Week4 Report

姓名：Yitong WANG(王奕童) 11910104@mail.sustech.edu.cn

学号：11910104

实验课时段：周五5-6节

实验课教师：Yun SHEN(沈昀) sheny@mail.sustech.edu.cn

实验课SA:

- Yining TANG(汤怡宁) 11811237@mail.sustech.edu.cn
- Yushan WANG(王宇杉) 11813002@mail.sustech.edu.cn

Q1 父子进程区分 & 执行顺序

Q1-1 父子进程区分

考虑使用 `man fork` 指令查看相关文档说明，并找到下图中的说明：

注意红色框中的话：如果父进程使用 `fork` ，则返回数值为子进程的 `pid` ；如果子进程使用 `fork` ，则返回数值为0。

因此父子进程的区分方法是根据 `fork` 方法的返回值。

Q1-2 执行顺序

考虑课件中的代码：

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]){
  printf("A\n");
  fork();
  printf("B\n");
  fork();
  printf("C\n");
  return 0;
}
```

进行多次运行，发现其中的运行结果会不一致：

因此可知父子进程的执行顺序是不固定的。

Q2 waitpid()函数原型，参数含义，与具体功能

Q2-1 函数原型

考虑使用 man waitpid 指令查看相关文档说明，并找到下图中的说明：

```
WAIT(2)                  Linux Programmer's Manual                  WAIT(2)

NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *wstatus);

       pid_t waitpid(pid_t pid, int *wstatus, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                        /* This is the glibc and POSIX interface; see
                           NOTES for information on the raw system call. */

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       waitid():
           Since glibc 2.26: _XOPEN_SOURCE >= 500 ||
               _POSIX_C_SOURCE >= 200809L
           Glibc 2.25 and earlier:
               _XOPEN_SOURCE
                   || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
                   || /* Glibc versions <= 2.19: */ _BSD_SOURCE

DESCRIPTION
       All  of  these  system  calls  are used to wait for state changes in a
       child of the calling process, and obtain information about  the  child
       whose  state  has  changed.   A  state change is considered to be: the
       child terminated; the child was stopped by a signal; or the child  was
       resumed  by a signal.  In the case of a terminated child, performing a
       wait allows the system to release the resources  associated  with  the
       child;  if  a wait is not performed, then the terminated child remains
       in a "zombie" state (see NOTES below).

       If a child has already changed state, then these calls return  immedi-
       ately.   Otherwise, they block until either a child changes state or a
       signal handler interrupts the call (assuming that system calls are not
       automatically  restarted  using  the SA_RESTART flag of sigaction(2)).
       In the remainder of this page, a child whose  state  has  changed  and
Manual page waitpid(2) line 1/413 11% (press h for help or q to quit)
```

课件waitpid的函数原型为：

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Q2-2 参数含义与具体作用

在文档说明中找到对应参数:

```
wait() and waitpid()
    The wait() system call suspends execution of the calling thread  until
    one of its children terminates.  The call wait(&wstatus) is equivalent
    to:

        waitpid(-1, &wstatus, 0);

    The waitpid() system call suspends execution of the calling thread un-
    til  a child specified by pid argument has changed state.  By default,
    waitpid() waits only for terminated children, but  this  behavior  is
    modifiable via the options argument, as described below.

    The value of pid can be:

    < -1   meaning  wait  for  any child process whose process group ID is
           equal to the absolute value of pid.

    -1     meaning wait for any child process.

    0      meaning wait for any child process whose process  group  ID  is
           equal to that of the calling process at the time of the call to
           waitpid().

    > 0    meaning wait for the child whose process ID  is  equal  to  the
           value of pid.

    The  value  of  options is an OR of zero or more of the following con-
    stants:
```

- 参数pid
  - 表示进程的等待状态
  - >0: 只等待进程id等同于 pid 的子进程
  - -1: 等待所有子进程退出,效果等同于 wait
  - =0: 等待同一进程组的任何子进程
  - <-1: 等待一个指定进程组的所有子进程,其中进程组的id为pid的绝对值。
- 参数wstatus
  - 表示传出的参数,如果设为NULL则表示无需传出该参数
  - 如果不是NULL,则会将进程的状态返回值放入wstatus指向的整数中。
- 参数options
  - 表示来控制 waitpid 的提供额外选项
  - WNOHANG:非阻塞的,立即返回,如果没有发现已退出的子进程可收集,则返回0

Q3 僵尸进程情况4

情况4：父进程不执行wait()，父进程比子进程先结束

这种情况即为子进程没有结束，但是父进程先结束，形成了"孤儿进程"。子进程会重新分配一个父进程，这个父进程有可能是init进程，也有可能是已注册的祖父进程。

下列代码可以验证：

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

int main(void){
        int pid = fork();
        if(pid == 0){
                sleep(10);
                printf("This is a child. Child pid = %d, ppid = %d\n", getpid(), getppid());
        }else {
                printf("This is a parent. Child = %d, pid = %d, ppid = %d\n",pid , getpid(), get
                printf("After child ends.\n");
        }
        printf("Which process?\n");

        return 0;
}
```
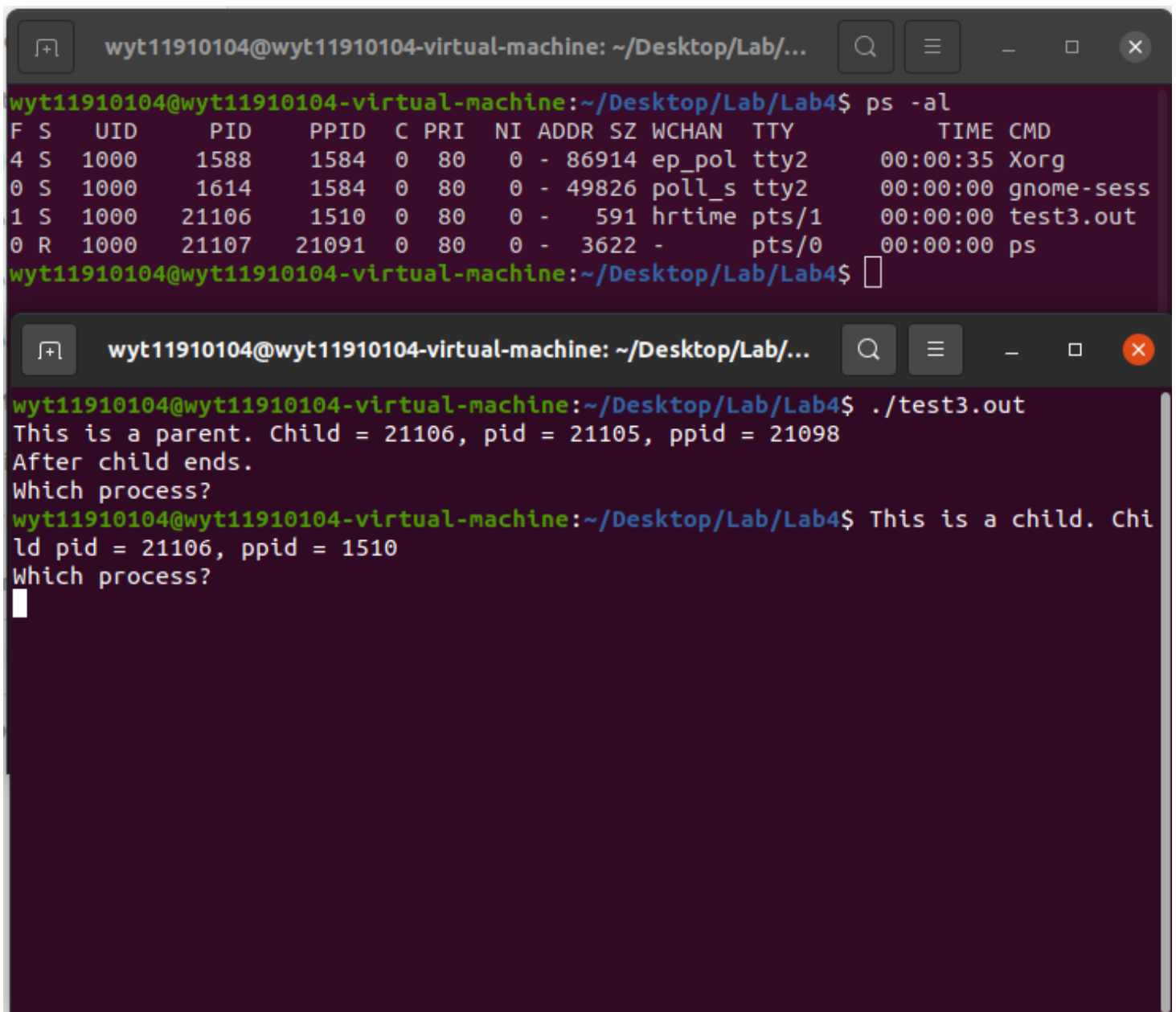
这里由进程号为1510的命令行启动进程号为21105的父进程，然后父进程再生成进程号为21106的子进程。可以看到父子进程均未wait，且父进程优先于子进程结束。从以下的运行结果来看，子进程的ppid就被分配为命令行进程的pid————1510。

```
wyt11910104@wyt11910104-virtual-machine: ~/Desktop/Lab/...

wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ ps -al
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   1000   1588   1584  0  80   0 - 86914 ep_pol tty2     00:00:35 Xorg
0 S   1000   1614   1584  0  80   0 - 49826 poll_s tty2     00:00:00 gnome-sess
1 S   1000  21106   1510  0  80   0 -   591 hrtime pts/1    00:00:00 test3.out
0 R   1000  21107  21091  0  80   0 -  3622 -      pts/0    00:00:00 ps
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ 
```

```
wyt11910104@wyt11910104-virtual-machine: ~/Desktop/Lab/...

wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ ./test3.out
This is a parent. Child = 21106, pid = 21105, ppid = 21098
After child ends.
Which process?
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ This is a child. Chi
ld pid = 21106, ppid = 1510
Which process?
```

Q4 僵尸进程代码实现与状态截图

考虑以下C语言代码的僵尸进程实现：

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/types.h>
 4
 5 int main(){
 6
 7         pid_t pid = fork();
 8
 9         if(pid == 0){
10                 printf("Child! pid = %d\n", getpid());
11
12                 sleep(5);
13
14                 printf("Child has dead!\n");
15         }else {
16                 printf("Parent! pid = %d\n", getpid());
17
18                 sleep(25);
19
20                 printf("Parent has dead!\n");
21         }
22
23         return 0;
24 }
```

（代码参考：https://blog.csdn.net/lvxin15353715790/article/details/89852259 ）

这段代码首先fork以创建子进程，然后子进程只休眠5秒，而父进程休眠25秒，因此子进程先结束，产生了僵尸进程的状态。

- 截图1：运行前

- 截图2：父子进程同时运行

```
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ ps -al
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   1000   1588   1584  0  80   0 - 86914 ep_pol tty2     00:00:44 Xorg
0 S   1000   1614   1584  0  80   0 - 49826 poll_s tty2     00:00:00 gnome-sess
0 S   1000  21434  21372  0  80   0 -   624 hrtime pts/1    00:00:00 zombie.out
1 S   1000  21435  21434  0  80   0 -   624 hrtime pts/1    00:00:00 zombie.out
0 R   1000  21436  21249  0  80   0 -  3622 -      pts/0    00:00:00 ps
```

- 截图3：子进程已成僵尸态，父进程仍然运行

```
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ ps -al
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   1000   1588   1584  0  80   0 - 86914 ep_pol tty2     00:00:44 Xorg
0 S   1000   1614   1584  0  80   0 - 49826 poll_s tty2     00:00:00 gnome-sess
0 S   1000  21434  21372  0  80   0 -   624 hrtime pts/1    00:00:00 zombie.out
1 Z   1000  21435  21434  0  80   0 -     0 -      pts/1    00:00:00
0 R   1000  21438  21249  0  80   0 -  3622 -      pts/0    00:00:00 ps
```

- 截图4：父子进程都退出

```
wyt11910104@wyt11910104-virtual-machine:~/Desktop/Lab/Lab4$ ps -al
F S   UID    PID   PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   1000   1588   1584  0  80   0 - 87786 ep_pol tty2     00:00:45 Xorg
0 S   1000   1614   1584  0  80   0 - 49826 poll_s tty2     00:00:00 gnome-sess
0 R   1000  21453  21249  0  80   0 -  3622 -      pts/0    00:00:00 ps
```

Q5 子进程exec实现"ps -al"

考虑以下代码以实现需求功能：

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4
5 int main(){
6
7         pid_t pid = fork();
8
9         if(pid == 0){
10                printf("Child! pid = %d\n", getpid());
11
12                execl("/bin/ps","/bin/ps", "al", NULL);
13
14                printf("Child has dead!\n");
15        }else {
16                printf("Parent! pid = %d\n", getpid());
17
18                printf("Parent is waiting ...\n");
19
20                wait(NULL);
21
22                printf("Parent has dead!\n");
23        }
24
25        return 0;
26 }
```

运行结果如下:

可以看到父进程产生了子进程，并且运行了"ps -al"，父进程再等待子进程结束后才运行输出 `Parent has dead!` 。

Q6 pipe中父子进程的功能

1. 父进程创建管道 `int pipe(int pipe_fd[2])` ，其中 `pipefd[2]` 为两个文件描述符， `fd[0]` 对应读， `fd[1]` 对应写。（实验课课件内容）
2. 父进程创建子进程，父子进程共享文件描述符（即同一管道）（实验课课件内容）
3. 这段代码的运行流程是：

- 首先尝试fork创建子进程
- 根据创建的进程和接受的信号执行相应的方法：
    - SIGALRM: 子进程接收到就执行write_data，父进程接收到就执行read_data；
    - SIGINT：子进程接受到就执行finish_write，父进程接收到就执行finish_read；
- 父子进程通讯：
    - 子进程在write_data中向父进程发送SIGALRM信号
    - 父进程在read_data中向子进程发送SIGALRM信号
    - 子进程自己会通知自己SIGALRM信号
- Ctrl + C则会结束代码的运行流程