# Linux的补充

linux的组成：

Linux可以简单理解为很多库来构成的：

内核：直接与硬件交互；

运行在系统上的软件是不能直接与内核进行交互的，而是通过其他库，这些库再来调用内核;

eg:glibc里面就包含了pthread\_create()方法，java就是通过调用这个函数，他再去调用内核函数；

重编译glibc：

1. yum search java |grep -i --color jdk

yum命令查看java版本

1. yum install -y java-1.8.0-openjdk

安装完成可以使用java命令

1. yum install -y java-1.8.0-openjdk-devel.x86\_64

使用javac命令

1. 安装gcc：yum install -y gcc

用来编译C文件的，因为glibc就是c文件写的

1. 将需要安装的glibc复制到linux，eg 目录为/home/glibc
2. 解压到当前目录：

tar -zxvf glibc-2.19.tar.gz

1. 修改/home/glibc/glibc-2.19/nptl下的pthread\_mutex\_lock.c添加：

fprintf(stderr,"current mutex pid=%lu\n",pthread\_self());

//打印调用当前方法的线程ID

1. 创建一个编译后的目录eg glibc\_out，并cd到该目录
2. 编译：

../configure --prefix=/usr --disable-profile --enable-add-ons --with-headers=/usr/include --with-binutils=/usr/bin

--prefix=/usr 编译后的位置，覆盖系统的版本位置/usr/lib

1. make install

java调用执行本地方法：

1. 先编译JAVA文件：

javac SyncDemo.java

1. 将java文件编译为.h文件

javah SyncDemo

1. 把这个**GetOSThraedIdNative**.c编译成为一个动态链接库，这样在java代码里会被laod到内存：

生成.so文件

gcc -fPIC -I /usr/lib/jvm/java-1.8.0-openjdk/include -I /usr/lib/jvm/java-1.8.0-openjdk/include/linux -shared -o lib**GetOSThraedIdNative**.so **GetOSThraedIdNative**.c

libLubanThreadNative 此处的名称与System.*loadLibrary*的参数加lib+名字；

System.*loadLibrary*(**"GetOSThraedIdNative"**);

1. 做完这一系列事情之后需要把这个.so文件加入到path，这样java才能load到

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:{libLubanThreadNative.so}所在的路径

1. 使用java命令执行java文件

java code：

**public class** SyncDemo {  
 Object o = **new** Object();  
 **static** {  
 //本地方法C文件变异成.so文件对应的文件名  
 System.*loadLibrary*(**"GetOSThraedIdNative"**);  
 }  
 //执行结果为 t1 t2两个线程轮流获取锁 依次执行  
 **public static void** main(String[] args) {  
 SyncDemo sync = **new** SyncDemo();  
 sync.start();  
 }  
 //因为这两个匿名内部类 所以字节码文件是三个  
 **public void** start(){  
 Thread t1 = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while** (**true**) {  
 **try** {  
 TimeUnit.*MILLISECONDS*.sleep(**500**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 sync();  
 }  
 }  
 });  
 Thread t2 = **new** Thread(() -> {  
 **while** (**true**) {  
 **try** {  
 TimeUnit.*MILLISECONDS*.sleep(**500**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 sync();  
 }  
 });  
 t1.setName(**"t1"**);  
 t2.setName(**"t2"**);  
 t1.start();  
 t2.start();  
 }  
   
 //因为无法通过java代码获取OS层面的线程ID进行比较 所以增加一个本地方法  
 **public native void** getOSThraedId();  
 **public void** sync(){  
 **synchronized** (o){  
 getOSThraedId();  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
   
 }  
}

C code:

**#include**<pthread.h>  
**#include**<stdio.h>  
**#include**<stdlib.h>  
*//调用这个本地方法的JAVA类类名编译成的.h文件***#include "SyncDemo.h"**JNIEXPORT **void** JNICALL Java\_SyncDemo\_getOSThraedId(JNIEnv \*env, jobject c1){  
 printf(**"current java native tid:%lu-----\n"**,pthread\_self());  
 usleep(**700**);  
}

# 并发编程模型

## 进程和线程的区别和联系

一个程序至少有一个进程,一个进程至少有一个线程。

1. 根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位;
2. 开销方面：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。
3. 所处环境：在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行（通过CPU调度，在每个时间片中只有一个线程执行）
4. 内存分配方面：系统在运行的时候会为每个进程分配不同的内存空间, 进程在执行过程中拥有独立的内存单元；

而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。

1. 包含关系：没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

## java线程有OS(Linux)线程的关系

pthraed\_create()方法的四个参数说明：

|  |  |  |
| --- | --- | --- |
| pthread\_t \*thread | 传出参数，调用之后会传出被创建线程的id | 定义 pthread\_t pid; 继而取地址获取&pid |
| const pthread\_attr\_t \*attr | 线程属性，关于线程属性是linux的知识 | 在学习pthread\_create函数的时候一般传NULL，保持默认属性 |
| void \*(\*start\_routine) (void \*) | 线程的启动后的主体函数，相当于java当中的run方法 | 需要你定义一个函数，然后传函数名即可 |
| void \*arg | 前面参数主体函数的参数 | 如果没有可以传NULL |

通过C++来调用Linux的一个线程：

**#include** <pthread.h>*//头文件***#include** <stdio.h>  
pthread\_t pid;*//定义一个变量，接受创建线程后的线程id  
//定义线程的主体函数***void**\* thread\_entity(**void**\* arg){  
 printf(**"i am new Thread!"**);  
}  
*//main方法，程序入口，main和java的main一样会产生一个*进程*，继而产生一个main*线程**int** main() {  
 *//调用操作系统的函数*pthread\_create*创建线程，注意四个参数* pthread\_create(&pid,NULL,thread\_entity,NULL);  
 *//usleep是睡眠的意思，那么这里的睡眠是让谁睡眠呢？  
 //为什么需要睡眠？如果不睡眠会出现什么情况*  usleep(**100**);  
 printf(**"main\n"**);  
}

这就说明了系统内核也是支持多线程；

thread.start()方法调用的核心是：

**private native void** start0();

是一个native 本地方法

java调用的是start0(),这个本地方法肯定是一个C或者C++文件

JDK包含的内容：

1. 一系列sun公司提供的java库
2. 本地C文件(作用：1、实现调用操作系统的函数‘、；2、调用JVM的代码hotspot；)
3. C++写的hotspot虚拟机这个项目；

本地方法C文件调用的应该是hotspot虚拟机的代码，而hotspot的c++代码来调用os的函数；

重点 - java调用start方法来开启线程：

java通过start()方法开启一个线程，就是通过hotspot代理最终调用的os的pthread\_create()方法来创建一个新线程；所以可以说JAVA的线程和操作系统OS的线程是一一对应的，即java对线程的操作方法都可以在OS上找到对应的操作；

start()是来调用系统的函数的，所以start()方法不能写实现，而是通过调用对应的run()方法来实现相应的逻辑；

C语言调用java方法

java调用java方法：通过class对象来new 对应的java对象，然后调用它的方法；

C语言调用也是一样的，通过JVM找到对应的class文件，然后new对象来调用方法：

1. 定义变量jclass clas对象；
2. clas = (\*env)->FindClass(env,“Test”); //Test是对于java文件名；
3. 定义一个变量来承接对象jobject obj；jmethod cid对应java的方法；
4. 找到class之后调用cid = (\*env)->GetMethodID(env,cls,”<init>”,”()v”)

这里调用的方法<init>为java的构造方法，()v则说明是无参的；

这里的cid的c是constructor 构造方法的缩写；

1. 然后获取实例化对象：

obj = (\*env)->NewObject(env,clas,cid);

1. 调用方法：

定义对象jmethod rid；

int ret = (\*env)->CallIntMethod(env,obj,”run”,”()v”);

同获取构造函数的方法，第三个参数run是被调用的java方法名，第四个参数为该方法需传入的参数，()v表示传入参数为空；

1. 最终实现了C反调java方法；

重点 - JAVA的Thread调用start()方法是怎么调回run方法的？

1. thread调用start()，start调用的是本地方法start0()；
2. start0()方法调用的是操作系统的pthraed\_create()创建线程，并传入四个参数；
3. pthraed\_create()会调用一个主体函数java\_start，而这个主体函数通过JNI反射来实现调用JAVA的run();

java为什么不通过本地C文件直接调用OS的相关函数：

假如直接调用，则这个C文件会被十分频繁的调用去与OS函数进行交互；所以它调用了hotspot的c++的javaThread，而他去通过对应的方法与OS进行交互；

# 并发偏向锁的存在

## Thraed的API

1. 中断停止线程

线程一直处于运行的状态，比如阻塞(线程阻塞通常是指一个线程在执行过程中暂停，以等待某个条件的触发。)或者while(true);

JVM不推荐直接停止一个线程，因为会导致有些资源来不及释放，当前线程更改的变量来不及还原；一定要让一个线程执行结束；

当前线程一般有三个状态：

1. 正在执行：应该等执行结束终止；
2. 真的阻塞：解阻塞，然后继续；
3. 无限循环：改变循环条件，继续执行；
4. 修改静态变量

**private static** Thread *thread*;  
**private static boolean** *running* = **true**;  
**public static void** main(String[] args) **throws** InterruptedException {  
 *run*();  
 TimeUnit.*SECONDS*.sleep(**1**);  
 *running* = **false**;  
}  
**private static void** run() {  
 *thread* = **new** Thread(() -> {  
 **while** (*running*)  
 System.*out*.println(**"running"**);  
 });  
 *thread*.start();  
}

这样做的弊端，可能单前线程不能立即结束；

并且这样处理也是线程可见性变差；

**while** (*running*){  
 System.*out*.println(**"running"**);  
 **try** {  
 TimeUnit.*MILLISECONDS*.sleep(**1100**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println(**"sleep over ..."**);  
}

线程休眠结束后的逻辑还是会被执行

PS：指令重拍

如果while循环内不执行任何操作，这样即使*running改为false*也不会终止线程；只要有执行内容就会终止，原因：

因为JVM会发生指令重拍；

这里证明了指令重拍可以发生在编译阶段；它也可以执行在执行阶段；

指令重排是指JVM在编译Java代码的时候，或者CPU在执行JVM字节码的时候，对现有的指令顺序进行重新排序。

比如：代码int i=0;int k=0;i++;

执行时如果按顺序执行，到i++时，CPU需要去寄存器中重新寻找i的地址，比较麻烦，所以CPU在实际执行过程中会进行优化变成int i=0;i++;int k=0;这样直接可以i++；

分为保守优化和激进优化；

指令重排的目的是为了在不改变程序执行结果的前提下，优化程序的运行效率。需要注意的是，这里所说的不改变执行结果，指的是不改变单线程下的程序执行结果。

然而，指令重排是一把双刃剑，虽然优化了程序的执行效率，但是在某些情况下，会影响到多线程的执行结果。

如果while循环内没有执行内容，会被CPU将指令优化为：

*thread* = **new** Thread(**new** Runnable() {  
 **boolean** temp = *running*;  
 **public void** run() {  
 **while** (!temp){}  
 }  
});

用一个临时变量来承接，不用再频繁去寻找running的地址，但是也造成了temp一直为false而使得当前执行不会结束；

而如果里面有方法的调用，则CPU担心有方法溢出，就不再执行这个优化了；

如果只调用一个i++之类的也不会被终止；

但是如果main()方法中不休眠一定时间，却会终止：

这是因为当主线程开启一个新线程，CPU 80%的概率会优先执行主线程，即先执行*running* = **false**;然后编译到指令重拍后的**boolean** temp = *running = false*;所以新开的线程就终止了；

1. interrupt()

其作用是中断此线程（此线程不一定是当前线程，而是指调用该方法的Thread实例所代表的线程），但实际上只是给线程设置一个中断标志，线程仍会继续运行。

如果当前线程在执行过程(使用的是让当前线程休眠一定时间来模拟)中被直接调用interrupt()会抛出异常，并且当前线程不会终止:

java.lang.InterruptedException: sleep interrupted

所以终止线程推荐以下两种方法：

1. 假设当前线程被阻塞，就需要解阻塞，仍然让当前线程执行结束；
2. 处于while(flag)；flag=true循环中，JVM将flag改为false；

即调用*thread*.isInterrupted()属性，默认为true

**private static** Thread *thread*;  
**public static void** main(String[] args) **throws** InterruptedException {  
 *run*();  
 TimeUnit.*MILLISECONDS*.sleep(**500**);  
 *thread*.interrupt();  
}  
**public static void** run() {  
 *thread* = **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while** (!*thread*.isInterrupted()){  
 System.*out*.println(**"running"**);  
 }  
 }  
 });  
 *thread*.start();  
}

1. synchronized关键字

对象的五种状态

无锁、偏向锁、轻量锁、重量级锁和被GC标记状态

synchronized关键字可以实现的锁分为三类：

synchlironized 1.6之前都是重量级锁；

使用synchronized关键字实现重量级锁的步骤：

1. 调用OS函数来实现锁，是通过int pthread\_mutex\_t mt;(PS:这里的int类型只是辅助理解它的参数类型为整型)这个参数，然后调用函数pthread\_mutex\_init(mt)来获取锁；
2. 里面pthread\_mutex\_lock(mt);对mt执行加1操作，如果+1操作不成功，就没有获取，等待前一个锁的释放；
3. 获取锁执行结束后执行pthread\_mutex\_unlock(mt)函数来-1变成0，释放锁；
4. 偏向锁

实际情况下方法一定要保证线程安全，但是实际情况不一定有线程互斥，所以偏向锁是synchronized锁的对象如果没有资源的竞争的情况下存在的；

偏向锁第一次会调用os函数，后续是不会再调用OS函数；

偏向锁是当前线程锁的对象获取锁，并在锁上打上标记；

比如：

**public synchronized void** sync(){  
 System.*out*.println(Thread.*currentThread*().getName());  
}

sync这个方法加入只有一个线程在运行，而且**synchronized**锁的对象是当前对象，相当于**synchronized(this)**，这时是没有竞争，**synchronized**就是偏向锁；

偏向锁的证明：

调用锁就必然调用os的pthread\_mutex\_lock(mt)函数来对锁对象上锁；

对于偏向锁，第一次会pthread\_mutex\_lock(mt)函数来上锁，而后续调用了synchronized关键字也不会再调用这个函数；

证明的思路：执行java代码时将Linux执行OS函数的线程ID都打印出来，同时在java代码synchronized代码块中也将当前的执行线程ID打印出来；因为实现锁会调用OS函数pthread\_mutex\_lock(mt)，此时会打印出对应的ID，如果与JAVA synchronized代码块中打印的线程ID一致；根据两个语句打印的次数可以判断锁的类型；

1. 轻量级锁
2. 重量级锁

借助于OS的函数来实现的锁；因为java调用OS函数比较麻烦，JVM需要从内核态切换为用户态；

synchronized作为重量级锁是为了多个线程在互斥时能够正常的运行；

线程互斥：指同一资源同时只允许一个访问者对其进行访问，具有唯一性和排他性；但是互斥无法限制访问者对资源的访问顺序，即访问是无序的；