国科大操作系统研讨课任务书

RISC-V 版本

版本 2025 Fall

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，我们的操作系统一次只能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**的功能。

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，以及锁的实现。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 了解操作系统中定时器中断的触发和处理流程，实现一个定时器中断，并在其基础上实现进程的抢占式调度。

任务五 实现复杂调度算法的功能，使不同速度的用户进程可以通过内核的调度算法达到相同的进度。

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。由于引入了定时器中断 (Timer Interrupt)，定时器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

在操作系统中，进程 (Process) 是资源分配的单位，线程 (Thread) 是调度的单位；二者之间存在一处较大的区别，即对虚拟内存 (Virtual Memory) 的管理方式不同。由于（秋季学期）理论课暂未讲解到虚拟内存的知识，故研讨课暂不涉及此处，我们首先需要为操作系统构建一个进程的管理机制，而非线程。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”**。即使这时已经可以运行用户程序的代码，我们实际上都是在这个内核进程内，并没有切换到一个新的用户进程中去。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。

> **Note 3.1** 在这一段落中提到了 `main` 函数。同学们是否已经理解了框架内容，并知道主函数的位置？如果不知道的话，同学们还需要多加熟悉。如果同学有自己的想法，在后续的实验当中可以对框架的文件结构进行合理的自主调整。

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地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。从这个 Project 开始，我们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

若单论操作系统与其中的“进程”本身，我们可能认为概念比较抽象，难以具体描述；为了描述和控制进程的运行，我们在操作系统当中为进程定义了一个数据结构，并在新建进程时分配一个，即我们所说的进程控制块 (Process Control Block，简称 PCB)。PCB 是进程的重要组成部分，记录了操作系统用于描述进程当前状态和控制进程的全部信息，可能包含进程号、进程状态、发生任务切换时保存的现场（通用寄存器的值）、进程使用的动态地址空间等信息。如果说进程是抽象的概念，那么 PCB 即是对这种抽象的具体代表，是操作系统感知进程的通道，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？我们的计算机中有不少操作系统可以管理的“硬件资源”，常见（也是我们目前的实验能够涉及到的）的有 CPU 和内存两者。值得注意的是，这些硬件资源不能同时被不同的进程所占有，因此我们要保证在约定的条件内，进程对他们的访问具有“原子性”。

目前的操作系统是跑在单核的系统上的，只有一个 CPU，因此这一个 CPU 对应的硬件应该是不同进程所“共享”的。CPU 中有 GPR、CSR，这些寄存器作为硬件资源，能够用来存储数据，进程也需要他们提供数据或控制信号，因此不同进程都需要使用。如果需要切换到不同的进程，这些硬件资源不复为一个进程所占有，一些在不同进程间并不共享的寄存器（比如通用寄存器 GPR）就应当被保留到内存中，待此进程再次被调度至某一 CPU 上运行时再取出并使用。

内存也是被共享的硬件资源，而更准确地说，内存地址空间才是一个被共享的资源，但鉴于我们并未启用虚拟内存，进行良好的内存管理，并不是每一个进程都有一个内存地址空间，内存地址空间这个“硬件”资源也没有能够被分配到实际的硬件资源上，我们只能手动将内存分为不同的几个部分，令进程分开使用。如果你还对上述的解释不甚理解，那么不妨这么看：在编译器和操作系统的配合之下，C 程序运行需要内存上开辟好的栈空间，在这里也一样，我们需要为每一个运行的进程分配其所需要的栈空间。而且每产生一个进程，我们还需要为其分配代码段、数据段等内存空间供其运行时使用。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，避免空闲内存的分配冲突。

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。

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

那么，管理内存有哪些方案呢?

简单内存分配（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]。

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，感兴趣的同学也可以自行调研诸如伙伴内存分配算法等更先进的方法。我们推荐大家在 Project 4 实现对虚拟内存管理的支持之后，尝试实现此类复杂分配算法。

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。对我们经常使用的操作系统和编译器而言，C 语言程序的运行在底层逻辑上是需要用户能够使用的栈的；但是内核又需要一个单独的栈。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，因此操作系统的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：单内核栈和多内核栈 [4]。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。

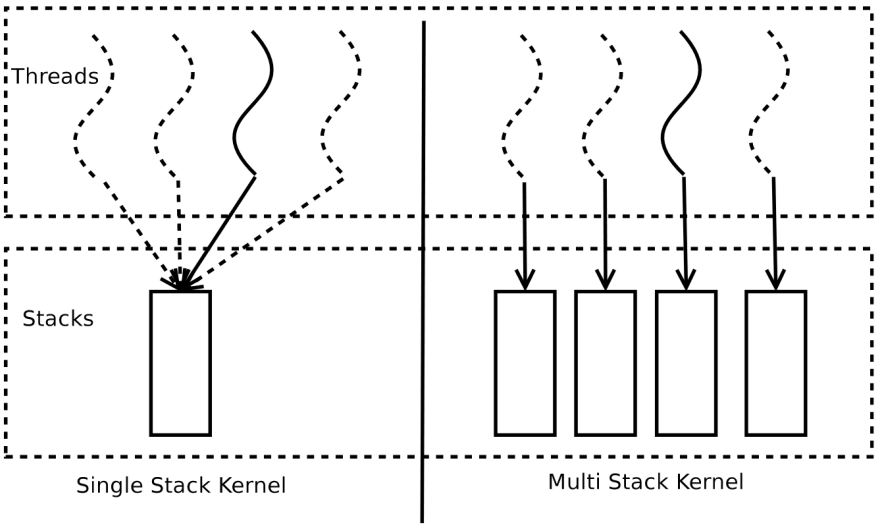

图 P2-1: 两种内核栈设计 [4]

单内核栈的好处是能节约很多内存，不用给每个进程都开一个栈。它的缺点也很显然，陷入内核的这一部分代码必须被划为临界区，并被原子执行完，不能被中断。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。本实验的 start code 也是按照多内核栈设计给出的框架，而大家所熟知的 xv6 则是使用的另一种模式。这种能力对于一些追求低延迟的系统来说是很重要的。以 Linux 为代表的具有可抢占内核的系统基本上都采用了这种设计。

3.4 任务的切换

拥有了 PCB 之后，我们就可以管理进程从而实现进程的切换了。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（当然也可以是 PCB 当中，如果你想要更改这里的实现，可以自行对框架做出改动），然后从其他进程的 PCB 中选择一个可以被执行的，从这个 PCB 对应的栈里保存的现场进行恢复，以实现到下一个进程的切换。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，简单地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` 指向的 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”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，通过周期性触发定时器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 1 中首先实现这种方式。抢占式调度涉及到定时器中断处理等操作，我们在 Project 2 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，而这个函数下一次返回时，已经是进程 B 在运行了。之后，若其他函数调用 `switch_to` 函数，进程 A 可能又会被唤醒，这时 CPU 的指令执行流才会返回到原本的进程 A 当中。进程 A 相当于被“蒙上了双眼”，并不知道在调用此函数后发生了什么，只是以为在调用之后返回而已。当然，同学们可以不用 `switch_to` 这个函数名、使用其他的设计，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

Project 0 中介绍过的内容（通用寄存器使用约定）来进行设计，也可以自行查阅 RISC-V 手册 [5] 的第 3.2 节去了解更多的信息。这里提醒大家：是 ABI 对 GPR 的使用约定，使得 `switch_to` 函数调用后需要保存的现场只占了 32 个 GPR 的一部分。

3.5 PCB 的初始化与准备

在初次进入操作系统内核后，我们需要初始化一系列全局变量，用以存储 PCB 的内存资源也不例外。同学们需要思考：我们实验框架中用以存储 PCB 的资源是什么？它是否需要在开启内核后进行初始化？

在一个进程**准备运行之前**，PCB 需要做好准备的工作。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，我们可能需要将对应程序的入口地址保存到 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，开启进程的运行。这里会出现一个问题：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，当我们准备一个新的 PCB，并且试图切换到新准备好的 PCB 时，会发现没有办法恢复现场，因为这是新制作的 PCB，根本没有上次被保存的东西。这个问题需要同学们自己考虑一个合适的设计。可以考虑两种做法：

1. 制作一个假的现场，即 PCB 在初始化的过程中将一个假的现场发在栈上，供进程切换的时候使用，而这个假现场中就包含这个入口地址，从而在第一次执行的时候从假的现场中恢复执行。

2. 准备一个被调度过的进程，通过复制（操作系统中的 `fork` 操作）这个进程并进行修改，产生一个可以使用的新进程，以运行给定的程序负载。

另外，还需要强调一点。请通过阅读框架代码，找到上述两段初始化、准备 PCB 的过程，分别对应框架代码中的什么函数，并尝试使用他们（或自行设计）完成实验。

3.6 进程调度

进程调度的算法有很多种，比较有名的包括 CFS、BFS、FCFS、多级反馈队列等等。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算法 (RR, Round Robin)：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 队列的实现

在看过上面对于调度的介绍之后，大家会发现操作系统需要一个准备队列。而这个队列最简洁的实现是采用一个链表。相信链表这种数据结构以及相关的函数，大家都在数据结构课中有所训练，因此我们并没有在 start code 中提供队列相关库函数的实现，而是只写出了一些相关的数据结构于 `list.h` 中，建议大家自行建设一个队列操作的库，以实现自己的调用需求。

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 PCB，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P2-2所示。

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

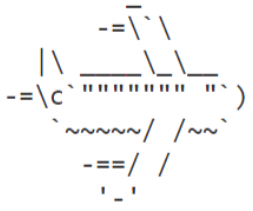

图 P2-2: P2-task1 参考运行结果

实验步骤

完成了 Project 1 之后，我们在 git 的远程仓库中提供了 Project 2 的分支，使用 `git merge` 命令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 Project 2 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 Project 1 部分的代码就不会被改掉，可以继续在 Project 2 中沿用。当然这个过程中可能会有补丁报错的情况，请同学们根据报错的位置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。这里需要强调，git 工具的使用应当是**计算机专业学生的常识**，需要同学们自己通过**自学**等方式进行基础的掌握，并具备通过查阅手册了解高级使用方法的能力。

下面给出了 Project 2 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，请在调用 switch_to 时保证，tp 寄存器中保存的内容为 current_running。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 tiny_libc/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. 取消任务中对 sys_yield 系统调用的注释(若已被注释),并运行测试任务 "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 Project 2 的前两个任务中，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 3 中会详细介绍）；同时，由于内核与用户分开编译，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。等到任务 3 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 3 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。内核明确地知道在调用这个函数后就进入进程切换了，且因为是内核主动调用 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在函数调用前都已经保存在栈上了，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。

3. 前两个进程 "print1" 与 "print2" 后计数器的计数进度应该保持基本一致，不应出现一个很多、一个很少的情况。之后的任务中，大多情况下都需要保持调度的均衡性。

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，非原子性的操作同时进行就会造成不可预见的问题，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的情况，可能会造成最终结果的出错。为了处

理多个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进行访问。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 自旋锁

自旋锁的实现很简单，就是设置一个变量，当一个进程要进入临界区的时候，首先检查这个变量，如果这个变量状态为无进程访问，并且此时没有其他冲突进程在访问这个变量，那么该进程就进入临界区，并且将这个变量置为有进程访问状态。如果这个变量状态为有其他冲突进程在访问，那么就不断使用 `while` 循环重试，直到可以进入临界区。

4.2 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，因此就出现了互斥锁。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地从阻塞队列中重新放到就绪队列，并获得锁。因此，使用互斥锁可以节约 CPU 资源，并避免出现死锁。

4.3 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，后面的进程才被唤醒，再去获取锁继续执行。请注意自己的锁的设计是否能够允许多个进程同时争抢一把锁。

我们的互斥锁拥有已经定义好的 API，请按照 API 的规定进行内部代码设计。

- 初始化（`int sys_mutex_init(int key)`）：接受 `key` 值作为参数，返回一个 `int` 类型的 `handle` 表示某一把互斥锁。要求在一个 `key` 值的生命周期内（即从无进程使用此 `key` 开始，直到最后一个还在使用此 `key` 的进程退出、不复存在），相同的 `key` 总是对应相同的 `handle`。刚刚进入其生命周期的锁应当是未被占有的。

  可自行规定生命周期有交叠的不同的 `key` 是否可以对应相同的 `handle`，也可自行规定有效 `handle` 的范围。

  如果同学们不是很理解上面的定义，可以理解为该函数把用户给的 `key` 映射到一个具体的锁的 ID。

- 申请（`void sys_mutex_acquire(int mutex_idx)`）：申请 `handle` 对应的互斥锁。如果此锁已被占有，就切换至其他进程继续等待。否则，占有这把锁。

- 释放（`void sys_mutex_release(int mutex_idx)`）：释放自己拥有的这把锁。

  并未规定是否可以释放别人占有的锁。一个合理的操作应当是无法释放，不过也可以认为这是编程者犯的错误，规定为允许释放，以简化内核编程压力。

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P2-3所示。

```
> [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.
```

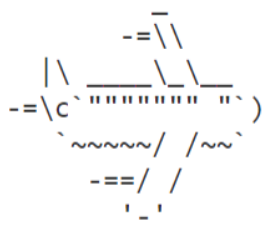

图 P2-3: P2-task2 参考运行结果

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. 实现 `tiny_libc/syscall.c` 中的锁的相关调用的跳转表简易版本。

4. 取消任务中对 `sys_yield` 系统调用的注释（若已被注释），并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 Project 4 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 (interrupt) 和异常 (exception) 统称为例外 (trap)。通俗点说，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，比如系统调用 `syscall`；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 [6]。

在这次实验中，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 要求实现定时器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 则实现了任务的抢占式调度，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 态用于操作系统，User 态用于普通的用户程序。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。本实验使用了这三个特权级。在本实验中，我们需要设置定时器，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。

> **Note 5.1** 随着 RISC-V 新拓展的添加，也可能会在新拓展中出现其他特权级，比如 H 拓展带来的 Hypervisor。

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[6]

读写 CSR 可以使用 `csrr`、`csrw`、`csrc`、`csrs` 等等指令进行，具体的指令可以参考 [7]，或者前面章节中给出的伪指令表。

> **Note 5.2** 这里应该一些应该大家自己关注的话题：RISC-V 的手册（再次强调，请自行阅读 RISC-V 指令集手册！同学们到如今一定能在网上找得到官方的 ISA 手册）中，Zicsr 扩展提供了与 CSR 相关的三条指令：`csrrw/csrrc/csrrs`。在 Project 0 中，我们也介绍过几条伪指令，如 `csrr/csrw`。这些伪指令后加上立即数或者 CSR 名称使用，都可以对指定的 CSR 进行操作。我们的实验框架在 `csr.h` 中定义了一些宏，并使用了这些宏作为 CSR 指令的相关目标。大家可以不用这些宏，而是直接使用对应 CSR 的名称进行汇编代码的编写。

5.2 例外处理流程

例外的触发

当发生异常时，处理器会将发生异常的地址放入 `sepc` 寄存器，然后跳转到 `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P2-6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，这也是低 2 位用作 `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，`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 `stvec_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` 寄存器的结构如图P2-4所示。当 `sie` 寄存器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` 寄存器的作用可以理解为，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` 寄存器和 `sstatus` 寄存器中的 `SIE` 位的区别。

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 P2-4: Supervisor interrupt-enable register (`sie`).[6]

`sstatus` 寄存器的结构如图P2-5所示。该寄存器中同样也有一个 `SIE` 位，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` 寄存器里面的 `SIE` 置为 0，将 `SPIE` 置为原来的 `SIE` 值。当执行 `sret` 时，硬件会将 `SPIE` 置为 1，`sstatus` 寄存器中的 `SIE` 置为原来的 `SPIE` 值。`sie` 寄存器不会变化。

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

17	16 … 15	14 … 13	12 … 9	8	7	6	5	4 … 2	1	0
WPRI	XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	UBE	SPIE	**WPRI**	SIE	**WPRI**
1	2	2	4	1	1	1	1	3	1	1

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

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

图 P2-6: Supervisor trap vector base address register (`stvec`).[6]

例外的处理

触发例外的情况有很多，中断就是例外的一种，中断又可以分为定时器中断、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，跳转到该例外对应的中断处理函数的入口。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P2-7所示。

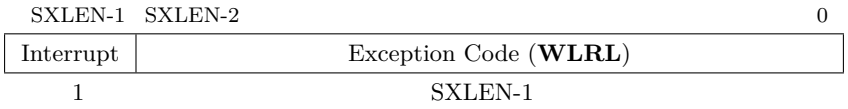

图 P2-7: Supervisor Cause register `scause`.[6]

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。本次实验中的定时器中断就是表中的 Supervisor timer interrupt；而系统调用就是表中的 Environment call from U-mode。

第三级：相应的中断处理函数负责具体地处理每个中断。例如定时器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` 寄存器里面的 `SIE` 位清 0，原先的 `SIE` 值被保存到 `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` 函数，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动重新打开中断。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` 值。而 `SPIE` 值是中断发生前 `SIE` 的状态，此时应该是开启状态。所以 `SPIE` 后，`SIE` 位自然是开启状态。

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 P2-3: Supervisor cause register (`scause`) values after trap.[6]

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` 寄存器所指向的地址，也就是发生异常前运行到的地址。这里提示一点：对于系统调用来说，需要大家返回 `sepc+4` 的地址位置。这是因为，`sepc` 寄存器默认指向的是触发系统调用的 `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 `sepc` 寄存器的值，使例外结束

后跳回 `sepc+4` 的位置。

5.3 任务 3S: 打印例外信息

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”。

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。

5.4 带有内核态保护的系统调用

在之前的任务中，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，但是，作为用户进程的任务是不应该被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。

系统调用中断也是例外的一种，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，因此我们还需要对内核的代码实现一步系统调用的封装，以提供给用户进程使用。

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：**start code 中 `tiny_libc` 中的功能是可以在用户态调用的，其余功能都是内核态的，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。这一点也通过内核与用户的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。

5.5 关于例外处理的相关说明

从 Project 2 的 task 3（A-core 及以上）开始，我们的操作系统就正式有了特权级之分。用户代码跑在 User mode，内核代码跑在 Supervisor mode，用户程序就不能直接通过跳转表去调用内核的函数了（这也是我们 task1、task2 一直在做的事情），而必须

通过硬件支持的 `syscall` 机制。而由于两个特权级的执行过程中，进程需要执行不同的程序，因此我们需要在进程切换到内核态之后，保存一次用户态现场。那么，无论是由于定时器中断还是 `syscall`，为了发生进程切换，用户进程就需要先陷入内核才能调用 `do_scheduler()`，那么完成进程切换就需要做两次保存、恢复现场的操作：

1. 进程从用户态陷入内核需要 save context（保存全部的通用寄存器和部分控制状态寄存器，暂且称为用户态上下文；
2. `do_scheduler()` 在内核态调用 `switch_to()` 做进程上下文切换还需要保存一次（保存 callee-saved 通用寄存器，暂且称为内核态上下文）。
3. 而后 `switch_to()` 恢复新进程的内核态上下文并返回。
4. 但此时（虽已切到新进程）仍然处于内核态，再执行 `ret_from_exception()` 恢复新进程的用户态上下文之后才回到新进程的用户态。

内核代码在主存里只有一份，但“内核态”并非所有进程公共的，每个用户进程都有在用户态和内核态的时候；而我们如果遵循框架中使用的多内核栈的设计，那每个用户进程在内核态便都有自己在内核态单独拥有的资源。概括来说，完成进程切换需要两次保存上下文，一次发生于用户态切到内核态，一次发生于从当前进程切到新进程。希望大家能仔细思考，分清这些基本概念和思想。

5.6 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` 等函数。
4. 使用所有给出的测试任务（syscall 版本），打印出正确结果（如图P2-8所示）。

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 内核栈初始化函数（`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 4 中进一步完善。

```
> [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).
```

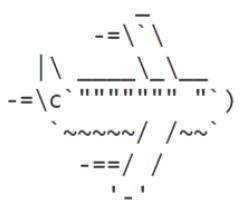

图 P2-8: P2-task3 参考运行结果

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`，其主要完成例外处理入口相关内容：保存现场、根据 `scause` 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_CONTEXT` 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` 寄存器所指向的地址。

7. 实现 `tiny_libc/syscall.c` 中的 `invoke_syscall`，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` 中的 `handle_syscall`，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` 中的 `do_sleep` 以及 `timer.c` 中的 `check_sleeping`，并在 `do_scheduler` 中唤醒所有可以唤醒的 PCB。

10. 取消任务中对 `sys_yield` 系统调用的注释（若已被注释），并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 态的 `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 User 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将其从睡眠队列（sleep queue）调整到就绪队列（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. 在例外发生之后、处理之前，我们需要保存通用寄存器现场至某处。保存的操作可能需要一个通用寄存器来辅助，请了解 `sscratch` 寄存器会对这件事产生什么帮助。

6. 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。

7. 在之前的实验中，我们一直使用 `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。

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

9. **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。

5.7 定时器中断

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，需要不断使用 `sys_yield` 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，我们需要使用定时器中断去打断正在运行的任务，并在定时器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。

定时器相关的寄存器都在 Machine 级，Supervisor 级无法直接控制相关的寄存器，需要使用 `set_timer` 设置定时器的触发，设置的内容为下次触发定时器中断的时钟数。因此，设置之前需要读取当前时间，计算下次定时器中断的时间后再设置进去。

同时，为了允许定时器中断，各位同学需要使能定时器中断，这一点详见 RISC-V 特权级手册以及任务书前面的描述。

5.8 任务 4：定时器中断、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. 实现定时器中断处理逻辑，并基于定时器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 3 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，以处理定时器中断。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` 中对定时器中断进行初始化，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. 关于如何正确地开始一个任务的第一次调度，在抢占调度模式下是和非抢占调度模式下不同的，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

本实验的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。

5.9 任务 5：复杂调度算法

Project 2 的 A-core 需要完成的任务到任务 4 为止，任务 5 为 C-core 需要完成的任务：复杂调度算法。

实验要求

1. 在测试程序中，我们提供了 5 个飞行速度不同的 `fly` 测试程序。从程序中可以看到，每个 `fly` 程序的 `CYCLE_PER_MOVE` 值不相等，因此在每次循环中，飞机前进一格所花的时间不相同。那么在操作系统使用公平的调度算法时，五个 `fly` 程序的飞行进度就会不相同。

2. 在本任务中，每个 `fly` 的飞行路径上都设有一个检查点，要求五个 `fly` 程序在同时启动的情况下能够同时经过各自的检查点，再同时到达终点。这就要求操作系统在不知道每个用户程序具体的飞行速度时，能根据五个 `fly` 程序的实时飞行位置，去调整进程的优先级，从而让进度较慢的进程能拥有更大或更多的时间片。

3. 这样的调度算法需要同学们自己去实现。另外，每个 `fly` 程序需要通过系统调用 `set_sche_workload` 向操作系统报告当前位置，这个系统调用也需要同学们根据测试程序自己去实现。

要点解读

1. 我们希望同学们实现动态调整飞机速度的算法,因此调度算法不应简单的按照 `fly` 程序的飞行速度换算时间片比例，而应当根据每次程序报告的位置进行动态调整。

2. 在呈现效果上，应该保证视觉上每个飞机都处于飞行状态，不能出现：飞机飞一段距离停下，等待其他飞机飞完再飞的视觉效果。也就是说，在动态调整时间片大小的过程中，应避免某个进程被分到的时间片为 0，导致飞机停下的场面。

Project 2 功能总结

在下表，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，注意随着评分等级升高，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，定时器中断处理，抢占式调度
C-core	实现复杂调度算法

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 start code 里实现了驱动（位于 `drivers/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 I/O，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P2-9所示。

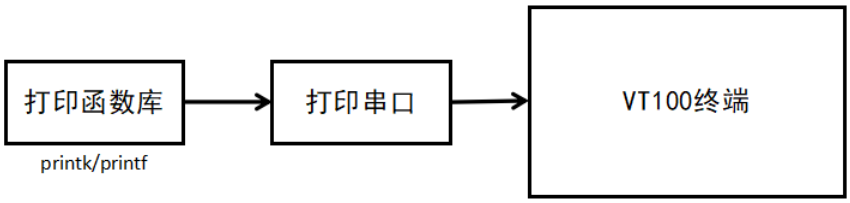

图 P2-9: 打印流程

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地进行输出，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，然后每次输出时往模拟的显存里写数据（`screen_write` 方法），在定时器中断时再在处理函数（`irq_timer`）里去一次性地

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，在多任务模式下不会造成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，我们的做法依旧还是直接往串口写数据。如图P2-10所示。

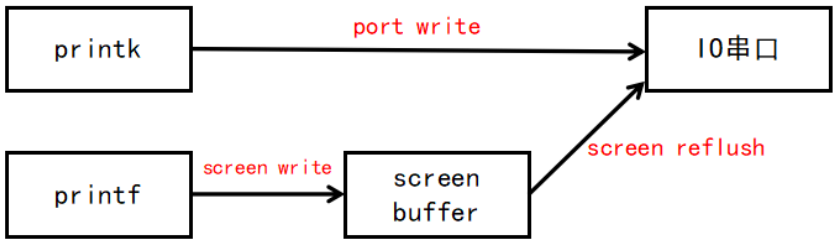

图 P2-10: 开发板打印过程

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有变化的字符，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 Project 2 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定时器中断或者系统调用这样的例外才能进入 Supervisor 态，并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 Project 4 的内容)

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，所以请大家在最终检查之前也使用 `loadbootd` 来自己验证一遍。

6.3 内核调试方法——printl

在 Project 2 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。这样，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 1、2 中将 printl 放入跳转表（参考 printk），也可以在任务 3、4 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能够进一步提高。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，方便未来的学弟学妹们。

另外，希望大家让自己的程序多打印出日志文件 (log)，希望大家多使用这个函数。优秀的日志能够对开发工作起到相当大的帮助。日志对定位错误而言，也是有直接使用 gdb 进行 debug 所不能及之特长的。

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-2019].

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

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

[7] A. B. Palmer Dabbelt, Michael Clark, "Risc-v assembly programmer's manual." https://github.com/riscv-non-isa/riscv-asm-manual/, 2024. [Online; accessed 01-October-2024].