CSE221 Lecture 14

Aronya Baksy

November 2024

1 User Threads vs Kernel Threads

- **User Threads**: Flexible, customizable and less overhead
- **Kernel Threads**: OS-aware, make use of parallelism in hardware (multi-core arch.) and better isolation between each other

2 Scheduler Activations: Effective Kernel Support for User-level Parallelism

2.1 Goals

- Contrast kernel and user level threads, show that user level threads perform better
- describe the design, implementation, and performance of a new kernel interface and user-level thread package:
  - When kernel is not involved, same performance as user-level threads
  - If kernel calls are involved, schedule threads to ensure that priority order is maintained, processors aren't idle, and waiting threads can be scheduled off the CPU
  - Simple customization policies for scheduling

2.2 Costs of kernel-level threads

- **Cost of trapping** to kernel for any operations (even when a CPU switches between 2 threads in the same process addr. space)
- **Cost of generality**, i.e. kernel-level thread libraries are not flexible to needs of different apps (e.g. kernel level thread lib might enforce priority scheduling but apps might be better served using FIFO scheduling)

2.3 Drawbacks of existing user-level thread implementations

- Kernel threads are the **wrong abstraction** for user-level thread management because: they block/resume/are pre-empted without notifying user space, and are scheduled oblivious of the user state
- User threads that are scheduled onto a kernel thread can block that kernel thread (and all the waiting user threads) for I/O, having more kernel threads causes issues with regards to progress
- User threads can't be pre-empted while holding locks (time slicing is a bad idea for threads holding spinlocks)
- Ensuring logical correctness when kernel threads are used is tough, as there may not be enough kernel threads to run user threads

2.4 Approach

- Provide each app with a number *virtual multiprocessors*
- Kernel level schedules processors to user-level applications (i.e. address spaces)
- User-level threading system schedules that app's threads between the processors assigned to it
- kernel notifies the user-level addr. space thread scheduler of every kernel event whenever kernel changes number of processors associated with that user-level addr. space

- thread system in each address space notifies the kernel of the subset of user thread operations that can affect processor allocation decisions (i.e. when the user level needs more/less CPUs from kernel)
- A **scheduler activation** vectors control from the kernel to the address space thread scheduler on a kernel event.
- **Scheduler Activation**:
  - Execution context for user threads (like a kernel thread)
  - Notifies user level of kernel events
  - Provide space in the kernel to save the current user thread's CPU context when it is pre-empted by the kernel
- Scheduler activations **invariant**: there are always exactly as many running scheduler activations as there are processors assigned to that user addr. space.
- Main **difference between kernel threads and scheduler activation**:
  - Once an activation's user thread is stopped by the kernel, it is never resumed by the kernel
  - A new activation is created to notify the user space that the current user thread is stopped, then the user space system cleans the old activation, informs the kernel that the old activation can be **reused** and decides which thread can run on that activation
- **upcall** points
  - add processor, processor preempted, activation blocked/unblocked
  - what happens when an activation blocks:
    - old activation blocks
    - OS creates virtual processor, assigns it to process
- Notifying the kernel of user-level events affecting processor allocation (**downcalls**)
  - thread system only needs to tell the kernel about a small subset of possible events.
  - notify kernel when getting more runnable threads than processors, or more processors than threads.
  - need to make assumption that thread systems are honest in reporting their processor and running thread value
- Scheduler activations use **upcalls** from kernel to user thread management system:

2.4.1 Read Call

1. User thread issues a read call, blocking its SA
2. Kernel creates a new SA because the original SA is blocked
3. New SA notifies user level scheduler to schedule a new thread on the original SA
4. When read() returns, kernel pre-empts a different core to use its SA for the upcall, then notify user level that both the caller of read, as well as the pre-empted core are available for scheduling threads

2.5 Main Issues

- thread priorities may cause additional preemptions
- kernel interaction with application is entirely in terms of scheduler activations
- scheduler activations work properly even when a preemption or page fault occurs in the user-level threads
- user-level thread that has blocked in kernel may still need to execute in kernel mode when IO completes

3 The Linux Scheduler: A decade of wasted cores

3.1 Goals

- **Claim**: Linux scheduler does not follow the invariant "make sure that ready threads are scheduled on available cores" for **multicore systems**
- Discover and fix bugs in Linux scheduling on multicore systems
- Build new tools that check for violation of the invariant online and visualize scheduling activity

3.2 How CFS Works

- CFS is an implementation of the weighted fair queueing (WFQ) scheduling algorithm
- The interval (during which all threads must run atleast once) is divided among threads proportionally to their weight (i.e. priority or niceness)
- Once a thread's vruntime (real runtime/weight) exceeds its assigned timeslice, the thread is pre-empted from the CPU if other runnable threads are available
- Threads are organized in a runqueue (a **red-black tree**) in which the threads are sorted in the increasing order of their vruntime
- Multicore: each core must maintain its own runqueue, periodically run a **load-balancing algorithm** that will keep the queues roughly balanced
- Questions for load-balancing:
  - **Metric**: **load**, which is the combination of the thread's weight and its average CPU utilization (add group factor using **cgroups**)
  - **Algorithm**: Hierarchically group cores into **scheduling domains** based on the physical resource organization
  - In a scheduling domain, the sets of cores among which the load balancing is performed are called scheduling groups.
  - The core will find the busiest scheduling group other than its own and will steal tasks from the busiest core in that group.
  - **Optimization**: run the load-balancing algorithm only on the designated core for the given scheduling domain

3.3 Bugs in CFS

3.3.1 Group Imbalance Bug

- The load balancing algorithm is not stealing tasks from more loaded cores on other nodes
- This is because we compare the **average load** of all the cores in the other scheduling group (line 11, Figure 1), which may be skewed by a small number of high load threads running on **some** cores of that group
- **Bug fix**: Compare the *minimum load* of SGs, ensures that no core of a sched. group will remain overloaded while a core of another sched. group has a smaller load

3.3.2 Scheduling Group Construction Bug

- Scheduling groups are constructed based on machine organization (how many hops is a node from a single reference node), but all the NUMA nodes use the same reference (typically node 0)
- Problem: cores in the same SG will not steal work from each other, causing imbalance
- **Bug Fix**: within each domain, create SGs based on **that core's perspective** (i.e. the core within the domain that runs the load balancer)
- Now scheduling imbalances are easier to detect across NUMA nodes now

Algorithm 1 Simplified load balancing algorithm.

```
    {Function running on each cpu cur_cpu:}
 1: for all sd in sched_domains of cur_cpu do
 2:     if sd has idle cores then
 3:         first_cpu = 1st idle CPU of sd
 4:     else
 5:         first_cpu = 1st CPU of sd
 6:     end if
 7:     if cur_cpu ≠ first_cpu then
 8:         continue
 9:     end if
10:     for all sched_group sg in sd do
11:         sg.load = average loads of CPUs in sg
12:     end for
13:     busiest = overloaded sg with the highest load
            (or, if nonexistent) imbalanced sg with highest load
            (or, if nonexistent) sg with highest load
14:     local = sg containing cur_cpu
15:     if busiest.load ≤ local.load then
16:         continue
17:     end if
18:     busiest_cpu = pick busiest cpu of sg
19:     try to balance load between busiest_cpu and cur_cpu
20:     if load cannot be balanced due to tasksets then
21:         exclude busiest_cpu, goto line 18
22:     end if
23: end for
```

Figure 1: CFS Multi-core Load Balancing Algorithm

Name	Description	Kernel version	Impacted applications	Maximum measured performance impact
Group Imbalance	When launching multiple applications with different thread counts, some CPUs are idle while other CPUs are overloaded.	2.6.38+	All	13×
Scheduling Group Construction	No load balancing between nodes that are 2-hops apart	3.9+	All	27×
Overload-on-Wakeup	Threads wake up on overloaded cores while some other cores are idle.	2.6.32+	Applications that sleep or wait	22%
Missing Scheduling Domains	The load is not balanced between NUMA nodes	3.19+	All	138×

Table 4: Bugs found in the scheduler using our tools.

Figure 2: Summary of scheduler bugs found by authors

3.3.3 Overload-on-Wakeup Bug

- a thread that was asleep may wake up on an overloaded core while other cores in the system are idle
- Why: optimization bug that forces thread to wake up on a core within the same NUMA node
- **Bug Fix**: wake up thread on same core if it is idle, else if there are idle cores in the system, we wake up the thread on the core that has been idle for the longest time, else fall back to old algorithm.
- Low overhead because scheduler already maintains list of idle cores, might impact power consumption as idle cores go into power save mode

3.3.4 Missing Scheduling Domains Bug

- When a core is disabled and then re-enabled using the /proc interface, load balancing between any NUMA nodes is no longer performed.
- **Why**: incorrect update of a global variable representing the number of scheduling domains in the machine
- **Bug Fix:** in code that re-generates domains when a core is disabled, a function call to the code which re-generates domains *across NUMA nodes* was removed during a refactor, that was added back

3.4 Summary

- The addition of autogroups coupled with the hierarchical load balancing introduced the Group Imbalance bug.
- Asymmetry in new complex NUMA systems triggered the Scheduling Group Construction bug.
- Cache-coherency overheads on modern multi-node machines motivated the cache locality optimization that caused the Overload-on-Wakeup bug.
- "NUMA-ness" of modern systems was responsible for the Missing Scheduling Domains bug.
- Additional tools:
  - **Online Sanity Checker** that ensures that the scheduling invariant is maintained always (checker runs at period S, if violation detected, monitors sched. events for M time, raises bug if invariant still violated)
  - Visualization Tool that uses kernel instrumentation with small overheads, to show salient scheduling activity over time