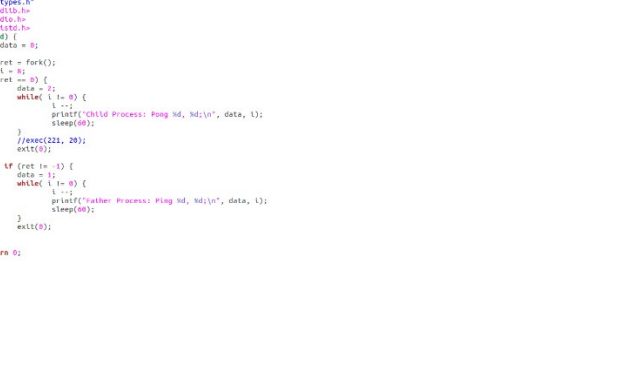


一. exercise1: 请把上面的程序, 用 gcc 编译, 在你的 Linux 上面运行, 看看真实结果是啥 (有一些细节需要改, 请根据实际情况进行更改)。

首先修改一下 app 中 main.c 的代码，使它通过编译。(修改的部分就不概述了，图片上都显示了)



The screenshot shows a Windows desktop with a taskbar at the bottom containing icons for File Explorer, Microsoft Edge, and Visual Studio Code. The Visual Studio Code window is open, displaying a C program in a file named `main.c`. The program is a simple ping utility that forks a child process to send ICMP echo requests. The code is as follows:

```
#include <stdio.h>
#include <types.h>
#include <stdlib.h>
#include <unistd.h>
#include <unistd.h>

int main(int argc) {
    int data = 0;

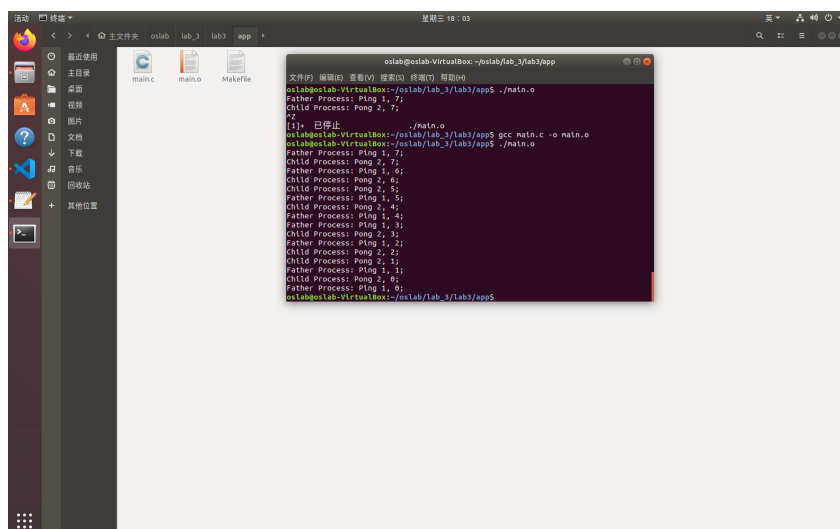
    int ret = fork();
    if (ret == 0) {
        data = 1;
        while(1) {
            //exec(221, 20);
            printf("Child Process: Ping 8d, 8d\n", data, 1);
            sleep(40);
        }
    }
    else if (ret != 0) {
        data = 1;
        while(1) {
            //printf("Father Process: Ping 8d, 8d\n", data, 1);
            sleep(40);
        }
    }

    return 0;
}
```

The terminal window at the bottom shows the program running successfully. The output is:

```
Child Process: Ping 8d, 8d\n
Father Process: Ping 8d, 8d\n
```

然后运行这个程序，运行结果如下，与预期相符，两个进程不共享 i，是在不同的地址空间中的。



二. exercise2: 请简单说说, 如果我们想做虚拟内存管理, 可以如何进行设计 (比如哪些数据结构, 如何管理内存)?

答:我们采用分页机制,可以合理的解决多进程内存划分的问题,好像每个进程都独占了 4GB 的内存空间。



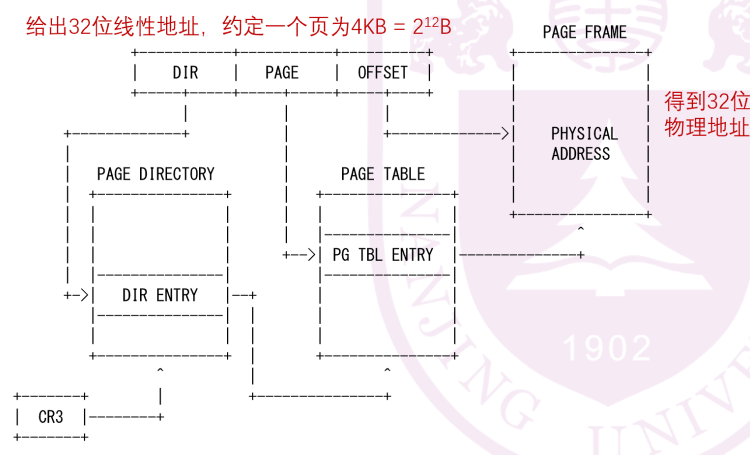
采用上图的方式将虚拟空间和物理空间划分为 4KB 一个单元；每个进程维护一个页表用来表示自己的虚拟空间中的一个单元被映射到内存中的哪个单元, 那些没有被映射的单元暂时装在磁盘中, 需要的时候再从磁盘装入主存, 并且修改页表中的映射关系。

虚拟页号	在不在主存	对应哪个物理页
0x0	在	0x100
0x1	在	0x30
0x2	不在	N/A
0x3	不在	N/A
0x4	不在	N/A
0x5	在	0x1234
...	...	...

这张图是页表的大概形式, 每个进程都有一个。

通过线性地址, 我们去页表中查询对应的页表项, 得到页框号, 再结合我们一开始有的页内地址合起来就是我们需要的物理地址了。

又因为一个页表 4MB, 在内存中找到连续的 4MB 的数组空间不容易, 因此我们将一个页表拆分为两级页表



采用图中所示的方法把线性地址转化为物理地址 (即通过 CR3 寄存器找到页目录表的基地址, 然后通过线性地址对应的部分找到页目录项, 再通过页目录项找到对应页表的基地址, 然后再找出对应页表项和对应的物理地址)

challenge1: 请发挥你的编码能力, 在实验框架代码上设计一款虚拟内存管理机制。并说明你的设计 (可以和上面的 exercise 结合, 此题是 bonus, 不写不扣分~)。(没做)

三 . exercise3: 我们考虑这样一个问题: 假如我们的系统中预留了 100 个进程, 系统中运行了 50 个进程, 其中某些结束了运行。这时我们又有一个进程想要开始运行 (即需要分配给它一个空闲的 PCB), 那么如何能够以  $O(1)$  的时间和  $O(n)$  的空间快速地找到这样一个空闲 PCB 呢?

答: 再另外开一个  $O(n)$  的 vector S, 存放空闲的 pcb 对应的 pid 号, 初始的时候, 这个 vector S 包含了 1-100; 后来每次分配出去一个 pcb 就把对应的 S 中的项 pop 出去; 每次一个 pcb 归还的时候, 就再添加进来即可; 这样每次寻找空闲 pcb 的时候直接取 S 中的第一个项就可以了, 是  $O(1)$  的复杂度

四 . exercise4: 请你说说, 为什么不同用户进程需要对应不同的内核堆栈?

答: 因为每一个进程都有自己的现场信息, 以这个 P1, P2 互相切换的例子为例: 如果 P1

与 P2 都采用同一个 kernel stack，那么首先 P2 的用户态现场信息已经在最开始的时候被保存在了这个 kernel stack 中，在 P1 的时间片用完，需要转到 P2 去执行的时候，需要获得一开始保存的 P2 的现场信息的时候，会发现当前栈指针对应的是 P1 的用户态的现场信息，无法得到 P2 的现场信息，如果 pop 掉 P1 的现场信息，那么会丧失 P1 的信息，也不妥；所以必须采用不同的内核栈。

五 . exercise5: stackTop 有什么用？为什么一些地方要取地址赋值给 stackTop？

答：是 pcb[current]这个进程对应的内核栈的栈顶指针

以手册中时间中断程序进行进程切换的例子为例；P1 首先借助 TSS 中的 esp0 找到自己的内核堆栈，并且将 P1 的 stackframe 压入栈中。然后在切换到 P2 的时候，接收到一个 pid 为 2 的 current 值，通过这个找到 P2 的内核堆栈的栈顶指针，然后再 pop 出 P2 的信息，iret 到 P2 的用户进程进行执行。

```
// user process

pcb[1].stackTop = (uint32_t)&(pcb[1].regs);

pcb[1].state = STATE_RUNNABLE;

pcb[1].timeCount = 0;
```

这里将 regs 的地址赋值给 stacktop 的目的是，这里 regs 相当于是当前进程用户态的信息，我将它需要保存到内核栈中，以便于后续从内核态转到用户态开始执行，所以这时候，我们需要把 stacktop 给挪到下面去（也就是挪到这些现场信息的下方，以便于找到这些现场信息；说的通俗一点就是把这些我们主动设定的用户态信息压栈）

自己的一个问题：

```
tss.esp0 = (uint32_t)&(pcb[current].stackTop);
```

为啥还要加一个取地址符号？（感觉这里手册写错了）

答：不是的，一开始我对于 esp0 的理解不正确；每个进程的 esp0 的值是固定的，就是这个进程 stacktop 指针的地址；在执行一个进程的时候，我们可以通过 esp0 来取出这个进程内核栈对应的栈顶指针的值，然后就可以将 esp 的值置为 stacktop，转到内核态进行执行。

六 . exercise6: 请说说在中断嵌套发生时，系统是如何运行的？

答：如果发生中断嵌套我们可以分两种情况：1. 不需要切换进程，这个时候我们只需要在内核栈里面保存现场信息，然后跳到对应的中断处理程序进行处理，然后中断返回的时候 pop 出现场信息即可（这时候其实不用把 sf 保存到 pcb[current].stacktop 中）2. 如果比如说发生了时钟中断，又或者在键盘中断的时候发生了时钟中断的嵌套，这个时候我们需要用到进程切换的函数了。这个时候按照手册上的汇编指令（如图）：

```

tmpStackTop = pcb[current].stackTop;
pcb[current].stackTop = pcb[current].prevStackTop;
tss.esp0 = (uint32_t)&(pcb[current].stackTop);
asm volatile("movl %0, %%esp:::m"(tmpStackTop)); // switch kernel stack
asm volatile("popl %gs");
asm volatile("popl %fs");
asm volatile("popl %es");
asm volatile("popl %ds");
asm volatile("popal");
asm volatile("addl $8, %esp");
asm volatile("iret");

```

我们是需要用到 stacktop 这个指针的值的，因此这个时候，我们需要在 irq\_handle 函数中用 sf 来保存 stacktop 的值，否则再切换回这个进程的时候，sf 的信息就丢失了。因为我们切换回这个进程的时候，我们是将这个进程的 stacktop 的值赋值给 esp 指针，而由于我们没有保存 sf 到 stacktop 中，因此我们直接忽视了这个 sf 的信息，会发生错误。

七 . exercise7: 那么，线程为什么要以函数为粒度来执行？（想一想，更大的粒度是.....，更小的粒度是.....）

答：考虑更大的粒度：那就是一个（文件）程序了。如果线程是以一个个程序为粒度的话，那么肯定是需要程序与程序之间共享诸如全局变量这种资源的，那么也就是说两个线程之间就不可能共享全局变量了——就会演变成进程的概念。

考虑更小的粒度：那就是一条条汇编指令了。如果是这样就完全没必要划分出线程这个概念了，指令的集合不就是函数吗，我们已经存在这个概念了。函数独有的资源也不包括栈区，pc 和寄存器了，基本上是没有的，所以没必要搞到这么小的粒度。

challenge2: 请说说内核级线程库中的 pthread\_create 是如何实现即可。

答：上网百度了一下源码：

```

1 int __pthread_create_2_1 (newthread, attr, start_routine, arg)
2     pthread_t *newthread;
3     const pthread_attr_t *attr;
4     void *(*start_routine) (void *);
5     void *arg;
6 {
7     STACK_VARIABLES;
8
9     const struct pthread_attr *iattr = (struct pthread_attr *) attr;
10    if (!iattr || iattr == NULL)
11        iattr = &default_attr;
12
13    struct pthread *pd = NULL;
14    ALLOCATE_STACK (iattr, &pd);
15
16    pd->header.self = pd;
17    pd->header.tcb = pd;
18
19    pd->start_routine = start_routine;
20    pd->arg = arg;

```

主要在于 ALLOCATE\_STACK 宏，它用来用来分配线程栈，并在栈底创建 pthread 结构并初始化。然后设置执行线程函数的地址 start\_routine 和参数地址 arg。

challenge3: 你是否能够在框架代码中实现一个简易内核级线程库，只需要实现 create, destroy 函数即可，并仍然通过时钟中断进行调度，并编写简易程序测试。不写不扣分，写出来本次实验直接满分。

答：没做

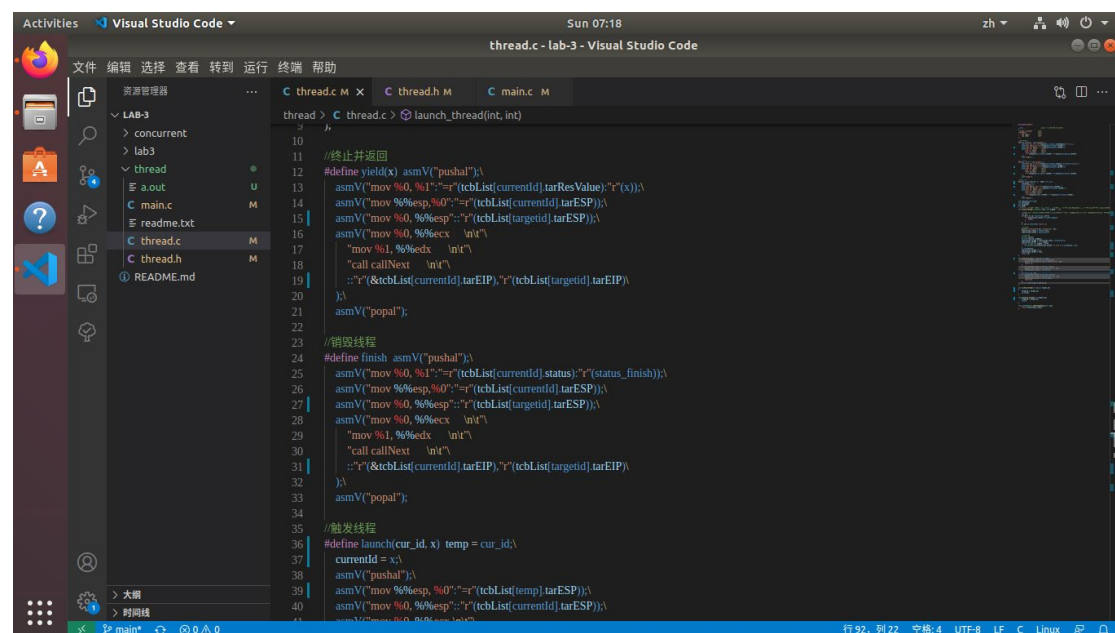
challenge4: (不写不扣分, 写了加分)

1. 在 create\_thread 函数里面, 我们要用线性时间搜索空闲 tcb, 你是否有更好的办法让它更快进行线程创建?

答: 我觉得可以采用空间换时间的方法, 比如说把所有空闲的 tcb 的 id 号都装到一个 vector 中, 每次要创建的时候就直接用 o(1)时间去里面取, 每次销毁一个进程的时候就从里面 pop 出对应的 id 号码就可以了。初始值设定为所有 id 号都是空闲的。(也可以用链表, 不用 vector, 这样每次申请内存不需要连起来的一大块区域, 比较方便)

2. 我们的这个线程库, 父子线程之间关联度太大, 有没有可能进行修改, 并实现一个调度器, 进行线程间自由切换? (这时 tcb 的结构可能需要更改, 一些函数功能也可以变化, 你可以说说思路, 也可以编码实现)

答: 我认为 tcb 里没有必要保存父线程的 srceip 和 srcesp; 因为这个就代表我一个线程销毁或者 yield 的时候, 我必须回到父进程去执行, 父子线程关联度太大了; 我们可以在这个结构体中只保留自己的当前的 eip 和 esp 而不保存父线程的, 然后在 yield 和 destroy 函数中可以多加入一个参数 target\_id, 表示的是我们暂停或者消灭这个线程之后希望跳转到哪个 id 号对应的线程执行。



```
thread.c - lab-3 - Visual Studio Code
10 // 终止并返回
11 #define yield(x) asmV("pushal");
12 asmV("mov %0, %1": "=r"(tcbList[currentId].tarResValue): "r"(x));
13 asmV("mov %%esp, %0": "=r"(tcbList[currentId].tarESP));
14 asmV("mov %0, %%esp": "=r"(tcbList[targetId].tarESP));
15 asmV("mov %0, %%ecx": "=r"(x));
16 "mov %1, %%edx": "=r"(x));
17 "call callNext": "=r"(x));
18 "r"(tcbList[currentId].tarEIP), "r"(tcbList[targetId].tarEIP);
19 );
20 asmV("popal");
21 // 销毁线程
22 #define finish asmV("pushal");
23 asmV("mov %0, %1": "=r"(tcbList[currentId].status): "r"(status_finish));
24 asmV("mov %%esp, %0": "=r"(tcbList[currentId].tarESP));
25 asmV("mov %0, %%esp": "=r"(tcbList[targetId].tarESP));
26 asmV("mov %0, %%ecx": "=r"(x));
27 "mov %1, %%edx": "=r"(x));
28 "call callNext": "=r"(x));
29 "r"(tcbList[currentId].tarEIP), "r"(tcbList[targetId].tarEIP);
30 );
31 asmV("popal");
32 // 触发线程
33 #define launch(cur_id, x) temp = cur_id;
34 currentId = x;
35 asmV("pushal");
36 asmV("mov %%esp, %0": "=r"(tcbList[temp].tarESP));
37 asmV("mov %0, %%esp": "=r"(tcbList[currentId].tarESP));
38 asmV("mov %0, %%ecx": "=r"(x));
39 "mov %1, %%edx": "=r"(x));
40 "call callNext": "=r"(x));
41 "r"(tcbList[currentId].tarEIP), "r"(tcbList[targetId].tarEIP);
42 );
```

这样的话这里的 finish 和 yield 宏就可以把 srceip 和 srcesp 改为 tcbList[target\_id].tarEip 与 tcbList[target\_id].tarEsp, 这样就可以通过 callnext 函数实现保存当前进程的 eip 并且转到对应线程去执行的功能, 降低了父子线程的依赖。

3. 修改这个线程库或者自由设计一个线程库。

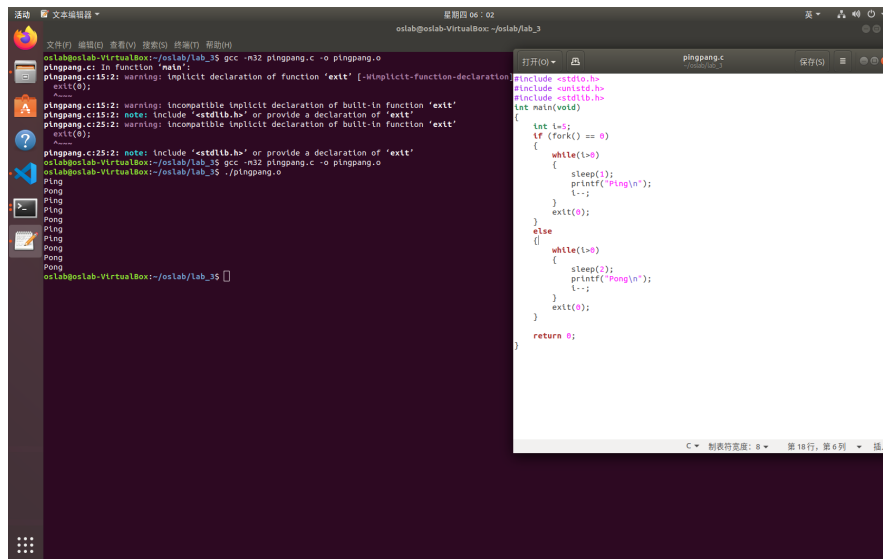
答: 经过改动之后, 实现了我上面所说的进程库, 具体代码见我上交的 thread 文件夹, 跑出来的结果与助教 gg 写的代码是一样的。这里需要注意的是, 我改完其他的之后, 会出现一些问题, 见下图。





时候，这个程序就可以在自己的栈中访问到需要的参数了。

八 . exercise8: 请用 fork, sleep, exit 自行编写一些并发程序，运行一下，贴上你的截图。  
(自行理解，不用解释含义，帮助理解这三个函数)



```
oslab@oslab-VirtualBox:~/oslab/lab_35 gcc -m32 pingpong.c -o pingpong.o
pingpong.c: in function 'main':
pingpong.c:15:2: warning: implicit declaration of function 'exit' [-Wimplicit-function-declaration]
    exit(0);
    ^
pingpong.c:15:2: warning: incompatible implicit declaration of built-in function 'exit'
pingpong.c:15:2: note: include '<stdlib.h>' or provide a declaration of 'exit'
pingpong.c:25:2: warning: incompatible implicit declaration of built-in function 'exit'
pingpong.c:25:2: note: include '<stdlib.h>' or provide a declaration of 'exit'
oslab@oslab-VirtualBox:~/oslab/lab_35 gcc -m32 pingpong.c -o pingpong.o
oslab@oslab-VirtualBox:~/oslab/lab_35 ./pingpong.o
Ping
Pong
Ping
Pong
Ping
Pong
Ping
Pong
oslab@oslab-VirtualBox:~/oslab/lab_35
```

自己提出的另外俩问题:

Regs.eax 有啥用?? --是不是就是刚创建的时候初始化用户态进程的一些信息? 后面就用不到了?

答: 后面其实也用的到, 也就是这个 regs 和我们开辟的 stack 数组合起来变成了我们这个进程的内核栈了

Fork 时候的 pid 没有初始化?

答: (无所谓的)

九 . exercise9: 请问, 我使用 loadelf 把程序装载到当前用户程序的地址空间, 那不会把我 loadelf 函数的代码覆盖掉吗? (很傻的问题, 但是容易产生疑惑)

答: loadelf 的代码在内核空间, 而程序是被加载到用户程序的地址空间的, 两个空间不重叠。

补充:

做实验的时候犯下了两个错误:

一个是内存分配的时候 1 号进程对应的是 0x200000 而不是 0x300000

第二个是 sleep 和 exit 的时候应该把 time count 置为最大, 否则没法被调下来!