Alex Ferguson
CS333
4/24/17

Project 3 Writeup

Project Description

Project 3 was geared around familiarizing ourselves with the concept of a few different things. First, we designed a new, more efficient method to manage process scheduling, process allocation and deallocation, “active” process state transitions, and zombie processing. This was accomplished using linked lists. Second, we added new control sequences to further expand console debugging functionality. Third, we learned about the issues surrounding the implementation of atomicity during these new process management techniques and the complications that arise related to concurrency.

Project Deliverables

The following features were added to the xv6 operating system:

A new flag was added to the makefile in order to ensure that all new functionality could easily be turned on and off to switch between the prior method of process management and the newly implemented method of process management.

1. Added new methods to ensure proper transition between states during a processes life cycle in xv6 and proper maintenance of the list invariant requirement.
   Prototypes:
   ```
   static void assert_state(struct proc* p, enum procstate state);
   static int remove_from_list(struct proc** sList, struct proc* p);
   static int add_to_list(struct proc** sList, enum procstate state, struct proc* p);
   static int add_to_ready(struct proc* p, enum procstate state);
   static void exit_helper(struct proc** sList);
   static void wait_helper(struct proc** sList, int* hk);
   ```

2. Added a new system of management for processes as they go from unused to creation to active to deallocation. This system of management incorporates a linear linked list for each possible process state:
   - Ready – holds processes that are considered runnable.
   - Free – holds shell processes that are used for allocation of new processes.
   - Sleep – holds processes that are waiting on a child process or some sort of I/O.
   - Zombie – holds processes that are ready to be reaped.
   - Running – holds processes that are currently running in the CPU.
   - Embryo – holds processes that are in the process of being created.

3. Added new console control sequences to print out information about the different lists to aid in the debugging process.
   - ctrl-r – prints PIDs for all processes in the ready list
     Output Format:
     Ready List Processes:
     1 → 2 → 3 → … → *n* (*n* = nth PID of process in sleep list)
   - ctrl-f – prints an integer value representing the number of processes in the free list
     Output Format:
     Free List Size: *N* processes (*N* = number of total processes in the free list)
   - ctrl-s – prints PIDs for all processes in the sleep list
     Output Format:
     Sleep List Processes:
     1 → 2 → 3 → … → *n* (*n* = nth PID of process in sleep list)
   - ctrl-z – prints both the PID and PPID for all processes in the zombie list
     Output Format:
     Zombie List Processes:
     (1, PPID1) → (2, PPID2) → … → (*n*, PPID*n*) (*n* = nth PID #, PPID*n* = *n*th PPID #)

     <u>Prototypes:</u>
     void free_length(void)
     void display_ready(void)
     void display_sleep(void)
     void display_zombie(void)

4. Added new test programs written by Mark Morrissey in order to meet minimum testing requirements for new features added to the xv6 operating system. These include:
   zombietest.c
   loopforever.c
   pstest.c

5. Added Round Robin scheduling to the way that xv6 adds and removes processes from the ready list.

6. Implemented code and structure to maintain the key invariant of having a process reside on one and only one list at any given time during execution.

<u>Implementation</u>

Since all of the functions listed in the project deliverables section 1 are used primarily during process management their implementation will be combined with the discussion of implementation of section 2 of the deliverables here.

1. New method of process management
   <u>Files Modified:</u> proc.c, proc.h

- A new struct was added to proc.c (lines 12-19) called StateLists. This struct contains the new lists that are used to hold processes as they transition between states in xv6. This struct contains the list for free (line 13), embryo (line 14), ready (line 15), running (line 16), sleep (line 17), and zombie (line 18).

- The proc structure was modified in proc.h to include a new next pointer (line 73). This next pointer allows for the creation of the linked list structures in the StateLists struct.
- A StateLists variable was added to the ptable structure (line 24) in proc.c, called pLists, that gives access to the lists that are initialized and manipulated while xv6 is booted.
- Static method prototypes for all helper functions used to manipulate the lists were added to proc.c (lines 35-40). The implementation of these files are as follows:
    - **assert_state(struct proc* p, enum procstate state)** proc.c - lines 985-981
      The assert state function takes a pointer to a proc and a procstate argument in order to compare the state of the proc argument passed in against that state. If the proc's state and that enum state match a simple return happens so that execution can continue. Otherwise, if the states dont match, a panic occurs with a message stating that the states do not match. The assert_state function is one of the key functions helping maintain the invariant that a process can be on one and only one list at a time.
    - **remove_from_list(struct proc** sList, struct proc* p)** proc.c – lines 994-1018
      The remove_from_list function takes two arguments: a list and a process. The function first checks both the list and the process against null to make sure that the arguments passed in actually contain a value. If this check fails a -1 is returned as an indication of failure. If the checks pass, a current pointer is set to the head of the list for traversal purposes. The function works by traversing through the list and comparing the memory address of the process passed into the function against the current pointer in the list. If the two memory addresses match then it has found the process that it is looking for and it removes it from the list and returns a 1 as a sign of success. If it makes it through the entire list without finding a matching memory address then a -1 is returned to signify it failed to find a matching address. Having the remove from list function work by searching for a matching memory address helps to maintain the key invariant of having a process on one and only one list at a time, ensuring that you cannot remove the process from the list incorrectly.
    - **add_to_list(struct proc** sList, enum procstate state, struct proc* p)** proc.c – lines 1021-1030
      The add_to_list function takes three arguments: a list, an enum procstate, and a process. It begins by checking to make sure that the process passed in is not null. If the process is null a -1 is returned to signify that the process is null and there is no reason to add anything to the list. If the process isn't null then an assert is performed to make sure that the state of the process being added to the list matches the state that the process is expected to be in, in order to be added into the list. E.G. - If the process being added to the zombie list isn't in the zombie state it will panic stating that the states do not match. This helps to maintain the key invariant of having a process on one and only one list at a time. If it passes the assert test then the process is added to the front of the list and the head of the list is updated to that newly added process. A zero is returned as a sign of successful addition to the front of the list.
    - **add_to_ready(struct proc* p, enum procstate state)** proc.c – lines 1033-1050
      The add_to_ready function takes two arguments: a pointer to a process and an enum procstate argument. The function first checks to make sure that the process passed in isn't null. If it is null a -1 is returned to signify that the process passed in was null and there is no need to add to the ready list. If it passes this check then an assert is performed to make sure that the process that is about to be added to the ready list is in the RUNNABLE state that it needs to be in, in order to be on the ready list. If the process isn't in the right state the assert_state function will panic and halt execution.
      Adding to the ready list is different than adding to other lists because it needs to follow

round robin scheduling where we remove a job from the front of the list and add a job to the end of the list. For this reason, the next thing that happens in add_to_ready is a check to see if the ready list is empty. If the list is empty then we can add the process right to the front of the list since it is the only job. If the list is not empty then a pointer is set to the beginning of the list and traversed to the end. The job is then added to the end of the ready list to satisfy round robin scheduling.

- **exit_helper(struct proc** sList)** proc.c – lines 1053-1062
  The exit_helper function takes one argument: a pointer to a list. This function is used in the exit function to help search a list for processes whose parent is the currently running process. If the processes parent is the currently running process it sets that processes parent to initproc. It traverses through the entire list looking for processes that meet this condition.
- **wait_helper(struct proc** sList, int * hk)** proc.c – lines 1065-1074
  The wait_helper function takes two arguments: a pointer to a list and a pointer to an integer. This function is used in the wait function to help search a list for processes whose parent is the currently running process. It mimics pass by reference by using pointers and sets the hk variable (havekids) to 1 if it finds a process whose parent is the currently running process. This means the currently running process will get put to sleep in order to wait for its children to finish executing.

In order to avoid concurrency issues, all sections of code that manipulate a list follow the process of acquiring the lock, performing all necessary operations, and then releasing the lock when done. Conditional compilation was used to ensure that all the necessary modifications ran only when the CS333_P3P4 flag is enabled.

- Changes were made to the following existing functions in the **proc.c** file to incorporate the new method of process management:
  - **allocproc()** - The proc pointer, p, was set to the head of the free list (line 69). It is then checked against null in order to decide whether or not the process should be removed from the free list for allocation purposes (line 70). If the process is null execution proceeds by releasing the lock and returning from the allocproc call with a 0. If the process isn't null it is then removed from the free list and an assert_state is performed (lines 71 and 72) to make sure that process' state matches the state list it came from (UNUSED). If the assert passes a goto is used to jump to found. The state of the process is changed to EMBRYO and the process is added to the embryo list using the add_to_list function described above. The lock is then released and execution continues as normal (lines 80-87). This marks a processes transition from the free list to the embryo list during allocation.
  - **Userinit()** - Each of the new lists were initialized to 0 (null) in order to ensure an appropriate starting state for the lists; empty (lines 133-138). A loop through the proc array is then performed in order to allocate NPROC number of empty processes, adding them to the free list (lines 141 and 142). Execution continues as normal by a call to allocproc(), described above, which puts the process on the embryo list. The newly created initial process is then removed from the embryo list, an assert is performed to make sure its state matches EMBRYO, its state is changed to RUNNABLE, and it is set directly to the head of the ready list since it is the first process to be created (lines 166-177).
  - **fork()** - Execution of the fork function is largely the same except for a few minor changes. If copying of the process from state p fails its check, the process is removed from the embryo list, assert_state is performed to make sure the process' state matches EMBRYO where it just came from, it's state is changed to UNUSED, and it is added back to the free

list (lines 219-231). Execution continues as normal until the process has been fully created. It is then removed from the embryo list, an assert is performed to make sure the process' state matches EMBRYO where it just came from, and then the process is passed into the add_to_ready function to ensure that it is added to the ready list according to round robin scheduling conventions since it is not the first process to have been created (lines 254-263).

- **exit()** - a new exit function was written (lines 314-365) to incorporate the new lists that were created. Execution of the new exit function differs from the prior exit function once we hit line 343. The exit_helper function described above is used with the embryo, ready, running, and sleep lists (lines 343-346) to check whether or not they contain any processes whose parent is the currently running process. The zombie list was taken care of in a while loop instead of using the exit_helper function because of the need to wake up the initproc when you find a process whose parent is the currently running process (lines 350-357).
- **wait()** - a new wait function was written (lines 413-466) to incorporate the new lists that were created. It uses the same infinite for loop (line 420) as the prior version of wait and initially sets the havekids value to 0. It then uses the wait_helper function, passing in the havekids flag, to search the embryo, ready, running, and sleep lists for any process whose parent is the currently running process (lines 427-430). If it finds one it sets the havekids flag to 1. The zombie list is handled separately because, in the zombie state, the child process is cleared and sent back to the free list so that it can be used again to create a new process if necessary (lines 434-455). If either the havekids flag hasn't been set to 1 to signify a child process needs to complete execution or the currently running process' killed flag hasn't been set, the lock is released and the wait function is exited with a -1 value to signify these conditions haven't been met (lines 458-461). Otherwise, the currently running process is put to sleep in order to let child processes finish execution (line 464).
- **scheduler()** - a new scheduler was implemented (lines 521-557). This new scheduler follows similar behavior to the old scheduler with modifications to use the new ready list. An infinite for loop is used to ensure that the scheduler is always checking for processes in the ready list to pass to the running list. The idle flag is set to 1 to assume idle (line 530). The ready list is checked for a process and, if there is a process to run, it is removed from the front of the ready list in accordance to round robin scheduling, an assert is performed to make sure that the process is in the RUNNABLE state, the idle flag is set to 0 to signify non-idle, the currently running process is set to that process, and it is added to the running list with the add_to_list function (lines 532-549). When the process is done running the currently running process is set to 0 (null) to signify the running process is done (line 548). The lock is then release (line 551) and idle is checked to see if the sti() and hlt() functions should be run (lines 552-555).
- **yield()** - yield was modified to use the new helper functions to remove the currently running process from the running list, an assert is performed to make sure that the process is in the RUNNING state, the process' state is changed to RUNNABLE, and it is added to the ready list before a call to sched() occurs (lines 606-616).
- **sleep()** - the sleep function was modified to incorporate the new sleep list (lines 644-681). The currently running process is removed from the running list using remove_from_list(), an assert is performed to ensure the process is in the RUNNING state, the process' state is changed to SLEEPING, and the process is added to the sleep list with the add_to_list() function before a call to sched() (lines 664-671).

- **wakeup1()** - a new wakeup1 function was implemented (lines 697-710). The sleep list is checked for a process and, while there are processes on the list, each process is checked against the channel that it was put to sleep on (line 702). If the channels match the process is removed from the sleep list, an assert is performed to make sure the process is in the SLEEPING state, the process' state is changed to RUNNABLE, and the process is added to the end of the ready list using the add_to_ready() function (lines 702-706). This function ensures that all processes sleeping on the channel passed in as an argument are placed on the ready list in the appropriate order.
- **Kill()** - a new kill function was implemented (lines 747-800). The new kill function takes an integer argument that represents the PID of the process that the kill function is searching for. Kill searches through the embryo, ready, and running, and sleep lists for a process matching the PID passed in as an arg. If it finds the process in the embryo, ready, or running list it sets that process' killed flag to 1 and returns 0 to signify that it successfully found and switched that process' killed flag (lines 754-782). If the process is found in the sleep list the process' killed flag is set to 1, the process is removed from the sleep list, an assert is performed to make sure the process' state is in the correct SLEEPING state, the process' state is changed to RUNNABLE, and the process is added to the end of the ready list using the add_to_ready() function (lines 784-796). If the process isn't found on any of the lists a -1 is returned to signify it wasn't found.

2. New console control sequences
   Files Modified:
   defs.h, console.c, proc.c

- Prototypes for all functions involved in the implementation of the new console control sequences were added to the defs.h file (lines 121-124).
- The consoleintr() function was modified to include both an integer flag (lines 193-196) and new switch cases for each of the new console control sequences (lines 217-228). The integer flag is first set to 0 and, if the control sequence is pressed, the flag gets set to 1 in its switch case to signify its corresponding function should be run and a break out of the switch statement occurs. Once out of the switch statement the flag is checked to decide whether or not one of the corresponding functions should be run (lines 246-257).

The following control sequence functions were implemented in the proc.c file in order to have access to the lists that they operate on. Each of the following functions operate by first acquiring the lock, then performing their necessary operations, and then releasing the lock to avoid concurrency issues.

- **free_length(void)** – The free_length function initializes a count variable (line 892) to 0 and traverses through the free list, incrementing the count variable for each process that resides in the free list. Once traversal through the free list has completed the count variable is then displayed to the screen in the manner described in deliverables above (lines 887-904).
- **display_ready(void) –** (lines 907-927) The display_ready function traverses through the ready list and displays the PID of any processes that are in the RUNNABLE state. If the ready list is empty the function displays the message "No processes currently in ready" (line 913). Otherwise, it traverses the list and displays all PIDs in the list in the manner described in deliverables above.
- **display_sleep(void)** - (lines 930-950) The display_sleep function traverses through the sleep list and displays the PID of any processes that are in the sleeping state. If the sleep list is empty

the function displays the message “No processes currently in sleep” (line 940). Otherwise, it traverses the list and displays all PIDs in the list in the manner described in deliverables above.

- **display_zombie(void) –** (lines 953-982) The display_zombie function traverses through the zombie list and displays both the PID and PPID of any processes that are in the zombie list. If the zombie list is empty the function displays the message “No processes currently in zombie” (line 958). Otherwise it traverses the list displaying all PIDs and PPIDs of processes in the list.

Testing

1) Free list initialization test

   In order to test that the free list is being initialized properly I will use the output from the ctrl-p sequence and the fact that, because the free list is initialized based on the number of elements in the ptable.proc array (64), I expect there to be 62 processes on the free list upon booting xv6. This is because the init and sh processes will have been created and will be sitting in the sleep list.

```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ PID    Name    UID     GID     PPID    State   Elapsed  CPU    PCs
2        sh      10      10      1       sleep   3.28     0.01   801052c5 80100aaf 80101fe9 801012af 80106733 80106576 8010788e 80107689
1        init    10      10      1       sleep   3.34     0.02   801052c5 80104f61 80107376 80106576 8010788e 80107689

$
$ Free List Size: 62
```

Figure 1: Free list initialization

   As shown above in Figure 1, when xv6 first boots up and the ctrl-p sequence is pressed we can see that the init and sh processes are sitting in the sleep list. When the new ctrl-f sequence is pressed we can see that there are 62 out of the 64 total processes available in the free list. This is exactly as I expected since free list initialization is fairly straightforward.
   This test PASSES.

2) Free list/Zombie list transition test

   In order to test that the transition to and from the free list and the transition to and from the zombie list functions correctly I created a file named zombietest.c. This file includes code written by Mark Morrissey that forks a bunch of child processes that immediately call exit. The parent is then put to sleep long enough to see them in the zombie list. I will first output the free list size which I expect to be 62. I will then run the zombietest command and, upon its completion, I will use the ctrl-p sequence followed by the new ctrl-z sequence to show that the processes are in the zombie state and match output from ctrl-p. I will also press the ctrl-f sequence to show the change in free list size. Upon fully finishing the zombietest command I will end the test by pressing the ctrl-f sequence again to show that the free list size has gone back to the expected number of processes; 62.

```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ Free List Size: 62
zombietest &
$ Process 9: setting UID and GID to 9
Process 8: setting UID and GID to 8
Process 6: setting UID and GID to 6
Process 7: setting UID and GID to 7
Process 5: setting UID and GID to 5
PID      Name    UID     GID     PPID    State   Elapsed  CPU    PCs
9        zombietest      9       9       4       zombie  6.51     0.09
8        zombietest      8       8       4       zombie  6.53     0.00
7        zombietest      7       7       4       zombie  6.54     0.02
6        zombietest      6       6       4       zombie  6.56     0.05
4        zombietest      10      10      1       sleep   6.63     0.20   801052c5 80107454 80106576 8010788e 80107689
5        zombietest      5       5       4       zombie  6.57     0.03
2        sh      10      10      1       sleep   17.40    0.04   801052c5 80100aaf 80101fe9 801012af 80106733 80106576 8010788e 80107689
1        init    10      10      1       sleep   17.44    0.03   801052c5 80104f61 80107376 80106576 8010788e 80107689
Zombie List Processes(/PPIDs)
(5, 4) -> (7, 4) -> (8, 4) -> (6, 4) -> (9, 4)
Free List Size: 56
zombie!
Free List Size: 62
```

Figure 2: Free list/Zombie list transitions

As we can see in Figure 2 above the free list started off at the expected size of 62. The zombietest command is then run and the ctrl-p sequence followed by the ctrl-z sequence is then pressed. We can see that the processes listed in the ctrl-p sequence that display zombie for their state are also the the processes that appear when the ctrl-z sequence is pressed. We can also see that the number of processes that the ctrl-f sequence shows (56) is accurate since there are 8 processes in total displayed in the ctrl-p sequence output. Once the zombietest command finished and displayed zombie! the ctrl-f sequence displayed that the number of processes on the free list was back to 62 which is what I expected.
Because all behavior and output during this test match what is to be expected the free list/zombie list test PASSES.

3) Kill command test

In order to test the kill command I will use a combination of the zombietest command and the kill command. I will first use the ctrl-p sequence to show the two initial processes. I will then run the zombietest command and press ctrl-p again to show the available processes to kill. I will then pick a process and use the kill command on it. After the kill command I will press the ctrl-p sequence again and use the ctrl-z sequence to show that the process chosen to be killed successfully transitioned state to the zombie list. I expect that the chosen process will successfully end up in the zombie state and the ctrl-z output will show that one process in the zombie list.

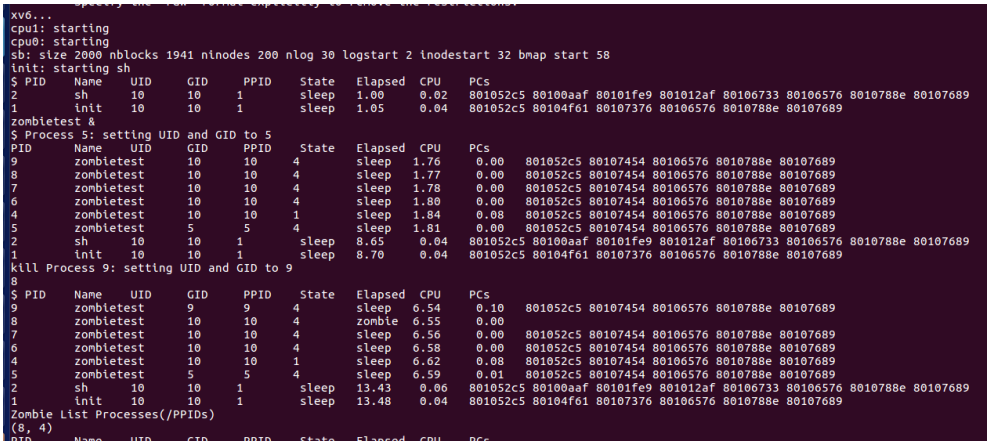


Figure 3: Kill command

As we can see in Figure 3 above the initial two processes get put to sleep and, when the zombietest command is run, ctrl-p shows each of these processes in the sleep state. I chose to kill process 8 and, after the kill command was used and ctrl-p was pressed again, the process successfully transitioned to the zombie state. The ctrl-z sequence correctly displays the killed process and its PPID below the last press of ctrl-p.
The kill command test PASSES.

4) Round Robin test

In order to test that round robin scheduling is being used properly I will use a combination of the loopforever command that was written by Mark Morrissey and the output from the ctrl-r sequence to show that processes are removed from the front of the list and added to the back of the list. I expect that the lists will be changing fairly quickly due to the speed at which state transitions can occur but the output should show round robin scheduling being employed.

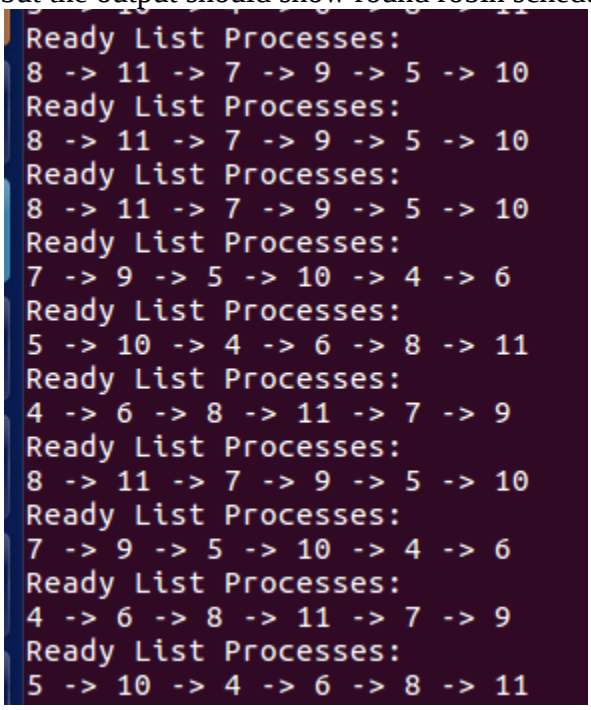


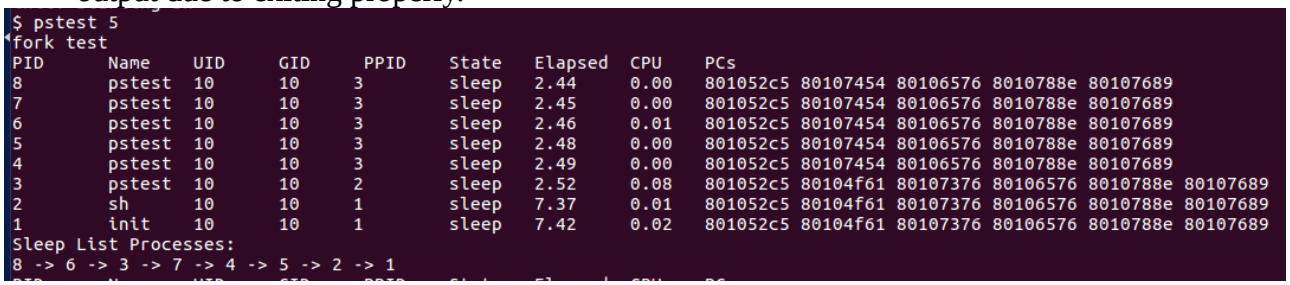
Figure 4: Round Robin Scheduling

As we can see in Figure 4 above, the output from the ctrl-r sequence shows that processes are being successfully removed from the front of the ready list (leftmost side) and added to the back of the ready list (rightmost side). This behavior can be seen as we read from the topmost line to the bottommost line and see that each number is ahead of the number to its left in the list and, when removed from the list, appears after those numbers when added back to the ready list.

Because this behavior is consistent with round robin scheduling the round robin test PASSES.

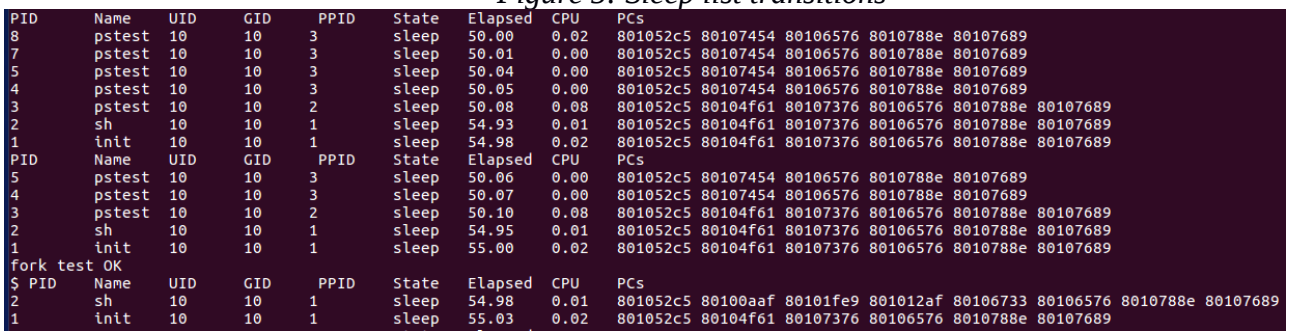
5) Sleep list transition test

In order to test that processes are being transitioned to and from the sleep list properly I will use a combination of the pstest command and ctrl-p output to show that processes are being put to sleep and removed from sleep appropriately. The test works by forking a bunch of child processes and then putting them to sleep for a short duration. I expect that the processes will show up on the sleep list and gradually switch to runnable or running and then disappear from output due to exiting properly.



```
$ pstest 5
fork test
PID      Name    UID     GID     PPID    State   Elapsed  CPU    PCs
8        pstest  10      10      3       sleep   2.44     0.00   801052c5 80107454 80106576 8010788e 80107689
7        pstest  10      10      3       sleep   2.45     0.00   801052c5 80107454 80106576 8010788e 80107689
6        pstest  10      10      3       sleep   2.46     0.01   801052c5 80107454 80106576 8010788e 80107689
5        pstest  10      10      3       sleep   2.48     0.00   801052c5 80107454 80106576 8010788e 80107689
4        pstest  10      10      3       sleep   2.49     0.00   801052c5 80107454 80106576 8010788e 80107689
3        pstest  10      10      2       sleep   2.52     0.08   801052c5 80104f61 80107376 80106576 8010788e 80107689
2        sh      10      10      1       sleep   7.37     0.01   801052c5 80104f61 80107376 80106576 8010788e 80107689
1        init    10      10      1       sleep   7.42     0.02   801052c5 80104f61 80107376 80106576 8010788e 80107689
Sleep List Processes:
8 -> 6 -> 3 -> 7 -> 4 -> 5 -> 2 -> 1
```

Figure 5: Sleep list transitions



```
PID      Name    UID     GID     PPID    State   Elapsed  CPU    PCs
8        pstest  10      10      3       sleep   50.00    0.02   801052c5 80107454 80106576 8010788e 80107689
7        pstest  10      10      3       sleep   50.01    0.00   801052c5 80107454 80106576 8010788e 80107689
5        pstest  10      10      3       sleep   50.04    0.00   801052c5 80107454 80106576 8010788e 80107689
4        pstest  10      10      3       sleep   50.05    0.00   801052c5 80107454 80106576 8010788e 80107689
3        pstest  10      10      2       sleep   50.08    0.08   801052c5 80104f61 80107376 80106576 8010788e 80107689
2        sh      10      10      1       sleep   54.93    0.01   801052c5 80104f61 80107376 80106576 8010788e 80107689
1        init    10      10      1       sleep   54.98    0.02   801052c5 80104f61 80107376 80106576 8010788e 80107689
PID      Name    UID     GID     PPID    State   Elapsed  CPU    PCs
5        pstest  10      10      3       sleep   50.06    0.00   801052c5 80107454 80106576 8010788e 80107689
4        pstest  10      10      3       sleep   50.07    0.00   801052c5 80107454 80106576 8010788e 80107689
3        pstest  10      10      2       sleep   50.10    0.08   801052c5 80104f61 80107376 80106576 8010788e 80107689
2        sh      10      10      1       sleep   54.95    0.01   801052c5 80104f61 80107376 80106576 8010788e 80107689
1        init    10      10      1       sleep   55.00    0.02   801052c5 80104f61 80107376 80106576 8010788e 80107689
fork test OK
$ PID    Name    UID     GID     PPID    State   Elapsed  CPU    PCs
2        sh      10      10      1       sleep   54.98    0.01   801052c5 80100aaf 80101fe9 801012af 80106733 80106576 8010788e 80107689
1        init    10      10      1       sleep   55.03    0.02   801052c5 80104f61 80107376 80106576 8010788e 80107689
```

Figure 6: Sleep list transitions done

As we can see in Figures 5 and 6 above the pstest command forks some child processes and puts them to sleep. The ctrl-s sequence output shows these processes in the sleep list which means that they had to have transitioned from free to embryo to ready to running and finally into sleep to have gotten there. As the fork finishes the number of processes in the ctrl-p sequence output diminishes until just the original two processes, init and sh, remain. In order for the processes to have effectively removed from this output they would have had to have been woken up, transferred through ready and runnable, and into the zombie state to appropriately exit without any errors.

Because the ctrl-s sequence and the ctrl-p sequence output matches closely the sleep list transition test PASSES.

6) Control sequence tests

Due to the above testing and the need to use each of the control sequences successfully to complete those tests I do not feel the need to do separate test cases for each of the control sequences. I would like to argue that each of the above tests used the ctrl-f, ctrl-r, ctrl-s, or ctrl-z sequences appropriately throughout the test and, each time the sequences were used, they showed appropriate behavior and listed the appropriate information when used. They also showed information that matched information reflected via ctrl-p and the ps command. Because of consistently accurate behavior and supporting information across the ctrl-p sequence and the ps command the control sequence testing PASSES.