

实验 2 进程管理

张配天-2018202180

2020 年 5 月 3 日

目录

| | |
|--------|---|
| 1 实验目的 | 1 |
| 2 实验方法 | 1 |
| 3 程序结构 | 2 |
| 4 实验结果 | 3 |
| 5 问题分析 | 3 |
| 6 代码 | 5 |

1 实验目的

编写一段 c 程序, 实现子进程的创建、子进程的传递、捕捉信号、子进程的通信和加锁。

2 实验方法

因为对线程相关的指令不够熟悉, 在学习作业要求附带的函数说明后又在网上查找相关资料, 进行学习总结。下面针对每个题目给出思路

I. 创建两个子进程并输出信息 ('*child i*')。

- 记两个子进程分别为 P1 和 P2
- 利用 fork 函数创建子进程, 并利用 switch 语句操控主进程和子进程, 使用全局变量 pid1 和 pid2 记录两个子进程的编号, 方便在函数中访问
- 由于创建两个子进程, 为了防止第一个子进程也创建二级子进程 (孙子), 于是先 fork 一个子进程, 在判断为父进程后再创建第二个
- 创建成功后直接在子进程中输出信息 ('*child i*'), 由于创建的时间有差异, 因此会按顺序输出 '*child 1*' 和 '*child 2*'

II. 父进程捕捉键盘的中断信号, 并使子进程输出信息 ('*child process i is killed*')

- 利用 `signal` 函数, 在父进程中捕捉 `SIG_INT` 并将其关联到父进程处理函数 `deposit_ESC_P`, 在该函数中子进程发送用户自定义信号 16,
- 在 P1 中接受自定义信号 16 并将其关联到 `deposit_ESC_1`, 在该函数中输出信息 (*'child process 1 is killed!'*),
- P2 同理, 区别在于不同的处理函数 `deposit_ESC_2`

III. 利用管道实现子进程和父进程之间的通信, 分别按顺序输出信息 (*'child process i is sending a message'*)

- 利用 `pipe` 创建管道, 利用 `close` 关闭特定端口, 利用 `read` 和 `write` 分别实现读取和写入
- 父进程在 `fork` 前创建好管道, 这样子进程才可以继承管道的权限, 每个进程都掌握着管道的读取端和写入端。按照题目要求, 父进程关闭写入端, 两个子进程关闭读取端, 并分别向管道中写入信息

IV. 加锁后重新实现第三问功能

- 利用 `lockf` 函数实现加锁和解锁

以上只是初步思路, 在实现过程中遇到的数个问题及解决方案, 均会在问题分析模块进行详细阐述

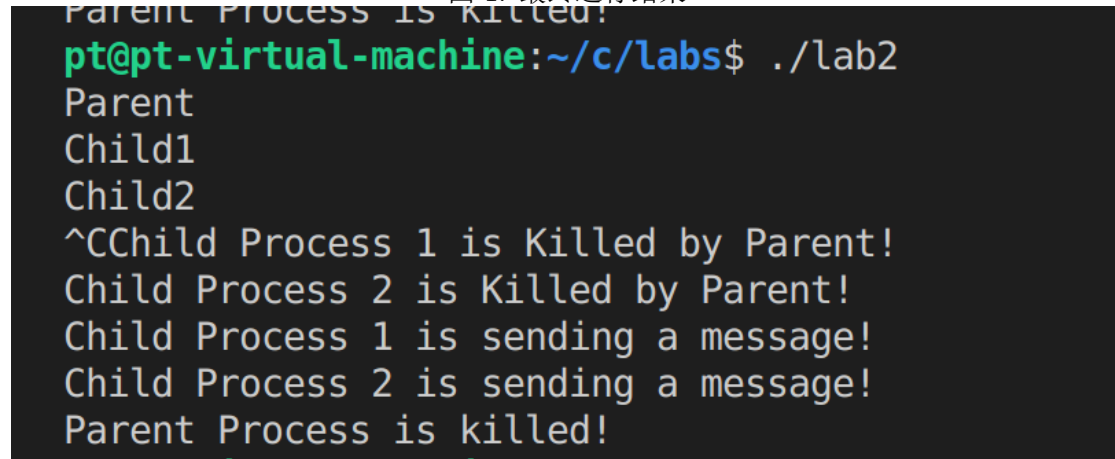
3 程序结构

```
lab2.c
├── pid1,pid2
├── isToWait = 1 在自定义函数中修改, 若为 1 则一直等待
├── void waiting() 循环空转等待键盘指令
├── deposit_ESC_1(int)
├── deposit_ESC_2(int)
├── deposit_ESC_P(int)
├── main()
│   ├── P1(子进程)
│   └── P2(子进程)
```

4 实验结果

在此处给出最终的结果,即实现了所有功能后的运行结果:

图 1: 最终运行结果



```
Parent Process is Killed!
pt@pt-virtual-machine:~/c/labs$ ./lab2
Parent
Child1
Child2
^CChild Process 1 is Killed by Parent!
Child Process 2 is Killed by Parent!
Child Process 1 is sending a message!
Child Process 2 is sending a message!
Parent Process is killed!
```

可见,实现了相应功能,且所有输出均按给定顺序

5 问题分析

针对每一问遇到的问题进行阐述:

- I. 最开始不理解 fork 后程序运行规律,在实践中慢慢理解老师上课讲的 fork 就是把主进程变成两个同样程序,拥有相同的资源,并且执行同样的代码的程序(其中一父一子)。由此,值得注意的是,在父进程中修改的信号关联,(比如将 SIG_INT 关联到自定义函数上)不会在子进程中实现,这一点给我造成了很大的麻烦
- II. 这一问的关键在于 signal 函数捕捉信号,而难点在于输出顺序,下面分开说明:

- 正如 I 所阐述,在这一问中要实现让子进程接受 SIG_INT 并输出信息,绝不能仅仅在父进程中将 SIG_INT 和自定义函数关联起来,在子进程中必须把 SIG_INT 忽视,否则子进程会在键盘接收到 ctrl+c 后按照默认将其判定为 SIG_INT,从而子进程陷入 waiting 函数的死循环之中,导致无法输出信息,正如图 2 所示
- 同时,为了保证 P1 和 P2 的输出信息顺序,所以不能在 deposit_ESC_P 中直接向两个子进程发送信息,否则由于操作系统对于两个子进程调度的不确定性,会导致相反的输出顺序,如图 3 所示:
对此我们有两个解决方案:
 - (a) 利用 sleep 函数,在父进程接收到键盘中断,向 P1 发送信号后休眠 1ms 再向 P2 发送信号,但不美观。
 - (b) 利用 signal 函数,让 P1 在接受主函数的信号并输出信息后再发信号给 P2,这样就实现了顺序,且用全局变量保存 pid1 和 pid2 的优势得以体现。

图 2: 子进程未忽视 SIG_INT

```
pt@pt-virtual-machine:~/c/labs$ ./lab2
Parent
Child1
Child2
^C^C
^C
^C
```

图 3: 同时向两个子进程发送自定义信号

```
pt@pt-virtual-machine:~/c/labs$ ./lab2
Parent
Child1
Child2
^CChild Process 2 is Killed by Parent!
Child Process 1 is Killed by Parent!
```

III. 这一问关键在于两个进程向同一个管道的写入顺序, 但在明确顺序之前, 我还遇到了数个问题, 下面分开说明:

- 无论是向管道写入还是向管道读入, 必须要严格要求读入和写入的字节数, 否则会造成乱码, 如图 4 图 5 所示

图 4: 读取的字符长度大于实际读取的字符长度

```
pt@pt-virtual-machine:~/c/labs$ ./lab2
Parent
Child2
Child1
Child 1 is sending a message!P50F00oz&0000`x&0x&00&000F0 h~0&P50F00t0&000:
0&00& 50F00v0&00& 50F0p0&y0&0s0&H00&000&0x&z0&0'0#00
p0&p0&000&@70F0" 0&(0&0
&000& 0&0&000F0000000&070F0000F0&060F0060F080&0&0be0
0&070F0070F00&!0&070F00&000&0060F0060F00
0&0000h00F00&Genu `00&070F000000uU0y0&00uU00uU^CChild Process 1 is
Killed by Parent!
```

- 利用管道写入端的互斥性, 用 write 函数直接写入字符串 (一次性写入), 保证两个进程写入互不干扰
- 利用 deposit_ESC_1 和 deposit_ESC_2, 调整 P1 向 P2 发信号代码的顺序 (放在写入后), 即可实现 P1 先写入,P2 后写入的要求, 若不然, 则有可能造成相反的写入顺序

图 5: 写入的字符长度大于实际写入的字符长度

```

pt@pt-virtual-machine:~/c/labs$ ./lab2
Parent
Child1
Child2
^CChild Process 1 is Killed by Parent!
Child Process 2 is Killed by Parent!
Child Processor 1 is sending a message!
ge!
Parent Process is killed!
pt@pt-virtual-machine:~/c/labs$

```

- 同时, 主函数必须使用 `wait` 等待 `P1,P2` 退出后再进行读取, 否则会造成僵尸进程和类似的错误结果

IV. 如 III 所述, 直接 `write` 字符串不会有两个进程的互相干扰, 因此考虑把字符串的每一个字符循环输入到管道之中, 对比如下:

- 不给循环写入的部分加锁, 有可能导致两个进程争夺写入端, 从而输出不连贯的字符串, 如图 6 所示, 因为字符串太短, 发生这种事的概率极小

图 6: 未加锁

```

Parent Process is killed!
pt@pt-virtual-machine:~/c/labs$ ./lab2
Parent
Child1
Child2
^CChild Process 1 is Killed by Parent!
Child Process 2 is Killed by Parent!
Child Process 1 is sending a messChild Process 2 is sending a message!
age!
Parent Process is killed!

```

- 加锁之后, 进行数遍测试, 未出现上述情况

6 代码

```
1     #include<stdio.h>
2     #include<pthread.h>
3     #include<signal.h>
4     #include<sys/types.h>
5     #include<unistd.h>
6     #include<string.h>
7     //解决wait的warning
8     #include <sys/wait.h>
9
10    int istoWait = 1;
11    int fd[2];
12    pid_t pid1,pid2;
13    char info_1[50] = "Child Process 1 is sending a message!\n";
14    char info_2[50] = "Child Process 2 is sending a message!\n";
15
16    void waiting(){
17        while (istoWait == 1)
18        {
19            /* code */
20        }
21    }
22
23    void deposit_ESC_1(int what){
24        istoWait = 0;
25        printf("Child Process 1 is Killed by Parent!\n");
26
27
28        close(fd[0]);
29        //向管道写入信息并关闭管道
30        //用循环写入，这样方便对比加锁前后的输出情况
31        lockf(1,1,0);
32        for(int i = 0;i < strlen(info_1);i++){
33            //printf("%c",info_1[i]);
34            write(fd[1],info_1+i,1);
35        }
36
37        //write(fd[1],info_1,strlen(info_1));
38        close(fd[1]);
39        lockf(1,0,0);
40        kill(pid2,17);
```

```
41         exit(0);
42     }
43     void deposit_ESC_2(int what){
44         istoWait = 0;
45         printf("Child Process 2 is Killed by Parent!\n");
46         close(fd[0]);
47         //写入管道并关闭通道
48         //同上
49         lockf(1,1,0);
50         for(int i = 0;i < strlen(info_2);i++){
51             //printf("%c",info_2[i]);
52             write(fd[1],info_2+i,1);
53         }
54         //write(fd[1],info_2,strlen(info_1));
55
56         close(fd[1]);
57         lockf(1,0,0);
58         exit(0);
59     }
60     void deposit_ESC_P(int what){
61         istoWait = 0;
62
63         kill(pid1,16);
64
65         //用sleep和信号都可以完成两个子进程的顺序
66         //sleep(1);
67         //kill(pid2,17);
68     }
69
70     int main(){
71         //int fd_2[2];
72         if (pipe(fd) == -1)
73         {
74             printf("Error in creating pipes");
75         }
76
77         pid1 = fork();
78         //int pid2 = fork();
79         switch (pid1)
80         {
81
82             case 0/* constant-expression */:
```

```
83     istoWait = 1;
84     //忽视键盘中断，否则无法跳出循环
85     signal(SIGINT,SIG_IGN);
86     //关联自定义信号
87     signal(16,deposit_ESC_1);
88     printf("Child1\n");
89
90     waiting();
91
92     //关闭读入管道
93
94     //lockf(1,0,0);
95     break;
96
97 case -1:
98     printf("Error in creating process1\n");
99     break;
100 default:
101     printf("Parent\n");
102     //创建读入缓存区
103     char buff[100];
104
105     //if (pipe(fd_2) == -1)
106     //{
107     //    printf("Error in creating pipes");
108     //}
109     pid2 = fork();
110     switch (pid2)
111     {
112     case 0:
113         istoWait = 1;
114         //关联信号
115         signal(SIGINT,SIG_IGN);
116         signal(17,deposit_ESC_2);
117         printf("Child2\n");
118         //等待键盘指令
119         waiting();
120         //关闭读入通道
121
122
123         break;
124     case -1:
```



```
125         printf("Error in creating process2\n");
126     default:
127
128         //不做处理，因为和下面一样都是父进程
129         break;
130     }
131     //关闭写入通道
132     close(fd[1]);
133
134     //关联信号，一定要把sigint忽视
135     signal(SIGINT, deposit_ESC_P);
136     //等待键盘指令
137     waiting();
138     //接受到ctrlC后发送信号
139
140     //等待子进程结束
141     //必须用循环，否则等待一个的话会造成僵尸进程和输出问题
142     while(-1 != wait(0)){
143         ;
144     }
145     //从同一个buff中读取信息
146     read(fd[0], buff, sizeof(buff));
147
148     //写入输出中
149     write(STDOUT_FILENO, buff, strlen(info_1)+strlen(info_2));
150
151     printf("Parent Process is killed!\n");
152     exit(0);
153     break;
154 }
155 return 0;
156 }
```