国科大操作系统研讨课任务书

RISC-V 版本

版本 *1.0*

目录

Project 2
简易内核实现

1 实验说明

在之前的实验里，我们了解了操作系统的引导过程，并且自己亲手制作了操作系统的镜像，并最终能将其从开发板上启动起来，但是我们的“操作系统”只能打印出一行“Hello OS!”，不具有操作系统应该有的任何功能。因此，在这一节，我们将对我们现有的操作系统进行加工润色，使其具备**任务调度**和**锁**功能。

通过本次实验，你将学习操作系统进程调度、例外处理等知识，掌握进程的阻塞和唤醒、锁的实现。调度和例外处理是操作系统非常重要的部分，也是实现一个完整操作系统首先需要考虑的问题，本次实验涉及到的知识点比较多，很多地方可能比较难于理解，因此除了任务书中提及的内容，希望同学们可以多查阅相关资料，充分了解 RISC-V 架构的相关知识。本次实验的内容如下：

任务一 了解操作系统中进程的管理和系统调用，实现进程控制块、进程切换、非抢占式调度。

任务二 了解操作系统中进程的各种状态以及转化方式，实现进程的阻塞、进程的唤醒、互斥锁。

任务三 了解操作系统中用户态和内核态的基础交互方式系统调用，并实现一些带内核态隔离的系统调用处理。

任务四 了解操作系统中时钟中断的触发和处理流程，实现一个时钟中断，并在其基础上实现进程的抢占式调度。

任务五 实现 fork 系统调用，使用户进程可以自己创建多个子进程，并具备不同的调度优先级，分别执行不同的程序内容。

从本次实验开始，我们将提供给大家一些基础的代码框架，包括了稍微复杂一些的文件结构和各个功能对应的文件。框架里会给出一些基础的库函数 (比如打印函数)，以及一些实现流程的指导。当然我们不希望实验课的内容成为“填空”，而是希望每个同学能真正的去实现一个属于自己的操作系统。因此框架里除了库函数大部分内容是空缺的，同学们需要自己去实现操作系统的全部内容。换言之，我们给出的框架结合任务书更具备一些指导意义。

我们更希望的是同学们能够通过任务书、基础的代码框架去知道如何去实现一个操作系统，最终自己去按照自己的想法去实现，而不是被禁锢到我们给大家提供的框架内。

因此，如果有同学可以自己“开山立派”，完全自己实现一个内核，我们也是非常鼓励的，同学们可以脱离、修改我们的框架。

最后再次强调的是，本章是操作系统实验课中最为重要且有一定难度的一部分，通过本次实验，你的内核将“初具雏形”，为后续的进程通信、内存管理、文件系统等模块的实现打下坚实基础。希望同学们可以耐下性子，认真学习。

2 本章解读

这一部分的要点是：

1. 掌握进程管理的基本原理
2. 理解并实现锁
3. 理解中断处理和进程调度的基本概念

这一章完成的是一个操作系统最根本的功能：调度进程。实际上，最早的操作系统就是起源与此。早期还在用纸带机的年代，每个程序就是一卷纸带。人们最开始只能读一个程序到计算机，然后执行完，再执行下一个程序。这样做，程序在读取纸带或者其他设备的时间就被浪费了。处理器空等耗时的外部设备的运行。为了改变这一状态，人们写了一个简单的管理程序。管理程序把从纸带上读到的程序先放在内存里，然后在一个程序等待耗时的外部设备的操作时，换自动换上另一个程序执行。这样处理效率就得到了大幅提升。

本章的内容正是进程调度，及其衍生而来的各种概念和算法。完成了这些，也就完成了最最基本的一个操作系统的功能。

3 进程和系统调用

大家都知道，进程是操作系统的资源分配单位，而线程是操作系统的基本调度单位。二者比较大的区别在于对虚存的管理，但目前而言我们不涉及虚存的相关内容，并且由于接下来我们的主要工作重心在进程的调度方面，不涉及线程的概念。在进入到 kernel 的 start 函数后，我们打印出了“Hello OS!”，我们可以认为此时操作系统已经拥有了一个内核进程。但是想开启另一个新的进程，我们需要进行 PCB 初始化、任务切换这两步后，才可以开始运行一个新的进程。而在程序中，如果程序希望使用操作系统提供的一些功能，就可以用到系统调用，这一概念使得操作系统的某些功能被封装成一个用户可见的接口，使操作系统可以更好的管理和隔离硬件资源，并且使应用程序的开发具有更好的兼容性。我们在这个 Project 中也会要求大家实现各种系统调用，从而使同学们的操作系统具备各种功能。

3.1 进程控制块

为了描述和控制进程的运行，操作系统需要为每个进程定义一个数据结构去描述一个进程，这就是我们所说的进程控制块 (Process Control Block)，简称 PCB。它是进程重

要的组成部分，它记录了操作系统用于描述进程的当前状态和控制进程的全部信息，比如：进程号、进程状态、发生任务切换时保存的现场 (通用寄存器的值)、栈地址空间等信息。操作系统就是根据进程的 PCB 来感知进程的存在，并依此对进程进行管理和控制，PCB 是进程存在的唯一标识。在本次实验中，同学们需要自己思考 PCB 应存储的信息，实现 PCB 数据的初始化。

当然，在本次实验中，尚不太涉及到进程和线程的区别。大家可以先把 pcb 简单地当作一个只会拥有一个线程的进程的控制块。

3.2 进程所需资源的管理

在3.1节中我们提到过，进程本质上是资源单位。进程的运行是需要资源的。那么，具体需要哪些资源呢？最基本的，运行需要内存和处理器。有了内存和处理器这两种资源，程序才可能运行。

在前面的实验中，我们提到过，C 程序的运行需要准备好运行所需的栈空间。在这里也是一样，我们需要为每一个需要运行的进程分配其所需要的栈空间，并为其分配处理器的执行时间。分配处理器时间我们可以通过调度器来做。那么分配栈空间怎么办呢？每来一个进程，我们就需要为其分配内存空间供其运行。显然，我们需要建立一个管理并分配空闲内存空间的方式，来管理我们手头的空闲内存。

我们建议的地址空间划分如所表P2-1所示，从 0x50500000 开始都是空闲的。可以供大家任意使用。

地址范围	建议用途
0x50000000-0x50200000	BBL 代码及其运行所需的内存
0x50200000-0x50500000	Kernel 的数据段/代码段等
0x50500000-0x60000000	供内核动态分配使用的内存

表 P2-1: 地址空间用途划分

那么，管理内存有哪些方案呢？

最简化方案（推荐） 本次实验中，进程一旦创建就不会退出，也不会中途创建进程。进程所需的栈空间一般设置为 4K 或者 8K 的固定大小。所以可以简单的只实现一个 alloc 函数，每调用一次从可用的内存中，分配一段给进程用。不考虑回收。待后续实验复杂后再改。

固定内存分配 由于我们的进程很小，一般只需要分配一页给每个进程作栈空间就可以。所以可以用一个 freeList 记录空闲的内存块。每个内存块只需要 4K 的固定大小就可以。每次分配时从 freeList 中分配一块，回收时从 freeList 中回收一块。

伙伴内存分配 伙伴内存分配技术 [1] 的分配方式为：每次分配时，递归地把一块大的内存分成两半，直到内存块的大小比较适合请求的大小为止。回收内存时，如果相邻两块内存都被释放了，就拼回一块大内存。具体实现可以参考 [2] 所述内容。

SLAB 内存分配 SLAB 是一种相对复杂的分配策略，具体可以参考 [3]。

在初始化 PCB 时，你需要自己设计分配算法，并通过你的内存分配算法，为每个进程分配栈空间。

3.3 内核栈的设计

系统中的栈如何设计是一个古老的话题。C 函数的运行是需要栈的。当然，我们在上一节中已经讨论了内存空间怎么分配的问题。但是对于内核而言，还有另外一个问题：内核需要单独的栈。这是因为，内核其实相当于整个系统的管理者和服务者。它负责管理各个资源，并为所有的程序提供一些公共的服务。这就意味着，内核必须有比普通进程更高的权限。为了避免各类安全性问题，内核也需要有自己独立的运行空间。

内核栈的设计常见的有两种方案 [4]：单内核栈和多内核栈。如图P2-1所示，单内核栈设计所有的进程在进入到内核时，共享同一个内核栈。多内核栈设计则每一个进程都有一个独立的内核栈。

单内核栈的缺点是很显然的：内核必须一次性执行完，中间不能中断。

多内核栈设计则无此缺点。每个系统调用执行的过程中，可以随时被抢占，轮换到另一个进程执行。在我们的实验中，虽然内核态执行不会被抢占，但我们推荐采用的是**多内核栈设计**（start-code 也是按照多内核栈设计给出的框架）。

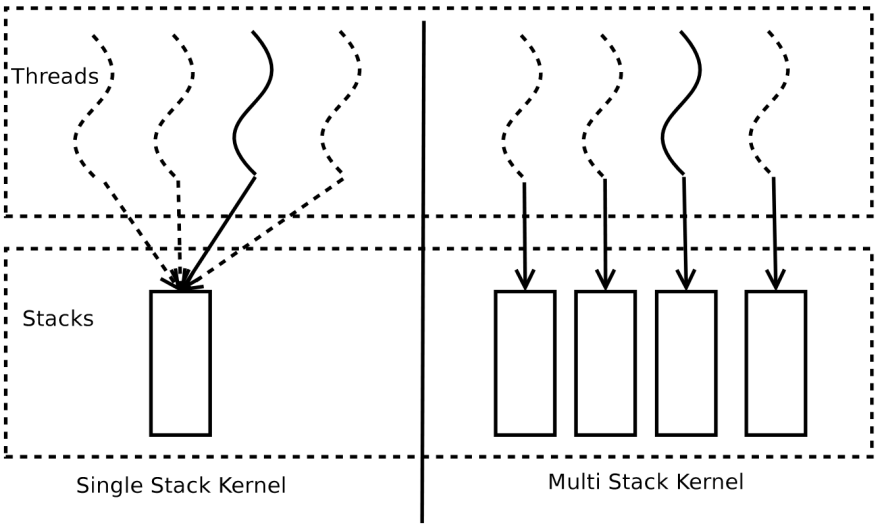

图 P2-1: 两种内核栈设计 [4]

3.4 任务的切换

拥有了 PCB 之后，我们就可以去管理进程从而去实现进程的切换了。当进程发生切换的时候，操作系统就会将当前正在运行进程的现场（寄存器的值）保存到 PCB 中，然后从其他进程的 PCB 中选择一个，对这个 PCB 里的保存的现场进行恢复，从而实现跳到下一个进程的操作。这个选择下一个将要运行的进程的切换过程也就是我们平常所说的**调度**。

任务切换的过程是本实验的难点，不仅概念上不好理解，也有很多细节问题容易出错。不过简单的来理解任务切换，我们实际上只需要一个全局的 PCB 指针 current_running，它指向哪个 PCB，说明那个 PCB 对应的任务为正在运行的任务。在进行保存和恢复的时候只对这个 current_running 对应 PCB 进行保存和恢复。具体流程如图P2-2所示。

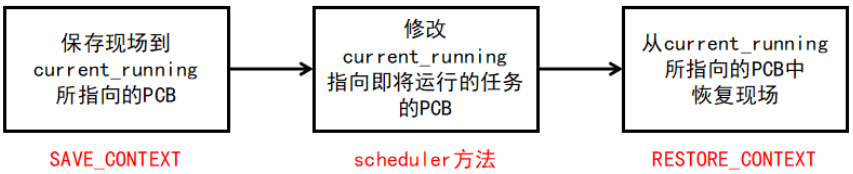

图 P2-2: current_running 指针和任务切换的关系

调度过程的触发方式大致可分为两种：第一种是在不具备中断处理能力时，通过进程自己使用调度方法去“主动”的交出控制权的非抢占式调度；第二种是在具备了中断处理能力后，通过周期性触发时钟中断去触发调度方法，从而使得进程“被迫”交出控制权的抢占式调度。非抢占式调度只需要设计好 PCB、实现进程的现场保存和现场恢复、实现调度函数即可，因此我们将在任务一中首先实现这种方式。抢占式调度涉及到时钟中断处理等操作，我们在 Project2 后面的任务中进行实现。无论哪种调度方式，其核心的调度算法可以是一样的。

那么问题是，切换到底该怎么实现，需要做哪些事情呢？在我们的 start_code 中，我们准备了一个汇编函数框架 switch_to。这里把这个函数单拿出来介绍的原因是，在从进程 A 切换到进程 B 的过程中，进程 A 调用了 switch_to 这个函数，而这个函数返回时，已经是进程 B 在运行了。当然同学们可以不用 switch_to 这个函数名，也可以有其他的设计。但无论如何，一定有一个函数具有这样的特性：在进入和离开这个函数的时刻，正在运行的进程是不同的。而这一动作可以分成两个部分来理解：一是将进程 A 的执行中断，二是将进程 B 的执行恢复。既然进程 A 将来还是要继续执行的，那么在它的执行被中断的时候，一定要保存它的执行现场，用硬件的话来说，就是要保存执行所需要的寄存器的值。至于保存的位置，则可以选择保存到这个进程专属的栈空间或者 PCB 里面。相对应的，有了上次保存的现场，恢复进程 B 的现场也就很简单了，从上次保存的位置把寄存器的值恢复即可。

这里稍微多说几句：什么是执行所需要的寄存器呢？这个问题需要同学们自己去思考。

3.5 PCB 初始化

对于 PCB 的初始化，我们需要给予一个进程 id 号（pid）、状态（status）等，此外，为了能让我们的进程运行起来，我们要将入口地址在初始化的时候保存到 PCB 中，然后在该进程第一次运行时（从上个进程切换到这个进程时），跳转到这个地址，开启一个进程的运行。看到这里，聪明的你可能已经发现一个问题了：在任务切换时，我们假定所有的任务切换都是从 switch_to 的地方被保存的。然而，当我们初始化 PCB，并且

试图切换到新初始化出来的 PCB 时，会发现根本没有办法恢复现场。因为这是新初始化出来的 PCB, 根本没有可以恢复的东西。那怎么办呢？这个问题需要同学们自己考虑一个合适的设计。

3.6 进程调度

进程调度的算法有很多种，比较有名包括 CFS、BFS、多级反馈队列、先来先服务等等。为了简化大家的实现，这里建议大家采用最朴素的轮转调度：将所有处于 READY 状态的进程直接放入到准备队列中，每次取出队头作为 current_running。发生进程轮转时，如果 current_running 仍然是 READY 状态就再放回准备队列。

3.7 系统调用

上面讲的进程调度的功能，应该是操作系统提供给用户程序的一种功能。用户不需要关心实际的调度策略也不应该访问到内核的数据结构，而用户可以使用内核给出的访问接口去调用内核的功能。这样的访问接口我们通常称之为系统调用。

在任务 1 的阶段，由于我们还没有用隔离用户态内核态的方式去实现保护，实际编译中我们也是把测试程序和内核编译在了一起，所以我们现在的进程都可以认为是内核进程，也难以实现真正的隔离。但是我们也可以实现这样的封装，使操作系统具备系统调用的功能。例如我们会实现一个最基础的输出系统调用 printf，它在任务 1 的阶段只是简单的调用一下内核中的 printk 函数来执行，而有了保护态之后，printf 就需要先用 ecall 命令先进入内核态，然后再执行 printk 函数并把结果返回。任务 1 中还将以 sys_yield 这样的调度功能为例实现系统调用，从而使我们的操作系统可以支持多个程序分时复用一起运行。而任务 2 中我们将继续添加互斥锁的系统调用，使多个进程可以共享临界区资源。

3.8 任务 1：任务启动与非抢占式调度

实验要求

1. 设计进程相关的数据结构，如：Process Control Block，使用给出的测试代码，对 PCB 进行初始化等操作。
2. 实现任务的现场保存和现场恢复。
3. 实现非抢占式调度的系统调用 sys_yield，以及输出系统调用 printf。
4. 运行给定的测试任务，能正确输出结果，如图P2-3所示。

文件说明

见表P2-2。

编号	文件/文件夹	说明
1	arch/riscv	RISC-V 架构相关内容，主要为汇编代码以及相关宏定义 · boot/bootload.S：引导代码，请使用 P1 实现的代码 · include：头文件，包含一些宏定义，不需要修改 · kernel/entry.S：内核中需要汇编实现的部分，涉及任务切换，异常处理等内容，本次实验任务 1,2,3,4 的补全部分 · kernel/syscall.S：系统调用相关汇编实现，本次实验任不需要补全 · sbi/common.c：SBI 调用以及功能寄存器的访问，不需要修改
2	drivers 文件夹	驱动相关代码 · screen.c：打印相关函数，不需要修改 (具体介绍请看第五节内容)
3	include 文件夹	头文件
4	init 文件夹	初始化相关 · init/main.c：内核的入口，操作系统的起点
5	kernel 文件夹	内核相关文件 · irq/irq.c：中断处理相关，本次实验不需补全 · locking/lock.c：锁的实现，本次实验任务 2 补全 · sched/sched.c：任务的调度相关，一个任务的调度、挂起、唤醒等逻辑主要在这个文件夹下实现，本次实验任务 1,2 补全。部分函数下次实验补全 · sched/time.c：时间相关函数，本次实验不需补全 · syscall/syscall.c：系统调用相关实现，本次实验不需要补全
6	libs 文件夹	提供的库函数 · string.c：字符串操作函数库 · printk：打印函数库，包括用户级的 printf、内核级的 printk
7	test 文件夹	我们实验的测试任务，该文件夹下任务的正确运行是实验完成的评判标准之一
8	tools 文件夹	工具 · creatimage.c：请使用之前 P1 实现的 createimage.c
9	Makefile 文件	Makefile 文件，如果新增文件需要自行修改
10	riscv.lds 文件	链接器脚本，不需要修改

表 P2-2: P2 文件说明

图 P2-3: current_running 指针和任务切换的关系

实验步骤

1. 完成 sched.h 中 PCB 结构体设计，以及 kernel.c 中 init_pcb 的 PCB 初始化方法。

2. 实现 entry.S 中的 SAVE_CONTEXT、RESTORE_CONTEXT 宏定义，以及 switch_to 汇编函数，使其可以将当前运行进程的现场保存在 current_running 指向的 PCB 中，以及将 current_running 指向的 PCB 中的现场进行恢复。注意，请在 switch_to 时保证，current_running 同时也被保存在了 tp 寄存器中。代码的其他部分假定了 tp 和 current_running 是等价的。

3. 实现 sched.c 中 scheduler 方法，使其可以完成任务的调度切换。

4. 运行给定的测试任务（test.c 中 sched1_tasks 数组中的三个任务），其中 print_task1、print_task2 任务在屏幕上方交替的打印字符串“This task is to test scheduler”，drawing_task 任务在屏幕上画出一个飞机，并不断移动。

注意事项

1. printk 是已经在 start_code 中提供好的内核打印函数，在任务一阶段需要简单封装为 printf，功能完全相同，但是提供给测试程序使用。后面需要将 printf 改造为用户态用的系统调用。

2. 在保存现场的时候需要保存所有（32 个）通用寄存器的值，虽然都是通用寄存器，但是作用也是不同的，请同学们查阅资料了解这 32 个通用寄存器的功能，并思考如何保存和恢复现场不会破坏现场寄存器的值。

3. 除通用寄存器外，sstatus、sepc、sbadaddr、scause 都需要被保存。

4. 请为每个进程初始化好 GP 寄存器。GP 寄存器的值需要和 heads.S 一致，因为这次跑的实验都是内核进程，所以 gp 寄存器都用和内核一样的就好。

4 锁的实现

当两个进程需要对同一个数据进行访问时，如果没有锁的存在，二者同时访问，那么就会造成不可预见的问题，因为操作并不一定是原子的。可能出现第一个进程修改了一半后，第二个线程继续在第一个线程没修改完的基础上进行修改，这就可能会造成最终结果的出错。因此对访问的数据加锁，要求一次最多只能有一个线程对其访问。而被加锁的操作区域我们通常称之为临界区。对于锁的实现方法有很多，比较常见且经典的有自旋锁、互斥锁等。

4.1 互斥锁

自旋锁在进入临界区失败时需要不停的重试，因此会浪费 CPU 资源。而互斥锁的实现方法为一旦进程请求锁失败，那么该进程会自动被挂起到该锁的阻塞队列中，不会被调度器进行调度。直到占用该锁的进程释放锁之后，被阻塞的进程会被占用锁的进程主动的从阻塞队列中重新放到就绪队列，并获得锁。因此，使用互斥锁的话可以节约 CPU 资源。

4.2 任务 2：互斥锁的实现

实验要求

了解操作系统内的**任务调度机制**，学习和掌握**互斥锁**的原理。实现任务的阻塞和解除阻塞的逻辑，实现一个互斥锁，要求多个进程同时访问同一个锁的时候，后访问的进程被挂起到阻塞队列。第一个进程释放该锁后，第二个进程才被唤醒，再去获取锁继续执行。完成实验后使用给出的测试任务可以打印出指定的结果，如图P2-4所示。

文件介绍

请基于任务 1 的项目代码继续进行实现。

实验步骤

1. 完成 sched.c 中的 do_unblock 方法、do_block 方法，要求其完成对进程的挂起和解除挂起操作。

2. 实现互斥锁的操作（位于 lock.c 中）：锁的初始化（do_mutex_lock_init)、申请（do_mutex_lock_acquire)、释放（do_mutex_lock_release）方法。

3. 运行给定的测试任务（test.c 中 lock_tasks 数组中的三个任务），可以打印出给定结果：两个任务轮流抢占锁，抢占成功会在屏幕打印“Hash acquired lock and running”，抢占不成功会打印“Applying for a lock”表示还在等待。

图 P2-4: current_running 指针和任务切换的关系

注意事项

1. 一个任务执行 do_block 时因为被阻塞，需要切换到其他的任务，因此涉及到任务的切换，因此需要保存现场，重新调度，恢复现场。

2. 请思考一个进程在获取锁失败后会被挂起到哪个队列里，以及在锁的释放时如何找到这个队列进行 unblock 操作。

3. 在设计完成锁之后，请考虑设计的合理性以及拓展性，比如：是否支持一个进程获取多把锁，是否支持两个以上进程同时请求锁并被阻塞。

5 例外处理

在开始这一节之前，请大家注意：S-core，A-core，C-core 三个级别将在这一节有着不一样的课程要求，请大家阅读之后根据自己的实际情况选做。

在 RISC-V 中，将中断和异常统称为例外。通俗的点说，它是程序在正常执行过程中的强制转移。产生例外的原因有很多，有一些例外是主动触发的，比如 syscall 调用，有一些例外是被动触发的，比如硬件异常。RISC-V 例外处理相关的官方文档参考 RISC-V 特权体系结构手册 [5]。

在这次实验大家需要掌握和实现 RISC-V 下例外处理流程，并且实现时钟中断以及系统调用的例外处理代码。经过本次实验，你的操作系统将具备例外处理能力，实现任务的抢占式调度，以及系统调用处理模块。

5.1 RISC-V 特权体系结构概述

RISC-V 的特权体系结构中，特权级划分为 3 个层级：Machine、Supervisor 和 User。Machine 态用于 BIOS 等底层环境。Supervisor 用于操作系统，User 是普通的用户程序。

RISC-V 比较有特色的是，如果一些嵌入式系统不需要 3 个级别，那么也可以不实现 Supervisor 态，只使用 User 和 Machine 两个状态。

在本实验中，我们需要设置时钟，进行中断处理等。这些操作大多都是通过操作 CSR 寄存器完成的。Supervisor 态的寄存器如表P2-3所示。

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Trap Handling			
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.

表 P2-3: Currently allocated RISC-V supervisor-level CSR addresses[5]

读写 CSR 寄存器可以使用 csrr、csrw、csrc、csrs 等等指令进行，具体的指令可以参考 [6]，或者前面章节中给出的伪指令表。可以看到，特权寄存器可以分成两组，一组是例外的触发相关的控制寄存器，另一组是例外处理相关的寄存器。

5.2 例外处理流程

我们将例外处理的流程分为两部分，一部分是例外的触发，另一部分是例外的处理。

例外的触发

当发生异常时，处理器执行地址会将发生异常的地址放入 SEPC 寄存器，然后自动跳转到 STVEC 中存放的地址处（这个过程是硬件自动完成的）。这个地址就是中断处理函数的入口。STVEC 的结构如图P2-7所示。除了第 2 位以外，其余的部分都是中断处理函数的地址。SXLEN 代表处理器的位数，我们这里是 64 位，所以 SXLEN 是 64。由于 RISC-V 是 4 字节对齐的，所以地址的低 2 位一定是 0。于是，低 2 位就可以移作他用。这也是为什么低 2 位是 MODE 的原因。MODE 一共有两种：

Direct 发生任意的例外，处理器都会将 pc 寄存器的值设置为 STVEC 的 base 的值。换句话说，就是 stvec 存放的地址就是中断处理函数的入口地址。

Vectored 发生例外时，硬件会将 pc 设置为 BASE+4*cause。这种模式下，大家可以按照 cause 类型组织一个跳转表，然后把表的首地址存到 stvec 中。

本实验的 start_code 是按照 Direct 模式设计的，建议大家使用 Direct 模式。

在设置好中断处理函数以后，一旦有例外被触发，就会自动跳到中断处理函数处，从而开始中断处理的流程。但中断能否触发还取决于两个关键的寄存器：sie 和 sstatus。

SIE 的结构如图P2-5所示。当 SIE 的对应位清空的时候，代表屏蔽相应的例外。如果 SIE 的对应位为 1，则代表打开相应的例外。SIE 的作用可以理解为，是否使能中断。一旦该位被清空，则代表此类中断彻底被屏蔽。注意区分 SIE 和 SSTATUS 寄存器中的 SIE 的作用的区别。

SXLEN-1 … 10	9	8 … 6	5	4 … 2	1	0
WPRI	SEIE	**WPRI**	STIE	**WPRI**	SSIE	**WPRI**
SXLEN-10	1	3	1	3	1	1

图 P2-5: Supervisor interrupt-enable register (`sie`).[5]

SSTATUS 的结构如图P2-6所示。该寄存器中同样也有一个 SIE 位置。它同样可以用于使能中断。当硬件发生中断时，硬件会自动将 SSTATUS 里面的 SIE 置为 0，将 SPIE 置为原来的 SIE。当执行 SRET 时，硬件会将 SPIE 置为 1，SSTATUS 中的 SIE 置为原来的 SPIE。SIE 寄存器不会变化。

SXLEN-1	SXLEN-2 … 34	33 … 32	31 … 20	19	18	17 …
SD	**WPRI**	UXL[1:0]	**WPRI**	MXR	SUM	**WPRI** …
1	SXLEN-35	2	12	1	1	1 …

16 15	14 13	12 … 9	8	7 … 6	5	4 … 2	1	0
XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	UBE	SPIE	**WPRI**	SIE	**WPRI**
2	2	4	1	1	1	1	3	1	1

图 P2-6: Supervisor-mode status register (`sstatus`) for RV64.[5]

SXLEN-1 … 2	1 … 0
BASE[SXLEN-1:2] (**WARL**)	MODE (**WARL**)
SXLEN-2	2

图 P2-7: Supervisor trap vector base address register (`stvec`).[5]

例外的处理

触发例外的情况有很多，中断就是例外的一种，中断又可以分为时钟中断、设备中断等，而设备中断又可以分为键盘中断、串口中断等。那么如何在发生一个例外后，准确的判断例外发生的原因，最后跳转到负责处理该例外的代码去执行呢？

其实在 RISC-V 下，以中断处理为例（假设为 Direct 模式），我们将中断例外的处理分为三级，每一级的处理过程如下：第一级：各种情况下例外的总入口，即 STVEC 中存放的地址。每当 CPU 发现一个例外，都会从执行地址跳转到这个例外向量入口，这也是 RISC-V 架构下所有例外的总入口，这第一级的跳转是由硬件完成的，并不需要我们去实现；

第二级：这部分是处理例外的第二阶段，它主要完成对例外种类的确定，然后根据不同种类的例外，跳转到该例外的入口。对于例外种类的确定，可以通过 CSR 中的 scause 寄存器来区分不同例外的入口，scause 寄存器的结构如图P2-8所示。

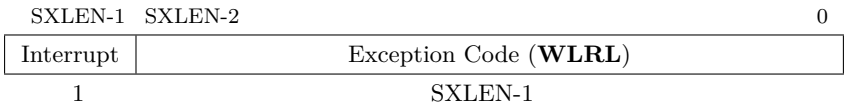

图 P2-8: Supervisor Cause register `scause`.[5]

具体的发生中断或异常的原因见表P2-4。可以看到，当 Interrupt 域为 1 的时候，说明触发的例外类型为中断。本次实验中的时钟中断，就是表中的 Supervisor timer interrupt。而系统调用就是表中的 Environment call from S-mode。

根据具体的 cause 跳转到对应的中断处理函数。

第三级：相应的中断处理函数负责具体地处理每个中断。例如时钟中断的处理函数就需要视情况进行重新调度等。

例外处理基本流程

当例外发生时，处理器会自动将 SSTATUS 里面的 SIE 位置 0，原先的 SIE 被保存到 SPIE。因此，当例外发生时，其他例外就自动被屏蔽了。我们需要保存发生例外时的现场，并调用相应的例外处理函数。对于例外的处理，针对不同的例外需要具体实现。

在 start_code 的设计里，保存的现场、例外原因以及 stval 寄存器的值会被传递给 interrupt_helper。这样设计是为了以最快的速度进入到 C 语言的代码。避免大段编写汇编代码。在 C 语言函数中，再根据例外触发原因，调用不同的例外处理函数即可。在处理例外后对保存的现场进行还原，最后进行例外的返回。这里值得大家注意的是，我们并不需要自己手工重新打开中断。因为在还原现场时，sstatus 的值也会被还原。还原后，sret 指令会使得硬件自动把 sie 设置为 spie。而 spie 是中断发生前 sie 的状态，应该是开启状态。所以 sret 后，sie 自然是开启状态。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Available for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	≥64	*Reserved*

表 P2-4: Supervisor cause register (`scause`) values after trap.[5]

例外的返回

例外的返回是通过 sret 指令进行返回的。当一个例外发生时，硬件会自动的将发生例外的地址保存到 sepc 寄存器，之后跳到例外处理入口。当例外处理结束后，使用 sret 指令就可以返回 sepc 所指向的地址，也就是发生异常前运行到地址。这里提示一点：对于系统调用来说，需要大家返回 sepc+4 的位置。这是因为，sepc 是触发系统调用的 ecall 指令，如果还返回到这个 ecall 的地址，就又会触发一遍系统调用，就没完没了了。所以需要自行修改 spec 的值，使其跳回 sepc+4 的位置。

S-core 的要求

在例外处理这一部分，我们对 S-core 的要求是在例外发生时只打印报错信息，不需要例外的返回。这里的报错信息需要包括发生例外的指令地址以及出错的地址（这两者不一定相等，因为有些例外是因为指令本身出错导致的，有些例外是指令里调用的地址出错导致的）。那么 S-core 的例外处理流程只需要做一件事情就是打印信息。这个打印操作需要实现在例外的入口。

A-core 和 C-core 的同学需要接着看本节下面的任务书内容，完成后续任务。

5.3 带有内核态保护的系统调用

之前的任务里，我们实现的系统调用都是直接调用了内核提供的函数本身，但其实作为用户进程的任务应该是不允许直接访问内核代码段的，这样我们的系统调用就需要使用例外这一接口来实现。

这样的系统调用也是例外的一种，只不过这种中断是用户主动触发的，我们触发系统调用中断的方式是在使用 ecall 汇编指令，当触发系统调用中断时和处理其他例外一样，处理器会自动跳入例外处理入口，保存用户态现场，然后进入到内核的系统调用处理代码段，当调用完内核代码后返回用户态现场。

在以后的测试任务中，我们的任务都是用户进程，我们的实现代码在内核态，因此最后我们还需要对内核的代码实现一步系统调用的封装，提供给用户进程使用。

tiny_libc

为了更贴近真实的环境，start_code 单独实现了一个超小型的 libc 库。为了严格区分用户态和内核态，我们进行如下约定，凡是 start code 中 tiny_libc 中的功能以及 sys/syscall.h 是可以在用户态调用的，其余都是内核态的功能，用户态不得直接调用。更具体的，所有用户态的程序，只允许使用 tiny_libc/include 中定义的功能以及 sys/syscall.h 中定义的功能。而内核态则反过来，不得使用用户态的这些功能。

当然，大部分用户态要运行的测试程序都是由 start_code 提供的，但还是在此做出说明，希望大家能够理解用户态和内核态的区别，以及 C 库的作用。

系统计时器

在一般的系统中，计时器会是一个单独的功能，为了体现这一点，我们的 start code 中搭好了一个简易的 timer 队列。当然，有部分函数需要大家自己实现。timer 的作用是定时，实现到多少个 tick 的时候自动调用某个回调函数。这里，我们希望大家采用 timer 来实现 sleep 的功能。如果想看真实系统中的 timer 实现，可以参考 RT-Thread 实时嵌入式系统 [7]。

为了辅助实现 timer，start code 中实现好了一个小的 timer 对象池，感兴趣的同学建议理解一下相关的代码。

5.4 任务 3：系统调用

实验要求

1. 掌握 RISC-V 下系统调用处理流程，实现系统调用处理逻辑。
2. 实现系统调用 sleep 方法。
3. 使用给出的测试任务，打印出正确结果（如图P2-9和图P2-10）。
4. 将任务 1 和任务 2 的 sys_yield，printf 以及锁的三个函数改造成调用内核函数的用户态系统调用。

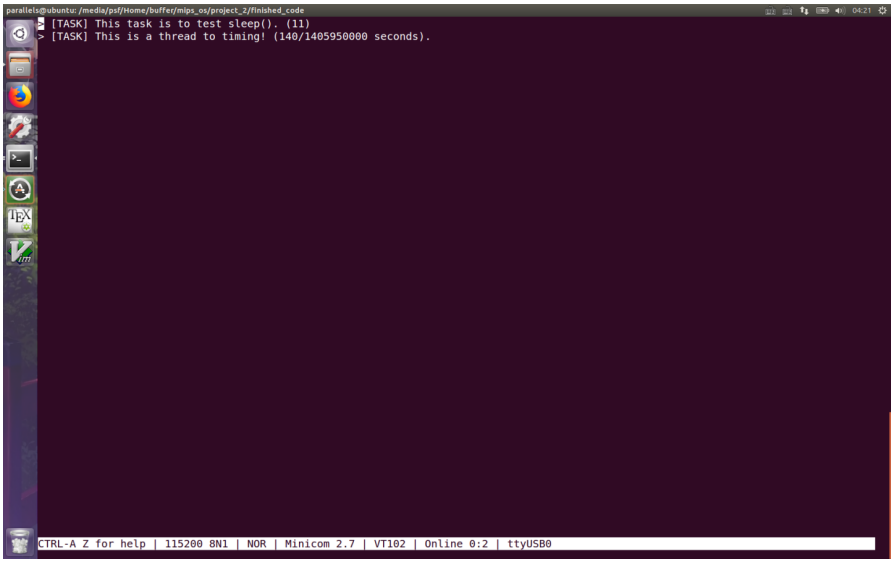

图 P2-9: 实验结果 _1

图 P2-10: 实验结果 _2

文件说明

请基于任务 2 的代码继续进行实验。

实验步骤

1. 完善 main.c 中系统调用相关初始化（init_syscall)。
2. 完善 tinylibc 中的 syscall.S 中的 invoke_syscall 方法，需要完善的内容为调用 ecall 指令发起一次系统调用。
3. 理解 kernel/syscall/syscall.c 中 handle_syscall 方法，理解如何根据系统调用号选择要跳转的系统调用函数进行跳转。
4. 实现 sched.c 中 do_sleep 方法以及 check_sleeping 方法，并实现其系统调用 sleep 方法。
5. 改造我们之前完成的 sys_yield，mthread_mutex_init，mthread_mutex_lock，mthread_mutex_unlock 以及 printf 为内核系统调用版本。
6. 运行给定的测试任务（test.c 中 timer_tasks 数组中的两个任务、test.c 中 sched2_tasks 数组中的三个任务），要求打印出正确结果。

注意事项

1. 了解 RISC-V 下 ecall 指令的作用，该指令会触发系统调用例外。RISC-V 在所有特权级下都用 ecall 执行系统调用。Supervisor 态 ecall 会触发 machine 态的例外，user 态的 ecall 会触发 supervisor 态的中断。所以大家务必注意，要让 USER 模式的进程运行在用户态。
2. sleep 方法的功能为将调用该方法的进程挂起到全局阻塞队列，当睡眠时间达到后再由调度器从阻塞队列（blocked queue）将其加入到就绪队列（ready queue）中继续运行。
3. main.c 的开头会调用 SBI 函数读取 CPU 频率，请大家参考 kernel/sched/time.c 文件，使用 get_timer 函数获取当前 CPU 时间
4. 三个关于锁的系统调用需要给它们安排新的系统调用号，start-code 给出的系统调用号的定义都在 include/os/syscall_number.h 中。
5. 请认真思考系统调用模块的可拓展性，使得自己的设计便于拓展。
6. 在我们之前的实验中我们一直是使用 printk 作为输出函数，在具备了系统调用模块后我们可以使用用户级的打印函数 printf，但是使用 printf 的前提是完成系统调用 sys_write（其实就是将 screen_write 封装为系统调用，在 printf 函数里调用），请参考第五节打印函数的内容了解 printf 函数。

5.5 时钟中断

在之前的任务里我们已经实现了任务的非抢占式调度，但是你可能已经看出了问题，那就是在我们的任务运行时，需要不断的使用 sys_yield 方法去交出控制权，但其实在一个操作系统中，决定交不交出控制权的不是任务本身，而是操作系统。因此我们需要使用时钟中断去打断正在运行的任务，并在时钟中断的例外处理部分进行任务的切换，从而实现基于时间片的抢占式调度。

时钟相关的寄存器都在 machine 级，supervisor 级无法直接控制相关的寄存器。需要使用 sbi_set_timer 设置时钟的触发。关于 SBI 的详细描述，请查看官方文档 [8]。设置的内容为下次触发时钟中断的时钟数，因此设置之前需要读取当前时间，计算下次时钟中断的时间后设置进去。

5.6 任务 4：时钟中断、抢占式调度

实验要求

1. 掌握 RISC-V 下中断处理流程，实现中断处理逻辑。
2. 实现时钟中断处理逻辑，并基于时钟中断实现轮转式抢占式中断。
3. 运行给定的测试任务，能正确输出结果（如图P2-11所示）。

图 P2-11: 实验结果

文件说明

请基于任务 3 的代码继续进行实验。

实验步骤

1. 完善 trap.S 中 setup_exception 代码，主要完成异常处理相关的初始化内容：设置 STVEC 寄存器，打开 CSR 的 SIE 寄存器的中断使能。
2. 完善 irq.c 中 init_exception 代码，主要初始化各个 cause 对应的处理函数。
3. 完善 entry.S 中 exception_handler_entry，主要完成例外处理入口相关内容：保存现场、根据 cause 寄存器的例外触发状态跳转到中断分发函数 (interrupt_helper)。
4. 完善 irq.c 中 reset_timer 代码，主要每次触发时钟中断时重设 timer，重新调度等，具体内容请同学们自己思考如何实现。
5. 运行 test.c 中 sched2_tasks 数组中的三个任务以及 lock2_tasks 数组中的三个任务，要求打印出和任务 1 一样的正确结果。

注意事项

1. 关于如何正确的开始一个任务的第一次调度，在抢占调度下是和非抢占调度是不同的，希望大家仔细思考如何在抢占调度模式下对一个任务发起第一次调度。
2. 有了中断以后，打印可能会互相争用屏幕，就像多个进程同时使用一台打印机似的，每个人都随便打出来一部分。所以如果想打印出完美的效果，要么在打印时关闭抢占，要么保存和恢复每个进程打印光标的位置。start code 中添加了部分保存光标位置的代码，请大家在调度的时候，恢复一下 pcb 中记录的光标位置。

要点解读

这里关键是要理解中断的概念，以及为何我们要做中断处理。这些大家在教科书上已经学过了，请结合实践仔细思考。中断可能会带来各种各样的混乱。当你发现有一些离奇的错误的时候，可以考虑是否是自己的栈出了问题。栈指针一旦设置错误或者保存恢复得不正确，很有可能带来难以调试的错误。所以当有时候搞不清楚错在哪里时，可以考虑一下是不是栈寄存器设置错了。

5.7 任务 5：fork 和优先级调度

到任务 4 为止就是 Project2 的 A-core 需要完成的任务了，任务 5 为 C-core 需要完成的任务，包括了 fork 和优先级调度两个功能。

实验要求

1. 实现 fork 系统调用，使进程可以使用 fork 系统调用复制一个和自身相同的进程。
2. 实现 prior 系统调用，使进程可以使用 prior 系统调用设定自己的调度优先级。

3. 实现测试程序，包含如下功能：父进程循环打印一个语句，其中包含一个从 0 开始随着循环次数不断增长的数字（格式例如 This's father process(12))。循环打印的同时循环等待键盘输入，在收到键盘输入后 fork 一个子进程。子进程根据父进程的输入先调用 prior 系统调用设定自己的优先级（例如输入为 3，则优先级设为 3)，另外子进程也循环打印相似的语句（例如 This's child process(4))，但是包含的数字从 fork 时父进程的数字开始增长。

要点解读

1. 子进程被创建出来时，会复用父进程的代码并复制堆栈数据到自己的堆栈位置，这样子进程才能知道创建时父进程的数字增长进度以及父进程接收到的输入数字。

2. 子进程的优先级越高，显示上它的数字增长就越快。

3. 如果父进程使用全局变量的话，子进程也使用这个变量的话会有两个进程共享变量的问题，所以请不要使用全局变量。

4. 如果把 fork 封装成像其他系统调用一样的系统调用的话，那么在调用 ecall 指令之前会经过调用名为 sys_fork 的 C 语言函数，这样它的进入这个函数时会修改 sp 和 fp 寄存器，导致子进程在退出这个函数的时候堆栈被加载成父进程的堆栈位置。类似的，如果我们在子进程中返回了上一级函数，也会导致堆栈的错误。所以我们这里建议大家使用汇编函数实现 sys_fork，并不要使用 return 语句返回上一层函数，这样避免 C 语言在编译时自动加入对 sp 和 fp 寄存器的修改，给后面的处理带来麻烦。

5. 由于存在上面这两个问题，我们可以看出这样实现的 fork 不够完美，而这两个问题在引入了虚拟内存之后会轻松的得到完美解决。在没有虚存的情况下如何完美的解决，有兴趣的同学可以考虑一下这个问题。

Project2 功能总结

在这里，我们给大家总结了 S-core，A-core，C-core 需要完成的功能，请大家查看，注意评分等级越高，需要完成的功能是叠加的。随着同时运行的进程不断增多，对操作系统稳定性的要求就也越高。也就是说，做 C-core 的同学需要完成下面列出的所有任务，让包括这些功能的进程同时运行。

评分等级	需要完成的任务
S-core	非抢占式调度，锁，进入例外打印报错
A-core	带内核态保护的系统调用，时钟中断处理，抢占式调度
C-core	优先级调度，fork

表 P2-5: 各个等级需要完成的任务列表

6 附录：打印函数

这一节的内容是关于打印相关的函数设置，并不会介绍新的任务，只是便于同学们理解 start-code 中的相关代码，建议同学仔细阅读这一节之后再配合阅读 start-code。

关于该实验的打印驱动我们在实验的 start_code 里实现了（位于 drivers 文件夹 screen.c 中），并分别给出了内核级的打印方法 printk 以及用户级的打印方法 printf（位于 libs 文件夹下）。

6.1 VT100 控制码

对于开发板的打印，我们只能使用串口 IO，因此在输出的时候是往串口寄存器写字符，然后串口通过 VT100 虚拟终端最终呈现到屏幕的，如图P2-12所示。

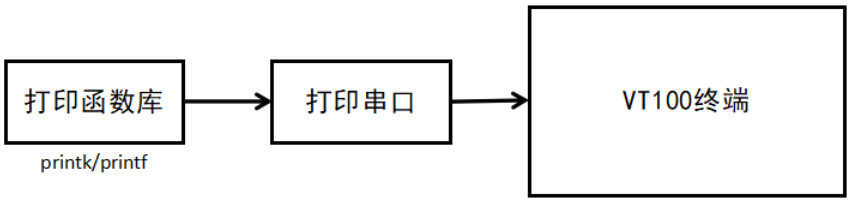

图 P2-12: 打印流程

VT100 是一个终端类型定义，VT100 控制码是用来在终端扩展显示的代码，我们只需要往串口输入一些特定的字符串，就可以完成 VT100 终端的打印控制，比如光标移动，字体变色等功能，部分控制码如P2-6所示。

编号	控制码	描述	编号	控制码	描述
1	\033[0m	关闭所有属性	6	\033[y;xH	设置光标位置到（x,y）
2	\033[nA	光标上移 n 行	7	\033[2J	清屏
3	\033[nB	光标下移 n 行	8	\033[?25l	隐藏光标
4	\033[nC	光标左移 n 行	9	\033[?25h	显示光标
5	\033[nD	光标右移 n 行			

表 P2-6: VT100 部分控制码

6.2 屏幕模拟驱动

由于我们开发板的打印只能通过串口一个字符一个字符的进行输出，没有显存这么一说，因此我们的做法就是在内存模拟一块显存，然后每次往模拟的显存里写数据（screen_write 方法），每次在时钟中断处理函数（irq_timer）里去一次性的将模拟显存里的数据刷新到串口里（screen_reflush 方法）。这么做的好处就是可以在处理一些进程对屏幕的占用时更加清楚，不会造成由于多任务模式下屏幕打印混乱的情况。当然，这种做法只是对已经具备了中断的情况而言，因此屏幕模拟驱动只适用于用户及的 printf

方法，而对于内核级的 printk 方法，我们的做法依旧是每次还是直接往串口写数据。如图P2-13所示。

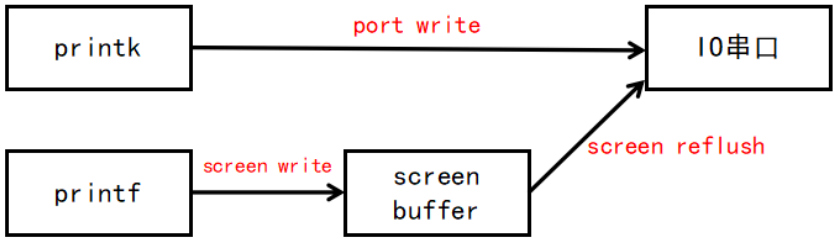

图 P2-13: 开发板打印过程

6.3 打印优化

由于每个时钟周期都需要进行 screen buffer 的刷新，一般而言，buffer 的大小为 80 * 35 大小，因此每次需要刷新上千的字符到串口，这无疑是非常耗时的，因此我们可以只修改从这上一次刷新到这一次刷新期间修改过位置的字符，这样就可以大大的提升我们的打印速度了。关于优化的具体实现可以参考 driver/screen.c 中 screen_reflush 方法。

参考文献

[1] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, pp. 623–624, Oct. 1965.

[2] @ 我的上铺叫路遥, "伙伴分配器的一个极简实现." https://coolshell.cn/articles/10427.html, 2013. [Online; accessed 31-August-2021].

[3] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 1994.

[4] M. Warton, *Single Kernel Stack L4*. PhD thesis, 11 2005.

[5] "The risc-v instruction set manual volume ii: Privileged architecture v1.10," 2017.

[6] A. B. Palmer Dabbelt, Michael Clark, "Risc-v assembly programmer's manual." https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md, 2019. [Online; accessed 27-August-2021].

[7] RT-Thread, "Rt-thread 文档-时钟管理." https://www.rt-thread.org/document/site/programming-manual/timer/timer/, 2019. [Online; accessed 20-September-2021].

[8] A. P. Palmer Dabbelt, "Risc-v supervisor binary interface specificationl." https://github.com/riscv/riscv-sbi-doc/blob/master/riscv-sbi.adoc, 2019. [Online; accessed 2-September-2021].