**Lecture Notes for CS347: Operating Systems**

*Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay*

# 1. Introduction to Operating Systems

## 1.1 What is an operating system?

- Simply put, an operating system (OS) is a middleware between the computer hardware and the software that a user wants to run on it. An OS serves the following goals:

    1. *Convenience.* Instead of every user software having to rewrite the code required to access the hardware, the common utilities are bundled together in an OS, with a simple interface exposed to software developers of user programs.
    2. *Good performance and efficient hardware resource utilization.* This common code to access the hardware can be optimized to efficiently utilize the hardware resources, and provide good performance to the user programs.
    3. *Isolation and protection.* By mediating access to all hardware resources, the OS tries to ensure that multiple programs from multiple users do not encroach upon each other.

- What is the hardware that the OS manages? There are three main components in a typical computer system.

    1. CPU. The CPU runs instructions corresponding to user or OS code. Every CPU architecture has an instruction set and associated registers. One of the important registers is the program counter (PC or IP or EIP, you will find several names for it), which stores the memory address of the next instruction to be executed in memory. Some registers store values of the operands of the instructions. Some store pointers to various parts of the program memory (base of stack, top of stack etc.). Registers provide fast access but are few in number, and can only be used to hold limited amount of data.
    2. Main memory. Main memory (RAM) is slower than registers, but can hold a lot of data. Most of the code and data of user programs and the OS is held in main memory. Data and instructions are "loaded" from the memory into CPU registers during execution, and "stored" back into memory after execution completes.

        Note: there are also several levels of caches between main memory and CPU registers, which cache instructions and data. In the hierarchy of CPU registers to cache to main memory to disk, as you go down the list, access delay increases, size increases, and cost decreases. The OS mostly doesn't concern itself with managing the CPU caches.
    3. I/O devices. Examples include network cards (which transfer data to and from the network), secondary storage disks (which acts as a permanent backup store for the main memory), keyboard, mouse, and several other devices that connect the CPU and memory to the external world. Some I/O devices like the network card and disk are block I/O devices (i.e., they transfer data in blocks), while some like the keyboard are character devices.

- There are several ways of building an OS. For example, a **monolithic** operating system is one where most of the code is built as a single executable (e.g., Linux, and other Unix-line systems). On the other hand, **microkernel** operating systems are built in a more modular fashion, with a very minimal core, and several services running on top of this microkernel. We will focus on monolithic operating systems like Linux and xv6 in this course.
- An OS consists of a **kernel** (the core of the OS, that holds its most important functionalities) and **system programs** (which are utilities to access the core kernel services). For example, the command `ls` is a system program / executable, whose job is to query the OS and list files. Some kernels (like Linux) also have dynamic pieces of code that can be loaded at runtime (called **kernel modules**), to add functionality to the kernel without having to reboot with a new executable.
- Linux and xv6 are mostly written in C, with the architecture-dependent code written in assembly. Note that the OS cannot make use of C libraries or other facilities available to user-space developers, and kernel development is a different beast altogether.

## 1.2 Processes and Interrupts

- There are two fundamental concepts when understanding what an OS does: **processes** and **interrupts**. A process is a basic unit of execution for an OS. The most fundamental job of an OS is to run processes on the CPU—it is (almost) always running some process (either a user process or its own system/kernel process or an *idle* process if it has nothing else to do!). An interrupt is an event that (true to its name) interrupts the running of a process. When an interrupt occurs, the OS saves the **context** of the current process (consisting of some information like the program counter and registers, so that it can resume this process where it left off), switches the program counter to point to an **interrupt handler**, and executes code that handles the interrupt. After servicing the interrupt, the OS goes back to executing processes again (either the same process that was running before, or some other process), by reloading the context of the process. So, a typical OS has an event-driven architecture.

- To run a process, the OS allocates physical memory for the process, and loads the address of the first instruction to execute in the CPU's program counter register, at which point the CPU starts running the process. The **memory image** of a process in RAM has several components, notably: the user code, user data, the **heap** (for dynamic memory allocation using `malloc` etc.), the **user stack**, linked libraries, and so on. The user stack of a process is used to store temporary state during function calls. The stack is composed of several stack frames, with each function invocation pushing a new stack frame onto the stack. The stack frame contains the parameters to the function, the return address of the function call, and a pointer to the previous stack frame, among other things. The stack frame of a function is popped once the function returns. The physical memory of a system has the memory images of all running processes, in addition to kernel code/data.

- When running a process, the program counter of the CPU points to some address in the memory image of the process, and the CPU generates requests to read code in the memory image of the process. The CPU also generates requests to read/write data in the memory image. These requests from CPU to read and write the memory of a process must be served by the memory hardware during the execution of the process. How does the CPU know the addresses of the memory image of a process in physical memory? The CPU does not need to know the actual physical addresses in the memory. Instead, every process has a *virtual* memory address space starting from 0 to the maximum value that depends on the architecture of the system, and the number of bits available to store memory addresses in CPU registers. For example, the virtual address space of a process ranges from 0 to 4GB on 32-bit machines. Virtual addresses are assigned to code and data in a program by the compiler in this range, and the CPU generates requests for code/data at these virtual addresses at runtime. These virtual addresses requested by the CPU are converted to actual *physical* addresses (where the program is stored) before the addresses are seen by the physical memory hardware. The OS, which knows where all processes are located in physical memory, helps in this address translation by maintaining the information needed for this translation. (The actual address translation is offloaded to a piece of specialized hardware called the Memory Management Unit or MMU.) This separation of addresses ensures that each process can logically number its address space starting from 0, while accesses to the actual main memory can use physical addresses.

## 1.3 What does an OS do?

- Most of the code of an OS deals with managing processes, and enabling them to do useful work for the user. Some of the things done by a modern operating system are:

    1. *Process management*. A good part of the OS code deals with the creation, execution, and termination of processes. A computer system typically runs several processes. Some processes correspond to executions of user programs—these are called user processes. Note that a process is an execution of a program. You can create multiple processes, all running the same program executable. Some processes, called system/kernel processes, exist to do some work for the operating system.
    2. *Process scheduling.* The OS has a special piece of code called the (CPU) scheduler, whose job is picking processes to run. For example, after servicing an interrupt, the OS invokes the scheduler code to decide if it should go back to running the process it was running before the interrupt occurred, or if it should run another process instead. Modern operating systems timeshare the CPU between several concurrent processes, giving each process the illusion that it has a CPU to itself.
    3. *Inter-process communication and synchronization.* The OS provides mechanisms for processes to communicate with each other (inter-process communication or IPC), share information between them, and for processes to synchronize with each other (e.g., for one process to do something only after another process has done a previous step).
    4. *Memory management.* An OS creates a process by allocating memory to it, creating its memory image, and loading the process executable into the memory image from, say, a secondary storage disk. When a process has to execute, the memory address of the instruction to run is loaded into the CPU's program counter, and the CPU starts executing the process. The OS also maintains mappings (called page tables) from the logical addresses in the memory image of a process to physical addresses where it stored the memory image of the process during process creation. These mappings are used by the memory hardware to translate logical addresses to physical addresses during the execution of a process.
    5. *I/O management.* The OS also provides mechanisms for processes to read and write data from external storage devices, I/O devices, network cards etc. The I/O subsystem of an OS is involved in communicating instructions from processes to devices, as well as processing interrupts raised by the devices. Most operating systems also implement networking protocols like TCP/IP that are required for reliably transferring data to/from processes on different machines.

## 1.4 User mode and kernel mode of a process

- Where does an OS reside? When does it run? And how can we invoke its services? For example, if a user program wants to take the help of the OS to open a file on disk, how does it go about doing it? The OS executable is not a separate entity in memory or a separate process. Instead, the OS code is mapped into the virtual address space of *every* process. That is, some virtual addresses in the address space of every process are made to point to the kernel code. So, to invoke the code of the OS that, for example, opens a file, a process just needs to jump (i.e., set the program counter) to the part of memory that has this kernel function.

- Does this mean that we have as many copies of the OS code in memory, one for each process? No. A virtual address of a piece of kernel code in any process points to the same physical address, ensuring that there is only one actual physical copy of the kernel code. Thus the separation of virtual and physical addresses also lets us avoid duplication of memory across processes. Processes share libraries and other large pieces of code also in this fashion, by keeping only one physical copy in memory, but by mapping the library into the virtual address space of every process that needs to use it.

- However, the kernel data being mapped to the address space of every process may lead to security issues. For example, what if a malicious user modifies the kernel datastructures in a dangerous manner? For this reason, access to the kernel portion of the memory of the process is restricted. At any point of time, a process is executing in one of two modes or privilege levels: **user mode** when executing user code, and **kernel mode** when executing kernel code. A special bit in the CPU indicates which mode the process is in, and illegal operations that compromise the integrity of the kernel are denied execution in user mode. For example, the CPU executes certain privileged instructions only in kernel mode. Further, the memory subsystem which allows access to physical memory will check that a process has correct privilege level to access memory corresponding to the kernel.

- Every user process has a user stack to save intermediate state and computation, e.g., when making function calls. Similarly, some part of the kernel address space of a process is reserved for the **kernel stack**. A kernel stack is separate from the user stack, for reasons of security. The kernel stack is part of the kernel data space, and is not part of the user-accessible memory image of a process. When a process is running code in kernel mode, all data it needs to save is pushed on to the kernel stack, instead of on to the regular user stack. That is, the kernel stack of a process stores the temporary state during execution of a process in kernel mode, from its creation to termination. The kernel stack and its contents are of utmost importance to understand the workings of any OS. While some operating systems have a per-process kernel stack, some have a common interrupt stack that is used by all processes in kernel mode.

- A process switches to kernel mode when one of the following happens.

    1. **Interrupts.** Interrupts are *raised* by peripheral devices when an external event occurs that requires the attention of the OS. For example, a packet has arrived on the network card, and the network card wants to hand the packet over to the OS, so that it can go back to receiving the next packet. Or, a user has typed a character on a keyboard, and the keyboard

hardware wishes to let the OS know about it. Every interrupt has an associated number that the hardware conveys to the OS, based on which the OS must execute an appropriate interrupt handler or interrupt service routine in kernel mode to handle the interrupt. A special kind of interrupt is called the timer interrupt, which is generated periodically, to let the OS perform periodic tasks.

2. **Traps.** Traps are like interrupts, except that they are not caused by external events, but by errors in the execution of the current process. For example, if the process tries to access memory that does not belong to it, then a segmentation fault occurs, which causes a trap. The process then shifts to kernel mode, and the OS takes appropriate action (terminate the process, dump core etc.) to perform damage control.
3. **System calls.** A user process can request execution of some kernel function using system calls. System calls are like function calls that invoke portions of the kernel code to run. For example, if a user process wants to open a file, it makes a system call to open a file. The system call is handled much like an interrupt, which is why a system call is also called a software interrupt.

- When one of the above events (interrupt, trap, system call) occur, the OS must stop whatever it was doing and service the interrupt / trap. When the interrupt is raised, the CPU first shifts to kernel mode, if it wasn't already in kernel mode (say, for servicing a previous trap). Then uses a special table called the **interrupt descriptor table (IDT)** to jump to a suitable interrupt handler in the kernel code that can service the interrupt. The CPU hardware and the kernel interrupt handler code will together **save context**, i.e., save CPU state like the program counter and other registers, before executing the interrupt handler. After the interrupt is handled, the saved context is restored, the CPU switches back to its previous state, and process execution resumes from where it left off. This context during interrupt handling is typically saved onto, and restored from, the kernel stack of a process.

- In a multicore system, interrupts can be delivered to one or more cores, depending on the system configuration. For example, some external devices may choose to deliver interrupts to a single specific core, while some may choose to spread interrupts across cores to load balance. Traps and system calls are of course handled on the core that they occur.

## 1.5 System Calls

- System calls are the main interface between user programs and the kernel, using which user programs can request OS services. Examples of system calls include system calls to create and manage processes, calls to open and manipulate files, system calls for allocating and deallocating memory, system calls to use IPC mechanisms, system calls to provide information like date and time, and so on. Modern operating systems have a few hundred system calls to enable user programs use kernel functionalities. In Linux, every system call has a unique number attached to it.

- Some system calls are **blocking**, while some are **non-blocking**. When a process makes a blocking system call, the process cannot proceed further immediately, as the result from the system call will take a little while to be ready. For example, when a process P1 reads data from a disk via a system call, the data takes a long period (few milliseconds, which is a long period for a CPU executing a process) to reach the process from disk. The process P1 is said to block at this point. The OS (i.e., the process P1 running in kernel mode for the system call) calls the scheduler, which switches to executing another process, say P2, in the meantime. When the data to unblock P1 arrives, say, via an interrupt, the process P2 running at that time marks the process P1 as unblocked as part of handling the interrupt. The original process P1 is then scheduled to run at a later time by the CPU. Not all system calls are blocking. For example, the system call to get the current time can return immediately.

- Note that user programs typically do not invoke system calls directly (though there is nothing wrong in doing so). They instead use functions provided by a library like the C library (e.g., libc), that eventually makes the system calls. Usually, every system call has a wrapper in these libraries. For example, when you use the `printf` function, this function's implementation in the C library eventually makes the `write` system call to write your data to the screen. A main advantage of not calling the system calls directly is that the user program does not need to know the intricacies of the system calls of the particular operating system, and can invoke more convenient library versions of underlying systems calls. The compilers and language libraries can then worry about invoking the suitable system calls with the correct arguments. Most operating systems try to have a standard API of system calls to user programs/compilers/libraries for portability. POSIX (Portable Operating Systems Interface) is a set of standards that specifies the system call API to user programs, along with basic commands and shell interfaces. An operating system is POSIX-compliant if it implements all the functions in the POSIX API. Most modern operating systems are POSIX compliant, which means that you can take a user program written on one OS and run it on another OS without needing to make any changes.

## 1.6 Context Switching

- A running process periodically calls the scheduler to check if it must continue execution, or if another process should execute instead. Note that this call to the scheduler to switch context can only happen by the OS, i.e., when the process is already in kernel mode. For example, when a timer interrupt occurs, the OS checks if the current process has been running for too long. The scheduler is also invoked when a process in kernel mode realizes that it cannot run any more, e.g., because it terminated, or made a blocking system call. Note that not all interrupts/traps necessarily lead to context switches, even it handling the interrupts unblocks some process. That is, a process need not context switch every time it handles an interrupt/trap in kernel mode.

- When the scheduler runs, it uses a policy to decide which process should run next. If the scheduler deems that the current process should not run any more (e.g., the process has just made a blocking system call, or the process has been running for too long), it performs a **context switch**. That is, the context of this process is saved, the scheduler finds a new process to run, the context of the new process is loaded, and execution of the new process begins where it left off. A context switch can be voluntary (e.g., the process made a blocking system call, or terminated) or involuntary (e.g., the process has run for too long).

- Note that saving context happens in two cases: when an interrupt or trap occurs, and during a context switch. The context saved in these two situations is somewhat similar, mostly consisting of CPU program counters, and other registers. And in both cases, the context is saved as structures on the kernel stack. However, there is one important difference between the two cases. When a process receives an interrupt and saves its (user or kernel) context, it reloads the same context after the interrupt handling, and resumes execution in user or kernel mode. However, during a context switch, the kernel context of one process is saved on one kernel stack, and the kernel context of another process is loaded from another kernel stack, to enable the CPU to run a new process.

- Note that context switching happens from the kernel mode of one process to the kernel mode of another: the scheduler never switches from the user mode of one process to the user mode of another. A process in user mode must first save the user context, shift to kernel mode, save the kernel context, and switch to the kernel context of another process.

## 1.7 Performance and Concurrency

- One of the goals of an OS is to enable user programs achieve good performance. Performance is usually measured by **throughput** and **latency**. For example, if the user application is a file server that is serving file download requests from clients, the throughput could be the number of files served in a unit time (e.g., average number of downloads/sec), and the latency could be the amount of time taken to complete a download (e.g., average download time across all clients). Obviously, high throughput and low latency are generally desirable.

- A user application is said to be saturated or running at capacity when its performance (e.g., as measured by throughput) is close to its achievable optimum value. Applications typically hit capacity when one or more hardware resources are fully utilized. The resource that is the limiting factor in the performance of an application is called the **bottleneck** resource. For example, for a file server that runs a disk-bound workload (e.g., reads files from a disk to serve to users), the bottleneck resource could be the hard disk. That is, when the file server is at capacity, the disk (and hence the application) cannot serve data any faster, and are running at as high a utilization as possible. On the other hand, for a file server with a CPU-bound workload (e.g., generates file content dynamically after performing computations), the bottleneck could be the CPU. For a file server that always has the files ready to serve in memory, the bottleneck could be its network link.

- For applications to get good performance, the OS must be well-designed to enable a process to efficiently utilize hardware resources, especially the bottleneck resource. However, a sometimes a process cannot fully utilize a resource, e.g., a process may block and not use the CPU when it is waiting for data from disk. Therefore, to achieve good utilization of hardware resources in the overall system, operating systems rely on **concurrency** or **multiprogramming**, i.e., running multiple processes simultaneously. For example, when a process makes a blocking system call, concurrent execution of several processes ensures that the CPU is fully utilized by other processes while the original process blocks.

- While concurrency is good for performance, having too many running processes can create pressure on the main memory. Further, if an application is composed of multiple processes, the processes will have to spend significant effort in communicating and coordinating with each other, because processes do not share any memory by default. To address these concerns, most operating systems provide light-weight processes called **threads**. Threads are also units of execution, much like processes. However, while every process has a different image, and does not share anything with other processes in general, all the threads of a process share a major portion of the memory image of the process. Multiple threads in a process can execute the program executable independently, so each thread has its own program counter, CPU register state, and so on. However, all threads in a process share the global program data and other state of the process, so that coordination becomes easier, and memory footprint is lower. Creating multiple threads in a process is one way to make the program do several different things concurrently, without incurring the overhead of having separate memory images for each execution.

- A note on **concurrency** and **parallelism**, and the subtle difference between the two. Concur-

rency is executing several processes/threads at the same time. Concurrent execution can happen on a single CPU itself (by timesharing the CPU among several processes), or in parallel across multiple CPUs (if the system has several processors or cores). Parallel executions of processes (e.g., on multiple cores) is not a requirement for concurrency, though the two concepts go together usually.

## 1.8 Booting

- Here is roughly what happens when a computer boots up. It first starts executing a program called the BIOS (Basic Input Output System). The BIOS resides in a non-volatile memory on the motherboard. The BIOS sets up the hardware, accesses the boot sector (the first 512-byte sector) of the boot disk (the hard disk or any other storage). The boot sector contains a program called the **boot loader**, whose job is to load the rest of the operating system. The boot loader is constrained to fit within the 512 byte boot sector. For older operating systems, the boot loader could directly load the OS. However, given the complexity of modern hardware, file systems and kernels, the boot loader now only loads another bigger boot loader (e.g., GRUB) from the hard disk. This bigger boot loader then locates the kernel executable, loads the kernel into memory, and starts executing instructions on the CPU. The boot loaders are typically written in architecture-dependent assembly language.

- The usual way to create new processes in Unix-like systems is to do a `fork`, which creates a child process as a copy of the parent process. A child process can then `exec` or execute another program binary (that is different from the parent's executable), by overwriting the memory image copied from the parent with another binary. When the OS boots up, it creates the first process called the `init` process. This init process then forks off many child processes, which in turn fork other processes, and so on. In some sense, the init process is the ancestor of all processes running on the computer. These forked off child processes run several executables to do useful work on the system.

- Note that a shell / terminal where you execute commands is also a process. This process reads user input from the keyboard and gets the user's command. It then forks a new process, and the child process executes the user command (say, an executable like `ls`). The main process then proceeds to ask the user for the next input. The GUI available in a modern OS is also implemented using system processes.