Description

For this project, I explored the creation of more efficient, modern process management within xv6. I predominately focused on process management efficiency by replacing the previous, inefficent method of traversing a single array (`ptable.proc[]`) with a new method that uses state lists. The new approach to process management encompassed process scheduling, process allocation and deallocation, and new transactional semantics for all process state transitions. With the new method, I also learned about complex concurrency controls in the xv6 kernel. Console debugging facilities were expanded with new control sequences.

Deliverables

The following features were added to xv6:

- New state lists were added to track process state. These lists are used during state transitions for processes and have replaced accessing the ptable in all but three functions (`userinit()`, `getprocs()`, and `procdump()`). The benefit of this is performance. Nearly all process access is now $\Theta(1)$ as opposed to the previous $\Theta(n)$. The only exeception is accessing processes in the ready list which is still $\Theta(n)$; however, n is much smaller than the maximum number of processes.
- Four new helper functions, `insertHead()`, `insertTail()`, `removeFromStateList()`, and `assertState()`, were added to manage insertion and removal in the new state lists.
- Control-R, Control-S, Control-Z, and Control-F mechanisms, which display debugging information for the new state lists.

Implementation

State Lists

To implement the new state lists, the following files were modified:

- `proc.h`. The `proc` struct was modified to include a new field `struct proc *next` which points to the next process within a state list (Line 78).
- `proc.c` was modified in the following was for the implemenation of process state lists:
  - Lines 14 – 21 The new `StateLists` struct was added. It contains the 6 lists which correspond to process states: `free`, `ready`, `running`, `sleeping`, `zombie`, and `embryo`.
  - Lines 28 The `ptable` struct was modified to include the new `StateLists` struct named `pLists`.
  - To prevent concurrency issues, all state transitions – removal and insertion into a state list – are performed after acquiring the ptable lock.
  - State lists are initialized within the `userinit()` function (Lines 149 – 164). All list are initialized to be empty. The free list is initialized to contain the maximum number of processes, defined by the macro `NPROC`.
  - The function `allocproc()` was modified to remove the head of the free list and add it to the embryo list Lines 89 – 94. If `allocproc()` cannot successfully allocate a *kernel* stack, then the processes is added back to the free list (Lines 108 – 108).
  - New processes are created with the `fork()` function which has been modified to use the newly implemented state lists. If the new process created by `fork()` is not successfully copied, then the process is removed from the embryo list and placed back in the free list (Lines 239 – 244). The process is finally added to the ready list at the end of the function (Lines 272 – 277).
  - The `yield()` function was also modified to include state transitions using the newly implemented lists. For it, the currently running process is removed from the running list and added to the ready list (Lines 634 – 639).
  - The `sleep()` function was also modified to handle state transitions from the running list to the sleeping list (Lines 690 – 695).
  - An alternative `wakeup1()` funciton was defined within conditional compilation for the `CS333_P3P4` flag (Lines 722 – 737). The funciton handles transitions for any sleeping processes on the channel to become runnable (added to the ready list).
  - Similarly, an alternative `kill()` function was implemented (Lines 773 – 821). The function searches through all state lists for a particular PID. If the process is found, then the killed flag is set. Slepping processes are woken up by transitioning from the sleep list to the ready list (Lines 807 – 818).

  - The `exit()` and `wait()` functions were modified as follows:
    * Lines 330 – 393 An alternative `exit()` function was implemented to be used if the conditional compilation flag, `CS333_P3P4`, is defined. The function searches all state lists (except for the free list) for any children of the current process, sets their parent process to be the initial process. The current process is then removed from the running list and placed in the zombie list.
    * Lines 440 – 504 Similar to `exit()`, an alternative wait function was implemented for state lists in the event that the conditional compilation flag, `CS333_P3P4` is defined. The function searches all state lists (except for the free list) for children processes. If the process has children then it is put to sleep. Children that are in the zombie list are reaped and transition to the free list.

- The `scheduler()` function was modified to use the new state lists by implementing an alternative function for the conditional compilation flag `CS333_P3P4` (Lines 560 – 602). The new state list based scheduler checks if the ready list is not empty; if there is a ready process, then the head of the ready list is dequeued and changed to running. By still using timer interrupts for time slices and enqueueing ready processes at the end of the list, the scheduler maintains a round robin style.

Helper Functions

Helper functions were added to improve list managment of the state lists. The implementation modified `proc.c` as follows:

- Lines 1002 – 1010 defines an insertion function for inserting to the head of a list, `insertHead()`. This function is used for all state lists except for the `ready` list to maintain efficeient state transitions.
- Lines 1012 – 1031 defines an insertion function for inserting to the tail of a state list named `insertTail()`. This is used to maintain the *FIFO* nature of *runnable* processes.
- Lines 1033 – 1054 defines a function, `removeFromStateList()` to remove a specified process `struct proc *p` from a `struct StateLists **sList`. This helper function is used for all state transitions wherein a process must be removed from one state list and placed in another.
- Lines 1056 – 1063 contains the definition for the `assertState()` helper function which is used to verify that the invariant is maintained by checking the state of a process to the expected state. Failure results in panic.

Control Commands

The following files were modified to add the new control sequences control−r, control−s, control−f, and control−z:

- `console.c`. The new control commands were implementd by modifying various functions.
  - Lines 216 – 227 The `consoleintr()` function for handling console interruptions was modified to include new cases for r, z, f, and s. This is used to recognized the new commands.
  - Lines 247 – 254 Similar to the control−p command, helper functions are called after releasing the console lock. These functions are definied within the file `proc.c`.
- `proc.c`. Four functions were added to proc.c to be used for the new control commands:
  - Lines 1065 – 1080 A `printReadyList()` function was added which acquires the ptable lock and the prints the pid of each process within the ready list. The information is displayed in the form `1 -> 2` where 1 is pid of the head of the list, 2 is the next pid, and so on. If the list is empty then an appropriate message is displayed.
  - Lines 1082 – 1097 A similar function, `printSleepList()`, was implemented to be used for control−s. Similar to the `printReadyList()` described above, `printSleepList()` acquires the ptable lock and loops through the sleep state list printing the pid of each process. Again, an appropriate message is displayed in the case of an empty sleeping list.
  - Lines 1100 – 1115 `printZombieList()` is the third function for printing state list information. The function is used by the control−z command. This function alao acquires the ptable lock, loops through the state list, and prints information for each process. However, this function displays both the PID and PPID of each process in the form `)(PID, PPID) ->` . If the zombie state list is empty then an appropriate message is displayed.
  - Lines 1117 – 1127 `freeListSize()` is used by control−f to print the number of unused process (i.e. the size of the free list). The function acquires the ptable lock and walks down the free state list incrementing a counter. The counter is then printed.

Testing

Free List Initialization

I tested that the free list is correctly intialized by using the control−f command after the userinit function is done and the shell is created and compare the results to the control−p output. I expectd to see control−f report that there are 62 unused processes since two should be used for init and sh. The output of the test is shown below:

As expected, control−f reports that there 62 unused processes.

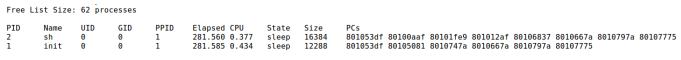

Figure 1: Free List Initialization

This test `PASSES`.

Free List Updating

This test was broken into two sub−tests. First, I tested that the free list is correctly updated for process allocation by creating an extra processe (for simplicity I simply made another shell) and then used the control−f command. Then for the second test, I deallocated the process by calling kill on the newly created shell. I expected the output of control−f to show the size of the free list to be one less after allocation anf for the number of free processes to be back to the initial 62 after deallocation. The output for both sub−tests is shown below:

From the first control−f output, I see that the number of free processes in the free list is one less as

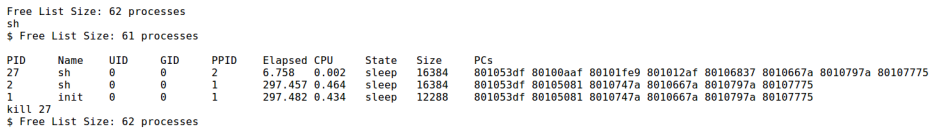

Figure 2: Allocation and Deallocation Free List Updating Test

expected.
This sub−test `PASSES`.
In the second control−f after the shell is killed, the number of free processes is back to the inital 62.
This sub−test `PASSES`.
Since both sub−tests passed, this test `PASSES`.

Round Robin Scheduling

To demonstrate that the round robin scheduling is maintained by the scheduler, I used the `testSched()` test program in combination with the control−r command to show that the scheduler is let each process run and that they are running in the correct order. I expect the output of control−r to show that whatever process was running (i.e. missing from the ready list) was later at the end of the list. I also expect the output to verify that processes at the head of the list are the next to run. The output for the test can be seen below:

The output shows (with some error due to my clicking speed) that process at the beginning of the list

```
Ready List Processes:
30 -> 32 -> 31 -> 34 -> 35 -> 36 -> 33 -> 37 -> 38 -> (NULL)
Ready List Processes:
36 -> 33 -> 37 -> 38 -> 39 -> 40 -> 30 -> 32 -> 31 -> (NULL)
Ready List Processes:
39 -> 40 -> 30 -> 32 -> 31 -> 34 -> 35 -> 36 -> 33 -> (NULL)
Ready List Processes:
32 -> 31 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> (NULL)
Ready List Processes:
30 -> 33 -> 32 -> 31 -> 34 -> 35 -> 36 -> 37 -> 38 -> (NULL)
Ready List Processes:
33 -> 32 -> 31 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> (NULL)
Ready List Processes:
35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 30 -> 33 -> 32 -> (NULL)
Ready List Processes:
35 -> 36 -> 37 -> 38 -> 39 -> 40 -> 30 -> 33 -> 32 -> (NULL)
Ready List Processes:
37 -> 38 -> 39 -> 40 -> 30 -> 33 -> 32 -> 31 -> 34 -> (NULL)
Ready List Processes:
32 -> 31 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> (NULL)
Ready List Processes:
32 -> 31 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40 -> (NULL)
```

Figure 3: Round Robin Scheduling Test

run before other processes and are later queued to the end of the list.
This test `PASSES`.

Process Death

This test was divided into three sub−tests. The first sub−test used the `kill()` function to show that a process is removed from the zombie list (which is shown via the control−z command). The second sub−test showed that `exit()` correctly causes a process to move to the zombie list (again shown with control−z). The third sub−test showed that `wait()` causes a process to move to the free list. All sub−tests are performed with a test program `testProcDeath.c` running in the background, but each subtest will be concerned with different portions of the output. The test program loops and creates zombie processes via `exit()` for children and proceeds to call `kill()` on children to remove them from the list. The output which I will refer to for the first two sub−tests can be seen below:

For the first sub−test, I expected each call to `kill()` to remove the pid from the zombie list. As seen in

```
Zombie List Processes:
No Zombie Processes
Zombie List Processes:
(PID5,PPID4) -> (NULL)
Reaped PID:5
Zombie List Processes:
(PID6,PPID4) -> (NULL)
Reaped PID:6
Reaped PID:7
Zombie List Processes:
(PID8,PPID4) -> (NULL)
Reaped PID:8
Reaped PID:9
```

Figure 4: Process Death SuB−Test 1 and 2

the output, each time a pid is reaped, it is removed from the list.
Thus, this sub−test `PASSES`.
For the second sub−test, I expect the exiting children to be placed in the zombie list. This behavior can be seen in the addition of new PID zombies in the control−z output.
This sub−test `PASSES`.
For the final sub−test, I used the control−f command at the beginning of the test program. I expected the free list size to be 11 less than the size at initialization since the test program made 10 children (11 processes total for the test program and its children). After each `wait()` within the test program, the size of the free list should increase by one. The output can be seen below:

The results shown in the output above show 51 free process at the beginning of the test program, but after each `wait()` there is one more free process as expected.
This sub−test `PASSES`. Since all three sub−tests passed, this test `PASSES`.

```
Free List Size: 51 processes
Free List Size: 51 processes
Reaped PID:5
Free List Size: 52 processes
Reaped PID:6
Free List Size: 53 processes
Reaped PID:7
Free List Size: 54 processes
Reaped PID:8
Free List Size: 55 processes
Reaped PID:9
Free List Size: 56 processes
Reaped PID:10
Free List Size: 57 processes
Reaped PID:11
Free List Size: 58 processes
Reaped PID:12
Free List Size: 59 processes
Reaped PID:13
Free List Size: 60 processes
Reaped PID:14
zombie!
Free List Size: 62 processes
```

Figure 5: Process Death Sub−Test 3

Console Commands

This test was divided into sub−tests for each command. For the first test, I tested the control−f command in a similar fashion as before by using the command after initialization and after making the maximum number pf processes. I expected the output of this test to show 62 unused processes after initialization and 0 after 64 have been made. The output of the test can be seen below:

As expected, the output matches my predictions.

```
18      sh      0       0       17      68.636  0.014   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
17      sh      0       0       16      70.563  0.014   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
16      sh      0       0       15      71.059  0.014   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
15      sh      0       0       14      71.508  0.014   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
14      sh      0       0       13      72.032  0.009   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
13      sh      0       0       12      72.624  0.014   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
12      sh      0       0       11      73.193  0.014   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
11      sh      0       0       10      73.807  0.020   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
10      sh      0       0       4       74.692  0.230   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
4       sh      0       0       3       133.558 0.044   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
3       sh      0       0       2       134.140 0.009   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
2       sh      0       0       1       135.249 0.252   sleep   16384   801053df 80105081 8010747a 8010667a 8010797a 80107775
1       init    0       0       1       135.272 0.377   sleep   12288   801053df 80105081 8010747a 8010667a 8010797a 80107775
Free List Size: 1 processes
sh
$ Free List Size: 0 processes
```

Figure 6: Ctrl−f 64 Active Processes Test

Thus, this sub−test `PASSES`.

For the next sub−test, I tested the control−s command after initialization and after making multiple shells (sleeping processes). I expected the output to display more processes on the list as shells were created. I also expected the order to be indicative of inserting to the head of the list. The output is below:

The output indicated that as shells were created, more processes were added to the sleeping list in the

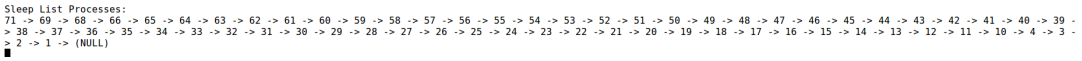

Figure 7: Ctrl−s 64 Sleeping Processes Test

correct order.
This sub−test PASSES.
For the third sub−test, I tested the control−r command. I already indirectly verified the output of control−r in the round robin testing, so I tested that the command correctly displayed nothing for an empty ready list. The output is below:
The output showed that there are no ready processes as suspected.

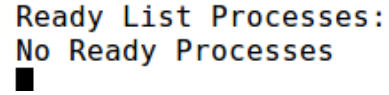

Figure 8: No Ready Processes Test

This sub−test PASSES.
The last sub−test was for the control−z command. Similarly, the addition and subtraction of processes from the zombie list was indirectly tested in the process death subsection, thus I simply tested that the command correctly displayed an empty list when no zombie processes exist. The output for the test is the following:
The command output showed no zombie process.

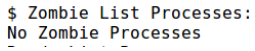

Figure 9: Empty Zombie Test

This sub−test PASSES.
Since all sub−tests passed, this test PASSES.