|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
| 北京交通大学软件学院  **《操作系统》课程**  **实验报告** | | |

|  |
| --- |
| **姓名: 张鑫成** |
| **学号: 20271055** |

目录

[1 实验目的 2](#_Toc29135)

[2 实验过程设计 3](#_Toc20557)

[3 源代码及注释 3](#_Toc10046)

[4 运行结果与分析 11](#_Toc19482)

[5 实验总结 12](#_Toc7598)

# 1 实验目的

（说明：写出本次实验的主要实验目的与原因。）

在上一节实现的操作系统还是比较原始，一个应用会独占 CPU 直到它出错或主动退出。操作系统还是以程序的一次执行过程（从开始到结束）作为处理器切换程序的时间段。为了提高效率,我们需要引入新的操作系统概念任务、任务切换、任务上下文，在此基础之上，通过任务切换的方式来增加应用运行效率。

# 2 实验过程设计

（说明：写出本次实验的主要实验流程或个人实验的实施过程。）

1. 实现多应用的内存放置
2. 通过手写汇编代码来实现\_\_switch，然后通过调用该函数来实现任务的切换
3. 结合实际使用，确定知道何时调用该函数，以及传入函数的两个参数 —— 分别代表正待换出和即将被换入的两条Trap控制流。

# 3 源代码及注释

（说明：写出本次实验使用的命令或编写的源代码。请在下列表格中说明源代码的文件名和代码功能概述或者命令名和该命令的主要作用。要求针对核心功能代码进行注释。）

表 3-1 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/build.py |
| **主要功能:** | 脚本定制工具:为每个应用定制了各自的链接脚本 |
| **源代码:**   1. # user/build.py 2. import os 3. base\_address = 0x80400000 4. step = 0x20000 5. linker = 'src/linker.ld' 6. app\_id = 0 7. apps = os.listdir('src/bin') 8. apps.sort() 9. for app in apps: 10. app = app[:app.find('.')] 11. lines = [] 12. lines\_before = [] 13. with open(linker, 'r') as f: 14. for line in f.readlines(): 15. lines\_before.append(line) 16. line=line.replace(hex(base\_address), hex(base\_address+step\*app\_id)) 17. lines.append(line) 18. with open(linker, 'w+') as f: 19. f.writelines(lines) 20. os.system('cargo build --bin %s --release' % app) 21. print('[build.py] application %s start with address %s' %(app, hex(base\_address+step\*app\_id))) 22. with open(linker, 'w+') as f: 23. f.writelines(lines\_before) 24. app\_id = app\_id + 1   /\*在循环中  第14~20行，找到src/linker.ld中的BASE\_ADDRESS=0x80400000;这一行，并将后面的地址替换为和当前应用对应的一个地址；  第21行，使用cargo build构建当前的应用，注意我们可以使用--bin参数来只构建某一个应用；  第23~24行,将src/linker.ld还原。  \*/ | |

表 3-2 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/loader.rs |
| **主要功能:** | 避免应用的内核初始化覆盖，调用load\_apps函数实现加载到不同的物理地址 |
| **源代码:**   1. // os/src/loader.rs 2. pub fn load\_apps() { 3. extern "C" { fn \_num\_app(); } 4. let num\_app\_ptr = \_num\_app as usize as \*const usize; 5. let num\_app = get\_num\_app(); 6. let app\_start = unsafe { 7. core::slice::from\_raw\_parts(num\_app\_ptr.add(1), num\_app + 1) 8. }; 9. // clear i-cache first 10. unsafe { asm!("fence.i" :::: "volatile"); } 11. // load apps 12. for i in 0..num\_app { 13. let base\_i = get\_base\_i(i); 14. // clear region 15. (base\_i..base\_i + APP\_SIZE\_LIMIT).for\_each(|addr| unsafe { 16. (addr as \*mut u8).write\_volatile(0) 17. });   // load app from data section to memory  let src = unsafe {  core::slice::from\_raw\_parts(  app\_start[i] as \*const u8,  app\_start[i + 1] - app\_start[i]  )  };  let dst = unsafe {  core::slice::from\_raw\_parts\_mut(base\_i as \*mut u8, src.len())  };  dst.copy\_from\_slice(src);  }  }  //计算第i个应用应该被加载的地址内存   1. fn get\_base\_i(app\_id: usize) -> usize { 2. APP\_BASE\_ADDRESS + app\_id \* APP\_SIZE\_LIMIT 3. } | |

表 3-3 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/task/switch.S |
| **主要功能:** | 在阶段 [1] 有两个参数分别是当前 A 任务上下文指针current\_task\_cx\_ptr和即将被切换到的 B 任务上下文指针next\_task\_cx\_ptr，  阶段[2]根据B任务上下文保存的内容来恢复ra寄存器、s0~s11寄存器以及sp寄存器。  阶段[3]通过恢复sp寄存器换到了任务 B 的内核栈上，进而实现了控制流的切换。 |
| 源代码:   1. ld s\n, (\n+2)\*8(a1) 2. .endm 3. .section .text 4. .globl \_\_switch 5. \_\_switch: 6. # \_\_switch( 7. # current\_task\_cx\_ptr: \*mut TaskContext, 8. # next\_task\_cx\_ptr: \*const TaskContext 9. # ) 10. # save kernel stack of current task 11. sd sp, 8(a0) 12. # save ra & s0~s11 of current execution 13. sd ra, 0(a0) 14. .set n, 0 15. .rept 12 16. SAVE\_SN %n 17. .set n, n + 1 18. .endr 19. # restore ra & s0~s11 of next execution 20. ld ra, 0(a1) 21. .set n, 0 22. .rept 12 23. LOAD\_SN %n 24. .set n, n + 1 25. .endr 26. # restore kernel stack of next task 27. ld sp, 8(a1) 28. ret | |

表 3-4 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/task/switch.rs |
| **主要功能:** | 将这段汇编代码中的全局符号\_\_switch解释为一个 Rust 函数  调用该函数来完成切换功能而不是直接跳转到符号 \_\_switch 的地址。 |
| **源代码:**   1. // os/src/task/switch.rs 2. global\_asm!(include\_str!("switch.S")); 3. use super::TaskContext; 4. extern "C" { 5. pub fn \_\_switch( 6. current\_task\_cx\_ptr: \*mut TaskContext, 7. next\_task\_cx\_ptr: \*const TaskContext 8. ); 9. } | |

表 3-5 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/syscall.rs |
| **主要功能:** | 用户库对应的实现和封装 |
| **源代码:**   1. // user/src/syscall.rs 2. pub fn sys\_yield() -> isize { 3. syscall(SYSCALL\_YIELD, [0, 0, 0])} | |

表 3-6 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | user/src/lib.rs |
| **主要功能:** | 用户库对应的实现和封装 |
| **源代码:**   1. // user/src/lib.rs 2. pub fn yield\_() -> isize { sys\_yield() } | |

表 3-7 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/task/task.rs |
| **主要功能:** | 定义每个应用的运行状态  定义保存信息 |
| **源代码:**  // os/src/task/task.rs   1. #[derive(Copy, Clone, PartialEq)]pub enum TaskStatus { 2. UnInit, // 未初始化 3. Ready, // 准备运行 4. Running, // 正在运行 5. Exited, // 已退出}   //通过 #[derive(...)] 可以让编译器为你的类型提供一些 Trait 的默认实现。   1. #[derive(Copy, Clone)]pub struct TaskControlBlock { 2. pub task\_status: TaskStatus, 3. pub task\_cx: TaskContext,}   //定义保存信息 | |

表 3-8 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/task/mod.rs |
| **主要功能:** | 作为全局的任务管理器管理这些用任务控制块描述的应用  重用并扩展之前初始化TaskManager的全局实例TASK\_MANAGER |
| **源代码:**   1. // os/src/task/mod.rs 2. pub struct TaskManager { 3. num\_app: usize, 4. inner: UPSafeCell<TaskManagerInner>, 5. } 6. struct TaskManagerInner { 7. tasks: [TaskControlBlock; MAX\_APP\_NUM], 8. current\_task: usize, 9. } 10. lazy\_static! { 11. pub static ref TASK\_MANAGER: TaskManager = { 12. let num\_app = get\_num\_app(); 13. let mut tasks = [ 14. TaskControlBlock { 15. task\_cx: TaskContext::zero\_init(), 16. task\_status: TaskStatus::UnInit 17. }; 18. MAX\_APP\_NUM 19. ]; 20. for i in 0..num\_app { 21. tasks[i].task\_cx = TaskContext::goto\_restore(init\_app\_cx(i)); 22. tasks[i].task\_status = TaskStatus::Ready; 23. } 24. TaskManager { 25. num\_app, 26. inner: unsafe { UPSafeCell::new(TaskManagerInner { 27. tasks, 28. current\_task: 0, 29. })}, 30. } 31. };}   第 12 行：调用loader子模块提供的get\_num\_app接口获取链接到内核的应用总数，后面会用到；  第 13~19 行：创建一个初始化的tasks数组，其中的每个任务控制块的运行状态都是UnInit：表示尚未初始化；  第 20~23 行：依次对每个任务控制块进行初始化，将其运行状态设置为Ready：表示可以运行，并初始化它的任务上下文；  从第 24 行开始：创建TaskManager实例并返回。 | |

表 3-9 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/syscall/process.rs |
| **主要功能:** | 实现 sys\_yield 和 sys\_exit 系统调用 |
| **源代码:**   1. // os/src/syscall/process.rs 2. use crate::task::suspend\_current\_and\_run\_next; 3. pub fn sys\_yield() -> isize { 4. suspend\_current\_and\_run\_next(); 5. 0} 6. use crate::task::exit\_current\_and\_run\_next; 7. pub fn sys\_exit(exit\_code: i32) -> ! { 8. println!("[kernel] Application exited with code {}", exit\_code); 9. exit\_current\_and\_run\_next(); 10. panic!("Unreachable in sys\_exit!");} | |

表 3-10 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/task/mod.rs |
| **主要功能:** | 1. 实现suspend\_current\_and\_run\_next和exit\_current\_and\_run\_next 2. 先修改当前应用的运行状态，然后尝试切换到下一个应用。 3. 实现方法run\_next\_task 4. 实现方法find\_next\_task 5. 初始化 |
| **源代码:**   1. // os/src/task/mod.rs 2. pub fn suspend\_current\_and\_run\_next() { 3. mark\_current\_suspended(); 4. run\_next\_task(); 5. } 6. pub fn exit\_current\_and\_run\_next() { 7. mark\_current\_exited(); 8. run\_next\_task(); 9. } 10. fn mark\_current\_suspended() { 11. TASK\_MANAGER.mark\_current\_suspended();} 12. fn mark\_current\_exited() { 13. TASK\_MANAGER.mark\_current\_exited();} 14. impl TaskManager { 15. fn mark\_current\_suspended(&self) { 16. let mut inner = self.inner.borrow\_mut(); 17. let current = inner.current\_task; 18. inner.tasks[current].task\_status = TaskStatus::Ready; 19. } 20. fn mark\_current\_exited(&self) { 21. let mut inner = self.inner.borrow\_mut(); 22. let current = inner.current\_task; 23. inner.tasks[current].task\_status = TaskStatus::Exited; 24. }} 25. fn run\_next\_task() { 26. TASK\_MANAGER.run\_next\_task();} 27. impl TaskManager { 28. fn run\_next\_task(&self) { 29. if let Some(next) = self.find\_next\_task() { 30. let mut inner = self.inner.exclusive\_access(); 31. let current = inner.current\_task; 32. inner.tasks[next].task\_status = TaskStatus::Running; 33. inner.current\_task = next; 34. let current\_task\_cx\_ptr = &mut inner.tasks[current].task\_cx as \*mut TaskContext; 35. let next\_task\_cx\_ptr = &inner.tasks[next].task\_cx as \*const TaskContext; 36. drop(inner); 37. // before this, we should drop local variables that must be dropped manually 38. unsafe { 39. \_\_switch( 40. current\_task\_cx\_ptr, 41. next\_task\_cx\_ptr, 42. ); 43. } 44. // go back to user mode 45. } else { 46. panic!("All applications completed!"); 47. } 48. }} 49. impl TaskManager { 50. fn find\_next\_task(&self) -> Option<usize> { 51. let inner = self.inner.exclusive\_access(); 52. let current = inner.current\_task; 53. (current + 1..current + self.num\_app + 1) 54. .map(|id| id % self.num\_app) 55. .find(|id| { 56. inner.tasks[\*id].task\_status == TaskStatus::Ready 57. }) 58. }} 59. for i in 0..num\_app { 60. tasks[i].task\_cx = TaskContext::goto\_restore(init\_app\_cx(i)); 61. tasks[i].task\_status = TaskStatus::Ready;} | |

表 3-6 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/task/context.rs |
| **主要功能:** | 初始化 |
| **源代码:**   1. impl TaskContext { 2. pub fn goto\_restore(kstack\_ptr: usize) -> Self { 3. extern "C" { fn \_\_restore(); } 4. Self { 5. ra: \_\_restore as usize, 6. sp: kstack\_ptr, 7. s: [0; 12], 8. } 9. }} | |

表 3-6 源代码及注释

|  |  |
| --- | --- |
| **文件名:** | os/src/loader.rs |
| **主要功能:** | 初始化 |
| **源代码:**   1. // os/src/loader.rs 2. pub fn init\_app\_cx(app\_id: usize) -> usize { 3. KERNEL\_STACK[app\_id].push\_context( 4. TrapContext::app\_init\_context(get\_base\_i(app\_id), USER\_STACK[app\_id].get\_sp()), 5. )} | |

# 4 运行结果与分析

（说明：此部分请写出代码或命令运行结果及其分析。运行结果可以通过截图的方式提交,此外需要对截图进行分析与说明。）

在labs目录下，使用git pull更新仓库。

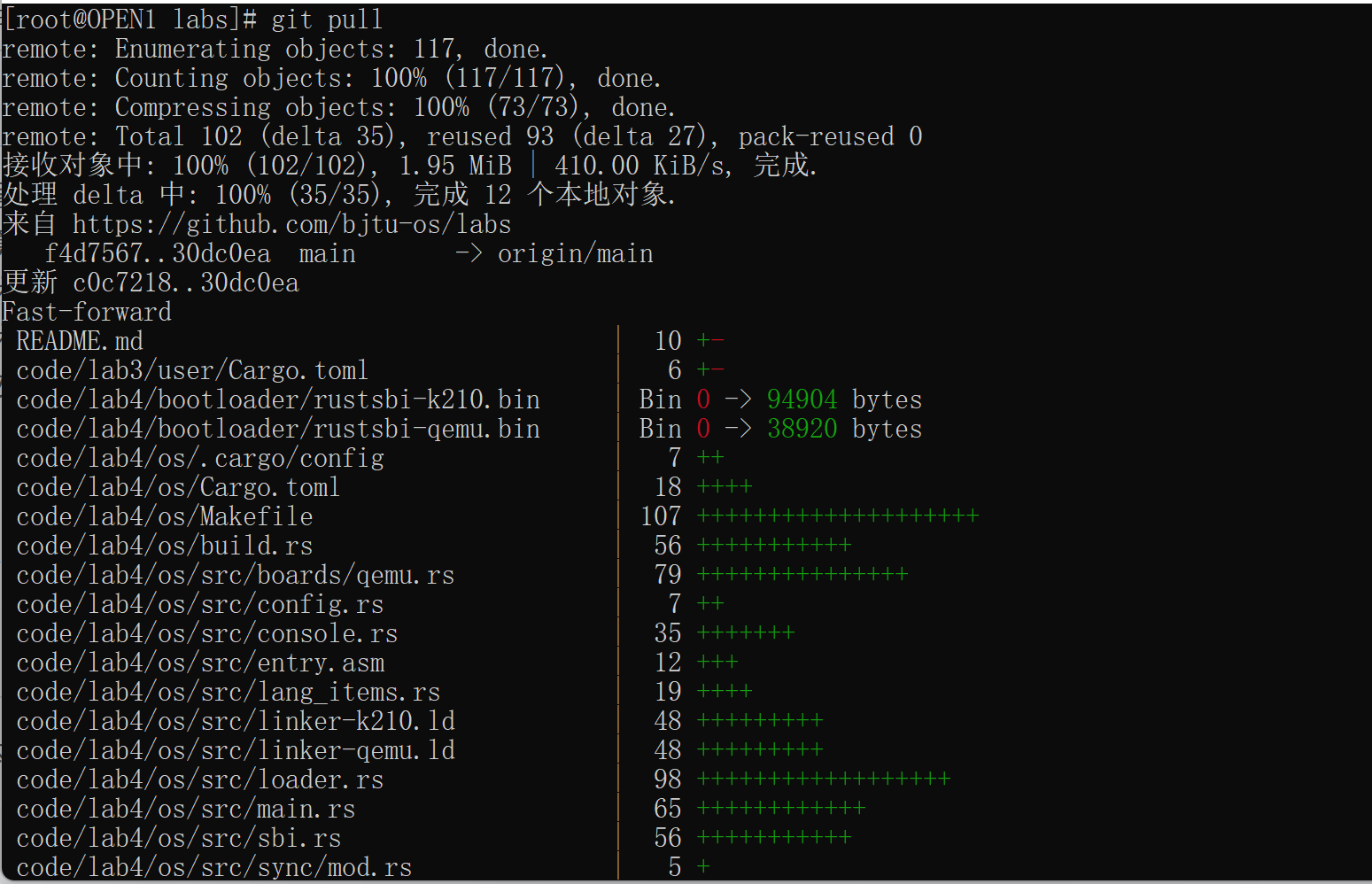


图 4.1 git pull仓库截图

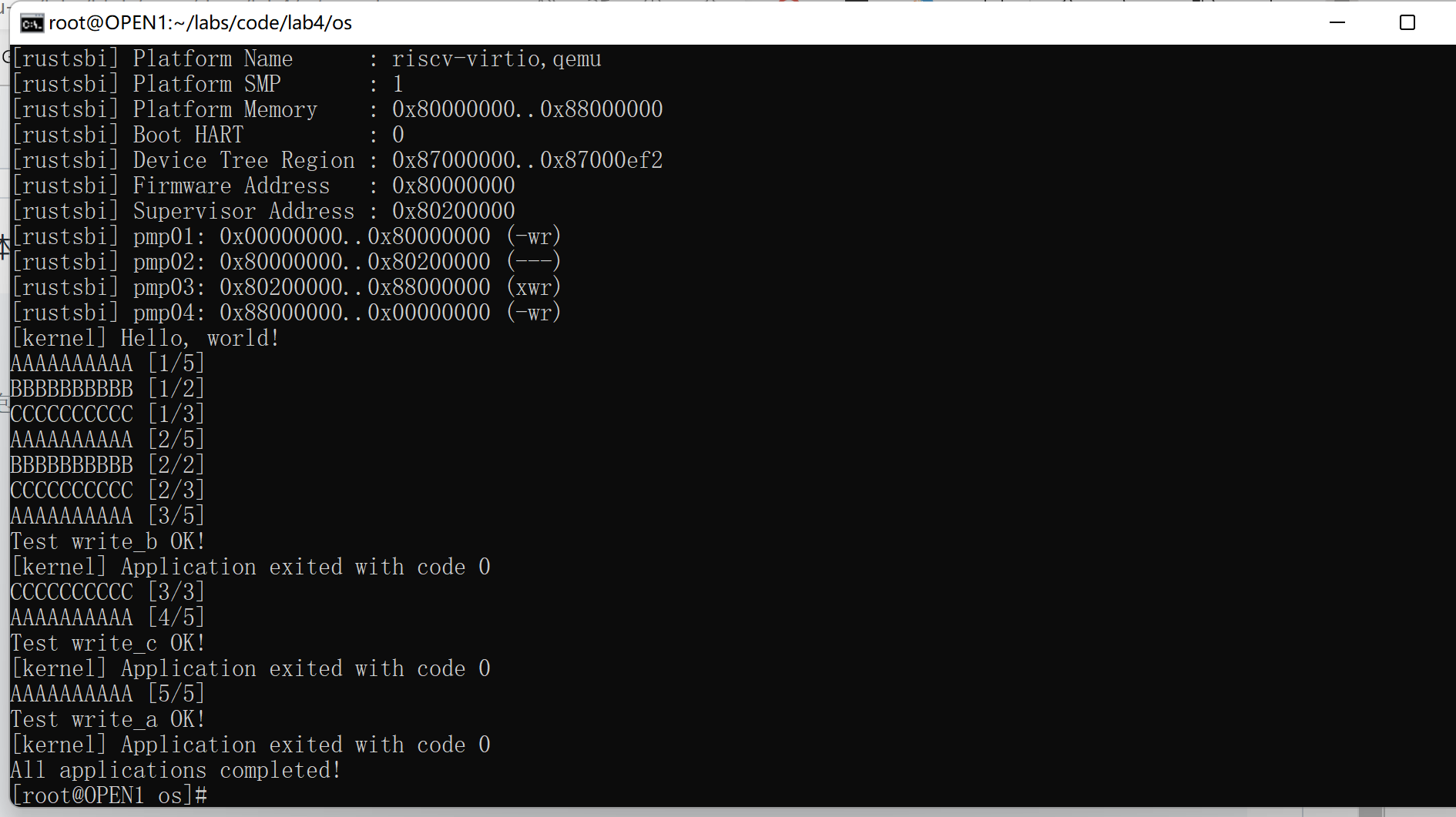


图4.2 构建并运行代码图

# 5 实验总结

（说明：此部分请写出针对本次实验的总结。也可以写出对于相同功能的不同实现方案，或是书写针对现有实验的改进方案等独创的且与本次实验有关的内容。）

上一节实现的操作系统比较原始：一个应用会独占 CPU 直到它出错或主动退出。操作系统还是以程序的一次执行过程（从开始到结束）作为处理器切换程序的时间段。

为了提高效率，引入新的操作系统概念任务、任务切换、任务上下文。

如果把应用程序执行的整个过程进行进一步分析，可以看到，当程序访问 I/O 外设或睡眠时，其实是不需要占用处理器的。

于是我们可以把应用程序在不同时间段的执行过程分为两类：占用处理器执行有效任务的计算阶段和不必占用处理器的等待阶段。这些阶段就形成了一个我们熟悉的“暂停 - 继续 …”组合的控制流或执行历史。从应用程序开始执行到结束的整个控制流就是应用程序的整个执行过程。

通过手写汇编代码来实现\_\_switch，然后通过调用该函数来实现任务的切换分为三个阶段：

1. 阶段[1]有两个参数分别是当前 A 任务上下文指针current\_task\_cx\_ptr和即将被切换到的 B 任务上下文指针next\_task\_cx\_ptr，
2. 阶段[2]根据B任务上下文保存的内容来恢复ra寄存器、s0~s11寄存器以及sp寄存器。
3. 阶段[3]通过恢复sp寄存器换到了任务 B 的内核栈上，进而实现了控制流的切换。

同时，在实际使用的时候，我们需要知道何时调用该函数，以及如何确定传入函数的两个参数 —— 分别代表正待换出和即将被换入的两条 Trap 控制流。

暂时考虑 CPU 只能单向地通过读取外设提供的寄存器信息来获取外设处理 I/O 的完成状态。多道程序的思想在于：内核同时管理多个应用。如果外设处理 I/O 的时间足够长，那我们可以先进行任务切换去执行其他应用；在某次切换回来之后，应用再次读取设备寄存器，发现 I/O 请求已经处理完毕了，那么就可以根据返回的 I/O 结果继续向下执行了。

这种任务切换，是让应用主动调用sys\_yield系统调用来实现的，这意味着应用主动交出 CPU 的使用权给其他应用。

一个应用会持续运行下去，直到它主动调用sys\_yield系统调用来交出 CPU 使用权。内核将很大的权力下放到应用，让所有的应用互相协作来最终达成最大化 CPU 利用率，充分利用计算资源这一终极目标。

通过实际编写，上述理论知识的实践，定义并使用任务控制块，定义全局的任务管理器，实现sys\_yield（暂时放弃对CPU的当前使用权）和sys\_exit（应用退出执行）系统调用，最后配置相关启动函数，最后实现任务切换。