Description

For this project, I learned about the implementation of a Multi-Level Feedback Queue (MLFQ) scheduler within xv6. In particular, the implementation of new ready state queues with corresponding priorties and to better match the MLFQ algorithm, a time *budget* based demotion, and *periodic* priority inflation to prevent starvation. This project also modified the previous `ps` and control commands to display the new information for debugging purposes.

Deliverables

The following features were added to xv6:

- New ready state lists which are indexed by their priority. The priority ranges from the highest priority 0 down to a parameter MAX.
- A new system call, `setpriority()`, which changes the priority of a process.
- A method for promoting processes as a means of starvation prevention within the MLFQ algorithm. Process promotion was based on the model of periodic priority adjustment as opposed to a full priority reset.
- Process demotion by lowering a process' priority after it uses all of alloted budget of time within the CPU.
- The `ps`, control–p, control–r commands were updated to display the new priority information.
- The new MLFQ algorithm was implemented since it an MLFQ scheduler minimizes response time (by maintaining Round Robin scheduling at each priority level; however, it also minimizes turnaround time without knowledge of job length. The later is the primary benefit of this new implementation over the previous round robin scheduler which exhibits poor turnaroud time.

Implementation

Priority Queues

The following files were modified to add the new priority queues

- `param.h`. A new macro, `MAX`, was added (Line 20). The macro is the maximum priority value.
- `proc.h`. The `proc struct` was modified to include a new field `int prio` which stores the per-process priority (Line 79).
- `proc.c`. The file was modified in the following ways:
  - The ready state list in the `ptable` struct was modified to be an array of ready state lists of size `MAX + 1` (Line 15).
  - Within `proc.c`, the following preexisting functions were modified to manage the new ready lists and priorities:
    * `userinit()`. The ready lists are initialized in `userinit()` with the other lists by looping through all ready lists from 0 to `MAX` and assigning 0 to them (Lines 166 – 168). Also, the first process is placed in the 0th ready list after successful creation (Line 206).
    * `allocproc()`. The per-process priority field `priority` was initialized to be 0 (the highest priority) in the `allocproc()` function (Line 141).
    * `fork()`. After successful creation, the child process is inserted in the highest priority queue (0th queue) (Line 288).
    * `exit()`. To search for abandonded children within all the new ready list structures, additional looping from the 0th indexed priority queue to the `MAX` indexed priority ready list was required (Lines 370 –376).
    * `wait()`. Similar to `exit()`, with the new ready lists, `wait()` required additional looping from 0 to `MAX` to adequately search for children (Line 484 – 491).
    * yield(). The function now requires insertion of the previosuly running process, `proc`, into the correct ready list indexed by `proc->prio` (Line 666).
    * wakeup1(). Insertion into the appropriate ready list from the sleping list is achieved by selecting the ready list by `p->prio` based indexing (Line 761).
    * kill(). The `kill()` function must search through all ready lists to look for a process PID, as such, it loops and indexes from 0 to `MAX` in the same fashion as `exit()` and `wait()` (Lines 808 – 816). Also, if `kill()` needs to wake a process before killing, it must insert into the ready list indexed by the priority of the process (Line 845).
  - A new helper function, `asserPrio()` was created which asserts that the priority of a process matches the priority of the list which it was removed from in order to maintain th einvariance of priorities across ready lists (Lines 1093 – 1100).

MLFQ

The `scheduler()` function in `proc.c` was changed to facilitate a MLFQ scheduling algorithm. The scheduler, like `exit()`, `wait()`, and `kill()`, also requires the looping methodology of searching all ready lists starting at 0 down to `MAX` (Lines 591 – 617). The order of the loop is to guarantee that the process removed from the head of the ready list is in the highest available queue. In the advent of a successfully found, highest priority process, the loop breaks and starts at 0 again to ensure that the scheduler is always checking the highest priority first. Since `scheduler()` also removes processes from the ready list, and assertion of both the state and priority is required (Lines 600 – 603).

Promotion

The following files were modified to include periodic priority promotion which is a means of starvation prevention:

- `param.h`. A new macro, `TIME_TO_PROMOTE`, was included (Line 21). The value of this macro was chosen via *hand tuning*.
- `proc.c`. The folowing were added to the file:

  - A new field, `uint PromoteAtTime` was added to the `ptable struct` (Line 29). This is to determine the time, in `ticks`, at which promotion of all processes will occur. The new field is intialized in `userinit()` to the macro `TICKS_TO_PROMOTE` (Line 159).
  - The promotion decision is handled in `scheduler()` (Lines 589 – 590). Once the global variable `ticks` reaches or exceeds the promotion timer, a helper function `prioAdjust()` is called.
  - `prioAdjust()` handles the actual promotion of all active processes (Lines 1225 – 1247). All active processes are searched and if their priority is nonzero then it is decremented (an increase in priority). Processes in the ready state require all possible queues from the 1th to the MAX index be searched. If a processes exists in said list then it is removed, the priority adjusted, and then enqueued to the tail of the next highest priority queue. The per-process budget (discussed in more detail later) is left unadjusted. Finally, the `PromoteAtTime` field is updated to be the current `ticks` value plus the `TICKS_TO_PROMOTE` value.

- Priority adjustment (promotion) is required to prevent starvation such that long running jobs with lower priority are not potentially left unrun due to the existance of higher priority processes.

Demotion

The following files were modified to include the demotion portion of the MLFQ algorithm:

- `param.h`. Another macro, `BUDGET`, was added to set the *CPU time budget* of all processes (Line 22).
- `proc.h`. A new field, `int budget`, was added to the `proc struct` (Line 79). This is the *per-process* budget which will be used for demotion.
- `proc.c`. The new `budget` field of each process is initialized to the `BUDGET` macro within `allocproc()`. The decision to demote occurs via a call to the helper function `budgetUpdate()` before an active process enters the `sched()` function. This happens in yield (Lines 665) and `sleep()` (Lines 722). `budgetUpdate()` checks whether the process spent more time in the CPU than budgeted; if so, the priority is decreased by 1 level (incrementing the value of `prio` by one) and the *per-process* budget is reset (Lines 1249 – 1260).

Set Priority System Call

The following files were modified to add the `setpriority()` system call.

- `user.h`.
- `syscall.h`. The `setpriority()` system call number was created by appending to the existing list (Line 32).
- `syscall.c`. Modified to include the kernel–side function prototype (Line 114); an entry into the function dispatch table `syscalls[]` (Line 152); and an entry into the `syscallnames[]` array to print the system call name when the `PRINT_SYSCALLS` flag is defined (Line 192). All prototypes hre are defined as taking a *void* parameter as the function call arguments are passed into the kernel on stack. Each implementation (e.g. `sys_setpriority()`) retrieves the arguments from the stack according to the syntax of the system call.
- `usys.S`. The user–side stub for the new system call was added (Line 40).
- `sysproc.c`. Contains the kernel–side implementation of the system call in `sys_setpriority()` (Lines 166 – 175). This routine removes two integers from the stack and passes them into the new routine `setpriority()` in `proc.c`.
- `proc.c`. Contains the routine `setpriority()` (Lines 1185 – 1223). The funciton takes two integer arguments, `pid` and `priority`, where `pid` is the PID of the process which will have its `prio` field set to `priority`. If the PID is not found or the priority is outside of the bounds of 0 and `MAX`, then the function returns an appropriate integer to denote failure (–1). All active processes are searched for the given PID. For *sleeping* or *running* processes, the priority is simply changed. For ready processes, all ready lists must be searched. If the priority of a ready process is changed then it must be removed from the current ready list, update the priority, reset the budget, and then enqueue into the new ready list. Success is denoted by `setpriority()` returning 0.

Update Commands

The following console commands were updated to reflect the addition of priorities:

- The control–p output now displays the priority of each active process under the column header `Prio` (Lines 883, 925).
- Control–r was modified to now display each ready list priority level indexed from 0 to MAX along with the list's contents or `EMPTY` if the list is empty (Lines 1117 – 1129). In addition, the command also prints the *per-process* budget of each process in a ready list in the form $(PID_i, BUDGET_i)$ where i is the priority level (Line 1124).
- The `ps` command was changed to display the new process priority by modifyng the following files:
  - `uproc.h`. A new field, `uint prio`, was added to the `uproc struct` which will contain the priority of the process (Line 12).
  - `proc.c`. `getprocs()` was changed to include copying a process' priority to the `uproc table` (Line 1011).
  - `ps.c`. The `printtable()` was changed to include a `Prio` column lable in the header (Line 82); it also now prints the priority as well (Line 96).

Testing

Round Robin Scheduling

I tested that round robin scheduling is maintained in a single priority level using the provided test program, `testSched.c`, which creates slowly runnning children, in conjunction with the updated control–r command to display the budget. I expected that within a single priority level, no process at the end or middle of the queue should run before the process at the front of the queue. Here is the output for the test:

As expected, the budget of processes does not change before all other processes before it run and thus

```
Prio 0: (14, 214) -> (12, 1418) -> (7, 204) -> (6, 1322) -> (9, 630) -> (5, 1220) -> (13, 191) -> (10, 741) -> (11, 841)
Prio 1: EMPTY
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
Ready List Processes:
Prio 0: (9, 590) -> (5, 1180) -> (13, 151) -> (10, 701) -> (11, 801) -> (4, 131) -> (8, 461) -> (14, 165) -> (12, 1370)
Prio 1: EMPTY
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
```

Figure 1: Round Robin SCheduling for Single Priority Levels Test

have their budget updated.
This test `PASSES`.

Promotion of Sleeping and Running Processes

I tested the promotion of sleeping/running processes using a small test program, `testsetprio.c`, which created sleeping processes and set their priority to be very low (i.e. the priorirty is set to be `MAX`) so that the change in priority could be observed with the control–p commnad. I expected that over time, the priority would be adjusted upward. The output for the test is below:

As seen in the control–p output above, the prio of the sleeping processes floated upwards over time after

```
Setting PID(5) to be MAX
Success!!

PID     Name            UID     GID     PPID    Prio    Elapsed CPU     State   Size
5       testsetprio     0       0       2       2       2.182   0.183   sleep   12288
6       testsetprio     0       0       5       0       2.153   0.072   zombie  12288
2       sh              0       0       1       0       49.913  0.045   sleep   16384
1       init            0       0       1       0       49.940  0.036   sleep   12288

PID     Name            UID     GID     PPID    Prio    Elapsed CPU     State   Size
5       testsetprio     0       0       2       1       2.589   0.215   run     12288
6       testsetprio     0       0       5       0       2.560   0.072   zombie  12288
2       sh              0       0       1       0       50.320  0.045   sleep   16384
1       init            0       0       1       0       50.347  0.036   sleep   12288
```

Figure 2: Sleeping Priority Change Test

being set to the minimum priority. The same holds true for running processes which I tested in a similar manner but lost the picture.
Thus, this test `PASSES`.

Budget Reset by Demotion

I tested the budget reset feature of demotion by using the provided test pogram, `testSched.c`, in conjunction with the updated control–r command which also displays the budget. I expected that upon demotion (as seen by a process moving to a lower priority) the budget will be reset to the value of `BUDGET` or 1700.

The output for this test is bellow:
As seen above, when a process is demoted, the budget is reset to the starting value of 1700.

```
Ready List Processes:
Prio 0: (10, 591) -> (11, 691) -> (4, 21) -> (8, 351) -> (14, 55) -> (12, 1260) -> (7, 44) -> (6, 1162) -> (9, 470)
Prio 1: EMPTY
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
Ready List Processes:
Prio 0: (9, 420) -> (5, 1011) -> (10, 531) -> (11, 631) -> (8, 291)
Prio 1: (4, 1700) -> (13, 1700) -> (7, 1700) -> (14, 1700)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
```

Figure 3: Budget Reset Test

This test `PASSES`.

MAX Priority of 2

The testing of the new MLFQ scheduler with a `MAX` priority of 2 was divided into 3 sub–tests all which had the `MAX` parameter in `param.h` changed to 2 and all three utilized the `testSched` test program.
For the first sub–test, I tested that the scheduler selects processes from the highest priority queue by observing the change in output of control–r over time as well as the output of control–p. I expected that *running* processes in the control–p output should have the highest priority (genereally zero). The output for this test can be seen below:
As expected, the processes in the *running* state have the highest priorities.

```
Ready List Processes:
Prio 0: (7, 670) -> (9, 371) -> (8, 32) -> (4, 565) -> (11, 81) -> (10, 920) -> (13, 1020) -> (12, 540)
Prio 1: (5, 1700)
Prio 2: EMPTY
Ready List Processes:
Prio 0: (13, 980) -> (12, 500) -> (6, 119) -> (14, 163) -> (7, 621) -> (9, 321) -> (4, 515)
Prio 1: (5, 1700) -> (8, 1700)
```

Figure 4: MAX of 2 Highest Priority Test

This sub–test `PASSES`.

For the second sub–test, I verified that processes correcly move between ready lists during promotion by observing the change in the output of control–r over time. I expected that promoted processes to move to the next highest ready queue (unless they were already in the highest queue). Here is the output from the test:
One caveat when interpretting the output, is that the control–r output requires the `ptable` lock so there is

```
Prio 0: (13, 980) -> (12, 500) -> (6, 119) -> (14, 163) -> (7, 621) -> (9, 321) -> (4, 515)
Prio 1: (5, 1700) -> (8, 1700)
Prio 2: EMPTY
Ready List Processes:
Prio 0: (12, 450) -> (5, 1670) -> (6, 70) -> (14, 112) -> (7, 571) -> (9, 272) -> (4, 464)
Prio 1: (8, 1700) -> (11, 1700)
Prio 2: EMPTY
```

Figure 5: Promotion for MAX of 2 Test

some delay between each process being promoted and the current ordering of the queue above. That aside, it was reasonably clear that when a process floated up, it was enqueued to the end of the next highest queue (in this case from 1 to 0) which is the correct behavior.
Thus, this sub–test `PASSES`.

For the last sub–test, I verified that processes correcly move down ready lists during demotion by observing the change in the output of control–r over time while running the `testSched` program in the background. I expected that demoted processes to move to the next lowest ready queue (unless they were already in the lowest queue) and for their budget to be reset to the value of `BUDGET` (1700). Here is the output from the test:
It was reasonably clear from the output above that when a process was demoted, it was enqueued to the

```
Prio 0: (7, 670) -> (9, 371) -> (8, 32) -> (4, 565) -> (11, 81) -> (10, 920) -> (13, 1020) -> (12, 540)
Prio 1: (5, 1700)
Prio 2: EMPTY
Ready List Processes:
Prio 0: (13, 980) -> (12, 500) -> (6, 119) -> (14, 163) -> (7, 621) -> (9, 321) -> (4, 515)
Prio 1: (5, 1700) -> (8, 1700)
Prio 2: EMPTY
```

Figure 6: Demotion for MAX of 2 Test

end of the next lowest queue which is the correct behavior.

Thus, this sub–test `PASSES`.
Since all sub–tests passed, this test `PASSES`.

MAX Priority of 6

The testing of the new MLFQ scheduler with a `MAX` priority of 6 was divided into 3 sub–tests all which had the `MAX` parameter in `param.h` changed to 2 and all three utilized the `testSched` test program.
For the first sub–test, I tested that the scheduler selects processes from the highest priority queue by observing the change in output of control–r over time as well as the output of control–p. I expected that *running* processes in the control–p output should have the highest priority (genereally zero). The output for this test can be seen below:
As expected, the processes in the *running* state have the highest priorities.

```
Prio 0: (9, 420) -> (5, 1011) -> (10, 531) -> (11, 631) -> (8, 291)
Prio 1: (4, 1700) -> (13, 1700) -> (7, 1700) -> (14, 1700)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
Ready List Processes:
Prio 0: (6, 1031) -> (9, 341) -> (5, 932) -> (10, 451) -> (8, 221)
Prio 1: (4, 1700) -> (13, 1700) -> (7, 1700) -> (14, 1700)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
```

Figure 7: Selection for MAX of 6 Test

This sub–test `PASSES`.

For the second sub–test, I verified that processes correcly move between ready lists during promotion by observing the change in the output of control–r over time. I expected that promoted processes to move to the next highest ready queue (unless they were already in the highest queue). Here is the output from the test:
One caveat when interpretting the output, is that the control–r output requires the `ptable` lock so there is

```
Prio 0: (6, 1031) -> (9, 341) -> (5, 932) -> (10, 451) -> (8, 221)
Prio 1: (4, 1700) -> (13, 1700) -> (7, 1700) -> (14, 1700)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
Ready List Processes:
Prio 0: (12, 1009) -> (8, 92) -> (11, 431) -> (4, 1580) -> (6, 902) -> (9, 211)
Prio 1: (13, 1700) -> (7, 1700) -> (14, 1700)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
```

Figure 8: Promotion for MAX of 6 Test

some delay between each process being promoted and the current ordering of the queue above. That aside,

it was reasonably clear that when a process floated up, it was enqueued to the end of the next highest queue (in this case from 1 to 0) which is the correct behavior.
Thus, this sub–test `PASSES`.

For the last sub–test, I verified that processes correcly move down ready lists during demotion by observing the change in the output of control–r over time while running the `testSched` program in the background. I expected that demoted processes to move to the next lowest ready queue (unless they were already in the lowest queue) and for their budget to be reset to the value of `BUDGET` (1700). Here is the output from the test:
It was reasonably clear from the output above that when a process was demoted, it was enqueued to the

```
Prio 0: (10, 591) -> (11, 691) -> (4, 21) -> (8, 351) -> (14, 55) -> (12, 1260) -> (7, 44) -> (6, 1162) -> (9, 470)
Prio 1: EMPTY
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
Ready List Processes:
Prio 0: (9, 420) -> (5, 1011) -> (10, 531) -> (11, 631) -> (8, 291)
Prio 1: (4, 1700) -> (13, 1700) -> (7, 1700) -> (14, 1700)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
```

Figure 9: Demotion for MAX of 6 Test

end of the next lowest queue which is the correct behavior.
Thus, this sub–test `PASSES`.
Since all sub–tests passed, this test `PASSES`.

MAX Priority of 0

The testing of the new MLFQ scheduler with 0 levels (`MAX` priority of 0) was divided into two sub–tests. For the the first sub–test, I tested that the scheduler behaves like a simple round robin scheduler. I did this using the `testSched` test program running in the background while observing the control–r command's output. I expected the scheduler to continue selecting the head of the ready queue and never processes in the middle. The output was the following:

As seen in the output, the scheduler maintains Round Robin scheduling.

```
Ready List Processes:
Prio 0: (12, 202) -> (9, 112) -> (11, 171) -> (10, 140) -> (13, 219) -> (14, 261) -> (4, 1660) -> (6, 1700) -> (7, 30)
Ready List Processes:
Prio 0: (13, 179) -> (14, 221) -> (4, 1620) -> (6, 1660) -> (7, 1690) -> (8, 29) -> (5, 1630) -> (12, 152) -> (9, 62)
Ready List Processes:
```

Figure 10: MAX Priority of 0 Tests

Thus, this sub–test `PASSES`.

For the second sub–test, I tested that promotion and demotion have no affect on the queue by using the previous testing methodology and output. I expected no programs to change position with a budget reset and that with no need to change priority, promotion would also have no affect. I used the output above for both tests. No processes change their position in the queue (only their budget changes) as a result of promotion and demotion.

This sub–test `PASSES`. Since both sub–tests passed, this test `PASSES`.

Set Priority System Call

Testing the new `setpriority()` system call will be divided into four sub–tests.

For the first sub–test, I tested that the function changes both the priority and budget of a process. I tested this by running the `testSched` test program in the background which periodically calls `setpriority()` and observe the change in priority and budget using the control–r command. I expected the priority to change to the new value (ignoring possible changes due to promotion) and for the budget to be reset to the value of `BUDGET`. The output for the test is below:
In the output above I observed that after a priority is set to some new value, both the priority value

```
10: new prio 5
Ready List Processes:
Prio 0: (14, 1290) -> (11, 890) -> (9, 29) -> (7, 680) -> (6, 970) -> (13, 1081) -> (8, 1660) -> (5, 341)
Prio 1: EMPTY
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: (10, 1690)
Prio 6: EMPTY
```

Figure 11: Set Priority Test

and budget value correctly change (the value of `BUDGET` for this test was 1700 minus some time since the process was running during the `setpriority()` call).
This sub–test `PASSES`.

For the second sub–test, I tested that ready processes which change priority are correctly moved to a new queue again using the `testSched` test program in the background in conjunction with the control–r command. I expected processes to move to a new ready list of the corresponding priority (or possibly one different due to promotion timing events).. The output will be the same as the previous test. As shown by the output, after updating a process' priority, it is placed in a new queue.
This sub–test `PASSES`.

The next sub–test tested that calling `setpriority()` on a process with the same priority does not change the plac in the queue. I tested this using the same methodology as the past two sub–tests. I expected that the placement of a process in the ready queue does not change due to a `setpriority()` call with the same priority (with some possibility of error due to processes running between the system call and the control–r output). The output was the following:
The output conformed to the expected results. After setting the priority to the same priority, the relative

```
5: new prio 0
7: new prio 2
Ready List Processes:
Prio 0: (11, 160) -> (6, 720) -> (12, 441) -> (14, 530) -> (5, 1420) -> (10, 1290) -> (9, 1099) -> (8, 910)
Prio 1: (7, 1690)
Prio 2: EMPTY
Prio 3: EMPTY
Prio 4: EMPTY
Prio 5: EMPTY
Prio 6: EMPTY
```

Figure 12: Set Same Priority Test

location of the processes did not change though there was a slight change in the overall queue due to processes running.
Thus, this subtest `PASSES`.

For the last sub–test, I tested the new `setpriority()` system call with invalid arguments using the `testsetprio` test program. I expected the system call to gracefully fail and return –1 (whihc will be indicated in the test program). The output can be seen below:
The new system call handles both invalid PIDs and priorities as expected.

```
Setting PID(4) to be 0
Success!!
Setting PID(4) to be MAX
Success!!
Setting PID(4) to be an invalid priority
setpriority call failed as expected.
Calling setpriority with an invalid PID(74)
setpriority call failed as expected.
```

Figure 13: Invalid PID and Priority Test

This sub–test `PASSES`.
Since all sub–tests passed, this test `PASSES`.

Updates Commands

The testing of the new updated commands was broken into three sub–tests.
For the first sub–test, I used the `testSched` program in the background and compared the priority output of control–p to the information in the control–r output. I expected the two commands to display the same priority values for processes across the two displays. The output is below:
I observed the same priority across both displays.

```
Ready List Processes:
Prio 0: (12, 430) -> (6, 161) -> (5, 765) -> (7, 541) -> (9, 1310) -> (8, 851) -> (14, 1590) -> (13, 650)
Prio 1: (10, 1700)
Prio 2: EMPTY

PID     Name        UID     GID     PPID    Prio    Elapsed CPU     State   Size    PCs
14      testSched   0       0       4       0       19.280  3.546   runble  12288
13      testSched   0       0       4       0       19.301  2.787   runble  12288
12      testSched   0       0       4       0       19.338  3.018   runble  12288
11      testSched   0       0       4       0       19.370  3.136   runble  12288
10      testSched   0       0       4       1       19.391  3.407   runble  12288
9       testSched   0       0       4       0       19.412  3.821   run     12288
8       testSched   0       0       4       0       19.431  4.280   run     12288
7       testSched   0       0       4       0       19.467  2.907   runble  12288
6       testSched   0       0       4       0       19.480  3.287   runble  12288
4       testSched   0       0       1       0       19.520  4.116   runble  12288
5       testSched   0       0       4       0       19.494  4.381   runble  12288
2       sh          0       0       1       0       25.230  0.109   sleep   16384   80105524 80100aaf 80101fe9 801012af 80106f0f 80106d52
1       init        0       0       1       0       25.284  0.041   sleep   12288   80105524 8010511b 80107b52 80106d52 801080ad 80107ea8
```

Figure 14: Updated Control−p Test

This sub–test `PASSES`.

The second sub–test was similar, I used the `testSched` program in the background and compared the priority output of the `ps` command to the information in the control–r output. I expected the two commands to show the same priority values for processes across the two displays. The output is below:
I observed the same priority across both displays.

```
$ Ready List Processes:
Prio 0: (14, 632) -> (4, 884) -> (6, 471) -> (10, 1222) -> (13, 1320) -> (8, 493) -> (12, 989) -> (11, 577) -> (9, 875)
Prio 1: EMPTY
Prio 2: EMPTY
ps
PID     Name        UID     GID     PPID    Prio    Elapsed CPU     State   Size
17      ps          0       0       2       0       0.234   0.010   run     49152
14      testSched   0       0       4       0       5.684   1.228   runble  12288
13      testSched   0       0       4       0       5.757   0.540   runble  12288
12      testSched   0       0       4       0       5.807   0.871   runble  12288
11      testSched   0       0       4       0       5.861   1.283   runble  12288
10      testSched   0       0       4       0       5.941   0.638   runble  12288
9       testSched   0       0       4       0       5.982   0.975   run     12288
8       testSched   0       0       4       0       6.021   1.367   runble  12288
7       testSched   0       0       4       0       6.056   1.164   runble  12288
6       testSched   0       0       4       0       6.069   1.389   runble  12288
4       testSched   0       0       1       0       6.107   0.976   runble  12288
5       testSched   0       0       4       0       6.080   1.440   runble  12288
2       sh          0       0       1       0       10.478  0.109   sleep   16384
1       init        0       0       1       0       10.504  0.032   sleep   12288
```

Figure 15: Updated PS Command Test

This sub–test `PASSES`.

The verification of the contents for control–r can be seen across the plethora of other tests.
This sub–test `PASSES`.
Since all sub–tests passed, this test `PASSES`.