Mel Krusniak 10/31/21
CPSC 429 Assignment 1 (Complete)

Overview

This lab report is similar to the submission from Parts 1-3, with more attention given to commentary on each part (and of course the addition of part 4.) For reasons I'll explain below, I implemented each part locally on my own machine, which does use the x86 64-bit architecture on an Intel i5 processor.

All code is accessible at the Git repository here. I've tried to include only original code within the files where it is inserted (but no other files - the kernel source is huge!), which is tricky because much of it goes inline into the kernel source code itself and there's no way to add a system call without building the entire kernel around it.

Part 1: Hello World from a kernel module
Part 2: Adding a new system call
Part 3: Interfacing from user space to the kernel on x86
Part 4: Business logic of `memuse`

Part 1: Hello World from a kernel module

Implementation

The code for Part 1 is in the `hello/` directory at the root of the Git repository.

I used the Linux Kernel Module Programming Guide to guide me through the process. The code is relatively simple, so I'll include it inline here:

```
// hello.c

#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("GPL");

int init_module() {
      pr_alert("Hello, world!\n");
      return 0;
}
void cleanup_module() {
      pr_alert("Goodbye, world.\n");
}
```

```
// Makefile

obj-m += hello.o
all:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

However, in this seemingly simple code there are a few interesting tidbits to note:

- **`pr_alert`**: This is essentially a macro for a call to `printk` with the alert urgency level. For reasons that I think are related to my kernel configuration, `dmesg` doesn't seem to respect urgency levels appropriately with `printk`, thus the workaround.
- **`obj-m += hello.o`**: This expresses the same idea as setting `KDIR` but uses the newer kbuild kernel build system. The targets below that also use kbuild.
- **`init_module, cleanup_module:`** These happen to be the default names for the corresponding functions in kernel modules, meaning there's nothing else I need to add to make them read properly.

Doing `dmesg | grep Hello` shows that "Hello world!" is printed as requested when the module is loaded with `insmod hello.ko` (after making the module).

Commentary

Writing a kernel module is pretty easy! As long as the appropriate parts are all there, it seems that there is little that can go seriously wrong, and deploying one is easy.

The functionality of kernel modules does seem to change over the years. For instance, I was required to explicitly specify a module license, which appears to be a new (and probably worthwhile) requirement. However, those changes aren't always well documented, so writing this kind of code is a good exercise in reading changelogs and Github comment chains.

Mel Krusniak 10/31/21
CPSC 429 Assignment 1 (Complete)

Part 2: Adding a new system call

The code for Parts 2 and 3 is in the `memuse/` directory.

Setup

By complete chance, I had an extra Ubuntu partition available on which I could test kernel changes. I did not use a VM as a result - I installed the kernel changes directly on my device. I used the 5.11.0-37 HWE Linux kernel.

Why can't modules add system calls?

There are a few reasons:

- **It's a security vulnerability.** Kernel modules are encapsulated in a way that prevents them from doing anything too irreversible to the computer (i.e., if one fails to load, it's not necessarily an unrecoverable failure.) On the other hand, if a kernel module managed to, for instance, corrupt the entry for a very important syscall like open or fork, the system might not even be bootable.
- **Modules should be separable from the kernel** in a way that doesn't make sense for a syscall. The nice thing about modules is that you don't have to recompile the kernel to add them - they can be loaded and unloaded. However, system calls are compiled with the entire kernel. So there would be a lot of logistical hurdles in adding a system call dynamically - for instance, you would need to manually specify libc wrappers.
- **Syscalls need to be standard and long-term.** One important feature of system calls is that they're dependable in the longest term and on all systems - many of UNIX's original system calls are still available. Modules, on the other hand, are not necessarily designed in that way.

What if the syscall number is wrong?

If we try something like `syscall(-1)`, the return value of `syscall` is -1. This is somewhat irritating, as we would also like to use a negative return value from the syscall itself (i.e., to indicate that the syscall was called, but something went wrong). The easiest way to avoid this kind of problem is to use the libc wrappers, avoiding the issue of specifying the right syscall number altogether (which is one reason why they exist).

Code walkthrough

Making memuse involved adding to a few kernel files, which I've noted inline below:

```
// kernel/sys.c

[...]

SYSCALL_DEFINE2(memuse, pid_t, pid, int, mem_type) {
    struct task_struct *p;
```

```
	p = find_task_by_vpid(pid);
	if (!p) {
		pr_alert("memuse: custom syscall could not
				find process with specified pid\n");
		return -1;
	} else if(! (mem_type == 1 || mem_type == 2 || mem_type == 3)){
		pr_alert("memuse: invalid memory type\n");
		return -1;
	} else {
		pr_alert("memuse: process with pid %d found;
				memory type %d\n", pid, mem_type);
	}
	return 1;
}
```

```
// include/linux/syscalls.h

[...]
asmlinkage long sys_memuse(pid_t pid, int mem_type);
```

```
// arch/x86/entry/syscalls/syscall_64.tbl

[...]
442 64     memuse                 sys_memuse
```

`sys.c` happens to be where many miscellaneous syscalls go. We have access to the entire kernel source code, so in this case there's no reason not to add memuse directly to that file - then we don't have to worry about configuring the build system to look for another file.

`syscall_64.tbl` is the system call table. As you can see, on my system the next available number happens to be syscall #442, so that will be taken by `memuse`.

Checking input validity

In `memuse`, I use the `find_task_by_vpid` function used by other system calls in `sys.c`. That function returns null on an invalid PID, which is very useful for us! We don't actually care what it returns yet, as long as it exists.

I didn't add any additional arguments to the syscall - so I'm assuming that as long as the PID is valid, we should be able to get the associated memory usage. We do have to check that the memory type argument is one of the three valid values, though.

Testing and running

We have to make and install the kernel (and restart the machine) before the syscall is ready:

```
<kernel_source>/$ sudo make -j4
<kernel_source>/$ sudo make -j4 modules modules_install
```

Then, to actually test the system call, a convenient thing to do is to use the runner's own PID:

```
// runner.c

[...]
printf("Own PID: %d\n", p);
printf("%d\n", syscall(442, 4522, 1));
```

This example calls the syscall with ID 442 (which happens to be memuse) and uses it to see if PID 4522 exists.

Commentary

The difficult part here is picking out specific parts of the Linux kernel from a very large system. (I once heard a quip that the Linux kernel is a bit like Rhode Island, in that if you want to say a codebase is "the size of Rhode Island" - that is, huge - you compare it to the Linux kernel.) The fact that the kernel is under active development is also, once again, somewhat confusing.

I hadn't realized before how much of the kernel was processor-specific. I know that one draw of many Unix-esque systems is that they can be run on even very old hardware or somewhat exotic architectures, and I guess this is the consequence. Fortunately, the relative popularity of x86 makes it easy to find guidance on implementing new features.

Part 3: Interfacing from user space to the kernel on x86

Implementation

Very fortunately (though maybe not surprisingly with x86 being so common), my machine has an x86 processor. This is important, because the instructions that give kernel access are processor-specific!

The `syscall` manual page indicates that it uses the value in the `%rax` register for the syscall number and uses `%rdi` and `%rsi` for the first two arguments. So to invoke a system call manually, we move the relevant values into those registers and then execute the `syscall` instruction:

```
// runner.c

[...]
// check for PID 1452
asm("movq $1452, %rdi");
asm("movq $1, %rsi");
asm("movq $442, %rax");
asm("syscall");
```

Maximum number of parameters

According to the syscall manual page, on x86 a maximum of six explicit arguments are permitted, which are stored in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` in that order - with the limiting factor being the number of registers available on the processor. (In the memory hierarchy, registers are the fastest, most expensive/hardest to add, and closest-to-processor place to store a value.)

If you wanted to pass more arguments than that, your best option would be to pass a pointer (perhaps to a struct), so you could store the additional arguments in main memory. Some architectures do this explicitly; the manual page indicates that the mips architecture always stores the first four arguments in registers and the last four in memory.

Commentary

Inline assembly is not a default C feature, but it seems to be the most precise way to specify what is happening under the hood here. In the actual implementation of the `syscall` function, I believe it is declared as `extern` with backing assembly code elsewhere, which amounts (in a somewhat less readable way) to the same thing. I wondered if assembly would be required to express a system call without the wrapper, but without having done anything similar before the difficult part of implementing this for me was finding the correct syntax for including assembly code in this way. With that done, implementation was a matter of reading manual pages carefully.

Part 4: Business logic of `memuse`

In order to actually check every page of memory in use, there's quite a bit more to do. We have a `task_struct` already - it's returned by `find_task_by_vpid` - and we can access the corresponding `mm_struct` easily. From there, we must scroll through each virtual memory area as `vm_area_struct`s (they form a linked list) and find those which seem to satisfy the criteria of the given memory type. Finally, for each virtual memory area, we check every page in that area for physical presence.

Getting memory type

This is not too difficult. The three types of memory usage we are charged with detecting are distinct enough that we have more than enough information to discern between them.

- **Call stack**: A block that grows down (as designated by flags in the `vm_area_struct`) must be the stack; this is its defining characteristic.
- **Heap**: `mm_struct` contains information on where the heap starts and ends, so any memory block that starts within that region is in the heap. (Heap entries do not seem to have the "`VM_GROWSUP`" flag. I assume this is because while the region where the heap may be can grow, individual entries themselves do not grow, but I'm not positive.)
- **Kernel**: The start of the stack is the highest user-level address; anything above that must be part of the kernel stack.

The relevant code snippet is below:

```
// kernel/sys.c

SYSCALL_DEFINE2(memuse, pid_t, pid, int, mem_type) {
      [...]
      int type = 0;
      if(vma->vm_start >= p->mm->start_brk
                  && vma->vm_start < p->mm->brk) type = 3; //heap
      else if(vma->vm_flags & VM_GROWSDOWN) type = 1; //call stack
      else if(vma->vm_start >= p->mm->start_stack) type = 2; //kernel
      [...]
}
```

Checking pages for physical usage

Actually getting page information from a virtual memory address requires a lot of offsetting and calculation which is abstracted away in a series of macros. For my system, the hierarchy is:

pgd ⇒ p4d ⇒ pud ⇒ pmd ⇒ `pte` (page table entry)

where we can use `pte_present()` on the final result to tell if the given page is actually loaded or not. From what I've read, the intermediate steps seem to depend somewhat on the system, with some omitting the p4d or pud levels.

The resulting loop looks something like this:

```
// kernel/sys.c

SYSCALL_DEFINE2(memuse, pid_t, pid, int, mem_type) {
      [...]
      for (vpage = vma->vm_start; vpage < vma->vm_end;
                                          vpage += PAGE_SIZE) {
            pgd_t * pgd = pgd_offset(p->mm, vpage);
            if(pgd_none(*pgd) || pgd_bad(*pgd)) {
                  pr_alert("memuse: bad pgd"); return -1;
            }
            p4d_t *p4d = p4d_offset(pgd, vpage);
            if(p4d_none(*p4d) || p4d_bad(*p4d)) {
                  pr_alert("memuse: bad p4d"); return -1;
            }
            pud_t *pud = pud_offset(p4d, vpage);
            if(pud_none(*pud) || pud_bad(*pud)) {
                  pr_alert("memuse: bad pud"); return -1;
            }
            pmd_t * pmd = pmd_offset(pud, vpage);
            if(pmd_none(*pmd) || pmd_bad(*pmd)) {
                  pr_alert("memuse: bad pmd"); return -1;
            }
            pte_t * pte = pte_offset_map(pmd, vpage);
            if(!pte) {
                  pr_alert("memuse: could not get page table entry");
                  return -1;
            }

            if(pte_present(*pte)) result += PAGE_SIZE;
      }
      [...]
}
```

Testing memuse

To test the call, it's useful to have some functions to increase different parts of memory:

```
// runner.c

void increase_heap_active(int s) {
	char * a = malloc(s);
	for(int i = 0; i < s; i++) a[i] = 'a';
}

void increase_heap_passive(int i) {
	malloc(i);
}

void increase_stack(int i) {
	if(i > 0) increase_stack(i-1);
}
```

There's an important distinction here: when we increase the size of the heap (i.e., with `malloc`), we'll see different behavior based on whether we actually use the resulting space or not. This is why I've given both an "active" and "passive" option to increase the heap.

By running these functions as desired in runner.c we can see `memuse` track their exact effect:

	Call stack (bytes)	**Kernel stack** (bytes)	**Heap** (bytes)
Do nothing	12288	4096	4096
Increase heap (active) 20000 bytes allocated & used	12288	4096	24576
Increase heap (passive) 20000 bytes allocated, not used	12288	4096	8192
Increase stack 20000 calls made	651264	4096	4096

As we might expect, the call stack increases in size with recursion and the heap increases with allocations. If we do an allocation but don't use the contents, only the first page of the allocated memory is actually loaded, which memuse correctly detects. (There is also some heap usage before we do `malloc` at all. Unix seems to usually store `argv` on the stack, but it could be something else in the process outside the scope of `runner.c`.) Finally, the kernel stack is never more than one page long, no matter the stack or heap growth.

Mel Krusniak 10/31/21
CPSC 429 Assignment 1 (Complete)

Commentary

About halfway through implementing `memuse`, I realized I had made some decisions that would affect my programming speed. Specifically, every time I changed the call, I would have to rebuild the kernel and restart my system to allow it to be used. While it's exciting to be able to modify my computer's internals so easily, this was a tedious process! It probably would have been much easier if I had set `memuse` to call an external kernel module which could be loaded, as suggested in the spec - but I worried that I would spend just as much time implementing that.

On the other hand, there is something satisfying about specifying exactly every moving part in the system call. I have a feeling I may find in the next assignment that this will be easier in a higher-level language, but I also feel that I have a better idea of what's going on in my system, or at least I understand the scope of what I don't know. (For instance, without detailed understanding of paging, there is no way to diagnose issues such as memory thrashing - but most high-level languages and systems work hard to abstract those details away.)