Homework 2

W4118 Spring 2021

UPDATED: Sunday 1/31/2021 at 10:37pm EST
DUE: Monday 2/8/2021 at 11:59pm EST

All non-programming, written problems in this assignment are to be done by yourself. Group collaboration is permitted only on the kernel programming problems. All homework submissions (both individual and group) are to be made via **Git**.

You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this **file** in the top-level directory of your homework submission. Homeworks submitted without this file will not be graded. **Be aware that commits pushed after the deadline will not be considered.** Refer to the homework policy section on the **class web site** for further details.

Individual Written Problems:

The GitHub repository you will use can be initialized **here**. To clone this repository, use the following command: `git clone git@github.com:W4118/s21-hmwk2-written-UserName.git` (Replace UserName with your own GitHub username). This repository will be accessible only by you, and you will use the same SSH public/private key-pair used for homework 1.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points.

1. Exercise 3.11
2. Exercise 3.12
3. Exercise 4.6
4. Exercise 4.7
5. Exercise 4.8
6. Exercise 4.14

Group Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository for your group has been

setup already on Github. You don't need the Github Classroom link for the group assignment. It can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk2-teamN.git
```

(Replace `teamN` with the name of your team, e.g. `team0`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as **git-pull**, **git-merge**, **git-fetch**.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with the checkpatch.pl script on the default path in the provided VM. Errors from the script in your submission will cause a deduction of points.

The kernel programming for this assignment will be run using your Linux VM. As a part of this assignment, you will be experimenting with Linux platforms and gaining familiarity with the development environment. Linux platforms can run on many different architectures, but the specific platform we will be targeting is the X86_64 CPU family. All of your kernel builds will be done in the same Linux VM from homework 1. You will be developing with the Linux 5.4 kernel.

For this assignment, you will write a system call to dump the process tree and a user space program to use the system call (70 points).

---

1. **Build your own Linux 5.4 kernel and install and run it in your Linux VM.**

   Kernel building instructions for VM
   Build and run a custom kernel for your VM. The source code of of the VM is located at `master` branch of your team repo. You can checkout to that branch.

   If you have enough disk space, you can clone the `master` branch in a separate directory so that you can keep the compiled files for both branch. This will save your time on

compiling.

1. In your repo, run `git checkout master` to switch to the "master" branch.
2. Make the config file for the VM kernel.

```
make w4118_defconfig
```

3. Run `make -jN`, where N is the number of cores on your VM. Wait for the kernel to compile.
4. Install the new kernel by running:

```
sudo make modules_install && sudo make install
```

5. Reboot your vm.
6. In the boot selection screen(called `grub`), which shows up immediately after the VMWare logo screen, select "Advanced options for Ubuntu GNU/Linux" and choose the kernel identified by "w4118".
7. You are now running your custom kernel! You can use `uname -a` to check the version of the kernel

2. **Write a new system call in Linux**

General Description

The system call you write should take three arguments and copy the process tree information to a buffer in a breadth-first-search (BFS) order. You should only include processes in your buffer and count only the number of processes, not threads.

The prototype for your system call will be:

```
int ptree(struct prinfo *buf, int *nr, int root_pid);
```

You should define *struct prinfo* as:

```
struct prinfo {
        pid_t parent_pid;       /* process id of parent */
        pid_t pid;              /* process id */
        long state;             /* current state of process */
        uid_t uid;              /* user id of process owner */
        char comm[16];          /* name of program executed */
        int level;              /* level of this process in the sub
```

```
};
```

in `include/linux/prinfo.h` as part of your solution.

Use the following function to get the `task_struct` given pid (Don't worry about namespaces and virtual pids):

```
static struct task_struct *get_root(int root_pid)
{
        if (root_pid == 0)
                return &init_task;

        return find_task_by_vpid(root_pid);
}
```

Parameters description

- **buf** It should point to a buffer to store the process tree's data. The data stored inside the buffer should be in BFS order: processes at a higher level (level 0 is considered to be higher than level 10) should appear before processes at a lower level.
- **nr** represents the size of this buffer (number of entries). The system call copies at most that many entries of the process tree data to the buffer and stores the number of entries actually copied in `nr`.
- **root_pid** represents the pid of the root of the subtree you are required to traverse. Note that information of nodes outside of this subtree shouldn't be put into `buf`.
- **Return value:** Your system call should return 0 on success.

Additional Requirements

- You should fill the buffer when traversing the tree in BFS order, till it's full. E.g. if we have a tree as below:

```
        0
      /   \
     1     2
    / \   / \
   3   4 5  6
```

  and you are given a buffer of size `4 * sizeof(struct prinfo)`, the buffer should be

```
[
prinfo {pid-0, parent-pid-0, level-0},
prinfo {pid-1, parent-pid-0, level-1},
prinfo {pid-2, parent-pid-0, level-1},
prinfo {pid-3, parent-pid-1, level-2};
]
```

- You should not traverse more of the tree than needed. i.e. only traverse the first and at most `nr` processes.
- There should be no duplicate information of processes inside the buffer.
- If a value to be set in `prinfo` is accessible through a pointer which is null, set the value in `prinfo` to 0.
- Your system call should be assigned the number **436** and be implemented in a file `ptree.c` in the kernel directory, i.e. `kernel/ptree.c`.
- Your algorithm shouldn't use recursion since the size of the function stack in the kernel is very small.
- Your code should handle errors that could occur. At a minimum, your system call should detect and return following error number:
  - `-EINVAL`: if `buf` or `nr` are null, or if the number of entries is less than 1
  - `-EFAULT`: if `buf` or `nr` are outside the accessible address space.

Hints

- Linux maintains a list of all processes in a doubly linked list. Each entry in this list is a `task_struct` defined in **include/linux/sched.h**.
- When traversing the process tree data structures, it is necessary to prevent the data structures from changing in order to ensure consistency. For this purpose the kernel relies on a special lock, the `tasklist_lock`. You should grab this lock before you begin the traversal, and only release the lock when the traversal is completed. While holding the lock, your code must not perform any operations that may result in a sleep, such as memory allocation, copying of data into and out from the kernel etc. Use the following code to grab and then release the lock:

```
read_lock(&tasklist_lock);
do_some_work();
read_unlock(&tasklist_lock);
```

- In order to learn about system calls, you may find it helpful to search the Linux kernel for other system calls and see how they are defined. Take a look at **include/linux/syscalls.h**.

3. **Test your new system call**

   General Description

   Write a simple C program which calls `ptree` with the root_pid as an argument. If no argument is provided, your program should return the entire process tree. The program should be in the `test` branch of your team repo, and your makefile should generate an executable named **test**. Since you do not know the tree size in advance, you should start with some reasonable buffer size for calling `ptree`, then if the buffer size is not sufficient for storing the tree, repeatedly double the buffer size and call `ptree` until you have captured the full process tree requested. Print the contents of the buffer from index 0 to the end. For each process, you must use the following format for program output:

```
printf("%s,%d,%d,%ld,%d,%d\n", buf[i].comm, buf[i].pid,
            buf[i].parent_pid, buf[i].state, buf[i].uid, buf[i].lev
```

   Example program output:

```
swapper/0,0,0,0,0,0
systemd,1,0,1,0,1
kthreadd,2,0,1,0,1
systemd-journal,2924,1,1,0,2
....
kworker/u128:1,1034,2,1026,0,2
kworker/3:2,1419,2,1026,0,2
...
zsh,1654,1653,1,1000,5
...
```

   Hints

- The `ps` command in the VM will help in verifying the accuracy of information printed by your program.
- Although system calls are generally accessed through a library (libc), your test program should access your system call directly. This is accomplished by utilizing the generall purpose `syscall(2)` system call (consult its man page for more details).

Compiling for the VM:
You can compile your test program with a standard toolchain to make it run on the VM so you don't need to specify any arguments, just `make` it.

4. **Investigate the Linux process tree**

   `gdb` is a debugger on the Linux platform. It has a command line interface. Use `gdb` to debug an arbitrary program on the VM, choose a breakpoint, run and halt gdb there. Then, use the program you developed in part 3 to find the program that gdb is attached to and find out the status code of it. Try to explain the status code. **Hint:** The program that the gdb is attached to is not runnable. Learn to use GDB in 3 seconds:

```
$: gdb ./test
GNU gdb (Ubuntu 8.1-0ubuntu3) ...
...
Reading symbols from ./test...done.
(gdb) break 49
Breakpoint 1 at 0x4007e7: file test.c, line 49
(gdb) run
Starting program: ..../test
Breakpoint 1, .... at test.c:50
(gdb)
```