操作系统实验报告

实验七：进程与通信——扩展 PCB 的结构实现进程控制原语

学院：　　数据科学与计算机学院

班级：　　16级计算机科学与技术 教务2班

姓名：　　郑映雪

学号：　　16337327

完成时间：　　2018.6.12

一、实验目的

1. 完善实验 6 中的二状态进程模型，实现五状态进程模型，从而使进程可以分工合作，并发运行。

2. 了解派生进程、结束进程、阻塞进程等过程中父、子进程之间的关系和分别进行的操作。

3. 理解原语的概念并实现进程控制原语 do_fork(), do_exit(), do_wait(), wakeup, blocked。

二、实验要求

在实验五或更后的原型基础上，进化你的原型操作系统，原型保留原有特征的基础上，设计满足下列要求的新原型操作系统：

(1)实现控制的基本原语 do_fork()、 do_wait()、do_exit()、blocked()和 wakeup()。

(2)内核实现三系统调用 fork()、 wait()和 exit() ，并在 c 库中封装相关的系统调用.

(3)编写一个 c 语言程序，实现多进程合作的应用程序。

多进程合作的应用程序可以在下面的基础上完成：由父进程生成一个字符串，交给子进程统计其中字母的个数，然后在父进程中输出这一统计结果。

参考程序如下：

```
char str[80]=”129djwqhdsajd128dw9i39ie93i8494urjoiew98kdkd”;
int LetterNr=0;
void main() {
int pid;
char ch;
pid=fork();
if (pid==-1) printf(“error in fork!”);
if (pid)  {   ch=wait(); printf(“LetterNr=”); ntos(LetterNr); }
Else {   CountLetter(str);   exit(0);}
}
```

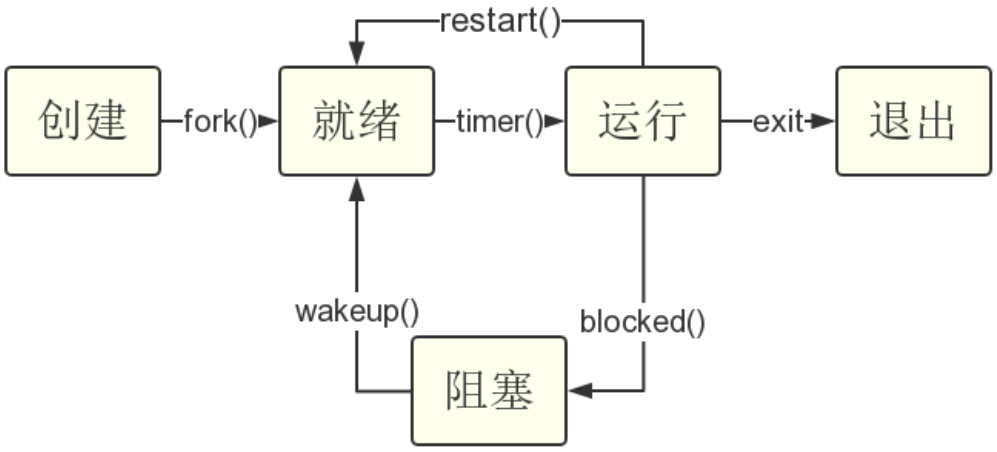
编译连接你编写的用户程序，产生一个 com 文件，放进程原型操作系统映像盘中。

三、实验方案

1、相关原理

a.五状态模型

在上一个实验中，我们实现的模型是二状态模型，且多个进程之间没有直接的合作。在本次实验中，通过对进程原语的编写，我们可以实现五状态模型。如下图所示：



在本次实验中，老师给的课件里有说明，进程需要按需产生子进程，一组进程分工合作，并发运行，各自完成一定的工作。在并发的同时，需要协调一些事件的时序，实现简单的同步。

b.do_fork 的原理

由五状态模型图可知，fork 操作可以创建进程。进程的创建主要工作是完成一个进程映像的构造。在本次实验中，我们实现父子进程共享代码段和全局数据段，子进程的 CS 和 IP 继承自父进程。在老师的课件中，基本把 fork 操作讲述得很详细了，fork 调用功能如下：

①寻找一个自由的 PCB 块，如果没有，创建失败，调用返回－1；

②以当前进程为父进程，复制父进程的 PCB 内容到自由 PCB 中；

③产生一个唯一的 ID 作为子进程的 ID，存入至 PCB 的相应项中；

④为子进程分配新栈区，从父进程的栈区中复制整个栈的内容到子进程的栈区中；

⑤调整子进程的栈段和栈指针，子进程的父亲指针指向父进程；

⑥在父进程的调用返回 ax 中送子进程的 ID，子进程调用返回 ax 送 0。

c.wait 的原理

当 fork 建立好父子进程后，父进程如果想在子进程执行完毕后处理接下来的事情，就需要一个 wait 实现同步。此时内核的进程增加一种阻塞态，当进程调用 wait 系统调用时，内核将当前进程阻塞，并调用进程调度过程挑选另一个就绪进程接权。这点与实验六相似，故此处不详细叙述了。

d.exit 的原理

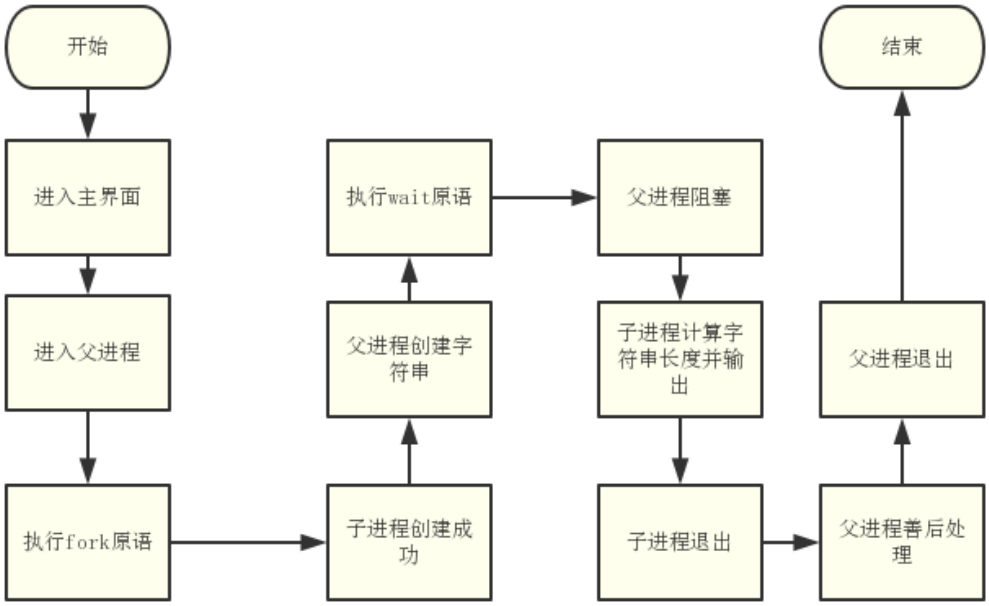
父进程调用 wait 后被阻塞。而子进程终止时调用 exit()，向父进程报告这一事件，可以传递一个字节的信息给父进程，并解除父进程的阻塞，该子进程占用的所有资源被收回，系统中减少了一个进程，并调用进程调度过程挑选另一个就绪进程接权。

e.本次实验中的做法

老师提供了一个用户程序，是用来统计字符串的长度的。具体需要实现：由父进程生成字符串，子进程统计字母个数，父进程输出统计结果。实现这一目的的方法就是将以上的原语封装进系统调用中，并在用户程序中进行系统调用实现 fork、wait 和 exit，当然，这是已经做过的实验四&五的事，所以并不困难。

2、程序流程图

在本次实验中，我删减了部分流程，只留下与本实验相关的操作。在本实验中，流程图如下：



3、程序关键模块

a. do_fork 模块

fork 模块中，我们创建了子进程，并将子程序的 id 送入 ax。随后子进程继承父进程的栈结构，并将父亲指针指向父进程。代码如下：

```
int do_fork() {
	int sub_ID;
	print("Do forking...\r\n");
	sub_ID = createSubPCB();
	if (sub_ID == -1) {
		current_process_PCB_ptr->regs.ax = -1;
		return -1;
	}
	sub_PCB = &PCB_LIST[sub_ID];
	current_process_PCB_ptr->regs.ax = sub_ID;
	sub_ss = sub_PCB->regs.ss;
	f_ss = current_process_PCB_ptr->regs.ss;
	stack_size = 0x100;
	stackCopy();
	PCB_Restore();
}
```

b.wait 模块和 block 模块。

子进程在开始执行前，需要将父进程阻塞，此时就需要 wait 模块调用 block 模块来进行这个操作。这个操作比较简单，只要把进程状态设为 BLOCKED 即可。代码如下：

```
void do_wait() {
	print("Do waiting...\r\n");
	blocked();
}

void blocked() {
	current_process_PCB_ptr->status = PCB_BLOCKED;
	schedule();
	PCB_Restore();
}
```

c.exit 模块和 wakeup 模块

进程结束后，进程占有的资源回收，系统中进程数量减。重新调度，父进程处理后事。代码如下：

```
void do_exit(int ss) {
	print("exiting succeed!\r\n");
	PCB_LIST[current_process_number].status = PCB_EXIT;
	PCB_LIST[current_process_PCB_ptr->FID].status = PCB_READY;
	PCB_LIST[current_process_PCB_ptr->FID].regs.ax = ss;
	current_seg -= 0x1000;
	process_number--;
	wakeup();
	print(">>");
}

void wakeup() {
	if (process_number == 1)
		kernal_mode = 1;
	schedule();
	PCB_Restore();
}
```

d.封装进系统调用

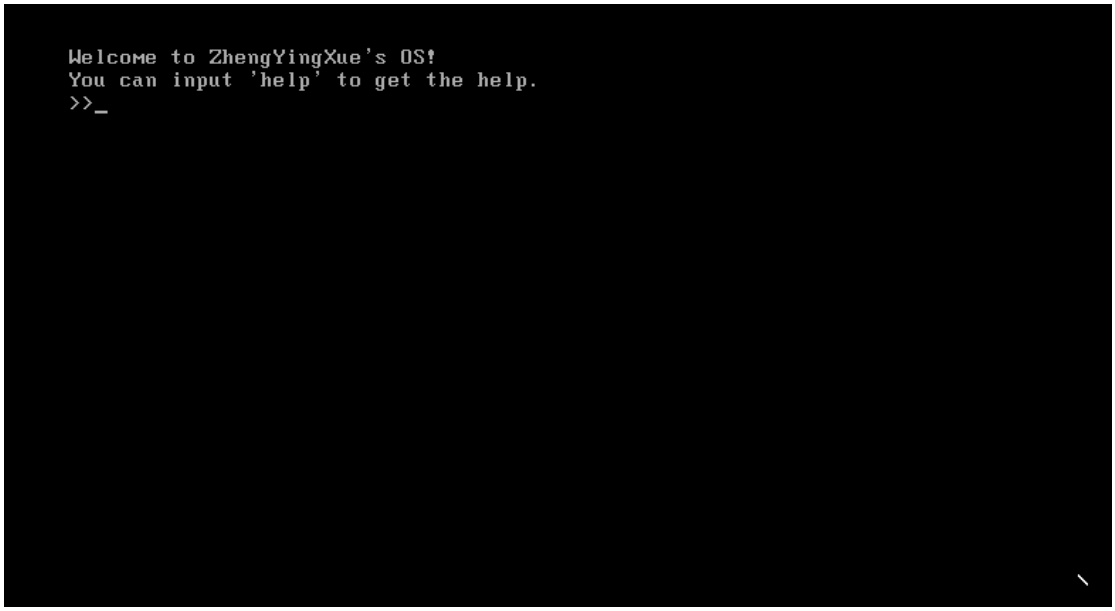
以上的功能，在内核中封装进系统调用才可以被用户程序直接使用。我将原本的 33 号中断（21h）中的前四个功能号设置成调用上面的程序。为了不使报告冗长，以 fork 为例，代码如下：

```
……
        cmp ah, 1
        je to_forking
……
to_forking:
        pop es
        pop ds
        pop bp
        jmp forking
……
forking:
        .386
        push ss
        push gs
        push fs
        push es
        push ds

        .8086
        push di
        push si
        push bp
        push sp
        push dx
        push cx
        push bx
        push ax

        mov ax,cs
        mov ds, ax
        mov es, ax

        call _save_PCB
        call near ptr _do_fork
        iret
……
```

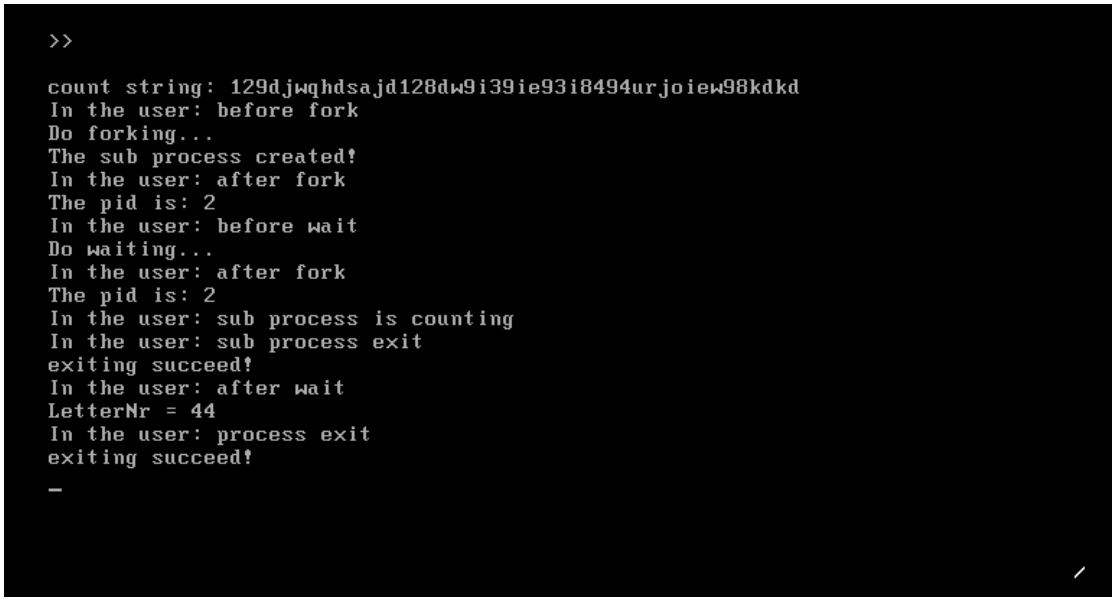
其余的代码部分已经在以前的实验报告中分析过，故此处就不贴上来了。

四、实验过程

1、进入内核。



2、输入“test”，得到如下显示：



我们可以完整地看到过程。首先父进程创建了字符串，此时还没有 fork。后来进行 fork 之后，子进程成功创建，执行 wait，父进程成功阻塞。子进程此时开始为单词计数，计数完成后执行 exit 退出。随后父进程善后，输出字符串的

长度 44，然后父进程退出。由此，用户程序的演示完毕。

由于本实验只涉及此用户程序的测试，所以别的功能就不重复截图了。

五、实验总结

这次实验是在实验六的基础上进行一点扩充。总体来说主要是为了实现进程控制原语实现父子进程交互，所以测试的内容也不多。

在实验进行的过程中，解决了上次实验中头文件无法读取的问题。在头文件中添加条件指示符#ifndef，生成的 com 文件就不会像上次一样莫名其妙被吞掉一部分了。不过中间遇到过“类型不匹配”的问题，上网搜了原因，发现是没有声明函数。可是我的函数并没有前面调用后面这种操作，放在别人的电脑上运行也不会报错，我知道可能是我的环境有问题，但是为了继续之后的实验，我还是老老实实把函数都声明了一遍，这才成功编译了。

一开始在运行用户程序的时候，卡在了开始的地方。说明从 fork 开始就有问题。最后测试过后发现是在复制栈段的时候，没有给子进程的栈段单独开辟新的空间，可以说这是一个粗心造成的低级错误了。后来在用户程序运行的时候，卡死在了父进程阻塞之后，之后就回不来了。对比一下参考原型发现是我在子程序退出之后没有将进程数-1，即没有加上 process_number--这一行，导致没有办法返回父进程将结果输出，还是一种粗心的错误。

还遇到的一个比较玄学但目前还没办法解决的问题是，当一开始执行 run（即上一个实验中）后，因为设置的是过一段时间再退出，而退出后再执行本次实验中执行 test 指令时，之前的四个象限飞翔字符的子程序也一起并发运行了……因为四个象限的子程序和 test 的父子进程的子程序在显示上并没有彼此独立，所以那些飞翔的字符还直接遮挡了父子进程的用户程序的内容……我仔细比对了代码，发现是在上一个实验当中，我对四个用户程序的调度与本次实验中对父子进程的调度是一样的。也就是说此次的用户程序实际上也是上一次用户程序的队列的一员，而在运行一次 run 之后，四个用户程序加入到了父子进程所在的队列当中。所以当在此之后再执行 test 时，等于说是再执行一次调度，导致原本在队列里的用户程序也运行了。而一进入内核就输入 test 时，队列里只添加了

本次实验中测试的用户程序，所以不会出现这种情况。这种情况我目前还没有找到解决的办法，好在与本次实验中需要实现的原语关系不是很大，所以我先提交目前这一版本。

通过这一实验，我对理论课上学习的进程调度的原语有了更多的认识。其实这一内容，包括信号量，我在理论课上学得还不是完全理解的，有些懵逼，但通过这次实验，我对进程控制的原语有了更进一步的理解了。我相信等我做完实验八，我也会对信号量的认识更加清楚。

六、参考文献

1、凌老师的实验七课件。

2、参考原型：

https://github.com/AngryHacker/OS_Lab/tree/master/Lab07

3、进程的描述与控制 进程的状态和转换 三态模型和五态模型

https://blog.csdn.net/sdr_zd/article/details/78748497