2017310367 정다솔 #homework2

Multiprocessor systems help computer easy to make a single CPU much faster without using too much power. However, there are many difficulties when we use more than a single CPU. One of the problems for the operating system is multiprocessor scheduling.

10.1 Background: Multiprocessor Architecture

There are some differences between single-CPU hardware and multi-CPU hardware such as the use of hardware caches and the way of sharing data. In a system with a single CPU, caches which copy the data in the main memory of the system help the processor run programs faster. By keeping frequently accessed data in a cache, the system can use the large, slow memory fast.

When program issues load the first time, it takes long time to get data in main memory. The processor anticipates that the data may be reused and puts a copy into the CPU cache. If the program later uses this same data again, the CPU first checks for it in the cache and uses it. Therefore, the program runs faster. Caches are based on temporal locality and spatial locality. Temporal locality is that if data has been used once, it might be used again in the future. Spatial locality is that if data at address x is used, it might be used data near x as well. Because locality exist in many programs, hardware systems can guess about which data to put in a cache.

If you have multiple processors in a single system with a single shared main memory, caching with multiple CPUs is much more complicated. For example, if the program on CPU 1 modifies the value at address A, it might update its cache with the new data, but the data is not in main memory because of its slow speed. Then the OS decides to move it to CPU 2. The program re-reads the value at address A from main memory because there is no data CPU 2’s cache, and thus the system fetches old value from main memory. This is the problem of cache coherence.

The basic solution is provided by the hardware which monitoring memory accesses. Other solution is to use an old technique known as bus snooping. Each cache pays attention to memory updates by observing the bus, and CPU checks an update for a data which holds in its cache.

10.2 Don’t Forget Synchronization

When programs access shared data items or structures across CPUs, mutual exclusion primitives should be used to guarantee correctness. For example, if there is a shared queue being accessed on multiple CPUs concurrently without locks, updating elements from the queue concurrently will not work. It needs locks to atomically update the data structure.

If threads on two CPUs enter the code sequence to remove an element from a shared linked list at the same time. Thread 1 stored the current value in its variable. Then if Thread 2 executes the same code, it also will store the same value in its own variable. Thus, instead of each thread removing an element, each thread will make problems trying to remove the same element. The solution is to use locking. Unfortunately, this solution also has problems such as performance speed.

10.3 One Final Issue: Cache Affinity

One final issue is cache affinity. When a process run on a particular CPU, it is faster to run it on the same CPU used before because its state is already present in the caches on that CPU. Therefore, a multiprocessor scheduler should prefer to keep a process on the same CPU considering cache affinity.

10.4 Single-Queue Scheduling

SQMS(single queue multiprocessor scheduling) is reusing the basic framework for single processor scheduling, by putting all processes that need to be scheduled into a single queue. Advantage of SQMS is simplicity because it does not need much work to take policy.

However, there are some disadvantage of SQMS. The first problem is a lack of scalability. To ensure SQMS code accesses the single queue on multiple CPUs, locks have to be into the code. However, if these single lock increases, it might reduce performance. The second problem is cache affinity. Each process becomes bouncing around from CPU to CPU doing exactly the opposite of cache affinity, because each CPU just choose the next one from the shared queue. To solve this problem, SQMS schedulers include affinity mechanism to make process continue to run on the same CPU. Moreover, there is affinity for moving others around to balance load. In this way, processes are not moved across processors, so it could maintain affinity. Then by deciding to migrate a different job, it can get affinity fairness as well.

Therefore, it is easy to make an existing single-CPU scheduler which has only a single queue. However, it does not easy to maintain cache affinity.

10.5 Multi-Queue Scheduling

MQMS(multi-queue multiprocessor scheduling) is using multiple queues because of the problems of single-queue schedulers. MQMS consists of multiple scheduling queues that follow a particular scheduling discipline. A process is placed on exactly one scheduling queue by some heuristic, and it is scheduled independently. Therefore, it can avoid the problems in the single-queue such as information sharing and synchronization.

For example, if there are two CPUs and some processes enter the system, then each CPU has a scheduling queue and the OS has to decide the queue position of each processes by the queue scheduling policy like round robin. MQMS has an advantage that it provides cache affinity and it is okay if lock and cache contention increased.

However, MQMS has a disadvantage such as load imbalance. If one of the processes finished among 2-2 processes. Then run our round-robin policy on each queue of the system, each CPU has 2-1 processes. CPU1 has only one process, on the other hands, CPU2 has twice as much then CPU1. If all processes are finished in CPU1, CPU1 will be left idle.

The solution is to move processes which called migration. By migrating a process from one CPU to another, true load balance can be achieved. If one CPU is idle and the other has some jobs, it will be easy to migrate processes to idle CPU. A single migration does not solve the problem when there is one process in CPU1, so we continue migration of one or more processes. Therefore, the system has to decide to enact migration by using work stealing. With a work-stealing, a queue that has little process will occasionally check another queue to steal one or more processes. If the queue looks at other queues too often, it will suffer from high overhead. On the other hand, if the queue doesn’t look at other queues often, it might be in danger of suffering from imbalances.

10.6 Linux Multiprocessor Schedulers

In the Linux community, there are three different schedulers popular: the O(1) scheduler, the Completely Fair Scheduler (CFS), and the BF Scheduler (BFS). Both O(1) and CFS use multiple queues, on the other hand, BFS uses a single queue. The O(1) scheduler is a priority-based scheduler, changing a process’s priority and then scheduling those with highest priority. CFS is a deterministic proportional-share approach. BFS is also proportional-share but based on a more complicated scheme known as Earliest Eligible Virtual Deadline First (EEVDF).

10.7 Summary

We have seen various approaches to multiprocessor scheduling. Both SQMS and MQMS have advantage and disadvantages. Whichever approach you take, there is no simple answer.