Alex Ferguson
5/9/17
CS333

Project 4

Project Description

Project 4 was geared around having us implement a multi-level feedback queue and all the relevant functionality that accompanies a multi-level feedback queue. Changes to the ready list implemented in Project 3 allowed for multiple levels of queues in our ready list instead of just a single queue. These changes introduced us to the concept of process starvation and involved some other changes to the xv6 system in order to appropriately handle and avoid cases where a process might starve. We learned about process promotion, demotion, and the use of a budget to distinguish between shorter and longer running processes and how to handle them in a multi-level feedback queue.

Project Deliverables

The following features were added to the xv6 operating system:

1) Changes were made to the proc structure (and the uproc) structure to accommodate the new budget and priority values for each process (just priority for uproc). These new fields are set as unsigned integers.
   uint priority
   uint budget
2) New helper functions were created to properly assign a priority to a process while it is in the active state; one for while a process is in one of the ready lists and one for while a process is in either the sleep or running list.
   static int search_and_set(struct proc** sList, int pid, int prio)
   static int search_and_set_ready(int pid, int prio)
3) Modifications to the ready list from Project 3 were made in order to change the ready list from a single list into an array of multiple ready lists (queues).
   struct proc* ready[MAX+1]
4) Modifications were made to the add_to_ready function in order to accommodate the newly created array of ready lists instead of just a single ready list. The prototype and argument list remained the same.
   static int add_to_ready(struct proc* p, enum procstate state)
5) A new system call was added in order to test that processes are being assigned a priority appropriately during execution.
   int setpriority(int pid, int priority)
6) New constants were defined in order to accommodate the new number or ready lists, the default budget for a process in ticks, and the time in ticks that should elapse between each promotion:
   #define TICKS_TO_PROMOTE XXX
   #define DEFBUDGET XXX
   #define MAX N
   where XXX represents an integer ticks value and N represents the # of priority queues – 1.
7) A new field was added to the ptable structure to accommodate the time in ticks when a promotion should occur.
   uint promote_at_time
8) New helper functions were created to properly handle a priority promotion.
   static int priority_promotion()
   static int promote_list(struct proc** list)

9) Modified the ctrl-p sequence and the ps command to output the new priority information related to each process.
10) Modified the ctrl-r sequence to output processes and budgets in the newly created array of queues.
    Ready List Processes:
    Queue 0: (pid, budget) → (pid, budget) → (pid, budget)
    ….
    Queue MAX: (pid, budget) → (pid, budget) → (pid, budget)
    where each pid is the pid of a process in that queue.

Implementation

1) Changes to the proc structure and the uproc structure
   Files Modified: proc.h, uproc.h
   A priority and a budget were added to the proc.c file (lines 73 and 74). The priority of a process corresponds to the queue that the process gets added to in the array of ready queues. The budget is used to distinguish between shorter and longer running processes. Longer running processes tend to use up their budget and get added to the lower priority queues. The lower priority queues correspond to higher priority numbers (numbers closer to MAX).
2) Changes to the ready list
   Files Modified: proc.c
   Two changes were made to the way the ready list is handled:
   - A declaration for an array of ready queues in proc.c (line 24) was added to change the single ready queue from project 3 to an array of ready queues, simulating the levels of a multi-level feedback queue. The array is of size MAX+1 where MAX is defined below.
   - The add_to_ready function (lines 1125-1156) was modified to use the new array of ready queues instead of just a single ready queue (lines 1143-1155). The overall change to the behavior of the function is fairly minor, in that the function now uses the priority of the process passed in as an argument to determine which index of the array of ready queues to insert the process into. Using some pseudocode this looks like so:
     if (!ptable.pLists.ready[p->priority])
     add to the beginning of the list since its the only process
     struct proc* t = ptable.pLists.ready[p->priority];
     while (t→next)
     traverse to the end of the list
     t→next = p
     p→next = 0
     Doing the insertion this way maintains round robin scheduling for each queue in the array.

3) New setpriority system call
   Files Modified: usys.S (line 40), user.h (line 35), syscall.h (line 32), syscall.c (lines 109, 141, 176), sysproc.c (lines 173-182), defs.h, proc.c
   - A new system call was added to the xv6 operating system following the same procedure used in the prior projects to add a system call. The kernel side routine is implemented in sysproc.c and simply uses argint to pull the pid and priority arguments off of the stack (lines 179 and 180). It then offloads the work by returning a call to the set_priority helper function which takes those two arguments and does the work in proc.c. The set_priority helper function and functions used in its implementation are described below.

- A new prototype was added to the defs.h file (line 126). The function prototype looks like: **int set_priority(int pid, int priority)**
  The prototype was added here so that the setpriority system call implemented in sysproc.c could have a helper function in proc.c to gain access to the different lists that contain processes.
- The set_priority helper function is implemented in proc.c (lines 1184-1208) and called in the setpriority system call function. It takes two integer arguments: a pid and a priority. First, set_priority performs two checks; one to make sure that the pid is greater than 0 and the other check makes sure that the priority passed in is in the range 0 < priority < MAX (lines 1187-1190). These two checks ensure that the pid and the priority entered are in a valid range and returns -1 if they are not. set_priority then uses a call to the holding() method (line 1192) to make sure that the lock is being held before proceeding to call on two helper functions; search_and_set and search_and_set_ready (lines 1194, 1198, 1202). These two functions are described next. If the return value from either of these functions are 0 that means that it successfully found and changed the priority of the process. The lock is then released and set_priority returns 0 to the calling routine to indicate success. Otherwise, the set_priority function releases the lock if it is being held and returns -1 to indicate that nothing was changed.
- A static declaration was added for the search_and_set function (line 55). The search_and_set function takes 3 arguments: a list argument, an integer pid argument, and an integer priority argument. It checks the list argument to make sure it isn't null. If it is it returns -1 to indicate there is nothing to search through. Otherwise the function traverses the list and checks the pid passed in against the pid of each process in the list. If no match is found -2 is returned to indicate no match. If it finds a matching process id it checks to make sure the priority argument is different than the process' current priority. If they are the same there is no need to change anything and 1 is returned to indicate its already that priority number. If they are different the process' priority is changed to that argument and 0 is returned to indicate the change happened successfully. This function handles the running and sleep lists since there is no need to adjust the process' ready queue upon changing the priority.
- A static declaration was added for the search_and_set_ready function (line 56). This function takes two integer arguments: a pid and a priority. This function is used with the newly created array of ready lists due to the fact that, when we change the priority of a process, it needs to be removed from its current priority queue and added to the queue corresponding to its new priority. A for loop is used to loop through the array and search through each queue for a process with an id that matches the integer id passed in. It functions the same as the search_and_set function except, when it finds a matching pid, it sets the priority of that process to the priority argument passed in to the function and then removes it from the current queue and then adds it to the new one. A 0 is returned to signify a successful priority change, removal, and and addition to the new priority queue. Otherwise the return value matches the return values used in the search_and_set function.

4) Scheduler modifications for priority promotion
   Files Modified: proc.c
   - A for loop was added to the scheduler to loop through the newly created array of ready queues. The loop starts with the highest priority queue (queue 0) and works its way to the lowest priority queue (MAX). At each step of the way, before checking to see if a process needs to be run, the scheduler checks to see if it is time to run a priority promotion (lines

572-575). It does this by checking to see if the value of the global variable ticks has reached the same value as the ptable.promote_at_time variable which is set to a default value of TICKS_TO_PROMOTE. If the two values are the same a priority promotion is done by running the priority_promotion() helper function followed by updating the value of the promote_at_time variable to the ticks value + the TICKS_TO_PROMOTE value (lines 573-574). Whether or not a priority promotion happens, the for loop proceeds by looking at each priority level for a process to dequeue in round robin fashion. When it finds one and moves it to running it then begins iteration through the for loop over again to check for processes added to the highest priority queue instead of continuing to lower priority queues (lines 576-594).

- The priority_promotion() helper function (lines 1238-1259) first checks to see whether or not the lock is being held. If the lock isn't currently held it acquires it (lines 1241-1242). If the value of MAX is 0 then -1 is returned to signify that a promotion doesn't need to occur. This is because, if there is only one queue then the scheduler should act as a simple round robin scheduler. If the value of MAX is greater than 0 that means there is more than one queue and the function proceeds by running the promote list function on both the running and sleep list (lines 1222-1223). The for loop starts in the highest priority queue (queue 0) and, on each iteration, checks to see if there are processes in the queue one lower than it in priority. If there are processes in the lower queue, the promote_list() helper function is run and the lower priority queue is then added to the end of the queue above it in priority. After the for loop is finished, if the lock is held, it is then released. A return value of 1 signifies that the promotion happened successfully.
- The promote_list() helper function (lines 1263-1276) takes a single argument: a pointer to a list of processes. This function simply traverses through the list passed in and decrements the priority value of each process in the list by 1. By doing this it ensures that when the processes are dequeued from the list and added back to the ready list, they are added to the appropriate priority queue.
- The yield and sleep functions were modified to accommodate the process' new budget value. When a process yields and gives up the CPU voluntarily or when the process leaves the running queue and enters the sleeping queue, its budget is calculated by subtracting the time the process spent in the CPU from its budget. If its budget is less than 0, as long as it hasn't already reached a priority value of MAX, its priority is demoted down a level and its budget is reset. The budget is reset so that, when the process is run again it can accurately determine whether it should be demoted again after finishing running. (yield: lines 652-656 sleep: lines 713-717)

5) Modifications to the ctrl-p sequence and ps command
   Files Modified: proc.c, ps.c
   - In the procdump function in proc.c the print statement that outputs process information for the ctrl-p sequence was modified to also output the process' priority value (lines 882 and 897).
   - The getproc_helper function in proc.c was modified to copy the process' priority value into the table of active uprocs (line 932). This in turn lets the ps command also output the priority value of each active process.
   - The for loop in the ps.c file was modified to also print out the priority value of each active process in the table of active processes (lines 28 and 29).

6) Modifications to the ctrl-r sequence
Files Modified: proc.c
The display_ready() function was modified to incorporate the new array of ready queues. The function now uses a for loop to iterate through each index of the array. At each index, if there are processes in the list at that index they are displayed and the loop continues.

7) Priotest command
Files Added/Modified: priotest.c, Makefile, runoff.list
The priotest command was added as a method of testing whether or not the setpriority system call works properly. It takes two command line arguments: the first is the pid of a process and the second is a priority. It uses the atoi routine to convert these two arguments to their integer equivalent and then passes those two arguments into the user-side setpriority function which then calls the kernel side setpriority function to change the priority of the process with the pid matching that passed into the command via the command line. Based on the return value of the function a relevant message is printed to the console indicating success or some sort of failure.

Testing

For my testing section the output from the ctrl-p sequence and the ps command shows the PRIORITY in the state column due to the fact that the long process name pushes that information over a bit.

1. Round Robin testing
   In order to test whether or not Round Robin scheduling is being enforced by the scheduler I will change the value of MAX to 0. By doing this, the scheduler will only have to work with one queue. I will then run the loopforever command to create processes that are constantly in the active state and, by pressing the ctrl-r sequence, we can hopefully see that processes are being dequeued from the front of the queue and added to the back of the queue in Round Robin fashion.

```
Queue 0: (9, 8783) -> (7, 8788) -> (10, 8948) -> (11, 9000) -> (5, 6241) -> (12, 8795)
Ready List Processes:
Queue 0: (12, 8795) -> (4, 8983) -> (6, 6842) -> (8, 8141) -> (9, 8782) -> (7, 8787)
Ready List Processes:
Queue 0: (6, 6841) -> (8, 8140) -> (9, 8781) -> (11, 9000) -> (7, 8786) -> (10, 8946)
Ready List Processes:
Queue 0: (10, 8946) -> (4, 8983) -> (5, 6239) -> (12, 8793) -> (6, 6840) -> (8, 8139)
Ready List Processes:
Queue 0: (5, 6238) -> (12, 8792) -> (6, 6839) -> (11, 9000) -> (8, 8138) -> (9, 8779)
Ready List Processes:
Queue 0: (6, 6839) -> (11, 9000) -> (8, 8138) -> (9, 8779) -> (10, 8945) -> (5, 6238)
Ready List Processes:
Queue 0: (12, 8791) -> (11, 9000) -> (6, 6838) -> (8, 8137) -> (9, 8778) -> (10, 8944)
Ready List Processes:
Queue 0: (7, 8783) -> (11, 9000) -> (12, 8790) -> (6, 6837) -> (8, 8136) -> (9, 8777)
Ready List Processes:
Queue 0: (7, 8783) -> (11, 9000) -> (12, 8790) -> (6, 6837) -> (8, 8136) -> (9, 8777)
Ready List Processes:
Queue 0: (6, 6837) -> (8, 8136) -> (9, 8777) -> (4, 8983) -> (10, 8943) -> (5, 6236) -> (11, 9000)
Ready List Processes:
Queue 0: (12, 8789) -> (6, 6836) -> (8, 8135) -> (4, 8983) -> (9, 8776) -> (10, 8942)
```

Figure 1: Round Robin Scheduling

As we can see in Figure 1 above, there is a consistent pattern showing that processes are removed from the front of the queue and added to the back. For example, as seen above in the very beginning process 9 is removed from the front of the queue and ran followed by process 7. In the next line we can see that both 9 and 7 have been added to the back of the queue after the other processes in the queue. We can also see this a few lines further down the image when process 10 is removed from the queue. We can see that processes 5 and 12, which came after process 10, are moving closer to the front of the queue. On the next line after that we see that process 10 has been added to the back of the queue as well as process 5 which was next on the

queue in the line above it. This shows consistent Round Robin scheduling behavior when a single queue is used. The same algorithm is used when the value of MAX is increased which means that Round Robin scheduling is enforced for all values of MAX.
Because the behavior shows consistent Round Robin scheduling:
The Round Robin test PASSES.

2. Setpriority system call testing
   The setpriority system call will contain four sub-tests in order to ensure proper functioning. In order for the setpriority system call test to pass as a whole, all four subtests must pass. For these tests the value of MAX was set to 5.
   - **setpriority non-runnable sub-test**
     In order to test whether or not the setpriority system call sets the priority of a non-runnable process successfully I will boot xv6 and use the ctrl-p sequence to show the initial two processes, init and sh, sitting in the sleep queue. I will then use the priotest command which uses the setpriority system call to change the priority of the sh process from its initial value to a different value. I will then use the ctrl-p sequence again to show that the value has changed. I expect the change to accurately reflect when the ctrl-p sequence is used to output the process information.

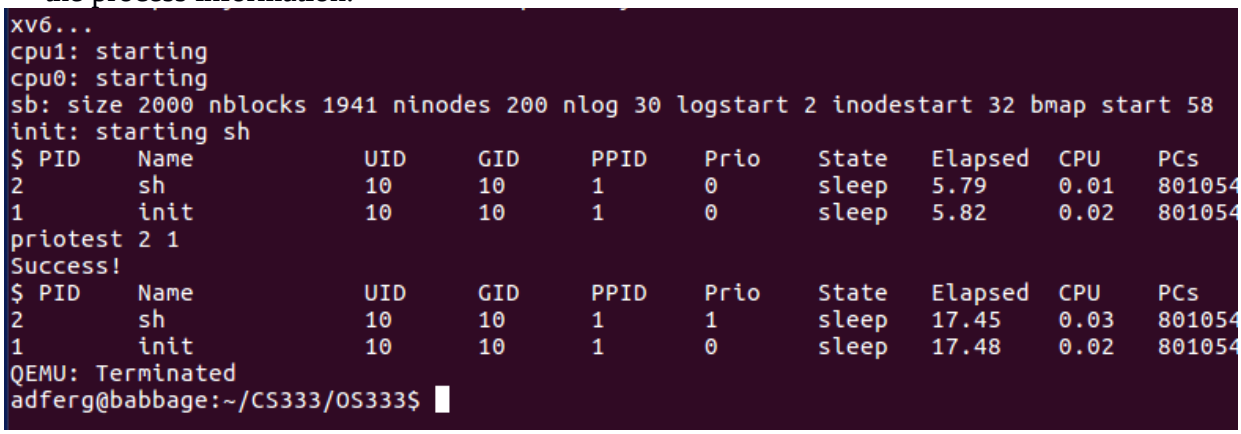


Figure 2: setpriority system call non-runnable

As we can see in Figure 2 above, the ctrl-p sequence shows the initial values for each of the processes in the sleep queue. When priotest is run and given the PID of the shell, 2, it successfully changes the priority of the shell from 0 to 1 (given as the second argument). The second press of ctrl-p shows the successful change from priority 0 to priority 1.
The setpriority non-runnable sub-test PASSES.

- **setpriority runnable sub-test**
  In order to test whether or not the setpriority system call successfully changes the priority of a runnable process and moves it to the correct priority queue I will use the loopforever command to create a bunch of processes that cycle between active states. I will use the ctrl-r sequence to show that the processes are in the queue corresponding to their priority and I will then use the priotest command to change a process' priority. I will follow that up with another ctrl-r press to show that the process has successfully moved from the old priority queue to the new one. I expect that the process will accurately change from its current priority queue to the priority queue matching the priority I changed it to.

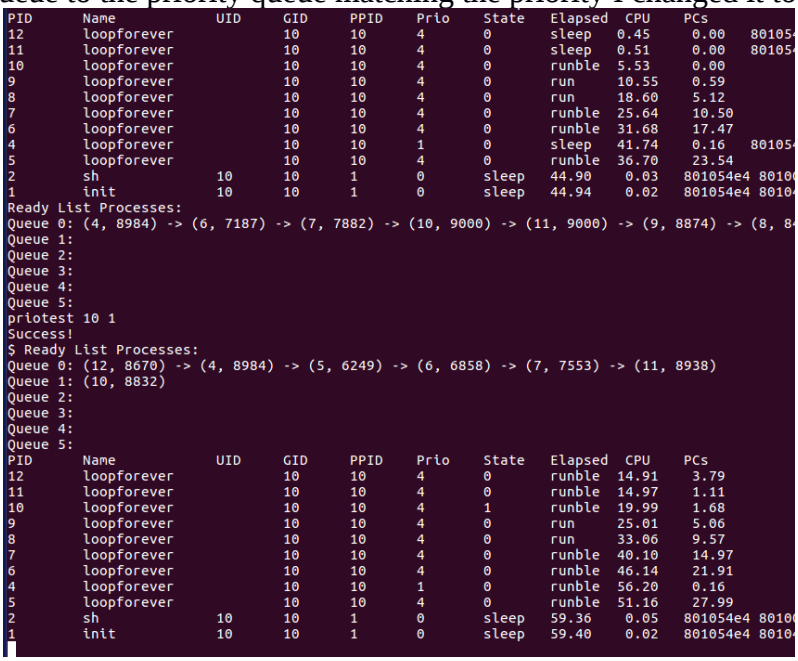


Figure 3: setpriority system call runnable

As we can see in Figure 3 above, loopforever creates a bunch of processes that cycle between active states. The ctrl-p sequence shows the process info for these processes and shows that there are processes in the runnable state. The ctrl-r sequence also verifies that the runnable processes are in the priority queue corresponding to their current priority. I chose to change the priority of runnable process 10 from 0 to 1. After the change the ctrl-r sequence shows that the process has successfully changed from priority queue 0 to priority queue 1. The ctrl-p sequence also verifies that process 10's priority accurately reflects the change. It has successfully changed the priority and moved to the correct priority queue.
The setpriority runnable sub-test PASSES.

- **setpriority invalid priority sub-test**
  In order to test whether or not the setpriority system call successfully handles an invalid priority I will boot xv6 and use the ctrl-p sequence to output actual process information. I will then use the priotest command and give it a VALID pid number and an INVALID priority number, once for a priority below the acceptable range (priority < 0) and once for a priority above the acceptable range (priority > MAX). Each time I will use ctrl-p to show that no changes have occurred to the process I tried to change. I expect that the priotest will accurately handle both cases by outputting a fitting error message and that no changes to the process will occur.

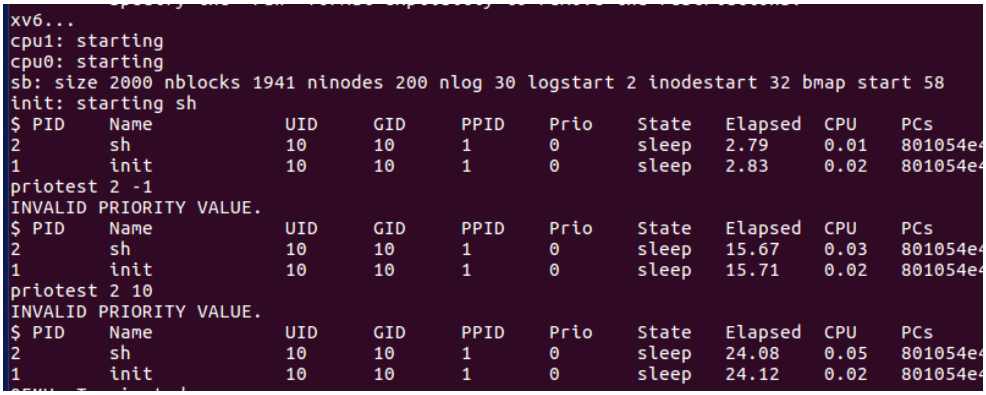


Figure 4: invalid priority

As we can see in Figure 4 above, when we give the setpriority system call a VALID pid and an invalid priority that is outside of the acceptable range (0 ≤ priority ≤ MAX) it successfully indicates failure and doesn't change the value of the process' priority. This is accurately seen when using the ctrl-p sequence as well.
The invalid priority sub-test PASSES.

- **setpriority invalid pid sub-test**

In order to test whether or not the setpriority system call successfully handles an invalid PID I will boot xv6 and use the ctrl-p sequence to output actual process information. I will then use the priotest command and give it an INVALID pid number and a VALID priority number. I will then use ctrl-p to show that no changes have occurred to any current processes. I expect the setpriority system call to indicate that an invalid PID was entered for any PID value that is not currently shown in the ctrl-p output.

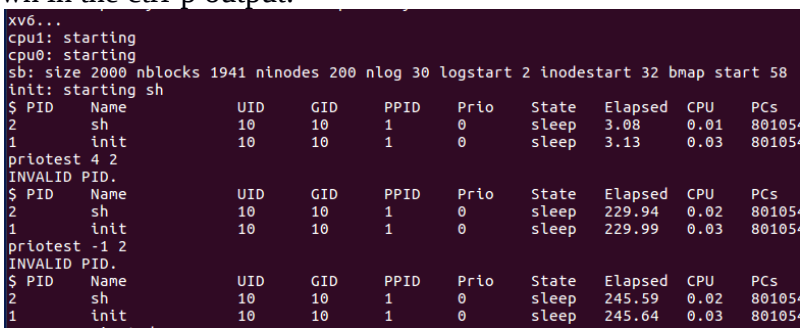


Figure 5: invalid PID

As we can see in Figure 5 above, when we give the setpriority system call an INVALID pid and a valid priority it successfully indicates failure and priotest outputs a valid error message. We can also see that no changes to current processes have occurred.
The invalid PID sub-test PASSES.
Because all setpriority system call sub-tests PASSED the setpriority test PASSES as a whole.

3. MLFQ testing

   The MLFQ testing section will contain two sub-tests, one to make sure that the MLFQ always selects the first process of the highest priority queue (queue 0) and one to make sure that the MLFQ functions appropriately for different values of MAX.

   - First process highest priority sub-test

     In order to test that the MLFQ grabs the first process off the highest priority queue I will first use the loopforever command to create processes that cycle between active states. I will then use the ctrl-r sequence to output the different level queues to see if I can find instances where the MLFQ is accurately choosing the right process to run. I expect that the output from the ctrl-r sequence will show that the process is being grabbed from the highest priority queue each time a process is run.

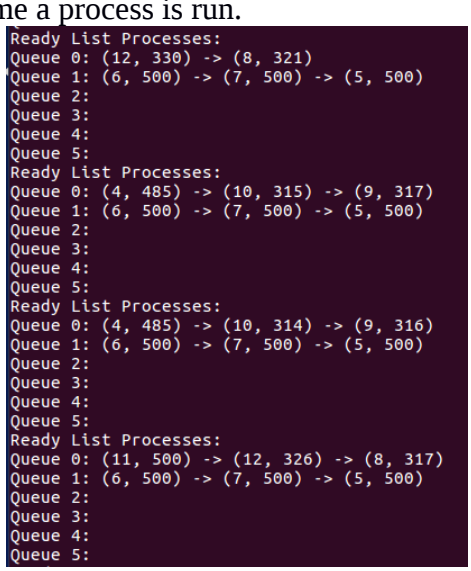


Figure 6: MLFQ Scheduling

As we can see in Figure 6 above, the ctrl-r sequence shows the ready queues as the processes are cycling between runnable and running. Even though there are processes sitting in queue 1 which is a lower priority than queue 0 they are not being run before queue 0. Processes in queue 0 are consistently being chosen by the scheduler while processes in queue 1 are still sitting there waiting to be run. We can also see an order to the way that the processes are being chosen in queue 0 between ctrl-r presses. Process 12 is removed before process 8 and we can see that they get added back in the last ctrl-r press at the bottom of Figure 6. This shows its picking the first process of the highest priority queue.

The first process highest priority sub-test PASSES.

- MLFQ MAX value sub-tests

  In order to test that the MLFQ functions properly for different values of MAX I will change the value of MAX in proc.c. I will test against MAX values of 0, 1, 3, and 7 and I will set the value of TICKS_TO_PROMOTE to a value that allows each of the processes to gradually float to the lower priority queues when the loopforever command is used. I expect that for a value of 0 the MLFQ will behave as a simple Round Robin scheduler with a single queue. For all other values the MLFQ will behave appropriately, allowing processes to float to lower priority queues and stay there once they hit the MAX level queue. This will be reflected when the ctrl-p and ctrl-r sequences are pressed.

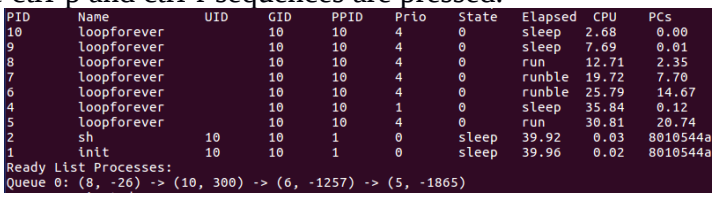


Figure 7: MAX value 0

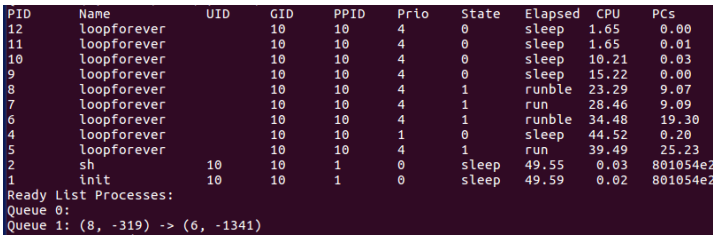


Figure 8: MAX value 1

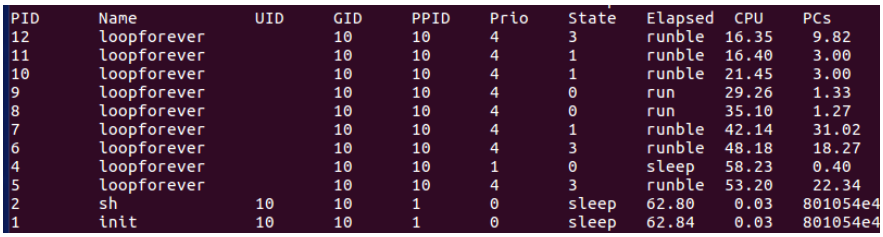


Figure 9: MAX value 3

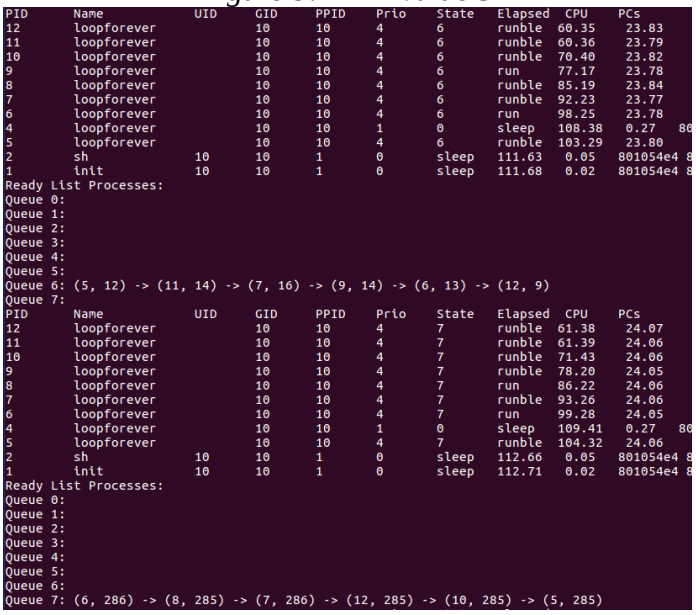


Figure 10: MAX value 7

As we can see in Figure 7 above, when the value of MAX is set to 0 there is no need to do any promotion and the scheduler functions as a simple Round Robin scheduler using a single queue. The output from ctrl-p and ctrl-r show this behavior. We can also see in Figures 8-10 that when the value for MAX is set to 1, 3, and 7 the scheduler acts as an MLFQ and the processes gradually use up their budget and float toward the lower priority queues in all instances. A combination of the ctrl-p and ctrl-r sequences also shows the priority of each of the processes and the queue that they are currently sitting in which matches that priority.
The MLFQ MAX value sub-test PASSES.
Because both sub-tests PASSED the MLFQ testing PASSES as a whole.

4. Promotion test
   In order to test that promotion is happening successfully I will set the value of TICKS_TO_PROMOTE to a value low enough to show that, as the processes are being run and demoted when they use up their budget, they occasionally get promoted at around that interval in time. I will demote processes after 4 seconds and promote active processes at around 7 seconds. I expect the priority values to fluctuate by being demoted by a level and then promoted a level, gradually lowering in priority as time continues.

```
PID     Name            UID     GID     PPID    Prio    State   Elapsed  CPU     PCs
10      loopforever             10      10      4       0       sleep   0.55     0.00   8010
9       loopforever             10      10      4       0       run     9.57     0.56
8       loopforever             10      10      4       1       run     17.59    8.78
7       loopforever             10      10      4       2       runble  24.61    12.30
6       loopforever             10      10      4       2       runble  30.65    15.30
4       loopforever             10      10      1       0       sleep   40.72    0.10   8010
5       loopforever             10      10      4       2       runble  35.67    18.30
2       sh              10      10      1       0       sleep   44.06    0.06   801054e4 801
1       init            10      10      1       0       sleep   44.10    0.03   801054e4 801
PID     Name            UID     GID     PPID    Prio    State   Elapsed  CPU     PCs
10      loopforever             10      10      4       0       sleep   3.06     0.00   8010
9       loopforever             10      10      4       1       run     12.08    3.07
8       loopforever             10      10      4       2       runble  20.10    9.57
7       loopforever             10      10      4       2       runble  27.12    12.87
6       loopforever             10      10      4       2       runble  33.16    15.88
4       loopforever             10      10      1       0       sleep   43.23    0.10   8010
5       loopforever             10      10      4       2       run     38.18    18.87
2       sh              10      10      1       0       sleep   46.57    0.06   801054e4 801
1       init            10      10      1       0       sleep   46.61    0.03   801054e4 801
PID     Name            UID     GID     PPID    Prio    State   Elapsed  CPU     PCs
10      loopforever             10      10      4       0       sleep   5.49     0.00   8010
9       loopforever             10      10      4       0       run     14.51    5.50
8       loopforever             10      10      4       1       runble  22.53    10.18
7       loopforever             10      10      4       1       runble  29.55    13.48
6       loopforever             10      10      4       1       run     35.59    16.48
4       loopforever             10      10      1       0       sleep   45.66    0.10   8010
5       loopforever             10      10      4       1       runble  40.61    19.48
2       sh              10      10      1       0       sleep   49.00    0.06   801054e4 801
1       init            10      10      1       0       sleep   49.04    0.03   801054e4 801
```

Figure 11: Priority Promotion

As we can see in Figure 11 above, as processes are running, using up their budget, and getting demoted the promotion occurs successfully and promotes the active processes by one level. Numerous other examples of this are shown in prior tests above.
The promotion test PASSES.

5. Demotion testing
   In order to test that demotion is happening successfully I will set the value of TICKS_TO_PROMOTE to a very high value and use loopforever to create a bunch of active processes. I will then use ctrl-p and ctrl-r to show that the processes are gradually floating to the lower priority queues. I expect to see a correlation between ctrl-p and ctrl-r presses and also expect to see the processes change queues across ctrl-r presses.

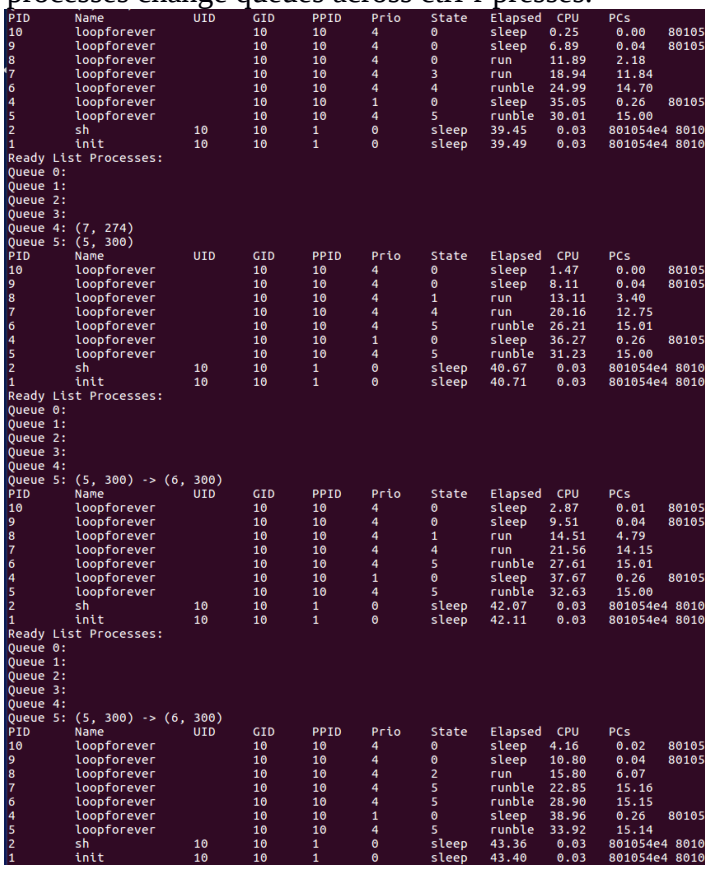
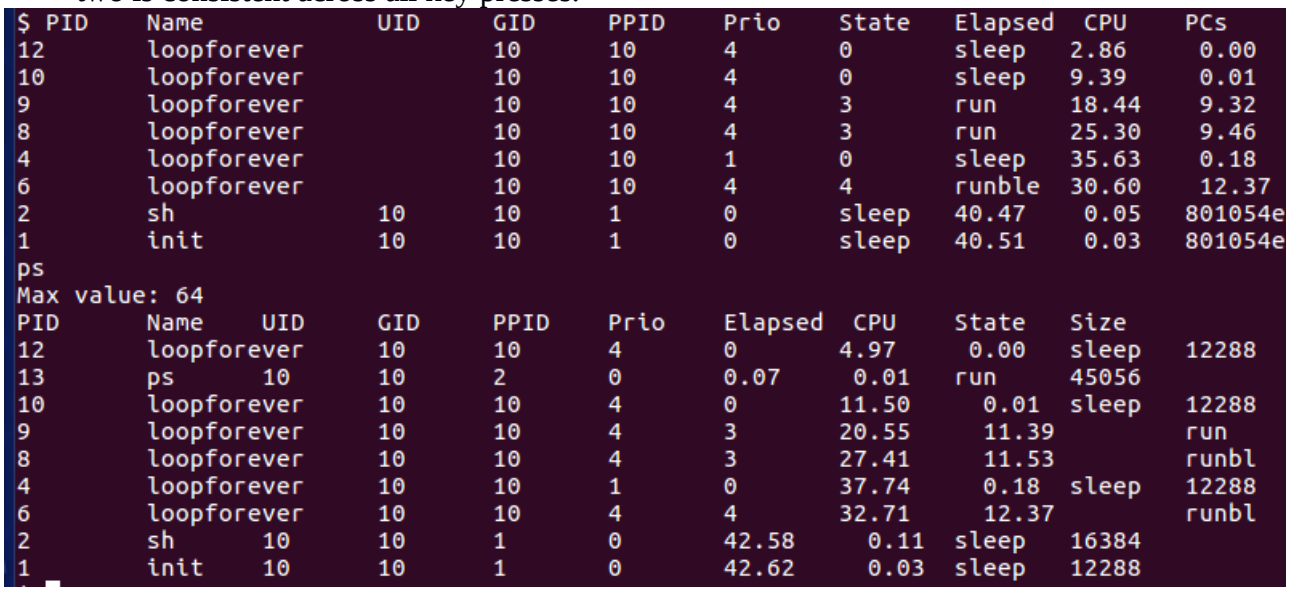


Figure 12: Demotion

   As we can see in Figure 12 above, the ctrl-p sequence shows that the priority of processes are gradually increasing as they cycle between runnable and running and use up their budget. The ctrl-r sequence also shows that the runnable processes are being placed in the lower priority queues corresponding to their priority. This behavior shows that demotion is occurring successfully.
   The demotion test PASSES.

6. Ctrl-P/Ctrl-R sequence testing
   I believe that the ctrl-p and ctrl-r sequences have been appropriately tested throughout prior tests in the testing section of my document. In many of the images above a combination of the ctrl-p sequence output and the ctrl-r sequence output show information that is accurate and supports and validates one another.
   For this reason the Ctrl-P/Ctrl-R sequence test PASSES.

7. PS command testing
   In order to test that the PS command is outputting accurate information I will use the loopforever command to create a bunch of active processes. I will then use a combination of the ctrl-p sequence and the ps command to show that the priority information being printed by the two is consistent across all key presses.



```
$ PID    Name            UID     GID     PPID    Prio    State   Elapsed  CPU    PCs
12       loopforever             10      10      4       0       sleep   2.86     0.00
10       loopforever             10      10      4       0       sleep   9.39     0.01
9        loopforever             10      10      4       3       run     18.44    9.32
8        loopforever             10      10      4       3       run     25.30    9.46
4        loopforever             10      10      1       0       sleep   35.63    0.18
6        loopforever             10      10      4       4       runble  30.60    12.37
2        sh              10      10      1       0       sleep   40.47    0.05   801054e
1        init            10      10      1       0       sleep   40.51    0.03   801054e
ps
Max value: 64
PID      Name    UID     GID     PPID    Prio    Elapsed  CPU    State   Size
12       loopforever     10      10      4       0       4.97     0.00   sleep   12288
13       ps      10      10      2       0       0.07     0.01   run     45056
10       loopforever     10      10      4       0       11.50     0.01  sleep   12288
9        loopforever     10      10      4       3       20.55     11.39         run
8        loopforever     10      10      4       3       27.41     11.53         runbl
4        loopforever     10      10      1       0       37.74     0.18  sleep   12288
6        loopforever     10      10      4       4       32.71     12.37         runbl
2        sh      10      10      1       0       42.58     0.11  sleep   16384
1        init    10      10      1       0       42.62     0.03  sleep   12288
```

Figure 13: PS command output

As we can see in Figure 13 above, the output from the ctrl-p sequence matches the output from the ps command which shows that the priority information is the same. Even though the output is a bit disgusting since I ran short on time the information is consistent across the two outputs. The PS command test PASSES.