1 Project Report [45 pts.]

1.1 Project Description [1 pt.]

A brief description of the project.

1.2 Project Deliverables [10 pts., one per deliverable]

- New system calls.
  System call name, a short description, and the function prototype.

  1. setuid. Set the UID of the current process to *value*.
     ```
     int
     setuid(int value);
     ```
  2. setgid.
  3. getuid.
  4. getgid.
  5. getppid.
  6. getprocs. Fills in a provided table of uproc structures with information about active processes in the system up to a maximum.
     ```
     #include "uproc.h"
     int
     getprocs(uint max, struct uproc *table);
     ```

- Process total CPU time.
  Each process now stores the starting CPU "time" so that the elapsed CPU time since process start can be calculated.

- The `ps` command.
  A new user command that will print out the status of active processes in the system.

- The `time` command.
  A new user command that will time the execution of commands.

- The ctrl−p console command.
  Modified to display uid, gid, ppid, and elapsed process time.

1.3 Implementation [13 pts.]

Preferred: breakdown implementation by deliverable; combining deliverables as appropriate. For example, the implementation for the `ps` user command and the `getprocs()` system call can be grouped together.

System call return values must be described. If a call cannot fail, then set the return to 0 and in the description, indicate that 0 is success and use a dummy for failure (e.g., −1.)

1. New System Calls [5 pts., 1 points each.]

   Overview of system call implementation. *e.g., "using the steps outlined in assignment 1 . . . ".* No need to explain what each file is for as that occurred in project one.

   Do not forget to show the function prototype and to detail limitations on function parameters, etc. *The uid and gid fields in the process structure may only take on values* $0 \leq value \leq 32767$. (from the project description).

- `setuid()` system call. List files and line numbers. Explain parameters and note which helper functions were used to get arguments from the stack. Indicate possible return values and their interpretation.
- `setgid()` system call.
- `getuid()` system call.
- `getgid()` system call.
- `getppid()` system call. Must note that the first process (`init`) has no parent and so will be its own parent *as a special case*.

2. Elapsed Process Time [2 pts.]
   File and line numbers with short explanations

3. `getprocs()` System Call [2 pts.]
   File and line numbers with short explanations. The system call entry point is in sysprocs.c but the implementation is in `proc.c` as the call accesses the `ptable` structure and that can only be accessed in `proc.c`. Include the `uproc` structure.

4. New user commands [4 pts.]

   - `ps` command [2 pts.]
     File and line numbers with short explanations
   - `time` command [2 pts.]
     File and line numbers with short explanations
   - Any new commands written to test uid/gid/ppid system calls. Will probably be referenced in the test section. No additional points for describing them here.

5. ctrl−p command [2 pts.]
   File and line numbers with short explanations

1.4 Required Tests

This section must include enough information to successfully demonstrate that the deliverables are correctly implemented. This includes testing with invalid parameters for system calls.

The report may include screenshots or copy-and-paste of test outputs. Screenshots are preferred. Test outputs with no explanation are insufficient. Each test much contain:

1. Test name. Each test has a descriptive name. This allows tests to be referenced from elsewhere in the report.
2. Test description (e.g., what is tested)
3. Expected results
4. Test results (screenshot or listing)
5. Discussion of test results
6. A clear indication as to whether the test PASSED or FAILED. If some test is designed to show a limitation (e.g., invalid uid), then a test showing an error when an error was expected would still be a test that PASSED. This is why the expected results part is important.

Tests [21 pts.]

- `setuid(), setgid()` system calls. Tests include:
  1. Setting using a valid parameter. [1 pt. each]
  2. Setting using an invalid parameter. [1 pt. each]
- `getuid(), getgid(), getppid()` system calls.
  One or more tests showing that the correct values are returned. These routines cannot fail, but user test program(s) *must check for a return code* and handle an error code as appropriate. The user code cannot assume that a system call can not fail. [1 pt. each, 3 pts. total]

  Note that `getpid()` testing can use the `ps` command or ctrl−p console sequence.
- Shell built-in commands. Tests to show that the commands work properly. A simple test would be: get then set then get. [2 pts. total]
- `fork()` system call. Write a user program that sets its own uid/gid to a new value, forks, and then prints out the information again, demonstrating that the uid/gid is properly inherited from the parent process. [3 pts.]
- `ps` command and `getprocs()` system call with `max` = 1, 16, 64, 72. This will likely require a recompilation. Easiest way to demonstrate this is to have the `ps` command print the `max` parameter prior to calling `getprocs()`. It is acceptable for the `ps` command to take an argument that is the value for this parameter. [5 pts.]
- Elapsed CPU time in ctrl−p. Easiest way to show this is a combination of `ps` and ctrl−p. [1 pts.]
- `time` command. [5 pts., 1 points each]
  1. `time .` The time command with no arguments. Output can complain about no command to time or it can time the NULL command. Timing the NULL command is preferred.
  2. `time ls`. Output should be reasonable.
  3. `time echo abc`. This test ensures that the entire command line is passed to the timed command.
  4. `time time echo abc`. Ensures generality of the time command.
  5. `time badfilename`. Time command with invalid filename.

2 Project Code [60 pts.]

2.1 UID/GID Setup [6 pts.]

- UID/GID defaults must be set in `userinit`. Defaults must be defined in `params.h`.
- UID/GID must be inherited from the parent process in `fork()`.

2.2 System Calls [24 pts.]

- `setuid()` [4 pts.]
- `setgid()` [4 pts.]
- `getuid()` [2 pt.]
- `getgid()` [2 pt.]

- `getppid()` [2 pts.]
  - These affect only the currently executing process, so may be implemented completely within `sysproc.c`. Note that the UID/GID setters in `sysproc.c` must ensure that $0 \leq$ UID/GID ≤ 32767. The getters can assume that the stored value is correct.
- `getprocs()` [10 pts.]
  - Must be primarily implemented in `proc.c`, with the implementation in `sysproc.c` being a wrapper (primarily to extract the arguments off the stack).

2.3 CPU Time [5 pts.]

`cpu_ticks_total` must store the total ticks that the process has used up in the CPU.

2.4 User Commands [15 pts.]

- `ps` [5 pts.]
  - Must dynamically create an array of `uproc` structs using `malloc`, pass a pointer that structure into `getprocs`, and print out all of its information on return for every process.
- `time` [10 pts.]

2.5 Console Control Sequences [5 pts.]

- ctrl−p [5 pts.]
  - Must print out all of the information in Project 1 as well as the `UID`, `GID`, `PPID`, and `CPU total time` additions for this project.

2.6 Coding Style [5 pts.]

All system calls must be implemented in a consistent manner with the existing code base. For example for `getprocs()`:

- `getprocs()` must be declared in `user.h`
- `sys_getprocs()` must have `void` arguments, and must be declared in `syscall.c` with the other `extern` prototypes.

  Kernel side `getprocs()` must be implemented in `proc.c`. The wrapper function, `sys_getprocs()` calls this routine with appropriate arguments.

System calls must **not** print anything to the console, although console control sequences do print to the console. System calls *are required* to return an appropriate error code using the existing convention in the code base (where negative values represent errors). All user programs are required to check the return code from any system call and handle errors as appropriate.