Compilation Test:

This section will show that my project 4 code compiles correctly with the CS444_P4 macro turned off, when CS333_PROJECT is set to “0” in the Makefile. (All other tests will be performed with the CS333_P4 macro turned on, when CS333_PROJECT is set to “4” in the Makefile.)

The expected outcome is that xv6 will compile correctly with CS333_PROJECT set to “0”. Since the output is so long, I will use two screenshots to display the results. They will both display the current date and time as well as the value for CS333_PROJECT. This information should be the same for both screenshots (with a slight variation in the times), demonstrating that they are from the same compilation sequence.

```
awurtz@babbage:~/CS333/xv6-pdx$ date
Wed 26 May 2021 05:18:47 PM 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 -MD -ggdb -m32 -Werror -fno-omit-frame-poir
essive-loop-optimizations -fno-stack-protector -DPDX_XV6 -fno-pie -no-pie -fno-pic -O -nostdinc -I. -c bootm
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -Og -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-poir
essive-loop-optimizations -fno-stack-protector -DPDX_XV6 -fno-pie -no-pie -fno-pic -nostdinc -I. -c bootasm.
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
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.

```
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.11931 s, 42.9 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00344385 s, 149 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
360+1 records in
360+1 records out
184572 bytes (185 kB, 180 KiB) copied, 0.00861232 s, 21.4 MB/s
awurtz@babbage:~/CS333/xv6-pdx$ grep "CS333_PROJECT ?=" Makefile
CS333_PROJECT ?= 0
awurtz@babbage:~/CS333/xv6-pdx$ date
Wed 26 May 2021 05:19:00 PM PDT
awurtz@babbage:~/CS333/xv6-pdx$
```

Figure 2: Compilation with CS333_PROJECT Set to 0.

As expected, figures 1 and 2 show that xv6 successfully compiled with CS333_PROJECT set to 0. The grep command shows that it is set to 0 in both screenshots, and the date command shows that the screenshots were taken within several seconds of each other.

This test **PASSES**.

MAXPRIO == 0

This set of tests will show that when MAXPRIO == 0, the scheduler operates as a single round robin queue, that setpriority() fails for all non-zero values, and that the code should not attempt promotion or demotion when MAXPRIO == 0.

Subtest 1: This subtest shows that when MAXPRIO == 0, the scheduler operates as a single round-robin queue.

The expected outcome of this test is that there will be a single priority queue containing RUNNABLE processes, and that this queue will operate as a FIFO queue. The screen shot will include multiple outputs of ctrl-r, which prints the priority queue. This should show the processes moving up through the queue in order and being added back onto the end when they are done running. Since xv6 has two CPUs running processes from a single queue, the processes may not be added back into the ready list in the same order as they were removed. This is expected behavior.

```
$ Ready List Processes:
Prio 0: (34, 0.300) -> (41, 0.300) -> (32, 0.300) -> (60, 0.300) -> (19, 0.300) -> (39, 0.300) -> (59, 0.300) -> (35, 0.300) -> (37, 0.300) -> (29
(20, 0.300) -> (44, 0.300) -> (55, 0.300) -> (57, 0.300) -> (51, 0.300) -> (47, 0.300) -> (7, 0.300) -> (46, 0.300) -> (45, 0.300) -> (49, 0.300)
(40, 0.300) -> (48, 0.300) -> (54, 0.300) -> (56, 0.300) -> (24, 0.300) -> (23, 0.300) -> (26, 0.300) -> (12, 0.300) -> (16, 0.300) -> (38, 0.300)
(63, 0.300) -> (3, 0.300) -> (25, 0.300) -> (18, 0.300) -> (15, 0.300) -> (17, 0.300) -> (21, 0.300) -> (11, 0.300) -> (5, 0.300) -> (22, 0.300) -
$ Ready List Processes:
Prio 0: (13, 0.300) -> (40, 0.300) -> (48, 0.300) -> (54, 0.300) -> (56, 0.300) -> (24, 0.300) -> (23, 0.300) -> (26, 0.300) -> (12, 0.300) -> (16
(8, 0.300) -> (63, 0.300) -> (3, 0.300) -> (25, 0.300) -> (18, 0.300) -> (15, 0.300) -> (17, 0.300) -> (21, 0.300) -> (11, 0.300) -> (5, 0.300) ->
(36, 0.300) -> (34, 0.300) -> (9, 0.300) -> (41, 0.300) -> (32, 0.300) -> (60, 0.300) -> (19, 0.300) -> (39, 0.300) -> (59, 0.300) -> (35, 0.300)
(62, 0.300) -> (42, 0.300) -> (20, 0.300) -> (44, 0.300) -> (55, 0.300) -> (57, 0.300) -> (51, 0.300) -> (47, 0.300) -> (7, 0.300) -> (46, 0.300)
$ Ready List Processes:
Prio 0: (28, 0.300) -> (36, 0.300) -> (34, 0.300) -> (9, 0.300) -> (41, 0.300) -> (32, 0.300) -> (60, 0.300) -> (19, 0.300) -> (39, 0.300) -> (59,
(50, 0.300) -> (62, 0.300) -> (42, 0.300) -> (20, 0.300) -> (44, 0.300) -> (55, 0.300) -> (57, 0.300) -> (51, 0.300) -> (47, 0.300) -> (7, 0.300)
(53, 0.300) -> (10, 0.300) -> (13, 0.300) -> (40, 0.300) -> (48, 0.300) -> (54, 0.300) -> (56, 0.300) -> (24, 0.300) -> (23, 0.300) -> (26, 0.300)
(14, 0.300) -> (30, 0.300) -> (8, 0.300) -> (63, 0.300) -> (3, 0.300) -> (25, 0.300) -> (18, 0.300) -> (15, 0.300) -> (17, 0.300) -> (21, 0.300) -
$ Ready List Processes:
Prio 0: (10, 0.300) -> (13, 0.300) -> (40, 0.300) -> (48, 0.300) -> (54, 0.300) -> (56, 0.300) -> (24, 0.300) -> (23, 0.300) -> (26, 0.300) -> (12
(30, 0.300) -> (8, 0.300) -> (63, 0.300) -> (3, 0.300) -> (25, 0.300) -> (18, 0.300) -> (15, 0.300) -> (17, 0.300) -> (21, 0.300) -> (11, 0.300) -
(28, 0.300) -> (36, 0.300) -> (34, 0.300) -> (9, 0.300) -> (41, 0.300) -> (32, 0.300) -> (60, 0.300) -> (19, 0.300) -> (39, 0.300) -> (35, 0.300)
(50, 0.300) -> (42, 0.300) -> (62, 0.300) -> (20, 0.300) -> (44, 0.300) -> (55, 0.300) -> (57, 0.300) -> (51, 0.300) -> (7, 0.300) -> (47, 0.300)
```

Figure 3: RUNNABLE Priority Queue While Running P4-test

Figure 3 shows that the processes move though the queue in a round robin fashion. The third line of the first ctrl-r print out starts with process 40, then 48, 54, 56, 24, and so on. You can clearly see that this order is maintained in the second ctrl-r printout where the seconds process from the front is 40, then 48, 54, 56, 24, and so on. Process 13 was cutout of the first printout because the lines were longer than my screen, but it is very clear from the other processes that order is maintained as processes move through the queue. This is further reinforced by the two subsequent ctrl-r printouts.

This subtest **PASSES.**

Subtest 2: This subtest shows that setpriority() fails for any value other than 0 when MAXPRIO == 0.

This subtest relies on a test program called testsetprio() which takes two command line arguments. The first is the pid of the process whose priority is to be updated, and they second is the new priority. The test displays MAXPRIO, followed a getpriority call displaying the processes original priority. It then prints the successfully updated priority or an error message describing the invalid input.

The expected behavior for this test is that, with MAXPRIO == 0, that setpriority() will fail/return an error for any priority other than zero. It is also expected that attempting to setpriority() to 0 will be successful, even though all processes already have a priority of 0.

```
$ testsetprio 2 0
setpriority() test:
MAXPRIO is 0
Initial Priority: getpriority(2) = 0
setpriority() was successful!
Updated Priority: getpriority(2) = 0
$ testsetprio 2 1
setpriority() test:
MAXPRIO is 0
Initial Priority: getpriority(2) = 0
Error: invalid priority
Usage: testsetprio [<pid> <prio>]
$ testsetprio 2 -1
setpriority() test:
MAXPRIO is 0
Initial Priority: getpriority(2) = 0
Error: invalid priority
Usage: testsetprio [<pid> <prio>]
$
PID     Name        UID       GID     PPID    Prio
1       init        0         0       1       0
2       sh          0         0       1       0
$
```

Figure 4: Attempts to setpriority() to 0, 1, and -1

As expected, Figure 4 shows that setpriority() fails for priority values other than 0. The first test shows that setpriority() is successful for 0, and the subsequent two tests show it fails for the values of 1 and -1. While setpriority() will always fail for negative numbers it only fails for 1 in this case because MAXPRIO == 0 and 1 < 0.

This subtest **PASSES.**

Subtest 3: This subtest shows that the code does not attempt promotion/demotion when MAXPRIO == 0.

For this subtest, MAXPRIO and TICKS_TO_PROMOTE are set to "0". The DEFAULT_BUDGET is set to 300. Then p4-test is run so that there is a long queue of RUNNABLE processes moving through the ready list. The MLFQ algorithm is designed to check for promotion conditions every time that the scheduler runs and check for the demotion conditions every time a process leaves the RUNNING state. Because TICKS_TO_PROMOTE is set to 0, if the code should check for promotion conditions every time the scheduler runs.

The expected results are that xv6 will run properly and successful move processes through the ready list because it will not attempt to promote or demote processes when MAXPRIO == 0. Since the code should not attempt to promote/demote any processes. Since maintaining the budget is part of demotion (which should not be attempted), the budget will not change. It will be clear that promotion does not occur since the process continue to cycle through a single priority queue in order (except for slight variations about how they are added to the end of the queue since both CPUs share a single ready list).

```
Ready List Processes:
Prio 0: (40, 0.300) -> (45, 0.300) -> (48, 0.300) -> (33, 0.300) -> (28, 0.300) -> (22, 0.300) -> (15, 0.300) -> (23, 0.300) -> (6, 0.300) -> (16,
(25, 0.300) -> (27, 0.300) -> (8, 0.300) -> (5, 0.300) -> (9, 0.300) -> (11, 0.300) -> (12, 0.300) -> (42, 0.300) -> (10, 0.300) -> (19, 0.300) ->
(31, 0.300) -> (32, 0.300) -> (36, 0.300) -> (49, 0.300) -> (34, 0.300) -> (37, 0.300) -> (38, 0.300) -> (39, 0.300) -> (41, 0.300) -> (59, 0.300)
(62, 0.300) -> (55, 0.300) -> (43, 0.300) -> (54, 0.300) -> (57, 0.300) -> (58, 0.300) -> (63, 0.300) -> (44, 0.300) -> (50, 0.300) -> (3, 0.300)
$ Ready List Processes:
Prio 0: (31, 0.300) -> (32, 0.300) -> (36, 0.300) -> (49, 0.300) -> (34, 0.300) -> (37, 0.300) -> (38, 0.300) -> (39, 0.300) -> (41, 0.300) -> (59
(62, 0.300) -> (55, 0.300) -> (43, 0.300) -> (54, 0.300) -> (57, 0.300) -> (58, 0.300) -> (63, 0.300) -> (44, 0.300) -> (50, 0.300) -> (3, 0.300)
(47, 0.300) -> (40, 0.300) -> (45, 0.300) -> (48, 0.300) -> (28, 0.300) -> (33, 0.300) -> (22, 0.300) -> (15, 0.300) -> (23, 0.300) -> (6, 0.300)
(26, 0.300) -> (25, 0.300) -> (27, 0.300) -> (8, 0.300) -> (5, 0.300) -> (9, 0.300) -> (11, 0.300) -> (42, 0.300) -> (12, 0.300) -> (10, 0.300) ->
$ Ready List Processes:
Prio 0: (46, 0.300) -> (47, 0.300) -> (40, 0.300) -> (45, 0.300) -> (48, 0.300) -> (28, 0.300) -> (33, 0.300) -> (22, 0.300) -> (15, 0.300) -> (23
(24, 0.300) -> (26, 0.300) -> (25, 0.300) -> (27, 0.300) -> (8, 0.300) -> (5, 0.300) -> (9, 0.300) -> (11, 0.300) -> (42, 0.300) -> (12, 0.300) ->
(29, 0.300) -> (30, 0.300) -> (31, 0.300) -> (32, 0.300) -> (49, 0.300) -> (36, 0.300) -> (34, 0.300) -> (37, 0.300) -> (38, 0.300) -> (39, 0.300)
(14, 0.300) -> (61, 0.300) -> (62, 0.300) -> (55, 0.300) -> (43, 0.300) -> (54, 0.300) -> (57, 0.300) -> (63, 0.300) -> (58, 0.300) -> (44, 0.300)
```

Figure 5: Ready List while DEFAULT_BUDGET == 300 and TICKS_TO_PROMOTE == 0

As expected, all processes continue cycling through the Priority 0 queue and the budgets do not change from their default value. The 3 Ready List printouts show that the processes are moving through the queue in order, budgets unchanged. This shows that demotion is not being attempted because the budgets are never entering a state where demotion might occur. It is also apparent that promotion is not being attempted because processes maintain the correct ordering in the queue until the enter the CPU. If the algorithm attempted to promote them to priority 0, it would remove them from the list and add them to the end. This is not happening. Instead, we see the expected round-robin behavior.

This subtest **PASSES**.

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

MAXPRIO = 2

This set of tests will show that when MAXPRIO == 2 the multilevel feedback queue scheduler works correctly including appropriate process selection, promotion, and demotion.

Subtest 1: This subtest shows that the scheduler always selects the first process on the highest priority non-empty list when MAXPRIO == 2.

In order to demonstrate the order in which processes are scheduled, I used a ran 10 processes in infinite loops in a user program called loopforever. In order to slow down the rate at which processes move through the ready list I set SCHED_INTERVAL to 100 (the default value is 10). This allowed processes to spend more time on the ready list before they were scheduled, making it possible to see how they move from the ready to RUNNING state.

The expected result of this test is that processes at the head of the highest non-empty list will always be selected to run next.

```
$ Ready List Processes:
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (5, 0.100) -> (6, 0.100) -> (12, 0.100) -> (10, 0.100) -> (7, 0.100) -> (14, 0.100) -> (8, 0.100) -> (9, 0.100) -
$
PID     Name         UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init         0       0       1       2       175.677 0.088   sleep   12288   801042bf 80104492 801060ad 80105
2       sh           0       0       1       2       175.580 0.049   sleep   16384   801042bf 80104492 801060ad 80105
3       loopforever  0       0       2       2       166.809 0.089   sleep   12288   801042bf 80104492 801060ad 80105
4       loopforever  0       0       3       0       161.769 46.677  runble  12288
5       loopforever  0       0       3       0       156.762 40.359  run     12288
6       loopforever  0       0       3       0       151.757 34.388  run     12288
7       loopforever  0       0       3       0       146.677 30.296  runble  12288
8       loopforever  0       0       3       0       141.577 27.698  runble  12288
9       loopforever  0       0       3       0       136.477 24.898  runble  12288
10      loopforever  0       0       3       0       131.377 22.801  runble  12288
11      loopforever  0       0       3       0       126.277 21.600  runble  12288
12      loopforever  0       0       3       0       121.177 20.100  runble  12288
13      loopforever  0       0       3       0       116.077 18.801  runble  12288
14      loopforever  0       0       3       0       116.072 21.701  runble  12288
$ Ready List Processes:
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (7, 0.100) -> (14, 0.100) -> (8, 0.100) -> (9, 0.100) -> (13, 0.100) -> (11, 0.100) -> (4, 0.100) -> (5, 0.100) -
$
PID     Name         UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init         0       0       1       2       175.917 0.088   sleep   12288   801042bf 80104492 801060ad 80105
2       sh           0       0       1       2       175.820 0.049   sleep   16384   801042bf 80104492 801060ad 80105
3       loopforever  0       0       2       2       167.049 0.089   sleep   12288   801042bf 80104492 801060ad 80105
4       loopforever  0       0       3       2       162.009 46.677  runble  12288
5       loopforever  0       0       3       2       157.002 40.459  runble  12288
6       loopforever  0       0       3       2       151.997 34.488  runble  12288
7       loopforever  0       0       3       2       146.917 30.396  runble  12288
8       loopforever  0       0       3       2       141.817 27.698  run     12288
9       loopforever  0       0       3       2       136.717 24.898  run     12288
10      loopforever  0       0       3       2       131.617 22.901  runble  12288
11      loopforever  0       0       3       2       126.517 21.600  runble  12288
12      loopforever  0       0       3       2       121.417 20.200  runble  12288
13      loopforever  0       0       3       2       116.317 18.801  runble  12288
14      loopforever  0       0       3       0       116.312 21.801  runble  12288
$ Ready List Processes:
Prio 2: (4, 0.100) -> (5, 0.100) -> (6, 0.100) -> (12, 0.100) -> (10, 0.100) -> (7, 0.100)
Prio 1: (8, 0.100) -> (9, 0.100)
Prio 0: (14, 0.100)
$
PID     Name         UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init         0       0       1       2       176.178 0.088   sleep   12288   801042bf 80104492 801060ad 80105
2       sh           0       0       1       2       176.081 0.049   sleep   16384   801042bf 80104492 801060ad 80105
3       loopforever  0       0       2       2       167.310 0.089   sleep   12288   801042bf 80104492 801060ad 80105
4       loopforever  0       0       3       2       162.270 46.677  run     12288
5       loopforever  0       0       3       2       157.263 40.459  run     12288
6       loopforever  0       0       3       2       152.258 34.488  runble  12288
7       loopforever  0       0       3       2       147.178 30.396  runble  12288
8       loopforever  0       0       3       1       142.078 27.798  runble  12288
9       loopforever  0       0       3       1       136.978 24.998  runble  12288
10      loopforever  0       0       3       2       131.878 22.901  runble  12288
11      loopforever  0       0       3       1       126.778 21.700  runble  12288
12      loopforever  0       0       3       2       121.678 20.200  runble  12288
13      loopforever  0       0       3       1       116.578 18.901  runble  12288
14      loopforever  0       0       3       0       116.573 21.801  runble  12288
```

Figure 6: Ctrl-r and Ctrl-p Output During "Loop Forever"

As expected, Figure 6 shows very clearly that the processes are being selected from the head of the highest non-empty list. The first ready list printout shows that processes 5 and 6 are at the head of list 0, which is the highest priority queue that is not null. The following ctrl-p print out shows that they are indeed scheduled to run next. This pattern continues even as different priority queues become populated. There is some discrepancy between the printouts, for instance processes 12 and 10 managed to enter and leave the cpu between my ctrl-p/ctrl-r, but their movement through the list suggests that the scheduler is still behaving as expected.

This subtest **PASSES**.

Subtest 2: This subtest shows that the promotion correctly moves processes on the ready lists to the next higher priority list (if one exists) and maintains correct ordering when MAXPRIO == 2.

The expected result for this test is that when processes are promoted, they will move up to the next highest priority queue, and that they will maintain their ordering from before the promotion.

```
$ Ready List Processes:
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (6, 0.100) -> (7, 0.100) -> (9, 0.100) -> (10, 0.100) -> (5, 0.100) -> (12, 0.100) -> (14, 0.100) -> (4, 0.100) -> (11, 0.100)
$ Ready List Processes:
Prio 2: (NULL)
Prio 1: (5, 0.100) -> (12, 0.100) -> (14, 0.100) -> (4, 0.100) -> (11, 0.100) -> (8, 0.100) -> (13, 0.100) -> (6, 0.100) -> (7, 0.100)
Prio 0: (NULL)
$ Ready List Processes:
Prio 2: (NULL)
Prio 1: (14, 0.100) -> (4, 0.100) -> (11, 0.100) -> (8, 0.100) -> (13, 0.100) -> (6, 0.100) -> (7, 0.100)
Prio 0: (9, 0.100) -> (10, 0.100)
```

Figure 7: Process Promotion from Prio 0 to Prio 1.

As expected, Figure 7 shows the priority 0 processes being promoted to priority 1, maintaining their original order. I was rapidly pushing ctrl-r, but between prints, several processes were removed from the list. However, the processes maintained their order from before to after their promotion. Process 5 is in the middle of the priority 0 ready list, and by the time we see then next print out, it is at the head of the priority 1 ready list. The processes behind processes 5 are in the same order in both lists. This shows that promotion maintains the order or ready lists.

This subtest **PASSES**.

Subtest 3: This subtest shows that demotion correctly moves a process to the next lower priority list (if one exists) when the processes budget is used up when MAXPRIO == 2.

The expected outcome is that this test will show processes being successfully demoted to the next lower priority queue when their budget reaches 0. This will be demonstrated through their progression though the Ready lists, which also show their budget.

```
$ Ready List Processes:
Prio 2: (63, 0.091) -> (14, 0.092) -> (21, 0.087) -> (35, 0.089) -> (28, 0.088) -> (7, 0.090) -> (49, 0.087) -> (56, 0.086) -> (42, 0.087) -> (3, 0.091)
Prio 1: (45, 0.001) -> (61, 0.008) -> (4, 0.008) -> (19, 0.001) -> (33, 0.001)
Prio 0: (13, 0.100) -> (37, 0.100) -> (58, 0.100) -> (6, 0.100) -> (25, 0.100) -> (55, 0.100) -> (59, 0.100) -> (36, 0.100) -> (48, 0.100) -> (24, 0.100)
(15, 0.100) -> (23, 0.100) -> (38, 0.100) -> (54, 0.100) -> (51, 0.100) -> (17, 0.100) -> (62, 0.100) -> (44, 0.100) -> (9, 0.100) -> (39, 0.100) -> (11,
(47, 0.100) -> (34, 0.100) -> (27, 0.100) -> (5, 0.100) -> (31, 0.100) -> (12, 0.100) -> (52, 0.100) -> (60, 0.100) -> (50, 0.100) -> (20, 0.100) -> (46,
$ Ready List Processes:
Prio 2: (63, 0.091) -> (14, 0.092) -> (21, 0.087) -> (35, 0.089) -> (28, 0.088) -> (7, 0.090) -> (49, 0.087) -> (56, 0.086) -> (42, 0.087) -> (3, 0.091)
Prio 1: (NULL)
Prio 0: (15, 0.100) -> (23, 0.100) -> (38, 0.100) -> (54, 0.100) -> (51, 0.100) -> (17, 0.100) -> (62, 0.100) -> (44, 0.100) -> (9, 0.100) -> (39, 0.100)
(47, 0.100) -> (34, 0.100) -> (27, 0.100) -> (5, 0.100) -> (31, 0.100) -> (12, 0.100) -> (52, 0.100) -> (60, 0.100) -> (50, 0.100) -> (20, 0.100) -> (46,
(57, 0.100) -> (45, 0.100) -> (61, 0.100) -> (19, 0.100) -> (4, 0.100) -> (33, 0.100) -> (13, 0.090) -> (58, 0.090) -> (37, 0.090) -> (6, 0.090) -> (25,
(53, 0.090) -> (24, 0.090) -> (8, 0.090) -> (16, 0.090)
```

Figure 8: Ready List Processes During P4-Test

As expected, Figure 8 shows that processes are demoted to the next-lower priority queue when their budget runs out. Figure 8 shows that all the processes in the priority 1 queue are reaching the end of their budget, and they have all been demoted with their budgets reset by the next printout. Prio 1 is NULL, and you can see that processes 45, 61, 19, 4, and 33 have moved to priority list 0. It makes sense that they maintained their order (with one pair-wise switch) because they ran in order and were demoted after they exited the cpu.

This subtest **PASSES.**

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

MAXPRIO = 6

This set of tests will show that when MAXPRIO == 6 the MLFQ scheduler works correctly, including correct process selection, promotion, and demotion.

Subtest 1: This subtest shows that the scheduler always selects the first process on the highest priority non-empty list when MAXPRIO == 6.

In order to demonstrate the order in which processes are scheduled, I used a ran 10 processes in infinite loops in a user program called loopforever. In order to slow down the rate at which processes move through the ready list I set SCHED_INTERVAL to 100 (the default value is 10). This allowed processes to spend more time on the ready list before they were scheduled, making it possible to see how they move from the ready to RUNNING state.

The expected result of this test is that processes at the head of the highest non-empty list will always be selected to run next.

```
$ Ready List Processes:
Prio 6: (NULL)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (14, 0.100) -> (5, 0.100) -> (13, 0.100) -> (8, 0.100) -> (12, 0.100) -> (6, 0.100) -> (7, 0.100) -> (11, 0.100) -> (9, 0.100)
$
PID     Name        UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init        0       0       1       6       392.827 0.081   sleep   12288   801042be 80104491 801060ac 801054a7 801064f2
2       sh          0       0       1       6       392.739 0.040   sleep   16384   801042be 80104491 801060ac 801054a7 801064f2
3       loopforever 0       0       2       6       388.080 0.088   sleep   12288   801042be 80104491 801060ac 801054a7 801064f2
4       loopforever 0       0       3       0       383.042 86.096  runble  12288
5       loopforever 0       0       3       0       378.035 79.700  run     12288
6       loopforever 0       0       3       0       373.030 75.388  runble  12288
7       loopforever 0       0       3       0       367.927 70.294  runble  12288
8       loopforever 0       0       3       0       362.827 68.191  runble  12288
9       loopforever 0       0       3       0       357.727 65.796  runble  12288
10      loopforever 0       0       3       0       352.627 63.201  runble  12288
11      loopforever 0       0       3       0       347.527 61.800  runble  12288
12      loopforever 0       0       3       0       342.427 60.600  runble  12288
13      loopforever 0       0       3       0       337.327 59.300  runble  12288
14      loopforever 0       0       3       0       337.322 61.600  run     12288
$ Ready List Processes:
Prio 6: (NULL)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (9, 0.100) -> (10, 0.100) -> (4, 0.100) -> (14, 0.100) -> (5, 0.100) -> (13, 0.100) -> (8, 0.100) -> (12, 0.100)
Prio 0: (6, 0.100)
$
PID     Name        UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
1       init        0       0       1       6       393.281 0.081   sleep   12288   801042be 80104491 801060ac 801054a7 801064f2
2       sh          0       0       1       6       393.193 0.040   sleep   16384   801042be 80104491 801060ac 801054a7 801064f2
3       loopforever 0       0       2       6       388.534 0.088   sleep   12288   801042be 80104491 801060ac 801054a7 801064f2
4       loopforever 0       0       3       1       383.496 86.096  runble  12288
5       loopforever 0       0       3       1       378.489 79.800  runble  12288
6       loopforever 0       0       3       0       373.484 75.488  runble  12288
7       loopforever 0       0       3       0       368.381 70.394  runble  12288
8       loopforever 0       0       3       1       363.281 68.291  runble  12288
9       loopforever 0       0       3       1       358.181 65.796  run     12288
10      loopforever 0       0       3       1       353.081 63.201  run     12288
11      loopforever 0       0       3       0       347.981 61.900  runble  12288
12      loopforever 0       0       3       1       342.881 60.700  runble  12288
13      loopforever 0       0       3       1       337.781 59.400  runble  12288
14      loopforever 0       0       3       1       337.776 61.700  runble  12288
```

Figure 9: Process Selection During Loopforever

As expected, processes are selected and run from the highest priority non-empty queue to be run. It is very clear from figure 7 that the processes at the head of the ready lists were run immediately afterward. You can see that processes were promote from priority 0 to priority 1 and that the scheduler continued to select processes from the highest priority non-empty queue, even after process promotion occurred.

This subtest **PASSES.**

Subtest 2: This subtest shows that the promotion correctly moves processes on the ready lists to the next higher priority list (if one exists) and maintains correct ordering when MAXPRIO == 6.

The expected outcome of this test is that processes will be promoted to the next highest ready list, maintaining their ordering from the original list. Some shifting may occur because processes are actively being run and moved through the CPUs, but the overall order should remain the same.

```
$ Ready List Processes:
Prio 6: (7, 0.098) -> (42, 0.099) -> (35, 0.100) -> (63, 0.098) -> (14, 0.099) -> (56, 0.098) -> (3, 0.098) -> (21, 0.099) -> (49, 0.095) -> (28, 0.100)
Prio 5: (NULL)
Prio 4: (38, 0.091) -> (18, 0.090) -> (23, 0.090) -> (47, 0.090) -> (6, 0.090) -> (51, 0.090) -> (46, 0.090) -> (60, 0.090) -> (50, 0.090) -> (4, 0.090) -> (45, 0.090) -> (39, 0.090) -> (8, 0.090) -> (37,
(57, 0.090) -> (52, 0.090) -> (61, 0.090) -> (13, 0.090) -> (54, 0.090) -> (58, 0.090) -> (22, 0.090) -> (17, 0.090) -> (11, 0.090) -> (53, 0.090) -> (48, 0.090) -> (12, 0.090) -> (41, 0.090) -> (59, 0.09
(10, 0.090) -> (15, 0.090) -> (43, 0.090) -> (31, 0.090) -> (33, 0.090) -> (44, 0.090) -> (27, 0.090) -> (24, 0.090) -> (36, 0.090) -> (30, 0.090) -> (20, 0.090) -> (25, 0.090) -> (40, 0.091) -> (5, 0.092
(16, 0.091) -> (29, 0.090) -> (62, 0.090) -> (19, 0.090)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (NULL)
$ Ready List Processes:
Prio 6: (28, 0.100) -> (7, 0.100) -> (42, 0.100) -> (35, 0.100) -> (63, 0.100) -> (14, 0.100) -> (56, 0.100) -> (3, 0.100) -> (21, 0.100) -> (49, 0.100)
Prio 5: (51, 0.070) -> (60, 0.070) -> (46, 0.070) -> (50, 0.070) -> (4, 0.070) -> (45, 0.070) -> (39, 0.070) -> (8, 0.070) -> (37, 0.070) -> (57, 0.070) -> (26, 0.070) -> (61, 0.070) -> (52, 0.070) -> (13
(17, 0.070) -> (11, 0.070) -> (53, 0.070) -> (54, 0.070) -> (48, 0.070) -> (22, 0.070) -> (12, 0.070) -> (41, 0.060) -> (55, 0.060) -> (59, 0.060) -> (15, 0.060) -> (10, 0.060) -> (43, 0.060) -> (31, 0.06
(44, 0.060) -> (24, 0.060) -> (27, 0.060) -> (20, 0.060) -> (36, 0.060) -> (30, 0.060) -> (40, 0.060) -> (25, 0.060) -> (5, 0.060) -> (16, 0.060) -> (32, 0.060) -> (62, 0.060) -> (29, 0.060) -> (19, 0.060
(38, 0.060) -> (34, 0.060) -> (23, 0.060) -> (18, 0.060)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (NULL)
```

Figure 10: Process Promotion from Prio 4 to Prio 5

As expected, Figure 10 shows that processes are promoted to the next highest priority queue, maintaining their order when MAXPRIO is set to 6. The highlighted processes clearly demonstrate that the order was maintained after promotion. The slight shifting in the ready list is expected, since some processes entered the CPU between control sequences.

This subtest **PASSES.**

Subtest 3: This subtest shows that demotion correctly moves a process to the next lower priority list (if one exists) when the processes budget is used up when MAXPRIO == 6.

The expected outcome of this test is that processes will be demoted when their budget runs about and that they will be moved to the next lower priority list. This will be demonstrated using a series of ctrl-r prints and the p4-test program.

```
$ Ready List Processes:
Prio 6: (63, 0.088) -> (42, 0.092) -> (49, 0.092) -> (21, 0.093) -> (28, 0.094) -> (7, 0.094) -> (56, 0.094) -> (35, 0.095) -> (14, 0.092) -> (3, 0.090)
Prio 5: (10, 0.030) -> (46, 0.030) -> (37, 0.031) -> (34, 0.030) -> (40, 0.023) -> (26, 0.020) -> (23, 0.020) -> (33, 0.024) -> (20, 0.020) -> (8, 0.019) -> (25, 0.020) -> (43, 0.021) -> (57, 0.020) -> (45, 0.021) -> (53, 0.020)
(24, 0.020) -> (18, 0.021) -> (41, 0.023) -> (6, 0.021) -> (44, 0.021) -> (55, 0.022) -> (52, 0.021) -> (12, 0.020) -> (30, 0.020) -> (19, 0.020) -> (50, 0.020) -> (16, 0.020) -> (27, 0.010) -> (29, 0.020) -> (11, 0.027) ->
(17, 0.022) -> (60, 0.026) -> (36, 0.029) -> (61, 0.022) -> (15, 0.029) -> (31, 0.020) -> (51, 0.020) -> (32, 0.020) -> (4, 0.029) -> (39, 0.020) -> (54, 0.021) -> (47, 0.020) -> (58, 0.020) -> (9, 0.021) -> (22, 0.020) ->
(13, 0.020) -> (59, 0.021) -> (5, 0.024) -> (38, 0.024)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (NULL)
$ Ready List Processes:
Prio 6: (3, 0.090)
Prio 5: (NULL)
Prio 4: (8, 0.100) -> (25, 0.100) -> (53, 0.100) -> (30, 0.100) -> (29, 0.100) -> (50, 0.100) -> (39, 0.100) -> (47, 0.100) -> (58, 0.100) -> (22, 0.100) -> (13, 0.100) -> (62, 0.100) -> (48, 0.100) -> (46, 0.100) -> (10, 0.100)
(34, 0.100) -> (33, 0.100) -> (23, 0.100) -> (45, 0.100) -> (57, 0.100) -> (18, 0.100) -> (24, 0.100) -> (41, 0.100) -> (6, 0.100) -> (44, 0.100) -> (52, 0.100) -> (55, 0.100) -> (12, 0.100) -> (16, 0.100) -> (36, 0.100) ->
(17, 0.100) -> (60, 0.100) -> (61, 0.100) -> (15, 0.100) -> (31, 0.100) -> (4, 0.100) -> (32, 0.100) -> (9, 0.100) -> (54, 0.100) -> (59, 0.100) -> (38, 0.100) -> (37, 0.100) -> (5, 0.100) -> (19, 0.100) -> (43, 0.100) ->
(51, 0.100) -> (40, 0.100) -> (11, 0.100) -> (27, 0.090) -> (26, 0.090) -> (20, 0.090)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (NULL)
```

Figure 11: Process Demotion from Prio 5 to Prio 4

As expected, Figure 11 shows that processes are correctly demoted to the next lower priority queue when their budget runs out. In Figure 11, you can see that several processes were demoted between the two Ready List print outs. Their budgets were dwindling when they were on the priority 5 live, but after they were demoted to priority 4 their budgets were reset.

This subtest **PASSES.**

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

Setpriority()

This test will show that the setpriority() system call and helper function work correctly, including properly updating priority when given valid input, and returning an error when given an invalid PID or priority.

Subtest 1: This subtest shows that setpriority() changes the priority and that the budget is reset when given a valid PID and priority.

The expected outcome of this test is that setpriority() will successfully change the priority and reset the process's budget when it is given a valid PID and priority as arguments. This test uses a test program called testsetprio. It takes the command line arguments <pid> <prio> and calls setpriority(pid, prio) using those arguments, as well as printing out some helpful information.

```
$ Ready List Processes:
Prio 6: (12, 0.999) -> (13, 1.000)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (5, 0.975)
Prio 1: (NULL)
Prio 0: (6, 0.800) -> (10, 0.400) -> (7, 0.198)
$testsetprio 5 5
setpriority() test:
MAXPRIO is 6
Initial Priority: getpriority(5) = 2
setpriority() was successful!
Updated Priority: getpriority(5) = 5
$ Ready List Processes:
Prio 6: (15, 1.000) -> (17, 1.000) -> (14, 0.999)
Prio 5: (5, 0.990)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (10, 0.900) -> (9, 1.000) -> (11, 0.400) -> (6, 0.200) -> (12, 0.660) -> (7, 0.711)
```

Figure 12: Assigning New Priority to Process 5 Using Setpriority()

As expected, Figure 12 shows process 5's priority being changed from 2 to 5. The read lists show that process 5 moved priority queues appropriately and that its budget was reset. The DEFAULT_BUDGET for this test was set to 1000, so it appears that process 5 must have made it once through the CPU before the second ctrl-r went though, however it is still clear that its budget was reset because it increased after setpriority() was called.

This subtest **PASSES.**

Subtest 2: This subtest shows that changing the priority of a process on a ready list correctly moves the process to the list corresponding to the new priority.

The expected outcome of this test is that once a new priority is successful set for a process, that process will move to the appropriate ready list and its budget will be resent. This test uses the same test program as the previous subtest to call setpriority() using command line arguments.

```
Ready List Processes:
Prio 6: (14, 0.986) -> (42, 0.989) -> (56, 0.989) -> (4, 0.439) -> (49, 0.989) -> (35, 0.995) -> (63, 0.992) -> (7, 0.991) -> (21, 0.993) ->
Prio 5: (59, 0.500) -> (58, 0.500) -> (64, 0.500) -> (8, 0.409) -> (62, 0.500) -> (61, 0.501) -> (6, 0.400) -> (9, 0.401) -> (12, 0.408) ->
(19, 0.405) -> (18, 0.408) -> (24, 0.406) -> (26, 0.400) -> (20, 0.413) -> (23, 0.451) -> (22, 0.400) -> (25, 0.400) -> (30, 0.400) -> (32,
(37, 0.400) -> (38, 0.400) -> (43, 0.400) -> (36, 0.400) -> (39, 0.402) -> (41, 0.407) -> (45, 0.404) -> (44, 0.406) -> (40, 0.311) -> (47,
(53, 0.401) -> (55, 0.401) -> (57, 0.400)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (5, 1.000)
$ testsetprio 12 3
Now verify that your system is working by pressing C-p and then C-r.
setpriority() test:
MAXPRIO is 6
Initial Priority: getpriority(12) = 5
setpriority() was successful!
Updated Priority: getpriority(12) = 3
$ Ready List Processes:
Prio 6: (56, 0.987) -> (35, 0.994) -> (7, 0.991) -> (21, 0.991) -> (28, 0.988) -> (63, 0.990) -> (14, 0.985) -> (4, 0.426) -> (42, 0.988) ->
Prio 5: (27, 0.200) -> (34, 0.200) -> (33, 0.201) -> (37, 0.200) -> (38, 0.200) -> (43, 0.200) -> (36, 0.200) -> (41, 0.207) -> (39, 0.202)
(51, 0.200) -> (48, 0.200) -> (53, 0.201) -> (52, 0.201) -> (55, 0.201) -> (57, 0.200) -> (54, 0.200) -> (60, 0.200) -> (58, 0.201) -> (59,
(9, 0.102) -> (10, 0.109) -> (15, 0.162) -> (17, 0.173) -> (11, 0.060) -> (16, 0.116) -> (13, 0.110) -> (18, 0.108) -> (19, 0.105) -> (20, 0
(32, 0.100) -> (25, 0.100)
Prio 4: (NULL)
Prio 3: (12, 1.000)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (5, 1.000)
```

Figure 13: Changing Priority of Process 12 from Prio 5 to Prio 3

As expected, Figure 13 shows that when setpriority() is used to change a processes priority when it is in the RUNNABLE state, it is moved to the appropriate ready list corresponding to its new priority. I used testsetpriority to change process 12's priority from 5 to 3. You can see that it moved from ready[5] to ready[3] and that its budget was reset.

This subtest **PASSES.**

Subtest 3: This subtest shows that 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.

The expected outcome for this test is that the process whose priority is set to its current priority will maintain its position in the appropriate ready list and its budget will be reset. Since the scheduler is actively running processes while this test is taking place, it is expected that the priority queue may shift a bit, but the update process will maintain the same ordering in the queue regardless of shifting.

```
$ Ready List Processes:
Prio 6: (14, 1.499) -> (16, 1.500) -> (4, 1.476)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (6, 1.100) -> (7, 1.005) -> (5, 0.700) -> (9, 0.700) -> (8, 1.400)
$ testsetprio 5 0
setpriority() test:
MAXPRIO is 6
Initial Priority: getpriority(5) = 0
setpriority() was successful!
Updated Priority: getpriority(5) = 0
$ Ready List Processes:
Prio 6: (4, 1.470) -> (14, 1.499) -> (16, 1.500) -> (17, 1.500)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (5, 1.500) -> (9, 0.200) -> (8, 0.950) -> (10, 1.000) -> (6, 0.500)
```

Figure 14: Setting Process Priority to Its Current Priority

As expected, Figure 14 shows that process 5 maintains its position in ready list 0 when setpriority() is used to set its priority to its current value (0) and it's budget has been reset to its default (which for this test was 1500). In the first ready list print out, process 5 is third on the list and in the second printout it is first. Despite this small shift, it is still followed by processes 9 and 8, plus you can see that process 6 (which was initially first in the queue) has been added back to the end. This all shows that process 5 did not change positions in the queue even though setpriority ran (and reset its budget, as expected).

This subtest **PASSES.**

Subtest 4: This subtest shows that calling setpriority() with an invalid PID and/or priority returns a relevant error code and leaves the process priority and budget unmodified.

The expected outcome of this test is that setpriority() will return an error if either the PID or priority is invalid.

```
PID     Name        UID        GID     PPID    Prio    Elapsed CPU
1       init        0          0       1       6       3.490   0.080
2       sh          0          0       1       6       3.401   0.039
$ testsetprio 2 7
setpriority() test:
MAXPRIO is 6
Initial Priority: getpriority(2) = 6
Error: invalid priority
Usage: testsetprio [<pid> <prio>]
$ testsetprio 0 2
setpriority() test:
MAXPRIO is 6
Initial Priority: getpriority(0) = -1
Error: invalid pid
Usage: testsetprio [<pid> <prio>]
getpriority(0) = -1
```

Figure 15: Testing Setpriority() with Invalid PID and Priority

As expected, Figure 15 shows that setpriority() successfully returns an error in response to both an invalid PID and invalid priority. The first test uses a valid PID (2) and an invalid priority(7). This test results in an error saying the priority is invalid. The second test uses a valid priority (2) but an invalid PID(0). This test prints an error message saying that the PID is invalid.

This subtest **PASSES.**

Since all 4 subtests pass, this test **PASSES.**

Getpriority()

This set of tests will show that the getpriority() system call and helper function works correctly, meaning it shows the correct priority active processes and that it returns and error if the PID is not found or the process is in the UNUSED state.

Subtest 1: This subtest shows that getpriority() returns the correct priority for the current process.

The expected outcome of this test is that getpriority() will return the correct priority for the current process when it is given a valid PID. For this test, I added couple lines of code to the test program loopforever that print the PID of the current process and then prints the return value of getpriority() for the current process. It is expected that these values will be correct.

```
$ loopforever 5&
$ Current Process: 4
getpriority(4) = 6

PID     Name         UID      GID     PPID    Prio
1       init         0        0       1       6
2       sh           0        0       1       6
4       loopforever  0        0       1       6
```

Figure 16: Running Getpriority() on the Current Process

As expected, this test shows that getpriority() correctly returns the priority of the current process. The current process has PID == 4 and its priority is 6. This is confirmed by the ctrl-p printout following the getpriority() test.

This subtest **PASSES.**

Subtest 2: This subtest shows that getpriority() returns the correct priority for any process other than the current process.

The expected outcome of this test is that getpriority() will return the correct priority for any process other than the current process. The test uses the testsetprio program since getpriority() is used to display the initial and updated priorities process specified by setpriority().

```
$ testsetprio 2 4
setpriority() test:
MAXPRIO is 6
The current process is: 12
getpriority(12) = 6
Initial Priority: getpriority(2) = 6
setpriority() was successful!
Updated Priority: getpriority(2) = 4
$
PID     Name        UID     GID     PPID    Prio
1       init        0       0       1       6
2       sh          0       0       1       4
4       loopforever 0       0       1       6
5       loopforever 0       0       4       0
6       loopforever 0       0       4       0
8       loopforever 0       0       4       0
9       loopforever 0       0       4       0
10      loopforever 0       0       4       0
11      loopforever 0       0       4       0
```

Figure 17: Getpriority() Values for Processes Other than the Current Process

As expected, Figure 17 shows that getpriority() displayed the correct priority for a process other than the current process. In Figure 17, you can see that the current process has the PID 12 while getpriority() correctly returns the priority of process 2 (sh) before and after its priority is updated by setpriority().

This subtest **PASSES.**

Subtest 3: This subtest shows that getpriority() returns an error (-1) if PID is not found or process is in the UNUSED state.

The expected outcome of this test is that getpriority() will return an error if the PID is not found or if the process is in the UNUSED state.

```
PID     Name        UID     GID     PPID    Prio
1       init        0       0       1       6
2       sh          0       0       1       6
$ testsetprio 4 1
setpriority() test:
MAXPRIO is 6
The current process is: 3
getpriority(3) = 6
Initial Priority: getpriority(4) = -1
Error: invalid pid
Usage: testsetprio [<pid> <prio>]
getpriority(4) = -1
```

Figure 18: Getpriority() Error from Invalid PID

As expected, Figure 18 shows that getpriority() correctly returns an error when it is given an invalid PID. Figure 18 also shows getpriority() successfully returning a value for a valid PID(3).

This subtest **PASSES.**

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

Ps Command

This test will show that the ps command correctly displays the process priority.

The expected outcome of this test is that the ps command will correctly display the priority of the process along with other information.

```
Now verify that your system is working by pressing C-p and then C-r.
ps

PID     Name        UID         GID     PPID    Prio    Elapsed CPU     State   Size
1       init        0           0       1       6       19.631  0.101   sleep   12288
2       sh          0           0       1       6       19.525  0.059   sleep   16384
65      p4-test     0           0       4       5       2.420   0.270   sleep   12288
4       p4-test     0           0       1       6       14.360  0.448   sleep   12288
5       p4-test     0           0       4       5       14.335  0.889   sleep   12288
6       p4-test     0           0       4       5       14.329  0.890   sleep   12288
7       p4-test     0           0       4       6       14.324  0.001   sleep   12288
8       p4-test     0           0       4       5       14.309  0.890   sleep   12288
9       p4-test     0           0       4       5       14.304  0.990   sleep   12288
10      p4-test     0           0       4       5       13.281  0.750   sleep   12288
11      p4-test     0           0       4       5       13.276  0.747   sleep   12288
12      p4-test     0           0       4       5       13.271  0.740   sleep   12288
13      p4-test     0           0       4       5       13.266  0.740   sleep   12288
14      p4-test     0           0       4       6       13.251  0.001   sleep   12288
15      p4-test     0           0       4       5       12.211  0.690   sleep   12288
16      p4-test     0           0       4       5       12.206  0.690   sleep   12288
17      p4-test     0           0       4       5       12.201  0.688   sleep   12288
18      p4-test     0           0       4       5       12.196  0.680   sleep   12288
19      p4-test     0           0       4       5       12.181  0.690   sleep   12288
20      p4-test     0           0       4       5       11.141  0.729   sleep   12288
21      p4-test     0           0       4       6       11.136  0.001   sleep   12288
22      p4-test     0           0       4       5       11.131  0.726   sleep   12288
23      p4-test     0           0       4       5       11.126  0.729   sleep   12288
24      p4-test     0           0       4       5       11.121  0.730   sleep   12288
25      p4-test     0           0       4       5       10.091  0.560   sleep   12288
26      p4-test     0           0       4       5       10.086  0.558   sleep   12288
27      p4-test     0           0       4       5       10.081  0.560   sleep   12288
28      p4-test     0           0       4       6       10.076  0.000   sleep   12288
29      p4-test     0           0       4       5       10.061  0.554   sleep   12288
```

Figure 19: Ps Command Output During P4-Test

As expected, Figure 19 shows the output of the ps command during p4-test. The priories are correctly displayed, and the output is well formatted.

This test **PASSES.**

Ctrl-p Command

This test will show that control-p correctly displayed the process priority.

The expected outcome of this test is that the ctrl-p sequence will correctly display the process priority along with other information about the process.

```
PID     Name         UID        GID     PPID    Prio    Elapsed CPU     State   Size     PCs
1       init         0          0       1       6       47.065  0.099   sleep   12288    801042e9
2       sh           0          0       1       6       46.966  0.052   sleep   16384    801042e9
4       loopforever  0          0       1       6       34.648  0.083   runble  12288
5       loopforever  0          0       4       0       29.592  13.689  runble  12288
6       loopforever  0          0       4       0       24.585  12.405  runble  12288
7       loopforever  0          0       4       0       19.580  10.084  runble  12288
8       loopforever  0          0       4       2       14.565  6.401   run     12288
9       loopforever  0          0       4       6       9.465   0.300   run     12288
10      loopforever  0          0       4       6       4.365   0.002   runble  12288
$ Ready List Processes:
Prio 6: (4, 1.480) -> (10, 1.498) -> (12, 1.500)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (6, 1.500) -> (5, 0.025) -> (7, 0.510)
```

Figure 20: Ctrl-P Output During Loopforever Program

As expected, Figure 20 shows the proper control-p output, including the process priorities. It is correctly and well formatted.

This test **PASSES.**

Ctrl-r Command

This test will show that control-r correctly displays all ready lists, from highest to lowest priority, and the budget for each process.

The expected outcome of this test is that ctrl-r will correctly print all of the ready lists from highest to lowest priority, and the budget for each process.

```
$ Ready List Processes:
Prio 6: (10, 0.100) -> (4, 0.100) -> (9, 0.100)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (8, 0.060) -> (5, 0.050)
$ Ready List Processes:
Prio 6: (10, 0.100) -> (4, 0.100)
Prio 5: (NULL)
Prio 4: (NULL)
Prio 3: (NULL)
Prio 2: (NULL)
Prio 1: (NULL)
Prio 0: (9, 0.100) -> (7, 0.050) -> (8, 0.060)
$
PID     Name         UID      GID     PPID    Prio    Elapsed CPU     State   Size
1       init         0        0       1       6       53.768  0.095   sleep   12288
2       sh           0        0       1       6       53.665  0.048   sleep   16384
4       loopforever  0        0       1       6       35.455  0.086   runble  12288
5       loopforever  0        0       4       0       30.372  19.297  run     12288
6       loopforever  0        0       4       0       25.365  13.204  runble  12288
7       loopforever  0        0       4       0       20.360  7.590   runble  12288
8       loopforever  0        0       4       0       15.348  3.721   runble  12288
9       loopforever  0        0       4       0       10.338  0.880   run     12288
10      loopforever  0        0       4       6       5.328   0.000   runble  12288
11      loopforever  0        0       4       6       0.318   0.000   runble  12288
```

Figure 21: Output of Ctrl-R During Loopforever Program

As expected, Figure 21 shows that ctrl-r does correctly print out all ready lists from highest to lowest priority, and it correctly display the budget for each process. The screen shot also includes a ctrl-p print out to verify correctness of the output.

This test **PASSES.**