Operating System Concepts
What is an operating system, what does it do, and how do you talk to it?

Paul Krzyzanowski
Rutgers University

January 26, 2012 [original version: September 13, 2010]

1 What's an operating system?

A number of definitions of **operating system** abound. A basic definition could be that it's *a program that lets you run other programs*. If it didn't do at least that then we would never have programs running on our computers except for the operating system itself. We need to differentiate this from a command interpreter or a windowing system, however, either of which could be just another program that requests the operating system to run other programs based on user requests. Another definition that we can use is that *an operating system is a program that provides controlled access to a computer's resources*. These resources include the CPU (process scheduling), memory (memory management), display, keyboard, mouse (device drivers), persistent storage (file systems), and the network.

At the most basic level, the operating system provides a level of abstraction (a set of APIs) so that user programs don't have to deal with the details of accessing the hardware. This is essentially what early operating systems in the 1950s gave us as well as what we got from early personal computer operating systems, such as MS-DOS. On a more sophisticated level, the operating system is also responsible for enforcing proper protections and some concept of "fairness". You don't want a process to use an unfair share of the memory or CPU. You also don't want one process to be able to overwrite the memory of another or to access files on the disk that another user wants restricted.

1.1 Building an operating system: monolithic, layered, modular?

Older operating systems were largely **monolithic**. Every system-related function was performed by one block of code that was part of the operating system.

As the functionality of the system grew, the operating system software became increasingly more difficult to maintain and understand, so operating system software started to get modular. A **modular** design allows certain pieces of the operating system to be replaced or installed as needed. For example, components such as the process scheduler can be replaced with alternate versions. Specific device drivers, such as iSCSI or USB can be created as a separate modules and "plugged in" to the kernel (linked into the kernel after the base part has loaded). The modules of Linux are an example of this structure.

The later trend in operating system design was to keep the core of the operating system small and move as much as possible into separate programs that have the proper security privileges to do what just they need. The kernel will invoke separate processes to deal with things such as parsing a file system, interfacing with the network, or even making a process scheduling decision. This approach is called a **microkernel**.

1.2 What's a kernel?

When people refer to an operating system, they often talk not just about the program that controls access to resources, but also the command interpreter, utilities, and other programs and drivers that make up one's perception of an operating system — the entire operating environment. The **kernel** is the core component of the operating system; the central program that manages resource access and process scheduling.

An example of some of the things a typical operating system kernel does is:

- controls execution of processes: allowing their creation, termination, and inter-process communication
- schedules processes fairly for execution on the processor or processors
- allocates memory for an executing process. If the system runs low on free memory, it frees memory by writing regions of memory to secondary memory. If it writes the entire process, then the system is a swapping system. If it writes pages of memory, it is called a paging system.
- allocates secondary memory for storing and retrieving data (file systems) and enforcing proper access permissions and mutual exclusion
- allows processes controlled access to peripheral devices: displays, keyboard, mouse, disk drives, networks — also enforcing proper access permissions and mutual exclusion

2 Am I in user mode?

A processor contains certain instructions that are only available to "privileged programs", such as the operating system. These instructions include those to manipulate memory mappings, set timers, define interrupt vectors, access restricted memory, or halt the processor. Clearly, it is not a good idea for regular programs to be allowed to mess with these instructions since they could wreak havoc on the system. They will also be able to circumvent any protection mechanisms that have been put in place since a process will now be able to interact with any device and any memory as it sees fit.

Most processors have a flag in a status register that tells them whether they're running in **user mode** or **kernel mode**. Kernel mode is also known as **privileged**, **system**, or **supervisor** mode. If a process is running in kernel mode, it has full powers to access all of these restricted operations. If a process is running in user mode, it does not.

The processor starts up in kernel mode and the boot loader runs in kernel mode. The operating system is loaded and, it too, runs in kernel mode. The processes that it launches run in user mode.

Note: When we cover machine virtualization, we will see that there's a third mode for running a hypervisor (a virtual machine manager). Don't worry about this for now. Also, most

Intel processors start up in what's known as *real mode*, which is a 16-bit 8086-compatible mode with 20-bit addressing. From this mode, the processor can switch to 32-bit or 64-bit mode. Intel processors actually have additional modes but we won't concern ourselves with them here.

2.1 Switching between user and kernel modes

If a processor is running in kernel mode, it can switch itself to user mode by setting the status register that defines its operating mode. Once it is running in user mode, however, it cannot simply set itself to run in kernel mode since that would be a privilege violation.

Traps, also known as **software interrupts**, let us do the user mode to kernel mode transition. A trap is an instruction that forces the program to jump to a well-known address based on the number of the trap. The trap (or a hardware interrupt, for that matter) acts as a spontaneous subroutine call. The current program counter is pushed on the stack and the program jumps to a well-known address. That address usually contains a *jump* instruction (*vector*) to the code that will handle that trap. The action of taking a trap causes the processor to switch from user mode to kernel mode.

The entire set of trap vectors (all the addresses that various parameters to a *trap* instruction jump to) is set up by the boot loader and can later be modified by the operating system when it starts up. When a user program executes a trap (e.g., INT instruction on older Intel processors; other processors and all current Intel/AMD processors have some form of a *syscall* instruction), control is transferred to the appropriate trap vector and the processor now runs in kernel mode. Operating system code is run and, when the operating system is ready to transfer control back to the user, it executes a **return from exception** instruction (IRET on Intel processors). That switches the processor back to user mode operation and transfers control to the address on the top of the stack.

2.2 Violations

What happens if a user program tries to execute an instruction that is available only in kernel mode? ... Or tries to access a memory location that is not available to it? What generally happens in this case is that a trap takes place. Depending on what happened it would be one of several traps, such as a memory access violation, an illegal instruction violation, or a register access violation. The trap switches the processor's execution to kernel mode and switches control to the operating system, which then decides on a course of action. The address is defined by the trap vector, which is set up when the operating system starts up.

The attempted operation may have been a valid one, such as accessing a perfectly valid memory location that the operating system just did not load into memory yet. In this case, the operating system would load the needed chunk of memory and restart the instruction (return from exception). If the action was not a valid one, such as attempting to access an invalid memory location, the operating system may send a specific signal to the process. If the process does not have a signal handler for it, it will just be killed. When we look at machine virtualization, we will see that there's a case where the operating system may need to simulate the action of the privileged instruction and then return control back to the next instruction in the user's process.

To summarize, there are several ways where the flow of control switches to the kernel:

1. Via a software trap (interrupt). For example, via an INT instruction or a system call instruction, such as the *sysenter* on Intel processors, *syscall* on AMD processors, or *swi* on ARM processors.

2. Because of an access violation (invalid memory, invalid instruction, etc.).

3. Via a hardware interrupt, such as a timer or a network device that is tied to the processor's interrupt controller.

Another note: Intel processors before the Core 2 Duo did not generate a trap when a user program tried to execute a privileged instruction; they just ignored it. As we'll see when we look at virtualization, this made creating a virtual machine a big headache.

3 The system call

A program often needs to read a file, write to some device, or maybe even run another program. All these are require operating system intervention. The interface between the operating system and user programs is the set of system calls (*open*, *close*, *read*, *fork*, *execve*, etc). Making a **system call** uses the trap mechanism to switch to a well-defined point in the kernel, running in kernel mode.

To execute a system call usually involves storing the parameters on a specially created stack, storing the number of the system call (each function has a corresponding number), and then issue a trap to switch control to the operating system operating in kernel mode.

For example, a call to the *getpid* system call (get process ID) on Intel/Linux systems puts the number 20 into register eax (20 happens to be the number corresponding to the *getpid* system call) and then executes INT 0x80, which generates a trap.

The kernel will have to be careful to save all user registers and restore everything before it returns.

All this ugliness is hidden from the programmer with a set of library routines that correspond to each of the system calls. The library routines save the parameters, issue the trap, and copy the results back onto the user's stack so the system call looks exactly like a regular function call.

4 Preemption

How do we ensure that the operating system always gets a chance to regain control over the computing environment? For example, we don't want to have one process spinning in a loop that is keeping the operating system or any other process from getting anything done.

To give the operating system a chance to to run periodically, we program a timer interrupt. On Linux/Intel systems, we set the 8254 (or 8253) Programmable Interval Timer to generate an interrupt (IRQ 0) approximately every 10 milliseconds, although this interval can be modified based on the environment. This means that, no matter what is going on with a process, the operating system will receive a periodic interrupt from the timer and get to run. It may do a number of things when it runs: increment a time of day counter; see whether any input is waiting on input devices; see whether any outstanding transmits have completed; and see whether it is time to let another process run. This last decision is called **process scheduling** and is handled by the **scheduler** component of the operating system. If several processes are loaded in memory, the operating system can resume any one of them by loading the appropriate process' saved state into the processor's registers and executing a return from exception.

An interrupt or trap results in a **mode switch**. This simply means that the processor is switched from running in user mode to running in kernel mode.

This mode switch is often accompanied by a **context switch**. A context switch means that the processor's context (its registers, flags, program counter, stack pointer, memory mapping, etc.) is saved in the kernel's memory. When the kernel is ready to switch control back to a user process, the scheduler component decides which process gets to have the processor next. It then restores the saved state of the process and returns from the trap. A mode switch does not save and restore the program's context. A context switch does.

5 Devices

Peripheral devices are most any hardware that connects to your processor. They could be components built onto the main board or external plug-in devices. Peripherals include input/output devices that interact with the user, such as mice, keyboards, hardware audio codecs, video displays, and printers. They also include mass storage devices such as disk drives, Blu-ray players, and flash memory cards. Other devices include network controllers which allow computers to communicate with other computers over a communications network.

Some devices, such as disks, are addressable. For instance, you may be able to read block 2107 on a disk and later on read that same block again and get the same data. These are called **block devices**. Block devices lend themselves to **caching** — storing frequently used blocks in memory to avoid having to fetch them from the actual device, which would take much more time. This operating system cache of frequently used data blocks is called a **buffer cache**.

Devices such as keyboards, printers, and mice have non-addressable data. They either produce or consume an ever-changing stream of data. Such devices are called **character devices**. They send and/or receive byte streams and are not suitable for long term caching.

Finally, **network devices** are similar to character devices (and often lumped in the same category) in that they operate on ever-changing streams of data instead of addressable blocks. They are often put into a separate category because networks are packet-based and send and receive messages instead of arbitrary byte streams. Depending on the type of network and communications employed, some data might be considered special and be sent as expedited packets. Packet schedulers can also affect how packet streams are interleaved on the network link.

The code for managing a specific device is called a **device driver**.

5.1 Accessing devices

Devices, such as network or USB controllers, come with their own control logic that is usually governed by a microcontroller or some other processor that resides on the device. The control logic supports some form of programming interface to allow the host system to send requests to transmit data, receive data, see if data is ready, write data, read data, seek to a location, or get the status of a device. The actual functions, of course, depend on the device.

An clean way of interacting with peripheral devices is to map the device's command registers onto the system memory bus. This way, accessing certain memory locations will not reference system memory but instead read and write data to the device registers. The beauty of this scheme is that the programmer can use standard memory load and store instructions. Moreover, the memory protection mechanisms provided by the processor and set up by the operating system now cover device access too. This technique of mapping device input/output to memory is called **memory-mapped I/O**.

Device registers also tell you when data is ready (e.g., a packet has been received) or if the device is ready for more data (e.g., a packet has been transmitted). We could have our operating system sit in a busy loop and check these registers. Unfortunately, we will chew up tons of processor cycles just waiting for the device and not be able to use the time to do anything else. Instead, whenever we get a periodic clock interrupt (e.g., every 10 milliseconds), we can have the operating system go and check those device registers. That avoids the busy loop.

An alternative, and more popular, approach to this is to have the peripheral device trigger an interrupt whenever the device has data or when the device informs the system that it has finished transmitting and can accept more data. The benefit of an interrupt is that the processor does not have to do any checking until the device pokes it. The downside is that the interrupt will force a mode switch and that may be time consuming. On some systems, it may also invoke a context switch, which is even more costly. On startup, a device driver registers an **interrupt handler**. Whenever the interrupt takes place, the interrupt handler is invoked and is responsible for handling that interrupt. Since interrupts are precious resources (a processor may have only 16 or so of them), a device driver may sometimes have to share an interrupt line with other device drivers. In this case, the kernel will call each driver that registered that interrupt in sequence.

Moving data between the device and the kernel can also be accomplished by reading and writing the device registers (i.e., reading and writing specific memory locations to which those registers have been mapped). This is called **programmed I/O** since software is responsible for moving the bytes between main memory and the device via the device's registers. Some devices support **DMA** (direct memory access). This is a technique where the device is allowed direct access to system memory and can, on its own, read data from a region of memory or write data to it. The most common use of DMA is for disk access, where the disk controller will read a requested block of data from the disk, transfer it to a specific region of the system's memory, and then indicate (e.g., via an interrupt) that the operation is complete and the data is ready.

6 Files and file systems

In operating systems, a **file** is traditionally a collection of data with a name associated with it along with other information (length, create time, access time, owner, users who can modify the file, etc.). In many systems, a file may also refer to a device or even a process. Files are often organized in a hierarchical structure into sets of directories (sometimes referred to as *folders*, mostly by users of windowing interfaces that adopt a desktop metaphor) and files. A directory is a name of an entity that contains zero or more files or directories. A file system is generally a collection of directories and their files and subdirectories that occupies an entire disk or a defined section of a disk. It is a region of storage upon which the logical structure of file attributes and contents is laid.

The file system is a logical construct a bunch of data structures stored on the physical disk. The device driver for the disk sees it simply as a collection of fixed-size blocks that it can read and write. The operating system provides a **buffer cache** for storing frequentlyaccessed blocks in memory. The **file system** is responsible for managing the file system that is implemented on those blocks. Most operating systems support a variety of file systems. For instance, an Apple Mac's native file system is HFS+ while an SHXC card that's plugged in from a digital camera is formatted with a Microsoft exFAT file system and a Blu-ray disk will have a UDF (Universal Disk Format, also known as ISO/IEC 13346) file system on it.

7 When I log in, don't I type commands to the operating system?

Yes, with some systems, mostly older and more primitive ones. With CP/M and early versions of MS-DOS, the console command processor was part of the operating system. Usually, the user's interface to the system is a command interpreter, also known as a **shell** or a window manager. On UNIX systems, the name of the shell program is specified for each user in the password file. The shell is run when the user logs in. The windowing system may be spawned from a startup script for that user. The UNIX shell is a little language that supports redirecting input and output from and to other programs and/or files and allows conditional tests, looping constructs, and variable assignments. Together with the standard UNIX system utilities, one can use the shell as a basis for significant programming projects.

8 What's a process?

A **process** is ultimately the reason why we have an operating system and the computer and is thus one of the most important concepts in operating systems. A process is essentially a program in execution (one program may correspond to no processes if it's not running or to several processes if it's run multiple times). On a **multitasking** (or multiprogramming) system, several processes appear to run at once. In reality (unless there are multiple processors, in which case it's also a multiprocessing system) that is not the case. The processor is continually stopping one process and starting another, context switching among them. Multitasking works as well as it does because most programs (especially interactive or data intensive ones) spend most of their time sleeping and waiting on input or output. Operating systems that work like this are called **time-sharing systems**.

Processes may start other processes. A new process becomes a child of the parent process. Processes can be able to send signals to each other, send messages and share data.

9 Structure of a typical operating system

10 Using system calls under UNIX systems

All system calls under the various versions of UNIX (and indeed on just about every other operating system) are enveloped in library functions. To the user they appear no different from any other system library functions. On most UNIX systems (various flavors of Linux, OS X, FreeBSD, OpenBSD, NetBSD), the functions that invoke the system calls are in the library called libc, which also includes non-system call functions (such as *printf* and *strtok*). System calls are documented in section 2 of the programming manual and can be found on-line with the man command. For information on using the man command, run man man, which will show the manual page for the man command.

Most system calls return a positive result upon successful completion (check the man page for the particular call you're using) and a –1 for an error. In the case of an error, a

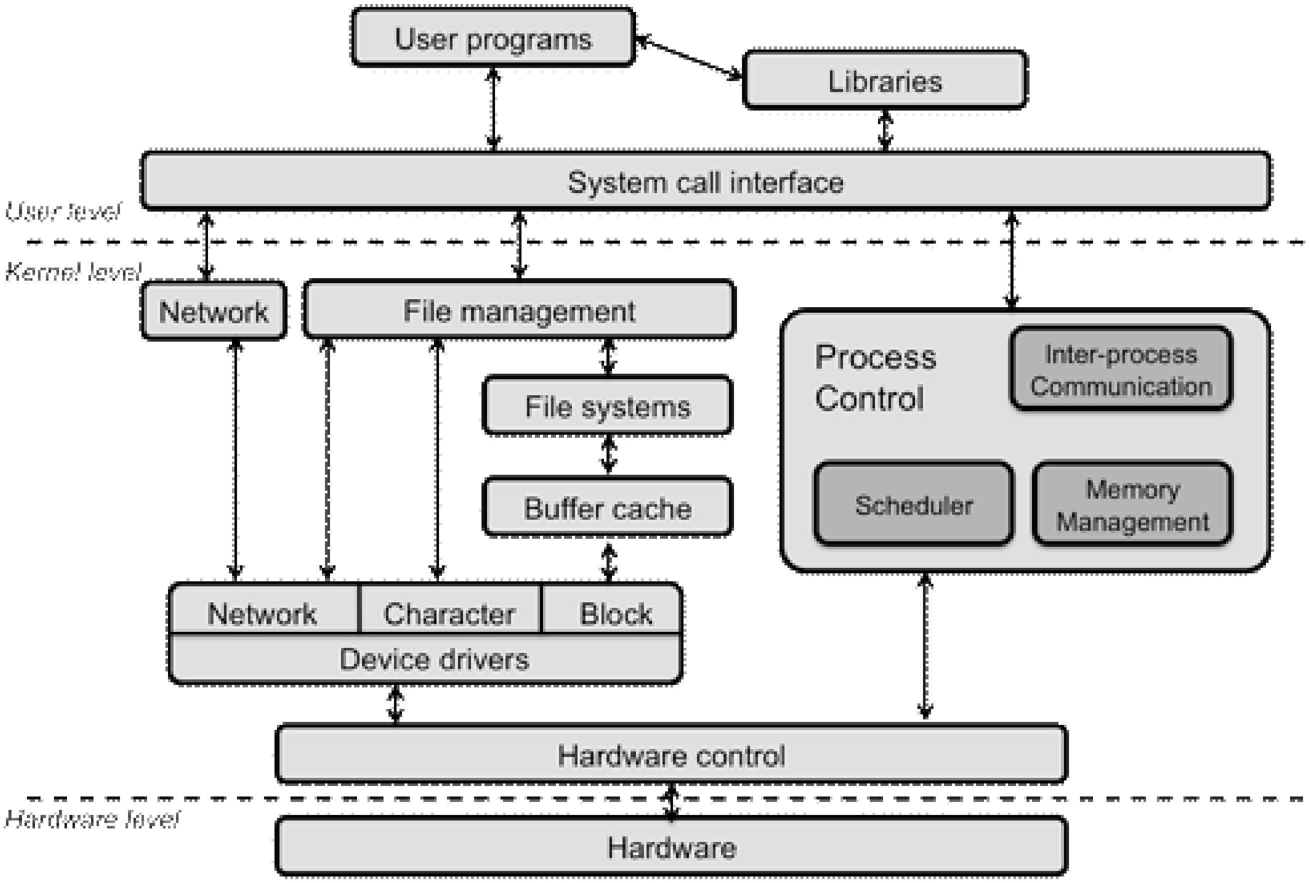

Figure 1: OS Structure

global variable, errno, is set to a particular error code. An exhaustive list of error codes is described in the introduction to section2 of the manual (man2intro) and is listed in the include file errno.h (/usr/include/sys/errno.h). The library function (not system call) *perror* consults a table to print the error as descriptive text. This is often useful for debugging (for info, run manperror). For example, the *stat* system call gives us information about a file. Running man2stat tells us what it does and how to invoke the function (it also refers us to the man page for *mknod* to find out how to check for the mode of a file). We can write a little program that will loop over each argument and print the file size (in bytes) for that file:

Download this file

Save this file by control-clicking or right clicking the download link and then saving it as stat.c. Compile this program via:

```
gcc -o stat stat.c
```

If you don't have gcc, You may need to substitute the gcc command with cc or another name of your compiler. Run the program:

```
./stat *
```

to see info about the files in your current directory. Run the program with an argument that does not correspond to a valid file name to see *perror* in action.

11 References

- Linux Multitasking[1], Chapter 6, KernelAnalysis-HOWTO, Linux Documentation Project
- Kernel command using Linux system calls[2] - Explore the SCI and add your own calls, M. Tim Jones, Consultant Engineer, Emulex IBM developerWorks Technical Library
- The Implementation of a Linux System Call[3], Phil Kearns, Fall 2008
- Linux Device Drivers[4], Chapter 9, by Alessandro Rubini & Jonathan Corbet 2nd Edition June 2001, ISBN 0–59600–008–1

[1] http://www.linux.org/docs/ldp/howto/KernelAnalysis-HOWTO-6.html

[2] http://www.ibm.com/developerworks/linux/library/l-system-calls/"

[3] http://www.cs.wm.edu/∼cjk/syscall.pdf

[4] ttp://www.xml.com/ldd/chapter/book/ch09.html