Project 3 Report
CS333 - Intro to Operating Systems
Winter 2018

Ryan Hoover

2/11/18

0.1 Description

For this assignment, I have taken steps to modernize process management in xv6. All process management is impacted. New console control sequences have been implemented to support testing and debugging of the new process management facility. Increasing the efficiency of process management is the primary focus of this project. The current mechanisms are inefficient in that they traverse a single array (`ptable.proc[]`) that contains all processes, regardless of state.

The new approach will encompass:

- Process scheduling; process allocation and deallocation; transactional semantics for all process state transitions.
- Expand the console debug facility with new control sequences.
- Implement complex concurrency control in the xv6 kernel.
- Use a new Makefile flag (`-DCS333_P3P4`) for conditional compilation. The same flag will be used in Project 4.

0.1.1 New State Lists & Associated Functionality

Six new lists will be added to the current state transition and management approach. Each list corresponds to a state of the xv6 state transition model. The addition of these lists does not mean that we abandon the array of processes that is initialized on boot, but rather will serve to hide how processes are *stored* from how they are *used*. These new lists will be used to more efficiently traverse the existing process array when looking for processes in one of the six possible states.
Every process will be on one of these six lists. Except for `userinit`, `procdump`, and `getprocs`, the code in `proc.c` no longer references the `ptable.proc[]` array.

0.1.2 How & Why

The addition of the state lists affect process management mainly by improving efficiency. In its current form, xv6 traverses the entire `ptable.proc[]` array until it finds a process that matches its search criteria. Almost every function in `proc.c` that accesses a process or performs a state change uses the process table in its entirety. With the addition of the new state lists, this traversal can be kept to a specific list. By following the state transition diagram, and tracing the associated functions that exist already, we can identify which state a process is in when it is accessed within a function, and which state that process will be changed to. This state assertion allows the implementation to directly reference one of these lists, instead of traversing potentially the entire process array.
Insertion and removal from the lists can now be accomplished with a complexity of $O(1)$ rather than $O(n)$.

0.1.3 New Control Commands

New console control sequences have been added to print information about the various state lists that have been added to xv6:

- `control-R`. Print the Process ID's (PID) of all the processes that are currently on the `ready` list.
- `control-F`. Print the number of processes that are on the `free` list.
- `control-S`. Print the PIDs of all the processes that are currently on the `sleep` list.
- `control-Z`. Print the PIDs and Parent PID's (PPIDs) of all the processes that are currently on the `zombie` list.

0.2 Deliverables

The following features were added to xv6:

0.2.1 Process State Lists

To implement the new process state lists to manage state transitions in `proc.c`, a few simple additions were made to the foundational process structure. A `struct StateLists` was added to `proc.c`, and a reference to `StateLists`, "`pLists`", was added to the `ptable` structure.
`StateLists` contains six `proc` pointers, one for each possible process state: `ready`, `free`, `sleep`, `zombie`, `running`, and `embryo`. These pointers will serve as the "head" of their respective lists. Another `proc` pointer, "`next`", was added to the process structure in `proc.h`, which will be used to form the body of the lists, with processes pointing to the next process in their list. These connections are managed by helper functions.
The stubs for the additions are as follows:

- `proc.c`:
  - StateLists structure:

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

  - Additional line in `struct ptable`:
    `struct StateLists pLists;`
- `proc.h`:
  - Within the process structure, `struct proc* next;`

0.2.2 List Management Helper Functions

The addition of the State Lists will inevitably add some complexity to the functionality of `proc.c`. The methods of list manipulation will be identical across all six of the lists, so the common functionality was rolled into several helper functions. These take list and process pointer arguments, to specify which list we are manipulating, and which entry within the list. Another function is there to assert a process is in the correct state (based on a passed in `enum`), or to call `panic` if somehow a process arrives in a state incompatible with the intended lists to be added/removed from.
The stubs for the list management helper functions are as follows:
- `static void assertState(struct proc* p, enum procstate state);`
Compare argument to current process's enumerated state identifier, panic if assertion fails.
- `static int addToStateListHead(struct proc** sList, struct proc* p);`
Initialize list if previously empty, or insert argument process at front of list, pushing other elements back in order.
- `static int addToStateListEnd(struct proc** sList, struct proc* p);`
Initialize list if previously empty, or insert argument process at the end of the list, maintaining previous order.
- `static int removeFromStateList(struct proc** sList, struct proc* p);`

Iterate through specified state list and remove specified process.
- `static struct proc* removeHead(struct proc** sList);`
Remove the first process in the specified list, and return the pointer to that process to the calling routine. Primarily used for the scheduler, the first available "ready" process will be queued.

0.2.3 New Console Control Sequences

Debugging commands were added to print out information about newly implemented state lists.
The additions are as follows:

- `control-R`. Print the PIDs of every process currently on the "Ready" list.
- `control-F`. Print the number of processes on the "Free" list.
- `control-S`. Print the PIDs of every process currently on the "Sleep list."
- `control-Z`. Print PIDs and Parent PIDs of every process currently on the "Zombie" list.

0.2.4 Changes to Process Table Access & Performance Benefits

Since the new State Lists have been implemented, there are only three instances in `proc.c` where the `ptable.proc[]` array is directly accessed: in `getprocs()`, `procdump()`, and `userinit()`.
The process table is allocated and filled with "Unused" processes upon xv6 booting, within `userinit()`. This preliminary allocation is needed, as this is where the process data is actually *stored*, while the State Lists only point to different indices when they are used. Every other method where the process table was referenced in `proc.c` has been changed to reference one or more of the new State Lists instead.
The lists offer improved performance, potentially at a complexity of $O(1)$, as opposed to the $O(n)$ performance the process table offers. Free list insertion and removal can be managed with a complexity of $O(1)$. Insertion into the `sleep`, `zombie`, `running`, and `embryo` lists can also be accomplished in $O(1)$ time. These processes can be inserted at the head of their lists, eliminating traversal entirely. Some processes can also be removed with the same complexity, such as `ready` processes being scheduled to run. Other routines, such as search and remove, or add to the end of a list, will remain $O(n)$, but the size of n will be greatly reduced.

0.3 Implementation

This section will identify all files and line numbers for modifications, without including sceenshots or code.

0.3.1 Process State Lists

New `struct StateLists` & Modifications to `proc`, `ptable` Structures

The addition of the process state lists and their implementation resides almost entirely within `proc.c`.
In `proc.h` (Line 18), a `struct proc` pointer, called `next`, has been added to the process structure. This allows the `proc` to point to another process, thus enabling the formation of a state list.
In `proc.c`:

- Lines 14 - 21. `struct StateLists` is defined here. It contains six `struct proc` pointers, one to be the head of each of the new lists: `ready`, `free`, `sleep`, `zombie`, `running`, and `embryo`.
- Line 35. A reference to `struct StateLists`, called `pLists`, has been added to `struct ptable`.

Handling Concurrency Issues

Concurrency issues were handled rather simply in this project, as the addition of helper functions to manipulate the various State Lists enabled simplicity in adding and removing processes from a list at the time of their state change. Xv6 already handles concurrency with the *spinlock* feature, and the state changes that already take place within the code are wrapped in an `acquire/release` of the `ptable` lock. The state changes themselves that take place within `proc.c` are simply wrapped in three statements: assert the state of the process first to ensure it is of type that it is expected to be in the given function, and then remove it from the list. The process then changes state in one line, and then our helper function to add it to the specified list is run. These can take place in four lines of code, and the acquisition of the *spinlock* in the first place means that this all takes place *atomically*.
The majority of functions in `proc.c` that deal with state changes were changed minimally. Because these state transitions and the list manipulation functions can be added easily, not many changes were needed. Even functions that previously traversed the `ptable` retained the same core functionality after the State Lists modifications were made, and thus the original concurrency controls were left as they were.

- `kill(int)` (Lines 982 – 1065). This function had to be significantly rewritten, which required shuffling of the concurrency control mechanisms. Because this function accesses any process, regardless of *state*, until it finds the matching process, a traversal of each list is initiated until the `p->pid == pid` condition is matched. Sleeping processes that match the comparison are "woken up" into a `RUNNABLE` state (Line 1003). A `release(&ptable.lock)` was added within each loop, if a `pid` matches the passed-in process ID the process. This would kill the process, and `kill` could return true, at which point the lock is released. If no processes are found within a particular list, the next list is searched. If no match is found within any of the six lists, the lock is released and `-1` is returned as error (Line 1070).

Initialization of State Lists

The State Lists are initialized in `userinit` (Lines 231 – 282). The `free` list is initialized in a loop that traverses the `ptable.proc[]` array and adds each index to the list (Lines 237 – 242). At the end of `userinit` the process is made `RUNNABLE`, the `ready` list pointer is directed at it, and the process's `next` pointer is set to `NULL` (Lines 274 – 277). The remainder of the lists, (`sleep`, `zombie`, `embryo`, and `running`) are set to `NULL`, or `0`.

State List Transitions

Creating a New Process occurs in `userinit` and `fork`, which both call `allocproc` to do the actual allocation.

- In the P3 version of `allocproc` (Lines 125 – 184), the form of the original was kept mostly intact. The major difference is the removal of the `ptable.proc[]` traversal, which has been changed to simply assigning a `proc` pointer to the front of the `ptable.pLists.free` list, and if the list is not `NULL`, then to continue on with the initialization (Lines 132 – 134). The first process on the free list is then chosen to be allocated, with the state transition on Line 145 wrapped in our list management helper functions (Lines 141 – 148). These functions assert the state of the incoming process to ensure that it is on the list we intend to remove it from, removed from the list, changed state to `EMBRYO`, and then is added to the `embryo` State List. If allocation of the kernel stack fails, the transition from `free` list to `embryo` list is reversed, with the process being changed back to an `UNUSED` state (Lines 154 – 163).

- `userinit` (Lines 226 – 277). This revised method simply adds assertion and list removal and addition wrappers to the change from `EMBRYO` to `RUNNABLE` (Lines 264 – 272). Since it is the first process in the CPU, we simply assign the `readylist` to it, and it's `next` pointer to `0` (Lines 271, 272).

- `fork` (Lines 352 – 411). The revisions to this function are also minimal, simply wrapping the state changes in list manipulation helpers. If the copy fails after `alloproc` returns, then the process's state is changed back to `UNUSED`, and the lists are updated accordingly (Lines 363 – 374). If the function executes normally and the process is made `RUNNABLE`, the process is added to the *end* of the `ready` list (Lines 397 – 406). This is to implement a "Round-Robin" scheduler.

Yield, Sleep, & Waking Up were changed minimally in order to manage the state lists, and with the exception of `wakeup1`, which required a change to the `ptable.proc[]` traversal, the only changes were wrapping the state changes in our list manipulation helpers.

- `yield` (Lines 816 – 836). The process is removed from the `running` list, and is then added to the end of the `ready` list after its state change from `RUNNING` to `RUNNABLE`.

- `sleep` (Lines 863 – 907). Similar to `yield`, the only changes here involved wrapping the state change from `RUNNING` to `SLEEPING`, and the associated list removal from `ptable.pLists.running`, and the subsequent addition to `ptable.pLists.sleep`.

- `wakeup1` (Lines 924 – 946). This involved a bit more change, as the original function traversed the `ptable.proc[]` array. Instead, there is a check to see if the `ptable.pLists.sleep` State List is `NULL`, and if not, to assign a `struct proc` pointer, `current`, to the head of the list (Line 929). The `sleep` list is then traversed to check for any processes sleeping on `chan`, in which case they are woken up, their state changed to `RUNNABLE`, removed from the `sleep` list, and added to the `ready` list (Lines 935 – 942). This is repeated until the entire `sleep` list has been traversed, greatly reducing the size of n in a complexity of $O(n)$.

Killing a Process involves finding the process to be killed by matching the argument `int` to the `pid` of a given process. This process had to be split up from its single traversal of the `ptable.proc[]` array to a traversal of all six State Lists (Lines 982 – 1065). A `SLEEPING` process that matches the argument `pid` is also changed to a `RUNNABLE` state, with its lists being updated accordingly (Lines 993 – 1000). Traversal of lists will occur until a matching `pid` is found, at which case the lock is released and the function exits successfully, or there is no match, and the function releases the lock and reports failure. Only one process can be killed at a time. This is one of the few cases within the new implementation where worst case of size n could be equal to the original process table.

Modifications to Exit & Wait

These modifications essentially involved splitting the `ptable.proc[]` traversal into several State List traversals.

- `exit` (Lines 460 – 545). The original function accesses the entire `ptable.proc[]` array (process table), looking for abandoned child processes. If one is found, it is passed to `init`. If a `ZOMBIE` state is encountered, a call to `wakeup1(initproc)` is made. The changes to `exit` simply involve splitting this single process table traversal into six State Lists traversals (Lines 488 – 530), each accomplishing the same thing as the original array loop. The `RUNNING` process is then changed to `ZOMBIE`, and the appropriate State Lists are updated (Lines 532 – 540).

- `wait` (Lines 592 – 661). The original function traversed the process table and checked if the current process was equal to `proc`, and would increment the local variable `havekids` if the match occurred. Also, if a process was a `ZOMBIE`, it was deallocated and changed to `UNUSED`. The same functionality holds here, we just need to split it up across the different lists, but not all. We start at the `ptable.pLists.zombie` list, and traverse looking for a child process (Lines 603 – 627). If one is found, the deallocation occurs, the state changed, and the process is removed from the `zombie` list and added to the `unused` list (Lines 611 – 618). This would return the `pid` to the calling routine. If no `zombie` children are reaped, then we move on to the `ready`, `running`, and `sleep` lists, looking for children (Lines 629 – 651).

Changes to the Scheduler & Maintaining Round Robin

A simple algorithm was implemented with the addition of the State lists in order to maintain the round robin style: any additions to the `ready` list were added at the end of the lists, and the `scheduler` always takes the first process from the list. This ensures a first-in, first-out queue that avoids starvation. The new `RUNNABLE` processes that are added to the end of the `ready` list incrementally make their way to the front, as processes are removed from the front to be scheduled, and the next in line moves into the front spot.

- `scheduler` (Lines 721 – 760). This change involved doing away completely with the `ptable.proc[]` array. Instead, we now simply remove the first process on the `ptable.pLists.ready` list. (Lines 733 – 749). A helper function removes the first process (Line 735) and then adds it to the `running` list after the state change, and before its context switch (Line 744).

- `userinit` (Line 271). This is the first process allocated on boot, so the `ptable.pLists.ready` pointer is assigned to this process, after being removed from the `embryo` list.

- `fork` (Line 404). After being removed from the `embryo` list, a process is moved to the end of the `ready` list.

- `yield` (Line 830). After being removed from the `running` list, the process is added back to the end of the `ready` list.

- `wakeup1` (Line 940). After being "woken up" and removed from the `sleep` list, the process is added to the back of the `ready` list.

0.3.2 Helper Functions

To implement the Process State Lists, and to maintain the concurrency requirements, five separate helper functions were implemented to manipulate the new lists. Rolling the similar list management functionalities into reusable functions enabled the code to maintain its simplicity and readability.

Maintaining the Invariant

Xv6 must guarantee that a process exists only on a specific list a specific time, reflecting its current state. A helper function (`assertState`) is called to assert the state of a given process and a specified state, where a list change takes place. This ensures that a given process is not added to the wrong list, and that our code will not attempt to remove it from the wrong list. A `panic` call is made if the assertion fails, and error message is displayed.

Helper Functions Implemented

Five helper functions were implemented in `proc.c` (Lines 1185 – 1287), to maintain cincurrency standards, the invariant quality of state list transitions, and to cleanly and easily manipulate the various State Lists. The following functions were added:

- `static void assertState(struct proc* p, enum procstate state)` (Lines 1185 – 1195).
  A panic is called if the `proc` argument is `NULL` (Lines 1188 – 1190), or if the `proc`'s `state` does not match the passed-in `enum` (Lines 1191 – 1193).

- `static int addToStateListHead(struct proc** sList, struct proc* p)` (Lines 1197 – 1214).
  If the argument State List, `sList`, is `NULL`, then the passed-in `proc` is initialized as the first element in the list (Lines 1202 – 1206). Otherwise, the argument `proc` is set as the first element in the list, its `next` pointer directed at the front element of the `sList`, finally setting the head of the `sList` to be the new `proc` (Lines 1208, 1209).

- `static int addToStateListEnd(struct proc** sList, struct proc* p)` (Lines 1216 – 1235).
  If the argument State List, `sList`, is `NULL`, then the passed-in `proc` is initialized as the first element in the list (Lines 1222 – 1226). Otherwise, a temporary `proc` pointer, `current`, is set to the first element of the `sList`, and a loop is called to traverse `current` to the end of the list. The last element's `next` pointer is set to the argument `proc`, `p`, and `p`'s next pointer is set to `NULL` (Lines 1228 – 1233).

- `static int removeFromStateList(stuct proc** sList, struct proc* p)` (Lines 1238 – 1274).
  This function searches the specified list by comparing the argument pointer, `p`, with each element in the list until it finds a match. First, the function checks to see if `p` is the first element in the list (Line 1244). If so, then no traversal is needed, so it performs another check. If `p` is the first and *only* item in the list, then it's `next` pointer is set to `NULL`, removing it from the list, and the `sList` argument is set to `NULL`, making the list empty (Lines 1246 – 1250). If `p` is just the first item in a list of more than one, then a temporary `proc` pointer is assigned to hold onto the next element in `sList`, `p`'s `next` pointer is set to `NULL` to remove it, and the `sList` head is set to the element being held by the temporary pointer, repairing the list (Lines 1253 – 1256). If `p` is *not* the first element in `sList`, then two temporary pointers are established to point at the list: `current` starts at the 2nd element in `sList` (we already know `p` isn't the 1st element) and performs the comparison at each element, while `previous` follows one element behind `current` in order to hold onto the rest of the list, in case `p` is found (Lines 1261 – 1271). If `current` finds a match, then `previous`'s `next` pointer holds onto `current`'s `next` pointer (`NULL` if `current is the last element`) and `p` is removed from the list (Lines 1265, 1266).

- `static struct proc* removeHead(struct proc** sList)` (Lines 1277 – 1287).
  This function is used when the calling routine knows that it just wants the first element off of a given list. It returns a `struct proc` pointer to the removed process, which will need to be caught by the calling routine. It performs the same removal steps as the conditional in `removeFromStateList` that checks for `p` being the first element (Lines 1277 – 1287).

Control Commands

Additional console debug commands were added in Project 3: `control-R`, `control-F`, `control-S`, and `control-Z`. Support for these commands was added in `console.c` (Lines 217 – 228; 248 – 270), implementation for them was added to `proc.c` (Lines 1290 – 1373), and stubs for them were added to `proc.h` (Lines 91 – 94).

The following files were edited to implement the control commands:

- `console.c`. The function `consoleitr` (Line 190) was edited to emulate the execution of `control-P`, which increments an indicator `int` to perform an action after the switch statement. The functions implemented in `proc.c` cannot be called within the switch, due to the console lock, so an indicator is set a specific number to indicate which control function is to be called, depending on which letter the user specified (Lines 217 – 228). After the switch statement, the integer set earlier will determine which corresponding control command should be called (Lines 248 – 270).
- `proc.c`. added four functions to implement the new console commands:

```
void printReadyList(void)
void printFreeList(void)
void printSleepList(void)
void printZombieList(void)
```

Printing the `ready`, `sleep`, and `zombie` lists are nearly identical, except that `zombie` prints out the `PID` and `PPIDs` of the processes on the list, instead of just the `PIDs`. All three functions assign a temporary pointer to the head of their respective lists, and traverse the entire list, displaying each element's Process ID (Lines 1290 – 1307; 1332 – 1373). Printing the `free` list is slightly simpler: this function traverses the `free` list and increments an `int` at each element, then displays the count (Lines 1310 – 1329).

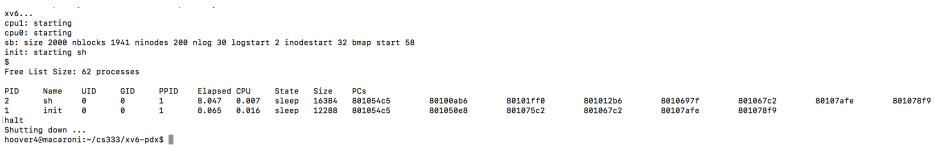

Figure 1: xv6 is booted, `ctrl-F` is pressed, followed by `ctrl-p`.

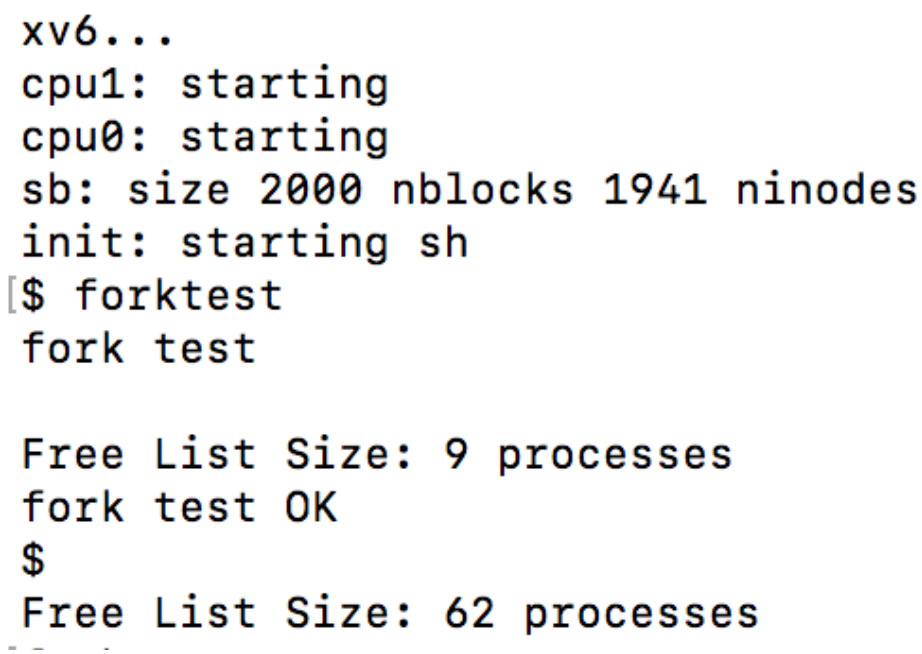

Figure 2: Showing `free list being updated`

0.4 Required Tests

0.4.1 Test 1

Demonstrate that the free list is correctly initialized when xv6 is booted.

In (Figure 1), we see that the `free` list has 62 processes. The max number of processes, `NPROC`, is set to 64. After boot, `shell` and `init` are the only two active processes, which are at this time in the `sleep` state. 64 total processes, minus the two sleeping processes, gives us the 62 processes in the `free` list

Result: This test `PASSES`.

0.4.2 Test 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`).

For this test, I ran a command that would allocate as many processes as allowed, `forktest`, and pressed `control-F` during its execution, which displayed much less than the maximum of 64 processes (62 with `init` and `sh` in `sleep`), and then again after the command executed, once again showing 62 `UNUSED` processes. This demonstrates that the `free` list is correctly updated. See (Figure 2).

Result: This test `PASSES`.

```
$ ps
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
190     ps      0       0       2       0.015   0.007   run     45056
2       sh      0       0       1       618.794 0.085   sleep   16384
1       init    0       0       1       618.806 0.014   sleep   12288
$ kill 2
init: starting sh
zombie!
```

Figure 3: Shell transitions to `ZOMBIE` before `UNUSED`, and then is reallocated.

```
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size    PCs
2       sh      0       0       1       4.828   0.007   sleep   16384   801054c5
1       init    0       0       1       4.844   0.014   sleep   12288   801054c5
kill 2
init: starting sh
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size    PCs
4       sh      0       0       1       2.518   0.017   sleep   16384   801054c5
1       init    0       0       1       10.559  0.032   sleep   12288   801054c5
```

Figure 4: Following the state transitions, `sh` must go to `ZOMBIE`, then `UNUSED`, before being reallocated.

0.4.3 Test 3

Demonstrate the `kill` shell command causes a process to correctly transition to the `ZOMBIE` and then `UNUSED` states.

For this test, we can kill the `shell` process, and watch it be reallocated with a new PID. See (Figures 3 & 4).

While running my test program `zombieFree`, I can see that the `kill` command that it calls (Line 28 transfers a process from `ZOMBIE to UNUSED`. See (Figure 5).

Result: This test `PASSES`.

0.4.4 Test 4

Demonstrate that round-robin scheduling is enforced. Specifically, the processes that are already in the `ready` lists are scheduled before processes added afterwards; any processes 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.

A slight modification to `zombieTest.c` allows the created processes to spin indefinitely, (Line 13), allowing us to hit `ctrl-R` to see that the processes are being scheduled and run repeatedly. By pressing `ctrl-P` we can also see that the processes are continually gaining more time in the CPU, and that the time spent in the CPU is very similar. See (Figures 6, 7, 8, & 9).

Result: This test `PASSES`.

0.4.5 Test 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`).

This test can be seen while running zombieFree, the sleep list is updated several times. First all of the processes are put to sleep, and then run, and then exited to the `ZOMBIE` state. We can catch this mass sleep list usage at first (Figures 10 & 11)

```
19      zombieFree      0       0       3       12.495  0.034   zombie  12288
18      zombieFree      0       0       3       12.506  0.023   zombie  12288
17      zombieFree      0       0       3       12.521  0.027   zombie  12288
16      zombieFree      0       0       3       12.529  0.032   zombie  12288
15      zombieFree      0       0       3       12.539  0.039   zombie  12288
14      zombieFree      0       0       3       12.557  0.033   zombie  12288
13      zombieFree      0       0       3       12.562  0.025   zombie  12288
12      zombieFree      0       0       3       12.568  0.038   zombie  12288
11      zombieFree      0       0       3       12.576  0.030   zombie  12288
10      zombieFree      0       0       3       12.595  0.036   zombie  12288
9       zombieFree      0       0       3       12.610  0.029   zombie  12288
8       zombieFree      0       0       3       12.619  0.042   zombie  12288
7       zombieFree      0       0       3       12.628  0.044   zombie  12288
6       zombieFree      0       0       3       12.637  0.037   zombie  12288
5       zombieFree      0       0       3       12.646  0.036   zombie  12288
4       zombieFree      0       0       3       12.655  0.046   zombie  12288
3       zombieFree      0       0       2       12.698  0.811   sleep   12288   801054c5
2       sh      0       0       1       18.608  0.029   sleep   16384   801054c5        8(
1       init    0       0       1       18.623  0.014   sleep   12288   801054c5        8(
Reaped 5
Reaped 6
Reaped 7
Reaped 8

Free List Size: 5 processes
Reaped 9

Free List Size: 6 processes
Reaped 10

Free List Size: 7 processes
Reaped 11
```

Figure 5: `zombieFree` executing, creating many `ZOMBIE` processes and changing them to `UNUSED`.

```
Ready List Processes:
7 -> 10 -> 12 -> 13 -> 14 -> 11 -> 16 -> 17 -> 15 -> 19 -> 18 -> 20 -> 21 -> 22 -> 3 -> 5 -> 4 -> 6
Ready List Processes:
21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 3 -> 5 -> 4 -> 6 -> 8 -> 9 -> 7 -> 10 -> 12 -> 13 -> 14 -> 11 -> 16 -> 17 -> 15 -> 19
Ready List Processes:
25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 3 -> 5 -> 4 -> 6 -> 8 -> 9 -> 7 -> 10 -> 12 -> 13 -> 14 -> 11 -> 16 -> 17 -> 15 -> 19 -> 18 -> 20 -> 21 -> 22
Ready List Processes:
21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 -> 31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 3 -> 5 -> 4 -> 6 -> 8 -> 9 -> 7 -> 10 -> 12 -> 13 -> 14 -> 11 -> 16 -> 17 -> 15 ->
 19
Ready List Processes:
36 -> 37 -> 38 -> 39 -> 40 -> 41 -> 42 -> 43 -> 44 -> 45 -> 3 -> 5 -> 4 -> 6 -> 8 -> 9 -> 7 -> 10 -> 12 -> 13 -> 14 -> 11 -> 16 -> 17 -> 15 -> 19 -> 18 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 26 -> 27 ->
 28 -> 29 -> 30 -> 31 -> 32 -> 33
```

Figure 6: The PIDs are shown being rotated from the first in line to appearing in the back, after having been scheduled and descheduled, and scheduled again

```
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size    PCs
52              0       0       420     0.131   0.000   embryo  0
51      zombieFree      0       0       3       0.371   0.000   runble  12288
50      zombieFree      0       0       3       0.601   0.010   runble  12288
49      zombieFree      0       0       3       0.835   0.020   runble  12288
48      zombieFree      0       0       3       1.509   0.030   runble  12288
47      zombieFree      0       0       3       1.514   0.060   runble  12288
46      zombieFree      0       0       3       2.148   0.060   runble  12288
45      zombieFree      0       0       3       2.154   0.090   runble  12288
44      zombieFree      0       0       3       2.558   0.090   runble  12288
43      zombieFree      0       0       3       2.564   0.110   runble  12288
42      zombieFree      0       0       3       2.952   0.110   runble  12288
41      zombieFree      0       0       3       3.327   0.130   runble  12288
40      zombieFree      0       0       3       3.701   0.150   runble  12288
39      zombieFree      0       0       3       4.237   0.170   runble  12288
38      zombieFree      0       0       3       4.242   0.200   runble  12288
37      zombieFree      0       0       3       4.577   0.200   runble  12288
36      zombieFree      0       0       3       4.582   0.219   runble  12288
35      zombieFree      0       0       3       4.897   0.220   runble  12288
34      zombieFree      0       0       3       4.903   0.239   runble  12288
33      zombieFree      0       0       3       5.198   0.239   runble  12288
32      zombieFree      0       0       3       5.203   0.261   runble  12288
31      zombieFree      0       0       3       5.339   0.260   runble  12288
30      zombieFree      0       0       3       5.344   0.270   runble  12288
29      zombieFree      0       0       3       5.857   0.270   run     12288
28      zombieFree      0       0       3       5.862   0.310   run     12288
27      zombieFree      0       0       3       5.978   0.320   runble  12288
26      zombieFree      0       0       3       5.983   0.330   runble  12288
25      zombieFree      0       0       3       6.089   0.330   runble  12288
24      zombieFree      0       0       3       6.094   0.340   runble  12288
23      zombieFree      0       0       3       6.388   0.340   runble  12288
22      zombieFree      0       0       3       6.394   0.370   runble  12288
21      zombieFree      0       0       3       6.480   0.370   runble  12288
20      zombieFree      0       0       3       6.486   0.380   runble  12288
19      zombieFree      0       0       3       6.564   0.380   runble  12288
18      zombieFree      0       0       3       6.640   0.390   runble  12288
17      zombieFree      0       0       3       6.645   0.400   runble  12288
16      zombieFree      0       0       3       6.701   0.400   runble  12288
15      zombieFree      0       0       3       6.706   0.410   runble  12288
14      zombieFree      0       0       3       6.761   0.410   runble  12288
13      zombieFree      0       0       3       6.812   0.420   runble  12288
12      zombieFree      0       0       3       6.940   0.430   runble  12288
11      zombieFree      0       0       3       6.945   0.460   runble  12288
10      zombieFree      0       0       3       7.049   0.460   runble  12288
9       zombieFree      0       0       3       7.077   0.490   runble  12288
8       zombieFree      0       0       3       7.102   0.500   runble  12288
7       zombieFree      0       0       3       7.144   0.510   runble  12288
6       zombieFree      0       0       3       7.163   0.530   runble  12288
5       zombieFree      0       0       3       7.222   0.540   runble  12288
4       zombieFree      0       0       3       7.230   0.605   runble  12288
3       zombieFree      0       0       2       7.250   0.614   runble  12288
2       sh      0       0       1       12.210  0.015   sleep   16384   801054c5
1       init    0       0       1       12.226  0.017   sleep   12288   801054c5
```

Figure 7: NOTE: The name zombieFree is longer than the name field in ctrl-p. and throws off the alignment CPU time is shown one column to the right.

PID	Name	UID	GID	PPID	Elapsed	CPU	State	Size	PCs
64	zombieFree		0	0	3	0.878	0.020	runble	12288
63	zombieFree		0	0	3	1.174	0.020	runble	12288
62	zombieFree		0	0	3	1.179	0.030	runble	12288
61	zombieFree		0	0	3	1.464	0.030	runble	12288
60	zombieFree		0	0	3	1.470	0.040	runble	12288
59	zombieFree		0	0	3	2.304	0.040	run	12288
58	zombieFree		0	0	3	2.309	0.070	runble	12288
57	zombieFree		0	0	3	3.114	0.080	runble	12288
56	zombieFree		0	0	3	3.119	0.110	runble	12288
55	zombieFree		0	0	3	3.374	0.110	runble	12288
54	zombieFree		0	0	3	3.380	0.120	runble	12288
53	zombieFree		0	0	3	3.625	0.120	runble	12288
52	zombieFree		0	0	3	3.875	0.130	runble	12288
51	zombieFree		0	0	3	4.115	0.140	runble	12288
50	zombieFree		0	0	3	4.345	0.150	runble	12288
49	zombieFree		0	0	3	4.579	0.160	runble	12288
48	zombieFree		0	0	3	5.253	0.170	run	12288
47	zombieFree		0	0	3	5.258	0.200	runble	12288
46	zombieFree		0	0	3	5.892	0.200	runble	12288
45	zombieFree		0	0	3	5.898	0.230	runble	12288
44	zombieFree		0	0	3	6.302	0.230	runble	12288
43	zombieFree		0	0	3	6.308	0.250	runble	12288
42	zombieFree		0	0	3	6.696	0.250	runble	12288
41	zombieFree		0	0	3	7.071	0.270	runble	12288
40	zombieFree		0	0	3	7.445	0.290	runble	12288
39	zombieFree		0	0	3	7.981	0.310	runble	12288
38	zombieFree		0	0	3	7.986	0.340	runble	12288
37	zombieFree		0	0	3	8.321	0.340	runble	12288
36	zombieFree		0	0	3	8.326	0.359	runble	12288
35	zombieFree		0	0	3	8.641	0.360	runble	12288
34	zombieFree		0	0	3	8.647	0.379	runble	12288
33	zombieFree		0	0	3	8.942	0.379	runble	12288
32	zombieFree		0	0	3	8.947	0.401	runble	12288
31	zombieFree		0	0	3	9.083	0.400	runble	12288
30	zombieFree		0	0	3	9.088	0.410	runble	12288
29	zombieFree		0	0	3	9.601	0.410	runble	12288
28	zombieFree		0	0	3	9.606	0.450	runble	12288
27	zombieFree		0	0	3	9.722	0.450	runble	12288
26	zombieFree		0	0	3	9.727	0.460	runble	12288
25	zombieFree		0	0	3	9.833	0.460	runble	12288
24	zombieFree		0	0	3	9.838	0.470	runble	12288
23	zombieFree		0	0	3	10.132	0.470	runble	12288
22	zombieFree		0	0	3	10.138	0.500	runble	12288
21	zombieFree		0	0	3	10.224	0.500	runble	12288
20	zombieFree		0	0	3	10.230	0.510	runble	12288
19	zombieFree		0	0	3	10.308	0.510	runble	12288
18	zombieFree		0	0	3	10.384	0.520	runble	12288
17	zombieFree		0	0	3	10.389	0.530	runble	12288
16	zombieFree		0	0	3	10.445	0.530	runble	12288
15	zombieFree		0	0	3	10.450	0.540	runble	12288
14	zombieFree		0	0	3	10.505	0.540	runble	12288
13	zombieFree		0	0	3	10.556	0.550	runble	12288
12	zombieFree		0	0	3	10.684	0.560	runble	12288
11	zombieFree		0	0	3	10.689	0.590	runble	12288
10	zombieFree		0	0	3	10.793	0.590	runble	12288
9	zombieFree		0	0	3	10.821	0.620	runble	12288
8	zombieFree		0	0	3	10.846	0.630	run	12288
7	zombieFree		0	0	3	10.888	0.640	runble	12288
6	zombieFree		0	0	3	10.907	0.660	runble	12288
5	zombieFree		0	0	3	10.966	0.670	runble	12288
4	zombieFree		0	0	3	10.974	0.727	runble	12288
3	zombieFree		0	0	2	10.994	0.722	sleep	12288

Figure 8: Tracking CPU time increase are equally gradual in each process.

```
64      zombieFree      0       0       3       2.535   0.070   runble  12288
63      zombieFree      0       0       3       2.831   0.070   runble  12288
62      zombieFree      0       0       3       2.836   0.080   runble  12288
61      zombieFree      0       0       3       3.121   0.080   runble  12288
60      zombieFree      0       0       3       3.127   0.090   runble  12288
59      zombieFree      0       0       3       3.961   0.100   runble  12288
58      zombieFree      0       0       3       3.966   0.120   runble  12288
57      zombieFree      0       0       3       4.771   0.130   runble  12288
56      zombieFree      0       0       3       4.776   0.160   runble  12288
55      zombieFree      0       0       3       5.031   0.160   runble  12288
54      zombieFree      0       0       3       5.037   0.170   runble  12288
53      zombieFree      0       0       3       5.282   0.170   runble  12288
52      zombieFree      0       0       3       5.532   0.180   runble  12288
51      zombieFree      0       0       3       5.772   0.190   runble  12288
50      zombieFree      0       0       3       6.002   0.200   runble  12288
49      zombieFree      0       0       3       6.236   0.210   runble  12288
48      zombieFree      0       0       3       6.910   0.220   runble  12288
47      zombieFree      0       0       3       6.915   0.250   runble  12288
46      zombieFree      0       0       3       7.549   0.250   runble  12288
45      zombieFree      0       0       3       7.555   0.280   runble  12288
44      zombieFree      0       0       3       7.959   0.280   runble  12288
43      zombieFree      0       0       3       7.965   0.300   runble  12288
42      zombieFree      0       0       3       8.353   0.300   runble  12288
41      zombieFree      0       0       3       8.728   0.320   runble  12288
40      zombieFree      0       0       3       9.102   0.340   runble  12288
39      zombieFree      0       0       3       9.638   0.360   runble  12288
38      zombieFree      0       0       3       9.643   0.390   runble  12288
37      zombieFree      0       0       3       9.978   0.390   run     12288
36      zombieFree      0       0       3       9.983   0.409   run     12288
35      zombieFree      0       0       3       10.298  0.420   runble  12288
34      zombieFree      0       0       3       10.304  0.439   runble  12288
33      zombieFree      0       0       3       10.599  0.439   runble  12288
32      zombieFree      0       0       3       10.604  0.461   runble  12288
31      zombieFree      0       0       3       10.740  0.460   runble  12288
30      zombieFree      0       0       3       10.745  0.470   runble  12288
29      zombieFree      0       0       3       11.258  0.470   runble  12288
28      zombieFree      0       0       3       11.263  0.510   runble  12288
27      zombieFree      0       0       3       11.379  0.510   runble  12288
26      zombieFree      0       0       3       11.384  0.520   runble  12288
25      zombieFree      0       0       3       11.490  0.520   runble  12288
24      zombieFree      0       0       3       11.495  0.530   runble  12288
23      zombieFree      0       0       3       11.789  0.530   runble  12288
22      zombieFree      0       0       3       11.795  0.560   runble  12288
21      zombieFree      0       0       3       11.881  0.560   runble  12288
20      zombieFree      0       0       3       11.887  0.570   runble  12288
19      zombieFree      0       0       3       11.965  0.570   runble  12288
18      zombieFree      0       0       3       12.041  0.580   runble  12288
17      zombieFree      0       0       3       12.046  0.590   runble  12288
16      zombieFree      0       0       3       12.102  0.590   runble  12288
15      zombieFree      0       0       3       12.107  0.600   runble  12288
14      zombieFree      0       0       3       12.162  0.600   runble  12288
13      zombieFree      0       0       3       12.213  0.610   runble  12288
12      zombieFree      0       0       3       12.341  0.610   runble  12288
11      zombieFree      0       0       3       12.346  0.650   runble  12288
10      zombieFree      0       0       3       12.450  0.639   runble  12288
9       zombieFree      0       0       3       12.478  0.671   runble  12288
8       zombieFree      0       0       3       12.503  0.680   runble  12288
7       zombieFree      0       0       3       12.545  0.690   runble  12288
6       zombieFree      0       0       3       12.564  0.710   runble  12288
5       zombieFree      0       0       3       12.623  0.720   runble  12288
4       zombieFree      0       0       3       12.631  0.777   runble  12288
3       zombieFree      0       0       2       12.651  0.722   runble  12288
```

Figure 9: Further tracing of updated CPU time

```
Sleep List Processes:
1 -> 2 -> 13 -> 57 -> 12 -> 22 -> 59 -> 17 -> 44 -> 21 -> 18 -> 5 -> 7 -> 24 -> 64 -> 14 -> 3 -> 9 -> 35 -> 31 -> 32 -> 54 -> 4 -> 53 -> 58 -> 25 -> 20 -> 15 -> 40 -> 45 -> 55 -> 23 -> 52 -> 36 -> 30 ->
 10 -> 11 -> 28 -> 29 -> 49 -> 16 -> 51 -> 47 -> 63 -> 39 -> 60 -> 48 -> 61 -> 27 -> 37 -> 8 -> 38 -> 50 -> 34 -> 33 -> 62 -> 46 -> 56 -> 41 -> 19 -> 26 -> 42 -> 6

Sleep List Processes:
1 -> 2 -> 36 -> 32 -> 28 -> 53 -> 64 -> 45 -> 4 -> 58 -> 60 -> 49 -> 52 -> 55 -> 16 -> 10 -> 25 -> 29 -> 23 -> 51 -> 40 -> 63 -> 48 -> 27 -> 30 -> 61 -> 47 -> 33 -> 11 -> 37 -> 19 -> 39 -> 50 -> 22 -> 3
4 -> 8 -> 46 -> 62 -> 56 -> 38 -> 21 -> 5 -> 41 -> 26 -> 13 -> 35 -> 42 -> 7 -> 57 -> 43 -> 31 -> 24 -> 6 -> 59 -> 12 -> 9 -> 18 -> 17 -> 15 -> 14 -> 44 -> 54 -> 3

Sleep List Processes:
1 -> 2 -> 28 -> 40 -> 27 -> 51 -> 8 -> 58 -> 55 -> 16 -> 22 -> 11 -> 33 -> 61 -> 19 -> 37 -> 29 -> 47 -> 39 -> 5 -> 21 -> 30 -> 48 -> 50 -> 62 -> 38 -> 35 -> 34 -> 43 -> 46 -> 56 -> 41 -> 24 -> 13 -> 18
 -> 59 -> 3 -> 54 -> 26 -> 6 -> 31 -> 60 -> 12 -> 42 -> 15 -> 17 -> 57 -> 36 -> 14 -> 44 -> 9 -> 7 -> 20 -> 63 -> 64 -> 45 -> 53 -> 25 -> 52 -> 49 -> 4 -> 10 -> 32

Sleep List Processes:
1 -> 2 -> 3
```

Figure 10: Sleep list heavily populated, and then emptied.

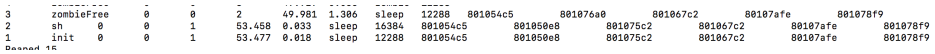

Figure 11: Sleep processes after initial allocation.

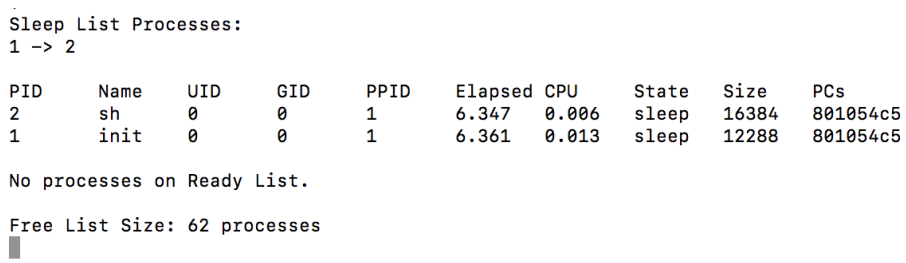

Figure 12: ctrl-s and ctrl-f demonstrated

Result: This test `PASSES`.

0.4.6 Test 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`).

Several screenshots up to this point have demonstrated the updating of the zombie lists, most notably (Figure 5).

Result: This test `PASSES`.

0.4.7 Test 7

Demonstrate that output for the console commands `control-R`, `control-F`, `control-S`, and `control-Z` are correct.

(Figure 12) (Figures 13)

Result: This test `PASSES`.

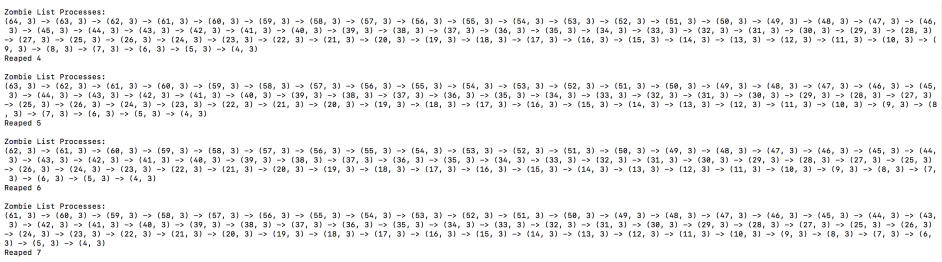

Figure 13: Showing ctrl-z