Multiprocessor Scheduling

This page is a stub.
You can help the wiki by *accurately* adding more contents to it.

Contents

Introduction

Scheduling Algorithms can get tricky when more than one CPU is involved, and with Intel's Hyperthreading technology (one complex CPU appearing as multiple CPUs), MultiProcessors OSdev'ing come to the hobbyist as well...

All of a sudden, the details of the CPU architecture become even more important than with single processor systems. How is the Memory Management Unit (MMU) designed? Some CPUs are able to hold page tables for more than one process in their Translation Lookaside Buffer (TLB) - it would benefit the system performance if the same process would always run on the same CPU, instead of a different one each time it is scheduled.

Another issue is multithreading. On a single-CPU system, multithreading (i.e., more than one control flow in a single address space) is a fine thing, and if done right can greatly benefit a system's performance. But on a multiprocessor system, there are even greater benefits to win - while at the same time it gets harder to achieve them.

Scheduler Activations

Scheduler activations are a mechanism that give the benefits of user-level threads while being implemented in the kernel and providing kernel-thread functionality. The kernel-level scheduler schedules tasks or processes out of which any task could dispatch a process-scheduler. Whenever that task will be selected and dispatched by the system-wide task scheduler in the kernel, it will execute the process scheduler in user-space. The process-level scheduler will have a structure of user-level threads which can be created, paused, waited, and destroyed in user-mode itself. 'm' number of tasks can refer to the process scheduler (as there are 'm' no. of kernel threads in the M:N model) and that scheduler will hold 'n' number of user-level threads. This allows super-fast management of threads in user-space without any context switches.

Process-level scheduling will be particularly useful for processes which have threads which have a short lifespan, e.g. input & output threads. It will also be helpful when a considerable number of threads are sleeping or waiting on others. Those dependencies could be resolved in user-space itself.

This model can be extended in multi-processor systems by having separate run-queues for each kernel-thread. It is just like how the linux kernel maintains separate task queues on each CPU in the system. This reduces concurrency overhead in the process scheduler and allows lock-less scheduling algorithms. But it will induce higher levels of complexity.

Scheduling with each CPU assigned with its own run-queue

Traditional scheduling techniques use one data structure to hold all the tasks in the system. This is many costs and disadvantages which bring down the efficiency and output of the whole operating system, namely -

1. **Synchronization Overhead** - When each CPU will access the *global run queue* of tasks in the system, they will have to lock specific parts of the data structure, if not the whole, and that synchronization will greatly impact parallelism in the system.

2. **Caching & TLB Usage** - If each CPU uses the same data structure, then all tasks will keep jumping from one CPU to the other. This means the tasks will never get the opportunity to fill the cache and TLB with its own code & data. In modern processors, efficient usage of L1, L2, L3 & L4 cache is crucial for best performance and reduction in TLB flushing speeds execution in virtual memory.

To overcome these limitations, each CPU is assigned its own run queue which it accesses without acquiring any lock. Operations on the run queues can be done by turning interrupts off to avoid the *run queue balancer* (we'll introduce this a bit later) from accessing the run queue while the real-scheduler is using it.

Task Balancing or Run-queue Balancing

All multi-processor schedulers suffer from load imbalances between different processors in the system. This means that one processor could be heavily bombarded with tasks while another could be taking a nap (bit joking). This inconsistency is solved the kernel's task balancer. Whenever the task balancer runs (at specific intervals of kernel-time) to balance tasks between two domains of CPUs, it transfers tasks from the heavily loaded one to the less loaded one. It could execute one the heavily loaded processor, less loaded processor, or both depending on its implementation. *It takes tasks from heavily-loaded processors and assigns them to a idle or less-loaded processor in the system. It will access the task queues (it can also take sleeping tasks) of both processors. For lock-*

less multiprocessor schedulers, the interrupts should be turned off (unless already in an interrupt context, where the next interrupt is queued until the IRET executes) while accessing the run-queue for any operation like adding or removing tasks from it.

Other facts ...

One thing that I have read on the subject is that some threads may be *pinned* on a specific CPU while other threads will execute on the first available CPU. This can make sense, for instance, to make the GUI server's main thread 'resident' on a CPU to achieve high responsiveness (other threads that want to interact with the GUI can do so without incurring a context switch penalty :)"

Pinning a thread to a CPU also helps the CPU's cache. If a thread is jumping between CPUs each time it is run, it never gets chance to fill either CPU's cache with code or data: it might run for a moment, and load some instructions from main memory, then get preempted. Next time it is scheduled, it might be running on a different CPU, where it would need to load the cache all over again. If the scheduler can assign each thread its own favourite CPU, it can help increase cache performance. Of course, if two threads with the same favourite CPU want to run at the same time on a dual processor system, one of them will have to take the 'wrong' CPU.

SVR4 Unix had a nice synchronization tool (a sort of smart spinlock) for multiprocessors that allowed a thread to busy-wait for a resource as long as the resource holder was active on another CPU and to enter the SLEEP state when the holder wasn't running. This has been proved in papers (which i lost the references) that this method pays and leads to better performances than always entering the SLEEP state when a resource was busy.

A simple method

One simple way of multiprocessor scheduling is to assign each CPU it's own scheduler. Then when you load a thread it is assigned to a certain CPU. This deals with pinning, but you would have to keep the workloads balanced after thread termination.

See Also

Articles

- Multiprocessing

Forum Threads

- SMP Compatibility

External Links

- Multiprocessor Specification (http://www.intel.com/design/pentium/datashts/242016.htm)
- Process Sleep and Wakeup on a Shared-memory Multiprocessor (http://doc.cat-v.org/plan_9/4th_edition/papers/sleep), describes the approach taken in Plan 9 from Bell Labs (http://plan9.cat-v.org). Paper written by Rob Pike (http://genius.cat-v.org/rob-pike/), Ken Thompson (http://genius.cat-v.org/ken-thompson/) and others.