并发 -2

参考书： OSTEP Chapter 28, 29

Lock-based Concurrent Data Structures

Basic Idea & Posix Library

- lock 本质是一个变量
  - 两个状态：unlocked 或 locked
  - Unlocked: 没有任何线程获得该锁
  - Locked: 有唯一的一个线程获得该锁
- POSIX 提供了 mutual exclusion 锁
  - 即 mutex

```
pthread_mutex_t lock =
PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);
balance = balance + 1;
pthread_mutex_unlock(&lock);
```

基于锁的并发数据结构

- 并发数据结构
  - 允许多线程并发访问
  - 线程安全
  - 高性能（追求线性扩展）

- 如何基于锁实现常用的并发数据结构？
  - Counter
  - Linked List
  - Queue
  - Hash Table

A Simple Counter

```
typedef struct __conter_t {
   int value;
} counter_t

void init(counter *c) { c->value=0; }
void increment(counter_t *c) { c->value++; }
void decrement(counter_t *c) { c->value--; }
int get(counter_t *c) { return c->value; }
```

Lock-based Approach

- 基于 mutex 锁实现
  - 同一时刻只有一个线程执行临界区

1. Acquire lock
2. . . . // Critical sections
3. Release lock

A Lock-based Counter

```
typedef struct __conter_t {
    int value;
    pthread_mutex_t lock;
} counter_t
```

```
void init(counter *c) {
    c->value=0;
    pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}
```

A Lock-based Counter

```
void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc
}
```

锁通常还保证了内存可见性

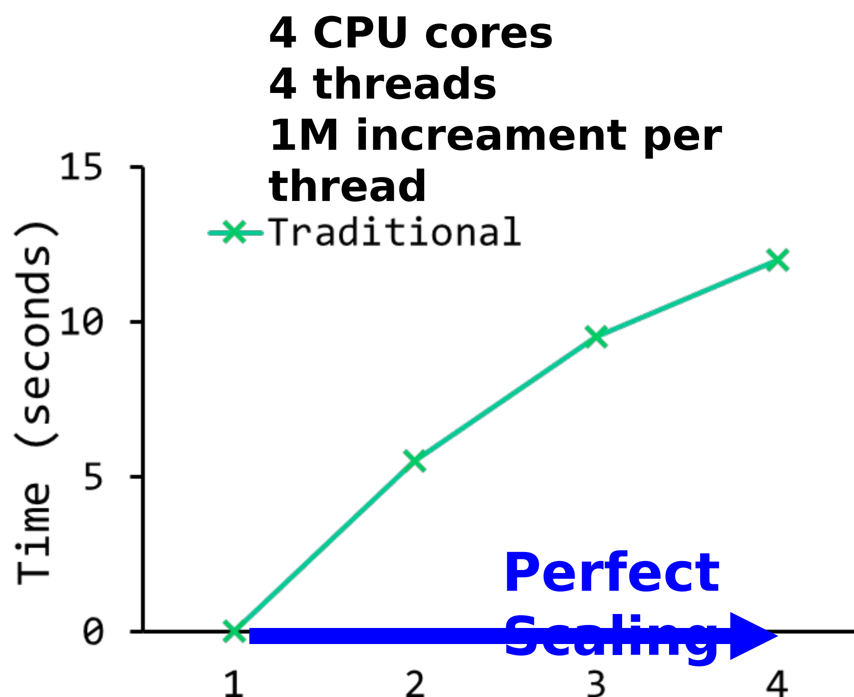
- 一个线程 W1 修改了数据结构，然后另一个线程 R1 读取数据结构
  - W1->R1: R1 应该能读到 W1 的结果，否则逻辑不正确
  - CPU 多核， W1 和 R1 分在不同核上
    - 没有锁的话， R1 可能只是看到旧的 cache ， W1 不一定触发其他核上的 cache 更新

Performance

- 传统 Lock-based Counter
  - Simple
  - Works correctly
  - Poor performance

Note that if the data structure is not too slow, you are done!

No need to do something fancy if something simple will work.



Scalable Counting

- Sloppy Counter
  - 一个 global counter，多个 local physical counters (one per CPU core)
  - 整体表现为一个 counter

Example:
A machine with 4 CPUs
- 4 local counter (**L1-L4**)
- 1 global counter (**G**)

Time	**L1**	**L2**	**L3**	**L4**	**G**

Sloppy Counter

- Scalable Write (`increment/decrement`)
  - Local counter 由对应的 local lock 来控制同步
  - 每个线程总是在同一个 CPU 核上增加 local counter （线程绑定到 CPU 核）
- Scalable Read (`get`)
  - Local counter 的值周期性地加到 global counter 上（同时 local counter 清零）
  - Global counter 由 global lock 来控制同步

Sloppy Counter

- Sloppiness (S)
  - The frequency of local-to-global transfer
  - Smaller S: more precise and more non-scalable
  - Bigger S: more imprecise and more scalable

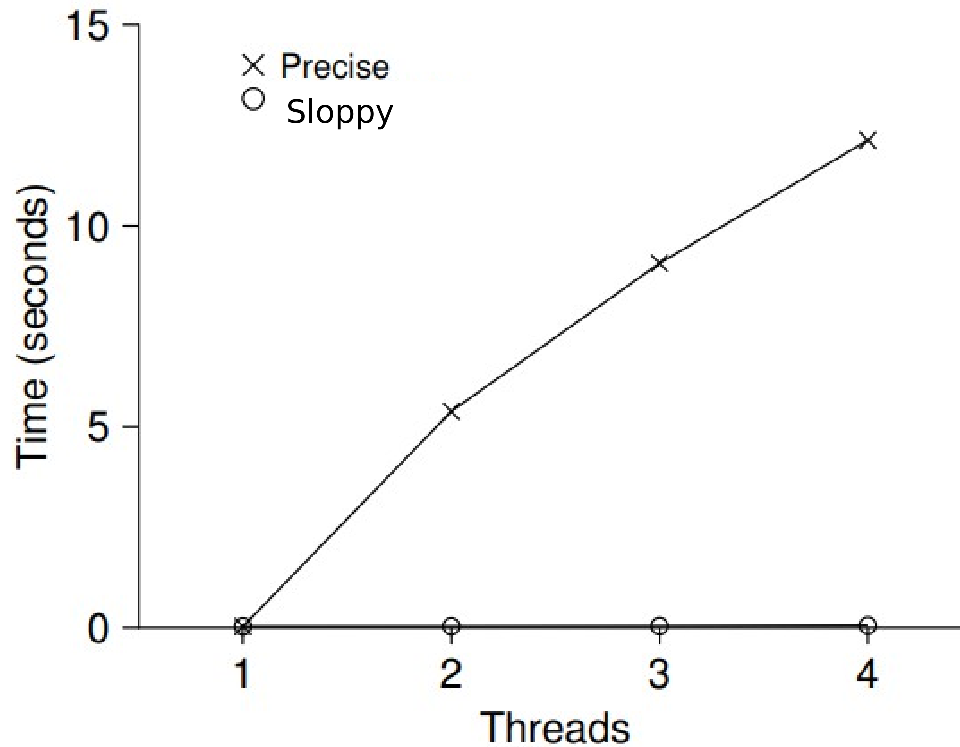
- Exact (non-scalable) Read
  - Acquire all the local locks and the global lock

Example

- A machine with 4 CPUs
  - S = 5

Time	L1	L2	L3	L4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5->0	1	3	4	5 (from L1)
7	0	2	4	5->0	10 (from L4)

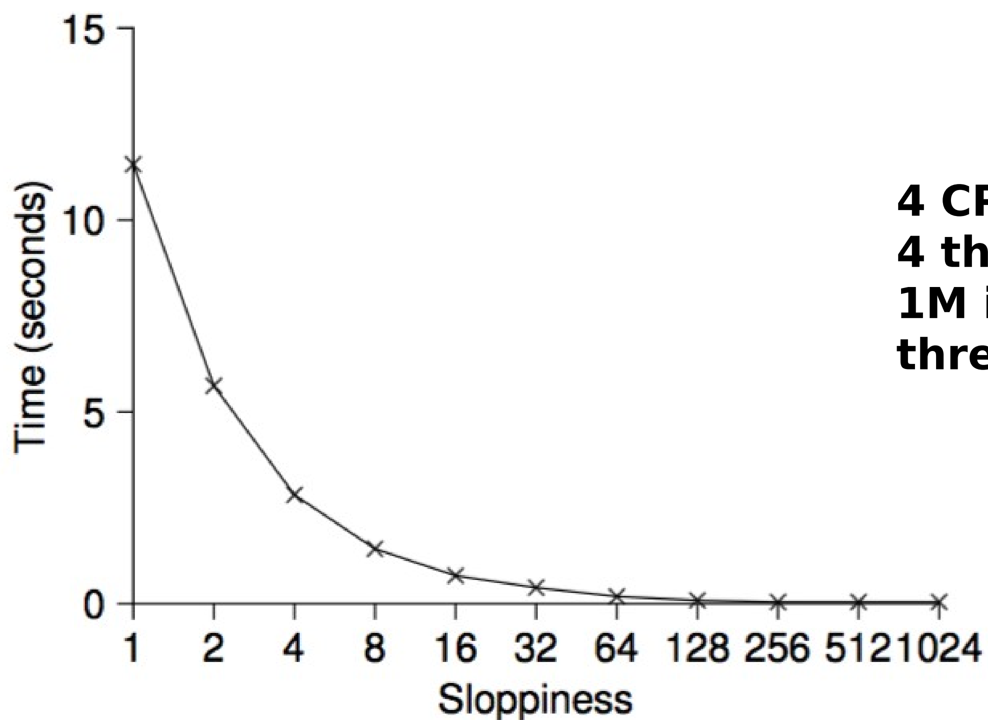
Example



4 CPU cores
4 threads
1M increament per thread

Example

- A machine with 4 CPUs
  - Each thread adds the counter 1 million times



4 CPU cores
4 threads
1M increament per thread

Sloppy Counter

```
typedef struct __counter_t {
  int             global        // global count
  pthread_mutex_t glock;        // global lock
  int             local[NCPUS]; // local counter (per cpu)
  pthread_mutex_t llock[NCPUS]; // ... and locks
  int             threshold;    // update frequency
} counter_t;
```

Sloppy Counter

```
// init: record threshold, init locks, init values
//       if all local counts and global count
void init (counter_t *c, int threshold) {
   c->threshold = threshold;
   c->global = 0;
   pthread_mutex_init(&c->glock, NULL);
   int i;
   for (i = 0; I < NCPUS; i++) {
      c->local[i] = 0;
      pthread_mutex_init(c->llock[i], NULL);
   }
}
```

Sloppy Counter

```
// update: usually, just grab local lock and update local
//         amount once local count has risen by 'threshold',
//         grab global lock and transfer local values to it
void update (counter_t *c, int threadID, int amt) {
   int cpu = threadID % NCPUS;
   pthread_mutex_lock(&c->llock[cpu]);
   c->local[cpu] += amt;                // assumes amt > 0
   if (c->local[cpu] >= c->threshold) { // transfer
      pthread_mutex_lock(&c->glock);
      c->global += c->local[cpu];
      pthread_mutex_unlock(&c->glock);
      c->local[cpu] = 0;
   }
```

Sloppy Counter

```
// get: just return global amount (which may not be
perfect)
int get (counter_t *c) {
   pthread_mutex_lock(&c->glock);
   int val = c->global;
   pthread_mutex_unlock(&c->glock);
   return val; // only approximate!
}
```

Concurrent Linked Lists

- Simple Concurrent Linked List
  - 利用锁来同步整个 insert 和 lookup 函数
  - 不要忘记在异常控制流上释放锁（容易出 bug）

A recent study of Linux kernel patches found that a huge fraction of bugs (nearly 40%) are found on such rarely-taken code paths

Simple Linked List

```
typedef struct __node_t { // basic node structure
  int             key;
  struct __node_t *next;
} node_t;

typedef struct __list_t { // basic list structure
  node_t          *head;
  pthread_mutex_t lock;
} list_t

void List_Init(list_t *L) {
   L->head = NULL;
   pthread_mutex_init(&L->lock, NULL);
}
```

Simple Linked List

```
int List_Insert(list *L, int key) {
   pthread_mutex_lock(&L->lock);
   node_t *new = malloc(sizeof(node_t));
   if (new == NULL) {
      perror("malloc");
      pthread_mutex_unlock(&L->lock);
      return -1; // fail
   }
   new->key = key;
   new->next = L->head;
   L->head = new;
   pthread_mutex_unlock(&L->lock);
   return 0; // success
}
```

exceptional path

Simple Linked List

```
int List_Lookup(list *L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            pthread_mutex_unlock(&L->lock);
            return 0; // success
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return -1; // failure
}
```

exceptional path

Concurrent Linked Lists

- 我们可以避免在 exceptional path 上调用 unlock 吗？
  - List_Insert(): malloc() 本身是线程安全的
  - List_Lookup(): 的 exit path 很常见

Simple Linked List

```
void List_Insert(list *L, int key) {
   // synchronization not needed
   node_t *new = malloc(sizeof(node_t));
   if (new == NULL) {
      perror(“malloc”);
      return;
   }
   new->key = key;

   // just lock critical section
   pthread_mutex_lock(&L->lock);
   new->next = L->head;
   L->head = new;
   pthread_mutex_unlock(&L->lock);
}
```

Simple Linked List

```
int List_Lookup(list *L, int key) {
   int rv = -1;
   pthread_mutex_lock(&L->lock);
   node_t *curr = L->head;
   while (curr) {
      if (curr->key == key) {
         rv = 0;
         break;
      }
      curr = curr->next;
   }
   pthread_mutex_unlock(&L->lock);
   return rv;
}
```

Scaling Linked Lists

- Hand-over-hand locking (a.k.a. lock coupling)
  - 使用 node 级别的锁，而不是整个 list 级别的锁
  - 在释放当前结点锁之前，先抢占下一个结点的锁
  - 但是，频繁的获得 / 释放锁，代价很大，所以这种方案 impractical

MORE CONCURRENCY ISN'T NECESSARILY FASTER
刻意追求并发并不一定让程序性能更好

Concurrent Queues

- Simple (always work) Solution
  - Add a big lock

- Michael and Scott Concurrent Queues
  - Add two locks, one for head, and one for tail
  - Queue_Enqueue always uses tail lock
  - Queue_Dequeue always uses head lock
  - A dummy node enables the separation of head and tail operations

Michael and Scott Concurrent Queue

```
typedef struct __node_t {
  int              key;
  struct __node_t *next;
} node_t;

typedef struct __queue_t {
  node_t          *head;
  node_t          *tail;
  pthread_mutex_t  head_lock;
  pthread_mutex_t  tail_lock;
} queue_t
```

Michael and Scott Concurrent Queue

```
void Queue_Init(queue_t *q) {
   node_t *tmp = malloc(sizeof(node_t));
   tmp->next = NULL;
   q->head = q->tail = temp;
   pthread_mutex_init(&q->head_lock, NULL);
   pthread_mutex_init(&q->tail_lock, NULL);
}
```

add a dummy node

Michael and Scott Concurrent Queue

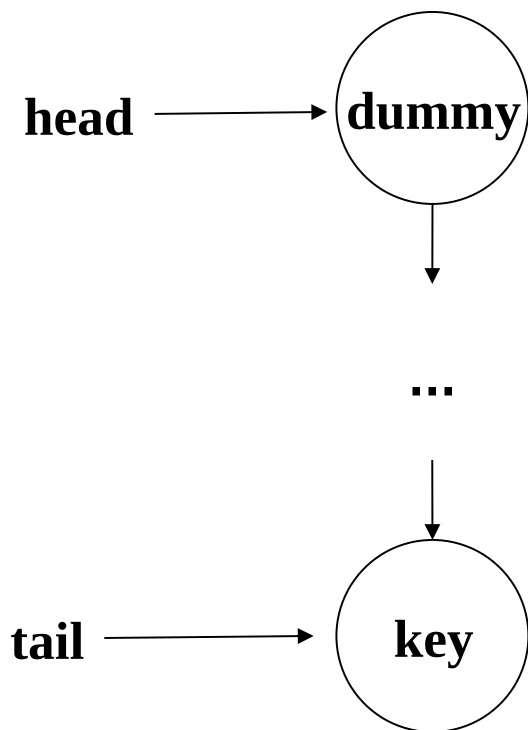
```
void Queue_Enqueu(queue_t *q, int value) {
   node_t *tmp = malloc(sizeof(node_t));
   assert(tmp != NULL);
   tmp->value = value;
   tmp->next = NULL;

   pthread_mutex_lock(&q->tail_lock);
   q->tail->next = tmp;
   q->tail = tmp;
   pthread_mutex_unlock(&->tail_lock);
}
```

将新节点加到队尾

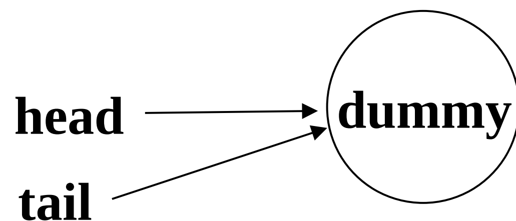
Michael and Scott Concurrent Queue

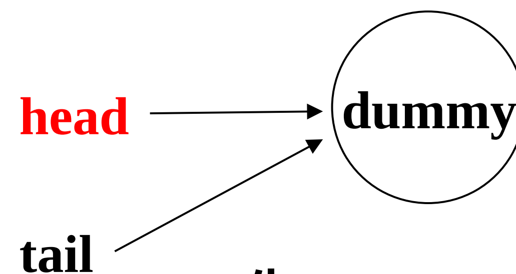
```
void Queue_Dequeu(queue_t *q, int *value) {
   pthread_mutex_lock(&q->head_lock);
   node_t *tmp = q->head;
   node_t *new_head = tmp->next;
   if (new_head == NULL) {
      phtread_mutex_unlock(&q->head_lock);
      return -1; // queue was empty
   }

   *value = new_head->value;
   q->head = new_head;
   pthread_mutex_unlock(&->head_lock);
   free(tmp);
   return 0;
}
```

Michael and Scott Concurrent Queue



ok





先 dequeu，再 enqueu，ok

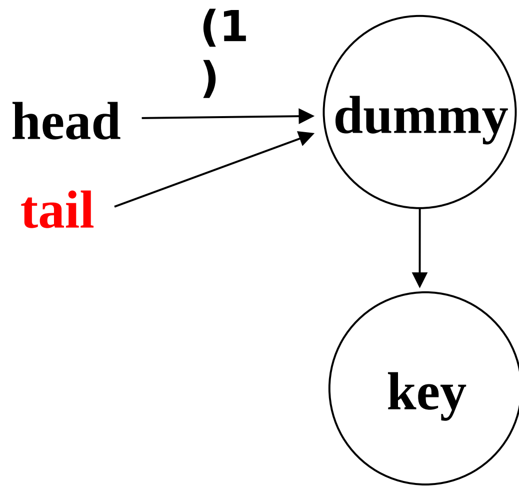
先开始 dequeu，中间夹杂 enqueu 操作:

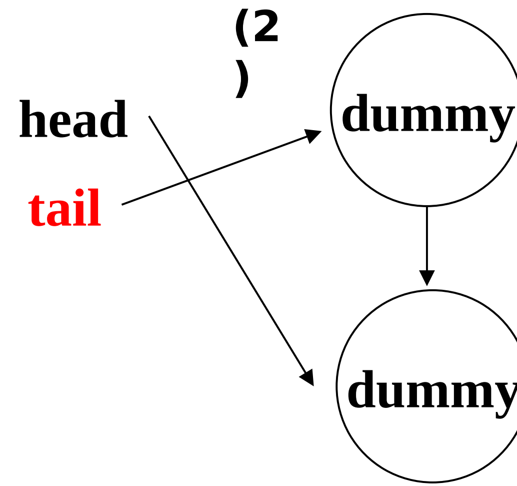
`node_t *tmp = q->head;` 后 enqueu 一个 node:

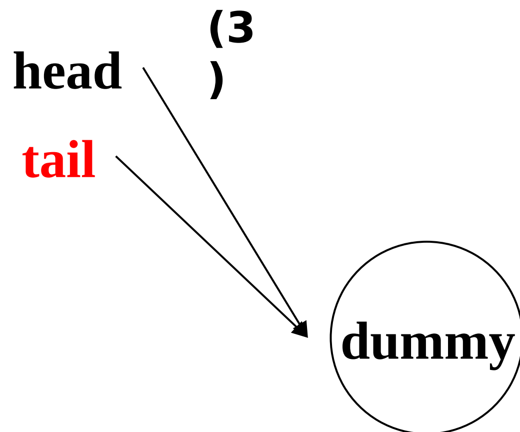
可以 dequeu 新插入数据

`node_t *new_head = tmp->next;` 后 enqueu:

Michael and Scott Concurrent Queue







先开始 enqueu，中间夹杂 dequeu，ok

Concurrent Hash Table

```
#define BUCKETS (101)

typedef struct __hash_t {
  list_t lists[BUCKETS];
} hash_t

void Hash_Init(hash_t *H) {
   int I;
   for (I = 0; I < BUCKETS; i++) {
      List_Init(&H->lists[i]);
   }
}
```

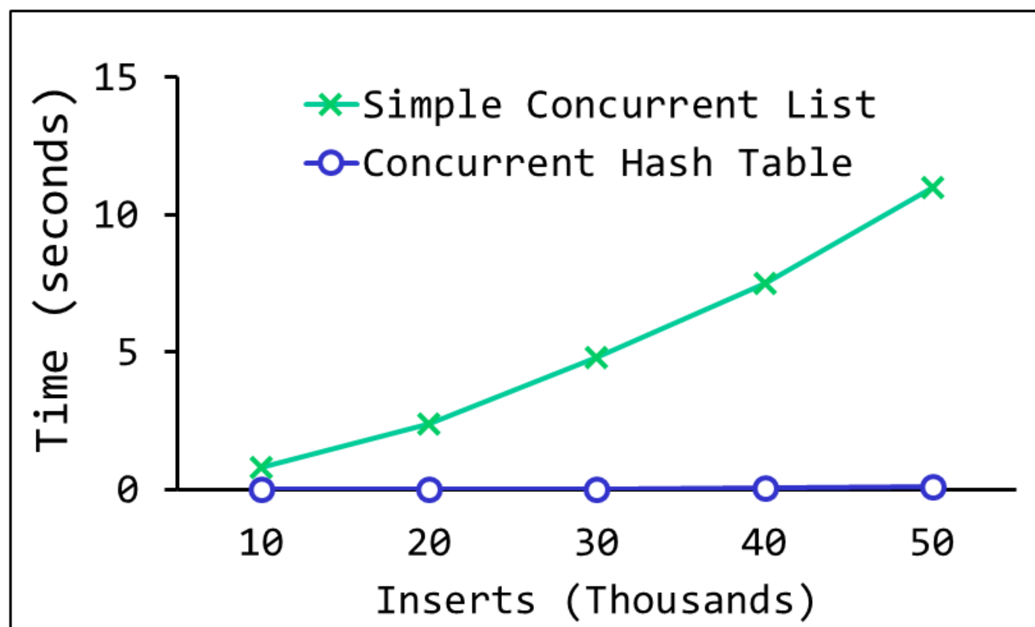
Hash table 中每个 bucket 是一个 linked list

Concurrent Hash Table

```
void Hash_Insert(hash_t *H, int key) {
   int bucket = key % BUCKETS;
   return List_Insert(&H->lists[bucket], key);
}

void Hash_Lookup(hash_t *H, int key) {
   int bucket = key % BUCKETS;
   return List_Lookup(&H->lists[bucket], key);
}
```

Concurrent Hash Tables

- Scaling Concurrent Hash Tables
  - Based on concurrent linked lists
  - Each bucket is represented by a list
  - A lock per hash bucket



4 CPU cores
4 threads
10K-50K updates per thread

锁的类型

- 互斥量 pthread_mutex_t
  - pthread_mutex_init();
  - pthread_mutex_destroy();
  - pthread_mutex_lock(); 阻塞
  - pthread_mutex_trylock(); 非阻塞
  - Pthread_mutex_timelock(); 阻塞时间
  - pthread_mutex_unlock();

锁的类型

- 信号量 sem_t
  - 初值可 >1
  - sem_init();
  - sem_wait(); //P()
  - sem_post(); //V()

锁的类型

- 读写锁 pthread_rwlock_t
  - 读读可并发
  - 读写、写写互斥（e.g.，正在读，新来的读可并行，新来的写要等待）
  - 不同操作系统策略不同，但一般不会读优先
  - pthread_rwlock_init();
  - pthread_rwlock_destroy();
  - pthread_rwlock_rdlock();
  - pthread_rwlock_wrlock();
  - pthread_rwlock_unlock();

锁的类型

- 自旋锁 pthread_spinlock_t
  - 与互斥量区别：不通过休眠使进程阻塞，而是一直忙等（旋转）；适合锁持有时间短、线程不希望在 context switch 和 thread scheduling 上花费太多时间
  - pthread_spin_init();
  - pthread_spin_destroy();
  - pthread_spin_lock();
  - pthread_spin_trylock();
  - pthread_spin_unlock();

锁的类型

- 两阶段锁
  - 第一阶段 spin
  - 如果超时，仍没获得锁，则 sleep

无锁化

- CAS (compare and swap)
  - 加锁原因：缺少原子性的 Load-Update-Store
    - Load
    - Atom{if (val == loaded) update & store}

  - CAS：原子性操作，由硬件提供支持
  - 原理性代码：

```
1  int CAS(long *addr, long old, long new)
2  {
3      /* Executes atomically. */
4      if(*addr != old)
5          return 0;
6      *addr = new;
7      return 1;
8  }
```

无锁化

- CAS 的汇编实现
  - Lock 前缀：设置处理器的 LOCK# 信号 (锁定总线 , 阻止其它处理器接管总线访问内存 , 独占内存), 直到 LOCK 前缀指定的执行执行结束 , 使这条指令为原子操作
    - Cmpxchgl %2, %1：判定 %1(*ptr) 与 eax(old) 是否相等，如果相等则 %1=%2 （ new ），否则不修改
    - Sete: %0(ret) = zf (0 或 1)：返回 1 表示成功交换

```
int CAS(unsigned long *dst, unsigned long oldVal, unsigned long newVal){
        unsigned char ret;
        __asm__ __volatile__ (
                "  lock cmpxchgq %2,%1\n"
                "  sete %0\n"
                : "=q" (ret), "=m" (*dst)
                : "r" (newVal), "m" (*dst), "a" (oldVal)
                : "memory");
        if (ret)
                return 1;
        return 0;
}
```

C 语言嵌入汇编

- "__volatile__" 表示编译器不要优化代码，后面的指令 保留原样
- 输出部分 :”=q” (ret), “=m” (*ptr)
- 输入部分： :”r” (new), “m” (*ptr), “a” (old)
  - a-eax; q-eax,ebx,ecx,edx; r-”q”+esi, edi
  - m 内存变量
- 破坏描述 (clobbered registers) 部分： :”memory”, “%eax”(编译器就不会去用 eax)
- https://blog.csdn.net/u011006622/article/details/89496500

Lock-free Concurrent List

- 如何利用 CAS 实现一个无需 mutex 的 concurrent list？

```
void List_Insert(list *L, int
key) {
   // synchronization not needed
   node_t *new =
malloc(sizeof(node_t));
   if (new == NULL) {
      perror("malloc");
      return;
   }
   new->key = key;

   // just lock critical section
   pthread_mutex_lock(&L->lock);
   new->next = L->head;
   L->head = new;
   pthread_mutex_unlock(&L-
>lock);
```

```
do {
        new->next = L->head;
}while(CAS(&(L->head), new->next, new) == 0);
```

Thread & Lock: Advanced Contents I

锁的实现原理

- A1. 禁止 - 恢复中断，中间形成临界区

```
1   void lock() {
2       DisableInterrupts();
3   }
4   void unlock() {
5       EnableInterrupts();
6   }
```

  – 优点：正确，线程不能被打断
  – 缺点：
    - 进程特权必须足够高，能够关中断；并可能被滥用
    - 多核情况不管用，没法禁止其他核中断
    - 时间长可能导致中断丢失，严重错误
    - 效率低，开关中断执行时间长

锁的实现原理

- A2. 简单 spin-wait 方案

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;          // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

锁的实现原理

- A2. 简单 spin-wait 方案
  - 不正确！ 可能都抢到锁

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

锁的实现原理

- A3. Test And Set (Atomic Exchange)
  - 上一个方案的问题， flag 的读和写不是原子性的
  - 解决方案：原子性的 test and set
    - SPARC – ldstub, load/store unsigned byte
    - x86 - xchg

```
1       int TestAndSet(int *old_ptr, int new) {
2           int old = *old_ptr; // fetch old value at old_ptr
3           *old_ptr = new;     // store 'new' into old_ptr
4           return old;         // return the old value
5       }
```

锁的实现原理

- A3. Test And Set (Atomic Exchange)

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 indicates that lock is available, 1 that it is held
7       lock->flag = 0;
8   }
9
10  void lock(lock_t *lock) {
11      while (TestAndSet(&lock->flag, 1) == 1)
12          ; // spin-wait (do nothing)
13  }
14
15  void unlock(lock_t *lock) {
16      lock->flag = 0;
17  }
```

锁的实现原理

- A3. Test And Set (Atomic Exchange)
  - 评价
    - 正确性： OK
    - 公平性：没有任何保障，可能有饥饿问题
    - 性能：
      - 单核：性能差
      - 多核：可能 OK，如果其他核上没有其他任务（不用 lock# 的话）
        - » 假设一个线程刚释放锁（ lock->flag = 0 ）
        - » 另外两个线程在 2 个核上同时抢锁，都读到 lock->flag=0

锁的实现原理

- A4. Compare and Swap
  - Test and Set 的高级版本
    - X86: cmpxchg
    - 实现锁，效果与 test and set 相同

```
1   int CompareAndSwap(int *ptr, int expected, int new) {
2       int actual = *ptr;
3       if (actual == expected)
4           *ptr = new;
5       return actual;
6   }
```

```
1   void lock(lock_t *lock) {
2       while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3           ; // spin
4   }
```

锁的实现原理

- A6. Fetch and Add
  - 利用硬件提供的原子性的 Fetch and Add 操作:

```
1   int FetchAndAdd(int *ptr) {
2       int old = *ptr;
3       *ptr = old + 1;
4       return old;
5   }
```

锁的实现原理

- A6. Fetch and Add
  - 锁的实现：

```
1   typedef struct __lock_t {
2       int ticket;
3       int turn;
4   } lock_t;
5
6   void lock_init(lock_t *lock) {
7       lock->ticket = 0;
8       lock->turn   = 0;
9   }
10
11  void lock(lock_t *lock) {
12      int myturn = FetchAndAdd(&lock->ticket);
13      while (lock->turn != myturn)
14          ; // spin
15   }
16
17  void unlock(lock_t *lock) {
18      lock->turn = lock->turn + 1;
19  }
```

锁的实现原理

- A6. Fetch and Add
  - 与之前解决方案的一点不同：
    - 保证进程调度的顺序
    - 按照到达的顺序，因为 myturn 的值从小到大递增

锁的实现原理

- Too Much Spinning: What Now?
  - Spin 消耗 CPU
  - 仅仅依靠硬件支持不够，还需要 OS 的支持，才能解决这个问题

锁的实现原理

- B1. Just Yield
  - 放弃 CPU，从 running 状态变为 ready 状态

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

  - 缺点：
    - 线程数量多（100）时，线程来回切换的代价不小
    - 公平性、饥饿问题

锁的实现原理

- B2. Queues: Sleeping Instead Of Spinning
  - B1 的问题：主动权完全交给 scheduler
  - 我们自己做一个队列，把 lock 等待的 thread 放进去管理呢?
  - 利用操作系统的支持， Solaris 中的 park() 和 unpark(threadID)
    - Park: 调用线程 sleep
    - Unpark： 唤醒指定线程

```
1   typedef struct __lock_t {
2       int flag;
3       int guard;
4       queue_t *q;
5   } lock_t;
6
7   void lock_init(lock_t *m) {
8       m->flag  = 0;
9       m->guard = 0;
10      queue_init(m->q);
11  }
12
13  void lock(lock_t *m) {
14      while (TestAndSet(&m->guard, 1) == 1)
15          ; //acquire guard lock by spinning
16      if (m->flag == 0) {
17          m->flag = 1; // lock is acquired
18          m->guard = 0;
19      } else {
20          queue_add(m->q, gettid());
21          m->guard = 0;
22          park();
23      }
24  }
25
26  void unlock(lock_t *m) {
27      while (TestAndSet(&m->guard, 1) == 1)
28          ; //acquire guard lock by spinning
29      if (queue_empty(m->q))
30          m->flag = 0; // let go of lock; no one wants it
31      else
32          unpark(queue_remove(m->q)); // hold lock (for next thread!)
33      m->guard = 0;
34  }
```

锁的实现原理

- B2. Queues: Sleeping Instead Of Spinning
  - 有一个 bug:
    - 如果 thread1 在进入 park() 前 [L22 前]， 调度器切换到另一个 thread2， thread2 之前持有锁，现在释放了。
    - 那么 thread1 就永远沉睡了
    - 被称为 wakeup/waiting race

锁的实现原理

- B2. Queues: Sleeping Instead Of Spinning
  - Solaris 解决方法：
    - setpark()： 说自己将要 park，如果在自己 park 前被打断，而且另一个线程调用了 unpark，那么该线程调用 park 函数时不会进入 sleep 状态，而是立刻返回。
    - 同时有另一套机制，让内核调度去选择把锁交给哪个等待线程

```
1    queue_add(m->q, gettid());
2    setpark(); // new code
3    m->guard = 0;
```

锁的实现原理

- B3. Linux mutex 实现
  - 基于 futex
  - futex_wait(address, expected)：如果 address 处的值等于 expected，那么调用线程 sleep，否则直接返回;
  - futex_wake(address)：唤醒队列中的一个 thread（address 相同就是在一个队列中）
  - 在 lowlevellock.h 中，属于 nptl 库（GNU libc 库的一部分）

```
1   void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
4     if (atomic_bit_test_set (mutex, 31) == 0)
5       return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        /* We have to wait now. First make sure the futex value
13           we are monitoring is truly negative (i.e. locked). */
14        v = *mutex;
15        if (v >= 0)
16          continue;
17        futex_wait (mutex, v);
18    }
19  }
20
21  void mutex_unlock (int *mutex) {
22    /* Adding 0x80000000 to the counter results in 0 if and only if
23       there are not other interested threads */
24    if (atomic_add_zero (mutex, 0x80000000))
25      return;
26
27    /* There are other threads waiting for this mutex,
28        wake one of them up.  */
29    futex_wake (mutex);
30  }
```