Project 4 Test Report

Alexander DuPree

March 7, 2020

Introduction

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

```
|13:17:19|adupree@babbage:[xv6-pdx]> grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
|13:17:47|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, 0.147209 s, 34.8 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.0150478 s, 34.0 kB/s
317+1 records in
317+1 records out
162804 bytes (163 kB, 159 KiB) copied, 0.0127893 s, 12.7 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 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 0x801dc130--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 4 in the Makefile*
Assertions:

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

Status: **PASS**

```
|13:21:43|adupree@babbage:[xv6-pdx]> grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 4
|13:21:45|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 _loopforever _p2-test _testsetuid _testuidgid _p4-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 975 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, 0.137409 s, 37.3 MB/s
1+0 records in
1+0 records out
512 bytes copied, 0.00900375 s, 56.9 kB/s
362+1 records in
362+1 records out
185664 bytes (186 kB, 181 KiB) copied, 0.0138267 s, 13.4 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
createdelete ok
linkunlink test
linkunlink ok
```

Figure 3: Compilation and boot with CS333_PROJECT set to 4 and execution of `usertests`.

```
fork test
fork test OK
bigdir test
bigdir ok
uio test
pid 591 usertests: trap 13 err 0 on cpu 1 eip 0x33ce addr 0x0--kill proc
uio test done
exec test
ALL TESTS PASSED
$
```

Figure 4: Completion of `usertests` with output elided, CS333_P4 is defined

The command `grep "CS333_PROJECT ?=" Makefile` shows that the CS333_PROJECT macro is indeed set to 4. 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.

Updated Commands

This section presents all tests related to the new information displayed for the `ps` program, CTRL+P and CTRL+R interrupts. Test cases follow closely those outlined in the rubric.

Test Case: *CTRL+P,* `ps` *and CTRL+R*
Note: `MAXPRIO` = 4, `DEFAULT_BUDGET` = 10000, `TICKS_TO_PROMOTE` = 100000
Assertions:

1. CTRL+P correctly displays all active processes and their priorities
2. `ps` output matches CTRL+P, excluding program counters
3. CTRL+R displays all ready lists, from highest to lowest priority, and the budget for each process

Status: **PASS**

```
10:44:17 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
$ loopforever 5 &
$
Created 4 children. Starting loopforever on parent

PID     Name         UID        GID     PPID    Prio    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       4       8.211     0.029 sleep   12288    8010430e 80104424 80105ebc 801052ad 801062e3 801061d9
2       sh           0          0       1       4       8.182     0.036 sleep   16384    8010430e 801002c4 80101913 80100f3f 801055e0 801052ad
4       loopforever  0          0       1       4       1.090     0.459 runble  12288
5       loopforever  0          0       4       4       1.064     0.436 runble  12288
6       loopforever  0          0       4       4       1.060     0.429 run     12288
7       loopforever  0          0       4       4       1.042     0.409 run     12288
8       loopforever  0          0       4       4       1.028     0.400 runble  12288
$ ps

PID     Name         UID        GID     PPID    Prio    Elapsed   CPU   State   Size
1       init         0          0       1       4       10.337    0.029 sleep   12288
2       sh           0          0       1       4       10.310    0.054 sleep   16384
4       loopforever  0          0       1       4       3.217     1.298 runble  12288
5       loopforever  0          0       4       4       3.192     1.276 runble  12288
6       loopforever  0          0       4       4       3.186     1.276 runble  12288
7       loopforever  0          0       4       4       3.169     1.249 run     12288
8       loopforever  0          0       4       4       3.156     1.231 runble  12288
9       ps           0          0       2       4       0.297     0.025 run     49152
$
Ready List Processes:

Priority 4: (6, 7954) -> (7, 7949) -> (8, 8009)

Priority 3:

Priority 2:

Priority 1:

Priority 0:
$
```

Figure 5: Output of CTRL+P, CTRL+R and the `ps` program

The first command we run is `loopforever 5 &` to get 5 processes that will take turns on the ready list in the background. Then we use the CTRL+P interrupt, as we can see in figure 5 it properly displays the priority of each process. In this case the budget is set so high that each process will stay in the `MAXPRIO` queue. Then we run the `ps` program, which matches the output of CTRL+P. Finally, we use CTRL+R to display the ready lists. In figure 5, three of the five background processes (the other 2 are running) are on the max priority ready list and their budgets are all less than the `DEFAULT_BUDGET` which was 10000.

Set and Get Priority

This section presents all tests related to the set and get priority system calls. These tests make use of the `setpriority` and `getpriority` user programs which simply take the command line args, forward them to the correct system call and report the pass/fail status. Test cases follow closely those outlined in the rubric.

Test Case: *getpriority*
Note: `MAXPRIO` = 3, `DEFAULT_BUDGET` = 100000, `TICKS_TO_PROMOTE` = 1000000 I.E. No promotions or demotions during this test.
Assertions:

1. Retrieves correct priority for the relevant process
2. Returns an error PID is not found

Status: **PASS**

```
|16:58:41|adupree@babbage:[xv6-pdx]> make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.
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    Prio    Elapsed   CPU   State   Size     PCs
1       init         0          0       1       3       1.866     0.023 sleep   12288    8010
2       sh           0          0       1       3       1.840     0.014 sleep   16384    8010
$ getpriority 2
PID: 2, PRIORITY: 3
$ getpriority 123

getpriority: ERROR, failed to get priority
$ getpriority -1

getpriority: ERROR, failed to get priority
$
```

Figure 6: Getting priority of processes

Because the On startup we use CTRL+P to display the current processes. As we can see the priority of the `sh` process is 3. Calling `getpriority 2` returns the expected value of 3. Furthermore, attempting to call `getpriority` with PID's that don't exist or are out of bounds results in an error.

Test Case: *setpriority*
Note: `MAXPRIO` = 3, `DEFAULT_BUDGET` = 100000, `TICKS_TO_PROMOTE` = 1000000
Assertions:

1. setpriority changes the priority and resets the budget
2. Setting the priority of a process on a ready list to the same priority it already has does not change the position in the list for that process
3. Calling setpriority() with an invalid PID and/or priority returns an error and leaves the process priority and budget unmodified

Status: **PASS**

```
|17:53:16|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=
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 10 &
$
Created 9 children. Starting loopforever on parent

Ready List Processes:

Priority 3: (5, 99477) -> (4, 99464) -> (7, 99507) -> (6, 99481) -> (8, 99500) -> (9, 99519) -> (10, 99520) -> (11, 99547)

Priority 2:

Priority 1:

Priority 0:
$ setpriority 7 3; setpriority 9 0
PID: 7, PRIORITY: 3
PID: 9, PRIORITY: 0
$
Ready List Processes:

Priority 3: (13, 96643) -> (5, 96548) -> (4, 96532) -> (8, 96574) -> (6, 96552) -> (7, 96593) -> (11, 96630)

Priority 2:

Priority 1:

Priority 0: (9, 100000)
$ setpriority 123 -1

setpriority: ERROR, failed to set new priority
$
Ready List Processes:

Priority 3: (8, 94900) -> (4, 94846) -> (6, 94858) -> (10, 94923) -> (7, 94903) -> (12, 94926) -> (11, 94944)

Priority 2:

Priority 1:

Priority 0: (9, 100000)
$
```

Figure 7: Output of CTRL+P, CTRL+R and the `ps` program

The first command we run is `loopforeer 10 &`, which instantiates 10 processes that'll loop forever in the background. Next we use CTRL+R to display the current state of the ready lists. Since the `DEFAULT_BUDGET` and `TICKS_TO_PROMOTE` are set so high, the MLFQ will essentially run in round robin order with the max priority queue. Then we call `setpriority 7 3`, which attempts to set process `7` to its current priority, and `setpriority 9 0` which will demote process `9` to the lowest priority. The subsequent use of CTRL+R shows that prorcess `9` was indeed set

to 0 and it's budget was reset to the default value. We can also see that process `7` did not change lists and its budget was not reset (since its budget is roughly the same as everyone elses in the queue). Lastly, we call `setpriority 123 -1` to generate an error and CTRL+R to show that processes did not have their priority or budget arbitrarily modified.

Multi-Level-Feedback Queue

This section presents all tests related to operation of the Multi-Level-Feedback Queue scheduler algorithm. Test cases follow closely those outlined in the rubric.

Test Case: *With MAXPRIO set to 0*
Note: `MAXPRIO` = 0, `DEFAULT_BUDGET` = 1000, `TICKS_TO_PROMOTE` = 10000
Assertions:

1. Scheduler operates as a single round-robin queue
2. `setpriority` for any value other than 0 fails
3. No promotion of demotion occurs

Status: **PASS**

First, the command `grep "DEFAULT_BUDGET|MAXPRIO|TICKS_TO_PROMOTE" pdx.h` is performed to verify that the kernel constants are indeed set to 1000, 0, and 10000 respectively. On kernel boot the first thing we do is run `loopforever 10 &` to get 10 runnable processes to loop forever in the background. Then, we hold down the CTRL+R interrupt to display how the MLFQ scheduler is utilizing the ready lists. First we notice that since `MAXPRIO` is 0 there is only one ready list. Because there is only one ready list the scheduler acts as a single round-robin queue with no promotion or demotions occuring. The highlighted sections of Figure 8 below presents chains of processes that move from the front of the queue to the back of the queue and maintain their order which is evidence of the round-robin algorithm. Furthermore, we can see the budgets bottom out at or below a zero value and are no longer updated. Because no promotion/demotion occurs the budgets for each value are no longer reset. Lastly, in Figure 9 we attempt to set the priority of process `10` in three different ways. Attempting to set the priority to any value other than 0 caused an error.

```
|10:30:39|adupree@babbage:[xv6-pdx]> grep "DEFAULT_BUDGET\|MAXPRIO\|TICKS_TO_PROMOTE" pdx.h
#define MAXPRIO 0
#define DEFAULT_BUDGET 1000
#define TICKS_TO_PROMOTE 10000
|10:30:41|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,media
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 10 &
$
Created 9 children. Starting loopforever on parent

Ready List Processes:

Priority 0: (7, 590) -> (8, 600) -> (5, 563) -> (9, 629) -> (10, 641) -> (11, 661) -> (12, 660) -> (13, 678)
$
Ready List Processes:

Priority 0: (4, 324) -> (6, 341) -> (7, 360) -> (8, 369) -> (5, 333) -> (9, 400) -> (10, 411) -> (11, 431)
$
Ready List Processes:

Priority 0: (7, 230) -> (8, 237) -> (9, 273) -> (5, 204) -> (10, 281) -> (11, 301) -> (12, 300) -> (13, 316)
$
Ready List Processes:

Priority 0: (4, 85) -> (6, 103) -> (7, 120) -> (8, 127) -> (9, 163) -> (5, 92) -> (10, 172) -> (11, 191)
$
Ready List Processes:

Priority 0: (7, 21) -> (8, 30) -> (9, 63) -> (5, -9) -> (10, 72) -> (11, 92) -> (13, 108) -> (12, 91)
$
Ready List Processes:

Priority 0: (13, 18) -> (12, 1) -> (6, -6) -> (4, -8) -> (7, -9) -> (8, 0) -> (9, -7) -> (5, -9)
$
Ready List Processes:

Priority 0: (7, -9) -> (8, 0) -> (9, -7) -> (5, -9) -> (10, -8) -> (11, -8) -> (13, -2) -> (12, -9)
$
Ready List Processes:

Priority 0: (13, -2) -> (12, -9) -> (6, -6) -> (4, -8) -> (7, -9) -> (8, 0) -> (5, -9) -> (9, -7)
$
Ready List Processes:

Priority 0: (10, -8) -> (11, -8) -> (13, -2) -> (12, -9) -> (6, -6) -> (4, -8) -> (7, -9) -> (8, 0)
$
```

Figure 8: Holding down CTRL+R with `MAXPRIO` set to 0

```
Priority 0: (10, -8) -> (11, -8) -> (13, -2) -> (12, -9) -> (6, -6) -> (4, -8) -> (7, -9) -> (8, 0)
$ setpriority 10 -1; setpriority 10 1; setpriority 10 0

setpriority: ERROR, failed to set new priority

setpriority: ERROR, failed to set new priority
PID: 10, PRIORITY: 0
$
Ready List Processes:

Priority 0: (7, -9) -> (10, -8) -> (12, -9) -> (13, -2) -> (9, -7) -> (8, 0) -> (4, -8) -> (11, -8)
$
```

Figure 9: `setpriority` fails for any value other than 0

Test Case: *With MAXPRIO set to 2*
Assertions:

1. Scheduler always selects the first process on the highest priority non-empty list
2. Promotion correctly moves processes to the next higher priority list.
3. Demotion correctly moves a process to the next lower priority list.

Status: PASS

```
|11:36:11|adupree@babbage:[xv6-pdx]> grep "DEFAULT_BUDGET\|MAXPRIO\|TICKS_TO_PROMOTE" pdx.h
#define MAXPRIO 2
#define DEFAULT_BUDGET 100000
#define TICKS_TO_PROMOTE 15000
|11:36:16|adupree@babbage:[xv6-pdx]> make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6
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 6 &
$
Created 5 children. Starting loopforever on parent

Ready List Processes:

Priority 2: (4, 99507) -> (6, 99520) -> (5, 99512) -> (8, 99570)

Priority 1:

Priority 0:
$ setpriority 4 0; setpriority 5 1
PID: 4, PRIORITY: 0
PID: 5, PRIORITY: 1
$
Ready List Processes:

Priority 2: (8, 95939) -> (6, 95880)

Priority 1: (5, 100000)

Priority 0: (4, 100000)
$
```

Figure 10: Creating 6 background processes and changing priority of a couple of processes

The commands in Figures 10 and 11 occur in the same session and have the `DEFAULT_BUDGET` set to a value much greater than `TICKS_TO_PROMOTE` effectively turning off demotions in the scheduler. With this setup we use `loopforever 6 &` to instantiate 6 background processes, that will stay in the `MAXPRIO` queue because the budget is so high. Then we set the priority of process `4` to 0 and process `5` to 1, which correctly moves them to the respective ready list. Then in Figure 11 we repteadly invoke CTRL+R until a promotion occurs. First, note that the budgets for process `5`, and `4` remain unchanged. This is because the Scheduler always selects the first process on the highest priority non-empty list. With the default budget set so high, the lower priority processes effectively get starved out until they're promoted back to `MAXPRIO`. On the last invocation of CTRL+R we can see the promotion correctly moves all processes to the next higher priority list.

Lastly, to prove demotion is executing properly, in Figure 12, we set the `DEFAULT_BUDGET` to 1000 and `TICKS_TO_PROMOTE` to a really large number, effectively turning off promotions. We can

```
$
Ready List Processes:

Priority 2: (8, 95449) -> (6, 95392)

Priority 1: (5, 100000)

Priority 0: (4, 100000)
$
Ready List Processes:

Priority 2: (7, 94271) -> (8, 94299)

Priority 1: (5, 100000)

Priority 0: (4, 100000)
$
Ready List Processes:

Priority 2: (8, 89732) -> (7, 89702)

Priority 1: (5, 100000)

Priority 0: (4, 100000)
$
Ready List Processes:

Priority 2: (6, 87618) -> (5, 98452) -> (8, 87672)

Priority 1: (4, 100000)

Priority 0:
$
```

Figure 11: Repeteadly using CTRL+R until a promotion occurs

```
|11:57:53|adupree@babbage:[xv6-pdx]> grep "DEFAULT_BUDGET\|MAXPRIO\|TICKS_TO_PROMOTE" pdx.h
#define MAXPRIO 2
#define DEFAULT_BUDGET 1000
#define TICKS_TO_PROMOTE 10000000
|11:57:56|adupree@babbage:[xv6-pdx]> make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6
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 6 &
$
Created 5 children. Starting loopforever on parent

Ready List Processes:

Priority 2: (8, 610) -> (9, 630) -> (4, 542) -> (6, 600)

Priority 1:

Priority 0:
$
Ready List Processes:

Priority 2: (7, 121) -> (5, 83) -> (8, 120) -> (9, 140)

Priority 1:

Priority 0:
$
Ready List Processes:

Priority 2:

Priority 1: (5, 760) -> (6, 760) -> (7, 770) -> (4, 760)

Priority 0:
$
```

Figure 12: CTRL+R until a demotion occurs

see in Figure 12 that after the budgets of the background processes expire, they get moved down to the next lower priority list and their budgets are reset.

Test Case: *With MAXPRIO set to 6*
Assertions:

1. Scheduler always selects the first process on the highest priority non-empty list
2. Promotion correctly moves processes to the next higher priority list.
3. Demotion correctly moves a process to the next lower priority list.

Status: **PASS**

```
|18:47:33|adupree@babbage:[xv6-pdx]> grep "DEFAULT_BUDGET\|MAXPRIO\|TICKS_TO_PROMOTE" pdx.h
#define MAXPRIO 6
#define DEFAULT_BUDGET 100000
#define TICKS_TO_PROMOTE 15000
|18:47:44|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=r
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 10 &
$
Created 9 children. Starting loopforever on parent

Ready List Processes:

Priority 6: (11, 99810) -> (12, 99820) -> (4, 99736) -> (13, 99805) -> (6, 99765) -> (7, 99760) -> (8, 99770) -> (9, 99790)

Priority 5:

Priority 4:

Priority 3:

Priority 2:

Priority 1:

Priority 0:
$ setpriority 4 0; setpriority 5 1; setpriority 6 2; setpriority 7 3; setpriority 8 4
PID: 4, PRIORITY: 0
PID: 5, PRIORITY: 1
PID: 6, PRIORITY: 2
PID: 7, PRIORITY: 3
PID: 8, PRIORITY: 4
$
```

Figure 13: Creating 10 background processes and modifying priorities

```
$
Ready List Processes:

Priority 6: (13, 95424) -> (9, 95409) -> (11, 95411)

Priority 5:

Priority 4: (8, 100000)

Priority 3: (7, 100000)

Priority 2: (6, 100000)

Priority 1: (5, 100000)

Priority 0: (4, 100000)
$
Ready List Processes:

Priority 6: (13, 93806) -> (9, 93788) -> (11, 93791)

Priority 5: (8, 100000)

Priority 4: (7, 100000)

Priority 3: (6, 100000)

Priority 2: (5, 100000)

Priority 1: (4, 100000)

Priority 0:
$
Ready List Processes:

Priority 6: (10, 91689) -> (13, 91697) -> (9, 91679)

Priority 5: (8, 100000)

Priority 4: (7, 100000)

Priority 3: (6, 100000)

Priority 2: (5, 100000)

Priority 1: (4, 100000)

Priority 0:
$
```

Figure 14: CTRL+R before and after periodic promotion

Similar to the previous test case, we first use `grep` to verify the values of `MAXPRIO`, `DEFAULT_BUDGET`, and `TICKS_TO_PROMOTE`; which are set to 6, 100000, and 15000 respectively. Again, because the budget is so high we are effectively turning off demotions in the scheduler. After instantiating 10 background processes we set the priority of processes `4` through `8` to priority 0 to 4 respectively.

Then, in Figure 14 we immeidiately inovoke CTRL+R to show that these processes were successfully added to the correct ready list. After, we wait some time for a promotion to occur and then hit CTRL+R again. As we can see in Figure 14 every process (except those in `MAXPRIO` queue) were promoted up one level. We can also deduce that the scheduler is still only selecting the first process on the highest priority non-empty list because the budgets for the lower priority processes remain unchanged after each iteration of CTRL+R.

```
|19:15:32|adupree@babbage:[xv6-pdx]> grep "DEFAULT_BUDGET\|MAXPRIO\|TICKS_TO_PROMOTE" pdx.h
#define MAXPRIO 6
#define DEFAULT_BUDGET 1000
#define TICKS_TO_PROMOTE 15000000
|19:15:34|adupree@babbage:[xv6-pdx]> make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.
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

Ready List Processes:

Priority 6: (8, 500) -> (4, 455) -> (5, 475)

Priority 5:

Priority 4:

Priority 3:

Priority 2:

Priority 1:

Priority 0:
$
Ready List Processes:

Priority 6:

Priority 5: (8, 691) -> (4, 680) -> (5, 680)

Priority 4:

Priority 3:

Priority 2:

Priority 1:

Priority 0:
$
```

Figure 15: Checking for correct demotion

Lastly, to demonstrate demotion is occuring properly we configure the `DEFAULT_BUDGET` to be 1000 and `TICKS_TO_PROMOTE` to be extremely large. Then after instantiating some background processes we wait for the budgets to expire and a demotion to occur, at which point we use CTRL+R to confirm that demotion correctly moves a process to the next lower priority list.