Virginia Tech ❖ Bradley Department of Electrical and Computer Engineering

ECE 5984 Linux Kernel Programming
Spring 2017

Large Project: Process Hierarchy Pseudo-Filesystem

1 Introduction

Contrary to a regular filesystem, a *pseudo*-filesystem does not manipulate an actual non-volatile storage device. It rather resides in main memory, its content being created at boot time, or at runtime on demand. Examples of pseudo-filesystems are *procfs* (mounted in `/proc`), *sysfs*, etc. They are generally used to provide the user with information about the kernel: for example, one can see statistics about main memory usage in `/proc/meminfo`. They can also be used to control the kernel behavior: for example, various system caches can be flushed by writing into `/proc/sys/vm/drop_caches`. Thus, pseudo-filesystems can be seen as an interface between user space and kernel space.

The objective of this project is to implement in Linux a pseudo-filesystem, giving to the user information about the hierarchy of processes currently running on the system, and allowing to send a signal to these processes.

The following concepts from the course will be involved in that project:

- File systems;
- Process management.

2 Project description

Once mounted, each folder in this filesystem is named after the PID of the corresponding process.The directory tree replicates the process hierarchy. A folder related to a process P contains the following:

- **Several files**:
  - One file *<tid>.status* for each of the threads belonging to P. These files hold various information about the thread with thread identifier *tid*:
    - Its current state (`TASK_RUNNING`, `TASK_STOPPED`, etc.);
    - The fact that the thread is a user or a kernel thread;
    - The current CPU executing the thread;
    - The start time of the thread;
    - The name of the running executable;
    - Various information about the memory usage (ex: stack pointer of each thread, memory usage of the related process, etc.);
    - Various other information of your choice.
  - *signal*, which when written sends a specific signal to the process. In order to send a signal through that file, an integer representing the number of the signal to send should be written, for example:

    ```
    $ echo "9" > /path/to/the/file
    ```

    See `man 7 signal` for a description of Linux signals and their associated numbers.
- **Potentially one or several sub-folders**, each one of them representing a child of the process: the directory tree is representative of the process hierarchy. Each sub-folder follows the same model as the parent (status and signal files plus potential sub-folders).

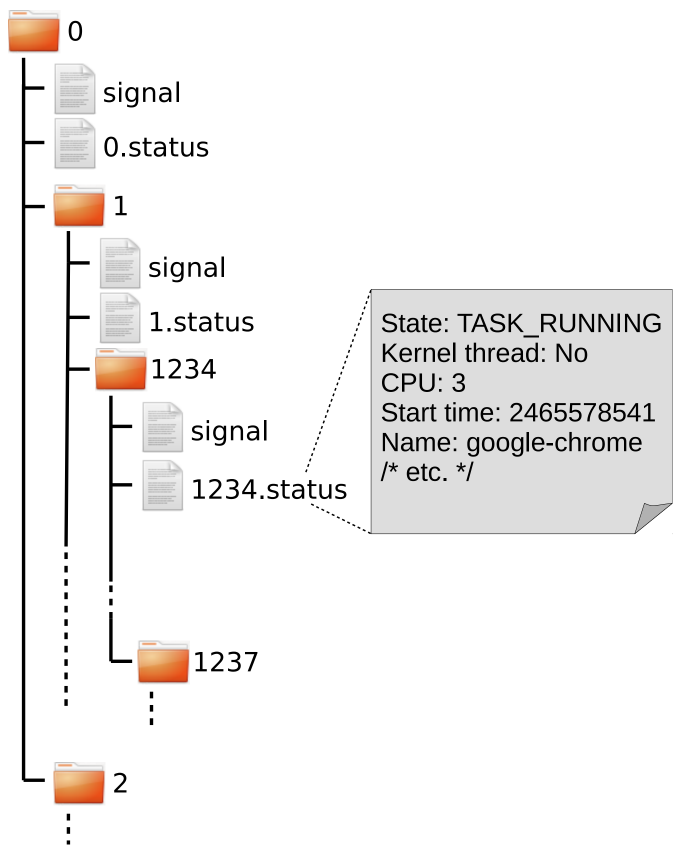

Figure 1: Illustration of the pseudo-filesystem file & directory tree

The pseudo-filesystem hierarchy is illustrated on Figure 1.

Guidelines on creating a very basic pseudo-filesystem can be found here: `https://lwn.net/Articles/57369/`. However, note that this guide is relatively old (2002). Other sources of inspiration might be found in the kernel code itself, for example the implementation of the very simple `ramfs` filesystem, or the library `libfs` (still in the kernel sources) providing a substantial amount of helper function for writing simple filesystems.

It is also requested to develop a small C application that will be used as a test case. The application should spawn multiple children (for example using `fork`), and threads. The created hierarchy should be reflected in the implemented filesystem. A signal handler for a user defined signal (for example `SIGUSR1`) should be installed in one of the children in order to test the related signal file.

3 Requirements and VM development environment

The implementation should be done in a kernel module, compiling against Linux 4.0. This is not a group project and it should be performed by each student on its own.

3.1 Virtual machines

A virtual machine is available to optionally serve as a development environment. It is an enhanced and up-to-date version of the VM used in project 3 (scheduling). It runs a regular Ubuntu guest operating system, and contains the *Qemu* emulator that is invaluable to debug (and explore) kernel code at runtime in conjunction with GDB. Several versions of the VM are available:

- **VirtualBox + Qemu**: compatible with Linux/Windows/Mac hosts but slow as Qemu is using emulation. Link: `http://bit.ly/2m4uzFg;`

- **VMware player + KVM**: compatible only with Linux and Windows, but the KVM environment is significantly faster than a Qemu one. Link: `http://bit.ly/2mPboQ5` ;
- An archive containing only a **KVM environment**: compatible with native Linux hosts, best performance without the VirtualBox/VMware virtualization layer. Link: `http://bit.ly/2mnLlSk`.

The id/password combos are for all VMs (VirtualBox/VMware/Qemu/KVM): `user/a`.

Note that concerning the VMware VM, after importing it into VMware player, you need to manually activate the forwarding of hardware virtualization extensions for KVM to work correctly. This is done by going to the VM option window → Hardware → Processors → Virtualize Intel VT-x/EPT or AMD-V/RVI.

4 Results to be handed - Deadline: 2017-03-24 11:59 PM

The following is expected to be handed by 2017-03-24, 11:59 PM:

1. The module sources and the associated Makefile. The module should compile against Linux 4.0 ;
2. The source of the C application(s) used for testing, and if needed a README file explaining how to launch the testing procedure.

All of this should be contained in a *tarball*, with the following format: `<VT PID>.project4.tar.gz`. For example: `johndoe.project4.tar.gz`