# The OS as a Society of Processes

Peter J. Denning

5/25/19

One of the most important jobs of a kernel is to manage the parallel execution of many processes and keep them from interfering with one another. The central idea is that the operating system is the orchestrator of a large number of processes. Some of the process are created when users give commands to the shell. Others are permanent, non-terminating service processes created by the operating system; these service processes are often called daemons, after a half-human, half-god, usually beneficent being extolled in Greek mythology.

This view of a system is a sharp contrast with a program-oriented view, which would see the system as a large library of programs to be executed on command by the user. That view was tried in very early operating systems, but led to hopeless complexity in managing the currency when multiple programs were started independently by different users.

## Kernel and Processes

The kernel is a set of OS routines that perform very sensitive functions such as switching the CPU state, allocating memory, responding to interrupt signals, controlling access to files and other digital objects, and managing virtual machines. Any kernel error can disrupt many users or destroy user information. Expert system programmers design and build kernel routines; they devote considerable effort to making sure their code is trustworthy, error-free, tamper-proof -- and fast.

The kernel itself contains a subset called the *microkernel*, which is a relatively small part that provides the most basic services including interrupts, CPU multiplexing, basic I/O, main memory management, and message services between parallel processes. The microkernel must be exceptionally trustworthy because is the only part of the OS that has unrestricted access to all the hardware and memory.

OS and hardware designers jointly developed a common means to restrict access to parts of the hardware and memory. They invented two execution modes for the CPU: a privileged *kernel mode* (access to everything) and non-privileged *user mode* (access only to non-sensitive areas of memory and non-sensitive instructions). The OS automatically places the CPU into the kernel mode when a user process calls a kernel routine, and restores user mode when that kernel routine returns control. In the user mode, some CPU instructions are off limits, for example those that switch the CPU state, change the boundaries of a memory region, or respond to signals from I/O devices.

The OS provides two interfaces for users. The more visible one is the *graphical user interface* (GUI), which provides windows, desktop, icons, mouse, drag-anddrop, and other familiar user functions. The less visible one is the *command-line interface*, or *shell*, which interacts directly with the kernel via one-line text commands. Most general-purpose users work via the GUI as it provides a consistent universal interface to any program. Most system programmers and network engineers work with the shell because it is very compact and fast for those who know the shell command set. Moreover, the tools and programs they use are tailored to this compact text-based interface. Although for general purpose users the GUI is far easier to use than the shell, it is no more powerful because every GUI action translates into a command line equivalent – for example, clicking on a file icon is ultimately invokes the same system call as typing a command `open(file)`.

When a user requests the OS to run a program, the OS creates a *process* for the program. A process autonomously executes a particular program within the resource limits granted by the OS. Each user can create many processes. The kernel multiplexes the CPU among the existing processes, thereby allowing them to proceed concurrently. Note that *concurrent* refers to what users see, an illusion programs running of simultaneously. Despite that illusion the hidden mechanisms of the kernel execute the processes of the programs serially in small time slices, one after another.

To manage all the behind the scenes tasks supporting the users programs the OS is dependent on many additional interacting processes owned by the OS. This overall organization has been characterized as a "society of cooperating processes". Most of the processes you see in the process control panel are *non-terminating*; they are ever ready to provide services on request to other processes. The remainder of the processes, particularly those implementing transactional user commands, are *terminating* processes – they quit after performing the specific tasks they were invoked to perform. The set of all currently executing processes constitutes the "society". The society as a whole is designed to operate continuously and is restarted (rebooted) only occasionally. The idea of a continuously-running society of processes is a sharp contrast to the idea of a library of programs, which run when called by a user and terminate when their jobs are done.

## Where Do Processes Come From?

The non-terminating service processes are created by the OS when it is booting up, from a list of service processes.

One of these system processes is `login`, which asks a user at a workstation for credentials. On validation of credentials, `login` creates a `shell` system process that interacts with the user via the keyboard (for input) and display (for output). The user types a command line to the shell, specifying a pipeline (series) of one or more processes that carry out the command. The OS kernel creates and activates those as children (subordinate) processes. The shell goes to sleep until all its children complete, and then presents the response to the user.

## What is a Process?

You can visualize a process as a program in execution in a private memory region. A non-terminating process is a giant loop that starts at a *homing position* waiting for a request to arrive. The arrival wakes up the process, which then performs the task specified by the parameters of the request. When done it returns to its homing position.

A terminating process has a start position (entry point) that is like the homing position except the process does not cycle back. Instead it performs the task specified by the parameters of the request and, when it is done, it executes an `exit` kernel call. The `exit` terminates the process and notifies its parent of its completion.

Within a process, the path followed by the CPU executing instructions is called a thread (or sometimes *execution stream*). Most operating systems allow processes to contain multiple threads. Multiple threads enable parallel execution of some subtasks, thereby speeding up process completion time. All of a process' threads share the single memory space of the process. System (and application) programmers have to face the challenge of ensuring that the threads cooperate and do not interfere with each other.

## What Does the Shell Expect of the Kernel?

From the sketch of how a shell creates processes to carry out tasks specified in command lines, we can see four main functions the shell expects of the kernel.

### *1. Process Creation, Management, and Deletion*

The OS creates a "virtual machine" to run an executable program. The virtual machine executing a program is called a process.

Level 8 (see the OS map) houses the functions for process management. The `create_process` command inserts the executable code into a standard process template and then attaches the resulting process to the ready list for execution. The `delete_process` command unlinks the template and erases it. The parameters of the create-process command include the standard input and output, which are pointers to info-objects (see below). The accompanying virtual machine notes lay out details of how the shell constructs the virtual machines and their inputs and outputs, corresponding to a shell command.

### *2. Info Objects and Directories*

Processes use *info objects* (information objects) for their inputs and outputs. Info objects are typically files, devices, and pipes. A file is a named entity containing a sequence of bits; it can be accessed for reading (copying bits out) or writing (copying bits in). A device is a hardware entity used either for input (e.g., keyboard, mouse) or output (e.g., display, printer) but not both. A pipe is a transmission channel that connects the output of one process to the input of another. We use the term *stream* for the bits in info objects: a file contains a stream, an input device generates a stream, an output device absorbs a stream, and a pipe communicates a stream.

Every process has a single *standard input port* `IN`, and *standard output port* `OUT`. Any info object can be connected to `IN` or `OUT`. If the process code says `READ(IN)` the OS feeds it the stream of its `IN` object. If the process code says `WRITE(OUT)`, the OS feeds a stream to its `OUT` object.

A *directory* is a table, each entry of which associates a symbolic name chosen by the user with an internal pointer to an info object. Directories permit users to use and organize familiar names for objects; the associated pointer allows the OS to find the object. Directories can point to other directories and be arranged into a hierarchy, often called *directory tree*. A directory points to info objects, but is not itself an info object and it cannot be linked to a process's `IN` or `OUT` ports.

## *3. Messages*

The OS provides a message system that allows processes to send messages to each other rapidly and efficiently. The message system is called *interprocess communication* (IPC). Messages are mostly used to request services from nonterminating service processes, and to receive back the responses. Messages can also be sent over the network to service processes on other computers, via an interface called *remote procedure call*. The network protocol stack is therefore part of the IPC system.

## *4. Memory Management*

Each process is assigned a private region of the main memory (usually RAM) for its exclusive use. The OS defines the region with a descriptor (base, length) and does not permit the CPU to access outside the currently designated region during execution. When a process is loaded into memory for execution, a *linker* program gathers all its program modules and library components into a single, executable file. That executable file is joined with a stack to form the *address space* of the process. Thus, the process' address space holds all of a process' instructions and data, including copies of OS library code the process requires to execute. The operating system constructs a *memory map* to associate addresses with physical storage locations in the computer's main memory.

Often there is insufficient main memory space to hold the entire address space. The operating system stores the whole space as a file in the secondary memory, and based on usage, loads subsets into the main memory. The memory map indicates which addresses are loaded and which are not. The operating system maintains consistency between the main memory set and the address space file by copying back any segments that were modified. Thus, the main memory contents can vary and the address space file is always the master copy.

Authorized users are allocated space on the secondary memory (commonly referred to as DISK or HD even though rotating disks are in decline) to hold their files permanently.  By default only the owner can read or write a file, but the owner can grant permission to others.

## *5. Concurrency Control*

One of the most common things processes do when they interact is synchronize by sending and receiving signals.  Synchronize means that the receiving process cannot proceed past a checkpoint until an expected signal is received from a sending process – thus guaranteeing that the receiver cannot get ahead of a critical checkpoint in the sender.  For example, a service process waits until a user process sends a signal requesting service, and then the requesting process waits until the service process returns a response.  The kernel provides tools for synchronization, enabling parallel processes to successfully navigate interprocess procedure calls, locking shared resources, making processes run in a prescribed order (*serializing*), and avoiding deadlocks.  The *semaphore* is the most used synchronization tool within the OS kernel.

Another thing the kernel does for processes is to multiplex the available CPUs among them.  *Multiplex* means to cycle the CPU among a set of processes, giving each an interval of execution called a *time slice*.  This gives the illusion that the processes are executing in parallel.  The dispatcher is the portion of the kernel that does this and typically uses some combination of *round-robin scheduling* (to determine the order of execution) and *context-switching* (to safely transition the CPU to the next process).

## *6. Protected entry*

The set of all the function-names appearing in the user interfaces of the OS levels is the kernel's API (user interface).  The kernel programs implementing the functions are made by highly skilled professional systems programmers who have gone to great lengths to assure their programs are reliable, correct, and trustworthy. Their claims of reliability, correctness, and trust are all based on the assumption that when any kernel routine is called, the CPU always starts executing instructions at its entry point (designated first instruction).  Operating systems include special mechanisms to assure that kernel calls always start executing at their designated entry points.

A common practice in systems programs is to begin with validity checks on the incoming parameter values.  If any parameter fails to pass the test, the program will immediately return an error code to its caller and will not attempt to execute the kernel function.   If a caller could bypass the validity checks by tricking the CPU to start executing instructions after the checks, the caller could provide invalid data and cause the kernel function to malfunction.  Operating systems prevent this from happening by forcing all kernel calls to start with their entry points.

Forced entry is usually done with a *kernel entry directory* that lists function names and their kernel entry points.  The directory is private to the OS.  A caller cannot call directly, but must ask the OS to make the call on its behalf.  For example, a process wanting to call open-file will tell the operating system to call open-file, and the OS will get the correct entry point from the directory.

## Resources

Attached slide sets:

*Virtual machines* – an overview of the process structures that implement commands – start with examples of simple commands and what we want them to do, then depict with VM diagrams – define create and delete kernel commands for VMs consistent with usage in the shell description

*Shell* – an overview of the way the shell recognizes commands

*Time-sharing* – an overview of how basic multiplexing works to allow a set of processes to share a CPU and make progress in parallel.