HW2 (W4118 Fall 2024)

w4118.github.io/homework/hw2.html

W4118

> **DUE: Wednesday 10/2/2024 at 11:59pm ET**

> Last modified: Fri Sep 27, 0005 hrs (part 4, question 2)

General instructions

All homework submissions are to be made via Git. You must submit a detailed list of references as part of your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional sta d include this file in the top-level c branch of your team repo. **Be awa not be considered.** Refer ite for further details.

Group programming s. We will let you know when the GitHub. It can be cloned using the m number, e.g. `team0`. You can find

```
$ git clone git@gi
```

> **IMPORTANT**: rminal of your **VM**, instead of it into your VM. The filesys machine might use a case-ins setting on macs). Cloning the repository to a case-insensitive filesystem might end up clobbering some kernel source code files. See this post for some examples.

This repository will be accessible to all members of your team, and all team members are expected to make local commits and push changes or contributions to GitHub equally. You should become familiar with team-based shared repository Git commands such as git-pull, git-merge, git-fetch. For more information, see this guide.

There should be at least five commits **per member** in the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the Linux kernel coding style. You **must** check your commits with the `run_checkpatch.sh` script provided as part of your team repository. Errors from the script in your submission will cause a deduction of points. (Note that the script only checks the changes up to your latest commit. Changes in the working tree or staging area will not be checked.)

The kernel programming for this assignment will be run using your Linux VM. As 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 platforms we will be targeting are the X86_64 or Arm64 CPU families. All of your kernel builds will be done in the same Linux VM from homework 1. You will be developing with the Linux 6.8 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.

For students on Arm computers (e.g. macs with M1/M2/M3 CPU): if you want your submission to be built/tested for Arm, you must create and submit a file called `.armpls` in the top-level directory of your repo; feel free to use the following one-liner:

```
$ cd "$(git rev-pa                                      add .armpls &&
git commit -m "Arm
```

You should do this f   mit for grading.

For all programming   as well as a single README file   lease do NOT submit kernel image   n your solution differs from what wa   are welcome to include a test run in   rks. **It should also state explicitl   submission and how much tim**   README should be placed in the top   epo (on the same level as the `l`

Part 1: Build y   and run it in your Linux VM

You will need to install your own custom kernel in your VM to do this assignment. The source code for the kernel you will use is located in your team repository on GitHub. Follow the instructions provided here to build and install the kernel in your VM.

Part 2: Write a new system call in Linux

General description

The system call you write will retrieve information from each thread associated with each process in some subset of the process tree. That thread information will be stored in a buffer and copied to user space.

Within the buffer, threads associated with the same process (main thread) should be grouped together. These thread groups should be sorted in breadth-first-search (BFS) order of the associated process within the process tree. Within each thread group, the threads should be sorted in ascending order of PID. You may ignore PID rollover for the purposes of ordering threads. See the hint on `PID vs TGID` below for more information on what we mean when we say the PID of a thread. See also `Additional requirements` below for an example ordering.

The prototype for your system call will be:

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

You should define `struct tskinfo` as:

```
struct tskinfo {
    pid_t pid;              /* process id */
    pid_t tgid;             /* thread group id */
    pid_t parent_pid;       /* process id of parent */
    int level;              /* level of this process in the subtree */
    char comm[16];                                                  node */
    unsigned long                                                   le() */
    unsigned long
};
```

You should put this [illegible] part of your solution. Note that t [illegible] nel source tree. The `uapi/` directory [illegible] should include structures used as a [illegible] ay find it helpful to look at ho [illegible] This is another structure that is par [illegible] cess times.

To ensure that user [illegible] t of header files, you must do the foll [illegible] ee:

```
$ sudo make header
```

This copies the UAPI header files to `/usr/`. You should now be able to see your new header file as `/usr/include/linux/tskinfo.h`. You should do this every time you update a header file that is part of the user space API of the kernel.

Description of parameters

- **buf** points to a buffer to store the thread data from the process tree. The thread data stored inside the buffer should be sorted first according to the associated main process in BFS order, where processes at a higher level (level 0 is considered to be higher than level 10) should appear before processes at a lower level. Then, when listing threads within a process's thread group, the threads should be stored in ascending order of PID.

- **nr** points to an integer that represents the size of this buffer (number of entries). The system call copies at most `*nr` entries of thread data to the buffer, and stores the number of entries actually copied in `*nr`. Note that the actual size of the buffer will be `*nr * sizeof(struct tskinfo)` bytes.
- **root_pid** represents the PID of the thread that will serve as the root of the subtree you are required to traverse. See the hint below on `PID vs TGID` for more information on what we mean when we say the PID of a thread. Information of nodes outside this subtree shouldn't be put into `buf`. **If the thread with PID `root_pid` is part of a thread group with multiple threads, they (and all of their children) should be included in `buf`, unless the maximum number of entries is reached.** Note that this means you might end up including threads whose PID is smaller than `root_pid`, as long as these threads are in the same thread group as the thread with PID `root_pid`.
- **Return value:** The function defining your system call should return 0 on success, and return an appropriate error on failure such that `errno` contains the correct error code.

Additional require

- You should fill the buffer when traversing the tree primarily in **BFS order** of process (main thread), then for each thread within a process in **ascending order of PID** until either every thread is stored, or the number of threads reaches `nr`. E.g. If we have a tree as below:

  ```
            0
          /   \
        1,2   3,4,5
      /   \   / \
     6     7 8   9
  ```

  and you are given a buffer where `*nr` is `10`, the buffer should be filled as follows (excluding program counters):

  ```
  [
  tskinfo {pid-0, tgid-0, ppid-0, level-0, … },
  tskinfo {pid-1, tgid-1, ppid-0, level-1, … },
  tskinfo {pid-2, tgid-1, ppid-0, level-1, … },
  tskinfo {pid-3, tgid-3, ppid-0, level-1, … },
  tskinfo {pi
  tskinfo {pi
  tskinfo {pi
  tskinfo {pi
  tskinfo {pi
  tskinfo {pi
  ]
  ```

  Note that in th , and 9) and 3 threads (2, 4, re children of 4. In this scenari out (see the hint below on `PID` vels are zero-indexed, and k with PID `root_pid` (this

- There should e buffer.

- If a value to be hich is null, set the value in `tskinfo` to 0.

- Your system call should be assigned the number **462** and be implemented in a file `ptree.c` in the `kernel/` directory, i.e. `kernel/ptree.c`. Note again that this path is relative to the root directory of your kernel source tree. You will need to modify the appropriate kernel `Makefile` so that it is aware of your new source code file, or it will not know to compile your source code file with the rest of the kernel code.

- Retrieving some of the information for your `struct tskinfo` will be architecture-specific (i.e. it will be implemented differently depending on whether your platform is x86-64 or ARM64). You should call generic functions from your main `kernel/ptree.c` file for retrieving these values, but implement them in `arch/[x86 or arm64]/kernel/ptree.c` (and make sure you modify the `Makefile` in the same folder appropriately). This ensures that only the correct retrieval function is included when your kernel is compiled. Although you are only required to complete solutions for one architecture, defining these functions in both `arch/x86` and `arch/arm64` will allow you to develop alongside students with a different platform. Note that you can put declarations for these functions in `include/linux/ptree.h`.

- Your algorithm shouldn't use recursion since the size of the function stack in the kernel is quite small, typically only **8KB**.

- Your code should handle errors that could occur. For example, some error numbers your system call should detect and return include:

    - `-EINVAL` less than 1.
    - `-EFAULT` pace.

Hints

- This syscall in e recommend that you appro elopment and testing to chec ay:

    - To unde ant information, you shou hich returns the informat
    - Now exp ss in the given subtree ill help you understa without worrying about the nuances of listing threads versus processes.
    - Finally, write the system call that works for both processes and threads. Think carefully about how threads intersect with the process tree, and how you must modify your BFS algorithm to include them. You may find this blog post helpful for relating processes and threads.