CMPSCI 377 Operating Systems **Fall 2012**

Lecture 3: September 11

Lecturer: Prashant Shenoy *TA: Sean Barker & Demetre Lavigne*

3.1 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. In general, a process is composed of a section of memory containing both the application code and its data (the stack and heap), the values of any registers in use, and potentially information about I/O devices being used by the process. Specifically, a process has the execution context (program counter, registers, etc.) which is everything the process needs to run.

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.

3.1.1 Synchronization Example

Alice and Bob open a shared bank account. Their initial balance is \$0. Each deposit \$100. We would expect that the balance now be \$200. Consider the following sequence of operations:
(1)Alice reads Balance (she reads \$0),
(2)Alice calculates new balance: Balance + 100 = \$100,
(3)Bob reads Balance (He also reads \$0, as Alice has not yet written the new Balance),
(4)Bob calculates new balance: Balance + 100 = \$100,
(5)Alice writes Balance (she writes \$100),
(6)Bob writes Balance (he also writes \$100).

This sequence leads to the final balance being \$100 (which is incorrect). If we consider Alice and Bob to

be processes running within an operating system, this can occur if the scheduling of the two processes is interleaved. This example clearly illustrates the importance of synchronization between Alice's and Bob's processes. Synchronization is a feature that the OS can provide to allow multiple processes to coordinate their activities to ensure that applications with multiple processes behave as they are intended by controlling how they are scheduled and are able to manipulate data. For example, the OS could enforce the following sequence which would produce correct results:
(1)Alice reads $0,
(2)Alice increments,
(3)Alice writes $100,
(4)Bob reads $100,
(5)Bob increments,
(6)Bob writes $200.

There may be other sequences of reads and writes that would produce correct results.

3.2 Memory and Secondary Storage Management

Main memory is the only one directly accessible to the CPU. The CPU continuously reads instructions stored there and executes them. Any data actively operated on is also stored there in uniform manner.

The OS is responsible for allocating/deallocating memory space for processes. When a process is created, the OS grants it a region of memory, and the process may request additional memory as it runs. The OS provides the illusion of infinite memory to each process with the use of virtual memory. The OS must maintain the mappings from virtual to physical memory (which are stored in a data structure called a *page table*). It also decides how much memory to allocate to each process, and when a process should be removed from memory.

If memory becomes a limited resource, then the OS's virtual memory system may move process data from main memory to secondary storage (disk). Data stored to disk differs from main memory in that it is not directly accessible by the CPU and is significantly slower to read or write from. The operating system is responsible for transparently moving portions of a process's memory state between main memory and secondary storage.

3.3 The File System

Secondary storage devices, such as hard disks, are not directly used for storage. A file system is generally used as a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems are mainly used for persistent storage devices. Unlike data stored in memory, data stored into a file system backed by a persistent storage device will remain accessible after the system is turned off and on. File systems use storage devices such as hard disks or CD-ROMs to persist the data.

There are many different file systems in use today such as NTFS in Windows, EXT3 in Linux, or ZFS in Solaris. All of these file systems provide the abstraction of logical entities (files) to store data, and a directory structure to control where files are located. However, each file system may save the data to disk in different ways by varying the disk block size or layout.

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sectors, generally a power of 2 in size (512 bytes or 1, 2, or 4 KB are most common). The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. File systems typically have directories which associate file names with files, usually by connecting the file name to an index in a file allocation table of some sort, such as the FAT in a DOS file system, or an inode in a Unix-like file system. File system provides a standard interface to create, delete and manipulate (read, write, extend, rename, copy, protect) files and directories. It also provides general services such as backups, maintaining mapping information, accounting, and quotas.

The OS is responsible for various low-level disk functions. It is necessary for the OS to schedule disk operations, manage the disk head movement, and provide error handling. Some file system functions can be provided by the OS rather than the file system itself. For example, managing the amount of free space might be the responsibility of the OS or the file system depending on the system configuration.

3.4 I/O Systems

The I/O system supports communication with external devices, like terminal, printer, keyboard, mouse etc. It also supports buffering and spooling of I/O. Spooling refers to a process of transferring data by placing it in a temporary working area where another program may access it for processing at a later point in time. The OS provides a common system for buffering and spooling which can be used by the specific device drivers created for individual I/O devices.

The I/O system provides a general interface, hiding the differences among devices. In Linux, all devices are presented mimicking a file system interface. This can simplify interaction with I/O devices because familiar, file based access methods such as open, read, and write can be used to communicate with the device.

3.5 Distributed Systems

Distributed systems deals with hardware and software systems containing more than one processing or storage element, concurrent processes, or multiple programs, running under a loosely or tightly controlled regime. In distributed computing a program is split up into parts that run simultaneously on multiple computers communicating over a network. Distributed programs often must deal with heterogeneous environments, network links of varying latencies, and unpredictable failures in the network or the computers. The OS can support a distributed file system on a distributed system. There are many different types of distributed computing systems and many challenges to overcome in successfully designing one. The main goal of a distributed computing system is to connect users and resources in a transparent, open, and scalable way. Ideally this arrangement is more fault tolerant and more powerful than using stand-alone computer systems.

Examples of distributed systems include networked file systems, many web applications such as Facebook or twitter, or an instant messaging system. Distributed systems can be built using low level socket communication or higher level abstractions such as remote procedure calls.

3.6 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).

3.6.1 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.

3.7 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.

3.7.1 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.

3.7.2 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).

3.7.3 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.

3.7.4 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.

3.7.5 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.

3.7.6 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.