



课程实验报告

0x03 内核线程与缺页异常

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 4 月 28 日

目录

1 代码补全	3
1.1 进程管理器初始化	3
1.2 进程调度实现	3
1.3 内核线程创建	5
1.4 缺页异常处理	7
1.5 进程退出	11
2 结果展示	12
2.1 进程切换测试	12
2.2 缺页异常测试	12
3 思考题	13
3.1 q1	13
3.2 q2	14
3.3 q3	14

1 代码补全

1.1 进程管理器初始化

? Question

在 `src/proc/mod.rs` 中，可以看到待补全的 `init` 函数，这个函数将内核包装成进程，并将其传递给 `ProcessManager`，使其成为第一个进程。

```

1  /// init process manager Rust
2  pub fn init() {
3      let proc_vm =
4          ProcessVm::new(PageTableContext::new()).init_kernel_vm();
5      trace!("Init kernel vm: {:#?}", proc_vm);
6      // kernel process
7      // let kproc = { /* FIXME: create kernel process */ };
8      let kproc = Process::new(
9          "kernel".to_string(),
10         None,
11         Some(proc_vm),
12         None,
13     );
14     manager::init(kproc);
15     info!("Process Manager Initialized.");
16 }
```

1.2 进程调度实现

? Question

利用 `as_handler` 宏重新定义中断处理函数，在其中调用 `crate::proc::switch` 函数，进行进程调度切换。之后在 `src/proc/mod.rs` 中，补全 `switch` 函数的实现，利用进程管理器所提供的功能，补全 `save_current` 和 `switch_next` 函数。

src/interrupt/clock.rs

重新定义时钟中断处理函数

```

1  pub unsafe fn register_idt(idt: &mut
    InterruptDescriptorTable) {
2      unsafe {
3          idt[Interrupts::IrqBase as u8 + Irq::Timer as u8]
4              .set_handler_fn(clock_handler)
5              .set_stack_index(gdt::CONTEXT_SWITCH_IST_INDEX);
6      }
7  }
8
9  pub extern "C" fn clock(mut context: ProcessContext) {
10     crate::proc::switch(&mut context);
11     super::ack();
12 }
13 as_handler!(clock);

```

src/proc/mod.rs

补全进程调度函数

```

1  pub fn switch(context: &mut ProcessContext) {
2      x86_64::instructions::interrupts::without_interrupts(|| {
3          // FIXME: switch to the next process
4          //      - save current process's context
5          //      - handle ready queue update
6          //      - restore next process's context
7          let manager = get_process_manager();
8          manager.save_current(context);
9          manager.switch_next(context);
10     });
11 }

```

src/proc/manager.rs

补全进程管理器的 save_current 和 switch_next 函数

```

1  pub fn save_current(&self, context: &ProcessContext) {
2      // FIXME: update current process's tick count
3      self.current().write().tick();
4      self.current().write().save(context);
5      if self.current().read().status() != ProgramStatus::Dead {

```

```
6         self.push_ready(processor::get_pid());
7     } else {
8         debug!("Process #{} is dead.", processor::get_pid());
9     }
10 }
11 pub fn switch_next(&self, context: &mut ProcessContext) -> ProcessId
12 {
13     // FIXME: fetch the next process from ready queue
14     // FIXME: check if the next process is ready,
15     //         continue to fetch if not ready
16     let mut pid = processor::get_pid();
17
18     while let Some(next) = self.ready_queue.lock().pop_front() {
19         let proc = self.get_proc(&next).unwrap();
20
21         if !proc.read().is_ready() {
22             debug!("Process #{} is {:?}", next,
23                 proc.read().status());
24             continue;
25         }
26
27         if pid != next {
28             // FIXME: restore next process's context
29             proc.write().restore(context);
30             // debug!("Switch to process #{}", next);
31             // FIXME: update processor's current pid
32             processor::set_pid(next);
33             pid = next;
34         }
35     }
36
37     // FIXME: return next process's pid
38     pid
39 }
```

1.3 内核线程创建

? Question

在 `src/utils/mod.rs` 中，可以看到用于测试创建内核线程的函数定义：

```
1 pub fn new_test_thread(id: &str) -> ProcessId {
2     let mut proc_data = ProcessData::new();
3     proc_data.set_env("id", id);
4
5     crate::proc::spawn_kernel_thread(
6         utils::func::test,
7         format!("#{}_test", id),
8         Some(proc_data),
9     )
10 }
```

此函数调用了在 `src/proc/mod.rs` 中定义的 `spawn_kernel_thread` 函数。它关闭中断，之后将函数转化为地址以使其能够赋值给 `rip` 寄存器，之后将进程的信息传递给 `ProcessManager`，使其创建所需进程。

在 `src/proc/process.rs` 中，根据你的内存布局预设和当前进程的 PID，为其分配初始栈空间。

参考 `bootloader` 中为内核分配栈空间的代码，克隆内核页表，使用 `elf::map_range` 函数来进行新的页面的映射。完成栈分配后，将栈顶地址返回。

src/proc/mod.rs

补全内核线程创建函数

```
1 pub fn spawn_kernel_thread(
2     &self,
3     entry: VirtAddr,
4     name: String,
5     proc_data: Option<ProcessData>,
6 ) -> ProcessId {
7     let kproc = self.get_proc(&KERNEL_PID).unwrap();
8     let page_table = kproc.read().clone_page_table();
9     let proc_vm = Some(ProcessVm::new(page_table));
10    let proc = Process::new(name, Some(Arc::downgrade(&kproc)),
11        proc_vm, proc_data);
12    // alloc stack for the new process base on pid
13    let stack_top = proc.alloc_init_stack();
14    proc.write().init_stack_frame(entry, stack_top);
```

```

14 // FIXME: push to ready queue
15 let pid = proc.pid();
16 self.add_proc(pid, proc);
17 self.push_ready(pid);
18 // FIXME: return new process pid
19 pid
20 }

```

src/proc/process.rs

补全内核线程栈分配函数

```

1 pub fn init_stack_frame(&mut self, entry: VirtAddr,
   stack_top: VirtAddr) {
2     self.value.stack_frame.stack_pointer = stack_top;
3     self.value.stack_frame.instruction_pointer = entry;
4     self.value.stack_frame.cpu_flags =
5         RFlags::IOPL_HIGH | RFlags::IOPL_LOW |
6         RFlags::INTERRUPT_FLAG;
7
8     let selector = get_selector();
9     self.value.stack_frame.code_segment = selector.code_selector;
10    self.value.stack_frame.stack_segment = selector.data_selector;
11
12    trace!("Init stack frame: {:#?}", &self.stack_frame);
13 }

```

1.4 缺页异常处理

? Question

首先，在 src/interrupt/exception.rs 中，重新定义缺页异常的处理函数：

```

1 pub extern "x86-interrupt" fn page_fault_handler(
2     stack_frame: InterruptStackFrame,
3     err_code: PageFaultErrorCode,
4 ) {
5     if !crate::proc::handle_page_fault(Cr2::read(), err_code) {
6         warn!(

```

```

7         "EXCEPTION: PAGE FAULT, ERROR_CODE: {:?}\n\nTrying to
          access: {:#x}\n{:#?}",
8         err_code,
9         Cr2::read(),
10        stack_frame
11    );
12    // FIXME: print info about which process causes page fault?
13    panic!("Cannot handle page fault!");
14 }
15 }

```

之后，你需要完善缺页异常的相关处理函数：

1. 在 `ProcessManager` 中，检查缺页异常是否包含越权访问或其他非预期的错误码。
2. 如果缺页异常是由于非预期异常导致的，或者缺页异常的地址不在当前进程的栈空间中，直接返回 `false`。
3. 如果缺页异常的地址在当前进程的栈空间中，把缺页异常的处理委托给当前的进程。

你可能需要为 `ProcessInner` 和 `ProcessVm` 添加用于分配新的栈、更新进程存储信息的函数。

4. 在进程的缺页异常处理函数中：

分配新的页面、更新页表、更新进程数据中的栈信息。

src/interrupt/exception.rs

重新定义缺页异常的处理函数

```

1  pub extern "x86-interrupt" fn page_fault_handler( Rust
2      stack_frame: InterruptStackFrame,
3      err_code: PageFaultErrorCode,
4  ) {
5      let addr = Cr2::read().unwrap();
6
7      if !crate::proc::handle_page_fault(addr, err_code) {
8          warn!(
9              "EXCEPTION: PAGE FAULT, ERROR_CODE: {:?}\n\nTrying to
              access: {:#x}\n{:#?}",
10             err_code, addr, stack_frame

```



```

11     );
12     debug!("{:#?}",
           proc::manager::get_process_manager().current());
13     panic!("Failed to handle page fault.");
14 }
15 }

```

src/proc/manager.rs

补全缺页异常处理函数

```

1  pub fn handle_page_fault(&self, addr: VirtAddr,
    err_code: PageFaultErrorCode) -> bool {
2      // FIXME: handle page fault
3      if !err_code.contains(PageFaultErrorCode::PROTECTION_VIOLATION)
4      {
5          let cur_proc = self.current();
6          trace!(
7              "Page Fault! Checking if {:#x} is on current process's
              stack",
8              addr
9          );
10         if cur_proc.pid() == KERNEL_PID {
11             info!("Page Fault on Kernel at {:#x}", addr);
12         }
13
14         let mut inner = cur_proc.write();
15         inner.handle_page_fault(addr)
16     } else {
17         false
18     }
19 }

```

src/proc/vm/stack.rs

若缺页异常发生在当前进程的栈空间中，将缺页异常的处理委托给当前进程

```

1  pub fn handle_page_fault(
2      &mut self,

```

```
3     addr: VirtAddr,
4     mapper: MapperRef,
5     alloc: FrameAllocatorRef,
6 ) -> bool {
7     if !self.is_on_stack(addr) {
8         return false;
9     }
10
11     if let Err(m) = self.grow_stack(addr, mapper, alloc) {
12         error!("Grow stack failed: {:?}", m);
13         return false;
14     }
15
16     true
17 }
18 fn grow_stack(
19     &mut self,
20     addr: VirtAddr,
21     mapper: MapperRef,
22     alloc: FrameAllocatorRef,
23 ) -> Result<(), MapToError<Size4KiB>> {
24     debug_assert!(self.is_on_stack(addr), "Address is not on
25         stack.");
26
27     // FIXME: grow stack for page fault
28     let new_start_page = Page::containing_address(addr);
29     let page_count = self.range.start - new_start_page;
30
31     elf::map_range(
32         new_start_page.start_address().as_u64(),
33         page_count,
34         mapper,
35         alloc,
36     )?;
37
38     self.range = Page::range(new_start_page, self.range.end);
39     self.usage += self.range.count() as u64;
40
41     Ok(())
42 }
```

1.5 进程退出

src/proc/process.rs

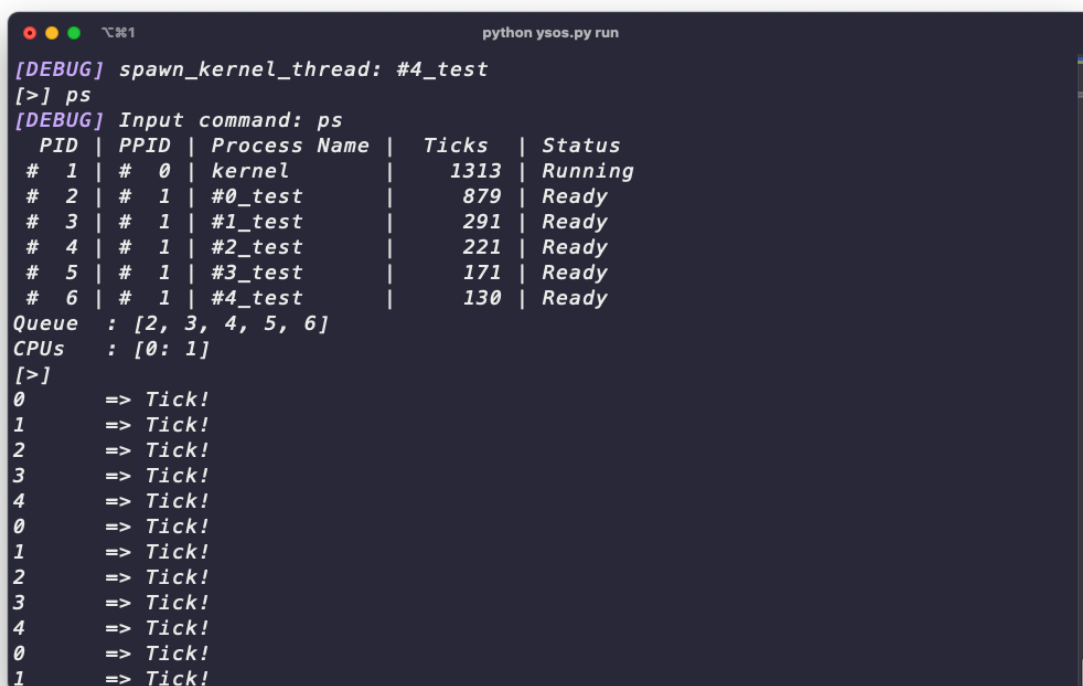
补全进程退出函数

```
1  pub fn kill(&mut self, ret: isize) { Rust
2      // FIXME: set exit code
3      self.exit_code = Some(ret);
4      // FIXME: set status to dead
5      self.status = ProgramStatus::Dead;
6      // FIXME: take and drop unused resources
7      self.proc_vm.take();
8      self.proc_data.take();
9      print!("Process {}# killed.", self.name);
10 }
```

2 结果展示

2.1 进程切换测试

创建多个 test 进程，通过 tick 查看被调度次数。使用 ps 命令查看进程状态，观察进程的状态变化。结果如下图所示。



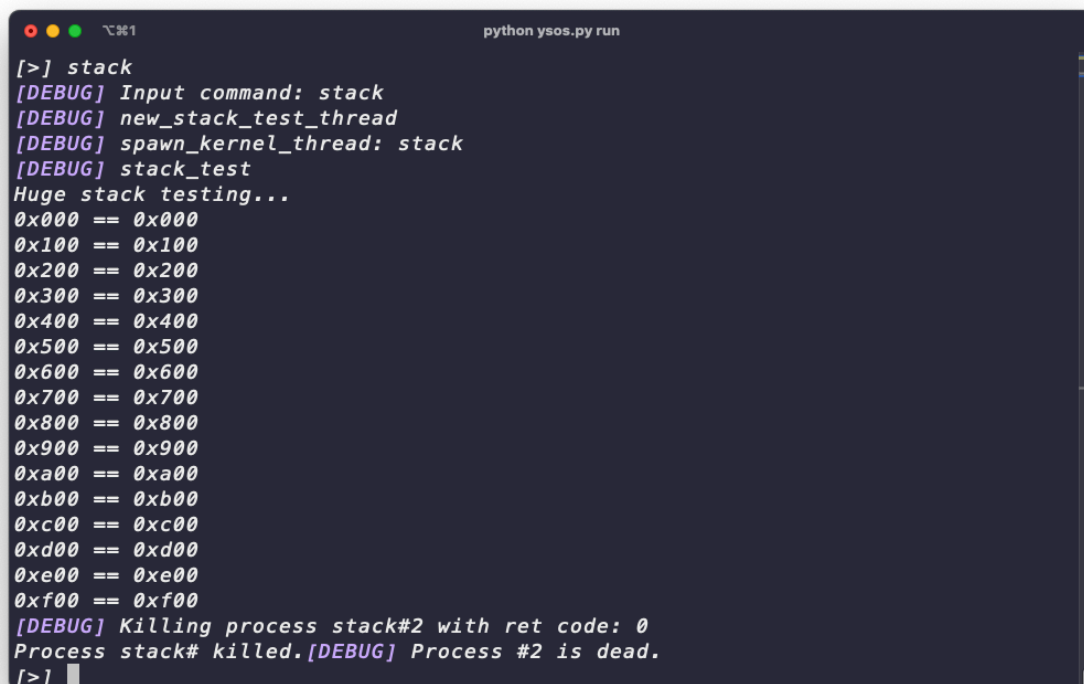
```
[DEBUG] spawn_kernel_thread: #4_test
[>] ps
[DEBUG] Input command: ps
  PID | PPID | Process Name | Ticks | Status
# 1 | # 0 | kernel | 1313 | Running
# 2 | # 1 | #0_test | 879 | Ready
# 3 | # 1 | #1_test | 291 | Ready
# 4 | # 1 | #2_test | 221 | Ready
# 5 | # 1 | #3_test | 171 | Ready
# 6 | # 1 | #4_test | 130 | Ready
Queue : [2, 3, 4, 5, 6]
CPUs  : [0: 1]
[>]
0      => Tick!
1      => Tick!
2      => Tick!
3      => Tick!
4      => Tick!
0      => Tick!
1      => Tick!
2      => Tick!
3      => Tick!
4      => Tick!
0      => Tick!
1      => Tick!
```

图 2.1.1: 进程切换测试结果

从图中可以看出，一共创建了 4 个进程，分别是 #0_test、#1_test、#2_test、#3_test 和 #4_test。它们的状态均为 Ready。每个进程的 tick 数量也在不断增加，说明它们都在被调度执行。

2.2 缺页异常测试

通过 stack 函数不断创建新的栈，观察缺页异常的处理情况。结果如下图所示。



```
[>] stack
[DEBUG] Input command: stack
[DEBUG] new_stack_test_thread
[DEBUG] spawn_kernel_thread: stack
[DEBUG] stack_test
Huge stack testing...
0x000 == 0x000
0x100 == 0x100
0x200 == 0x200
0x300 == 0x300
0x400 == 0x400
0x500 == 0x500
0x600 == 0x600
0x700 == 0x700
0x800 == 0x800
0x900 == 0x900
0xa00 == 0xa00
0xb00 == 0xb00
0xc00 == 0xc00
0xd00 == 0xd00
0xe00 == 0xe00
0xf00 == 0xf00
[DEBUG] Killing process stack#2 with ret code: 0
Process stack# killed.[DEBUG] Process #2 is dead.
[>]
```

图 2.2.2: 缺页异常测试结果

从图中可以看出，没有触发 kernel panic。说明缺页异常的处理函数能够正确地处理进程栈内的缺页异常，能够正确地更新进程的栈信息。

3 思考题

3.1 q1

? Question

为什么在初始化进程管理器时需要将它置为正在运行的状态？能否通过将它置为就绪状态并放入就绪队列来实现？这样的实现可能会遇到什么问题？

- 保持内核控制权

内核进程作为系统管理和调度的入口，如果放入就绪队列，可能会被调度器切换出去。这会导致一些关键的内核任务延迟执行，从而影响系统稳定性和响应能力。通过将它置为正在运行状态，可以确保内核始终保持主动控制权。

- 避免调度问题

如果将内核进程放入就绪队列，系统在某些调度策略下可能会优先切换到其他不那么关键的任务，从而导致内核调度逻辑不一致。这可能会使得内核进程在需要执行紧急操作或初始化操作时被延迟调用，进而产生死锁或状态不一致。

- 初始化安全性

内核进程在初始化阶段可能涉及资源分配、页表设置等关键操作。如果这些操作被中断或在等待状态下切换，可能会引发竞争或者错误的资源管理。将内核进程设置为“正在运行”的状态，可以确保这些初始化操作完整执行，然后再允许系统调度其他任务。

3.2 q2

? Question

在 `src/proc/process.rs` 中，有两次实现 `Deref` 和一次实现 `DerefMut` 的代码，它们分别是为了什么？使用这种方式提供了什么便利？

1. 对于 `Process` 实现的 `Deref`

实现了 `Deref<Target = Arc<RwLock<ProcessInner>>>` 后，使得我们可以直接将一个 `Process` 当作其内部的 `Arc<RwLock<ProcessInner>>` 使用，方便调用如 `read()` 和 `write()` 等方法，而不必每次都取出内层字段。例如，可以直接写 `process.read().xxx()` 来访问 `ProcessInner` 内部的数据。

2. 对于 `ProcessInner` 实现的 `Deref` 和 `DerefMut`

这里将 `Target` 定义为 `ProcessData`（注意，`ProcessData` 是 `ProcessInner` 中的一个 `Option`，这里 `unwrap` 了并提供了直接访问方式）。这样，当你操作一个 `ProcessInner` 时，就可以像直接访问 `ProcessData` 那样操作它的成员，而不必显式地通过 `proc_data` 字段访问。`Deref` 提供了不可变访问，而 `DerefMut` 则提供了可变访问。

这种方式的便利在于它提供了类似于“自动解引用”的语法糖，使得访问嵌套数据结构中的内容更加直观和便捷，而不用每次都进行显式地字段访问或 `Option unwrap`，从而使代码更加简洁易读。

3.3 q3

? Question

中断的处理过程默认是不切换栈的，即在中断发生前的栈上继续处理中断过程，为什么在处理缺页异常和时钟中断时需要切换栈？如果不为它们切换栈会分别带来哪些问题？请假设具体的场景、或通过实际尝试进行回答。

1. 缺页异常

当缺页异常发生时，触发异常的地址可能正好落在当前进程的用户栈或其正在动态扩展的栈区域上。如果用原来的（可能不完整、甚至损坏的）栈作为异常处理的工作栈，容易导致：

- 进一步缺页或异常：当前栈内容失效可能导致异常处理中再次触发缺页（甚至出现双重异常）；
- 栈溢出风险：异常处理逻辑可能需要额外的栈空间，而当前栈空间若已经不足，容易引起溢出和不可预知的错误。

切换到一个预先分配、受保护的内核栈，可以保证缺页异常处理函数在一个稳定的环境中运行，从而安全地扩展或修复进程的栈内存映射。

2. 时钟中断

时钟中断频繁且需要确保调度、上下文切换等关键操作的正确执行。如果一直在当前进程的栈上处理时钟中断：

- 状态污染：当前进程栈可能正处于用户或任务特定的上下文中，中断处理过程中对栈的使用可能破坏该上下文；
- 嵌套中断问题：如果中断在处理时钟中断过程中被再次触发，可能因原栈空间不足而引发栈溢出或数据竞争。

切换到一个专门的内核中断栈（通过 IST 等机制）可以隔离中断逻辑与用户或任务逻辑，保证中断处理过程独立、快速并且安全。