Description

For this project, I learned about locks on kernel level, concurrent data structures; implementing new system calls to show process ownership and information via a process's UID, GID, PPID; implementing tracking of the CPU usage time for processes; implementing user level commands to display process state and process execution time; and to again modify the control−p display to include the new information.

Deliverables

The following features were added to xv6:

- `getuid ()`, `getgid ()`, `getppid ()` system calls which return the value of the UID, GID, or PPID respectively. UID/GID values are limited to values within the boundaries of 0 and 32767. To adhere to convention, the PPID of the first process is always itself.
- Two new system calls, `setuid ()` and `setgid ()`, which set the UID or GID of a process.
- `_set` and `_get` built−in commands for getting or setting the UID/GID of the shell.
- the funcctionality of these new system calls is tested via a user command `testuidgid`.
- Each process now records the value of `ticks` when the process enters the CPU. The value is used to calculate the amount of time a process spends in the CPU.
- A new user command, `time`, to display the number of seconds a program takes to run. The command can be called with no arguments to display the time to run the `time` command or with multiple `time` commands thus displaying the subsequent time for the nested `time` commands.
- Another new user command, `ps` which displays information on all currently active process. The number of processes displayed is limited to a maximum value.
- Modifications to the existing control−p mechainsm, which displays debugging information, to include the UID, GID, PPID, and CPU usage time for each process.

Implementation

Get UID, GID, PPID System Calls

The following files were modified to add the `getuid()`, `getgid()`, and `getppid()` system calls.

- `user.h`. The user−side functions for the system calls were added (Lines 32 – 34). The functions take a *void* parameter as an argument and return the repective ID as a uint. The prototypes are:
  `uint getuid(void);`
  `uint getgid(void);`
  `uint getppid(void);`

- `syscall.h`. The `getuid()`, `getgid()`, and `getppid()`, system call numbers were created by apending to the existing list (Lines 26 – 28).

- `syscall.c`. Modified to include the kernel−side function prototypes (Lines 106 – 108); entries in the *function dispatch table* `syscalls[]` (Lines 141 – 143); entries into the `syscallnames[]` to print the system call name when the `PRINT_SYSCALLS` flag is defined (Lines 178 – 180). All prototypes here are defined as taking a *void* parameter as the function call arguments are passed into the kernel on the stack.

- `usys.S`. The user−side stubs for the new system calls were added (Lines 34 – 36). These stubs use a macro that essentially just traps into kernel−mode.

- `sysproc.c`. Contains the kernel−side implementation of the system calls in `sys_getuid()` (Lines 111 – 115), `sys_getgid()` (Lines 117 – 121), and `sys_getppid()` (Lines 123 – 131). The routines operate in a similar fashion by returning the value of the correct field from the `proc struct`. `getppid ()` checks the current pid field to determine if the process is `init` which has no parent and by UNIX convention is its own parent; otherwise it returns the pid field of the current process' parent.

- `proc.h`. Two new fields were added to `struct proc` named `uint uid` and `uint gid` for storing the uid and gid for each process (Lines 72 – 73).

- `param.h`. A new macro, `DEFAULTUID`, was added to be the default UID/GID for the `init` process (Lines 17).

- `proc.c`. The routines `userinit()` (Lines 121 – 124) and `fork()` (Lines 170 – 174) were modified to correctly set the UID and GID on process creation. Within `fork()`, UID and GID are copied to the child process from the parent process. `userinit()` was modified to set the UID and GID of the first process to a default value, `DEFAULTUID`.

Set UID, GID System Calls

- `user.h`. The user−side functions for the system calls were added (Lines 35 – 36). The `setuid()` and `setgid()` system calls take a `uint` parameter as an argument which is the value to be set as the UID or GID and return an `int` with −1 being failure and 0 being success. The set function prototypes are:
  `int setuid(uint);`
  `int setgid(uint);`

- `syscall.h`. The `setuid()` and `setgid()` system call numbers were created by appending to the existing list (Lines 29 – 30).

- `syscall.c`. Modified to include the kernel−side function prototypes (Lines 109 – 110); entries in the *function dispatch table* `syscalls[]` (Lines 144 – 145); entries in the `syscallnames[]` to print the system call name when the `PRINT_SYSCALLS` flag is defined (Lines 181 – 182). All prototpyes are defined as taking a *void* parameter as the function call arguments are passed into the kernel on the stack.

- `usys.S`. The user−side stubs for the new system calls were added (Lines 37 – 38). These stubs use a macro that essentially traps into the kernel−mode.

- `sysproc.c`. Contains the kernel−side implementation of the system calls in `sys_setuid()` (Lines 133 – 141) and `sys_setgid()` (Lines 143 – 151). The routines get an `int` argument off of the stack, return −1 if failure, otherwise return the user−side function `setuid()` or `setgid()` defined in the `proc.c` file is called with the `int` argument correctly cast to an `uint` (Lines 140, 150). Correct UID/GID value checking is handled within the user−side function and in the event of failure returns −1; otherwise, they return 0.

- `proc.c`. The user−side implementations for `setuid ()` and `setgid()` were added (Lines 618 – 638). The functions check that the value being set is within the specified boundaries for UIDs and GIDs; if not, then the function returns −1 to indicate an error, otherwise the function assigns the argument value to the correct `proc struct` field amd returns 0.

Built−in _get and _set UID, GID Commands

Functionality for the shell parser to understand built−in commands was added to `sh.c` to set and get the UID/GID of the currently executing shell. The code for this is wrapped in the `USE_BUILTINS` conditional compilation flag (line 260 – 265).

Get and Set UID, GID, PPID Test Program

A new user command `testuidgid` was added to test the new system calls `getuid()`, `getgid()`, `getppid()`, `setuid()`, and `setgid()` in the file `testuidgid.c`. The test program is designed to test setting invalid UID/GIDs (defined as outside the boundaries of 0 and 32767) and to test valid inputs (within and including the boundaries). The results are printed to the standard output.

CPU Time

- `proc.h`. Two new fields were added to the `proc struct`, `uint cpu_ticks_int` and `uint cpu_ticks_total` (Lines 74 – 75).

- `proc.c`. The routines `scheduler()` was modified to increment the `cpu_ticks_in` when a process enters the CPU after a *context switch* (Lines 334 – 336). Similarly, the routine `sched()` was modified to set the `cpu_ticks_total` after the *context switch* for a process to leave the CPU (Lines 379 – 381).

Getprocs System Call

- `user.h`. The user−side function prototype for the `getprocs()` system call was added (Lines 37). The function takes a `uint` which is the max size of the userprocs table and a pointer to an `uproc struct`. The prototype is:
  `int getprocs(uint max, struct uproc*);`
  The file `uproc.h` contains the `uproc` definition.

- `syscall.h`. The `getproc ()` system call number was created by appending to the existing list (Line 31).

- `syscall.c`. Modified to include the kernel−side function prototype (Line 111); an entry in the *function dispatch table* `syscalls[]` (Line 146); an entry in the `syscallnames[]` to print the system call name when the `PRINT_SYSCALLS` flag is defined (Line 183).

- `usys.S`. The user−side stub for the new system call was added (Line 38).

- `sysproc.c`. Contains the kernel−side implementation of the system call `sys_getprocs()` (Lines 153 – 163). This routine removes the int argument and pointer argument off of the stack and passes them into the routine `getprocs()` defined in the file `proc.c`. The `int` argument is expected to be a `uint` and is thus cast to one; the pointer argument is expected to be a `struct uproc*`. The routine `getprocs ()` can fail and returns −1 in such an event, otherwise it returns the number of entries in the `struct uproc*` either of which is returned via `sys_getprocs()` (Lines 158 – 163).

- `proc.c`. The user−side function `getprocs()` was added (Lines 640 – 673). The `uint` argument `max` is maximum number of entries that can be added to the `uproc struct` table; the `struct uproc*` is a pointer to a `uproc struct` and will be utilized like an array. The routine acquires the `ptable` lock (Line 646) and then loops through the `ptable` looking for any active process; the information of active processes is copied over to the `uproc` table and the number of entities added is update. The routine releases the `ptable` lock and returns. If no entries into the table are made then the return value is −1; otherwise, the return value is the number of entries made.

PS Command

The `ps` user command was implemented in `ps.c`. This command invokes the new `getprocs ()` system call to fill in the `uproc struct` table. The command displays a header indicating the contents for each column with one process printed per line to the standard output. Due to stack size restrictions, the table of `uproc struct` is dynamically allocated on the heap (lines 28, 55); the max size of the table is set within the code to a default size (line 25) or a value from an array of max sizes (line 48). If multiple arugments are passed into the `ps` command, then a variety of table sizes is displayed. The table is filled by calling `getprocs ()` (lines 40, 62).

PS Test Program

A new user command `testps` was implemented in the file `testps.c`. The command is intended to be ran in the background while providing multiple, slow processes to test the `ps` user command. The command invokes multiple `fork()` system calls as well a slowly spinning loop (Lines 21 – 22). The test does not print any information to the console.

Time Command

The `time` user command was implemented in `time.c`. This command displays the runtime of process. The command invokes the `uptime()` system call to determine the value of the global variable `ticks` when a process starts (lines 19, 26). The command then invokes two more system calls `fork()` and `exec()` to create a child process (line 27 – 30). The arguments of the `time` command are passed as the path and arguments for `exec` within the child process. The parent function invokes `wait()` (line 31). The runtime is printed to the console in seconds as an elapsed time from process creation (line 44 – 56). The `time` command can run with multiple `time` commands as arguments or no arguments; error messages are displayed in the event of a system call failing (though the time to run is still displayed).

Control–p Modifications

The control–p console command prints debugging information to the console. The following modifications were made to capture and display .

- `procdump()`. This routine in `proc.c` was modified to:
  - Print a new header (line 545) to the console.
  - Format the CPU time to be displayed in seconds (line 592 – 601). This section calculates time as seconds and thousandths of a second since the granularity of the `cpu_ticks_total` variable is at thousandths of a second.
  - Include the UID, GID, PPID, and CPU time in the display of process information on the console (lines 578 – 585).

Testing

UID/GID Test Program

The test was be split into two phases but both with use the same figure. My first test showed that set/get UID and GID as well as get PPID all behaved correctly. I expected the output to show a successful change of UID or GID to a valid number. I also expected the PPID to correctly be the PID of the shell from which it was called (this will be verified by a control–p before the test).
Here is the output for the first test:

```
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       1.465   0.350   sleep   12288
2       sh      0       0       1       1.439   0.189   sleep   16384
testuidgid
Current UID is: 0
Setting UID to to a negative number
SUCCESS: setuid call failed for incorrect UID
Current UID should be unchanged. Current UID is: 0
Setting UID to a number > 32767
SUCCESS: setuid call failed for incorrect UID
Current UID should be unchanged. Current UID is: 0
Setting UID to a valid number (42)
SUCCESS: setuid call returned for valid UID
Current UID should be changed to 42. Current UID is: 42
Setting UID to a valid number (32767)
SUCCESS: setgid call returned for valid GID
Current GID should be changed to 32767. Current GID is: 32767
Setting UID to a valid number (0)
SUCCESS: setgid call returned for valid GID
Current GID should be changed to 0. Current GID is: 0
Current GID is: 0
Setting GID to to a negative number
SUCCESS: setgid call failed for incorrect GID
Current GID should be unchanged. Current GID is: 0
Setting GID to a number > 32767
SUCCESS: setgid call failed for incorrect GID
Current GID should be unchanged. Current GID is: 0
Setting GID to a valid number (42)
SUCCESS: setgid call returned for valid GID
Setting GID to a valid number (32767)
SUCCESS: setgid call returned for valid GID
Current GID should be changed to 32767. Current GID is: 32767
Setting GID to a valid number (0)
SUCCESS: setgid call returned for valid GID
Current GID should be changed to 0. Current GID is: 0
The parent process should be that of the shell. My parent process is: 2
All tests Succeeded!
$
```

Figure 1: UID/GID Test Program

The `testuidgid` output shows UID/GID being set to some arbitrary, correct value, followed by a get which shows the same value. I also tested the set/get for the boundary values of 0 amd 32767 making sure they are valid and successfully changed as well. The output shows the PPID of the `testuidgid()` process as 2 which corresponds to the PID of the shell.
This sub−test `PASSES`.

For the second test, I continued to examine the same output and looked at sections attempting to set

the UID and GID to invalid numbers. I expect the test to not set the UID or GID to an incorrect value. The output for the second test is the same as above. The output in this case indicates that the test was successful and did not set the UID/GID to a number outside of the allowed boundaries. Also note that the UID/GID remains unchanged in the case of a failed set.
This sub−test `PASSES`.
Since all sub−tests passed, this test `PASSES`.

Control–P

For this test, I compared the output of `control−p` when the `CS333_P2` flag was off to the new `control−p` output when the `CS333_P2` flag is on. I expected the output to include columns not found in the previous version of `control−p` such as UID, GID, PPID, and CPU Time.
Here is the output for `control−p` with `CS333_P2` off:

```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
PID     State   Name    Elapsed  PCs
1       sleep   init    0.166    80104da9 80104b33 80106541 80105736 80106935 80106730
2       sleep   sh      0.140    80104da9 80100a09 80101f3e 80101209 801058fb 80105736 80106935 80106730
```

Figure 2: Control–p Test 1

Now, here is the `control−p` output with `CS333_P2` on:

```
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size    PCs
1       init    0       0       1       1.253   0.170   sleep   12288   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
2       sh      0       0       1       1.228   0.189   sleep   16384   80104e56 80100a05 80101f3f 80101205 80105d3c 80105b7f 80106e7f 80106c7a
```

Figure 3: Control–p Test 2

When the `CS333_P2` flag is on, the output included the new columns not found in the output with `CS333_P2` turned off.
This test `PASSES`.

PS Command

I verified the output of the `ps` command by comparing the output to that of `control−p`. I expected the two results to be very similar minus two discrepencies: `control−p` will not show a `ps` process running since it was not running when `control−p` was pressed, and there will be slight differnce in `sh` elapsed and CPU time since there is a delay in typing and typing makes `sh` and *active* process. This test is necessary since `ps` will be used to test other features.
Here is the output of `control−p` and `ps`:

```
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       180.719 0.350   sleep   12288
2       sh      0       0       1       180.693 0.275   sleep   16384
ps
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       181.925 0.350   sleep   12288
2       sh      0       0       1       181.899 0.350   sleep   16384
4       ps      0       0       2       0.032   0.054   run     45056
$
```

Figure 4: PS Test

As expected, there were the aforementioned minor discrepencies; otherwise, the outputs contain the same information.
This test `PASSES`.

Built--in Shell Commands to Set UID/GID

This test was broken into two phases which use the same figure. First I tested that the built–in commands correctly set and then get the UID and GID of the shell, then I tested that child processes inherited the new UID and GID values by using the previously verified `ps` command. I expected the first test to simply set the UID and GID to some arbotrarily chosen values using the built–in `_set` and then verified the set by getting the UID and GID using `_get`.
Here is the output of the test, with the first half being relavent for the first test:

```
$ _set uid 42
$ _set gid 23
$ _get uid
42
$ _get gid
23
$ ps
PID     Name    UID     GID     PPID
1       init    0       0       1
2       sh      42      23      1
5       ps      42      23      2
$
```

Figure 5: Built–in Test

The outputs of the `_get uid` and `_get gid` both matched the values used to set the UID and GID. Thus, this sub−test `PASSES`.

For the second test, I called the `ps` user command after the previous `_set` and `_get` commands to test that child processes inherit UIDs and GIDs. I expected the UID and GID or the child process to match the `sh` process' UID and GID.
Refer to the previous figure's bottom half for this test's output. The output shows both the `sh` and `ps` processes having the same UID and GID values.
This sub−test `PASSES`.
Since both sub−tests passed, this test `PASSES`.

Getprocs System Call

I tested the newly added `getprocs()` system call with 64 actively running processes in two phases to verify the correctness of the new system call. I first tested the correct output with `MAX` set to the values 1, 16, 64, 72 by calling `ps` with an additional argument which tests all four cases. I expected the output for `MAX` set to 1 to be only the first process `init`; the output for `MAX` set to 16 to be the first 16 process; all 64 process for `MAX` set to 64; all 64 processes and no extra with `MAX` set to 72. The 64 active processes were mostly `sh` with the two execeptions being the first, `init`, and the last `ps`.
Here is the output for the first test:

```
**** MAX: 1
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       429.208 0.350   sleep   12288

**** MAX: 16
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       429.213 0.350   sleep   12288
2       sh      42      23      1       429.187 0.740   sleep   16384
6       sh      42      23      2       145.049 0.252   sleep   16384
7       sh      42      23      6       143.842 0.020   sleep   16384
8       sh      42      23      7       143.133 0.020   sleep   16384
9       sh      42      23      8       142.496 0.009   sleep   16384
10      sh      42      23      9       141.892 0.014   sleep   16384
11      sh      42      23      10      141.314 0.014   sleep   16384
12      sh      42      23      11      140.732 0.009   sleep   16384
13      sh      42      23      12      140.162 0.014   sleep   16384
14      sh      42      23      13      139.610 0.014   sleep   16384
15      sh      42      23      14      139.050 0.014   sleep   16384
16      sh      42      23      15      138.570 0.014   sleep   16384
17      sh      42      23      16      138.092 0.054   sleep   16384
19      sh      42      23      17      134.587 0.009   sleep   16384
20      sh      42      23      19      133.530 0.020   sleep   16384

**** MAX: 64
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       429.254 0.350   sleep   12288
2       sh      42      23      1       429.228 0.740   sleep   16384
6       sh      42      23      2       145.090 0.252   sleep   16384
7       sh      42      23      6       143.883 0.020   sleep   16384
8       sh      42      23      7       143.174 0.020   sleep   16384
9       sh      42      23      8       142.537 0.009   sleep   16384
10      sh      42      23      9       141.933 0.014   sleep   16384
11      sh      42      23      10      141.355 0.014   sleep   16384
12      sh      42      23      11      140.773 0.009   sleep   16384
13      sh      42      23      12      140.203 0.014   sleep   16384
14      sh      42      23      13      139.651 0.014   sleep   16384
15      sh      42      23      14      139.091 0.014   sleep   16384
16      sh      42      23      15      138.611 0.014   sleep   16384
17      sh      42      23      16      138.133 0.054   sleep   16384
19      sh      42      23      17      134.628 0.009   sleep   16384
20      sh      42      23      19      133.571 0.020   sleep   16384
21      sh      42      23      20      132.718 0.014   sleep   16384
22      sh      42      23      21      132.114 0.014   sleep   16384
23      sh      42      23      22      131.580 0.020   sleep   16384
```

Figure 6: Testing MAX Values of 1, 16, 64, and 72 with 64 Processes

The output shows that for `MAX` being 1, only the `init` process was included in the print of the `uproc` table; for 16, only the first 16; for 64 all active processes are displayed; and the same for 72 with no extras. Note that the PIDs are not exactly 1−64, this is due to other processes being made during the 62 shells. This sub−test `PASSES`.

For the second test, I used `control−p` as a comparison between `ps` with 64 active process to verify that the information in the `uproc` table is correct. I expected the outputs to be comprable outside of a slight time delay and `control−p` not showing `ps` as an active proc.
The outputs are here:

```
$ ps
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       60.669  0.170   sleep   12288
2       sh      0       0       1       60.644  0.252   sleep   16384
3       sh      0       0       2       48.631  0.014   sleep   16384
4       sh      0       0       3       47.800  0.014   sleep   16384
5       sh      0       0       4       47.083  0.014   sleep   16384
6       sh      0       0       5       45.662  0.014   sleep   16384
7       sh      0       0       6       44.995  0.009   sleep   16384
8       sh      0       0       7       44.318  0.027   sleep   16384
9       sh      0       0       8       43.703  0.014   sleep   16384
10      sh      0       0       9       43.103  0.014   sleep   16384
11      sh      0       0       10      42.513  0.009   sleep   16384
12      sh      0       0       11      41.905  0.020   sleep   16384
13      sh      0       0       12      41.300  0.009   sleep   16384
14      sh      0       0       13      40.717  0.020   sleep   16384
15      sh      0       0       14      40.141  0.014   sleep   16384
16      sh      0       0       15      39.543  0.014   sleep   16384
17      sh      0       0       16      38.944  0.014   sleep   16384
18      sh      0       0       17      38.351  0.014   sleep   16384
19      sh      0       0       18      37.753  0.014   sleep   16384
20      sh      0       0       19      37.163  0.014   sleep   16384
21      sh      0       0       20      36.564  0.014   sleep   16384
22      sh      0       0       21      35.933  0.014   sleep   16384
23      sh      0       0       22      35.262  0.014   sleep   16384
24      sh      0       0       23      34.591  0.014   sleep   16384
25      sh      0       0       24      33.979  0.014   sleep   16384
26      sh      0       0       25      33.302  0.009   sleep   16384
27      sh      0       0       26      32.646  0.020   sleep   16384
28      sh      0       0       27      31.982  0.014   sleep   16384
29      sh      0       0       28      31.302  0.014   sleep   16384
30      sh      0       0       29      30.632  0.009   sleep   16384
31      sh      0       0       30      29.969  0.014   sleep   16384
32      sh      0       0       31      29.090  0.020   sleep   16384
33      sh      0       0       32      27.922  0.014   sleep   16384
34      sh      0       0       33      27.258  0.014   sleep   16384
35      sh      0       0       34      26.436  0.020   sleep   16384
36      sh      0       0       35      25.784  0.014   sleep   16384
37      sh      0       0       36      24.784  0.014   sleep   16384
38      sh      0       0       37      23.922  0.014   sleep   16384
39      sh      0       0       38      23.123  0.044   sleep   16384
41      sh      0       0       39      21.575  0.014   sleep   16384
42      sh      0       0       41      20.705  0.014   sleep   16384
43      sh      0       0       42      19.794  0.014   sleep   16384
44      sh      0       0       43      18.890  0.020   sleep   16384
45      sh      0       0       44      17.861  0.020   sleep   16384
46      sh      0       0       45      16.990  0.020   sleep   16384
47      sh      0       0       46      16.262  0.009   sleep   16384
48      sh      0       0       47      15.537  0.020   sleep   16384
49      sh      0       0       48      14.822  0.014   sleep   16384
50      sh      0       0       49      14.154  0.014   sleep   16384
51      sh      0       0       50      13.459  0.014   sleep   16384
52      sh      0       0       51      12.358  0.020   sleep   16384
53      sh      0       0       52      8.947   0.014   sleep   16384
54      sh      0       0       53      8.214   0.020   sleep   16384
55      sh      0       0       54      7.311   0.014   sleep   16384
56      sh      0       0       55      6.596   0.014   sleep   16384
57      sh      0       0       56      5.773   0.014   sleep   16384
58      sh      0       0       57      5.125   0.014   sleep   16384
59      sh      0       0       58      4.265   0.014   sleep   16384
60      sh      0       0       59      3.552   0.014   sleep   16384
61      sh      0       0       60      2.778   0.020   sleep   16384
62      sh      0       0       61      2.013   0.009   sleep   16384
63      sh      0       0       62      1.326   0.020   sleep   16384
64      ps      0       0       63      0.032   0.065   run     45056
```

Figure 7: PS for 64 Active Processes Test

```
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size    PCs
1       init    0       0       1       125.844 0.170   sleep   12288   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
2       sh      0       0       1       125.819 0.252   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
3       sh      0       0       2       113.806 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
4       sh      0       0       3       112.975 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
5       sh      0       0       4       112.258 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
6       sh      0       0       5       110.837 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
7       sh      0       0       6       110.170 0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
8       sh      0       0       7       109.493 0.027   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
9       sh      0       0       8       108.878 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
10      sh      0       0       9       108.278 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
11      sh      0       0       10      107.688 0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
12      sh      0       0       11      107.080 0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
13      sh      0       0       12      106.475 0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
14      sh      0       0       13      105.892 0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
15      sh      0       0       14      105.316 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
16      sh      0       0       15      104.718 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
17      sh      0       0       16      104.119 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
18      sh      0       0       17      103.526 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
19      sh      0       0       18      102.928 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
20      sh      0       0       19      102.338 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
21      sh      0       0       20      101.739 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
22      sh      0       0       21      101.108 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
23      sh      0       0       22      100.437 0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
24      sh      0       0       23      99.766  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
25      sh      0       0       24      99.154  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
26      sh      0       0       25      98.477  0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
27      sh      0       0       26      97.821  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
28      sh      0       0       27      97.157  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
29      sh      0       0       28      96.477  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
30      sh      0       0       29      95.807  0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
31      sh      0       0       30      95.144  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
32      sh      0       0       31      94.265  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
33      sh      0       0       32      93.097  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
34      sh      0       0       33      92.433  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
35      sh      0       0       34      91.611  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
36      sh      0       0       35      90.959  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
37      sh      0       0       36      89.959  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
38      sh      0       0       37      89.097  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
39      sh      0       0       38      88.298  0.044   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
41      sh      0       0       39      86.750  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
42      sh      0       0       41      85.880  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
43      sh      0       0       42      84.969  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
44      sh      0       0       43      84.065  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
45      sh      0       0       44      83.036  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
46      sh      0       0       45      82.165  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
47      sh      0       0       46      81.437  0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
48      sh      0       0       47      80.712  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
49      sh      0       0       48      79.997  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
50      sh      0       0       49      79.329  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
51      sh      0       0       50      78.634  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
52      sh      0       0       51      77.533  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
53      sh      0       0       52      74.122  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
54      sh      0       0       53      73.389  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
55      sh      0       0       54      72.486  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
56      sh      0       0       55      71.771  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
57      sh      0       0       56      70.948  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
58      sh      0       0       57      70.300  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
59      sh      0       0       58      69.440  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
60      sh      0       0       59      68.727  0.014   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
61      sh      0       0       60      67.953  0.020   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
62      sh      0       0       61      67.188  0.009   sleep   16384   80104e56 80104b99 8010697f 80105b7f 80106e7f 80106c7a
63      sh      0       0       62      66.501  0.027   sleep   16384   80104e56 80100a05 80101f3f 80101205 80105d3c 80105b7f
```

Figure 8: Control–p for 64 Active Processes Test

The outputs show that both `ps` and `control−p` produce similar outputs.
Thus, the sub−test `PASSES`.
Since all sub−tests passed, this test `PASSES`.

Elapsed CPU Time

I used `control−p` to test that the correct elapsed CPU time is being calculated by using `control−p` multiple times. In each `control−p`, I expected the CPU time for `init` to remain unchanged since `init` should remain asleep during this test and the value should have been smaller than `sh`. By typing in the shell, I expected the `sh` CPU time to change by small amounts. The chnage in CPU time would never be greater than the elapsed time.
The output for this test is here:

```
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       47.141  0.170   sleep   12288
2       sh      0       0       1       47.118  0.252   sleep   16384
ps
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       49.241  0.170   sleep   12288
2       sh      0       0       1       49.218  0.324   sleep   16384
4       ps      0       0       2       0.030   0.005   run     45056
$
PID     Name    UID     GID     PPID    Elapsed CPU     State   Size
1       init    0       0       1       50.729  0.170   sleep   12288
2       sh      0       0       1       50.706  0.350   sleep   16384
```

Figure 9: Elapsed CPU Time Test

The output showed a consistent, small value for the CPU time of `init`. The output also shows small changes in CPU time for `sh` from an intermediate `ps`. That change in CPU time is less than the change in elapsed time.
Thus, this test `PASSES`.

Time Command

I tested the new user command `time` in three phases. First, I tested that `time` called with no arguments and an invalid argument; next, I tested `time` with a valid command argument `echo` that also had an argument, `qed`; the third test checked the validity of the calculated time by using `control−p` at the start of a long process and `control−p` after the `time` process finished. I expected that for the first test, a command of `time` with no arguments will determine the time to run time `time`.
For a call with an invalid argument, an error message will be displayed follwed by the time to run the failed command i.e. the time to display the error message.
Here are the two outputs for the first test:

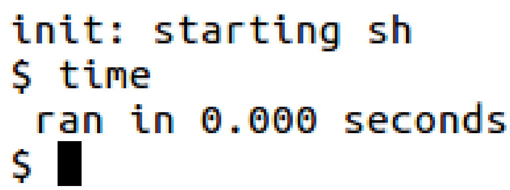

Figure 10: Time Test with no Arguments

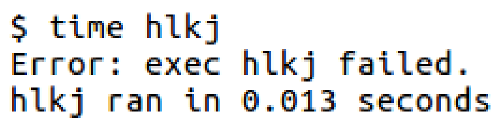

Figure 11: Time Test with Invalid Arguments

The outputs correctly display the explained expected results.
Thus this sub−test `PASSES`.
For the second test, I expected the valid command argument to first be executed, in this case it would display qed to the console, followed by the time in seconds to run the processs.
Here is the output for the second test:

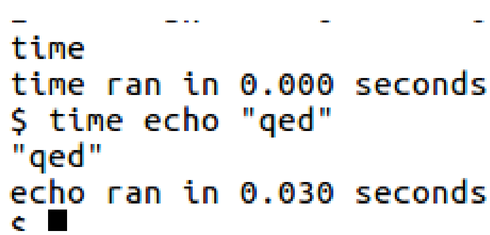

Figure 12: Time Test with Echo qed

The output first shows the results of the echo command followed by the calculated runtime.
This sub−test also `PASSES`.

For the third and final sub−test, I expected the difference in elapsed time between the first and second `control−p` to be approximately the same as the printed result from `time`.
Here is the output for this test:
The difference in elapsed time for the two `control−p` displays is around 6 seconds. The calculated time

```
time testps

PID     Name    UID     GID     PPID    Elapsed CPU
1       init    0       0       1       53.251  0.170
2       sh      0       0       1       53.231  0.324
7       time    0       0       2       0.459   0.009
8       testps  0       0       7       0.430   0.005
9       testps  0       0       8       0.408   0.860
10      testps  0       0       9       0.397   0.819
testps ran in 6.023 seconds
$
PID     Name    UID     GID     PPID    Elapsed CPU
1       init    0       0       1       59.339  0.170
2       sh      0       0       1       59.319  0.350
```

Figure 13: Elapsed Time Test

to run the `testps` command is also 6 seconds. Note, the discrepency is due to the delay of my subpar clickspeed.
This sub−test `PASSES`.
Since all three sub−tests passed, this test `PASSES`.