CMPSCI 377 Operating Systems **Fall 2008**

Lecture 4: September 11

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

4.1 Processes

A process is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently. A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be 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(s)(OS) running them and so may differ in detail from one OS to another. A single computer processor executes one or more (multiple) 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 OS schedules and manages the processes. 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.

4.1.1 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 increments Balance: Balance += $100 (new Balance is $100), (3)Bob reads Balance (He also reads $100, as Alice has not yet written the new Balance), (4)Bob increments Balance: Balance += $100 (new Balance is $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). This example clearly illustrates the importance of synchronization between Alice's and Bob's processes. In particular, the following sequence (if enforced) would produce correct results: (1)Alice reads, (2)Alice increments, (3)Alice writes, (4)Bob reads, (5)Bob increments, (6)Bob writes. There may be other sequences of reads and writes that would produce correct results.

4.2 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. Secondary storage differs from primary storage in that it is not directly accessible by the CPU. The computer usually uses its input/output channels to access secondary storage and transfers desired data using intermediate

area in primary storage. Secondary storage does not lose the data when the device is powered downit is non-volatile. Per unit, it is typically also an order of magnitude less expensive than primary storage. Consequently, modern computer systems typically have an order of magnitude more secondary storage than primary storage and data is kept for a longer time there. OS is responsible for allocating/deallocating memory space for processes. It maintains the mappings from virtual to physical memory (which are stored in page tables). It also decides how much memory to allocate to each process, and when a process should be removed from memory.

4.3 File System

File system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files. 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.

4.4 I/O Systems

It supports communication with external devices, like terminal, printer, keyboard, mouse etc. It 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. This temporary working area could be a file or storage device, but probably not a buffer. The I/O system provides a general device driver interface, hiding the differences among devices, often mimicking the file system interface.

4.5 Distributed Systems

Distributed system deals with hardware and software systems containing more than one processing element 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 drastically more fault tolerant and more powerful than many combinations of stand-alone computer systems.

4.6 System Calls

A system call is the mechanism used by an application program to request service from the operating system. Generally, operating systems provide a library that sits between normal programs and the rest of the operating system, usually an implementation of the C library (libc), such as glibc. 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. On Unix-based and POSIX-based systems, popular system calls are open, read, write, close, wait, exec, fork, exit, and kill. 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). System calls are mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use. This is done to show up a clean interface to the user, hiding details of the OS interface for system calls. Implementing system calls requires a control transfer which involves some sort of architecture specific feature. A typical way to implement this is to use a software interrupt or trap. 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.

4.6.1 Parameter Passing

Three methods to pass the parameters: (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.

4.7 OS organizations

4.7.1 Monolithic kernel

A monolithic kernel is a kernel architecture where the entire kernel is run in kernel space in supervisor mode. In common with other architectures (microkernel, hybrid kernels), the kernel defines a high-level virtual interface over computer hardware, with a set of primitives or system calls to implement operating system services such as process management, concurrency, and memory management in one or more modules. Even if every module servicing these operations is separate from the whole, the code integration is very tight and difficult to do correctly, and, 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).

4.7.2 Layered architecture

Layered architecture organizes the kernel into a hierarchy of layers. Layer $n + 1$ uses services (exclusively) supported by layer n. Mostly it is very difficult to modularize the whole OS into such layers. Usually, parts of the OS are separately layered. This architecture is easier to extend and evolve.

4.7.3 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). If the microkernel has a kernelmode-usermode distinction, the microkernel is the only part of the system executing in a kernel mode. The actual operating-system services are provided by "user-mode" servers. These include device drivers, protocol stacks, file systems and user-interface code. Though it is easier to maintain a microkernel (as compared to a monolithic kernel), not many implementations of it exist. The reason is its inefficiency, owing to more context switches between the user and kernel mode.

4.7.4 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. Example - Mac OS X.