LFthreads: A lock-free thread library

Anders Gidenstam

PostDoc, AG1, Max-Planck-Institut für Informatik, Germany

Joint work with:

Marina Papatriantafilou

Distributed Computing and Systems group
Department of Computer Science and Engineering,
Chalmers University of Technology, Göteborg, Sweden

Outline

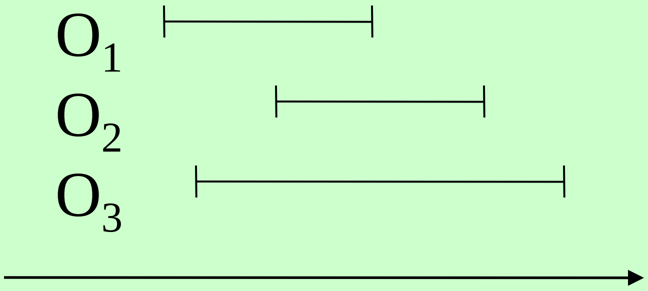
- Introduction
  - Lock-free synchronization
  - The Problem & Background
- LFthreads
  - Overview
  - Lock-free thread-blocking synchronization
- Experiments
- Conclusions

Synchronization on a shared object

- Lock-free synchronization
  - Concurrent operations without enforcing mutual exclusion
  - Avoids:
    - Blocking (or busy waiting), convoy effects and priority inversion
  - *Progress Guarantee*
    - At least one operation always makes progress
- Synchronization primitives
  - Built into CPU and memory system
    - Atomic read-modify-write (i.e. a critical section of one instruction)
  - Examples: Compare-and-Swap, Load-Linked / Store-Conditional

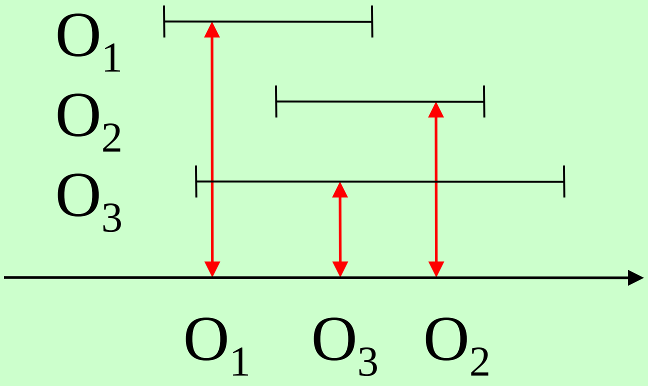
Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation there must be one single time instant during its duration where the operation appears to take effect.
    - The observed effects should be consistent with a sequential execution of the operations in that order.



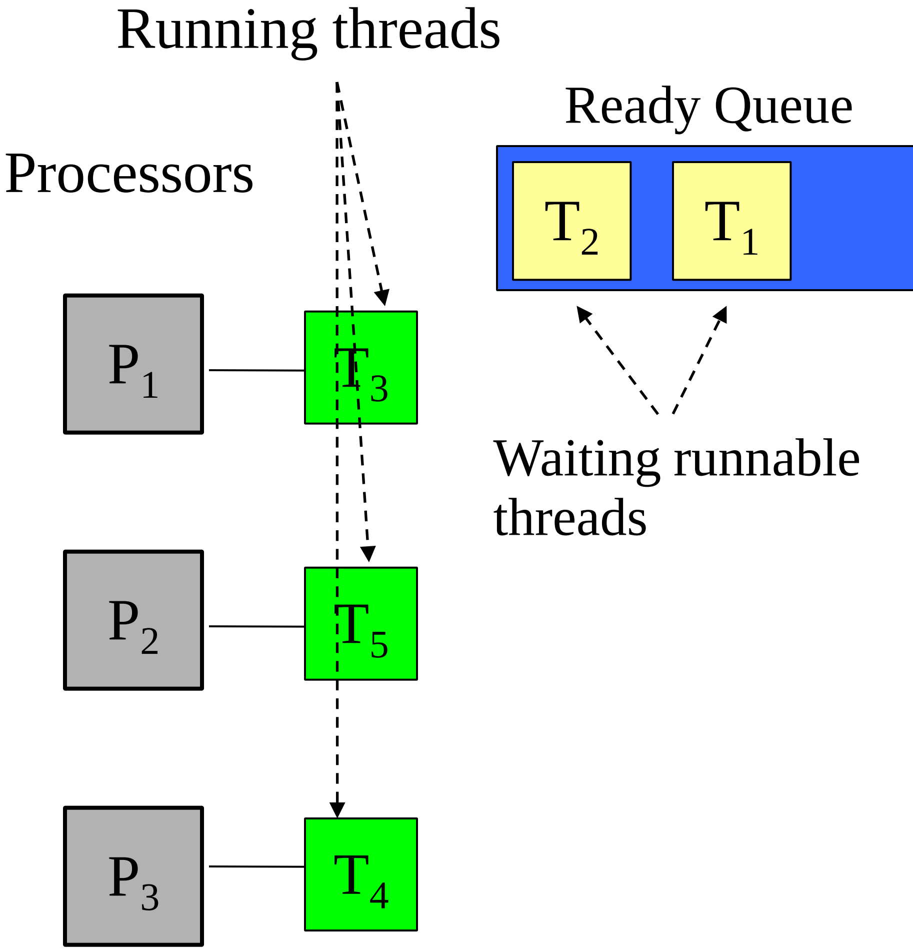
Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation there must be one single time instant during its duration where the operation appears to take effect.
    - The observed effects should be consistent with a sequential execution of the operations in that order.



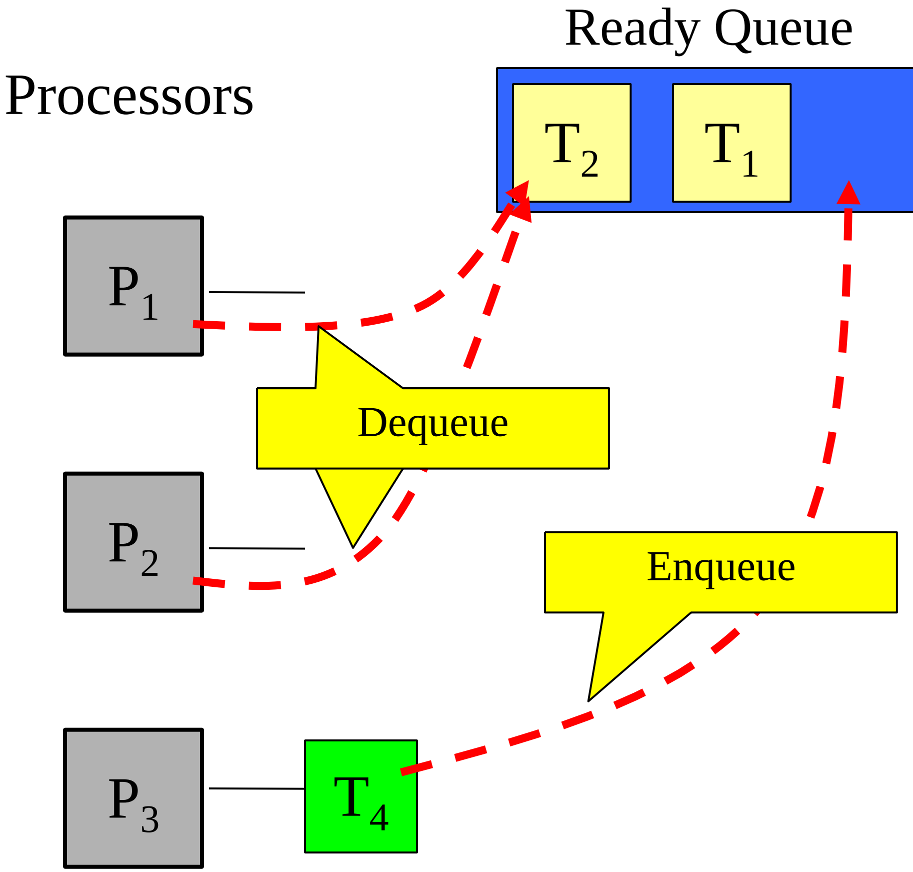
The Problem

- Multithreading on a multiprocessor
  - Thread multiplexing and scheduling
  - Thread synchronization objects
  - Aim: The POSIX pthread API implemented in a lock-free way
- Motivation: Why lock-free?
  - Processors should always be able to do useful work
    - In spite of others being slow: page faults, interrupts, h/w problems
  - Improved performance?



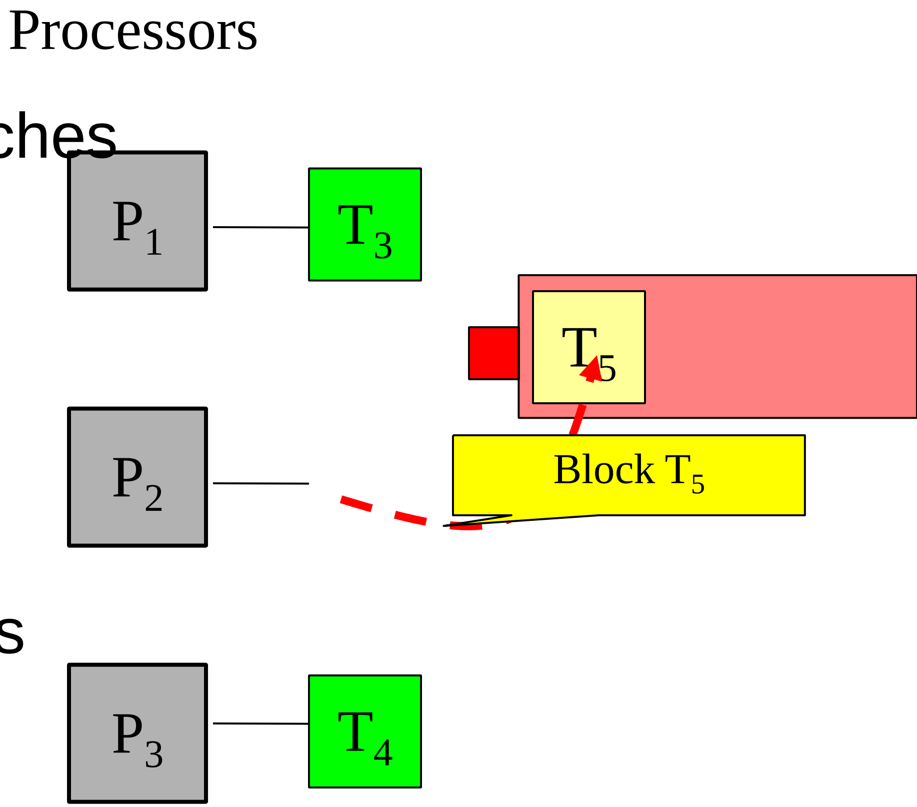
The Problem(s)

- Contention on the ready queue
  - Non-blocking work-stealing; Hood, [Blumhofe et al, 1994], [Blumhofe et al, 1998]
  - Lesser Bear, [Oguma et al, 2001]
  - In LFthreads: a lock-free queue



The Problem(s)

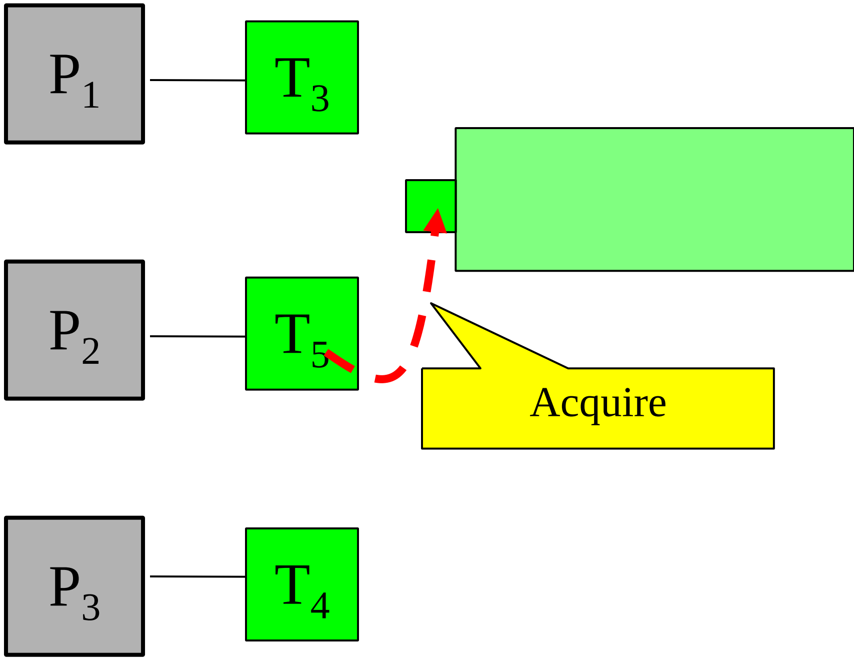
- Blocking synchronization
  - Often undesirable
    - Hinders progress
    - Expensive context switches
  - Sometimes required
    - Waiting for some event
    - Legacy applications
  - Goal in LFthreads:
    - Lock-free for processors
    - Resilience to slow operations/processors



The Problem(s)

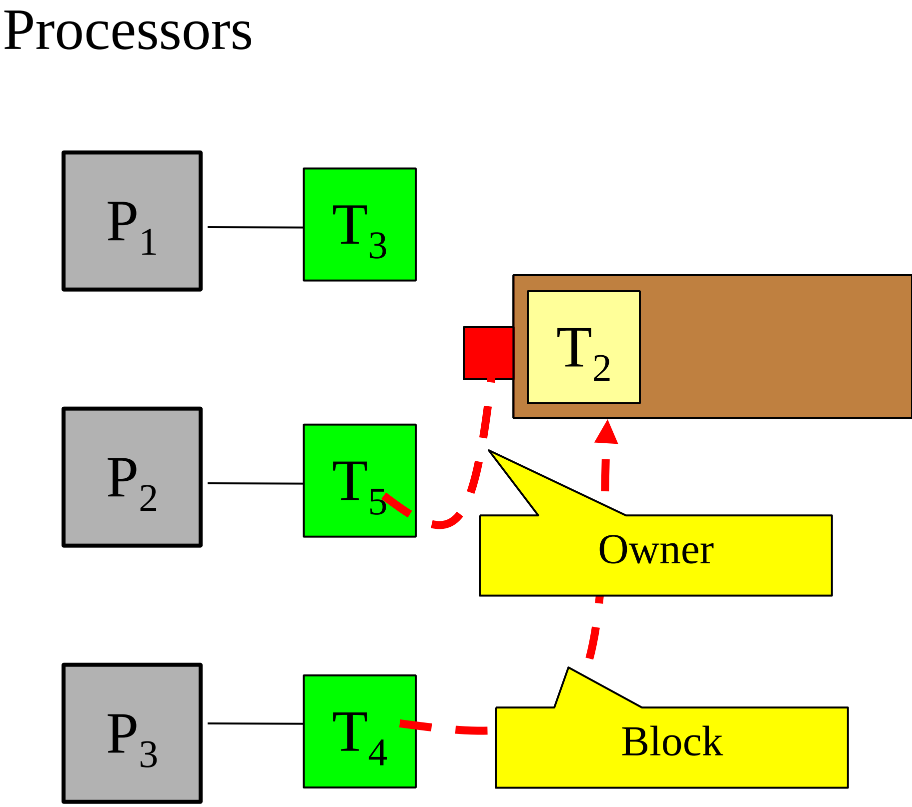
- Implementing blocking synchronization
  - Keep common case fast
    - Often no contention
    - User / kernel level split implementation
      - E.g. Linux, Solaris
  - Critical sections are short
    - Spinning v.s. blocking [Zahorjan et al, 1991]

Processors



The Problem(s)

- Implementing blocking synchronization
  - Keep common case fast
    - Often no contention
    - User / kernel level split implementation
      - E.g. Linux, Solaris
  - Critical sections are short
    - Spinning v.s. blocking [Zahorjan et al, 1991]
  - Enlisting the scheduler
    - [Devi et al, 2006]
    - [Kontothanassis et al, 1997]

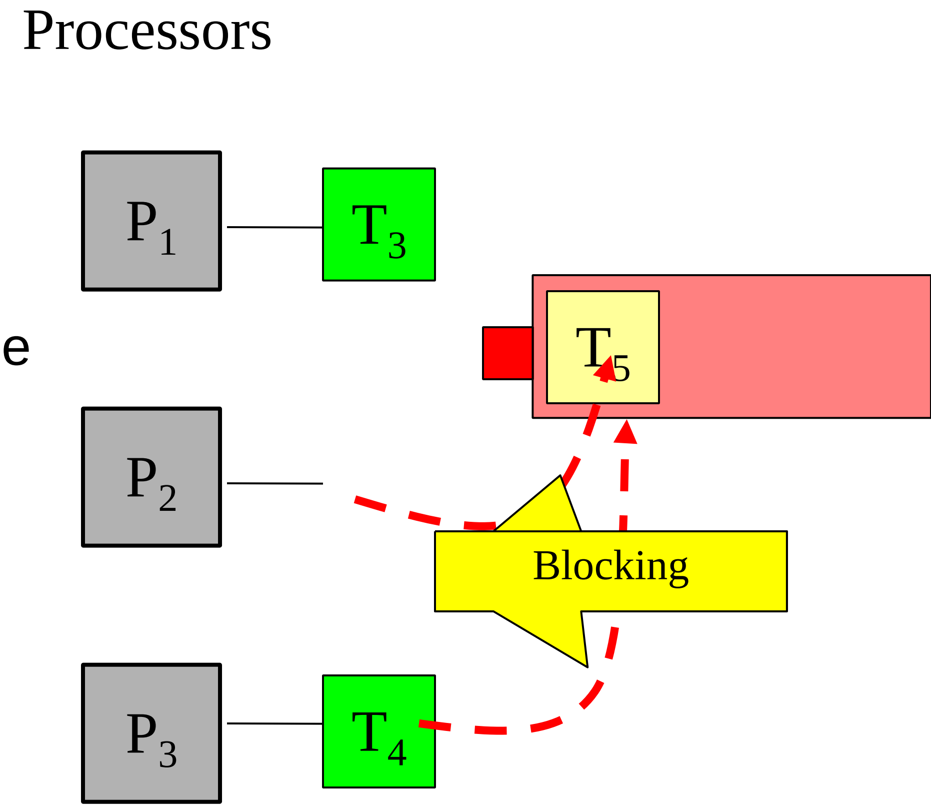


The Problem(s)

- Implementing blocking synchronization
  - There can be concurrent operations – synchronization needed
    - Mutual exclusion?
    - A slow processor could force others to spin
    - Slow: e.g. page faults, interrupts, h/w problems

Kernel partial solution:

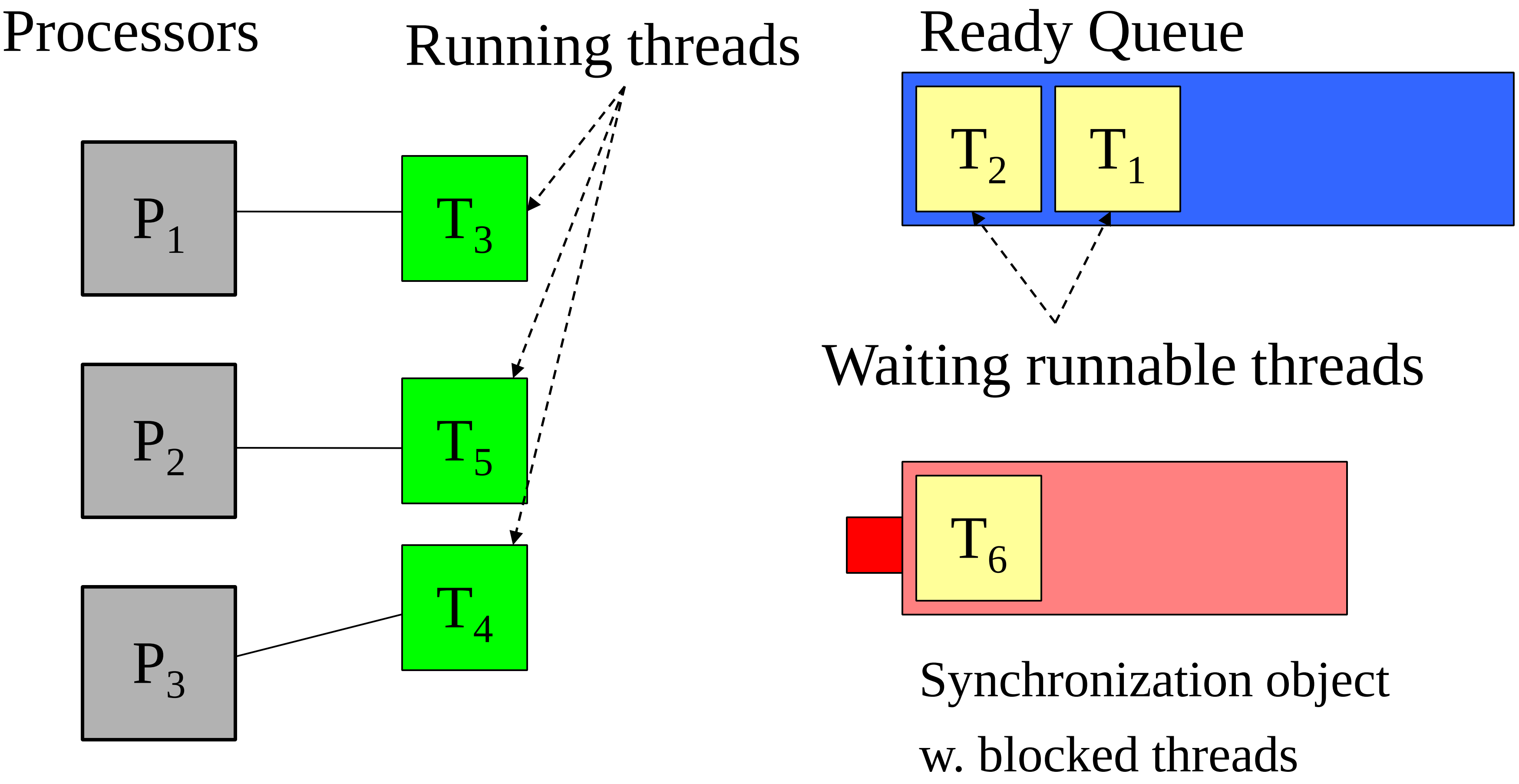
- spin lock + disable IRQ



More Related Work

- Lock-free threading
  - Hood. R. Blumofe, D. Papadopoulos, 1998.
  - Lesser Bear. H. Oguma, Y. Nakayama, 2001.
- Lock-free operating systems kernels
  - Synthesis. H. Massalin, 1992.
  - Cache Kernel. M. Greenwald, D. Cheriton, 1999.
  - A. Gavare, P. Tsigas, 2005.

LFthreads – system overview



Thread synchronization objects

- Mutual exclusion object – mutex
  - Two states: unlocked / locked
  - Three operations for threads
    - lock(m) - Locks m. If m is already locked the thread is blocked.
    - trylock(m) - Tries to lock m. Returns false if m is already locked.
    - unlock(m) - Unlocks m.

Mutex implementation

- “Typical” mutex implementation

```
type mutex_t is record
state : enum (UNLOCKED, LOCKED);
waiting : Queue of thread_t;
slock : spin_lock_t;
```

- Note: The spin-lock protects the mutex_t record from concurrent updates by different processors
  - Operations cannot overlap
  - Processors might be forced to spin waiting for others

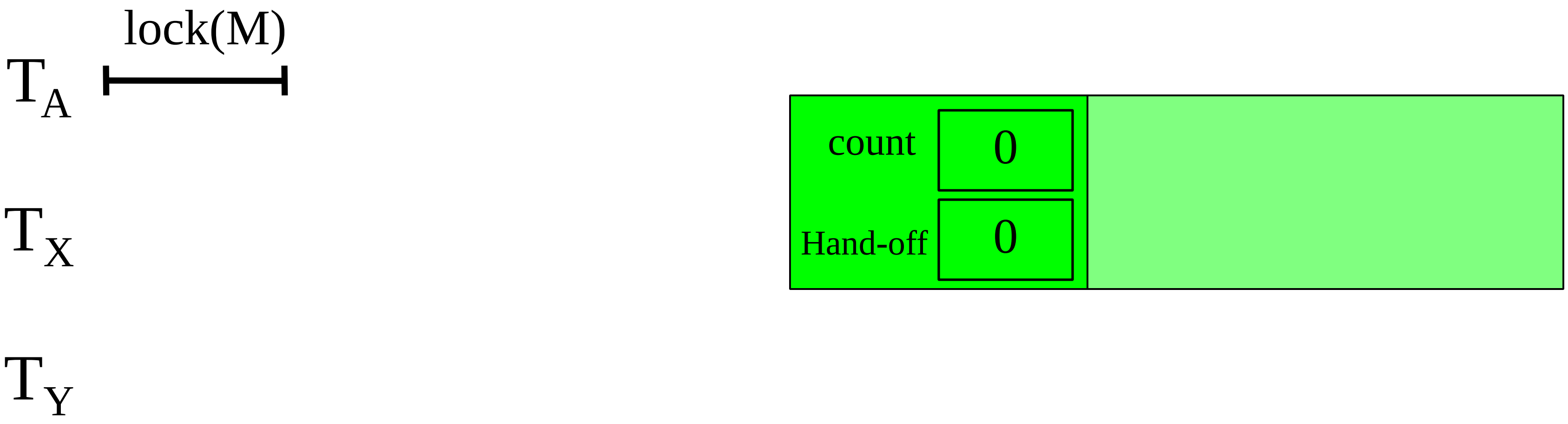
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

waiting : *Lock-free Queue* of thread_t;

hand-off : integer;



lock(M)

T_A

T_X

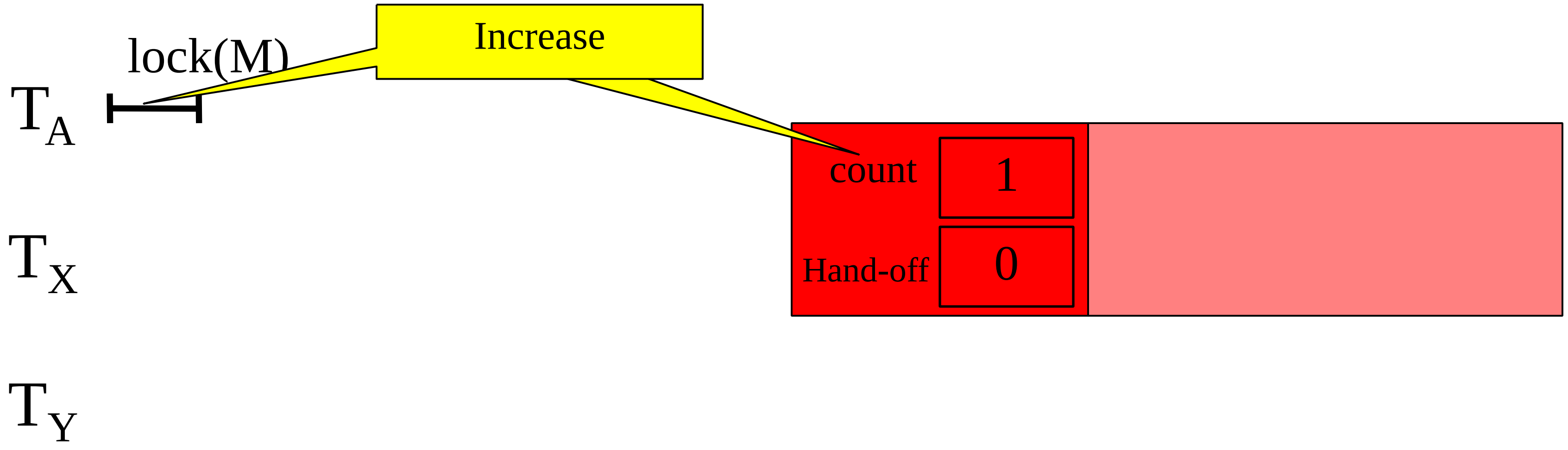
T_Y

count 0

Hand-off 0

Lock-free mutex implementation
The Hand-off method

```
type mutex_t is record
count : integer;
waiting : Lock-free Queue of thread_t;
hand-off : integer;
```



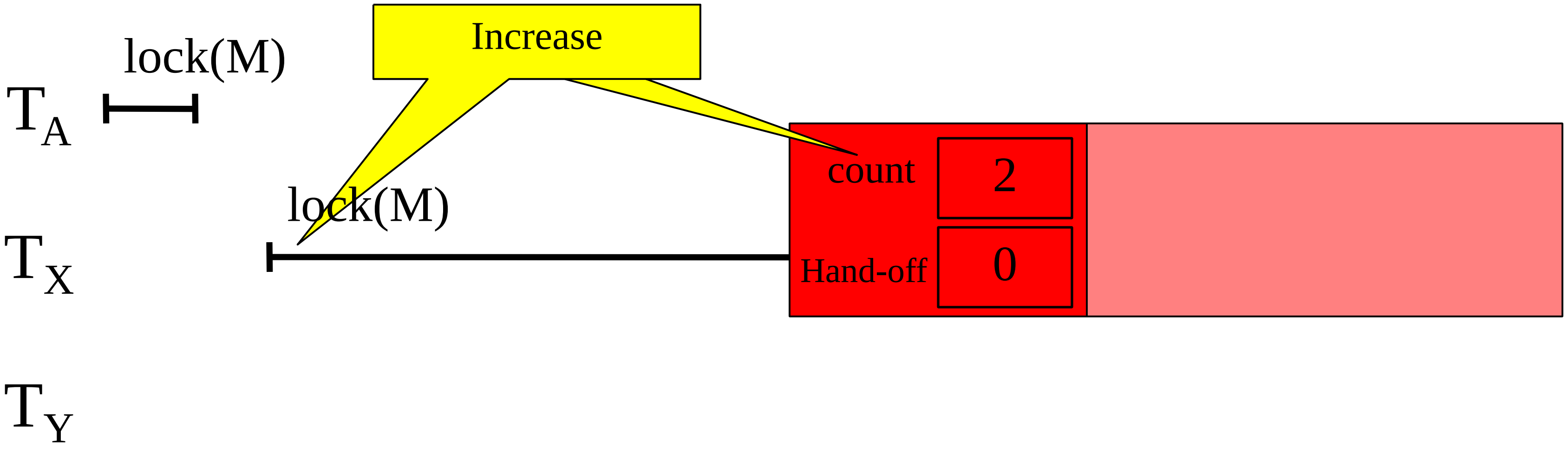
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



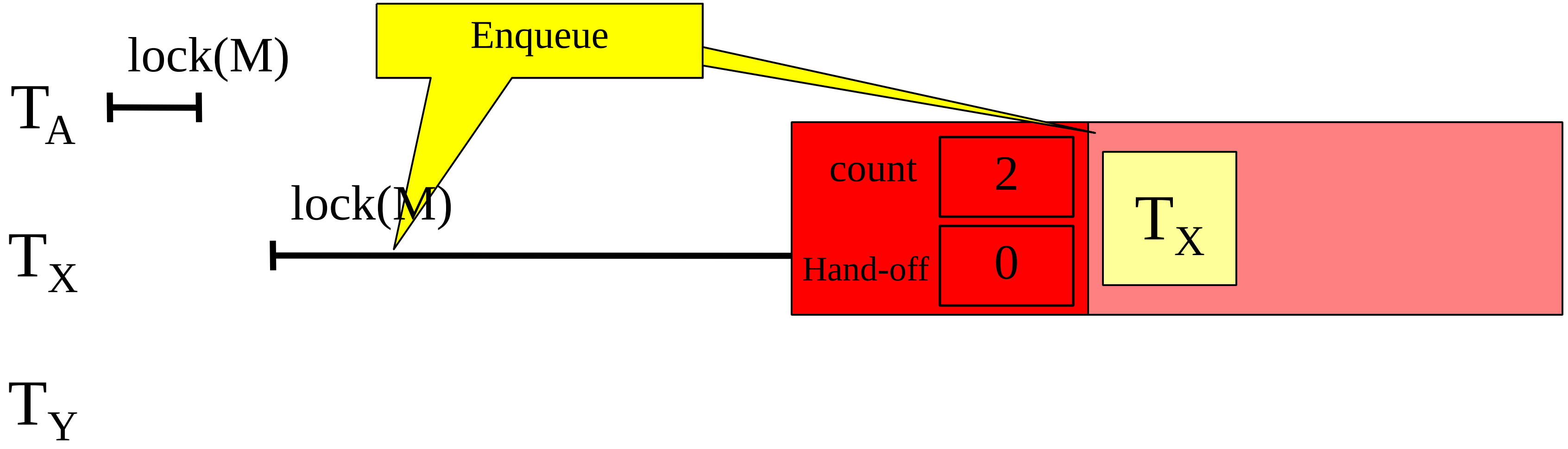
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



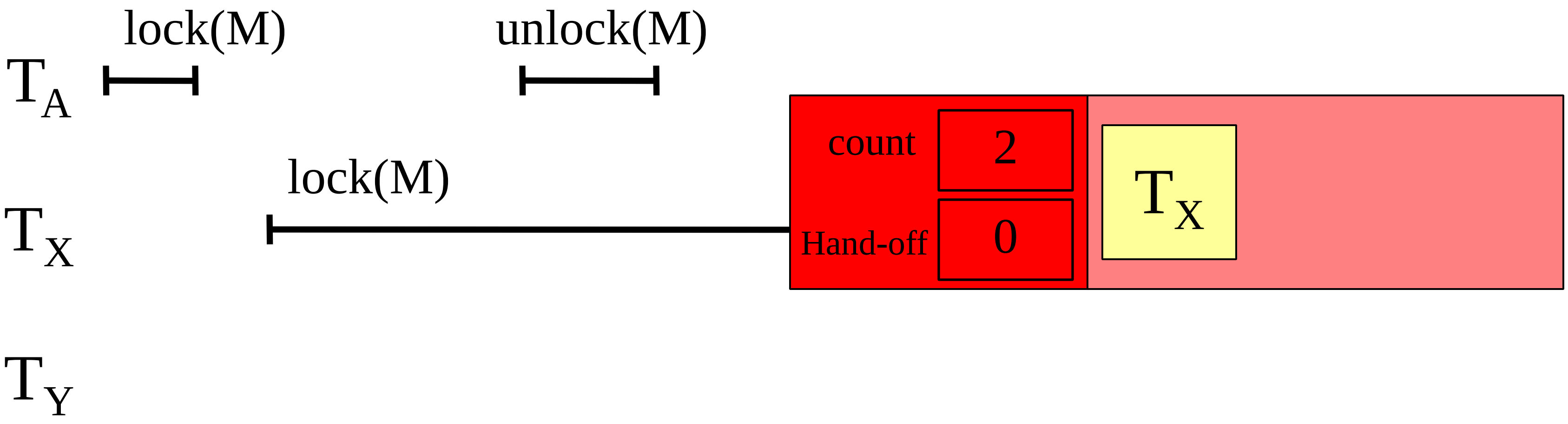
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



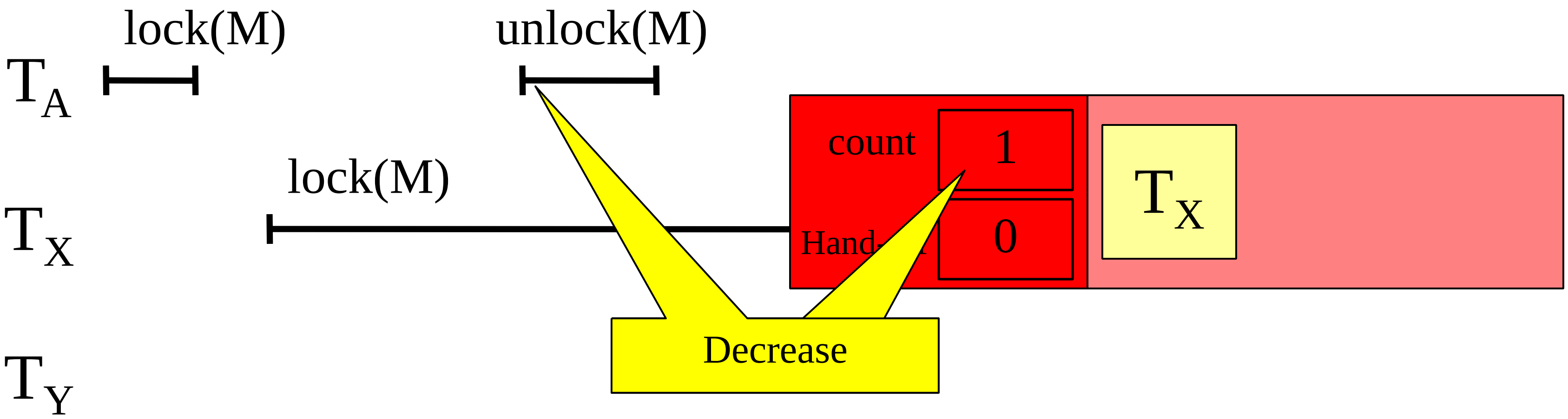
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



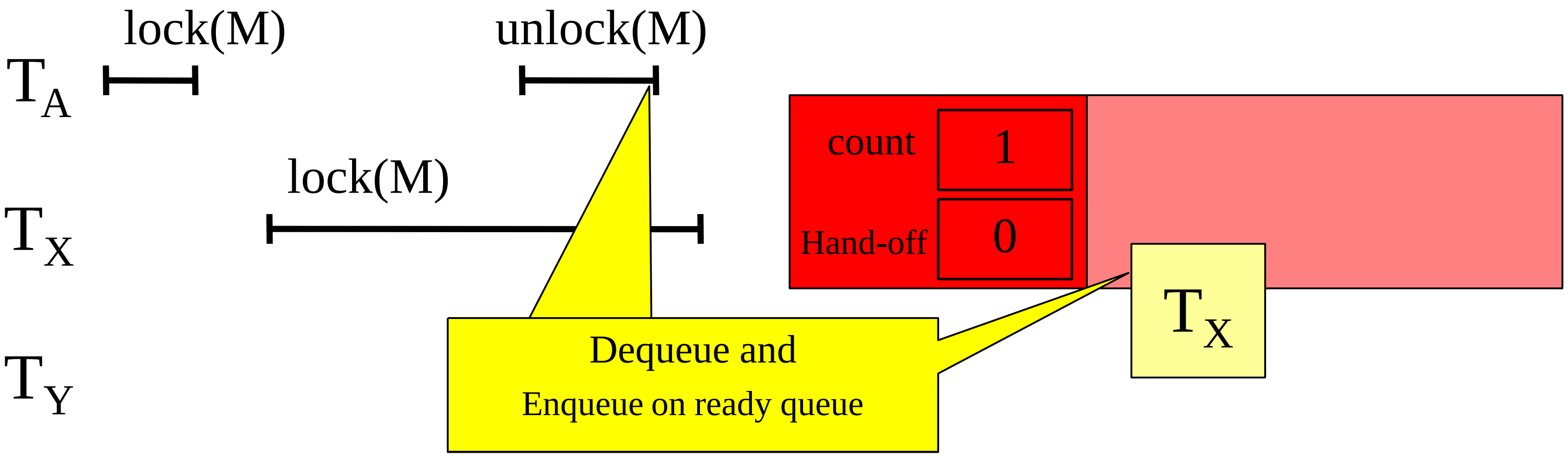
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



Dequeue and

Enqueue on ready queue

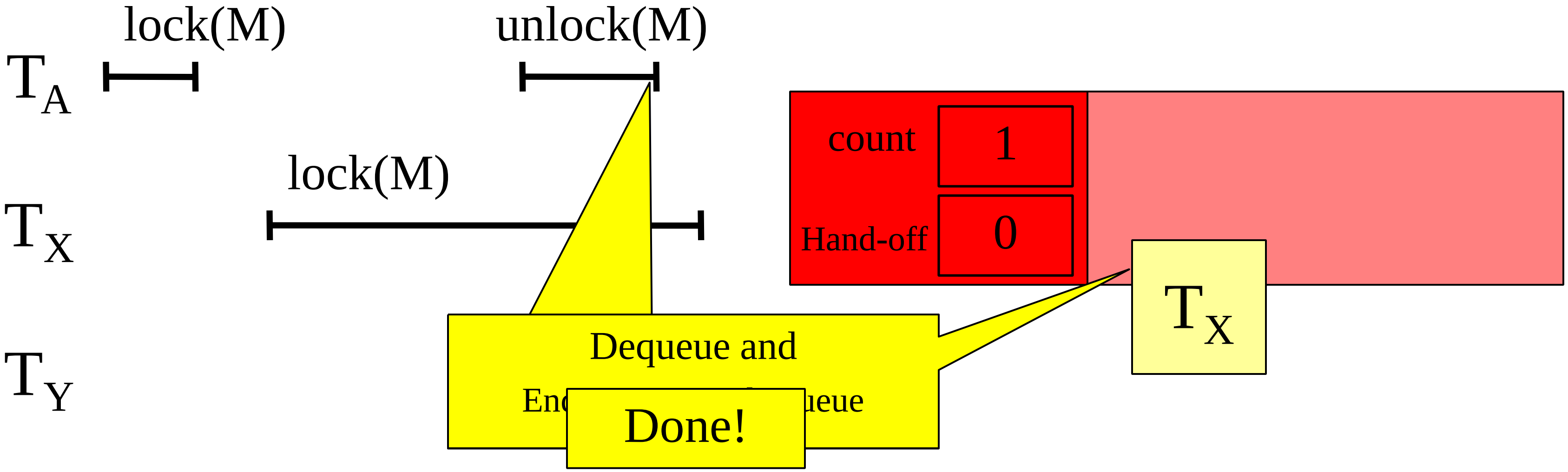
Lock-free mutex implementation
The Hand-off method

type mutex_t **is record**

count : integer;

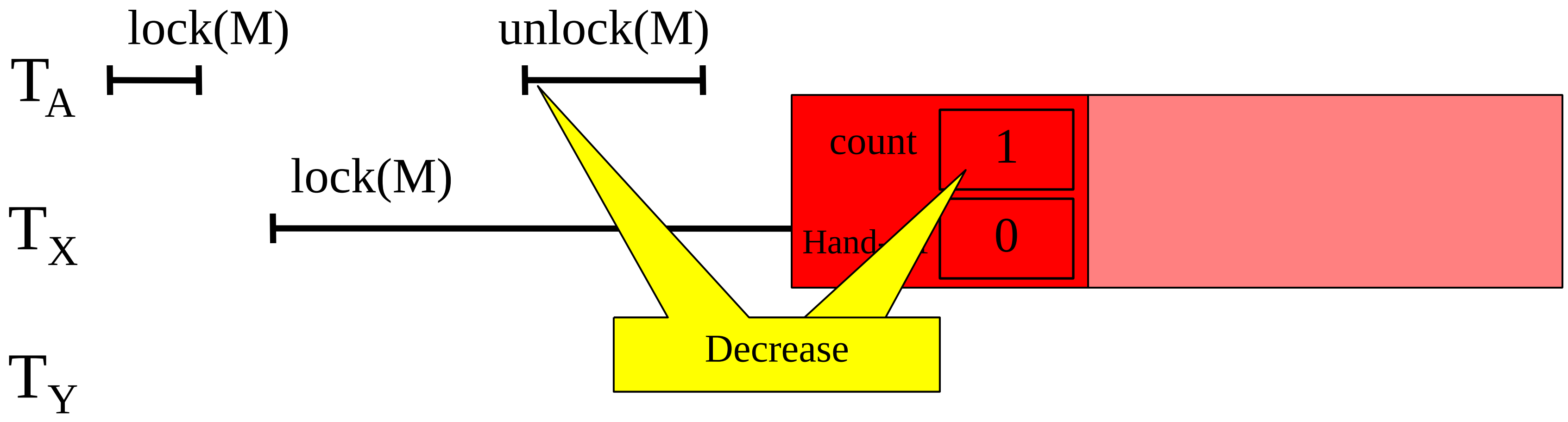
waiting : Lock-free Queue of thread_t;

hand-off : integer;



Lock-free mutex implementation: Slow lock (i)

```
type mutex_t is record
count : integer;
waiting : Lock-free Queue of thread_t;
hand-off : integer;
```



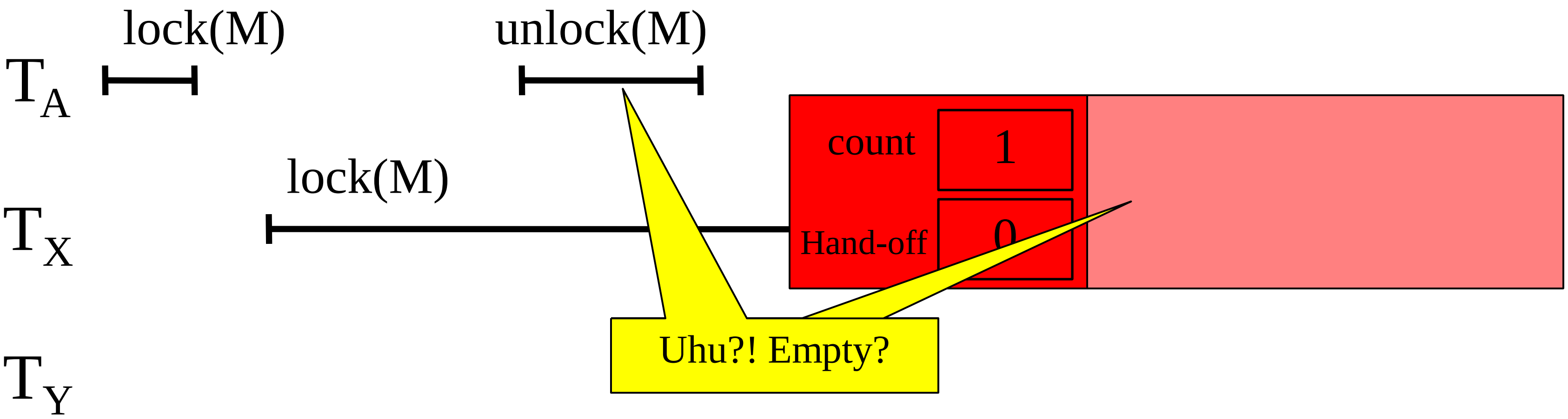
Lock-free mutex implementation: Slow lock (i)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



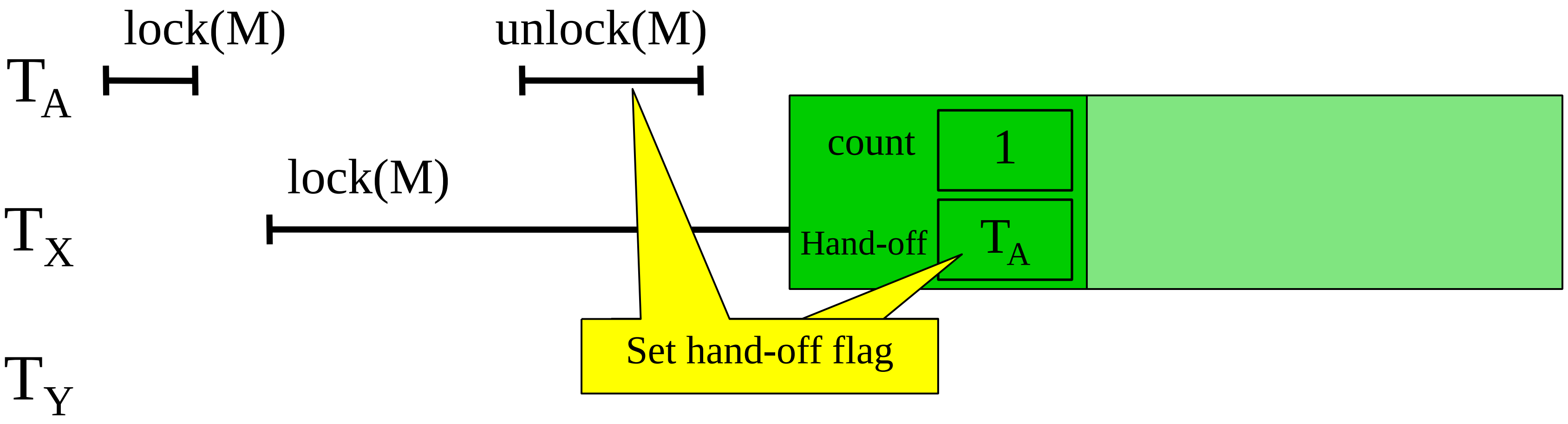
Lock-free mutex implementation: Slow lock (i)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



Set hand-off flag

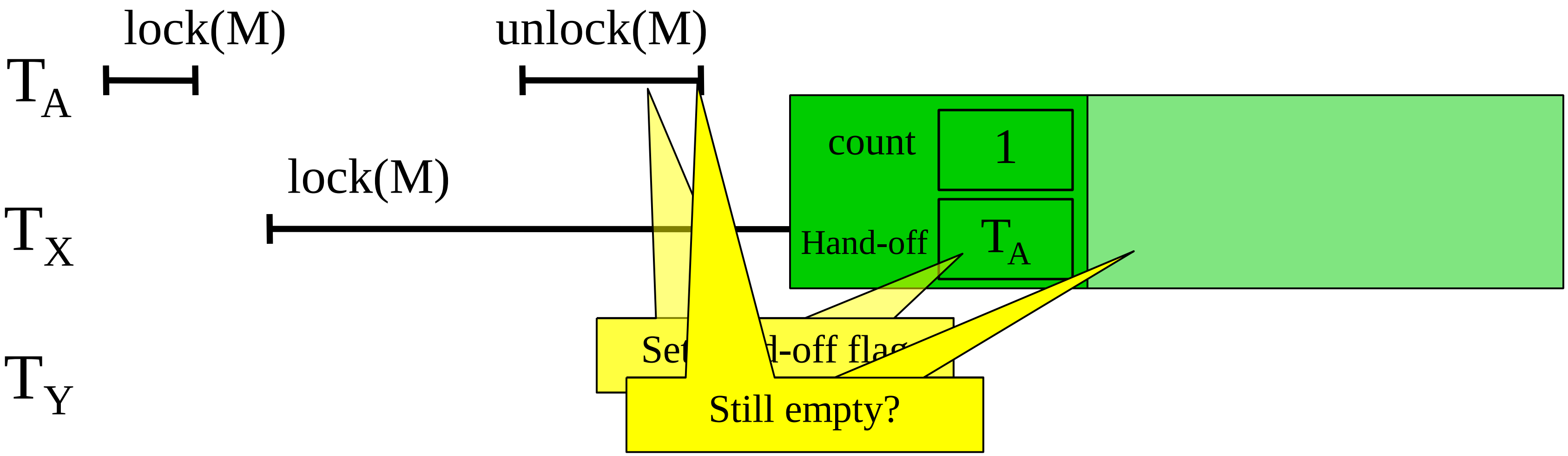
Lock-free mutex implementation: Slow lock (i)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



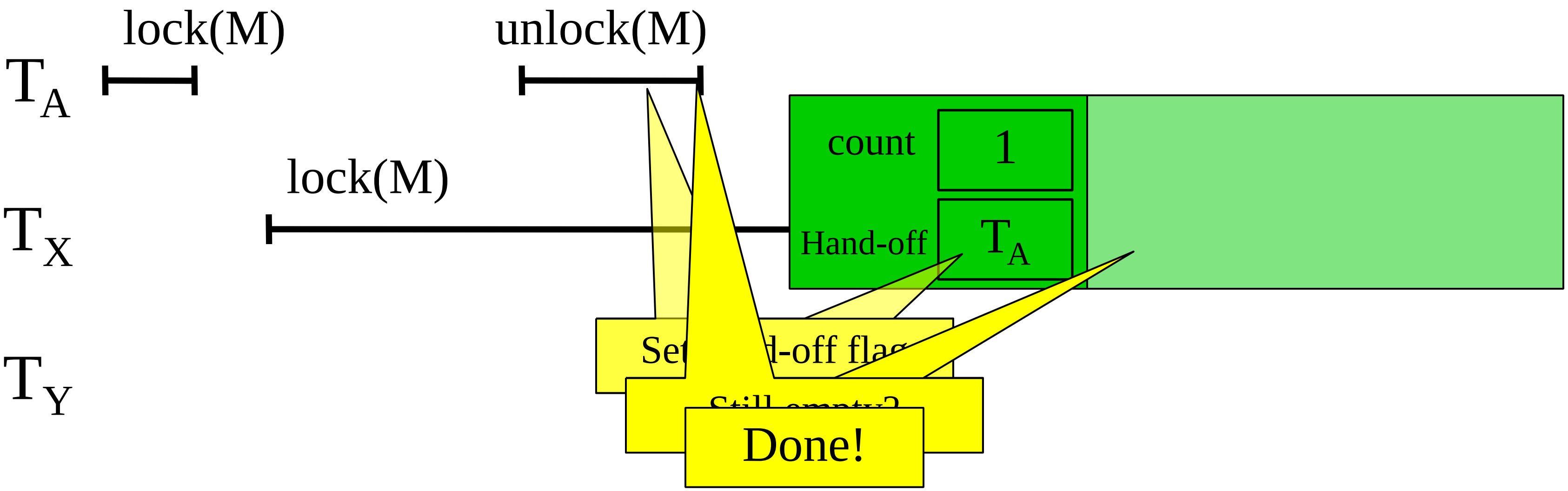
Lock-free mutex implementation: Slow lock (i)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



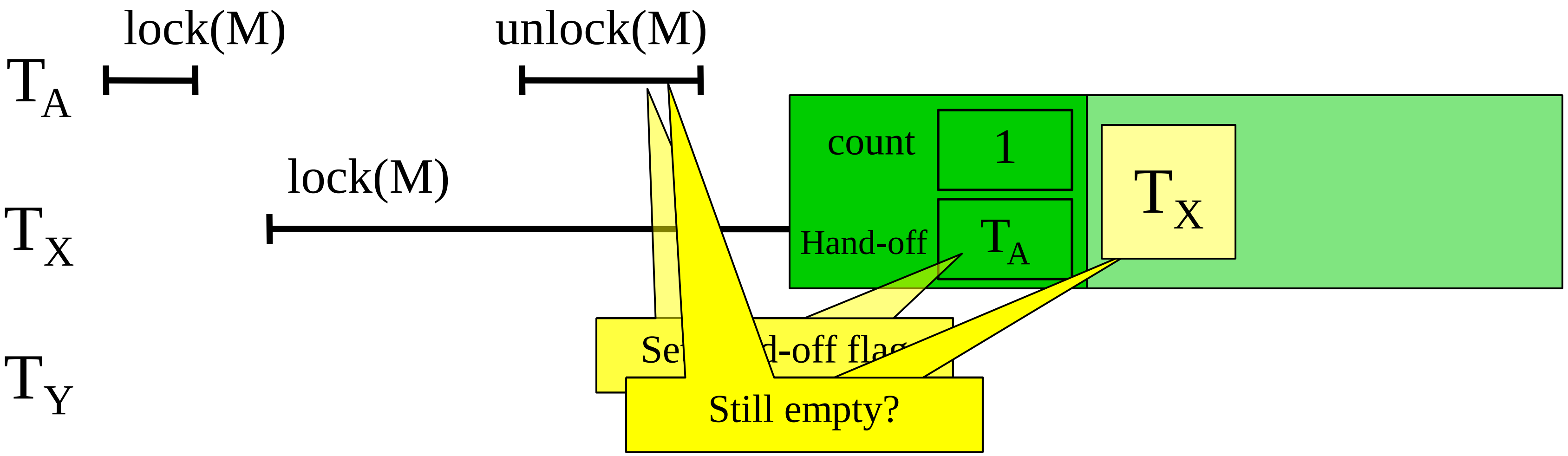
Lock-free mutex implementation: Slow lock (ii)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



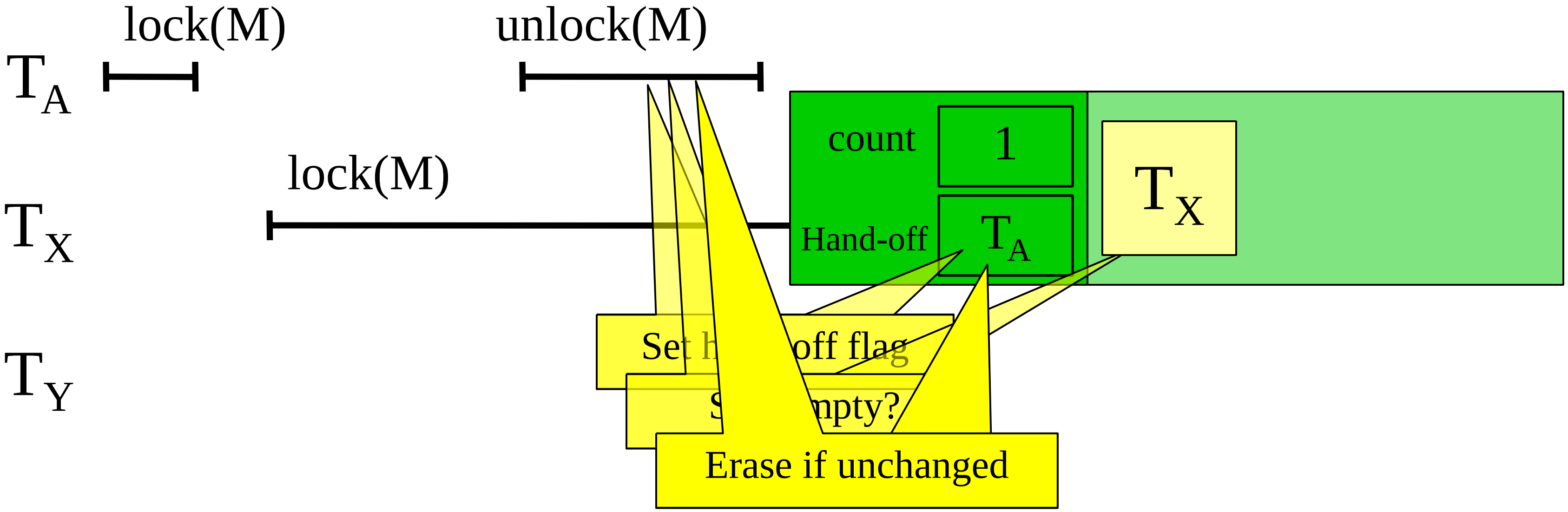
Lock-free mutex implementation: Slow lock (ii)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



lock(M)

unlock(M)

lock(M)

T_A

T_X

T_Y

count

1

Hand-off

T_A

T_X

Set h off flag

S mpty?

Erase if unchanged

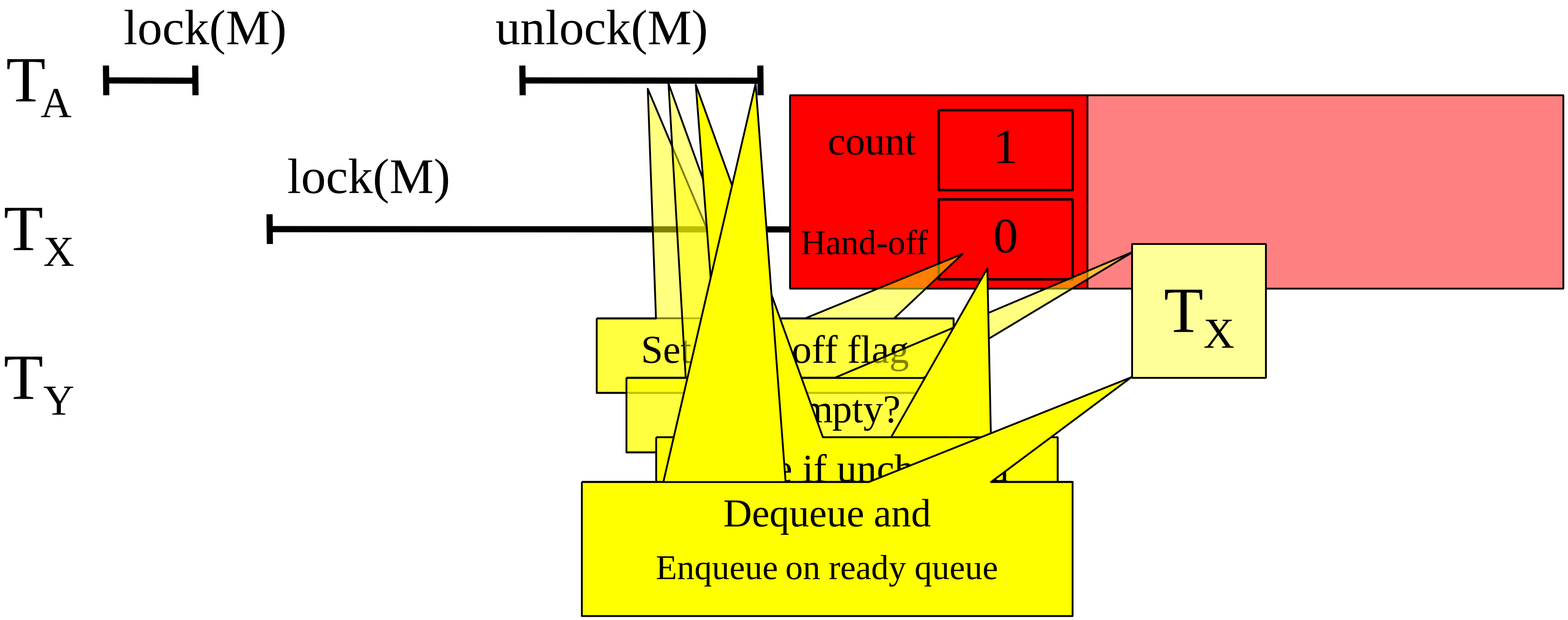
Lock-free mutex implementation: Slow lock (ii)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



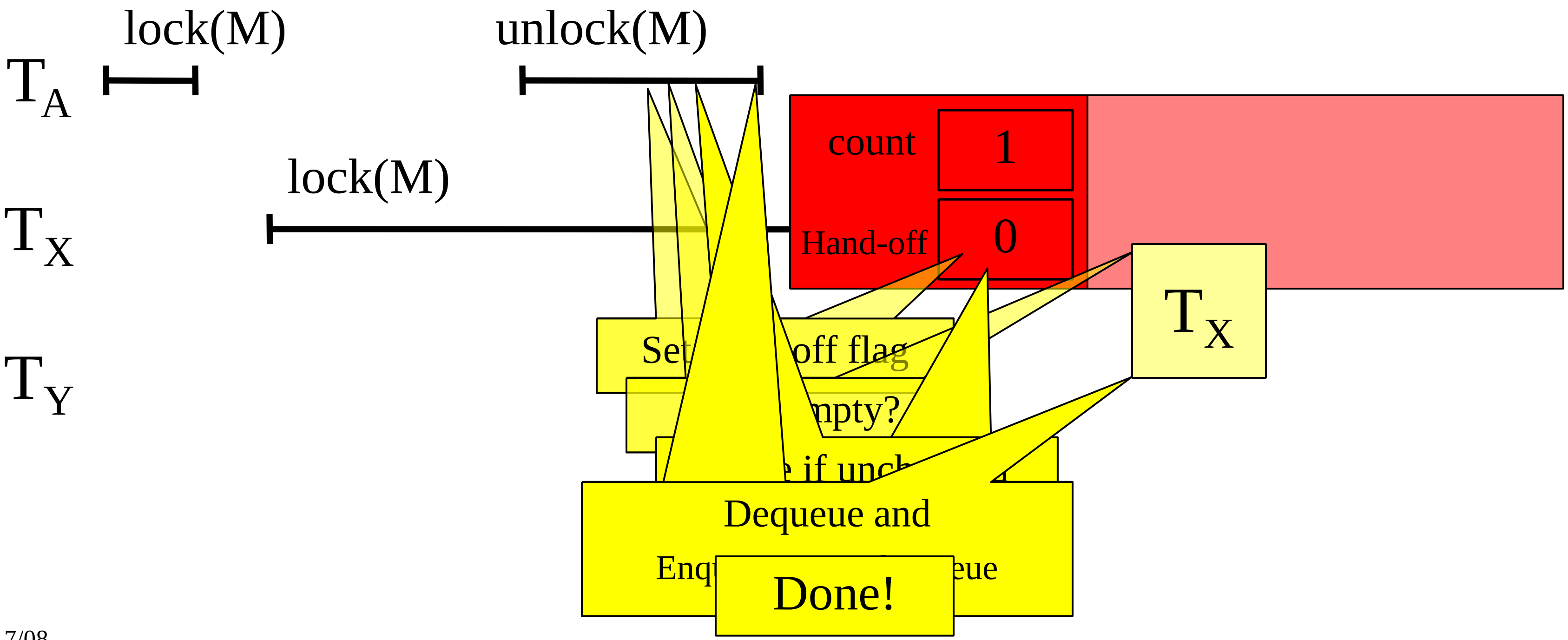
Lock-free mutex implementation: Slow lock (ii)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



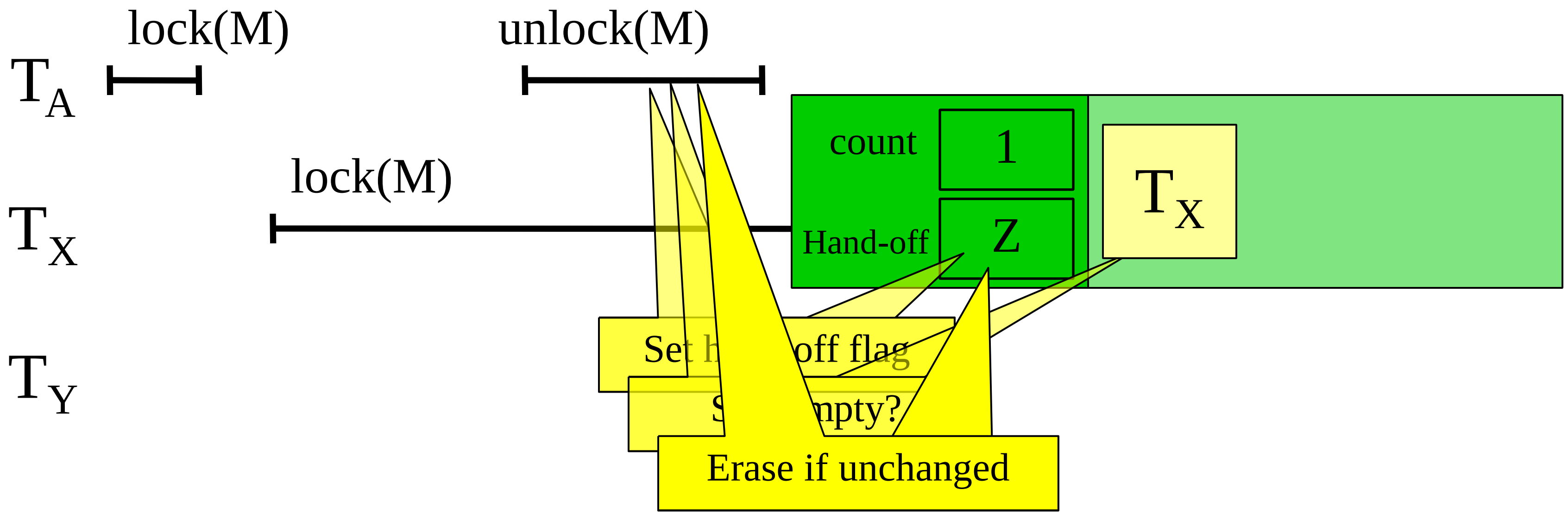
Lock-free mutex implementation: Slow lock (iii)

type mutex_t **is record**

count : integer;

waiting : Lock-free Queue of thread_t;

hand-off : integer;



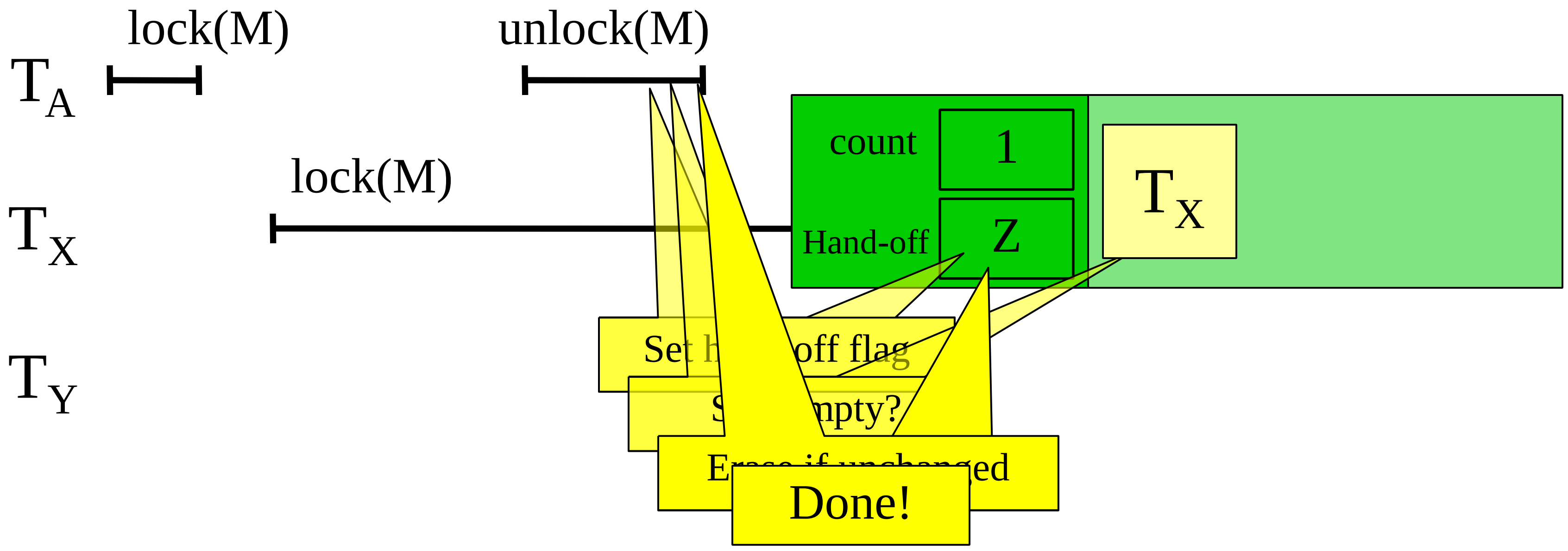
Lock-free mutex implementation: Slow lock (iii)

type mutex_t **is record**

count : integer;

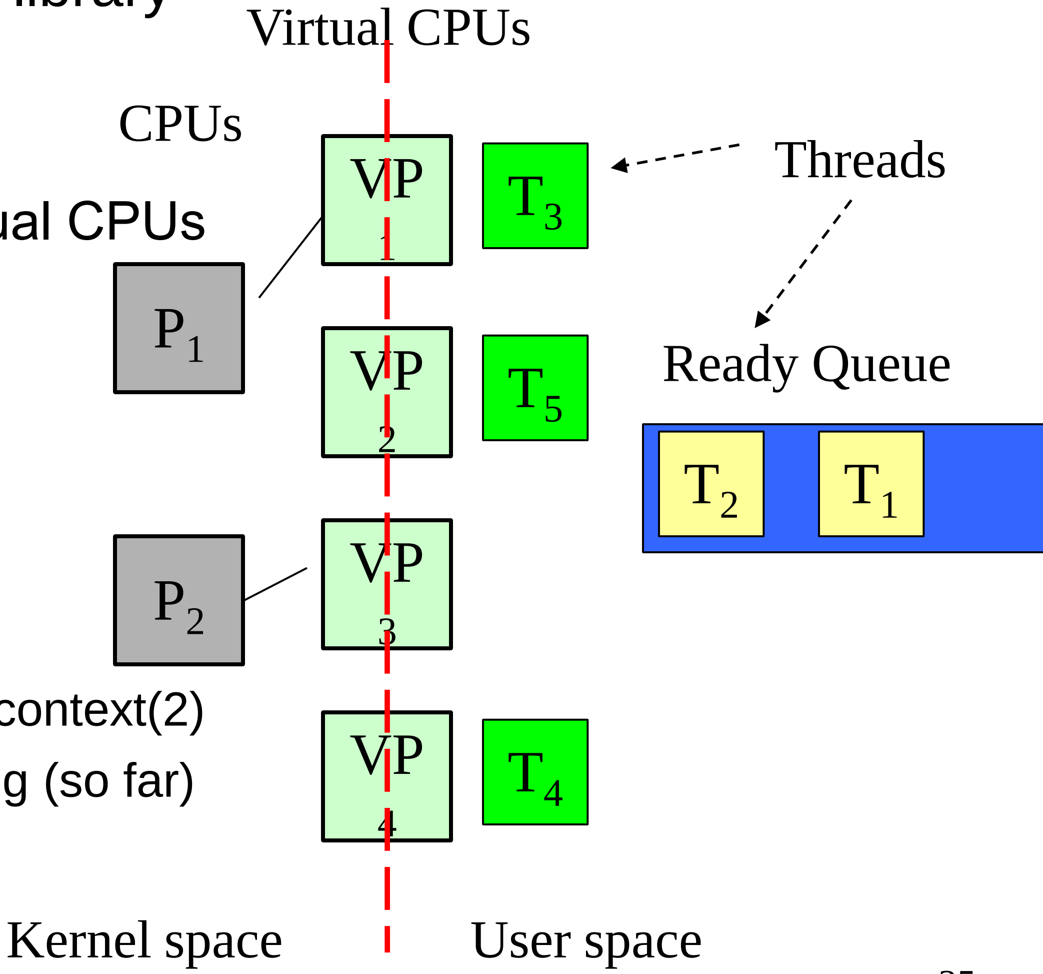
waiting : Lock-free Queue of thread_t;

hand-off : integer;



Proof-of-concept implementation

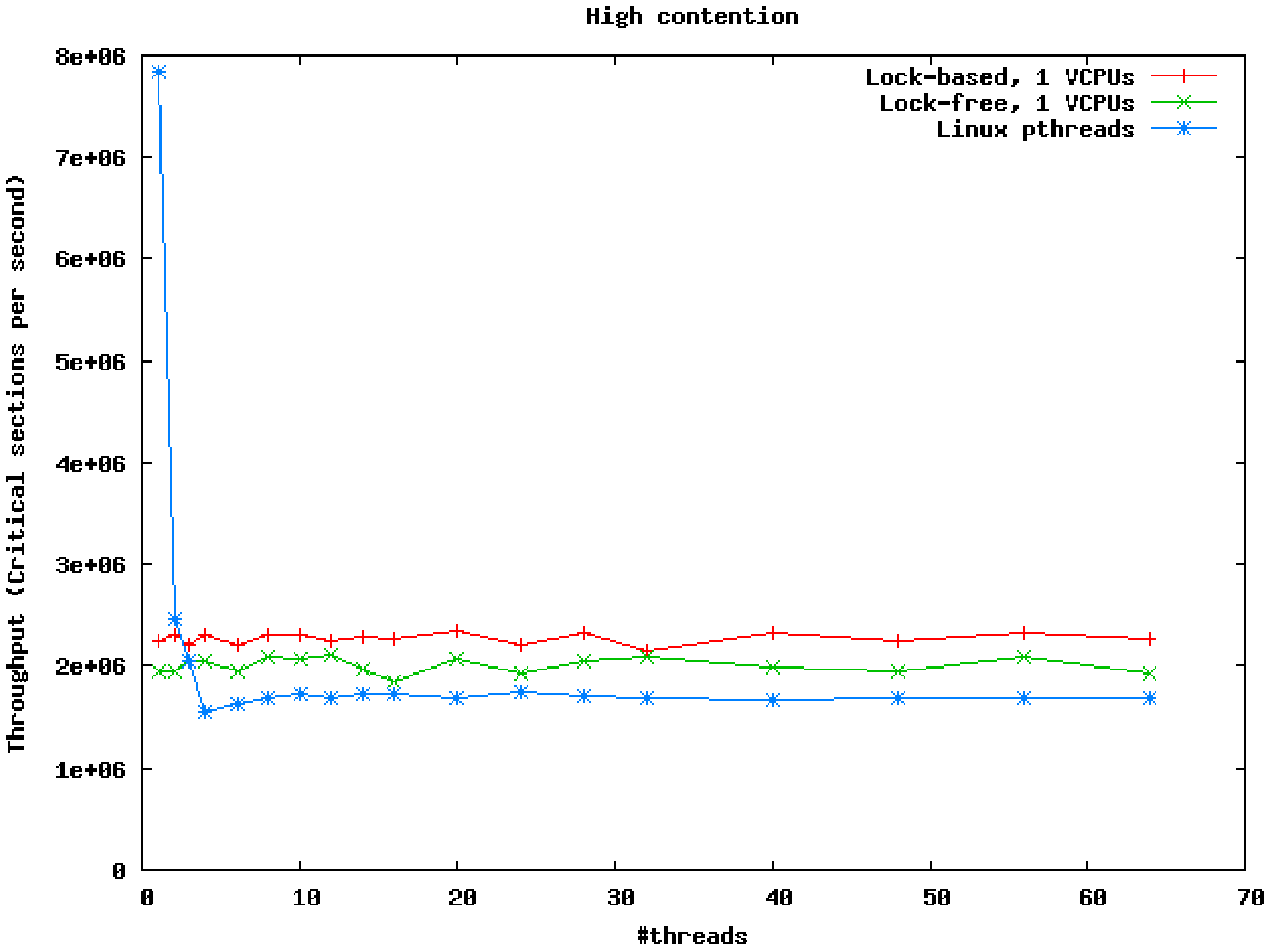
- LFthreads user-level thread library
  - Linux/IA32 platform
  - “Cloned” processes as virtual CPUs
    - Share the same address space, file descriptor table etc
  - User-level thread contexts based on
    - SUSv2 setcontext(2) / getcontext(2)
    - Non-preemptive scheduling (so far)
  - POSIX compliance a goal



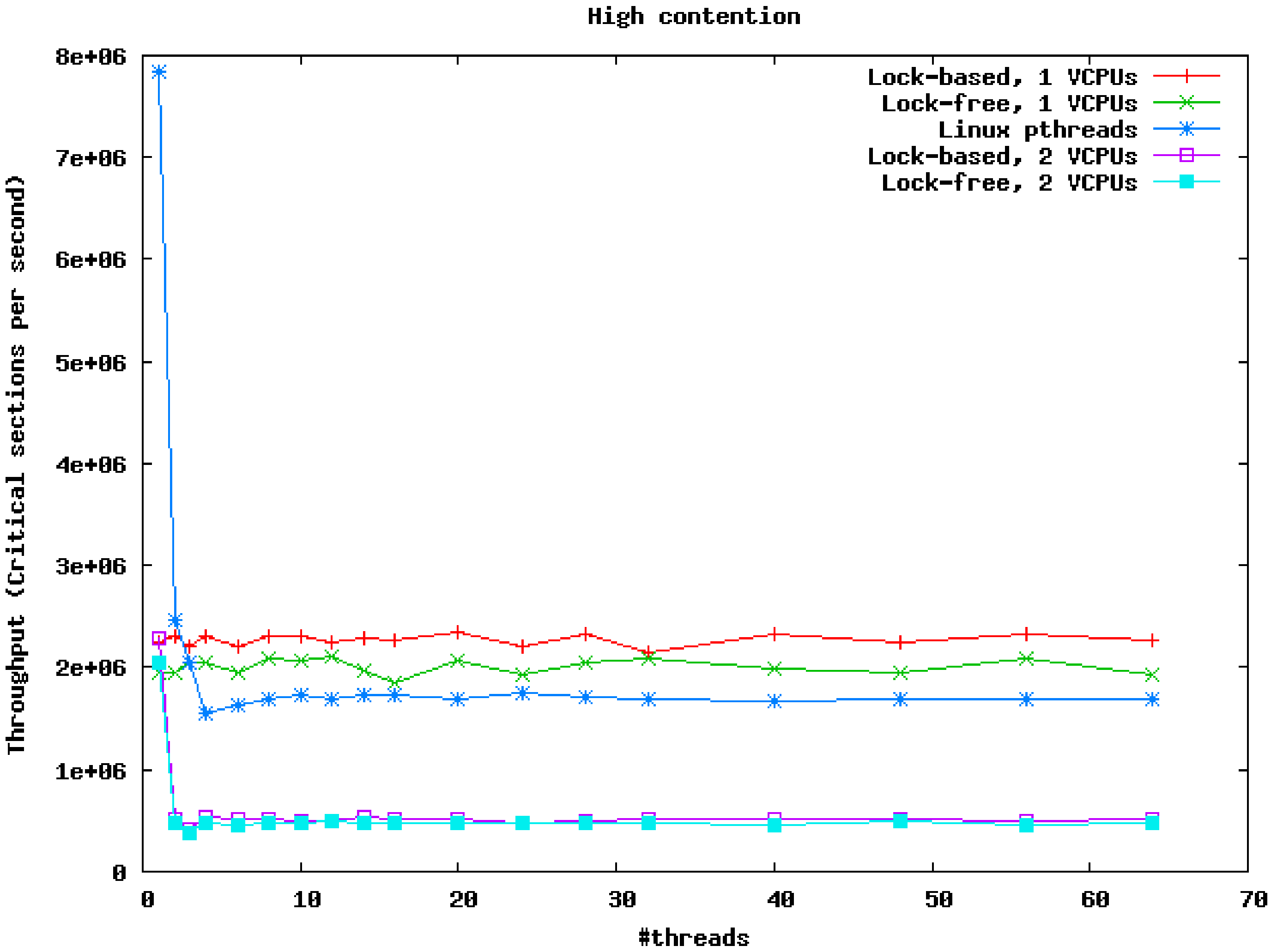
Experimental evaluation

- Micro benchmark
  - Threads competing for a critical section
    - High contention
      - Work: 1 -1
    - Low contention
      - Work: 1 - 1000
  - Configurations
    - LFthreads with 1, 2, 4, 8,16 virtual CPUs
      - lock-free mutex
      - spin-lock-based mutex
    - Platforms standard pthreads (2.6.x kernel)
  - 2x dual AMD Opteron processors

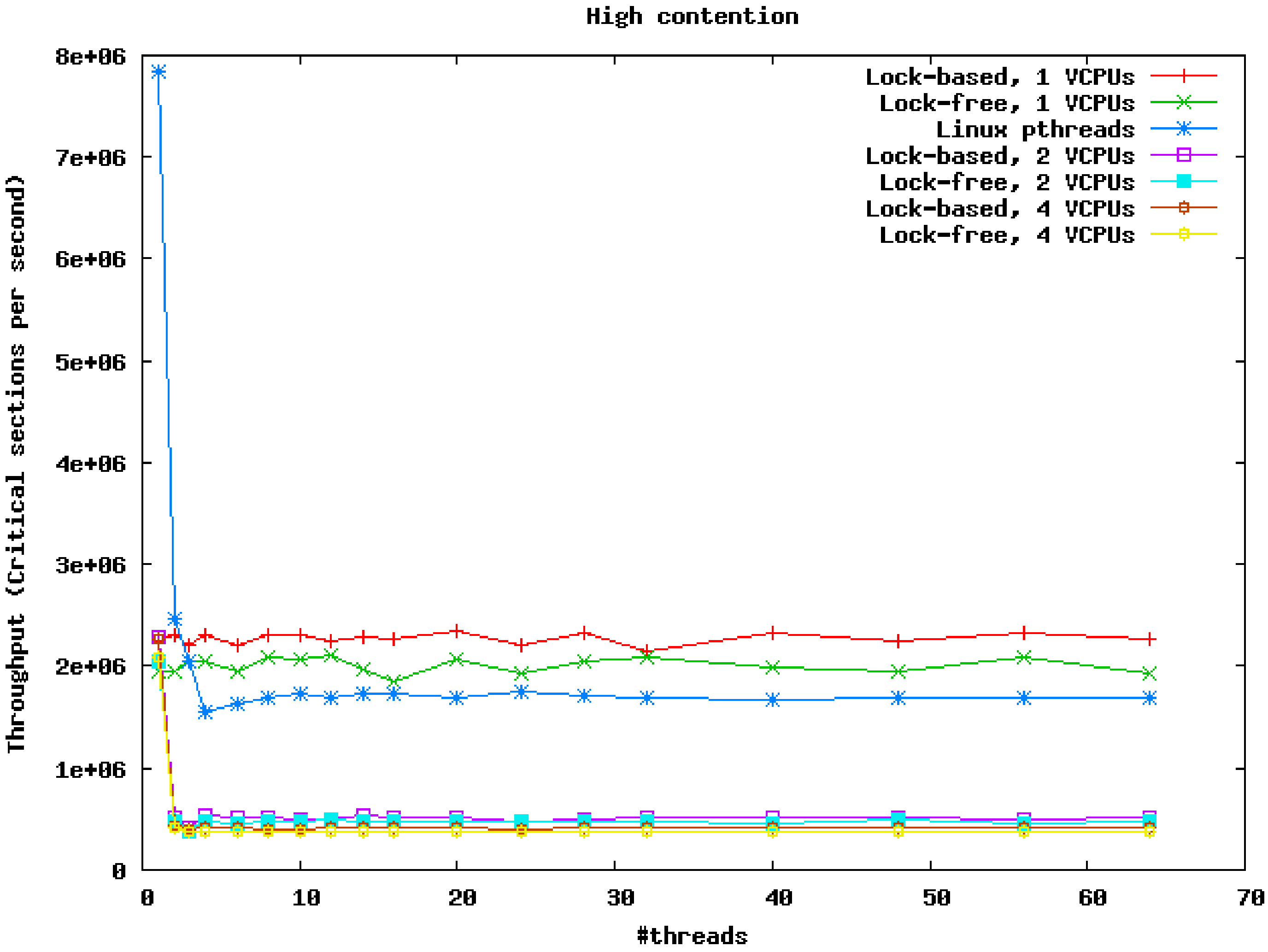
Experimental evaluation (i)



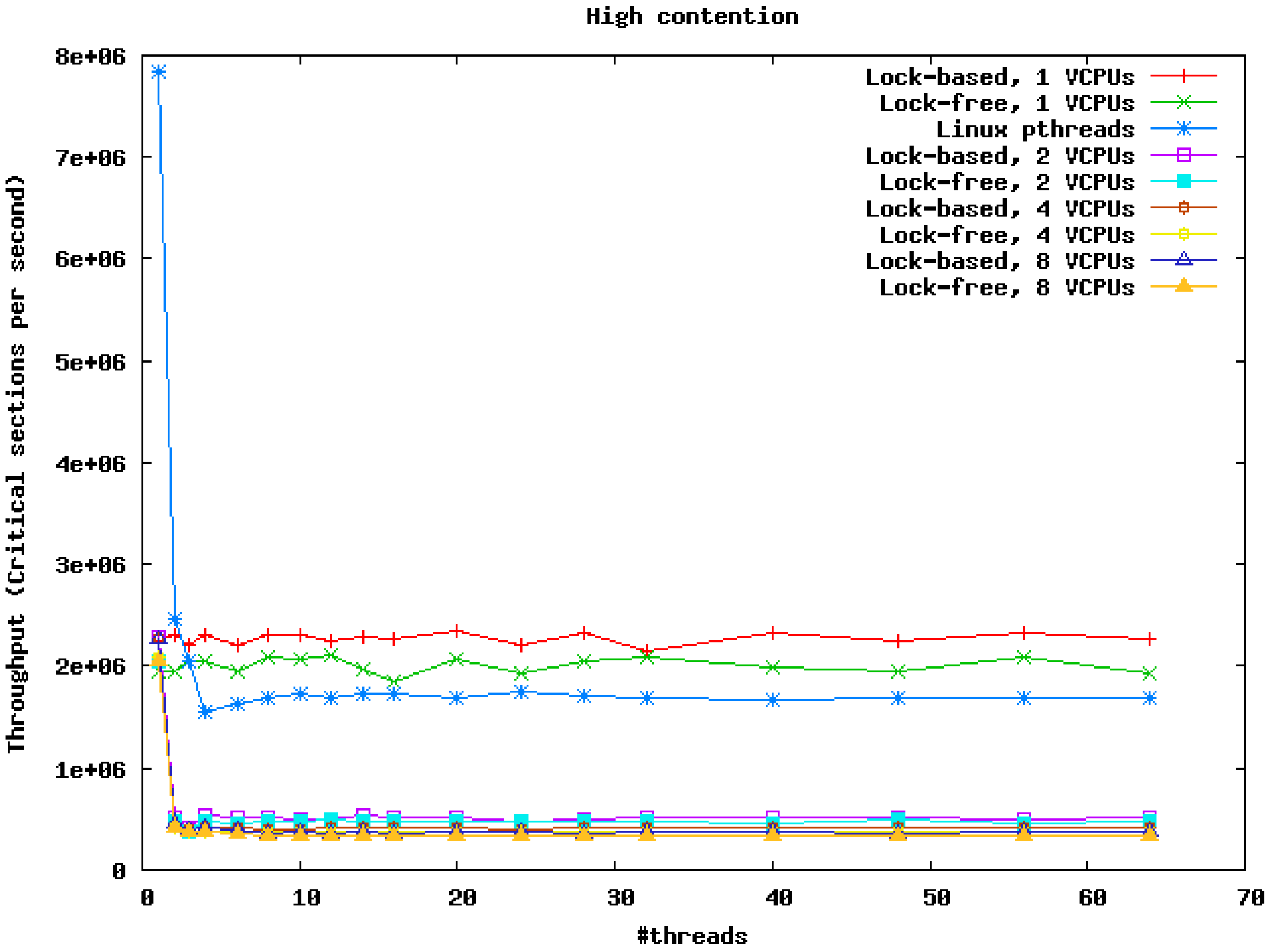
Experimental evaluation (i)



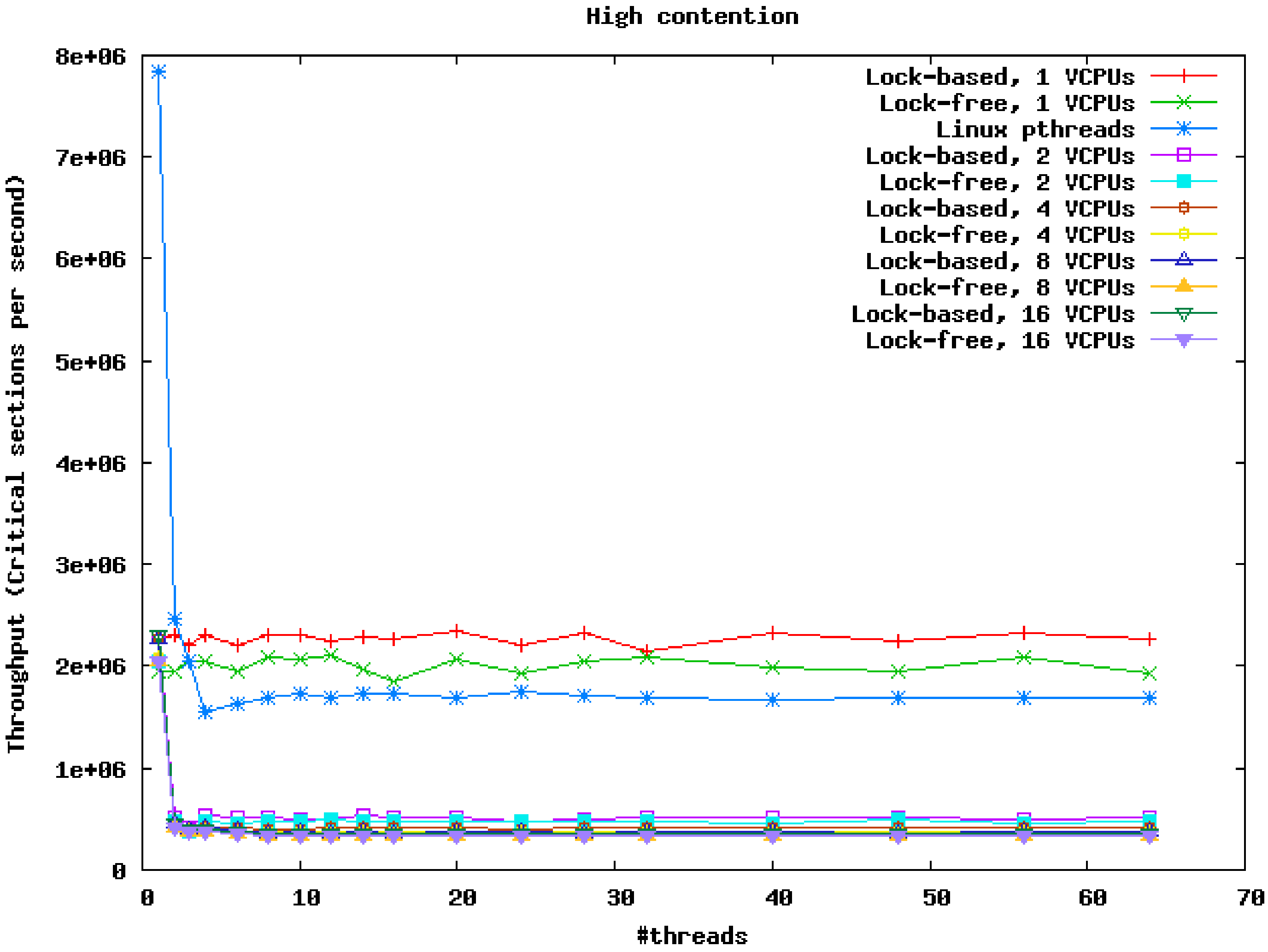
Experimental evaluation (i)



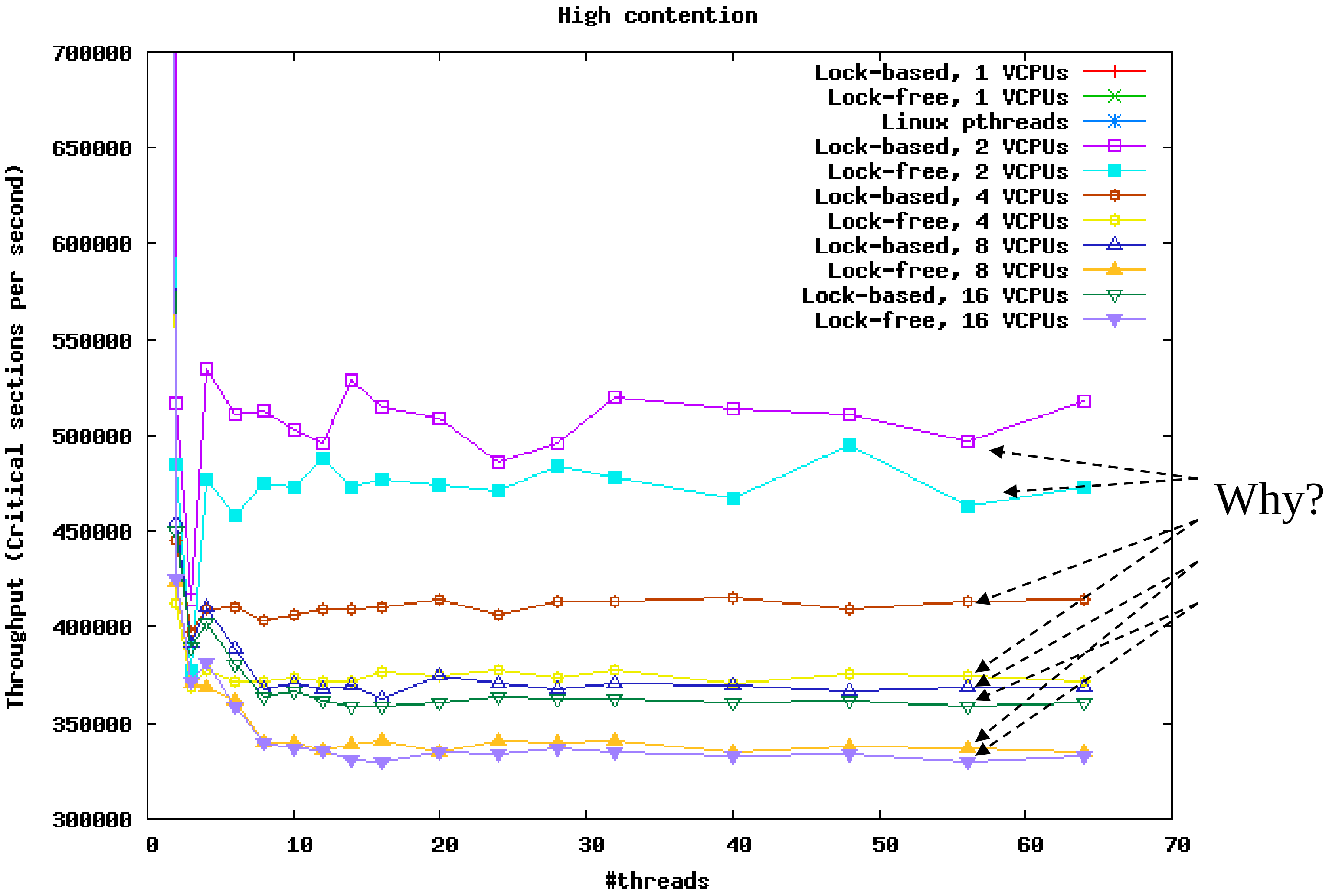
Experimental evaluation (i)



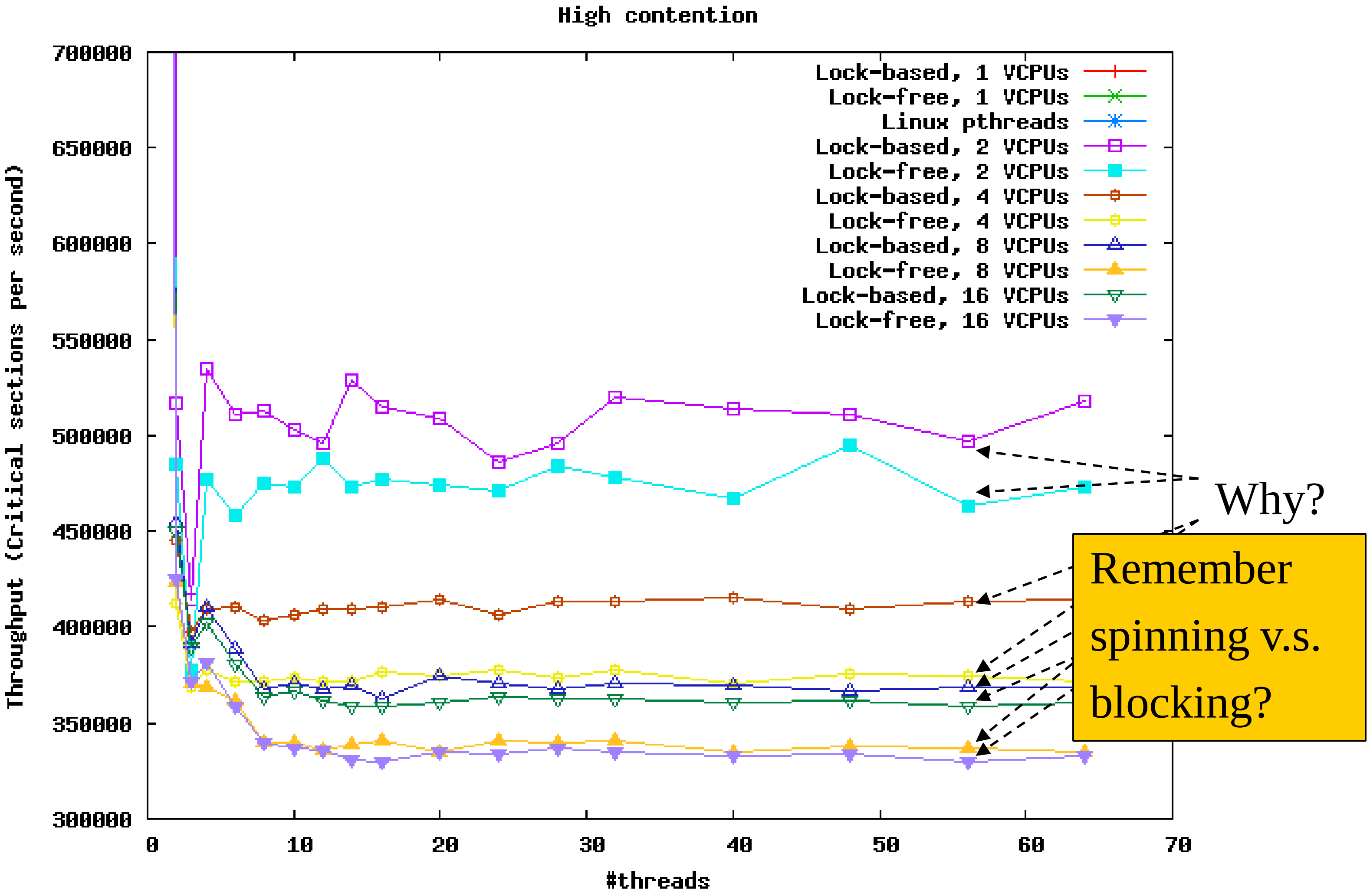
Experimental evaluation (i)



Experimental evaluation (i)



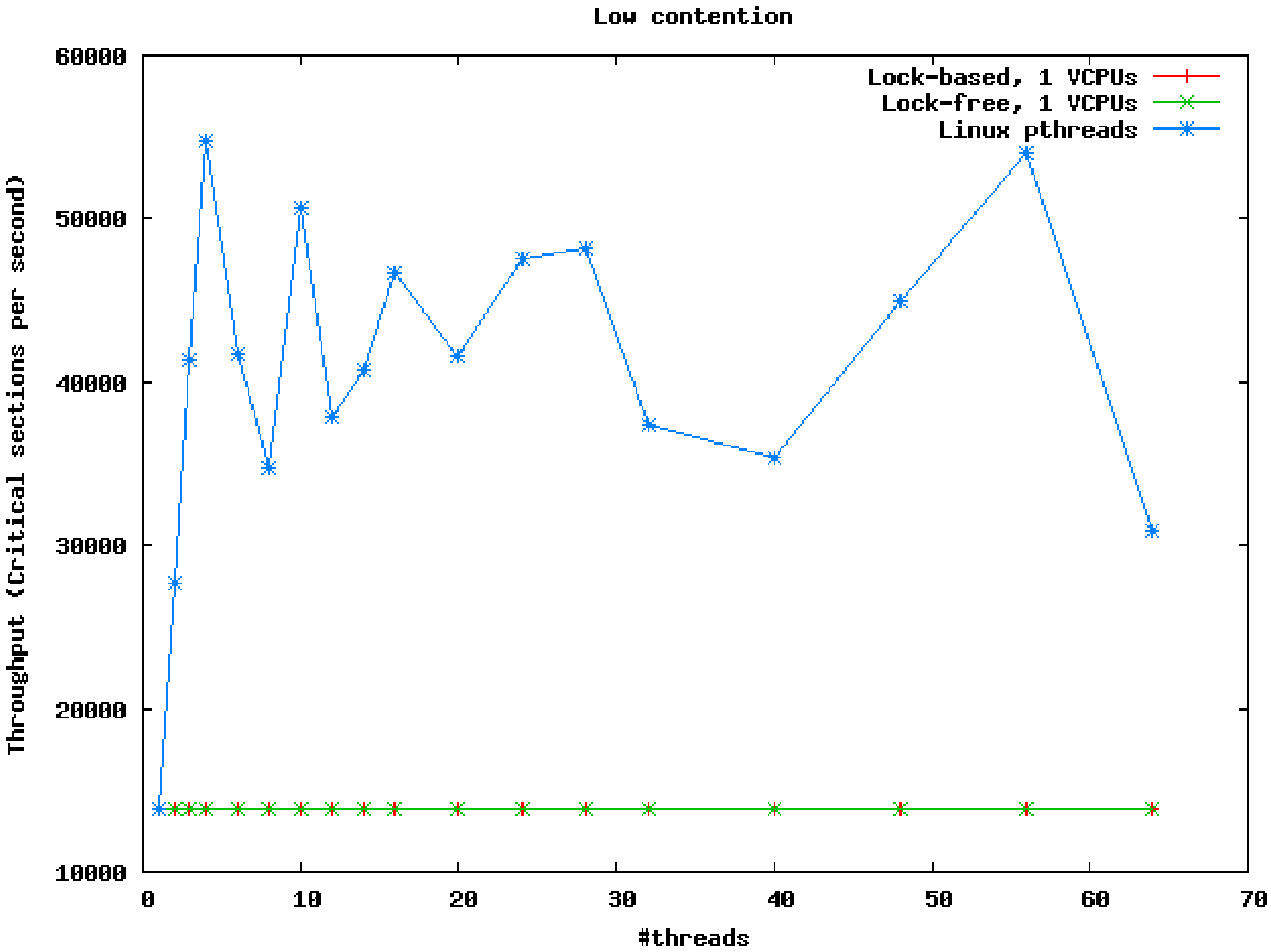
Experimental evaluation (i)



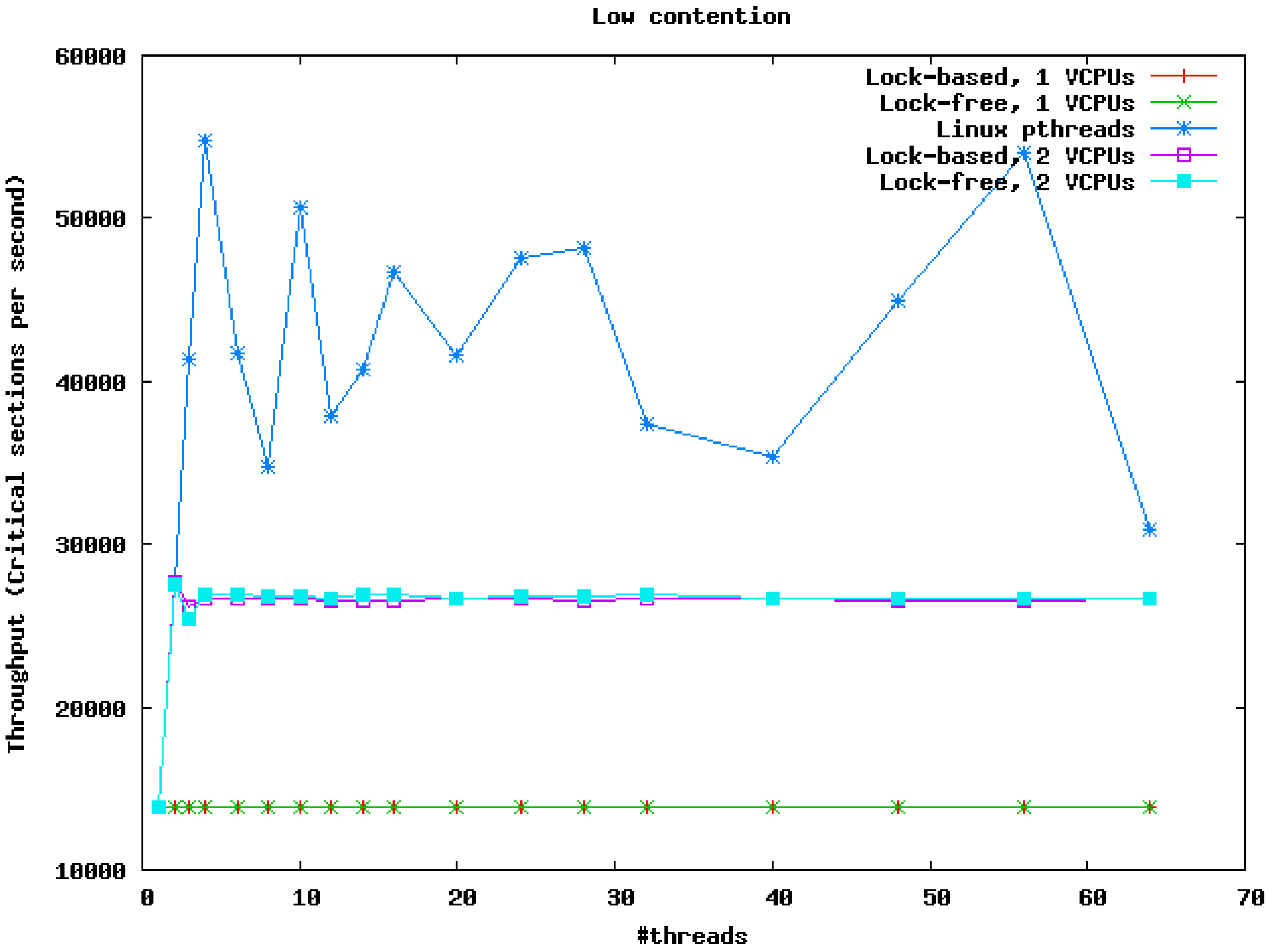
Why?

Remember spinning v.s. blocking?

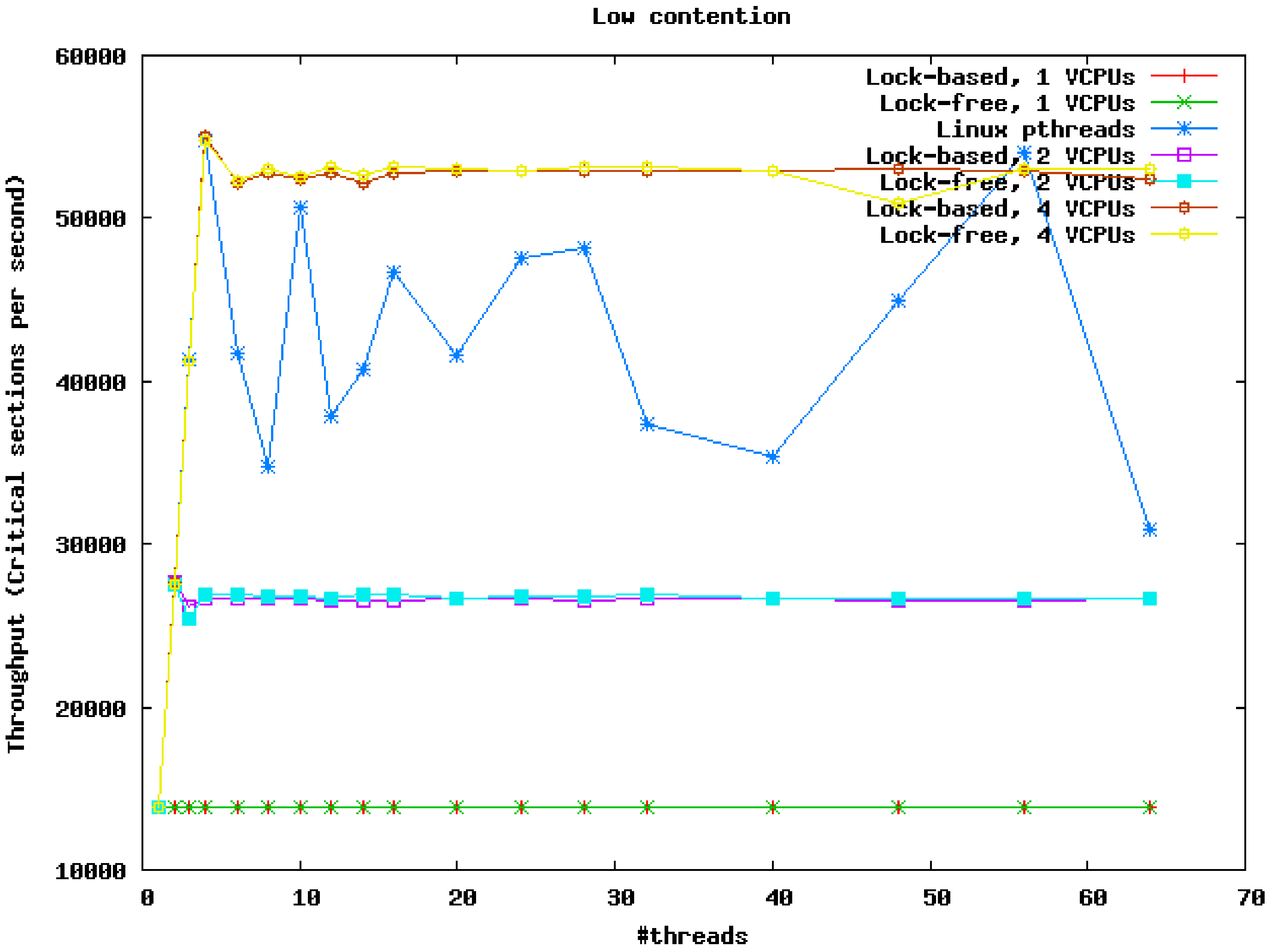
Experimental evaluation (ii)



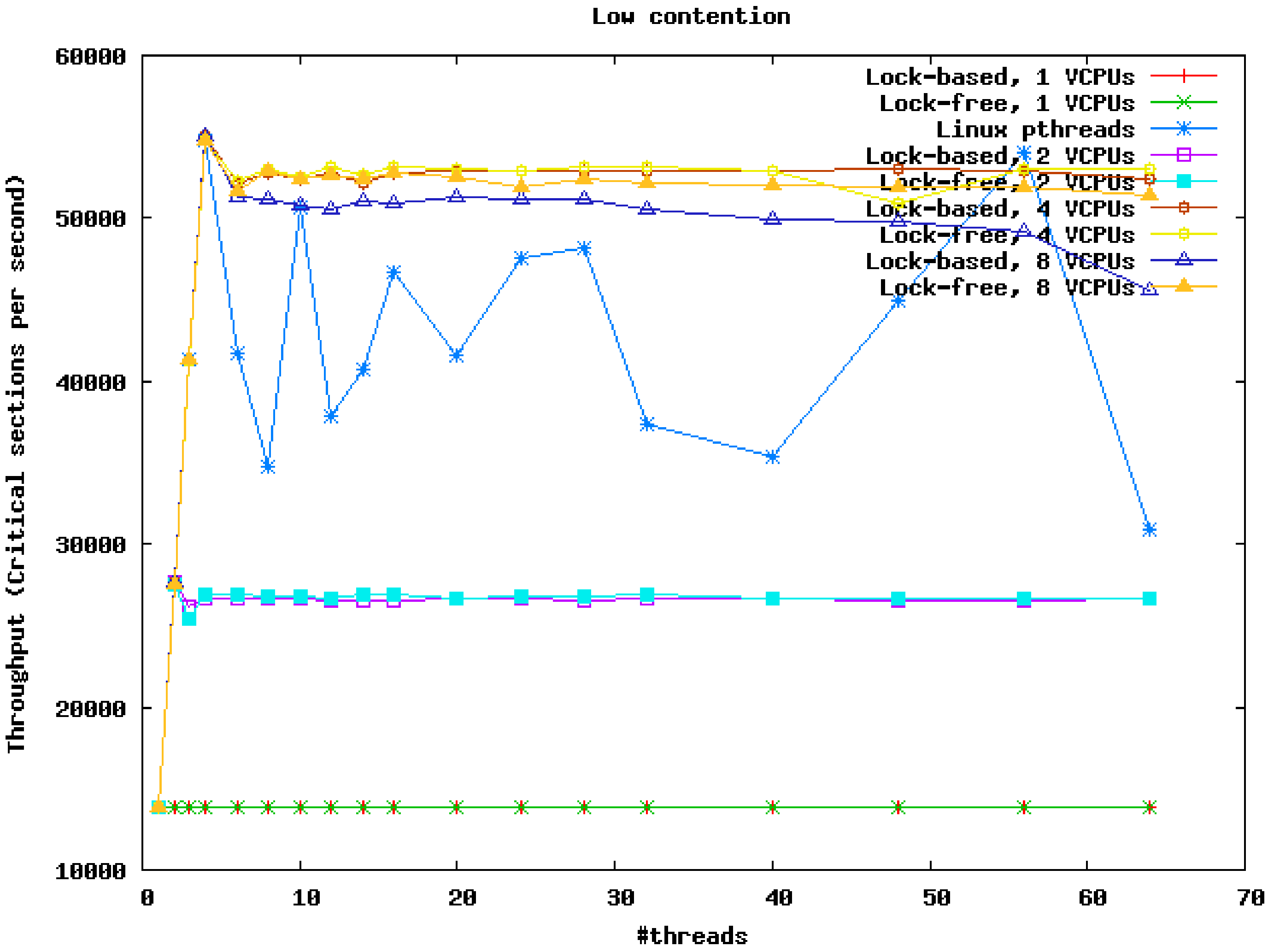
Experimental evaluation (ii)



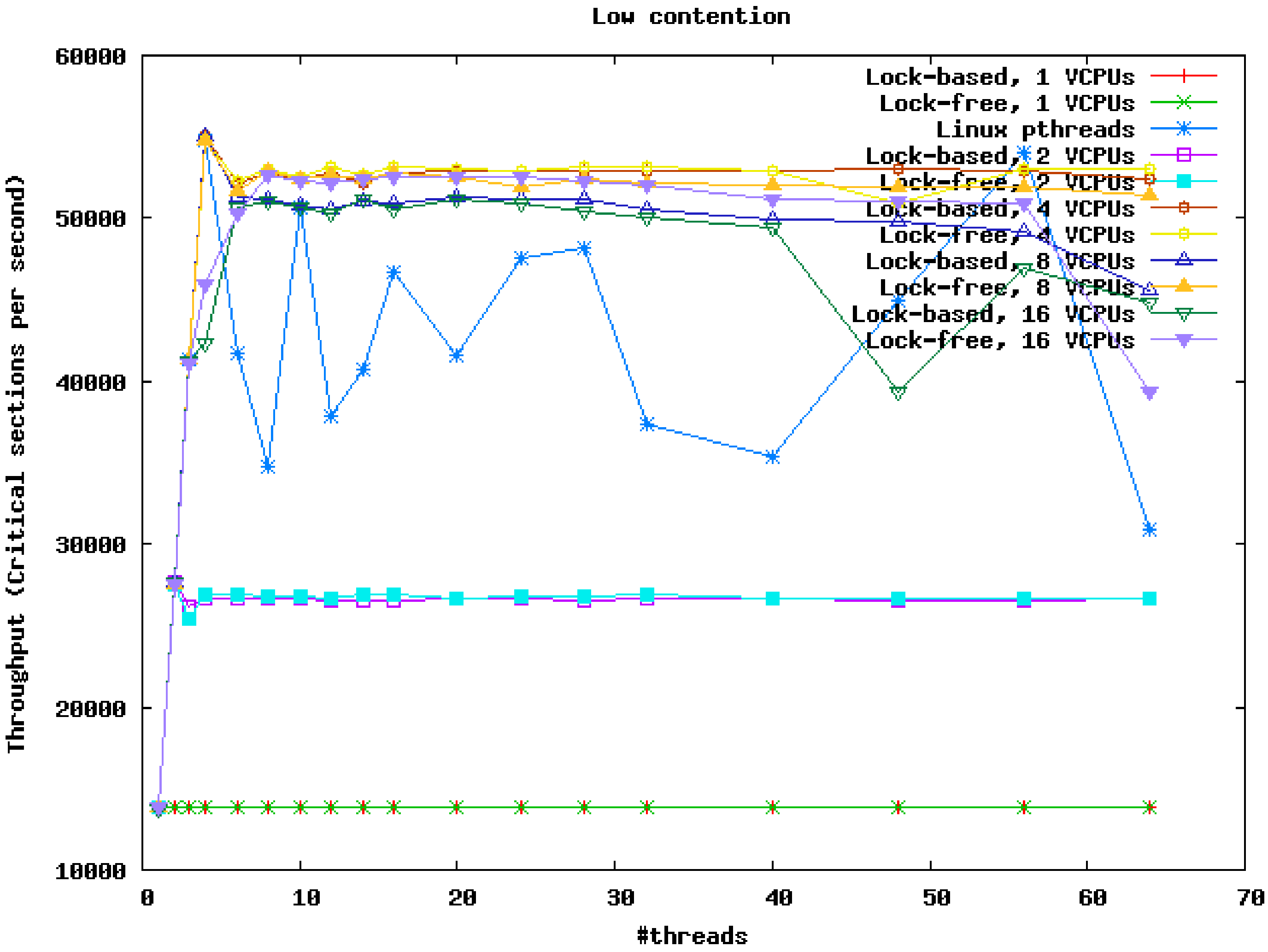
Experimental evaluation (ii)



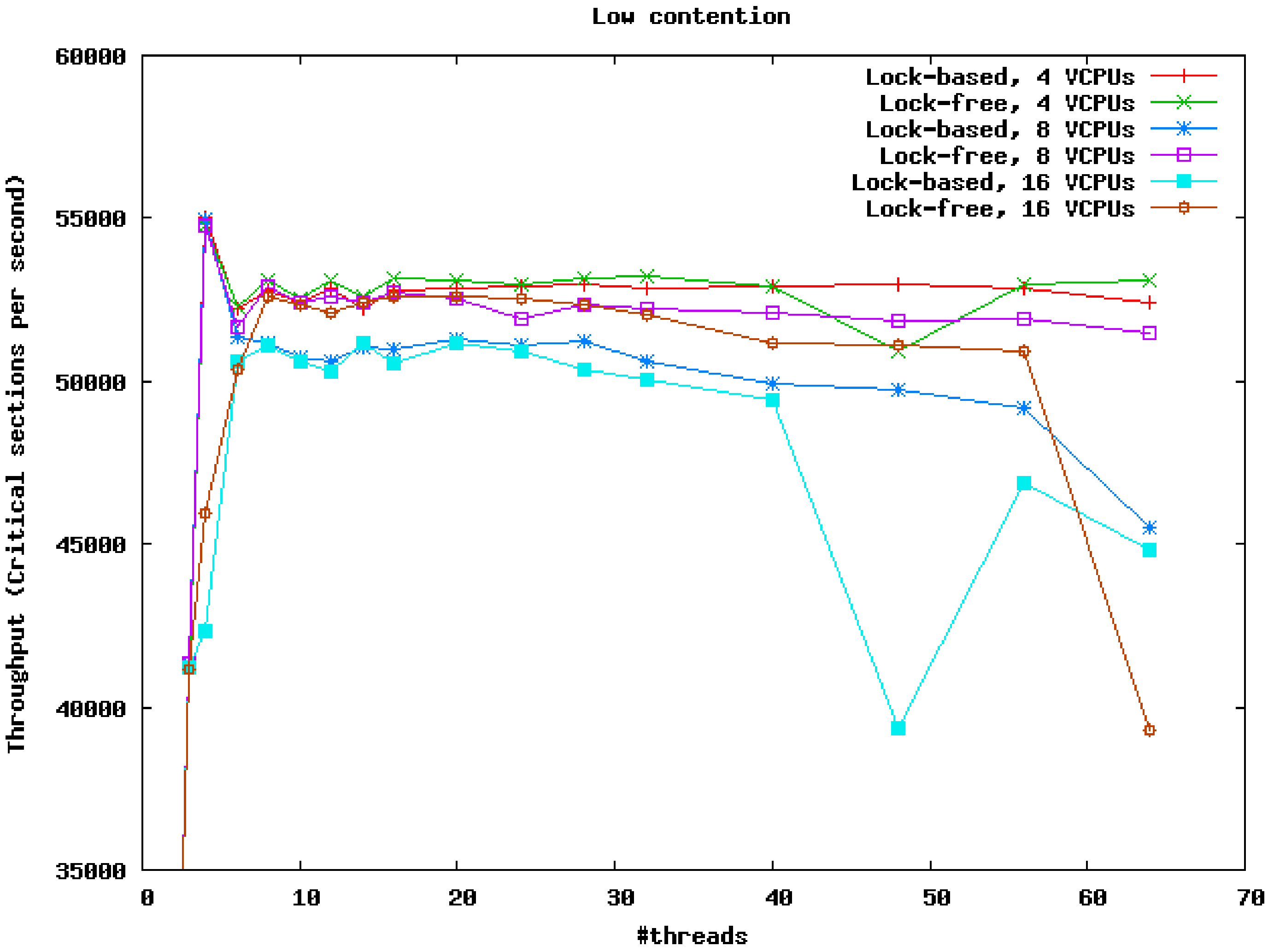
Experimental evaluation (ii)



Experimental evaluation (ii)



Experimental evaluation (ii)



Conclusions and future work

- LFthreads: Lock-free user-level thread library
  - Lock-free thread-blocking synchronization object
  - The hand-off method
- Future work:
  - More synchronization objects
    - Condition variable
    - Semaphore
    - Read-Write locks
    - Signals, Cancellation etc
  - More POSIX pthread compliance
  - Improved scheduling – e.g. lock-free work-stealing

Thank you for listening!

Questions?