中山大學

SUN YAT-SEN UNIVERSITY

实验报告

实验三：内核线程与多核异常

姓　　名	元朗曦
学　　号	23336294
班　　级	计算机八班
专　　业	计算机科学与技术
学　　院	计算机学院

2 合并实验代码

在 pkg/kernel/src/utils 文件夹中，增量代码补充包含了如下的模块：

- regs.rs：对需要保存的一系列寄存器进行了封装，规定了其输出方式，补全了进程切换时需要使用的汇编代码及 as_handler 宏。
- func.rs：定义了用于测试执行的两个函数，其中 test 用以验证进程调度、并发的正确性，huge_stack 用以验证处理缺页异常的正确性。

在 pkg/kernel/src/proc 文件夹中，增量代码补充包含了如下的模块：

- context.rs：进程上下文的定义和实现，其中包含了加载、保存进程上下文的相关函数。
- data.rs：进程数据结构体的定义，这里存储的数据在进程被杀死后会被释放，包含了使用 Arc 保护的线程间共享的数据（子进程相关内容将在下次实验中使用）。
- vm/{mod.rs,stack.rs}：进程的虚拟内存管理，包含了栈空间的分配和释放函数，以及一些常量的定义。
- manager.rs：进程管理器的定义和实现，时钟中断最终会通过进程管理器来进行任务切换。
- paging.rs：进程页表的存储、切换所用数据，使用 load 函数加载进程页表到 Cr3 寄存器，使用 clone 函数来获得当前页表的副本，用于创建新进程。
- pid.rs：使用元组结构体将一个 u16 作为进程 ID，需要为 new 函数确保获取唯一的 PID。
- processor.rs：对处理器的抽象，使用 AtomicU16 来存储当前正在运行的进程的 PID，使用 set_pid 函数来设置当前进程的 PID，使用 get_pid 函数来获取当前进程的 PID。
- process.rs：进程结构体的核心实现，包含了进程的状态、调度计数、退出返回值、父子关系、中断上下文等内容，是管理进程的核心模块。

> ⚠ **注意**
>
> 增量代码在 pkg/kernel/Cargo.toml 中添加了对 volatile 包的依赖，但没有修改根目录的 Cargo.toml，这导致 rust-analyzer 在我们打开项目内的 Rust 代码时输出警告。
>
> 解决办法是在根目录的 Cargo.toml 中 [workspace.dependencies] 块的末尾添加

```toml
volatile = { version = "0.5.0", default-features = false }
```

3 进程管理器的初始化

1. 创建内核结构体

   补全 pkg/kernel/src/proc/mod.rs 中 init 函数如下：

```rs
/// init process manager
pub fn init() {
    let proc_vm = ProcessVm::new(PageTableContext::new()).init_kernel_vm();

    trace!("Init kernel vm: {:#?}", proc_vm);

    // kernel process
```

```
    let kproc = Process::new(
        String::from("kernel"),
        None,
        Some(proc_vm),
        None);
    manager::init(kproc);

    info!("Process Manager Initialized.");
}
```

2. 初始化内核进程

   被创建后，进程将会被传递至 `manager::init()` 函数。在这个函数中，需要将初始化进程设置为当前唯一正在运行的进程。设置内核进程的状态为 `Running`，并将其 PID 加载至当前的 CPU 核心结构体中。补全 `pkg/kernel/src/proc/manager.rs` 中 `init` 函数如下：

```rs
pub fn init(init: Arc<Process>) {
    // DONE: set init process as Running
    init.write().resume();

    // DONE: set processor's current pid to init's pid
    processor::set_pid(init.pid());

    PROCESS_MANAGER.call_once(|| ProcessManager::new(init));
}
```

4 进程调度的实现

我们将在 TSS 中声明一块新的中断处理栈，并将它加载到时钟中断的 IDT 中。首先修改 `pkg/kernel/src/memory/gdt.rs` 如下：

```rs
// ...

pub const DOUBLE_FAULT_IST_INDEX: u16 = 0;
pub const PAGE_FAULT_IST_INDEX: u16 = 1;
pub const CONTEXT_SWITCH_IST_INDEX: u16 = 2;

// ...

lazy_static! {
    static ref TSS: TaskStateSegment = {
        // ...

        tss.interrupt_stack_table[CONTEXT_SWITCH_IST_INDEX as usize] = {
            const STACK_SIZE: usize = IST_SIZES[3];
            static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
            let stack_start = VirtAddr::from_ptr(addr_of_mut!(STACK));
            let stack_end = stack_start + STACK_SIZE as u64;
```

```
            info!(
                "Context Switch Stack: 0x{:016x}-0x{:016x}",
                stack_start.as_u64(),
                stack_end.as_u64()
            );
            stack_end
        };

        tss
    };
}
```

我们利用 `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。`pkg/kernel/src/interrupts/clock.rs` 实现如下：

```rs
use super::consts;

use crate::memory::gdt;
use crate::proc::ProcessContext;

use x86_64::structures::idt::{
    InterruptDescriptorTable,
    InterruptStackFrame
};

pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    unsafe {
        idt[consts::Interrupts::IrqBase as u8 + consts::Irq::Timer as u8]
            .set_handler_fn(clock_handler)
            .set_stack_index(gdt::CONTEXT_SWITCH_IST_INDEX);
    }
}

pub extern "C" fn clock(mut context: ProcessContext) {
    crate::proc::switch(&mut context);
    super::ack();
}
as_handler!(clock);
```

之后我们在 `pkg/kernel/src/proc/mod.rs` 中，补全 `switch` 函数的实现：

```rs
pub fn switch(context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        // DONE: switch to the next process
        //   - save current process's context
        //   - handle ready queue update
        //   - restore next process's context
```

```
        let manager = get_process_manager();
        manager.save_current(context);
        manager.push_ready(processor::get_pid());
        manager.switch_next(context);
    });
}
```

再利用进程管理器所提供的功能，在 `pkg/kernel/src/proc/manager.rs` 中，补全 `save_current` 和 `switch_next` 函数：

```rs
impl ProcessManager {
    // ...

    pub fn save_current(&self, context: &ProcessContext) {
        // DONE: update current process's tick count
        self.current().write().tick();

        // DONE: save current process's context
        self.current().write().save(context);
    }

    pub fn switch_next(&self, context: &mut ProcessContext) -> ProcessId {
        // DONE: fetch the next process from ready queue
        let mut next_pid = processor::get_pid();

        // DONE: check if the next process is ready, continue to fetch if not ready
        while let Some(pid) = self.ready_queue.lock().pop_front() {
            let proc = self.get_proc(&pid).unwrap();

            if proc.read().is_ready() {
                next_pid = pid;
                break;
            }
        }

        // DONE: restore next process's context
        let next_proc = self.get_proc(&next_pid).unwrap();
        next_proc.write().restore(context);

        // DONE: update processor's current pid
        processor::set_pid(next_pid);

        // DONE: return next process's pid
        next_pid
    }

```

```
37      // ...
38  }
```

5 进程信息的获取

5.1 环境变量

补全 `pkg/kernel/src/proc/mod.rs` 中的 `env` 函数，使得外部函数可以获取到当前进程的环境变量：

```rs
pub fn env(key: &str) -> Option<String> {
    x86_64::instructions::interrupts::without_interrupts(|| {
        // DONE: get current process's environment variable
        get_process_manager().current().read().env(key)
    })
}
```

5.2 进程返回值

在 `pkg/kernel/src/utils/mod.rs` 中，补全 `wait` 函数：

```rs
fn wait(pid: ProcessId) {
    loop {
        // DONE: try to get the status of the process
        // HINT: it's better to use the exit code
        let proc = manager::get_process_manager().get_proc(&pid).unwrap();
        let code = proc.read().exit_code();

        if code.is_none() {
            x86_64::instructions::hlt();
        } else {
            break;
        };
    }
}
```

这一功能的实现在未来将被用于 `wait_pid` 系统调用，从而让父进程可以等待子进程的退出，或查询其他进程的返回值等。

6 内核线程的创建

本次实验被称做“内核线程”，其原因在于将要创建的进程并不需要对页表、权限、代码段等进行特殊的设置，它们的内存空间和内核是共享的，因此可以当作“线程”来看待。

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

```rs
pub fn new_test_thread(id: &str) -> ProcessId {
    let mut proc_data = ProcessData::new();
    proc_data.set_env("id", id);

    spawn_kernel_thread(
```

```
6           utils::func::test,
7           format!("#{}_test", id),
8           Some(proc_data),
9       )
10  }
```

此函数调用了在 pkg/kernel/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赋值给 rip 寄存器，并将进程的信息传递给 ProcessManager，使其创建所需进程。

我们首先补全 pkg/kernel/src/proc/manager.rs 中的 spawn_kernel_thread 函数：

```rs
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)), proc_vm,
        proc_data);
11
12      // alloc stack for the new process base on pid
13      let stack_top = proc.alloc_init_stack();
14
15      // DONE: set the stack frame
16      proc.write().init_stack_frame(entry, stack_top);
17
18      let pid = proc.pid();
19      // DONE: add to process map
20      self.add_proc(pid, proc);
21      // DONE: push to ready queue
22      self.push_ready(pid);
23      // DONE: return new process pid
24      pid
25  }
```

然后在 pkg/kernel/src/proc/process.rs 中，根据我们的内存布局预设和当前进程的 PID，为内核线程分配初始栈空间：

```rs
1   pub fn alloc_init_stack(&self) -> VirtAddr {
2       self.write().vm_mut().init_proc_stack(self.pid)
3   }
```

这会调用 pkg/kernel/src/proc/vm/mod.rs 中定义的 init_proc_stack 函数：

```rs
1   pub fn init_proc_stack(&mut self, pid: ProcessId) -> VirtAddr {
```

```rust
        // DONE: calculate the stack for pid
        let stack_top_addr = STACK_INIT_TOP - (pid.0 as u64 - 1) * STACK_MAX_SIZE;
        let frame_allocator = &mut *get_frame_alloc_for_sure();

        map_range(stack_top_addr, STACK_DEF_PAGE, &mut self.page_table.mapper(),
        frame_allocator).unwrap();

        let stack_top_vaddr = VirtAddr::new(stack_top_addr);

        self.stack = Stack::new(
            Page::containing_address(stack_top_vaddr),
            STACK_DEF_PAGE,
        );

        stack_top_vaddr
    }
```

7 缺页异常的处理

在操作系统进行虚拟内存管理的时候经常会遇到缺页中断，作为可恢复的异常，它发生的可能性有很多：

- 内存页被标记为懒分配，只有当进程访问到这一页面时才会被分配。
- 部分可执行的代码段尚未被加载到内存中，需要从磁盘文件进行加载。
- 内存被交换到了磁盘上，再次使用需要交换回来。
- 内存页面被标记为只读，在进程尝试写页面的时候触发了 Copy on Write 机制，需要进行页面的复制。
- 进程访问量权限不允许的内存区域，比如用户态进程尝试访问内核空间的内存。

在本实验设计中，并不会完全的实现上述的所有功能，只实现一个功能来作为缺页异常处理的示例：**为栈空间进行自动扩容**。初始化时，我们从栈顶（此进程具有的 4 GiB 的最大位置）开始，向下分配了 4 KiB 的栈空间。当进程使用的栈一直增长，直到超过了 4 KiB 的栈空间时，就会触发缺页异常。

在触发缺页异常时，尝试访问的地址会被保存在 Cr2 寄存器中，同时缺页异常的错误码也会随着中断栈一起传递给中断函数。

首先，我们在 `pkg/kernel/src/interrupt/exception.rs` 中，重新定义缺页异常的处理函数：

```rust
pub extern "x86-interrupt" fn page_fault_handler(
    stack_frame: InterruptStackFrame,
    err_code: PageFaultErrorCode,
) {
    let addr = Cr2::read().unwrap();

    if !crate::proc::handle_page_fault(addr, err_code) {
        warn!(
```

```
            "EXCEPTION: PAGE FAULT, ERROR_CODE: {:?}\n\nTrying to access:
            {:#x}\n{:#?}",
            err_code, addr, stack_frame
        );
        crate::proc::current_process_info();
        panic!("Cannot handle page fault!");
    }
}
```

它会调用 pkg/kernel/src/proc/mod.rs 中的 handle_page_fault 函数：

```rs
pub fn handle_page_fault(addr: VirtAddr, err_code: PageFaultErrorCode) ->
bool {
    x86_64::instructions::interrupts::without_interrupts(|| {
        get_process_manager().handle_page_fault(addr, err_code)
    })
}
```

进而调用 pkg/kernel/src/proc/manager.rs 中的 handle_page_fault 函数：

```rs
pub fn handle_page_fault(&self, addr: VirtAddr, err_code:
PageFaultErrorCode) -> bool {
    // DONE: handle page fault
    if err_code.contains(PageFaultErrorCode::PROTECTION_VIOLATION) {
        return false;
    }

    let proc = self.current();
    trace!("Page Fault! Checking if {:#x} is on current process's stack", addr);

    if proc.pid() == KERNEL_PID {
        info!("Page Fault on kernel at {:#x}", addr);
    }

    proc.write().handle_page_fault(addr);

    true
}
```

8 进程的退出

在正常的操作系统中，进程的退出通常是由一个系统调用实现的。和时钟中断类似，系统调用也会保存进程的上下文，之后调用内核中的函数，完成进程的退出和切换，因此可以很好的确保进程不会再次被调度。

但是目前我们没有实现系统调用，内核线程的退出是它主动调用内核的 process_exit 来实现的，这样的函数调用没有进程上下文的存储和恢复过程，因此需要进行一些额外的处理，并等待下一次的时钟中断来完成进程的切换。

首先补全 pkg/kernel/src/proc/process.rs 中的相关部分：

```rs
impl ProcessInner {
    // ...

    /// Save the process's context
    /// mark the process as ready
    pub(super) fn save(&mut self, context: &ProcessContext) {
        // DONE: save the process's context
        self.context.save(context);
        // DONE: mark the process as ready
        if self.status == ProgramStatus::Running {
            self.status = ProgramStatus::Ready;
        }
    }

    /// Restore the process's context
    /// mark the process as running
    pub(super) fn restore(&mut self, context: &mut ProcessContext) {
        // DONE: restore the process's context
        self.context.restore(context);
        // DONE: restore the process's page table
        self.vm().page_table.load();
        // DONE: mark the process as running
        if self.status == ProgramStatus::Ready {
            self.status = ProgramStatus::Running;
        }
    }

    pub fn parent(&self) -> Option<Arc<Process>> {
        self.parent.as_ref().and_then(|p| p.upgrade())
    }

    pub fn kill(&mut self, ret: isize) {
        // DONE: set exit code
        self.exit_code = Some(ret);
        // DONE: set status to dead
        self.status = ProgramStatus::Dead;
        // DONE: take and drop unused resources
        self.proc_data.take();
        self.proc_vm.take();
    }
}
```

为了确保进程不会再次被调度，我们需要修改 `switch` 操作。修改 `pkg/kernel/src/proc/manager.rs` 中的 `save_current` 函数如下：

```rs
pub fn save_current(&self, context: &ProcessContext) {
```

```rs
    let proc = self.current();
    let pid = proc.pid();

    let mut inner = proc.write();
    // DONE: update current process's tick count
    inner.tick();
    // DONE: save current process's context
    inner.save(context);

    let status = inner.status();
    drop(inner);

    if status != ProgramStatus::Dead {
        self.push_ready(pid);
    } else {
        debug!("Process #{} {:#?} is dead.", pid, proc);
    }
}
```

我们将 push_ready 的操作合并到了 save_current 函数中，并添加了对进程状态的判断，确保只有在进程处于“运行”状态时才会被放入就绪队列。之后还需要在 switch 函数中删去原来的 push_ready 行：

```rs
pub fn switch(context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        // DONE: switch to the next process
        //   - save current process's context
        //   - handle ready queue update
        //   - restore next process's context
        let manager = get_process_manager();
        manager.save_current(context);
        manager.switch_next(context);
    });
}
```

9 思考题

1. 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？

> **解答**
>
> 1. 立即控制权： 当进程管理器被初始化之后，它要立即对系统中的其他进程进行调度。如果将它置为“就绪”并加入就绪队列，那么调度器自己也得依赖调度算法来获得运行权，而这可能会导致延迟。操作系统设计者通常希望调度器能够持续独占或至少优先获得控制权，以便及时作出决策。
> 2. 避免被调度器自身抢占： 如果将进程管理器也当作一个普通进程对待，放入就绪队列中，它就有可能在某些调度算法下被抢占掉。例如，其他高优先级进程可能会使调度器无法及时运行，这会影响系统对新任务的分派和调度效率。
> 3. 潜在的同步问题： 进程管理器不仅仅负责简单地从就绪队列中选择进程运行，它还要管理上下文切换、处理中断以及响应异步事件。如果将它置于就绪队列中，它可能会与其他进程竞争 CPU，从而引入竞争条件和同步问题，甚至可能导致调度上的不一致或者死锁情况。
> 4. 系统内核设计的考虑： 在许多操作系统设计中，调度器和进程管理器往往以内核线程或特殊的内核代码形式存在，其运行权被设计为始终保持在“运行”状态。这样可以确保对系统事件立即响应，而不用等待调度器重新从就绪队列中被选中运行。

2. 在 `pkg/kernel/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？

> **解答**
>
> 1. 对 `Process` 的 `Deref` 实现
>
>   当我们对 `Process` 进行解引用操作时，它会自动转换为内部的 `Arc<RwLock<ProcessInner>>`。这样一来，调用者可以直接利用 `Process` 来访问封装的 RwLock 方法，而无需首先取出 `inner` 字段。例如，直接调用类似 `inner.read()` 或 `inner.write()` 的方法。
>
> 2. 对 `ProcessInner` 的 Deref 实现
>
>   为 `ProcessInner` 实现 `Deref`，将其内部的 `Option<ProcessData>` 中的 `ProcessData` 取出（必须存在，否则会 `panic`），从而使得 `ProcessInner` 对象在需要 `ProcessData` 的地方能够自动解引用。这样，对于只需要只读 `ProcessData` 的操作，就可以直接在 `ProcessInner` 对象上调用 `ProcessData` 的方法，而不必显式地取出 `proc_data` 字段。
>
> 3. 对 `ProcessInner` 的 `DerefMut` 实现
>
>   类似地，`DerefMut` 允许对 `ProcessInner` 做可变解引用，直接对 `ProcessData` 进行修改操作。这样一来，当需要改变 `ProcessData` 的内容时，可以直接将 `ProcessInner` 当作 `ProcessData` 来对待，而不必将其包装层拆开。

3. 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。

解答

1. 缺页异常（Page Fault）

   当发生缺页异常时，很可能是因为某个进程访问了未映射或不合法的内存区域，而导致的异常。在处理缺页异常时，如果继续使用当前中断发生时的栈（通常是用户栈或者已有的内核栈）会带来以下问题：

   - 如果未映射地址正是与当前栈有关的页面，内核在中断异常处理过程中可能也会因为栈页缺失而触发递归缺页异常，使问题更加复杂，甚至导致系统崩溃。
   - 有的场景下，用户栈处于低地址区域，而内核需要调用一系列复杂的异常处理代码（如加载缺失页、重新设置页表等），这些操作可能需要占用较多的栈空间。如果继续在原有栈上运行，栈空间不足可能引发溢出或破坏原有栈数据，导致内核态数据错乱。

因此，为了保证缺页异常处理代码能够在一个干净且足够的内核栈上运行，操作系统通常在进入异常处理时切换到专用的内核栈（通常每个 CPU/进程会有固定的内核栈）。

1. 时钟中断（Timer Interrupt）

   时钟中断是一种周期性、异步中断，它用于驱动内核的调度和计时器机制。处理时钟中断时若不切换栈可能会出现以下问题：

   - 时钟中断有可能在任何时刻打断正在执行的代码。如果在原有栈上处理，而该栈正好处于调用深度较深、数据结构复杂的时候，中断处理过程中调用的调度或调时函数会使用同一栈，可能导致栈空间不足或者破坏原有函数调用的栈帧。
   - 一些时钟中断处理需要快速执行且调用较多内核服务（例如调度决策、抢占检查等），为了保持这些服务的独立性和稳定性，使用独立的内核栈可以避免干扰正常运行任务的栈数据，同时也便于保证时钟中断处理的实时性和可靠性。

对于缺页异常，假设一个进程在用户态运行时因为访问某个内存区域触发缺页异常，而此区域刚好临近用户栈的边缘。如果不切换到独立内核栈，内核陷入异常时可能会在“坏的”用户栈上执行，这时如果内核需要动态分配栈空间来加载缺失页的内容，可能会进一步触发新的缺页异常或者栈溢出，整个异常链条失控；对于时钟中断，假设一个进程长时间运行后栈深度已接近内核栈的上限，此时突然发生时钟中断。如果继续使用原有栈处理时钟中断，调度程序等关键函数可能由于栈空间不足而无法完成状态保存、队列操作等工作，结果可能导致调度错误，甚至使当前进程信息被破坏，从而引发系统的不稳定性。