OS Lab 3 实时调度算法

lab任务与评分

1. lab的任务简介 本lab主要是在linux调度系统的基础上完成一套实时调度算法，确保实时内核的实时性。涉及到的知识点有：线程的创建与调度，中断子系统，时间子系统。目前可以解决优先级反转的算法还在实现，所以lab3暂无bonus部分。 通过lab3，大家可以理解rros是如何改造linux的调度系统，达到保证内核的实时性的目标，并且让linux内核作为一个优先级最低的调度实体来运行通用的应用。
2. lab的前置操作

下载本仓库中的lab3.patch，然后你可以在lab2的基础上重新开辟一个分支来完成此题。

```
git checkout -b lab3
git reset --hard lab2_commit
git apply lab3.patch
git add .
git commit -m "lab3 base"
```

此处的`lab2_commit`为你`git log`中为lab2提交学号的commit号，或者提交学号commit的前一个commit号。

最后一定要将本项目中的`.config`文件放到rros工程项目中替换原有的`.config`文件，本次实验的`.config`和之前的有所不同，成功编译后就可以开始本次lab之旅了。

1. 说明lab的评分规则和Due
   - 各部分测试的分值如下：

项目	分值
`__rros_enqueue_fifo_thread`	20
`lookup_fifo_class`	15

项目	分值
`do_clock_tick`	20
`__rros_schedule`	10
`rros_enable_tick setup_proxy`	20
报告	15
合计	100

- **Due: 2023.6.10**

lab基础知识讲解

实时操作系统基本知识

通用操作系统上运行任务的正确性依赖于任务执行的结果，而实时任务的正确性不仅依赖于结果的正确性还需要确保任务在规定的deadline内执行完成。我们将任何一次的deadline都不能miss的任务叫做硬实时任务（hard real-time tasks），将可以容忍miss一些deadline的任务叫做软实时任务（soft real-time tasks）。如果任务的deadline被miss了之后，程序才执行完成得到结果，那么我们可以根据这个结果是否有价值，将软实时任务分为狭义的软实时任务和准实时任务（firm real-time tasks），狭义的实时任务在deadline后得到的结果仍然是有价值的，而准实时任务则不然。

实时操作系统（Real-time Operating System, RTOS）[1]是一个可以在确定的极小时间内对外部相应做出反应的操作系统。一般来说，RTOS和通用操作操作系统（General Purpose Operating System，GPOS）的区别在于，RTOS在设计时就不考虑吞吐量，而是一个在对外界的事件做出反应时的确定极小时延。一个可以确定满足硬实时任务需要/也就是meet deadline的操作系统，可以称之为硬实时操作系统（hard RTOS），会提供各种必要的机制来满足一个确定性的最小时延要求，而一个只能通常meet deadline的操作系统，我们称之为软实时操作系统（soft RTOS）。

RTOS主要是围绕着在一个确定的极小响应时间来设计的，会改造OS中的各个子系统来适应这个目标，这个响应时间必须要是确定的，所以更关注响应时间的worst case而不是mean value，所以不能认为快是RTOS的主要特点，有确定性的快才是。比如：

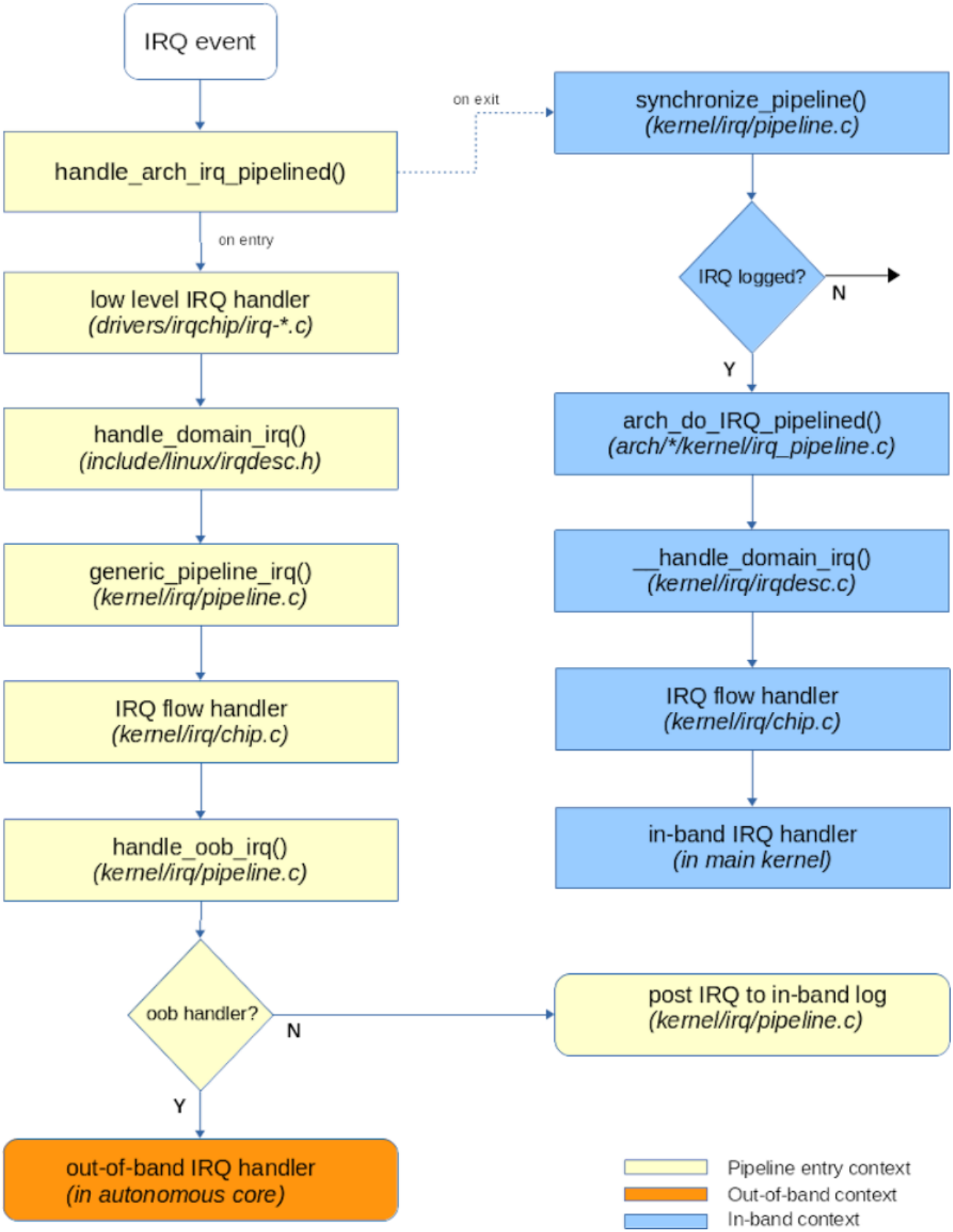
1. 对于调度系统，RTOS通常会设计为一个可抢占的内核，以减少对高优先级任务的响应时间；在引入同步机制后，为了防止优先级反转问题的出现（这也是我们这个Lab主要需要解决的内容），可能会采用天花板算法或者优先级继承算法。
2. 对于内存子系统，关键的数据结构不会采用动态内存分配的机制；为了确定性，可能不使用MMU、Cache、TLB等硬件；内存子系统的分配算法；在下一个Lab中，我们将会涉及到一个实时的内存子系统是如何设计的。

接下来，我们会讲解实时调度算法的基本知识，然后说明在rros系统中这个算法应该如何实现。

实时调度算法的基本知识

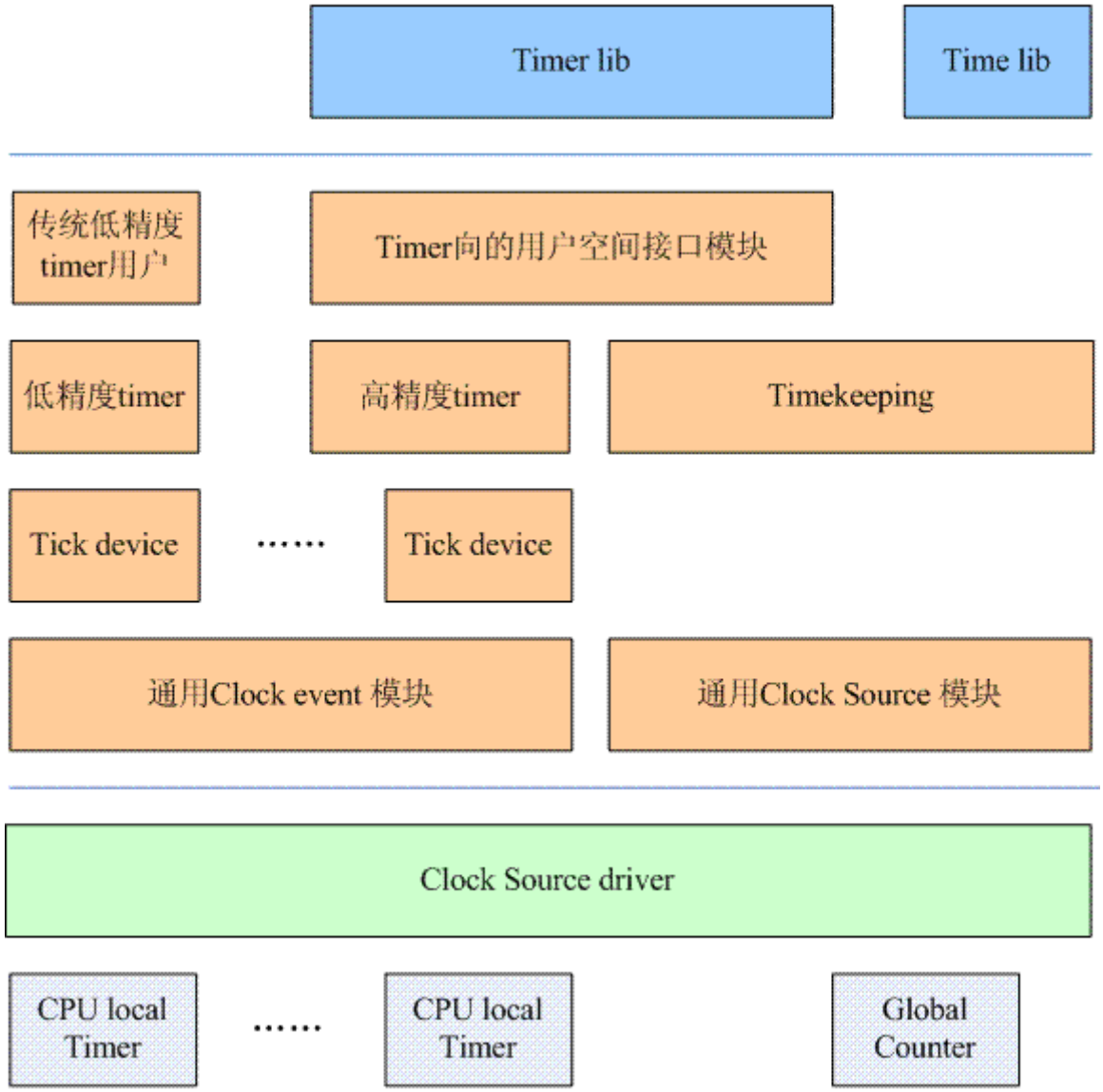
理解rros的中断和时间子系统

`dovetail`会拦截linux的中断，然后让rros先处理，大概的处理流程如下图所示。



可以看到一个中断会先被oob的系统处理，如果这个中断是oob的，那么会调用对应的oob handler来处理，如果是inband的中断，那么会放到一个记录的区域，然后在带内获得运行权限后，传递给带内。因为dovetail拦截了中断，那么需要考虑三个方面的事情，首先需要考虑修改对linux关键上下文的影响，还需要考虑对锁的影响，还有一个是对rcu的影响。我们的lab不会涉及到dovetail的这些复杂的内容，需要做的是补全dovetail需要调用的接口。可以看到在`dovetail`拦截了linux的中断后，我们可以让rros优先处理所有的中断，而不会被linux的任何行为所打断。

在介绍rros的时间子系统之前，需要说明一下linux的时钟子系统。关于Linux的时间子系统网络上有很多讲解的资料，可以参考《深入理解linux内核》[2]或者蜗窝科技[3]的时间子系统系列博客。我们在这里仅做一个简要的概述。Linux的时间子系统主要分为定时和计时两部分。前者是来描述如果一个定时器到期后，应该做出何种反应；后者是用来描述os是如何维护一个时间轴，以便我们可以准确的获取不同精度的绝对/相对时间。Linux的时间子系统组成如下图：



clock event和clock source就是分别针对这两个功能的底层硬件描述，系统中有多少个硬件，就会注册多少个相应的结构体。而每一个cpu都会选择一个最合适的设备来作为当前CPU的tick device来周期性的对当前CPU发出tick中断。rros的时间子系统会替换掉linux的tick device，更换其时钟中断处理函数，将tick发送到rros中驱动rros的调度子系统，只有当rros中没有任何任务时，才会将tick传递给linux，驱动linux的调度。

我们在接下来的lab中的第三个任务就是完善上面讲述的这套机制。

调度算法

在了解实时调度算法之前，我们需要首先复习一下一般的调度算法原理及其架构实现。

一般来说，调度算法分为两部分，调度的机制和调度的策略。 调度的机制分为两种情况：调度的机制由一个周期性的tick驱动，在每一次tick时，都会在tick的结尾判断当前是否需要进行调度，这种情况属于被动调度；当然线程也可以主动调用如sleep之类的函数，主动放弃cpu的所有权，然后选择一个其他的线程进行执行，这种情况属于主动调度。 而调度的策略是在选择其他进程执行时，决定我们应该选择哪一个进程的算法，有时间片

轮转，最短运行时，fifo等多种策略，而且我们在一个内核中同时定义多种调度策略，具体来说我们规定了每一次选择进程时，不同调度策略的优先级，我们会先选择优先级最高的调度算法，从它的队列中选择可以执行的线程，如果没有找到，我们再从次最高优先级的调度算法线程队列中选择，依次类推。

我们lab的第二个和第三个任务就是完善调度机制和调度的策略。

lab实操

我们的lab分为三个小节，一个是rros的fifo调度算法；第二个小节是rros是如何设计一个调度子系统来同时调度实时任务并且将linux的任务作为优先级最低的任务进行调度；第三个小节是了解linux是如何处理时钟中断的，rros又是如何改造的。

rros的fifo调度算法

fifo算法顾名思义就是一个根据入队时间先进先出的算法，rros的fifo算法有优先级的feature，同一个优先级下根据先进先出排序，不同的优先级下根据fifo的原则来排序。我们需要实现这个算法的入队和出队算法部分。

1. 接口1 入队 `__rros_enqueue_fifo_thread`

rros在一个新的进程诞生或者一个旧的进程获得资源时，会通过`rros_enqueue_thread`来让这个进程进入fifo的调度队列，进而能够在pick_next_thread时找到对应的队列。

2. 接口2 挑选队列中的线程 `lookup_fifo_class`

rros在每一次调用`schedule`函数进行调度时，都需要从fifo的runnable queue中挑选一个线程，如果挑选不到可以执行的线程，那就需要在更低优先级的队列中挑选，idle就是一个比fifo优先级更低的队列。

rros的实时调度算法 一个基本的调度器设计

1. 接口1 对系统中的tick做出响应 `do_clock_tick`

在每一次系统的时钟tick到来时，系统会调用`do_clock_tick`函数查看当前是哪一个时钟到期了，然后进行处理。

2. 接口2需要在实现了“rros的中断和时间子系统”小节中的两个接口后实现

rros的中断和时间子系统

rros会通过更换底层`clock_event_device`中的相关函数，来劫持系统的tick。主要是两个函数：`rros_core_tick`是rros用来处理如果tick到来后应该怎么办的函数，而`proxy_set_next_ktime`是用来确定下一个tick什么时候响的函数。

注意： 只有当你完成了接口1和2时，才能完成下一个任务，接口1和2没有单独的测试，如果你正常完成了代码，系统将不会打印"rust_kernel: init clock_proxy_device error!" 和 "rros: cpd new error!"错误。

1. 接口1 截获系统中的tick: `rros_enable_tick`

`rros_enable_tick`是来劫持系统中原本的`clock_event_device`。

2. 接口2 具体对clock device中的函数完成替换 `setup_proxy`

主要替换`handle_oob_event`与`proxy_set_next_ktime`函数。

rros的实时调度算法 一个基本的调度器设计

1. 接口2 上下文切换 `__rros_schedule`

`__rros_schedule`可以确认当前的系统是否需要线程切换。在`__rros_schedule`里面，rros通过`dovetail_context_switch`调用linux中的`context_switch`进行上下文切换。

lab测试方法

当编写完函数后，可以在命令行中执行`qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd ../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append "rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 2048`来运行内核，测试编写的api代码是否通过。

如果通过全部测试，将会有以下输出

```
[ 0.342510] rros: [RAND]Pass test fifo enqueue with the prority.
[ 0.346441] rros: [RAND]Pass test lookup fifo class.
[ 0.667724] rros: [RAND]Pass test handle clock tick.
[ 0.691443] rros: [RAND]Pass test context switch between threads.
```

注意`rros_enable_tick`和`setup_proxy`没有单独的测试接口，如果运行成功，将不会看到"rros: cpd new error!"和"rust_kernel: init clock_proxy_device error!"错误。

如果没有通过某一个测试，将会看到测试不成功的提示

```
[2.3608357] rros: Failed to pass fifo enqueuewth the nroritw
[2.3698357] caused by: Test failed on kerne./rros/fifo test.rs:104:9
```

引用

[1] https://en.wikipedia.org/wiki/Real-time_operating_system
[2] https://baike.baidu.com/item/%E6%B7%B1%E5%85%A5%E7%90%86%E8%A7%A3LINUX%E5%86%85%E6%A0%B8%EF%BC%88%E7%AC%AC%E4%B8%89%E7%89%88%EF%BC%89/9829376?fromtitle=%E6%B7%B1%E5%85%A5%E7%90%86%E8%A7%A3LINUX%E5%86%85%E6%A0%B8&fromid=6153389&fr=aladdin
[3] http://www.wowotech.net/timer_subsystem/time-subsyste-architecture.html