张伯阳 1551265

1. Linux 下线程的基本概念

线程的基本概念

在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列/指令序列”;对于每个进程至少有一个执行线程;

线程同步的几种方式

线程的最大特点是资源的共享性，但资源共享中的同步问题是多线程编程的难点。linux 下提供了多种方式来处理线程同步，最常用的是互斥锁、条件变量和异步信号。

1）互斥锁（mutex）

通过锁机制实现线程间的同步。同一时刻只允许一个线程执行一个关键部分的代码。

```
int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutex_attr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex *mutex);
int pthread_mutex_destroy(pthread_mutex *mutex);
int pthread_mutex_unlock(pthread_mutex *
```

(1)先初始化锁 init()或静态赋值 pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIER

attr_t 有:

PTHREAD_MUTEX_TIMED_NP:其余线程等待队列

PTHREAD_MUTEX_RECURSIVE_NP:嵌套锁,允许线程多次加锁,不同线程,解锁后重新竞争

PTHREAD_MUTEX_ERRORCHECK_NP:检错,与一同,线程请求已用锁,返回 EDEADLK;

PTHREAD_MUTEX_ADAPTIVE_NP:适应锁,解锁后重新竞争

(2)加锁,lock,trylock,lock 阻塞等待锁,trylock 立即返回 EBUSY

(3)解锁,unlock 需满足是加锁状态,且由加锁线程解锁

(4)清除锁,destroy(此时锁必需 unlock,否则返回 EBUSY,//Linux 下互斥锁不占用内存资源

线程的等待与唤醒

等待唤醒机制

等待唤醒机制，是指一个线程 A 调用了对象 Object 的 wait()方法进入等待状态，而另一个线程 B 调用了对象 Object 的 notify()或者 notifyAll()方法，线程 A 收到了通知后，从对象 Object 的 wait()方法返回，进而执行后续操作。

上述两个线程通过 Object 对象来完成交互，而对象上的 wait()和 notify()/notifyAll()的关系就如同开关信号一样，用来完成等待方和通知方的交互工作。

等待/通知的相关方法是任意 Java 类对象都具备的，Object 类都有实现

等待/通知的相关方法

notify()：通知一个对象上等待的线程，使其从 wait()方法返回，而返回的前提是该线程获得了对象的锁。

notifyAll()：通知所有等待在该对象上的线程

wait()：调用该方法的线程进入等待状态，只有等待另外的线程的通知或被中断才会返回，需要注意，调用 wait()方法后，会释放对象锁。
wait(long mills)：超时等待一段时间（单位毫秒），如果没有就超时返回。
wait(long mills , int n)：对超时时间更细粒度的控制，可以达到纳秒
调用 wait()和 notify()/notifyAll()需要注意的细节：
使用 wait()和 notify()/notifyAll()时需要先**对调用对象加锁**。
调用 wait()方法后，线程状态由运行状态(Running)变为等待状态(Waiting)，并将当前线程放置到对象的等待队列。
notify()/notifyAll()方法调用后，等待线程依旧不会从 wait()返回，需要调用 notify()/notifyAll()的线程释放锁之后，等待线程才有机会从 wait()返回。
notify()方法将等待队列中的一个等待线程移动到同步队列中，而 notifyAll()方法则是将等待队列中所有的线程全部移到同步队列，**被移动的线程有等待状态(Waiting)变为阻塞状态(Blocked)**。
从 wait()方法返回的前提是获得了调用对象的锁。
从上述细节中可以看到，等待唤醒机制依托于同步机制(同步对象锁)，其目的就是确保 wait()方法返回时能够感应到通知线程对变量做出修改。
在 Thread 的 API 与 wait()有一个很像的方法 sleep()方法，他们 2 者的区别如下：
sleep()在休眠的过程中是一直持有锁的，而 wait()是执行之后释放锁的。

线程和进程的相同点和不同点(准备一个实例，同时用线程和进程方式完成，比较之间的异同)

进程是程序执行时的一个实例，即它是程序已经执行到课中程度的数据结构的汇集。从内核的观点看，进程的目的就是担当分配系统资源（CPU 时间、内存等）的基本单位。
线程是进程的一个执行流，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。一个进程由几个线程组成（拥有很多相对独立的执行流的用户程序共享应用程序的大部分数据结构），线程与同属一个进程的其他的线程共享进程所拥有的全部资源。
"进程——资源分配的最小单位，线程——程序执行的最小单位"
进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

总的来说就是：进程有独立的地址空间，线程没有单独的地址空间（同一进程内的线程共享进程的地址空间）。（下面的内容摘自 Linux 下的多线程编程）
使用多线程的理由之一是和进程相比，它是一种非常"节俭"的多任务操作方式。我们知道，在 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，当然，在具体的系统上，这个数据可能会有较大的区别。
使用多线程的理由之二是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。
除了以上所说的优点外，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：
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。
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。
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。
从函数调用上来说，进程创建使用 fork()操作；线程创建使用 clone()操作。

```
#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;
void *thread1()
{
	printf ("thread1 : I'm thread 1\n");
	for (i = 0; i < MAX; i++)
	{
		printf("thread1 : number = %d\n",number);
		pthread_mutex_lock(&mut);
			number++;
		pthread_mutex_unlock(&mut);
		sleep(2);
	}
pthread_exit(NULL);
}
void *thread2()
{
	printf("thread2 : I'm thread 2\n");
	for (i = 0; i < MAX; i++)
```

```
        {
                printf("thread2 : number = %d\n",number);
                pthread_mutex_lock(&mut);
                        number++;
                pthread_mutex_unlock(&mut);
                sleep(3);
        }
pthread_exit(NULL);
}
void thread_create(void)
{
        int temp;
        memset(&thread, 0, sizeof(thread));          //comment1
        /*创建线程*/
        if((temp = pthread_create(&thread[0], NULL, thread1, NULL)) != 0)
//comment2
                printf("线程 1 创建失败!\n");
        else
                printf("线程 1 被创建\n");
        if((temp = pthread_create(&thread[1], NULL, thread2, NULL)) != 0)  //comment3
                printf("线程 2 创建失败");
        else
                printf("线程 2 被创建\n");
}
void thread_wait(void)
{
        /*等待线程结束*/
        if(thread[0] !=0) {                  //comment4
                pthread_join(thread[0],NULL);
                printf("线程 1 已经结束\n");
        }
        if(thread[1] !=0) {                //comment5
                pthread_join(thread[1],NULL);
                printf("线程 2 已经结束\n");
        }
}
int main()
{
        /*用默认属性初始化互斥锁*/
        pthread_mutex_init(&mut,NULL);
        printf("主函数创建线程\n");
        thread_create();
        thread_wait();
        return 0;
}
```