1. *编写一段程序，使用系统调用 fork()创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每个进程在屏幕上显示一个字符，父进程显示“a”；子进程 1 显示“b”；子进程 2 显示“c”。多运行几次，观察并分析显示结果。*

➢ 实验代码：

```
8     int child_process_num = 2;
9     int i;
10    pid_t pid;
11    for (i = 0; i < child_process_num && pid > 0; i ++)
12    {
13        pid = fork();                    1. 父进程创建两个子进程
14    }
15
16    if (pid < 0)                         2. 检查子进程是否创建成功
17    {   // fork failed !
18        printf("fork failed!\n");
19        exit(-1);
20    }
21    else if(pid > 0)                     3. 父进程打印输出 'a'
22    {   // parent process
23        printf("a");
24    }
25    else
26    {   // pid == 0, child process
27
28        if (i == 1)                      4. 子进程1打印输出'b'
29        {   // 1st child process
30            printf("b");
31        }
32        else if (i == 2)
33        {   // 2ed child process         5. 子进程2打印输出'c'
34            printf("c");
35        }
```

➢ **实验结果：**

（多次运行，得到如下两种不同的输出结果）

结果 1：输出 bca

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
bca
```

结果 2：输出 cba

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
cba
```

➢ **结果分析：**

因为进程的异步性，父进程和子进程并发执行时并不能保证两进程间指令执行的先后顺序。在结果 1 中，是子进程 1、子进程 2、父进程依此执行了打印指令；而在结果 2 中，使子进程 2、子进程 1、父进程先后进行打印。

2. ***修改程序，将每个进程输出一个字符改为每个进程输出一句话，观察分析显示结果；***

➢ **实验代码：**

```
8       int child_process_num = 2;
9       int i;
10      pid_t pid;
11      for (i = 0; i < child_process_num && pid > 0; i ++)
12      {
13          pid = fork();              1. 父进程创建两个子进程
14      }
15
16      if (pid < 0)
17      {   // fork failed !
18          printf("fork failed!\n"); 2. 检查子进程是否创建成功
19          exit(-1);
20      }
21      else if(pid > 0)
22      {   // parent process          3. 父进程打印一句话输出
23          printf("the parent process\n");
24      }
25      else
26      {   // pid == 0, child process
27
28          if (i == 1)                4. 子进程1打印一句话
29          {   // 1st child process
30              printf("the 1st child process\n");
31          }
32          else if (i == 2)
33          {   // 2ed child process   5. 子进程2打印一句话
34              printf("the 2ed child process\n");
35          }
```

- **实验结果：**（多次测试，得到如下 4 中不同的输出结果）

结果 1 输出顺序：子进程 1、父进程、子进程 2。

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
the 1st child process
the parent process
the 2ed child process
```

结果 2 输出顺序：父进程、子进程 2、子进程 1。

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
the parent process
the 2ed child process
the 1st child process
```

结果 3 输出顺序：子进程 1、子进程 2、父进程。

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
the 1st child process
the 2ed child process
the parent process
```

结果 4 输出顺序：父进程、子进程 1、子进程 2。

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
the parent process
the 1st child process
the 2ed child process
```

- **结果分析：**

与问题 1 的原因相同，因为进程的异步性，父进程和子进程并发执行时并不能保证两进程间指令执行的先后顺序，因而在多次运行得到了不同的执行结果。

3. 如果在父进程 fork 之前，输出一句话，这句话后面不加“\n”或加“\n”，结果有什么不同，为什么？

- 实验代码：

```
8       int child_process_num = 2;
9       int i;
10      pid_t pid;
11      // printf("something before fork in parent process --- \n");
12      printf("something before fork() in parent process --- ");
13
14      for (i = 0; i < child_process_num && pid > 0; i ++)
15      {
16          pid = fork();
17      }
18
19      if (pid < 0)
20      {   // fork failed !
21          printf("fork failed!\n");
22          exit(-1);
23      }
24      else if(pid > 0)
25      {   // parent process
26          printf("the parent process\n");
27      }
28      else
29      {   // pid == 0, child process
30
31          if (i == 1)
32          {   // 1st child process
33              printf("the 1st child process\n");
34          }
35          else if (i == 2)
36          {   // 2ed child process
37              printf("the 2ed child process\n");
```

换行版本

1. 父进程前输出一句话

不换行版本

2. 父进程创建2个子进程

3. 检查子进程是否创建成功

4. 父进程输出一句话

5. 子进程1输出一句话

6. 子进程2输出一句话

➢ **实验结果 1**（添加换行“\n”）：

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
something before fork in parent process ---
the parent process
the 2ed child process
the 1st child process
```

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
something before fork in parent process ---
the 1st child process
the parent process
the 2ed child process
```

➢ **实验结果 2**（未添加换行“\n”）：

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
something before fork() in parent process --- the parent process
something before fork() in parent process --- the 2ed child process
something before fork() in parent process --- the 1st child process
```

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
something before fork() in parent process --- the 1st child process
something before fork() in parent process --- the parent process
something before fork() in parent process --- the 2ed child process
```

➢ **结果分析：**

添加换行符“\n”与不添加换行符的情况，在多次测试时均得到了多种输出顺序，这与之前实验与分析是相符的。

特别的，发现当添加换行符“\n”时，在父进程前添加的输出仅输出一次（这符合代码逻辑）；但是当不添加换行符“\n”时，父进程前的语句却反常的输出了 3 次（父进程、子进程 1、子进程 2 各一次）。

分析原因，是因为 printf 有数据缓冲区，缓冲区被复制到了子进程。

当不添加“\n”时，父进程前的输出没有刷新缓冲区，即这个字符串仍在缓冲区中；而接下来 fork 函数创建子进程，子进程会复制父进程的数据段和堆帧段，其中包含 printf 的缓冲区，也就是这个字符串也被复制到子进程的 printf 的缓冲区中了；因此，当子进程打印输出时，输出内容前也会有这段被复制的字符串。

当添加“\n”时，父进程前这个字符串在输出时刷新 printf 的缓冲区，缓冲区内容清空，因而不会因 fork 将内容也复制到子进程，也就不会输出多次了。

4. *如果在程序中使用系统调用 lockf 来给临界资源加锁，可以实现临界资源的互斥访问。观察显示结果并分析原因。*

   a） 将 lockf 加在输出语句前后运行试试；

   b） 将一条输出语句变成多条输出语句试试；

   c） 将 lockf 语句放在循环语句外部或内部试试。

➢ *关于 lockf 函数：*

lockf（fd，mode，size），对指定区域（有 size 指示）进行加锁或解锁；

其中，fd 是文件描述字；

mode 是锁定方式，mode=1 表示加锁，mode=0 表示解锁；

size 是指定文件 fd 的指定区域，用 0 表示从当前位置到文件结尾。

（4a）将 lockf 加在输出语句前后运行。

➢ 实验代码：

```
24      else if(pid > 0)    -- 父进程输出打印
25      {   // parent process
26          lockf(1, 1, 0);     // lock  -- 加锁
27          printf("the parent process\n");
28          lockf(1, 0, 0);     // unlock-- 解锁
29      }
30      else
31      {   // pid == 0, child process
32
33          if (i == 1)      -- 子进程1打印输出
34          {   // 1st child process
35              lockf(1, 1, 0);     // lock     -- 加锁
36              printf("the 1st child process\n");
37              lockf(1, 0, 0);     // unlock   -- 解锁
38          }
39          else if (i == 2)-- 子进程2打印输出
40          {   // 2ed child process
41              lockf(1, 1, 0);     // lock     -- 加锁
42              printf("the 2ed child process\n");
43              lockf(1, 0, 0);     // unlock   -- 解锁
44          }
45          else
46          {
47              printf("error!\n");
48              exit(-1);
49          }
```

➢ 实验结果：

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
the parent process
the 1st child process
the 2ed child process
```

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
the 1st child process
the parent process
the 2ed child process
```

➢ 结果分析：

多次运行，得到了不同的输出顺序。因为在加锁区域**仅调用了 printf 一条语句**，不会被打断，也就是在多进程并发运行时一次 printf 输出的内容不会被分割，因而在不加锁的情况下表现与这里的加 lockf 相同。

（4b）将一条输出语句变成多条输出语句

➢ 实验代码：

```
20      if (pid < 0)
21      {   // fork failed !
22          printf("fork failed!\n");
23          exit(-1);
24      }
25      else if(pid > 0)                    ---- 父进程
26      {   // parent process                          -- line_num == 2
27          lockf(1, 1, 0);     // lock     <-- 循环外部加锁
28          for (int j = 1; j <= line_num; j ++)        -- 父进程循环打印两行输出
29              printf("No.%d line of text in parent process\n", j);
30          lockf(1, 0, 0);     // unlock
31      }
32      else
33      {   // pid == 0, child process
34
35          if (i == 0)
36          {   // 1st child process        ---- 子进程1
37              lockf(1, 1, 0);     // lock   <-- 加锁
38              for (int j = 1; j <= line_num; j ++)     -- 子进程1循环打印两行输出
39                  printf("No.%d line of text in the 1th child process\n", j);
40              lockf(1, 0, 0);     // unlock
41          }
42          else if (i == 1)
43          {   // 2ed child process        -- --子进程2
44              lockf(1, 1, 0);     // lock   <-- 加锁
45              for (int j = 1; j <= line_num; j ++)
46                  printf("No.%d line of text in the 2ed child process\n", j);
47              lockf(1, 0, 0);     // unlock
48          }
```

-- 子进程2循环打印两行输出

➢ 实验结果：

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
No.1 line of text in parent process
No.2 line of text in parent process
No.1 line of text in the 1th child process
No.2 line of text in the 1th child process
No.1 line of text in the 2ed child process
No.2 line of text in the 2ed child process
```

```
No.1 line of text in parent process
No.2 line of text in parent process
No.1 line of text in the 2ed child process
No.2 line of text in the 2ed child process
No.1 line of text in the 1th child process
No.2 line of text in the 1th child process
```

```
No.1 line of text in the 1th child process
No.2 line of text in the 1th child process
No.1 line of text in parent process
No.2 line of text in parent process
No.1 line of text in the 2ed child process
No.2 line of text in the 2ed child process
```

➢ 结果分析：

上述实验代码在每个进程中循环打印两行输出，在循环外部加锁（包含整个循环语句）。多次运行，得到多种输出结果。3 个进程（父进程、子进程 1、子进程 2）见的输出顺序有多种，但是 1 个进程的两行输出（for 循环中）总是相连的、不被分隔的。

分析原因，通过 lockf 使得在整个循环区域将标准输出（文件描述符为 fid=1） 这个临界资源不能被其他进程共享，从而在当前进程完成循环中的两行输出前其他进程不能得到标准输出（屏幕）资源，也就使得一个进程的循环中的两行输出不会被分割。

（4c）将 lockf 语句放在循环语句外部或内部

➢ 实验代码：

```
25      else if(pid > 0)
26      {   // parent process                          ---- 父进程
27          for (int j = 1; j <= line_num; j ++)       -- 循环打印输出 line_num 次
28          {
29              lockf(1, 1, 0);     // lock            <-- 循环内加锁
30              printf("No.%d line of text in parent process\n", j);
31              lockf(1, 0, 0);     // unlock          <-- 解锁
32          }
33      }
34      else
35      {   // pid == 0, child process
36          if (i == 0)
37          {   // 1st child process                   ---- 子进程1
38              for (int j = 1; j <= line_num; j ++)
39              {
40                  lockf(1, 1, 0);     // lock
41                  printf("No.%d line of text in the 1th child process\n", j);
42                  lockf(1, 0, 0);     // unlock
43              }
44          }
45          else if (i == 1)                           ---- 子进程2
46          {   // 2ed child process
47              for (int j = 1; j <= line_num; j ++)
48              {
49                  lockf(1, 1, 0);     // lock
50                  printf("No.%d line of text in the 2ed child process\n", j);
51                  lockf(1, 0, 0);     // unlock
52              }
53          }
```

➢ 实验结果：

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
No.1 line of text in the 1th child process
No.2 line of text in the 1th child process
No.1 line of text in parent process
No.2 line of text in parent process
No.1 line of text in the 2ed child process
No.2 line of text in the 2ed child process
```

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
No.1 line of text in the 1th child process
No.2 line of text in the 1th child process
No.1 line of text in the 2ed child process
No.1 line of text in parent process
No.2 line of text in parent process
No.2 line of text in the 2ed child process
```

```
o2igin@Ubuntu2304:~/Projects/OS-2023Fal/lab2$ ./a.out
No.1 line of text in parent process
No.1 line of text in the 1th child process
No.2 line of text in the 1th child process
No.2 line of text in parent process
No.1 line of text in the 2ed child process
No.2 line of text in the 2ed child process
```

➢ **结果分析：**

本次实验将 4b 中的 lockf 从循环外部移动到循环内部，发现多次运行得到的不同输出结果中，不仅不同进程间的输出顺序不同，而且同一进程在循环中的两行输出也有被分隔的情况。

分析原因，是因为此时将 lockf 移动到循环内部，当循环的一次迭代结束下次（运行完解锁）且在进入下次迭代（再次加锁）前，此段时间内标准输出（频幕）资源时没有被占用加锁的，也就是说其他进程可在两次循环间争得频幕资源进行输出，因此出现了实验结果中同一进程的两行输出被分格的情况。

5. 以上各种情况都多运行几次，观察每次运行结果是否都一致？为什么？

上述各种情况的多次运行结果及分析已在实验报告前文给出。总得来说，因为多进程并行时的异步性和不可复现性，在不加锁的情况下难以保证互斥和进程间指令执行的先后顺序，从而得到不同的运行结果。而我们可通过 lockf 对资源加锁的方式保证必要的互斥。

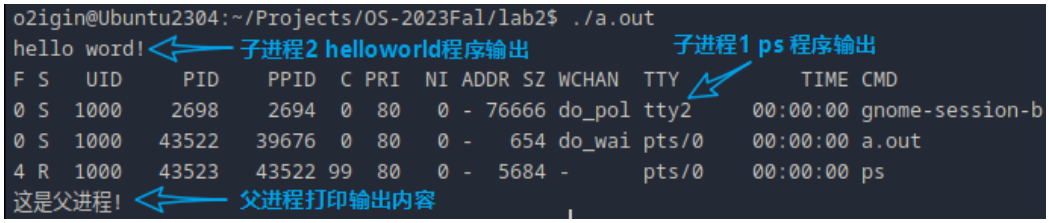
6. 补充实验——exec 函数

在上次 fork 实验的基础上，修改程序，两个子进程分别调用 exec()函数，一个子进程执行系统命令 ps –l -a；另一个子进程执行自己编写的任一程序（例如最简单的在屏幕上打印 hello world 的程序）。父进程则打印一句话：“这是父进程!”

- 实验代码：

```
21      if (pid < 0)
22      {   // fork failed !
23          printf("fork failed!\n");
24          exit(-1);
25      }
26      else if(pid > 0)
27      {   // parent process
28          printf("这是父进程!");        1. 父进程中打印输出
29      }
30      else
31      {   // pid == 0, child process
32          if (i == 0)                  2. 子进程1中执行ps程序
33          {   // 1st child process, 执行ls程序
34              char *argv_list[] = {"ps", "-l", "-a", NULL};
35              int res = execv("/bin/ps",argv_list);
36              if (res < 0)
37                  printf("ls exec error!\n");
38          }
39          else if (i == 1)             3. 子进程2中执行helloworld
40          {   // 2ed child process,  执行helloworld程序
41              char *argv_list[] = {"helloworld", NULL};
42              int res = execv("./helloworld",argv_list);
43              if (res < 0)
44                  printf("helloworld exec error!\n");
45          }
46          else
47          {
48              printf("error!\n");
49              exit(-1);
```

➢ 实验结果：



思考：系统是如何创建进程？当父进程 fork 子进程后，父进程和子进程从程序什么位置开始执行？为什么？

系统如何创建进程：系统首先通过 fork 函数创建子进程，而后执行 exec 函数在当前子进程中执行新的程序。

Fork 后从何位置开始执行：父进程和子进程从 fork 函数的下一条语句开始执行。因为当父进程调用 fork()函数创建子进程时，复制父进程的地址空间到子进程，这个复制过程包括父进程的代码段、数据段和堆栈段等。其中堆栈段包括程序计数器（PC），因此子进程也会从 fork 下一条语句开始执行。