CS-550 Section 01 Spring 2020 Assignment 03

Tracking Page Faults

Goals

- Learn about the kernel trace capability - an important tool that enables both learning about the kernel, and an important debug mechanism for kernel software.
- Learn about page faults - how they occur, and how they are typically distributed
- Learn about benchmarking tools to exercise the kernel, while tracing the results of those benchmarks.

[Description] [Grading Guidelines]

Description

This assignment is based on the kernel Kprobe mechanism. Kprobes enable you to dynamically set a breakpoint in any kernel routine and collect debugging and performance information non-disruptively. You can probe almost any kernel code exposed symbol, specifying a handler routine to be invoked when the code at that symbol is executed.

1. Learn how to use kprobes in the Linux kernel. The web site https://www.kernel.org/doc/Documentation/kprobes.txt provides basic documentation of kprobes. Read through this documentation to get a basic understanding of kprobes.

2. Download, build, and run the kprobe examples in https://elixir.bootlin.com/linux/latest/source/samples/kprobes. You can use these as a basis for your coding.

3. Get familiar with the handle_mm_fault() function in the Linux kernel. Get a feel for when this function is invoked, and what it does.

4. Create a kernel module called **pf_probe_A** that takes the process-ID of an active process as an argument as a module parameter, and prints the virtual addresses that cause page faults to the system log using printk(). Your code should work for any arbitrary target process.

   #### Hints:

   - In the architecture that the CSVB machines use, X86-64, when a function is called the first parameters is in the di family of registers (%rdi for 64 bit, %edi for 32 bit, %di for 16 bit, etc.), the second parameter is in the si family of registers, the third in dx, and the fourth in cx. You may assume these conventions hold when your kernel module is executed.
   - In order to find the process ID which caused the page fault, you need to start at the vm area structure, find the memory manager structure that describes the virtual memory, find the canonical owner task structure in the memory manager structure, and find the pid in the task structure.

5. Create a second kernel module called **pf_probe_B** that stores the time and address of each page fault related to a specific process instead of printing to the console. Then, when the

module is unloaded, print to the console a scatter plot of all the page fault information collected that has time on the X axis, and virtual page number on the Y axis. Scale the plot so that it is 30 lines high and 70 columns wide. Include appropriate labels and titles.

Hints:

  - You may assume each page is 4K large. That means offset in the page takes up the rightmost 12 bits of the address.
  - Code in the Linux kernel will not do 64 bit division. Use google to figure out how to divide 64 bit fields (like addresses) in the kernel in order to scale your plot.
  - Put a check in your code to make sure you calculate the row/column of a page fault correctly, and that it is within the bounds of your graph. If not, print a debug message so you can fix it instead of causing a segmentation violation in the kernel, and having to restart your CSVB operating system.
  - I wrote a shell script to trace a specific command. The code is simple, but to save you time:

```
#! /bin/bash
# Run command from parameters in background
$* &
# Save pid of last background command
pid=$!
# Install probe, running on the pid specified
sudo insmod pf_probe_B.ko tpid=$pid
# wait for the background job to complete
wait $pid
# Remove the module
sudo rmmod pf_probe_B
# Look at the last 50 lines of the console print
dmesg | tail -50
```

6. Test your scatter plot on at least three different types of target applications, such as kernel compilation (compute and I/O intensive), sysbench (compute intensive), or iperf (network I/O intensive). Look for interesting trends in memory access patterns. Describe your findings in a short report (two or three paragraphs), and be prepared to defend your conclusions in the interview.

7. **Extra Credit Section** (optional). Create a third kernel module called **pf_probe_C** which writes up to three separate scatter plots - one for page faults in the user's code segment, one for page faults in the user's data segment, and one for page faults that are not in either the code segment or the data segment. (Most of these will be in the stack.) If no page faults occur in one of the segments, leave the plot out. Rerun the tests in the previous section and see if the results are clearer, or if you can form any new conclusions with this finer grained analysis.

Hints

  - The memory management structure keeps track of the start and end of the code and data segments.

Grading Guidelines

Tar and gzip a directory that contains the following:

- Your kernel Module source code, **pf_probe_A.c**, **pf_probe_B.c** and optionally **pf_probe_C.c**
- **Makefile** - the makefile you used to build the kernel modules
- A **README** file that describes any special features or shortcomings of your code.

- A file called **report.txt** that contains your findings concerning trends in memory access patterns.

We will grade your results as follows:

- 25 points if your kernel module, **pf_probe_A** builds and runs correctly.
- 40 points if your kernel module, **pf_probe_B** builds and runs correctly.
- 20 points if your **report.txt** correctly identifies memory access trends for different kinds of applications.
- 15 points if you code handles all error conditions, is clean, modular, well commented, and produces clean readable output.
- Up to 15 extra points if your submission contains a correct implementation of kernel module **pf_probe_C**. Note - these extra credit points can ONLY be used to offset deductions for this lab. You cannot earn more than 100 points for this lab. You cannot apply extra credit on this lab to any other assignment.
- 10 point late penalty for every 24 hours (or part of 24 hours) your submission is late.

Note that as long as your code runs, you may make your own implementation decisions about anything not explicitly described in the instructions and grading criteria above. Use the README file to document specific decisions you make.