

レポート提出票

科目名: 情報工学実験1

実験課題名: 課題5 システム・プログラミング

実施日: 2020年 7月 6日

学籍番号: 4619055

氏名: 辰川力駆

共同実験者:

_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

1 実験目的

本実験では、システムコールを用いたシステムプログラミングを学ぶと共に、プロセスの基本概念、並行プログラミングの基礎を理解する。ここで対象とするオペレーティングシステム (Operating System、OS) は、POSIX(IEEE Std 1003.1) 準拠の各種 UNIX や Linux など UNIX 系 OS である。

2 実験の原理 (理論)

2.1 カーネル

すべてのコンピュータには、OS と呼ばれる基礎的なプログラムの集合があり、その集合の中で最も重要なプログラムがカーネル (Kernel) である。

カーネルの重要な役割は、ハードウェア (デバイス) の制御およびユーザプログラムに対する実行環境の提供である。MS-DOS などの OS ではユーザプログラムが直接デバイスを操作することが許されているが、UNIX 系の OS ではユーザプログラムがデバイスを直接操作したり、任意のメモリ位置へのアクセスすることが禁止されている。同じ種類のデバイスでもメーカーが違えば細かい制御方法が異なるため、カーネルはそのようなデバイス毎の詳細をユーザから隠蔽し、代わりにデバイスを操作するための共通インターフェース (システムコール) を提供する。ユーザプログラムとデバイスの間にカーネルが入ることの利点としては、ユーザは各デバイスの制御方法の詳細を知る必要がないためプログラミングが容易となることや、カーネルが処理を実行する前に処理要求の妥当性の確認ができるため、システムのセキュリティを大幅に向上させられることが挙げられる。また、カーネルが同じインターフェースを提供する限り同じプログラムが実行できるため、プログラムの可搬性も向上させることができる。

2.2 システムコール

ハードウェアと直接やり取りができるのはカーネルだけに限定されているため、ユーザプログラムでハードウェアを操作したいときは、カーネルを通して間接的に操作することになる。そのために使うのがシステムコールである。プログラムからのシステムコールの呼び出し方の見た目は通常の変数と変わらないが、カーネルに仕事させるためにはシステムコールを使わなければならない。通常の変数とシステムコールの違いは、それがカーネルに対する明示的な要求かどうかである。

2.3 ライブラリ関数

プログラムを作る際、システムコール以外にも様々なライブラリ関数を使用できる。最も代表的なものとしては標準 C ライブラリがある。標準 C ライブラリには `printf()`、`exit()`、`strlen()` などが含まれる。ライブラリ関数もシステムコールを使って実装されている場合があり、例えば `printf()` は `write()` というシステムコールが使われている。通常、各システムコールに対応するライブラリ関数が用意されているが、逆に各ライブラリ関数に対してシステムコールが存在するわけではない。算術関数などはシステムコールを使う必要がない。

関数の定義を調べるためには `man` コマンドが利用可能である。man ページはいくつかのセクションに分かれている。例えば `printf` にはコマンドとライブラリ関数があるため、それぞれ

```
$ man 1 printf # ユーザコマンド
$ man 3 printf # ライブラリ関数
```

のようにセクションを指定して検索する必要がある。以後、本実験では関数名の後ろにそれがシステムコールならば(2)、ライブラリ関数ならば(3)というようにセクション番号をつけてどの関数を使用しているかわかるようにする。

2.4 プロセス

プロセスの考え方は、どのような OS でも基本となる考え方であり、プログラムの実体として定義されている。プロセスは、プログラムを実行した際などに生成され、場合によっては子プロセスをさらに生成し、処理が終わると最後は消滅する。カーネルはプロセスにシステムの資源を割り当て、状態を管理する必要があるので、詳細は触れないがプロセスディスクリプタを用いてプロセスを管理している。各プロセスには重複しない番号 (プロセス ID) が振られており、ユーザはプロセス ID を使うことで、プロセスを一意に指定することができる。新しく生成されたプロセスのプロセス ID は直前に割り振られたプロセス ID+1 となる。プロセス ID が 0 のプロセスはシステム起動時に生成される特別なプロセスである。

2.5 ファイル

ファイルとはバイト列として構造化された情報の入れ物である。UNIX 系 OS ではファイルはすべて一つの木構造で階層的に管理されており、カーネルはファイルの中身を関知しない。枝分かれの部分はディレクトリと呼ばれる特別なファイルがあり、その枝にぶら下がっている下位の階層のファイルを管理している。ディレクトリの中で最も上位に位置するのをルートディレクトリと呼び、ルートディレクトリから順番にディレクトリを辿ることで任意のファイルを指定することができる。この木構造の中には、ディレクトリだけでなくテキストファイルやバイナリファイル、シンボリックリンク、デバイスファイルが含まれる。これらは全てファイルとして扱われる。

一般ファイルは、テキストファイルやプログラムなどである。カーネルはファイルの中身を単にバイト列として構造化されたものとして扱う。このようなファイルのことをストリームファイルと呼ぶ。デバイスファイルは、各種デバイスである。デバイスファイルもストリームファイルとして一般ファイルと同じように扱うことができる。プログラムからストリームファイルにアクセスするとき、ファイルディスクリプタというものを使う。ファイルディスクリプタはプログラムから見ると単なる整数であるが、カーネル側ではオープンされたストリームファイルと対応づけられている。通常は `open(2)` を用いてファイルをオープンし、対応するファイルディスクリプタを取得する必要があるが、標準入力、標準出力、標準エラー出力の 3 つはプログラム実行時に自動的にオープンされる。それぞれのファイルディスクリプタは 0、1、2 である。通常、標準入力にはキーボードが、標準出力および標準エラー出力にはディスプレイが割り当てられている。

3 実験環境

- MacBook Pro(16-inch,2019)
 - ProductName: Mac OS X
 - ProductVersion: 10.15.5
 - BuildVersion: 19F101
 - プロセッサ: 2.6 GHz 6 コア Intel Core i7
 - メモリ: 16 GB 2667 MHz DDR4
- gcc version : 9.3.0

4 実験結果・考察

4.1 レポート課題 1

fgets の他に gets という関数があるが、この関数は“絶対に”使ってはならない。gets の定義を調べ、使用してはならない理由について説明せよ。

gets 関数は `char *gets(char *s)` である。標準入力から 1 行を読み込み、改行コード `\n` または EOF にたどり着くと、ヌル文字 `\0` を追加して `char *` 型で返します。入力した文字列が長くて用意している限界の配列の文字数を超えた場合、割り当てられたメモリ範囲を超えて書き込みが行われ、バッファオーバーフローとなる。よって gets 関数が引き起こす暴走が非常に危険であるから“絶対に”使ってはならない。

これに対処する解決法としては、gets を使わずに、代わりに fgets を使えばよい。

4.2 レポート課題 2

問題 2-1 ~ 問題 2-3 の結果をそれぞれまとめて考察せよ。

問題 2-1

ソースコード `read_2_1byte.c` とソースコード `read_3_fgetc.c` のプログラムを用いて、様々なファイルの処理時間を計測して比較せよ。時間の計測には `time` 関数を用い、読み込んだファイルは `/dev/null` へリダイレクトせよ。

ソースコード 1: read_2_1byte.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #define NBUF 1 //buf size
8
9  void die(const char *s)
10 {
11     perror(s);
12     exit(1);
13 }
14
15 void cat(const char *path)
16 {
17     ssize_t n;
18     unsigned char buf[NBUF];
19     int fd = open(path, O_RDONLY);
20     if (fd < 0)
21         die(path);
22     for (;;)
23     {
24         n = read(fd, buf, NBUF);
25         if (n < 0)
26             die(path);
27         if (n == 0)
28             break;
29         if (write(STDOUT_FILENO, buf, n) < 0)
30             die(path);
31     }
32     if (close(fd) < 0)
33         die(path);
34 }
35
36 int main(int argc, char *argv[])
37 {
38     if (argc < 2)
39     {
40         fprintf(stderr, "usage:%s file\n", argv[0]);
41         exit(1);
42     }
43     int i;
44     for (i = 1; i < argc; ++i)
45     {
46         cat(argv[i]);
47     }
48     return 0;
49 }

```

ソースコード 2: read_3_fgetc.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void die(const char *s)
5  {
6      perror(s);
7      exit(1);
8  }
9
10 void cat(const char *path)
11 {
12     FILE *f = fopen(path, "r");
13     if (!f)
14         die(path);
15
16     int c;
17     while ((c = fgetc(f)) != EOF)
18     {
19         if (putchar(c) < 0)
20             die(path);
21     }
22
23     if (fclose(f))
24         die(path);
25 }
26
27 int main(int argc, char *argv[])
28 {
29     if (argc < 2)
30     {
31         fprintf(stderr, "usage:%s file\n", argv[0]);
32         exit(1);
33     }
34
35     int i;
36     for (i = 1; i < argc; ++i)
37     {
38         cat(argv[i]);
39     }
40
41     return 0;
42 }
```

表 1: read_2_1byte.c と read_3_fgetc.c の処理時間

ファイル名	read_2_1byte.c			read_3_fgetc.c		
	real	user	sys	real	user	sys
sample_10.txt	0m0.006s	0m0.002s	0m0.002s	0m0.005s	0m0.001s	0m0.002s
sample_100.txt	0m0.007s	0m0.002s	0m0.003s	0m0.005s	0m0.001s	0m0.002s
sample_4096.txt	0m0.013s	0m0.005s	0m0.007s	0m0.005s	0m0.001s	0m0.002s
sample_8192.txt	0m0.024s	0m0.007s	0m0.013s	0m0.006s	0m0.002s	0m0.002s
sample_1000000.txt	0m1.325s	0m0.498s	0m0.791s	0m0.038s	0m0.027s	0m0.003s
sample_10000000.txt	0m12.755s	0m4.850s	0m7.616s	0m0.189s	0m0.174s	0m0.007s

問題 2-1 の結果は上記の表に示した。この表により、fgetc(3)、putchar(3) を用いて実行したときの方が実行時間が短かったと分かる。また、sample_10.txt や sample_100.txt など実行時間が早いので差が少ない。

問題 2-2

ソースコード read_2_1byte.c の NBUF を 2048 に変更し、実行時間がどのように変化するか調べよ。

表 2: read_2_1byte.c と read_2_2048byte.c の処理時間

ファイル名	read_2_1byte.c			read_2_2048byte.c		
	real	user	sys	real	user	sys
sample_10.txt	0m0.006s	0m0.002s	0m0.002s	0m0.006s	0m0.002s	0m0.002s
sample_100.txt	0m0.007s	0m0.002s	0m0.003s	0m0.009s	0m0.002s	0m0.003s
sample_4096.txt	0m0.013s	0m0.005s	0m0.007s	0m0.009s	0m0.002s	0m0.003s
sample_8192.txt	0m0.024s	0m0.007s	0m0.013s	0m0.007s	0m0.002s	0m0.003s
sample_1000000.txt	0m1.325s	0m0.498s	0m0.791s	0m0.012s	0m0.002s	0m0.004s
sample_10000000.txt	0m12.755s	0m4.850s	0m7.616s	0m0.028s	0m0.005s	0m0.013s

問題 2-2 の結果は上記の表に示した。この表により、NBUF を 2048 にして実行したときの方が実行時間が短かったと分かる。また、問題 2-1 より明らかに早くなっているため、ライブラリ関数を用いるより、システムコールを用いる方が早くなることが分かった。

問題 2-3

ソースコード read_2_1byte.c、ソースコード read_3_fgetc.c、問題 2-2 で修正したコード、ソースコード read_3_fgets.c において、システムコールがどのように呼ばれているか確認せよ。

ソースコード 3: read_3_fgets.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7
8  #define NBUF 2048 //buf size
9
10 void die(const char *s)
11 {
12     perror(s);
13     exit(1);
14 }
15
16 void cat(const char *path)
17 {
18     unsigned char buf[NBUF];
19     FILE *f = fopen(path, "r");
20     if (!f)
21         die(path);
22     int c;
23     while (fgets(buf, NBUF, f) != NULL)
24     {
25         printf("%s\n", buf);
26     }
27     if (fclose(f))
28         die(path);
29 }
30
31 int main(int argc, char *argv[])
32 {
33     if (argc < 2)
34     {
35         fprintf(stderr, "usage:%s_file\n", argv[0]);
36         exit(1);
37     }
38     int i;
39     for (i = 1; i < argc; ++i)
40     {
41         cat(argv[i]);
42     }
43
44     return 0;
45 }
```


できません

4.3 レポート課題3

演習2 で作ったプログラムを基に、標準入力から読み込み、その行数を出力するコマンドを作成せよ（「wc -l」と同等）。

ソースコード 4: kadai3.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7
8  #define NBUF 2048 //buf size
9
10 void die(const char *s)
11 {
12     perror(s);
13     exit(1);
14 }
15
16 int totalcnt = 0; //total の行数を数える変数
17 void wc_l(const char *path)
18 {
19     int cnt = 0; // 行数を数える変数
20     unsigned char buf[NBUF];
21
22     FILE *f = fopen(path, "r");
23     if (!f)
24         die(path);
25     int c;
26     while (fgets(buf, NBUF, f) != NULL)
27     {
28         cnt++; // 一行読み込むごとにカウントする
29     }
30     if (fclose(f))
31         die(path);
32
33     printf("%8d_%s\n", cnt, path); // ファイル名と行数を出力する
34     totalcnt += cnt; // 行数をtotal に足す
35 }
36
37 int main(int argc, char *argv[])
38 {
39     if (argc < 2)
40     {
41         fprintf(stderr, "usage:%s_file\n", argv[0]);
42         exit(1);
```

```

43     }
44     int i;
45     if (argc == 2) // 一つのtxtしか読み込まない場合は行数のtotalがいない
46     {
47         for (i = 1; i < argc; ++i)
48         {
49             wc_l(argv[i]);
50         }
51     }
52     else // 複数の行数を読み込んだ場合は最後にtotalの行数を出力する。
53     {
54         for (i = 1; i < argc; ++i)
55         {
56             wc_l(argv[i]);
57         }
58         printf("%8d_%s\n", totalcnt, "total"); //totalの行数を表示
59     }
60
61     return 0;
62 }

```

上記に作成したソースコード (kadai3.c) を示した。

```

$ ./kadai3 sample_100.txt
10 sample_100.txt
$ ./kadai3 sample_10.txt sample_4096.txt
1 sample_10.txt
410 sample_4096.txt
411 total

```

4.4 レポート課題4

演習5で作成したプログラムにリダイレクト「>」の機能を実装せよ。親プロセス側、小プロセス側どちらでファイルにリダイレクトしても構わないが、実行して正常に動作することを確認めること。また、実行結果をレポートに記載すること。

ソースコード 5: kadai4.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7  #include <fcntl.h>
8
9  #define MAX_LINE_IN 1000
10 #define MAX_ARGS 30
11
12 void redirect(char *args[], int potnt); // リダイレクトする関数
13

```

```

14  int main(int argc, char *argv[])
15  {
16      int pid, status;
17      char line_in[MAX_LINE_IN];
18      char *args[MAX_ARGS];
19      int nargs;
20
21      //tokenize
22      for (;;)
23      {
24          printf(">");
25          if (fgets(line_in, MAX_LINE_IN, stdin) == NULL)
26              exit(0);
27          line_in[strlen(line_in) - 1] = '\0';
28
29          char *token = strtok(line_in, " ");
30          nargs = 0;
31          args[nargs++] = token;
32          while (token != NULL)
33          {
34              if (token != NULL)
35              {
36                  token = strtok(NULL, " ");
37                  args[nargs++] = token;
38              }
39          }
40          args[nargs] = '\0';
41
42          if (strcmp(args[0], "exit") == 0)
43          {
44              return 0;
45          }
46
47          if (fork() == 0)
48          {
49              int i;
50              for (i = 0; args[i] != NULL; i++)
51              {
52                  if (strcmp(args[i], ">") == 0) // 出力先を切り替えるリダイレクトが存在すればそこへ出力
53                  {
54                      redirect(args, i); // リダイレクトする関数
55                  }
56              }
57              execvp(args[0], args);
58              printf("command not found\n");
59              exit(1);
60          }
61          else
62          {
63              wait(&status);
64          }

```

```

65     }
66
67     return 0;
68 }
69
70 void redirect(char *args[], int point) // リダイレクトする関数
71 {
72     int fd;
73
74     args[point] = '\\0'; // ">"があった場所を終端文字にする
75     fd = open(args[point + 1], O_WRONLY | O_CREAT, 0664);
76     close(STDOUT_FILENO);
77     dup(fd);
78     execvp(args[0], args);
79 }

```

上記に作成したソースコード (kadai4.c) を示した。

5 結論

今回の実験を通してシステムコールやライブラリ関数の意味、用途を理解し、実際に C 言語とともに用いてそれらを実装かつ実行することができた。また、シェルを C 言語を用いて自作し、それによってシェルそのものの仕組みがいかなるものであるかを学ぶことができた。

参考文献

- [1] 東京理科大学工学部情報工学科 情報工学実験 1 2020 年度東京理科大学工学部情報工学科出版
- [2] C 言語講座：gets() と scanf() の問題点の解決
<http://www1.cts.ne.jp/clab/hsample/I0/I016.html>
 最終閲覧日:2020/7/6
- [3] LaTeX で色付きソースコードを貼り付け
<http://yu00.hatenablog.com/entry/2015/05/14/214121>
 最終閲覧日:2020/7/6