中山大學

SUN YAT-SEN UNIVERSITY

操作系统实验报告

实验三：内核线程与缺页异常

姓　　名：　刘家祥

学　　号：　23336152

教学班号：　计科二班

专　　业：　计算机科学与技术

院　　系：　计算机学院

2024~2025 学年第二学期

内核线程与缺页异常

一. 合并实验代码

在 `~/ysos` 目录下创建 `0x03` 文件夹，将 `0x02` 的代码复制到 `0x03` 目录下。

```
cp -rf ~/ysos/0x02/* ~/ysos/0x03/
```

随后使用 `git pull` 更新本地实验教程仓库之后，将 `src` 文件夹复制到 `0x03` 目录下，同名文件直接覆盖。

```
cp -rfT ~/ysos/YatSenOS-Tutorial-Volume-2/src/0x03/pkg/kernel ~/ysos/0x03/pkg/
kernel
```

二. 进程管理器的初始化

1. 文件 `pkg/kernel/src/lib.rs`

使用 `pub mod proc` 引用进程模块，并在 `crate::init` 函数中调用 `proc::init` 函数（位于内存初始化之后、启用中断之前）。

```
pub mod proc;
```

```
proc::init(); // init process manager
```

2. 文件 `pkg/kernel/src/proc/mod.rs`

实现 `init` 函数，创建内核进程。

```rust
/// 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 = {
        // 创建内核进程数据
        let proc_data = ProcessData::new();

        // 创建内核进程，强制使用 KERNEL_PID
        let process = Process::new_with_pid(
            KERNEL_PID,
            String::from("kernel"),
            None, // 内核进程没有父进程
            Some(proc_vm),
            Some(proc_data)
        );

        // 确认 PID 设置正确
        assert_eq!(process.pid(), KERNEL_PID, "Kernel PID mismatch");

        process
    };
    manager::init(kproc);

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

3. 文件 `pkg/kernel/src/proc/manager.rs`

实现 `init` 函数，设置初始状态为运行状态，设置当前 CPU 的 PID 为初始进程 PID，在最后添加 `print_process_list` 调用代码。

```rust
use alloc::{collections::*, format, sync::Arc};
pub fn init(init: Arc<Process>) {
    // 设置初始进程为运行状态
    init.write().status = ProgramStatus::Running;
    // 设置处理器的当前pid为初始进程的pid
    processor::set_pid(init.pid());
}
```

4. 文件 `pkg/kernel/src/proc/process.rs`

创建内核结构体，指定内核 PID 为 1，设置内核进程初始状态。这里创建一个新的方法 `new_with_pid`。

```
// ...
use crate::proc::vm::ProcessVm;
use alloc::sync::{Arc, Weak};
// ...
    pub status: ProgramStatus,
    pub context: ProcessContext,
    // ...
}

// 新增一个可以指定 PID 的创建方法
    pub fn new_with_pid(
        pid: ProcessId,
        name: String,
        parent: Option<Weak<Process>>,
        proc_vm: Option<ProcessVm>,
        proc_data: Option<ProcessData>,
    ) -> Arc<Self> {
        let name = name.to_ascii_lowercase();
        let proc_vm = proc_vm.unwrap_or_else(||
ProcessVm::new(PageTableContext::new()));
        let inner = ProcessInner {
            name,
            parent,
            status: ProgramStatus::Ready,
            context: ProcessContext::default(),
            ticks_passed: 0,
            exit_code: None,
            children: Vec::new(),
            proc_vm: Some(proc_vm),
            proc_data: Some(proc_data.unwrap_or_default()),
        };

        trace!("New process {}#{} created with specific PID.", &inner.name, pid);

        // 创建进程结构体，使用指定的 PID
        Arc::new(Self {
            pid,
            inner: Arc::new(RwLock::new(inner)),
        })
    }
```

三. 进程调度的实现

1. 修改时钟中断

移除计数器等模块, 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。

> **注意：** 为什么需要为时钟中断分配独立的栈空间? 尝试回答思考题 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**。

1.1. 时钟中断 (Timer Interrupt)

1.1.1. 问题场景

时钟中断是实现抢占式调度的基础, 它可能在任何指令执行时发生, 包括用户态程序或内核代码执行期间。此时，当前进程的栈可能已经非常接近其分配的边界（即将耗尽）。

1.1.2. 不切换栈的后果

时钟中断处理程序通常需要执行一些操作, 比如保存当前进程的上下文、选择下一个要运行的进程（调度）、恢复新进程的上下文等。这些操作都需要一定的栈空间。如果中断发生时，当前栈所剩空间非常小，中断处理程序自身的执行（包括函数调用、局部变量分配）就可能耗尽剩余的栈空间，导致栈溢出。这可能会触发缺页异常（如果栈下方有保护页）或更糟的是覆盖其他内存区域，导致难以追踪的错误和系统不稳定。

1.1.3. 切换栈的好处

为时钟中断分配独立的 IST 保证了无论当前进程的栈使用情况如何，时钟中断处理程序（以及后续的调度器）总是有充足、干净的栈空间来安全、可靠地执行其任务。这确保了内核调度的核心机制不会因为某个进程栈空间不足而崩溃。

1.2. 代码实现：`pkg/kernel/src/memory/gdt.rs`

包括添加时钟中断的 IST 索引、修改 IST_SIZES 来为时钟中断添加栈空间和实现时钟中断的 IST 栈等内容。

```
// 添加时钟中断的IST索引
pub const TIMER_IST_INDEX: u16 = 2;
// 修改IST_SIZES为4个元素，为时钟中断添加栈空间
pub const IST_SIZES: [usize; 4] = [0x1000, 0x1000, 0x1000, 0x1000];
...
// 添加时钟中断的IST栈
  tss.interrupt_stack_table[TIMER_IST_INDEX as usize] = {
      const STACK_SIZE: usize = IST_SIZES[3];
      static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE]; // 定义静态栈数组
      let stack_start = VirtAddr::from_ptr(addr_of_mut!(STACK)); // 获取栈底虚拟地址
      let stack_end = stack_start + STACK_SIZE as u64; // 计算栈顶虚拟地址
      info!( // 记录日志
          "Timer IST        : {:#018x}-{:#018x}", // 使用更清晰的格式化
          stack_start.as_u64(),
          stack_end.as_u64()
      );
      stack_end
  };
```

1.3. 代码实现：`pkg/kernel/src/interrupt/clock.rs`

包括导入 IST 索引常量，为时钟中断设置专用 IST 栈，设计实现中断处理逻辑等等。

```
// 导入时钟中断IST索引常量
use crate::memory::gdt::TIMER_IST_INDEX;
...
pub unsafe fn register_idt(idt: &mut InterruptDescriptorTable) {
    unsafe {
        idt[Interrupts::IrqBase as u8 + Irq::Timer as u8]
            .set_handler_fn(clock_handler) // 设置处理函数
            // 设置专用 IST 栈
            .set_stack_index(TIMER_IST_INDEX); // 使用导入的常量
    }
}

// 实际的时钟中断处理逻辑
pub extern "C" fn clock(mut context: ProcessContext) {
    // 在这里调用进程切换函数
    crate::proc::switch(&mut context);

    // 发送中断结束信号
    super::ack();
}
```

```
// 使用 as_handler 宏生成真正的中断处理函数入口点
as_handler!(clock);
```

2. 文件 `pkg/kernel/src/proc/mod.rs`

补全 `switch` 函数的实现。

```
pub fn switch(context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let process_manager = get_process_manager();
        // 保存当前进程上下文
        process_manager.save_current(context);

        // 获取当前进程
        let current_process = process_manager.current();
        let current_pid = current_process.pid();

        // 使用读取锁检查进程状态
        let status = current_process.read().status();

        // 如果进程不是 Dead 或 Blocked 状态，将其添加回就绪队列
        if status != ProgramStatus::Dead && status != ProgramStatus::Blocked {
            // 将当前进程（如果适用）放回就绪队列
            process_manager.push_ready(current_pid);

            // 获取写锁并更新状态和 tick
            let mut proc_guard = current_process.write();
            proc_guard.pause(); // 设置为 Ready 状态
            proc_guard.tick();  // 更新进程的时间片计数
        }
        // else: 如果进程是 Dead 或 Blocked，则不放回队列，自然淘汰

        // 选择并切换到下一个进程
        let _next_pid = process_manager.switch_next(context); // switch_next 内部
会更新 context

    });
}
```

3. 文件 `pkg/kernel/src/proc/manager.rs`

利用进程管理器提供的功能，补全 `save_current` 和 `switch_next` 函数。(代码太长放最后了)

4. 文件 `pkg/kernel/src/proc/process.rs`

实现 `save` 和 `restore` 函数。

```
    /// 保存进程当前的上下文信息
    pub(super) fn save(&mut self, context: &ProcessContext) {
        // 克隆传入的上下文，保存到进程内部状态中
        self.context = context.clone();
    }

    /// 恢复进程的上下文信息到指定的 context 变量中
    pub(super) fn restore(&mut self, context: &mut ProcessContext) {
        // 将保存的上下文克隆回传出的 context 引用
        *context = self.context.clone();

        // 加载该进程的页表到 CR3 寄存器
        self.vm().page_table.load();

        // 将进程状态设置为 Running
        self.resume(); // resume 内部会将 status 设置为 ProgramStatus::Running
    }
```

四. 进程信息的获取

1. 文件 `pkg/kernel/src/proc/mod.rs`

补全 `env` 函数，使得外部函数可以获取到当前进程的环境变量。

```
pub fn env(key: &str) -> Option<String> {
    x86_64::instructions::interrupts::without_interrupts(|| {
        // 获取当前进程管理器 -> 获取当前进程 -> 获取读锁 -> 调用 env 方法
        get_process_manager().current().read().env(key)
    })
}
```

2. 文件 `pkg/kernel/src/utils/mod.rs`

补全 `wait` 函数。

```rust
/// 等待指定 PID 的进程结束
pub fn wait(pid: ProcessId) {
    loop {
        // 调用内核接口获取目标进程的退出码
        match crate::proc::get_exit_code(pid) {
            Some(_exit_code) => {
                // 如果返回 Some，说明进程已退出，跳出循环
                break;
            }
            None => {
                // 如果返回 None，说明进程尚未退出或不存在
                // 使用 hlt 指令让 CPU 进入低功耗状态，等待下一次中断唤醒
                // 这是一种简单的忙等待替代方案，避免空转浪费 CPU 资源
                x86_64::instructions::hlt();
                // 注意：实际应用中可能需要更复杂的等待机制，如信号量或条件变量
            }
        }
    }
}
```

3. 为 `ProcessManager` 添加获取进程退出码的函数

在 `pkg/kernel/src/proc/manager.rs` 中为 `ProcessManager` 添加 `get_exit_code` 函数。使用 `Option<isize>` 作为返回值：`None` 表示进程未退出或不存在，`Some(code)` 表示进程已退出，其退出码为 `code`。

```rust
    /// 获取指定 PID 进程的退出码
    pub fn get_exit_code(&self, pid: ProcessId) -> Option<isize> {
        // 尝试根据 PID 获取进程对象
        if let Some(proc) = self.get_proc(&pid) {
            // 获取进程内部数据的读锁，并调用其 exit_code 方法
            // Process::exit_code() 会进一步访问 ProcessInner 的 exit_code 字段
            proc.read().exit_code()
        } else {
            // 如果进程不存在于管理器中，返回 None
            None
        }
    }
```

4. 文件 `pkg/kernel/src/proc/process.rs`

补全 `ProcessInner` 结构体中的 `kill` 方法。

```rust
    /// 终止进程并设置退出码
    pub fn kill(&mut self, ret: isize) {
        // 记录退出码
        self.exit_code = Some(ret);

        // 将进程状态设置为 Dead
        self.status = ProgramStatus::Dead;

        // 清理资源：释放进程数据和虚拟内存管理结构
        // Option::take() 会取出 Some(value) 并留下 None
        self.proc_data.take();
        self.proc_vm.take();

        // 注意：子进程、打开的文件等其他资源可能需要在此处或之后由父进程/系统进行清理
    }
```

五. 内核线程的创建

在 `pkg/kernel/src/proc/manager.rs` 中补全 `spawn_kernel_thread` 方法。（代码太长放最后了）。

六. 缺页异常的处理

1. 文件 `pkg/kernel/src/interrupt/exception.rs`

重新定义缺页异常处理函数 `page_fault_handler`，使其调用进程管理器的处理函数。这里有一个 `fixme` 我选择获取发生 page fault 时获取当前进程的 PID。

2. 文件 `pkg/kernel/src/proc/manager.rs`

完善缺页异常处理函数 `handle_page_fault`，检查错误码并委托给当前进程处理。

```rust
    /// 处理缺页异常
    /// addr: 发生缺页的虚拟地址
    /// err_code: 页面错误码
    /// 返回值: true 表示成功处理，false 表示处理失败（如权限错误）
    pub fn handle_page_fault(&self, addr: VirtAddr, err_code: PageFaultErrorCode)
→ bool {
        // 获取当前正在运行的进程
        let current_process = self.current();

        // 检查错误码是否包含保护违规（Protection Violation）
        // 这通常意味着尝试写入只读页面、在用户态访问内核页面等权限问题
        if err_code.contains(PageFaultErrorCode::PROTECTION_VIOLATION) {
            warn!( // 记录警告日志
                "Process {}#{} triggered protection violation page fault at
{:#x}, error code: {:?}",
                current_process.read().name(), // 获取进程名
                current_process.pid(),         // 获取进程 PID
                addr,
                err_code
            );
            // 对于保护违规，通常无法恢复，直接返回 false
            return false;
        }

        // 对于其他类型的缺页（如页面不存在 PRESENT=0），尝试让进程自己处理
        // 获取进程内部数据的写锁
        let mut proc_inner = current_process.write();

        // 调用进程内部的 handle_page_fault 方法
        // 这个方法通常会委托给 ProcessVm 来处理，例如尝试分配物理页并映射
        proc_inner.handle_page_fault(addr)
    }
```

3. 文件 `pkg/kernel/src/proc/vm/stack.rs`

在进程的缺页异常处理函数（`ProcessInner::handle_page_fault`）内部，如果判断缺页发生在栈区域且是合法的栈增长请求，则调用 `Stack::grow`（或类似方法）来处理。

补全 `Stack::grow` 函数（或类似功能）的实现逻辑：（**代码太长放最后了**）

3.1. 计算需要映射的新页面的范围：从包含 `addr` 的页面（`Page::containing_address(addr)`）一直到当前栈已分配区域的底部（`self.range.start`）。

3.2. 使用物理帧分配器 `alloc` 分配足够的物理帧。

3.3. 使用页表映射器 `mapper` 的 `map_range`（或类似功能）将这些物理帧映射到对应的虚拟页面范围，并设置正确的页表项标志（可写 `WRITABLE`、用户态可访问 `USER_ACCESSIBLE` 等）。

3.4. 更新 `Stack` 结构体的信息：将 `self.range.start` 更新为新分配范围的最底部的页面，并增加 `self.usage`（记录栈占用的总页面数）。

3.5. 成功时返回 `true`，如果分配或映射失败则返回 `false`。

七. 进程的退出

进程退出涉及以下几个关键步骤：

1. 1. 设置状态与清理资源

当一个进程需要退出时（例如调用 `exit` 系统调用），最终会触发 `ProcessInner::kill(ret: isize)` 方法。

- 调用链：`syscall::exit` -> `proc::process_exit` -> `ProcessManager::kill_current` -> `ProcessManager::kill` -> `Process::kill` -> `ProcessInner::kill`
- `ProcessInner::kill` 的核心操作：
  - `self.exit_code = Some(ret);`：保存退出码 `ret`。
  - `self.status = ProgramStatus::Dead;`：将进程状态标记为 `Dead`。
  - `self.proc_data.take();`：移除进程特定的数据（`ProcessData`）。
  - `self.proc_vm.take();`：移除进程的虚拟内存管理结构（`ProcessVm`），这通常会触发相关页表的清理（取决于 `ProcessVm` 的 `Drop` 实现）。

```rust
// pkg/kernel/src/proc/process.rs
// 在 ProcessInner::kill 方法中:
pub fn kill(&mut self, ret: isize) {
    // 设置退出码
    self.exit_code = Some(ret); // <--- 存储退出码

    // 设置状态为死亡
    self.status = ProgramStatus::Dead; // <--- 设置状态为 Dead

    // 释放不再需要的资源 (进程数据和虚拟内存)
    self.proc_data.take(); // <--- 删除 ProcessData
    self.proc_vm.take();   // <--- 删除 ProcessVm
}
```

2. 2. 防止进程被再次调度

调度器在选择下一个要运行的进程时，必须跳过状态为 `Dead` 的进程。

- 在 `proc::switch` 函数中：当保存完当前进程上下文后，检查其状态。如果不是 `Dead` 或 `Blocked`，才将其 `pid` 添加回 `ready_queue`。这意味着 `Dead` 进程不会再回到就绪队列。

```rust
// pkg/kernel/src/proc/mod.rs (switch 函数内)
// ... 保存上下文 ...
let status = current_process.read().status();
if status != ProgramStatus::Dead && status != ProgramStatus::Blocked {
    process_manager.push_ready(current_pid); // <--- Dead 进程不会执行这里
    // ... 更新状态为 Ready ...
}
// ... 切换到下一个进程 ...
```

- 在 `ProcessManager::switch_next` 函数中：从 `ready_queue` 取出 `pid` 后，获取进程并检查其状态。只有状态为 `Ready` 的进程才会被恢复执行。这是一个双重保险。

```rust
// pkg/kernel/src/proc/manager.rs (switch_next 循环内)
if let Some(next_process) = self.get_proc(&next_pid) {
    let mut next_inner = next_process.write();
    if next_inner.status() == ProgramStatus::Ready { // <--- Dead 进程不会通过此检查
        next_inner.restore(context);
        // ... 设置 PID, 返回 ...
    }
    // ... 否则继续循环 ...
}
```

3. 3. 存储并允许查询退出码

退出码存储在 `ProcessInner` 的 `exit_code: Option<isize>` 字段中。

- 查询接口：`ProcessManager::get_exit_code(pid: ProcessId)` 函数提供了查询机制。它通过 `pid` 找到 `Process` 对象，读取其内部状态，并返回 `exit_code`。

```
// pkg/kernel/src/proc/manager.rs
pub fn get_exit_code(&self, pid: ProcessId) -> Option<isize> {
    if let Some(proc) = self.get_proc(&pid) {
        // 通过 Process 的 Deref 获取 RwLockGuard<ProcessInner>，再调用 exit_code()
        proc.read().exit_code() // <--- 读取存储的退出码
    } else {
        None // 进程不存在
    }
}
```

- 暴露给外部：这个 `get_exit_code` 方法通过 `pkg/kernel/src/proc/mod.rs` 中的公共接口暴露，最终被 `utils::wait` 等函数使用。

八. 思考题

1. 内核进程初始化状态

问题：为什么在初始化进程管理器时需要将内核进程置为“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？

回答：

在进程管理器初始化时，将内核进程（通常是 PID 为 1 或 0 的特殊进程）直接设置为 `Running` 状态，而不是 `Ready` 状态放入就绪队列，主要基于以下原因：

- 初始执行上下文的连续性：内核完成初始化代码（如内存管理、中断设置等）后，CPU 已经在执行内核代码。这个执行流本身就构成了第一个进程（内核进程）的初始执行上下文。将其标记为 `Running` 是对系统当前实际状态的直接反映。系统不需要经历一次“停止当前执行 -> 保存 -> 从队列取出内核进程 -> 加载 -> 运行”的复杂切换，因为它已经在运行了。
- 首次上下文切换的简化：当第一个调度事件（如时钟中断）发生时，调度器需要：a) 保存**当前正在运行**进程的上下文；b) 选择下一个进程；c) 恢复下一个进程的上下文。如果内核进程被标记为 `Running`，调度器在第一次触发时，可以明确地知道要保存哪个进程（内核进程）的上下文。如果内核进程仅是 `Ready` 状态在队列中，那么在第一次调度时，“当前正在运行”的是什么？调度器应该保存谁的上下文？这就需要额外的逻辑来处理这个“首次运行”的特殊情况。

- 避免引导阶段的复杂性：将初始执行流直接视为 `Running` 的内核进程，简化了从无进程管理到有进程管理的过渡。如果将其放入队列，调度器在首次运行时需要处理“当前无运行进程，从队列启动第一个进程”的逻辑，增加了引导过程的复杂度。

如果尝试将内核进程置为 `Ready` 并放入队列，可能遇到的问题：

- 引导逻辑复杂化

如上所述，调度器需要特殊处理第一次调度：识别出当前没有 `Running` 进程，然后从队列中取出内核进程并加载其上下文，而不是执行标准的“保存当前 -> 加载下一个”流程。

- 首次调度可能失败或不一致

调度器的 `switch_next` 函数通常假设存在一个可以被替换掉的“当前”进程。如果启动时没有 `Running` 进程，`switch_next` 可能无法正确执行其保存步骤，或者需要额外的分支来处理这种情况。

- 潜在的调度器空转或死锁

如果调度器设计不当，且启动时就绪队列仅包含内核进程（状态为 `Ready`），调度器在查找 `Running` 进程失败后，可能会错误地认为无事可做而进入空闲（`hlt`）或死循环，而不是主动加载队列中的 `Ready` 进程。

2. `Deref` 和 `DerefMut` 的应用

问题：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？

回答：

在 `pkg/kernel/src/proc/process.rs` 文件中，`Deref` 和 `DerefMut` trait 的实现是为了利用 Rust 的解引用强制多态（Deref Coercion）特性，以简化对嵌套数据结构的访问，提高代码的可读性和人体工程学。具体来说：

2.1. `impl core::ops::Deref for Process`

- **目标类型**：`Arc<RwLock<ProcessInner>>`
- **作用**：这个实现允许将一个 `Process` 类型的实例（它本质上是一个包含 `pid` 和 `inner: Arc<RwLock<ProcessInner>>` 的结构体）直接当作其内部的 `Arc<RwLock<ProcessInner>>` 来使用。
- **便利性**：当有一个 `process: Arc<Process>` 时，可以直接调用 `process.read()` 或 `process.write()` 来获取内部 `ProcessInner` 的读写锁守卫，而不需要写成 `process.inner.read()` 或 `process.inner.write()`。这隐藏了 `inner` 字段的访问细节。

2.2. `impl core::ops::Deref for ProcessInner`

- **目标类型**：`ProcessData`

- **作用**：这个实现允许将一个 `ProcessInner` 类型的**不可变**引用（通常是通过 `RwLockReadGuard<ProcessInner>` 间接获得）直接当作其内部 `proc_data: Option<ProcessData>` 字段解包后的 `ProcessData` 来使用。注意，这通常涉及到 `.as_ref().expect(...)` 来处理 `Option`。
- **便利性**：当持有一个 `guard: RwLockReadGuard<ProcessInner>` 时，可以直接通过 `guard.some_field_in_process_data` 来访问 `ProcessData` 中的字段，而不需要写成 `guard.proc_data.as_ref().expect("Process data missing").some_field_in_process_data`。这极大地简化了对进程特定数据的只读访问。

2.3. `impl core::ops::DerefMut for ProcessInner`

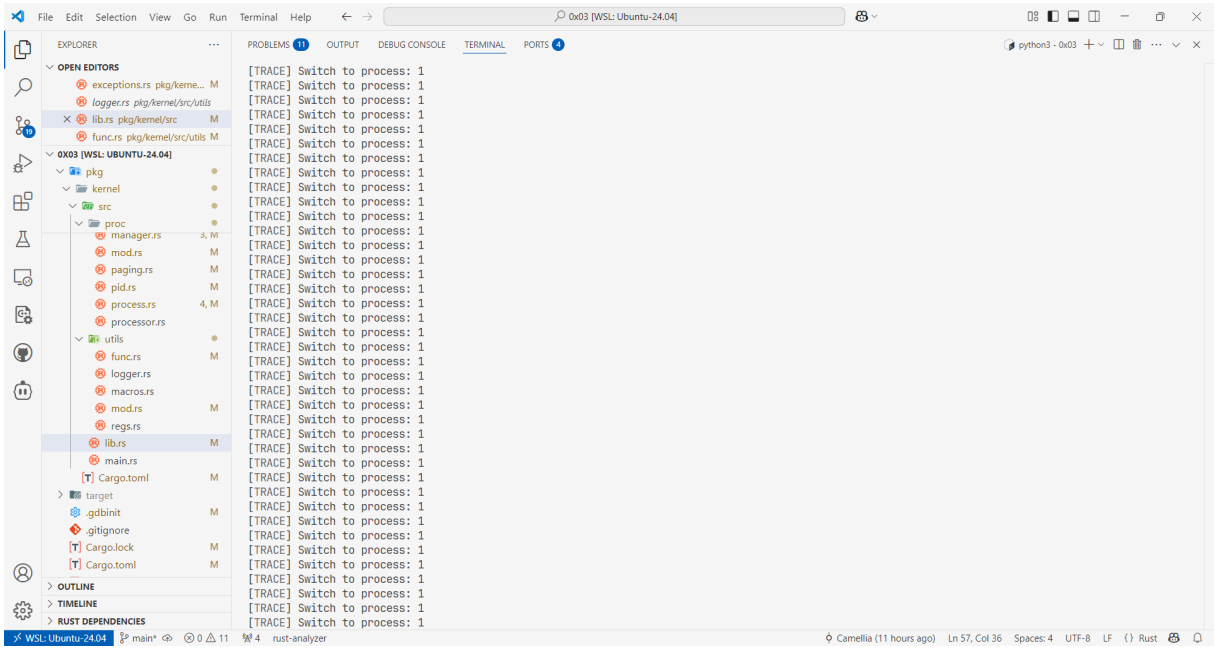
- **目标类型**：`ProcessData`
- **作用**：与上一个类似，但用于**可变**访问。它允许将一个 `ProcessInner` 类型的**可变**引用（通常通过 `RwLockWriteGuard<ProcessInner>` 间接获得）直接当作其内部 `proc_data: Option<ProcessData>` 字段解包后的 `ProcessData` 来进行修改。同样涉及到 `.as_mut().expect(...)`。
- **便利性**：当持有一个 `mut guard: RwLockWriteGuard<ProcessInner>` 时，可以直接通过 `guard.some_field_in_process_data = new_value;` 来修改 `ProcessData` 中的字段，而不需要写成 `guard.proc_data.as_mut().expect("Process data missing").some_field_in_process_data = new_value;`。这简化了对进程特定数据的可变访问。

2.4. 总体收益

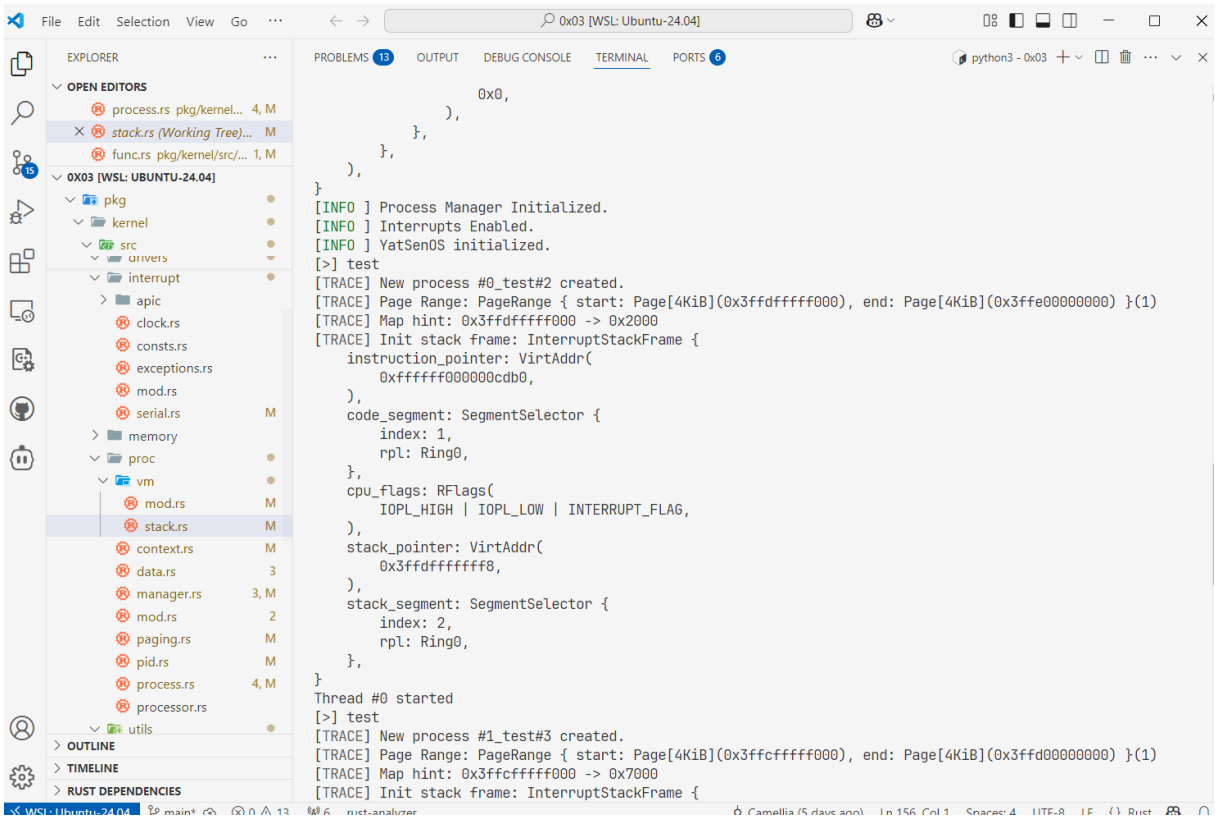
- **减少样板代码**：避免了重复的 `.inner`，`.proc_data`，`.as_ref()`，`.as_mut()`，`.expect()` 调用链。
- **提高可读性**：代码更接近直接访问所需数据的意图，隐藏了底层的封装（`Arc`，`RwLock`，`Option`）。
- **改善人体工程学**：使得使用这些嵌套结构更加方便自然。

这种利用 `Deref` 和 `DerefMut` 的模式是 Rust 中常见的提高代码表达力和减少冗余的技术。

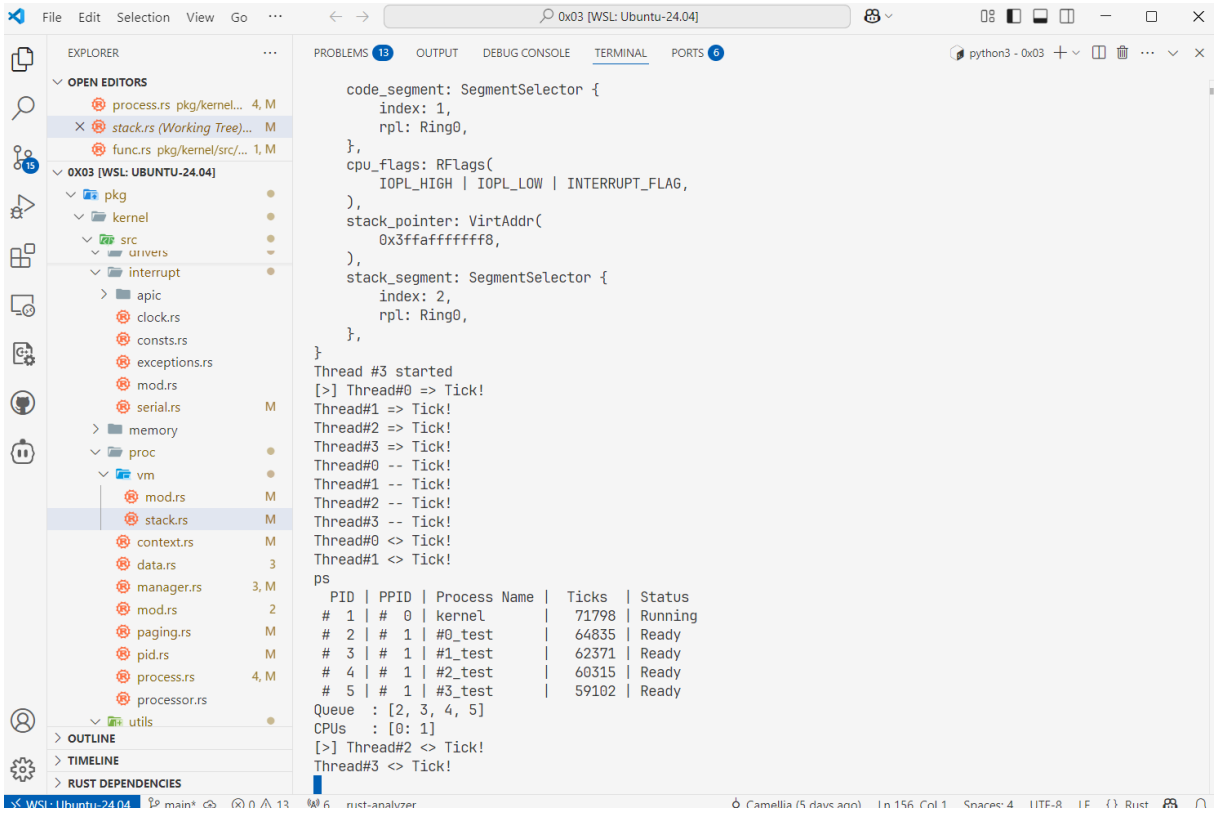
九. 阶段成果展示



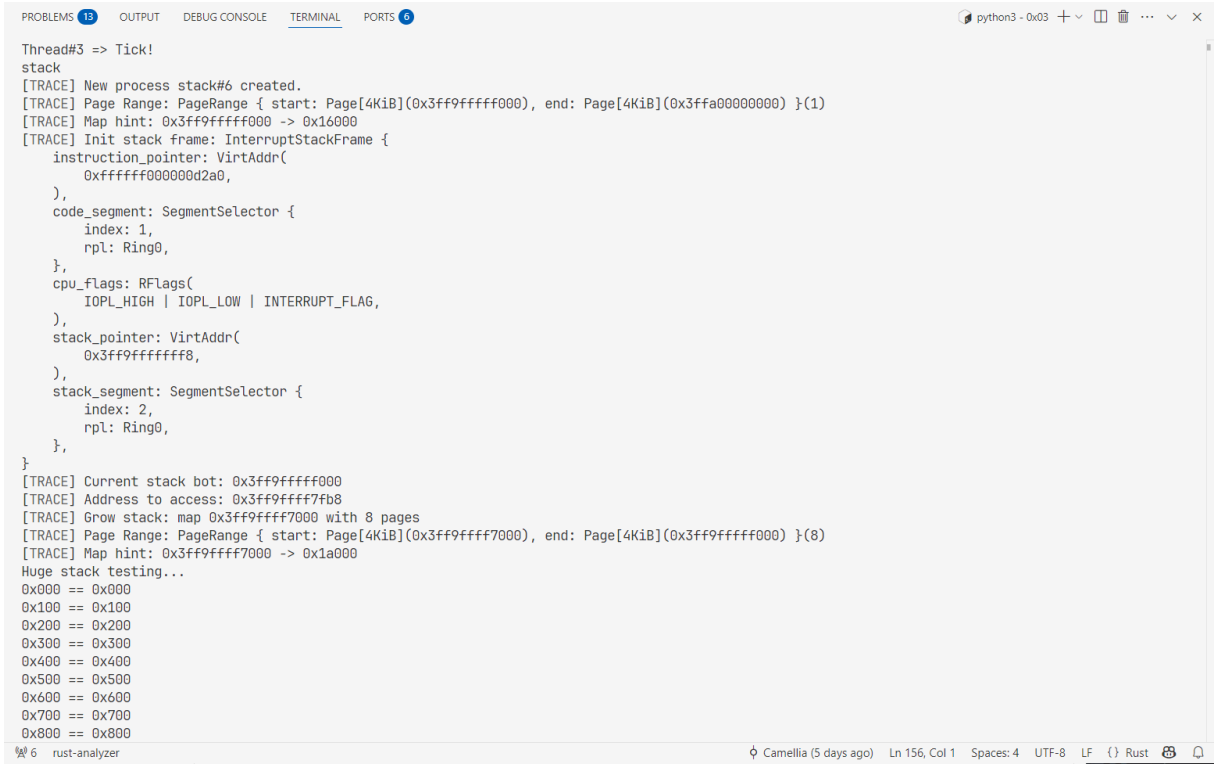
可以观察到内核线程（即进程 1 在被不断调度）。

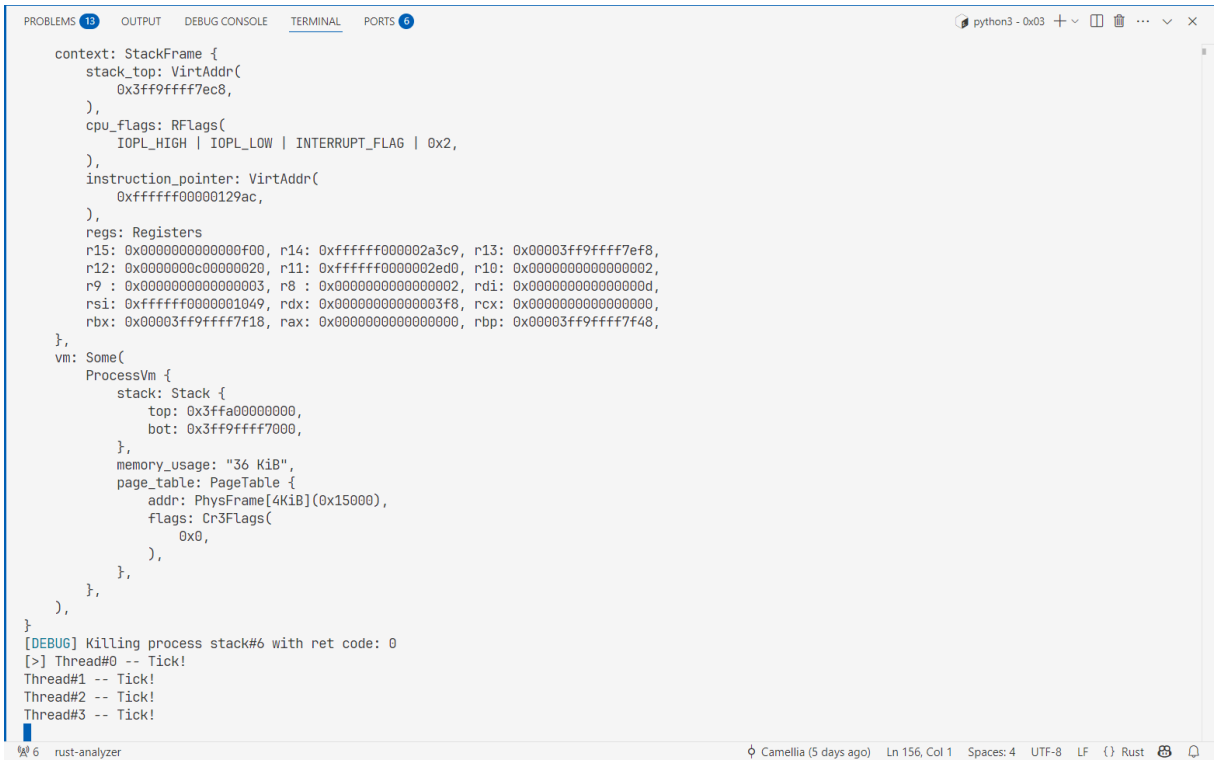


可以看到通过 `test` 指令可以成功创建新的进程并显示其页面映射地址。



可以看出 `ps` 指令可以列出当前的进程的 ID、PID 等内容。





- `[TRACE] Current stack bot: 0x3ff8fffff000` 和 `[TRACE] Address to access: 0x3ff8ffff7fb8` 从可见进程尝试访问当前栈底以下的地址，这通常会触发缺页异常。
- 从 `[TRACE] Grow stack: map 0x3ff8ffff7000 with 8 pages` 可见系统正确处理了缺页异常，通过分配 8 个新页来扩展栈空间，而不是崩溃。这表明系统能够处理大的栈需求。
- `Huge stack testing...` 和随后的 `0x... == 0x...` 输出: 表明内核线程正在运行其逻辑，并且能够访问其扩展后的栈空间。
- `[DEBUG] Killing process stack#7 with ret code: 0` 进程“stack”正常结束（退出码为 0），而不是因为栈溢出或其他错误而崩溃。
- `[>]:` 命令提示符再次出现，这暗示了 stack 命令等待了进程 stack#7 结束之后才返回。如果命令没有等待，提示符可能会在进程结束之前就出现。

十. 长代码展示

1. `manager.rs` 的保存上下文和切换进程部分

```rust
pub fn save_current(&self, context: &ProcessContext) {
        // 获取当前进程
        // 获取当前运行进程的 PID
        let current_pid = processor::get_pid();
        // 根据 PID 获取进程对象
        let current_process = self.get_proc(&current_pid).expect("Current process
not found");

        // 获取进程内部数据的写锁
        let mut proc_inner = current_process.write();
        // 调用进程内部的 save 方法保存上下文
        proc_inner.save(context);
        // 注意：save_current 不再负责将进程放回就绪队列，
        // 这个逻辑移到了 switch 函数中，以确保状态检查和入队操作原子地进行。
    }

    pub fn switch_next(&self, context: &mut ProcessContext) -> ProcessId {
        // 获取就绪队列的互斥锁
        let mut ready_queue = self.ready_queue.lock();

        // 循环从就绪队列前端取出 PID
        while let Some(next_pid) = ready_queue.pop_front() {
            // 尝试获取该 PID 对应的进程对象
            if let Some(next_process) = self.get_proc(&next_pid) {
                // 获取进程内部数据的写锁
                let mut next_inner = next_process.write();

                // 检查进程状态是否为 Ready
                if next_inner.status() == ProgramStatus::Ready {
                    // 如果是 Ready，则恢复其上下文
                    next_inner.restore(context); // restore 会设置状态为 Running 并
加载页表

                    // 更新当前 CPU 正在运行的 PID
                    processor::set_pid(next_pid);

                    // 成功切换，释放锁并返回新的 PID
                    drop(next_inner); // 释放写锁
                    drop(ready_queue); // 释放队列锁
                    return next_pid;
                }
```

```
                // else: 如果进程不是 Ready (可能是 Dead 或 Blocked)，则忽略它，继续循环
查找下一个
                // 锁 next_inner 会在此处自动释放
            }
            // 如果 get_proc 返回 None (进程已不存在)，也继续循环
        }
        // 循环结束，说明就绪队列为空或所有队列中的进程都不是 Ready 状态

        // 就绪队列为空，没有用户进程可调度，切换回内核进程
        // 注意：此时 ready_queue 的锁已被释放

        // 获取内核进程对象
        let kernel_process = self.get_proc(&KERNEL_PID).expect("Kernel process
not found");
        let mut kernel_inner = kernel_process.write(); // 获取写锁

        // 恢复内核进程的上下文
        kernel_inner.restore(context);

        // 更新当前 CPU 的 PID 为内核 PID
        processor::set_pid(KERNEL_PID);

        // 释放锁并返回内核 PID
        drop(kernel_inner);
        KERNEL_PID // 返回内核进程 PID
    }
```

2. `manager.rs` 中的 `spawn_kernel_thread` 方法

```
pub fn spawn_kernel_thread(
        &self,
        entry: VirtAddr,
        entry: VirtAddr, // 线程入口点虚拟地址
        name: String,    // 线程名称
        proc_data: Option<ProcessData>, // 可选的进程数据
    ) -> ProcessId {
        // 获取内核进程作为父进程。内核线程总是内核进程的子线程。
        let kproc = self.get_proc(&KERNEL_PID).expect("Kernel process not
found");
```

```
        // 内核线程共享内核的地址空间，因此克隆内核进程的页表。
        let page_table = kproc.read().clone_page_table();
        let proc_vm = Some(ProcessVm::new(page_table)); // 创建新的 ProcessVm 但使用
克隆的页表

        // 创建新的进程结构体，父进程是内核进程（使用 Weak 引用避免循环）
        let new_thread = Process::new(
            name,
            Some(Arc::downgrade(&kproc)), // 父进程是内核进程
            proc_vm,
            proc_data, // 使用传入的 proc_data
        );

        // 获取新线程的 PID
        let pid = new_thread.pid();

        // 为新线程分配初始栈空间
        let stack_top = new_thread.alloc_init_stack();

        // 获取新线程内部数据的写锁，以设置初始上下文
        let mut inner = new_thread.write();
        // 初始化栈帧，设置入口点（entry）和栈顶指针（stack_top）
        inner.context.init_stack_frame(entry, stack_top);
        drop(inner); // 释放写锁

        // 将新创建的线程添加到进程管理器的全局映射中
        self.add_proc(pid, new_thread.clone()); // clone 增加 Arc 引用计数

        // 将新线程加入就绪队列，等待调度器调度执行
        self.push_ready(pid);

        // 返回新进程的PID
        pid
    }
```

3. `stack.rs` 中补全 `grow_stack` 函数。

```
fn grow_stack(
        &mut self,
        addr: VirtAddr,
        mapper: MapperRef,
```

```
        alloc: FrameAllocatorRef,
    ) -> Result<(), MapToError<Size4KiB>> {
        debug_assert!(self.is_on_stack(addr), "Address is not on stack.");
        // 获取需要访问的页面
        let page = Page::containing_address(addr);

        // 计算需要新增的页面数量（每次增加32页，约128KB）
        let growth_pages = 32u64;
        // 确保不超过栈的最大页面数
        if self.usage + growth_pages > STACK_MAX_PAGES {
            return Err(MapToError::FrameAllocationFailed);
        }
        // 计算新的栈底页面
        let new_start_page = if page < self.range.start {
            // 如果缺页的地址在当前栈底以下，则以该页为新栈底
            page
        } else {
            // 否则保持当前栈底不变
            self.range.start
        };
        // 计算需要映射的页面范围
        let pages_to_map_count = self.range.start - new_start_page;
        if pages_to_map_count == 0 {
            // 如果不需要映射新页面，则返回成功
            return Ok(());
        }
        // 计算映射的起始地址
        let map_addr = new_start_page.start_address().as_u64();
        // 映射新的页面
        trace!(
            "Grow stack: map {:#x} with {} pages",
            map_addr,
            pages_to_map_count
        );
        let new_range = elf::map_range(map_addr, pages_to_map_count, mapper,
alloc)?;
        // 更新栈的范围和使用量
        self.range = PageRange {
            start: new_range.start,
            end: self.range.end,
        };
        self.usage += pages_to_map_count;

        Ok(())
    }
```