A Brief Description of the Load Balancer

|  |  |
| --- | --- |
| Status: | Released |
| Version: | 1.00 |
|  |  |
| Team/Department : | Kernel/Core Development |

Contents

1 Purpose and Scope 3

2 Introduction 3

3 New Load Balancer 3

3.1 Periodic Rebalance 4

3.2 Idle Pull 6

3.3 Priority Balancer 7

4 Further Information 8

4.1 References 8

4.2 Open Issues 9

4.3 Document History 9

# Purpose and Scope

This document describes the new load balancer in the kernel.

Please note that the load balancer is not necessarily finalised and could change in the future as new data on its performance and new hardware architectures become available.

# Introduction

The current SMP kernel (as present in ^3) , sometimes referred to as the SMP Safe kernel, contains a primitive form of load balancing that is simply there to allow thread migration. This has enabled SMP safe testing to take place so that engineers can ensure their code will work on SMP systems. The scheduling implementation in this kernel is very simple:

* For each CPU there is a thread ready list. This makes the system more scalable and reduces lock contention
* Within each CPU scheduling works exactly as it did in the EKA2 kernel. Highest priority runs thread always runs first to the exclusion of any thread in that ready list. Threads of equal priority are round robined. Threads have a finite timeslice, usually 20ms
* When a thread goes from being blocked to being ready to run, the scheduler needs to decide which CPU to move it to. This is the only load balancing decision present in the ^3 kernel and the only thing that can cause thread migration. The scheduler chooses the CPU that is running the lowest priority thread. If there is more than CPU running lowest priority thread (e.g. all idle) and the last CPU is amongst those, then it chooses the last CPU

This form of load balancing is only useful for test purposes and cannot be productised. It has a number of pathologies:

1. The second highest priority ready thread does not always get to run even if more than one CPU is active, since it may be stuck on the same ready list as the highest priority thread.
2. If a high priority thread wakes up a lower priority thread, the lower priority thread is likely to be assigned to a different core to the thread which woke it up. This is a frequently occurring case, for example threads woken up by a timer expiry, by a user input event or by a device driver signalling incoming data. This will often cause inefficient use of CPUs, especially on a two-core system, since all threads woken up in this fashion end up on the other core and the timer or driver thread will probably not run for very long, after which time that core may go idle and leave all the work on the other core.
3. Synchronous IPC will often end up ping-ponging between cores. If you start with just the client thread running, it sends a message to the server, which wakes up and is assigned to the other (idle) core. The client then immediately blocks waiting for the reply, so the first core goes idle. When the server completes the message, the client wakes up and is assigned to its original core.
4. The system can persist indefinitely with one core busy and all other cores idle, even if there are several runnable threads.
5. If there are several equal priority threads they will not in general get equal execution time. For example if there are 2 processors and 3 equal priority threads the best that you can get is one thread gets an entire CPU to itself and the other two get 50% of a CPU each.

In order to solve most of these issues and improve performance a new load balancer has been written. The new load balancer is a combination of priority balancing, idle pull and periodic load distribution.

# New Load Balancer

The new scheduler has a number of similarities to its predecessor. There is still one ready list per CPU and no global ready list as such. As in the previous design, for each CPU there is a sub-scheduler which owns the ready list, and handles the scheduling, for its CPU. The load balancing however adds new components and changes the old behaviour. It is composed of three different parts:

* Periodic Rebalance
* Priority Balancer
* Idle pull

Each of these is described in more detail in the following sections.

## Periodic Rebalance

This component periodically checks which threads have been active in the system, and then redistributes those threads amongst the engaged (not retired) CPU ready lists to equalize load. Threads are considered active if they are either running, ready to run, or waiting on a mutex (NFastMutex or DMutex). The reason for considering a thread as active when waiting on a mutex is that the thread actually wants to run, but it is being blocked because another thread is accessing the critical section. This contrasts with being blocked on a semaphore for example, where the thread has voluntarily yielded.

The periodic rebalance (PRB) runs every 107 nanokernel ticks. It runs in its own thread which is also subject to balancing. It is driven by a standard NTimer. The choice of periodicity is driven by the fact that 107 is prime number so the balancer is less likely to clash with other periodic activities in the system. Currently the value used for this timer is not configurable or subject to any dynamic adjustment.

The PRB also has a mechanism to turn itself off when the system goes idle. Basically when it runs it detects if the only thing that has run in the last cycle (last 107 ticks) is itself, and in this case it will turn off. It achieves this by checking what threads have been made ready in the last cycle, if these only include only DfcQ1 and the load balancer thread then the PRB will turn itself off. DfcQ1 underlies the timer that actives the load balancer thread. When a new thread is made ready the PRB is reactivated.

The PRB makes use of the following lists:

* iLbQ list: there is one these per CPU, threads go onto this list when they are made ready and leave it when the PRB runs. They also leave the list if the thread dies (or for a group if the group is destroyed). Generally speaking though this list is simply a place holder to store all threads that have been made ready between runs of the PRB. The list is locked using the ready list lock, so when a thread comes onto the list (when made ready) no additionally locking actually takes place as the lock is already held.
* iBalanceList: this contains threads that are of interest to the PRB. These are threads that have not been made ready in the last cycle of the PRB, but that have been previously processed by the load balancer and not yet dropped of the list due to inactivity. The threads may or may not be very active. Threads get added to this list if the PRB processes them as candidates for migration or threads that are active enough that are of interest to balancing. Threads leave this list if they have not been active for a while (currently 200ms see below), or if they are dying, or have very small active times but tend to get CPU when they need it.
* Internally the balancer also has a couple of temporary lists which it uses to store threads and to decide how to apportion them to the engaged CPUs.

The balancer computes and makes use of the following stats:

**Active time:** This is the % time since last PRB cycle that the thread has been active, i.e. ready to run, running, or waiting on a mutex (NFastMutex or DMutex).

**Run time:** This is the % time spent running on the CPU in the last PRB cycle.

**Ratio of run time to active time:** This can be thought off as a quality of service indication. Values close to 1 indicate threads that have enjoyed high CPU access, values close to 0 indicate the opposite.

Averages over time of the above values are also maintained.

The PRB also computes a measure to indicate how CPU bound a thread is. A thread that is very CPU bound is considered to be “heavy”. When a thread is new, this value is initialised on the basis of priority, low priority threads are considered heavy and high priority threads are considered light. Thereafter the value is updated on basis of recent activity by the PRB every time it runs. Basically if in 7 or more out the last 8 PRB cycles the thread's active time was bigger than 90% then the threads is considered to be heavy, otherwise it is considered light

When the PRB runs it first transfers all then all threads from iLbQ lists and iBalanceList into a single temporary queue. The balancer then scans this queue and updates the run, active and quality of service statistics for each thread in it. It also prunes those threads that have no interest to load balancing. This is done by checking when the thread was last active. If a thread has not been active in the last 200ms or more then the PRB will not keep it in the iBalancerList. The thread is said to be dropped. A thread which has used only a small percentage (currently <1%) of a CPU time in the last period and which has had relatively good QOS (>50%) is considered 'gravel' (or background noise) and will also be dropped. For all remaining threads, that are not dropped, if there is more than one active CPU and a thread is not cycling[[1]](#footnote-1), then it is added to sorted queue which is then used for load balancing. If a thread is cycling then it will added back into iBalancerList instead, so that it can be used in the next run of the PRB. If there is only one active CPU (because the others are retired) then the threads are placed in the iBalancer list and the PRB is abandoned as there are no CPUs available to spread the load.

If there are threads in the sorted list then the PRB can proceed. This list is sorted according to the following criteria: firstly priority of the threads, highest first. Secondly, on how CPU bound threads are, light threads go first then heavy. The third criteria for sorting, differs depending on the whether the threads are light or heavy:

**Light:** Lights threads are sorted in descending order of average run time.

**Heavy Threads:** Are sorted in descending order of quality of service ratio.

For each CPU the PRB maintains a measure of availability. This represents how much load a CPU can take. Before going through the sorted list this availability is initialised to be 100 %. Then the an estimation of the background noise or activity, referred to as gravel above, is divided by the number of engaged CPUs and then subtracted from each CPU’s availability. Then the PRB goes though the sorted list assigning threads to each CPU and updating the availability of each CPU based on the expected load of those threads assigned to it. For each priority in descending order the algorithm first assigns the light threads as follows:

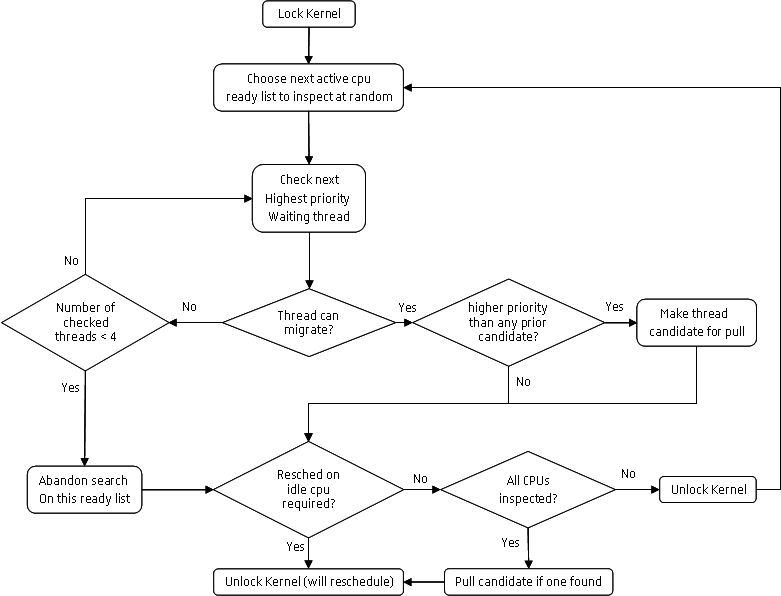
* Find the CPU with highest availability, CPUx. If the thread has run recently, and the last CPU the threads ran on has high availability then use that CPU.
* Assign the thread to CPUx and subtract from its availability the expected load of the thread.

Then heavy threads are done. If the number of heavy threads is smaller than the number of available CPUs then they assigned in the same way as light threads. The only difference is that each used CPU is marked as being maxed out. If the number of heavy threads is larger than the number of available CPUs, then each CPU takes a proportion of the heavy threads. Generally the proportion is linear to the amount of capacity that CPU has left and the amount of heavy threads at that priority to balance. If the capacity left in the CPUs is very low then they heavy threads are just shared equally. Due to the ordering by quality of service, the net effect is that that if there are more heavy threads than active CPUs, then the threads are rotated so that on average each thread gets the same share of CPU.

The relevant code is located in /sf/os/kernelhwsrv/kernel/eka/nkernsmp/nk\_bal.[cpp|h]

## Idle Pull

When a CPU that is not being retired goes idle, just before invoking the idle handler it searches the ready lists of other CPUs to see if there is any work it could be doing.



The diagram above depicts how the idle pull mechanism operates. In an outer loop, all the engaged CPUs are checked in a random order. For each CPU, in an inner loop, the idle pull code searches for the highest priority thread that can be migrated to the idle CPU. This inner loop search holds the lock of the ready list that is being checked. A thread can migrate if the following conditions are met:

* Thread is not currently running
* Thread it is not frozen[[2]](#footnote-2)
* Thread has an affinity that would allow it to run on the idle CPU
* Thread is not holding a fast mutex
* Thread is not cycling
* Thread is not currently migrating

If all conditions are not met, then the search moves onto the next highest priority thread in the ready list. Only up to four threads are checked to limit the amount of time the ready list lock is held. If the conditions are met, and the threads priority is higher than any previous candidate (or there was no prior candidate), the thread is the made the candidate for idle pull. If a candidate is found, or the limit of four thread look ups is met, the search moves on to the next CPU’s ready list. Once all engaged CPUs have been searched, if a candidate thread is has been found it is migrated to the idle CPU. During the search the kernel is locked on the idle CPU, however if another thread is made ready on it, the search is aborted and the kernel is unlocked to allow the reschedule to take place.

The relevant code is located in the NKern::Idle() and TSubScheduler::IdlePullSearch functions and the SIdlePullThread class (/sf/os/kernelhwsrv/kernel/eka/nkernsmp/sched.cpp)

## Priority Balancer

This component of the scheduler runs when a thread becomes ready to run after it has blocked. This component chooses a CPU for the thread when there is no preferred CPU (set by periodic rebalancer) or transient CPU (set by idle pull). It checks each engaged CPU in turn starting with the CPU the thread last ran on, and for each one obtains the following information:

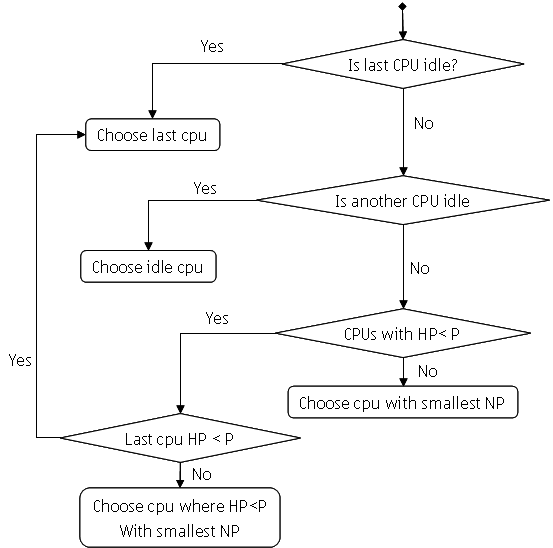
* Is the CPU idle?
* HP[x] : The highest priority thread running on CPUx.
* NP[c,x] : The number of threads on CPUx in the same priority class as the thread being made ready.

Of these metrics, only the last one is new. Thread priorities are now broken up into four classes, depending on the priority value:

|  |  |
| --- | --- |
| Class | Priority value |
| 0 | P<12 |
| 1 | 16>P>=12 |
| 2 | 27>P>=16 |
| 3 | P>=27 |

Each sub-scheduler maintains a count of how many ready threads per priority class it owns.

Based on these metrics the priority balancer picks a CPU as follows (where P is the readying thread’s priority):



1. If the thread would preempt on the last CPU it run on, and this CPU was idle choose that. (Because loop starts on the last CPU the loop would break early in this situation)
2. Otherwise if another CPU is idle choose that.
3. Otherwise check all those CPU with a lower HP than this thread and:
   1. If last CPU the thread run on has lower HP choose that.
   2. Otherwise choose the CPU with lowest number of threads in same priority class as this one
4. Otherwise (if all CPUs are running higher HP than this thread) choose the one with lowest number of threads in same priority class

The relevant code is located in the NSchedulable::ReadyT function (/sf/os/kernelhwsrv/kernel/eka/nkernsmp/sched.cpp)

# Further Information

## References

| No. | Document Reference | Version | Description |
| --- | --- | --- | --- |
|  |  |  |  |

## Open Issues

None.

## Document History

| Date | Version | Status | Description |
| --- | --- | --- | --- |
| 2009-12-14 | 0.1 | Draft | Initial version |
| 2010-02-05 | 0.2 | Draft | Added some corrections from code review |
| 2010-02-08 | 0.3 | Draft | Further updates from core review |
| 2010-05-26 | 0.4 | Draft | Updates based on recent kernel developments |
| 2010-05-26 | 1.0 | Released | Moved to EPL template |

1. Internally the kernel provides a mechanism whereby a thread can execute code on all CPUs. This is known as a broadcast, and is there primarily to support certain cache maintenance functions on ARM11MP. To perform the broadcast, the thread is cycled onto each CPU in turn and allowed to execute a function. [↑](#footnote-ref-1)
2. A thread can choose to invoke the NKern::FreezeCpu API. This function will “freeze” that thread to the CPU it is currently running on. After that the thread cannot migrate to another CPU until t calls NKern::EndFreezeCpu. [↑](#footnote-ref-2)