联航精英训练营

UNIGRESS ELITE TRAINING CAMP

Linux高级编程（六）

张勇涛

多线程编程

进程与线程

线程与进程

线程和进程比较有下面两个：

1. 它是一种非常"节俭"的多任务操作方式优点。

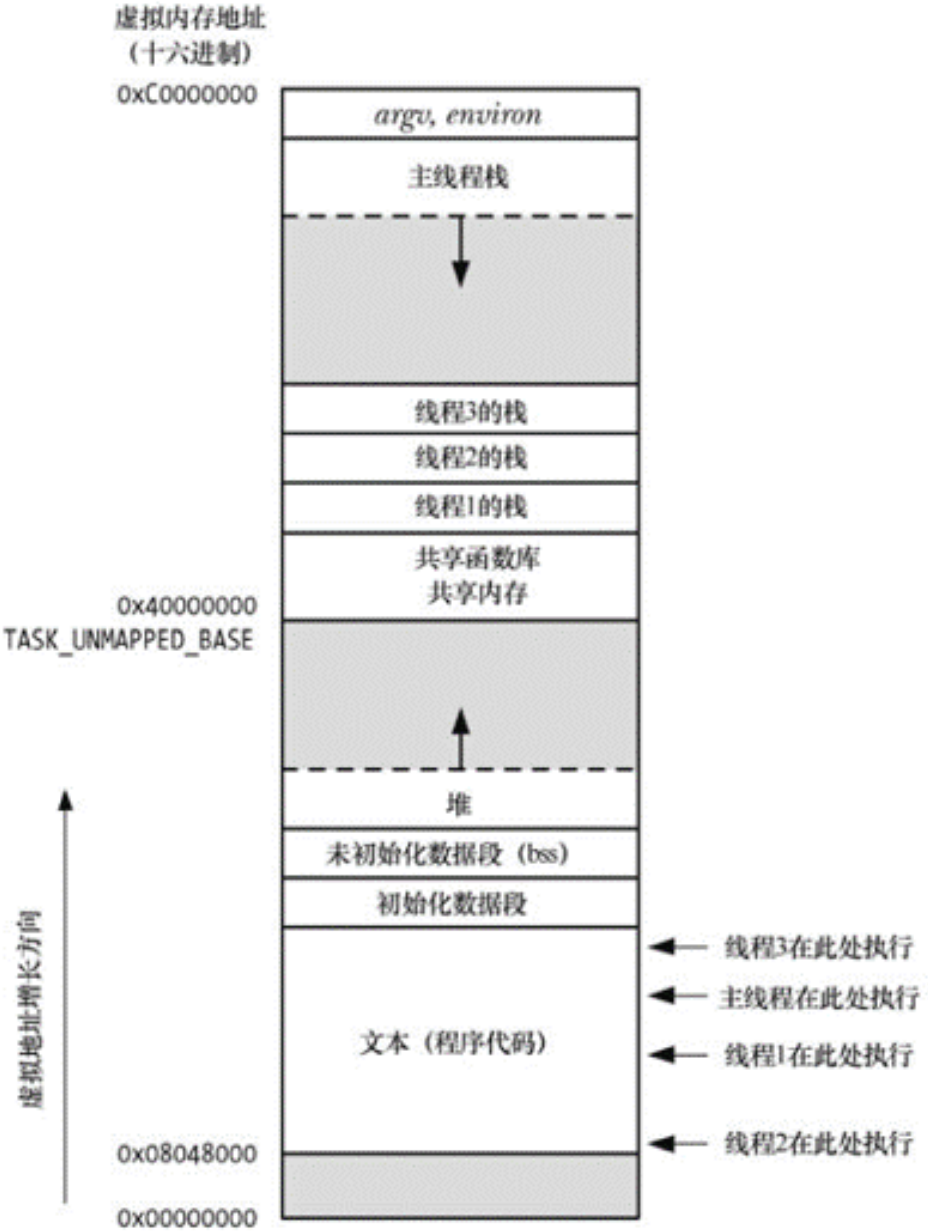
Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种“昂贵”的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。据统计，总的说来，一个进程的开销大约是一个线程开销的30倍左右。

2. 线程间方便的通信机制。

对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为static的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

线程与进程

线程执行开销小，而不利于资源的管理和保护，而进程则相反。



同时执行 4 个线程的进程（Linux/x86-32）

线程的其他优点

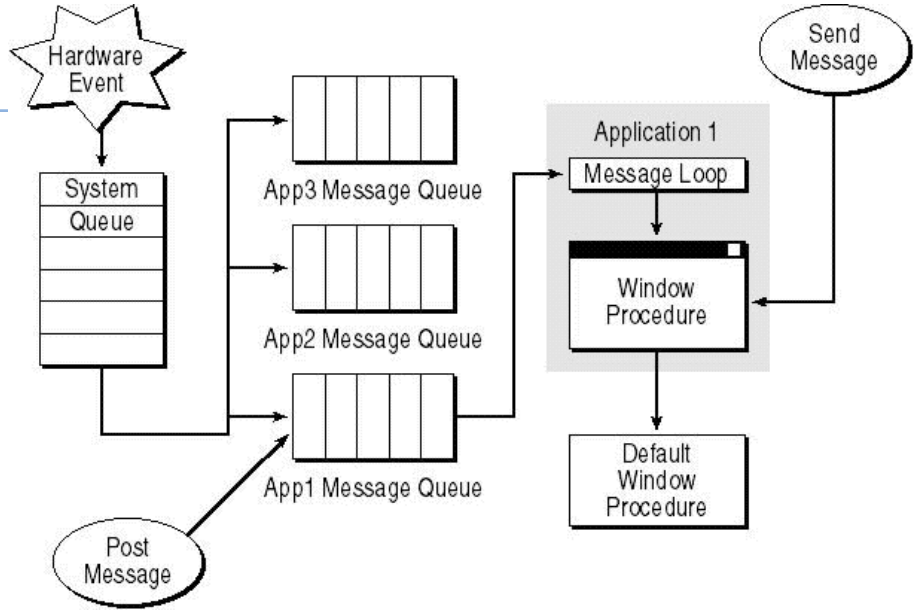
不和进程比较，多线程程序作为一种多任务、并发的工作方式，当然有以下的优点：

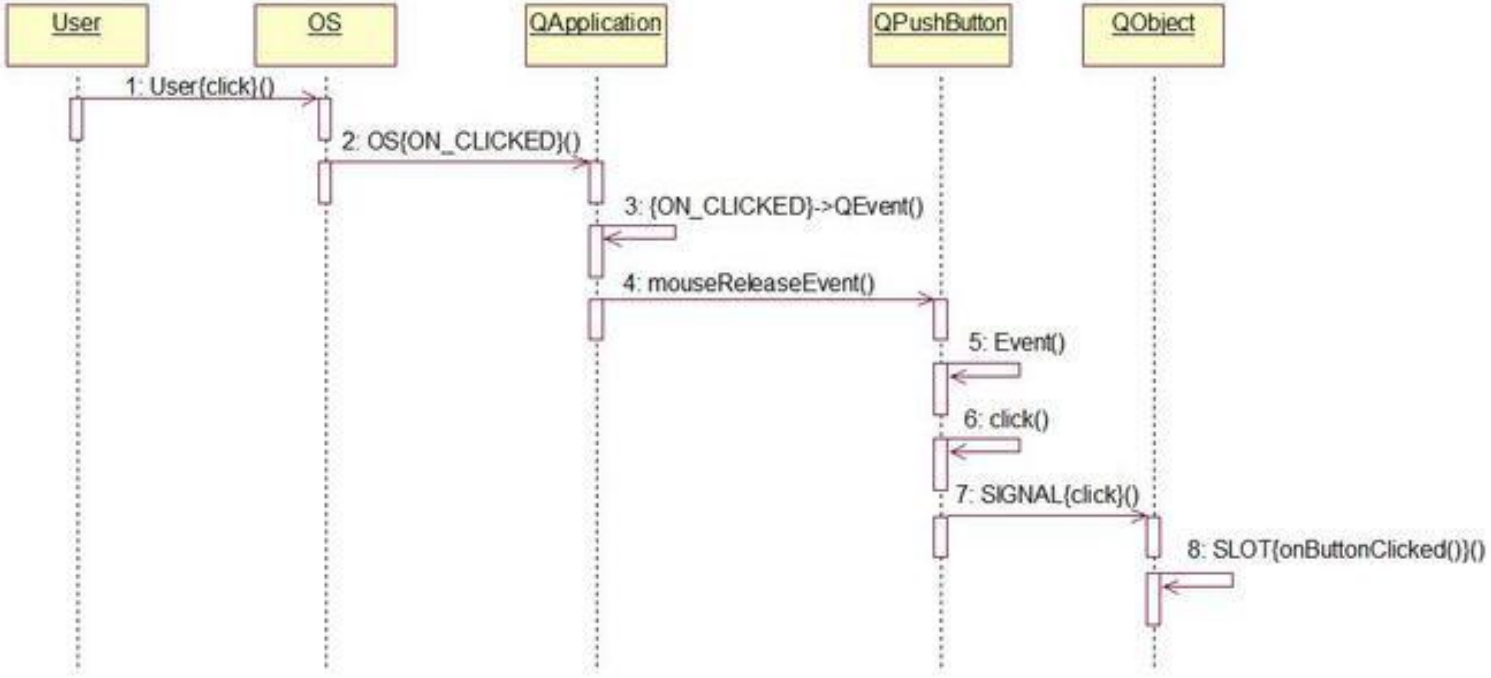
1) **提高应用程序响应。**这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。

2) **使多CPU系统更加有效。**操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

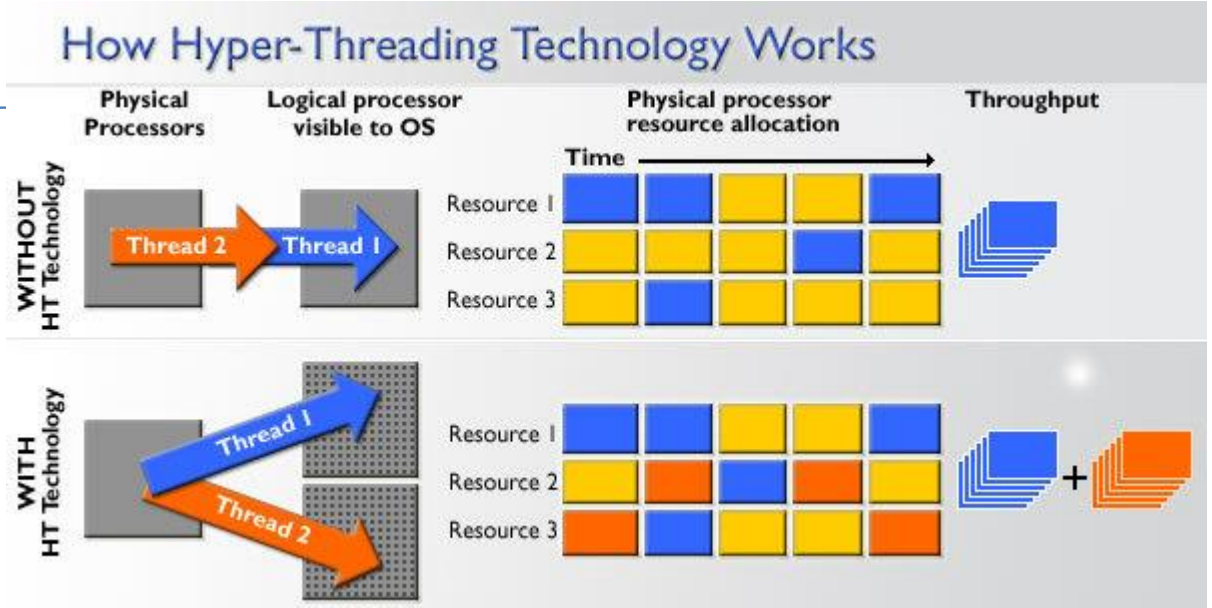
3) **改善程序结构。**一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

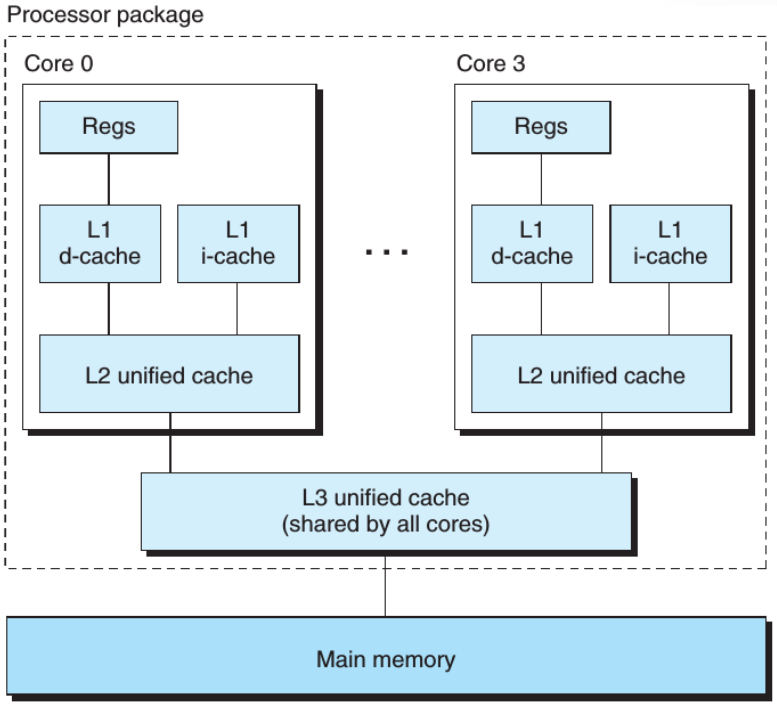
消息机制

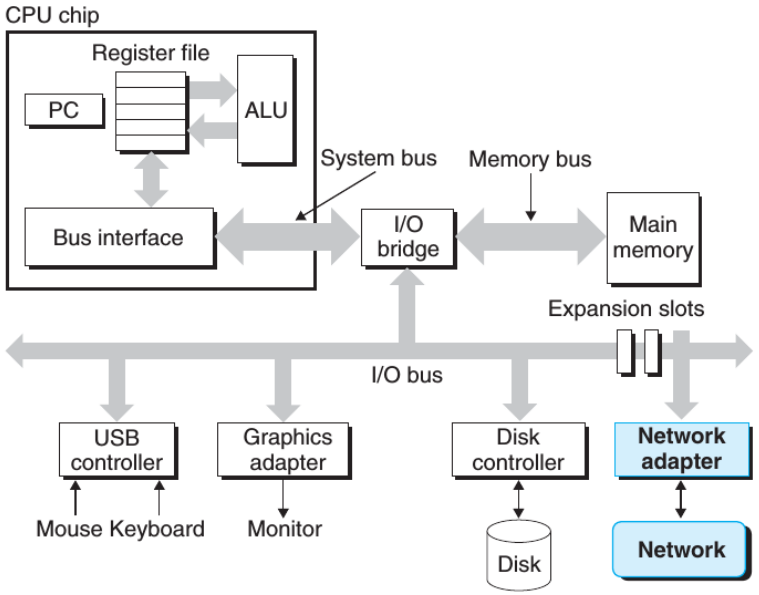

Hardware Event
System Queue
App3 Message Queue
App2 Message Queue
App1 Message Queue
Post Message
Application 1
Message Loop
Window Procedure
Default Window Procedure
Send Message


User
OS
QApplication
QPushButton
QObject
1: User{click}()
2: OS{ON_CLICKED}()
3: {ON_CLICKED}->QEvent()
4: mouseReleaseEvent()
5: Event()
6: click()
7: SIGNAL{click}()
8: SLOT{onButtonClicked()}()

多核与超线程







线程概念

- **线程**（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

- 线程是进程中的一个实体，是CPU调度和分配的基本单位。

线程共享资源

同一进程的多个线程共享同一地址空间，因此Text Segment、Data Segment都是共享的，如果定义一个函数，在各线程中都可以调用，如果定义一个全局变量，在各线程中都可以访问到。

除此之外，各线程还共享以下进程资源和环境：

- 文件描述符表
- 每种信号的处理方式（SIG_IGN、SIG_DFL或者自定义的信号处理函数）
- 当前工作目录
- 用户id和组id

线程独享资源

- 线程id
- 上下文，包括各种寄存器的值、程序计数器和栈指针
- 栈空间
- errno变量
- 信号屏蔽字
- 调度优先级

线程与进程的区别

线程与进程的区别可以归纳为以下4点：

1. 地址空间和其它资源（如打开文件）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。
2. 通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。
3. 调度和切换：线程上下文切换比进程上下文切换要快得多。
4. 在多线程OS中，进程不是一个可执行的实体。

线程库

- 在Linux上线程函数位于libpthread共享库中，因此在编译时要加上-lpthread选项。

线程控制

创建线程

```
#include <pthread.h>

int pthread_create(pthread_t  *restrict thread,
                   const pthread_attr_t *restrict attr,
          void *(*start_routine)(void*), void *restrict arg);
```

返回值：成功返回0，失败返回错误号。
thread：　　线程标识符
attr:　　线程属性设置，没有特殊设定，设置为NULL
start_routine：线程函数起始地址
arg：　　传递给start_routine的参数

restrict

备注:

关键字restrict只用于限定指针；

该关键字用于告知编译器，所有修改该指针所指向内容的操作全部都是基于(base on)该指针的，即不存在其它进行修改操作的途径；这样的后果是帮助编译器进行更好的代码优化，生成更有效率的汇编代码。

创建线程例子

- pthread.c

- thread_t类型是一个地址值，属于同一进程的多个线程调用getpid(2)可以得到相同的进程号，而调用pthread_self(3)得到的线程号各不相同。

- 由于**pthread_create的错误码不保存在errno中**，因此不能直接用perror(3)打印错误信息，可以先用strerror(3)把错误码转换成错误信息再打印。

- **如果任意一个线程调用了exit或_exit，则整个进程的所有线程都终止，从main函数return也相当于调用exit。**

终止线程

- 从线程函数return。

  这种方法对主线程不适用，从main函数return相当于调用exit。

- 一个线程可以调用pthread_cancel终止同一进程中的另一个线程。

- 线程可以调用pthread_exit终止自己。

线程退出

#include <pthread.h>

void pthread_exit(void *retval)

retval： pthread_exit调用者线程的返回值，可由其他函数和pthread_join来检测获取。

注意，pthread_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了

pthread_cancel

#include <pthread.h>
int pthread_cancel(pthread_t id)
返回值: 若成功返回0,否则返回错误编号

pthread_cancel并不等待线程终止,它仅仅**提出请求**

等待线程退出

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

int pthread_join(pthread_t th，void　** value_ptr)

th:　　等待线程的标识符

value_ptr：用户定义指针，用来存储被等待线程的返回值,

调用该函数的线程将挂起等待，直到id为thread的线程终止。

等待线程退出

thread线程以不同的方法终止，通过pthread_join得到的终止状态是不同的，总结如下：

■如果thread线程通过return返回，value_ptr所指向的单元里存放的是thread线程函数的返回值。

■如果thread线程被别的线程调用pthread_cancel异常终止掉，value_ptr所指向的单元里存放的是常数PTHREAD_CANCELED。

#define **PTHREAD_CANCELED** ((void *) -1)

■如果thread线程是自己调用pthread_exit终止的，value_ptr所指向的单元存放的是传给pthread_exit的参数。

■如果对thread线程的终止状态不感兴趣，可以传NULL给value_ptr参数。

例子

- pthread_exit.c

- 一般情况下，线程终止后，其终止状态一直保留到其它线程调用**pthread_join**获取它的状态为止。
- 但是线程也可以被置为**detach**状态，这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。
- 不能对一个已经处于**detach**状态的线程调用**pthread_join**，这样的调用将返回**EINVAL**。
- 对一个尚未**detach**的线程调用**pthread_join**或**pthread_detach**都可以把该线程置为**detach**状态，也就是说，不能对同一线程调用两次**pthread_join**，或者如果已经对一个线程调用了**pthread_detach**就不能再调用**pthread_join**了。

pthread_detach

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

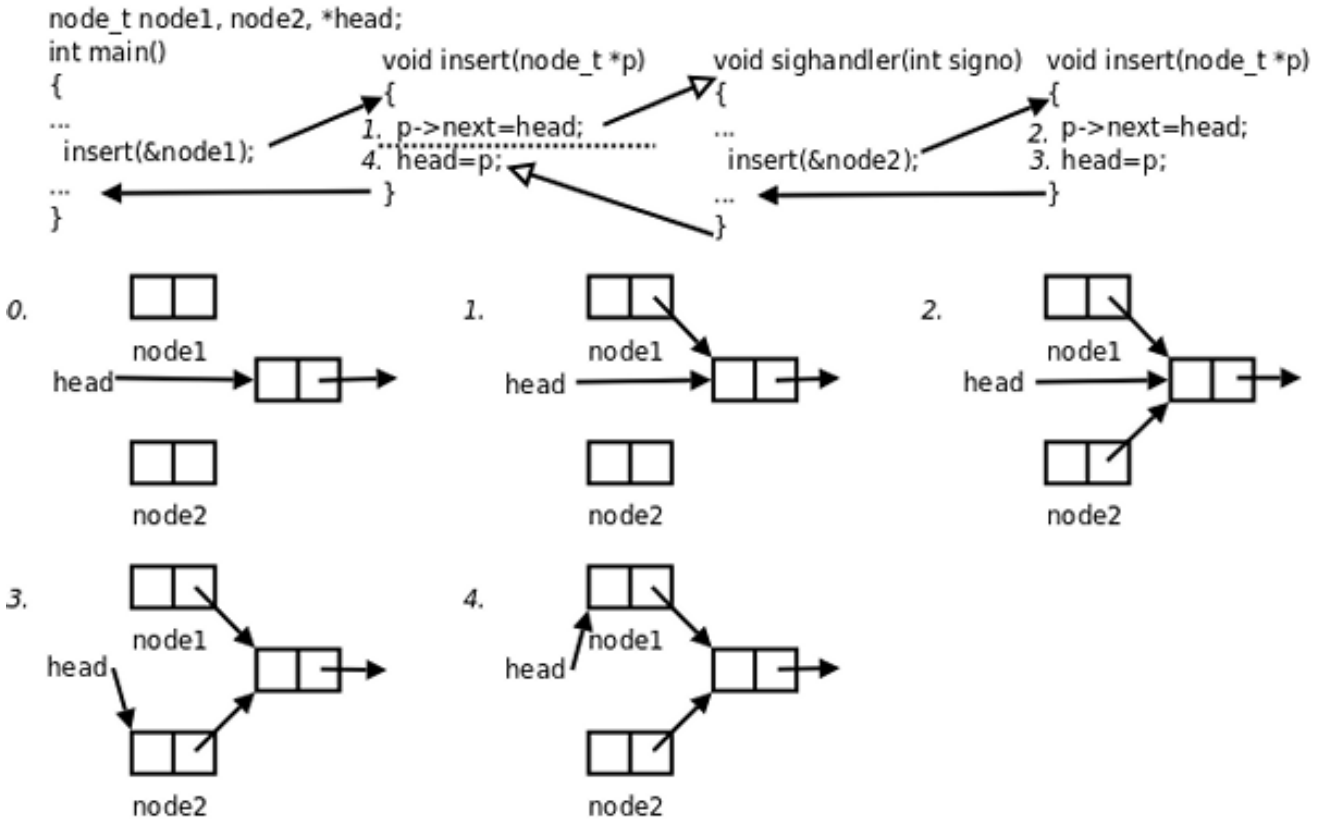
返回值：成功返回0，失败返回错误号。

线程间同步

访问冲突

先分析**counter.c**程序

- 多个线程同时访问共享数据时可能会冲突，这跟信号时所说的可重入性是同样的问题。



访问冲突

- 对于多线程的程序，访问冲突的问题是很普遍的，解决的办法是引入互斥锁（Mutex，Mutual Exclusive Lock），获得锁的线程可以完成“读-修改-写”的操作，然后释放锁给其它线程，没有获得锁的线程只能等待而不能访问共享数据，这样“读-修改-写”三步操作组成一个原子操作，要么都执行，要么都不执行，不会执行到中间被打断，也不会在其它处理器上并行做这个操作。

互斥锁

- 互斥锁的操作主要包括以下几个步骤：

互斥锁初始化：**pthread_mutex_init**
互斥锁上锁：**pthread_mutex_lock**
互斥锁判断上锁：**pthread_mutex_trylock**
互斥锁解锁：**pthread_mutex_unlock**
消除互斥锁：**pthread_mutex_destroy**

mutex的初始化和销毁

- mutex用**pthread_mutex_t**类型的变量表示，可以这样初始化和销毁：

#include <pthread.h>

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

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

pthread_mutex_t mutex = **PTHREAD_MUTEX_INITIALIZER**;

返回值：成功返回0，失败返回错误号

mutex的初始化和销毁

pthread_mutex_init函数对**mutex**做初始化，参数attr设定Mutex的属性，如果attr为NULL则表示缺省属性。

用pthread_mutex_init函数初始化的**mutex**可以用**pthread_mutex_destroy**销毁。

如果Mutex变量是静态分配的（**全局变量或static变量**），也可以用宏定义PTHREAD_MUTEX_INITIALIZER来初始化，相当于用**pthread_mutex_init**初始化并且attr参数为NULL。

mutex的加锁与解锁

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

返回值：成功返回0，失败返回错误号。

mutex的加锁与解锁

- 一个线程可以调用**pthread_mutex_lock**获得**Mutex**，如果这时另一个线程已经调用**pthread_mutex_lock**获得了该**Mutex**，则当前线程需要挂起等待，直到另一个线程调用**pthread_mutex_unlock**释放**Mutex**，当前线程被唤醒，才能获得该**Mutex**并继续执行。

- 如果一个线程既想获得锁，又不想挂起等待，可以调用**pthread_mutex_trylock**，如果**Mutex**已经被另一个线程获得，这个函数会失败返回**EBUSY**，而不会使线程**挂起**等待。

例子

- mutex.c

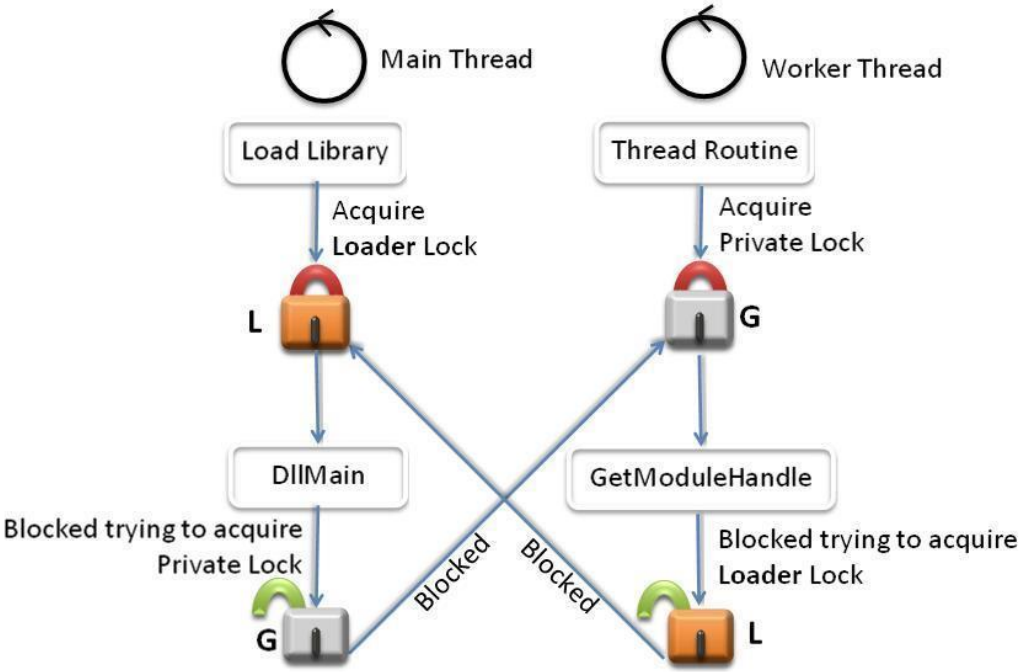
分析mutex的实现

```
lock:
    if(mutex > 0)
    {
        mutex = 0;
        return 0;
    } else
        挂起等待;
    goto lock;

unlock:
    mutex = 1;
    唤醒等待Mutex的线程;
    return 0;
```

死锁

- 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。



避免死锁的原则

- 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，以免死锁。

Condition Variable

- 线程间的同步的另外一种情况:

线程A需要等**某个条件**成立才能继续往下执行，现在这个条件不成立，线程A就阻塞等待，而线程B在执行过程中使这个条件成立了，就唤醒线程A继续执行。

在pthread库中通过条件变量（Condition Variable）来**阻塞**等待一个条件，或者**唤醒**等待这个条件的线程。

条件变量的初始化和销毁

```
#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);
pthread_cond_t cond =
PTHREAD_COND_INITIALIZER;
```

返回值：成功返回0，失败返回错误号。

pthread_cond_init函数初始化一个Condition Variable，**attr**参数为NULL则表示缺省属性，

pthread_cond_destroy函数销毁一个Condition Variable。

条件变量的操作

```
#include <pthread.h>

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);
```

返回值：成功返回0，失败返回错误号。

条件变量的使用

- 一个**condition variable**总是和一个**mutex**搭配使用的。一个线程可以调用**pthread_cond_wait**在一个Condition Variable上阻塞等待。

- 这个函数做以下三步操作：
  1. 释放Mutex
  2. 阻塞等待
  3. 当被唤醒时，重新获得Mutex并返回

条件变量

- **pthread_cond_timedwait**函数还有一个额外的参数可以设定等待超时，如果到达了abstime所指定的时刻仍然没有别的线程来唤醒当前线程，就返回ETIMEDOUT。
- 一个线程可以调用**pthread_cond_signal**唤醒在某个Condition Variable上等待的另一个线程
- 也可以调用**pthread_cond_broadcast**唤醒在这个Condition Variable上等待的所有线程。

例子

- 见producer.c

Posix信号量

- 信号量（semaphore）是一种用于提供不同进程间或一个给定进程的不同线程间同步手段的原语。

- 信号量的使用主要是用来保护共享资源，使得资源在一个时刻只有一个进程（线程）所拥有。
- 信号量的值为正的时候，说明它空闲。所测试的线程可以锁定而使用它。若为0，说明它被占用，测试的线程要进入睡眠队列中，等待被唤醒。

- Posix信号量分为有名信号量和无名信号量（也叫基于内存的信号量）。

Posix有名信号量

有名信号量既可以用于线程间的同步也可以用于进程间的同步。

1) 由sem_open来创建一个新的信号量或打开一个已存在的信号量。其格式为：

- **sem_t *sem_open(const char *name,int oflag,mode_t mode,unsigned int value);**
  返回：若成功则为指向信号量的指针，若出错则为SEM_FAILED 其中，第三、四个参数可以没有，主要看第二个参数如何选取。
  oflag参数：可以是0、O_CREAT或O_CREAT|O_EXCL。如果指定O_CREAT标志而没有指定O_EXCL，那么只有当所需的信号量尚未存在时才初始化它。但是如果所需的信号量已经存在也不会出错。 但是如果在所需的信号量存在的情况下指定O_CREAT|O_EXCL却会报错。
  mode参数：指定权限位。
  value参数：指定信号量的初始值。该初始值不能超过SEM_VALUE_MAX（这个常值必须至少为32767）.二值信号量的初始值通常为1，计数信号量的初始值则往往大于1。
  用sem_close来关闭该信号量。

Posix有名信号量

2) 使用sem_unlink删除信号量：
int sem_unlink(const char *name); 返回：成功返回0，出错返回-1

3) 获取信号量的当前值：
int sem_getvalue(sem_t *sem,int *valp); 返回：成功返回0，出错返回-1
sem_getvalue在由valp指向的整数中返回所指定信号量的当前值。如果信号量当前已上锁，那么返回值或为0，或为某个负数，绝对值即为等待等待该信号量解锁的线程数。

Posix有名信号量

4)信号量的等待：(P操作，也称为递减down 或 上锁lock)
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
返回：成功返回0，出错返回-1

sem_wait函数测试所指定信号量的值，如果该值大于0，就将它的值减1并立即返回；如果该值等于0，调用线程就被投入睡眠中，直到该值变为大于0，这时再将它减1，函数随后返回。“测试并减1”操作必须是原子的。sem_wait和sem_trywait的差别是：当所指定信号量的值已经是0时，后者并不将调用的进程投入睡眠。相反，它返回一个EAGAIN错误。如果被某个信号中断，sem_wait就可能过早的返回，返回的错误为EINTR。

5)信号量挂出：(V操作，也称为递增up 或解锁unlock)
int sem_post(sem_t *sem);返回：成功返回0，出错返回-1 将所指定的信号量值加1

采用Posix信号量实现生产者-消费者问题

对生产者-消费者问题进行扩展，把共享缓冲区用作一个环绕缓冲区，即生产者填写最后一项后回头来填写第一项，消费者也这么操作。此时需要维持三个条件：

（1）当缓冲区为空时，消费者不能试图从其中去除一个条目

（2）当缓冲区填满时，生产者不能试图往其中放置一个条目

（3）共享变量可能描述缓冲区的当前状态（下标、计数和链表指针），因此生产者和消费者的所有缓冲区操作都必须保护起来，以避免竞争。

给出使用信号量的方案展示三种不同类型的信号量：

（1）定义mutex二元信号量保护两个临界区。

（2）定义nempty的计数信号量统计共享缓冲区中的空槽位数。

（3）定义nstored的计数信号量统计共享缓冲区中已填写的槽位数。

Posix基于内存的信号量

Posix有名信号量创建时候是用一个name参数标识，它通常指代文件系统中的某个文件。而基于内存的信号量是由应用程序分配信号量的内存空间，即分配一个sem_t数据类型的内存空间，然后由系统初始化它们的值。操作函数如下：

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);  //初始化内存信号量

int sem_destroy(sem_t *sem);  //摧毁信号量
```

如果shared=0，那么待初始化的信号量是在同一进程的各个线程间共享的，否则该信号量是在进程间共享的，此时该信号量必须存放在某种类型的共享内存区中，使得用它的进程能够访问该共享内存区。value是该信号量的初始值。

Semaphore

- 信号量（semaphore）和mutex类似，表示可用资源的数量，和**mutex**不同的是这个数量可以大于1

```
#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);
```

Semaphore

- **semaphore**变量的类型为**sem_t**，
- **sem_init**()初始化一个**semaphore**变量，**value**参数表示可用资源的数量，**pshared**参数为0表示信号量用于**同一进程的线程间同步**

例子

- 见producer2.c

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.