

Project 2 Report: Rough Draft

Project Description:

The goal of project 2 was to solidify the concept of adding new system calls and new commands to xv6 using the same procedures we familiarized ourselves with during project 1. Adding these new system calls and commands to xv6 gave us the opportunity to gain an understanding of the existing fork, wait, and exec system calls. Adding the new system calls also introduced the concept of process ownership, time spent in the CPU for processes, process execution time, and further solidified the idea of outputting relevant process information.

Project Deliverables:

The following functionality was added to the xv6 operating system:

- New system calls
 1. setuid – Set the UID of a process to the uint passed into the function.
 prototype: int setuid(uint);
 2. setgid – Set the GID of a process to the uint passed into the function.
 Prototype: int setgid(uint);
 3. getuid – Return the UID of a process to the calling routine.
 Prototype: uint getuid(void);
 4. getgid – Return the GID of a process to the calling routine.
 Prototype: uint getgid(void);
 5. getppid – Return the parent process' ID to the calling routine.
 Prototype: uint getppid(void);
 6. getprocs – Populates a table of uproc structures with information regarding currently running processes up to a provided maximum value.
 Prototype: #include "user.h"
 int getprocs(uint max, struct uproc* table);
- Process total CPU time
 Each process now contains a variable to store the start "time" the process entered the CPU and a variable to calculate the elapsed time that the process spent in the CPU based on when it started.
- The ps command
 A new command that prints out information related to processes that are deemed "active" in the system.
- The time commands
 A new command that times the execution of commands that are typed into the xv6 shell.
- The ctrl-p console commands
 Modified the output for the control p console command to include the uid, gid, ppid, elapsed time, and CPU time of a process.

Implementation

- New system calls

All system calls were added using the same process utilized in project one to create a new system call.

Default values for a process uid and gid are defined in param.h (lines 15 and 16). These two values are used for the init process since it is the first process to be created. In the userinit() routine the first process' uid and gid are set to the DEFAULTUID and DEFAULTGID values.

1. setuid() system call

- * usys.S – line 37
- * user.h – line 32
- * syscall.h – line 29
- * syscall.c – lines 106, 137, 171
- * sysproc.c – lines 133 - 143

setuid() is implemented in sysproc.c starting on line 133. It takes a void argument since it is the kernel side implementation and it grabs all necessary arguments off the stack. The argint() routine is used to grab the uint value passed into the user side version of the function from the stack. A check is then performed to make sure that the value is within the range $0 < \text{value} < 32767$. If it is not within that range setuid() returns -1 to indicate failure. If it is within that range setuid() proceeds by assigning that value to the process' uid and then returning 0 as a sign of successful assignment.

2. setgid() system call

- * usys.S – line 38
- * user.h – line 33
- * syscall.h – line 30
- * syscall.c – lines 107, 138, 172
- * sysproc.c – lines 146 – 156

setgid() is implemented in sysproc.c starting on line 146. It takes a void argument since it is a kernel side implementation and it grabs all necessary arguments off the stack. The argint() routine is used to grab the uint value passed into the user side version of the function from the stack. A check is then performed to make sure that the value is within the range $0 < \text{value} < 32767$. If it is not within that range setgid() returns -1 to indicate failure. If it is within that range setgid() proceeds by assigning that value to the process' gid and then returning 0 as a sign of successful assignment.

3. getuid() system call

- * usys.S – line 34
- * user.h – line 29
- * syscall.h – line 26
- * syscall.c – lines 103, 134, 168
- * sysproc.c – lines 111 - 115

getuid() is implemented in sysproc.c starting on line 111. It takes a void argument in both the kernel and user side implementation and its sole duty is to return the process' uid to the calling routine. There is no need to pass any arguments into this function or retrieve anything off the stack since it just returns the desired value.

4. `getgid()` system call

- * `usys.S` – line 35
- * `user.h` – line 30
- * `syscall.h` – line 27
- * `syscall.c` – lines 104, 135, 169
- * `sysproc.c` – lines 118 - 122

`getgid()` is implemented in `sysproc.c` starting on line 118. It takes a void argument in both the kernel and user side implementation and its sole duty is to return the process; `gid` to the calling routine. There is no need to pass any arguments into this function or retrieve anything off the stack since it just returns the desired value.

5. `getppid()` system call

- * `usys.S` – line 36
- * `user.h` – line 31
- * `syscall.h` – line 28
- * `syscall.c` – lines 105, 136, 170
- * `sysproc.c` – lines 125 - 129

`getppid()` is implemented in `sysproc.c` starting on line 125. It takes a void argument in both the kernel and user side implementation and its sole duty is to return the process id of a process' parent process. It uses a ternary expression to test whether or not the process' parent is null or not. If the process' parent isn't null it returns the parents process id. If the process' parent is null that means that it is the first process to have been created. In that instance it returns its own process id.

- Elapsed process time

- * `proc.h` – lines 71 and 72
- * `proc.c` – lines 316 and 358

Two new variables were added to `proc.h` (lines 71 and 72); `uint cpu_ticks_total` and `uint cpu_ticks_in`. In file `proc.c` `cpu_ticks_in` and `cpu_ticks_total` are initialized to 0 in the `allocproc()` routine (lines 75-76). `cpu_ticks_in` is set to the global variable `ticks` in the `scheduler()` (line 316) routine to capture the “time” when the process enters the cpu. In the `proc.c` `sched()` routine (line 358) the `cpu_ticks_total` variable is calculated by subtracting the current ticks value from the `cpu_ticks_in` value which is the value calculated everytime the process enters the cpu. The `cpu_ticks_total` variable represents the total time that the process spent in the cpu.

- Getprocs() system call
 - * usys.S – line 39
 - * user.h – line 34
 - * syscall.h – line 31
 - * syscall.c – lines 108, 139, 173
 - * sysproc.c – lines 159 - 169
 - * proc.c – lines
 - * uproc.h – entire file

The getprocs() system call is implemented in the sysproc.c file starting on line 159. It grabs two arguments off the stack, one an integer using the argint() function and another a uproc table that gets filled with processes that are considered “active” (running, runnable, sleeping). Argptr() is used to grab the table of uprocs off the stack. The integer passed into the function corresponds to the MAX value defined in the ps.c file described below and passed into the user side getprocs() function. If the value of max is less than 0 the kernel side getprocs() function returns -1 as a sign of failure. If no failure is reported past this point the getprocs() function then returns a call to a helper function called getproc_helper() (prototype in uproc.h bottom of file) which takes the MAX value as its first argument and the uproc table as its second argument.

The getproc_helper() function is implemented in the proc.c file in order to gain access to the ptable structure containing processes. The getproc_helper() prototype in uproc.h looks like:

```
int getproc_helper(int m, uproc* table);
```

getproc_helper() takes in the MAX argument (size of the uproc table) and the uproc table to be filled with active processes. A pointer to a proc, named p, is declared as well as an iterating variable named i. A for loop then works its way through the ptable and checks each process in that table to see if it is in an “active” state (running, runnable, sleeping). If the process is “active”, that process is then copied into the table of uprocs at index i, i is then incremented to the next index in the uproc table to copy to, and then the loop checks to see if there are more processes to check for “active” status. Once the loop has finished checking for “active” processes, getproc_helper then returns I back to the getprocs() routine to signify how many “active” processes were added to the uproc table. getprocs() returns this value as its result.

The uproc.h file was created to define what a uproc is. A uproc contains all relevant information to be displayed about a process that is considered “active”(described above). This information includes: a uid for pid, uid, gid, and ppid, a uint for the processes elapsed time and its time spent in the cpu, a uint for the processes size, and char arrays for the processes state and name. STRMAX is defined to 32 for the size of the state and name arrays. The prototype for the getproc_helper() function is also declared here.

- New user commands

Each corresponding file for commands was added to the Makefile under UPROGS and added to the runoff.list file following the same procedure in assignment 1.

1. ps

- ps.c – entire file
- Makefile – line 157
- runoff.list – line 84

The ps command starts by allocating a table of uproc structs (line 7) using the malloc function defined in user.h and a MAX variable defined on line 3 of ps.c. MAX acts as the variable that determines the size of the uproc table, which in turn reflects the maximum number of processes that can be stored in the table. A check is then performed to make sure the table isn't null (line 9). If the table is null for some reason then the command proceeds by printing an error message indicating malloc failed and then exiting. If the allocation worked appropriately, the user side getprocs is called, passing in MAX and the newly allocated table of uproc (line 15). This invokes the kernel side getprocs routine that functions as described above under the getprocs() system call.

Once the getprocs() system call has finished and returned, the table has been filled with the processes that are deemed "active". ps proceeds by using the return value of getprocs(), an integer value representing the number of processes copied into the uproc table, to loop through the uproc table and display the corresponding processes (lines 21-41). Once the loop is done a call to exit() occurs.

2. time

- time.c – entire file
- Makefile – line 158
- runoff.list – line 85

The time command displays the amount of time that the program takes to run. To do this, the time command takes in arguments from the command line using int argc and char* argv[] in the files main declaration. Argc represents the number of command line arguments passed into main and argv[] represents the actual arguments in string format. The idea behind the way time is implemented is to use calls to fork() (line 7), exec() (line 15), and wait() (line 17) to create separate child processes in which to run exec() on each of the arguments in the argv[] array and then wait for each of those processes to finish in order to calculate the time it took for each of them to run. The return value of fork() is used to determine whether the process is the parent process or one of the child processes to run. If the return value of fork() is less than 0 that means the call to fork failed and the command prints an error and exits(). Otherwise the command proceeds by grabbing the current time in ticks (line 8), incrementing to the next argument in the array passed in (line 14) and calling exec using that argument and argv (line 15). The parent process waits for the child process calling exec() to finish (line 17) and then grabs the time in ticks after it finishes (line 18). The time in seconds and partial seconds is calculated for each program run (lines 19 and 20) and that information is then printed (lines 21-24) before exit() is called.

3. idtest

- idtest.c – entire file
- Makefile – line 156
- runoff.list – line 83

The idtest.c file was added as a command in order to test whether or not setuid(), setgid(), getuid(), getgid(), and getppid() function according to the project documentation. A few functions are created in this file to test each individual aspect separately:

1. uidTest(uint nval) (lines 7 - 19)

This function grabs the current uid (line 10) using the getuid() system call and then prints that uid and then prints what the function is going to change the uid to (lines 11 and 12). It then calls the setuid() system call passing in the desired value to change to which invokes the kernel side setuid() system call and changes the uid. Getuid() is then called to grab the new uid and test whether or not the change happened appropriately. The new uid is then printed to display the change.

2. gidTest(uint nval) (lines 21 – 33)

The gidTest() function works in exactly the same way as uidTest() is described above except that it calls setgid() and getgid() to test that the setgid() and getgid() system calls work appropriately.

3. forkTest(uint nval) (lines 35 – 64)

The forkTest() function prints whether or not the value should be inherited based on whether it is appropriate or not. It calls uidTest(nval) and gidTest(nval), passing in the argument nval, in order to test that the fork() functions properly by having the child process inherit the uid and gid of the parent process. If setuid() and setgid() indicate failure (they are given a number less than 0 or greater than 32767) then the forktest prints an error message to indicate that the values are inappropriate (lines 48-51). Otherwise fork() is called for the child process and getuid() and getgid() are called and the changes are printed (lines 56-58).

4. invalidTest(uint nval) (lines 66 - 80)

The invalidTest() function calls setuid() (line 70) and setgid() (line 76) system calls and prints whether or not those functions indicate success for a value that is appropriate or failure for a value that is inappropriate. Appropriate values are in the range $0 < nval < 32767$ and inappropriate values are outside of that range in either direction.

5. testuidgid() (lines 82 - 118)

The testuidgid() function calls each of the above named functions with both appropriate and inappropriate values in order to make sure they function appropriately.

6. int main() (lines 120 – 125)

This main function simply calls testuidgid() to run all appropriate tests.

- Ctrl-p command
* proc.c – lines 520 – 544

Modified the ctrl-p sequence to print out the new uid, gid, ppid, and cpu_ticks_total variables. The header print statement (line 520) was changed to print these new columns. The new variables were added to each print statement and a check to see whether or not the procs parent was null or not was added to make sure the appropriate PPID was printed (either its own for init or its parent for others) (lines 531-534). For cpu_ticks_total seconds are calculated by dividing that value by 100 (). Partial seconds are calculated by modding the cpu_ticks_total value by 100. Those values are used together to print the CPU portion of the ctrl-p sequence.

Testing

1. setuid() and setgid() system call test

This section will be split into two parts; one to test the setuid() system call and one to test the setgid() system call, each one testing against a value that should be acceptable and one that should not. Each test will use the uidTest() and gidTest() functions shown in Figure 1:

```
static void
uidTest(uint nval)
{
    uint uid = getuid();
    printf(1, "Current UID is: %d\n", uid);
    printf(1, "Setting UID to %d\n", nval);
    if (setuid(nval) < 0)
        printf(2, "Error. Invalid UID: %d\n", nval);
    setuid(nval);
    uid = getuid();
    printf(1, "Current UID is: %d\n", uid);
    sleep(5 * TPS); // now type control-p
}

static void
gidTest(uint nval)
{
    uint gid = getgid();
    printf(1, "Current GID is: %d\n", gid);
    printf(1, "Setting GID to %d\n", nval);
    if (setgid(nval) < 0)
        printf(2, "Error. Invalid GID: %d\n", nval);
    setgid(nval);
    gid = getgid();
    printf(1, "Current GID is: %d\n", gid);
    sleep(5 * TPS); // now type control-p
}
```

Figure 1: uidTest() and gidTest()

◦ setuid() subtest

The setuid() test will show three tests. Test 1 will show an appropriate value in the range $0 < \text{value} < 32767$, test 2 will show a value below 0, and test 3 will show a value above 32767. I expect the first test with a value in the range $0 < \text{value} < 32767$ to pass and I expect the second and third tests with a value below 0 and a value above 32767 to fail.

Test 1:

```
$ idtest
Current UID is: 10
Setting UID to 100
Current UID is: 100
```

Figure 2: uidTest()

As shown above in Figure 2 my first test successfully changed the default uid value of from 10 to 100.

Test 2:

```
Setting UID to -1. This test should FAIL
SUCCESS! The setuid system call indicated failure
```

Figure 3: uid value below range

As we can see in Figure 3 above setting the uid value to a negative value below the acceptable range caused the setuid system call to successfully indicate failure.

Test 3:

```
Setting UID to 32800. This test should FAIL
SUCCESS! The setuid system call indicated failure
```

Figure 4: uid value above range

As we can see in Figure 4 above setting the uid value to a value above the acceptable range caused the setuid system call to successfully indicate failure.

In each case, setting the uid to an appropriate value, a value below the acceptable range, and a value above the acceptable range, the setuid system call behaved appropriately by either setting the uid to the appropriate value or successfully indicating failure when an inappropriate value was provided for the uid. I will also demonstrate that setuid() is functioning properly by showing that it successfully changes the shell uid in Figure 5 below:

```
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 State Elapsed CPU PCs
1 init 10 10 1 sleep 1.22 0.03 80104e5b 80104b99 8010686e 80105a6e 80106d86 80106b81
2 sh 10 10 1 sleep 1.17 0.01 80104e5b 80100a05 80101f3f 80101205 80105c2b 80105a6e 80106d86 80106b81
_set uid 20
$ PID Name UID GID PPID State Elapsed CPU PCs
1 init 10 10 1 sleep 9.17 0.03 80104e5b 80104b99 8010686e 80105a6e 80106d86 80106b81
2 sh 20 10 1 sleep 9.12 0.01 80104e5b 80100a05 80101f3f 80101205 80105c2b 80105a6e 80106d86 80106b81
_set gid 20
$ PID Name UID GID PPID State Elapsed CPU PCs
1 init 10 10 1 sleep 66.71 0.03 80104e5b 80104b99 8010686e 80105a6e 80106d86 80106b81
2 sh 20 20 1 sleep 66.66 0.01 80104e5b 80100a05 80101f3f 80101205 80105c2b 80105a6e 80106d86 80106b81
```

Figure 5: Ctrl-p output

We can see that the setuid system call successfully set the shell uid from 10 to 20.

Since all tests above indicate successful setuid() behavior according to my expectations the setuid() test PASSES.

- setgid() subtest

The setgid() test will show three tests. Test 1 will show an appropriate value in the range $0 < \text{value} < 32767$, test 2 will show a value below 0, and test 3 will show a value above 32767. I expect the first test with a value in the range $0 < \text{value} < 32767$ to pass and I expect the second and third tests with a value below 0 and a value above 32767 to fail.

Test 1:

```
Current GID is: 10
Setting GID to 200
Current GID is: 200
```

Figure 6: setgid()

As shown above in Figure 6 my first test successfully changed the default gid value of from 10 to 200.

Test 2:

```
Setting GID to -1. This test should FAIL  
SUCCESS! The setgid system call indicated failure
```

Figure 7: gid value below range

As we can see in Figure 7 above setting the gid value to a negative value below the acceptable range caused the setgid system call to successfully indicate failure.

Test 3:

```
Setting GID to 32800. This test should FAIL  
SUCCESS! The setgid system call indicated failure
```

Figure 8: gid value above range

As we can see in Figure 8 above setting the gid value to a value above the acceptable range caused the setgid system call to successfully indicate failure.

In each case, setting the gid to an appropriate value, a value below the acceptable range, and a value above the acceptable range, the setgid system call behaved appropriately by either setting the gid to the appropriate value or successfully indicating failure when an inappropriate value was provided for the gid. I will also demonstrate that setgid() is functioning properly by showing that it successfully changes the shell gid in Figure 5 above.

We can see that the setgid system call successfully set the shell gid from 10 to 20. Since all tests above indicate successful setgid() behavior according to my expectations the setgid() test PASSES.

Because both subtests PASS, the setuid and setgid system call test PASSES.

2. getuid(), getgid(), and getppid() system calls test

In order to test that the getuid, getgid, and getppid system calls are behaving appropriately I will use a combination of the uidTest() function, the gidTest() function, and the ps command/ctrl-p sequence to show that the appropriate values are being returned by each of these system calls. The uidTest() and gidTest() functions are shown in Figure 1 above.

In my idtest.c file uidTest() and gidTest() are designed to print an error message when the value it tries to set the uid or gid to is not within the acceptable range ($0 < \text{value} < 32767$). Even though getuid and getgid can't really fail since they just return a value that is already checked for errors I can say with some certainty that getuid and getgid will always work appropriately since the call to setuid/setgid would print an error otherwise.

- getuid() and getgid() subtests

In idtest.c I will set the value to change the uid and gid to, to 150. The default value is defined in param.h as 10. First the two system calls should return the default value. After setuid and setgid change these values to 150 the getuid and getgid system calls should return 150 as the new uid/gid.

```
$ idtest  
main-loop: WARNING:  
Current UID is: 10  
Setting UID to 150  
Current UID is: 150
```

Figure 9: getuid() results

```
Current GID is: 10  
Setting GID to 150  
Current GID is: 150
```

Figure 10: getgid() results

As we can see in Figure 9 and Figure 10 above, the `getuid` and `getgid` system calls first return a value of 10 before the `setuid` and `setgid` system calls change them to 150. After the `setuid` and `setgid` system calls finish, the `getuid` and `getgid` system calls return the appropriate value of 150 reflecting that the change happened appropriately. Because the `getuid` and `getgid` system calls behaved according to my expectations these two subtests PASS.

- `getppid()` subtest

The `getppid` system call works by first checking to see if the process has a parent process or not by using a ternary operator. If the process has a parent it should return the parent's pid as its result. If the process doesn't have a parent it means that it was the first process to be created and, as a result, should return its own pid. Based on this information I expect, when running the `ctrl-p` sequence and double checking with the `ps` command, that the first two processes will have the same ppid and any process after that will have a ppid that matches the pid of the process above it in the output.

```
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   State  Elapsed  CPU   PCs
1      init    10    10    1      sleep  1.81     0.03  80104e5b 80104b99 80106
2      sh      10    10    1      sleep  1.77     0.01  80104e5b 80100a05 80101
ps
main-loop: WARNING: I/O thread spun for 1000 iterations
PID   Name   UID   GID   PPID   Elapsed  CPU   State  Size
1      init    10    10    1      4.75     0.03  sleep  12288
2      sh      10    10    1      4.71     0.01  sleep  16384
3      ps      10    10    2      0.05     0.01  run   45056
$
```

Figure 11: ctrl-p/ps output test for `getppid` system call

As we can see in Figure 11 above the first two processes, `init` and `sh`, have the same ppid in the `ctrl-p` output. Looking at the output from the `ps` command we can see that the `ps` process has a ppid that matches the pid of `sh`, the process above it in the output. This confirms that the `getppid` system call is successfully returning the pid of the process' parent process. The behavior of the `getppid` system call acts exactly as I suspected. This subtest PASSES.

Because all subtests (`getuid`, `getgid`, and `getppid`) PASS the testing of all three system calls receives a PASS.

3. Shell built-in commands test

In order to test the shell built in commands I will perform two separate tests. The first test will test the `_set uid`, `_set gid`, `_get uid`, and `_get gid` commands on their own. The second test will test `_set uid` and `_set gid` and then show the `ctrl-p/ps` command output to show that the shell and each new process inherits these values appropriately.

- Shell built-in commands test

When using `_set uid`, `_set gid`, `_get uid`, and `_get gid` on their own I will first get the value to show that it gets the default value of 10. I will then change that value with `_set` and `_get` again to show that the value has appropriately changed. I expect these built-ins to behave by successfully changing the uid/gid values.

```

xv6...
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ _get uid
10
$ _set uid 20
$ _get uid
20
$ _get gid
10
$ _set gid 20
$ _get gid
20
$

```

Figure 12: shell built-in command set/get uid/gid test

As we can see in Figure 12 above the `_get uid` built in command successfully retrieves the DEFAULTUID value of 10. When changed using `_set uid 20` and then retrieved again via `_get uid` the value has been successfully changed to 20. The same successful behavior is observed when using `_get gid` and `_set gid` built-in commands. These two functions behave as expected.

This subtest PASSES.

- Shell built-in ctrl-p/ps test

When performing this test I will first use the ctrl-p sequence to show the uid/gid values before any changes have occurred. I will then use the shell built in commands `_set uid` and `_set gid` followed by the ctrl-p sequence and the `ps` command to show that the changes have happened successfully and to show that further processes inherit the same modified values.

```

init: starting sh
$ PID      Name      UID      GID      PPID      State    Elapsed   CPU      PCs
1          init       10       10       1          sleep    6.28      0.03     80104e5b
2          sh         10       10       1          sleep    6.22      0.01     80104e5b
$ _set uid 15
$ _set gid 15
$ PID      Name      UID      GID      PPID      State    Elapsed   CPU      PCs
1          init       10       10       1          sleep    22.26     0.03     80104e5b
2          sh         15       15       1          sleep    22.20     0.01     80104e5b
ps
PID      Name      UID      GID      PPID      Elapsed   CPU      State    Size
1          init       10       10       1          27.61     0.03     sleep    12288
2          sh         15       15       1          27.55     0.01     sleep    16384
3          ps         15       15       2          0.05      0.01     run      45056
$

```

Figure 13: shell built-in ctrl-p/ps command test output

As we can see in Figure 13 above initially the shell has the DEFAULTUID and DEFAULTGID values when ctrl-p is pressed. After using `_set uid` and `_set gid` to change the uid/gid values to 15 and pressing ctrl-p again, we can see that the uid/gid values for the shell have successfully changed. Furthermore we can also see that, when the `ps` command is typed, the new uid/gid values for shell are successfully inherited by `ps`.

This subtest PASSES.

Because both subtests pass, the shell built-in commands test as a whole PASSES.

4. fork() system call test

For the fork system call test I will use the forkTest(uint nval) function implemented in the idtest.c file and displayed in Figure 14 below. This function sets the uid and gid of a process to the value passed into the function before performing a fork(). It then prints the value of the process uid and gid and performs the fork() followed by printing the child process uid and gid.

```
static void
forkTest(uint nval)
{
    uint uid, gid;
    int pid;

    printf(1, "Setting UID to %d and GID to %d before fork(). Value"
           " should be inherited\n", nval, nval);

    if (setuid(nval) < 0)
        printf(2, "Error.Invalid UID: %d\n", nval);
    if (setgid(nval) < 0)
        printf(2, "Error.Invalid GID: %d\n", nval);

    printf(1, "Before fork(), UID = %d, GID = %d\n", getuid(), getgid());
    pid = fork();
    if (pid == 0) { // child
        uid = getuid();
        gid = getgid();
        printf(1, "Child: UID is: %d, GID is: %d\n", uid, gid);
        sleep(5 * TPS); // now type control-p
        exit();
    }
    else
        sleep(10 * TPS);
}
```

Figure 14: forkTest(uint nval) function used for fork system call test

I expect that, given what the fork system call does, the uid and gid of the parent process will be successfully inherited by the child process.

```
Setting UID to 111 and GID to 111 before fork(). Value should be inherited
Before fork(), UID = 111, GID = 111
Child: UID is: 111, GID is: 111
```

Figure 15: fork system call results

As we can see in Figure 15 above the fork test performed exactly as expected. First it printed out what it was going to set the uid and gid to before the fork; a value of 111 for each. It then prints the parent's value and performs the fork. After the fork is performed the child has successfully inherited the parent's uid and gid value of 111. Further evidence can be seen using the ctrl-p sequence and ps commands to see how the parent uid and gid are inherited after the fork system call.

The fork system call test PASSES.

5. ps command and getprocs() test

For the ps command and getprocs test I will set my program to print the value of MAX before it runs getprocs() so that we can see that it works for each different value: 1, 16, 32, and 64. I expect that this command will work across all different values for max and display the active processes appropriately each time.

```
sb: size 2000 mblocks 1941 nlnodes 200 nlog 30 logstart 2 nlnodestart 32
init: starting sh
$ ps
Max value: 16
PID      Name     UID      GID      PPID      Elapsed    CPU      State     Size
1         init     10       10       1          2.48       0.02     sleep     12288
2         sh       10       10       1          2.44       0.02     sleep     16384
3         ps       10       10       2          0.02       0.01     run       45056
$
```

Figure 16: getprocs with max set to 16

```
init: starting sh
$ ps
Max value: 32
PID      Name     UID      GID      PPID      Elapsed    CPU      State     Size
1         init     10       10       1          1.09       0.04     sleep     12288
2         sh       10       10       1          0.99       0.02     sleep     16384
3         ps       10       10       2          0.03       0.02     run       45056
$
```

Figure 17: getprocs with max set to 32

```
$ ps
Max value: 64
PID      Name     UID      GID      PPID      Elapsed    CPU      State     Size
1         init     10       10       1          1.03       0.04     sleep     12288
2         sh       10       10       1          0.95       0.02     sleep     16384
3         ps       10       10       2          0.03       0.02     run       45056
$
```

Figure 18: getprocs with max set to 64

As shown in the figures above (16 – 18) I was able to get getprocs() working with all values for MAX except for a value of 1. I'm sure if I had slightly more time I could figure out why it isn't working but for now this will have to do. For values of MAX set to 16, 32, and 64 it appropriately displays all the relevant active process information to the screen.

Because the getprocs function doesn't properly display processes for a MAX value of 1 I can't say that this section PASSES every test but it does however PASS for 3 out of the 4 subtests with a fix for the value of 1 coming soon.

6. Elapsed cpu time test

In order to test the elapsed cpu time I will use a combination of the ctrl-p sequence and the ps command to see whether or not the two elapsed times closely match one another after multiple key presses. I expect that the init process will remain constant since it only spends a brief amount of time in the cpu during the beginning. The sh command should vary slightly each press since it gets updated each time you ctrl-p or use the ps command. The ps command should be different most times since it is a new process each time it is pressed.

```
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 State Elapsed CPU PCs
1 init 10 10 1 sleep 3.11 0.03 80104e5b 80104b99 8010686e
2 sh 10 10 1 sleep 3.07 0.01 80104e5b 80100a05 80101f37
ps
PID Name UID GID PPID State Elapsed CPU State Size
1 init 10 10 1 6.69 0.03 sleep 12288
2 sh 10 10 1 6.65 0.03 sleep 16384
3 ps 10 10 2 0.03 0.01 run 45056
$ PID Name UID GID PPID State Elapsed CPU PCs
1 init 10 10 1 sleep 9.49 0.03 80104e5b 80104b99 8010686e
2 sh 10 10 1 sleep 9.45 0.03 80104e5b 80100a05 80101f37
ps
PID Name UID GID PPID State Elapsed CPU State Size
1 init 10 10 1 13.71 0.03 sleep 12288
2 sh 10 10 1 13.67 0.04 sleep 16384
4 ps 10 10 2 0.02 0.01 run 45056
$ PID Name UID GID PPID State Elapsed CPU PCs
1 init 10 10 1 sleep 15.90 0.03 80104e5b 80104b99 8010686e
2 sh 10 10 1 sleep 15.86 0.04 80104e5b 80100a05 80101f37
ps
PID Name UID GID PPID State Elapsed CPU State Size
1 init 10 10 1 18.77 0.03 sleep 12288
2 sh 10 10 1 18.73 0.06 sleep 16384
5 ps 10 10 2 0.05 0.03 run 45056
$
```

Figure 19: elapsed cpu time test

As we can see in the figure above the elapsed time is growing in a consistent manner. This is to be expected because I allowed a certain amount of time between each key press to get a bit of a time difference. We can also see that the ps command closely matches with the ctrl-p sequence, differing only due to the fact that PSU servers were lagging hardcore and there was time between each press. I was also correct in assuming that init would stay constant throughout each press and across ctrl-p/ps presses. We can also see that sh was slightly different with each press growing in a small positive direction. I believe this is because it is updated each time a press occurs. We can also see that ps varies by press. This is because it is a new process each time the ps press occurs and we can see this because they have differing PID values across each ps press.

I am convinced, because all of my assumptions were correct across presses, that this test PASSES.

7. time command test

In order to test the time command I will demonstrate how the time command handles five different cases. I expect, in each of these cases, that the output will match what is described under each separate case.

Case 1: the null command

I expect that the time command will display a running time of 0.00 seconds since nothing is typed into the command line as an argument for time to try and time.

```
SD: SIZE 2000 mblocks 1941
init: starting sh
$ time
(null) ran in 0.00 seconds
```

Figure 20: Time null case

As we can see in Figure 20 above, when given no argument to time, the time command simply states that (null) ran in 0.00 seconds. This makes sense because if there is nothing to run there shouldn't be a time involved in running nothing.

Case 1 PASSES.

Case 2: Time ls

I expect that the time command will first display the results of ls and then display a time greater than 0.00 seconds to indicate that ls took a small amount of time to run.

```
init: ran in 0.00 seconds
$ time ls
main-loop: WARNING: I/O thread spun for 1000 iterations
.      1 1 512
..     1 1 512
README 2 2 1973
cat     2 3 13504
echo    2 4 12712
forktest 2 5 8432
grep    2 6 15140
init    2 7 13272
kill    2 8 12788
ln       2 9 12688
ls      2 10 14916
mkdir   2 11 12844
rm       2 12 12820
sh       2 13 26536
stressfs 2 14 13416
usertests 2 15 58604
wc       2 16 14020
zombie  2 17 12488
halt     2 18 12444
date     2 19 13960
idtest  2 20 15956
ps       2 21 14224
time    2 22 13484
console 3 23 0
ls ran in 0.08 seconds
$
```

Figure 21: Time ls case

As we can see in Figure 21 above, the output for the ls command is first displayed and then the time in seconds that ls took to run is displayed below that output indicating that ls took a small amount of time to run. This behavior is exactly as expected for this case.

Case 2 PASSES.

Case 3: Time echo abc

I expect that, in this case, the output for time will first include abc since echo simply prints its arguments. It will then display the amount of time that echo took to run which should be a value greater than 0.00 since it will take a small amount of time for echo to run.

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

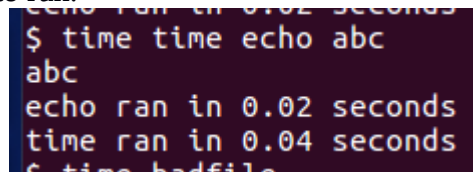
Figure 22: Time echo abc

As we can see in Figure 22 above the time command behaved exactly as expected. First abc is printed since that is the expected behavior of echo. The time that echo took to run is then printed. The value is 0.02 seconds which makes sense since it should take a short amount of time for echo to run.

Case 3 PASSES.

Case 4: time time echo abc

I expect that, in this case, we will see similar results to case 3. First abc will be printed followed by the amount of time it took echo to run which will be greater than 0.00. This time, since there is an extra time command, it will also print the amount of time it took for the second time command to time echo. This value should be slightly higher than the time it took echo to run.



```
echo ran in 0.02 seconds
$ time time echo abc
abc
echo ran in 0.02 seconds
time ran in 0.04 seconds
$ time badfile
```

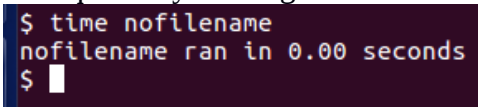
Figure 23: time time echo abc

As we can see in Figure 23 above the time command behaved exactly as expected. First abc is printed to the screen as a result of echo. Then the amount of time it took echo to run is printed followed by the amount of time it took the second time command to time echo to run. The value for the time command is also slightly higher than the value for the echo command which makes sense since it should take a slightly longer time for time to time the echo command.

Case 4 PASSES.

Case 5: time nofilename

I expect that, in this case, since we are trying to time a file that does not exist time will simply state that it took 0.00 seconds to run the nofilename since it is a file that does not exist and doesn't require any running time.



```
$ time nofilename
nofilename ran in 0.00 seconds
$
```

Figure 24: time nofilename

As we can see in Figure 24 above, my assumptions were correct. It took precisely 0.00 seconds for nofilename to run because the file name does not exist. I assume that it just treats that nofilename as a string and, because nothing by that name exists, there is nothing to time. Hence why the value is 0.00 seconds.

Case 5 PASSES.

Because all 5 separate cases have passed their tests the time command test as a whole PASSES.