Project 8: System Calls (4%)

ENEE 447: Operating Systems — Spring 2016

Assigned: Monday, Apr 4; **Due:** Friday, Apr 15

Purpose

In this project you will implement the system-call (or "syscall") facility that you designed in class. The set of system calls that an operating system provides is the user application's *de facto* API to the system, making it an extremely important facility.

The System Call Facility

The operating system's syscall facility has to provide several functions that are often at odds with one another:

- It has to provide access to hardware such as permanent storage, networks, video monitor, user input (keyboard, touch screen, mouse), audio output, haptic output, timers, clocks, etc. User applications should be able to perform all or nearly all functions that the hardware mechanisms offer. The division between what is exposed to the user and what is not exposed is a trade-off that the OS designer must work through. For instance, a user-level facility should be able to store something permanently into a flash device; should it also have the ability to initiate garbage collection within that flash device? Probably not. Exposing that to user software would simplify OS design but would create potential security holes.
- It has to provide certain performance guarantees to the user applications. The API and its implementation has to be designed such that no one application can hog a particular resource, thereby cutting off access for other applications. Such denial-of-service attacks may actually be unintentional: for instance, a buggy application could accidentally fill up all available disk space or consume all available network bandwidth. The operating system should prevent this. In addition, the lower the overhead that the operating system imposes for its management of hardware resources, the better. Thus, there is an obvious trade-off: one could guarantee the security of hardware facilities through multiple layers of checks, management code, watchdog timers, etc., but doing so might slow down an application's access to the hardware in question, to the point that nobody bothers to use the facility in question (or the operating system in question), favoring lower-overhead designs. For instance, this is why the Windows PC was such a successful gaming platform for so many years: the operating system basically gave up all aspects of security to the running application, so that the game software could extract every bit of potential performance from the machine. Now that hardware platforms, even hand-held ones, have tremendous performance characteristics, such a trade-off is no longer unnecessary, and thus one sees numerous games manufacturers today embracing more secure platforms like iOS.
- It has to protect the hardware from the user application. Not *all* hardware facilities should be exposed to the user application. As mentioned above, the division between what is exposed to the user and what is not exposed is a trade-off that the OS designer must work through: exposing everything makes the job of the OS simpler in one regard (less to do), but much harder in another (lots of unanticipated consequences, such as security holes). Buggy applications, if given low-level access to hardware mechanisms, can not only cause unintended havoc, they could even damage the hardware in question, and so the operating system should provide higher-level access to the mechanisms that make it more difficult for programmers to make mistakes.

- It should raise the level of abstraction significantly. This is an extension of the previous point. The operating system's syscall facility should make it easier for user-level software *not* to cause problems: hardware interfaces are notoriously difficult to program, and notoriously easy to screw up. If one simply exposed the low-level hardware interface to the user application, then accessing the hardware would necessarily cause lots of headaches. The syscall facility should allow user-level code to interact with the hardware without having to know *any* of the low-level details. For instance, disks are block-level devices that one accesses via control registers; yet user-level code only interacts with the disk through file names and read/write operations of arbitrary size. Similarly, you have been interacting with the RPi's LEDs through control registers for months now; a user-level application should not have to know any of that.
- It has to protect the operating system from the user application. Last but not least, the syscall facility needs an implementation that does not compromise security, such that the operating system's protections can be subverted. User-level applications should not be able to "root" the machine; user-level applications should not be able to read or write each other's data in memory, in flash, or on disk; user-level applications should not be able to bypass the operating system and talk to hardware resources directly (unless that is a specific design decision).

The interfaces that a user application has with which to interact with the operating system are the register file, main memory, and a single *trap* instruction. Thus, any interface is relatively restricted. You made the following design decisions:

1. the interface will be through the register file
2. register file values will include:
    - function number
    - device number
    - pointer to an argument array
    - size of array
3. system call types will include I/O and process types
4. The functions to support include the following:
    - LED operations (write)
    - print to screen (write)
    - SD card access (read/write)
    - timer/clock operations (read/write)
    - create new process (creates an address space and a thread within it)
    - create new thread within existing process
    - yield CPU to another thread/process

All of these operations can be handled by the following system calls:

- int read (int device type, int id, char *buf, int len);
- int write (int device type, int id, char *buf, int len);
- int new_process (char *executable);
- int new_thread (pfv_t entrypoint);

- void yield();

These can all be implemented through a syscall facility in which the arguments are collected into an array of the form that the receiving function expects.

Your Task

Build a syscall facility in your operating system. It should have the following components:

User-Level Functions

Implement the following functions in a separate module:

- int read (unsigned int device type, int id, char *buf, int len);
- int write (unsigned int device type, int id, char *buf, int len);
- int new_process (char *executable);
- int new_thread (pfv_t entrypoint);
- void yield();

These should all take in the various arguments and call the syscall wrapper, described below. The **int id** values are placeholders for you to be able to indicate things like files, or executables, or different types of clock/timer, etc.

Syscall Wrapper

Write a single function that collects arguments into a single form that the kernel's syscall handler can unpack and use. The syscall wrapper should look like the following:

```
int    syscall ( int function, int device ID, char *args, int argsize );
```

This is what the user-level functions call. You need to define the read/write functions and the various device IDs in a common header file (for instance, "syscall.h"), and that should be used by both the syscall wrapper and the kernel trap handler. The function **syscall**() should take its inputs and assemble them as follows:

- **r0** will contain a char * pointing to a memory location
- **r1** will contain an unsigned int specifying the size of the character array in bytes
- **r2** will contain an unsigned int specifying the device ID, or a zero value, meaning *none*
- **r3** will contain an unsigned int specifying the function number, or a zero value, meaning *none*

Once these are in place, **syscall**() should call the ARM's **svc** instruction, as we did previously in the semester. When the **svc** instruction returns, the **syscall**() function should return to the user code whatever value is left in **r0**, which is the return-value register as specified by the ARM API.

Devices

You should implement read/write handlers for the following device types:

1. LED
2. monitor
3. SD card
4. timers/clocks

Trap Handler

The SVC handler is invoked when an ARM **svc** instruction is executed. Your handler should look in **r2** for the device ID. If it is zero, there is no device; or, rather, there is a *default* device. Use the value as an index into a device array to locate a set of interfaces for that particular device.

Your handler should next look in **r3** for the function number. If it is zero, there is no function; or, rather, there is a *default/null* function. For each device there will be a pointer for a **read**() function, a pointer for a **write**() function, and a pointer for a **null**() function (when the function number is zero). For the default device there will be a pointer for the **null**() function (when the function number is zero), and any number of functions beyond: you must at least implement **newproc**() and **newthread**(). The pointers should be arranged in an array, which is indexed by the function number, noting that the **null**() function should be the one indexed by a 0 value for the function number. The interrupt handler should index the array and call the function. When the system-call function returns, the interrupt handler should return back to user code.

System-Call Functions

As mentioned above, for each device there will be a **read**() function, a **write**() function, and a **null**() function (for when the function number is zero):

- int null (void);
- int read (char *args, unsigned int len);
- int write (char *args, unsigned int len);

For the default/null device you should have the following functions:

- int null (void);
- int newproc (char *args, unsigned int len);
- int newthread (char *args, unsigned int len);

Pointers to these functions are arranged in arrays reached through the device ID. Each of the functions, for this project, should simply print to the screen a “hello world” message to indicate which function for which device was reached.

Build It, Load It, Run It

Once you have it working, show us.