

深圳大学实验报告

课程名称： 操作系统

实验项目名称： 实验一 并发程序设计

学 院： 计算机与软件学院

专 业： 计算机科学与技术

指导教师： 阮元

报告人： 刘睿辰 学号： 2018152051 班级： 数计班

实 验 时 间： 2021.3.10-2021.4.3

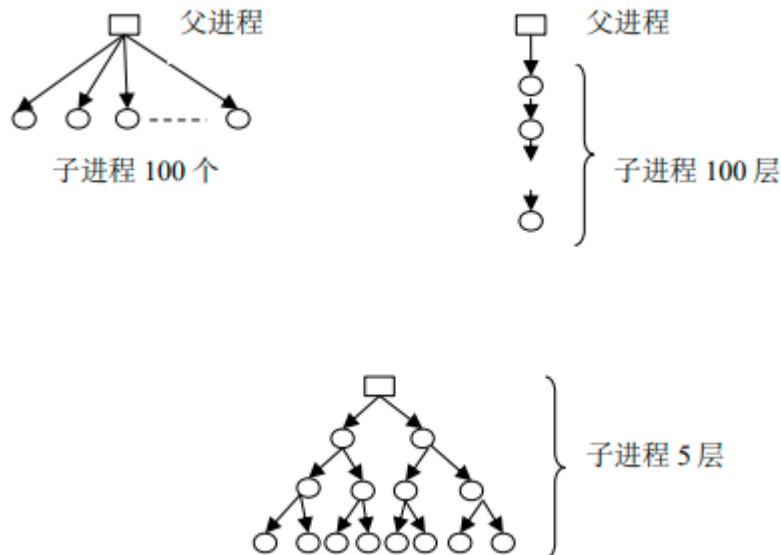
实验报告提交时间： 2021.4.3

一、实验目的与要求：

通过进程的创建、撤销和运行加深对进程概念和进程并发执行的理解，明确进程与程序之间的区别。

二、方法、步骤：

1. 掌握在 linux 中编程编译运行的方法，试验你的第一个 helloworld 程序；
2. 学习预备材料和后面的阅读例程，理解函数 `fork()`、`execl()`、`exit()`、`getpid()` 和 `waitpid()` 的功能和用法；
3. 编写 `hello-loop.c` 程序(在 **helloworld 例程基础上加一个死循环**)。使用 `gcc hello-loop.c -o helloworld` 生成可执行文件 `hello-loop`。并在同一个目录下，通过命令 “`./hello-loop`” 执行之。使用 `top` 和 `ps` 命令查看该进程，记录进程号以及进程状态；
4. 使用 `kill` 命令终止 `hello-loop` 进程；
5. 使用 `fork()` 创建子进程，形成以下父子关系：



通过检查进程的 `pid` 和 `ppid` 证明你成功创建相应的父子关系并用 `pstree` 验证其关系。注意在执行前后，分别检查 `/proc/slabinfo` 中进程控制块 `PCB` 的数量是否能反映出你创建的进程数量变化？

6. 编写代码实现孤儿进程，用 `pstree` 查看孤儿进程如何从原父进程位置转变为 `init` 进程所收养的过程；编写代码创建僵尸进程，用 `ps j` 查看其运行状态；
7. 编写一个代码，使得进程循环处于以下状态 5 秒钟运行 5 秒钟阻塞（例如可以使用 `sleep()`），并使用 `top` 或 `ps` 命令检测其运行和阻塞两种状态，并截图记录。

三、实验过程及内容：

1. 在 linux 环境下运行 helloworld 程序，检测 linux 环境是否正常。

这里面我们需要知道 linux 环境下 C 代码的编译。通过计算机体系结构相关课程的学习，我们了解到我们需要首先建立 `helloworld.c` 文件，在这份文件中编写 C 代码

然后进行编译。编译指令为

gcc helloworld.c -o helloworld

这里面我们通过该指令形成可执行文件 helloworld, 然后通过以下指令就可以执行。

./helloworld

所以我们首先在文本编辑器里编辑如图 1 所示的代码, 然后 gcc 编译, 最后运行结果如图 2 所示, linux 环境正常。

```
1 #include <stdio.h>
2 int main ()
3 {
4     printf("hello_world!\n");
5 }
```

图 1. helloworld 样例程序

```
ruichen@ruichen-virtual-machine:~/桌面/oslab$ gcc helloworld.c -o helloworld
ruichen@ruichen-virtual-machine:~/桌面/oslab$ ./helloworld
hello world!
```

图 2. helloworld 程序编译运行结果

2. 理解函数 fork()、execl()、exit()、getpid() 和 waitpid() 的功能和用法。

1) fork 函数

函数定义: `pid_t pfid = fork();`

函数解释: `pid_t` 类型是一个进程号类型, 在头文件 `<sys/types.h>` 中定义。`fork()` 函数会产生一个新的子进程, 其子进程会复制父进程的数据与堆栈空间, 并继承父进程的用户代码, 组代码, 环境变量、已打开的文件代码、工作目录和资源限制等。关于这个函数的返回值, 如果 `fork()` 成功则在父进程会返回新建立的子进程代码 (PID), 而在新建立的子进程中则返回 0。如果 `fork` 失败则直接返回 -1, 失败原因存于 `errno` 中。该过程如图 3 所示。

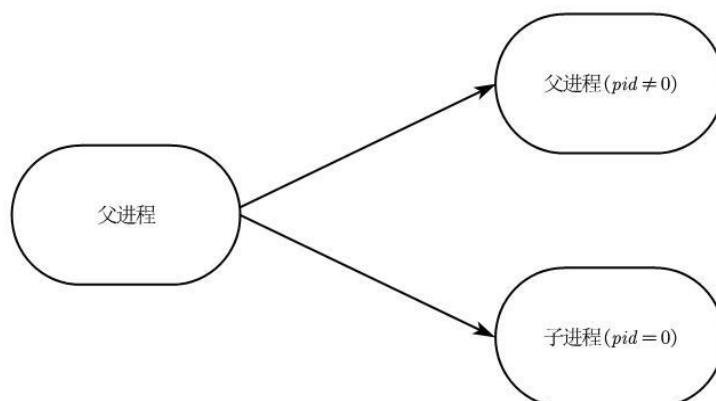


图 3. fork() 函数原理图

2) waitpid 函数

函数定义: `pid_t waitpid(pid_t pid, int * status, int options);`

函数调用: `waitpid(pid, NULL, 0);`

函数解释: `waitpid()` 会暂时停止目前进程的执行, 直到有信号来到或子进程结束。如果在调用 `waitpid()` 时子进程已经结束, 则 `wait()` 会立即返回子进程结束

状态值。子进程的结束状态值会由参数 `status` 返回，而子进程的进程识别码也会一并返回。如果不在意结束状态值，则参数 `status` 可以设成 `NULL`。参数 `pid` 为欲等待的子进程识别码，其他数值意义如下

`pid < -1` 等待进程组识别码为 `pid` 绝对值的任何子进程；

`pid = -1` 等待任何子进程，相当于 `wait()`；

`pid = 0` 等待进程组识别码与目前进程相同的任何子进程；

`pid > 0` 等待任何子进程识别码为 `pid` 的子进程；

关于返回值，如果执行成功则返回子进程识别码(PID)，如果有错误发生则返回-1。失败原因存于 `errno`。

3) `getpid` (取得进程识别码)

函数定义: `pid_t getpid(void);`

函数解释: `getpid()` 用来取得目前进程的进程识别码，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。该函数返回目前进程的进程识别码。

这里面再介绍一个 `getppid` 函数, 这个函数返回目前这个进程的父进程的进程号。有时候为了确定两个进程之间的关系可以这个函数和 `getpid` 结合使用。

4) `exit` (正常结束进程)

函数定义: `void exit(int status);`

函数解释: `exit()` 用来正常终结目前进程的执行，并把参数 `status` 返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。

这里，如果 `status = x (x ≠ 0)` 表示异常退出，`status = 0` 则表示正常退出。

5) `execl` 函数 (执行文件)

函数定义: `int execl(const char * path, const char * arg, ...);`

函数解释: `execl()` 用来执行参数 `path` 字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的 `argv(0)`、`argv[1]`……，最后一个参数必须用空指针 (`NULL`) 作结束。

关于返回值，如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 `errno` 中。

调用 `ls` 命令范例: `execl("/bin/ls", "/bin/ls", "-l", "/etc", NULL)`。

3. 编写 `hello_loop.c` 程序 (在 `helloworld` 例程基础上加一个死循环)。使用 `gcc hello_loop.c -o helloworld` 生成可执行文件 `hello_loop`。并在同一个目录下，通过命令 “`./hello_loop`” 执行之。使用 `top` 和 `ps` 命令查看该进程，记录进程号以及进程状态。

我们编写 `hello_loop.c` 程序，如图 4 所示。

```
1 #include <stdio.h>
2 int main ()
3 {
4     while(1)
5     {
6         printf("hello_world!\n");
7     }
8 }
```

图 4. `hello_loop.c` 程序代码

这是一个死循环，然后我们在 `linux` 命令行编译之后运行，发现程序不断输出

helloworld 这个语句。这时符合我们的设计的，但是我们需要看一下进程号以及进程状态。首先，我们用 ps aux 指令来看一下进程状态，如图 5 所示，可以发现进程号为 3946，状态为 R+，这个代表运行中。

root	3901	4.5	0.0	0	0	?	I	3月27	0:14	[kworker/u256:0]
ruichen	3938	0.1	0.2	24476	5000	pts/0	Ss	00:01	0:00	bash
ruichen	3946	26.9	0.0	4508	740	pts/0	R+	00:01	0:31	./hello_loop
ruichen	3953	0.0	0.1	41424	3556	pts/1	R+	00:03	0:00	ps aux

图 5. ps aux 指令查看进程状态

然后我们再用 top 指令查看进程实时状态，动态地查看系统运维状态。简单来说，top 指令就是 linux 系统的“任务管理器”。top 指令的运行结果如图 6 所示。

进程	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
2414	ruichen	20	0	907472	43408	29752	R 30.1	2.2	2:00.27	gnome-terminal-
3946	ruichen	20	0	4508	740	676	R 26.2	0.0	0:04.87	hello_loop
3901	root	20	0	0	0	0	I 19.2	0.0	0:02.45	kworker/u256:0
769	root	20	0	713472	44952	6116	S 13.2	2.2	13:51.72	snappd
1697	gdm	20	0	888656	46112	33496	S 3.0	2.3	0:00.56	gsd-color
1946	ruichen	20	0	414124	57788	33668	S 2.6	2.9	0:15.88	Xorg
2077	ruichen	20	0	3001800	179836	70900	S 2.3	8.9	0:28.72	gnome-shell
308	root	20	0	0	0	0	S 0.7	0.0	0:33.48	jbd2/sda1-8
501	root	-51	0	0	0	0	S 0.3	0.0	0:00.36	irq/16-vmwgfx
1116	root	20	0	189260	7576	5052	S 0.3	0.4	0:03.48	vmtotlsd

图 6. top 指令查看进程状态

从图 6 中我们看到，hello_loop 进程的 PID 是 3946，CPU 占用率为 26.2%。CPU 占用率事实上是动态变化的。

4. 使用 kill 命令终止 hello_loop 进程。如图 7 所示，我们输入进程号，然后杀死，结果如图 8 所示，进程已经被杀死。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab$ kill 3946
```

图 7. kill 指令杀死进程

```
hello world!
hello world!
hello world!已终止
```

图 8. kill 指令杀死进程结果

5. 使用 fork() 创建子进程，形成不同的父子关系。
- 5.1 构造一个父进程拥有 100 个子进程的关系 (tree.c)。
- 一个父进程拥有 100 个子进程的进程关系树如图 9 所示。

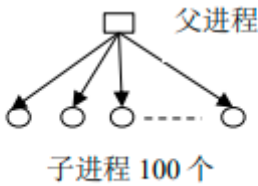


图 9. 进程关系 1

根据进程树我们知道，我们需要进行 100 次循环，然后每次在进程号为 0，也就是子进程的位置加输出信息，输出当前子进程的 PID 以及该子进程的父进程的 PID。根据我们的设计，这 100 个子进程的父进程的 PID 都应该是相等的。代码如图 10 所示。

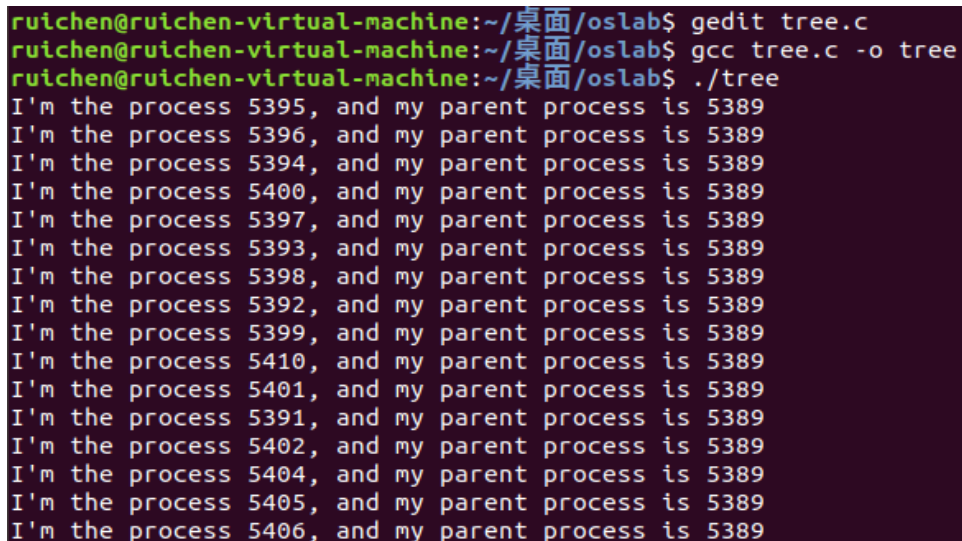
```
for (int i=0; i<100; i++)
{
    fpid = fork();

    if (fpid < 0)
    {
        printf("error\n");
    }
    else if (fpid == 0)
    {
        printf("I'm the process %d, and my parent process is %d\n",getpid(),getppid());
        sleep(50);
        return 0;
    }
}
```

图 10. 让父进程拥有 100 个子进程的代码

此外，当循环结束的时候我们添加一个死循环 while(1)，这样保持进程不终止，我们有充裕的时间来查看进程树。

我们运行程序，得到结果图如图 11 所示。



```
ruichen@ruichen-virtual-machine:~/桌面/oslab$ gedit tree.c
ruichen@ruichen-virtual-machine:~/桌面/oslab$ gcc tree.c -o tree
ruichen@ruichen-virtual-machine:~/桌面/oslab$ ./tree
I'm the process 5395, and my parent process is 5389
I'm the process 5396, and my parent process is 5389
I'm the process 5394, and my parent process is 5389
I'm the process 5400, and my parent process is 5389
I'm the process 5397, and my parent process is 5389
I'm the process 5393, and my parent process is 5389
I'm the process 5398, and my parent process is 5389
I'm the process 5392, and my parent process is 5389
I'm the process 5399, and my parent process is 5389
I'm the process 5410, and my parent process is 5389
I'm the process 5401, and my parent process is 5389
I'm the process 5391, and my parent process is 5389
I'm the process 5402, and my parent process is 5389
I'm the process 5404, and my parent process is 5389
I'm the process 5405, and my parent process is 5389
I'm the process 5406, and my parent process is 5389
```

图 11. tree.c 代码运行结果

显然，结果显示父进程 PID 都是相等的。为了检查数目，我们需要将进程树打印出来。我们使用 pstree 指令，外加参数 -p 可以输出进程号，然后输入父进程的 PID。从图 11 中我们很明显看出父进程的 PID 就是 5389。所以我们打印进程树的指令就是

pstree -p 5389

我们打印出的进程树如图 12 所示。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab$ pstree -p 5389
tree(5389)---tree(5390)
              |---tree(5391)
              |---tree(5392)
              |---tree(5393)
              |---tree(5394)
              |---tree(5395)
              |---tree(5396)
              |---tree(5397)
              |---tree(5398)
              |---tree(5399)
              |---tree(5400)
              |---tree(5401)
              |---tree(5402)
              |---tree(5403)
              |---tree(5404)
              |---tree(5405)
              |---tree(5406)
              |---tree(5407)
              |---tree(5408)

```

图 12. tree.c 所创建的进程树（1）

我们发现进程树是正序排列的，那我们直接去看最下面的部分，如图 13 所示。

```

tree(5478)
---tree(5479)
---tree(5480)
---tree(5481)
---tree(5482)
---tree(5483)
---tree(5484)
---tree(5485)
---tree(5486)
---tree(5487)
---tree(5488)
---tree(5489)

```

图 13. tree.c 所创建的进程树（2）

我们可以看出从 5390 到 5489，一共 100 个子树，这也就创建成功了。

下面我们来看一下进程控制块 PCB 的数量的变化。在没运行程序的时候，我们查看 PCB 的数量如图 14 所示，为 711 个。当运行了该程序之后，我们发现进程控制块的数量变成了 810 个，增加了 100 个。

```

root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# cat /proc/slabinfo |grep task_struct
task_struct 711 725 6016 5 8 : tunables 0 0 0 : slabdata 145 145 0
root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# cat /proc/slabinfo |grep task_struct
task_struct 810 810 6016 5 8 : tunables 0 0 0 : slabdata 162 162 0

```

图 14. tree.c 导致的 PCB 数量的变化

5.2 构造一个链条式的结构 (list.c)

一个链条式的结构如图 15 所示。

在这种结构中，每个父进程都只有一个子进程。

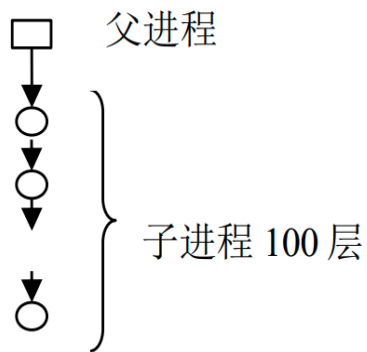


图 15. 链条式结构

当父进程生成了一个子进程之后，父进程不能再产生任何子进程，所以需要马上跳出循环。新产生的子进程则可以输出父子信息，这样方便我们确定从属关系。

注意，在循环的最后我们要有 sleep 语句，否则将会产生孤儿进程，这在后面会加以说明。

代码如图 16 所示。

```
pid_t pid;
for (int i=0; i<100; i++)
{
    pid = fork();
    if (pid < 0)
    {
        printf("error\n");
    }
    else if(pid != 0)
    {
        break;
    }
    else
    {
        printf("I am the process %d, and my parent is process %d\n",getpid(),getppid());
    }
}
while(1){sleep(1);}
return 0;
```

图 16. 链条式结构代码

代码运行结果如图 17 所示。

```
root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# ./list
I am the process 3065, and my parent is process 3064
I am the process 3066, and my parent is process 3065
I am the process 3067, and my parent is process 3066
I am the process 3068, and my parent is process 3067
I am the process 3069, and my parent is process 3068
I am the process 3070, and my parent is process 3069
I am the process 3071, and my parent is process 3070
I am the process 3072, and my parent is process 3071
I am the process 3073, and my parent is process 3072
I am the process 3074, and my parent is process 3073
I am the process 3075, and my parent is process 3074
I am the process 3076, and my parent is process 3075
I am the process 3077, and my parent is process 3076
I am the process 3078, and my parent is process 3077
```

图 17. list.c 运行结果 (1)

从图 17 中的运行结果中我们可以看出，第 n 条语句的父进程就是第 $n-1$ 条语句的进程号，这也就显示出这是一个线性的关系。

我们观察到最后的语句，发现是 3164，这个和第一个子进程 3065，发现正好是 100 个子进程，如图 18 所示。

```
I am the process 3159, and my parent is process 3158
I am the process 3160, and my parent is process 3159
I am the process 3161, and my parent is process 3160
I am the process 3162, and my parent is process 3161
I am the process 3163, and my parent is process 3162
I am the process 3164, and my parent is process 3163
```

图 18. list.c 运行结果 (2)

然后我们用 `pstree` 指令打印出进程树，如图 19 所示，显示这是一个线性的结构。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab$ pstree -p 3064
list(3064)---list(3065)---list(3066)---list(3067)---list(3068)---list(3069)---l+
```

图 19. list.c 创建的进程树

然后我们检测一下 PCB 数量的变化。如图 20 所示，

```
root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# cat /proc/slabinfo |grep task_struct
task_struct 712 740 6016 5 8 : tunables 0 0 0 : slabdata 148 148 0
root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# cat /proc/slabinfo |grep task_struct
task_struct 810 810 6016 5 8 : tunables 0 0 0 : slabdata 162 162 0
```

图 20. list.c 导致的 PCB 数量变化

可以看到，原来的 PCB 数量为 712，程序运行之后 PCB 数量为 810，增加了 98 个。

5.3 构造二叉树结构 (tree2.c)

建立二叉树结构如图 21 所示。

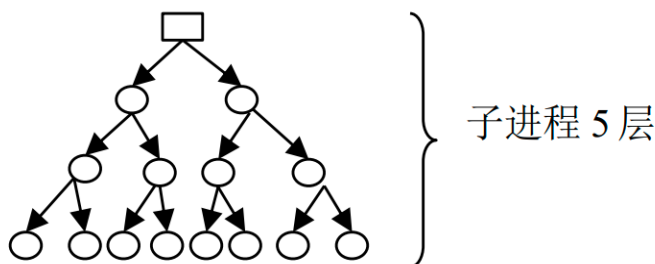


图 21. tree2.c 生成的二叉树结构

我们通过分析二叉树的结构，我们发现如果要建立 5 层二叉树，则应该循环 5 次，每一次要经历两次 `fork()` 操作。但是我们不能连着进行两次 `fork()` 操作，我们来看一下原因。

如图 22 所示，如果两次 `fork()` 操作连续的话，第一次 `fork()` 会在第一个节点生成两个子树。第二次 `fork()` 操作会在两个节点分别生成子树。

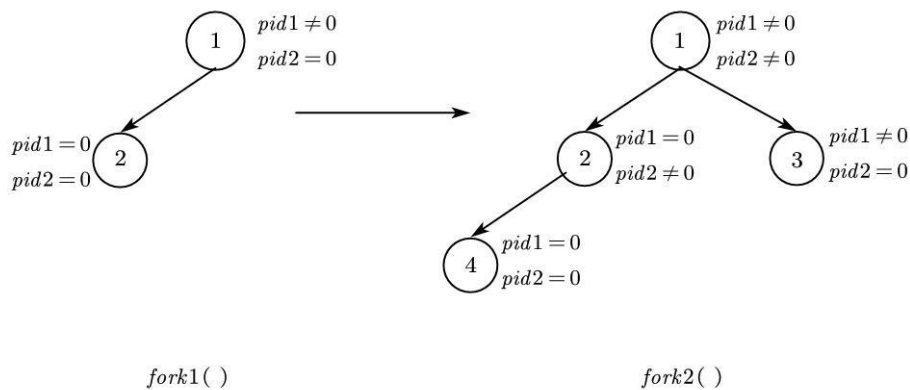


图 22. 连续两次 fork() 的结果

如果按照如图 22 所示的情况，一次循环两个节点分别形成子树，那么第二次循环之后，原来生成子树的 2 号节点将会有三个子树（第二次循环也会生成两个子树），这样的话就不符合设计要求。所以我们需要加一个判断条件，也就是仅仅当 pid1 不为 0 的时候才能添加子树。而当某节点的 pid1 和 pid2 都不为 0 的时候，此时不能再生成子树，所以需要退出循环。

我们的代码如图 23 所示。

```

for(int i=0; i<5; i++)
{
    pid_t fpid1 = fork();
    pid_t fpid2 = 0;
    printf("this is %d, and its parent is %d\n",getpid(),getppid());
    if (fpid1 > 0)
    {
        fpid2 = fork();
        printf("this is %d, and its parent is %d\n",getpid(),getppid());
    }
    if(fpid1>0 && fpid2>0)
    {
        break;
    }
}
  
```

图 23. 生成进程二叉树的代码

接下来我们运行代码，如图 24 所示，我们观察到根节点应该是 4323。然后我们从这个节点开始打印进程树，如图 25 所示。

```

root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# ./tree2
this is 4323, and its parent is 3334
this is 4323, and its parent is 3334
this is 4325, and its parent is 4323
this is 4325, and its parent is 4323
this is 4325, and its parent is 4323
this is 4324, and its parent is 4323
this is 4324, and its parent is 4323
this is 4324, and its parent is 4323
  
```

图 24. 生成进程二叉树的代码运行结果

```

ruichen@ruichen-virtual-machine:~/桌面/oslab$ pstree -p 4323
tree2(4323)---tree2(4324)---tree2(4328)---tree2(4334)---tree2(4338)---tree2(434+
tree2(4340)---tree2(434+
tree2(434+
tree2(434+
tree2(4335)---tree2(4336)---tree2(434+
tree2(434+
tree2(434+
tree2(4339)---tree2(434+
tree2(434+
tree2(4329)---tree2(4331)---tree2(4337)---tree2(434+
tree2(434+
tree2(4382)---tree2(438+
tree2(438+
tree2(4333)---tree2(4362)---tree2(436+
tree2(436+
tree2(4363)---tree2(436+
tree2(436+
tree2(4325)---tree2(4326)---tree2(4368)---tree2(4371)---tree2(437+
tree2(437+
tree2(4373)---tree2(437+
tree2(437+
tree2(4369)---tree2(4370)---tree2(437+
tree2(437+
tree2(4372)---tree2(437+
tree2(437+
tree2(4327)---tree2(4330)---tree2(4350)---tree2(435+
tree2(435+
tree2(4351)---tree2(435+
tree2(435+
tree2(4332)---tree2(4352)---tree2(435+
tree2(435+
tree2(4353)---tree2(435+
tree2(435+
tree2(436+
tree2(436+

```

图 25. tree2.c 所创建的进程树

图 25 右侧显示不全，我们以 4324 为根节点再次打印，结果如图 26 所示，这样我们就确定这就是一个有 5 层子进程的二叉树。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab$ pstree -p 4324
tree2(4324)---tree2(4328)---tree2(4334)---tree2(4338)---tree2(4343)
tree2(4383)
tree2(4340)---tree2(4348)
tree2(4349)
tree2(4335)---tree2(4336)---tree2(4341)
tree2(4345)
tree2(4339)---tree2(4344)
tree2(4347)
tree2(4329)---tree2(4331)---tree2(4337)---tree2(4342)
tree2(4346)
tree2(4382)---tree2(4384)
tree2(4385)
tree2(4333)---tree2(4362)---tree2(4364)
tree2(4366)
tree2(4363)---tree2(4365)
tree2(4367)

```

图 26. tree2.c 所创建的进程树（部分）

我们再来看一下 PCB 数量的变化。如图 27 所示，增加了 60 个 PCB。我们知道，一共有 5 层子树，那么子进程数量为

$$\sum_{i=1}^5 2^i = 62$$

原来有 PCB 数量为 705，程序运行之后为 765，增加了 60 个，这是符合预期的。

```

root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# cat /proc/slabinfo |grep task_struct
task_struct 705 750 6016 5 8 : tunables 0 0 0 : slabdata 150 150 0
root@ruichen-virtual-machine:/home/ruichen/桌面/oslab# cat /proc/slabinfo |grep task_struct
task_struct 765 765 6016 5 8 : tunables 0 0 0 : slabdata 153 153 0

```

图 27. tree2.c 所导致的 PCB 数量变化

6. 编写代码实现孤儿进程以及僵尸进程，并查看进程状态。(orphan.c 以及 zombie.c)

如果一个正常运行的子进程，父进程退出了但是子进程还在，该进程此刻是孤儿进程，被 init 收养；一个正常运行的子进程，如果此刻子进程退出，父进程没有及时调用 wait 或 waitpid 收回子进程的系统资源，该进程就是僵尸进程，如果系统收回了，就是正常退出。

为了构造孤儿进程，我们就要让父进程退出而其子进程在其后退出。所以我们的设计是，让子进程睡眠时间长于父进程，这样就实现父进程先于子进程退出。代码如图 28 所示。

```

pid_t fpid = fork();
if(fpid < 0)
{
    printf("Error\n");
}
else if (fpid == 0) //子进程
{
    printf("This is a son process.\n");
    sleep(60);
}
else //父进程
{
    printf("This is a parent process.\n");
    sleep(10);
}

```

图 28. orphan.c 代码

我们运行代码，调用 pstree 指令查看进程树，在如图 29 所示的位置找到了 orphan.c 运行的位置。父进程的 PID 为 5070，由它产生的子进程的 PID 为 5071。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab$ pstree -p 4885
bash(4885)——orphan(5070)——orphan(5071)

```

图 29. orphan.c 运行出来的进程树

时隔一段时间之后，再看 orphan 的位置。现在父进程已经退出，只剩下子进程（PID 为 5071），它被 init(systemd)所收养，如图 30 所示。

现在我们再看僵尸进程。

当父进程没有退出而子进程正常退出之后，父进程没有及时调用 wait 或 waitpid 收回子进程的系统资源，该进程就是僵尸进程。为了构建僵尸进程，我们让子进程正常退出，而父进程执行时间长于子进程却不会收回子进程的资源。代码如图 31 所示。

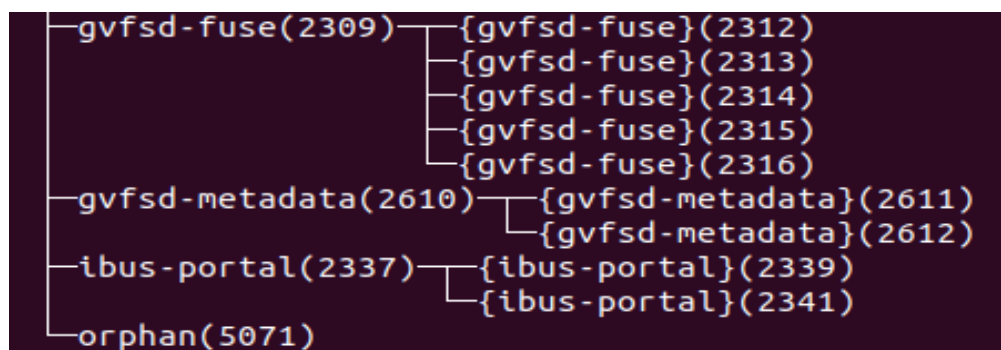


图 30. 子进程(PID=5071)被 init(systemd)收养

```

int fpid = fork();
if (fpid < 0)
{
    printf("Error!\n");
}
else if (fpid == 0)
{
    sleep(5);
    exit(0);
}
else
{
    sleep(30);
}
  
```

图 31. 构造僵尸进程

运行该代码，如图 32 所示，我们得到了这样的运行结果。父进程的 PID 为 5327，其子进程的 PID 为 5328，然后我们用 `ps j` 指令查看状态，如图 33 所示，子进程 (PID=5328) 已经成为了僵尸进程。

```

ruichen@ruichen-virtual-machine:~/桌面/oslab$ ./zombie
this is 5327, its parent is 4885
this is 5328, its parent is 5327
  
```

图 32. zombie.c 运行结果

2708	4885	4885	4885	pts/1	5327	Ss	1000	0:00	bash
2708	4907	4907	4907	pts/0	5329	Ss	1000	0:00	bash
4885	5327	5327	4885	pts/1	5327	S+	1000	0:00	./zombie
5327	5328	5327	4885	pts/1	5327	Z+	1000	0:00	[zombie] <defunct>
4907	5329	5329	4907	pts/0	5329	R+	1000	0:00	ps j

图 33. 僵尸进程结果

7. 编写代码，使进程运行 5 秒钟然后休眠 5 秒，然后查看进程状态。(cons.c)

有了前面的基础，这个程序就相对容易了。我们知道 `sleep` 函数中的参数可以控制程序休眠一段时间，单位为秒 (linux 环境下，windows 依然是毫秒)。但是如何让程序运行 5 秒呢？

在 C 语言中，在标准 C 语言中 `<time.h>` 头文件中宏定义的一个常数 `CLOCKS_PER_SEC`,

用于将 `clock()` 函数的结果转化为以秒为单位的量。而我们知道在 C 语言中, `clock()` 返回该程序从启动到函数调用占用 CPU 的时间, 单位为毫秒。那我们就可以设置一个开始时间, 然后不断用 `clock()` 截取时间, 直到时间差为 5 秒。至于单位转化的问题, 用 `clock()/CLOCKS_PER_SEC` 即可完成转化。

代码如图 34 所示。

```
while(1)
{
    clock_t start = clock()/CLOCKS_PER_SEC;
    int i = 0;
    for (i=0; i<5; ) //running
    {
        clock_t time = clock()/CLOCKS_PER_SEC;
        i = time - start;
    }
    sleep(5); //sleep
}
```

图 34. 交替进程代码

运行代码, 然后我们使用 `ps` 指令查看一下运行时间。我们一般使用 `ps -eo` 指令, 然后输入进程号, 使其输出启动时间以及运行时间, 这里仅仅代表了程序运行的时间, 不包括程序休眠的时间。结果如图 35 所示, 结果表明程序确实运行了 5 秒钟。

```
ruichen@ruichen-virtual-machine:~/桌面/oslab$ ps -eo pid,lstart,etime|grep 3498
3498 Sat Mar 27 01:07:17 2021 00:05
```

图 35. 交替进程代码运行时间

至于休眠时间, 我们已经在系统函数 `sleep` 中进行了规定。

下面我们用 `top` 函数来实时观测该进程的状态。运行程序, 然后在另一个窗口运行 `top` 指令, 可以观察到如图 36 和如图 37 两种进程状态。这就证明程序先运行然后休眠的过程。

进程	USER	PR	NI	VIRT	RES	SHR	%CPU	%MEM	TIME+	COMMAND
5632	ruichen	20	0	4376	712	648	R 52.7	0.0	0:02.41	cons

图 36. 交替进程—CPU 占用率不为 0, 程序处于 running 状态

5635	ruichen	20	0	4376	768	704	S 23.0	0.0	0:04.99	cons
------	---------	----	---	------	-----	-----	--------	-----	---------	------

图 37. 交替进程—CPU 占用率为 0, 程序处于 sleep 状态

如图 36 所示, 显示的是进程处于 running 状态; 如图 37 所示, 程序处于 sleep 状态, 而且 TIME+ 参数代表 CPU 时间总计, 4.99 秒的 running 也符合我们的设计。

四、实验结论：

本次实验通过熟悉 `fork()` 函数、`sleep()` 函数、`waitpid()` 函数等等，实现了进程的创建、不同形状进程树的构造、进程的杀死、孤儿进程、僵尸进程以及交替进程的构造。结果图在第三部分中都已经显示。

五、实验体会：

本次实验第一次通过 `fork` 函数构造进程。在分析进程树的时候，由于父进程会构造子进程，新构建的子进程也会构造子进程，所以有时候想建立设计形态的进程树并非易事，需要仔细思考之后才能创建成功。

关于孤儿进程以及僵尸进程，我们要养成良好的编程习惯，否则生成孤儿进程以及僵尸进程之后将会浪费系统资源。

实验附件

1. `helloworld.c` 为环境测试代码；
2. `hello_loop.c` 为循环代码；
3. `tree.c` 为创建一个父进程拥有 100 个子进程的代码；
4. `list.c` 为创建一个 100 层子进程且每个父进程只有一个子进程的线性结构的代码；
5. `tree2.c` 为创建二叉树结构进程树的代码；
6. `orphan.c` 为创建孤儿进程的代码；
7. `zombie.c` 为创建僵尸进程的代码；
8. `cons.c` 为创建交替进程的代码。

指导教师批阅意见：

成绩评定：

指导教师签字：

年 月 日

备注：