19CS2106S
Lecture Notes
Introduction to Operating Systems and Kernel
Session – 1

List of Topics	Learning Outcomes:	Questions answered in this Lecture notes:
1. Introduction to Operating Systems 2. Kernel Architecture	1. Understand operating systems 2. Understand and explore UNIX kernel Architecture	1. What is an OS and why do you want one? 2. Why study operating systems? 3. Explain various components of the UNIX v6 kernel 4. What is a kernel? 5. What is a shell? 6. What is a process? 7. What is a system call

References:

1. Maurice J. Bach, The Design of The Unix Operating System, 2013, Pearson. Chapter 2.1, Pages 19 - 22
2. Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci- Dusseau, ArpaciDusseau Books, Createspace Independent Publishing Platform (2018). http://pages.cs.wisc.edu/~remzi/OSTEP/
3. Michael Kerrisk, The Linux Programming Interface, 2010, No Starch Press.

1. What is an Operating System?

There is a body of software, that is responsible for making it easy to run programs (even allowing you to seemingly run many at the same time), allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software is called the operating system (OS), as it is in-charge of making sure the system operates correctly and efficiently in an easy-to-use manner.

The primary way the OS does this is through a general technique that we call virtualization. That is, the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.

Because virtualization allows many programs to run (thus sharing the CPU), and many programs to concurrently access their own instructions and data (thus sharing memory), and many programs to access devices (thus sharing disks and so forth), the OS is sometimes known as a resource manager. Each of the CPU, memory, and disk is a resource of the system; it is thus the operating system's role to manage those re-sources, doing so efficiently or fairly or indeed with many other possible goals in mind.

Operating System (OS): Software that converts hardware into a useful form for applications

Users
Applications
Operating System
Hardware

what an OS actually does? it takes physical resources, such as a CPU, memory, or disk, and virtualizes them. One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use.

2. What DOES OS Provide?

Role #1: Abstraction - Provide standard library for resources

What is a resource?

Anything valuable (e.g., CPU, memory, disk)

What abstraction does modern OS typically provide for each resource?

CPU:
- process and/or thread

Memory:
- address space

Disk:
- files

Advantages of OS providing abstraction?

1. Allow applications to **reuse** common facilities
2. Make different devices **look the same**
3. Provide higher-level or more **useful functionality**

Challenges

1. What are the correct abstractions?
2. How much of hardware should be exposed?

Role #2: Resource management – Share resources well

Advantages of OS providing resource management?

1. **Protect applications** from one another
2. Provide **efficient access to resources** (cost, time, energy)
3. Provide **fair access** to resources

Challenges

1. What are the correct mechanisms?
2. What are the correct policies?

The term **operating system** is commonly used with two different meanings:

1. To denote the entire package consisting of the central software **managing a computer's resources and all of the accompanying standard software tools**, such as command-line interpreters, graphical user interfaces, file utilities, and editors.
2. More narrowly, to refer to the central software that **manages and allocates** computer resources (i.e., the CPU, RAM, and devices).

3. The Kernel

- The **kernel** is the central module of an operating system (OS). It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the **essential services required by other parts of the operating system** and applications. The kernel code is usually loaded into a protected area of memory to prevent it from being **overwritten by programs** or other parts of the operating system.
- Typically, the kernel is responsible for **memory management, process and task management, and disk management.** The kernel connects the system hardware to the application software.
- Although it is **possible to run programs on a computer without a kernel**, the presence of a kernel greatly simplifies the writing and use of other programs and increases the power and flexibility available to programmers. The kernel does this by providing a software layer to manage the limited resources of a computer.
- The Linux kernel executable typically resides at the pathname **/boot/vmlinuz**, or something similar. The derivation of this filename is historical. On early UNIX implementations, the kernel was called **unix.** Later UNIX implementations, which implemented virtual memory, renamed the kernel as **vmunix**. On Linux, the filename mirrors the system name, with the z replacing the final x to signify that the kernel is a compressed executable.

4. Unix Family

- Unix and Unix-like operating systems are a family of computer operating systems that derive from the **original Unix System from Bell Labs** which can be traced back to 1965.
- Fortunately for UNIX, a young Finnish hacker named Linus Torvalds decided to write his own version of UNIX which borrowed heavily on the principles and ideas behind the original system, but not from the codebase, thus avoiding issues of legality. He enlisted help from many others around the world, took advantage of the sophisticated GNU tools that already existed, and soon Linux was born (as well as the modern open-source software movement).
- Every operating system has a kernel. For example, the **Linux kernel is used** numerous operating systems including Linux, FreeBSD, Android and others.
- As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux, as it was free and could be readily modified to suit their needs; indeed, it is hard to imagine the success of these new companies had such a system not existed. As smart phones became a dominant user-facing platform, Linux found a stronghold there too (via Android), for many of the same reasons. And Steve Jobs took his **UNIX-based NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops (though many users of Apple technology are probably not even aware of this fact). *Mac OS* X is based *on BSD* UNIX. Thus, UNIX lives on, more important today than ever before. The computing gods, if you believe in them, should be thanked for this wonderful outcome.
- Richard Stallman started the **GNU project** (a recursively defined acronym for "GNU's not UNIX") to develop an entire, freely available, UNIX-like system, consisting of a kernel and all associated software packages, and encouraged others to join him. In 1985, Stallman founded the **Free Software Foundation (FSF)**, a nonprofit organization to support the GNU project as well as the development of free software in general. Because a significant part of the program code that constitutes what is commonly known as the Linux system actually derives from the GNU project, Stallman prefers to use the term GNU/Linux to refer to the entire system. The question of naming (Linux versus GNU/Linux) is the source of some debate in the free software community. Since this course is primarily concerned with the API of the Linux kernel, we'll generally use the term Linux.
- **Linux** is the most prominent example of a "real" **Unix** OS. It runs on anything and supports way more hardware than BSD or OS X.

5. Tasks performed by the kernel:

Among other things, the kernel performs the following tasks:

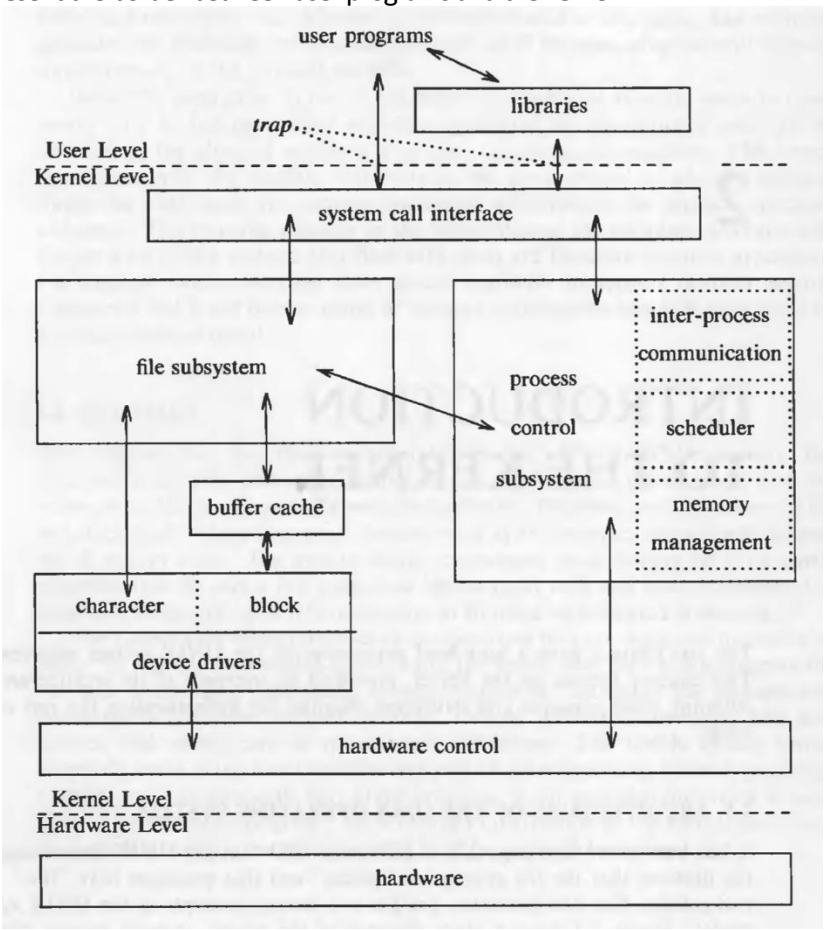
1. **Process scheduling:** A computer has one or more central processing units (CPUs), which **execute the instructions of programs**. Like other UNIX systems, Linux is a **preemptive multitasking operating system**, Multitasking means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the **kernel process scheduler (rather than by the processes themselves).**

2. **Memory management:** While computer memories are enormous by the standards of a decade or two ago, the size of software has also correspondingly grown, so that physical memory (RAM) remains a limited resource that the **kernel must share among processes in an equitable and efficient fashion**. Like most modern operating systems, Linux employs virtual memory management, a technique that confers two main advantages:
   a) Processes are **isolated from one another and from the kernel**, so that one process can't read or modify the memory of another process or the kernel.
   b) **Only part of a process needs to be kept in memory**, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.
3. **Provision of a file system:** The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.
4. **Access to devices:** The devices (mice, monitors, keyboards, disk and tape drives, and so on) attached to a computer allow communication of information between the computer and the outside world, permitting input, output, or both. **The kernel provides programs with an interface that standardizes and simplifies access to devices**, while at the same time arbitrating access by multiple processes to each device.
5. **Creation and termination of processes:** The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) that it needs in order to run. Such an **instance of a running program is termed a process**. Once a process has completed execution, the kernel ensures that the **resources it uses are freed for subsequent reuse by later programs**.
6. **System call application programming interface API:** Processes can request the kernel to perform various tasks using **kernel entry points known as system calls.**
7. **Networking:** The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes **routing of network packets to the target system.**

6. Unix system v6 Kernel:

The two entities, files and processes, are the two central concepts in the UNIX system model.

- The *file subsystem* is on the left and the *process control subsystem* is on the right.
- The diagram shows 3 levels: user, kernel, and hardware.
- The system call and library interface represent the border between user programs and the kernel.



1. **Libraries:** A library function is simply one of the multitude of functions that constitutes the standard C library. The purposes of these functions are very diverse, including such tasks as opening a file, converting a time to a human-readable format, and comparing two character strings. Many library functions don't make any use of system calls (e.g., the string manipulation functions). On the other hand, some library functions are layered on top of system calls. For example, the fopen() library function uses the open() system call to actually open a file. Often, library functions are designed to provide a more caller-friendly interface than the underlying system call. For example, the printf() function provides output formatting and data buffering, whereas the write() system call just outputs a block of bytes. Similarly, the malloc() and free() functions perform various bookkeeping tasks that make them a much easier way to allocate and free memory than the underlying brk() system call.
2. **System call interface:**
   - A system call is a controlled entry point into the kernel, **allowing a process to request that the kernel perform some action on the process's behalf**. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication. (The syscalls(2) manual page lists the Linux system calls.)

- System calls look like ordinary function calls in C programs, and **libraries map these function calls to the primitives** needed to enter the operating system.
- **Assembly language programs may invoke system calls directly without a system call library,** however. Programs frequently use other libraries such as the standard I/O library to provide a more sophisticated use of the system calls. The libraries are linked with the programs at compile time and are thus part of the user program for purposes of this discussion.

3. File subsystem

- The **file subsystem** manages files, allocating **file space**, administering free space, controlling access to files and retrieving data for users.
- The file subsystem accesses file data using a **buffering mechanism** that regulates data flow between the **kernel and secondary storage devices**. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. **Device drivers** are the kernel modules that control the operation of peripheral devices. **Block I/O** devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system.
- Raw devices, sometimes called character devices, include all devices that are not block devices.

4. Process control subsystem

- The **process control subsystem** is responsible for process synchronization, interprocess communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact **when loading a file into memory for execution**. the process subsystem reads executable files into memory before executing them.
- The **memory management** module controls the allocation of memory. If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute.
- The **scheduler** module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it is the highest priority eligible process available.

5. Interprocess communication

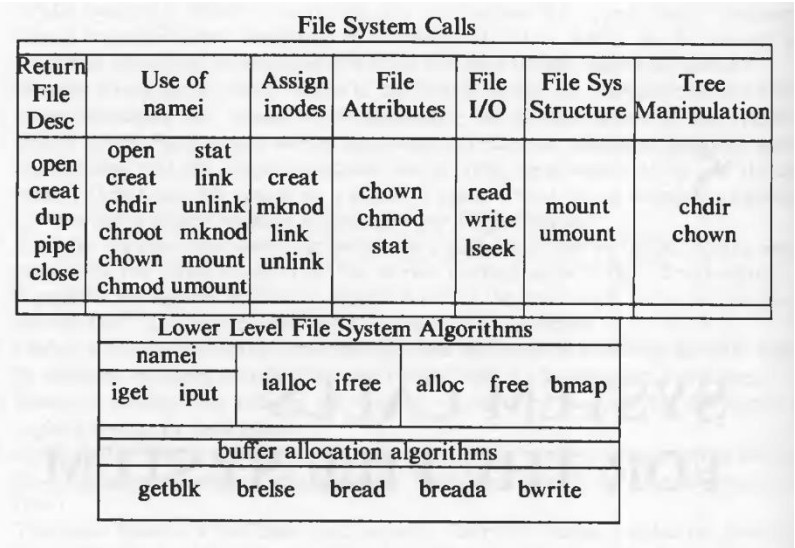
- A running Linux system consists of numerous processes, many of which operate **independently** of each other. Some processes, however, **cooperate to achieve their intended purposes**, and these processes need methods of communicating with one another and synchronizing their actions. One way for processes to communicate is by reading and writing information in disk files. However, for many applications, this is too slow and inflexible. There are several forms of interprocess communication, ranging from asynchronous signaling of events to synchronous transmission of messages between processes including the following: **signals, pipes, sockets, file locking, message queues, semaphores, shared memory**.

6. Hardware control

- Finally, the hardware control is responsible for **handling interrupts and for communicating with the machine**. Devices such as disks or terminals may interrupt the CPU while a process is executing. If so, the kernel may **resume execution** of the interrupted process after servicing the interrupt: Interrupts are *not* serviced by special processes but by **special functions in the kernel**, called in the context of the currently running process.

7. Architecture of the UNIX Operating System:

- **File subsystem micro view**



- **Process control sub system micro view**

System Calls Dealing with Memory Management			System Calls Dealing with Synchronization				Miscellaneous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

8. The Shell

A **shell** is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands. Such a program is sometimes known as a command interpreter.
The term login shell is used to denote the process that is created to run a shell when the user first logs in.
Whereas on some operating systems the command interpreter is an integral part of the kernel, on UNIX systems, the shell is a user process. Many different shells exist, and different users (or, for that matter, a single user) on the same computer can simultaneously use different shells. A number of important shells have appeared over time:

1. **Bourne shell (sh):** This is the oldest of the widely used shells, and was written by Steve Bourne. It was the standard shell for Seventh Edition UNIX. The Bourne shell contains many of the features familiar in all shells: I/O redirection, pipelines, filename generation (globbing), variables, manipulation of environment variables, command substitution, background command execution, and functions. All later UNIX implementations include the Bourne shell in addition to any other shells they might provide.
2. **C shell (csh):** This shell was written by Bill Joy at the University of California at Berkeley. The name derives from the resemblance of many of the flow-control constructs of this shell to those of the C programming language. The C shell provided several useful interactive features unavailable in the Bourne shell, including command history, command-line editing, job control, and aliases. The C shell was not backward compatible with the Bourne shell. Although the standard interactive shell on BSD was the C shell, shell scripts (described in a moment) were usually written for the Bourne shell, so as to be portable across all UNIX implementations.
3. **Korn shell (ksh):** This shell was written as the successor to the Bourne shell by David Korn at AT&T Bell Laboratories. While maintaining backward compatibility with the Bourne shell, it also incorporated interactive features similar to those provided by the C shell.
4. **Bourne again shell (bash):** This shell is the GNU project's reimplementation of the Bourne shell. It supplies interactive features similar to those available in the C and Korn shells. The principal authors of bash are Brian Fox and Chet Ramey. Bash is probably the most widely used shell on Linux. (On Linux, the Bourne shell, sh, is actually provided by bash emulating sh as closely as possible.)

The shells are designed not merely for interactive use, but also for the interpretation of **shell scripts**, which are text files containing shell commands. For this purpose, each of the shells has the facilities typically associated with programming languages: variables, loop and conditional statements, I/O commands, and functions.
Each of the shells performs similar tasks, albeit with variations in syntax. Unless referring to the operation of a specific shell, we typically refer to "the shell," with the understanding that all shells operate in the manner described. Most of the examples in this course that require a shell use bash, but, unless otherwise noted, the reader can assume these examples work the same way in other Bourne-type shells.