CS 390
Chapter 2 Homework Solutions

2.1 `What is the purpose of ...`

System calls are used by user-level programs to request a service from the operating system.

2.2 `What is the purpose of ...`

The purpose of a command interpreter is to provide a simple environment in which users may perform common computing tasks. The command interpreter should be thought of as a software layer that sits between the user and the other systems programs. The command interpreter does its job by reading commands from the user (or from a file, in the case of batch systems) and then performing the commands. It performs the commands by directly making system calls itself, or by executing other programs to perform the requested task.

Note that in most operating systems, the command interpreter is itself just a regular user-level program, and not part of the kernel. (See, however, additional exercise B below) This modular design allows the command interpreter and the kernel to be updated and extended independently of one another.

2.4 `What is the purpose of ...`

System programs are provided by the operating system vendor to perform common tasks (such as backing up files, removing or renaming files, creating directories, formatting disks, setting the system date and time, etc.). These are tasks that are complex and specialized and that do not belong in the kernel, but without which the computer system would be useless to most users. OS vendors include them along with the kernel so that users don't have to write these programs themselves or purchase software to perform these tasks.

2.6 `List five services provided by ...`

Here are six. You may be able to think of others.

1. User interface - Allows the user interact with the computer without having to have knowledge of how to program the machine, or how to interact directly with the kernel.
2. Program execution - Hides the details of process layout, memory allocation, etc. from the user.
3. I/O operations - Prevents accidental damage to data structures on secondary storage devices. Hides the low-level details of I/O device operation from the programmer. Presents a consistent interface to devices which masks the individual differences between devices.
4. File-system manipulation - Presents the file system using a simple model (for example, the tree-structured directory model of Unix) while hiding the complexity of the actual storage structure.
5. Communications - As with I/O operations, hides the low-level details of communication from the programmer and provides a consistent interface over which to communicate.
6. Error detection - Provides for an automated way to deal with errors, and hides low-level errors from the user.

"Impossible" is a bit strong here, since theoretically, user-level programs could provide all of these services, provided the program could access the hardware directly. In fact, in early computers without sophisticated operating systems, user-level programs were required to do their own I/O operations, file system manipulation, and error detection. As you might imagine, mistakes were common, and it was easy to crash or even destroy an entire system, simply by issuing a bad command or executing a buggy program.

Modern OS's form a protective software layer around the hardware, so any user-level program running on the system must do so indirectly through the kernel if it wants to perform potentially hazardous operations.

The only exception here, perhaps, is the user interface. A program could, if it wanted to, provide its own user interface. In fact in the Unix family of operating systems (like Linux, Solaris and MacOS),

the GUI that the user sees is just another user process that is started when the user logs in.

2.7 `Why do some systems store ...`

Some systems may not have an actual hard disk. Others may only have a small amount of secondary storage. In both of these cases, the OS must be stored in firmware.

Disks are omitted if the system is small, inexpensive, or operates in situations where a disk failure would be catastrophic. Examples include calculators (which would be too inexpensive if equipped with a disk), smart watches (which are too small for a mechanical disk but may have a small amount of solid state secondary storage), portable music players (which are built as inexpensively as possible so may have minimal disk space), and embedded medical devices, like pacemakers (where the failure of a mechanical disk might cause the device to halt.)

2.8 `How could a system be ...`

Normally, the bootstrap program loads the operating system at boot time. One design in use is to make the bootstrap program load a separate program other than an OS kernel at boot time. This separate program is called a boot manager. The boot manager knows the names and locations of the different OSes installed on the computer, and can load them into memory and execute them as required. The boot manager is configurable, either at boot time or by a separate user-level program that can be run before a reboot. Linux uses a boot manager known as GrUB (Grand Unified Bootloader), so Linux users can boot into either Linux or Windows if they choose.

The bootstrap program itself resides in hardware and thus cannot be changed. Because the bootstrap program is fixed, it always loads a program from a fixed location in secondary storage. Either the OS (in the case of a machine that has as single OS installed) or the boot manager (in case of a machine that offers a choice of OSes) resides at this fixed location.

Additional Exercises

A `List at least five major activities of an OS with regard to file management.`

Here are nine:

1. Create files
2. Delete files
3. Open a file
4. Close a file
5. Read from a file
6. Write to a file
7. Set file attributes (such as owner, permission, creation time, etc.)
8. Read file attributes
9. Map the contents of a file on to one or more physical locations on a secondary storage device

In addition, if the operating system supports the concept of directories, we need to be able to perform each of these actions on directories as well.

B `Would it be possible for the user to develop a new command interpreter using the system-call interface provided by the operating system?`

Yes, it is possible to develop a new command interpreter using the system-call interface on those OS's where the interpreter is not tightly integrated into the system (i.e., most operating systems other than Windows.)

The Unix family of operating systems provides multiple command-line interpreters (sh, bash, csh, etc.) The 'sh' in each of these programs stands for and is pronounced as "shell", because the command-line interpreter is thought to serve as a protective shell around the system programs. Linux also provides multiple GUI command interpreters (Gnome, KDE, Xfc, etc.) The user is free to switch among the multiple command-line and GUI command interpreters.

One of the best ways to learn to program in a Unix environment is to write your own command line interpreter.

Bonus question: The command interpreter for the Windows family of operating systems is called Windows Explorer. (It is sometimes referred to as File Explorer or just Explorer.) In the *United States vs. Microsoft* anti-trust trial of 1998, Microsoft claimed that it was not possible to remove Explorer from the Windows 95™ operating system without completely breaking Windows.

Assuming that this is true[1], why do you think Microsoft made the Windows' user interface so tightly integrated with the operating system kernel?

C `What are the two models of interprocess communication (IPC)? What are the strengths and weaknesses of each model?`

The two models of interprocess communication (IPC) are shared memory and message passing.

IPC with shared memory is fast, since different processes (or a process and the kernel) can communicate simply by reading or writing the shared memory. However, some method of synchronization is required in order to avoid situations where two or more processes attempt to access their shared memory at the same time. In addition, distributed systems don't have any common primary memory, and thus can not use this method.

IPC with message passing requires less synchronization than shared memory, since there is no overlap between the physical memory of the processes. There is, however, more overhead required to setup message passing between processes, and to send and receive messages.

D `What is the main advantage of the micro-kernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?`

Because they are implemented using a small kernel and a collection of separate privileged system processes, microkernel OS's are easier to design and maintain then other types of kernels.

Rather than making system calls directly to the microkernel, user level programs make most service requests to system processes using

[1] It's not true.

IPC. These system processes fulfill the service requests by making system calls to the microkernel or requests to other system processes.

The main disadvantage of microkernels is that more communication overhead may be required to perform system services, resulting in an OS that may be slower than an equivalent layered or modular system. The OS community is still debating exactly how much slower a microkernel must be, and whether the design benefits of microkernels outweigh the costs associated with the slow-down.