# Project 2
# Processes

## Introduction

This project will focus primarily on processes.

In this project, you will become familiar with:

1. Locks for kernel-level data structures; concurrency.
2. Implementing new system calls to support process ownership.
3. Implementing a new system call to obtain process information.
4. Implementing tracking for the amount of time a process uses a CPU.
5. Implementing a new user-level command to display process state.
6. Implementing a new user-level command to time process execution.
7. Modifying the xv6 console to display additional process information.
8. Writing a project report to properly document project work.

## UIDs and GIDs and PPIDs

At this point xv6 has no concept of users or groups. You will begin to add this feature to xv6 by adding a `uid` and `gid` field to the process structure, where you will track process *ownership*. These should be of type `unsigned int` since negative UIDs and GIDs make no sense in this context.

The `ppid` is the "parent process identifier" or parent PID. The `proc` structure does not need a `ppid` field as the parent can, and should, be determined on–the–fly. Look carefully at the existing proc structure in proc.h to see what is needed. **Note** that process number one, the `init` process is a special case as it has no parent. **Process one should be shown as its own parent**. This is consistent with Unix/Linux.

You will need to add the following system calls.

```
uint getuid(void)        // UID of the current process
uint getgid(void)        // GID of the current process
uint getppid(void)       // process ID of the parent process

int setuid(uint)         // set UID
int setgid(uint)         // set GID
```

Your kernel code cannot assume that arguments passed into the kernel are valid and must check them for the correct range. The `uid` and `gid` fields in the process structure may only take on values $0 \leq \text{value} \leq 32767$. You are required to provide tests that show this bound being enforced by the kernel-side implementation of the system calls.

The following code is a starting point for writing a test program that demonstrates the correct functioning of your new system calls. This example is missing several important tests and fails to check return codes, which is very bad programming. You should fix the shortcomings of this code or write a new test program that properly demonstrates correction functionality for **all** test cases.

```
int
testuidgid(void)
{
    uint uid, gid, ppid;

    uid = getuid();
    printf(2, "Current UID is: %d\n", uid);
```

```
    printf(2, "Setting UID to 100\n");
    setuid(100);
    uid = getuid();
    printf(2, "Current UID is: %d\n", uid);

    gid = getgid();
    printf(2, "Current GID is: %d\n", gid);
    printf(2, "Setting GID to 100\n");
    setgid(100);
    gid = getgid();
    printf(2, "Current GID is: %d\n", gid);

    ppid = getppid();
    printf(2, "My parent process is: %d\n", ppid);
    printf(2, "Done!\n");

    return 0;
}
```

**Other Necessary Modifications** You have modified the process structure to include the uid and gid for the process, but that isn't all the work necessary to support these new features. The `fork()` system call allocates a new process structure and copies all the information from the original process structure to the new one, with the exception of the `pid`. But you modified the process structure! You will need to find the code for the `fork()` system call and make sure to copy the uid and gid of the current process to the new child process.

Not all processes are created with `fork()`. The first process, which eventually becomes the `init` process, is created piece–by–piece at boot time. The routine `userinit()` in the file `proc.c` is where this initialization takes place. Add a #define statement in `param.h` for the default uid and gid of the first process. This will make your code easier to read.

You should also be able to set the uid and gid of the currently executing shell with appropriate built-in commands. The shell includes in the parser the ability to identify built-ins as built-ins begin with an underscore (_). Take a look at the shell (`sh.c`) to see what conditional compilation flag is necessary to turn on this functionality. The following built-in commands have been implemented.

```
_set uid int
_set gid int
_get uid
_get gid
```

You should include some tests that show the uid/gid for the shell being changed and that any program you run from the command line inherits the correct uid and gid.

## Process Execution Time

Currently, your xv6 system tracks when a process enters the system and displays *elapsed* time in the console command "control–p". You will now track how much CPU time a process uses.

There are two situations where a context switch occurs in xv6: one to put a process into a CPU and one to take a process out of its current CPU. The currently running process is removed from its CPU in the routine `sched()` and a RUNNNABLE process is put into a CPU in the routine `scheduler()`, both are in `proc.c`.

You will need to add two new fields to the process structure.

```
uint  cpu_ticks_total;    // total elapsed ticks in CPU
uint  cpu_ticks_in;       // ticks when scheduled
```

You do not need a `cpu_ticks_out` field.

The `cpu_ticks_in` value will be set when the process enters a CPU. The `cpu_ticks_total` will be updated when the process is removed from its CPU.

A new process is allocated in the routine `allocproc()` in the file `proc.c`. Initialize these two new fields to zero in that routine.

## The "ps" Command

Xv6 does not have the `ps` command like Linux, so you will add your own. This command is used to find out information regarding active[1] processes in the system. In order to write your `ps` program, you will need to add another system call.

In the file `proc.h` you will find `struct proc`. When xv6 is running, there is an array of `proc structs` in the data structure named `ptable` in proc.c. All information for each process is in a `struct proc` in the `proc[]` array.

The `ps` will print the following information for each active process

1. process id (as decimal integer)
2. process uid (as decimal integer)
3. process gid (as decimal integer)
4. parent process id (as decimal integer)
5. process elapsed time (as a floating point number)
6. process total CPU time (as a floating point number)
7. state (as string)
8. size (as decimal integer)
9. name (as string)

You'll print one line for each process, with a header indicating the contents for each column.

The system call that you need to add is called `getprocs`:

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

Note that there is a new structure **uproc**. This new struct is there because you do not need all the information stored in the `struct proc`. Due to restrictions on the size of the stack in xv6, you will need to allocate memory for this data structure from the heap; that is, use `malloc`. If you create the data structure correctly you will be able to access it as though it were an array. You will pass in a pointer to the array of uproc structs that the kernel will fill in. The argument `max` is the maximum number of entries that your struct uproc will hold; that is, the size of the array. Your `getprocs()` call should return the actual number of entries used in the table on success and -1 on any error. Your `ps` program should do something sane when an error is returned. You must test the system call with at least `max` set equal to 1, 16, 64, and 72.

[1] We define "active" here to be a process in the RUNNABLE, SLEEPING, or RUNNING state. You can also include processes in the ZOMBIE state but in no circumstances should you include processes in the UNUSED or EMBRYO states.

Use this definition for the `uproc` structure. Put it in a file named `uproc.h`.

```
#define STRMAX 32

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

The value for `STRMAX` should be able to take on *any* non-negative value and your routine should still work correctly.

## The "time" Command

Your `time` command will determine the number of seconds that a program takes to run.
**Example**

```
$ time forktest
fork test
fork test OK
forktest ran in  2.14 seconds.
$
$ time echo "abc"
"abc"
echo ran in 0.3 seconds.
$
```

The last line of each test is the output from the `time` command. The previous lines are output from the `forktest` and `echo` commands. The `echo` test demonstrates how the entire command line is passed to the named command. As a more complex example, consider this test:

```
$ time time echo "abc"
"abc"
echo ran in 0.4 seconds.
time ran in 0.7 seconds.
$
```

This test shows the `time` command being called twice. In reading the line left–to–right, we can see that the first `time` command is timing the second `time` command. The second `time` command is timing the `echo` command that will echo the string ``abc'' to standard output. Since the `echo` command will just print *all* its arguments to standard out, the exact order of the commands is important in this example. This next example should help the student to understand better:

```
$ time echo "abc" time
"abc" time
echo ran in 0.4 seconds.
$
```

Additionally, your command will properly handle the case of no program name being provided. In this example, the program name is left out:

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

You will want to base your timing information on the kernel global variable called `ticks`. If you review the list of system calls, you will find an existing call that will suffice.

You will add the `time` command to your system the same way that you added your `date` command.

## Modifying the Console

Recall that the console, `console.c`, processes the control – p command. Update the output of this command to include the new process information added in this project. This means that the control – p command should have the same headings as the `ps` command with the addition of the existing output for PC (program counter) information. Here is an example:

```
$
PID     Name        UID     GID     PPID    Elapsed CPU     State   Size    PCs
1       init         0       0       1      1.48    0.3     sleep   12288   80105469 801050a3 8010
2       sh           0       0       1      1.41    0.1     sleep   16384   80105469 80100a4f 8010

$ ps

PID     Name        UID     GID     PPID    Elapsed CPU     State   Size
1       init         0       0       1      26.48   0.3     sleep   12288
2       sh           0       0       1      26.41   0.2     sleep   16384
22      ps           0       0       2      0.4     0.1     run     49152
$
```

## Required Tests

The following tests are required. Students may decide to provide additional tests, but these tests must be shown with the reasoning for each test explained, example output, and a discussion of the test with an indication as to whether the test PASSED or FAILED.

1. Correct setting of the UID/GID with a valid parameter for the system call.
2. Correct behavior of the UID/GID system calls with invalid parameters (including returning an error code).
3. Correct PPID returned for the getppid system call.
4. Using the shell built–in commands to show proper functioning of the UID/GID system calls.
5. Correct setting of the UID/GID by the `fork()` system call.
6. `getprocs()` tests for the parameter `max` set to 1, 16, 64 and 72 to show that the data structure is filled properly for the specific values of `max`.
7. `ps` command. Show that the correct processes are output and that the information matches CTRL-P.
8. Correct elapsed time shown in CTRL-P.

9. Time command:

    No argument case

    Bad command (e.g. time fasdfas)

    Command with no arguments (e.g. time forktest)

    Command with one or more arguments (e.g. time echo abc)