Multicore & Synchronization Overview

Department of Computer Science & Technology
Tsinghua University

Contents

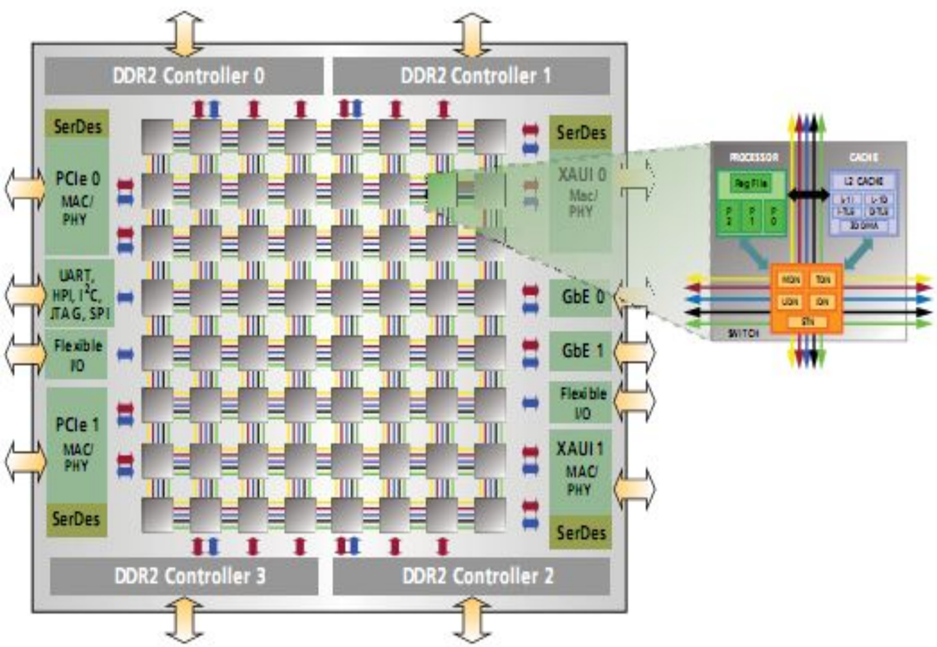
- Multicore Architecture
- Linux Synch Primitives
- Lock-awared scheduling

多核挑战

内存访问延时

设计复杂度　**多核处理器**　功耗

晶体管数目　　时钟频率



工业界

- 普遍采用
- Intel/AMD/IBM/Sun/…

涉及领域

- 无处不在
- 服务器/PC/笔记本/嵌入式系统/…

多核挑战

多核(CMP) V.S. 对称多处理器(SMP)

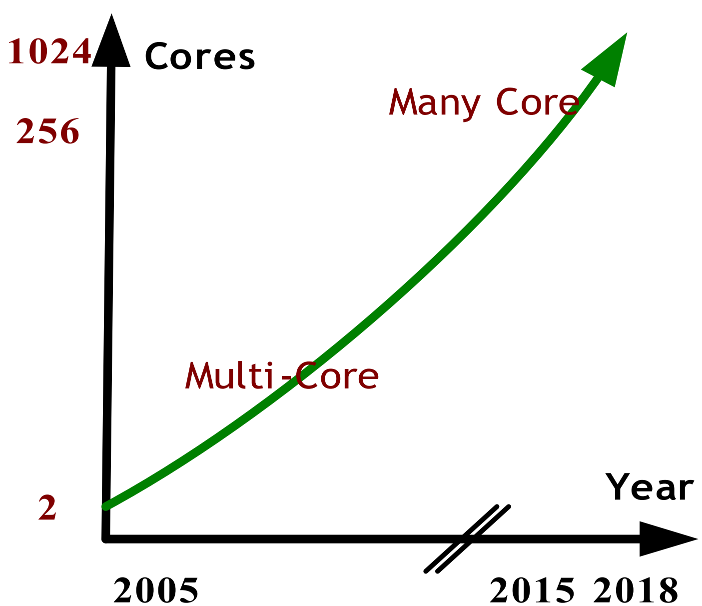
核的数目更多

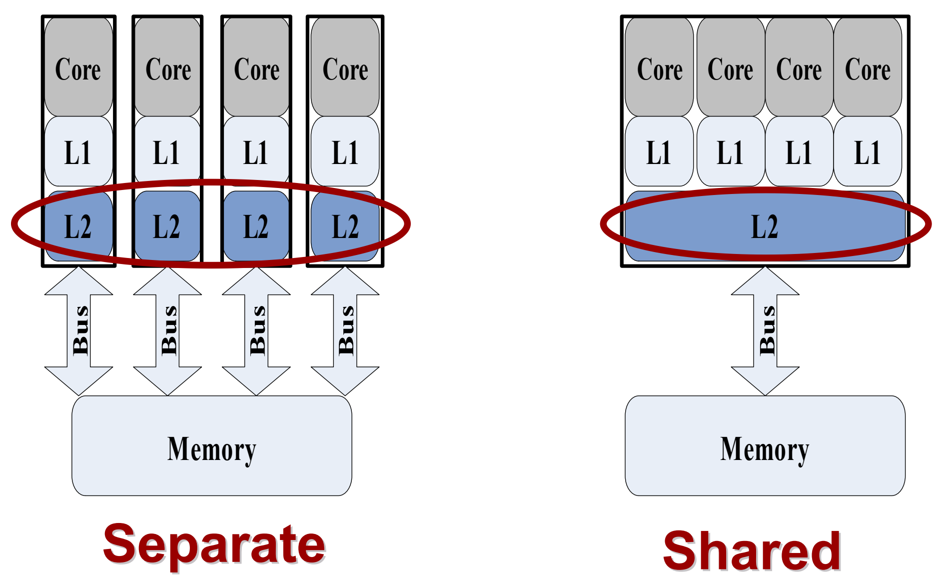
SMP: 低端(2CPU), 中级(4~8CPU),高端(>16CPU)

CMP: 4~8 cores 多核系统, **1000+ cores (<10年)**

E.g. Intel's 80 cores chip & Tilera's 64 core chip

硬件资源共享(e.g., 最后级缓存)





针对系统服务接口的分析和比较

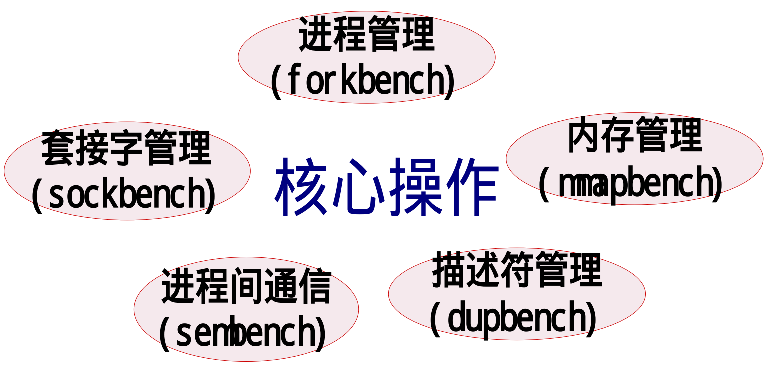
问题: 操作系统在多核平台上的可扩展性瓶颈是什么 如何产生

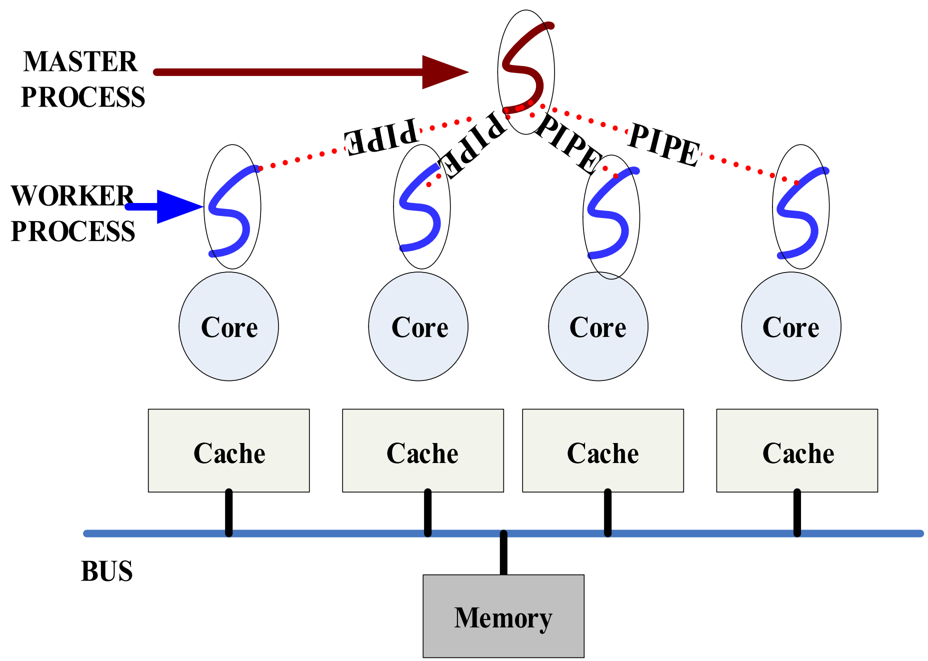
解决方案: 针对系统服务接口的操作系统可扩展性分析和比较

针对系统服务接口的分析和比较

测试程序集

核心、重要操作 由统一框架管理





针对系统服务接口的分析和比较

操作系统

- Linux
- OpenSolaris
- FreeBSD

硬件平台

- AMD NUMA 系统 8 Opteron 4 核 = 32核心

分析工具

- Linux: Oprofile /proc/lock_stat
- Solaris: Dtrace lockstat
- FreeBSD: lock profiling

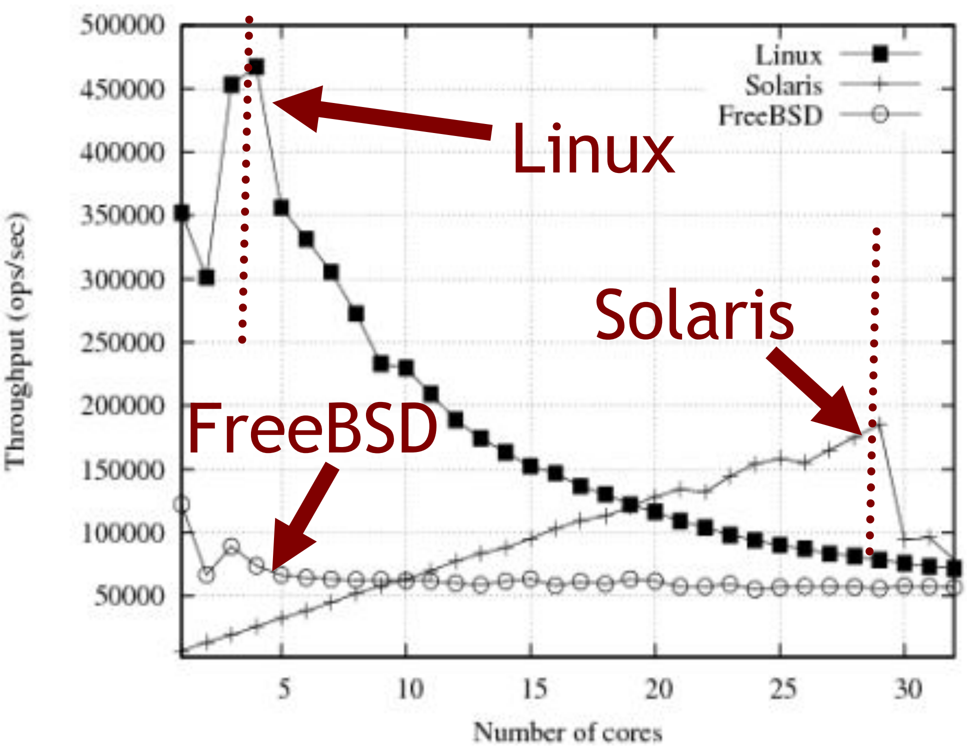
绑定接口

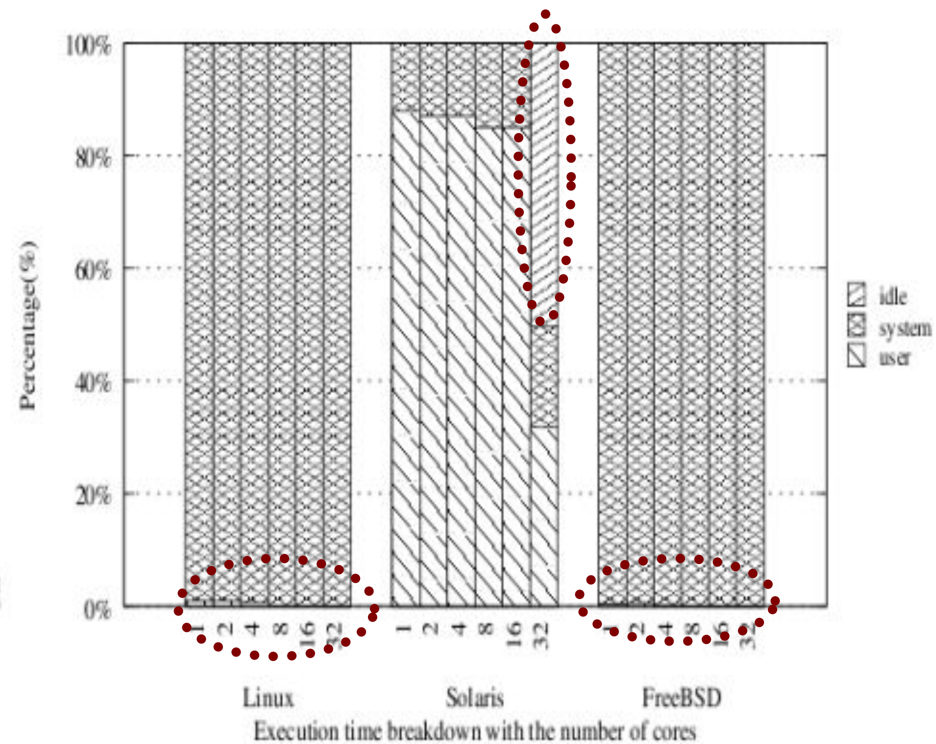
- Linux sched_setaffinity()
- Solaris pset_bind()
- FreeBSD cpuset_setaffinity()

针对系统服务接口的分析和比较

mmapbench

每个进程映不断射同一文件的500Mbytes 读取每一页的第一字节 解除映射





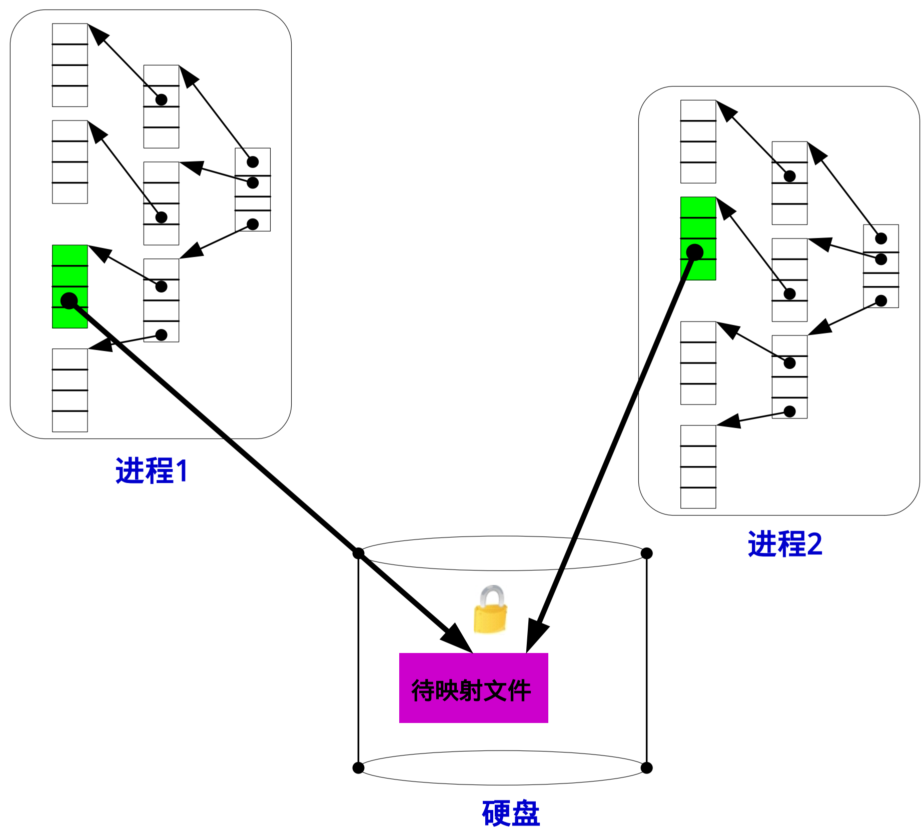
所有系统均有可扩展性问题

用户态↓ 空闲时间增加↑

针对系统服务接口的分析和比较

瓶颈

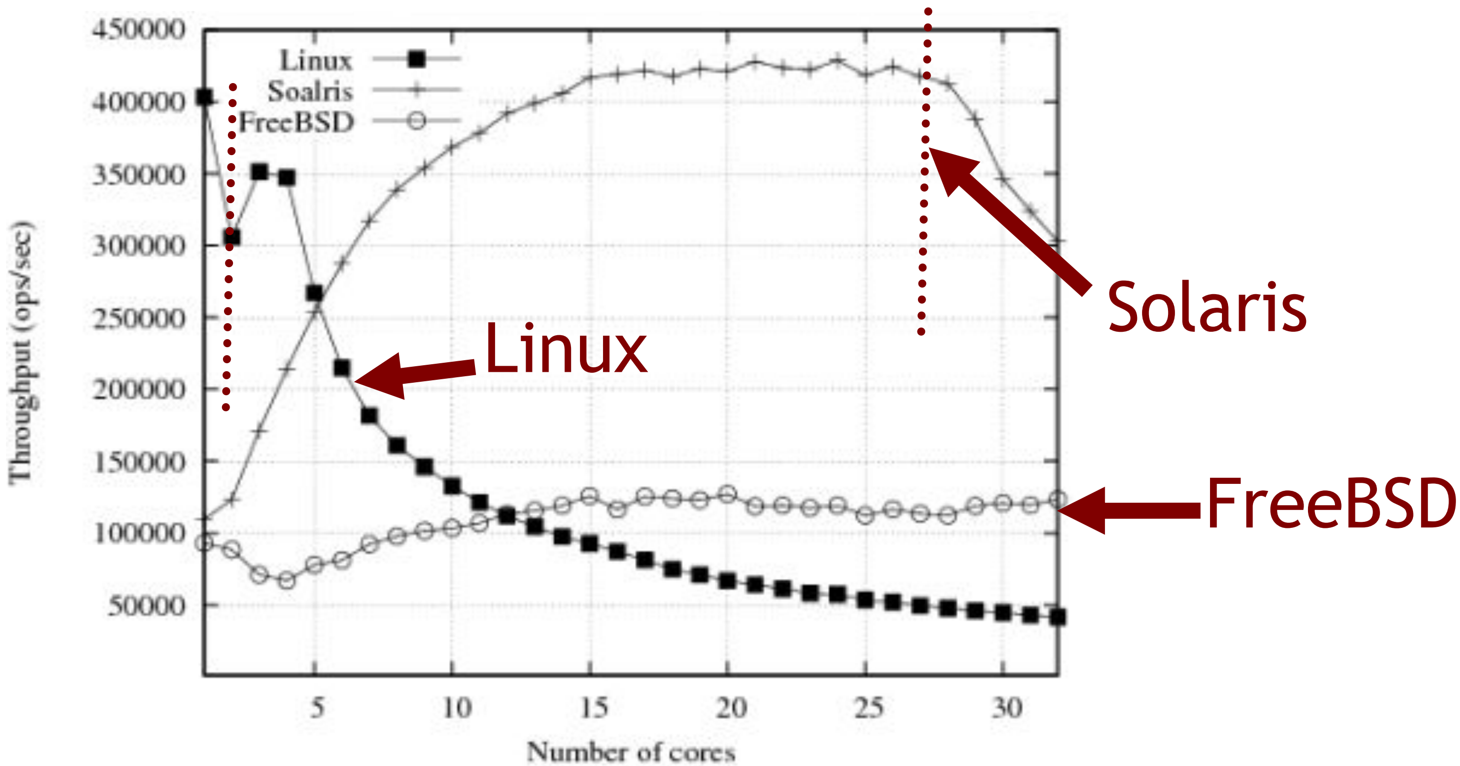
- 保护相同文件的锁竞争
- 锁获取时机不同
  - Linux 整个映射
  - Solaris 统计数据更新
  - FreeBSD 映射策略



针对系统服务接口的分析和比较

sockbench

- 每个进程不断调用socket()和close()

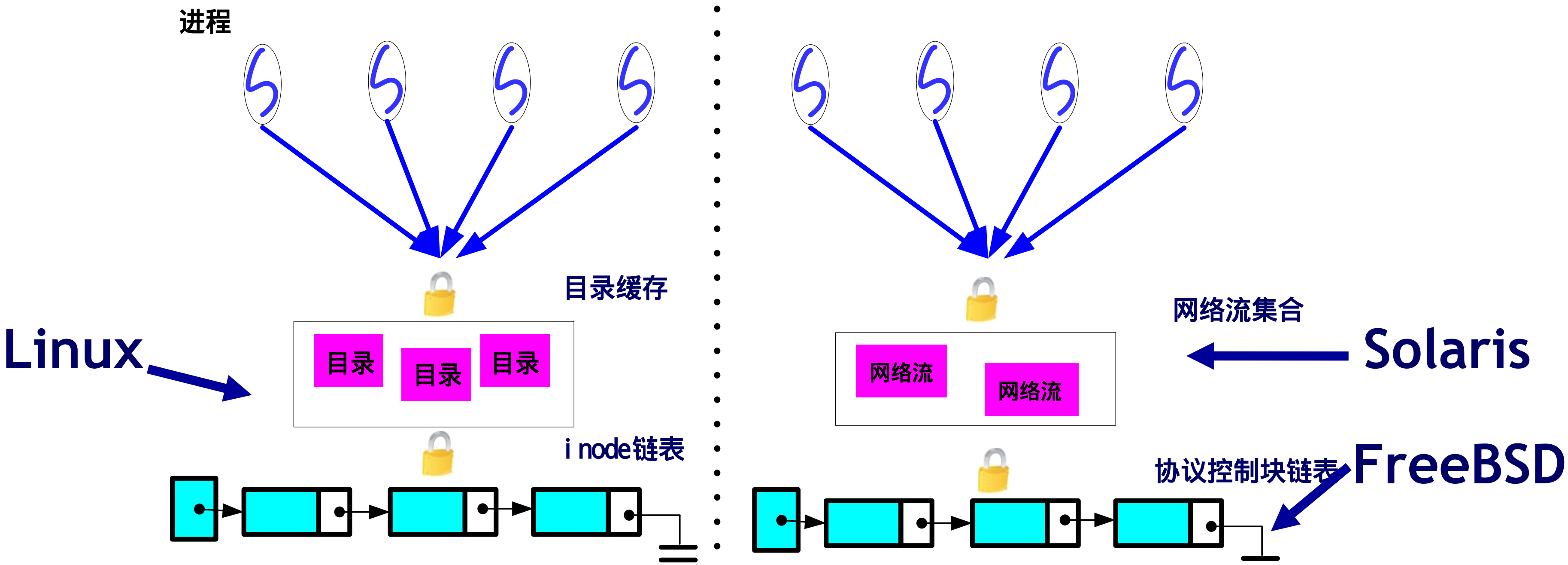


所有系统都有可扩展性问题

针对系统服务接口的分析和比较

瓶颈

- 目录缓存锁竞争 inode链表锁竞争(linux)
- 网络协议栈竞争
  - Solaris 引用计数更新
  - FreeBSD 协议控制块链表维护



瓶颈概要

	Linux	Solaris	FreeBSD
forkbench	建立、删除VMA导致保护内存映射文件的锁竞争	缺页异常在内存映射文件的读写锁竞争	缺页异常在内存映射文件的互斥锁竞争
mmapbench	建立、删除VMA导致保护内存映射文件的锁竞争	设置内存放置策略导致内存映射文件读写锁竞争	更新和查找vnode信息在内存映射文件互斥锁竞争
dupbench	完全可扩展	关闭文件描述符在哈希表的自适应互斥锁上竞争	witness开销随着核数线性增加 (去掉则完全可扩展)
sembench	保护全局信号量的读锁有竞争	完全可扩展	完全可扩展
sockbench	全局目录缓存和全局inode链表的自旋锁竞争	建立和删除流导致网络协议栈的引用计数竞争	保护全局的协议控制块链表的读写锁竞争

1. 操作系统中保护共享数据结构的同步原语是影响可扩展性的重要因素
2. 锁竞争可能导致可扩展性随着核数的增加而下降(锁颠簸现象)

Contents

- Multicore Architecture
- Linux Synch Primitives
- Lock-awared scheduling

Linux Synch Primitives

Various types of synchronization techniques used by the kernel

Technique	Description	Scope
Per-CPU variables	Duplicate a data structure among the CPUs	All CPUs
Atomic operation	Atomic read-modify-write instruction to a counter	All CPUs
Memory barrier	Avoid instruction reordering	Local CPU or All CPUs
Spin lock	Lock with busy wait	All CPUs
Semaphore	Lock with blocking wait (sleep)	All CPUs
Seqlocks	Lock based on an access counter	All CPUs
Local interrupt disabling	Forbid interrupt handling on a single CPU	Local CPU
Local softirq disabling	Forbid deferrable function handling on a single CPU	Local CPU
Read-copy-update (RCU)	Lock-free access to shared data structures through pointers	All CPUs

Linux Synch Primitives

- Atomic operations
  - memory bus lock, read-modify-write ops
- Memory barriers
  - avoids compiler, cpu instruction re-ordering
- Interrupt/softirq disabling
  - Local, global
- Spin locks
  - only on SMP systems; keep them short!
  - general, read/write, big reader
- Semaphores
  - general, read/write

Atomic operators

- Simplest synchronization primitives
  – Primitive operations that are indivisible
- Two types
  – methods that operate on integers
  – methods that operate on bits
- Implementation
  – Assembly language sequences that use the atomic read-modify-write instructions of the underlying CPU architecture
- How do these help?

Atomic integer operators

```
atomic_t v;
atomic_set(&v, 5);          /* v = 5 (atomically) */
atomic_add(3, &v);          /* v = v + 3 (atomically) */
atomic_dec(&v);             /* v = v - 1 (atomically) */

printf("This will print 7: %d\n", atomic_read(&v));
```

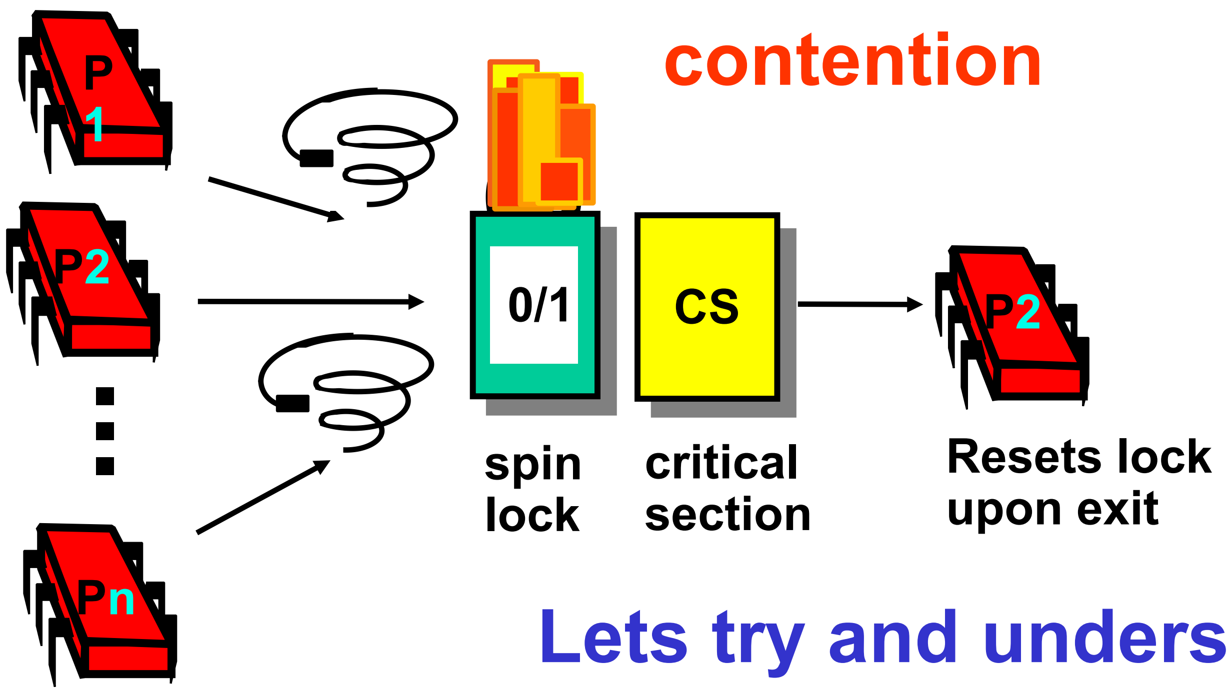
Beware:

- Can only pass atomic_t to an atomic operator
- atomic_add(3,&v); and
{
atomic_add(1,&v);
atomic_add1(2,&v);
}
are not the same! … Why?

Atomic Operations

- Many instructions not atomic in hw (smp)
  - rmw instructions: inc, test-and-set, swap
  - unaligned memory access
  - rep instructions
- Compiler may not generate atomic code
  - even i++ is not necessarily atomic!
- Linux – atomic_ macros
  - atomic_t – 24 bit atomic counters
- Intel implementation
  - lock prefix byte 0xf0 – locks memory bus

Basic Spin-Lock



…lock suffers from contention

Lets try and understand this phenomena

Review: Test-and-Set

```
public class RMW extends Register {
  int value;

  public synchronized int TAS() {
    int result = value;
    value = 1;
    return result;
  }
}
```

remember old value

new value is 1

return old value

Spin Locks

- busy wait – meaningless on UP
- spinlock_t struct
  - int lock field – 1 for unlock, <= 0 locked
- macros
  - spin_lock_init(),
  - spin_lock(),
  - spin_unlock(),
  - spin_unlock_wait(),
  - spin_is_locked(),
  - spin_try_lock()

spin_lock

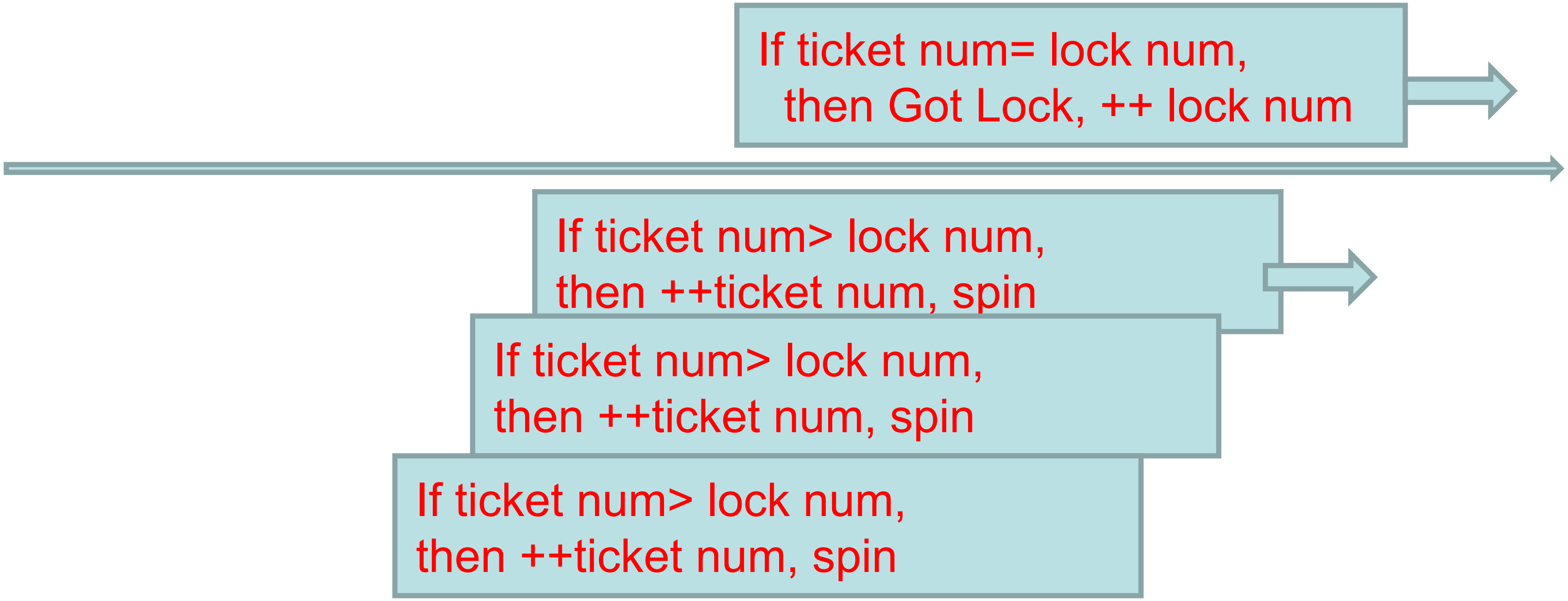
```
1: lock; decb slp # atomically decrement
   jns   3f       # if clear sign bit jump forward to 3
2: cmpb $0,slp    # spin – compare to 0
   pause          # spin – wait
   jle 2b         # spin – go back to 2 if <= 0 (locked)
   jmp 1b         # unlocked; go back to 1 to try to lock again
3:                # we have acquired the lock …
```

Notes

- The spin loop is actually assembled into a separate code section so that the straight line case falls through without fetching spin code.
- spin_unlock merely writes 1 into the lock field.

Spin lock

- Ticket Spin Lock
  - 2.6.26+



Read/Write Spin Locks

- allow multiple readers, single writer
- rwlock_t – really two “fields”
  - high order 8 bits:1 no writer, 0 otherwise
  - low order 24 bits: # of readers (2’s comp)
    - available: 0x01000000
    - one writer: 0x00000000
    - one reader: 0x00ffffff
- implementation: similar to basic spinlock
- reader priority

The Big Reader Lock

- reader optimized r/w spinlock
- r/w spinlock suffers cache contention
  - on lock and unlock because of write to rwlock_t
- per-CPU, cache-aligned lock arrays
  - one for reader portion, another for writer portion
- to read: set bit in reader array, spin on writer
  - acquire when writer lock free; very fast!
- to write: set bit and scan ALL reader bits
  - acquire when reader bits all free; very slow!

Semaphores

- sleep lock; general counting semaphores
- struct semaphore
  - count (atomic_t):
    - >0 free; 0 inuse, no waiters; <0 inuse, waiters
  - wait: wait queue
  - sleepers: 0 (none), 1 (some), occasionally 2
- implementation requires lower-level synch!
  - atomic updates, spinlock, interrupt disabling
- optimized assembly code for normal case (down())
  - C code for slower "contended" case (__down())

Semaphores

- up() is easy
  - atomically increment; wake_up() if necessary
- uncontended down() is easy
  - atomically decrement; continue
- contended down() is really complex!
  - basically increment sleepers and sleep
  - loop because of potentially concurrent ups/downs
- still in down() path when lock is acquired

- A fast binary semaphore: mutex
  - How about adaptive mutex?

Memory Barriers

- Compilers and hw re-order memory accesses
  - as an (unobservable?) optimization
  - true on SMP and even UP systems!
- Memory barrier – instruction to hw/compiler to complete all pending accesses before issuing more
  - read memory barrier – acts on read requests
  - write memory barrier – acts on write requests
- Linux macros – mb(), rmb(), wmb(), smp versions
- Intel –
  - certain instructions act as barriers: lock, iret, control regs
  - rmb – asm volatile("lock;addl $0,0(%%esp)":::"memory")
    - add 0 to top of stack with lock prefix
  - wmb – Intel never re-orders writes, just for compiler

Completions

- slightly higher-level, FIFO semaphores
  - solves a subtle synch problem on SMP
- up/down may execute concurrently
  - this is a good thing (when possible)
- ops: complete(), wait_for_complete()
  - spinlock and wait_queue
  - spinlock serializes ops
  - wait_queue enforces FIFO

Local Interrupt Disabling

- basic primitive in original UNIX
- doesn't protect against other CPUs
- Intel: "interrupts enabled bit"
  - cli to clear (disable), sti to set (enable)
- enabling is often wrong; need to restore
  - local_irq_save()
  - local_irq_restore()
- SPARC is weird: interrupt flag part of register window; must restore in same context

Disabling Deferred Functions

- disabling interrupts disables deferred functions
- possible to disable deferred functions but not all interrupts
- ops (macros):
  - local_bh_disable()
  - local_bh_enable()

What is RCU? (1)

- Reader-writer synchronization mechanism
- Writers use atomic commit points create new versions
- Readers can access old versions independently of subsequent writers
- Writers incur substantial overhead

What is RCU's Environment?

- Operating system kernels
- Many read-mostly data structures
- Motivates asymmetric approaches greatly favoring readers

RCU Key Idea

- Example: module unloading

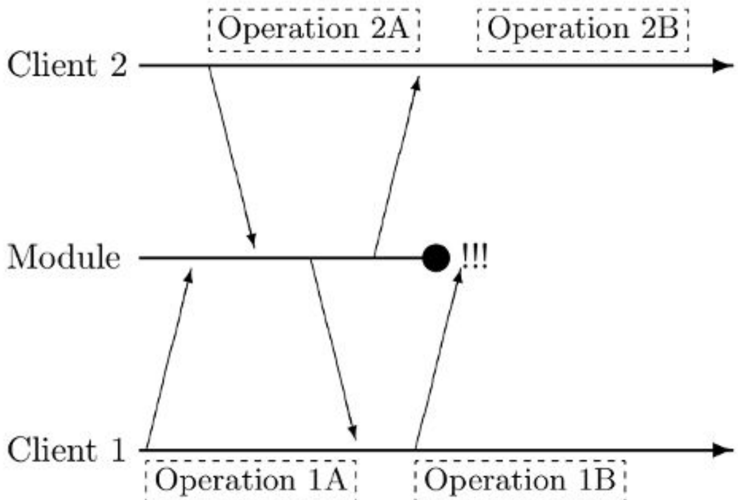


Figure 1: Race Between Teardown and Use of Service

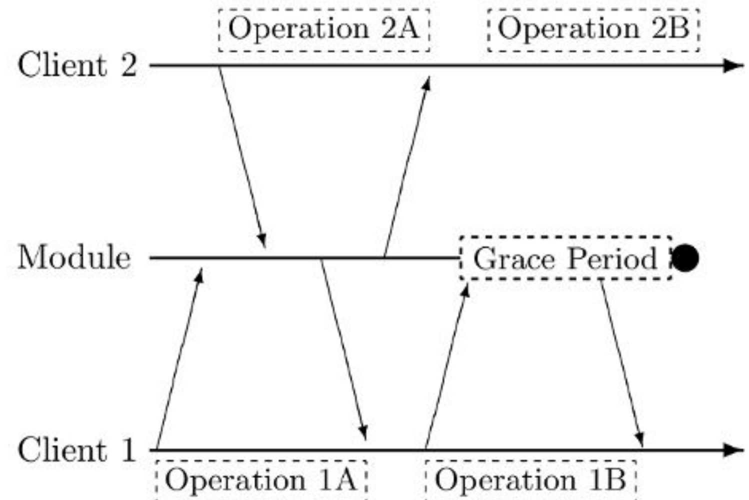


Figure 2: Read-Copy Update Handling Race

- Give ongoing operations a grace period to finish

RCU Performance Challenges

- How can readers signal writers?
  - Can reader completion be inferred from existing system state?
- How can destruction be deferred without burdening readers?
  - Can costs of destruction be associated with writers and batched to amortize overhead?
- How can memory consumption be limited?

How Does RCU Address Overheads?

- Lock Contention
  - Readers need not acquire locks: no contention!!!
  - Writers can still suffer lock contention
    - But only with each other, and writers are infrequent
    - Very little contention!!!
- Memory Latency
  - Readers do not perform memory writes
  - No need to communicate data among CPUs for cache consistency
    - Memory latency greatly reduced

Choosing Synch Primitives

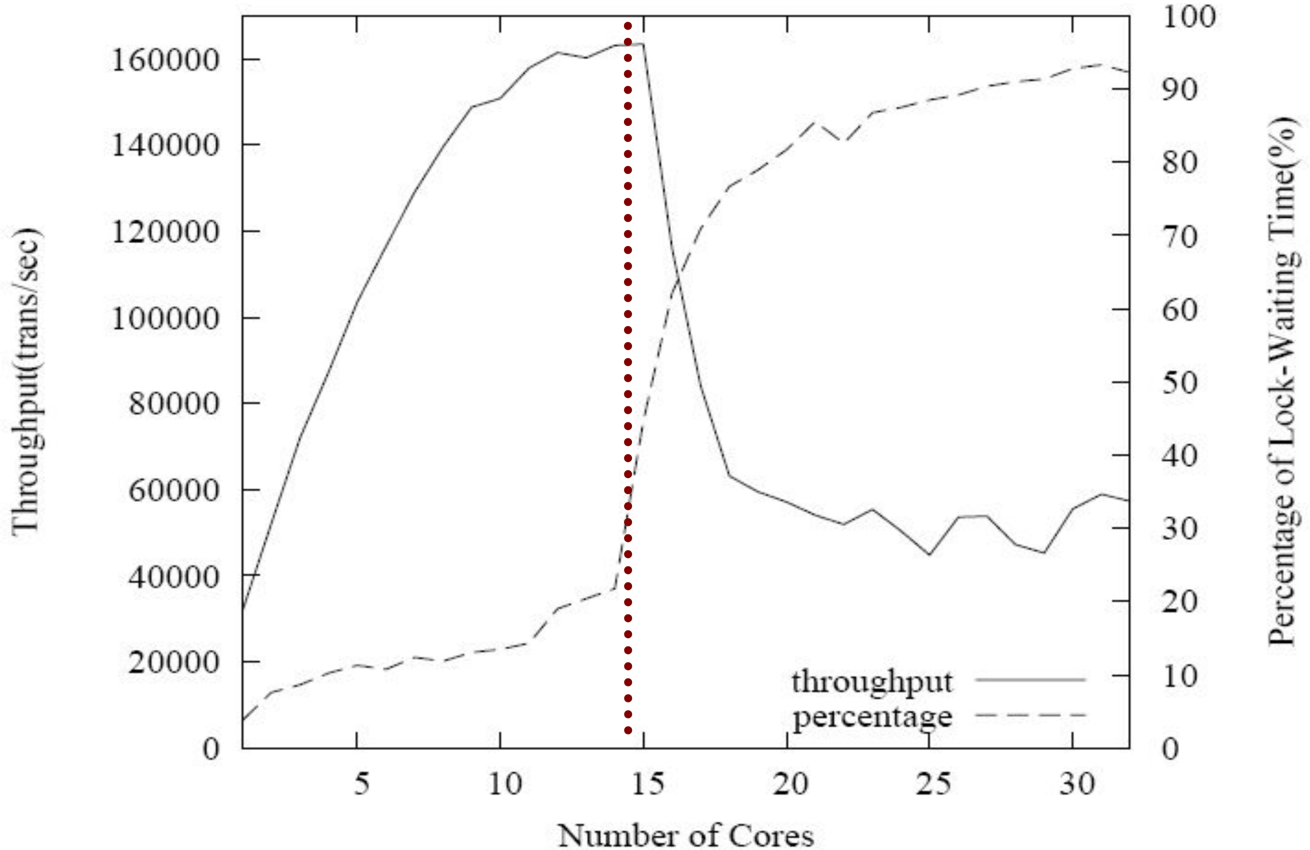
- avoid synch if possible! (clever instruction ordering)
  - example: inserting in linked list (needs barrier still)
- use atomics or rw spinlocks if possible
- use semaphores if you need to sleep
  - can’t sleep in interrupt context
  - don’t sleep holding a spinlock!
- complicated matrix of choices for protecting data structures accessed by deferred functions
  - Love has a nice summary

Contents

- Multicore Architecture
- Linux Synch Primitives
- Lock-awared scheduling

锁竞争感知调度算法

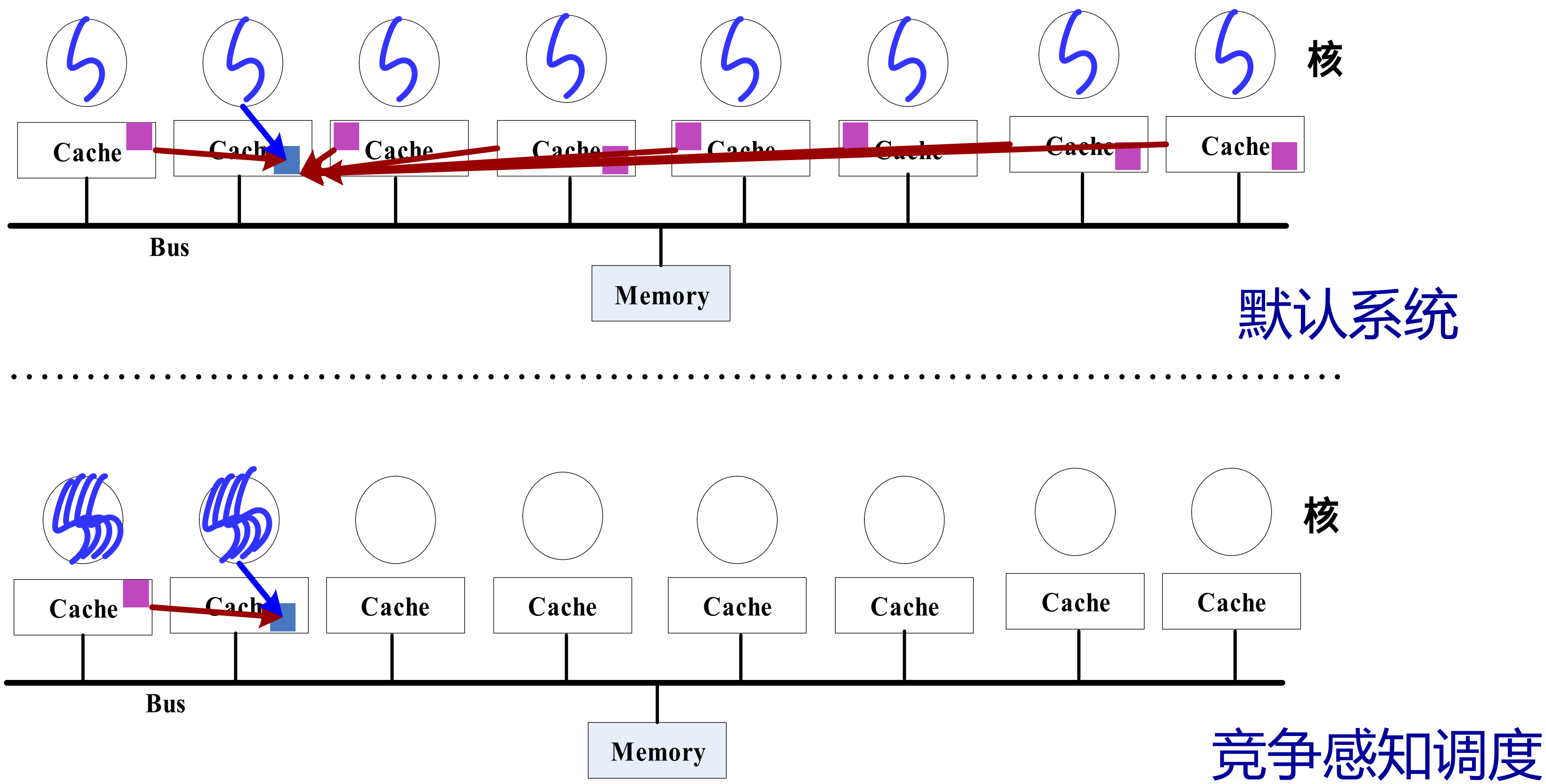
锁颠簸现象产生时平均等锁时间百分比会大幅度增加



Parallel postmark

锁竞争感知调度算法

基本方法



锁竞争感知调度算法

挑战

- 如何确定锁颠簸产生的时机
- 如何为锁密集任务分配计算资源
- 如何进行负载均衡
- 如何适应程序用锁行为的变化

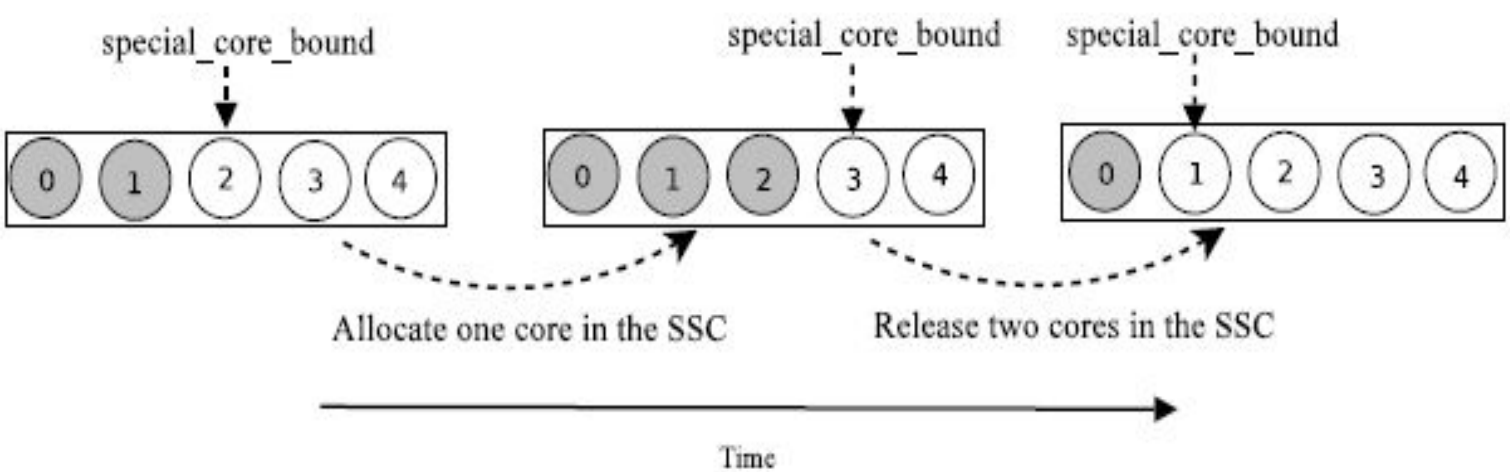
锁竞争感知调度算法

确定锁颠簸的时机

- 每个任务在线计算每个时间片内等锁时间百分比 若大于阈值 则认为是用锁密集任务 读取时间计数器开销可忽略

如何为锁密集任务分配资源

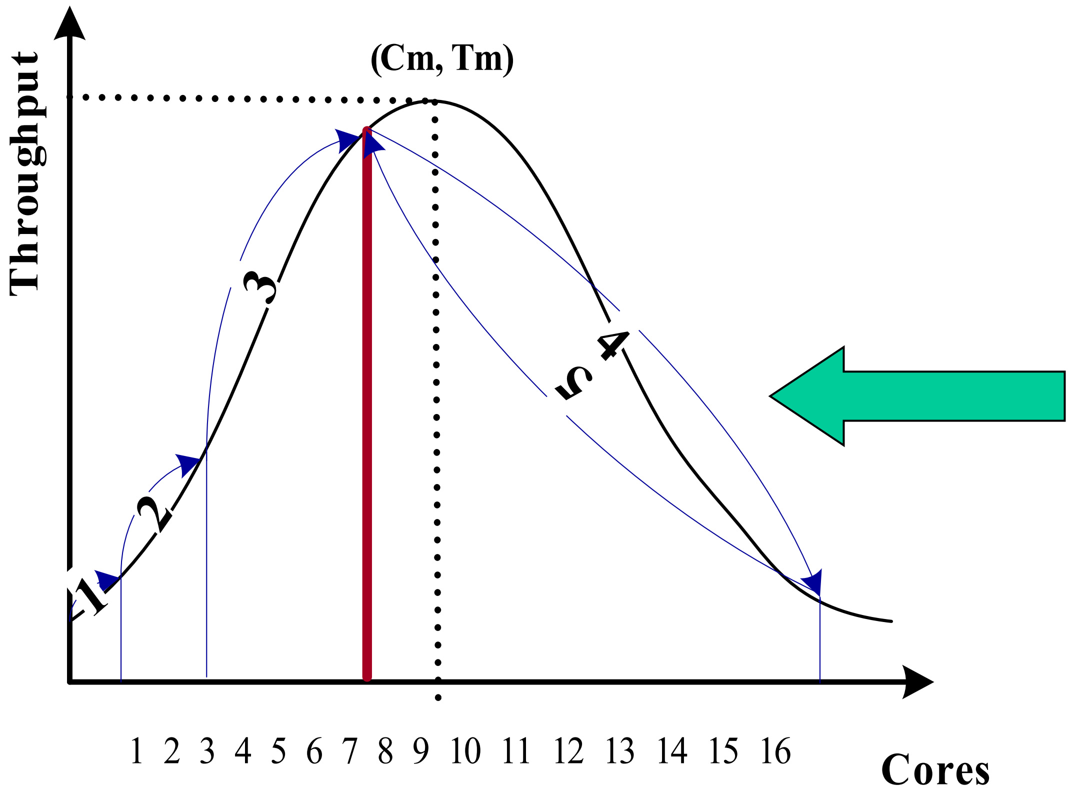
- 使用哪些核



锁竞争感知调度算法

如何为锁密集任务分配资源

使用多少核 在线测量误差 投票器 迁移状态机



锁竞争感知调度算法

- 如何进行负载均衡
  - 分离式负载均衡 避免全局均衡
- 如何适应程序用锁行为变化
  - 关键参数检测 锁密集任务数 在线模型吞吐量
  - 固定时间触发计时器

锁竞争感知调度算法

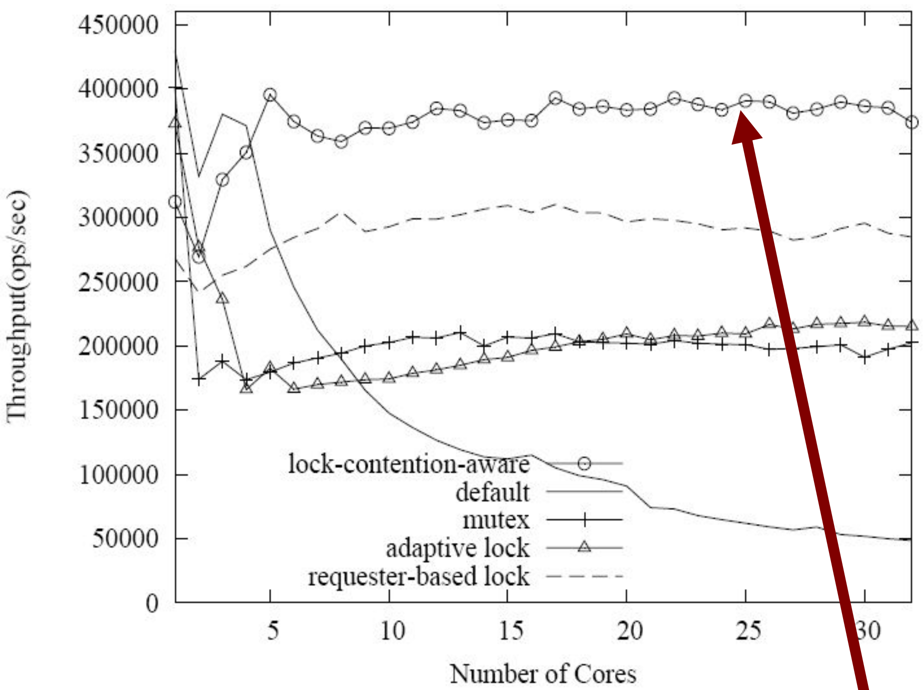
实验和评价

- 实现在Linux内核2.6.29.4和2.6.32
- AMD NUMA 32核 Intel NUMA 32核
- 测试程序:
  - 内存映射(mmapbench)
  - 套接字管理(sockbench)
  - 并行文件名查找(parallel find)
  - 内核编译(kernbench)
  - 文件服务器(parallel postmark)
  - 并行文件内容查找(parallel grep)

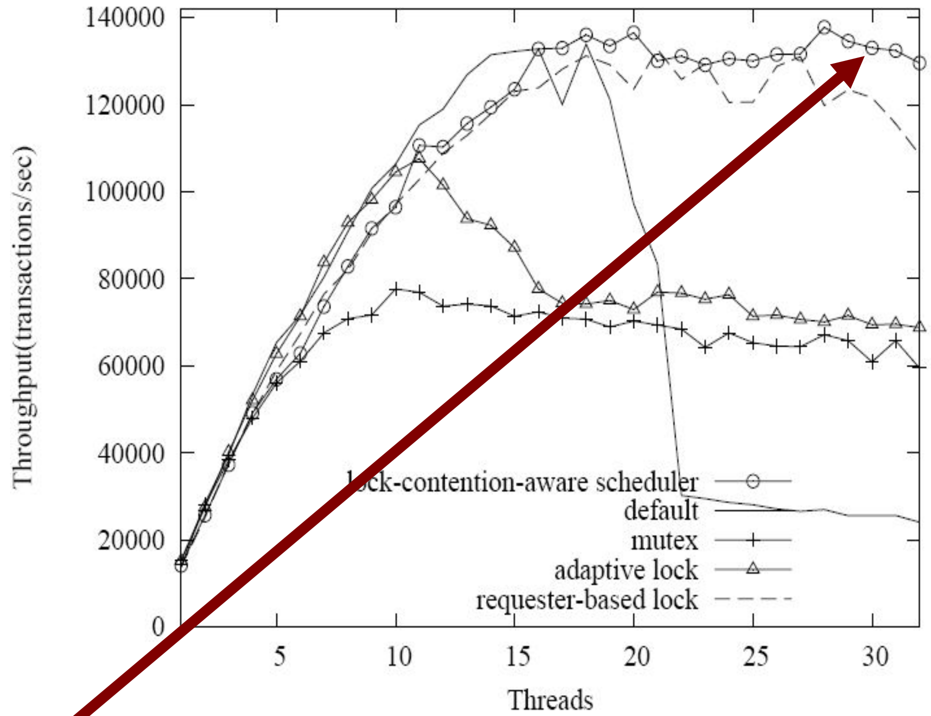
锁竞争感知调度算法

与基于等待者数目锁 互斥锁 自适应锁比较可扩展性

sockbench



parallel postmark



完全避免且维持峰值