CSC 4304 - Systems Programming
Fall 2010

LECTURE - IV

# FILE I/O

Tevfik Koşar

Louisiana State University

September 2nd, 2010

## Summary of Last Class

- Advanced Structures in C
  - Memory Manipulation in C
  - Pointers & Pointer Arithmetic
  - Parameter Passing
  - Structures
  - Local vs Global Variables
  - Dynamic Memory Management



## In Today's Class

- File I/O
  - buffered vs unbuffered I/O
  - opening and closing files
  - reading from / writing to files
  - seeking files
  - formatted I/O vs unformatted I/O



## Buffered vs Unbuffered I/O

- Unbuffered I/O: each read write invokes a system call in the kernel.
  - read, write, open, close, lseek
- Buffered I/O: data is read/written in optimal-sized chunks from/to disk --> streams
  - standard I/O library written by Dennis Ritchie



## Unbuffered I/O



## Open a File

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- Parameters:
  - pathname: the name of the file
  - flags:
    - O_RDONLY: read-only access
    - O_WRONLY: write-only access
    - O_RDWR: read-write access
    - O_CREAT: if the file does not exist, then create it
  - mode: when you create a file, this specifies the access rights to it
    - 0600: read-write access for you, nothing for the others
    - 0644: read-write access for you, read-only access for the others

## Open a File

- `open()` returns an integer:
  - $-1$ means "error:" the file could not be open
  - $\geq 0$: this is the "file descriptor" of the open file. Save it in a variable, you need to pass it to all the other file-related functions.
- You don't need to specify the "mode" unless you will be creating a new file

```
fd = open("this_file_already_exists",O_RDONLY);
```

- You can combine multiple flags together:

```
fd = open("foo",O_RDWR|O_CREAT,0644);
```



## Read From a File

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

- Parameters:
  - `fd`: the file descriptor of the file you want to read from
  - `buf`: the buffer where the file's content should be stored
  - `count`: how many bytes to read
- `read()` returns an integer:
  - $-1$ means "error:" the reading has failed
  - $\geq 0$: tells you how many bytes were actually read (if the value is lower than what you requested, then you know you have reached the end of the file)



## Write into a File

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

- Parameters:
  - `fd`: the file descriptor of the file you want to write to
  - `buf`: the buffer containing the data to be written
  - `count`: how many bytes to write
- `read()` returns an integer:
  - $-1$ means "error:" the writing has failed
  - $\geq 0$: tells you how many bytes were actually written (usually the amount you have requested, otherwise you may have found a very strange exception)



## Close a File

```
#include <unistd.h>

int close(int fd);
```

- Parameters:
  - `fd`: the file descriptor to close
- `close()` returns an integer:
  - $-1$ means "error:" the closing has failed
  - 0: OK
- Do not forget to close file descriptors when you have finished using a file!



## Special Files

- When your program starts, 3 file descriptors are created automatically for you
  - You can use them as you like
  - You are not obliged to close them after you finished using them
- These file descriptors are:
  - 0: this is the standard input for your program. If the user types something while the program is running, it can be read from file descriptor 0. You cannot write in descriptor 0.
  - 1: this is the standard output for your program. Writing to file descriptor 1 will display the written message on screen.
  - 2: this is the standard error output for your program. Writing to file descriptor 2 will display the written message on screen. Use this one to output error messages.



## Seeking a File

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

off_t lseek(int fd, off_t offset, int whence);
```

Returns: new file offset if OK, -1 on error

whence: where to start the offset i.e. SEEK_SET, SEEK_CUR, SEEK_END

## Example

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

int main(){
        char buf1[] = "abcdefghij";
        char buf2[] = "klmnopqrst";
        int fd;
        if ( (fd = open("file.txt", O_RDWR | O_CREAT, 0755)) < 0)
                { printf("open error!"); exit(-1);}

        if ( write(fd, buf1, 10) != 10)
                { printf("buf1 write error!"); exit(-1);}


        if ( lseek(fd, 40, SEEK_SET) == -1)
                { printf("buf1 seek error!"); exit(-1);}

        if ( write(fd, buf2, 10) != 10)
                { printf("buf2 write error!"); exit(-1);}
        close(fd); exit (0);
}
```



## Buffered I/O



## Standard I/O Library

- Difference from File I/O
  - File Pointers vs File Descriptors
  - fopen vs open
    - When a file is opened/created, a *stream* is associated with the file.
    - FILE object
      - File descriptor, buffer size, # of remaining chars, an error flag, and the like.
  - stdin, sdtout, stderr defined in <stdio.h>
    - STDIO_FILENO, STDOUT_FILENO,…



## Buffering

- Goal
  - Use the minimum number of read and write calls.
- Types
  - Fully Buffered
    - Actual I/O occurs when the buffer is filled up.
    - A buffer is automatically allocated when the first-time I/O is performed on a stream.
    - flush: standard I/O lib vs terminal driver



## Buffering

- Line Buffered
  - Perform I/O when a newline char is encountered! – usually for terminals.
  - Caveats
    - The filling of a fixed buffer could trigger I/O.
    - The flushing of all line-buffered outputs if input is requested.
- Unbuffered
  - Expect to output asap, e.g. using write()
  - E.g., stderr



## Buffering

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode,
   size_t size);
```

- Full/line buffering if buf is not NULL (BUFSIZ)
  - Terminals
- mode: _IOFBF, IOLBF, _IONBF (<stdio.h>)
  - Optional size → st_blksize (stat())
- #define BUFSIZ 1024 (<stdio.h>)
- They must be called before any op is performed on the streams!

## Buffering

- ANSI C Requirements
  - Fully buffered for stdin and stdout unless interactive devices are referred to.
    - SVR4/4.3+BSD – line buffered
  - Standard error is never fully buffered.

#include <stdio.h>
int fflush(FILE *fp);
  - All output streams are flushed if fp == NULL



## Opening a Stream

- #include <stdio.h>
- FILE *fopen(const char *pathname, const char *type);

- opens a specified file
- types:
  - r : open for reading
  - w : create for writing or truncate to 0
  - a : open or create for writing at the end of file
  - r+ : open for reading and writing
  - w+: create for reading and writing or truncate to 0
  - a+ :open or create for reading and writing at the end of file

  - use b to differentiate text vs binary , e.g. rb, wb ..etc



## Restrictions

| Type | r | w | a | r+ | w+ | a+ |
|---|---|---|---|---|---|---|
| File exists? | Y | | | Y | | |
| Truncate | | Y | | | Y | |
| R | Y | | | Y | Y | Y |
| W | | Y | Y | Y | Y | Y |
| W only at end | | | Y | | | Y |

* When a file is opened for reading and writing:

- Output cannot be directly followed by input without an intervening *fseek, fsetpos, or rewind*
- Input cannot be directly followed by output without an intervening *fseek, fsetpos, or rewind*



## Closing a Stream

#include <stdio.h>
int fclose(FILE *fp);
  - Flush buffered output
  - Discard buffered input
  - All I/O streams are closed after the process exits.

- setbuf or setvbuf to change the buffering of a file before any operation on the stream.



## Reading and Writing from/to Streams

- Unformatted I/O
  - Character-at-a-time I/O, e.g., getc
    - Buffering handled by standard I/O lib
  - Line-at-a-time I/O, e.g., fgets
    - Buffer limit might need to be specified.
  - Direct I/O, e.g., fread
    - Read/write a number of objects of a specified size.
    - An ANSI C term, e.g., = object-at-a-time I/O



## Reading a Char

#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
  - getchar == getc(stdin)
  - Differences between getc and fgetc
    - getc could be a macro
    - Argument's side effect, exec time, passing of the function address.
  - unsigned char converted to int in returning

## Error/EOF Check

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
void clearerr(FILE *fp);
int ungetc(int c, FILE *fp);
```

- An error flag and an EOF flag for each FILE
- No pushing back of EOF (i.e., -1)
  - No need to be the same char read!



## Writing a Char

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

- putchar(c) == putc(c, stdout)
- Differences between putc and fputc
  - putc() can be a macro.



## Example 1

```
#include <stdio.h>
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```



## Line-at-a-Time I/O

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
```

- Include '\n' and be terminated by *null*
- Could return a partial line if the line is too long.

```
char *gets(char *buf);
```

- Read from stdin.
- No buffer size is specified → overflow
- *buf does not include '\n' and is terminated by *null.*



## Line-at-a-Time I/O

```
#include <stdio.h>
char *fputs(const char *str, FILE *fp);
```

- Include '\n' and be terminated by *null.*
- No need for line-at-a-time output.

```
char *puts(const char *str);
```

- *str does not include '\n' and is terminated by *null.*
- puts then writes '\n' to stdout.



## Example 2

```
#include <stdio.h>
main()
{
    int bufsize = 1024;
    char buf[bufsize];

    while (fgets(buf, bufsize, stdin) != NULL)
        fputs(buf, stdout);
}
```

## Standard I/O Eficiency

- Copy stdin to stdout using:

| | total time | kernel time |
|---|---|---|
| • fgets, fputs : | 2.6 sec | 0.3 sec |
| • fgetc, fputc : | 5 sec | 0.3 sec |
| • read, write : | 423 sec | 397 sec (1 char at a time) |



## Summary

- File I/O
  - buffered vs unbuffered I/O
  - opening and closing files
  - reading from / writing to files
  - seeking files
  - formatted I/O vs unformatted I/O


- Next Week: Continue File I/O
- Read Ch.3 from Stevens
- HW-1 due Sep 9th
- Project-1 out next week



## Acknowledgments


- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), and B. Knicki (WPI).