Lab-4

1. 原理：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？

   线程切换函数是`Process::prepare_next_thread()`，我们看这个函数

```
pub fn prepare_next_thread(&mut self) -> *mut Context {
        // 向调度器询问下一个线程
        if let Some(next_thread) = self.scheduler.get_next() {
            // 准备下一个线程
            let context = next_thread.prepare();
            self.current_thread = Some(next_thread);
            context
        } else {
            // 没有活跃线程
            if self.sleeping_threads.is_empty() {
                // 也没有休眠线程，则退出
                panic!("all threads terminated, shutting down");
            } else {
                // 有休眠线程，则等待中断
                self.current_thread = Some(IDLE_THREAD.clone());
                IDLE_THREAD.prepare()
            }
        }
}
```

   该get_next()函数没有修改页表，因此看prepare函数的代码

```
pub fn prepare(&self) -> *mut Context {
        // 激活页表
        self.process.inner().memory_set.activate();
        // 取出 Context
        let parked_frame = self.inner().context.take().unwrap();
        // 将 Context 放至内核栈顶
        unsafe { KERNEL_STACK.push_context(parked_frame) }
}
```

   可以看到activate函数是用来激活页表的

```
pub fn activate(&self) {
        // satp 低 27 位为页号，高 4 位为模式，8 表示 Sv39
        let new_satp = self.root_ppn.0 | (8 << 60);
        unsafe {
            // 将 new_satp 的值写到 satp 寄存器
            llvm_asm!("csrw satp, $0" :: "r"(new_satp) :: "volatile");
            // 刷新 TLB
```

```
            llvm_asm!("sfence.vma" :::: "volatile");
        }
    }
```

该函数将新线程根页表的物理页号写入了satp寄存器，并刷新了TLB，这样就换入了新线程的页表

页表切换是进程/线程切换的一个过程，而线程切换是程序运行时发生时钟中断（时间片到了）或是程序主动让出CPU导致的，所以此时执行的是内核代码，内核代码执行完后就会继续执行在程序代码。

页表的切换不会影响操作系统的执行，因为每个进程的MemorySet中都有new_kernel函数创建的Segment，这些segment就是内核程序的页表

2. 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）：
    - 一种情况不会出现问题
    - 一种情况导致异常无法处理（指无法进入 `handle_interrupt`）
    - 一种情况导致产生嵌套异常（指第二个异常能够进行到调用 `handle_interrupt`，不考虑后续执行情况）
    - 一种情况导致一个用户进程（先不考虑是怎么来的）可以将自己变为内核进程，或以内核态执行自己的代码

    • 只需让线程不产生中断即可，答案给出的`loop {}`是最简单的方法。
    • 此处存疑，我认为就算线程把sp搞丢了，发生中断是也能进入interrupt.asm，__interrupt函数会保存错误的sp，并调用handle_interrupt，此时异常也应该能够被处理，但是已经无法正常恢复中断了
    • 这个没想出来，单线程中不可能出现嵌套异常，所以只能多线程了，切换页表的时候会产生嵌套异常。
    • 没想出来，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
3. 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）

    这个只需要在supervisor_external函数中加一个match分支处理下就好了，根据ascii表查出Ctrl+C的值为3，所以为3时进入分支就好

```
match c {
        3 => {
            println!("terminated: killed by user");
```

```
            PROCESSOR.lock().kill_current_thread();
            return PROCESSOR.lock().prepare_next_thread();
        },
```

4. 实验：实现进程的 `fork()`。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 F 进行 fork”。fork 后应当为目前的进程复制一份几乎一样的拷贝。

> 旧题目：这个题目有一些问题，会导致线程中对栈上的指针失效。如果已经完成了 clone() 实验，推荐但不必须重新做 fork()。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。

很快就写完了代码，然后。。。不知道问什么就是有bug，调试了2个小时才找到原因，现在gdb已经用的非常熟练了。。。。

刚开始发现程序启动按下c后fork 函数调用后卡住不动

然后打开gdb调试，发现卡在了 copy_nonoverlapping 这个函数，它是用来原线程的栈的内容拷贝到新线程的栈里的，而且我还发现，每次都是两次缺页中断后，程序进入无响应状态，没有报错，像是被阻塞了。

将 copy_nonoverlapping 注释掉，程序在两次page_fault之后fault 了，这个没想出来具体是为什么

```
Thread {
    thread_id: 0x2,
    stack: Range {
        start: VirtualAddress(
            0x1080000,
        ),
        end: VirtualAddress(
            0x1100000,
        ),
    },
    context: None,
```

```
} terminated: stval page is not mapped
cause: Exception(LoadPageFault), stval: 0
```

后来觉得很奇怪，为什么会卡住不动呢，如果出错的话，也应该有报错信息才对，在另一个终端下打开top，发现CPU占用率100%，要么是程序死循环了，要么是CPU在轮询等待释放某种资源。

仔细检查了下，死循环是没有的。那么只可能是轮询等待，

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
10406 root      20   0 1637772  42696  11540 S 100.0  4.2  30:30.23 qemu-system-ris
    1 root      20   0    3788   1512   1480 S   0.0  0.1   0:00.04 bash
   18 root      20   0    3784   1196   1196 S   0.0  0.1   0:00.01 bash
   35 root      20   0    4748   1368   1364 S   0.0  0.1   0:00.01 su
   36 root      20   0    6904   2784   2396 S   0.0  0.3   0:00.19 zsh
  481 root      20   0    3784   2504   2504 S   0.0  0.2   0:00.01 bash
```

后来debug发现config文件中用户进程最多使用的物理页面数量，只有16个，共64KB，而线程的运行栈大小设置的却是512KB！于是 我猜测copy_nonoverlapping在按位复制的时候，应该是以很多个页面为单位的，这样就需要在申请很多个物理页，甚至多到64KB内存都不够。

果断改变了KERNEL_PROCESS_FRAME_QUOTA的大小，改为256，程序果然能够正常运行了。不过我还是有些不放心，线程运行栈的大小是512KB，256个物理页面大小是1MB，完全装的下，会不会是物理页分配器出了问题，而不是copy_nonoverlapping的问题？

于是做了几组实验，修改KERNEL_PROCESS_FRAME_QUOTA的大小，看是否能够运行成功

Aa quota	缺页次数	是否运行成功
256	0	是
156	100	是
130	127	否
132	124	是
131	125	是

观察这组数据，130是一个临界点，决定了是否能够成功执行copy_nonoverlapping，而且成功时quota和缺页次数之和总为256，但是copy_nonoverlapping是一个intrinsics函数，我还没找到源代码，所以具体原因我还不得而知

测试平台：macbook air M1 8G，BigSur 11.4，qemu 5.0，docker engine 20.10.7，ubuntu:18.04 arm

KERNEL_PROCESS_FRAME_QUOTA已改为192，能够通过测试

```
page_fault 50
page_fault 51
page_fault 52
page_fault 53
page_fault 54
page_fault 55
page_fault 56
page_fault 57
page_fault 58
page_fault 59
page_fault 60
page_fault 61
page_fault 62
page_fault 63
page_fault 64
cloned current thread
thread 2
thread 1
thread 2
200 tick
terminated: killed by user
thread 1
300 tick
thread 1
terminated: killed by user
src/process/processor.rs:87: 'all threads terminated, shutting down'
```