CIS–4020 Lab
Fork Watcher

Last Revised: September 22, 2024

1 Introduction

In this lab you will modify the kernel so that each time a new process is created, information about that operation is gathered and stored in a circular buffer inside the kernel. As with the previous lab you will write a module that provides a `/proc` file interface to this information. You will also write an ordinary user mode application that formats the raw data in the `/proc` file in a more user friendly way.

To prepare for this lab you should look over the following materials.

- The `clone` manual page in section 2 of the Linux manual.
- The kernel source for `kernel_clone` and `copy_process`.
- The manual page for the `getpwuid` function.
- The error code definitions in
  - `include/asm-generic/errno-base.h`
  - `include/asm-generic/errno.h`

2 Gathering Data

The first step is to instrument the kernel so that it collects "interesting" data about each fork operation that occurs. Since all of the process creation system calls ultimately call `kernel_clone` it is sufficient to add your instrumentation there (or perhaps in functions called by `kernel_clone` such as `copy_process`). The information you should gather includes

- The clone flags passed to `kernel_clone`.
- The user ID (UID) associated of the calling process.
- The process ID (PID) associated with the calling process.
- The PID of the new process.
- The name of the command associated with the calling process.
- The return value of the clone operation.

Because Linux supports multiple namespaces and since `kernel_clone` can, under certain conditions, create a new process in a new namespace you might want to also output information about the namespace of the calling process and the namespace of the new process. However, this is not required in a first version of this lab (you can assume that only a single namespace is being used).

Define a structure (say, `struct fork_info`) that can hold the necessary information. This definition needs to be shared between your modified kernel and the module that implements the `/proc` file so you will want to put it in a header file (say, `fork_info.h`). Store this new header file in the same location as the other kernel headers under `include/linux`.

2.1 Circular Buffer

You will also need to define a circular buffer to hold the fork information records. This buffer is an array (you decide how large) with two pointers or index variables that define the next available slot and the next slot from which a record should be extracted. As the buffer fills, these pointers should wrap around and reuse the space. If the `next_in` pointer catches up with the `next_out` pointer, old data should be overwritten. In this way the buffer will only hold information about that last n forks, where n is the size of the buffer.

Note that because multiple threads can call `kernel_clone` at the same time, it is possible that multiple threads will be accessing the buffer simultaneously (also the module might be accessing the buffer as well). To ensure normal functioning in this case, you should provide appropriate locking as described in class.

Notice that the circular buffer is an example of the producer/consumer problem where the kernel code "produces" fork records and the module "consumes" them. However, unlike the classic producer/consumer problem, we don't want to block the kernel if the buffer fills. In that case the old records should be overwritten. We also don't want to block the module if the buffer is empty so that any attempt to read the fork records will not hang forever waiting for more. These changes simplify the locking strategy required.

Be aware that the kernel is designed to work with very limited stack space. Once inside the kernel you are using the kernel stack (not the stack of the application) and you must be careful not to overflow that stack. Avoid declaring "large" variables as local to any function. We may not have any problems with this, but again you should consider it.

3 Module

As with Lab #2 you will need to write a module that outputs the fork information records through a `/proc` file. Use the same `seq_file` API as you did in the previous lab. Since the application that reads the `/proc` file might take a long time to do it, there is a possibility that the fork information buffer could get new records added to it during the `seq_file` iteration. Again we will try to ignore this issue in this lab, but you should be aware of it in case there are problems related to it. Fully correct handling of this is potentially difficult and some compromises may be necessary.

4 Formatting Program

You are allowed to output raw numbers in a user unfriendly format to the `/proc` file. However, for this lab you are also asked to write a user mode application that read the `/proc` file in the usual way and reformats those numbers nicely. Your program should do (at least) the following

- Break out the clone flags into symbolic names.
- Convert UID values into actual user names (use the C library function `getpwuid`).
- Convert return values into proper symbolic names. You can consult the `clone` manual page for a list of possible error returns and cross reference that list with the numbers in the appropriate `errno.h` files (see the list in the Introduction section).

5 Report

Write a report for this lab following the lab report template provided by your instructor. Include a listing of your module code, your user mode application, and a summary of any other source files you added or changed in an appendix. Be sure to include a description of your code that explains what it does and how it works.