Programming Assignment 2: Linux System Calls
CSE 3320.002/900 and CSE 3320.003/901
Due: October 21, 2020 5:30PM

Description

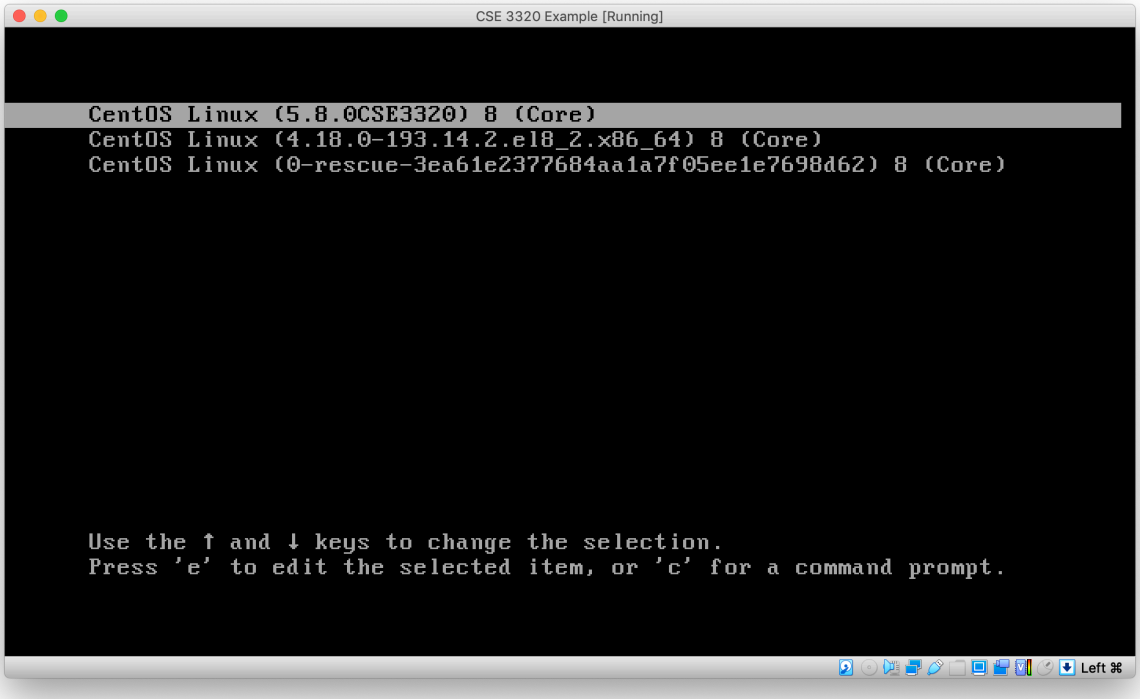
In this assignment you will be become familiar with compiling and installing a new kernel and adding new system calls to the Linux operating system. In addition you will also gain experience moving memory from user space to kernel space and back.

Updating your kernel source

Before you begin the assignment you must get the latest source updates from github.

1. Boot your VM.

2. Select Centos Linux 4.18.0-193.14.2.el8_2.x86_64. (The second in the screenshot below) Make sure this selection does not contain cse3320. This is your default vanilla kernel. You will always be able to boot into this kernel as a recovery
3. Wait for the OS to boot and login using the password `cse3320` .



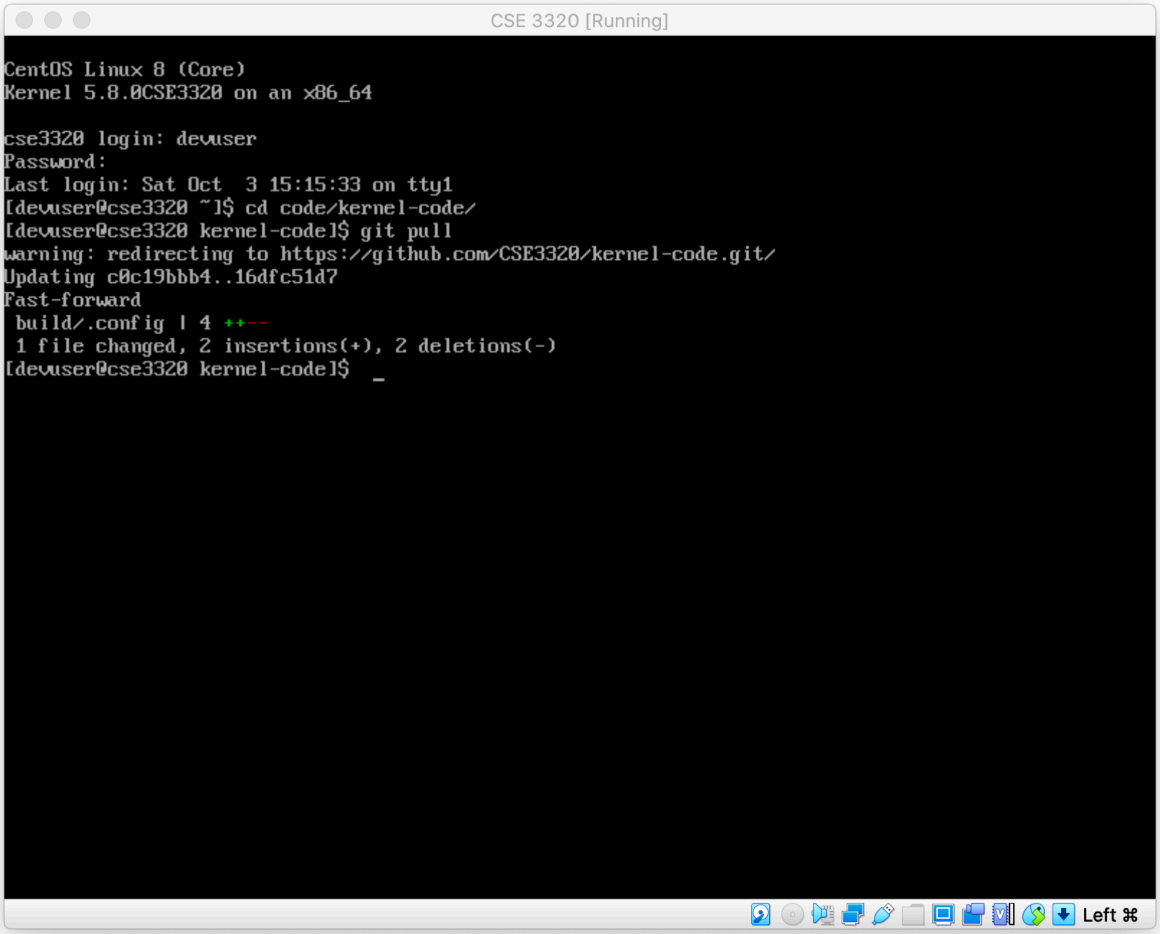
```
      CentOS Linux (5.8.0CSE3320) 8 (Core)
      CentOS Linux (4.18.0-193.14.2.el8_2.x86_64) 8 (Core)
      CentOS Linux (0-rescue-3ea61e2377684aa1a7f05ee1e7698d62) 8 (Core)





      Use the ↑ and ↓ keys to change the selection.
      Press 'e' to edit the selected item, or 'c' for a command prompt.
```

4. Change to the kernel source directory: `cd ~/code/kernel-code`
5. Get the latest kernel updates: `git pull`



```
CentOS Linux 8 (Core)
Kernel 5.8.0CSE3320 on an x86_64

cse3320 login: devuser
Password:
Last login: Sat Oct  3 15:15:33 on tty1
[devuser@cse3320 ~]$ cd code/kernel-code/
[devuser@cse3320 kernel-code]$ git pull
warning: redirecting to https://github.com/CSE3320/kernel-code.git/
Updating c0c19bbb4..16dfc51d7
Fast-forward
 build/.config | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
[devuser@cse3320 kernel-code]$ _
```

Compiling and Installing the kernel

6. Change to the kernel source directory by typing:

```
cd ~/code/kernel-code/linux-5.8
```

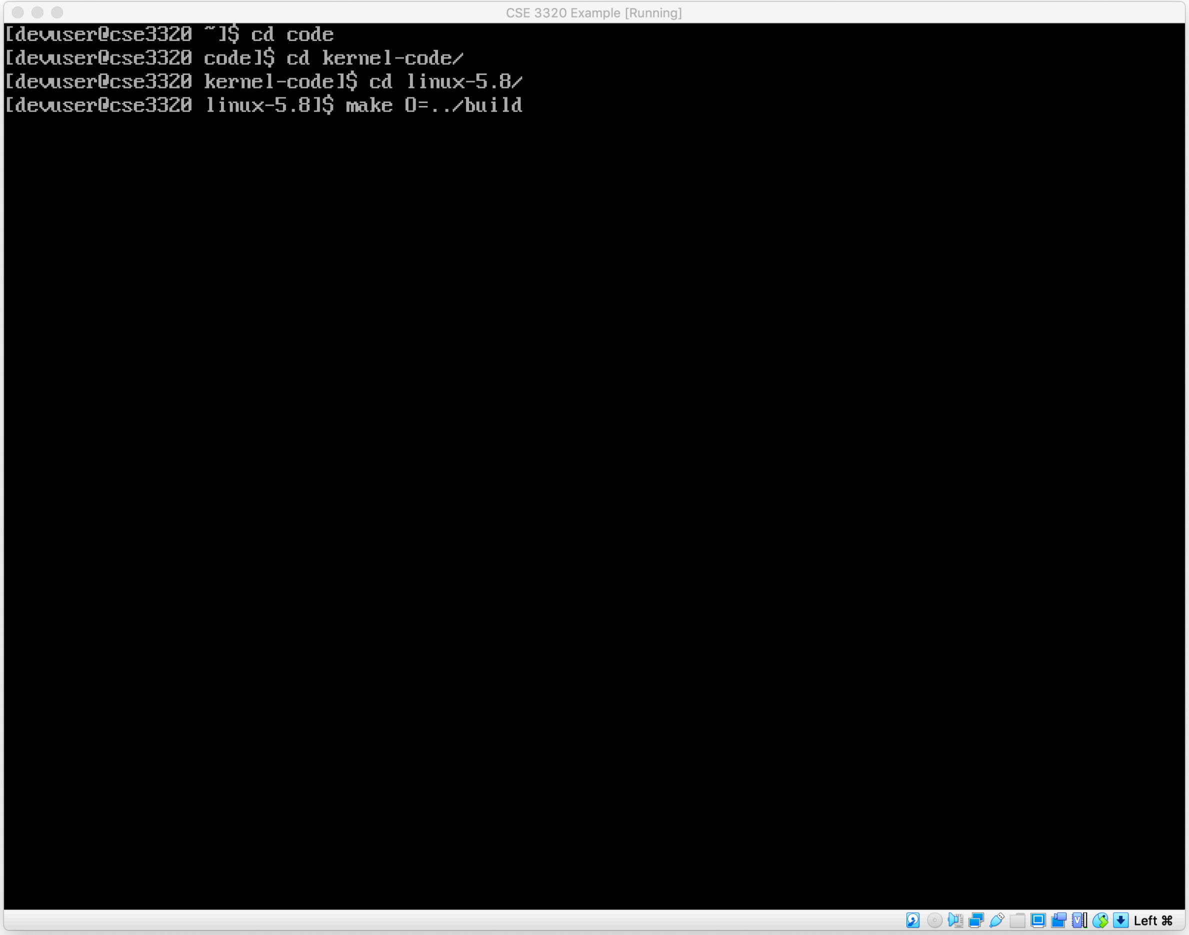
We will be building our kernel out-of-tree. This will allow us to keep our object files separate from our source code.

7. To build the kernel type:

```
make O=../build
```

NOTE: That is a capital letter "O" and not a numeral "0".

The first time you build the kernel it can take you a very long time to finish the compilation. Subsequent builds of the kernel will not take as long.



```
[devuser@cse3320 ~]$ cd code
[devuser@cse3320 code]$ cd kernel-code/
[devuser@cse3320 kernel-code]$ cd linux-5.8/
[devuser@cse3320 linux-5.8]$ make O=../build
```

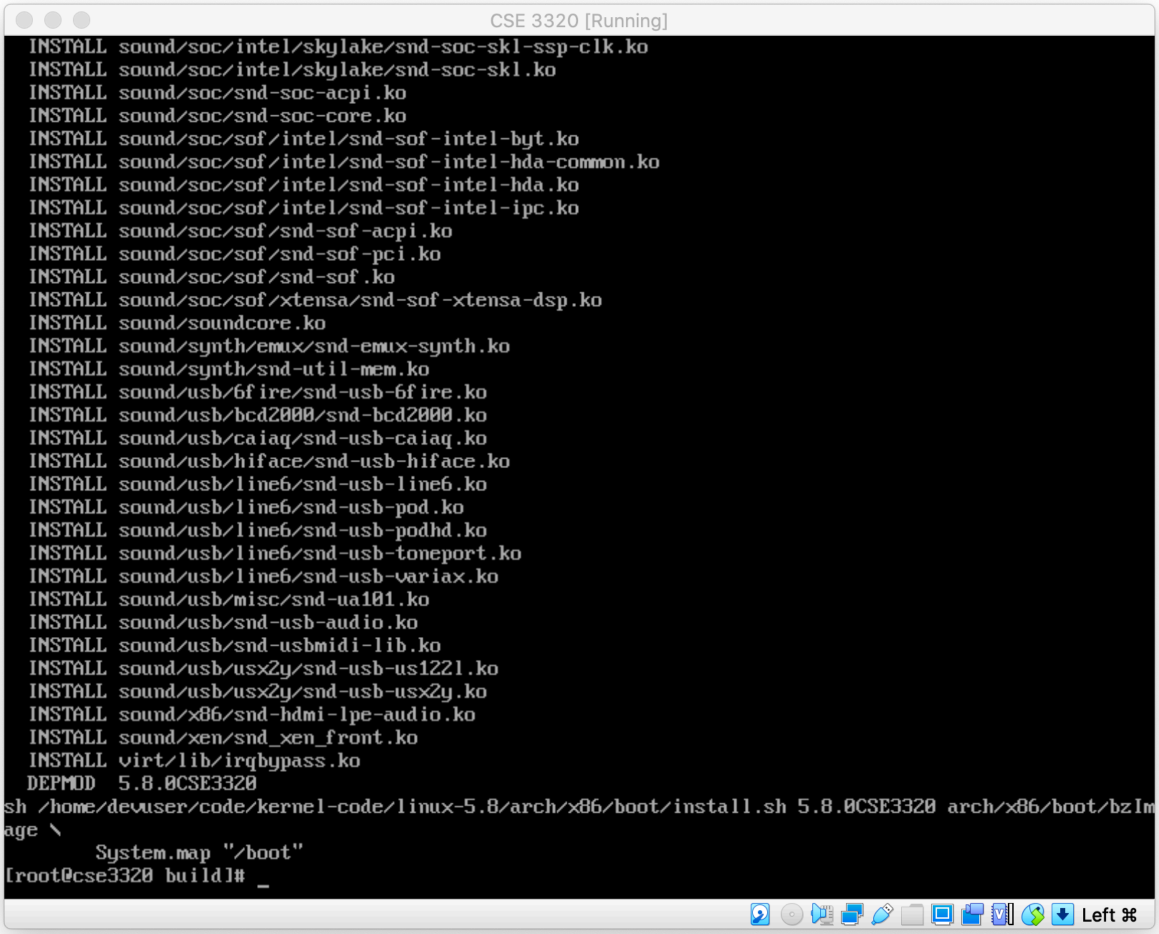
Once the build is done change to superuser by typing: `su` and entering the password `cse3320`

Change to the build directory where all your object files and the new kernel image are located by typing:

```
cd ../build
```

Install your new kernel by typing:

```
make modules_install install
```

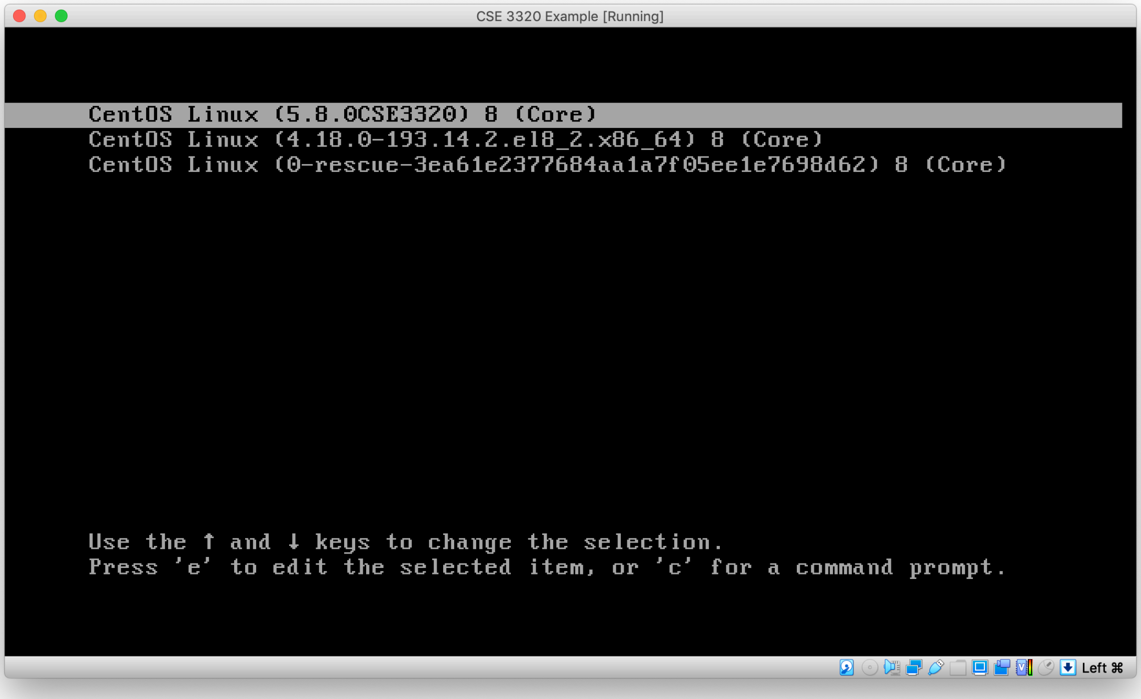


```
  INSTALL sound/soc/intel/skylake/snd-soc-skl-ssp-clk.ko
  INSTALL sound/soc/intel/skylake/snd-soc-skl.ko
  INSTALL sound/soc/snd-soc-acpi.ko
  INSTALL sound/soc/snd-soc-core.ko
  INSTALL sound/soc/sof/intel/snd-sof-intel-byt.ko
  INSTALL sound/soc/sof/intel/snd-sof-intel-hda-common.ko
  INSTALL sound/soc/sof/intel/snd-sof-intel-hda.ko
  INSTALL sound/soc/sof/intel/snd-sof-intel-ipc.ko
  INSTALL sound/soc/sof/snd-sof-acpi.ko
  INSTALL sound/soc/sof/snd-sof-pci.ko
  INSTALL sound/soc/sof/snd-sof.ko
  INSTALL sound/soc/sof/xtensa/snd-sof-xtensa-dsp.ko
  INSTALL sound/soundcore.ko
  INSTALL sound/synth/emux/snd-emux-synth.ko
  INSTALL sound/synth/snd-util-mem.ko
  INSTALL sound/usb/6fire/snd-usb-6fire.ko
  INSTALL sound/usb/bcd2000/snd-bcd2000.ko
  INSTALL sound/usb/caiaq/snd-usb-caiaq.ko
  INSTALL sound/usb/hiface/snd-usb-hiface.ko
  INSTALL sound/usb/line6/snd-usb-line6.ko
  INSTALL sound/usb/line6/snd-usb-pod.ko
  INSTALL sound/usb/line6/snd-usb-podhd.ko
  INSTALL sound/usb/line6/snd-usb-toneport.ko
  INSTALL sound/usb/line6/snd-usb-variax.ko
  INSTALL sound/usb/misc/snd-ua101.ko
  INSTALL sound/usb/snd-usb-audio.ko
  INSTALL sound/usb/snd-usbmidi-lib.ko
  INSTALL sound/usb/usx2y/snd-usb-us122l.ko
  INSTALL sound/usb/usx2y/snd-usb-usx2y.ko
  INSTALL sound/x86/snd-hdmi-lpe-audio.ko
  INSTALL sound/xen/snd_xen_front.ko
  INSTALL virt/lib/irqbypass.ko
  DEPMOD  5.8.0CSE3320
sh /home/devuser/code/kernel-code/linux-5.8/arch/x86/boot/install.sh 5.8.0CSE3320 arch/x86/boot/bzIm
age \
        System.map "/boot"
[root@cse3320 build]# _
```

Reboot your VM by typing:

```
reboot
```

8. An the GRUB screen select your new kernel “5.8.0CSE3320”



```
      CentOS Linux (5.8.0CSE3320) 8 (Core)
      CentOS Linux (4.18.0-193.14.2.el8_2.x86_64) 8 (Core)
      CentOS Linux (0-rescue-3ea61e2377684aa1a7f05ee1e7698d62) 8 (Core)






      Use the ↑ and ↓ keys to change the selection.
      Press 'e' to edit the selected item, or 'c' for a command prompt.
```

Getting the test programs for Assignment 1

The course github page has two test programs to be used to test your code for this assignment.

9. Open a terminal and change to your home directory, if you are not there, by typing:

```
cd ~
```

10. Grab the source from github by typing:

```
git clone https://github.com/CSE3320/System-Call-Test-Case
```

Part 1: hello System Call (10 points)

At this point, you know how to compile and install a new kernel image. The image you created while compiling the kernel is exactly the same as the original kernel. In part one of this assignment you will add a simple system call (hello) to the kernel, that receives no parameters and prints your name that gets logged with the kernel messages and returns 0. Although it is is simple it illustrates the kernel system call mechanism and the interaction between user programs and the kernel.

1. Modify the kernel system call table so that it can call your new system call. You will add an entry for your new system call to the kernel's system call table using number 548.

2. Edit the file `arch/x86/entry/syscalls/syscall_64.tbl` and add a line after entry 547 for your new system call. NOTE: you must use tabs between the columns:

```
548        64         hello      sys_hello
```

The first column is the system call number. You should also choose the next available number, which in our case is 440. The second column says that this system call is common is for 64-bit CPUs since our VM is 64-bit. The third column is the name of the system call, and the fourth is the name of the function implementing it. By convention this is the syscall name, prefixed by sys_.

3. Edit the file `include/linux/syscalls.h`. Around line 1121 after the sys_fork(), add a line declaring your new system call:

```
asmlinkage long sys_hello( void );
```

4. Now you are ready to write your system call. Edit the file `kernel/sys.c` and add an entry for your function:

```
SYSCALL_DEFINE0( hello )
{
     printk( KERN_WARNING "YOUR NAME  YOUR ID\n");
     return 0;
}
```

Replace YOUR NAME and YOUR ID with your name and ID.

SYSCALL_DEFINEN is a family of macros that make it easy to define a system call with N arguments. The first argument to the macro is the name of the system call (without sys_ prepended to it). The remaining arguments are pairs of type and name for the parameters. Since our system call has one argument, we use SYSCALL_DEFINE0. In part two you will use SYSCALL_DEFINE2.[1]

5. Build and install your new kernel and reboot the VM.

Once your system is rebooted, test your system call with the name.c program in System-Call-Test-Case directory.

6. Compile name.c with:

```
gcc name.c -o name
```

[1] https://brennan.io/2016/11/14/kernel-dev-ep3/

7. Run the program with:

```
./name
```

8. Verify your program output your name in the kernel log by typing:

```
dmesg
```

The kernel log will be output on the console and you should see output similar to below:

```
[ 4150.494039] Trevor Bakker 1000xxxxxx
```

Part 2: Add a System Call to Collect Process Info (90 points)

In the second part of the assignment you are going to add a more useful system call into the Linux kernel. This system call will allow a program running in user-mode to get detailed information about a certain process such as parent PID, process state, priority, etc. This system call, called `procstat` will take a process id (PID) and a pointer to a proc_stat struct as argument. The system call will access the process control block (PCB) of the process whose PID was passed as argument and will fill in the `proc_stat` data structure with the corresponding values in the process PCB. In Linux the process PCB is called task_struct and is defined in file `~/kernel-code/linux-5.8/include/linux/sched.h`). Your system call should return 0 if process_info was successfully filled in. Otherwise it should return the following error codes:

- ESRCH ("No such process"), if a process with the given PID does not exist.
- EINVAL ("Invalid argument"), if there are errors while filling in data structure process_info.
- EFAULT ("Bad Address") if there is an error writing to user space. Continue reading and you will understand how this error could occur.

In include/linux/ create a file called `procstat.h` in this file add the following code:

```
#ifndef __PROC_STAT__
#define __PROC_STAT__

struct proc_stat {
  int pid;
  int parent_pid;
  long user_time;
  long sys_time;
  long state;
  unsigned long priority;
  unsigned long normal_priority;
  unsigned long static_priority;
  unsigned long rt_priority;
  int time_slice;
  unsigned policy;
  unsigned long num_context_switches;
  unsigned long task_size;
  unsigned long total_pages_mapped;
  char name[255];
};

#endif
```

Define a new system call, `procstat`, for system call number `549.` Your system call will take two parameters, e.g. use `SYSCALL_DEFINE2.`

The first parameter of the system call shall be a PID. The second parameter shall be a pointer to a `proc_stat` struct`.` You will need to access the fields of the `task_struct` for the process whose PID was passed as a parameter to fill in the 13 fields of the `proc_stat` data structure.

You will find that the function `find_task_by_pid_ns` defined in `include/linux/sched.h`. NOTE: The first parameter to `find_task_by_pid_ns` is the PID of the process to find. The second parameter is `&init_pid_ns` .

Note that not all field values in your struct `proc_stat` will exactly match the names of fields in the `task_struct`, so you may need to read through some code before you get the right values for these fields. Some of the items may be in structures pointed to by the `task_struct` . Other fields like num_context_switches should be a combination of voluntary and involuntary context switch values.

The code for your function should be in `kernel/sys.c` as with your hello function and you will need a declaration in `syscalls.h.`

After filling in your `proc_stat` structure with values from `task_struct` you will need to call `copy_to_user` to pass the return values from kernel space to user space.

After implementing your function, compile, install your kernel and reboot your VM.

Testing Your Code

In the `System-Call-Test-Case` directory type:

```
make
```

This will build the two test applications.

Start the test application, `prio`, and pass in a priority. It will output it's PID which you can use to pass into the `proc_stat_test` function.

Submitting your Assignment

You will be submitting your code as a gzipped tarball. There is a python package_changes.py that will find all your source code change and create the tarball for you.

```
cd ~/code/kernel-code/linux-5.8
./package_changes.py —user [your netid] —assignment 2
```

NOTE: two dashes before user and assignment. You can then use sftp to transfer the file from your VM to your host OS and submit it via Canvas.

Grading

The assignment will be graded out of 100 points. Code that does not compile will earn 0. Your header file, `procstat.h`, must be in the correct spot and the structure defined precisely as above or it will result in a 0. Points will be deducted for missing functionality as well as submissions that do not follow the submission guidelines.

Programs will be graded with, at minimum, the testing programs in System-Call-Test-Case. Please be sure your code changes are consistent with them and do not break them. Kernel panics will result in large point deductions even if it performs other functionality correctly.

Part 1 Grading Rubric

Requirement	Points Possible
Name / ID	8
No extra debug output	2

Part 2 Grading Rubric

Requirement	Points Possible
Each status field	6 per field
No extra debug output	10

Administrative

This assignment must be coded in C. Any other language will result in 0 points.

Your programs will be compiled and graded on a stock cse3320 VM.

Your gzipped tarball is to be turned in via Canvas. Submission time is determined by the blackboard system time. You may submit your programs as often as you wish. Only your last submission will be graded.