6.828 2014 Lec 18: scaling OSes ==

Plan Scaling OSes Concurrent hash tables Reference counters RadixVM Scalability commutativity rule

Scaling kernels --

Goal: scale with many cores Many applications rely heavily on OS file system, network stack, etc. If OS isn't parallel, then apps won't scale ==> Execute systems calls in parallel

Problem: sharing OS maintains many shared data structures proc table buffer cache scheduler queue etc. They protected by locks to maintain invariants Application may contend on the locks Limits scalability

OS evolution for scaling Early versions of UNIX big kernel lock Fine-grained locking and per-core data structures Lock-free data structures Today: case study of scaling VM subsystem

Example: process queue One shared queue Every time slice, each core invokes scheduler() scheduler() locks scheduling queue If N is large, invocations of scheduler() may contend Contention can result in dramatic performance collapse Scalable locks avoid collapse But still limits nuber of scheduler() invocations per sec.

Observation: sharing is unintended The threads are *not* sharing N cores, N threads Can run each thread on its own core Can we void sharing when apps are not sharing?

One idea: avoiding sharing in common case Each core maintains its own queue scheduler() manipulates its own queue No sharing -> no contention

Extreme version: share nothing (e.g., Barrelfish) Run an independent kernel on each core Treat the shared-memory machine as a distributed systems Maybe even no cache-coherent shared memory Downside: no balancing If one 1 cores has N threads, and other cores have none Someone must worry about sharing Kernel, app, or user-level scheduler Use distributed-systems techniques

Today's focus: use shared-memory wisely Computer has shared memory and OS has shared data structures Arrange that kernel doesn't introduce sharing in the common case Keep per-core data structure (scheduling queue per core) Load-balance only when a queue is empty Tricky when we want sharing Want shared buffer cache Want shared physical pages Some structures work well for sharing without unintended sharing E.g., hash tables. Others are challenging E.g., reference counts

Concurrent hash tables --

Hash tables Used for shared caches (name cache, buffer cache) map block# to block Implementations one lock per hash table (bad scaling) lots of unintended sharing one lock per bucket (better scaling) blocks that map to same buck share unintendedly lock-free lists per bucket (see below) little unintended sharing case: search with on list, while someone is removing

Lock-free bucket

```
struct element {
  int key;
  int value;
  struct element *next;
};
```

```
struct element *bucket;



void push(struct element *e) {
    again:
        e->next = bucket;
        if (cmpxchg(&bucket, e->next, e) != e->next)
            goto again;
}



struct element *pop(void) {
    again:
        struct element *e = bucket;
        if (cmpxchg(&bucket, e, e->next) != e)
            goto again;
        return e;
 }
```

No changes to search

More complicated to remove from middle, but can be done

Challenge: Memory reuse (ABA problem) stack contains three elements top -> A -> B -> C CPU 1 about to pop off the top of the stack, preempted just before cmpxchg(&top, A, B) CPU 2 pops off A, B, frees both of them top -> C CPU 2 allocates another item (malloc reuses A) and pushes onto stack top -> A -> C CPU 1: cmpxchg succeeds, stack now looks like top -> B -> C this is called the "ABA problem"
(memory switches from A-state to B-state and back to A-state without being able to tell)

Solution: delay freeing until safe E.g., Arrange time in epochs Free when all processors have left

previous

Reference counters --

Challenge: involves true sharing Many resources in kernel are reference counted Often a scaling bottleneck (once unintended sharing is removed)

Reference counter Design 1: inc, dec+iszero in lock/unlock content on cache-line for lock Design 2: atomic increment/decrement content on cache-line for refcnt Design 3: per-core counters inc/dec: apply op to per-core value iszero: add up all per-core values and check for zero need per-core locks for each counter space overhead is # counters * # cores

Refcache An object has a shared reference count Per-core cache of deltas inc/dec compute a per-core delta iszero(): applies all deltas to global reference counter If global counter drops to zero, it stays zero Space Uncontended reference counters will be evicted from cache Only cores that use counter have delta

Challenge: determining if counter is zero Don't want to call iszero() on-demand it must contact all cores Idea: compute iszero() periodically divide time into epochs (~10 msec) at end of an epoch core flushes deltas to global counter if global counter drops to zero, put on review queue for 2 epochs later if no core has it on its review queue 2 epochs later: if global counter is still zero, free object why wait 2 epochs? See example in paper in Figure 1 More complications to support weak references

Epoch maintainance global epoch = min(per-core epochs) each core periodically increase per-core epoch each 10 msec call flush()+review() one core periodically compute global epoch

RadixVM --

Case study of avoiding unintended sharing VM system (ops: map, unmap, page fault) Challenges: reference counters semantics of VM operations when unmap returns page must be unmapped at all cores

Goal: no intended sharing for VM ops Ops on different memory regions No cache-line transfer when no sharing Ok to have sharing when memory regions overlap That is intended sharing

What data structures do we need to maintain? Some information per hardware page (e.g., ppinfo array in jos) Modern OSes don't use an array like jos (too expensive in terms of memory) Modern OSes use a balanced tree Lock-free balanced trees are tricky Unintended sharing (see figure 6)

Solution: radix tree Behaves like hardware page tables Disjoint lookups/inserts will access disjoint parts of the tree (see figure 7) stores a separate copy of the mapping metadata for each page metadata also stores pointers to physical memory page Folds repeated entries (unlike jos array) No range queries Freeing nodes in tree using refcache

TLB shootdown Unmap requires that no core has the page mapped before returning The core running the unmap must send TLB shootdowns to the cores that have the page in their TLB Which cores do have the page in their TLB? Easy to determine if the processor has a software-filled TLB At a TLB misses the kernel can record the core # for the page x86 has a hardware-filled TLB Solution: per-core page tables The paging hardware will set the accesses bit only in the per-core page table

Maps/unmaps for overlapping region simple: locks enforce ordering of concurrent map/unmap on overlapping regions acquire locks, going left to right page-fault also takes a lock

Implementation sv6 (C++ version of xv6)

Eval Metis and microbenchmarks

Avoiding unintentionial sharing --

Scalable commutativity rule rule: if two operations, commute then there is a scalable (conflict-free) implementation non-overlapping map/unmap are example of a general rule intuition: if ops commute, order doesn't matter communication between ops must be unnecessary
http://pdos.csail.mit.edu/papers/commutativity:sosp13.pdf

Edit By MaHua