情報科学研究部用テキスト

テキストエディタを作ろう（実装編： C）

Go Suzuki

目次

ライセンス

1. ファイルを読み書きしよう 楽

では，ラインエディタと呼ばれるものを最初に作ってみよう．ラインエディタは ed が有名である．

1.1. 作るモノ

```
$ ./a.out hoge.txt
> p0,3
hoge
piyo
fuga
funya
> a1
puru
pura
.
> p0,5
hoge
piyo
puru
pura
fuga
funya
> r4,100000
> p0,1000
hoge
piyo
```

```
puru
> q（プログラム終了）
```

作るラインエディタは，引数に読み書きするファイル名を指定する．`>` を表示してコマンドを待機する．コマンドは次の通りである．

(1) q：プログラムを終了する

(2) w：変更を保存する（ファイルに書き出す）

(3) p：指定された範囲の行を表示する

(4) r：指定された範囲の行を削除する

(5) a：指定された行の後に入力された行を追加する（`.` で入力終了．）

範囲指定は，`<start>,<end>` という書式で行う．`<start>` 以上 `<end>` 未満であることを表す．`<start>` とだけ渡された場合は，`<start>` 以上 `<start> + 1` 未満であると解釈する．

1.2. 可変長配列

テキストファイルのデータを保存するために，可変長配列を用意しよう．そのため，次の構造体を用意しよう．

```
struct buffer {
    char * buf;  // buffer
    int len; // length
    int cap; // capacity
};
```

`buf` はバッファへのポインタ，`cap` は確保したバッファのサイズ，`len` は実際の文字列のサイズである．

そして，次の関数を用意してみよう．

```
int newString(char * content, int size, struct buffer * result)
    // content の内容で size 分だけのバッファを確保し，result に格納する．返り値は成功したか．
int insertString(struct buffer * me, char * content, int start, int size)
    // me に content（長さは size）を start の位置から挿入する． 返り値は成功したか．
void removeString(struct buffer * me, int start, int size)
    // me を start の位置から size 分だけ削除する，
```

実装する際は絵を描いて考えてみよう．

1.2.1. 易 `newString` のヒント

- `malloc` を用いてバッファを確保し，`memcpy` で内容をコピーしよう．

1.2.2. 難 `insertString` のヒント

- `me->cap < (me->len + size)` のときは，バッファを確保しなおそう．
- まず，`start` の位置から最後までの文字を `size` だけ後に動かそう．
- 動かしたら，`start` の位置に `content` を書き込もう．

1.2.3. 易 `removeString` のヒント

- `start+size` から最後までの文字を `size` だけ前に動かさなきゃいけない．

1.3. 読み書きする関数を用意しよう

まずはファイルを読み書きする関数を用意してみよう．

```
int openFile(struct buffer * buf, char * path)
    // ファイル読み込みする．内容は buf に格納される．返り値は成功したか
int writeFile(struct buffer * buf, char * path)
    // ファイル書き込みする．返り値は成功したか
```

ファイル読み込みは `fopen` を `r` モードで開く． 戻り値チェックなどは忘れずに行おう．（`openFile` 関数）そして，読み込んで，作った可変長配列に格納してみよう．

1.3.1. 易 `openFile` のヒント

- `newString` して可変長配列を作ろう．
- `gets` と `insertString` でどんどん読み込んでいこう．

1.3.2. 易 `writeFile` のヒント

- `fwrite` で一気にやろう．

1.4. 各処理を作っていこう

各コマンドに対応する処理を作っていこう．

```
// 全て start は始まりの行，end は終わりの行です！
```

```
void printLines(struct buffer * buf, int start, int end)
    // 内容を表示する.
int appendLines(struct buffer * buf, int start, int end)
    // 追加する. end は無視しよう. 返り値は成功したか
void removeLines(struct buffer * buf, int start, int end)
    // 削除する.
```

1.4.1. 易 `printLines` のヒント

- `for` ループと `putc` で 1 文字ずつ処理していこう.
- `\n` が来たら，次の行に移った合図だ.

1.4.2. 難 `appendLines` のヒント

- `printLines` と同様に行をカウントしていこう.
- 挿入は `insertString` を使おう.
- とりあえず 1 行あたり 1024 文字まで入力できる， と仮定しておこう. （可変長はコンソール API を使わなきゃいけない）

1.4.3. 易 `removeLines` のヒント

- `printLines` と同様に行をカウントしていこう.

1.5. コマンドを解析しよう

コマンドの範囲選択のところを解析する関数を作ろう.

```
int parse(char * buf, int * start, int * end) // 返り値は成功したか
```

1.5.1. 難 ヒント

- まず， どこからどこまでが数字かを判断して， しっかりとカンマがあるか， あるいは改行が来て終わるか判別しよう.
- 書式が正しいことを確認したら `sscanf` を用いて，数を読み取ろう.
- 改行ならば終わり，カンマがあるならば，次の数のところからまた `sscanf` を用いよう.

1.6. main を用意しよう

ここまでできた君ならば行けるはず！

1.7. メモリマップトファイルを用いてファイル読み込みをして

みよう

1.7.1. 仮想メモリ

我々が使っているポインタは，アドレスを示すが，そのアドレスは主記憶装置上の物理的な位置を直接指し示すものではない．直接指し示すアドレスのことを物理アドレス，physical address と呼び，我々が普段触っているアドレスを論理アドレス，仮想アドレス，virtual address と呼ぶ．CPU にはメモリ管理ユニット，Memory Management Unit, MMU があって，私たちがポインタをデリファレンス，dereference（`int * a; *a = 0;` とすること）すると，それが高速に仮想アドレスから物理アドレスに変換してくれる．そして，そのメモリ管理ユニットを管理してくれるのが OS である．また，この仕組みのことを仮想メモリ，virtual memory と呼ぶ．

メモリ管理ユニットのおかげで，あるプログラムのメモリ上のデータを他のプログラムから改竄されることを防いだり，メモリの断片化が起こったときに領域を移動させることで解決したり，プロセスがたくさんメモリを欲しがったときにすぐに新しい領域を確保したり，メモリがいっぱいになったときに，データをディスクに退避させる，といったことができるようになる．

1.7.2. メモリマップトファイル

メモリ管理ユニットは参照先が見つからないときに，OS に助けを求める．そのときに，ファイルの内容をメモリに読み込んで，読み込んだ場所を指すようにする．すると，メモリ管理の仕組みによってファイルの読み書きができるようになる！この仕組みやこの仕組みによって読み書きされるファイルのことをメモリマップトファイル，memory mapped file と呼ぶ．システムコールは何回も呼び出すと遅くなるのに対し，この方法であれば高速に大容量のファイルを読み書きすることができる．

1.7.3. メモリマップトファイルと普通の読み書きを比較してみよう

ソースコードをそれぞれ示す．

ライブラリの読み書き

```
#include <stdio.h>

int main(void) {
    char buf[16];
    FILE * fp = fopen("hoge.c", "r");
    for(int i = 0; i < 10000; ++i) {
        fseek(fp, 0L, SEEK_SET);
        while(fread(buf, sizeof(char), 16, fp) != 0);
    }
```

```
    fclose(fp);
    return buf[0];
}
```

read

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(void) {
    char buf[16];
    int fp = open("hoge.c", O_RDONLY);
    for(int i = 0; i < 10000; ++i) {
        lseek(fp, 0L, SEEK_SET);
        while(read(fp, buf, sizeof(buf)) != 0);
    }
    close(fp);
    return buf[0];
}
```

メモリマップトファイル

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
int main(void) {
    int fd;
    char *map;

    struct stat statBuf;

    if (stat("hoge.c", &statBuf) != 0) {
        printf("Error : can't stat¥n");
        return 1;
    }
```

```
    fd = open("hoge.c", O_RDONLY);
    if(fd < 0) {
        printf("Error : can't open file\n");
        return -1;
    }
    map = (char*)mmap(NULL, statBuf.st_size, PROT_READ, MAP_SHARED, fd,
0);
    if(map == MAP_FAILED) {
        printf("Error : mmap failed\n");
        return -1;
    }

    char buf[16];
    int iterate = 0;
    for(int i = 0; i < 10000; ++i)
        for(iterate = 0; iterate < statBuf.st_size; iterate += 16)
            memcpy(buf, (map + iterate),
size <= (iterate + 16) ? (statBuf.st_size - iterate) : 16);
    close(fd);
    munmap(map, statBuf.st_size);
    return 0;
}
```

実行結果は次のようになった.

ライブラリの読み書きの速度

```
$ time ./a.out

real    0m0.035s
user    0m0.013s
sys     0m0.022s
```

`read` の読み書きの速度

```
$ time ./a.out

real    0m0.162s
```

```
user    0m0.029s
sys     0m0.133s
```

メモリマップトファイルの速度

```
$ time ./a.out

real    0m0.004s
user    0m0.003s
sys     0m0.001s
```

速い！特に，`sys`（カーネルが消費した時間）が大きく違う．

だが，注意としては，闇雲にページを作るとOSの処理が遅くなること（たくさんページテーブルエントリができてしまう．）が挙げられる．大きいファイルで，何回も読み込まれる可能性のあるもののみメモリマップトファイルを用いるようにしよう．