18648 Lab1 Writeup

Mengwen He, Tarun Ala, Hongbao Zhang

September 30, 2016

1 How does a system call execute? Explain the steps in detail from making the call in the userspace process to returning from the call with a result.

In general, the hardware of the CPU enforces that a process is not supported to be able to access kernel memory and call kernel functions. However, system calls are an exception to this general rule, and they provide a set of communication interfaces between user-space and kernel-space.

1. A process in the user-space makes a system call.
   - Explicitly/implicitly(wrapped library) calls the function *syscall(system call number, arguments...)*.
2. The process fills the registers with a system call number and several arguments.
   - On x86, the system call number is fed to the kernel via the *eax* register.
   - On x86-32, the registers *ebx*, *ecx*, *edx*, *esi*, and *edi* contain, in order, the first five arguments.
   - In the unlikely case of six or more arguments, a single register is used to hold a pointer to user-space where all the parameters are stored.
3. Then it calls a special instruction (software interrupt or trap).
   - The software interrupt triggers a switch from the user-mode to the kernel-mode. On x86, the interrupt number is 0x80.
   - Then this trap jumps to a well-defined location in the kernel(that location is readable by user processes, but it is not writable after the Linux kernel 2.6.x(but you can use hook method to modify it). Its location in the Linux kernel is System.map) for the execution of exception vector 128, the trap handler named *system_call()*, which is architecture-depended and implemented in assembly in entry_64.S for x86-64.
4. The execution of *system_call()* checkes the system call number, which in turn represents what kernel service the process requested.
   - It firstly checks the validity of the given system call number by comparing it to *NR_syscalls*. If it is larger than or equal to *NR_syscalls*, the function restures *-ENOSYS*.
   - Otherwise, the specified system call is invoked for the next step: *call * sys_call_table(, %rax,8)* (*rax* is the reqister storing the system call number)
5. The OS uses the function *sys_call_table()* to look for the address of the kernel function to call from the table of system calls.
   - The system call table on x86 is defined in syscall_64.tbl.
   - Because each element in the system call table is 64 bits, the kernel multiplies the given system call number by 4 (left bit-shift 2) to arrive at its location in the system call table. On x86-32, the code is similar, with the 8 (left bit-shift 3) replaced by 4.
6. The OS calls the function found from the system call table, and after it returns, does a few system checks and then return back to the process in user-space.
   - The return value is sent to user-space also via register. On x86, it is written into the *eax* register.

2 Define re-entrancy and thread-safety.

- In computing, a computer program or subroutine is called reentrant if it can be interrupted in the middle of its execution, and then be safely called again (”reenterted”) before its previous invocations complete execution. This reentrant function allow multiple concurrent invocations that do not interfere with each other.
- Reentrancy is the property of a thread to execute while a previous invocation of the thread is running concurrently along with it. Hence there should be no dependency between both the threads such as static or global data. They shouldnt modify their own code and cannot work with resources that are locked at some point.
- Thread safety is the ability of the kernel to run multiple threads at the same time without encountering any problems with any of the problems. This can be achieved by locking critical resources and scheduling the threads accordingly

3 What does it mean for a kernel to be preemptive? Is the Linux kernel you are hacking on preemptive?

Preemption is the process of stopping the execution at the moment so that another process with a higher priority could run. The stopped task can later be run by storing the data in register or rolling back to the initial state. The kernel is preemptive. This shows up when Uname -a is run on the tablet.

4 When does access to data structures in userspace need to be synchronized?

If a thread is executing in user space it cannot access the physical memory directly. It has to go through kernel’s virtual memory management. In this case we do not need synchronization. Critical section is part of the code that if executed by two processes in parallel, it could result in data corruption. This would be happening when the code is accessing some shared resource. Synchronization is needed in order for avoiding the corruption of a shared resource. Even in the case where two process/thread are running in kernel mode and want to access a shared resource, they need to apply Synchronization mechanism (spinlocks or mutex).

5 What synchronization mechanism can be used to access shared kernel data structures safely on both single- and multi-processor platforms?

Disabling Interrupts to Implement the Critical Section

Disable interrupts while a process enters its critical section, then enabled when it leaves it.

Lock Manipulation as a Critical Section

Use the enter( ) system call to wait for a critical section to become available.

Semaphores, (the basis of modern solutions) Preemption is the process of stopping the execution at the moment so that another process with a higher priority could run. The stopped task can later be run by storing the data in register or rolling back to the initial state. The kernel is preemptive. This shows up when Uname -a is run on the tablet. The semaphore will allow one process to control the shared resource while the other process waits for the resource to be released.

The Driver-Controller Interface Behavior

The busy and done hardware flags are used like signaling semaphores. The driver communicates with the controller by setting busy, and the controller informs its state to driver using the done flag.

6 What is the container_of macro used for in the kernel code? In rough terms, how is it implemented?

A convenience macro that is used to obtain a pointer to a structure from a pointer to some other structure contained within it.

(1) to_i2c_driver() macro is defined as: container_of(d, struct i2c_driver, driver)

and is used in code as: i2c_drv = to_i2c_driver(drv); where drv is a pointer to a struct device_driver.

(2) It becomes i2c_drv = (const typeof(((struct i2c_driver *)0)-¿driver) *__mptr = drv; (struct i2c_driver *)((char *)__mptr - offsetof(struct i2c_driver, driver));) when we do macro expansion

(3) The first line of the macro sets up a pointer that points to thestruct device_driverpassed to the code. The second line of the macro finds the real location in memory of thestruct i2c_driverthat we want to access.

(4) Suppose: struct i2c_driver char name[32]; struct device_driver driver; ;

above equation can be reduced to i2c_drv = (const struct device_driver *__mptr = drv; (struct i2c_driver *)((char *)__mptr - 0x20);)

container_of allows you to simplify your data structures by omitting pointers to parent structures.

It's used within the linked list implementation so that the list node can be an element of any structure, and anyone can find the parent structure without carrying around an explicit pointer.

Another example is struct work_struct. A workqueue work function receives a work_struct as an argument, and it used to have a generic "data" payload. This data value was removed, making the structure smaller, as the work function can call container_of to find its parent structure.