Buffered I/O

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss buffering and user-buffering concepts including block size & kernel
- Explain the standard library components
- Discuss several buffered-I/O operations such as opening, closing, and seeking
- Explain several means to read/write files including single chars, lines, & binary
- Explain buffering control and types

Buffering?

- All I/O is done in ***blocks***
  - So even if you write a single byte, the OS will actually write an entire block
    - This can be very inefficient as the OS has to fix up your I/O ensuring everything is block aligned
    - The entire situation is made worse by situation in which we read the same bytes over and over
  - Solution: **User-buffered I/O**
    - Applications read/write naturally, but the I/O occurs in units of the file system block size

User-Buffered I/O

- User-Buffered implies that the buffering happens in user space rather than the OS
  - buffering aimed to improve performance
- If we write in blocks, we can incur enormous gains from fewer writes
  - The bigger the block, the more you write, so the less often you write
  - However, our performance degrades if we are not using block boundaries
    - we're still writing partial blocks

Block Size

- In practice, blocks are usually powers of 2
  - 512, 1024, 2048, 4096, or 8192
- We can determine a precise block size through **stat**
  - In practice, we don't really care
- Your primary goal is not to pick an oddball size (like 1130) and pick something that is a integer multiple of the actual block size
  - Since we're talking powers of 2s, this is easy…
    - Pick 4096 or 8192
      - All the smaller sizes are multiples of these
- Ok, so now we have a size… done… You wish!
  - Programs work in bytes (integers, characters, etc.), lines, etc.
  - Our user-buffer handles the difference…
    - all reads/writes go through a buffer!

Standard Library

- The standard C library provides the standard I/O library (stdio)
  - Provides platform-independent, user-buffering
- **File pointers**
  - Stdio routines do NOT operate using file descriptors
  - Instead, the use a file pointer
    - Maps to a file descriptor
    - Represented by a pointer to the **FILE typedef**
  - Now that we're using pointes, we'll be referring to **streams**
    - Streams are a portable way of reading and writing data

Opening a File

- **`FILE *fopen(char *name, char *mode)`**
  - Returns a pointer to a FILE
    - Or NULL if the open failed
  - Legal values for mode include:
    - "r" – open a text file for read only
    - "w" – create text file for writing (discard previous contents)
    - "a" – open or create a text file for append operations
    - "r+" – open text file for updating (reading and writing)
    - "w+" – create text file for updating (discard old file)
    - "a+" – open a create a text file for appending operations
  - Appending a "b" to the end of any of these modes indicates the file that is being opened is a binary file

Opening a file with fds

- **`FILE *fdopen(int fd, const char *mode)`**
  - Returns a pointer to a FILE
    - Or NULL if the open failed
  - The modes are the same for fopen
- NOTE:
  - Once a file descriptor is opened as a stream, it should not be used for direct I/O

Closing a Stream

- **`int fclose(FILE *stream)`**
  - Any buffered data is flushed
  - On success, it returns 0
    - Otherwise returns EOF and sets errno appropriately
- We can close all streams!
  - **`int fcloseall(void)`**
    - All streams flushed and closed
    - Always returns 0

Reading from a Stream

- The standard C library provides many ways to read from a stream, but we'll focus on three simple ideas:
  - Read one character at a time
  - Read an entire line at a time
  - Reading binary data

Reading A Single Character

- **`int fgetc(FILE *stream)`**
  - Returns it as a unsigned char cast as an int
    - It needs the range for error reporting

```
int c = fgetc(stream);
if (c == EOF)
   /* error */
else
   printf("c=%c\n",(char)c)
```


Reading an Entire Line

- **`char * fgets(char *s, int n, FILE *stream)`**
  - Reads at most n-1 characters in the array s which is terminated by "\0" stopping if a newline is encountered
    - That newline is included in the array
  - Returns s or NULL if EOF or other error occurs

Reading Arbitrary Strings

- Sometimes you wish to read to a delimiter other than a new line
  - We have to get back to **fgetc**

```
char *s; int c; s = str;
while (--n >0 && (c = fgetc(stream)) != EOF)
   *s++ = c;
*s = '\0'

char *s;int c = 0; s = str;
while (--n >0 && (c = fgetc(stream)) != EOF && (*s++ = c)
!= d);
if (c == d)
   *--s = '\0'
else {
 *s = '\0'
```


Reading Binary Data

- **int fread(void *buf, size_t size, size_t num, FILE *stream)**
  - The number of elements read is returned, not the number of bytes
  - must be opened with "b" suffix

Buffered Binary I/O Example

```
char *name = "testfile";
   FILE *fp;

   fp = fopen(name, "rb");

   int x;

   int count = 0;
   int read = fread(&x,sizeof(int),1,fp);

   while (read != 0) {
       count++;
       printf("%s\t%d\n", x, read);
       read = fread(&x,sizeof(int),1,fp);
   }
   printf("COUNT: %d\n",count);
   fclose(fp);
```

Writing to a Stream

- Again, there are three popular approaches to writing to streams:
  - Writing a single character
  - Writing a string of characters
  - Writing binary data

Writing a Single Character

- **`int fputc(int c, FILE *stream)`**
  - Writes the unsigned character c to the stream
  - On success, it returns c…
    - Otherwise returns EOF

```
if (fputc('p', stream) == EOF)
   /*error */
```


Writing a String of Characters

- **`int fputs(const char *s, FILE *stream)`**
  - Writes the entire string s on the stream
  - Returns non-zero if successful, or EOF for error

```
if (fputc("Michael is awesome\n", stream) == EOF)
   /*error */
```

Writing Binary Data

- **`int fwrite(void *buf, size_t size, size_t num, FILE *stream)`**
  - Writes num elements each size bytes in length from the data pointed at buf
  - The number of elements written is returned
  - must be opened with “b” suffix

Seeking?

- Seeking allows us to manipulate the current file position
  - **`int fseek (FILE *stream, long offset, int whence)`**
    - Whence controls the function:
      - SEEK_SET
        - » position = offset
      - SEEK_CUR
        - » position = current position + offset
      - SEEK_END
        - » Position = EOF + offset
    - If successful, returns 0, otherwise returns -1

If you like convenience

- **`int fsetpos(FILE *stream, fpos_t *pos)`**
  - Basically fseek with whence set to SEEK_SET
- **`void rewind(FILE *stream)`**
  - Basically fseek(stream, 0, SEEK_SET)

Get the File Position

- `long ftell (FILE *stream)`
  - On success, returns current file position
    - Otherwise, returns -1 and sets errno
- We also have:
  - `int fgetpos(FILE *stream, fpos_t *pos)`
    - On success, returns 0 and sets the pos to current position

Remember to Flush

- `int fflush(FILE *stream)`
  - If stream is NULL, all streams are flushed
  - On success, returns 0, otherwise returns EOF
  - Note:
    - This only flushes user buffers to kernel
    - Not necessarily going to disk, you need to flush the kernel buffers using fsync() to do that

Errors & EOF

- Some (most) of the standard I/O interfaces discussed do a poor job of communicating failures
  - Is it EOF or a literal failure?
  - Standard I/O provides two interfaces to determine whether error or EOF
    - `int feof(FILE *stream)`
      - If EOF has been encountered, it returns nonzero and a 0 otherwise
    - `int ferror(FILE *stream)`
      - If error indicator has been set, return nonzero, 0 otherwise

Obtaining the File Descriptor

- Mixing standard I/O calls with system calls is not normally advised!
  - Buffering is happening on user and kernel side… so you should flush before doing so
  - Generally speaking though, this should never happen
  - int fileno(FILE *stream)
    - Returns the fd on success, -1 otherwise

Controlling the Buffering

- User-buffered I/O is based on buffering in user-space, so we can control it
  - There are three types:
    - Unbuffered
    - Line-Buffered
    - Block-Buffered
  - Usually, the default buffering type is generally optimal
    - Can be set with setvbuf function
      - _IONBF
      - _IOLBF
      - _IOFBF

Summary

- Discussed buffering and user-buffering concepts including block size & kernel
- Explained the standard library components
- Discussed several buffered-I/O operations such as opening, closing, and seeking
- Explained several means to read/write files including single chars, lines, & binary
- Explained buffering control and types

Questions?