Project 2 Test Report

Alexander DuPree

January 21, 2020

Introduction

The following test report documents the tests performed for project two. The test cases and strategies closely follow the project two rubric.

Each section contains test cases related to the sections topic. Each test case will describe the name of the test, the expected result, actual result, as well as a discussion and indication of the Pass/Fail status. The actual result will be provided in the form of a screen shot of the console.

Compilation

This section presents all tests related to compiling the xv6 kernel. Test cases follow closely those outlined in the rubric.

Test Case: *With CS333_PROJECT set to 0 in the Makefile*
Assertions:

1. Code correctly compiles
2. Kernel successfully boots

Status: **PASS**

```
|11:48:29|adupree@babbage:[xv6-pdx]> grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
|11:48:33|adupree@babbage:[xv6-pdx]> make clean run
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie  _halt
rm -rf dist dist-test
make -s clean
make -s qemu-nox
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 1941 total 2000
balloc: first 646 blocks have been allocated
balloc: write bitmap block at sector 58
boot block is 448 bytes (max 510)
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.117014 s, 43.8 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.011157 s, 45.9 kB/s
327+1 records in
327+1 records out
167856 bytes (168 kB, 164 KiB) copied, 0.0243694 s, 6.9 MB/s
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

Figure 1: Compilation and boot with CS333_PROJECT set to 0.

The command `grep "CS333_PROJECT ?=" Makefile` shows that the CS333_PROJECT macro is set to 0. The following command `make clean run` demonstrates that the code correctly compiles and the kernel successfully boots. Furthermore, the commands were executed within seconds

of each other, indicating that tampering is not a possibility.

Test Case: *With CS333_PROJECT set to 2 in the Makefile*
Assertions:

1. Code correctly compiles
2. Kernel successfully boots

Status: **PASS**

```
|11:53:01|adupree@babbage:[xv6-pdx]> grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 2
|11:53:04|adupree@babbage:[xv6-pdx]> make clean run
rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
*.o *.d *.asm *.sym vectors.S bootblock entryother \
initcode initcode.out kernel xv6.img fs.img kernelmemfs \
xv6memfs.img mkfs .gdbinit \
_cat _echo _forktest _grep _init _kill _ln _ls _mkdir _rm _sh _stressfs _usertests _wc _zombie  _halt _date _ps _time _testsetuid  _testuidgid _p2-test
rm -rf dist dist-test
make -s clean
make -s qemu-nox
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 1941 total 2000
balloc: first 873 blocks have been allocated
balloc: write bitmap block at sector 58
boot block is 448 bytes (max 510)
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.152379 s, 33.6 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.00990448 s, 51.7 kB/s
333+1 records in
333+1 records out
170676 bytes (171 kB, 167 KiB) copied, 0.015729 s, 10.9 MB/s
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
```

Figure 2: Compilation and boot with CS333_PROJECT set to 2, CS333_P2 is defined.

The command `grep "CS333_PROJECT ?=" Makefile` shows that the CS333_PROJECT macro is indeed set to 2. The following command `make clean run` demonstrates that the code correctly compiles and the kernel successfully boots. Furthermore, the commands were executed within seconds of each other, indicating that tampering is not a possibility.

PS program, CTRL-P, CPU time, getprocs() system call

This section presents all tests related to the `ps` user program, `CTRL+P` interrupt, and the `getprocs()` system call. Test cases follow closely those outlined in the rubric.

Test Case: *Running the PS program*
Assertions:

1. Correctly displays process information
2. Elapsed CPU time is correct and formatted correctly
3. Information closely matches the `CTRL+P` interrupt
4. PS does not display `EMBRYO` or `UNUSED` processes
5. `getprocs()` copies process info into the user space up to the number of processes or to the size of the given table

Status: **PASS**

```
|13:55:22|adupree@babbage:[xv6-pdx]> 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 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       1.086     0.029 sleep   12288    80103841 80103956 801050ac 8010449a 8010543e 80105334
2       sh           0          0       1       1.058     0.011 sleep   16384    80103841 801002c4 8010185d 80100e89 801047d0 8010449a 8010543e 80105334
$ ps

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size
1       init         0          0       1       2.198     0.029 sleep   12288
2       sh           0          0       1       2.170     0.021 sleep   16384
3       ps           0          0       2       0.023     0.008 run     49152
$ ps -m 1

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size
1       init         0          0       1       6.070     0.029 sleep   12288
$ ps -m 72

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size
1       init         0          0       1       9.474     0.029 sleep   12288
2       sh           0          0       1       9.445     0.041 sleep   16384
5       ps           0          0       2       0.017     0.007 run     49152
$
```

Figure 3: Compilation and boot with CS333_PROJECT set to 0.

On startup, we immeditately call the `CTRL+P` interrupt to have a basis of comparison for the `ps` program. First thing to note, `CTRL+P` displays the new header information for project 2. This includes the process UID/GID, and the total time in the CPU.

We then execute the `ps` user program. As we can see the output closely resembles the output for `CTRL+P`. However, `ps` does not output the program counters, and also displays information for the `ps` process itself. You will also note that the CPU time for the `sh` process has increased slightly, and subsequent calls `ps` results in larger CPU times. This makes sense because the shell would have used time slices to parse the input and fork the child process. This indiciates that the CPU time tracker for processes is working correctly.

Lastly, my implementation of `ps` allows the user to specify the 'MAX' parameter for the `getprocs()` system call. Whatever the user supplies as an argument to `'-m'` will be used as the

`uproc*` table size which is passed into the `getprocs()` system call. As we can see in Figure 3, calling `ps -m 1` results in only the first process being displayed, and calling `ps -m 72` displays all three system processes. It is important to note that neither `EMBRYO` or `UNUSED` are displayed.

UID, GID, and PPID Tests

This section presents all tests related to the User Identifier (UID), Group Identifier (GID), and the Parent Process Identifier (PPID). Specifically, these tests demonstrate that the identifier fields are properly set and can be retrieved through system calls. Test cases follow closely those outlined in the rubric.

Test Case: *Setting/Getting the UID and GID with Built-In Shell Commands*
Assertions:

1. Correctly gets the UID/GID and displays to console
2. Correctly sets the UID/GID in the shell
3. Child processes correctly inherit the new UID/GID values
4. UID/GID cannot be set to numbers outside the range of $0 \leq x \leq 32767$

Status: PASS

```
|12:15:15|adupree@babbage:[xv6-pdx]> 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 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ _get uid
0
$ _get gid
0
$ _set uid 42
$ _set gid 43
$ ps

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size
1       init         0          0       1       18.371    0.023 sleep   12288
2       sh           42         43      1       18.347    0.038 sleep   16384
3       ps           42         43      2       0.021     0.010 run     49152
$ _set uid 32768
Invalid _set parameter
$ _set gid 32768
Invalid _set parameter
$ ps

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size
1       init         0          0       1       52.887    0.023 sleep   12288
2       sh           42         43      1       52.862    0.070 sleep   16384
4       ps           42         43      2       0.020     0.004 run     49152
$
```

Figure 4: Compilation and boot with CS333_PROJECT set to 0.

The first set of commands, highlighted in the Yellow box, demonstrates that assertion *(1)* is true. The call to `_get uid` and `_get gid` both display `0` to the console because the `sh` process inherits from the init process, whose UID/GID is `0`.

Assertions *(2)* and *(3)* are demonstrated true in the second set of commands, highlighted in the **Green** box. First, we set the UID/GID for the `sh` process to 42 and 43 respectively. Then we execute the `ps` user program which will inherit the new UID/GID values from the parent `sh` process and then display the UID/GID for all system processes.

Finally, the commands in the **Red** box demonstrate that the fourth assertion is also true. Attempting to set the UID or GID to 32768 results in an error message. Furthermore, the subsequent call to `ps` shows that the UID/GID of the `sh` process was not changed after the failure.

Test Case: *Running the 'testuidgid' test suite*
Assertions:

1. Correctly set / get UID, GID, and retrieve PPID
2. PPID for processes with no parents is properly handled
3. Correctly handle attempting to set UID/GID to invalid values
4. Child processes correctly inherit UID/GID values

Status: **PASS**

```
init: starting sh
$ testuidgid

Current UID is: 0
Setting UID to 100
Current UID is: 100

Confirm with CTRL-P:

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       6.902     0.027 sleep   12288    80103841 80103956 801050ac 8010449a 8010543e 80105334
2       sh           0          0       1       6.875     0.016 sleep   16384    80103841 80103956 801050ac 8010449a 8010543e 80105334
3       testuidgid   100        0       2       0.615     0.056 sleep   16384    80103841 80105179 8010449a 8010543e 80105334
$
Current GID is: 0
Setting GID to 200
Current GID is: 200

Confirm with CTRL-P:

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       13.279    0.027 sleep   12288    80103841 80103956 801050ac 8010449a 8010543e 80105334
2       sh           0          0       1       13.253    0.016 sleep   16384    80103841 80103956 801050ac 8010449a 8010543e 80105334
3       testuidgid   100        200     2       6.993     0.623 sleep   16384    80103841 80105179 8010449a 8010543e 80105334
$
My parent process is: 2

Setting UID to 111 and GID to 112 before fork(). Value should be inherited
Before fork(), UID = 111, GID = 112
Child: UID is: 111, GID is: 112

Confirm with CTRL-P:

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       17.730    0.027 sleep   12288    80103841 80103956 801050ac 8010449a 8010543e 80105334
2       sh           0          0       1       17.702    0.016 sleep   16384    80103841 80103956 801050ac 8010449a 8010543e 80105334
3       testuidgid   111        112     2       11.444    0.978 sleep   16384    80103841 80103956 801050ac 8010449a 8010543e 80105334
4       testuidgid   111        112     3       1.350     0.058 sleep   16384    80103841 80105179 8010449a 8010543e 80105334
$
Setting UID to 32768. This test should FAIL
SUCCESS! The setuid system call indicated failure

Setting GID to 32768. This test should FAIL
SUCCESS! The setgid system call indicated failure

Setting UID to -1. This test should FAIL
SUCCESS! The setuid system call indicated failure

Setting GID to -1. This test should FAIL
SUCCESS! The setuid system call indicated failure
** TEST UID/GID: All Tests Pass! **
$
```

Figure 5: Compilation and boot with CS333_PROJECT set to 0.

The `testuidgid` test suite, executes a series of tests to demonstrate that the UID, GID, and PPID functionality is correct. First, we set the UID of the current process to `100` and use the `CTRL-P` interrupt to verify the results. We then do the same manner of test for the GID.

Next we set the UID/GID to 111 and 112 respectively and fork the process to see if the values are correctly inherited. With `CTRL-P` we can see that the child process does indeed inherit the parents UID and GID. Furthermore, the child's PPID is `3` which is the correct parent process

ID. It's also important to note that the PPID for the `init` process is the same as it's PID, this is becasue the `init` process does not have a parent process.

Lastly, we attempt to set the the UID/GID to values just outside the valid boundary of $0 \leq x \leq 32767$. Beacuse each attempt to set the identifier to an invalid value returned failure, these tests pass.

Time program

This section presents all tests related to the `time` user program. Test cases follow closely those outlined in the rubric.

Test Case: *Calling time with no arguments and an invalid argument*
Assertions:

1. Displays the time it execute (null) or invalid argument.
2. Time does not crash the kernel

Status: PASS

```
|14:43:22|adupree@babbage:[xv6-pdx]> 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 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ time
(null) ran in 0.8 seconds
$ time not-a-program
not-a-program ran in 0.8 seconds
$
```

Figure 6: Running `time` with no arguments and an invalid argument

First, we run `time` with no arguments and the output matches the output provided in the project description. Next, when we run `time` with an invalid argument, the program displays the argument and the subsequent time. Note that both invocations resulted in the same time result of 0.8 seconds. This indicates that in both cases the program underwent the same control flow, where the `exec` system call failed, and the child process exited almost immediately.

Test Case: *Calling time with a valid command and subsequent argmuments*
Assertions:

1. Time correctly executes the valid command
2. Time correctly passes the arguments to the forked process

Status: **PASS**

```
|14:51:21|adupree@babbage:[xv6-pdx]> 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 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ time time echo "Hello World"
"Hello World"
echo ran in 0.19 seconds
time ran in 0.40 seconds
$
```

Figure 7: Calling time with multiple commands and arguments

The command `time time echo "Hello World"` shows that both our assertions are true. First, `time` is executed with the arguments `time echo "Hello World"`. The initial `time` process forks itself and executes the next command, `time`, with the arguments `echo "Hello World"`. This process repeats itself one more time with the execution of the `echo` command with the arguments `"Hello World"`. Finally, as the processes finish execution, the parent `time` processes print the elapsed time. Because the initial arguments perpuated isself through multiple layers and the correct commands were executed, we can be confident that our program is working as expected.

Test Case: *Calling the time command with a long running process*
Assertions:

1. The calculated time is accurate

Status: **PASS**

```
|15:03:18|adupree@babbage:[xv6-pdx]> 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 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ps; time usertests; ps;

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size
1       init         0          0       1       8.794     0.029 sleep   12288
2       sh           0          0       1       8.765     0.020 sleep   16384
3       sh           0          0       2       0.023     0.007 sleep   49152
4       ps           0          0       3       0.017     0.008 run     49152
usertests starting
arg test passed
createdelete test
createdelete ok
```

Figure 8: Invoking ps and time usertests

```
exec test
ALL TESTS PASSED
usertests ran in 134.213 seconds

PID     Name        UID        GID     PPID    Elapsed   CPU   State   Size
1       init        0          0       1       143.110   0.029 sleep   12288
2       sh          0          0       1       143.081   0.020 sleep   16384
3       sh          0          0       2       134.341   0.020 sleep   49152
593     ps          0          0       3       0.017     0.008 run     49152
$
```

Figure 9: time usertests result and subsequent ps call

To establish that the `time` command is accurate, we bound the command between two `ps` commands. As we can see the time result is nearly identical to the elapsed time of the `sh` process that was forked to execute the command sequence. This strongly indicates that the calculated time is indeed accurate. It is important to note that I opted to use `ps` over `date` (which was suggested in the rubric) to bound the command because `date` does not read form the `ticks` global variable instead reads time from the cmos, wich QEMU emulates. Because of this, when running the command `date; time usertests; date;` there is a 4-5 second discrepency. As such, I wanted to present a test that had less ambiguity and decided to bound the command with `ps`.