CS 450 – Operating Systems

Spring 2018

02/28/2018

Mayank Bansal
A20392482

Programming Assignment 2

PART A

The write(fd, n, 10) system call

We shall trace the write syscall based on the user program. The first step in this process is that the user called the write system call in the user program. We shall trace step by step what happens after that.

Step 1: User calls ***write(fd, n, 10)***

This will then invoke a function defined in usys.S

```
SYSCALL(write)
```

The following macro will execute:

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

Which will become:

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_write  %eax; \
    int $T_SYSCALL; \
    ret
```

This value of SYS_write comes from the definition in ***syscall.h***

```
#define SYS_write           16
```

Here, ***eax*** means the value of SYS_write should be put in the ***eax*** register. Since 16 is the value of SYS_write, the ***eax*** register will have 16.

The value of ***int*** becomes 64 as the value of T_SYSCALL is 64 in ***traps.h***:

```
#define T_SYSCALL       64      // system call
```

Step 2: trap function call

The trap function in trap.c is called next:

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  …
  …
  …
}
```

In this function, we check if the passed trapno is a valid systemcall. If the syscall is valid, then it calls ***syscall();***

Step 3: syscall function

The syscall function is defined in the syscall.c file such as:

```

void syscall(void) {
    struct proc *curproc = myproc();


    if (num >= 0 && num < NELEM(syscalls) && syscalls[num])
        curproc->tf->eax = syscalls[num]();
    else {
        cprintf("%d %s: unknown sys call %d\n",
                curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

Here, the function checks the value of the ***eax*** register. If the value is a valid trap number. If it is, then the function calls the function as defind in syscall.c

```
static int (*syscalls[])(void) = {
        [SYS_write]    sys_write,

      …
      …
};
```

the sys_write function is defined in the sysfile.c file.

```

int sys_write(void) {
    struct file *f;
    int n;
    char *p;

    if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
        return -1;
    return filewrite(f, p, n);
}
```

Step 4: sys_write

The sys_write function calles argfd() as defined in sysfile.c

```
// Fetch the nth word-sized system call argument as a file
descriptor
// and return both the descriptor and the corresponding struct
file.
static int argfd(int n, int *pfd, struct file **pf) {
    int fd;
    struct file *f;

    if (argint(n, &fd) < 0)
        return -1;
    if (fd < 0 || fd >= NOFILE || (f = myproc()->ofile[fd]) == 0)
```

```
        return -1;
    if (pfd)
        *pfd = fd;
    if (pf)
        *pf = f;
    return 0;
}
```

Here we check for a valid file descriptor. Since the user gave an invalid file descriptor, it returns **-1**. This value is passed up the stack of calls that the user ***write()*** called. This value is passed to syswrite() which passes it to syscall() function.

The syscall function then branches into:

```
if (num >= 0 && num < NELEM(syscalls) && syscalls[num])
    curproc->tf->eax = syscalls[num]();
else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
}
```

Now that the ***eax*** register is set to ***-1.***

Step 5: handle trap error

Since the eax register is set to ***-1***

The ***ret*** instruction in the assembly code then returns the value of the eax register. Since the value is negative, it indicates that an error occurred, so the error is handled and returned to the user.

```
    movl $SYS_write  %eax; \
    int $T_SYSCALL; \
    ret
```

The error handler will handle this error and show the user the appropriate message.

PART B

1. Building the project

```
Make the files:

$ make

Start Emulator:

$ make qemu-nox

Start the syscallCount.c program using:

CS-450$ syscallCount
```

2. Code Modifications

2.1 usys.S

added line:

(make a syscall)

```
SYSCALL(getCallCount)
```

2.2 syscall.h

added line:

(define syscall number)

```
#define SYS_getCallCount    22
```

2.3 user.h

added line:

(make syscall available)

```
int getCallCount(int *counts, int size);
```

2.4 proc.h

Modified proc structure to add callCount[23] so we can store all the callCounts for a particular process

```
// Per-process state
struct proc {
    uint sz;                         // Size of process memory (bytes)
    pde_t *pgdir;                    // Page table
    char *kstack;                    // Bottom of kernel stack for this
process
    enum procstate state;            // Process state
    int pid;                         // Process ID
    struct proc *parent;             // Parent process
    struct trapframe *tf;            // Trap frame for current syscall
    struct context *context;         // swtch() here to run process
    void *chan;                      // If non-zero, sleeping on chan
    int killed;                      // If non-zero, have been killed
    struct file *ofile[NOFILE];      // Open files
    struct inode *cwd;               // Current directory
    char name[16];                   // Process name (debugging)
    int callCount[23];
};
```

2.5 sysfile.c

Added the following lines of code:

```
/***********************
 * System Call Counter *
 ***********************/

/*
 * Initalize counterArray
 * and the initFlag
 */
int callCounts[23];
int initFlag = 0;

void updateCount(int syscall);

/*
 * Function to update count
 * of a particular system call
 *
 * initialize the count array
 * to zeros if this is the first
 * time this function is being
 * called
 *
 * then update at the syscall
 * number array position
 */
```

updateCount adds the sysCall for a particular syscall ID for a particular process.

```
void updateCount(int syscall) {

    struct proc *curproc = myproc();

    // check init flag
    if (initFlag == 0)

        // init all counts to 0
        for (int i = 0; i < 23; i++)
            curproc->callCount[i] = 0;

    // set init flag
    initFlag = 1;

    // increment count for syscall
    curproc->callCount[syscall]++;
}

/*
 * Function to copy counts
 * for usage in the main program
 */
```

getCallCount first gets an array and size pointers and copies the callCount from the current process to the count array. We can use this count array in the parent program (test program) to run through all the counts.

```
int getCallCount(void) {

    // init pointer
    int * counts;
    int size;

    // get counts pointer and size
    argint(1, &size);
    argptr(0, (void*)&counts, size);

    struct proc *curproc = myproc();
    // copy count array to pointer
    for (int i = 0; i < 23; i++)
        counts[i] = curproc->callCount[i];

    return 0;
}
```

2.6 syscall.h

Added the following lines of code:

```
// Static array to keep track of the counts for each syscall
extern int count_array[23];

// Lock for spinlocking
struct spinlock lock;

// Check if initialized
int init = 0;

void updateCount(int syscall);

void syscall(void) {
    struct proc *curproc = myproc();

    // initalize lock
    if (init == 0) {
        initlock(&lock, "getCallCount lock");
        init = 1;
    }

    // get syscall number
    int num = curproc->tf->eax;

    // lock
    acquire(&lock);

    // callUpdate counter for the process
    updateCount(num);

    // release lock
    release(&lock);

    if (num >= 0 && num < NELEM(syscalls) && syscalls[num])
        curproc->tf->eax = syscalls[num]();
    else {
        cprintf("%d %s: unknown sys call %d\n",
                curproc->pid, curproc->name, num);
        curproc->tf->eax = -1;
    }
}
```

Here we init a lock for the getCallCount, so that we aren't messing with random resources and have resource conflics. We get the syscall number from the current process, set a lock, updateCount for that process, releast the lock and the procede with the syscall.

3. Test Program

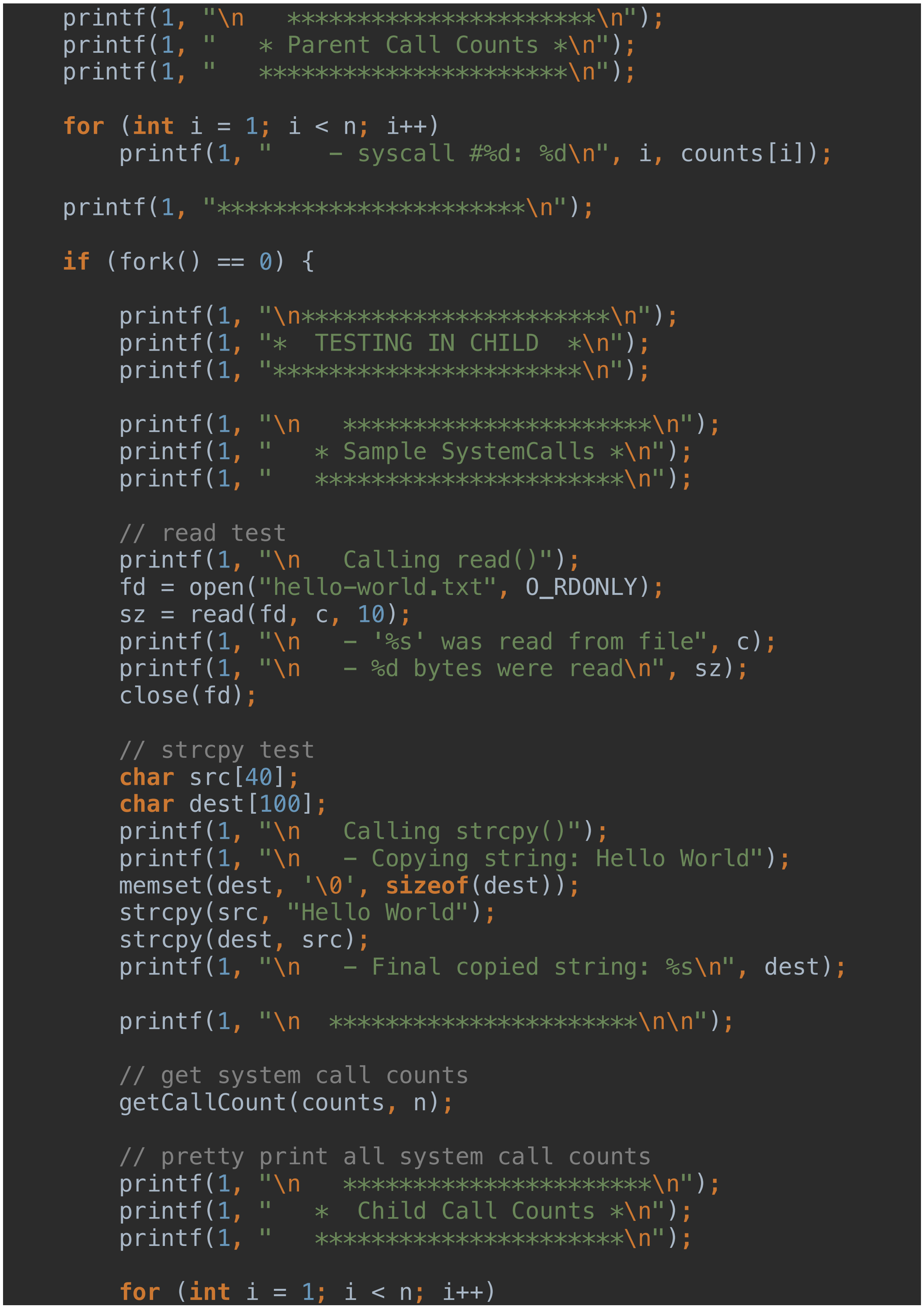
This is the test program that I used for testing the system calls commented appropriately. Here I first simply print the initial count of the parent process.

After that, I run some system calls in the parent, and then call the **getCallCount()** and pretty print it so that we can compare with the child.

Once we're done with that, we **fork()** and run some tests in the child process. In the child process. Once we're done running tests in the child process, we call the **getCallCount()** and pretty print the results.

We have to **wait()** in the parent process and wait for the child to finish.

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char *argv[]) {

    int n = 23;
    int counts[n];


    /*********************
     * Print Init Counts *
     *********************/
    // get system call counts
    getCallCount(counts, n);

    // pretty print all system call counts
    printf(1, "\n**********************\n");
    printf(1, "*  Init Call Counts  *\n");
    printf(1, "**********************\n");

    for (int i = 1; i < n; i++)
        printf(1, " - syscall #%d: %d\n", i, counts[i]);

    printf(1, "**********************\n\n");

    /*********************
     * Test System Calls *
     *********************/

    printf(1, "\n**********************\n");
    printf(1, "*  TESTING IN PARENT *\n");
    printf(1, "**********************\n");

    printf(1, "\n   **********************\n");
    printf(1, "   * Sample SystemCalls *\n");
    printf(1, "   **********************\n");

    // mkdir test
    printf(1, "\n   Calling mkdir()");
    mkdir("Hello World");
    printf(1, "\n   - Directory 'Hello World' created\n");

    // write test
    int sz, fd;
    char *c = (char *) malloc(sizeof("Hello World"));
    printf(1, "\n   Calling write()");
    fd = open("hello-world.txt", O_CREATE | O_WRONLY);
    write(fd, "Hello World", strlen("Hello World"));
    printf(1, "\n   - File hello-world.txt created");
    printf(1, "\n   - 'Hello World' written to file");
    printf(1, "\n   - %d bytes were written\n",
sizeof(strlen("Hello World")));
    close(fd);

    printf(1, "\n   **********************\n\n");

    // get system call counts
    getCallCount(counts, n);

    // pretty print all system call counts
```



```
printf(1, "\n   **********************\n");
printf(1, "   * Parent Call Counts *\n");
printf(1, "   **********************\n");

for (int i = 1; i < n; i++)
    printf(1, "    - syscall #%d: %d\n", i, counts[i]);

printf(1, "**********************\n");

if (fork() == 0) {

    printf(1, "\n**********************\n");
    printf(1, "*  TESTING IN CHILD  *\n");
    printf(1, "**********************\n");

    printf(1, "\n   **********************\n");
    printf(1, "   * Sample SystemCalls *\n");
    printf(1, "   **********************\n");

    // read test
    printf(1, "\n   Calling read()");
    fd = open("hello-world.txt", O_RDONLY);
    sz = read(fd, c, 10);
    printf(1, "\n   - '%s' was read from file", c);
    printf(1, "\n   - %d bytes were read\n", sz);
    close(fd);

    // strcpy test
    char src[40];
    char dest[100];
    printf(1, "\n   Calling strcpy()");
    printf(1, "\n   - Copying string: Hello World");
    memset(dest, '\0', sizeof(dest));
    strcpy(src, "Hello World");
    strcpy(dest, src);
    printf(1, "\n   - Final copied string: %s\n", dest);

    printf(1, "\n  **********************\n\n");

    // get system call counts
    getCallCount(counts, n);

    // pretty print all system call counts
    printf(1, "\n   **********************\n");
    printf(1, "   *  Child Call Counts *\n");
    printf(1, "   **********************\n");

    for (int i = 1; i < n; i++)
```

```
            printf(1, "    - syscall #%d: %d\n", i, counts[i]);

        printf(1, "   **********************\n");
    }

    wait();

    exit();

}
```

4. Test Program Output

Here we can see the results of our test program. Here we can see the parent initial call counts are 0 except for #7 and #22 (**exec** & **getCallCount**)

```
mayankbansal@mayankbansal-
VirtualBox:/media/sf_Ubuntu_VM_Shared/cs450-operating-systems/xv6-
public$ make qemu-nox
qemu-system-i386 -nographic -drive
file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart
32 bmap start 58
init: starting sh
CS450$ syscallCount

**********************
*  Init Call Counts  *
**********************
 - syscall #1: 0
 - syscall #2: 0
 - syscall #3: 0
 - syscall #4: 0
 - syscall #5: 0
 - syscall #6: 0
 - syscall #7: 1
 - syscall #8: 0
 - syscall #9: 0
 - syscall #10: 0
 - syscall #11: 0
 - syscall #12: 1
 - syscall #13: 0
 - syscall #14: 0
 - syscall #15: 0
 - syscall #16: 0
 - syscall #17: 0
 - syscall #18: 0
 - syscall #19: 0
 - syscall #20: 0
 - syscall #21: 0
 - syscall #22: 1
**********************
```

Now we run some tests in the parent like mkdir, write.
The mkdir calls the MKDIR syscall and write will invoke one **open(#20)**, **close(#21), getCallCount(#22)**

write(#16) is also called

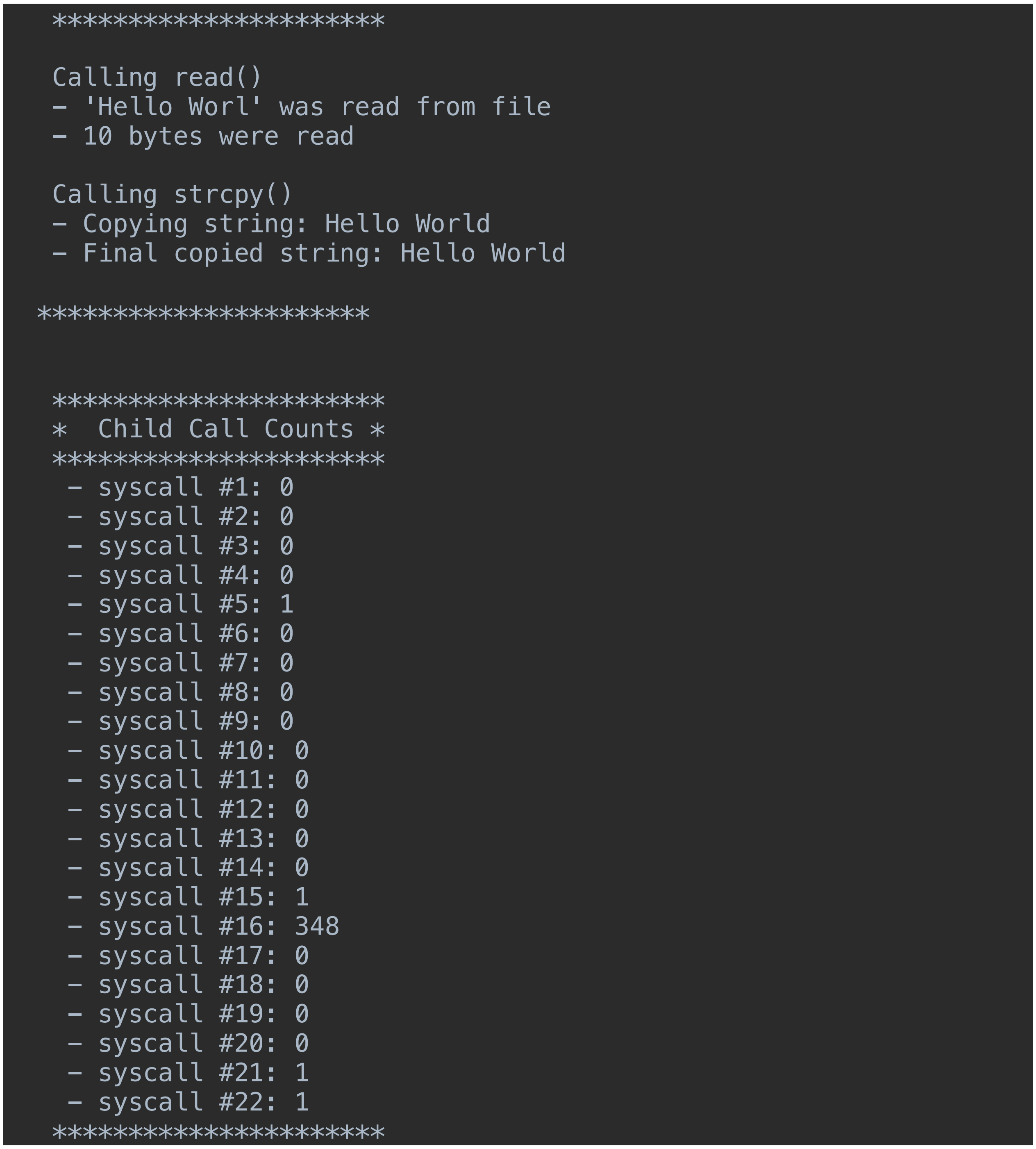
In the results we can see the **getCallCount(#22)** went from 1 to 2 since we called it last time. All the predicted results were verified to be true.

```
**********************
*  TESTING IN PARENT *
**********************

   **********************
   * Sample SystemCalls *
   **********************

   Calling mkdir()
   - Directory 'Hello World' created

   Calling write()
   - File hello-world.txt created
   - 'Hello World' written to file
   - 4 bytes were written

   **********************
   **********************
   * Parent Call Counts *
   **********************
    - syscall #1: 0
    - syscall #2: 0
    - syscall #3: 0
    - syscall #4: 0
    - syscall #5: 0
    - syscall #6: 0
    - syscall #7: 1
    - syscall #8: 0
    - syscall #9: 0
    - syscall #10: 0
    - syscall #11: 0
    - syscall #12: 2
    - syscall #13: 0
    - syscall #14: 0
    - syscall #15: 1
    - syscall #16: 831
    - syscall #17: 0
    - syscall #18: 0
    - syscall #19: 0
    - syscall #20: 1
    - syscall #21: 1
    - syscall #22: 2
**********************
```

Now, we test in the child. We call one **READ(#5)** and read the file that we just wrote into. Since we again have to open and close the file, **OPEN(#15), CLOSE(#21)** are also called. We also test a strcopy.

We then call **getCallCount(#22)**. We can see that the count is only one, as the program tracks counts for different processes. The child process called it only once, so we have only that updated. Similarly**, OPEN(#15)** and **CLOSE(#21)** show up only once as we called them only once in the child process.

```


**********************
*  TESTING IN CHILD  *
**********************

   **********************
   * Sample SystemCalls *
```



```
   **********************

   Calling read()
   - 'Hello Worl' was read from file
   - 10 bytes were read

   Calling strcpy()
   - Copying string: Hello World
   - Final copied string: Hello World

  **********************


   **********************
   *  Child Call Counts *
   **********************
    - syscall #1: 0
    - syscall #2: 0
    - syscall #3: 0
    - syscall #4: 0
    - syscall #5: 1
    - syscall #6: 0
    - syscall #7: 0
    - syscall #8: 0
    - syscall #9: 0
    - syscall #10: 0
    - syscall #11: 0
    - syscall #12: 0
    - syscall #13: 0
    - syscall #14: 0
    - syscall #15: 1
    - syscall #16: 348
    - syscall #17: 0
    - syscall #18: 0
    - syscall #19: 0
    - syscall #20: 0
    - syscall #21: 1
    - syscall #22: 1
   **********************
```

The reason for testing it with a parent and child process is so that we know which process is calling which system calls. It verifies that the syscall counter is running properly. We won’t count any unnecessary calls that the user program didn’t call.