Project 3 Test Report

Alexander DuPree

February 12, 2020

Introduction

The following test report documents the tests performed for project three. The test cases and strategies closely follow the project three 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
3. `usertests` run to completion with all tests passed

Status: **PASS**

```
|14:06:12|adupree@babbage:[xv6-pdx]> grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
|14:06:25|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 648 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, 0.11129 s, 46.0 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.0105287 s, 48.6 kB/s
317+1 records in
317+1 records out
162804 bytes (163 kB, 159 KiB) copied, 0.0134172 s, 12.1 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
$ usertests
usertests starting
arg test passed
createdelete test
```

Figure 1: Compilation and boot with CS333_PROJECT set to 0 and execution of `usertests`

```
empty file name
empty file name OK
fork test
fork test OK
bigdir test
bigdir ok
uio test
pid 591 usertests: trap 13 err 0 on cpu 1 eip 0x33ce addr 0x800dbba0--kill proc
uio test done
exec test
ALL TESTS PASSED
$
```

Figure 2: Completion of `usertests` with output elided

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. Lastly, we can see that the execution of `usertests` is initiated in the same session, and Figure 2 shows that the tests run to completion and all tests pass.

Test Case: *With CS333_PROJECT set to 3 in the Makefile*
Assertions:

1. Code correctly compiles
2. Kernel successfully boots
3. `usertests` run to completion with all tests passed

Status: **PASS**

```
|14:11:48|adupree@babbage:[xv6-pdx]> grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 3
|14:11:50|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 _time _ps _testsetuid _testuidgid _p2-test _p3-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 908 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, 0.127171 s, 40.3 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.0110166 s, 46.5 kB/s
346+1 records in
346+1 records out
177432 bytes (177 kB, 173 KiB) copied, 0.010175 s, 17.4 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
$ usertests
usertests starting
arg test passed
createdelete test
```

Figure 3: Compilation and boot with CS333_PROJECT set to 3 and execution of `usertests`.

```
fork test OK
bigdir test
bigdir ok
uio test
pid 591 usertests: trap 13 err 0 on cpu 0 eip 0x33ce addr 0x80149970--kill proc
uio test done
exec test
ALL TESTS PASSED
$
```

Figure 4: Completion of `usertests` with output elided, CS333_P3 is defined

The command `grep "CS333_PROJECT ?=" Makefile` shows that the CS333_PROJECT macro is indeed set to 3. 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. Lastly, we can see that the execution of `usertests` is initiated in the same session and Figure 4 demonstrates that the tests run to completion and all tests pass.

State Lists and Console Commands

This section presents all tests related to the initilization, use, and maintenance of the kernels process state lists and debug console commands. Test cases follow closely those outlined in the rubric.

Test Case: *Initialization and use of* `UNUSED` *list and CTRL+F console interrupt*
Assertions:

1. `UNUSED` list is correctly initialized after Xv6 boots
2. `UNUSED` list is correctly updated when a process is allocated and deallocated
3. CTRL+F correctly shows the number of `UNUSED` processes.

Status: **PASS**

```
|19:43:41|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
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       2.071     0.041 sleep   12288    80103efb 80104029 8010
2       sh           0          0       1       2.028     0.013 sleep   16384    80103efb 801002c4 8010
$
Free List Size: 62
$ loopforever 2 &
$
Created 1 children, starting loopforever on parent

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       9.487     0.041 sleep   12288    80103efb 80104029 8010
2       sh           0          0       1       9.444     0.028 sleep   16384    80103efb 801002c4 8010
4       loopforever  0          0       1       0.867     0.849 run     12288
5       loopforever  0          0       4       0.847     0.834 run     12288
$
Free List Size: 60
$ kill 5; kill 4
$ zombie!
zombie!

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       19.353    0.047 sleep   12288    80103efb 80104029 8010
2       sh           0          0       1       19.310    0.040 sleep   16384    80103efb 801002c4 8010
$
Free List Size: 62
$
```

Figure 5: Checking free list size after boot, creating two procsses, re-checking list size, killing the two processes, final list size check

After Xv6 boots we immediately use CTRL+P to display init and sh processes, then we use CTRL+F to print out the number of processes in the `UNUSED` list. Since the max number of procs is 64, and there are 2 used processes after startup, we can conclude that 62 is the correct number of processes in the `UNUSED` list. This proves our first assertion that the `UNUSED` list is correclty initialized after boot to be true.

Next, we create 2 new background processes with `loopforever 2 &`. The program `loopforever` is a simple user program that, when given an integer N, will create N-1 children processes that will

loop forever, after which the parent itself will also loop forever. After creating the processes we use the same sequence of CTRL+P to display all used processes and CTRL+F to get the current length of the `UNUSED` list. Since, after creating 2 more processes, the length of the `UNUSED` list is 60 we can conclude that the `UNUSED` list is correctly updated after process allocation.

Lastly, we kill the newly created processes and then use CTRL+P and CTRL+F to get the state of the operation system. The bottom of Figure 5 shows that after killing the processes the size of the free list is once again 62. This demonstrates that the `UNUSED` list is being correctly updated after a process is deallocated.

Test Case: `RUNNABLE` *list, round-robin scheduling, and CTRL+R console interrupt*
Assertions:

1. `RUNNABLE` list is correctly updated to contain all `RUNNABLE` processes
2. Round-Robin scheduling is maintained
3. CTRL+R correctly shows the current order of the `RUNNABLE` list.

Status: **PASS**

```
|20:03:45|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,medi
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
$ loopforever 15 &
$
Created 14 children, starting loopforever on parent

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       11.188    0.025 sleep   12288    80103efb 80104029 80105897 8
2       sh           0          0       1       11.160    0.027 sleep   16384    80103efb 801002c4 80101913 8
4       loopforever  0          0       1       3.096     0.477 runble  12288
5       loopforever  0          0       4       3.076     0.462 runble  12288
6       loopforever  0          0       4       3.071     0.450 run     12288
7       loopforever  0          0       4       3.064     0.449 run     12288
8       loopforever  0          0       4       3.059     0.440 runble  12288
9       loopforever  0          0       4       3.041     0.419 runble  12288
10      loopforever  0          0       4       2.998     0.419 runble  12288
11      loopforever  0          0       4       2.991     0.409 runble  12288
12      loopforever  0          0       4       2.955     0.409 runble  12288
13      loopforever  0          0       4       2.948     0.380 runble  12288
14      loopforever  0          0       4       2.826     0.380 runble  12288
15      loopforever  0          0       4       2.819     0.369 runble  12288
16      loopforever  0          0       4       2.763     0.359 runble  12288
17      loopforever  0          0       4       2.708     0.359 runble  12288
18      loopforever  0          0       4       2.701     0.349 runble  12288
$
Ready List Processes:
(4) -> (18) -> (7) -> (6) -> (8) -> (9) -> (5) -> (10) -> (11) -> (13) -> (14) -> (12) -> (16)
$
Ready List Processes:
(8) -> (9) -> (5) -> (10) -> (11) -> (13) -> (14) -> (12) -> (16) -> (15) -> (17) -> (4) -> (18)
$
Ready List Processes:
(8) -> (9) -> (5) -> (10) -> (11) -> (13) -> (14) -> (12) -> (16) -> (15) -> (17) -> (4) -> (18)
$
Ready List Processes:
(10) -> (13) -> (14) -> (12) -> (16) -> (15) -> (17) -> (4) -> (18) -> (7) -> (6) -> (8) -> (9)
$
Ready List Processes:
(16) -> (15) -> (17) -> (4) -> (18) -> (7) -> (6) -> (8) -> (9) -> (5) -> (10) -> (13) -> (14)
$
Ready List Processes:
(18) -> (7) -> (6) -> (8) -> (9) -> (5) -> (10) -> (13) -> (14) -> (11) -> (12) -> (16) -> (15)
$
Ready List Processes:
(18) -> (7) -> (6) -> (8) -> (9) -> (5) -> (10) -> (13) -> (14) -> (11) -> (12) -> (16) -> (15)
$
```

Figure 6: Creating 15 background processes and holding down CTRL+R

For this test we use the command `loopforever 15 &` to instantiate 15 background processes that will increment a counter variable in an infinite loop. After we create processes we use CTRL+P to show that all our processes are either in the `RUNNABLE` or `RUNNING` states. After which, we hold down CTRL+R to display the current order of the `RUNNABLE` list. At first, the red highlighted processes are at the head of the list. Then, after the interrupt regisers again we can see

the same sequence of processes is now at the back of the list and the green highlighted processes have moved toward the front of the list. This demonstrates that the round-robin scheduling is being maintained.

Test Case: `SLEEPING` *list and CTRL+S console interrupt*
Assertions:

1. `SLEEPING` list is correctly updated when a process sleeps and is woken up.
2. CTRL+S correctly displays all `SLEEPING` processes

Status: **PASS**

```
|21:24:29|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,inde
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
$ loopforever 3 --wait &
$
Created 2 children. Kill children to wakeup parent

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       7.168     0.025 sleep   12288    80103efb 80104029 80
2       sh           0          0       1       7.142     0.026 sleep   16384    80103efb 801002c4 80
4       loopforever  0          0       1       1.159     0.047 sleep   12288    80103efb 80104029 80
5       loopforever  0          0       4       1.133     1.108 run     12288
6       loopforever  0          0       4       1.129     1.098 run     12288
$
Sleep List Processes:
(1) -> (2) -> (4)
$ kill 6; kill 5;
$
Parent Awake! Finished waiting for children! Starting loopforever on parent

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       14.182    0.025 sleep   12288    80103efb 80104029 80
2       sh           0          0       1       14.157    0.036 sleep   16384    80103efb 801002c4 80
4       loopforever  0          0       1       8.173     1.423 run     12288
$
Sleep List Processes:
(1) -> (2)
$ kill 4
$ zombie!

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       103.908   0.029 sleep   12288    80103efb 80104029 80
2       sh           0          0       1       103.882   0.046 sleep   16384    80103efb 801002c4 80
$
Sleep List Processes:
(2) -> (1)
$
```

Figure 7: Running loopforever with parent asleep, and resulting sleep list after child exits

To demonstrate that the `SLEEPING` list is being properly maintained we first run the command `loopforever 3 --wait &`. The `--wait` flag will tell the parent process to wait (and thus go to sleep) for its 2 children. We can see that after using CTRL+P and CTRL+S that the parent process, with a PID of 4, is indeed asleep, and is at the back of the sleep list. After killing the parents children the parent will be woken up and then start its infinite loop. We can see this in Figure 7, the parent process is in the 'run' state and is no longer on the sleep list processes.

Furthermore, when we kill the parent process the `init` process will be awoken to deal with the Zombie and then placed back onto the `SLEEPING` list. This is proven to be true at the bottom of Figure 7, since we can see that the init process (PID 1) is now at the back of the list. Lastly, this is also further evidence that the wait system call moves the reaped children back to the `UNUSED` list, since the child processes appear in CTRL+P interrupt.

Test Case: *Process death,* `ZOMBIE` *list and CTRL+Z interrupt*
Assertions:

1. kill() correctly moves processes to the `ZOMBIE` list
2. exit() correctly moves processes to the `ZOMBIE` list
3. init will reap orphaned children if the parent is killed
4. CTRL+Z correctly displays all processes on the `ZOMBIE` list along with their PPIDs

Status: PASS

```
|21:06:43|adupree@babbage:[xv6-pdx]> make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.im
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
$ loopforever 5 &
$
Created 4 children. Starting loopforever on parent

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       31.176    0.023 sleep   12288    80103efb 80104
2       sh           0          0       1       31.153    0.028 sleep   16384    80103efb 80100
4       loopforever  0          0       1       2.959     1.193 run     12288
5       loopforever  0          0       4       2.935     1.173 run     12288
6       loopforever  0          0       4       2.931     1.169 runble  12288
7       loopforever  0          0       4       2.924     1.169 runble  12288
8       loopforever  0          0       4       2.919     1.159 runble  12288
$
Zombie List Processes:

$ kill 5; kill 6; kill 7; kill 8;
$
PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       59.582    0.023 sleep   12288    80103efb 80104
2       sh           0          0       1       59.560    0.063 sleep   16384    80103efb 80100
4       loopforever  0          0       1       31.365    13.298run     12288
5       loopforever  0          0       4       31.341    11.961zombie  0
6       loopforever  0          0       4       31.336    11.987zombie  0
7       loopforever  0          0       4       31.330    11.984zombie  0
8       loopforever  0          0       4       31.324    12.054zombie  0
$
Zombie List Processes:
(5, 4) -> (6, 4) -> (7, 4) -> (8, 4)
$ kill 4
zombie!
$zo mbie!
zombie!
zombie!
zombie!

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       73.838    0.053 sleep   12288    80103efb 80104
2       sh           0          0       1       73.816    0.095 sleep   16384    80103efb 80100
$
```

Figure 8: Using loopforever, kill, and CTRL+Z to demonstrate process death

```
|21:24:31|adupree@babbage:[xv6-pdx]> make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 5
init: starting sh
$ loopforever 5 --child_exit &
$
Created 4 children. Starting loopforever on parent

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       27.661    0.023 sleep   12288    801
2       sh           0          0       1       27.638    0.029 sleep   16384    801
4       loopforever  0          0       1       1.401     1.389 run     12288
5       loopforever  0          0       4       1.377     0.001 zombie  0
6       loopforever  0          0       4       1.372     0.001 zombie  0
7       loopforever  0          0       4       1.364     0.001 zombie  0
8       loopforever  0          0       4       1.358     0.0000zombie  0
$
Zombie List Processes:
(5, 4) -> (6, 4) -> (7, 4) -> (8, 4)
$ kill 4
zombie!
$zombie!
 zombie!
zombie!
zombie!

PID     Name         UID        GID     PPID    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       47.131    0.043 sleep   12288    801
2       sh           0          0       1       47.107    0.043 sleep   16384    801
$
```

Figure 9: Using loopforever with the `--child_exit` option

Figure 8 shows we test that the `ZOMBIE` list is being correctly updated after using kill, and that orphans are being properly handled. At the top of Figure 8 we use `loopforever 5 &` to start 5 background processes (1 parent and four children), then we use CTRL+P to show the processes and their states and then CTRL+Z to display the zombie. As expected, since there are no zombie processes yet, CTRL+Z doesn't displa any information. Next, we kill all the child processes, and then use CTRL+P again. As expected kill moves the process onto the `ZOMBIE` list. Furthermore, when we use CTRL+Z all the killed processes are displayed, along with their parents process ID. Lastly, we kill the parent process which will wake up `init` to reap the process and its children causing it to print `"zombie!"` one time for each of the reaped processes.

Figure 9 performs the same test as Figure 8, however we use loopforever with the `--child_exit` option. This will direct the forked child processes to use the `exit()` system call instead of execute an infinite loop. As such, we can see that `exit()` does properly place the processes on the `ZOMBIE` list.