

# COL331 Assignment 1

Aniruddha Deb  
2020CS10869

February 2023

## 1 Install Linux

### 1.1 MacOS

The first attempt of the assignment was using QEMU on my Mac. Ubuntu Server 22.04 was used, and the initial 20GB disk was partitioned into a 1GB boot and 19GB root partition. The kernel images were kept in boot and everything else went in the root partition. Installing linux was done by downloading the kernel tarball (downloads from the git repo were very slow), decompressing it and following the instructions as mentioned in the assignment. An extra parameter, `CONFIG_SYSTEM_REVOCATION_KEYS`, also had to be set to `""` to let the kernel compile. All other parameters were kept as is, and defaults were used.

The kernel took around 6-7 hours to compile (on a dual core virtualized machine), and I realized that this approach was infeasible if I was to actually develop on the kernel.

### 1.2 GCP

After slow compile times on my Mac, I decided to move to Google Cloud Platform. They offer \$300 worth of free credit, and their compute VM's support ubuntu server 22.04.1

On an eight core E2-highcpu instance, the kernel compiled in around 35-40 minutes. Installing modules and the kernel itself took another 5-10 minutes. The only issue was that you couldn't see the grub screen on boot, so the installed kernel had to be made the default. This is done by default when you run the `make install` command, so nothing had to be changed.

Running `uname -r` on the VM after bootup shows the kernel as 6.1.6, showing that the new kernel was successfully installed (default kernel is 5.15.6)

#### Contingency Management

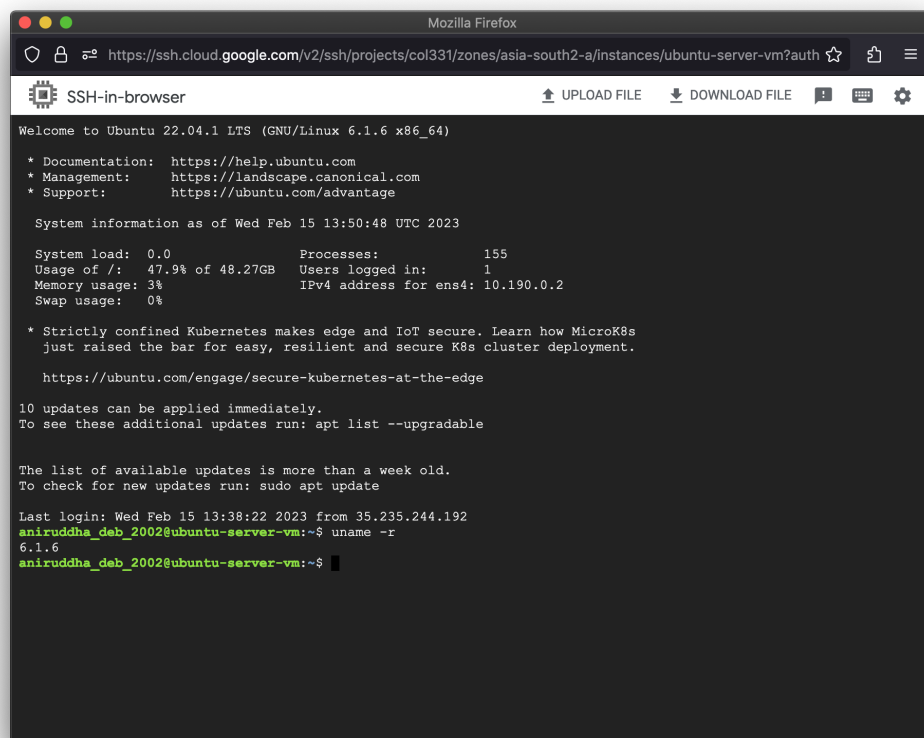
Not having access to grub takes away the ability to boot into 5.15 if something goes wrong, such as if you changed a part of the bootloader code and the kernel won't boot, and neither can you SSH onto it to edit it. In this case, GCP provides recovery instructions. The gist of the recovery instructions are that you need to create a new boot disk, boot the VM using that boot disk, mount the old boot disk manually and then do recovery, or tweak the grub config on the old boot disk.

## Ramdisk compression

The initial image that the bootloader loads into memory (before the filesystem is initialized) is called a **ramdisk**. This is made using the kernel image and the `mkinitramfs` command, which compresses the image and a few modules together, which are then loaded into memory. The ramdisk generated by 6.1.6 compilation with all the default modules takes up a lot of space (500 MB). This can be reduced by not bundling in most of the modules, but only the dependent ones by changing the `MODULES` parameter in `/etc/initramfs-tools/initramfs.conf` to `dep` so that the compressor guesses which modules to include.

## Kernel Configuration

Copying and pressing enter for all the options is an inefficient way to configure the kernel, as most of the modules that it would compile are not used by the VM (eg filesystems, display drivers, network adapters, device drivers). You can run `make menuconfig` to launch a GUI that lets you configure which parts of the linux kernel to compile. This reduces the size of the kernel source on disk after build, and also reduces kernel compilation times by 50-60%.



```

Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 6.1.6 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Wed Feb 15 13:50:48 UTC 2023

System load:  0.0               Processes:    155
Usage of /:   47.9% of 48.27GB   Users logged in: 1
Memory usage: 3%               IPv4 address for ens4: 10.190.0.2
Swap usage:   0%

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.

https://ubuntu.com/engage/secure-kubernetes-at-the-edge

10 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Wed Feb 15 13:38:22 2023 from 35.235.244.192
aniruddha_deb_2002@ubuntu-server-vm:~$ uname -r
6.1.6
aniruddha_deb_2002@ubuntu-server-vm:~$
```

Figure 1: Google Compute Engine VM (8 cores, 8 GB RAM) on compiled kernel

## 2 System Calls

For adding system calls, [The kernel.org documentation](#) and [Stephen Brennan's Tutorial](#) were used. The first edit was to add the structures used to `include/linux/pid.h`, so that they can be used across the system.

After this, the system call tables were edited to add entries for the three system calls (note that we call `register` as `registr` as the former is a reserved C keyword). For symmetry, `deregistr` is also similarly named.

In `syscall_32.tbl`:

```
1 451 i386      registr    sys_registr
2 452 i386      fetch      sys_fetch
3 453 i386      deregistr   sys_deregistr
```

In `syscall_64.tbl`:

```
1 548 common    registr    sys_registr
2 549 common    fetch      sys_fetch
3 550 common    deregistr   sys_deregistr
```

In `unistd.h` (`__NR_SYSCALLS` was also updated):

```
1 #define __NR_registr 451
2 __SYSCALL(__NR_registr, sys_registr)
3 #define __NR_fetch 452
4 __SYSCALL(__NR_fetch, sys_fetch)
5 #define __NR_deregistr 453
6 __SYSCALL(__NR_deregistr, sys_deregistr)
```

The syscalls were then linked in `syscalls.h`

```
1 asmlinkage int sys_registr(pid_t pid);
2 asmlinkage int sys_fetch(struct pid_ctxt_switch *stats);
3 asmlinkage int sys_deregistr(pid_t pid);
```

Finally, the actual syscalls were implemented in a new file, `kernel/ctx_switch.c`

```
1 SYSCALL_DEFINE1(registr, pid_t, pid) { ... }
2 SYSCALL_DEFINE1(fetch, struct pid_ctxt_switch*, pid) { ... }
3 SYSCALL_DEFINE1(deregistr, pid_t, pid) { ... }
```

And this file is added to the Makefile in the `kernel` directory.

```
1 obj-y += ctx_switch.o
```

## 2.1 registr

The `registr` syscall is implemented using a linked list: after the appropriate range checks for the `pid`, all we do is append a node to the linked list with the appropriate PID when the system call is called. The new node is allocated in kernel space, using `kmalloc`.

```
1 tmp = kmalloc(sizeof(*tmp), GFP_KERNEL);
2 tmp->pid = pid;
3
4 list_add_tail(&tmp->next_prev_list, &pid_ll);
```

GFP stands for *Get Free Pages*. More information on kernel memory allocation was taken from [this reference](#).

### Limiting the number of pids

A possible security vulnerability here is that a user may spuriously add several billion pid's and use up all the kernel memory. The implementation takes care of this by setting a limit on the number of pid's that can be kept track of (currently 256)

## 2.2 fetch

The `fetch` syscall iterates over all the members of the linked list using the `list_for_each_entry` macro, obtaining the task struct corresponding to the given pid. If the struct exists, it accumulates the `nvcsw` and `nivcsw` values to a local variable. It then copies the values in the local variable to user memory using `copy_to_user`.

```
1 struct pid_ctxt_switch pid_stats = {.nvctx = 0, .ninvctx = 0};
2 struct pid_node *n;
3
4 list_for_each_entry(n, &pid_ll, next_prev_list) {
5     // if task exists, add stats to pid_stats
6 }
7
8 if (copy_to_user(pid, &pid_stats, sizeof(pid_stats))) return -EINVAL;
```

### Reallocated pid's?

If a pid registered with this syscall is reallocated, then we may obtain spurious numbers. I couldn't find a way to find out if a pid has been reallocated, so this bug exists.

## 2.3 deregistr

The `deregistr` syscall looks for the given pid in the list. If the pid is found, it deletes it and decrements the number of pids registered by one.

```
1 struct pid_node *n;
2 char found = 0;
3
4 list_for_each_entry(n, &pid_ll, next_prev_list) {
5     // if pid matches, found = 1 and break
6 }
7
```

```

8 if (found) {
9     // remove n from pid_ll, free memory for the node
10    // and decrement n_pids by 1
11 }

```

### Synchronization?

When SMP was first introduced, the entire kernel was locked using a Big Kernel Lock (BKL). Now, synchronization in the kernel is more fine-grained. I couldn't find an authoritative resource to check if system calls are multithreaded or not, and if race conditions can originate if, say two users call the system call (one calls fetch and one calls register). Since performance is of most importance, I refrained from locking the list here.

## 2.4 Testing

A test was written to test the three syscalls and placed in the tools/testing/selftests directory. This test takes in pids as command line arguments, adds all the pids passed via the registr syscall, fetches the total voluntary and involuntary context switches and finally deregisters all the registered pids.

```

1 int *pids = (int*)malloc(sizeof(int)*(argc-1));
2 char *p;
3 for (int i=1; i<argc; i++) {
4     pids[i-1] = (int)strtol(argv[i], &p, 10);
5     if (syscall(548, pids[i-1]) != 0) {
6         // error
7         printf("ERROR: could not register process %d, syscall returned
8             %d\n", pids[i-1], errno);
9     }
10 }
11 printf("Registered all processes\n");
12
13 // fetch the changes
14 struct pid_ctxt_switch *ctx_switch = (struct
15     pid_ctxt_switch*)malloc(sizeof(struct pid_ctxt_switch));
16 if (syscall(549, ctx_switch) != 0) {
17     printf("ERROR: could not fetch context switches, syscall returned %d\n",
18         errno);
19 }
20
21 // print the switches
22 printf("Voluntary changes: %lu\n", ctx_switch->nvctx);
23 printf("Involuntary changes: %lu\n", ctx_switch->ninvctx);
24
25 // deregister processes
26 for (int i=0; i<argc-1; i++) {
27     if (syscall(550, pids[i]) != 0) {
28         printf("ERROR: could not deregister pid %d, syscall returned %d\n",
29             pids[i], errno);
30     }
31 }
32 printf("Deregistered all processes\n");

```

## 3 Kernel Module

The kernel module was separately implemented in the `sig_mod.c` file, outside the kernel. The module uses a linked list to store (pid,signal) pairs whose size is limited to 256 pairs (to avoid the overflow described previously). Every second, the module iterates over all the items in the list, and sends the appropriate signal to the corresponding pid.

On being loaded, the module creates the procfile, initializes the pid list mutex and sets the timer to execute after one second. On being removed, the module cleans up all nodes in the list, deletes the timer (synchronously, which waits if a timer event is running to complete) and then destroys the list mutex

### 3.1 Proc file I/O

Newer iterations of the linux kernel feature a newer proc filesystem API, which is not particularly well documented. I used [this reference](#) for writing the write and read handlers, and seeing how to register/deregister proc files with the kernel.

```
1 static struct proc_dir_entry *sig_target;
2
3 static ssize_t sig_tgt_read(struct file *file, char __user *ubuf, size_t
  count, loff_t *ppos) {
4     char buf[BUFSIZE];
5     int len=0;
6     if(*ppos > 0 || count < BUFSIZE)
7         return 0;
8     // write to buffer here using sprintf
9     if(copy_to_user(ubuf,buf,len))
10         return -EFAULT;
11     *ppos = len;
12     return len;
13 }
14
15 static ssize_t sig_tgt_write(struct file *file, const char __user *ubuf,
  size_t count, loff_t *ppos) {
16     int num,c;
17     char buf[BUFSIZE];
18     if(*ppos > 0 || count > BUFSIZE)
19         return -EFAULT;
20     if(copy_from_user(buf, ubuf, count))
21         return -EFAULT;
22     // read from buf here using sscanf
23     // create a node and add it to the list using the values read
24     c = strlen(buf);
25     *ppos = c;
26     return c;
27 }
28
29 static const struct proc_ops sig_target_ops = {
30     .proc_write = sig_tgt_write,
31     .proc_read = sig_tgt_read
32 };
33
34 // register sig_target = proc_create(PROCFILE_NAME, 00622, NULL,
  &sig_target_ops);
```

This is safe from buffer overflows.

## Procfile permissions

The permissions for the procfile are set to **0622**. This gives **only root** the ability to read the registered signals, and not users. This prevents one user from seeing the signals and processes registered by another user, making this module secure.

## 3.2 Timer

A 1 second timer is used to periodically send signals to the registered processes. The implementation details were taken from *Linux Kernel Development, 3rd ed.* by Robert Love (chapter 11). The timer takes a function to callback while being defined, and then is set to be called at a certain duration in the future.

```
1 void send_signals(struct timer_list* timer) {
2
3     // lock mutex
4     // for each entry, send a signal
5     //     If the task_struct for the entry is not found, delete it
6     // unlock mutex
7
8     // so that the timer is again called after 1s
9     mod_timer(timer, jiffies+DELAY);
10 }
11
12 DEFINE_TIMER(timer, send_signals);
```

The signals themselves are sent using the `send_sig_info` function, which takes in a `kernel_siginfo` struct whose field `si_signo` is set to the required signal number.

The timer is registered on startup using `add_timer` and deregistered on shutdown using `del_timer_sync`, which, as previously described, waits for timer callbacks (if any) to terminate.

## 3.3 Synchronization

All access to the pid list are thread-safe and locked via a mutex: this is to allow multiple users to safely add signal requests without leading to race conditions. According to [this source](#), module calls may be called in parallel on a multi-user, multi-cpu system, and hence this safety is necessary. The mutex is implemented using the methods in `include/linux/mutex.h`

```
1 DEFINE_MUTEX(mutex);
2
3 mutex_init(&mutex);
4
5 mutex_lock(&mutex);
6 // critical code accessing pid linked list
7 mutex_unlock(&mutex);
8
9 mutex_destroy(&mutex);
```

### DoS attack on file

By sending in a large number of pids, one expects that the system goes into starvation while adding the pids to the file and not giving up the lock, thereby preventing the timer callback from sending signals. This is prevented to a degree by limiting the number of such requests. However, this check is done in a synchronous fashion, hence it is possible that the system does get starved. I would need a lock-free algorithm that can check the length of the list as well as add (not remove) to the list to prevent this from happening.

## 4 Submission Details

The diff file was generated after cleaning the build directory using `make mrproper`, and running `git diff --no-index <original> <new> --output ctxtswitch.patch`. This patch is compatible with the kernel and can be applied, with the benefit of being much smaller than the patch generated by the command mentioned. Git diff respects the `.gitignore` files in the directories thereby making a much smaller patch file.

The directory structure is as follows:

```
assignment1_hard_2020CS10869
├── ctxttrack.patch
├── report.pdf
├── sig_mod
│   ├── Makefile
│   └── sig_mod.c
```

The kernel module can be made using the `make` command, and loaded/removed using `sudo insmod` and `sudo rmmod`.