2020高校开发者大赛--初赛文档

## 项目名称

赛题编号：4

赛题名称：LUTF - Linux Userspace Task Framework

项目地址：<https://github.com/Xiyou-LUG>

项目管理：<https://www.tapd.cn/31119725/prong/stories/stories_list>

## 团队信息

### 团队名称

团队名称：Xiyou-LUG

团队编号：1382578369

## 项目详细方案

### 对题目项目背景进行分析

#### 对题目项目需求总体分析

该题目是实现一个Linux用户态下任务调度框架，原生的Linux是支持多任务的，但是由于是内核统一管理，所以任务的粒度都是内核所固定的。这在一些数据处理相关联的场景下，就需要对任务支持的粒度更加细化，来更好的处理这些任务CPU密集型的任务，避免内核态下任务的频繁切换。所以提出一种Linux用户态下多任务的想法，对比目前的协程的优势和不足，我们打算设计出内嵌在OS中的负责任务调度的小OS。最后希望达到如下目标：支持百万级的任务数量、实现FIFO调度策略、后期加入其他更高效的调度算法。

#### 协程角度进行技术背景分析

##### 赛题分析

基于赛题的角度进行分析，我们需要实现一个基于 Linux 操作系统的用户态任务调度，主要应对的场景是计算密集型。

##### 使用用户态调度的优点

* 以 java 语言举例,单个线程 stack 初始化超过1mb，而单个任务的栈空间大小根据实际业务需求设计。
* 内核线程与普通线程 1:1，每当需要涉及执行流切换的时候,需要陷入内核态保存一些内核线程的上下文信息，开销很大。

因此对任务进行更加细粒度的划分，实现用户态的任务切换与调度是很有必要性的，我们根据赛题目标现在提供两种实现思路。

* 模拟 Linux 操作系统内核的任务调度（调度单位为task），实现一个用户态版内核，实现任务调度（调度单位为任务）。
* 将任务抽象为协程，实现类似与go语言协程调度的一个任务调度框架。

因为赛题提出的任务概念与协程十分类似，所以我基于第二种思路，我们参看了目前开源的协程框架（云风的coroutine，腾讯的libco，go1.5.1语言的协程调度器），并对其中各个框架进行了分析。

#### 协程框架分析

##### 云风 coroutine

介绍

这是我们对云风 coroutine 源码做的一份[注释](https://github.com/RandyLambert/LibraryCodeComments/tree/master/coroutine)，云风协程库实现了单线程的任务调度，不支持自动调度，主要基于 Linux 中的 ucontext 簇函数实现任务切换，用 coroutine 结构体定义协程。在使用的时候在协程中手动 yield 从而放弃执行，之后跳转到全局的 schedule ，schedule 找到下一个可运行的协程，进行切换。

优点

* 代码简洁明白，结构体设计值得参考。
* 使用 ucontext 簇函数比较好的实现了上下文切换。
* 使用共享栈机制，减少了内存压力，使百万并发可以实现。

缺点

* 没有实现任务的自动切换，不符合项目要求。
* 只支持单线程调度。
* 没有实现抢占式调度。

##### 腾讯 libco

介绍

libco 自从 2013 年之后，在腾讯微信后台大规模使用，他是一套经受住考验的协程调度框架，libco实现了任务调度，原理是通过 hook 一些阻塞的 Linux 系统调用（read，write，gethostname，mysqlclient等），利用 Linux I/O 多路复用技术（我们同样对 epoll 源码做了一份比较详细的[注释](https://github.com/RandyLambert/LibraryCodeComments/tree/master/epoll)）,每当调用阻塞的系统函数，在对应的 epoll 实例上面注册对应的 epoll 事件，等到可读或者可写事件触发，之后切回对应协程进行处理；对于任务切换，实现内联汇编函数进行上下文的保存与切换，更加高效，libco 默认使用有栈协程，如果需要并发数过大，也可以调整为共享栈模式，并且可以调整共享栈的数量。

优点

* 基于超时和对阻塞系统调用的 hook 实现了自动切换。
* 可以根据实际情况选择共享栈模式或者有栈模式。

缺点

* 使用内联汇编进行上下文保存，取消了 sigmask 和 浮点数上下文,因此不符合 LUTF 的需求。
* libco 主要面向于 I/O 密集型任务实现任务调度，也不符合 LUTF 所要求计算密集型任务调度的需求。

##### go 1.5.1 的协程调度框架

介绍

go 协程调度主要基于 GMP 模型，下面是对 go 1.5.1协程调度框架的解释。

* M---System Thread 系统线程。
* P---Processor go语言实现的协程处理器。
* G---Goroutine 协程。

1. 每一个 System Thread 有一个 Processor , 每个 Processor 挂着一个正在运行的协程队列，Processor 依次运行挂在上面的协程，如果一个协程运行时间很长，有可能会阻塞住整个协程队列，所以在 go 程序运行起来的时候会有一个守护线程，去记录每个 Processor 运行完成的协程数量，如果有一段时间发现某个 Processor 很长时间执行完的协程数量没有发生变化，则会往协程的任务栈中插入一个特别的标记，当这个协程运行的时候遇见非内联函数时，就会读到这个标记，就会把自己中断下去，把自己插到等候协程队列的队尾，然后切换成别的协程运行。
2. 另一个机制是，如果某一个协程被系统中断了，例如IO，需要等待，为了提高系统并发， Processor 会把他移动到另一个可用的系统线程之中。继续去运行所挂队列中其他的协程，当被中断的协程被唤醒完成之后，会把自己加到全局等待队列里或者某个 Processor 的协程等待队列中。
3. 当一个协程中断，他在寄存器里的运行状态也会保存在协程对象里,如果在次运行会将运行状态弹回寄存器中。

优点

* 作为 go 语言安身立命的根本, go 的协程调度实现了真正的通用性。
* 实现了多线程的任务调度，可以完美的利用多核性能。
* 性能在多次迭代之后十分优秀。

缺点

* 在 go1.5.1 的时候, go 实现的是基于协作式的抢占式调度器，并不是真正的抢占式调度(但是 go 在1.14之后实现了基于信号的真抢占式调度)

##### libgo

目前正在研究

#### 协程角度总结

以上几种协程框架都有自己的优势和局限性，不能单纯去参考某一个框架，需要根据实际要求和需要，尽量做出最佳的设计与实现。

#### Linux0.11进程分析

为了更好的理解赛题，组员高星坤去了解Linux0.11下进程的创建、销毁、以及调度策略。在Linux操作系统上可以同时运行多个进程依靠的是分时技术，分时技术把CPU运行时间分成一个个规定长度的时间片，让每个进程在一个时间片内运行，在单核CPU上虽然某时刻只能运行一个进程，但是由于是键盘很短，所以好像所有进程都在运行。我们的LUTF也许可以参考进程的分时技术，把每一个任务分配一定的时间片，在用户态下依靠定时信号SIGALARM来模拟中断，实现对任务分时，但是由于大多是CPU密集型任务，所以我们的时间片也许需要更长一点(Linux0.11 15毫秒)，避免任务切换的频率过高。

Linux0.11下内核通过对进程表对进程进行管理，每个进程在进程表中占有一项。在Linux系统中，每个进程对应一个task\_struct称为PCB，我们的任务也需要一个任务对象，我们称为TCB，详细介绍见下文。

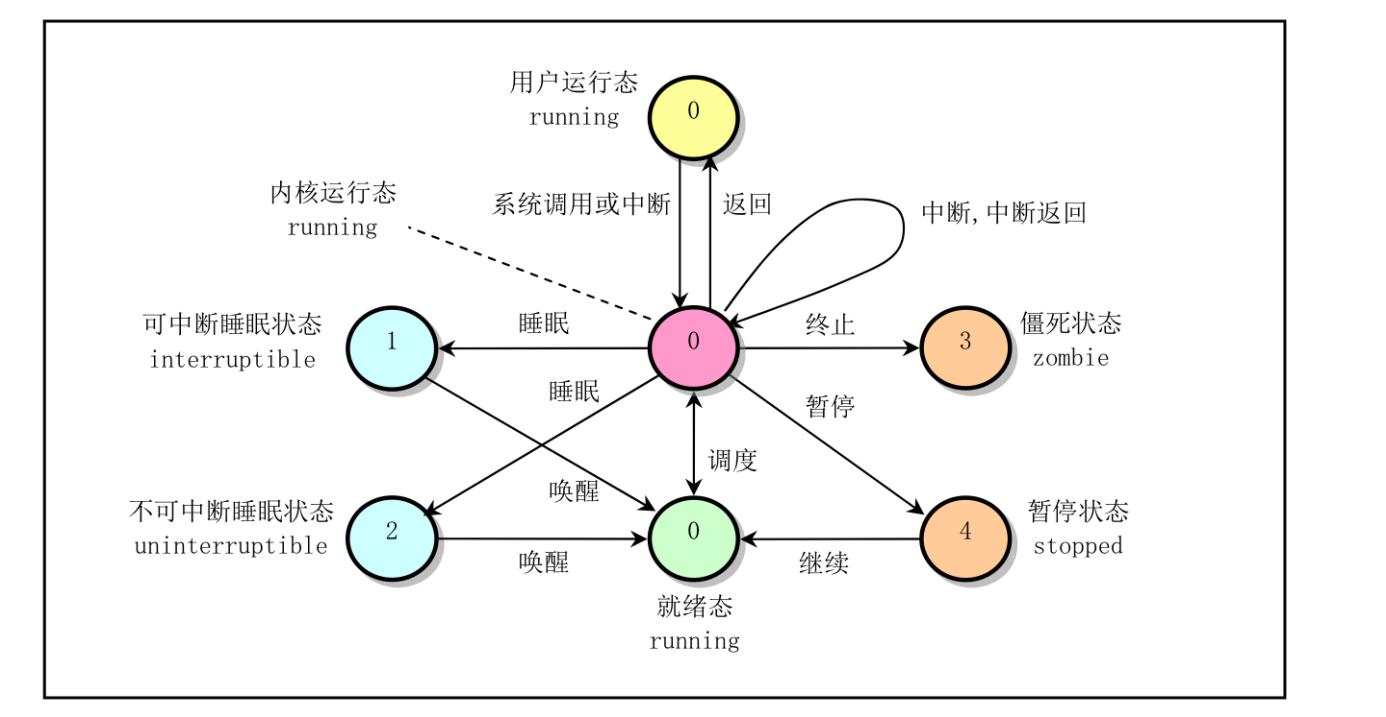
进程的状态有运行状态、可中断睡眠状态、不可中断睡眠状态、暂停状态和僵死状态，对应的我们的任务暂时设计了运行状态(TASK\_RUNNING)、阻塞状态(TASK\_BLOCKED)和就绪态(TASK\_READY)三种状态。图1为Linux0.11进程状态转换关系图。

图1 进程状态及转换关系

紧接着我们的任务就是完成创建任务、任务调度、任务切换和终止任务，除此之外还要考虑任务对堆栈的使用。

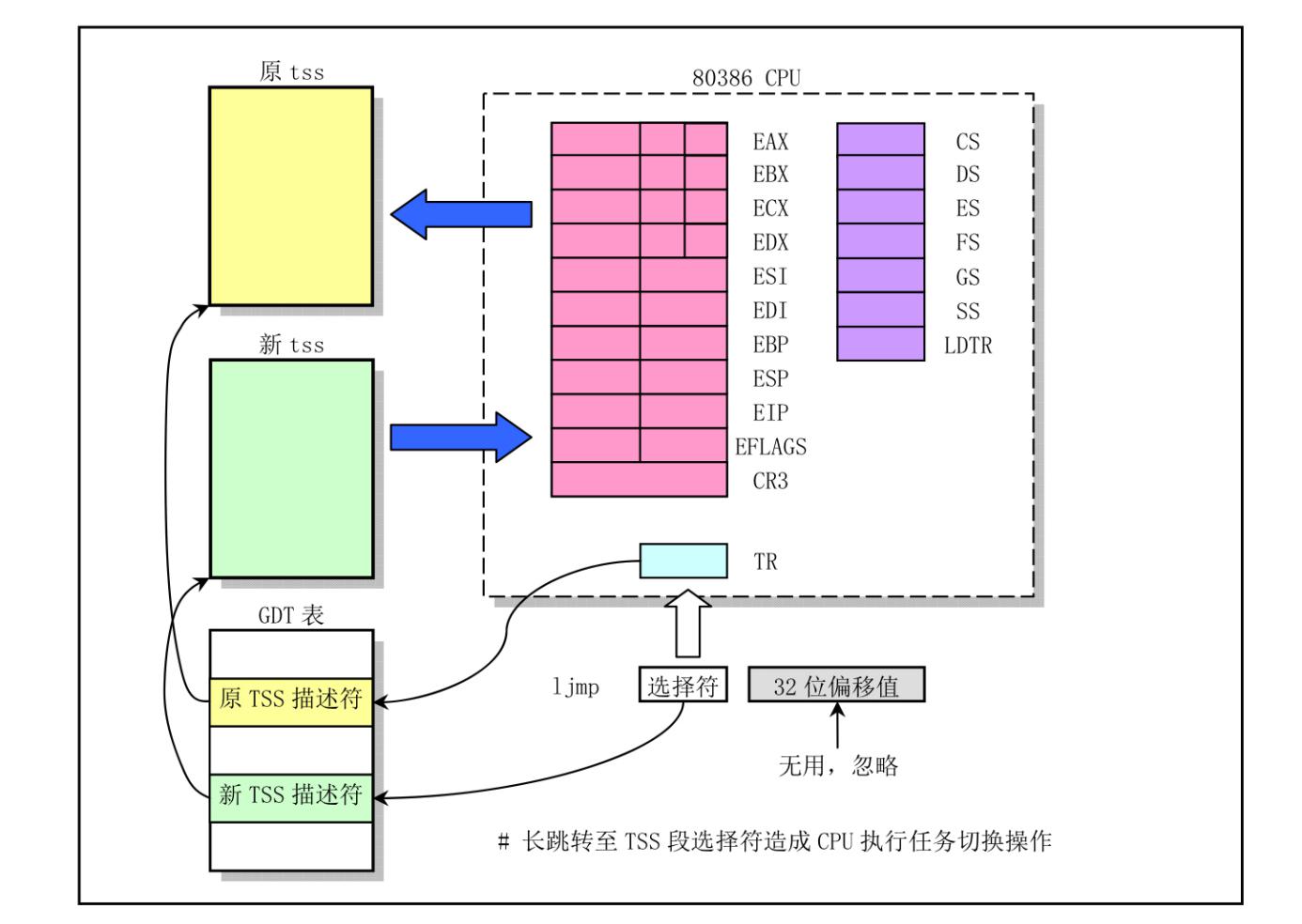


图2 进程切换操作示意图

#### 用户线程分析（多对一模型）

传统操作系统中，拥有资源和独立调度的基本单位都是进程。在引入线程的操作系统中，线程是独立调度的基本单位，进程是资源拥有的基本单位。同一进程中，线程的切换不会引起进程切换。在不同进程中进行线程切换，如从一个进程内的线程切换到另一个进程中的线程时，会引起进程切换。

根据操作系统内核是否对线程可感知，可以把线程分为内核线程和用户线程。

在多线程操作系统中，各个系统的实现方式并不相同，在有的系统中实现了用户级线程，有的系统中实现了内核级线程。

##### 用户级线程实现方式

有关线程管理的所有工作都是由应用程序完成，内核意识不到线程的存在，应用程序可以通过使用线程库设计多线程程序，通常，应用程序从单线程起始，在该线程中开始运行，在其运行的任何时刻，可以通过调用线程库中的派生例程创建一个在相同进程中运行的新线程。

用户级线程只是存在于用户空间，此类线程的创建、撤销、线程之间的同步与通信功能，都无须利用系统调用实现。用户进程会利用线程库来控制用户线程。

由于线程在进程内切换的规则远比进程调度和切换的规则简单，不需要在用户态和内核态进行切换，所以两线程之间的切换速度快，代价小。

由于这里的处理器时间片分配是以进程为基本单位，所以每个线程执行的时间相对减少，为了操作系统中加入线程支持，采用了在用户空间增加运行库来实现线程，这些运行库被称为“线程包”，用户线程是不被操作系统所感知的。

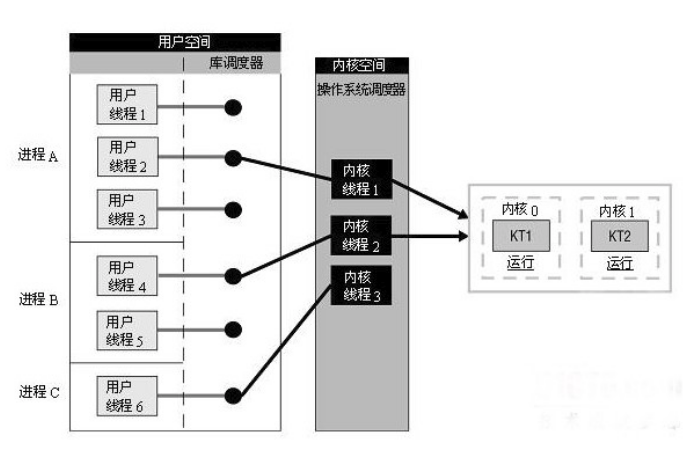


图3 用户线程调度

如图3可看出，库调度器会从进程的多个线程中选取一个线程，然后该线程和线程所属进程被允许进入内核空间，与其中一个内核线程关联起来。

内核线程将被操作系统调度器指派到处理器运行。用户级线程是一种“多对一”的线程映射。

##### 用户级线程的特点

内核对线程包是一无所知的。所以从内核角度考虑，就是按照正常的方式来进行线程管理，即单线程进程。在用户级线程中，每个进程里的线程表由运行时系统管理。当一个线程转换到就绪状态或阻塞状态时，在该线程表中存放重新启动该线程所需的信息，与内核在进程表中存放的进程的信息完全一样

##### 用户级线程的优点

* 创建和销毁线程、线程切换代价等线程管理操作的代价比内核线程小得多，因为保存线程状态的过程和调用程序都只是在用户态本地进行；
* 线程能利用的表空间和堆栈空间比内核级线程多；
* 线程的调度不需要内核直接参与，控制简单。

##### 用户级线程的缺点

* 线程发生I/O、页面失效或页面故障引起阻塞时，如果调用阻塞系统调用，则内核由于不知道有多线程的存在，而会阻塞整个进程，从而阻塞所有线程；
* 一个单独的进程内部，没有时钟中断，所欲不可能用轮转调度的方式调度线程；
* 资源调度按照进程进行，多处理机下，同一个进程中的线程只能在同一处理机下分时复用。

#### 内核线程

##### 内核线程与普通进程的异同

1. 跟普通进程一样，内核线程也有优先级和被调度。当和用户进程拥有相同的static\_prio 时，内核线程有机会得到更多的cpu资源；
2. 内核线程的bug直接影响内核，很容易搞死整个系统, 但是用户进程处在内核的管理下，其bug最严重的情况也只会使进程崩溃；

##### 内核线程模块对于本次项目借鉴意义

内核线程的缺陷：

* 由用户态陷入内核态，这种切换的消耗较大，用户态线程的切换无需陷入内核，速度较快，在设计任务调度切换的时候应该注意保留任务的高速缓存；
* 同步效率：资源的同步和数据共享比整个进程的数据同步和共享要低一些。

内核线程的优点：

* 内核线程的使用是廉价的；
* 处理器资源分配合理，可以在全系统范围内竞争处理器资源；

可见内核线程的使用资源少，但是调度开销大，同步效率低

### 对实现题目的方案设计

#### 方案一设计：类虚拟机

#### 架构图

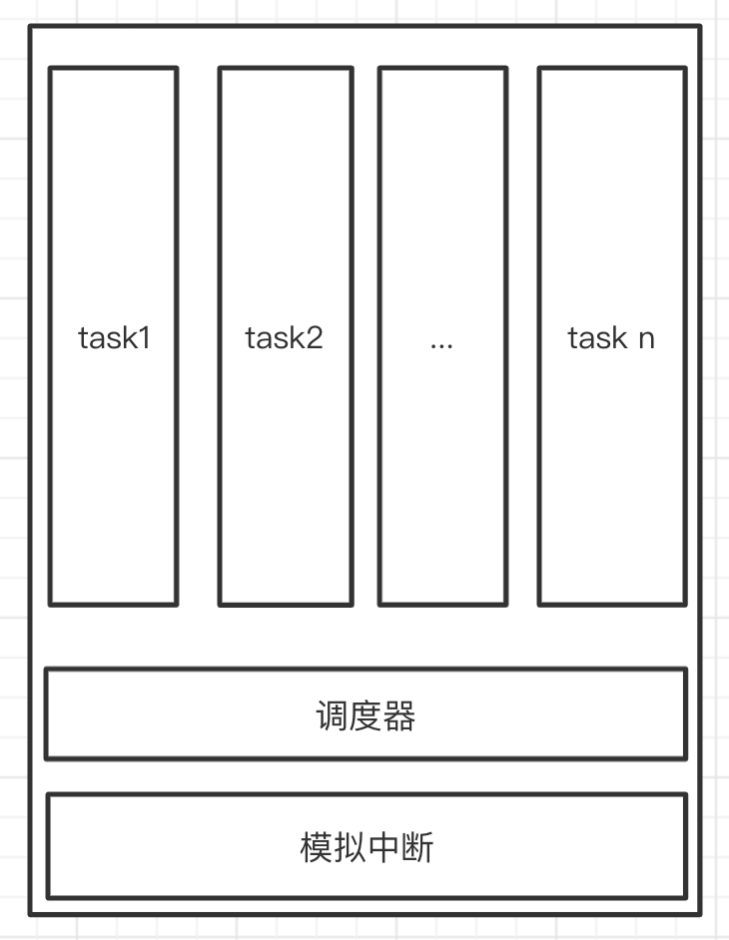


图4 类虚拟机架构示意图

1.1、 模拟中断模块

底层使用虚拟化技术模拟出时钟中断，这样在使用时钟中断的时候更加的自由，摆脱使用信号的诸多弊端。使用模拟中断的好处在于任务切换的时候更加的方便，每个任务的切换都可以做到透明，不需要调用换出函数切换到调度器再调度下一个任务。使用显式的换出无法将各个任务真正的分离；处理时钟中断的处理函数时，时钟中断处于屏蔽状态。当任务第一次执行的时候，在调用任务处理函数之前打开时钟中断，这样调度器与任务就实现了分离，也就是每个任务的执行流真正的分离。任务分离后就可在不同的线程中切换，虽然会带来切换代价，但在多核处理机上表现会更好。背景描述该架构面向计算等场景，那么应该多考虑多核处理机。

1.2、多任务调度模块

使用时钟中断驱动调度器，每次时钟中断判断分配给任务的时间片是否到期，如果没有到期就将时间片减一，如果到期就调用schedule函数切换任务。调度器中可以有不同的调度策略，暂时使用的调度策略为FIFO。

通过调度器维护一个任务队列。每创建一个任务或者任务时间片到期将该任务加入到队尾，schedule函数从对头得到一个任务作为下一个要运行的任务。

后期实现在多核处理机运行的框架时，暂定的框架如图5所示。多任务模块启动时，会根据处理器核心数创建n个线程。为了提升多核平台的资源利用率和系统运行性能，将每个线程绑定在一个独立的处理器运行运行。全局会有一个所有任务的记录，当一个处理线程的任务执行完毕后，可以根据全局记录的任务从别的处理器线程获得任务，实现任务与处理器线程之间的负载均衡。

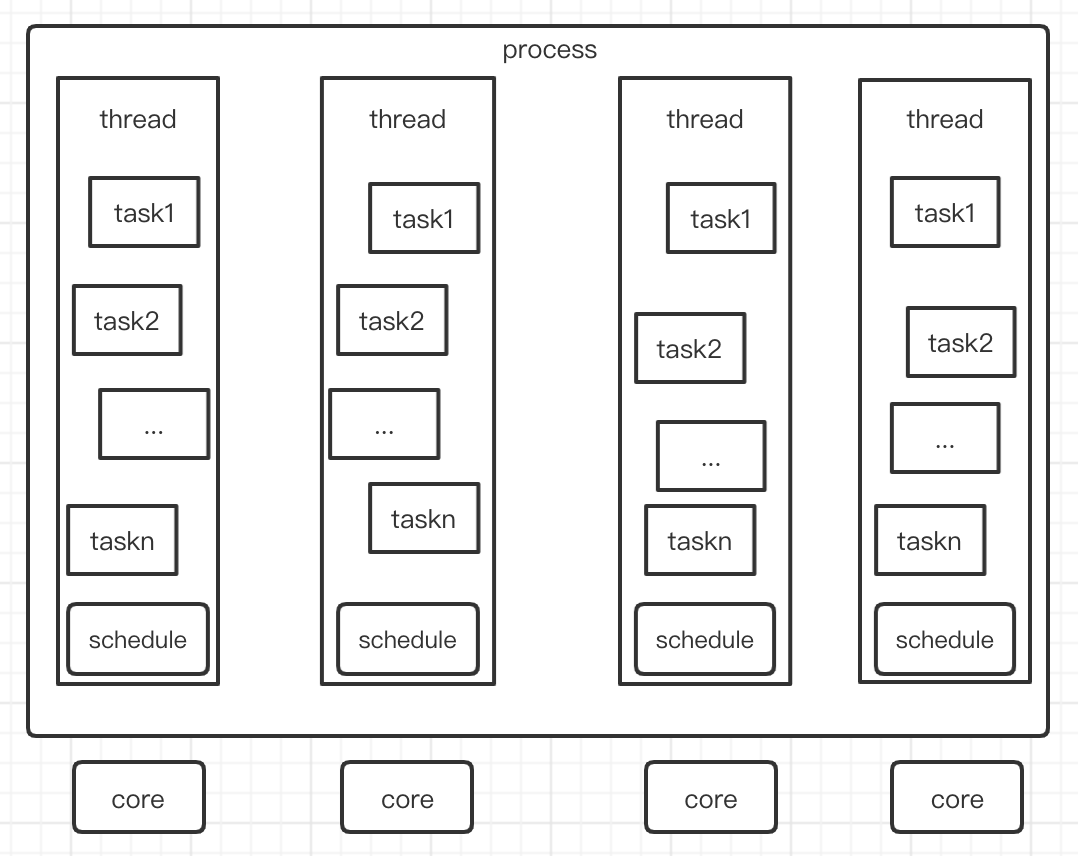


图5 多任务运行架构

#### 流程图

##### 时钟中断处理函数

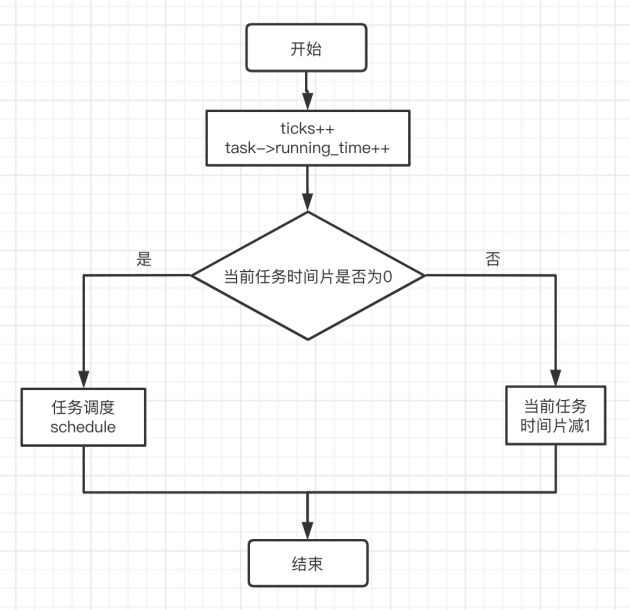


图6 时钟处理函数流程图

ticks为该进程第一次处理时间中断后至今的滴答数，总滴答数。task->running\_timer为当前任务占用CPU的时间。

##### 调度器schedule

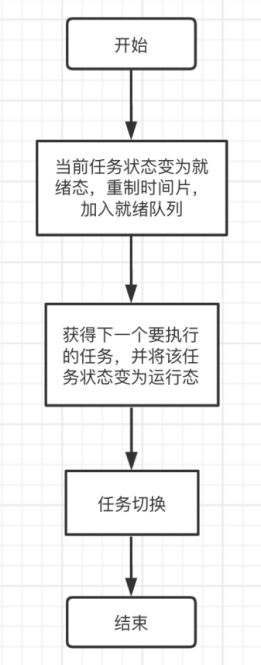


图7 调度器流程图

获得下一个执行的任务暂时使用的策略为FIFO，即从任务队列的对头取出任务。

##### 任务切换switch\_to\_next

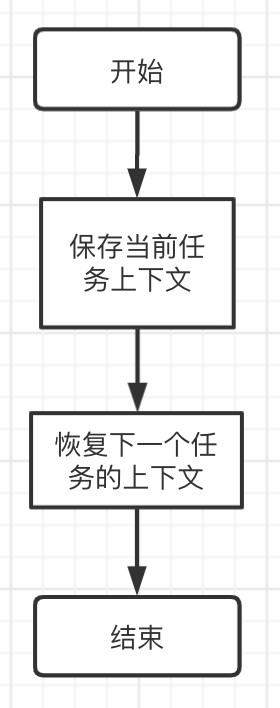


图8 任务切换流程图

第一次执行的任务的上下文在创建的时候指定好，其中rip寄存器的位置位置指向统一的函数，该函数首先打开中断然后调用任务函数。

#### 方案二设计：类协程

目前我们现在有两种设计方案，因为直接做一个用户态的操作系统去调度目前难度过大，而且对于性能上面也不能做出保证，因此我们第一版采用第二种实现方式，将任务抽象为协程，实现一个基于共享栈机制协程调度框架。

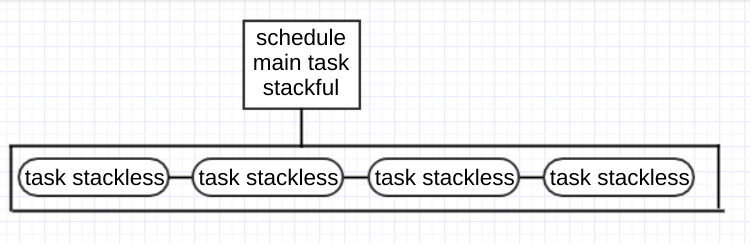


图9 类协程架构示意图

/\*\*

\* 抽象任务调度器的结构体

\*/

struct schedule

{

char stack[STACK\_SIZE]; // 运行时栈

ucontext\_t main; // 主任务的上下文

int nco; // 当前存活的任务个数

int cap; // 任务管理器的当前最大容量，即可以同时支持多少个协程。可以进行扩容

int running; // 正在运行的任务ID

struct coroutine \*\*co; // 用于存放任务

};

/\*

\* 抽象任务结构体

\*/

struct coroutine

{

coroutine\_func func; // 任务所用的函数

void \*ud; // 任务参数

ucontext\_t ctx; // 任务上下文

struct schedule \*sch; // 该任务所属的调度器

ptrdiff\_t cap; // stack区已经分配的内存大小

ptrdiff\_t size; // 当前任务运行时栈，保存起来后的大小

int status; // 任务当前的状态

int times; // 时间片

char \*stack; // 当前任务的保存起来的运行时栈

};

实现方案：

通过设置定时时间，（设置最短时间为1毫秒）定时发送信号触发对应的定时处理函数，模拟时钟中断的形式。每次中断进入信号处理函数，在信号处理函数中，首先查询有没有更高优先级的协程出现，如果有，则通过 swapcontext() 函数调度到该任务，如果没有，则将当前正在运行的 coroutine 时间片 times 进行 -- 操作，如果 times 的值减为 0 ，切换到 schedule ，去调度另一个任务继续执行。

通过这种方式，我们可以通过 signal 模拟出来一个简单的基于时间片和拥有优先级调度的任务调度框架。

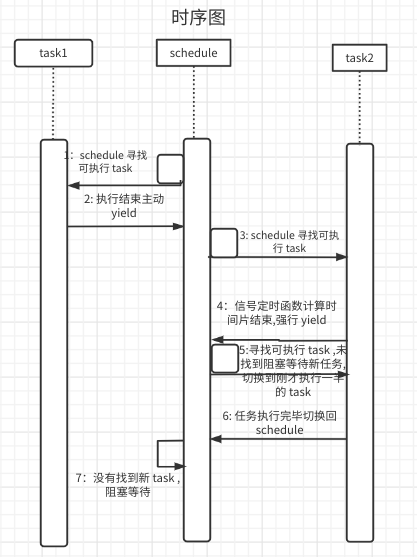


图10 调度时序图

未来改进方向：

1. 第一版实现的单线程中的协程调度，之后为了能使用多核性能，会改进为多线程调度。
2. 会更加深入的研究了解 go 语言的协程调度，尤其是 1.14 之后的,，go 实现了基于信号的真抢占式调度，与项目所要求比较接近。
3. 尝试自己完成协程上下文切换，提高性能。

#### 提供工作进度计划

暂定大致工作计划

|  |  |
| --- | --- |
| 时间 | 任务 |
| 2020年11月25日——2020年12月08日 | 相关知识学习、资料查询、简单demo实现和文档编写 |
| 2020年12月08日——2020年12月10日 | 最后文档修改和提交文档 |
| 2020年12月10日——2020年03月05日 | 方案二开发、模拟中断的实现 |
| 2020年03月05日——2020年03月20日 | 方案一开发（移植工作） |
| 2020年03月20日——2020年03月31日 | 文档编写和作品提交 |

#### 实现难点分析

* 模拟中断的实现
* 功能分层化、模块化，方便后期移植
* 百万级任务调度和存储空间的使用
* 为提高效率根据ABI自定义上下文

#### **目前实现进度**

实现了类协程的任务调度，包括初始化主任务，创建任务，任务调度及任务退出。使用SIGALARM信号进行调度。运行截图如图4所示。

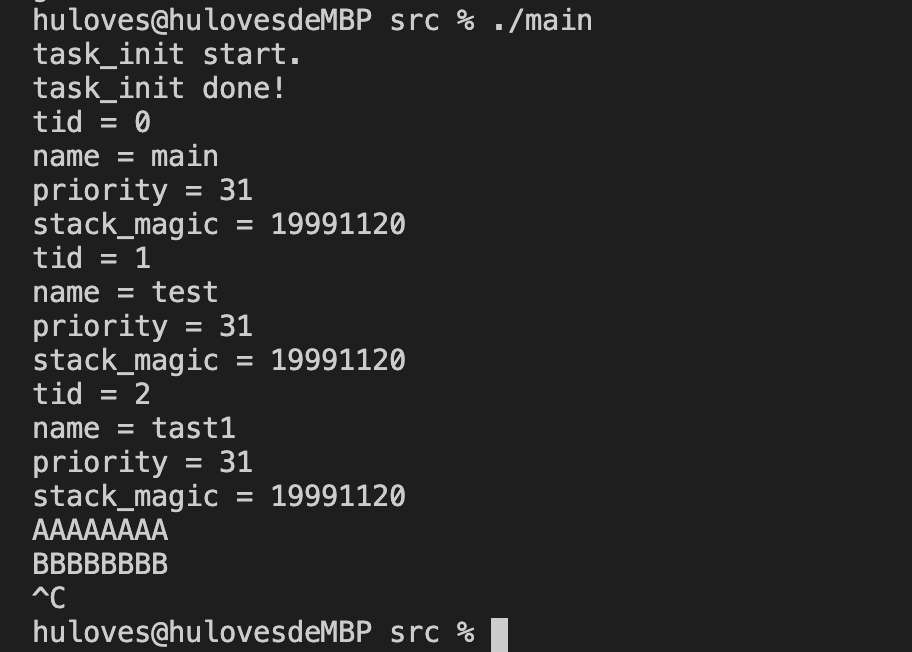


图11 当前demo运行截图

如图4所示，首先进行初始化，tid为0的任务为主任务，创建了两个任务test和test1。主任务调用pause()阻塞。终端显示了创建的三个任务的信息，然后开始调度test和test1任务。代码如下所示，整体代码见“一”中链接，下文也会展示链接。

#include "task.h"

#include "analog\_interrupt.h"

#include <stdio.h>

#include <unistd.h>

void test(void\* args)

{

printf("AAAAAAAA\n");

task\_exit(current\_task);

}

void test1(void\* args)

{

printf("BBBBBBBB\n");

task\_exit(current\_task);

}

int main()

{

task\_init();

print\_task\_info(current\_task);

task\_start("test", 31, test, NULL);

task\_start("tast1", 31, test1, NULL);

struct task\_struct\* ptask = tid2task(1);

print\_task\_info(ptask);

ptask = tid2task(2);

print\_task\_info(ptask);

interrupt\_enable();

while(1) {

pause();

}

return 0;

}

当前项目分析：

主任务并没有参与任务调度，所以暂时将主任务挂起。基于信号量机制进行抢占在执行任务函数的时候是通过函数调用的形式执行，若想再次调度下一个任务必须要显式得返回schedule，没有内核使用中断方便。中断的优势在于它相对于CPU完全独立，x86架构中只要将IF位置位即可接收外部中断（8289芯片设置完成），使用该机制很容易将各个线程执行流分离，只需要在调用线程函数前打开中断即可（线程的中断栈及线程栈初始化完成）。

在现有的协程框架中并没有找到解决将各个任务执行性完全独立的方案，所以经过思考决定在调度器schedule之下模拟出中断层，使用该中断作为任务切换的基础。

基于该模拟中断方案可更好的适配多核处理器的场景。该项目是以新兴计算等场景为背景，在这样的背景之下，多核处理机的应用十分广泛。在设计时应考虑到多核的问题，如何在提高核心数量的同时将效率成线性增长。因为各个执行流相互之间完全独立，故可将任务分配到不同的执行线程执行。线程之间也是完全独立，将线程在不同CPU之间调度不会影响执行的结果。

但考虑团队技术栈的原因，在之前没有接触过虚拟化技术，所以会先实现方案二。在实现方案二的时候尽量将功能模块分层化、模块化，在后期移植的时候方便移植。与切换相关的函数、调度函数以及信号处理函数部分进行模块化，后期只需改部分变函数调用即可。

项目开发过程会进行版本管理，最后会将两个方案进行比较。最终效率结果还将以最终测试结果为准。

完整项目代码地址：<https://github.com/Xiyou-LUG>

### 对技术实现提供验证方案

* 1. 需求分析
* 根据对于本次项目的了解，我们的LUTF更加需要适应cpu密集型的任务,我们将会同时执行多个求数独的函数以模拟cpu密集型任务测试压力。
* 对于中断我们将会写出测试脚本定时发送信号，实现＂守护＂的职责。
* 我们将对任务计算量特别大的任务进行测试，同时执行任务计算量较小的任务，测试计算量较大的任务是否会优先执行，较小的任务是否会正常且合理的调度。
* 我们将对任务特权级较大的任务进行测试，同时执行任务特权级较小的任务，测试特权级较大的任务是否会优先执行，较小的任务是否会正常且合理的调度。

2、对框架可行性进行验证

* 本次测试将对每个接口使用多种不同的函数和传值进行测试。
* 对于程序的数据，将会进行验证在我们的方案下数据是可靠并发的。

3、与现有的调度框架进行性能对比

与此同时，我们将会在linux下使用多线程/多进程/多协程对同样的任务函数进行测试，对比执行完成的时间，和同时间内可执行的任务数量。

* 根据要求执行百万数量不同权级的任务，负载均衡地分配给8个线程执行，记录执行完成时间。
* 根据要求执行百万数量不同权级的任务，负载均衡地分配给8个进程执行，记录执行完成时间。
* 根据要求执行百万数量不同权级的任务，负载均衡地分配给多个协程执行，记录执行完成时间。
* 在本任务框架下执行百万数量不同权级的任务，负载均衡地分配给多个任务调度借口执行，记录执行完成时间。
* 将所有的时间进行记录，比较完成时间，得出本任务框架的性能优化比例。