计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(9):140−144 [doi: 10.15888/j.cnki.csa.005993]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

一种多文件任务调度算法①

汪　谦, 丁明波

(中国石油大学(华东) 计算与通信工程学院, 青岛 266580)

摘　要: 计算机在处理多文件任务的时候, 会出现同时读写文件的情况, 文件将会出现数据读写不全或数据缺失. 在 Linux 内核中, 单处理器情况下, 通过同步机制来进行任务的分配和处理, 其中经典的有原子操作, 信号量机制, 互斥锁等实现方案. 在多处理器系统中则是通过 test-and-set 原语操作来实现. 本文通过设计一种多文件任务调度的算法, 避免整个系统发生互斥访问. 本文通过 Matlab 编程实现该算法, 其结果表明本文提出的多文件调度算法能够有效的并行执行多文件任务.

关键词: 多任务; 任务调度; 并行计算; 文件锁

引用格式: 汪谦,丁明波.一种多文件任务调度算法.计算机系统应用,2017,26(9):140–144. http://www.c-s-a.org.cn/1003-3254/5993.html

Algorithm for Scheduling Multi-Task

WANG Qian, DING Ming-Bo

(School of Computer & Communication, China University of Petroleum, Qingdao 266580, China)

Abstract: When the computer is handling multi-file tasks, it may read and write a file at the same time, resulting in the failure of the file data to be fully read and written or in the loss of some data. In the Linux kernel, with the single processor, task allocation and processing is made with the synchronization mechanism. The classic approach is atomic operations, semaphore mechanisms, mutexes, etc. In the multi-processor systems, the test-and-set primitive operation is made to solve the problem. In this paper, we design a new task scheduling scheme for multitasking to avoid mutual exclusion access. We use a Matlab program to realize the algorithm, and the result shows that the algorithm can effectively realize the multi-file tasks parallel execution.

Key words: multi-file; task scheduling; concurrent computation; file lock

随着计算机行业的蓬勃发展, 计算机运算速度也逐渐遇到了一个瓶颈. 计算机的处理器发展趋势为多核处理器, 操作系统系统则向分布式系统发展, 为了提高计算机整体的运算速度和提高资源其利用率, 系统通过动态分配不同计算机中的多个通用的物理和逻辑资源来实现系统调度任务, 从而提高计算机运算速度.

计算机在执行运算的时候, 操作系统首先对任务进行调度, 在任务的处理过程中如果出现多个任务同时访问计算机的临界资源, 会导致计算机发生死锁, 现在的计算机在多核或分布式操作系统的情况下, 更容易出现多个任务同时访问计算机的临界资源的问题. 针对该问题, 国内外许多优秀的学者都提出了一些高性能的锁机制. ML Scott 等在 2000 年提出了一种 MCS Spinlock[1-3], 该自旋锁通过链表高效的解决了自旋锁的可扩展问题, 保证了自旋锁的公平性. Choi S 等在 2010 年提出了一种基于分布式管理器的主机锁机制[4,5], 该锁机制支持集群中的多个客户端共享磁盘. 该锁机制还有一个显著的特点, 随着锁的请求速度的增加, 该锁机制的优势越明显, 它很好地提高了处理速度与利用率. 上述方法虽然能够实现对文件进行加锁和

① 基金项目: 中国石油大学(华东)研究生创新工程资助项目(CX2013028); 中央高校基本科研业务类专项基金(14CX02032A)
收稿时间: 2016-12-21; 采用时间: 2017-02-17

共享文件, 但在对共享文件进行处理的时候并不能保证集群机器对共享文件不被多台机器同时读写, 同时集群机器有较高的利用率.

本文的系统主要对文件进行处理, 在对文件进行操作的过程中, 多个任务或者多个处理器很容易同时对某个文件进行读写. 但是一个任务如果对正在被其他任务读取的文件进行读写操作, 可能会出现读操作的任务读取到一些不完整的或者已经被破坏的数据.

本文系统设计如下, 主计算机(本文中称为 Master 机器)产生需要计算的文件, 多台从计算机(本文中称为 Slave 机器)来获取 Master 机器产生的文件, 计算该文件并将结果保存.

Slave 机器需要扫描 Master 是否生成新的文件, 如果有新的文件产生, 需要对文件进行判断是否有其他 Slave 机器正在执行该文件, 如果产生新文件之后没有其他 Slave 机器执行该任务, 则将该文件分配给 Slave 机器, 其他情况下该 Slave 进入忙等待状态. 在查看文件有没有被其他 Slave 机器执行的时候, 需要对文件进行加文件锁[6-8], 告知其他机器该文件正在被读写. 整个系统需要能够形成一个流水线执行操作, 保证每台机器能够保证高效的执行, 缩短整个系统处理文件的时间.

本系统中, 需要实现文件锁机制, 但是简单的文件锁, 不能够保证所有的文件在读写的过程中不会同时被多个 Slave 机器读写, 导致文件数据不全或者文件处理出错, 整个系统的任务处理出现问题.

本文设计并实现了一种新型的多文件任务调度的算法, 解决了上述文件在读写过程中, 同时被同台 Slave 读写的问题.

1 系统简介

Master 机器在生成一个新的文件之后, 如果文件较多或文件运算时间很长的情况下, 整个执行过程将会耗费很长时间. 本文假设, 如果将 Master 机器产生的文件, 通过网络连接或者共享资源的方式, 分配给多个 Slave 机器, Slave 机器并行的计算文件任务, 整个过程在不同的机器上, 可以减少大量的计算时间. 此时 Master 机器只需要产生文件和对文件任务进行调度, 不需要执行文件计算任务, Slave 则对任务进行计算, Master 和 Slave 之间相对独立的运行, 大量减少 Master 机器对文件生成和运行的时间. 整个系统的总体构造图如图 1 所示.

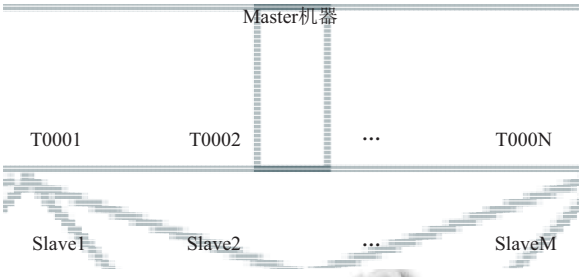

图 1 系统总体构造图

当出现多台 Slave 机器和 Master 机器在同时执行任务的情况下, 整个系统将会是一个大的集群, 对系统的鲁棒性需要很强大的要求, 需要有一个对文件任务分配比较稳定的调度算法来避免整个系统分配和执行文件任务过程中产生同时读写文件的问题.

系统要求 Master 机器生成文件和 Slave 机器执行任务相对独立, 保证 Master 机器和 Slave 机器在运行较多文件任务的情况下能够正常的执行下去.

2 常用锁机制简介

2.1 文件锁机制

文件锁是 Linux 2.6 内核及其之后的版本提出的概念[9], 早期的 Linux 版本只支持对整个文件进行加锁, 因此不能运行数据库等对多文件处理要求较高的程序. 在文件进行操作的时候, 对文件进行加文件锁, 可以保证当前文件在执行的过程中其他文件不能对该文件进行读写.

Linux 支持的文件锁主要包括劝告锁和强制锁:

劝告锁是一种类似生产者和消费者工作机制的锁, 内核对文件提供加锁机制并对文件进行检测是否已经加锁, 但是内核对文件锁不进行控制和协调. 劝告锁机制不能防止进程对文件的访问, 只能通过各个进程在对文件读写时候, 检查其他进程是否已经对该文件加锁.

强制锁是一种采取强制作用的文件锁, 内核对文件进行强制加锁, 当对文件进行读写操作时候, 内核都要检查该文件是否被加锁和其他进程调用时候会不会违反其强制锁的约束. 如果文件被加上了读写锁, 其他进程对这个文件进行读写的时候就会被阻止.

文件锁机制在 Linux 2.6 之后的内核中使用, 其实现的过程较为复杂, 在 Windows 或其他的操作系统中

不能直接使用. 本系统的算法借鉴其强制锁的实现思路, 来完成系统的设计.

2.2 Test_and_set 锁机制

Test_and_set[9]是一种不可中断的原语操作, 是特定的汇编指令, 用来交换两个内存某一单元的值, 将新值写入内存特定的位置并传回其旧值. 多个进程存取内存的情况下, 如果一个进程正在执行 test_and_set, 在它执行完成前, 其他的进程不可以执行 test_and_set.

下面代码展示的是使用 test_and_set 来实现锁机制.

```
function lock(lock) {
  while(test_and_set(lock) == 1);
}
function unlock(lock) {
  lock = 0;
}
```

1) 程序读取 lock, 如果 lock=0, 设置 lock=1, 程序加锁, 读取临界区资源.

2) 如果 lock=1, 直接返回 1, 继续进行忙等待状态.

Test_and_set 在执行的时候, 需要硬件配合完成, 系统需要较大资源开销来保证内存、缓存、以及寄存器等硬件之间数据一致. 同时, test_and_set 不能保证任务按照 FIFO 顺序获取锁, 特殊情况下, 部分程序需要很长时间才能获得锁.

2.3 MCS Spinlock 锁机制

MCS Spinlock 是基于链表的高性能、可扩展的自旋锁. 如图 2 所示, 将全体锁申请者的信息构成一个单向链表, 锁申请者在使用前必须分配一个 mcs_lock_node 结构体, mcs_lock 是一个指向最后一个申请者的 mcs_lock_node 结构的指针, 并且当前进程锁未被使用.

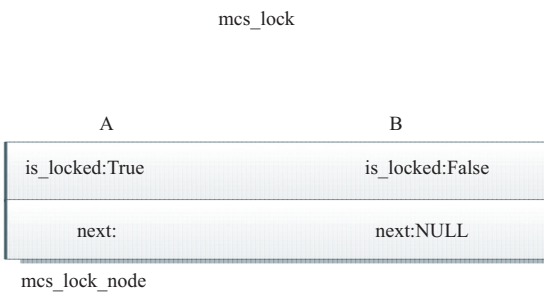

图 2 mcs_lock 结构图

mcs_lock 的结构体的伪代码为:

```
struct mcs_lock_node {
  mcs_lock_node next;
  int is_locked;
}
strcut mcs_lock {
  mcs_lock_node queue;
}
```

每个处理器都代表着一个 msc_lock_node, 当某个 mcs_lock_node 需要执行时:

1) 将该 mcs_lock_node 加入 mcs_lock 队列;

2) 如果当前队列还有其他的 node 在等待, 则设置 is_locked=1, 进入忙状态, 等待其他 node 的执行;

3) 当等待队列执行到该 mcs_lock_node 的时候, 唤醒该 mcs_lock_node, 并设置 is_locked=0, 执行 mcs_lock_node 的操作;

4) 新添加的或者任务执行结束的处理器需要继续执行任务, 则按照 1, 2, 3 步骤来继续执行.

MCS spinlock 在多线程任务的系统中能够实现较好的性能, 本文实现的多任务调度算法, 则通过改进的 MCS Spinlock 来实现的.

2.4 本系统的多任务调度算法

为了防止文件任务在处理的时候被多个 Slave 同时进行读写, 从而产生文件读写错误的情况. 本系统将 Slave 执行一个文件任务的过程设计为闭环模式, 在 Slave 执行某个文件任务的时候, 该文件任务不能被其他 Slave 读写, 保证该文件执行的时候处于加锁状态, Slave 执行完该文件之后才可以执行其他文件任务, 保证整个处理过程的完整性. 文件处理过程中, 需要两个队列, Master 机器生成的任务文件为 file_task_queue 队列, Slave 机器在计算为 slave_queue 队列, 两个队列结构体伪代码如下:

```
struct file_task_queue {
  int file_id;
  int current_file_queue;
}
struct slave_queue {
  string slave_name;
  bool is_idle;
}
```

这里 file_id 表示当前 Master 生成的文件任务的 id 编号, current_file_queue 表示已经执行完文件的编

号, slave_name 表示 Slave 机器的主机名(本文中称为 S01, S02 …), is_idle 表示当前 Slave 是否空闲.

在队列首部的 Slave 的 is_idle 为 False, 表示该 Slave 机器不是空闲, 队列后续的 slave_queue 的 is_idle 都为 True, 表示空闲, 可以接受分配任务.

系统的设计的两个队列的效果图, 如图 3, 系统对 Slave 和任务两个队列进行调度执行效果, 如图 4.

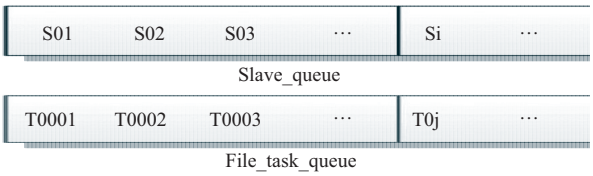

图 3 系统队列图

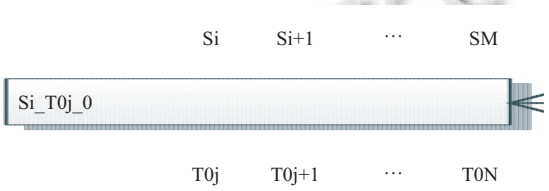

图 4 队列执行效果图

通过获取 Slave_queue 和 File_task_queue 两个队列的头部数据, 生成一个标识文件 Si_T0j_0, 这时候告知 T0j(表示任务编号)任务已经被分配, 同时 Si(Slave 机器编号)机器 is_idle 为 False, 该 Slave 机器不是空闲状态.

整个算法的过程如图 5 所示.

1) Slave 开始运行之后就在 Slave_queue 队列中添加 S0N 标记, 表明该 Slave 处于空闲状态;

2) Master 从 Slave_queue 队列中获知 S0N 空闲, File_task_queue 队列中的 file_id(这里称为 M), 给 S0N 分配 T000M 文件, 生成标记文件 S0N_T000M_0 标志文件(其中_0 表示文件未被执行, _1 表示文件已经被执行), 在 Slave_queue 中删除 SN 标志;

3) Slaver0N 扫描文件得到 S0N_T000M_0, 执行 T000M 文件, 当文件执行完之后, 继续在 Slave_queue 中添加 S0N 节点, 同时 File_task_queue 中的 file_id 加 1 操作, 将标记文件 S0N_T000M_0 删除, 并生成标记文件 S0N_T000M_1, 方便系统以后检查文件执行过程;

4) 当文件任务被 Slave 执行完之后, 继续重复 1, 2, 3 步骤执行, 系统高效地对文件任务进行调度和执行.

根据上述算法的介绍, Salve 机器处理一个任务的过程是一个闭环的处理过程, 其执行的流程图如图 6, 处理过程如下:

1) 从 Slave 空闲队列里面找到某个空闲的 Slave 机器 Slave N, 对该 Slave N 加标记 S0N, 告知其处于空闲状态, 能够被分配文件任务.

2) Master 机器通过识别 Slave 队列, 获取队列队首 S0N 标志, 从而知道 Slave N 为空闲, 则在任务的队列中队首选择需要执行的任务 Task M, 将 Slave 和需要执行的文件设置成 S0N_T0M_0 标志, 并且删掉 S0N 标志, 此时 Slave N 处于忙状态, 同时 Task M 处于加锁状态只能被 Slave N 进行读取.

3) 当 Task M 执行完之后, Master 机器获知对标志 S0N_T0M_1, 这时 Slave N 已经执行完一个文件任务, Slave N 恢复空闲状态.

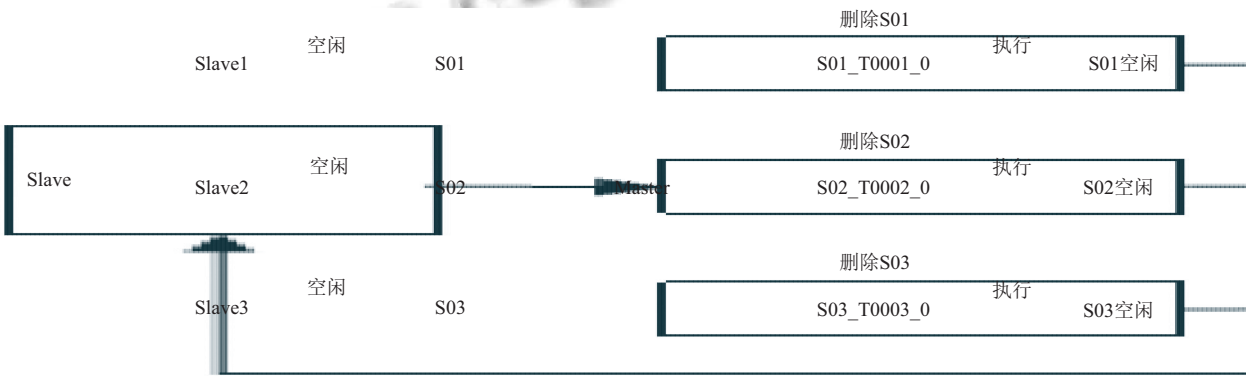

图 5 算法过程图

Slave 在完成一个文件任务的处理之后, 转而进入下一个文件任务, 每一个任务的执行都是一个完整的闭环的过程. 当系统中存在着多个 Slave 机器和多个任务的时候, 整个系统将会处于高效且并行的状态运行.

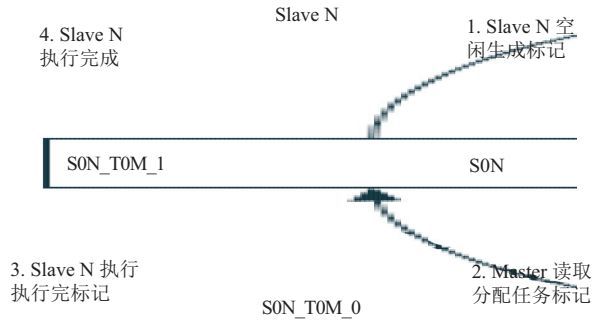

图 6 Slave 执行效果图

3 实验部分

本文实验在 Windows 环境中, 通过 Matlab 编程环境实现. 使用的是 4 台 Windows 7 64 位电脑, 其中的一台电脑作为 Master 机器来生成任务, 剩下的三台电脑作为 Slave 机器来对 Master 机器生成的任务进行监控, 解析和计算. Master 机器上运行着 Master 程序, Slave 机器上则运行着 Slave 程序.

整个系统运行在一个共享文件夹中, 对任务文件的生成, 读写都在该共享文件夹中运行.

Master 机器在分别产生 100, 500, 1000 个小文件任务(Slave 处理文件的时间小于 1S)的时候, 各个 Slave 机器所执行的任务个数, 如表 1 所示.

表 1 Slave 执行小文件效果

任务个数	Slave 1 执行个数	Slave 2 执行个数	Slave 3 执行个数
100	36	33	31
500	179	165	156
1000	350	329	321

当 Master 机器分别产生 100、500、1000 个大文件任务(Slave 处理文件的时间大于 10S)的时候, 各个 Slave 及其所执行的任务个数, 如表 2 所示.

表 2 Slave 及其执行大文件效果

任务个数	Slave 1 执行个数	Slave 2 执行个数	Slave 3 执行个数
100	35	33	32
500	171	167	162
1000	343	329	328

结果显示, 整个系统能够较好的执行 Master 产生的文件任务, Slave 执行文件的数量基本相等, 调度程序在对文件处理的时候具有较好的性能. 同时, 每个文件任务在处理的时候都是相对独立, 没有出现多个 Slave 读写同一个文件的情况.

系统设计的多任务调度算法, 在 Windows 或其他操作系统下也能够比较容易编程实现, 不需要添加其他硬件支持, 仅仅需要设置网络接口或者机器在局域网的情况下, 使用共享文件夹的方式就可以执行.

4 结语

本文提出的算法实现简单, 能更方便地处理多文件任务并行调度和执行的问题, Slave 机器在一个闭环操作之后, 立即进入下一个闭环操作, 整个过程具有较强的鲁棒性. 系统在保证整个运行环境稳定的条件下, 能够较好的处理文件任务, 并对集群中机器有着较高的使用率, 同时大幅度降低使用一个机器来执行整个计算过程需要的时间.

参考文献

1 Mellor-Crummey JM, Scott ML. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. on Computer Systems, 1991, 9(1): 21–65. [doi: 10.1145/103727.103729]
2 付智杰, 周群彪. MCS spinlock 的 Linux 内核模块实现. 微计算机应用, 2009, 30(7): 55–59.
3 Peng ZW, Xu XA. The analysis of Linux kernel spinning lock on SMP. Journal of Jiangxi Institute of Education (Comprehensive), 2005, 26(3): 23–25, 28.
4 Snaman W, Thiel D. The VAX/VMS distributed lock manager. Digital Technical Journal, 1987.
5 Choi S, Choi M, Lee C, *et al*. Distributed lock manager for distributed file system in shared-disk environment. Proc. of the 10th International Conference on Computer and Information Technology (CIT). Bradford, UK. 2010. 2706–2713.
6 周超, 刘云朋. Linux 文件锁技术的分析与实现. 电脑开发与应用, 2009, 22(4): 43–44, 51.
7 博韦, 西斯特. 深入理解 LINUX 内核. 陈莉君，张琼声，张宏伟, 译. 3 版. 北京: 中国电力出版社, 2007.
8 陈莉君. Linux 操作系统内核分析. 北京: 人民邮电出版社, 2000.
9 Zhou C, Liu Y P. Analysis and realization of file lock in Linux. Computer Development & Applications, 2009, 22(4): 43–44, 51.
10 Alistarh D, Attiya H, Gilbert S, *et al*. Fast randomized test-and-set and renaming. Proc. of the 24th International Conference on Distributed Computing Cambridge, MA, USA. 2010. 94–108.