CS 258, Midterm Exam, Winter 2017

Rules of Engagement. This is a take-home, open-book, open-manual, open-shell, open-Internet exam. Solutions are due by noon of Sunday February 26. **Start early, do not postpone!** Some problems may require more time than you might expect!

You are expected to use (at least) DTrace, the OpenGrok source code browser[1] and the Modular Debugger's running kernel inspection capability (`mdb -k`). You may use any other tools you find useful. The documents in the class directory `http://www.cs.dartmouth.edu/~sergey/cs258/` and the textbook index might be useful, too.

For each problem, you should "show your work": the output of the above tools on your actual platform (virtual or physical).[2] For OS kernel code lines, provide the filename and the line number, or the OpenGrok URL pointing to the right line.

You are allowed to discuss the use of tools with your fellow students, *but not the solutions themselves*. For example, sharing a tracing trick is OK, but sharing part of a solution as such is not. Note that in most exercises you are free to choose your targets (which may help you avoid conflicts with the above rule). If in doubt, ask.

Submit your work as an ASCII text or a PDF file named `<YourName>-midterm-w17.txt` or `<YourName>-midterm-w17.pdf`, or a tarball named `<YourName>-midterm-w17.tar.gz` that contains a directory named `<YourName>-midterm-w17` (with your solutions inside :)). No MS Word files, please, and please do *not* use spaces in any filenames!

Note: The default system for these problems is Illumos, due to the great flexibility of DTrace and MDB. You may choose to do some or all or the problems on GNU/Linux instead of Illumos if you so desire (e.g., using *SystemTap*, *FTrace* with Brendan Gregg's *perf-tools* `http://www.brendangregg.com/linuxperf.html`, or *Kprobes*); note, however, that Illumos' tools for examining a running kernel are much more versatile and stable. Linux will likely be more work, unless you've been working on a Linux project idea and practiced with Linux tools already. I will give extra points for Linux solutions, in recognition of the extra work involved.

General hint: Several of these problems require so-called destructive actions by MDB and/or DTrace. In MDB, use the command `$W` to enable writing memory; use writing formats (see `::formats ! grep write`) to actually write memory. In DTrace, use the `-w` to enable its destructive actions. With DTrace destructive actions enabled, you can suspend a process by using the `stop()` action. See DTrace User Guide for details.

Problem 1. *RSS is Magic.*

1. For each process that uses *libc.so*,[3] calculate how many pages of *libc*'s code are mapped but not loaded into the process. What is the average ratio of loaded *libc.so*'s code size to its overall code size?

Your answer may, of course, vary depending on the system load. Show your logic and commands.

[1] `http://src.illumos.org/source/`

[2] You can record your entire shell session with the `script <filename>` command, or, when working in MDB, with the `::log <filename>` command.

[3] In your Illumos VM, *libc* may show as */usr/lib/libc/libc_hwcap1.so.1* in a *pmap* enumeration of mapped virtual address ranges. This is due to the OS choosing the optimal variant of *libc* for your hardware capabilities, from the variants in */usr/lib/libc/*, and then mounting that specific variant file as */lib/libc.so.1*. So although *ldd* shows a dependency on */lib/libc.so.1*, *pmap* and the kernel see the actual mapped file as */usr/lib/libc/libc_hwcap1.so.1*. More info in `https://blogs.oracle.com/darren/entry/whats_this_lofs_mount_onto`. It may be a nice project to explore this optimization.

2. Select a process, and make it load a page of *libc* code that it has not loaded before. Write a DTrace script to detect this load and show which page has been loaded as a result of your interaction with the process. Alternatively, you may use MDB to confirm the load and the page's identity.

Note: You are free to choose any process you like. I think it will be easier if you choose an interactive one.

3. Choose a process that uses *libc*, and enumerate *continuous* ranges of virtual addresses of *libc*'s loaded pages in this process. How long is the longest contiguous piece? For extra credit, create a map of these contiguous sections vs the full code of *libc*.

Problem 2. *Process, know thyself.*

Write a program that will print out its own memory map without making any calls to *pmap*, *mdb*, *dtrace*, shell, or reading */proc*. The process should rely on its own code and the *libc* standard library to perform the task.

Problem 3. *Won't you be my neighbor?*

Write a DTrace script to report every time a physical page is mapped into a selected process, the PFN of the page, and the reason (such as the intended function for this page: text, global data from file, anonymous memory, etc.) for this mapping.

How frequently are adjacent physical pages mapped into the same process?[4] How frequently do they correspond to consecutive ranges of virtual addresses? Which process has the most consecutive physical pages mapped?

This guide may help[5]: `https://www.princeton.edu/~unix/Solaris/troubleshoot/SolarisMemory.pdf`

Your answer may, of course, vary depending on the system load. Show your logic and commands.

Problem 4. .

Write the function `traceme()` that, when called at the start of a program, would print to *stderr* a message every time a dynamically linked library function if called.

For example,

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

void* traceme();

int main (int argc, char *argv[])
{
     DIR *dp = NULL;
```

[4]There are several ways to interpret this question. For example: what is the chance that a process' address space has two adjacent PFNs mapped into it? Pick the interpretation you find the easiest to answer experimentally and explain your choice.

[5]Note that it dates back to 2008. The new paging algorithm it describes is the unified one we looked at in class.

```
     struct dirent *dptr = NULL;

     traceme();

     // Open the directory stream
     if(NULL == (dp = opendir("/tmp") )){
          printf("\n Cannot open /tmp\n");
          exit(1);
     }

     // Read the directory contents
     while(NULL != (dptr = readdir(dp)) ){
          printf(" [%s] ",dptr->d_name);
     }
     // Close the directory stream
     closedir(dp);
     printf("\n");

     return 0;
}
```

should print to `stderr` something like

```
opendir
readdir
printf
readdir
printf
...
closedir
putchar
```

(some compilers would change `printf("\n")` into a `putchar` or `puts`.

Your code should not execute outside programs but can use any libraries you need. You may put your `traceme()` in the same file or into a separate library.

I will accept solutions that "work on your machine", so long as they work on every run and with a reasonable variation of the body of the main program.

Note: There is a tool for doing this properly from outside of the process: *ltrace*.

Problem 5.

Pick a kernel Kmem object cache and write a program that will cause that Kmem cache to allocate a new slab. You may use any programming language for the program (since, ultimately, its purpose is to consume OS resources via system calls).

Catch the event of a new slab being allocated via a DTrace probe. Compare the size of the object with the number of new allocations that happen before a new slab is needed.