2803ICT Systems and Distributed Computing

Standard I/O

Dr. René Hexel

Standard IO Library

- A file is basically one or more blocks of characters or bytes stored on a file system.
- C/C++ provides a standard IO library for accessing files in various ways that is supported by all common Operating Systems
- Must use `#include <stdio.h>`
- The standard IO library provides lower level access than C++ <streams> but higher than kernel file system functions.
- This is called buffered IO
- Write once – run anywhere!
  - As long as there is a C/C++ compiler.

Stdio: File Access

- A file must be opened before you can do anything with it, but to do this you need to specify what you intended to use the file for using the function:

```
FILE* fopen(char *filename, char *mode )
```

- ***mode*** - specifies the intended access type
  - **"r"** Opens for reading. The file must exist
  - **"w"** Opens a file for writing. Any existing data is deleted
  - **"a"** Opens for appending without removing the EOF marker; creates the file first if it doesn't exist.
  - **"r+"** Opens for reading and writing. The file must exist.
  - **"w+"** Opens a file for reading & writing. Existing data is deleted
  - **"a+"** Opens for reading & appending; updates the EOF marker when transfer is done; creates the file if it doesn't exist.
- ***Returns***:
  - a handle to the opened file. Returns NULL to indicate an error

Stdio: Opening Files

- When a file is opened with the **"a"** or **"a+"** access type, all write operations occur at the end of the file.
- The file pointer can be repositioned using **fseek** or **rewind**, but is always moved back to the end of the file before any write operation is carried out. Existing data cannot be overwritten.
- When the **"r+", "w+",** or **"a+"** access type is specified, both reading and writing are allowed. But, when you switch between reading and writing, there must be an intervening **fflush**, **fsetpos**, **fseek**, or **rewind** operation

Stdio: Opening Binary or Text Files

- If you are reading a file with binary data and it contains a 0x1A (CTRL+Z) anywhere in the file the function will think that it is the end of the file. To avoid this fopen() can be told to interpret the file as binary and not text.
- The translation mode can be specified by appending either a t or a b to the end of the mode string:
  - **t** - Open in text (translated) mode. In this mode, CTRL+Z is interpreted as an end-of-file character. Also, CR-LF characters are translated into single linefeeds on input, and linefeed characters are translated to CR-LF combinations on output
  - **b** - Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed.
- Example: `fopen("file.exe", "rb");`

Stdio: Closing Files and Errors

```
#include <stdio.h>
int fclose(FILE *stream);
int fcloseall(void);
```

- Flushes all data to disk and closes the file
- ***Returns***
  - *fclose()* – if successful returns 0 else EOF and sets errno
  - *fcloseall()* – returns EOF to indicate an error

```
int ferror (FILE *stream);
```

- ***Returns*** – nonzero if there has been a file IO error

```
int feof (FILE *stream);
```

- ***Returns*** – nonzero if the EOF has been reached

```
void clearerr (FILE *stream);
```

- Resets the error and EOF indicators for a file

Stdio: Buffering Control

int setvbuf(FILE *stream, char *buf, int mode, size_t size);

- ***Mode*** – must be one of the following:
  - _IONBF Unbuffered – data sent directly to kernel
  - _IOLBF Line-buffered – data sent to kernel at each EOL
  - _IOFBF Block-buffered – “fully buffered” default for files
- ***Buf & size***
  - Points to a buffer of size bytes to be used for buffering
  - Otherwise If it is NULL a buffer will be allocated automatically
  - If supplied, Buf, must exist when the stream is closed
  - Ignored in case of _IONBF
- ***Returns*** – 0 if successful, otherwise non zero
- This function must be called immediately after the steam is opened, before any other stream operation
- **Warning: do not forget to explicitly close the streams before Buf goes out of scope.**

Stdio: Removing & Renaming Files

```
int remove (const char* filename );
```

- File must be closed for this call to work
- ***Returns***: 0 if successful otherwise 1, and sets errno to an appropriate OS specific code

```
int rename(char *oldname, char *newname );
```

- Will fail if both oldpath and newpath do not reside on the same file system.
- ***Returns***: 0 if successful otherwise 1, and sets errno to an appropriate OS specific code

Stdio: Reading char Data

- Reading and writing to files can be performed as single bytes, character strings, entire blocks or formatted data.

```
int fgetc( FILE *stream );
```

- ***Returns*** – the next char in the file or EOF
- To read an entire string up to a given length, a newline character or the end of the file we use

```
char* fgets(char *str, int n, FILE *strm);
```

- ***str*** – a pointer to a character buffer of n chars in length
- ***n*** – length of str character buffer (must allow for NULL)
- ***Returns*** – a pointer to a string of up to n chars in length if successful otherwise NULL

```
int ungetc(int c, FILE* stream);
```

- Puts the one character back into the file
- ***Returns*** – EOF on failure

Stdio: Reading Block Data

- To read any arbitrary block of data ignoring any newline and EOF characters

```
size_t fread(void *buf,
             size_t size,
             size_t count,
             FILE *stream );
```

- ***Buf*** – a pointer to where the data will be stored
- ***Size*** – the size of each data element
- ***Count*** – how many data elements to read
- ***Returns*** – the number of elements read (not bytes),
- use ferror() or feof() to check for errors

Stdio: Writing Data

```
int fputc(int c, FILE *stream );
```

- ***Returns*** – c if successful, otherwise EOF

```
int fputs(char *string, FILE *strm);
```

- ***string*** – a NULL terminated string
- ***Returns*** – a non negative number otherwise EOF

```
size_t fwrite(void *buf, size_t size,
              size_t count, FILE *stream);
```

- ***Buf*** – a pointer to data to be written
- ***Size*** – the size of each data element
- ***Count*** – how many data elements to read
- ***Returns*** – the number of elements written (not bytes)

```
int fflush(FILE *stream);
```

- Writes data in memory buffers to kernel disk buffer

Stdio: Controlling the File Position

- Seeking allows you to change the position in the file that reading or writing will take place

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

- ***Whence*** - can be one of
  - SEEK_SET - count offset from the start of file
  - SEEK_CUR - count offset from the current location
  - SEEK_END - count offset from the end of file
- ***Returns***
  - 0 if successful and may undo the last ungetc().
  - -1 on error and sets errno.

```
int fsetpos(FILE *stream, fpos_t *pos);
```

- Same as fseek() with whence = SEEK_SET

Stdio: Getting the File Position

```
#include <stdio.h>
void rewind(FILE *stream);
```

- is equivalent to fseek(stream, 0, SEEK_SET);

```
long ftell(FILE *stream);
```

- ***Returns*** - current file position or –1 on error

```
int fgetpos(FILE *stream, fpos_t *pos);
```

- ***pos*** – ptr to variable where FP will be written
- ***Returns*** – zero if successful otherwise –1

Stdio: Formatted File IO

int fprintf(FILE *_stream_, char *_format_ [, _args…_]);

- Simular to printf() except that the formatted output is written to a file instead of the console
- Example: fprintf(infile, "%s = %d", "Value", 5+1);

output ⇒ "Value = 6"

int fscanf(FILE *_stream_, char *_format_ [, _args…_]);

- Tries to match file data according to the given format string and match the data to the correct arguments
- Example: assume the file contents is "Value = 9"

```
char str [20];
int   num;              //- note we pass the address of num
fscanf(infile, "%s = %d", str, &num);
```

results ⇒ str = "Value", num = 9

Stdio: System Supplied Files

- It is possible for the system to create temporary files for you to use with a guaranteed unique filename

```
char *tmpnam(char *string);
```

- ***String*** – a pointer to a buffer where the generated filename will be written to
- ***Returns*** –address of the buffer the string was written

```
FILE* tmpfile(void)
```

- ***Returns*** – a pointer to a temporary file, opened as "wb+" that must be explicitly closed.
- Stdio also provides three predefined file streams
  - FILE* stdin ⟹ keyboard input
  - FILE* stdout ⟹ line buffered output to console window
  - FILE* stderr ⟹ unbuffered output toconsole window


POSIX File IO

Unix: User Commands

- Many Unix Shell Commands have corresponding systems calls that do the same thing

Shell	System
mkdir	mkdir()
rmdir	rmdir()
rm	unlink()
mv	link → unlink → rename
ln	link()
pwd	getcwd()

- Other commands have no equivalent:
  - cat, cp, pc, du, ls

Unix Devices

- A Unix file is a sequence of m bytes:
  - B0, B1, ... , Bk , ... , Bm-1
- All I/O devices are represented as files:
  - /dev/sda2 (/usr disk partition)
  - /dev/tty2 (terminal)
- Even the kernel is represented as a file:
  - /dev/kmem (kernel memory image)
  - /proc (kernel data struct
- We can access these just like files
- Examples
  - Who > /dev/ttyq3 cp /etc/passwd /dev/ttyq2

Unix Devices

- Some special devices
  - /dev/null - a bottomless data pit
  - /dev/zero - a endless fountain of zeros
  - /dev/kmem - kernel RAM as a file
- File permissions, owner, type, etc are stored in inodes and works the same way as for files
- Lseek is not defined for some devices
- BUT devices are different from files
  - Baud rates, parity bits stop bits, echo mode
  - CR/LF translation, buffering etc
- You can control these device config settings

Unix: File & Device IO

- Many system File IO calls directly correspond to those of stdio but they are unbuffered
- In addition the kernel provides functions for managing directories and for more advanced file access including
  - Asynchronous IO
  - File sharing (covered in another lecture)
  - File Mapping (covered in another lecture)
- In stdio, files are manipulated through FILE* streams but the uses kernel integer file descriptors instead
- There are functions that allow stdio streams to be converted to kernel files and vice versa.
- Most of these functions are part of POSIX - "Portable Operating System Interface for Unix"

Interoperation with C Stdio.h

- To convert an open file descriptor (fd) to a stream

```
#include <stdio.h>
FILE* fdopen (int fd, const char *mode);
```

- ***Mode*** – must be the same as the used with open()
- Notes:
  - The “w” modes will not cause truncation.
  - The file position is not changed.
  - Closing the stream will close the file descriptor as well.
- ***Returns***
  - a valid FILE* on success otherwise NULL
- To obtain the file descriptor for a FILE* stream, use

```
int fileno(FILE *stream);
```

- ***Returns***
  - a valid file descriptor or –1 on failure and sets errno = EBADF.

Unix: Opening Files

```
#include <fcntl.h>
int open(char *name, int flags);
int open(char *name, int flags, mode_t mode);
```

- ***Returns***
  - an integer file descriptor if successful or –1 on error
- ***Mode*** – any combination of the following (using |)
  - S_IRUSR - owner has read permission
  - S_IWUSR - owner has write permission
  - S_IXUSR - owner has execute permission
  - S_IRGRP - group has read permission
  - S_IWGRP - group has write permission
  - S_IXGRP - group has execute permission
  - S_IROTH - others have read permission
  - S_IWOTH - others have write permission
  - S_IXOTH - others have execute permission

Unix: Opening Files - Flags

- ***Open*** – any "|" combination of the following
  - O_APPEND - add to the end of the file
  - O_RDONLY - open only for reading
  - O_WRONLY - open only for writing
  - O_RDWR - open for both reading and writing
  - O_CREAT - create the file if it doesn't exist
  - O_CREAT | O_EXCL - generate an error if file already exists
  - O_TRUNC - if file already exists set length to 0
  - O_NOCTTY - if it is a device don't let it be the controlling terminal for the current process
  - O_NONBLOCK - sets nonblocking mode if possible
  - O_DSYNC - make write wait for physical completion
  - O_RSYNC - make reads wait until all writes are done
  - O_SYNC - same as DSYNC but also waits for attributes to be updates such as size & modification time
- **Different version of UNIX have different open flags**

Blocking and Non-Blocking IO

- Blocking
  - When you attempt to read from a file and the data is not available straight away the function will wait for the data to be ready.
  - During this time the function goes to sleep
  - This is the default operation of most functions
- Non-blocking
  - You can control whether you want the function call to return immediately and report that it failed
  - You can do this by using the O_NONBLOCK
  - A errno of EAGAIN means to try again later

Unix: Creating Files

`int creat(char *name, mode_t mode);`

- This is equivalent to
  - **Open(name, O_WRONLY | O_CREAT | O_TRUNC, mode);**

```
#include <sys/stat.h>       //- for mode masks
#include <fcntl.h>          //- for open()
…
mode_t mode = S_IRUSR | S_IWRUSR | S_IRGRP;

int fd1 = open ("/etc/hosts", O_RDONLY);
int fd2 = open("myfile.txt", O_CREAT|O_TRUNC,  mode);
int fd3 = creat("wsfile.txt", mode);

if (fd1 < 0) printf("Can't open /etc/hosts");
if (fd2 < 0) printf("Cant't create myfile.txt");
if (fd3 < 0) printf("Cant't create wsfile.txt");
```

Unix: Closing Files

```
#include <unistd.h>
int close(int fd);
```

- ***Returns***:
  - 0 if successful otherwise it returns -1, and sets errno appropriately.
- Notes:
  - Closing a file does not mean that any data you have written to the file has been committed (flushed) to the disk. To make sure the data has been committed to disk needs to be sync-ed
  - Do not try to close an already closed file
- Example

```
int fd = open("/etc/hosts", O_RDONLY);
if (fd < 0) printf("Can't open /etc/hosts");
else if ( close(fd) < 0 )
    printf("bad file descriptor");
```

Unix: Reading Files

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t len);
```

- ***buf*** – a pointer to preallocated buffer that is to receive the data must be equal to len bytes in length
- ***len*** – the number of bytes to read from fd
- ***Returns***
  - the number of bytes actually read or –1 on error and errno is appropriately set.
  - The file position is advanced by the number of bytes read
- The number of bytes read may be < len because;
  - An end of file (EOF) is reached
  - The program was interrupted or an IO error occurred
  - No data is currently available to be read

Unix: Reading Files Example

```
char buf[512];
long word;

int fd = open("myfile.c", O_RDONLY);;

int ret = read(fd, &word, sizeof(long));
if (ret < 0) printf("error reading long");

ret = read(fd, buf, sizeof(buf));
if (ret < 0) printf("error reading file");

close(fd);
```

Unix: Writing Files

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t len);
```

- ***buf*** – a pointer to data that is to be written
- ***len*** – the size of buf in bytes
- ***Returns***
  - the number of bytes actually written or –1 on error and errno is appropriately set.
  - The file position is advanced by the number of bytes written
- If the O_SYNC flag wasn't used on open() you can…

```
int fsync(int fd)          //- sync data and attribs
int fdatasync(int fd)      //- sync data only
int sync()                 //- sync all open files
```

- ***Returns*** – 0 if successful otherwise –1 and set errno

Example: Std Device IO

- //- Copying standard input to standard output
- //- one byte at a time.

```
int main(void)
{
    char c;
    while( read(STDIN_FILENO, &c, 1) != 0)
    write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

Example: cp

- The cp command creates or truncates a file and then copies data into it
  - %cp srcfile dstfile
- Process
  - Open source file
  - Create copy file
  - Read source →buffer
  - Write buffer → copy file
  - Close source
  - Close copy
- The bigger the buffer the less loop iterations
- Try to minimise changing from user mode code to kernel mode code as changing over is slow

Example: cp

- File creation

```
int fd = creat(name, 0644);
```

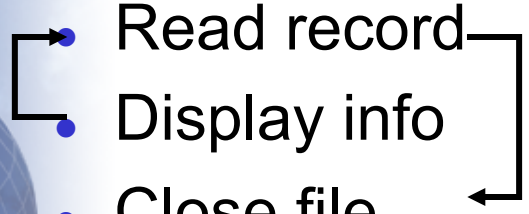
  - Creates or truncates a file
  - Then opens the file for writing
  - If create sets the mode to 0644
  - Returns –1 on error otherwise a file descriptor
- File Writing

```
int n = write(fd, buffer, num);
```

  - Writes num chars to file
  - Returns number of chars sent or –1 on error

Example: Who

- Who is a command for seeing who is logged in
- Outputs: login name, time, terminal, location
- Where to get more information about who?
  - Read man pages: man who
  - Search man pages : man –k user | more
  - Read the .h files in: /usr/include
- Who works by
  - Open utmp (or wtmp)
  - Read record
  - Display info
  - Close file



Example: Who – Reading Files

- We need to use open(), read() and close()

```
int fd = open(name, mode);
```

  - Mode = O_RDONLY
  - Returns –1 on error or an into on success

```
int n = read(fd, array, numchars);
```

  - Returns number of chars read or –1 on error

```
close(fd);
```

  - Closes the file

Example: Who – Basic Code

```
#include <stdio.h>
#include <utmp.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
   struct utmp     record;
   int             utmpfd;
   int reclen = sizeof(record);
   if (utmpfd = open(UTMP_FILE, O_RDONLY)) == -1)
   {
       perror(UTMP_FILE);          //- defined in utmp.h
       exit (-1);
   }
   while (read(utmpfd, &record, reclen) == reclen)
       print_info(&record);
   close (utmpfd);
   return 0;
}
```

Example: Who – Tidying it up

- Need to write the function print_info
  - Suppress blank entries
  - Format the time display
- Check the man pages/textbook to find out how
  - ut_type (utmp.h)
  - char *ctime(const time_t*) - convert time to a string
- Speed it up by reducing the number of system calls as changing from user mode code to kernel mode code is slow
  - Use buffering – read a bunch of records in at a time and then process them one by one

Unix File Seeking

```
#include <unistd.h>
off_t lseek(int fd, off_t pos, int origin);
```

- ***pos*** – the position to set the file pointer relative to the origin
- ***Origin*** – can be one of
  - SEEK_SET - move pos bytes from the start of the file
  - SEEK_CUR - move pos bytes from the current position
  - SEEK_END - move pos bytes from the end of the file
- ***Returns*** – returns the new file position if successful, otherwise returns –1 and sets errno as appropriate
- If you seek past the end of a file and write to it, the "hole" in between will be padded with zeros, but these zeros will not take up any disk space

Unix: Truncating Files & fctl

```
#include <unistd.h>
int ftruncate(int fd, off_t len);
int truncate(char *path, off_t len);
```

- Returns: 0 on success, otherwise -1, and set errno as appropriate.

- The fctl function is used to change a files' properties

```
#include <fcntl.h>
int fcntl(int fd, int cmd, …);
```

- ***Cmd*** – is one of
  - F_DUPFD - duplicate existing file descriptor
  - F_GETFD, F_SETFD - Get/Set descriptor flag FD_CLOEXEC
  - F_GETFL, F_SETFL - Get/Set file status flags (eg O_RDWR)
  - F_GETOWN, F_SETOWN, - Get/Set asynchronous IO ownership
  - F_GETLK, FSETLK, F_STLKW – Get/Set record locks

dup, dup2

- Dup is used to create duplicate connections to already open files –
  - Dup always returns the lowest free descriptor
- Mainly used for redirecting stdio IO
  - first close the stdin or stdout (0 and 1) file descriptors
  - then doing a dup() on another filedescriptor will map it to the lowest available file descriptor, which is now 0 or 1.

```
#include <unistd.h>
int dup(int fd);
int dup(int oldfd, int newfd);
```

Blocking and Multiplexed IO

- Multiplexed I/O allows a program to concurrently block on multiple file descriptors, and receive notification when any one of them becomes ready to read or write.
- Basic program logic:
  1. Tell me when any of these file descriptors are ready for I/O.
  2. Go to sleep until one or more file descriptors are ready.
  3. If Woken up: What is ready?
  4. Handle all file descriptors ready for I/O, without blocking.
  5. Go back to step 1, and start over.
- Multiplexed I/O solutions:
  - select()
  - pselect() – similar to select but uses nanosec timer
  - poll()

Blocking and Multiplexed IO

```
#include <sys/select.h>
int select(int maxfd1, fd_set* readfds,
           fd_set* writefds, fd_set* exceptfds,
           struct timeval* tvptr);
```

- **Returns** – number of descriptors ready to read
- ***maxfd1*** – the valued of the highest fd + 1
- ***readfds, writefds, exceptfds*** – pointers to fd_set variables that is manipulated using one of:
  - FD_CLR(int fd, fd_set *set);
  - FD_ISSET(int fd, fd_set *set); //- on return see if ready
  - FD_SET(int fd, fd_set *set); //- wait for given fd
  - FD_ZERO(fd_set *set); //- initialise the fd_set
- ***tvptr –*** pointer to timeval structure, (how long to wait)
  - if NULL then wait infinitively, otherwise wait for specified amount of seconds before timing out

Blocking and Multiplexed IO

```
#include <poll.h>
int poll(struct pollfd fdarray[], nfds_t nfds,
         int timeout);
```

- Similar to select() but used pollfd structure

```
struct pollfd {
    int fd;          //- the fd to check or <0 to ignore
    short events;    //- events to watch
    short revents;   //- events that have occurred (on return)
}
```

  - ***Events can be one of (see text or man page for details)***
  - ***POLLIN, POLLRDNORM, POLLTDBAND, POLLPRI***
  - ***POLLOUT, POLLWDNORM, POLLWRBAND***
  - ***Revents also include: POLLERR, POLLHUP, POLLNVAL***
- ***fdarray –*** an array of pollfd structures with one element per fd
- ***nfds*** – the number of elements/descriptors in fdarray
- ***Timeout*** – time in millisecs wait forever if < 0, don't wait if 0

Unix: Directory Control

- Directory functions

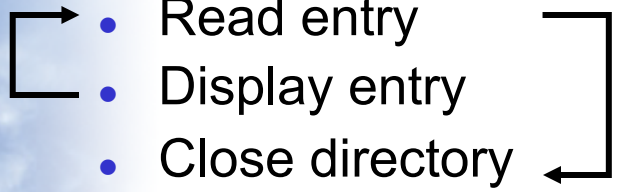
```
#include <dirent.h>
int  mkdir(const char* path, mode_t mode)
int  rmdir(const char* path);
int  chdir(const char* path);
int  fchdir(int fd);
char* getcwd(char* bug, size_t size);
```

- Directory access

```
DIR* opendir(const char *name);
int  closedir (DIR *dir);
int  dirfd(DIR *dir);
struct dirent* readdir(DIR *dir);
void rewinddir(DIR* dp);
void telldir(DIR* dp);
void seekdir(DIR* dp, long loc);
```

Example: ls

- The ls command displays the contents of directories and lists the file properties
  - ls ls –l ls -a
  - ls /tmp ls –l /tmp ls -s
  - ls main.c ls –l main.c ls –lu
- Process
  - Open Directory
  - Read entry
  - Display entry
  - Close directory



- A directory is just a special type of file
- Could use open(), read(), close() but the format of a directory file is different for different systems so use opendir(), readdir, closedir() instead
- Need to use #include <dirent.h>

Example: ls – Basic Code

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
int main(int argc, char* argv[])
{
   if (argc == 1) do_ls(".")
   else while (--argc)
   {
        printf("%s:\n", *++argv);
        do_ls(*argv);
   }
}
void do_ls(char dirname[])
{
   DIR              *dir_ptr;
   struct dirent    *direntp;
   if ((dir_ptr = opendir(dirname)) == NULL)
        fprintf(stderr, "ls: can't open %s\n", dirname);
   else {
        while ((dirent = readdir(dir_ptr))!= NULL)
               printf("%s\n", dirent->d_name);
        closedir(dir_ptr);
   }
}
```

Example: ls – Improvements

- Problem the file attributes are not sorted
  - Use qsort() on the array of entries
  - No attributes are provided
- To get attributes we need to use stat()
  - stat(name, pointer to stat structure)
- stat also gives us the uid
  - To convert uid to a name we use getpwuid()
  - This returns a ptr to a struct with user info

Unix: Get File Attributes

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

- Get attributes for specified file

```
int stat(const char*, struct stat*)
```

- Get attributes for an open file

```
int fstat(int fd, struct stat*)
```

- Get attributes for a symbolic link

```
int lstat(const char*, struct stat*)
```

- **Example**

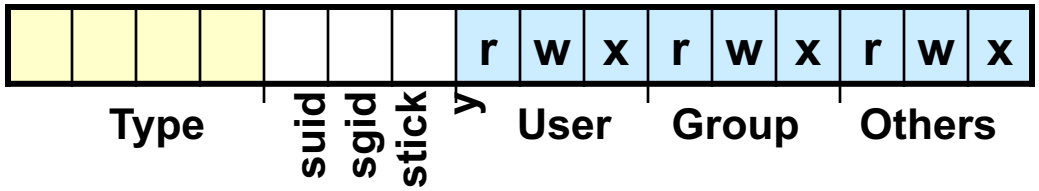
```
struct stat status
int ret = stat("myfile.c" &status);
if (ret < 0) printf("can't get stat");
```

Unix: struct stat

```
struct stat
{
   dev_t     st_dev;         //- ID of device
   containing file
   ino_t      st_ino;        //- inode number
   mode_t  st_mode; //- permissions
   nlink_t   st_nlink;       //- number of hard links
   uid_t      st_uid;        //- user ID of owner
   gid_t      st_gid;        //- group ID of owner
   dev_t     st_rdev;        //- device ID (if special
   file)
   off_t       st_size;      //- total size in bytes
   blksize_t st_blksize; //- blocksize for filesystem
   I/O
   blkcnt_t  st_blocks; //- number of blocks
   allocated
   time_t    st_atime;   //- last access time
   time_t    st_mtime;  //- last modification time
   time_t    st_ctime;        //- last status change
   time
};
```

Unix: Get File Permissions

- st_mode is a 16 bit value



- Bit fields are examined using binary masks
- Use bitwise AND “&” to mask the bits you want

Value	101	101	001
Mask	000	111	000
Result	000	101	000
	usr	grp	oth

- Example Code
  - if (ST_MODE & 0x04) printf(“readable by others”);
  - if (ST_MODE & 0x02) printf(“writable by others”);
- Masks to check file type are in <sys/stat.h>

Unix: Get File Type

- You can use predefined macros int <Stat.h> to examine the file type
  - S_ISREG() - regular file
  - S_ISDIR() - directory file
  - S_ISCHR() -character special file
  - S_ISBLK() - block special file
  - S_ISFIFO() - FIFO
  - S_ISLNK() - symbolic link
  - S_ISSOCK() - socket
- Example

```
struct stat sb;
int fd = open (“test.c”, O_RDONLY);
if (fstat (fd, &sb) == -1) exit(01);
if (S_ISREG (sb.st_mode)) printf(“it is a regular file”);
```

Unix: Testing File Access

```
#include <unistd.h>
int access(const char* fname, int mode)
```

- ***mode*** – any "|" combination of any of the following
  - R_OK - can be read
  - W_OK - can be written
  - X_OK - can be executed
  - F_OK - file exists?
- Set the default file mode creating mask that will be used when any file is created

```
mode_t umask(mode_t cmask)
```

- **cmask** – any "|" combination of access mode
  - Uses S_IRUSR, S_ IWGRP etc
  - Any bits that are on in the mask are turned off in the file mode

Unix: Setting File Access & Owner

- These are used to change the file permissions on existing files

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

- ***mode*** - any combination of access mode flags
- Changing the user id

```
int chown(char* fname, uid_t usr, gid_t grp)
int fchown(int fd, uid_t usr, gid_t grp)
int lchown(char* fname, uid_t usr, gid_t grp)
```

- ***Usr*** – user id unchaged if value = -1
- **Grp** - group id unchaged if value = -1

Unix: File System Control

- Change modification and access time

```
int utime(char* path, stuct utimbuf* time);
```

- Renaming and Removing Files

```
int remove(char* path);        //- same as unlink
int rename(char* name, char* newname);
```

- Links

```
Int link(char* path, char* newpath);
int unlink(char* path);//- remove ANY dir entry
Int symlink(char* path, char* linkname);
Int readlink(char* path, char* buf, size_t num);
```

Unix File Locking

- There are different kinds of file locking mechanisms in UNIX to coordinate shared file access
- The two most common mechanisms are fcntl and flock

```
#include <stdio.h>

void flockfile (FILE *stream);

void funlockfile (FILE *stream);
```

- File locks under UNIX are by default not enforced. This means that cooperating processes may use locks to coordinate access to a file among themselves, but programs are also free to ignore locks and access the file in any way they choose to.


Win32-POSIX File IO

Windows Supports POSIX.1

- The POSIX API is mapped onto the Win32 API
- Windows prefixes the function names with an "_" automatically if you forget to do so
- There are also UNICODE versions starting with "_w"
- The parameters may be a little different
  - int _close(int fd) – Close file
  - int _commit(int fd) – Flush file to disk
  - int _creat(char* str, int mode) – Create file
  - int _dup(int fd) – duplicates fd
  - int _eof(int fd) – Test for end of file
  - int _lseek(int fd, long offset, int origin) – move file pointer
  - int _open(char* str, int flag, int mode) – Open file
  - int _read(int fd, void* buf, int cout) – Read data from fd
  - long _tell (int fd) – Get file-pointer position
  - int _umask(int mode), – Set file-permission mask
  - int _write (int fd, void* buf, int cout) – Write data to file

Windows POSIX.1 File Access

- int _chsize(int fd, long size); - Change file size
- long _filelength(int fd); - Get file length
- int _fstat(int fd, struct _stat *buf); - Get file-status
- int _isatty(int fd); - Check if char device
- int _locking(int fd, int mode, long nbytes); - Lock parts of file
- int _setmode (int fd, int mode) - Set file translation mode
- int _access(char *path, int mode); - Check file permissions
- int _chmod(char *path, int mode); - Change file permissions
- int remove (char *path) - Delete file
- int rename (char *olds, char *news) - Rename file
- int _stat(const char *path, struct _stat *buffer); - get status
- int _umask(int mode), - Set default permissions
- int _unlink(char *path); - Delete file

Windows POSIX.1 Directory Access

- int _chdir(char* dirname); - Change current directory
- int _chdrive(int drive); - Change current drive
- int _mkdir(char* path), - Make new directory
- int _rmdir(char* path), - Remove directory
- void _searchenv(char *filename, char *varname, char *buffer);
  - Search for file using environment paths
- char* _getcwd(char* buf, int len),
  - Get current working directory for default drive
- char* _getdcwd(int drive, char* buf, int len),
  - Get current working directory for specified drive

Windows POSIX.1 Conversions

- Creates a FILE* stream handle for a file that was opened as a File Descriptor

```
FILE* _fdopen(int fd, char* mode);
```

- Gets the 'C' file descriptor associated with a stream.

```
int _fileno(FILE* stream)
```

- Gets the OS file handle associated with existing C run-time file descriptor

```
long _get_osfhandle(int fd)
```

- Associates C run-time file descriptor with an existing operating-system file handle

```
int _open_osfhandle (int fd, flags)
```

Windows POSIX.1 Extra

- Additional path manipulation functions

char* _fullpath(char *absPath, char *relPath, size_t len);

- Expand a relative path to its absolute path name

void _makepath(char *path, char *drive, char *dir,
char *fname, char *ext);

- Create a pathname from separate components

void _splitpath(char *path, char *drive, char *dir,
char *fname, char *ext);

- Parse path into components

char* _mktemp(char *template);

- Create unique filename from a template string


Win32 File IO

Basic Win32 File Functions

- CreateFile
- ReadFile
- WriteFile
- CloseHandle
- DeleteFile
- CopyFile
- MoveFile
- GetFileAttributes
- Temporary File Names

Windows File Names

- Full pathname can start with a drive name (A:, C:)
- Or with a “share” name (\\servername\sharename)
- Pathname separator is a backslash — \
  - You can also use / in C
- Directory and file names are case insensitive but they are case retaining
- Directory and file names cannot use ASCII 1–31 or any of < > : " | but you can have blanks
- Cannot be one of several reserved words:
  - CON, PRN, AUX, CLOCK$, NUL, COMx, LPTx(x = 1 to 9)

Windows File Names

- File and directory names can be up to 255 characters long
- Pathnames are limited to MAX_PATH (260)
- Prefix the pathname with \\?\to allow for very long filenames (up to 32K)
- A period (.) separates a file's name from its extension
- A File name can have more than one period (.)
- A single period (.) and two periods (..) as directory names indicate the current directory and it's parent

Creating Files

```
HANDLE CreateFile(
        LPCTSTR lpFileName,
        DWORD dwDesiredAccess,
        DWORD dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD dwCreationDisposition,
        DWORD dwFlagsAndAttributes,
        HANDLE hTemplateFile);
```

- **Returns: A `HANDLE` to an open file object, or *INVALID_HANDLE_VALUE* in case of failure**

CreateFile Parameters – 1

```
HANDLE CreateFile(
                LPCTSTR lpFileName,
                DWORD dwDesiredAccess,
                DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
```

lpFileName

- Pointer to a null-terminated string of the file, pipe, or other named object to open or create

dwDesiredAccess

- Access using GENERIC_READ or GENERIC_WRITE
- Note: Combine flags with |

CreateFile Parameters – 2

```
HANDLE CreateFile(
                LPCTSTR lpFileName,
                DWORD dwDesiredAccess,
                DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
```

dwShareMode

- 0 - Cannot be shared; not even this process can open another handle
- FILE_SHARE_READ - Others can read concurrently
- FILE_SHARE_WRITE - Others can write concurrently

lpSecurityAttributes

- Pointer to security descriptor
- Set to NULL – every one has full control over object

CreateFile Parameters – 3

```
HANDLE CreateFile(
                LPCTSTR lpFileName,
                DWORD dwDesiredAccess,
                DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
```

dwCreationDisposition

- CREATE_NEW — Fails if the file exists
- CREATE_ALWAYS — An existing file will be overwritten
- OPEN_EXISTING — Fail if the file does not exist
- OPEN_ALWAYS — Open it or create it if it doesn't exist
- TRUNCATE_EXISTING — File length will be set to zero

CreateFile Parameters – 4

```
HANDLE CreateFile(
                LPCTSTR lpFileName,
                DWORD dwDesiredAccess,
                DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
```

dwFlagsAndAttributes

- Specifies up to 16 file attributes and flag attributes
- These are ignored when an existing file is opened
  - `FILE_ATTRIBUTE_NORMAL` — No other attributes are set
  - `FILE_ATTRIBUTE_READONLY` — Cannot write or delete
  - `FILE_FLAG_OVERLAPPED` — For asynch I/O
  - `FILE_FLAG_DELETE_ON_CLOSE`
  - `FILE_FLAG_NO_BUFFERING`
  - `FILE_FLAG_SEQUENTIAL_SCAN` and `FILE_FLAG_RANDOM_ACCESS` provide performance *hints*

CreateFile Parameters – 5

```
HANDLE CreateFile(
                LPCTSTR lpFileName,
                DWORD dwDesiredAccess,
                DWORD dwShareMode,
                LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                DWORD dwCreationDisposition,
                DWORD dwFlagsAndAttributes,
                HANDLE hTemplateFile);
```

hTemplateFile

- Sets the attributes of the new file to be the same as those of an existing file
- Normally set to NULL
- Handle of an open file that specifies extended attributes to apply to a newly created file
- Current file is open as GENERIC_READ

Closing Files

```
BOOL CloseHandle(HANDLE hObject)
```

- **Return**: `TRUE` if the function succeeds; `FALSE` otherwise
- *This function is general purpose and is used to close handles for nearly all object types*

Reading Files

```
BOOL ReadFile(HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead,
              LPOVERLAPPED lpOverlapped)
```

- Starts at current file position (for the handle) and advances the position by num chars read
- **Returns**:
  - `TRUE` if the read succeeds even if no bytes were read due to an attempt to read past the end of file
  - `FALSE` indicates an invalid handle, or other invalid parameter.

ReadFile Parameters

- **hFile**
  - File handle with `GENERIC_READ` access
- **lpBuffer**
  - Memory buffer to receive the input data
- **nNumberOfBytesToRead**
  - Number of bytes you expect to read
- ***lpNumberOfBytesRead**
  - Returns actual number of bytes transferred
  - Return value of Zero indicates end of file
- **lpOverlapped**
  - Points to `OVERLAPPED` structure (`NULL` for now)

Writing Files

```
BOOL WriteFile(HANDLE hFile,
    LPCVOID *lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped)
```

- **Return**: `TRUE` if the function succeeds; `FALSE` otherwise or if it is completing asynchronously
- Success does not necessarily mean data was actually written to disk. Use **FILE_FLAG_WRITE_THROUGH** with **CreateFile** to force it is written synchronously to disk
- If handle is at the end of the file, Windows will extend the length of the file

File Pointers

- The system keeps track of where a file was last read from or written to using a 64 bit File Pointer (FP)
- When a file is first opened, the file pointer is set to the beginning of the file, which is offset zero (FP=0)
- Each read/write advances the file pointer by the number of bytes being read or written
- For example, if the file pointer is at 0, and a read request of 5 bytes is made, then FP = 5 immediately after the read
- When the file pointer reaches the end of a file and a process attempts to read from the file no error occurs, but no bytes are read

Random File Access

```
DWORD SetFilePointer(HANDLE hFile,
                     LONG lDistanceToMove,
                     PLONG lpDistanceToMoveHigh,
                     DWORD dwMoveMethod);
```

- **hFile**
  - The handle of and open file. This handle must be created with the GENERIC_READ or GENERIC_WRITE access rights
- **lDistanceToMove**
  - Positive values move forward, negative move backwards
- **lpDistanceToMoveHigh**
  - NULL if 32 bit move is enough, otherwise a pointer to the high 32 bits of a signed 64 bit distance
- **dwMoveMethod**
  - FILE_BEGIN - move relative to the start of the file
  - FILE_CURRENT - move relative to the current position
  - FILE_END - move relative to the end of the file
- If the function succeeds, the return value is the value of the new file pointer. Otherwise, the return value is an error code.

Random File Access – 2

```
BOOL SetFilePointerEx(HANDLE hFile,
    LARGE_INTEGER lDistanceToMove,
    PLARGE_INTEGER lpNewFilePointer,
    DWORD dwMoveMethod);
```

- This function is just for moving the file pointer in huge files that require 64 bit distances
- **Return:** If a function succeeds, the return value is nonzero. Otherwise, the return value is zero

```
typedef union _LARGE_INTEGER
{
    struct {
        DWORD LowPart;  //- 32 bit
        LONG HighPart;  //- 32 bit
    };
    LONGLONG QuadPart;  //- 64 bit
} LARGE_INTEGER, *PLARGE_INTEGER;
```

Asynchronous or Overlapped IO

- With Overlapped IO, ReadFile and WriteFile are nonblocking, they return immediately before the read or write operation is finished.
- To do Overlapped IO the file must be created with FILE_FLAG_OVERLAPPED and the lpOverlapped parameter must point to an Overlapped structure

```
typedef struct _OVERLAPPED  //- simplified
{
    DWORD Offset;          //- Start File Posn low word
    DWORD OffsetHigh       //- Start FilePosn High word
    HANDLE hEvent;         //- Event to be triggered when done
} OVERLAPPED;
```

  - hEvent is an event that will be set to the signaled state when the operation is done. Must be set to zero or a valid event handle created using CreateEvent()
  - More about events in another lecture

Blocking Overlapped Read/Write

- If a file was not created with FILE_FLAG_OVERLAPPED but the lpOverlapped parameter is not NULL then the operation will block until it is done.

```
RECORD r;                          //- some arbitrary data structure
DWORD nRd, nWrt;
LARGE_INTEGER FilePos;
OVERLAPPED ov = { 0, 0, 0, 0, NULL };

//- Update data in the 5000'th record.
FilePos.QuadPart = 5000 * sizeof (RECORD);
ov.Offset = FilePos.LowPart;
ov.OffsetHigh = FilePos.HighPart;
ReadFile(hFile, &r, sizeof(RECORD), &nRd, &ov);
r.data = "Whatever";               //- Update the record.
WriteFile(hFile, &r, sizeof(RECORD), &nWrt, &ov);
```

Deleting Files

```
BOOL DeleteFile(LPCTSTR lpFileName)
```

- **lpFileName** - name of an existing file to delete
- Returns:
  - **ERROR_INVALID_NAME**, in case of a non-existing file
  - **ERROR_ACCESS_DENIED**, in case of a read-only file
- You can not delete an open file
- In the ANSI version of this function, the name is limited to `MAX_PATH` characters. To extend this limit to 32,767 wide characters, call the Unicode version of the function and prepend "\\?\" to the path.

Copying Files

```
BOOL CopyFile(LPCTSTR lpExistingFile,
              LPCTSTR lpNewFileName,
              BOOL fFailIfExists)
```

- if **`fFailIfExists`** is **`TRUE`** the function will fail if a file with the new name already exists
- The new file preserves the old file's attributes, including time stamps
- If the function succeeds, the return value is non-zero. Otherwise, the return value is zero.

CopyFile Example

```
int _tmain(int argc, LPTSTR argv[])
{
  if(argc !=3)
  {
     printf("Usage: cpWC inputfile outputfile\n");
     return 1;
  }
  if (!CopyFile(argv[1], argv[2], FALSE))
  {
     printf("CopyFile Error: %d\n", GetLastError());
     return 2;
  }
  return 0;
}
```

Creating Links

```
BOOL CreateHardLink(LPCTSTR lpFileName,
                    LPCTSTR lpExistingFileName,
                    LPSECURITY_ATTRIBUTE lpSecurityAttribute);
```

- Create a hard link to an existing file (this is not the same as the shortcuts created by the shell)
- Set *lpSecurityAttribute* to NULL for now
- Only supported on the NTFS file system
- Returns non-zero (TRUE) if the function succeeds, otherwise returns zero (FALSE).
- **`CreateSymbolicLink()`** – supported in Vista, Win7 but not XP

Moving / Renaming Files

```
BOOL MoveFile(LPCTSTR lpExistingFileName,
              LPCTSTR lpNewFileName);
BOOL MoveFileEx(LPCTSTR lpExistingFileName,
              LPCTSTR lpNewFileName,
              DWORD dwFlags);
```

- For MoveFile the source and target files must be on the same drive otherwise use MoveFileEx
- ***lpExistingFileName***
  - Name of existing file or directory
- **`dwFlags` specify how the file is moved**
  - `MOVEFILE_REPLACE_EXISTING`
  - `MOVEFILE_WRITETHROUGH`
  - `MOVEFILE_DELAY_UNTIL_REBOOT`

File Attributes

```
DWORD GetFileAttributes(LPCTSTR lpFileName);
```

- The return value is a binary combination of the flags:
  - 0x0001 = FILE_ATTRIBUTE_READONLY
    0x0002 = FILE_ATTRIBUTE_HIDDEN
    0x0004 = FILE_ATTRIBUTE_SYSTEM
    0x0010 = FILE_ATTRIBUTE_DIRECTORY
    0x0020 = FILE_ATTRIBUTE_ARCHIVE
  - 0x0040 = FILE_ATTRIBUTE_DEVICE
    0x0080 = FILE_ATTRIBUTE_NORMAL
    0x0100 = FILE_ATTRIBUTE_TEMPORARY
    0x0200 = FILE_ATTRIBUTE_SPARSE_FILE
    0x0400 = FILE_ATTRIBUTE_REPARSE_POINT
    0x0800 = FILE_ATTRIBUTE_COMPRESSED
    0x1000 = FILE_ATTRIBUTE_OFFLINE
    0x2000 = FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
    0x4000 = FILE_ATTRIBUTE_ENCRYPTED
    0x10000 = FILE_ATTRIBUTE_VIRTUAL

File Attributes

```
BOOL GetFileAttributesEx(LPCTSTR lpFileName,
          GET_FILEEX_INFO_LEVELS fInfoLevelId,
          LPVOID lpFileInformation );
```

- **fInfoLevelId**
  - This must be GetFileExInfoStandard
- *lpFileInformation*

  This parameter is a **WIN32_FILE_ATTRIBUTE_DATA** structure

```
BOOL WINAPI SetFileAttributes(LPCTSTR lpFileName,
                              DWORD dwFileAttributes );
```

- **dwFileAttributes – any combination of**
  - 0x0001 = FILE_ATTRIBUTE_READONLY
    0x0002 = FILE_ATTRIBUTE_HIDDEN
    0x0004 = FILE_ATTRIBUTE_SYSTEM
    0x0020 = FILE_ATTRIBUTE_ARCHIVE
  - 0x0080 = FILE_ATTRIBUTE_NORMAL
    0x0100 = FILE_ATTRIBUTE_TEMPORARY
    0x1000 = FILE_ATTRIBUTE_OFFLINE
    0x2000 = FILE_ATTRIBUTE_NOT_CONTENT_INDEXED

File Size Attribute

```
DWORD GetFileSize(HANDLE hFile,
                  LPDWORD lpFileSizeHigh);
```

- Return: If the function succeeds, the return value is the low-order 32 bit word of the file size. Otherwise, the return value is INVALID_FILE_SIZE

```
BOOL GetFileSizeEx(HANDLE hFile,
                   PLARGE_INTEGER lpFileSize);
```

- Return: If a function succeeds, the return value is nonzero. Otherwise, the return value is zero

```
DWORD GetCompressedFileSize(LPCTSTR lpFileName,
                  LPDWORD lpFileSizeHigh);
```

- **DOES NOT NEED AN OPEN HANDLE**

Resizing Files

- Each file stream has the following
  - File size: the size of the data in a file in bytes
  - Allocation size: the size of the disk space allocated for a file
  - Valid data length: the length of data in a file that is actually written. This value <= the file size

```
BOOL WINAPI SetEndOfFile(HANDLE hFile);
```

- This function resizes (physical) a file based on the current file pointer – either extend or truncate

```
BOOL SetFileValidData(HANDLE hFile,
                      LONGLONG ValidDataLength);
```

- Sets the logical end of the file that must be greater than the current valid data length, but less than the current file size
- Used to avoid zeroing data when extending the file length – leaving random data in the file

Win32 Time Structures / Formats

- SYSTEMTIME
  - Year, month, day, hour, minute, second, and millisecond taken from internal hardware clock
- FILETIME (64 bit int)
  - Stored as 100 nanosecond intervals since January 1, 1601
- Local time
  - A SYSTEMTIME or FILETIME converted to the system's local time zone
- Several Time-related Functions to convert formats
  - `FileTimeToSystemTime()`
  - `SystemTimeToFileTime()`
  - `CompareFileTime()`
  - `FileTimeToLocalFileTime()`
  - `SystemTimeToTzSpecificLocalTime()`

FileTime Attributes

```
BOOL GetFileTime(HANDLE hFile,
                 LPFILETIME lpftCreation,
                 LPFILETIME lpftLastAccess,
                 LPFILETIME lpftLastWrite);

BOOL SetFileTime(HANDLE hFile,
                 LPFILETIME lpftCreation,
                 LPFILETIME lpftLastAccess,
                 LPFILETIME lpftLastWrite);
```

- **Returns: If the function succeeds, the return value is nonzero. Otherwise, the return value is zero.**

GetFileTime Example – 1

```
BOOL  GetLastWriteTime (HANDLE hFile, LPTSTR lpszString)
{
   FILETIME  ftCreate, ftAccess, ftWrite, ftLocal;
   SYSTEMTIME stUTC;
   //-- Get the file's times info.
   if (!GetFileTime(hFile, &ftCreate, &ftAccess, &ftWrite))
         return FALSE;
   //-- Convert the last-write time to local time.
   FileTimeToLocalFileTime(&ftWrite, &ftLocal);
   FileTimeToSystemTime(&ftLocal, &stUTC);
   //-- Build a string showing the date and time.
   sprintf(lpszString, "%02d/%02d/%d %02d:%02d",
                  stUTC.wMonth, stUTC.wDay,
                  stUTC.wYear, stUTC.wHour, stUTC.wMinute);
   return TRUE;
}
```

GetFileTime Example – 2

```
//-- Get current system time and convert to a file time.
SYSTEMTIME SysTime;
FILETIME NewFTime, *pAccessed, *pModified;
GetSystemTime (&SysTime);
SystemTimeToFileTime(&SysTime, &NewFTime);
if (SetAccessTime)  pAccessed = &NewFTime;
if (SetModTime)     pModified = &NewFTime;

//-- Do not change the create time.
if (!SetFileTime(hFile, NULL, pAccessed, pModified))
   printf("SetTime Failed");
```

File Type Attribute

```
DWORD GetFileType(HANDLE hFile);
```

- The return value is one of the following
  - FILE_TYPE_CHAR
  - FILE_TYPE_DISK
  - FILE_TYPE_PIPE
  - FILE_TYPE_REMOTE
  - FILE_TYPE_UNKNOWN

File Attributes

```
HANDLE FindFirstFile(LPCTSTR lpFileName,
                     LPWIN32_FIND_DATA lpffd)
```

- **lpSearchFile**
  - The filename you want to get attributes for
  - Wildcards are OK (`*` and `?`)
- **lpffd**
  - Points to a **WIN32_FIND_DATA** structure that will be filled in
- Returns: A "search handle"
  - **INVALID_HANDLE_VALUE** indicates failure

```
BOOL FindNextFile(HANDLE hFindFile,
                  LPWIN32_FIND_DATA lpffd)
```

- Returns `FALSE` when no more files can be found

```
BOOL FindClose(HANDLE hFindFile)
```

WIN32_FIND_DATA

```
typedef struct  _WIN32_FIND_DATA
{
  DWORD dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD nFileSizeHigh;
  DWORD nFileSizeLow;
  DWORD dwReserved0;
  DWORD dwReserved1;
  TCHAR cFileName[MAX_PATH];     //- file name
  TCHAR cAlternateFileName[14]; //- 8.3 name
} WIN32_FIND_DATA;
```

File Attributes Example

```
#define ErrorExit(str) { printf(str); return; }
main()
{
   WIN32_FIND_DATA FileData;
   TCHAR szDirPath[] = TEXT("c:\\TextRO\\");
   BOOL fFinished = FALSE;              //- Create a new directory.

   if (!CreateDirectory(szDirPath, NULL))
         ErrorExit("Could not create new directory.\n");
   //- Start searching for text files in the current directory.
    HANDLE hSearch = FindFirstFile(TEXT("*.txt"), &FileData);
   if (hSearch == INVALID_HANDLE_VALUE)
        ErrorExit("No text files found.\n ");

   // Copy each .TXT file to the new directory and change it to read only
   while (!fFinished)
   {
        TCHAR szNewPath[MAX_PATH];
        lstrcpy(szNewPath, szDirPath);
        lstrcat(szNewPath, FileData.cFileName);
```

File Attributes Example

```
//- from previous slide
   if (CopyFile(FileData.cFileName, szNewPath, FALSE))
   {
      DWORD dwAttrs = GetFileAttributes(FileData.cFileName);
      if (dwAttrs == INVALID_FILE_ATTRIBUTES) return;
      if (!(dwAttrs & FILE_ATTRIBUTE_READONLY))
         SetFileAttributes(szNewPath, dwAttrs |
                           FILE_ATTRIBUTE_READONLY);
   }
   else ErrorExit ("Could not copy file.\n");

   if (!FindNextFile(hSearch, &FileData))
   {
      if (GetLastError() == ERROR_NO_MORE_FILES)
       {
         printf("Copied all text files.\n");
         fFinished = TRUE;
      }
      else ErrorExit ("Could not find next file.\n");
   }
}
```

More File Attributes

```
BOOL GetFileInformationByHandle(HANDLE hFile,
      LPBY_HANDLE_FILE_INFORMATION lpFileInformation);

typedef struct _BY_HANDLE_FILE_INFORMATION
{
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD dwVolumeSerialNumber;
    DWORD nFileSizeHigh;
    DWORD nFileSizeLow;
    DWORD nNumberOfLinks;
    DWORD nFileIndexHigh;
    DWORD nFileIndexLow;
} BY_HANDLE_FILE_INFORMATION;
```

Temporary File Names

```
UINT GetTempFileName(LPCTSTR lpPathName,
                     LPCTSTR lpPrefixString,
                     UINT uUnique,
                     LPTSTR lpTempFileName);
```

- Used for creating a name for a temporary file
  - If *uUnique* is zero, the function attempts to form a unique file name using the current system time and will create it, if it is not zero the filename must be unique and you must create it yourself
- The name it creates is in the form of

  <path>\<pre><uuuu>.tmp
  - path is specified by the lpPathName parameter
  - pre is the first 3 letters of the *lpPrefixString* string
  - Uuuu is the Hexadecimal value of *uUnique*
- **Return:** If the function succeeds, the return value specifies the unique numeric value of the file. If fails, the return value is zero.

Setting Mount Points

```
BOOL SetVolumeMountPoint(LPCTSTR lpszVolMountPoint,
                         LPCTSTR lpseVolumeName);
```

- Used for mounting drives (volumes) at aspecified volume mount point
- For example,
  - Mount a drive (the second argument) at the first argument
  - `SetVolumeMountPoint ("C:\\mycd\\", "D:\\");`
- **Return:**
  - TRUE if the function succeeds, FALSE if it fails

Win32 Directory & Device Functions

- CreateDirectory
- RemoveDirectory
- SetCurrentDirectory
- GetCurrentDirectory

- Standard IO Devices
- IO Redirection
- Console IO
- Cheat: System Calls

Creating a Directory

```
BOOL CreateDirectory(LPCTSTR lpPathName,
        LPSECURITY_ATTRIBUTE lpSecurityAttribute);

BOOL CreateDirectoryEx(
        LPCTSTR lpTemplateDir,
        LPCTSTR lpNewDirectory,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

- *lpTemplateDir*
  - The path of a directory to copy attributes from
- Returns non-zero (TRUE) if the function succeeds, otherwise returns zero (FALSE). Possible errors are
  - **ERROR_ALREADY_EXISTS**
  - **ERROR_PATH_NOT_FOUND**

Deleting a Directory

```
BOOL RemoveDirectory(LPCTSTR lpPathName);
```

- This function marks a directory for deletion on close. The directory is not removed until the handle to the directory is closed.
- The directory must be empty
- Returns non-zero (TRUE) if the function succeeds, otherwise returns zero (FALSE).

Setting the Current Directory

```
BOOL SetCurrentDirectory(
                    LPCTSTR lpPathName);
```

- The system maintains the “current” directory on each drive for each process. All threads in a process share the current directory
- *lpPathName* –can be a relative or a full path
- Returns non-zero (TRUE) if the function succeeds, otherwise returns zero (FALSE).

Getting the Current Directory

```
DWORD GetCurrentDirectory(
                        DWORD nBufferLength,
                        LPCTSTR lpBuffer);
```

- ***lpBuffer***
  - The buffer where the absolute path of the current directory will be written to.
- ***nBufferLength***
  - The size of *lpBuffer* in number of characters. The size must be big enough to store the terminating NULL
- To determine the required buffer size, call

  DWORD reqiredsize = **GetCurrentDirectory(0, NULL);**
- **Returns:**
  - If the function succeeds, the return value is either the required buffer size or the number of chars written into the buffer, not including the terminating null character. If the function fails, the return value is zero

GetCurrent Directory Example

```
#define DIRLEN MAX_PATH + 2        //- allow CRLf

int main(int argc, char* argv[])
{
   char pwdstr[DIRLEN];
   DWORD nDirLen = GetCurrentDirectory(DIRLEN, pwdstr );

   if (nDirLen == 0)
            printf("Failure getting pathname");
   if (nDirLen > DIRLEN )
            printf("Pathname is too long");

   printf("%s\n", pwdstr );
   return 0;
}
```

Standard IO Devices

- Three “standard” devices that programs can read or write to
  - stdin – normally keyboard input
  - Stdout – normally console output
  - stderr – error reporting - normally the console
- Unlike Unix file descriptors (0,1 & 2) for these win32 requires a handle to access them
- IO on these standard devices can be redirected to other devices or to files by changing the handles associated with them
- To get a handle to the console you can use **`CreateFile()`** with the reserved pathnames
  - CONIN$ - keyboard input
  - CONOUT$ - console output

Redirecting IO

`HANDLE GetStdHandle(DWORD` *nStdHandle*`)`

- Gets the handle of the current standard device
- **`nStdHandle` – must be one of**
  - STD_INPUT_HANDLE
  - STD_OUTPUT_HANDLE
  - STD_ERROR_HANDLE

BOOL SetStdHandle(DWORD *nStdHandle*, **HANDLE** *hHandle*)

- Changes the current standard device
- **`nStdHandle`**
  - Same possible values as GetStdHandle
- **`hHandle`**
  - Open file that is to be the new standard device

Console IO

- ```
  BOOL ReadConsole(HANDLE hConsoleInput,
                   LPVOID lpBuffer,
                   DWORD nNumberOfCharsToRead,
                   LPDWORD lpNumberOfCharsRead,
                   LPVOID pInputControl);
  ```
- lpReserved must be NULL
- Length parameters are in terms of characters, rather than bytes

```
BOOL WriteConsole(HANDLE hConsoleOutput,
                  const VOID* lpBuffer,
                  DWORD nNumberOfCharsToWrite,
                  LPDWORD lpNumberOfCharsWritten,
                  LPVOID lpReserved );
```
```

Console Configuration

BOOL SetConsoleMode(HANDLE hConsole, DWORD fdevMode)

- hConsole
  - Handle to Console opened with creatfile()
  - Must have GENERIC_WRITE (even for input-only device)
- fdevMode
  - Specifies how characters are processed some flags are
  - ENABLE_LINE_INPUT - read a line at a time
    - ENABLE_INSERT_MODE - insert text, don't overwrite
  - ENABLE_ECHO_INPUT - display chars as read
  - ENABLE_PROCESSED_INPUT - process ctrl characters
  - ENABLE_PROCESSED_OUTPUT
  - ENABLE_WRAP_AT_EOL_OUTPUT
- Also some applications don't use a console eg server/GUI

BOOL FreeConsole(VOID) - release console if not used

BOOL AllocConsole(VOID) - create a new console

Windows System Calls

- Lets you invoke any operating system command
- Must include <process.h> or <stdlib.h>

```
int system( const char *command );
```

- The command assumed to be an Windows command.
  - If command is not NULL, system returns the value that is returned by the command interpreter. A return value of – 1 indicates an error, and errno is set to one of the following :
    - E2BIG - Argument list is too long.
    - ENOENT - Command interpreter cannot be found.
    - ENOEXEC - Command-interpreter file is not executable.
    - ENOMEM – Insufficient memory or other memory error
  - If command is NULL, and the command interpreter is found, returns nonzero, otherwise returns 0 & sets errno = ENOENT
  - You must explicitly flush (using fflush or _flushall) or close any stream before calling system.
- Example; system(“dir \path”);


Portable Code

Writing Portable Code

- The Standard C library (CLIB) support most file I/O without significant performance impact.
- CLIB does not provide directory management or file locking etc
- Another possible solution is to use the normal POSIX functions, such as open() and read()....
  - A simple header file, and conditional compilation allows you to use the POSIX function names.

Equivalent Directory Functions

	C Library	Posix	Win32
Console IO	getc, scanf, gets	read	ReadConsole
	putc, printf, puts	write	WriteConsole
		ioctl	SetConsoleMode
Directory		mkdir	CreateDirectory
		closedir	FindClose
		Opendir	FindFirstFile
		readdir	FindNextFile
		getcwd	GetCurrentDir
	remove	rmdir unlink	RemoveDirectory
		readdir	SearchPath
		chdir	SetCurrentDirectory

Equivalent System Functions

	C Library	Posix	Win32
Errors	perror	strerror	FormatMessage
	errno	errno	GetLastError
SystemInfo		uname	GetVersion
		getrusage	GetSystemInfo
File Locks		fcntl(F_GETLK)	LockFile, etc
		fcntl(F_GETLK)	UnlockFile, etc

Equivalent File Sys Functions

C Library	Posix	Win32
fclose	close	CloseHandle
fopen	open, creat	CreateFile
fread	read	ReadFile
fwrite	write	WriteFile
remove	unlink	DeleteFile
fflush	fsync	FlushFileBuffers
rename		MoveFile, etc
tmpnam, tmpfile		GetTempFileName
	link	CreateHardLink
	stat, fstat	GetFileAttributes
ftell, fseek	stat, fstat	GetFileSize
	chsize	SetEndofFile
	fcntl	SetFileAttributes
fseek, rewind	lseek	SetFilePointer

Equivalent Time Functions

C Library	Posix	Win32
time, localtime		GetLocalTime
mktime		SystemTimetoFileTime
Time, gmtime		GetSystemTime
difftime		CompareFileTime
gmtime		FileTimeToSystemTime
	utime	SetFileTime

Example Sequential File Copy

- Basic cp file copy program
- Usage: cp file1 file2
- Using C stdio
- Using POSIX (on unix)
- Using Win32 POSIX
- Using Win32
- Using Win32 (faster version)
- Using Win32 builtin function

C library Implementation

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 256

int main (int argc, char *argv [])
{
          size_t bytes_in, bytes_out;
          char rec [BUF_SIZE];
          if (argc != 3) return 1;
          FILE * in_file = fopen (argv [1], "rb");
          FILE * out_file = fopen (argv [2], "wb");
          CheckFile (in_file, argv [1]);
          CheckFile (out_file, argv [2]);
          while ((bytes_in = fread (rec, 1, BUF_SIZE, in_file)) > 0)
          {
              bytes_out = fwrite (rec, 1, bytes_in, out_file);
              if (bytes_out != bytes_in)
               {
                  perror ("Fatal write error.");
                  return 4;
              }
          }
          fclose (in_file);
          fclose (out_file);
          return 0;
}
```

UNIX Implementation

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define BUF_SIZE 256

int main (int argc, char *argv [])
{
        ssize_t bytes_in, bytes_out;
        char rec [BUF_SIZE];
        if (argc != 3) return 1;
        int input_fd = open (argv [1], O_RDONLY);
        int output_fd = open (argv [2], O_WRONLY | O_CREAT, 0666);
        CheckFD (input_fd , argv [1]);
        CheckFD (output_fd , argv [2]);

        while ((bytes_in = read (input_fd, &rec, BUF_SIZE)) > 0)
        {
             bytes_out = write (output_fd, &rec, (size_t)bytes_in);
             if (bytes_out != bytes_in)
              {
                  perror ("Fatal write error.");
                  return 4;
             }
        }
        close (input_fd);
        close (output_fd);
        return 0;
}
```

Win32 POSIX Implementation

```
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define BUF_SIZE 8192

int main (int argc, char *argv [])
{
        int bytes_in, bytes_out;
        char rec [BUF_SIZE];
        if (argc != 3) return 1;
        int input_fd = _open (argv [1], O_RDONLY);
        int output_fd = _open (argv [2], O_WRONLY | O_CREAT, 0666);
        CheckFD (input_fd , argv [1]);
        CheckFD (output_fd , argv [2]);
        while ((bytes_in = _read (input_fd, &rec, BUF_SIZE)) > 0)
        {
            bytes_out = _write (output_fd, &rec, (unsigned int)bytes_in);
            if (bytes_out != bytes_in)
             {
                 perror ("Fatal write error.");
                 return 4;
            }
        }
        _close (input_fd);
        _close (output_fd);
        return 0;
}
```

Win32 Implementation

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 256

int main (int argc, LPTSTR argv [])
{
          DWORD nIn, nOut;
          CHAR Buffer [BUF_SIZE];
          if (argc != 3) return 1;
          HANDLE hIn = CreateFile (argv [1], GENERIC_READ, FILE_SHARE_READ, NULL,
                    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

          HANDLE hOut = CreateFile (argv [2], GENERIC_WRITE, 0, NULL,
                    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
             CheckHandle(hIn, argv [1]);
             CheckHandle(hOut, argv [2]);
          while (ReadFile (hIn, Buffer, BUF_SIZE, &nIn, NULL) && nIn > 0)
          {
               WriteFile (hOut, Buffer, nIn, &nOut, NULL);
               if (nIn != nOut)
                {
                    printf ("Fatal write error: %x\n", GetLastError ());
                    return 4;
               }
          }
          CloseHandle (hIn);
          CloseHandle (hOut);
          return 0;
}
```

Fast Win32 Implementation

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 8192

int main (int argc, LPTSTR argv [])
{
        DWORD nIn, nOut;
        CHAR Buffer [BUF_SIZE];
        if (argc != 3) return 1;
        HANDLE hIn = CreateFile (argv [1], GENERIC_READ, 0, NULL, OPEN_EXISTING,
                     FILE_ATTRIBUTE_NORMAL | FILE_FLAG_SEQUENTIAL_SCAN, NULL);
        HANDLE hOut = CreateFile (argv [2], GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
                     FILE_ATTRIBUTE_NORMAL | FILE_FLAG_SEQUENTIAL_SCAN, NULL);
         CheckHandle(hIn, argv [1]);
         CheckHandle(hOut, argv [2]);
        //--  Set the output file size --.
        DWORD FsLow = GetFileSize (hIn, NULL);
        SetFilePointer (hOut, FsLow, NULL, FILE_BEGIN);
        SetEndOfFile (hOut);

        while (ReadFile (hIn, Buffer, BUF_SIZE, &nIn, NULL) && nIn > 0)
        {
            WriteFile (hOut, Buffer, nIn, &nOut, NULL);
            if (nIn != nOut)
            {
                printf ("Fatal write error: %x\n", GetLastError ()); return 4;
            }
        }
        CloseHandle (hIn);
        CloseHandle (hOut);
        return 0;
}
```

Win32 CopyFile Implementation

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 256

int main (int argc, LPTSTR argv [])
{
      if (argc != 3) return 1;
     if (!CopyFile (argv [1], argv [2], FALSE))
     {
        printf ("CopyFile Error: %x\n", GetLastError ());
        return 2;
     }
     return 0;
}
```

Common functions

```
void CheckFile(FILE* pf, char* fname)
{
        if (pf != NULL)  return;
        printf("File Error %s", fname);
        exit(-1);
}
void CheckFD(int fd, char* fname)
{
        if (fd >= 0)  return;
        printf("File Error %s", fname]);
        exit(-1);
}
void CheckHandle(int hf, char* fname)
{
        if (hf != INVALID_HANDLE_VALUE)  return;
        printf("File Error %s", fname);
        exit(-1);
}
```