第七天多任务互斥与同步

一、回顾知识点

1.1 线程的概念

为了减少(多进程)的系统开销，从进程中演化出了线程。

线程存在于进程中，共享进程的资源。

线程是进程中的独立控制流，由环境（包括寄存器组和程序计数器）和一系列的执行指令组成。

每个进程有一个地址空间和一个控制线程(主线程)

线程是轻量级的进程（LWP：light weight process），在Linux环境下线程的本质仍是进程。查看指定进程的 `LWP` 号：`ps -Lf pid`。

线程和进程的比较

```
1) 调度：
        线程是CPU调度和分派的基本单位
        进程是系统中程序执行和资源分配的基本单位
2) 拥有资源：
      线程自己一般不拥有资源（除了必不可少的程序计数器，一组寄存器和栈），但
它可以去访问其所属进程的资源，如进程代码段，数据段以及系统资源（已打开的文
件，I/O 设备等）

3) 系统开销：
      同一个进程中的多个线程可共享同一地址空间，因此它们之间的同步和通信的实
现也变得比较容易。
      在进程切换时候，涉及到整个当前进程 CPU 环境的保存以及新被调度运行的进程
的 CPU 环境的设置；而线程切换只需要保存和设置少量寄存器的内容，并不涉及存储
器管理方面的操作，从而能更有效地使用系统资源和提高系统的吞吐量。

4) 并发性：
```

不仅进程间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行。

1.2 多线程优势

线程共享的资源

```
1) 文件描述符表
2) 每种信号的处理方式
3) 当前工作目录
4) 用户 ID 和组 ID
```

线程非共享的

```
1) 线程 id
2) 处理器现场和栈指针(内核栈)
3) 独立的栈空间(用户空间栈)
4) errno 变量
5) 信号屏蔽字
6) 调度优先级
```

线程的优缺点

优点:

```
1. 提高程序并发性
2. 开销小
3. 数据通信、共享数据方便
```

缺点:

```
1. 库函数，不稳定
2. 调试、编写困难、gdb 不支持
3. 对信号支持不好 优点相对突出，缺点均不是硬伤。Linux 下由于实现方法导致进程、线程差别不是很大。
```

使用多线程的目的:

```
1) 多任务程序的设计
2) 并发程序设计
3) 网络程序设计
4) 数据共享
```

在多 CPU 系统中，实现真正的并行

1.3 线程的创建

> 程序中一旦使用线程函数 必须加上线程库 `-lpthread`

获取线程号

```
#include <pthread.h>
pthread_t pthread_self(void);
```

创建线程函数

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg );
```

参数:

```
thread：线程标识符地址。
attr：线程属性结构体地址，通常设置为 NULL
start_routine：线程函数的入口地址。
arg：传给线程函数的参数。
```

返回值：成功，0；失败，非 0

与 fork 不同的是 pthread_create 创建的线程不与父线程在同一点开始运行，而是从指定的函数开始运行，该函数运行完后，该线程也就退出了。
线程依赖进程存在的，如果创建线程的进程结束了，线程也就结束了。

【小结】子线程的执行由CPU调度，不确定顺序， 即使设置线程优先级也是如何。

1.4 线程等待

> 等待子线程结束，并回收子线程资源
>
> 如果等待的线程不结束，那么`pthread_join`阻塞。

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

功能: 等待线程结束（此函数会阻塞），并回收线程资源，类似进程的 wait() 函数。如果线程已经结束，那么该函数会立即返回。

参数:

```
thread：被等待的线程号。
retval：用来存储线程退出状态的指针的地址
```

返回值: 成功 0, 失败 (非 0)

1.5 线程分离

一般情况下，线程终止后，其终止状态一直保留到其它线程调用 `pthread_join` 获取它的状态为止。但是线程也可以被置为 detach 状态，这样的线程一旦终止就立刻 回收它占用的所有资源，而不保留终止状态。

不能对一个已经处于 detach 状态的 线程调用 `pthread_join`，这样的调用将返回 `EINVAL` 错误。也就是说，如果已经对 一个线程调用了`pthread_detach` 就不能再调用 `pthread_join`。

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

功能：

使调用线程与当前进程分离，分离后不代表此线程不依赖于当前进程，线程分离的目的是将线程资源的回收工作交由系统自动来完成，也就是说当被分离的线程结束之后，系统会自动回收它的资源。所以，此函数不会阻塞。

返回值： 成功：0; 失败，非 0

1.6 线程的退出

在进程中我们可以调用 `exit` 函数或 `_exit` 函数来结束进程，在一个线程中我们可以通过以下三种在不终止整个进程的情况下停止它的控制流(子线程)。

```
1、线程从执行函数中返回。
2、线程调用 pthread_exit 退出线程。
3、线程可以被同一进程中的其它线程取消
```

线程退出函数

```
#include <pthread.h>
void pthread_exit(void *retval);
```

`retval`：存储线程退出状态的指针

一个进程中的多个线程是共享该进程的数据段，因此，通常线程退出后所占用的资源并不会释放。

线程的取消

> 取消线程是指取消一个正在执行线程的操作.

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

成功返回 0，失败返回出错编号。

`pthread_cancel` 函数的实质是发信号给目标线程 thread，使目标线程退出。

线程退出清理

> 和进程的退出清理一样，线程也可以注册它退出时要调用的函数，这样的函数称为线程清理处理程序(thread cleanup handler)。

【注意】线程可以建立多个清理处理程序。处理程序在栈中，故它们的执行顺序与它们注册时的顺序相反。

注册清理函数：

> 将清除函数压栈， 即注册清理函数。

```
#include <pthread.h>
void pthread_cleanup_push(void (* routine)(void *), void *arg);
```

参数:

```
routine：线程清理函数的指针
arg：传给线程清理函数的参数
```

弹出清理函数

> 将清除函数弹栈，即删除清理函数。

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

参数：

```
execute:线程清理函数执行标志位。
        非 0，弹出清理函数，执行清理函数。
        0，弹出清理函数，不执行清理函数
```

当线程执行以下动作时会调用清理函数：

```
1、调用 pthread_exit 退出线程。
2、响应其它线程的取消请求。
3、用非零 execute 调用 pthread_cleanup_pop。
```

无论哪种情况 `pthread_cleanup_pop` 都将删除上一次 `pthread_cleanup_push` 调用注册的清理处理函数。

1.7 线程的属性【扩展】

> Linux 下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。

线程属性主要包括如下属性：

```
作用域（scope）
栈尺寸（stack size）
栈地址（stack address）
优先级（priority）
分离的状态（detached state）
调度策略和参数（scheduling policy and parameters）。
```

默认的属性为非绑定、非分离、缺省的堆栈、与父进程同样级别的优先级.

线程属性初始化及销毁函数

```
//线程属性初始化
int pthread_attr_init(pthread_attr_t *attr);
函数返回值：成功：0；失败：错误号

//线程属性资源销毁
int pthread_attr_destroy(pthread_attr_t *attr);
函数返回值：成功：0；失败：错误号
```

通过属性进行线程分离

> 先设置线程的分离属性、再去创建线程。

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

获取程属性，分离or 非分离
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstat);

参数：
    attr：已初始化的线程属性
    detachstate：
        分离状态PTHREAD_CREATE_DETACHED（分离线程）
        PTHREAD_CREATE_JOINABLE（非分离线程）
```

设置线程的栈空间

```
//设置栈的地址
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t
stacksize);
成功：0；失败：错误号

//得到栈的地址
int pthread_attr_getstack(pthread_attr_t *attr, void **stackaddr, size_t
*stacksize);
成功：0；失败：错误号
参数：
    attr：指向一个线程属性的指针
    stackaddr：返回获取的栈地址
    stacksize：返回获取的栈大小

//设置线程所使用的栈空间大小
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
成功：0；失败：错误号
//得到线程所使用的栈空间大小
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
成功：0；失败：错误号

参数：
    attr：指向一个线程属性的指针
    stacksize：返回线程的堆栈大小
```

设置线程的优化级

> 通过`struct sched_param`结构体设置。

```
struct sched_param {
    int __sched_priority;     // 优先级
};
```

它的取值范围取决于操作系统和调度策略。

在Linux系统中，常用的调度策略是基于优先级的实时调度策略（`SCHED_FIFO`和`SCHED_RR(Round-Robin Scheduling， 轮询调度）`）。在这些调度策略下，`sched_priority`的取值范围通常是1到99，其中1表示最低优先级，99表示最高优先级。

核心函数：

```
// 设置线程属性的调度策略
pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
    policy：SCHED_RR 轮论调度、SCHED_FIFO 先进先出队列调度、SCHED_OTHER 其它

// 获取或设置线程属性的调度参数
pthread_attr_getschedparam(pthread_attr_t *attr, sched_param *param);
pthread_attr_setschedparam(pthread_attr_t *attr, sched_param *param);

// 设置某一个线程的调度策略与参数
int pthread_setschedparam (pthread_t tid, int policy, const struct sched_param
*param)

// 获取某一个线程的调度策略与参数
int pthread_getschedparam (pthread_t __target_thread,
                  int *__restrict __policy,
                  struct sched_param *__restrict __param)
```

二、多任务互斥和同步

2.1 互斥和同步概述

同步和互斥是用于解决如下两个问题：

1）在多任务操作系统中，同时运行的多个任务可能都需要访问/使用同一种资源。

2）多个任务之间有依赖关系，某个任务的运行依赖于另一个任务。

互斥:

一个公共资源同一时刻只能被一个进程或线程使用，多个进程或线程不能同时使用公共资源。

POSIX 标准中进程和线程同步和互斥的方法，主要有信号量和互斥锁两种方式。

同步:

两个或两个以上的进程或线程在运行过程中协同步调，按预定的先后次序运行。

【注意】同步是特殊的互斥。

2.2 互斥锁

> 安装帮助文档：`sudo apt-get install manpages-posix-dev -y`

2.2.1 互斥锁的概念

用于线程的互斥。

mutex 互斥锁是一种简单的加锁的方法来控制对共享资源的访问。

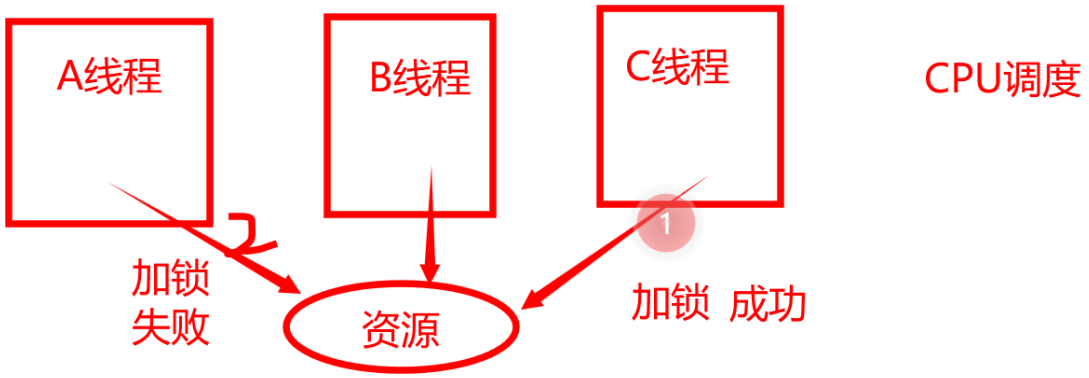
互斥锁只有两种状态,即加锁(lock)和解锁（unlock）。

互斥锁的操作流程如下：

1）在访问共享资源前，对互斥锁进行加锁；

2）在访问完成后释放互斥锁；

3）对互斥锁进行加锁后，任何其他试图再次对互斥锁加锁的线程将会被阻塞，直到锁被释放。



2.2.2 初始化互斥锁

> 不管多少个任务 如果是完成互斥 只需要一把锁。

`mutex` 用 `pthread_mutex_t` 数据类型表示，在使用互斥锁前,必须先对它进行初始化。

2.2.2.1 静态初始化

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```

2.2.2.2 动态初始化

```
#include <pthread.h>

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

参数:

- `mutex`：互斥锁地址。类型是 `pthread_mutex_t`
- `attr`：设置互斥量的属性，通常可采用默认属性，即可将 `attr` 设为 NULL。

返回值: 成功返回 0，失败返回非 0

2.2.3 互斥锁上锁

> 对互斥锁上锁，若已经上锁，则调用者一直阻塞到互斥锁解锁。 【阻塞的】

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

返回值：成功返回 0，失败返回非 0

2.2.4 互斥锁上锁2

> 对互斥锁上锁，若已经上锁，则上锁失败，函数立即返回。 【非阻塞的】

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

返回值：成功返回 0，失败返回非 0

2.2.5 互斥锁解锁

> 对指定的互斥锁解锁。

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

返回值：成功返回 0，失败返回非 0

2.2.6 销毁互斥锁

> 销毁指定的一个互斥锁

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

返回值：成功返回 0，失败返回非 0

如1： 不使用互斥锁，两个线程任务交叉执行

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

void *printer(void *data)
{
    char *msg = (char *)data;
    while (*msg)
    {
        printf("%c", *msg++);
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}

int main(int argc, char const *argv[])
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, printer, "disen666");
    pthread_create(&tid2, NULL, printer, "lucy888");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("---over--\n");
    return 0;
}
```

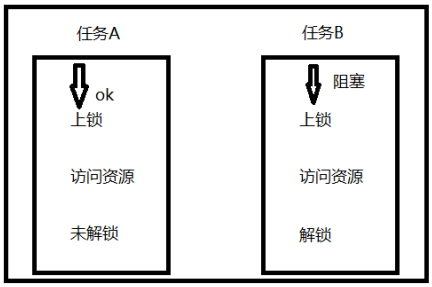
```
disen@qfxa:~/code3/day07$ ./a.out
lduicsye8n86866---over--
disen@qfxa:~/code3/day07$
```

如2： 使用互斥锁，依次执行线程任务

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

// 静态初始化
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *printer(void *data)
{
    char *msg = (char *)data;
    pthread_mutex_lock(&mutex); // 申请上锁， 可能阻塞
    while (*msg)
    {
        printf("%c", *msg++);
        fflush(stdout);
        sleep(1);
    }
    pthread_mutex_unlock(&mutex); // 解锁
    return NULL;
}

int main(int argc, char const *argv[])
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, printer, "disen666");
    pthread_create(&tid2, NULL, printer, "lucy888");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("---over--\n");

    // 销毁互斥锁
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

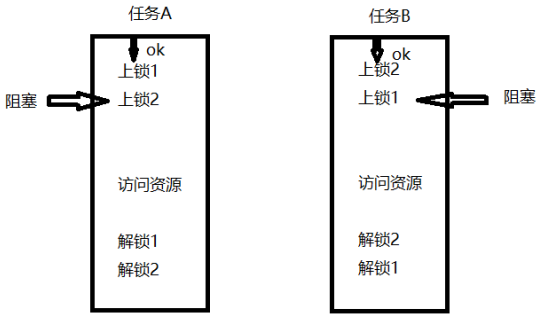
```
disen@qfxa:~/code3/day07$ gcc nomutex1.c -lpthread
disen@qfxa:~/code3/day07$ ./a.out
disen666lucy888---over--
```

2.2.7 死锁



出现死锁1：上完锁 未解锁

解决办法：上锁和解锁 必须一一对应使用



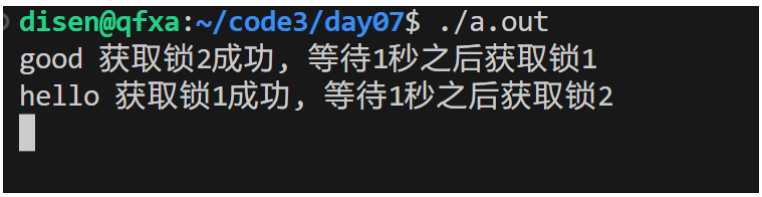
出现死锁2: 多把锁的上锁顺序问题 导致死锁

解决办法：规定好上锁顺序

如： 两个锁使用错误，导致死锁

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

pthread_mutex_t mutex1, mutex2;

void *task1(void *data)
{
    char *taskName = (char *)data;
    pthread_mutex_lock(&mutex1);
    printf("%s 获取锁1成功, 等待1秒之后获取锁2\n", taskName);
    sleep(1);
    pthread_mutex_lock(&mutex2);
    printf("%s 获取锁2成功\n", taskName);
    printf("%s\n", taskName);

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

void *task2(void *data)
{
    char *taskName = (char *)data;
    pthread_mutex_lock(&mutex2);
    printf("%s 获取锁2成功, 等待1秒之后获取锁1\n", taskName);
    sleep(1);
    pthread_mutex_lock(&mutex1);
    printf("%s 获取锁1成功\n", taskName);
    printf("%s\n", taskName);

    pthread_mutex_unlock(&mutex1);
    pthread_mutex_unlock(&mutex2);
    return NULL;
}

int main(int argc, char const *argv[])
{
    // 动态初始化互斥锁
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);
```

```
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, task1, "hello");
    pthread_create(&tid2, NULL, task2, "good");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // 销毁锁
    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);
    return 0;
}
```



如2： 解决死锁问题, 一定要按顺序加锁与解锁

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

pthread_mutex_t mutex1, mutex2;

void *task1(void *data)
{
    char *taskName = (char *)data;
    pthread_mutex_lock(&mutex1);
    printf("%s 获取锁1成功, 等待1秒之后获取锁2\n", taskName);
    sleep(1);
    pthread_mutex_lock(&mutex2);
    printf("%s 获取锁2成功\n", taskName);
    printf("%s\n", taskName);

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}

void *task2(void *data)
{
    char *taskName = (char *)data;
    pthread_mutex_lock(&mutex1);
    printf("%s 获取锁1成功, 等待1秒之后获取锁2\n", taskName);
    sleep(1);
    pthread_mutex_lock(&mutex2);
    printf("%s 获取锁2成功\n", taskName);
    printf("%s\n", taskName);

    pthread_mutex_unlock(&mutex2);
    pthread_mutex_unlock(&mutex1);
    return NULL;
}
```

```c
int main(int argc, char const *argv[])
{
    // 动态初始化互斥锁
    pthread_mutex_init(&mutex1, NULL);
    pthread_mutex_init(&mutex2, NULL);

    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, task1, "hello");
    pthread_create(&tid2, NULL, task2, "good");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    // 销毁锁
    pthread_mutex_destroy(&mutex1);
    pthread_mutex_destroy(&mutex2);
    return 0;
}
```

```
disen@qfxa:~/code3/day07$ ./a.out
hello 获取锁1成功，等待1秒之后获取锁2
hello 获取锁2成功
hello
good 获取锁1成功，等待1秒之后获取锁2
good 获取锁2成功
good
```

2.3 读写锁

> 如果就两个任务，一个读、一个写，建议使用互斥锁。
>
> 如果3个或3个以上的任务，多个读或多个写，建议使用读写锁。

2.3.1 读写锁的概述

读写锁（Read-Write Lock）是一种并发控制机制，用于在多线程环境下对共享资源进行读写操作的同步和互斥。

读写锁允许多个线程同时对共享资源进行读取操作，但在有线程进行写入操作时，其他线程无法进行读取或写入操作，从而保证了对共享资源的安全访问。

读写锁通常有两种状态：读取状态和写入状态。当没有线程进行写入操作时，多个线程可以同时获取读取锁，从而并发地读取共享资源。这样可以提高并发性能，因为读取操作不会修改共享资源，不会产生数据竞争。

当有线程获取写入锁时，其他线程无法获取读取锁或写入锁，从而实现了独占式的写入操作。这是为了保证数据的一致性和完整性，因为写入操作可能会修改共享资源，需要排他地进行。

读写锁的基本特点如下：

- 多个线程可以同时获取读取锁，实现并发的读取操作。
- 写入锁是独占的，当有线程获取写入锁时，其他线程无法获取读取锁或写入锁。
- 当有线程持有读取锁时，其他线程可以继续获取读取锁，但不能获取写入锁。
- 读取锁和写入锁之间的优先级关系可以根据具体的实现策略而定，如读优先或写优先。

读写锁的使用场景通常是在读操作远远多于写操作的情况下，通过允许多个线程同时读取来提高并发性能。然而，需要注意的是，如果读操作的频率非常高，而写操作的频率很低，那么读写锁可能会导致写入操作的饥饿，即写入操作一直得不到执行。

2.3.2 读写锁的API

> 读写锁的类型： `pthread_rwlock_t`

2.3.2.1 动态初始化

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict_rwlock,
                        const pthread_rwlockattr_t *restrict_attr);
```

参数：

- `rwlock`：指向要初始化的读写锁指针。
- `attr`：读写锁的属性指针。如果 attr 为 NULL 则会使用默认的属性初始化读写锁，否则使用指定的 attr 初始化读写锁。

2.3.2.2 静态初始化

```
pthread_rwlock_t my_rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

这种方法等价于使用 NULL 指定的 attr 参数调用 `pthread_rwlock_init()` 来完成动态初始化，不同之处在于`PTHREAD_RWLOCK_INITIALIZER` 宏不进行错误检查。

2.3.2.3 销毁读写锁

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

2.3.2.4 申请读锁

阻塞方式申请：

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

以阻塞方式在读写锁上获取读锁（读锁定）。

如果没有写者持有该锁，并且没有写者阻塞在该锁上，则调用线程会获取读锁。

如果调用线程未获取读锁，则它将阻塞直到它获取了该锁。一个线程可以在一个读写锁上多次执行读锁定。

线程可以成功调用 `pthread_rwlock_rdlock()`函数 n 次，但是之后该线程必须调用 `pthread_rwlock_unlock()` 函数 n 次才能解除锁定。

非阻塞方式申请：

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

用于尝试以非阻塞的方式来在读写锁上获取读锁。

如果有任何的写者持有该锁或有写者阻塞在该读写锁上，则立即失败返回。

2.3.2.5 申请写锁

阻塞方式申请：

```
#include <pthread.h>
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

在读写锁上获取写锁（写锁定）。

如果没有写者持有该锁，并且没有写者读者持有该锁，则调用线程会获取写锁。

如果调用线程未获取写锁，则它将阻塞直到它获取了该锁。

非阻塞方式申请：

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

用于尝试以非阻塞的方式来在读写锁上获取写锁。

如果有任何的读者或写者持有该锁，则立即失败返回。

2.3.2.6 释放读写锁

```
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

无论是读锁或写锁，都可以通过此函数解锁。

如：

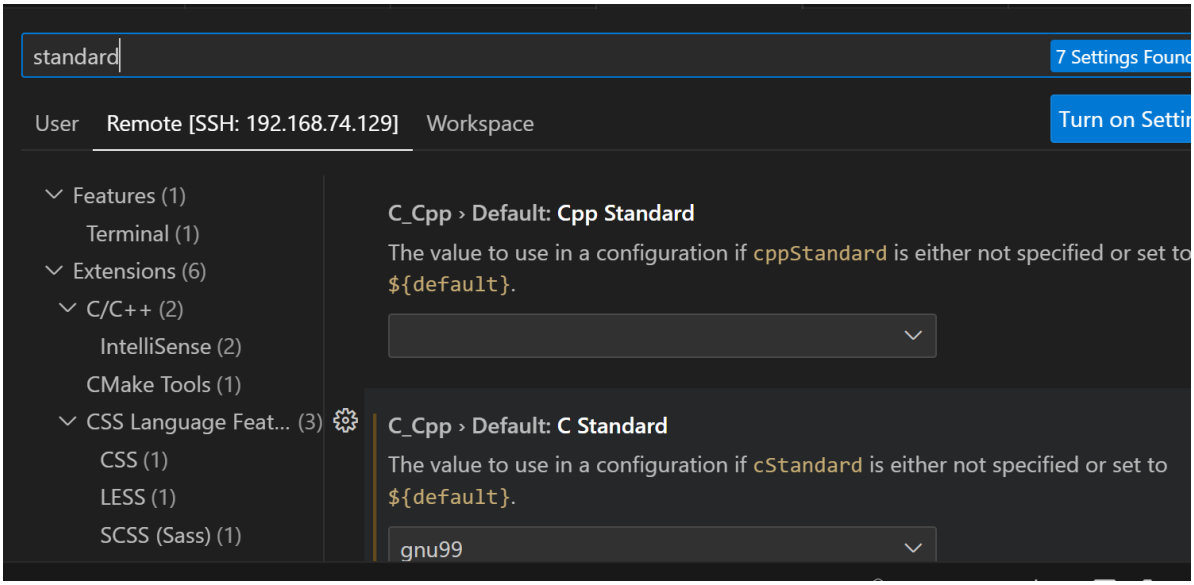
```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

// 定义读写锁
pthread_rwlock_t rwlock;

void *read_task(void *data)
{
    int *n = (int *)data;

    while (1)
    {
        pthread_rwlock_rdlock(&rwlock); // 申请读锁

        printf("(%ld)线程读取num: %d\n", pthread_self(), *n);
        pthread_rwlock_unlock(&rwlock); // 解锁
        usleep(500 * 1000);
    }

    return NULL;
}
void *write_task(void *data)
{
```

```c
    int *n = (int *)data;
    while (1)
    {
        pthread_rwlock_wrlock(&rwlock);
        (*n)++;
        printf("(%ld)线程写num: %d\n", pthread_self(), *n);
        pthread_rwlock_unlock(&rwlock);

        sleep(1);
    }
    return NULL;
}

int main(int argc, char const *argv[])
{
    // 初始化读写锁
    pthread_rwlock_init(&rwlock, NULL);

    int num; // 公共资源
    pthread_t tid1, tid2, tid3;
    pthread_create(&tid1, NULL, read_task, &num);
    pthread_create(&tid2, NULL, read_task, &num);
    pthread_create(&tid3, NULL, write_task, &num);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    pthread_rwlock_destroy(&rwlock);
    return 0;
}
```

```
disen@qfxa:~/code3/day07$ gcc rwlock1.c -lpthread
disen@qfxa:~/code3/day07$ ./a.out
(140695940011776)线程读取num: 0
(140695931619072)线程读取num: 0
(140695923226368)线程写num: 1
(140695931619072)线程读取num: 1
(140695940011776)线程读取num: 1
(140695931619072)线程读取num: 1
(140695940011776)线程读取num: 1
(140695923226368)线程写num: 2
```

【扩展】vscode_ssh环境下支持GNU99

在vscode-ssh编辑器，按`ctrl+,`



在打开设置界面，选择"Remote[SSH:xxx.xxx.xxx.xxx]", 并在输入框，输入`standard`，在`C Standared`位置上，选择`gnu99`。则在代码中定义读写锁时不会报`红波浪线`(未在头中找到定义)。

【说明】`gnuXX标准` = `cXX标准` + `GNU extension`

练习： 银行存取款

> 设计两个线程任务函数，一个完成取款，一个完成存款。

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

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *task1(void *data)
{
    int *money = (int *)data;
    // 存款任务
    pthread_rwlock_wrlock(&rwlock);
    // pthread_rwlock_rdlock(&rwlock);
    //  查询余额
    printf("存款任务-余额: %d\n", *money);
    // pthread_rwlock_unlock(&rwlock);
    sleep(1);

    // 从键盘读取存入的金额
    printf("请输入存款金额: ");
    fflush(stdout);
    int m;
    scanf("%d", &m);
    *money += m;
    // 修改余额并打印结果
    printf("存款成功, 余额为: %d\n", *money);
    pthread_rwlock_unlock(&rwlock);
    pthread_exit(NULL);
}

void *task2(void *data)
```

```c
{
    int *money = (int *)data;
    // 取款任务
    pthread_rwlock_wrlock(&rwlock);
    // pthread_rwlock_rdlock(&rwlock);
    //  查询余额
    printf("取款任务-余额: %d\n", *money);
    // pthread_rwlock_unlock(&rwlock);
    sleep(1);

    // 从键盘读取取出的金额
    printf("请输入取款金额: ");
    fflush(stdout);
    int m;
    scanf("%d", &m);
    if (*money >= m)
    {
        *money -= m;
        // 修改余额并打印结果
        printf("取款成功, 余额为: %d\n", *money);
    }
    else
    {
        printf("取款失败，余额不足\n");
    }

    pthread_rwlock_unlock(&rwlock);
    pthread_exit(NULL);
}

int main(int argc, char const *argv[])
{
    int money = 1000; // 存款

    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, task1, &money);
    pthread_create(&tid2, NULL, task2, &money);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_rwlock_destroy(&rwlock);
    return 0;
}
```

```
disen@qfxa:~/code3/day07$ ./a.out
存款任务-余额: 1000
请输入存款金额: 200
存款成功，余额为: 1200
取款任务-余额: 1200
请输入取款金额: 1500
取款失败，余额不足
```

2.4 条件变量

> 与互斥锁不同，条件变量是用来等待而不是用来上锁的，条件变量本身不是锁。
>
> 条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥 锁同时使用。
>
> 条件变量的两个动作： 条件不满足, 阻塞线程; 当条件满足, 通知阻塞的线程开始工作。
>
> 条件变量的类型: `pthread_cond_t`。

2.4.1 条件变量初始化

动态条件变量初始化

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

`attr`：条件变量属性，通常为默认值，传 NULL 即可。

静态初始化条件变量

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

2.4.2 销毁条件变量

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

2.4.3 等待条件满足

2.4.3.1 阻塞等待一个条件变量

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict_cond,
                      pthread_mutex_t *restrict_mutex);
```

功能:

- a) 阻塞等待条件变量 cond（参1）满足
- b) 释放已掌握的互斥锁（解锁互斥量）相当于 pthread_mutex_unlock(&mutex);
    - a) b) 两步为一个原子操作。
- c) 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)`;

参数:

- `cond`：指向要初始化的条件变量指针
- `mutex`：互斥锁

2.4.3.2 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);
```

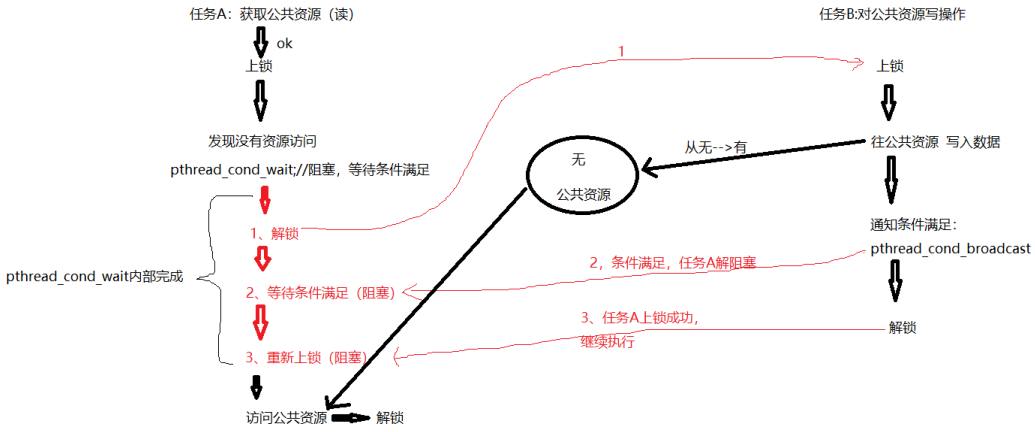
参数:

- `cond`：指向要初始化的条件变量指针
- `mutex`：互斥锁
- `abstime`：绝对时间

2.4.4 唤醒等待的线程

```
#include <pthread.h>
// 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);

// 唤醒全部阻塞在条件变量上的线程
int pthread_cond_broadcast(pthread_cond_t *cond);
```

2.4.5 条件变量的工作原理



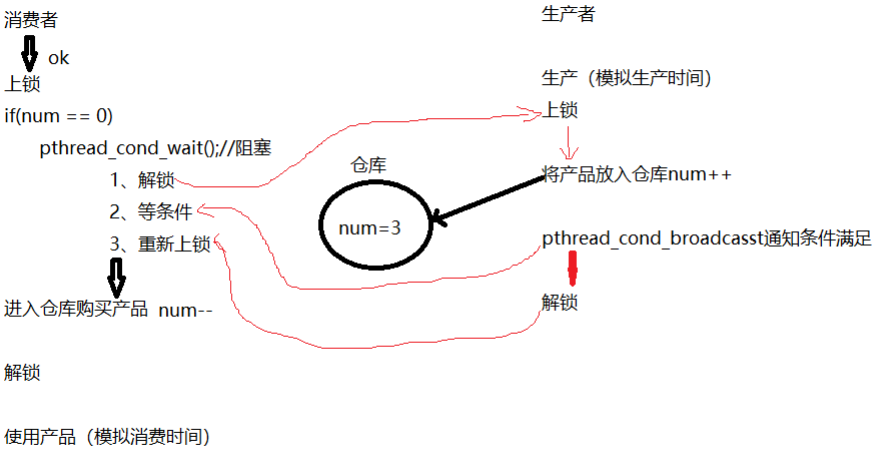
2.4.6 案例：生产者与消费者

一个仓库刚开始有3个产品。

如果生产者生产后进入仓库存放产品，消费者就不能进入仓库购买。

如果消费者进入仓库购买产品，生产者就不能进入仓库存放产品。

如果仓库的商品为0，消费者不能进入仓库购买。



如：

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

// 互斥锁和条件变量的初始化
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *producer_task(void *data)
{
    int *n = (int *)data;
    while (1)
    {

        pthread_mutex_lock(&mutex);
        (*n)++;
        printf("生产线程(%ld)生产了 %d 产品\n", pthread_self(), *n);

        // 发出通知，让等待消费的线程恢复（条件满足）
        pthread_cond_broadcast(&cond);

        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    pthread_exit(NULL);
}
void *consumer_task(void *data)
{
    int *n = (int *)data;
    while (1)
    {
        pthread_mutex_lock(&mutex);
        while (*n == 0)
        {
            pthread_cond_wait(&cond, &mutex);
        }
        printf("消费者(%ld) 消费了 %d 产品\n", pthread_self(), *n);
        (*n)--;
        pthread_mutex_unlock(&mutex);
        sleep(1);
```

```
    }
    pthread_exit(NULL);
}

int main(int argc, char const *argv[])
{
    int num = 3; // 3个产品
    pthread_t threads[5];
    // 创建2个生产线程
    for (int i = 0; i < 2; i++)
    {
        pthread_create(&threads[i], NULL, producer_task, &num);
    }

    // 创建3个消费者线程
    for (int i = 2; i < 5; i++)
    {
        pthread_create(&threads[i], NULL, consumer_task, &num);
    }

    for (int i = 0; i < 5; i++)
    {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    return 0;
}
```

```
disen@qfxa:~/code3/day07$ ./a.out
生产线程(139962739033856)生产了 4 产品
生产线程(139962730641152)生产了 5 产品
消费者(139962722248448) 消费了 5 产品
消费者(139962713855744) 消费了 4 产品
消费者(139962705463040) 消费了 3 产品
消费者(139962705463040) 消费了 2 产品
消费者(139962713855744) 消费了 1 产品
生产线程(139962730641152)生产了 1 产品
消费者(139962722248448) 消费了 1 产品
生产线程(139962739033856)生产了 1 产品
消费者(139962713855744) 消费了 1 产品
生产线程(139962730641152)生产了 1 产品
消费者(139962722248448) 消费了 1 产品
生产线程(139962739033856)生产了 1 产品
消费者(139962705463040) 消费了 1 产品
```

2.5 信号量

2.5.1 信号量的概念

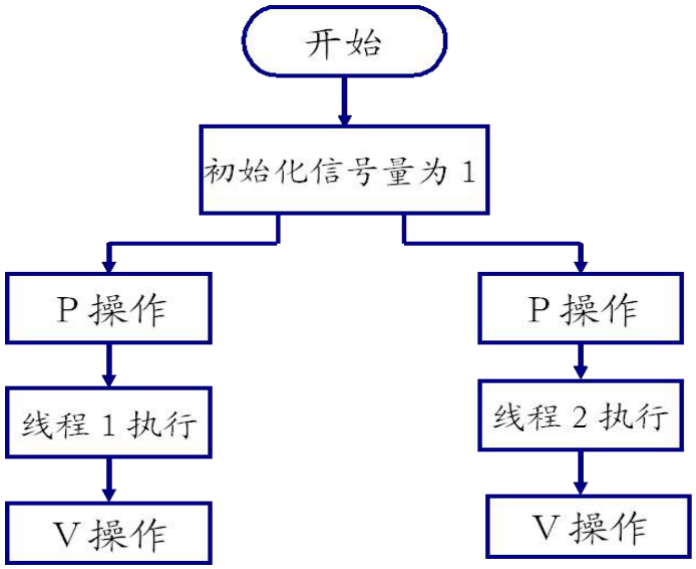
信号量广泛用于进程或线程间的同步和互斥，信号量本质上是一个非 负的整数计数器，它被用来控制对公共资源的访问。

当信号量值大于 0 时，则可以访问，否则将阻塞。PV 原语是对信号量的操作，一次 P 操作使信号量减１，一次 V 操作使信号量加１。

信号量数据类型为：`sem_t`

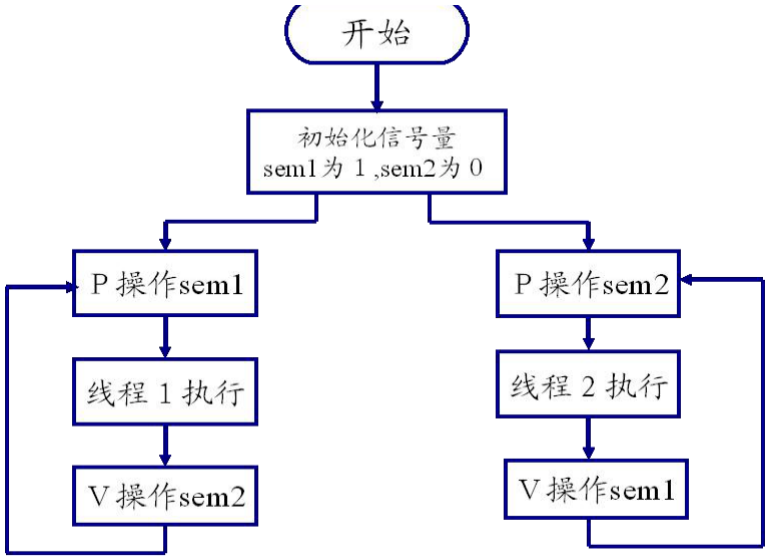
信号量完成互斥

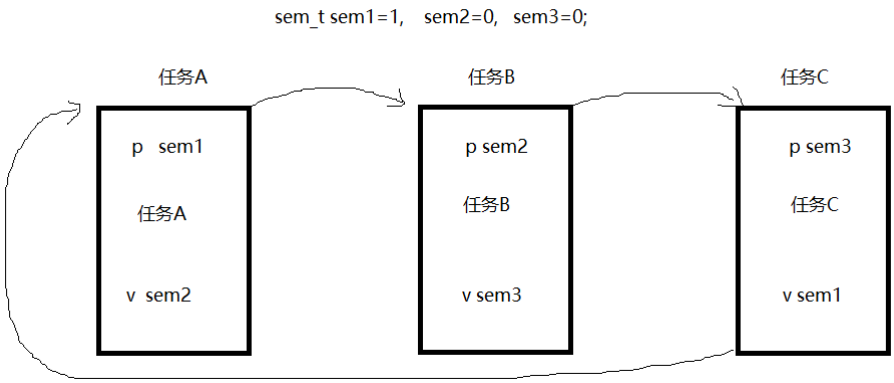
> 不管有多少个任务，只要是互斥，只要一个信号量，并且初始化1.



信号量用于同步

> 有几个任务就需要有几个信号量，先执行为任务的信号初始化为1，其他信号量初始化为0，所有任务P 自己的信号，V 下一个任务的信号量。





2.5.2 信号量的API

2.5.2.1 信号量的初始化

> 创建一个信号量并初始化它的值。一个无名信号量在被使用前必须先初始化。

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

参数：

- `sem`：信号量的地址
- `pshared`：等于 0，信号量在线程间共享（常用）；不等于 0，信号量在进程间共享。
- value：信号量的初始值

返回值： 成功0，失败 - 1

2.5.2.2 P操作

功能：将信号量减1,如果信号量的值为 0 则阻塞,大于 0 可以减1

```
int sem_wait(sem_t *sem);
```

功能： 尝试将信号量减一,如果信号量的值为 0 不阻塞,立即返回 ,大于 0 可以减1

```
int sem_trywait(sem_t *sem);
```

返回值： 成功返回 0 失败返回-1

2.5.2.3 V操作

> 将信号量的值加 1 并发出信号唤醒等待线程。

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

返回值:成功返回 0 失败返回-1

2.5.2.4 销毁信号量

```
int sem_destroy(sem_t *sem);
```

返回值： 成功返回 0 失败返回-1

2.5.2.5 获取信号量的计数值

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

2.6 有名信号量

2.6.1 有名信号量概念

其实 POSIX 的信号量有两种：

- 1、无名信号量
- 2、有名信号量

前面我们介绍的就是无名信号量，无名信号量一般用于线程间同步或互斥。
而有名信号量一般用于进程间同步或互斥。

2.6.2 创建有名信号量

> 创建一个信号量

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
//信号量存在
sem_t *sem_open(const char *name, int oflag);

 //信号量不存在
sem_t *sem_open(const char *name, int oflag,
                       mode_t mode, unsigned int value);
```

参数：

- name：信号量文件名。
- `flags`：`sem_open` 函数的行为标志,同open()的flag。
- mode：文件权限(可读、可写、可执行)的设置。
- value ：信号量初始值。

返回值： 成功返回信号量的地址，失败返回 `SEM_FAILED`

2.6.3 信号量的关闭

```
#include <semaphore.h>
int sem_close(sem_t *sem);
```

返回值： 成功返回 0，失败返回-1。

2.6.4 信号量的删除

> 删除信号量的文件

```
#include <semaphore.h>
int sem_unlink(const char *name);
```

返回值： 成功返回 0，失败返回-1