Project Two Report
Introduction to Operating Systems
Spring 2017

Andy Keene

Description

For this assignment, I learned about locks used for concurrent data structures in the kernel; expanded the process structure to contain ownership information and CPU use-time; implemented new user commands to see current process statuses, and track the run-time of user level commands; and how to use wrappers for system calls that effect files or access the proc table data structure .

Deliverables

The following features were added to xv6:

- New system calls to support process ownership and retrieve active process information
  - setuid sets the UID of the current process to *value*
    ```
    int
    setuid(uint value);
    ```
  - setgid sets the GID of the current processes to *value*
    ```
    int
    setgid(uint value);
    ```
  - getuid returns the UID of the current process
    ```
    int
    getuid(void);
    ```
  - getgid returns the current GID of the current process
    ```
    int
    getgid(void);
    ```
  - getppid returns the PPID (parent process ID) of the current process
    ```
    int
    getppid(void);
    ```
  - getprocs fills in the provided `uproc` structure with information of the currently active processes up to a maximum number of the given `max` and returns the number of processes placed in the table.
    ```
    int
    getprocs(uint max, struct uproc *table);
    ```
- Process total CPU time. Each process now stores `cpu_ticks_in` (the most recent time it started running) and `cpu_ticks_total` (the total time spent in the CPU) so that the process's total time running can be tracked and calculated.
- A new user command, `ps`, that prints the current statues of active processes to the standard output.
- A new user command, `time`, that will time the execution of commands.
- A modification of the `ctrl-p` console command to include the `uid`, `gid`, `ppid`, and total execution time information of each currently running process.

Implementation

New System Calls For Process

Using the process outlined in project one, the following system calls were implemented by adding: a user-side header in `user.h`; creating a system call number in `syscall.h`; updating the system call jump table, the system call name table, and adding a kernel side header in `syscall.c`; a user-side stub in `usys.S`; and implementation in `sysproc.c`. To support process ownership new fields `uint uid` and `uint gid` were added to the `proc` structure definition in proc.h (lines 61-62). INITUID and INITGID were defined in param.h (lines 15-16) with which to initialize these fields for the `init` process in proc.c (lines 108-109). All other processes inherit their UID, GID from their parent process in fork (proc.c lines 169-170). Since the process stores a pointer to its parent, any PPIDs will be calculated on the fly. Each new system call's description and it's corresponding file changes are as follows:

- The `setuid()` system call sets the process's UID to the given *value*. This *value* is retrieved from the stack as an `int` using `argint()` before being cast to an `uint`, where $0 \leq value \leq 32767$. If `setuid()` fails to retrieve the argument from the stack or if the given *value* is out of bounds then -1 is returned while upon success 0 is returned. The files modified to support this system call are as follows:
  - user.h (line 35)
  - usys.S (line 37)
  - syscall.h (line 29)
  - syscall.c (lines 106, 138, 172)
  - sysproc.c (lines 129-142)

  The function prototype is:

  ```
  int
  setuid(uint value);
  ```

- The `setgid()` system call sets the process's GID to the given *value*. This *value* is retrieved from the stack as an `int` using `argint()` before being cast to an `uint`, where $0 \leq value \leq 32767$ (it was passed from the user side as a `uint` so this conversion is safe). If `setgid()` fails to retrieve the argument from the stack or if the given *value* is out of bounds then -1 is returned while upon success 0 is returned. The files modified to support this system call are as follows:
  - user.h (line 36)
  - usys.S (line 38)
  - syscall.h (line 30)
  - syscall.c (lines 107, 139, 173)
  - sysproc.c (lines 144-157)

  The function prototype is:

  ```
  int
  setgid(uint value);
  ```

- The `getuid()` system call returns the process's UID. This system call cannot fail and will simply return whatever value is stored as UID in the calling process as a `uint`. If this value is out of bounds it would imply that the `setuid()` system call or the defined initial values are incorrect. The files modified to support this system call are as follows:

- user.h (line 32)
- usys.S (line 34)
- syscall.h (line 26)
- syscall.c (lines103, 135, 169)
- sysproc.c (lines 110-114)

The function prototype is:

```
uint
getuid(void);
```

- The `getgid()` system call returns the process's GID. This system call cannot fail and will simply return whatever value is stored in the GID field of the calling process as a `uint`. The files modified to support this system call are as follows:

  - user.h (line 33)
  - usys.S (line 35)
  - syscall.h (line 27)
  - syscall.c (lines 104, 136, 170)
  - sysproc.c (lines 116-120)

  The function prototype is:

```
uint
getgid(void);
```

- The `getppid()` system call returns the process's PID (parent ID). This system call also cannot fail and will simply return the `pid` field of the parent process as a `uint`. Note that the initial process, `init`, is considered it's own parent (special case). The files modified to support this system call are as follows:

  - user.h (line 34)
  - usys.S (line 36)
  - syscall.h (line 28)
  - syscall.c (lines 105, 137, 171)
  - sysproc.c (lines 122-127)

  The function prototype is:

```
uint
getppid(void);
```

- The `getprocs()` system call fills in the `uproc` table array given as a parameter for *at most* the given `max` number of processes. The user, or calling process, is responsible for correctly allocating an array of `uproc` structures and passing in a valid maximum value (i.e. a valid index). `getprocs()` uses the helper functions `getptr()` to retrieve the pointer to the `uproc` table from the stack, and `getint()` to retrieve the `max` argument. If `getprocs` fails to retrieve the arguments from the stack then -1 is returned where a return value greater than or equal to 0 indicates how many individual process statuses were placed in the table. Additionally since `getprocs` must access the `ptable` data structure, the system call `getprocs()` in sysproc.c acts as a wrapper for the `getprocs` function defined in proc.c.

The files modified to support this system call are as follows:

- user.h added the struct uproc type declaration (line 3), and the system call function header (line 37)
- usys.S (line 39)
- syscall.h (line 31)
- syscall.c (lines 108, 140, 174)
- sysproc.c defines the wrapper system call function (lines 159-172) which must include `uproc.h` (line 9) to use the given `uproc` pointer.
- proc.c defines the core function of the `getproc()` system call (lines 559-591). Here the `ptable` is accessed, using the given spin lock, and the `states` array is used to define the string corresponding to process state; process information is copied into the `uproc` table accordingly.
- uproc.h file was added and defines the `uproc struct` which is also outlined below
- defs.h added the `uproc` type declaration (line 11) and `getprocs` function prototype (line 122) implemented in proc.c so that it may be called from sysproc.c.

The function prototype is:

```
int
getprocs(uint max, struct uproc *table);
```

The uproc structure (with `STRMAX` defined as 32) is:

```
struct uproc {
    uint pid;
    uint uid;
    uint gid;
    uint ppid;
    uint elapsed_ticks;
    uint cpu_total_ticks;
    uint size;
    char state[STRMAX];
    char name[STRMAX];
};
```

Process Execution Time

The following files were modified to support tracking the execution time of a process:

- `cpu_ticks_total` and `cpu_ticks_total` were added to the `proc` structure defined in proc.h (lines 72-73) to track the most recent time the process was scheduled to run in the CPU, and to count the total time running on the CPU respectively.
- These fields are updated, before and after the process runs, in the scheduler and de-scheduler routines in proc.c (lines 320, 363). Each field is set to 0 during `allocproc` (lines 70-71).

New User Commands

- The new user command `time` was added to calculate and display the time elapsed to run the command that follows it. For example `time echo "abc"` will allow `echo` to execute and then display the time it took to do so. The file time.c was added and contains the program to run this user command. The `time` user command was also added to the Makefile (line 158).

- The new user command `ps` was added to display currently active process information to standard-out using the new `getprocs()` system call. It is written to display up to a maximum of 64 processes. The file ps.c was added and contains the program to run for this user command. The `ps` user command was also added to the Makefile (line 157).

- Additionally a new user command `test` was added to help automate testing for the new system calls pertaining to UID, GID, and PPID. The program, found in the new test.c file, demonstrates adding valid and invalid UIDs and GIDs as well as testing *if* GIDs and UIDs are correctly inherted from the parent process. The `ps` user command was also added to the Makefile (line 156).

ctrl-p Modifications

The ctrl-p console command prints debugging information to the console. `procdump` was refactored in proc.c to print process information in a prettier fashion using a helper function `printnum` (lines 515-522) – and to include the process UID, GID, PPID, and CPU run-time in its output (lines 538, 546-549). Since `elapsed_time` was implemented in project 1, and `cpu_total_ticks` is already calculated no additional calculation were needed.

Testing

Valid set UID/GID

To test that the `setuid()` and `setgid()` system calls correctly set *valid* numbers we will use the user command `test` to help automate the process of 1) Pause to press ctrl-p and see the current UID/GID of the `test` process 2) Print the number's x, where x is in the valid range for the UID and GID fields (11 and 50 will be used) 3) Call `setuid()/setgid()` with x 4) Pause before exiting to allow the user to press `ctrl-p` and see what number the UID/GID is set to for `test` user process in the ptable and 4) print the return code of the system call. For both `setuid()` and `setgid()` both a return code of 0 and an output of `ctrl-p` displaying the number x for the respective UID/GID field is expected.

```
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
$ test

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       3.35    0.04    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       3.30    0.04    sleep   16384   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
4       test    0       0       2       1.16    0.00    sleep   12288   80104e52 8010690a 80105a31 80106d50 80106b4b

Setting GID to 50...
SUCCESS: GID is: 50, Return code: 0
Setting UID to 11...
SUCCESS: UID is: 11, Return code: 0

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       8.75    0.04    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       8.70    0.04    sleep   16384   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
4       test    11      50      2       6.56    0.03    sleep   12288   80104e52 8010690a 80105a31 80106d50 80106b4b
$
$ $
```

Figure 1: valid setuid() and setgid()

Because the initial UID/GID shown by ctrl-p were shown as 0, `setgid()` was called with 50, `setuid()` was called with 11, and ctrl-p shows a GID of 50 and UID of 11 for test immediately after with return codes of 0 (success) for each call, this test **PASSES**.

Invalid set UID/GID

To test that the `setuid()` and `setgid()` system calls correctly error on *invalid* numbers we will use the user command `test` to automate the process of 1) printing the current UID/GID, 2) print the number 38000, where 38000 is *not* in the valid range for the UID/GID fields 3) call `setuid()/setgid()` with 38000 as the argument4) pause, allowing the user to press `ctrl-p` and see what number the UID/GID is set to for `test` user process in the ptable and 4) print the return code of the system call. For both `setuid()/setgid()` the GID/UID field is expected to stay the same, and a return an error code of -1.

```
$ test
Setting GID to 38000...

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       2.68    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       2.62    0.03    sleep   16384   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
3       test    0       0       2       0.89    0.02    sleep   16384   80104e52 8010690a 80105a31 80106d50 80106b4b
FAIL: UID is: 0, Return code: -1
Setting UID to 38000...

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       5.07    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       5.01    0.03    sleep   16384   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
3       test    0       0       2       3.28    0.02    sleep   16384   80104e52 8010690a 80105a31 80106d50 80106b4b
FAIL: UID is: 0, Return code: -1
```

Figure 2: invalid setuid() and setgid()

Because `setgid()` and `setuid()` were called with 38000, and ctrl-p shows the original GID/UID (unchanged) for test immediately after, and each call returned an error code of -1 this test **PASSES**.

getuid(), getgid() and getppid()

To test that `getuid()`, `getgid()`, `getppid()` are working correctly each system call will be invoked in a function *within* the `test` user command. The result from each call, the IDs, will be printed to standard-out, and compared against the output from ctrl-p. We expect the printed UID, GID, and PPID printed from `test` to match the the information of the `test` process in ctrl-p's output (`test` is the process that these system calls are evoked from). We also expect that the PPID of `test` will match the PID of the shell process; because `test` is forked from `sh` (the shell process) and thus, `sh` is its parent.

```
init: starting sh
$ test
(test) PPID: 2, UID: 0, GID:0

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       4.48    0.04    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       4.42    0.03    sleep   16384   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
3       test    0       0       2       0.71    0.01    sleep   16384   80104e52 8010690a 80105a31 80106d50 80106b4b
```

Figure 3: invalid setuid() and setgid()

In the figure above we can see that the output line of `test` marked with "(test)" has a UID of 0, GID of 0, and PPID of 2. This matches the `test` process information printed in ctrl-p's output immediately after. We also see that the PID of `sh` is 2, which matches the PPID of `test`. Because the `..get()` system call return values match the process information output in ctrl-p and because the PPID of `test` matched the PID of `sh`, this test **PASSES**.

Built in shell commands for UID and GID

Next we will test whether the built in shell commands `_get uid`, `_get gid`, `_set uid int`, and `_set gid int` work correctly. To enable these commands for the shell we first must turn off the `DUSE_BUILTINS` flag in the Makefile (line 74). Next, since these system calls are invoked from the shell process, we will perform the following tests:

Built in shell commands for UID

- 1) ctrl-p to see the shell's starting UID
- 2) Call `_get uid` to verify this built in is working correctly
- 3) Call `_set uid` with a value 56
- 4) Call `_get uid` to see the *now current* UID value
- 5) Verify that the UID for `sh` did change by pressing ctrl-p

When performing this test we expect to see: that the first call to `_get uid` matches the UID of `sh` in ctrl-p's output; a return value of 0 for `_set uid` 56; and a subsequent `_get uid` return value of 56 to match the second output of ctrl-p.

```
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       1.38    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       1.32    0.01    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b

$ _get uid
0
$ _set uid 56
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       34.96   0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      56      0       1       34.90   0.03    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b

$ _getuid
56
$
```

Figure 4: get/setuid (sh)

Because the output of the test matches our expected output, a matching UID to ctrl-p's information, and a subsequent change to the new value where `_get uid` and ctrl-p match again, this subtest **PASSES**.

Built in shell commands for GID

- 1) ctrl-p to see the shell's starting GID
- 2) Call `_get gid` to verify this built in is working correctly
- 3) Call `_set gid` with a value 71
- 4) Call `_get gid` to see the *now current* GID value
- 5) Verify that the GID for `sh` did change by pressing ctrl-p

When performing this test we expect to see: that the first call to `_get uid` matches the GID of `sh` in ctrl-p's output 1^st output; and a subsequent `_get uid` return value of 71 to match the second output of ctrl-p (after having set the GID to 71).

Because the output of the test matches our expected output, a matching GID to ctrl-p's information, and a subsequent change to the new value where `_get gid` and ctrl-p match again, this subtest **PASSES**.

Because both subtests **PASS**, this test **PASSES**.

```
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       3.97    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       3.91    0.01    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b

$ _get gid
0
$ _set gid 71
$ _get gid
71
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       20.43   0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       71      1       20.37   0.03    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b
```

Figure 5: get/setgid (sh)

Fork() and ID inheritance

To test that the GID and UID are properly inherited from the parent process we will write a program that sets its UID and GID, prints it, then forks and has the child process print *its* UID and GID. We expect that the UID and GID will match the parent process's UID and GID. Although test getuid(), getgid() and getppid() already demonstrated that getppid() behaves as expected, we will again have the parent process prints its PID, and the child its PPID; we expect these to match.

```
init: starting sh
$ test
(parent) Setting UID to 191, GID to 67
(parent) Before fork... UID: 191, GID: 67, PID: 3
(child) UID: 191, GID: 67, PID: 4, PPID:3

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       3.80    0.04    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       3.74    0.02    sleep   16384   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
3       test    191     67      2       0.88    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
4       test    191     67      3       0.85    0.00    sleep   12288   80104e52 8010690a 80105a31 80106d50 80106b4b
```

Figure 6: fork test

From the output, it is clear that the parent was successful in setting its UID to 191 and GID to 67 (fork_test() in test.c also checks the return code of each system call). In the subsequent line we also see that the child has the same UID of 191 and GID of 67; this assertion by the child is verified by using ctrl-p's output to examine that both the parent (PID 3) and child (PID 4) have these values set. Additionally, we see that the child's PPID is 3, which is in fact its parent. Thus, this test **PASSES**.

PS command and getprocs() system call

To test that the `ps` user command, and consequently it's `getprocs()` system call dependency, is working correctly we will recompile the program with maximum values of 1, 16, 64, and 72, and have `ps` print its current maximum value. Each time we will run `ps` then immediately use ctrl-p to dump all currently active processes.

We expect that `ps` will print either it's compiled maximum number of processes (1, 16, 64, or 72), *or* the total number of processes active, which ever is least; for example if there are 12 active processes but `ps` has a maximum value of 64, we expect to see 13 processes displayed (12 plus `ps`), while on the other hand if there are 100 processes active and `ps` has a maximum value of 4, we expect to see 4 processes displayed. Further, for the output of `ps` we expect the PID, UID, GID, and PPID fields to match the output in ctrl-p, with the Elapsed Time field being larger in ctrl-p. PCs will not be printed in `ps` and we cannot make any assertions about the CPU field other than if it changes, it must increase. We cannot guarantee that the Size field will not change either.

We also expect that since `ps` gets active processes while it is running, that it (`ps`) itself will be displayed in the table as long as the maximum value it is compiled is larger than than the total number of other active processes. For example with a maximum value of 1 cannot guarantee that `ps` will be the process displayed since there will be more than one process active at that time (i.e. `sh, init`). The results of these runs and the PASS/FAIL status are broken into into the following subtests.

PS command with a maximum of 1

```
$ ps
(ps) max value: 1

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size
1       init    0       0       1       2.74    0.03    sleep   12288
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       3.05    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       2.99    0.04    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b
```

Figure 7: ps-1t

The PID, UID, GID, and PPID fields printed by `ps` match the respective output in ctrl-p; the elapsed time increases between `ps` and ctrl-p; the CPU time does not decrease; and the number of processes printed is one, so the results match our expectations for a maximum value of one. Thus, this subtest **PASSES**.

PS command with a maximum of 16

```
init: starting sh
$ ps
main-loop: WARNING: I/O thread spun for 1000 iterations
(ps) max value: 16

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size
1       init    0       0       1       5.38    0.02    sleep   12288
2       sh      0       0       1       5.33    0.02    sleep   16384
3       ps      0       0       2       0.03    0.02    run     45056
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       6.73    0.02    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       6.68    0.02    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b
```

Figure 8: fps-16

The PID, UID, GID, and PPID fields printed by `ps` match the respective output in ctrl-p; the elapsed time increases between `ps` and ctrl-p; the CPU time does not decrease; the number of processes printed by `ps` is 3 which is the total number active processes *including* `ps` (i.e. one greater than ctrl-p), so the results match our expectations for a maximum value value that is larger than the number of current processes. Thus, this subtest **PASSES**.

PS command with a maximum of 64

```
$
$ ps
(ps) max value: 64

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size
1       init    0       0       1       5.29    0.02    sleep   12288
2       sh      0       0       1       5.23    0.02    sleep   16384
4       ps      0       0       2       0.03    0.02    run     45056
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       6.16    0.02    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       6.10    0.02    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b
```

Figure 9: ps-64

The PID, UID, GID, and PPID fields printed by `ps` match the respective output in ctrl-p; the elapsed time increases between `ps` and ctrl-p; the CPU time does not decrease; the number of processes printed by `ps` is 3 which is the total number active processes *including* `ps` (i.e. one greater than ctrl-p), so the results match our expectations for a maximum value value that is larger than the number of current processes. Thus, this subtest **PASSES**.

PS command with a maximum of 72

```
init: starting sh
$
$ ps
(ps) max value: 72

PID     Name    UID     GID     PPID    ELapsed CPU     State   Size
1       init    0       0       1       3.69    0.02    sleep   12288
2       sh      0       0       1       3.64    0.06    sleep   16384
4       ps      0       0       2       0.02    0.02    run     45056
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       5.79    0.02    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       5.74    0.06    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b
```

Figure 10: ps-72

The PID, UID, GID, and PPID fields printed by `ps` match the respective output in ctrl-p; the elapsed time increases between `ps` and ctrl-p; the CPU time does not decrease; the number of processes printed by `ps` is 3 which is the total number active processes *including* `ps` (i.e. one greater than ctrl-p), so the results match our expectations for a maximum value value that is larger than the number of current processes. Thus, this subtest **PASSES**.

Because all subtests passed, this test **PASSES**,

CPU Time

To test that the CPU time is displaying properly, we will press ctrl-p to identify the CPU time for the shell process `sh`, then we will hit the return key repeatedly forcing `sh` to process null commands and thus increase it's CPU usage. Then we will hit ctrl-p to see its CPU time again. We expect that since the the `sh` processed multiple null commands, and it must do so using the CPU, that it's CPU time will have increased. More so, we expect this increase to be approximate to the *difference* in Elapsed Time field values of ctrl-p. The result is as follows:

```
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       2.95    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       2.90    0.01    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b

$
$
$
$
$
$
$
PID     Name    UID     GID     PPID    ELapsed CPU     State   Size    PCs
1       init    0       0       1       4.66    0.03    sleep   12288   80104e52 80104b90 8010682c 80105a31 80106d50 80106b4b
2       sh      0       0       1       4.61    0.08    sleep   16384   80104e52 80100a05 80101f3f 80101205 80105beb 80105a31 80106d50 80106b4b

$
```

Figure 11: CPU time

As the results illustrate, the CPU time of `sh` did increase by 0.07 seconds between the initial and final ctrl-p outputs. Thus this test **PASSES**.

time command

To test the `time` user command, we will establish subtests for executing; `time`, `time ls`, `time echo abc`, `time time echo time`, and `time badcommand!`. Since expectations differ for each execution subtests will be established.

time (null)

For the execution of `time` we expect the `null` command will be timed, and should complete in *approximately* 0.00 seconds (set up of execution takes non-zero time), so we expect `time` to display that the `null` command executed in such a time.

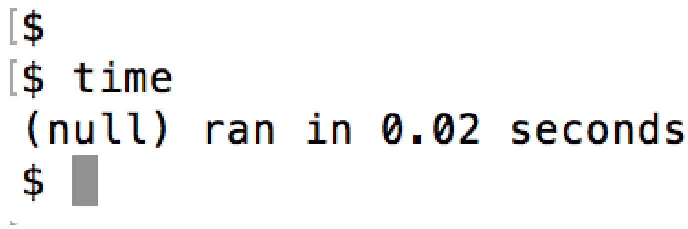

Figure 12: time null

Because `time` printed the command (null) and stated its execution was 0.02 which is approximately 0.00, this subtest **PASSES**.

time ls

For the execution of `time ls` we expect that the execution of `ls` will take non-zero time, within a reasonable range of under 1 second. We also expect that we will see the output of `ls` before we see the output of `time`.

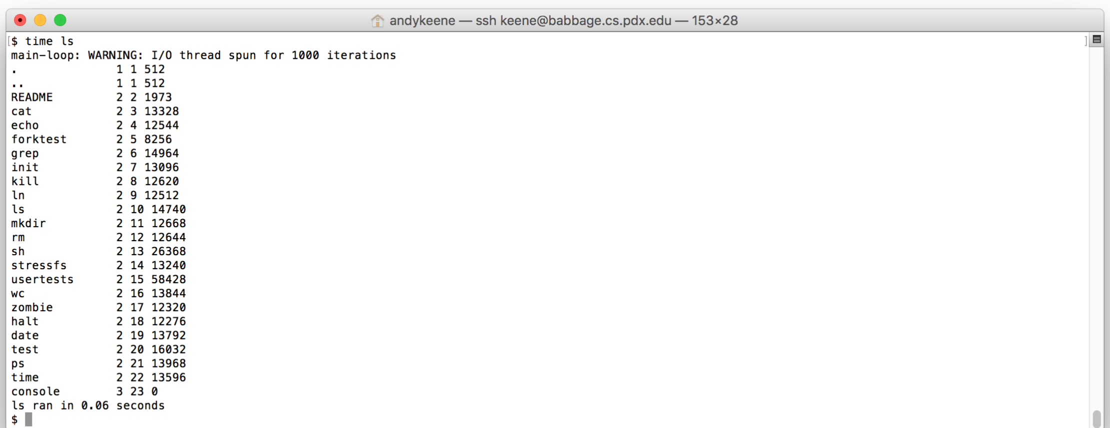

Figure 13: time ls

Because the output of `ls` appears first, followed by `time`s output that ls ran in 0.6 seconds (within reason of 1 second) this subtest **PASSES**.

time echo abc

For the execution of `time echo abc` we expect that the execution of `echo abc` will take non-zero time, within reason. We also expect that we will see the output of `echo abc` before we see the output of `time` since `abc` is the argument to `echo`.

Because the output of `echo abc` appears first (where `abc` was the argument for `echo`, followed by `time`s output that echo ran in 0.03 seconds (within reason) this subtest **PASSES**.

```
         Specify the 'raw' format explicitly to remove the restrictions.
xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
[$
[$
[$
[$ time echo abc
abc
echo ran in 0.03 seconds
$
```

Figure 14: time echo abc

time time echo abc

For the execution of `time time echo abc` we expect that the execution of `echo abc` will print to standard-out. Following `echo`s output we expect to see, like before, that `echo` ran in non-zero time (approximately 0.03 seconds as established) from the *second* `time` command, finally followed by a `time ran in ...` output from the first `time` command timing the second `time`, where this time must be greater than the execution time of `echo abc`.

```
$ time time echo abc
abc
echo ran in 0.02 seconds
time ran in 0.05 seconds
$
```

Figure 15: time time echo abc

Because the output of `echo abc` appears first, followed by a reasonable time of `echo` (0.02 seconds) by the second `time` command and because we see an output of `time ran in 0.05 seconds` which is greater than the execution time of `echo` alone, this subtest **PASSES** and ensures the generality of the `time` command.

time badcommand

To verify `time` will respond to a failure to execute a command and time it, we will test the execution of `time badcommand`. `time` was implemented to throw an error on failure to execute a command, but *not* to exit; because we may still want to see timings from the commands that before the failure. As a result of this implementation choice, we expect to see that when given a bad command (i.e. `badcommand`) that an error will be displayed that the command did not execute, but `time` will still report a time for the execution (the attempted execution) of the bad command before exiting normally. Also, due to our expectations that `badcommand` will fail to execute, we expect an attempted run time zero seconds.

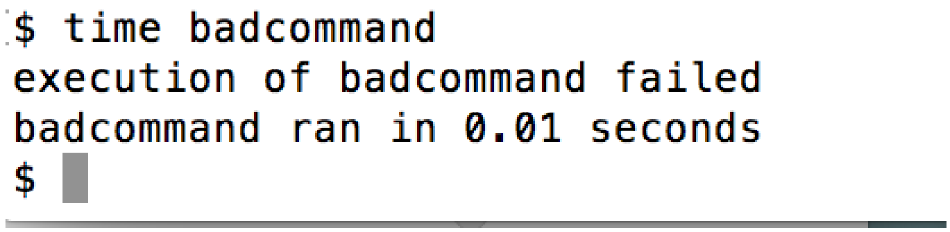

Figure 16: time-badcommand

Because `time badcommand` prompted the user that `badcommand` failed, but still reported a time of 0.01 seconds (near zero seconds) for its attempted execution, this test behaves according to the expectations set by its implementation. Thus, this test **PASSES**.