Unix System Programming

Files

Outline

Last Week:

- UNIX history/interface
- The PC revolution
- UNIX overview - process, shell, file
  - » system calls vs library routines
- Basic file I/O - open(), close(), read(), write(), lseek()
- Standard file I/O library - fopen(), fclose(), …

This Week:

- UNIX history - more on the key players
- Efficiency read/write
- The File
- File pointer
- File control/access

UNIX Key Players/Time Line

- 1969 Ken Thompson (Unix OS) - ARPANET
- 1971 Dennis Ritchie creates "C" language (1973 UNIX-C)
- 1977 Bill Joy (BSD released, TCP/IP-1980, open source, Internet backbone, Sun Microsystem in 1982 - NFS)
- 1984 Richard Stallman (RMS, emacs, GPL, GNU, HURD-91)
- 1985 Steve Jobs (NeXT-Mach, Mac OS X - 2001)
- 1985 Avie Tevanian (CMU/Mach)
- 1991 Linus Torvalds (Linux, based on Minix-Tannenbaum)

read/write and efficiency

- Evaluated by copyfile that reads from one file and writes to another:

```
while( nread = read( infile, buffer, BUFSIZE )
    if( write( outfile, buffer, nread ) < nread )
      close_return( outfile, infile );
```

- Time Command
  - » Granularity is a factor (50, 60, 100 ticks per second)
  - » User time (not system call)
  - » System time (kernel time, e.g. performing read() and writes())
  - » Real time (elapsed time from start to completion)
- What is an appropriate BUFSIZE?
  - » 1 byte?
  - » 512 bytes?
  - » 1000 bytes?

read/write and efficiency (cont)

- 68,307 byte file on computer running SVR 4 UNIX with block size 512

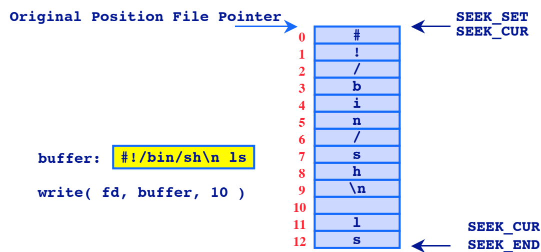
BUFSIZE	Real Time	User Time	System Time
1	24.49	3.13	21.16
64	0.46	0.12	0.33
512	0.12	0.02	0.08
4096	0.07	0.00	0.05
8192	0.07	0.01	0.05

- 1 byte at a time bad performance
- Best performance when BUFSIZE is a multiple of block size
  - » Less system calls, reduces context switches

File Pointer

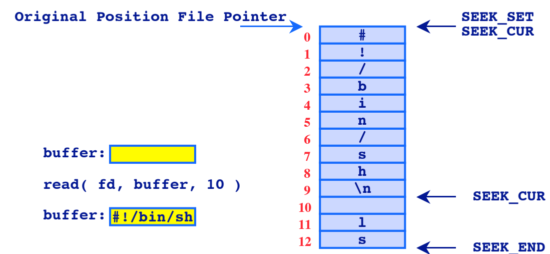
- Both `read()` and `write()` changes the file pointer.
- The pointer is incremented by exactly the number of bytes read or written.
- `lseek()` - repositions the file pointer for direct access to any part of the file

write() - File Pointer



- Output file descriptor traditional 1 for standard output, better to use constants defined in `<unistd.h>` : `STDOUT_FILENO`.

read() - File Pointer



- Output file descriptor traditional 0 for standard output, better to use constants defined in `<unistd.h>` : `STDIN_FILENO`.
- Recall: Returns -1 on error, 0 end of file, or #bytes read

lseek()

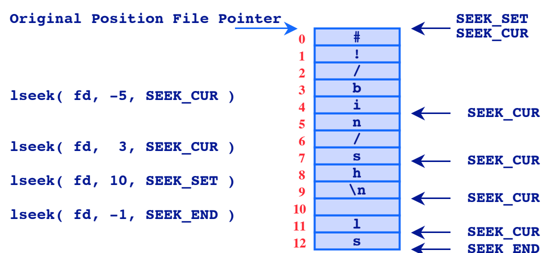
```
#include        <sys/types.h>
#include        <unistd.h>
long lseek( int fd, off_t offset, int whence );
```

- Repositions the offset of the file descriptor `fd` to argument `offset`.
- `Whence constants:`
  - » SEEK_SET (usually 0)
    - – The file pointer is set to `offset` bytes from beginning of file (default 0)
  - » SEEK_CUR (usually 1)
    - – The file pointer is set to its current location plus `offset` bytes (default 1, may be negative).
  - » SEEK_END (usually 2)
    - – The file pointer is set to the size of the file plus `offset` bytes.
- The return value is the new value of the pointer if the routine has executed successfully (`offset` of 0 returns current value of pointer, -1 indicates an error, negative offsets possible for non-regular files)

lseek: Simple Examples

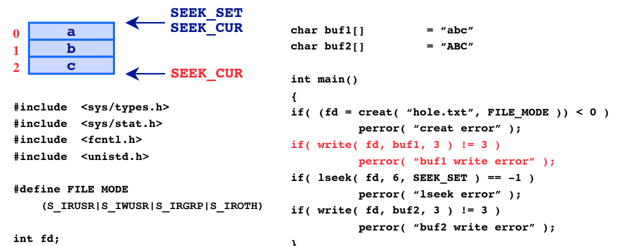
- **Random access**
  - » Jump to any byte in a file
- **Move to byte #16**
  - » `newpos = lseek( file_descriptor, 16, SEEK_SET );`
- **Move forward 4 bytes**
  - » `newpos = lseek( file_descriptor, 4, SEEK_CUR );`
- **Move to 8 bytes from the end**
  - » `newpos = lseek( file_descriptor, -8, SEEK_END );`

lseek - Examples



- lseek(fd, (off_t) -1, SEEK_END) - 1 bytes *before* the end of file
- OK to specify a position *beyond* the end of a file - next write creates a hole
- Not OK to specify a position *before* the beginning of the file

lseek - Hole (1)



- OK to specify a position *beyond* the end of a file - next write creates a hole
- Not OK to specify a position *before* the beginning of the file

lseek - Hole (2)

0	a
1	b
2	c
3	
4	
5	

SEEK_SET
SEEK_CUR
SEEK_CUR

```
char buf1[]          = "abc"
char buf2[]          = "ABC"

int main()
{
if( (fd = creat( "hole.txt", FILE_MODE )) < 0 )
          err_sys( "creat error" );
if( write( fd, buf1, 3 ) != 3 )
          err_sys( "buf1 write error" );
if( lseek( fd, 6, SEEK_SET ) == -1 )
          err_sys( "lseek error" );
if( write( fd, buf2, 3 ) != 3 )
          err_sys( "buf2 write error" );
}
```

- OK to specify a position *beyond* the end of a file - next write creates a hole
- Not OK to specify a position *before* the beginning of the file

lseek - Hole (3)

0	a
1	b
2	c
3	\0
4	\0
5	\0
6	A
7	B
8	C

SEEK_SET
SEEK_CUR
SEEK_CUR

```
{cinnamon:ingrid:35} od -c hole.txt
0000000    a   b   c  \0  \0  \0   A   B   C
0000011
```

```
char buf1[]          = "abc"
char buf2[]          = "ABC"

int main()
{
if( (fd = creat( "hole.txt", FILE_MODE )) < 0 )
          err_sys( "creat error" );
if( write( fd, buf1, 3 ) != 3 )
          err_sys( "buf1 write error" );
if( lseek( fd, 6, SEEK_SET ) == -1 )
          err_sys( "lseek error" );
if( write( fd, buf2, 3 ) != 3 )
          err_sys( "buf2 write error" );
}
```

- subsequent write cause file to be extended
- All bytes that have not been written are read back as 0.

File Control - via fcntl()

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

int fcntl( int fd, int cmd );
int fcntl( int fd, int cmd, long arg );
int fcntl( int fd, int cmd, struct lock *ldata )
```

- Performs operations on an open file, pertaining to the `fd`, the file descriptor
- Performs a variety of functions instead of having a single well-defined role
- Possible values of cmd is listed in `fcntl.h`
- Third parameter and its type depends on `cmd`

fcntl: `cmd` – get/set file status flags

- F_GETFL
  - Returns the current file status flags as set by `open()`.
  - Access mode can be extracted from AND'ing the return value
    - return_value & O_ACCMODE
      - e.g. O_WRONLY
- F_SETFL
  - Sets the file status flags associated with fd.
  - Only O_APPEND, O_NONBLOCK and O_ASYNC may be set.
  - Other flags are unaffected

File Status Flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for read & write
O_APPEND	append on each write
O_NONBLOCK	Non blocking mode
O_SYNC	wait for writes to finish
O_ASYNC	asynchronous I/O

fcntl: `cmd` – get/set file status flags

- Example: takes a single command line argument that specifies a file descriptor and prints out a descriptor of the file flags for that descriptor

```
{saffron} a.out 0  # stdin file descriptor
read only
{saffron} a.out 1  # stdout file descriptor
write only
{saffron} a.out 2  # stderr file descriptor
read write
```


Example 1: fcntl - F_GETFL

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

int main( int argc, char *argv[] )
  {
  int accmode, val;

  if( argc != 2 )
       {
       fprintf( stderr, "usage: a.out <descriptor#>" );
       exit(1);
       }

  if( (val = fcntl( atoi(argv[1]), F_GETFL, 0 ) ) < 0 )
       {
       perror( "fcntl error for fd" );
       exit( 1 );
       }

  accmode = val & O_ACCMODE;
```

```
if( accmode == O_RDONLY )
        printf( "read only" );
  else if(accmode == O_WRONLY )
        printf( "write only" );
  else if( accmode == O_RDWR )
        printf( "read write" );
  else
        {
        fprintf( stderr, "unknown access mode" );
        exit(1);
        }

  if( val & O_APPEND )
        printf( ", append");
  if( val & O_NONBLOCK)
        printf(", nonblocking");
  if( val & O_SYNC )
        printf(", synchronous writes");
  putchar( '\n' );
  exit(0);
}
```


fcntl - FGET_FL & FSET_FL

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

/* flags are file status flags to turn on */
void set_fl( int fd, int flags )
  {
  int val;

  if( (val = fcntl( fd, F_GETFL, 0 )) < 0 )
        {
        perror( "fcntl F_GETFL error" );
        exit( 1 );
        }
  val |= flags;     /* turn on flags */
  if( fcntl( fd, F_SETFL, val ) < 0 )
        {
        perror( "fcntl F_SETFL error" );
        exit( 1 );
        }
  }
```


errno and perror()

- Unix provides a globally accessible integer variable that contains an error code number
- Error variable: errno – errno.h
- perror(“ a string “): a library routine, not a system call

```
{atlas} more /usr/include/sys/*errno.h
.
.
.
#define EPERM           1       /* Operation not permitted */
#define ENOENT          2       /* No such file or directory */
#define ESRCH           3       /* No such process */
#define EINTR           4       /* Interrupted system call */
#define EIO             5       /* I/O error */
#define ENXIO           6       /* No such device or address */
.
.
.
```



```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main()
  {
  extern int errno;
  int fd;

  /* open file "ugh" for reading */
  if( fd = open( "ughugh", O_RDONLY ) == -1 )
        {
        fprintf( stderr, "Error %d\n", errno );
        perror( "ugh" );
        }
  } /* end main */

{saffron:ingrid:57} gcc ugga.c -o ugga
{saffron:ingrid:57} ls
ugga ugga.c
{saffron:ingrid:57} ./ugga
Error 2
ugh: No such file or directory
```


The Standard IO Library

- `fopen,`
- `fclose,`
- `printf, fprintf, sprintf, scanf, fscanf, getc, putc, gets, fgets, etc.`
- `#include <stdio.h>`

Why use read()/write()

- Maximal performance
  - IF you know exactly what you are doing
  - No additional hidden overhead from stdio
- Control exactly what is written/read at what times

File Concept - An Abstract Data Type

- File Types
- File Operations
- File Attributes
- Internal File Structure

File Types

- Regular files (text or binary)
- Directory files (names and pointers of files)
- Character special files (used by certain devices)
- Block special files (typically disk devices)
- FIFOs (used for interprocess communication)
- Sockets (usually for network communication)
- Symbolic Links (points to another file)

File Mix on a Typical System

File Type	*Count*	*Percentage*
regular file	30,369	91.7%
directory	1,901	5.7
symbolic link	416	1.3
char special	373	1.1
block special	61	0.2
socket	5	0.0
FIFO	1	0.0

File Operations

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file

Files Attributes: Meta-Data

System information on disk associated with each file:

- Name – only information kept in human-readable form.
- Type – needed for systems that support different types.
- Location – pointer to file location on device/disk.
- Size – current file size.
- Protection bits – controls who can do reading, writing, executing.
- Time, date, and user identification – data for protection, security, and usage monitoring.
- Special file?
  - » Directory, Symbolic link, …
  - » Information about files are kept in the directory structure, which is maintained on the disk (later)

```
{atlas:maria:143} ls -lig ch11.ppt
231343 -rw-r--r--   1 profs     815616 Nov  4  2002 ch11.ppt
```


Obtaining File Information

Great for analyzing files.

- `stat(), fstat(), lstat()`
- Retrieve all sorts of information about a file
  - » Which device it is stored on
  - » Don't need access right to the file, but need search rights to directories in path leading to file
  - » Information:
    - – Ownership/Permissions of that file,
    - – Number of links
    - – Size of the file
    - – Date/Time of last modification and access
    - – Ideal block size for I/O to this file

stat, fstat, lstat

```
#include <sys/stat.h>
#include <unistd.h>
int stat( const char *file_name, struct stat *buf );
int fstat( int fd, struct stat *buf );
int lstat( const char *file_name, struct stat *buf );
```

- **stat( ), fstat( )**
  - Stats the file pointed to by *file_name* or by *fd* and fills in *buf*.
- **lstat( )**
  - Same as `stat( )` except that the symbolic link is stated itself (i.e. do not follow the link).

struct stat

```
struct stat
  {
  dev_t     st_dev;     /* device num.           */
  dev_t     st_rdev;    /* device # special files */
  ino_t     st_ino;     /* i-node num.           */
  mode_t    st_mode;    /* file type, perms      */
  nlink_t   st_nlink;   /* num. of links         */
  uid_t     st_uid;     /* uid of owner          */
  gid_t     st_gid;     /* group-id of owner     */
  off_t     st_size;    /* size in bytes         */
  time_t    st_atime;   /* last access time      */
  time_t    st_mtime;   /* last mod. time        */
  time_t    st_ctime;   /* last stat chg time    */
  long      st_blksize; /* best I/O block size   */
  long      st_blocks;  /* # of 512 blocks used  */
  }
```


st_dev & st_rdev

- **`st_dev` holds the device number of the *file system* where the file is located:**
  - usually a hard disk
- **`st_rdev` holds the device number for a *special file*.**
  - A special file is used to describe a device (peripheral) attached to the machine:
  - CD drives, keyboard, hard disk, microphone, etc.
  - Special files are usually stored in `/dev`

st_mode

- **File types (regular file, directory, socket, …)**
- **File permissions**

st_mode: Getting the Type Information

- **AND the `st_mode` field with `S_IFMT` to get the type bits.**
- **then test the result against:**
  - `S_IFREG` Regular file
  - `S_IFDIR` Directory
  - `S_IFSOCK` Socket
  - etc.

Example

```
struct stat sbuf;
      :
if( stat( file, &sbuf ) == 0
  if( ( sbuf.st_mode & S_IFMT ) == S_IFDIR )
          printf( "A directory\n" );
```

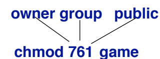
Type Info. Macros

- Modern UNIX systems include test macros in `<sys/stat.h>` and `<linux/stat.h>`:
  - » `S_ISREG()` regular file
  - » `S_ISDIR()` directory file
  - » `S_ISCHR()` char. special file
  - » `S_ISBLK()` block special file
  - » `S_ISFIFO()` pipe or FIFO
  - » `S_ISLNK()` symbolic link
  - » `S_ISSOCK()` socket

Type Info. Macros: Example

```
struct stat sbuf;
    :
  if( stat(file, &sbuf ) == 0 )
    {
    if( S_ISREG( sbuf.st_mode ) )
        printf( "A regular file\n" );
    else if( S_ISDIR(sbuf.st_mode) )
        printf( "A directory\n" );
    else ...
    }
```


st_mode: Permission Code



- Determines who can access and manipulate a directory or file
- Mode of access: read, write, execute
- Three classes of users (3 fields of 3 bits each)

			RWX
a) owner access	7	⇒	1 1 1
b) group access	6	⇒	1 1 0
c) public access	1	⇒	0 0 1

```
drw-r—r---  maria profs 512 May 15 22:15 hello.txt
```

- Group contains a set of users

```
chgrp mgroup game
```


chmod shell command

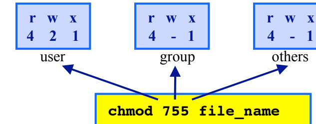
```
chmod [-options] modes file/directory
```

- options ⇒ option: -R
- modes:
  - » Who? u|g|o|a
  - » Operator? =|+|-
  - » Permissions? r|w|x

Example:

```
chmod u=rwx,g+w,o-w  maria.txt
```

- » Octal - one octal per user, representing 3 bit positions



chmod and fchmod

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod( const char *path, mode_t mode ) ;
int fchmod( int fd, mode_t mode );
```

- Change permissions of a file.
- The mode of the file given by *path* or referenced by *fd* is changed.
- *mode* is specified by OR'ing the following.
  - S_ISUID, S_ISGID, S_ISVTX, S_I{R,W,X}{USR,GRP,OTH}
- Effective uid of the process must be zero (superuser) or must match the owner of the file.
- On success, zero is returned. On error, -1 is returned

chmod example

- Modify permission on files foo and bar

```
{atlas} ls -l foo bar
-rw------- 1 maria    0 Nov 15    15:43  bar
-rw-rw-rw- 1 maria    0 Nov 15    15:43  foo
```

- So that new state is

```
{atlas} ls -l foo bar
-rw-r--r-- 1 maria    0 Nov 15    15:43  bar
-rw-rwlrw- 1 maria    0 Nov 15    15:43  foo
```

- Group execute is listed as 'l' to signal mandatory locking

Example: chmod()

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(void)
  {
  struct stat statbuf;

  /* turn on set-group-ID and turn off group-execute */
  if( stat("foo", &statbuf) < 0 )
       {
       perror("stat error for foo" );
       exit(1);
       }
  if( chmod("foo",  (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0 )
       {
       perror("chmod error for foo");
       exit(1);
       }
```



```
/* set absolute mode to "rw-r--r--" */
if( chmod("bar", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) < 0)
       {
       perror("chmod error for bar");
       exit(1);
       }
   exit(0);
   }
```


chown, fchown, lchown

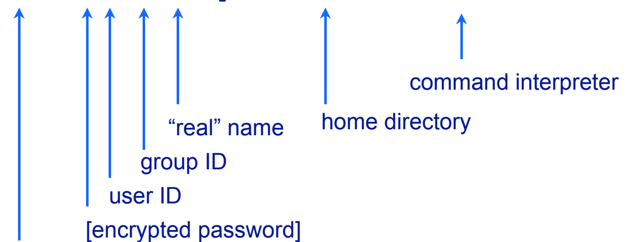
```
#include <sys/types.h>
#include <unistd.h>
int chown( const char *path, uid_t owner, gid_t group );
int fchown( int fd, uid_t owner, gid_t group );
int lchown( const char *path, uid_t owner, gid_t group );
```

- Change user ID of a file and the group ID of a file.
- Only the superuser may change the owner of a file.
- The owner of a file may change the group of the file to any group of which that owner is a member.
- When the owner or group of an executable file are changed by a non-superuser, the S_ISUID and S_ISGID mode bits are *cleared*.

st_uid: Users and Ownership: /etc/passwd

- Every file is owned by one of the system's users – identity is represented by the user-id (UID) of owner (st_uid in stat)
- Password file associated UID with system users.

`maria:x:65:20:M. Hybinette:/home/maria:/bin/ksh`



/etc/group

- Information about system groups

`faculty:x:23:maria,eileen,dkl`

Real uids

- The uid of the user who *started* the program is used as its *real uid*.
- The real uid affects what the program can do (e.g. create, delete files).
- For example, the uid of `/usr/bin/vi` is `root`:
  - `$ ls -alt /usr/bin/vi`
    `lrwxrwxrwx 1 root root 20 Apr 13...`
- But when I use `vi`, its *real uid* is `maria` (not `root`), so I can only edit my files.
- Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the `stat` structure that we will talk about shortly.

Effective uids

- Normally executing program's *effective uid* is the same as the real uid, however sometimes a process may change to use the owner's ID of a file/program.
  - » the uid of the program *owner*
  - » e.g. the `passwd` program changes to use its effective uid (`root`) so that it can edit the `/etc/passwd` file
- The process determines its *effective uid* by looking at the file's mode flag (`st_mode`)
- This feature is used by many system tools, such as logging programs.

Real and Effective Group-ids

- There are also real and effective group-ids.
- Usually a program uses the *real group-id* (i.e. the *group-id of the user*).
- Sometimes useful to use *effective group-id* (i.e. group-id of program *owner*):
  - » e.g. software shared across teams

Extra File Permissions

- *Octal Value* *Meaning*

Octal Value	Meaning
04000	Set user-id on execution. Symbolic: `--s --- ---`
02000	Set group-id on execution. Symbolic: `--- --s ---`
01000	Save-text-image (sticky bit) Symbolic: `--- --- --t`

- These specify that a program should use the effective user/group id during execution.
- For example:
  - » `$ ls -alt /usr/bin/passwd`
    `-rwsr-xr-x 1 root root 25692 May 24...`

Sticky Bit

Octal	Meaning
01000 execution.	Save text image on Symbolic: `--- --- --t`

- This specifies that the program code should stay resident in memory after termination.
  - » this makes the start-up of the next execution faster
- Obsolete due to virtual memory.

`st_mode`: Permissions

- This field contains type and permissions (12 lower bits) of file in bit format.
- It is extracted by AND-ing the value stored there with various constants
  - » see `man stat`
  - » also `<sys/stat.h>` and `<linux/stat.h>`
  - » some data structures are in `<bits/stat.h>`

Getting Permission Information

- AND the `st_mode` field with one of the following masks and test for non-zero:

Mask	Value	Meaning
`S_IRUSR`	0400	user read
`S_IWUSR`	0200	user write
`S_IXUSR`	0100	user execute
`S_IRGRP`	0040	group read
`S_IWGRP`	0020	group write
`S_IXGRP`	0010	group execute
`S_IROTH`	0004	other read
`S_IWOTH`	0002	other write
`S_IXOTH`	0001	other execute

- <sys/stat.h>

Example

```
struct stat sbuf;
    :
printf( "Permissions: " );
if( ( sbuf.st_mode & S_IRUSR ) != 0 )
    printf( "user read, " );
if( ( sbuf.st_mode & S_IWUSR ) != 0 )
    printf( "user write, " );
    :
```

- Or use octal values, which are easy to combine:

```
if( ( sbuf.st_mode & 0444 ) != 0 )
  printf( "readable by everyone\n" );
```


st_mode: Getting Mode Information

- AND the `st_mode` field with one of the following masks and test for non-zero:
  - » `S_ISUID` set-user-id bit is set
  - » `S_ISGID` set-group-id bit is set
  - » `S_ISVTX` sticky bit is set
- Example:

```
if( (sbuf.st_mode & S_ISUID) != 0 )
  printf( "set-user-id bit is set\n" );
```


The superuser

- Most system admin. tasks can only be done by the *superuser* (also called the *root* user)
- Superuser
  - » has access to all files/directories on the system
  - » can override permissions
  - » owner of most system files
- Shell command: `su <username>`
  - » Set current user to superuser or another user with proper password access

User Mask: umask

- Unix allows "masks" to be created to set permissions for "newly-created" directories and files.
- The umask command automatically sets the permissions when the user creates directories and files (umask stands for "user mask").
- Prevents permissions from being accidentally turned on (hides permissions that are available).
- Set the bits of the umask to permissions you want to mask out of the file permissions.
- This process is useful, since user may sometimes forget to change the permissions of newly-created files or directories.

```
fd = open( path, O_CREAT, mode ) ⇒
  fd = open( path O_CREAT, (~umask) & mode)
```


umask (1)

- Defaults (executable must be manually set - after they are created)

File Type	Default Mode
Non-executable files	666
Directories	777

From this initial mode, Unix subtracts the value of the umask.

mask	Directory (777)	File (666)
0	7 (rwx)	6 (rw-)
1	6 (rw-)	6 (rw-)
2	5 (r-x)	4 (r--)
3	4 (r--)	4 (r--)
4	3 (-wx)	2 (-w-)
5	2 (-w-)	2 (-w-)
6	1 (--x)	0 (---)
7	0 (---)	0 (---)

umask: Calculations (2)

- If you want a file permission of 644 on a regular file, the umask would need to be 022.

```
Default Mode            666
umask                  -022
New Allowable File Mode  644
```

- Bit level: new_mask = mode & ~umask

```
   umask   = 000110110 = ---rw-rw = 0066
  ~umask   = 111001001
    mode   = 110110110 = rw-rw-rw = 0666
new_mask   = 110000000 = rw------ = 0600
```

umask (3)

- **Common Settings:**

mask	Directory (777)	File (666)
000 (public)	777 (rwx rwx rwx)	666 (rw- rw- rw-)
011 (public)	766 (rwx rw- rw-)	666 (rw- rw- rw-)
022 (write protected)	755 (rwx r-x r-x)	644 (rw- r-- r--)
007 (project private)	770 (rwx rwx ---)	660 (rw- rw- ---)
077 (private)	700 (rwx --- ---)	600 (rw- --- ---)

umask

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t       umask( mode_t mask );
```

- Set file mode creation *mask* and returns the old value. *mask* is formed as the bitwise OR of any of the nine file permission constants from <sys/stat.h>: S_IRUSR, S_IWUSR, S_IXUSR, …
- There is no error return
- When creating a file, permissions are turned off if the corresponding bits in *mask* are set.
- Return value
  - This system call always succeeds and the previous value of the mask is returned.
  - "umask" shell command

Example: umask

```
int main(void)
  {
  umask(0);                                    /* --- --- --- */

  if( creat( "foo", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH ) < 0 )
        {           /* rw- rw- rw- **/
        perror("creat error for foo");
        exit(1);
        }

   umask( S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH );       /* --- rw- rw- */
   if( creat( "bar", S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH) < 0 )
        {            /* rw- rw- rw- */
         perror("creat error for bar");
         exit(1);
        }
  exit(0);
  }

{saffron:maria:68} ls -ltra foo bar
-rw-rw-rw-       1 maria    faculty          0 Apr  1 20:35 foo
-rw-------       1 maria    faculty          0 Apr  1 20:35 bar
```