Unix System Interface

The main topic of this lecture is Chap 8 (K&R)

Unix System Interface.

Command Line Arguments

```
 #include <stdio.h>
/* echo command-line arguments - version 1- page 115 K&R*/
main(int argc, char *argv[])
{
        int i;

        for( i  = 1;  i  <  argc;  i++)
                printf("%s  " , argv[i] );
        printf ("\n");
        return  0;
}
```

The program myecho.c compiled as
gcc myecho.c -o myecho
will produce the executable file myecho.

$ myecho hello, world
produces the output
hello, world

Command Line Arguments

argc and argv are a C mechanism for getting program arguments from the command line.

argc and argv parameters are passed automatically to the main () function by the operating system.

argc is the number of command line words, including the command name

if the program is executed with the command

    myecho hello, world

then argc is 3.

Command Line Arguments

The declaration

```
char   *argv[]
```

indicates that argv is an array of pointers to characters, i.e., argv is an array of character strings.

The number of entries of argv is argc.
The first one, argv[0], is the program name.
The rest, argv[1] ... argv[argc-1], are the command line arguments.

In the command

```
myecho  hello, world
```

argv[0] - "myecho"
argv[1] - "hello,"
argv[2] - "world"

Command Line Arguments

argc is the number of command-line arguments the program is invoked with

argv is a pointer to an array of character strings that contain the argument,one per string

argv[0] is the name by which the program is invoked

argc is at least 1
argv[1] is the first operational argument
argv[argc-1] is the last operational argument
argv[argc] is a null pointer

Command Line Arguments

```
#include <stdio.h>
/* echo command-line arguments - version  2- page 115
K&R*/
main(int argc, char  *argv[])
{
      while(--argc  >  0)
           printf ("%s " ,  *++argv);
      printf( "\n");
      return  0;
}
```

$ myecho hello, world
produces the output
hello, world

Command Line Arguments

argc is the number of command-line arguments the program is invoked with

argv is a pointer to an array of character strings that contain the argument,one per string

argv[0] is the name by which the program is invoked

argc is at least 1
argv[1] is the first operational argument
argv[argc-1] is the last operational argument
argv[argc] is a null pointer

Command Line Arguments

```
#include <stdio.h>

/* echo command-line arguments - version  3*/
main(int argc, char  **argv)
{
      while(--argc  >  0)
           printf ("%s %s" ,  *++argv,  (i < argc-1)  ?  " " : "");
      printf( "\n");
      return  0;
}
```

$ myecho hello, world
produces the output
hello, world

Command Line Arguments

Note that each element of the array argv [ ] is a string. This aspect is brought out by the program below

```
#include <stdio.h>
/* power program with command line arguments*/
main(int  argc,  char  *argv[ ])
{
        double  x;
        int      n;

        if (argc < 3) {
                printf (" missing arguments,  power [base] [index]\n");
                exit(1);
        }
        else {
                x = atof (argv [1]);
                n = atoi (argv [2]);
                printf ("%f \n", power(x, n));
                exit(0);
        }
}
```

Unix System Interface

Unix provides its services through a set of system calls.

System calls are essentially functions within the OS that can be called by user programs.

System calls are employed for reasons :

- to improve efficiency
- to access a facility not in the library

System calls can help to understand ANSI C library
ANSI C library is modeled on UNIX facilities

Unix System Calls for I/O

Used in the implementation of standard library.

In Unix, all I/O is done by reading or writing files.

In Unix, all peripheral devices including keyboard and screen are files in the file system. This enables a single homogeneous interface that handles all communication between a program and a peripheral device.

Before reading / writing a file it is necessary to open file.

File Descriptors

Before reading / writing a file it is necessary to open file.

To write on a file may also involve creating it or discarding its previous contents.

The system checks the user's rights to access a file.
If ok, it returns to the program a file desciptor.

A file descriptor is a small non negative integer

A file descriptor is used to identify a file.

File Descriptors

All information about an open file is maintained by the system

The user program refers to the file by the file descriptor

I/O involving keyboard and screen handled in a special wayto make this convenient.

When the shell (command interpreter) runs a program, three files are open with file descriptors 0, 1 & 2.

stdin - 0
stdout - 1
stderr - 2

File Descriptors

The command

prog <infile >outfile

redirects the input of the program prog to the file infile and the output to outfile.

In this case the shell changes the default assignment for file descriptors 0 and 1 to the named files.

Normally the file descriptor 2 is attached to the screen, so the error messages can go there.

File assignments are changed ny the shell, and not by the program. Program does not know where input comes from nor where its output goes, so long as it uses 0 for input and 1 and 2 for output.

Low level I/O - Read and Write

Input and output uses the read and write system calls.

```
int  n_read  =  read (int  fd, char  *buf,  int  n);

int  n_written  =  write (int  fd, char  *buf,  int  n);
```

The first argument is a file descriptor

The second argument is a character array in the user program that specifies the

- where the data is to go to (read)
- where the data comes from (write)

The third argument is the number of bytes to be transferred

Low level I/O - Read and Write

```
int  n_read  =  read (int  fd, char  *buf,  int  n);

int  n_written  =  write (int  fd, char  *buf,  int  n);
```

on reading, the number of bytes returned (n_read) may be less than the number requested (int n)

For read , a return value of zero indicates end of file and -1 indicates an error.

For write , the return value is the number of bytes written. An error has occured if it is not equal to the number requested.

Low level I/O - Read and Write

```
int  n_read  =  read (int  fd, char  *buf,  int  n);

int  n_written  =  write (int  fd, char  *buf,  int  n);
```

Any number of bytes can be read or written in one call.
Most common values are

1 (one character at a time)
1024 or 4096 (physical block size on a peripheral)

Larger sizes will be more efficient because fewer system calls are made.

Copy program

```
#include  "syscalls.h"

main( )  /*copy input to output */
{
        char buf[BUFSIZ];
        int n;

         while ( ( n  = read (0, buf, BUFSIZ) )  >  0)
                 write (1,  buf,  n);
         return  0;
}
```

syscalls.h contains the function prototypes for system calls into a file. This is not standard.

getchar () - version 1

```
#include  "syscalls.h"

/* unbuffered single character input */
int   getchar (void)
{
       char c;

      return  (read (0,  &c, 1) ==  1)  ?  (unsigned  char) c : EOF;
}
```

Note the use of typecast in the return expression.
Casting c to unsigned char in the return statement eliminates any problem of sign extension.

getchar () - version 2

```
#include  "syscalls.h"

/* getchar: buffered version*/
int   getchar (void)
{
       static char buf [BUFSIZ];
       static char *bufp = buf;
       static   int  n = 0;

      if (n  ==  0) {  /* buffer is empty  */
            n  =  read (0, buf, sizeof buf);
            bufp  =  buf;
      }
      return  (--n  >=  0) ?  (unsigned  char) *bufp++ : EOF;
}
```

Open and Creat - for opening files

A file must be explicitely opened in order to read or write it.

Default standard input, output and error are exceptions

open is like fopen, instead of returning a file pointer it returns a file descriptor, an integer.

```
#include <fcntl.h>

int   fd;
int   open (char *name,  int  flags,  int  perms);

fd  =  open (name, flags, perms);
```

Open and Creat - for opening files

```
int fd;
int   open (char *name,  int  flags,  int  perms);

fd  =  open (name, flags, perms);
```

The name argument is a character string containg the file name. The second argument, flags, is an int that specifies how the file is to be opened, its main values are

```
O_RDONLY     open for reading only
O_WRONLY     open for writing only
O_RDWR       open for both read and write
```

Open and Creat - for opening files

To open an existing file for reading,

```
fd  =  open (name, O_RDONLY, 0);
```

The perms argument is always 0 for uses of open here.

It is an error to open a file that does not exist.

Open and Creat - for opening files

The system call creat is used to create new files or to rewrite old ones

```
int fd;
int   creat (char *name,  int  perms);

fd  =  open (name, perms);
```

Returns a file descriptor if it was able to create a file and -1 if not

If the file already exists, creat will truncate it to zero length, thereby discarding its previous contents. It is not an error to create a file that already exists .

If the file does not already exist creat creates it with the permissions specified by the perms argument

File Permissions

In Unix there are 9 bits of permission information associated with a file that control read, write and execute access for the owner of the file, for the owner's group and for all others

A three digit octal number is used for specifying the permissions

e.g. 0755 specifies
- read, write and execute permissions for the owner
- read and execute permissions for group
- read and execute permissions for others

Lseek - Random Access

Input and output are normally sequential , i.e, each read / write takes place at a position in the file right after the previous one.

A file can also be read / written in any arbitrary order.

System call lseek to move around in a file without reading / writing any data

```
long    lseek (int fd, long offset, int origin);
```

sets the current position in the file whose descriptor is fd, to offset which is taken relative to the location specified by origin. Subsequent reading / writing begin at that position.

Lseek - Random Access

long lseek (int fd, long offset, int origin);

origin can be 0, 1, 2 to specify that offset is to be measured from the beginning, from the current position or from the end of file

Upon successful completion, lseek returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of -1 is returned

To append a file , seek to the end before writing

```
lseek(fd, 0L, 2)
```

To get back to the beginning

```
lseek(fd, 0L, 0)
```

lseek - example

```
#include  "syscalls.h"

/* get:  read n bytes from position pos */
int   get (int  fd,  long pos, char  *buf,  int  n)
{
       if  (lseek (fd,  pos,  0)  >=  0)  /* get to pos  */
             return  read (fd,  buf,  n);
       else
             return  -1;
}
```

Unix Directory Structure

Unix supports hierarchical file system.
The root directory / at the top

A directory is a file that contains a list of file names and some indication of where they are located.

The ' location ' is an index to another table called inodelist.

The inode for a file is where all the information about a file except its name is kept.

The directory entry generally consists of only two items, the filename and the inode number (K&R page 179)

ls is the directory listing program.