Project Three Report
Introduction to Operating Systems
Spring 2017

Andy Keene

Description

For this assignment I learned about improving process management by implementing the xv6 state transition diagram using lists (per state) to increase run-time efficiency of state transitions while maintaining the invariant that each process may be on one and only one list at a time. I also learned about adding console ctrl commands, using locks to support atomicity, and about using conditional compilation to both produce two functioning versions of the same operating system and to compile code that demonstrates the list invariant.

Deliverables

The following features were added to xv6:

- xv6 state lists were added to replace the use of the process array `ptable` and increase efficiency. The lists, and transitions between were modeled after the state transition diagram where each list corresponds to a single process state (note that the diagram omits the transitions between ZOMBIE, UNUSED and EMBRYO), with the invariant that *each process is on one and only one list at a time*. The lists and their corresponding state are as follows:

Figure 1: State transition diagram

  - Free list. Each process on the free list is the UNUSED state.
  - Embryo list. Each process on the embryo list is the EMBRYO state.
  - Ready list. Each process on the ready list is in the RUNNABLE state.
  - Sleep list. Each process on the sleep list is in the SLEEPING state.
  - Running list. Each process on the running list is in the RUNNING state.
  - Zombie List. Each process on the zombie list is in the ZOMBIE state.

- New console control sequences were added to display information of the corresponding state lists.
  - ctrl−r now displays the PID of each processes on the ready list, which corresponds to the processes in the RUNNABLE state. The output displays as:

    Ready List Processes:
    $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow n$
  - ctrl−s now displays the PID of each processes on the sleep list, which corresponds to the processes in the SLEEPING state. The output displays as:

    Sleeping List Processes:
    $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow n$

- ctrl−z now displays the PID and PPID of each processes on the zombie list, which corresponds to the processes in the ZOMBIE state. The output displays as:

  Zombie List Processes:
  $(PID_1, PPID_1) \rightarrow (PID_2, PPID_2) \rightarrow (PID_3, PPID_3) \rightarrow \ldots \rightarrow (PID_n, PPID_n)$
- ctrl−f now displays the number processes on the free list, which corresponds to the processes in the UNUSED state. The output displays as:

  Free List Size: n

Implementation

State Lists

Note that all changes regarding state lists and their corresponding transitions were done in proc.c, so all line numbers mentioned in regards to these changes refer to the file proc.c. It should also be noted that all changes for this project use the conditional compilation flag `CS333_P3P4`; often the conditional compilation lines will not included in the line numbers listed for changes. To implement the use of state lists in xv6, the following struct to define the state lists was added to proc.c (lines 16-23):

```
struct StateLists {
  struct proc* ready;
  struct proc* free;
  struct proc* sleep;
  struct proc* zombie;
  struct proc* running;
  struct proc* embryo;
};
```

`struct StateLists pLists` was added to the `ptable` struct (line 30) and a `struct proc * next` component was added to the `proc` structure (proc.h line 74) to support linking the processes together in the lists. Next, the following generic helper functions were added to support state transitions between the lists:

- `removeFromStateList(struct proc** stateList, struct proc* p, enum procstate state)` (lines 194-217). Asserts that a lock is held, that p is in the given state, then removes process p from the state list in $O(|statelist|)$ time.

- `popHeadFromStateList(struct proc** stateList, struct proc** p, enum procstate state)` (lines 227-242). Asserts that a lock is held, that the head is in the given state, then removes and returns the head of the state list in $O(1)$ time.

- `appendToStateList(struct proc** stateList, struct proc* p, enum procstate state)` (lines 267-287). Asserts that a lock is held, adds the given process p to the end of state list in $O(|statelist|)$ time, then changes it's state to the given state.

- `prependToStateList(struct proc** stateList, struct proc* p, enum procstate state)` (lines 293-306). Asserts that the lock is held, then adds the given process p to the head of the state list in $O(1)$ time, and changes its state to the given state.

All state transitions follow the procedure of: obtaining the ptable lock, removing the process from its list through a helper, adding the process to the new list through a helper, then releasing the ptable lock. Each call to the aforementioned helper functions passes the process to be added/deleted or returned through, the state list and it's corresponding state (i.e. UNUSED, RUNNABLE, etc.); and all state changes such as `p->state = UNUSED)` occur within helper functions that add processes to the list, while the state is asserted to be correct in functions that remove processes. All helper functions accessing a list assert that the lock is held and panic if it is not - this eliminates race conditions within the xv6 state lists and supports atomicity of state transitions which in part helps to ensure our list invariant that is maintained. This invariant is further demonstrated to be held by turning on the `DEBUG` flag which calls `checkProcs` (lines 1155-1175) and `findProc` (lines 1112-1140) before a process is removed, and after a process is added. These functions counts how many lists a process is on and panics if it is not exactly one.

Transitions to Free List

The free list is initialized using the existing process table through a call to the helper function `initUnused(void)` (lines 250-261) from `userinit()` (line 400). After this call, all processes are on the free list. All processes are added to, and removed from the free list using `popFromHead` and `prependToStateList` both of which occur in $O(1)$ complexity - thus the free list is managed in $O(1)$ time. Process are *added* to the free list in the following places:

- If the kernel fails to allocate memory for the process in `allocproc()` it is removed from the embryo list and placed on the free list (lines 361-364).
- If copying a processes into p fails in `fork()` it is removed from the embryo list and placed on the free list (lines 477-478).
- In `wait()` a zombie process may transition from the zombie list and be placed on the free list (lines 673-674).

Transitions to Embryo List

Processes are added to the embryo list using `prependToSateList()` and thus occur in $O(1)$ time. Since processes are removed from the free list in $O(1)$ time, transitions from free to embryo occur in $O(1)$ time. Processes are *added* to the embryo list in the following places:

- If `allocproc()` is successful in finding an UNUSED process, it is removed from the free list and added to the embryo list (lines 336, 349).

Ready List

To maintain a FIFO scheduling order, processes are *only* added to the end of ready list using `appendToSateList()` and thus occur in $O(n)$ time. Processes are only removed from the head of ready list when the scheduler runs, which removes in $O(1)$. Processes are *added* to the ready list in the following places:

- In `userinit()` initproc is removed from the embryo list and added to the ready list (lines 430-433).
- In `fork()` if a process is created it is removed from the embryo list and added to the ready list (lines 508-512).
- In `yield()`, which is invoked by the process or through an interrupt, a processes is moved from the running list and added to the ready list (lines 847-848).
- In `wakeup1()` each process sleeping on the channel will be removed from the sleep list and moved to the ready list (lines 946-947)
- In `kill()` the process being killed, *if* it is sleeping, will be removed from the sleep list and moved to the ready list (lines 1004-1005).

Transitions to Running List

Transitions to the running list may only occur within the scheduler, where the scheduler removes the first process on the ready list using `popHeadFromStateList` and adds it to the head of the running list using `prependToStateList`. Since both of these list manipulations occur in $O(1)$ time, the transition to the running list occurs in $O(1)$ time. Also, since processes are only added to the end of the running list and removed from the front, a FIFO ordering is maintained.

- If the scheduler is successful in finding a process at the head of the ready list, it is removed and added to the front of the running list (lines 764-765).

Transitions to Sleep List

Processes are added to the sleep list using `prependToSateList()` and thus occur in $O(1)$ time, though they are removed in $O(n)$ time. Processes are added to the sleep list in the following places:

- In `sleep()` the process calling is running so it is removed from the running list and added to the sleep list (lines 902-903).

Transitions to Zombie List

Processes are added to the zombie list using `prependToSateList()` and thus occur in $O(1)$ time, though they are removed in $O(n)$ time. Processes are added to the zombie list in the following places:

- In `exit()` the process exiting is running so it is removed from the running list and added to the zombie list (lines 597-598).

Other Process Management Optimizations

To optimize process management various dependencies on the process table, `ptable`, were removed. Specifically this occurred in `wait`, `kill`, `wakeup1`, and `exit`.

- `wait` (lines 651-693) now uses a method `hasChildren(struct proc * p)` (lines 168-186) to search the embryo, ready, sleep, and running lists for children of the parent - it is called on line 661. `hasChildren` uses `findChild(struct proc* stateList, struct proc* parent)` (lines 150-161) to identify whether a list contains a child of the process. The search is stopped as soon as a child is found. The number of processes inspected is reduced by the size of the free list (i.e. $O(NPROC - |free|)$). The zombie list is treated differently since if a child is found it is reaped by removing it from the zombie list and adding it to the free list.

- `kill` (lines 990-1011) now uses the helper function `findProcess(struct proc** p, int pid)` (lines 117) to obtain the process with the given PID. `findProcess` just calls `getProcess(struct proc* stateList, struct proc** p, int pid)` (lines 95-110) on the embryo, ready, running, sleep, and zombie lists. `kill` obtains the process through the pointer argument and proceeds to handle killing it appropriately. Since `kill` no longer uses the proc table, and instead uses the lists its efficiency has increased to $O(NPROC - |free|)$.

- `wakeup1` (lines 935-952) now *only* uses the sleep list to search for processes to wake up and transition them from the sleep list to the runnable list. The search is done in-line. This efficiency is now $O(|sleep|)$ where $O(|sleep|) \leq O(NPROC|)$ (but commonly much, much less).

- `exit()` (lines 563-693) now abandons it's children by calling `abandonChildren` (lines 71-88) for the embryo, ready, running, sleep and zombie lists. `abandonChildren(struct proc* stateList, struct proc* parent)` searches through the list given and abandons all the parents (calling process) children, by changing their PPID, to `initproc`. If the child is in the ZOMBIE state, `initproc` is woken. There is a slight efficiency performance in this modification as `exit` no longer has to look through the entire process table - the number of processes inspected is now reduced by the size of the free list $NPROC - |free|$.

  *$|free|$ in $O(NPROC - |free|)$ refers to the size of free at the point the function obtains the lock, commonly this is a non-zero value. The only places where the process table array is still used, as allowed by the project description, is in `userinit()`, `procdump()`, and `getprocs()` - all other references have been replaced with the use of process lists.

New Console Commands

To support new console commands, changes were made to the files: console.c to trigger the appropriate printing in response to the ctrl−char input; defs.h to define the prototype for the kernel side function that prints the appropriate information; and to proc.c to define the function for displaying the list information. All functions such as `dofreelistinfo()` that display list information obtain the ptable lock before traversing the appropriate list and release it upon completion.

- ctrl−r
  - Function prototype was added to defs.h (line 124)
  - In console.c a flag was initialized (line 194), logic for the case of input ’R’ was added (lines 219-221), and the logic for triggering the kernel side function to display information was added (lines 250-252)
  - In proc.c the function `readylistinfo()` was added (lines 1180-1195) to display the PID of each process on the ready list as:

    Ready List Processes:
    $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow n$
- ctrl−s
  - Function prototype was added to defs.h (line 125)
  - In console.c a flag was initialized (line 195), logic for the case of input ’S’ was added (lines 225-227), and the logic for triggering the kernel side function to display information was added (lines 256-258)
  - In proc.c the function `sleepinglistinfo()` was added (lines 1217-1233) to display the PID of each process on the sleep list as:

    Sleeping List Processes:
    $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow n$
- ctrl−z
  - Function prototype was added to defs.h (line 126)
  - In console.c a flag was initialized (line 195), logic for the case of input ’Z’ was added (lines 228-231), and the logic for triggering the kernel side function to display information was added (lines 259-261)
  - In proc.c the function `zombielistinfo()` was added (lines 1237-1255) to display the PID and PPID of each process on the zombie list as:

    Zombie List Processes:
    $(PID_1, PPID_1) \rightarrow (PID_2, PPID_2) \rightarrow (PID_3, PPID_3) \rightarrow \ldots \rightarrow (PID_n, PPID_n)$
- ctrl−f
  - Function prototype was added to defs.h (line 123)
  - In console.c a flag was initialized (line 194), logic for the case of input ’F’ was added (lines 222-224), and the logic for triggering the kernel side function to display information was added (lines 251-255)

- In proc.c the function `freelistinfo()` was added (lines 1200-1213) to display the number of process on the free list as:

  Free List Size: n

Testing

Required Tests

For reference the required tests are outlined below.

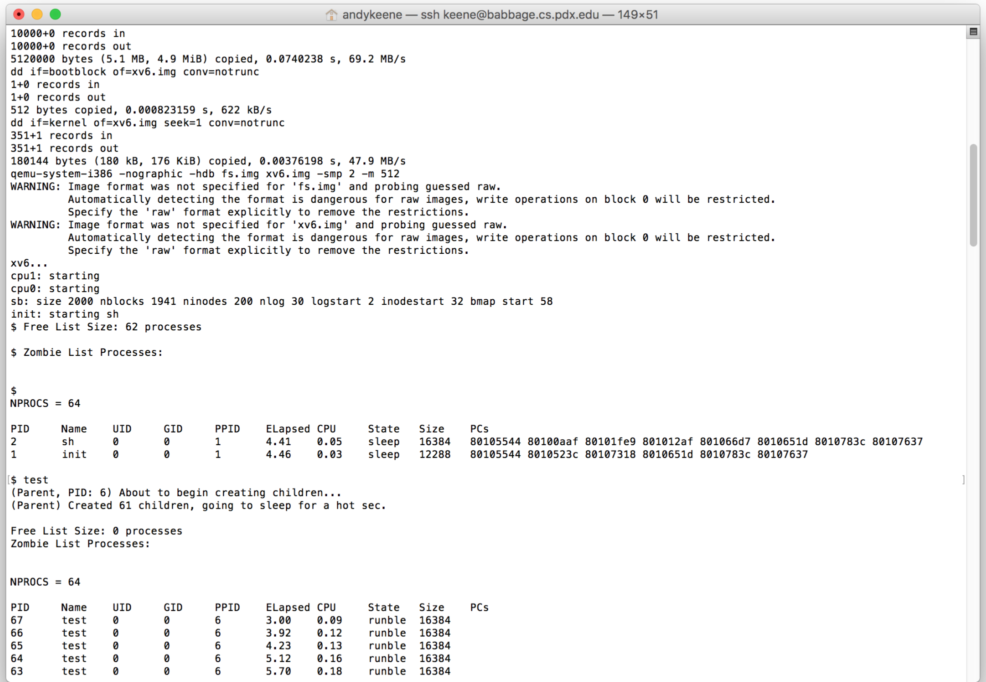
1. Demonstrate that the free list is correctly initialized when xv6 is booted. Note that `init` and `sh` should be the only two active processes immediately after boot while the rest are unused. Recall that the `NPROC` variable represents the maximum number of processes in xv6.
2. Demonstrate that the free list is correctly updated when a new process is allocated (state transitions from `UNUSED`) and when a process is deallocated (state transitions to `UNUSED`).
3. Demonstrate the `kill` shell command causes a process to correctly transition to the `ZOMBIE` and then `UNUSED` states.
4. Demonstrate that round-robin scheduling is enforced. Specifically, the processes that are already in the ready list are scheduled before processes added afterwards; any process transitioning to the `RUNNING` state are removed from the front of the ready list. Processes transitioning to the `RUNNABLE` state must be added at the back of the ready list.
5. Demonstrate that the sleep list is correctly updated when a process sleeps (state transitions to `SLEEPING`) and when processes are woken (state transitions from `SLEEPING`).
6. Demonstrate that the zombie list is correctly updated when a process exits (state transitions to `ZOMBIE`) and when a process is reaped (state transitions from `ZOMBIE`).
7. Demonstrate that output for the console commands ctrl−r, ctrl−f, ctrl−s and ctrl−z is correct (7.a, 7.b, 7.c, and 7.d respectively).

Free, ctrl−f and Zombie, ctrl−z (Requirements 1, 2, 6, 7.b, 7.d)

This test will demonstrate: that the free list is correctly updated when a process is allocated (transitions from UNUSED) and when a process is deallocated (transitions to UNUSED); and that the Zombie list is updated correctly when a process exits (transitions from RUNNING to ZOMBIE) and when a process is reaped (transitions from ZOMBIE to UNUSED). ctrl−f and ctrl−z will also be shown to be correct. The function, `free_zombie_tests` will be added in the user program `test` (test.c lines 107-169).

- The user program `test` will pause at start, allowing us to press ctrl−r, ctrl−z and ctrl−p to establish a baseline for the state lists at the start of xv6. Since the `#DEBUG` flag is set while testing, ctrl−p will display the value of NPROC, which represents the maximum number of processes available in the system (added to proc.c lines 1048-1050). We expect that since NPROC represents the number of available processes, and ctrl−p will display all processes *not* on the UNUSED list by virtue of the code, call these *active processes*, we know that the number of UNUSED processes will be $|NPROCS| - \textit{active processes}$. Thus, we expect the output of ctrl−f to match this number.

- Next, `test` will fork children and save their PIDs until `fork()` returns -1 signifying a process allocation failed. `test` will then sleep, allowing us to press ctrl−f ctrl−p and ctrl−z. Note, all children will spin-run. Since `fork()` may fail when there are no more processes on the UNUSED list to allocate, we expect the UNUSED list to be empty. We expect ctrl−f will now display there are 0 processes on the UNUSED list, ctrl−z will display nothing since all children are spinning and not exiting, and ctrl−p will now show an *additional* number of active processes (in the running or runnable state) equal to that of the initial ctrl−f output. The total number of processes shown by ctrl−p should be equal to the value of NPROC, and the parent should display a number of children equal to one less than the initial number of free processes (it had to be created itself!). This demonstrates that process allocation, from UNUSED occurs correctly, that the free list was correctly initialized, and that the output of ctrl−f is correct.
- The parent process `test` will now, in reverse order of creation, kill then reap a child process. We will be notified before, and after, allowing us to press ctrl−f and ctrl−z at each step. Before any process is killed, we expect the Zombie list to be empty and the Free List output size to be that of its last output. (initially this should be 0). Since `kill()` sends a process to the Zombie state (if it is not already there), when we press ctrl−z and ctrl−f after a process is killed, but before it's reaped, we expect ctrl−z to contain *only* the process killed and ctrl−f to not have changed, where the PID of the ZOMBIE process matched the PID of the child being killed, and its PPID matches the parent. After the process is reaped we expect the Zombie list to again be empty and the free list to have increased by 1. With comparison to ctrl−p this demonstrates the that transitions to and from the ZOMBIE state are correct, that transitions to the UNUSED state are correct.
- We will continue this process until the reaping completes. After all children processes have been reaped and the parent exits we expect that ctrl−p will match its original process listing, that ctrl−f will display it's initial value, and that ctrl−z will be empty! Output from ctrl−z and ctrl−f in matching semantic output from ctrl−p in all stages demonstrate their correctness, thus showing requirements 7.b and 7.d are met.



```
andykeene — ssh keene@babbage.cs.pdx.edu — 149×51
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0740238 s, 69.2 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000823159 s, 622 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
351+1 records in
351+1 records out
180144 bytes (180 kB, 176 KiB) copied, 0.00376198 s, 47.9 MB/s
qemu-system-i386 -nographic -hdb fs.img xv6.img -smp 2 -m 512
WARNING: Image format was not specified for 'fs.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
WARNING: Image format was not specified for 'xv6.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.
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 processes

$ Zombie List Processes:


$
NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
2       sh      0       0       1       4.41    0.05    sleep   16384   80105544 80100aaf 80101fe9 801012af 801066d7 8010651d 8010783c 80107637
1       init    0       0       1       4.46    0.03    sleep   12288   80105544 8010523c 80107318 8010651d 8010783c 80107637

$ test
(Parent, PID: 6) About to begin creating children...
(Parent) Created 61 children, going to sleep for a hot sec.

Free List Size: 0 processes
Zombie List Processes:


NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
67      test    0       0       6       3.00    0.09    runble  16384
66      test    0       0       6       3.92    0.12    runble  16384
65      test    0       0       6       4.23    0.13    runble  16384
64      test    0       0       6       5.12    0.16    runble  16384
63      test    0       0       6       5.70    0.18    runble  16384
```

Figure 2: Initial list output and after forking

Here we see that ctrl−p displays the two processes, `init` and `sh` along with the value of NPROC being 64. Since NPROC is 64 and there are 2 *active processes*, based on our expectations the output from the free list should be $64 - 2 = 62$. The free list output matches this. Additionally, ctrl−z is empty which matches ctrl−p at this point. This meets our expectations for this stage of the test, thus this step **PASSES**.

PID	Name	UID	GID	PPID	ELapsed	CPU	State	Size	PCs
67	test	0	0	6	3.00	0.09	runble	16384	
66	test	0	0	6	3.92	0.12	runble	16384	
65	test	0	0	6	4.23	0.13	runble	16384	
64	test	0	0	6	5.12	0.16	runble	16384	
63	test	0	0	6	5.70	0.18	runble	16384	
62	test	0	0	6	6.26	0.20	runble	16384	
61	test	0	0	6	7.11	0.23	runble	16384	
60	test	0	0	6	7.39	0.24	runble	16384	
59	test	0	0	6	7.93	0.26	runble	16384	
58	test	0	0	6	8.20	0.27	run	16384	
57	test	0	0	6	8.98	0.30	runble	16384	
56	test	0	0	6	9.23	0.31	runble	16384	
55	test	0	0	6	9.48	0.32	runble	16384	
54	test	0	0	6	10.22	0.35	run	16384	
53	test	0	0	6	10.94	0.39	runble	16384	
52	test	0	0	6	11.41	0.42	runble	16384	
51	test	0	0	6	12.10	0.43	runble	16384	
50	test	0	0	6	12.55	0.46	runble	16384	
49	test	0	0	6	14.09	0.53	runble	16384	
48	test	0	0	6	14.52	0.55	runble	16384	
47	test	0	0	6	15.15	0.58	runble	16384	
46	test	0	0	6	15.36	0.59	runble	16384	
45	test	0	0	6	15.96	0.62	runble	16384	
44	test	0	0	6	16.16	0.64	runble	16384	
43	test	0	0	6	16.73	0.66	runble	16384	
42	test	0	0	6	17.10	0.68	runble	16384	
41	test	0	0	6	17.83	0.72	runble	16384	
40	test	0	0	6	18.00	0.73	runble	16384	
39	test	0	0	6	18.34	0.75	runble	16384	
38	test	0	0	6	18.67	0.77	runble	16384	
37	test	0	0	6	19.16	0.80	runble	16384	
36	test	0	0	6	19.62	0.83	runble	16384	
35	test	0	0	6	20.08	0.86	runble	16384	
34	test	0	0	6	20.37	0.88	runble	16384	
33	test	0	0	6	20.64	0.91	runble	16384	
32	test	0	0	6	20.78	0.89	runble	16384	
31	test	0	0	6	20.91	0.92	runble	16384	
30	test	0	0	6	21.29	0.95	runble	16384	
29	test	0	0	6	21.66	0.97	runble	16384	
28	test	0	0	6	21.89	0.99	runble	16384	
27	test	0	0	6	22.00	1.01	runble	16384	
26	test	0	0	6	22.21	1.02	runble	16384	
25	test	0	0	6	22.41	1.05	runble	16384	
24	test	0	0	6	22.50	1.06	runble	16384	
23	test	0	0	6	22.69	1.08	runble	16384	
22	test	0	0	6	22.86	1.10	runble	16384	
21	test	0	0	6	23.02	1.12	runble	16384	
20	test	0	0	6	23.17	1.15	runble	16384	
19	test	0	0	6	23.31	1.16	runble	16384	

Figure 3: ctrl-p output after forking

In figure 2, we see that parent created 61 children and has a PID of 6. In figures 3 and 4, using the output of ctrl−p, we see that the parent did in fact create 61 children (I'll let you count this one yourself); thus the number of process displayed by ctrl−p increased by the number of children, plus the parent, processes created. In total there are now 64 active processes, which means since NPROC is 64, that there are no free processes left. ctrl−f is also seen in figure 2 displaying 0 processes on its list. This meets our expectations for this stage of the test, thus this step **PASSES**.

andykeene — ssh keene@babbage.cs.pdx.edu — 149×53

```
53      test    0       0       6       10.94   0.39    runble  16384
52      test    0       0       6       11.41   0.42    runble  16384
51      test    0       0       6       12.10   0.43    runble  16384
50      test    0       0       6       12.55   0.46    runble  16384
49      test    0       0       6       14.09   0.53    runble  16384
48      test    0       0       6       14.52   0.55    runble  16384
47      test    0       0       6       15.15   0.58    runble  16384
46      test    0       0       6       15.36   0.59    runble  16384
45      test    0       0       6       15.96   0.62    runble  16384
44      test    0       0       6       16.16   0.64    runble  16384
43      test    0       0       6       16.73   0.66    runble  16384
42      test    0       0       6       17.10   0.68    runble  16384
41      test    0       0       6       17.83   0.72    runble  16384
40      test    0       0       6       18.00   0.73    runble  16384
39      test    0       0       6       18.34   0.75    runble  16384
38      test    0       0       6       18.67   0.77    runble  16384
37      test    0       0       6       19.16   0.80    runble  16384
36      test    0       0       6       19.62   0.83    runble  16384
35      test    0       0       6       20.08   0.86    runble  16384
34      test    0       0       6       20.37   0.88    runble  16384
33      test    0       0       6       20.64   0.91    runble  16384
32      test    0       0       6       20.78   0.89    runble  16384
31      test    0       0       6       20.91   0.92    runble  16384
30      test    0       0       6       21.29   0.95    runble  16384
29      test    0       0       6       21.66   0.97    runble  16384
28      test    0       0       6       21.89   0.99    runble  16384
27      test    0       0       6       22.00   1.01    runble  16384
26      test    0       0       6       22.21   1.02    runble  16384
25      test    0       0       6       22.41   1.05    runble  16384
24      test    0       0       6       22.50   1.06    runble  16384
23      test    0       0       6       22.69   1.08    runble  16384
22      test    0       0       6       22.86   1.10    runble  16384
21      test    0       0       6       23.02   1.12    runble  16384
20      test    0       0       6       23.17   1.15    runble  16384
19      test    0       0       6       23.31   1.16    runble  16384
18      test    0       0       6       23.51   1.19    runble  16384
17      test    0       0       6       23.63   1.21    runble  16384
16      test    0       0       6       23.68   1.21    runble  16384
15      test    0       0       6       23.78   1.23    runble  16384
14      test    0       0       6       23.87   1.26    runble  16384
13      test    0       0       6       23.91   1.27    runble  16384
12      test    0       0       6       24.05   1.32    runble  16384
11      test    0       0       6       24.14   1.34    runble  16384
10      test    0       0       6       24.17   1.35    runble  16384
9       test    0       0       6       24.19   1.36    runble  16384
8       test    0       0       6       24.23   1.40    runble  16384
7       test    0       0       6       24.24   1.41    runble  16384
6       test    0       0       2       26.28   1.35    sleep   16384   80105544 801073f6 8010651d 8010783c 80107637
2       sh      0       0       1       39.20   0.06    sleep   16384   80105544 8010523c 80107318 8010651d 8010783c 80107637
1       init    0       0       1       39.25   0.03    sleep   12288   80105544 8010523c 80107318 8010651d 8010783c 80107637

Marked [60] child with PID of 67 for death
Free List Size: 0 processes
```

Figure 4: more of ctrl-p output after forking

Figure 4 shows the beginning of the reapings; we see that before that any child has been reaped that the size of the free list is still 0, that after a child has been marked it appears on the zombie list, *and* that after the child is reaped (but before another one is killed) the zombie list becomes empty and the free list has increased by 1; this is also confirmed by the matching PID and PPID of the child and parent. Figure 5 shows more of this procedure. This meets our expectations for this stage of the test, thus this step **PASSES**.

```
Marked [60] child with PID of 67 for death
Free List Size: 0 processes
Zombie List Processes:
(67, 6)
Reaped [60] child with PID of 67
Free List Size: 1 processes
Zombie List Processes:

Marked [59] child with PID of 66 for death
Free List Size: 1 processes
Zombie List Processes:
(66, 6)
Reaped [59] child with PID of 66
Free List Size: 2 processes
Zombie List Processes:

Marked [58] child with PID of 65 for death
Free List Size: 2 processes
Zombie List Processes:
(65, 6)
Reaped [58] child with PID of 65
Free List Size: 3 processes
Zombie List Processes:

Marked [57] child with PID of 64 for death
Free List Size: 3 processes
Zombie List Processes:
(64, 6)
Reaped [57] child with PID of 64
Free List Size: 4 processes
Zombie List Processes:

Marked [56] child with PID of 63 for death
```

Figure 5: Reaping children

```
(Parent) Reaping complete...
$ $ $
NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
2       sh      0       0       1       2185.86 0.07    sleep   16384   80105544 80100aaf 80101fe9 801012af 801066d7 8010651d 8010783c 80107637
1       init    0       0       1       2185.91 0.03    sleep   12288   80105544 8010523c 80107318 8010651d 8010783c 80107637
Free List Size: 62 processes
Zombie List Processes:

$
```

Figure 6: After parent exits

For the final stage of this test, as seen in figure 6, the free list has returned to its initial size (again matching $|NPROCS| - active\ processes$) and that the zombie list is empty (matching ctrl−p). This meets our expectations for this stage of the test, thus this step **PASSES**.

Since at each step, or stage, of this step, this test **PASSES**; and further, because this test demonstrates requirements 1, 2, 6, 7.b, and 7.d we can conclude requirements 1, 2, 6, 7.b, and 7.d are met.

Shell Kill Command (Requirement 3)

For this test we will demonstrate requirement 3, that the shell `kill` command correctly causes a process to transition to the ZOMBIE and UNUSED states accordingly. To this we will create an additional function `inf_loops()` in the user `test` program (test.c lines 153-165) that will fork 5 children where all of whom, including the parent, will spin indefinitely. The steps, and corresponding expectations of this test are laid out as follows:

- Before beginning we will verify the statuses of each list with ctrl−p, ctrl−z, and ctrl−f respectively. ctrl−f and ctrl−z were demonstrated to be correct in the test *Free, ctrl−f and Zombie, ctrl−z (Requirements 1, 2, 6, 7.b, 7.d).*
- We will invoke the `test` program using the command `$ test &` which will allow us to regain shell access in order to kill the created processes. We expect that the parent will display its PID.
- Next, we will press ctrl−p, ctrl−z, and ctrl−f to see that the free list decreased by 6, that there are no zombie sinces all created processes are spinning, and that 6 additional processes named *test* now appear in the runnable/running state on ctrl−p.

- One by one we will kill the processes in descending order toward the parent. We must kill the parent last to enforce that the children are not automatically inherited by `initproc`. We will press ctrl−z before and after killing each process. We expect the zombie list to grow by one between each `kill` until the parent process is killed, and that format of the zombie list `(PID, PPID)` will match the child just killed, and it's parent PID which was printed previously.

- Before killing the parent we will press ctrl−f and ctrl−p. Since the parent is still running, the children have not yet been reaped so we expect to see all 5 children on the zombie list (in reverse order of death), in the ZOMBIE state on ctrl−p, and that the free list *has not* increased in size.

- We will finish by killing the parent. After killing the parent, we expect to see a printed line of "zombie!" for each child we killed, and the following output of ctrl−z to be empty; we also expect that after killing the parent ctrl−p and ctrl−f will show that all the processes returned to the UNUSED list (implicitly by an increase in size of the free list and not shown as an *active process* on ctrl−p).

```
init: starting sh
$
NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
2       sh      0       0       1       1.60    0.02    sleep   16384   8010559f 80100aaf 80101fe9 801012af 801067b2 801065f8 80107917 80107712
1       init    0       0       1       1.66    0.02    sleep   12288   8010559f 80105297 801073f3 801065f8 80107917 80107712
Free List Size: 62 processes

$ Zombie List Processes:


$ test &
$ Parent of Infiite Loops: 6

$
NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
11      test    0       0       6       5.38    1.36    run     16384
10      test    0       0       6       5.59    1.38    runble  16384
9       test    0       0       6       5.75    1.60    runble  16384
8       test    0       0       6       5.79    1.41    runble  16384
6       test    0       0       1       5.88    1.74    runble  16384
7       test    0       0       6       5.83    1.56    runble  16384
2       sh      0       0       1       19.10   0.21    sleep   16384   8010559f 80100aaf 80101fe9 801012af 801067b2 801065f8 80107917 80107712
1       init    0       0       1       19.16   0.02    sleep   12288   8010559f 80105297 801073f3 801065f8 80107917 80107712

$ Zombie List Processes:


$ Free List Size: 56 processes
```

Figure 7: Before and after spin start

Here we see there are 62 free processes, that the parent has a PID of 6, and that after the spin tests are started through invoking `test` that the free list decreased by 6, the correct amount, that the zombie list is empty and that all children and the parent are in the runnable/running state. This meets our expectations for the first two stages.

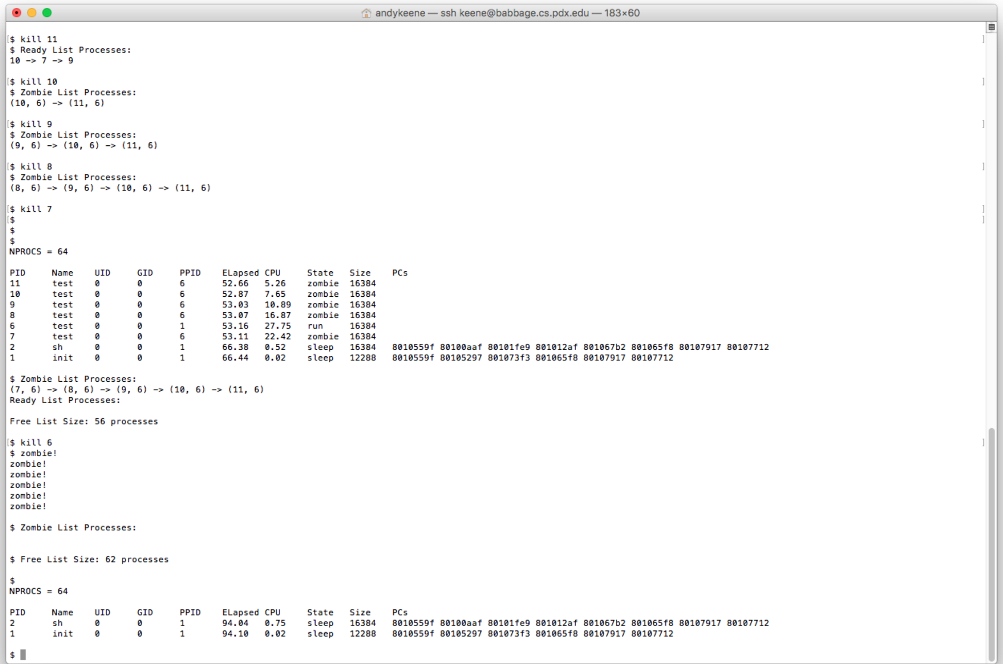


Figure 8: Killing the children and parent

Figure 2 shows that: the zombie list increasing by 1 with each child killed; that the order (in reverse order of being killed); before the parent is killed we see all of the children on the zombie list and all of them in the ZOMBIE state on the ctrl−p output; and finally, that when the parent is killed the free list increases to its original size, the zombie list, and the ZOMBIE states in ctrl−p are all cleared.

Because our expectations for each stage of this test were met, this test **PASSES**.

Sleep, ctrl-s (Requirements 5, 7.c)

This test will demonstrate that both the sleep list and ctrl-s function correctly:

- Before executing the user program `test` we will will press ctrl-p and ctrl-s. This will be our baseline for the currently active processes before the test begins. Since the initial process is always sleeping on boot, and `sh` is waiting for input, we expect to find these on the list.
- Next the user program test will be called. Here, the function `sleep_test()` (test.c lines 145-151) will be called. The `test` process will print its PID and then go to sleep for 5 seconds. Since the process must be running to print a message, we know it entered the SLEEPING state from RUNNING. After the message, we will press ctrl−p and ctrl−s again, expecting to find the process PID now in the SLEEPING sate, and on the sleeping list in addition to `init` and `sh` (`sh` is still waiting for our user program to exit). Note that we may have to press ctrl−s a few times since `test` may be woken on interrupts, but put back to sleep because it is not time to wake up yet - inherent to the kernel.
- The process will notify us it is awake and exiting. We now expect it to be off of the sleeping list, and no longer in the output of ctrl−p since it exited; thus we expect ctrl−p and ctrl−s to return to their initial outputs.

As we can see in figure 1: the initial output of ctrl−p and ctrl−s initially match with the expected processes on the list; next we can see that when `test` goes to sleep that we did have to press ctrl−s multiple times before finding it on the sleeping list but that the ctrl−p output asserts all processes on the list are

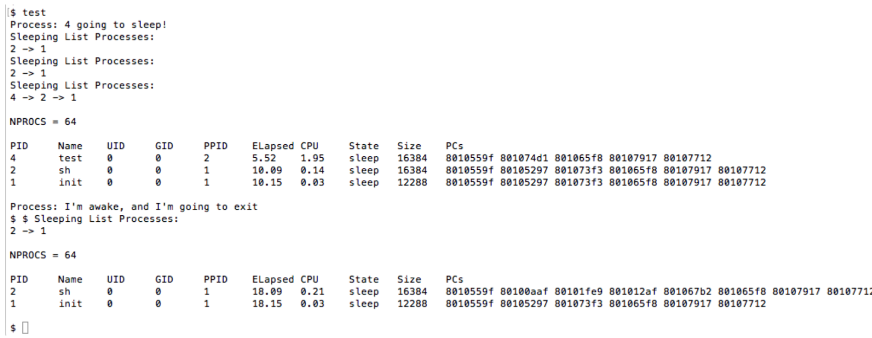


Figure 9: Sleep

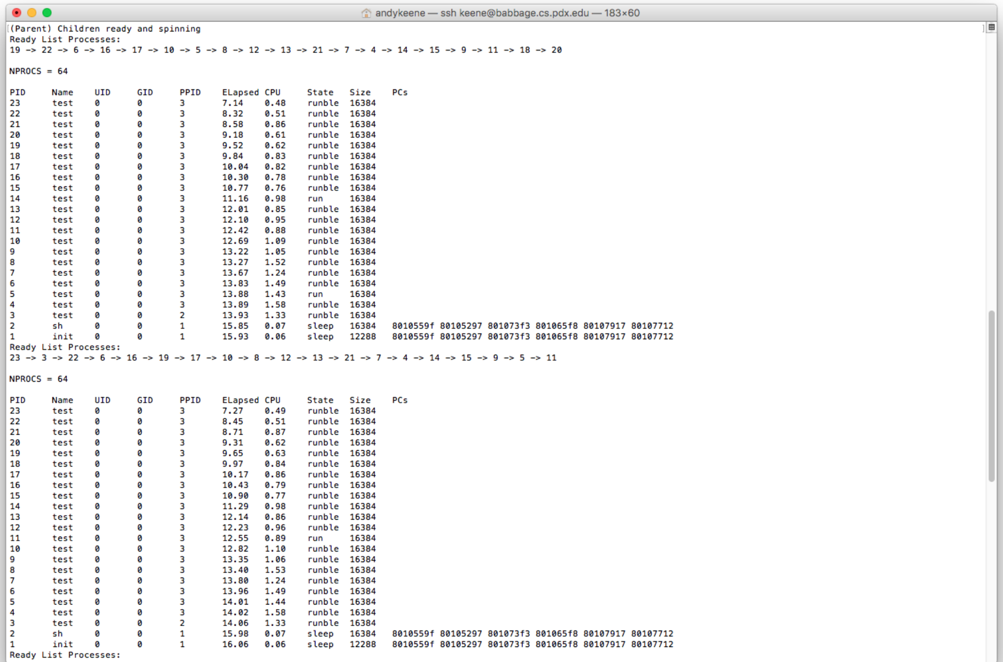
in the correct state; and after `test` exits, the sleeping list returns to it's initial state (`sh` is waiting on input) where `test` is no longer to be found. Since all of our expectations were met, we have demonstrated that processes make the correct transitions from and to the SLEEPING state/list. Additionally, ctrl−p matches ctrl−s at each stage, demonstrating requirement 7.c. Thus, this test **PASSES**.

Round robin, ctrl-r (Requirements 4, 7.a)

This test will demonstrate that processes scheduling enforces round robin, and that ctrl−r is working correctly. A function `round_robin()` will be added to the user program `test` (text.c lines 172-184). This process will create 20 children who will spin, along with the parent after the children's creation (21 total). We will then:

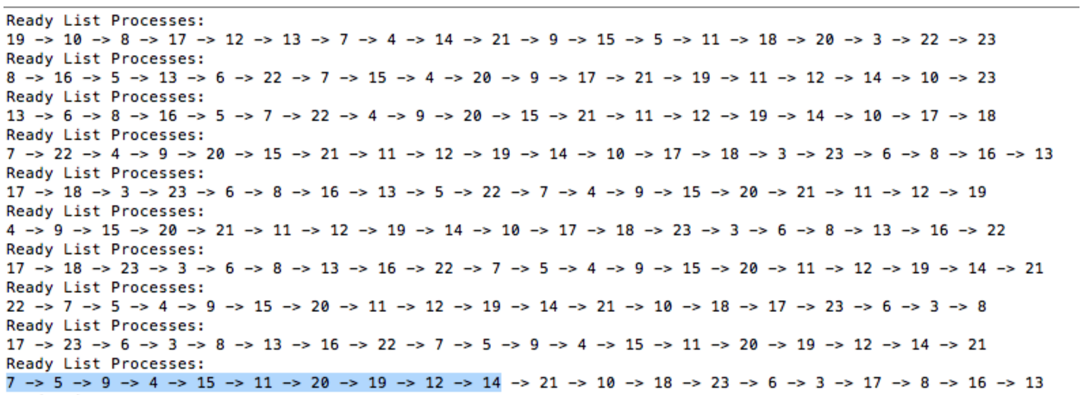
- We will press ctrl−p and ctrl−r repeatedly to establish that ctrl−r does in fact display the ready list. Since two processes may be running at a time (two CPUs) and because context switches are far faster than we, we must acknowledge a tolerant difference of *at most* two processes between the outputs. The best way to assert this is that there must be at least 19 processes on the runnable list, and the output of ctrl−p must have at least 19 `test` processes in the RUNNABLE state, with the remaining in the RUNNING state (per list invariant they must be on one and only one).
- Next we will hold down ctrl−r. Again, with two CPUs, only 2 processes may be running at a time. Thus, from the 21 processes (children and parent), only two may be missing so we expect to see at least 19 on each output. Additionally, we expect that since round robin is maintained that unless the scheduler has made it through the entire list, if some line is A, D, E, J, H, K, P, Q, W and the next line starts with a K, then K must at least be followed by P, Q, W (since K didn't run P, Q and W must not have either, thus the ordering remains) and P, Q, W must be followed by A, D, E and the two processes that *were* running, in some order since these ran, and must be inserted into the back of the list.

Asides: Some disorder in the rear of the list may occur between prints on ctrl−r. With two CPUs it may occur where one beats the other in placing a process back on the runnable list even though it was removed after the other. Thus disorder may introduce itself by virtue of insertion order between the CPUs. This is why we are concerned with looked at the order of the middle of the list as outlined in the second stage of the test. It is also possible that ctrl-p or ctrl-r may obtain the lock in-between `sched` and `scheduler()` so that *more* than 19 processes may be found in the RUNNABLE state which is why we focus on *at least* 19.



```
(Parent) Children ready and spinning
Ready List Processes:
19 -> 22 -> 6 -> 16 -> 17 -> 10 -> 5 -> 8 -> 12 -> 13 -> 21 -> 7 -> 4 -> 14 -> 15 -> 9 -> 11 -> 18 -> 20

NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
23      test    0       0       3       7.14    0.48    runble  16384
22      test    0       0       3       8.32    0.51    runble  16384
21      test    0       0       3       8.58    0.86    runble  16384
20      test    0       0       3       9.18    0.61    runble  16384
19      test    0       0       3       9.52    0.62    runble  16384
18      test    0       0       3       9.84    0.83    runble  16384
17      test    0       0       3       10.04   0.82    runble  16384
16      test    0       0       3       10.30   0.78    runble  16384
15      test    0       0       3       10.77   0.76    runble  16384
14      test    0       0       3       11.16   0.98    run     16384
13      test    0       0       3       12.01   0.85    runble  16384
12      test    0       0       3       12.10   0.95    runble  16384
11      test    0       0       3       12.42   0.88    runble  16384
10      test    0       0       3       12.69   1.09    runble  16384
9       test    0       0       3       13.22   1.05    runble  16384
8       test    0       0       3       13.27   1.52    runble  16384
7       test    0       0       3       13.67   1.24    runble  16384
6       test    0       0       3       13.83   1.49    runble  16384
5       test    0       0       3       13.88   1.43    run     16384
4       test    0       0       3       13.89   1.58    runble  16384
3       test    0       0       2       13.93   1.33    runble  16384
2       sh      0       0       1       15.85   0.07    sleep   16384   8010559f 80105297 801073f3 801065f8 80107917 80107712
1       init    0       0       1       15.93   0.06    sleep   12288   8010559f 80105297 801073f3 801065f8 80107917 80107712
Ready List Processes:
23 -> 3 -> 22 -> 6 -> 16 -> 19 -> 17 -> 10 -> 8 -> 12 -> 13 -> 21 -> 7 -> 4 -> 14 -> 15 -> 9 -> 5 -> 11

NPROCS = 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
23      test    0       0       3       7.27    0.49    runble  16384
22      test    0       0       3       8.45    0.51    runble  16384
21      test    0       0       3       8.71    0.87    runble  16384
20      test    0       0       3       9.31    0.62    runble  16384
19      test    0       0       3       9.65    0.63    runble  16384
18      test    0       0       3       9.97    0.84    runble  16384
17      test    0       0       3       10.17   0.86    runble  16384
16      test    0       0       3       10.43   0.79    runble  16384
15      test    0       0       3       10.90   0.77    runble  16384
14      test    0       0       3       11.29   0.98    runble  16384
13      test    0       0       3       12.14   0.86    runble  16384
12      test    0       0       3       12.23   0.96    runble  16384
11      test    0       0       3       12.55   0.89    run     16384
10      test    0       0       3       12.82   1.10    runble  16384
9       test    0       0       3       13.35   1.06    runble  16384
8       test    0       0       3       13.40   1.53    runble  16384
7       test    0       0       3       13.80   1.24    runble  16384
6       test    0       0       3       13.96   1.49    runble  16384
5       test    0       0       3       14.01   1.44    runble  16384
4       test    0       0       3       14.02   1.58    runble  16384
3       test    0       0       2       14.06   1.33    runble  16384
2       sh      0       0       1       15.98   0.07    sleep   16384   8010559f 80105297 801073f3 801065f8 80107917 80107712
1       init    0       0       1       16.06   0.06    sleep   12288   8010559f 80105297 801073f3 801065f8 80107917 80107712
Ready List Processes:
```

Figure 10: ctrl-r and ctrl-p



```
Ready List Processes:
19 -> 10 -> 8 -> 17 -> 12 -> 13 -> 7 -> 4 -> 14 -> 21 -> 9 -> 15 -> 5 -> 11 -> 18 -> 20 -> 3 -> 22 -> 23
Ready List Processes:
8 -> 16 -> 5 -> 13 -> 6 -> 22 -> 7 -> 15 -> 4 -> 20 -> 9 -> 17 -> 21 -> 19 -> 11 -> 12 -> 14 -> 10 -> 23
Ready List Processes:
13 -> 6 -> 8 -> 16 -> 5 -> 7 -> 22 -> 4 -> 9 -> 20 -> 15 -> 21 -> 11 -> 12 -> 19 -> 14 -> 10 -> 17 -> 18
Ready List Processes:
7 -> 22 -> 4 -> 9 -> 20 -> 15 -> 21 -> 11 -> 12 -> 19 -> 14 -> 10 -> 17 -> 18 -> 3 -> 23 -> 6 -> 8 -> 16 -> 13
Ready List Processes:
17 -> 18 -> 3 -> 23 -> 6 -> 8 -> 16 -> 13 -> 5 -> 22 -> 7 -> 4 -> 9 -> 15 -> 20 -> 21 -> 11 -> 12 -> 19
Ready List Processes:
4 -> 9 -> 15 -> 20 -> 21 -> 11 -> 12 -> 19 -> 14 -> 10 -> 17 -> 18 -> 23 -> 3 -> 6 -> 8 -> 13 -> 16 -> 22
Ready List Processes:
17 -> 18 -> 23 -> 3 -> 6 -> 8 -> 13 -> 16 -> 22 -> 7 -> 5 -> 4 -> 9 -> 15 -> 20 -> 11 -> 12 -> 19 -> 14 -> 21
Ready List Processes:
22 -> 7 -> 5 -> 4 -> 9 -> 15 -> 20 -> 11 -> 12 -> 19 -> 14 -> 21 -> 10 -> 18 -> 17 -> 23 -> 6 -> 3 -> 8
Ready List Processes:
17 -> 23 -> 6 -> 3 -> 8 -> 13 -> 16 -> 22 -> 7 -> 5 -> 9 -> 4 -> 15 -> 11 -> 20 -> 19 -> 12 -> 14 -> 21
Ready List Processes:
7 -> 5 -> 9 -> 4 -> 15 -> 11 -> 20 -> 19 -> 12 -> 14 -> 21 -> 10 -> 18 -> 23 -> 6 -> 3 -> 17 -> 8 -> 16 -> 13
Ready List Processes:
```

Figure 11: round robin

In figure 1 we can see that there are in fact exactly 19 processes on the runnable list (all of which are `test` PIDs, while on the output of ctrl−p there all of these PIDs are in the RUNNABLE or RUNNING state.

Each line on the runnable list never has less than 19 processes, all of the PIDs match those of the `test` processes. If we take the front of each output of ctrl−r (head of the list) and find it on the list above we can see that the ordering from it to the tail is maintained and after which we find elements that were ahead of it previously or not on the list (running). For example, on the highlighted line we find 7 (the head) on the line above and see from it to the end of the list is *7, 5 , 9 , 4 , 15 , 11 , 20 , 19 , 12 , 14 , 21* - this exact order is maintained on the highlighted line with the elements previously ahead of 7 now at the end of the list! This matches our expectations.

Because ctrl−r and ctrl−p meet our expectations for the similarities we can prove, and because the round robin ordering described in the second stage was demonstrated we can conclude that this test **PASSES**. Thus requirements 3 and 7.a are fulfilled.

As a resulting of all tests **PASSING** we can conclude that requirements 1-7 have been fulfilled. It should also be noted that the `DEBUG` flag was turned on during testing, so before each addition and after each removal from a list, `checkprocs()` verified, using the process table array, that each process was on one and only one list; additionally, the assertion by each helper function that the lock is held when accessing a list helps to ensure atomicity of transitions. We believe this is sufficient supporting evidence that the mandatory invariant is held.