国科大操作系统研讨课任务书

Project 2

（Part I）

版本：4.0

一、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，使其具备**任务调度**的功能。

通过本次实验，你将学习操作系统任务调度等知识，掌握任务的阻塞和唤醒、线程锁的实现。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 MIPS 架构的相关知识。本次实验的内容如下：

（1）任务一：了解操作系统中任务的本质，实现进程控制块、进程切换（现场保存、现场恢复）、非抢占式调度。

（2）任务二：了解操作系统中任务的各种状态以及转化方式，实现任务的阻塞、任务的唤醒、互斥锁。

从本次实验开始，我们将提供给大家一些基础的代码框架，里面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。换而言之，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。

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。

二、进程

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任务的调度方面，从这个角度而言，进程和线程是一样的。因此大家可以在这里暂时的认为进程和线程是一样的，没有什么差别。

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**内核线程**。但是想开启另一个新的用户进程，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。

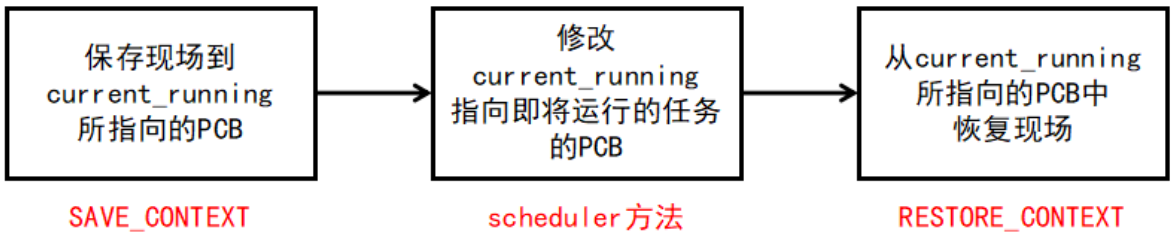
2.1 进程控制块

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，实现和设计 PCB 的数据结构及相关功能。

2.2 任务的切换

拥有了 PCB 之后，我们就可以利用它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**。

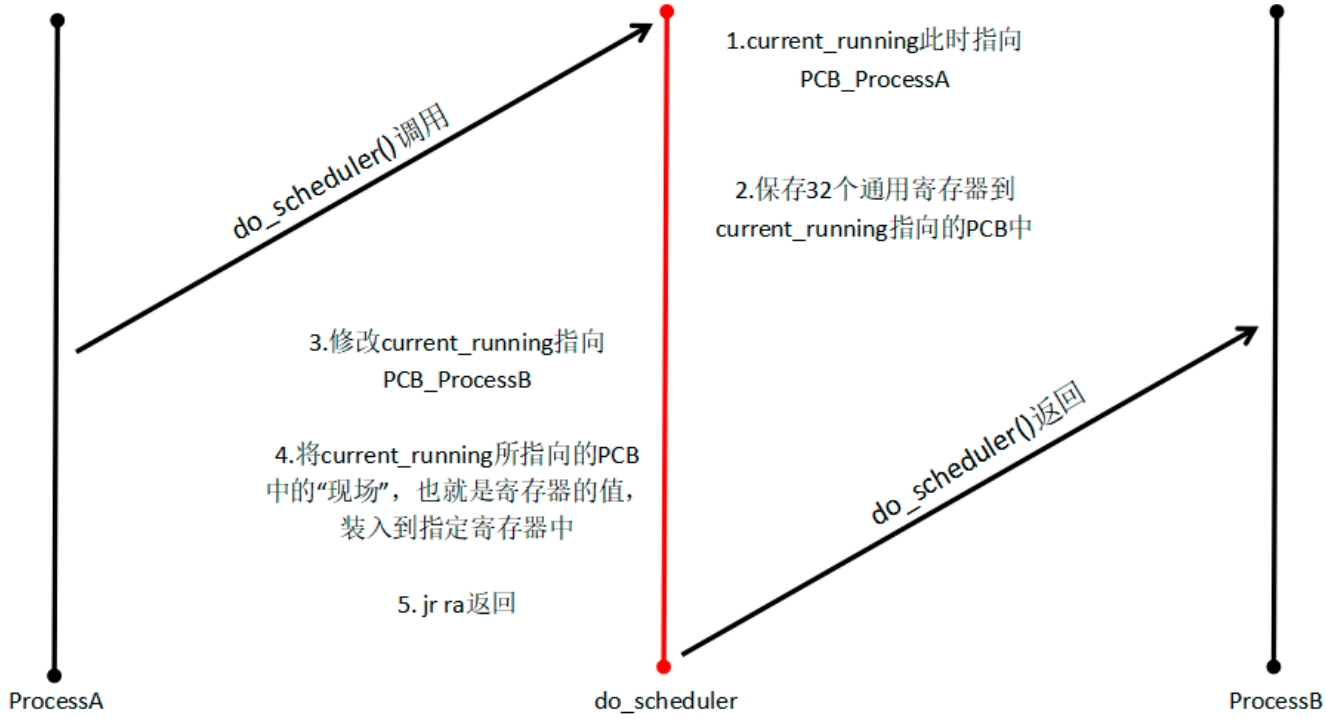
关于任务切换，我们实际上只需要一个全局的 PCB 指针 current_running，它指向哪个 PCB，说明那个 PCB 对应的任务为正在运行的任务，在进行保存和恢复的时候只对这个 current_running 对应 PCB 进行保存和恢复。具体流程如下图所示：



current_running 指针和任务切换的关系

关于调度的触发方式，在本次实验需要大家去实现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**。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，因此我们首先实现它。抢占式调度涉及到时钟中断处理等操作，我们将在 Project2 的第二部分中进行实现。

在这里，我们以非抢占式调度为例，给出一个较为完整的任务切换流程图：

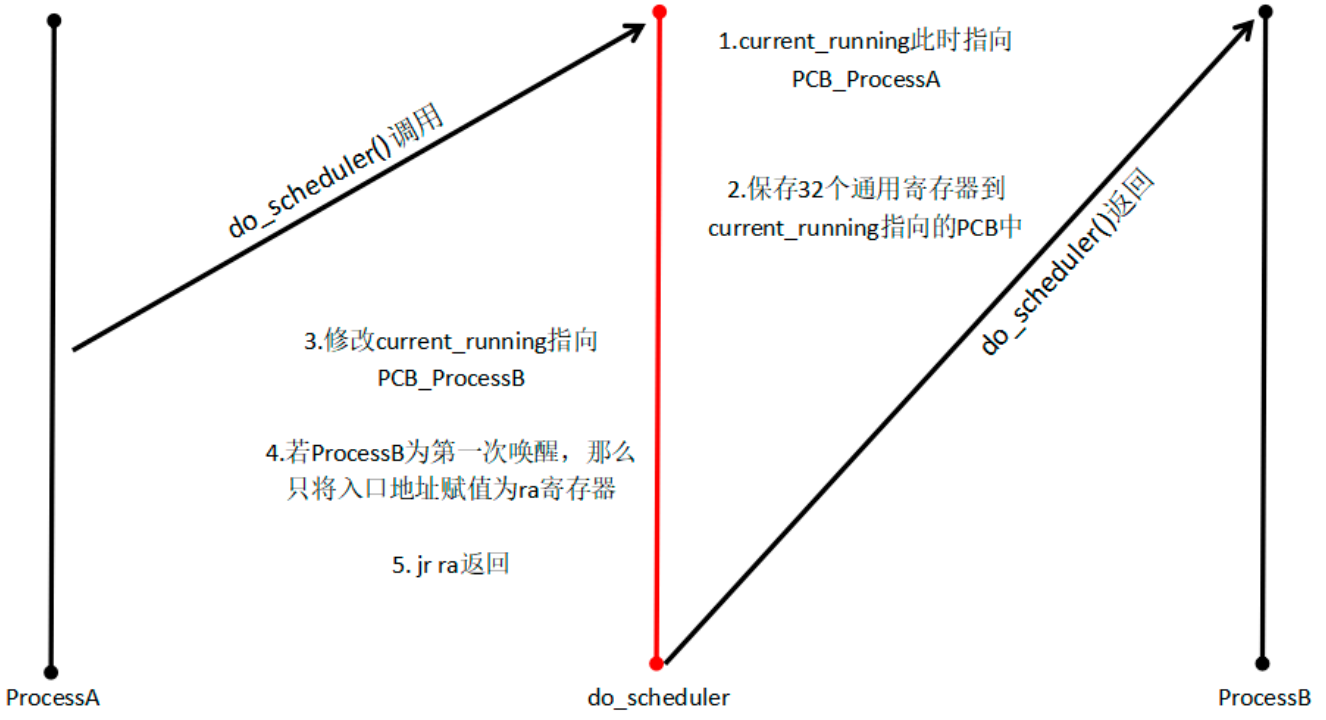


2.3 PCB 初始化

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。

关于如何如何完成这个过程，我们给予大家的思路是保存一个进程的入口地址，在进程第一次被唤醒时将入口地址赋值到 ra 寄存器，然后在进程切换函数的最后使用 jr ra 指令进行返回。

对于一个第一次运行的任务，它的调度流程图如下：



2.4 任务 1：任务启动与非抢占式调度

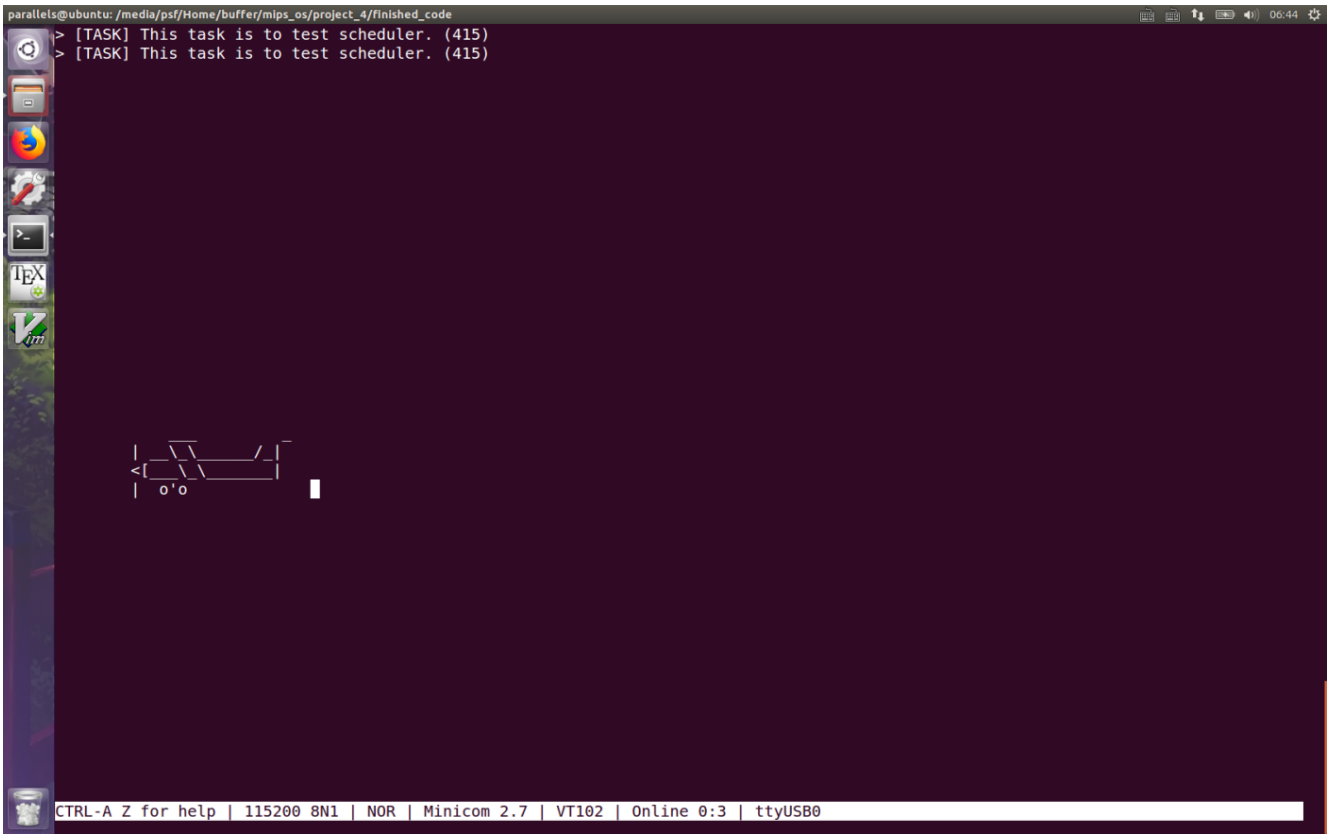
2.4.1 实验要求

（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）实现非抢占式调度方法。

（4）运行给定的测试任务，能正确输出结果，如下图：



任务执行成功图

2.4.2 文件说明

编号	文件/文件夹	说明
1	arch/mips	• MIPS 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**任务 4** 补全的部分 • pmon/common.c：MIPS 的 BIOS 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	• irq/irq.c：中断处理相关，本次实验**任务 3** 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**1,2,3，4 补全** • sched/queue.c：队列操作库，不需要修改 • sched/time.c：时间相关函数，**任务 3** 补全 • syscall/syscall.c：系统调用相关实现，**任务 4** 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，其中用户级的 printf 函数需要在实现系统调用后补全 sys_write 方法才可以使用
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	ld.script 文件	• 链接器脚本，不需要修改

2.4.3 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，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。

（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 printk_task1、printk_task2 任务在屏幕上方交替的打印字符串“*This task is to test scheduler*”，drawing_task1 任务在屏幕上画出一个飞机，并不断移动。

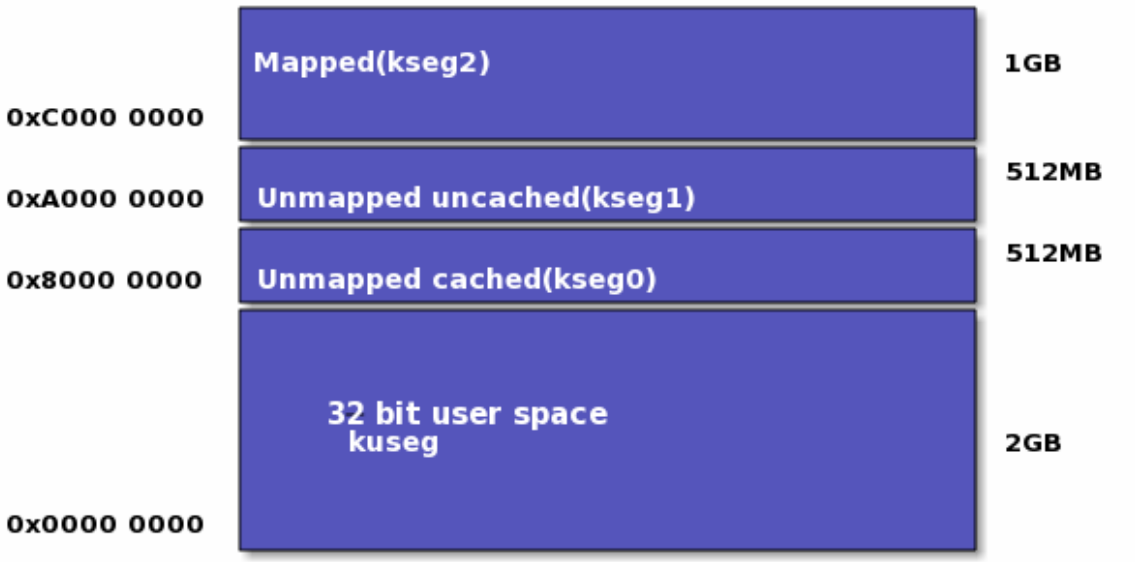
2.4.4 注意事项

（1）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。

（2）在现场返回时请考虑使用什么指令完成，具体可以了解 ra 寄存器以及 jr 指令的作用。

（3）每个任务在初始化的时候需要为其分配栈空间，请同学们自行思考栈的分配实现，对于栈的基地址，同学们可以自行决定，**我们推荐给大家的栈基址为 0xa0f00000**。

（4）在 MIPS 下，内存空间布局如下：



32 位 MIPS 下虚拟内存空间

Kuseg：0×00000000～0×7FFFFFFF（2G）。这些地址是用户态可用的地址。 在有 MMU 的机器里，这些地址将一概被转换。除非 MMU 已经设置好，否则不应该使用这些地址；对于没有 MMU 的处理器，这些地址的行为与具体处理器有关。 如果想要你的代码能够移植到无 MMU 的处理器上，或者能够在无 MMU 的处理器间移植，应避免使用这块区域。

Kseg0：0×80000000～0×9FFFFFFF（512M）。只要把最高位清零，这些地址就会转换成物理地址，映射到连续的 低端 512M 的物理空间上。这段区域的地址几乎总是要通过高速缓存来存取（write-back，或 write-through 模式），所以在高速缓存初 始化之前，不能使用。因为这种转换极为简单，且通过 Cache 访问，因而常称为这段地址为 Unmapped Cached。这个区域在无 MMU 的系统中用来存放大多数程序和数据；在有 MMU 的系统中用来存放操作系统核心。

KSeg1：0xA0000000～0xBFFFFFFF（512M）。这些地址通过把最高三位清零，映射到连续的 512M 的物理地址，即 Kseg1 和 Kseg0 对应同一片物理内存。但是与通过 Kseg0 的访问不同，通过 Kseg1 访问的话，不经过高速缓存。 Kseg1 是唯一的在系统加电/复位时，能正常访问的地址空间，这也是为什么复位入口点（0xBFC00000）放在这个区域的原因，即对应复位物理地址 为 0×1FC00000。因而，一般情况下利用该区域存储 Bootlooder；大多数人也把该区域用作 IO 寄存器（这样可以保证访问时，是直接访问 IO，而不是访问 cache 中内容），因而建议 IO 相关内容也要放在物理地址的 512M 空间内。

KSeg2：0xC0000000～0xFFFFFFFF（1G）。这块区域只能在核心态下使用，并且要经过 MMU 的转换，因而在 MMU 设置好之前，不要存取该区域。除非你在写一个真正的操作系统，否则没有理由使用 Kseg2。

MIPS CPU 运行时有三种状态：用户模式（User Mode）；核心模式（Kernel Mode）；管理模式（Supervisor Mode）。其中管理模式不常用。用户模式下，CPU 只能访问 KUSeg；当需要访问 KSeg0、Kseg1 和 Kseg2 时，必须使用核心模式或管理模式。

在这个 project 中，我们推荐大家使用 KSeg1 的地址，与 Project1 使用的空间一样。但是请大家注意的是 Kseg0 与 KSeg1 之间的关系，避免与 BIOS 所在的 KSeg0 发生冲突。

三、线程锁

当两个线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。可能出现第一个线程修改了一半后，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**。

3.1 线程的状态

线程从创建、运行到结束总是处于下面五个状态之一：新建状态、就绪状态、运行状态、阻塞状态及死亡状态。

3.1.1 新建状态（create）

新创建一个线程。此时程序还没有开始运行线程中的代码。一个新创建的线程并不自动开始运行，要执行线程，必须给线程运行分配系统资源，并调度线程运行。

3.1.2 就绪状态（ready）

处于就绪状态的线程并不会立刻运行，线程还必须同其他线程竞争 CPU 时间，只有获得 CPU 时间才可以运行线程。因为在单 CPU 的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程，哪一个线程下一个开始运行是由系统的线程调度方法来控制的。

3.1.3 运行状态（running）

当线程获得 CPU 时间后，它才进入运行状态，真正开始执行。对于单核处理器，从微观的角度而言，同一时间只有一个线程处于运行状态。

3.1.4 阻塞状态（blocked）

所谓阻塞状态是正在运行的线程由于某种原因，暂时让出 CPU，这时其他处于就绪状态的线程就可以获得 CPU 时间，进入运行状态。线程运行过程中，可能由于各种原因进入阻塞状态，这个线程会从就绪队列（ready queue）进入到阻塞队列（block queue），阻塞队列里的任务是不会被调度器调度出来运行的，只有到达某一条件，这个任务才会被从阻塞队列取出，重新放回就绪队列（至于是放在就绪队列头还是就绪队列尾，取决于实验中设定的线程调度算法），从而继续运行。下面是几种会造成任务阻塞的情况，在以后的实验中我们会一一实现：

（1）**方法**造成的阻塞：线程通过主动调用 sleep 方法进入睡眠状态，当睡眠时间结束后，线程会被调度器重新加入到就绪队列。

（2）**锁**造成的阻塞：线程试图得到一个锁，而该锁正被其他线程持有，那么这个线程就会被阻塞；当持有该锁的线程释放锁时，会主动的将因为该锁而被阻塞的线程放回就绪队列。

（3）**同步原语**造成的阻塞：关于该部分会在之后的实验详细说明，如果有兴趣的同学可以自行查阅相关资料。

3.1.5 终止状态（termination）

终止状态指的是一个线程的生命周期结束，需要注意的是，线程在终止时，需要释放其占用的资源，比如 PCB、锁、栈空间等。以下为几个造成线程终止的原因，这些原因我们在以后的实验也会一一实现：

（1）线程正常退出而自然终止，比如使用 exit 方法进行退出。

（2）线程被杀死，比如被 kill 方法杀死。

3.2 自旋锁

自旋锁的实现很简单，就是设置一个变量，当一个线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临界区，并且此时没有其他冲突线程在访问这个变量，那么该线程就进入临界区，并且将这个变量置为有线程访

问状态。如果这个变量状态为有其他冲突线程在访问，那么就不断的使用 while 循环重试，直到可以进入临界区。

因此可以看出来，使用自旋锁的坏处就是如果一旦获取锁不成功，那么线程就会一直循环尝试获取锁，这极大的浪费了 CPU 资源。因此就出现了互斥锁的实现方法。

3.3 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线程请求锁失败，那么该线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。直到占用该锁的线程释放锁之后，被阻塞的线程会被占用锁的线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。

3.4 任务 2：线程间互斥锁的实现

3.4.1 实验要求

了解操作系统内的**任务调度机制**，学习和掌握**自旋锁、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，要求多个线程同时访问同一个锁的时候，后访问的线程被挂起到阻塞队列。第一个线程释放该锁后，第二个线程才被唤醒，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，如下图：

任务执行成功图

3.4.2 文件介绍

请基于任务 1 的项目代码继续进行实现。

3.4.3 实验步骤

（1）完成 sched.c 中的 do_unblock 方法、do_block 方法，要求其完成对线程的**挂起**和**解除挂起**操作。

（2）实现互斥锁的操作（位于 lock.c 中）：锁的释放（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。

3.4.4 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）请思考一个线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：是否支持一个线程获取多把锁，是否支持两个以上线程同时请求锁并被阻塞。