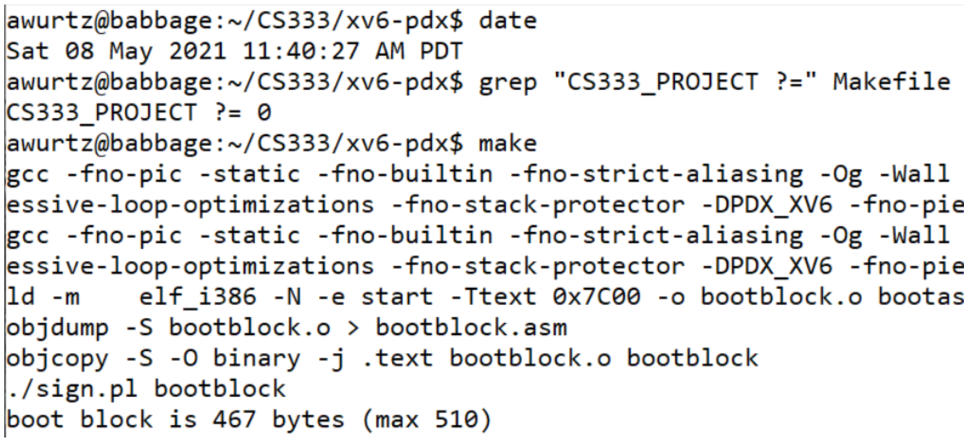
Compilation Test

This section will show that my project 3 code compiles and runs `usertests` to completion with all tests passed with the project flag set to "3" and also set to "0".

Subtest 1: Compile with `CS333_PROJECT` set to 0.

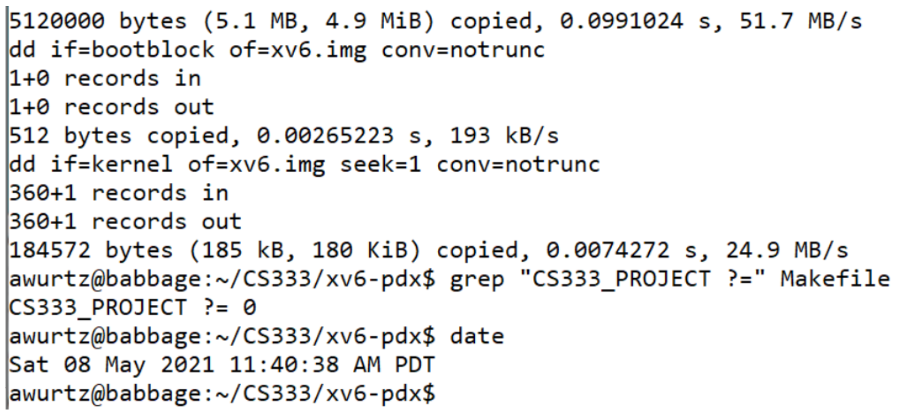
Since the listing is so long, this will require two screen shots. In the first screen shot, the current date and time is displayed as well as the value for the `CS333_PROJECT`, verifying that it is set to 0. The second screen shot displays the same information. This is used to show that the two screen shots are from the same compilation sequence.

The expected outcome is that `xv6` will correctly compile with the project flag set to "0".



```
awurtz@babbage:~/CS333/xv6-pdx$ date
Sat 08 May 2021 11:40:27 AM PDT
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
awurtz@babbage:~/CS333/xv6-pdx$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Og -Wall
essive-loop-optimizations -fno-stack-protector -DPDX_XV6 -fno-pie
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Og -Wall
essive-loop-optimizations -fno-stack-protector -DPDX_XV6 -fno-pie
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootas
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
boot block is 467 bytes (max 510)
```

Figure 1: Compilation with `CS333_PROJECT` set to 0.



```
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0991024 s, 51.7 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00265223 s, 193 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
360+1 records in
360+1 records out
184572 bytes (185 kB, 180 KiB) copied, 0.0074272 s, 24.9 MB/s
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
awurtz@babbage:~/CS333/xv6-pdx$ date
Sat 08 May 2021 11:40:38 AM PDT
awurtz@babbage:~/CS333/xv6-pdx$
```

Figure 2: Compilation with CS333_PROJECT set to 0.

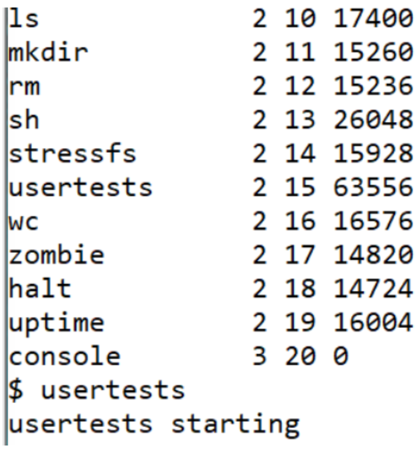
As expected, you can see that xv6 correctly compiles with the project flag set to 0. The date print outs show that these two screen shots are from the same compilation and the grep commands show that the project flag was, in fact, set to 0.

This subtest **PASSES**.

Subtest 2: Run `usertests` with `CS333_PROJECT` set to 0.

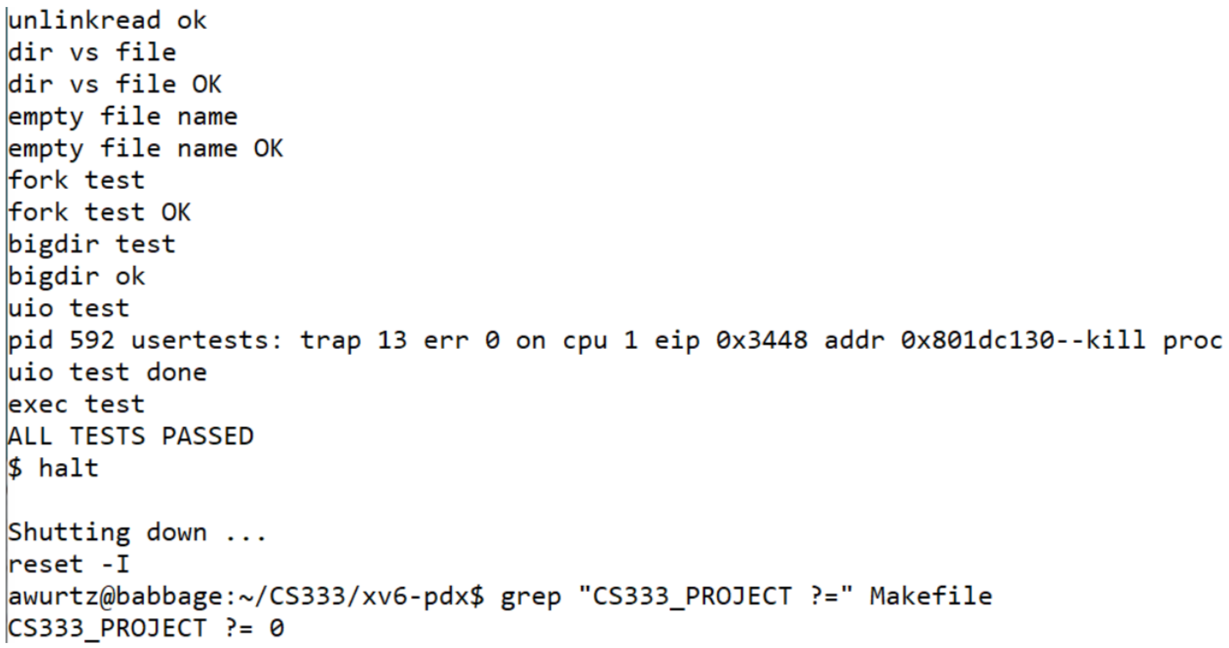
In this test, I compiled and ran xv6 with the CS333_PROJECT flag set to 0 and then ran usertests. Since usertests is so long, this will require two screenshots. The first will show ls and usertests starting. The second will shows that all usertests passed and a grep command showing that the CS333_PROJECT flag is set to 0.

The expected outcome is that xv6 will successfully run usertests with all tests passed.



```
ls             2 10 17400
mkdir          2 11 15260
rm             2 12 15236
sh             2 13 26048
stressfs       2 14 15928
usertests      2 15 63556
wc             2 16 16576
zombie         2 17 14820
halt           2 18 14724
uptime         2 19 16004
console        3 20 0
$ usertests
usertests starting
```

Figure 3: Running usertests with CS333_PROJECT set to "0"



```
unlinkread ok
dir vs file
dir vs file OK
empty file name
empty file name OK
fork test
fork test OK
bigdir test
bigdir ok
uio test
pid 592 usertests: trap 13 err 0 on cpu 1 eip 0x3448 addr 0x801dc130--kill proc
uio test done
exec test
ALL TESTS PASSED
$ halt

Shutting down ...
reset -I
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
```

Figure 4: Running usertests with CS333_PROJECT Set to "0"

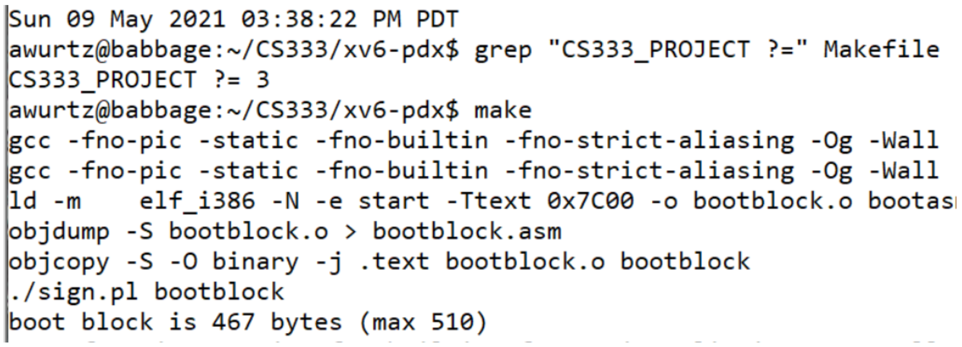
The first screenshot shows that usertests in starting, and that this version of xv6 does not have the usertests that are available when the project flag is set > 0. The second screenshot shows that all tests were passed and that grep command shows that the project flag is set to “0” is the Makefile.

This subtest **PASSES**.

Subtest 3: Compile with `CS333_PROJECT` set to 3.

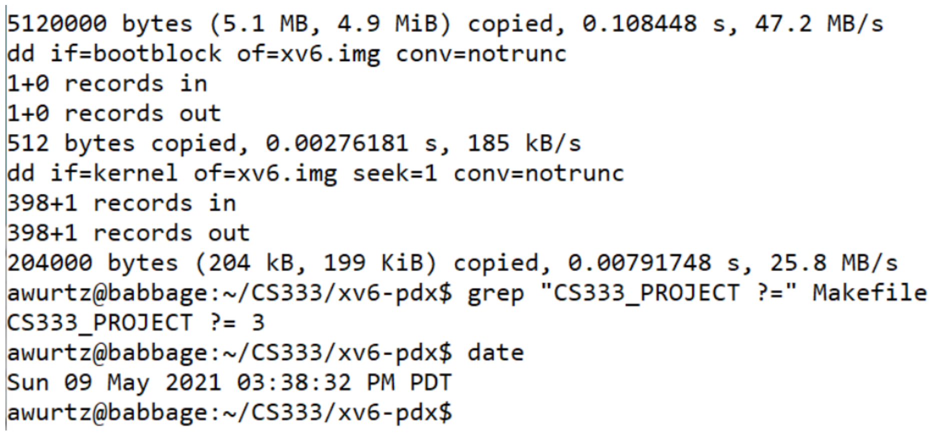
Since the listing is so long it will require two screen shots. In the first screen shot, the current date and time is displayed as well as the value for the `CS333_PROJECT`, verifying that it is set to 3. The second screen shot displays the same information. This is used to show that the two screen shots are from the same compilation sequence.

The expected outcome is that `xv6` will correctly compile with the project flag set to “3”.



```
Sun 09 May 2021 03:38:22 PM PDT
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 3
awurtz@babbage:~/CS333/xv6-pdx$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Og -Wall
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Og -Wall
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootas
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
boot block is 467 bytes (max 510)
```

Figure 5: Compiling xv6 with CS333_PROJECT set to "3"



```
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.108448 s, 47.2 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00276181 s, 185 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
398+1 records in
398+1 records out
204000 bytes (204 kB, 199 KiB) copied, 0.00791748 s, 25.8 MB/s
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 3
awurtz@babbage:~/CS333/xv6-pdx$ date
Sun 09 May 2021 03:38:32 PM PDT
awurtz@babbage:~/CS333/xv6-pdx$
```

Figure 6: Compiling xv6 with CS333_PROJECT set to "3"

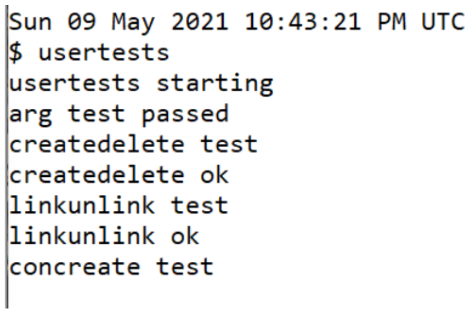
As expected, xv6 successfully compiles with CS333_PROJECT set to “3”. The first screenshot shows the date, project flag value, and the make command. The seconds screenshot shows that xv6 has successfully compiled, as well as the date and the grep command showing that both screenshots are from the same compilation.

This subtest **PASSES**.

Subtest 4: Run `usertests` with `CS333_PROJECT` set to 3.

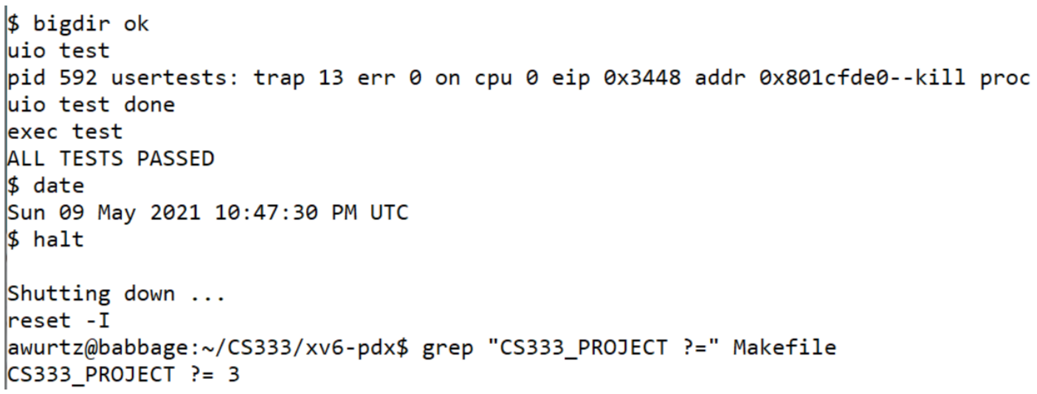
This test will require two screenshots, since the output of usertests is so long. The first will show the date and usertests beginning to run in xv6. The second will shows that all tests have passed, and print the date again, as well as a grep command showing the value of CS333_PROJECT flag, to show that these two screen shots are from the same instance of xv6.

The expected outcome of this test is that all usertests will pass with the CS333_PROJECT flag set to "3".



```
Sun 09 May 2021 10:43:21 PM UTC
$ usertests
usertests starting
arg test passed
createdelete test
createdelete ok
linkunlink test
linkunlink ok
concreate test
```

Figure 7: Running usertests with CS333_PROJECT set to "3"



```
$ bigdir ok
uio test
pid 592 usertests: trap 13 err 0 on cpu 0 eip 0x3448 addr 0x801cfde0--kill proc
uio test done
exec test
ALL TESTS PASSED
$ date
Sun 09 May 2021 10:47:30 PM UTC
$ halt

Shutting down ...
reset -I
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 3
```

Figure 8: Running usertests with CS333_PROJECT set to "3"

As expected, all usertests passed when the CS333_PROJECT flag was set to "3". The first screenshot shows the tests starting and the second screenshot shows that all tests were successful. The grep commands shows that the project flag was, in fact, set to 3, and the date command in each screen shot shows that that these two shots are from the same instance of xv6/usertests since they were taken relatively close together (and usertests takes a long time to run).

This subtest **PASSES**.

Since all four subtests pass, this test **PASSES**.

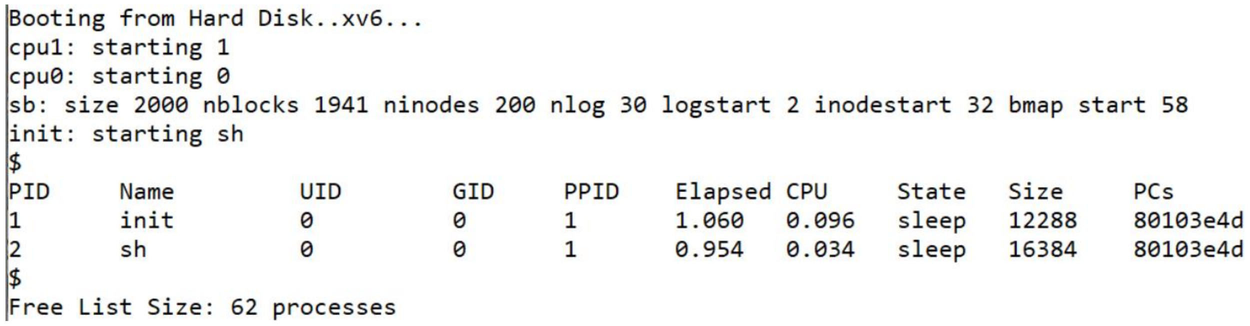
`UNUSED` List Tests

This section will show that the `UNUSED` list is correctly initialized after `xv6` is fully booted and that the `UNUSED` list is correctly updated when a process is allocated and deallocated.

Subtest 1: This test will show that the UNUSED list is correctly initialized after xv6 is fully booted.

When xv6 boots, there are two active processes (init and sh). This means that there should be 62 processes on the UNUSED list when xv6 boots. This test shows that the UNSED list is correctly initialized using the ctrl-f command once xv6 has fully booted.

The expected outcome of this test is that xv6 will show 62 processes on the UNUSED list and two active processes, init and sh, when it is fully booted.



```
Booting from Hard Disk..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        1.060   0.096   sleep   12288    80103e4d
2       sh           0         0       1        0.954   0.034   sleep   16384    80103e4d
$
Free List Size: 62 processes
```

Figure 9: Free List and Active Processes when Xv6 is Fully Booted

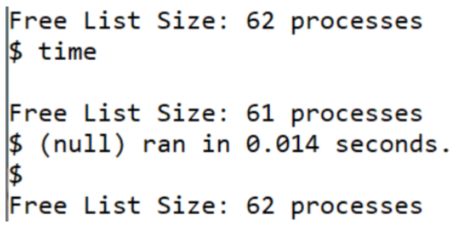
As expected, this test shows that the UNUSED list is properly initialized when xv6 is fully booted. The screenshot shows that the total number of processes running is 64, including init and the shell, plus 62 unused processes in the UNUSED list.

This subtest **PASSES**.

Subtest 2: This test shows that the UNUSED list is correctly updated when a process is allocated and deallocated.

This tests prints the number of processes on the UNSUED list, then runs time. It prints the number of processes on the UNUSED list while time is running and the prints the number of UNUSED processes again when it has finished executing.

The expected outcome of this test is that the number of UNUSED processes will decrease by one while the time command is running, and that it will increase by one when the time command finishes running. The number of unused processes before and after time runs should be the same.



```
Free List Size: 62 processes
$ time

Free List Size: 61 processes
$ (null) ran in 0.014 seconds.
$
Free List Size: 62 processes
```

Figure 10: UNUSED List Before, During, and After "time" Command

As expected, figure 10 shows that the number of unused processes decreased from 62 to 61 while the time command was running and increased to 62 to after time had finished. This shows that the UNUSED list is properly updated when new processes run and complete.

This subtest **PASSES**.

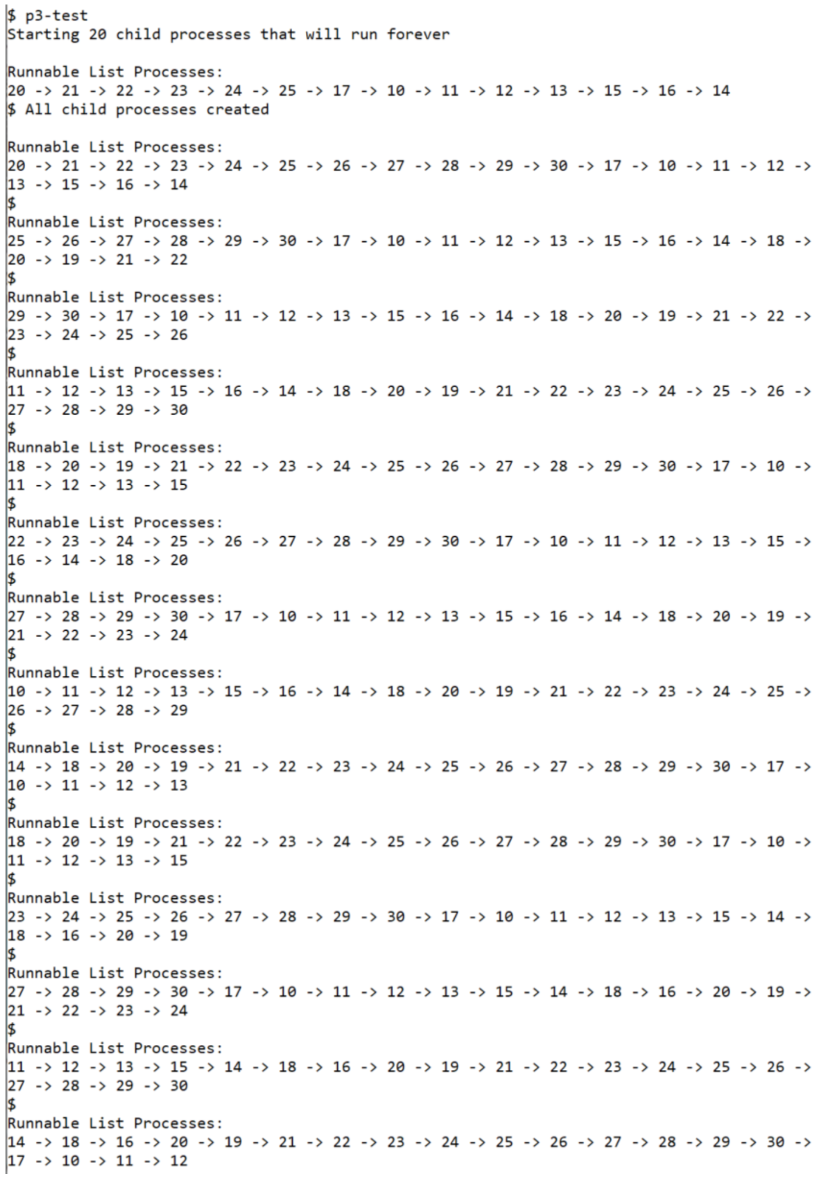
Since both subtests pass, this test **PASSES**.

Round-Robin Scheduling

This section will show that round-robin scheduling is maintained in Project 3.

This test will show that round-robin scheduling is maintained by printing the RUNNABLE list continuously while running the p3-test program. The p3-test program creates 20 child processes and has then run in an infinite loop. This means that a continuously updated RUNNABLE list will show that the processes are being scheduled using a FIFO or round-robin system.

The expected output of this test is that the items at the front RUNNABLE list will be scheduled first and that processes will be added to the end of the RUNNABLE list after their timeslice expires.



```
$ p3-test
Starting 20 child processes that will run forever

Runnable List Processes:
20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 -> 16 -> 14
$ All child processes created

Runnable List Processes:
20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 -> 10 -> 11 -> 12 ->
13 -> 15 -> 16 -> 14
$
Runnable List Processes:
25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 -> 16 -> 14 -> 18 ->
20 -> 19 -> 21 -> 22
$
Runnable List Processes:
29 -> 30 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 -> 16 -> 14 -> 18 -> 20 -> 19 -> 21 -> 22 ->
23 -> 24 -> 25 -> 26
$
Runnable List Processes:
11 -> 12 -> 13 -> 15 -> 16 -> 14 -> 18 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 ->
27 -> 28 -> 29 -> 30
$
Runnable List Processes:
18 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 -> 10 ->
11 -> 12 -> 13 -> 15
$
Runnable List Processes:
22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 ->
16 -> 14 -> 18 -> 20
$
Runnable List Processes:
27 -> 28 -> 29 -> 30 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 -> 16 -> 14 -> 18 -> 20 -> 19 ->
21 -> 22 -> 23 -> 24
$
Runnable List Processes:
10 -> 11 -> 12 -> 13 -> 15 -> 16 -> 14 -> 18 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 ->
26 -> 27 -> 28 -> 29
$
Runnable List Processes:
14 -> 18 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 ->
10 -> 11 -> 12 -> 13
$
Runnable List Processes:
18 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 -> 10 ->
11 -> 12 -> 13 -> 15
$
Runnable List Processes:
23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 -> 14 ->
18 -> 16 -> 20 -> 19
$
Runnable List Processes:
27 -> 28 -> 29 -> 30 -> 17 -> 10 -> 11 -> 12 -> 13 -> 15 -> 14 -> 18 -> 16 -> 20 -> 19 ->
21 -> 22 -> 23 -> 24
$
Runnable List Processes:
11 -> 12 -> 13 -> 15 -> 14 -> 18 -> 16 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 ->
27 -> 28 -> 29 -> 30
$
Runnable List Processes:
14 -> 18 -> 16 -> 20 -> 19 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 ->
17 -> 10 -> 11 -> 12
```

Figure 11: RUNNABLE Process List while P3-test is Running

As expected, Figure 12 shows the processes moving on and off of the RUNNABLE list in a first-in, first-out pattern. Due to latency and the lack of precision of control sequences, there are some gaps in the print statements, but it is very clear that the processes are moving through the queue in chronological order, and that they are being added back onto the end. While not all of the PIDs are in order, due to the nature of forking children and process interleaving, they continue to cycle through the RUNNABLE list in the same order as they move in and out of the CPUs in a round-robin fashion.

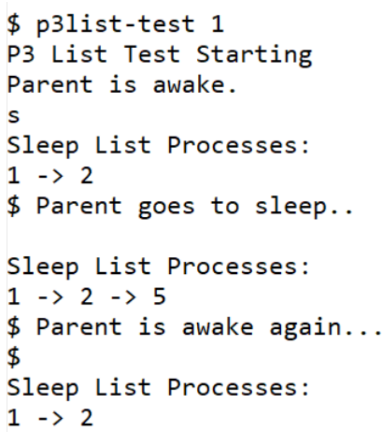
This test **PASSES**.

`SLEEPING` List Test

This section shows that the `SLEEPING` list is correctly updated when a process sleeps and is woken up.

This test uses the test program p3list-test to show that processes are properly added to and removed from the SLEEPING list. This program creates a child process, sleeps, calls wait to collect the exited child, and then runs through a long loop with an empty print statement. This gives me time to hit ctrl-s at each stage in the process.

The expected outcome of this test is that a new process will appear on the SLEEPING list when the test program says that the process is sleeping, and that the process will be removed when the test program says that it wakes up.



```
$ p3list-test 1
P3 List Test Starting
Parent is awake.
s
Sleep List Processes:
1 -> 2
$ Parent goes to sleep..

Sleep List Processes:
1 -> 2 -> 5
$ Parent is awake again...
$
Sleep List Processes:
1 -> 2
```

Figure 12: SLEEPING List During p3list-test

As expected, Figure 12 shows the process 5 (the parent process for the test program) moves onto the SLEEPING list when it says it is sleeping, and it is removed from the SLEEPING list when it says it is awake. This shows that the SLEEPING list is properly updated, as expected.

This test **PASSES**.

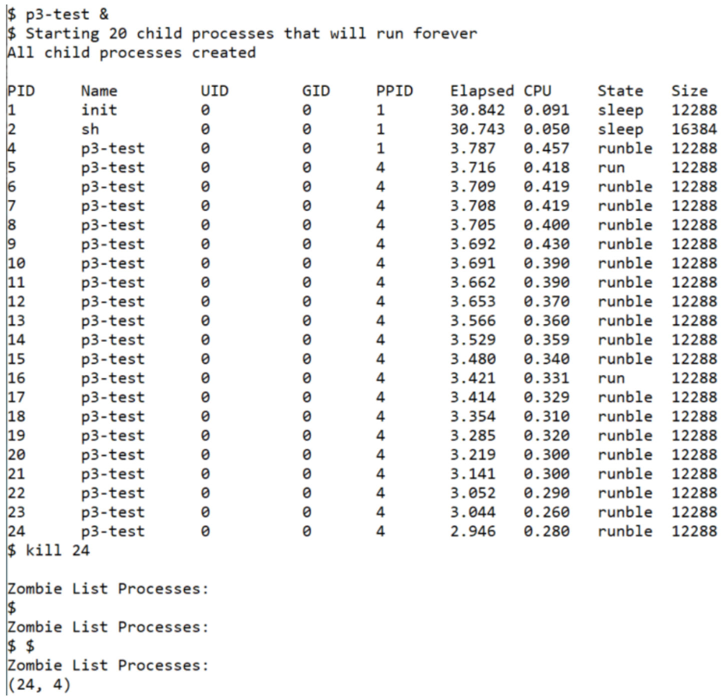
Process Death Tests

This section shows that the `ZOMBIE` and `UNUSED` lists are correctly managed by the `kill()`, `exit()`, and `wait()` system calls.

Subtest 1: Show that the kill() system call correctly causes a process to move to the ZOMBIE list.

This test will show that a process killed using the kill() system call is moved to the ZOMBIE list. It uses p3-test and the kill user program (which uses the kill system call) to demonstrate this.

The expected outcome is that once a process is killed, it will be appear on the ZOMBIE list.



```
$ p3-test &
$ Starting 20 child processes that will run forever
All child processes created

PID     Name        UID         GID     PPID    Elapsed CPU     State   Size
1       init        0           0       1       30.842  0.091   sleep   12288
2       sh          0           0       1       30.743  0.050   sleep   16384
4       p3-test     0           0       1       3.787   0.457   runble  12288
5       p3-test     0           0       4       3.716   0.418   run     12288
6       p3-test     0           0       4       3.709   0.419   runble  12288
7       p3-test     0           0       4       3.708   0.419   runble  12288
8       p3-test     0           0       4       3.705   0.400   runble  12288
9       p3-test     0           0       4       3.692   0.430   runble  12288
10      p3-test     0           0       4       3.691   0.390   runble  12288
11      p3-test     0           0       4       3.662   0.390   runble  12288
12      p3-test     0           0       4       3.653   0.370   runble  12288
13      p3-test     0           0       4       3.566   0.360   runble  12288
14      p3-test     0           0       4       3.529   0.359   runble  12288
15      p3-test     0           0       4       3.480   0.340   runble  12288
16      p3-test     0           0       4       3.421   0.331   run     12288
17      p3-test     0           0       4       3.414   0.329   runble  12288
18      p3-test     0           0       4       3.354   0.310   runble  12288
19      p3-test     0           0       4       3.285   0.320   runble  12288
20      p3-test     0           0       4       3.219   0.300   runble  12288
21      p3-test     0           0       4       3.141   0.300   runble  12288
22      p3-test     0           0       4       3.052   0.290   runble  12288
23      p3-test     0           0       4       3.044   0.260   runble  12288
24      p3-test     0           0       4       2.946   0.280   runble  12288
$ kill 24

Zombie List Processes:
$
Zombie List Processes:
$ $
Zombie List Processes:
(24, 4)
```

Figure 13: Killing Process 24 in p3-test

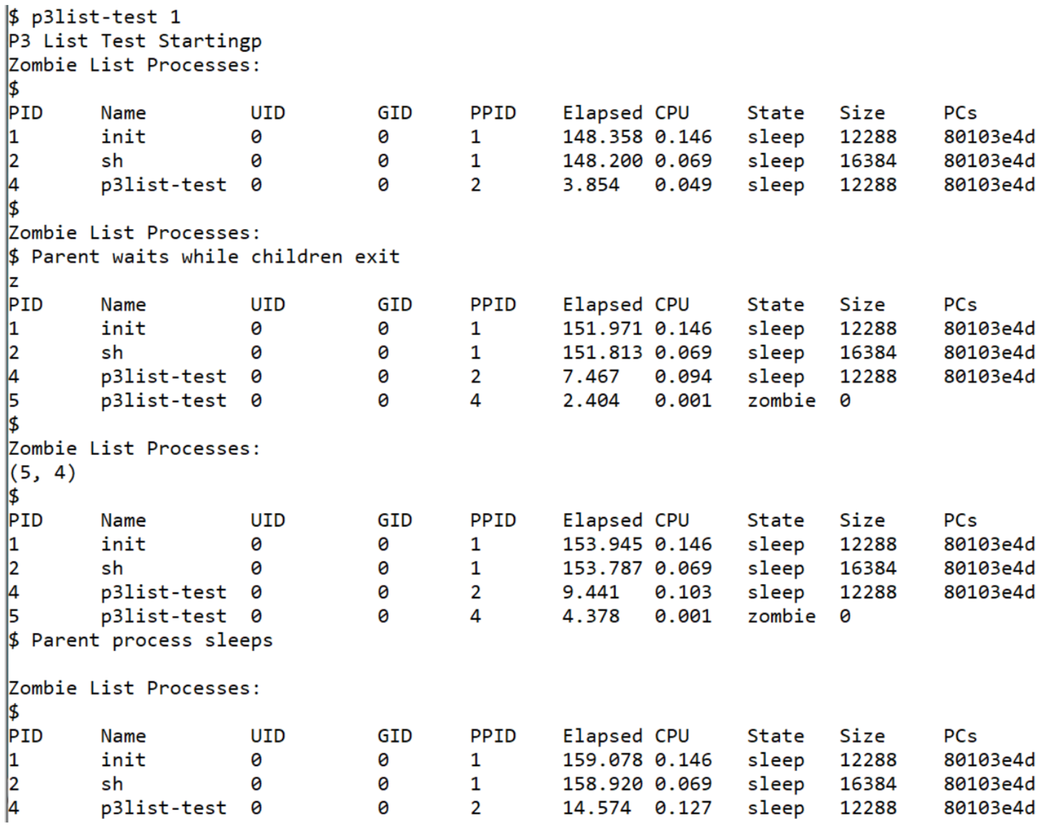
Figure 13 shows p3-test running many child processes in the background. After process 24 is killed, it appear on the the ZOMBIE process list, as expected. This shows that the kill() system call correctly causes the killed process to move to the ZOMBIE list until it is reaped by its parent.

This subtest **PASSES**.

Subtest 2: Show that the `exit()` system call correctly caused a process to move to the `ZOMBIE` list.

For this subtest, I ran the test program p3list-test with the command line argument 1. When `argv[1]` is 1, this test program forks once to create a single child process. That child process immediately calls `exit()`, which should move it to the `ZOMBIE` list. The parent process sleeps and then calls `wait()`. While the parent is sleeping, ctrl-z should show that the child process is in the ZOMBIE list, waiting to be reaped by its parent. After the parent wakes up and runs wait, the child process will have been moved to the `UNUSED` list and will no longer appear on the `ZOMBIE` list.

The expected result for this test is that the `ZOMBIE` list will be empty when p3list-test starts, that the child process will appear on the `ZOMBIE` list while the parent is `SLEEPING`. When parent process wakes up, it calls `wait()` and will immediately reap the child process (which will then be removed from the `ZOMBIE` list and added the `UNUSED` list).



```
$ p3list-test 1
P3 List Test Startingp
Zombie List Processes:
$
PID     Name         UID        GID     PPID    Elapsed CPU     State   Size     PCs
1       init         0          0       1       148.358 0.146   sleep   12288    80103e4d
2       sh           0          0       1       148.200 0.069   sleep   16384    80103e4d
4       p3list-test  0          0       2       3.854   0.049   sleep   12288    80103e4d
$
Zombie List Processes:
$ Parent waits while children exit
z
PID     Name         UID        GID     PPID    Elapsed CPU     State   Size     PCs
1       init         0          0       1       151.971 0.146   sleep   12288    80103e4d
2       sh           0          0       1       151.813 0.069   sleep   16384    80103e4d
4       p3list-test  0          0       2       7.467   0.094   sleep   12288    80103e4d
5       p3list-test  0          0       4       2.404   0.001   zombie  0
$
Zombie List Processes:
(5, 4)
$
PID     Name         UID        GID     PPID    Elapsed CPU     State   Size     PCs
1       init         0          0       1       153.945 0.146   sleep   12288    80103e4d
2       sh           0          0       1       153.787 0.069   sleep   16384    80103e4d
4       p3list-test  0          0       2       9.441   0.103   sleep   12288    80103e4d
5       p3list-test  0          0       4       4.378   0.001   zombie  0
$ Parent process sleeps

Zombie List Processes:
$
PID     Name         UID        GID     PPID    Elapsed CPU     State   Size     PCs
1       init         0          0       1       159.078 0.146   sleep   12288    80103e4d
2       sh           0          0       1       158.920 0.069   sleep   16384    80103e4d
4       p3list-test  0          0       2       14.574  0.127   sleep   12288    80103e4d
```

Figure 14: Ctrl-p and Ctrl-Z Output while p3list-test is Running.

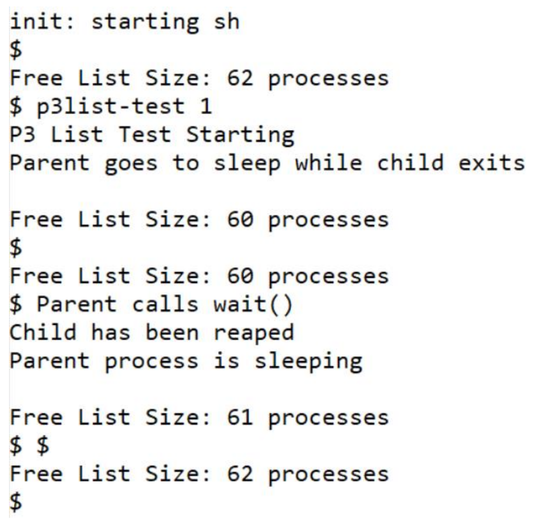
As expected, this test shows the child process (PID = 5) being added to the ZOMBIE list while the parent process sleeps. When the parent process wakes it up calls wait, reaping the child, and then goes back to sleep, as expected. Process 5 is a can be seen as a zombie in middle ctrl-p and ctrl-z printouts, but it has been removed from the ZOMBIE list in the final ctrl-z and ctrl-p print outs.

This subtest **PASSES**.

Subtest 3: Show that the wait() system call correctly causes a process to move to the UNUSED list.

This test will show that the wait() system call correctly causes a process to move the UNUSED list using the test program p3list-test. In this test program, a child is created and then exits. The parent then calls wait() in order to reap the child. After reaping the child, the parent sleeps. This test will show that the number of UNUSED processes decreased while the child is running and increases again when the child is reaped.

The expected outcome of this test is that the number of processes in the UNUSED list will decrease by two when the test starts running, decrease by one when the parent calls wait and reaps the child process, and decrease by one again when the parent process exits.



```
init: starting sh
$
Free List Size: 62 processes
$ p3list-test 1
P3 List Test Starting
Parent goes to sleep while child exits

Free List Size: 60 processes
$
Free List Size: 60 processes
$ Parent calls wait()
Child has been reaped
Parent process is sleeping

Free List Size: 61 processes
$ $
Free List Size: 62 processes
$
```

Figure 15: UNSUED List When Parent Calls Wait to Reap Child Process

Figure 15 shows that the UNUSED list changes as expected. Before running the test program there are 62 processes on the list. After the program begins there are only 60. That is because the test program creates two processes, the parent and the child. Figure 15 shows that after the parent process calls wait and reaps its child (but before the parent exits) there are 61 UNUSED processes. This shows that the wait() system call successfully moves the child process to the UNSUED list.

This subtest **PASSES.**

Since all 3 subtests pass, this test **PASSES**.