一. 文件的基本知识

（这一章原书讲得比较乱，我自己看看博客稍微整理了一下
其实也不是原书的问题，C对于文件处理比较弱，推荐使用C++的ifstream ofstream）

1. 文件的定义：文件通常是在**磁盘或固态硬盘**上的一段已命名的存储区。

2. 文件的类型：文件有两种类型：①**文本文件**——利用某种编码解释其中的二进制内容（编辑器通过对应的字符集解码可以查看其中的内容）。②**二进制文件**——包含原始的二进制内容，可能是图片，视频等，不能用编辑器查看。

3. C访问文件的两种途径（一般我们把文件都默认为是**文本文件**）：①**二进制模式**——可以访问文件的原样每一个字节（C语言的输入输出都是**字节流**）②**文本模式**——C语言要把本地环境映射为C模式（比如行末尾和文件末尾都改成\n），读写都要映射。

（注意这两种模式对UNIX和Linux的实现都完全一致）

4. 三个概念的区别：

底层I/O：使用OS提供基本的I/O服务

标准I/O：是ANSI C建立的一个标准I/O模型，是一个标准函数包和stdio.h头文件中的定义，具　　有一定的可移植性。

文件I/O：使用指向FILE指针的函数都是文件I/O（我认为与标准I/O并不冲突）

5.标准I/O的机理（理解缓冲区）

①调用fopen打开文件，创立**缓冲区**和**包含文件和缓冲区数据的结构（~~我个人认为这是把文件从~~磁盘上~~读取到子内存中~~）（有点小问题，这里的结构应该就是FILE类型，其中的成员包含了文件的属性。）**

②考虑文件输入，首先文件中的缓冲大小数据块就被拷贝到缓冲区中（缓冲区大小因系统而异，而且**所有输入函数都使用相同的缓冲区**），文件指针一个一个读取缓冲区中的字符，指导缓冲区空了为止，这时如果文件还没读完（C使用一种叫做文件结尾指示器的数据。要理解文件是一种**结构**，这个文件结尾指示器是这个结构的一个成员），继续放入缓冲区。

③读到文件结尾以后，结尾治时期设置为真，下一次被调用的输入函数将返回EOF

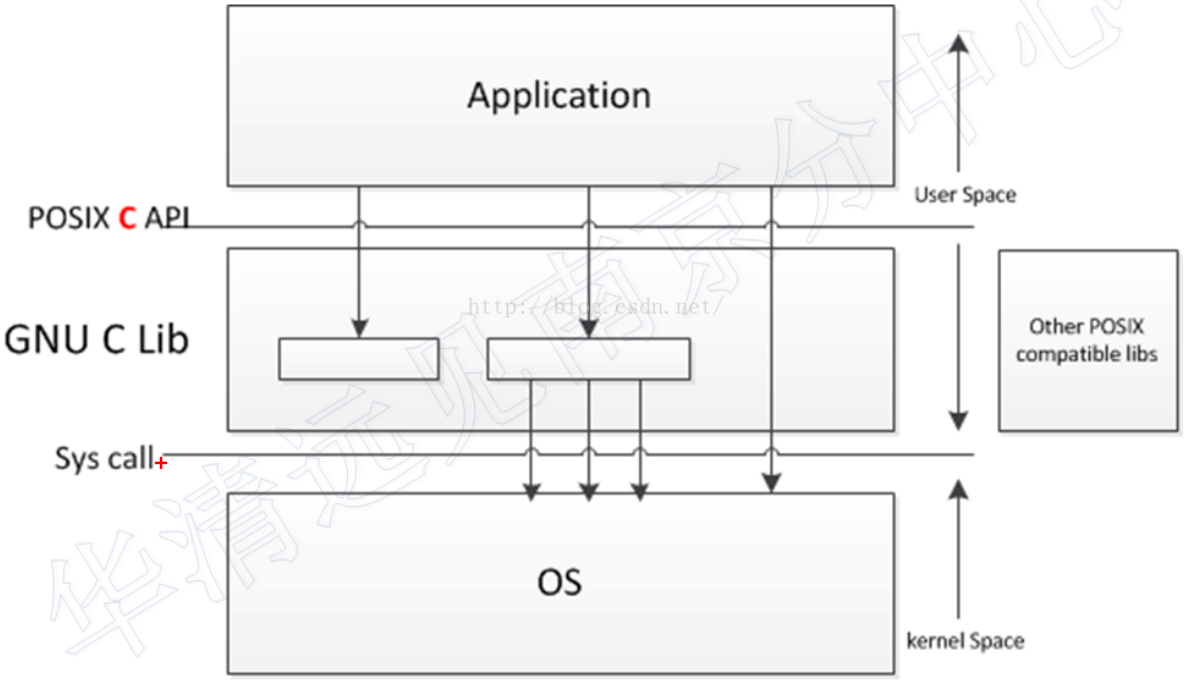
（其实从结果上看和不用缓冲区的底层IO没什么区别，那么我们**为什么要引入缓冲呢**?

答：通过文件I/O读写文件时，每次操作都会执行相关系统调用。这样处理的好处是直接读写实际文件，坏处是频繁的系统调用会增加系统开销，标准I/O可以看成是在文件I/O的基础上封装了缓冲机制。先读写缓冲区，必要时再访问实际文件，从而**减少了系统调用的次数。**）

6. C语言的三个标准文件：①stdin; ②stdout; ③stderr;

（引入stderr的目的是弥补了输出定向不到屏幕上的缺陷）

7.拓展



其实我们刚刚讨论的都可以使用这一种图来表示

（注：**POSIX——Portable Operating System Interface 可移植操作系统接口**）

①如图所示，OS提供了一种sys call，通过这组接口用户可以实现**操作系统内核**所提供的多种功能，如分配内存，创建进程，实现进程的相互通信等。**但我们不建议直接使用sys call。** ②为什么不建议呢？很大一部分原因在于不同系统之间接口不同，**不好移植。**再者其接口实现的功能过于简单。

③为了解决难以移植的问题，C语言使用了一种叫做C库的东西，它是对系统底层接口的封装，具有良好的移植能力。现在我们不需要直接调用系统底层的接口，我们可以直接使用**应用程序编程接口（API）**

二. 各种函数的学习

1. fopen

①prototype: FILE * fopen(char * filename, int mode)

②关于MODE：这个我不过多解释了，提示一下加“+”代表读写均可，加‘b’代表用二进制模式打开 ③文件打开失败讲返回NULL指针，处理：

while (!(fp = fopen(*FILENAME, MODE*)) { ... }

④注意事项：文件要与源程序放在同一个目录下。

⑤还有一点，在vs环境下我们使用fopen要报错，需要使用更加安全的fopen_s,在GNU公共域中的gcc也是不能使用，要使用fopen64（）；

2. fclose

①prototype: int fclose(FILE * fp)

②返回值：正常关闭返回0，异常关闭返回-1

③异常处理：if (fclose(*FILENAME*)) { ... }

3. getc

①prototype: int getc(FILE * fp)

②作用：从文件中读取一个字符，读到文件结尾时返回EOF（区别于fgets，遇到EOF或错误都返回NULL，毕竟人家返回的是指针）

4. putc

①prototype: int putc(int ch, FILE * fp)

②作用：讲一个字符输出到fp文件中。

5. fprintf

①prototype: int fprintf(FILE * fp, char * format, ...);

②作用：在fp中添加格式化后的字符串。

③举例：fprintf(stderr, "Can't open file %s", *filename*);

6. fscanf

①prototype: int fscanf(FILE * fp, char format, ...);

②作用：将文件中的数据填入后面参数表中的地址中。

7. fseek

①prototype: int fseek(FILE * fp, long offset, int base);

②offset:相对于base的偏移量，可正可负。必须是long类型。

③base：有三种 SEEK_SET(){文件开始处}, SEEK_CUR(){当前位置}, SEEK_END(){文件末尾}

8. ftell

①prototype: long ftell(FILE * fp);

②作用：返回当前的读写位置

9. rewind

①prototype: void rewind(FILE * fp);

②作用：讲位置指针置于文件的开头；

10. fflush

①prototype: int fflush(FILE * fp)

②作用：刷新缓冲区，讲缓冲区中所有的未写入的数据发送到输出文件。

11. setvbuf

①prototype: int setvbuf(FILE * restrict fp, char * restrict buf, int mode, size_t size);　　②各参数作用：fp——识别待处理的流，buf——指向待处理的缓冲区， mode——模式，size——缓冲区的大小

③模式详解：_IOFBF{完全缓冲，等缓冲区满了才刷新} _IOLBF{行缓冲，缓冲区满或写入一个换行符} _IONBF{无缓冲}

（记忆方法：io（输入输出）-f（full）-l（line）-n（no）buffer（缓冲））

④存在意义：默认缓冲区大小跟操作系统和文件系统有关，当自己要建立一个新的缓冲区（如嵌入式开发）时，可以使用

12. fwrite

①prototype: size_t fwrite(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict fp);　　②新出现的事物：*void——通用类型指针

③各种参数解释： ptr——各种数据的指针，size——一个数据类型要存储多少字节（毕竟之前只给了首地址）， nmemb——代写入数据块的数量， fp——文件指针

例：double a[10];

fwrite(a, sizeof(double), 10, fp);

④作用：将ptr中的数据写入文件中。

13. fread

①size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict fp);

②作用：将文件中的数据传入ptr上。

（fread和fwrite存在的意义：之前的IO函数都是面向文本的，用于处理字符和字符串，这导致我们的数字在文件中也是按照字符和字符串的格式来处理的，而我们可以使用fread和fwrite保存**最原始的二进制数据**，不过要注意这样保存的文件已经无法用文本编辑器打开了）

14. feof

①prototype: int feof(FILE * fp)

②作用：文件结束返回非零值，未结束返回0

③判断当前位置是否处于文件末尾:　if (!feof(fp)) {...}

15. ferror

①prototype: int ferror(FILE * fp)

②作用：文件遇到错误返回非0值，否则返回0

③判断文件是否遇到错误：if (ferror(fp)) {...}