操作系统 PBL 文献综述

所在学院：计算机学院

所在专业：计算机科学与技术

PBL 主题：进程与线程死锁

PBL 题目：java 多线程中的同步与死锁

小组成员：黄继升 (16041321) 陈乐聪(16272211)

郑耶 (16051735) 奚尊严(16051631)

一、参考论文

1.《基于 Java 多线程的并发机制的研究和实现》—— 王金海，江务学

2.《浅析 Java 多线程中的同步和死锁》—— 赵元媛

3.《Linux 下的多线程编程》—— 杨传安，王国夫，张海勋

4.《操作系统中的死锁检测》—— 张海鹏，李曦

二、小组分工

（1）报告撰写、论述： 黄继升 16041321

（2）PPT 制作： 陈乐聪 16272211

三、文献综述

3.1 进程线程

3.1.1 进程和线程的概念比较

进程是一段静态代码即程序在处理机上的一次动态的运行过程,是一个具有产生、发展和消亡的过程。

线程是比进程更小的执行单位,一个进程有一个或多个线程,线程也是一个具有创建、存在和消亡的动态过程。

3.1.2 职能比较

线程是进程的重要组成部分。传统的进程身兼两职:作为资源的分配和 CPU 的调度的基本单位，为了更好地实现开发程序的并发性,让进程摆脱繁重的任务,将其职能分离出来 ,让称为线程的实体分担，这也就产生了线程。进程是资源分配的基本单位，线程是 CPU 调度的基本单位。进程申请并获取所需资源，其对应的一些线程便在这些资源内活动并利用之。

3.1.3 系统开销比较

因为进程和线程在上述职能的区别，所以线程之间转换及其间的切换都比进程做等量工作所需要的系统开销小，而且更容易实现多线程之间的同步和通信。

3.2 Java 进程与线程

3.2.1 java 线程的生命周期

Java 中的线程具有动态的生命周期，一个完整的周期经历五种状态，其间的转换及相应的线程控制方法如图 1 所示。

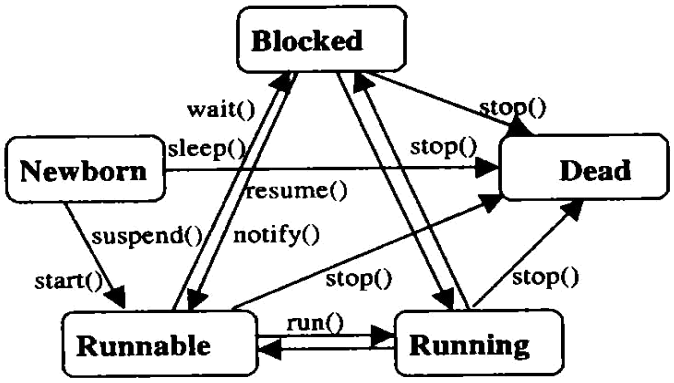

图 1　线程状态转换及控制方法图

(1)新生态(Newborn)：创建了(new)线程类子类的实例并初始化后，该对象就处于新生态，此时有了相应的存储空间和相应进程的资源。

(2)就绪态(Runnable)：处于新生态的线程被启动后(start()方法启动线程)，就处于就绪态，即进入就绪队列等待 CPU 时间片的到来。此时已具备了运行的条件。

(3)运行态(Running)：处于就绪态的线程获得 CPU 时间片即处于运行态，此时执行对应线程的 run()方法的代码来完成相应的操作。

(4)阻塞态(Blocked)：正在运行或处于就绪态的线程由于某种原因，让出 CPU 并暂时终止自己的执行(sleep 方法和 wait 方法)，即进入阻塞态，只有当被阻塞的原因被解除时方可转入就绪或运行态。

(5)死亡态(Dead) :一个线程完成了其所有操作或被提前强行终止时即死亡 ，此时线程不能再被恢复和执行(stop()方法)。

3.2.2 java 线程的创建方式

在大型实用系统软件的开发过程中，通常需要多个任务在宏观上同时执行，同时也需要充分利用系统资源和提高程序的执行效率，多线程能提供完好的解决方案。Java 是一种完全意义上的面向对象的程序设计语言，提供了丰富的类库(包)和应用接口，来方便用户开发利用。Java 中创建线程有两种方法。

(1) 继承 Thread 子类

Java.lang.Thread 是系统提供的用来表示进程的类,Thread 类中的许多方法提供了比较完整的多线程处理的功能。可以用 Thread 来定义子类，构造用户的线程。例如:

```
public MyThread extends Thread
{
  …//相关属性和方法定义
  public void run()
  {
    … // 线程体代码
  }
  public static void main(String args[])
  {
```

```
    Thread t =new MyThread();
    …
    t.start();//启动线程 t
    …
  }
}
```

其中 run()方法至关重要，也是整个线程的核心，其代码是线程所要执行的内容，start()方法的作用是调用 run()方法。

(2) 实现 Runnable 接口

Java 中提供了一个实现多线程的接口 Runnable，该接口只有一个方法 run()，用户通过重载该方法以实现线程的相关操作，有了 Runnable 接口就可被系统自动识别并执行。例如:

```
public class MyThread implements Runnable
{
  …//相关属性定义
  public void run()
  {
    … // 线程所要执行操作的代码
  }
}
```

启动的方法有两种，但其实质还是一致的，如下所示：

```
MyThread my = new MyThread();
Thread t = new Thread(my);
t.start();
或
MyThread my = new MyThread();
new Thread(my).start ;
```

3.2.3 两种线程创建方式比较

第一种方法层次清晰，逻辑分明，使用简便，实质上也实现了 Runnable 接口;第二种方法可以弥补第一种方法的不足，即可以实现多重继承(Java 中的类只能单重继承),例如在小程序 Applet 中的应用。Runnable 接口还要以 Thread 类为框架来实现 。

3.3 Java 并发机制的分析和研究

在并发编程中，我们通常会遇到三个问题：原子性问题，可见性问题，有序性问题。只有这三个问题得到解决，才能保证线程并发执行的正确性。

3.3.1 基于 synchronized 和 monitor 的互斥机制

并发性的引入，在多道程序设计中，为了达到某种实际目的，程序中总会存在着一些不能“同时”(实质是宏观上的同时)被两个以上的线程访问的程序段，即临界资源。Java 中利用 synchronized 锁定标志来定义临界资源，这里的临界资源可以是方法，也可以是代码段。在程序中有 synchronized 标志的方法

或代码段在任一时刻只能有一个线程可以进入访问 ,即实现了临界资源的互斥访问。在实现原理上有引入了类似于信号量机制的监视器管程(Monitor), Java 为每一个对象都分配一个管程 ,其作用是负责管理线程对临界资源的访问。一个线程访问某临界资源 ,就获得了该临界资源所属对象的管程 ,并且管程“上锁”,其它想访问该临界资源的线程必须等待至管程“开锁”。在一段时间内只有一个线程拥有监视器 ,拥有监视器的线程才能访问相应的资源 ,并锁定资源不让其他线程访问 ,所有其他的线程在试图访问被锁定的资源时被挂起 ,等待监视器解锁。synchronized 可以保证变量的原子性和可见性。

3.3.2 volatitle 关键字解析

之前使用 synchronized 同步锁可以给多个线程访问的代码块加锁以保证线程安全性 ,但是弊端也比较明显 ,就是加锁后多个线程需要判锁 ,比较消耗资源。因此 java 也引入了一种轻量级的解决方案——volatile 关键字，用来修饰被不同线程访问和修改的变量。

首先谈一谈 JMM，即 Java 内存模型。其作用是使得 java 程序在不同平台上的内存访问效果一致。它决定了一个线程对共享变量的写入何时对另一个线程可见。JMM 的结构组成为主内存和工作内存，实际上可以类比 CPU、高速缓存和内存之间的关系。

主内存是由所有线程共享，共享变量在主内存中存储的是其“本身”；而工作内存是指每个线程都有自己的工作空间，共享变量在主内存存储的是其“副本”。每个线程对共享变量的所有操作全在自己的工作内存中进行，每个线程只能访问自己的工作内存，变量值的传递只能通过主内存完成。

volatile 关键字可以保证可见性和有序性。但无法保证原子性。

3.3.3 基于 wait()和 notify()的交互通信机制

synchronized 解决了临界资源的互斥访问问题，在现实应用中，还会要求线程之间进行协同合作工作，即在对临界资源的互斥访问的同时，线程之间还要相互通信和互多线程之间的交互问题，提供了 3 个标准的 Object 类的方法：wait()，notify()和 notifyAll()。wait()作用是使当前运行的线程由运行态转为阻塞态，进入等待队列中的 wait()集中，并且释放管程(监视器 monitor)；notify()的作用是唤醒 wait()集中的队首线程(或按照一定的算法来唤醒某个线程)，使线程由阻塞态转为可运行态；notifyAll()的作用是将所有 wait()集中的线程都唤醒。wait()和 notify()的配对使用能很好地实现互斥中通信机制，以切实解决上述列举的问题。而且，wait()和 notify()、notifyAll()方法的特性决定了这一对方法必须在 synchronized 方法或块中调用。

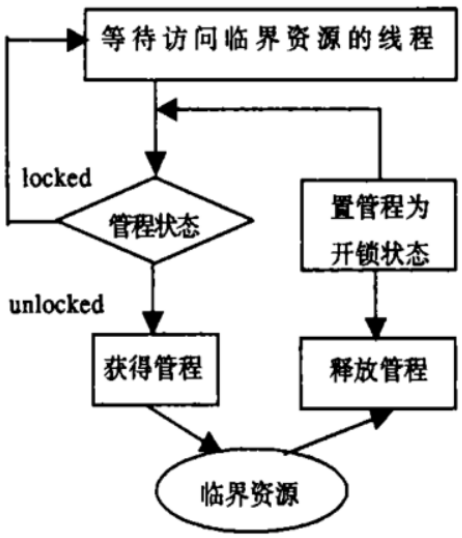

图 2 locked unlocked 互斥机制实现原理图

3.3.4 通过互斥和交互通信机制实现完全意义的同步

同步是指多个线程在一些关键点上的互相等待和互通消息。Java 的 synchronized 实现了线程的互斥机制，wait()和 notify()实现了线程间的互通消息的通信机制，分别解决了数据的一致性和多线程之间的协同合作的问题，实现了完全意义上的同步。图 3 将 Java 的同步机制和操作系统级的有信号灯和 P 、V 原语实现的同步机制作一简单比较。

```
synchronized(this)                    …
{   …                                 P(empty)
    wait();                           P(mutex)
    …                                     CS;
    notify();                         V(mutex)
    …                                     P(full)
}                                     …
```

图 3　Java 同步机制与操作系统同步机制的比较图

3.4 死锁

3.4.1 死锁问题

多线程在使用互斥机制实现同步时，存在“死锁”的潜在危险。死锁是由于两个或多个线程都无法得到相应的监视器而造成相互等待的现象。例如，在某一多线程的程序中有两个共享资源 A 和 B，并且每一个线程都需要获得这两个资源后才可以执行，这是一个同步问题，但是如果没有合理地安排获取这些资源

的顺序，就有可能出现线程 1 已经获取资源 A 的锁，由于某种原因被阻塞，此时线程 2 启动并获得资源 B 的锁 ,再去获得资源 A 的锁时发现线程 1 已经获取，因此等待线程 1 释放 A 锁。线程 1 从阻塞中恢复以后继续执行，欲获取资源 B 的锁，却发现 B 锁已被线程 2 获得，因此也陷入等待。在这种情况下，程序已无法向前推进，在没有外力的情况下，也不会自动退出，因而造成了严重的死锁问题。导致死锁的根源在于不适当地运用 Synchronized 关键字来管理线程对特定对象的访问，其产生的必要条件有如下几点：

①互斥条件：就是说多个线程不能同时使用同一资源，比如，当线程 A 使用该资源时，B 线程只能等待 A 释放后才能使用；

②占有且等待条件：就是某线程必须同时拥有 N 个资源才能完成任务，否则它将占用已经拥有的资源直到拥有它所需的所有资源为止，就好像游戏中，必须两个球都拿到了，才能释放；

③不可剥夺条件：就是说所有线程的优先级都相同，不能在别的线程没有释放资源的情况下，夺走其已占有的资源；

④循环等待条件：就是没有资源满足的线程无限期地等待。

目前处理死锁的基本方法主要有三种，一是预防死锁，二是避免死锁，三是检测和解除死锁。

遗憾的是，Java 技术并不在语言级别上支持死锁的避免，因此在编程中必须小心地避免死锁，而避免死锁的有效原则是：

1.当线程因为某个条件未满足而受阻，不能让其继续占用资源。

2.如果有多个对象需要互斥访问，应确定线程获得锁的顺序，并保证整个程序以相反的顺序释放锁。

因此，在 java 多线程编码中，主要是以死锁预防为主。

3.4.2 死锁预防

线程的同步机制中，可能会出现这样一种现象，几个线程各自拥有所需的部分资源，而又同时需要别的线程所拥有的部分资源，并等待着别的线程释放，否则都不能向前推进,这样一种由于资源竞争和分配而造成的一种僵持的现象即为死锁。Java 的并发机制是基于 JVM 级的,类似于操作系统的同步机制,死锁具有很大的隐藏性,用户使用时,可能很容易导致死锁故障,而 Java 又不能避免死锁和恢复死锁,只能在使用时对死锁进行预防。一般来说预防死锁主要是破坏产生死锁的四个条件。根据文献提供一些方法和策略，可以用来预防死锁。

(1) 线程的任务划分要明确 、合理。多线程的并发是死锁产生的根源，必须结合实际需要和 Java 多线程机制的特性合理安排各个线程的任务;保证资源的均衡性;尽量减少线程数。

(2) 正确使用同步机制 。synchronized 的使用是产生用户级死锁的直接原因，慎重考虑同步方法之间的调用;尽量避免同步方法之间的嵌套调用。

(3) 正确使用通信机制 。wait()和 notify()方法能解决一定的死锁问题，通常 wait()和 notify()配对使用 ，适当的时候考虑是否要用 notifyAll()方法。

(4) 临界资源要精练。临界资源是产生死锁的物质基础，尽量使临界资源精练，以减少资源竞争而产生的冲突。

3.4.5 一种 linux 的死锁检测算法

关于 java 多线程程序中的死锁检测,而且目前已经有比较成熟的工具可以直接检查 java 程序中的死锁，如 jstack、lockstat 等. 由于操作系统代码量大、对性能敏感，过去关于操作系统死锁方面的研究比较少。死锁检测工具是在操作系统层实现的，但是该工具只能检测应用程序中的死锁，并不能检测操作系统本身的死锁。这里提出一种动态检测死锁的方法，可以准确检测出操作系统中的死锁。对操纵系统性能的影响小于 1%，而且不需要修改内核源码和任何的应用程序。

当进程发生死锁时，进程既是锁的持有者，也是锁的等待者，而且进程间会形成一个循环等待图. 本文采用的死锁检测算法就是从锁的持有者中筛选出锁的等待者，然后进一步判断它们之间是否形成一个循环等待图.

3.4.5.1 死锁检测算法

(1) 每隔一定时间(search_cycle 指定)检查锁持有者链表 HOLDER_LIST 是否为空.

(2) 若不为空则检查这些锁的持有者是不是锁的等待者。

(3) 若检查结果为 ture，则检查这些锁的持有者与等待者之间是否会形成一个循环等待图。

(4) 如果存在循环等待图则说明发生了死锁。

```
1 while (1) {
2       sleep (search _cycle )
3       if HOLDER _LIST not NULL
4             search the lock  _holders from HOLDER   _LIST
5             if lock _holders are lock  _waiters
6                   if there is a cycle in waits  _for graph
7                         report deadlock bug
8                   else
9                         deadlock free
10            else
11                  deadlock free
12      else
13            deadlock free
14 }
```

图 4 死锁检测算法

3.4.5.2 获取锁的持有者算法：

使用可以动态探测 Linux 内核函数的工具 time 的进程 systemtap，它既不需要重新编译内核，也不需要修改任何源程序和库函数。

systemtap 既可以在函数的入口处进行探测，也可以在函数的出口处进行探测。若在加锁函数退出的地方进行探测，那么就可以获得锁的持有者信息，因为只有成功获得锁，进/线程才能从加锁函数中退出，否则便处于等待状态。

为了唯一标识进程加锁和解锁操作，我们可以使用由进程号(pid)、锁类型(lock_type)、资源地址(resource_addr)组成的三元组<pid, lock_type, resource_addr>。

(1) 当进程加锁的时候，将<pid, lock_type, resource_ addr>插入死锁检测程序维护的链表 HOLDER_LIST 中.

(2) 当进程解锁时，将<pid, lock_type, resource_ addr>从 HOLDER_LIST

链表中删除. 那么 HOLDER_ LIST 链表中保存的便是那些只有成功加锁而没有解锁的进程.

(3) 当死锁检查周期 search_cycle 时间到的时候,便可以从 HOLDER_LIST 中获得锁的持有者。

```
1 when thread lock
2       insert <pid , lock _type , resource _addr > to HOLDER _LIST
3 when thread unlock
4       delete <pid, lock _type , resource _addr > from HOLDER _LIST
5 while (1){
6       sleep(search _cycle)
7       if HOLDER_LIST not NULL
8             search the lock holders from HOLDER    _LIST
9 }
```

图 5 获得锁持有者算法

3.4.5.3 获取锁的等待者算法

虽然可以使用 systemtap 在加锁函数的入口处进行探测, 就可以获得该锁的所有申请者。因为申请者中除去锁的持有者, 剩下的便是等待该锁的进程。但是这种方式效率很低。所以 , 在这里提出一种通过筛选异常进程的方法来获得锁的等待者。

异常进程：等待时间超过阈值 threshold_time 的进程。

死锁进程必定是异常进程，异常进程中只有一部分是因为等待锁而异常的,为了获取这些锁的等待者,还需要对判定为异常的进程进行筛选。

计算等待时间：使用进程描述符 task_struct 结构体中的 last_arrival 为参考, 来判断进程的等待(包括忙等)时间是否达到 threshold_time。在进程的状

态切换中，只有当进程由就绪状态切换到运行态的时候，才改变 last_arrival 的值。因此，无论进程是在阻塞状态还是在运行态(忙等)，该值均不会改变。使用当前的系统时间减去 last_arrival 便是进程睡眠或忙等的时间，假若最后这个差值一旦超过了 threshold_time，那么就认为该进程是异常进程。

计算阈值 threshold_time：threshold_time 必须大于进程持有锁的时间 lock_time，否则会将正常持有锁的进程判为异常进程。另外阈 threshold_time 也必须大于进程的时间片 time_slice，这是为了筛选因为忙等而异常的进程。因此，阈值 threshold_time 应该大于两者中的最大值，即 thread_time > max (lock_time, time_slice)。

统计进程持有锁时间 lock_time 加锁函数与解锁函数之间的时间便是线程持有该锁的时间。操作系统中的锁可以分成两类：睡眠锁和非睡眠锁。睡眠锁允许锁的获得者在锁保护的临界区中睡眠，而非睡眠锁则不允许。由于睡眠锁粒度大，因此统计进程持有睡眠锁的时间比较有意义。操作系统中的睡眠锁有信号量(semaphores)、读/写信号量(read/write semaphore)和互斥体(mutex)，针对不同的使用场景，这些睡眠锁提供了不同的加锁函数。经过前人的研究和分析，互斥体 mutex 允许线程持有锁的时间是最长的。

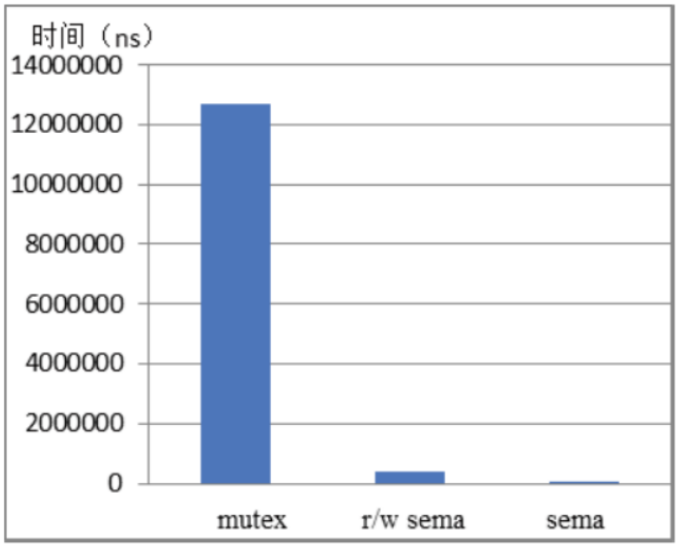

图 6 线程持有睡眠锁的时间

统计时间片 time_slice:在进程的描述符中会记录进程总的运行时间(sum_exec_runtime)以及运行次数(pcount)，这两者之间的比值便是进程的平均时间片 time_slice。通过 linux 压力测试工具 stress 可以模拟 cpu 密集型的任务，最后可以统计出进程的 time_slice。

所以经过以上统计可以得到 threshold 设定为 10s 是比较合适的。

3.4.5.4 筛选异常进程算法

当进程执行加锁函数的时候，内核会将加锁函数的返回地址压入进程的内核栈。当进程从加锁函数成功退出的时候，再将该地址从栈中弹出。因加锁函数而异常的进程，其内核栈上必定存在加锁函数的返回地址，内核提供的函数 kallsyms_lookup 可以根据该地址找到对应的加锁函数。因此，可以通过判断异常进程的内核栈上有无加锁函数，来从异常进程中筛选出锁的等待者。

3.4.5.5 循环等待图

根据死锁检测算法，为了检查线程间是否发生了死锁，需要判断线程的资源

依赖关系是否会形成一个循环等待图。若线程 T1、T2、T3 发生死锁，那么它们的资源依赖关系将如图 6 所示。

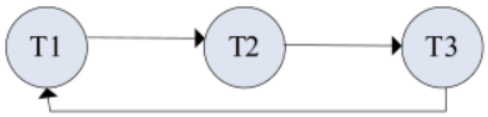

图 7 死锁中的循环等待图

3.4.5.6 论文实验结果

为了检测死锁检测方法的有效性，使用操作系统中的信号量模拟三个线程访问三种临界资源，如图 8 所示。每个线程获得一种资源后，延迟 1 秒钟，然后申请另外一种资源. 线程第一次申请的是三种不同的资源，可以成功获得，第二次申请是其它线程已经拥有的资源，由于加锁顺序不当导致发生死锁。

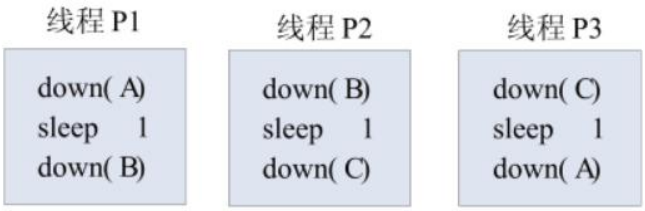

图 8 三个线程产生的信号量死锁

使用本文提供的死锁检测程序，检查结果如图 9 所示。临界资源使用资源地址来标识，进程 2471 拥有资源 a0(资源地址前 16 位均一样，只使用后两位标识)，申请资源 c0。进程 2472 拥有资源 c0，申请资源 e0。进程 2473 拥有资源 e0，申请资源 a0。通过资源依赖的传递关系，可以得出进程 2471 指向

自己的依赖关系：2471->2471。因此，进程间便存在死锁。

```
1  pid 2471   acquire resource ffffffffa00903 a0
2  pid 2472   acquire resource ffffffffa00903 c0
3  pid 2473   acquire resource ffffffffa00903 e0
4  pid 2471  wait resource ffffffffa00903 c0
5  pid 2472  wait resource ffffffffa00903 e0
5  pid 2473  wait resource ffffffffa00903 a0
7  2471 ->2472 2472 ->2473 2473 ->2471
8  above process has dead lock
```

图 9 三线程死锁检测结果