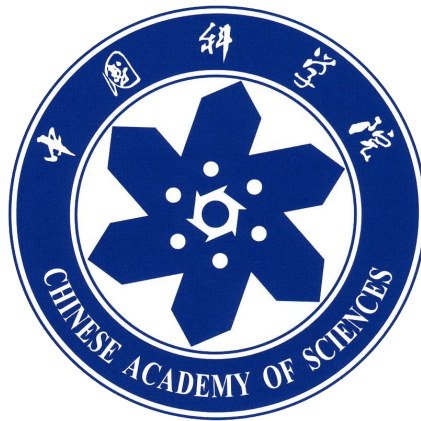


国科大操作系统研讨课任务书

Project 2



版本：3.0

一、实验说明.....	3
二、进程.....	4
2.1 进程控制块.....	4
2.2 任务的切换.....	4
2.3 PCB 初始化.....	5
2.4 任务 1：任务启动与非抢占式调度.....	6
三、线程锁.....	9
3.1 线程的状态.....	9
3.2 自旋锁.....	9
3.3 互斥锁.....	10
3.4 任务 2：线程间互斥锁的实现.....	10
四、例外处理.....	12
4.1 例外处理流程.....	12
4.2 CPO 控制寄存器.....	15
4.3 时钟中断.....	15
4.4 任务 3：时钟中断、基于优先级的抢占式调度.....	15
4.5 系统调用.....	17
4.6 任务 4：系统调用.....	18
五、打印函数.....	20
5.1 VT100 控制码.....	20
5.2 屏幕模拟驱动.....	20
5.3 打印优化.....	21

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润色，使其具备**任务调度**、**例外处理**等功能。

通过本次实验，你将学习操作系统任务调度、例外处理等知识，掌握任务的阻塞和唤醒、线程锁的实现。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 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。
- (3) **任务三**：了解 MIPS 下例外处理的流程，实现操作系统中例外处理、时钟中断处理、抢占式调度、计数函数。
- (4) **任务四**：了解 MIPS 下系统调用处理流程，实现操作系统中系统调用、sleep 系统调用方法。

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。

在进入到了 `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**。

2.2 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**。

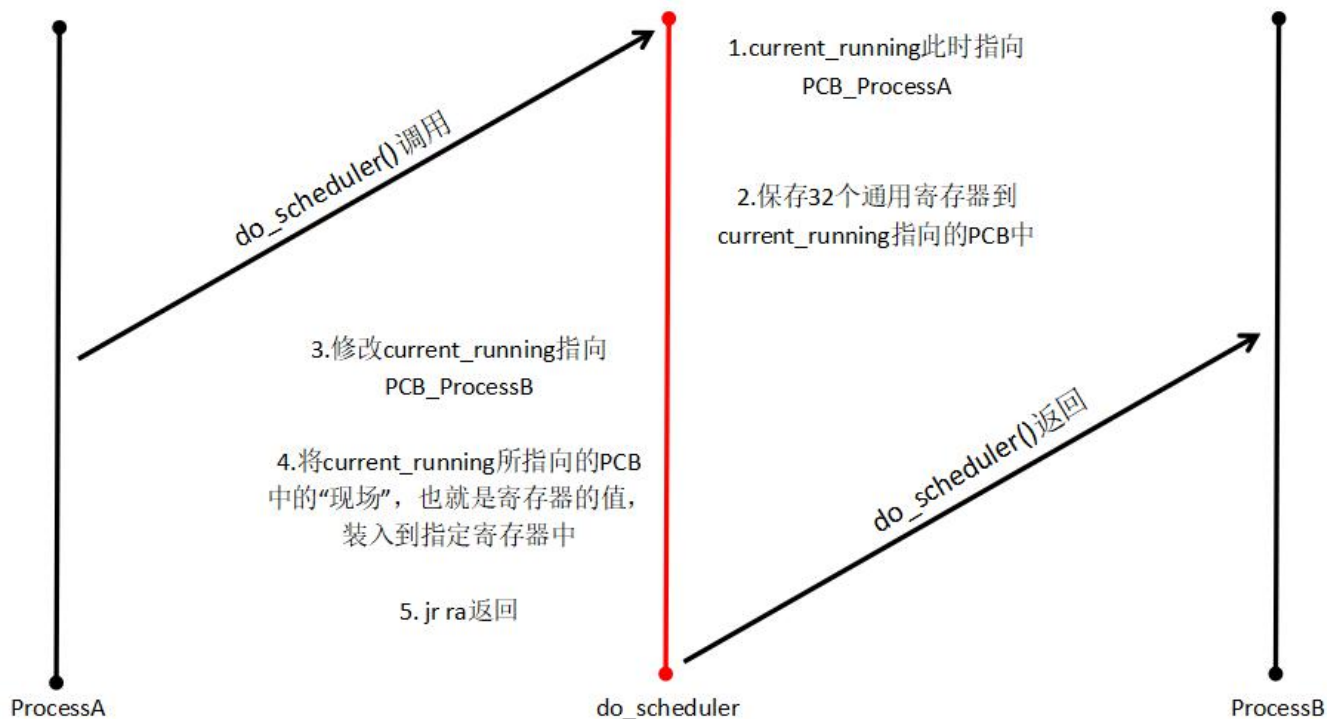
关于任务切换，我们实际上只需要一个全局的 **PCB** 指针 `current_running`，它指向哪个 **PCB**，说明那个 **PCB** 对应的任务为正在运行的任务，在进行保存和恢复的时候只对这个 `current_running` 对应 **PCB** 进行保存和恢复。具体流程如下图所示：



`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“被强迫”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，因此我们首先实现；抢占式调度涉及到时钟中断处理等操作，我们在后面任务中进行实现。

在这里以非抢占式调度为例，我们给出一个较为完整的任务切换流程图：

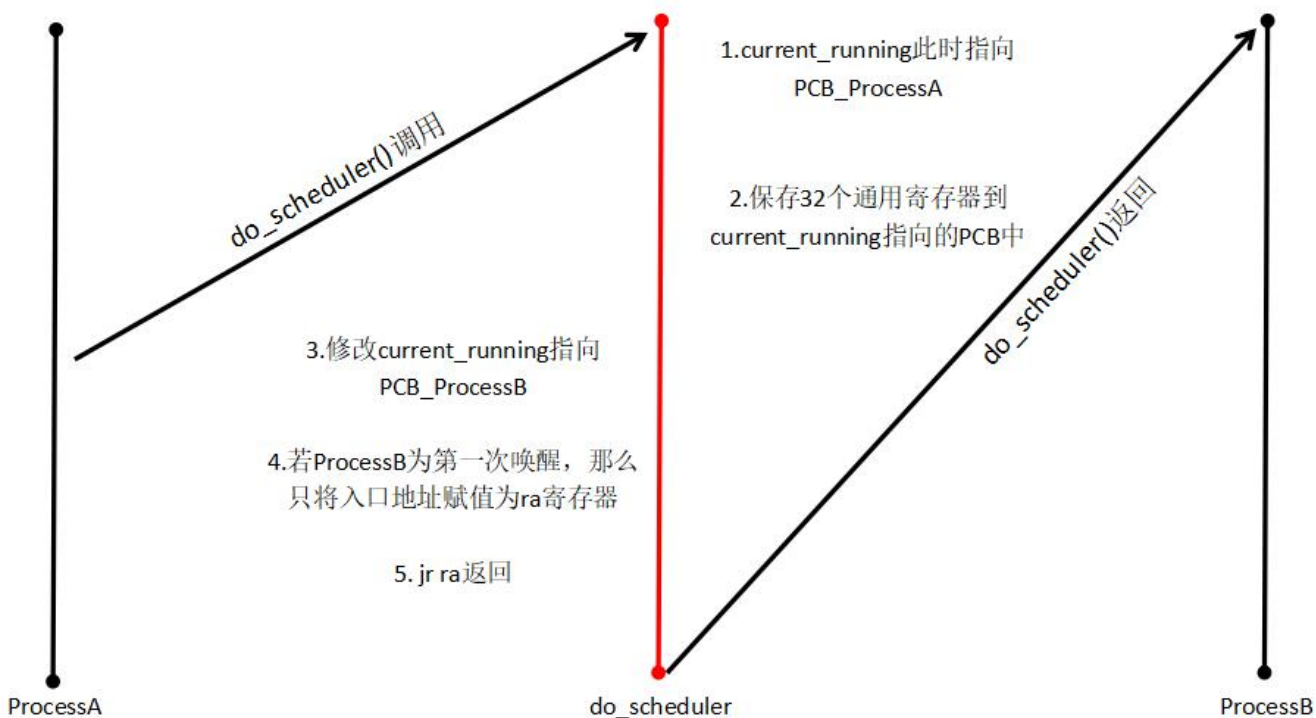


2.3 PCB 初始化

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。

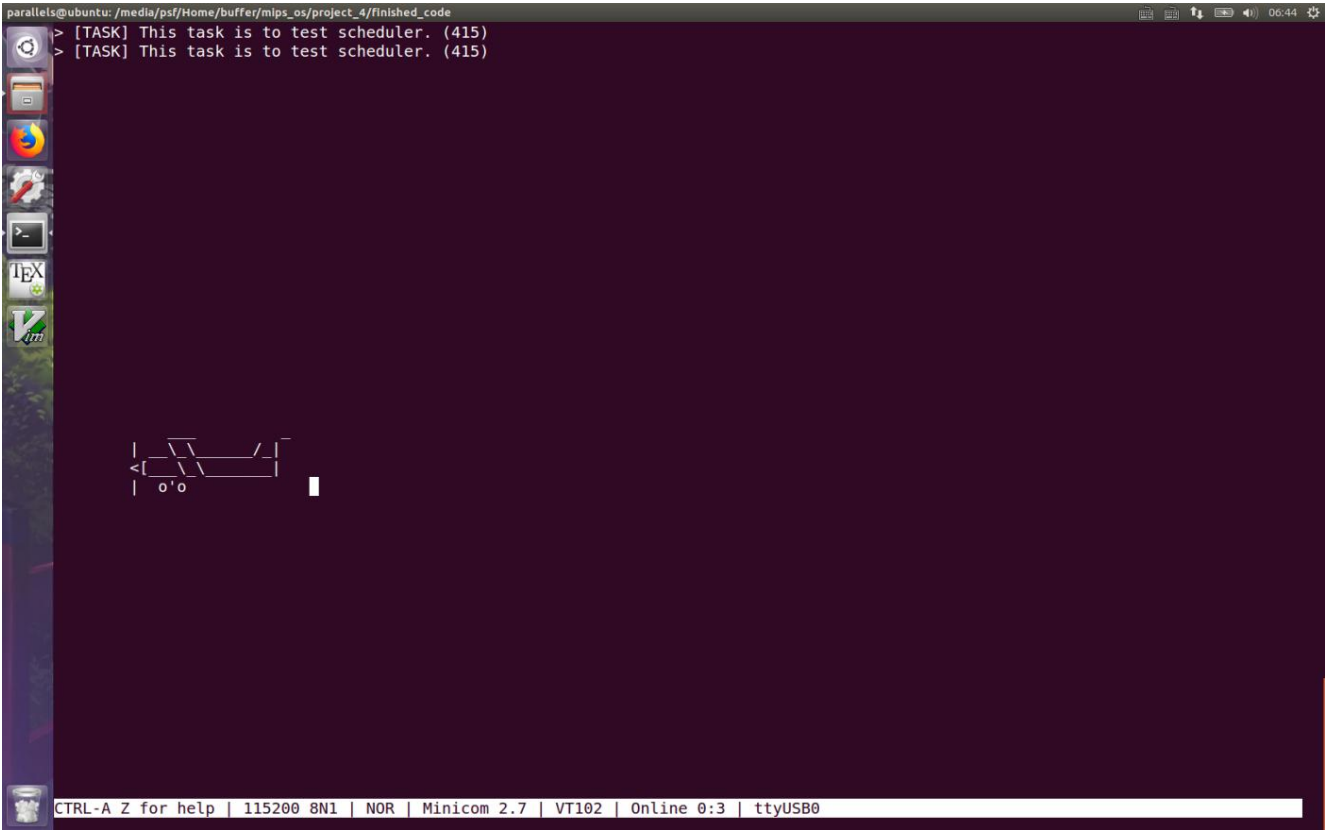
对于一个第一次运行的任务，它的调度流程图如下：



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	<div><div>• MIPS 架构相关内容，主要为汇编代码以及相关宏定义</div><div>• boot/bootload.S: 引导代码，请使用 P1 实现的代码</div><div>• include: 头文件，包含一些宏定义，不需要修改</div><div>• 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</div><div>• 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</div><div>• pmon/common.c: MIPS 的 BIOS 调用以及功能寄存器的访问，不需要修改</div></div>
2	drivers 文件夹	<div><div>• 驱动相关代码</div><div>• screen.c: 打印相关函数，不需要修改(具体介绍请看第五节内容)</div></div>

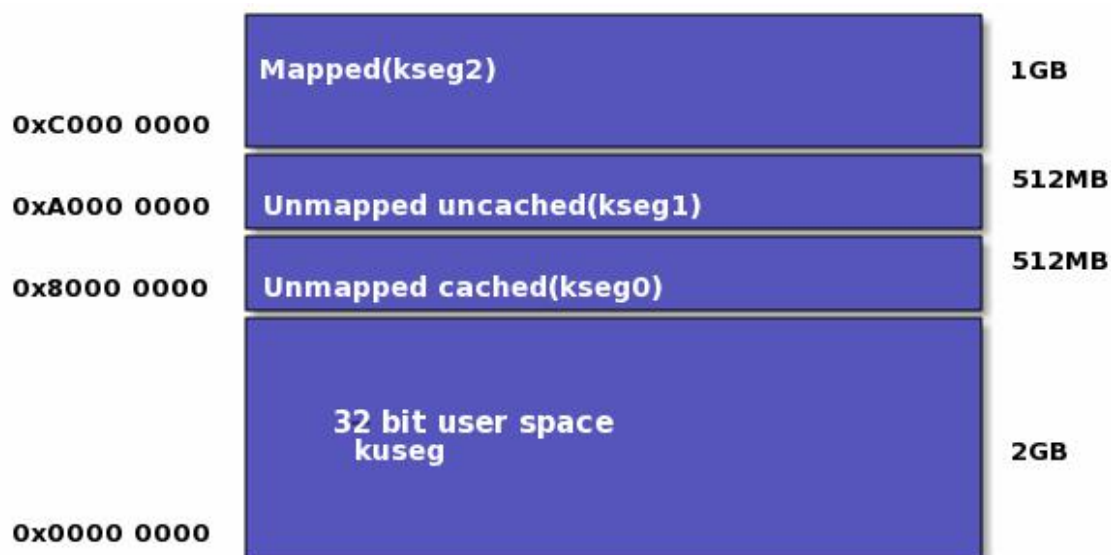
3	include 文件夹	• 头文件
4	init 文件夹	• 初始化相关 • init/main.c: 内核的入口, 操作系统的起点
5	kernel 文件夹	• irq/irq.c: 中断处理相关, 本次实验 任务 3 补全 • locking/lock.c: 锁的实现, 本次实验 任务 2 补全 • sched/sched.c: 任务的调度相关, 一个任务的调度、挂起、唤醒等逻辑主要在这个文件夹下实现, 本次实验 任务 1, 2, 3, 4 补全 • sched/queue.c: 队列操作库, 不需要修改 • sched/time.c: 时间相关函数, 任务 3 补全 • syscall/syscall.c: 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: 0x00000000~0x7FFFFFFF (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: 0x80000000~0x9FFFFFFF (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 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 0x1FC00000。因而，一般情况下利用该区域存储 Bootloader；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。

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：



```
parallels@ubuntu: /media/paf/Home/buffer/mlps_os/project_4/finished_code
> [TASK] This task is to test scheduler. (5)
> [TASK] This task is to test scheduler. (5)
> [TASK] Has acquired lock and running.(3)
> [TASK] Applying for a lock.
```

Diagram of a lock: A rectangle with a diagonal line from the top-left to the bottom-right. The top-left corner is labeled '1' and the bottom-right corner is labeled '0'. Below the rectangle is the text '0 0'.

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | ttyUSB0

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。

3.4.5 附加题

在设计完成锁之后，请考虑设计的合理性以及拓展性，要求：（1）实现后支持一个线程获取多把锁。（2）要求多个线程（两个以上）在请求一个锁导致被阻塞后，唤醒的时候**根据其线程优先级**依次被唤醒。

提示：每个锁应该维护一个阻塞队列，请求这个锁不成功的任务都需要被挂到这个队列中。

四、例外处理

在 MIPS 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。

在这次实验大家需要掌握和实现 MIPS 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。

4.1 例外处理流程

4.1.1 例外的触发

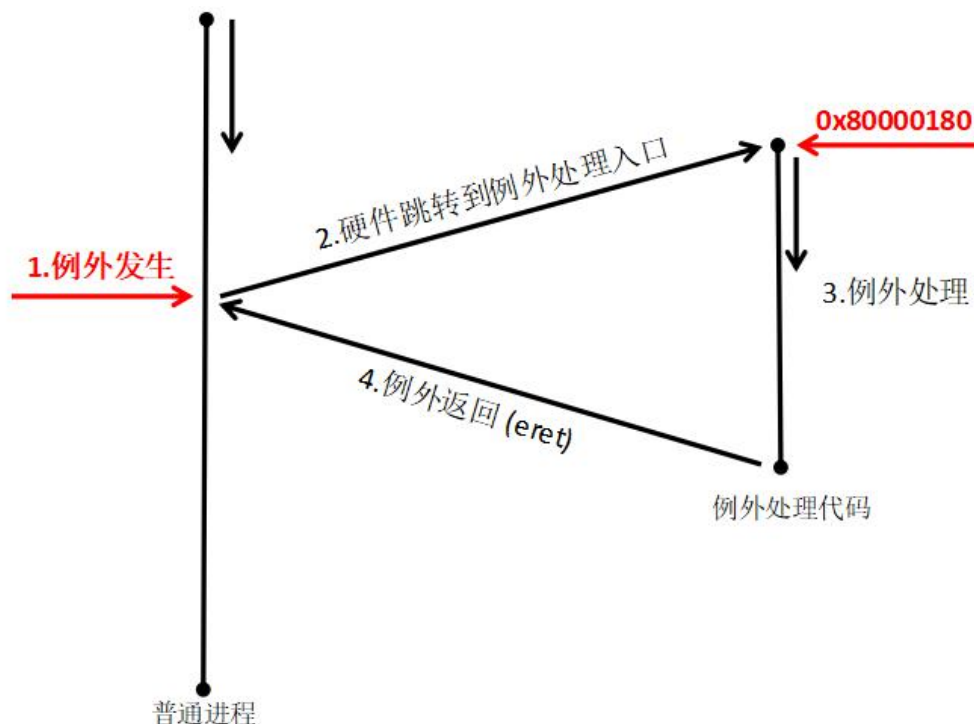
当发生异常时，处理器执行地址会将发生异常的地址放入 `CPO_EPC` 寄存器，然后自动跳转到一个固定的例外处理入口（这个过程是硬件自动完成的），例外处理入口如下表：

BEV 位	例外类型	例外向量地址
	Cold Reset/ NMI	0xFFFFFFFF BFC00000
BEV = 0	TLB Refill (EXL=0)	0xFFFFFFFF 80000000
	XTLB Refill (EXL=0)	0xFFFFFFFF 80000000
	Others	0xFFFFFFFF 80000180
BEV = 1	TLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	XTLB Refill (EXL=0)	0xFFFFFFFF BFC00200
	Others	0xFFFFFFFF BFC00380

例外入口地址

启动时(Boot-Time)例外向量(状态寄存器中的 BEV 位=1)地址位于既不通过 Cache 进行存取，也无需地址映射的地址空间。在正常操作期间(BEV 位=0)，普通例外的向量地址位于需要通过 Cache 进行存取的地址空间。

对于我们本次的任务而言，产生一个中断处理器就会跳转到 `0x80000180` 这个位置，需要一提的是，在发生例外时，跳转到例外处理入口是由硬件做的，但是例外的处理需要软件完成，如下图所示：



跳转到这里后，需要执行这个地方的例外处理代码，这些代码是需要我们自己写，并在内核初始化的时候将例外处理代码拷贝到这个位置。关于更多例外向量位置的信息请参考《龙芯 2F 处理器用户手册》第六章——处理器例外。

4.1.2 龙芯 1c 开发板的 4 级中断例外处理

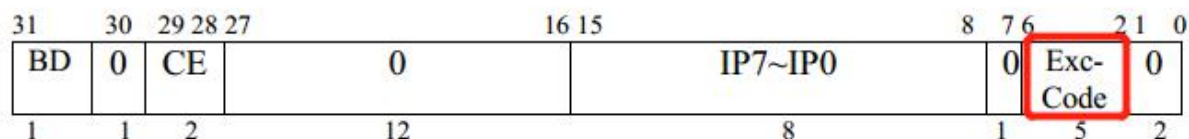
触发例外的情况有很多，中断就是例外的一种，中断又可以分为时钟中断、设备中断等，而设备中断又可以分为键盘中断、串口中断等。那么如何在发生一个例外后，准确的判断例外发生的原因，最后跳转到负责处理该例外的代码去执行呢？

其实在 MIPS 下，以中断处理为例，我们将中断例外的处理分为四级，每一级的处理过程如下：

第一级：各种情况下例外的总入口，即上小节提到的一个固定例外处理的入口，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 MIPS 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，这部分代码也是需要我们拷贝到例外向量地址部分的代码（具体见 4.1.1 节）。

对于例外种类的确定，是通过 CP0 的 cause 寄存器中 ExcCode 位域来区分不同例外的入口，CP0 CAUSE 寄存器的结构如下：



CP0_CAUSE 寄存器结构

以下为不同 Exccode 对应的例外类型，可以看到，当 ExcCode 为 0 的时候，说明触发的例外类型为中断(INT)：

例外代码	Mnemonic	描述
0	INT	中断
1	MOD	TLB 修改例外
2	TLBL	TLB 例外（读或者取指令）
3	TLBS	TLB 例外（存储）
4	ADEL	地址错误例外（读或者取指令）
5	ADES	地址错误例外（存储）
6	IBE	总线错误例外（取指令）
7	DBE	总线错误例外（数据引用：读或存储）
8	SYS	系统调用例外
9	BP	断点例外
10	RI	保留指令例外
11	CPU	协处理器不可用例外
12	OV	算术溢出例外
13	TR	陷阱例外
14	-	保留
15	FPE	浮点例外
16—22	-	保留
23	WATCH	WATCH 例外
24—30	-	保留
31	-	保留

Exccode 字段对应的例外种类

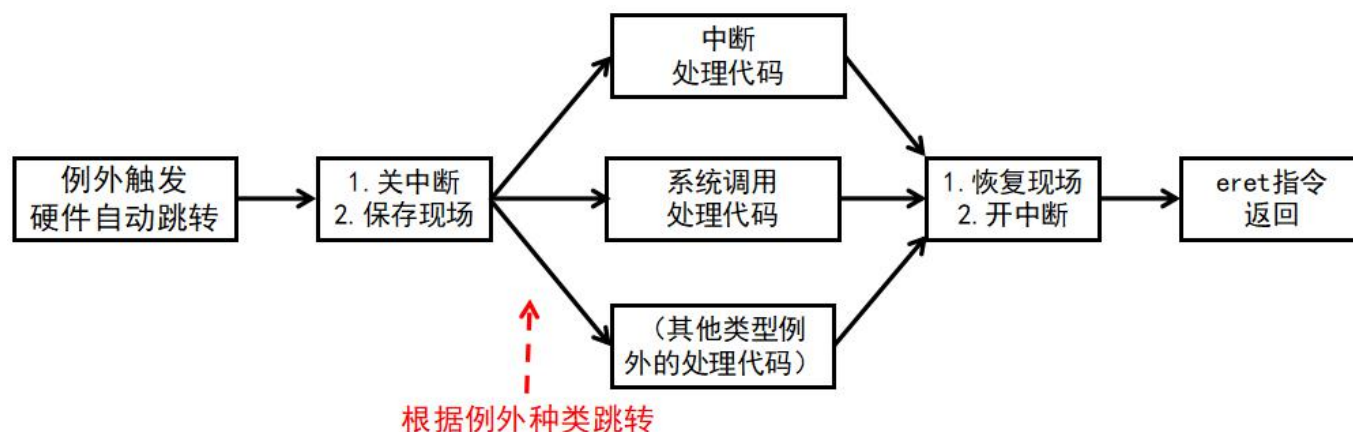
第三级：不同中断的各个入口，根据 IP7~IP0 的 8 位域（CP0_STATUS 寄存器中的）来区分，其中，时钟中断是 IP7=1，IP6~IP0 为 0，时钟中断只需要判断到第三级，外设中断需要判断到第四级才能确定是哪种外设引起的中断；

第四级：每个外设中断的入口（龙芯 1c 将所有外设分为五组，对应 5 组中断寄存器组，查看 5 组寄存器中各个 SR 寄存器中哪个位域是 1，就对应是哪一种外设产生的中断），具体可以参考《龙芯 1C300 处理器数据手册 v1.3》5.3 中断配置寄存器。

通过这四级例外的查找，我们最终可以知道到底是哪种情况触发的中断，并对不同的中断进行处理。这四级例外除了第一级的例外处理入口的跳转是由硬件完成的，其余几级都是由软件编程完成的，是本次实验需要同学们完成的。

4.1.3 例外的处理

在处理例外处理前我们需要通过修改 CP0_STATUS 寄存器的值关中断，然后保存发生例外时的现场；对于例外的处理，针对不同的例外需要具体实现，触发例外的原因会以保存在 CP0_CAUSE 寄存器中，在处理时根据例外触发代码跳转到不同的例外处理位置即可；在处理例外后对保存的现场进行还原，然后开中断，最后进行例外的返回。具体处理流程如下图：



例外处理流程

4.1.4 例外的返回

例外的返回是通过 `eret` 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 `CP0_EPC` 寄存器，之后跳到例外处理入口。当例外处理结束后，使用 `eret` 指令就可以返回 `CP0_EPC` 所指向的地址，也就是发生异常前运行到地址。

4.2 CP0 控制寄存器

MIPS 架构里除了 32 个通用寄存器以外还设计了 32 个 CP0 寄存器，用于处理例外，它们的功能各不相同，以下几个为本次实验会用到的 CP0 寄存器，如果同学们想了解更多，请详细参考[《龙芯 2F 处理器用户手册》第五章——CP0 控制寄存器](#)，里面对每个 CP0 寄存器有详细的说明。

4.3 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 `do_scheduler` 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。

注：时钟中断的触发涉及 `CP0_COUNT`、`CP0_COMPARE` 寄存器，`CP0_COUNT` 寄存器的值每个时钟周期会自动增加，当 `CP0_COUNT` 和 `CP0_COMPARE` 寄存器的值相等时会触发一个时钟中断。

4.4 任务 3：时钟中断、基于优先级的抢占式调度

4.5.1 实验要求

- (1) 掌握 MIPS 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询式抢占式中断。
- (3) 在简单的调度方法上改进，实现基于优先级的抢占式调度。
- (4) 运行给定的测试任务，能正确输出结果。



The screenshot shows a terminal window with a dark purple background. The title bar indicates the user is 'parallels@ubuntu' in the directory '/media/psf/home/buffer/mips_os/project_4/finished_code'. The terminal output consists of several lines of text: '[TASK] This task is to test scheduler. (5)', '[TASK] This task is to test scheduler. (5)', '[TASK] Has acquired lock and running.(3)', and '[TASK] Applying for a lock.'. Below the text is a small diagram of a task with a box labeled '0' and a line labeled 'o'. The terminal status bar at the bottom shows 'CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7 | VT102 | Offline | ttyUSB0'.

4.5.2 文件说明

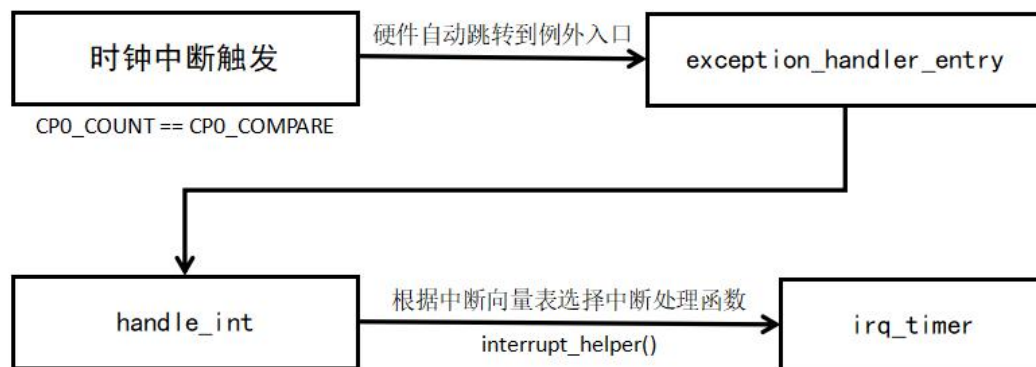
请基于任务 3 的代码继续进行实验。

4.5.3 实验步骤

- (1) 完善 main.c 中 init_exception 代码，主要完成异常处理相关的初始化内容：将例外处理代码拷贝到例外处理入口、初始化例外向量表，初始化 CP0_STATUS、CP0_COUNT、CP0_COMPARE 等异常处理相关寄存器。
- (2) 完善 entry.S 中 exception_handler_entry，主要完成例外处理入口相关内容：关中断、保存现场、根据 CP0_CAUSE 寄存器的例外触发状态跳转到中断处理函数（handle_int）。
- (3) 完善 entry.S 中 handle_int 代码，主要完成的内容为：跳转到中断向量处理函数（interrupt_helper 方法）、恢复现场、开中断。
- (4) 在 irq.c 中完善中断处理的相关代码中断向量处理（interrupt_helper 方法）、时钟中断处理函数（irq_timer 方法），具体内容请同学们自己思考如何实现。
- (5) 考虑调度算法，完善 sched.c 中 scheduler 方法，使其基于优先级和等待时间进行调度（优先级越高，越先进行调度；等待时间越长，越先进行调度）。
- (6) 修改 test.c 中 sched1_tasks 数组中的三个任务（去除 do_scheduler 方法的调用），要求打印出和任务 1 一样的正确结果。

4.5.4 注意事项

- (1) 由于 MIPS 下例外入口地址是固定的，因此我们要将我们自己实现的例外入口函数（exception_handler_entry）拷贝到该指定位置，具体实现请同学们自行思考。
- (2) 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。
- (3) 时钟中断方法调用栈如下，请认真思考每个函数应该对应哪一级的例外处理：



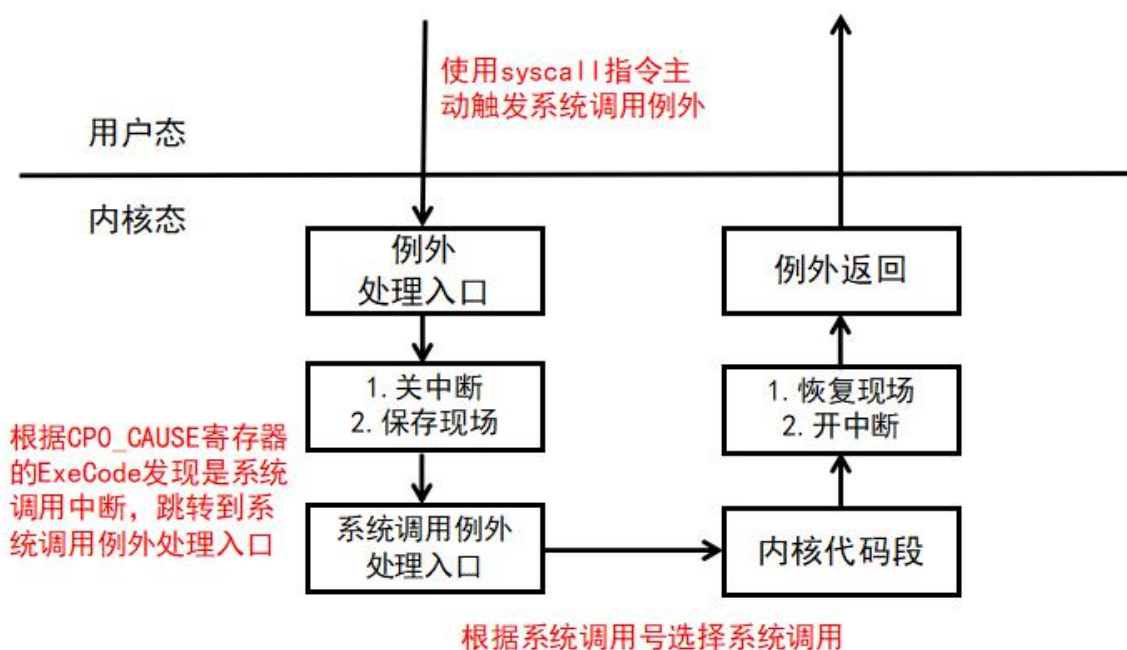
例外处理对应的函数栈

其中 `exception_handler_entry` 的代码就是例外处理中的第二级，`handle_int` 和 `interrupt_helper` 的代码就是例外处理中的第三级，`irq_timer` 代码就是处理时钟中断的具体代码。因为时钟中断处理不需要考虑第四级例外处理，因此只处理三级即可。

4.5 系统调用

之前的任务里，都是直接调用了内核的代码，但其实作为用户进程的任务是不允许直接访问内核代码段的，需要通过内核给出的访问接口去访问内核代码段，调用内核的功能。这个访问接口我们通常称之为**系统调用**。

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 `syscall` 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，这和中断处理流程大致一样，具体的流程如下图所示：



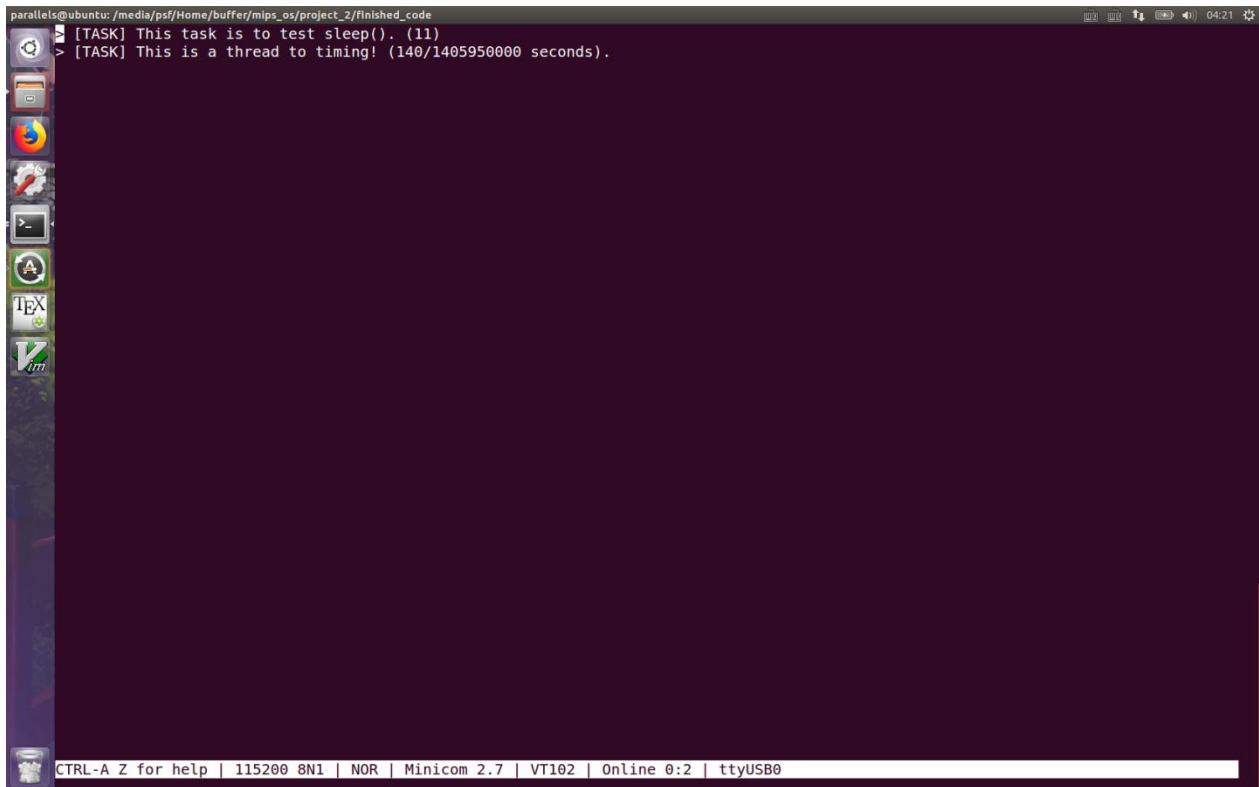
系统调用处理流程

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。

4.6 任务 4：系统调用

4.5.1 实验要求

- (1) 掌握 MIPS 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。



```
parallels@ubuntu: /media/psf/Home/buffer/mips_os/project_2/finished_code
> [TASK] This task is to test sleep(). (11)
> [TASK] This is a thread to timing! (140/1405950000 seconds).
```



```
parallels@ubuntu: /media/psf/Home/buffer/mips_os/project_4/finished_code
> [TASK] This task is to test scheduler. (5)
> [TASK] This task is to test scheduler. (5)
> [TASK] Has acquired lock and running.(3)
> [TASK] Applying for a lock.
```

Diagram of a lock structure:

```
graph LR
    Lock[ ] --- Box[0]
    Lock --- o[o]
```

4.5.2 文件说明

请基于任务 3 的代码继续进行实验。

4.5.3 实验步骤

- (1) 完善 main.c 中系统调用相关初始化 (init_syscall)。
- (2) 完善 syscall.S 中的 invoke_syscall 方法，需要完善的内容为调用 syscall 指令发起一次系统调用。
- (3) 完善 entry.S 中 exception_handler_entry，需要完成的内容为：根据 CP0_CAUSE 寄存器的内容跳转到指定系统调用处理方法 (handle_syscall)。
- (4) 完善 entry.S 中 handle_syscall 方法、syscall.c 的 system_call_helper 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。
- (5) 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。
- (6) 完善我们的用户态打印函数库 printf。
- (7) 实现我们之前完成的 lock 的系统调用。
- (8) 完成 sys_move_cursor 移动光标的系统调用。
- (9) 运行给定的测试任务 (test.c 中 timer_tasks 数组中的三个任务、test.c 中 sched2_tasks 数组中的三个任务、test.c 中 lock_tasks 数组中的任务)，要求打印出正确结果。

4.5.4 注意事项

- (1) 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。
- (2) 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。
- (3) 了解 MIPS 下 syscall 指令的作用，该指令会触发系统调用例外。
- (4) 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) 对于系统调用的处理不像中断处理那么繁琐，但处理的流程大致相似，请同学们仔细思考二者的区别，代码框架内正确的系统调用方法调用栈如下：



系统调用处理函数栈

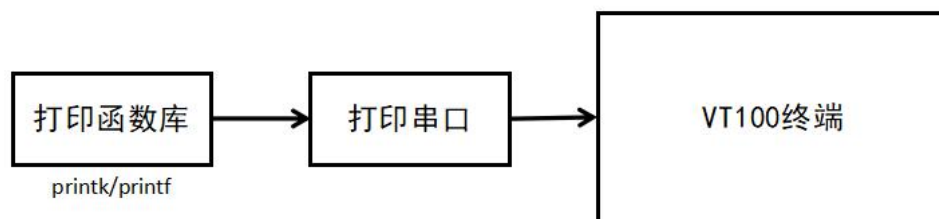
- (6) 对于系统调用相关锁的测试任务已经下载了 test_lock2.c 中，修改 makefile 中的 test_lock1.c 为 test_lock2.c 即可。

五、打印函数

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）。为了方便同学们对该库的理解，同学们可以通过这章了解打印机制，并思考为何这么做。

5.1 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，如下图：



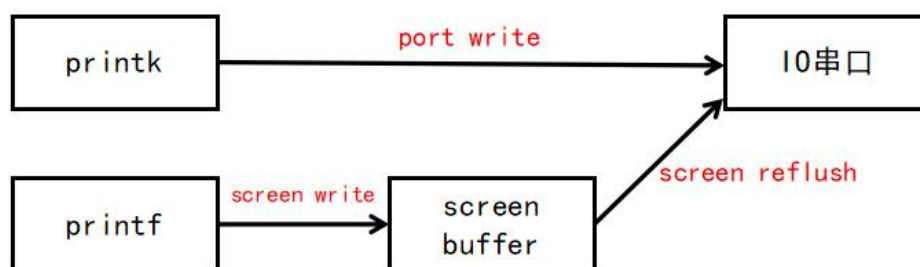
VT100 是一个终端类型定义，VT100 控制码是用来在终端扩展显示的代码，我们只需要往串口输入一些特定的字符串，就可以完成 VT100 终端的打印控制，比如光标移动，字体变色等功能，以下为部分控制码：

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			

5.2 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，我们的做法依旧还是每次还是直接往串口写数据。如下图：



5.3 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。