Operating-System Services

An OS provides an environment for the execution of programs. We can identify common classes of provided services.

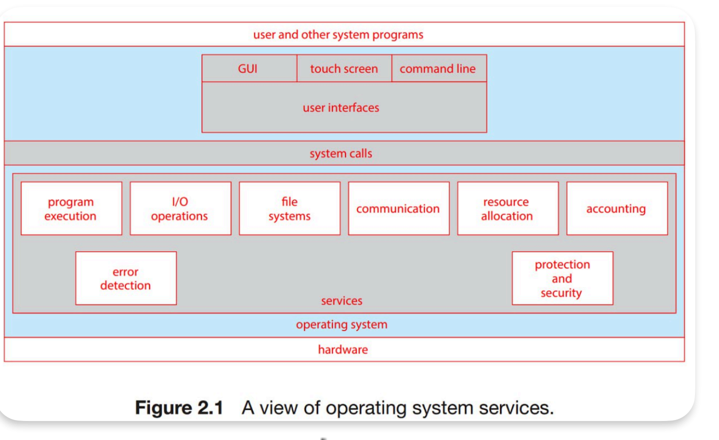


Figure 2.1 A view of operating system services.

One set of OS services provides functions that are helpful for users:

- User interface. The interface can take a several forms. It can be graphical user interface or command-line interface for example.
- **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally.
- **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the OS must provide a means to do I/O.
- **File-system manipulation.** Many OS provide a variety of file systems and means to work with files; programs need to read and write files and directories. Some OS include permissions managment to allow or deny access to files or directories.
- **Communications.** Some process may need to exchange information with another. Such communication may occur on the same computer or on different computer systems tied together by a network. Communication may be implemented via shared memory, in which processes read and write to a shared section of memory, or message passing, in which packets of information in predefined format are moved between processes by OS.
- **Error detection.** The OS needs to be detecting and correcting errors constantly. For each type of error, the OS should take the appropriate action to ensure correct and consistent computing.

Another set of OS functions exists for ensuring the efficient operation of the system itself.

- **Resource allocation.** When there are multiple processes running at the same time, resources must be allocated to each of them.
- **Logging.** We want to keep track of which programs use how much and what kind of resources.
- **Protection and security.** Protection involves ensuring that all access to system resources is controlled. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others. Security starts with requiring each user to authenticate to the system.

System calls

System calls provide an interface to the services made avaible by an OS.

To call system call usually developers use API functions. Behind the scenses the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

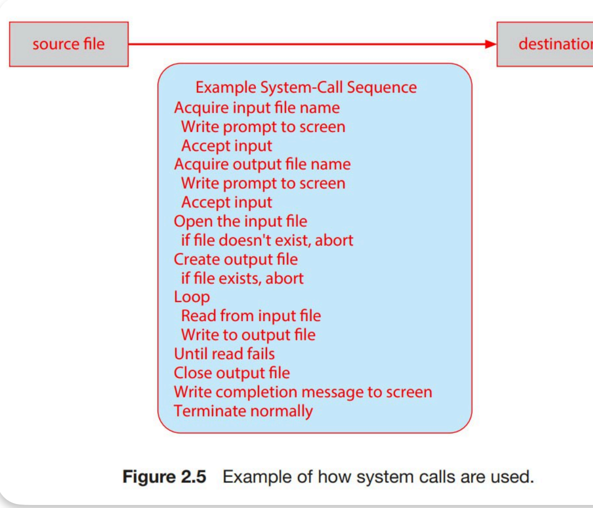


Figure 2.5 Example of how system calls are used.

example of API that invokes system call

```
#include <unistd.h>

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

return value | function name | parameters

Another important factor in handling system calls is the run-time environment — the full suite of software needed to execute applications written in a given programming language, including its compilers and interpreters as well as libs and loaders. The RTE provides a system-call interface that serves as the link to system calls made avaible by the system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within OS.

A number associated with each system call and system-call interface maintains a table indexed according to these numbers. The system-call interface then invokes the intended system call in the OS kernel and return the status of the system call.

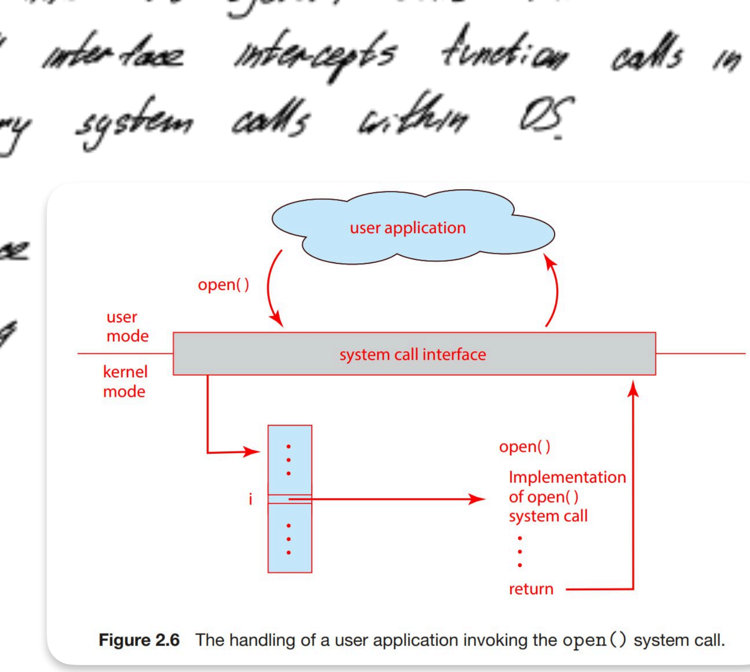


Figure 2.6 The handling of a user application invoking the open() system call.

Types of system-calls

System calls can be grouped roughly into six major categories:

- process control
- file managment
- device managment
- information maintenance
- communications
- protections

- Process control
  - create process, terminate process
  - load, execute
  - get process attributes, set process attributes
  - 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
- Protection
  - get file permissions
  - set file permissions

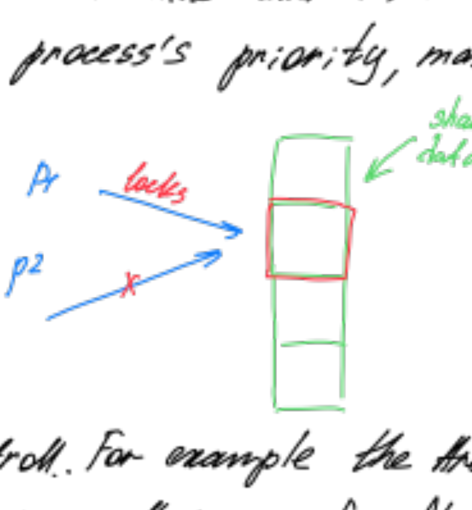
Figure 2.8 Types of system calls.

Process control. If a program ends execution abnormally, a demp of memory is sometimes taken and an error message is generated. The demp is written to a special log file on disk and may be examined by a debugger. Under either normal or abnormal circumstances, the OS must transfer control to the invoking command interpreter. The command interpreter then reads the next command.

A process executing one programm may want to load() or execute() another program. This feature allows the command interpreter to execute a program as directed by a user command or the click. If control returns to existing program when the new one terminates we must save the memory image of existing program; thus we have effectivly created a mechanism for one program to call another one.

If we create new processes we should be able to control its execution. This control requires the ability to determine and resets the attributes of a process, including the process's priority, maximum allowed execution time and so on.

Quite often processes may share data. To ensure the integrity of the data being used, OS often provide system calls allowing a process to lock shared data.



There are many variations in process controll. For example the Arduino platform doesn't provide an OS; instead is small piece of software known as boot loader loads the compiled program (sketch). Once the sketch has been loaded, it begins running. If another sketch is loaded, it replaces the existing sketch.

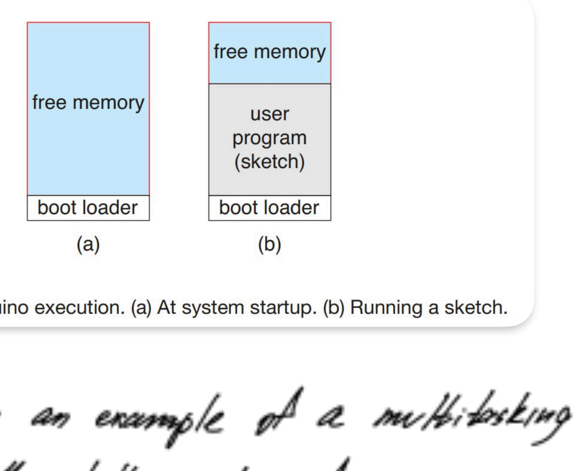


Figure 2.9 Arduino execution. (a) At system startup. (b) Running a sketch.

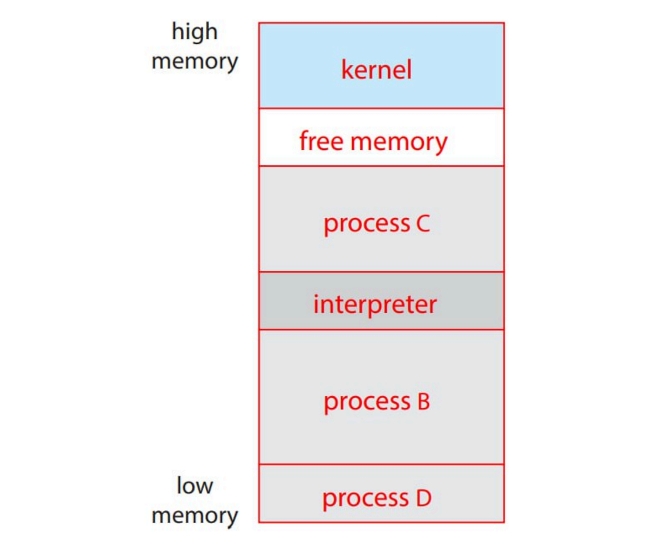


Figure 2.10 FreeBSD running multiple programs.

FreeBSD is an example of a multitasking system. Usually shell awaiting for users command to execute. To start new process the shell executes **fork()** system call. Then the selected program is loaded into memory via an **exec()** sys call, and the program is executed. When the process is done, it executes an **exit()** sys call to terminate returning status code 0.

Device managment. A process may need several resources to execute — main memory, disk drives, access to files and so on. If the resources are avaible, they can be granted, and control can be returned to the user process. Otherwise, will have to wait until sufficient resources are avaible.

A system require to first request() a device to ensure exclusive use of it. After we finished with the device, we release() it. Once the device has been requested we can read() write() reposition() the device just as we can with files. Many OS merge I/O devices and files into a combined file — device structure.