线程并发访问

Andrew Huang <bluedrum@163.com>

课程内容:

- 可重入函数和线程安全函数
- Linux线程锁
- Linux 线程同步
- Windows多线程编程

可重入函数和线程安全函数

1. 可重入函数和线程安全函数

- 单线程程序只有一个控制流。不需要考虑一个资源（比如静态或全局变量如何处理）被同时访问或并发访问,但是多线程程序必须考虑并发访问一个资源。为了保证资源的完整性，为多线程程序写的代码必须是可重入的和线程安全的。
  - **线程安全的(Thread-Safe)**：如果一个函数在同一时刻可以被多个线程安全地调用，就称该函数是线程安全的。线程安全函数解决多个线程调用函数时访问共享资源的冲突问题。
  - **可重入(Reentrant)**：函数可以由多于一个线程并发使用，而不必担心数据错误。可重入函数可以在任意时刻被中断，稍后再继续运行，不会丢失数据。可重入性解决函数运行结果的确定性和可重复性。

可重入函数和线程安全关系

- 两者之间的关系：
  - 1、一个函数对于多个线程是可重入的，则这个函数是线程安全的。
  - 2、一个函数是线程安全的，但并不一定是可重入的。
  - 3、可重入性要强于线程安全性。
- 两者的侧重点不一样,不安全的原因主要在于使用全局共享的资源.
  - 静态变量,全局变量

2. 可重入函数

- 可重入的函数需要做到如下三点
  - 不在函数内部使用静态或全局数据，
  - 不返回静态或全局数据，所有数据都由函数的调用者提供。
  - 使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据。
  - 如果必须访问全局变量，利用加锁机制(如互斥量)来保护全局变量。
  - 不调用不可重入函数

函数可重入化

- **C标准库中很多函数是不可重入的.因为在函数内部使用静态数据**
  - **strtok()是非可重入的，因为它在内部存储了被标记分割的字符串；ctime()函数也是非可重入的，它返回一个指向静态数据的指针，而该静态数据在每次调用中都被覆盖重写。**
  - 为了解决可重入问题,很多C库提供可以重入版本,如strtok_r() 就是strtok的可重入版本

```
char *strtoupper(char *string)
{//不可重入版本
    static char buffer[MAX_STRING_SIZE];
    int index;

    for (index = 0; string[index]; index++)
        buffer[index] = toupper(string[index]);
    buffer[index] = 0
    return buffer;
}
```

3. 线程函数编写

- 在C语言中，局部变量是在栈上分配的。因此，任何未使用静态数据或其他共享资源的函数都是线程安全的
- 以下是线程安全,

```
int diff(int x, int y)
{
    int delta;

    delta = y - x;
    if (delta < 0)
        delta = -delta;

    return delta;
}
```

多线程并发控制

- 多线程编程的主要问题
  - 是对共享数据的保护.即在多个线程同时访问同一个数据时,保证数据读写安全.
  - 线程安全函数除了尽量不使用静态或全局变量,另一个主要手段是加锁
  - 全局共享的数据一般建议使用volatile关键字保护
    - **volatile int a[10];**
- pthread 线程一般通三种机制来完成并发控制
  - 线程互斥锁 (pthread mutex)
  - 线程条件变量(pthread cond)

- Posix匿名信号量

pthread 互斥锁

1. 线程互斥锁

- 一种在多线程程序中同步访问手段是使用互斥量。程序员给某个对象加上一把“锁”，每次只允许一个线程去访问它。如果想对代码关键部分的访问进行控制，你必须在进入这段代码之前锁定一把互斥量，在完成操作之后再打开它。
- 可以通过使用pthread的互斥接口保护数据，确保同一时间只有一个线程访问数据。互斥量从本质上说是一把锁，在访问共享资源前对互斥量进行加锁，在访问完成后释放互斥量上的锁。对互斥量进行加锁以后，任何其他试图再次对互斥量加锁的线程将会被阻塞直到当前线程释放该互斥锁。如果释放互斥锁时有多个线程阻塞，所以在该互斥锁上的阻塞线程都会变成可进行状态，第一个变成运行状态的线程可以对互斥量加锁，其他线程在次被阻塞，等待下次运行状态。
- mutex即为互斥的含义
- 只在同一个进程内的线程之间有效,无法跨进程使用.
- 相当于资源数只为1的信号量.
- 操作非常类似信号量操作,因此线程锁能做的事,可以用等效的信号量代码来操作.
- 线程锁机制同时也不是异步信号安全的，也就是说，不应该在信号处理过程中使用互斥锁，否则容易造成死锁。

2. 线程互斥锁的数据结构

- 互斥量用pthread_mutex_t数据类型来表示，在使用互斥量以前，必须首先对它进行初始化
  - 可以把它置为常量PTHREAD_MUTEX_INITIALIZER(只对静态分配的互斥量)
  - 也可以通过调用pthread_mutex_init函数进行初始化，如果动态地分配互斥量，那么释放内存前需要调用pthread_mutex_destroy.

3. 线程互斥锁使用

- 初始化一个互斥锁
  - **pthread_mutex_init()**等同于sem_init()
- 互斥锁加锁
  - **pthread_mutex_lock()**,等同于sem_wait()
- 互斥锁测试
  - **pthread_mutex_trylock()**,等于于sem_trywait()
- 互斥锁解锁
  - **pthread_mutex_unlock(),**等同于sem_post()
- 互斥锁销毁
  - **pthread_mutex_destory,**等同于sem_destroy()

初始化锁

名称::	pthread_mutex_init
功能：	初始化互斥锁。
头文件：	#include <pthread.h>
函数原形：	int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutex_t *attr);
参数：	mutex　互斥量 attr　　互斥锁属性
返回值：	若成功则返回 0，否则返回错误编号。

销毁锁

名称::	pthread_mutex_destroy
功能：	释放对互斥变量分配的资源
头文件：	#include <pthread.h>
函数原形：	int　pthread_mutex_destroy(pthread_mutex_t *mutex);
参数：	
返回值：	若成功则返回 0，否则返回错误编号。

上锁解锁

名称::	pthread_mutex_lock/　　pthread_mutex_trylock/ pthread_mutex_unlock
功能：	对互斥量加/减锁
头文件：	#include <pthread.h>

函数原形：	int pthread_mutex_lock(pthread_mutex_t *mutex);
	int pthread_mutex_trylock(pthread_mutex_t *mutex);int
	pthread_mutex_unlock(pthread_mutex_t *mutex);

参数：	
返回值：	若成功则返回 0，否则返回错误编号。

实例

```
char buffer[1024];
int buffer_has_item=0;

//
pthread_mutex_t mutex;
struct timespec delay;
```

```
/*接上一页*/
int   main ( void ){
      pthread_t reader;
      /*  创建一个全局互斥量,NULL 表示默认属性  ,用于保护 buffer*/
      pthread_mutex_init (&mutex,NULL);
      pthread_create(&reader, NULL, (void *)&reader_function, NULL);
      writer_function();
      return 0;
}
void writer_function (void){
      int i,cnt =0;
      while(1){
            pthread_mutex_lock (&mutex);
            if (buffer_has_item==0){
                  strcpy(buffer,"hello %d");
                  printf("write(%d) %s\n",cnt,buffer);

                  buffer_has_item=1;
                  cnt ++;
                  /* sleep(3);*/
            }
            pthread_mutex_unlock(&mutex);
            /*  线程延时一段时间  */
            /* pthread_delay_np(&delay); */
            for(i=0; i< 100;i++);
     }
}
void reader_function(void){
      int i,cnt=0;
      while(1){
            pthread_mutex_lock(&mutex);
            if(buffer_has_item==1){

                  printf("read(%d) %s\n",cnt,buffer);
                  buffer_has_item=0;
                  cnt ++;
            }
            /*  解锁*/
            pthread_mutex_unlock(&mutex);
            /* pthread_delay_np(&delay);*/
            for(i=0; i< 100;i++);
      }
}
```

4. 互斥锁

- 互斥量需要时间来加锁和解锁。锁住较少互斥量的程序通常运行得更快。所以，互斥量应该尽量少，够用即可，每个互斥量保护的区域应则尽量小。
- 互斥量的本质是串行执行。如果很多线程需要领繁地加锁同一个互斥量，则线程的大部分时间就会在等待，这对性能是有害的。如果互斥量保护的数据(或代码)包含彼此无关的片段，则可以特大的互斥量分解为几个小的互斥量来提高性能。这样，任意时刻需要小互斥量的线程减少，线程等待时间就会减少。所以，互斥量应该足够多(到有意义的地步)，每个互斥量保护的区域则应尽量的少。

posix匿名信号量

1. posix匿名信号量

- posix命名信号量。这些信号量由一个name参数标识，它通常指代文件系统中的某个文件。然而posix也提供基于内存的信号量，它们由应用程序分配信号量的内存空间，然后由系统初始化它们的值。
- 因为没有全局的名字，所以只能在同一进程中使用．多用于多线程的并发控制．

2. sem_init:初始化匿名信号量

名称:：	sem_init/sem_destroy
功能：	初始化/关闭信号等
头文件：	#include <semaphore.h>
函数原形：	int sem_init(sem_t *sem,int shared,unsigned int value); int sem_getvalue(sem_t *sem);
参数：	sem　指向信号量的指针 shared　作用范围 value　信号量初始值
返回值：	若成功则返回 0，否则返回-1。

- 基于内存的信号量是由sem_init初始化的。sem参数指向必须由应用程序分配的sem_t变量。如果shared为0，那么待初始化的信号量是在同一进程的各个线程共享的，否则该信号量是在进程间共享的。当shared为零时，该信号量必须存放在即将使用它的所有进程都能访问的某种类型的共享内存中。跟sem_open一样，value参数是该信号量的初始值。
- 使用完一个基于内存的信号量后，我们调用sem_destroy关闭它。
- 除了sem_open和sem_close外，其它的poisx有名信号量函数都可以用于基于内存的信号量。

3. 命名信号量与匿名信号量区别

- sem_open不需要类型与shared的参数，有名信号量总是可以在不同进程间共享的。
- sem_init不使用任何类似于O_CREAT标志的东西，也就是说，sem_init总是初始化信号量的值。因此，对于一个给定的信号量，我们必须小心保证只调用一次sem_init。

- sem_open返回一个指向某个sem_t变量的指针，该变量由函数本身分配并初始化。但sem_init的第一个参数是一个指向某个sem_t变量的指针，该变量由调用者分配，然后由sem_init函数初始化。
- **posix**有名信号量是通过内核持续的，一个进程创建一个信号量，另外的进程可以通过该信号量的外部名（创建信号量使用的文件名）来访问它。**posix**基于内存的信号量的持续性却是不定的，如果基于内存的信号量是由单个进程内的各个线程共享的，那么该信号量就是随进程持续的，当该进程终止时它也会消失。如果某个基于内存的信号量是在不同进程间同步的，该信号量必须存放在共享内存区中，这要只要该共享内存区存在，该信号量就存在。

多线程加锁

```
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <pthread.h>
#incude <stdlib.h>

void *thread_function(void *arg); /*线程入口函数*/
void print(void); /*共享资源函数*/

sem_t bin_sem; /*定义信号灯*/
int value; /*定义信号量的灯*/

int main()
{
int n=0;
pthread_t a_thread;

if((sem_init(&bin_sem,0,2))!=0) /*初始化信号灯，初始值为 2*/
{
    perror(“sem_init”);
    exit(1);
}
while(n++<5) /*循环创建 5 个线程*/
{
    if((pthread_create(&a_thread,NULL,thread_function,NULL))!=0)
{
    perror(“Thread creation failed”);
    exit(1);
}
}
pthread_join(a_thread,NULL);/*等待子线程返回*/
}
```

```
/*接上一页*/
void *thread_function(void *arg)
{
sem_wait(&bin_sem); /*等待信号灯*/
print();
sleep(1);
sem_post(&bin_sem); /*挂出信号灯*/
printf("I finished,my pid is %d\n",pthread_self());
pthread_exit(arg);
}

void print()
{
printf("I get it,my tid is %d\n",pthread_self());
sem_getvalue(&bin_sem,&value); /*获取信号灯的值*/
printf("Now the value have %d\n",value);
}
```

pthread 线程同步

1. 线程同步

- **同步就是线程等待某个事件的发生。只有当等待的事件发生线程才继续执行，否则线程挂起并放弃处理器。当多个线程协作时，相互作用的任务必须在一定的条件下同步。**
- **Linux下的C语言多线程编程有多种线程同步机制，最典型的是条件变量(condition variable)。**
- **条件变量所做的事情也能被信号量所代替,但条件变量本身不是原子操作,所以任何对条件变量的操作都要用一个线程锁来保护.**
  - **这也是大部代码看同步出现线程锁和线程条件变量的例子**

2. 条件变量概念

- 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。
- 条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。
- 条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

3. 线程条件变量

- 创建条件变量
  - **int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)**
- 销毁条件变量
  - **int pthread_cond_destroy(pthread_cond_t *cond)**
- 等待
  - **int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)**
- 激活
  - **pthread_cond_signal**

条件变量初始化

名称：	pthread_cond_init
目标：	条件变量初始化
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
参数:	cptr 条件变量 attr 条件变量属性
返回值：	成功返回 0，出错返回错误编号。

条件变量销毁

名称：	pthread_cond_destroy
目标：	条件变量摧毁
头文件：	#include < pthread.h>
函数原形:	int pthread_cond_destroy(pthread_cond_t *cond);
参数:	cptr 条件变量
返回值：	成功返回 0，出错返回错误编号。

条件变量等待

名称：	pthread_cond_wait/pthread_cond_timedwait
目标：	条件变量等待
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex); int pthread_cond_timedwait(pthread_cond_t *cond,pthread_mutex_t mytex, const struct timespec *abstime);
参数:	cond 条件变量 mutex 互斥锁
返回值：	成功返回 0，出错返回错误编号。

- 第一个参数***cond**是指向一个条件变量的指针。第二个参数***mutex** 则是对相关的互斥

锁的指针。函数**pthread_cond_timedwait** 函数类型与函数**pthread_cond_wait** ，区别在于，如果达到或是超过所引用的参数 ***abstime**，它将结束并返回错误**ETIME**．**pthread_cond_timedwait** 函数的参数***abstime**指向一个**timespec** 结构。

条件变量激活

名称：	pthread_cond_signal/pthread_cond_broadcast
目标：	条件变量通知
头文件：	#include < pthread.h>
函数原形：	int pthread_cond_signal(pthread_cond_t *cond); int pthread_cond_broadcast(pthread_cond_t *cond);
参数:	cond 条件变量
返回值：	成功返回 0，出错返回错误编号。

- 参数*cond是对类型为pthread_cond_t的一个条件变量的指针。当调用pthread_cond_signal时一个在相同条件变量上阻塞的线程将被解锁。如果同时有多个线程阻塞，则由调度策略确定接收通知的线程。如果调用pthread_cond_broadcast,则将通知阻塞在这个条件变量上的所有线程。一旦被唤醒，线程仍然会要求互斥锁。如果当前没有线程等待通知，则上面两种调用实际上成为一个空操作。如果参数*cond指向非法地址，则返回值EINVAL。

使用框架

```
/* 初始化条件变量及保护锁*/
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;/*初始化互斥锁*/
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;/*初始化条件变量*/

/*等待通知 */
   pthread_mutex_lock(&mutex);
   pthread_cond_wait(&cond,&mutex);
  pthread_mutex_unlock(&mutex);

/* 发送通知*/
 pthread_mutex_lock(&mutex);/*锁住互斥量*/
    pthread_cond_signal(&cond);/*条件改变，发送信号，通知 t_b 进程*/
   pthread_mutex_unlock(&mutex);/*解锁互斥量*/

/* 销毁条件变量及保护锁*/
pthread_mutex_destroy(&mutex);
 pthread_cond_destroy(&cond);
```

澡堂模型

- 参考代码 bathhouse.c

生产者—消费者模型

生产者－消费者模型

- 生产者－消费者模型是指：
  - 1．生产者进行生产将物品放入仓库，同一时间只能有一个生产者将物品放入仓库，如果仓库满，生产者等待。
  - 2．消费者从仓库中取出物品，同一时间只能有一个消费者取出物品，如果仓库空，消费者等待；如果仓库满,则生产者等待.
  - 3．生产者将物品放入仓库时消费者不能同时取；
  - 4．消费者取物品时生产者不能放入物品；
- 总之，就是生产者群体或消费者群体内部是互斥的，两个群体之间是同步的。

程序实例

- 生产者-消费者问题是一个著名的进程同步问题。具体描述：
- 有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。
- 为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有10个缓冲区的缓冲池，生产者进程将它所生产的产品放入一个缓冲区中；
- 消费者进程可从一个缓冲区中取走产品去消费。
- **尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品**

```
void producer(void)
{
        int m;
        while(i<10)
        {
                pthread_mutex_lock(&mutex);
                buffer[i]=i+1;
                i=i+1;
                printf("\nthere are 10 blocks in the buffer:0,1,2,3,4,5,6,7,8,9\n");
                printf("\nbuffer:");
                for(m=0;m<10;m++)
                        printf("%3d",buffer[m]);
                printf("\nthe number added by the producer is:");
                printf("%d", buffer[i]);
                printf("\npointer is %d",i);
                sem_post(&full);
                pthread_mutex_unlock(&mutex);
        }
}
```

```
/*接上一页*/
void consumer(void)
{
        int m;
        while(j<10)
        {
                int x;
                pthread_mutex_lock(&mutex);
                sem_wait(&full);
                if(buffer[j]==0) break;
                x=buffer[j];
                buffer[j]=0;
                j=j+1;
                printf("\nbuffer:");
                for(m=0;m<10;m++)
                        printf("%3d",buffer[m]);
                printf("\n %d is removed form the buffer by consumer",x);
                printf("\nThe present pointer is %d",j);
                pthread_mutex_unlock(&mutex);
                sem_post(&empty);
        }
}
```

Windows 线程并发控制

1. Windows下的互斥同步机制(可选)

- **临界区**（Critical Section）：临界区不是内核对象，在用户态实现了同一进程中线程的互斥。由于使用时不需要从用户态切换到核心态，所以速度很快（X86系统上约为20个指令周期），但其缺点是不能跨进程同步，同时不能指定阻塞时的等待时间，只能无限等待。
  - 类似于线程**互斥锁,速度最快**
- **互斥体**：（Mutex）：互斥体实现了和临界区类似的互斥功能，但区别在于：互斥体是内核对象，可以实现跨进程互斥，但需要在用户态和核心态之间切换，速度比临界区慢得多（X86系统上约为600个指令周期），同时可以指定阻塞时的等待时间。
  - 类似于Posix有名信号量
- **事件**（Event）：事件也是内核对象，具有“信号态”和“无信号态”两种状态。当某一线程等待一个事件时，如果事件为信号态，将继续执行，如果事件为无信号态，那么线程被阻塞。线程能够指定阻塞时的等待时间。
  - 类似于pthread_cond
- **信号量**（Semaphore）：信号量是一个资源计数器，当某线程获取某信号量时，信号量计数首先减1，如果计数小于0，那么该线程被阻塞；当某县城释放某信号量时，信号量

计数首先加1，如果计数小于或等与0，那么唤醒某被阻塞的线程并执行之。对信号量的总结如下：

- 1．如果计数器m大于0，表示还有m个资源可以访问，此时信号量线程等待队列中没有线程被阻塞，新的线程访问资源也不会被阻塞；
- 2．如果计数器m等与0，表示没有资源可以访问，此时信号量线程等待队列中没有线程被阻塞，但新的线程访问资源会被阻塞；
- 3．如果计数器m小于0，表示没有资源可以访问，此时信号量线程等待队列中有abs（m）个线程被阻塞，新的线程访问资源会被阻塞；

- 信号量常被用于保证对多个资源进行同步访问。

2. 互斥量的操作

- 互斥量类型 HANDLE
  - 等同**pthread_mutex_t**
- 创建一个互斥量 CreateMutex
  - **pthread_mutex_init**
- 关闭/销毁一个互斥量CloseHandle
  - **pthread_destroy**
- 加锁一个互斥量 WaitForSingleObject
  - **pthread_mutex_lock**
- 解锁一个互斥量ReleaseMutex
  - **pthread_mutex_unlock**

思考题

- 同样程序你可以用多线程和多进程模型来实现,你是如何权衡采用哪一种体系结构的?
- 在程序设计中,对公共资源(比如缓冲区等)的操作和访问经常需要使用锁来进行保护,但在大并发系统中过多的锁会导致效率很低,通常有那些方法可以尽量避免或减少锁的使用?

课堂练习

- 把自己写的链表库改成支持多线程的版本.
- 把澡堂模型的同步机制由条件变量改成用信号量来控制
- 将test_mutex例子用Windows来实现
  - 1.线程用_beginThread或CreateThread
  - 2.加锁用WaitSingleObject来lock,用ReleaseMutex来解锁
  - 具体对应表参见上页,调用请参见MSDN
  - 最好定义一组相同接口的宏来封装操作系统细节,