UNIX System Programming

Files and Directories

Last Week

- UNIX history/interface
- open(), close()
- read(), write()
- Efficiency and read/write
- FILE pointer

Last Week: Efficiency

- Block Size (last week)

Last Week: File Pointer

- Both read() and write() will change the file pointer.
- The pointer will be incremented by exactly the number of bytes read or written.

lseek

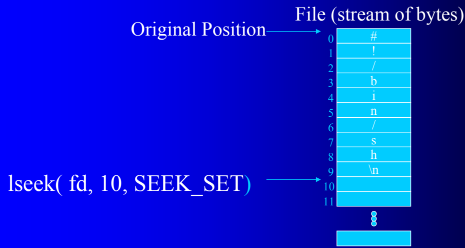
```
#include <sys/types.h>
#include <unistd.h>
off_t lseek( int fd, off_t offset, int whence );
```

- Repositions the offset of the file descriptor *fd* to the argument offset.
- *whence*
  - SEEK_SET
    - The offset is set to offset bytes.
  - SEEK_CUR
    - The offset is set to its current location plus offset bytes.
  - SEEK_END
    - The offset is set to the size of the file plus offset bytes.

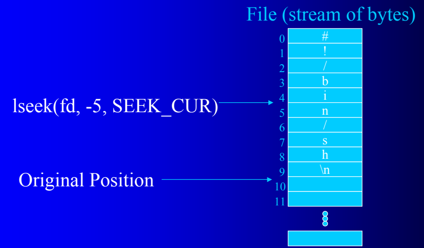
lseek: Examples

- 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 );`

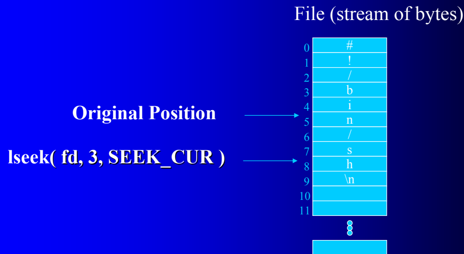
lseek - SEEK_SET (10)



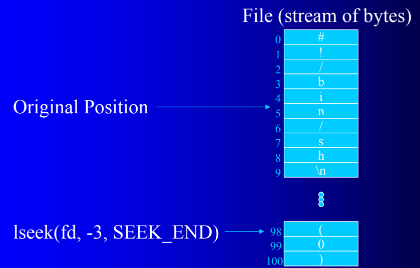
lseek - SEEK_CUR (-5)



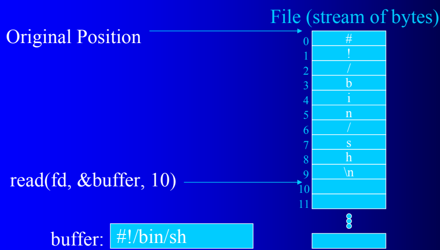
lseek - SEEK_CUR(3)



lseek - SEEK_END (-3)

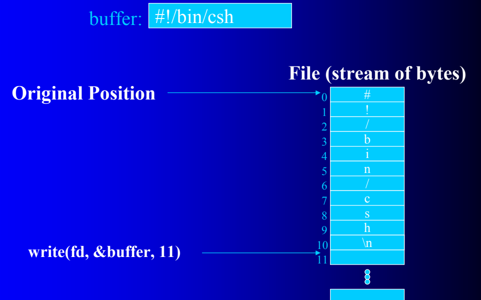


Read – File Pointer



Write – File Pointer

buffer: #!/bin/csh



Example #1: lseek

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

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void)
  {
   int fd;

   if( (fd = creat("file.hole", 0640)) < 0 )
      {
      perror("creat error");
      exit(1);
      }
```


Example #1: lseek (2)

```
if( write(fd, buf1, 10) != 10 )
     {
     perror("buf1 write error");
     exit(1);
     }
 /* offset now = 10 */
if( lseek(fd, 40, SEEK_SET) == -1 )
     {
     perror("lseek error");
     exit(1);
     }
/* offset now = 40 */
if( write(fd, buf2, 10) != 10  )
     {
     perror("buf2 write error");
     exit(1);
     }
/* offset now = 50 */
exit(0);
}
```


Example #2: lseek - appending

File control of open files: 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 pertaining to *fd*, the file descriptor
- Specific operation depends on cmd

fcntl: cmd

- 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

Example 1: fcntl()

```
#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, "unkown 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);
}
```


Example #2: fcntl

```
#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 accesible integer variable that contains an error code number
- Error variable: errno – errno.h
- perror(“ a string “): a library routine

{saffron:ingrid:10} more /usr/include/asm/*errno.h

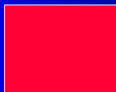
```
#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

#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 */
```


errno and perror()

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

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

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


{saffron:ingrid:57} errno
Error 2
hello: 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

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

File Types

- Regular files
- Directory files
- Character special files
- Block special files
- FIFOs
- Sockets
- Symbolic Links

File Operations

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

Files Attributes

- Name
- Type
- Location
- Size
- Protection
- Time,date and user identification

Users and Ownership: /etc/passwd

- Every File is owned by one of the system's users – identity is represented by the user-id (UID)
- Password file assoicate UID with system users.

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

command interpreter
home directory
"real" name
group ID
user ID
[encrypted password]
login name

/etc/group

- Information about system groups

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

list of group members
group ID
[encrypted group password]
group name

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.

Effective uids

- Programs can change to use the *effective uid*
  - 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
- 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*
`04000`	Set user-id on execution. Symbolic: `--s --- ---`
`02000`	Set group-id on execution. Symbolic: `--- --s ---`

- 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`	Save text image on execution. 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.

The superuser

- Most sys. 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

File Mode (Permission)

- S_IRUSR -- user-read
- S_IWUSR -- user-write
- S_IXUSR -- user-execute
- S_IRGRP -- group-read
- S_IWGRP -- group-write
- S_IXGRP -- group-execute
- S_IROTH -- other-read
- S_IWOTH -- other-write
- S_IXOTH -- other-execute

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.

umask: Calculations (1)

- **Defaults**

File Type	**Default Mode**
Non-executable files	666
Executable files	777
Directories	777

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

umask: Calculations (2)

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

Default Mode	666
umask	-022
New 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

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

- Set file mode creation *mask* and return the old value.
- 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.
  - cf. "umask" shell command

Example: umask

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

int main(void)
  {
  umask(0);

  if( creat( "foo",
        S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH ) < 0 )

        {
        perror("creat error for foo");
        exit(1);
        }

  umask( S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH );
  if( creat( "bar",
        S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH) < 0 )
        {
         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
```

access()

- `#include <unistd.h>`
```
int access(char *pathname,
           int access_mode);
```
- According to the *real uid*, can the program using `access()` access the file? Return 0 if ok, -1 on error.
  - e.g. jim is using maria's edit program but it should only edit jim's files

Mode Values

Mode Values		*Meaning*
`R_OK`	`04`	Has calling process read access?
`W_OK`	`02`	Has calling process write access?
`X_OK`	`01`	Can calling process execute the file?
`F_OK`	`0`	Does file exist?

Example: access

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

int main( int argc, char *argv[] )
  {
   if( argc != 2 )
      {
      fprintf(stderr, "usage: a. out <pathname>");
      exit(1);
      }
   if( access( argv[1], R_OK ) < 0 )
         perror( "User cannot read the file" );
     else
         printf( "read access OK\n" );
     if( open(argv[1], O_RDONLY) < 0 )
         perror("open error");
     else
         printf("open for reading OK\n");
     exit(0);
  }
```


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.

Example: chmod (1)

```
#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);
      }
```


Example: chmod (2)

```
 /* 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 );
```

- The owner of the file specified by *path* or by *fd*.
- 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*.

Obtaining File Information

Great for analysing 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

struct stat

We will look at these in detail.

```
struct stat
  {
  dev_t st_dev;      /* device num.          */
  dev_t st_rdev;     /* device # spcl files  */
  ino_t st_ino;      /* i-node num.          */
  mode_t st_mode;    /* file type,mode,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_dev** holds the device number of the *file system* where the file is located:
  - usually a hard disk

st_rdev

- **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, harddisk, microphone, etc.
- Special files are usually stored in **/dev**

st_rdev Format

- Two parts: major device number, minor device number.
- *Major device number*: specifies the device type. The system uses it to choose the right device driver.
- *Minor device number*: represents the actual device
  - port number, drive number, etc.

`ls -l`

- Major and minor device numbers can be displayed with `ls -l` /dev/ttyp0:

crw-rw-rw-   1 root   tty   3,   0 Apr 11 2002   /dev/ttyp0

`st_ino` (I-node number)

- Each file has a unique *i-node number* (index number).
- The i-node number can be used to look up a file's information (*i-node*) in a system table (the *i-list*).
- A file's i-node contains:
  - user and group ids of its owner
  - permission bits
  - etc.

File Types

1. Regular File (text/binary)
2. Directory File
3. Character Special File

   e.g. I/O peripherals, such as `/dev/ttyp0`
4. Block Special File

   e.g. cdrom, such as `/dev/mcd`
5. FIFO (named pipes)
6. Sockets
7. Symbolic Links

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

`st_mode` Field

- 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 the Type Information

- `AND` the `st_mode` field with `S_IFMT` to get the type bits.
- 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

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 ...
  }
```


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");
```


Getting Permission Info.

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

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" );
```


POSIX Flags for st_mode Field

- S_IFMT: 00170000 bit mask for the file type bit fields.
- S_IFSOCK: 0140000 socket.
- S_IFLNK: 0120000 symbolic link.
- S_IFREG: 0100000 regular file
- S_IFBLK: 0060000 block device
- S_IFDIR: 0040000 directory
- S_IFCHR: 0020000 character device
- S_IFIFO: 0010000 fifo
- S_ISUID: 0004000 set-user-id bit (POSIX)
- S_ISGID: 0002000 set-group-id bit (POSIX)
- S_ISVTX: 0001000 sticky bit
- S_IRWX{U,G,O}, S_I{R,W,X}USR, S_I{R,W,X}GRP, S_I{R,W,X}OTH

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).

Example: lstat ()

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

int main( int argc, char *argv[] )
  {
  int i;
  struct stat buf;
  char *ptr;

  for( i = 1; i < argc; i++ )
      {
      printf( "%s: ", argv[i] );
      if( lstat( argv[i], &buf ) < 0 )
            {
            perror("lstat error");
            continue;
            }
```



```
      if (S_ISREG(buf.st_mode))
            ptr = "regular";
      else if( S_ISDIR(buf.st_mode) )
            ptr = "directory";
      else if( S_ISCHR(buf.st_mode) )
            ptr = "character special";
      else if( S_ISBLK(buf.st_mode) )
            ptr = "block special";
      else if( S_ISFIFO(buf.st_mode) )
            ptr = "fifo";
#ifdef S_ISLNK
      else if( S_ISLNK(buf.st_mode) )
            ptr = "symbolic link";
#endif
#ifdef S_ISSOCK
      else if( S_ISSOCK(buf.st_mode) )
            ptr = "socket";
#endif
      else
            ptr = "** unknown mode **";
```



```
      printf( "%s\n", ptr );
      } /* end for loop */
  exit(0);
  } /* end main */
```

```
% a.out /bin/ls /etc /devices/pseudo/ptsl@0:ttyr9 /devices/sbus@3,0/SUNW,fas@3,8800000/sd@6,0:c
/bin/ls: regular
/etc: directory
/devices/pseudo/ptsl@0:ttyr9: character special
/devices/sbus@3,0/SUNW,fas@3,8800000/sd@6,0:c: block special
```