CHAPTER 4

SCHEDULERS IN XEN

4.1 Scheduling Background

Before outlining the schedulers Xen uses, the terminology groundwork is laid down and the goals a virtual machine scheduler seeks to accomplish are described.

4.1.1 Proportional Share and Fair-Share Schedulers

A CPU scheduler determines which processes, in Xen's case, virtual machines, run on the CPU at any given time. Scheduling correctly makes the scheduler virtually invisible to the user or application. Over the past couple decades, schedulers ranging in complexity were implemented with varying degrees of success. A majority of them fall in two groups: proportional share (PS) and fair-share. Proportional share (PS) schedulers attempt to give CPU time to each virtual machine fairly and instantaneously. For example, if 10 CPU hungry VMs are running on one system, a PS scheduler will give each 10 percent of the CPU's time. PS schedulers are evaluated on fairness. In this example, it should be noted as how close to 10 percent of CPU usage will each VM be maintained. If 5 of the VMs were to become inactive, the scheduler would assign the 5 running machines 20 percent CPU time. Now, all 10 VMs become active again. It would make sense to award some additional CPU time to the 5 VMs that had been inactive. PS schedulers, aiming to provide instantaneous sharing among the active clients, would not award additional time. In contrast, a fair-share scheduler would give additional time to the previously inactive VMs, allowing them to catch up. Fair-share schedulers attempt to provide time-averaged, proportional sharing based on the actual use measured over long time periods.

4.1.2 Work Conserving and Non Work Conserving Schedulers

CPU schedulers fall into two additional categories depending on how they manage CPU idle time: work-conserving (WC) and non work-conserving. Work-conserving means that in a case of two VMs, one of these blocked, the other VM can consume the entire CPU. Non work-conserving enforces caps. In other words, each VMs owns a percentage of the CPU. When the cap is set for a VM, the CPU allotment will never exceed that amount. In a case of two VMs, each VM will get up to 50 percent of CPU, but either VM will not be able to get more than 50 percent even if the rest of the CPU is idle.

4.1.3 Scheduler's Four Goals

Virtual machine schedulers retain 4 main design goals. The first goal is fairness, which is the ability of a VM to get a fair portion of CPU resources. Fairness can be tweaked through scheduling parameters, including weight, cap, and quanta, which are described later. Fairness is more than allowing a VM to run for a given amount of time within a timeframe. Using network data transfers as an example, each bit of data sent creates more work in the future. Not allowing a network workload to run in a timely manner prevents it from receiving a fair share of CPU time when it is made available.

A VM scheduler's second goal is to work well with latency-sensitive work- loads. Ideally, if a latency-sensitive workload uses less than its fair share of CPU, it should run as well when the system is loaded as it does when the system is idle. If a VM would use more than its fair share, then the performance should degrade gracefully.

The third aspect to consider is hyperthreading, an Intel feature designed to make each CPU core appear as two boosting performance. A VM running in a core by itself will have better performance than a VM sharing a core with a VM through hyperthreading. The scheduler should take this into account when determining the each VM's fair share.

The final scheduler goal is power efficiency. Powering down CPU cores or sockets into deeper sleep states can save power for relatively idle systems. When needed, CPU cores and sockets should be able to perform at full capacity. A scheduler needs to either implement this power-vs-performance tradeoff, or provide support for another system to do so.

4.2 Overview of the Scheduler Interface

Xen provides an abstract interface to schedulers. This is defined by a structure that contains pointers to functions used to implement the functionality of the scheduler. Figure 4.1 shows this interface. This is somewhat familiar with a method of defining interfaces in static object oriented languages such as C++ or Java.

```
1  struct scheduler {
2    char *name;              /* full name for this scheduler  */
3    char *opt_name;          /* option name for this scheduler*/
4    unsigned int sched_id;   /* ID for this scheduler         */
5
6    void   (*init)           (void);
7    int    (*init_domain)    (struct domain *);
8    void   (*destroy_domain) (struct domain *);
9    int    (*init_vcpu)      (struct vcpu *);
10   void   (*destroy_vcpu)   (struct vcpu *);
11   void   (*sleep)          (struct vcpu *);
12   void   (*wake)           (struct vcpu *);
13   struct task_slice (*do_schedule) (s_time_t);
14   int    (*pick_cpu)       (struct vcpu *);
15   int    (*adjust)         (struct domain *,
16                            struct xen_domctl_scheduler_op *);
17   void   (*dump_settings)  (void);
18   void   (*dump_cpu_state) (int);
19 };
```

Figure 4.1 : Interface to a Xen scheduler

When adding a new scheduler, it is necessary to create one of these structures pointing to the newly implemented scheduling functions, and to add it to a static array of available schedulers. At boot time, the correct scheduler can be selected by specifying an argument to the hypervisor. The hypervisor reads "sched=*{*scheduler*}*" from the list of boot parameters and attempts to match "scheduler" to the opt name of the schedulers defined as described in Figure 4.1.

Not all of the functions defined by the structure need to be defined for any given scheduler. Any initialized with a NULL pointer are simply ignored. The simplest valid scheduler would set almost all of the functions to NULL, although this might not actually be very useful.

Current versions of Xen include two schedulers, the older *Simple EDF* (*SEDF*) and the newer *Credit Scheduler*. SEDF is more stable, as it has had longer to undergo testing, but has a few limitations that are causing it to be gradually phased out in favor of the Credit Scheduler.

Adding a new scheduler is something that requires modification to the hypervisor sources and recompilation. This is not as much of a problem as it may appear. Generally, each scheduler can be separated out into its own source file, and the only modification required to the rest of the Xen sources is to add it to the list of available schedulers at the top of scheduler.c. This makes it relatively easy to maintain a scheduler outside the main Xen tree.

As well as a list of available schedulers, the scheduler.c file contains all of the scheduler-independent code. Each of the functions in the scheduler definition structure has an analog in this source file, which performs any general operations and then calls the scheduler function (if one exists). For example, the schedule() function defined in this file deschedules the running domain and then calls the do schedule() function for the current scheduler. This returns the new task to be scheduled and the time for which it should run. The generic code then sets a timer to fire at the end of the designated quantum, and runs the new task.

Unlike most of the other scheduler functions, do schedule() is not optional. All of the other functions are called via a macrom which tests for a non-NULL value and returns 0 if one is found; however, this one does not. This means that a scheduler that does not implement do schedule() will crash the hypervisor.

Adding a new scheduler is not something that most users of Xen are likely to need to do; however, it is possible that some users of Xen may require scheduling beyond that provided by the existing schedulers. Although the Credit Scheduler is highly configurable, it is not always possible to coerce it to a particular set of needs.

4.3 Existing Schedulers in Xen

Xen is unique among VM platforms because it allows users to choose among different CPU schedulers. But this choice comes with the burden of choosing the right scheduler and configuring it. For the purpose of this research, three different CPU schedulers are studied that were introduced, all allowing users to specify CPU allocation via CPU shares (weights). Below, I have briefly characterized their main features that motivated their inclusion in Xen at the time.

4.3.1 Borrowed Virtual Time (BVT)

This is a fair-share scheduler based on the concept of virtual time, dispatching the runnable VM with the smallest virtual time first. Additionally, BVT provides low-latency support for real-time and interactive applications by allowing latency sensitive clients to “warp” back in virtual time to gain scheduling priority. The client effectively “borrows” virtual time from its future CPU allocation.

The scheduler accounts for running time in terms of a minimum charging unit (mcu), typically the frequency of clock interrupts. The scheduler is configured with a context switch allowance C, which is the real time by which the current VM is allowed to advance beyond another runnable VM with equal claim on the CPU (the basic time slice or time quantum of the algorithm). C is typically some multiple of mcu. Each runnable domain 2 receives a share of CPU in proportion to its weight weighti. To achieve this, the virtual time of the currently running Domi is incremented by its running time divided by weighti. In summary, BVT has the following features:

• preemptive (if warp is used), WC-mode only;

• optimally-fair: the error between fair share and actual allocation is never greater than context switch allowance C plus one mcu ;

• low-overhead implementation on multiprocessors as well as uni-processors.

The lack of NWC-mode in BVT severely limited its usage in a number of environments, and led to the introduction of the next scheduler in Xen.

4.3.2 Simple Earliest Deadline First (SEDF)

This uses real-time algorithms to deliver guarantees. Each domain specifies its CPU requirements with a tuple (si, pi, xi), where the slice si and the period pi together represent the CPU share that domain requests: Dom_i will receive at least si units of time in each period of length pi. The boolean flag x_i indicates whether Dom_i is eligible to receive extra CPU time (WC-mode). SEDF distributes this slack time fairly manner after all runnable domains receive their CPU share. One can allocate 30% CPU to a domain by assigning either (3 ms, 10 ms, 0) or (30 ms, 100 ms, 0). The time granularity in the definition of the period impacts scheduler fairness. The words domain and virtual machine are used interchangeably.

For each domain Dom_i, the scheduler tracks two additional values (di, ri):

• di - time at which Dom_i's current period ends, also called the deadline. The runnable domain with earliest deadline is picked to be scheduled next;

• ri - remaining CPU time of Dom_i in the current period.

In summary, SEDF has the following features:

• preemptive, WC and NWC modes;

• fairness depends on a value of the period.

• implements per CPU queue: this implementation lacks global load balancing on multiprocessors.

4.3.3 Credit Scheduler

On recent versions of Xen, the *Credit scheduler* is used by default. Each domain has two properties associated with it, a weight and a cap. The weight determines the share of the physical CPU time that the domain gets, whereas the cap represents the maximum. Weights are relative to each other; if all domains have a weight of 128, this has the same effect as giving all domains a weight of 256. In contrast, the cap is an absolute value, representing a proportion of the total CPU that can be used.

By default, the Credit Scheduler is work-conserving. Given two virtual machines with priorities of 128 and 256, the first gets half as much CPU time as the first while both are busy, but can use the whole CPU if the second is idle. The cap is used to force a non-work-conserving mode. If all domains have a cap, and the sum of all caps is below the total CPU capacity, the scheduler does not run any domains for some of the time.

The Credit Scheduler uses a fixed-size 30ms quantum. At the end of each quantum, it selects a new VCPU to run from a list of those that have not already exceeded their fair allotment. If a physical CPU has no underscheduled VCPUs, it tries to pull some from other physical CPUs.

Whether a CPU is over- or underscheduled depends on how it has spent itscredits. Credits are awarded periodically, based on the priority. Consider the following example domains:

1. Priority 64, cap 25%.
2. Priority 64, no cap.
3. Priority 128, no cap.

At the start of a scheduling interval, the first two domains will have 64 credits, whereas the last will have 128. Although all CPUs have work to do, they will be scheduled in a round robin manner. Eventually, the first two domains will be out of credits, and the third one will get all of the CPU to itself for a little while.

If the last domain is idle, the first and second will get equal shares of the CPU until the first has reached its cap of 25%. At this point, the second VCPU continues to run. By doing this, it quickly exhausts its allowance of credits, and is moved into the "overscheduled" queue in the next accounting process. Meanwhile, the other VCPUs continue to accrue credits. At the next accounting, they will be classified as "underscheduled."

Any new allocation of credits that occurs while domain 1 is capped takes this into account, and divide the credits that would be awarded to domain 1 between the other two. This means that the priority of a domain should not be larger than the percentage of the CPU allocated to its cap, or slightly surprising results will occur.

The scheduler ticks every 10ms, subtracting credits from the running VCPU, and caps the minimum number of credits as the number that would be achieved by a process running for one complete time slice having started with no credits.

This minimum value has little effect on the scheduling algorithm. If one VCPU is getting enough runtime to be exceeding the minimum threshold, the others must be either capped or idle. Because idle and capped VCPUs are ignored when determining the allocation of credits, the running VCPU will get more credits than it otherwise would, balancing out the drop. When the other VCPUs have work to do again, they will be factored into the credit allocation and the currently running VCPU will be throttled back to its fair share.