CIS 657 (POS) fall 2013 Lab 8 –

Implement Page Fault Rate Monitor via System call using dynamic kernel linker

# INTRODUCTION

In the last lab, we created a System call using dynamic kernel linker to collect System wide page faults and swap usage information. In this lab we implement Page Fault Rate (PFR) monitor which will collect process wise page faults and swap usage information. PFR monitor will be implemented via System call using dynamic kernel linker and it would also involve modifying kernel data structures and code to record process specific page fault information.

# Paging and Page Faults

* Excerpts from the textbook “The Design and Implementation of FreeBSD Operating System” by Marshall Kirk McKusick.

“When the memory-management hardware detects an invalid virtual address, it generates a trap to the system. This page-fault trap can occur for several reasons. Most BSD programs are created in a format that permits the executable image to be paged into main memory directly from the filesystem. When a program in a demand-paged format is first run, the kernel marks as invalid the pages for the text and initialized-data regions of the executing process. The text and initialized data regions share an object that provides fill-on-demand from the filesystem. As part of mapping in the object, the kernel traverses the list of pages associated with the object and marks them as resident and copy-on-write in the newly created process. For a heavily used executable with most of its pages already resident, this prepaging reduces many of its initial page faults. As missing pages of the text or initialized-data region are first referenced, or write attempts are made on pages in the initialized-data region, page faults occur.

Page faults can also occur when a process first references a page in the uninitialized-data region of a program. Here, the anonymous object managing the region automatically allocates memory to the process and initializes the newly assigned page to zero. The idle process fills part of its time by zeroing out pages on the free list so that the system does not have to do the zero fill when servicing the first reference to a page for an anonymous object. Other types of page faults arise when previously resident pages have been reclaimed by the system in response to a memory shortage.

The handling of page faults is done with the vm\_fault() routine; this routine services all page faults. Each time vm\_fault() is invoked, it is provided the virtual address that caused the fault. The first action of vm\_fault() is to traverse the vm\_map\_entry list of the faulting process to find the entry associated with the fault. The routine then computes the logical page within the underlying object and traverses the list of objects to find or create the needed page. Once the page has been found, vm\_fault() must call the machine-dependent layer to validate the faulted page and return to restart the process.”

# Relevant Files

* sys/vm/vm\_map.h
* sys/sys/proc.h
* sys/vm/vm\_fault.c

# Tasks (90)

The tasks in this lab are divided into two categories:

1. **Requires kernel modification**: To record Process wise page faults and page fault rate (page faults per second) you will have to modify kernel data structures and code. This will require you to rebuild and install kernel for the changes to take effect.
2. **Implemented in System call**: The system call implemented via dynamic kernel linker will be used to retrieve and display page faults and swap usage information. We have provided skeleton code for this. The skeleton code defines a **struct perprocpf**, it has fields for capturing process id, process name, process state(wait/run), process page faults, process page fault rate, total VM for process and current VM usage for process. In system call **function “pfr\_monitor”**, an array of struct **perprocpf** is declared. You are supposed to iterate over all processes in system and populate the array. You also have to sort the array in descending order of page faults rate, the skeleton code takes care of displaying the output. But your user level program has to take care of refreshing the output every second.

Below is the list of requirements for this lab:

|  |  |
| --- | --- |
| Requirements | Points |
| Calculate and Print Page Faults and Page Fault rate (Page faults per second) per process | 30 |
| Print System wide total page faults in last second | 10 |
| Total VM and current VM usage per process | 10 |
| Process id, name and state(run/wait)  P.S. Threads in following state can be considered as running  TDS\_CAN\_RUN, TDS\_RUNQ, TDS\_RUNNING | 10 |
| Output sorted in descending order by Page fault rate | 10 |
| Program that compiles correctly | 10 |
| Program that runs without crash | 10 |

# Submission (10)

Create and attach a README (txt/word/pdf) file at the end of the lab. It doesn't need to be comprehensive, but it should at least cover the following content:

* Which tasks are done, and which are not?
* What’s your basic idea to achieve this task?
* Where is your main function?
* Your user level program and what it does?

If you can only finish some of the tasks in this project, please make sure that your code can at least be compiled and installed and also clearly state in the README file about the missing parts of your project.

**Checklist:** To submit your lab, you need to:

* Attach the \*.tgz file (make sure kernel compiles and runs with this snapshot).
* Attach the **source code of the kernel module**
* **Makefile**
* **User level program**
* Create and attach a **README** file.
* Send this email to the TAs keeping Dr. Chapin < chapin@syr.edu> in the CC with subject line “CIS657: Lab 8”

Saurabh Sabnis < spsabnis@syr.edu >

Prasoon Pandya < pdpandya@syr.edu >