

操作系统第二次作业

作业安排

5.1

5.2

5.4

操作系统第二次作业

201708010407-吴嘉豪

作业安排

操作系统:

第五章: 5.1 5.2 5.4

第七章: 7.1~7.7

提交方式: 电子档(word,pdf两份)

文件命名: 学号-姓名

提交时间: 10月8号晚上22:00前

5.1

答: 设置初始 `x = 100`, 在子进程中将`x`减一, 在父进程中将`x`加三, 最后的结果是, **子进程和父进程中的变量相互独立, 互不影响**. 因此子进程中`x`的值从100变成99, 父进程中的值从100变成103.

执行结果如图所示

```
(base) jamey@~/githubs/ostep-code/cpi/我的作业$ cd "/home/jamey/git
hubs/ostep-code/cpu-api/我的作业/" && gcc 1.c -o 1 && "/home/jamey/githubs/ostep-code/cpu-api/我的作业/"1
My pid: 7864
child pid = 7865, x = 100
child pid = 7865, x = 99
parent pid = 7864, x = 100
parent pid = 7864, x = 103
(base) jamey@~/githubs/ostep-code/cpu-api/我的作业$
```

代码如下

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4
5 int main() {
6     printf("My pid: %d\n", getpid());
7     int x = 100;
8     int child = fork(); // ! fork() 函数是在unistd.h 中的!
9     if (child < 0) { // fork失败
10         printf("fork error !\n");
11     } else if (child == 0) { // fork成功, 子进程
12         printf("child pid = %d, x = %d\n", getpid(), x);
13         x--;
```

```

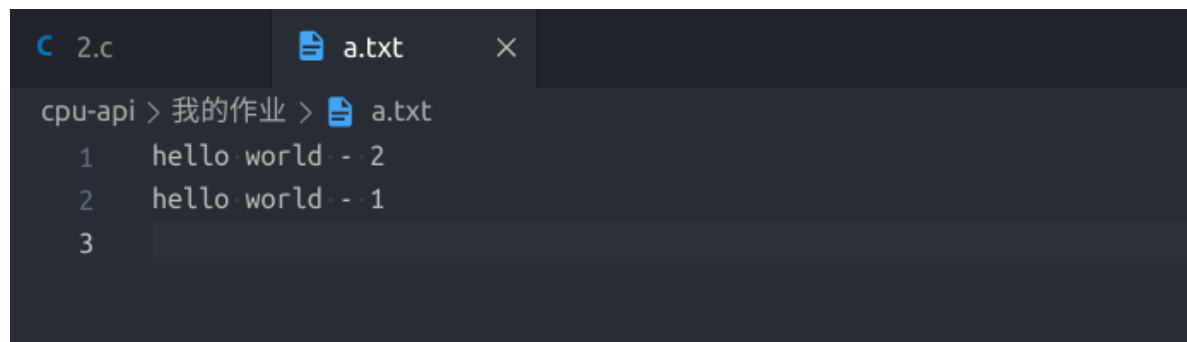
14     printf("child pid = %d, x = %d\n", getpid(), x);
15 } else {
16     wait(NULL); // ! wait() 是在 <sys/wait.h> 中
17     printf("parent pid = %d, x = %d\n", getpid(), x);
18     x += 3;
19     printf("parent pid = %d, x = %d\n", getpid(), x);
20 }
21 }

```

5.2

答: 子进程和父进程都可以访问open()返回的文件描述符. 因为文件是独立于进程存在的.

如果并发分别写入一行字符, 那么将分别写入一行字符. 看似互不影响

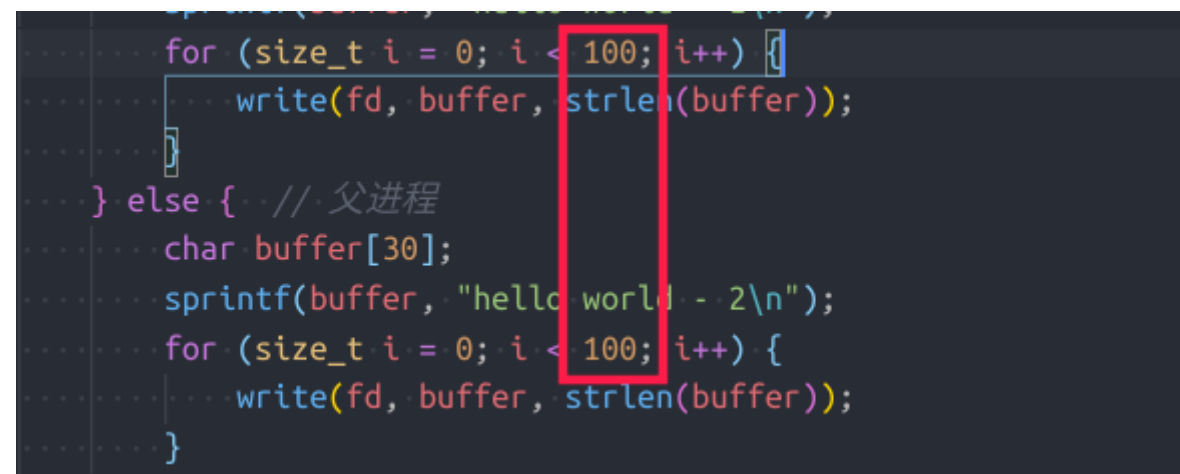


```

cpu-api > 我的作业 > a.txt
1 hello world - 2
2 hello world - 1
3

```

如果分别在循环中写入100行字符, 如图所示



```

... for (size_t i = 0; i < 100; i++) {
...     write(fd, buffer, strlen(buffer));
... }
... } else { // 父进程
...     char buffer[30];
...     sprintf(buffer, "hello world - 2\n");
...     for (size_t i = 0; i < 100; i++) {
...         write(fd, buffer, strlen(buffer));
...     }
... }

```

那么父进程和子进程中**每一个write交替写入文件**, 但是总的写入次数 $100+100 = 200$ 不变.



```

138 hello world - 1
139 hello world - 2
140 hello world - 1
141 hello world - 2
142 hello world - 1
143 hello world - 2
144 hello world - 1
145 hello world - 2

```

```
193 hello world --1
194 hello world --1
195 hello world --1
196 hello world --1
197 hello world --1
198 hello world --1
199 hello world --1
200 hello world --1
201
```

尝试超高并发写入文件的情况. 在每个进程中并行的写入**一百万行字符**.

```
17 ..... sprintf(buffer, "hello world --1\n");
18 ..... for (size_t i = 0; i < 1000000; i++) {
19 .....     write(fd, buffer, strlen(buffer));
20 ..... }
21 ..... } else { // 父进程
22 .....     char buffer[30];
23 .....     sprintf(buffer, "hello world --2\n");
24 .....     for (size_t i = 0; i < 1000000; i++) {
25 .....         write(fd, buffer, strlen(buffer));
26 .....     }
27 ..... }
```

查看结果, 发现**依然成功地写入了两百万行字符**. 并没有出现漏写入的情况. 子进程和父进程中每一行的写入依然是交替进行的.

```
1999994 hello world --1
1999995 hello world --1
1999996 hello world --1
1999997 hello world --1
1999998 hello world --1
1999999 hello world --1
2000000 hello world --1
2000001
```

代码如下:

```
1  #include <assert.h>
2  #include <fcntl.h> // ! 有这个才能调用open函数!
3  #include <stdio.h>
4  #include <string.h>
5  #include <sys/stat.h>
6  #include <sys/types.h> /*提供类型pid_t,size_t的定义*/
7  #include <sys/wait.h>
8  #include <unistd.h>
9
10 int main() {
11     int fd = open("a.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR |
S_IWUSR);
12     int child = fork();
13     if (child < 0) { // 打开失败
14         printf("error\n");
```

```

15     } else if (child == 0) { // 子进程
16         char buffer[30];
17         sprintf(buffer, "hello world - 1\n");
18         for (size_t i = 0; i < 100; i++) {
19             write(fd, buffer, strlen(buffer));
20         }
21     } else { // 父进程
22         char buffer[30];
23         sprintf(buffer, "hello world - 2\n");
24         for (size_t i = 0; i < 100; i++) {
25             write(fd, buffer, strlen(buffer));
26         }
27     }
28     close(fd);
29 }

```

5.4

1. 带 l 的 exec 函数: execl, execlp, execl, 表示后边的参数以可变参数的形式给出且都以一个空指针结束
2. 带 p 的 exec 函数: execlp, execvp, 表示第一个参数 path 不用输入完整路径, 只给出命令名即可, 它会在环境变量 PATH 当中查找命令
3. 不带 l 的 exec 函数: execv, execvp 表示命令所需的参数以 char *arg[] 形式给出且 arg 最后一个元素必须是 NULL
4. 带 e 的 exec 函数: execl 表示, 将环境变量传递给需要替换的进程

以上四点引用自: https://blog.csdn.net/mantis_1984/article/details/52710443

问: 为什么同样的基本调用会有这么多变种?

答: 从上面的四点对命令的解释可以知道, 调用 exec 的形式有很多, 不同的 exec 变体功能都是一样的, 但形式不一样, 比如有以下多种情况:

1. 直接使用 ls 命令
2. 或者写出完整的路径 /bin/ls
3. 是否添加自定义的环境变量
4. 是将命令的每个部分分开来传递
5. 还是先写入到一个 char *argv[] 中然后传递

这符合我们调用程序的多种习惯和系统组织程序调用(Path)的方式.

下面是使用 man 命令得到的各个调用的原型

```

1  int execl(const char *path, const char *arg, ...
2      /* (char *) NULL */);
3  int execlp(const char *file, const char *arg, ...
4      /* (char *) NULL */);
5  int execl_e(const char *path, const char *arg, ...
6      /*, (char *) NULL, char * const envp[] */);
7  int execv(const char *path, char *const argv[]);
8  int execvp(const char *file, char *const argv[]);
9  int execvpe(const char *file, char *const argv[],
10             char *const envp[]);

```

下面展示在子进程中使用各种exec变体调用ls的方式

1. execl:

```
1 | execl("/bin/ls", "ls", NULL);
```

2. execlp(可传递自定义的环境变量)

```
1 | char const *envpath[] = {"PATH=/bin", "AA=/usr", NULL};
2 | execlp("/bin/ls", "ls", NULL, envpath);
```

3. execlp

```
1 | execlp("/bin/ls", "ls", NULL);
```

或者

```
1 | execlp("ls", "ls", NULL);
```

4. execv

```
1 | char *myargs[2];
2 | myargs[0] = strdup("ls");
3 | myargs[1] = NULL;
4 | execv("/bin/ls", myargs);
```

5. execvp

```
1 | char *myargs[2];
2 | myargs[0] = strdup("ls");
3 | myargs[1] = NULL;
4 | execvp("ls", myargs);
5 | // 或者 execvp("/bin/ls", myargs);
```

执行的结果都是(正确打印出了文件列表):

```
(base) jamey@:~/githubs/ostep-code/cpu-api/我的作业$ 1 1.c 2 2.c 3 3.c 4 4.c a.txt
```