无锁化编程
基础篇

张云开(仇恕)
@Chinainvent

Obstruction-free

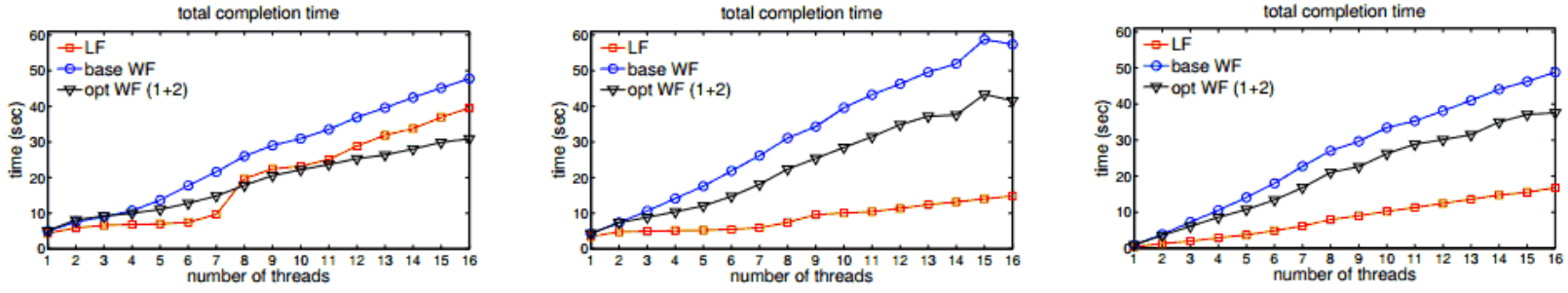
- 定义：某个线程，在孤立的情况下，可在有限步骤内完成任务
  - 何谓孤立：其他竞争的线程都处于suspend状态
  - 要求：其他线程做到一半的任务，可以被撤消

Starvation-free

- Starvation的定义：某个线程，永久地，无法获取到某个资源。
- Starvation-free：即指某个线程，总是可以获取到某个资源。
  - 获取资源的时间，未作为严格限制，可长可短。
  - starvation-free的线程，可以是阻塞的。

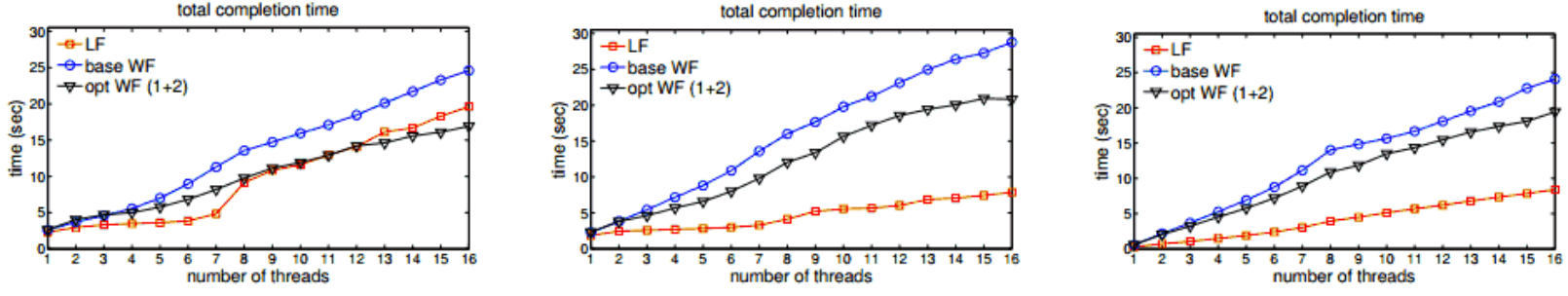
Wait-free

- 定义：任意一个线程，可在有限步骤内完成任务，而不会受到其他线程的影响
  - 非阻塞(obstruction-free)
  - 非饥饿(starvation-free)
- wait-free的算法很少
  - 2011，Kogan/Petrank，以色列理工大学
    - 多生产者-多消费者，并发访问的wait-free队列
    - 性能比lock-free队列差
    - 优点是：可控制每个操作的最差响应时间



(a) CentOS-operated machine (b) RedHat-operated machine (c) Ubuntu-operated machine

Figure 7. Performance results of the `enqueue-dequeue` benchmark.



(a) CentOS-operated machine (b) RedHat-operated machine (c) Ubuntu-operated machine

Figure 8. Performance results of the 50% `enqueues` benchmark.

Lock-free

- 定义：至少有一个线程，可在有限步骤内完成任务
  - 非阻塞(obstruction-free)
  - 部分线程，可能处于饥饿(starvation)状态
    - 例如：Retry-and-loop，但一直无法拿到资源
- wait-free属于lock-free

Compare-And-Swap(CAS)

- 常见C接口：`bool CAS(T *ptr, T old_val, T new_val)`
- AT&T 汇编：
  - `lock cmpxchg new_val, ptr`
  - 辅助寄存器：`%eax(old_val) and ZF`(作返回值)
- 伪代码(整个过程是一个原子操作)：

```
bool CAS(T *ptr, T old_val, T new_val)
{
  if (*ptr == old_val) {
    *ptr = new_val;
    return true;
  }

  return false;
}
```

Memory Barriers（内存屏障）

- 为啥需要内存屏障？
  - 编译器优化，打乱代码的顺序
  - CPU的乱序执行、延期及合并、预加载、分支预测、以及多种类型的缓存，都会打乱代码的顺序
- 内存屏障的种类
  - Write Memory Barriers
  - Data Dependency Barriers
  - Read Memory Barriers
  - General Memory Barriers

Load/Store操作

- Load操作：从内存，读数据，到寄存器
- Store操作：从寄存器，写数据，到内存
- 一条赋值语句的分解：
  - A = B;
  - 等价于：{`%reg = Load B; Store A = %reg`}
  - 如果A,B的地址，是内存对齐的，CPU可确保：
    - Load/Store，都是原子操作
    - 这两条指令，不会乱序

CPU的确定性

- 相互依赖的内存访问，不会乱序，例如：
  - Q = P; B = *Q;
  - B = *Q; Q = P;
- 相互交叠的内存访问，不会乱序，例如：
  - *Q = A; B = *Q;
  - B = *Q; *Q = A;
- 以上两种情况，若乱序，都会影响运行结果

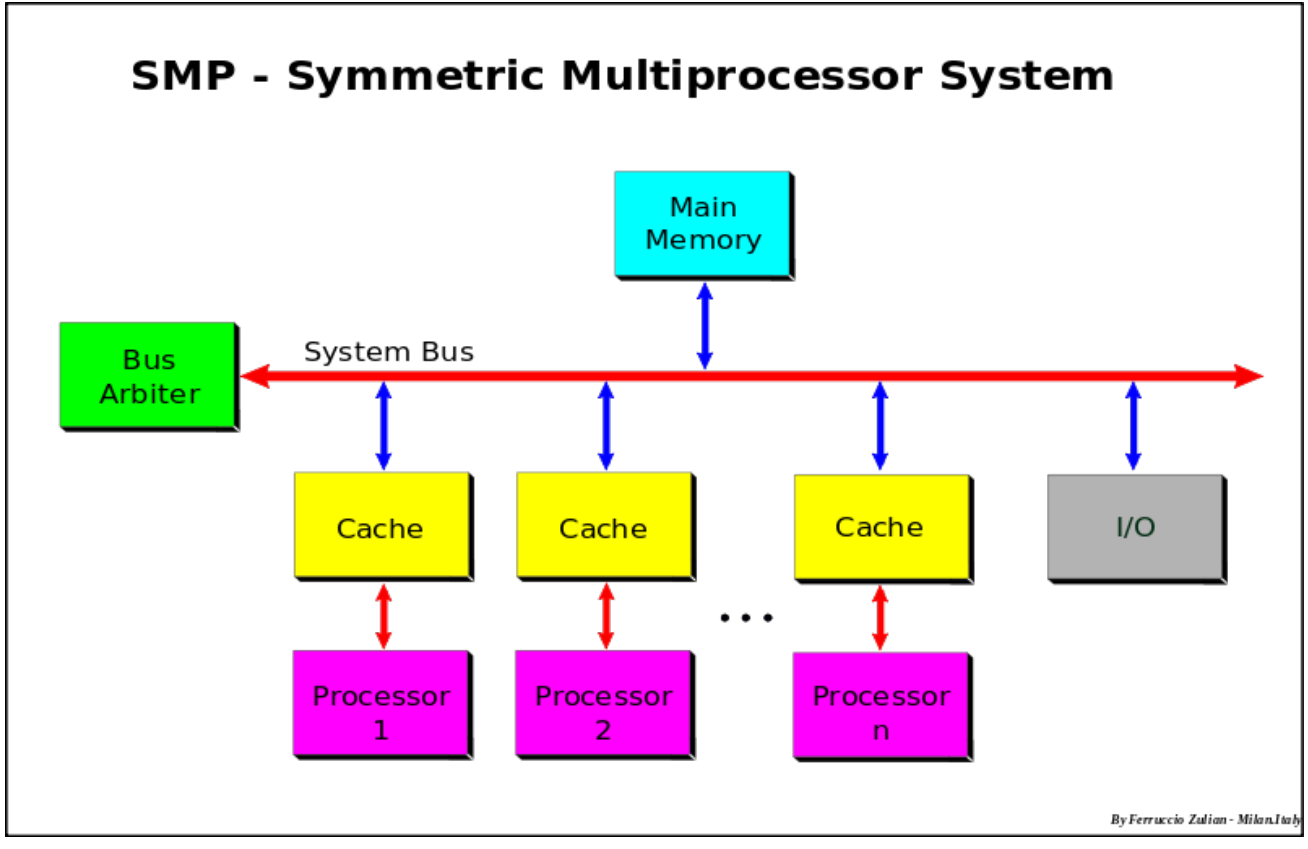
CPU的不确定性

- 无相互依赖的Store/Load操作，可能被乱序：
  - A = *P; B = *Q;
- 相互重叠的内存访问，可能被合并，例如：
  - X = *A; Y = *(A + 4);

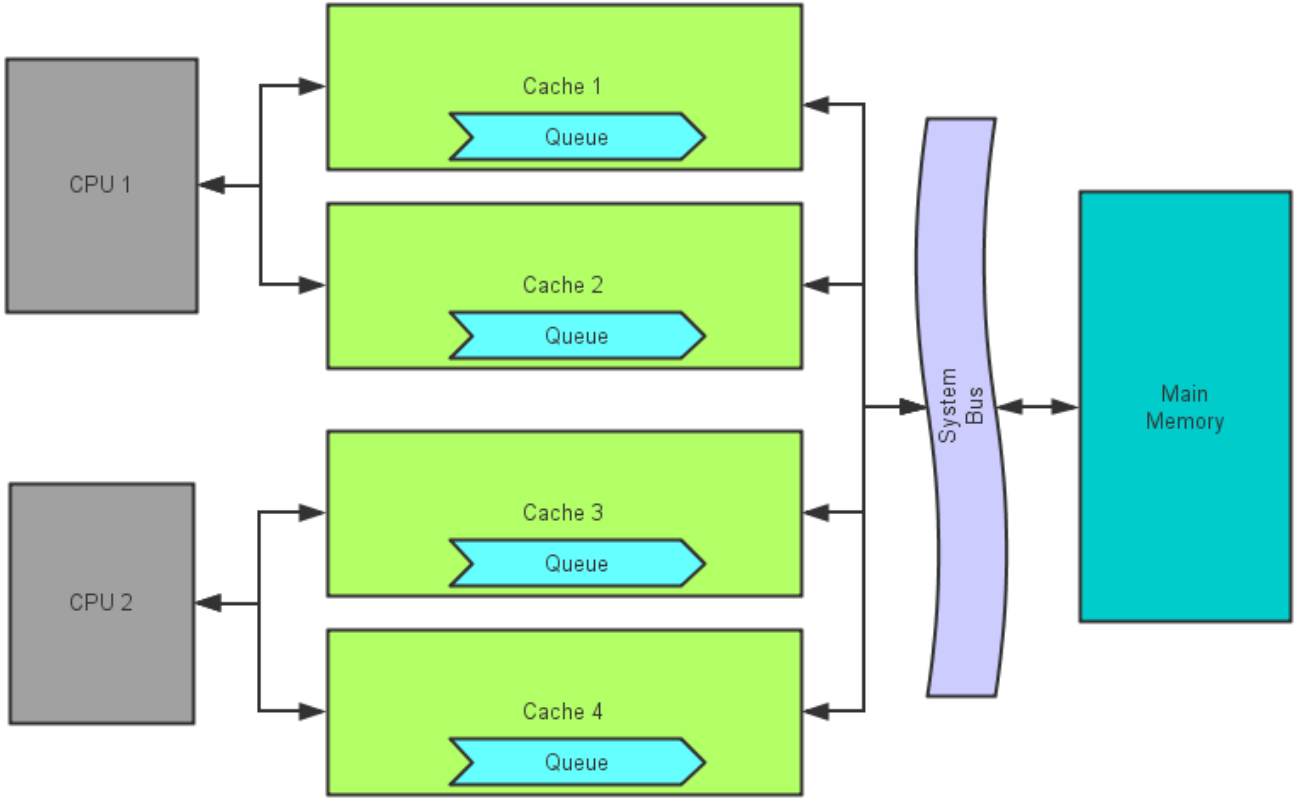
  {X, Y} = LOAD{*A, *(A + 4)} //变为一条load指令
- 乱序规则，可以归纳为一个原则：

  乱序后，在单个CPU下，不能影响程序的运行结果

SMP架构介绍


SMP - Symmetric Multiprocessor System
Main Memory
Bus Arbiter
System Bus
Cache
Cache
Cache
I/O
Processor 1
Processor 2
Processor n
By Ferruccio Zulian - Milan,Italy

DEC Alpha处理器的双cache结构



Write Memory Barriers

- 示例代码：

```
Init{int A = -1; int B = -1; int *P = NULL; int *Q = NULL;}

CPU 1                   CPU 2
===============         ===============
A = 100;                Q = P;
P = &A;                 B = *Q;
```

- 注意：
  - CPU 2的代码，是相互交叠的内存访问，不会乱序。

在CPU 2中，如果Q == &A，B == ？

Write Memory Barriers

- 写屏障的作用：
  - 所有在写屏障之前的Store操作，都会发生在该屏障之后的所有Store操作之前。
- 示例代码：

```
Init{int A = -1; int B = -1; int *P = NULL; int *Q = NULL;}

CPU 1                   CPU 2
===============         ===============
A = 100;
<write barrier>         Q = P;
P = &A;                 B = *Q;
```

在CPU 2中，如果Q == &A，B必等于100吗？

Write Memory Barriers

- 如果Q==&A，B可能不等于100，Why?

```
 Init{int A = -1; int B = -1; int *P = NULL; int *Q = NULL;}

CPU 1                        CPU 2
=======================    =======================
A = 100;
<write barrier>
<Cache 1: modify A=100>    <Cache 3: busy>        //Cache3发现4的更新，
                           <Cache 3: queue A=100>//由于处于busy状态，A暂存入队列
P = &A;                    Q = P;
                           <Cache 4: request P>
<Cache 2: modify P=&A>     <Cache 4: commit P=&A> //Cache4发现2的更新，及时同步
                           <Cache 4: read P>      //至此，Q == &A
                           B = *Q;
                           <Cache 3: read *Q>  //*Q为3更新前的A(-1)，因此B == -1
                           <Cache 3: unbusy>
                           <Cache 3: commit A=100> //Cache3不再忙，同步A == 100
```

内存屏障的不确定性

- 特定类型的内存屏障，只能影响特定类型的内存访问
  - 例如：写屏障，只影响Store操作，不影响Load
- 作用于某个CPU的内存屏障，其产生的效果，并不会对其他CPU有直接的影响。(为什么？)
- 其他CPU需要借助于***相对应***的内存屏障，才能获得间接的效果。
- 上一页的问题，该如解决？

Data Dependency Barriers

- 上例中，若Q==&A，如何确保B==100？

```
Init{int A = -1; int B = -1; int *P = NULL; int *Q = NULL;}

CPU 1                   CPU 2
===============         ===============
A = 100;                Q = P;
<write barrier>         <data dependency barrier>
P = &A;                 B = *Q;
```

- 数据依赖屏障的作用：
  - 对于存在依赖关系的Load操作，所有在该屏障之前的Load操作，所观察到的发生在其他CPU上的**Store操作序列**，都能被在该屏障之后的所有Load操作所感知

Data Dependency Barriers

- 数据依赖屏障，是如何工作的：

```
 Init{int A = -1; int B = -1; int *P = NULL; int *Q = NULL;}

CPU 1                       CPU 2
=======================     =======================
A = 100;
<write barrier>
<Cache 1: modify A=100>     <Cache 3: busy>        //Cache3发现4的更新，
                            <Cache 3: queue A=100>//由于处于busy状态，A暂存入队列
P = &A;                     Q = P;
                            <Cache 4: request P>
<Cache 2: modify P=&A>      <Cache 4: commit P=&A> //Cache4发现2的更新，及时同步
                            <Cache 4: read P>      //至此，Q == &A
                            <data dependcy barrier>
                            B = *Q;
                            <Cache 3: unbusy>
                            <Cache 3: commit A=100> //Cache3不再忙，同步A == 100
                            <Cache 3: read *Q>  //读*Q被推迟，直到队列中的A被同步
```

Read Memory Barriers

- 读屏障的作用：
  - 具有数据依赖屏障的功能，并且
  - 所有在读屏障之前的Load操作，都会发生在该屏障之后的所有Load操作之前
- 若X等于200，Y必等于100吗？

```
Init{int A = -1; int B = -1; int X = -1; int Y = -1;}

CPU 1                   CPU 2
===============         ===============
A = 100;                X = B;
<write barrier>         <data dependency barrier>
B = 200;                Y = A;
```

Read Memory Barriers

- 无依赖关系的Load，数据依赖屏障不起作用
- 使用读屏障后，若X等于200，Y必等于100：

```
Init{int A = -1; int B = -1; int X = -1; int Y = -1;}

CPU 1                   CPU 2
===============         ===============
A = 100;                X = B;
<write barrier>         <read barrier>
B = 200;                Y = A;
```

- 通用屏障：
  - 作用：(读屏障+写屏障)

常用的内存屏障指令

- 编译器级别
  - 禁止编译器优化造成的乱序，无法阻止CPU的乱序
  - GCC的通用内存屏障(编译器级别)：
    - `asm volatile("" ::: "memory")`
- CPU级别
  - GCC的通用内存屏障(CPU级别)
    - `__sync_synchronize (...)`
  - X86-64的内存屏障(摘自Linux实现)
    - `#define smp_mb() asm volatile("mfence":::"memory")`
    - `#define smp_rmb() asm volatile("lfence":::"memory")`
    - `#define smp_wmb() asm volatile("sfence" ::: "memory")`
    - `#define smp_read_barrier_depends() do { } while (0)  //why?`

C语言volatile关键字

- 编译器级别，指导编译器的优化策略
- 作用：强制从内存读(拒绝寄存器缓存)

```
//g_cnt是全局变量，会被其他线程实时更新
int g_cnt;

int foo()
{
  int cnt;
              /* 优化前
  for (;;){
    cnt = g_cnt;
    //do something
  }

  return 0;
}
```

```
int g_cnt;

int foo()
{
  int cnt;
优化后 */
  cnt = g_cnt; //g_cnt值被存于寄存器
  for (;;){
    //从寄存器直接读取g_cnt的值
    //do something
  }

  return 0;
}
```

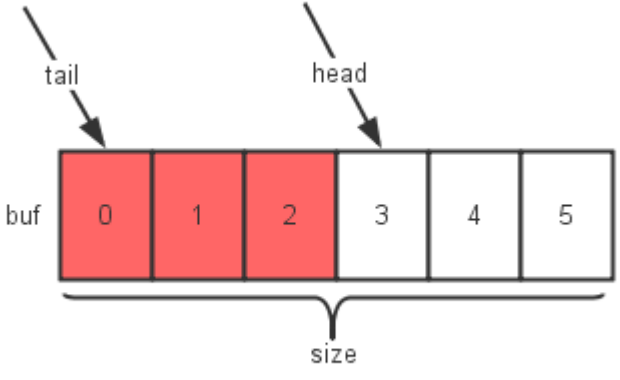
C语言volatile关键字

- 解决方法:使用volatile关键字
  - 定义时:`volatile int g_cnt;`
  - 或访问时:`cnt = *((volatile int *)&g_cnt);`
- ACCESS_ONCE():
  - 这是Linux kernel中常用到的宏
  - 为了便于使用volatile来修饰被访问的变量
  - `#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))`

Lock-free Ring buffer

- 接口：put()/get()/empty()/full()等
- 实现：
  - empty()：tail == head
  - full()：(head + 1)%size == tail
  - 可使用内存屏障实现无锁的put()/get()

```
struct ring_t {
  void **buf;
  int size;
  volatile int head;
  volatile int tail;
};
```



Lock-free Ring buffer

- ```
  bool
  ring_put(ring_t r, void *data)
  {
    int idx;

    idx = (r->head + 1) % r->size;

    if (idx == r->tail)
      return false;

    // 对应ring_get的smp_wmb()
    smp_read_barrier_depends();

    r->buf[idx] = data;

    // 防止head的更新提前
    smp_wmb();

    r->head = idx;

    return true;
  }
  ```

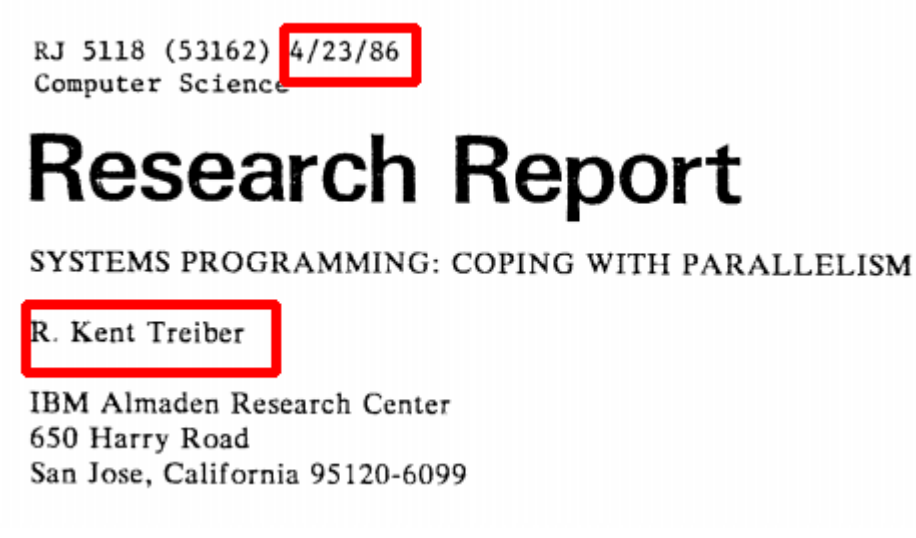
- ```
  bool
  ring_get(ring_t r, void **data)
  {
    int idx;

    idx = r->tail;

    if (idx == r->head)
      return false;

    // 对应ring_put的smp_wmb()
    smp_read_barrier_depends();

    *data = r->buf[idx];

    // 防止tail的更新提前
    smp_wmb();

    r->tail = (r->tail + 1) % r->size;

    return true;
  }
  ```

Lock-free Ring buffer

- ring_put()方法解析：
  - 先不考虑其数据依赖屏障（与get()是类似的）
  - 加入写屏障，防止head更新发生在buf[idx]的更新之前
- ring_get()方法解析：
  - idx与buf[idx]是相互依赖的关系
  - 当Ring刚从(idx == head)状态转变为(idx +1 == head)状态时：
    - if不成立，即说明idx的Load操作，感知到了head值的Store更新
    - 若无数据依赖屏障，从buf[idx]读到的值，可能不是最新值
    - 依赖屏障，能够让其后的Load操作，感知到在其之前的相互依赖的Load操作所感知到的Store操作序列（即buf[idx]和head的更新）

Lock-free LIFO(Stack)

- 世界上第一个lock-free的LIFO，汇编实现
  - 由R.Kent Treiber发表于1986年的计算科学杂志上
  - 这个LIFO，称之为Free Element List
  - 后面的学者，称之为「basic lock-free stack」



RJ 5118 (53162) 4/23/86
Computer Science

Research Report

SYSTEMS PROGRAMMING: COPING WITH PARALLELISM

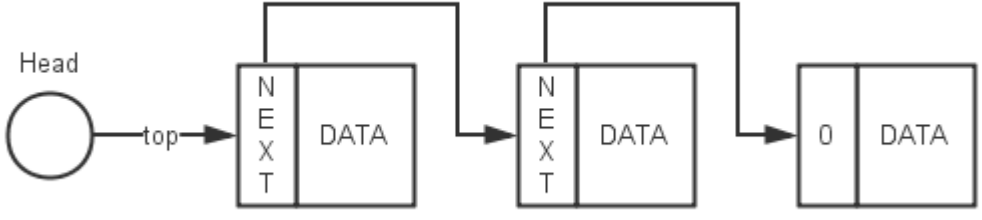
R. Kent Treiber

IBM Almaden Research Center
650 Harry Road
San Jose, California 95120-6099

Lock-free LIFO(Stack)

- 接口：push()/pop()
- 实现：
  - 拥有一个指向栈顶的Head指针
  - 基于Compare-And-Swap实现push()/pop()操作
  - TrafficServer的freelist，就是采用这种实现

```
struct stack_t {
  void *top;
};
```



Lock-free LIFO(Stack)

- push()

```
void
stack_push(stack_t *s, void *data)
{
  void *old_top, *new_top;

  do {
    old_top = s->top;

    new_top = data;

    *((void **)new_top) = old_top;

  } while (!CAS(&s->top,
                old_top,
                new_top));

}
```

- pop()

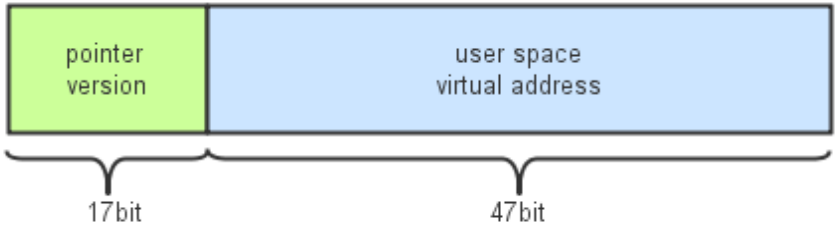
```
void*
stack_pop(stack_t *s)
{
  void *old_top, *new_top;

  do {
    old_top = s->top;

    if (!old_top)
      return NULL;

    new_top = *((void **)old_top);

  } while (!CAS(&s->top,
                old_top,
                new_top));

  return old_top;
}
```

ABA Problem

- 上一页的代码，存在ABA的问题
- 描述：
  - 线程P1，从共享的内存中(top指针)，读取到A值
  - 线程P2，抢占到CPU，线程P1被操作系统挂起
  - 线程P2，把top指针的值修改为B，最后又改回A值
  - 线程P1，被唤醒，发现top值未被修改，故可成功执行CAS()操作
- 后果：
  - 线程P1，无法完整地感知到，其他线程的更新

x86-64 Virtual Address

- 如果解决ABA问题？
  - 为指针，引入「版本号」
- X86-64架构的虚拟地址(Linux)
  - 高地址范围(内核地址空间)：
    - FFFF8000 00000000 - FFFFFFFF FFFFFFFF
  - 低地址范围(用户地址空间)：
    - 00000000 00000000 - 00007FFF FFFFFFFF
    - 可用空闲的高位存「版本号」，建议采用高16位



Lock-free LIFO(Stack)

- 改进后的push()/pop()

```
void
stack_push(stack_t *s, void *data)
{
  int version;
  void *old_top, *new_top;

  do {
    old_top = s->top;

    version = PTR_VER_GET(old_top);
    new_top = PTR_VER_SET(data, version+1);
    *((void **)new_top) = old_top;

  } while (!CAS(&s->top,
                old_top,
                new_top));

}
```

```
void*
stack_pop(stack_t *s)
{
  int version;
  void *old_top, *new_top;

  do {
    old_top = s->top;

    if (!PTR_ADDR_GET(old_top))
      return NULL;

    new_top = *((void **)old_top);
    version = PTR_VER_GET(old_top);
    new_top = PTR_VER_SET(new_top, version+1);

  } while (!CAS(&s->top,
                old_top,
                new_top));

  return old_top;
}
```

Lock-free LIFO(Stack)

- Treiber的Stack有什么缺点？
- 为Treiber Stack引入退避算法
  - 见论文《A Scalable Lock-free Stack Algorithm》

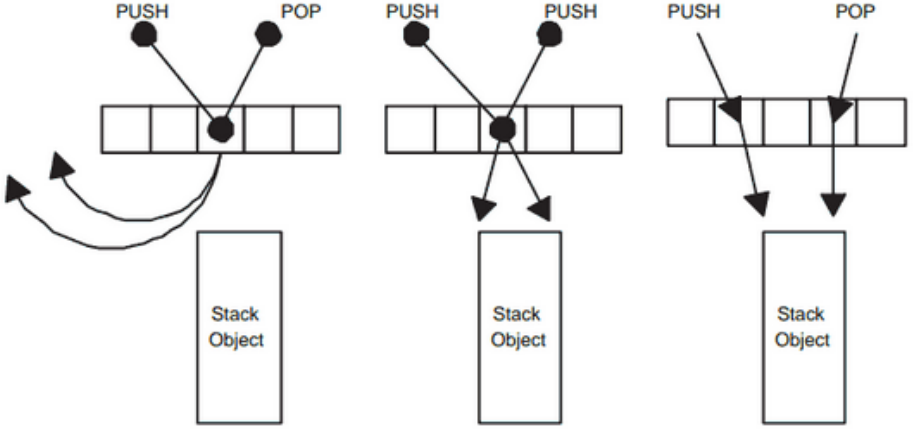
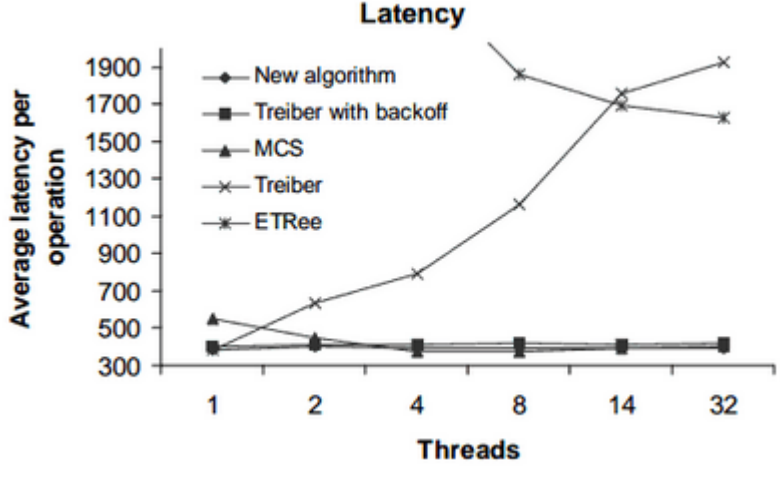
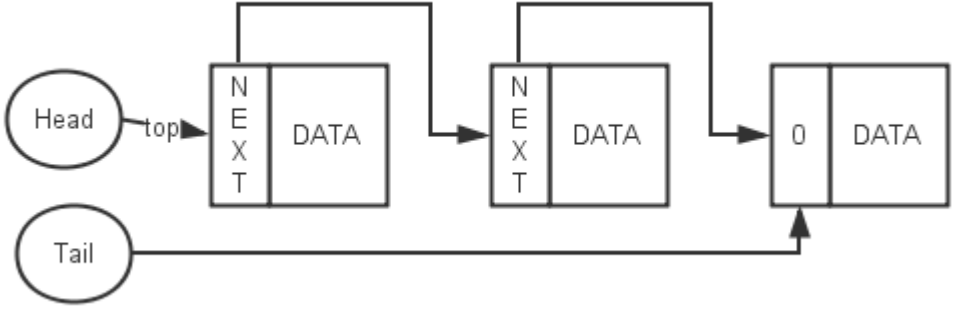


Figure 5: Collision scenarios



Lock-free FIFO(Queue)

- 接口：enqueue()/dequeue()
- 实现：
  - 见论文《Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms》



Read-Copy Update(RCU)

- 待续