課題要旨

低レベル API である `open` ， `close` ， `read` は，それぞれファイルを開く，閉じる，読み込むという機能を持つ．バッファリングは，ファイルの読み書きを効率化するために行われるが，低レベル API では行われない．本課題ではバッファリングをする高レベル API を作成する．

ソースコードの説明

ソースコードは別途提出しているほか，付録として p.3 に貼付している．

構造体の定義 バッファリングをするに際して，「ファイル」を定義するため，その構造体内にバッファを持つ必要がある．バッファは `buffer` として `char` 型の配列で定義し，配列の位置を指すポインタを `index` として `int` 型で定義する．また，バッファサイズをヘッダで `MyBufferSize` として定義する． `int` 型の `count` は， `read` を用いて `buffer` へ何オクテット書き込んだかを保持する．この構造体を `my_file` として定義する．

ファイルを開く `my_file` 型のポインタ関数として `my_fopen` を定義する．ファイル名のポインタを引数として受け取る． `open` を用いてファイルを開き，そのファイルディスクリプタを変数 `fd` に保持する．正常にファイルが開かれた場合， `my_file` 型のポインタ変数 `*fp` を宣言し，そのために必要なメモリを `malloc` を用いて動的に確保する．それぞれのメンバを初期化[1)]して， `fp` を返す．正常にファイルを開けなかった場合は， `NULL` を返す．

ファイルを閉じる ファイルを閉じる関数として `int` 型の `my_fclose` を定義する．引数として，`my_file` 型のポインタを受け取る． `close` を用いてファイルを閉じる． `close` の仕様により，正常に閉じられた場合は `0` を，閉じられなかった場合は `-1` を返す．このとき， `my_fopen` で動的に確保したメモリを `free` を用いて解放する．

1 文字単位の入力 指定されたファイルを 1 文字単位で入力する関数として `int` 型の `my_getc` を定義する．引数として， `my_file` 型のポインタを受け取る．バッファリングの機構として，`fp->index` が `fp->count` より大きくなった場合，つまりバッファを全て読み取った場合，または `fp->buffer` が空である場合は， `read` を用いて `MyBufferSize` オクテットを読み取り，`fp->buffer` に格納する．このとき， `fp->index` ，を初期化し， `fp->count` に `read` で読み取ったオクテット数を格納する．ここで， `read` で読み取ったオクテッド数が 0 以下である場合， `EOF` を返す． `size = 0` のとき，これ以上読み取ることができない状況であり， `size = -1` のとき，エラーが発生した状況である． `fp->buffer` に内容があり， `(*fp).index` が読み取ったサイズと等しくなるまで， `fp->buffer` 上の `fp->index` 番地を返し， `fp->index` をインクリメントする．`fp->buffer` がすべて読み取られた場合， `EOF` を返す．

---

1) 本課題では，学習のためソースコード上ではアロー演算子を利用していない． `(*fp).index = 0` と `fp->index = 0` は同義である．

考察

バッファリングの機構により， `read` を用いてファイルを読み取る回数が減り，ファイルの読み取りが効率化されるとされている．しかし，配列が空であるかどうかを判定するために，今回であれば `MyBufferSize` 回の判定を行う必要がある．このため， `MyBufferSize` が大きいほど，初回のファイル読み取りに時間がかかると考えられる．グローバル関数に `is_buffer_empty` を定義し，`fp->buffer` が空であるかどうかをファイルを開いて初めて `my_fgetc` を呼び出すときに判定することで， `MyBufferSize` 回の判定を行う必要がなくなる．

感想

すでにある `read` ， `open` などの仕様をしっかりと確認しつつ，コーディングすることが求められた．本来，プログラミングする際に，用いる関数の仕様をしっかりと確認することが重要であるが，今回の課題を通して，その重要性を再認識することができた．

付録：ソースコード

Listing 1 `assignment1.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MyBufferSize 5

bool isBlank = false;

struct my_file
{
    int fd;                    // file descriptor
    int count;                 // number of characters read from buffer
    int index;                 // index of next character to be read from buffer
    char buffer[MyBufferSize]; // buffer
};

struct my_file *my_fopen(char *filename)
{
    int fd;
    fd = open(filename, O_RDONLY); // Request OS to open file
    isBlank = false;
    if (fd != EOF)
    {
        struct my_file *fp;                                    // file structure's memory pointer.
        fp = (struct my_file *)malloc(sizeof(struct my_file)); // Allocate memory for file structure.
        (*fp).fd = fd;
        (*fp).count = 0;
        (*fp).index = 0;
        return fp;
    }
    else
    {
        return NULL; // When failed to open, return NULL.
    }
}

int my_fclose(struct my_file *fp)
{
    int r;
    r = close((*fp).fd); // Request OS to close file.
    free(fp);            // Free up memory allocalted by malloc for file structure.
    return r;            // When success to close file, return 0, otherwise return -1(EOF).
}

// Check if array is blank. (If array is blank, return ture.)
bool is_array_blank(char *array, int size)
{
    for (int i = 0; i < size; i++)
    {
        if (array[i] != '\0')
            return false;
    }
    return true;
}

int my_fgetc(struct my_file *fp)
{
    int c, size;
    // When count is 0 or buffer is blank, read file and store it in buffer.
    if ((*fp).index == (*fp).count || is_array_blank((*fp).buffer, MyBufferSize))
    {
        // {-1, numberOfBytes} = read(fileDescriptor, buffer, size);
        size = read((*fp).fd, (*fp).buffer, MyBufferSize);
        (*fp).index = 0; // initialize index
        (*fp).count = size;
```

```
69          if (size <= 0) // When failed or end of the file, return EOF.
70              return EOF;
71      }
72      if ((*fp).index == (*fp).count)
73          return EOF;
74      c = (*fp).buffer[(*fp).index]; // Get character from buffer[index]
75      (*fp).index++;                 // increment index
76      return c;
77  }
78
79  // FOR DEBUG (Auther: Prof. SHIKIDA)
80  void print_filestr(struct my_file *fp)
81  {
82      int i;
83      fprintf(stderr, "count:%d index:%d  |", fp->count, fp->index);
84      for (i = 0; i < MyBufferSize; i++)
85      {
86          fprintf(stderr, " %02x", fp->buffer[i]);
87      }
88      fprintf(stderr, " |\n");
89  }
90
91  // (Auther: Prof. SHIKIDA)
92  int main(int argc, char *argv[])
93  {
94      struct my_file *fp;
95      if (argc != 2)
96      {
97          fprintf(stderr, "Usage: myfile filename\n");
98          return 1;
99      }
100     fp = my_fopen(argv[1]);
101     if (fp == NULL)
102     {
103         fprintf(stderr, "my_fopen: can't open %s\n", argv[1]);
104         return 1;
105     }
106     print_filestr(fp);
107     int c;
108     for (;;)
109     {
110         c = my_fgetc(fp);
111         print_filestr(fp);
112         if (c == EOF)
113             break;
114
115         printf("c:%02x\n", c);
116     }
117     my_fclose(fp);
118     return 0;
119 }
```

Listing 2 `output`

```
1  $ cat test.txt
2  abcdefg
3  $ gcc assignment1.c ; ./a.out test.txt
4  count:0 index:0  | 00 00 00 00 00 |
5  count:5 index:1  | 61 62 63 64 65 |
6  c:61
7  count:5 index:2  | 61 62 63 64 65 |
8  c:62
9  count:5 index:3  | 61 62 63 64 65 |
10 c:63
11 count:5 index:4  | 61 62 63 64 65 |
12 c:64
13 count:5 index:5  | 61 62 63 64 65 |
14 c:65
15 count:3 index:1  | 66 67 0a 64 65 |
16 c:66
17 count:3 index:2  | 66 67 0a 64 65 |
18 c:67
19 count:3 index:3  | 66 67 0a 64 65 |
20 c:0a
21 count:0 index:0  | 66 67 0a 64 65 |
```