Oslab3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

1. **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

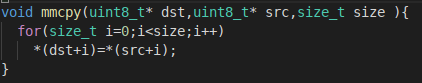
\*在进程表中找一个空位（state\_dead），只考虑找到的情况

\*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



\*这里只能一个字节一个字节拷贝，否则结果不正确！



\*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）

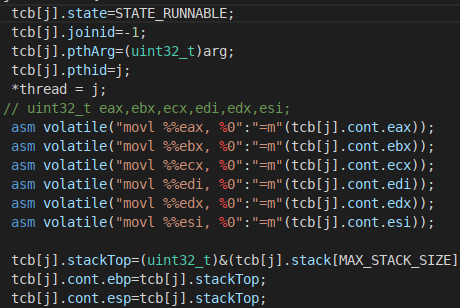
（4）syscallExit

杀死当前进程，执行进程调度（按时间中断里的方法）

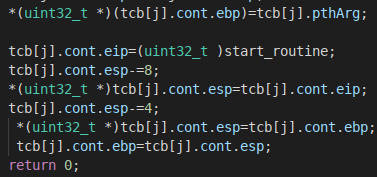
2.线程切换思路

（1）pthread\_create

在数组中找一个空位，初始化



此时ebp、esp都在栈顶，然后初始化线程栈的状态



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

|  |
| --- |
| Arg |
|  |
| 函数入口 |
| 旧ebp |
|  |

旧ebp->

esp,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

|  |
| --- |
| Arg |
|  |
| 函数入口 |
| 旧ebp |
|  |

Ebp,esp->

<Leave后变成>

|  |
| --- |
| Arg |
|  |
| 函数入口 |
| 旧ebp |
|  |

Ebp->

Esp->

<ret后变成>

|  |
| --- |
| Arg |
|  |
| 函数入口 |
| 旧ebp |
|  |

Ebp->

Esp->

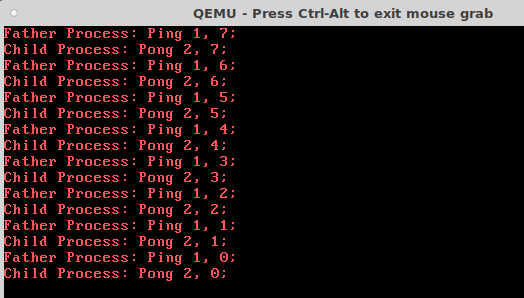
刚进入ping、pong时会有push ebp来建立栈帧

此时arg刚好在ebp+8字节的位置，即为函数调用的参数

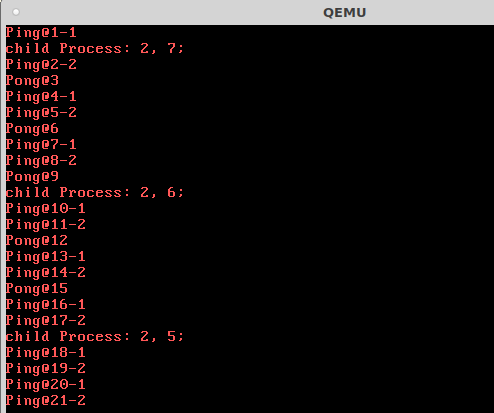
这样就解释了问题1！线程切换到此结束

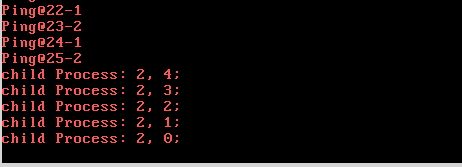
三．实验结果

进程结果：



线程结果：





1. 实验收获
2. 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
3. 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
4. 学会了c语言内嵌汇编的基本使用
5. 实验建议

暂无