Operating Systems–II: CS3523
Spring 2019
Programming Assignment 6: Linux Virtual Memory
Last date for submission: 20th April 2019, midnight

Setup:

a. Install virtualbox on your laptop.
b. Create a virtual machine with 1GiB RAM, 8GB HDD (dynamic vdi), 1 CPU, 2 network cards (1 NAT, and 1 host only network)
c. Install Centos 7 64-bit on it (minimal centos / no GUI, enable kdump) . Check the kernel version.
d. Yum install crash, gcc, perl, strace rpms.
e. Download kernel-devel-<kversion>, kernel-debuginfo-<kversion>, kernel-debuginfo-common-<kversion> rpms from centos downloads website. Make sure exact match of kernel version with running kernel.
f. You should be able to login via ssh through the host only network to the VM.
g. Run your programs inside the VM.

If you dont use VM setup, your laptop may hang, crash, and go bad. Also TAs will not evaluate your code. Use centos in VM, as Ubuntu is not suitable for kernel programmers.

Problem Statement 1:
Background:

- Linux maintains separate virtual address space for each process's userspace. But there is one kernel address space for all processes.

Goal: The goal of this assignment is to see anatomy of virtual address space of processes using kernel debugger.

Details: On the virtual machine running centos, run the following experiments and create a .txt report with observations.

Experiment 1:

a. As root user, install kernel-debuginfo rpms.
b. Collect kernel dump of the running system using "echo c > /proc/sysrq-trigger"
c. The system reboots and dump is captured in a time-stamped directory under /var/crash/
d. Launch kernel debugger "crash" on the kdump collected using "crash /usr/lib/debug/modules/<version>/vmlinux ./vmcore"
e. Try out few commands like "bt", "ps", "dev"

Experiment 2:

a. As a normal user, run two processes of the "sample" program shared in the class in two different ssh sessions.

b. While the process are still running, collect pmap of both the processes from a third ssh session and save them.
c. While the process are still running, collect kernel dump from the third ssh session.
d. In the kernel debugger, analyze the processes' address spaces using "ps", "vm -m" "vm -p". Notice the page frame number of the libc's text segment. It should be same for both the processes.
e. Capture the outputs from (a), (b), (c), (d) (only necessary lines), in a separate .txt report.

Problem Statement2: (relatively difficult)
Background:

- Linux supports shared memory interprocess communication between processes using Posix shm_open(), mmap() , shm_unlink().

Goal: The goal of this assignment is to develop our own shared memory interprocess communication mechanism in Linux Kernel using pseudo character device driver.

Details: Enhance the "mykmod" Linux character device driver shared in the class, to add read(2), write(2), lseek(2), mmap(2) syscall support on device special file. The driver already supports open(2) , and close(2) system calls. Each device special file represents a pseudo device of 1MB. Also add open, close, fault operations in vm_operations_struct for mmap of the 1MB to support demand paging. Enhance the utility "mykmod_test.cpp" shared in the class to test the read/write via syscall and memory mapping.

You also have to implement an userspace library called **libmyshm.so** which exports/declares two APIs viz.,

```
int myshm_open(const char *name, int oflag, mode_t mode);
int myshm_unlink(const char *name);
```

in libmyshm.hpp.

In Linux, the library source files are denoted as *.hpp & *.cpp. The library binaries are stored in .so format. So your source files will be: libmyshm.hpp & libmyshm.cpp. The library functions have to be re-entrant, thread safe.

Hint: myshm_open() creates a device special file using mknod(2) if it does not exist, and opens it using open(2). myshm_unlink() deletes the file using unlink(2). Create a .txt report with observations.

Experiment 1:

a. Test with single process doing modification using write(2), and fetching using read(2) to a device special file.
b. Test with single process doing modification using mmap'd write, and fetching using mmap'd read.

c. Test with one process doing modification using write(2), and other process fetching using read(2).
d. Test with one process doing modification using mmap'd write, and other process fetching using mmap'd read.
e. Test (a), (b), (c) (d) using multiple device special files. There shouldn't be any interference/data corruptions between different device special files. Writing to one file should not change contents of other files.
f. Capture the outputs from (a), (b), (c) (d) (only necessary lines), in a separated .txt report. Notice the number of iterations it succeeded.

TAs may test your driver, library with a their utility to stress your code. So ensure they are bug free.

Problem Statement 3:
Background:

- Linux kernel implements swapping to swap devices or files. Swapping activity starts when there is memory pressure.

Goal: The goal of this assignment is to observe swapping activity during memory pressure.

Details: Write a C/C++ program "memleak.cpp" that iteratively does "sleep(1), malloc(100MB) and write()" in an infinite loop. There should not be any free() in the program. Upon malloc failure, the program should print number of iterations succeeded, and out of the loop and wait for user's input. Create a .txt report with observations. Your program may be killed by oom killer before printing iteration number. But that is ok. In such case, print iteration number in every iteration

Experiment 1:
a. As a sudo or root, switch off swap using "swapoff <device_or_file>" [to find device/file name use "swapon -s" command]
b. As a normal user, check free memory and swap using "free -mh"
c. As a normal user, run "vmstat -t 1" in one terminal.
d. As a normal user, run the memleak program in another terminal.
e. Capture the outputs from (a), (b), (c) (d) (only necessary lines), in a separate .txt report. Notice the number of iterations it succeeded.

Experiment 2:
a. As a sudo or root, switch on swap using "swapon <device_or_file>". [to find device/file name use "swapon -s" command]
b. As a normal user, check free memory and swap using "free -mh"
c. As a normal user, run "vmstat -t 1" in one terminal.
d. As a normal user, run the memleak program in another terminal.

e. Capture the outputs from (a), (b), (c) (d) (only necessary lines) in a separate .txt report. Notice the number of iterations it succeeded.

Submission Instructions:

1. Place the source code (above 3 files), readme, .txt report in a folder : Assgn6-linuxvmm-<RollNo>
2. And zip the folder.
3. Upload the zip file.

If you don't follow these guidelines, then your assignment will not be evaluated. Upload all these on the classroom by the specified deadline.

Evaluation:

- Problem1
    - Setting up VM, debugger to collecting kernel dumps: 20 marks.
    - Report (outputs and observations) : 10 marks
- Problem2
    - Developing device driver with shared memory facility: 40 marks.
    - Report (outputs and observations) : 10 marks
- Problem3
    - Swapping experiments without and with swap device: 10 marks.
    - Report (outputs and observations) : 10 marks

- If the code does not compile, no marks will be given.
- If there are correctness issues, segmentation faults, kernel panics, marks will be deducted from the component that the bug belongs to.

Any queries in the problem statement to be sent to the TAs before the timeline given in the assignment post.