

Week4 Process

一、实验概述

学习使用fork(), getpid(), exec(), wait(), signal(), kill()等调用, 了解zombie进程, 学习pipe,FIFO的使用。

二、实验目的

1. 学习创建进程
2. 学习父子进程
3. 学习简单的进程通信

三、实验内容

1. fork()
2. getpid()
3. wait()
4. Zombie
5. exec()
6. kill(), signal()
7. Pipe, FIFO

四、实验流程及相关知识点

第一步. 通过fork()创建进程

调用fork()函数, 可以为当前进程创建一个“副本”子进程。"副本"子进程几乎完全复制父进程的所有信息。

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int i = 0;
    fork(); //创建子进程
    i++;
    printf("%d\n", i);
    while(1);
    return 0;
}
```

父子进程的PID是不同的, 一般情况下PID小的是父进程。

The screenshot shows a terminal window with the following content:

```

sy@sy-OSlab:~/Desktop/lab$ gcc -o fork fork.c
sy@sy-OSlab:~/Desktop/lab$ ./fork
1
1

```

Below the terminal output is a window titled "sy@sy-OSlab: ~/Desktop/lab" containing a terminal with the command `ps -al` and its output:

```

sy@sy-OSlab:~/Desktop/lab$ ps -al
F S      UID        PID      PPID    C  PRI   NI     ADDR  SZ  WCHAN    TTY          TIME CMD
4 S      1000        1574        1572    0   80    0  - 75072  ep_pol  tty2         00:00:20 Xorg
0 S      1000        1609        1572    0   80    0  - 49824  poll s  ttv2         00:00:00 gnome-sess
0 R      1000        4690        3770   99   80    0  -   624  -      pts/0         00:00:03 fork
1 R      1000        4691        4690   99   80    0  -   624  -      pts/0         00:00:03 fork
0 R      1000        4692        4652    0   80    0  -   5013  -      pts/1         00:00:00 ps

```

Below the terminal output is a process tree diagram:

```

└─{gnome-shell-cat}(1799)
   └─gnome-terminal-(3758)
      └─bash(3770)
         └─fork(4690)
            └─fork(4691)
               └─bash(4652)
                  └─pstree(4720)
                     └─{gnome-terminal-}(3759)

```

请尝试以下代码，猜测可能的运行结果。

```
#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;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ ./sample1
A
B
B
C
C
C
C
sy@sy-OSlab:~/Desktop/lab$ ./sample1
A
B
C
B
sy@sy-OSlab:~/Desktop/lab$ C
C
C
./sample1
A
B
B
C
C
C
C
```

观察以上结果可以发现，父子进程的执行顺序是不确定的。

那么在代码中如何区分父子进程呢？

请尝试"man fork"指令。（Tips: 如果遇到困难，查手册）

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and `errno` is set appropriately.

通过阅读手册，我们得知可以通过fork()的返回值判断“当前”两个进程哪个是父进程，哪个是子进程。

```
#include<stdio.h>
#include<unistd.h>
int main(){
    if(fork()){
        printf("I'm parent process.\n");
    }else{
        printf("I'm child process.\n");
    }
    return 0;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ gcc -o fork fork.c
sy@sy-OSlab:~/Desktop/lab$ ./fork
I'm parent process.
I'm child process.
```

第二步. 通过getpid()获得进程号

除了区分父子进程，我们经常还需要获得进程的信息，如自己的进程号、父进程的进程号，比较常用的一个函数是getpid()，该函数可以获得自己的进程号。

```
int main(){
    if(fork()){
        printf("I'm parent process %d.\n",getpid());
    }else{
        printf("I'm child process %d.\n",getpid());
    }
    return 0;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ gcc -o fork fork.c
sy@sy-OSlab:~/Desktop/lab$ ./fork
I'm parent process 5764.
I'm child process 5765.
```

请尝试通过getppid()获得父进程的pid。

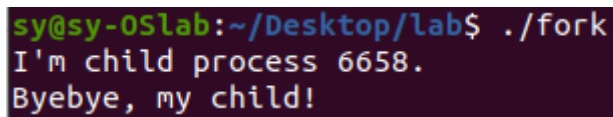
第三步. 学习使用wait()

前面实验的过程中，可以发现父子进程之间是没有固定的执行顺序的，如果我们想要它们以固定的顺序执行，可以使用wait()。

wait()的原型是 `pid_t wait(int *wstatus)`，当wait()被执行时，当前进程会被挂起(suspend)，直到子进程结束，子进程结束后会给父进程发送 `SIGCHLD` 中断，wait()收到中断后会结束阻塞并使系统回收子进程的相关资源。

因此，我们可以通过在父进程中调用wait()来使子进程优先执行。

```
int main(){
    if(fork()){
        wait(NULL);
        printf("Byebye, my child!\n");
    }else{
        printf("I'm child process %d.\n",getpid());
    }
    return 0;
}
```



```
sy@sy-05lab:~/Desktop/lab$ ./fork
I'm child process 6658.
Byebye, my child!
```

注意参数 `int *wstatus` 并非指定某个子进程，而是当子进程结束时用于存储子进程的结束状态值。

如果我们不需要结束状态值，则可以将参数指定为NULL，即 `wait(NULL)`。

那么现在请回答一个问题：

如果一个父进程有多个子进程，它等待的是哪个子进程？是任意一个子进程还是某个子进程还是全部子进程？

请通过 `man wait` 自己找到答案。

另一个常用的wait指令是waitpid()，也请通过 `man wait` 学习如何使用。

第四步. Zombie

前面一步我们学习了wait()的使用，wait()除了可以使子进程优先执行外，还有一个很重要的作用：**回收子进程的相关资源**（比如pid）。这个功能是非常重要的，比如系统的pid是有限的，如果长期有大量进程pid未被回收，可能导致pid被用完从而无法分配pid给新的进程。

关于wait()的执行，这里可以排列组合出几种情况：

1. 父进程执行wait()，父进程比子进程后结束
2. 父进程执行wait()，父进程比子进程先结束（由于wait()的执行，这种情况并不会出现）
3. 父进程不执行wait()，父进程比子进程后结束
4. 父进程不执行wait()，父进程比子进程先结束

情况1，子进程结束后，父进程会通过wait回收子进程的相关资源。

情况3则比较复杂了，会出现一种情况：**子进程虽然已经结束了，但是父进程未结束并且未回收尸体（资源），子进程变成了僵尸（Zombie process）**。那么这样的僵尸进程怎样处理呢？

要解决僵尸进程，我们需要先解决另一个问题：**如果子进程没有结束但是父进程结束了（孤儿进程），那么子进程会变成无父之子吗？** 答案是不会，子进程会被过继给init进程或者注册过的祖父进程（孤儿院）以确保每个进程都一定有父进程。

回到僵尸进程的问题，如果因为父进程不回收而导致出现僵尸进程，那么在父进程结束后，僵尸进程会被过继给“继父”，新的父进程会自动回收僵尸进程。

现在，请尝试自己回答情况4的结果会是什么？

第五步. 学习使用exec()

实际编写程序时，我们经常需要fork一个新的进程，但是我们大部分时候并不需要fork一个新的“自己”，因此如何fork一个和原进程不同的进程是很重要的。但我们要创建新的进程只能通过fork()，所以我们需要通过exec()把fork()出的副本进行“重写”以让新的进程执行与父进程不同的功能。

```
int main(){
    printf("before execl ...\n");
    execl("/bin/ls", "/bin/ls", NULL);
    printf("after execl ...\n");
    return 0;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ ./exec
before execl ...
exec execl.c fork fork.c sample1 sample1.c
```

需要特别注意exec()一旦执行，原本的进程将不复存在，进程的代码段全部被替换，并且会分配新的资源，因而原进程exec()之后的代码不会再被执行（即使exec()执行结束）。

exec家族有很多不同参数列表的兄弟姐妹都可以实现exec的功能，请自己 `man` 一下。

第六步. 学习使用kill(), signal()

第一周我们学习过通过kill -9来强制终止进程。这里的9即9号中断信号SIGKILL，该信号用于强制结束进程，并且这个信号不能被阻塞、处理和忽略。可以通过 `kill -l` 查看中断号对应的中断信号。

```
sy@sy-OSlab:~/Desktop/lab$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO      30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

而本周的学习中我们提到子进程结束时发送SIGCHLD中断给父进程，这个信号是第17号信号，SIGCHLD是允许被忽略的。（可以通过 `man wait` 查看“如果父进程忽略这个信号会出现什么情况”，看不懂请问老师）

现在，我们学习下如何发送、处理、忽略中断信号。

发送中断信号：

代码中可以通过 `int kill(pid_t pid, int sig)` 发送中断信号，其中pid代表要接收进程的进程号，sig代表要发送的中断信号的中断号。

处理中断信号：

signal()虽然长得非常像发送信号的方法，但我们已经有kill()了，所以signal()并不是用来发送信号的。并且signal()也不是用来获取信号的。

signal()实际上是用来注册信号的处理方式的，`sighandler_t signal(int signum, sighandler_t handler)`；中signum代表中断号，handler则是接到signum对应的中断信号后的响应方式，handler的值可以是SIG_IGN，SIG_DFL 或者一个函数。当handler是一个函数时，一旦进程接收到signum对应的中断信号，则handler会代替默认的中断处理方式进行处理。

忽略中断信号：

通过 `signal(signum, SIG_IGN)`；可以忽略signum对应的中断信号。请注意 SIGKILL 和 SIGSTOP 不能被忽略。

请阅读并运行以下代码

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void child_ding(){
    printf("child ding~~~~~\n");
}

void ctrlc_ding(){
    printf("ctrl+c ding~~~~~\n");
}

int main(){
    if(fork()){
        signal(SIGALRM, child_ding);
        signal(SIGINT, ctrlc_ding);
        signal(SIGCHLD, SIG_IGN);
        while(1){}
    }else{
        while(1){
            sleep(2);
            kill(getppid(),SIGALRM);
        }
    }
}
```

观察运行结果，尝试点击ctrl+c并观察结果。（还可以尝试用另一个terminal通过kill发送中断）

第七步. 学习Pipe, FIFO的使用

Pipe, 管道

管道是通过把两个进程之间的标准输入输出连接起来进行通信的工具。通过管道我们可以进行进程间的通信。

管道的使用过程：

1. 父进程创建管道 `int pipe(int pipefd[2])`，其中pipefd[2]为两个文件描述符，fd[0]对应读，fd[1]对应写。
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）
3. 父进程从写端写入数据，子进程从读端读取数据。反之亦然。

请运行并理解以下代码：

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

int pid;
int pipe_fd[2];
char buff[1024], input[1024];

void write_data(){
    sleep(1);
    printf("\nMy pid is %d\n", getpid());
    printf("Please input a string:\n");
    scanf("%s", input);
    write(pipe_fd[1], input, strlen(input)); //写端写入
    printf("Write finished\n");
    kill(getppid(), SIGALRM);
}

void finish_write(){
    close(pipe_fd[1]);
    printf("%d finish write\n", getpid());
    exit(0);
}

void read_data(){
    sleep(1);
    printf("\nMy pid is %d\n", getpid());
    printf("read begins\n");
    memset(buff, 0, sizeof(buff));
    read(pipe_fd[0], buff, 1024); //读端读取
    printf("Read finished\n");
    printf("Message is: %s\n", buff);
    kill(pid, SIGALRM);
}

void finish_read(){
    close(pipe_fd[0]);
    printf("%d finish read\n", getpid());
    exit(0);
}

int main(){

    if (pipe(pipe_fd) < 0){
        printf("pipe create failed\n");
    }

    if ((pid = fork()) < 0){
        printf("fork failed\n");
    }

    if (!pid){
```

```

        printf("child process begins, pid = %d\n", getpid());
        signal(SIGALRM, write_data);
        signal(SIGINT, finish_write);
        kill(getpid(), SIGALRM);
        while (1){
        }
    }
    else{
        printf("parent process begins, pid = %d\n", getpid());
        signal(SIGALRM, read_data);
        signal(SIGINT, finish_read);
        while (1){
        }
    }
}
}

```

FIFO, 有名管道

有名管道的使用过程：

1. 在terminal中通过命令 `mkfifo 管道名` 创建有名管道，有名管道实际是一个文件，可以通过 `file 管道名` 来查看这个文件的具体信息
2. 通过 `echo 内容 > 管道名` 来写入内容（写入后当前terminal会阻塞）
3. 在另一个terminal通过 `cat 管道名` 来读取内容

```

sy@sy-OSlab:~/Desktop/lab$ ls
Ding Ding.c exec exec.c fork fork.c sample1 sample1.c
sy@sy-OSlab:~/Desktop/lab$ mkfifo myfifo
sy@sy-OSlab:~/Desktop/lab$ ls
Ding Ding.c exec exec.c fork fork.c myfifo sample1 sample1.c
sy@sy-OSlab:~/Desktop/lab$ file myfifo
myfifo: fifo (named pipe)
sy@sy-OSlab:~/Desktop/lab$ echo hello > myfifo
sy@sy-OSlab:~/Desktop/lab$ 

```

The screenshot shows a terminal window with the following commands and output:

```

sy@sy-OSlab:~/Desktop/lab$ ls
Ding Ding.c exec exec.c fork fork.c sample1 sample1.c
sy@sy-OSlab:~/Desktop/lab$ mkfifo myfifo
sy@sy-OSlab:~/Desktop/lab$ ls
Ding Ding.c exec exec.c fork fork.c myfifo sample1 sample1.c
sy@sy-OSlab:~/Desktop/lab$ file myfifo
myfifo: fifo (named pipe)
sy@sy-OSlab:~/Desktop/lab$ echo hello > myfifo
sy@sy-OSlab:~/Desktop/lab$ 

```

The terminal window title is "sy@sy-OSlab: ~/Desktop/lab". The output of the `cat myfifo` command is "hello".

Pipe vs FIFO

PIPE	FIFO
PIPE is un-named IPC object	FIFO is named IPC object
PIPE doesn't exist in the filesystem, vanish after program exit or one of end closed.	FIFO exists in the filesystem. Even the program exit, FIFO file remain till system reboot.
PIPE can only communicate among the related process (parent and child, sibling).	FIFO can be used for unrelated process. Even not in a local computer (in network).
PIPE is often used between two processes.	FIFO is often used in multiple processes communicating.
PIPE no control over ownership and permission.	FIFO, as a file, need to control ownership and permission.

六、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 如何创建进程
2. 父子进程的区别
3. 如何回收子进程的资源
4. 如何使用exec家族
5. 发送、处理、忽略中断信号
6. 如何使用管道和有名管道

七、下一实验简单介绍

在下一实验实验中，我们将在ucore上进行中断处理。