0slab3 进程切换实验报告

一. 实验目的

1.学习基于时间中断进行进程切换的原理

2.尝试设计纯用户态的非抢占式的线程切换

3.加深对汇编指令、栈帧构造等计算机底层知识的理解

二. 实验设计

（一）程序的流程是:

（1）Bootloader 从实模式进入保护模式，加载内核至内存，并跳转执行

（2）内核初始化 IDT ，初始化 GDT ，初始化 TSS ，初始化串口，初始化 8259A，...

（3）启动时钟源

（4）加载用戶程序至内存

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

（6）切换至用戶程序的内核堆栈，弹出用戶程序的现场信息，返回用戶态执行用戶程序

（二）需要完成的任务:

（1）irqHandle.c 的 timerHandle,syscallFork,syscallSleep,syscallExit 实现内核态进程切换机制

（2）pthread.c 中 pthread_create、pthread_join、pthread_exit、pthread_yield 实现用户态线程切换

（三）代码思路

1.进程切换思路

（1）timerHandle

每次时钟中断来临需要:

*把每个阻塞的进程阻塞时间减 1，减到 0 则转换为就绪态

*把当前进程的时间片减 1，没减到 0 则直接返回，不进行调度，否则按如下方法进程切换

*当前进程设置为就绪态，重设时间片，寻找新的就绪的进程来占用 cpu，设置为运行态

*选出新进程后输出 pid，，恢复用户栈（把 stackTop 和 preStackTop 交换，完成内核栈和用户栈切换）

*设置 tss 的 esp0 方便下一次用户态进入内核态

*把 esp 寄存器的值设为新进程内核的栈顶，完成内核栈切换

*弹出内核中现场信息，返回新进程的用户态

（2）syscallFork

*在进程表中找一个空位（state_dead），只考虑找到的情况

*开中断，复制父进程资源给子进程，关中断

*拷贝栈、设置状态、pid、初始化 pcb 结构体，拷贝父亲进程寄存器给子进程

```
//son->regs=father->regs;
mmcpy((uint8_t*)&(son->regs),(uint8_t*)&(father->regs),sizeof(son->regs));
```

*这里只能一个字节一个字节拷贝，否则结果不正确!

```
void mmcpy(uint8_t* dst,uint8_t* src,size_t size ){
  for(size_t i=0;i<size;i++)
    *(dst+i)=*(src+i);
}
```

*设置段寄存器，使用宏 USEL 即可，按照 pdf 上的栈区划分，内核线程占用 0-0x100000

*主进程为 0x200000 开始，34、56、78 分别为 3 个用户进程的 code 段和 data 段（数据栈区）

（3）syscallSleep

阻塞当前进程，执行进程调度（按时间中断里的方法）

（4）syscallExit

杀死当前进程，执行进程调度（按时间中断里的方法）

2.线程切换思路

（1）pthread_create

在数组中找一个空位，初始化

```
tcb[j].state=STATE_RUNNABLE;
tcb[j].joinid=-1;
tcb[j].pthArg=(uint32_t)arg;
tcb[j].pthid=j;
*thread = j;
// uint32_t eax,ebx,ecx,edi,edx,esi;
asm volatile("movl %%eax, %0":"=m"(tcb[j].cont.eax));
asm volatile("movl %%ebx, %0":"=m"(tcb[j].cont.ebx));
asm volatile("movl %%ecx, %0":"=m"(tcb[j].cont.ecx));
asm volatile("movl %%edi, %0":"=m"(tcb[j].cont.edi));
asm volatile("movl %%edx, %0":"=m"(tcb[j].cont.edx));
asm volatile("movl %%esi, %0":"=m"(tcb[j].cont.esi));

tcb[j].stackTop=(uint32_t)&(tcb[j].stack[MAX_STACK_SIZE]
tcb[j].cont.ebp=tcb[j].stackTop;
tcb[j].cont.esp=tcb[j].stackTop;
```

此时 ebp、esp 都在栈顶，然后初始化线程栈的状态

```
*(uint32_t *)(tcb[j].cont.ebp)=tcb[j].pthArg;

tcb[j].cont.eip=(uint32_t )start_routine;
tcb[j].cont.esp-=8;
*(uint32_t *)tcb[j].cont.esp=tcb[j].cont.eip;
tcb[j].cont.esp-=4;
 *(uint32_t *)tcb[j].cont.esp=tcb[j].cont.ebp;
 tcb[j].cont.ebp=tcb[j].cont.esp;
return 0;
```

设置 eip 为函数入口地址、将栈顶内容设为参数、ebp 减 8 位置设为返回地址、ebp-12 位置压入旧 ebp 值（即建立一个栈帧），问题 1：这个栈帧有什么用呢，后面就知道了

此时初始化后的新线程栈状态如下

旧 ebp->	Arg
	函数入口
esp,ebp->	旧 ebp

然后 return 0，新建的线程并不立刻切换

（2）pthread_exit

设置当前线程状态为 dead，将之前 join 自己的线程唤醒，执行线程调度 pthread_schedule（）

（3）pthread_join

*如果当前加入一个 dead 进=线程，则返回错误-1，因为 dead 线程无法 exit，那么当前进程将永远无法被唤醒

*设置当前线程为阻塞，设置被加入线程的 joinid 为当前线程

*保存当前线程现场信息（注意，这个时候保存的是 join 函数里的用户栈帧情况，而不是 ping、pong 函数的栈帧情况），问题 2：这对之后有什么影响呢？下面接着分析!

*执行线程切换

（3）pthread_yield

*阻塞当前线程，保存线程信息，同 join 函数，一样保存的并不刚好是 ping、pong 的现场!

*线程切换

（4）线程切换函数 pthread_schedule

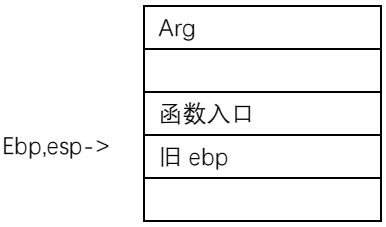
*找到一个就绪的线程，通过 tcb 中的信息恢复现场信息

*最值得一提的是，这里的现场指的是 yield、join 里面所保存的现场，仅仅恢复这个现场，esp 和 ebp 的值还没有恢复，eip 也没有设置，怎么办?

*return;语句，首先 leave 完成 esp 和 ebp 栈帧的切换，切换完后 esp 刚好指向函数现场的入口，这个入口什么时候保存的呢？要么在初始化时被设置在栈中恰好的位置上了，要么就是由 pingpong 函数自身调用了 yield 函数时，入口地址被压栈了

*所以再用 ret 的功能即可，leave 和 ret，用 return 代替，这就解决了问题 2，利用 return 返回真正的现场

*对于问题 1，create 时初始化的栈帧结构有什么用呢，我们分析一下



<Leave 后变成>

<ret 后变成>

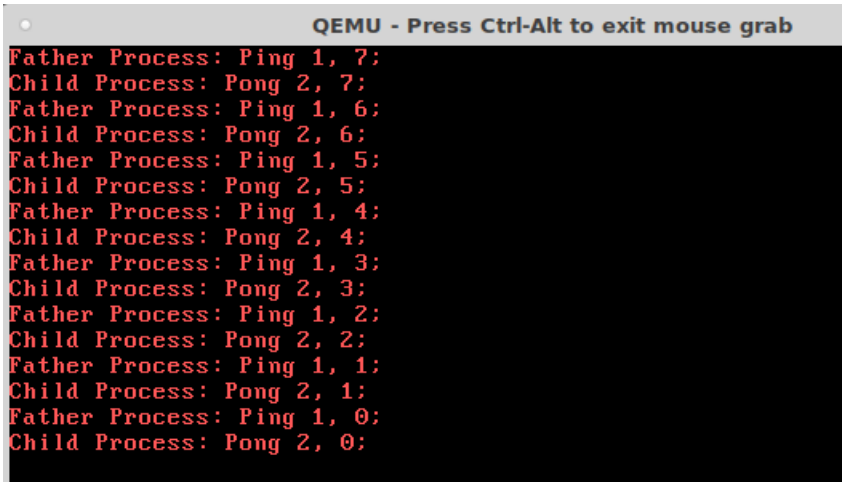
刚进入 ping、pong 时会有 push ebp 来建立栈帧

此时 arg 刚好在 ebp+8 字节的位置，即为函数调用的参数

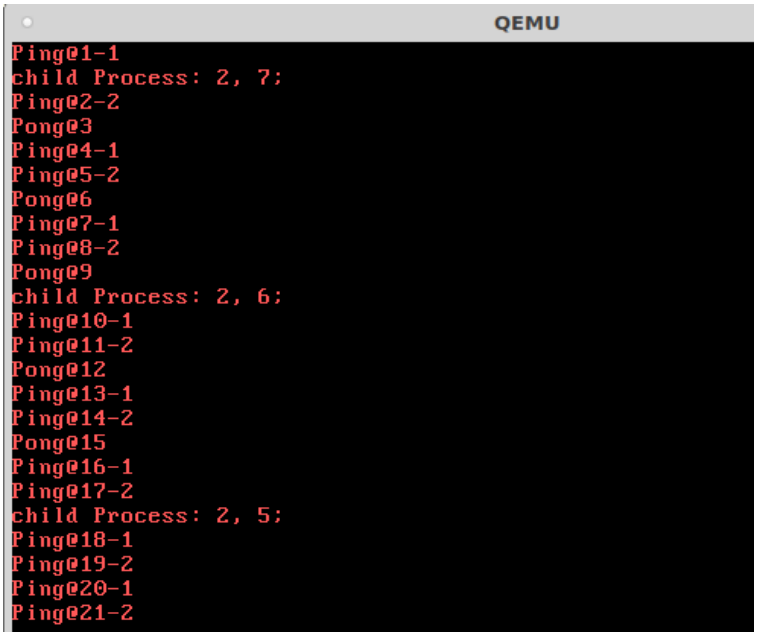
这样就解释了问题 1！线程切换到此结束

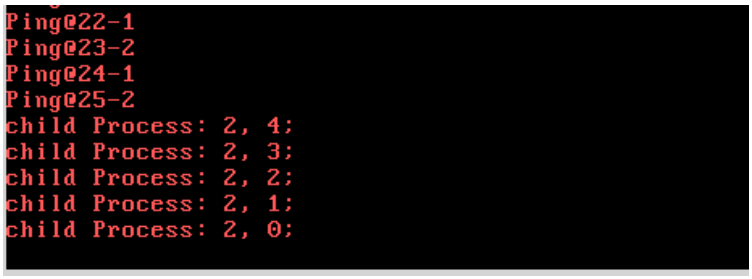
三. 实验结果

进程结果：



线程结果：





三. 实验收获

（1） 利用 objdump 查看汇编级别代码会带来意想不到的帮助，比如本实验利用 objdump 寻找函数返回的跳转地址是否符合预期

（2） 熟悉了栈的结构，加深提高了对汇编指令的理解和运用，理解了并发过程中栈是如何切换的

（3） 学会了 c 语言内嵌汇编的基本使用

四. 实验建议

暂无