19CS2106S
Lecture Notes
Operating System Interfaces
Session – 3

SESSION NUMBER: 03 Session Outcome:

List of Topics	**Learning Outcomes:**	**Questions Answered from this Lecture notes:**
Operating system interfaces Operating system organization Abstracting physical resources User mode, kernel mode, and system calls Kernel organization	• Understand Operating system Interfaces and organization. • Understanding User mode, kernel mode, system calls and • Kernel organization	• Types of OS user Interface • Requirements of OS as Hardware Interface • Understand User mode, kernel mode and system calls.

Time	Topic	BTL	Teaching Learning Method	ALM
10	Operating system interfaces	2		
10	Operating system organization: Abstracting physical resources	2	Lecture & Discussion	
20	User mode, kernel mode, and system calls, Kernel organization	2	Lecture & Discussion	LTC
5	Conclusion & Summary			

References

1. Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing. CH 1 Page No [17-20]
2. Russ Cox, Frans Kaashoek, Robert Morris, xv6: a simple, Unix-like teaching operating system", Revision 11, CH 0 Page No [7-16] https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf

User Interface

A User interface (UI) facilitates communication between an application and its user by acting as an intermediary between them. Each application including the operating system is provided with a specific UI for effective communication. The two basic function of a user interface of an application is to take the inputs from the user and to provide the output to the users. However, the types of inputs taken by the UI and the types of output provided by the UI may vary from one application to another.
A user interface of any operating system can be classified into one of the following types:
Graphical user interface (GUI)
Command line user interface (CLI)

1) Graphical user interface (GUI)

The graphical user interface is a type of GUI that enables the users to interact with the operating system by means of point-and-click operations. GUI contains several icons representing pictorial representation of the variables such as a file, directory, and device. The graphical icon provided in the UI can be manipulated by the users using a suitable pointing device such as a mouse, trackball, touch screen and light pen. The other input devices like keyboard can also be used to manipulate these graphical icons. GUIs are considered to be very user- friendly interface because each object is represented with a corresponding icon. Unlike the other UIs the users need not provide text command for executing tasks.

Some advantages of GUI based operating system

- The GUI interface is easy to understand and even the new users can operate on them on their own.
- The GUI interface visually acknowledges and confirms each type of activities performed by the users. For example when the user deletes a file in the Windows operating system, then the operating system asks for the confirmation before deleting it.
- The GUI interface enables the users to perform a number of tasks at the same time. This features of the operating system are also known as multitasking.

2) Command line Interface (CLI)

Command line interface is a type of UI that enables the users to interact with the operating system by issuing some specific commands. In order to perform a task in this interface, the user needs to type a command at the command line. When the user enters the key, the command line interpreter received a command. The software program that is responsible for receiving and processing the commands issued by the user. After processing the command are called command line interpreter, the command line interpreter displays the command prompt again along with the output of the previous command issued by the user. The disadvantages of the CLI is that the user needs to remember a lot to interact with the operating system. Therefore these types of interface are not considered very friendly from the users perspective.
Example: In order to perform a task, we need to type a command at the command prompt denoted by C:\> to copy a text file, say al.text, from the C drive of our computer system. To the D drive, we need to type the copy command at the command prompt.

GUI vs CLI Comparison Table

Basis of comparison	GUI	CLI
Basic	This user interface enables the user to interact with electronic devices with the help of graphical icons and visual indicators.	This user interface enables a user to give a command to interact with an electronic device
Ease of understating	The graphical user interface is visually intuitive. It is easy to understand for beginners	Due to the need to remembering commands, it is difficult to handle and requires expertise.
Memory Requirement	It requires more memory as it consists of a lot of graphical components.	It requires less memory as compared to GUI.
Speed	It generally uses a mouse to execute commands. The speed of GUI is Slower than CLI.	Because the keyboard is used to execute the commands, the speed of the CLI is Faster than GUI
Appearance	One can change the appearance with a customizable option.	It is not possible to change the appearance
Flexibility	More flexible than CLI	Less flexible than GUI
Device used	Keyboard and mouse	Keyboard
Precision	Low as compared to the CLI	High as compared to the GUI

OS Hardware Interface

The job of an operating system is to share a computer among multiple programs and to provide a more useful set of services than the hardware alone supports. The operating system manages and abstracts the low-level hardware, so that, for example, a word processor need not concern itself with which type of disk hardware is being used. It also shares the hardware among multiple programs so that they run (or appear to run) at the same time. Finally, operating systems provide controlled ways for programs to interact, so that they can share data or work together
A key requirement for an operating system is to support several activities at once. For example, using the system call interface described in chapter 0 a process can start new processes with fork. The operating system must time-share the resources of the computer among these processes. For example, even if there are more processes than there are hardware processors, the operating system must ensure that all of the processes make progress. The operating system must also arrange for isolation between the processes. That is, if one process has a bug and fails, it shouldn't affect processes that don't depend on the failed process. Complete isolation, however, is too strong, since it should be possible for processes to interact; pipelines are an example. Thus an operating system must fulfil three requirements:

1. Multiplexing,
2. Isolation
3. Interaction.

Virtualization

The terms "logical" and "physical" appear often in descriptions of operating systems. These terms refer to different points of view regarding a system resource. The physical view refers to the actual physical state of affairs. The logical view is the view of the

application programmer. For resources other than the processor and memory, the logical view is reflected in the operating system application program interface (API) or application binary interface (ABI).
Virtualization refers to process involved in converting a physical view to a logical view. Virtualization serves two important purposes:

- It creates a simpler abstract view for application programmers.
- It minimizes resource management problems by allowing resources to be shared by several processes without conflict. In fact, one of the most common differences between the physical and logical views is that a physical resource may be shared by several processes, but logically, the processes are unaware of each other. Each process can have its own logical resource, even though there may only be a single physical resource.

Resources
The resources managed by an operating system are virtualized in different ways.
Processors: Physically, a processor can only execute one program at a time, but most operating systems support the appearance of executing several programs (or multiple instances of the same program) simultaneously. Each instance of program execution is called a process. Though several processes may be executing, their logical view is that they have the processor all to themselves.
Memory: When several processes are executing at the same time, the code and data that they are using must be kept in memory to avoid the overhead of transfers from another storage device such as a disk. Consider two processes executing the same program. For example, suppose you have two word processor windows open, each working on a different file. At the machine language level, a particular variable in the two processes appears to to have the same address in both processes. That is, you have two variables with the same logical address. But in fact, the two variables must be able to have different values so they must refer to two different physical addresses.
Displays : Physically, most computers have only one display device. However, most operating systems allow several processes to access the display simultaneously. Each uses a logical display called a window.
When a process does graphics on a display device, it must specify coordinates for points, line endpoints, etc. If two processes are using the display at the same time, the coordinates that they provide do not have to be physical screen coordinates. Instead, each process specifies coordinates relative to its own window. This allows the processes to execute without being aware of other processes that are using the display.
Mice and Keyboards : Physically, most computers have a single mouse and a single keyboard. Logically, each process has its own mouse and keyboard. A GUI application program just uses event handlers to respond to what appears to be its own mouse and keyboard.
Some processes, in response to mouse events, need to use the coordinates of the mouse. As with display coordinates, the mouse coordinates are relative to the window for the process.
Disk Storage: Physically, data on a disk is organized into blocks, typically 512 bytes each. For application programs, there are two logical views.
First, the disk data is organized into files and directories. The operating system API provides functions for manipulating these files and directories as units, disregarding their contents. For example, a file may be moved from one directory to another.
Second, programs frequently deal with files as sequential streams of data. That is, the file data is dealt with using the same functions that would be used for truly sequential streams of data, such as the input from a keyboard.
Printers : Physically, printers can only print one thing at a time. Logically, several processes can print data at the same time. The processes do not need to wait for printing from other processes to complete.
The elimination of the wait for output is part of the logical view of most output devices. Processes usually have no need to wait for an output operation to complete.
Networks : Many computer systems have a single physical network connection. However, several processes may need network connections simultaneously. For example, a user may be using one process and network connection to download new software, and another process and network connection to send mail. Each process has its own logical network connection.

Virtualization Mechanisms
There are two general mechanisms for giving the appearance of multiple instances of a resource where there is physically only one resource.

1. **Time-Shared Resources :** Processors, mice, keyboards, printers and network connections are time shared between processes. The important issue for time-shared resources is how the physical resource is switched from one process to another. This issue is dealt with in different ways, depending on the resource.
2. **Partitioned Resources :**Memory, displays, and disks are partitioned resources. These resources are partitioned into "pieces" that are assigned to individual processes. When a resource is partitioned, there is usually some concept of "location" of data handled by the resource. This necessitates a translation mechanism so that different processes do not have to be aware of the physical location of their logical resources. The important issue for partitioned resources is how this translation is done. This issue is dealt with in different ways, depending on the resource.

Abstracting physical resources
To achieve strong isolation for applications it’s helpful to forbid applications from directly accessing sensitive hardware resources, and instead to abstract the resources into services. For example, applications interact with a file system only through open, read, write, and close system calls, instead of read and writing raw disk sectors. This provides the application with the convenience of pathnames, and it allows the operating system (as the implementor of the interface) to manage the disk.

User mode, kernel mode, and system calls
Strong isolation requires a hard boundary between applications and the operating system. If the application makes a mistake, we don't want the operating system to fail or other applications to fail. Instead, the operating system should be able to clean up the failed application and continue running other applications. To achieve strong isolation, the operating system must arrange that applications cannot modify (or even read) the operating system's data structures and instructions and that applications cannot access other process's memory. Processors provide hardware support for strong isolation. For example, the x86 processor, like many other processors, has two modes in which the processor can execute instructions: kernel mode and user mode. In kernel mode the processor is allowed to execute privileged instructions. For example, reading and writing the disk (or any other I/O device) involves privileged instructions. If an application in user mode attempts to execute a privileged instruction, then the processor doesn't execute the instruction, but switches to kernel mode so that the software in kernel mode can clean up the application, because it did something it shouldn't be doing. Figure 0-1 in Chapter 0 illustrates this organization. An application can execute only user-mode instructions (e.g., adding numbers, etc.) and is said to be running in user space, while the software in kernel mode can also execute privileged instructions and is said to be running in kernel space. The software running in kernel space (or in kernel mode) is called the kernel. An application that wants to read or write a file on disk must transition to the kernel to do so, because the application itself can not execute I/O instructions. Processors provide a special instruction that switches the processor from user mode to kernel mode and enters the kernel at an entry point specified by the kernel. (The x86 processor provides the int instruction for this purpose.) Once the processor has switched to kernel mode, the kernel can then validate the arguments of the system call, decide whether the application is allowed to perform the requested operation, and then deny it or execute it. It is important that the kernel sets the entry point for transitions to kernel mode; if the application could decide the kernel entry point, a malicious application could enter the kernel at a point where the validation of arguments etc. is skipped.

Kernel organization
A key design question is what part of the operating system should run in kernel mode. One possibility is that the entire operating system resides in the kernel, so that the implementations of all system calls run in kernel mode. This organization is called a monolithic kernel.
In this organization the entire operating system runs with full hardware privilege. This organization is convenient because the OS designer doesn't have to decide which part of the operating system doesn't need full hardware privilege. Furthermore, it easy for different parts of the operating system to cooperate. For example, an operating system might have a buffer cache that can be shared both by the file system and the virtual memory system.
A downside of the monolithic organization is that the interfaces between different parts of the operating system are often complex (as we will see in the rest of this text), and therefore it is easy for an operating system developer to make a mistake. In a monolithic kernel, a mistake is fatal, because an error in kernel mode will often result in the kernel to fail. If the kernel fails, the computer stops working, and thus all applications fail too. The computer must reboot to start again. To reduce the risk of mistakes in the kernel, OS designers can minimize the amount of operating system code that runs in kernel mode, and execute the bulk of the operating system in user mode. This kernel organization is called a microkernel.

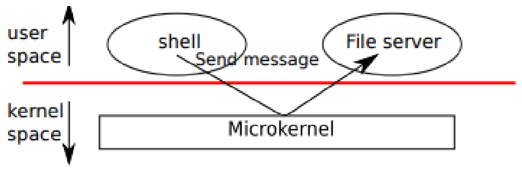

Figure 1-1. A microkernel with a file system server

<u>**XV6 Services**</u>
This section outlines xv6's services—
1. Processes and memory
2. File descriptors and pipes
3. File system

Shell (User Interface of XV6)
The shell, which is the primary user interface to traditional Unix-like systems, uses them. The shell's use of system calls illustrates how carefully they have been designed. The shell is an ordinary program that reads commands from the user and executes them. The fact that the shell is a user program, not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shell is easy to replace; as a result, modern Unix systems have a variety of shells to choose from, each with its own user interface and scripting features. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell.

Processes and memory
An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 can time-share processes: it transparently switches the available CPUs among the set of processes waiting to execute. When a process is not executing, xv6 saves its CPU registers, restoring them when it next runs the process. The kernel associates a process identifier, or pid, with each process.

A process may create a new process using the **fork system call**. Fork creates a new process, called the child process, with exactly the same memory contents as the calling process, called the parent process. Fork returns in both the parent and the child. In the parent, fork returns the child's pid; in the child, it returns zero.
For example, consider the following program fragment:

```
int pid = fork();
if(pid > 0)
  {
            printf("parent: child=%d\n", pid);
            pid = wait();
            printf("child %d is done\n", pid);
  }
else if(pid == 0)
  {
            printf("child: exiting\n");
            exit();
  }
else
  {
            printf("fork error\n");
   }
```

The **exit** system call causes the calling process to stop executing and to release resources such as memory and open files.
The **wait** system call returns the pid of an exited child of the current process; if none of the caller's children has exited, wait waits for one to do so.
In the example, the output lines

```
parent: child=1234
child: exiting
```

might come out in either order, depending on whether the parent or child gets to its printf call first.
After the child exits the parent's wait returns, causing the parent to print parent: child 1234 is done
Although the child has the same memory contents as the parent initially, the parent and child are executing with different memory and different registers: changing a variable in one does not affect the other. For example, when the return value of wait is stored into pid in the parent process, it doesn't change the variable pid in the child. The value of pid in the child will still be zero.

The **exec system call** replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. xv6 uses the ELF format,. When exec succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. Exec takes two arguments: the name of the file containing the executable and an array of string arguments.
Xv6 allocates most user-space memory implicitly: fork allocates the memory required for the child's copy of the parent's memory, and exec allocates enough memory to hold the executable file. A process that needs more memory at run-time (perhaps for malloc) can call sbrk(n) to grow its data memory by n bytes; sbrk returns the location of the new memory. Xv6 does not provide a notion of user

I/O and File descriptors

A file descriptor is a small integer representing a kernel-managed object that a process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. For simplicity we'll often refer to the object a file descriptor refers to as a ‘‘file’’; the file descriptor interface abstracts away the differences between files, pipes, and devices, making them all look like streams of bytes.
Internally, the xv6 kernel uses the file descriptor as an index into a per-process table, so that every process has a private space of file descriptors starting at zero. By convention, a process reads from file descriptor 0 (standard input), writes output to file descriptor 1 (standard output), and writes error messages to file descriptor 2 (standard error). As we will see, the shell exploits the convention to implement I/O redirection and pipelines. The shell ensures that it always has three file descriptors open (8707), which are by default file descriptors for the console.
The **read and write system calls** read bytes from and write bytes to open files named by file descriptors.
The call **read(fd, buf, n)** reads at most n bytes from the file descriptor fd, copies them into buf, and returns the number of bytes read. Each file descriptor that refers to a file has an offset associated with it. Read reads data from the current file offset and then advances that offset by the number of bytes read: a subsequent read will return the bytes following the ones returned by the first read. When there are no more bytes to read, read returns zero to signal the end of the file.
The call **write(fd, buf, n)** writes n bytes from buf to the file descriptor fd and returns the number of bytes written. Fewer than n bytes are written only when an error occurs. Like read, write writes data at the current file offset and then advances that offset by the number of bytes written: each write picks up where the previous one left off.
The **close** system call releases a file descriptor, making it free for reuse by a future open, pipe, or dup system call (see below). A newly allocated file descriptor is always the lowest-numbered unused descriptor of the current process.
File descriptors are a powerful abstraction, because they hide the details of what they are connected to: a process writing to file descriptor 1 may be writing to a file, to a device like the console, or to a pipe.

Pipes

A pipe is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing. Writing data to one end of the pipe makes that data available for reading from the other end of the pipe. Pipes provide a way for processes to communicate.

If no data is available, a read on a pipe waits for either data to be written or all file descriptors referring to the write end to be closed; in the latter case, read will return 0, just as if the end of a data file had been reached. The fact that read blocks until it is impossible for new data to arrive is one reason that it's important for the child to close the write end of the pipe before executing wc above: if one of wc's file descriptors referred to the write end of the pipe, wc would never see end-of-file.

File system

The xv6 file system provides data files, which are uninterpreted byte arrays, and directories, which contain named references to data files and other directories. The directories form a tree, starting at a special directory called the root. A path like /a/b/c refers to the file or directory named c inside the directory named b inside the directory named a in the root directory /. Paths that don't begin with / are evaluated relative to the calling process's current directory, which can be changed with the chdir system call.

There are multiple system calls to create a new file or directory: **mkdir** creates a new directory, open with the O_CREATE flag creates a new data file, and **mknod** creates a new device file.

mknod creates a file in the file system, but the file has no contents. Instead, the file's metadata marks it as a device file and records the major and minor device numbers (the two arguments to mknod), which uniquely identify a kernel device. When a process later opens the file, the kernel diverts read and write system calls to the kernel device implementation instead of passing them to the file system.

The **unlink** system call removes a name from the file system. The file's inode and the disk space holding its content are only freed when the file's link count is zero and no file descriptors refer to it.

Shell commands for file system operations are implemented as user-level programs such as mkdir, ln, rm, etc. This design allows anyone to extend the shell with new user commands by just adding a new user-level program. In hindsight this plan seems obvious, but other systems designed at the time of Unix often built such commands into the shell (and built the shell into the kernel).