

Unix系统高级编程

并发冲突和线程同步

Unit28

互斥锁



并发冲突



并发冲突

- 当多个线程同时访问其共享的进程资源时，如果不能相互协调配合，就难免会出现数据不一致或不完整的问题。这种现象被称为线程间的并发访问冲突，简称并发冲突

- 假设有整型全局变量g_cn被初始化为0

```
int g_cn = 0;
```

- 启动两个线程，同时执行如下线程过程函数，分别对该全局变量做一百万次累加

```
void* start_routine (void* arg) {  
    int i;  
    for (i = 0; i < 1000000; ++i) ++g_cn;  
    return NULL; }
```

- 两个线程结束后，g_cn的值理想情况下应该是两百万，但实际情况却往往少于两百万，且每次运行的结果不尽相同

并发冲突 (续1)

- 理想中的原子 “++”

线程1		内存	线程2	
指令	eax	g_cn	eax	指令
movl g_cn, %eax	0	0		
addl \$1, %eax	1	0		
movl %eax, g_cn	1	1		
		1	1	movl g_cn, %eax
		1	2	addl \$1, %eax
		②	2	movl %eax, g_cn

- 原子操作通常被认为是不可分割的，即在执行完该操作之前不会被任何其它任务或事件中断。理论上只有在单条指令中完成的操作才可被视为原子操作

并发冲突 (续2)

- 现实中的非原子 “++”

线程1		内存	线程2	
指令	eax	g_cn	eax	指令
movl g_cn, %eax	0	0		
		0	0	movl g_cn, %eax
addl \$1, %eax	1	0		
		0	1	addl \$1, %eax
movl %eax, g_cn	1	1		
		①	1	movl %eax, g_cn

- 一组非原子化的操作常常会因为线程切换而导致未定义的结果。这时就必须借助人力的方法，迫使其被原子化



并发冲突

【参见：conflict.c】

课堂练习

- 并发冲突



互斥锁



互斥锁

- 初始化互斥锁

```
#include <pthread_h>
```

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

成功返回0，失败返回错误码

- *mutex*: 互斥锁
- *attr*: 互斥锁属性，同步一个进程内的多个线程，取NULL

- 例如

- `pthread_mutex_init (&g_mutex, NULL);`
- `pthread_mutex_t g_mutex =`
`PTHREAD_MUTEX_INITIALIZER;`

互斥锁（续1）

- 销毁互斥锁

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

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

成功返回0，失败返回错误码

- *mutex*: 互斥锁

- 释放和互斥锁*mutex*有关的一切内核资源
- 例如
 - `pthread_mutex_destroy (&g_mutex);`



互斥锁 (续2)

- 加锁互斥锁

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

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

成功返回0，失败返回错误码

- *mutex*: 互斥锁

- 例如

- `pthread_mutex_lock (&g_mutex);`

- 任何时刻只会有一个线程对特定的互斥锁加锁成功并持有该互斥锁，其它试图对其加锁的线程只能在此函数的阻塞中等待，直到该互斥锁的持有者线程将其解锁为止



互斥锁 (续3)

- 解锁互斥锁

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

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

成功返回0，失败返回错误码

- *mutex*: 互斥锁

- 例如

- `pthread_mutex_unlock (&g_mutex);`

- 对特定互斥锁加锁成功的线程通过此函数将其解锁，那些阻塞于对该互斥锁加锁的线程中的一个会被唤醒，并得到该互斥锁，并从pthread_mutex_lock函数中返回



互斥锁 (续4)

- 例如

- for (i = 0; i < 1000000; ++i) {
 pthread_mutex_lock (&g_mutex);
 ++g_cn;
 pthread_mutex_unlock (&g_mutex);
}

- 任何时候都只会有一个线程执行++g_cn, 其它试图执行++g_cn的线程则阻塞于pthread_mutex_lock函数, 直到那个执行++g_cn的线程在完成计算后, 通过pthread_mutex_unlock函数解锁该互斥锁, 这时处于阻塞状态的线程之一将被唤醒, 并从pthread_mutex_lock函数中返回, 执行++g_cn, 其它线程继续在阻塞中等待



基于互斥锁的线程同步

【参见：mutex.c】

课堂练习

- 基于互斥锁的线程同步

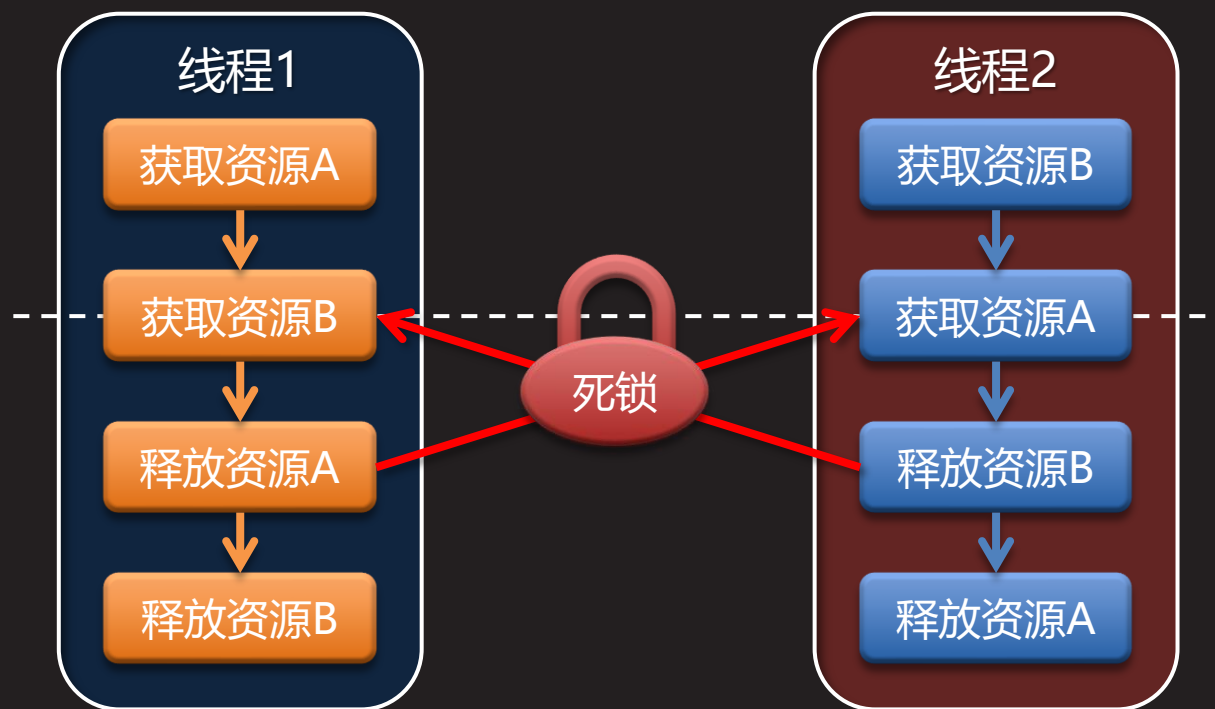


死锁问题



死锁问题

- 死锁系两个或两个以上的线程在运行过程中为了争夺资源，而形成的一种互相等待现象。若无外力作用，它们都将无法进行下去。此时称系统处于死锁状态或系统产生了死锁。这些处于互等待状态的线程则称为死锁线程



死锁问题

【参见：deadlock.c】

课堂
练习

- 死锁问题



死锁问题（续1）

- 产生死锁的四个必要条件

- 独占排它

- 线程以独占的方式使用其所获得的资源，即在一段时间内不允许其它线程使用该资源。这段时间内，任何试图请求该资源的线程只能在阻塞中等待，直到资源被其拥有者主动释放

- 请求保持

- 线程已经拥有了至少一个资源，但又试图获取已被其它线程拥有的资源，因此只能在阻塞中等待，同时对自己已经获得的资源坚守不放

- 不可剥夺

- 线程已经获得的资源，在其未被使用完之前，不可被强制剥夺，而只能由其拥有者自己释放

- 循环等待

- 线程集合 $\{T_0, T_1, T_2, \dots, T_n\}$ 中， T_0 等待 T_1 占有的资源， T_1 等待 T_2 占有的资源，……， T_n 等待 T_0 占有的资源，形成环路



条件变量



生产者消费者问题



生产者消费者问题

- 生产者消费者(Producer-Consumer)问题, 亦称有界缓冲区(Bounded-Buffer)问题
- 两个线程共享一个公共的固定大小的缓冲区, 其中一个线程作为生产者, 负责将消息放入缓冲区; 而另一个线程则作为消费者, 负责从缓冲区中提取消息
- 假设缓冲区已满, 若生产者线程还想放入消息, 就必须等待消费者线程从缓冲区中提取消息以产生足够的空间
- 假设缓冲区已空, 若消费者线程还想提取消息, 就必须等待生产者线程向缓冲区中放入消息以产生足够的数据
- 生产者和消费者线程之间必须建立某种形式的同步, 以确保为其所共享的缓冲区既不发生上溢, 也不发生下溢



条件变量



条件变量

- 初始化条件变量

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

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

成功返回0，失败返回错误码

- *cond*: 条件变量
- *attr*: 条件变量属性，同步一个进程内的多个线程，取NULL
- 例如
 - `pthread_cond_init (&g_cond, NULL);`
 - `pthread_cond_t g_cond = PTHREAD_COND_INITIALIZER;`



条件变量 (续1)

- 销毁条件变量

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

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

成功返回0，失败返回错误码

- *cond*: 条件变量

- 释放和条件变量cond有关的一切内核资源
- 例如
 - `pthread_cond_destroy (&g_cond);`



条件变量 (续2)

- 等待条件变量

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

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

成功返回0，失败返回错误码

- *cond*: 条件变量
- *mutex*: 互斥体
- *abs_timeout*: 等候时限(始自UTC19700101T000000)



条件变量（续3）

- 以上函数会令调用线程进入阻塞状态，直到条件变量 **cond** 收到信号为止，阻塞期间互斥体 **mutex** 被解锁
- 条件变量必须与互斥体配合使用，以防止多个线程同时进入条件等待队列时发生竞争
 - 线程在调用 `pthread_cond_wait` 函数前必须先通过 `pthread_mutex_lock` 函数锁定 **mutex** 互斥体
 - 在调用线程进入条件等待队列之前，**mutex** 互斥体一直处于锁定状态，直到调用线程进入条件等待队列后才被解锁
 - 当调用线程即将从 `pthread_cond_wait` 函数返回时，**mutex** 互斥体会被重新锁定，回到调用该函数之前的状态



条件变量 (续4)

- 和pthread_cond_wait相比pthread_cond_timedwait函数多了一个*abs_timeout*参数, 在条件变量*cond*收到信号之前, 如果等候时限已到, 该函数会提前解除阻塞, 并返回ETIMEDOUT错误码
- 例如
 - pthread_mutex_lock (&g_mutex);
 - ...
 - if (条件满足)
 - pthread_cond_wait (&g_cond, &g_mutex);
 - ...
 - pthread_mutex_unlock (&g_mutex);



条件变量 (续5)

- 向指定的条件变量发送信号

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

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

成功返回0，失败返回错误码

- *cond*: 条件变量
- 通过pthread_cond_signal函数向条件变量*cond*发送信号，与该条件变量相对应的条件等待队列中的第一个线程，将离开条件等待队列，并在重新锁定*mutex*互斥体后，从pthread_cond_wait函数中返回



带条件变量的生产者消费者模型

【参见：cond.c】

- 带条件变量的生产者消费者模型



条件变量（续6）

- 通过pthread_cond_broadcast函数向条件变量`cond`发送信号，与该条件变量相对应的条件等待队列中的所有线程，将同时离开条件等待队列，但只有第一个重新锁定`mutex`互斥体的线程，能够从pthread_cond_wait函数中返回，其余线程则继续为等待`mutex`互斥体而阻塞
- 例如
 - pthread_cond_signal (&g_cond);
 - pthread_cond_broadcast (&g_cond);



条件变量 (续7)

- 被pthread_cond_broadcast函数唤醒的线程，在从pthread_cond_wait函数返回后，并不能确定自己一定是第一个获得*mutex*互斥体的线程。导致其进入等待状态的条件很可能因先于该线程获得*mutex*互斥体的线程动作而重新得到满足。因此有必要对该条件再做一次判断，以决定是向下执行还是继续等待

- pthread_mutex_lock (&g_mutex);

...

while (条件满足)

pthread_cond_wait (&g_cond, &g_mutex);

...

pthread_mutex_unlock (&g_mutex);



带条件变量的生产者消费者模型

【参见： `broadcast.c`】

- 带条件变量的生产者消费者模型



总结和答疑

