CSC 374/407: Computer Systems II

Lecture 7

Joseph Phillips
De Paul University

2017 September 5

Copyright © 2012-2017 Joseph Phillips
All rights reserved

Reading

- Bryant & O'Hallaron "*Computer Systems, 3rd Ed.*"
  - Chapter 10: System Level I/O
- Hoover "*System Programming*"
  - Chapter 5: Input/Output

Topics

High-level C file Input-Output

Iterating over directories

Getting file details

High level C Input-Output

Next lecture will discuss reading and writing ***a buffer of bytes*** efficiently

For now we'll concentrate on the ***high-level approach*** is good for dealing with ***lines, ints***, ***floats***, ***words***, *etc*.

Uses *`FILE*`* `stream` (or *`filePtr`*) instead *`int fileDescriptor`*.

Existing *`FILE*`* files:

- *`stdin`* ("standard input")
- *`stdout`* ("standard output")
- *`stderr`* ("standard error")

fopen()

FILE fopen(const char* pathname, const char* typePtr)*

- Opens file *pathname* according to *typePtr*:
- Returns ptr on success or *NULL* otherwise.

typePtr: can be

- "*r*": reading from beginning
- "*r+*": reading and writing from beginning
- "*w*": writing from beginning (truncated if exists, else create)
- "*w+*": reading and writing from beginning (truncated if exists, created otherwise)
- "*a*": writing from end (create if not exists)
- "*a+*": reading and writing from end (create if not exists)

fgets(), fgetc()

char fgets(char* bufferPtr, int bufferLen, FILE* filePtr)*

- Reads up to *bufferLen-1* characters from *filePtr* into *bufferPtr*. Reads '\n' into buffer too.
- Returns *bufferPtr* on success, else *NULL*.

int fgetc(FILE filePtr)*

- Reads up to *1* character from *filePtr*.
- Returns that char success, else *EOF*.

fprintf()

int fprintf(FILE filePtr, const char* format, . . .)*

- Prints to substituted *format* to *filePtr*.
- Substitutions include:
  - %d: Substitute in integer as decimal number
  - %x, %X: Substitute in integer as hexadecimal number
  - %c: Substitute in character
  - %s: Substitute in string
  - %g, %f: Substitute in floats and doubles
  - %p: Substitute in pointer value
- Returns returns number chars printed.
- *printf()* is the same as *fprintf(stdout,..)*

fflush(), fclose()

int fflush(FILE filePtr)*

- Flushes *filePtr* to disk, screen *etc*.
- *fflush(stdout)*:
  - Works fine in Linux,
  - May be problematic in Microsoft C.
- Returns *0* on success, otherwise *errno* is set.

fclose(FILE filePtr)*

- Closes *filePtr*.
- Returns *0* on success, otherwise *errno* is set.

feof(FILE filePtr)*

- Returns non-*0* if the end of *filePtr* has been reached
- Returns *0* otherwise.

Your turn!

Write a program that takes two parameters:

`$lineCounter string filename`

that counts and returns the number of lines of *`filename`* that begin with string *`string`*.

- If *`filename`* cannot be opened it writes an error message to *`stderr`*.

Well, there is `fscanf()`, but . . .

Just so you've seen it:

`int fscanf(FILE* filePtr, const char* format, . . )`

- Returns number of items read

Better to use *`fgets()`*, then

`int sscanf(const char* source, const char* format, . . .)`

- What goes in format? Largely the same codes as for *`fprintf()`* (next slide).

`int strtol(const char*,char**,int)`

- Returns integer: *`strtol("123",0,10) == 123`*

`double strtod(const char*,char**)`

- Returns double: *`strtod("12.3",0)==12.3`*

Like `FILE*` but want buffered objects instead of lines?

`size_t fread(void* ptr, size_t size, size_t numItems, FILE* filePtr)`

- Reads *`numItems`* of size *`size`* from *`filePtr`* and puts them in *`ptr`*.
- Returns number *items* read.

`size_t fwrite(const void* ptr, size_t size, size_t numItems, FILE* filePtr)`

- Writes *`numItems`* of size *`size`* from *`ptr`* to *`filePtr`*.
- Returns number *items* written.

Your turn!

Write a program that reads from 0 to N int pairs:

- Ignore blank lines or lines with just spaces
- Ignore comment lines whose first non-space char is #
- Ignore any spaces up to the two ints, and between them
- Uncommented letters, *etc*. are errors.

```
#  Ignore this comment line
12 34    # Good
  56 78 # Okay
1 # Bad
Also bad
```

Eeww! Parsing!

- What's the best programming structure to read an unbounded number of lines?
- Useful stuff:
  - *`int isdigit(char c), int isspace(char c)`*

stdout vs. stderr

Q: Why might it be useful to distinguish between output messages and error messages?

A: For debugging!

```
#include <stdlib.h>
#include <stdio.h>
/* $ ./stdoutVsStderr
 * I'm an ordinary msg.
 * I'm the error msg.
 * $ ./stdoutVsStderr 2> error.txt
 * I'm an ordinary msg.
 * $ cat error.txt
 * I'm the error msg.
 */
int   main  ()
{
  fprintf(stdout,"I'm an ordinary msg.\n");
  fprintf(stderr,"I'm the error msg.\n");
  return(EXIT_SUCCESS);
  }
```

Is using `FILE*` as efficient as `int fd`?

Probably not (**`FILE*`** uses *`int fd`*), but it is buffered.

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("T");
    printf("h");
    printf("i");
    printf("s");
    printf(" ");
    printf("i");
    printf("s");
    printf("n");
    printf("'");
    printf("t");
    printf(" ");
    printf("e");
    printf("f");
    printf("f");
    printf("i");
    printf("c");
    printf("i");
    printf("e");
    printf("n");
    printf("t");
    printf("\n");
    fflush(stdout);
    return(EXIT_SUCCESS);
}
```

Is using `FILE*` as efficient as `int fd`?

```
$ strace ./printf_sys_call_ex
execve("./printf_sys_call_ex", ["./printf_sys_call_ex"],
  [/* 46 vars */]) = 0
brk(0)                                  = 0x8fa7000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=63949, …})
                                        = 0
mmap2(NULL, 63949, PROT_READ, MAP_PRIVATE, 3, 0)
                                        = 0xb7fb3000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3,
  "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\360\364
  @\0004\0\0\0"..., 512)                = 512
. . .
write(1, "This isn\'t efficient\n", 21This isn't efficient
)                                       = 21
```

Manipulating files and filesys

There are several other system calls for the Unix file system including:

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

```
unlink(const char* filename);
```

– Removes (erases) files.

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

– Changes file permissions

```
chdir(const char *path);
```

– Changes the working directory

Iterating over files in directory

Like *fopen(), fgets(), fclose()* but for directories

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

DIR*           opendir  (const char* name);
struct dirent* readdir  (DIR *dir);
int            closedir (DIR*);

struct dirent
{
  ino_t   d_ino;        // inode number
  off_t   d_off;        // offset to next dirent
  ushort  d_reclen;     // length of record
  uchar   d_type;       // type of file
  char    d_name[256]; // filename
};
```

Your turn!

Write a program `lister` that takes an optional command line argument

- *`./lister dirName`*
  - Lists directory *`dirName`* (assume it exists)
- *`./lister badDirName`*
  - Prints an error message to **`stderr`** if badDirName is not a directory or if don't have permission to read it.
- *`./lister`*
  - Lists the items in the current directory (".")

Finding details about a file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char* path, struct stat* buf);
struct stat
{ dev_t     st_dev;   // Device ID
  ino_t     st_ino;   // inode
  mode_t    st_mode;  // what type of "file"
  nlink_t   st_nlink; // num hard links
  uid_t     st_uid;   // user ID of owner
  gid_t     st_gid;   // group id of owner
  dev_t     st_rdev;  // Device ID (special files)
  off_t     st_size;  // Total size in bytes
  blksize_t st_blksize;//Filesys' block size
  blkcnt_t st_blocks;// Num allocated blocks
  time_t    st_atime,st_mtime,st_ctime;
  // Access (read or write), modify (change
  metadata), change (write) times
};
```

`stat`, cont'd

What type of file is that?

Use these macros on *`st_mode`*:

- *`S_ISREG(m)`*: Regular file
- *`S_ISDIR(m)`*: Directory
- There are others (block & char devices, symbolic links, FIFOs and sockets)

Your turn!

Revise your *`lister`* program into *`lister2`* that for files will print:

- the size in bytes for files
- "*`(dir)`*" for directories
- "*`(other)`*" of entries other than a file or directory

`stat`, cont'd

"*Hey buddy, got the time?*" Recall:

```
struct stat
{
  . . .
  time_t   st_atime;  // Last Access (read or write)
  time_t   st_mtime;  // Last Modify (metadata)
  time_t   st_ctime;  // Last Change (write)
};
```

Printing the time:

```
#include <time.h>
char* ctime(time_t* );
```

- Returns c-string telling time in human-readable form

Your turn!

Revise *`lister2`* to print the last change (write) time for all entries

How would you modify your program to recursively descend into directories (other than “.” and “..”)

I don’t have time, but also check out:

`fdopen(int fd, const char *mode)`: Associates a stream with the existing file descriptor, `fd`. The mode of the stream (one of the values `"r"`, `"r+"`, `"w"`, `"w+"`, `"a"`, `"a+"`) must be compatible with the mode of the file descriptor.

`freopen()`: function opens the file whose name is the string pointed to by path and associates the stream pointed to by stream with it.

`fseek()`: function sets the file position indicator for the stream pointed to by stream.

Next time: Low-level I/O and Sockets