多任务处理环境下线程和进程的运行效率对比以及解决方法

郝泓毅

前言

在传统的操作系统中，为了提高资源利用率和系统吞吐量，通常会采用多道程序技术及将多个程序同时装入内存，使他们并发进行，即传统意义上的程序不在独立运行。此时，将资源分配和独立运行的基本单位都是进程，操作系统所具有的四大特征也都是基于进程而实现的。由此可见进程是一个极为重要的概念。而在操作系统中引入进程的目的是使多个程序并发执行，以提高资源利用率和系统吞吐量，操作系统再次引入线程，为了减少并发执行使所付出的时空开销，使操作系统有更好的并发性。在这种情况下我决定根据论文探索和研究线程和进程在多任务处理环境下的运行效率对比以及现成的解决方法。

进程 线程 纤程

1.线程

1.1 线程的定义

线程的概念与子程序的概念类似，是一个可独立执行的子程序。一般线程定义是进程式内的可以执行的单元，它是系统分配 CPU 时间资源的基本单元。一个应用程序可以创建多个线程，生成多个不同的执行流，并同时运行这些线程。线程机制使得系统具有多任务的功能，这样用户就可以同时运行多个应用程序，而每个应用程序中又可以同时运行多个线程，这些线程并发地运行在同一个进程之中。一个进程至少拥有 一个线程，即主线程。主线程终止，进程也终止。主线程以函数地址形式（通常为 Main 或 WinMam 函数的 地址）被启动代码提供给操作系统。也可以根据需要创建其它线程，每个线程都共享创建它们的父进程的内 存空间、全局变量和系统资源。

1.2 线程与进程的差别

进程由两个部分组成：一个是进程内核对象，一个是地址空间。同样线程也由两个部分组成：一个是线程的内核对象，操作系统用它来管理线程；线程

的内核对象同时也是系统用来存放线程相关信息的地方。一个是线程堆栈，它用于维护线程在执行代码时所需要的有关函数参数和局部变量。 进程是死板的。进程不执行任何操作，它只是线程的容器。线程是“动态”的，线程总是在某个进程环境 中创建只是在该进程中具有它的整个生存期。线程代码的执行和对数据的操作是在它的进程地址空间中 完成的。因此，在一个进程环境中运行的多个线程将共享单个地址空间。这些线程能够执行相同的代码和 操作相同的数据；还能共享内核对象句柄。进程需要相对较大的地址空间，这些空间用于存放系统资源及其相关的信息；而线程只有一个内核对象 和一个堆栈，所需要保存的信息也比较少。

1.3 线程的状态

一个新建的线程在它的生命期中有五种基本状态。

新建 一个新生的线程对象处于新建状态，它有了相应的内存空间和其它资源并被初始化。

就绪 处于新建状态的线程被启动后将进入线程队列排队等待 CPU 时间片，此线程已具备运行的条件。在此线程获得 CPU 资源时就可以脱离创建它的主线程而独立开始自己的生命周期。而原来处于阻塞状态的线程被解除阻塞也将进入就绪状态。

运行 就绪状态的线程被调度获得 CPU 处理器资源时就进入运行状态。每个线程对象都有一个重要的 run（）方法。在线程对象被调度执行时，它将自动调用本对象的 run（）方法，从第一个语句开始执行这个线程的操作和功能。

阻塞 一个正在执行的线程在某些特殊情况下，将进入阻塞状态，例如被人为地挂起。在阻塞状态时，线程不能进入排队队列。只有引起阻塞的原因被解除后，线程才能进入就绪状态，进入就绪状态才能进入线程排队队列等待 CPU 的资源，以便于从原来中止处继续运行状态。

消亡 线程消亡的原因有二：其一是正常运行的线程完成了它的全部工作而退出；其二是线程被强迫地终止执行。

线程的各个状态之间的转换及线程生命周期的演进，由系统运行的状态、同时存在的其它线程以及线程 本身的算法所共同决定。在创建和使用线程时要注意利用线程的方法进行控制。

2.进程

进程（process）这个概念最早是由麻省理工学院的 J．H．Saltzer 于 1966 年提出的并应用于 MULTICS 系统设计中。而 IBM 公司则称“进程”为任务（Task）并首先在 CTSS／360 系统中实现。

2.1 进程和进程的状态

进程概念的引入是为了描述系统中的各种并发活动，但是由于并发活动的复杂性人们强调的侧重各不相同，时至今日还没有统一的定义。在 1978 年全国操作系统会议上将进程定义为：进程是一个具有独立功 能的程序关于某个数据集合的一次运行活动。 进程具有动态性和并发性两个重要的属性。

动态性：程序是有序指令的集合　是一个静止的概念。进程是程序的执行，是一个动态的概念。程序作为一种资源可以长期存在。进程是程序的一次执行进程具有生命过程，即有自己的诞生、执行和消亡的生命周期。

并发性：几个进程可以同时存在于一个系统中各个进程各自独立地以不可预知的速度向前推进。进程是系统进行资源分配和调度的一个独立单位。各个进程有着运行和等待交替的运作规律。需要注意：对于 一个单 CPU 的计算机系统，任何时刻只能有一个进程占用 CPU 资源其它进程只能等待。

进程有就绪（Ready）、运行（Runing）和阻塞（Blocked）三种基本状态．它们在一定的条件下可以相互转换。

2.2 子进程

所谓的子进程就是在一个进程所拥有的地址空间内再启动一个新的进程，这个新的进程相对于启动它的进程称为子进程，而启动子进程的进程称之为父进程。

在程序的开发过程中，经常会遇到需要执行另外一块代码的操作。完成此项操作可以是：函数调用执行新的线程或者是使用新的进程。 调用函数时源程序必须停止运行等待被调用的函数执行完毕后再继续执行。函数调用适用于单任务 操作系统的情况下。线程调用可以使子线程和主线程分别运行但是当子线程和主线程需要交换数据时线程的使用比较复杂。

子进程是一个比较好的方法。新的进程可能也需要对地址空间中数据进行改动。此时，最好让进程在它自己的地址空间中运行，并且只让它访问父进程地址空间中的数据　这样就能够保护与手头正在执行的任 务无关的数据。在 Windows 系统中，进程之间共享数据的方法主要有动态数据交换（DDE）、OLE、管道、使 用剪贴板和内存文件映射等。 新的进程可以完全脱离父进程，同时它也可以像函数调用－样，只有当子进程执行完毕之后，父进程再继续执行。

3 纤程

纤程是一个必须由应用程序手动调度的执行单元。每一个线程可以调度多个纤程。一般来讲，纤程和设计得很好的多线程应用程序相比 并没有很大的优势。不过，使用纤程可以更容易地操作应用程序，而这些应用程序管理着自己的线程。从系统的观点来看 纤程对创建它的线程有认同感。换句话来说，如果纤程要访问线程本地存储器的话，它将访问创建它的那个线程的纤程的本地存储器。如果纤程调用了 ExaThread 函数，则创建纤程的线程将退出。不过，和线程相比纤程并不具有与其完全相同的状态信息。纤程维护的是一个堆栈和其中保存的纤程创建数据。纤程并没有抢先调度机制，用户可以在纤程之间切换。系统调度线程运行，如果正在运行的纤程的线程被抢先调度，则当前在线程中运行的纤程将被抢先调度。

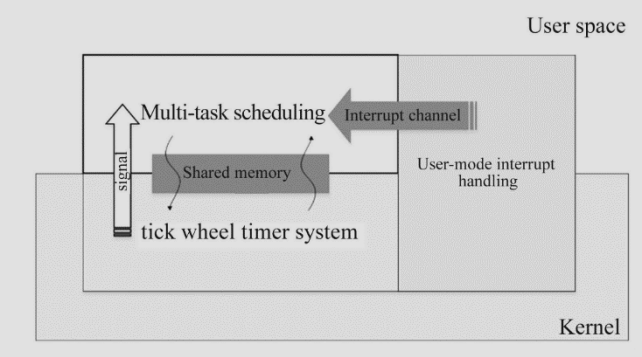
Linux 操作系统的多线程技术

1．操作系统线程技术现状

目前，许多流行的多任务操作系统都提供线程机制，按照系统管理策略线程可分为用户级线程和内核级线程两种基本类型。用户级线程不需要内核支持，可以在用户程序中实现线程调度、同步与互斥都需要用户程序自己完成．内核级线程需要内核参与由内核完成线程调度并提供相应的系统调用，用户程序可以通过这些接口函数对线程进行一定的控制和管理。相比之下，用户级线程具有更高的灵活性 它可以由用户程序创建并进行管理和控制以完成特殊的应用要求．例如多媒体实时过程、互连网数据下载、数据采集等。

2.Linux 操作系统多任务调度的实现框架 ULight 框架设计

如图 1 所示，ＵＬｉｇｈｔ实时多任务调度系统共包 括三个模块：多任务调度模块、定时器模块以及用户 态中断处理模块．多任务调度模块运行在用户态，定 时器模块运行在内核态，而用户态中断处理模块则 贯穿内核态和 用户 态．定时器模块为多任务调度模 块提供基本的定时服务，以支持任务的休眠操作，而 休眠操作为任务调度提供了抢占点；而且定时器模 块可以提供用于模拟时钟中断的定时信号，以支持 任务的分时调 度．多任务调度模块和定时器模块通 ６３６ 中国科学技术大学学报 第４７卷 过共享内存相互传递信息，而定时器模块会在有定 时器到期时向多任务调度模块发送指定信号．用户 态中断处理模块则通过搭建内核态和用户态之间的 中断处理快速通道，将中断传递给多任务调度模块， 并由多任务调度模块中的用户态线程中完成对中断 的最终处理。



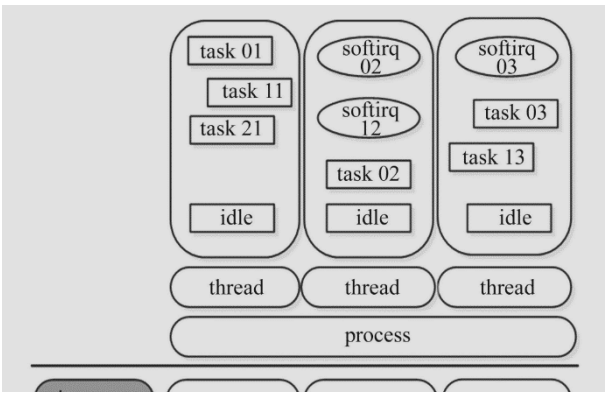
图一 ULight 框架设计示意图

多任务模块是在启动时创建一个用户态进程作为基础进程，然后根据处理器的核数 N 在该进程中创建(N-1)个线程，称为基础线程。这样做的目的是提升多核平台的资源利用率和系统运行性能。每个基础线程都会绑定在独立的处理器核上，以提升数据局部性和缓存命中率。然而，并非所有处理器核上都会绑定基础线程，因为定时器系统需要独占一个处理器核来执行其任务。

基础线程之间有全局的锁和核间信号量，而每个基础线程内部也有独立的核内信号量，用于控制任务对资源的互斥访问。每个任务都有独立的 ID，并且所有任务都在基础线程内运行。为了保证调度效率，创建后的任务不会被 ULight 动态迁移，但用户可以根据需要使用 ULight 提供的程序接口自行迁移任务。当基础线程内没有需要运行的任务时，CPU 会进入休息状态以节省功耗。

ULight 中的所有基础线程都被设置为 Linux 实时线程，以确保任务执行效率和对中断的响应速率。因此，ULight 启动时需要初始化的资源主要包括两个部分：基础进程和基础线程的创建与绑定，以及全局锁、核间信号量和核内信号量的初始化。

总结起来，ULight 的多任务模块在启动时创建一个基础进程作为用户态进程，并在其中创建多个基础线程，每个线程绑定在一个独立的处理器核上。基础线程之间通过全局锁和核间信号量进行通信，而核内信号量用于控制任务对资源的互斥访问。ULight 的设计旨在提高多核平台的资源利用率和系统性能，并通过设置实时线程来保证任务执行效率和对中断的响应速率。



图二 ULight 多任务运行架构

3.Linux 操作系统多任务处理环境下的定时器算法应用

3.1 定时器原理

在面向对象程序设计中，定时器是专门用于计时的一种控件。在设置了相应的时间间隔后，开始倒计时，当所设定的时间间隔一旦到达，系统便会自动触发定时器事件来完成相应的操作，同时，系统自动恢复所设置的时间间隔，开始下一轮的倒计时，直至定时器的使能属性设置为假或时间间隔被修改为 0 时为止。

3.2 算法的描述

3.2.1 算法的说明

该算法使用时间戳来区分多个定时任务的触发，并分为算法准备阶段和算法运行阶段两个阶段：

在算法准备阶段，首先设置一个整型全局变量来表示系统的统一时间标志，并将其初值设为数据类型所允许的最大值。然后，为定时器设置一个固定的时间间隔（通常为 1 秒），并启动定时器。

在算法运行阶段，涉及到添加定时器任务和处理定时器事件。在添加定时器任务时，将当前整型全局变量的值减去任务所需的时间间隔，作为任务的时间戳（到达时间）。每当定时器的时间间隔到达并触发定时器事件时，首先将整型全局变量减 1，然后检查所有定时器任务的时间戳。如果某个任务的时间戳与当前时间戳相同，则执行该任务。为了支持该算法，还需要设置两个全局数组来存储所有定时器任务的时间戳和任务描述。

简化的算法描述如下：

1.在算法准备阶段：

- 设置一个整型全局变量作为系统统一时间标志，初始值为最大值。
- 设置定时器的固定时间间隔，通常为 1 秒，启动定时器。

2.在算法运行阶段：

- 添加定时器任务：
- 将当前整型全局变量减去任务所需的时间间隔，作为任务的时间戳。
- 存储任务的时间戳和任务描述到全局数组中。

3.处理定时器事件：

- 当定时器的时间间隔到达时：
- 将整型全局变量减 1。
- 检查所有定时器任务的时间戳：
- 如果某个任务的时间戳与当前时间戳相同，则执行该任务。

该算法利用时间戳来标记定时任务的到达时间，并通过比较时间戳来确定要执行的任务。这种方法可以解决多定时任务的区分问题，提供了一种简单而有效的算法来处理定时器事件。

3.2.2 算法流程图

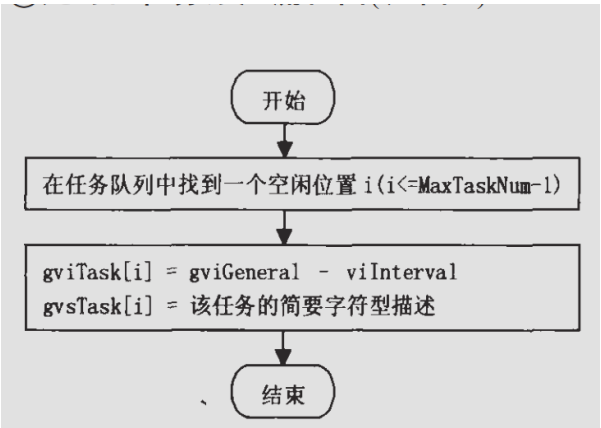
1. **变量说明**

●gviGeneral: 整型全局变量，初值可设为数据类 型允许的最大值，用于描述系统统一时间标志。

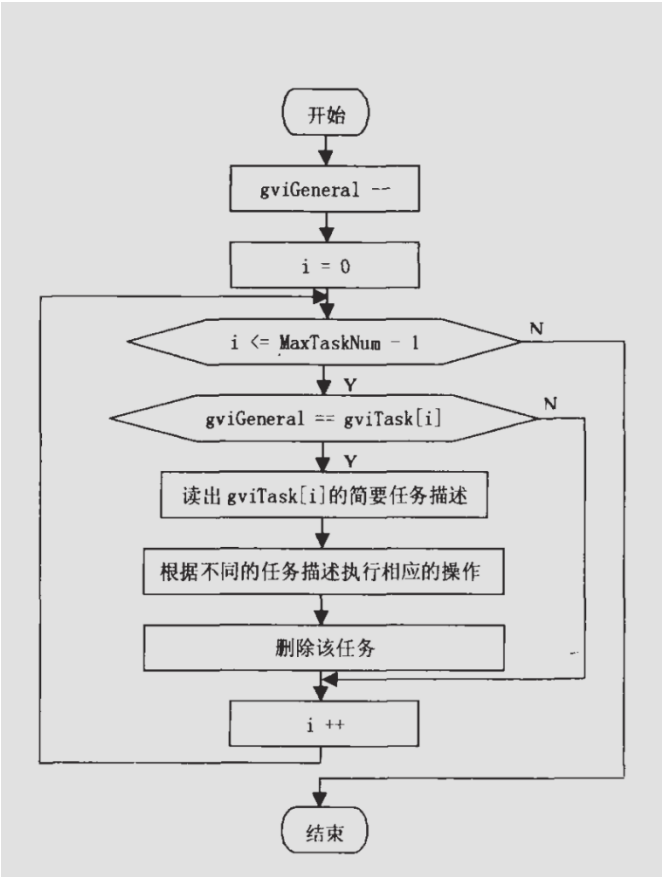
●gviTask[MaxTaskNum]: 整型全局数组，存储 各个定时器任务的“时间戳”的队列。

●gvsTask[MaxTaskNum]: 字符串全局数组，存 储各个定时器任务简要描述信息的队列。

●viInterval: 整型局部变量，存贮各个定时器任 务所需的时间间隔。



图三 定时器任务添加流程图



图四 定时器事件处理流程图

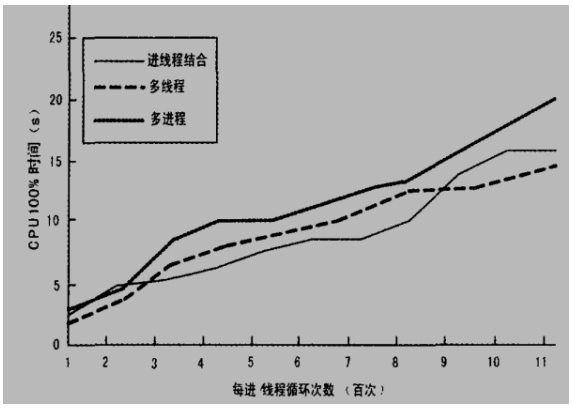
4.多线程与多进程的执行效率对比

测试代码如下：

```
FILE *logFile=NULL;
main()
{
    int i=0,j=0;
    logFile=fopen(TEST_LOGFILE,"a+");
    for(i=0;i<255;i++)
    {
        if(fork()==0)
        {
            for(j=0;j<10;j++){
                printf("hello world\n");
                fprintf(logfile,"Hello world\n");
            }
```

```
        exit(0);}
    }
    wait(0);
    return 0;
}
```

测试结果如图所示：



5.对比结论

采用多线程和多进程分别执行相同数据处理任务，其效率与完成任务量有着直接的关系，当完成的处理任务量较大的时候，多线程的优势不明显，而任务量较小的时，多线程要优于多线程。

对比项	多进程	多线程	结论
数据共享、同步	数据共享复杂，需要用IPC；数据是分开的，同步简单	数据共享简单，数据同步复杂	各有优势
系统资源	占用系统资源多，利用率低	占用系统资源少、利用率高	线程占优
创建、销毁与切换	程序复杂、速度慢	程序简单、速度快	线程占优
编程与调试	简单	复杂	进程占优
可靠性	多进程间不会互扰	如出现异常，会引起进程异常	进程占优

改进方法

根据上表的情况，针对在多线程与多进程的选择上可以得出以下结论：如果需要进行大量的数据计算优先使用多线程，进行消息收发与消息处理优先使用多线程，进行消息解码和业务优先使用多线程。

参考文献

[1]赵海廷.进程、线程和纤程[J].武汉工程职业技术学院学报,2004,(04):36-41.

[2]杨坤.多线程软件执行效率与改进方法研究[J].计算机与网络,2011,37(11):38-40.

[3]肖明魁.Python 语言多进程与多线程设计探究[J].计算机光盘软件与应用,2014,17(15):66-67.

[4]卢启衡,冯晓红.多任务处理环境下的定时器应用算法[J].现代计算机,2007,(01):100-102.

[5]张旭,顾乃杰,苏俊杰.一种 Linux 用户态实时多任务调度框架[J].中国科学技术大学报,2017,47(08):635-643.