Information Technology

Year: 2021-2022

Spring Semester

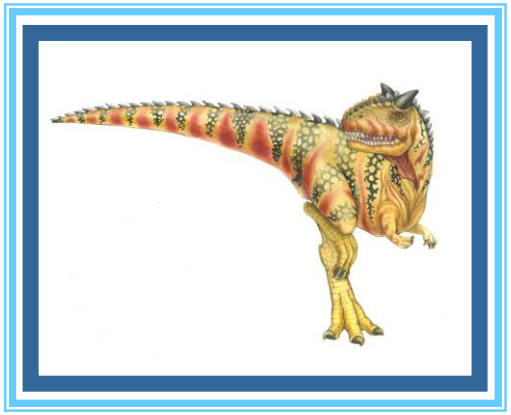
- **Operating Systems**
- Dr. Wafaa Samy

EELU

الجامعة المصرية للتعلم الإلكتروني الأهلية

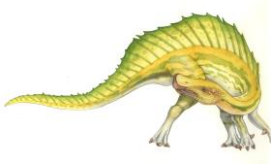
THE EGYPTIAN E-LEARNING UNIVERSITY

Chapter 2: Operating-System Services (Part 2)



Outline

- Operating System Services
- User and Operating System-Interface
- **System Calls (Cont.)**
- **System Services**
- **Linkers and Loaders**
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging



Application Programming Interface (API)

- Application developers design programs according to an **application programming interface** (**API**).
- The **API specifies a set of functions** that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.
- Three most common APIs are **Windows API** for Windows systems, **POSIX API** for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and **Java API** for programs that run on the Java virtual machine (JVM).
- A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called **libc**.
- The **functions that make up an API typically invoke the actual system calls** on behalf of the application programmer.

Example of Standard API

- For example, the **Windows function CreateProcess()** (used to create a new process) actually invokes the **NTCreateProcess() system call** in the Windows kernel.

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 into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

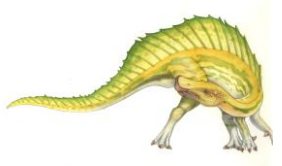
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns –1.

Application Programming Interface (API)

- Why would an application programmer prefer programming according to an API rather than invoking actual system calls?
  1. **Program portability**: An application programmer designing a program using an API can expect her program to compile and run on any system that supports the same API.
  2. Actual **system calls can often be more detailed and difficult to work with than the API** available to an application programmer.
  3. Another important factor in handling system calls is the **run-time environment** (**RTE**)—**the full suite of software needed to execute applications written in a given programming language**, including its compilers or interpreters as well as other software, such as libraries and loaders.
     - The **RTE** provides a **system-call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system kernel and returns the status of the system call.

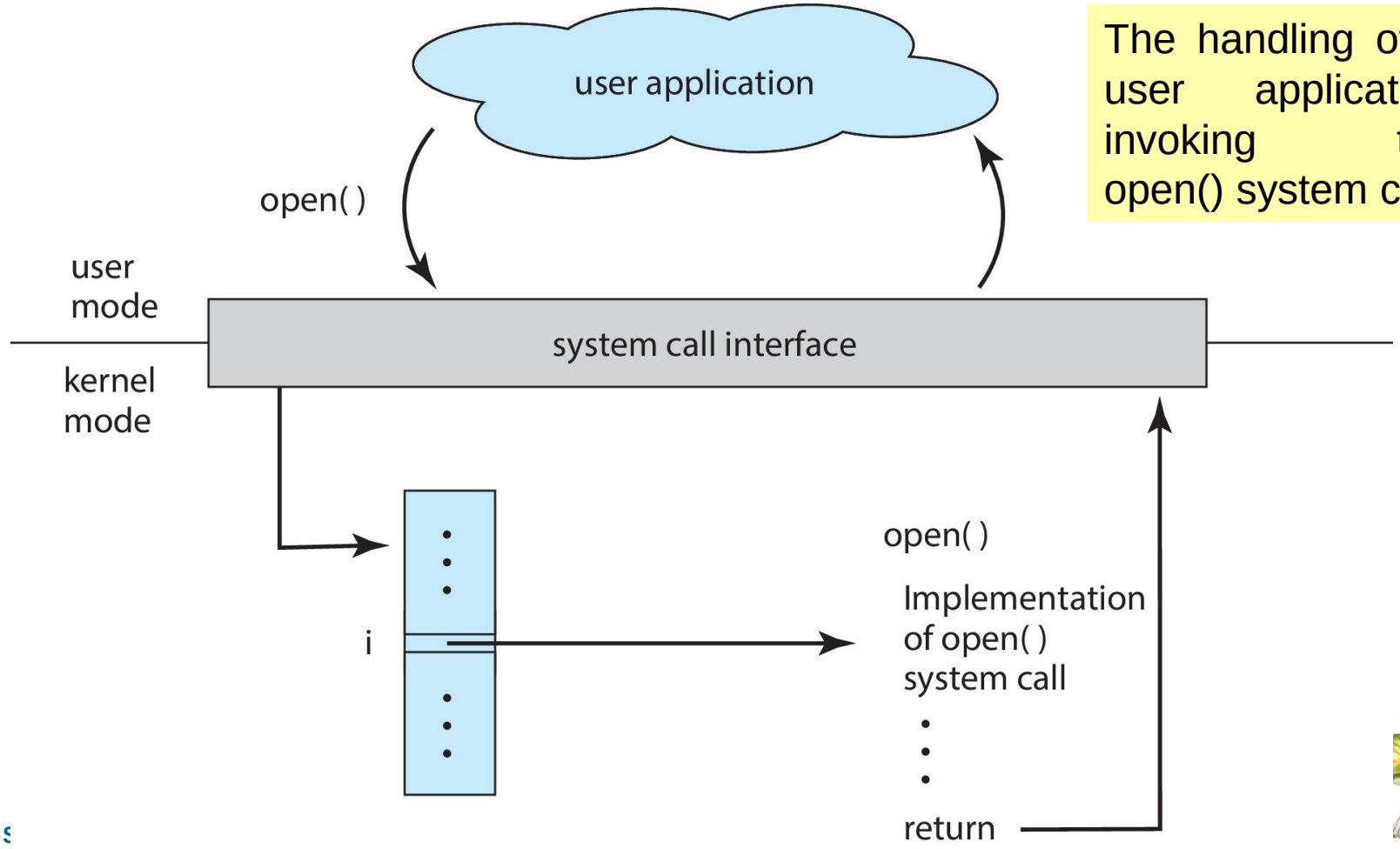
System Call Implementation

- Typically, a **number is associated with each system call**.
- **System-call interface** maintains a table indexed according to these numbers.
  - The **system call interface** invokes the intended system call in OS kernel and returns status of the system call and any return values.
- The caller need know nothing about how the system call is implemented or what it does during execution:
  - Just needs to obey API and understand what OS will do as a result of the execution of that system call.
  - Most details of OS interface are hidden from programmer by the API and are managed by the **RTE**:
    - Managed by run-time support library (set of functions built into libraries included with compiler).



API – System Call – OS Relationship

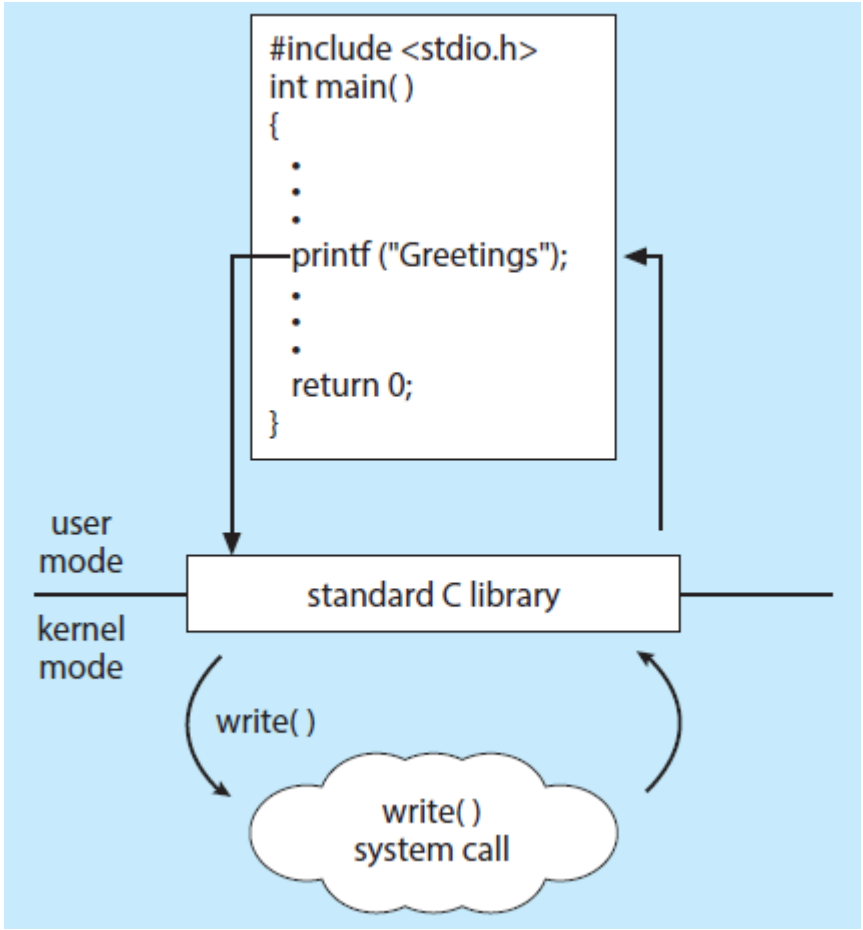
- The relationship among an **API**, the **system-call interface**, and **OS**, which illustrates how the operating system handles a user application invoking the **open() system call**.



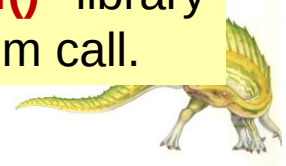
The handling of a user application invoking the open() system call.

Standard C Library Example

- The standard C library provides a portion of the **system-call interface** for many versions of UNIX and Linux.
- For 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.



C program invoking **printf()** library call, which calls **write()** system call.

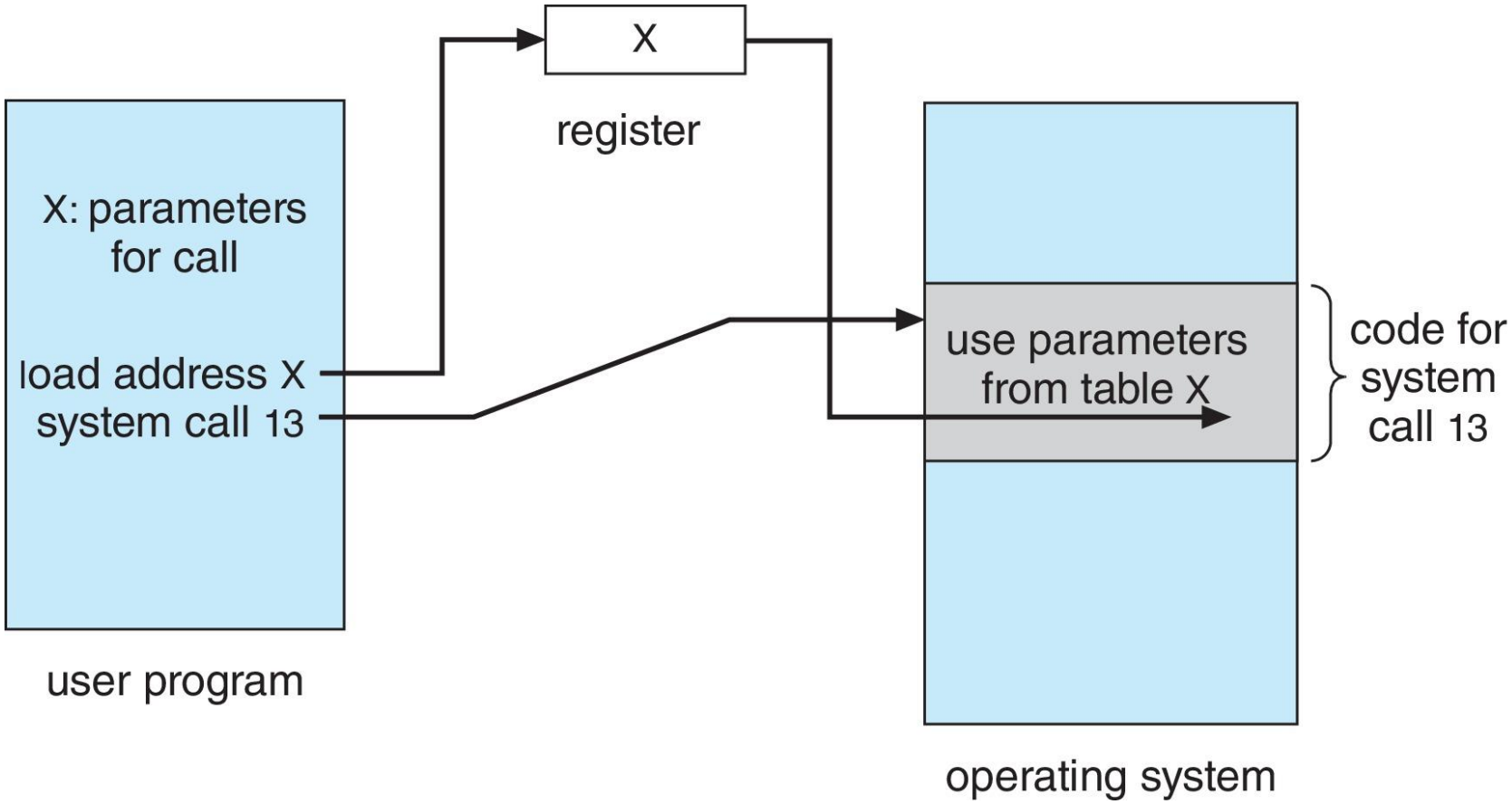


System Call Parameter Passing

- Often, **more information is required than simply identity of desired system call**: Exact type and amount of information vary according to OS and system call.
  - For example, to get input, we may need to specify the file or device to use as the source, as well as the address and length of the memory buffer into which the input should be read.
- **Three general methods** used to pass parameters to the OS:
  1. Simplest: **pass the parameters in registers**:
     - Drawback: In some cases, parameters may be more than registers.
  2. **Parameters stored in a block, or table, in memory**, and **address of block passed as a parameter in a register**.
     - This approach taken by Linux and Solaris. In Linux, if there are **five or fewer parameters**, registers are used. If there are more than five parameters, the block method is used.
  3. **Parameters placed**, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.
- Some operating systems prefer the **Block** and **stack** methods because **they do not limit the number or length of parameters** being passed.

Advantage

Parameter Passing via Table



Types of System Calls

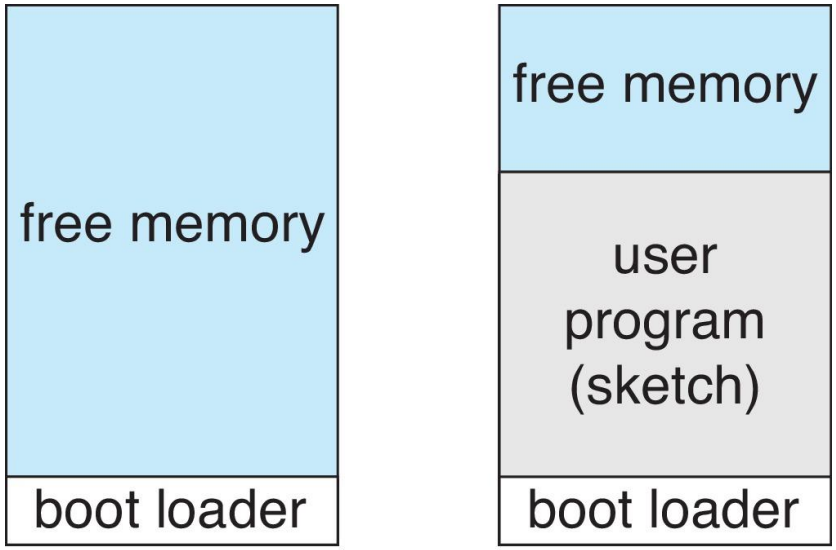
- **System calls** can be grouped roughly into six major categories:
  1. Process control.
  2. File management.
  3. Device management.
  4. Information maintenance.
  5. Communications.
  6. Protection.

Types of System Calls (Cont.)

1. Process control:
   - **Create process**, **terminate process**.
   - **End, abort:** A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()).
     - **Dump memory** if error: If a program terminated abnormally, or runs into a problem and causes an error trap, a dump of memory is sometimes taken (**written to a special log file on disk**) and an error message generated.
     - **Debugger:** system program determines errors or **bugs**, **single step** execution.
   - **Load**, **execute**: A process executing one program may want to **load()** and **execute()** another program.
   - **Get process attributes, set process attributes**. To control execution of a new created process requires to determine and reset the attributes of a process.
   - **Wait event, signal event** (i.e. processes signal when event occurred).
     - **Wait for time**. Wait for a certain amount of time to pass (wait_time()).
   - **Allocate and free memory**.
   - **Locks** for **managing access to shared data between processes**.
     - No other process can access the data until the lock is released.

Example: Arduino

- A **single-tasking** system.
- Single memory space.
- Programs (sketch) loaded via USB into flash memory.
- **Boot loader** loads program.
- Once the sketch has been loaded, it begins running, waiting for the events that it is programmed to respond to.
- No operating system.
  - The standard Arduino platform does not provide an operating system; instead, a small piece of software known as a **boot loader** loads the sketch into a specific region in the Arduino's memory.



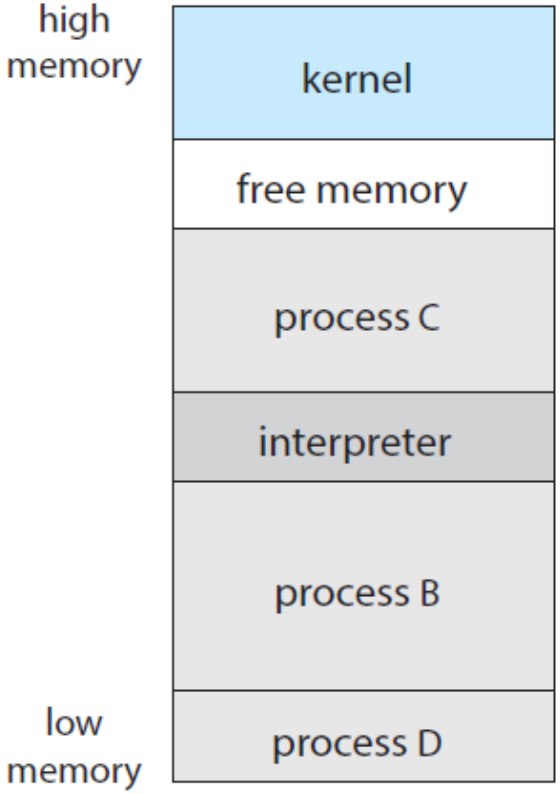
The Arduino is a simple hardware platform consisting of a microcontroller along with input sensors that respond to a variety of events, such as changes to light, temperature, and barometric pressure, etc.

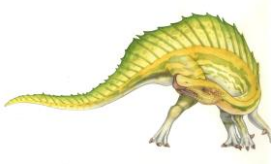
Example: Arduino (Cont.)

- To write a program for the Arduino, we first write the program on a PC and then upload the compiled program (known as a **sketch**) from the PC to the Arduino's flash memory via a USB connection.
- For example, if the Arduino's temperature sensor detects that the temperature has exceeded a certain threshold, the sketch may have the Arduino start the motor for a fan.
- An Arduino is considered a **single-tasking system**, as **only one sketch can be present in memory at a time**; if another sketch is loaded, it replaces the existing sketch.
- Furthermore, the Arduino provides **no user interface beyond hardware input sensors**.

FreeBSD running Multiple Programs

- **FreeBSD** (derived from Berkeley UNIX) is an example of a **multitasking system**.
- When a user logs on to the system, the **shell** of the user's choice is run, awaiting commands and running programs the user requests.
- **FreeBSD** is a multitasking system, the command interpreter may continue running while another program is executed.
- To start a new process, the shell executes a **fork()** system call. Then, the selected program is loaded into memory via an **exec()** system call, and the program is executed.

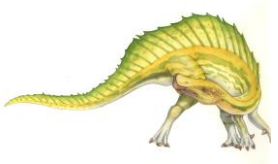




Types of System Calls (Cont.)

2. File management:
   - Create file, delete file.
   - Open, close file.
   - Read, write, reposition (e.g. rewind or skip to the end of the file).
   - Get and set file attributes (e.g. file name, file type, protection codes, accounting information, etc.).

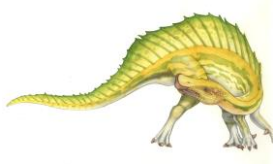
- We may need these **same sets of operations for directories** if we have a directory structure for organizing files in the file system.
- Some operating systems provide many more calls, such as calls for file **move()** and **copy()**.
  - Others might provide an API that performs those operations using code and **other system calls**, and others might provide **system programs** to perform the tasks.



Types of System Calls (Cont.)

3. Device management:
   - Request device, release device.
   - Read, write, reposition.
     - Once the device has been requested and allocated to us.
   - Get device attributes, set device attributes.
   - Logically attach or detach devices.

- A **process** may need several **resources** to execute (e.g. main memory, disk drives, access to files, etc.). If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
- The various resources controlled by the OS can be thought of as devices.
  - Some of these devices are **physical devices** (e.g. disk drives), while others can be thought of as abstract or **virtual devices** (e.g. files).



Types of System Calls (Cont.)

4. Information maintenance:
   - Get time or date, set time or date.
     - Many system calls exist simply for the purpose of **transferring information between the user program and the operating system** like a system call to return the current time() and date().
   - Get system data, set system data.
     - Other system calls may **return information about the system**, such as **the version number of the operating system**, the **amount of free memory or disk space**, and so on.
   - Get (or set) process, file, or device attributes.
     - OS keeps information about all its processes, and **system calls are used to access this information**.

- Another set of system calls is helpful in **debugging** a program. Many systems provide **system calls to dump() memory**. This provision is useful for debugging.

Types of System Calls (Cont.)

5. Communications:
   - Create, delete communication connection.
   - **Send, receive messages** if **message passing model** to **host name** or **process name.**
     - From **client** to **server.**
   - **Shared-memory model** **create and gain access to memory regions**.
   - Transfer status information.
   - Attach and detach remote devices.

- There are **two common models** of **inter-process communication**:
  - The **message-passing model**: the communicating processes exchange messages with one another to transfer information.
  - The **shared-memory model**.

Types of System Calls (Cont.)

6. Protection:
   - Get and set permissions.
     - set_permission() and get_permission() system calls.
     - Manipulate the permission settings of resources such as files and disks.
       - Get file permissions.
       - Set file permissions.
   - Allow and deny user access.
     - allow_user() and deny_user() system calls.
     - Specify whether particular users can—or cannot—be allowed access to certain resources.

- Control access to resources:
  - **Protection** provides a mechanism for controlling access to the resources provided by a computer system.

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

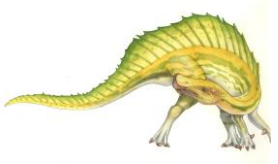
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

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

System Services

System Services

- **System programs** (or **system utilities**) **provide a convenient environment for program development and execution**.
- **System programs** can be divided into:
  1. File manipulation.
  2. Status information sometimes stored in a file.
  3. Programming language support.
  4. Program loading and execution.
  5. Communications.
  6. Background services.
  7. Application programs.
- Most **users' view of the operation system** is defined by the **application** and **system programs**, rather than by the actual system calls.



System Services (Cont.)

- **System programs** provide a convenient environment for program development and execution:
  - Some of them are simply user interfaces to system calls.
  - Others are considerably more complex.

1. **File management** - These programs create, delete, copy, rename, print, dump, list, and generally **manipulate files and directories**.
2. **Status information**:
   - Some programs ask the system for info - date, time, amount of available memory, disk space, number of users, or similar status information.
   - Others are more complex, providing detailed performance, logging, and debugging information.
   - Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI.
   - Some systems implement a **registry** - used to **store and retrieve configuration information**.

System Services (Cont.)

3. **File modification**:
   - Text editors to create and modify files stored on disk or other storage devices.
   - There may be special commands to search contents of files or perform transformations of the text.
4. **Programming-language support**: Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.
5. **Program loading and execution** – Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language.
6. **Communications** – These programs provide the mechanism for creating virtual connections among processes, users, and computer systems:
   - Allow users to send messages to one another's screens, browse web pages, send e-mail messages, log in remotely, transfer files from one machine to another.

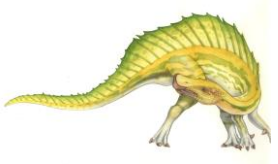
System Services (Cont.)

7. **Background Services:**
   - Known as **services**, **subsystems**, **daemons**.
   - Typical systems have dozens of daemons.
   - Launch at boot time:
     - Some for system startup, then terminate.
     - Some from system boot to shutdown.
   - Provide facilities like **disk checking**, **process scheduling** that start processes according to a specified schedule, **error logging** (system error monitoring services), and **printing servers**.

8. **Application programs:** include **web browsers**, **word processors** and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and **games**.
   - Don't pertain to system.
   - Run by users.
   - Not typically considered part of OS.
   - Launched by command line, mouse click, finger poke.

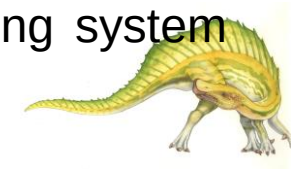
Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations.

Linkers and Loaders



Linkers and Loaders

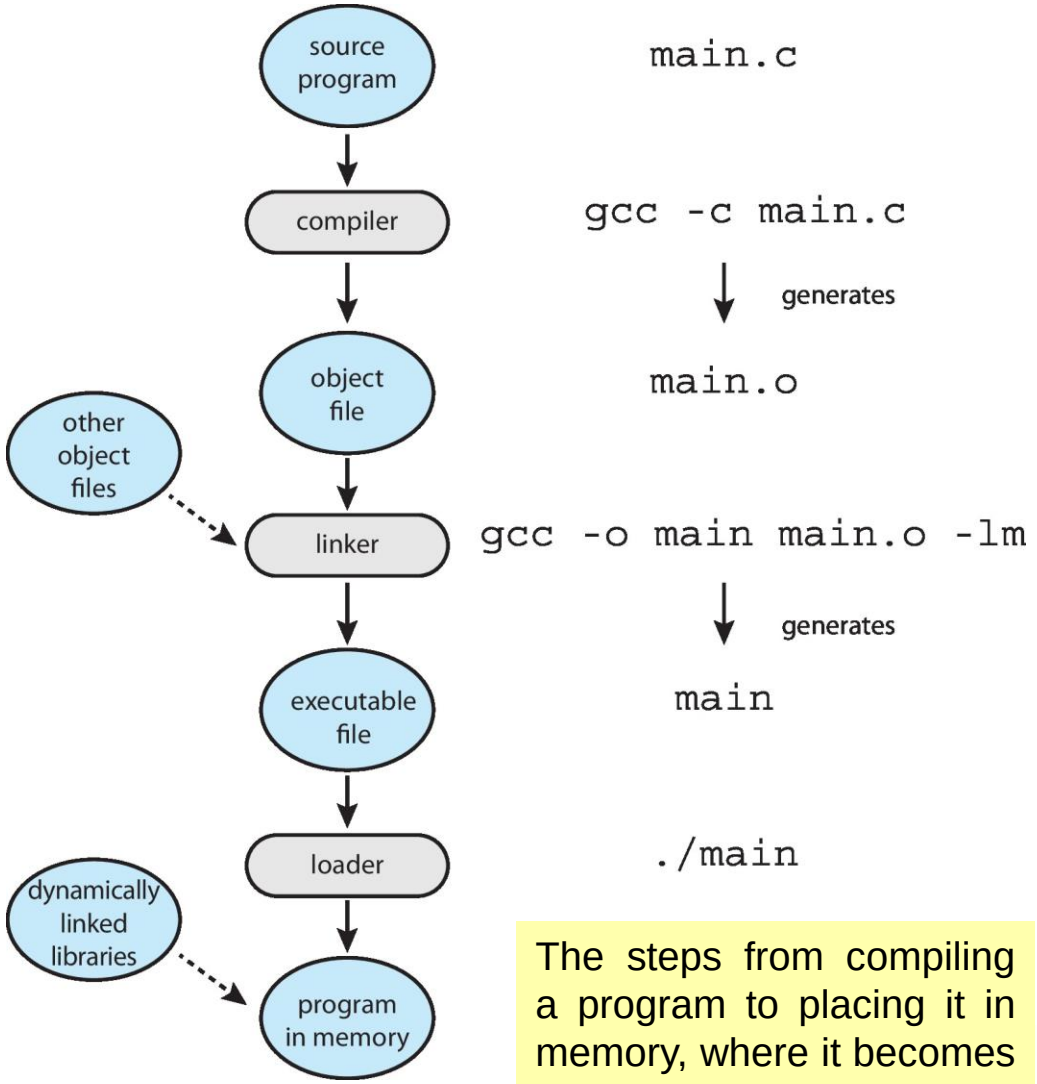
- **Source code compiled** into **object files** designed to be loaded into any physical memory location – a format known as **relocatable object file.**
- **Linker** combines these relocatable **object files** into single **binary executable file**.
  - Also brings in libraries: During the **linking phase**, other object files or libraries may be included as well (e.g. the standard C or math library).
- Program resides on secondary storage as **binary executable**, it must be brought into memory by **loader** to be executed:
  - An activity associated with **linking** and **loading** is **relocation** which assigns final addresses to program parts and adjusts code and data in program to match those addresses.
- Modern general purpose systems don't link libraries into executables:
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are **loaded as needed**, **shared by all that use the same version** of that same library (**loaded once**).
- **Object**, **executable files have standard formats**, so operating system knows how to load and start them.



The Role of the Linker and Loader

- When a program name is entered on the command line on UNIX systems—for example, ./main.
- The **shell** first **creates a new process** to run the program using the **fork()** system call.
- The shell then invokes the **loader** with the **exec()** system call, passing exec() the name of the executable file.
- The **loader** then **loads the specified program into memory** using the address space of the newly created process.

- (When a **GUI** interface is used, double-clicking on the icon associated with the executable file invokes the **loader** using a similar mechanism).



The steps from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core.

EELU

الجامعة المصرية للتعلم الإلكتروني الأهلية

THE EGYPTIAN E-LEARNING UNIVERSITY

THANK YOU FOR WATCHING

QUESTIONS?