Homework 6

W4118 Spring 2021

UPDATED: Tuesday 4/13/2021 at 4:00pm EST
DUE: Saturday 4/17/2021 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

Please follow the Github Classroom link: **Homework 6**. This will result in a GitHub repository you will use that can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk6-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.

Please note that the link and the created repo is for the individual written part only. The workflow of this written part is the same as Homework 1. For group assignment, the workflow is different. Refer to the description below.

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points. For problems referring to the Linux kernel, please use the same kernel version as you use for the group kernel programming assignment. Please include all your answers in a file named written.txt that is uploaded to your GitHub repository for this assignment. Graphs or charts can be uploaded as a separate PDF file, for which a URL has been included in your written.txt that can be used on to open the file.

1. Exercise 9.21
2. Exercise 10.13
3. Exercise 11.15
4. Exercise 11.19
5. Exercise 12.10
6. Exercise 12.15

Group Programming Problems

Group programming problems are to be done in your assigned **groups**. The Git repository your entire group will use to submit the group programming problems can be cloned using:

```
git clone git@github.com:W4118/s21-hmwk6-teamN.git
```

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` and `git-fetch`. You can see the documentation for these by typing `man git-pull`, etc. You may need to install the `git-doc` package first (e.g. `sudo apt install git-doc`).

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 latest version of **checkpatch**. Errors or warnings from the script in your submission will cause a deduction of 5 points.

The kernel programming for this assignment will be done using your x86 VMs. You can continue using the same VM that you used for Homework 2.

Pseudo File Systems

On Linux/Unix systems, a **pseudo file system** is a virtual file system that is used to access information from the kernel. It is so termed because it does not represent a real, physical storage facility. Instead, it resides entirely in main memory and provides a place to store files and directory entries that represent kernel or other (static or dynamic) information. Because pseudo file systems are not real file systems, they consume no storage space and only a limited amount of memory.

The most well-known example of a pseudo file system is *procfs*, short for *process file system*. It was originally intended to provide information about processes, but nowadays is used as a wider interface to many kernel data structures. It is traditionally mounted at `/proc`. Most of it is read-only, but some files allow kernel variables and behavior to be modified. See the *proc(1)* manpage for further details. Other pseudo file systems in Linux, for instance, are *sysfs*, *ramfs*, *devpts*, *debugfs*, *tmpfs* and *sockfs*, to name a few.

PpageFS: Process Pages File System

For this assignment, you are to implement a pseudo file system called **PpageFS**, which will export kernel information about the process physical page usage information of currently running processes. This assignment will demonstrate Linux's file systems behavior and teach you how to construct a file system (without the need for the complete logic to handle real storage). You should be able to mount your filesystem anywhere using the mount command, which specifies where to mount your filesystem. For example, you can mount your filesystem at /mnt using:

```
mount -t ppagefs none /mnt
```

When *PpageFS* is mounted, it provides the following structure. The contents of the root of the file system should be a set directories named *PID.ProcessName*, the PID and ProcessName should be the pid and name of the currently running process in the system. Within that directory, there should be should be two files named *total* and *zero*, which store the amount the total and zero physical pages mapped into the address space of the process in a string, respectively. Physical pages here refer to actual pages of physical memory resident in RAM. All files should be readable and executable for everyone.

Specifically,

- *total* should count all distinct physical pages of RAM mapped into the address space of the process.
- *zero* should count all distinct physical pages of RAM for which the contents are all zeros, including (1) a page that is mapped to a zero page or (2) a page that is not mapped to a zero page but only contains zeros.
- Distinct physical page means: If a page is mapped more than once for a process, it is only counted as one page for the process; if a page is mapped by multiple processes, it is counted as one page for each process.
- You should only count regular pages; do not count huge pages.

For example, when you `cat total` and the process has 100 virtual pages, 5 of which are mapped into the same physical pages, you should get `96` on your shell. You can get the physical page mapping information by walking the page table of a process.

You should implement *PpageFS* at `fs/ppagefs` and properly create or modify the `Makefile` to build your code.

You may find existing code of pseudo file systems extremely useful as it provides countless examples. Specifically, you are encouraged to study the code in `fs/ramfs` (a simple ram file system), `fs/debugfs` (a simple file system for debugging purposes) and `fs/devpts` (which creates files in response to adding/removing certain kernel objects). Both of them make use of the API provided by the generic file system library in `fs/libfs.c`. *These examples contain nearly everything that you need to complete this assignment.*

1. Implement the *PpageFS* file system. Your implementation should take care of setting the ownership and the permissions for all files and directories and implementing the relevant file system operations as well as appropriate acquiring locks and/or mutexes. The name of a process may contain invalid characters for a filename, in that case, you should properly escape the name so that it won't break the filesystem.

2. To test your file system, write a simple program called *map_pages* that creates a process which maps several zero pages, transform some zero pages to non-zero page and scrub those pages to zero pages. Then write another program called *inspect_pages* that takes a PID as the argument and check the total and zero pages mapped by PID before and after each operation. Put your test programs in the test directory of your GitHub repo.