("map's new size after writing a new value is %d, the value at map[%zu] is %c\n", size\_2, length, map[length]);

if(size\_1 == size\_2){

map\_close\_error(length, fd, map);

printf("The sizes are the same. \n");

exit(1);

}

else{

map\_close\_error(length, fd, map);

printf("The sizes are different.\n");

exit(0);

}

}

int test\_5(){

char \*map;

size\_t length = 6009;

char buf[1];

struct stat sb;

int fd, offset;

fd = create\_fd(length);

printf("Executing Test #5 (writing into a hole): \n");

if((map = mmap(NULL, length, PROT\_WRITE, MAP\_SHARED, fd, 0)) == MAP\_FAILED){

error\_message("ERROR: could not mmap file from memory.", strerror(errno));

exit(255);

}

map[length] = 'X';

buf[0] = 'A';

printf("map's original size is %zu, writing a new value one byte beyond the last byte: X\n", length);

if(lseek(fd, length + 16, SEEK\_SET) < 0){

error\_message("ERROR: could not lseek properly", strerror(errno));

exit(255);

}

if(write(fd, buf, 1) < 0){

error\_message("ERROR: could not write properly", strerror(errno));

exit(255);

}

if(lseek(fd, length, SEEK\_SET) < 0){

error\_message("ERROR: could not lseek properly", strerror(errno));

exit(255);

}

if(read(fd, buf, 1) < 0){

error\_message("ERROR: could not read properly", strerror(errno));

exit(255);

}

printf("Read from \'one byte beyond last byte\': %c\n", buf[0]);

if(buf[0] == 'X'){

map\_close\_error(length, fd, map);

printf("Writing into hole is visible. \n");

exit(0);

}else{

map\_close\_error(length, fd, map);

printf("Writing to hole is not visible. \n");

exit(1);

}

}

int test\_6(){

int fd, length = 2000;

char \*map;

printf("Executing Test #6 (reading beyond the edges): \n");

if((map = mmap(NULL, 8192, PROT\_WRITE|PROT\_READ, MAP\_SHARED, fd, 0)) == MAP\_FAILED){

error\_message("ERROR: could not mmap file from memory.", strerror(errno));

exit(255);

}

printf("map[3000]: \'%d\'\n", map[3000]);

printf("map[6000]: \'%d\'\n", map[6000]);

map\_close\_error(length, fd, map);

exit(0);

}

int main(int argc, char \*\*argv){

for(int i = 0; i < 32; i++){

signal(i, sig\_handler);

}

if(argc > 2 || argc == 1){

error\_message("ERROR: not enough input arguments/too many input arguments", 0);

return 1;

}

else{

switch(atoi(argv[1])){

case 1:

test\_1();

break;

case 2:

test\_23(MAP\_SHARED);

break;

case 3:

test\_23(MAP\_PRIVATE);

break;

case 4:

test\_4();

break;

case 5:

test\_5();

break;

case 6:

test\_6();

break;

default:

error\_message("ERROR: unidentified input test value, choose a value from 1-6.", strerror(errno));

return 1;

}

}

}