CALAB Master Programming Project

(PART 1) User-level loader

Program loading and memory mapping

The goal of this assignment is to familiarize you with how programs are loaded, dynamically paged, and some of the mechanics of signal handling and memory mapping.

You are going to write a program that takes the file name of another program as an argument. Your program will load the named program into memory and let it execute. We will call your program a loader program because it loads the target into memory (and then executes it). We will also call it a pager because it potentially loads your executable into memory a page at a time.

Executables on Linux are stored in ELF format. Read about the format, as your loader program will have to read and interpret it.

The operating system, as one of its basic services, loads programs into memory for them to execute. Programs (like the shell) call execv or one of its variants to get the operating system to load the program into memory and start it executing as a process. Start this lab by reading the code that implements the execve system call in the Linux kernel (at least at one point the function was called do_execve_common) and describe in pseudo-code the steps that occur in creating the memory image of a new process. Focus on how the user-space memory is set up rather than how kernel data structures are modified. Pay attention to load_elf_binary. You might want to step through the execution of the kernel code in a debugger, like you practiced in the first lab.

Your loader will load the program image into its own address space and execute it from within its own address space. In this way, your user-level program is different from the execve code which is building a program to execute in a newly created address space. Eventually your loader transfers control to the entry point of the loaded program via a jmp instruction. Note that you do not call the loaded program's main function, you jump to the entry point. Think about what is different about these two actions.

Your loader will control the memory behavior of the program under test (the loaded program) using mmap. The kernel demand pages executables to reduce physical memory use. Your loader will (eventually) demand page to reduce virtual memory use (which then indirectly reduces physical memory use). Your loader is a user-level program that uses mmap and signal handling to do the kernel-like work of loading and demand paging an executable. The kernel manages page tables--your program manages mmap. The kernel manages memory faults--your program manages signals.

To make your job easier, you can statically link your loader program. You should also statically link the program under test. (These are two separate programs). I'd suggest linking each into disjoint address regions, and I'd suggest placing your loader program into a region that is currently unused by system libraries. Documentation for gcc and the linker (ld) will instruct you on how to statically link and how to change the placement of your code. Your loader does not need to dynamically relocate itself or the program under test, though supporting dynamic linking will surely get you extra credit. You must test for and gracefully exit in case your program under test wants to load itself on top of your loader program. So think about what should happen if you try to run ./apager apager (see the Interface section for executable names).

All-at-once loading

To start, have your loader map the entire program (with the proper permissions). Map an anonymous region for the program's bss. Write at least two test programs that your loader will load and execute. Measure the execution time and memory use of these programs when they are run by your loader. Describe what functionality of your loader your test programs exercise. What happens when the program under test calls malloc?

Hints

Read the man page for the readelf and/or objdump utilities and try them out on some binaries.

Make sure you map the sections from the file at the memory addresses specified in the ELF headers. You should treat the addresses in the headers as absolute addresses.

Some sections of an executable are not backed with file contents, so they will take up more space in memory than they do in the file (ex: BSS). Make sure that for the first part of the lab, you map the entirety of each section, not just the part backed by the file.

When transferring control to the child program, make sure the stack looks *exactly* like it would if the kernel were transferring control to the child. Specifically make sure that the auxiliary vectors are properly configured (see create_elf_tables as called from load_elf_binary in the Linux source code) and that the stack pointer points to right below argv on the stack before you use a jmp instruction to transfer control.

You should zero all register contents before starting the program under test. Libc checks rdx, and if it is non-zero it interprets it as a pointer during program shutdown.

A stack is thread-local storage. Allocate a new memory area for the program under test's stack, don't share your loader's stack with the loaded program. Of course you will have to carefully construct the initial state of stack memory for your loaded program, just as the kernel must prepare the stack of every process that executes.

Usually if you get a segfault during ctype_init from a zero %fs it is because your stack is set up incorrectly and libc thinks things are in different places than they are.

A potential workflow is (1) load the elf file (here is now linux does it); (2) build the stack; (3) use inline assembly to clean up register state and transfer control to the entry point.

Demand loading (demand paging)

Now implement demand paging for the program that your loader is executing. To start, only map a single page of the executable under test. Set up signal handlers to catch segmentation violations and any other signal you need. In the signal handler, determine what address your executable is trying to access, and map only that page. Run the same tests on you demand paged loader as your all-at-once loader. Demonstrate at least one program that runs faster for the all-at-once loader, and one that uses less memory for the demand paging loader. Present your findings.

Memory access errors

Consider the following program.

```
int
main() {
    int *zero = NULL;
    return *zero;
}
```

What does your demand pager do with this program? What should it do? Implement and describe a method for your demand pager to preserve memory access errors present in the program under test.

(optional) Hybrid loader

Now implement a hybrid loader that maps all text and initialized data at program startup, but maps all bss memory on demand. But for every fault, you may map 2 pages, the demand page and another page. Implement a prediction algorithm that choses the second page. Make whatever comparisons for whatever metrics you find most interesting. Find a workload that runs faster with your prediction algorithm than without and explain why. Also map 3 pages according to some heuristic and compare that to whatever is most illuminating.

Please report how much time you spent on the lab.

Interface

You should have a Makefile which generates 3 binaries: apager (all-at-once), dpager (demand paging), hpager (hybrid). They should accept a single command line argument which is the target program under test. Your loaders should print a message to standard error about every mapping they do for the test program. The message should include the base memory address, file offset (if applicable), and size.

Part 2: User-level thread implementation with the user-level loader

Understand the concept of user-level thread

http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Part 2 will be released on your request