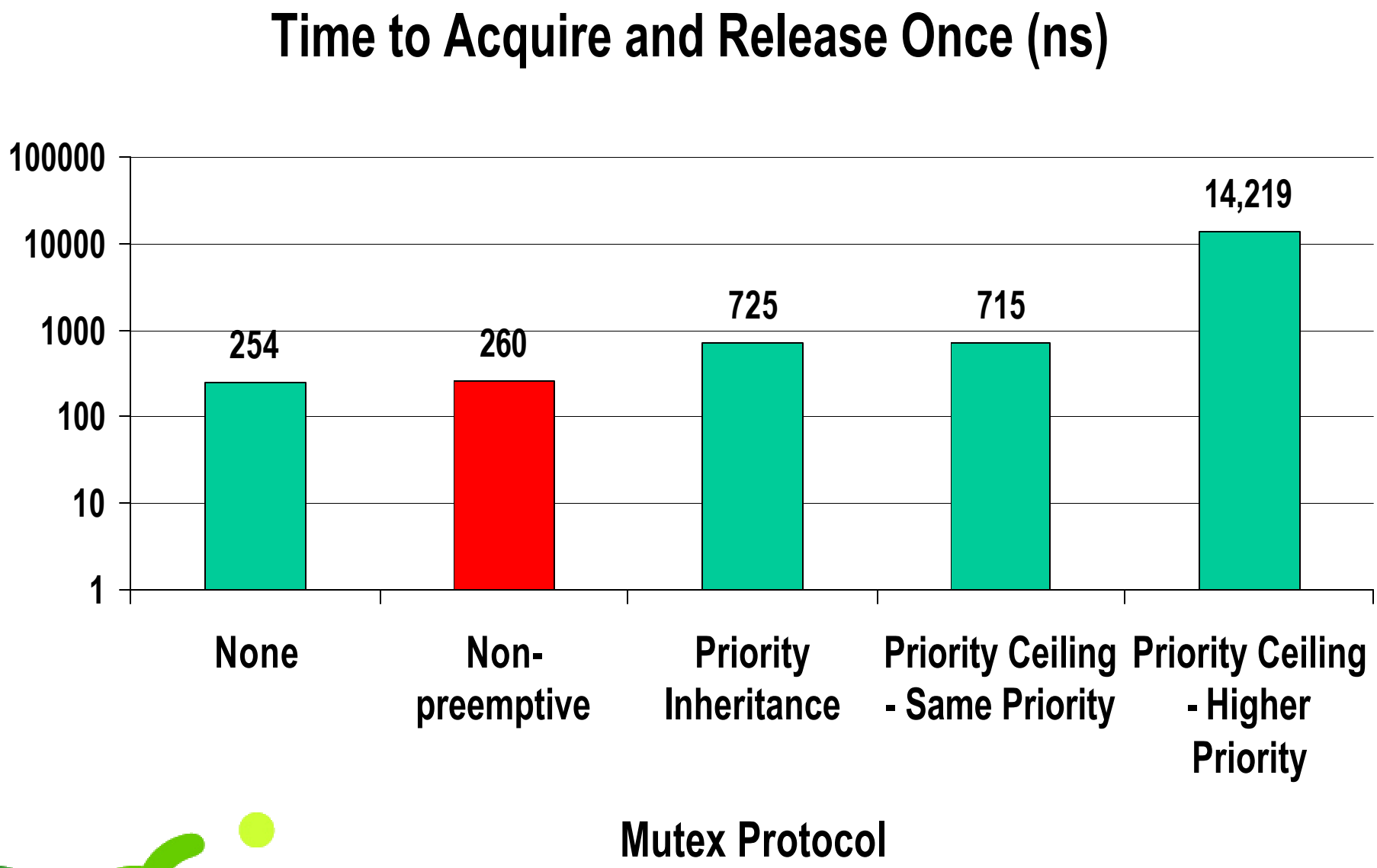
Low Overhead Real-Time Computing

- ***General Purpose OS's can be highly unpredictable***
  - **Linux response times seen in the 100's of milliseconds**
- ***Work around this by isolating from interrupts and other processes and kernel threads***
- ***Put as much functionality into user space as possible***

User Space Functionality

- *Memory map hardware and clocks, and handle interrupts in user space*
- *Use shared memory for IPC*
- *Memory map special kernel pages or use virtual system calls*
- *Do uncontended locking with IRIX usyncs or Linux futuxes*

Mutex Protocol Performance



Overview

- *General Purpose OS's are complicated and unpredictable*

- *Good RT performance may be had by protecting cpu's from outside interference*

- *User level interfaces to common features further reduce latency and unpredictability*

- *Summary: Lock out and avoid as much of the kernel as possible*

Methods of Blocking Interruptions

- *Restrict cpu's to specific threads*
- *Isolate cpu's from TLB and cache invalidations*
- *Direct hardware interrupts elsewhere*
- *Turn off the timeshare scheduling interrupt*
- *Ward off any remaining kernel background tasks*

Lock in User Space When Possible

- *IRIX? usyncs and Linux? futexes attempt to handle most locking actions with atomic primitives outside the kernel*

- *Contended locks force accessors into the kernel to use wait queues*

- *Usually just costs a few instructions vs entering and leaving the kernel for a free lock*

SGI? IRIX? Non-preemptive Mutexes

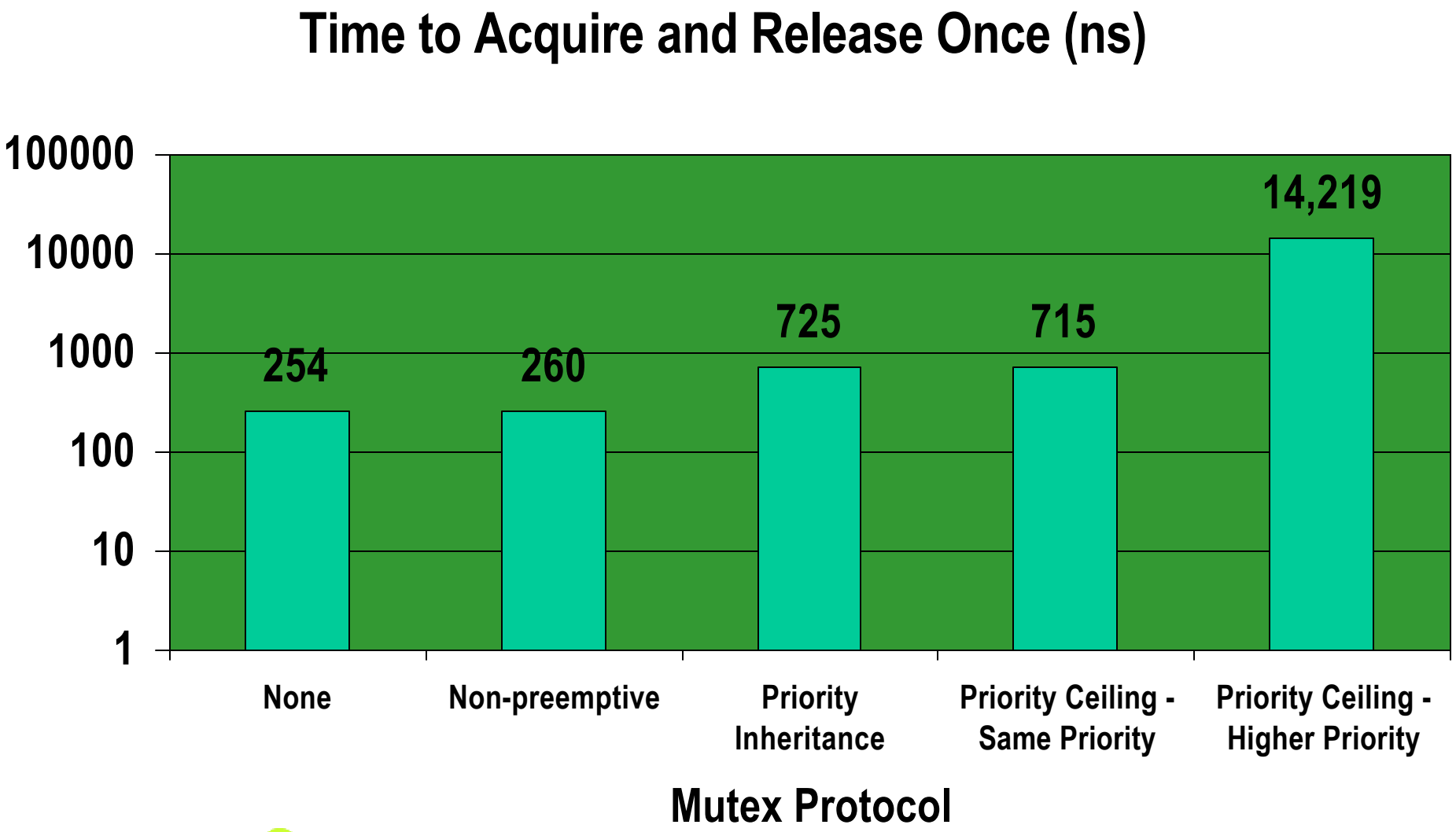
- Priority Inheritance and Priority Ceiling both have overhead even with uncontended locks

- IRIX non-preemptive mutexes provide pseudo - priority ceiling protection by preventing a lock holder from being preempted by any other thread

- Lock holders are required to yield the cpu if they prevented an actual preemption attempt

- The two varieties of non-preemptive mutexes protect against other pthreads in the same process and against any threads in the system

Non-preemptive mutex performance



Provide Direct Access to Hardware

- *Memory map or allow programmed I/O access to hardware features*
- *Example: mmap() the Real-Time Clock*
    - *380ns clock_gettime() vs 2.4us from the kernel*
- *Example: mmap() a PCI device*
    - *full register level access from user space*

User Level Interrupts

- *Handle hardware interrupts in user space asynchronously*
- *Works great with memory mapped hardware*
- *31.2us to wake a waiting user thread*
- *13.6us to handle with a User Level Interrupt*

Direct I/O

- *Bypass the buffer cache by directly writing to and reading from user space to disk*

- *Eliminates memory copies and kernel overhead*

- *Allows the process to implement its own caching algorithm*

Inter Process Communication

- *Place message queues in shared memory areas*

- *2.9us to enqueue a message in shared memory*

- *6.7us to enqueue a message through the kernel*

Memory Map the Kernel

- *Provide useful per-thread data in special mapped pages of kernel memory*

- *Provides quick access to the current cpu ID, and scheduling and timing information*

- *Provides a window for the thread to indicate its signal mask and scheduling hints without entering the kernel*

Context Switches

- *Some pthread implementations map multiple pthreads onto the same kernel thread*

- *Some things like signals become difficult but the benefit is that the kernel is less involved*

- *Many to one mapped pthreads can context switch in 1.6us vs 27.2us for one to one*