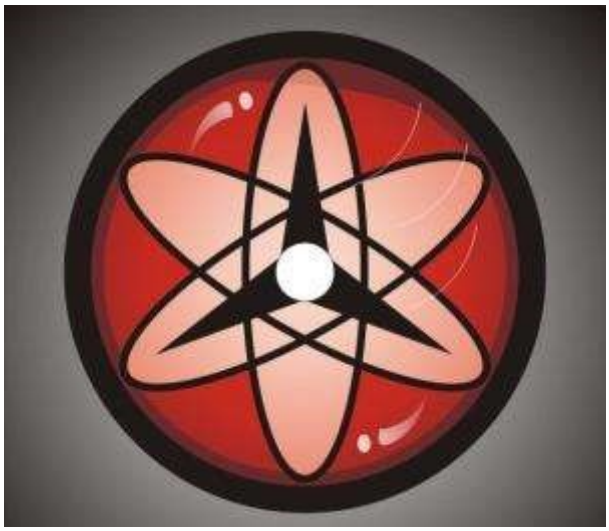
第3章-气功-系统编程简述

仰望星海, 编程行业类比玄幻小说中修真体系 [炼气 -> 筑基 -> 金丹 -> 元婴 -> 化神]. 那时候在学校里或者野路子中锻炼并感应着天地间的元气, 在练气期幸福而不知睡眠. 感受着编程行业斑驳交割的元气, 最终选择几类元气开始自己的练气生涯. 期间勤奋些的或者时间够了, 一舒心中豪情啪的一声进入筑基期. 心随意动, 修炼生涯正式展开. 蹭蹭的我们进入了门派中磨炼. 随着门派体系或者一些心有不甘的选手日夜操戈, 自我驱动跃升成为人魔大战的前线主力. 挥洒鲜血, 几朝凝华金丹成. 此时的战场才刚刚拉开序幕. 同样以前修炼遗留的隐患也一并爆发. 无数人在此厮杀, 对抗域外天魔. 此刻或者在远古战场中获得奇遇, 又或者占有一个门派的全力支持, 通过大毅力和机缘破吾金丹, 晋升元婴大佬. 隐射一方, 出手之间自带领域威势. 回顾也是白骨功成, 为门派马首是瞻. 唯独极个别生时天资聪慧, 道心自明的元婴大佬. 忘却红尘迷恋, 占一代之强气运, 耐一世之大孤独, 甩手间风云变幻, 天雷滚滚, 超脱物外, 万中无一化神巨擘独立无为之境, 位于东方. 无一丝情感遥望着心的远方, 立于缥缈峰 ~ 窥探浩瀚 ~

各位看官, 化神难道就是编程的至高境界了吗? 然而一切才刚刚开始, 这里先不表. 本书讲的气功, 等同于练气期修炼的法术. 打通和操作系统联系的基本关节. 专业程序或多或少依赖于平台, 不同平台的修炼会有所大不同. 本章是在一个平台间练就一门气功入门, 海典剑芒贯双江更多平台知识留给那些化神大佬惠泽气运新人吧.

3.1 原子操作

原子操作一个古老的庞大话题, 常见简单有在不主动放弃 CPU 情况下解决资源竞争问题. 在简述原子库之前简单科普些基本原子操作.



3.1.1 引言

举个简单例子:

```
volatile int i = 0;

// ++i 大致可以拆分为下面三步
//
// 1. 把 i 的值放入寄存器中
// 2. 把寄存器中的值加 1
// 3. 返回寄存器中值并设置给 i
++i;
```

以上执行会导致一个问题, 如果两个线程同时执行到 步骤 1 那么造成一个现象是 i 最终没有预期的大. 如何避免上面问题呢? 常见思路是互斥. 当然这里有更好路子, 利用编译器提供的原子操作. 本质利用 CPU 提供的原子指令的封装(CPU 提供总线锁定和缓存锁定保证复杂内存操作的原子性). 说直接点, 可以用编译器提供这方面基础能力, 让我们实现原子相加. 例如 GCC 就提供不少像下面指令.

```
type __sync_add_and_fetch (type * ptr, type value, ...);
type __sync_lock_test_and_set (type * ptr, type value, ...);
bool __sync_bool_compare_and_swap (type * ptr, type old, type new, ...);
```

这类原子操作的特殊表达式可以直接边查编译手册, 边写个小例子, 就会知道窍门. 我们简单解释下, __sync_add_and_fetch 等同于将 ptr 指向的内存加上 value 值, 并且返回最终加好的值. __sync_lock_test_and_set 的意思是把 value 的值给 ptr 指向的内存, 并且返回 ptr 原先指向的内存值. __sync_bool_compare_and_swap 的意思是判断 ptr 指向的值和原先的 old 相等吗, 相等将其设置为 new. 并且返回 ptr 指向值和 old 相等与否的 bool 值. 为了让大家更好理解, 不妨封装一层, 请细看注释:

```
// v += a ; return v;
#define atom_add(v, a)      __sync_add_and_fetch(&(v), (a))
// type tmp = v ; v = a; return tmp;
#define atom_set(v, a)      __sync_lock_test_and_set(&(v), (a))
// bool b = v == c; b ? v=a : ; return b;
#define atom_cas(v, c, a)   __sync_bool_compare_and_swap(&(v), (c), (a))
```

以上定义了 add set cas 原子操作. 随后原子基础库中会封装更多更常用的. 更多细节可以细查 man 手册, 一切无所遁形.

3.1.2 原子自旋锁

spinlock.h

```c
#pragma once

#include <stdbool.h>
#include <stdatomic.h>

/*
    https://zh.cppreference.com/w/c/atomic

    https://en.cppreference.com/w/c/atomic/atomic_compare_exchange

    _Bool atomic_compare_exchange_strong(volatile A * obj,
                                         C * expected, C desired);

    _Bool atomic_compare_exchange_weak( volatile A * obj,
                                       C * expected, C desired );

    if (memcmp(obj, expected, sizeof *obj) == 0) {
        memcpy(obj, &desired, sizeof *obj);
        return true;
    } else {
        memcpy(expected, obj, sizeof *obj);
        return false;
    }
*/

inline bool atomic_bool_compare_exchange_weak(volatile atomic_bool * obj) {
    bool expected = false;
    return atomic_compare_exchange_weak(obj, &expected, true);
}

inline bool atomic_bool_compare_exchange_strong(volatile atomic_bool * obj) {
    bool expected = false;
    return atomic_compare_exchange_strong(obj, &expected, true);
}

// spin lock

inline void atomic_flag_lock(volatile atomic_flag * lock) {
    while (atomic_flag_test_and_set(lock)) {}
}

inline void atomic_flag_unlock(volatile atomic_flag * lock) {
    atomic_flag_clear(lock);
}

inline bool atomic_flag_trylock(volatile atomic_flag * lock) {
    return atomic_flag_test_and_set(lock) == 0;
}
```

代码已经表述了一切好的坏的有得没得, 推荐当字帖临摹多抄写 ~ 这些代码很短, 使用起来也很容易. 例如在上一章写了个 chars 字符串. 他不是线程安全的. 可以利用原子自旋锁, 简单改成线程安全版本:

```
#include <chars.h>
#include <spinlock.h>

struct cstr {
        atomic_flag lock;
        struct chars str;
};

// 初始化
struct cstr a = { .lock = ATOMIC_FLAG_INIT };

// 加锁
atomic_flag_lock(&a.lock);

// 使用
// 各种对于 astr.str 操作都是线程安全的
// ...
chars_appends(&a.str, "100");

printf("str = %s, cap = %zu, len = %zu\n", chars_get(&a.str), a.str.cap, a.str.len);

// 释放锁
atomic_flag_unlock(&a.lock);

printf("a = 0x%p, a.str = 0x%p, a.str.str = 0x%p\n", &a, &a.str, a.str.str);

// 销毁
free(a.str.str);
```

以上是原子自旋锁使用的核心步骤. 当然了, 装波的事情远远还没有结束. 普通开发要求下编程本身就那些东西, 讲明白后大家就很容易懂. 切记编程路上多真善美, 增加修成元婴可能性. 当然金丹大圆满也都能够很好的胜任大部分生涯工作. 上面原子锁仍然可以优化, 例如采用忙等待和阻塞混合编程, 降低 CPU 空转, 等等优化, 具体可以研究 pthread mutex 做的拓展了解二者使用场景. 总而言之在解决资源竞争问题上, 消耗最小不一定是无锁编程, 更舒爽的方式, 往往是通过业务优化避免冲突的产生, 减少锁粒度和频率. 我们在做开发时候, **如果没想明白场景要不要用原子锁的时候, 那请直接用互斥锁**, 别犹豫.

3.1.3 内存顺序简单扫盲

在很多年前, 我们因为不同平台原子库形态各异, 要费大力气进行跨平台封装. 现在好了太多了, 认真研究标准中 **stdatomic.h** 完全足够胜任绝大多数奇葩业务了. 这里会简单扫盲 stdatomic.h 中引入的六种不同 memory order 来控制同步的粒度, 可能会获得更好的程序性能吧. 这六种 order 分别是:

```
/* 7.17.3 Order and consistency */
typedef enum memory_order {
  memory_order_relaxed = __ATOMIC_RELAXED,
  memory_order_consume = __ATOMIC_CONSUME,
  memory_order_acquire = __ATOMIC_ACQUIRE,
  memory_order_release = __ATOMIC_RELEASE,
  memory_order_acq_rel = __ATOMIC_ACQ_REL,
  memory_order_seq_cst = __ATOMIC_SEQ_CST
} memory_order;
```

- **memory_order_relaxed 宽松内存顺序：**

没有同步或顺序制约, 仅对此操作要求原子性. 带 memory_order_relaxed 标签的原子操作不考虑线程间同步操作, 其他线程可能读到新值, 也可能读到旧值. 只保证当前操作的原子性和修改顺序一致性. 例如:

```
// atomic init
atomic_int x = 0, y = 0;

// 线程 1 操作
int a = atomic_load_explicit(&y, memory_order_relaxed); // A
atomic_store_explicit(&x, a, memory_order_relaxed);     // B

// 线程 2 操作
int b = atomic_load_explicit(&x, memory_order_relaxed); // C
atomic_store_explicit(&y, 28, memory_order_relaxed);    // D
```

允许产生结果 a == 28 && b == 28. 因为即使线程 1 中 A 先序于 B 且线程 2 中 C 先序于 D, 但没法保证 y 的修改顺序中 D 比 A 先执行, x 的修改顺序中 B 比 C 先执行. 这就会导致 D 在 y 上的副效应, 可能可见于线程 1 中的加载 A, 同时 B 在 x 上的副效应, 可能可见于线程 2 中的加载 C. **宽松内存顺序的典型应用场景是计数器自增**. 例如引用计数器, 因为这只要求原子性保证自增 OK, 但不要求顺序或同步(注意计数器自减要求进行 memory_order_acquire 获取内存顺序同步)

- **memory_order_consume 消费内存顺序：**

有此内存顺序的加载操作, 在其影响的内存位置进行消费操作: 当前线程中依赖于当前加载的该值的读或写不能被重排到此加载前. 其他释放同一原子变量的线程的对数据依赖变量的写入, 为当前线程所可见. 在大多数平台上, 这只影响到编译器优化. 例如线程 1 中的原子存储带标签 memory_order_release 而线程 2 中来自同一原子对象的加载带标签 memory_order_consume, 则线程 1 视角中依赖先序于原子存储的所有内存写入(非原子和宽松原子的), 会在线程 B 中加载操作所携带依赖进入的操作中变成可见副效应, 即一旦完成原子加载, 则保证线程 2 中, 使用从该加载获得的值的运算符和函数, 能见到线程 1 写入内存的内容. 同步仅在释放和消费同一原子对象的线程间建立. 其他线程能见到与被同步线程的一者或两者相异的内存访问顺序.

```
// atomic init
int a = 0;
atomic_int x = 0;

// 线程 1 操作
a = 1;
// memory_order_release 释放内存顺序
// 后面所有与这块内存有关的读写操作都无法被重排到这个操作之前
atomic_store_explicit(&x, 1, memory_order_release);

// 线程 2 操作
while (atomic_load_explicit(&x, memory_order_consume) != 1) {
    if (a == 1) { // a 可能是 1 也可能是 0

    }
}
```

更好理解的是下面这个例子, a 的值一定为 0. 但多数编译器没有跟踪依赖链, 均将 memory_order_consume 消费内存顺序操作提升为 memory_order_acquire 获得内存顺序操作.

```
atomic_int x = 0;

int a = atomic_load_explicit(&x, memory_order_consume);
// a 的值一定是 0, memory_order_consume 后面与这块内存的相关代码不会重排到他前面
x = 1;
```

- **memory_order_release 释放内存顺序：**

有此内存顺序的存储操作进行释放操作, 当前线程中的读或写不能被重排到此存储后. 当前线程的所有写入, 可见于获得该同一原子变量的其他线程(获得内存顺序), 并且对该原子变量的带依赖写入变得对于其他消费同一原子对象的线程可见. 例如一些原子对象被存储-释放, 而有数个其他线程对该原子对象进行读修改写操作, 则会形成"释放序列": 所有对该原子对象读修改写的线程与首个线程同步, 而且彼此同步, 即使他们没有 memory_order_release 语义. 这使得单产出-多消费情况可行, 而无需在每个消费线程间强加不必要的同步. 同样 unlock 也全靠 memory_order_release 释放内存顺序

- **memory_order_seq_cst 序列一致内存顺序：**

有此内存顺序的加载操作进行获得操作, 存储操作进行释放操作, 而读修改写操作进行获得操作和释放操作, 会加上存在一个单独全序, 其中所有线程以同一顺序观测到所有修改. 如果是读取就是 acquire 语义, 如果是写入就是 release 语义, 如果是读写就是 acquire-release 语义. 通常情况下编译器默认使用 memory_order_seq_cst. 在你不确定如何选取这些 memory order, 可以直接用此内存顺序.

```
/* 7.17.8 Atomic flag type and operations */
typedef struct atomic_flag { atomic_bool _Value; } atomic_flag;

#define atomic_flag_test_and_set(object) __c11_atomic_exchange(     \
        &(object)->_Value, 1, __ATOMIC_SEQ_CST)

#define atomic_flag_clear(object) __c11_atomic_store(               \
        &(object)->_Value, 0, __ATOMIC_SEQ_CST)
```

但当 memory_order_acquire 及 memory_order_release 与 memory_order_seq_cst 混合使用时, 会产生诡异的结果. 对于 memory_order_seq_cst 需要了解的注意点: 1' memory_order_seq_cst 标签混合使用时, 程序的序列一致保证就会立即丧失 2' memory_order_seq_cst 原子操作相对于同一线程所进行的其他原子操作可重排.

有了简单基础, 自身陆续学习 C11 stdatomic.h 理念和功能, 完全可以没有包袱的直接使用. 如果有些平台不支持 C11, 如果只是学习阶段, 那可以跳过不支持平台, 去支持平台去学习, 年轻人就应该有拒绝者的朝气和态度.

3.2 POSIX 线程库

对于 POSIX 标准线程库, 也就是我们在 Linux 上使用的 pthread 线程库. 首先为其罗列些常用的 api 提纲. 先看 PThread Attribute Functions 系列

```
//
// pthread_attr_init    - 初始化线程环境
// pthread_attr_destroy - 销毁线程环境
// attr         : pthread_attr_t 线程环境
// return       : 0 标识成功, -1 标识失败
//
extern int __cdecl pthread_attr_init (pthread_attr_t * attr);
extern int __cdecl pthread_attr_destroy (pthread_attr_t * attr);

//
// pthread_attr_setdetachstate - 设置线程的运行结束后的分离属性
// attr         : pthread_attr_t 线程环境
// state        : 默认 PTHREAD_CREATE_JOINABLE, 需要 pthread_join 清理遗留数据
//                      : PTHREAD_CREATE_DETACHED 属性等同于 pthread_detach, 结束即销毁
// return       : 0 表示成功
//
extern int __cdecl pthread_attr_setdetachstate (pthread_attr_t * attr,
                                                int state);
```

有了线程环境相关操作, 再来看看线程构建的相关操作 PThread Functions

```
//
// pthread_create - 创建一个线程，并自启动实体运行
// tid                  : 返回创建线程的句柄 pthread_t 类型变量
// attr                 : 线程创建初始化的量，pthread_attr_xxxx 系列设置
// start                : 线程创建成功后运行的实体
// arg                  : start 运行时需要的额外参数
// return               : 返回 0 表示成功，-1 表示失败 并会设置 errno
//
extern int __cdecl pthread_create (pthread_t * tid,
                                   const pthread_attr_t * attr,
                                   void * (__cdecl * start) (void *),
                                   void * arg);

//
// pthread_equal - 两个线程 id 比较
// t1                   : 线程 id
// t2                   : 线程 id
// return               : 1 表示二者相同，0 表示二者不同
//
extern int __cdecl pthread_equal (pthread_t t1, pthread_t t2);

//
// pthread_exit - 退出当前线程
// value_ptr    : 会保存在线程的私有变量中，留给 pthread_join 得到
// return       : void
//
extern void __cdecl pthread_exit (void * value_ptr);

//
// pthread_join - 等待线程 pthread_create -> start 函数运行结束
// thread       : 线程 id
// value_ptr    : 返回 start 返回值，或 pthread_exit 设置的值
// return       : 0 表示成功，-1 标识失败
//
extern int __cdecl pthread_join (pthread_t thread, void ** value_ptr);
```

线程互斥量, 基本和 pthread_create 使用频率差不多. 加上手工注释希望大家能够感性认识他, Mutex Attribute Functions 相关操作. 前面一直忘记说了, 展示过很多系统层的源码实现, 我们是基于此刻假定这些实现就应该这样, 因为他会因时而异.

```
#define PTHREAD_MUTEX_INITIALIZER ((pthread_mutex_t)(size_t) -1)

// pthread_mutex_init    - 初始化线程互斥量
// pthread_mutex_destroy - 销毁线程互斥量, 必须和 pthread_mutex_init 成对
extern int __cdecl pthread_mutex_init (pthread_mutex_t * mutex,
                                       const pthread_mutexattr_t * attr);
extern int __cdecl pthread_mutex_destroy (pthread_mutex_t * mutex);

// pthread_mutex_lock   - 加锁
// pthread_mutex_unlock - 解锁
extern int __cdecl pthread_mutex_lock (pthread_mutex_t * mutex);
extern int __cdecl pthread_mutex_unlock (pthread_mutex_t * mutex);
```

上面 PTHREAD_MUTEX_INITIALIZER 初始化的互斥量, 本质也需要调用 pthread_mutex_destroy, 但如果默认跟随系统生命周期, 可以不用调用. 对于 POSIX 线程, 假如调用了 pthread_xxx_init 那么非特殊情况最好都需要调用 pthread_xxx_destroy. 对于 pthread 我们包装一下.

thread.h

```c
#pragma once

#include <pthread.h>
#include <semaphore.h>

#include "struct.h"
#include "spinlock.h"

//
// pthread_async - 启动无需等待的线程
// frun     : node_f or ... 运行主体
// arg      : 运行参数
// return   : 0 is success, -1 is error
//
extern int pthread_async(void * frun, void * arg);

//
// thread_async - 强制启动无需等待线程
// frun     : node_f or ... 运行主体
// return   : void
//
inline void thread_async(void * frun) {
    IF(pthread_async(frun, NULL));
}

//
// pthread_run - 启动线程
// p        : 指向线程 id 的指针
// frun     : node_f or ... 运行主体
// arg      : 运行参数
// return   : 0 is success, -1 is error
//
extern int pthread_run(pthread_t * p, void * frun, void * arg);

//
// pthread_end - 等待线程运行结束
// tid      : 线程 id
// return   : void
//
inline void pthread_end(pthread_t id) {
    pthread_join(id, NULL);
}
```

讲的气功, 等同于练气期修炼的法术. 打通和操作系统联系的基本关节. 专业程序或多或少依赖后续可以通过 pthread_async 来启动设置好分离属性的线程. 如果想 window 上使用 POSIX pthread 线程库, 可以学习 github GerHobbelt 大神的 pthread-win32 项目. 从此以后, 你要的一切 pthread 都会给你! pthread 比 C11 thrd 好用很多, 线程这块推荐用 pthrad POSIX 标准, 坑少点.

thread.c

```c
#include "thread.h"

static pthread_attr_t attr;

static pthread_attr_t rtta;

//
// pthread_init - thread 使用方需要手动初始化
// attr     : pthread_attr_t 变量
// return   : void
//
void pthread_init(void) {
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_attr_init(&rtta);
}

//
// pthread_async - 启动无需等待的线程
// frun     : node_f or ... 运行主体
// arg      : 运行参数
// return   : 0 is success, -1 is error
//
inline int
pthread_async(void * frun, void * arg) {
    pthread_t id;
    return pthread_create(&id, &attr, frun, arg);
}


//
// pthread_run - 启动线程
// p        : 指向线程 id 的指针
// frun     : node_f or ... 运行主体
// arg      : 运行参数
// return   : 0 is success, -1 is error
//
inline int
pthread_run(pthread_t * p, void * frun, void * arg) {
    return pthread_create(p, &rtta, frun, arg);
}
```

为保护所爱的人去战斗 <->

3.2.1 pthread 练手

陆续 1. 运用搭建好的 pthread 模块, 写个 demo 练练手. 2. 使用了 pthread 读写锁相关操作. 用于熟悉 api 而构造演示示例.

```c
#include "thread.h"

struct rwarg {
    pthread_rwlock_t lock;  // 读写锁

    unsigned id;            // 标识

    // conf 配置
    struct {
        char * description; // 描述
    };
};

// write - 写线程，随机写字符
void write(struct rwarg * arg);
// reads - 读线程
void reads(struct rwarg * arg);

/*
 * 主函数测试线程读写逻辑
 * 少量写线程，大量读线程测试
 */
int main(int argc, char * argv[]) {
    // 初始化 rwarg::rwlock
    struct rwarg arg = {
        // 这种初始化只能用在对象生命周期和程序生命周期一致情况，不需要 destroy
        .lock = PTHREAD_RWLOCK_INITIALIZER,
        .description = "爱自己，爱家庭，爱亲友",
    };

    // 写程序跑起来
    pthread_async(reads, &arg);

    // 读线程跑起来
    for (int i = 0; i < 10; ++i)
        pthread_async(reads, &arg);

    // 写程序跑起来
    pthread_async(reads, &arg);

    // 简单等待一下
    puts("sleep input enter:");
    return getchar();
}

// write - 写线程，随机写字符
void
write(struct rwarg * arg) {
    pthread_rwlock_wrlock(&arg->lock);
    ++arg->id;
```

```
    arg->description = arg->id%2 ? "为人民服务" : "才刚刚开始";
    printf("write id[%u][%s]\n", arg->id, arg->description);
    pthread_rwlock_unlock(&arg->lock);
}

// reads - 读线程
void
reads(struct rwarg * arg) {
    pthread_rwlock_rdlock(&arg->lock);
    ++arg->id;
    printf("reads id[%u][%s]\n", arg->id, arg->description);
    pthread_rwlock_unlock(&arg->lock);
}
```

手握 pthread 神器随便写了上面点 demo. 关于 pthread rwlock 存在一个隐患就是 pthread_rwlock_unlock 这个 api. 也能看出来他不区分读解锁, 还是写解锁. 这就存在一个问题, 当大量读操作存在时候, 会极大降低写加锁机会的期望, 使写操作饥渴. 后面会带大家手写个读写锁, 用于感受一下远古时期那些妖魔大能弥留在天地之间, 万仞无边的意念 ~ 关于 POSIX 线程库 pthread 就到这里了. 看看头文件, 查查手册, 再不济看看源码一切仍然是那么自然.

3.3 读写锁

pthread 已经提供了读写锁, 为什么还要没事瞎搞呢. 对于这个问题我是这样解释的.

- 1 pthread 读写锁存在写操作饥渴的隐患, 写写顺便了解原子自旋锁的基本基本原理

特别是 3 问题很严重, 不妨自行构造多读少写情况自行验证. 下面对读写锁进行详细分析. 读写锁主要还是为了解决, 大量'读'和少量'写'的业务而设计的. 如果读写均衡, 那用 pthrad rwlock 效果更好. 我们这里写优先的读写锁满足 3 个特征:

- 1 当读写锁是写加锁状态时, 在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞
- 2 当读写锁在读加锁状态时, 再以读模式对他加锁的线程都能得到访问权，但以写模式加锁的线程将会被阻塞
- 3 当读写锁在读加锁状态时, 如果有线程试图以写模式加锁，读写锁通常会阻塞随后的读模式加锁.

从上面表述可以看出, pthread 的线程库对于特征 3 没有支持, 默认还是平等竞争. 这里写个默认写锁优先级高于读锁, 对其有遏制效果, 用于解决少量写饥渴问题.

3.3.1 读写锁练习

rwlock.c

```c
#include "struct.h"
#include "spinlock.h"

//
// 读写锁被用来保护经常需要读，但是不频繁修改修改的数据.
// 特性是：可重复读，互斥写.
// 传统 pthread 提供读写锁，读和写是交叉的，一段时间内读多那么读获取锁很容易，
// 同样写多的时候写获取锁很容易. 因而在读操作频繁时候，容易造成写饥渴.
//
// 我们这里实现是，读锁会压制写锁. 优势是少量写场景写锁不会被饥渴.
// 实现方面采用原子自旋方式，适用于业务短小快速高频领域.
//
// 没有一种方案会通杀所有场景，具体需要因地制宜.
// atomic, pthread mutex, pthread cond, 业务 之间取舍侧重.
// 最好方案还是通过业务来避免锁. 万不得已最稳方案还是成熟 pthread 提供相关 api 能力.
//
// 这里代码只是用于学习和交流场景上
//

// create atom write and read lock
// struct rwlock need zero is atom interface extend
// need var init struct rwlock rw = { 0,0 };
struct rwlock {
    atomic_bool wlock;
    atomic_long rlock;
};

// atomic_w_lock - add write lock
extern void atomic_w_lock(struct rwlock * rw) {
    // write lock
    while (!atomic_bool_compare_exchange_weak(&rw->wlock)) {}
    // 等待读锁释放
    while (atomic_load(&rw->rlock)) {}
}

// atomic_r_lock - add read lock
extern void atomic_r_lock(struct rwlock * rw) {
    for (;;) {
        // 等待读完毕，并设置内存屏障
        while (atomic_load(&rw->wlock)) {}

        // 乐观的添加读计数
        atomic_fetch_add(&rw->rlock, 1);

        // 没有写占用，就可以读了
        if (!atomic_load(&rw->wlock))
            break;

        // 还是有写，收回刚添加的读计数
        atomic_fetch_sub(&rw->rlock, 1);
```

```
    }
}

// atomic_w_unlock - unlock write lock
extern void atomic_w_unlock(struct rwlock * rw) {
    assert(atomic_load(&rw->wlock));
    atomic_store(&rw->wlock, false);
}

// atomic_r_unlock - unlock read lock
extern void atomic_r_unlock(struct rwlock * rw) {
    assert(atomic_load(&rw->rlock));
    atomic_fetch_sub(&rw->rlock, 1);
}

// atomic_r_trylock - try add read lock
extern bool atomic_r_trylock(struct rwlock * rw) {
    if (atomic_load(&rw->wlock))
        return false;

    // 乐观的添加读计数
    atomic_fetch_add(&rw->rlock, 1);

    if (atomic_load(&rw->wlock)) {
        // 还是有写，收回刚添加的读计数
        atomic_fetch_sub(&rw->rlock, 1);
        return false;
    }

    return true;
}

// atomic_w_trylock - try add write lock
extern bool atomic_w_trylock(struct rwlock * rw) {
    // 存在读锁，直接返回
    if (atomic_load(&rw->rlock)) {
        return false;
    }

    // 尝试抢占写锁
    if (!atomic_bool_compare_exchange_strong(&rw->wlock)) {
        return false;
    }

    // 存在读锁，释放申请到写锁，直接返回
    if (atomic_load(&rw->rlock)) {
        // 存在读锁，释放写锁
        atomic_store(&rw->wlock, false);
        return false;
    }
```

```
    return true;
}
```

通过 rwlock.c 可以看出来这里是分别对读和写进行加锁和解锁的. rwlock 中 rlock 和 wlock 两个字段就是直接表现, 本质通过两把交叉的锁模拟出一把读写锁. 来来回回, 虚虚实实, 互相打配合 ~

到这里关于读写锁的炫迈已经嚼完了. 读写锁应用场景也很窄, 例如配置中心用于解决配置读取和刷新可能会尝试使用. 读写锁用于学习原子操作特别酷炫, 但不推荐实战使用, 因为往往能很容易被更高效的业务设计或者使用安全性更高的优质库的 API 所替代 ~

3.4 阅读理解 - time.h 二次封装

独自在野外游历, 狭路遇大妖, 为保命不计后果决绝吃下小药丸, 疾飞而撤.

All knowledge is, in final analysis, history.

All sciences are, in the abstract, mathematics.

All judgements are, in their rationale, statistics.

这里赠送个 time 时间模块的阅读理解, 做为这个练气的功法的额外回馈. 重复一下下, 程序员的世界硬通货是数据结构和操作系统. 也许自然世界也是, 万物是数据结构, 宇宙是操作系统. 基础库中一定会有 time 相关的时间业务模块库. 例如业务常见字符串和时间戳来回转, 是否同一天, 同一周, 时间开始点什么鬼的. ok, 阅读理解开始吧!

times.h

```c
#pragma once

#include <time.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <inttypes.h>

//
// ~ 力求最小时间业务单元 ~
// 1s 秒 = 1000ms 毫秒 = 1000000us 微秒 = 1000000000ns 纳秒
//
// const (
//     Nanosecond  time_t   = 1
//     Microsecond          = 1000 * Nanosecond
//     Millisecond          = 1000 * Microsecond
//     Second               = 1000 * Millisecond
//     Minute               = 60 * Second
//     Hour                 = 60 * Minute
// )

#if defined(__linux__) && defined(__GNUC__)

#include <unistd.h>
#include <sys/time.h>

#endif

/* A year not divisible by 4 is not leap.
 * If div by 4 and not 100 is surely leap.
 * If div by 100 *and* not by 400 is not leap.
 * If div by 100 and 400 is leap. */
inline bool is_leap_year(time_t year) {
    return ((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 400 == 0));
}

/* We use a private localtime implementation which is fork-safe. The logging
 * function of Redis may be called from other threads. */
extern void localtime_get(struct tm * restrict p, time_t t);

// times_t - 时间串类型
// len = 33 + 1, buf = 2022-03-26 15:38:29.934089326 CST
// len = 53 + 1, buf = 2022年03月26日 15时38分29秒.934071664纳秒 CST
typedef char times_t[64];

//
// times_get - 解析时间串，返回时间戳
// ns       : 时间串内容
// out      : 返回得到的时间戳
// outm     : 返回得到的时间结构体
```

```
// return   : 返回 true 表示构造成功
//
extern bool times_get(times_t ns, time_t * out, struct tm * outm);

//
// time_get - 解析时间串, 返回时间戳
// ns       : 时间串内容
// return   : < 0 is error
//
extern time_t time_get(times_t ns);

//
// time_day_equal - 判断时间戳是否是同一天
// n        : 第一个时间戳
// t        : 第二个时间戳
// return   : true 表示同一天
//
extern bool time_day_equal(time_t n, time_t t);

//
// time_week_equal - 判断时间戳是否是同一周
// n        : 第一个时间戳
// t        : 第二个时间戳
// return   : true 表示同一周
//
extern bool time_week_equal(time_t n, time_t t);

//
// times_day_equal - 判断时间串是否是同一天
// ns       : 第一个时间串
// ts       : 第二个时间串
// return   : true 表示同一天
//
extern bool times_day_equal(times_t ns, times_t ts);

//
// times_week_equal - 判断时间串是否是同一周
// ns       : 第一个时间串
// ts       : 第二个时间串
// return   : true 表示同一周
//
extern bool times_week_equal(times_t ns, times_t ts);

// TIMES_FMT_STR - "{年}.{月}.{日}.{时}.{分}.{秒}.{纳秒} {时区}"
#define TIMES_FMT_STR "%04d-%02d-%02d %02d:%02d:%02d.%09ld %s"

//
// times_fmt - 通过 fmt 格式最终拼接一个字符串
// fmt      : 推荐遵循 TIMES_FMT_STR 意图
// out      : 最终保存的内容
// sz       : buf 长度
```

```
// return   : 返回生成串长度
//
int times_fmt(const char * fmt, char out[], size_t sz);

//
// times_buf - 存储带毫秒时间串 "2016-07-10 22:38:34 500"
// ns       : 返回生成串
// return   : 返回生成串长度
//
inline int times_buf(times_t ns) {
    return times_fmt(TIMES_FMT_STR, ns, sizeof(times_t));
}

// times TLS time str 版本
extern const char * times(void);
```

道友是否想起 localtime_r 函数. 这种函数带着浓浓的 linux api 的设计口味. 标识是可重入的. 这就扯到另一个常被误导的话题了可重入和线程安全. 可重入是基于操作系统中断层面的定义, 多数是系统发生中断瞎比调用这个函数仍然没有问题, 表示此函数可重入. 线程安全呢? 他是线程调度层面的定义, 希望多个线程之间瞎比调用这个函数, 程序最终运行结果仍然能够符合预期思路. 二者有相关性, 例如多数可线程安全的函数可重入. 但而二者是个完全不同的概念. 例如 malloc 内部通过锁来实现线程安全, 如果调用 malloc 过程中发生中断, 中断程序再次调用 malloc 那么两次 lock 操作会导致死锁. 因而有些线程安全的函数是不可重入. 另外一个例子就是一个函数打开文件读取内容这是可重入的, 但却不是线程安全的(文件读写非线程安全), 因为可重入函数不一定线程安全. 希望到这里你能对可重入和线程安全有个更清晰的认识. 那继续剖析上面的 times.h 接口设计. 推荐随后的代码可以全部拔到你的项目中. 他们也算是久经考验的忠诚战士. 首先看一个飘逸的字符串解析为系统时间结构的函数.

```c
// times_tm - 从时间串中提取出来年月日时分秒
bool times_tm(times_t ns, struct tm * outm) {
    if (ns == NULL) return false;

    int c = *ns;
    if (c == 0 || c < '0' || c > '9') return false;

    int num = 0;

    // https://en.cppreference.com/w/c/chrono/tm#cite_note-leapsecond-1
    // /* ISO C `broken-down time' structure.  */
    // struct tm
    // {
    //   int tm_sec;                   /* Seconds.     [0-60] (1 leap second) */
    //   int tm_min;                   /* Minutes.     [0-59] */
    //   int tm_hour;                  /* Hours.       [0-23] */
    //   int tm_mday;                  /* Day.         [1-31] */
    //   int tm_mon;                   /* Month.       [0-11] */
    //   int tm_year;                  /* Year - 1900.  */
    //   int tm_wday;                  /* Day of week. [0-6] */
    //   int tm_yday;                  /* Days in year.[0-365] */
    //   int tm_isdst;                 /* DST.         [-1/0/1]*/
    // };
    // 实现深度绑定 tm 结构结构，构建最小可用实体.
    // 有些字段没有过度处理，例如 tm_wday，tm_yday 和 tm_isdst
    int * es = &outm->tm_sec;
    int * py = &outm->tm_year;
    do {
        if (c >= '0' && c <= '9') {
            num = 10 * num + c - '0';
            c = *++ns;
            continue;
        }

        *py-- = num;
        if (py < es)
            break;

        // 去掉特殊字符，重新开始
        for (;;) {
            if ((c = *++ns) == 0)
                return false;
            if (c >= '0' && c <= '9')
                break;
        }
        num = 0;
    } while (c != 0);

    // 内存没有从 tm_year 解析到 tm_sec
    if (py > es) return false;
```

```
    if (py == es) {
        // 补上最后一个缺口
        *es = num;
    }
    outm->tm_mon -= 1;
    outm->tm_year -= 1900;
    // fix memory dirty data
    outm->tm_wday = outm->tm_yday = 0;
    outm->tm_isdst = 0;
    return true;
}

//
// times_get - 解析时间串，返回时间戳
// ns       : 时间串内容
// out      : 返回得到的时间戳
// outm     : 返回得到的时间结构体
// return   : 返回 true 表示构造成功
//
bool
times_get(times_t ns, time_t * out, struct tm * outm) {
    time_t t;
    struct tm m;

    // 先高效解析出年月日时分秒
    if (!times_tm(ns, &m))
        return false;

    // 得到时间戳，失败返回 false
    if ((t = mktime(&m)) < 0)
        return false;

    // 返回最终结果
    if (out) *out = t;
    if (outm) *outm = m;
    return true;
}

//
// time_get - 解析时间串，返回时间戳
// ns       : 时间串内容
// return   : < 0 is error
//
inline time_t
time_get(times_t ns) {
    struct tm m;
    // 先高效解析出年月日时分秒
    if (!times_tm(ns, &m)) {
        return -1;
    }
```

```
    // Window or Linux 正常处理, Window 11 WSL2 中库调用存在 BUG
    // 在 struct tm 中 tm_isdst 为夏令时时候, mktime return -1; 并且会修改 daylight
    //
    // 得到时间戳, < 0 标识失败
    return mktime(&m);
}
```

又好又快, 思路是将解析时间字符串, 分隔为一系列的数值. 再巧妙利用指针移位赋值. 继续看两个时间戳是否是同一天的小学生的数学分析.

```
//
// time_day_equal - 判断时间戳是否是同一天
// n         : 第一个时间戳 UTC
// t         : 第二个时间戳 UTC
// return    : true 表示同一天
//
inline bool
time_day_equal(time_t n, time_t t) {
    // UTC(世界协调时间)
    // 世界协调时间(UTC)与世界协调时间(UTC)没有时差.
    // CST(中国标准时间)
    // 中国标准时间(CST)比世界协调时间(UTC)早08:00小时. 该时区为标准时区时间, 主要用于 亚洲
    // UTC [World] + 8 * 3600 = CST [China] | UTC [World] = CST [China] - timezone (8 * 3600)
    // 其他地区也类似 UTC 和 CST 关系, 存在 timezone = UTC - LOC -> LOC = UTC - timezone
    n = (n - _timezone) / (24 * 3600);
    t = (t - _timezone) / (24 * 3600);
    return n == t;
}

//
// time_week_equal - 判断时间戳是否是同一周
// n         : 第一个时间戳
// t         : 第二个时间戳
// return    : true 表示同一周
//
bool
time_week_equal(time_t n, time_t t) {
    time_t p;
    struct tm m;
    // n = max(n, t), t = min(n, t)
    if (n < t) {
        p = n; n = t; t = p;
    }

    // 得到 n 表示的当前时间
    localtime_get(&m, n);
    // 得到当前时间到周一起点的时间差
    p = (time_t)(m.tm_wday ? m.tm_wday - 1 : 6) * 24 * 3600
      + (time_t)m.tm_hour * 3600
      + (time_t)m.tm_min * 60
      + m.tm_sec;

    // [week start, n], n = max(n, t), , week start = n - p
    // t = min(n, t) >= week start 表示在同一周内
    return t >= n - p;
}
```

8UL * 3600 科普一下, GMT(Greenwich Mean Time) 代表格林尼治标准时间, 也是咱们代码中 time(NULL) 返回的时间戳. 而中国北京标准时间采用的 CST(China Standard Time UT+8:00). 因而需要

在原先的标准时间戳基础上加上 8h, 就得到咱们中国皇城的时间戳. 说到时间业务上面, 推荐用新的标准函数 timespec_get 替代 传统特定的系统库的 gettimeofday! 精度更高, 更规范.

扩展一点, 假如有个策划奇葩需求, 我们规定一天的开始时间是 5 时 0 分 0 秒. 现实世界默认一天开始时间是 0 时 0 分 0 秒. 那你会怎么做呢？其实有很多处理方式, 只要计算好偏移量就可以. 例如我们假如在底层支持. 可以这么写.

```
#define DAYNEWSTART_INT (timezone + 5UL * 3600 + 0 * 60 + 0)

inline bool time_isday(time_t n, time_t t) {
    n = (n - DAYNEWSTART_INT) / (24 * 3600);
    t = (t - DAYNEWSTART_INT) / (24 * 3600);
    return n == t;
}
```

同样的对于如果判断是否是同一周什么鬼, 也是减去上面偏移量. 当然这样需求最好拒绝! 大家多写多用, 将吸星大法练习精深. 本书很多素材最初来自于写游戏服务器业务时感悟. 扯一点题外话, 游戏相比其他互联网项目而言, 开宝箱的几率很高. 技术上多数吃老本, 新技术落后. 业务上面增删改查不多. 整个行业偏重客户端和策划玩法. 那把剩下关于 times_t 操作补全.

```
//
// time_day_equal - 判断时间戳是否是同一天
// n         : 第一个时间戳 UTC
// t         : 第二个时间戳 UTC
// return    : true 表示同一天
//
inline bool
time_day_equal(time_t n, time_t t) {
    // UTC(世界协调时间)
    // 世界协调时间(UTC)与世界协调时间(UTC)没有时差.
    // CST(中国标准时间)
    // 中国标准时间(CST)比世界协调时间(UTC)早08:00小时. 该时区为标准时区时间, 主要用于 亚洲
    // UTC [World] + 8 * 3600 = CST [China] | UTC [World] = CST [China] - timezone (8 * 3600)
    // 其他地区也类似 UTC 和 CST 关系, 存在 timezone = UTC - LOC -> LOC = UTC - timezone
    n = (n - timezone) / (24 * 3600);
    t = (t - timezone) / (24 * 3600);
    return n == t;
}

//
// time_week_equal - 判断时间戳是否是同一周
// n         : 第一个时间戳
// t         : 第二个时间戳
// return    : true 表示同一周
//
bool
time_week_equal(time_t n, time_t t) {
    time_t p;
    struct tm m;
    // n = max(n, t), t = min(n, t)
    if (n < t) {
        p = n; n = t; t = p;
    }

    // 得到 n 表示的当前时间
    localtime_get(&m, n);
    // 得到当前时间到周一起点的时间差
    p = (time_t)(m.tm_wday ? m.tm_wday - 1 : 6) * 24 * 3600
      + (time_t)m.tm_hour * 3600
      + (time_t)m.tm_min * 60
      + m.tm_sec;

    // [week start, n], n = max(n, t), , week start = n - p
    // t = min(n, t) >= week start 表示在同一周内
    return t >= n - p;
}

//
// times_day_equal - 判断时间串是否是同一天
// ns        : 第一个时间串
```

```
// ts       : 第二个时间串
// return   : true 表示同一天
//
bool
times_day_equal(times_t ns, times_t ts) {
    time_t n = time_get(ns);
    if (n < 0) return false;
    time_t t = time_get(ts);
    return t < 0 ? false : time_day_equal(n, t);
}

//
// times_week_equal - 判断时间串是否是同一周
// ns       : 第一个时间串
// ts       : 第二个时间串
// return   : true 表示同一周
//
bool
times_week_equal(times_t ns, times_t ts) {
    time_t n = time_get(ns);
    if (n < 0) return false;
    time_t t = time_get(ts);
    return t < 0 ? false : time_week_equal(n, t);
}

//
// times_fmt - 通过 fmt 格式最终拼接一个字符串
// fmt      : 推荐遵循 TIMES_FMT_STR 意图
// out      : 最终保存的内容
// sz       : buf 长度
// return   : 返回生成串长度
//
int
times_fmt(const char * fmt, char out[], size_t sz) {
    struct tm m;
    struct timespec s;

    timespec_get(&s, TIME_UTC);
    localtime_get(&m, s.tv_sec);

    return snprintf(out, sz, fmt,
                    m.tm_year + 1900, m.tm_mon + 1, m.tm_mday,
                    m.tm_hour, m.tm_min, m.tm_sec,
                    s.tv_nsec, tzname[0]);
}

// times TLS time str 版本
extern const char * times(void) {
    static _Thread_local times_t out;

    struct tm m;
```

```
    struct timespec s;

    timespec_get(&s, TIME_UTC);
    localtime_get(&m, s.tv_sec);

    sprintf(out, TIMES_FMT_STR,
            m.tm_year + 1900, m.tm_mon + 1, m.tm_mday,
            m.tm_hour, m.tm_min, m.tm_sec,
            s.tv_nsec, tzname[0]);

    return out;
}
```

对于比较的问题, 用草纸画画涂涂就明白了. 其中使用的 **localtime_get** 从 redis 中扒下来, 阅读起来非常不错.

```c
#include "times.h"

/* This is a safe version of localtime() which contains no locks and is
 * fork() friendly. Even the _r version of localtime() cannot be used safely
 * in Redis. Another thread may be calling localtime() while the main thread
 * forks(). Later when the child process calls localtime() again, for instance
 * in order to log something to the Redis log, it may deadlock: in the copy
 * of the address space of the forked process the lock will never be released.
 *
 * This function takes the timezone 'tz' as argument, and the 'dst' flag is
 * used to check if daylight saving time is currently in effect. The caller
 * of this function should obtain such information calling tzset() ASAP in the
 * main() function to obtain the timezone offset from the 'timezone' global
 * variable. To obtain the daylight information, if it is currently active or not,
 * one trick is to call localtime() in main() ASAP as well, and get the
 * information from the tm_isdst field of the tm structure. However the daylight
 * time may switch in the future for long running processes, so this information
 * should be refreshed at safe times.
 *
 * Note that this function does not work for dates < 1/1/1970, it is solely
 * designed to work with what time(NULL) may return, and to support Redis
 * logging of the dates, it's not really a complete implementation. */
void
localtime_get(struct tm * restrict p, time_t t) {
    t -= timezone;                       /* Adjust for timezone. */
    // Different countries have different daylight saving time rules,
    // which are reserved for business
    // t += 3600 * daylight;                /* Adjust for daylight time. */
    int days = (int)(t / (3600 * 24));      /* Days passed since epoch. */
    int seconds = (int)(t % (3600 * 24));   /* Remaining seconds. */

    p->tm_isdst = daylight;
    p->tm_hour = seconds / 3600;
    p->tm_min = seconds % 3600 / 60;
    p->tm_sec = seconds % 3600 % 60;

    /* 1/1/1970 was a Thursday, that is, day 4 from the POV of the tm structure
     * where sunday = 0, so to calculate the day of the week we have to add 4
     * and take the modulo by 7. */
    p->tm_wday = (days + 4) % 7;

    /* Calculate the current year. */
    p->tm_year = 1970;
    for (;;) {
        /* Leap years have one day more. */
        int days_this_year = 365 + is_leap_year(p->tm_year);
        if (days_this_year > days) break;
        days -= days_this_year;
        p->tm_year++;
    }
```

```
    p->tm_yday = days;  /* Number of day of the current year. */

    /* We need to calculate in which month and day of the month we are. To do
     * so we need to skip days according to how many days there are in each
     * month, and adjust for the leap year that has one more day in February. */
    int mdays[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    mdays[1] += is_leap_year(p->tm_year);

    p->tm_mon = 0;
    while (days >= mdays[p->tm_mon]) {
        days -= mdays[p->tm_mon];
        p->tm_mon++;
    }

    p->tm_mday = days + 1; /* Add 1 since our 'days' is zero-based. */
    p->tm_year -= 1900;    /* Surprisingly tm_year is year-1900. */
}
```

大致意思说系统自带 localtime 甚至是 localtime_r 在 Redis 有些场景下也不安全, 会导致死锁. 例如: 父进程中同时有两个线程运行, 如果线程 1 调用 localtime_r 过程中, 线程 2 fork 一个子进程, 并且该子进程(例如打日志)也调用 localtime_r, 那么子进程会死锁. 因为进程地址空间重合(子进程继承父进程锁信息), 这个锁永远不会释放.

时间核心业务就带大家操练到这. 还有什么搞不定, 如果需要, 基于这些基础和思路再细细琢磨推敲 ~ 必然事半功倍. 最后提醒使用这个库需要事先 tzset 初始化时区, 夏令时等等信息.

```
main init
    // Now 'timezome' global is populated. Obtain timezone and daylight info.
    tzset();
```

3.5 拓展练习 - 链表定时器

我们讲解了数据结构 list, 原子操作 atomic, 多线程 pthread, 时间 time 等数据结构和系统的相关能力. 我们尝试基于这些能力带大家做个小练习, 写一个很傻定时器小练习. Let's go

```c
#include "list.h"
#include "times.h"
#include "thread.h"

// timer_node 定时器结点
struct timer_node {
$LIST
    int id;             // 定时器 id
    void * arg;         // 执行函数参数
    node_f ftimer;      // 执行的函数事件
    struct timespec t; // 运行的具体时间
};

// timer_node id compare
static inline int timer_node_id_cmp(int id,
                                    const struct timer_node * r) {
    return id - r->id;
}

// timer_node time compare 比较
static inline int timer_node_time_cmp(const struct timer_node * l,
                                      const struct timer_node * r) {
    if (l->t.tv_sec != r->t.tv_sec)
        return (int)(l->t.tv_sec - r->t.tv_sec);
    return (int)(l->t.tv_nsec - r->t.tv_nsec);
}

// timer_list 链表对象管理器
struct timer_list {
    atomic_int id;            // 当前 timer node id
    atomic_flag lock;         // 自旋锁
    volatile bool status;     // true is thread loop, false is stop
    struct timer_node * list; // timer list list
};

// 定时器管理单例对象
static struct timer_list timer = { .id = 1, .lock = ATOMIC_FLAG_INIT };

// get atomic int 1 -> INT_MAX -> 1
static inline int timer_list_id() {
    // 0 -> INT_MAX -> INT_MIN -> 0
    int id = atomic_fetch_add(&timer.id, 1) + 1;
    if (id < 0) {
        // INT_MAX + 1 -> INT_MIN
        // 0x7F FF FF FF + 1 -> 0x80 00 00 00

        // INT_MIN & INT_MAX => 0x80 00 00 00 & 0x7F FF FF FF => 0x00 00 00 00
        // id = atomic_fetch_and(&timer.id, INT_MAX) & INT_MAX;
        // Multiple operations atomic_fetch_and can ensure timer.id >= 0
        atomic_fetch_and(&timer.id, INT_MAX);
```

```
        // again can ensure id >= 1
        id = atomic_fetch_add(&timer.id, 1) + 1;
    }
    return id;
}

// timer_list_sus - 得到等待的微秒事件, <= 0 表示可以执行
inline int timer_list_sus(struct timer_list * tl) {
    struct timespec * v = &tl->list->t, t[1];
    timespec_get(t, TIME_UTC);
    return (int)((v->tv_sec  - t->tv_sec ) * 1000000 +
                 (v->tv_nsec - t->tv_nsec) / 1000);
}

// timer_list_run_node - 线程安全, 需要再 loop 之后调用
inline void timer_list_run_node(struct timer_list * tl) {
    atomic_flag_lock(&tl->lock);
    struct timer_node * node = tl->list;
    tl->list = list_next(node);
    atomic_flag_unlock(&tl->lock);

    node->ftimer(node->arg);
    free(node);
}

// 运行的主 loop, 基于 timer 管理器
static void timer_list_run(struct timer_list * tl) {
    // 正常轮循, 检查时间
    while (tl->list) {
        int sus = timer_list_sus(tl);
        if (sus > 0) {
            usleep(sus);
            continue;
        }

        timer_list_run_node(tl);
    }

    // 已经运行结束
    tl->status = false;
}

// timer_new - timer_node 定时器结点构建
static struct timer_node * timer_new(int s, node_f ftimer, void * arg) {
    struct timer_node * node = malloc(sizeof(struct timer_node));
    node->id = timer_list_id();
    node->arg = arg;
    node->ftimer = ftimer;
    timespec_get(&node->t, TIME_UTC);
    node->t.tv_sec += s / 1000;
```

```
    // nano second
    node->t.tv_nsec += (s % 1000) * 1000000;
    return node;
}

//
// timer_add - 添加定时器事件
// ms         : 执行间隔毫秒, <= 0 表示立即执行
// ftimer     : node_f 定时器行为
// arg        : 定时器参数
// return     : 定时器 id, < 0 标识 error
//
int timer_add(int ms, void * ftimer, void * arg) {
    int id;
    struct timer_node * node;

    if (ms <= 0) {
        ((node_f)ftimer)(arg);
        return 0;
    }

    node = timer_new(ms, ftimer, arg);
    id = node->id;

    atomic_flag_lock(&timer.lock);

    list_add(&timer.list, timer_node_time_cmp, node);

    // 判断是否需要开启新的线程
    if (!timer.status) {
        if (pthread_async(timer_list_run, &timer)) {
            PERR("pthread_async id = %d", id);
            id = -1;
        } else {
            timer.status = true;
        }
    }

    atomic_flag_unlock(&timer.lock);

    // 尝试释放资源
    if (id == -1)
        free(node);
    return id;
}

//
// timer_del - 删除定时器事件
// id         : 定时器 id
// return     : void
//
```

```
void timer_del(int id) {
    struct timer_node * node;

    if (id <= 0 || timer.list == NULL) {
        POUT("id = %d, list = %p", id, timer.list);
        return;
    }

    atomic_flag_lock(&timer.lock);
    node = list_pop(&timer.list, timer_node_id_cmp, (void *)(intptr_t)id);
    atomic_flag_unlock(&timer.lock);

    free(node);
}
```

核心设计思路 **struct timer_list** 设计, 通过管理 **struct timer_node * list** 定时器链表. 代码整体非常适合做练习作业.

```
// timer_list 链表对象管理器
struct timer_list {
    atomic_int id;            // 当前 timer node id
    atomic_flag lock;         // 自旋锁
    volatile bool status;     // true is thread loop, false is stop
    struct timer_node * list; // timer list list
};

// 定时器管理单例对象
static struct timer_list timer = { .id = 1, .lock = ATOMIC_FLAG_INIT };

// get atomic int 1 -> INT_MAX -> 1
static inline int timer_list_id() {
    // 0 -> INT_MAX -> INT_MIN -> 0
    int id = atomic_fetch_add(&timer.id, 1) + 1;
    if (id < 0) {
        // INT_MAX + 1 -> INT_MIN
        // 0x7F FF FF FF + 1 -> 0x80 00 00 00

        // INT_MIN & INT_MAX => 0x80 00 00 00 & 0x7F FF FF FF => 0x00 00 00 00
        // id = atomic_fetch_and(&timer.id, INT_MAX) & INT_MAX;
        // Multiple operations atomic_fetch_and can ensure timer.id >= 0
        atomic_fetch_and(&timer.id, INT_MAX);

        // again can ensure id >= 1
        id = atomic_fetch_add(&timer.id, 1) + 1;
    }
    return id;
}
```

以上 timer.c 模块实现思路, 核心是利用 list 构建了一个升序链表, 通过额外异步分离线程 loop 获取链表结点去执行. **timer_list_id** 中 **atomic_fetch_add** 和 **atomic_fetch_and** 设计非常有意思, 前者保证原子自增, 后者保证 **>= 0**.

```
拓展举例，int 4 字节 32 位二进制(补码)，最终转成 10 进制值计算公式如下，多细品
_________________(2)
x1 x2 ... x31 x32 = -x1*2^(32-1) + x2*2^(32-2) + ... + x31*2^(32-31) + x32*2^(32-32)
```

定时器一个通病, 不要放入阻塞函数, 容易失真. timer 使用方面也很简单, 例如一个技能, 吟唱 1s, 持续伤害 2s. 构造如下:

```
struct skills {
        int id;
        bool exist; // 实战走状态机，true 表示施法状态中
};

// 007 号技能 火球术，没有释放
struct skills fireball = { 007, false };

static void skills_end(struct skills * kill) {
        ...
        if (kill.id == fireball.id) {
                puts("火球术持续输出结束...");
                kill.exist = false;
        }
        ...
}

static void continued(struct skills * kill) {
        ...
        if (kill.id == fireball.id) {
                puts("火球术吟唱成功，开始持续输出...");
                kill.exist = true;
                timer_add(2000, skills_end, kill);
        }
        ...
}

static void start(struct skills * kill) {
        ...
        if (kill.id == fireball.id) {
                puts("火球术开始吟唱...");
                kill.exist = false;
                timer_add(1000, continued, kill);
        }
        ...
}
```

调用 start 就可以了, 火球术吟唱, 持续输出. 中间打断什么鬼, 那就自己扩展. 后期根据标识统一绘制显示. 以上是简单到吐的思路说不定也很有效. 有点像优化过的 select 特定的时候出其不意 ~

对于定时器常见的实现有三类套路. 一种是有序链表用于解决, 大量重复轮询的定时结点设计的. 另一种是采用时间堆构建的定时器, 例如小顶堆, 时间差最小的在堆顶, 最先执行. 还有一种时间片结构, 时间按照一定颗度转呀转, 转到那就去执行那条刻度上的链表. 总的而言定时器的套路取舍得看应用的场景. 这篇 timer 阅读理解是基于有序链表. 可以说起缘 list, 终于 list. 希望这篇拓展练习能加深你对当前所学知识点的掌握和运用. 多想想多实操, 很多实战交给系统库或者优质库去做这块.

3.6 展望

这章目的是为了大家对系统编程有一点点感受. 先给大家抛砖引玉, 试图解开开发中基础操作奥秘. 学会一个优质平台中一种方法, 应对不同平台的封装策略可学可不学看自己兴趣. 为以后步入练气大圆满, 漫天空气炮和偏地是坑铺展一个好的开始 ~ 同样在心里希望, 多陪陪爱我们的人和我们爱的人, 房子票子那种法宝有最好, 没有也不影响 **你所求的道** -

以梦为马(节选)
海子

面对大河我无限惭愧

我年华虚度 空有一身疲倦

和所有以梦为马的诗人一样

岁月易逝 一滴不剩