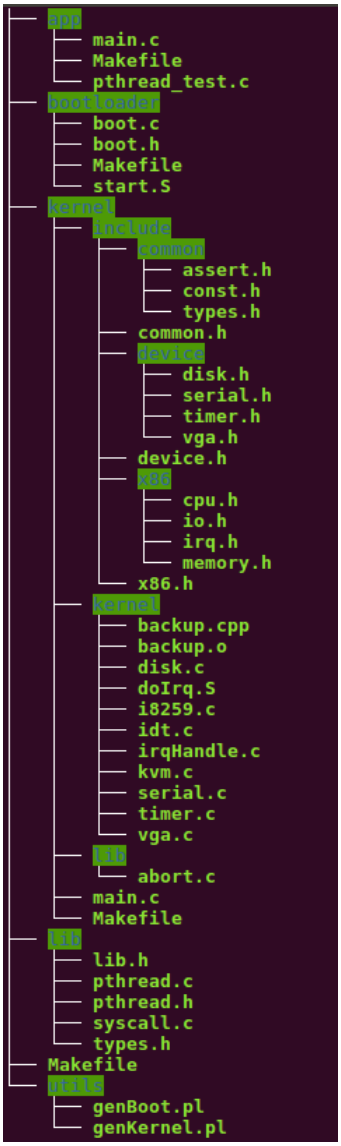
Lab3 进线程切换

一、进程切换流程

代码框架



Bootloader从实模式进入保护模式，加载内核至内存，并跳转执行。（对应bootloader文件夹的内容）

内核初始化IDT，初始化GDT，初始化TSS；初始化串口，初始化8259A等硬件设备。（对应kernel/kernel下的内核初始化部分）

启动时钟源，开启时钟中断处理。（在timer.c中开启）

加载用户程序至内存。（用户程序的加载对应kvm.c的loadumain，会kvm.c的initProc会创建第一个用户进程，其对应的入口就是函数loadumain）

初始化内核IDLE线程的进程控制块（Process Control Block），初始化用户程序的进程控制块

需要进程切换时，切换至用户程序的内核堆栈，转到另一个进程的内核栈，弹出用户程序的现场信息，返回用户态执行新的用户程序。

具体的进程切换流程、线程切换流程会在后面详细叙述。

二、框架代码解读

框架代码类似实验二，需要填写的内容在kernel/kernel/irqHandle.c的系统调用处理函数中和lib/pthread.c的线程相关函数中。

段表

在kvm.c的initSeg()函数中已经预先设置好了各进程的对应的段表，因此无需再次手动设置。cs寄存器是代码段寄存器，其他段寄存器包括数据段寄存器、堆栈段寄存器，一个进程中，cs寄存器对应的段描述符和其他寄存器对应的段描述符不同。

中断

在kernel/kernel/idt.c的initIdt()函数中设置了中断描述符，每个描述符对应着一个中断处理函数的入口地址。

执行int指令引发中断时，标志寄存器、CS:EIP寄存器会（硬件）入栈，然后根据调用号在中断描述符表中找到相应描述符，并将对应数据放入CS:EIP中，跳转到处理函数。

系统调用int $0x80$和时钟中断int0x20

每当需要系统调用时，封装的函数会跳转到lib/syscall.c，syscall()会保存现场（标志寄存器等以外的寄存器）然后执行int $0x80进行系统调用。$int$0x80对应的函数是irqSyscall，在kernel/kernel/doIrq.S中定义，这个函数会压入错误号（0）和调用号（0x80）然后跳转到asmDoIrq中。

由initIdt()知int $0x20对应时钟中断irqTimer（同样在kernel/kernel/doIrq.S中定义）。它会压入错误号（0）和调用号（0x20）然后进入asmDoIrq。

asmDoIrq会把所有通用寄存器（pusha指令）、段寄存器（这些寄存器保存了进程状态）压入核心栈，然后执行中断处理函数irqHandle。irqHanle会根据调用号（其参数sf->eax）来选择相关的服务例程执行。

参数sf(栈帧)的结构，恰好和现在栈中push进去的进程状态相同。

```
struct StackFrame {
    uint32_t gs, fs, es, ds;//手动push的段寄存器
    uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;//pusha指令
    uint32_t irq, error;//手动push进去的错误号和中断调用号
    uint32_t eip, cs, eflags, esp, ss;//int指令时，硬件执行
};
```

在中断处理完毕时，会手动pop段寄存器、通用寄存器（popa指令），最后执行iret指令回到用户态（当使用iret指令返回到一个不同的保护级别时，iret不仅会从堆栈弹出CS:EIP和eflags，还会弹出堆栈段选择子及堆栈指针分别到ss与esp寄存器）。

TSS

TSS在任务切换过程中起着重要作用，通过它实现任务的挂起和恢复。Linux2.4以后，全部进程使用一个TSS。中断后，需要切换到核心态（特权级切换，一个任务切换），首先要切换到内核栈，此时CPU会用tss的esp0去置换%esp寄存器，跳转到核心栈。

irqHandle时栈的变化

我觉得这是本次实验中最难理解的部分。进入irqHandle()函数时，会执行如下代码。

```
uint32_t tmpStackTop = pcb[current].stackTop; //1
pcb[current].prevStackTop = pcb[current].stackTop; //2
pcb[current].stackTop = (uint32_t)sf; //3
/*根据sf->eax执行相关函数*/
pcb[current].stackTop = tmpStackTop;
```

这个过程中栈的变化如下，RetAddr是irqHandle函数的返回地址，在amsDoIrq函数call指令之前把原来的esp当作参数压入栈，sf的值就是原来的esp，它指向压入栈的各个寄存器。

我认为在这里对栈内容的分析，成为了后面完整填写这个函数的关键。

三、填写完整代码

进程切换

总体来说，进程切换时的思路还是比较清楚的，但实现起来没那么容易。

是如何运行到第一个用户进程的？在kvm.c的进程块初始化内容里，我们只看见了IDLE进程、第一个用户进程的创建代码，而没有看到明显的跳转函数。在初始化函数中有一个重要的汇编指令int $0x20，它会“手动”引发时钟中断，进而执行irqHandle函数，进一步跳转到timerHandle，进行进程切换。所以，我的书写思路不是先写“看似”简单的fork指令，而是先写时钟中断的处理函数timerHandle，保证程序在运行的时候能够正确跳转到用户进程，从磁盘加载相关内容，进而继续执行。实际上，在写fork的时候，也发现了许多写timerHandle时的bug，在写其他函数的时候，又发现了fork的bug。

timerHandle

这个是由时钟中断的处理函数。思路比较简单，需要遍历一遍进程控制块，若某个进程的state为STATE_BLOCKED（本次实验中只有SLEEP调用把状态改为阻塞），则其sleeptime减1，减为0时状态改为RUNNABLE。当前运行的进程timecount加一，若达到上限，则说明消耗完一轮运行的时间片，此时需要进程切换。把current的状态改为RUNNABLE。当然如果没有运行到时间上限，可以直接return，然后经过正

常的中断返回流程，回到用户态，如果此时不return，强行执行后面的切换代码，会带来其他bug。

需要切换进程时，遍历进程控制块，找到一个除current进程以外的，状态为RUNNABLE的进程，并切换到这个进程，timecount为0。如果找不到这样的进程时，把current改为0，0号进程就是idle进程。找新的可执行线程的时候我觉得最好让i从current+1开始，虽然说从1开始也行，但是会出现刚刚消耗完时间片的进程又被重复执行的问题。

把current改为合适的值后，需要恢复现场。由上一部分的分析可以得知，stackTop的位置保存的是进程的核心栈，所以把%esp寄存器的值改为stackTop以便在执行pop和popal以及iret操作的时候能够恢复寄存器的值。这个时候stackTop要改为prevStackTop，即让stackTop回到调用前的核心栈。所以tss.esp0也要更换为这个核心栈地址，因此进程发生了变化，保证下一次任务切换的时候，esp0能够正确替换%esp。但是为什么tss的ss0不需要修改？我的想法是所有进程虽然有不同的内核栈，但是他们的ss选择的段描述符保存的应该是用户栈的偏移，而内核栈的偏移没有变，永远是ss0这个选择符（ss0在initSeg中被设置为了SEG_KDATA），但是具体位置是由stackTop这个位置决定的，所以esp0需要修改。

syscallFork

这个函数是在复制current进程。在创建完新的进程后，我并没有立即切换进程，而是继续执行当前进程。

为了创建新的进程，需要在进程控制表里找一个空闲的块（STATE_DEAD），新的进程号为i。由填写的段表内容知，新进程的cs寄存器高位应该为i*2+1，其他段寄存器高位应该为i*2+2。新进程的通用寄存器等内容完全和current相同，但是不能通过pcb[i].regs=pcb[current].regs这样赋值，需要一个一个赋值。因为新进程还为执行过，timeCount设置为0。

新进程的stackTop和prevStackTop为什么不能像创建用户进程时候那样写？我觉得是因为此时current的核心栈已经保存了一些内容，不是完全空的，所以stackTop和原始值有个偏差，而新进程的stackTop和current的stackTop的相对偏移量应该相同，因此不能直接复制，prevStackTop也是如此。

最后设置返回值，父进程（current）返回值为i，子进程（i）返回值为0。

syscallSleep

由调用syscall的过程可知，ecx保存了睡眠长度。把当前进程的状态改为阻塞（阻塞状态也可能由其他原因引起，但在本次实验中只由sleep引起）。然后修改sleeptime为sf->ecx。

接着切换到一个可以执行的进程，和timerHandle的进程切换方法一致。

syscallExit

把current进程的状态改为DEAD表示终止。接着再寻找一个可以执行的进程并切换，和timerHandle的进程切换方法一致。

小结

我对进程切换时栈的具体变化并不熟悉，因此在写程序的时候遇见了很多bug。最先遇见了无限fork的bug，检查后发现可能是由于没有显示调用exit函数引起的，因为没有exit的调用，1号进程永远也不会被置为DEAD，进程调度的时候会再一次选择1号进程，但是在执行完1号进程的所有代码后，eip的值并不确切，也不一定会重头开始执行用户程序；另一个猜想是栈的问题，所以在创建进程的时候不能直接复制stackTop的值。而似乎fork创建的新进程不会再一次回到fork前面，因为切换的时候是中断处理的流程，应该回到正确的地址才对。还遇见了强行改变tss.ss0带来的bug，在理清思路后把这句删掉了。

sleep和exit操作的时候貌似可以直接调用timerHandle，反正他们都是在切换进程。但是正常执行timerHandle的时候，当前进程都是处于RUNNING的状态，此时调用timerHandle会重复执行时钟中断的处理内容，导致各种乱跳转的bug，所以不能直接调用timerHandle，但是可以直接粘贴timerHandle后面切换的代码。

在使用putChar输出debug的时候，可以扩展这个函数，使其能够输出更长的一个字符串、数字等。

线程切换

pthread_create

线程创建类似进程的创建，需要设置eip、esp、ebp的值，不过不需要设置其他寄存器的值。eip的值设为传入函数指针，表示新线程的入口地址。stackTop可以指向stack的底端。线程的用户栈有大小限制，这也是为什么需要指定一个用户栈大小的上限MAX_STACK_SIZE。

创建线程的时候同样不需要切换线程。由于本次实验由单核CPU实现，所以线程切换需要显式的调度。

pthread_exit

把当前线程的state改为dead，表示可以被新的线程覆盖。在current线程执行完毕后，可能还会有其他被block的线程，是因为显式调用了join函数，导致必须先执行完current线程再执行其他进程，需要把joinid为current的线程的state改为runnable。然后选择一个合适的线程执行。

pthread_join

join函数需要等待thread进程结束。但是传入的时候current并不一定和thread相等。所以可以阻塞current线程，并找到一个可以执行的线程然后继续运行。故pthread_join的调用者会被阻塞，直到thread线程执行完毕调用pthread_exit之后，才会恢复当前线程。在当前线程被阻塞后，寻找一个新的可以执行的线程，并切换。

pthread_yield

个人觉得yield是线程切换中最困难的部分。

首先是需要保存当前运行线程的上下文，然后修改寄存器，恢复选择的新的线程。由于所有线程被视作一个进程，他们共享一个核心栈，所有线程之间不会出现在内核态的操作。

我的想法是使用push指令压入所有寄存器，如此以来，即便是在取线程控制块时发生了寄存器的变化（因为涉及到数据段的访问），仍然能够通过压入栈中的这些值来保存原本每个线程的上下文。选择好合适的线程后，又一次通过push指令压入新线程保存的寄存器（cont），然后使用pop指令恢复到真正的寄存器中。

最后push参数，然后修改eip跳转。然而，eip寄存器是无法直接访问的，在内联汇编中编译器会报出bad register name %eip的错误，那么怎么保存eip的值呢？我们可以间接访问。我们知道，汇编中调用其他函数的指令是call，这个指令会把函数的返回地址压入栈，ret指令会间接修改eip的值。

于是，在调用yield函数前，函数的返回地址会被放到0x4(%ebp)，所以可以把cont.eip设置为这个地址，cont.esp设置为%ebp+0x8，cont.ebp设置为old ebp（完全仿照真正汇编时栈的结构）。其实这里使用jmp指令进行跳转也是可以的，只不过需要注意下压入的参数。这样做，不用担心程序会执行到奇怪的地方引发段错误等，在跳到新的线程的时候，会直接跳到那个线程调用yield函数的后面一句，保证整个流程正确执行。

小结

写的时候会担心，线程切换的时候强行跳转，会不会改乱了返回。实际上并不会，因为每个线程在结束的时候会显式调用pthread_exit，保证能够正确回到主线程中。而且切换线程后的地址均为yield函数执行完毕的地址，所以yield的切换也不会有问题。

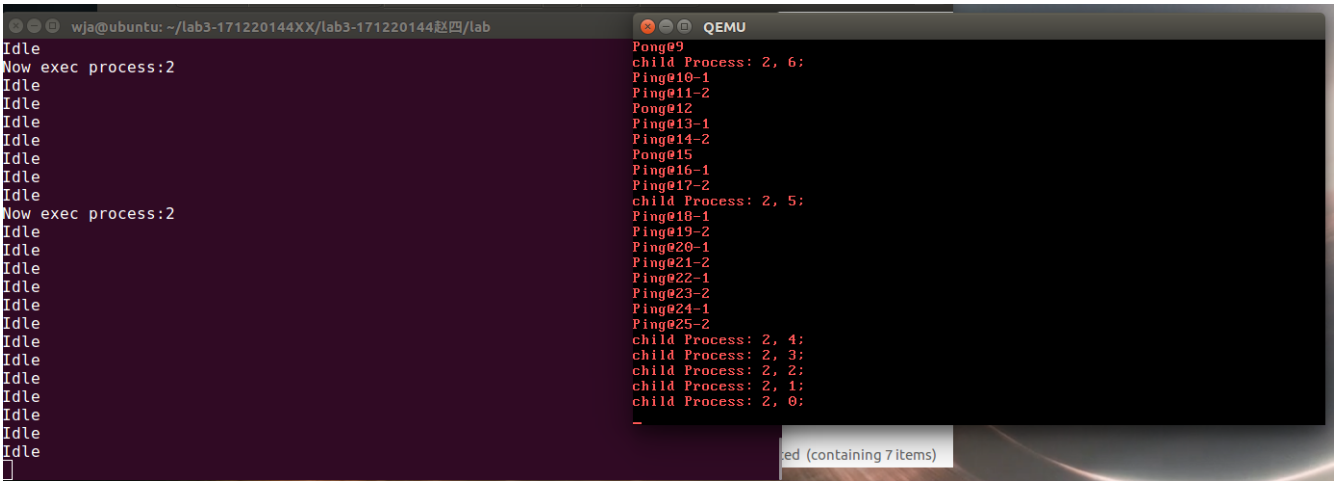
最先在切换线程的时候遇见了非常奇怪的bug，输出结果是乱码（比如每层循环的时候的i值），但是在检查原因的时候，我写了一句if(i == 2) while(1) {}，即i为2的时候就让整个线程卡住，实际情况是线程的确被卡住了，这就说明i确实为2，但是输出的i是乱码。

反汇编结果显示：

左边的编译器是gcc6.3，右边的编译器是gcc5.4。可以看到，在访问全局变量的时候，左边的编译器使用了%ebx寄存器，而右边没有。因此出错的原因可能是在线程切换的时候没有完全正确恢复寄存器的值，导致printf格式串的地址出现了偏差。也有可能是栈的内容发现了变化导致格式串地址出现了偏差。

解决办法是把gcc6.3编译器换为了gcc5.4（需要更换debian9的源然后卸载并重装gcc，操作比较麻烦，这里我强行换成了ubuntu16.04进行实验）。

运行结果


wja@ubuntu: ~/lab3-171220144XX/lab3-171220144赵四/lab
Idle
Now exec process:2
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Now exec process:2
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
Idle
QEMU
Pong@9
child Process: 2, 6;
Ping@10-1
Ping@11-2
Pong@12
Ping@13-1
Ping@14-2
Pong@15
Ping@16-1
Ping@17-2
child Process: 2, 5;
Ping@18-1
Ping@19-2
Ping@20-1
Ping@21-2
Ping@22-1
Ping@23-2
Ping@24-1
Ping@25-2
child Process: 2, 4;
child Process: 2, 3;
child Process: 2, 2;
child Process: 2, 1;
child Process: 2, 0;
ed (containing 7 items)