同步概念

所谓同步，即同时起步，协调一致。不同的对象，对“同步”的理解方式略有不同。如，设备同步，是指在两个设备之间规定一个共同的时间参考；数据库同步，是指让两个或多个数据库内容保持一致，或者按需要部分保持一致；文件同步，是指让两个或多个文件夹里的文件保持一致。等等

而，编程中、通信中所说的同步与生活中大家印象中的同步概念略有差异。“同”字应是指协同、协助、互相配合。主旨在协同步调，按预定的先后次序运行。

线程同步

同步即协同步调，按预定的先后次序运行。

线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能。

举例 1： 银行存款 5000。柜台，折：取 3000；提款机，卡：取 3000。剩余：2000

举例 2： 内存中 100 字节，线程 T1 欲填入全 1， 线程 T2 欲填入全 0。但如果 T1 执行了 50 个字节失去 cpu，T2 执行，会将 T1 写过的内容覆盖。当 T1 再次获得 cpu 继续 从失去 cpu 的位置向后写入 1，当执行结束，内存中的 100 字节，既不是全 1，也不是全 0。

产生的现象叫做“与时间有关的错误”(time related)。为了避免这种数据混乱，线程需要同步。

“同步”的目的，是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间等等都需要同步机制。

因此，**所有“多个控制流，共同操作一个共享资源”的情况，都需要同步。**

数据混乱原因：

1. 资源共享（独享资源则不会）
2. 调度随机（意味着数据访问会出现竞争）
3. 线程间缺乏必要的同步机制。

以上 3 点中，前两点不能改变，欲提高效率，传递数据，资源必须共享。只要共享资源，就一定会出现竞争。只要存在竞争关系，数据就很容易出现混乱。

所以只能从第三点着手解决。使多个线程在访问共享资源的时候，出现互斥。

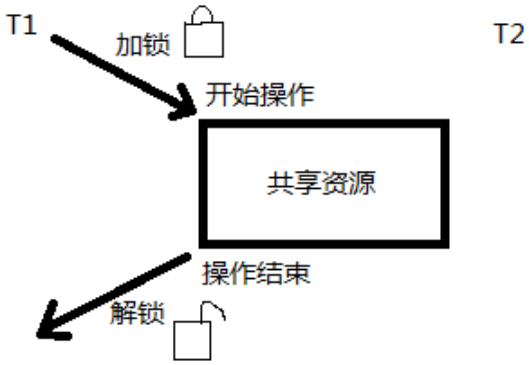
互斥量 mutex

Linux 中提供一把互斥锁 mutex（也称之为互斥量）。

每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的，

但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。



但，应注意：同一时刻，只能有一个线程持有该锁。

当 A 线程对某个全局变量加锁访问，B 在访问前尝试加锁，拿不到锁，B 阻塞。C 线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上是操作系统提供的一把“建议锁”（又称“协同锁”），建议程序中有多线程访问共享资源的时候使用该机制。但，并没有强制限定。

因此，即使有了 mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

主要应用函数：

pthread_mutex_init 函数

pthread_mutex_destroy 函数

pthread_mutex_lock 函数

pthread_mutex_trylock 函数

pthread_mutex_unlock 函数

以上 5 个函数的返回值都是：成功返回 0， 失败返回错误号。

pthread_mutex_t 类型，其本质是一个结构体。为简化理解，应用时可忽略其实现细节，**简单当成整数看待。**

pthread_mutex_t mutex; 变量 mutex 只有两种取值 1、0。

pthread_mutex_init 函数

初始化一个互斥锁(互斥量) ---> 初值可看作 1

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);

参 1：传出参数，调用时应传 &mutex

restrict 关键字：只用于限制指针，告诉编译器，所有修改该指针指向内存中内容的操作，只能通过本指针完成。不能通过除本指针以外的其他变量或指针修改

参 2：互斥量属性。是一个传入参数，通常传 NULL，选用默认属性(线程间共享)。 参 APUE.12.4 同步属性

1. 静态初始化：如果互斥锁 mutex 是静态分配的（定义在全局，或加了 static 关键字修饰），可以直接使用宏进行初始化。e.g. pthead_mutex_t muetx = PTHREAD_MUTEX_INITIALIZER;

2. 动态初始化：局部变量应采用动态初始化。e.g. pthread_mutex_init(&mutex, NULL)

pthread_mutex_destroy 函数

销毁一个互斥锁

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_lock 函数

加锁。可理解为将 **mutex--**（或 -1），操作后 mutex 的值为 0。

int pthread_mutex_lock(pthread_mutex_t *mutex);

pthread_mutex_unlock 函数

解锁。可理解为将 **mutex ++**（或 +1），操作后 mutex 的值为 1。

int pthread_mutex_unlock(pthread_mutex_t *mutex);

pthread_mutex_trylock 函数

尝试加锁

int pthread_mutex_trylock(pthread_mutex_t *mutex);

加锁与解锁

lock 与 unlock：

lock 尝试加锁，如果加锁不成功，线程阻塞，阻塞到持有该互斥量的其他线程解锁为止。

unlock 主动解锁函数，**同时将阻塞在该锁上的所有线程全部唤醒**，至于哪个线程先被唤醒，取决于优先级、调度。默认：先阻塞、先唤醒。

例如：T1 T2 T3 T4 使用一把 mutex 锁。T1 加锁成功，其他线程均阻塞，直至 T1 解锁。T1 解锁后，T2 T3 T4 均被唤醒，并自动再次尝试加锁。

可假想 mutex 锁 init 成功初值为 1。lock 功能是将 mutex--。而 unlock 则将 mutex++。

lock 与 trylock：

lock 加锁失败会阻塞，等待锁释放。

trylock 加锁失败直接返回错误号（如：EBUSY），不阻塞。

加锁步骤测试：

看如下程序：该程序是非常典型的，由于共享、竞争而没有加任何同步机制，导致产生于时间有关的错误，造成数据混乱：

1、 pthread_mutex_t lock; ----->全局变量
2、 pthread_mutex_init(&lock, NULL);
3、 pthread_mutex_lock(&lock);
4、 访问共享数据
5、 pthread_mutex_unlock(&lock);
6、 pthread_mutex_destroy(&lock);

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *tfn(void *arg)
{
    srand(time(NULL));
    while (1) {

        printf("hello ");
        sleep(rand() % 3);  /*模拟长时间操作共享资源，导致 cpu 易主，产生与时间有关的错误*/
        printf("world\n");
        sleep(rand() % 3);
    }
    return NULL;
}
int main(void)
{
    pthread_t tid;
    srand(time(NULL));
    pthread_create(&tid, NULL, tfn, NULL);
    while (1) {
        printf("HELLO ");
        sleep(rand() % 3);
        printf("WORLD\n");
        sleep(rand() % 3);
    }
    pthread_join(tid, NULL);
    return 0;
}
```

【mutex.c】

【练习】：修改该程序，使用 mutex 互斥锁进行同步。

1. 定义全局互斥量，初始化 init(&m, NULL)互斥量，添加对应的 destry
2. 两个线程 while 中，两次 printf 前后，分别加 lock 和 unlock
3. 将 unlock 挪至第二个 sleep 后，发现交替现象很难出现。

   线程在操作完共享资源后本应该立即解锁，但修改后，线程抱着锁睡眠。睡醒解锁后又立即加锁，这两个库函数本身不会阻塞。

   所以在这两行代码之间失去 cpu 的概率很小。因此，另外一个线程很难得到加锁的机会。
4. main 中加 flag = 5 将 flg 在 while 中-- 这时，主线程输出 5 次后试图销毁锁，但子线程未将锁释放，无法完成。

5. main 中加 pthread_cancel()将子线程取消。 【pthrd_mutex.c】

结论：

1、在访问共享资源前加锁，访问结束后立即解锁。锁的“粒度”应越小越好。
2、互斥锁，本质是结构体。我们可以看成整数。初值为1
3、加锁：--操作，阻塞线程。
解锁：++操作，唤醒阻塞在锁上的线程。
4、try锁：尝试加锁，成功--， 失败则返回，同时设置错误号 EBUSY

死锁

1. 线程试图对同一个互斥量 A 加锁两次。
2. 线程 1 拥有 A 锁，请求获得 B 锁；线程 2 拥有 B 锁，请求获得 A 锁

【作业】：编写程序，实现上述两种死锁现象。

读写锁

与互斥量类似，但读写锁允许更高的并行性。其特性为：**写独占，读共享，写锁优先级高。**

读写锁状态：

特别强调：读写锁只有一把，但其具备两种状态：

1. 读模式下加锁状态 (读锁)
2. 写模式下加锁状态 (写锁)

读写锁特性：

1. 读写锁是“写模式加锁”时， 解锁前，所有对该锁加锁的线程都会被阻塞。
2. 读写锁是“读模式加锁”时， 如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
3. 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高**

读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**

读写锁非常适合于对数据结构读的次数远大于写的情况。

主要应用函数：

pthread_rwlock_init 函数

pthread_rwlock_destroy 函数

pthread_rwlock_rdlock 函数

pthread_rwlock_wrlock 函数

pthread_rwlock_tryrdlock 函数

pthread_rwlock_trywrlock 函数

pthread_rwlock_unlock 函数

以上 7 个函数的返回值都是：成功返回 0， 失败直接返回错误号。

pthread_rwlock_t 类型 用于定义一个读写锁变量。

pthread_rwlock_t rwlock;

pthread_rwlock_init 函数

初始化一把读写锁

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);

参 2：attr 表读写锁属性，通常使用默认属性，传 NULL 即可。

pthread_rwlock_destroy 函数

销毁一把读写锁

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

pthread_rwlock_rdlock 函数

以读方式请求读写锁。（常简称为：请求读锁）

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

pthread_rwlock_wrlock 函数

以写方式请求读写锁。（常简称为：请求写锁）

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

pthread_rwlock_unlock 函数

解锁

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

pthread_rwlock_tryrdlock 函数

非阻塞以读方式请求读写锁（非阻塞请求读锁）

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

pthread_rwlock_trywrlock 函数

非阻塞以写方式请求读写锁（非阻塞请求写锁）

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

读写锁示例

看如下示例，同时有多个线程对同一全局数据读、写操作。

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int counter;
pthread_rwlock_t rwlock;

/* 3 个线程不定时写同一全局资源，5 个线程不定时读同一全局资源 */
void *th_write(void *arg)
{
	int t, i = (int)arg;
	while (1) {
		pthread_rwlock_wrlock(&rwlock);
		t = counter;
		usleep(1000);
		printf("=======write %d: %lu: counter=%d ++counter=%d\n", i, pthread_self(), t, ++counter);
		pthread_rwlock_unlock(&rwlock);
		usleep(10000);
	}
	return NULL;
}
void *th_read(void *arg)
{
	int i = (int)arg;

	while (1) {
		pthread_rwlock_rdlock(&rwlock);
		printf("----------------------------read %d: %lu: %d\n", i, pthread_self(), counter);
		pthread_rwlock_unlock(&rwlock);
		usleep(2000);
	}
	return NULL;
}
int main(void)
{
	int i;
	pthread_t tid[8];
```

```
    pthread_rwlock_init(&rwlock, NULL);

    for (i = 0; i < 3; i++)
        pthread_create(&tid[i], NULL, th_write, (void *)i);
    for (i = 0; i < 5; i++)
        pthread_create(&tid[i+3], NULL, th_read, (void *)i);
    for (i = 0; i < 8; i++)
        pthread_join(tid[i], NULL);

    pthread_rwlock_destroy(&rwlock);
    return 0;
}
```
【rwlock.c】

条件变量：

条件变量本身不是锁！但它也可以造成线程阻塞。通常与互斥锁配合使用。给多线程提供一个会合的场所。

主要应用函数：

pthread_cond_init 函数

pthread_cond_destroy 函数

pthread_cond_wait 函数

pthread_cond_timedwait 函数

pthread_cond_signal 函数

pthread_cond_broadcast 函数

以上 6 个函数的返回值都是：成功返回 0， 失败直接返回错误号。

pthread_cond_t 类型　用于定义条件变量

pthread_cond_t cond;

pthread_cond_init 函数

初始化一个条件变量

int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);

参 2：attr 表条件变量属性，通常为默认值，传 NULL 即可

也可以使用静态初始化的方法，初始化条件变量：

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_cond_destroy 函数

销毁一个条件变量

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_wait 函数

阻塞等待一个条件变量

int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

函数作用：

1. 阻塞等待条件变量 cond（参 1）满足
2. 释放已掌握的互斥锁（解锁互斥量）相当于 pthread_mutex_unlock(&mutex);

   1.2.两步为一个原子操作。
3. 当被唤醒，pthread_cond_wait 函数返回时，解除阻塞并重新申请获取互斥锁 pthread_mutex_lock(&mutex);

pthread_cond_timedwait 函数

限时等待一个条件变量

int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);

参 3： 参看 man sem_timedwait 函数，查看 struct timespec 结构体。

```
struct timespec {
    time_t tv_sec;      /* seconds */  秒
    long   tv_nsec;     /* nanosecondes*/  纳秒
}
```

形参 abstime：绝对时间。

如：time(NULL)返回的就是绝对时间。而 alarm(1)是相对时间，相对当前时间定时 1 秒钟。

```
struct timespec t = {1, 0};
pthread_cond_timedwait (&cond, &mutex, &t);  只能定时到 1970 年 1 月 1 日 00:00:01 秒(早已经过去)
```

正确用法：

```
time_t cur = time(NULL);  获取当前时间。
struct timespec t;   定义 timespec 结构体变量 t
t.tv_sec = cur+1;  定时 1 秒
pthread_cond_timedwait (&cond, &mutex, &t);  传参
```

参 APUE.11.6 线程同步条件变量小节

在讲解 setitimer 函数时我们还提到另外一种时间类型：

```
struct timeval {
    time_t      tv_sec;   /* seconds */  秒
    suseconds_t tv_usec;  /* microseconds */  微秒
};
```

pthread_cond_signal 函数

唤醒至少一个阻塞在条件变量上的线程

```
int pthread_cond_signal(pthread_cond_t *cond);
```

pthread_cond_broadcast 函数

唤醒全部阻塞在条件变量上的线程

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

生产者消费者条件变量模型

线程同步典型的案例即为生产者消费者模型，而借助条件变量来实现这一模型，是比较常见的一种方法。假定有两个线程，一个模拟生产者行为，一个模拟消费者行为。两个线程同时操作一个共享资源（一般称之为汇聚），生产向其中添加产品，消费者从中消费掉产品。

看如下示例，使用条件变量模拟生产者、消费者问题：

```
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct msg {
	struct msg *next;
	int num;
};
struct msg *head;

pthread_cond_t has_product = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *consumer(void *p)
{
	struct msg *mp;
	for (;;) {
		pthread_mutex_lock(&lock);
		while (head == NULL) {		//头指针为空,说明没有节点	可以为 if 吗
			pthread_cond_wait(&has_product, &lock);
		}
		mp = head;
		head = mp->next;			//模拟消费掉一个产品
		pthread_mutex_unlock(&lock);

		printf("-Consume ---%d\n", mp->num);
		free(mp);
```

```
            sleep(rand() % 5);
        }
    }
    void *producer(void *p)
    {
        struct msg *mp;
        while (1) {
            mp = malloc(sizeof(struct msg));
            mp->num = rand() % 1000 + 1;            //模拟生产一个产品
            printf("-Produce ---%d\n", mp->num);

            pthread_mutex_lock(&lock);
            mp->next = head;
            head = mp;
            pthread_mutex_unlock(&lock);

            pthread_cond_signal(&has_product);   //将等待在该条件变量上的一个线程唤醒
            sleep(rand() % 5);
        }
    }
    int main(int argc, char *argv[])
    {
        pthread_t pid, cid;
        srand(time(NULL));

        pthread_create(&pid, NULL, producer, NULL);
        pthread_create(&cid, NULL, consumer, NULL);

        pthread_join(pid, NULL);
        pthread_join(cid, NULL);
        return 0;
    }
```

【conditionVar_product_consumer.c】

条件变量的优点：

相较于 mutex 而言，条件变量可以减少竞争。

如直接使用 mutex，除了生产者、消费者之间要竞争互斥量以外，消费者之间也需要竞争互斥量，但如果汇聚（链表）中没有数据，消费者之间竞争互斥锁是无意义的。有了条件变量机制以后，只有生产者完成生产，才会引起消费者之间的竞争。提高了程序效率。

信号量

进化版的互斥锁（1 --> N）

由于互斥锁的粒度比较大，如果我们希望在多个线程间对某一对象的部分数据进行共享，使用互斥锁是没有办法实现的，只能将整个数据对象锁住。这样虽然达到了多线程操作共享数据时保证数据正确性的目的，却无形中导

致线程的并发性下降。线程从并行执行，变成了串行执行。与直接使用单进程无异。

信号量，是相对折中的一种处理方式，既能保证同步，数据不混乱，又能提高线程并发。

主要应用函数：

sem_init 函数

sem_destroy 函数

sem_wait 函数

sem_trywait 函数

sem_timedwait 函数

sem_post 函数

以上 6 个函数的返回值都是：成功返回 0， 失败返回-1，同时设置 errno。(注意，它们没有 pthread 前缀)

sem_t 类型，本质仍是结构体。但应用期间可简单看作为整数，忽略实现细节（类似于使用文件描述符）。

sem_t sem; 规定信号量 sem 不能 < 0。头文件 <semaphore.h>

信号量基本操作：

sem_wait: 1. 信号量大于 0，则信号量-- （类比 pthread_mutex_lock）
| 2. 信号量等于 0，造成线程阻塞
对应
|
sem_post： 将信号量++，同时唤醒阻塞在信号量上的线程 （类比 pthread_mutex_unlock）

但，由于 sem_t 的实现对用户隐藏，所以所谓的++、--操作只能通过函数来实现，而不能直接++、--符号。

信号量的初值，决定了占用信号量的线程的个数。

sem_init 函数

初始化一个信号量

int sem_init(sem_t *sem, int pshared, unsigned int value);

参 1：sem 信号量

参 2：pshared 取 0 用于线程间；取非 0（一般为 1）用于进程间

参 3：value 指定信号量初值

sem_destroy 函数

销毁一个信号量

int sem_destroy(sem_t *sem);

sem_wait 函数

给信号量加锁 --

int sem_wait(sem_t *sem);

sem_post 函数

给信号量解锁 ++

int sem_post(sem_t *sem);

sem_trywait 函数

尝试对信号量加锁 -- (与 sem_wait 的区别类比 lock 和 trylock)

int sem_trywait(sem_t *sem);

sem_timedwait 函数

限时尝试对信号量加锁 --

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

参 2：abs_timeout 采用的是绝对时间。

定时 1 秒：

```
time_t cur = time(NULL); 获取当前时间。
struct timespec t;  定义 timespec 结构体变量 t
t.tv_sec = cur+1; 定时 1 秒
t.tv_nsec = t.tv_sec +100;
sem_timedwait(&sem, &t); 传参
```

生产者消费者信号量模型

【练习】：使用信号量完成线程间同步，模拟生产者，消费者问题。

【sem_product_consumer.c】

分析：

规定： 如果□中有数据，生产者不能生产，只能阻塞。

如果□中没有数据，消费者不能消费，只能等待数据。

定义两个信号量：S 满 = 0， S 空 = 1 （S 满代表满格的信号量，S 空表示空格的信号量，程序起始，格子一定为空）

所以有：

```
T 生产者主函数 {
    sem_wait(S 空);
    生产....
    sem_post(S 满);
}
```

```
T 消费者主函数 {
    sem_wait(S 满);
    消费....
    sem_post(S 空);
}
```

假设： 线程到达的顺序是:T 生、T 生、T 消。

那么： T 生 1 到达，将 S 空-1，生产，将 S 满+1
T 生 2 到达，S 空已经为 0， 阻塞
T 消 到达，将 S 满-1，消费，将 S 空+1

三个线程到达的顺序是：T 生 1、T 生 2、T 消。而执行的顺序是 T 生 1、T 消、T 生 2

这里，S 空 表示空格子的总数，代表可占用信号量的线程总数-->1。其实这样的话，信号量就等同于互斥锁。

但，如果 S 空=2、3、4……就不一样了，该信号量同时可以由多个线程占用，不再是互斥的形式。因此我们说信号量是互斥锁的加强版。

【推演练习】： 理解上述模型，推演，如果是两个消费者，一个生产者，是怎么样的情况。

【作业】：结合生产者消费者信号量模型，揣摩 sem_timedwait 函数作用。编程实现，一个线程读用户输入， 另一个线程打印“hello world”。如果用户无输入，则每隔 5 秒向屏幕打印一个“hello world”；如果用户有输入，立刻打印“hello world”到屏幕。