

Programming Assignment 2

CS450 Fall, 2021

Isaias Rivera

Oct 11, 2020

Task 1 - Trace close(fd)

Kernel	Hardware	User	File : line Nos. : func	Description
		Define invalid fd call close(fd)	test.c : 6-7 : main	Our program calls close(fd) with a bad file descriptor.
	SYS_close to %eax return to vector64		usus.S : 17 : close	Load 21 , the syscall number for close , to be passed to trap .
	call alltraps(T_SYSCALL)		vectors.S : 317-321 : vector64	Call alltraps with the number 64 to indicate that we are trapping with a syscall.
	Push trapframe tf for system call call trap(tf)		trapasm.S : 4-20 : alltraps	Create trapframe to switch to kernel space. Then goto function trap .
Check that trapno is T_SYSCALL Set current process tf with given tf call syscall()			trap.c : 39-43 : trap	64 means syscall. Save trapframe to current process, then call syscall .
Get trap num from current process curproc syscalls[num] == sys_close call sys_close()			syscall.c : 135-139 : syscall	Get %eax from the trapframe from earlier, which should be 21 . Then, call syscalls[21] which is sys_close .
call argfd(0, &fd, &f)			sysfile.c : 99 : sys_close	Get the parameter fd for this syscall.
Get current process fd and f Find that fd is invalid return -1 to sys_close return -1 to syscall			sysfile.c : 29-30 : argfd	Using fd , find the open file and check that it is valid. It is not valid, return -1 .
			sysfile.c : 100 : sys_close	Parameter fd is invalid. Return -1 .
Set current process curproc tf->eax to -1 return to trap return to alltraps			syscall.c : 145 : syscall	Set %eax to -1 to indicate that this syscall failed.
	Pop trapframe tf return to main		trap.c : 46 : trap trapasm.S : 21-32 : trapret	Return to hardware. Restore trapframe and go back to our program.
		printf return value of close(fd)	test.c : 8 : main	Print feedback to user that the syscall returned -1

Task 2 - System Call countTraps()

Design

In order to implement the system call `countTraps` I had add additional kernel code. `countTraps`, as the assignment describes it, should count whenever any trap occurs. However, because I was somewhat confused as to what the assignment is asking for exactly when it comes to each *type* of trap, I implemented counters for every process which count *both* the traps that occur and the system calls that get called.

Each process is represented with the struct `proc`. I took advantage of this by adding the attributes `trapcounts` and `syscounts`. Both of these attributes are arrays that are long enough to store a counter for each possible system call and trap. With these counters I can now increment them whenever we enter `trap` or `syscall`, using `myproc` to get the current running process. However, before I can increment the arrays, I need to define which indexes were for what trap number or system call number.

Counting Traps to OS

For the trap numbers, because the actual numbers in `traps.h` ranged from 0-500, I had to map each `trapno` to the numbers 0-22 to make it cleaner to implement. This was done with the function `trapnos`.

I also needed a way to represent each `trapno` with a string. I did this with the array `trapnames`, where each `trapno` mapped to the appropriate string. This would be used when printing to console.

After all this, I now then increment the array when traps occur. This would be in the `trap` function, where I get the current running process' struct and increment the appropriate `trapcounts` index. The index depends on what `trapframe` is given to `trap`, this is where `trapnos` helps map this to the appropriate index.

Counting System calls

The implementation for counting system calls is very similar to counting the traps to OS.

Unlike the trap numbers, because each system call number, including `countTraps`, is under the continuous range of 1-22 I can index them directly without mapping them.

I also represented each system call number with a string with the array `sysnames`.

After all this, I can now increment the array when a system call is called. This would be in the `syscall` function, where I get the current running process' struct and increment the appropriate `syscounts` index. The index depends on the system call number given to `syscall`.

Implementing the Actual System Call

Now that I can count each trap and system call, I need to let the user print it. This is done in the system call `countTraps`. Implementing the system call function itself into xv6 was relatively straightforward as I just followed how the other system calls were implemented. After that, I just had to print each trap or system call counter next to the appropriate string name. I only print the counters for traps / system calls that are non-zero.

One more thing that I also needed to do for both counters is reset them whenever a `proc` gets reused. This is done when child processes are reaped.

Calling

To call the function in a user process, it first has to include "`user.h`" and then it can call the function `countTraps()`. The system call will then print everything out onto the console. `countTraps` also returns the total number of traps that occurred.

Changes made

File Name	Changes
<code>def.h</code>	I added the headers for both <code>sysname</code> and <code>trapname</code> . These are used to get the string name of a syscall or trapno respectively.
<code>proc.c</code>	Reset each counter for a <code>proc</code> when it is reaped.
<code>proc.h</code>	Added both counter arrays <code>trapcounts</code> and <code>syscounts</code> to <code>struct proc</code> .
<code>syscall.c</code>	Added header for the system call <code>countTraps</code> . Added <code>countTraps</code> system call to the array <code>syscalls</code> to <i>tell</i> syscall that it exists. Defined the function <code>sysname</code> which uses a lookup table to return the string name of a system call. Increment <code>syscounts</code> in <code>syscall</code> .
<code>syscall.h</code>	Defined <code>SYS_countTraps</code> as 22.
<code>test.c</code>	User process with the different test cases.
<code>trap.c</code>	Defined the function <code>trapnos</code> which uses a lookup table to return the string name of a trap number. Added <code>countSyscalls</code> , which prints the counts that each system call has been called. Added <code>sys_countTraps</code> , which prints the counts that each trap has occurred and also calls <code>countSyscalls</code> . Increment <code>trapcounts</code> in <code>trap</code> .
<code>user.h</code>	Added header for system call <code>countTraps</code> .
<code>usys.S</code>	Added macro call for system call <code>countTraps</code> .

Test Cases

For the test cases I figured out how to call each system call, however, because most of the traps cause the process to exit, I was uncertain as to how to test each trap. Furthermore, because the counters are for each individual process, the counters get reset after a process exits. The traps that do actively show up on the counters are the system calls and the hardware interrupts.

Each test case is a function call in the same source file `test.c`. To run a test case, change the define `TEST_CASE` to a number 0-4 then recompile and run.

Test Case 0

Code

```
// +0 syscalls
void testCase0(void) {}

// +3 syscalls because sh calls both sbrk and exec, and test calls countTraps
int main(int argc, char *argv[]) {
    runTestCase();

    countTraps();
    exit();
}
```

Output

```
-----[ Syscalls ]-----
exec           : 1
sbrk           : 1
countTraps     : 1
Total Syscalls : 3

-----[ Traps ]-----
System call    : 3
Total Traps    : 3
```

Description

Because the base program `test.c` is started by the shell `sh.c` it starts with 2 traps are already counted. One is `sbrk` from the `malloc` at `sh.c:200` and the other is the `exec` at `sh.c:78`. And because `countTraps` is also a system call, that too gets counted for a total of +3 for the system call count.

Every other test case includes running main.

Test Case 1

Code

```
// +7 syscalls
void testCase1(void) {
    printf(3, "Hello!\n"); // Each char calls sys_write
}
```

Output

-----[Syscalls]-----

exec	: 1
sbrk	: 1
write	: 7
countTraps	: 1
Total Syscalls	: 10

-----[Traps]-----

Interrupts	: 1
System call	: 10
Total Traps	: 11

Description

Because of the way `printf` works, each `char` in the string `"Hello!\n"` calls the system call `write`. 7 calls to `write` for the 7 chars in `"Hello!\n"`. This test case shows that, even if the user program does not call a syscall directly, the syscalls are still counted.

Test Case 2

Code

```
// +2 syscalls
void testCase2(void) {
    if (fork() == 0) {
        int a = 1 / 0;
        exit();
    }
    wait();
}
```

Output

-----[Syscalls]-----

fork	: 1
wait	: 1
exec	: 1
sbrk	: 1
countTraps	: 1
Total Syscalls	: 5

-----[Traps]-----

Interrupts	: 1
System call	: 5
Total Traps	: 6

Description

This test case shows how a child process does not add onto the parent process counter, as it is a separate process.

Test Case 3

Code

```
// +40 syscalls
void testCase3(void) {
    int p[2], i, fd;
    char *arg = "Hi";
    char *err = (char *)-1;
    char inbuf[16];
    struct stat fs;

    for (i = 0; i < 10; i++) {
        if (fork() == 0) {
            close(1);
            fd = open("backup", O_CREATE | O_RDWR);
            dup(fd);
            printf(1, "Child: Hello!\n");
            close(fd);
            exec("echo", &arg);
        }
        wait();
        kill(0);
        getpid();
    }
}
```

Output

-----[Syscalls]-----

fork	: 10
wait	: 10
kill	: 10
exec	: 1
getpid	: 10
sbrk	: 1
countTraps	: 1
Total Syscalls	: 43

-----[Traps]-----

Interrupts	: 4
System call	: 43
Total Traps	: 47

Description

This test case calls multiple system calls 10 times, meaning the counter should also increment by 10 for each call, which it does. This case is also another example of how a child process does not add onto the parent process counter.

Test Case 4

Code

```
// +124 syscalls
void testCase4(void) {
    ... // Initial part omitted due to space, it is the same as testCase3
    for (i = 0; i < 8; i++) {
        sleep(0);
        uptime();
        sbrk(0);
    }
    for (i = 0; i < 5; i++) {
        fd = open("backup", O_CREATE | O_RDWR);
        pipe(p);
        write(fd, arg, 2);
        read(fd, &inbuf, 2);
        fstat(fd, &fs);
        close(fd);
        mknod("", -1, -1);
        unlink(err);
        link(err, err);
        mkdir("test");
        dup(0);
        chdir("");
    }
}
```

Output

```
-----[ Syscalls ]-----
fork           : 10
wait           : 10
pipe           : 5
read           : 5
kill           : 10
exec           : 1
fstat          : 5
chdir          : 5
dup            : 5
getpid        : 10
sbrk          : 9
sleep         : 8
uptime        : 8
open          : 5
write         : 5
mknod         : 5
unlink        : 5
link          : 5
mkdir         : 5
close         : 5
countTraps    : 1
Total Syscalls : 127

-----[ Traps ]-----
Interrupts     : 7
System call    : 127
Total Traps    : 134
```

Description

This test case adds on to test case 3. It calls each system call possible at a set number of times. Keeping in mind that `sbrk` is called once beforehand, these counters all add up to match the for loops for each respective system call.