FSO - Aula 3

Nicolas Chagas Souza

07/11/2022

Operating system services

- OS provide an environment for execution of programs and services to programs and users
- The specif services vary among different OS, but we can identify common classes

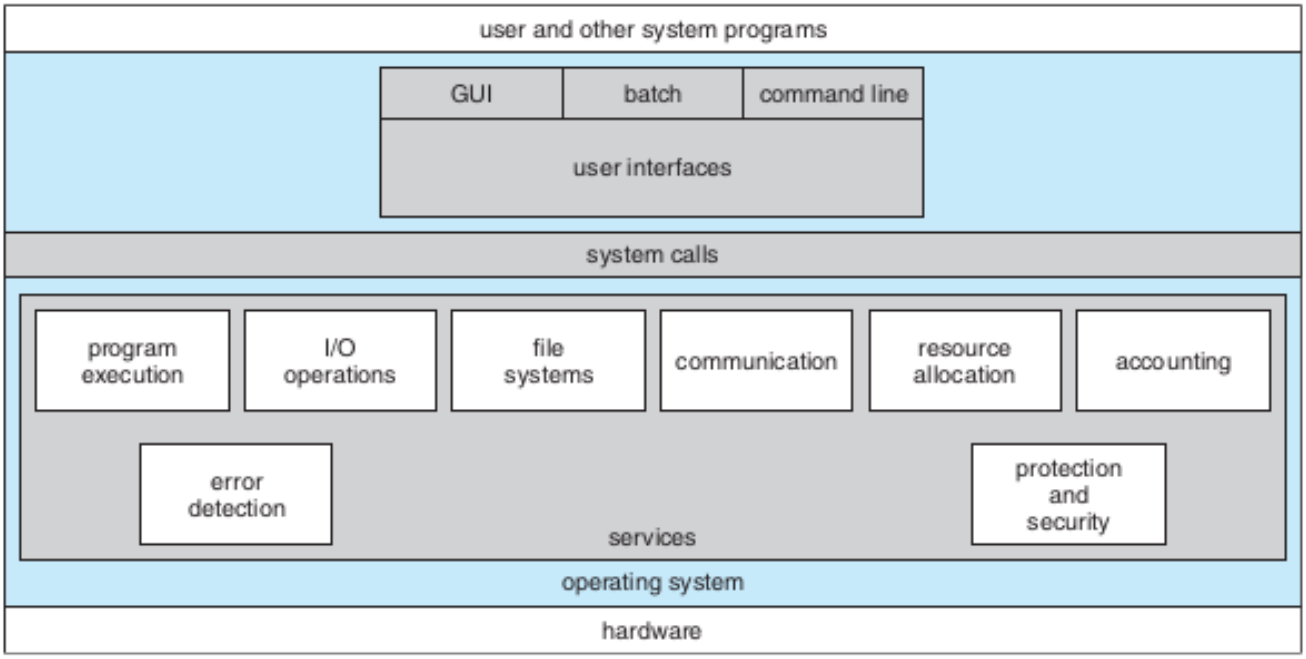


Figure 2.1 A view of operating system services.

Set of operating-system services that are helpful do the user:

- User interface: Almost all OS have a user interface (UI)
  - Varies between CLI, GUI and Batch
- Program execution: The system must be able to load a program into memory and to run that program, and execution, either normally or abnormally (indicating error)
- I/O operations: Users usually cannot control I/O devices directly, therefore, the OS must provide a means to do I/O.
- File-system manipulation: Programs need to read and write files and directories, some OS also include permissions management to allow or deny access to files or directories based on file ownership
- Communications: Communication may occur between processes that are executing on the same computer or between processes that are execution on different computer systems tied together by a computer network. The communication may be implemented via **shared memory** or **message passing**
- Error detection: the OS needs to be detecting and correcting erros constantly. Erros may occur in the CPU and memory hardware, in I/O devices and the user program

Some OS functions exists not for helping the user, but rather for ensuring the efficient operation of the system itself:

- Resource allocation: the OS manages many different types of resources, such as CPU cycles, main memory, I/O devices and file storage
- Accounting: we want to keep track of which users use how much and what kinds of computer resources
- Protection and security: the owners of information stored in a multiuser or networked computer system may want ot control use of that information. When several separate processes execute concurrently, it shouldn't be possible for one process to interfere with the others or the OS itself.
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders require user authentication, extends to defending external I/O devices from invalid access attempts

User and Operating-System Interface

Command Interpreters (CLI)

- Allows direct command entry, fetches a command from user and executes it.
- Some OS include the command interpreter in the kernel, others, like Windows and UNIX, treats the command interpreter as a special program that's running when a job's initiated or when a user first logs on.
- On system with multiple command interpreters to choose from, the interpreters are known as **shells**.

Graphical User Interfaces

- User friendly desktop metaphor interface
  - Usually mouse, keyboard and monitor
  - Icons represent files, programs, actions, etc
  - Invented at Xerox PARC
- Most systems now include both CLI and GUI interfaces
  - MS Windows is GUI with the CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

System Calls

- Provide an interface to the services made available by an operating system
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level API rather than direct system call use
  - The most common APIs are Win32 API for Windows, POSIX API for POSIX-based system (all versions of UNIX, Linux and Mac OS X) and Java API for the JVM.

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

- Typically, a number is associated with each system call, and the **system-call interface** maintains a table indexed according to these numbers.
- The system call interface invokes the intended system call in OS kernel and return status of the system call and any return values.
- The caller need to know nothing about how the system call is implemented, just need to obey the API and understand what OS will do as a result call. Most details of OS interface is hidden from programmer by API.

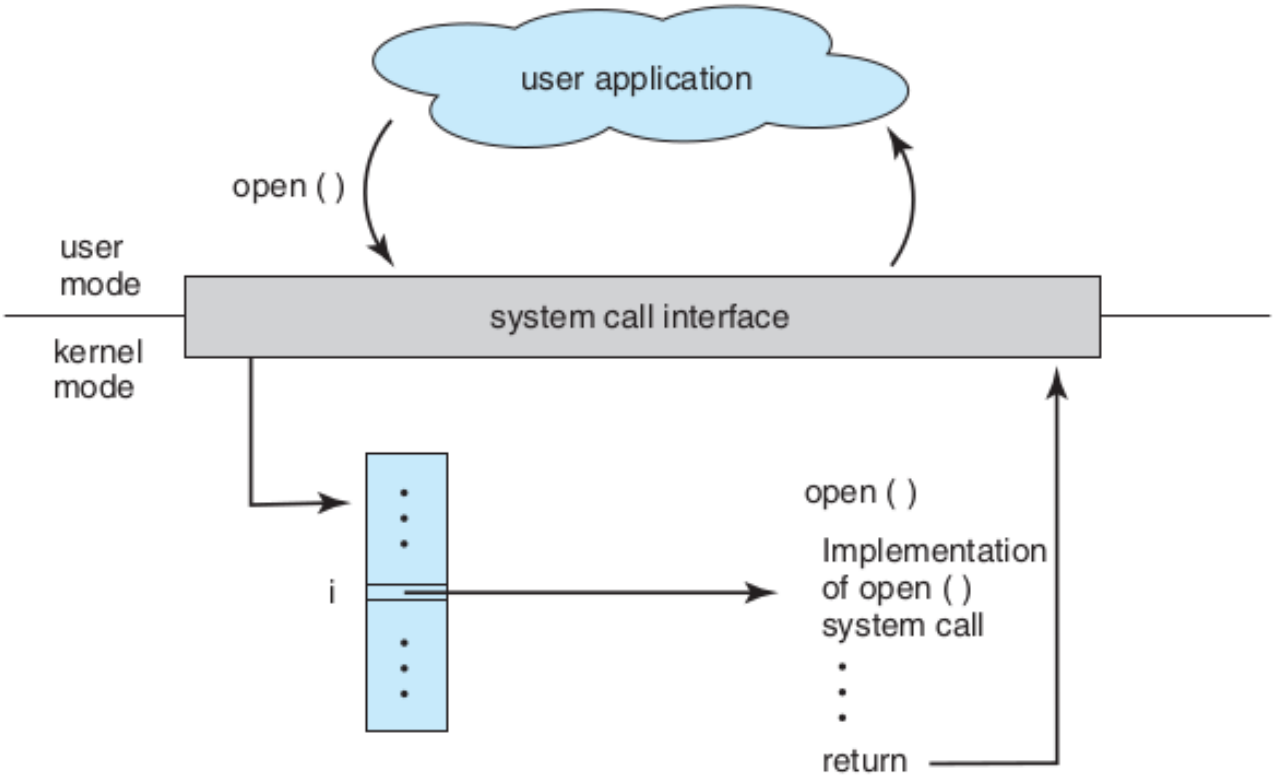


Figure 2.6 The handling of a user application invoking the `open()` system call.

System Call Parameter Passing

There are three general methods used to pass parameters to the OS:

- Simplest: passing the parameters in registers.
- Parameters stored in a block or table in memory, and address of block passed as a parameter in a register (approach taken by Linux and Solaris)
- Parameters **pushed** onto the **stack** by the program and **popped** off the stack by the OS.

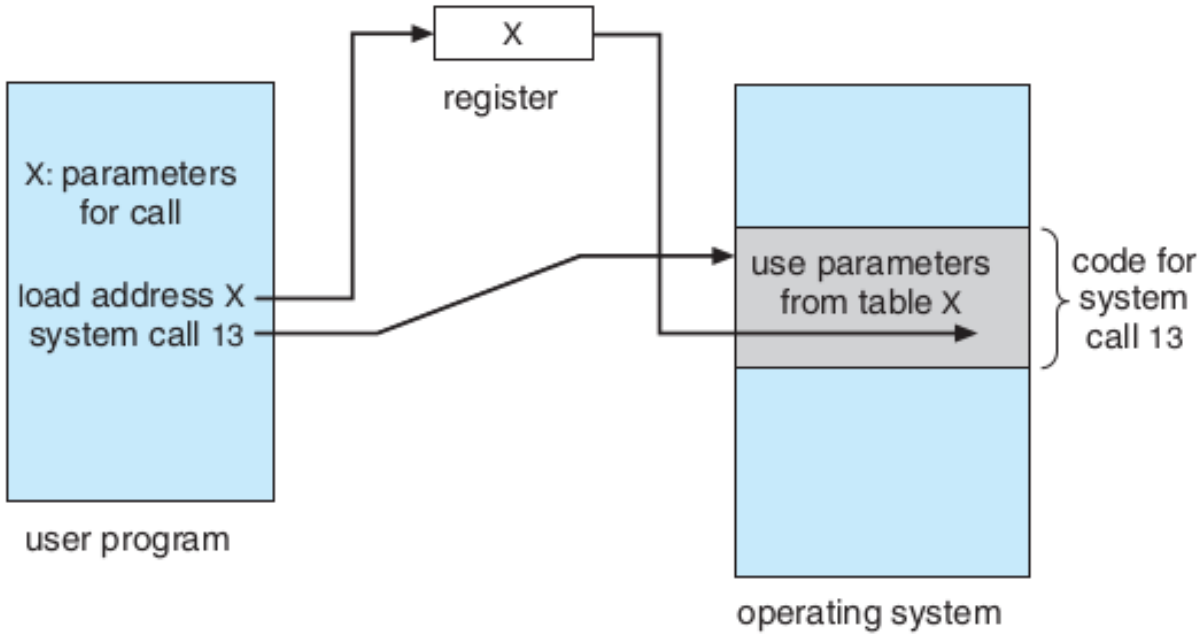


Figure 2.7 Passing of parameters as a table.

Types of System Calls

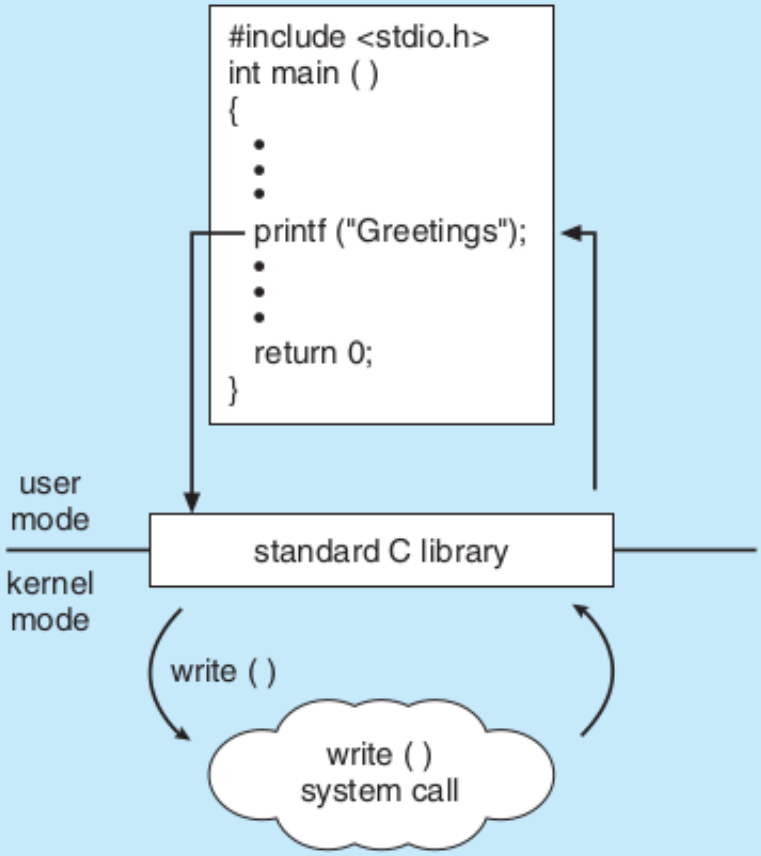
- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	write()
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communication	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

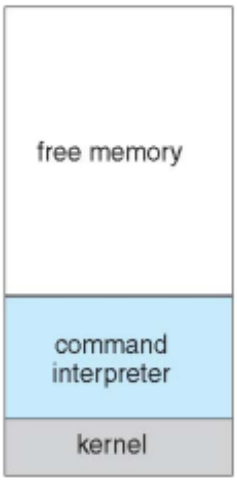
EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program. This is shown below:



Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded



(a)

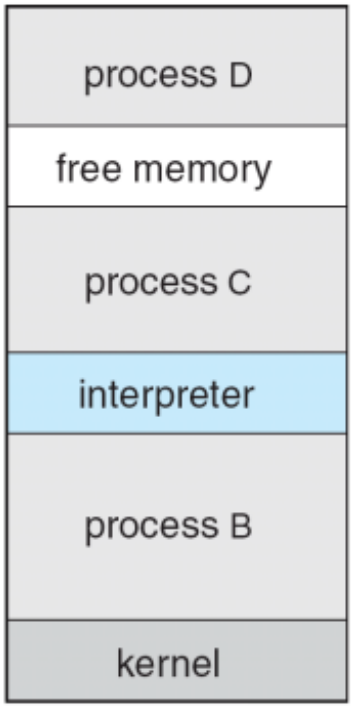
At system startup

(b)

running a program

Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



System Programs

- System programs provide a convenient environment for program development and execution, they can be divided into:
  - File manipulation
  - Status information sometimes stored in a File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Background services
  - Application programs