操作系统实验报告 PA5

221900180 田永铭

2024 年 5 月 17 日

目录

一、实验要求

在 PA4 基础上，增加 fork 系统调用实现，使得支持如下示范应用运行（仅供参考）：

```
#include "myos.h"

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

二、实验环境

- 操作系统：wsl2
- 编程语言：汇编语言 + C 语言
- 使用工具：makefile;qemu;objdump;dd;nasm 等
- 虚拟系统版本:Ubuntu-22

三、实验原理

- **总体原理：**在内核实现需要利用特权指令的功能，将其整理成接口暴露给上层的应用程序，应用程序通过调用接口，实现对内核函数的使用。
- **fork 的原理：**

当一个进程调用 fork() 系统调用时，操作系统会创建一个新的进程，这个新的进程几乎完全是父进程的副本。新进程拥有父进程的代码、数据和上下文信息。在创建新进程时，操作系统会复制父进程的内存空间、文件描述符和其他相关的资源。父进程和子进程之间的唯一区别在于它们的 PID（进程标识符）不同，这是操作系统用来区分不同进程的唯一标识符。

在 fork() 调用之后，通常会在新的子进程中调用 exec() 或者其他类似的函数来执行不同的程序。这样做的一个常见用例是在 Unix 系统中的 shell 中执行命令，shell 会调用 fork() 创建一个新进程，然后在新进程中调用 exec() 来加载并执行用户指定的命令。

需要注意的是，fork() 调用会返回两次：在父进程中，它返回子进程的 PID，而在子进程中，它返回 0。这样可以让父进程和子进程根据返回值来执行不同的逻辑。

而我们的实验就是在 i8086 系统中实现 fork()。

- **其它原理：**已在 PA4 中详细解释，此处不再赘述。

四、实验结果展示

- **视频展示：** 221900180_ 田永铭 _ 实验 5_ 视频演示.mp4
- **助教本地运行检验：** 只需要解压代码压缩文件，在 wsl2 环境下打开终端，只需运行 “make”。如果没有配置好 qemu 路径，需要将 qemu 路径在代码对应位置填写完整。如果再运行不了，可直接直接跑 elf。重复 make 前需要先 make clear 一下。注意，新要求的代码和旧要求的代码我都会上传，请区分。

五、最终成功的实验过程

注意： 此部分着重介绍最终成功的实验过程，而尝试过程、失败过程和各种 debug 过程见后面部分。

实验主要分为以下几个步骤 (在 PA4 的基础上改进)：

1. 理清流程和思路

2. 简化 PA4 的代码，防止本次实验 512 字节不够用
3. 增加 fork 中断
4. 修改 sleep_handler
5. 修改 clock_handler
6. 写入磁盘，查看运行效果
7. 进行一些优化

5.1理清流程和思路

吸取上次实验的教训，我先理清本次实验的流程和思路，流程图如下图所示：

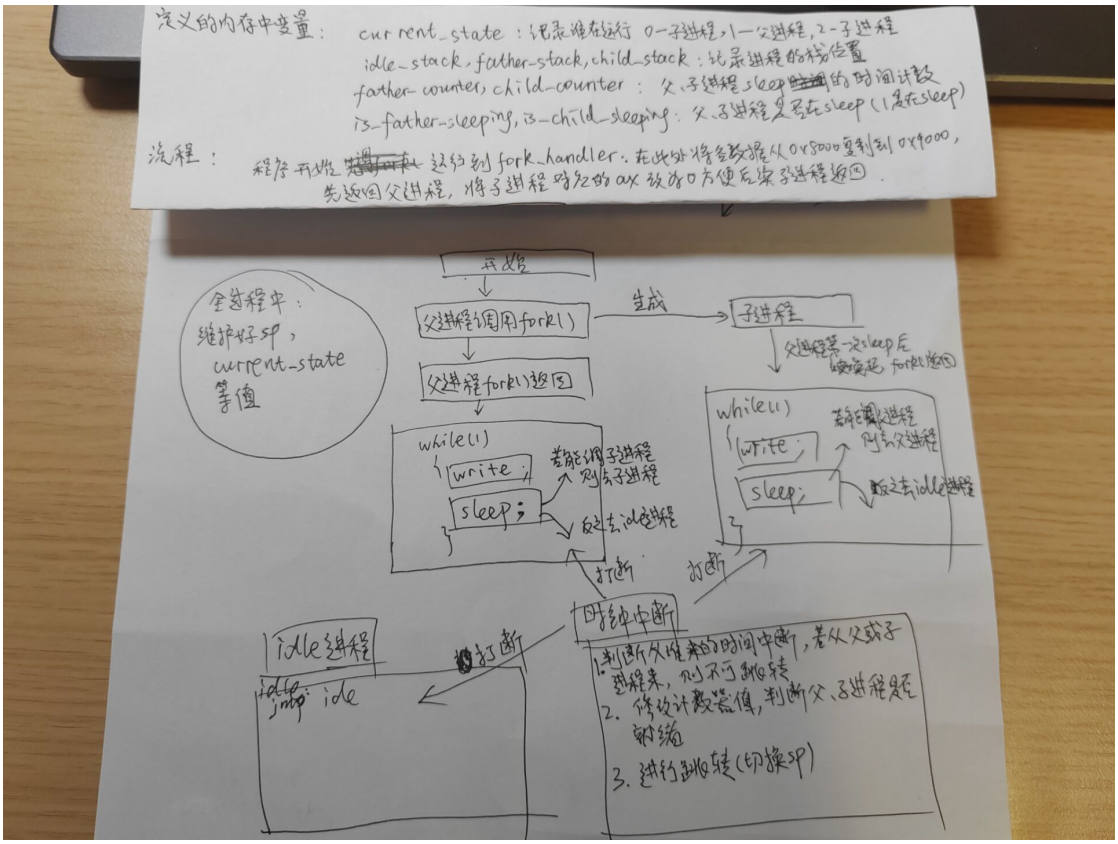

图 1: 流程图

一番梳理后我有了清晰的思路，接下来便是逐一实现。

5.2简化 PA4 的代码，防止本次实验 512 字节不够用

由于本次实验所有的码量比较大，很容易就会超过 512 字节，所以我在动手前先简化了 PA4 实验 kernel.asm 中的代码，主要减少代码的点在于：

1. **封装了清屏部分的代码操作：**封装成了一个函数供调用
2. **简化了重置打印行号的操作：**之前用了一个冗余的计数器，现已简化
3. **简化统计时钟中断时长的方式：**，原本加法改为减法，从而少用了一个计数器
4. **更细节的简化：**封装时钟中断返回的指令，修改内存中的变量的值利用 “word” 方式从三条语句简化成一条语句等等

经过简化，代码量大大减少，可以开始本次实验了。

5.3增加 fork 中断

模仿上一次实验添加中断的方式，可以轻易地添加 fork() 中断，返回值最好设置成 short 方便后续访问 ax 参数。我采用的是 0x82 号中断，而在 fork 中断处理程序中，我们要做的就是拷贝数据、预设子进程返回值为 0、返回父进程，实现代码如下：

首先是 api.asm 中的 fork 实现，需要保护一下 ax 寄存器的数值，即返回值的保护，代码如下：

```
fork:
        ;ax是返回值,先返回父进程1
        mov eax,1
        push eax

        int 0x82

        pop eax
        retd
```

接着是 kernel.asm 中的 fork 实现，代码如下：

```
fork_handler:
        ;从当前sp将各数据复制到+0x1000的地址上
        mov si, sp
        mov di, sp
        add di, 0x1000
```

```
        mov cx, 0x300    ; 超过512个字节即可

        ;复制过程
        copy_loop:
        mov al, [si] ; 将源数据段中的一个字节加载到 AL 寄存器
        mov [di], al ; 将 AL 寄存器中的字节写入目标数据段
        inc si       ; 指向源数据段的下一个字节
        inc di       ; 指向目标数据段的下一个字节
        loop copy_loop ; 循环，直到 CX 寄存器计数为零

        ;复制过程中di改变了，需要重新初始化di成栈顶
        mov di,sp
        add di,0x1000
        mov [child_stack],di ;保存子进程栈指针

        ;修改子进程栈中fork的返回值ax,子进程返回0，short类型
        mov word [di+6], 0

        ;fork返回,先返回父进程，在api中已经将父进程的ax保护为1
        ;而子进程ax返回0
        iret
```

此步需要注意，不能忘了重新初始化 di，因为复制过程中修改了 di 的数值。由此便实现了 fork_handler 函数，它会先返回父进程，待到子进程上台后，子进程 fork() 返回的返回值 ax 正是预先设定的 0，从而实现不同进程的 fork() 返回控制。

5.4修改 sleep_handler

sleep_handler 中的内容相对来说比较容易实现，首先判断是谁调用的 sleep_handler，然后各自处理，判断应该跳转到哪里执行。

```
sleep_handler:
        cli      ;此处关中断防止干扰
                 ;后面每一个分支都有唯一与之对应的sti关中断
        ;根据现在运行状态判断是谁调用了sleep
        cmp word [current_state], 1
        je father_sleep_handler
```

```
        cmp word [current_state], 0
        je child_sleep_handler
        iret

;父进程进入sleep的处理
father_sleep_handler:
        imul bx, 50             ;20ms*50=1s
        mov [father_counter], bx
        mov [father_stack], sp

        ;判断子进程是否可调度
        cmp word [is_child_sleeping], 1
        je father_change_to_idle
        jmp father_change_to_child

;子进程进入sleep的处理
child_sleep_handler:
        imul bx, 50             ;20ms*50=1s
        mov [child_counter], bx
        mov [child_stack], sp

        ;判断父亲进程是否可调度
        cmp word [is_father_sleeping], 1
        je child_change_to_idle
        jmp child_change_to_father
```

具体跳转的时候，实现都比较类似，限于篇幅，在这里仅展示从父进程跳转到子进程的代码，代码如下：

```
father_change_to_child:
        mov word [current_state], 0      ;修改当前状态存储器的内容
        mov [father_stack], sp   ;更新现在的sp指针
        mov sp, [child_stack]    ;修改sp指针到要跳转的地方
        mov word [is_father_sleeping], 1 ;修改父进程是否睡眠的状态
        sti    ;这里的cli与sleep_handler中的cli相互对应
        jmp int_exit     ;时钟中断返回
```

至此，sleep_handler 已经实现。

5.5修改 clock_handler

这一部分的修改比较繁琐，因为时钟中断可能是从父进程、子进程、idle 进程三种情况来的，跳转的结果也有很多种，需要细心实现，代码如下:

首先更新 sp，通过 current_state 判断更新谁的。

```
;时钟中断处理
clock_handler:
        ;首先更新sp，通过current_state判断更新谁的
        mov cx, [current_state]
        cmp cx, 0 ;子进程
        je clock_child
        cmp cx, 1 ;父进程
        je clock_father
        cmp cx,2  ;idle进程
        je clock_idle

        ;由父进程进来的时钟中断
        clock_father:
        mov [father_stack],sp
        jmp handle_clock

        ;由子进程进来的时钟中断
        clock_child:
        mov [child_stack],sp
        jmp handle_clock

        ;由idle进程进来的时钟中断
        clock_idle:
        mov [idle_stack],sp
        jmp handle_clock
```

接下来程序统一转到 handle_clock 进行处理，代码如下：

```
;再统一处理时钟中断
handle_clock:
```

```
;这个函数的所有分支都会汇入judge_if_change

;首先计算以下进中断的时候是否有没在sleep的进程
;如果有，cx会小于2，方便后面处理
mov cx, [is_father_sleeping]
mov dx, [is_child_sleeping]
add cx, dx

;由于绝大部分时间都在sleep，为了省字节;
;不妨父进程和子进程计数器值都更新一下
dec byte [father_counter]
dec byte [child_counter]

;为了考虑极端情况，判断是否苏醒不用计数器是否为0;
;而用计数器是否减到小于1的值
;防止父进程子进程同时能唤醒，父进程先回去了
;子进程计数器为-1而无法后续被唤醒

;判断父进程是否能够苏醒
judge_father:
        cmp word [father_counter], 1
        jl father_awake

;判断子进程是否能够苏醒
judge_child:
        cmp word [child_counter], 1
        jl child_awake
        jmp judge_if_change

;父进程苏醒
father_awake:
        mov word [is_father_sleeping], 0
        jmp judge_child ;需要继续判断子进程能否苏醒

;子进程苏醒
child_awake:
```

```
        mov word [is_child_sleeping], 0
        jmp judge_if_change

    ;判断是否从时钟中断跳转到别处
    judge_if_change:
        ;cx小于2代表子进程或者父进程至少有一个在运行,不做跳转
        cmp cx, 2
        jl int_exit

        ;为了考虑极端情况，判断进程能否跳转不用计数器是否为0;
        ;而用计数器是否减到小于1的值
        ;防止父进程子进程同时能唤醒，父进程先回去了,
        ;子进程计数器为-1而无法后续被唤醒

        ;优先苏醒父进程
        cmp word [is_father_sleeping], 0
        je clock_change_to_father ;

        ;其次苏醒子进程
        cmp word [is_child_sleeping], 0
        je clock_change_to_child  ;

        jmp int_exit

    ;时钟中断返回(含EOF)
    int_exit:
        mov al,20h
        out 20h,al
        iret
```

限于篇幅，时钟中断中跳转部分的代码仅展示从时钟中断跳转到父进程的代码，跳转到子进程的代码完全类似，代码如下:

```
;sp的值已经在刚进入时钟中断的时候更新给对应的进程了
clock_change_to_father:
    mov word [current_state], 1
    mov sp, [father_stack]
```

```
        jmp int_exit
```

至此，已经实现了 clock_handler。由于逻辑比较复杂，所以我对照整理好的流程图仔细检查了流程，要注意不能漏掉任何必需的步骤，需要做到非常仔细。

5.6写入磁盘，查看运行效果

在 wsl2 中利用 make 命令运行 makefile，进行各文件的编译，和最终虚拟机的运行。实验结果视频四、结果展示视频在开头已经展示，下面再展示一张图片：

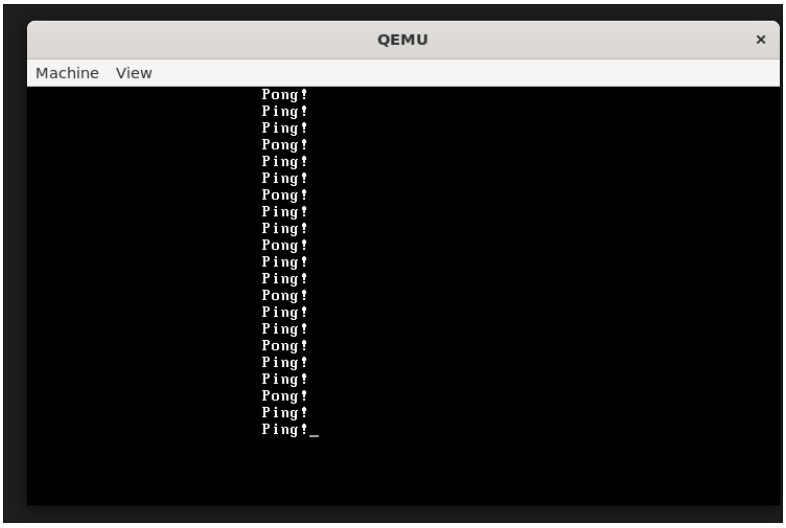

图 2: 运行结果

运行结果完全符合预期，输出结果 “pong + ping + 停顿 1s + ping + 停顿 1s” 的无限循环。

5.7进行一些优化

5.7.1 优化条件跳转语句

这次实验的条件跳转语句非常多，再加上刚写完复杂的逻辑代码比较混乱，所以为了代码的美观，我将条件跳转语句以此修改，加上标签，尽量保持了跳转的 “对称性”，例如从父进程跳转到子进程的代码应当和从子进程跳转到父进程的代码相对称。这样进一步增加了代码的可读性。

5.7.2 优化整个代码结构

代码的结构一开始比较混乱，标签各处都是、位置混乱，所以我重新编排了代码的标签位置，按照严格的逻辑顺序编排，代码结构如下：

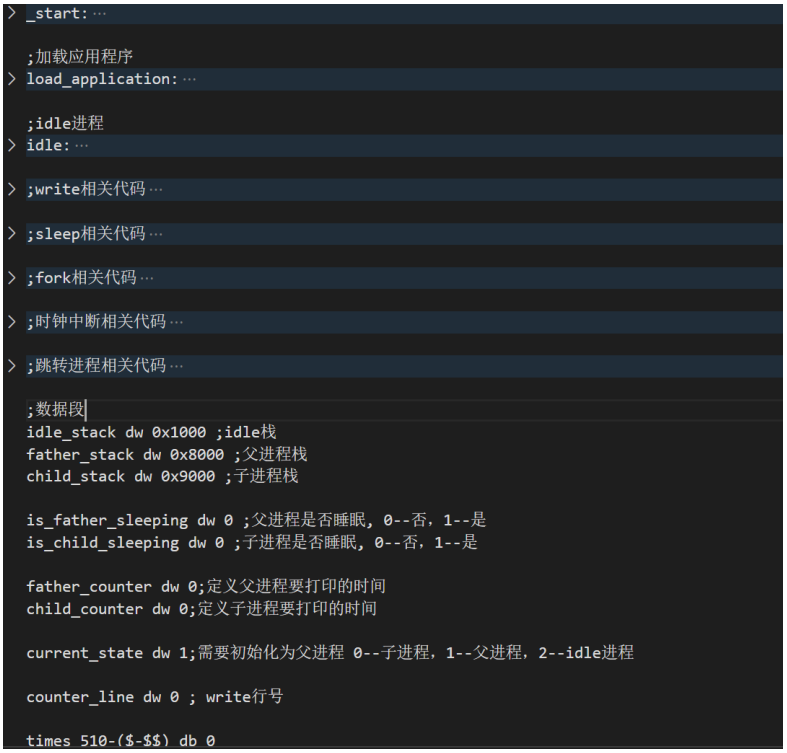

图 3: 代码结构

更小的代码结构如下 (以跳转部分代码为例):

```
;跳转进程相关代码
    ;sleep中的跳转
    ;注意修改是否在睡眠状态这一步要放在后面执行，防止时钟中断的干扰
        father_change_to_child:···

        child_change_to_father:···

        father_change_to_idle:···

        child_change_to_idle:···

        change_to_idle:···

    ;clock中的跳转
    ;sp的值已经在刚进入时钟中断的时候更新给对应的进程了
        clock_change_to_father:···

        clock_change_to_child:···
```

图 4: mini 代码结构

六、各种尝试、失败、debug 过程

这部分是体现作业是自己做的的核心，这次作业，我遇到了非常非常多的问题  , 我解决了绝大部分，才得以得到上一部分展示的成功，接下来我将详细展示。

6.1 short or int——16 位和 32 位代码没有注意区分!

由于 fork() 在 kernel.asm 中的实现需要预先修改子进程返回的值为 0，而这就需要访问这个位置的参数。一开始我的 myos.h 中的 fork 返回值不小心写成 int 了，这就与我 kernel 中的实现相不匹配。这个 bug de 了很久，但是修改起来简单，只需要在头文件中定义成 short 类型返回即可。

6.2 始终是父进程在运行!

在编写 kernel.asm 中 clock_handler 的代码的时候，我犯了一个错误，本意想判断如果是父进程或者子进程进来的时钟中断，则不能跳转，但是写成了判断父进程是否在睡眠，在睡眠就直接返回了，导致子进程无法执行到。发现这个 bug 后，我增加了如下代码：

```
;首先计算以下进中断的时候是否有没在sleep的进程;
```

```
;如果有，cx会小于2，方便后面处理
mov cx, [is_father_sleeping]
mov dx, [is_child_sleeping]
add cx, dx

.....(省略一段代码)

;在要判断跳转的时候再利用cx这个值
;cx小于2代表子进程或者父进程至少有一个在运行,不做跳转
cmp cx, 2
jl int_exit
```

6.3 fork() 中复制数据后忘了再初始化栈！

复制过程中 di 改变了，需要重新初始化 di 成栈顶才行，一开始我忘了。

6.4 超字节超字节超字节！

尽管已经有了预防，已经在做 PA5 之前将 PA4 的代码狠狠简化了，但是在实现 PA5 的过程中还是不断地超过 512 字节。这也给本次实验带来了很多困扰。当然解决方法和简化 PA4 的方法差不多，这里不再赘述。

6.5 忘了 iret！

优化 PA4 代码的时候，我将清屏的代码进行了封装，但是在中断调用清屏的时候需要 iret，我却没有 iret，导致 bug，这也 de 了很久。

6.6 其他尝试：不用数据段 current_state 储存状态能不能实现

一开始我没有使用 current_state 来实现，我预期的是直接通过访问栈对应位置的参数来知晓是谁在运行，但是可惜实现过程 bug 累累，不是 bugA 出现就是 bugB 出现，最终放弃，还是采用内存中数据段的方式来清晰地实现。

6.7 其他问题

都是一些比较明显的 bug，但是由于心急以及代码本身比较复杂，犯了不少。

七、新增要求：利用段地址实现

注意：此部分以前内容都是实现的旧要求，此部分实现新增要求。 以上实现不能满足最新应用程序的要求，应用程序如下：

```
#include "myos.h"

static int t;

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

因此，我需要做出修改。具体来说，我的修改分为以下三个部分:

7.1 减少字节

因为我字节本来就要满了，所以必须得删一些。我把所有能改的 16 寄存器改成 8 位的寄存器，但又因此犯了一个错误：有些地方的 “mov word” 没有对应改成 “mov byte”。同

时，我新代码不再处理打印打到屏幕底端的 bug 省去一个变量和相关操作。另外，我利用 jmp 跳转应用程序，iret 方式在 PA4 中已经实现，这里为了省字节就退而求其次。

7.2 fork 实现成将数据复制到另一个段地址上

修改后的 fork 如下：

```
;fork相关代码
;fork中断处理,段地址从0x0000:0x8000到0x1000:0x8000
fork_handler:
        mov [child_stack],sp ;保存子进程栈指针

        ;切到子进程段进行复制
        mov word es, [child_ds]

        mov di, sp
        mov si, sp
        mov cx, 0x300
        cld
        rep movsb

        ;es保存为父进程段，此后不再改变，可以被利用
        xor ax, ax
        mov es, ax

        ;修改子进程段地址的内容
        mov word ds, [child_ds]

        ;重新更新di指向sp
        mov di, sp
        ;修改子进程栈中fork的返回值ax,子进程返回0，short类型
        mov [di+6], ax

        ;修改回段地址为0x0000
        mov ds, ax

        ;fork返回,先返回父进程，在api中已经将父进程的ax保护为1
```

```
        ;而子进程ax返回0
        iret
```

终究还是采用了助教的方式，注意以上代码，es 最后是能够保存内核和父进程的段地址的，这方便内核 write 和 sleep 的处理。

7.3 简要修改内核代码

1.write 的修改较为简单，只需要利用 [es: 内核中定义的全局变量] 的方式来获取全局变量就可以了。而 clock 和 sleep 需要根据子进程还是父进程用不同的 ds(之前保存在两个变量中)。
具体来说，子进程睡眠时将 ds 重置为 0，代码如下：

```
xor ax,ax
mov ds,ax
mov ss,ax
```

2. 进时钟中断需要先把 ds 改成 0，然后若是子进程再改为 0x1000，这边函数用的比较多，全部展示比较混乱，略去不看。
3. 跳转子进程的时候将 ds 改为子进程的，代码如下：

```
father_change_to_child:
        mov byte [current_state], 0
        mov [father_stack], sp
        mov sp, [child_stack]
        mov byte [is_father_sleeping], 1

        ;段地址换成子进程的
        mov ss, [child_ds]
        mov ds, [child_ds]
        sti    ;这里的cli与sleep_handler中的cli相互对应
        jmp int_exit

clock_change_to_child:
        mov byte [current_state], 0
        mov sp, [child_stack]

        ;段地址换成子进程的
```

```
        mov ss, [child_ds]
        mov ds, [child_ds]
        jmp int_exit
```

经过如上修改，已经能够实现应用程序。接下来是新增内容的总结部分：

7.4 新增部分总结

实现这部分代码的时候我也犯了一些错误：
1. 因为要省字节，我把能省的寄存器都从 16 位改成 8 位了，但是有几处忘了把 mov word 修改成 mov byte；
2.fork 里面 ds 改成子进程后最后忘了改回来；
3. 进 clock 我过早把 ds 改成子进程了，后面还用了内核里的全局变量也没加 es 就用错了。
通过解决这些问题，我理解了助教所说的利用段地址的实现方式。

段地址：在分段内存管理中，内存被分成不同的段，每个段有自己的基地址和长度。每个段可以用来存储不同类型的数据，如代码段、数据段和堆栈段。

fork()：当一个进程调用 fork() 时，操作系统创建一个新进程（子进程），并复制父进程的地址空间，使子进程成为父进程的副本。

很佩服早期实现操作的人们，他们就是不断地找到问题、不断地修改和进步，才实现了操作系统。被新要求这么一折腾，我也体会了其中的痛楚与伟大。

另外，此部分的实现与同学交流后实现，在此致谢。

八、总结与反思

- 理论好理解不代表实验好做，实验需要非常多的细节，有了上次的经验，我学会了先理清流程，先画出流程图，思路明确了再实现，这才得以比较顺利地实现
- STFW,RTFM 是计算机大类学生必备的素养，知识需要主动检索和学习
- 通过本次实验，我亲手实现了 fork()，对 fork() 调用有了更深的理解。在调试各种 bug 的过程中，我更加坚信，这两句话永远是对的：1. 机器永远是对的 2. 未经检验的代码永远是错的——蒋炎岩

九、其它参考文献

除了正文给出的参考文献，我参考的文献还有:

fork () 函数的底层实现原理

jmp 系列跳转指令总结
fork 原理–Linux 的底层实现
嵌入式软件之链接脚本.ld
makefile 介绍