MODULAR SCHEDULER FRAMEWORK

AND

FAIR SCHEDULER FOR MINIX 3

A Project submitted in the partial fullfillment for the requirements for the award of degree of

BACHELOR OF ENGINEERING IN INFORMATION TECHNOLOGY

By

DEBJIT BISWAS - BE/250/05
DHRUV SHUKLA - BE/327/05

DEPARTMENT OF INFORMATION TECHNOLOGY

BIRLA INSTITUTE OF TECHNOLOGY
MESRA, RANCHI

2009

Certificate Of Approval

The foregoing project entitled **"Modular Scheduler Framework and Fair Scheduler for MINIX3"**, is hereby approved as a creditable study of research topic and has been presented in satisfactory manner to warrant its acceptance as prerequisite to the degree for which it has been submitted.

It is understood that by this approval, the undersigned do not necessarily endorse any conclusion drawn or opinion expressed therein, but approve the project for the purpose for which it is submitted.

(Internal Examiner) (External Examiner)

(Head of the Deparment)

Declaration Certificate

This is to certify that the work presented in the project entitled **"Modular Scheduler Framework and Fair Scheduler for MINIX3"** in partial fulfillment of the requirement for the award of degree of **Bachelor of Engineering in Information Technology** of Birla Institute of Technology, Mesra, Ranchi is an authentic work carried out under my supervision and guidance.

To the best of my knowledge, the content of this project does not form a basis for the award of any previous Degree to anyone else.

Date:

Abhijit Mustafi
Dept. of Computer Science
Birla Institute of Technology
Mesra, Ranchi

Contents

Chapter 1

Introduction

When Mr. Andrew S. Tanenbaum wrote the MINIX operating system, he wouldn't have thought that it would inspire Linus Torvalds to write Linux, the most revolutionary Operating System in the recent times. Back then MINIX was only meant to be an educational operating system. Now with the introduction of an entirely new version of MINIX - MINIX 3, things are different. MINIX 3 now aims a place in the operating system market, and Mr. Tanenbaum has decided to target the fast expanding embedded systems market and resource limited systems. MINIX 3's unique features include its high reliability and small kernel size due to its microkernel architecture.

Work done by Linux kernel developers Con Kolivas and Ingo Molnar on the Linux process scheduler has shown how fairness of the scheduling algorithm can help the interactivity and responsiveness of the system.

1.1 What is MINIX 3?

MINIX 3 is a new open-source operating system designed to be highly reliable, flexible, and secure. It is loosely based somewhat on previous versions of MINIX but is fundamentally different in many key ways. MINIX 1 and 2 were intended as teaching tools; MINIX 3 adds the new goal of being usable

as a serious system on resource-limited and embedded computers and for applications requiring high reliability.

MINIX 3 is extremely small, with the part that runs in kernel mode under 4000 lines of executable code. The parts that run in user mode are divided into small modules, well insulated from one another. For example, each device driver runs as a separate user-mode process so a bug in a driver (by far the most significant source of bugs in any operating system), cannot bring down the entire OS. In fact, most of the time when a driver crashes it is automatically replaced without requiring any user intervention, without requiring rebooting, and without affecting running programs. These features, the tiny amount of kernel code, and other aspects significantly enhance system reliability. MINIX 3 is initially targeted at the following areas:

- Applications where very high reliability is required
- Single-chip, small-RAM, low-power, $100 laptops for Third-World children
- Embedded systems (e.g., cameras, DVD recorders, cell phones)
- Education (e.g., operating systems courses at universities)

1.2 What is a scheduler?

When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. When more than one processes are in the ready state, and there is only one CPU available, the operating system must decide which process to run first. The part of the operating system that makes this choice is called the scheduler. The algorithm it uses is called the scheduling algorithm. See also [Tan06] and [Tan01].

1.3 Problems with the MINIX 3 scheduler

MINIX 3 uses a simple priority based multilevel scheduling algorithm. Theoretically multilevel scheduling algorithms are the best regarding responsiveness as discussed by Kleinrock [Kle76]. This increase in responsiveness results in a bias towards the interactive tasks. Also, many heuristics are used by most scheduling algorithms to identify the I/O bound processes and prioritise them, which puts the CPU bound processes at a disadvantage. These heuristics can never be perfect.

The MINIX 3 scheduler code is spread among many files which makes modifying the scheduler extremely difficult. At present, the scheduler is very simple but when it expands maintaining it would be difficult. Also MINIX 3 has to fulfil the role of an educational OS, and a complicated and confusing code structure may discourage the students.

Since MINIX 3 is based on the microkernel architecture, there are several system tasks (e.g. the file server, process manager etc.) and drivers which run outside the kernel as user processes. There is no mechanism by which the system tasks can be scheduled differently or ahead of user processes. What is needed is set of different scheduler policies.

The shortcomings of the present MINIX 3 scheduler can be summarised as:

- MINIX 3 scheduler needs to more interactive and fair, as humans expect simple machines to much more responsive than a complex one and MINIX 3 is aimed at embedded systems.
- The scheduler code is disorganised, which may discourage students from studying it.
- Existence of only a single scheduler policy.

Chapter 2

Project Goals

The Goals of this project are:

1. Implementation of a fair scheduler in the MINIX 3 operating system and then studying the interactive performance of the system.
2. Implementation of a modular scheduler framework which will facilitate the introduction and new scheduling policies. Note that the modularity will only be at source code level.
3. Implementation of scheduler related system calls based on the POSIX [IEEE04] standard.

Chapter 3

Background

Proportional share resource management provides a flexible and useful abstraction for multiplexing a scarce resource among user and applications. The basic idea is each client has an associated weight, and resources are allocated to the clients in proportion to their respective weights. Because of its usefulness, many proportional share scheduling mechanisms have been developed.

Proportional share scheduling mechanisms were first developed decades ago with the introduction of weighted round-robin scheduling. Later, fair-share algorithms based on controlling priority values were developed and incorporated into some UNIX based operating systems. These earlier mechanisms were typically fast, requiring only constant time to select a client for execution. However, they were limited in the accuracy with which they could achieve proportional sharing. As a result, starting in the late 1980s, fair queueing algorithms were developed first for network packet scheduling and later for CPU scheduling. These algorithms provided better proportional sharing accuracy.

Previous proportional sharing mechanisms can be classified into four categories: those that are fast but have weaker proportional fairness guarantees, those that map well to existing scheduler frameworks in current commercial operating systems but have no well-defined proportional fairness guarantees,

those that have strong overhead, and those that have weaker proportional fairness guarantees but have higher scheduling overhead. The four categories correspond to round-robin, fair-share, fair queueing, and lottery mechanisms.

3.1 Fair-Share

Fair-share schedulers arose as a result of a need to provide proportional sharing among users in a way compatible with a UNIX-style time-sharing framework. In UNIX time-sharing, scheduling is done based on multi-level feedback with a set of priority queues. Each client has a priority which is adjusted as it executes. The scheduler executes the client with the highest priority. The idea of fair-share was to provide proportional sharing among users by adjusting the priorities of a user's clients suitably. Fair-share provides proportional sharing by effectively running clients at different frequencies, as opposed to WRR which only adjusts the size of the clients' time quanta. Fair-share schedulers were compatible with UNIX scheduling frameworks and relatively easy to deploy in existing UNIX environments. Unlike round-robin scheduling, the focus was on providing proportional sharing to groups of users as opposed to individual clients. However, the approaches were often ad-hoc and it is difficult to formalize the proportional fairness guarantees they provided. Empirical measurements show that these approaches only provide reasonable proportional fairness over relatively large time intervals. It is almost certainly the case that the allocation errors in these approaches can be very large.

3.2 Proportional Share Fairness

We use the same definition of fairness as Caprita et al. [CCN+05] which is based on the Generalized Processor Sharing model [Kle76].

The idea is that there are multiple clients requesting access to a shared resource. Further, these clients have different privileges which are expressed as weights. The weights define how much service a client should receive in

relation to other clients and how large its share of the resource should be. For example, a client with a weight of 2 should receive twice as much service as a client with a weight of 1. Further, in case the sum of all weights is 10, it should receive 20% of the available resource.

Formally, this can be expressed as follows: Let $W_{C_a}(t_1, t_2)$ be the amount of service received, i.e. execution time, by a client C_a in the interval between t_1 and t_2. Given many arbitrary clients, C_j and their associated positive weights ϕ_j a scheduling algorithm is fair if and only if

$$\forall a. W_{C_a}(t_1, t_2) = (t_2 - t_1)\frac{\phi_a}{\sum_j \phi_j} \tag{3.1}$$

is true for all intervals $[t_1\ ,\ t_2]$.

This means that each client should receive service exactly *proportional* to its *share* of the total weight.

Service Error Obviously, no real world scheduling algorithm can be entirely fair. This is due to hardware and efficiency constraints that make it necessary to allocate the CPU in discrete time units (TUs). As only one client can advance at one point in time, a client gets ahead of its proportional share of service while running and falls behind while being queued, thus creating an unfair state. The difference between what a client should have received in an interval and what it actually received is called the *service error*.

Let $W_{C_a}(t_1, t_2)$ denote the work actually received by C_a in $[t_1, t_2]$. Then the service error $E_{C_a}(t_1, t_2)$ of a client C_a is defined as:

$$E_{C_1}(t_1, t_2) = W_{C_a}(t_1, t_2) - (t_2 - t_1)\frac{\phi_a}{\sum_j \phi_j} \tag{3.2}$$

A negative service error indicates that a client has fallen behind, while a positive service error signals disproportionate resource consumption. Note

that while it may not be possible to avoid service errors at all times, it is at least possible to achieve perfect fairness every $\sum_j \phi_j$ rounds. Therefore it must be the goal of real world proportional share schedulers to minimise the maximum absolute service errors in between. Ideally it should be possible to give constant bounds for the maximum and minimum service errors for a scheduling algorithm, i.e. it should have an $O(1)$ service error.

Virtual Time When deciding which client should receive service next, a proportional share scheduler has to consider past resource allocation. Ideally, given two clients C_a and C_b, we would like to simply compare their received service times $W_{C_a}(t_1, t_2)$ and $W_{C_b}(t_1, t_2)$ and select the one which received less service so far. But doing so in terms of the absolute number of allocated time units is pointless, as clients with larger weights should receive more service than clients with smaller weights.

A measure for a client's normalised progress can be obtained by first scaling the received service by the weight to account for different rights to the resource. The result is called *virtual time* [DKS89, NVZ01] and is defined as follows:

$$VT_{C_a}(t) = \frac{W_{C_a}(0, t)}{\phi_a} \tag{3.3}$$

The virtual times of clients can be used to compare their past resource allocation directly, as differences in weights allow for proportional differences in absolute service times.

Queue Virtual Time While virtual time allows us to tell whether a client is ahead of another client easily, it does not tell us whether these clients are ahead or behind of their proportional share, i.e. whether they have a positive or negative service error. Starting from a fair state, we can derive a term that allows for such a comparison.

In a fair state, equation(3.1) holds. Assuming $t_0 = 0$, the equation can be rewritten as follows:

$$\begin{aligned} \forall a. W_{C_a}(0,t) &= t * \frac{\phi_a}{\sum_j \phi_j} \\ \Leftrightarrow \forall a. \frac{W_{C_a}(0,t)}{\phi_a} &= t * \frac{1}{\sum_j \phi_j} \\ \Leftrightarrow \forall a. VT_{C_a}(t) &= \frac{t}{\sum_j \phi_j} \end{aligned}$$

The term on the right side of the last equation is independent of the client weight ϕ_a. This shows on the one hand that in a fair state all virtual times must be equal and on the other hand gives us an easy to compute figure allowing for quick comparisons of clients virtual time. This is called *queue virtual time* [NVZ01].

$$QVT(t) = \frac{t}{\sum_j \phi_j} \tag{3.4}$$

As the virtual time of a client should always be equal to the queue virtual time, we can infer from a smaller virtual time that the client has fallen behind its proportional allocation and vice versa for larger virtual times.

3.3 Defining Interactivity and Reponsiveness

These two terms, usually used interchangeably in operating system literature are difficult to define quantities. They are heavily dependent on the perception of the user and are difficult to measure (if one cannot define something how can it be measured).

We would prefer to define these quantities as:

Responsiveness is the rate at which the processes can proceed under different kind of loads.

Interactivity is the scheduler latency that a process experiences under dif-

ferent load conditions

The two definitions are very close, but there is a subtle difference. Responsiveness will tend to measure the long-term behaviour of a process, whereas interactivity will measure the short-term behaviour or the 'snappiness'.

Chapter 4

Virtual Time Round Robin

Virtual Time Round Robin (VTRR) was developed by Prof. Jason Nieh et al. at the Network Computing Lab of Columbia University, NY, and published at the 2001 USENIX Annual Technical Conference. It strives to provide good proportional fairness while maintaining $O(1)$ time complexity. The algorithm is also remarkable for its simplicity and ease of implementation [NVZ01].

4.1 Background

There are two ways to allocate shares of a time multiplexed resource to weighted clients:

1. Adjusting the time slice length.
2. Adjusting the frequency of resource allocation.

The first method is easy to implement and due to its simple structure offers $O(1)$ time complexity. Though it has been used extensively in historic operating systems , it has several drawbacks. It is often used in a multi-level feedback scheme (e.g. the traditional UNIX scheduler) that is hard to tackle

analytically. Further, tests of weighted round robin, which does not use a feedback scheme and can therefore be analyzed, show that the resulting long time slices result in large service errors of both the executing and the waiting clients [NVZ01].

The problem of large service errors lead to the development of the second method [DKS89]. By allocating only small, fixed length time slices, and selecting clients with different frequencies, it is possible to achieve service errors bounded by 1 time unit [BZ96]. The problem that arises is how to determine the next client that may receive service without violating the service error bounds. Historically, this has been done by using priority queues and heaps. Unfortunately this causes a time complexity of $O(logn)$ or even $O(n)$. Thus, at the end of the '90s, known algorithms either had weak proportional sharing accuracy or high scheduling overhead [NVZ01].

4.2 The Algorithm

The high scheduling overhead of algorithms such as weighted fair queueing [DKS89] result from the need to maintain a sorted queue. This is because that clients with large virtual times should not be selected earlier than clients with small virtual times, as the latter have to catch up. In order to achieve $O(1)$ time complexity this constant reordering must be avoided. The goal of VTRR is to combine the efficiency of round robin without giving up too much proportional sharing accuracy.

Fundamental Idea Obviously a client's selection frequency is directly proportional to its weight, as it does not make sense to give more service to clients with smaller weights. Put the other way around, a frequency adjusting proportionally sharing scheduler must select a client C_a with a weight ϕ_a at least as often as all other clients C_o with $\phi_o \leq \phi_a$. We can use this natural order of the clients for efficient client selection:

1. Sort all clients C_i by their weight ϕ_i in descending order. We then can

state:

$$\forall C_x, C_y.x < y \rightarrow \phi_x \geq \phi_y \tag{4.1}$$

2. Beginning at the front of the queue, select clients in a round robin manner.

3. Before selecting the next client, check whether it would get too far ahead of its fair allocation. If so, reset to the beginning of the queue.

This simple scheme works because a client C_x with weight ϕ_x in the front of the queue will at least be selected as often as all clients C_y where $y < x$ and $\phi_y \leq \phi_x$, as a client from the back is never selected before all other clients in front of it have been selected. To avoid giving too much service to the clients in the back, a reset criteria, called virtual time inequality, is needed. Because of the ordering of the clients, we only have to test the next client, allowing us to perform the scheduling decision in constant time. Note that sorting the queue is only necessary once during initialization and on the (very seldom) event of a weight change.

Recall from the definition of service error that at least at the end of every $\sum_j \phi_j$ rounds, perfect fairness can be achieved. This period is called a scheduling cycle. To attain perfect fairness at the end of every cycle, VTRR keeps a time counter for each client and imposes a time counter invariant on the queue. The time counter invariant states that in addition to the ordering by descending weight, the queue should also be ordered by descending time counters. As we will see, this invariant is used in combination with the reset criteria to guarantee fairness at the end of scheduling cycles.

Detailed Description VTRR maintains the queue virtual time QVT and three values for each client C_a :

1. The share ϕ_a which is never changed by VTRR.

2. The virtual finish time VFT_a . It determines the client's progress.

3. A time counter CT_a . The time counter is initialized to the client's weight.

At the end of each time slice, the queue virtual time and the running client's virtual finish time are incremented by one scaled time unit and the client's time counter is decremented by one.

After updating the state variables of the current client C_{cur} , the next client is scheduled. The decision whether to select the client following in the queue $C_{cur+1} = C_{next}$ or to reset to the beginning and select C_1 is based on two criteria:

1. The time counter invariant:

$$CT_{cur} \geq CT_{next} \tag{4.2}$$

   If the inequality does not hold, i.e. $CT_{cur} < CT_{next}$, the invariant is violated and the next client C_{next} is selected without any further questions in order to restore the invariant.

2. The virtual time inequality .

$$VFT_{next}(t) - QVT(t+1) \leq \frac{1}{\phi_{next}} \tag{4.3}$$

   If the inequality is true, C_{next} is selected. Otherwise C_1 is selected.

What is the reasoning behind the inequalities?

The time counter invariant guarantees, that VTRR does not "forget" a client at the end of the queue. It holds at the start of a cycle and the only way it might get violated is by decreasing the time counter of the current task CT_{cur} that was previously equal to the counter CT_{next} . In this case the difference is exactly 1 and can be corrected by running C_{next} once. It ensures that we cannot reset to the first client C_1 at the end of a cycle before attaining perfect fairness. If after a reset the time counter of the first client equals zero ($CT_1 = 0$), we know due to the time counter invariant that all time counters must equal zero, signaling the end of a cycle. Additional maintenance is needed in that case as all time counters have to be reinitialized to each client's weight.

Summarized VTRR can be expressed as:

"Perform a simple round robin over the sorted queue as long as the time counter invariant is violated or the next client has to catch up, reset to the first client otherwise."

4.3 Dynamic Considerations

In a real world setup, clients may arrive and block at any time. While this is – strictly speaking – not a responsibility of the scheduler, fairness constraints should not be violated in spite of modifications of the run queue. Especially it should not be possible to receive undue service by blocking for short times. Therefore VTRR has to handle the arrival of new clients, the blocking of queued clients and the return of blocking clients.

New Client Arrival New clients have not yet received any service, so their virtual time is zero. Similarly, the time counter has not yet been decremented, so it equals the client's weight. But obviously this would cause VTRR to misbehave, as the algorithm depends on the time counter in- variant and comparable virtual finish times. The solution is to initialize a new client with values that fit into the current state.

As a new client is neither behind nor ahead of its allocation, the virtual time should equal the queue virtual time , allowing us to derive the correct virtual finish time. Similarly, the correct time counter value can be calculated from the global progress in the current scheduling cycle, which can be expressed as the ratio of the sum of all counters to the sum of all weights.

Let C_n be the new client arriving at t_0 . The virtual finish time and time counter of C_n will be set according to:

$$VFT_n \leftarrow QVT(t_0) + \frac{1}{\phi_n} \tag{4.4}$$

$$TC_n \leftarrow \phi_n * \frac{\sum_j TC_j}{\sum_j \phi_j} \tag{4.5}$$

Keep in mind that C_n is not yet part of the sums used for the calculation of TC_n . After the state variables are properly initialized, C_n has to be inserted at the correct position in the sorted queue.

Client Removal A client can be removed from the queue without disturbing the rest of the clients. Therefore client removal should be straight forward. But as the client may return, e.g. it might be waiting for IO operations to finish, the current time counter and the current virtual finish time have to be kept, as they are needed for reinsertion. Further, it is advantageous to save pointers to the previous and next client in the queue, as it may speed up the reinsertion of the client.

Blocked Client Return To reinsert returning clients, e.g. clients that have run before, the same steps are necessary that are performed for for new clients, too. Additionally safeguards have to be implemented to prevent a client from gaming the system. Particularly, a client may not gain a smaller virtual finish time than it had before blocking and it may not gain a larger time counter, either.

Let C_r be the client returning at t_1 . Further, let VFT_r and CT_r be the virtual finish time and time counter saved during blocked client removal. The virtual finish time and time counter will be updated according to:

$$VFT_r \leftarrow max\left\{QVT(t_1) + \frac{1}{\phi_r}, VFT_r^*\right\} \tag{4.6}$$

$$TC_r \leftarrow min\left\{\phi_r * \frac{\sum_j TC_j}{\sum_j \phi_j}, CT_r^*\right\} \tag{4.7}$$

The last step is to reinsert the client into the queue.

4.4 Why VTRR ?

We chose VTRR due to a variety of reasons, which are as follows:

- VTRR effectively has a time complexity of $O(1)$. By reusing the mutilevel priority queue structure of the MINIX3 the client insertion and reinsertion was reduced to $O(1)$ from $O(n)$.
- The algorithm is very easy to implement. We have implemented the algorithm by modifying less than 100 lines of code.
- A lot scheduler related code and data structures from the original MINIX3 scheduler could be reused.

4.5 Implementation Details

The following changes were applied to the MINIX3 kernel code to implement VTRR:

Process Parameters added:

p_actual_priority MINIX3 priorities are in reverse order i.e. zero priority is the highest and 15 is the lowest. This had to be reversed so that the time counter decrement can be performed.

p_time_counter This is the time counter as defined by the VTRR algorithm.

p_vft This is the virtual finish time of the process.

Scheduler Parameters added:

qvt This is Queue Virtual Time as defined by the VTRR algorithm.

qvt_stride This is the amount by which qvt is incremented after each quantum.

total_share This the sum of all the weights (p_actual_priority) of the processes that are runnable.

Chapter 5

Modular Scheduler

Scheduling Classes have been introduced which implement an extensible hierarchy of scheduler modules. These modules encapsulate scheduling policy details and are handled by the scheduler core without the core code assuming about them too much.

The introduction of scheduling classes makes the core scheduler quite extensible. These classes (the scheduler modules) encapsulate the scheduling policies. Each scheduler module has to implement each of the functions suggested by the generic scheduling class.

Some Advantages of this design strategy are:

1. **Easier augmentation and exclusion of scheduler policies**: This kind of design allows a programmer to easily introduce new scheduling policies into the kernel. The original code is modified to a minimal extent. This also means that testing of new scheduling algorithms become very easy.

2. **Separate policy for real time tasks**: Real time tasks usually require special treatment by the scheduler. A different policy for real-time tasks can be easily implemented in this kind of design.

3. **Easier for students to understand the code**: Since MINIX 3 is

widely used as an educational operating system in many universities across the world, many of MINX 3's users are students. The design strategy used here will allow students to easily understand the source code and experiment with their own code.

5.1 Design Issues

The major design issue was which functions should be removed from the core scheduler and assigned to the scheduler modules The initial design included only the following functions in the module(these have been implemented):

- The enqueue function which is run each time a task becomes runnable. Based on the scheduling policy of the process a particular enqueue funtion is called which enqueues the process in the whatever datastructure that is being used the scheduler policy.
- The dequeue function which is run when a task is no longer runnable.
- The pick_proc function chooses the most appropriate process eligible to run next.

Later the following additional functions were added (these have not been implemented):

- The preempt_curr is run before a task is preempted.
- The task_tick function is run at every clock tick.

5.2 Implementation

To implement the modular scheduler framework the following step were followed:

- A header `sched.h` was added as suggested by the The Open Group Base Specifications Issue 6 [IEEE04]. The constants `SCHED_FIFO`, `SCHED_RR` were defined.
- The sturcture `sched_class` was defined which encapsulates all the functions that are to be implemented by a scheduler module i.e. the ones decided above.
- Two new attributes were added to the process structure `p_sched_policy` and `p_sched_class` were the first attribute is scheduler policy that is being used to schedule the process and the secon atrribute is a pointer to the scheduling class.
- Next all the scheduler related functions in the source code of the MINIX 3 kernel was replaced by function calls to the appropiate functions in the scheduler modules. All the calls were of the form `proccess_pointer->sched_class->function`. The `process_pointer` points to process in the context of which the function will be called. `sched_class` is the scheduling class which the process is member of.

Chapter 6

Future Plans

Unfortunately there are no formally proven bounds for the maximum and minimum service errors known. Further, while multi-processor capability could be added to VTRR, it is only specified for uni-processor systems, rendering it an ill choice for most modern operating systems. Therefore VTRR can be considered a step into the right direction, but it is far from an optimal solution.

Also how the algorithm behaves in the microkernel environment of MINIX is yet to seen. The algorithms performance has to be studied with this in mind.

- Implementing other fair schedulers such as GR^3
- Implementing VTRR using the modular scheduler framework.
- Measuring the performance of VTRR in microkernel architecture context.

Appendix A

Modular Scheduler Patch

Following is the patch to MINIX 3 version 3.1.3a which implements the modular scheduler framework:

```
diff -uNrB /mnt/hda4p3/src/kernel/main.c /mnt/hda4p3/src_copy/kernel/main.c
--- /mnt/hda4p3/src/kernel/main.c 2007-03-30 20:47:32.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/main.c 2008-10-25 01:27:21.000000000 +0530
@@ -71,7 +71,9 @@
  ip = &image[i]; /* process' attributes */
  rp = proc_addr(ip->proc_nr); /* get process pointer */
  ip->endpoint = rp->p_endpoint; /* ipc endpoint */
+ rp->p_sched_class = ip->sched_class;
  rp->p_max_priority = ip->priority; /* max scheduling priority */
+ rp->p_sched_policy = ip->sched_policy;
  rp->p_priority = ip->priority; /* current priority */
  rp->p_quantum_size = ip->quantum; /* quantum size in ticks */
  rp->p_ticks_left = ip->quantum; /* current credit */
diff -uNrB /mnt/hda4p3/src/kernel/proc.c /mnt/hda4p3/src_copy/kernel/proc.c
--- /mnt/hda4p3/src/kernel/proc.c 2007-03-21 15:15:01.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/proc.c 2008-10-25 01:43:14.000000000 +0530
@@ -44,6 +45,9 @@
 #include "proc.h"
 #include <signal.h>
 #include <minix/portio.h>
+#include "sched_classes.h"
+
+#define sched_class_highest (&rt_sched_class)

 /* Scheduling and message passing functions. The functions are available to
  * other parts of the kernel through lock_...(). The lock temporarily disables
@@ -497,32 +501,12 @@
  * The mechanism is implemented here.   The actual scheduling policy is
  * defined in sched() and pick_proc().
  */
-  int q;   /* scheduling queue to use */
-  int front; /* add to front or back */
-
 #if DEBUG_SCHED_CHECK
   check_runqueues("enqueue1");
```

```
   if (rp->p_ready) kprintf("enqueue() already ready process\n");
 #endif

-  /* Determine where to insert to process. */
-  sched(rp, &q, &front);
-
-  /* Now add the process to the queue. */
-  if (rdy_head[q] == NIL_PROC) { /* add to empty queue */
-      rdy_head[q] = rdy_tail[q] = rp;  /* create a new queue */
-      rp->p_nextready = NIL_PROC; /* mark new end */
-  }
-  else if (front) { /* add to head of queue */
-      rp->p_nextready = rdy_head[q]; /* chain head of queue */
-      rdy_head[q] = rp; /* set new queue head */
-  }
-  else { /* add to tail of queue */
-      rdy_tail[q]->p_nextready = rp; /* chain tail of queue */
-      rdy_tail[q] = rp; /* set new queue tail */
-      rp->p_nextready = NIL_PROC; /* mark new end */
-  }
-
+  rp->p_sched_class->enqueue(rp);
   /* Now select the next process to run, if there isn't a current
    * process yet or current process isn't ready any more, or
    * it's PREEMPTIBLE.
@@ -548,10 +532,6 @@
  * it has blocked.  If the currently active process is removed, a new process
  * is picked to run by calling pick_proc().
  */
-  register int q = rp->p_priority; /* queue to use */
-  register struct proc **xpp; /* iterate over queue */
-  register struct proc *prev_xp;
-
   /* Side-effect for kernel: check if the task's stack still is ok? */
   if (iskernelp(rp)) {
  if (*priv(rp)->s_stack_guard != STACK_GUARD)
@@ -563,23 +543,9 @@
   if (! rp->p_ready) kprintf("dequeue() already unready process\n");
 #endif

-  /* Now make sure that the process is not in its ready queue. Remove the
-   * process if it is found. A process can be made unready even if it is not
-   * running by being sent a signal that kills it.
-   */
-  prev_xp = NIL_PROC;
-  for (xpp = &rdy_head[q]; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {
-
-      if (*xpp == rp) { /* found process to remove */
-          *xpp = (*xpp)->p_nextready; /* replace with next chain */
-          if (rp == rdy_tail[q]) /* queue tail removed */
-              rdy_tail[q] = prev_xp; /* set new tail */
-          if (rp == proc_ptr || rp == next_ptr) /* active process removed */
-              pick_proc(); /* pick new process to run */
-          break;
-      }
-      prev_xp = *xpp; /* save previous in chain */
-  }
+  rp->p_sched_class->dequeue(rp);
+  if (rp == proc_ptr || rp == next_ptr) /* active process removed */
+    pick_proc(); /* pick new process to run */

 #if DEBUG_SCHED_CHECK
   rp->p_ready = 0;
@@ -630,19 +596,25 @@
  * clock task can tell who to bill for system time.
  */
   register struct proc *rp; /* process to run */
-  int q; /* iterate over queues */
+  const struct sched_class *class;
```

```
   /* Check each of the scheduling queues for ready processes. The number of
    * queues is defined in proc.h, and priorities are set in the task table.
    * The lowest queue contains IDLE, which is always ready.
    */
-  for (q=0; q < NR_SCHED_QUEUES; q++) {
-      if ( (rp = rdy_head[q]) != NIL_PROC) {
-          next_ptr = rp; /* run process 'rp' next */
-          if (priv(rp)->s_flags & BILLABLE)
-              bill_ptr = rp; /* bill for system time */
-          return;
-      }
+  class = sched_class_highest;
+  for ( ; ; ) {
+    if ( (rp = class->pick_proc()) != NIL_PROC ) {
+      next_ptr = rp;  /* run process 'rp' next */
+      if (priv(rp)->s_flags & BILLABLE)
+        bill_ptr = rp; /* bill for system time */
+      return;
+    }
+    class = class->next;
   }
   panic("no ready process", NO_NUM);
 }
diff -uNrB /mnt/hda4p3/src/kernel/proc.h /mnt/hda4p3/src_copy/kernel/proc.h
--- /mnt/hda4p3/src/kernel/proc.h 2007-02-01 23:20:02.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/proc.h 2008-10-25 01:22:47.000000000 +0530
@@ -21,6 +21,8 @@
   short p_rts_flags; /* process is runnable only if zero */
   short p_misc_flags; /* flags that do not suspend the process */

+  struct sched_class *p_sched_class;
+  char p_sched_policy;          /* current scheduling policy */
   char p_priority; /* current scheduling priority */
   char p_max_priority; /* maximum scheduling priority */
   char p_ticks_left; /* number of scheduling ticks left */
diff -uNrB /mnt/hda4p3/src/kernel/sched.h /mnt/hda4p3/src_copy/kernel/sched.h
--- /mnt/hda4p3/src/kernel/sched.h 1970-01-01 05:30:00.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/sched.h 2008-10-24 19:03:46.000000000 +0530
@@ -0,0 +1,26 @@
+#ifndef SCHED_H
+#define SCHED_H
+
+
+/*
+ * Scheduling policies
+ */
+#define SCHED_FIFO 1
+#define SCHED_RR 2
+#define SCHED_IDLE 3
+
+struct sched_param {
+ int sched_priority;
+};
+
+struct sched_class {
+ const struct sched_class *next;
+
+ void (*enqueue) (struct proc *rp);
+ void (*dequeue) (struct proc *rp);
+ struct proc * (*pick_proc) ();
+};
+
+
+#endif /* SCHED_H */
diff -uNrB /mnt/hda4p3/src/kernel/sched_classes.h
          /mnt/hda4p3/src_copy/kernel/sched_classes.h
--- /mnt/hda4p3/src/kernel/sched_classes.h 1970-01-01 05:30:00.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/sched_classes.h 2008-10-25 20:10:37.000000000 +0530
```

```
@@ -0,0 +1,8 @@
+#ifndef SCHED_CLASSES
+#define SCHED_CLASSES
+
+#include "sched_idle.c"
+#include "sched_rt.c"
+
+#endif /* SCHED_PROFILES */
diff -uNrB /mnt/hda4p3/src/kernel/sched_idle.c /mnt/hda4p3/src_copy/kernel/sched_idle.c
--- /mnt/hda4p3/src/kernel/sched_idle.c 1970-01-01 05:30:00.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/sched_idle.c 2008-10-24 19:56:20.000000000 +0530
@@ -0,0 +1,68 @@
+/*
+ * idle-task scheduling class.
+ */
+#include "type.h"
+#include "sched.h"
+
+struct proc *rdy_head_idle;  /* the idle queue */
+struct proc *rdy_tail_idle;  /* the idle queue */
+
+PRIVATE _PROTOTYPE( void enqueue_idle, (struct proc *rp));
+PRIVATE _PROTOTYPE( void dequeue_idle, (struct proc *rp));
+PRIVATE _PROTOTYPE( struct proc *pick_idle, (void));
+
+PRIVATE void enqueue_idle(struct proc *rp)
+{
+  if (rdy_head_idle == NIL_PROC) { /* add to empty queue */
+    rdy_head_idle = rdy_tail_idle = rp; /* create a new queue */
+    rp->p_nextready = NIL_PROC; /* mark new end */
+  }
+}
+
+PRIVATE void dequeue_idle(rp)
+register struct proc *rp; /* this process is no longer runnable */
+{
+/* A process must be removed from the scheduling queues, for example, because
+ * it has blocked.  If the currently active process is removed, a new process
+ * is picked to run by calling pick_proc().
+ */
+  register int q = rp->p_priority; /* queue to use */
+  register struct proc **xpp; /* iterate over queue */
+  register struct proc *prev_xp;
+
+  /* Now make sure that the process is not in its ready queue. Remove the
+   * process if it is found. A process can be made unready even if it is not
+   * running by being sent a signal that kills it.
+   */
+  prev_xp = NIL_PROC;
+  for (xpp = &rdy_head_idle; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {
+
+      if (*xpp == rp) { /* found process to remove */
+          *xpp = (*xpp)->p_nextready; /* replace with next chain */
+          if (rp == rdy_tail_idle) /* queue tail removed */
+              rdy_tail_idle = prev_xp; /* set new tail */
+          break;
+      }
+      prev_xp = *xpp; /* save previous in chain */
+  }
+}
+
+PRIVATE struct proc *pick_idle()
+{
+  struct proc *rp;
+  if ( (rp = rdy_head_idle) != NIL_PROC)
+    return rp;
+
+  panic("no ready process", NO_NUM);
+}
```

```
+
+
+/*
+ * Simple, special scheduling class for the idle task:
+ */
+static const struct sched_class idle_sched_class =
+{ NULL,
+  enqueue_idle,
+  dequeue_idle,
+  pick_idle
+};
diff -uNrB /mnt/hda4p3/src/kernel/sched_rt.c /mnt/hda4p3/src_copy/kernel/sched_rt.c
--- /mnt/hda4p3/src/kernel/sched_rt.c 1970-01-01 05:30:00.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/sched_rt.c 2008-10-25 20:08:27.000000000 +0530
@@ -0,0 +1,145 @@
+/*
+ * Real-Time Scheduling Class (mapped to the SCHED_FIFO and SCHED_RR
+ * policies)
+ */
+#include "type.h"
+#include "sched.h"
+
+#define MAX_RT_PRIO 15
+
+struct proc *rdy_head_rt[MAX_RT_PRIO+1]; /* the real time queues */
+struct proc *rdy_tail_rt[MAX_RT_PRIO+1]; /* the real time queues */
+
+PRIVATE _PROTOTYPE( void enqueue_rt, (struct proc *rp));
+PRIVATE _PROTOTYPE( void dequeue_rt, (struct proc *rp));
+PRIVATE _PROTOTYPE( void sched_rt, (struct proc *rp, int *queue, int *front));
+PRIVATE _PROTOTYPE( struct proc *pick_rt, (void));
+
+/*===========================================================================*
+ *                                sched_rt                                   *
+ *===========================================================================*/
+
+PRIVATE void sched_rt(rp, queue, front)
+register struct proc *rp; /* process to be scheduled */
+int *queue; /* return: queue to use */
+int *front; /* return: front or back */
+{
+/* This function determines the scheduling policy.  It is called whenever a
+ * process must be added to one of the scheduling queues to decide where to
+ * insert it.  As a side-effect the process' priority may be updated.
+ */
+  int time_left = (rp->p_ticks_left > 0); /* quantum fully consumed */
+
+  /* Check whether the process has time left. Otherwise give a new quantum
+   * and lower the process' priority, unless the process already is in the
+   * lowest queue.
+   */
+  if (! time_left) { /* quantum consumed ? */
+      rp->p_ticks_left = rp->p_quantum_size;  /* give new quantum */
+      if (rp->p_priority < MAX_RT_PRIO) {
+          rp->p_priority += 1; /* lower priority */
+      }
+  }
+
+  /* If there is time left, the process is added to the front of its queue,
+   * so that it can immediately run. The queue to use simply is always the
+   * process' current priority.
+   */
+  *queue = rp->p_priority;
+  *front = time_left;
+}
+
+/*===========================================================================*
+ *                                enqueue_rt                                 *
+ *===========================================================================*/
```

```
+PRIVATE void enqueue_rt(rp)
+register struct proc *rp; /* this process is now runnable */
+{
+/* Add 'rp' to one of the queues of runnable processes.  This function is
+ * responsible for inserting a process into one of the scheduling queues.
+ * The mechanism is implemented here.   The actual scheduling policy is
+ * defined in sched() and pick_proc().
+ */
+  int q;   /* scheduling queue to use */
+  int front; /* add to front or back */
+
+  /* Determine where to insert to process. */
+  sched_rt(rp, &q, &front);
+
+  /* Now add the process to the queue. */
+  if (rdy_head_rt[q] == NIL_PROC) { /* add to empty queue */
+      rdy_head_rt[q] = rdy_tail_rt[q] = rp; /* create a new queue */
+      rp->p_nextready = NIL_PROC; /* mark new end */
+  }
+  else if (front) { /* add to head of queue */
+      rp->p_nextready = rdy_head_rt[q]; /* chain head of queue */
+      rdy_head_rt[q] = rp; /* set new queue head */
+  }
+  else { /* add to tail of queue */
+      rdy_tail_rt[q]->p_nextready = rp; /* chain tail of queue */
+      rdy_tail_rt[q] = rp; /* set new queue tail */
+      rp->p_nextready = NIL_PROC; /* mark new end */
+  }
+}
+
+/*===========================================================================*
+ *                                dequeue_rt                                 *
+ *===========================================================================*/
+PRIVATE void dequeue_rt(rp)
+register struct proc *rp; /* this process is no longer runnable */
+{
+/* A process must be removed from the scheduling queues, for example, because
+ * it has blocked.  If the currently active process is removed, a new process
+ * is picked to run by calling pick_proc().
+ */
+  register int q = rp->p_priority; /* queue to use */
+  register struct proc **xpp; /* iterate over queue */
+  register struct proc *prev_xp;
+
+  /* Now make sure that the process is not in its ready queue. Remove the
+   * process if it is found. A process can be made unready even if it is not
+   * running by being sent a signal that kills it.
+   */
+  prev_xp = NIL_PROC;
+  for (xpp = &rdy_head_rt[q]; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {
+
+      if (*xpp == rp) { /* found process to remove */
+          *xpp = (*xpp)->p_nextready; /* replace with next chain */
+          if (rp == rdy_tail_rt[q]) /* queue tail removed */
+              rdy_tail_rt[q] = prev_xp; /* set new tail */
+          break;
+      }
+      prev_xp = *xpp; /* save previous in chain */
+  }
+}
+
+/*===========================================================================*
+ *                                pick_rt                                    *
+ *===========================================================================*/
+PRIVATE struct proc *pick_rt()
+{
+/* Decide who to run now.  A new process is selected by setting 'next_ptr'.
+ * When a billable process is selected, record it in 'bill_ptr', so that the
+ * clock task can tell who to bill for system time.
```

```
+ */
+  register struct proc *rp; /* process to run */
+  int q; /* iterate over queues */
+
+  /* Check each of the scheduling queues for ready processes. The number of
+   * queues is defined in proc.h, and priorities are set in the task table.
+   * The lowest queue contains IDLE, which is always ready.
+   */
+  for (q=0; q < MAX_RT_PRIO; q++) {
+      if ( (rp = rdy_head_rt[q]) != NIL_PROC) {
+          return rp;
+      }
+  }
+  return NIL_PROC;
+}
+
+static const struct sched_class rt_sched_class = {
+ &idle_sched_class,
+ enqueue_rt,
+ dequeue_rt,
+ pick_rt,
+};
diff -uNrB /mnt/hda4p3/src/kernel/table.c /mnt/hda4p3/src_copy/kernel/table.c
--- /mnt/hda4p3/src/kernel/table.c 2007-03-22 21:45:33.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/table.c 2008-10-25 20:13:22.000000000 +0530
@@ -32,6 +33,8 @@
 #include "proc.h"
 #include "ipc.h"
 #include <minix/com.h>
+#include "sched.h"
+#include "sched_classes.h"

 /* Define stack sizes for the kernel tasks included in the system image. */
 #define NO_STACK 0
@@ -109,20 +112,20 @@
 #define no_c { 0 }, 0

 PUBLIC struct boot_image image[] = {
-/* process nr, pc,flags, qs,  queue, stack, traps, ipcto, call,  name */
-{IDLE,  idle_task,IDL_F,  8, IDLE_Q, IDL_S,     0,     0, no_c,"idle"  },
-{CLOCK,clock_task,TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0, no_c,"clock" },
-{SYSTEM, sys_task,TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0, no_c,"system"},
-{HARDWARE,      0,TSK_F,  8, TASK_Q, HRD_S,     0,     0, no_c,"kernel"},
-{PM_PROC_NR,    0,SRV_F, 32,      3, 0,     SRV_T, SRV_M, c(pm_c),"pm"    },
-{FS_PROC_NR,    0,SRV_F, 32,      4, 0,     SRV_T, SRV_M, c(fs_c),"vfs"   },
-{RS_PROC_NR,    0,SRV_F,  4,      3, 0,     SRV_T, SYS_M, c(rs_c),"rs"    },
-{DS_PROC_NR,    0,SRV_F,  4,      3, 0,     SRV_T, SYS_M, c(ds_c),"ds"    },
-{TTY_PROC_NR,   0,SRV_F,  4,      1, 0,     SRV_T, SYS_M,c(tty_c),"tty"   },
-{MEM_PROC_NR,   0,SRV_F,  4,      2, 0,     SRV_T, SYS_M,c(mem_c),"memory"},
-{LOG_PROC_NR,   0,SRV_F,  4,      2, 0,     SRV_T, SYS_M,c(drv_c),"log"   },
-{MFS_PROC_NR,   0,SRV_F, 32,      4, 0,     SRV_T, SRV_M, c(fs_c),"mfs"   },
-{INIT_PROC_NR,  0,USR_F,  8, USER_Q, 0,     USR_T, USR_M, no_c,"init"  },
+/* process nr, pc,flags, qs, class, policy, queue, stack, traps, ipcto, call,  name */
+{IDLE,  idle_task,IDL_F,  8, &idle_sched_class, SCHED_IDLE, IDLE_Q, IDL_S,
+  0,     0, no_c,"idle"  },
+{CLOCK,clock_task,TSK_F,  8,   &rt_sched_class,   SCHED_RR, TASK_Q, TSK_S,
+  TSK_T,     0, no_c,"clock" },
+{SYSTEM, sys_task,TSK_F,  8,   &rt_sched_class,   SCHED_RR, TASK_Q, TSK_S,
+  TSK_T,     0, no_c,"system"},
+{HARDWARE,      0,TSK_F,  8,   &rt_sched_class,   SCHED_RR, TASK_Q, HRD_S,
+      0,     0, no_c,"kernel"},
+{PM_PROC_NR,    0,SRV_F, 32,   &rt_sched_class,   SCHED_RR,      3,     0,
+  SRV_T, SRV_M, c(pm_c),"pm"    },
+{FS_PROC_NR,    0,SRV_F, 32,   &rt_sched_class,   SCHED_RR,      4,     0,
+  SRV_T, SRV_M, c(fs_c),"vfs"   },
+{RS_PROC_NR,    0,SRV_F,  4,   &rt_sched_class,   SCHED_RR,      3,     0,
+  SRV_T, SYS_M, c(rs_c),"rs"    },
+{DS_PROC_NR,    0,SRV_F,  4,   &rt_sched_class,   SCHED_RR,      3,     0,
+  SRV_T, SYS_M, c(ds_c),"ds"    },
```

```
+{TTY_PROC_NR,   0,SRV_F,  4,   &rt_sched_class,   SCHED_RR,      1,     0,
  SRV_T, SYS_M,c(tty_c),"tty"   },
+{MEM_PROC_NR,   0,SRV_F,  4,   &rt_sched_class,   SCHED_RR,      2,     0,
  SRV_T, SYS_M,c(mem_c),"memory"},
+{LOG_PROC_NR,   0,SRV_F,  4,   &rt_sched_class,   SCHED_RR,      2,     0,
  SRV_T, SYS_M,c(drv_c),"log"   },
+{MFS_PROC_NR,   0,SRV_F, 32,   &rt_sched_class,   SCHED_RR,      4,     0,
  SRV_T, SRV_M, c(fs_c),"mfs"   },
+{INIT_PROC_NR,  0,USR_F,  8,   &rt_sched_class,   SCHED_RR, USER_Q,     0,
  USR_T, USR_M, no_c,"init"  },
 };

 /* Verify the size of the system image table at compile time. Also verify that
diff -uNrB /mnt/hda4p3/src/kernel/type.h /mnt/hda4p3/src_copy/kernel/type.h
--- /mnt/hda4p3/src/kernel/type.h 2007-02-16 21:24:28.000000000 +0530
+++ /mnt/hda4p3/src_copy/kernel/type.h 2008-10-25 01:31:18.000000000 +0530
@@ -17,6 +17,8 @@
   task_t *initial_pc; /* start function for tasks */
   int flags; /* process flags */
   unsigned char quantum; /* quantum (tick count) */
+  struct sched_class *sched_class;
+  int sched_policy;
   int priority; /* scheduling priority */
   int stksize; /* stack size for tasks */
   short trap_mask; /* allowed system call traps */
@@ -57,4 +59,8 @@

 typedef int (*irq_handler_t)(struct irq_hook *);

+#ifndef NIL_PROC
+#define NIL_PROC ((struct proc *) 0)
+#endif /* NIL_PROC */
+
 #endif /* TYPE_H */
```

Appendix B

VTRR Scheduler Patch

Following is the patch to MINIX 3 version 3.1.3a which implements the VTRR scheduler:

```
--- src/kernel/clock.c 2008-10-28 18:48:01.000000000 +0530
+++ src_v1/kernel/clock.c 2009-05-06 11:47:31.000000000 +0530
@@ -107,9 +107,9 @@
    */
   if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEMPTIBLE) {
       if(prev_ptr->p_rts_flags == 0) { /* if it was runnable .. */
-lock_dequeue(prev_ptr); /* take it off the queues */
-       lock_enqueue(prev_ptr); /* and reinsert it again */
+ scheduler_tick(prev_ptr);
       } else {
+ scheduler_tick(prev_ptr);
  kprintf("CLOCK: %d not runnable; flags: %x\n",
  prev_ptr->p_endpoint, prev_ptr->p_rts_flags);
       }
diff -uNrb src/kernel/main.c src_v1/kernel/main.c
--- src/kernel/main.c 2008-10-28 18:48:01.000000000 +0530
+++ src_v1/kernel/main.c 2009-05-06 16:04:57.000000000 +0530
@@ -73,6 +73,9 @@
  ip->endpoint = rp->p_endpoint; /* ipc endpoint */
  rp->p_max_priority = ip->priority; /* max scheduling priority */
  rp->p_priority = ip->priority; /* current priority */
+ rp->p_actual_priority = ACTUAL_PRIO(rp->p_priority);
+ rp->p_time_counter = rp->p_actual_priority;
+ rp->p_vft = 0;
  rp->p_quantum_size = ip->quantum; /* quantum size in ticks */
  rp->p_ticks_left = ip->quantum; /* current credit */
  strncpy(rp->p_name, ip->proc_name, P_NAME_LEN); /* set process name */
@@ -168,6 +171,9 @@
    * Return to the assembly code to start running the current process.
    */
   bill_ptr = proc_addr(IDLE); /* it has to point somewhere */
+  qvt = 0;
+  scheduler_cycle = 0;
+  reset_scheduler();
   announce(); /* print MINIX startup banner */
   restart();
```

```
 }
diff -uNrb src/kernel/proc.c src_v1/kernel/proc.c
--- src/kernel/proc.c 2008-10-28 18:48:00.000000000 +0530
+++ src_v1/kernel/proc.c 2009-05-07 16:30:46.000000000 +0530
@@ -499,12 +499,14 @@
  */
   int q;   /* scheduling queue to use */
   int front; /* add to front or back */
+  char p_time_counter;

 #if DEBUG_SCHED_CHECK
   check_runqueues("enqueue1");
   if (rp->p_ready) kprintf("enqueue() already ready process\n");
 #endif

+  rp->p_vft = MAX(qvt + (rp->p_quantum_size/rp->p_actual_priority), rp->p_vft);
   /* Determine where to insert to process. */
   sched(rp, &q, &front);

@@ -527,6 +529,25 @@
    * process yet or current process isn't ready any more, or
    * it's PREEMPTIBLE.
    */
+  for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
+    if (! isemptyp(rp) && rp->p_rts_flags == 0 && rp != proc_addr(IDLE))
+      total_time_counter += rp->p_time_counter;
+  }
+  if (total_share <= 0) {
+    p_time_counter = rp->p_actual_priority;
+  }
+  else
+    p_time_counter = (rp->p_actual_priority*total_time_counter)/total_share;
+
+  if (rp->p_sched_cycle == scheduler_cycle)
+    rp->p_time_counter = MIN (p_time_counter, rp->p_time_counter);
+  else
+    rp->p_time_counter = p_time_counter;
+
+  rp->p_sched_cycle = scheduler_cycle;
+
+  total_share += rp->p_actual_priority;
+
   if(!proc_ptr || proc_ptr->p_rts_flags ||
     (priv(proc_ptr)->s_flags & PREEMPTIBLE)) {
      pick_proc();
@@ -581,6 +602,9 @@
       prev_xp = *xpp; /* save previous in chain */
   }

+  total_share -= rp->p_actual_priority;
+  rp->p_sched_cycle = scheduler_cycle;
+
 #if DEBUG_SCHED_CHECK
   rp->p_ready = 0;
   check_runqueues("dequeue2");
@@ -601,17 +625,6 @@
  */
   int time_left = (rp->p_ticks_left > 0); /* quantum fully consumed */

-  /* Check whether the process has time left. Otherwise give a new quantum
-   * and lower the process' priority, unless the process already is in the
-   * lowest queue.
-   */
-  if (! time_left) { /* quantum consumed ? */
-      rp->p_ticks_left = rp->p_quantum_size;  /* give new quantum */
-      if (rp->p_priority < (IDLE_Q-1)) {
-          rp->p_priority += 1; /* lower priority */
-      }
-  }
```

```
-
   /* If there is time left, the process is added to the front of its queue,
    * so that it can immediately run. The queue to use simply is always the
    * process' current priority.
@@ -632,18 +645,56 @@
   register struct proc *rp; /* process to run */
   int q; /* iterate over queues */

-  /* Check each of the scheduling queues for ready processes. The number of
-   * queues is defined in proc.h, and priorities are set in the task table.
-   * The lowest queue contains IDLE, which is always ready.
-   */
+  if (prev_ptr && prev_ptr->p_nextready != NIL_PROC) {
+    rp = prev_ptr->p_nextready;
+  }
+  else {
+    if (! prev_ptr || prev_ptr->p_priority >= MIN_USER_Q)
+      q = 0;
+    else
+      q = prev_ptr->p_priority+1;
+
+    for (;q<NR_SCHED_QUEUES; q++) {
+      if ( rdy_head[q] != NIL_PROC) {
+ rp = rdy_head[q];
+ break;
+      }
+    }
+  }
+  if (!prev_ptr || rp->p_priority == IDLE_Q) {
+    goto find_first;
+  }
+
+  if (rp->p_time_counter > prev_ptr->p_time_counter) {
+    next_ptr = rp;
+    if (priv(rp)->s_flags & BILLABLE)
+      bill_ptr = rp; /* bill for system time */
+    return;
+  }
+  else if (prev_ptr->p_vft - qvt - qvt_stride <
+    (prev_ptr->p_quantum_size/prev_ptr->p_actual_priority)) {
+    kprintf ("vft inequality\n");
+    next_ptr = rp;
+    if (priv(rp)->s_flags & BILLABLE)
+      bill_ptr = rp; /* bill for system time */
+    return;
+  }
+  else {
+    goto find_first;
+  }
+
+ find_first:
   for (q=0; q < NR_SCHED_QUEUES; q++) {
       if ( (rp = rdy_head[q]) != NIL_PROC) {
           next_ptr = rp; /* run process 'rp' next */
           if (priv(rp)->s_flags & BILLABLE)
               bill_ptr = rp; /* bill for system time */
+      if (rp != proc_addr(IDLE) && rp->p_time_counter == 0)
+ reset_scheduler();
           return;
       }
   }
+
   panic("no ready process", NO_NUM);
 }

@@ -667,10 +718,7 @@
       if (! isemptyp(rp)) { /* check slot use */
    lock(5,"balance_queues");
    if (rp->p_priority > rp->p_max_priority) { /* update priority? */
```

```
-      if (rp->p_rts_flags == 0) dequeue(rp); /* take off queue */
        ticks_added += rp->p_quantum_size; /* do accounting */
-      rp->p_priority -= 1; /* raise priority */
-      if (rp->p_rts_flags == 0) enqueue(rp); /* put on queue */
    }
    else {
        ticks_added += rp->p_quantum_size - rp->p_ticks_left;
@@ -691,6 +739,56 @@
 }

 /*===========================================================================*
+ * reset_scheduler      *
+ *===========================================================================*/
+PUBLIC void reset_scheduler()
+{
+  register struct proc *rp;
+
+ /*kprintf ("Resetting scheduler(%d,%d)\n", total_share, total_time_counter);*/
+
+  total_share = 0;
+  qvt = 0;
+  for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
+    if (! isemptyp(rp) && rp->p_rts_flags == 0 && rp != proc_addr(IDLE)) {
+      lock(6, "reset_scheduler");
+      rp->p_time_counter = rp->p_actual_priority;
+      total_share += rp->p_actual_priority;
+    }
+    unlock(6);
+  }
+  if (total_share > 0)
+    qvt_stride = Q_SIZE/total_share;
+  else
+    qvt_stride = 0;
+
+  scheduler_cycle = (scheduler_cycle < MAX_SCHED_CYCLE) ?
+    scheduler_cycle++ : 0;
+
+  pick_proc();
+
+}
+
+/*===========================================================================*
+ * scheduler_tick      *
+ *===========================================================================*/
+PUBLIC void scheduler_tick(rp)
+register struct proc *rp;
+{
+  int time_left = (rp->p_ticks_left > 0); /* quantum fully consumed */
+
+  if (total_share > 0)
+    qvt += rp->p_quantum_size/total_share;
+
+  rp->p_time_counter--;
+
+  if (! time_left) { /* quantum consumed ? */
+      rp->p_ticks_left = rp->p_quantum_size;  /* give new quantum */
+  }
+  rp->p_vft += rp->p_quantum_size/rp->p_actual_priority;
+}
+
+/*===========================================================================*
   * lock_send      *
   *===========================================================================*/
 PUBLIC int lock_send(dst_e, m_ptr)
diff -uNrb src/kernel/proc.h src_v1/kernel/proc.h
--- src/kernel/proc.h 2008-10-28 18:48:01.000000000 +0530
+++ src_v1/kernel/proc.h 2009-05-07 11:28:32.000000000 +0530
@@ -22,10 +22,15 @@
   short p_misc_flags; /* flags that do not suspend the process */
```

```
   char p_priority; /* current scheduling priority */
+  char p_actual_priority;
+  char p_time_counter;
   char p_max_priority; /* maximum scheduling priority */
   char p_ticks_left; /* number of scheduling ticks left */
   char p_quantum_size; /* quantum size in ticks */

+  long int p_vft;
+  int p_sched_cycle;
+
   struct mem_map p_memmap[NR_LOCAL_SEGS];   /* memory map (T, D, S) */

   clock_t p_user_time; /* user time in ticks */
@@ -120,6 +125,10 @@
 #define MIN_USER_Q   14 /* minimum priority for user processes */
 #define IDLE_Q   15    /* lowest, only IDLE process goes here */

+#define MAX_SCHED_CYCLE 100
+#define Q_SIZE 8
+#define ACTUAL_PRIO(p) (IDLE_Q - p + 1)
+
 /* Magic process table addresses. */
 #define BEG_PROC_ADDR (&proc[0])
 #define BEG_USER_ADDR (&proc[NR_TASKS])
@@ -149,4 +158,9 @@
 EXTERN struct proc *rdy_head[NR_SCHED_QUEUES]; /* ptrs to ready list headers */
 EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES]; /* ptrs to ready list tails */

+long int qvt;
+long int qvt_stride;
+long int total_share;
+int scheduler_cycle;
 #endif /* PROC_H */
diff -uNrb src/kernel/proto.h src_v1/kernel/proto.h
--- src/kernel/proto.h 2008-10-28 18:48:01.000000000 +0530
+++ src_v1/kernel/proto.h 2009-05-06 01:28:39.000000000 +0530
@@ -33,6 +33,8 @@
 _PROTOTYPE( void lock_enqueue, (struct proc *rp) );
 _PROTOTYPE( void lock_dequeue, (struct proc *rp) );
 _PROTOTYPE( void balance_queues, (struct timer *tp) );
+_PROTOTYPE( void reset_scheduler, (void) );
+_PROTOTYPE( void scheduler_tick, (struct proc *rp) );
 #if DEBUG_ENABLE_IPC_WARNINGS
 _PROTOTYPE( int isokendpt_f, (char *file, int line, endpoint_t e, int *p, int f));
 #define isokendpt_d(e, p, f) isokendpt_f(__FILE__, __LINE__, (e), (p), (f))
diff -uNrb src/kernel/system/do_exec.c src_v1/kernel/system/do_exec.c
--- src/kernel/system/do_exec.c 2008-10-28 18:47:58.000000000 +0530
+++ src_v1/kernel/system/do_exec.c 2009-05-06 01:31:34.000000000 +0530
@@ -51,7 +51,8 @@
   } else {
    strncpy(rp->p_name, "<unset>", P_NAME_LEN);
   }
-
+  rp->p_time_counter = rp->p_actual_priority;
+  rp->p_vft = 0;
   return(OK);
 }
 #endif /* USE_EXEC */
diff -uNrb src/kernel/system/do_fork.c src_v1/kernel/system/do_fork.c
--- src/kernel/system/do_fork.c 2008-10-28 18:47:58.000000000 +0530
+++ src_v1/kernel/system/do_fork.c 2009-05-06 01:48:15.000000000 +0530
@@ -55,6 +55,9 @@
   rpc->p_user_time = 0; /* set all the accounting times to 0 */
   rpc->p_sys_time = 0;

+  rpc->p_time_counter = rpc->p_actual_priority;
+  rpc->p_vft = 0;
+
```

```
   /* Parent and child have to share the quantum that the forked process had,
    * so that queued processes do not have to wait longer because of the fork.
    * If the time left is odd, the child gets an extra tick.
diff -uNrb src/kernel/system/do_nice.c src_v1/kernel/system/do_nice.c
--- src/kernel/system/do_nice.c 2008-10-28 18:47:59.000000000 +0530
+++ src_v1/kernel/system/do_nice.c 2009-05-06 01:31:25.000000000 +0530
@@ -48,6 +48,7 @@
        */
       RTS_LOCK_SET(rp, NO_PRIORITY);
       rp->p_max_priority = rp->p_priority = new_q;
+      rp->p_actual_priority = ACTUAL_PRIO(new_q);
       RTS_LOCK_UNSET(rp, NO_PRIORITY);

       return(OK);
diff -uNrb src/kernel/table.c src_v1/kernel/table.c
--- src/kernel/table.c 2008-10-28 18:48:01.000000000 +0530
+++ src_v1/kernel/table.c 2009-05-06 04:26:06.000000000 +0530
@@ -110,19 +110,19 @@

 PUBLIC struct boot_image image[] = {
 /* process nr, pc,flags, qs,  queue, stack, traps, ipcto, call,  name */
-{IDLE,  idle_task,IDL_F,  8, IDLE_Q, IDL_S,     0,     0, no_c,"idle"  },
-{CLOCK,clock_task,TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0, no_c,"clock" },
-{SYSTEM, sys_task,TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0, no_c,"system"},
-{HARDWARE,      0,TSK_F,  8, TASK_Q, HRD_S,     0,     0, no_c,"kernel"},
-{PM_PROC_NR,    0,SRV_F, 32,      3, 0,     SRV_T, SRV_M, c(pm_c),"pm"    },
-{FS_PROC_NR,    0,SRV_F, 32,      4, 0,     SRV_T, SRV_M, c(fs_c),"vfs"   },
-{RS_PROC_NR,    0,SRV_F,  4,      3, 0,     SRV_T, SYS_M, c(rs_c),"rs"    },
-{DS_PROC_NR,    0,SRV_F,  4,      3, 0,     SRV_T, SYS_M, c(ds_c),"ds"    },
-{TTY_PROC_NR,   0,SRV_F,  4,      1, 0,     SRV_T, SYS_M,c(tty_c),"tty"   },
-{MEM_PROC_NR,   0,SRV_F,  4,      2, 0,     SRV_T, SYS_M,c(mem_c),"memory"},
-{LOG_PROC_NR,   0,SRV_F,  4,      2, 0,     SRV_T, SYS_M,c(drv_c),"log"   },
-{MFS_PROC_NR,   0,SRV_F, 32,      4, 0,     SRV_T, SRV_M, c(fs_c),"mfs"   },
-{INIT_PROC_NR,  0,USR_F,  8, USER_Q, 0,     USR_T, USR_M, no_c,"init"  },
+{IDLE,  idle_task,IDL_F,  Q_SIZE, IDLE_Q, IDL_S,     0,     0, no_c,"idle"  },
+{CLOCK,clock_task,TSK_F,  Q_SIZE, TASK_Q, TSK_S, TSK_T,     0, no_c,"clock" },
+{SYSTEM, sys_task,TSK_F,  Q_SIZE, TASK_Q, TSK_S, TSK_T,     0, no_c,"system"},
+{HARDWARE,      0,TSK_F,  Q_SIZE, TASK_Q, HRD_S,     0,     0, no_c,"kernel"},
+{PM_PROC_NR,    0,SRV_F,  Q_SIZE,      3, 0,     SRV_T, SRV_M, c(pm_c),"pm"  },
+{FS_PROC_NR,    0,SRV_F,  Q_SIZE,      4, 0,     SRV_T, SRV_M, c(fs_c),"vfs" },
+{RS_PROC_NR,    0,SRV_F,  Q_SIZE,      3, 0,     SRV_T, SYS_M, c(rs_c),"rs"  },
+{DS_PROC_NR,    0,SRV_F,  Q_SIZE,      3, 0,     SRV_T, SYS_M, c(ds_c),"ds"  },
+{TTY_PROC_NR,   0,SRV_F, Q_SIZE,      1, 0,     SRV_T, SYS_M,c(tty_c),"tty"  },
+{MEM_PROC_NR,   0,SRV_F,Q_SIZE,      2, 0,     SRV_T, SYS_M,c(mem_c),"memory"},
+{LOG_PROC_NR,   0,SRV_F,  Q_SIZE,      2, 0,     SRV_T, SYS_M,c(drv_c),"log" },
+{MFS_PROC_NR,   0,SRV_F, Q_SIZE,      4, 0,     SRV_T, SRV_M, c(fs_c),"mfs" },
+{INIT_PROC_NR,  0,USR_F,  Q_SIZE, USER_Q, 0,     USR_T, USR_M, no_c,"init"  },
 };

 /* Verify the size of the system image table at compile time. Also verify that
```

Bibliography

[BZ96] Jon Bennet and Hui Zhang. WF2 Q: Worst-case Fair Weighted Fair Queueing. In *Proceedings of INFOCOM '96*, pages 120–128, 1996.

[CCN+05] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoquiang Zheng. Group Ratio Round Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 337–352, 2005.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM '89*, pages 1–12, 1989.

[IEEE04] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6*. IEEE Std 1003.1, 2004 Edition. Available at http://www.opengroup.org/onlinepubs/009695399/.

[Kle76] Leonard Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. Wiley-Interscience, First Edition, 1976. pp. 156-188.

[Kol07] Con Kolivas. *RSDL completely fair starvation free interactive cpu scheduler*, 2007. Available at http://lwn.net/Articles/224654/

[Mol07] Ingo Molnar. *Modular Scheduler Core and Completely Fair Scheduler*, 2007. Available at http://lwn.net/Articles/230501/

[NVZ01] Jason Nieh, Chris Vaill, and Hua Zhong. Virtual Time Round Robin: An O(1) Propor- tional Share Scheduler. In *Proceedings of*

the 2001 USENIX Annual Technical Confer- ence, pages 245–259, 2001.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Second Edition, 2001.

[Tan06] Andrew S. Tanenbaum. *Operating System Design and Implementation*. Prentice Hall, Third Edition, 2006.