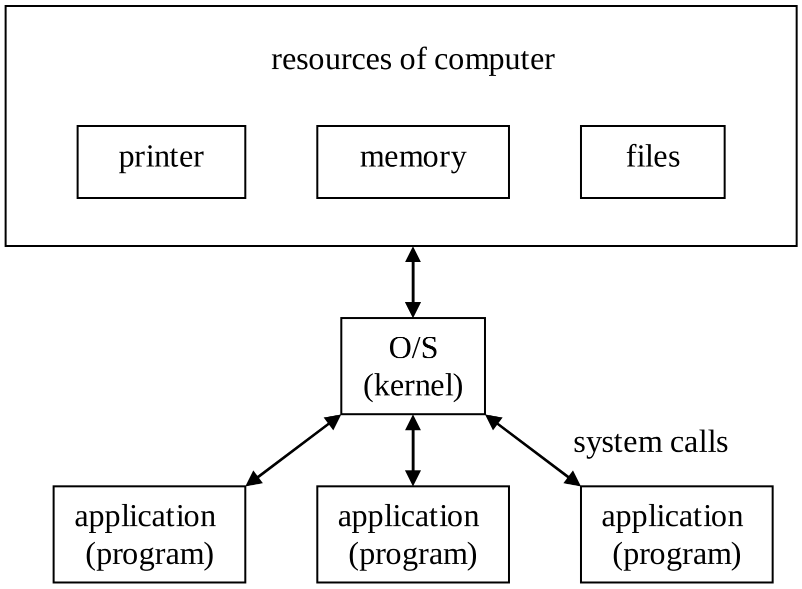
ECE 222 System Programming Concepts
Lecture notes – System Calls

An operating system (O/S) is a program with two jobs: it manages the resources of the computer (peripherals, files, memory, etc.) and it manages all other programs running on the computer.



The programs we write access the resources of the computer by calling functions executed by the kernel. These functions are called **system calls**. A collection of system calls is sometimes referred to as an application program interface (API).

There are dozens of system calls supported by any linux or unix kernel. Some common examples include (none of these lists is all-inclusive):

- file I/O: open(), read(), write(), close(), creat() (yes there is no e – this is not a typo), lseek(), link(). These system calls ask the O/S to access a file.
- process control: fork(), execl(), execv(), wait(), system(). These system calls ask the O/S to run another program, or control how it runs.
- memory management: mmap(), brk(). These system calls ask the O/S to provide memory to be used by the application program.
- time management: time(), setitimer(), settimeofday(), alarm(). These system calls ask the O/S to access the system clock, in some cases taking an action that affects a program when a given time interval has passed.
- interprocess communication (IPC): signal(), pause(), kill(), sigaction(). These system calls ask the O/S to send a message to a program. For most of these system calls, the messages usually originate in the kernel. Although they can be used to send messages from one application program to another, there are more advanced system calls better suited for this job.

Other system calls include those used for message passing, shared memory, semaphores, thread management, and network management (sockets). All of these are beyond the scope of this course, and are typically covered in a full operating systems class.

Note that we have already seen some of these system calls. Others look a lot like other functions we have already seen. For example, what is the difference between open() and fopen(), or read() and fread()? The former are system calls, meaning the code executed is part of the kernel. The latter are C standard library calls. The code for them is sitting in /usr/lib/libc.a, where we have already seen the code for printf() and other functions. The calling of those functions is handled differently. If a library is linked statically, then the code is already part of the running program, and is executed directly. If a library is linked dynamically, then the kernel handles it (the details beyond the scope of this class).

For most programs, the differences between system calls and library calls are moot. However, when coding for device drivers or other low-level operations, or for serious applications, the differences can become very important:

1. Standard library functions are built on top of system calls. For example, the malloc() and related functions are built on top of the mmap() and brk() functions, meaning they call them to get the job done. The malloc() function is in the C standard library, while mmap() is a system call. This means that greater control over the computer's resources can be obtained through system calls.
2. Standard library functions tend to be more standardized than system calls. They both have standards bodies: the ANSI group defines library standards, while the POSIX group defines system call standards. However, since system calls provide direct access to the kernel, different O/S's (including different variations of unix/linux) will have at least slightly different system calls.
3. System calls cause a "context switch", meaning that the application program stops while the kernel program runs for a while to perform the requested operation. Because of this, a system function call takes longer than a normal function call within your own program. Standard library calls often optimize operations to minimize the number of system calls, thus speeding up program execution.

As an example of the latter, consider again the open()/read()/write()/close versus fopen()/fread()/fwrite()/fclose() function calls. The former are system calls, while the latter are C standard library function calls. The latter are more efficient, because they use buffering. When an fread() call is made, more than the requested amount is read() from the file. The extra bytes are held in a buffer, local to the C standard library, and not directly accessible by your program. When your program next calls fread(), it may be able to satisfy the request using bytes already in its buffer, eliminating the need for another read() system call.

The man pages for system calls are stored in a separate folder than the man pages for library calls, and both are stored separate from system programs:

```
man 2 stat          [provides man page for system call stat()]
```

man 3 printf	[provides man page for C library call printf()]
man 1 ls	[provides man page for ls system program]

By default, man will start in “section 1”, and look upwards. So if you leave out the number, you get the lowest section number that has a man page with the given name. This can be confusing, because for example there is a stat program in section 1, as well as the stat() system call in section 2.

The file I/O system calls all work very similarly to the C standard library calls, except for the latter include buffering. Therefore we will not look more closely at them. We will however take a closer look at two more groups of system calls: process control, and basic IPC.

Summary:
What is a system call; why would a program make one?
What are the differences between a system call and a library function call?
Is malloc() a system call or a library call?
What is the difference between read() and fread()?