CS 258, Midterm Exam, Winter 2016

Rules of Engagement. This is a take-home, open-book, open-manual, open-shell, open-Internet exam. The solutions are due by noon of Tuesday February 23.

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`), and 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-w16.txt` or `<YourName>-midterm-w16.pdf`, or a tarball named `<YourName>-midterm-w16.tar.gz` that contains a directory named `<YourName>-midterm-w16` (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*, *Kprobes*, or *FTrace*); 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.

Problem 1. *File Caching Is Magic.*

Find the top 5 files that are cached (in Illumos' I/O cache) more than others. Count only the files that *aren't currently open or mmap-ed.*

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

Problem 2. *Mutually Assured Signaling.*

Write a program that waits for a signal, then sends the same signal back to whatever process sent it. (Have it sleep periodically rather than sit in a tight loop).

Run two identical copies of your program; call them Oceania and Eurasia. They will sit and wait for each other forever, of course. You are allowed to send them any signals, but not terminate them or change their code.

Now send a signal from a third program to Oceania—and fake its origin to make it appear that it comes from Eurasia. That should start Oceania and Eurasia ping-ponging each other with signals back and forth. Record these exchanges.

[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.

You are allowed to manipulate any kernel state you need to achieve your goal (show transcript of MDB or DTrace commands).

Problem 3. *No, I want that page! Waaah!*

Write a program that maps a given physical page, identified by its PFN supplied as a command-line argument, into its process.

Calls to MDB from inside the program are not allowed, but you can check prior to running the program or while running it if the physical page is already mapped in some other process. If so, and the mapping cannot be shared, it's OK to exit and start again.

You are allowed to manipulate any kernel state you need to achieve your goal (show transcript of MDB or DTrace commands).

Problem 4. *Faster than a speeding mmap.*

Intercept *mmap* requests from a process and slip in a file you control instead of the file requested. You can write a simple program that opens some file to serve a test subject.

Hint: With DTrace, you can suspend a process using the `stop()` action. This is a "destructive" action; you need to call DTrace with `-w` to enable these.

Problem 5. *It wasn't my fault!*

Write a DTrace script to catch a minor page fault and describe the code path difference between it and a major fault. Where is the check that made this difference?

Hint: See page 473 of the textbook for a discussion of major vs minor faults.

Problem 6. *I dream of write permissions.*

A program *mmap*s a page as read-only into a program, then attempts to write a byte to it. Of course, it will fail—but make it succeed, without changing the program's code. Variant: a program opens a file as read-only, and then attempts to write a byte into it, and naturally fails—make it succeed!

You are allowed to use any destructive actions in MDB and DTrace to achieve your goal (show transcript of MDB or DTrace commands).

Problem 7. *I am not afraid of you, exit(2)!*

Consider the following program, compiled as dynamically linked:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
        /*
         *   Add code here.
         */

        printf("Hello runtime!\n");
        exit(1);
        printf("Be seeing you, runtime!\n");
        return 0;
}
```

Add code where indicated to cause the program to proceed past the *exit()* to the final *printf*. You are allowed to add any pointer operations, arithmetic, and memory reads and writes, but no preprocessor directives, branches, loops, or system calls.

Problem 8 (open-ended) *Shhh. I am hunting pages!*

This problem is open-ended: the more comprehensive your solution, the more credit you will get. An in-depth exploration of these questions would make a good final project.

- Track a physical page through its uses by the kernel (which segments handle it, which subsystems use it, etc.) Cause some workload to exercise several different uses of the page—or as many as you can. How many different uses can you make a page go through?
- Track objects in a Kmem cache. How often do these get reused, on average? Pick a task such as browsing or compilation of a larger program or library to create a system load on the relevant object caches.

For these problems, you may need to first collect a log and then post-process it with your favorite scripting language.