Chapter: 1. UNIX and ANSI Standards

1.1 Introduction

System Software

It performs the basic functions necessary to start and operate a computer. It controls and monitors the various activities and resources of a computer and makes it easier and more efficient way to use the computer.

System software can be classified into three categories.

1. System Control Software: Programs that manages system resources and functions.
2. System Support Software: Programs that supports the execution of various application.
3. System Development Software: Programs that assist system devalopers in designing and developing information.

System Control Software includes programs, that monitor, control and Co-ordinate systems and manage the resources and functions of a computer system. The most important System Control Software is Operating System.

System Support Software is a software that supports and facilitates, the smooth and efficient operation of a computer. There are 4 major categories of System Support Softwares such as utility softwares, language translator, database management system and performance statistics softwares.

System Development Software helps system developers to design and bulid better system.

Systems Programming

The development and maintenance of operating system software or we can say the development of above said different types of systems softwars.

Standards

The ISO (International Standards Organization) defines “standards are documented agreements containing technical specifications or other precise criteria to be used consistently as rules, guidelines or definitions of characteristics to ensure that materials, products, processes and services are fit for their purpose”.

Definition or format that has been approved by a recognized standards organization or by industry. Standards exist for Programming Languages, Operating Systems, Data Formats, Communication Protocols, and Electrical Interfaces.

From a user's point of view, standards are extremely important in the computer industry because they allow the combination of products from different manufacturers to create a customized system. Without standards, only hardware and software from the same company could be used together. In addition, standard user interfaces can make it much easier to learn how to use new applications.

Most official computer standards are set by one of the following organizations:

- ANSI (American National Standards Institute)
- ITU (International Telecommunication Union)
- IEEE (Institute of Electrical and Electronic Engineers)
- ISO (International Standards Organization)
- VESA (Video Electronics Standards Association)

1.2 The ANSI C Standard

This standard was proposed by American National Standards Institute (ANSI) in the year 1989 for C programming Language standard called X3.159-1989 to standardize the C programming language constructs and libraries.

1.2.1 Major differences between ANSI C and K & R C

- ANSI C supports Function Prototyping
- ANSI C support of the const & volatile data type qualifier
- ANSI C support wide characters and internationalization, Defines setlocale function
- ANSI C permits function pointers to be used without dereferencing
- ANSI C defines a set of preprocessor symbols

1.2.1.1 Function Declaration or Function Prototype

ANSI C supports Function Prototype of C++ language. Proto-type or function declaration is an interface to the compiler by giving details such as the number, type of arguments and the type of return values.

It helps to compilers to check for function calls in user programs that pass invalid numbers of arguments or compatible argument data types. These is the major weekness of K& R C (Brian Kernighan and Dennis Ritche)compilers: Because the invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

Syntax: **return_type function_name(argument list);**

- **return_type :** The return type is data type of the value that is return or sent from the function.
- **function_name:** A function should be given a descriptive name. In general, the same rules that apply to variable names also apply to the function name.
- **argument List :** This list contains the type and names of variables that must be passed to the function.
  for example: int large(int x, int y);

In a function declaration, the names of the variables are dummy variables & there fore, they are optional so that above declaration can be rewritten as follows.

```
int  large( int , int );
```

For functions that take a variable number of arguments, their declaration should have "…" specified as the last argument to each functions.

For example: `int  large( int  x, int y, … );`

1.2.1.2 "const" Qualifier , const Arguments and "volatile" qualifier

The const type qualifier is similar to C-style #define macro. The value attached to the const qualifier can not be changed during the program execution. Instead of using literal values like 100 or 200(assume that it occurs at 100 places in the program) in applications involving arrays, say, you may use constant names which would be more readable. Also whenever there is any change made to this say from 100 to 150 or from 200 to 300 you could do this easily by changing at one place.

Example:

```
for (int i=0;i<100;i++)
{
        //not a good method to use 100
}
```

Instead of the above method,

```
const int MAX=100;
 for (int i=0;i<MAX;i++)
{
         // more readable
}
```

The value of constant variable cannot be changed. Example: MAX=200; //Wrong

In C++, an argument to a function can be declared as const as shown below.

```
int  strlen(const char *p);
int  length(const string &s);
```

The qualifier *const* tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

The "volatile" qualifier

A variable is declared as *volatile* when its value can possibly be changed in ways outside either the control or detection of the compiler (with out the knowledge of compiler).

Example: A variable updated by the system clock, it giving hint to the compiler's optimization algorithm not to remove any redundant statements.

The volatile qualifier is used in much the same way as is the const qualifier.

```
Example1:      volatile int display;

Example2:      char getio( )
               {
                        volatile  char*  ioport = 0x7777;
                        char  ch  = *ioport;                //reads first byte of data
                              ch  = *ioport;               //reads second byte of data
               }
```

In the above example, if the *ioport* variable is not to be declared to be "volatile" when the program is compiled, the compiler may eliminate the second ch=ioport statement as it is considered redundant with respect to the previous statement.

1.2.1.3 ANSI C support wide characters and internationalization, setlocale Function

ANSI C supports internationalization by allowing C progrms to use wide characters use more than one byte of storage per character. These are used in the countries where the ASCII character set is not the standard. For example, the Korean character requires two bytes per character.

ANSI C also defines ***setlocale*** function, which allows users to specify the format of date, monetary, and real number representation.

For example most of the countries display the date in "day/month/year" format, whereas the US displays the date in "month/day/year" format.
The function **setlocale** and its arguments are defined in the header **locale.h** header.

```
#include <locale.h>

char *setlocale(int  category,  char *locale);
```

Possible values for the locale argument of *setlocale* are C, POSIX, en_US locales refers to the UNIX, POSIX and US locales.

Possible values for the category argument of *setlocale* are:

Category Value	Effect on standard C function/Macros
LC_CTYPE	Affects the behavior of the <ctype.h> macros.
LC_TIME	This category applies to formatting date and time values.
LC_MONETARY	This category applies to formatting monetary.
LC_NUMERIC	Affects the number representation formats via *printf* and *scanf* functions.
LC_ALL	Combines the effects of all the above.

Return Value:

On success, *setlocale* returns a string to indicate the locale that was in effect prior to invoking the function.

On error, *setlocale* returns a NULL pointer.

Example:
```
#include <locale.h>
#include <stdio.h>
 main ( )
{
   char *old_locale;
   old_locale = setlocale(LC_ALL,"C");
   printf("Old locale was %s\n",old_locale);
}
```
The above program prints the output: Old locale was C.

1.2.1.4 ANSI C permits function pointers to be used without dereferencing

C++ specifies that a function pointer may be used like a function name. No dereference is needed when calling a function whose address is contained in the pointer. For example, the following statements define a function pointer *funptr*, which contains the address of the function ***fun***;

```
void   fun (float xyz, const char  * );
void  (*funptr)(float, const char *)=fun;
```

The function *fun* may be invoked by either directly calling *fun* or via the *funptr*. The following two functions are functionally equivalent:

```
fun (20.5,”Dreams”);
funptr(20.5, ”Dreams“);
```

But in K & R C requires *funptr* be dereferenced to call *fun*, thus an equivalent statement to the above, using K & R C syntax is:

```
(*funptr)(20.5, “Dreams”);
```

1.2.1.5 ANSI C defines a set of preprocessor symbols

ANSI C defines a set of *cpp* (C preprocessor) symbols which may be used in user programs. These symbles are assigned actual values at compile time.

***cpp* symbol**	**Use**
__DATE__	String literal, Value is the Date that a module containing this symbol is compiled.
__FILE__	String literal, Value is the Name of the current file being processed.
__LINE__	Decimal constant, Number of the current source file line being processed.
__STDC__	Constant, if value is 1 if a compiler is ANSI C conforming, 0 otherwise.
__TIME__	String literal, value is the Time that a module containing this symbol is compiled.

The following *test.cpp* program illustrates the uses of these symbols.

```
#include <iostream.h>
#include <stdio.h>
int main()
{
    #ifdef  __STDC__ == 0
          printf("cc is not ANSI C compliant\n");
    #else
          printf(" %s compiled at %s:%s. This statement is at line %d\n",
                                          __FILE__, __DATE__, __TIME__, __LINE__);
    #endif
          return 0;
  }
```

Sample output of the program is:

test.cpp compiled at DEC 17 2005:11:40:10. This statement is at line 11
Above output indicates that C++ supports only __FILE__, __DATE__, __TIME__, __LINE__, but not __STDC__.

1.3 The ANSI / ISO C++ Standard

The C++ language is one of the Object Oriented Programming (OOP) languages. It was developed by Bjarne Stroustrup at At &T Bell Laboratories in Murray Hill, New Jersey, USA, in the early eighties. C++ is an extension of C with a major addition of the class construct features of Simula 67. Since the class was a major addition to the original C language, Stroustrup called the new language "**C with Classes",** However, later in 1983 the name were changed to C++ the idea of C++ comes from C increment operator ++. Therefore C++ is an incremented version of C. The three most important facilities that C++ adds on to C are classes, function overloading, & operator overloading.

In 1989, Bjarne Stroustrup published "*The Annotated C++ Reference Manual"* , this manual become the base for the draft ANSI C++ standard, as developed by the X3I16 committee of ANSI. At 1990s the WG21 committee of the ISO joined the ANSI X3J16 committee to develop a unify ANSI/ISO C++ standard. A draft version of ANSI/ISO standard was published in 1994.

1.3.1 Major Differences between ANSI and C++

- **Function Declaration or Function Prototype**

  C++ requires tht all functions must be declared and defined before they can be referenced. But ANSI C supports C++ type as well as K & R C function declaration means function can be referenced before their declaration or defination in the function.

- **Functions that take a variable number of arguments**

  ```
  int  large(  );
  ```

  The above declaration in C can be called with exactly NULL actual arguments mean without any arguments.

But in ANSI the above declaration can be re written as:

int **large(…)** ;

Which means *large* function can be called with any number of actual arguments.

But in C++ the above declaration can be treated as:

int **large**(void);

Which means *large function* may not accept any argument when it is called.

- **Type safe linkage , Linkage Directives**

C++ encrypts external function names for type safe linkage this ensures that external function which is incorrectly declared and referenced in a module will cause the link editor(/bin/ld) to report undefined function name but ANSI C does not support the type safe linkage technique.

All C++ compilers support both C and C++ linkage. Linkage directives are used when we wish to call a function which is written in another programming language say "C". Complier must be told that, different requirements apply when the function is called.

Through linkage directives a programmer indicates to the complier that a function is written in a different programming language.
A linkage directive can have two forms:

- It can be in a single statement form.

```
extern "C" void exit( int );
```

- A compound statement form.

```
extern  "C" {
            int  printf("hello");
            int  printf("how are you ");
        }
```

1.4 POSIX Standards

POSIX is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software designed to run on variants of the UNIX Operating System. They are formally designated as IEEE 1003 and the international standard name is ISO / IEC 9945. POSIX is a acronym for Portable Operating System Interface. There are three subgroups in POSIX they are POSIX.1, POSIX.1b and POSIX.1c.

- POSIX.1 committee proposes a stdandard for base operating system APIs. This stadndard is formally known as the IEEE standard 1003.1-1990. This standard specifies the APIs for the file manipulation and processes (for Process Creation and Control).

- POSIX.1b committee proposes a stdandard for real time operating system APIs. This stadndard is formally known as the IEEE standard 1003.4-1993 This standard specifies the APIs for the interprocess communication (Semaphores, Message Passing , Shared Memory).

- POSIX.1c committee proposes a stdandard for multithreaded programming interface. This standard specifies the APIs for Thread Creation, Control, and Cleanup, Thread Scheduling ,Thread Synchronization and for Signal Handling .

To ensure a user program conforms to the POSIX.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each program(before the inclusion of any header files) as:

```
 #define   _ POSIX_SOURCE
```

or specify the –D_ POSIX_SOURCE option to a C++ compiler during compilation.

```
$g++   –D_ POSIX_SOURCE  filename.cpp
```

This manifested constant is used by cpp to filterout all non-POSIX.1 and non-ANSI C standard codes(example : functions, data types and manifested constants) from headers used by the user program.

POSIX.1b defines different manifested constant to check conformance of user programs to thatstandard. The new macro is _POSIX_C_SOURCE and its value is a time stamp indicating the POSIX version to which a user program conforms. The possible values of the _POSIX_C_SOURCE macro are:

_POSIX_C_SOURCE value	**Meaning**
199808L	First version of POSIX.1 compliance
199009L	Second version of POSIX.1 compliance
199309L	POSIX.1 and POSIX.1b complience

Each _POSIX_C_SOURCE value consists of the year and month that a POSIX standard was approved by IEEE as a standard. The L suffix in a value indicates that the value ‘s data type is a long integer.

In general a user program that must be strictly POSIX.1and POSIX.1b compliant may be written as follows:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L

#include <iostream.h>
#include <unistd.h>

int main( )
{
   ....
 }
```

There is also one more constant which will tells you about the POSIX version to which your system conforms. That constant defined in the header *unistd.h*. The following program checks and displays the _POSIX_VERSION constant of the system on which it is run.

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L

#include <iostream.h>
#include <unistd.h>

int main( )
{
        #ifdef _POSIX_VERSION
           cout << "System conforms to POSIX: " << _POSIX_VERSION << endl;
        #else
           cout << "_POSIX_VERSION is undefined\n";
        #endif
           return 0;
  }
```

1.4.1 POSIX Feature Test Macros

Some UNIX features are optional to be implemented on a POSIX-conforming system. Thus, POSIX.1 defines a set of feature test macros, which, if defined on a system has implemented the corresponding features.

Feature Test Macro	Effects if defined on a System
_POSIX_JOB_CONTROL	It allow us to start multiple jobs(groups of processes) from a single terminal and control which jobs can access the terminal and which jobs are to run in the background. Hence It supports BSD version Job Control Feature.
_POSIX_SAVED_IDS	Each process running on the system keeps the saved set-UID and set-GID, so that it can change effective user ID and group ID to those values via *setuid* and *setgid* APIs respectively.
_POSIX_CHOWN_RESTRICTED	If the defined value is -1, users may change ownership of files owned by them. Otherwise only users with special previlege may change ownership of any files on a system.
_POSIX_NO_TRUNC	If the defined value is -1, any long path name passed to an API is silently truncated to NAME_MAX bytes, otherwise error is generated.
_POSIX_VDISABLE	If the defined value is -1, there is no disabling character for special characters for all terminal device files, otherwise the value is the disabling character value.

The following program prints the POSIX defined configuration options supported on any system:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE     199309L

#include <iostream.h>
#include <unistd.h>
```

```
int main( )
{
        #ifdef _POSIX_JOB_CONTROL
           cout << "System supports job control\n";
        #else
           cout << "System does not support job control\n";
        #endif

        #ifdef _POSIX_SAVED_IDS
           cout << "System supports saved set-UID and saved set-GID\n";
        #else
           cout << "System does not support saved set-UID and saved set-GID\n";
        #endif

        #ifdef _POSIX_CHOWN_RESTRICTED
           cout << "chown restricted option is: " <<     _POSIX_CHOWN_RESTRICTED
                 << endl;
        #else
           cout << "System does not support system-wide chown_restricted option\n";
        #endif

        #ifdef _POSIX_NO_TRUNC
           cout << "Pathname truncation option is: " <<  _POSIX_NO_TRUNC << endl;
        #else
           cout << "System does not support system-wide pathname truncation option\n";
        #endif

        #ifdef _POSIX_VDISABLE
           cout << "Disable character for terminal files is: " << _POSIX_VDISABLE
                 << endl;
        #else
           cout << "System does not support _POSIX_VDISABLE\n";
        #endif

        return 0;
   }
```

```
OutPut:

        System supports job control
        System supports saved set-UID and saved set-GID
        chown restricted option is: 1
        Pathname truncation option is: 1
        Disable character for terminal files is:
```

1.4.2 Limits Checking at Compile Time and at Run Time

The POSIX.1 and POSIX.1b standards specify a number of parameters that describe capacity limitations of the system. These limits can be fixed constants for a given operating system, or they can vary from machine to machine. For example, some limit values may be configurable by the system administrator, either at run time or by rebuilding the kernel, and this should not require recompiling application programs.

Each of the following limit parameters has a macro that is defined in `<limits.h>`. Each of these parameters also has another macro, with a name starting with `` `_POSIX' ``, which gives the lowest value that the limit is allowed to have on *any* POSIX system.

Compile Time Limit	**Minimum Value**	**Meaning**
_POSIX_ARG_MAX	4096	The maximum combined length of the *argv* and *environ* arguments that can be passed to the `exec` functions.
_POSIX_CHILD_MAX	6	For the maximum number of simultaneous processes per real user ID.
_POSIX_NGROUPS_MAX	0	The maximum number of supplementary group IDs per process.
_POSIX_OPEN_MAX	16	The maximum numbers of files that a single process can have open simultaneously.
_POSIX_STREAM_MAX	8	The maximum numbers of streams that a single process can have open simultaneously.
_POSIX_LINK_MAX	8	Maximum number of links file may have. Thus, you can always make up to eight names for a file without running into a system limit.
_POSIX_TZNAME_MAX	3	The maximum length of a time zone name.
_POSIX_SSIZE_MAX	32767	The maximum value that can be stored in an object of type `ssize_t`.
_POSIX_MAX_CANON	255	The maximum number of bytes in a canonical input line from a terminal device.
_POSIX_MAX_INPUT	255	The maximum number of bytes in a terminal device input queue
_POSIX_NAME_MAX	14	The maximum number of bytes in a file name component.
_POSIX_PATH_MAX	256	The maximum number of bytes in a file name
_POSIX_PIPE_BUF	512	The maximum number of bytes that can be written atomically to a pipe
_POSIX_AIO_LISTIO_MAX	2	The maximum number of I/O operations that can be specified in a list I/O call.
_POSIX_AIO_MAX	1	The maximum number of outstanding asynchronous I/O operations.
_POSIX_TIMER_MAX	32	Maximum number of timers that can be used simultaneously by a process.
_POSIX_RTSIG_MAX	8	Maximum number of real time signal
_POSIX_SIGQUEUE_MAX	32	Maximum number of real time signals that a process may queue at any one time.
_POSIX_MQ_OPEN_MAX	2	Maximum number of message queues that may be accessed simultaneously per process.
_POSIX_MQ_PRIO_MAX	2	Maximum number of message priorities that can be assigned to messages.

1.4.3 Definition of `sysconf`

To find out the actual implemented configuration limits system wide or on individual objects during run time POSIX.1 defines the functions *sysconf*, *pathconf* and *fpathconf.*

```
#include <unistd.h>

long  sysconf (int  flimit_name) ;
long  fpathconf(int fildes, int flimit_name));
long  pathconf(const char *path, int  flimit_name);
```

The sysconf() is used to inquire about runtime system parameters. The *flimit_name* argument should be one of the `_SC_` symbols listed below.

The functions pathconf() and fpathconf() are used to find out the value that applies to any particular file.

For pathconf(), the *path* argument points to the pathname of a file or directory.
For fpathconf (), the *fildes* argument is an open file descriptor.

The *flimit_name* argument should be one of the `_PC_` constants listed below.

The normal return value from above functions is the value you requested. A value of `-1` is returned on error.

1.4.3.1 Constants for *sysconf* Parameters

Here are the symbolic constants for use as the *parameter* argument to `sysconf`. The values are all integer constants (more specifically, enumeration type values).

Limit Value	***sysconf* return value**
_SC_ARG_MAX	The maximum combined length of the *argv* and *environ* arguments that can be passed to the `exec` functions.
_SC_CHILD_MAX	For the maximum number of simultaneous processes per real user ID.
_SC_OPEN_MAX	The maximum numbers of files that a single process can have open simultaneously.
_SC_STREAM_MAX	The maximum numbers of streams that a single process can have open simultaneously.
_SC_NGROUPS_MAX	The maximum number of supplementary group IDs per process.
_SC_JOB_CONTROL	Inquire about the parameter corresponding to `_POSIX_JOB_CONTROL`.
_SC_SAVED_IDS	Inquire about the parameter corresponding to `_POSIX_SAVED_IDS`.
_SC_VERSION	Inquire about the parameter corresponding to `_POSIX_VERSION`.
_SC_CLK_TCK	Number of clock ticks per seconds.
_SC_TIMERS	Inquire about the parameter corresponding to `_POSIX_TIMERS`.
_SC_AIO_LISTIO_MAX	the maximum number of I/O operations that can be specified in a list I/O call
_SC_TIMER_MAX	`Maximum number of timers that can be used simultaneously by a process.`
_SC_MQ_OPEN_MAX	Maximum number of message queues that may be accessed

	simultaneously per process.
_SC_RTSIG_MAX	Maximum number of real time signal.
_SC_SIGQUEUE_MAX	Maximum number of real time signals that a process may queue at any one time.

Examples of *sysconf*

For example, here is how to test whether JOB CONTROL, OPEN MAX and CHILD MAX are supported:

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>

int main( )
{
   int value;
   if ((value=sysconf(_SC_OPEN_MAX))==-1)
      perror("sysconf");
   else
      cout << "OPEN_MAX: " << value << endl;
   if ((value=sysconf(_SC_JOB_CONTROL))==-1)
      perror("sysconf");
   else
     cout << "JOB_CONTROL: " << value << endl;
   if(( value = sysconf (_SC_CHILD_MAX))== -1)
        perror("sysconf");
   else
     cout << "CHILD_MAX: " << value << endl;
   return 0;
}
```

Output:

```
OPEN_MAX: 1024
JOB_CONTROL: 1
CHILD_MAX: 999
```

Here are the symbolic constants that you can use as the *flimit_name* argument to `pathconf` and `fpathconf`. The values are all integer constants.

Limit Value	***pathconf* return value**
_PC_LINK_MAX	Maximum number of links a file may have.
_PC_MAX_INPUT	Maximum capacity in bytes of a terminal input queue.
_PC_NAME_MAX	Maximum length in bytes, of a file name.
_PC_PATH_MAX	Maximum length in bytes, of a path name
_PC_PIPE_BUF	Maximum size of a block of data that may be automatically read from or written to a pipe file.
_PC_CHOWN_RESTRICTED	Inquire about the value of _POSIX_CHOWN_RESTRICTED.
_PC_NO_TRUNC	Inquire about the value of _POSIX_NO_TRUNC.
_PC_VDISABLE	Inquire about the value of _POSIX_VDISABLE

Example of *pathconf* and *fpathconf*

For example, here is how to test whether PATH MAX, CHOWN_RESTRICTED are supported:

```
#define  _POSIX_SOURCE
#define  _POSIX_C_SOURCE 199309L
#include  <stdio.h>
#include  <iostream.h>
#include  <unistd.h>

int main( )
{
   int value;

   if ((value=pathconf("/",_PC_PATH_MAX))==-1)
       perror("pathconf");
   else
      cout << "Max path name: " << (value+1) << endl;
   if ((value=fpathconf(0,_PC_CHOWN_RESTRICTED))==-1)
       perror("fpathconf");
   else
      cout << "chown_restricted for stdin: " << value << endl;
   return 0;
}
```

Output:

```
Max path name: 4097
chown_restricted for stdin: 1
```

1.4.4 Basic Definitions

File Discriptor

File discriptor are small nonegative integers that the kernel uses to identify the files being accessed by a particular process. Whenever kernel opens an exsisting file or create a new file , it returns a file discriptor that we can use when we want to read / write to the file.

By default all shells opens three discriptors whenever a new program is run. These are standard input that is 0, the standard output that is 1, and the standard error that
is 2.

Path Name

A sequence of zero or more filenames separated by slashes and optionally starting with a slash (/) character forms a path name. A path name that begins with a slash called an absolute pathname (Example: */home/chandu/reva/cse/xyz.c*), otherwise it called as relative path name (Example: $cd .. , $ ls . , $ cat cse/xyz.c).

Primitive System Data Types

The datatypes that ends in _t are called primitive system data types. They are usually defined in the header <sys/types.h>. They are usually defined with the C typedef declaration. The purpose is to prevent programs from using specific data types (such as int, short or Long) to allow each implementation to choose which data type is required for a particular system. For example everywhere we need to store a process id we'll allocate a variable of type *pid_t.*

1.5 The POSIX.1 FIPS Standard

FIPS stands for Federal Information Processing Standard. This standard was developed by National Institute of Standards and Technology formarly known as National Bureau of Standards. The latest version of this standrd, FIPS 151-1, is based on the POSIX.1- 1998 standard. Specifically, the FIPS standard is a restriction of the POSIX.1-1998 standard, Thus a FIPS 151-1 conforming system is also POSIX.1-1998 conforming , but not vice versa.

The following features to be implemented in all FIPS conforming systems:

_POSIX_JOB_CONTROL	_POSIX_JOB_CONTROL must be defined.
_POSIX_SAVED_IDS	_POSIX_SAVED_IDS must be defined.
_POSIX_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED must be defined and its value is not -1, it means users with special previlege may change ownership of any files on a system.
_POSIX_VDISABLE	POSIX_VDISABLE must be defined and its value is not -1.
_POSIX_NO_TRUNC	Must be defined and its value is not -1, Long path name is not support.
NGROUP_MAX	Symbol’s value must be at least 8.
The read and write API should return the number of bytes that have been transferred after the APIs have been interrupted by signals.	
The group ID of a newly created file must inherit the group ID of its containing directory.	

1.6 XPG (The X/Open Portability Guide) / The X/Open Standard

The X/Open Portability Guide, published by the X/Open Company, Ltd., is a more general standard than POSIX. X/Open owns the Unix copyright and the XPG specifies the requirements for systems which are intended to be a Unix system.

The GNU C library complies to the X/Open Portability Guide, Issue 4.2, with all extensions common to XSI (X/Open System Interface) compliant systems and also all X/Open UNIX extensions.

The additions on top of POSIX are mainly derived from functionality available in System V and BSD systems. Some of the really bad mistakes in System V systems were corrected, though. Since fulfilling the XPG standard with the Unix extensions is a precondition for getting the Unix brand chances are good that the functionality is available on commercial systems.

1.7 Most frequently asked questions

1. What are the major differences between ANSI C and K & R C. Explain each with an example?

2. What is POSIX standards and explain different subset of POSIX standards.

3. What is POSIX standard? Give the structure of the program to filter out non-POSIX Compliant codes from a user program.

4. Write a C++ program that prints the POSIX defined configuration options supported on any given system using Feature Test Macros.

5. Write a C/C++ POSIX compliant program to check the following limits

1) Maximum Path Length
2) Maximum Characters in a File Name
3) Maximum open files per Process
4) Clock ticks

6. Write a C/C++ POSIX compliant program to check the following limits.

1) Maximum number of Child Process
2) Maximum File Name
3) Maximum number of Files can be Opened
4) Clock ticks

7. Write a note on POSIX Feature Test Macros

8. What are the different features implemented in all FIPS conforming systems?

Chapter 2. UNIX and POSIX APIs

2.1 Introduction

UNIX system provides a common set of APIs commonly known as system calls, which may be called by users' programs to perform system specific functions and also to directly manipulate systems objects such as files and processes.

Defines Common set of APIs for performing the followings:

- To Determine System Configuration and User information : sysconf, pathconf and fpathconf APIs.
- For File Manipulation : read, write, chown, link and utime etc.
- For Process Creation and Control : fork, vfork, wait and waitpid etc.
- Inter-Process Communication : APIs of messages, Shared memory, memory map and semaphores.
- Network Communication : APIs of socket programming.

2.2 API

This is an abbreviation for "Application Programming Interface," by which an Application program (a complete program that performs a specific function directly for the user) can access the computer's operating system.

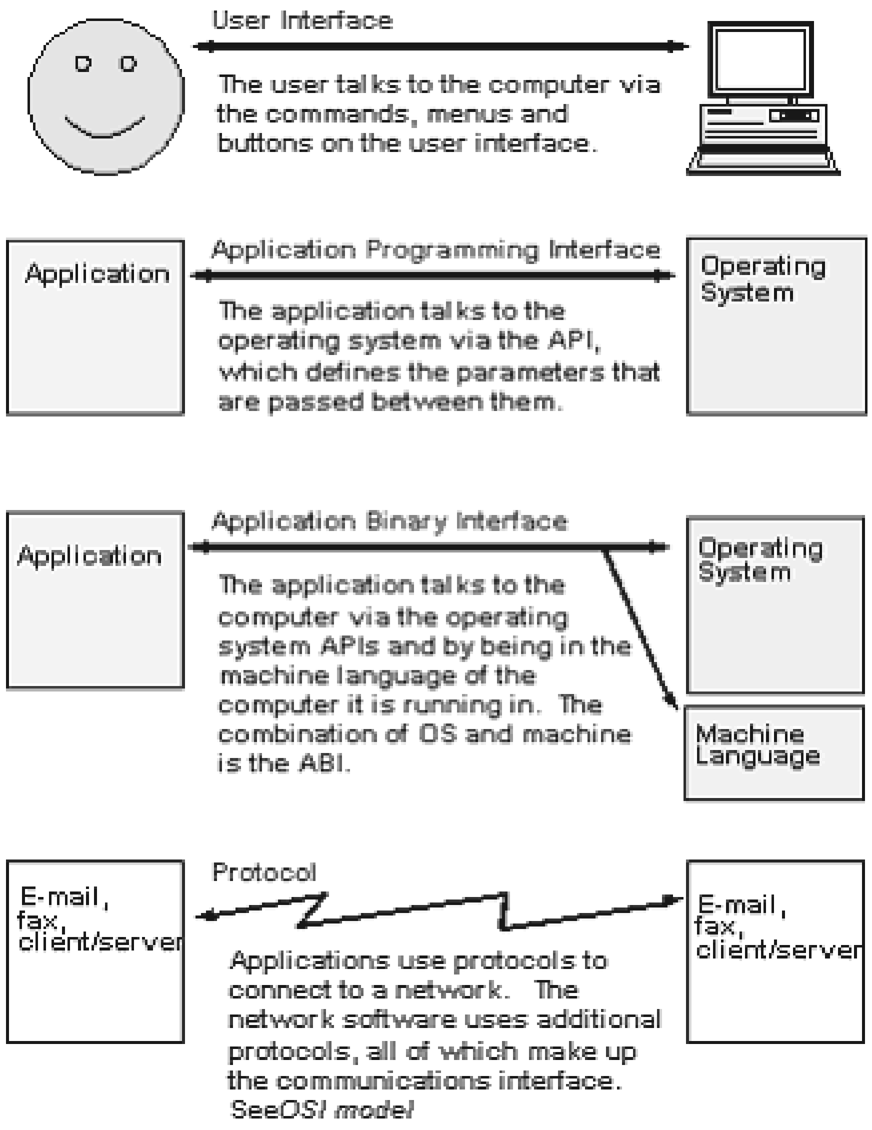
"Application Program Interface" - A series of software routines and development tools that comprise an interface between computer applications and lower level services and functions (e.g. the operating system, device drivers, and other low-level software). Hence APIs serve as building blocks for programmers putting together software applications.

An API is a set of instructions or rules that enable two operating systems or software applications to communicate or interface together.

Application Programming Interface is a language and message format used by an application program to communicate with the operating system or some other control program such as a database management system (DBMS) or communications protocol.

APIs are implemented by writing function calls in the program, which provide the linkage to the required subroutine for execution. Thus, an API implies that some program module is available in the computer to perform the operation or that it must be linked into the existing program to perform the tasks.

An application-programming interface (API) is a set of definitions of the ways one piece of computer software communicates with another. It is a method of achieving abstraction, usually (but not necessarily) between lower-level and higher-level software.


User Interface
The user talks to the computer via the commands, menus and buttons on the user interface.
Application
Application Programming Interface
Operating System
The application talks to the operating system via the API, which defines the parameters that are passed between them.
Application
Application Binary Interface
Operating System
The application talks to the computer via the operating system APIs and by being in the machine language of the computer it is running in. The combination of OS and machine is the ABI.
Machine Language
E-mail, fax, client/server
Protocol
E-mail, fax, client/server
Applications use protocols to connect to a network. The network software uses additional protocols, all of which make up the communications interface. See OSI model

2.3 System calls and Library calls

All operating systems provide service points through which programs request services from the kernel. All versions of UNIX provides a well defined limited number of entry points directly into the kernel are called system calls(APIs), that user cannot change. These are built into the kernel.

The functions are not entry points to the kernel, although they may invoke one or more of the kernel's system calls.
For example the *printf* function may invoke the *write* system call to display the output.

An application can call either a system call or a library function. Also some library functions invoke a system calls as shown in figure.

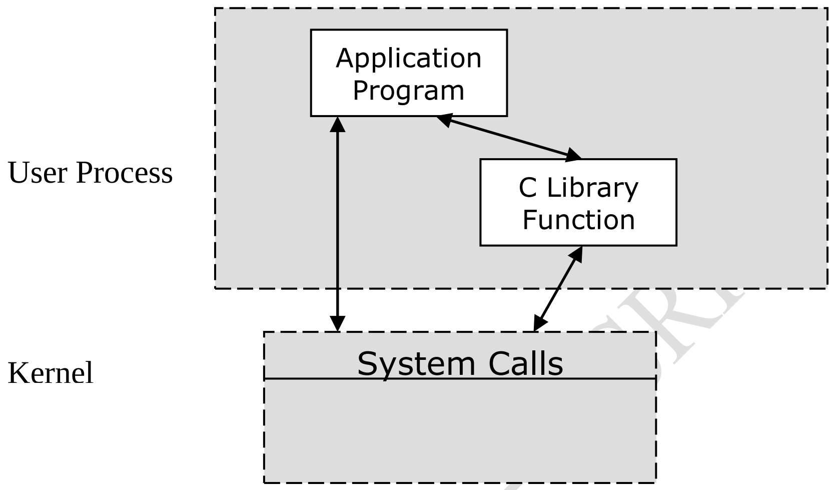


Figure: Difference between C Library function and System Calls

System calls usually provide a minimal interface while library functions often provide more elaborated functionality.

From an implementers point of view the distinction between a system call and a library function is fundamental. But from a users' point of view the difference is not as critical. So both system calls and library functions appear as normal C functions.

System calls	Library calls
Executed by the operating system	Executed in the user program
Perform simple single operations	May perform several tasks
Built into kernel	May call System calls

2.4 Context Switching

Most Unix APIs access their Unix kernel's internal resources, thus when one of these APIs is invoked by a process, the execution context of the process is switched by the kernel from a *user mode* to *kernel mode*.

A *user mode* is the normal execution context of any user process, and it allows the process to access its specific data only.
A *kernel mode* is the protective execution environment that allows a user process to access kernels data in a restricted manner.

When the APIs execution completes, the user process is switched back to the user mode. This context switching for each API call ensures that process access kernels data in a controlled manner and minimizes any chance of a run way user application may damage an entire system. So in general calling an APIs is more time consuming than calling a user function due to the context switching. Thus for those time critical applications, user should call their system APIs only if it is absolutely necessary.

2.5 The POSIX APIs

Most POSIX.1 and POSIX .1b APIs are derived from UNIX APIs. The POSIX.1b committee creates a new set of APIs for Inter Process Communication using messages, shared memory and semaphores.

In general POSIX's APIs uses and behaviors are similar to those of UNIX APIs. However users' programs should define the _POSIX_SOURCE for POSIX.1 APIs and _POSIX_C_SOURCE for both POSIX.1 and POSIX.1b APIs.

2.6 An APIs common Characteristics

All most all APIs returns an integer value which indicates termination status of their execution. Specifically an APIs returns -1 value, it means APIs execution has failed and the global variable *errno* (it is declared in *errno.h* header) is set with an error code.

An user process may call the *perror* or *sterror* function to print a diagnostic message of the failure.

If an API execution is successful, it returns either zero or a pointer to some record where user requested information is stored.

Some error statuses that may be assigned to *errno* by any API are defined in the <errno.h> header. Following table shows Some Error Codes and their meaning:

Errors	Meaning
EPERM	API was aborted because the calling process does not have the super user privilege.
EINTR	An APIs execution was aborted due to signal interruption.
EIO	An Input/Output error occurred in an APIs execution.
ENOEXEC	A process could not execute program via one of the Exec API.
BADF	An API was called with an invalid file descriptor.
ECHILD	A process does not have any child process which it can wait on.
EAGAIN	An API was aborted because some system resource it is requested was temporarily unavailable. The API should call again later.
ENOMEM	An API was aborted because it could not allocate dynamic memory.
EACCESS	The process does not have enough privilege to perform the operation.
EFAULT	A pointer points to an invalid address.
EPIPE	An API attempted to write data to a pipe which has no reader.
ENOENT	An invalid file name was specified to an API.

Most frequently asked questions

1. Explain common characteristics of an API with an example.
2. What is an API? How they are different from the 'C' Library functions? Calling an API is More time consuming than calling a User Function. Justify?