OS 課題レポート 2

21T2166D 渡辺大樹

2025 年 7 月 6 日

1　演習資料 5

1.1　演習 1: cat コマンドの実装 (標準 I/O)

資料に示された標準 C ライブラリ関数を用いた cat コマンドのソースコードをコンパイルし、実行した。

ソースコード 1　catstd.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        FILE *f;
        int c;
        f = fopen(argv[i], "r");
        if (f == 0) {
            perror(argv[i]);
            exit(1);
        }
        while ((c = fgetc(f)) != EOF) {
            if (putchar(c) < 0)
                exit(1);
        }
        fclose(f);
    }
    exit(0);
}
```

ソースコード 2　コンパイルと実行

```
$ cc -o catstd catstd.c
$ ./catstd catstd.c
```

実行結果として、‘catstd.c‘自身のソースコードが表示された。これは cat コマンドの挙動として正しく、想定通りの結果となった。

1.2 演習 2: cat コマンドの実装 (低レベル I/O)

次に、システムコールを用いた cat コマンドのソースコードをコンパイルし、実行した。

ソースコード 3　catsys.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 2048
#define TRUE 1

void docat(const char *path);

int main(int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; i++) {
        docat(argv[i]);
    }
    exit(0);
}

void docat(const char *path) {
    int fd;
    unsigned char buf[BUFFER_SIZE];
    int n;

    fd = open(path, O_RDONLY);
    if (fd < 0) {
        perror(path);
        exit(1);
    }
    printf("current␣fd=%d\n", fd);
    while (TRUE) {
        n = read(fd, buf, sizeof buf);
        if (n < 0) {
```

```
33              perror(path);
34              exit(1);
35          }
36          if (n == 0) {
37              break;
38          }
39          if (write(STDOUT_FILENO, buf, n) < 0) {
40              perror(path);
41              exit(1);
42          }
43      }
44      if (close(fd) < 0) {
45          perror(path);
46          exit(1);
47      }
48  }
```

ソースコード 4　コンパイルと実行

```
1  $ cc -o catsys catsys.c
2  $ ./catsys catsys.c
```

実行結果として、ファイルディスクリプタの値（‘current fd=3‘）とソースコードが表示された。標準入出力 (0, 1, 2) 以外のディスクリプタが割り当てられており、想定通りの結果となった。

1.3　演習 3: Makefile の利用

ソースコードを ‘catmain.c‘と ‘catsub.c‘に分割し、‘Makefile‘を用いてビルドを行った。

ソースコード 5　Makefile

```
1  cat: catmain.o catsub.o
2          cc -o cat catmain.o catsub.o
3  catmain.o: catmain.c
4          cc -c catmain.c
5  catsub.o: catsub.c
6          cc -c catsub.c
```

ソースコード 6　make の実行

```
1  $ make
2  cc -c catmain.c
3  cc -c catsub.c
4  cc -o cat catmain.o catsub.o
```

‘make‘コマンドにより、依存関係に従ってコンパイルとリンクが自動的に行われた。

次に ‘touch‘コマンドで ‘catsub.c‘のタイムスタンプを更新し、再度 ‘make‘を実行した。

ソースコード 7　touch と make の再実行

```
$ touch catsub.c
$ make
cc -c catsub.c
cc -o cat catmain.o catsub.o
```

1.3.1　考察

‘catsub.c‘のみが更新されたため、‘make‘は ‘catsub.c‘のコンパイルと、最終的な実行ファイル ‘cat‘のリンクのみを行った。‘catmain.c‘は変更されていないため、再コンパイルは行われなかった。これにより、‘make‘がファイルのタイムスタンプを元に、必要な処理のみを実行する効率的なビルドツールであることが確認できた。

1.4　演習 4: gdb によるデバッグ

‘-g‘オプションを付けてデバッグ情報を付与してコンパイルし、‘gdb‘でデバッグを行った。

ソースコード 8　Makefile (デバッグ用)

```
cat: catmain.o catsub.o
        cc -o cat catmain.o catsub.o
catmain.o: catmain.c
        cc -c -g catmain.c
catsub.o: catsub.c
        cc -c -g catsub.c
clean:
        /bin/rm cat catmain.o catsub.o
```

ソースコード 9　gdb の実行

```
$ make clean
$ make
$ gdb ./cat
(gdb) b docat
Breakpoint 1 at 0x11c9: file catsub.c, line 18.
(gdb) run catsub.c
Starting program: /path/to/cat catsub.c
current fd=3

Breakpoint 1, docat (path=0x7fffffffe4b9 "catsub.c") at catsub.c:18
18 fd = open(path, O_RDONLY);
```

```
(gdb) s
23 printf("current␣fd=%d\n", fd);
(gdb) p path
$1 = 0x7fffffffe4b9 "catsub.c"
(gdb) quit
```

1.4.1 考察

‘gdb‘を用いて、指定した関数 (‘docat‘) でプログラムを停止させたり、ステップ実行 (‘s‘) で 1 行ずつ処理を進めたり、変数 (‘path‘) の内容を確認したりすることができた。これにより、プログラムの動作を詳細に追跡し、問題を発見・修正するデバッグ作業が可能になることがわかった。

2 演習資料 6

2.1 演習 1: fork によるプロセス生成とゾンビプロセス

‘fork‘で子プロセスを生成し、親プロセスが ‘waitpid‘を呼ばずに終了するプログラムを実行した。

ソースコード 10 fork_zombie.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    int pid;
    if ((pid = fork()) < 0) {
        perror("fork␣was␣failed");
        exit(1);
    }
    if (pid == 0) {
        printf("I␣am␣a␣child\n");
        printf("My␣process␣id␣is␣%d.\n", getpid());
        exit(0);
    } else {
        printf("I␣am␣the␣parent.\n");
        sleep(10);
        exit(0);
    }
}
```

実行中に ‘ps‘コマンドでプロセスの状態を確認した。

ソースコード 11　実行と ps コマンド

```
$ cc -o fork_zombie fork_zombie.c
$ ./fork_zombie &
[1] 12345
I am the parent.
I am a child
My process id is 12346.
$ ps aj
  PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND
...
 12345 12346 12345 12345 pts/0 12345 Z+ 0 0:00 [fork_zombie] <defunct>
...
```

2.1.1 考察

子プロセスは先に終了したが、親プロセスが ‘waitpid‘で終了ステータスを回収しなかったため、子プロセスは終了処理を完了できず「ゾンビプロセス」(‘Z+‘, ‘¡defunct¿‘) となった。これは、OS が子プロセスの終了情報を親プロセスが受け取るまで保持し続けるために発生する。親プロセスが終了すると、ゾンビプロセスは init プロセスに引き取られて消滅する。

2.2 演習 2: fork と exec による別プログラムの実行

‘fork‘で生成した子プロセスで ‘exec‘システムコールを呼び出し、‘ls -la‘コマンドを実行した。

ソースコード 12　fork_exec.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    int pid;
    if ((pid = fork()) < 0) {
        perror("fork␣was␣failed");
        exit(1);
    }
    if (pid == 0) {
        printf("(c)␣I␣am␣a␣child.\n");
        printf("(c)␣I␣will␣be␣ls␣command!\n");
        execl("/bin/ls", "ls", "-la", NULL);
        perror("exec␣was␣failed");
        exit(1);
```

```
18      } else {
19          printf("(p)␣I␣am␣the␣parent.\n");
20          int status;
21          waitpid(pid, &status, 0);
22          printf("(p)␣I␣will␣terminate␣myself.\n");
23          exit(0);
24      }
25  }
```

2.2.1 考察

実行すると、子プロセスが ‘execl‘を呼び出した時点で、自身のプロセスイメージが ‘/bin/ls‘に置き換えられ、‘ls -la‘の実行結果が表示された。‘execl‘が成功すると、それ以降の子プロセスのコード (e.g., ‘perror‘) は実行されない。これにより、‘fork‘と ‘exec‘を組み合わせることで、現在のプロセスから新しいプログラムを起動するという、シェルの基本的な機能が実現できることがわかった。

2.3 演習 3: pipe による単方向通信

‘pipe‘を用いて ‘ls -lR . — grep ”file”‘と同様の処理を行うプログラムを実行した。

ソースコード 13　pipe_oneway.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4
5   void die(const char s[]) {
6       perror(s);
7       exit(1);
8   }
9
10  int main(int argc, char *argv[]) {
11      int pid;
12      int fds[2];
13
14      if (argc < 2) {
15          fprintf(stderr, "usage:␣%s␣file\n", argv[0]);
16          exit(1);
17      }
18      if (pipe(fds) < 0) {
19          die("I␣cannot␣make␣a␣pipe");
20      }
```

```
21      if ((pid = fork()) < 0) {
22          die("fork␣was␣failed");
23      }
24      if (pid == 0) {
25          close(0);
26          dup2(fds[0], 0);
27          close(fds[0]);
28          close(fds[1]);
29          execl("/bin/grep", "grep", argv[1], NULL);
30          die("exec␣was␣failed");
31      } else {
32          close(1);
33          dup2(fds[1], 1);
34          close(fds[0]);
35          close(fds[1]);
36          execl("/bin/ls", "ls", "-lR", ".", NULL);
37          die("exec␣was␣failed");
38      }
39  }
```

2.3.1 考察

親プロセス (‘ls‘) の標準出力 (fd=1) をパイプの書き込み口 (‘fds[1]‘) に、子プロセス (‘grep‘) の標準入力 (fd=0) をパイプの読み込み口 (‘fds[0]‘) に ‘dup2‘で接続することで、プロセス間のデータ連携が実現できた。不要なディスクリプタを ‘close‘することが、意図しない動作を防ぐために重要である。シェルのパイプライン機能が、これらのシステムコールによって実現されていることが深く理解できた。

2.4 演習 4: pipe による双方向通信

2 組のパイプを用いて、親プロセスと子プロセス (‘bc‘) が双方向通信を行うプログラムを実行した。

ソースコード 14　pipe_twoway.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      int pid;
8      int c2p[2], p2c[2];
```

```
 9
10      pipe(c2p);
11      pipe(p2c);
12
13      if ((pid = fork()) == 0) {
14          close(0); dup2(p2c[0], 0);
15          close(1); dup2(c2p[1], 1);
16          close(p2c[0]); close(p2c[1]);
17          close(c2p[0]); close(c2p[1]);
18          execl("/usr/bin/bc", "bc", "-lq", NULL);
19          exit(1);
20      }
21      close(p2c[0]); close(c2p[1]);
22      FILE *out = fdopen(p2c[1], "w");
23      FILE *in = fdopen(c2p[0], "r");
24
25      fprintf(out, "s(1.0)\n");
26      fflush(out);
27      char buf[256];
28      fgets(buf, sizeof(buf), in);
29      printf("sin(1.0)␣=␣%s", buf);
30
31      wait(NULL);
32      exit(0);
33  }
```

2.4.1 考察

2 組のパイプを用意し、一方は親から子へ、もう一方は子から親への通信路として使用することで、双方向の対話的な処理が実現できた。親プロセスは ‘bc‘に計算式を送り、‘bc‘から計算結果を受け取ることができた。‘fdopen‘を用いてファイルディスクリプタを ‘FILE*‘ストリームとして扱うことで、‘fprintf‘や ‘fgets‘といった高水準な I/O 関数が利用でき、実装が容易になる点も確認できた。