更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

个人微信

公众号

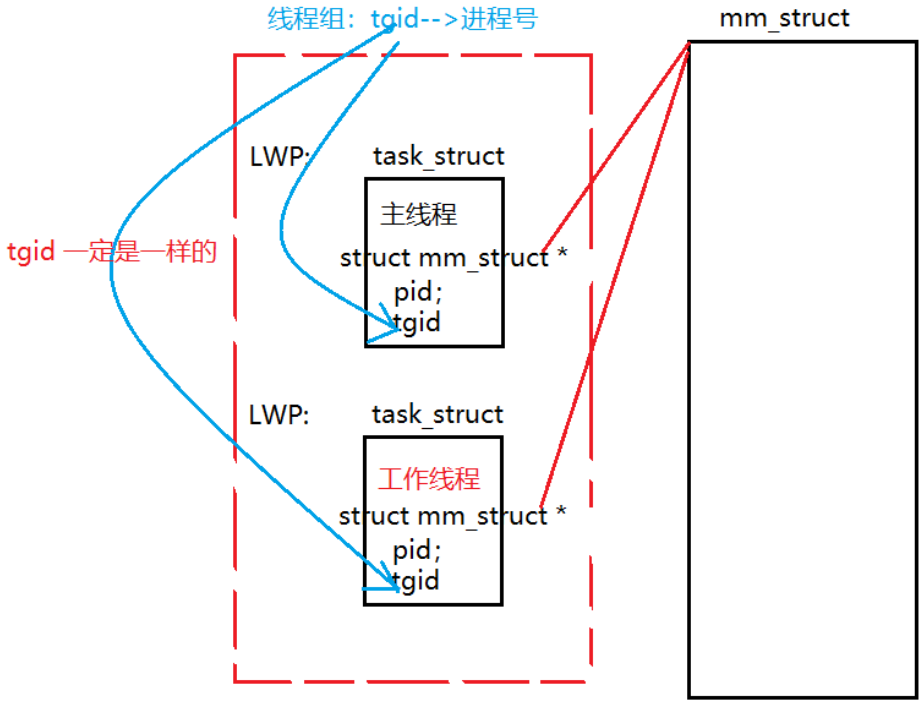
1.什么是线程?

linux 内核中是没有线程这个概念的，而是轻量级进程的概念：LWP。一般我们所说的线程概念是 C 库当中的概念。

1.1 线程是怎样描述的?

线程实际上也是一个 task_struct，工作线程拷贝主线程的 task_struct，然后共用主线程的 mm_struct。线程 ID 是在用 task_struct 中 pid 描述的，而 task_struct 中 tgid 是线程组 ID，表示线程属于该线程组，对于主线程而言，其 pid 和 tgid 是相同的，我们一般看到的进程 ID 就是 tgid。

即:



获取线程 ID 和主线程 ID 的值：

用户态	系统调用	mm_struct对应的结构
线程ID	pid_t gettid(void)	pid_t pid
进程ID	pid_t getpid(void)	pid_t tgid

但是获取该 gettid 系统调用接口并没有被封装起来，如果确实需要获取线程 ID，可使用：

```
#include <sys/syscall.h>
int TID = syscall(SYS_gettid);
```

则对线程组而言，所有的 tgid 一定是一样的，所有的 pid 一定是不一样的。主线程 pid 和 tgid 一样，工作线程 pid 和 tgid 一定不一样。

1.2 如何查看一个线程的 ID

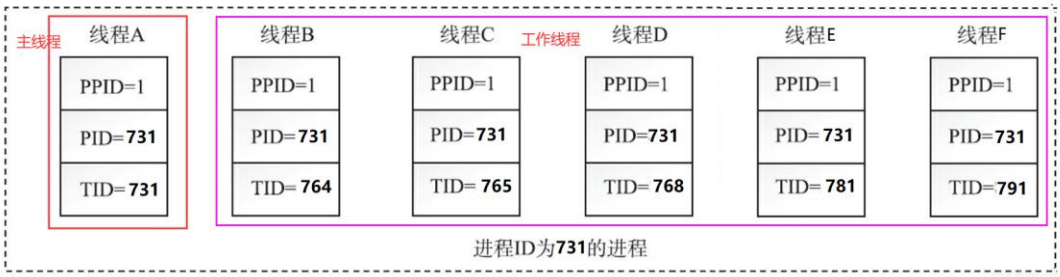
命令：ps -eLf

```
UID          PID    PPID     LWP  C NLWP STIME TTY          TIME CMD
polkitd      731       1     731  0    6 10:20 ?        00:00:01 /usr/lib/polkit-1/polkitd --no-debug
polkitd      731       1     764  0    6 10:20 ?        00:00:00 /usr/lib/polkit-1/polkitd --no-debug
polkitd      731       1     765  0    6 10:20 ?        00:00:00 /usr/lib/polkit-1/polkitd --no-debug
polkitd      731       1     768  0    6 10:20 ?        00:00:00 /usr/lib/polkit-1/polkitd --no-debug
polkitd      731       1     781  0    6 10:20 ?        00:00:00 /usr/lib/polkit-1/polkitd --no-debug
polkitd      731       1     791  0    6 10:20 ?        00:00:00 /usr/lib/polkit-1/polkitd --no-debug
```

LWP：线程ID，即gettid()返回的值

NLWP：线程组内线程的个数

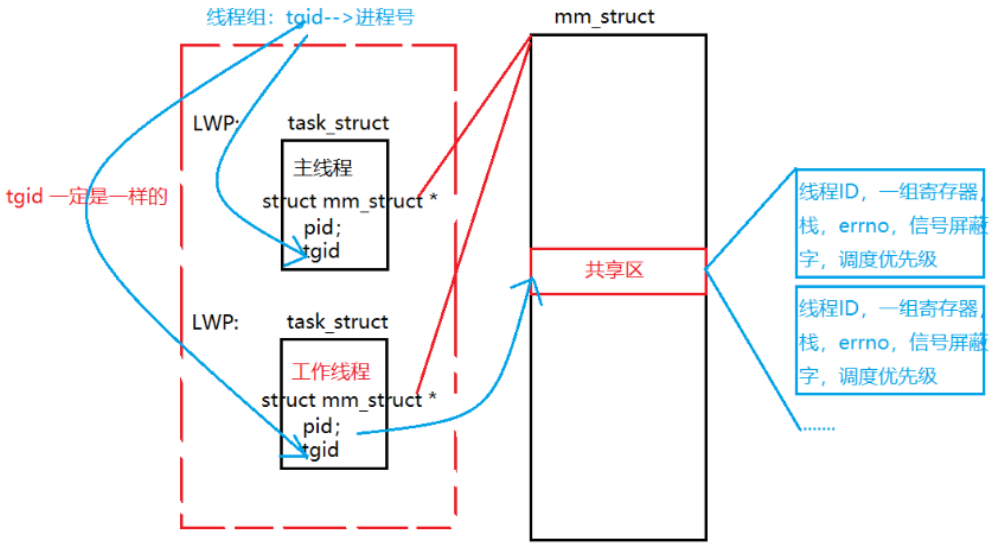
上述 polkitd 进程是多线程的，进程 ID 为 731，进程内有 6 个线程，线程 ID 为 731，764，765，768，781，791。



1.3 多线程如何避免调用栈混乱的问题？

工作线程和主线程共用一个 mm_struct，如果都向栈中压栈，必然会导致调用栈出错。

实际上工作线程压栈是压了共享区，该共享区包含了许多线程独有的资源。如图：



每一个线程，默认在共享区中占有的空间为 8M，可以使用 ulimit -s 修改。

进程是资源分配的基本单位，线程是调度的基本单位。

1.3.1 线程独有资源

- 线程 ID
- 一组寄存器
- errno
- 信号屏蔽字
- 调度优先级

1.3.2 线程共享资源和环境

- 文件描述符表
- 信号的处理方式
- 当前工作目录
- 用户 id 和组 id

1.4 为什么要有多线程?

举个生活中的例子， 这就好比去银行办理业务。 到达银行后， 首先取一个号码， 然后坐下来安心等待。 这时候你一定希望， 办理业务的窗口越多越好。 如果把整个营业大厅当成一个进程的话， 那么每一个窗口就是一个工作线程。

1.4.1 线程带来的优势

1、线程会共享内存地址空间。

2、创建线程花费的时间要少于创建进程花费的时间。

3、终止线程花费的时间要少于终止进程花费的时间。

4、线程之间上下文切换的开销， 要小于进程之间的上下文切换。

5、线程之间数据的共享比进程之间的共享要简单。

6、充分利用多处理器的可并行数量。（线程会提高运行效率，但当线程多到一定程度后，可能会导致效率下降，因为会有线程调度切换。）

1.4.2 线程带来的缺点

健壮性降低：多个线程之中， 只要有一个线程不够健壮存在 bug（如访问了非法地址引发的段错误） ， 就会导致进程内的所有线程一起完蛋。

线程模型作为一种并发的编程模型， 效率并没有想象的那么高， 会出现复杂度高、 易出错、 难以测试和定位的问题。

1.5 注意

1、并不是只有主线程才能创建线程， 被创建出来的线程同样可以创建线程。

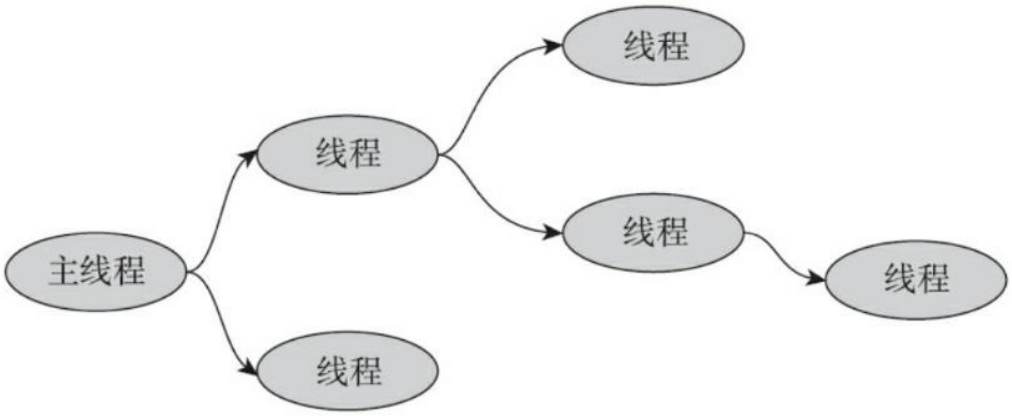
2、不存在类似于 fork 函数那样的父子关系， 大家都归属于同一个线程组，进程 ID 都相等， group_leader 都指向主线程， 而且各有各的线程 ID。

通过 group_leader 指针， 每个线程都能找到主线程。 主线程存在一个链表头，后面创建的每一个线程都会链入到该双向链表中。

3、并非只有主线程才能调用 pthread_join 连接其他线程， 同一线程组内的任意线程都可以对某线程执行 pthread_join 函数。

4、并非只有主线程才能调用 pthread_detach 函数， 其实任意线程都可以对同一线程组内的线程执行分离操作。

线程的对等关系：



同一线程组的线程，没有层次关系

2.线程创建

接口：int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routine) (void *), void *arg);

参数解释

1、thread：线程标识符，是一个出参

2、attr：线程属性

3、star_routine：函数指针，保存线程入口函数的地址

4、arg：给线程入口函数传参

返回值：成功返回 0，失败返回 error number

详解：

第一个参数是 pthread_t 类型的指针， 线程创建成功的话，会将分配的线程 ID 填入该指针指向的地址。 线程的后续操作将使用该值作为线程的唯一标识。

第二个参数是 pthread_attr_t 类型， 通过该参数可以定制线程的属性， 比如可以指定新建线程栈的大小、 调度策略等。 如果创建线程无特殊的要求， 该值也可以是 NULL， 表示采用默认属性。

第三个参数是线程需要执行的函数。 创建线程， 是为了让线程执行一定的任务。 线程创建成功之后， 该线程就会执行 start_routine 函数， 该函数之于线程， 就如同 main 函数之于主线程。

第四个参数是新建线程执行的 start_routine 函数的入参。

pthread_create 错误码及描述：

返回值	描述
EAGAIN	系统资源不够，或者创建线程的个数超过系统对一个进程中线程总数的限制
EINVAL	第二个参数attr值不合法
EPERM	没有合适的权限来设置调度策略或参数

2.1 传入参数 arg 的选择

传入参数	分析	是否可行
临时变量	临时变量的生命周期，临时变量的值会改变，传递临时变量有可能导致越界的问题	不可行
结构体对象	和临时变量相同	不可行
结构体指针	释放时，在线程不会使用该指针以后	可行
this指针		可行

不要使用临时变量传参，使用堆上开辟的变量可以。

例：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void *ThreadWork(void *arg)
{
  int *p = (int*)arg;
  printf("i am work thread:%p,    data:%d\n",pthread_self(),*p);
  pthread_exit(NULL);
}
int main()
{
  int i = 1;
  pthread_t tid;
  int ret = pthread_create(&tid,NULL,ThreadWork,(void*)&i);//不要传临时变量，这里是示范
  if(ret != 0)
  {
```

```
    perror("pthread_create");
    return -1;
  }
  while(1)
  {
    printf("i am main work thread\n");
    sleep(1);
  }
  return 0;
}
```

2.2 线程 ID 以及进程地址空间

线程获取自身的 ID：

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

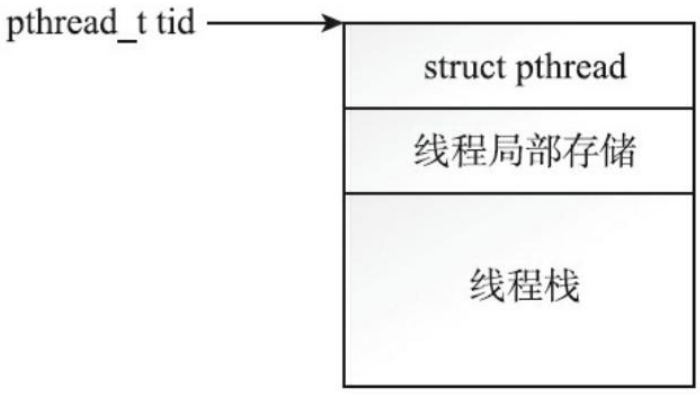
判断两个线程 ID 是否对应着同一个线程：

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

返回为 0 时，则表示两个线程为同一个线程，非 0 时，表示不是同一个线程。

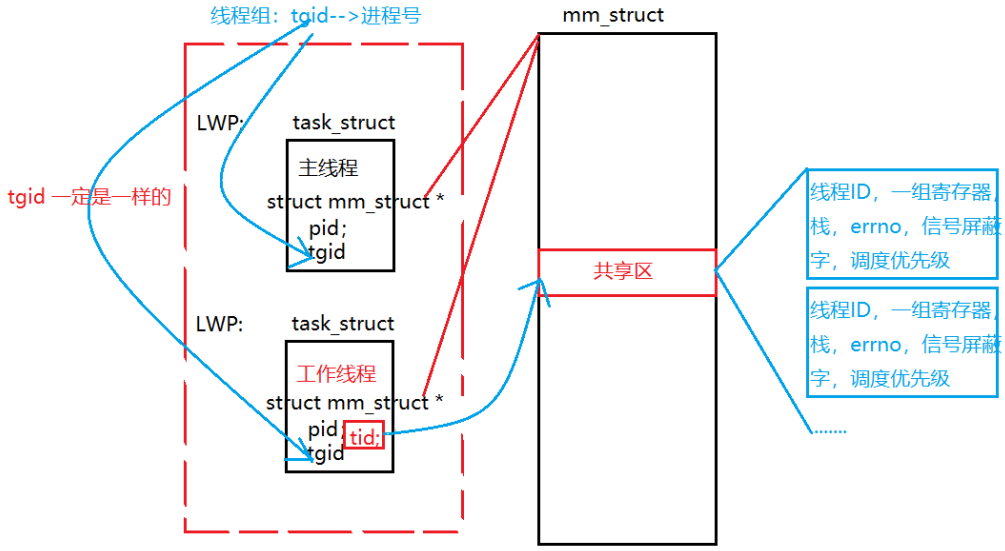
用户调用 pthread_create 函数时， 首先要为线程分配线程栈， 而线程栈的位置就落在共享区。 调用 mmap 函数为线程分配栈空间。 pthread_create 函数分配的 pthread_t 类型的线程 ID， 不过是分配出来的空间里的一个地址， 更确切地说是一个结构体的指针。

公众号:一口Linux



线程ID的本质是内存地址

即：



2.3 线程注意点

1、线程 ID 是进程地址空间内的一个地址， 要在同一个线程组内进行线程之间的比较才有意义。 不同线程组内的两个线程， 哪怕两者的 pthread_t 值是一样的， 也不是同一个线程。

2、线程 ID 就有可能会被复用：

1、线程退出。

2、线程组的其他线程对该线程执行了 pthread_join， 或者线程退出前将分离状态设置为已分离。

3、再次调用 pthread_create 创建线程。

2.4 线程创建出来的默认值

线程创建的第二个参数是 pthread_attr_t 类型的指针， pthread_attr_init 函数会将线程的属性重置成默认值。

线程属性及默认值：

属性	默认值	说明
contentionscope	PTHREAD_SCOPE_SYSTEM	进程调度相关，线程只支持在OS范围内竞争CPU资源
Detach state	PTHREAD_CREATE_DETACHED	可分离状态
Stack address	NULL	不指定线程开辟的基地址
Stack size	8196(KB)	默认线程栈大小为8M
Guard size	0	警戒缓冲区
Scheduling priority	0	进程调度相关，优先级为0
Scheduling policy	SCHED_OTHER	进程调度相关，调度策略为SCHED_OTHER
Inherit scheduler	PTHREAD_EXPLICIT_SCHED	进程调度相关，继承启动进程的调度策略

如果确实需要很多的线程， 可以调用接口来调整线程栈的大小：

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t *attr,size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr,size_t
*stacksize);
```

3.线程终止

线程终止，但进程不会终止的方法：

1、入口函数的 return 返回，线程就退出了

2、线程调用 pthread_exit(NULL)，谁调用谁退出

```
#include <pthread.h>

void pthread_exit(void *retval);
```

参数：retval 是返回信息，”临终遗言“，可以给可以不给

该变量不能使用临时变量。

可使用：全局变量、堆上开辟的空间、字符串常量。

pthread_exit 和线程启动函数（start_routine） 执行 return 是有区别的。在 start_routine 中调用的任何层级的函数执行 pthread_exit（） 都会引发线程退出， 而 return， 只能是在 start_routine 函数内执行才能导致线程退出。

3、其它线程调用了 pthread_cancel 函数取消了该线程

```
int pthread_cancel(pthread_t thread);
```

thread：线程标识符

调用该函数的执行流可以取消其它线程，但是需要知道其它线程的线程标识符，也可以执行流自己取消自己，传入自己的线程标识符。

如果线程组中的任何一个线程调用了 exit 函数， 或者主线程在 main 函数中执行了 return 语句， 那么整个线程组内的所有线程都会终止。

4.线程等待

4.1 线程等待接口

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

参数	解释
thread	要等待的线程标识符
retval	接收返回值：若是return退出的，接受入口函数的返回值，pthread_exit退出的，接受该函数的参数，pthread_cancel退出的，void** 保存的是PTHREAD_CANCLED = （void *）-1

调用该函数，该执行流在等待线程退出的时候，该执行流是阻塞在 pthread_joind 当中的。

4.2 线程等待和进程等待的不同

第一点不同之处是进程之间的等待只能是父进程等待子进程， 而线程则不然。线程组内的成员是对等的关系， 只要是在一个线程组内， 就可以对另外一个线程执行连接（join） 操作。

第二点不同之处是进程可以等待任一子进程的退出 ， 但是线程的连接操作没有类似的接口， 即不能连接线程组内的任一线程， 必须明确指明要连接的线程的线程 ID。

pthread_join()错误码：

返回值	说明
ESRCH	传入的线程ID不存在，查无此线程
EINVAL	线程不是一个joinable线程
EINVAL	已有其它线程捷足先登，链接目标线程
EDEADLK	死锁，如自己链接自己

4.3 为什么要等待退出的线程?

如果不连接已经退出的线程， 会导致资源无法释放。 所谓资源指的又是什么呢？

1、已经退出的线程， 其空间没有被释放， 仍然在进程的地址空间之内。

2、新创建的线程， 没有复用刚才退出的线程的地址空间。

如果不执行连接操作， 线程的资源就不能被释放， 也不能被复用， 这就造成了资源的泄漏。

纵然调用了 pthread_join， 也并没有立即调用 munmap 来释放掉退出线程的栈， 它们是被后建的线程复用了。 释放线程资源的时候， 若进程可能再次创建线程， 而频繁地 munmap 和 mmap 会影响性能， 所以将该栈缓存起来， 放到一个链表之中， 如果有新的创建线程的请求， 会首先在栈缓存链表中寻找空间合适的栈， 有的话， 直接将该栈分配给新创建的线程。

例：

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

```
#include <unistd.h>
#include <sys/syscall.h>


void *ThreadWork(void *arg)
{
  int *p = (int*)arg;
  printf("pid :  %d\n",syscall(SYS_gettid));
  printf("i am work thread:%p,    data:%d\n",pthread_self(),*p);
  sleep(3);
  pthread_exit(NULL);
}

int main()
{
  int i = 1;
  pthread_t tid;
  int ret = pthread_create(&tid,NULL,ThreadWork,(void*)&i);//不要传临时变量，这里是示范
  if(ret != 0)
  {
    perror("pthread_create");
    return -1;
  }
  pthread_join(tid,NULL);//线程等待
  while(1)
  {
    printf("i am main work thread\n");
    sleep(1);
  }
  return 0;
}
```

5.线程分离

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

默认情况下， 新创建的线程处于可连接（Joinable） 的状态， 可连接状态的线程退出后， 需要对其执行连接操作， 否则线程资源无法释放， 从而造成资源泄漏。

公众号:一口Linux

如果其他线程并不关心线程的返回值， 那么连接操作就会变成一种负担： 你不需要它， 但是你不去执行连接操作又会造成资源泄漏。 这时候你需要的东西只是：线程退出时， 系统自动将线程相关的资源释放掉， 无须等待连接。

可以是线程组内其他线程对目标线程进行分离， 也可以是线程自己执行 pthread_detach 函数。

线程的状态之中， 可连接状态和已分离状态是冲突的， 一个线程不能既是可连接的， 又是已分离的。 因此， 如果线程处于已分离的状态， 其他线程尝试连接线程时， 会返回 EINVAL 错误。

pthread_detach 错误码：

返回值	说明
ESRCH	传入线程的ID不存在，无此线程
EINVAL	线程不是一个joinable线程，已经处于分离状态

注意：这里的已分离不是指线程失去控制，不归线程组管，而是指线程退出后，系统会自动释放线程资源。若是线程组内的任意线程执行了 exit 函数，即使是已分离的线程，也仍会收到影响，一并退出。

6.线程安全

线程安全中涉及到的概念：

临界资源：多线程中都能访问到的资源
临界区：每个线程内部，访问临界资源的代码，就叫临界区

6.1 什么是线程不安全?

多个线程访问同一块临界资源，导致资源产生二义性的现象。

6.1.1 举一个例子

- 假设现在有两个线程 A 和 B，单核 CPU 的情况下，此时有一个 int 类型的全局变量为 100，A 和 B 的入口函数都要对这个全局变量进行–操作。

- 线程 A 先拿到 CPU 资源后，对全局变量进行–操作并不是原子性操作，也就是意味着，A 在执行–的过程中有可能会被打断。假设 A 刚刚将全局变量的值读到寄存器当中，就被切换出去了，此时程序计数器保存了下一条执行的指令，上下文信息保

存寄存器中的值，这两个东西是用来线程 A 再次拿到 CPU 资源后，恢复现场使用的。

- 此时，线程 B 拿到了 CPU 资源，对全局变量进行了–操作，并且将 100 减为了 99，回写到了内存中。

- A 再次拥有了 CPU 资源后，恢复现场，继续往下执行，从寄存器中读到的值仍为 100，减完之后为 99，回写到内存中为 99。

上述例子中，线程 A 和 B 都对全局变量进行了 - 操作，全局变量的值应该变为 98，但程序现在实际的结果为 99，所以这就导致了线程不安全。

6.2 如何解决线程不安全现象?

解决方案只需做到下述三点即可：

1、代码必须要有互斥的行为： 当一个线程正在临界区中执行时， 不允许其他线程进入该临界区中。

2、如果多个线程同时要求执行临界区的代码， 并且当前临界区并没有线程在执行， 那么只能允许一个线程进入该临界区。

3、如果线程不在临界区中执行， 那么该线程不能阻止其他线程进入临界区。

则本质上，我们需要对该临界区加一把锁：

公众号:一口 Linux

用锁来保护临界区

锁是一个很普遍的需求， 当然用户可以自行实现锁来保护临界区。 但是实现一个正确并且高效的锁非常困难。 纵然抛下高效不谈， 让用户从零开始实现一个正确的锁也并不容易。 正是因为这种需求具有普遍性， 所以 Linux 提供了互斥量。

6.3 互斥量接口

6.3.1 互斥量的初始化

1、静态分配：

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

2、动态分配：

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

变量	说明
mutex	要初始化的互斥量
attr	设置互斥量的属性，默认为NULL

调用 int pthread_mutex_init（）函数后，互斥量是处于没有加锁的状态。

6.3.2 互斥量的销毁

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

注意：

1、使用 PTHREAD_MUTEX_INITIALIZER 初始化的互斥量无须销毁。

2、不要销毁一个已加锁的互斥量， 或者是真正配合条件变量使用的互斥量。

3、已经销毁的互斥量， 要确保后面不会有线程再尝试加锁。

当互斥量处于已加锁的状态， 或者正在和条件变量配合使用， 调用 pthread_mutex_destroy 函数会返回 EBUSY 错误码。

6.3.3 互斥量的加锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const
struct timespec *restrict abs_timeout);
```

第一个接口：int pthread_mutex_lock(pthread_mutex_t *mutex);

1、该接口是阻塞加锁接口。

2、mutex 为传入互斥锁变量的地址

3、如果 mutex 当中的计数器为 1，pthread_mutex_lock 接口就返回了，表示加锁成功，同时计数器当中的值会被更改为 0.

4、如果 mutex 当中的计数器为 0，pthread_mutex_lock 接口就阻塞了，pthread_mutex_lock 接口没有返回了，阻塞在函数内部，直到加锁成功

第二个接口：int pthread_mutex_trylock(pthread_mutex_t *mutex);

1、该接口为非阻塞接口

2、mutex 中计数器为 1 时，加锁成功，计数器置为 0，然后返回

3、mutex 中计数器为 0 时，加锁失败，但也会返回，此时加锁是失败状态，一定不要去访问临界资源

4、非阻塞接口一般都需要搭配循环来使用。

第三个接口：int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict abs_timeout);

1、带有超时时间的加锁接口

2、不能直接获取互斥锁的时候，会等待 abs_timeout 时间

3、如果在这个时间内加锁成功了，直接返回，不需要再继续等待剩余的时间，并且表示加锁成功

4、如果超出了该时间，也返回了，但是加锁失败了，需要循环加锁

上述三个加锁接口，第一个接口用的最多。

6.3.4 互斥量的解锁

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

1. 对上述所有的加锁接口，都可使用该函数解锁
2. 解锁的时候，会将互斥锁当中计数器的值从 0 变为 1，表示其它线程可以获取互斥量

6.4 互斥锁的本质

1、在互斥锁内部有一个计数器，其实就是互斥量，计数器的值只能为 0 或者为 1

2、当线程获取互斥锁的时候，如果计数器当前值为 0，表示当前线程不能获取到互斥锁，也就是没有获取到互斥锁，就不要去访问临界资源

3、当前线程获取互斥锁的时候，如果计数器当前值为 1，表示当前线程可以获取到互斥锁，也就是意味着可以访问临界资源

公众号:一口Linux

本文档版权属于公众号:一口 Linux 号主所有，转载传播请联系作者 vx:yikoupeng

6.5 互斥锁中的计数器如何保证了原子性?

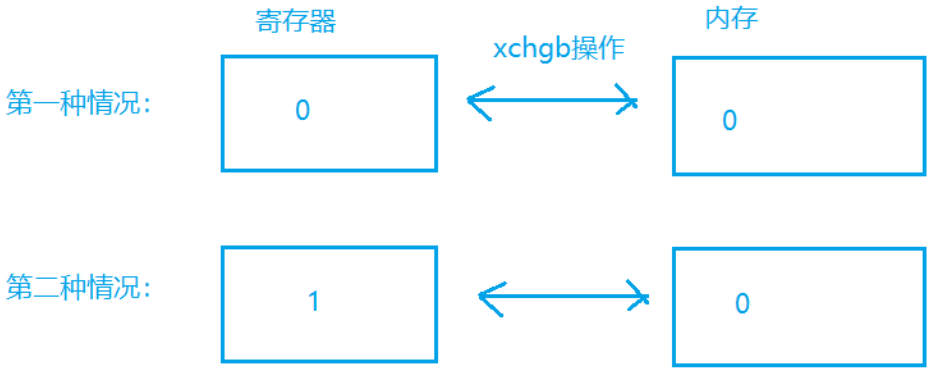
获取锁资源的时候（加锁）：

1、寄存器当中值直接赋值为 0

2、将寄存器当中的值和计数器当中的值进行交换

3、判断寄存器当中的值，得出加锁结果

两种情况：



xchgb操作：确保了交换操作原子性，要么没做，要么已经完成了

寄存器当中值为1的时候，表示可以加锁

寄存器当中值为0的时候，表示不可以加锁

例：4 个线程，对同一个全局变量进行减减操作

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#define NUMBER 4
int g_val = 100;

pthread_mutex_t mutex;//定义互斥锁


void *ThreadWork(void *arg)
{
  int *p = (int*)arg;
```

```
  pthread_detach(pthread_self());//自己分离自己，不用主线程回收它的资源了
  while(1)
  {
    pthread_mutex_lock(&mutex);//加锁
    if(g_val > 0)
    {
      printf("i am pid : %d,i get g_val : %d\n",(int)syscall(SYS_gettid),g_val);
      --g_val;
      usleep(2);
    }
    else{
      pthread_mutex_unlock(&mutex);//在所有可能退出的地方，进行解锁
      break;
    }
    pthread_mutex_unlock(&mutex);//解锁

  }
  pthread_exit(NULL);
}

int main()
{
  pthread_t tid[NUMBER];
  pthread_mutex_init(&mutex,NULL);//互斥锁初始化
  int i = 0;
  for(;i < NUMBER;++i)
  {
    int ret = pthread_create(&tid[i],NULL,ThreadWork,(void*)&g_val);//不要传临时变量，这里是示范
    if(ret != 0)
    {
      perror("pthread_create");
      return -1;
     }
  }
  //pthread_join(tid,NULL);//线程等待
  //pthread_detach(tid);//线程分离
  pthread_mutex_destroy(&mutex);//销毁互斥锁
  while(1)
  {
    printf("i am main work thread\n");
```

公众号:一口Linux

```
    sleep(1);
  }
  return 0;
}
```

6.6 互斥锁公平嘛?

互斥锁是不公平的。

内核维护等待队列， 互斥量实现了大体上的公平； 由于等待线程被唤醒后，并不自动持有互斥量， 需要和刚进入临界区的线程竞争（抢锁）， 所以互斥量并没有做到先来先服务。

6.7 互斥锁的类型

1、PTHREAD_MUTEX_NORMAL： 最普通的一种互斥锁。 它不具备死锁检测功能，如线程对自己锁定的互斥量再次加锁， 则会发生死锁。

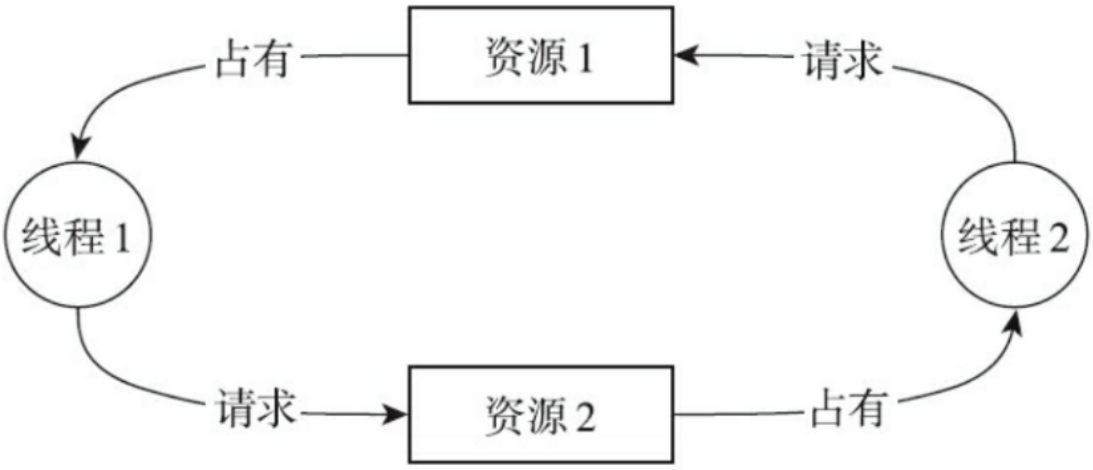
2、
PTHREAD_MUTEX_RECURSIVE_NP： 支持递归的一种互斥锁， 该互斥量的内部维护有互斥锁的所有者和一个锁计数器。 当线程第一次取到互斥锁时， 会将锁计数器置 1， 后续同一个线程再次执行加锁操作时， 会递增该锁计数器的值。 解锁则递减该锁计数器的值， 直到降至 0， 才会真正释放该互斥量，此时其他线程才能获取到该互斥量。 解锁时， 如果互斥量的所有者不是调用解锁的线程， 则会返回 EPERM。

3、
PTHREAD_MUTEX_ERRORCHECK_NP： 支持死锁检测的互斥锁。 互斥量的内部会记录互斥锁的当前所有者的线程 ID（调度域的线程 ID） 。 如果互斥量的持有线程再次调用加锁操作， 则会返回 EDEADLK。 解锁时， 如果发现调用解锁操作的线程并不是互斥锁的持有者， 则会返回 EPERM。

4、自旋锁，自旋锁采用了和互斥量完全不同的策略， 自旋锁加锁失败， 并不会让出 CPU， 而是不停地尝试加锁， 直到成功为止。 这种机制在临界区非常小且对临界区的争夺并不激烈的场景下， 效果非常好。自旋锁的效果好， 但是副作用也大， 如果使用不当， 自旋锁的持有者迟迟无法释放锁， 那么，自旋接近于死循环， 会消耗大量的 CPU 资源， 造成 CPU 使用率飙高。 因此，使用自旋锁时， 一定要确保临界区尽可能地小， 不要有系统调用， 不要调用 sleep。 使用 strcpy/memcpy 等函数也需要谨慎判断操作内存的大小， 以及是否会引起缺页中断。

5、PTHREAD_MUTEX_ADAPTIVE_NP：自适应锁，首先与自旋锁一样， 持续尝试获取， 但过了一定时间仍然不能申请到锁， 就放弃尝试， 让出 CPU 并等待。PTHREAD_MUTEX_ADAPTIVE_NP 类型的互斥量， 采用的就是这种机制。

6.8 死锁和活锁



死锁的产生（简单场景）

线程 1 已经成功拿到了互斥量 1， 正在申请互斥量 2， 而同时在另一个 CPU 上，线程 2 已经拿到了互斥量 2， 正在申请互斥量 1。 彼此占有对方正在申请的互斥量，结局就是谁也没办法拿到想要的互斥量， 于是死锁就发生了。

6.8.1 死锁概念

死锁是指在一组进程中的各个进程均占有不会释放的资源，但因互相申请被其它进程所占有不会释放的资源而处于一种永久等待的状态。

6.8.2 死锁的四个必要条件

1、互斥条件：一个资源只能被一个执行流使用

2、请求与保持条件：一个执行流因请求资源而阻塞时，对已获得的资源不会释放

3、不剥夺条件：一个执行流已获得的资源，在未使用完之前，不能强行剥夺

4、循环等待条件：若干执行流之间形成一种头尾相接的循环等待资源的关系

6.8.3 避免死锁

1、破坏死锁的四个必要条件（实际上只能破坏条件 2 和 4）

2、加锁顺序一致（按照先后顺序申请互斥锁）

3、避免未释放锁的情况

4、资源一次性分配

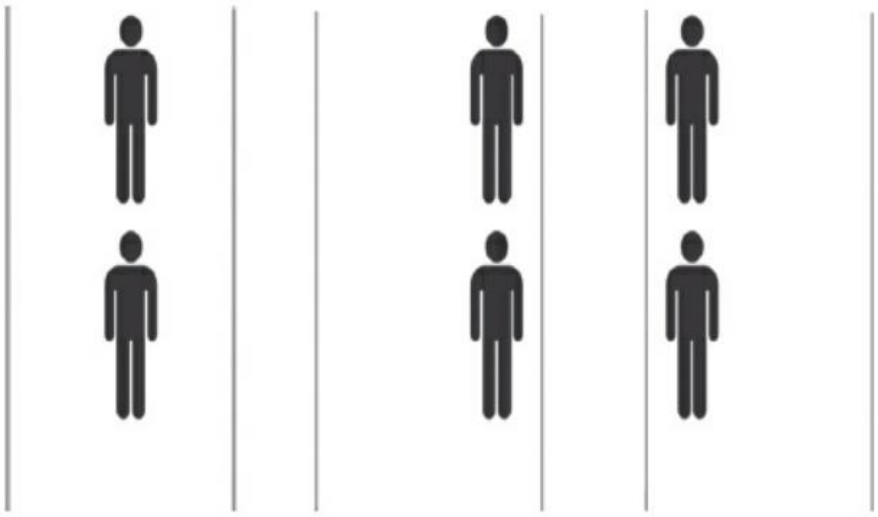
6.8.4 活锁

避免死锁的另一种方式是尝试一下，如果取不到锁就返回。

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,const
struct timespec *restrict abs_timeout);
```

这两个函数反映了一种，不行就算了的思想。

trylock 不行就回退的思想有可能会引发活锁（live lock） 。 生活中也经常遇到两个人迎面走来， 双方都想给对方让路， 但是让的方向却不协调， 反而互相堵住的情况 。 活锁现象与这种场景有点类似。



让路总让到一起，变成堵路

线程 1 首先申请锁 mutex_a 后， 之后尝试申请 mutex_b， 失败以后， 释放 mutex_a 进入下一轮循环， 同时线程 2 会因为尝试申请 mutex_a 失败，而释放 mutex_b， 如果两个线程恰好一直保持这种节奏， 就可能在很长的时间内两者都一次次地擦肩而过。 当然这毕竟不是死锁， 终究会有一个线程同时持有两把锁而结束这种情况。 尽管如此， 活锁的确会降低性能。

6.8.5 死锁调试

查看多个线程堆栈：thread apply all bt
跳转到线程中：t 线程号
查看具体的调用堆栈：f 堆栈号
直接从 pid 号用 gdb 调试：gdb attach pid

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#define NUMBER 2

pthread_mutex_t mutex1;//定义互斥锁
pthread_mutex_t mutex2;


void *ThreadWork1(void *arg)
{
  int *p = (int*)arg;
  pthread_mutex_lock(&mutex1);

  sleep(2);

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

void *ThreadWork2(void *arg)
{
  int *p = (int*)arg;
  pthread_mutex_lock(&mutex2);

  sleep(2);

  pthread_mutex_lock(&mutex1);
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
  return NULL;
}
int main()
```

公众号:一口Linux

```
{
  pthread_t tid[NUMBER];
  pthread_mutex_init(&mutex1,NULL);//互斥锁初始化
  pthread_mutex_init(&mutex2,NULL);//互斥锁初始化
  int i = 0;
  int ret = pthread_create(&tid[0],NULL,ThreadWork1,(void*)&i);
  if(ret != 0)
  {
    perror("pthread_create");
    return -1;
  }
  ret = pthread_create(&tid[1],NULL,ThreadWork2,(void*)&i);
  if(ret != 0)
  {
    perror("pthread_create");
    return -1;
  }
  //pthread_join(tid,NULL);//线程等待
  //pthread_join(tid,NULL);//线程等待
  //pthread_detach(tid);//线程分离
  pthread_join(tid[0],NULL);
  pthread_join(tid[1],NULL);
  pthread_mutex_destroy(&mutex1);//销毁互斥锁
  pthread_mutex_destroy(&mutex2);//销毁互斥锁
  while(1)
  {
    printf("i am main work thread\n");
    sleep(1);
  }
  return 0;
}
```

在上述代码中，一定会出现死锁，线程 1 拿到了互斥锁 1，又再去申请线程 2 的互斥锁 2，线程 2 拿到了互斥锁 2 又再去申请线程 1 的互斥锁 1。

开始调试：

1、找到进程号

公众号：一口Linux

```
[root@localhost test]# ps aux | grep ./create
test       5028  0.0  0.0  22896   388 pts/0    Sl+  11:22   0:00 ./create
root       5035  0.0  0.0 112824   992 pts/1    R+   11:22   0:00 grep --color=auto ./create
[root@localhost test]#
```

2、开始调试

```
[root@localhost test]# ps aux | grep ./create
test       5028  0.0  0.0  22896   388 pts/0    Sl+  11:22   0:00 ./create
root       5035  0.0  0.0 112824   992 pts/1    R+   11:22   0:00 grep --color=auto ./create
[root@localhost test]# gdb attach 5028
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-100.el7
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
attach: 没有那个文件或目录.
Attaching to process 5028
Reading symbols from /home/test/code/11_24/deadlock/create...done.
Reading symbols from /lib64/libpthread.so.0...(no debugging symbols found)...done.
[New LWP 5030]
[New LWP 5029]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Loaded symbols for /lib64/libpthread.so.0
Reading symbols from /lib64/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00007f3ef6cf3017 in pthread_join () from /lib64/libpthread.so.0
Missing separate debuginfos, use: debuginfo-install glibc-2.17-307.el7.1.x86_64
(gdb)
```

使用gdb开始调试

3、查看多个线程堆栈

```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
attach: 没有那个文件或目录.
Attaching to process 5028
Reading symbols from /home/test/code/11_24/deadlock/create...done.
Reading symbols from /lib64/libpthread.so.0...(no debugging symbols found)...done.
[New LWP 5030]
[New LWP 5029]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Loaded symbols for /lib64/libpthread.so.0
Reading symbols from /lib64/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
0x00007f3ef6cf3017 in pthread_join () from /lib64/libpthread.so.0
Missing separate debuginfos, use: debuginfo-install glibc-2.17-307.el7.1.x86_64
(gdb) thread apply all bt        查看多个线程堆栈

Thread 3 (Thread 0x7f3ef691b700 (LWP 5029)): 工作线程2
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f3ef6cf3e9b in _L_lock_883 () from /lib64/libpthread.so.0
#2  0x00007f3ef6cf3d68 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x000000000040081f in ThreadWork1 (arg=0x7ffee23a1c8c) at deadlock.c:19
#4  0x00007f3ef6cf1ea5 in start_thread () from /lib64/libpthread.so.0
#5  0x00007f3ef6a1a8dd in clone () from /lib64/libc.so.6
        工作线程1
Thread 2 (Thread 0x7f3ef611a700 (LWP 5030)):
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f3ef6cf3e9b in _L_lock_883 () from /lib64/libpthread.so.0
#2  0x00007f3ef6cf3d68 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x000000000040086c in ThreadWork2 (arg=0x7ffee23a1c8c) at deadlock.c:32
#4  0x00007f3ef6cf1ea5 in start_thread () from /lib64/libpthread.so.0
#5  0x00007f3ef6a1a8dd in clone () from /lib64/libc.so.6
        主线程
Thread 1 (Thread 0x7f3ef7111740 (LWP 5028)):
#0  0x00007f3ef6cf3017 in pthread_join () from /lib64/libpthread.so.0
#1  0x0000000000400937 in main () at deadlock.c:58
(gdb)
```

4、跳转到线程中

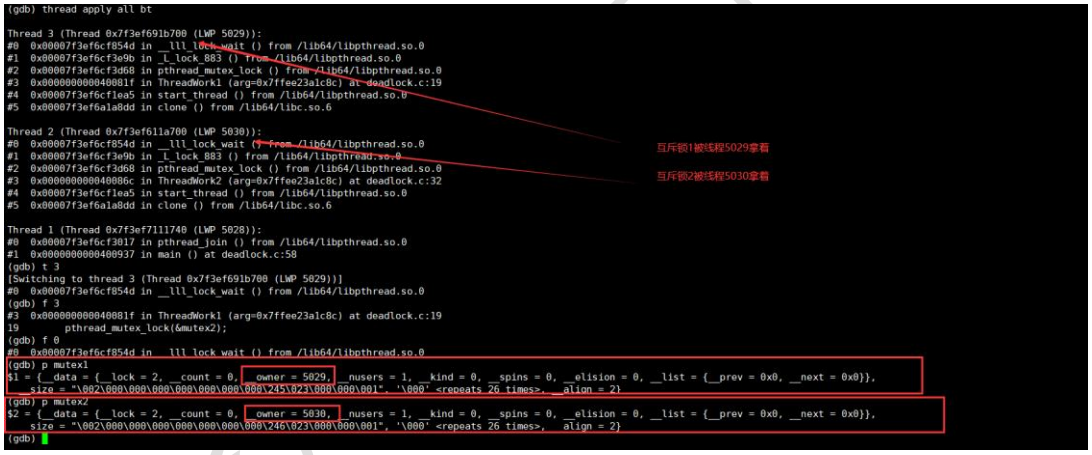
```
(gdb) thread apply all bt

Thread 3 (Thread 0x7f3ef691b700 (LWP 5029)):
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f3ef6cf3e9b in _L_lock_883 () from /lib64/libpthread.so.0
#2  0x00007f3ef6cf3d68 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x000000000040081f in ThreadWork1 (arg=0x7ffee23a1c8c) at deadlock.c:19
#4  0x00007f3ef6cf1ea5 in start_thread () from /lib64/libpthread.so.0
#5  0x00007f3ef6a1a8dd in clone () from /lib64/libc.so.6

Thread 2 (Thread 0x7f3ef611a700 (LWP 5030)):
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f3ef6cf3e9b in _L_lock_883 () from /lib64/libpthread.so.0
#2  0x00007f3ef6cf3d68 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x000000000040086c in ThreadWork2 (arg=0x7ffee23a1c8c) at deadlock.c:32
#4  0x00007f3ef6cf1ea5 in start_thread () from /lib64/libpthread.so.0
#5  0x00007f3ef6a1a8dd in clone () from /lib64/libc.so.6

Thread 1 (Thread 0x7f3ef7111740 (LWP 5028)):
#0  0x00007f3ef6cf3017 in pthread_join () from /lib64/libpthread.so.0
#1  0x0000000000400937 in main () at deadlock.c:58
(gdb) t 3    跳转到3号线程中
[Switching to thread 3 (Thread 0x7f3ef691b700 (LWP 5029))]
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
(gdb)
```

5、查看具体调用堆栈

```
(gdb) thread apply all bt

Thread 3 (Thread 0x7f3ef691b700 (LWP 5029)):
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f3ef6cf3e9b in _L_lock_883 () from /lib64/libpthread.so.0
#2  0x00007f3ef6cf3d68 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x000000000040081f in ThreadWork1 (arg=0x7ffee23a1c8c) at deadlock.c:19
#4  0x00007f3ef6cf1ea5 in start_thread () from /lib64/libpthread.so.0
#5  0x00007f3ef6a1a8dd in clone () from /lib64/libc.so.6

Thread 2 (Thread 0x7f3ef611a700 (LWP 5030)):
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
#1  0x00007f3ef6cf3e9b in _L_lock_883 () from /lib64/libpthread.so.0
#2  0x00007f3ef6cf3d68 in pthread_mutex_lock () from /lib64/libpthread.so.0
#3  0x000000000040086c in ThreadWork2 (arg=0x7ffee23a1c8c) at deadlock.c:32
#4  0x00007f3ef6cf1ea5 in start_thread () from /lib64/libpthread.so.0
#5  0x00007f3ef6a1a8dd in clone () from /lib64/libc.so.6

Thread 1 (Thread 0x7f3ef7111740 (LWP 5028)):
#0  0x00007f3ef6cf3017 in pthread_join () from /lib64/libpthread.so.0
#1  0x0000000000400937 in main () at deadlock.c:58
(gdb) t 3
[Switching to thread 3 (Thread 0x7f3ef691b700 (LWP 5029))]
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
(gdb) f 3
#3  0x000000000040081f in ThreadWork1 (arg=0x7ffee23a1c8c) at deadlock.c:19
19        pthread_mutex_lock(&mutex2);
(gdb) f 0
#0  0x00007f3ef6cf854d in __lll_lock_wait () from /lib64/libpthread.so.0
(gdb)
```

查看具体的调用堆栈

6、查看互斥锁 1 和互斥锁 2，分别被谁拿着



6.9 读写锁

6.9.1 什么是读写锁？

大部分情况下，对于共享变量的访问特点：只是读取共享变量的值，而不是修改，只有在少数情况下，才会真正的修改共享变量的值。

在这种情况下，读请求之间是同步的，它们之间的并发访问是安全的。然而写请求必须锁住读请求和其它写请求。

即读线程可多个同时读，而写线程只允许同一时间内一个线程去写。

6.9.2 读写锁接口

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

//初始化
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
              const pthread_rwlockattr_t *restrict attr);
```

参数	解释
restrict rwlock	读写锁
restrict attr	读写锁的属性

读写锁的默认属性：

属性	值	解释
竞争范围	PTHREAD_PROCESS_PRIVATE	进程内部竞争读写锁
策略	PTHREAD_RWLOCK_PREFER_READER_NP	读者优先

对于调用 pthread_rwlock_init 初始化的读写锁，在不需要读写锁的时候，需要调用 pthread_rwlock_destroy 销毁。

6.9.3 读者加锁

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock); //阻塞类型的读加锁接口
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock); //非阻塞类型的读加锁接口
```

最大的好处就是，允许多个线程以只读加锁的方式获取到读写锁；

本质上，读写锁的内部维护了一个引用计数，每当线程以读方式获取读写锁时，该引用计数+1；

当释放以读加锁的方式的读写锁时，会先对引用计数进行-1，直到引用计数的值为 0 的时候，才真正释放了这把读写锁。

6.9.4 写者加锁

```
#include <pthread.h>
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);// 非阻塞写
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);//阻塞写
```

写锁用的是独占模式，如果当前读写锁被某写线程占用着，则不允许任何读锁通过请求，也不允许任何写锁请求通过，读锁请求和写锁请求都要陷入阻塞，直到线程释放写锁。

6.9.5 解锁

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

不论是读者加锁还是写者加锁，都采用该接口进行解释。

读者解锁，只有当引用计数为 0 的时候，才真正释放了读写锁。

6.9.6 读写锁的竞争策略

对于读写锁而言，目前有两种策略，读者优先和携着优先；

读写锁的类型有如下几种：

```
PTHREAD_RWLOCK_PREFER_READER_NP, //读者优先
PTHREAD_RWLOCK_PREFER_WRITER_NP, //很唬人， 但是也是读者优先
PTHREAD_RWLOCK_PREFER_WRITER_NONRECURSIVE_NP, //写者优先
PTHREAD_RWLOCK_DEFAULT_NP = PTHREAD_RWLOCK_PREFER_READER_NP
```

读者优先：读锁来请求可以立即响应，只要有一个读锁没完成，那么写锁就无法写。这种策略是不公平的，极端情况下，写现场很可能被饿死，即线程总是拿不到锁资源。

写者优先：只要线程申请了写锁，那么在写锁后面到来的读锁请求就会统统被阻塞，不能先于写锁拿到锁。

读写锁实现中的变量及含义

变量	含义
_lock	管理读写锁全局竞争的锁，无论是读锁写锁还是解锁，都会执行互斥
_writer	写锁持有者的线程ID，如果为0，则表示当前无线程持有写锁
_nr_readers	读锁持有线程的个数
_nr_readers_queued	读锁的排队等待线程的个数
_nr_writers_queue	写锁的排队等待线程的个数

对于读请求而言：如果

1. 无线程持有写锁，即_writer = 0.

2. 采用读者优先策略或者当前没有写锁申请请求，即 _nr_writers_queue = 0

3. 当满足这两个条件时，读锁请求立即获得读锁，返回之前执行 _nr_readers++，表示多了一个线程正在读

4. 不满足这两个条件时，执行_nr_readers_queued++，表示增加了一个读锁等待者，然后调用 futex，陷入阻塞。醒来之后，执行_nr_readers_queued- -，再次判断是否满足条件 1，2

对于写请求而言：如果

1. 无线程持有写锁，即_writer = 0.

2. 没有线程持有读锁，即_nr_readers = 0.

3. 如果上述条件满足，就会立即拿到锁，将_writer 置为当前线程的 ID

4. 如果不满足，则执行_nr_writers_queue++， 表示增加了一个写锁等待者线程，然后执行 futex 陷入等待。醒来后，先执行_nr_writers_queue- -，再继续判断条件 1，2

对于解锁，如果当前是写锁：

1. 执行_writer = 0.，表示释放写锁。

2. 根据_nr_writers_queue 判断有没有写锁，如果有则唤醒一个写锁，如果没有写锁等待者，则唤醒所有的读锁等待者。

对于解锁，如果当前是读锁：

1. 执行_nr_readers- -，表示读锁占有者少了一个。

2. 判断_nr_readers 是否等于 0，是的话则表示当前线程是最后一个读锁占有者，需要唤醒写锁等待者或读锁等待者

公众号:一口Linux

3. 根据_nr_writers_queue 判断是否存在写锁等待者，若有，则唤醒一个写锁等待线程

4. 如果没有写锁等待者，判断是否存在读锁等待者，若有，则唤醒全部的读锁等待者

读写锁很容易造成，读者饿死或者写者饿死。

也可以设计公平的读写锁。

代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <fcntl.h>

#define THREADCOUNT 100

static int count = 0;
static pthread_rwlock_t lock;

void* Read(void* i)
{
  while(1)
  {
    pthread_rwlock_rdlock(&lock);
    printf("i am 读线程 : %d, 现在的 count 是%d\n",
(int)syscall(SYS_gettid), count);
    pthread_rwlock_unlock(&lock);
    //sleep(1);
  }
}

void* Write(void* i)
{
  while(1)
  {
    pthread_rwlock_wrlock(&lock);
    ++count;
    printf("i am 写线程 : %d, 现在的 count 是: %d\n",
(int)syscall(SYS_gettid), count);
    pthread_rwlock_unlock(&lock);
```

```
    sleep(1);
  }
}


int main()
{
  //close(1);
  //int fd = open("./dup2_result.txt", O_CREAT | O_RDWR);
  //dup2(fd, 1);
  pthread_t tid[THREADCOUNT];
  pthread_rwlock_init(&lock, NULL);
  for(int i = 0; i < THREADCOUNT; ++i)
  {
    if(i % 2 == 0)
    {
      pthread_create(&tid[i], NULL, Read, (void*)&i);
    }
    else
    {
      pthread_create(&tid[i], NULL, Write, (void*)&i);
    }
  }

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

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

上述代码很容易触发线程饿死。
读饿死或者写饿死。

公众号:一口Linux

本文档版权属于公众号:一口 Linux 号主所有，转载传播请联系作者 vx:yikoupeng

7.线程间同步

7.1 为什么需要线程同步?

线程同步是为了对临界资源访问的合理性。

例如：

就像工厂里生产车间没有原料了， 所有生产车间都停工了， 工人们都在车间睡觉。 突然进来一批原料， 如果原料充足， 你会发广播给所有车间， 原料来了， 快来开工吧。 如果进来的原料很少， 只够一个车间开工的， 你可能只会通知一个车间开工。

7.2 如何做到线程间同步?

条件等待是线程间同步的另一种方法。

如果条件不满足， 它能做的事情就是等待， 等到条件满足为止。 通常条件的达成， 很可能取决于另一个线程， 比如生产者-消费者模型。 当另外一个线程发现条件符合的时候， 它会选择一个时机去通知等待在这个条件上的线程。有两种可能性， 一种是唤醒一个线程， 一种是广播， 唤醒其他线程。

则在这个情况下，需要做到：

1、线程在条件不满足的情况下， 主动让出互斥量， 让其他线程去折腾， 线程在此处等待， 等待条件的满足；

2、一旦条件满足， 线程就可以立刻被唤醒。

3、线程之所以可以安心等待， 依赖的是其他线程的协作， 它确信会有一个线程在发现条件满足以后， 将向它发送信号， 并且让出互斥量。

7.3 条件变量

本质上是 PCB 等待队列 + 等待接口 + 唤醒接口。

7.3.1 条件变量的初始化

静态初始化

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

动态初始化

```
pthread_cond_init(pthread_cond_t *cond,const pthread_condattr_t
*attr);
```

参数	说明
cond	条件变量，传递地址
attr	条件变量属性，一般传入NULL

7.3.2 条件变量的等待

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

为什么这两个接口中有互斥锁？

条件不会无缘无故地突然变得满足了， 必然会牵扯到共享数据的变化。 所以一定要有互斥锁来保护。 没有互斥锁， 就无法安全地获取和修改共享数据。

同步并没有保证互斥，而保证互斥是使用到了互斥锁。

```
pthread_mutex_lock(&m)
while(condition_is_false)
{
        pthread_mutex_unlock(&m);
        //解锁之后， 等待之前， 可能条件已经满足， 信号已经发出， 但
是该信号可能会被错过
        cond_wait(&cv);
        pthread_mutex_lock(&m);
}
```

上面的解锁和等待不是原子操作。 解锁以后， 调用 cond_wait 之前，如果已经有其他线程获取到了互斥量， 并且满足了条件， 同时发出了通知信号， 那么 cond_wait 将错过这个信号， 可能会导致线程永远处于阻塞状态。 所以解

锁加等待必须是一个原子性的操作， 以确保已经注册到事件的等待队列之前，不会有其他线程可以获得互斥量。

那先注册等待事件， 后释放锁不行吗？ 注意， 条件等待是个阻塞型的接口，不单单是注册在事件的等待队列上， 线程也会因此阻塞于此， 从而导致互斥量无法释放， 其他线程获取不到互斥量， 也就无法通过改变共享数据使等待的条件得到满足， 因此这就造成了死锁。

```
pthread_mutex_lock(&m);
while(condition_is_false)
        pthread_cond_wait(&v,&m);//此处会阻塞
/*如果代码运行到此处， 则表示我们等待的条件已经满足了，
*并且在此持有了互斥量
*/
/*在满足条件的情况下， 做你想做的事情。
*/
pthread_mutex_unlock(&m);
```

pthread_cond_wait 函数只能由拥有互斥量的线程来调用， 当该函数返回的时候， 系统会确保该线程再次持有互斥量， 所以这个接口容易给人一种误解，就是该线程一直在持有互斥量。 事实上并不是这样的。 这个接口向系统声明了我在 PCB 等待序列中之后， 就把互斥量给释放了。 这样其他线程就有机会持有互斥量，操作共享数据， 触发变化， 使线程等待的条件得到满足。

pthread_cond_wait 内部会进行解锁逻辑，则一定要先放到 PCB 等待序列中，再进行解锁。

```
while(condition_is_false)
        pthread_cond_wait(&v,&m);//此处会阻塞
if(condition_is_false)
        pthread_cond_wait(&v,&m);//此处会阻塞
```

唤醒以后， 再次检查条件是否满足， 是不是多此一举？

因为唤醒中存在虚假唤醒（spurious wakeup） ， 换言之，条件尚未满足，pthread_cond_wait 就返了。 在一些实现中， 即使没有其他线程向条件变量发送信号， 等待此条件变量的线程也有可能会醒来。

条件满足了发送信号， 但等到调用 pthread_cond_wait 的线程得到 CPU 资源时， 条件又再次不满足了。 好在无论是哪种情况， 醒来之后再次测试条件是否满足就可以解决虚假等待的问题。

pthread_cond_wait 内部实现逻辑：

1. 将调用 pthread_cond_wait 函数的执行流放入到 PCB 等待队列当中
2. 解锁
3. 等待被唤醒
4. 被唤醒之后:

1、从 PCB 等待队列中移除出来

2、抢占互斥锁

情况 1：拿到互斥锁，pthread_cond_wait 就返回了

情况 2：没有拿到互斥锁，阻塞在 pthread_cond_wait 内部抢锁的逻辑中

当阻塞在 pthread_cond_wait 函数抢锁逻辑中时，一旦执行流时间耗尽，意味着线程就被切换出来了，程序计数器就保存的是抢锁的指令，上下文信息保存的就是寄存器的值

当再次拥有 CPU 资源后，恢复抢锁逻辑

直到抢锁成功，pthread_cond_wait 函数才会返回

7.3.3 条件变量的唤醒

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

pthread_cond_signal 负责唤醒等待在条件变量上的一个线程。

pthread_cond_broadcast，就是广播唤醒等待在条件变量上的所有线程。

先发送信号，然后解锁互斥量，这个顺序是必须的嘛？

先通知条件变量、 后解锁互斥量， 效率会比先解锁、 后通知条件变量低。因为先通知后解锁， 执行 pthread_cond_wait 的线程可能在互斥量已然处于加锁状态的时候醒来， 发现互斥量仍然没有解锁， 就会再次休眠， 从而导致了多余的上下文切换。

7.3.4 条件变量的销毁

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

注意：

1、永远不要用一个条件变量对另一个条件变量赋值， 即 pthread_cond_t cond_b = cond_a 不合法， 这种行为是未定义的。

2、使用 PTHREAD_COND_INITIALIZE 静态初始化的条件变量， 不需要被销毁。

3、要调用 pthread_cond_destroy 销毁的条件变量可以调用 pthread_cond_init 重新进行初始化。

4、不要引用已经销毁的条件变量， 这种行为是未定义的。

例：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#define NUMBER 2

int g_bowl = 0;
pthread_mutex_t mutex;//定义互斥锁
pthread_cond_t cond1;//条件变量
pthread_cond_t cond2;//条件变量


void *WorkProduct(void *arg)
{
  int *p = (int*)arg;
  while(1)
  {
    pthread_mutex_lock(&mutex);
    while(*p > 0)
    {
      pthread_cond_wait(&cond2,&mutex);//条件等待，条件不满足，陷入阻
塞
    }
    ++(*p);
    printf("i am workproduct :%d,i
product %d\n",(int)syscall(SYS_gettid),*p);
    pthread_cond_signal(&cond1);//通知消费者
    pthread_mutex_unlock(&mutex);//释放锁
  }
  return NULL;
}
```

公众号:一口Linux

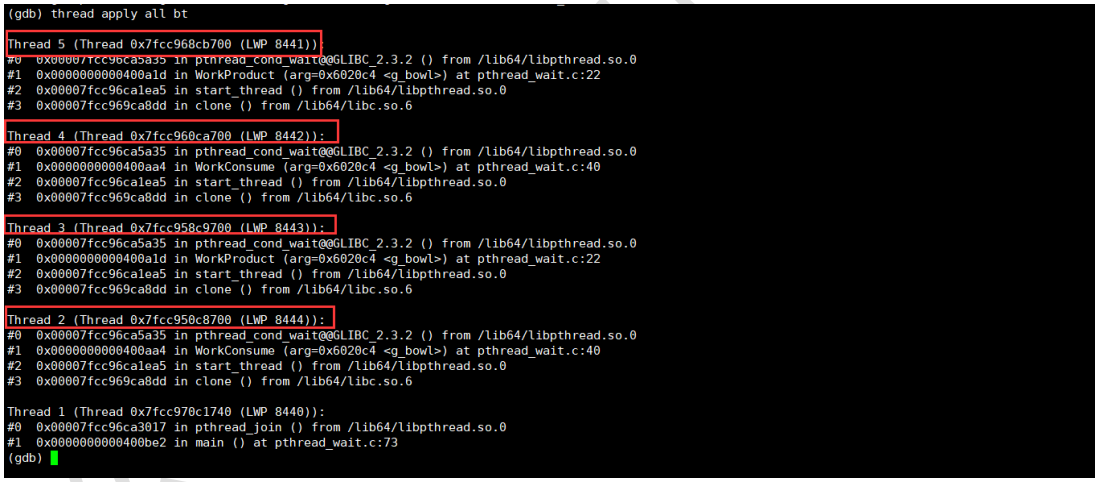
```
void *WorkConsume(void *arg)
{
  int *p = (int*)arg;
  while(1)
  {
    pthread_mutex_lock(&mutex);
    while(*p <= 0)
    {
      pthread_cond_wait(&cond1,&mutex);//条件等待，条件不满足，陷入阻
塞
    }
    printf("i am workconsume :%d,i
consume %d\n",(int)syscall(SYS_gettid),*p);
    --(*p);
    pthread_cond_signal(&cond2);//通知生产者
    pthread_mutex_unlock(&mutex);//释放锁
  }
  return NULL;
}
int main()
{
  pthread_t cons[NUMBER],prod[NUMBER];
  pthread_mutex_init(&mutex,NULL);//互斥锁初始化
  pthread_cond_init(&cond1,NULL);//条件变量初始化
  pthread_cond_init(&cond2,NULL);//条件变量初始化
  int i = 0;
  for(;i < NUMBER;++i)
  {
    int ret =
pthread_create(&prod[i],NULL,WorkProduct,(void*)&g_bowl);
    if(ret != 0)
    {
      perror("pthread_create");
      return -1;
    }
    ret = pthread_create(&cons[i],NULL,WorkConsume,(void*)&g_bowl);
    if(ret != 0)
    {
      perror("pthread_create");
      return -1;
    }
  }
  for(i = 0;i < NUMBER;++i)
  {
```

公众号:一口Linux

```
    pthread_join(cons[i],NULL);//线程等待
    pthread_join(prod[i],NULL);
  }
  pthread_mutex_destroy(&mutex);//销毁互斥锁
  pthread_cond_destroy(&cond1);
  pthread_cond_destroy(&cond2);
  while(1)
  {
    printf("i am main work thread\n");
    sleep(1);
  }
  return 0;
}
```

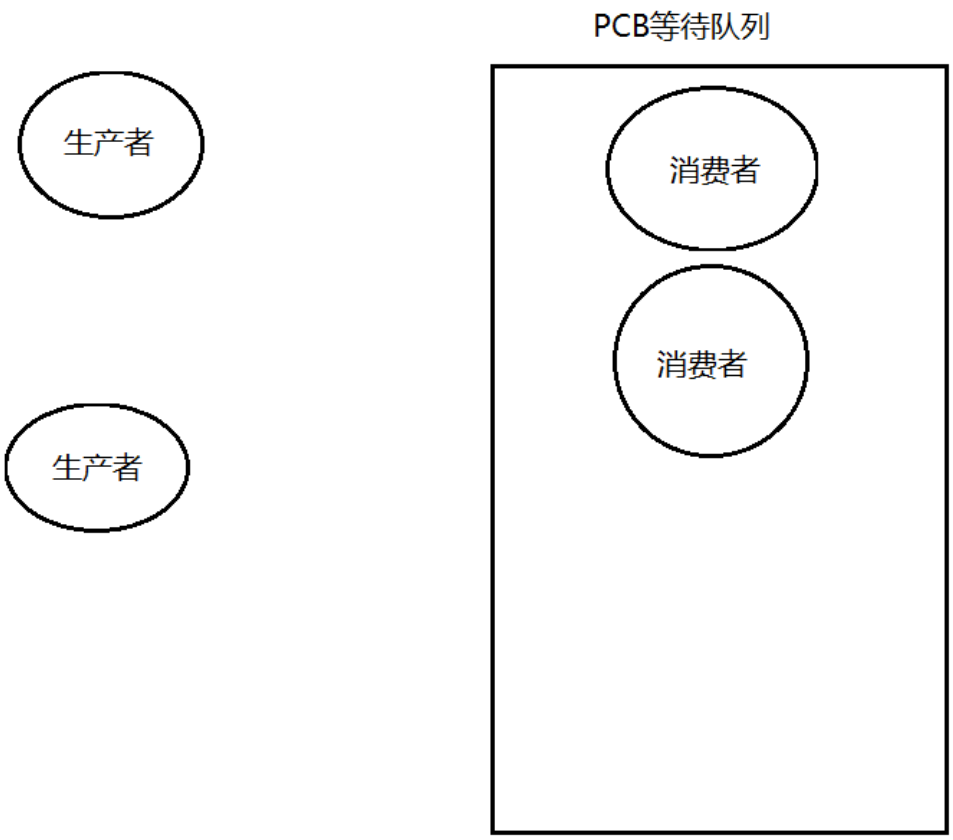
在这里为什么有两个条件变量呢？
若所有的线程只使用一个条件变量，会导致所有线程最后都进入 PCB 等待队列。

thread apply all bt 查看：



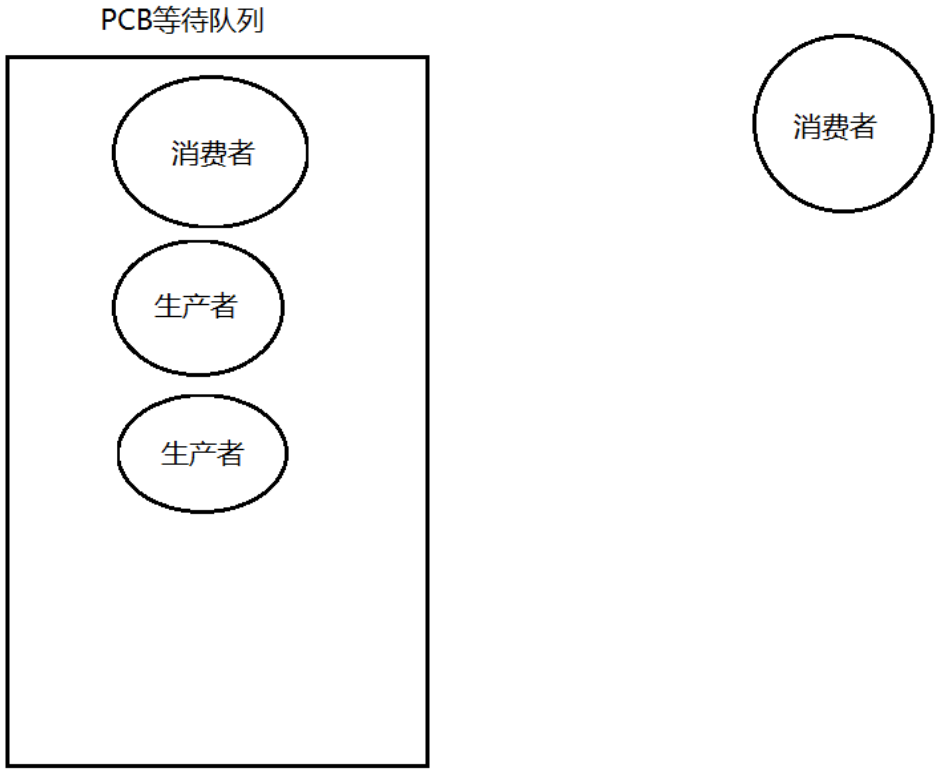
7.3.5 情况分析：两个生产者，两个消费者，一个 PCB 等待队列

1、最开始的情况，两个消费者抢到了锁，此时生产者未生产，则都放入 PCB 等待队列中

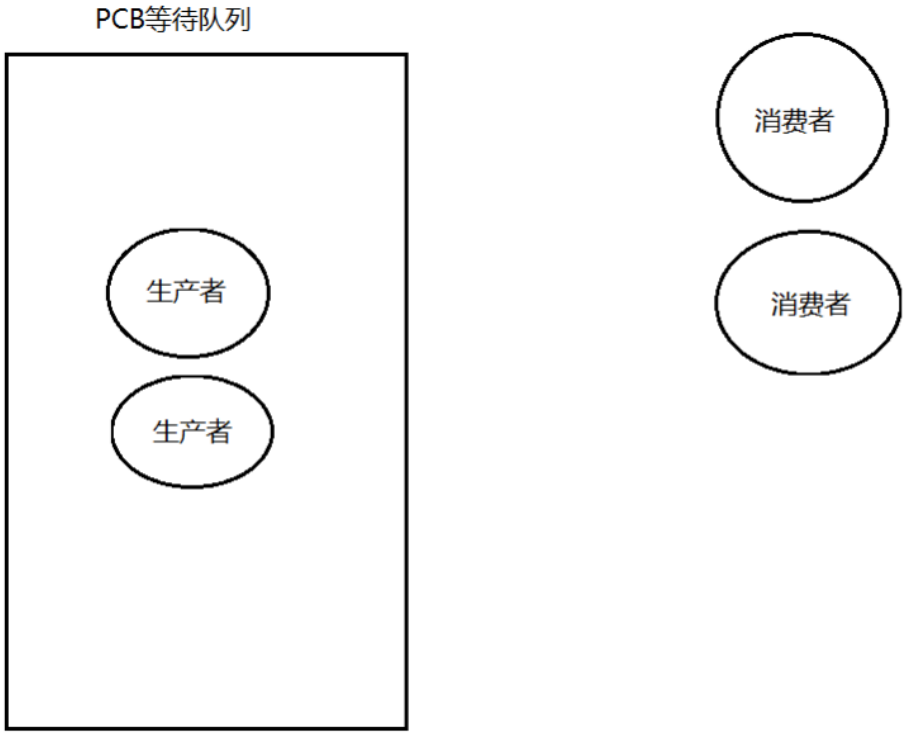


2、一个生产者抢到了锁，生产了一份材料，唤醒一个消费者，此时三者抢锁，若两个生产者分别先后抢到了锁，则都进入 PCB 等待队列中

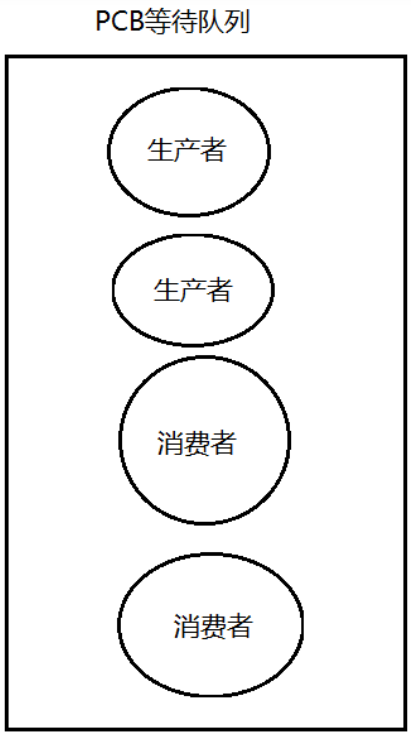
公众号：一



3、只有一个消费者，则必会抢到锁，消费材料，唤醒 PCB 等待队列，若此时唤醒的是，消费者，则现在是这样一个情况：



4、两个消费者在外边抢锁，一定都会进入 PCB 等待队列中



解决上述问题可采用两种方法：

1、使用 int pthread_cond_broadcast(pthread_cond_t *cond);，唤醒 PCB 等待队列中所有的线程。此时所有线程都会同时执行抢锁逻辑，太消费资源了。此方法不妥

2、采用两个 PCB 等待序列，一个放生产者，一个放消费者，生产者唤醒消费者，消费者唤醒生产者。

8.线程取消

8.1 线程取消函数接口

```
int pthread_cancel(pthread_t thread);
```

一个线程可以通过调用该函数向另一个线程发送取消请求。 这不是个阻塞型接口， 发出请求后， 函数就立刻返回了， 而不会等待目标线程退出之后才返回。

调用 pthread_cancel 时， 会向目标线程发送一个 SIGCANCEL 的信号， 该信号就是 kill -l 中消失的 32 号信号。

线程的默认取消状态是 PTHREAD_CANCEL_ENABLE。即是可被取消的。

取消类型	取消方式	说明
PTHREAD_CANCEL_DEFERRED	异步取消	线程可能在任何时间点（可能是立即取消，但也不一定）取消线程。
PTHREAD_CANCEL_ASYNCHRONOUS	延迟取消	程会一直执行，直到遇到一个取消点，这种方式也是新建线程的默认取消类型。

什么是取消点？ 可通过 man pthreads 查看取消点
就是对于某些函数， 如果线程允许取消且取消类型是延迟取消， 并且线程也收到了取消请求， 那么当执行到这些函数的时候， 线程就可以退出了。

8.2 线程取消带来的弊端

目标线程可能会持有互斥量、 信号量或其他类型的锁， 这时候如果收到取消请求， 并且取消类型是异步取消， 那么可能目标线程掌握的资源还没有来得及释放就被迫退出了， 这可能会给其他线程带来不可恢复的后果， 比如死锁（其他线程再也无法获得资源） 。

注意：

轻易不要调用 pthread_cancel 函数， 在外部杀死线程是很糟糕的做法，毕竟如果想通知目标线程退出， 还可以采取其他方法。

如果不得不允许线程取消， 那么在某些非常关键不容有失的代码区域， 暂时将线程设置成不可取消状态， 退出关键区域之后， 再恢复成可以取消的状态。

在非关键的区域， 也要将线程设置成延迟取消， 永远不要设置成异步取消。

8.2 线程清理函数

假设遇到取消请求， 线程执行到了取消点， 却没有来得及做清理动作（如动态申请的内存没有释放， 申请的互斥量没有解锁等） ， 可能会导致错误的产生， 比如死锁， 甚至是进程崩溃。

为了避免这种情况， 线程可以设置一个或多个清理函数， 线程取消或退出时，会自动执行这些清理函数， 以确保资源处于一致的状态。

如果线程被取消， 清理函数则会负责解锁操作。

```
void pthread_cleanup_push(void (*routine)(void *),void *arg);
```

```
void pthread_cleanup_pop(int execute);
```

这两个函数必须同时出现， 并且属于同一个语法块。

何时会触发注册的清理函数：？

1、当线程的主函数是调用 pthread_exit 返回的， 清理函数总是会被执行。

2、当线程是被其他线程调用 pthread_cancel 取消的， 清理函数总是会被执行。

3、当线程的主函数是通过 return 返回的， 并且 pthread_cleanup_pop 的唯一参数 execute 是 0 时， 清理函数不会被执行.

4、线程的主函数是通过 return 返回的， 并且 pthread_cleanup_pop 的唯一参数 execute 是非零值时， 清理函数会执行一次。

代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#define NUMBER 2
int g_bowl = 0;

pthread_mutex_t mutex;//定义互斥锁

void clean(void *arg)
{
  printf("Clean up:%s\n",(char*)arg);
  pthread_mutex_unlock(&mutex);//释放锁
}

void *WorkCancel(void *arg)
{
  pthread_mutex_lock(&mutex);
  pthread_cleanup_push(clean,"clean up handler");//清除函数的 push
  struct timespec t = {3,0};//取消点
  nanosleep(&t,0);
  pthread_cleanup_pop(0);//清除
  pthread_mutex_unlock(&mutex);
```

```
}

void *WorkWhile(void *arg)
{
  sleep(5);
  pthread_mutex_lock(&mutex);
  printf("i get the mutex\n");//若能拿到资源，则表示取消清理函数成
功!
  pthread_mutex_unlock(&mutex);
  return NULL;
}
int main()
{
  pthread_t cons,prod;
  pthread_mutex_init(&mutex,NULL);//互斥锁初始化
  int ret = pthread_create(&prod,NULL,WorkCancel,(void*)&g_bowl);//该
线程拿到锁，然后挂掉
  if(ret != 0)
  {
    perror("pthread_create");
    return -1;
  }
  int ret1 = pthread_create(&cons,NULL,WorkWhile,(void*)&ret);//测试
该线程是否可以拿到锁
  if(ret1 != 0)
  {
    perror("pthread_create");
    return -1;
  }

  pthread_cancel(prod);//取消该线程
  pthread_join(prod,NULL);//线程等待
  pthread_join(cons,NULL);//线程等待
  pthread_mutex_destroy(&mutex);//销毁互斥锁
  while(1)
  {
    sleep(1);
  }
  return 0;
}
```

结果：只要拿到锁，就表明线程清理函数成功了。

公众号:一口Linux

```
[test@localhost cancel]$ ./create
Clean up:clean up handler
i get the mutex
```

9.多线程与 fork()

永远不要在多线程程序里面调用 fork。

Linux 的 fork 函数， 会复制一个进程， 对于多线程程序而言， fork 函数复制的是用 fork 的那个线程， 而并不复制其他的线程。 fork 之后其他线程都不见了。 Linux 存在 forkall 语义的系统调用， 无法做到将多线程全部复制。

多线程程序在 fork 之前， 其他线程可能正持有互斥量处理临界区的代码。 fork 之后， 其他线程都不见了， 那么互斥量的值可能处于不可用的状态，也不会有其他线程来将互斥量解锁。

10.生产者与消费者模型

10.1 生产者与消费者模型的本质

本质上是一个线程安全的队列，和两种角色的线程（生产者和消费者）

存在三种关系：

1、生产者与生产者互斥

2、消费者与消费者互斥

3、生产者与消费者同步+互斥

10.2 为什么需要生产者与消费者模型?

生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生成完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列中取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费解耦的。

10.3 优点

1、解耦

2、支持高并发

3、支持忙闲不均

10.4 实现两个消费者线程，两个生产者线程的生产者消费者模型

生产者生成时用的同一个全局变量，故对该全局变量进行了加锁。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <queue>
#include <sys/syscall.h>

#define PTHREAD_COUNT 2
int data = 0;//全局变量作为插入数据
pthread_mutex_t mutex1;
class ModelOfConProd{
  public:
    ModelOfConProd()//构造
    {
      _capacity = 10;
      pthread_mutex_init(&_mutex,NULL);
      pthread_cond_init(&_cons,NULL);
      pthread_cond_init(&_prod,NULL);
    }
    ~ModelOfConProd()//析构
    {
      _capacity = 0;
      pthread_mutex_destroy(&_mutex);
      pthread_cond_destroy(&_cons);
      pthread_cond_destroy(&_prod);
    }
```

```
    void Push(int data)//push 数据，生产者线程使用的
    {
      pthread_mutex_lock(&_mutex);
      while((int)_queue.size() >= _capacity)
      {
        pthread_cond_wait(&_prod,&_mutex);
      }
      _queue.push(data);
      pthread_mutex_unlock(&_mutex);
      pthread_cond_signal(&_cons);
    }


    void Pop(int& data)//pop 数据，消费者线程使用的
    {
      pthread_mutex_lock(&_mutex);
      while(_queue.empty())
      {
        pthread_cond_wait(&_cons,&_mutex);
      }
      data = _queue.front();
      _queue.pop();
      pthread_mutex_unlock(&_mutex);
      pthread_cond_signal(&_prod);
    }

  private:
    int _capacity;//容量大小，限制容量大小
    std::queue<int> _queue;//队列
    pthread_mutex_t _mutex;//互斥锁
    pthread_cond_t _cons;//消费者条件变量
    pthread_cond_t _prod;//生产者条件变量
};

void *ConsumerStart(void *arg)//消费者入口函数
{
  ModelOfConProd *cp = (ModelOfConProd *)arg;
  while(1)
  {
    cp->Push(data);
    printf("i am pid : %d,i
push :%d\n",(int)syscall(SYS_gettid),data);
    pthread_mutex_lock(&mutex1);//++的时候，给该全局变量加锁
    ++data;
```

公众号：一口Linux

```
    pthread_mutex_unlock(&mutex1);
  }
}


void *ProductsStart(void *arg)//生产者入口函数
{
  ModelOfConProd *cp = (ModelOfConProd *)arg;
  int data = 0;
  while(1)
  {
    cp->Pop(data);
    printf("i am pid : %d,i
pop :%d\n",(int)syscall(SYS_gettid),data);
  }
}


int main()
{
  ModelOfConProd *cp = new ModelOfConProd;
  pthread_mutex_init(&mutex1,NULL);
  pthread_t cons[PTHREAD_COUNT],prod[PTHREAD_COUNT];
  for(int i = 0;i < PTHREAD_COUNT; ++i)
  {
    int ret = pthread_create(&cons[i],NULL,ConsumerStart,(void*)cp);
    if(ret < 0)
    {
      perror("pthread_create");
      return -1;
    }
    ret = pthread_create(&prod[i],NULL,ProductsStart,(void*)cp);
    if(ret < 0)
    {
      perror("pthread_create");
      return -1;
    }
  }


  for(int i = 0;i < PTHREAD_COUNT;++i)
  {
    pthread_join(cons[i],NULL);
    pthread_join(prod[i],NULL);
```

公众号:一口Linux

```
    }


    pthread_mutex_destroy(&mutex1);

    return 0;
}
```

11.写多线程时应注意

1. 先考虑代码的核心逻辑（先实现）
2. 考虑核心逻辑中是否访问临界资源或者说执行临界区代码，如果有就需要保持互斥
3. 考虑线程之间是否需要同步

公众号:一口Linux