线程学习

--16级成员 高嘉两 ≫

目录

CONTENTS

1.进程是什么，为什么要有进程，有fork,有exec?

2.线程又是什么，为什么有了进程还要有线程?

简单来说，进程 fork就是为了让多个程序一起运行起来。提高资源的利用率和系统的吞吐量。

也可以让一个程序自己变成多个同时顺序进行，并发的执行任务，提高效率。

再引入线程概念，则是为了减少程序在并发执行时所付出的时空开销，使操作系统具有更好的并发行。

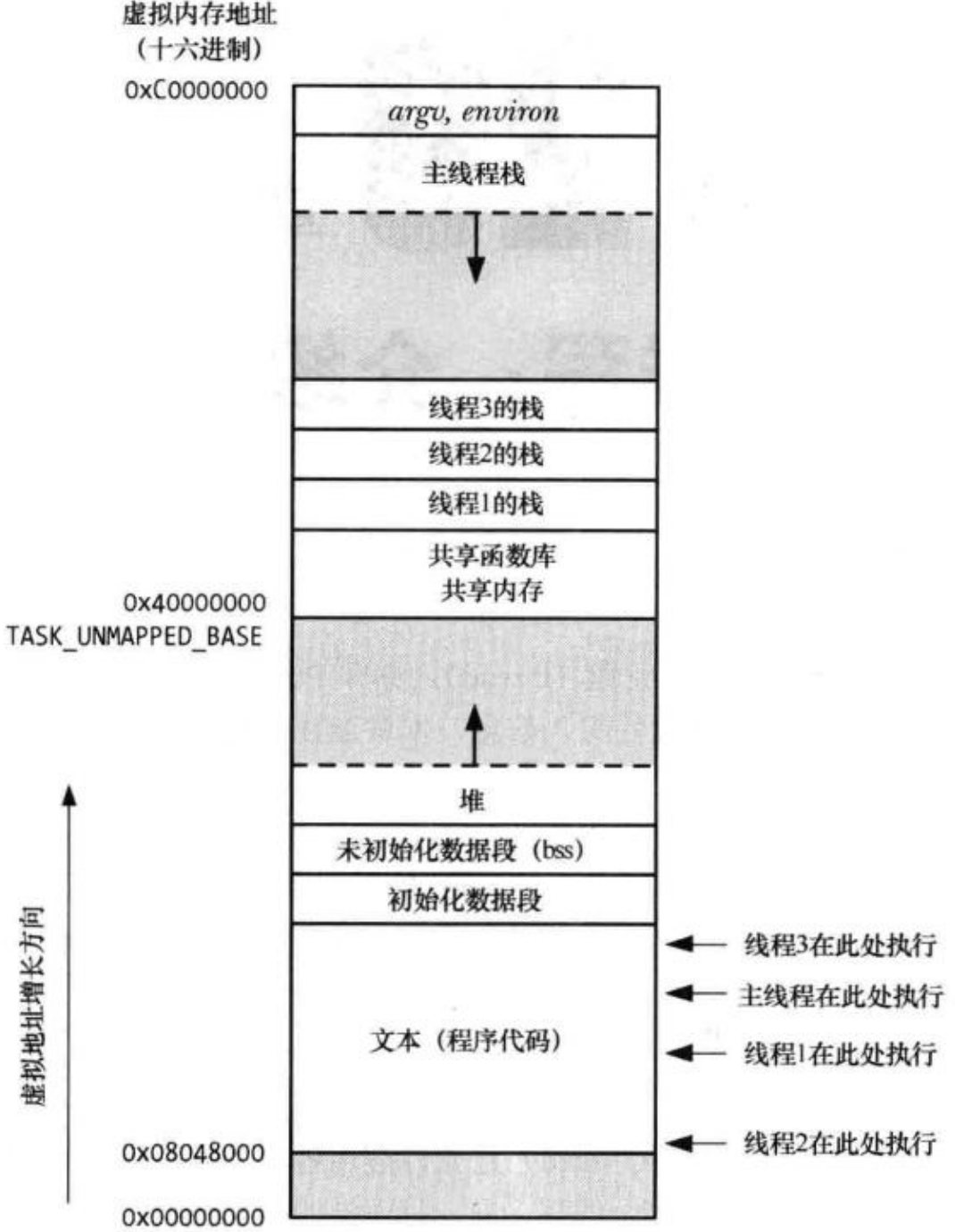


图 29-1：同时执行 4 个线程的进程（Linux/x86-32）

优劣异同

进程

操作系统调度切换多个线程要比切换调度进程在速度上快的多。而且进程间内存无法共享，通讯也比较麻烦。不适合大量通讯。

在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。

每个进程互相独立，地址空间大，不影响主程序的稳定性，子进程崩溃没关系。

逻辑控制复杂，需要和主程序交互。

线程

线程间方便的通信机制，同一进程下的线程之间共享数据空间，快捷方便

线程可以被认为是“轻量级”的进程。相比于进程，线程之间有比较麻烦的同步和加锁控制。

线程能够提高的总性能有限，而且线程多了之后，线程本身的调度和锁机制本身就很耗费资源。

线程和进程各有优缺点，编程的时候我们到底是用多线程还是多进程。不能一概而论，线程和进程各有存在的目的和意义，要根据自己程序的需求来选择。火狐多线程，谷歌多进程。chrome每一个网页标签就用一个进程来处理，所以某个标签崩溃了不会导致整个浏览器崩溃，这是选择进程的优点。缺点也是显而易见的，开一堆网页后，发现系统有一堆的进程，电脑就开始卡。

同时也和操作系统平台本身实现有关，Linux的线程发展是由进程而来，线程的底层实现和进程一样，相比于Windows，linux的线程开销就稍微大一些。

创建到结束

建立　　fork()
　　vfork()
　　pthrad(pthread_t *,pthread_attr_t,
　　void*(*)(void *),void * arg)

终止　　return , exit(), _exit(), pthread_exit()

获取返回值　wait(int *) waitpid(pid_t ,int *,int) pthred_join,pthread_detach

获取属性...

属性设置...

互斥锁,读写锁

互斥锁只有两种状态：加锁状态、不加锁状态；

读写锁有三种状态：读模式下加锁状态，写模式下加锁状态，不加锁状态；

```
int pthread_mutex_lock(pthread_mutex_t *);
int pthread_mutex_trylock(pthread_mutex_t *);
int pthread_mutex_unlock(pthread_mutex_t *);
```

信号量

pthread_cond_init
pthread_cond_destroy
pthread_cond_signal
pthread_cond_broadcast
pthread_cond_wait
pthread_cond_timedwait

errno

该变量用于保存程序的错误码，如果程序执行正确，则该变量不会更新。这些错误码通常是被定义在errno.h中以E开头的宏。

errno可以把最后一次调用C的错误代码保留，如果最后一次调用C函数成功，则不会更改errno。

```
char * strerror(int errno)
void perror(const char *s)
```

临界资源

虽然多个进程可以共享系统中的各种资源，但其中许多资源一次只能为一个进程所使用，我们把一次仅允许一个进程使用的资源称为临界资源。许多物理设备都属于临界资源，如打印机等。此外，还有许多变量、数据等都可以被若干进程共享，也属于临界资源。

临界区

进程中访问临界资源的那段代码就叫临界区。

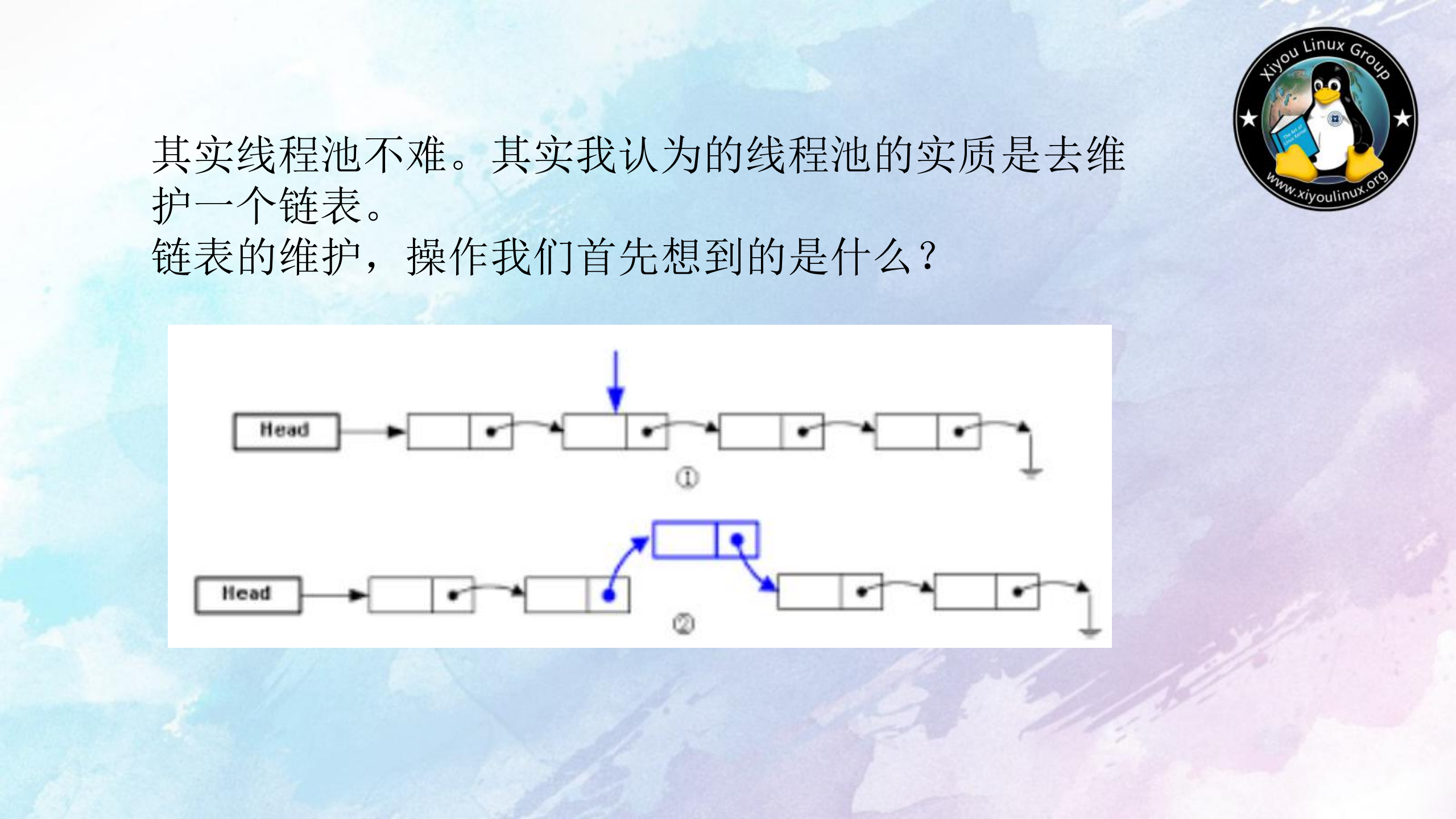
对临界资源的访问，必须互斥地进行

全局变量的访问问题；
死锁问题；

线程安全

当多个线程访问同一个资源时，如果不用考虑这些线程在运行时环境下的调度和交替运行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个资源的行为都可以获取正确的结果，那这个资源是线程安全的。

线程池



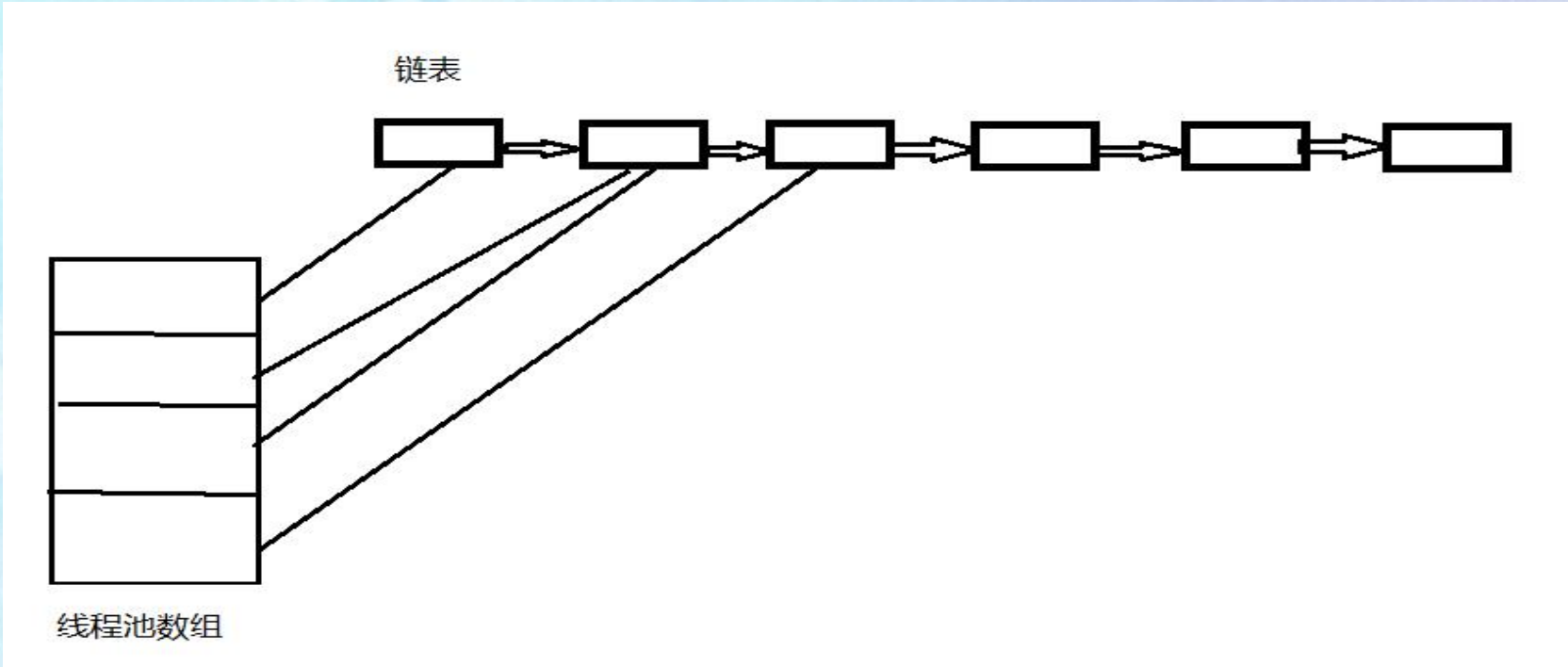
其实线程池不难。其实我认为的线程池的实质是去维护一个链表。

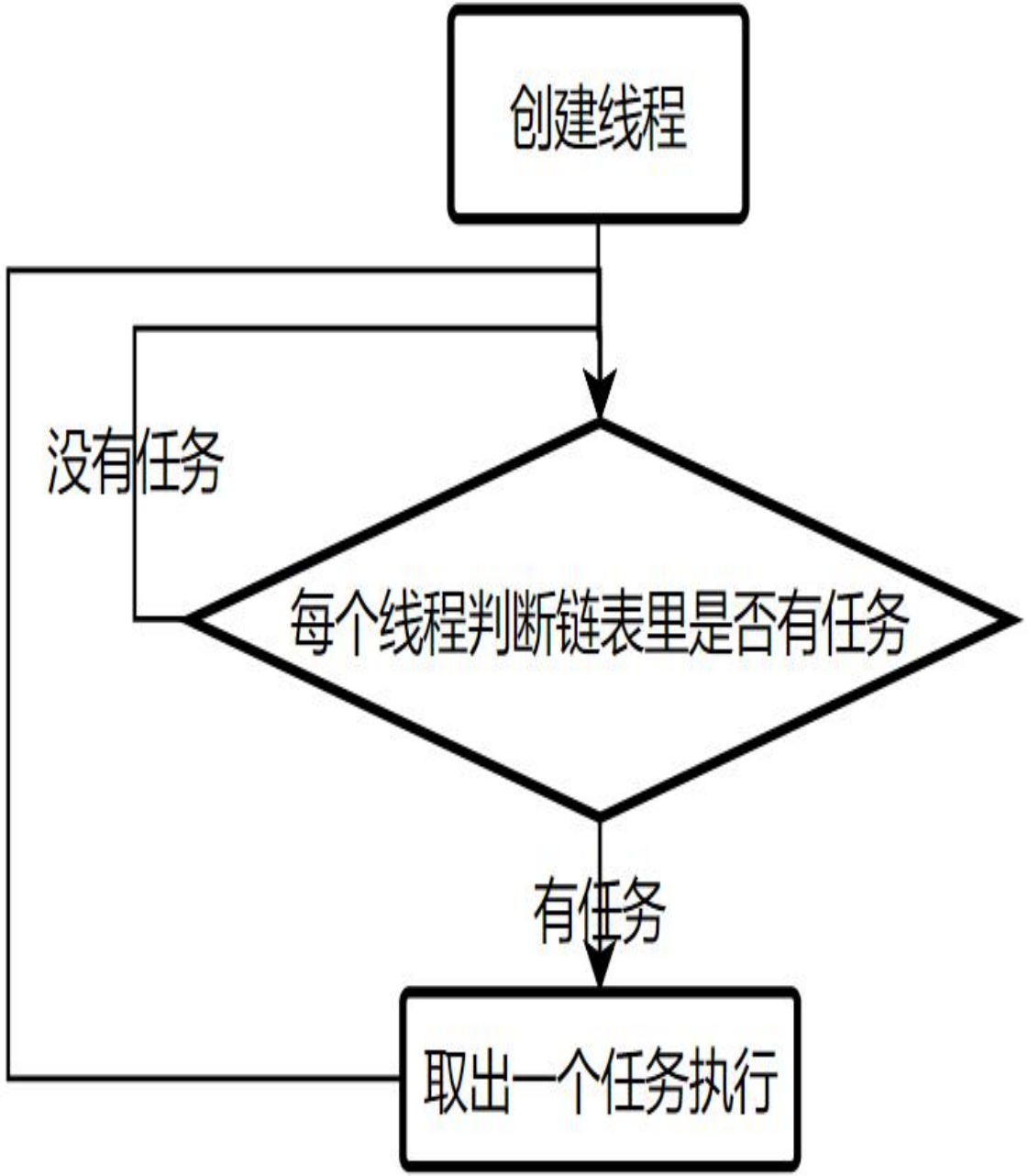
链表的维护，操作我们首先想到的是什么？

对就是增删改查排序等等。只不过我们现在是多个线程去维护他，多个线程一起的去对链表进行，增 删 改查。

那么问题又来了，多个线程同时进行增删改查造成什么后果呢?

如果要解决怎么解决呢?





既然是多线程就来看看线程是如何管理这个链表的.

很简单就是死循环去判断列表里面有没有东西，如果有就取出链表里面一个任务，然后执行，执行完再去看有没有任务.

死循环效率不高，使用信号量？

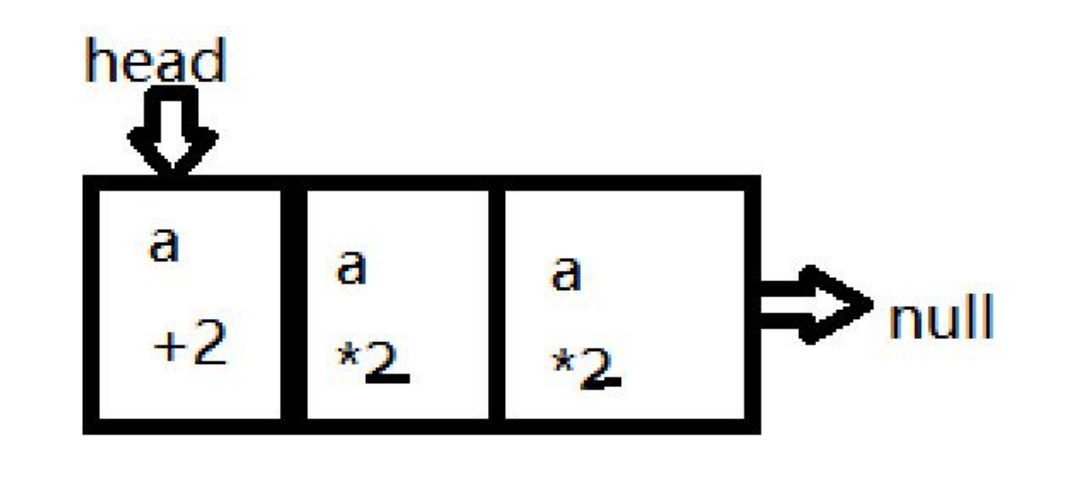
多线程取任务加锁

再回来看任务链表，我们一直在说任务链表，但这个链表里的结构体到底是什么呢？

假如我们有如下两种任务:

```
void add      (int a) { P(a+2); R; }
void multipy (int a) {P(a*2); R; }
```

这样我们的结构体里面应该有哪些成员？



再继续看刚刚我们的线程池,定义了很多东西,我们为什么不将这个池子也做一个结构体,对池子进行统一的维护.


```
typedef struct
{
    pthread_mutex_t queue_lock; //锁；
    pthread_cond_t queue_cond;  //条件变量；
    pthread_worker * head;  //任务链表头
    pthread_t *queue_pthread;//线程指针
    int max_pthread;    //最大线程数量
    int sum_queue_work;//任务对列中任务个数；

    int destroy;          //是否销毁池子标记
}pthread_pool;
```

```
typedef struct worker//任务通用结构体
{
    void *(*work)(void *argv);  //执行函数；
    void argv;                  //传入的参数；
    struct worker * next;       //链表；
}pthread_worker;
```

上面有一个细节:if,whlie
pthread_cond_signal在多处理器上可能同时唤醒多个线程，其实有些实现为了简单在单处理器上也会唤醒多个线程. 规范要求pthread_cond_signal至少唤醒一个pthread_cond_wait上 的线程，

花了半天时间,终于把线程池学了一遍,可是回头想想我们为什么非要用线程池呢?
我们就不能来一个任务建一个线程去执行,执行完退掉.不是也很方便吗?
比如刚刚那个需求:

```
{
        S(a,f);
        if(f==1)    pthread_create(&pt,NULL,(void *)add,(void)a);
        if(f==2)    pthread_create(&pt,NULL,(void *)multipy ,(void)a);
}
```

为什么非要用一个线程池呢?

线程池&池化思想

线程池中预先创建了一些工作线程，他们不断的从工作队列中取出任务，然后执行，当执行完之后，会继续执行工作队列中的下一个任务，减少了创建和销毁线程的次数，每个线程都可以一直被重用，避免了创建和销毁的开销。另外，可以根据系统的实际承载能力，方便的调节线程池中线程的数目，防止因为消耗过量的系统资源而导致系统崩溃的问题。

活动的线程也消耗系统资源，如果线程的创建数量没有限制，当大量的客户连接服务器的时候，就会创建出大量的工作线程，他们会消耗大量的内存空间，导致系统的内存空间不足，影响服务器的使用。

所以是否使用线程池,线程池初始开多大,都是我们在具体任务需求还要细酌.更有甚者将线程池的线程数设置为动态的,弹性的,当任务数远多余进程数我们给线程池添加线程,当任务数过小,我们适当的释放一些线程.

THANKS