CMPSCI 377 Operating Systems **Fall 2013**

Lecture 3: September 11

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

3.1 System Calls

A system call is programming interface for an application to request service from the operating system. Generally, operating systems provide a library (or high-level Application Program Interface or API) that sits between normal programs and the rest of the operating system, such as the POSIX library for managing processes in Linux. This library handles the low-level details of passing information to the kernel and switching to supervisor mode, as well as any data processing and preparation which does not need to be done in privileged mode. Ideally, this reduces the coupling between the operating system and the application, and increases portability. Many of today's operating systems have hundreds of system calls. For example, Linux has 319 different system calls. FreeBSD has about the same (almost 330). Writing applications that utilize libraries instead of system calls can simplify development and make it easier to deploy software to different operating systems.

Implementing system calls requires a control transfer which involves some sort of architecture specific feature. Typically, each system call is associated with a number. When a system call is made, it triggers a software interrupt or trap which uses the number to find the proper system call in a lookup table. Interrupts transfer control to the kernel so software simply needs to set up some register with the system call number they want and execute the software interrupt. The caller of the system call doesn't need to know anything about how the system call was implemented. The details of the OS implementation are generally hidden from the programmer (who simply need to follow the API).

3.1.1 Parameter Passing

Passing parameters to the kernel for a system call must be performed differently than when using an ordinary functional call. This is because a system call is performed by the kernel itself, which typically runs in a completely different address space than the process which made the call. Thus it is not possible to simply place system call parameters onto the process' stack as this will not be readily available to the kernel. There are three main methods to pass the parameters required for a system call: (1) Pass the parameters in registers (this may prove insufficient when there are more parameters than registers). (2) Store the parameters in a *block*, or table, in memory, and pass the address of block as a parameter in a register. This approach is used by Linux and Solaris. (3) Push the parameters onto a stack; to be popped off by the OS. Block and stack methods do not limitt the number or length of parameters passed.

3.2 OS organizations

The structure of operating systems has evolved as new systems have been developed. However, there is no one best OS organization, and different structures have their own benefits and drawbacks.

3.2.1 The Kernel

The *kernel* is the protected part of the OS that runs in kernel mode. This is typically used to protect the critical OS data structures from being read or modified by user programs. Depending on the OS organization being used, the kernel may be broken up into different components to either simplify OS development or to support different types of security. Thus different operating systems have different boundaries between the kernel and user space, depending on the functionality which is protected by the operating system. There is some debate as to what (and how much) functionality should go into the kernel.

3.2.2 Monolithic kernel

A monolithic kernel is a kernel architecture where the entire kernel is run as a single privileged entity running on the machine. Functionality in a monolithic kernel can be broken up into modules, however, the code integration for each module is very tight. Also, since all the modules run in the same address space, a bug in one module can bring down the whole system. However, when the implementation is complete and trustworthy, the tight internal integration of components allows the low-level features of the underlying system to be effectively utilized, making a good monolithic kernel highly efficient. In a monolithic kernel, all the systems such as the filesystem management run in an area called the kernel mode. The main problem with this organization is *maintainability.* Examples - Unix-like kernels (Unix, Linux, MS-DOS, Mac OS).

3.2.3 Layered architecture

Layered architecture organizes the kernel into a hierarchy of layers. Each layer provides a different type of functionality. Layers utilize the functionality of the layer below it and export new abstractions to the layers above it. Thus Layer $n+1$ uses services (exclusively) supported by layer n. This provides greater modularity compared to a monolithic kernel since a change to one layer only impacts those immediately above or below it. However, it is very restrictive to modularize the whole OS into such layers. This can make the OS design more difficult, and it requires additional copying and book-keeping since many calls may need to propagate through several layers.

While most modern operating systems do not use a layered architecture for the full design, the concept is used within several important OS subsystems. For example, the network functionality built in many OS kernels uses a layered architecture.

3.2.4 Microkernel

A microkernel is a minimal computer operating system kernel which, in its purest form, provides no operating-system services at all, only the mechanisms needed to implement such services, such as low-level address space management, thread management, and inter-process communication (IPC). The actual operating-system services are instead provided by “user-mode” servers. These include device drivers, protocol stacks, file systems and memory management code. A micokernel has the benefit of allowing for very customizable and modular operating systems. This greatly simplifies development of the OS since different modules are completely independent. However, not many systems use micokernels due to performance concerns caused by the overhead of frequent context switches between user and kernel mode. The amount of interprocess communication needed to coordinate actions between different services is also a performance concern.

3.2.5 Hybrid kernel

Hybrid kernel is a kernel architecture based on combining aspects of microkernel and monolithic kernel architectures used in computer operating systems. It packs more OS functionality into the kernel than a pure microkernel. This improves performance since less messaging between kernel and user space is required. Example - Mac OS X is based on the Mach microkernel.

3.2.6 Modular Approach

Modules are used by many modern operating system to divide a monolithic style kernel into more manageable components. Modules can be loaded dynamically to adjust the features provided by the OS. The modules together act and run like on big piece of code, but they are written independently. Modules are similar to layers, but modules can more easily communicate with one another. A modular system allows the OS to reduce its footprint and provide for cleaner development. In Linux the command *lsmod* lists the modules and they can be dynamically loaded and unloaded.

3.3 Processes

A process is an instance of a computer program that is being sequentially executed by a computer system which may be able to run multiple such processes concurrently. A computer program itself is just source code compiled into machine instructions, while a process is the actual *execution* of those instructions. There may be many processes running simultaneously for different programs. They can be anything from user programs and command scripts to system programs (such as print spoolers, network listeners, etc.). Alternatively, there may be several processes associated with the same program; for example, opening up several windows of the same program typically means more than one process is being executed.

In the computing world, processes are formally defined by the operating system running them and so may differ in detail from one OS to another. Specifically, a process has the execution context (program counter, registers, etc.) which is everything the process needs to run.

In general, a process state contains:

- The code for the program
- The program's static data
- Heap for dynamic data storage, and the heap pointer
- Program Counter(PC)
- Call stack
- CPU register values
- A set of OS resources in use, such as open files.
- Process execution state(Ready, Running, etc;) ...

The OS provides abstractions to allow the control of processes: starting, stopping, pausing, and resuming them. The OS must determine the order that processes are scheduled, and how individual processes are mapped to the hardware CPUs which will actually run them. A single computer processor executes one or more instructions at a time (per clock cycle), one after the other. To allow users to run several programs at once (e.g., so that processor time is not wasted waiting for input from a resource), single-processor computer systems can perform time-sharing. Time-sharing allows processes to switch between being executed and waiting (to continue) to be executed. In most cases this is done very rapidly, providing the illusion that several processes are executing 'at once'. (This is known as concurrency or multiprogramming.) The operating system keeps its processes separated and allocates the resources they need so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways. Shared memory is one example of how processes might interact with each other.