国科大操作系统研讨课任务书

RISC-V 版本

版本 2023

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包括了操作系统和用户程序的镜像文件，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一次完整执行一个应用程序之后再切换到下一个程序运行，无法让多个程序同时运行。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 实现线程的创建 thread_create，使用户进程可以创建线程并协同执行。

需要强调的是，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。由于引入了时钟中断，时钟的不确定性和可中断指令流的特点会给大家的理解和调试带来不少难点。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基础。因此一个鲁棒性高的例外处理和任务调度功能会对你后续的 Project 起到重要作用。希望同学们可以认真学习，遇到问题时多和老师同学进行交流。

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. 理解并实现锁和进程阻塞
3. 理解中断处理和进程调度的基本概念

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，我们暂不涉及。我们首先为操作系统构建进程管理机制。在进入到 kernel 的 main 函数后，我们打印出了 “Hello OS!”，并且可以跳转到指定的用户程序中去执行，我们可以认为此时操作系统已经拥有了一个内核进程，但即使这时已经可以运行用户程序的代码，但我们实际上都是在一个进程内，并没有切换到一个新的用户进程中去。想开启另一个新的进程，我们需要进行进程控制块初始化、任务切换这两步后，才可以开始运行一个新的进程。

此外在用户程序中，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 开始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更完善的功能。

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。

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？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，我们需要为每一个运行的进程分配其所需要的栈空间。每产生一个进程，我们还需要为其分配代码段、数据段等内存空间供其运行。显然，我们首先需要建立一个能管理并分配空闲内存空间的方式，来管理开发板上的空闲内存。

我们建议的地址空间划分如所表P2-1所示，从 0x52500000 开始都是空闲的，可以供大家任意使用。

注意，在初始化 PCB 时，同学们需要为每个进程分别分配内核栈空间与用户栈空间，我们也在 mm.c 中为大家提供了一个简单的内存分配算法。当然，感兴趣的同学也

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
0x50500000-0x52000000	供内核动态分配使用的内存
0x52000000-0x52500000	用户程序的数据段/代码段等
0x52500000-0x60000000	供用户动态分配使用的内存

表 P2-1: 地址空间用途划分

可以自行调研诸如伙伴内存分配算法等更先进的方法。

3.3 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）保存到栈中，然后从其他进程的 PCB 中选择一个，从这个 PCB 对应的栈里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。任务切换实际上是一种程序修改自身运行环境和状态的行为，是操作系统内核独有的一种行为，在其他应用场景很少会遇到。因此任务切换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的过程，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。

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？这个问题就需要同学们参考预备课中介绍过的内容来进行设计，也可以自行查阅 RISC-V 手册 [1] 的第 3.2 节去了解更多的信息。

3.4 PCB 的初始化

在一个进程做好准备开始运行之前，PCB 需要做好初始化的工作。我们需要给予 PCB 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，我们要将对应程序的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。这里会出现一个问题：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, 根本没有上次被保存的东西。那么这个问题需要同学们自己考虑一个合适的设计。简单来说，PCB 需要在初始化的过程中将这个入口地址做成一个假的现场，在第一次执行的时候从假的现场中恢复执行。

3.5 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。

3.6 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 switch_to 切换。
3. 同时运行测试任务”print1”、”print2” 和”fly”，能正确输出结果。参考结果如图P2-1所示。

实验步骤

完成了 Project1 之后，我们在 git 上提供了 Project2 的分支，通过 git merge 命令将能获取到本次实验的新内容更新。打上这个补丁之后，代码文件夹就会自动补上 Project2 提供的新文件和对原有文件的修改。这样每个同学自己实现的原本的 Project1 部分的代码就不会被改掉，可以继续在 Project2 中沿用。当然这个过程中可能会有补丁报错的情

```
> [TASK] This task is to test scheduler. (226)
> [TASK] This task is to test scheduler. (225)
```

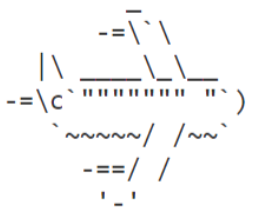

图 P2-1: P2-task1 参考运行结果

况，请同学们根据报错的位置再去手动修改对应的文件。如果冲突比较多，手动修改出现了混乱，难以恢复，可以强制回退到 merge 前的版本，尝试重新 merge 和修改。

下面给出了 Project2 任务 1 的实验步骤：

1. 完成 sched.h 中 PCB 结构体设计，以及 main.c 中 init_pcb 的 PCB 初始化方法。
2. 实现 entry.S 中的 switch_to 汇编函数，使其可以将当前运行进程的执行现场保存在 current_running 指向的 PCB 中，以及将 current_running 指向的 PCB 中的执行现场进行恢复。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 do_scheduler 方法，使其可以完成任务的调度切换。
4. 实现 syscall.c 中的 sys_yield、sys_move_cursor、sys_write、sys_reflush 方法的简易版本。具体来说，将内核函数挂载到 jmptable，并在 sys_函数中调用 call_jmptab 来使用这些内核函数。
5. 运行测试任务”print1”、”print2” 和”fly”。其中”print1”、”print2” 任务在屏幕上方交替的打印字符串 “This task is to test scheduler”，”fly” 任务在屏幕上画出一个飞机，并从左向右不断移动。

注意事项

1. 在 Project2 的前两个任务中，所用到的测试程序虽然使用了系统调用的 API（头文件为 unistd.h），但内部实现仍然是跳转表，这还不是真正的系统调用。这是因

为在前两个任务中，内核与用户程序之间仍然运行在 RISC-V 的同一特权级（任务三中会详细介绍），同时由于内核与用户分开编译，因此用户程序需要暂时使用内核提供的的跳转表函数进行过渡。而等到任务三实现了系统调用之后，内核与用户程序之间特权级分离，这时大家就需要抛弃跳转表，重新实现系统调用函数。对于 S-core 的同学，在任务三中不需要实现系统调用，可以一直沿用跳转表。

2. switch_to 函数是在内核主动切换进程时调用的。内核明确的知道在调用这个函数后就进入进程切换了，且因为内核是主动调用 switch_to，所以遵循函数调用的相关约定。所有需要由调用者保存的寄存器已经在函数调用前都保存在栈上了，switch_to 中只需要保存所有应该由被调用者保存的寄存器即可。

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，第二个进程继续在第一个进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。为了处理多进程竞争单个资源这个普遍性问题，操作系统必须引入一种“锁” 的机制。进程访问数据前，对要访问的数据加锁。要求一次最多只能有一个进程对其访问。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，并避免出现死锁。

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P2-2所示。

文件介绍

请基于任务 1 的项目代码继续进行实现。

```
> [TASK] This task is to test scheduler. (96)
> [TASK] This task is to test scheduler. (96)
> [TASK] Has acquired lock and running.(2)
> [TASK] Applying for a lock.
```

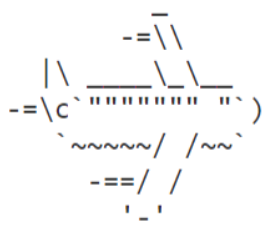

图 P2-2: P2-task2 参考运行结果

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）：锁机制的全局初始化（init_locks），以及互斥锁的初始化（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”lock1” 和”lock2”（跳转表 API 版本），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。

注意事项

1. 由于各个进程间的地址空间应当是分隔开来的（在 Project4 中会通过虚存机制彻底做到这一点），这使得 lock1 进程和 lock2 进程无法在同一地址空间中使用同一把锁。因此，大家需要把互斥锁作为内核的资源进行管理：用户程序通过 key 定位到内核中具体的一个互斥锁，然后初始化后得到一个句柄 (handle)，再通过这个 handle 去获得与释放互斥锁。

2. 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，需要保存现场，重新调度，恢复现场。

3. 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。

4. 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。也就是说，前面的任务 1 和任务 2 是所有同学都要完成的内容。

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 [2]。

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 A-Core 和 C-Core 要求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。A-Core 和 C-Core 则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P2-2所示。

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 [3]，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P2-5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。

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 P2-2: Currently allocated RISC-V supervisor-level CSR addresses[2]

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P2-3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 P2-3: Supervisor interrupt-enable register (sie).[2]

SSTATUS 的结构如图P2-4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 SSTATUS.SIE 为 0 时，所有的中断都被屏蔽。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。

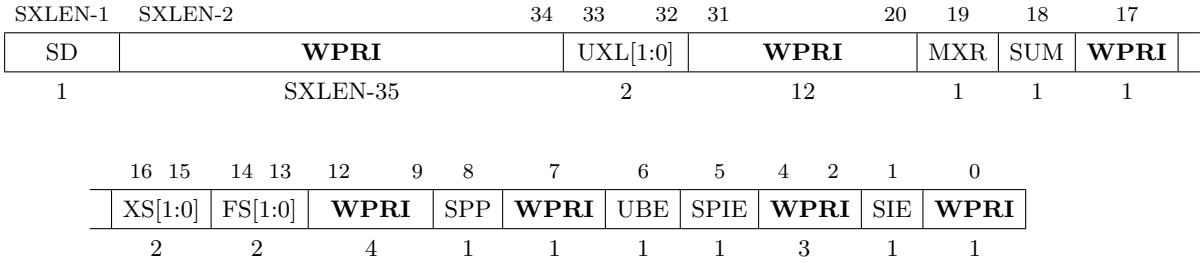

图 P2-4: Supervisor-mode status register (`sstatus`) for RV64.[2]

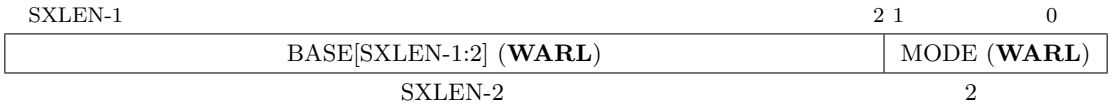

图 P2-5: Supervisor trap vector base address register (`stvec`).[2]

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P2-6所示。

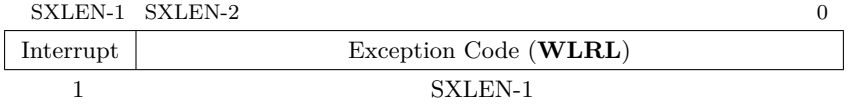

图 P2-6: Supervisor Cause register `scause`.[2]

具体的发生中断或异常的原因见表P2-3。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。

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 P2-3: Supervisor cause register (`scause`) values after trap.[2]

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。

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。

S-Core 正常情况下是不需要例外处理的。但是不能排除仍然会有没考虑到的各种软、硬件的例外会发生，如果不进行例外处理，操作系统一般会直接“跑飞”了。因此，S-core 需要做的唯一的例外处理就是报告“例外的发生”。除了要求的出错地址、寄存器等信息外，同学们可以自行设计更友好的“现场报告”。

任务 3S: 打印例外信息

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的相关代码段，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；而内核态则反过来，不得使用用户态的这些功能。这一点也通过内核用户分开编译得到了一定程度上的保证，感兴趣的同学可以自行查看 Makefile 了解具体是如何做到分开编译的。

虽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。

当然，在有了虚存机制后，用户态和内核态的安全隔离完全由硬件来保证，不需要依靠程序员的“自觉”了。

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. 实现例外入口的初始化、程序上下文的保存与恢复、以及例外结束返回逻辑。
3. 实现 sys_yield、sys_move_cursor、sys_write、sys_sleep 等方法。
4. 使用所有给出的测试任务（syscall 版本），打印出正确结果（如图P2-7所示）。

```
> [TASK] This task is to test scheduler. (92)
> [TASK] This task is to test scheduler. (90)
> [TASK] Has acquired lock and running.(4)
> [TASK] Applying for a lock.
> [TASK] This task is to test sleep. (2)
> [TASK] This is a thread to timing! (9/9142859 seconds).
```

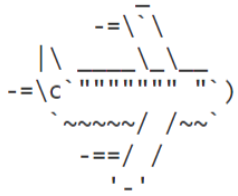

图 P2-7: P2-task3 参考运行结果

文件说明

请基于任务 2 的代码继续进行实验。

实验步骤

1. 完成 main.c 中系统调用相关初始化（init_syscall），以及完善 PCB 内核栈初始化方法（init_pcb_stack）。
2. 完成 trap.S 中 setup_exception 部分代码，本任务中需要设置 STVEC 寄存器；该函数的功能将会在任务四中进一步完善。
3. 完善 irq.c 中的 init_exception 部分代码，以初始化例外入口表（exc_table）和例外入口地址。
4. 实现 entry.S 中的 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例外/中断分发函数（interrupt_helper）。
5. 实现 entry.S 中的 SAVE_CONTEXT、RESTORE_CONTEX 宏定义，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。
6. 实现 entry.S 中的 ret_from_exception，主要完成例外处理收尾相关的内容：恢复现场，并使用 sret 指令返回到 sepc 寄存器所指向的 pc。
7. 实现 tiny_libc/syscall.c 中的 invoke_syscall 方法，需要完成的内容为：使用嵌入式汇编，将参数放入对应的寄存器后调用 ecall 指令发起一次系统调用。同时完成文件内的若干 syscall API。
8. 实现 syscall/syscall.c 中 handle_syscall 方法，需要完成的内容为根据系统调用号选择要跳转的系统调用函数进行跳转。
9. 实现 sched.c 中 do_sleep 方法以及 timer.c 中的 check_sleeping 方法，并在 do_scheduler 中唤醒所有可以唤醒的 PCB。
10. 运行所有给定的测试任务（syscall 版本），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睡眠队列（sleep 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 read_fdt 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. start-code 给出的系统调用号的定义在 arch/riscv/include/asm/unistd.h（内核使用）和 tiny_libc/include/syscall.h（用户使用）中，大家要新增系统调用号的话，注意保证两个文件内的调用号一致。

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 和 sys_reflush（其实就是将 screen_write 和 screen_reflush 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。

7. 从本任务开始，测试程序需要切换到 syscall 版本（头文件为 unistd.h）

8. 从本任务开始，我们需要同学们在提交检查的时候使用 loadbootd 加载内核，关于 loadbootd 的介绍会放在附录中。

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 set_timer 设置时钟的触发。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。

同时，为了允许时钟中断，各位同学需要使能时钟中断，这一点详见 RISC-V 特权级手册以及任务书前面的描述。

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（要求注释掉所有的 sys_yield），能正确输出和任务三一样的结果。

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，打开全局中断使能。
2. 完善 irq.c 中 init_exception 代码，对中断入口表（irq_table）进行初始化。
3. 完成 irq.c 中 handle_irq_timer 函数，以处理时钟中断。处理方法包括：重新设置 timer、重新调度等，具体内容请同学们自己思考如何实现。
4. 在 main.c 中对时钟中断进行初始化，同时将 while (1) 中的 do_scheduler 注释掉，换成下方的 enable_preempt。
5. 运行给定的所有测试程序（syscall 版本，要求注释掉其中所有的 sys_yield），要求打印出和任务 3 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。

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：实现线程的创建 thread_create

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，主要为实现在进程中创建线程并协同执行。

虽然我们还没有涉及到虚存机制，但我们也具备了实现线程的基本条件：所有的用户进程的地址空间其实是可以共享的。请准备挑战 C-Core 的同学，仔细回忆线程的基本定义，一个线程应该具有哪些私有的和共享的资源。记住，线程是执行的基本单位，也就是线程是可以独立调度的；而线程之间是可以共享进程资源的。另外，对 C-Core 的同学，测试程序是要自己来设计的。

实验要求

1. 实现 thread_create 系统调用，通过这个系统调用可以创建一个新的线程并执行指定的代码。
2. 实现 thread_yield 系统调用，通过这个系统调用，可以实现同一个线程组中的线程的执行权限的切换。

3. 请同学们自己实现测试程序。该程序会创建两个子线程。每个线程对各自的一个从 0 开始的变量进行累加（每次加 1），当一个线程发现自己的变量值比另一个线程的变量值大 20 的时候通过 thread_yield 主动让出执行权限。两个线程在屏幕的不同位置打印各自变量的值，主线程打印两个线程执行 thread_yield 的次数。

要点解读

1. 注意线程的概念，两个线程要执行的函数和对应的函数参数应作为 thread_create 的参数。线程之间是共享代码段和数据段，但是有单独的堆栈。
2. 主线程要打印线程 thread_yield 的次数，由于线程之间共享数据，这个功能可以用一个共享变量来实现。不可以用内核锁，因为这样的开销比较大，不符合线程的特点。
3. 本测试需要 fly 进程一起运行。由于时钟中断和其他进程的存在，两个计数线程不一定是按每次 20 步进的，如果看不出效果可以适当调整这个跨步。
4. 由于线程或进程的退出涉及内存的回收等操作，为了简化，新创建出来的线程和主线程最后都可以用一个死循环让它不退出。有兴趣的同学们也可以考虑一下进程线程退出时需要做什么，把这一功能实现。
5. 请同学们自行设计输出，使呈现出来的输出可以看出两个创建出来的线程也是通过调度交替执行的。

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	实现线程的创建 thread_create

表 P2-4: 各个等级需要完成的任务列表

6 附录

6.1 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（位于 libs 文件夹下）以及用户级的打印方法 printf（位于 tiny_libc 文件夹下）。除此之外，位于 libs 文件夹下的 printv 方法仅供 vt100 系列函数调用，其他函数不应该直接调用 printv。

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P2-8所示。

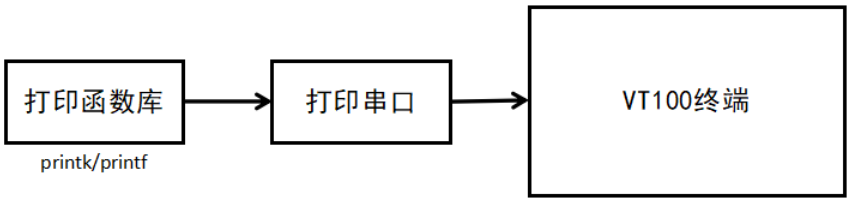

图 P2-8: 打印流程

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P2-5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 P2-5: VT100 部分控制码

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P2-9所示。

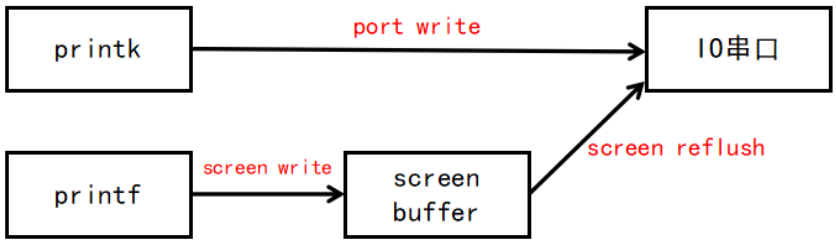

图 P2-9: 开发板打印过程

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。

6.2 关于 DASICS 功能的介绍

在 Project2 的任务 3 中，我们将用户程序运行到了 RISC-V 的 User 态中，而非像之前那样一直运行在 Supervisor 态。在这种状态下，程序的运行只有通过时钟中断或者系统调用这样的例外才能进入 Supervisor 态，才能跳到内核的代码去执行。但是，如果同学们的代码有意无意的在 User 态访问了内核所在的内存地址空间的话，系统是无法报错的。这是因为我们现在还没有启用虚存，没有对内存地址的保护。(虚存这部分内容理论课还没有讲到，研讨课将是 Project4 的内容)

但是为了确保同学们能做到用户与内核之间的地址隔离,我们在开发板硬件和 QEMU 上都加入了名为 DASICS 的功能。该功能通过增加几组硬件寄存器，使得执行在用户地址区间的代码无法访问内核地址区间的数据，一旦访问就会发生例外（DASICS 例外的例外号大于等于 24)。而用户地址区间和内核地址区间的范围已经被我们设置到了寄存器中，该范围就是表P2-1中所介绍的范围。所以请大家分配内存空间的时候满足表中所列出的范围限制。

上面的任务注意事项中也已经告诉大家了，从任务 3 开始大家需要使用 loadbootd 代替 loadboot 命令加载内核并启动，这就是启动 DASICS 功能的标志。大家在调试代码的时候可以为了方便暂时不打开这一功能，但是在检查中我们是会要求使用的，所以请大家最终检查之前自己也使用 loadbootd 来验证一遍。

6.3 内核调试方法——printl

在 Project2 中，我们有了一块小屏幕以及屏幕的驱动函数 screen_* API（位于 drivers/screen.c）。然而根据往年实验课的反馈，大家在内核中使用 print 调试大法的时候，往往需要考虑不能弄乱了测试程序在屏幕上的输出，导致同学们需要在屏幕上小心翼翼地找一个地方输出调试语句，费时又费力。

因此，我们为大家提供了 printl 调试方法。printl 的格式与 printk 相同，但该方法会将调试语句统一输出到 QEMU 日志文件之中（路径为 ~/OSLab-RISC-V/oslab-log.txt）。大家可以使用多块终端进行调试：终端 A 使用 make run/debug 运行 qemu，终端 B 使用 tail -f ~/OSLab-RISC-V/oslab-log.txt 来动态查看日志文件的写入情况，终端 C 使用 make gdb 接入 riscv64-gdb 进行调试。从而大家在使用 print 大法调试的时候，就不必关心 screen 的细节了。

此外，如果同学们想要在用户态使用 printl，可以在任务一、二中将 printl 放入跳转表（参考 printk），也可以在任务三、四中将其封装为系统调用。如果各位有这一方面的需求的话，就请大家自己动手、丰衣足食了。

对往届的同学，printl 函数只在 QEMU 上有用，在 PYNQ 板卡上将会视为一条空函数。今年的同学可以在 PYNQ 板子上也使用 printl 了，希望大家的调试效率会进一步提高。这个功能是上一届的同学贡献的，也欢迎有兴趣的同学在完成课程后，帮我们改进课件方便未来的师弟师妹们。

参考文献

[1] A. W. David Patterson, *RISC-V Reader*. 2018. Available at http://riscvbook.com/chinese/RISC-V-Reader-Chinese-v2p1.pdf.

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

[3] 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].