```
#define CACHELINE 64
struct lock {
  // t-s, t-t-s, t-s-exp
  volatile unsigned int locked;

  // ticket
  volatile unsigned int next_ticket;
  volatile int now_serving;

  // anderson
  volatile struct {
    volatile int x;
    char cache_line[CACHELINE];
  } has_lock[100];
  volatile unsigned int queueLast;
  unsigned int holderPlace;
};

static inline unsigned int
TestAndSet(volatile unsigned int *addr)
{
  unsigned int result;
  unsigned int new = 1;

  // x86 atomic exchange.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (new) :
               "cc");
  return result;
}



/*
 * Test-and-Set (Anderson Table I)
 */
void
t_s_acquire(struct lock *lock)
{
  while(TestAndSet(&lock->locked) == 1)
    ;
}

void
t_s_release(struct lock *lock)
{
  lock->locked = 0;
}

```

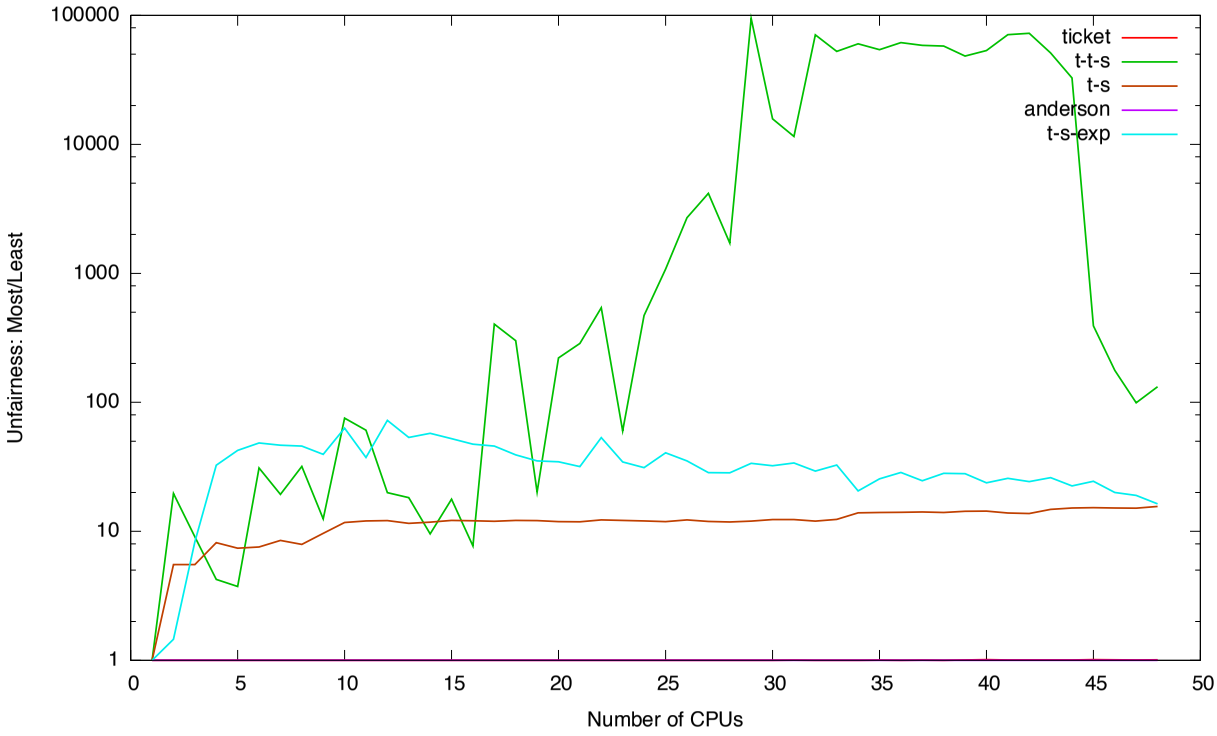


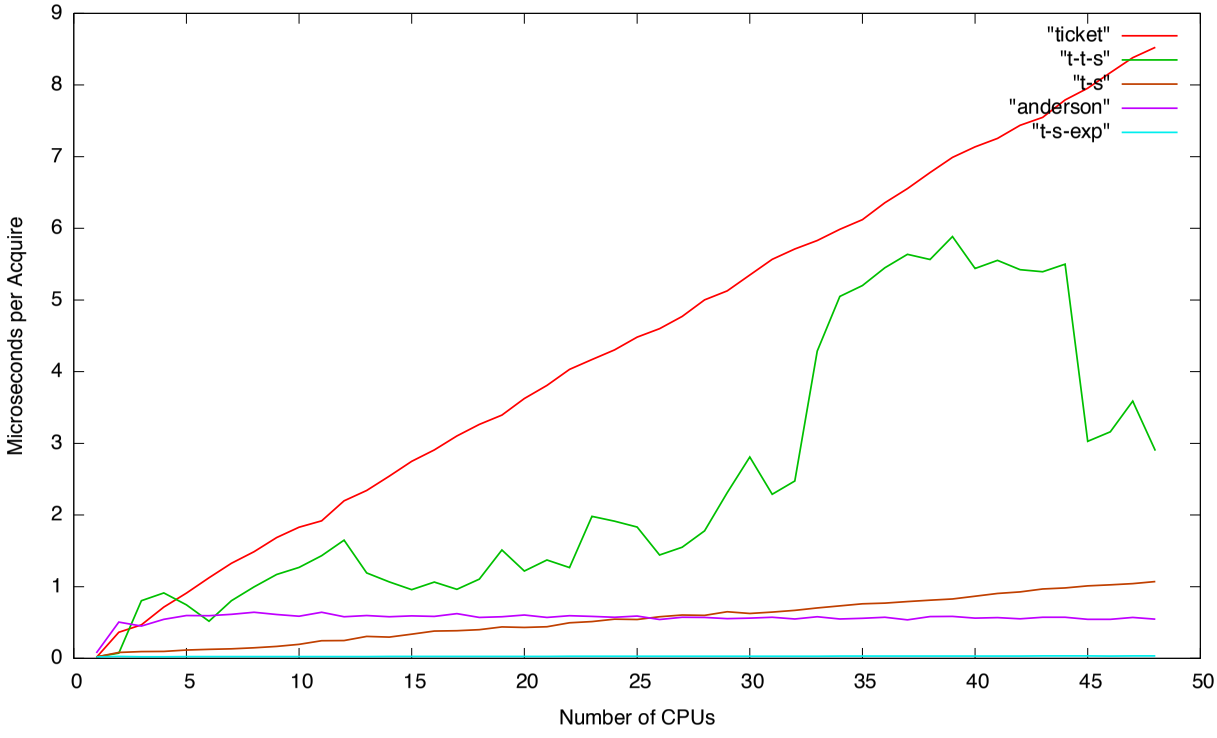
```
/*
 * Test-and-Test-and-Set (Anderson Table II)
 */
void
t_t_s_acquire(struct lock *lock)
{
  while(1){
    while(lock->locked)
      ;
    if(TestAndSet(&lock->locked) == 0)
      return;
  }
}



/*
 * Test-and-Test-and-Set with exponential delay
 * between each reference (Anderson Table IV).
 * Simplified -- no randomness.
 */
void
t_s_exp_acquire(struct lock *lock)
{
  int delay = 1;
  int i, junk = 0;
  volatile int junkjunk;

  while(lock->locked == 1 ||
        TestAndSet(&lock->locked) == 1){
    // delay
    int howlong = xrandom(delay);
    for(i = 0; i < howlong; i++)
      junk = junk * 3 + 1;
    // double the delay
    if(delay < 1000000)
      delay *= 2;
  }

  junkjunk = junk;
}

```

```
91  /*
92   * Atomically increment *p and return
93   * the previous value.
94   */
95  static __inline unsigned int
96  ReadAndIncrement(volatile unsigned int *p)
97  {
98      int v = 1;
99      __asm __volatile (
100     "   lock; xaddl   %0, %1 ;    "
101     : "+r" (v),
102       "=m" (*p)
103     : "m" (*p));
104
105     return (v);
106 }
107
108
109
110 /*
111  * Ticket Lock (Anderson Section IV.B 4th paragraph)
112  */
113 void
114 ticket_acquire(struct lock *lock)
115 {
116   int me = ReadAndIncrement(&lock->next_ticket);
117   while(lock->now_serving != me)
118     ;
119 }
120
121 void
122 ticket_release(struct lock *lock)
123 {
124   lock->now_serving += 1;
125 }
126
```



```
126 /*
127  * Queuing Lock (Anderson Table V)
128  */
129 void
130 anderson_acquire(struct lock *lock)
131 {
132   int myPlace = ReadAndIncrement(&lock->queueLast);
133   while(lock->has_lock[myPlace % numprocs].x == 0)
134     ;
135   lock->has_lock[myPlace % numprocs].x = 0;
136   lock->holderPlace = myPlace;
137 }
138
139 void
140 anderson_release(struct lock *lock)
141 {
142   int nxt = (lock->holderPlace + 1) % numprocs;
143   lock->has_lock[nxt].x = 1;
144 }
```


Unfairness: Most/Least
Number of CPUs
ticket
t-t-s
t-s
anderson
t-s-exp
100000
10000
1000
100
10
1
0
5
10
15
20
25
30
35
40
45
50


Microseconds per Acquire
Number of CPUs
"ticket"
"t-t-s"
"t-s"
"anderson"
"t-s-exp"
9
8
7
6
5
4
3
2
1
0
0
5
10
15
20
25
30
35
40
45
50