**Linux C 多线程编程:基于条件变量和信号量的生产者-消费者问题**

[](https://www.zhihu.com/people/linuxsao-di-seng)

[**Linux百里**](https://www.zhihu.com/people/linuxsao-di-seng)

​关注他

多线程编程可以说每个程序员的基本功，同时也是开发中的难点之一，本文以Linux C为例，讲述了线程的创建及常用的几种线程同步的方式，最后对多线程编程进行了总结与思考并给出代码示例。

**一、创建线程**

多线程编程的第一步，创建线程。创建线程其实是增加了一个控制流程，使得同一进程中存在多个控制流程并发或者并行执行。

线程创建函数，其他函数这里不再列出，可以参考pthread.h。

**#include**<pthread.h>

**int** **pthread\_create**(

pthread\_t **\*restrict** **thread**, */\*线程id\*/*

**const** pthread\_attr\_t **\*restrict** attr, */\*线程属性，默认可置为NULL，表示线程属性取缺省值\*/*

**void** **\***(**\***start\_routine)(**void\***), */\*线程入口函数\*/*

**void** **\*restrict** arg */\*线程入口函数的参数\*/*

);

代码示例：

**#include**<stdio.h>

**#include**<string.h>

**#include**<stdlib.h>

**#include**<unistd.h>

**#include**<pthread.h>

**char\*** **thread\_func1**(**void\*** arg) {

pid\_t pid **=** getpid();

pthread\_t tid **=** pthread\_self();

printf("%s pid: %u, tid: %u (0x%x)\n", (**char\***)arg, (**unsigned** **int**)pid, (**unsigned** **int**)tid, (**unsigned** **int**)tid);

**char\*** msg **=** "thread\_func1";

**return** msg;

}

**void\*** **thread\_func2**(**void\*** arg) {

pid\_t pid **=** getpid();

pthread\_t tid **=** pthread\_self();

printf("%s pid: %u, tid: %u (0x%x)\n", (**char\***)arg, (**unsigned** **int**)pid, (**unsigned** **int**)tid, (**unsigned** **int**)tid);

**char\*** msg **=** "thread\_func2 ";

**while**(1) {

printf("%s running\n", msg);

sleep(1);

}

**return** NULL;

}

**int** **main**() {

pthread\_t tid1, tid2;

**if** (pthread\_create(**&**tid1, NULL, (**void\***)thread\_func1, "new thread:") **!=** 0) {

printf("pthread\_create error.");

exit(EXIT\_FAILURE);

}

**if** (pthread\_create(**&**tid2, NULL, (**void\***)thread\_func2, "new thread:") **!=** 0) {

printf("pthread\_create error.");

exit(EXIT\_FAILURE);

}

pthread\_detach(tid2);

**char\*** rev **=** NULL;

pthread\_join(tid1, (**void** **\***)**&**rev);

printf("%s return.\n", rev);

pthread\_cancel(tid2);

printf("main thread end.\n");

**return** 0;

}

**二、线程同步**

有时候我们需要多个线程相互协作来执行，这时需要线程间同步。线程间同步的常用方法有：

* 互斥
* 信号量
* 条件变量

我们先看一个未进行线程同步的示例：

**#include**<stdio.h>

**#include**<string.h>

**#include**<stdlib.h>

**#include**<unistd.h>

**#include**<pthread.h>

**#define LEN 100000**

**int** num **=** 0;

**void\*** **thread\_func**(**void\*** arg) {

**for** (**int** i **=** 0; i**<** LEN; **++**i) {

num **+=** 1;

}

**return** NULL;

}

**int** **main**() {

pthread\_t tid1, tid2;

pthread\_create(**&**tid1, NULL, (**void\***)thread\_func, NULL);

pthread\_create(**&**tid2, NULL, (**void\***)thread\_func, NULL);

**char\*** rev **=** NULL;

pthread\_join(tid1, (**void** **\***)**&**rev);

pthread\_join(tid2, (**void** **\***)**&**rev);

printf("correct result=%d, wrong result=%d.\n", 2**\***LEN, num);

**return** 0;

}

运行结果：correct result=200000, wrong result=106860.。

**互斥**

这个是最容易理解的，在访问临界资源时，通过互斥，限制同一时刻最多只能有一个线程可以获取临界资源。

其实互斥的逻辑就是：如果访问临街资源发现没有其他线程上锁，就上锁，获取临界资源，期间如果其他线程执行到互斥锁发现已锁住，则线程挂起等待解锁，当前线程访问完临界资源后，解锁并唤醒其他被该互斥锁挂起的线程，等待再次被调度执行。

“挂起等待”和“唤醒等待线程”的操作如何实现？每个Mutex有一个等待队列，一个线程要在Mutex上挂起等待，首先在把自己加入等待队列中，然后置线程状态为睡眠，然后调用调度器函数切换到别的线程。一个线程要唤醒等待队列中的其它线程，只需从等待队列中取出一项，把它的状态从睡眠改为就绪，加入就绪队列，那么下次调度器函数执行时就有可能切换到被唤醒的线程。

主要函数如下：

**#include** <pthread.h>

**int** **pthread\_mutex\_init**(pthread\_mutex\_t **\*restrict** mutex,

**const** pthread\_mutexattr\_t **\*restrict** attr); */\*初始化互斥量\*/*

**int** **pthread\_mutex\_destroy**(pthread\_mutex\_t **\***mutex); */\*销毁互斥量\*/*

**int** **pthread\_mutex\_lock**(pthread\_mutex\_t **\***mutex);

**int** **pthread\_mutex\_trylock**(pthread\_mutex\_t **\***mutex);

**int** **pthread\_mutex\_unlock**(pthread\_mutex\_t **\***mutex);

用互斥解决上面计算结果错误的问题，示例如下：

**#include**<stdio.h>

**#include**<string.h>

**#include**<stdlib.h>

**#include**<unistd.h>

**#include**<pthread.h>

**#define LEN 100000**

**int** num **=** 0;

**void\*** **thread\_func**(**void\*** arg) {

pthread\_mutex\_t**\*** p\_mutex **=** (pthread\_mutex\_t**\***)arg;

**for** (**int** i **=** 0; i**<** LEN; **++**i) {

pthread\_mutex\_lock(p\_mutex);

num **+=** 1;

pthread\_mutex\_unlock(p\_mutex);

}

**return** NULL;

}

**int** **main**() {

pthread\_mutex\_t m\_mutex;

pthread\_mutex\_init(**&**m\_mutex, NULL);

pthread\_t tid1, tid2;

pthread\_create(**&**tid1, NULL, (**void\***)thread\_func, (**void\***)**&**m\_mutex);

pthread\_create(**&**tid2, NULL, (**void\***)thread\_func, (**void\***)**&**m\_mutex);

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

pthread\_mutex\_destroy(**&**m\_mutex);

printf("correct result=%d, result=%d.\n", 2**\***LEN, num);

**return** 0;

}

运行结果：correct result=200000, result=200000.

如果在互斥中还嵌套有其他互斥代码，需要注意死锁问题。

**产生死锁的两种情况：**

* 一种情况是：如果同一个线程先后两次调用lock，在第二次调用时，由于锁已经被占用，该线程会挂起等待别的线程释放锁，然而锁正是被自己占用着的，该线程又被挂起而没有机会释放锁，因此就永远处于挂起等待状态了，产生死锁。
* 另一种典型的死锁情形是：线程A获得了锁1，线程B获得了锁2，这时线程A调用lock试图获得锁2，结果是需要挂起等待线程B释放锁2，而这时线程B也调用lock试图获得锁1，结果是需要挂起等待线程A释放锁1，于是线程A和B都永远处于挂起状态了。

**如何避免死锁：**

1. 不用互斥锁（这个很多时候很难办到）
2. 写程序时应该尽量避免同时获得多个锁。
3. 如果一定有必要这么做，则有一个原则：如果所有线程在需要多个锁时都按相同的先后顺序（常见的是按Mutex变量的地址顺序）获得锁，则不会出现死锁。 （比如一个程序中用到锁1、锁2、锁3，它们所对应的Mutex变量的地址是锁1<锁2<锁3，那么所有线程在需要同时获得2个或3个锁时都应该按锁1、锁2、锁3的顺序获得。如果要为所有的锁确定一个先后顺序比较困难，则应该尽量使用pthread\_mutex\_trylock调用代替pthread\_mutex\_lock调用，以避免死锁。）

**条件变量**

条件变量概括起来就是：一个线程需要等某个条件成立（而这个条件是由其他线程决定的）才能继续往下执行，现在这个条件不成立，线程就阻塞等待，等到其他线程在执行过程中使这个条件成立了，就唤醒线程继续执行。

相关函数如下：

**#include** <pthread.h>

**int** **pthread\_cond\_destroy**(pthread\_cond\_t **\***cond);

**int** **pthread\_cond\_init**(pthread\_cond\_t **\*restrict** cond,

**const** pthread\_condattr\_t **\*restrict** attr);

**int** **pthread\_cond\_timedwait**(pthread\_cond\_t **\*restrict** cond,

pthread\_mutex\_t **\*restrict** mutex,

**const** **struct** timespec **\*restrict** abstime);

**int** **pthread\_cond\_wait**(pthread\_cond\_t **\*restrict** cond,

pthread\_mutex\_t **\*restrict** mutex);

**int** **pthread\_cond\_broadcast**(pthread\_cond\_t **\***cond);

**int** **pthread\_cond\_signal**(pthread\_cond\_t **\***cond);

举个最容易理解条件变量的例子，“生产者-消费者”模式中，生产者线程向队列中发送数据，消费者线程从队列中取数据，当消费者线程的处理速度大于生产者线程时，会产生队列中没有数据了，一种处理办法是等待一段时间再次“轮询”，但这种处理方式不太好，你不知道应该等多久，这时候条件变量可以很好的解决这个问题。

下面是代码：

**#include**<sys/types.h>

**#include**<unistd.h>

**#include**<stdlib.h>

**#include**<stdio.h>

**#include**<pthread.h>

**#include**<errno.h>

**#include**<string.h>

**#define LIMIT 1000**

**struct** data {

**int** n;

**struct** data**\*** next;

};

pthread\_cond\_t condv **=** PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t mlock **=** PTHREAD\_MUTEX\_INITIALIZER;

**struct** data**\*** phead **=** NULL;

**void** **producer**(**void\*** arg) {

printf("producer thread running.\n");

**int** count **=** 0;

**for** (;;) {

**int** n **=** rand() **%** 100;

**struct** data**\*** nd **=** (**struct** data**\***)malloc(**sizeof**(**struct** data));

nd**->**n **=** n;

pthread\_mutex\_lock(**&**mlock);

**struct** data**\*** tmp **=** phead;

phead **=** nd;

nd**->**next **=** tmp;

pthread\_mutex\_unlock(**&**mlock);

pthread\_cond\_signal(**&**condv);

count **+=** n;

**if**(count **>** LIMIT) {

**break**;

}

sleep(rand()**%**5);

}

printf("producer count=%d\n", count);

}

**void** **consumer**(**void\*** arg) {

printf("consumer thread running.\n");

**int** count **=** 0;

**for**(;;) {

pthread\_mutex\_lock(**&**mlock);

**if** (NULL **==** phead) {

pthread\_cond\_wait(**&**condv, **&**mlock);

} **else** {

**while**(phead **!=** NULL) {

count **+=** phead**->**n;

**struct** data**\*** tmp **=** phead;

phead **=** phead**->**next;

free(tmp);

}

}

pthread\_mutex\_unlock(**&**mlock);

**if** (count **>** LIMIT)

**break**;

}

printf("consumer count=%d\n", count);

}

**int** **main**() {

pthread\_t tid1, tid2;

pthread\_create(**&**tid1, NULL, (**void\***)producer, NULL);

pthread\_create(**&**tid2, NULL, (**void\***)consumer, NULL);

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

**return** 0;

}

条件变量中的执行逻辑：

关键是理解执行到int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex)

这里时发生了什么，其他的都比较容易理解。执行这条函数前需要先获取互斥锁，判断条件是否满足，如果满足执行条件，则继续向下执行后释放锁；如果判断不满足执行条件，则释放锁，线程阻塞在这里，一直等到其他线程通知执行条件满足，唤醒线程，再次加锁，向下执行后释放锁。（简而言之就是：释放锁-->阻塞等待-->唤醒后加锁返回）

实现细节可看源码[pthread\_cond\_wait.c](https://link.zhihu.com/?target=https%3A//code.woboq.org/userspace/glibc/nptl/pthread_cond_wait.c.html)和[pthread\_cond\_signal.c](https://link.zhihu.com/?target=https%3A//code.woboq.org/userspace/glibc/nptl/pthread_cond_signal.c.html" \t "_blank)

上面的例子可能有些繁琐，下面的这个代码示例则更为简洁：

**#include**<sys/types.h>

**#include**<unistd.h>

**#include**<stdlib.h>

**#include**<stdio.h>

**#include**<pthread.h>

**#include**<errno.h>

**#include**<string.h>

**#define NUM 3**

pthread\_cond\_t condv **=** PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t mlock **=** PTHREAD\_MUTEX\_INITIALIZER;

**void** **producer**(**void\*** arg) {

**int** n **=** NUM;

**while**(n**--**) {

sleep(1);

pthread\_cond\_signal(**&**condv);

printf("producer thread send notify signal. %d\t", NUM**-**n);

}

}

**void** **consumer**(**void\*** arg) {

**int** n **=** 0;

**while** (1) {

pthread\_cond\_wait(**&**condv, **&**mlock);

printf("recv producer thread notify signal. %d\n", **++**n);

**if** (NUM **==** n) {

**break**;

}

}

}

**int** **main**() {

pthread\_t tid1, tid2;

pthread\_create(**&**tid1, NULL, (**void\***)producer, NULL);

pthread\_create(**&**tid2, NULL, (**void\***)consumer, NULL);

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

**return** 0;

}

运行结果：

producer thread send notify signal. 1 recv producer thread notify signal. 1

producer thread send notify signal. 2 recv producer thread notify signal. 2

producer thread send notify signal. 3 recv producer thread notify signal. 3

**信号量**

信号量适用于控制一个仅支持有限个用户的共享资源。用于保持在0至指定最大值之间的一个计数值。当线程完成一次对该semaphore对象的等待时，该计数值减一；当线程完成一次对semaphore对象的释放时，计数值加一。当计数值为0时，线程挂起等待，直到计数值超过0.

主要函数如下：

**#include** <semaphore.h>

**int** **sem\_init**(sem\_t **\***sem, **int** pshared, **unsigned** **int** value);

**int** **sem\_wait**(sem\_t **\***sem);

**int** **sem\_trywait**(sem\_t **\***sem);

**int** **sem\_post**(sem\_t **\*** sem);

**int** **sem\_destroy**(sem\_t **\*** sem);

代码示例如下：

**#include**<sys/types.h>

**#include**<unistd.h>

**#include**<stdlib.h>

**#include**<stdio.h>

**#include**<pthread.h>

**#include**<errno.h>

**#include**<string.h>

**#include**<semaphore.h>

**#define NUM 5**

**int** queue[NUM];

sem\_t psem, csem;

**void** **producer**(**void\*** arg) {

**int** pos **=** 0;

**int** num, count **=** 0;

**for** (**int** i**=**0; i**<**12; **++**i) {

num **=** rand() **%** 100;

count **+=** num;

sem\_wait(**&**psem);

queue[pos] **=** num;

sem\_post(**&**csem);

printf("producer: %d\n", num);

pos **=** (pos**+**1) **%** NUM;

sleep(rand()**%**2);

}

printf("producer count=%d\n", count);

}

**void** **consumer**(**void\*** arg){

**int** pos **=** 0;

**int** num, count **=** 0;

**for** (**int** i**=**0; i**<**12; **++**i) {

sem\_wait(**&**csem);

num **=** queue[pos];

sem\_post(**&**psem);

printf("consumer: %d\n", num);

count **+=** num;

pos **=** (pos**+**1) **%** NUM;

sleep(rand()**%**3);

}

printf("consumer count=%d\n", count);

}

**int** **main**() {

sem\_init(**&**psem, 0, NUM);

sem\_init(**&**csem, 0, 0);

pthread\_t tid[2];

pthread\_create(**&**tid[0], NULL, (**void\***)producer, NULL);

pthread\_create(**&**tid[1], NULL, (**void\***)consumer, NULL);

pthread\_join(tid[0], NULL);

pthread\_join(tid[1], NULL);

sem\_destroy(**&**psem);

sem\_destroy(**&**csem);

**return** 0;

}

信号量的执行逻辑：

当需要获取共享资源时，先检查信号量，如果值大于0，则值减1，访问共享资源，访问结束后，值加1，如果发现有被该信号量挂起的线程，则唤醒其中一个线程；如果检查到信号量为0，则挂起等待。

可参考源码[sem\_post.c](https://link.zhihu.com/?target=https%3A//code.woboq.org/userspace/glibc/nptl/sem_post.c.html)