

## 操作系统第二次作业

作业安排

5.1  
5.2  
5.4  
7.1  
7.2  
7.3  
7.4  
7.5  
7.6  
7.7

# 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());
```

```

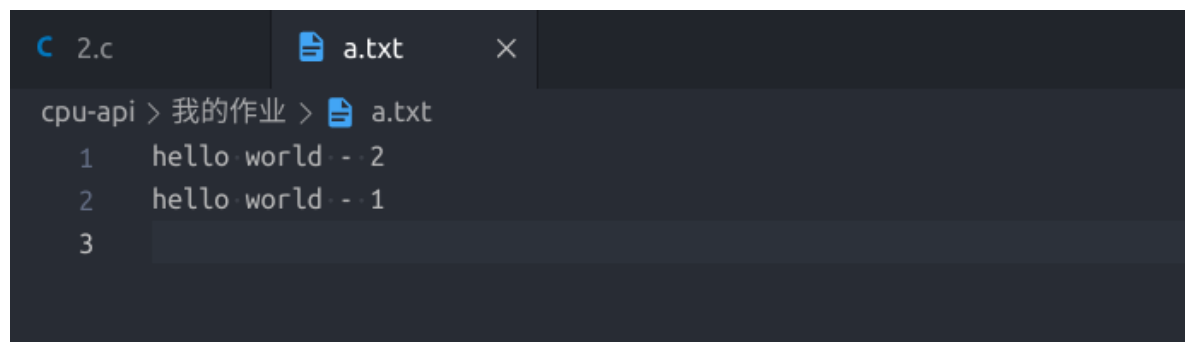
7   int x = 100;
8   int child = fork(); // ! fork() 函数是在unistd.h 中的!
9   if (child < 0) { // fork失败
10      printf("fork error !\n");
11  } else if (child == 0) { // fork成功, 子进程
12      printf("child pid = %d, x = %d\n", getpid(), x);
13      x--;
14      printf("child pid = %d, x = %d\n", getpid(), 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

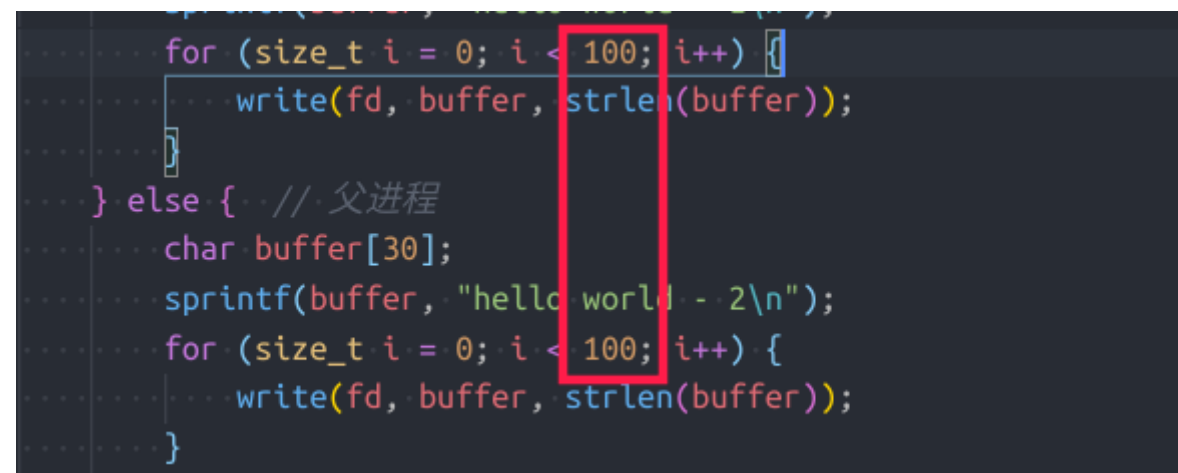


```

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
146  hello world -- 1
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 -- %d\n", i);
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,execle, 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: execle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](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(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. execl(可传递自定义的环境变量)

```

1  char const *envpath[] = {"PATH=/bin", "AA=/usr", NULL};
2  execl("/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

```

## 7.1

### FIFO:

1. 响应时间: 0, 200, 400
2. 平均响应时间: 200
3. 周转时间: 200, 400, 600
4. 平均周转时间: 400

运行命令 `python2 scheduler.py -p FIFO -l 200,200,200 -c` 查看结果.

```
** Solutions **

Execution trace:
[ time  0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
[ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
[ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )

Final statistics:
Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
Job   1 -- Response: 200.00 Turnaround 400.00  Wait 200.00
Job   2 -- Response: 400.00 Turnaround 600.00  Wait 400.00

Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

### SJF:

1. 响应时间: 0, 200, 400
2. 平均响应时间: 200
3. 周转时间: 200, 400, 600
4. 平均周转时间: 400

运行命令 `python2 scheduler.py -p SJF -l 200,200,200 -c` 查看结果

```
** Solutions **

Execution trace:
[ time  0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
[ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
[ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )

Final statistics:
Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
Job   1 -- Response: 200.00 Turnaround 400.00  Wait 200.00
Job   2 -- Response: 400.00 Turnaround 600.00  Wait 400.00

Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

## 7.2

### FIFO:

1. 响应时间: 0, 100, 300
2. 平均响应时间: 133.33
3. 周转时间: 100, 300, 600
4. 平均周转时间: 333.33

运行命令 `python2 scheduler.py -p FIFO -l 100,200,300 -c` 查看结果.

```
** Solutions **

Execution trace:
[ time  0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
[ time 100 ] Run job 1 for 200.00 secs ( DONE at 300.00 )
[ time 300 ] Run job 2 for 300.00 secs ( DONE at 600.00 )

Final statistics:
Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
Job   1 -- Response: 100.00 Turnaround 300.00  Wait 100.00
Job   2 -- Response: 300.00 Turnaround 600.00  Wait 300.00

Average -- Response: 133.33  Turnaround 333.33  Wait 133.33
```

**SJF:**

1. 响应时间: 0, 100, 300
2. 平均响应时间: 133.33
3. 周转时间: 100, 300, 600
4. 平均周转时间: 333.33

运行命令 `python2 scheduler.py -p SJF -l 100,200,300 -c` 查看结果.

```
** Solutions **

Execution trace:
[ time  0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
[ time 100 ] Run job 1 for 200.00 secs ( DONE at 300.00 )
[ time 300 ] Run job 2 for 300.00 secs ( DONE at 600.00 )

Final statistics:
Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
Job   1 -- Response: 100.00 Turnaround 300.00  Wait 100.00
Job   2 -- Response: 300.00 Turnaround 600.00  Wait 300.00

Average -- Response: 133.33  Turnaround 333.33  Wait 133.33
```

## 7.3

对 100, 200, 300 这三个作业使用**RR**调度

1. 响应时间: 0, 1, 2
2. 平均响应时间: 1
3. 周转时间: 298, 499, 600
4. 平均周转时间: 465.67

使用命令 `python2 scheduler.py -p RR -l 100,200,300 -q 1 -c` 查看结果

```

[ time 597 ] Run job    2 for 1.00 secs
[ time 598 ] Run job    2 for 1.00 secs
[ time 599 ] Run job    2 for 1.00 secs ( DONE at 600.00 )

Final statistics:
Job   0 -- Response: 0.00  Turnaround 298.00  Wait 198.00
Job   1 -- Response: 1.00  Turnaround 499.00  Wait 299.00
Job   2 -- Response: 2.00  Turnaround 600.00  Wait 300.00

Average -- Response: 1.00  Turnaround 465.67  Wait 265.67

```

## 7.4

在满足下面几个对工作负载的假设的前提下:

1. 每个工作的运行时间是已知的。
2. 所有的工作同时到达。
3. 一旦开始, 每个工作保持运行直到完成。
4. 所有的工作只是用 CPU (即它们不执行 IO 操作) 。

对于作业按运行长度**非递减顺序增长**的工作负载, SJF提供与FIFO相同的周转时间.

## 7.5

假设有 $n$ 个顺序到达的作业, 工作长度分别为 $J_1, J_2, J_3, \dots, J_n$ .  $Q$ 为RR调度的量子长度. 则

**当 $J_1 = J_2 = J_3 = \dots = J_{n-1} = Q$ 时, SJF与RR提供相同的响应时间**

## 7.6

**除长度最长的工作以外, 其他工作随着工作长度的增加, SJF的响应时间会增加.**

**如果增加工作长度最长的工作的长度, 那么SJF的响应时间不会增加**

对于 100, 200, 300 的作业, 模拟程序结果如下

```

Final statistics:
Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
Job   1 -- Response: 100.00 Turnaround 300.00  Wait 100.00
Job   2 -- Response: 300.00 Turnaround 600.00  Wait 300.00

Average -- Response: 133.33 Turnaround 333.33  Wait 133.33

```

对于作业 120, 200, 300 的作业, 模拟程序结果如下



```
Final statistics:
Job  0 -- Response: 0.00  Turnaround 120.00  Wait 0.00
Job  1 -- Response: 120.00  Turnaround 320.00  Wait 120.00
Job  2 -- Response: 320.00  Turnaround 620.00  Wait 320.00

Average -- Response: 146.67  Turnaround 353.33  Wait 146.67
```

对于作业 120, 240, 300 的作业, 模拟程序结果如下

```
Final statistics:
Job  0 -- Response: 0.00  Turnaround 120.00  Wait 0.00
Job  1 -- Response: 120.00  Turnaround 360.00  Wait 120.00
Job  2 -- Response: 360.00  Turnaround 660.00  Wait 360.00

Average -- Response: 160.00  Turnaround 380.00  Wait 160.00
```

对于作业 120, 240, 360 的作业, 模拟程序结果如下

```
Final statistics:
Job  0 -- Response: 0.00  Turnaround 120.00  Wait 0.00
Job  1 -- Response: 120.00  Turnaround 360.00  Wait 120.00
Job  2 -- Response: 360.00  Turnaround 720.00  Wait 360.00

Average -- Response: 160.00  Turnaround 400.00  Wait 160.00
```

## 7.7

随着量子长度的增加, RR的响应时间会增加.

假设 $n$ 个作业的工作长度分别为 $J_1, J_2, J_3, \dots, J_n$ , 量子长度为 $Q$ , 那么总的响应时间之和 $T_{res}$ 可以表示为

$$T_{res} = \sum_{i=1}^n \min(J_i, Q)$$