Project 2
Processes

Introduction

This project will focus primarily on processes.

In this project, you will become familiar with:

1. Concurrency control for kernel-level data structures.
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 will be of type `unsigned int` since negative UIDs and GIDs make no sense in this context. Note that when these values are passed into the kernel, they will be taken off the stack as `int`s. There is no issue with this as you will convert them to `unsigned int`s immediately. It is, however, critical, that the function prototypes in `user.h` declare values as unsigned as you will see below.

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 the `init` process is a special case as it has no parent. Your code must account for any process whose parent pointer is `NULL`. For any such pointer, you will display the PPID to be the same as the process PID. Do not modify a parent pointer that is set to `NULL`; leave it that way as it becomes important in a later project.

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 so your kernel code must check the values 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 correct functionality for **all** test cases. You can also take a look at `testuidgid.c` and `testsetuid.c` for inspiration. For whatever tests you use, you need to argue how they address the rubric requirements.

```
int
main(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");

    exit();
}
```

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 `pdx.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, and connect that to what happens when you set `CS333_PROJECT` to `2` in the `Makefile`. The following built-in commands have been implemented in `sh.c`, but the corresponding system calls may not be implemented yet. In that case, just turn off the correct flag in the Makefile. You will need to implement the corresponding system calls before turning on the built-ins that depend on them.

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

You should include 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 RUNNABLE 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, i.e. when it leaves the RUNNING state.

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 processes in the system. We define "active" here to be a process in the RUNNABLE, SLEEPING, RUNNING, or ZOMBIE state. Processes in the UNUSED or EMBRYO states are not considered active. In order to write your `ps` program, you will need to add another system call.

You will find `struct proc`, “the process structure”, in the file `proc.h`. 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 `ptable` also contains a field `lock`, that you use when you need to access the other contents of the `ptable`, such as the `proc[]` array, in an *atomic transaction*. Whenever you need to read or modify multiple values (words of memory) in the `ptable` *indivisibly*, you must follow the pattern below. This locking discipline ensures that these accesses happen in an atomic transaction, i.e. all at once with respect to other atomic transactions that use the `ptable` lock:

1. Acquire the `ptable` lock
2. Access the contents of the `ptable`
3. Release the `ptable` lock

This pattern assumes that you do not already hold the `ptable` lock, and that you've accessed everything you intend to indivisibly access before releasing the lock.

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

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

You'll print one line for each process, with a header indicating the contents for each column. You'll have to load multiple values to get all of these fields for each process. If we don't ensure that these values are accessed in an atomic transaction, we could observe a state of the `ptable` that never existed at any one point in time, because some values could be updated by another CPU while your CPU is in the process of loading them[1].

Note that the xv6 `printf` routine does not support floating point numbers. This is fine since, for the values we will be calculating, the integer portion before the decimal point can be calculated with integer division and the integer portion after the decimal can be calculated using modulo arithmetic.

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

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

[1]Note that `procdump()` does not access the `ptable` in an atomic transaction. Try hitting control−p a few times while multiple processes are cycling between the RUNNING and RUNNABLE states (e.g. after running `p3-test`) to see an example of such an impossible `ptable` state, where 3 processes appear to be “running” despite the fact that xv6 is only running on 2 CPUs.

First, note that the ptable data structure is statically declared in proc.c. This means that even though you'll implement sys_getprocs in sysproc.c as usual, it will need to call a helper in proc.c in order to actually access the ptable.

Additionally, 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 and you should never make a kernel data structure visible to a user program as the user program could modify the data. Due to restrictions on the size of the stack in xv6, you will need to allocate memory from the heap for this data structure *in the user program*; that is, use malloc in your ps.c. If you create the data structure correctly you will be able to access it as though it were an array from inside the kernel. 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 array of struct uprocs can hold. getprocs should only copy entries for active processes, and should return the actual number of entries copied into 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.

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  0.915 seconds.
$
$ time echo "abc"
"abc"
echo ran in 0.041 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 0.031 seconds.
time ran in 0.066 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 0.056 seconds.
$
```

Your command will behave the same way when no program name is provided:

```
$ time
(null) ran in 0.016 seconds.
```

Note that this last program is "timing" a NULL command. This is correct behavior and how the time command works on Linux. You do not have to put in a special check for a NULL or invalid command in your code.

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 returns the current value of `ticks`.

You will add the `time` command to your system the same way that you added your `ps` command. In particular, note that all you need to add is a user program `time.c` that uses existing system calls to run another process and time it. Observe that `_time` is appended to `CS333_UPROGS` when `CS333_PROJECT` is set to 2 or higher in the `Makefile`. This will cause `make` to try and build the user program it finds in `time.c`, and include the resulting executable in the filesystem you have access to from inside xv6.

Modifying the Console

The console, `console.c`, processes control sequences such as 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. The code that handles control sequences in `console.c` calls a helper function in `proc.c` as the proc data structure should, to the extent possible, only be handled in `proc.c`. Here is an example output. Note that because of the resolution of the `ticks` variable that the various times are represented to three digits past the decimal. This will require some special checks in your updated `procdump()` routine!

```
$
PID     Name        UID         GID     PPID    Elapsed CPU     State   Size    PCs
1       init        0           0       1       1.249   0.033   sleep   12288   801039c6 80103adb 80104fd
2       sh          0           0       1       1.213   0.012   sleep   16384   801039c6 801002c4 8010181
$ ps

PID     Name        UID         GID     PPID    Elapsed CPU     State   Size
1       init        0           0       1       2.549   0.033   sleep   12288
2       sh          0           0       1       2.513   0.018   sleep   16384
3       ps          0           0       2       0.022   0.011   run     49152
$
```

Required Tests

All tests are defined in the accompanying rubric.

Before You Submit

As described in the *Survival Guide*, you are required to test that your submission works properly before submission. There is a `Makefile` target to help you. You are **required** to run "`make dist-test`" to verify that your kernel compiles and runs correctly, including all tests. A common error is to not put the name of new files into `runoff.list`. Once you have verified that the submission is correct, use "`make tar`" to build the archive file that you will submit. Submit your project report as a separate PDF document. **Failure to submit your report in PDF format will result in a grade of zero for the report. No exceptions.**