linux 多线程编程

Posted on 2011-11-18 19:29 Biffo Lee 阅读(12890) 评论(0) 编辑 收藏

原文地址：

1.Linux“线程”

进程与线程之间是有区别的，不过Linux内核只提供了轻量进程的支持，未实现线程模型。Linux是一种“多进程单线程”的操作系统。Linux本身只有进程的概念，而其所谓的“线程”本质上在内核里仍然是进程。

大家知道，进程是资源分配的单位，同一进程中的多个线程共享该进程的资源（如作为共享内存的全局变量）。Linux中所谓的“线程”只是在被创建时clone了父进程的资源，因此clone出来的进程表现为“线程”，这一点一定要弄清楚。因此，Linux“线程”这个概念只有在打冒号的情况下才是最准确的。

目前Linux中最流行的线程机制为LinuxThreads，所采用的就是线程 - 进程“一对一”模型，调度交给核心，而在用户级实现一个包括信号处理在内的线程管理机制。LinuxThreads由Xavier Leroy (Xavier.Leroy@inria.fr)负责开发完成，并已绑定在GLIBC中发行，它实现了一种BiCapitalized面向Linux的Posix 1003.1c“pthread”标准接口。Linuxthread可以支持Intel、Alpha、MIPS等平台上的多处理器系统。

按照POSIX 1003.1c 标准编写的程序与Linuxthread 库相链接即可支持Linux平台上的多线程，在程序中需包含头文件pthread. h，在编译链接时使用命令：

```
gcc -D -REENTRANT -lpthread xxx. c
```

其中-REENTRANT宏使得相关库函数(如stdio.h、errno.h中函数) 是可重入的、线程安全的(thread-safe)，-lpthread则意味着链接库目录下的libpthread.a或libpthread.so文件。使用Linuxthread库需要2.0以上版本的Linux内核及相应版本的C库(libc 5.2.18、libc 5.4.12、libc 6)。

2.“线程”控制

线程创建

进程被创建时，系统会为其创建一个主线程，而要在进程中创建新的线程，则可以调用pthread_create：

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

start_routine为新线程的入口函数，arg为传递给start_routine的参数。

每个线程都有自己的线程ID，以便在进程内区分。线程ID在pthread_create调用时回返给创建线程的调用者；一个线程也可以在创建后使用pthread_self()调用获取自己的线程ID：

```
pthread_self (void) ;
```

线程退出

线程的退出方式有三：

（1）执行完成后隐式退出；

（2）由线程本身显示调用pthread_exit 函数退出；

```
pthread_exit (void * retval) ;
```

（3）被其他线程用pthread_cance函数终止：

```
pthread_cance (pthread_t thread) ;
```

在某线程中调用此函数，可以终止由参数thread 指定的线程。

如果一个线程要等待另一个线程的终止，可以使用pthread_join函数，该函数的作用是调用pthread_join的线程将被挂起直到线程ID为参数thread的线程终止：

```
pthread_join (pthread_t thread, void** threadreturn);
```

3.线程通信

线程互斥

互斥意味着“排它”，即两个线程不能同时进入被互斥保护的代码。Linux下可以通过pthread_mutex_t 定义互斥体机制完成多线程的互斥操作，该机制的作用是对某个需要互斥的部分，在进入时先得到互斥体，如果没有得到互斥体，表明互斥部分被其它线程拥有，此时欲获取互斥体的线程阻塞，直到拥有该互斥体的线程完成互斥部分的操作为止。

下面的代码实现了对共享全局变量x 用互斥体mutex 进行保护的目的：

```
int x; // 进程中的全局变量
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); //按缺省的属性初始化互斥体变量mutex
pthread_mutex_lock(&mutex); // 给互斥体变量加锁
... //对变量x 的操作
```

```
phtread_mutex_unlock(&mutex); // 给互斥体变量解除锁
```

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语言编程有多种线程同步机制，最典型的是条件变量(condition variable)。pthread_cond_init用来创建一个条件变量，其函数原型为：

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

pthread_cond_wait和pthread_cond_timedwait用来等待条件变量被设置，值得注意的是这两个等待调用需要一个已经上锁的互斥体mutex，这是为了防止在真正进入等待状态之前别的线程有可能设置该条件变量而产生竞争。pthread_cond_wait的函数原型为：

```
pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

pthread_cond_broadcast用于设置条件变量，即使得事件发生，这样等待该事件的线程将不再阻塞：

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

pthread_cond_signal则用于解除某一个等待线程的阻塞状态：

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

pthread_cond_destroy 则用于释放一个条件变量的资源。

在头文件semaphore.h 中定义的信号量则完成了互斥体和条件变量的封装，按照多线程程序设计中访问控制机制，控制对资源的同步访问，提供程序设计人员更方便的调用接口。

```
sem_init(sem_t *sem, int pshared, unsigned int val);
```

这个函数初始化一个信号量sem 的值为val，参数pshared 是共享属性控制，表明是否在进程间共享。

```
sem_wait(sem_t *sem);
```

调用该函数时，若sem为无状态，调用线程阻塞，等待信号量sem值增加(post)成为有信号状态；若sem为有状态，调用线程顺序执行，但信号量的值减一。

```
sem_post(sem_t *sem);
```

调用该函数，信号量sem的值增加，可以从无信号状态变为有信号状态。

4.实例

下面我们还是以名的生产者/消费者问题为例来阐述Linux线程的控制和通信。一组生产者线程与一组消费者线程通过缓冲区发生联系。生产者线程将生产的产品送入缓冲区，消费者线程则从中取出产品。缓冲区有N 个，是一个环形的缓冲池。

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 16 // 缓冲区数量
struct prodcons
{
    // 缓冲区相关数据结构
    int buffer[BUFFER_SIZE]; /* 实际数据存放的数组*/
    pthread_mutex_t lock; /* 互斥体lock 用于对缓冲区的互斥操作 */
    int readpos, writepos; /* 读写指针*/
    pthread_cond_t notempty; /* 缓冲区非空的条件变量 */
    pthread_cond_t notfull; /* 缓冲区未满的条件变量 */
};
/* 初始化缓冲区结构 */
void init(struct prodcons *b)
{
    pthread_mutex_init(&b->lock, NULL);
    pthread_cond_init(&b->notempty, NULL);
    pthread_cond_init(&b->notfull, NULL);
    b->readpos = 0;
    b->writepos = 0;

}
/* 将产品放入缓冲区，这里是存入一个整数*/
```

```
/* 将产品放入缓冲区,这里是存入一个整数 */
void put(struct prodcons *b, int data)
{
    pthread_mutex_lock(&b->lock);
    /* 等待缓冲区未满*/
    if ((b->writepos + 1) % BUFFER_SIZE == b->readpos)
    {
        pthread_cond_wait(&b->notfull, &b->lock);
    }
    /* 写数据,并移动指针 */
    b->buffer[b->writepos] = data;
    b->writepos++;
    if (b->writepos >= BUFFER_SIZE)
        b->writepos = 0;
    /* 设置缓冲区非空的条件变量*/
    pthread_cond_signal(&b->notempty);
    pthread_mutex_unlock(&b->lock);
}
/* 从缓冲区中取出整数*/
int get(struct prodcons *b)
{
    int data;
    pthread_mutex_lock(&b->lock);
    /* 等待缓冲区非空*/
    if (b->writepos == b->readpos)
    {
        pthread_cond_wait(&b->notempty, &b->lock);
    }
    /* 读数据,移动读指针*/
    data = b->buffer[b->readpos];
    b->readpos++;
    if (b->readpos >= BUFFER_SIZE)
        b->readpos = 0;
    /* 设置缓冲区未满的条件变量*/
    pthread_cond_signal(&b->notfull);
    pthread_mutex_unlock(&b->lock);
    return data;
}

/* 测试:生产者线程将1 到10000 的整数送入缓冲区,消费者线
   程从缓冲区中获取整数,两者都打印信息*/
#define OVER ( - 1)
struct prodcons buffer;
void *producer(void *data)
{
    int n;
    for (n = 0; n < 10000; n++)
    {
        printf("%d --->\n", n);
        put(&buffer, n);
    } put(&buffer, OVER);
    return NULL;
}


void *consumer(void *data)
{
    int d;
    while (1)
    {
        d = get(&buffer);
        if (d == OVER)
            break;
        printf("--->%d \n", d);
    }
    return NULL;
}

int main(void)
{
    pthread_t th_a, th_b;
    void *retval;
    init(&buffer);
    /* 创建生产者和消费者线程*/
    pthread_create(&th_a, NULL, producer, 0);
    pthread_create(&th_b, NULL, consumer, 0);
    /* 等待两个线程结束*/
    pthread_join(th_a, &retval);
    pthread_join(th_b, &retval);
    return 0;
}
```

5.WIN32、VxWorks、Linux线程类比

目前为止，笔者已经创作了《基于嵌入式操作系统VxWorks的多任务并发程序设计》（《软件报》2006年5~12期连载）、《深入浅出Win32多线程程序设计》（天极网技术专题）系列，我们来找出这两个系列文章与本文的共通点。

看待技术问题要瞄准其本质，不管是Linux、VxWorks还是WIN32，其涉及到多线程的部分都是那些内容，无非就是线程控制和线程通信，它们的许多函数只是名称不同，其实质含义是等价的，下面我们来列个三大操作系统共同点详细表单：

事项	WIN32	VxWorks	Linux
线程创建	CreateThread	taskSpawn	pthread_create
线程终止	执行完成后退出；线程自身调用ExitThread函数即终止自己；被其他线程调用函数TerminateThread函数	执行完成后退出；由线程本身调用exit退出；被其他线程调用函数taskDelete终止	执行完成后退出；由线程本身调用pthread_exit 退出；被其他线程调用函数pthread_cance终止
获取线程ID	GetCurrentThreadId	taskIdSelf	pthread_self
创建互斥	CreateMutex	semMCreate	pthread_mutex_init
获取互斥	WaitForSingleObject、WaitForMultipleObjects	semTake	pthread_mutex_lock
释放互斥	ReleaseMutex	semGive	phtread_mutex_unlock
创建信号量	CreateSemaphore	semBCreate、semCCreate	sem_init
等待信号量	WaitForSingleObject	semTake	sem_wait
释放信号量	ReleaseSemaphore	semGive	sem_post

6.小结

本章讲述了Linux下多线程的控制及线程间通信编程方法，给出了一个生产者/消费者的实例，并将Linux的多线程与WIN32、VxWorks多线程进行了类比，总结了一般规律。鉴于多线程编程已成为开发并发应用程序的主流方法，学好本章的意义也便不言自明。

完

```
#include <stdio.h>
#include <stdio.h>
#include <pthread.h>
void thread(void)
{
    int i;
    for(i=0;i<3;i++)
        printf("This is a pthread.\n");
}

int main(void)
{
    pthread_t id;
    int i,ret;
    ret=pthread_create(&id,NULL,(void *) thread,NULL);
    if(ret!=0){
        printf ("Create pthread error!\n");
        exit (1);
    }
    for(i=0;i<3;i++)
        printf("This is the main process.\n");
    pthread_join(id,NULL);
    return (0);
}
```

编译：

gcc example1.c -lpthread -o example1

```
#include <pthread.h>
#include <stdio.h>
#include <sys/time.h>
#include <string.h>
#define MAX 10

pthread_t thread[2];
pthread_mutex_t mut;
int number=0, i;

```

```
11  void *thread1()
12  {
13      printf ("thread1 : I'm thread 1\n");
14      for (i = 0; i < MAX; i++)
15          {
16              printf("thread1 : number = %d\n",number);
17              pthread_mutex_lock(&mut);
18              number++;
19              pthread_mutex_unlock(&mut);
20              sleep(2);
21          }
22      printf("thread1 :主函数在等我完成任务吗？\n");
23
24      pthread_exit(NULL);
25
26  }
27
28  void *thread2()
29  {
30      printf("thread2 : I'm thread 2\n");
31      for (i = 0; i < MAX; i++)
32          {
33              printf("thread2 : number = %d\n",number);
34              pthread_mutex_lock(&mut);
35              number++;
36              pthread_mutex_unlock(&mut);
37              sleep(3);
38          }
39      printf("thread2 :主函数在等我完成任务吗？\n");
40      pthread_exit(NULL);
41  }
42
43
44  void thread_create(void)
45  {
46      int temp;
47      memset(&thread, 0, sizeof(thread)); //comment1
48      //创建线程
49      if((temp = pthread_create(&thread[0], NULL, thread1, NULL)) != 0) //comment2
50          printf("线程1创建失败!\n");
51      else
52          printf("线程1被创建\n");
53      if((temp = pthread_create(&thread[1], NULL, thread2, NULL)) != 0) //comment3
54          printf("线程2创建失败");
55      else
56          printf("线程2被创建\n");
57  }
58
59  void thread_wait(void)
60  {
61      //等待线程结束
62      if(thread[0] !=0) { //comment4
63          pthread_join(thread[0],NULL);
64          printf("线程1已经结束\n");
65      }
66      if(thread[1] !=0) { //comment5
67          pthread_join(thread[1],NULL);
68          printf("线程2已经结束\n");
69      }
70  }
71
72  int main()
73  {
74      //用默认属性初始化互斥锁
75      pthread_mutex_init(&mut,NULL);
76      printf("我是主函数哦，我正在创建线程，呵呵\n");
77      thread_create();
78      printf("我是主函数哦，我正在等待线程完成任务阿，呵呵\n");
79      thread_wait();
80      return 0;
```

```
81 }
```

编译：

```
gcc -lpthread -o thread_example lp.c
```

分类: 技术

Biffo Lee
关注 - 0
粉丝 - 8
+加关注

« 上一篇：[命令技巧]ls
» 下一篇：如何保存想要的网页内容