UC Day18

预习课

预习内容

并发冲突

并发冲突

并发冲突

知识讲解

- 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，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，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

- 事例代码

```
int g_cn = 0;

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


并发冲突

- 理想的原子操作

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

并发冲突

- 实际的原子操作

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

直播课见

UC

C/C++教学体系

目录

线程同步

POSIX线程

- 缺省情况下，一个进程中的线程是以异步方式运行的，即各自运行各自的，彼此间不需要保持步调的协调一致
- 某些情况下，需要在线程之间建立起某种停等机制，即一或多个线程有时必须停下来，等待另外一或多个线程执行完一个特定的步骤以后才能继续执行，这就叫同步!

知识讲解

POSIX线程

知识讲解

- 并发冲突问题
  - 任何时候只允许一个线程持有共享数据，其它线程必须阻塞于调度队列之外，直到数据持有者不再持有该数据为止
- 资源竞争问题
  - 任何时候只允许部分线程拥有有限的资源，其它线程必须阻塞于调度队列之外，直到资源拥有者主动释放其所拥有的资源为止
- 条件等待问题
  - 当某些条件一时无法满足时，一些线程必须阻塞于调度队列之外，直到令该条件满足的线程用信号唤醒它们为止

互斥锁

互斥锁

知识讲解

- 线程间可以通过互斥锁解决资源竞争的问题。
- 任何时候都只能一个线程持有互斥锁，即加锁成功，在其持有该互斥锁的过程中，其它线程对该锁的加锁动作都会引发阻塞，只有当持有互斥锁的线程主动解锁，那些在加锁动作上阻塞的线程中的一个采用恢复运行并加锁成功。
- pthread_mutex_t 表示互斥锁数据类型

互斥锁

知识讲解

- #include <pthread_h>
- int pthread_mutex_init(pthread_mutex_t* mutex, pthread_mutexattr_t const* attr);
  - 功能：初始化互斥体
  - 参数：mutex：互斥体
    attr：互斥体属性
  - 返回值：成功返回0，失败返回错误码。
  - 也可以静态方式初始化互斥锁：
  - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

互斥锁

知识讲解

- #include <pthread_h>
  - int pthread_mutex_destroy(pthread_mutex_t* mutex);
  - 功能：销毁互斥体
  - 参数：mutex：互斥体
  - 返回值：成功返回0，失败返回错误码

互斥锁

- #include <pthread.h>
- int pthread_mutex_lock (pthread_mutex_t* mutex);
  - 功能：锁定互斥体
  - 参数：mutex 互斥体
  - 成功：返回0，失败返回错误码

知识讲解

互斥锁

- #include <pthread.h>
- int pthread_mutex_unlock (pthread_mutex_t* mutex);
  - 功能：解锁互斥锁
  - 参数：mutex 互斥锁
  - 成功：返回0，失败返回错误码

知识讲解

条件变量

条件变量

知识讲解

- 一个线程在某种条件不满足的情况下，无法进行后续工作，这时它就可以睡入某个条件变量，这时会有其它线程为其创建条件，一旦条件满足可以唤醒那些在相应条件变量中睡眠的线程继续运行。
- 通过pthread_cond_t 类型来表示条件变量

条件变量

知识讲解

- 初始化条件变量
  - #include <pthread.h>
  - int pthread_cond_init (pthread_cond_t* cond,const pthread_condattr_t* attr);
  - 功能：初始化条件变量
  - 参数：cond 条件变量
    attr 条件变量属性
  - 返回值：成功返回0，失败返回错误码
  - 也可以静态方式初始化条件变量
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

条件变量

知识讲解

- 销毁条件变量
  - #include <pthread.h>
  - int pthread_cond_destroy(pthread_cond_t* cond)
  - 功能：销毁条件变量
  - 参数：cond 条件变量
  - 返回值：成功返回0，失败返回错误码

条件变量

知识讲解

- 睡入条件变量
  - #include <pthread.h>
  - int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);
  - 功能：睡入条件变量
  - 参数：cond 条件变量
    mutex 互斥锁
  - 返回值：成功返回0，失败返回错误码

条件变量

知识讲解

- pthread_cond_wait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

知识讲解

- 唤醒条件变量
  - #include <pthread.h>
  - int pthread_cond_signal(pthread_cond_t* cond);
  - 功能：唤醒在条件变量中睡眠的一个线程
  - 参数：cond 条件变量
  - 返回值：成功返回0，失败返回错误码

条件变量

- 一个线程调用pthread_cond_wait函数进入阻塞状态，直到条件变量cond收到信号为止，阻塞期间互斥锁mutex会被释放。另一个线程通过pthread_cond_signal函数向条件变量cond发送信号，唤醒在其中睡眠的一个线程，该线程即从pthread_cond_wait函数中返回，同时重新获得互斥锁mutex

知识讲解

生产者消费者问题

知识讲解

- 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

生产者消费者问题

- 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

知识讲解

谢谢

UC Day18

复习课

线程ID

线程ID

- pthread_t类型的线程ID是POSIX线程库内部维护的线程的唯一标识，通常表现为一个很大的整数，跨平台，可移植。
  - #include <pthread.h>
  - pthread_t pthread_self(void);
  - 返回调用线程的(POSIX线程库的)TID

知识讲解

线程ID

- syscall(SYS_gettid)函数返回是一个long int类型整数，是系统内核产生线程唯一标识。一个进程的PID其实就是它的主线程的TID。
  - #include <sys/syscall.h>
  - long int syscall(SYS_gettid);
  - 返回调用线程的(系统内核的)TID

知识讲解

线程ID

知识讲解

- pthread_t类型在不同系统会被实现为不同的数据类型，甚至可能会使用结构体。因此判断两个线程ID是否相等或不相等，最好不要使用"=="或"!="运算符，因为这些关系运算符只能用于C语言内置的简单类型，而对于结构类型的数据不适用。
  - #include <pthread.h>
  - int pthread_equal(pthread_t t1, pthread_t t2);
  - 若两个参数所表示的TID相等返回非零，否则返回0

下节课见