## 实验

1. **线程切换之中，页表是何时切换的？页表的切换会不会影响程序 / 操作系统的运行？为什么？**

我们可以找到线程切换的代码

pub fn prepare\_next\_thread(&mut self) -> \*mut Context

调用了prepare的代码

|  |
| --- |
|  |

let parked\_frame= self.inner().context.take().unwrap();

unsafe { KERNEL\_STACK.push\_context(parked\_frame) }

不会影响操作系统的执行，

1. **设计：如果不使用 sscratch 提供内核栈，而是像原来一样，遇到中断就直接将上下文压栈，请举出（思路即可，无需代码）：**
2. 只运行一个非常善意的线程，比如 loop {}
3. 寄存器出问题，比如无法找到入口函数
4. 运行两个线程。在两个线程切换的时候，会需要切换页表。但是此时操作系统运行在前一个线程的栈上，一旦切换，再访问栈就会导致缺页，因为每个线程的栈只在自己的页表中
5. 用户进程巧妙地设计 sp，使得它恰好落在内核的某些变量附近，于是在保存寄存器时就修改了变量的值。这相当于任意修改操作系统的控制信息
6. **实验：当键盘按下 Ctrl + C 时，操作系统应该能够捕捉到中断。实现操作系统捕获该信号并结束当前运行的线程（你可能需要阅读一点在实验指导中没有提到的代码）**

**首先我们需要打开外部中断，用到了一些后面的知识**

sie::set\_sext();

// 在 OpenSBI 中开启外部中断

\*PhysicalAddress(0x0c00\_2080).deref\_kernel() = 1u32 << 10;

// 在 OpenSBI 中开启串口

\*PhysicalAddress(0x1000\_0004).deref\_kernel() = 0x0bu8;

\*PhysicalAddress(0x1000\_0001).deref\_kernel() = 0x01u8;

// 其他一些外部中断相关魔数

\*PhysicalAddress(0x0C00\_0028).deref\_kernel() = 0x07u32;

\*PhysicalAddress(0x0C20\_1000).deref\_kernel() = 0u32;

**定义外部中断函数，其中ctrl-C的信号量是3**

fn supervisor\_external(context: &mut Context) -> \*mut Context {

let mut c = console\_getchar();

if c <= 255 {

if c == '\r' as usize {

c = '\n' as usize;

}

STDIN.push(c as u8);

}

let m=c as u8;

**if(c==3){// ctrl+c exit**

**let mut processor = PROCESSOR.lock();**

**let current\_thread = processor.current\_thread();**

**current\_thread.as\_ref().inner().dead=true;**

**}// ctrl+c退出当前线程**

context

}

**然后在handle\_interrupt接受这个中断**

Trap::Interrupt(Interrupt::SupervisorExternal) => supervisor\_external(context),

即可实现中断线程

1. **实验：实现线程的 clone()。目前的内核线程不能进行系统调用，所以我们先简化地实现为“按 C 进行 clone”。clone 后应当为目前的线程复制一份几乎一样的拷贝，新线程与旧线程同属一个进程，公用页表和大部分内存空间，而新线程的栈是一份拷贝。**

首先我们需要加入c 外部中断 c的ascii是99

也就是说我们需要实现fork()

pub fn fork(&self,context:&Context)->MemoryResult<Arc<Thread>>{

let stack=self.stack.clone();

let process=self.process.clone();

let thread = Arc::new(Thread {

id: unsafe {

THREAD\_COUNTER += 1;

THREAD\_COUNTER

},

stack,

process,

inner: Mutex::new(ThreadInner {

context:Some(context.clone()),

sleeping: false,

dead: false,

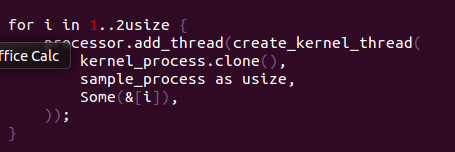
}),

});

}

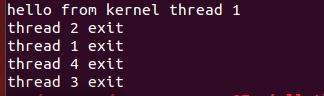
Ok(thread)

}



创建一个线程

进行fork之后



1. **线性调度的实现**

**在thread.rs中为Thread结构体增加perority参数，之后的函数不用怎么修改**

**定义结构体**

struct StriThread<ThreadType:Clone+Eq>{

birth\_time:usize,

stride:usize,

pass:usize,

pub thread:ThreadType,

}

pub struct StriScheduler<ThreadType:Clone+Eq>{

time:usize,

pool:Vec<StriThread<ThreadType>>

}

**增加线程**

fn add\_thread(&mut self,thread:ThreadType,priority:usize){

self.pool.push(StriThread{

birth\_time:self.time,

stride:BigStride/priority,

pass:BigStride/priority,

thread,

})

}

}

**获取最小stride线程**

fn get\_next(&mut self)->Option<ThreadType>{

self.time+=1;

let mut min=0;

if self.pool.len()>0{

for i in 1..self.pool.len(){

if self.pool[min].stride>self.pool[i].stride{

min=i;

}

}

self.pool[min].stride+=self.pool[min].pass;

return Some(self.pool[min].thread.clone());

}

None

}

**移除线程**

fn remove\_thread(&mut self,thread:&ThreadType){

for i in 0..self.pool.len(){

if self.pool[i].thread==\*thread{

self.pool.remove(i);

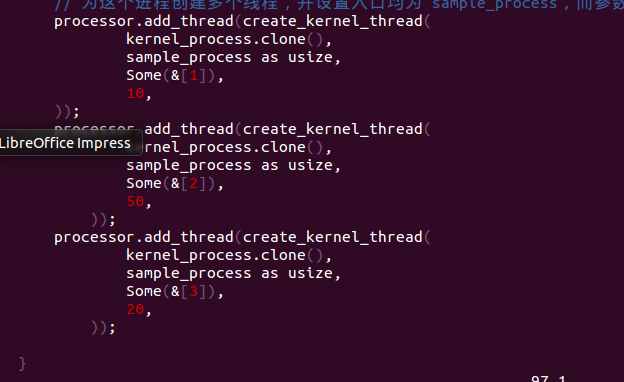
break;

}

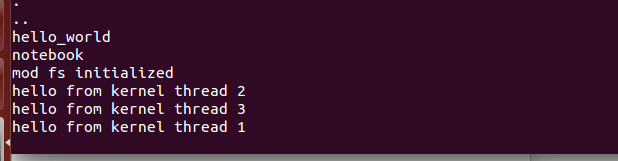
}

}

Main.rs定义



结果：



符合调度的标准