**write() System Call Testing**

**Test 1**

Command: $ test 1

Result: copying string

copied string ‘t1’

write count 34

Validity: Tests **write**() counter when printf is used. In this case, **write**() is called once for every character that get

printed to stdin. Per the specifications, strcpy was used, but makes no system calls so it does not affect the counter.

**Test 2**

Command: $ test 2 test.txt

Result: writing string ‘CS450-PA2’ to ‘test.txt’

write count 1

Validity: Tests **write**() counter when writing to a file. Unlike *test 1*, only one call to **write**() is made as the entire memory block is being written instead of character by character.

**Test 3**

Command: $ test 3 test.txt

Result: reading string ‘CS450-PA2’ from ‘test.txt’

read count 1

Validity: Tests **read**() counter when reading from a file. Similar to *test 2*, only a single call to **read**() is made since the file is read in bulk.

**Test 4**

Command: $ test 4 test.txt

Result: opening file ‘test.txt’

closing file ‘test.txt’

open count 1

close count 1

Validity: Tests **open**() and **close**() counters when dealing with a valid file descriptor. Each counter gets incremented once.

**Test 5**

Command: $ test 5

Result: exec count 1

Validity: Tests **exec**() counter when running a user program. In this test, no actual system call is made by the user, but the exec counter still returns a count of 1 as the *process structure* gets allocated before the actual program gets executed.

**Test 6**

Command: $ test 6

Result: exec count 1

exec’ing ‘test 5’

exec count 2

Validity: Tests **exec**() counter when exec’ing from a current user program. Like *test 5*, the excec counter starts with a count of 1. After the exec from within the user program, the exec counter increments to 2. This happens because in xv6, the new process keeps the same *process structure* as the one it got exec’ed from. This results in the system call counter keeping the same counts from the previous program.

**Test 7**

Command: $ test 7

Result: uptime: [int]

sleeping

uptime: [int + 100]

uptime count 2

sleep count 1

Validity: Tests **uptime**() and **sleep**() counters. To test whether **sleep**() gets invoked, a single call to **uptime**() was made before and after. This is the result for an uptime count of 2 and a sleep count of 1.

**Test 8**

Command: $ test 8

Result: sbrk address: [int]

incrementing end of data segment by 10

sbrk address: [int + 10]

sbrk count 3

Validity: Tests **sbrk**() counter. The reason sbrk has a count of 3 instead of the expected 2 is similar for the discrepancy in *test 5* and *test 6*. **sbrk**() gets called after the *process structure* gets allocated to initialize the stack frame for the user’s program, this resulting in a sbrk counter of 1 before any **sbrk**() call is made.

**Test 9**

Command: $ test 9

Result: getting count for 'system call 0'

'system call 0' count -1

getting count for 'system call 23'

'system call 23' count -1

getting count for 'system call fake'

'system call fake' count -1

getting count for getCountCall

getCallCount count 3

Validity: Tests **getCallCount**() counter when an invalid value is passed as a parameter. An invalid parameter can either be an integer that is not mapped to a system call (such as 0 or 23) or some other non-integer data type, such as a string. On error, **getCallCount**() returns -1 (similar to some of the other system calls). However, as **getCallCount**() was still successfully invoked each time, the getCallCount counter has a value of 3.

**Test 10**

Command: $ test 10

Result: opening file 'fake.txt'

open failed

open count 1

Validity: Tests the counter functionality for a system call that returns an error. In this case, **open**() still gets successfully called, so the counter correctly gets incremented to 1. In the event an invalid system call gets called (which is nearly impossible), the counter will never increment as syscall() would return an error before the counter can be manipulated.

**Test 11**

Command: $ test 12

Result: sleeping *x100*

sleep count 100

Validity: Load testing the counter functionality when many system calls are made. In this test, **sleep**() is invoked 100 times, resulting in a sleep count of 100.

**Test 12**

Command: $ test 12

Result: initial fork count 0

child pid: [int]

child fork count 0

child write count 33

getpid count 1

wait count 0

getCallCount count 4

parent fork count 1

parent write count 41

getpid count 0

wait count 1

getCallCount count 5

Validity: Tests **fork**(), **getpid**(), **write**(), **wait**(), and **getCallCount**() counters. This is one of the more complicated tests as it checks if a fork’ed process gets allocated properly. Here, the counters for the child process are separate from the parent process and no system calls from either process interfere with the other. In the child process, the wait counter has a value of 0 as **wait**() was never invoked in this process. Similarly in the parent process, **getpid**() was never invoked and thus the getpid counter has a value of 0. The rest of the counters follow similar patterns for the previous tests.