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

|  |
| --- |
| **姓名:韩熔** |
| **学号:20301036** |

目录

[1 实验目的 3](#_Toc118133029)

[2 实验过程设计 3](#_Toc118133030)

[3 源代码及注释 3](#_Toc118133031)

[4 运行结果与分析 4](#_Toc118133032)

[5 实验总结 4](#_Toc118133033)

# 1 实验目的

本章展现了操作系统在性能上的一系列功能改进：

通过提前加载应用程序到内存，减少应用程序切换开销

通过协作机制支持程序主动放弃处理器，提高系统执行效率

通过抢占机制支持程序被动放弃处理器，提高不同程序对处理器资源使用的公平性，也进一步提高了应用对 I/O 事件的响应效率

# 2 实验过程设计

（1）了解协作式操作系统，执行多道程序的应用并观察结果

（2）学习多道程序放置与加载

（3）学习任务切换

（4）学习多道程序与协作式调度

# 3 源代码及注释

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 build.py ，为每个应用定制了各自的链接脚本linker.ld |
| **源代码:**  **import os**  **base\_address = 0x80400000**  **step = 0x20000**  **linker = 'src/linker.ld'**  **app\_id = 0**  **apps = os.listdir('src/bin')**  **apps.sort()**  **for app in apps:**  **app = app[:app.find('.')]**  **lines = []**  **lines\_before = []**  **with open(linker, 'r') as f:**  **#找到 src/linker.ld 中的 BASE\_ADDRESS = 0x80400000**  **for line in f.readlines():**  **lines\_before.append(line)**  **line = line.replace(hex(base\_address), hex(base\_address+step\*app\_id))**  **lines.append(line)**  **#将后面的地址替换为和当前应用对应的一个地址；**  **with open(linker, 'w+') as f:**  **f.writelines(lines)**  **#使用 cargo build 构建当前的应用**  **#使用 --bin 参数来只构建某一个应用**  **os.system('cargo build --bin %s --release' % app)**  **print('[build.py] application %s start with address %s' %(app, hex(base\_address+step\*app\_id)))**  **#将 src/linker.ld 还原**  **with open(linker, 'w+') as f:**  **f.writelines(lines\_before)**  **app\_id = app\_id + 1** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/loader.rs |
| **主要功能:** | 将应用加载入内存中 |
| **源代码:**  **//! Loading user applications into memory**  **//!**  **//! For chapter 3, user applications are simply part of the data included in the**  **//! kernel binary, so we only need to copy them to the space allocated for each**  **//! app to load them. We also allocate fixed spaces for each task's**  **//! [`KernelStack`] and [`UserStack`].**  **use crate::config::\*;**  **use crate::trap::TrapContext;**  **use core::arch::asm;**  **#[repr(align(4096))]**  **#[derive(Copy, Clone)]**  **struct KernelStack {**  **data: [u8; KERNEL\_STACK\_SIZE],**  **}**  **#[repr(align(4096))]**  **#[derive(Copy, Clone)]**  **struct UserStack {**  **data: [u8; USER\_STACK\_SIZE],**  **}**  **static KERNEL\_STACK: [KernelStack; MAX\_APP\_NUM] = [KernelStack {**  **data: [0; KERNEL\_STACK\_SIZE],**  **}; MAX\_APP\_NUM];**  **static USER\_STACK: [UserStack; MAX\_APP\_NUM] = [UserStack {**  **data: [0; USER\_STACK\_SIZE],**  **}; MAX\_APP\_NUM];**  **impl KernelStack {**  **fn get\_sp(&self) -> usize {**  **self.data.as\_ptr() as usize + KERNEL\_STACK\_SIZE**  **}**  **pub fn push\_context(&self, trap\_cx: TrapContext) -> usize {**  **let trap\_cx\_ptr = (self.get\_sp() - core::mem::size\_of::<TrapContext>()) as \*mut TrapContext;**  **unsafe {**  **\*trap\_cx\_ptr = trap\_cx;**  **}**  **trap\_cx\_ptr as usize**  **}**  **}**  **impl UserStack {**  **fn get\_sp(&self) -> usize {**  **self.data.as\_ptr() as usize + USER\_STACK\_SIZE**  **}**  **}**  **/// Get base address of app i.**  **//base\_i 的计算方式**  **fn get\_base\_i(app\_id: usize) -> usize {**  **APP\_BASE\_ADDRESS + app\_id \* APP\_SIZE\_LIMIT**  **}**  **/// Get the total number of applications.**  **pub fn get\_num\_app() -> usize {**  **extern "C" {**  **fn \_num\_app();**  **}**  **unsafe { (\_num\_app as usize as \*const usize).read\_volatile() }**  **}**  **/// Load nth user app at**  **/// [APP\_BASE\_ADDRESS + n \* APP\_SIZE\_LIMIT, APP\_BASE\_ADDRESS + (n+1) \* APP\_SIZE\_LIMIT).**  **///应用在内核初始化的时候被加载到不同的物理地址**  **pub fn load\_apps() {**  **extern "C" {**  **fn \_num\_app();**  **}**  **let num\_app\_ptr = \_num\_app as usize as \*const usize;**  **let num\_app = get\_num\_app();**  **let app\_start = unsafe { core::slice::from\_raw\_parts(num\_app\_ptr.add(1), num\_app + 1) };**  **// clear i-cache first**  **unsafe {**  **asm!("fence.i");**  **}**  **// load apps**  **//第i个应用被加载到以物理地址 base\_i 开头的一段物理内存上**  **for i in 0..num\_app {**  **let base\_i = get\_base\_i(i);**  **// clear region**  **(base\_i..base\_i + APP\_SIZE\_LIMIT)**  **.for\_each(|addr| unsafe { (addr as \*mut u8).write\_volatile(0) });**  **// 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);**  **}**  **}**  **/// get app info with entry and sp and save `TrapContext` in kernel stack**  **//先调用 init\_app\_cx 构造该任务的 Trap 上下文（包括应用入口地址和用户栈指针）并将其压入到内核栈顶**  **pub fn init\_app\_cx(app\_id: usize) -> usize {**  **KERNEL\_STACK[app\_id].push\_context(TrapContext::app\_init\_context(**  **get\_base\_i(app\_id),**  **USER\_STACK[app\_id].get\_sp(),**  **))**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/switch.S |
| **主要功能:** | 切换Trap 控制流 |
| **源代码:**  **.altmacro**  **.macro SAVE\_SN n**  **sd s\n, (\n+2)\*8(a0)**  **.endm**  **.macro LOAD\_SN n**  **ld s\n, (\n+2)\*8(a1)**  **.endm**  **.section .text**  **.globl \_\_switch**  **\_\_switch:**  **# 阶段 [1]**  **#函数原型中的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，它们分别通过寄存器 a0/a1 传入**  **# \_\_switch(**  **# current\_task\_cx\_ptr: \*mut TaskContext,**  **# next\_task\_cx\_ptr: \*const TaskContext**  **# )**  **# 阶段 [2]**  **#恢复 ra 寄存器、s0~s11 寄存器以及 sp 寄存器**  **# save kernel stack of current task**  **sd sp, 8(a0)**  **# save ra & s0~s11 of current execution**  **sd ra, 0(a0)**  **.set n, 0**  **.rept 12**  **SAVE\_SN %n**  **.set n, n + 1**  **.endr**  **# 阶段 [3]**  **# restore ra & s0~s11 of next execution**  **ld ra, 0(a1)**  **.set n, 0**  **.rept 12**  **LOAD\_SN %n**  **.set n, n + 1**  **.endr**  **# restore kernel stack of next task**  **ld sp, 8(a1)**  **# 阶段 [4]**  **ret** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/context.rs |
| **主要功能:** | TaskContext的实现 |
| **源代码:** **//! Implementation of [`TaskContext`]**  **/// Task Context**  **#[derive(Copy, Clone)]**  **#[repr(C)]**  **pub struct TaskContext {**  **/// return address ( e.g. \_\_restore ) of \_\_switch ASM function**  **ra: usize,**  **/// kernel stack pointer of app**  **sp: usize,**  **/// callee saved registers: s 0..11**  **s: [usize; 12],**  **}**  **impl TaskContext {**  **//接着调用 TaskContext::goto\_restore 来构造每个任务保存在任务控制块中的任务上下文**  **pub fn goto\_restore(kstack\_ptr: usize) -> Self {**  **extern "C" { fn \_\_restore(); }**  **Self {**  **//设置任务上下文中的内核栈指针将任务上下文的 ra 寄存器设置为 \_\_restore 的入口地址**  **ra: \_\_restore as usize,**  **sp: kstack\_ptr,**  **s: [0; 12],**  **}**  **}**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/switch.rs |
| **主要功能:** | 通过调用该函数来完成切换功能 |
| **源代码:**  **// os/src/task/switch.rs**  **global\_asm!(include\_str!("switch.S"));**  **use super::TaskContext;**  **extern "C" {**  **pub fn \_\_switch(**  **current\_task\_cx\_ptr: \*mut TaskContext,**  **next\_task\_cx\_ptr: \*const TaskContext**  **);**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | user/src/syscall.rs |
| **主要功能:** | 封装sys\_yield 标准接口，使应用主动交出 CPU 所有权并切换到其他应用 |
| **源代码:**  **pub fn sys\_yield() -> isize {**  **syscall(SYSCALL\_YIELD, [0, 0, 0])**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | user/src/lib.rs |
| **主要功能:** | 封装sys\_yield 标准接口，使应用主动交出 CPU 所有权并切换到其他应用 |
| **源代码:**  **pub fn yield\_() -> isize { sys\_yield() }** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/task.rs |
| **主要功能:** | 管理任务控制块与任务运行状态 |
| **源代码:**  **//通过 #[derive(...)] 可以让编译器为类型提供一些 Trait 的默认实现**  **#[derive(Copy, Clone, PartialEq)]**  **//在内核中对每个应用分别维护它的运行状态**  **pub enum TaskStatus {**  **UnInit, // 未初始化**  **Ready, // 准备运行**  **Running, // 正在运行**  **Exited, // 已退出**  **}**  **#[derive(Copy, Clone)]**  **//将应用的更多信息保存在任务控制块的数据结构中**  **//是内核管理应用的核心数据结构**  **pub struct TaskControlBlock {**  **pub task\_status: TaskStatus,**  **pub task\_cx: TaskContext,**  **}** | |

|  |  |
| --- | --- |
| **文件名:** | os/src/task/mod.rs |
| **主要功能:** | 管理用任务控制块描述的应用  实现suspend\_current\_and\_run\_next 和 exit\_current\_and\_run\_nex |
| **源代码:**  **pub struct TaskManager {**  **num\_app: usize,//** **表示任务管理器管理的应用的数目**  **//将 TaskManagerInner 包裹在 UPSafeCell 内以获取其内部可变性以及单核上安全的运行时借用检查能力。**  **inner: UPSafeCell<TaskManagerInner>,**  **}**  **struct TaskManagerInner {**  **tasks: [TaskControlBlock; MAX\_APP\_NUM],//** **任务控制块数组**  **current\_task: usize,// CPU 正在执行的应用编号**  **}**  **lazy\_static! {**  **pub static ref TASK\_MANAGER: TaskManager = {**  **//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**  **let num\_app = get\_num\_app();**  **//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**  **let mut tasks = [**  **TaskControlBlock {**  **task\_cx: TaskContext::zero\_init(),**  **task\_status: TaskStatus::UnInit**  **};**  **MAX\_APP\_NUM**  **];**  **//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；**  **for i in 0..num\_app {**  **tasks[i].task\_cx = TaskContext::goto\_restore(init\_app\_cx(i));**  **tasks[i].task\_status = TaskStatus::Ready;**  **}**  **//创建 TaskManager 实例并返回**  **TaskManager {**  **num\_app,**  **inner: unsafe { UPSafeCell::new(TaskManagerInner {**  **tasks,**  **current\_task: 0,**  **})},**  **}**  **};**  **}**  **//实现suspend\_current\_and\_run\_next 和 exit\_current\_and\_run\_nex**  **//都是先修改当前应用的运行状态，然后尝试切换到下一个应用**  **pub fn suspend\_current\_and\_run\_next() {**  **mark\_current\_suspended();**  **run\_next\_task();**  **}**  **pub fn exit\_current\_and\_run\_next() {**  **mark\_current\_exited();**  **run\_next\_task();**  **}**  **//修改当前应用的运行状态**  **fn mark\_current\_suspended() {**  **TASK\_MANAGER.mark\_current\_suspended();**  **}**  **fn mark\_current\_exited() {**  **TASK\_MANAGER.mark\_current\_exited();**  **}**  **//调用了全局任务管理器 TASK\_MANAGER 的 mark\_current\_suspended 方法**  **impl TaskManager {**  **//首先获得里层 TaskManagerInner 的可变引用**  **fn mark\_current\_suspended(&self) {**  **let mut inner = self.inner.borrow\_mut();**  **let current = inner.current\_task;**  **//然后根据其中记录的当前正在执行的应用 ID 对应在任务控制块数组 tasks 中修改状态**  **inner.tasks[current].task\_status = TaskStatus::Ready;**  **}**  **fn mark\_current\_exited(&self) {**  **let mut inner = self.inner.borrow\_mut();**  **let current = inner.current\_task;**  **inner.tasks[current].task\_status = TaskStatus::Exited;**  **}**  **}**  **//切换到下一个应用run\_next\_task 的实现**  **fn run\_next\_task() {**  **TASK\_MANAGER.run\_next\_task();**  **}**  **//使用任务管理器的全局实例 TASK\_MANAGER 的 run\_next\_task 方法**  **impl TaskManager {**  **//调用 find\_next\_task 方法尝试寻找一个运行状态为 Ready 的应用并返回其 ID**  **fn run\_next\_task(&self) {**  **if let Some(next) = self.find\_next\_task() {**  **let mut inner = self.inner.exclusive\_access();**  **let current = inner.current\_task;**  **inner.tasks[next].task\_status = TaskStatus::Running;**  **inner.current\_task = next;**  **let current\_task\_cx\_ptr = &mut inner.tasks[current].task\_cx as \*mut TaskContext;**  **let next\_task\_cx\_ptr = &inner.tasks[next].task\_cx as \*const TaskContext;**  **drop(inner);**  **// before this, we should drop local variables that must be dropped manually**  **unsafe {**  **\_\_switch(**  **current\_task\_cx\_ptr,**  **next\_task\_cx\_ptr,**  **);**  **}**  **// go back to user mode**  **} else {**  **panic!("All applications completed!");**  **}**  **}**  **}**  **//方法 find\_next\_task的实现**  **impl TaskManager {**  **fn find\_next\_task(&self) -> Option<usize> {**  **let inner = self.inner.exclusive\_access();**  **let current = inner.current\_task;**  **(current + 1..current + self.num\_app + 1)**  **.map(|id| id % self.num\_app)**  **.find(|id| {**  **//找到 current\_task 后面第一个状态为 Ready 的应用**  **inner.tasks[\*id].task\_status == TaskStatus::Ready**  **})**  **}**  **}**  **//** **task::run\_first\_task的实现**  **impl TaskManager {**  **fn run\_first\_task(&self) -> ! {**  **let mut inner = self.inner.exclusive\_access();**  **//取出即将最先执行的编号为 0 的应用的任务上下文指针 next\_task\_cx\_ptr 并希望能够切换过去**  **let task0 = &mut inner.tasks[0];**  **task0.task\_status = TaskStatus::Running;**  **let next\_task\_cx\_ptr = &task0.task\_cx as \*const TaskContext;**  **drop(inner);**  **let mut \_unused = TaskContext::zero\_init();**  **// before this, we should drop local variables that must be dropped manually**  **unsafe {**  **\_\_switch(**  **//在启动栈上分配了一个名为 \_unused 的任务上下文，并将它的地址作为第一个参数传给 \_\_switch**  **&mut \_unused as \*mut TaskContext,**  **next\_task\_cx\_ptr,**  **);**  **}**  **panic!("unreachable in run\_first\_task!");**  **}**  **pub fn run\_first\_task() {**  **TASK\_MANAGER.run\_first\_task();**  **}** | |

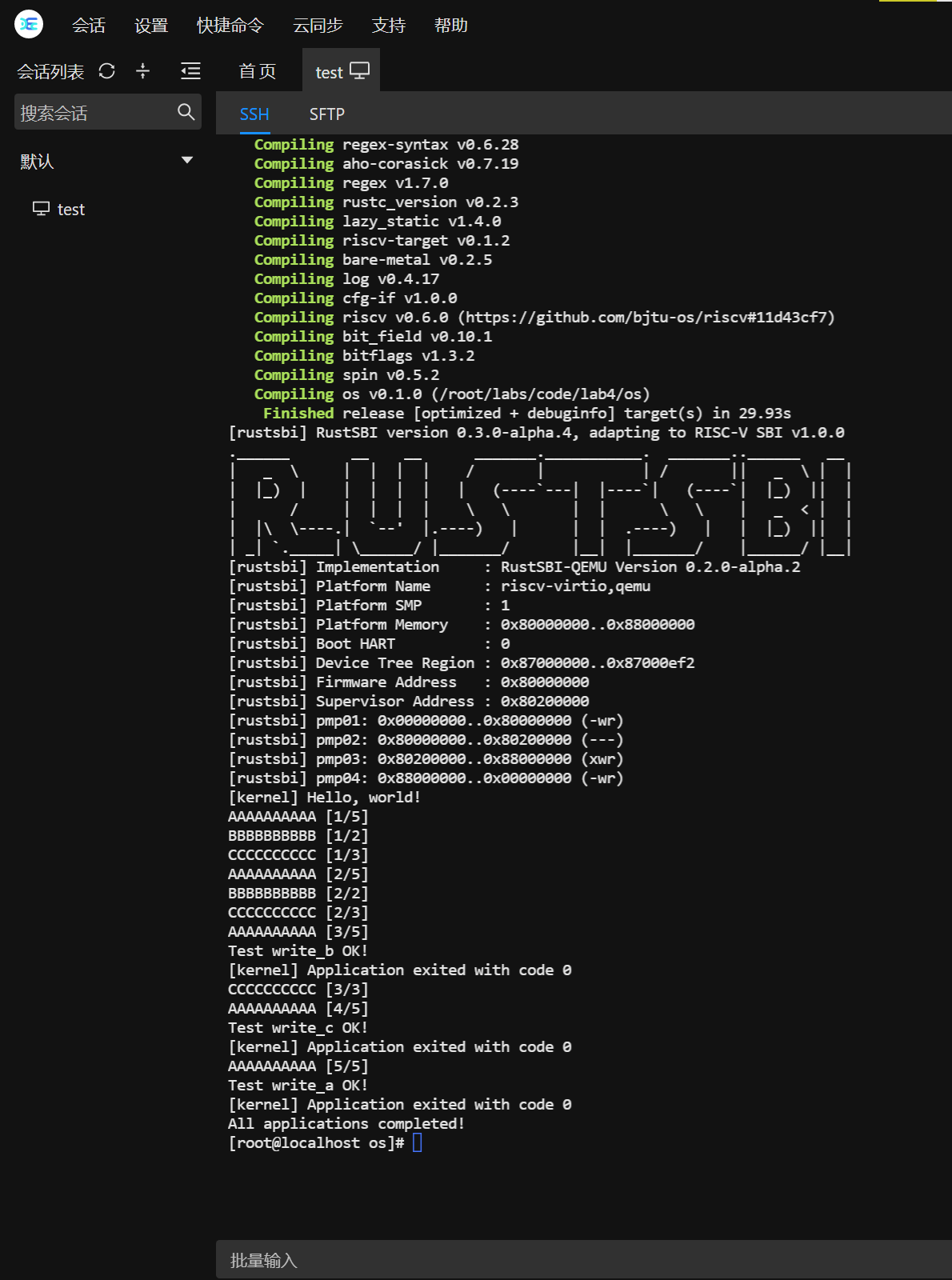
|  |  |
| --- | --- |
| **文件名:** | os/src/syscall/process.rs |
| **主要功能:** | 实现 sys\_yield 和 sys\_exit 系统调用 |
| **源代码:**  **use crate::task::suspend\_current\_and\_run\_next;**  **//** **sys\_yield 表示应用自己暂时放弃对CPU的当前使用权，进入 Ready 状态**  **pub fn sys\_yield() -> isize {**  **//用到了 task 子模块提供的 suspend\_current\_and\_run\_next 接口**  **//接口的作用是暂停当前的应用并切换到下个应用**  **suspend\_current\_and\_run\_next();**  **0**  **}**  **use crate::task::exit\_current\_and\_run\_next;**  **//** **sys\_exit 表示应用退出执行**  **pub fn sys\_exit(exit\_code: i32) -> ! {**  **println!("[kernel] Application exited with code {}", exit\_code);**  **//** 用到了**task 子模块提供的 exit\_current\_and\_run\_next 接口**  **//作用是退出当前的应用并切换到下个应用**  **exit\_current\_and\_run\_next();**  **panic!("Unreachable in sys\_exit!");**  **}** | |

表 3-2 关键命令解释

|  |  |  |
| --- | --- | --- |
| **序号** | **命令名** | **主要作用** |
| **1** | **make run -j $(nproc)** | 构建并运行代码 |

# 4 运行结果与分析

1. 更新代码lab4，构建并运行代码。多道程序的应用分别会输出一个不同的字母矩阵。当他们交替执行的时候，我们将看到字母行的交错输出。



# 5 实验总结

本节课实现了协作式操作系统，但任务调度的主动权在于应用程序的“自觉性”上，操作系统自身缺少强制的任务调度的手段。改进的方法就是设计一个更加公平和高效交互的抢占式操作系统。需要设计一个调度算法，要求每个应用只能连续执行一段时间，然后内核就会将它强制性切换出去，考虑的是是吞吐量、延迟、和公平性。