

理论题

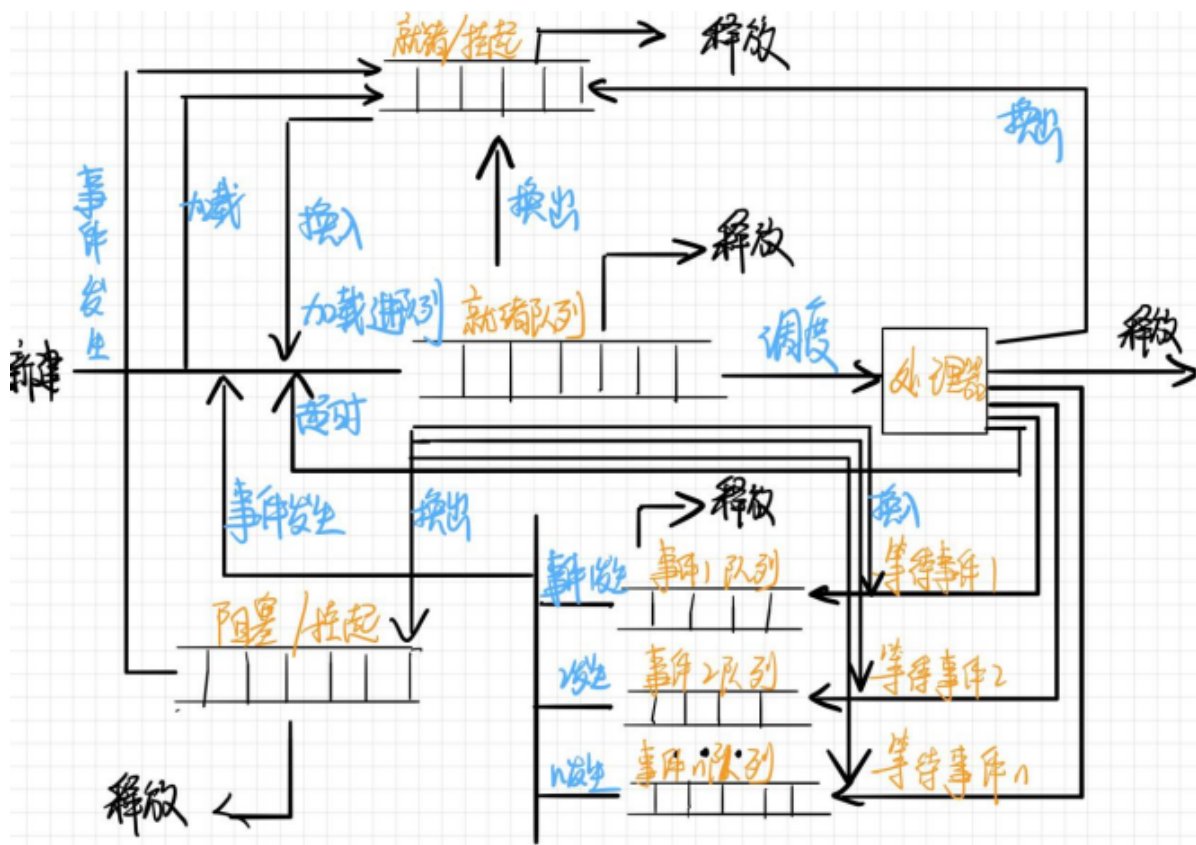
3.3

可能的转换	事件
新建态 to 就绪/挂起	无足够的内存空间分配给新进程
新建态 to 就绪态	操作系统准备好再接纳一个进程
就绪/挂起 to 就绪态	内存中无就绪进程；或处于就绪/挂起态的进程比处于就绪态的任何进程优先级高时
就绪态 to 就绪/挂起态	操作系统认为高优先级的阻塞态进程很快就会就绪，而此时的就绪态进程优先级低，则该就绪态进程可能会被挂起；或此时只能通过挂起就绪态进程来获得更多的内存空间，就绪态进程会被挂起成为就绪/挂起态
就绪态 to 运行态	需要选择一个新进程运行
运行态 to 就绪/挂起态	当一个运行程序进程的分配时间到期后，且此时阻塞/挂起队列中具有优先级更高的进程不再被阻塞，操作系统会抢占这个进程，或直接把运行进程转换到就绪/挂起队列中，释放内存空间。
运行态 to 就绪态	正在运行的进程已经达到“允许不中断执行”的最大时间段，或者被其他高优先级的进程抢占；或者自愿释放对处理器的控制。
各种状态 to 退出态	完成运行，或是出现了一些错误条件，或是被父进程终止，或是在父进程终止时终止
运行态 to 阻塞态	进程请求其必须等待某些事件。
阻塞/挂起态 to 就绪/挂起态	原处于阻塞/挂起的进程等待事件发生，则其有可能进入就绪/挂起态

可能的转换	事件
阻塞/挂起态 to 阻塞态	一个进程终止，一些内存空间被释放，而阻塞/挂起队列中有一个进程的优先级比就绪/挂起队列中所有进程的优先级都高，并且操作系统认为该进程事件会很快发生。
阻塞态 to 阻塞/挂起态	就绪进程为了维护基本的性能而需要更多的内存空间，会将一个阻塞态的程序挂起
阻塞态 to 就绪态	所等待的事件发生

不可能的转换	原因
阻塞/挂起态 to 新建态	进程已被操作系统接受，不可能在回到新建态的时候
阻塞/挂起态 to 就绪态	若阻塞/挂起态的进程等待事件发生且无内存空间分配给它时应进入就绪/挂起态，若有内存空间，则应先被加载到内存中来，变为阻塞态
阻塞/挂起态 to 运行态	若阻塞/挂起态的进程等待事件发生，要先想办法进入就绪态，才能运行，操作系统只会在就绪队列中选取进程来运行
阻塞态 to 就绪/挂起态	若阻塞态程序所等待的事件发生，则其会进入就绪态，若此时优先级低且内存空间已满，才会进入就绪/挂起态
阻塞态 to 新建态	进程已被操作系统接受，不可能在回到新建态的时候
阻塞态 to 运行态	若阻塞/挂起态的进程等待事件发生，要先进入就绪态才能被运行，操作系统只会在就绪队列中选取进程来运行
退出态 to 各种态	进程已经执行完毕，或出错而终止，被操作系统释放掉，不可能立即重新被操作系统接受

不可能的转换	原因
新建态 to 运行态	新的进程被操作系统接受之后先要被分配内存空间及资源，进入就绪态，才有可能进入其他状态，不能直接运行。
新建态 to 阻塞/挂起态	同上
新建态 to 阻塞态	同上
就绪/挂起 to 新建态	进程已被操作系统接受，不可能在回到新建态的时候
就绪/挂起 to 阻塞/挂起态	此转变无意义，内存空间状态没变；且处于就绪/挂起态的进程未被运行，也不知道是否要等待事件发生
就绪/挂起 to 阻塞态	处于就绪/挂起态的进程此时在内存中已经被释放掉了，没有被运行，它不可能出现需要等待事件的情况
就绪/挂起 to 运行态	操作系统应优先选择处于就绪队列中的进程来运行，即使处于就绪/挂起态的进程优先级更高，也应先变为就绪态才能被运行
就绪态 to 新建态	进程已被操作系统接受，不可能在回到新建态的时候
就绪态 to 阻塞/挂起态	处于就绪态的进程还没有被运行，它不知道自己有需要等待发生的事件，不会阻塞，更不会因为成为阻塞态后而占用过多内存而被挂起
就绪态 to 阻塞态	处于就绪态的进程还没有被运行，它不知道自己有需要等待发生的事件，不会阻塞并且挂起
运行态 to 新建态	进程已被操作系统接受，不可能在回到新建态的时候
运行态 to 阻塞/挂起态	处于运行态的程序当要等待某些事件发生时，应该转变为阻塞态，不会直接被移出内存



3.6答:

A. 将等待状态细分，每一个等待状态都分配有一个对应的队列，当影响某一等待进程的事件发生时，该进程进入相应队列，等到影响事件结束，恢复原进程时，可以快速定位到相应状态队列，减少时间的消耗。

B. 在这些状态下，允许程序被换出会降低电脑工作效率。例如，当发生页面错误等待时，进程等待换入一个页而使其可以执行，这时将进程换出毫无意义。

C.

当前状态:下一状态	当前正在执行	可计算 (驻留)	可计算 (换出)	各种等待状态 (驻留)	各种等待状态 (换出)
当前正在执行		重调度		等待	
可计算 (驻留)	调度		换出		
可计算 (换出)		换入			
各种等待状态 (驻留)		事件发生			换出
各种等待状态 (换出)			事件发生		

编程题

1. 调用fork () 函数

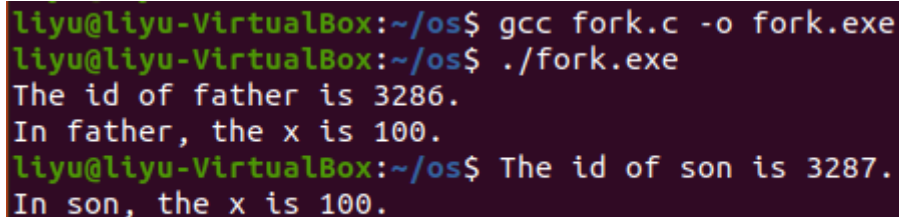
a. 父子进程共同访问同一变量

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>

int main(){
    int x = 100; //设置常量
    pid_t pid;
    pid = fork(); //调用fork () 函数

    if(pid == 0){ //调用fork () 函数之后，子进程返回值为0
        printf("The id of son is %d.\n", getpid());
        printf("In son, the x is %d.\n", x);
    }
    else{ //调用一次fork () 函数，会有两个返回值，一个为子进程，一个为父进程，且二者返回顺序随机
        printf("The id of father is %d.\n", getpid());
        printf("In father, the x is %d.\n", x);
    }
}
```

最后结果，对x的访问值是一样的。



```
liyu@liyu-VirtualBox:~/os$ gcc fork.c -o fork.exe
liyu@liyu-VirtualBox:~/os$ ./fork.exe
The id of father is 3286.
In father, the x is 100.
liyu@liyu-VirtualBox:~/os$ The id of son is 3287.
In son, the x is 100.
```

b. 在父子进程中修改x变量的值

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>

int main(){
    int x = 100;
    pid_t pid;
    pid = fork();

    if(pid == 0){ //调用fork () 函数之后，子进程返回值为0
        printf("The id of son is %d.\n", getpid());
        printf("In son, the x is %d.\n", x);
        printf("In son, the x+1 is %d.\n", x+1);
    }
}
```

```

else{//调用一次fork（）函数，会有两个返回值，一个为子进程，一个为父进程，且二者返回顺序随机
    printf("The id of father is %d.\n", getpid());
    printf("In father, the x is %d.\n", x);
    printf("In father, the x+1 is %d.\n", x+1);
}
}

```

最后结果是，在各自进程中修改的变量x互相独立，互不影响，x+1均为101.

```

The id of father is 3773.
In father, the x is 100.
In father, the x+1 is 101.
liyu@liyu-VirtualBox:~/os$ The id of son is 3774.
In son, the x is 100.
In son, the x+1 is 101.

```

2. open函数的调用

```

#include<string.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include<wait.h>
int main()
{
    int fd = open("./file.txt", O_RDWR);//以只写文件方式打开
    int pid = fork();//调用fork函数
    if(pid == 0){//子进程
        printf("Son is in file!\n");
        printf("Son's fd is %d\n", fd);
        write(fd, "This is son.\n", strlen("This is son.\n"));//若子进程可以继承文件描述符，则写入成功
    }else{
        printf("Father is in file!\n");
        printf("Father's fd is %d\n", fd);
        write(fd, "This is dad.\n", strlen("This is dad.\n"));
    }

    close(fd);
}

```

结果如图，子进程继承了父进程的文件描述符，父子进程均可使用同一fd进行文件读写

```

liyu@liyu-VirtualBox:~/os$ ./open.exe
Father is in file!
Father's fd is 3
Son is in file!
Son's fd is 3
liyu@liyu-VirtualBox:~/os$

```

字符串被写入文件

```
1 This is dad.
2 This is son.
```

3. 调用exec()所有变体

exec () 系列中的函数具有不同的行为:

l: 参数作为字符串列表传递给main ()

v: 参数作为字符串数组传递给main ()

p: 搜索新运行程序的路径e: 环境可以由调用方指定您可以将它们混合

因此具有:

```
int execl (const char * path, const char * arg, ...);
int execlp (const char * file, const char * arg, ...);
int execl_e (const char * path, const char * arg, ..., char * const envp []);
int execv (const char * path, char * const argv []);
int execvp (const char * file, char * const argv []);
int execve (const char * path, char * const argv [], char * const envp []);
```

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>

int main()
{
    char * const argv[] = {"ls", "-l", "-a", "-h", NULL};
    char * const envp[] = {NULL};

    /* 子进程调用exec函数 */
    if (fork() == 0)
    {
        printf("-----child1-----\n");
        execl("/bin/ls", "ls", "-l", "-a", "-h", NULL);
    }

    sleep(2);

    if(fork() == 0){
        printf("-----child2-----\n");
        execlp("ls", "ls", "-l", "-a", "-h", NULL);
    }

    if(fork() == 0){
        printf("-----child3-----\n");
        execl_e("/bin/ls", "ls", "-l", "-a", "-h", NULL, envp);
    }
}
```

```

}

sleep(2);

if(fork() == 0){
    printf("-----child4-----\n");
    execv("/bin/ls", argv);
}

sleep(2);

if(fork() == 0){
    printf("-----child5-----\n");
    execvp("ls", argv);
}

sleep(2);

if(fork() == 0){
    printf("-----child6-----\n");
    execvpe("ls", argv, envp);    // may not work
}

sleep(2);

printf("over!\n");
return 0;
}

```

结果如图，文件夹下所有文件名被罗列了6次

```

liyu@liyu-VirtualBox:~/os$ ./exec
-----child1-----
总用量 196K
drwxrwxr-x  2 liyu liyu 4.0K 4月  2 09:46 .
drwxr-xr-x 22 liyu liyu 4.0K 4月  2 09:34 ..
-rwxrwxr-x  1 liyu liyu 17K 4月  2 09:46 exec
-rw-rw-r--  1 liyu liyu 1020 4月  2 09:46 exec.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 23:51 execle
-rw-rw-r--  1 liyu liyu 487 3月 31 23:50 execle.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 23:38 execlp
-rw-rw-r--  1 liyu liyu 148 3月 31 23:38 execlp.c
-rw-rw-r--  1 liyu liyu 26 4月  1 00:28 file.txt
-rw-rw-r--  1 liyu liyu 664 3月 31 21:54 fork.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 21:54 fork.exe
-rwxrwxr-x  1 liyu liyu 17K 4月  1 00:28 open
-rw-rw-r--  1 liyu liyu 624 4月  1 00:28 open.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 23:23 open.exe
-rwxrwxr-x  1 liyu liyu 17K 4月  1 00:31 wait
-rw-rw-r--  1 liyu liyu 614 4月  1 00:31 wait.c
-rwxrwxr-x  1 liyu liyu 17K 4月  1 00:23 wait.exe

```



```

-----child2-----
总用量 196K
drwxrwxr-x 2 liyu liyu 4.0K 4月 2 09:46 .
drwxr-xr-x 22 liyu liyu 4.0K 4月 2 09:34 ..
-rwxrwxr-x 1 liyu liyu 17K 4月 2 09:46 exec
-rw-rw-r-- 1 liyu liyu 1020 4月 2 09:46 exec.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 23:51 execl
-rw-rw-r-- 1 liyu liyu 487 3月 31 23:50 execl.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 23:38 execlp
-rw-rw-r-- 1 liyu liyu 148 3月 31 23:38 execlp.c
-rw-rw-r-- 1 liyu liyu 26 4月 1 00:28 file.txt
-rw-rw-r-- 1 liyu liyu 664 3月 31 21:54 fork.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 21:54 fork.exe
-rwxrwxr-x 1 liyu liyu 17K 4月 1 00:28 open
-rw-rw-r-- 1 liyu liyu 624 4月 1 00:28 open.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 23:23 open.exe
-rwxrwxr-x 1 liyu liyu 17K 4月 1 00:31 wait
-rw-rw-r-- 1 liyu liyu 614 4月 1 00:31 wait.c
-rwxrwxr-x 1 liyu liyu 17K 4月 1 00:23 wait.exe

```

```

-----child3-----
total 196K
drwxrwxr-x 2 liyu liyu 4.0K Apr 2 09:46 .
drwxr-xr-x 22 liyu liyu 4.0K Apr 2 09:34 ..
-rwxrwxr-x 1 liyu liyu 17K Apr 2 09:46 exec
-rw-rw-r-- 1 liyu liyu 1020 Apr 2 09:46 exec.c
-rwxrwxr-x 1 liyu liyu 17K Mar 31 23:51 execl
-rw-rw-r-- 1 liyu liyu 487 Mar 31 23:50 execl.c
-rwxrwxr-x 1 liyu liyu 17K Mar 31 23:38 execlp
-rw-rw-r-- 1 liyu liyu 148 Mar 31 23:38 execlp.c
-rw-rw-r-- 1 liyu liyu 26 Apr 1 00:28 file.txt
-rw-rw-r-- 1 liyu liyu 664 Mar 31 21:54 fork.c
-rwxrwxr-x 1 liyu liyu 17K Mar 31 21:54 fork.exe
-rwxrwxr-x 1 liyu liyu 17K Apr 1 00:28 open
-rw-rw-r-- 1 liyu liyu 624 Apr 1 00:28 open.c
-rwxrwxr-x 1 liyu liyu 17K Mar 31 23:23 open.exe
-rwxrwxr-x 1 liyu liyu 17K Apr 1 00:31 wait
-rw-rw-r-- 1 liyu liyu 614 Apr 1 00:31 wait.c
-rwxrwxr-x 1 liyu liyu 17K Apr 1 00:23 wait.exe

```

```

-----child4-----
总用量 196K
drwxrwxr-x 2 liyu liyu 4.0K 4月 2 09:46 .
drwxr-xr-x 22 liyu liyu 4.0K 4月 2 09:34 ..
-rwxrwxr-x 1 liyu liyu 17K 4月 2 09:46 exec
-rw-rw-r-- 1 liyu liyu 1020 4月 2 09:46 exec.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 23:51 execl
-rw-rw-r-- 1 liyu liyu 487 3月 31 23:50 execl.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 23:38 execlp
-rw-rw-r-- 1 liyu liyu 148 3月 31 23:38 execlp.c
-rw-rw-r-- 1 liyu liyu 26 4月 1 00:28 file.txt
-rw-rw-r-- 1 liyu liyu 664 3月 31 21:54 fork.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 21:54 fork.exe
-rwxrwxr-x 1 liyu liyu 17K 4月 1 00:28 open
-rw-rw-r-- 1 liyu liyu 624 4月 1 00:28 open.c
-rwxrwxr-x 1 liyu liyu 17K 3月 31 23:23 open.exe
-rwxrwxr-x 1 liyu liyu 17K 4月 1 00:31 wait
-rw-rw-r-- 1 liyu liyu 614 4月 1 00:31 wait.c
-rwxrwxr-x 1 liyu liyu 17K 4月 1 00:23 wait.exe

```

```

-----child5-----
总用量 196K
drwxrwxr-x  2 liyu liyu 4.0K 4月  2 09:46 .
drwxr-xr-x 22 liyu liyu 4.0K 4月  2 09:34 ..
-rwxrwxr-x  1 liyu liyu 17K 4月  2 09:46 exec
-rw-rw-r--  1 liyu liyu 1020 4月  2 09:46 exec.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 23:51 execl
-rw-rw-r--  1 liyu liyu 487 3月 31 23:50 execl.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 23:38 execlp
-rw-rw-r--  1 liyu liyu 148 3月 31 23:38 execlp.c
-rw-rw-r--  1 liyu liyu 26 4月  1 00:28 file.txt
-rw-rw-r--  1 liyu liyu 664 3月 31 21:54 fork.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 21:54 fork.exe
-rwxrwxr-x  1 liyu liyu 17K 4月  1 00:28 open
-rw-rw-r--  1 liyu liyu 624 4月  1 00:28 open.c
-rwxrwxr-x  1 liyu liyu 17K 3月 31 23:23 open.exe
-rwxrwxr-x  1 liyu liyu 17K 4月  1 00:31 wait
-rw-rw-r--  1 liyu liyu 614 4月  1 00:31 wait.c
-rwxrwxr-x  1 liyu liyu 17K 4月  1 00:23 wait.exe
-----child6-----
total 196K
drwxrwxr-x  2 liyu liyu 4.0K Apr  2 09:46 .
drwxr-xr-x 22 liyu liyu 4.0K Apr  2 09:34 ..
-rwxrwxr-x  1 liyu liyu 17K Apr  2 09:46 exec
-rw-rw-r--  1 liyu liyu 1020 Apr  2 09:46 exec.c
-rwxrwxr-x  1 liyu liyu 17K Mar 31 23:51 execl
-rw-rw-r--  1 liyu liyu 487 Mar 31 23:50 execl.c
-rwxrwxr-x  1 liyu liyu 17K Mar 31 23:38 execlp
-rw-rw-r--  1 liyu liyu 148 Mar 31 23:38 execlp.c
-rw-rw-r--  1 liyu liyu 26 Apr  1 00:28 file.txt
-rw-rw-r--  1 liyu liyu 664 Mar 31 21:54 fork.c
-rwxrwxr-x  1 liyu liyu 17K Mar 31 21:54 fork.exe
-rwxrwxr-x  1 liyu liyu 17K Apr  1 00:28 open
-rw-rw-r--  1 liyu liyu 624 Apr  1 00:28 open.c
-rwxrwxr-x  1 liyu liyu 17K Mar 31 23:23 open.exe
-rwxrwxr-x  1 liyu liyu 17K Apr  1 00:31 wait
-rw-rw-r--  1 liyu liyu 614 Apr  1 00:31 wait.c
-rwxrwxr-x  1 liyu liyu 17K Apr  1 00:23 wait.exe
over!

```

4. wait () 函数

- 父进程中调用wait函数，返回值为子进程id

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<wait.h>

int main(){
    pid_t pid;
    pid = fork();

    if(pid == 0){//调用fork () 函数之后，子进程返回值为0
        printf("The id of son is %d.\n", getpid());
    }
    else{
        int son_id = wait(NULL);//父进程等待子进程执行
        printf("The id of father is %d.\n", getpid());
        printf("wait returns %d.\n", son_id);
    }
}

```

```
}  
}
```

执行结果如下

```
[Running] cd "/home/liyu/os/" && gcc wait.c -o wait && "/home/liyu/os/"wait  
The id of son is 16616.  
The id of father is 16615.  
Wait reterns 16616.  
  
[Done] exited with code=0 in 0.367 seconds
```

b. 子进程中调用wait(),返回-1

```
#include<stdio.h>  
#include<stdlib.h>  
#include<unistd.h>  
#include<sys/types.h>  
#include<wait.h>  
  
int main(){  
    pid_t pid;  
    pid = fork();  
    int res;  
  
    if(pid == 0){//调用fork () 函数之后，子进程返回值为0  
        if((res = wait(NULL) )== -1){//子进程中调用wait()  
            printf("wait reterns %d.\n", res);  
        }  
        printf("The id of son is %d.\n", getpid());  
    }  
    else{//调用一次fork () 函数，会有两个返回值，一个为子进程，一个为父进程，且二者返回顺序随机  
  
        printf("The id of father is %d.\n", getpid());  
    }  
}
```

执行结果如下

```
[Running] cd "/home/liyu/os/" && gcc wait.c -o wait && "/home/liyu/os/"wait  
The id of father is 16694.  
Wait reterns -1.  
The id of son is 16695.  
  
[Done] exited with code=0 in 0.59 seconds
```