Please **read this entire assignment**, before you start working on the code.

This lab is February 27 by midnight. Submit a single gzipped `tar` file to **TEACH**. If you don't remember how to create a gzipped `tar` file, you need to learn before you submit this assignment. If your submission is not a gzipped `tar` file, I will not grade your assignment.

There are many (**MANY**) parts to this assignment. Each one is fairly small. Just follow this document like it is a script or recipe and work through all the parts. I recommend you use `#ifdef` sections in your code to make it easier to track where you make changes to the `xv6` source code. I'm sure you have plans to refactor your code after the assignment is due, putting in comments, using mnemonic macros, and using conditional compilation blocks to separate new and old code. Instead, perform that before the due date.

This assignment is done entirely in the `xv6` environment.

This programming project is worth 200 points!!!

Part 0 – Clone The Starting Code – (0 points)

I strongly recommend you being with a clone (using `xv6-clone.bash` script) the xv6-lottery directory under `~chaneyr/Classes/cs444/xv6/xv6-lottery`. The rest of this document make frequent reference to portions of code in those files. The command line you'd run should look like this:

```
./xv6-clone.bash -s xv6-lottery -d xv6-lottery
```

There are many `#ifdef` sections in the code to guide you.

Part 1 – Some additional programs – (5 points)

Add the `mult.c` and `mfork.c` programs into the `xv6` system (by adding the C code, and editing the `Makefile`). Those programs will be a regular part of your `xv6` system. You will use these programs to test the correct functioning of your code and they will be used to test your code when grading your assignment. The C files `mult.c` and `mfork.c` can be found in my `Lab3` directory, `~chaneyr/Classes/cs444/Labs/Lab3`

The purpose of the `mult` program is to just take a long time to complete. The purpose of the `mfork` program is to just fork a number of processes running in the `xv6` kernel. Add these into the `UPROGS` macro in your `Makefile`.

Part 2 – Add some tracking information to each process (20 pts)

Add four members to the `struct proc` data structure (found in `proc.h`). Remember, you are the kernel developer and master level C programmer, so you should be comfortable manipulating the kernel structures. Of course, I put all this stuff within `#ifdef` blocks using a macro called `PROC_TIMES`.

1. Add a member type `struct rtcdate`. You'll find the definition of that structure in the `date.h` file. I like to call this member `begin_date`.
2. Add 3 members of type `unsigned int`. Good names for those 3 members are:

a. `ticks_total` – this will represent the total number of time ticks that the process has run.
b. `ticks_begin` – this will be used to help calculate the total number of time ticks the process has used.
c. `sched_times` – this will be used to count the number of times the process has been scheduled to run.

You can find the definition of `struct rtcdate` in the `date.h` file. While you are in the `date.h` file, make sure it has multiple-include protection.

Now that you have the new data members in the `struct proc` data structure, it's time to put them to use. When a process is first ***allocated***, set the `begin_date` to the "current" date/time. How do you know where a process is first allocated? You might look for a function called something like `allocproc()`. The `qemu` clock seems to synchronize to UTC time at startup. Finding the right call to get the date/time is a little awkward. I recommend you look in the `lapic.c` file for the function `cmostime()`. However, you probably don't want to spend too much time looking at that in that file, it's a bit icky. Calling the `cmostime()` function is a wee bit awkward, because you must pass an address or pointer. Remember that putting an `&` in front of a variable/structure when calling a function will pass the address of the variable/structure. Decide to ignore the `goto` and the label found in that function; we won't talk about them.

Once you set the `begin_date` member, set `ticks_total`, `ticks_begin`, and `sched_times` to zero in the same area of the code (when a new process is allocated).

Now let's look at the `scheduler()` code (in `proc.c`). You'll notice that the scheduler runs as an infinite loop (see the `for(;;)` ?). There are 2 places where you want to add a few lines of code into the `scheduler()` routine. One is just before the newly chosen process is scheduled and the other is just after that process returns back to the scheduler. The first place (before the newly chosen process runs), is pretty easy to find. Look for the place where the state of the process is set to `RUNNING`. I dropped an `#ifdef` block just after that. That block does 2 things, it increments the `sched_times` member for the chosen process, and sets the `ticks_begin` member to the current number of ticks that have accumulated for the vm. How do you find out how many "ticks" the system has been up? That is an excellent question. My recommendation is to look for a function called `uptime()`. *Unfortunately*, because of how the kernel function `uptime()` is defined and implemented (as a user code callable function `sys_uptime()`), you cannot directly call it. You have a choice to either 1) replicate the capability of `uptime()` in the `scheduler()` code (which I **do not** like), or 2) make a new function that returns the same thing (which I **do** like). If you decide to do the second option, I recommend you have `sys_uptime()` directly call your new function. I went with option 2; I don't like to duplicate code.

Now, about that second place to drop in a few lines of code. What do you think the following 2 lines of code from the `scheduler()` function do?

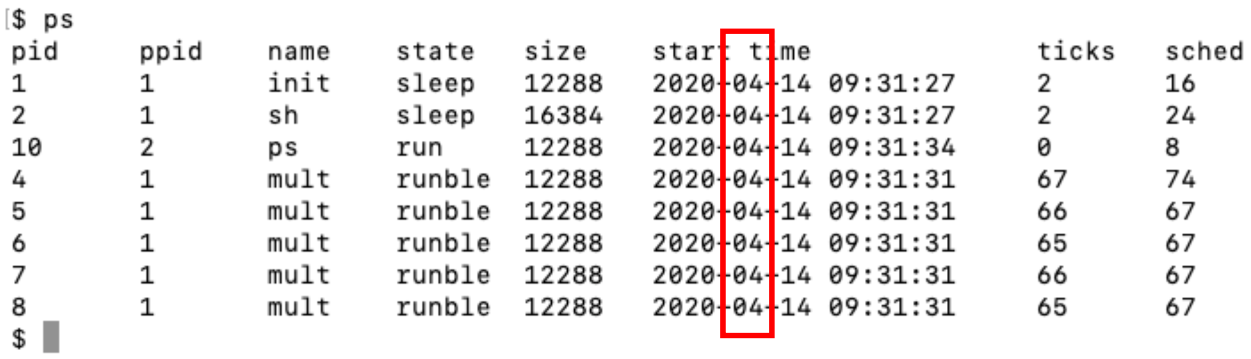
```
    swtch(&(c->scheduler), p->context);
    switchkvm();
```

In addition, notice that immediately after those lines of code, the `c->proc` variable is set to `0` (aka `NULL`). So, might this be that when a new process is actually scheduled to run by calling `swtch()` and `switchkvm()`, and it returns back to scheduler following those calls? Looks like a good place to update the `total_ticks` member of the process (after return from `switchkvm()`).

Part 3 – Modify `ps` to show time tracking information (5 pts)

Now that you have all this great time tracking information for each process, modify your `cps` function to display it.

You should display the information as follows.



```
$ ps
pid      ppid     name     state    size     start time              ticks    sched
1        1        init     sleep    12288    2020-04-14 09:31:27     2        16
2        1        sh       sleep    16384    2020-04-14 09:31:27     2        24
10       2        ps       run      12288    2020-04-14 09:31:34     0        8
4        1        mult     runble   12288    2020-04-14 09:31:31     67       74
5        1        mult     runble   12288    2020-04-14 09:31:31     66       67
6        1        mult     runble   12288    2020-04-14 09:31:31     65       67
7        1        mult     runble   12288    2020-04-14 09:31:31     66       67
8        1        mult     runble   12288    2020-04-14 09:31:31     65       67
$ 
```

One of the fun things you'll notice is that you may need to put a zero in where a value is represented as a single digit. Notice that the month shown in the `start time` is `04`, not just `4`. It's a simple trick, but worth learning. If you had a fully functioning `printf()` (or for this example `cprintf()`), it would be just a change to the format string. However, your `printf()` is not capable of that (and don't spend the month or 6 working on the format characters in the `xv6` version of `printf()`).

Part 4 – Add a `rand()` function (10 pts)

You will need to use a function the generates random numbers. More specifically, pseudo-random numbers. This does not need to be cryptographically secure random numbers, just something reasonable, such as the standard Unix/Linux `rand()` function returns.

Amazingly, if you happened to read to the bottom of the `man` page for `rand`, in section 3 of the `man` pages, you can see some source code for a version of `rand()` that works just fine for this purpose. Differing from the `man` page example, you will call your functions `rand()` and `srand()`, **NOT** `myrand()` and `mysrand()`. If you prefer to use some other swanky random number generator, that's fine but not required.

Your implementation of `rand()` must be done in 2 files: `rand.c` and `rand.h`. The `rand.c` file will contain the implementation of `rand()` and `srand()`. The `rand.h` file will contain the declarations (aka prototypes) of `rand()` and `srand()` **AND** must contain the macro `RAND_MAX`, yep you need a macro.

Based on the code in the `man` page, you'd establish `RAND_MAX` to be 32767 (2^15), but we are going to use 2^31. So, your `RAND_MAX` macro should be `(1 << 31)`. Yes, use parenthesis around the shift operation. Do you see in the code in the `man` page where the random value is returned? It does a `% 32767`. You will replace the `32767` with your `RAND_MAX` macro.

Great, you've written your code for `rand()` and `srand()`, but you need to get them into the kernel. They must be callable from within the kernel, so they must be linked with the kernel. Luckily, this is very easy. Up at the top of the `Makefile`, there is a variable called `OBJS`. At the end of the list of `.o` files in the `OBJS` variable, just add `rand.o\` (and don't omit the trailing backslash). Assuming that your `rand.c` and `rand.h` files don't have any compilation errors, simply running `make` should rebuild the `xv6` kernel with your new calls in it.

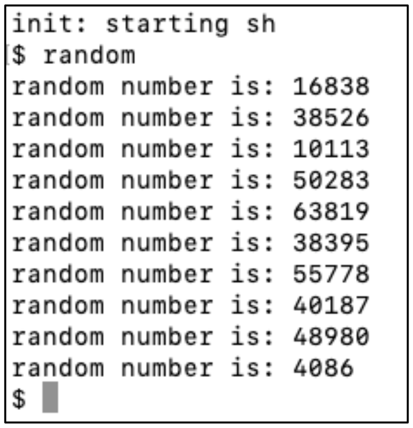
Part 5 – Create the `random` command (10 points)

Since we have a function `rand()`, let's go ahead and create a command that makes use of the `rand()` function. This is a user command in the same way that `cps` and `getppid` are new commands that we've added to `xv6`. Since we already have a file called `rand.c`, we will call our new command `random` and implement it in a file called `random.c`.

Add `rand.o` at the end of the `ULIB` variable in the `Makefile`. This causes all of the `xv6` commands to link with the `rand()` and `srand()` functions.

In `user.h`, include the `rand.h` file.

Write your `random.c` code. Have it loop 10 times and print the pseudo-random numbers generated. The output should look like the image to the right. If you run the random command multiple times, it will always produce the same output. That's what the *pseudo* in pseudo-random means.



```
init: starting sh
$ random
random number is: 16838
random number is: 38526
random number is: 10113
random number is: 50283
random number is: 63819
random number is: 38395
random number is: 55778
random number is: 40187
random number is: 48980
random number is: 4086
$ 
```

Be sure you add `_random` in the `PROGS` variable in the `Makefile`.

Part 6 – Modify the scheduler function (100 pts)

Now you going to make some real changes to the scheduler function. You are going to implement **lottery scheduling**. This would be an excellent time to review/read chapter 9 from the OSTEP book. This is another terrific time to use `#ifdef` blocks in your code to make it easy to go back and forth between a previous working version and a new version. I used `LOTTERY` as the `#define` in my code.

In the `proc.h` file, you are going to define 3 macros: `DEFAULT_NICE_VALUE`, `MAX_NICE_VALUE`, and `MIN_NICE_VALUE`. The values to use for these macros are: 25, 50, and 1. In the `struct proc` data structure, you need to add an additional member, called `nice_value`. The values we are going to use for the lottery scheduling will vary from 1 to 50, with 25 as the default "nice" value. A higher value means that the process has a higher probability to be scheduled. A lower nice value means the process has a lower probability to be scheduled.

Typically, when a process is allocated from them shell, it is assigned the default `nice` value (25 aka `DEFAULT_NICE_VALUE`). **However, when a child process is created via `fork()`, the child process inherits the nice value from its parent process**.

Implementing lottery scheduling is pretty easy. In the `scheduler()` function, sum the nice values for all the `RUNNABLE` processes. Generate a random number (using your brand spanking new random number generator) between 1 and the sum of nice values (put your `mod` hat on, there's a high % you'll need it). Loop through the process table for all `RUNNABLE` processes, summing the nice values (this is different from the initial sum of nice values before). As soon as the sum of nice values exceeds the random number, schedule that process. Easy peasy.

I will warn you about a condition that slowed me down. There will be times when there are not any `RUNNABLE` processes (i.e. the sum of nice values for all `RUNNABLE` processes is zero). When this is true, just keep looping in the scheduler.

Part 7 – Write a new system function called `renice()` (20 pts)

Just as your created system functions for `getppid()`, and `cps()`, you need to create a system function called `renice()`. The `renice()` system function **takes 2 arguments**, a `pid` and a new `nice` value. The process with the given `pid` has its `nice_value` changed to the given new value. Make sure the new `nice` value is between `MAX_NICE_VALUE`, and `MIN_NICE_VALUE`.

If the `nice` value is out of bounds, `renice()` returns a 1 and leaves the nice value of the process unchanged. If the `pid` given to `renice()` does not exist, return a 2. If `renice()` succeeds, return a 0.

Part 8 – Write programs called `renice` and `nice` (20 pts)

The new program `nice` takes 2 command line options: the `nice` value as `argv[1]]` and the name of the program to `exec` as `argv[2]`. It should first set its own `nice` value (using `renice()`) and then `exec` the program on the command line, in `argv[2]`. See the image below. Make sure to pass the rest of `argv` in the call to call to `exec()`.

The program `renice` takes a new `nice` value as `argv[1]` and applies that to **all** the `pid`s following on the command line. See the image below.

The other place to look for information about `nice` and `renice` are the `man` pages. Your implementation of `nice` and `renice` should behave a lot like Unix/Linux commands of the same name.

Part 9 – Modify `ps` to show the `nice` values (10 pts)

Now that you have `nice` values for all your programs and can change them, you need to add the ability to `ps` to view the `nice` values.

The following images on pages 7 and 8 (Example 1 and Example 2) shows you what this should look like.

When you have processes with large differences in `nice` values, you should notice that the processes with large `nice` values get scheduled more frequently and accumulate more time/ticks on the CPU. If you don't notice this, something is wrong with your implementation.

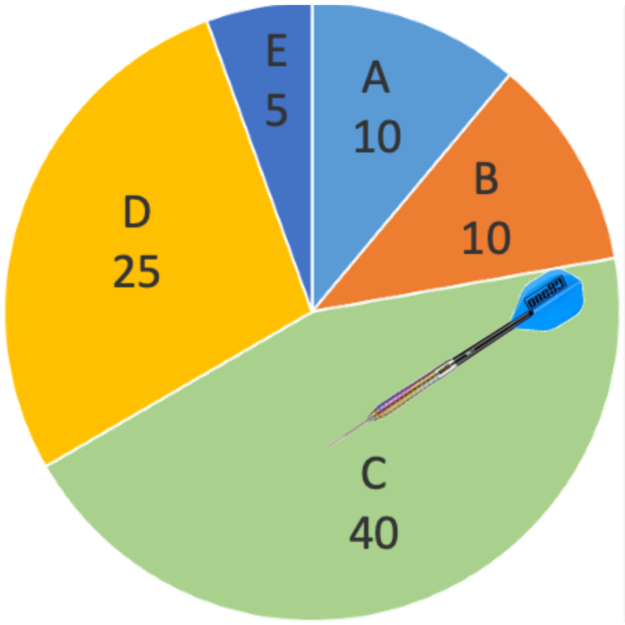
Submit to `TEACH`

When you are done with the `Lab3`, submit your code to `TEACH`. Remember how we used the command `"make teach"` to produce a `tar` and gzipped file that you can submit into `TEACH`? Do that and be done. If you don't have a `TEACH` account yet, it is very easy to have one created.

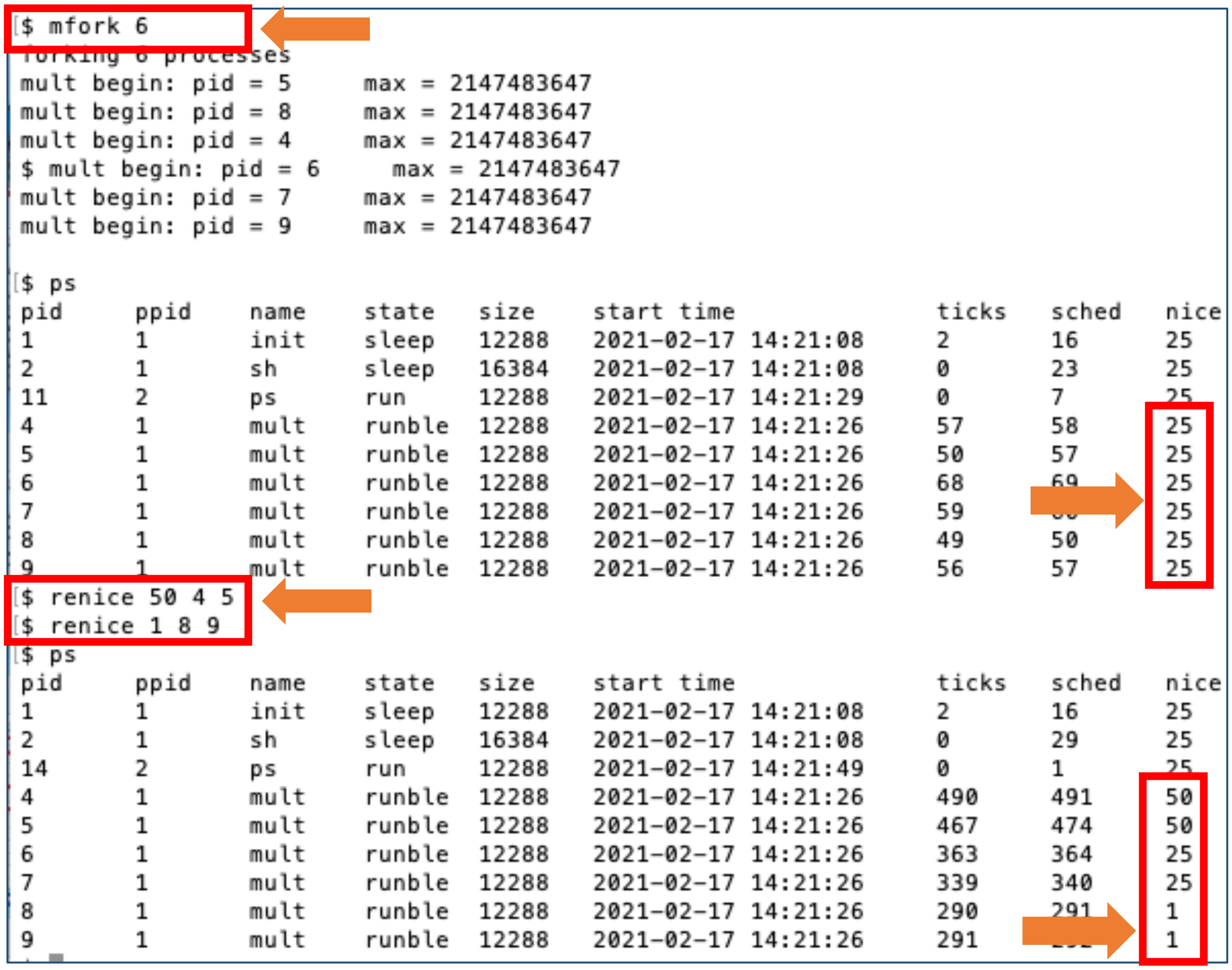
Refresher

You may want to review the slides from lecture 2 (Scheduling, it's in week 2). At the end, there is some discussion of and images for implementation of the Lottery Scheduler.

This of the nice values as the size of the area for each process in a dart target. When you throw the dart, assuming your aim is as good as is mine (completely random), you are more likely to land the dart in process C. however, you will also occasionally land in the other processes. The likelihood of landing in a given process is based on the nice value (the area for the process).



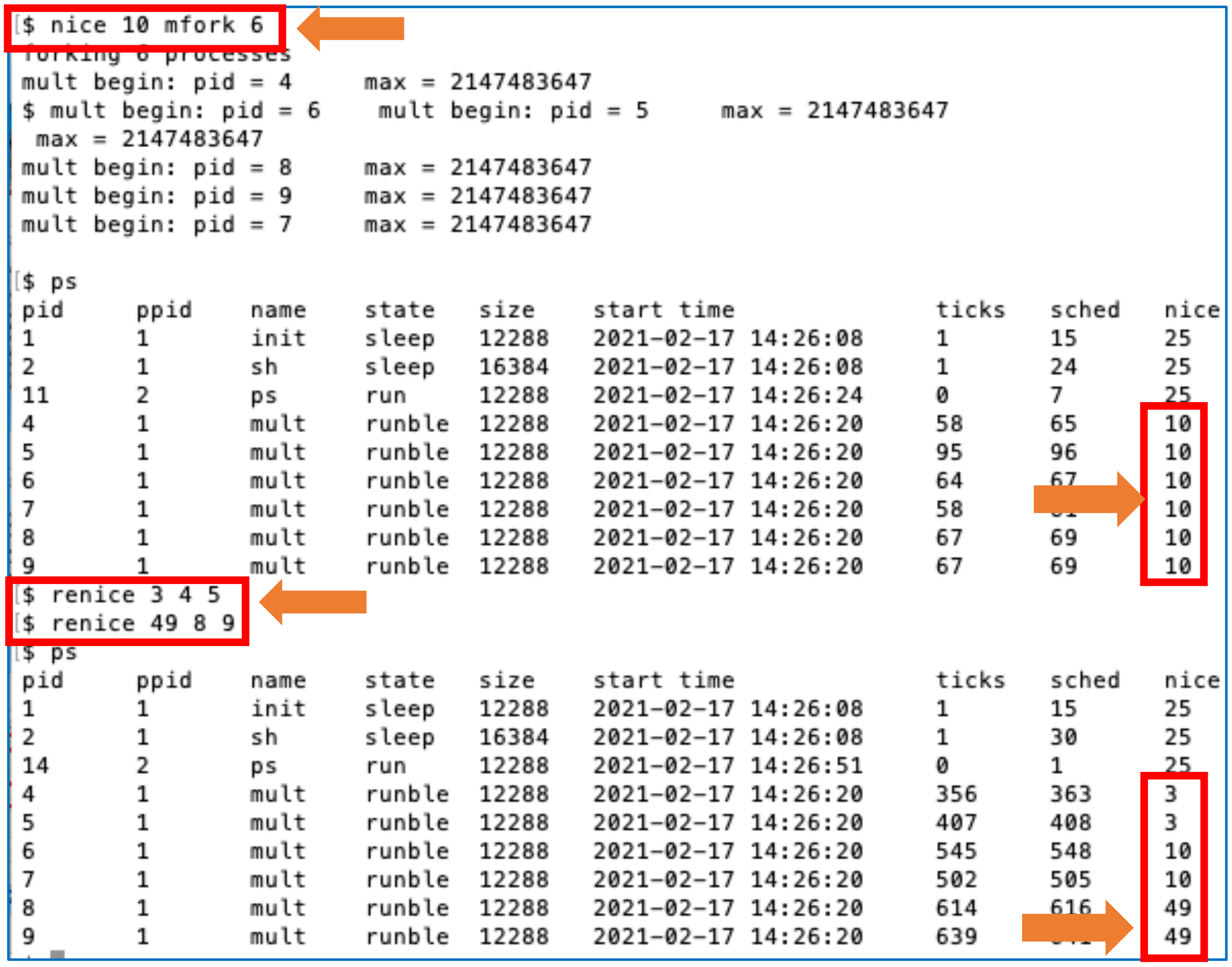
Example 1: Here, `mfork` started with the default *nice* value (25) and all the `mult` processes it created, inherited that same nice value. We then change the *nice* values of 4 of them with 2 different uses of the `renice` command.



```
[$ mfork 6
forking 6 processes
mult begin: pid = 5      max = 2147483647
mult begin: pid = 8      max = 2147483647
mult begin: pid = 4      max = 2147483647
$ mult begin: pid = 6      max = 2147483647
mult begin: pid = 7      max = 2147483647
mult begin: pid = 9      max = 2147483647

[$ ps
pid      ppid     name     state    size     start time               ticks    sched    nice
1        1        init     sleep    12288    2021-02-17 14:21:08      2        16       25
2        1        sh       sleep    16384    2021-02-17 14:21:08      0        23       25
11       2        ps       run      12288    2021-02-17 14:21:29      0        7        25
4        1        mult     runble   12288    2021-02-17 14:21:26      57       58       25
5        1        mult     runble   12288    2021-02-17 14:21:26      50       57       25
6        1        mult     runble   12288    2021-02-17 14:21:26      68       69       25
7        1        mult     runble   12288    2021-02-17 14:21:26      59       60       25
8        1        mult     runble   12288    2021-02-17 14:21:26      49       50       25
9        1        mult     runble   12288    2021-02-17 14:21:26      56       57       25
[$ renice 50 4 5
[$ renice 1 8 9
[$ ps
pid      ppid     name     state    size     start time               ticks    sched    nice
1        1        init     sleep    12288    2021-02-17 14:21:08      2        16       25
2        1        sh       sleep    16384    2021-02-17 14:21:08      0        29       25
14       2        ps       run      12288    2021-02-17 14:21:49      0        1        25
4        1        mult     runble   12288    2021-02-17 14:21:26      490      491      50
5        1        mult     runble   12288    2021-02-17 14:21:26      467      474      50
6        1        mult     runble   12288    2021-02-17 14:21:26      363      364      25
7        1        mult     runble   12288    2021-02-17 14:21:26      339      340      25
8        1        mult     runble   12288    2021-02-17 14:21:26      290      291      1
9        1        mult     runble   12288    2021-02-17 14:21:26      291      292      1
```

Example 2: Notice that when `mfork` is created using the `nice` command, all the processes are started with the *nice* value of the parent. In this case, the parent process created with the `nice` command, `mfork`, had a *nice* value of 10, so all the `mult` processes were started with a *nice* value of 10.



```
[$ nice 10 mfork 6
forking 6 processes
mult begin: pid = 4      max = 2147483647
$ mult begin: pid = 6     mult begin: pid = 5      max = 2147483647
 max = 2147483647
mult begin: pid = 8      max = 2147483647
mult begin: pid = 9      max = 2147483647
mult begin: pid = 7      max = 2147483647

[$ ps
pid      ppid     name     state    size     start time               ticks    sched    nice
1        1        init     sleep    12288    2021-02-17 14:26:08      1        15       25
2        1        sh       sleep    16384    2021-02-17 14:26:08      1        24       25
11       2        ps       run      12288    2021-02-17 14:26:24      0        7        25
4        1        mult     runble   12288    2021-02-17 14:26:20      58       65       10
5        1        mult     runble   12288    2021-02-17 14:26:20      95       96       10
6        1        mult     runble   12288    2021-02-17 14:26:20      64       67       10
7        1        mult     runble   12288    2021-02-17 14:26:20      58       61       10
8        1        mult     runble   12288    2021-02-17 14:26:20      67       69       10
9        1        mult     runble   12288    2021-02-17 14:26:20      67       69       10
[$ renice 3 4 5
[$ renice 49 8 9
[$ ps
pid      ppid     name     state    size     start time               ticks    sched    nice
1        1        init     sleep    12288    2021-02-17 14:26:08      1        15       25
2        1        sh       sleep    16384    2021-02-17 14:26:08      1        30       25
14       2        ps       run      12288    2021-02-17 14:26:51      0        1        25
4        1        mult     runble   12288    2021-02-17 14:26:20      356      363      3
5        1        mult     runble   12288    2021-02-17 14:26:20      407      408      3
6        1        mult     runble   12288    2021-02-17 14:26:20      545      548      10
7        1        mult     runble   12288    2021-02-17 14:26:20      502      505      10
8        1        mult     runble   12288    2021-02-17 14:26:20      614      616      49
9        1        mult     runble   12288    2021-02-17 14:26:20      639      641      49
```

Final note

The labs in this course are intended to give you basic skills. In later labs, we will ***assume*** that you have mastered the skills introduced in earlier labs. **If you don't understand, ask questions.**