操作系统实验五报告

智能科学与技术　　2024 年 5 月 17 日

实验任务

在 PA4 基础上，增加 fork 系统调用实现，使得支持实现 ping 与 pong 交错打印的应用程序。其中要求：

1）进程调度采用非抢占式优先级调度算法，优先级排序：用户父进程 > 用户子进程 > idle 进程；

2）fork 函数返回值，父进程大于 0，子进程等于 0

1 实验环境

1. 电脑操作系统：Windows 11 + 阿里云 Alibaba Cloud Linux（基于 centos）+wsl(ubuntu)
2. 编程语言：nasm 汇编语言 +C 语言混合编程
3. 辅助工具：

（1）使用 VScode 用于编辑各种文件。

（2）使用本机 Windows11 上的 QEMU 创建一个简单的虚拟机进行实验。

```
qemu-img create -f raw disk.raw 10M
```

（3）使用 Xshell 7 远程连接阿里云服务器，构建 Linux 系统环境，方便使用各种转换与调试工具（makefile，gcc，objcopy，objdump，nasm，ld），使用 Xftp7 进行本机与服务器的文件传输。

（4）使用 makefile 组织和管理整个实验项目的编译、部署和运行

（5）GDB 调试工具

注意：本实验在 Windows11 上的 QEMU 以及在 wsl（Ubuntu）上均可完成（直接进入 app 文件夹在命令行输入 make 和 make qemu 即可，因为 app 文件夹里也有 kernel.asm），但是其他操作系统与平台未尝试过

2 实验原理

回顾实验四，我们实现了生成 kernel.bin 与 app.bin 的基本流程，并实现了 write 和 sleep 系统调用。本次实验在整个实现流程上与 PA4 并无差异，只需要在 PA4 的基础上实现 fork 系统调用，以及实现三个进程的非抢占式调度即可。因此，基本可以明确本次实验的关键点与难点在于如何实现父进程与子进程，idle 进程三个进程之间的调度与切换。

2.1 fork 系统调用

首先实现 fork 系统调用。fork() 系统调用的主要目的是创建一个新的进程，使得父进程和子进程可以并发执行，各自独立运行。当调用 fork() 时，操作系统会创建一个新的进程，称为子进程，其代

码、数据、堆栈和其他相关资源与父进程几乎完全相同。子进程的创建是通过复制父进程的地址空间来实现的。这包括父进程的内存映像，其中包括代码段、数据段、堆和栈。然后，操作系统会调整子进程的某些部分，例如将子进程的进程 ID 设置为新值，以及更新进程状态等。子进程和父进程会继续执行 fork() 之后的代码。通常，通过 fork() 返回的值来区分父进程和子进程。在父进程中，fork() 返回子进程的进程 ID，而在子进程中，fork() 返回 0。

因此，我们在 PA4 的 syscall.asm 文件中添加 fork 系统调用函数的实现，在该函数中需要进行的是将 ax 设置为 1（因为 c 语言默认使用 ax 进行返回值的传递，设置为 1 则标识该进程为父进程），并将 ax 压进栈中（因为需要在 fork 内核函数中修改子进程的栈该位置的值为 0 以区分父进程和子进程），然后发出软中断陷入内核态并进入 fork 内核函数进行进程的复制与修改，在中断处理结束回来后，进行 pop ax 操作，因为父进程和子进程回来后的栈顶数值分别是 1 和 0，以此向 c 语言函数传递不同的返回值来区分不同的进程进行不同的操作。

然后需要在 kernel.asm 中实现 fork 的内核函数。首先为 fork 分配 82h 中断号，然后我们需要在 fork 内核函数中实现以下任务：

（1）复制代码段，数据段以及栈的内容到为子进程分配的内存地址（0x9000）中

（2）将从父进程复制而来的子进程的栈中所有返回地址修改为子进程所在内存的相应地址，并修改栈中的 fork 返回值为 0

2.2 实现进程调度与切换

最后，我们考虑如何实现进程调度与切换（共有 father->child,father->idle,child->father,child->idle,idle->father,idle->child 六种情况）。通过时钟中断（08h）和 sleep 内核函数，以及维护一些标志变量可以实现父进程，子进程与 idle 进程的择优调度。

由于 sleep 内核函数只可能在父进程或子进程运行过程中进入，则考虑在 sleep 后进入父进程与子进程互相调度切换的调度程序，以及从两个进程切换到 idle 的调度程序（主要负责 father->child,child->father,father->idle,child->idle，处理进程 sleep 后的调度），并在数据段为父进程和子进程维护一个状态变量（0 表示睡眠态，1 表示就绪态，2 为运行态）。

而时钟中断处理则负责进行父进程和子进程的睡眠时间的检测与递减，并且在两个进程睡眠时间结束后修改进程状态为就绪态，紧接着进行状态的检测和调度，由于父进程与子进程的优先度高，所以一旦它们中任意一个处于运行态，时钟中断处理就什么都不做，而一旦两个进程都不处于运行态（说明此时是运行 idle 进程），则必须按优先度顺序检测两个进程是否有处于就绪态的进程并进行调度（时钟中断处理主要负责 idle->father,idle->child 的调度），时钟中断处理按照严格的“递减睡眠时间”，“检测睡眠时间剩余并转换进程状态”，“检测进程状态并进行调度”的流程，与 sleep 后的调度程序一同保证了六种进程切换的实时更新与正确进行。

3 实验过程

按照 PA4 的流程，我们需要得到 kernel.bin 与 app.bin 两个二进制文件并分别将它们载入硬盘的第一个扇区和第二个扇区。而根据实验原理的分析，实际上，我们只需要在 PA4 的基础上修改 main_function.c，myos.h，syscall.asm 与 kernel.asm 四个文件，将 fork 系统调用加入 app.bin 中并将新的调度逻辑加入 kernel.asm 中。

3.1 生成 app.bin

参照 PA4 的流程，对于重复的部分不再赘述，根据实验原理的分析，给出相关文件代码与注释。

3.1.1 实现 API 库

myos.h

```
#ifndef MYOS_H
#define MYOS_H
// 声明write函数原型
void write(char *msg, short len);
// 声明sleep函数原型
void sleep(short seconds);
//声明 fork函数原型
short fork(void);
#endif /* MYOS_H */
```

syscall.asm

```
BITS 16
section .text
    global write
    global sleep
    global fork

write:
    push ebp
    mov ebp, esp ;建临时栈
    push ebx ;保存要用到的寄存器
    push ecx

    ;注意此处容易出错：在跳转write之前在栈中还会压入跳转前下一条指令的地址，所以第一个参
        数应该是+8
    mov ebx, [ebp+8] ;第一个参数，字符串地址，char*类型
    mov ecx, [ebp+12] ;第二个参数，字符串长度,在h头文件里是short类型

    int 0x80 ;write内核函数

    pop ecx
    pop ebx
    pop ebp

    retd ;32位回跳

sleep:
    push ebp
    mov ebp,esp
```

```
    ;dx传递秒数参数
    push edx
    mov edx,[ebp+8]
    int 0x81
    pop edx
    pop ebp
    retd

fork:
    ;使用eax作为返回值传递
    ;此时仍旧是father_stack
    mov eax,1 ;父进程返回值
    push eax ;压入栈中是为了修改子进程的返回值为0
    int 0x82
    ;要保证ax中是返回值
    pop eax
    retd
```

3.1.2 main 函数与组织文件

main_function.c

```
#include "myos.h"

int main(void)
{ if(fork()==0)
  {
    while(1)
    {
      write("Ping!",5);
      sleep(1);
    }
  }else{
    while(1)
    {
      write("Pong!",5);
      sleep(2);
    }
  }
}
```

应用程序实例要求根据 fork 不同的返回值识别不同的进程，对于父进程（fork 返回值为 1），只会执行循环输出“Pong!”，对于子进程（fork 返回值为 0），只会执行循环输出“Ping!”, 实际上，通过分析 app.elf 反汇编代码可以发现，fork 函数只会在父进程中执行一次，因为在比较一次 ax 的值之后就会进入各自的循环区域，无论是调用 sleep 还是 write 最后都只会回到各自的循环区域指令段（Figure 1）。

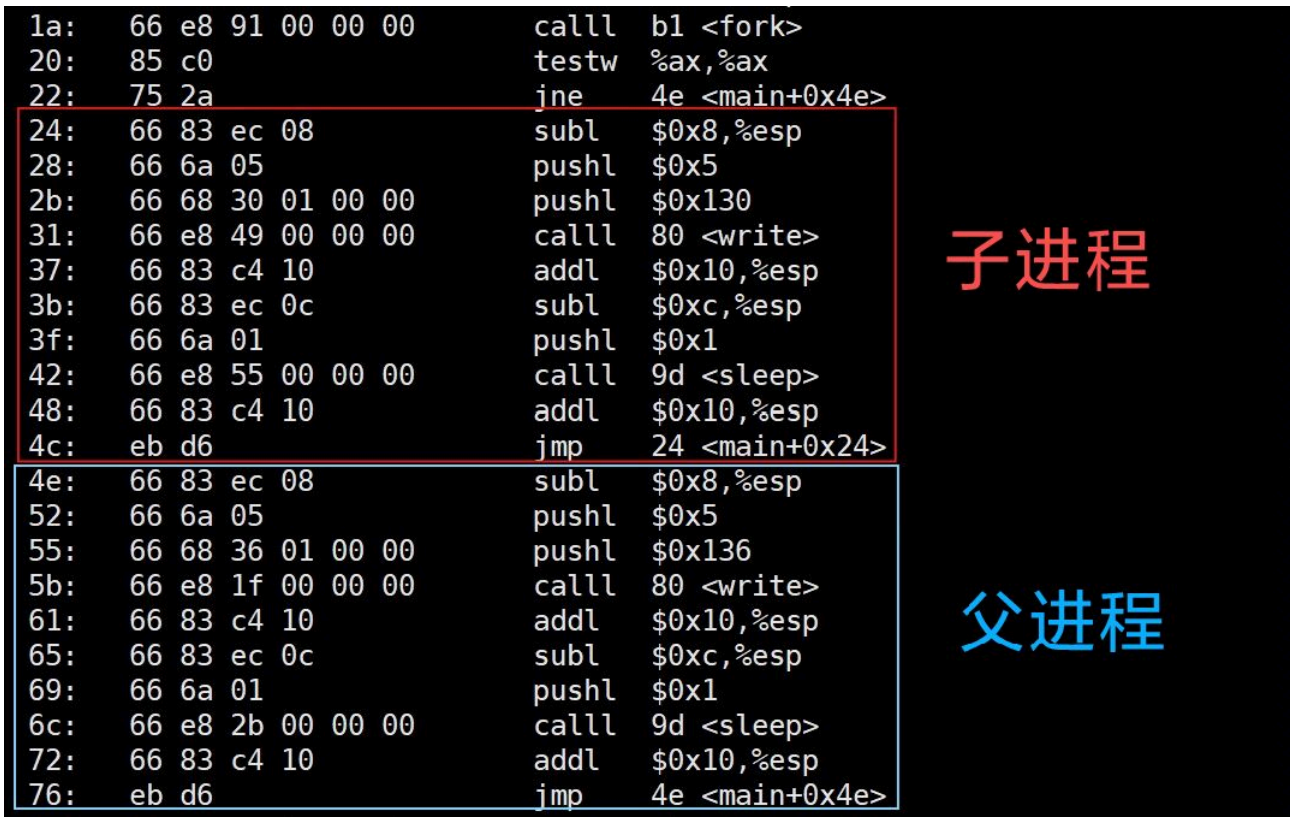

Figure 1: app.elf 反汇编代码中 main 函数片段

my.ld

```
SECTIONS {
  . = 0x0000;
  .text : {*(.text)}
  . = 0x0130;
  .rodata : {*(.data)}
}
```

注意，由于此次实验有两个 app 进程且位于不同的地址，所以字符串的编排需要使用相对地址，然后读取地址再针对不同的进程进行绝对地址的计算。

makefile

```
app.bin:app.elf
    objcopy -R.eh_frame -R .comment -O binary app.elf app.bin

app.elf:main.o syscall.o
    ld -m elf_i386 -T my.ld -e main main.o syscall.o -o app.elf

main.o:main_function.c myos.h
    gcc -m16 -ffreestanding -fno-pic -I. -c main_function.c -o main.o

syscall.o:syscall.asm
    nasm -f elf32 syscall.asm -o syscall.o

```

```
qemu:
    qemu-img create -f raw disk.raw 10M
    nasm -f bin -o kernel.bin kernel.asm
    dd if=kernel.bin of=disk.raw bs=512 count=1
    dd if=app.bin of=disk.raw bs=512 seek=1 count=1
    qemu-system-x86_64 -drive file=disk.raw,format=raw

clean:
    rm app.elf syscall.o main.o
```

3.2 实现内核 kernel.asm

参照 PA4 的流程，对于重复的部分不再赘述，根据实验原理的分析，给出相关文件代码与注释。重点讲述 fork 内核函数，调度程序与时钟中断处理程序。

3.2.1 数据段与内核初始化

数据段需要定义计数器变量，状态变量以及栈变量。

其中 state_father 与 state_child 分别用于记录父进程与子进程当前的状态，0 表示睡眠态，1 表示就绪态，2 为运行态，根据整体逻辑，一开始必然是父进程处于运行态，进行 fork 后子进程处于就绪态，为了节省指令字节，这两步在初始化进行，按照严谨的逻辑其实需要额外进行。

state_current 记录当前运行的进程是哪个进程，0 表示父进程，1 表示子进程，2 表示 idle 进程，该变量的设置是为了方便调度程序中对于当前运行进程的栈指针的保存。

FATHER_STACK，CHILD_STACK，IDLE_STACK 三个变量分别用于记录不同进程的栈顶地址，协助进程切换的进行。三个栈的栈底分别初始为 0x8000（father）,0x9000（child）和 0x6000（idle）。

```
data:
counter1 dw 0 ; 父进程计数器
counter2 dw 0 ; 子进程计数器
counter3 dw 0 ; 用于write换行

state_father dw 2 ;记录父进程状态,0表示睡眠态，1表示就绪态，2为运行态
state_child dw 1 ;记录子进程状态,0表示睡眠态，1表示就绪态，2为运行态
state_current dw 0 ; 记录当前运行的进程是哪个进程，0表示父进程，1表示子进程，2表示idle
    进程

FATHER_STACK dw 0x8000 ;父进程应用程序栈
IDLE_STACK dw 0x6000 ;idle栈
CHILD_STACK dw 0x9000 ;子进程应用程序栈
```

内核初始化，包括中断向量表的填写，设置时钟频率和 app 的加载。并给新增加的 fork 内核分配 82h 中断号。

```
;设置代码段的起始位置
BITS 16
SECTION MBR vstart=0x7c00
```

```
4  _start:
5      ;清屏
6      mov ax, 0600h
7      mov bx, 0700h
8      mov cx, 0
9      mov dx, 184fh
10     int 10h
11
12     ;初始化栈: 给应用程序app用的
13     xor ax,ax
14     mov sp,0x8000
15     mov ss,ax
16
17     ;初始化段地址
18     mov es,ax
19     mov ds,ax
20     ;写入中断向量表
21         ;时钟中断处理: 08h
22     mov word [32],clock_interrupt_handler
23     mov word [32+2],0
24
25         ;write: 80h
26     mov word [0x80*4],write_function
27     mov word [0x80*4+2],0
28
29         ;sleep: 81h
30     mov word [0x81*4],sleep_function
31     mov word [0x81*4+2],0
32
33         ;fork: 82h
34     mov word [0x82*4],fork_function
35     mov word [0x82*4+2],0
36
37     ;设置时钟频率
38     ; 设置8253的控制字, 选择通道0并设置为方式3
39     mov dx, 0x43
40     mov al, 0x36
41     out dx, al
42
43     ;设置计数初值为23863
44     mov dx, 0x40
45     mov ax, 23863
46     out dx, al
47     mov al, ah
48     out dx, al
49
50     ;加载app
```

```
    mov bx,0x8000
    mov ah, 2 ; BIOS读取扇区功能号
    mov al, 1 ; 读取扇区的数量
    mov ch, 0 ; 柱面号清零
    mov cl, 2 ; 从第二个扇区开始
    mov dh, 0 ; 磁头号清零
    mov dl, 0x80 ; 使用默认的驱动器号

    int 0x13 ; 调用BIOS中断13h读取磁盘扇区

    ;按严谨的逻辑需要这两个指令，但是为了节省指令字节，用数据段初始化代替
    ;mov word [state_father],2 ;将app父进程设为运行态
    ;mov word [state_current],0 ;标志当前进程为app father进程

    sti
    pushf
    push word 0 ;cs
    push word 0x8000 ;ip
    iret ;从内核到应用程序执行
```

3.2.2 write 内核函数

与上次实验相比并无大的差别，但是由于由于此次实验有两个 app 进程且位于不同的地址，所以 my.ld 字符串的编排使用相对地址，实际情况中，可以根据 state_current 中的进程标志来选择不同计算绝对地址的操作（实际是 add 0x8000 或者 0x9000），但其实两个进程的数据段都有“Ping!”和“Pong!”字符串，所以这个区分其实没多大影响，为了节省指令字节，本实验统一用父进程的数据段字符串。

```
 ;int 80h
write_function:
    ;xor ax,ax
    ;mov ds,ax
    ;mov es,ax
    ;bx=字符串相对地址, cx = 串长度
    add bx,0x8000
    mov bp,bx ; es:bp = 串地址
    mov ax,01301h ; ah = 13, al = 01h
    mov bx,000ch ;页号为 0(bh = 0) 黑底红字(bl = 0Ch,高亮)
    mov dh,[counter3] ;显示的行号
    mov dl,39 ;显示的列号
    int 10h
    inc byte [counter3]
    cmp dh,24 ;当到达页面最后一行，那么返回第一行输出
    jz zero
    iret
zero:
```

```
    mov word [counter3],0
    iret
```

3.2.3 fork 内核函数

根据实验原理，需要实现：

（1）复制代码段以及栈的内容到为子进程分配的内存地址中

（2）将从父进程复制而来的子进程的栈中所有返回地址修改为子进程所在内存的相应地址，并修改栈中的 fork 返回值为 0

```
;int 82h
fork_function:
    ;复制app 从0x8000到0x9000，连带着栈的内容一起复制，这里有个易错点，栈是从高位往低位
        存储的，而栈底初始化为0x8000，所以栈保存的内容在0x8000之上，需要从当前sp指向的
        地址开始复制，这样确保将栈的内容一同复制了
    ;xor ax, ax
    ;mov es, ax
    ;mov ds, ax

    ;复制到di
    mov di,sp
    add di,0x1000
    ;从sp栈顶开始复制
    mov si, sp
    mov cx, 0x250 ;592个字节
    cld
    rep movsb
;DS:SI
;ES:DI
;CX 复制字节数
;DF=0 （cld）
;rep movsb将CX个字节从DS:SI复制到ES:DI处，SI和DI会自动+2

    ;易错: di在上一步会不断增加，已经不再是子进程的栈顶了，需要重新初始化
    mov di,sp
    add di,0x1000
    mov [CHILD_STACK],di ;保存栈指针

    ;修改fork的栈内容
    ;栈顶是iret回去的IP，会回到fork系统调用发出中断后的下一条指令，要修改成相应的0x9***
    mov ax,[di]
    add ax,0x1000
    mov [di],ax ;修改iret的地址

    mov ax,0x0000
    mov [di+6],ax ;修改栈中返回值ax
```

```
    mov ax,[di+8]
    add ax,0x1000
    mov [di+8],ax ;修改从fork系统调用返回main函数的地址

    ;mov word [state_child],1 ;修改app的fork子进程为就绪态
    iret ;此处仍旧是返回app的父进程
```

3.2.4 sleep 内核函数 + 进程调度程序

根据实验原理中的分析，sleep 函数中的进程调度程序主要负责处理父进程或子进程发出 sleep 中断后的调度，即 father->child，child->father，father->idle，child->idle。首先根据 state_father 和 state_child 的状态区分是哪个进程发出的 sleep 中断，再进行对应的处理。

```
; int 81h
sleep_function:
    ; dx里装着秒数
    imul dx, 50 ; 将ax和cx的值相乘，结果存入dx
    cmp word [state_father],2
    jz father_sleep ;父进程处于运行态，说明从父进程来的
;如果不是父进程来的就是子进程来的，顺序执行下面的指令
child_sleep:
    ;dx中是倒计时的数值，将秒的值存入子进程对应的计数器counter2
    mov [counter2], dx
    ;检测父进程是否处于就绪态
    cmp word [state_father],1
    ;在跳转前修改子进程状态为睡眠态，之所以不在一开始就修改，是为了避免时钟中断处理的干
        扰（因为只要父进程状态仍处于运行态时钟中断处理就不会进行任何处理）
    mov word [state_child],0
    jz child_to_father ;处于就绪态则调度父进程
    ;否则调idle
    mov [CHILD_STACK],sp ;每次调度前都记得保存当前进程的栈指针
    jmp to_idle
;从父进程发出的sleep
father_sleep:
    ;dx中是倒计时的数值，将秒的值存入counter
    mov [counter1], dx
    ;检测子进程是否处于就绪态
    cmp word [state_child],1
     ;在跳转前修改父进程状态为睡眠态
    mov word [state_father],0
    jz father_to_child ;处于就绪态则调度子进程
    ;否则调idle进程
    mov [FATHER_STACK],sp ;每次调度前都记得保存当前进程的栈指针
    jmp to_idle

;调度idle进程：这是个通用的指令段，既可以从父进程进行调度切换，也可以从子进程进行调度切
```

```
    换，节约字节
to_idle:
    mov sp,[IDLE_STACK] ;切换栈
    mov word [state_current],2 ;当前运行进程改为idle

    sti
    pushf ;保存标志寄存器状态
    push word 0 ;cs
    push word idle ;ip
    iret ;使用iret从内核到进程函数idle

;sleep时间从子进程调度切换至父进程
child_to_father:
    mov word [state_father],2 ;父进程切换至运行态
    mov word [state_current],0 ;当前运行进程改为父进程
    ;保存当前运行进程栈指针，切换为即将运行的进程的栈
    mov [CHILD_STACK],sp
    mov sp,[FATHER_STACK]
    iret
;sleep时间从父进程调度切换至子进程
father_to_child:
    mov word [state_child],2 ;子进程切换至运行态
    mov word [state_current],1 ;当前运行进程改为子进程
    ;保存栈指针，切换栈指针
    mov [FATHER_STACK],sp
    mov sp,[CHILD_STACK]
    iret
```

3.2.5 idle

idle 只会运行在两个 app 进程都睡眠的情况下，且可以随时打断。

```
idle:
    jmp $
```

3.2.6 时钟中断处理程序（调度程序）

根据实验原理的分析，时钟中断处理程序主要负责 idle->father,idle->child 的调度，需要按照严格顺序执行“递减睡眠时间”，“检测时间剩余并转换进程状态”，“检测进程状态并进行调度”的流程来完成一次时钟中断处理。

```
;时钟中断处理例程
clock_interrupt_handler:
;首先更新当前运行的进程的栈地址，每个更新段update_**都会跳回到下面的check_father，相当
    于一个函数调用
    mov dx,[state_current]
```

```
    cmp dx,0
    jz update_father
    cmp dx,1
    jz update_chlid
    cmp dx,2
    jz update_idle
;检测父进程是否处于睡眠态
check_father:
    cmp word [state_father], 0
    jz handle_father_sleep ;是则进行对应计数器减一操作以及进行睡眠时间是否结束的检测。如果睡眠时间结束，则修改父进程状态为就绪态；无论结没结束最后都会回跳到下面的check_child，因此实现两个进程的睡眠时间减一操作不会互相干扰
;检测子进程是否处于睡眠态
check_child:
    cmp word [state_child], 0
    jz handle_child_sleep ;是则进行对应计数器减一操作以及进行睡眠时间是否结束的检测。如果睡眠时间结束，则修改子进程状态为就绪态；无论结没结束最后都会回跳到下面的choose

;完成以上的对于父进程和子进程的睡眠态处理后，无论它们处于什么状态，每次时钟中断发出后，都需要进行状态检测，并按照父进程>子进程>idle进程的优先级进行进程的选择与调度（因为idle是可以随时被打断的换成更高级的进程的，所以需要不断检测）
choose:
    ;父进程和子进程是否有在运行态的？有就什么都不干
    cmp word [state_father], 2
    jz back
    cmp word [state_child], 2
    jz back

    ;否则说明当前处于idle进程，那么需要进行父进程和子进程的状态检测，一旦有一个处于就绪态，就进行调度切换
    ;首先检测父进程是否处于就绪态(1)，是则进行调度切换
    cmp word [state_father], 1
    jz idle_to_father

    ;父进程不是处于就绪态，那么继续检测子进程是否处于就绪态(1)
    cmp word [state_child],1
    jz idle_to_child

    ;以上情况都不是，则原路返回，back封装了EOI以及iret
    jmp back
;到此时钟中断处理程序需要进行的一系列检测与调度就结束了，以下是一些上面用到的封装“函数”

;更新保存当前运行进程的栈地址，最后会回到时钟中断处理
update_father:
    mov [FATHER_STACK],sp
    jmp check_father
```

```
update_chlid:
    mov [CHILD_STACK],sp
    jmp check_father
update_idle:
    mov [IDLE_STACK],sp
    jmp check_father

;检测到父进程处于睡眠态后，进行对应计数器减一操作以及进行睡眠时间是否结束的检测。如果睡眠时间结束，则修改父进程状态为就绪态；无论结没结束最后都会回跳到时钟中断处理中的check_child
handle_father_sleep:
    cmp word [counter1],0
    mov word [state_father],1 ;恢复就绪态，这里用了假设的思路，假如上面的cmp成立，说明睡眠时间结束了，那么修改成就绪态后就跳回时钟中断处理，否则下面再恢复为睡眠态
    jz check_child ;时钟中断处理中的段标签
    ;进行到这说明睡眠未结束，将状态重新恢复为睡眠态，并进行减一操作，再返回时钟中断处理
    dec byte [counter1]
    mov word [state_father],0
    jmp check_child ;时钟中断处理中的段标签

;检测到子进程处于睡眠态后，进行对应计数器减一操作以及进行睡眠时间是否结束的检测。如果睡眠时间结束，则修改子进程状态为就绪态；无论结没结束最后都会回跳到时钟中断处理中的choose
handle_child_sleep:
    cmp word [counter2],0
    mov word [state_child],1
    jz choose ;时钟中断处理中的段标签

    dec byte [counter2]
    mov word [state_child],0
    jmp choose ;时钟中断处理中的段标签
;从idle进程调度切换至父进程
idle_to_father:
    mov sp,[FATHER_STACK] ;切换栈
    mov word [state_father],2 ;修改父进程状态为运行态
    mov word [state_current],0 ;修改当前运行进程为父进程
    jmp back
;从idle进程调度切换至子进程
idle_to_child:
    mov sp,[CHILD_STACK]
    mov word [state_child],2
    mov word [state_current],1
    jmp back
;封装了EOI和iret,节省字节
back:
    mov al,20h
    out 20h,al
```

```
87      iret
```

4 实验结果

启动虚拟机，根据 main 函数的 sleep 参数设置（父进程 sleep 2 秒，子进程 sleep 1 秒），在启动界面不断交错换行打印字符串“Pong！Ping！”和“Ping！”，且每个“Ping！”之间间隔 1s，每个“Pong！”之间间隔 2s, 连续输出“Pong！Ping！”后隔一秒会再输出 Ping！。

Figure 2: 启动 1s 后

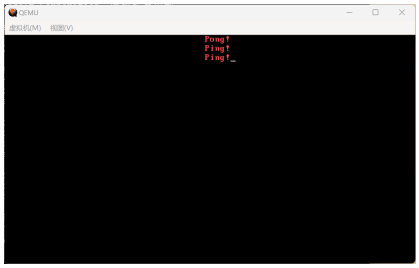

Figure 3: 启动 2s 后

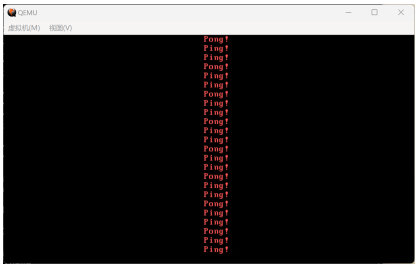

Figure 4: 启动 n s 后

5 实验改进: 新增全局变量

为了体现父进程与子进程之间的数据段互不干扰的性质，现增加新任务：修改 main_function.c，设置全局变量 t，在子进程进入 while 循环前将 t 设置为 1，父进程进入循环前将 t 设置为 2，然后分别作为参数传给 sleep，实现效果与上面的实验一致。

main_function.c

```
1   #include "myos.h"
2   int t=0;
3   int main(void)
4   { if(fork()==0)
5     {
6       t=1;
7       while(1)
8       {
9         write("Ping!",5);
10        sleep(t);
11      }
12    }else{
13      t=2;
14      while(1)
15      {
16        write("Pong!",5);
17        sleep(t);
18      }
19    }
20  }
```

该新任务的难点在于，通过 objump 反汇编 app.elf 发现，当我们 gcc 得到 app.bin 后，在 if 和 else 两个模块内对 t 进行修改以及使用 t 进行传参时，访问到的 t 的地址都是相同的，且 app.bin 已经固定无法修改。如何使得父进程和子进程拥有各自的变量 t 并且在同一段代码里能够读写不同地址的变量 t？其实我们注意到，当访问一个变量时，如指令 mov [variant_address],ax ，实际上是 mov [ds:variant_address],ax ，因为我们没有指出段地址的时候，会默认使用 ds 数据段地址寄存器存储的地址作为段地址，也就是说，一个变量的地址实际上是由 ds:variant_address 决定的。而反编译 app.elf 可以发现，在 main 函数的逻辑中，并没有指明 ds，ss，es 等段地址的处理，也就是说实际上 app 默认在内核所在的段上执行。

参考 Linux 的写时复制机制（在进行子进程的时候将子进程的地址映射到虚拟物理地址上）以及以上分析，为了能够在子进程中能够使用它自己的变量 t，我们可以设置父进程和子进程的数据段的偏移地址相同（也即在 main 函数中访问到的 t 的地址），然后设置段地址不同，这样在进行两个进程的切换时就不用考虑偏移地址，只需要修改段地址。我们可以考虑将子进程的栈，代码段，数据段复制到不同于父进程的另一个段上的相同偏移地址上，也即从 0x0000：0x8000 拷贝到 0x2000：0x8000 上，修改后的 fork 内核函数如下：

```
;int 82h
fork_function:
    ;复制app从0x0000:0x8000到0x2000:0x8000，连带着栈的内容,代码段，数据段一起复制，这里有个易错点，栈是从高位往低位存储的，而栈底初始化为0x8000，所以栈保存的内容在0x8000之上，需要从当前sp指向的地址开始复制，这样确保将栈的内容一同复制了
     xor ax, ax

    ;复制到es:di
    mov word es,[CHILD_ES]
    mov di,sp ;es:di=0x2000:sp

    ;从sp栈顶开始复制
    mov si, sp ;ds:si=00:sp
    mov cx, 0x250 ;592个字节
    cld
    rep movsb
;DS:SI
;ES:DI
;CX 复制字节数
;DF=0 （cld）
;rep movsb将CX个字节从DS:SI复制到ES:DI处，SI和DI会自动+2

    mov es,ax ;复原es段地址为0x0000
    mov [CHILD_STACK],sp ;保存子进程栈指针

    ;修改fork的栈内容
    ;注意，因为我们要修改的是子进程的栈的内容，子进程的栈在0x2000段，所以需要将数据段地址寄存器ds修改为子进程所在段
    mov word ds,[CHILD_ES]
    mov di,sp
```

```
28      mov [di+6],ax ;修改栈中返回值ax
29      mov ds,ax ;复原
30
31      iret ;此处仍旧是返回app的父进程
32  ...
33  data:
34  CHILD_ES dw 0x2000 ;子进程所在段地址
35  OTHER_ES dw 0x0000 ;父进程和idle以及内核所在段地址
```

接下来，修改内核 kernel.asm 文件，在所有需要切换到子进程的逻辑段中添加以下极为重要的俩个部分：修改数据段寄存器 ds 为 CHILD_ES(0x2000，为子进程分配的段地址)，修改栈段寄存器 ss 为 CHILD_ES。而结合上面的分析可知，实际上子进程和父进程仍旧是共用同一段代码段，然后需要使用不同的栈和不同的数据段，也即栈和数据段的段地址与父进程不同，而取数据时使用到的是 ds 段寄存器，使用栈时使用到的是 ss 段寄存器，所以只需要修改这两个寄存器即可。

在内核所有从子进程切换到其他进程的逻辑代码段内，也需要做同样的复原操作：修改数据段寄存器 ds 为 OTHER_ES(0x0000，原本的段地址)，修改栈段寄存器 ss 为 OTHER_ES。值得注意的是，在时钟中断处理和 write，sleep 内核函数中需要使用到众多变量用于判断，而所有的变量都在 0x0000 段，所以一定要添加相关的逻辑保证在不管在什么情况下（父进程或是子进程运行时）使用访问这些变量时段地址为 0x0000，比如可以修改 ds 为 0x0000 后再使用 mov [variant_address],ax 等指令，或者直接 mov [es:variant_address],ax（因为在代码逻辑中，可以确保 es 一直为 0x0000, 这样可以节省代码字节)。所有的确保机制我都在代码文件 kernel.asm 中指明并注释了，具体可以参考新代码文件的注释。

eg.

```
1   mov ss,[CHILD_ES]
2   mov ds,[CHILD_ES]
3   ...
4   cmp word [es:state_father],2
```

此外，以上实验过程是使用未添加新任务前的代码进行解释，在添加了新的任务后，对 my.ld 进行了修改（主要是将相对地址改成绝对地址），并且修改了 kernel.asm。kernel.asm 的主要逻辑并没有改变，只是为了空出代码字节，将 state_current 删除了，直接改为使用判断父进程还是子进程处于运行态来进行相关判断与执行。此外，kernel.asm 只修改了 fork 内核函数和 write 内核函数，以及上面的修改 ds,ss 和一些保证机制等小细节的修改。

注意，实验提交的是新代码，基于旧代码修改的部分可以在提交的代码中查看，已经给出详细的注释。

实验结果: 实验效果和之前的一致，不再赘述。

6 实验中的困难与解决

（1）实验过程的一大困难就是区别两个进程的相同功能的系统调用，因为对应的内核函数只有一个，如何识别是从哪个进程进行的调用？通过借助 state_current（记录当前运行的进程是哪个）或者 state_father 与 state_child 很好地解决了这个问题，但是会出现很多判断与跳转指令，导致 kernel.bin 的字节数刚好占满了 512 个字节，后续需要考虑更节省字节的逻辑与方法.

（2）对于实验过程中出现的易错点与踩过的坑已在代码注释中指出，不再赘述。（所有带有“注意：”字符串的注释都是易错点和踩过的坑）

7 参考文章

1. 时钟中断，调度器，任务切换
2. Linux 系统——fork() 函数详解
3. 操作系统教程（第五版）
4. 写时复制技术详解 (COW)
5. 段寄存器