

清 华 大 学

综 合 论 文 训 练

题目：rCore-Tutorial 操作系统中进程
调度算法的设计与实现

系 别：计算机科学与技术系

专 业：计算机科学与技术

姓 名：马思源

指导教师：陈 渝 副教授

2022 年 6 月 5 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签 名：_____ 导师签名：_____ 日 期_____

中文摘要

本文主要介绍为 rCore-Tutorial 教学操作系统实现多种经典单核进程调度算法的过程，对各种调度算法的分析比较，以及将单核 rCore-Tutorial 扩展为多核操作系统的方法与扩展后对多核进程调度的研究与分析。

进程是系统进行资源分配和调度的基本单位，是操作系统的基础。rCore-Tutorial 作为一个经典的、支持完备的包括内存管理、进程调度、文件系统等基本功能的教学用操作系统，其进程管理模块需要被着重分析与拓展。结合操作系统课程内容，进程管理模块的一个扩展方向便是更多调度方法的实现。本文从这一方面入手，在当前 rCore-Tutorial 进程模块的基础之上实现近十种经典进程调度算法，使 rCore-Tutorial 能够更好地模拟各类操作系统的运作需要，同时对已经实现的算法进行测试、分析与比较。

另一个进程调度扩展方向是多核操作系统的调度。本文同样在这个方向做出努力，首先将单核的 rCore-Tutorial 扩展为了多核操作系统，之后对其进程调度部分着重修改与测试，展示单核和多核调度的区别之处。

关键词：操作系统；进程；调度算法；多核

ABSTRACT

This paper mainly introduces the procedure of implementing a variety of classical single-core process scheduling algorithms for rCore-Tutorial tutoring operating system, the analysis and comparison of various scheduling algorithms, as well as the method of expanding single-core rCore-Tutorial into multi-core operating system and the research and analysis of multi-core process scheduling after expansion.

Process is the basic unit of resource allocation and scheduling and the basis of the operating system. As a classic tutoring operating system with complete support, including memory management, process scheduling, file system and other basic functions, the process management module of rCore-Tutorial needs to be analyzed and expanded. Combined with the course content of operating system, an extension direction of process management module is the implementation of more scheduling algorithms. Starting from this aspect, this paper implements nearly ten classical process scheduling algorithms based on the current rCore-Tutorial process module, so that rcore tutorial can better simulate the operation needs of various operating systems, while these implemented algorithms are testing and analyzing.

Another extension of process scheduling is the scheduling of multi-core operating systems. This paper also makes efforts in this direction. The rCore-Tutorial of single core is extended to a multi-core operating system, and then the process scheduling part is modified and tested to show the differences between single-core and multi-core scheduling.

Keywords: OS; process; scheduling algorithm; multi-core

目 录

第 1 章 引言	1
1.1 课题背景	1
1.1.1 操作系统与进程	1
1.1.2 rCore-Tutorial 的进程调度	2
1.2 相关研究工作	3
1.3 课题目标、意义与内容	3
第 2 章 rCore-Tutorial 进程管理模块架构	5
2.1 整体架构	5
2.2 进程管理模块	6
2.2.1 进程控制块	6
2.2.2 进程管理器	6
2.2.3 处理器管理结构	6
2.2.4 对外开放接口	7
2.3 与进程有关的重要系统调用	7
2.3.1 fork 系统调用	7
2.3.2 exec 系统调用	8
2.3.3 exit 系统调用	8
2.3.4 waitpid 系统调用	8
2.3.5 yield 系统调用	8
2.3.6 sleep 系统调用	8
2.4 进程的生命周期	9
2.5 当前调度策略	9
第 3 章 单核进程调度算法的实现	11
3.1 批处理系统的调度	11
3.1.1 最短作业优先调度算法	11
3.1.2 最高响应比优先调度算法	13
3.1.3 最短完成时间优先调度算法	13

3.2 交互式系统的调度	14
3.2.1 多级队列调度算法	14
3.2.2 多级反馈队列调度算法	15
3.2.3 彩票调度算法	16
3.2.4 步长调度算法	17
3.3 实时计算机系统的调度	18
3.3.1 单调速率调度算法	19
3.3.2 最早截止时间优先算法	19
第 4 章 进程调度算法的测试、分析与比较	21
4.1 SJF 与 STCF 调度策略对比	21
4.2 MQ 与 MLFQ 调度策略对比	22
4.3 两种 FSS 调度策略对比	24
4.4 实时调度策略对比	25
第 5 章 rCore-Tutorial 多核扩展	28
5.1 从单核到多核	28
5.1.1 多核操作系统的启动	28
5.1.2 其他改动	30
5.2 多核系统的进程调度	30
5.2.1 多处理机	30
5.2.2 switch 的安全保证	31
5.2.3 潜在的死锁	32
5.3 单核与多核调度对比	34
第 6 章 总结	35
插图索引	36
表格索引	37
参考文献	38
致 谢	39
声 明	41
附录 A 外文资料的书面翻译	43

主要符号对照表

OS	操作系统
PCB	进程控制块
PC	程序计数器
PROCESSOR	处理器抽象实例
FIFO	先进先出
FCFS	先来先服务
RR	时间片轮转
SJF	最短作业优先
HRRN	最高响应比优先
STCF	最短完成时间优先
MQ	多级队列
MLFQ	多级反馈队列
FSS	公平共享
RMS	单调速率调度
EDF	最早截止时间优先
DDL	截止期限
QEMU	可以模拟各种硬件设备的虚拟化模拟器
SMP	对称多处理

第 1 章 引言

本文以对进程调度的关注为出发点，以 rCore-Tutorial 为课题的实验对象，尝试对其进行进程调度方面的拓展。结合操作系统课程内容，本文中选取了近十种单核调度算法，逐一实现在 rCore-Tutorial 的进程调度模块中。此外，为了研究多核下的进程调度，将 rCore-Tutorial 升级为了多处理器操作系统，再进行进程调度的分析与测试。

本章将讲述课题的产生背景，分析现有的同类工作，对本文的内容与意义进行说明。

1.1 课题背景

1.1.1 操作系统与进程

操作系统（Operating System，OS）是计算机系统中最基本的软件。OS 控制和管理整个计算机系统的硬件和软件资源，负责合理地执行与调度应用程序，为用户和其它软件提供环境支持。随着计算机的发展，用户对正在执行的应用有着更高的管理控制要求，操作系统自身也需要更明确地获知内部运行程序的执行状态，于是进程的概念应运而生，并越加重要。

进程（Process）是 OS 对一个应用程序的一次运行过程的抽象，是 OS 进行资源分配和调度的基本单位。从用户角度看，当前程序所抽象出的进程具有独立的控制流和私有的地址空间，似乎独占了整个处理器；从内核角度看，进程是一块独立的、动态的数据空间，可以对其进行资源的分配与状态的监控、调整。

在内核中，进程具有完整的生命周期（图1.1），会经历从创建到运行最后退出的完整过程。在这一过程中，OS 需要负责进行进程在各个状态之间的切换，保证各个进程都能被正常执行。因此，进程管理模块便成为了 OS 中极其重要的一部分，以保证进程的正常创建、运行、暂停、退出等操作。同时，无论是在批处理系统或是分时多任务系统中，内核同时驻留的进程数一般均会超过处理机数，这就需要 OS 能够以适当的方法来管理进程，调整不同进程的状态，选择分配处理机。这一处理机的分配过程便被称为进程调度（Process Scheduling），OS 所选择的分配策略则被称为调度算法。

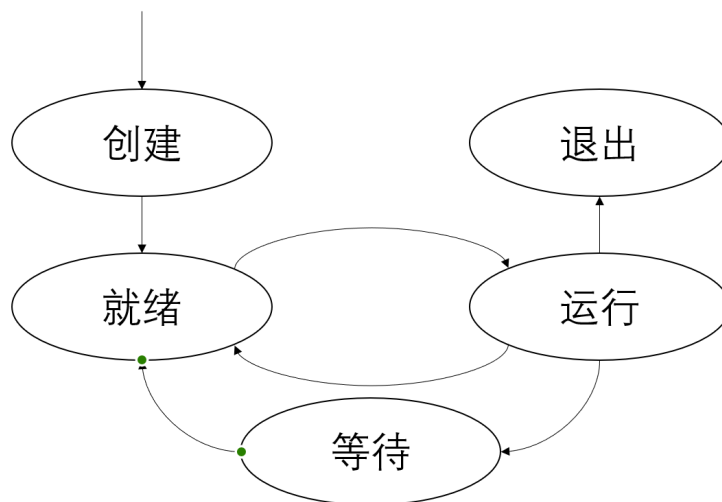


图 1.1 进程的五状态模型

1.1.2 rCore-Tutorial 的进程调度

rCore-Tutorial（第三版，全称应为 rCore-Tutorial-v3）是面向操作系统初学者使用 rust 语言编写的教学用单核操作系统。rCore-Tutorial 基于 Unix 实现，运行在 QEMU 模拟器或 K210 开发板的 RISC-V 平台上。它的实现过程分章节模拟了操作系统的发展历程，从裸机执行环境到批处理系统再到分时多任务系统，最后形成了具有内存管理、进程调度、文件系统、多线程并发等功能的完备操作系统。同时，每一章节配有详细的说明文档^[1]，对开发者友好，可以较为简便地进行该系统的缺漏补充与功能扩展。rCore-Tutorial 作为本校操作系统课程实验的教学系统，被许多同学所学习与实践，具有重要的教学意义。

总览 rCore-Tutorial 的整体架构，与操作系统课程内容相比，可以发现该教学系统仍有很大的拓展空间。考虑进程相关的内容，rCore-Tutorial 拥有着比较完备的进程管理模块，覆盖了进程状态监听、进程创建与执行、进程切换与退出等功能。但是，其在调度方面采用的是简单的时间片轮转的调度算法，而操作系统的经典调度算法除此之外仍有许多种，各种不同调度算法具有独特的执行逻辑与优缺点，可以进一步完善 rCore-Tutorial 的进程部分内容。因此，为 rCore-Tutorial 实现更多的进程调度算法，是该教学系统的一大拓展方向。

另一方面，在计算机普便是多处理器架构的当下，单核的 rCore-Tutorial 不符合主流。因此，将其扩展为多核操作系统可以让 rCore-Tutorial 与实际的联系更加紧密，让课程的教学内容更加贴近现实。同时，多处理机下的进程调度也是进程调度算法所要研究的重要一环，将 rCore-Tutorial 升级为多核之后，可以借助其来

对多核进程调度进行更加深入的分析与测试。

1.2 相关研究工作

Xv6 是 MIT 为其操作系统课程开发的操作系统。它以 Unix Version 6 (v6) 这一操作系统的架构为基础，使用 ANSI C 标准运行在 x86 多处理机平台之上，是一个经典且知名的教学用操作系统。事实上，rCore-Tutorial 的很多地方就参考了 xv6 的架构。在进程模块中，xv6 的控制流转换采用用户态——内核——调度器——内核——另一进程的用户态的方式，而非用户进程的直接切换。在调度方式上，xv6 进行多路复用，让等待 IO 的进程睡眠和让时间片用完的进程暂停，之后在队列选取另一进程执行。这一过程实际采用的调度算法仍是时间片轮转，这与 rCore-Tutorial 是一致的^[2]。

但与 rCore-Tutorial 不同的是，xv6 为多处理机系统，因此在进程调度中需要考虑同步互斥的要求。xv6 必须为进程表添加锁，调度器在进行调度的同时必须一直持有进程表的锁来保证进程不被错误抢占，同时保护进程上下文信息的正确性。

从 xv6 的实现可以看出，在进程调度算法的选择上，教学用操作系统一般均会采用一种较为简洁的调度算法，而忽略了其他经典算法的实现。因此，其他调度算法的嵌入是一个新的挑战，也是一次大的拓展。

1.3 课题目标、意义与内容

课题目标

本课题的目标主要包括两个方向：

- 拓展 rCore-Tutorial 进程调度，实现并分析多种单核调度算法。
- 将单核 rCore-Tutorial 拓展为多核操作系统，进行多核进程调度的分析。

课题意义

通过完成上述目标，可以

- 落实并拓展操作系统课程所学，加深对操作系统知识的理解。
- 为操作系统教学补全内容，提供了更完整的调度方面的参考实现。
- 提高自身的编程水平与计算机能力。

课题内容

按照论文结构，本课题的内容主要包括：

- 第二章：分析 rCore-Tutorial 的进程管理模块的实现与现有调度的进行过程。
- 第三章：介绍并实现各个经典的单核调度算法。
- 第四章：对实现的算法进行分析比较。
- 第五章：进行 rCore-Tutorial 的多核拓展，着重分析多核下的进程管理。

第 2 章 rCore-Tutorial 进程管理模块架构

本章将描述 rCore-Tutorial 教学系统的整体架构，并着重分析其进程管理部分的实现原理与方法。通过对现有架构的深入理解，来提供更深刻的进程管理逻辑认知，为下一步实现各种进程调度算法做好准备。

2.1 整体架构

rCore-Tutorial 作为教学用操作系统，其功能的完善以章节为单位逐步进行，以求清晰明了。截至当前，rCore-Tutorial 主要包括内存管理模块（Chapter 4 实现）、进程管理模块（Chapter 5 实现）、文件系统支持（Chapter 6 实现）等几个部分，此外还有进程通信（Chapter 7 实现）、多线程并发与同步互斥（Chapter 8 实现）、I/O 设备管理（Chapter 9 实现）等功能的实现。

内存管理模块采用页式内存管理模式，以 RISC-V 64 提供的 SV39 分页硬件机制为基础，在内核实现了对应的软件结构。SV39 多级页表建立了虚拟地址空间抽象，将内核与应用空间隔离，通过一个栈式分配器实现物理页帧管理。这一模块使每个应用拥有独立的虚拟地址空间，且使其与内核空间分离，提高了内存利用率，增强了系统安全。

文件系统模块实现了文件的打开、关闭与读写等功能，设置完整的文件系统抽象接口，与同时设计的 easy-fs 文件系统对接，为进程添加文件描述符表使用户应用能够进行文件的访问。

进程通信方面，rCore-Tutorial 实现了管道机制，来支持父子间的单向通信。rCore-Tutorial 也支持用户态和内核态的线程管理，并实现锁与信号量的机制以维护多线程的并发与线程间的同步互斥。之后，内核实现了对外设中断的处理，添加 I/O 设备驱动程序，扩展系统的 I/O 能力，让应用更便捷地访问外设。

由于 Chapter 8 多线程的加入使得进程不再成为处理机调度的直接对象，但是对于进程调度策略而言，其具体属性并不影响策略的执行。因此下面的考虑还是以 Chapter 5 实现的进程管理模块为基础，默认进程均为单线程，来让进程成为控制和调度的基本单位。

2.2 进程管理模块

rCore-Tutorial 的进程管理模块最初只是管理简单的任务（task），随着进程（process）这一概念的出现，在 Chapter 5 形成了完善的进程管理结构。以下不区分任务与进程，所提到的“任务（task）”均可等同于“进程（process）”。

2.2.1 进程控制块

进程控制块（Process Control Block, PCB）是进程管理的核心结构体，其中记录了进程从创建到退出所涉及到的各种数据信息。在内核看来，进程控制块便是一个进程的抽象。在 PCB 中存储了以下重要数据：

- 进程标识符 PID，它是进程的唯一标识，由栈式分配器 Pid Allocator 按序分配。
- 内核栈，进程在内核中所属的地址空间。

以上二者在进程创建时即会分配，不会发生改动，而下面的数据则可能在运行时随进程被修改，因此会采用一个互斥访问的包装包裹起来。

- 进程上下文 Task Context，其中存储了部分寄存器、内核栈位置和 pc 位置，用于进程切换。
- 进程状态 Task Status，记录当前进程的生存状态，包括就绪（Ready），运行（Running）等。
- 应用地址空间、父子进程关系等其他重要进程信息。

2.2.2 进程管理器

进程管理器（Manager）维护着一个队列，其中保存了当前处于就绪状态的所有进程。它对外开放 add 和 fetch 两个接口，用于将暂停的进程加回队列和从队列取出进程交由处理机执行。因此，通过修改 add 和 fetch 的逻辑，也就可以实现不同的进程调度方法，而这里也就是下面调度算法工作的主要实现之处。

当前队列的存取方式是简单的先进先出式（FIFO）。

2.2.3 处理器管理结构

处理器管理结构（Processor）是 CPU 的抽象，负责维护用户应用程序在 CPU 中的连续运行，记录 CPU 当前运行状态，并执行不同应用程序控制流的切换。

在全局初始化的 PROCESSOR 对象中，有成员 current，用来记录当前运行进程；成员 idle_task_cx，表示内核运行的 idle 控制流。当内核启动后，会进入 Processor 的 run_tasks() 方法中。该方法是一个 loop，将循环地向 Manager 获取可

执行的进程，修改其状态，并调用 `switch` 方法从 `idle` 切换到对应进程的控制流，此时离开 `run_tasks()`。当进程主动或被动放弃 CPU，`switch` 会再次被调用，此时由进程控制流返回了 `idle` 控制流，回到 `run_tasks()` 继续下一次调度。

同时 `Processor` 支持获得并取出当前进程的功能。

```
pub fn run_tasks() {
    loop {
        let mut processor = PROCESSOR.exclusive_access();
        if let Some(task) = fetch_task() {
            ..
            processor.current = Some(task);
            // release processor manually
            drop(processor);
            unsafe {
                __switch(idle_task_cx_ptr, next_task_cx_ptr);
            }
        } else {
            println!("no tasks available in run_tasks");
        }
    }
}
```

2.2.4 对外开放接口

进程模块还拥有一些对外开放的接口，让内核的其他部分发起进程调度。

- `suspend_current_and_run_next()` 暂停当前进程，加回就绪队列，切换控制流从当前进程到 `idle`。
- `block_current_and_run_next()` 置当前进程为阻塞态，不加回进程，只切换控制流，等到其他因素解决阻塞再处理当前进程。
- `exit_current_and_run_next()` 进程退出使用，回收资源，处理子进程，直接切回 `idle` 控制流。

2.3 与进程有关的重要系统调用

2.3.1 fork 系统调用

`fork` 系统调用是进程创建的基础。在初始状态，操作系统仅把设定好的一个初始进程 `INITPROC` 加入调度队列之中来执行，若要创建其他的进程则离不开 `fork` 的使用。当一个进程调用 `fork` 后，内核会创建一个和该进程几乎完全相同的新进程，包括相同的各个数据段与寄存器，而地址空间则是完全地拷贝过来。它

们的区别在于，原有进程和新进程有父子关系，同时 `fork` 的返回也不一样，对于新进程返回 0，而父进程返回新进程的 PID。在用户程序中，根据获取到的 `fork` 调用的返回值，便可以区分现在是位于新进程还是老进程之中。

2.3.2 exec 系统调用

`exec` 系统调用则是让当前进程能够执行指定的可执行文件，它加载指定文件的代码和数据，替换掉当前进程的地址空间，将一些记录清零，从而以一个全新进程的样子回到调度之中。一个一般的做法是，当前进程调用 `fork`，之后判断返回值，若为 0，说明进入了新进程中，则调用 `exec` 执行指定程序。这样以来，使用 `fork+exec` 的组合，就完整实现了指定进程的创建过程。

2.3.3 exit 系统调用

`exit` 系统调用用于进程的正常退出。每个进程执行正常完毕后，都会调用 `exit` 来进行部分资源的回收与退出码的写入，同时还会将当前进程的子进程挂靠在 `INITPROC` 下。

2.3.4 waitpid 系统调用

`waitpid` 系统调用则是连接父子进程的桥梁，父进程可以使用该系统调用获取子进程的返回状态，如上述 `exit` 的退出码，并回收子进程 `exit` 未回收的内核栈等资源。若父进程未调用 `waitpid` 先于子进程退出，则上述 `exit` 的挂靠操作则会让 `INITPROC` 负责进行 `waitpid` 的回收过程。

2.3.5 yield 系统调用

`yield` 系统调用则比较简单，当进程调用 `yield` 后，将会被暂停并重新加回就绪队列，使执行流回到 `Processor` 通过新的调度选择进程执行。需要注意的是，即使进程调用了 `yield`，如果满足优先调度条件仍会连续执行。

2.3.6 sleep 系统调用

`sleep` 系统调用的功能显而易见，就是让进程进入睡眠一段时间，到时被唤醒执行。最初 `sleep` 的实现是由用户监测时间，向操作系统发送 `yield` 请求来实现；现在的 `sleep` 则会在内核中设置定时器，将当前进程置为 `blocking` 状态，每隔固定时长检查是否有到期的定时器，修改进程状态为 `Ready`，加入就绪队列，进入调度程序。

2.4 进程的生命周期

如上所述, 进程会经历创建、运行、等待、退出等等生命历程。在 rCore-Tutorial 中, 进程调度模块对各个历程的实现如下:

- 创建: 操作系统为用户提供了 `fork` 和 `exec` 系统调用, 指定新进程的创建便是由当前进程先进行 `fork`, 再在子进程中使用 `exec` 实现的。同时, 最初需要有一个初始进程 `INITPROC`, 这个进程名是操作系统指定的, 用户需要在对应的用户文件中指明初始要做的事情从而启动整个进程执行过程。
- 运行: 操作系统启动后, 便会进入 `Processor` 的 `run_tasks()` 之中, 开始循环地向 `Manager` 获取需执行的进程。成功获取后, 进程状态被修改为 `Running`, 执行流切换至进程的位置, 开始该进程的执行。
- 停止与等待: 当进程由于主动或被动原因放弃 `CPU` 后, 它需要被暂停, 进入就绪队列等待。在对应的处理方法中, 该进程的状态被改为 `Ready`, 被 `Manager` 加回就绪队列, 并切换执行流至 `idle`, 回到 `Processor` 之内进行下一轮的获取进程。
- 退出: 当进程执行结束后需要调用 `exit` 退出。在 `exit` 中, 除了写入退出码等基本操作, 还需要对其子进程进行向 `INITPROC` 的挂靠, 以避免子进程无人回收造成内存泄漏, 同时回收掉部分资源, 剩下部分资源留待父进程调用 `waitpid` 回收。

总的来说, 进程由 `fork (+exec)` 创建, 被 `Manager` 选取进入 `Processor` 切换执行, 中间可能会被暂停加回就绪队列, 待执行完毕调用 `exit` 回收资源, 最终完成整个执行任务。

2.5 当前调度策略

rCore-Tutorial 当前的进程调度策略采用的是时间片轮转 (`Round-Robin`, `RR`) 调度算法。

如前所述, `Manager` 中采用的队列方式为简单的先进先出 (`FIFO`), 若无其他操作, 内核的调度策略也应是先进先出调度, 或者称为先来先服务 (`first come first service`, `FCFS`) 调度。但 rCore-Tutorial 打开了时钟中断, 设置了 `10ms` 的定时器, 每过去一个时间片则触发时钟中断。trap 对此的处理如下:

```
Trap::Interrupt(Interrupt::SupervisorTimer) => {  
    set_next_trigger();  
    check_timer();  
    suspend_current_and_run_next();  
}
```

- `set_next_trigger()` 设置了下一个时间片的定时器;
- `check_timer()` 检查上述的 `sleep` 系统调用是否有需被唤醒的进程;
- `suspend_current_and_run_next()` 则启动了调度。

这样，10ms 的时间片调度和 `yield`、`exit` 等其他启动调度的方法共同组成了 `rCore-Tutorial` 的时间片轮转调度算法。

第 3 章 单核进程调度算法的实现

本章将在前一章介绍的基础架构之上完成各种进程调度算法的实现。

随着计算机系统的发展，调度算法也不断推陈出新。按照不同操作系统的特点，可以将经典的单核调度算法划分为几个大类，包括批处理系统的调度、交互式系统的调度与实时计算机系统的调度。下面将依次分析各类算法的原理，并在 rCore-Tutorial 中实现之。

3.1 批处理系统的调度

批处理系统的任务以科学计算为主，IO 操作较少，主要耗时为处理器计算，且这一部分时间基本可以预测到。因此在批处理系统阶段的调度一般比较简单，先来先服务（first-come first-severd, FCFS）调度算法和最短作业优先（Shortest Job First, SJF）调度算法便是其中较为基础的两种。此外，对于最短作业优先算法的判定条件进行扩展，则产生了最高相应比优先（Highest Response Ratio Next, HRRN）调度算法。同时为了使用户可以随时申请进程的执行，支持可抢占的进程调度方式，在 SJF 算法的基础上提出了最短完成时间优先（Shortest Time-to-Completion First, STCF）调度算法。

3.1.1 最短作业优先调度算法

最短作业优先（SJF）调度算法根据进程的执行时间来分配 CPU 资源，在程序运行前，操作系统需要知道所有进程的预期执行时间，每次需要调度时，操作系统会选取预期执行时间最短的进程执行^[3]。

SJF 算法的关键在于如何知道下次 CPU 执行的长度。一种方法是让用户在创建进程作业时给出进程执行预期，估计一个较为精确的数值供操作系统直接使用。另一种方法则是利用过往数据估算一个进程的下一次 CPU 执行时间，一般采用式 (3.1)。

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \quad (3.1)$$

其中 τ_n 表示第 n 次的 CPU 预估时间，而 t_n 则代表第 n 次的 CPU 执行时间。对于会多次运行的进程，可以通过这一递推式得到下一次运行的预期时间。

实现

采用第一种方法，让用户程序提供进程的预期执行时间。在 PCB 中添加 `prediction` 项用于记录进程的预期，由于 `rCore-Tutorial` 的进程创建机制是 `fork+exec`，`prediction` 则是在 `exec` 执行可执行文件时同步传入。此后，将 `Manager` 中的就绪队列按 `prediction` 排序，保证每次取出的是最短的进程。

因此需要做的主要有：

- 修改用户库，为 `exec` 系统调用添加新的参数；
- 添加新的 PCB 信息 `prediction`，设其初值为较大的值，并在 `exec` 时修改；
- 修改 `Manager` 的 `add` 逻辑，实现进程按预期时间排序。

```
for queue in 0..self.ready_queue.len(){
    let task1 = self.ready_queue.get_mut(queue)
        .unwrap();
    let prediction1 = task1.inner_exclusive_access()
        .task_prediction;
    let running1 = task1.inner_exclusive_access()
        .task_isrunning;
    if running && !running1{
        self.ready_queue.insert(queue, task);
        return
    }
    else if !running && running1{
        continue;
    }
    else if prediction < prediction1 {
        self.ready_queue.insert(queue, task);
        return
    }
}
```

修改后的主要 `add_task` 逻辑如上。另一点需要注意的是，尽管 SJF 是非抢占的，但由于上述实现基于 `fork+exec` 机制，在一个新进程被创建时，要首先调度到这个新进程，执行 `exec` 添加了 `prediction` 的值后，再停止该进程，重新运行 `run_next` 来选择最短进程，而不是仍然运行新进程创建之前的进程，实际上这更像之后要提到的抢占式。因此为了保证新进程到来后不抢占原进程，添加标志 `isrunning`，对于正常运行中的进程此值为 `T`，而对刚执行完毕 `exec`，返回到就绪队列的新进程不置为 `T` 仍为 `F`，在添加任务时优先保证为 `T` 的进程在前面，再比较 `prediction`。这样，即使出现了一个新的更短的进程，它也会排在原来的进程后面，从而使原进程继续执行。

3.1.2 最高响应比优先调度算法

由于 SJF 的判断标准比较单一，在其基础上改进它的判断标准得到最高响应比优先（HRRN）调度算法，即选择就绪队列中响应比 R 值最高的一个进程 (3.2)。

$$R = \frac{W + S}{S} \quad (3.2)$$

其中 W 表示等待时间， S 表示执行时间。同样地，HRRN 为非抢占式的算法。与 SJF 相比，HRRN 将等待时间与执行时间一同纳入考虑之中，有助于解决长作业饥饿等问题。

实现

除了 SJF 添加的 `prediction` 信息已经记录了 S 的值，还需要在 PCB 中添加对等待时间 `waiting_time` 的记录。同时，由于主动放弃处理机的情况存在，还需要信息 `last_yield_time` 来记录进程本次暂停的时刻，该变量初始化为进程的到达时间。因此，`waiting_time` 由以下计算得到：

```
task_inner.task_waiting_time += get_time_ms()
                             - task_inner.task_last_yield_time;
```

其中 `get_time_ms` 为获取当前时刻的方法。获取了 `waiting_time` 后，将 SJF 的按预期排序修改为按照响应比进行排序，即可实现 HRRN 调度。由于内核中不使用浮点运算，这里不直接进行 R 值的计算比较，而是对两个进程的 WS 互乘进行比较。(3.3)的结果就说明进程 1 会先于进程 2 被调度。

$$W_1 S_2 > W_2 S_1 \quad (3.3)$$

3.1.3 最短完成时间优先调度算法

上面提到的 SJF 和 HRRN 算法不允许抢占，晚来一步的短进程可能不得不承受较长的周转时间，那么可以实现一种支持进程抢占的改进型 SJF 调度策略，即最短完成时间优先（STCF，也称 SRTF，最短剩余时间优先）调度算法^[3]。

如果在当前进程的执行过程中，一个预期时间小于当前进程执行的剩余时间的进程被加入就绪队列，那么应该让后者抢占 CPU 来优先执行，而不能像非抢占式的 SJF 算法一样允许当前运行进程先完成 CPU 执行，这就是 STCF 算法的原理。

实现

回顾 SJF 算法的实现，在不增加 `isrunning` 标志时，实际上就是一个抢占式的调度。那么在这一基础上，将每次调度的判断标志改为当前进程的剩余执行时间，来实现 STCF 调度。

为了计算剩余时间，类比 HRRN 来添加额外的记录信息。

- `task_last_start_time`: 记录上次进程被调度开始执行的时间；
- `task_complete_time`: 进程的预期剩余执行时间。

`task_complete_time` 的初始值为之前提到的 `prediction`。每当进程被调度开始执行，则将 `task_last_start_time` 记为当前时间；每当新进程到达，需要中断当前进程重新调度，则

```
task_inner.task_complete_time -= (get_time_ms() -  
    task_inner.task_last_start_time) as isize;
```

修改当前进程的剩余时间，在加回就绪队列时按照新的剩余时间排序，接受调度。

3.2 交互式系统的调度

随着互联网的发展，各种 I/O 设备成为计算机系统的基本配置，用户使用交互式应用的频率越来越高，驻留在内存中的应用也越来越多。这种情况下，无法再去预知执行时间，也就很难再去使用上面的调度方法。新的调度策略需要能够根据进程的动态运行状态进行调整，以应对各种复杂的情况。多级反馈队列（Multi-level Feedback Queue, MLFQ）便是一种比较灵活地支持动态调度的调度算法。

另一方面，由于大规模服务器的出现，为众多用户提供进程运行的支持，对不同用户或进程的公平性就成为了不得不考虑的一个问题。如果把公平性放在第一位，那么就产生一类新的调度策略——公平共享调度（Fair Share Scheduling, FSS），它按照每个进程的重要性来按比例分配处理器执行时间。其中，彩票（Lottery）调度和步长（Stride）调度是两类经典的 FSS 调度策略。

3.2.1 多级队列调度算法

多级队列调度（Multi-level Queue）算法是多级反馈队列调度的基础。该算法将就绪队列分成多个单独队列，根据进程的属性为其分配一个固定的优先级，再

按照优先级放入对应的就绪队列中。每个队列内部是单独调度的，比如沿用基础的 RR 调度，而队列间则一般按照队列的优先级进行抢占式调度，比如为 IO 密集型任务设置更高的优先级，一旦被唤醒就优先执行，而让 CPU 密集型进程处在更低的优先级^[3]。

实现

此处引入了新的进程属性——优先级（priority）。因此，为用户库与内核添加新的系统调用 `sys_set_priority` 来让用户程序有方法设置进程的优先级，同时将其记录在 PCB 之中。规定优先级的取值范围，并将就绪队列按照优先级的取值数划分为许多个，每个进程根据优先级的数值进入不同的队列中。同时，整体上保持 rCore-Tutorial 的 RR 调度不变，而在取任务时按优先级从高到低遍历各个队列，直至顺利取出一个可执行的任务。

3.2.2 多级反馈队列调度算法

事实上，进程在执行过程中具有很强的动态性，可能现在是 IO 密集型，而执行一段时间后就变成了 CPU 密集型。同时，同为 IO 密集型的进程，其密集程度也可能有差异，仅靠固定优先级的多级队列调度难以满足实际的调度需要。

因此，支持动态优先级的多级反馈队列（MLFQ）调度算法^[4]被提出。与 MQ 一样，MLFQ 拥有多个就绪队列，但不同的是它将在执行过程中动态地判断进程应属的优先级，从而让进程在不同队列中迁移。相比 MQ 调度，多级反馈队列调度主要增加的基本原则有：

- 创建进程并让进程首次进入就绪队列时，设置进程的优先级为最高优先级，进入最高队列。
- 进程用完其时间配额后，就会降低其优先级。
- 经过一段时间后，将所有就绪进程设回最高优先级，加回最高队列。

这样，那些持续占用 CPU 计算资源的进程将会被快速地降低优先级；而那些每次运行时间较短，会主动放弃 CPU 的 IO 进程等，则能够维持一个较慢的优先级降低速度，得到快速的响应处理。

实现

在 MQ 调度的基础上，同样划分多个就绪队列，不同的是 `add` 的逻辑被修改，所有进程初始均进入第一队列之中，设为最高优先级。每次进程得到执行，但由于时间片到期而被暂停时，下降其优先级，并把它放到下一级就绪队列中，直到降至最低优先级，则会保持在最末队列。`fetch` 的逻辑则变为每次从第一队列去取

出任务，直到第一队列为空才去下一队列寻找。

```
pub fn add(&mut self, task: Arc<TaskControlBlock>) {
    let task_inner = task.inner_exclusive_access();
    let priority = task_inner.task_priority;
    drop(task_inner);
    if priority == 0{
        self.ready_queue.get_mut(0).unwrap().push_back(task);
    }
    else{
        self.ready_queue.get_mut(priority - 1).unwrap()
            .push_back(task);
    }
}
pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
    for queue in (0..MLFQ_PRIORITY).rev() {
        let mlfq = self.ready_queue.get_mut(queue).unwrap();
        if !mlfq.is_empty() {
            return mlfq.pop_front();
        }
    }
    self.ready_queue.get_mut(MLFQ_PRIORITY - 1).unwrap()
        .pop_front()
}
```

此外，为了实现提升优先级的操作，仿照 `sleep` 来添加定时器，设定一个周期，每次到期后则将 `Manager` 中的进程均调整到第一队列中，并启动下个周期的计时。这样便可以根据进程当下的属性重新划分优先级，让长期处在低优先级、无法运行的进程能重新获得执行的机会。

3.2.3 彩票调度算法

彩票（Lottery）调度^[5]是一种很有意思的 FSS 调度，它模拟经济学的彩票的购买和中奖的随机性。该算法要求给每个进程发彩票，进程优先级越高，所得到的彩票就越多；然后每次由于时间片用光等原因需要调度时，进行一次随机抽奖，被抽中的彩票属于哪个进程，哪个进程就能运行。

由于总彩票数（进程数）已知，操作系统产生一个从 0 到总数的随机数作为获奖彩票号，拥有这个彩票号的进程中奖，并获得下一次处理器执行机会。通过在一段较长的时间内不断地抽取彩票，基于统计学原理，可以保证两个进程获得与优先级大致等比例的处理器的执行时间。

实现

首先需要 MQ 调度所实现的 `sys_set_priority` 优先级设定系统调用，设定每一级优先级对应 `TICKET_X`（如 100）张彩票。在 `Manager` 中，除了 `ready_queue` 就

绪队列，添加两个新的成员：

- `total_counts`：记录需要生成的彩票总数。
- `lottery_array`：记录每个进程所持有的彩票数，其序号与 `ready_queue` 对应。

在每次调用 `add` 时，和添加到就绪队列同步进行 `total_counts` 的增加和 `lottery_array` 的添加。重点在于 `fetch` 的逻辑。

```
pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
    let mut rng = ChaCha20Rng::seed_from_u64(
        get_time() as u64);
    let lucky_dog = rng.gen_range(0..self.total_counts);
    let mut tmp_sum = 0;
    for (ind, &val) in self.lottery_array.iter().enumerate() {
        tmp_sum += val * TICKET_X;
        if tmp_sum > lucky_dog {
            self.lottery_array.remove(ind);
            self.total_counts -= val * TICKET_X;
            return self.ready_queue.remove(ind);
        }
    }
    self.ready_queue.pop_front()
}
```

第一个需要解决的问题是如何进行彩票的随机抽奖。由于 `rCore-Tutorial` 建立在 `no_std` 的基础之上，`rust rand` 库的许多功能被禁用。因此这里使用了 `rand_chacha` 这一支持 `no_std` 的随机数生成库。

采用当前时刻作为随机数生成的种子，使用 `chacha20` 轮加密方式生成一个在 0 到总数之间的随机数。之后从 `lottery_array` 起始开始遍历，如果随机数落到了某个进程下，则取出该进程，同时进行 `total` 的减小和 `lottery_array` 的删除。

3.2.4 步长调度算法

步长（Stride）调度^[6]是另一种 FSS 调度策略。其基本思路是：每个进程有一个步长（Stride）属性值，这个值与进程优先级成反比；同时每个进程还会被记录已经走过的总步长，即行程（pass），内核每次选择拥有最小行程值的进程运行。

实现

首先设置一个大常数 `BIG_STRIDE`，用来计算每个进程的 `stride` 值：

$$stride = \frac{BIG_STRIDE}{priority} \quad (3.4)$$

得到与优先级成反比的 `stride`，作为进程的固有属性。此后，每次进程被选

出执行一个时间片后，则将其 *pass* 增加一个 *stride* 的大小。加回就绪队列时，类似 STCF 按剩余时间排序的方法，形成一个按 *pass* 排序的队列，每次取出队首的 *pass* 最小的进程执行。

Stride 存在一个溢出问题。考虑下面的数值均以 u8（8bit 无符号整型）存储，若进程 1 的 *pass* 为 250，进程 2 的 *pass* 为 254，二者的 *stride* 均为 10，按照 Stride 调度，下一步应该是进程 1 被调度，*pass* 变为 260，然后是进程 2 被调度。但由于 u8 的范围不超过 255，进程 1 执行后，其 *pass* 由于溢出变为了 4，那么下一次调度实际上还是选择了进程 1，进程 2 无法得到调度。

为了解决这一问题，规定优先级大于等于 2，这样每次增长的 *stride* 不会超过 $\frac{BIG_STRIDE}{2}$ 。若某次比较发现两个进程的 *pass* 差值超过了这一值，则说明发生了溢出。因此，进程 1 先于进程 2 被调度的条件是：

$$(pass_1 < pass_2) \wedge ((pass_2 - pass_1) \leq \frac{BIG_STRIDE}{2}) \quad (3.5)$$

或者

$$(pass_1 > pass_2) \wedge ((pass_1 - pass_2) > \frac{BIG_STRIDE}{2}) \quad (3.6)$$

这样才能得到正确的 *pass* 关系。

3.3 实时计算机系统的调度

实时计算机系统要求计算机能够在给定时间内对外部要求作出反应。通常实时计算机系统可以分为硬实时和软实时两类，硬实时是指任务完成时间必须在绝对的截止时间内，超过截止时间会产生灾难性的后果。软实时是指任务完成时间尽量在绝对的截止时间内，具有一定的弹性，可以偶尔接受超时的执行。实时计算机系统需要执行的任务是一组进程，其中每个进程的行为对于用户应该是可以预测的，它们称为实时进程。

进一步地，实时任务可以分为周期性和非周期性的。对于周期性任务，单调速率调度（Rate Monotonic Scheduling, RMS）算法^[7]使用静态抢占策略策略来调度之。而最早截止时间优先（Earliest Deadline First, EDF）调度算法^[7]则可以调度一般的实时任务以满足实时操作系统的需要。

3.3.1 单调速率调度算法

单调速率（RMS）调度算法的基本属性是可抢占和静态优先级。操作系统获取每个周期性任务的执行周期，并根据周期长短分配优先级，使周期短的进程可以以高优先级率先被调度。同时，抢占机制也允许新一轮开始的短周期进程来抢占未执行完毕的长周期进程。

在静态优先级调度算法中，单调速率调度是最优的一种，其可以满足绝大多数周期实时任务的调度需要。然而，在总进程数为 N 的前提下，虽然 CPU 的理论利用率极限为 1，但是在超过以下 CPU 利用率上界(3.7)时，RMS 无法保证这组进程一定能被其调度^[7]。

$$N(2^{1/N} - 1) \quad (3.7)$$

实现

修改 `exec` 系统调用，添加传入参数为周期值，不再设定明确的优先级而直接用周期来进行比较。在 `Manager` 中形成一个按周期排序的队列，周期越短排序越靠前，每次取出队首的周期最小的进程执行。

需要说明的是，这里用户程序周期任务的实现是采用 `sleep` 系统调用，睡眠时长设置为周期长度减去本轮执行使劲按。因此在等待下个周期开始执行时会处于阻塞态而不会被放置在就绪队列中，不会影响其他周期已到任务的执行。事实上，也可以略作修改地使用下面提到的 `cycle` 系统调用。

3.3.2 最早截止时间优先算法

最早截止时间优先（EDF）调度根据截止期限（deadline，DDL）动态分配优先级，对截止期限最早的进程分配最高的优先级。

EDF 的调度有些类似于 SJF 调度，但不同的是，SJF 的预期时间是一个不会改变的、与用户的传入值相同的定值，而 EDF 截止期限的确定依赖于进程的到达时间和用户的预期。同时，进程本身在执行时的行为也会动态修改 DDL，比如周期性任务，其 DDL 便随着每个周期的执行而动态增加。

EDF 算法对于实时任务而言是理论上最优的，CPU 利用率可以达到 100%。但是由于进程切换等固有的耗时操作，实际上的满利用率是不可能的

实现

和 RMS 有相似之处，不过判断标准由传入的周期改为传入的 DDL。对于任务而言，其传入的 DDL 是一个预先计算得到的固定值，也即 SJF 算法所提到的

预期时间，所以在内核中还需要与当前时间加和计算出实际的 DDL。

若任务是周期性的，就需要动态修改 DDL 的具体时间。添加一个新的系统调用 `cycle`，它类似于 `sleep`，用户需要在周期性任务的周期末尾调用，传入本进程的周期值，内核会做如下处理：

```
pub fn sys_cycle(period: usize) -> isize {
    let task = current_task().unwrap();
    let mut task_inner = task.inner_exclusive_access();
    let expire_ms = task_inner.task_deadline;
    task_inner.task_deadline += period;
    drop(task_inner);
    add_timer(expire_ms, task);
    block_current_and_run_next();
    0
}
```

将本进程的 DDL 增加了一个周期长度，之后类比 `sleep` 的做法添加了一个之前 DDL（即本周期末尾）的定时器，将进程阻塞。到了下个周期开始，定时器触发，任务被加回调度之中。

第 4 章 进程调度算法的测试、分析与比较

第三章实现了多种单核进程调度算法，本章将对其中的部分进行测试、分析与比较，以证明实现的正确性与完备性，同时验证各个算法的优缺点。测试使用 QEMU 模拟器模拟 RISC-V 64 平台架构，使得以 RISC-V 为基础编写的 rCore-Tutorial 顺利运行，正常执行各项功能。

4.1 SJF 与 STCF 调度策略对比

设计如下的测例：

```
static TESTS: &[&str] = &[
    "sjf1\0",
    "sjf2\0",
    "sjf3\0",
    "sjf4\0",
    "sjf5\0",
];

static TIMES: [usize; 5] = [
    10000,
    100000,
    1000,
    500,
    500,
];

..
    if i == 3 || i == 4{
        sleep(300);
    }
```

预设五个进程，其预期时间如 TIMES，但任务 1,2,3 是在开始时一起到达，而任务 4 则在 300ms 后到达，5 再晚 300ms。

按照 SJF，在前 500ms，由于任务 3 预期时间最短，优先执行任务 3，执行到 300ms 时，任务 4 到达，虽然时间更短，但不会发生抢占，5 到达时同理。直到 1000ms 时任务 3 执行完成，再按照 4512 的顺序执行。

而对于可抢占的 STCF 调度，前 300ms 是任务 3 执行，但 4 到达后，由于 $1000-300 > 500$ ，发生抢占，4 开始执行。再过 300ms，任务 5 到达，但此时 4 的剩余时间明显小于 5，4 继续执行。直到再过 200ms 4 执行完毕，5 开始执行，之后

是 3 的剩余部分和 12。

```
*****/
sjf1 Arrive at 87
sjf2 Arrive at 88
sjf3 Arrive at 88
I am sjf3
current time_msec = 162
sjf3 running...
sjf4 Arrive at 396
sjf3 running...
sjf3 running...
sjf5 Arrive at 703
sjf3 running...
808111
time_msec = 1167, delta = 1005ms, sjf3 OK!
I am sjf4
current time_msec = 1225
sjf4 running...
sjf4 running...
sjf4 running...
sjf4 running...
798059
time_msec = 1730, delta = 505ms, sjf4 OK!
I am sjf5
current time_msec = 1730
sjf5 running...
sjf5 running...
sjf5 running...
sjf5 running...
798059
time_msec = 2232, delta = 502ms, sjf5 OK!
I am sjf1
I am sjf2
QEMU: Terminated
```

(a) SJF 调度结果

```
*****/
sjf1 Arrive at 95
sjf2 Arrive at 96
sjf3 Arrive at 96
I am sjf3
current time_msec = 168
sjf3 running...
sjf4 Arrive at 400
I am sjf4
current time_msec = 425
sjf4 running...
sjf4 running...
sjf4 running...
sjf5 Arrive at 708
sjf4 running...
798059
time_msec = 965, delta = 540ms, sjf4 OK!
I am sjf5
current time_msec = 965
sjf5 running...
sjf5 running...
sjf5 running...
sjf5 running...
798059
time_msec = 1476, delta = 511ms, sjf5 OK!
sjf3 running...
sjf3 running...
sjf3 running...
808111
time_msec = 2266, delta = 2098ms, sjf3 OK!
I am sjf1
I am sjf2
QEMU: Terminated
```

(b) STCF 调度结果

图 4.1 SJF 与 STCF 调度对比

4.2 MQ 与 MLFQ 调度策略对比

下面设计比较方案来比较 MLFQ 相比于 MQ 的动态性优势。

测例包括三个任务，利用 `sleep` 系统调用采取睡眠唤醒方式模拟 IO 交互。任务一为单纯的 CPU 计算型；任务二在前半部分为 CPU 计算，之后变为 I/O 交互型；任务三则为交互型。

按照两个队列的 MQ 算法，任务一会被置于后台低队列，任务三则在前台高队列。对于任务二，这里假设算法按照其开始时的运行特点将其分在后台低队列。任务开始时，任务三将会一直得到优先的执行，而在任务三睡眠后任务一二会轮番执行，这符合交互式系统的调度要求。当任务二转为 I/O 交互时，应该让任务二也能得到较快的响应，但是由于任务二仍和任务一一起处于低队列，事实上的表现如图 4.2(b) 所示，任务三被唤醒时会得到优先执行，但任务二被唤醒后，既无

法抢占任务三，甚至在任务三睡眠后也可能无法优于任务一得到响应。

<pre>task 3 wake up!! task 3 is running.... task 3 is running.... task 2 is running.... task 1 is running.... task 2 is running.... task 3 wake up!! task 3 is running.... task 3 is running.... task 1 is running.... task 2 is running....</pre>	<pre>task 3 wake up!! task 3 is running.... task 2 wake up!! task 3 is running.... task 1 is running.... task 2 is running.... task 1 is running.... task 1 is running.... task 2 wake up!! task 3 wake up!! task 3 is running.... task 3 is running.... task 2 is running....</pre>
--	--

(a) 任务二为 CPU 计算型时 (b) 任务二为 I/O 型时

图 4.2 MQ 调度结果

而对于 MLFQ 算法，在任务二是 CPU 计算型时，和 MQ 基本一致，优先级下降比较慢的任务三会在每次唤醒时优先执行。当任务二状态转换后，则和 MQ 不再一致。I/O 型的任务二将不再和任务一一样快速下降优先级，这时优先级的一般顺序基本是 $3>2>1$ ，因此任务三仍会优先执行，但任务二则能先于一被响应。当进程优先级重置后，同为 I/O 型的任务二三回到了同一起跑线，因此或许还会出现任务二能够抢占任务三得到响应的现象，如图4.3(c)。

<pre>task 3 wake up!! task 3 is running.... task 3 is running.... task 2 is running.... task 1 is running.... task 2 is running.... task 3 wake up!! task 3 is running.... task 3 is running.... task 1 is running.... task 2 is running....</pre>	<pre>task 3 wake up!! task 3 is running.... task 2 wake up!! task 3 is running.... task 2 is running.... task 1 is running.... task 1 is running.... task 2 wake up!! task 2 is running.... task 1 is running....</pre>	<pre>task 3 is running.... task 2 wake up!! task 2 is running.... task 3 is running.... task 1 is running.... task 2 wake up!! task 2 is running.... task 1 is running....</pre>
--	---	--

(a) 任务二为 CPU 计算型时 (b) 任务二为 I/O 型时 (c) 优先级重置后时

图 4.3 MLFQ 调度结果

对比两种结果，可以发现，能够动态降低或升高优先级的 MLFQ 调度算法相对于 MQ 算法，在灵活性与即时处理之上有着更大的优势，能够更好地满足交互式系统的调度要求。

4.3 两种 FSS 调度策略对比

对于 FSS 调度的测试，设计内容如下的测例：

```
1 pub fn count_during(prio: isize) -> isize {
2     let start_time = get_time();
3     let mut acc = 0;
4     set_priority(prio);
5     loop {
6         spin_delay();
7         acc += 1;
8         if acc % 400 == 0 {
9             let time = get_time() - start_time;
10            if time > MAX_TIME {
11                return acc;
12            }
13        }
14    }
15 }
```

6 行的 `spin_delay` 是一个短暂延时。这样的测例共有六个，区别只在 4 行的优先级设定不同。因此，按照 FSS 调度准则，同时启动的各个测例进程得到的调度次数应该与优先级呈正比，也即最后的 `acc` 的值。观察 `acc`（图4.4的 `exitcode`）和 `prio` 的比例关系（图4.4的 `ratio`），即可检验 FSS 调度的实现效果。

```
priority = 10, exitcode = 181740800, ratio = 18174080
priority = 6, exitcode = 107915200, ratio = 17985866
priority = 7, exitcode = 134362000, ratio = 19194571
priority = 5, exitcode = 95888400, ratio = 19177680
priority = 9, exitcode = 201455200, ratio = 22383911
priority = 8, exitcode = 169357600, ratio = 21169700
```

(a) Lottery 调度结果

```
priority = 5, exitcode = 31210400, ratio = 6242080
priority = 6, exitcode = 39798000, ratio = 6633000
priority = 7, exitcode = 46374400, ratio = 6624914
priority = 8, exitcode = 51412000, ratio = 6426500
priority = 9, exitcode = 60354400, ratio = 6706044
priority = 10, exitcode = 67719600, ratio = 6771960
```

(b) Stride 调度结果

图 4.4 FSS 调度结果

图4.4(a)显示的 Lottery 调度结果大体上满足随 `prio` 的增大所得结果也增大的趋势，但 `ratio` 存在一定的波动。由于随机数生成的不确定性与所选的随机数生成方式的伪随机性，调度结果很难完全遵循数学期望。事实上，这一结果是运行

总时长为 12000ms 得到的，已经相对减小了随机性，最初的测试时长在 4000ms，甚至会得到 prio 增大 exitcode 反而减小的部分结果。但是可以预测的是，对于更长时间的进程，Lottery 可以逐渐维持 ratio 的稳定，更符合优先级正比的规律。

而图4.4(b)则是在 4000ms 总时长下得到的。可以看到，相比 Lottery，其 ratio 值非常稳定，更符合公平性的原则，因为 Stride 调度中的随机性很小，其每步调度都是可以计算的。

4.4 实时调度策略对比

```
static TESTS: &[&str] = &[
    "rms1\0",
    "rms2\0",
];

// static TIMES: [usize;2] = [
//     800,
//     500,
// ];

static PERIODS: [usize; 2] = [
    2000,
    1000,
];
```

首先用上面两个周期性测例对 RMS 调度进行测试，其运行时间与周期均给出。按照 RMS 调度准则，在前 500ms，任务 2 优先执行，之后 1 开始执行，但是到 1000ms 时任务 2 请求再次开始运行，因为任务 2 的周期更短，所以 1 被抢占，直到 1500ms 任务 2 运行结束。之后 1 恢复执行，到 1800ms 执行完成。此后到 2000ms 重复循环，可以实现这两个周期任务的正常运转。如图4.5(a)。

将二者的周期修改为如下值，

```
static PERIODS: [usize; 2] = [
    1500,
    1200,
];
```

计算此时的 CPU 利用率，为

$$\frac{800}{1500} + \frac{500}{1200} = 0.95 < 1 \quad (4.1)$$

理论上利用率小于 1 是可以被调度的。但是对两个进程而言，按照公式(3.7)

$$2(2^{1/2} - 1) = 0.828 < 0.95 \quad (4.2)$$

因此有无法调度的可能。事实也确实如此，如图4.5(b)所示，在 500ms 的任务 2 运行结束、任务 1 继续运行 700ms 时，任务 2 再次请求执行，优先被运行至 1700ms，此时任务 1 才能接手，但已经超出了其 1500ms 的周期，于是调度失败。

```
*****/
rms1 Arriving at 89
rms2 Arriving at 90
current time_msec = 26, I am rms2
798059
time_msec = 533, delta = 507ms, rms2 complete
current time_msec = 534, I am rms1
current time_msec = 1001, I am rms2
798059
time_msec = 1505, delta = 504ms, rms2 complete
562774
time_msec = 1840, delta = 1306ms, rms1 complete
current time_msec = 2002, I am rms2
```

(a) 成功调度结果

```
*****/
rms1 Arriving at 80
rms2 Arriving at 81
current time_msec = 6, I am rms2
798059
time_msec = 501, delta = 495ms, rms2 complete
current time_msec = 501, I am rms1
current time_msec = 1209, I am rms2
798059
time_msec = 1698, delta = 489ms, rms2 complete
562774
time_msec = 1775, delta = 1274ms, rms1 complete
Panicked at src/bin/rms1.rs:33, rms1 outlimit!
```

(b) 失败调度结果

图 4.5 RMS 调度结果

此时我们让 EDF 来调度这两个 RMS 无法处理的进程：

```
rms1 Arriving at 3095
rms2 Arriving at 3096
current time_msec = 21, I am rms2
798059
time_msec = 519, delta = 498ms, rms2 complete
current time_msec = 520, I am rms1
562774
time_msec = 1320, delta = 800ms, rms1 complete
current time_msec = 1320, I am rms2
798059
time_msec = 1815, delta = 495ms, rms2 complete
current time_msec = 1816, I am rms1
562774
time_msec = 2607, delta = 791ms, rms1 complete
current time_msec = 2608, I am rms2
798059
time_msec = 3105, delta = 497ms, rms2 complete
current time_msec = 3106, I am rms1
562774
time_msec = 3897, delta = 791ms, rms1 complete
current time_msec = 3897, I am rms2
798059
time_msec = 4394, delta = 497ms, rms2 complete
current time_msec = 4503, I am rms1
562774
time_msec = 5295, delta = 792ms, rms1 complete
current time_msec = 5296, I am rms2
798059
time_msec = 5792, delta = 496ms, rms2 complete
QEMU: Terminated
```

图 4.6 EDF 调度结果

可以看到，二者能够正常执行下去。EDF 调度与 RMS 的区别在于，初始时按照周期长短任务 2 优先于 1 执行，到 500ms 后任务 1 开始执行，虽然任务 2 在 1200ms 再次请求运行，但是此时任务 2 的 DDL 已经成为 $2*1200=2400>1500$ ，任务 1 优先于任务 2 了，于是任务 1 的第一次运行继续进行，正常结束。此后则动态变换二者优先级，最终使得调度成功。

所以，静态调度的 RMS 算法对于某些 CPU 利用率过高的情况无能为力，而动态调整优先级的 EDF 算法则可以达成理论 100% 的调度结果。

第 5 章 rCore-Tutorial 多核扩展

上一章在 rCore-Tutorial 实现了经典的单核调度算法，本章将尝试多核环境下的进程调度。由于 rCore-Tutorial 仅仅是单核操作系统，首要的任务便是将其扩展为多核。在本章节中，按照 Chapter 逐步将 rCore-Tutorial 扩展为完备的多核操作系统，支持单核下的各种功能，之后对多核的进程调度模块进行深入的分析，寻找潜在的问题，最后对多核相比于单核的速度提升进行测试分析。

5.1 从单核到多核

5.1.1 多核操作系统的启动

第一步是让 rCore-Tutorial 能在多核环境下正常启动。上一章提到，QEMU 模拟器为 rCore-Tutorial 模拟了 RISC-V 架构的硬件执行环境。若需要扩展到多核，则 QEMU 模拟器应该模拟一个多 CPU 的硬件平台。那么 QEMU 启动过程中，需要添加以下设置。

```
-smp $(SMP)
```

设定 SMP=4，QEMU 就会模拟四核处理器的启动。

下一步需要为内核编写多核启动程序。entry.asm 是内核的入口点，因此修改从这里开始。

```
.section .text.entry
.globl _start
_start:
    mv tp, a0

    add t0, a0, 1
    slli t0, t0, 16
    la sp, boot_stack
    add sp, sp, t0
    call rust_main

.section .bss.stack
.globl boot_stack
boot_stack:
    .space 4096 * 16 * 4
    .globl boot_stack_top
boot_stack_top:
```


开始时，a0 寄存器里有当前核的 id，将其存入 tp(thread pointer) 寄存器中，并新增加一个 harts 模块以汇编方法直接读取 tp 寄存器供内核使用；启动栈 'boot_stack' 由单一的 64K 变为了 64K*4，为四个核提供等同的启动栈。_start 设置每个核的栈指针位置，之后跳转入 rust_main 中。

rust_main 是内核 boot 的地点，是所有核共用的。但单核情况下的部分初始化工作只需进行一次，而另一部分初始化工作却需要每个核都要进行。这就要求对不同核要进行不同的行为设定。这里采用的做法是，设置一个 CONTROL_CPU 来进行只需一次的全局初始化，待其完成后，再让所有 CPU 共同进行通用初始化工作。

```
if cpu_id == CONTROL_CPU{
    println!("Global initialization start...");
    clear_bss();
    ..
    finish_global_init();
}
wait_global_init();
println!("Hello world from CPU {:x}!", cpu_id);
..
boot_finish();
wait_all_booted();
```

其中，finish_global_init() 发出全局初始化完成信号，wait_global_init() 接收到这一信号，取消其他核的阻塞，进行共同初始化工作；之后完成任务的核发出 boot_finish() 信号，然后通过 wait_all_booted() 等待所有核均启动完成。这样就保证了启动的同步性。这里使用了原子变量 AtomicBool 和 AtomicUsize 来判断是否全局初始化和记录完成核数，因为这两个信息被多核共享，有同步互斥的要求，而原子变量可以进行原子化操作，满足多核间的互斥需要。

最后要做的是规范 STDOUT 输出。多核开启后，会争抢同一 STDOUT 实例，导致输出会被字符级地截断。若对 STDOUT 加锁，则获取到的核可以完整地打印一块内容，释放锁后才会被其他核抢占。

```
static STDOUT: Mutex<Stdout> = Mutex::new(Stdout);

pub fn print(args: fmt::Arguments) {
    STDOUT.lock().write_fmt(args).unwrap();
    //Stdout.write_fmt(args).unwrap();
}
```

5.1.2 其他改动

事实上，rCore-Tutorial 具有较好的可扩展性。除了启动的改动和调度上的改动之外，多核所需的改动并不多。比较重要的有如下两点：

- 互斥结构的变更。单核 rCore-Tutorial 采用手写的 UPSafeCell 结构来实现所需的同步互斥要求，多核条件下需要把 UPSafeCell 修改为支持多核的互斥锁，这里采用 spin 库的 Mutex 锁，用 lock() 来实现互斥访问。诸如 PCB, Manager, Processor, 各种全局分配器等等都需要加上互斥锁。
- 单核的内存模块初始化是一步调用的，同时进行了页帧分配器 frame_allocator 和内核地址空间 kernel_space 的初始化。在多核条件下，frame_allocator 只需一次全局初始化，而内核空间需要每个核进行，那么二者要拆分开来，分别放置于上述 rust_main 的两处 “..” 部分。

5.2 多核系统的进程调度

多核操作系统的进程管理模块相比单核需要有比较大的修改。以单队列调度——也就是全局只有一个就绪队列，各个核都访问该队列来获取进程——为基础，需要对 rCore-Tutorial 进行相当一部分改动。

5.2.1 多处理机

前文提到，处理器管理结构 Processor 是 CPU 的抽象，那么当 CPU 的数量从一变为多个时，其基本结构也需要改动。

之前全局初始化的 PROCESSOR 仅是一个结构体，而现在它需要成为多个处理机的集合。

```
lazy_static! {  
    pub static ref PROCESSOR:Vec<Mutex<Processor>> = {  
        let mut pro_vec = Vec::new();  
        for i in 0..CPU_NUM{  
            pro_vec.push(Mutex::new(Processor::new()))  
        }  
        pro_vec  
    };  
}
```

以 Vec 数据结构存储各个 Processor 状态。每次需要获取时，首先要通过 harts 的 id() 方法读取到当前 CPU 的 id，再以 PROCESSOR[id()] 来获取对应的 CPU。

如果仅进行这一变化，看上去各个处理机的运行似乎互不干扰，但在实际的

测试过程中，会发现有时可能有内核的 PageFault，有时可能会卡死不动，说明多核环境还存在着冲突部分等待寻找。

5.2.2 switch 的安全保证

回顾第二章描述的进程的执行与切换过程，switch 是很重要的一部分。由于 switch 需要进行寄存器、栈指针的保存和 pc 的变更，该方法的主体是由汇编来实现的。

```
__switch:
    # 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
    # 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)
    ret
```

如注释所述，switch 首先保存了当前控制流下的内核栈、寄存器、返回地址，再恢复要切换的控制流的对应数据，最后使用 ret 完成控制流的切换。但是，由于 switch 这一方法一旦调用，基本就会离开当前函数体，所以调用者所持有的锁也需要在调用 switch 之前被释放掉。那么，回顾 suspend_current_and_run_next() 这一启动调度的方法，其执行顺序为暂停进程、加回队列、switch 切换，在加回队列与切换之间需要释放 current 进程的锁，再进行 switch。但此时，进程已被加回就绪队列，若另一 CPU 恰好又取出了该进程，在 run_tasks() 开始执行，同样执行到了 switch，那么若前一 CPU 的 switch 的 save 阶段还未完成，后者便开始进行这一进程的 restore 阶段，如图 5.1，则后一 CPU 就读取到了错误的数据，从而使两个处理机的执行产生冲突与错误，内核崩溃。

为了解决这一问题，思路主要是让前一 CPU 尽快完成 save，或者让后一 CPU 较晚取出进程。采用后一种思路，由于 suspend_current_and_run_next() 后面


```

1  let mut inner = task.inner_exclusive_access();
2  ..
3  let mut initproc_inner = INITPROC.inner_exclusive_access();
4  for child in inner.children.iter() {
5      child.inner_exclusive_access().parent =
6          Some(Arc::downgrade(&INITPROC));
7      initproc_inner.children.push(child.clone());
8  }

```

考虑一种情况，父子进程同时进入了 `exit` 中。父进程稍快一些，它拿到了 `INITPROC` 的锁，准备访问子进程；但此时子进程刚执行到 2 行，在上面持有自己的锁，于是父进程将等待。但是子进程的下一步是获取 `INITPROC`，而它又被父进程占有，于是产生死锁。

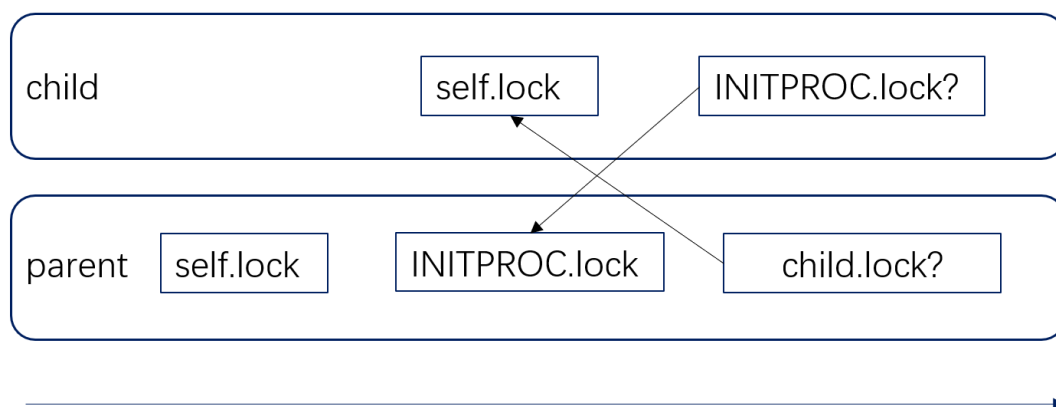


图 5.2 `exit` 存在的死锁

调整一下获取锁的顺序，比如将获取 `INITPROC` 放到获取当前任务 `inner` 的前面进行，这样同一时间只能有一个处理器获取到 `INITPROC` 的锁来进行下面的操作，从而解决的父子锁的冲突。

```

let mut initproc_inner = INITPROC.inner_exclusive_access();
..
let mut inner = task.inner_exclusive_access();
..
for child in inner.children.iter() {
    child.inner_exclusive_access().parent =
        Some(Arc::downgrade(&INITPROC));
    initproc_inner.children.push(child.clone());
}
..
drop(initproc_inner);

```

5.3 单核与多核调度对比

为了检验多核调度的实现，对单核和多核（四核）下的进程运行情况进行分析比较。测试机器为八核 ubuntu，在其上启动 QEMU 模拟器，为 rCore-Tutorial 模拟出四核处理器的硬件平台。

样例程序是一个运行时长大致在 1000ms 左右的计算程序，同时启动 1, 2, 3, 4, 5, 10 个样例进程，统计进程从到达完成所用时间如下：

表 5.1 样例程序组在单核和多核环境下的执行情况^①

样例程序数目	单核平均耗时/ms	四核平均耗时/ms
1	993(1032)	1100(1143)
2	1967(2043)	1108(1155)
3	2956(3080)	1084(1130)
4	3942(4110)	1068(1121)
5	4912(5125)	1318(1426)
10	9800(10217)	2600(2734)

^① 括号里的数值为包括启动程序和回收子进程的总用时。

对于单核调度，遵循耗时和程序数目的正比关系，即循环执行每个任务，最终用时为单一时间与个数的乘积。

对于四核调度，平均耗时在程序不超过 4 个时相差不大，符合四核逻辑，且应是在数目为 4 时才充分利用了 CPU 资源。当进程数达到 5 个时，多出一个进程被四个处理器平摊，总耗时应增加了单一进程的 $\frac{1}{4}$ ，与表中数值基本一致。当进程数大于 5 后，耗时将以线性关系增长，到达 10 后耗时为 5 的一倍，与表中数据一致。

比较单核和多核调度可以发现，当进程数为 1 时，单核耗时要小于多核，这与多核之间的进程切换开销大于单核的结论一致。当进程数增多到 4 期间，线性的单核耗时和基本不变的多核耗时相差越来越大。进程数在 5 以上时，虽然二者都按比例增长，但四核环境下的耗时增长仅为单核的 $\frac{1}{4}$ ，仍有非常大的差距。

第 6 章 总结

对 rCore-Tutorial 进程调度的拓展是一个全新的尝试。对于教学用操作系统，本项目不再局限于单纯的功能增加，而是对已有的功能向着课程内容进行更广泛且深入的拓展。在实现了如此多单核调度之后，rCore-Tutorial 的进程管理模块被大大充实，能为使用者提供更完整的进程调度参考。当然，仍有部分单核调度算法没有被本文所实现，而且对于已实现的进程调度的完备性还需要进一步的检验与测试，这些都是下一步工作可以开展的方向。

将 rCore-Tutorial 拓展为多核 OS 也是现实的需要。一个多处理机的教学系统更能贴合当下的计算机实际；另一方面，在已实现的多核 rCore-Tutorial 基础上，还可以进行更多的多核功能扩展，为 rCore-Tutorial 的进一步完善开辟了新的道路。同时，从单核到多核这一执行环境的巨大转变也为原有的 rCore-Tutorial 模块提出了新的要求，还有一些潜在的问题等待被发现和解决。

插图索引

图 1.1	进程的五状态模型	2
图 4.1	SJF 与 STCF 调度对比	22
图 4.2	MQ 调度结果.....	23
图 4.3	MLFQ 调度结果	23
图 4.4	FSS 调度结果	24
图 4.5	RMS 调度结果	26
图 4.6	EDF 调度结果.....	26
图 5.1	switch 的安全问题.....	32
图 5.2	exit 存在的死锁	33

表格索引

表 5.1 样例程序组在单核和多核环境下的执行情况	34
---------------------------------	----

参考文献

- [1] Chen Y, Wu Y F. rcore-tutorial-book-v3[EB/OL]. [2022-01-09]. <https://rcore-os.github.io/rcore-Tutorial-Book-v3>.
- [2] Cox R, Kaashoek F, Morris R. xv6: a simple, unix-like teaching operating system[J]. 2021.
- [3] Silberschatz A, Galvin P B, Gagne G. Operating system concepts[M]. John Wiley & Sons, 2006.
- [4] Arpaci-Dusseau R H, Arpaci-Dusseau A C. Operating systems: Three easy pieces[M]. CreateSpace Independent Publishing Platform, 2018.
- [5] Waldspurger C A, Weihl W E. Lottery scheduling: Flexible proportional-share resource management[C]//Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation. 1994: 1-es.
- [6] Waldspurger C A, Weihl W E. Stride scheduling: Deterministic proportional share resource management[M]. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.
- [7] Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment[J]. Journal of the ACM (JACM), 1973, 20(1): 46-61.

致 谢

衷心感谢导师陈渝副教授在项目完成过程中对我的悉心指导。陈老师提供的思路在我的研究过程中起到了重要的作用，整个工作的顺利完成也离不开陈老师对我所遇到的问题的解答。

感谢计算机系和辅导员在疫情期间所做的工作，让我能在疫情影响下减少不必要的顾虑，顺利完成毕业设计。

感谢家人和朋友们，让我始终能在整个项目过程中保持着积极向上的状态，充满着探索的动力。

感谢四年来遇到的每一位老师与同学，陪伴我度过这段美好的时光。

声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名：_____ 日 期：_____

附录 A 外文资料的书面翻译

操作系统历险记：使用 Rust 编写 RISC-V 操作系统

目录

A.0 环境配置与依赖安装	44
A.0.1 2020 年更新内容	44
A.0.2 构建与运行	45
A.0.3 完成!	45
A.1 掌控 RISC-V	45
A.1.1 概述	45
A.1.2 选择一个引导加载程序 (bootloader)	46
A.1.3 清除 BSS	47
A.1.4 进入 Rust 编程	48
A.1.5 裸机 Rust 的世界!	49
A.1.6 我们真正进入了 RUST!	51
A.2 通信	52
A.2.1 视频	52
A.2.2 概述	52
A.2.3 获取代码	52
A.2.4 通用异步收发传输器 (UART)	52
A.2.5 内存映射 I/O (MMIO)	53
A.2.6 NS16550a 寄存器	53
A.2.7 Rust 的 MMIO	54
A.2.8 寄存器图表	55
A.2.9 设置字长为 8 位	56
A.2.10 启用 FIFO	56
A.2.11 启用接收器中断	56
A.2.12 设置信号传输率 (BAUD)	56
A.2.13 读取 UART	58

A.2.14	写入 UART.....	58
A.2.15	加入到 Rust 主程序.....	59
A.2.16	我们来写 <code>println!</code>	60
A.2.17	为什么 <code>println</code> 有 3 个手臂?	62
A.2.18	为什么每次都要创建一个新的 <code>UartDriver</code> ?	62
A.2.19	将 UART 加载到 <code>lib.rs</code> 中.....	62
A.2.20	说出来, 我的小家伙.....	65
A.2.21	认真听, 我的小家伙.....	65
A.2.22	测试!.....	67
A.2.23	最后的控制台操作.....	67
A.2.24	哇, 这个代码好 <code>dirty</code>	69
A.2.25	通信是关键.....	70

本科生的外文资料书面翻译。

A.0 环境配置与依赖安装

2019 年 9 月 27 日

A.0.1 2020 年更新内容

Rust 现在对 RISC-V 的支持使我们能开箱即用！我们不通过工具链也能很容易地来构建项目。但在工具链之外，仍需要 QEMU（模拟器），在 `ch0.old.html` 的旧版教程中有相关介绍。

•

Rust 要求我们使用 ‘`rustup`’ 命令安装一些依赖。如果你没有 `rustup`，请在 <https://www.rust-lang.org/tools/install> 下载。

```
rustup default nightly
rustup target add riscv64gc-unknown-none-elf
cargo install cargo-binutils
```

由于我在系统中使用了语言特性（表示为 `#![features]`），我们必须使用 `nightly` 版本，即使 RISC-V 运行在稳定版本上。

A.0.2 构建与运行

在博客的 `git` 仓库中 (<https://github.com/sgmarz/osblog>) 你会看到一个名为 `cargo/config` 的文件。你可以根据你的系统来编辑这个文件：

```
[build]
target = "riscv64gc-unknown-none-elf"
rustflags = ['-Clink-arg=-Tsrc/lds/virt.lds']

[target.riscv64gc-unknown-none-elf]
runner = "qemu-system-riscv64 \
    -machine virt \
    -cpu rv64 \
    -smp 4 \
    -m 128M \
    -drive if=none,format=raw,file=hdd.dsk,id=foo \
    -device virtio-blk-device,scsi=off,drive=foo \
    -nographic \
    -serial mon:stdio \
    -bios none \
    -device virtio-rng-device \
    -device virtio-gpu-device \
    -device virtio-net-device \
    -device virtio-tablet-device \
    -device virtio-keyboard-device \
    -kernel "
```

配置文件显示了我们要构建的目标，也就是 `riscv64gc`。我们还需要指定我们的链接器脚本 `src/lds/virt.lds`，以便我们能确保正确的文件位置。最后，当我们输入 `cargo run` 时，会调用一个“runner”，它将运行 `riscv64 qemu`。另外注意在 `-kernel` 后面有一个空格，这是因为 `cargo` 会自动指定通过 `Cargo.toml` 配置的可执行文件。

A.0.3 完成！

这部分确实很无聊，如果配置过程一切顺利（在 ArchLinux 上配置很正常！），接下来的部分将有趣起来。

A.1 掌控 RISC-V

2019 年 9 月 27 日

A.1.1 概述

启动并进入 RISC-V 系统很简单。在各种方法中，我将要介绍我自己的方法——从物理内存地址 `0x8000_0000` 开始。幸运的是，QEMU 可以读取 ELF 文件，

所以它知道应该把我们的代码运行在哪个地址上。在整个过程中，我们将通过查看 QEMU 中包含的 `qemu/hw/riscv/virt.c` 源代码来获取一些信息。首先，先来看一下内存映射：

```
static const struct MemmapEntry {
    ^Ihwaddr base;
    ^Ihwaddr size;
} virt_memmap[] = {
    ^I[VIRT_DEBUG] = { 0x0, 0x100 },
    ^I[VIRT_MROM] = { 0x1000, 0x11000 },
    ^I[VIRT_TEST] = { 0x100000, 0x1000 },
    ^I[VIRT_CLINT] = { 0x2000000, 0x10000 },
    ^I[VIRT_PLIC] = { 0xc000000, 0x4000000 },
    ^I[VIRT_UART0] = { 0x10000000, 0x100 },
    ^I[VIRT_VIRTIO] = { 0x10001000, 0x1000 },
    ^I[VIRT_DRAM] = { 0x80000000, 0x0 },
    ^I[VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
    ^I[VIRT_PCIE_PIO] = { 0x03000000, 0x00010000 },
    ^I[VIRT_PCIE_ECAM] = { 0x30000000, 0x10000000 },
};
```

由此可见，我们的机器从 DRAM (VIRT_DRAM) 的第 0 字节，地址 `0x8000_0000` 开始。当我们再往前走一点，我们将对 CLINT (`0x200_0000`)、PLIC (`0xc00_0000`)、UART (`0x1000_0000`) 和 VIRTIO (`0x1000_1000`) 编程。现在不要担心这些是什么意思，我们只需要看看接下来要做什么！

完成这些后，我们需要在 RISC-V 汇编中完成以下工作：

1. 选择一个 CPU 引导加载程序（通常是 `id#0`）；
2. 将 BSS 部分清零；
3. 开始 Rust!

RISC-V 汇编类似于 MIPS 汇编，除了我们不需要给我们的寄存器加前缀。所有的指令都来自于 RISC-V 规范，你可以在以下网站获取：<https://github.com/riscv/riscv-isa-manual>。我们的编写对象是 RV64GC（RISC-V 64 位，一般扩展和压缩指令扩展）。

A.1.2 选择一个引导加载程序 (bootloader)

这个时候我们不考虑并行性、条件竞争或其他此类问题。相反，我们只让我们的一个 CPU 核心（RISC-V 中称为 HARTs[硬件线程]）做所有的工作。现在我们首先要深入研究特权级规范，来弄清我们要讨论的是哪个寄存器。因此，请在<https://github.com/riscv/riscv-isa-manual>下载。

我们将从 3.1.5 章中开始（Hart ID 寄存器 `mhartid`）。这个寄存器将告诉我们的 `hart` 编号。根据规范，我们必须有一个 `hart id #0`。所以，我们以这个 ID 来启动。

在你的 `src/asm/` 目录下创建一个 `boot.S` 文件。我们将在此启动并进入 Rust 的编写。

```
# boot.S
# bootloader for SoS
# Stephen Marz
# 8 February 2019
.option norvc
.section .data

.section .text.init
.global _start
_start:
    # Any hardware threads (hart) that are not bootstrapping
    # need to wait for an IPI
    csrr    t0, mhartid
    bnez    t0, 3f
    # SATP should be zero, but let's make sure
    csrw    satp, zero
.option push
.option norelax
    la      gp, _global_pointer
.option pop

3:
    wfi
    j       3b
```

这里 `csrr` 的意思是”控制状态寄存器读取”，利用 `csrr` 我们把 `hart` 标识符读到寄存器 `t0` 中，检查它是否为零。如果不是，我们就把它送去循环等待。之后，我们将监管者地址转换和保护（`satp`）寄存器设置为 0，这就是我们最终控制 MMU 的方式。由于我们还没有对虚拟内存的需求，我们用 `csrw`（控制状态寄存器写入）写入 0 来把它禁用掉。一些板子的复位向量会将 `mhartid` 加载到该板子的 `a0` 寄存器中，但有些板子或许不会这么做，所以我选择从最可靠的地方来获取 `hart ID`。

A.1.3 清除 BSS

全局的、未初始化的变量的初值都是 0，因为这些变量是在 BSS 段分配的。对于操作系统而言，我们要保证内存全是 0。幸运的是，在我们的链接器脚本中定义了 `_bss_start` 和 `_bss_end` 这两个字段，分别告诉我们 BSS 部分的开始和结束位置。因此，我们在 `.option pop` 和 3 之间添加以下内容：

```

# The BSS section is expected to be zero
    la      a0, _bss_start
    la      a1, _bss_end
    bgeu    a0, a1, 2f
1:
    sd      zero, (a0)
    addi    a0, a0, 8
    bltu    a0, a1, 1b
2:

```

在这里，我们使用 `sd`（双字 [64 位] 存储）将 0 存到内存地址 `a0`，该地址逐渐向 `_bss_end` 移动。

A.1.4 进入 Rust 编程

由于许多人都不喜欢编写大量的汇编，我们尽快跳入 Rust，虽然也会有人认为用 Rust 编程是最难的部分。我们的 Rust 不会那么难，不会一直和借用检查器 (borrow checker) 较劲。

为了进入 Rust 程序并使 CPU 处于一个确定的模式，我们将使用 `mret` 指令，这是一个陷入返回函数，允许我们将 `mstatus` 寄存器设置为我们的特权模式。因此，我们在 `boot.S` 中加入以下内容。

```

# Control registers, set the stack, mstatus, mepc,
# and mtvec to return to the main function.
# li      t5, 0xffff;
# csrw    medeleg, t5
# csrw    mideleg, t5
la      sp, _stack
# We use mret here so that the mstatus register
# is properly updated.
li      t0, (0b11 << 11) | (1 << 7) | (1 << 3)
csrwr    mstatus, t0
la      t1, kmain
csrwr    mepc, t1
la      t2, asm_trap_vector
csrwr    mtvec, t2
li      t3, (1 << 3) | (1 << 7) | (1 << 11)
csrwr    mie, t3
la      ra, 4f
mret
4:
    wfi
    j      4b

```

这段代码很长，中间还有些注释。我们要做的是将 [12:11] 位设置为 11，也

就是”机器模式 (machine mode)”。这将使我们能够访问所有的指令和寄存器。当然，我们或许已经处于这一模式了，但还是做一次比较好。

> 之后位 [7] 和位 [3] 将在总体上启用中断。然而，我们仍然需要通过 mie（机器中断使能）寄存器启用特定的中断，这一点在最后几行实现。

mepc 寄存器是”机器异常程序计数器”，它是我们要返回的内存地址。符号 kmain 是在 Rust 中定义的，是我们离开汇编、进入 Rust 的“逃生”通道。

mtvec（机器陷入向量）是一个内核函数，每当有陷入出现时就会被调用，比如系统调用、非法指令或者时钟中断。

在我们完成 Rust 的主函数后，我们将 ra（返回地址）置为等待状态。然后，mret 指令将我们刚才所做的一切，通过 mepc 寄存器跳回，这就是我们最终进入 Rust 的地方！

（2019 年 9 月 29 日添加）我们已经参考了 asm_trap_vector，但我们还没有实际编写它。不过我们很快就会开工，现在先在 src/asm/下创建一个名为 trap.S 的文件，并在其中添加以下内容。

```
# trap.S
# Assembly-level trap handler.
.section .text
.global asm_trap_vector
asm_trap_vector:
    # We get here when the CPU is interrupted
    # for any reason.
    mret
```

A.1.5 裸机 Rust 的世界！

现在我们已经进入了 rust。首先我们需要编辑 lib.rs，这个文件是由 cargo 命令为我们创建的。不要改变 lib.rs 的名称，否则 cargo 将永远不知道我们在干什么。lib.rs 将是我们的入口点与我们用来导入其他 Rust 模块的工具。不要把 kmain 看成是一段执行代码。相反，它将初始化我们所需要的一切，然后引起”大爆炸”，也就是说让所有代码都开始运行。操作系统主要是异步的，我们将使用时钟中断来指示内核开始运行，所以我们不能使用平时习惯的单线程编程方法。

当第一次打开 lib.rs 时，清空文件，因为里面没有我们需要用于内核的东西。我们需要自己定义一些东西来满足 Rust 的要求。由于我们不会使用标准库（标准库自己就依赖于内核，不能被用于我们的内核构建），我们必须首先定义 abort 和 panic_handler 再去做别的事情。就像这样：

```
// Steve Operating System
```

```

// Stephen Marz
// 21 Sep 2019
#![no_std]
#![feature(panic_info_message,asm)]

// //////////////////////////////////////
// / RUST MACROS
// //////////////////////////////////////
#[macro_export]
macro_rules! print
{
    ($($args:tt)+) => ({

        });
}
#[macro_export]
macro_rules! println
{
    () => ({
        print!("\r\n")
    });
    ($fmt:expr) => ({
        print!(concat!($fmt, "\r\n"))
    });
    ($fmt:expr, $($args:tt)+) => ({
        print!(concat!($fmt, "\r\n"), $($args)+)
    });
}

// //////////////////////////////////////
// / LANGUAGE STRUCTURES / FUNCTIONS
// //////////////////////////////////////
#[no_mangle]
extern "C" fn eh_personality() {}
#[panic_handler]
fn panic(info: &core::panic::PanicInfo) -> ! {
    print!("Aborting: ");
    if let Some(p) = info.location() {
        println!(
            "line {}, file {}: {}",
            p.line(),
            p.file(),
            info.message().unwrap()
        );
    }
    else {
        println!("no information available.");
    }
    abort();
}
#[no_mangle]

```

```
extern "C"
fn abort() -> ! {
    loop {
        unsafe {
            // The asm! syntax has changed in Rust.
            // For the old, you can use llvm_asm!, but the
            // new syntax kicks ass--when we actually get to
            // use it.
            asm! ("wfi");
        }
    }
}
```

我们使用 `#![no_std]` 来告诉 Rust 不会使用标准库。然后我们要求 Rust 允许我们的代码使用 `panic` 信息和内嵌式汇编功能。第一件要做的事是创建一个空的 `eh_personality` 函数。`#[no_mangle]` 关闭了 Rust 的名称处理功能，所以这个符号确实是 `eh_personality`。然后，`extern "C"` 告诉 Rust 使用 C 风格的 ABI。

之后，`#[panic_handler]` 告诉 Rust，我们定义的下一个函数将是我们的 `panic` 处理程序。Rust 调用 `panic` 有几个原因，我们将用我们的断言隐式地调用它。我让这个函数做的是打印出产生 `panic` 的源文件和行号。虽说我们还没有实现 `print!` 或 `println!`，但我们知道 Rust 中 `print` 和 `println` 的格式。顺便提一下，`->!` 意味着这个函数不会返回。如果 Rust 检测到它可以返回，编译器会报错。

最后，我们编写 `abort` 函数。它要做的就是不断循环 `wfi`（等待中断）指令。这使它正在运行的 `hart` 关闭，直到另一个中断发生。

A.1.6 我们真正进入了 RUST!

我们已经正式进入 rust，所以我们需要写出 `boot.S` 中指定的入口点，也就是 `kmain`。所以，在我们的 `lib.rs` 代码中添加：

```
#[no_mangle]
extern "C"
fn kmain() {
    // Main should initialize all sub-systems and get
    // ready to start scheduling. The last thing this
    // should do is start the timer.
}
```

当 `kmain` 返回时，它碰到 `wfi` 循环并挂起。这是我们所期望的，因为我们还没有告诉内核要做什么。

那么基本已经完工了，我们进入了 Rust 之中。不幸的是，在我们实现 `print!` 之前不会看到任何打印到屏幕上的东西。但是，现在的代码至少应该是能够正常

编译的！一般来说，优秀的作家会以一些引言或结束语来结束，但我并不是一位优秀的作家。

当你输入 `make run` 时，你的操作系统将尝试启动并进入 `Rust`。然而由于我们还没有编写与操作系统通信的驱动程序，所以什么也不会发生。输入 `CTRL-A+'x'` 退出模拟器。同时，你也可以通过输入 `CTRL-A+'c'` 来查看你所在的位置。你现在是在 `QEMU` 的控制台。输入 `"info registers"` 来查看模拟器在你的操作系统中的位置。

A.2 通信

2019 年 10 月 4 日：仅 `Patreon`

2019 年 10 月 6 日：公开

A.2.1 视频

带解说的教学视频可以在以下网站找到：<https://www.youtube.com/watch?v=1SR1sB8W248>。

A.2.2 概述

在什么都显示不出来的情况下，我们编写的这个操作系统有什么优点？简单来说确实没有。但我们已经让这个小家伙诞生了，并给它起了个名字，所以我们要进一步培养它！

我们将开始构建操作系统的各个部分，它们将像拼图一样，在之后组合在一起。有些会在相当早的时候就装配好（控制台和 `UART` 驱动有紧密的结合），有些会在以后组装。请耐心等待下去，我们最终一定会成功的！

A.2.3 获取代码

为了更容易跟上进度，请到<https://github.com/sgmarz/osblog>获取第二章的代码。

A.2.4 通用异步收发传输器（UART）

我们将首先使用 `UART`（通用异步收发传输器）进行通信。我们通过 `QEMU` 使用的虚拟机模拟了 `NS16550A` `UART` 芯片组。我们使用内存映射的 `I/O` 在地址 `0x1000_0000` 处控制这个 `UART` 系统。在这里，我们可以控制 `NS16550a` 的寄存器，它们都是 8 位的。

A.2.5 内存映射 I/O (MMIO)

基本上，我们要做的是设置一个指向地址 0x1000_0000 的指针。当我们写到这个地址或这个地址的某个偏移量时，我们实际上是在与 UART 设备而不是物理 RAM 对话。这是一个在嵌入式系统中众所周知的系统，称为 MMIO，也即内存映射的输入和输出。

A.2.6 NS16550a 寄存器

寄存器只是位于硬件设备上的一点内存。对于 NS16550a 来说，这个内存是一个字节一个字节地访问的。例如，在下面的寄存器布局中（见“寄存器图表”），你可以看到发送器（THR）和接收器（RBR）寄存器都正好是 8 位（1 字节）。当我从指向 0x10000000 的指针读取时，我从 RBR 中提取这 8 位。当我写到指向 0x10000000 的完全相同的指针时，我将会向 THR 发送。

这个 MMIO 系统很好，因为它不需要我们做除了从内存地址读写之外的任何特别的事情。由于 Rust 支持裸指针，它使我们能够相当容易地控制硬件。在 C++ 中，我们可以编写下面的内容来读写传输器。

```
void mmio_write(unsigned long address, int offset, char value)
{
    // We use volatile so that the optimizer on the
    // compiler doesn't think we're writing a value for
    // no reason.
    volatile char *reg = (char *)address;

    // To write, the pointer is on the left hand side of
    // the assignment operator.
    // Transmit the word 'A'
    // (which is the 8-bit value 65 or 0b0100_0001)
    *(reg + offset) = value;
}

char mmio_read(unsigned long address, int offset)
{
    // We use volatile so that the optimizer on the
    // compiler doesn't think we're writing a value for
    // no reason.
    volatile char *reg = (char *)address;

    // To read, the pointer is dereferenced on a read.
    // This reads from the receiver and returns the data.
    return *(reg + offset);
}
```

A.2.7 Rust 的 MMIO

Rust 确实有裸指针，但它没有 `volatile` 关键字。我们将指针作为对象使用，并使用 Rust 裸指针的 `write_volatile` 和 `read_volatile` 成员函数。比如说：

```
/// # Safety
///
/// We label the mmio function unsafe since
/// we will be working with raw memory. Rust cannot
/// make any guarantees when we do this.
fn unsafe mmio_write(address: usize, offset: usize, value: u8)
{
    // Set the pointer based off of the address
    let reg = address as *mut u8;

    // write_volatile is a member of the *mut raw
    // and we can use the .add() to give us another pointer
    // at an offset based on the original pointer's memory
    // address. NOTE: The add uses pointer arithmetic so it is
    // new_pointer = old_pointer + sizeof(pointer_type) *
    // offset
    reg.add(offset).write_volatile(value);
}

/// # Safety
///
/// We label the mmio function unsafe since
/// we will be working with raw memory. Rust cannot
/// make any guarantees when we do this.
fn unsafe mmio_read(address: usize, offset: usize, value: u8)
-> u8 {
    // Set the pointer based off of the address
    let reg = address as *mut u8;

    // read_volatile() is much like write_volatile() except it
    // will grab 8-bits from the pointer and give that value
    // to us.
    // We don't add a semi-colon at the end here so that the
    // value is "returned".
    reg.add(offset).read_volatile()
}
```

如果你对 `add` 感到奇怪，它是 Rust 对普通的 `ptr.offset` 方法的一个封装，但是 `offset` 的参数是 `isize`，意味着它可以是正的或负的。为了清楚起见，我使用了基础寄存器的 `.add()`，但你可以选择你想用的那种。

我在 `mmio_read` 的注释中提到了不加分号的问题。Rust 主要是一种“表达式语言”，正如这里所讲的：<https://doc.rust-lang.org/reference/statements-and-expressions>

ns.html。本质上，Rust 会将函数归结为返回值，对于这个函数来说，返回部分是一个 u8（无符号 8 位整数）。当我们加上分号时，我们告诉 Rust 放弃求值的结果，并返回 () 类型，Rust 称之为单元（unit）类型（见<https://doc.rust-lang.org/stable/rust-by-example/expression.html>）。

A.2.8 寄存器图表

NS16550a 的寄存器如下：

Bit No.	REGISTER ADDRESS											
	0 DLAB=0	0 DLAB=0	1 DLAB=0	2	2	3	4	5	6	7	0 DLAB=1	1 DLAB=1
	Receiver Buffer Register (Read Only)	Transmitter Holding Register (Write Only)	Interrupt Enable Register	Interrupt Ident. Register (Read Only)	FIFO Control Register (Write Only)	Line Control Register	MODEM Control Register	Line Status Register	MODEM Status Register	Scratch Register	Divisor Latch (LS)	Divisor Latch (MS)
	RBR	THR	IER	IIR	FCR	LCR	MCR	LSR	MSR	SCR	DLL	DLM
0	Data Bit 0 ⁽¹⁾	Data Bit 0	Enable Received Data Available Interrupt (ERBFI)	"0" if Interrupt Pending	FIFO Enable	Word Length Select Bit 0 (WLS0)	Data Terminal Ready (DTR)	Data Ready (DR)	Delta Clear to Send (DCTS)	Bit 0	Bit 0	Bit 8
1	Data Bit 1	Data Bit 1	Enable Transmitter Holding Register Empty Interrupt (ETBEI)	Interrupt ID Bit (0)	RCVR FIFO Reset	Word Length Select Bit 1 (WLS1)	Request to Send (RTS)	Overrun Error (OE)	Delta Data Set Ready (DDSR)	Bit 1	Bit 1	Bit 9
2	Data Bit 2	Data Bit 2	Enable Receiver Line Status Interrupt (ELSI)	Interrupt ID Bit (1)	XMIT FIFO Reset	Number of Stop Bits (STB)	Out 1	Parity Error (PE)	Trailing Edge Ring Indicator (TERI)	Bit 2	Bit 2	Bit 10
3	Data Bit 3	Data Bit 3	Enable MODEM Status Interrupt (EDSSI)	Interrupt ID Bit (2) ⁽²⁾	DMA Mode Select	Parity Enable (PEN)	Out 2	Framing Error (FE)	Delta Data Carrier Detect (DDCD)	Bit 3	Bit 3	Bit 11
4	Data Bit 4	Data Bit 4	0	0	Reserved	Even Parity Select (EPS)	Loop	Break Interrupt (BI)	Clear to Send (CTS)	Bit 4	Bit 4	Bit 12
5	Data Bit 5	Data Bit 5	0	0	Reserved	Stick Parity	0	Transmitter Holding Register (THRE)	Data Set Ready (DSR)	Bit 5	Bit 5	Bit 13
6	Data Bit 6	Data Bit 6	0	FIFOs Enabled ⁽²⁾	RCVR Trigger (LSB)	Set Break	0	Transmitter Empty (TEMT)	Ring Indicator (RI)	Bit 6	Bit 6	Bit 14
7	Data Bit 7	Data Bit 7	0	FIFOs Enabled ⁽²⁾	RCVR Trigger (MSB)	Divisor Latch Access Bit (DLAB)	0	Error in RCVR FIFO ⁽²⁾	Data Carrier Detect (DCD)	Bit 7	Bit 7	Bit 15

图 A.1 NS16550a 寄存器

这个寄存器图的重要部分是，发送器和接收器都在 0x1000_0000。当我们向这个地址写 8 位时，它把这个字符放到发送器中。当我们从这个地址读出 8 位时，它从接收器中送出一个字符。

在我们发送和接收之前，我们必须对 UART 控制器进行如下设置。

- 设置字长为 8 位（LCR[1:0]）。
- 启用 FIFO（FCR[0]）。
- 启用接收器中断（IER[0]）

A.2.9 设置字长为 8 位

字长描述了发送器和接收器的缓冲区一次可以包含多少位。之所以可以设置，是因为即使我们可以同时支持 8 位，但另一边的 UART 可能无法支持。然而，QEMU 是支持 8 位的，所以设为 8 位就是我们要做的。在线路控制寄存器（LCR）中，有两位用来控制字长。根据规范，如果我们在这两个位上各写一个 1，我们就会得到一个 8 位的字符长度。默认情况下，如果我们在每个位中写 0，我们得到会是一个 5 位的字长，这有点奇怪。

A.2.10 启用 FIFO

FIFO 通常被实现为一个硬件移位寄存器，允许一次存储多个字节。FIFO 是先入先出 (First-In, First-Out) 的缩写。当数据被添加到 FIFO 中时，它的顺序不变，所以当我从同一个 FIFO 中读取时，我得到的是第一条输入的数据。这很不错，因为当有人输入“hello”时，我们一次读出一个字节，可以按序得到 h、e、l、l、o。

A.2.11 启用接收器中断

启用接收器中断意味着每当数据被添加到接收器中时，CPU 就会通过中断得到通知。在本章中，我们将不实际处理中断，因为这需要我们对平台级中断控制器（PLIC）进行编程，我们将在后面进行。相反，在本章中，我们将对 UART 进行“轮询”，看数据何时到达。

A.2.12 设置信号传输率（BAUD）

由于这不是一个真正的 UART（只是模拟的），我们实际上不需要设置分频数来划分时钟以确定一个特定的波特率。我们的做法是是将 DLAB（除数锁存器访问位）设置为 1。然后基址 +0 和基址 +1 现在分别是除数的下 8 位和上 8 位。然后，通过将 DLAB 位清除为 0 来关闭锁存器。

```
/// Initialize the UART driver by setting
/// the word length, FIFOs, and interrupts
pub fn uart_init(base_addr: usize) {
    let ptr = base_addr as *mut u8;
    unsafe {
        // First, set the word length, which
        // are bits 0, and 1 of the line control register
        // (LCR) which is at base_address + 3
        // We can easily write the value 3 here or 0b11, but
        // I'm extending it so that it is clear we're setting
        // two individual fields
        //           Word 0           Word 1
```

```

//          ~~~~~          ~~~~~
let lcr = (1 << 0) | (1 << 1);
ptr.add(3).write_volatile(lcr);

// Now, enable the FIFO, which is bit index 0 of the
// FIFO control register (FCR at offset 2).
// Again, we can just write 1 here, but when we use
// left shift, it's easier to see that we're trying
// to write bit index #0.
ptr.add(2).write_volatile(1 << 0);

// Enable receiver buffer interrupts, which is at bit
// index 0 of the interrupt enable register
// (IER at offset 1).
ptr.add(1).write_volatile(1 << 0);

// If we cared about the divisor, the code below
// would set the divisor from a global clock rate
// of 22.729 MHz (22,729,000 cycles per second)
// to a signaling rate of 2400 (BAUD).
// We usually have much faster signalling
// rates nowadays, but this demonstrates what the
// divisor actually does.
// The formula given in the NS16500A specification
// for calculating the divisor is:
// divisor = ceil( (clock_hz) / (baud_sps x 16) )
// So, we substitute our values and get:
// divisor = ceil( 22_729_000 / (2400 x 16) )
// divisor = ceil( 22_729_000 / 38_400 )
// divisor = ceil( 591.901 ) = 592

// The divisor register is two bytes (16 bits), so we
// need to split the value 592 into two bytes.
// Typically, we would calculate this based on
// measuring the clock rate, but again, for our
// purposes [qemu], this doesn't really do anything.
let divisor: u16 = 592;
let divisor_least: u8 = divisor & 0xff;
let divisor_most: u8 = divisor >> 8;

// Notice that the divisor register DLL (divisor
// latch least) and DLM (divisor latch most) have the
// same base address as the receiver/transmitter and
// the interrupt enable register. To change what the
// base address points to, we open the "divisor
// latch" by writing 1 into the Divisor Latch Access
// Bit (DLAB), which is bit index 7 of the Line
// Control Register (LCR) which is at
// base_address + 3.
ptr.add(3).write_volatile(lcr | 1 << 7);

```

```

        // Now, base addresses 0 and 1 point to DLL and DLM,
        // respectively.
        // Put the lower 8 bits of the divisor into DLL
        ptr.add(0).write_volatile(divisor_least);
        ptr.add(1).write_volatile(divisor_most);

        // Now that we've written the divisor, we never have
        // to touch this again. In hardware, this will divide
        // the global clock (22.729 MHz) into one suitable
        // for 2,400 signals per second. So, to once again
        // get access to the RBR/THR/IER registers, we need
        // to close the DLAB bit by clearing it to 0.
        ptr.add(3).write_volatile(lcr);
    }
}

```

首先，你可能会注意到，函数本身不再是不安全 (unsafe) 的了。相反，函数主体的各个部分是不安全的。当我们写一个不安全块时，我们是在告诉 Rust “拿着我的啤酒 (我要做些危险的事了)”。如果函数被标记为不安全，比如 `unsafe fn uart_init`，那么函数主体中的所有代码都可以是安全或不安全的。当我们给单个不安全块贴上标签时，只有在我们无法避免的情况下，我们才会进入不安全的 Rust。

A.2.13 读取 UART

现在我们已经初始化了，现在只要有字符出现，我们就可以从 UART 读取。在之后的部分中这种读取将由一个中断触发，我们可以读取数值并将其存入 RAM 中的一个缓冲区。

```

fn uart_get(base_addr: usize) -> Option {
    let ptr = base_addr as *mut u8;
    unsafe {
        // Bit index #5 is the Line Control Register.
        if ptr.add(5).read_volatile() & 1 == 0 {
            // The DR bit is 0, meaning no data
            None
        }
        else {
            // The DR bit is 1, meaning data!
            Some(ptr.add(0).read_volatile())
        }
    }
}

```

A.2.14 写入 UART

同样地，我们也希望能够写入控制台。因此，让我们编写 `uart_write`。

```
fn uart_put(base_addr: usize, c: u8) {
    let ptr = base_addr as *mut u8;
    unsafe {
        // If we get here, the transmitter is empty, so
        // transmit our stuff!
        ptr.add(0).write_volatile(c);
    }
}
```

A.2.15 加入到 Rust 主程序

现在我们已经写好了 `uart` 函数，让我们创建一个 `uart` 模块。首先创建一个名为 `src/uart.rs` 的文件。然后将 `uart_init`、`uart_get` 和 `uart_put` 函数移入其中。我们将使用名为 Rust “trait” 的东西，这样我们就可以将我们的 UART 与任何想要写的东西联系起来……主要是 `>write!` 宏。

与 C++ 不同，Rust 的唯一结构性容器是 `struct`。Rust 的 `struct` 可以有方法和成员变量，就像 C++ 一样。所以，我们要把 `Uart` 驱动包装成一个名为 `Uart` 的结构。

```
pub struct Uart {
    base_address: usize,
}
```

我们正在存储 UART 的基地址。有一些平台有多个 UART! 然而，现在我们要把它与 `core::fmt::Write` trait 联系起来。这使我们可以使用已经内置在 Rust 中的宏 `write!`。要实现一个 trait，我们需要使用 `impl Write for Uart` 语法，同时确保我们使用 `core::fmt::Write`，以便将 trait 导入我们的 `uart.rs` 文件中。这个 trait 能确保某个（或某些）函数会存在。例如，下面的 `write_str` 是由 trait `Write` 要求的。这些行为非常像 Java 的 `interfaces` 或 C++ 的 `abstract` 类。

```
// Rust allows us to import multiple traits or structures or
// other namespaces in one line by using the braces { } as
// I've done here:
use core::fmt::{Error, Write};

// This is the memory load of the structure. Unlike C++,
// we don't define the member functions here. Instead, we'll
// use an impl block (implements or implementation).
pub struct Uart {
    base_address: usize,
}

// Here's the implementation block. Notice that impl Uart
// simply implements member functions in our already defined
```

```
// structure.
impl Uart {
    pub fn new(base_address: usize) -> Self {
        Uart {
            // Since our parameter is also named the same as
            // the member variable, we can just label it by
            // name.
            base_address
        }
    }
}

// This is a slightly different syntax. Write is this
// "trait", meaning it is much like an interface where we're
// just guaranteeing a certain function signature. In the
// Write trait, one is absolutely required to be implemented,
// which is write_str. There are other functions, but they
// all rely on write_str(), so their default implementation
// is OK for now.
impl Write for Uart {
    // The trait Write expects us to write the function
    // write_str which looks like:
    fn write_str(&mut self, s: &str) -> Result<(), Error> {
        for c in s.bytes() {
            self.put(c);
        }
        // Return that we succeeded.
        Ok(())
    }
}
```

A.2.16 我们来写 println!

println!() 通常与 stdout 对接, 但由于我们在写操作系统, 我们没有这些花哨的功能。所以, 我们要自己写。我们将用 Rust 的元编程语言来写, 以代码 macro_rules 开始。这种编程风格是我的第一个难点, 很难找到一个好的方法来向本科生讲清楚这种元编程的复杂性, 它本身就很像一种语言。

为了确保我们有一个全局宏, 我把 println 写在 lib.rs 中, 我把它称为”主”文件。也就是说, 我们为操作系统编写的所有模块, 如 UART、调度、进程, 都将是通过 lib.rs 文件连接的子模块。全局宏放在 lib.rs 的顶部, 用以下代码启动:

```
// //////////////////////////////////////
// / RUST MACROS
// //////////////////////////////////////
#[macro_export]
macro_rules! print
{
    ($($args:tt)+) => ({
```



```

        use core::fmt::Write;
        let _ = write!(crate::uart::UartDriver::new(
            0x1000_0000), $($args)+);
    });
}
#[macro_export]
macro_rules! println
{
    () => ({
        print!("\r\n")
    });
    ($fmt:expr) => ({
        print!(concat!($fmt, "\r\n"))
    });
    ($fmt:expr, $($args:tt)+) => ({
        print!(concat!($fmt, "\r\n"), $($args)+)
    });
}

```

在这段代码中，我同时指定了 `print` 和 `println`，当我们使用它们的时候，它们将逐渐拥有自己的感叹号。`println!` 所做的就是在我们要打印的东西的末尾添加一个换行。

那么，让我们来看看这个语法。我们通过指定 `#[macro_export]` 将我们的宏导出到其他的子模块中。这是位于阳面的 `#[macro_use]` 指令的阴面，它告诉 Rust，我们将使用一个 `crate`（库）的宏。

现在，我们进入”手臂 (arms)”，也就是等号-大于号，在 C# 等语言中也被称为”胖箭头”。但是正如我自己也在和腰带的尺寸作斗争，我将坚持使用”手臂”这一称呼。在这里，我们要指定一个匹配模式。每当我们写下 `println!(“Hello”)` 这样的东西时，Rust 就会尝试匹配我们指定的参数。所以，我使用美元符号”\$”来指定一个元变量。在这里，我们告诉 Rust，我们要把我们的参数标记为”令牌树 (token tree)” (tt) 参数，这就是 `$args:tt` 的来源。加号’+’告诉 Rust 这里可能有一个或多个匹配，所以要想编译，必须至少指定这些标记树参数中的一个，否则会产生错误。

然后我们来到了手臂’=>’。在这里，我们要告诉 Rust，如果我们匹配了这个臂膀，需要”编程执行”什么。在这里，注意我通过简单地添加大括号创建了一个代码域。由于我们的 UART 驱动实现了 `Write trait`，我们必须告诉 Rust 在调用”`write!`”时要”使用”该 trait。幸运的是，`write!` 的功能已经为我们写好了！（感谢 Rust 开发者！）

Rust 尽力成为一个好管家，告诉我们 `write!` 会返回一些东西。我们用 `let _` 来

告诉 Rust，我们知道这一点，但我们并不关心。下划线‘_’本质上是承认了返回，但还是要抛弃它的值。在 `write!` 里面，我指定了一个长的路径来获取我们的 `uart` 驱动 `crate::uart::UartDriver`。基本上，`crate` 指定了根目录，从那里我们添加了一个名为 `uart` 的模块（见下面的代码），在该模块中，我们有一个名为 `UartDriver` 的结构，它实现了一个名为 `new` 的静态函数。

最后，我们要告诉 Rust 将每个参数扩展到 `write!` 宏的末尾，这样所有的参数（记住：一个或多个）都直接传递给 `write` 宏。

对于 `println`，我使用了 `expr`，意思是“表达式”，以表明这个只是上一个的更严格版本。这篇博客并不打算成为一个编译器教程，所以我打算把它作为“未来”的工作来做——欢迎来到学术界！而现在我只想告诉你，Rust 会努力地匹配我们给 `println` 和 `print` 编写的内容。

A.2.17 为什么 `println` 有 3 个手臂？

与 `print` 不同，我们希望能够调用 `println!()`；来只打印一个换行符。如果我们对 `print!` 这样做，那就太愚蠢了，因为它什么都不会做。这就是为什么在 `print` 中，我们指定了加号‘+’，这样我们就可以匹配一个或多个参数。为了匹配零个或多个，我们会使用星号‘*’。

第二个手臂是 `$fmt:expr`，如果我们提供至少一个参数，Rust 就会匹配。在本例中，这是一个格式化字符串—类似于“Hello ”。请注意，Rust 使用了许多 C# 的风格（或者 C# 使用了 Rust 的风格—你们自己争论吧！），而不是 C 风格的 `%d`、`%s` 等。关于格式化的整个讨论可以在这里找到：<https://doc.rust-lang.org/std/fmt/index.html>。

第三个手臂将格式化字符串和需要被填充到格式化字符串中的参数结合起来。

A.2.18 为什么每次都要创建一个新的 `UartDriver`？

这篇文章已经变得相当长了。我们将使用“单例”模式，这样外面就只有一个 `UartDriver` 存在。然而，由于内存位置不会变化，我在这里的方法是简单地将该内存地址包裹到一个 `UartDriver` 结构中。剩下太长了，不再看了，总之我们会成功的！

A.2.19 将 UART 加载到 `lib.rs` 中

我使用我的 `lib.rs` 作为我的“守护者（keeper）”。它包含了一些代码，但它的工作是导入其他模块，并给我们一个路径来获取它们，就像我们在 `print` 宏中

写的 `crate::uart::UartDriver` 一样。Rust 不使用 `include` 预处理指令，比如 C++ 的 `#include`，而是让我们通过指定 `mod uart` 来导入模块。我在我的模块前面加了 `pub`，使它们成为公共模块。这很像 Rust 将所有变量默认为不可变的。可见性也是如此，利用 `pub` 关键字使一个本来是私有的模块变成公共的。

所以，现在我们的代码看起来是这样的：

```
#![no_std]
#![feature(panic_info_message,asm)]

// //////////////////////////////////////
// / RUST MACROS
// //////////////////////////////////////
#[macro_export]
macro_rules! print
{
    ($($args:tt)+) => ({
        use core::fmt::Write;
        let _ = write!(crate::uart::UartDriver::new(
            0x1000_0000), $($args)+);
    });
}
#[macro_export]
macro_rules! println
{
    () => ({
        print!("\r\n")
    });
    ($fmt:expr) => ({
        print!(concat!($fmt, "\r\n"))
    });
    ($fmt:expr, $($args:tt)+) => ({
        print!(concat!($fmt, "\r\n"), $($args)+)
    });
}

// //////////////////////////////////////
// / LANGUAGE STRUCTURES / FUNCTIONS
// //////////////////////////////////////
#[no_mangle]
extern "C" fn eh_personality() {}
#[panic_handler]
fn panic(info: &core::panic::PanicInfo) -> ! {
    print!("Aborting: ");
    if let Some(p) = info.location() {
        println!(
            "line {}, file {}: {}",
            p.line(),
            p.file(),
            info.message().unwrap()
        );
    }
}
```

```

        );
    }
    else {
        println!("no information available.");
    }
    abort();
}
#[no_mangle]
extern "C"
fn abort() -> ! {
    loop {
        unsafe {
            asm!("wfi":::"volatile");
        }
    }
}

// //////////////////////////////////////
// / ENTRY POINT
// //////////////////////////////////////
#[no_mangle]
extern "C"
fn kmain() {
    // I don't need to add crate in front here because we're
    // in the root module (lib.rs).
    // However, we had to add crate:: for the print! macro
    // since that metaprogrammed macro will expand into
    // other, non root modules, where the crate:: will be
    // necessary.
    // Remember, the default base address is 0x1000_0000.

    let mut my_uart = uart::UartDriver::new(0x1000_0000);
    my_uart.init();

    println!("This is my operating system!");
    println!("I'm so awesome. If you start typing something,
    I'll show you what you typed!");

    // Our goal is to repeat, or echo, what the user types to
    // us. This is just for testing!
}

// //////////////////////////////////////
// / RUST MODULES
// //////////////////////////////////////

pub mod uart;

```

我继续下一步，并预设了 `println!()`（去 <在此插入你的小组名称>），期望一切都能顺利。在我的人生中总会有那么一刻，当我开始 `make` 时，我屏住呼吸，只

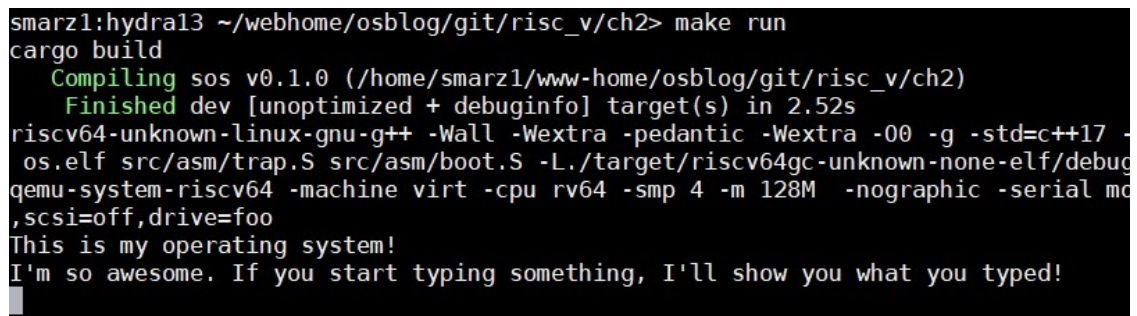
等着一连串的错误来告诉我我是一个多么糟糕的程序员。

有了这部分之后，`pub mod uart` 将在与 `lib.rs` 相同的目录下寻找一个叫 `uart.rs` 的文件。稍后，我们将创建目录并使用 `mod.rs`。如果你知道我在说什么，我们很快就能达到目的了！如果你不知道，别担心，我们以后总会到终点的。（是的，我说了两件相互矛盾的事情）。

由于这种导入是 Rust 的行为，而且我们写了 `uart.rs`，我们应该保持着积极的态度。

A.2.20 说出来，我的小家伙

让我们看看是否能让我们的孩子（操作系统）对我们说话。假设我们正确地写了 `write_str`，而且我们没有忘记先初始化 UART，我们应该看到一些显示。我一开始忘了，但我可以编辑我的帖子—谁能发现呢，对吧？



```
smarz1:hydra13 ~/webhome/osblog/git/risc_v/ch2> make run
cargo build
   Compiling sos v0.1.0 (/home/smarz1/www-home/osblog/git/risc_v/ch2)
   Finished dev [unoptimized + debuginfo] target(s) in 2.52s
riscv64-unknown-linux-gnu-g++ -Wall -Wextra -pedantic -Wextra -O0 -g -std=c++17 -
os.elf src/asm/trap.S src/asm/boot.S -L./target/riscv64gc-unknown-none-elf/debug
qemu-system-riscv64 -machine virt -cpu rv64 -smp 4 -m 128M -nographic -serial mo
,scsi=off,drive=foo
This is my operating system!
I'm so awesome. If you start typing something, I'll show you what you typed!
```

图 A.2 最初的显示

所以，运行 `make run`，让小家伙说话吧！如果你什么都得不到，请看一下我的第二章的 Git 仓库，网址是：<https://github.com/sgmarz/osblog>

A.2.21 认真听，我的小家伙

现在我们知道我们的孩子可以说话，让我们看看它是否可以听见。请记住，`write_str` 作为 `Write trait` 的一部分，调用我们的 `uart_put` 函数来逐个字符（严格地说，逐个 `u8`）填入。自从 UTF 以来，字符（`character`）已经越来越没落了。

我们最终会把这段代码移到带有缓冲区的控制台处理程序中，这样我们就可以获取整个字符串，而不仅仅是字符，但这里暂时还没这么高级：

```
let mut my_uart = uart::UartDriver::new(0x1000_0000);
my_uart.init();

println!("This is my operating system!");
println!("I'm so awesome. If you start typing something,
I'll show you what you typed!");
```

```
// Now see if we can read stuff:
// Usually we can use #[test] modules in Rust, but it would
// convolute the task at hand. So, we'll just add testing
// snippets.
loop {
    if let Some(c) = my_uart.get() {
        match c {
            8 => {
                // This is a backspace, so we essentially
                // have to write a space and backup again:
                print!("{}", 8 as char, ' ', 8 as char);
            },
            10 | 13 => {
                // Newline or carriage-return
                println!();
            },
            _ => {
                print!("{}", c as char);
            }
        }
    }
}
```

Rust 有一个循环 (loop)，它无条件地以最快的速度进行循环。我们应该加一些 break，但现在让我们以最快的速度读取字符。大多数非 Rust 程序员首先会看到的新语法是 `if let Some(c)`。记住 `my_uart.get()` 返回一个 Option，它要么是 Some，要么是 None。

使用 `if let`，我们可以从 option 中获取准确值，并同时查看 Option 是否为 Some。在上面的例子中，`if let` 只在 `my_uart.get()` 返回 Some 时执行其主体。请记住，我们还将从 UART 中取出的字符包装到这个 `Some()` 结构中（确切地说，是枚举 (enumerate)）。因此，这个块里面的代码意味着一个叫做 `c` 的新变量获得了被 Some 包装的字符。很好，对吗？

第二个语言结构是匹配 (match) 语句。它类似于 switch 语句，但没有许多不清楚的 break。在这种情况下，Rust 将检查变量 `c` 的内容，然后我们指定某些情况来检查 `c` 是否与之匹配。在上面的代码中，我们设定当 `c` 匹配 8 和 10 或 13 时会执行特定的代码。`|`（管道字符）允许我们为一个手臂指定多个情况。同样，就像 `print` 和 `println` 中的宏匹配一样，我们使用胖箭头来扩展手臂。

Case 8 是退格的数字代码。然而，退格的真正含义是”将光标向左移动一个字符”。所以，如果我们只是单纯的 `print`，光标会移动，但那一格的文本仍然在那里。因此，我们的解决方案是移动光标，画一个空格（这将使光标向右移动一

个空格)，然后再将光标向左移动一个空格。

Case 10 是 `\n`（换行）字符，Case 13 是 `\r`（回车）字符。根据你的终端和终端仿真器，你可能只有一个或另一个或两者都有。我们所做的就是调用 `println!()`，这将使 Rust 匹配第一个臂（没有任何表达式或 token 树的那个）。回顾那里的代码，我们看到这将只是打印一个换行。

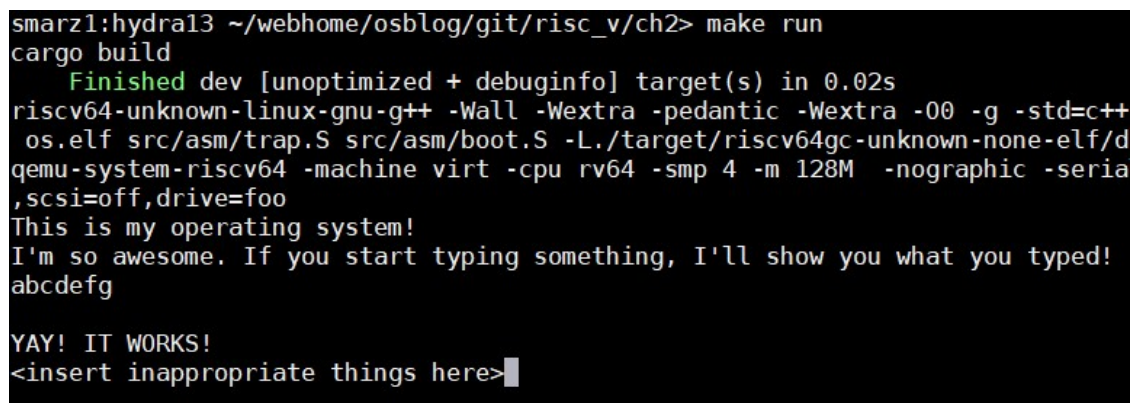
```
// This is println!s arm if no parameters are specified.  
( ) => ({  
    print!("\r\n")  
});
```

最后，下划线被用来包括其他所有的东西—有点像“非上述”选项。它很像 `switch` 语句的默认 (default) 情况。任何我们不能用上面的手臂匹配的东西都可以在这里匹配。Rust 强迫我们为每一种可能性都准备一个 `case`，因此，我们要么指定 256 种可能性 (0-255)，要么指定几种可能性，然后再指定一个万能的。我选择了后者。

附注：如果只有一行代码，Rust 并不要求我们为每个匹配臂添加括号。然而，由于我们将在这段代码的基础上进行开发，我为“未来的扩展”加上了大括号。

A.2.22 测试!

让我们看看这是否有效。



```
smarz1:hydra13 ~/webhome/osblog/git/risc_v/ch2> make run  
cargo build  
Finished dev [unoptimized + debuginfo] target(s) in 0.02s  
riscv64-unknown-linux-gnu-g++ -Wall -Wextra -pedantic -Wextra -O0 -g -std=c++  
os.elf src/asm/trap.S src/asm/boot.S -L./target/riscv64gc-unknown-none-elf/d  
qemu-system-riscv64 -machine virt -cpu rv64 -smp 4 -m 128M -nographic -seria  
,scsi=off,drive=foo  
This is my operating system!  
I'm so awesome. If you start typing something, I'll show you what you typed!  
abcdefg  
  
YAY! IT WORKS!  
<insert inappropriate things here>█
```

图 A.3 展示用户输入

A.2.23 最后的控制台操作

许多人不记得 ANSI 转义序列了，因为我们现在有 GUI。然而，每当你按下扩展键，如方向键时，这些将被传递给你的 UART 系统。现在，方向键只是在屏幕上移动光标，因为我们只是打印出准确的可见字符。

ANSI 转义序列是多个字节，以字节 0x1b（十进制 27）开始。下一个字节是 0x5b（十进制 91），是左括号字符 '['。然后剩下的是序列的参数。我们要捕捉上、下、左、右箭头的序列。所以，在你的匹配手臂上添加以下内容。

```
0x1b => {
  // Those familiar with ANSI escape sequences
  // knows that this is one of them. The next
  // thing we should get is the left bracket [
  // These are multi-byte sequences, so we can take
  // a chance and get from UART ourselves.
  // Later, we'll button this up.
  if Some(next_byte) = my_uart.get() {
    if next_byte == 91 {
      // This is a right bracket! We're on our way!
      if let Some(b) = my_uart.get() {
        match b as char {
          'A' => {
            println!("That's the up arrow!");
          },
          'B' => {
            println!("That's the down arrow!");
          },
          'C' => {
            println!("That's the right arrow!");
          },
          'D' => {
            println!("That's the left arrow!");
          },
          _ => {
            println!(
              "That's something else.....");
          }
        }
      }
    }
  }
},
```

现在，再次运行你的操作系统，但这一次试试方向箭头！这将有助于我们编写我们的 shell。

如果你打箭头太快，有时你会看到 A、B、C 或 D 出现。这是因为你会匹配转义序列，但错过了字节 91。因此，你的循环看起来只是一个普通的 A、B、C 或 D，甚至是左括号 '['。


```

smarz1:hydra13 ~/webhome/osblog/git/risc_v/ch2> make run
cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
riscv64-unknown-linux-gnu-g++ -Wall -Wextra -pedantic -Wextra -O0 -g -std=c++11
os.elf src/asm/trap.S src/asm/boot.S -L./target/riscv64gc-unknown-none-elf/de
qemu-system-riscv64 -machine virt -cpu rv64 -smp 4 -m 128M -nographic -serial
,scsi=off,drive=foo
This is my operating system!
I'm so awesome. If you start typing something, I'll show you what you typed!
That's the up arrow!
That's the down arrow!
That's the left arrow!
That's the right arrow!

```

图 A.4 响应箭头的输入

```

This is my operating system!
I'm so awesome. If you start typing something, I'll show you what you typed!
abcThat's the up arrow!
That's the down arrow!
That's the left arrow!
That's the right arrow!
That's the left arrow!
That's the right arrow!
That's the up arrow!
That's the left arrow!
That's the right arrow!
That's the left arrow!
ACThat's the up arrow!
That's the left arrow!
That's the right arrow!
DThat's the right arrow!
That's the up arrow!
DCThat's the left arrow!
That's the right arrow!
That's the up arrow!
DThat's the right arrow!
That's the up arrow!
That's the left arrow!
CThat's the up arrow!
That's the left arrow!
[CThat's the up arrow!
That's the left arrow!

```

图 A.5 输入过快的结果

A.2.24 哇，这个代码好 dirty

确实。你见过房子的施工吗？你现在看到的是我们在操作系统中铺设的脚手架。这种简单但有点肮脏的通信方式使我们能够聚合我们即将为操作系统的下一个部分建立的代码。这样做的好处是我们可以用很少的代码进行通信。当我们把更多的 Rust 语言结构放在 UART 驱动周围时，每个操作可能看起来就不那么明显了。

现在，我们只是想进行通信。我们从牙牙学语开始，然后可以开始将 UART 集成到一个控制台模块中，该模块将支持更多类似终端的功能。记住，让它能够

顺利地工作、玩耍、欢呼、畅饮（beer 是一个动词，对吗？），然后我们就可以开始重构代码了。我预计还有很多章节会是这样的。我们将探索已经被广泛研究的内存分配器，挑选一个版本，看看它是否适合我们，如果不适合，再挑选另一个。我认为这更像是一段旅程，而不是单纯的“这是最后成果，享受吧！”这种只重视结果的工作。已经有一个完成的作品了—如果这是你唯一的目标的话，那就是<https://fedoraproject.org/wiki/Architectures/RISC-V>。

A.2.25 通信是关键

使用 debug 程序是一个很大的挑战，尤其是当你不熟悉如何使用一般的调试器甚至某个特殊的调试器时。相反，我教导我的那些没有编程经验的学生每次都多打印出一些值，看看是否能显示出他们认为应该得到的东西。你现在可以与你的操作系统进行听说交流。你可以开始自己做些小实验，看看你能让它做什么很酷的事情，同时研究一下 Rust 的使用。我并不是一个专家，正如许多 Rust 开发者在我的 GitHub 上所证明的那样—尽管我很欢迎指导（我真的很欢迎！）。所以，也许你能找到比我的写法更整洁、更好的方法来编写 Rust。如果是这样，请告诉我们！