6 Multi-core scalability

In the days of uniprocessor computers, OS kernels allowed were mostly sequential (except for interrupts). Therefore, no mutual exclusion was necessary, except for data structures shared with interrupt handlers.

- Only one thread was allowed to execute within the kernel at a time.
- There was no preeption (except for interrupts.)
- This did not limit performance, because there was only one CPU anyway.

On multi-processor or multi-core hardware, accesses to shared state must be synchronized. The simplest approach is a master mutex lock for the whole kernel. Unfortunately this lock forms a performance bottleneck, which gets worse as the number of cores increases. As a result, OS designers have continuously refactored kernels to allow more concurrency. Current versions of Linux and Windows scale quite well to about 8-16 cores.

What are the causes of the concurrency bottleneck?

- Mutual exlusion locking: serializes access to shared data structures
- Accesses to shared cache lines from multiple cores are expensive (roughly speed of DRAM accesses). False sharing aggravates the problem.

Approaches to increasing concurrency:

- Increase the granularity of locking: whole kernel to per data structure to per data item.
- reduce the need for mutual exclusion locking: use lock-free sychronization.
- Reduce the need for shared data structures: per-core data structures. E.g., move state related to processes/threads currently executing on a core to a structure that is private to that core. Partition free lists, reference counters, statistics conters, etc.
- Affinity scheduling: try to schedule a thread on the same core every time. Works well with per-core state.
- Avoid false sharing in cache lines: careful allocation.