What is an operating system?

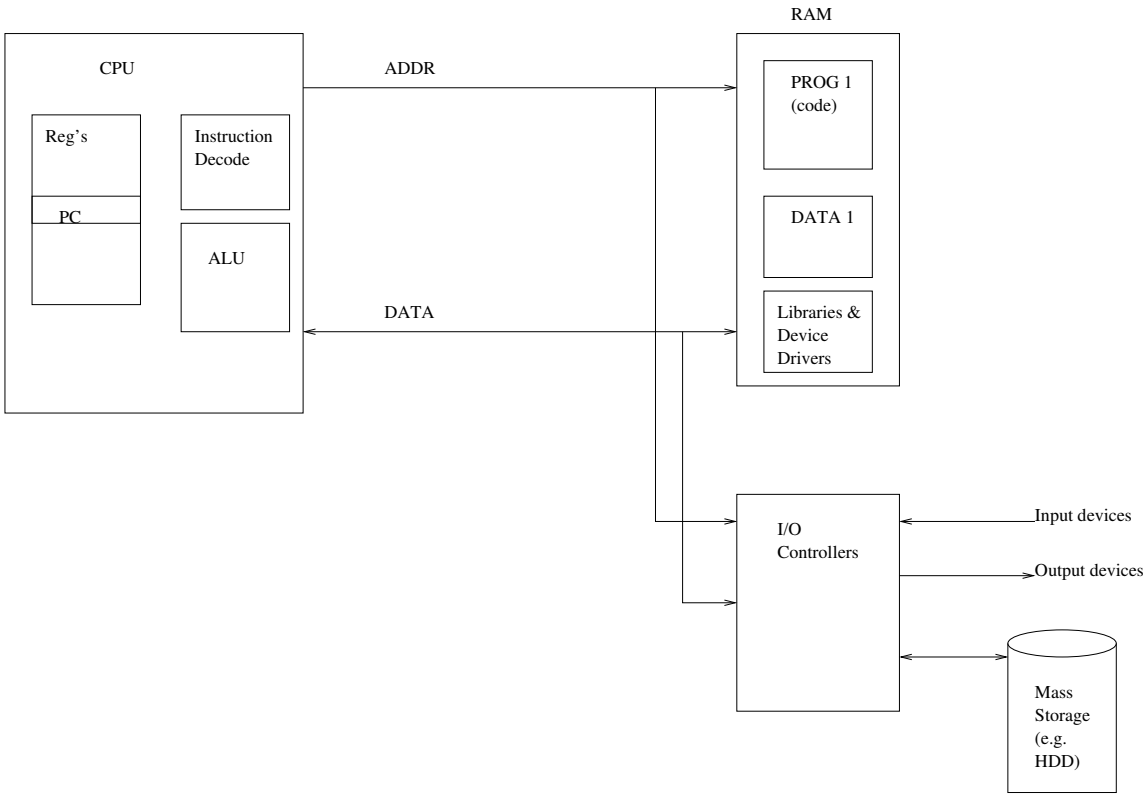
The term "operating system" means different things to different people. In order to understand the scope and role of an Operating System as it pertains to this course, let us briefly examine the history of computing. *The more things change, the more things remain the same*. The computing industry is the leader in wheel re-invention. Many of the classic problems in computing, including operating systems design, are "re-discovered" every few decades.

The earliest "computers" were really just calculators. They could perform a fixed task, such as summing a list of numbers, and some could be re-programmed in a sense by changing the wiring, but they were not capable of accepting and running an arbitrary software program.

With the introduction during World War II of Von Neummann architecture, the concept of the "computer program" was born. The operation of the computing machine could be described as a sequence of primitive instructions, or "opcodes", which could be represented as binary numbers. Therefore, it became possible to consider a computer program as another form of data which could be stored, processed and manipulated. Initial entry of a program into memory was typically in the form of punched paper tape or card stock media, or through manipulation of front-panel switches. Magnetic media such as drums, disks and tapes were later introduced to store and transport large amounts of data.

During these early days, any subroutines necessary for the operation of the machine itself, such as device drivers for I/O devices, had to be included with each program. Obviously this introduced considerable repetitive bulk. So, the very first "operating systems" were this collection of device driver subroutines and other commonly used routines that were kept semi-permanently in the machine's memory. (*It might be interesting to learn that most computer memory of this time period was magnetic "core" arrays, which had the property of retaining their contents with power off.*)

In this "batch" mode of operation, the OS remained resident while each user's "job" was loaded into memory, executed, and the output produced, typically on a line printer or card puncher. Only one job could be run at any given time, leading to long queues and poor development cycles (imagine waiting 45 minutes to find a syntax error!). A lot of a job's time was spent idly waiting for slow I/O operations to complete. The model below depicts, schematically, such a simple computer that does one program at a time.



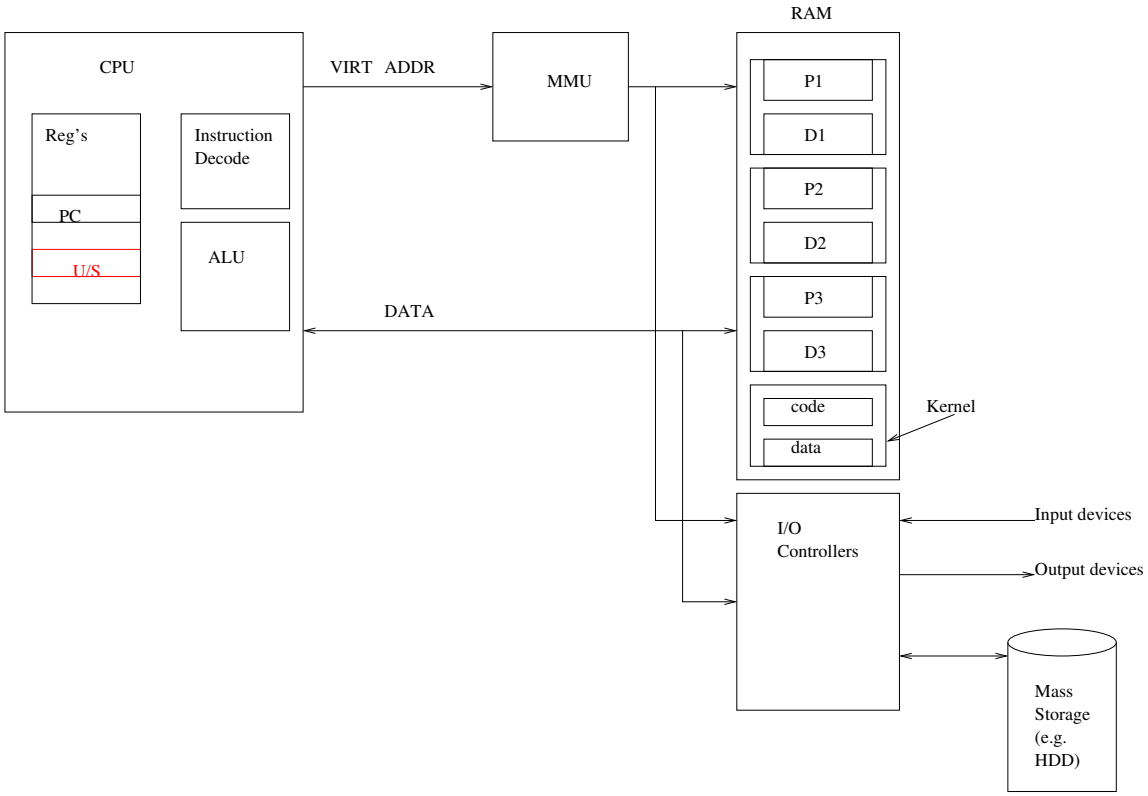
The advent of "multiprogramming", later called "multitasking", made use of this idle I/O wait time. Instead of being restricted to a single task, multiple jobs could be loaded into the machine. They would take turns executing; when a job blocked on I/O, another job would be selected to run. After the I/O operation finished, the first job would become eligible to run again and would run as soon as its turn came up. The operating system under a multitasking system has the responsibility of switching the use of the processor between jobs.

With multiple jobs, frequently belonging to multiple users, being executed at the same time on the same machine, the errant operation of a program was no longer a mere inconvenience. It became a serious problem in that if one program crashed, the entire machine crashed. In addition, a malfunctioning program could accidentally execute instructions or access memory locations or device registers that could physically damage the machine (many electromechanical peripherals such as printers and disk drives were controlled directly, such that it was possible to e.g. run the disk drive head off the end of the platter) or cause the destruction or corruption of data. Unacceptable!

The solution was to introduce hardware-based protection. Computers with such protection run in one of two modes. In the "Supervisor", aka "Kernel", aka "System", aka "Privileged", aka "Real" mode the system behaves identically to an unprotected system. In "User" mode, certain instructions and direct access to the machine's hardware are disallowed, including, obviously, the ability to switch the machine back into Supervisor

mode. Further protection is afforded by Memory Management hardware (MMU) which partitions the physical random-access memory (RAM) of the machine into sections. An attempt to read or write to an address outside of a program's assigned address space causes the operation to be denied and an error raised.

The Operating System could now be clearly viewed as having a "kernel", meaning the collection of operating system code which is run in Supervisor mode. When the machine is first bootstrapped, the kernel is loaded into memory and execution begins with the machine in Supervisor Mode. When the kernel selects a user program to run, it transfers control to the user program while at the same time de-activating Supervisor mode. The user program runs in User Mode, whereby it is prevented from causing harm to the system through errant operation.



As an aside, note that the rise of the "Personal Computer" in the late 1970s and early 1980s in many ways represented a regression. The Personal Computer was, in terms of architecture and sophistication, equivalent to the more primitive machines of the early 1960s. It was distinctly personal, single-user and single-tasking. The ability to do more than one thing at a time and the idea that a single bad program should not crash the entire machine took many years to re-enter the computing mainstream, although such concepts had been understood and accepted for over 20 years prior.

Today, operating systems are found on a wide variety of computing devices with varying amounts of memory, storage, processing and connectivity. Traditional categorization

based on market target (such as "desktop" or "server") is confusing at best because almost all operating systems have some crossover components. But here are some "tags" and their commonly understood meanings:

• Personal: designed to provide services to a human user who is locally connected to the computing device via human input/output devices such as keyboard, display or mouse/touchscreen.

• Server: designed to provide services to remote human users or non-human programs running on other devices, by means of a communications network.

• Embedded: designed primarily to monitor and/or control real-world physical systems. E.g. automotive engine controls, traffic lights, biomedical.

• Mobile: designed to provide connectivity and content services to mobile human users.

In this course, the majority of discussion will be of server and personal operating systems. We will concentrate primarily on the kernel, both its internal construction and the programming interfaces to it. Although there are software components, such as system libraries, which are often thought of as part of the operating system, we will attempt to keep the distinction clear between what happens within the kernel, and what occurs outside of it.

Services Provided by the Operating System

When people speak informally about Operating Systems, they often include software components which are not part of the kernel. These components may be libraries that are supplied by the system, programs which run in the background to provide services, or even entire applications. Here are some of the broad categories of services which are provided by the kernel itself, and which we will be examining during the course:

• Bootstrap/Initialization: Bringing the machine from an initial state after power-up or reboot to a functioning state.

• Device Drivers: Interfacing with devices at the hardware level and isolating the programmer or end user from those details.

• Storage/Filesystems: Managing bulk storage devices such as disks and organizing them to provide directories, files, etc.

• Memory Management: Controlling the pool of physical memory and allocating it as needed using the virtual memory illusion.

• Scheduling: Controlling the illusion of virtual processors and allocating processor resources fairly among competing processes.

• Security: Determining the identity and authorization profiles of users and programs, and enforcing the desired protections against unauthorized access.

• Communication: Allowing programs and users, either on the same machine or across a network, to communicate.

Illusions

The operating system kernel assumes control of the computer at boot time and remains in command as the system runs. The kernel periodically relinquishes control, in a deliberate and regulated manner, to user-mode processes. The kernel, in conjunction with the processor hardware, is able to maintain several critical *illusions* with respect to the standpoint of these user-level programs.

The first illusion is that of **virtual processors**, or **multi-tasking**. The use of the processor (for now, to simplify the material, we will assume a single-processor system, but later in the course we will fully explore modern multi-processor designs) is shared among the many user-level programs currently running on the computer. In effect, each program has the use of a virtual processor.

This sharing is done by **time-slicing**, such that program A runs on the processor for a number of instruction cycles, then another program B runs, then program C, etc. and eventually program A runs again. When execution resumes in program A, all registers and other critical state information are restored and execution continues just where it had left off, through a software technique known as *context switching*. Thus, the fact that other programs had run in the intervening time is transparent to the program's code.

The second illusion is **virtual memory**. When user-level code is executing, all addresses used in the program are virtual addresses. A component of the processor hardware called the **memory management unit** (MMU) sits between the virtual addresses which are generated by program execution (instruction fetches and data read/writes) and the physical addresses which are output on the address pins of the processor.

The kernel, by using the processor registers which control the MMU, is able to establish the mapping between virtual and physical addresses. This can be changed by the kernel as each program executes. Hence the kernel has the ability to control completely all of the memory which a user-level program can "see."

The kernel can also use the MMU and its control over virtual address space to allow the grand total of RAM in use by all of the programs on the computer exceed the actual amount of physical RAM. It does this by using disk or other storage to hold the data. When an area of a program's virtual address space does not currently have an actual place in physical memory to hold it, the MMU flags that area as missing. Then, when the program attempts to access the missing area, an interrupt is generated. This is known as a **page fault**. The kernel allocates an available chunk of physical RAM and maps it in, then resumes execution of the program.

Therefore, we see that the kernel has complete control over the world view of any program on the computer. This combination of virtual processor and virtual address

space forms a virtual equivalent of the the old-fashioned single-purpose computer for each program. Such a "virtual computer" (note: this is not a reference to virtual machine (VM) software such as VMWare or VirtualBox) is known as a **process**. The process is the cornerstone of multi-user, multi-tasking computing. In UNIX operating systems, each command launched from the shell such as `ls` is executed in a new process. We can think of a process as a brand-new virtual computer into which the program is loaded and executed. When the program has completed, the virtual computer is discarded.

Because of time-slicing, the actual run time of a program as observed by an external observer may vary from run to run, depending on how many other processes shared the use of the CPU. The operating system records the total amount of time for which a given process has use of the processor. This time is divided into two measurements: **user time** and **system time**. The former is the time the process spent executing its user-level code, and the latter is the time spent executing kernel-level code on the behalf of the process (e.g. responding to system calls as described below). The sum of these two measurements is the actual time spent doing useful work, and will always be less than or equal to the **real time** elapsed, which is also called **wall time**, i.e. the time spent by the user staring at the wall waiting for the program to complete.

We will explore processes, time-slicing and virtual memory at great length in subsequent units.

Kernel Re-entry: The System Call

A user-mode program finds itself executing within the protective bubble of a process. It is not able to touch control registers, I/O devices or other sensitive resources directly. There must be some means of penetrating that bubble in a controlled manner and gaining access to required resources and services. We shall see that there are three basic mechanisms, all of which are supported by hard-wired logic built in to the CPU. These are: hardware interrupt, instruction fault, and system call. In this unit, we will consider the last mechanism, as an introduction to user-kernel interfacing.

A System Call is a controlled transition from user mode back into kernel mode. This requires execution of hardware-specific, specialized opcodes. Because the method of making a system call will vary from platform to platform, this detail is isolated from the programmer by the system libraries. When writing a program in the C language, a system call appears to be just another function call. It is the convention among UNIX-like systems that system calls are documented in section (2) of the man pages, while other standard library functions are in section (3).

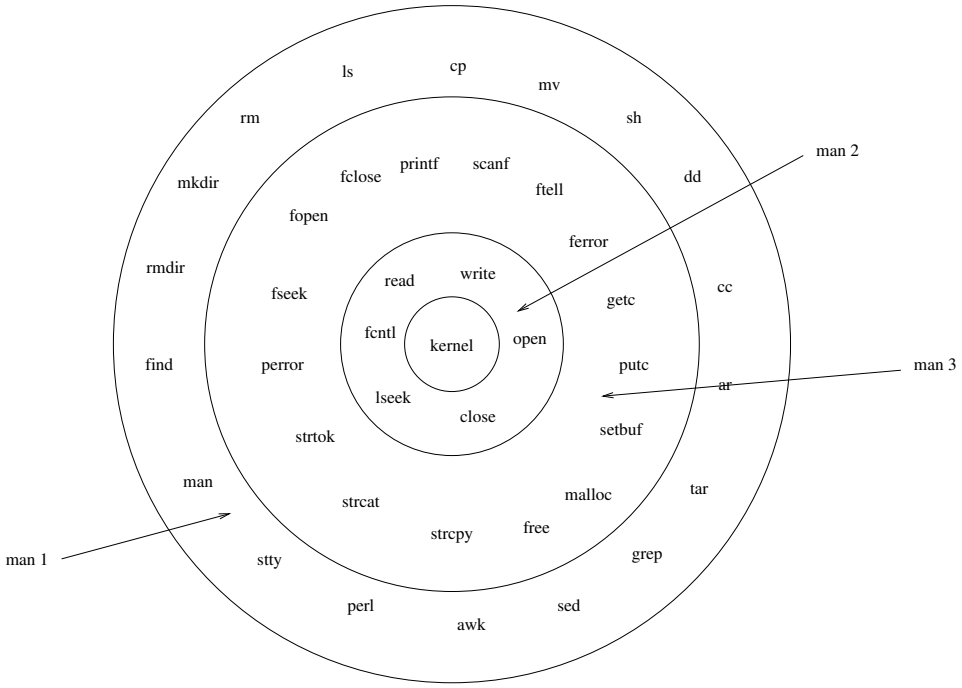
The set of system calls available to the user program is well-defined and well-documented. The kernel must be extra-careful to verify all of the parameters passed to it during the system call and avoid accidental or deliberate corruption of the system through garbage data.

The following is an overview of a system call:

• The user-mode program makes an ordinary function call to a function of a specified, standardized name, such as `open`. This function is provided by the system libraries (`libc`) and is a hybrid function: it is coded partially in C, and partially in assembly language.
• The system call takes the parameters which are passed as ordinary function call parameters, and places them in the proper place (usually registers) where the kernel expects them to be. It then invokes a special assembly language instruction which performs the system call.
• The CPU responds to this instruction by simultaneously elevating back to Supervisor mode and jumping to a pre-arranged location in the kernel.
• The kernel's system call entry handler verifies all the parameters and if they are correct, executes the system call. E.g. in the Linux kernel, the `open` system call in user mode corresponds to a function `sys_open` that is invoked in kernel mode.
• The system call may take some time to complete. During that time, other processes may be running. When the system call does complete, it has a return value. This is either -1 (indicating an error) or a non-negative number, which indicates success, and possibly a value (such as the number of bytes read or written).
• The kernel returns from the system call handler. Hardware restores to the User mode and execution resumes in the user-level library function, which takes the system call return value and makes it appear to be the return value from the ordinary C function.

UNIX Software component layers & man pages

The figure below depicts how the standard I/O and standard C library functions which are documented in man section 3, call system call functions which are in section 2 of the man pages, providing access via the system call interface to the kernel itself. The commands which are typically associated with the UNIX command-line environment, in section 1 of the man pages, are built on top of the standard library functions.



The list of defined error numbers is provided in an include file, which provides symbolic `#define` names for these constants. The return value used to flag an error, as well as which error codes may be returned, is part of the documentation for each system call. To call up, e.g., the documentation for the `open` system call, use the command `man 2 open`. The numeral refers historically to section 2 of the UNIX Reference Manual, which at one time was a printed document that came with your UNIX system. All system calls are contained in section 2. Under some UNIX systems, such as Solaris, the syntax would be `man -s2`.

It is convention that system commands are in section 1 of the manual. System calls are in section 2. Standard C library functions and functions in other libraries, such as the math library, are in section 3. These 3 sections are universal among all UNIX variants. Linux uses section 4 for information on specific hardware devices. Section 5 documents system file formats. Section 6 is for games. Section 7 documents network protocols, how the system bootstraps, character sets, and how certain kernel features works. Section 8 documents system maintenance and administration commands. You may find that sections 4 and higher are organized differently on different UNIX variants such as Solaris or BSD.

The UNIX I/O model

The set of system calls for performing I/O under UNIX is somewhat analogous to the stdio library routines. To open a file:

```
#include <fcntl.h>           /* Defines O_XXX constants */

int fd;                     /* Must be signed integer */
char *fname;

          fd=open(fname,O_RDONLY);     /* Similar to fp=fopen(fname,"r") */
          fd=open(fname,O_WRONLY|O_CREAT|O_TRUNC,0666);   /* fopen(fname,"w") */
          fd=open(fname,O_RDWR);                 /* fopen(fname,"r+") */
          fd=open(fname,O_RDWR|O_CREAT|O_TRUNC,0666);     /* fopen(fname,"w+") */
          fd=open(fname,O_WRONLY|O_CREAT|O_APPEND,0666);  /* fopen(fname,"a") */
          fd=open(fname,O_RDWR|O_CREAT|O_APPEND,0666);    /* fopen(fname,"a+") */
```

The value returned from open is known as a *file descriptor*. The file descriptor is essentially a "handle" or a "cookie" which can be used by the program in subsequent system calls to refer to the open file. It is a small integer with a range of 0 to N, where N is at least 19 on every variant of UNIX known to mankind, and is frequently much larger. (E.g. on Linux systems, the maximum number of open files is a configurable resource limit. You can observe these limits with the command `ulimit -a`. The default limit on file descriptors is 1024)

Once a program is done with the file, it can call `close(fd)`. This is analogous to `fclose(fp)`. All open files are also automatically closed when a program exits. It is often a good idea to explicitly close a file when you are done with it, and check the close for error, because there are some error conditions which can occur when writing to a file or file-like entity such as a network connection, which will not be detectable until the file is closed. Once closed, the file descriptor number may be given out again by a subsequent open system call. In fact, the lowest available (non-open) file descriptor number is generally given out.

We see that the open(2) system call can take either 2 or 3 parameters. The meaning of the third parameter, which sets the file access modes for newly created files, will be explained in a subsequent unit.

The second parameter to open(2) is an integer which is a bitwise combination of flags defined as O_XXXX in the header file `<fcntl.h>`. For example, on a particular Linux system, these constants are defined thus (some definitions have been ommited for clarity).

```
#define O_RDONLY             00
#define O_WRONLY             01
#define O_RDWR               02
#define O_CREAT            0100
#define O_EXCL             0200
#define O_TRUNC           01000
#define O_APPEND          02000
```

We see that the bits are defined as octal constants, and can be combined using the logical OR operator. This idiom is very common for UNIX system calls.

The `<fcntl.h>` header file is one which is packaged with the operating system. It must be coordinated so that the same bit definitions which were used to build the kernel

are also the ones exposed to user-level programmers.

Standard I/O streams

In the UNIX model, three file descriptors are special. Descriptors 0, 1 and 2 are known as standard input, standard output and standard error, respectively. The analogous stdio names are `stdin`,`stdout`,`stderr`.

By default, when the program is started from a command-line shell environment, standard input refers to the keyboard and standard output and standard error the screen. Use of the shell redirection syntax (such as > or |) changes which file is associated with particular standard streams, and is one of the most basic building blocks of the power of the UNIX operating system. More will be said about the mechanisms of I/O redirection in a later unit.

The three standard streams (0,1,2) may be assumed to always be open when the program begins, and should generally not be closed by the program. Doing so would defeat the intentions of the invoking user. However, we will see examples later of instances where it is necessary and proper to close and/or re-open these streams.

Error reporting

When a UNIX system call fails, it returns an out-of-the-ordinary value to indicate its failure, and sets a **global** (ick ick ick) integer variable called `errno` to indicate the nature of the error. As there is only one such variable, subsequent failed system calls will overwrite the error information of earlier calls.

Note: `errno` is only set when a system call fails. Since it is a global integer, the initial value is 0, which is "No error." If a particular system call fails, it will set `errno`. Subsequent system calls that do not fail will **not** overwrite that value. This global variable model becomes a problem with multi-threaded programming, but that is far beyond the scope of this first unit.

A program should always check for errors. It can attempt to take some action based on the specific error code:

```
#include <errno.h>          /* Defines error numbers */
#include <fcntl.h>          /* For O_RDONLY def */

int fd;
char *fname;

          if  ((fd=open(fname,O_RDONLY))<0)
          {
 /* The code below is meant as an example of making a decision based
          on errno.  Don't blindly copy it */
```

```
                if (errno == ENOENT)
                        fprintf(stderr,"Sorry, %s does not appear to be
                                the name of a valid file0,fname);
                else
                        fprintf(stderr,"Can't open file %s for reading:%s0,
                                fname,strerror(errno));
                return -1;
        }
```

In the above example, we see the use of the symbolic constant `ENOENT` to refer to the specific error ("No such file or directory"). We also see the use of the standard library function `strerror` which returns the error description. This illustrates an important point in error handling and reporting: Report as much information as possible! Don't hide information that might be useful in determining the reason for a problem. Once that information is discarded, it is lost forever. At the very least, for failed system calls, provide:

The operation that was being attempted ("open")
The object being operated on (the file name)
Other relevant parameters ("for reading")
The error returned (strerror(errno))

The textual error message which is generated by `strerror`, `perror` and the like, may vary from system to system or may not even be in the English language if "localization" is turned on. The assignment of specific `errno` integers to specific errors is also subject to change. The symbolic error names, such as `ENOENT`, however, are part of the standardized UNIX programming interface. Programmers should always use these when making decisions based on specific error results. The `man` pages document which error codes may be expected for a given system call (although beware, sometimes the documentation is not exhaustive). The Linux kernel defines over 100 error codes. As an example, the open(2) system call can return more than a dozen different error codes.

Although the example above illustrates looking at `errno` to make error-specific decisions, generally that is not necessary. In most cases, system calls should be checked for error and the error should be reported with `strerror`, etc.

Reading and Writing

The system call provided to read data from a file is called, oddly enough, `read`. Here is an example:

```
char buf[4096];
int j,lim;
        lim=sizeof buf;
        j=read(fd,buf,lim);
```

The first parameter to read is an open file descriptor, The second is a buffer (an array of characters) into which the data will be read. Sometimes the buffer is a static array, as above, sometimes the buffer has been dynamically allocated with malloc, but it better be something that actually has some memory associated with it! Finally, the third argument is a limit to the number of bytes to read into the buffer. Obviously, the buffer should be at least as large as this limit. The operating system will never try to write more than that number of bytes into the buffer. Note that the result is NOT terminated by a NUL (\0) character.

We can classify the return value from read into 3 categories:

j<0: An error has occurred.
j==0: The end of file has been reached (see discussion below)
0<j<=lim: j bytes of data have been read into the buffer.

The converse of read is write:

```
char buf[4096];
int n,j;

        strcpy(buf,"test");
        n=strlen(buf);
        j=write(fd,buf,n);
```

`write` attempts to write the `n` bytes residing at the address `buf` to the file `fd`. The return value falls into 4 categories:

j<0: An error occurred. The data were not written.
j==0: No data were written. This is almost always an error and can be treated the same as a negative result. Under some circumstances, such as non-blocking I/O on devices or network connections, this condition is a transient one and the write can be re-tried. This is beyond the scope of this introduction.
j==n: All of the data were written.
0<j<n: The first j bytes of data were written successfully.

The last case is an interesting one, and one in which many programmers make mistakes. This so-called "short write" can happen for a variety of reasons, which we shall explore in future units. It doesn't indicate a "hard error," but it *does* mean that bytes j .. n-1 were **not** written. If you overlook this condition and only check for write returning a negative value, you will potentially lose data. On the other hand, if you over-react and treat this short-write as a system call error, you'll find that there is no error, and you will terminate your efforts prematurely. Proper recovery is to re-try the write for the remaining n-j bytes starting at buf+j.

Just a bunch of bytes

The UNIX file model may be said to be "just a bunch of bytes". The UNIX kernel does not impose any kind of typing or structure on files. Every file is created and treated equally as an arbitrary and *opaque*(meaning that UNIX doesn't care what the contents are) sequence of bytes. There are no special characters; each of the 256 possible byte values is of equal stature. In particular, note that the NUL character (`'\0'`) has no special significance. Although it is used by C as a string terminator, it is neither needed nor noticed by read or write.

Also, there is no EOF (End Of File) character! The UNIX kernel knows where the end of a file is because it knows exactly how many bytes are in the file. By default, reads and writes are sequential. The kernel maintains a record for each file descriptor of where the last read or write left off, and the next one picks up there. A write beyond the existing end of file simply grows the file. A read past end of file returns 0 (after all, there are 0 bytes available to be read).

The current read/write position in an open file can be queried or reset through the `lseek` system call. Thus a file can be accessed either sequentially or randomly. The position is 0 when the file is first opened, although as a further wrinkle, when a file has been opened with the **O_APPEND** bit flag, all writes include an implicit and automatic seek to the current end of file at the moment of the write. This mode is often used for log files to ensure that data can never be overwritten.

When reading from the terminal, as is the case by default when reading from standard input, the event of hitting the end of file can be simulated when the user hits Control-D at the beginning of the line. This causes read to return 0 when it reaches this point. Subsequent reads will not continue to return 0, as would be the case when reading from an actual file, so it is possible to input multiple "files" from the keyboard. It is important to note that the actual Control-D character is NOT part of the input stream and will never be seen by the program. The fact that Control-D is the keyboard end-of-file is an arbitrary convention. Any character can be made the EOF character (read the man page `stty(1)`), often with amusing results.

The stdio library

The C programming language was designed as the first "portable" high-level language. This was a great challenge considering the vast collection of operating systems and machine architectures that existed, far more than have survived into common usage in the present day. Each operating system could be counted on to have its own way of doing what is a very universal function: input and output of files.

The Standard I/O library (stdio) is a wrapper designed to isolate the programmer from these details and allow hir to write code which can be readily compiled and executed on a multitude of systems. Underneath, the wrapper makes OS-specific system calls to accomplish its job. It also provides some "value-added" features such as error handling

and buffering.

Other libraries have been layered on top of the stdio library, most notably the `iostream` object library of the C++ run-time model.

It is assumed here that the reader is fully conversant in C programming with the stdio library and so no explanation of matters such as `printf` syntax will be given.

Under UNIX, the stdio library makes use of the UNIX file I/O system calls presented below.

Buffering and the stdio library

Buffering is a key concept in operating systems design. The principle is simple: given that you've done something, you're likely to do something very similar very soon. One of the most common uses of buffering is in the stdio library. Consider a program to change all linefeeds to <CR><LF> (in other words, converting from UNIX to DOS EOL conventions):

```
main()
{
char buf[1];
int n;
          while ((n=read(0,buf,1))>0)
          {
                    if (buf[0]=='\n')
                              write(1,"\r",1);
                    write(1,buf,1);
          }
}
```

The equivalent program using stdio runs faster: (**NOTE: Both versions of the program are seriously flawed in that they fail to check for or report errors !!!**)

```
#include <stdio.h>

main()
{
int c;
          while ((c=getc(stdin))!=EOF)
          {
                    if (c=='\n')
                              putc('\r',stdout);
                    putc(c,stdout);
          }
}
```

When you use the stdio functions such as putc and getc, or printf (which is internally built on a variant of putc), you are not actually making a system call. The `FILE *` is a pointer to a struct which is allocated by fopen. This struct contains a pointer to a buffer,

which is also allocated at user-level by fopen, and other related fields such as a count of the characters in the buffer. putc copies the supplied single character into the next space in the buffer. When the buffer is full, only then is the `write` system call made.

Let's say the buffer is 4,096 bytes (a common choice). Now instead of making 4,096 system calls, one character at a time, just one will be made. This saves the overhead of making a system call, which as we will see can involve a long code path with a lot of instructions. Of course, if the buffer is very large, there will be relatively little gain in economy, and the price will be excessive memory usage. The default buffer size is tuned to the optimal performance in most cases. For most programs, this layer of isolation is ideal. It allows the programmer to separate what he or she is trying to do from the low-level implementation details. In some cases, especially with programs that do system-level things (like formatting disks) this isolation gets in the way, and it is necessary to use the system calls directly.

In the case of input, a read system call is only made when the buffer is empty. Thereafter, functions such as getc, fgets, scanf, etc. simply copy the characters out of the buffer, until it gets empty again.

Sometimes we don't want to wait for an entire buffer before reading/writing. The stdio library supports 3 output buffering modes:

- _IONBF: Unbuffered. Data are written immediately. This is the default mode for stderr--you really want to see your errors as soon as they happen.
- _IOFBF: File buffering. Buffer is flushed when it is full.
- _IOLBF: Line buffering. Buffer is flushed when an end-of-line character is written. This is intended for terminals, and is the default for stdin and stdout.

• The `setvbuf` library call can be used to adjust the buffering mode of a FILE, and the size of the buffer. `fflush` can be used to force a buffer flush.