Chapter 2: Operating-System Structures

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 2)

Operating System Services – for user

- Operating-system provides many services to the user such as:
  - User interface to interact with OS - e.g., CLI, GUI, touch screen interface.
  - Program execution
  - I/O operations
  - File-system manipulation
  - Communications
  - Error detection

Operating System Services – for system

- Resource allocation
- Accounting
- Protection and Security

User Interface

Command Line Interpreter (CLI) allows direct command entry to OS.

- Primary job is to fetch a command from user and execute it.
- Windows and Unix CLI is not part of the kernel.
- In Unix/Linux they are called **shells** (e.g., bash shell in Linux).

- Sometimes the CLI itself contains the code to execute the command.
- Sometimes just names of programs (used by Unix)
  - Adding new features doesn't require shell modification!

Shells in Unix and Linux

- Multiple shells are available (you can write your own shell program!)
- *Shells do not have the code to execute the command*.
  - Eg: `rm file.text` in terminal → Invokes Shell → Shell searches for file `rm` → load `rm` in memory → executes it with `file.txt` as parameter
- Shell has no idea how '`rm`' command is implemented and the system call used to process the request.

Operating System Structure

- General-purpose OS is a very large program
- Various ways to structure an OS
  - Simple structure (Monolithic) – e.g., MS-DOS
  - Non-simple structure (Monolithic) – e.g., Original UNIX OS
  - Layered – an abstraction
  - Microkernel –Mach
  - Modules

Simple Monolithic Structure

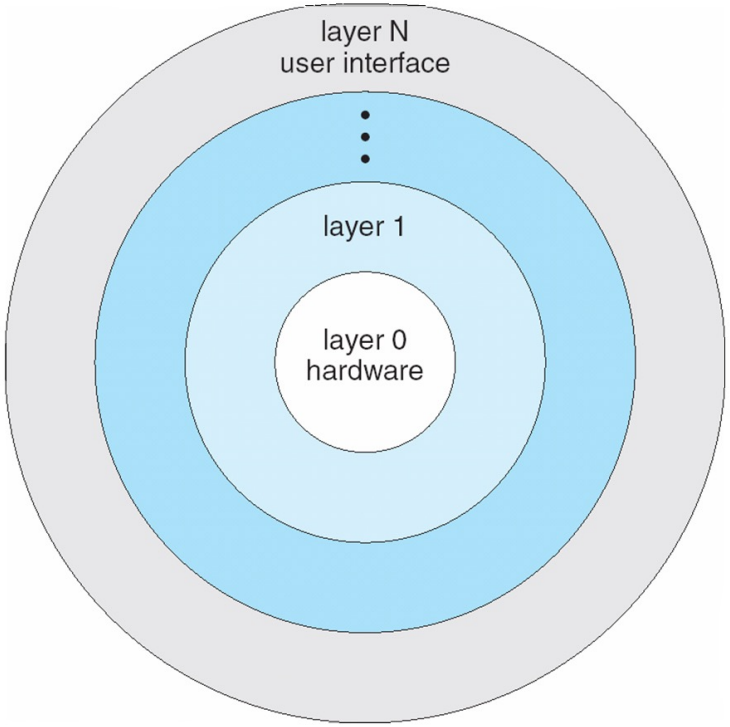
- Simplest monolithic structure has little to no structure at all.
- All of the functionality of the kernel (process, memory, file, I/O) is placed into a single, static binary file that runs in a single address space.
- **MS-DOS** is an example of the simplest monolithic structure
  - It is written to provide the most functionality in the least space.
  - Its interfaces and levels of functionality are not well separated.
    - Application programs can access I/O devices!

Non-simple Monolithic Structure -- Original UNIX OS

- The original UNIX operating system had limited structuring.
- Like MS-DOS it was limited by hardware functionality.
- The original UNIX OS consisted of two separable parts:
  - Systems programs
  - The kernel
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions
    - A large number of functions for one level

Layered Approach

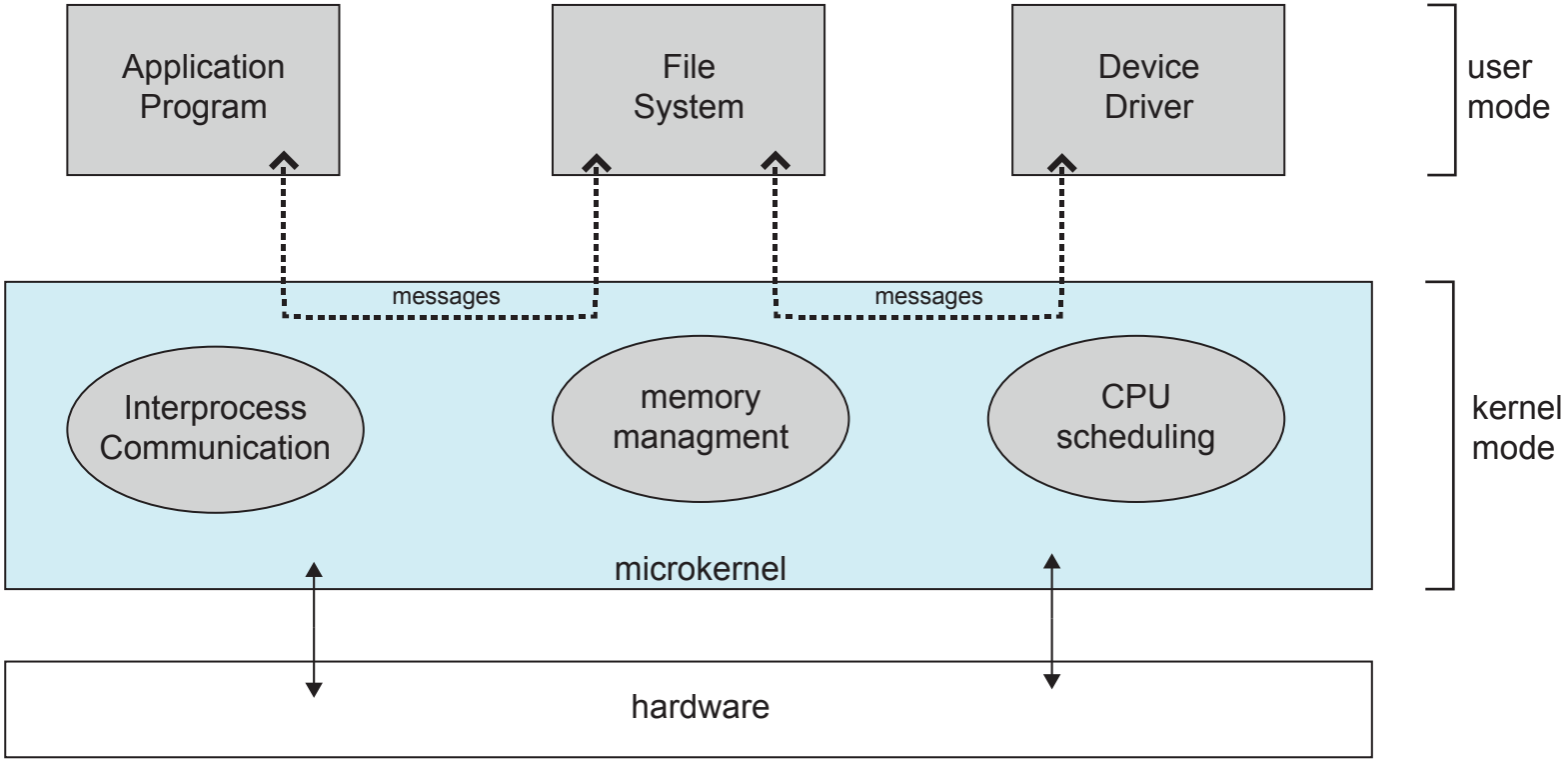
- OS divided into a number of layers (levels),
- Each built on top of lower layers.
  - The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- Layers are selected such that each layer uses functions (operations) and services of only *lower-level layers*
- Disadvantages:
  - Tricky to delineate the layers.
  - Slow - any user request needs to go through all the layers with correct function calls and parameters.



Microkernel System Structure

- Structures the Operating System by
  - Removing all the nonessential components from the kernel, and
  - Implementing them as user or system-level programs.
- Microkernels provide *minimal process and memory management*.
- Communication between the modules takes place using *message passing* (sharing data by passing messages).
- Examples:
  - Mach is an example of microkernel
  - Mac OS X (open source) kernel Darwin partly based on Mach
  - Windows NT, first release had a layered microkernel approach.)

Microkernel System Structure



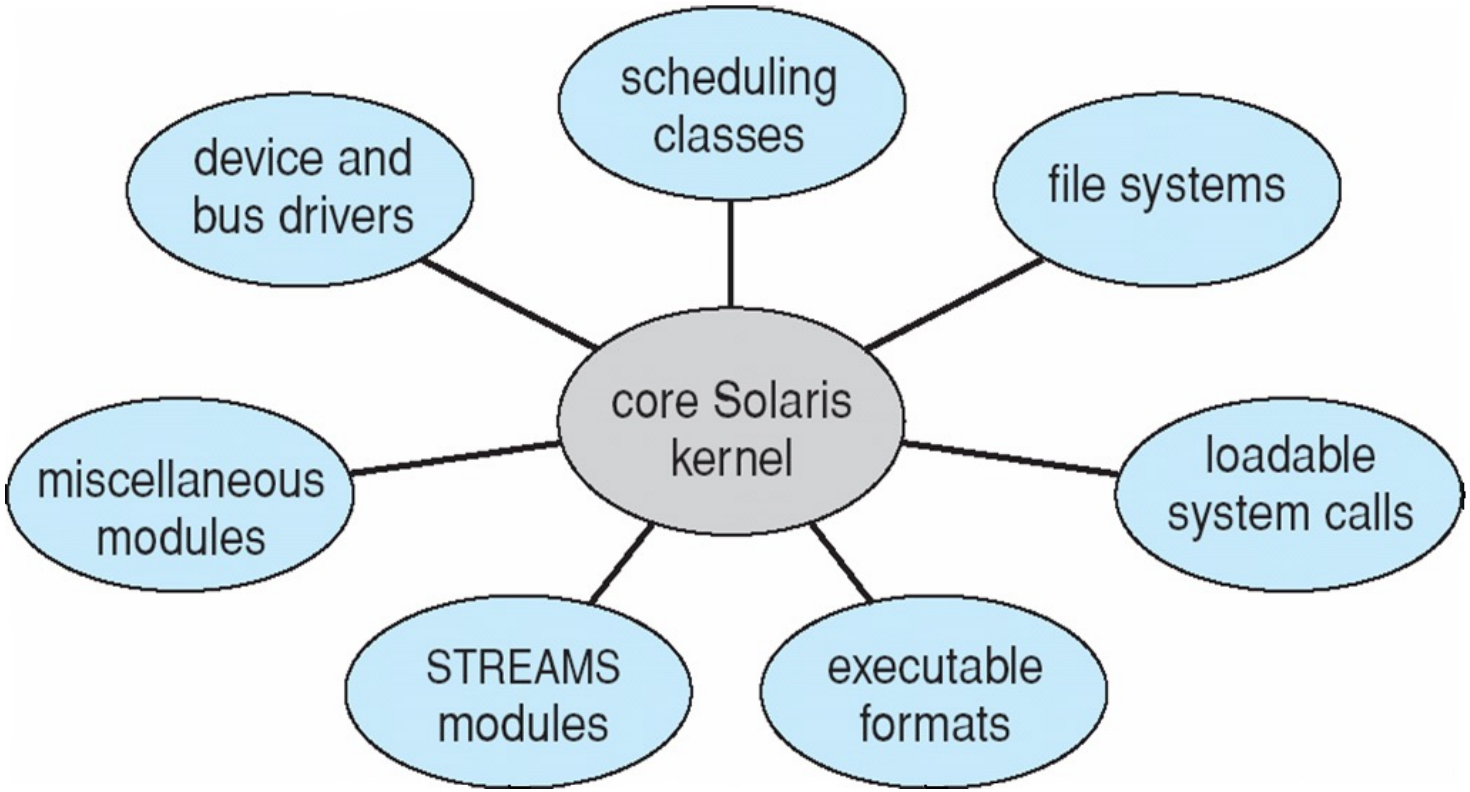
Microkernel – contd…

- Advantages:
  - Easier to extend a microkernel
  - Easier to port to new architectures
  - More reliable and secure
- Disadvantages:
  - Performance overhead of user space to kernel space communication

Modular Structure using Modules

- Kernel has a set of separate core components (with clearly defined interfaces).
- Additional services are linked in via modules (either at boot time or run time.)
- Each module is loadable as needed within the kernel
- Many modern operating systems (UNIX, Linux, Solaris, etc., and Windows) implement **loadable kernel modules** - so far, the best methodology for OS design.

Solaris Modular Approach



Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels are:
    - Monolithic for efficient performance.
    - Modular - dynamic loading of functionality

System Calls and System Programs

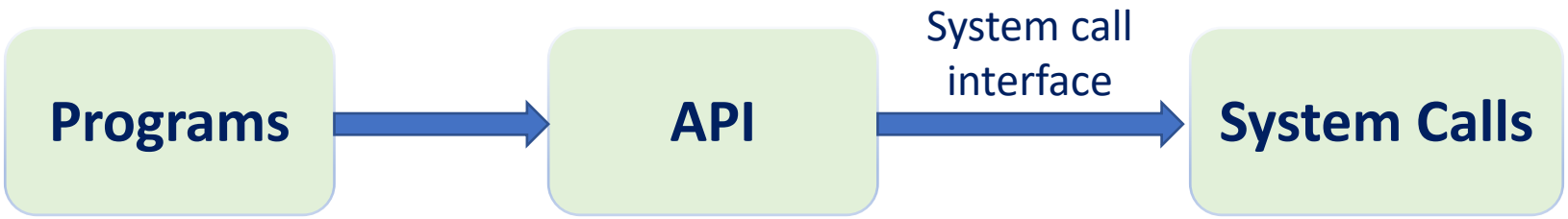
- What are system calls?
- How are system calls used?
  - What is an API?
- System Call Interface and Implementation
- System Programs

What are system calls?

- System Calls provide an interface to OS services
- System Calls are routines mostly written in a high-level language (C or C++).
- However, lower level task written in assembly.

How are system calls used?

- System Calls are accessed (by programs) via a high-level *Application Programming Interface (API)* rather than direct system call use.



What is an API?

- An API specifies a set of interfaces (functions) available to the programmer.
- These interfaces can be implemented as single or multiple system calls.
- Three most common APIs are:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Example of Standard API

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

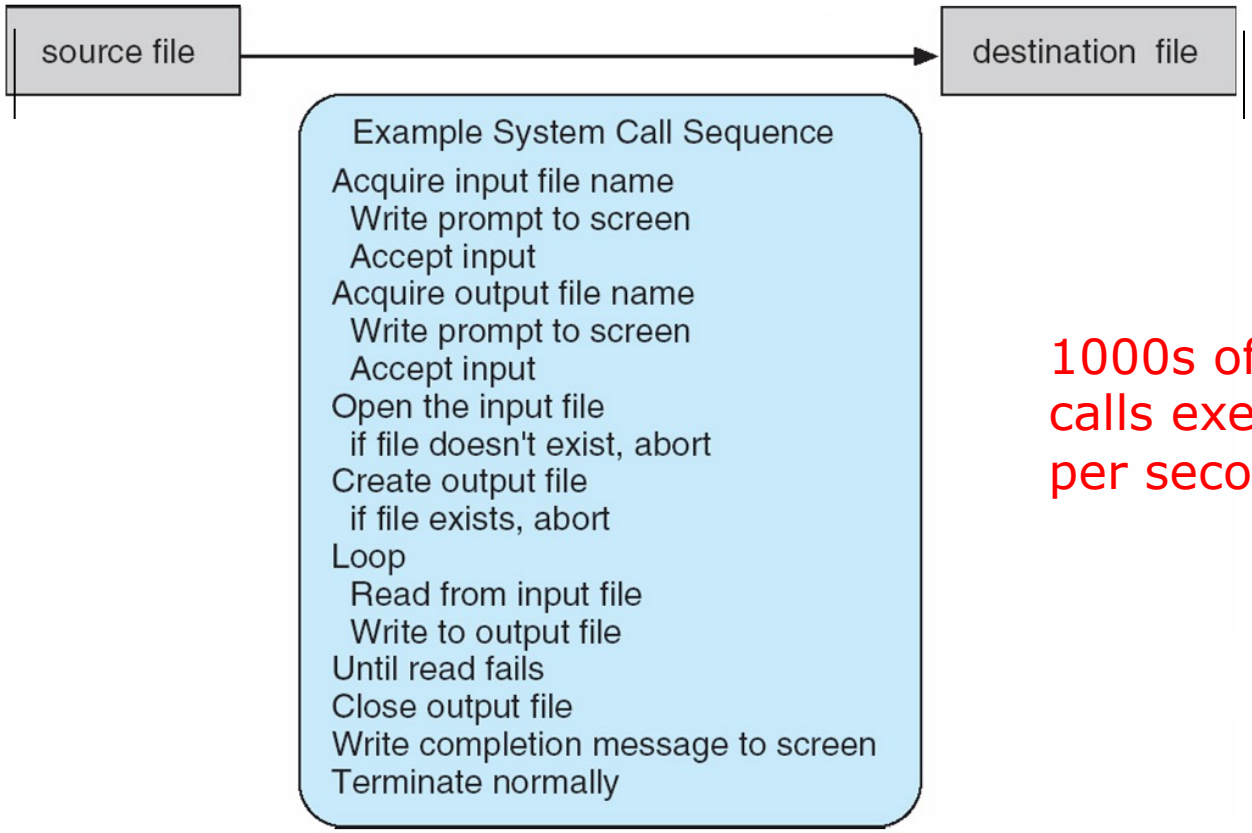
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `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.

How are System Calls used? - Example

System call sequence to copy the contents of one file to another file



1000s of system calls executed per second!

Note: Programmers don't see this level of detail.

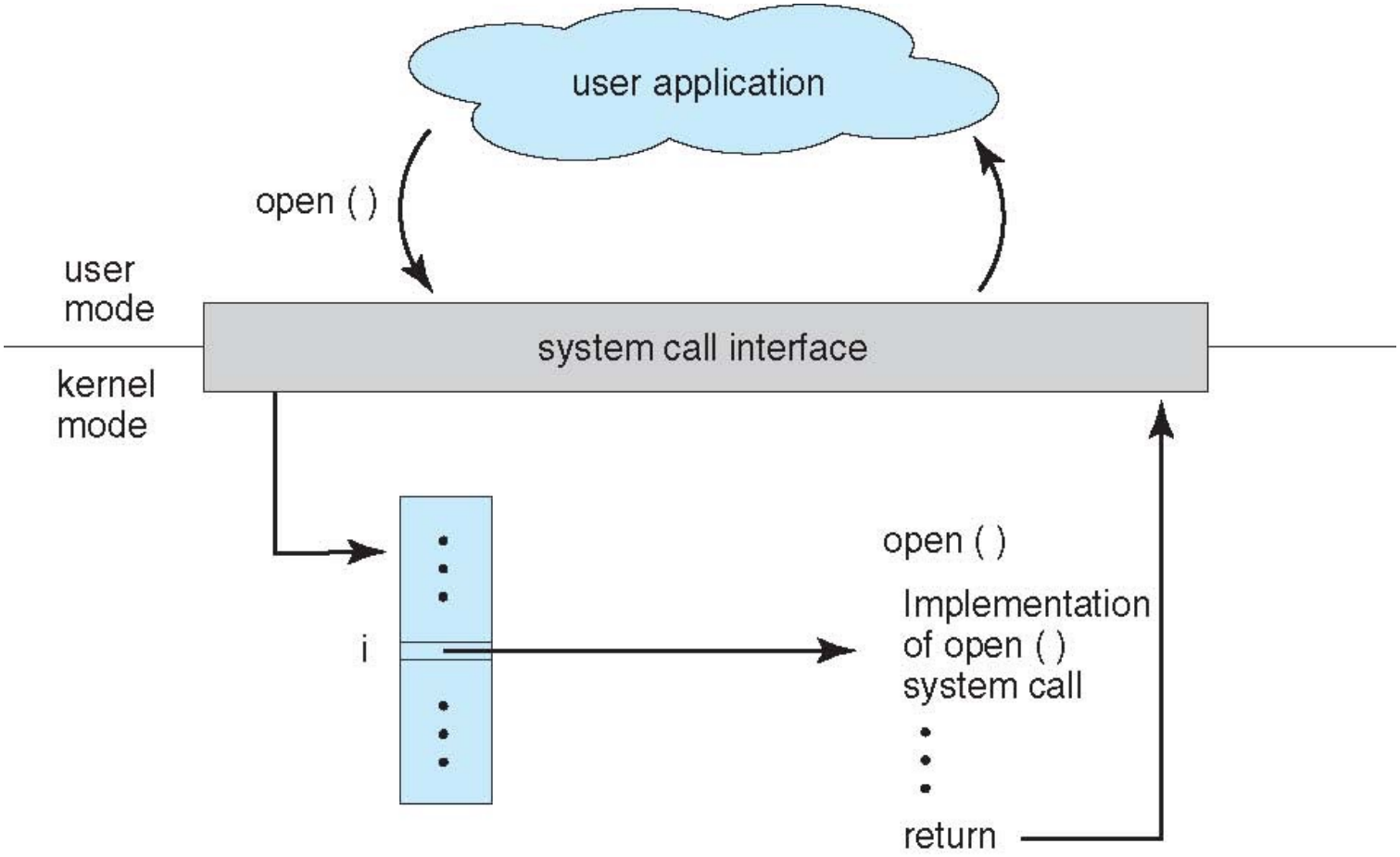
System Call Interface and Implementation

- The API makes system calls through the *system call interface.*
- **System call interface**
  - Each system call has a unique number associated with it
  - Maintains a table indexed according to these numbers.

Advantage:

- The user program need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result after the system call is invoked.
  - Most details of OS interface hidden from programmer by API

API – System Call – OS Relationship



System Call Parameter Passing

- Often, more information is required than simply knowing the identity of the desired system call.
  - Exact type and amount of information vary according to OS and call.
- Three general methods used to pass parameters to the OS
  - *Pass the parameters in registers* (simplest)
    - In some cases, may be more parameters than registers.

System Call Parameter Passing

- *Pass parameters stored in a block/table* (in memory), and address of block is passed as a parameter in a register
  - This approach taken by Linux and Solaris
- *Pass parameters placed, or pushed, on stack* that are then popped off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed.

Examples of Windows and Unix System Calls

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

System Programs

- **System program (also known as system utilities)** are programs associated with the operating system but not necessarily part of the kernel.
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- Constantly running system-program processes are known as **services**, **subsystems, daemons**
- System programs exist for several OS tasks such as: File management, programming-language support, program loading and execution, and communications