RTMINIX3

A MORE REAL-TIME MINIX 3

Author:
Bianco ZANDBERGEN

Supervisors:
R. TANGELDER
H. VAN LEEUWEN

October 6, 2009

Contents

1 Introduction

A real-time system is a system from which the correctness of the computational results depends on the time in which computations are finished. The computations have to be finished before a certain defined deadline. Small real-time embedded systems often don't use an operating system. When the complexity of the embedded system increases, resource management is getting more difficult. Complex real-time systems often use a real-time operating system (RTOS) to manage the resources.

Traditionally RTOS's are build from scratch with the requirements for real-time behaviour in mind. However it is also possible to turn a general purpose operating system into a real-time operating system. An example of this is GNU/Linux which has several real-time distributions (i.e MontaVista Linux [1]).

In this project we will focus on making Minix 3[1] more real-time. Minix 3 is an UNIX-like POSIX compatible general purpose operating system. Minix 3 is often critized for not being efficient due to the usage of a micro-kernel. The QNX RTOS [3] has proven that a micro-kernel can be used as the heart of a real-time operating system.

In this document we describe in detail the modifications we have made to Minix 3 in order to make it more real-time. We assume that the reader has basic knowledge about the Minix 3 OS, (real-time) operating systems and the C programming language.

[1]With Minix 3 we refer to Minix version 3.12a unless stated otherwise.

1.1 Overview of Minix 3[2]

Minix 3 is an Unix-like Operating System. The main goal of Minix 3 is providing a highly fault-tolerant operating system that can repair itself without user intervention in the sense that it can restart important parts of the operating system such as device drivers when they crash.

Minix 3 is based on a microkernel. This means that large parts of the Operating System are outside the kernel. The kernel itself is as small as possible. Reducing the size of the kernel reduces the risk of bugs in the kernel. The most important tasks of the Minix 3 kernel are process scheduling, inter-process communication (IPC) and facilitating a set of kernel calls that can be used by the parts of the Operating System that are implemented in user space. The kernel only implements tasks that cannot be done outside the kernel or are not efficient enough to be implemented in user space.

Minix 3 uses round-robin scheduling with multiple priorities (Multilevel Feedback Queue). Kernel tasks have the highest priorities, system servers and drivers have lower priorities and user processes have the lowest priorities. Processes that use their full quantum (calculation time) will be temporary lowered in priority. This will keep the system responsive in favour of I/O bound processes and will tame infinite loops.

Minix 3 uses message passing and piping as IPC (inter-process communication) mechanisms. Message passing is used between components of the operating system and between user processes and the operating system. Message passing uses fixed length messages. User processes are not allowed to communicate with eachother using messages. Instead they can use piping.

There are two important servers in the kernel: the clock task and the system task. The clock task in cooperation with the clock interrupt handler will keep the system time and deschedules processes that have used their full quantum. The system task offers a set of kernel calls used by system servers and drivers. The kernel restricts access to kernel calls for system servers and drivers. Drivers are implemented as user space servers. This ensures that a driver crash will not crash the system. Drivers do not have direct access to interrupts. Incoming interrupts are converted to messages that are send to the driver.

There are two user space servers which are the only servers user processes can communicate with. This means that a user process never communicates with the kernel directly. The first one is the file system server which takes care of file system related tasks. The second one is the process manager which takes care of processes and memory management. Minix 3 uses segmentation to protect memory. The process table is divided in three parts: the kernel, the file system and the process manager each have their own part of the process table containing only the information that is needed for their function. These tables are synchronized with each other.

[2]See Tanenbaum and Woodhull, 2006 [7] for more detailed information about Minix 3.

System servers and drivers that crash must be restarted automatically if it is possible to restart them. The reincarnation server observes them and restarts them if neccesary. Not all servers can be restarted though, some are vital and may not crash, otherwise the whole system will crash.

1.2 Other Projects

Our project is not the first with the goal of making Minix 3 or its predecessors real-time. We will briefly describe these other projects.

RT-MINIX

RT-MINIX is a project that adds real-time extensions to Minix 1 [2]. The code has also been ported to versions 1.5, 1.7.0 and 2.0.0. It has the following features:

- Rate-Monotonic scheduler.
- Earliest Deadline First scheduler.
- Real-time statistics collection.
- Timer resolution management.

MINIX4RT

MINIX4RT is a project based on Minix 2 [5]. It has the following features:

- Real-time sub-kernel, the Minix OS runs as a process above the sub-kernel.
- Real-time interrupt handling.
- Fixed priority real-time scheduling.
- Real-time IPC
- Real-time statistics collection.

Realtime Minix

Realtime Minix is based on Minix 3 [8]. It has the following features:

- Partly or fully API compatible with MINIX4RT.
- Separate system- and kernel calls for real-time processes.
- Virtual timers
- Real-time interrupt handling.
- Semaphores.

1.2.1 The need for this project

Our goal is to make Minix 3 more real-time, the first two projects are not based on Minix 3. The last project is based on Minix 3 but is not very well documented.

1.3 Goals

We had to narrow down our scope and focus on improvements that are feasible to implement within limited time. We have defined the following goals:

- Minix 3 as a soft real-time OS.
- Implement at least one real-time scheduler.
- Improve IPC performance for real-time processes.
- Implement semaphores.
- Properly document the system.

1.4 Development environment

Hardware
We have not used real hardware to develop our system.
Instead we have used virtualisation technology.
We used VMware Server 2.0 on Windows XP and VMware Fusion 2.0 on Mac OS X.

Software
Our system is based on Minix version 3.1.2a.
We have used Vim and Mined as text editors.
We used SSH and Telnet to remotely communicate with the system.
We used the standard FTP client and the Apache webserver for file transfers.
For installation of additional packages we used Packman, which is a simple software package manager.

2 Scheduling

In this section we will briefly discuss how scheduling is implemented in Minix 3. We have implemented two real-time schedulers which will be discussed in detail. Before we do that we will explain how kernel, system- and library calls are implemented in Minix 3 because knowledge about them is required to understand our schedulers.

2.1 Scheduling in Minix 3

2.1.1 Overview of scheduling in Minix 3

Minix 3 uses round-robin scheduling with multiple queues. Each queue represents a priority. Minix 3 has 16 scheduling queues (see figure 1). The highest priority processes are in queue 0 and the lowest in queue 15. Queues 0-6 are reserved for system processes. Queues 7-14 are reserved for user processes. The idle process is scheduled in queue 15 and has the lowest priority. If a new process has to be choosen to run next, it starts checking if there is a process in queue 0. If queue 0 is empty it will check queue 1 and so on. If there are no processes ready the idle process which has the lowest priority will run. Processes that were previously blocked and have to be unblocked will be placed in front of the queue if it has quantum (calculation time) left. This will give I/O bound processes a better response time. Processes have a maximum priority and start at this priority. Processes that use their full quantum will be lowered in priority but will always have a higher priority than the idle process. In this way CPU bound processes will not get in the way of I/O bound processes. If a process that is lowered in priority does not use its full quantum after it was lowered in priority, its priority can be increased untill it reaches its maximum priority. At determined intervals there will be checked if the priority of processes can be increased.

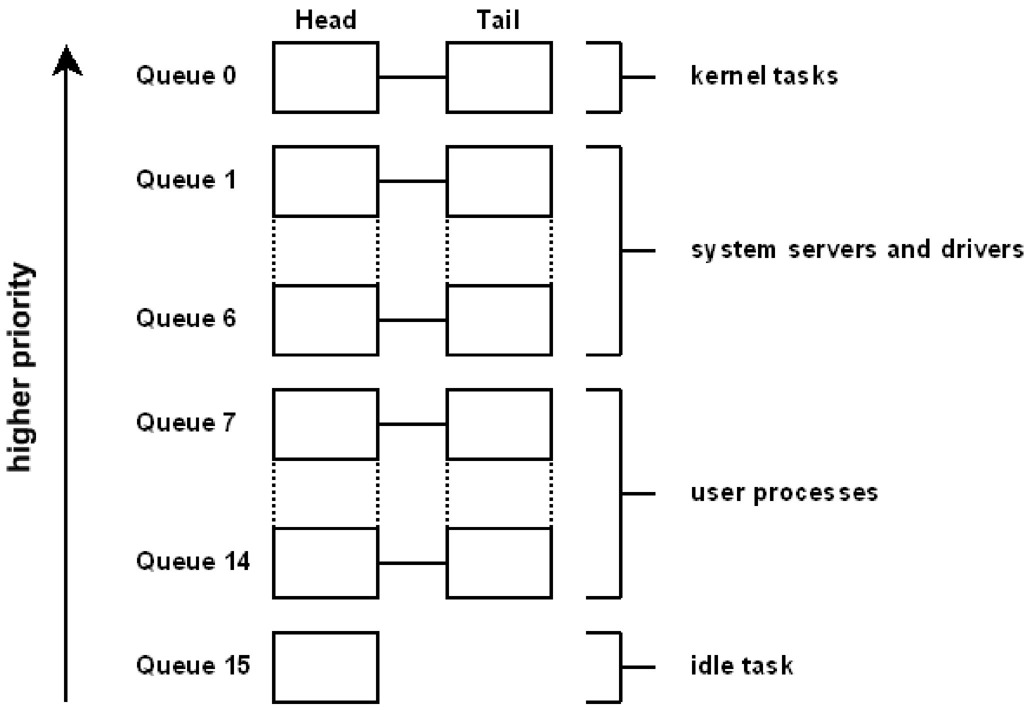

Figure 1: Scheduling queues in Minix 3.

2.1.2 Implementation of scheduling in Minix 3

In this section we discuss the implementation of scheduling at the source code level.

We start with the clock interrupt handler and clock task which are implemented in */usr/src/kernel/clock.c* . The clock timer is set to 60Hz. Each clock interrupt the *clock_handler()* function is called. First of all this function will update the system time. After that it will check if the current running process is preemptive. Kernel tasks (clock- and system task) are not preemptive. If the process is preemptive it will decrease the *p_ticks_left* field of the process entry which holds the quantum that the process has left. If the process is a system process, a user process will have to be billed for the system time and thus the *p_ticks_left* field of the process entry of the billable process will be decreased. Minix 3 also keeps count of the used system time by user processes by increasing the *p_sys_time* field of the billable user process. Then it will check if a timer has expired or if the running process has no quantum left. If this is the case it will send a notification to the clock task.

The clock task waits for incoming messages. If it receives a notification from the *clock_handler()* it will invoke *do_clocktick()* (despite the name this function is not called on every clock tick!). The *do_clocktick()* function will first check if the process running at the time of the clock interrupt has used its full quantum. If this is the case it will be removed from the scheduling queue by calling *lock_dequeue()* and rescheduled by calling *lock_enqueue()* . Furthermore it will check for and handle any expired timers.

Next we discuss the functions in */usr/src/kernel/proc.c* . The function *sched()* determines the scheduling policy. For a given process *sched()* returns to which queue it should be added and if it should be added in the front or back of the queue. If a process does not have quantum left it will get a new quantum and will be lowered in priority if possible. Processes that have quantum left will be added in the front of the queue and processes that do not, will be added to the back of the queue.

lock_dequeue() and *lock_enqueue()* are functions that call respectively *dequeue()* and *enqueue()* with interrupts disabled. The function *enqueue()* adds a process to the scheduling queues. It first calls *sched()* to know to which queue the process should be added and if it should be added to the front or back. After that it will insert the process in the right queue. Finally it will call *pick_proc()* to set the next running process. *dequeue()* removes a process from the scheduling queues. If the active process is removed a new process should be chosen and thus *pick_proc()* will be called.

pick_proc() simply iterates over all scheduling queues starting with the highest priority. The first process it finds should run next and is saved in *next_ptr* . If this process is billable for system time (all normal user processes are) *bill_ptr* will be set to this process. All system processes that run after this process will be billed to this process until a new billable process may run.

Processes that are not running at their maximum priority can be increased in priority. This is done by the *balance_queues()* function. It will iterate over the process table. For processes that are not running at their maximum priority first is checked if the process is scheduled. If it is scheduled it will be removed from the scheduling queues by calling *dequeue()* . The process will receive a new quantum and the priority is increased by decreasing the *p_priority* field in the process table entry (the lower the *p_priority* value the higher the priority). After that it will check if the process is runnable and if it is, it will be added to the scheduling queues by calling *enqueue()* . For processes that are running at their maximum priority the quantum will be renewed. After that it will have to set the period untill *balance_queues()* will be called again. This period is not static and depends on the number of ticks that are added to the processes. This number is compared to the value of Q_BALANCE_TICKS which is 100 by default. If the number of ticks added is larger than Q_BALANCE_TICKS this will be the number of ticks till the next run of *balance_queues()* . If the number of ticks added is smaller than Q_BALANCE_QUEUES the next run will be after Q_BALANCE_TICKS ticks.

2.1.3 How real-time is the Minix 3 scheduler?

If the Minix 3 scheduler already meets the demands of real-time applications there is no need for changing it. The most important demand is meeting deadlines. Unfortunately Minix 3 cannot guarantee that processes will meet their deadlines. Minix 3 isn't even aware of deadlines. Furthermore Minix 3 does not provide facilities for periodic processes. The conclusion is that the Minix 3 scheduler is not usable for real-time applications but it is possible to improve real-time behaviour of Minix 3.

2.2 Kernel, system- and library calls

Minix 3 makes a distinction between kernel and system calls. The reason for that is that large parts of the operating system runs in user space due to the micro-kernel architecture. These parts are implemented as seperate processes such as servers and drivers. System calls are handled by these servers and not the kernel. Kernel calls are (obviously) handled by the kernel. User processes are not allowed to make kernel calls. Instead they use system calls and if the system call needs service from the kernel, the server that implements the system call can do a kernel call. This is done for security reasons.

There are two servers that handle all system calls for user processes: The process manager and the file system. The process manager handles all process and memory related system calls and the file system all file related system calls.

To understand the implemented API calls for RTMINIX3 we first have to understand the call chain of a typical API call. The API call itself is implemented as a library function. This library function will do the system call. All system calls related to the real-time functionality needs service from the kernel. The system call will call a system library function that will do the kernel call. In the next sections we will discuss the implementation of kernel- system- and library calls.

2.2.1 Implementation of library functions

Library functions can be added in two ways: to an existing library or by creating a new library. The latter is a little bit more complex.

The POSIX library is a good candidate to learn how to add a function to an existing library. However the POSIX library should only contain functions described in the POSIX standard, so we don't recommend extending it for other than educational purposes. To add a function to this library create a new file in */usr/src/lib/posix/* with for example the following name: *my_lib_function.c*. Furthermore add this file to */usr/src/lib/posix/Makefile.in* under `libc\_FILES="..."`. *Makefile.in* is a configuration file to generate a Makefile. Because we have added a new file to *Makefile.in* we will have to regenerate the Makefile. To do this run "`make makefile`" in */usr/src/lib/posix/*. Now you can implement the library function in *my_lib_function.c*.

If one wants to create a new library for the library function create a new directory in */usr/src/lib/* with the library name: for example */usr/src/lib/example*. Modify */usr/src/lib/Makefile.in* and add the example directory under `SUBDIRS="..."`. After that regenerate the Makefile file by running "`make makefile`" in */usr/src/lib/*. Create two new files: */usr/src/lib/example/Makefile.in* and */usr/src/lib/example/my_lib_function.c*. Modify *Makefile.in* and add the code in listing 1. Generate the Makefile file by running "`make makefile`" in */usr/src/lib/example/*. Now you can implement the library function in *my_lib_function.c*.

Listing 1: Makefile.in file for a new library.

```
# Makefile for lib/example (libexample)
CFLAGS="-O -D_MINIX -D_POSIX_SOURCE"
LIBRARIES=libexample
libexample_FILES="my_lib_function.c"
TYPE=both
```

After writing the library function we will have to recompile the libraries. To do this run ”`make libraries`” in */usr/src/tools/*. This will recompile and install all libraries. We still need to define a prototype. For functions in the POSIX library this is done in */usr/src/include/unistd.h*. You can also create a new header file in */usr/src/include/*. Minix keeps two copies of the include files: */usr/include/* and */usr/src/include/*. Always modify the files in */usr/src/include/* and not in */usr/include/*. After modifying or adding files run ”`make includes`” in */usr/src/tools/*. This copies the files from */usr/src/include/* to */usr/include/* to keep them synchronized.

The library function is now ready to use. If the library function is not added to the POSIX library, a user program must be explicitly linked with the library using the compilers `-l` flag. For the example library it would be `-lexample`.

A typical library function used for doing a system call will create a new message. It will set the message fields and after that it will call *_syscall()* with the endpoint of the server that handles the system call, the system call number and a pointer to the message as arguments. Endpoints are a way to uniquely identify processes (see section 3.1 on page 58 for a more detailed description of endpoints). *_syscall()* will send the message and wait for an reply by calling *_sendrec()* with the specified endpoint and message pointer as arguments. The *m_type* field of the message is used for the return value of the system call. If this value is negative *_syscall()* will set errno to the absolute value of *m_type* and return -1. If the value is positive it will simply return the value. See listing 2 for an example library function doing a system call.

Listing 2: Implementation of setpriority() library function.

```
int setpriority(int which, int who, int prio)
{
        message m;

        m.m1_i1 = which;
        m.m1_i2 = who;
        m.m1_i3 = prio;

        return _syscall(MM, SETPRIORITY, &m);
}
```

2.2.2 Implementation of system calls

Both the process manager and file system can handle system calls for user processes. We only discuss the implementation in the process manager here. The process manager calls *get_work()* to receive a new message which will be saved in a global variable. This function not only receives the message but also sets some variables after receiving a message such as a pointer to the process entry in the process manager. It then analyzes the *m_type* field of the message. This field contains the system call number. System call numbers are defined in */usr/src/minix/callnr.h.* The process manager has an array with function pointers called *call_vec[]* . The system call number is mappable to this array for fast lookup. The process manager will invoke the function with the system call number as indice in the array. The invoked function does not take any parameters but uses global variables such as the received message. The return value of the invoked function will be send back to the calling process.

We will discuss adding a system call to the process manager but implementation for both the file system and process manager are the same. First we add a new prototype in */usr/src/servers/pm/proto.h,* for example:

```
_PROTOTYPE(int do_mycall, (void));
```

Minix is using a macro for creating prototypes. After that modify */usr/src/servers/pm/table.c* and find an unused slot in *call_vec[],* for example:

```
no_sys,    /* 69 = unused */
```

Replace *no_sys* with *do_mycall* and adjust the comment accordingly. Add the new system call in for example */usr/src/servers/pm/misc.c.* It is also possible to add the system call in a new file. You will have to edit the Makefile file if you do that. Modify */usr/src/include/minix/callnr.h.* Add a new macro for the system call number. For example:

```
#define MY_CALL 69
```

Run ”`make includes`” in */usr/src/tools/* after modifying *callnr.h.* We will have to recompile the kernel. The simplest way is to recompile the whole system by running ”`make fresh install`” in */usr/src/tools/.* After recompiling reboot the computer. You can test the system call after rebooting by creating a simple program creating a new message and invoking *_syscall().* This will omit the use of a library function.

System calls that needs service from the kernel invoke a system library function. All kernel calls are wrapped in system library functions. The system library is implemented in */usr/src/lib/syslib/.* These functions are prefixed with 'sys_'. These system library functions create a new message and sets the message fields. After that they call *_taskcall()* with the endpoint of the task (always the system task), the system call number and a pointer to the message as arguments. *_taskcall()* will use *_sendrec()* to send and receive a reply from the system task. *_taskcall()* returns the result of the kernel call.

2.2.3 Implementation of kernel calls

All kernel calls are handled by the system task. The implementation of the system task is similar to the process manager. The system task will call *receive()* to receive a message. It then analyzes the *m_type* field to get the kernel call number. The system task has also an array with function pointers called *call_vec[]*. The system task will invoke the function with the kernel call number as index in the array. After that it will try to send a reply back with the result of the kernel call.

To add a kernel call we first define the kernel call number in */usr/src/include/minix/com.h*. All kernel call numbers start with a ' SYS_ ' prefix. The kernel call number consists of a constant base number and the number of the kernel call. Add a new macro with with a number higher than the current highest number. After that increase the total number of kernel calls defined in NR_SYS_CALLS with one (Please note that the naming convention is not always consistent. The term system call is sometimes used while it should be kernel call). After that create a new prototype in */usr/src/kernel/system.h* with a message pointer as parameters. We now have to map the kernel call to a place in the function pointer array of the system task. This is done in */usr/src/kernel/system.c* in the *initialize()* function. This mapping is done using the MAP() macro, taking the system call number and the function name as parameters. Create a new file for the kernel call in */usr/src/kernel/system/*. Modify the Makefile file in */usr/src/kernel/system/* to include the added kernel call. You can now implement the kernel call.

After creating the kernel call you will have to recompile the kernel and reboot in order to use it. To recompile the system (and not only the kernel) run "`make fresh install`" in */usr/src/tools/*.

2.3 Real-time scheduling algorithms

General purpose schedulers aim for a high throughput and fast response for foreground processes. Real-time processes have demands that general purpose schedulers cannot meet such as meeting deadlines and support for periodic processes.

Choosing a scheduler for Real-time Minix 3
We have decided to implement the most common used static priority and dynamic priority scheduling algorithms. They are respectively Rate-Monotonic scheduling and Earliest Deadline First scheduling. In both cases we let the standard Minix 3 scheduler cooperate with the real-time scheduler. System processes and non real-time processes are scheduled using the standard scheduler.

Rate-Monotonic
Rate-Monotonic (RM) uses static priorities for processes. These priorities are assigned based on the period time. The smaller the period time, the higher the priority. RM is an optimal scheduling algorithm for static priorities. This means that if a set of tasks can be scheduled using any static priority scheduling algorithm, RM can schedule the set of tasks and meet their deadlines. RM has a worst case utilization bound of *ln 2* which is approximately 69.3%. If the system is overloaded it is predictable which processes will miss their deadlines as processes with low priorities will miss their deadlines first.

Earliest Deadline First
The priority of processes scheduled using Earliest Deadline First (EDF) is based on their deadline. The closer the process is to its deadline the higher the priority. The process closest to its deadline should run. EDF is an optimal scheduling algorithm for processes that are characterized by an arrival time, a calculation time and a deadline. This means that if this set of tasks can be scheduled by any real-time scheduling algorithm, EDF can schedule this set of tasks and meet their deadlines. If the deadline equals the period EDF has a utilization bound of 100%. If the system is overloaded it is unpredictable which processes will miss their deadlines. This is a major disadvantage for using EDF in critical systems.

2.4 Scheduler independent parts

In this section we will discuss modifications made to the scheduler that are used by all scheduling algorithms. We will also discuss debug and test functions.

2.4.1 Scheduling bridge

One scheduling queue is reserved for real-time processes. We must prevent that normal user processes can be added to this queue. The scheduling queues are one to one mappable to the minix priorities. If we prevent that a process gets the priority reserved for real-time processes, we prevent them from getting scheduled in the real-time queue. Processes with a priority higher than real-time processes can be lowered in priority and eventually get the priority reserved for real-time processes. On the other side processes that were lowered in priority can be increased in priority and eventually get the priority reserved by real-time processes.

We have decided that processes with a priority higher than real-time processes should never be lowered in priority to an priority lower than or equal to real-time processes. The lowest priority possible for these processes is the first priority higher than then real-time priority (figure 2 image a).

After implementing this we noticed that the system can't shutdown properly. Unfortunately we have not found the cause and as the shutdown process is a very complex process a workaround was the best option. We have added an API call to enable a "scheduling bridge". This will be called by the shutdown process so that it can properly shutdown. The scheduling bridge makes it possible for high priority processes to get low priorities. We still prevent them from getting the priority reserved for real-time processes. If the process is lowered in priority and the new priority is the priority reserved for real-time processes it will get the first priority lower than the real-time priority (figure 2 image b). The bridge works in both directions. A process that was lowered in priority and would get the real-time priority will get the first priority higher than the real-time priority.

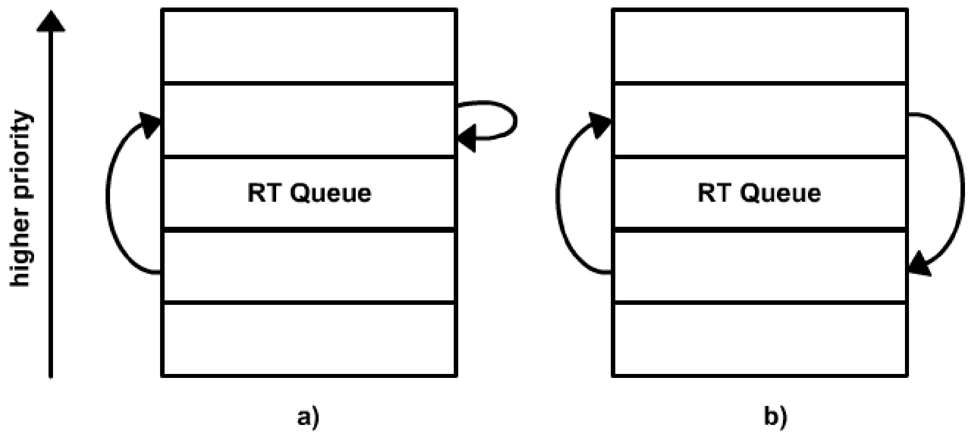

Figure 2: Scheduling bridge. Situation in image a does not allow high priority processes to get a lower priority than real-time processes, situation in image b does.

Implementation

The state of the scheduling bridge is saved in a global kernel variable called *rt_sched_bridge*. If the value of this variable is 0, high priority processes cannot get a priority lower than real-time processes. If the value is 1 they can. We will discuss the implementation of the API call controlling this variable first. After that we will discuss the scheduling bridge itself.

The API call is implemented as the following library function: *rt_set_sched_bridge(state)*. This library function is implemented in */usr/src/lib/rtminix3/rt_set_sched_bridge.c*. The state parameter can be 0 or 1. In practice only the value of 1 is used as the *rt_sched_bridge* kernel variable is 0 by default.

The library function creates a new message and sets the state as one of the fields in the message. After that it calls *_syscall()* to make a system call with the process manager as destination and RT_SET_SCHED_BRIDGE as system call number.

The RT_SET_SCHED_BRIDGE system call handled by the process manager is implemented in */usr/src/servers/pm/rt.c* in the function *do_rt_set_sched_bridge()*. The system call acts as a proxy for the user process to communicate with the kernel. The only thing it does is calling *sys_rt_sched_bridge()* with the bridge state specified in the message as parameter. *sys_rt_sched_bridge()* is a library function that will do the SYS_RT_SCHED_BRIDGE kernel call to set the *rt_sched_bridge* variable in the kernel. *sys_rt_sched_bridge()* is implemented in */usr/src/lib/syslib/sys_rt_sched_bridge.c*.

The SYS_RT_SCHED_BRIDGE kernel call is handled by the system task (as all kernel calls) and is implemented in */usr/src/kernel/system/do_rt_set_sched_bridge.c* in the function *do_rt_set_sched_bridge()*. This function simply sets the *rt_sched_bridge* value to the value specified in the received message, no checks are done. Although the name is the same as the function in the process manager, this is not the same function!

Now we have discussed how to set the bridge state we can focus on the actual implementation of the bridge. Two parts of the scheduler should be modified: the part that decreases priority of processes when they use their full quantum and the part that increases priority of processes that where previous lowered in priority.

Processes that used their full quantum will be lowered in priority by the *sched()* function of the scheduler implemented in */usr/src/kernel/proc.c*. This function decides for a process in which scheduling queue it should be added and if it should be added to the front or back of the queue. The code for possible decreasing of the priority will only be executed for non real-time processes. After decreasing the priority it will check if the priority is equal to the priority reserved for real-time processes (RT_Q). If this is the case it will check the bridge state. If the bridge state is 0 it will set the priority of the process to the first priority higher than the real-time priority. In practice this means that the process will keep the same priority as it had before decreasing it. If the bridge state is set to 1 (or any other positive number) it will set the priority of the process to the priority defined in USER_Q which is the first priority lower than the real-time priority.

The priority of processes that do not run at their maximum priority can be increased by the *balance_queues()* function implemented in */usr/src/kernel/proc.c*. This function loops through the whole process table. The priority of processes that do not have their maximum priority will be increased. After increasing the priority it will check if the priority equals to the priority reserved for real-time processes (RT_Q). If this is the case we check if the process is allowed to get the first priority higher than the real-time priority, and if it is it will get that priority.

2.4.2 Nice kernel call

The nice kernel call is used to change the priority of processes. It is implemented in */usr/src/kernel/system/do_nice.c*. This kernel call makes it possible for non real-time processes to get the real-time priority and entering the queue reserved for real-time processes. Real-time processes on the other hand can change their priority. Both cases should be avoided and we have modified the kernel call to do so. If any of these cases happens, the kernel call will return an error code.

2.4.3 Kernel logger

We have implemented logging functionality in the kernel. This provides valuable information that can be used to debug and prove the correctness of the scheduling algorithms implemented by us. The kernel logger can be used to log all kinds of events. It currently can log two types of events:

- Log the next running process on every context switch.
- Log the current running process on every clock interrupt.

The kernel logger consists of the following parts:

- A buffer in the kernel to store log entries.
- An API call to start logging.
- The code to log events.
- An API call to copy the kernel log to an user program.

Implementation
The buffer in the kernel is an array of *klog_entry* structures. The size of the buffer is defined in KLOG_SIZE. The declaration of the *klog_entry* structure and the size can be found in */usr/src/include/minix/klog.h.* The buffer itself is declared in */usr/src/kernel/proc.h.*

The API call to start logging is *klog_set().* It currently takes one parameter and that is the type of event to log. Logging will be automatically stopped if the buffer is full. The state of the kernel logger is saved in the global kernel variable *klog_state.* If the state is 0 it will not log. If the state is 1 it will log. The event type to log is saved in the global kernel variable *klog_type.* The API call parameter to set the type can have the the value defined in KLOG_CONTEXTSWITCH or KLOG_CLOCKINT. This API call sets the value of *klog_state* to 1 and the value of *klog_type* to the type specified in the API call. A system call and kernel call is involved to archieve this.

klog_set() is implemented as a library function of the klog library and is implemented in */usr/src/lib/klog/klog_set.c.* *klog_set()* creates a new message and sets the field containing the state of the kernel logger to 1 and the field containing the event type to the type specified. After that it will do a system call by calling *_syscall()* with the process manager as destination and KLOG_SET as system call number.

The system call handled by the process manager is implemented in */usr/src/servers/pm/klog.c* in the *do_klog_set()* function. The only thing this function does is invoking *sys_klog_set()* with the kernel logger state and event type as arguments. *sys_klog_set()* is a system library function that does the SYS_KLOG_SET kernel call. The *sys_klog_set()* function is implemented in */usr/src/lib/syslib/sys_klog_set.c.*

The SYS_KLOG_SET kernel call is implemented in */usr/src/kernel/system/do_klog_set.c* in the *do_klog_set()* function. It simply sets the values of the *klog_state* variable and the *klog_type* variable to the values specified in the received message.

The code to log the sequence of processes that are running is implemented in the *pick_proc()* function in */usr/src/kernel/proc.c.* This function is called on every context switch and chooses the next process to run. The code to add log entries first checks if it should log (checking the kernel log state and event type). If it can log it will check if the previous logged process is the same as the current process or not. This is only done for the clock and system task. These tasks can call functions like *enqueue()* and *dequeue()* which in turn can call *pick_proc().* As these tasks have the highest priority and are running (and thus scheduled) *pick_proc()* will return one of these tasks. Because these are not real context switches we don't want them to be logged. After that it will make a new log entry using the *klog_ptr* global kernel variable which points to the current log entry. After creating a new entry it will increase *klog_ptr.* If *klog_ptr* does not point to a valid entry any more because the buffer is full it will reset *klog_ptr* and it will set *klog_state* to 0 to stop logging. Currently the only indication if the buffer is full is a message printed to the console.

The code to log the current running process on every clock interrupt is implemented in */usr/src/kernel/clock.c* in the *clock_handler()* function. This code first checks if it should log (checking the kernel log state and event type). If it should log it adds a new log entry to the buffer. After that it will check if the buffer is full to stop logging.

After the kernel buffer is full we will have to copy the buffer to an user program to do something useful with it (i.e saving to disk). This has to be done before start logging again, otherwise the data in the buffer will be overwritten. This is done with the *klog_copy()* API call which as the name suggests copies the kernel log to the user program. *klog_copy()* is implemented as a library function of the klog library in */usr/src/lib/klog/klog_copy.c.* It takes a void pointer as argument. The user program should make a kernel log buffer of exactly the same size as the buffer in the kernel. It then can call *klog_copy()* with a pointer to this buffer as argument. *klog_copy()* creates a new message and sets the value of one of the fields to the pointer specified. It then invokes *_syscall()* to do a system call with the process manager as destination and KLOG_COPY as system call number.

The KLOG_COPY system call handled by the process manager is implemented in */usr/src/servers/pm/klog.c* in the *do_klog_copy()* function. The only thing this function does is calling *sys_klog_copy()* with the endpoint of the process and the pointer specified in the received message as arguments. Endpoints are a way to uniquely identify processes (see section 3.1 on page 58 for a more detailed description of endpoints). *sys_klog_copy()* will make the SYS_KLOG_COPY kernel call and passes the two arguments to the kernel.

The SYS_KLOG_COPY kernel call is handled by the system task and implemented in */usr/src/kernel/system/do_klog_copy.c* in the *do_klog_copy()* function. Even though the function name is the same as the function in the process manager described above, it is not the same function. The function in the process manager handles the system call and the function in the kernel handles the kernel call. This function first calculates the number of bytes to copy by multiplying the size of the *klog_entry* structure by the number of entries in the buffer (KLOG_SIZE). Copying is done using physical memory addresses. Both the addresses to the buffer in the kernel and the address stored in the pointer to the buffer in the user program are virtual addresses. They have to be converted to physical addresses. The address to the buffer in the kernel is converted using the *vir2phys()* function and the address to the buffer in the user program is converted using the *numap_local()* function. After that it will call *phys_copy()* with the virtual addresses of both buffers and the number of bytes to copy.

We have created an application to convert a log file to a graph.

2.4.4 Real-time scheduler dump

The *show_rt_data()* function in */usr/src/kernel/proc.c* dumps the interesting data from the real-time scheduler. If Rate-Monotonic (RM) is the active real-time scheduler it will dump the queue reserved for real-time processes. If Earliest Deadline First (EDF) is the active scheduler it will dump the four data structures used by the EDF scheduler. This function can be called at strategic places in the code for debugging purposes.

2.4.5 Debug key binding

The Information Server offers a few key bindings that dump useful information to the console. It can dump the following information by default:

- F1, Kernel process table
- F2, Process memory maps
- F3, System image
- F4, Process privileges
- F5, Boot monitor parameters
- F6, IRQ hooks and policies
- F7, Kernel messages
- F9, Scheduling queues
- F10, Kernel parameters
- F11, Timing details (if enabled)
- Shift + F1, Process manager process table
- Shift + F2, Signals
- Shift + F3, Filesystem process table
- Shift + F4, Device/Driver mapping
- Shift + F6, Reincarnation server process table
- Shift + F7, Memory free list
- Shift + F8, Data store contents

As you can see it already has a function to dump the scheduling queues. This function however only shows which processes are in which queues. This is not enough for real-time processes. We have added a keybinding (**Shift + F9**) that will invoke the *show_rt_data()* function discussed in the previous section.

Implementation

We have added a function *rt_sched_dmp()* to */usr/src/servers/is/dmp_kernel.c*. The prototype of this function can be found in */usr/src/servers/is/proto.h.* This function calls *sys_rt_show_data()* which is a system library function that will do a kernel call (SYS_RT_SHOW_DATA). The SYS_RT_SHOW_DATA kernel call is implemented in */usr/src/kernel/system/do_rt_show_data.c* in the *do_rt_show_data()* function. This function checks if there is a real-time scheduler set. If this is the case it will call *rt_show_data()* which will dump the scheduler information. If no real-time scheduler is set it will print a message that there is no real-time scheduler set. Now we will discuss how we have bound the *rt_sched_dmp()* function to the Shift + F9 key. The original information server does not hook to this key. We will have to add a hook for this key in the *init_server()* function in */usr/src/servers/is/main.c.* The last thing we have to do is add a new entry in the key map called *hooks.* This map binds the keys the information server is hooked to, to the functions that have to be invoked on a key press.

2.4.6 Setting the active real-time scheduler

After the system has booted there is no real-time scheduler set. Before starting any real-time processes the real-time scheduler must be set to one of the implemented real-time schedulers (RM or EDF). The real-time scheduler can only be changed if there are no real-time processes running. Each scheduler has its own library function to set the real-time scheduler. RM has the *rt_set_sched_rm()* function that is implemented in */usr/src/lib/rtminix3/rt_set_sched_rm.c.* This function takes one parameter to set if the priority of a processes scheduled by RM should be unique or not. EDF has the *rt_set_sched_edf()* function that takes no parameters and is implemented in */usr/src/lib/rtminix3/rt_set_sched_edf.c.* These library functions make use of the RT_SET_SCHED system call implemented in */usr/src/servers/pm/rt.c* (*do_rt_set_sched()* function). This system call makes use of the SYS_RT_SET_SCHED kernel call implemented in */usrc/src/kernel/system/do_rt_set_sched.c.*

2.5 Rate-Monotonic Scheduling

2.5.1 Requirements

The most important requirements for processes scheduled by Rate-Monotonic are:

- We assume that RM processes are periodic.
- Because our system can use multiple real-time schedulers (but only one at the same time) there should be a way to tell the system which scheduler to use.
- There should be a way to tell the system that the process is a real-time process.
- There should be a way to release remaining reserved calculation time.
- RM processes have fixed priorities.

2.5.2 Design

A RM scheduler can be implemented in (at least) two ways on Minix 3:

- using a queue for each RM priority.
- using a single priority based queue for all RM processes.

We have successfully made prototypes for both approaches.

Using multiple queues
This approach is easier to implement than the single scheduling queue approach. We do not need to extend the process structure as processes with a maximum priority in the range of the reserved RM queues are real-time and this priority can be used for RM scheduling. We will have to extend the number of scheduling queues that minix supports with the number of different priorities that processes scheduled using RM can have. The added queues are reserved for rate-monotonic processes. The queues for RM processes have a higher priority than non real-time user processes and a lower priority than system processes. Furthermore we will have to make sure that RM processes always have their maximum priority. This can be done by making sure that RM processes always have quantum left and thus the processes will never be lowered in priority. We also have to make sure that non real-time processes cannot get scheduled in a queue reserved for RM processes.

This approach also has some disadvantages. One of the most important is that by increasing the number of scheduling queues it will take more time to find a ready process and thus context switches take more time. Furthermore the number of priorities for RM is limited to the number of scheduling queues reserved for RM.

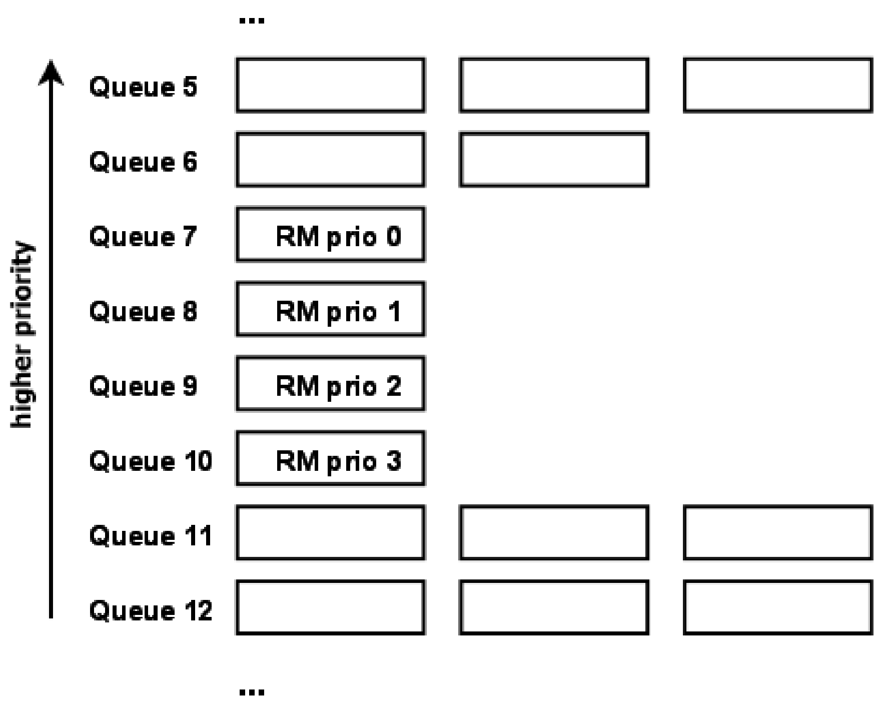

Figure 3: Using multiple (4) queues for RM processes.

Using a single prioritized queue
With this approach we do not use the scheduling priorities of Minix 3. All processes scheduled by RM share the same Minix 3 scheduling priority. In addition they have a RM priority. It uses a single scheduling queue which is sorted on RM priorities. We will have to make sure that for every insert into this queue we find the right place in the queue. To implement this system we will have to extend the process structure to save the RM priority. Large parts of the design are the same though. We still have to make sure that RM processes will not be lowered in priority. We also have to make sure that non real-time processes cannot be scheduled in the RM queue. We have choosen to use this approach as it is easier to merge with the EDF scheduler and it does not have the disadvantages of the first approach.

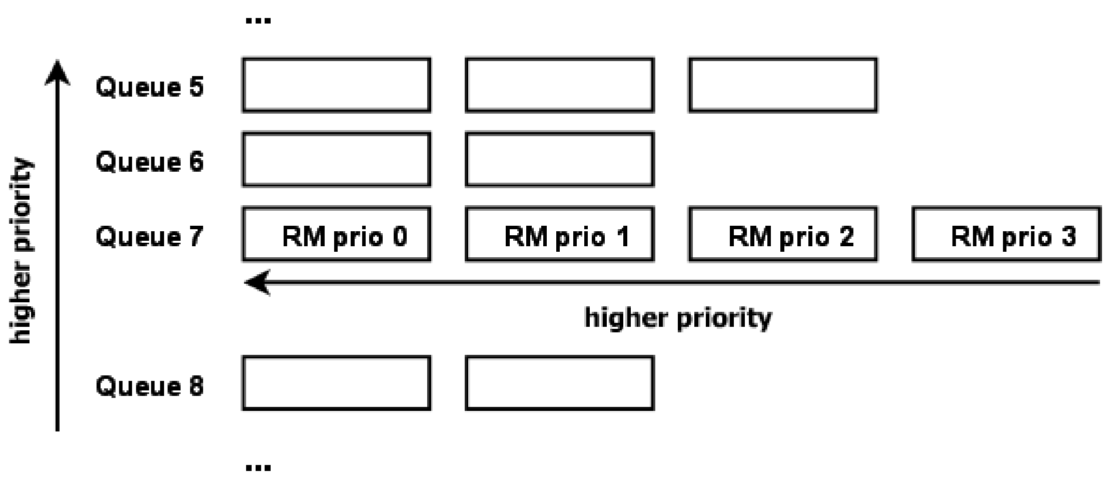

Figure 4: Using a single prioritized queue for RM processes.

2.5.3 Implementation

The implementation can be divided in several parts:

- Adding a library call which the process calls to get real-time.
- Adding a system call for transforming a process to real-time.
- Adding a kernel call for transforming a process to real-time.
- Modifying the scheduler to support RM.

Because the system call is relying on the existence of the kernel call and the library call is relying on the existence of the system call we will use a top-down approach to discuss these parts, starting with the kernel call.

The SYS_RT_SET kernel call
The SYS_RT_SET kernel call is implemented in */usr/src/kernel/system/sys_rt_set.c* This kernel call is used for all real-time schedulers. We will discuss only the parts that regard the Rate-Monotonic scheduler.

If the system task receives a kernel call request with the number defined in SYS_RT_SET it will invoke *do_rt_set()* and passes the received message as argument. *do_rt_set()* will first check if the scheduler specified in the received message is valid. The current real-time scheduler is saved in a global kernel variable called *rt_sched.* The scheduler is valid if *rt_sched* is set to a valid real-time scheduler and the scheduler specified in the message is the same as *rt_sched.* After that it will check if the process endpoint is valid using the *isokendpt()* function. Endpoints are a way to uniquely identify processes (see section 3.1 on page 58 for a more detailed description of endpoints). This function not only checks if the endpoint is valid but also returns the process number which is mappable to the process table. An endpoint can be invalid if for example the process is killed in the mean time. If the endpoint is valid *proc_addr()* is called with the process number as argument. This function will return a pointer to the process entry structure in the process table. After that it will check if the process is a real-time process using the is_rtp() macro. A process that is already real-time may not request to be real-time for a second time. Finally if all tests are passed it will invoke the right function to handle the scheduler specific parts. Each real-time scheduler has its own function to handle these parts. In the case of Rate-Monotonic *do_rt_set_rm()* will be called with the received message and the pointer to the process structure as arguments. The result of the call will be returned and the main loop of the system task will send a message back.

The first thing *do_rt_set_rm()* does is checking if the global kernel variable *rm_prio_policy* is set to PRIO_UNIQUE. If this is the case the rate-monotonic priority of a process should be unique. In that case it will loop through the process table trying to find a real-time process with the same priority as requested by the process that wants to get real-time. If it has found a process an error code will be returned. Next it will check if the process is currently scheduled. The process should not be scheduled normally since it is waiting for a reply from the process manager. But if it is scheduled we don't consider it as an fatal error and we just deschedule it by calling the *lock_dequeue()* function with a pointer to the process structure as argument. Now we can finally make the process real-time. The process structure in */usr/src/kernel/proc.h* has been extended with the *p_rt* field which indicates if a process is real-time (1) or not (0). This field is used by all real-time schedulers. We set the *p_rt* field in the process structure to 1. This indicates that this is a real-time process. We set *p_priority* and *p_max_priority* to the queue reserved for rate-monotonic processes. We set *p_rt_priority* to the rate-monotonic priority the process has requested. Finally we check if the process is runnable which is only the case if we descheduled it and thus should not happen normally. If it is runnable we will schedule the process by calling *lock_enqueue()* with a pointer to the process structure as argument.

The RT_SET system call

The RT_SET system call is implemented in */usr/src/servers/pm/rt.c*. If the process manager receives a system call request with the number defined in RT_SET it will invoke *do_rt_set()*. The *do_rt_set()* function simply checks which scheduler type is specified in the received message and calls *sys_rt_set()* with the right parameters. In the case of Rate-Monotonic these are the process' endpoint, the scheduler type and the Rate-Monotonic priority. *sys_set_rt()* is a system library function that will do the SYS_RT_SET kernel call and returns the result. This result is also returned by *do_rt_set()* and the main loop of the process manager will send a message back with the result to the process that did the system call. The implementation of this system call could be ommitted if a user process could directly do a kernel call, which is not possible for security reasons.

The rt_set_rm() library call
The *rt_set_rm()* library call is implemented in */usr/src/lib/rtminix3/rt_set_rm.c.* This library function is part of the API that that real-time applications use. This function takes the Rate-Monotonic priority as parameter. It makes the RT_SET system call supplying the scheduler type (Rate-Monotonic) and the Rate-Monotonic priority as parameters and returns the result of the system call.

Scheduler modifications
The following modifications have been made to the scheduler:

- If the real-time process is scheduled it should be inserted in the queue reserved for real-time processes. This queue is sorted on rate-monotonic priorities and thus the process should be inserted in the right place.
- Real-time processes may never be descheduled when they have used their full quantum. Real-time processes may also not be decreased in priority.

The *enqueue()* function in */usr/src/kernel/proc.c* schedules processes and is modified to insert processes in the queue reserved for real-time processes based on their priority. See Listing 3 for a simplified implementation. The original code for inserting a process in the scheduling queues will be executed only if the process is not real-time or the process is real-time and the real-time scheduler is set to EDF. The added code will be executed if the process is real-time and the real-time scheduler is set to RM. It first loops through the queue until the pointer to the next process has a lower RM priority than the process that we insert. If the queue is empty or the process we insert has the lowest priority the pointer will point to NIL_PROC which indicates an non-valid process (used to indicate the end of the queue). After it has found the right place in the list it will insert it. Each scheduling queue has a head and a tail which is saved in the *rdy_head* and *rdy_tail* arrays. Finally if the process is the last on the queue we set the *rdy_tail* entry for this queue to that process. We don't use the *rdy_tail* at all but we want to keep the debug routines happy as the queues are not consistent otherwise.

Listing 3: Modifications of the enqueue() function for RM.

```
PRIVATE void enqueue(rp)
register struct proc *rp;
{
  int q;      /* scheduling queue to use */
  int front; /* add to front or back */
  register struct proc **xpp;

  sched(rp, &q, &front);

  if(! is_rtp(rp) || (is_rtp(rp)
  && rt_sched == SCHED_EDF)) {
      /* removed original scheduling code */
  } else if (is_rtp(rp) && rt_sched == SCHED_RM) {
      xpp = &rdy_head[q];

      while (*xpp != NIL_PROC &&
            (*xpp)->p_rt_priority <= rp->p_rt_priority) {
          xpp = &(*xpp)->p_nextready;
      }

      rp->p_nextready = *xpp;
      *xpp = rp;

      if (rp->p_nextready == NIL_PROC) {
          rdy_tail[q] = rp;
      }
  }

  pick_proc();
}
```

The only reason processes will be lowered in priority is if they use their full quantum. If we make sure that a real-time process will never use their full quantum it will never be lowered in priority and never be descheduled/rescheduled because of it. The easiest way to archieve this (although maybe not the most elegant way) is to refill the quantum for real-time processes that are interrupted by the clock (see listing 4).

Listing 4: Modifications of the clock_handler() function for real-time processes.

```
/* generic accounting for real-time processes */
if (is_rtp(proc_ptr)) {

    /* make sure quantum never reaches zero.
     * We don't use p_ticks_left for RT processes.
     */
    proc_ptr->p_ticks_left = 100;
}
```

2.5.4 Testing

If we can prove that the scheduling queue used by the RM scheduler is always sorted on RM priority (increasing), we have proven the correctness of the RM scheduler. We have done this by regular using the Shift+F9 key binding that dumps the scheduling queue used by the RM scheduler.

As an example we start 8 processes, all with different priorities in a random order. The number in the name of the process is equal to the RM priority. These processes use more calculation time than the system can handle to keep the scheduling queue crowded. In a real life scenario it is better to use less than 69% CPU time to guarantee deadlines. If we press Shift+F9 the following is printed to the terminal:

```
RM Q: [2:71129:rm2] [3:71124:rm3] [4:71127:rm4] [5:71132:rm5]
      [6:71131:rm6] [7:71125:rm7] [8:71128:rm8]
```

The processes are printed in the following format: [RM priority:endpoint:name]. Process rm1 is not printed because it is blocked by calling the sleep() function. Process rm2 will be running if there are no processes with a higher priority (system processes) ready. This output corresponds to our expectations.

Using the kernel log function we can get a better view of when processes run. We have created a log of three processes scheduled with RM. The running process on each clock interrupt was saved. Using a small java application we have created a graph (figure 5).

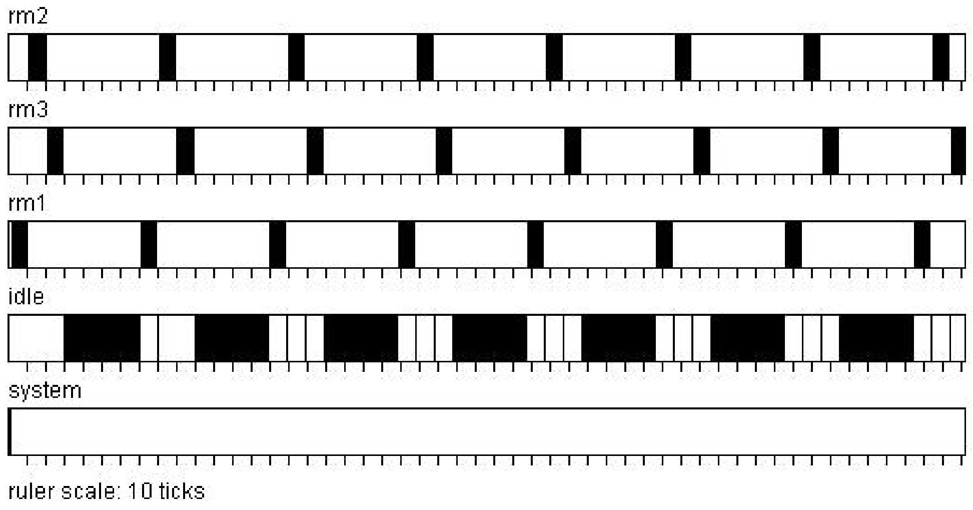

Figure 5: Three processes scheduled by RM.

2.5.5 Problems

In this section we discuss problems that are not addressed.

Support for periodic processes
We did not implement any specific features for periodic processes while the RM scheduler is often used for periodic processes. To give better support for period processes there should be a way for a process to indicate its period. There should also be a way to release the processor and wait for the next period. Currently we use the POSIX API sleep() function to wait till the next period. This is far from ideal because not all periods will have the same duration (i.e depends on the code path executed). The period will then drift.

2.6 Earliest Deadline First Scheduling

2.6.1 Requirements

In this section we will look at the requirements that EDF processes have and define the API.

The most important requirements for processes scheduled by EDF are:

- We assume that EDF processes are periodic.
- Because our system can use multiple real-time schedulers (but only one at the same time) there should be a way to tell the system which scheduler to use.
- There should be a way to tell the system that the process is a real-time process specifying the period and calculation time.
- There should be a way to release remaining reserved calculation time.
- Deadlines equals next period starts.
- The runnable process with the earliest deadline should run if no system processes with a higher priority are ready.

2.6.2 Design

It is possible to implement an EDF scheduler in user space [4]. This has been done by implementing the EDF scheduler in a separate user space process. The implementation uses two heaps for EDF processes: a deadline heap and a period heap. Each EDF process is in exactly one of the two heaps. The process with the earliest deadline will be increased in priority using the SYS_NICE kernel call. An advantage of implementing EDF in user space is that it is more modular and it does not need modifications to the scheduler in the kernel. Unfortunately this approach also has some disadvantages. The most important one is that the process manager is not aware of process states (only the kernel is). In our case the currently scheduled EDF process should be the process with the earliest deadline that is runnable. In this implementation if the currently scheduled EDF process blocks it will still keep the high priority untill it releases remaining reserved calculation time or misses the deadline and thus no other EDF process runs while the process is blocked. This is far from optimal. This is the reason why we have choosen to implement EDF in the kernel.

With the requirements of section 2.6.1 we can now define the API calls needed:

- rt_set_sched_edf()
- rt_set_edf(period, calculation_time)
- rt_nextperiod()

rt_set_sched_edf() sets the real-time scheduler to EDF. *rt_set_sched_edf()* should be called before starting up any EDF processes or by the EDF process that first starts up (before calling *rt_set_edf()*). *rt_set_edf()* takes the period and calculation time as parameters. Both are expressed in system ticks. This API call will transform a normal user process into a real-time process scheduled by EDF. *rt_nextperiod()* gives EDF processes a way to release remaining reserved calculation time and will let the process wait till the next period starts.

An EDF process can be in the following four states:

- The earliest deadline and scheduled.
- Not the earliest deadline and calculation time left in period.
- Calculation time left in period and blocked.
- Waiting for a new period.

The first state can be subdivided in running and not running. It can only run if there are no processes with a higher priority ready.

We use four data structures to keep hold of the EDF processes. Each of these reflects one of the four states described above. *edf_rp* is a data structure that holds 1 process. If an EDF process is scheduled, *edf_rp* holds the same process as the scheduling queue reserved for real-time processes. If no EDF process is scheduled, *edf_rp* has the value of NIL_PROC. The run queue holds processes that are ready to run but don't have the earliest deadline. This queue is sorted on deadline (ticks till next period, increasing). The block list contains processes that are blocked while they were scheduled, they still have calculation time left in the current period. The wait queue holds processes that are waiting for a new period and is sorted on next period start (deadline). A process will be added to this queue if it has used all calculation time or it voluntary released its remaining reserved calculation time. Figure 6 shows the four EDF data structures.

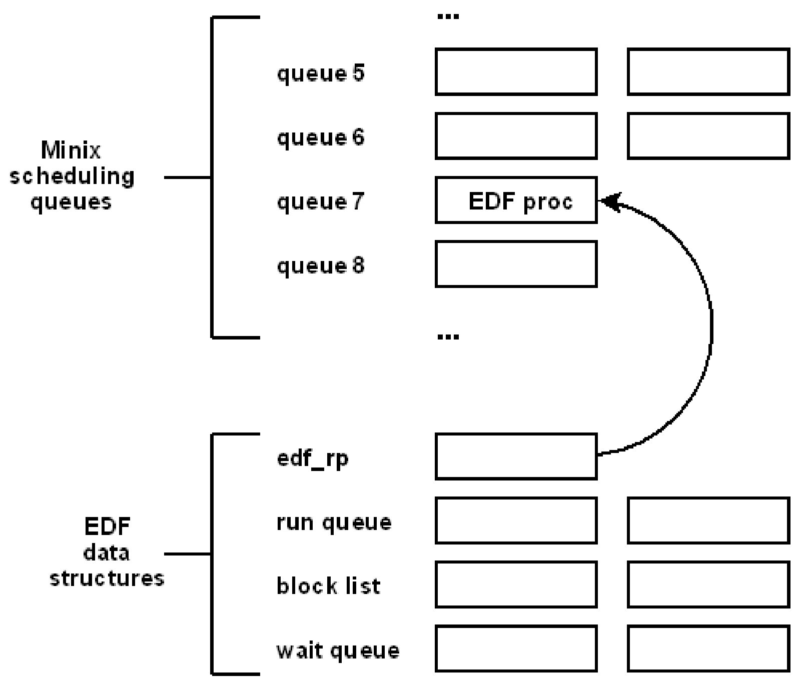

Figure 6: EDF data structures. The arrow indicates that if an EDF process is scheduled, the process in edf_rp is the same as the process in scheduling queue 7.

One of the things that should be decided is what to do if a process misses its deadline. Processes can miss their deadline if they are in *edf_rp,* the run queue or the block list. If a process that is in *edf_rp* misses its deadline it will be descheduled and the period will be renewed. It will be moved to the run queue. If a process in the run queue or block list misses the deadline only the period will be renewed.

2.6.3 Implementation

Process structure extensions
We have added a few new fields to the process structure (struct proc) in */usr/src/kernel/proc.h.* These are:

- char p_rt: indicates if a process is real-time (used by all real-time schedulers).
- int p_rt_period: the period of an EDF process in ticks.
- int p_rt_calctime: the calculation time of an EDF process in ticks.
- int p_rt_nextperiod: ticks till next period start. If a process has ticks left in current period this equals the deadline.
- int p_rt_ticksleft: calculation time left in current period in ticks.

- struct proc *p_rt_link: used by EDF to link processes to eachother in the EDF data structures.

We have also added a few fields for statistics gathering:

- unsigned int p_rt_periodnr: current period number which the process is in (starts at zero).
- unsigned int p_rt_totalreserved: total count of reserved calculation time in ticks.
- unsigned int p_rt_totalused: total used calculation time in ticks.
- unsigned int p_rt_missed_dl: number of missed deadlines.

Global kernel variables
To support the EDF scheduler four global kernel variables are added in */usr/src/kernel/proc.h*. These are:

- struct proc *edf_rp: pointer to current scheduled EDF process (points to NIL_PROC if there is none).
- struct proc *edf_run_head: pointer to the head of the EDF run queue.
- struct proc *edf_block_head: pointer to the head of the block list.
- struct proc *edf_wait_head: pointer to the head of the wait queue.

NEXTPERIOD bit
Each process entry has a *p_rts_flags* field. This field marks if a process is runnable. If the value of *p_rts_flags* is zero the process is runnable. We have added a bit that can be set in *p_rts_flags* called NEXTPERIOD. This bit indicates if a process is waiting for a new period or not.

rt_set_edf() API call
The implementation of the *rt_set_edf()* API call involves a system call and a kernel call. We will start at the kernel call and work our way down to the library function. We use the SYS_RT_SET kernel call which is a generic kernel call used by all real-time schedulers to transform normal user processes into real-time processes. This kernel call is implemented in */usr/src/kernel/system/sys_rt_set.c*. This kernel call consists of a generic part which does some sanity checks and retrieves the process entry in the process table of the process that wants to be real-time. At the end of this part it will call a scheduler specific function which in the case of EDF is *do_rt_set_edf()* (see listing 5 for the implementation). A pointer to the message received by the system task and a pointer to the process entry structure are supplied as arguments.

The first thing that *do_rt_set_edf()* does is checking if the requested period and calculation time parameters are valid. This consists of two checks. First, the parameters should be larger than zero (it makes no sense having a process with a period or calculation time of 0). Second, a process may not request a calculation time that is larger than the period.

After that it will check if the process is runnable. A process should not be runnable because it should be blocked waiting for a reply from the process manager. If the process is runnable we consider this as a fatal error and it will return with an error code. If the process is blocked and not scheduled we can modify the process entry fields. The *p_rt* field is set to 1 to indicate that this process is real-time. The *p_rt_period* and *p_rt_calctime* fields are set to the requested period and calculation time. The first period starts immediately so we set the *p_rt_nextperiod* field equal to the *p_rt_period* field and we set the *p_rt_ticksleft* field equal to the *p_rt_calctime* field. The *p_priority* and *p_max_priority* fields are set to the queue reserved for real-time processes. Because the process is blocked waiting for a reply from the process manager we put the process on the block list.

Listing 5: Implementation of the do_rt_set_edf() function.

```
PRIVATE int do_rt_set_edf(message *m_ptr,
                          struct proc * rp)
{
  if (m_ptr->EDF_PERIOD <= 0 || m_ptr->EDF_CALCTIME <= 0)
      return (EINVAL);

  if (m_ptr->EDF_CALCTIME > m_ptr->EDF_PERIOD)
      return (EINVAL);

  if ( rp->p_rts_flags == 0) return (EGENERIC);

  rp->p_rt = 1;

  rp->p_rt_period = m_ptr->EDF_PERIOD;
  rp->p_rt_calctime = m_ptr->EDF_CALCTIME;
  rp->p_rt_ticksleft = rp->p_rt_calctime;
  rp->p_rt_nextperiod = rp->p_rt_period;
  rp->p_priority = rp->p_max_priority = RT_Q;

  rp->p_rt_link = edf_block_head;
  edf_block_head = rp;

  return (OK);
}
```

The RT_SET system call is implemented in */usr/src/servers/pm/rt.c.* If the process manager receives a system call request with the number defined in RT_SET it will invoke the *do_rt_set()* function. Based on what scheduler type is defined in the request message it will call *sys_rt_set()*. *sys_rt_set()* is a system library function that will do the SYS_RT_SET kernel call and returns the result. In the case of EDF the following parameters are specified: The calling process endpoint, the scheduler type (SCHED_EDF), the requested period and the requested calculation time per period. Endpoints are a way to uniquely identify processes (see section 3.1 on page 58 for a more detailed description of endpoints).

Finally we discuss the *rt_set_edf()* library function which is part of the API that real-time processes use. This function is implemented in */usr/src/lib/rtminix3/rt_set_edf.c*. It takes two parameters: the period and the calculation time, both in ticks. It creates a new message and sets the scheduler type field to SCHED_EDF and the period and calculation time fields to the specified parameters. After that it will call and return the result of *_syscall()* with the process manager as destination, RT_SET as system call and a pointer to the message.

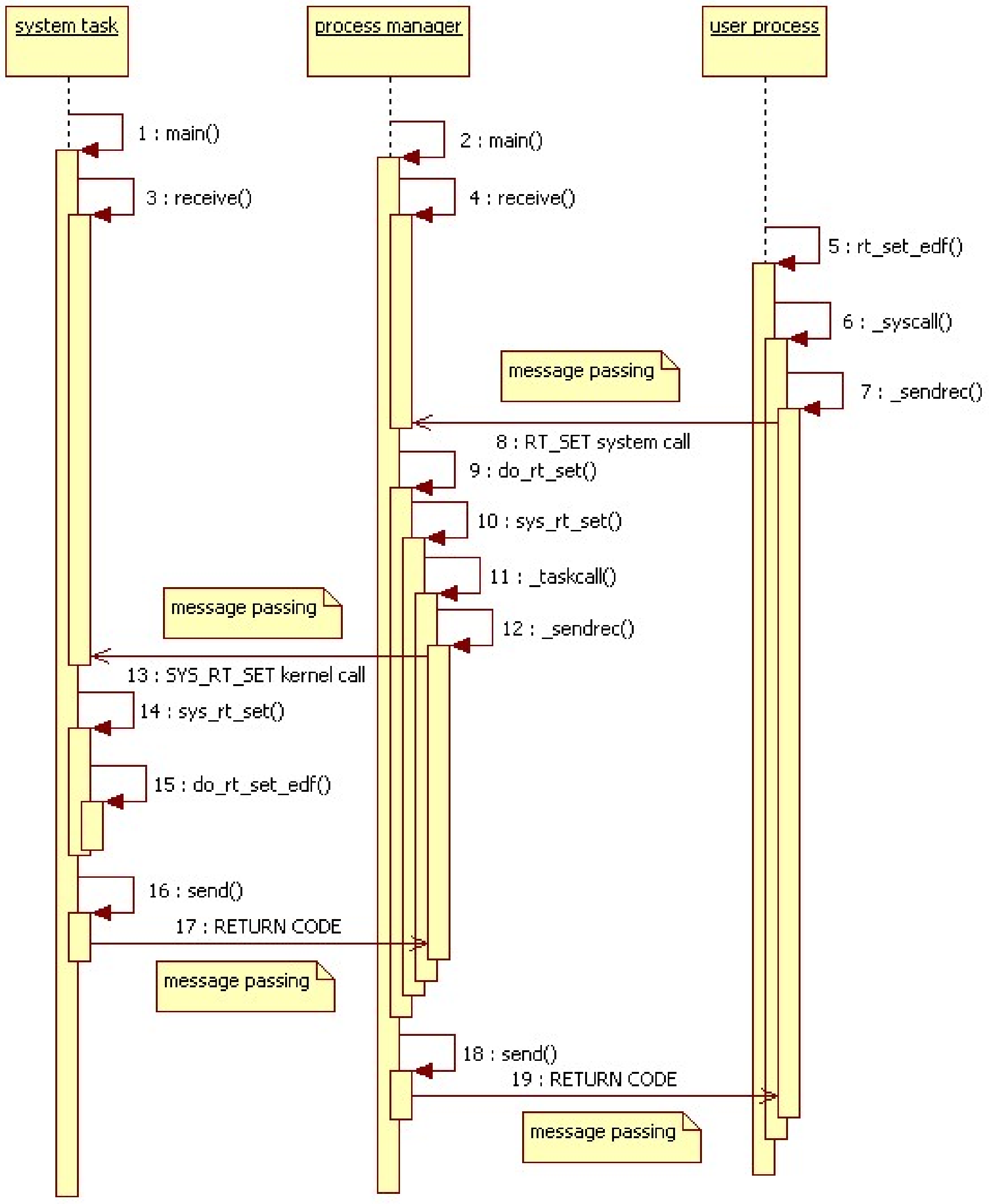

Figure 7: Call chain of the rt_set_edf() library function.

edf_sched() function
The *edf_sched()* function checks which EDF process should run next. It should always be called in the following situations:

- A process is descheduled.
- A process is added to the run queue.

This function is implemented in */usr/src/kernel/proc.c.* The first case to check is if *edf_rp* is set to NIL_PROC. This is the case if for example an EDF process is descheduled. If it is set to NIL_PROC it checks of the EDF run queue is not empty. If there are processes in the run queue it will move the process from the run queue to *edf_rp.* After that it will schedule the process in *edf_rp.* The second case to check is if *edf_rp* contains an EDF process and the head of the run queue contains a process with an earlier deadline than *edf_rp.* If this is the case *edf_rp* should be descheduled and added to the run queue. The head of the run queue is the new runnable EDF process and will be moved to *edf_rp.* Finally it will schedule the new process in *edf_rp.*

clock_handler()
The *clock_handler()* function handles the clock interrupt. It is implemented in */usr/src/kernel/clock.c.* The *clock_handler()* function renews the quantum (*p_ticks_left* field in the process entry) if the current running process is a real-time process. This is done to make sure that the quantum of real-time processes never reaches zero which would cause a decrease in priority. Furthermore the *clock_handler()* function will do accounting for EDF processes. Each of the four data structures used by the EDF scheduler will be updated. Some events need scheduling. If this is the case the variable *call_do_clocktick* will be set to 1 and at the end of the function it will check if *call_do_clocktick* is set to one. If it is set it will send a notification to the clock task. The clock task will handle the event and takes care of the scheduling.

The first data structure that is updated is *edf_rp.* If there is a process in *edf_rp, p_rt_ticksleft* (calculation time left) and *p_rt_nextperiod* (deadline) will be decreased. After that it will check if *p_rt_ticksleft* and/or *p_rt_nextperiod* has reached zero. In that case it will set *call_do_clocktick* to 1 because the process should be descheduled. The process has either used up all calculation time or missed the deadline.

The second data structure that is updated is the run queue containing processes which have calculation time left in the current period but cannot be scheduled because they are not the process with the earliest deadline. All processes in this queue will be updated. The *p_rt_nextperiod* field of the process entry will be decreased if it was not zero. if *p_rt_nextperiod* is zero it has missed the deadline. We will have to renew the period. We do this by resetting the deadline (*p_rt_nextperiod*) and the calculation time left in the current period (*p_rt_ticksleft*). The statistics fields of the process will also be updated (i.e increasing the missed deadlines counter). If one or more processes have missed the deadline, the run queue might be inconsistent because the deadlines for these processes were renewed and this queue is sorted on deadline (increasing). We will have to resort the run queue in that case. We use a simple bubble sort algorithm for this.

The third data structure that is updated is the wait queue containing processes waiting for the next period start. All processes in this queue will be updated. The *p_rt_nextperiod* field of the process entry will be decreased. If this field has reached zero the new period starts for this process. *call_do_clocktick* will be set to 1 so that the clock task will handle this event.

The last data structure that is updated is the block list containing processes with calculation time left in the current period but are blocked. All processes in this list will be updated. the *p_rt_nextperiod* field of the process entry will be decreased. If this field has reached zero it has missed the deadline. As it is still blocked we will leave the process in the list and renew the period. We do this by resetting the deadline (p_rt_nextperiod) and the calculation time left in current period (*p_rt_ticksleft*). Also some statistics fields will be updated (i.e increasing the missed deadlines counter).

do_clocktick()
If the clock task receives a notification it will invoke *do_clocktick()* to handle the event. Notifications do not contain any more nformation but who sent the message. Unfortunately we don't know what has caused the event by this lack of information. Three EDF related events can have occured:

- The currently scheduled EDF process has missed the deadline.
- The currently scheduled EDF process has used all calculation time.
- A new period starts for one or more processes in the wait queue.

All of the situations above are checked.

If the currently scheduled EDF process has missed the deadline it will be removed from the scheduling queues. The period is renewed by resetting the *p_rt_nextperiod* (deadline) and *p_rt_ticksleft* (calculation time left) fields of the process entry. Also some statistics fields are updated. After that it will be inserted into the run queue at the right place. After that *edf_rp* will be set to NIL_PROC and *edf_sched()* is called to check which process has the earliest deadline and should be scheduled.

If the currently scheduled EDF process has used up all calculation time it will also be removed from the scheduling queues. The process will be inserted in the wait queue at the right place. The NEXTPERIOD bit in the *p_rts_flags* field will be set. If any bit in *p_rts_flags* is set the process is not runnable, thus this makes sure that no other part of the kernel will schedule this process while waiting for the new period start. After that *edf_rp* will be set to NIL_PROC and *edf_sched()* is called to check which process has the earliest deadlne and should be scheduled.

The head of the wait queue will be checked for new period starts. Only the head will be checked because the queue is sorted on new period start. If the head has a new period start, the process will be removed from the wait queue and inserted into the run queue.

The NEXTPERIOD bit will be cleared because the process is not waiting for a new period anymore. Some statistics fields of the process will be updated and *edf_sched()* will be called to check which process has the earliest deadline. Because we removed the process from the wait queue there is a new head. The procedure above will repeat untill the head does not have a new period start or the head has no valid process (NIL_PROC).

mini_send() function
The *mini_send()* function is used to send a message to an other process. The function is implemented in */usr/src/kernel/proc.c.* We have modified *mini_send()* to support our EDF scheduler. This is needed because this function schedules or deschedules processes if needed. In particular there are two cases which have to be supported:

- The destination is a real-time process scheduled by EDF waiting for a message.
- The sender is a real-time process scheduled by EDF and the destination is not waiting.

In the first case the standard procedure is to unblock the destination by clearing the RECEIVING bit in *p_rts_flags* and if *p_rts_flags* is zero (runnable) it will call *enqueue()* with a pointer to the destination process entry as argument. The above method is used if the destination is a normal user process or a real-time process using RM scheduler. If the destination is a real-time process using the EDF scheduler it will be on the block list. We will have to remove it from the block list and add it to the run queue (if *p_rts_flags* is zero). After that we will call *edf_sched()* to check which process has the earliest deadline and should be scheduled now.

In the second case for normal user processes and processes using the RM scheduler it will set the SENDING flag in *p_rts_flags* of the sender. After that it will deschedule the sender. Furthermore it will put the sender on the destinations queue of processes whishing to send to the destination. In addition, if the sender is a real-time process scheduled by EDF, the sender will be put on the EDF block queue. After that *edf_rp* will be set to NIL_PROC and *edf_sched()* is called to determine which process has the earliest deadline and should be scheduled now.

mini_receive() function
The mini_receive() function is used to receive a message. The function is implemented in */usr/src/kernel/proc.c.* If no processes want to send a message to the caller, the caller will be blocked (RECEIVE bit set in *p_rts_flags*) and descheduled by calling *dequeue()* with a pointer to the callers process entry. In addition if the caller is a real-time process scheduled by EDF it will have to be put on the EDF block list. After that *edf_rp* will be set to NIL_PROC and *edf_sched()* is called to determine which process has the earliest deadline and should be scheduled now.

rt_nextperiod() API call

The implementation of the *rt_nextperiod()* API call involves a system call (RT_NEXTPERIOD) and a kernel call (SYS_RT_NEXTPERIOD). We will start at the kernel call and work our way down to the library function.

The SYS_RT_NEXTPERIOD kernel call is implemented in */usr/src/kernel/system/do_rt_nextperiod.c.* If the system task receives a kernel call request with the number defined in SYS_RT_NEXTPERIOD it will invoke *do_rt_nextperiod().* This function will first check if the RT scheduler is set to EDF. After that it will check if the endpoint supplied is valid by calling *isokendpt()* with the endpoint and an integer to store the process number (proc_nr) as parameters. If the endpoint is valid we can retrieve a pointer to the process by calling *proc_addr()* with the process number as parameter. Now we can check if this process is real-time using the *is_rtp()* macro. After that it will check if *p_rts_flags* is zero. It cannot be zero in normal situations as the process should be blocked waiting for a reply from the process manager. If it is zero we consider it as a fatal error and return an error. Because the process is blocked it can be found on the block list. We will remove the process from the block list and insert it in the wait queue. Finally we set the NEXTPERIOD bit in *p_rts_flags* indicating that this process is not schedulable as it is waiting for the next period.

The RT_NEXTPERIOD system call is implemented in */usr/src/servers/pm/rt.c.* If the process manager receives a system call request with the number defined in RT_NEXTPERIOD it will invoke *do_rt_nextperiod().* This function will call *sys_rt_nextperiod()* with the endpoint number of the calling process as argument and returns the result.

The *rt_nextperiod()* library function which is part of the API that real-time processes use is implemented in */usr/src/lib/rtminix3/rt_nextperiod.c.* It calls *_syscall()* with the process manager as destination and RT_NEXTPERIOD as system call and a pointer to a (empty) message. It will return the result of this call to the real-time process.

SYS_EXIT kernel call

The SYS_EXIT kernel call will be called for every process that exits. If the system task receives a kernel call request with the number defined in SYS_EXIT it will invoke *do_exit()* which is implemented in */usr/src/kernel/system/do_exit.c.* The *do_exit()* function calls *clear_proc()* and supplies a pointer to the process entry of the process that has exited. This function will do the real clean-up of the exited process. We have added code to properly exit real-time processes. For all real-time processes (RT scheduler independent) the *p_rt* field in the process entry will be set to 0. In addition for EDF processes all the EDF data structures are searched to find and remove the exited process from the data structure.

2.6.4 Testing

In this section we will discuss how we have tested the EDF scheduler.

edftop

Top is an UNIX program to monitor processes. The edftop program is a modified version to monitor processes scheduled by the EDF scheduler. This program is a very valuable tool for the real-time system programmer to check if the real-time processes behave as expected. See table 1 for the column descriptions. Below is the output of edftop while running two real-time processes:

Listing 6: Output of edftop in scenario 1.

```
load averages:  0.65, 0.26, 0.08
37 processes: 2 running, 35 sleeping
Mem: 181856K Free, 181416K Contiguous Free
CPU states:  66.12% user,   0.83% system,   0.00% kernel,  33.06% idle

PNR   ENDPT  ST PERIOD CTIME  PRD#  TLP TLNP     USED RESERVED  %U MD COMMAND
 33   71111 WNP    100    20    47    0   60      940      940 100  0 edf1
 34   71112 RUN    100    40    43   20   73     1720     1720 100  0 edf2
```

Column	Description
PNR	Process number
ENDPT	Process endpoint
ST	State of process (RUN = ready,WNP = wait for next period, BLK = blocked)
PERIOD	Period of the process
CTIME	Calculation time per period
PRD#	Current period number (not finished yet)
TLP	Calculation time left in current period
TLNP	Ticks left till next period in ticks (deadline)
USED	Number of total used ticks
RESERVED	Number of total reserved ticks
%U	Percentage of the reserved ticks used
MD	Number of missed deadlines
COMMAND	The command name of the process

Table 1: Description of the columns of edftop.

Printing missed deadlines

We have added the C macro EDF_PRINT_MD to /usr/src/kernel/config.h. If this macro is set to 1, the kernel will directly print to the terminal if a process has missed the deadline.

Dumping the four EDF data structures
Using the Shift+F9 key binding discussed in section 2.4.5 on page 23 we can dump the contents of the four EDF data structures.

Scenarios
We will try to show the correct working of the EDF scheduler by testing a few scenarios. We have created log files using the kernel log functionality. We have analyzed and visualized the results.

Scenario 1
In this scenario we will test running one real-time process. This process (with the name edf1) has a period of 100 ticks and a calculation time of 50 ticks. It will use all calculation time (the program is an infinite loop). We have logged every process that was running when a clock interrupt happened. EDF processes will be accounted in the clock interrupt handler. In this way we can check if this process gets the calculation time it has reserved.

Listing 7 shows a trimmed version of the log file.

Listing 7: A part of the log file of scenario 1.

```
113389 106709 edf1
113390 106709 edf1
<removed 46 entries from process edf1>
113437 106709 edf1
113438 106709 edf1
113439 -4 idle
113440 -4 idle
<removed 46 entries from process idle>
113487 -4 idle
113488 -4 idle
113489 106709 edf1
113490 106709 edf1
<removed 46 entries from process edf1>
113537 106709 edf1
113538 106709 edf1
113539 -4 idle
113540 -4 idle
<removed remaining entries>
```

The log file is in the following format:

<time in ticks> <process endpoint> <process name>NEWLINE

The first period of process edf1 starts at tick 113388. At tick 113389 process edf1 was billed for the first time for the calculation time in this period. After the first entry (tick 113389) edf1 has 49 ticks of calculation time left in the first period. At tick 113438 edf1 was billed for the last time in the first period. The result of subtracting 113388 from 113438 is 50, the calculation time in ticks process edf1 got in the first period. Starting from tick 113488 (exactly 100 ticks after the first period start) this process repeats again. Figure 8 shows a visual representation of the log file.

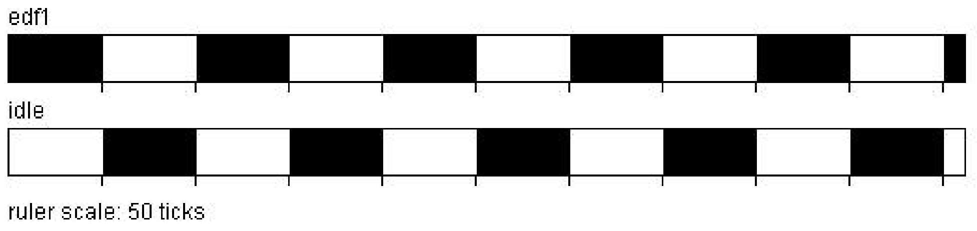

Figure 8: Schedule trace of scenario 1.

Scenario 2
In this scenario we will test multiple processes with different periods and calculation times. We have logged the processes that were running when the clock interrupt happened. Table 2 shows the processes that were created.

name	period	calculation time
edf1	50	10
edf2	100	20
edf3	50	5
edf4	100	10
CPU utilization: 60%		

Table 2: Processes started in scenario 2.

Listing 8 shows a trimmed version of a part of the log file.

Listing 8: A part of the log file of scenario 2.

```
598555 142198 edf1
<removed 8 entries from process edf1>
598564 142198 edf1
598565 142201 edf3
<removed 3 entries from process edf3>
598569 142201 edf3
598570 142202 edf4
<removed 8 entries from process edf4>
598579 142202 edf4
598580 142199 edf2
<removed 18 entries from process edf2>
598599 142199 edf2
598600 -4 idle
<removed 3 entries from process idle>
598604 -4 idle
598605 142198 edf1
<removed 8 entries from process edf1>
598614 142198 edf1
598615 142201 edf3
<removed 3 entries from process edf3>
598619 142201 edf3
598620 -4 idle
<removed 33 entries from process idle>
598654 -4 idle
598655 142198 edf1
<removed 8 entries from process edf1>
598664 142198 edf1
598665 142201 edf3
<removed 3 entries from process edf3>
598669 142201 edf3
598670 142202 edf4
<removed 8 entries from process edf4>
598679 142202 edf4
598680 142199 edf2
<removed 18 entries from process edf2>
598699 142199 edf2
598700 -4 idle
<removed remaining entries>
```

The first entry in the log is tick 598555 from process edf1. Because each log entry contains the running process while the clock interrupt happened, process edf1 has started on tick 598554. We have started the four real-time processes automatically using a shell script. The shell that executed the script had a higher priority then real-time processes to avoid delayed startup of the real-time processes.

Using the edftop application (see Listing 9 on page 49) we can see that processes edf2, edf3 and edf4 started one tick later than process edf1. Otherwise the TLNP (ticks left till next period) column of process edf1 would have the value 4, just like process edf3 which has the same period. We have put the results of the log analysis in four tables (table 3, 4, 5 and 6). For each of the four real-time processes we show the results of the first five periods in a table. Figure 9 on page 50 shows a visual representation of the log file. The results corresponds to our expectations.

period	period start	deadline	start	finished	runned time	missed deadline
1	598554	598604	598554	598564	10 ticks	no
2	598604	598654	598604	598614	10 ticks	no
3	598654	598704	598654	598664	10 ticks	no
4	598704	598754	598704	598714	10 ticks	no
5	598754	598804	598754	598764	10 ticks	no

Table 3: Statistics of process edf1 (period: 50 ticks, calculation time: 10 ticks)

period	period start	deadline	start	finished	runned time	missed deadline
1	598555	598655	598579	598599	20 ticks	no
2	598655	598755	598679	598699	20 ticks	no
3	598755	598855	598779	598799	20 ticks	no
4	598855	598955	598879	598899	20 ticks	no
5	598955	599055	598979	598999	20 ticks	no

Table 4: Statistics of process edf2 (period: 100 ticks, calculation time: 20 ticks)

period	period start	deadline	start	finished	runned time	missed deadline
1	598555	598605	598564	598569	5 ticks	no
2	598605	598655	598614	598619	5 ticks	no
3	598655	598705	598664	598669	5 ticks	no
4	598705	598755	598714	598719	5 ticks	no
5	598755	599805	598764	598769	5 ticks	no

Table 5: Statistics of process edf3 (period: 50 ticks, calculation time: 5 ticks)

period	period start	deadline	start	finished	runned time	missed deadline
1	598555	598655	598569	598579	10 ticks	no
2	598655	598755	598669	598679	10 ticks	no
3	598755	598855	598769	598779	10 ticks	no
4	598855	598955	598869	598879	10 ticks	no
5	598955	599055	598969	598979	10 ticks	no

Table 6: Statistics of process edf4 (period: 100 ticks, calculation time: 10 ticks)

Listing 9: Output of edftop in scenario 2.

```
load averages:  0.62, 0.60, 0.61
42 processes: 1 running, 41 sleeping
Mem: 173228K Free, 172728K Contiguous Free
CPU states:  64.46% user,   0.83% system,   0.00% kernel,  34.71% idle

PNR   ENDPT  ST PERIOD CTIME   PRD#   TLP TLNP      USED RESERVED   %U MD COMMAND
 42  142198 WNP     50    10    298     0    3      2980     2980  100  0 edf1
 43  142199 WNP    100    20    149     0   54      2980     2980  100  0 edf2
 45  142201 WNP     50     5    298     0    4      1490     1490  100  0 edf3
 46  142202 WNP    100    10    149     0   54      1490     1490  100  0 edf4
```

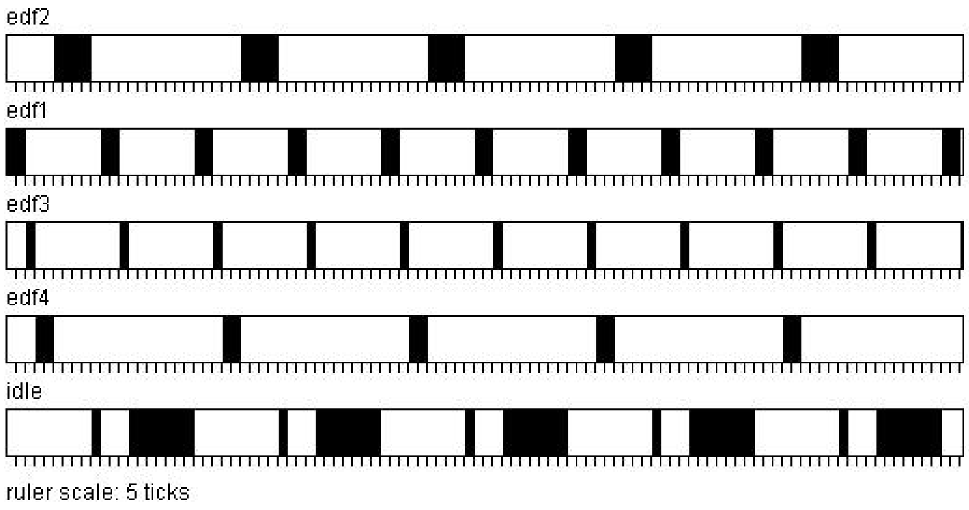

Figure 9: Schedule trace of scenario 2.

Scenario 3
In this scenario we will test the EDF scheduler under heavy load. We have created four real-time processes that reserved 98% of the CPU time. These processes are listed in table 7. The output of edftop (see listing 10) show that even after several thousands periods no deadlines are missed. This shows that our EDF scheduler works correctly under heavy load. Figure 10 shows a graphical representation of the first few periods.

name	**period**	**calculation time**
edf1	25	6
edf2	50	12
edf3	100	25
edf4	200	50
CPU utilization: 98%		

Table 7: Processes started in scenario 3.

Listing 10: Output of edftop in scenario 3.

```
load averages:  1.10, 1.08, 1.08
42 processes: 4 running, 38 sleeping
Mem: 175124K Free, 174744K Contiguous Free
CPU states:  99.17% user,   0.83% system,   0.00% kernel,   0.00% idle

PNR   ENDPT  ST PERIOD CTIME  PRD#  TLP TLNP     USED RESERVED  %U MD COMMAND
 66   71144 WNP     25     6  8061    0   12    48366    48366 100  0 edf1
 68   71146 WNP     50    12  4030    0   12    48360    48360 100  0 edf2
 70   71148 RUN    100    25  2015   25   62    50375    50375 100  0 edf3
 71   71149 RUN    200    50  1007    9   62    50350    50350 100  0 edf4
```

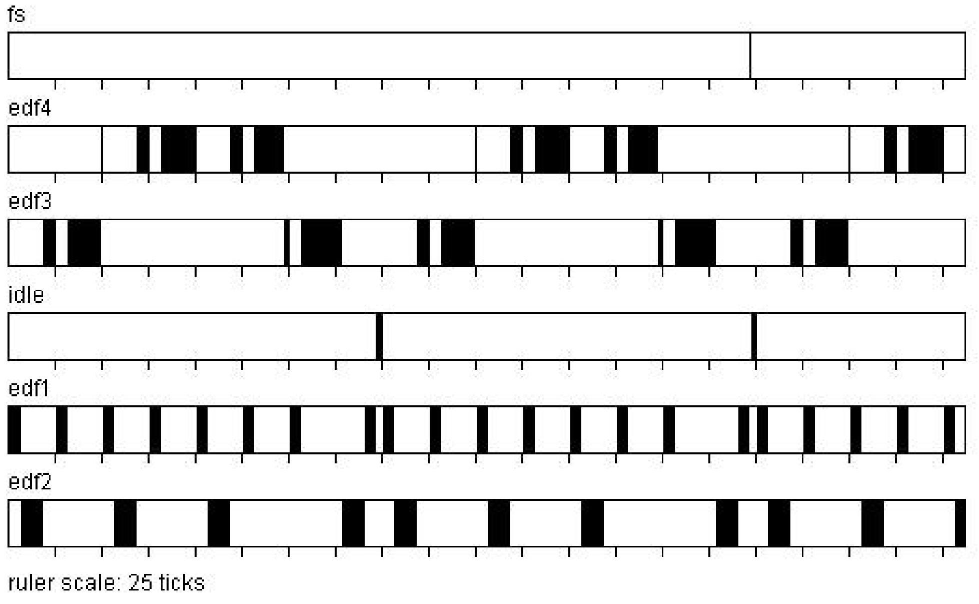

Figure 10: Schedule trace of scenario 3.

Scenario 4

Real-time processes can release their remaining reserved calculation time using the rt_nextperiod() API call. In this scenario we will test this functionality. We have created two real-time processes (listed in table 8). Both processes have the same period and calculation time. Process edf1 will use all calculation time (infinite loop). Process edf2 will not use all calculation time. It will release the reserved calculation time using the rt_nextperiod() API call and waits for the next period start.

Listing 11 shows the output of edftop. The USED and RESERVED columns show that process edf1 has used all reserved calculation time and process edf2 used 289 of the 800 reserved ticks. The %U column shows that process edf2 has used 36% of the reserved calculation time.

Listing 12 shows a part of the log file. The statistics for the first five periods of the two processes are extracted from the log file and are showed in table 9 and table 10. Figure 11 shows a visual representation of the log.

name	**period**	**calculation time**
edf1	100	25
edf2	100	25
CPU utilization: 50%		

Table 8: Processes started in scenario 4.

Listing 11: Output of edftop in scenario 4.

```
load averages:  0.34, 0.35, 0.36
37 processes: 1 running, 36 sleeping
Mem: 181792K Free, 181296K Contiguous Free
CPU states:  28.10% user,   0.83% system,   0.00% kernel,  71.07% idle

PNR   ENDPT  ST PERIOD CTIME  PRD#  TLP TLNP     USED RESERVED  %U MD COMMAND
 33 5082110 WNP    100    25    32    0   35      800      800 100  0 edf1
 34 4904416 WNP    100    25    32   16   35      289      800  36  0 edf2
```

Listing 12: A part of the log file of scenario 4.

```
555763 5082110 edf1
<removed 23 entries from process edf1>
555787 5082110 edf1
555788 4904416 edf2
<removed 7 entries from process edf2>
555796 4904416 edf2
555797 -4 idle
<removed 64 entries from process idle>
555862 -4 idle
555863 5082110 edf1
<removed 23 entries from process edf1>
555887 5082110 edf1
555888 4904416 edf2
<removed 7 entries from process edf2>
555896 4904416 edf2
555897 -4 idle
<removed 64 entries from process idle>
555962 -4 idle
555963 5082110 edf1
<removed 23 entries from process edf1>
555987 5082110 edf1
555988 4904416 edf2
<removed 7 entries from process edf2>
555996 4904416 edf2
555997 -4 idle
<removed remaining entries>
```

period	period start	deadline	start	finished	runned time	missed deadline
1	555762	555862	555762	555787	25 ticks	no
2	555862	555962	555862	555887	25 ticks	no
3	555962	556062	555962	555987	25 ticks	no
4	556062	556162	556062	556087	25 ticks	no
5	556162	556262	556162	556187	25 ticks	no

Table 9: Statistics of process edf1 (period: 100 ticks, calculation time: 25 ticks)

period	period start	deadline	start	finished	runned time	missed deadline
1	555762	555862	555787	555796	9 ticks	no
2	555862	555962	555887	555896	9 ticks	no
3	555962	556062	555987	555996	9 ticks	no
4	556062	556162	556087	556096	9 ticks	no
5	556162	556262	556187	556196	9 ticks	no

Table 10: Statistics of process edf2 (period: 100 ticks, calculation time: 25 ticks)

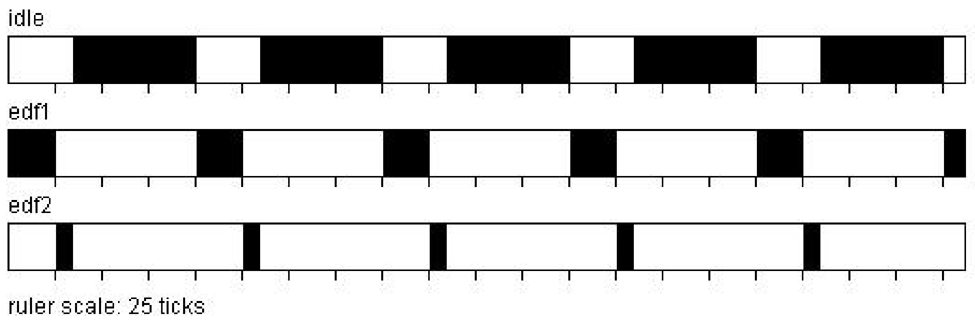

Figure 11: Schedule trace of scenario 4.

Scenario 5
In this scenario we have started two processes that reserved more calculation time than available. We have checked if the operating system noticed that deadlines are missed. The two processes started are listed in table 11. We used the edftop application to check if deadlines are missed. Listing 13 shows the output of edftop. It shows that both processes are in the 34rd period and have missed 17 deadlines.

Figure 12 shows a visual representation of the log. It is notable that the process that missed the deadline may run first in the next period. This happens because accounting for processes scheduled by EDF is done in a particular order and the period starts at the same time for both processes.

name	**period**	**calculation time**
edf1	100	50
edf2	100	60
CPU utilization: 110%		

Table 11: Processes started in scenario 5.

Listing 13: Output of edftop in scenario 5.

```
load averages:  2.75, 0.70, 0.24
37 processes: 5 running, 32 sleeping
Mem: 181856K Free, 175856K Contiguous Free
CPU states: 100.00% user,   0.00% system,   0.00% kernel,   0.00% idle

PNR   ENDPT  ST PERIOD CTIME  PRD#  TLP TLNP     USED RESERVED  %U MD COMMAND
 57   71135 RUN    100    50    34   50   77     1530     1700  90 17 edf1
 59   71137 RUN    100    60    34   37   77     1870     2040  92 17 edf2
```

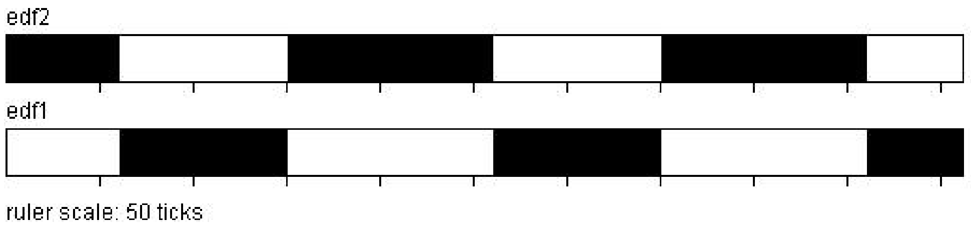

Figure 12: Schedule trace of scenario 5.

Scenario 6
In the previous scenarios we have used the kernel log functionality to log the running processes when a clock interrupt happens. In this scenario we will log all context switches. This shows that although real-time processes reserve calculation time, they can be preempted by processes with a higher priority (system processes) and might not get the calculation time they have reserved. We have started one real-time process with a period of 10 ticks and a calculation time of 5 ticks (50% CPU utilization). Figure 13 shows a visual representation of the log. The ruler is not linear and every stripe means the start of a new tick. It is notable that the clock task will run after each clock interrupt while it should only run if it has something to do.

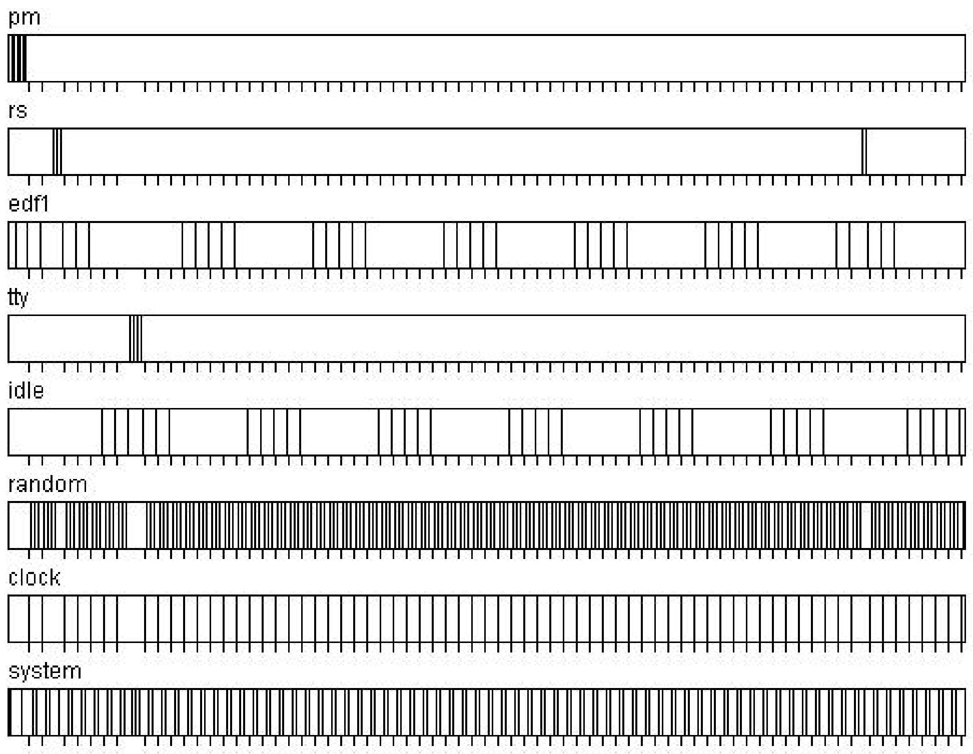

Figure 13: Schedule trace of scenario 6.

2.6.5 Problems

In this section we discuss problems that are not addressed and possible improvements.

Inaccuracy of billed run time
Real-time processes can be preempted by processes with a higher priority (usually system processes). This is a problem because processes are only billed when a clock interrupt happens. If a real-time process is preempted by an other process and this process finishes before the clock interrupt happens, the real-time process will be billed for processor time it did not use (it was used by the other process).

Consider the situation in figure 14. The real-time process is preempted twice and in both situations the other process finishes before the clock interrupt happens. The real-time process will be billed for using two ticks but it only got one tick. Increasing the tick rate will improve the accurary but is not a real solution because it will also increase the overhead (more clock interrupts).

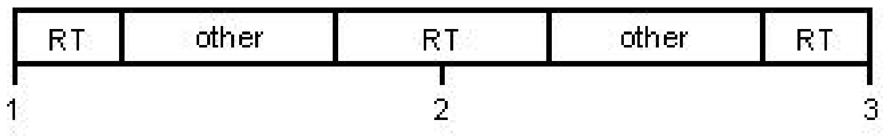

Figure 14: Preemption of a real-time process.

Clock task runs too often
Scenario 6 in section 2.6.4 shows that the clock task runs after each clock interrupt. This isn't supposed to happen. The clock task should only run when there is work to do:

- A timer has expired.
- A process used its full quantum (can't happen when a real-time process is running).
- The running process is real-time and missed the deadline.
- The running process is real-time and used all calculation time.
- A new period starts for one or more processes scheduled with EDF that are in the wait queue.

Efficiency can be improved
We have implemented EDF in a simple and easy to debug way. Unfortunately this is not the most efficient way. For example the *p_rt_nextperiod* field of the process structure holds the number of ticks till the next period starts. A disadvantage is that this field has to be updated for all real-time process on each clock interrupt. This can be implemented more efficient by saving the real time (system time) till the next period in this field. As the real time is updated on each clock interrupt you only have to compare the *p_rt_nextperiod* field with the real time. For the wait queue this means that only the head of this queue has to be compared (the queue is sorted on *p_rt_nextperiod).*

Clock drifting in virtual machines
We have tested our system only in a virtual machine. Because virtual machines work by time sharing host physical hardware, a virtual machine cannot exactly duplicate the timing behavior of a physical machine [9]. We have not noticed any problems due to this fact but they might exist and have influenced our test results.

3 Inter-process communication

One of the inter-process communication mechanisms that Minix 3 provides is message passing. Message passing is done in a synchronized way. If a process wants to send to a destination process that is not ready to receive the message, the sender will be blocked until the destination is ready. If a process wants to receive a message but there are no senders it will be blocked until a process sends a message. The only non-blocking way to communicate with other processes is through a notification. The sender does not have to wait till the receiver reads the notification. Notifications do have a disadvantage: the only information they contain is who has sent the notification. User processes are not allowed to communicate with eachother using message passing. Instead they may use other mechanisms such as piping. Message passing is used for user processes to communicate with system servers and for system servers and drivers to communicate with the kernel. The following API functions in Minix 3 are available:

- send()
- receive()
- _sendrec()
- notify()

3.1 Implementation in Minix 3

All of these functions are implemented as library functions written in assembly. They can be found in */usr/src/lib/i386/rts/_ipc.s.* These functions trap the kernel by generating a software interrupt. The trap will be caught and *sys_call()* will be called. The function *sys_call()* is implemented in */usr/src/kernel/proc.c.* The function *sys_call()* will do some checks and invoke the right function(s). We will now discuss the implementation of *send(), receive()* and *_sendrec().* We do not discuss the *notify()* function because we did not improved it. These functions are all implemented in */usr/src/kernel/proc.c.*

Endpoints
Before we discuss the API functions it is important to know what endpoints are in Minix 3. The endpoint of a process is generated when a process spawns. Endpoints are the only process identification that are usable through the whole system unlike for example Process ID's (PID's). Only the process manager is aware of PID's. Endpoints are used to distinguish a process in a process slot in the process table from an other process in the same slot. If for example a process is killed the process slot will become free and can be used again. Using process slot numbers it is impossible to distinguish the killed process from the new process in the same process slot. Endpoints however do have a relation with process slot number. It is possible to calculate the endpoint back to

a process number in the process tables of the kernel, process manager and file system. There are also some magic endpoint values such as ANY and NONE which stands for all process or no processes. Endpoints are used in the IPC API to address processes.

send() API call

If the *send()* library function was called, *sys_call()* does some specific checks (i.e if the process is allowed to send to the specified destination). After that it will call *mini_send(). mini_send()* first checks if the destination is blocked waiting for a message and if it does want to receive from the sender. The endpoint of processes from whom the destination wants to receive is saved in the *p_getfrom_e* of the process entry. If this endpoint is ANY or the same endpoint as the sender, the receiver is willing to receive the message. It will then copy the message from the sender to the destination. If the destination process is runnable it will be scheduled.

If the destination process is not blocked waiting for a message the sender will be descheduled and blocked. The SENDING bit in the *p_rts_flags* field of the sending process entry will be set indicating that this process is blocked while trying to send. The *p_sendto_e* field in the process entry will be set to the destination process. Finally the sending process will be added to the end of the list of processes whom want to send to the destination. This list is kept by the destination process. The *p_caller_q* field in the process entry of the destination process points to the first process of the list of processes whom want to send to this process.

receive() API call

If the *receive()* library function was called, *sys_call()* will invoke *mini_receive()* after doing some checks. *mini_receive()* first checks if the endpoint from whom the caller wants to receive is valid. After that it will check if the process wants to receive notifications. If it wants to, *mini_receive()* will check if there are notifications pending and the notification will be transformed into a message send back to the process. If there were no notifications available or the process didn't want to receive notifications (or wants to read them later) *mini_receive()* will check if there are processes that want to send a message to the caller process. The list of processes that want to send to the caller process is kept by the caller. The *p_caller_q* field in the process entry of the caller points to the first process in the list of processes wishing to send. These processes are linked together in the list using the *p_q_link* field in each process entry. It will loop through the list finding the first process from whom the caller wants to receive. This process will be removed from the list and the message will be copied from this process to the caller process.

If there are no processes in the list of processes from whom the caller wants to receive, or if the list is empty the caller will be blocked and descheduled. The endpoint from whom the caller wants to receive will be saved in the *p_getfrom_e* field of the process entry. The pointer to the message buffer will be saved in the *p_messbuf* field of the process entry. The RECEIVING bit in *p_rts_flags* field will be set to indicate that this process is blocked on receiving a message.

_sendrec() API call
If the *_sendrec()* library function is called *sys_call()* simply calls *mini_send()* and *mini_receive()* after eachother. Before doing that it will set the REPLY_PENDING bit in the *p_misc_flags* field of the process entry. This indicates that the process does not want to receive any notifications at this time.

3.2 Requirements

There is only one requirement regarding message passing: processes with a higher priority should be served before processes with a lower priority.

3.3 Design

The standard *send()* function will add the calling process to the end of the destinations list of processes wishing to send to the destination. We have modified this code to add processes in this list based on their priorities to improve real-time behaviour. In the case of non real-time processes or real-time processes scheduled using Earliest Deadline First it will only look at the Minix priorities. Rate-Monotonic scheduled processes share one Minix priority. If there are multiple processes scheduled using Rate-Monotonic we will also consider their Rate-Monotonic priorities.

3.4 Implementation

The *send()* is the only function that has been modified. Figure 14 shows the modified part. This part will add the sender to the queue of the receiver if the receiver is not waiting for a message. It will first loop through the queue (using a pointer pointer) until the next process in the list has a lower priority (higher value of the *p_priority* field). After it has found the right place in the queue it will add the process to the queue. If the sender is a process scheduled by the Rate-Monotonic (RM) scheduler it will also look at the RM priority. If the next process in the list is a process scheduled by RM and has a lower RM priority (higher value of the *p_rt_priority* field) it will break out of the while loop.

Listing 14: Modified part of send() function.

```
/* xpp is: struct proc **xpp */
xpp = &dst_ptr->p_caller_q;
while (*xpp != NIL_PROC &&
       (*xpp)->p_priority <= caller_ptr->p_priority) {

    /* If the process is scheduled by RM we
     * have to check the RM priority
     */
    if (is_rtp(caller_ptr) && rt_sched == SCHED_RM) {
        if (is_rtp(*xpp) && (*xpp)->p_rt_priority >
            caller_ptr->p_rt_priority) {
            break;
        }
    }

    /* get next in queue */
    xpp = &(*xpp)->p_q_link;
}

/* Add process to the destination's queue */
caller_ptr->p_q_link = *xpp;
*xpp = caller_ptr;
```

3.5 Testing

We have tested the modifications by printing the contents of the queue before adding the sender to the queue and after adding the sender to the queue. To enable printing this information we have made a macro in /usr/src/kernel/config.h called DEBUG_IPC. If this macro is set to 1 the debug information will be printed after the kernel is recompiled and the system is rebooted. Listing 15 shows an example of the debug information. The format of the processes is: [priority:process name].

Listing 15: IPC debug information log.

```
sep 14 22:31:50 192 kernel:  ---
Sep 14 22:31:50 192 kernel: IPC: recv Q bef. pm: [2:mem]
Sep 14 22:31:50 192 kernel: IPC: recv Q aft. pm: [2:mem] [3:ds]
Sep 14 22:31:50 192 kernel:  ---
Sep 14 22:31:50 192 kernel: IPC: recv Q bef. pm: [2:mem] [3:ds]
Sep 14 22:31:50 192 kernel: IPC: recv Q aft. pm: [2:mem] [3:ds] [3:rs]
Sep 14 22:31:50 192 kernel:  ---
Sep 14 22:31:50 192 kernel: IPC: recv Q bef. fs: [14:sh]
Sep 14 22:31:50 192 kernel: IPC: recv Q aft. fs: [14:sh] [14:syslogd]
Sep 14 22:31:50 192 kernel:  ---
Sep 14 22:31:50 192 kernel: IPC: recv Q bef. fs: [12:sh]
Sep 14 22:31:50 192 kernel: IPC: recv Q aft. fs: [3:pm] [12:sh]
Sep 14 22:31:50 192 kernel:  ---
Sep 14 22:31:50 192 kernel: IPC: recv Q bef. fs: [14:cron]
Sep 14 22:31:50 192 kernel: IPC: recv Q aft. fs: [14:cron] [14:syslogd]
Sep 14 22:31:50 192 kernel:  ---
Sep 14 22:31:50 192 kernel: IPC: recv Q bef. fs: [14:cron] [14:syslogd]
Sep 14 22:31:50 192 kernel: IPC: recv Q aft. fs: [3:pm] [14:cron] [14:syslogd]
```

3.6 Problems

There were no significant problems.

4 Semaphore server

We have added semaphores to Minix 3. We will discuss our implementation in detail in this section.

4.1 Requirements

We have the following basic requirements for the semaphore server:

- Support for binary semaphores.
- Support for mutexes.
- Support for counting semaphores.

4.2 Design

Semaphores are implemented as a separate user space server. This server should have a fixed endpoint otherwise the user applications cannot find the semaphore server. Only processes in the boot image have fixed endpoints. The semaphore server is part of the boot image. The number of semaphores supported is static. The server has an array of semaphore structures (*struct sem*). For each process that blocks on a semaphore a *sem_proc* structure is dynamically created to hold this process. This structure can be seen as a list element and can point to the next structure in the list. Each semaphore structure has a pointer to a *sem_proc* structure (which initially contains the value NULL).

Minix 3 uses segmentation for memory protection. This has the disadvantage that the space for malloc and stack cannot change at run time. This is the major reason that the number of semaphores supported is static. Memory allocation for the *sem_proc* structures are done dynamically. This is not a problem as the number of process slots in the process table is also limited. One can predict how much memory for malloc is needed. This is the number of processes that possible uses the semaphore server multiplied by the size of the *sem_proc* structure. In this way we can guarantee that this number of semaphores can be used. If the number of semaphores supported would be dynamical we cannot guarantee any number of semaphores, this is unacceptable for a real-time system.

Listing 16: Declaration of struct sem and struct sem_proc.

```
/* semaphore structure */
struct sem {
        int sem_type;
        int value;
        int size;
        int used;
        int owner;
        int user;
        struct sem_proc * listp;

};

/* used to save processes that are blocked
 * on a semaphore
 */
struct sem_proc {
        int endpoint;
        int prio;
        struct sem_proc * next;
};
```

4.3 Implementation

We will first discuss how to add a system server to the boot image. After that we will discuss the semaphore server internals.

Adding a system server to the boot image

The following steps can be taken to add a system server to the boot image:

- Create a new directory in */usr/src/servers/* for the new server.
- Implement the new server in this directory (create a new Makefile in this directory to build it).
- Modify */usr/src/servers/Makefile* to add this server.
- Renumber the process numbers in */usr/src/include/minix/com.h.* Add a new definition for the server you want to add and give it a number. It is recommended to add it before init.
- Add the server to the *boot_image* structure in */usr/src/kernel/table.c.* You might also need to create a new mask for the allowed kernel calls.
- Add the server to the */usr/src/tools/Makefile* file, use the same ordering as in */usr/src/kernel/table.c.*

Implementation of the semaphore server
The implementation of the semaphore server contains the following files:

- */usr/src/servers/ss/ss.c* contains the actual implementation.
- */usr/src/servers/ss/proto.h* contains the function prototypes.
- */usr/src/servers/ss/ss.h* contains declarations of variables and constants.
- */usr/src/include/minix/sem.h* contains return codes, message field names and others.
- */usr/src/lib/sem/* contains the API library files.

The main routine and initialization
The main routine first calls the *init()* function. This function will make sure that the *used* member of each semaphore structure is set to 0. Furthermore it will set the pointer to the first blocked process in the list to NULL. After initialization it will enter an infinite loop. In this loop it will do a call to *receive()* to get work. After receiving a message it will analyze the *m_type* field of the message to get the system call number. Using a switch statement the right function to handle the call will be invoked. After that it will try to send a message back if needed (i.e the calling process does not block on a semaphore).

Create a new semaphore
If the main routine receives a system call request with the number defined in SEM_CREATE it will invoke *sem_create()* with the type, value and size as parameters. Not all of these parameters have to be valid in any case. For example if the type is set to mutex, the value and size parameters are not used and may have any value. The first thing *sem_create()* does is checking if the type parameter is valid. It is valid if it has the value defined in SEM_MUTEX, SEM_BINARY or SEM_COUNTING. After that it will check if there is a semaphore free using the used global variable which contains the number of semaphores in use. After that is will do some type specific checks. For binary semaphores it will check if the value parameter is 0 or 1. For counting semaphores it will check if the value parameter is a positive number. It will also check if the value parameter does not have a larger value than the size parameter and if the size parameter is equal or larger than 1.

Now all checks are done we can create a new semaphore by taking an unused semaphore slot. We first try to find an unused semaphore slot by looping through the array of semaphores. If we have found one we set the *used* member of the semaphore structure to 1. We set the *sem_type* member to the type we want to create and increase the global *used* variable. For binary semaphores we set the *value* member and for counting semaphores the *value* and *size* members. After that we can return a handler to the calling process. This handler can be the index in the array of semaphores but there is a good reason not doing that. The array starts with index 0. An uninitialized semaphore handler will also have the value of 0. We should prevent that unused semaphore handlers can be used at all. Instead we return a virtual semaphore handler. In our case we just return the indice incremented by 1. All other semaphore related functions internally use real semaphore handlers (the indices in the semaphore array). Translation from virtual handler to real handler for these functions is done by the main routine. The handler is not returned on a normal way. As we have to return both a return code and the handler, the handler is returned as a field in the message. (The return code however is also returned in a message field: *m_type*).

Take a semaphore
If the main routine receives a system call request with the number defined in SEM_TAKE it will invoke *sem_take()* with the real semaphore handler and the block type as parameters. Processes can choose to block (WAIT_FOREVER) on a semaphore or not (NO_WAIT). If they choose not to block on a semaphore it will receive an error code if the semaphore was not available.

This function first checks if the semaphore handler is valid using the VALID_SEM() macro. This macro checks if the handler is in the range of used handlers and if that is the case, if the semaphore is currently in use. After that it will try to get the process structure of the calling process using the *sys_getproc()* library function. We need the process structure to get the priority of the process. Processes that are blocked will be added to the block list of the semaphore which is sorted on priority. After retrieving the process structure it will check if the semaphore is free. If this is the case it will change the value member of the semaphore structure and return OK. The main routine will send a message back to the calling process. If however the semaphore is not free, it will check if the process wants to wait or not. If it does not want to wait an error code will be returned and send back by the main routine. If it does want to wait a new *sem_proc* structure will be created containing the caller process' information. This structure will be added to the list of blocked processes on this semaphore.

Give back a semaphore
If the main routine receives a system call request with the number defined in SEM_GIVE it will invoke *sem_give()* with the real semaphore handler as parameter. This function first checks if the semaphore handler is valid using the SEM_VALID() macro. After that it will check if the semaphore isn't already free. If the semaphore is of mutex type it will check if the process that gives back the semaphore is the same as the process that took the semaphore. The value member of the semaphore structure will be changed accordingly. Finally it will check if there are processes blocked on this semaphore. If there are any it will try to unblock the first process. Unblocking may fail because the process may have been killed for example. It will try to unblock the next process in the list if the first one fails. This repeats until a process is succesfully unblocked or the list is empty. For each process removed from the list the memory will be freed using the

free() function. The values of the members of the semaphore structure will be changed to the unblocked process.

Delete a semaphore
If the main routine receives a system call request with the number defined in SEM_DELETE it will invoke *sem_delete()* with the real semaphore handler as parameter. This function first checks if the semaphore handler is valid using the SEM_VALID() macro. After that it will check if the semaphore is not taken. Binary and mutex semaphores should have the value of 1 and counting semaphores should the value equal to the size. If this is the case the semaphore can be deleted. The semaphore slot is released by setting the *used* member of the semaphore structure to 0. The global *used* variable holding the number of semaphores in use will be decremented.

Flushing a semaphore
Inspired by VxWorks [6] we have added a feature to flush a binary semaphore. This means that all processes blocked on the binary semaphore will be unblocked. This can be used to synchronize processes. The system call number is defined in SEM_FLUSH. This function first checks if the semaphore handler is valid using the SEM_VALID() macro. After that it will check if the type of the semaphore is binary. We only allow flushing for binary semaphores. After that it will remove and unblock all processes on the list of processes blocked on this semaphore. The value of the semaphore will not be changed.

Getting the value of a semaphore
We offer a function to get the value of a semaphore without doing a take. The system call number is saved in SEM_VALUE. This function should only be used for debugging and testing purposes.

Removing semaphores of exited processes
The process that creates a semaphore is seen as the owner of a semaphore. Processes that exit should clean up first. If they don't do that, for example if the process is killed semaphores owned by that process will be automatically deleted. For each process that exits the process manager will do a system call (SEM_PROCEXIT) to the semaphore server with the endpoint of the proces that has exited.

4.4 Testing

Dining Philosophers
We have implemented the Dining Philosophers problem using our semaphore API. You can find it at */usr/src/test/rtminix3/semaphores/dining_philosophers/*. Build the application by executing `./build` in that directory. Run the application by executing `./dp`

in that directory.

Batch tests
The dining philopophers application only tests a small part of the features of the semaphore server. To test all features and scenario's we have created a batch test. You can find the batch test here: */usr/src/test/rtminix3/semaphores/dining_philosophers/.* Execute '`make all`' from that directory to build the tests. Execute `./test.sh` to run the batch tests.

The shell script *test.sh* executes these tests one after another. After any test that exits the exit code will be checked. There are three possible exit codes:

- Test succeeded.
- Test failed but it does not have an influence on the other tests.
- Test failed and might have an influence on the other tests.

In the last case the batch test will be stopped immediately. Table 12 shows the implemented tests.

Test 1	Test the creation of a mutex semaphore under normal conditions.
Test 2	Test the creation of binary semaphore under normal conditions.
Test 3	Test the creation of counting semaphores under normal conditions.
Test 4	Test the creation of a semaphore with invalid type.
Test 5	Test the creation of a binary semaphore with invalid value parameters.
Test 6	Test the creation of a counting semaphore with invalid value parameter.
Test 7	Test the creation of a counting semaphore with invalid size parameter.
Test 8	Test sem_take() with invalid semaphore handler.
Test 9	Test sem_take() with mutex under normal conditions with NO_WAIT.
Test 10	Test sem_take() with binary semaphore under normal conditions with NO_WAIT.
Test 11	Test non-blocking sem_take() with counting semaphore under normal conditions.
Test 12	Test sem_give() with mutex under normal conditions.
Test 13	Test sem_give() with binary semaphore under normal conditions.
Test 14	Test sem_give() with counting semaphore under normal conditions.
Test 15	Test sem_give() with invalid semaphore handler.
Test 16	Test sem_give() with mutex while semaphore is not taken.
Test 17	Test sem_give() with binary semaphore while semaphore is not taken.
Test 18	Test sem_give() with counting semaphore while semaphore is not taken.
Test 19	Test sem_give() of taken mutex with wrong user.

Table 12: Implemented batch tests for the semaphore server.

4.5 Problems

In this section we discuss problems that are not addressed and possible improvements.

Extending the batch test
Due to time contraints we were not able to create enough tests to cover all parts of the semaphore server.

Deleting semaphores
We should only delete a semaphore if we are sure that no process is using it any more. In the set of processes that use a particular semaphore, the creator of the semaphore is seen as the owner of the semaphore. We currently assume that the owner of the semaphore is the last process in the set of processes that exits. This might not be the case in real scenario's. The semaphore server is not aware which processes are using a particular semaphore. A possible solution to this is that each process that wants to use a particular semaphore should register itself to the semaphore server. The semaphore server then can keep a list of processes using this semaphore and if all process in this list have exited it is save to delete the semaphore.

Priority inversion
The process scheduler in the kernel is not aware of the existence of semaphores because it is implemented in a user space server. This introduces priority inversion problems that are not solved yet.

priority of rate-monotonic scheduled processes
The semaphore server is not aware of which real-time scheduler is active. Processes scheduled by RM share one Minix priority. The RM priority of these processes is not considered when sorting the queue of processes blocked on a semaphore.

5 Conclusions

It is fairly easy to turn Minix 3 into a soft real-time operating system. We have succesfully added two real-time schedulers, prioritized message passing and semaphores. However there is still a lot of work to do to test our extensions, improve them and enhance the real-time behaviour in other parts of the operating system. Our project has opened the door for further development of a more real-time Minix 3.

References

[1] MontaVista Software Inc. *MontaVista Linux 6 datasheet*. [online] http://www.mvista.com/download/MontaVista-Linux-6-datasheet.pdf.

[2] Pablo J. Rogina and Gabriel Wainer. *New Real-Time Extensions to the MINIX operating system*, August 1999. [online] http://www.dc.uba.ar/people/proyinv/cso/rt-minix/RT-minix.doc.

[3] QNX Software Systems Ltd. *Microkernel RTOSs Simplify Software Testability*. [online] http://www.qnx.com/download/download/11725/microkernel_rtos_simplify_cots_testability.pdf.

[4] David van Moolenbroek. *Multimedia support for MINIX 3*, September 2007. [online] http://www.minix3.org/doc/moolenbroek_thesis.pdf.

[5] Pablo Andrés Pessolani. *MINIX4RT: A Real-Time Operating System Based on MINIX*, February 2006. [online] http://sites.google.com/site/minix4rt/Home/MINIX4RT.pdf.

[6] Wind River. *VxWorks semFlush() documentation*. [online] http://www.slac.stanford.edu/exp/glast/flight/sw/vxdocs/vxworks/ref/semLib.html#semFlush.

[7] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition) (Prentice Hall Software Series)*. Prentice Hall, January 2006.

[8] YS Vivek, L Bhuvan, G Ashrith, and DSVN Aditya. *Realtime Minix*, 2008. [online] http://realtimeminix.sourceforge.net.

[9] VMWare. *Timekeeping in VMware Virtual Machines*, 2008. [online] http://www.vmware.com/pdf/vmware_timekeeping.pdf.

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
EDF	Earliest Deadline First
I/O	Input/Ouput
IPC	Inter-Process Communication
OS	Operating System
PID	Process Identifier
POSIX	Portable Operating System Interface for UNIX
RM	Rate-Monotonic
RT	Real-Time
RTOS	Real-time Operating System

Glossary

Binary Semaphore
A type of semaphore with just two states.

Blocked
The state of a process in which it can not run because it is waiting for a particular event or needs service from the operating system.

Context Switch
The process of switching from one task to another in a multitasking operating system. A context switch involves saving the context of the running task and restoring the previously saved context of the other.

Counting Semaphore
A type of semaphore with more than two states. A counting semaphore is typically used to track multiple resources of the same type. An attempt to take a counting semaphore is blocked only if all of the available resources are in use.

Deadline
A point in time at which a computation must be completed.

Earliest Deadline First Scheduling
Scheduling policy that gives a processor to the process with the closest deadline.

Embedded System
A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function.

Inter-process Communication
A technique to exchange data between processes.

Kernel
The lowest part of an operating system that deals with process scheduling, memory management and process scheduling.

Kernel Call
A request by a process for a service performed by the kernel. In Minix kernel calls can only be made by drivers and system servers.

Mutex
An operating system data structure used by tasks to ensure exclusive access to a particular resource. Short for mutual exclusion.

Operating System
An operating system is the software that controls a computer and computer software and governs how they work together.

Preemptive
A process is preemptive when it can be interrupted without its cooperation.

Priority of a Process
Measurement of importance of a process.

Quantum
An amount of time a process may run.

Rate-Monotonic Scheduling
Is an optimal preemptive static-priority scheduling algorithm used in Real-Time operating systems.

Ready
The state of a process in which it can run but is not running.

Real-Time System
A system from which the correctness depends on the timeliness in which computations are finished..

Round-Robin Scheduling
A scheduling algorithm which assigns the CPU to processes in a circular order, handling processes without priority.

Scheduler
A piece of software that decides which process may run.

Scheduling
The way processes are assigned to run on the CPU.

Soft Real-time System
A real-time system which tollerates missing of deadlines and may continue with decreased service quality.

System Call
A request by an user process for a service performed by the operating system. In Minix system calls are handled by system servers.

System Server
A special process in Minix that is part of the operating system (but not the kernel).

Task
A task is a special process in Minix that is part of the kernel.

Trap
An interrupt generated by executing a special instruction.

User Space
Memory space that does not belong to the kernel.

Appendices

A Installation

RTMINIX3 can be installed in several ways. It is possible to install it from an installation CD. Furthermore it is possible to rebuild a standard Minix 3.1.2a installation to RTMINIX3 by patching or copying the source code. We also provide a pre-installed VMware virtual machine image.

Using the installation CD

Installation of RTMINIX3 is exactly the same as installing Minix 3. You can use the installation manual of Minix 3 to guide you through the installation.

Patching and rebuilding a Minix 3.1.2a installation

1. Install Minix 3.1.2a (if not already done).
2. Install GNU Patch. GNU Patch is not available in Packman. Either build it from source or install the binary by transferring it to your system (e.g downloading using urlget or FTP).
3. Transfer the patch file to your computer (e.g. using FTP).
4. Create a backup of the source directory:

   ```
   cp -r /usr/src/ /usr/src_backup/
   ```

5. Patch the source directory:

   ```
   /usr/gnu/bin/patch -p0 < /path/to/rtminix3.patch
   ```

6. Rebuild the system:

   ```
   cd /usr/src/tools && make fresh install
   ```

7. Reboot:

   ```
   reboot
   ```

8. Rebuild commands:

   ```
   cd /usr/src/commands && make all install
   ```

Copying the source code and rebuilding a Minix 3.1.2a installation

1. Install Minix 3.1.2a (if not already done).
2. Create a backup of the source directory:

   ```
   mv /usr/src/ /usr/src_backup/
   ```

3. Transfer the source code package (rtminix3.tar.bz2) to your computer (e.g. using FTP).
4. unzip the package to rtminix3.tar:

   ```
   bunzip2 rtminix3.tar.bz2
   ```

5. Extract the tarbal file to /usr/src/:

   ```
   tar xvf rtminix3.tar
   ```

6. Rebuild the system:

   ```
   cd /usr/src/tools && make fresh install
   ```

7. Reboot:

   ```
   reboot
   ```

8. Rebuild commands:

   ```
   cd /usr/src/commands && make all install
   ```

Using the VMware virtual machine image

1. Unzip the file containing the virtual machine image (RTMINIX3_vmware.zip).
2. Follow the instructions of your VMware software to add an existing virtual machine.
3. If asked for the vmx config file, provide the path to RTMINIX3.vmx

B Application Programming Interface

In this appendix we will describe the application programming interface that can be used by the real-time application programmer. The libraries are compatible with the Amsterdam Compiler Kit (ACK) compiler. Support for GCC is not tested.

B.1 Real-time library

This library contains all the functions related to the real-time functionality. Applications using this library should include the minix/rt.h header:

```
#include <minix/rt.h>
```

Applications using this library should be linked with librt using the compilers -l flag. An example:

```
cc -o myapp myapp.c -lrt
```

Function: int rt_set_sched_edf()
Sets the real-time scheduler to Earliest Deadline First. The real-time scheduler can only be set if there are no real-time processes running.

Arguments:
none

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EPERM	Permission denied. One or more real-time processes are running.

Function: int rt_set_sched_rm(prio_policy)
Sets the real-time scheduler to Rate-Monotonic. A unique Rate-Monotonic priority can be enforced with the prio_policy argument. The real-time scheduler can only be set if there are no real-time processes running.

Arguments:

int prio_policy	If set to PRIO_UNIQUE, enforce unique RM priorities. If set to PRIO_NOT_UNIQUE, do not enforce unique RM priorities.

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EPERM	Permission denied. One or more real-time processes are running.
EINVAL	Invalid prio_policy parameter. Must be PRIO_UNIQUE or PRIO_NOT_UNIQUE.

Function: int rt_set_edf(period, calctime)
Transforms a normal process into a real-time process scheduled by EDF. The real-time scheduler must be set to EDF prior to calling this function.

Arguments:

int period	period of the process in ticks.
int calctime	calculation time reserved in each period in ticks.

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EPERM	Permission denied. No real-time scheduler set or process is already real-time.
EINVAL	Invalid argument. The Arguments period and calctime must be a negative number. Argument calctime may not be larger than period.
EGENERIC	A fatal error occurred.

Function: int rt_set_rm(priority)
Transforms a normal process into a real-time process scheduled by RM. The real-time scheduler must be set to RM prior to calling this function.

Arguments:

int priority	The static (RM) priority of the process.

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EPERM	Permission denied. No real-time scheduler set or process is already real-time.
EINVAL	Invalid argument. Unique priority is enforced and the specified priority is in use.

Function: int rt_nextperiod()
Release remaining reserved calculation time and wait for the next period start. May only be used by processes scheduled by EDF.

Arguments:
none

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EPERM	Permission denied. Real-time scheduler is not set to EDF or process is not real-time.

Function: int rt_set_sched_bridge(state)
Set the scheduling bridge state. This state decides if processes with a higher priority than real-time processes can get a lower priority than real-time processes. State needs to be set to 1 before shutting down otherwise the system will not properly shutdown.

Arguments:

int state	If set to 0 processes with a higher priority than real-time processes will always have a higher priority than real-time processes. If set to 1 processes with a higher priority than real-time processes can get a lower priority than real-time processes.

Return value:
Returns 0.

B.2 Semaphore library

This library contains all the functions related to semaphores. Applications using this library should include the minix/sem.h header:

```
#include <minix/sem.h>
```

Applications using this library should be linked with libsem using the compilers -l flag. An example:

```
cc -o myapp myapp.c -lsem
```

The library functions do not make use of the errno variable but will directly return the error code.

Function: int sem_m_create(handler)
Creates a new mutex semaphore.

Arguments:

sem_t *handler	pointer to sem_t structure used as handler.

Return value:

OK	Operation succeeded. The specified handler contains the handler to the newly created mutex.
SEM_NO_AVAILABLE	All semaphore slots are in use, can't create new semaphore.

Function: int sem_b_create(handler, value)
Creates a new binary semaphore.

Arguments:

sem_t *handler	pointer to sem_t structure used as handler.
int value	Initial value (0 or 1).

Return value:

OK	Operation succeeded. The specified handler contains the handler to the newly created binary semaphore.
SEM_NO_AVAILABLE	All semaphore slots are in use, can't create new semaphore.
SEM_INVALID_VALUE	Invalid value specified.

Function: int sem_c_create(handler, value, size)
Creates a new binary semaphore.

Arguments:

sem_t *handler	pointer to sem_t structure used as handler.
int value	Initial value, must be smaller or equal to size.
int size	Size of the counting semaphore.

Return value:

OK	Operation succeeded. The specified handler contains the handler to the newly created counting semaphore.
SEM_NO_AVAILABLE	All semaphore slots are in use, can't create new semaphore.
SEM_INVALID_VALUE	Invalid value specified. Value was negative or larger than size.
SEM_INVALID_SIZE	Invalid size specified. Size was negative.

Function: int sem_take(handler, block)
Take a semaphore.

Arguments:

sem_t handler	Semaphore handler (not a pointer).
int block	If set to SEM_WAIT_FOREVER block until it is available. If set to NO_WAIT and the semaphore is not available, it will return immediately with SEM_IN_USE as return value.

Return value:

OK	Operation succeeded. Took semaphore.
SEM_INVALID_HANDLER	Invalid semaphore handler specified.
SEM_IN_USE	Semaphore is not available and the caller does not want to block.
SEM_INVALID_BLOCK	Invalid block argument speficied. Must be WAIT_FOREVER or NO_WAIT.

Function: int sem_give(handler)
Release a semaphore.

Arguments:

sem_t handler	Semaphore handler (not a pointer).

Return value:

OK	Operation succeeded. Released semaphore.
SEM_INVALID_HANDLER	Invalid semaphore handler specified.
SEM_ALREADY_FREE	Can't release semaphore because it is already released.
SEM_INVALID_MUTEX_GIVE	Only the process that took a mutex can release it.

Function: int sem_flush(handler)
Flush a binary semaphore. All processes blocked on the semaphore will be unblocked. Can be used for broadcast synchronization. It does not change the state of the semaphore.

Arguments:

sem_t handler	Semaphore handler (not a pointer).

Return value:

OK	Operation succeeded. Unblocked all processes there were previous blocked on this semaphore.
SEM_INVALID_HANDLER	Invalid semaphore handler specified.
SEM_INVALID_TYPE	Flushing is only allowed for binary semaphores.

Function: int sem_delete(handler)
Delete a semaphore. To delete a semaphore it must be released (or full if it is a counting semaphore).

Arguments:

sem_t handler	Semaphore handler (not a pointer).

Return value:

OK	Operation succeeded. Deleted the semaphore.
SEM_INVALID_HANDLER	Invalid semaphore handler specified.
SEM_IN_USE	Semaphore was not released.

Function: int sem_value(handler, value)
Get the value of a semaphore without taking it. Can be used for debugging purposes.

Arguments:

sem_t handler	Semaphore handler (not a pointer).
int *value	Pointer to variable to store the value.

Return value:

OK	Operation succeeded. Value is stored in the specified value argument.
SEM_INVALID_HANDLER	Invalid semaphore handler specified.

B.3 Kernel logger library

This library contains all the functions related to kernel logging. Applications using this library should include the minix/klog.h header:

```
#include <minix/klog.h>
```

Applications using this library should be linked with libklog using the compilers -l flag. An example:

```
cc -o myapp myapp.c -lklog
```

Function: int klog_set(type)
Starts kernel logging. Logging will automatically stop if the buffer is full.

Arguments:

int type	If it is set to KLOG_CONTEXTSWITCH it will log the next running process on every context switch. If it is set to KLOG_CLOCKINT it will log the running process when a clock interrupt happens.

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EINVAL	Invalid argument. Type must be KLOG_CONTEXTSWITCH or KLOG_CLOCKINT.

Function: int klog_copy(data)
Copies the log from the kernel to a user process.

Arguments:

struct klog_entry *data	Pointer to the first entry of an array with klog_entry structures. This array must have the same size as the array in the kernel and has the size defined in KLOG_SIZE.

Return value:
Returns 0 if the operation succeeded.
Returns -1 if the operation failed. An error code will be saved in the errno variable.

Error codes:

EFAULT	A fatal error occurred.

B.4 Standard applications

We have made a few applications that can be used directly in the command line interface or in a shell script.

rt_set_sched_edf
Sets the real-time scheduler to Earliest Deadline First.

rt_set_sched_rm_pnu
Sets the real-time scheduler to Rate-Monotonic. Unique RM priorities are not enforced.

rt_set_sched_rm_pu
Sets the real-time scheduler to Rate-Monotonic. Unique RM priorities are enforced.

klog_set_ci
Starts kernel logging and logs every running process when a clock interrupt happens until the buffer is full.

klog_set_cs
Starts kernel logging and logs every next running process on a context switch until the buffer is full.

klog_copy
Copies the kernel log and prints it to the console in the following format:

```
<time> <process endpoint> <process name>
```

To save the kernel log to a file use the following command:

```
klog_copy > filename.txt
```

C List of Modifications

Include files

/usr/src/include/lib.h (modified)
: Added SS macro which contains the endpoint of the semaphore server.

/usr/src/include/minix/callnr.h (modified)
: Added system call numbers.

/usr/src/include/minix/com.h (modified)
: Added kernel call numbers. Increased number of kernel calls (NR_SYS_CALLS macro). Added SS_PROC_NR macro which contains the endpoint of the semaphore server.)

/usr/src/include/minix/klog.h (created)
: Macro's and library (libklog) function prototypes for kernel logging.

/usr/src/include/minix/rt.h (created)
: Macro's and library (librt) function prototypes for real-time functionality.

/usr/src/include/minix/sem.h (created)
: Macro's and library (libsem) function prototypes for semaphores.

/usr/src/include/minix/syslib.h (modified)
: Added function prototypes for the added system library functions.

Kernel

/usr/src/kernel/.depend (modified, generated)
: Added dependencies for kernel source files.

/usr/src/kernel/clock.c (modified)
: Modified clock_handler() function and do_clocktick() function for accounting of real-time processes.

/usr/src/kernel/config.h (modified)
Added macro's to exclude added kernel calls. Added EDF_PRINT_MD macro to enable printing to the console if a process scheduled by EDF missed the deadline. Added DEBUG_IPC macro to enable printing debug information related to prioritized IPC.

/usr/src/kernel/proc.h (modified)
Added global kernel variables for scheduling and kernel logging. Added fields to the declaration of struct proc for real-time scheduling and statistics gathering. Added is_rtp() macro for checking if a process is real-time. Added RT_Q macro to hold the real-time queue number. Modified USER_Q macro to make place for the real-time queue. Added NEXTPERIOD macro that holds the flag to indicate that a process scheduled by EDF is blocked waiting for the next period.

/usr/src/kernel/proc.c (modified)
Added inclusion of <minix/rt.h> header. Modified enqueue() funcion to support RM and EDF. Added edf_sched() function that picks the next runnable EDF process. Added show_rt_data() function that prints RT scheduler debug information to the console. Modified mini_send() to support EDF and implement prioritized IPC. Modified mini_receive() function to support EDF. Modified sched() and balance_queues() functions to implement the scheduling bridge. Modified pick_proc() function to be able to log the next running process on every context switch.

/usr/src/kernel/proto.h (modified)
Added function prototypes for edf_sched() and show_rt_data() functions.

/usr/src/kernel/table.c (modified)
Added semaphore server to struct boot_image image[] array. Added SS_C macro which holds the allowed kernel calls for the semaphore server. Modified USR_M macro, which holds the processes to whom user processes may send messages, to allow sending messages to the semaphore server.

Kernel (kernel calls)

/usr/src/kernel/system.c (modified)
Modified initialize() function to map added kernel calls.

/usr/src/kernel/system.h (modified)
Added function prototypes for added kernel calls.

/usr/src/kernel/system/Makefile (modified)
Includes added kernel calls in the build process.

/usr/src/kernel/system/.depend (modified, generated)
Added dependencies for added and modified kernel calls.

/usr/src/kernel/system/do_exit.c (modified)
Added inclusion of <minix/rt.h> header. Added additional cleanup for exited real-time processes.

/usr/src/kernel/system/do_klog_copy.c (created)
Implementation of the SYS_KLOG_COPY kernel call.

/usr/src/kernel/system/do_klog_set.c (created)
Implementation of the SYS_KLOG_SET kernel call.

/srd/src/kernel/system/do_nice.c (modified)
Modified SYS_NICE kernel call to avoid non real-time processes getting the priority reserved for real-time processes.

/usr/src/kernel/system/do_rt_nextperiod.c (created)
Implementation of the SYS_RT_NEXTPERIOD kernel call.

/usr/src/kernel/system/do_rt_set.c (created)
Implementation of the SYS_RT_SET kernel call.

/usr/src/kernel/system/do_rt_set_sched.c (created)
Implementation of the SYS_RT_SET_SCHED kernel call.

/usr/src/kernel/system/do_rt_set_sched_bridge.c (created)
Implementation of the SYS_RT_SCHED_BRIDGE kernel call.

/usr/src/kernel/system/do_rt_show_data.c (created)
Implementation of the SYS_RT_SHOW_DATA kernel call.

Process Manager

/usr/src/servers/pm/Makefile (modified, generated)
Include rt.c and klog.c in the build process.

/usr/src/servers/pm/.depend (modified, generated)
Added dependencies of rt.c and klog.c

/usr/src/servers/pm/forkexit.c (modified)
Modified pm_exit() function to inform the semaphore server when a process exits.

/usr/src/servers/pm/klog.c (created)
System call implementations for kernel logging.

/usr/src/servers/pm/proto.h (modified)
Added function prototypes for added system calls.

/usr/src/servers/pm/rt.c (created)
System call implementations for real-time functionality.

/usr/src/servers/pm/table.c (modified)
Map the added system calls in the call_vec[] array.

Information Server

/usr/src/servers/is/dmp.c (modified)
Increased NHOOKS macro for the added key binding. Modified struct hook_entry to map shift + F9 to the rt_sched_dmp() function.

/usr/src/servers/is/dmp_kernel.c (modified)
Added implementation of rt_sched_dmp() function.

/usr/src/servers/is/main.c (modified)
Modified init_server() function to hook also key shift + F9.

/usr/src/servers/is/proto.h (modified)
Added function prototype for rt_sched_dmp() function.

Semaphore Server

/usr/src/servers/ss/Makefile (created)
Makefile to build the semaphore server.

/usr/src/servers/ss/.depend (created, generated)
Dependencies for the semaphore server source files.

/usr/src/servers/ss/proto.h (created)
Function prototypes for the semaphore server.

/usr/src/servers/ss/ss.h (created)
Main header file of the semaphore server.

/usr/src/servers/ss/ss.c (created)
Implementation of the semaphore server.

Real-time library

/usr/src/lib/rtminix3/Makefile.in (created)
Makefile generation file for librt.

/usr/src/lib/rtminix3/.depend-ack (created, generated)
ACK compiler dependencies for librt.

/usr/src/lib/rtminix3/.depend-gnu (created, generated)
GNU compiler dependencies for librt.

/usr/src/lib/rtminix3/Makedepend-ack (created, generated)
Dependency file generator for the ACK compiler

/usr/src/lib/rtminix3/Makedepend-gnu (created, generated)
Dependency file generator for the GNU compiler

/usr/src/lib/rtminix3/Makefile (created, generated)
Makefile for librt.

/usr/src/lib/rtminix3/rt_nextperiod.c (created)
Implementation of the rt_nextperiod() library function.

/usr/src/lib/rtminix3/rt_set_edf.c (created)
Implementation of the rt_set_edf() library function.

/usr/src/lib/rtminix3/rt_set_rm.c (created)
Implementation of the rt_set_rm() library function.

/usr/src/lib/rtminix3/rt_set_sched_bridge.c (created)
Implementation of the rt_set_sched_bridge() library function.

/usr/src/lib/rtminix3/rt_set_sched_edf.c (created)
Implementation of the rt_set_sched_edf() library function.

/usr/src/lib/rtminix3/rt_set_sched_rm.c (created)
Implementation of the rt_set_sched_rm() library function.

Kernel logger library

/usr/src/lib/klog/Makefile.in (created)
Makefile generation file for libklog.

/usr/src/lib/klog/.depend-ack (created, generated)
ACK compiler dependencies for libklog.

/usr/src/lib/klog/.depend-gnu (created, generated)
GNU compiler dependencies for libklog.

/usr/src/lib/klog/Makedepend-ack (created, generated)
Dependency file generator for the ACK compiler

/usr/src/lib/klog/Makedepend-gnu (created, generated)
Dependency file generator for the GNU compiler

/usr/src/lib/klog/Makefile (created, generated)
Makefile for libklog.

/usr/src/lib/klog/klog_copy.c (created)
Implementation of the klog_set() library function.

/usr/src/lib/klog/klog_set.c (created)
Implementation of the klog_copy() library function.

Semaphore library

/usr/src/lib/sem/Makefile.in (created)
Makefile generation file for libsem.

/usr/src/lib/sem/.depend-ack (created, generated)
ACK compiler dependencies for libsem.

/usr/src/lib/sem/.depend-gnu (created, generated)
GNU compiler dependencies for libsem.

/usr/src/lib/sem/Makedepend-ack (created, generated)
Dependency file generator for the ACK compiler.

/usr/src/lib/sem/Makedepend-gnu (created, generated)
Dependency file generator for the GNU compiler.

/usr/src/lib/sem/Makefile (created, generated)

/usr/src/lib/sem/sem_b_create.c (created)
Implementation of the sem_b_create() library function.

/usr/src/lib/sem/sem_c_create.c (created)
Implementation of the sem_c_create() library function.

/usr/src/lib/sem/sem_m_create.c (created)
Implementation of the sem_m_create() library function.

/usr/src/lib/sem/sem_delete.c (created)
Implementation of the sem_delete() library function.

/usr/src/lib/sem/sem_flush.c (created)
Implementation of the sem_flush() library function.

/usr/src/lib/sem/sem_give.c (created)
Implementation of the sem_give() library function.

/usr/src/lib/sem/sem_take.c (created)
Implementation of the sem_take() library function.

/usr/src/lib/sem/sem_value.c (created)
Implementation of the sem_value() library function.

System library

/usr/src/lib/syslib/Makefile.in (modified)
Added added system library functions.

/usr/src/lib/syslib/Makefile (modified, generated)
Includes the added system library functions in the build process.

/usr/src/lib/syslib/.depend-ack (modified, generated)
Dependencies for Ack compiler.

/usr/src/lib/syslib/sys_klog_copy.c (created)
Implementation of the sys_klog_copy() library function.

/usr/src/lib/syslib/sys_klog_set.c (created)
Implementation of the sys_klog_set() library function.

/usr/src/lib/syslib/sys_rt_nextperiod.c (created)
Implementation of the sys_rt_nextperiod() library function.

/usr/src/lib/syslib/sys_rt_set.c (created)
Implementation of the sys_rt_set() library function.

/usr/src/lib/syslib/sys_rt_set_sched.c (created)
Implementation of the sys_rt_set_sched() library function.

/usr/src/lib/syslib/sys_rt_sched_bridge.c (created)
Implementation of the sys_rt_sched_bridge() library function.

/usr/src/lib/syslib/sys_rt_show_data.c (created)
Implementation of the sys_rt_show_data() library function.

Commands

/usr/src/commands/reboot/halt.c (modified)
Added call to RT_SET_SCHED_BRIDGE system call to enable scheduling bridge.

/usr/src/commands/reboot/Makefile (modified)
halt.c has to be linked with librt after our modification.

/usr/src/commands/simple/Makefile (modified)
Includes the added programs in the build process.

/usr/src/commands/simple/edftop.c (created)
Implementation of edftop program which monitors processes scheduled by EDF.

/usr/src/commands/simple/klog_copy.c (created)
Implementation of klog_copy program to copy the kernel log.

/usr/src/commands/simple/klog_set_ci.c (created)
Implementation of klog_set_ci program which enables kernel logging and logs every running process when a clock interrupt happens.

/usr/src/commands/simple/klog_set_cs.c (created)
Implementation of klog_set_cs program which enables kernel logging and logs every next running process on a context switch.

/usr/src/commands/simple/rt_set_sched_edf.c (created)
Implementation of rt_set_sched_edf program which sets the real-time scheduler to EDF.

/usr/src/commands/simple/rt_set_sched_rm_pnu.c (created)
Implementation of rt_set_sched_rm_pnu program which sets the real-time scheduler to RM and does not enforce unique priorities.

/usr/src/commands/simple/rt_set_sched_rm_pu.c (created)
Implementation of rt_set_sched_rm_pnu program which sets the real-time scheduler to RM and enforces unique priorities.

Tests and examples

/usr/src/test/rtminix3/* (created)
Tests and examples for RM and EDF scheduler and semaphores.

Other

/usr/src/tools/Makefile (modified)
Added build support for the semaphore server in the boot image.

/usr/src/servers/Makefile (modified)
Added build support for the semaphore server in the boot image.

/usr/src/lib/Makefile.in (modified)
Added librt, libklog and libsem to the Makefile generation file.

/usr/src/lib/Makefile (modified, generated)

Includes librt, libklog and libsem in the build process.

D Source code

In this appendix we list all the modified or added source files. Makefiles and other configuration files and tests are not included.

Listings

Listing 17: /usr/src/include/lib.h

```
/* The <lib.h> header is the master header used by the library.
 * All the C files in the lib subdirectories include it.
 */

#ifndef _LIB_H
#define _LIB_H

/* First come the defines. */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */

/* The following are so basic, all the lib files get them automatically. */
#include <minix/config.h>       /* must be first */
#include <sys/types.h>
#include <limits.h>
#include <errno.h>
#include <ansi.h>

#include <minix/const.h>
#include <minix/com.h>
#include <minix/type.h>
#include <minix/callnr.h>

#include <minix/ipc.h>

#define MM                 PM_PROC_NR
#define FS                 FS_PROC_NR

/* synchronisation server */
#define SS                 SS_PROC_NR

_PROTOTYPE( int __execve, (const char *_path, char *const _argv[],
                        char *const _envp[], int _nargs, int _nenvps)   );
_PROTOTYPE( int _syscall, (int _who, int _syscallnr, message *_msgptr)  );
_PROTOTYPE( void _loadname, (const char *_name, message *_msgptr)       );
_PROTOTYPE( int _len, (const char *_s)                                  );
_PROTOTYPE( void _begsig, (int _dummy)                                  );

#endif /* _LIB_H */
```

Listing 18: /usr/src/include/minix/callnr.h

```
#define NCALLS            95    /* number of system calls allowed */

/* call nrs for RTMINIX3 */
#define RT_SET_SCHED  45
#define RT_SET        57
#define RT_NEXTPERIOD 58
#define RT_SET_SCHED_BRIDGE 64
#define KLOG_SET      49
#define KLOG_COPY     50

#define EXIT               1
#define FORK               2
#define READ               3
#define WRITE              4
#define OPEN               5
#define CLOSE              6
#define WAIT               7
#define CREAT              8
#define LINK               9
#define UNLINK            10
#define WAITPID           11
#define CHDIR             12
#define TIME              13
#define MKNOD             14
#define CHMOD             15
#define CHOWN             16
#define BRK               17
#define STAT              18
#define LSEEK             19
#define GETPID            20
#define MOUNT             21
#define UMOUNT            22
#define SETUID            23
#define GETUID            24
#define STIME             25
#define PTRACE            26
#define ALARM             27
#define FSTAT             28
#define PAUSE             29
#define UTIME             30
#define ACCESS            33
#define SYNC              36
#define KILL              37
#define RENAME            38
#define MKDIR             39
#define RMDIR             40
#define DUP               41
#define PIPE              42
#define TIMES             43
#define SYMLINK           45
#define SETGID            46
#define GETGID            47
#define SIGNAL            48
#define RDLNK             49
#define LSTAT             50
#define IOCTL             54
#define FCNTL             55
#define EXEC              59
#define UMASK             60
#define CHROOT            61
```

```
#define SETSID          62
#define GETPGRP         63

/* The following are not system calls, but are processed like them. */
#define UNPAUSE         65    /* to MM or FS: check for EINTR */
#define REVIVE          67    /* to FS: revive a sleeping process */
#define TASK_REPLY      68    /* to FS: reply code from tty task */

/* Posix signal handling. */
#define SIGACTION       71
#define SIGSUSPEND      72
#define SIGPENDING      73
#define SIGPROCMASK     74
#define SIGRETURN       75

#define REBOOT          76    /* to PM */

/* MINIX specific calls, e.g., to support system services. */
#define SVRCTL          77
#define PROCSTAT        78    /* to PM */
#define GETSYSINFO      79    /* to PM or FS */
#define GETPROCNR       80    /* to PM */
#define DEVCTL          81    /* to FS */
#define FSTATFS         82    /* to FS */
#define ALLOCMEM        83    /* to PM */
#define FREEMEM         84    /* to PM */
#define SELECT          85    /* to FS */
#define FCHDIR          86    /* to FS */
#define FSYNC           87    /* to FS */
#define GETPRIORITY     88    /* to PM */
#define SETPRIORITY     89    /* to PM */
#define GETTIMEOFDAY    90    /* to PM */
#define SETEUID         91    /* to PM */
#define SETEGID         92    /* to PM */
#define TRUNCATE        93    /* to FS */
#define FTRUNCATE       94    /* to FS */
```

Listing 19: /usr/src/include/minix/com.h

```
#ifndef _MINIX_COM_H
#define _MINIX_COM_H

/*===========================================================================*
 *                              Magic process numbers                        *
 *===========================================================================*/

/* These may not be any valid endpoint (see <minix/endpoint.h>). */
#define ANY             0x7ace  /* used to indicate 'any process' */
#define NONE            0x6ace  /* used to indicate 'no process at all' */
#define SELF            0x8ace  /* used to indicate 'own process' */
#define _MAX_MAGIC_PROC (SELF)  /* used by <minix/endpoint.h>
                                   to determine generation size */

/*===========================================================================*
 *              Process numbers of processes in the system image             *
 *===========================================================================*/

/* The values of several task numbers depend on whether they or other tasks
 * are enabled. They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
 * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
 * the same number as the previous task and is further unused. Note that the
 * order should correspond to the order in the task table defined in table.c.
 */

/* Kernel tasks. These all run in the same address space. */
#define IDLE             -4     /* runs when no one else can run */
#define CLOCK            -3     /* alarms and other clock functions */
#define SYSTEM           -2     /* request system functionality */
#define KERNEL           -1     /* pseudo-process for IPC and scheduling */
#define HARDWARE     KERNEL     /* for hardware interrupt handlers */

/* Number of tasks. Note that NR_PROCS is defined in <minix/config.h>. */
#define NR_TASKS          4

/* User-space processes, that is, device drivers, servers, and INIT. */
#define PM_PROC_NR        0     /* process manager */
#define FS_PROC_NR        1     /* file system */
#define RS_PROC_NR        2     /* reincarnation server */
#define MEM_PROC_NR   3 /* memory driver (RAM disk, null, etc.) */
#define LOG_PROC_NR       4     /* log device driver */
#define TTY_PROC_NR       5     /* terminal (TTY) driver */
#define DS_PROC_NR        6 /* data store server */
#define SS_PROC_NR    7 /* semaphore server */
#define INIT_PROC_NR  8 /* init -- goes multiuser */

/* Number of processes contained in the system image. */
#define NR_BOOT_PROCS   (NR_TASKS + INIT_PROC_NR + 1)

/*===========================================================================*
 *                         Kernel notification types                         *
 *===========================================================================*/

/* Kernel notification types. In principle, these can be sent to any process,
 * so make sure that these types do not interfere with other message types.
 * Notifications are prioritized because of the way they are unhold() and
 * blocking notifications are delivered. The lowest numbers go first. The
 * offset are used for the per-process notification bit maps.
 */
#define NOTIFY_MESSAGE            0x1000
```

```
#define NOTIFY_FROM(p_nr)        (NOTIFY_MESSAGE | ((p_nr) + NR_TASKS))
#  define PROC_EVENT    NOTIFY_FROM(PM_PROC_NR) /* process status change */
#  define SYN_ALARM     NOTIFY_FROM(CLOCK)      /* synchronous alarm */
#  define SYS_SIG       NOTIFY_FROM(SYSTEM)     /* system signal */
#  define HARD_INT      NOTIFY_FROM(HARDWARE)   /* hardware interrupt */
#  define NEW_KSIG      NOTIFY_FROM(HARDWARE)   /* new kernel signal */
#  define FKEY_PRESSED  NOTIFY_FROM(TTY_PROC_NR)/* function key press */
#  define DEV_PING      NOTIFY_FROM(RS_PROC_NR) /* driver liveness ping */

/* Shorthands for message parameters passed with notifications. */
#define NOTIFY_SOURCE           m_source
#define NOTIFY_TYPE             m_type
#define NOTIFY_ARG              m2_l1
#define NOTIFY_TIMESTAMP        m2_l2
#define NOTIFY_FLAGS            m2_i1

/*===========================================================================*
 *                Messages for BUS controller drivers                        *
 *===========================================================================*/
#define BUSC_RQ_BASE    0x300   /* base for request types */
#define BUSC_RS_BASE    0x380   /* base for response types */

#define BUSC_PCI_INIT           (BUSC_RQ_BASE + 0)      /* First message to
                                                         * PCI driver
                                                         */
#define BUSC_PCI_FIRST_DEV      (BUSC_RQ_BASE + 1)      /* Get index (and
                                                         * vid/did) of the
                                                         * first PCI device
                                                         */
#define BUSC_PCI_NEXT_DEV       (BUSC_RQ_BASE + 2)      /* Get index (and
                                                         * vid/did) of the
                                                         * next PCI device
                                                         */
#define BUSC_PCI_FIND_DEV       (BUSC_RQ_BASE + 3)      /* Get index of a
                                                         * PCI device based on
                                                         * bus/dev/function
                                                         */
#define BUSC_PCI_IDS            (BUSC_RQ_BASE + 4)      /* Get vid/did from an
                                                         * index
                                                         */
#define BUSC_PCI_DEV_NAME       (BUSC_RQ_BASE + 5)      /* Get the name of a
                                                         * PCI device
                                                         */
#define BUSC_PCI_SLOT_NAME      (BUSC_RQ_BASE + 6)      /* Get the name of a
                                                         * PCI slot
                                                         */
#define BUSC_PCI_RESERVE        (BUSC_RQ_BASE + 7)      /* Reserve a PCI dev */
#define BUSC_PCI_ATTR_R8        (BUSC_RQ_BASE + 8)      /* Read 8-bit
                                                         * attribute value
                                                         */
#define BUSC_PCI_ATTR_R16       (BUSC_RQ_BASE + 9)      /* Read 16-bit
                                                         * attribute value
                                                         */
#define BUSC_PCI_ATTR_R32       (BUSC_RQ_BASE + 10)     /* Read 32-bit
                                                         * attribute value
                                                         */
#define BUSC_PCI_ATTR_W8        (BUSC_RQ_BASE + 11)     /* Write 8-bit
                                                         * attribute value
                                                         */
#define BUSC_PCI_ATTR_W16       (BUSC_RQ_BASE + 12)     /* Write 16-bit
                                                         * attribute value
                                                         */
```

```
#define BUSC_PCI_ATTR_W32       (BUSC_RQ_BASE + 13)     /* Write 32-bit
                                                         * attribute value
                                                         */
#define BUSC_PCI_RESCAN         (BUSC_RQ_BASE + 14)     /* Rescan bus */

/*===========================================================================*
 *                Messages for BLOCK and CHARACTER device drivers            *
 *===========================================================================*/

/* Message types for device drivers. */
#define DEV_RQ_BASE   0x400     /* base for device request types */
#define DEV_RS_BASE   0x500     /* base for device response types */

#define CANCEL          (DEV_RQ_BASE +  0) /* force a task to cancel */
#define DEV_READ        (DEV_RQ_BASE +  3) /* read from minor device */
#define DEV_WRITE       (DEV_RQ_BASE +  4) /* write to minor device */
#define DEV_IOCTL       (DEV_RQ_BASE +  5) /* I/O control code */
#define DEV_OPEN        (DEV_RQ_BASE +  6) /* open a minor device */
#define DEV_CLOSE       (DEV_RQ_BASE +  7) /* close a minor device */
#define DEV_SCATTER     (DEV_RQ_BASE +  8) /* write from a vector */
#define DEV_GATHER      (DEV_RQ_BASE +  9) /* read into a vector */
#define TTY_SETPGRP     (DEV_RQ_BASE + 10) /* set process group */
#define TTY_EXIT        (DEV_RQ_BASE + 11) /* process group leader exited */
#define DEV_SELECT      (DEV_RQ_BASE + 12) /* request select() attention */
#define DEV_STATUS      (DEV_RQ_BASE + 13) /* request driver status */

#define DEV_REPLY       (DEV_RS_BASE + 0) /* general task reply */
#define DEV_CLONED      (DEV_RS_BASE + 1) /* return cloned minor */
#define DEV_REVIVE      (DEV_RS_BASE + 2) /* driver revives process */
#define DEV_IO_READY    (DEV_RS_BASE + 3) /* selected device ready */
#define DEV_NO_STATUS   (DEV_RS_BASE + 4) /* empty status reply */

/* Field names for messages to block and character device drivers. */
#define DEVICE          m2_i1    /* major-minor device */
#define IO_ENDPT        m2_i2    /* which (proc/endpoint) wants I/O? */
#define COUNT           m2_i3    /* how many bytes to transfer */
#define REQUEST         m2_i3    /* ioctl request code */
#define POSITION        m2_l1    /* file offset */
#define ADDRESS         m2_p1    /* core buffer address */

/* Field names for DEV_SELECT messages to device drivers. */
#define DEV_MINOR       m2_i1    /* minor device */
#define DEV_SEL_OPS     m2_i2    /* which select operations are requested */
#define DEV_SEL_WATCH   m2_i3    /* request notify if no operations are ready */

/* Field names used in reply messages from tasks. */
#define REP_ENDPT       m2_i1    /* # of proc on whose behalf I/O was done */
#define REP_STATUS      m2_i2    /* bytes transferred or error number */
#  define SUSPEND        -998    /* status to suspend caller, reply later */

/* Field names for messages to TTY driver. */
#define TTY_LINE        DEVICE   /* message parameter: terminal line */
#define TTY_REQUEST     COUNT    /* message parameter: ioctl request code */
#define TTY_SPEK        POSITION/* message parameter: ioctl speed, erasing */
#define TTY_FLAGS       m2_l2    /* message parameter: ioctl tty mode */
#define TTY_PGRP        m2_i3    /* message parameter: process group */

/* Field names for the QIC 02 status reply from tape driver */
#define TAPE_STAT0      m2_l1
#define TAPE_STAT1      m2_l2

/*===========================================================================*
```

```
 *                          Messages for networking layer                   *
 *==========================================================================*/

/* Message types for network layer requests. This layer acts like a driver. */
#define NW_OPEN         DEV_OPEN
#define NW_CLOSE        DEV_CLOSE
#define NW_READ         DEV_READ
#define NW_WRITE        DEV_WRITE
#define NW_IOCTL        DEV_IOCTL
#define NW_CANCEL       CANCEL

/* Base type for data link layer requests and responses. */
#define DL_RQ_BASE      0x800
#define DL_RS_BASE      0x900

/* Message types for data link layer requests. */
#define DL_WRITE        (DL_RQ_BASE + 3)
#define DL_WRITEV       (DL_RQ_BASE + 4)
#define DL_READ         (DL_RQ_BASE + 5)
#define DL_READV        (DL_RQ_BASE + 6)
#define DL_INIT         (DL_RQ_BASE + 7)
#define DL_STOP         (DL_RQ_BASE + 8)
#define DL_GETSTAT      (DL_RQ_BASE + 9)
#define DL_GETNAME      (DL_RQ_BASE +10)

/* Message type for data link layer replies. */
#define DL_INIT_REPLY   (DL_RS_BASE + 20)
#define DL_TASK_REPLY   (DL_RS_BASE + 21)
#define DL_NAME_REPLY   (DL_RS_BASE + 22)

/* Field names for data link layer messages. */
#define DL_PORT         m2_i1
#define DL_PROC         m2_i2   /* endpoint */
#define DL_COUNT        m2_i3
#define DL_MODE         m2_l1
#define DL_CLCK         m2_l2
#define DL_ADDR         m2_p1
#define DL_STAT         m2_l1
#define DL_NAME         m3_ca1

/* Bits in 'DL_STAT' field of DL replies. */
#  define DL_PACK_SEND          0x01
#  define DL_PACK_RECV          0x02
#  define DL_READ_IP            0x04

/* Bits in 'DL_MODE' field of DL requests. */
#  define DL_NOMODE             0x0
#  define DL_PROMISC_REQ        0x2
#  define DL_MULTI_REQ          0x4
#  define DL_BROAD_REQ          0x8

/*===========================================================================*
 *                   SYSTASK request types and field names                   *
 *===========================================================================*/

/* System library calls are dispatched via a call vector, so be careful when
 * modifying the system call numbers. The numbers here determine which call
 * is made from the call vector.
 */
#define KERNEL_CALL     0x600   /* base for kernel calls to SYSTEM */

#  define SYS_FORK       (KERNEL_CALL + 0)      /* sys_fork() */
#  define SYS_EXEC       (KERNEL_CALL + 1)      /* sys_exec() */
#  define SYS_EXIT       (KERNEL_CALL + 2)      /* sys_exit() */
#  define SYS_NICE       (KERNEL_CALL + 3)      /* sys_nice() */
#  define SYS_PRIVCTL    (KERNEL_CALL + 4)      /* sys_privctl() */
#  define SYS_TRACE      (KERNEL_CALL + 5)      /* sys_trace() */
#  define SYS_KILL       (KERNEL_CALL + 6)      /* sys_kill() */

#  define SYS_GETKSIG    (KERNEL_CALL + 7)      /* sys_getsig() */
#  define SYS_ENDKSIG    (KERNEL_CALL + 8)      /* sys_endsig() */
#  define SYS_SIGSEND    (KERNEL_CALL + 9)      /* sys_sigsend() */
#  define SYS_SIGRETURN  (KERNEL_CALL + 10)     /* sys_sigreturn() */

#  define SYS_NEWMAP     (KERNEL_CALL + 11)     /* sys_newmap() */
#  define SYS_SEGCTL     (KERNEL_CALL + 12)     /* sys_segctl() */
#  define SYS_MEMSET     (KERNEL_CALL + 13)     /* sys_memset() */

#  define SYS_UMAP       (KERNEL_CALL + 14)     /* sys_umap() */
#  define SYS_VIRCOPY    (KERNEL_CALL + 15)     /* sys_vircopy() */
#  define SYS_PHYSCOPY   (KERNEL_CALL + 16)     /* sys_physcopy() */
#  define SYS_VIRVCOPY   (KERNEL_CALL + 17)     /* sys_virvcopy() */
#  define SYS_PHYSVCOPY  (KERNEL_CALL + 18)     /* sys_physvcopy() */

#  define SYS_IRQCTL     (KERNEL_CALL + 19)     /* sys_irqctl() */
#  define SYS_INT86      (KERNEL_CALL + 20)     /* sys_int86() */
#  define SYS_DEVIO      (KERNEL_CALL + 21)     /* sys_devio() */
#  define SYS_SDEVIO     (KERNEL_CALL + 22)     /* sys_sdevio() */
#  define SYS_VDEVIO     (KERNEL_CALL + 23)     /* sys_vdevio() */

#  define SYS_SETALARM   (KERNEL_CALL + 24)     /* sys_setalarm() */
#  define SYS_TIMES      (KERNEL_CALL + 25)     /* sys_times() */
#  define SYS_GETINFO    (KERNEL_CALL + 26)     /* sys_getinfo() */
#  define SYS_ABORT      (KERNEL_CALL + 27)     /* sys_abort() */
#  define SYS_IOPENABLE  (KERNEL_CALL + 28)     /* sys_enable_iop() */
#  define SYS_VM_SETBUF  (KERNEL_CALL + 29)     /* sys_vm_setbuf() */
#  define SYS_VM_MAP     (KERNEL_CALL + 30)     /* sys_vm_map() */

/* requests for real-time minix 3 and kernel logging */
#  define SYS_RT_SET_SCHED      (KERNEL_CALL + 31) /* sys_rt_set_sched() */
#  define SYS_RT_SET            (KERNEL_CALL + 32) /* sys_rt_set() */
#  define SYS_RT_SHOW_DATA      (KERNEL_CALL + 33) /* sys_rt_show_data() */
#  define SYS_RT_NEXTPERIOD     (KERNEL_CALL + 34) /* sys_rt_nextperiod() */
#  define SYS_RT_SCHED_BRIDGE   (KERNEL_CALL + 35) /* sys_rt_sched_bridge() */
#  define SYS_KLOG_SET          (KERNEL_CALL + 36) /* sys_klog_set() */
#  define SYS_KLOG_COPY         (KERNEL_CALL + 37) /* sys_klog_copy() */

#define NR_SYS_CALLS    38      /* number of system calls */

/* Subfunctions for SYS_PRIVCTL */
#define SYS_PRIV_INIT           1       /* Initialize a privilege structure */
#define SYS_PRIV_ADD_IO         2       /* Add I/O range (struct io_range) */
#define SYS_PRIV_ADD_MEM        3       /* Add memory range (struct mem_range)
                                         */
#define SYS_PRIV_ADD_IRQ        4       /* Add IRQ */

/* Field names for SYS_MEMSET, SYS_SEGCTL. */
#define MEM_PTR         m2_p1   /* base */
#define MEM_COUNT       m2_l1   /* count */
#define MEM_PATTERN     m2_l2   /* pattern to write */
#define MEM_CHUNK_BASE  m4_l1   /* physical base address */
#define MEM_CHUNK_SIZE  m4_l2   /* size of mem chunk */
#define MEM_TOT_SIZE    m4_l3   /* total memory size */
#define MEM_CHUNK_TAG   m4_l4   /* tag to identify chunk of mem */
```

```
/* Field names for SYS_DEVIO, SYS_VDEVIO, SYS_SDEVIO. */
#define DIO_REQUEST      m2_i3    /* device in or output */
#    define DIO_INPUT         0   /* input */
#    define DIO_OUTPUT        1   /* output */
#define DIO_TYPE         m2_i1    /* flag indicating byte, word, or long */
#    define DIO_BYTE        'b'   /* byte type values */
#    define DIO_WORD        'w'   /* word type values */
#    define DIO_LONG        'l'   /* long type values */
#define DIO_PORT         m2_l1    /* single port address */
#define DIO_VALUE        m2_l2    /* single I/O value */
#define DIO_VEC_ADDR     m2_p1    /* address of buffer or (p,v)-pairs */
#define DIO_VEC_SIZE     m2_l2    /* number of elements in vector */
#define DIO_VEC_ENDPT    m2_i2    /* number of process where vector is */

/* Field names for SYS_SIGNARLM, SYS_FLAGARLM, SYS_SYNCALRM. */
#define ALRM_EXP_TIME    m2_l1    /* expire time for the alarm call */
#define ALRM_ABS_TIME    m2_i2    /* set to 1 to use absolute alarm time */
#define ALRM_TIME_LEFT   m2_l1    /* how many ticks were remaining */
#define ALRM_ENDPT       m2_i1    /* which process wants the alarm? */
#define ALRM_FLAG_PTR    m2_p1    /* virtual address of timeout flag */

/* Field names for SYS_IRQCTL. */
#define IRQ_REQUEST      m5_c1    /* what to do? */
#  define IRQ_SETPOLICY     1     /* manage a slot of the IRQ table */
#  define IRQ_RMPOLICY      2     /* remove a slot of the IRQ table */
#  define IRQ_ENABLE        3     /* enable interrupts */
#  define IRQ_DISABLE       4     /* disable interrupts */
#define IRQ_VECTOR       m5_c2    /* irq vector */
#define IRQ_POLICY       m5_i1    /* options for IRQCTL request */
#  define IRQ_REENABLE   0x001    /* reenable IRQ line after interrupt */
#  define IRQ_BYTE       0x100    /* byte values */
#  define IRQ_WORD       0x200    /* word values */
#  define IRQ_LONG       0x400    /* long values */
#define IRQ_ENDPT        m5_i2    /* endpoint number, SELF, NONE */
#define IRQ_HOOK_ID      m5_l3    /* id of irq hook at kernel */

/* Field names for SYS_SEGCTL. */
#define SEG_SELECT       m4_l1    /* segment selector returned */
#define SEG_OFFSET       m4_l2    /* offset in segment returned */
#define SEG_PHYS         m4_l3    /* physical address of segment */
#define SEG_SIZE         m4_l4    /* segment size */
#define SEG_INDEX        m4_l5    /* segment index in remote map */

/* Field names for SYS_VIDCOPY. */
#define VID_REQUEST      m4_l1    /* what to do? */
#  define VID_VID_COPY      1     /* request vid_vid_copy() */
#  define MEM_VID_COPY      2     /* request mem_vid_copy() */
#define VID_SRC_ADDR     m4_l2    /* virtual address in memory */
#define VID_SRC_OFFSET   m4_l3    /* offset in video memory */
#define VID_DST_OFFSET   m4_l4    /* offset in video memory */
#define VID_CP_COUNT     m4_l5    /* number of words to be copied */

/* Field names for SYS_ABORT. */
#define ABRT_HOW         m1_i1    /* RBT_REBOOT, RBT_HALT, etc. */
#define ABRT_MON_ENDPT   m1_i2    /* process where monitor params are */
#define ABRT_MON_LEN     m1_i3    /* length of monitor params */
#define ABRT_MON_ADDR    m1_p1    /* virtual address of monitor params */

/* Field names for _UMAP, _VIRCOPY, _PHYSCOPY. */
#define CP_SRC_SPACE     m5_c1    /* T or D space (stack is also D) */
#define CP_SRC_ENDPT     m5_i1    /* process to copy from */
```

```
#define CP_SRC_ADDR      m5_l1    /* address where data come from */
#define CP_DST_SPACE     m5_c2    /* T or D space (stack is also D) */
#define CP_DST_ENDPT     m5_i2    /* process to copy to */
#define CP_DST_ADDR      m5_l2    /* address where data go to */
#define CP_NR_BYTES      m5_l3    /* number of bytes to copy */

/* Field names for SYS_VCOPY and SYS_VVIRCOPY. */
#define VCP_NR_OK        m1_i2    /* number of successfull copies */
#define VCP_VEC_SIZE     m1_i3    /* size of copy vector */
#define VCP_VEC_ADDR     m1_p1    /* pointer to copy vector */

/* Field names for SYS_GETINFO. */
#define I_REQUEST       m7_i3     /* what info to get */
#   define GET_KINFO        0     /* get kernel information structure */
#   define GET_IMAGE        1     /* get system image table */
#   define GET_PROCTAB      2     /* get kernel process table */
#   define GET_RANDOMNESS   3     /* get randomness buffer */
#   define GET_MONPARAMS    4     /* get monitor parameters */
#   define GET_KENV         5     /* get kernel environment string */
#   define GET_IRQHOOKS     6     /* get the IRQ table */
#   define GET_KMESSAGES    7     /* get kernel messages */
#   define GET_PRIVTAB      8     /* get kernel privileges table */
#   define GET_KADDRESSES   9     /* get various kernel addresses */
#   define GET_SCHEDINFO   10     /* get scheduling queues */
#   define GET_PROC        11     /* get process slot if given process */
#   define GET_MACHINE     12     /* get machine information */
#   define GET_LOCKTIMING  13     /* get lock()/unlock() latency timing */
#   define GET_BIOSBUFFER  14     /* get a buffer for BIOS calls */
#   define GET_LOADINFO    15     /* get load average information */
#define I_ENDPT       m7_i4       /* calling process */
#define I_VAL_PTR       m7_p1     /* virtual address at caller */
#define I_VAL_LEN       m7_i1     /* max length of value */
#define I_VAL_PTR2      m7_p2     /* second virtual address */
#define I_VAL_LEN2_E    m7_i2     /* second length, or proc nr */
#   define GET_IRQACTIDS   16     /* get the IRQ masks */

/* Field names for SYS_TIMES. */
#define T_ENDPT       m4_l1       /* process to request time info for */
#define T_USER_TIME     m4_l1     /* user time consumed by process */
#define T_SYSTEM_TIME   m4_l2     /* system time consumed by process */
#define T_CHILD_UTIME   m4_l3     /* user time consumed by process' children */
#define T_CHILD_STIME   m4_l4     /* sys time consumed by process' children */
#define T_BOOT_TICKS    m4_l5     /* number of clock ticks since boot time */

/* vm_map */
#define VM_MAP_ENDPT              m4_l1
#define VM_MAP_MAPUNMAP           m4_l2
#define VM_MAP_BASE               m4_l3
#define VM_MAP_SIZE               m4_l4
#define VM_MAP_ADDR               m4_l5

/* Field names for SYS_TRACE, SYS_PRIVCTL. */
#define CTL_ENDPT     m2_i1       /* process number of the caller */
#define CTL_REQUEST     m2_i2     /* server control request */
#define CTL_MM_PRIV     m2_i3     /* privilege as seen by PM */
#define CTL_ARG_PTR     m2_p1     /* pointer to argument */
#define CTL_ADDRESS     m2_l1     /* address at traced process' space */
#define CTL_DATA        m2_l2     /* data field for tracing */

/* Field names for SYS_KILL, SYS_SIGCTL */
#define SIG_REQUEST     m2_l2     /* PM signal control request */
#define S_GETSIG            0     /* get pending kernel signal */
```

```
#define S_ENDSIG           1    /* finish a kernel signal */
#define S_SENDSIG          2    /* POSIX style signal handling */
#define S_SIGRETURN        3    /* return from POSIX handling */
#define S_KILL             4    /* servers kills process with signal */
#define SIG_ENDPT        m2_i1  /* process number for inform */
#define SIG_NUMBER      m2_i2   /* signal number to send */
#define SIG_FLAGS       m2_i3   /* signal flags field */
#define SIG_MAP         m2_l1   /* used by kernel to pass signal bit map */
#define SIG_CTXT_PTR    m2_p1   /* pointer to info to restore signal context */

/* Field names for SYS_FORK, _EXEC, _EXIT, _NEWMAP. */
#define PR_ENDPT        m1_i1   /* indicates a process */
#define PR_PRIORITY     m1_i2   /* process priority */
#define PR_SLOT         m1_i2   /* indicates a process slot */
#define PR_PID          m1_i3   /* process id at process manager */
#define PR_STACK_PTR    m1_p1   /* used for stack ptr in sys_exec, sys_getsp */
#define PR_TRACING      m1_i3   /* flag to indicate tracing is on/ off */
#define PR_NAME_PTR     m1_p2   /* tells where program name is for dmp */
#define PR_IP_PTR       m1_p3   /* initial value for ip after exec */
#define PR_MEM_PTR      m1_p1   /* tells where memory map is for sys_newmap */

/* Field names for SYS_INT86 */
#define INT86_REG86     m1_p1   /* pointer to registers */

/* Field names for SELECT (FS). */
#define SEL_NFDS        m8_i1
#define SEL_READFDS     m8_p1
#define SEL_WRITEFDS    m8_p2
#define SEL_ERRORFDS    m8_p3
#define SEL_TIMEOUT     m8_p4

/*===========================================================================*
 *                Messages for the Reincarnation Server                      *
 *===========================================================================*/

#define RS_RQ_BASE              0x700

#define RS_UP            (RS_RQ_BASE + 0)       /* start system service */
#define RS_DOWN          (RS_RQ_BASE + 1)       /* stop system service */
#define RS_REFRESH       (RS_RQ_BASE + 2)       /* restart system service */
#define RS_RESCUE        (RS_RQ_BASE + 3)       /* set rescue directory */
#define RS_SHUTDOWN      (RS_RQ_BASE + 4)       /* alert about shutdown */

#  define RS_CMD_ADDR           m1_p1           /* command string */
#  define RS_CMD_LEN            m1_i1           /* length of command */
#  define RS_PID                m1_i1           /* pid of system service */
#  define RS_PERIOD             m1_i2           /* heartbeat period */
#  define RS_DEV_MAJOR          m1_i3           /* major device number */

/*===========================================================================*
 *                Messages for the Data Store Server                         *
 *===========================================================================*/

#define DS_RQ_BASE              0x800

#define DS_PUBLISH       (DS_RQ_BASE + 0)       /* publish information */
#define DS_RETRIEVE      (DS_RQ_BASE + 1)       /* retrieve information */
#define DS_SUBSCRIBE     (DS_RQ_BASE + 2)       /* subscribe to information */

#  define DS_KEY                m2_i1           /* key for the information */
#  define DS_FLAGS              m2_i2           /* flags provided by caller */
#  define DS_AUTH               m2_p1           /* authorization of caller */
#   define DS_VAL_L1               m2_l1           /* first long data value */
#   define DS_VAL_L2               m2_l2           /* second long data value */

/*===========================================================================*
 *                Miscellaneous messages used by TTY                         *
 *===========================================================================*/

/* Miscellaneous request types and field names, e.g. used by IS server. */
#define FKEY_CONTROL            98      /* control a function key at the TTY */
#   define FKEY_REQUEST       m2_i1     /* request to perform at TTY */
#   define    FKEY_MAP          10      /* observe function key */
#   define    FKEY_UNMAP        11      /* stop observing function key */
#   define    FKEY_EVENTS       12      /* request open key presses */
#   define FKEY_FKEYS          m2_l1    /* F1-F12 keys pressed */
#   define FKEY_SFKEYS         m2_l2    /* Shift-F1-F12 keys pressed */
#define DIAGNOSTICS     100     /* output a string without FS in between */
#   define DIAG_PRINT_BUF      m1_p1
#   define DIAG_BUF_COUNT      m1_i1
#   define DIAG_ENDPT          m1_i2
#define GET_KMESS       101     /* get kmess from TTY */
#   define GETKM_PTR           m1_p1


#endif /* _MINIX_COM_H */
```

Listing 20: /usr/src/include/minix/klog.h

```
/* Main header file for kernel logging.
 * Processes using this functionality must be linked with libklog
 */

#ifndef KLOG_H
#define KLOG_H

/* number of entries in the kernel log */
#define KLOG_SIZE 1024

/* message fields */
#define KLOG_STATE m1_i1
#define KLOG_TYPE  m1_i2
#define KLOG_ENDPT m1_i1
#define KLOG_PTR   m1_p1

/* kernel log type */
#define KLOG_CONTEXTSWITCH  1
#define KLOG_CLOCKINT       2

/* declaration of the kernel log entry structure */
struct klog_entry {
    int klog_time;
    int klog_endpoint;
    char klog_type;
    char klog_name[8];
    int klog_data;
};

#ifndef _SYSTEM
/* library function prototypes */
int klog_set(int type);
int klog_copy(void *data);
#endif

#endif
```

Listing 21: /usr/src/include/minix/rt.h

```
/* Main header file for Real-time functionality.
 * Processes using these functionality should be linked with librt.
 */
#ifndef RT_H
#define RT_H

#include <ansi.h>

/* scheduler type definitions */
#define SCHED_UNDEFINED     0
#define SCHED_RM            1
#define SCHED_EDF           2

/* priority policies for Rate-Monotonic */
#define PRIO_UNIQUE         1
#define PRIO_NOT_UNIQUE     2


/* message fields for real-time related system calls */
#define RT_ENDPT        m7_i1
#define RT_SCHED        m7_i2
#define RM_PRIO         m7_i3
#define RM_PRIO_POLICY  m7_i4
#define EDF_PERIOD      m7_i3
#define EDF_CALCTIME    m7_i4
#define RT_SCHED_BRIDGE m1_i1

/* library function prototypes */
_PROTOTYPE( int rt_set_sched_edf, (void) );
_PROTOTYPE( int rt_set_sched_rm, (int prio_policy) );
_PROTOTYPE( int rt_set_edf, (int period, int calctime) );
_PROTOTYPE( int rt_set_rm, (int prio) );
_PROTOTYPE( int rt_set_sched_bridge, (int state) );
_PROTOTYPE( int rt_nextperiod, (void) );

#endif
```

Listing 22: /usr/src/include/minix/sem.h

```
/* Main header file for semaphores.
 * Processes using this functionality should be linked with libsem.
 */
#ifndef SEM_H
#define SEM_H

typedef int sem_t;

/* semaphore actions */
#define SEM_CREATE      1
#define SEM_TAKE        2
#define SEM_GIVE        3
#define SEM_FLUSH       4
#define SEM_DELETE      5
#define SEM_PROCEXIT    6
#define SEM_VALUE       7

/* semaphore types */
#define SEM_MUTEX       1
#define SEM_BINARY      2
#define SEM_COUNTING    3

/* semaphore return values */
#define SEM_INVALID_HANDLER     1
#define SEM_INVALID_TYPE        2
#define SEM_INVALID_VALUE       3
#define SEM_INVALID_SIZE        4
#define SEM_INVALID_BLOCK       5
#define SEM_NO_AVAILABLE        6
#define SEM_INVALID_MUTEX_GIVE  7
#define SEM_ALREADY_FREE        8
#define SEM_IN_USE              9

#ifndef OK
#define OK                      0
#endif

/* sem_take block types */
#define WAIT_FOREVER    0
#define NO_WAIT         1

/* message fields for IPC */
#define SEM_F_TYPE      m1_i1
#define SEM_F_VALUE     m1_i2
#define SEM_F_SIZE      m1_i3
#define SEM_F_HANDLER   m1_i1
#define SEM_F_BLOCK     m1_i2
#define SEM_F_ENDPT     m1_i1

#ifndef _SYSTEM
/* library function prototypes */
_PROTOTYPE(int sem_b_create, (sem_t *handler, int value));
_PROTOTYPE(int sem_m_create, (sem_t *handler));
_PROTOTYPE(int sem_c_create, (sem_t *handler, int value, int size));
_PROTOTYPE(int sem_take, (sem_t handler, int block));
_PROTOTYPE(int sem_give, (sem_t handler));
_PROTOTYPE(int sem_flush, (sem_t handler));
_PROTOTYPE(int sem_delete, (sem_t handler));
_PROTOTYPE(int sem_value, (sem_t handler, int *value));
#endif
```

```
#endif
```

Listing 23: /usr/src/include/minix/syslib.h

```
/* Prototypes for system library functions. */

#ifndef _SYSLIB_H
#define _SYSLIB_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

#ifndef _IPC_H
#include <minix/ipc.h>
#endif

#ifndef _DEVIO_H
#include <minix/devio.h>
#endif

/* Forward declaration */
struct reg86u;

#define SYSTASK SYSTEM

/*==========================================================================*
 * Minix system library.                                                    *
 *==========================================================================*/
_PROTOTYPE( int _taskcall, (int who, int syscallnr, message *msgptr));

_PROTOTYPE( int sys_abort, (int how, ...));
_PROTOTYPE( int sys_enable_iop, (int proc));
_PROTOTYPE( int sys_exec, (int proc, char *ptr,
                                char *aout, vir_bytes initpc));
_PROTOTYPE( int sys_fork, (int parent, int child, int *));
_PROTOTYPE( int sys_newmap, (int proc, struct mem_map *ptr));
_PROTOTYPE( int sys_exit, (int proc));
_PROTOTYPE( int sys_trace, (int req, int proc, long addr, long *data_p));

_PROTOTYPE( int sys_privctl, (int proc, int req, int i, void *p));
_PROTOTYPE( int sys_nice, (int proc, int priority));

_PROTOTYPE( int sys_int86, (struct reg86u *reg86p));
_PROTOTYPE( int sys_vm_setbuf, (phys_bytes base, phys_bytes size,
                                                phys_bytes high));
_PROTOTYPE( int sys_vm_map, (int proc_nr, int do_map,
        phys_bytes base, phys_bytes size, phys_bytes offset));

/* Shorthands for sys_sdevio() system call. */
#define sys_insb(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_INPUT, port, DIO_BYTE, proc_nr, buffer, count)
#define sys_insw(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_INPUT, port, DIO_WORD, proc_nr, buffer, count)
#define sys_outsb(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_OUTPUT, port, DIO_BYTE, proc_nr, buffer, count)
#define sys_outsw(port, proc_nr, buffer, count) \
        sys_sdevio(DIO_OUTPUT, port, DIO_WORD, proc_nr, buffer, count)
_PROTOTYPE( int sys_sdevio, (int req, long port, int type, int proc_nr,
        void *buffer, int count));

/* Clock functionality: get system times or (un)schedule an alarm call. */
_PROTOTYPE( int sys_times, (int proc_nr, clock_t *ptr));
_PROTOTYPE(int sys_setalarm, (clock_t exp_time, int abs_time));
```

```
/* Shorthands for sys_irqctl() system call. */
#define sys_irqdisable(hook_id) \
    sys_irqctl(IRQ_DISABLE, 0, 0, hook_id)
#define sys_irqenable(hook_id) \
    sys_irqctl(IRQ_ENABLE, 0, 0, hook_id)
#define sys_irqsetpolicy(irq_vec, policy, hook_id) \
    sys_irqctl(IRQ_SETPOLICY, irq_vec, policy, hook_id)
#define sys_irqrmpolicy(irq_vec, hook_id) \
    sys_irqctl(IRQ_RMPOLICY, irq_vec, 0, hook_id)
_PROTOTYPE ( int sys_irqctl, (int request, int irq_vec, int policy,
    int *irq_hook_id) );

/* Shorthands for sys_vircopy() and sys_physcopy() system calls. */
#define sys_biosin(bios_vir, dst_vir, bytes) \
        sys_vircopy(SELF, BIOS_SEG, bios_vir, SELF, D, dst_vir, bytes)
#define sys_biosout(src_vir, bios_vir, bytes) \
        sys_vircopy(SELF, D, src_vir, SELF, BIOS_SEG, bios_vir, bytes)
#define sys_datacopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, D, src_vir, dst_proc, D, dst_vir, bytes)
#define sys_textcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, T, src_vir, dst_proc, T, dst_vir, bytes)
#define sys_stackcopy(src_proc, src_vir, dst_proc, dst_vir, bytes) \
        sys_vircopy(src_proc, S, src_vir, dst_proc, S, dst_vir, bytes)
_PROTOTYPE(int sys_vircopy, (int src_proc, int src_seg, vir_bytes src_vir,
        int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));

#define sys_abscopy(src_phys, dst_phys, bytes) \
        sys_physcopy(NONE, PHYS_SEG, src_phys, NONE, PHYS_SEG, dst_phys, bytes)
_PROTOTYPE(int sys_physcopy, (int src_proc, int src_seg, vir_bytes src_vir,
        int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes));
_PROTOTYPE(int sys_memset, (unsigned long pattern,
                phys_bytes base, phys_bytes bytes));

/* Vectored virtual / physical copy calls. */
#if DEAD_CODE           /* library part not yet implemented */
_PROTOTYPE(int sys_virvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
_PROTOTYPE(int sys_physvcopy, (phys_cp_req *vec_ptr,int vec_size,int *nr_ok));
#endif

_PROTOTYPE(int sys_umap, (int proc_nr, int seg, vir_bytes vir_addr,
         vir_bytes bytes, phys_bytes *phys_addr));
_PROTOTYPE(int sys_segctl, (int *index, u16_t *seg, vir_bytes *off,
        phys_bytes phys, vir_bytes size));

/* Shorthands for sys_getinfo() system call. */
#define sys_getkmessages(dst)    sys_getinfo(GET_KMESSAGES, dst, 0,0,0)
#define sys_getkinfo(dst)        sys_getinfo(GET_KINFO, dst, 0,0,0)
#define sys_getloadinfo(dst)     sys_getinfo(GET_LOADINFO, dst, 0,0,0)
#define sys_getmachine(dst)      sys_getinfo(GET_MACHINE, dst, 0,0,0)
#define sys_getproctab(dst)      sys_getinfo(GET_PROCTAB, dst, 0,0,0)
#define sys_getprivtab(dst)      sys_getinfo(GET_PRIVTAB, dst, 0,0,0)
#define sys_getproc(dst,nr)      sys_getinfo(GET_PROC, dst, 0,0, nr)
#define sys_getrandomness(dst)   sys_getinfo(GET_RANDOMNESS, dst, 0,0,0)
#define sys_getimage(dst)        sys_getinfo(GET_IMAGE, dst, 0,0,0)
#define sys_getirqhooks(dst)     sys_getinfo(GET_IRQHOOKS, dst, 0,0,0)
#define sys_getirqactids(dst)    sys_getinfo(GET_IRQACTIDS, dst, 0,0,0)
#define sys_getmonparams(v,vl)   sys_getinfo(GET_MONPARAMS, v,vl, 0,0)
#define sys_getschedinfo(v1,v2)  sys_getinfo(GET_SCHEDINFO, v1,0, v2,0)
#define sys_getlocktimings(dst)  sys_getinfo(GET_LOCKTIMING, dst, 0,0,0)
#define sys_getbiosbuffer(virp, sizep) sys_getinfo(GET_BIOSBUFFER, virp, \
        sizeof(*virp), sizep, sizeof(*sizep))
```

```
_PROTOTYPE(int sys_getinfo, (int request, void *val_ptr, int val_len,
                                 void *val_ptr2, int val_len2)          );

/* Signal control. */
_PROTOTYPE(int sys_kill, (int proc, int sig) );
_PROTOTYPE(int sys_sigsend, (int proc_nr, struct sigmsg *sig_ctxt) );
_PROTOTYPE(int sys_sigreturn, (int proc_nr, struct sigmsg *sig_ctxt) );
_PROTOTYPE(int sys_getksig, (int *k_proc_nr, sigset_t *k_sig_map) );
_PROTOTYPE(int sys_endksig, (int proc_nr) );

/* NOTE: two different approaches were used to distinguish the device I/O
 * types 'byte', 'word', 'long': the latter uses #define and results in a
 * smaller implementation, but looses the static type checking.
 */
_PROTOTYPE(int sys_voutb, (pvb_pair_t *pvb_pairs, int nr_ports)         );
_PROTOTYPE(int sys_voutw, (pvw_pair_t *pvw_pairs, int nr_ports)         );
_PROTOTYPE(int sys_voutl, (pvl_pair_t *pvl_pairs, int nr_ports)         );
_PROTOTYPE(int sys_vinb, (pvb_pair_t *pvb_pairs, int nr_ports)          );
_PROTOTYPE(int sys_vinw, (pvw_pair_t *pvw_pairs, int nr_ports)          );
_PROTOTYPE(int sys_vinl, (pvl_pair_t *pvl_pairs, int nr_ports)          );

/* Shorthands for sys_out() system call. */
#define sys_outb(p,v)   sys_out((p), (unsigned long) (v), DIO_BYTE)
#define sys_outw(p,v)   sys_out((p), (unsigned long) (v), DIO_WORD)
#define sys_outl(p,v)   sys_out((p), (unsigned long) (v), DIO_LONG)
_PROTOTYPE(int sys_out, (int port, unsigned long value, int type)       );

/* Shorthands for sys_in() system call. */
#define sys_inb(p,v)    sys_in((p), (v), DIO_BYTE)
#define sys_inw(p,v)    sys_in((p), (v), DIO_WORD)
#define sys_inl(p,v)    sys_in((p), (v), DIO_LONG)
_PROTOTYPE(int sys_in, (int port, unsigned long *value, int type)       );

/* pci.c */
_PROTOTYPE( void pci_init, (void)                                       );
_PROTOTYPE( void pci_init1, (char *name)                                );
_PROTOTYPE( int pci_first_dev, (int *devindp, u16_t *vidp, u16_t *didp) );
_PROTOTYPE( int pci_next_dev, (int *devindp, u16_t *vidp, u16_t *didp)  );
_PROTOTYPE( int pci_find_dev, (U8_t bus, U8_t dev, U8_t func,
                                               int *devindp)            );
_PROTOTYPE( void pci_reserve, (int devind)                              );
_PROTOTYPE( void pci_ids, (int devind, u16_t *vidp, u16_t *didp)        );
_PROTOTYPE( void pci_rescan_bus, (U8_t busnr)                           );
_PROTOTYPE( u8_t pci_attr_r8, (int devind, int port)                    );
_PROTOTYPE( u16_t pci_attr_r16, (int devind, int port)                  );
_PROTOTYPE( u32_t pci_attr_r32, (int devind, int port)                  );
_PROTOTYPE( void pci_attr_w8, (int devind, int port, U8_t value)        );
_PROTOTYPE( void pci_attr_w16, (int devind, int port, U16_t value)      );
_PROTOTYPE( void pci_attr_w32, (int devind, int port, u32_t value)      );
_PROTOTYPE( char *pci_dev_name, (U16_t vid, U16_t did)                  );
_PROTOTYPE( char *pci_slot_name, (int devind)                           );

/* real-time minix 3*/
_PROTOTYPE( int sys_rt_set_sched, (int type, int policy) );
_PROTOTYPE( int sys_rt_set, (int endpoint, int sched, int param1, int param2) );
_PROTOTYPE( int sys_rt_nextperiod, (int endpoint) );
_PROTOTYPE( int sys_rt_sched_bridge, (int state) );
_PROTOTYPE( int sys_rt_show_data, (void) );

/* kernel logger */
_PROTOTYPE( int sys_klog_set, (int state, int type) );
_PROTOTYPE( int sys_klog_copy, (int endpoint, void * data_ptr) );
```

```
#endif /* _SYSLIB_H */
```

Listing 24: /usr/src/kernel/clock.c

```
/* This file contains the clock task, which handles time related functions.
 * Important events that are handled by the CLOCK include setting and
 * monitoring alarm timers and deciding when to (re)schedule processes.
 * The CLOCK offers a direct interface to kernel processes. System services
 * can access its services through system calls, such as sys_setalarm(). The
 * CLOCK task thus is hidden from the outside world.
 *
 * Changes:
 *   May 19, 2009   Support for real-time scheduling (EDF and RM) (Bianco
     Zandbergen)
 *   Oct 08, 2005   reordering and comment editing (A. S. Woodhull)
 *   Mar 18, 2004   clock interface moved to SYSTEM task (Jorrit N. Herder)
 *   Sep 30, 2004   source code documentation updated  (Jorrit N. Herder)
 *   Sep 24, 2004   redesigned alarm timers  (Jorrit N. Herder)
 *
 * The function do_clocktick() is triggered by the clock's interrupt
 * handler when a watchdog timer has expired or a process must be scheduled.
 *
 * In addition to the main clock_task() entry point, which starts the main
 * loop, there are several other minor entry points:
 *   clock_stop:        called just before MINIX shutdown
 *   get_uptime:        get realtime since boot in clock ticks
 *   set_timer:         set a watchdog timer (+)
 *   reset_timer:       reset a watchdog timer (+)
 *   read_clock:        read the counter of channel 0 of the 8253A timer
 *
 * (+) The CLOCK task keeps tracks of watchdog timers for the entire kernel.
 * The watchdog functions of expired timers are executed in do_clocktick().
 * It is crucial that watchdog functions not block, or the CLOCK task may
 * be blocked. Do not send() a message when the receiver is not expecting it.
 * Instead, notify(), which always returns, should be used.
 */

#include "kernel.h"
#include "proc.h"
#include <signal.h>
#include <minix/com.h>
#include <minix/rt.h>
#include <minix/klog.h>

/* Function prototype for PRIVATE functions. */
FORWARD _PROTOTYPE( void init_clock, (void) );
FORWARD _PROTOTYPE( int clock_handler, (irq_hook_t *hook) );
FORWARD _PROTOTYPE( int do_clocktick, (message *m_ptr) );
FORWARD _PROTOTYPE( void load_update, (void));

/* Clock parameters. */
#define COUNTER_FREQ (2*TIMER_FREQ) /* counter frequency using square wave */
#define LATCH_COUNT     0x00    /* cc00xxxx, c = channel, x = any */
#define SQUARE_WAVE     0x36    /* ccaammmb, a = access, m = mode, b = BCD */
                                /*   11x11, 11 = LSB then MSB, x11 = sq wave */
#define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
#define TIMER_FREQ  1193182L    /* clock frequency for timer in PC and AT */

#define CLOCK_ACK_BIT   0x80    /* PS/2 clock interrupt acknowledge bit */

/* The CLOCK's timers queue. The functions in <timers.h> operate on this.
 * Each system process possesses a single synchronous alarm timer. If other
 * kernel parts want to use additional timers, they must declare their own
 * persistent (static) timer structure, which can be passed to the clock
 * via (re)set_timer().
 * When a timer expires its watchdog function is run by the CLOCK task.
 */
PRIVATE timer_t *clock_timers;          /* queue of CLOCK timers */
PRIVATE clock_t next_timeout;           /* realtime that next timer expires */

/* The time is incremented by the interrupt handler on each clock tick. */
PRIVATE clock_t realtime;               /* real time clock */
PRIVATE irq_hook_t clock_hook;          /* interrupt handler hook */

/*===========================================================================*
 *                              clock_task                                   *
 *===========================================================================*/
PUBLIC void clock_task()
{
/* Main program of clock task. If the call is not HARD_INT it is an error.
 */
  message m;                    /* message buffer for both input and output */
  int result;                   /* result returned by the handler */

  init_clock();                 /* initialize clock task */

  /* Main loop of the clock task.  Get work, process it. Never reply. */
  while (TRUE) {

      /* Go get a message. */
      receive(ANY, &m);

      /* Handle the request. Only clock ticks are expected. */
      switch (m.m_type) {
      case HARD_INT:
          result = do_clocktick(&m);    /* handle clock tick */
          break;
      default:                          /* illegal request type */
          kprintf("CLOCK: illegal request %d from %d.\n", m.m_type,m.m_source);
      }
  }
}

/*===========================================================================*
 *                              do_clocktick                                 *
 *===========================================================================*/
PRIVATE int do_clocktick(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
register struct proc **xpp;
register struct proc *temp_p;
/* Despite its name, this routine is not called on every clock tick. It
 * is called on those clock ticks when a lot of work needs to be done.
 */

  /* A process used up a full quantum. The interrupt handler stored this
   * process in 'prev_ptr'.  First make sure that the process is not on the
   * scheduling queues.  Then announce the process ready again. Since it has
   * no more time left, it gets a new quantum and is inserted at the right
   * place in the queues.  As a side-effect a new process will be scheduled.
   */
  if (prev_ptr->p_ticks_left <= 0 && priv(prev_ptr)->s_flags & PREEMPTIBLE) {
      lock_dequeue(prev_ptr);           /* take it off the queues */
      lock_enqueue(prev_ptr);           /* and reinsert it again */
  }
```

```
/* Check if a clock timer expired and run its watchdog function. */
if (next_timeout <= realtime) {
      tmrs_exptimers(&clock_timers, realtime, NULL);
      next_timeout = clock_timers == NULL ?
              TMR_NEVER : clock_timers->tmr_exp_time;
}

/* Unfortunately we don't know what event triggered do_clocktick.
 * If the real-time scheduler is EDF we will have to check for the following
     things:
 * - The current scheduled EDF process has missed the deadline.
 * - The current EDF process has no ticks left in this period.
 * - Check for processes in the wait queue from which a new period starts.
 */
if (rt_sched == SCHED_EDF) {

    /* Check for missed deadline of currently scheduled EDF proc.
     * This is the case if there is a EDF process scheduled AND
     * that process has ticks left in this period AND the number of ticks left
     * till next period is 0. The number of ticks left till next period equals
         the deadline.
     */
    if (edf_rp != NIL_PROC && edf_rp->p_rt_ticksleft > 0 &&
        edf_rp->p_rt_nextperiod == 0) {

        /* Remove EDF process from scheduling queue */
        lock_dequeue(edf_rp);

        /* update total used ticks before ticksleft will be reset */
        edf_rp->p_rt_totalused += (edf_rp->p_rt_calctime -
            edf_rp->p_rt_ticksleft);

        /* New period for this process starts immiately because
         * a deadline is the same as a new period start.
         */
        edf_rp->p_rt_nextperiod = edf_rp->p_rt_period;

        /* New period new calculation time */
        edf_rp->p_rt_ticksleft = edf_rp->p_rt_calctime;

        /* update period counter for stats */
        edf_rp->p_rt_periodnr++;

        /* update deadline counter for stats */
        edf_rp->p_rt_missed_dl++;

        /* update total reserved ticks for stats*/
        edf_rp->p_rt_totalreserved += edf_rp->p_rt_calctime;

        /* Add process to the run queue sorted on deadline.
         * First find the right place in the queue.
         */
        xpp = &edf_run_head;
        while (*xpp != NIL_PROC && (*xpp)->p_rt_nextperiod <=
            edf_rp->p_rt_nextperiod) {
            xpp = &(*xpp)->p_rt_link;
        }

        /* Add process to the run queue */
        edf_rp->p_rt_link = *xpp;
        *xpp = edf_rp;
        /* This process is not scheduled any more */
        edf_rp = NIL_PROC;

        /* Check which process has the earliest deadline and may be scheduled
            now */
        edf_sched();
    }

    /* Check for no ticks left of currently scheduled EDF process.
     * This is the case if there is an EDF process scheduled AND
     * this process has no ticks left in the current period.
     */
    if (edf_rp != NIL_PROC && edf_rp->p_rt_ticksleft <= 0) {

        /* Remove process from scheduling queue */
        lock_dequeue(edf_rp);

        #if 0
        kprintf("do_clocktick: no ticks left: %d\n", edf_rp->p_endpoint);
        #endif

        /* Add process to the wait queue waiting for next period start.
         * First we have to find the right place in the queue.
         * The queue is sorted on next period start.
         */
        xpp = &edf_wait_head;
        while (*xpp != NIL_PROC && (*xpp)->p_rt_nextperiod <=
            edf_rp->p_rt_nextperiod) {
            xpp = &(*xpp)->p_rt_link;
        }

        /* Add process to the wait queue */
        edf_rp->p_rt_link = *xpp;
        *xpp = edf_rp;

        /* Set the NEXTPERIOD bit in the run time flags.
         * A process is only runnable if p_rts_flags == 0.
         * This ensures that no other code (i.e IPC code)
         * will add the process in the scheduling queue while the
         * process is still waiting for the next period start.
         */
        edf_rp->p_rts_flags |= NEXTPERIOD;

        /* This process is not scheduled any more */
        edf_rp = NIL_PROC;

        /* Check which process has the earliest deadline and may be scheduled
            now */
        edf_sched();
    }

    /* check for processes with a new period start.
     * We check only the head of the wait queue because the
     * queue is sorted on next period start. If the head of the queue has
     * p_rt_nextperiod == 0 it will be removed from the wait queue and added
     * to the run queue. We will repeat this process until the head of the
        queue has
     * ticks left till the next period because multiple processes can have a
        new period start
     * at the same time.
     */
    while (edf_wait_head != NIL_PROC &&
              edf_wait_head->p_rt_nextperiod == 0) {

           /* update total ticks used before ticksleft is reset */
           edf_wait_head->p_rt_totalused += (edf_wait_head->p_rt_calctime -
               edf_wait_head->p_rt_ticksleft);

           /* reset the ticks till next period */
           edf_wait_head->p_rt_nextperiod = edf_wait_head->p_rt_period;

           /* reset the calculation time left */
           edf_wait_head->p_rt_ticksleft = edf_wait_head->p_rt_calctime;

           /* This process will be runnable so clear the NEXTPERIOD bit in
            * p_rts_flags.
            */
           edf_wait_head->p_rts_flags &= ~(NEXTPERIOD);

           /* increase period counter for stats */
           edf_wait_head->p_rt_periodnr++;

           /* update total reserved ticks for stats */
           edf_wait_head->p_rt_totalreserved += edf_wait_head->p_rt_calctime;

           /* Add process to the run queue that is sorted on deadline.
            * First we have to find the right place in the list.
            */
           xpp = &edf_run_head;
           while (*xpp != NIL_PROC && (*xpp)->p_rt_nextperiod <=
               edf_wait_head->p_rt_nextperiod) {
               xpp = &(*xpp)->p_rt_link;
           }

           /* Add process to the run queue and
            * remove process from wait queue.
            */
           temp_p = edf_wait_head->p_rt_link; /* save the new head of wait queue
               to temp_p */
           edf_wait_head->p_rt_link = *xpp; /* set the next process pointer of the
               process we add */
           *xpp = edf_wait_head; /* point to the process we add */
           edf_wait_head = temp_p; /* set the new head of the wait queue, we saved
               this in temp_p */

           /* Because we changed the run queue we will have to check
            * which process has the earliest deadline and should be scheduled now.
            */
           edf_sched();

       }

   }

   /* Inhibit sending a reply. */
   return(EDONTREPLY);
}

/*===========================================================================*
 *                              init_clock                                   *
 *===========================================================================*/
PRIVATE void init_clock()
{
  /* Initialize the CLOCK's interrupt hook. */
  clock_hook.proc_nr_e = CLOCK;

  /* Initialize channel 0 of the 8253A timer to, e.g., 60 Hz, and register
   * the CLOCK task's interrupt handler to be run on every clock tick.
   */
  outb(TIMER_MODE, SQUARE_WAVE);        /* set timer to run continuously */
  outb(TIMER0, TIMER_COUNT);            /* load timer low byte */
  outb(TIMER0, TIMER_COUNT >> 8);       /* load timer high byte */
  put_irq_handler(&clock_hook, CLOCK_IRQ, clock_handler);
  enable_irq(&clock_hook);              /* ready for clock interrupts */

  /* Set a watchdog timer to periodically balance the scheduling queues. */
  balance_queues(NULL);                 /* side-effect sets new timer */
}

/*===========================================================================*
 *                              clock_stop                                   *
 *===========================================================================*/
PUBLIC void clock_stop()
{
/* Reset the clock to the BIOS rate. (For rebooting.) */
  outb(TIMER_MODE, 0x36);
  outb(TIMER0, 0);
  outb(TIMER0, 0);
}

/*===========================================================================*
 *                              clock_handler                                *
 *===========================================================================*/
PRIVATE int clock_handler(hook)
irq_hook_t *hook;
{
/* This executes on each clock tick (i.e., every time the timer chip generates
 * an interrupt). It does a little bit of work so the clock task does not have
 * to be called on every tick.  The clock task is called when:
 *
 *      (1) the scheduling quantum of the running process has expired, or
 *      (2) a timer has expired and the watchdog function should be run.
 *
 * Many global global and static variables are accessed here.  The safety of
 * this must be justified. All scheduling and message passing code acquires a
 * lock by temporarily disabling interrupts, so no conflicts with calls from
 * the task level can occur. Furthermore, interrupts are not reentrant, the
 * interrupt handler cannot be bothered by other interrupts.
 *
 * Variables that are updated in the clock's interrupt handler:
 *      lost_ticks:
 *              Clock ticks counted outside the clock task. This for example
 *              is used when the boot monitor processes a real mode interrupt.
 *      realtime:
 *              The current uptime is incremented with all outstanding ticks.
 *      proc_ptr, bill_ptr:
 *              These are used for accounting.  It does not matter if proc.c
 *              is changing them, provided they are always valid pointers,
 *              since at worst the previous process would be billed.
 */
  register unsigned ticks;
  register struct proc *ap; /* used for EDF accounting */
  char call_do_clocktick = 0; /* used for EDF accounting */
  char rq_md = 0; /* has some processed missed deadline in run queue? we need to
      resort in that case */
```

```
/* Acknowledge the PS/2 clock interrupt. */
if (machine.ps_mca) outb(PORT_B, inb(PORT_B) | CLOCK_ACK_BIT);

/* Get number of ticks and update realtime. */
ticks = lost_ticks + 1;
lost_ticks = 0;
realtime += ticks;

/* Update user and system accounting times. Charge the current process for
 * user time. If the current process is not billable, that is, if a non-user
 * process is running, charge the billable process for system time as well.
 * Thus the unbillable process' user time is the billable user's system time.
 */
proc_ptr->p_user_time += ticks;
if (priv(proc_ptr)->s_flags & PREEMPTIBLE) {
    proc_ptr->p_ticks_left -= ticks;
}
if (! (priv(proc_ptr)->s_flags & BILLABLE)) {
    bill_ptr->p_sys_time += ticks;
    bill_ptr->p_ticks_left -= ticks;

    /* real-time processes don't use p_ticks_left
     * but should they be billed too?
     * We currently don't bill them.
     * Remove the comments to do so.
     * Please note that a process can get a negative
     * p_rt_ticksleft if it can be billed.
     */

    /*if (is_rtp(bill_ptr)) {
        bill_ptr->p_rt_ticksleft--;
    }*/
}

/* Update load average. */
load_update();

/* Kernel schedule logging.
 * Keeps track of all processes running when clock interrupt
 * happens. First we check if we have to log.
 */
if (klog_state == 1 && klog_type == KLOG_CLOCKINT && klog_ptr < KLOG_SIZE) {
    klog[klog_ptr].klog_data = proc_ptr->p_nr; /* process number */
    klog[klog_ptr].klog_endpoint = proc_ptr->p_endpoint; /* process endpoint */
    klog[klog_ptr].klog_time = get_uptime(); /* time in ticks since boot */
    strcpy(klog[klog_ptr].klog_name, proc_ptr->p_name);   /* name of process */
    klog_ptr++;

    if (klog_ptr >= KLOG_SIZE) {
        /* kernel log buffer is full.
         * We disable logging by setting klog_state to 0.
         * We set klog_ptr to 0 for a next log start.
         */
        kprintf("klog buffer full!\n");
        klog_state = 0;
        klog_ptr = 0;
    }
}

/* generic accounting for real-time processes */
if (is_rtp(proc_ptr)) {
    /* make sure quantum never reaches zero.
     * We don't use p_ticks_left for RT processes.
     */
    proc_ptr->p_ticks_left = 100;
}

/* if the RT scheduler is EDF
 * we will have to do accounting on
 * all four EDF data structures.
 */
if (rt_sched == SCHED_EDF) {

    /* Do accounting for the currently
     * scheduled EDF process, if there is one.
     */
    if (edf_rp != NIL_PROC) {

        /* decrease ticks left in current period */
        edf_rp->p_rt_ticksleft--;

        if (edf_rp->p_rt_nextperiod > 0) {
            /* decrease ticks till next period (deadline) */
            edf_rp->p_rt_nextperiod--;
        }

        /* check if process has no ticks left
         * in current period AND if the process
         * has missed the deadline. In both cases
         * do_clocktick should be run
         */
        if (edf_rp->p_rt_ticksleft == 0 ||
            edf_rp->p_rt_nextperiod == 0) {
            call_do_clocktick = 1;

            #if EDF_PRINT_MD == 1
            if (edf_rp->p_rt_nextperiod == 0) {
                kprintf("Process %d %s has missed deadline (edf_rp)\n",
                    edf_rp->p_endpoint, edf_rp->p_name);
            }
            #endif
        }
    }

    /* Do accounting for the run queue.
     * We will check all processes in the queue.
     */
    ap = edf_run_head;
    while (ap != NIL_PROC) {
        if (ap->p_rt_nextperiod > 0) {

            /* decrease ticks till next period */
            ap->p_rt_nextperiod--;

            if (ap->p_rt_nextperiod == 0) {
                /* No ticks left till next period but the process
                 * has not used all calculation time.
                 * It missed the deadline. We leave the process in the queue
                 * but we start a new period.
                 */

                /* update total used ticks before ticksleft will be reset */
                ap->p_rt_totalused += (ap->p_rt_calctime - ap->p_rt_ticksleft);
                /* reset ticks left till next period
                 * and calculation time left in current period.
                 */
                ap->p_rt_nextperiod = ap->p_rt_period;
                ap->p_rt_ticksleft = ap->p_rt_calctime;

                /* update period counter for stats */
                ap->p_rt_periodnr++;

                /* update deadline counter for stats */
                ap->p_rt_missed_dl++;

                /* update total reserved ticks for stats */
                ap->p_rt_totalreserved += ap->p_rt_calctime;

                /* we need to resort the queue */
                rq_md = 1;

                #if EDF_PRINT_MD == 1
                kprintf("Process %d %s has missed deadline (RQ)\n",
                    ap->p_endpoint, ap->p_name);
                #endif
            }
        }
        ap = ap->p_rt_link; /* get next process in the queue */
    }

    /* If a process has missed the deadline in the run queue, the deadline
     * will be renewed. In that case we will have to resort the run queue to
     * keep it sorted on deadline. We use a bubble sort algorithm here.
     * The c node precedes the a and e node pointing to the node to which the
     * comparisons are being made.
     */
    if (rq_md != 0) {
        struct proc *a = NIL_PROC;
        struct proc *b = NIL_PROC;
        struct proc *c = NIL_PROC;
        struct proc *e = NIL_PROC;
        struct proc *tmp = NIL_PROC;

        while(e != edf_run_head->p_rt_link) {
            c = a = edf_run_head;
            b = a->p_rt_link;

            while(a != e) {
                if(a->p_rt_nextperiod > b->p_rt_nextperiod) {
                    if(a == edf_run_head) {
                        tmp = b->p_rt_link;
                        b->p_rt_link = a;
                        a->p_rt_link = tmp;
                        edf_run_head = b;
                        c = b;
                    } else {
                        tmp = b->p_rt_link;
                        b->p_rt_link = a;
                        a->p_rt_link = tmp;
                        c->p_rt_link = b;
                        c = b;
                    }
                } else {
                    c = a;
                    a = a->p_rt_link;
                }

                b = a->p_rt_link;
                if(b == e) {
                    e = a;
                }
            }
        }
    }

    /* Do accounting for the wait queue.
     * We will check all processes in the queue.
     */
    ap = edf_wait_head;
    while (ap != NIL_PROC) {
        if (ap->p_rt_nextperiod > 0) {

            /* decrease ticks till next period */
            ap->p_rt_nextperiod--;

            if (ap->p_rt_nextperiod == 0) {
                /* new period start for this process.
                 * We will have to call do_clocktick.
                 */
                call_do_clocktick = 1;
            }
        }
        ap = ap->p_rt_link; /* get next process in the queue */
    }

    /* Do accounting for the block list.
     * We will check all processes in the list.
     */
    ap = edf_block_head;
    while (ap != NIL_PROC) {
        if (ap->p_rt_nextperiod > 0) {

            /* decrease ticks till next period */
            ap->p_rt_nextperiod--;

            if (ap->p_rt_nextperiod == 0) {
                /* No ticks left till next period and the process
                 * has not used all calculation time. The process missed the
                     deadline.
                 * We leave the process in the block list but start a new
                     period.
                 */

                /* update total used ticks before ticksleft will be reset */
                ap->p_rt_totalused += (ap->p_rt_calctime - ap->p_rt_ticksleft);

                /* reset ticks left till next period
                 * and calculation time left in current period.
                 */
                ap->p_rt_nextperiod = ap->p_rt_period;
                ap->p_rt_ticksleft = ap->p_rt_calctime;

                /* update period counter for stats */
                ap->p_rt_periodnr++;

                /* update deadline counter for stats */
```

```
                  ap->p_rt_missed_dl++;

                  /* update total reserved ticks for stats */
                  ap->p_rt_totalreserved += ap->p_rt_calctime;

                  #if EDF_PRINT_MD
                  kprintf("Process %d %s has missed deadline (BQ)\n",
                      ap->p_endpoint, ap->p_name);
                  #endif
              }
          }
          ap = ap->p_rt_link; /* get next process in the list */
      }

      /* done with EDF accounting */
  }

  /* Check if do_clocktick() must be called. Done for alarms and scheduling.
   * Some processes, such as the kernel tasks, cannot be preempted.
   */
  if ((next_timeout <= realtime) || (proc_ptr->p_ticks_left <= 0) ||
      (call_do_clocktick != 0)) {
      prev_ptr = proc_ptr;                      /* store running process */
      lock_notify(HARDWARE, CLOCK);             /* send notification */
  }
  return(1);                                    /* reenable interrupts */
}

/*===========================================================================*
 *                              get_uptime                                   *
 *===========================================================================*/
PUBLIC clock_t get_uptime()
{
/* Get and return the current clock uptime in ticks. */
  return(realtime);
}

/*===========================================================================*
 *                              set_timer                                    *
 *===========================================================================*/
PUBLIC void set_timer(tp, exp_time, watchdog)
struct timer *tp;               /* pointer to timer structure */
clock_t exp_time;               /* expiration realtime */
tmr_func_t watchdog;            /* watchdog to be called */
{
/* Insert the new timer in the active timers list. Always update the
 * next timeout time by setting it to the front of the active list.
 */
  tmrs_settimer(&clock_timers, tp, exp_time, watchdog, NULL);
  next_timeout = clock_timers->tmr_exp_time;
}

/*===========================================================================*
 *                              reset_timer                                  *
 *===========================================================================*/
PUBLIC void reset_timer(tp)
struct timer *tp;               /* pointer to timer structure */
{
/* The timer pointed to by 'tp' is no longer needed. Remove it from both the
 * active and expired lists. Always update the next timeout time by setting
 * it to the front of the active list.
 */
```

```
  tmrs_clrtimer(&clock_timers, tp, NULL);
  next_timeout = (clock_timers == NULL) ?
        TMR_NEVER : clock_timers->tmr_exp_time;
}

/*===========================================================================*
 *                              read_clock                                   *
 *===========================================================================*/
PUBLIC unsigned long read_clock()
{
/* Read the counter of channel 0 of the 8253A timer.  This counter counts
 * down at a rate of TIMER_FREQ and restarts at TIMER_COUNT-1 when it
 * reaches zero. A hardware interrupt (clock tick) occurs when the counter
 * gets to zero and restarts its cycle.
 */
  unsigned count;

  outb(TIMER_MODE, LATCH_COUNT);
  count = inb(TIMER0);
  count |= (inb(TIMER0) << 8);

  return count;
}

/*===========================================================================*
 *                              load_update                                  *
 *===========================================================================*/
PRIVATE void load_update(void)
{
        u16_t slot;
        int enqueued = -1, q;   /* -1: special compensation for IDLE. */
        struct proc *p;

        /* Load average data is stored as a list of numbers in a circular
         * buffer. Each slot accumulates _LOAD_UNIT_SECS of samples of
         * the number of runnable processes. Computations can then
         * be made of the load average over variable periods, in the
         * user library (see getloadavg(3)).
         */
        slot = (realtime / HZ / _LOAD_UNIT_SECS) % _LOAD_HISTORY;
        if(slot != kloadinfo.proc_last_slot) {
                kloadinfo.proc_load_history[slot] = 0;
                kloadinfo.proc_last_slot = slot;
        }

        /* Cumulation. How many processes are ready now? */
        for(q = 0; q < NR_SCHED_QUEUES; q++)
                for(p = rdy_head[q]; p != NIL_PROC; p = p->p_nextready)
                        enqueued++;

        kloadinfo.proc_load_history[slot] += enqueued;

        /* Up-to-dateness. */
        kloadinfo.last_clock = realtime;
}
```

Listing 25: /usr/src/kernel/config.h

```
#ifndef CONFIG_H
#define CONFIG_H

/* This file defines the kernel configuration. It allows to set sizes of some
 * kernel buffers and to enable or disable debugging code, timing features,
 * and individual kernel calls.
 *
 * Changes:
 *   Jul 11, 2005       Created.  (Jorrit N. Herder)
 */

/* In embedded and sensor applications, not all the kernel calls may be
 * needed. In this section you can specify which kernel calls are needed
 * and which are not. The code for unneeded kernel calls is not included in
 * the system binary, making it smaller. If you are not sure, it is best
 * to keep all kernel calls enabled.
 */
#define USE_FORK            1    /* fork a new process */
#define USE_NEWMAP          1    /* set a new memory map */
#define USE_EXEC            1    /* update process after execute */
#define USE_EXIT            1    /* clean up after process exit */
#define USE_TRACE           1    /* process information and tracing */
#define USE_GETKSIG         1    /* retrieve pending kernel signals */
#define USE_ENDKSIG         1    /* finish pending kernel signals */
#define USE_KILL            1    /* send a signal to a process */
#define USE_SIGSEND         1    /* send POSIX-style signal */
#define USE_SIGRETURN       1    /* sys_sigreturn(proc_nr, ctxt_ptr, flags) */
#define USE_ABORT           1    /* shut down MINIX */
#define USE_GETINFO         1    /* retrieve a copy of kernel data */
#define USE_TIMES           1    /* get process and system time info */
#define USE_SETALARM        1    /* schedule a synchronous alarm */
#define USE_DEVIO           1    /* read or write a single I/O port */
#define USE_VDEVIO          1    /* process vector with I/O requests */
#define USE_SDEVIO          1    /* perform I/O request on a buffer */
#define USE_IRQCTL          1    /* set an interrupt policy */
#define USE_SEGCTL          1    /* set up a remote segment */
#define USE_PRIVCTL         1    /* system privileges control */
#define USE_NICE            1    /* change scheduling priority */
#define USE_UMAP            1    /* map virtual to physical address */
#define USE_VIRCOPY         1    /* copy using virtual addressing */
#define USE_VIRVCOPY        1    /* vector with virtual copy requests */
#define USE_PHYSCOPY        1    /* copy using physical addressing */
#define USE_PHYSVCOPY       1    /* vector with physical copy requests */
#define USE_MEMSET          1    /* write char to a given memory area */

/* for real-time */
#define USE_RT_SET_SCHED         1 /* set the real-time scheduler */
#define USE_RT_SET               1 /* transform a normal process to a real-time
    process */
#define USE_RT_SHOW_DATA         1 /* dump RT scheduler info */
#define USE_RT_NEXTPERIOD        1 /* give up calculation time for EDF scheduled
    processes */
#define USE_RT_SET_SCHED_BRIDGE 1 /* set the RT scheduling bridge state */
#define USE_KLOG_SET             1 /* set the kernel log state */
#define USE_KLOG_COPY            1 /* copy kernel log to user application */

/* Length of program names stored in the process table. This is only used
 * for the debugging dumps that can be generated with the IS server. The PM
 * server keeps its own copy of the program name.
 */
```

```
#define P_NAME_LEN         8

/* Kernel diagnostics are written to a circular buffer. After each message,
 * a system server is notified and a copy of the buffer can be retrieved to
 * display the message. The buffers size can safely be reduced.
 */
#define KMESS_BUF_SIZE   256

/* Buffer to gather randomness. This is used to generate a random stream by
 * the MEMORY driver when reading from /dev/random.
 */
#define RANDOM_ELEMENTS   32

/* This section contains defines for valuable system resources that are used
 * by device drivers. The number of elements of the vectors is determined by
 * the maximum needed by any given driver. The number of interrupt hooks may
 * be incremented on systems with many device drivers.
 */
#define NR_IRQ_HOOKS      16            /* number of interrupt hooks */
#define VDEVIO_BUF_SIZE   64            /* max elements per VDEVIO request */
#define VCOPY_VEC_SIZE    16            /* max elements per VCOPY request */

/* How many bytes for the kernel stack. Space allocated in mpx.s. */
#define K_STACK_BYTES   1024

/* This section allows to enable kernel debugging and timing functionality.
 * For normal operation all options should be disabled.
 */
#define DEBUG_SCHED_CHECK  0    /* sanity check of scheduling queues */
#define DEBUG_TIME_LOCKS   0    /* measure time spent in locks */

/* print missed deadlines of EDF scheduler */
#define EDF_PRINT_MD       0

/* print debug info for prioritized IPC */
#define DEBUG_IPC          0

#endif /* CONFIG_H */
```

Listing 26: /usr/src/kernel/proc.h

```
#ifndef PROC_H
#define PROC_H

/* Here is the declaration of the process table.  It contains all process
 * data, including registers, flags, scheduling priority, memory map,
 * accounting, message passing (IPC) information, and so on.
 *
 * Many assembly code routines reference fields in it.  The offsets to these
 * fields are defined in the assembler include file sconst.h.  When changing
 * struct proc, be sure to change sconst.h to match.
 */
#include <minix/com.h>
#include <minix/klog.h>
#include "protect.h"
#include "const.h"
#include "priv.h"

struct proc {
  struct stackframe_s p_reg;    /* process' registers saved in stack frame */

#if (CHIP == INTEL)
  reg_t p_ldt_sel;              /* selector in gdt with ldt base and limit */
  struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments */
#endif

#if (CHIP == M68000)
/* M68000 specific registers and FPU details go here. */
#endif

  proc_nr_t p_nr;               /* number of this process (for fast access) */
  struct priv *p_priv;          /* system privileges structure */
  short p_rts_flags;            /* process is runnable only if zero */
  short p_misc_flags;           /* flags that do suspend the process */

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

  struct mem_map p_memmap[NR_LOCAL_SEGS];   /* memory map (T, D, S) */

  clock_t p_user_time;          /* user time in ticks */
  clock_t p_sys_time;           /* sys time in ticks */

  struct proc *p_nextready;     /* pointer to next ready process */
  struct proc *p_caller_q;      /* head of list of procs wishing to send */
  struct proc *p_q_link;        /* link to next proc wishing to send */
  message *p_messbuf;           /* pointer to passed message buffer */
  int p_getfrom_e;              /* from whom does process want to receive? */
  int p_sendto_e;               /* to whom does process want to send? */

  sigset_t p_pending;           /* bit map for pending kernel signals */

  char p_name[P_NAME_LEN];      /* name of the process, including \0 */

  int p_endpoint;               /* endpoint number, generation-aware */

  /* fields for Real-time */
  char p_rt;                    /* is process real-time? */
  int p_rt_priority;    /* static priority used by RM scheduler */
  int p_rt_period;      /* period of current process, used by EDF scheduler */
  int p_rt_calctime;    /* calculation time in each period, used by EDF scheduler
      */
  int p_rt_nextperiod;  /* ticks till next period start, used by EDF scheduler */
  int p_rt_ticksleft;   /* calculation time left in current period, used by EDF
      scheduler */
  struct proc *p_rt_link; /* used by EDF scheduler to link processes in the EDF
      data structures */

  /* fields for Real-time: EDF statistics */
  unsigned int p_rt_periodnr; /* period counter */
  unsigned int p_rt_totalreserved; /* total reserved ticks */
  unsigned int p_rt_totalused; /* total used ticks */
  unsigned int p_rt_missed_dl; /* missed deadline counter */

#if DEBUG_SCHED_CHECK
  int p_ready, p_found;
#endif
};

/* Bits for the runtime flags. A process is runnable iff p_rts_flags == 0. */
#define SLOT_FREE       0x01    /* process slot is free */
#define NO_MAP          0x02    /* keeps unmapped forked child from running */
#define SENDING         0x04    /* process blocked trying to send */
#define RECEIVING       0x08    /* process blocked trying to receive */
#define SIGNALED        0x10    /* set when new kernel signal arrives */
#define SIG_PENDING     0x20    /* unready while signal being processed */
#define P_STOP          0x40    /* set when process is being traced */
#define NO_PRIV         0x80    /* keep forked system process from running */
#define NO_PRIORITY    0x100    /* process has been stopped */
#define NO_ENDPOINT    0x200    /* process cannot send or receive messages */
#define NEXTPERIOD     0x400   /* EDF process is waiting for next period */

/* Misc flags */
#define REPLY_PENDING   0x01    /* reply to IPC_REQUEST is pending */
#define MF_VM           0x08    /* process uses VM */

/* Scheduling priorities for p_priority. Values must start at zero (highest
 * priority) and increment.  Priorities of the processes in the boot image
 * can be set in table.c. IDLE must have a queue for itself, to prevent low
 * priority user processes to run round-robin with IDLE.
 */
#define NR_SCHED_QUEUES   16    /* MUST equal minimum priority + 1 */
#define TASK_Q                 0        /* highest, used for kernel tasks */
#define MAX_USER_Q          0    /* highest priority for user processes */
#define USER_Q                 8    /* default (should correspond to nice 0) */
#define MIN_USER_Q            14        /* minimum priority for user processes */
#define IDLE_Q                15    /* lowest, only IDLE process goes here */
#define RT_Q                7    /* Queue for real-time processes */

/* Magic process table addresses. */
#define BEG_PROC_ADDR (&proc[0])
#define BEG_USER_ADDR (&proc[NR_TASKS])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])

#define NIL_PROC          ((struct proc *) 0)
#define NIL_SYS_PROC      ((struct proc *) 1)
#define cproc_addr(n)     (&(proc + NR_TASKS)[(n)])
#define proc_addr(n)      (pproc_addr + NR_TASKS)[(n)]
#define proc_nr(p)        ((p)->p_nr)

#define isokprocn(n)      ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
```

```
#define isemptyn(n)       isemptyp(proc_addr(n))
#define isemptyp(p)       ((p)->p_rts_flags == SLOT_FREE)
#define iskernelp(p)      iskerneln((p)->p_nr)
#define iskerneln(n)      ((n) < 0)
#define isuserp(p)        isusern((p)->p_nr)
#define isusern(n)        ((n) >= 0)

/* The process table and pointers to process table slots. The pointers allow
 * faster access because now a process entry can be found by indexing the
 * pproc_addr array, while accessing an element i requires a multiplication
 * with sizeof(struct proc) to determine the address.
 */
EXTERN struct proc proc[NR_TASKS + NR_PROCS];   /* process table */
EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
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 */

/* macro for checking if a process is real-time */
#define is_rtp(p) ((p)->p_rt != 0)

/* variables for RT scheduling and kernel logging */

/* RT scheduler */
EXTERN int rt_sched;

/* scheduling bridge state */
EXTERN int rt_sched_bridge;

/* priority policy for Rate-Monotonic scheduling.
 * Priority can be enforced to be unique.
 */
EXTERN int rm_prio_policy;

/* current scheduled EDF process */
EXTERN struct proc *edf_rp;

/* Head of EDF processes that are ready */
EXTERN struct proc *edf_run_head;

/* Head of EDF processes that are waiting till next period starts */
EXTERN struct proc *edf_wait_head;

/* Head of EDF processes that are blocked and still have calculation time left */
EXTERN struct proc *edf_block_head;

/* kernel log state. If 1 log. */
EXTERN int klog_state;

/* what to log? */
EXTERN int klog_type;

/* kernel log */
EXTERN struct klog_entry klog[KLOG_SIZE];

/* Pointer to next entry in kernel log */
EXTERN int klog_ptr;

#endif /* PROC_H */
```

Listing 27: /usr/src/kernel/proc.c

```
/* This file contains essentially all of the process and message handling.
 * Together with "mpx.s" it forms the lowest layer of the MINIX kernel.
 * There is one entry point from the outside:
 *
 *   sys_call:        a system call, i.e., the kernel is trapped with an INT
 *
 * As well as several entry points used from the interrupt and task level:
 *
 *   lock_notify:     notify a process of a system event
 *   lock_send:       send a message to a process
 *   lock_enqueue:    put a process on one of the scheduling queues
 *   lock_dequeue:    remove a process from the scheduling queues
 *   edf_rp:          check which process with earliest deadline may now be
     scheduled (EDF scheduler only)
 *   show_rt_data:    dump schedule info for RM and EDF
 *
 * Changes:
 *   May 19, 2009     added support for EDF and RM scheduling, prioritized IPC
     (Bianco Zandbergen)
 *   Aug 19, 2005     rewrote scheduling code  (Jorrit N. Herder)
 *   Jul 25, 2005     rewrote system call handling  (Jorrit N. Herder)
 *   May 26, 2005     rewrote message passing functions  (Jorrit N. Herder)
 *   May 24, 2005     new notification system call  (Jorrit N. Herder)
 *   Oct 28, 2004     nonblocking send and receive calls  (Jorrit N. Herder)
 *
 * The code here is critical to make everything work and is important for the
 * overall performance of the system. A large fraction of the code deals with
 * list manipulation. To make this both easy to understand and fast to execute
 * pointer pointers are used throughout the code. Pointer pointers prevent
 * exceptions for the head or tail of a linked list.
 *
 *  node_t *queue, *new_node;   // assume these as global variables
 *  node_t **xpp = &queue;      // get pointer pointer to head of queue
 *  while (*xpp != NULL)        // find last pointer of the linked list
 *      xpp = &(*xpp)->next;    // get pointer to next pointer
 *  *xpp = new_node;            // now replace the end (the NULL pointer)
 *  new_node->next = NULL;      // and mark the new end of the list
 *
 * For example, when adding a new node to the end of the list, one normally
 * makes an exception for an empty list and looks up the end of the list for
 * nonempty lists. As shown above, this is not required with pointer pointers.
 */

#include <minix/com.h>
#include <minix/callnr.h>
#include <minix/endpoint.h>
#include "debug.h"
#include "kernel.h"
#include "proc.h"
#include <signal.h>
#include <string.h>
#include <minix/rt.h>

/* Scheduling and message passing functions. The functions are available to
 * other parts of the kernel through lock_...(). The lock temporarily disables
 * interrupts to prevent race conditions.
 */
FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dst_e,
                message *m_ptr, unsigned flags));
FORWARD _PROTOTYPE( int mini_receive, (struct proc *caller_ptr, int src,
                message *m_ptr, unsigned flags));
FORWARD _PROTOTYPE( int mini_notify, (struct proc *caller_ptr, int dst));
FORWARD _PROTOTYPE( int deadlock, (int function,
                register struct proc *caller, int src_dst));
FORWARD _PROTOTYPE( void enqueue, (struct proc *rp));
FORWARD _PROTOTYPE( void dequeue, (struct proc *rp));
FORWARD _PROTOTYPE( void sched, (struct proc *rp, int *queue, int *front));
FORWARD _PROTOTYPE( void pick_proc, (void));

#define BuildMess(m_ptr, src, dst_ptr) \
        (m_ptr)->m_source = proc_addr(src)->p_endpoint;         \
        (m_ptr)->m_type = NOTIFY_FROM(src);                             \
        (m_ptr)->NOTIFY_TIMESTAMP = get_uptime();                       \
        switch (src) {                                                  \
        case HARDWARE:                                                  \
                (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_int_pending;     \
                priv(dst_ptr)->s_int_pending = 0;                       \
                break;                                                  \
        case SYSTEM:                                                    \
                (m_ptr)->NOTIFY_ARG = priv(dst_ptr)->s_sig_pending;     \
                priv(dst_ptr)->s_sig_pending = 0;                       \
                break;                                                  \
        }

#if (CHIP == INTEL)
#define CopyMess(s,sp,sm,dp,dm) \
        cp_mess(proc_addr(s)->p_endpoint, \
                (sp)->p_memmap[D].mem_phys,     \
                (vir_bytes)sm, (dp)->p_memmap[D].mem_phys, (vir_bytes)dm)
#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 does not have cp_mess() in assembly like INTEL. Declare prototype
 * for cp_mess() here and define the function below. Also define CopyMess.
 */
#endif /* (CHIP == M68000) */

/*===========================================================================*
 *                              sys_call                                     *
 *===========================================================================*/
PUBLIC int sys_call(call_nr, src_dst_e, m_ptr, bit_map)
int call_nr;                    /* system call number and flags */
int src_dst_e;                  /* src to receive from or dst to send to */
message *m_ptr;                 /* pointer to message in the caller's space */
long bit_map;                   /* notification event set or flags */
{
/* System calls are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by 'proc_ptr'.
 */
  register struct proc *caller_ptr = proc_ptr;  /* get pointer to caller */
  int function = call_nr & SYSCALL_FUNC;        /* get system call function */
  unsigned flags = call_nr & SYSCALL_FLAGS;     /* get flags */
  int mask_entry;                               /* bit to check in send mask */
  int group_size;                               /* used for deadlock check */
  int result;                                   /* the system call's result */
  int src_dst;
  vir_clicks vlo, vhi;          /* virtual clicks containing message to send */

#if 0
  if (caller_ptr->p_rts_flags & SLOT_FREE)
  {
        kprintf("called by the dead?!?\n");
        return EINVAL;
  }
#endif

  /* Require a valid source and/ or destination process, unless echoing. */
  if (src_dst_e != ANY && function != ECHO) {
      if(!isokendpt(src_dst_e, &src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("sys_call: trap %d by %d with bad endpoint %d\n",
              function, proc_nr(caller_ptr), src_dst_e);
#endif
          return EDEADSRCDST;
      }
  } else src_dst = src_dst_e;

  /* Check if the process has privileges for the requested call. Calls to the
   * kernel may only be SENDREC, because tasks always reply and may not block
   * if the caller doesn't do receive().
   */
  if (! (priv(caller_ptr)->s_trap_mask & (1 << function)) ||
          (iskerneln(src_dst) && function != SENDREC
           && function != RECEIVE)) {
#if DEBUG_ENABLE_IPC_WARNINGS
      kprintf("sys_call: trap %d not allowed, caller %d, src_dst %d\n",
          function, proc_nr(caller_ptr), src_dst);
#endif
      return(ETRAPDENIED);              /* trap denied by mask or kernel */
  }

  /* If the call involves a message buffer, i.e., for SEND, RECEIVE, SENDREC,
   * or ECHO, check the message pointer. This check allows a message to be
   * anywhere in data or stack or gap. It will have to be made more elaborate
   * for machines which don't have the gap mapped.
   */
  if (function & CHECK_PTR) {
      vlo = (vir_bytes) m_ptr >> CLICK_SHIFT;
      vhi = ((vir_bytes) m_ptr + MESS_SIZE - 1) >> CLICK_SHIFT;
      if (vlo < caller_ptr->p_memmap[D].mem_vir || vlo > vhi ||
              vhi >= caller_ptr->p_memmap[S].mem_vir +
              caller_ptr->p_memmap[S].mem_len) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("sys_call: invalid message pointer, trap %d, caller %d\n",
                function, proc_nr(caller_ptr));
#endif
          return(EFAULT);               /* invalid message pointer */
      }
  }

  /* If the call is to send to a process, i.e., for SEND, SENDREC or NOTIFY,
   * verify that the caller is allowed to send to the given destination.
   */
  if (function & CHECK_DST) {
      if (! get_sys_bit(priv(caller_ptr)->s_ipc_to, nr_to_id(src_dst))) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("sys_call: ipc mask denied trap %d from %d to %d\n",
                function, proc_nr(caller_ptr), src_dst);
#endif
          return(ECALLDENIED);          /* call denied by ipc mask */
      }
  }
  /* Check for a possible deadlock for blocking SEND(REC) and RECEIVE. */
  if (function & CHECK_DEADLOCK) {
      if (group_size = deadlock(function, caller_ptr, src_dst)) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("sys_call: trap %d from %d to %d deadlocked, group size %d\n",
              function, proc_nr(caller_ptr), src_dst, group_size);
#endif
          return(ELOCKED);
      }
  }

  /* Now check if the call is known and try to perform the request. The only
   * system calls that exist in MINIX are sending and receiving messages.
   *   - SENDREC: combines SEND and RECEIVE in a single system call
   *   - SEND:    sender blocks until its message has been delivered
   *   - RECEIVE: receiver blocks until an acceptable message has arrived
   *   - NOTIFY:  nonblocking call; deliver notification or mark pending
   *   - ECHO:    nonblocking call; directly echo back the message
   */
  switch(function) {
  case SENDREC:
      /* A flag is set so that notifications cannot interrupt SENDREC. */
      caller_ptr->p_misc_flags |= REPLY_PENDING;
      /* fall through */
  case SEND:
      result = mini_send(caller_ptr, src_dst_e, m_ptr, flags);
      if (function == SEND || result != OK) {
          break;                                  /* done, or SEND failed */
      }                                           /* fall through for SENDREC */
  case RECEIVE:
      if (function == RECEIVE)
          caller_ptr->p_misc_flags &= ~REPLY_PENDING;
      result = mini_receive(caller_ptr, src_dst_e, m_ptr, flags);
      break;
  case NOTIFY:
      result = mini_notify(caller_ptr, src_dst);
      break;
  case ECHO:
      CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, caller_ptr, m_ptr);
      result = OK;
      break;
  default:
      result = EBADCALL;                          /* illegal system call */
  }

  /* Now, return the result of the system call to the caller. */
  return(result);
}

/*===========================================================================*
 *                              deadlock                                     *
 *===========================================================================*/
PRIVATE int deadlock(function, cp, src_dst)
int function;                                     /* trap number */
register struct proc *cp;                         /* pointer to caller */
int src_dst;                                      /* src or dst process */
{
/* Check for deadlock. This can happen if 'caller_ptr' and 'src_dst' have
 * a cyclic dependency of blocking send and receive calls. The only cyclic
 * depency that is not fatal is if the caller and target directly SEND(REC)
 * and RECEIVE to each other. If a deadlock is found, the group size is
 * returned. Otherwise zero is returned.
 */
  register struct proc *xp;                     /* process pointer */
  int group_size = 1;                           /* start with only caller */
  int trap_flags;

  while (src_dst != ANY) {                      /* check while process nr */
      int src_dst_e;
      xp = proc_addr(src_dst);                  /* follow chain of processes */
      group_size ++;                            /* extra process in group */

      /* Check whether the last process in the chain has a dependency. If it
       * has not, the cycle cannot be closed and we are done.
       */
      if (xp->p_rts_flags & RECEIVING) {        /* xp has dependency */
          if(xp->p_getfrom_e == ANY) src_dst = ANY;
          else okendpt(xp->p_getfrom_e, &src_dst);
      } else if (xp->p_rts_flags & SENDING) {   /* xp has dependency */
          okendpt(xp->p_sendto_e, &src_dst);
      } else {
          return(0);                            /* not a deadlock */
      }

      /* Now check if there is a cyclic dependency. For group sizes of two,
       * a combination of SEND(REC) and RECEIVE is not fatal. Larger groups
       * or other combinations indicate a deadlock.
       */
      if (src_dst == proc_nr(cp)) {             /* possible deadlock */
          if (group_size == 2) {                /* caller and src_dst */
              /* The function number is magically converted to flags. */
              if ((xp->p_rts_flags ^ (function << 2)) & SENDING) {
                  return(0);                    /* not a deadlock */
              }
          }
          return(group_size);                   /* deadlock found */
      }
  }
  return(0);                                    /* not a deadlock */
}

/*===========================================================================*
 *                              mini_send                                    *
 *===========================================================================*/
PRIVATE int mini_send(caller_ptr, dst_e, m_ptr, flags)
register struct proc *caller_ptr;       /* who is trying to send a message? */
int dst_e;                              /* to whom is message being sent? */
message *m_ptr;                         /* pointer to message buffer */
unsigned flags;                         /* system call flags */
{
/* Send a message from 'caller_ptr' to 'dst'. If 'dst' is blocked waiting
 * for this message, copy the message to it and unblock 'dst'. If 'dst' is
 * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
 */
  register struct proc *dst_ptr;
  register struct proc **xpp;
  int dst_p;

  dst_p = _ENDPOINT_P(dst_e);
  dst_ptr = proc_addr(dst_p);

  if (dst_ptr->p_rts_flags & NO_ENDPOINT) return EDSTDIED;

  /* Check if 'dst' is blocked waiting for this message. The destination's
```

```
   * SENDING flag may be set when its SENDREC call blocked while sending.
   */
  if ( (dst_ptr->p_rts_flags & (RECEIVING | SENDING)) == RECEIVING &&
       (dst_ptr->p_getfrom_e == ANY
         || dst_ptr->p_getfrom_e == caller_ptr->p_endpoint)) {
        /* Destination is indeed waiting for this message. */
        CopyMess(caller_ptr->p_nr, caller_ptr, m_ptr, dst_ptr,
                 dst_ptr->p_messbuf);
        if ((dst_ptr->p_rts_flags &= ~RECEIVING) == 0) {

        if (! is_rtp(dst_ptr) || rt_sched == SCHED_RM) {
            /* destination process is not real-time or real-time and using
             * Rate-Monotonic scheduling. In this case we use the normale
             * procedure which is simply add the process to the scheduling queue.
             */
            enqueue(dst_ptr);

        } else if (is_rtp(dst_ptr) && rt_sched == SCHED_EDF) {
            /* destination is a real-time process scheduled using EDF.
             * We will have to remove the process from the block list and
             * add it to the run queue. Afterwards we check which EDF process
             * has the earliest deadline and may now be scheduled.
             */

            /* Remove destination process from the block list.
             * We first have to find the process in the list.
             */
            xpp = &edf_block_head;
            while (*xpp != NIL_PROC && *xpp != dst_ptr) {
                xpp = &(*xpp)->p_rt_link;
            }

            /* *xpp == NIL_PROC if the list was empty or
             * if the list was not empty and the process was not found.
             */
            if (*xpp != NIL_PROC) {
                /* remove destination process from the block list.
                 * We do this to change the pointer that points to the
                     destination process
                 * to the next process pointer of the destination process.
                 */
                *xpp = (*xpp)->p_rt_link;
            }

            /* Add dst to EDF run queue which is sorted on deadline.
             * We first have to find the right place in the list. */
            xpp = &edf_run_head;
            while (*xpp != NIL_PROC && (*xpp)->p_rt_nextperiod <=
                dst_ptr->p_rt_nextperiod) {
                xpp = &(*xpp)->p_rt_link;
            }

            /* Add process to the run queue */
            dst_ptr->p_rt_link = *xpp;
            *xpp = dst_ptr;

            /* Check which process has the earliest deadline and
             * should be scheduled now.
             */
            edf_sched();

        } else {
            panic("mini_send() failed\n", NO_NUM);
        }
    }
} else if ( ! (flags & NON_BLOCKING)) {
        /* Destination is not waiting.  Block and dequeue caller. */
        caller_ptr->p_messbuf = m_ptr;
        if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
        caller_ptr->p_rts_flags |= SENDING;
        caller_ptr->p_sendto_e = dst_e;

    #if DEBUG_IPC == 1
    xpp = &dst_ptr->p_caller_q;
    if (*xpp != NIL_PROC) {
        kprintf(" ---\n");
        kprintf("IPC: recv Q bef. %s:", dst_ptr->p_name);
        while (*xpp != NIL_PROC) {
            kprintf(" [%d:%s]", (*xpp)->p_priority, (*xpp)->p_name);
            xpp = &(*xpp)->p_q_link;
        }
        kprintf("\n");
    }
    #endif

    /* Process is now blocked.  Put in on the destination's queue.
     * We use this queue as a prioritized queue. The queue is ordered
     * on p_priority which is the minix scheduling priority. This way
     * processes with a higher priority will be served earlier.
     * First we have to find the right place in the queue.
     */
        xpp = &dst_ptr->p_caller_q;
        while (*xpp != NIL_PROC && (*xpp)->p_priority <= caller_ptr->p_priority) {

        if (is_rtp(caller_ptr) && rt_sched == SCHED_RM) {
            /* The while loop will exit if the next process in the list has a
             * lower minix priority (higher p_priority value).
             * if multiple processes with the same priority
             * are in the queue it will add the process to the end, just before
                 the process
             * with a higher priority.
             * Processes scheduled using Rate-Monotonic share the same minix
                 priority.
             * The priority among multiple RM processes is defined by
                 p_rt_priority.
             * Because of this a RM process should not always be added to the end
                 of
             * the processes with the same priority. Instead one has to look at
                 the p_rt_priority
             * and if the next process is a RM process with a lower RM priority
                 (higher p_rt_priority value)
             * we will break the while loop.
             */
            if (is_rtp(*xpp) && (*xpp)->p_rt_priority >
                caller_ptr->p_rt_priority) {
                break;
            }
        }

        xpp = &(*xpp)->p_q_link;
    }

    /* Add process to the destination's queue */
    caller_ptr->p_q_link = *xpp;
    *xpp = caller_ptr;

    #if DEBUG_IPC == 1
    xpp = &dst_ptr->p_caller_q;
    if ( (*xpp)->p_q_link != NIL_PROC ) {
        kprintf("IPC: recv Q aft. %s:", dst_ptr->p_name);
        while (*xpp != NIL_PROC) {
            kprintf(" [%d:%s]", (*xpp)->p_priority, (*xpp)->p_name);
            xpp = &(*xpp)->p_q_link;
        }
        kprintf("\n");
    }
    #endif

    /* if the caller is a real-time process scheduled using EDF
     * we have to add the caller to the EDF block list.
     */
    if (is_rtp(caller_ptr) && rt_sched == SCHED_EDF) {

        /* First we search the right place in the list.
         * This is not very important and we could have
         * just add it to the front. In this case we add
         * the process to the end of the list.
         */
        xpp = &edf_block_head;
        while (*xpp != NIL_PROC) {
            xpp = &(*xpp)->p_rt_link;
        }

        /* Add process to the EDF block list */
        caller_ptr->p_rt_link = *xpp;
        *xpp = caller_ptr;

        /* process is removed from scheduling queue */
        edf_rp = NIL_PROC;

        /* Check which process with earliest deadline may be scheduled now */
        edf_sched();
    }
  } else {
        return(ENOTREADY);
  }
  return(OK);
}


/*===========================================================================*
 *                              mini_receive                                 *
 *===========================================================================*/
PRIVATE int mini_receive(caller_ptr, src_e, m_ptr, flags)
register struct proc *caller_ptr;       /* process trying to get message */
int src_e;                              /* which message source is wanted */
message *m_ptr;                         /* pointer to message buffer */
unsigned flags;                         /* system call flags */
{
/* A process or task wants to get a message.  If a message is already queued,
 * acquire it and deblock the sender.  If no message from the desired source
 * is available block the caller, unless the flags don't allow blocking.
 */
  register struct proc **xpp;
  register struct notification **ntf_q_pp;
  message m;
  int bit_nr;
  sys_map_t *map;
  bitchunk_t *chunk;
  int i, src_id, src_proc_nr, src_p;

  if(src_e == ANY) src_p = ANY;
  else
  {
        okendpt(src_e, &src_p);
        if (proc_addr(src_p)->p_rts_flags & NO_ENDPOINT) return ESRCDIED;
  }


  /* Check to see if a message from desired source is already available.
   * The caller's SENDING flag may be set if SENDREC couldn't send. If it is
   * set, the process should be blocked.
   */
  if (!(caller_ptr->p_rts_flags & SENDING)) {

    /* Check if there are pending notifications, except for SENDREC. */
    if (! (caller_ptr->p_misc_flags & REPLY_PENDING)) {

        map = &priv(caller_ptr)->s_notify_pending;
        for (chunk=&map->chunk[0]; chunk<&map->chunk[NR_SYS_CHUNKS]; chunk++) {

            /* Find a pending notification from the requested source. */
            if (! *chunk) continue;                     /* no bits in chunk */
            for (i=0; ! (*chunk & (1<<i)); ++i) {}      /* look up the bit */
            src_id = (chunk - &map->chunk[0]) * BITCHUNK_BITS + i;
            if (src_id >= NR_SYS_PROCS) break;          /* out of range */
            src_proc_nr = id_to_nr(src_id);             /* get source proc */
#if DEBUG_ENABLE_IPC_WARNINGS
            if(src_proc_nr == NONE) {
                kprintf("mini_receive: sending notify from NONE\n");
            }
#endif
            if (src_e!=ANY && src_p != src_proc_nr) continue;/* source not ok */
            *chunk &= ~(1 << i);                        /* no longer pending */

            /* Found a suitable source, deliver the notification message. */
            BuildMess(&m, src_proc_nr, caller_ptr);     /* assemble message */
            CopyMess(src_proc_nr, proc_addr(HARDWARE), &m, caller_ptr, m_ptr);
            return(OK);                                 /* report success */
        }
    }

    /* Check caller queue. Use pointer pointers to keep code simple. */
    xpp = &caller_ptr->p_caller_q;
    while (*xpp != NIL_PROC) {
        if (src_e == ANY || src_p == proc_nr(*xpp)) {
#if 0
            if ((*xpp)->p_rts_flags & SLOT_FREE)
            {
                kprintf("listening to the dead?!?\n");
                return EINVAL;
            }
#endif

            /* Found acceptable message. Copy it and update status. */
            CopyMess((*xpp)->p_nr, *xpp, (*xpp)->p_messbuf, caller_ptr, m_ptr);
            if (((*xpp)->p_rts_flags &= ~SENDING) == 0) enqueue(*xpp);
            *xpp = (*xpp)->p_q_link;            /* remove from queue */
            return(OK);                         /* report success */
        }
        xpp = &(*xpp)->p_q_link;                /* proceed to next */
    }
  }

  /* No suitable message is available or the caller couldn't send in SENDREC.
   * Block the process trying to receive, unless the flags tell otherwise.
   */
  if ( ! (flags & NON_BLOCKING)) {
      caller_ptr->p_getfrom_e = src_e;
      caller_ptr->p_messbuf = m_ptr;
      if (caller_ptr->p_rts_flags == 0) dequeue(caller_ptr);
      caller_ptr->p_rts_flags |= RECEIVING;

      /* if the caller is a real-time process scheduled
       * using EDF, we put this process on the edf block list.
       * The order of the block list is not important. In this case
       * we add the process to the end of the list.
       */
      if (is_rtp(caller_ptr) && rt_sched == SCHED_EDF) {

          /* find end of block list */
          xpp = &edf_block_head;
          while (*xpp != NIL_PROC) {
              xpp = &(*xpp)->p_rt_link;
          }

          /* Add process to the block list */
          caller_ptr->p_rt_link = *xpp;
          *xpp = caller_ptr;

          /* Process is not scheduled any more */
          edf_rp = NIL_PROC;

          /* Check if an EDF  process with earliest deadline can be scheduled */
          edf_sched();
      }

      return(OK);
  } else {
      return(ENOTREADY);
  }
}

/*===========================================================================*
 *                              mini_notify                                  *
 *===========================================================================*/
PRIVATE int mini_notify(caller_ptr, dst)
register struct proc *caller_ptr;       /* sender of the notification */
int dst;                                /* which process to notify */
{
  register struct proc *dst_ptr = proc_addr(dst);
  int src_id;                           /* source id for late delivery */
  message m;                            /* the notification message */

  /* Check to see if target is blocked waiting for this message. A process
   * can be both sending and receiving during a SENDREC system call.
   */
  if ((dst_ptr->p_rts_flags & (RECEIVING|SENDING)) == RECEIVING &&
      ! (dst_ptr->p_misc_flags & REPLY_PENDING) &&
      (dst_ptr->p_getfrom_e == ANY ||
      dst_ptr->p_getfrom_e == caller_ptr->p_endpoint)) {

      /* Destination is indeed waiting for a message. Assemble a notification
       * message and deliver it. Copy from pseudo-source HARDWARE, since the
       * message is in the kernel's address space.
       */
      BuildMess(&m, proc_nr(caller_ptr), dst_ptr);
      CopyMess(proc_nr(caller_ptr), proc_addr(HARDWARE), &m,
          dst_ptr, dst_ptr->p_messbuf);
      dst_ptr->p_rts_flags &= ~RECEIVING;       /* deblock destination */
      if (dst_ptr->p_rts_flags == 0) enqueue(dst_ptr);
      return(OK);
  }

  /* Destination is not ready to receive the notification. Add it to the
   * bit map with pending notifications. Note the indirectness: the system id
   * instead of the process number is used in the pending bit map.
   */
  src_id = priv(caller_ptr)->s_id;
  set_sys_bit(priv(dst_ptr)->s_notify_pending, src_id);
  return(OK);
}

/*===========================================================================*
 *                              lock_notify                                  *
 *===========================================================================*/
PUBLIC int lock_notify(src_e, dst_e)
int src_e;                      /* (endpoint) sender of the notification */
int dst_e;                      /* (endpoint) who is to be notified */
{
/* Safe gateway to mini_notify() for tasks and interrupt handlers. The sender
 * is explicitely given to prevent confusion where the call comes from. MINIX
 * kernel is not reentrant, which means to interrupts are disabled after
 * the first kernel entry (hardware interrupt, trap, or exception). Locking
 * is done by temporarily disabling interrupts.
 */
  int result, src, dst;

  if(!isokendpt(src_e, &src) || !isokendpt(dst_e, &dst))
        return EDEADSRCDST;

  /* Exception or interrupt occurred, thus already locked. */
  if (k_reenter >= 0) {
      result = mini_notify(proc_addr(src), dst);
  }

  /* Call from task level, locking is required. */
  else {
      lock(0, "notify");
      result = mini_notify(proc_addr(src), dst);
      unlock(0);
  }
  return(result);
}

/*===========================================================================*
 *                              enqueue                                      *
 *===========================================================================*/
PRIVATE void enqueue(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 */
  register struct proc **xpp;

#if DEBUG_SCHED_CHECK
  check_runqueues("enqueue");
  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(! is_rtp(rp) || (is_rtp(rp) && rt_sched == SCHED_EDF)) {

      /* process is not real-time or process is real-time and
       * scheduled using EDF.
       */
      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 */
      }
  } else if (is_rtp(rp) && rt_sched == SCHED_RM) {

      /* process is real-time and scheduled using
       * Rate-monotic. The queue used by Rate-Monotic is ordered by p_rt_priority
       * ascending. The lower the priority number the higher the priority.
       * We will have to search the queue for the right place to insert this
          process.
       */
      xpp = &rdy_head[q];

      /* find the right place in the scheduling queue */
      while (*xpp != NIL_PROC && (*xpp)->p_rt_priority <= rp->p_rt_priority) {
          xpp = &(*xpp)->p_nextready;
      }

      /* insert process in the scheduling queue */
      rp->p_nextready = *xpp;
      *xpp = rp;

      /* Even though we don't use the tail at all,
       * we want to keep the scheduler debug routines happy.
       */
      if (rp->p_nextready == NIL_PROC) {
          rdy_tail[q] = rp;
      }
  } else {
      panic("Unknown scheduler type in enqueue()", NO_NUM);
  }
  /* Now select the next process to run. */
  pick_proc();

#if DEBUG_SCHED_CHECK
  rp->p_ready = 1;
  check_runqueues("enqueue");
#endif
}

/*===========================================================================*
 *                              dequeue                                      *
 *===========================================================================*/
PRIVATE void dequeue(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;

  /* Side-effect for kernel: check if the task's stack still is ok? */
  if (iskernelp(rp)) {
        if (*priv(rp)->s_stack_guard != STACK_GUARD)
                panic("stack overrun by task", proc_nr(rp));
  }

#if DEBUG_SCHED_CHECK
  check_runqueues("dequeue");
  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 */
  }

#if DEBUG_SCHED_CHECK
  rp->p_ready = 0;
  check_runqueues("dequeue");
#endif
}

/*===========================================================================*
 *                              sched                                        *
 *===========================================================================*/
```

```
PRIVATE void sched(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. Only check when the process is NOT real-time!
   */
  if (! time_left && !is_rtp(rp)) {                             /* 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 */


          /* We don't want processes that are lowered in priority
           * end up in the real-time queue. There are two policies:
           * allow processes to get a lower priority than RT processes
           * or not. rt_sched_bridge holds this policy. Before the system shuts
           * down we will have to allow processes to get a lower priority than
           * RT processes otherwise the system will not shutdown properly.
           */
          if (rp->p_priority == RT_Q) {
              if (! rt_sched_bridge) {
                  /* We don't allow processes that have a higher priority than
                   * RT processes to get a lower priority than RT processes.
                   * The lowest priority is the priority above the RT priority.
                   * Please note that the lower the priority number the higher the
                   * priority.
                   */
                  rp->p_priority = RT_Q - 1;
              } else {
                  /* We allow processes with a high priority that is lowered in
                   * priority to get a lower priority than RT processes.
                   * The highest process below RT processes is USER_Q.
                   */
                  rp->p_priority = USER_Q;
              }
          }
      }

  }

  /* 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;
}

/*===========================================================================*
 *                              pick_proc                                    *
 *===========================================================================*/
PRIVATE void pick_proc()
```

```
{
/* 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 */
  static int prev_logged = NONE;

  /* 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 */

          /* Kernel schedule logging which
           * keeps track of all context switches
           * First check if we have to log.
           */
          if (klog_state == 1 && klog_type == KLOG_CONTEXTSWITCH && klog_ptr <
              KLOG_SIZE) {

              /* We hold the previous logged process in prev_logged because we
               * dont want double entries from the clock and system task.
               * These tasks can use dequeue and enqueue functions which call
                 pick_proc.
               * We want to avoid those 'fake' entries.
               */
              if (! ((rp->p_endpoint == CLOCK || rp->p_endpoint == SYSTEM) &&
                     (rp->p_endpoint == prev_logged))) {

                  klog[klog_ptr].klog_data = rp->p_nr; /* save process number */
                  klog[klog_ptr].klog_endpoint = rp->p_endpoint; /* save endpoint
                      */
                  klog[klog_ptr].klog_time = get_uptime(); /* save current time
                      in ticks since boot*/

                  strcpy(klog[klog_ptr].klog_name, rp->p_name); /* save process
                      name */

                  prev_logged = rp->p_endpoint; /* set the prev_logged var */
                  klog_ptr++; /* increase kernel log pointer */

                  if (klog_ptr >= KLOG_SIZE) {
                      /* Kernel log buffer is full.
                       * We disable logging by setting klog_state to 0.
                       * We also reset klog_ptr and prev_logged.
                       */
                      kprintf("klog buffer full!\n");
                      klog_state = 0;
                      klog_ptr = 0;
                      prev_logged = NONE;
                  }
              }
          }

          return;
      }
  }
}

/*===========================================================================*
 *                              balance_queues                               *
 *===========================================================================*/
#define Q_BALANCE_TICKS  100
PUBLIC void balance_queues(tp)
timer_t *tp;                                    /* watchdog timer pointer */
{
/* Check entire process table and give all process a higher priority. This
 * effectively means giving a new quantum. If a process already is at its
 * maximum priority, its quantum will be renewed.
 */
  static timer_t queue_timer;                   /* timer structure to use */
  register struct proc* rp;                     /* process table pointer  */
  clock_t next_period;                          /* time of next period   */
  int ticks_added = 0;                          /* total time added */

  for (rp=BEG_PROC_ADDR; rp<END_PROC_ADDR; rp++) {
      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 */

          /* We don't want processes that are highered in priority
           * to end up in the real-time queue. In normal cases they
           * get a priority higher than the RT priority. We just do a
           * check for safety if a process is allowed to get this priority.
           */
          if (rp->p_priority == RT_Q) {
              if (rp->p_max_priority <= (RT_Q - 1)) {
                  rp->p_priority = RT_Q - 1;
              } else {
                  /* should not happen */
                  rp->p_priority = USER_Q;
              }
          }

              if (rp->p_rts_flags == 0) enqueue(rp);    /* put on queue */
          }
          else {
              ticks_added += rp->p_quantum_size - rp->p_ticks_left;
              rp->p_ticks_left = rp->p_quantum_size;    /* give new quantum */
          }
          unlock(5);
      }
  }
#if DEBUG
  kprintf("ticks_added: %d\n", ticks_added);
#endif

  /* Now schedule a new watchdog timer to balance the queues again.  The
   * period depends on the total amount of quantum ticks added.
   */
  next_period = MAX(Q_BALANCE_TICKS, ticks_added);      /* calculate next */
  set_timer(&queue_timer, get_uptime() + next_period, balance_queues);
}

/*===========================================================================*
```

```
 *                              lock_send                                   *
 *==========================================================================*/
PUBLIC int lock_send(dst_e, m_ptr)
int dst_e;                      /* to whom is message being sent? */
message *m_ptr;                 /* pointer to message buffer */
{
/* Safe gateway to mini_send() for tasks. */
  int result;
  lock(2, "send");
  result = mini_send(proc_ptr, dst_e, m_ptr, NON_BLOCKING);
  unlock(2);
  return(result);
}

/*==========================================================================*
 *                              lock_enqueue                                *
 *==========================================================================*/
PUBLIC void lock_enqueue(rp)
struct proc *rp;                /* this process is now runnable */
{
/* Safe gateway to enqueue() for tasks. */
  lock(3, "enqueue");
  enqueue(rp);
  unlock(3);
}

/*==========================================================================*
 *                              lock_dequeue                                *
 *==========================================================================*/
PUBLIC void lock_dequeue(rp)
struct proc *rp;                /* this process is no longer runnable */
{
/* Safe gateway to dequeue() for tasks. */
  if (k_reenter >= 0) {
        /* We're in an exception or interrupt, so don't lock (and ...
         * don't unlock).
         */
        dequeue(rp);
  } else {
        lock(4, "dequeue");
        dequeue(rp);
        unlock(4);
  }
}

/*==========================================================================*
 *                              isokendpt_f                                 *
 *==========================================================================*/
#if DEBUG_ENABLE_IPC_WARNINGS
PUBLIC int isokendpt_f(file, line, e, p, fatalflag)
char *file;
int line;
#else
PUBLIC int isokendpt_f(e, p, fatalflag)
#endif
int e, *p, fatalflag;
{
        int ok = 0;
        /* Convert an endpoint number into a process number.
         * Return nonzero if the process is alive with the corresponding
         * generation number, zero otherwise.
         *
         * This function is called with file and line number by the
         * isokendpt_d macro if DEBUG_ENABLE_IPC_WARNINGS is defined,
         * otherwise without. This allows us to print the where the
         * conversion was attempted, making the errors verbose without
         * adding code for that at every call.
         *
         * If fatalflag is nonzero, we must panic if the conversion doesn't
         * succeed.
         */
        *p = _ENDPOINT_P(e);
        if(!isokprocn(*p)) {
#if DEBUG_ENABLE_IPC_WARNINGS
                kprintf("kernel:%s:%d: bad endpoint %d: proc %d out of range\n",
                file, line, e, *p);
#endif
        } else if(isemptyn(*p)) {
#if DEBUG_ENABLE_IPC_WARNINGS
        kprintf("kernel:%s:%d: bad endpoint %d: proc %d empty\n", file, line, e,
            *p);
#endif
        } else if(proc_addr(*p)->p_endpoint != e) {
#if DEBUG_ENABLE_IPC_WARNINGS
                kprintf("kernel:%s:%d: bad endpoint %d: proc %d has ept %d
                    (generation %d vs. %d)\n", file, line,
                e, *p, proc_addr(*p)->p_endpoint,
                _ENDPOINT_G(e), _ENDPOINT_G(proc_addr(*p)->p_endpoint));
#endif
        } else ok = 1;
        if(!ok && fatalflag) {
                panic("invalid endpoint ", e);
        }
        return ok;
}

/*===========================================================================*
 *              edf_sched                                                    *
 *===========================================================================*/
#define DEBUG_EDF_SCHED 0
PUBLIC void edf_sched(void)
{
  struct proc *temp_p;

  /* if no EDF process is scheduled, check if an EDF process
   * can be scheduled. If an EDF process is scheduled check if there is a
   * other process with an earlier deadline, and schedule that process.
   */

  if (edf_rp == NIL_PROC) {
      /* no EDF process is scheduled,
       * Check if the EDF run queue is not empty.
       * If it is not empty run the head process.
       */
      if (edf_run_head != NIL_PROC) {

          /* new EDF process to schedule */
          edf_rp = edf_run_head;

          /* remove the new process from the run queue */
          edf_run_head = edf_rp->p_rt_link;

          /* add process to the scheduling queue */
          lock_enqueue(edf_rp);
```

```
            #if DEBUG_EDF_SCHED == 1
            kprintf("edf_sched(): edf_rp = %s %d\n", edf_rp->p_name,
                edf_rp->p_endpoint);
            #endif
        } else {
            /* edf run queue is empty, no process to run */
            #if DEBUG_EDF_SCHED == 1
            kprintf("edf_sched(): edf_rp = NONE\n");
            #endif
        }
    } else if (edf_run_head != NIL_PROC &&
            edf_run_head->p_rt_nextperiod < edf_rp->p_rt_nextperiod) {

        struct proc * prev_p;
        /* The head process in the EDF run queue has an earlier deadline
         * than the current scheduled EDF process. Remove the current scheduled EDF
         * process from the scheduling queue and schedule the
         * process with the earliest deadline.
         */

        /* remove the current process from the scheduling queue */
        lock_dequeue(edf_rp);

        /* Remove the now scheduled EDF process from the run queue and
         * save it in prev_p. Add the previous running process to the EDF
         * run queue.
         */

        /* save the previous running process to prev_p */
        prev_p = edf_rp;

        /* We replace the head of the run queue so prev_p will be linked to the same
         * process as the new schedulable EDF process is linking. The new
             schedulable EDF process is
         * still the head of the EDF run queue.
         */
        prev_p->p_rt_link = edf_run_head->p_rt_link;

        /* new EDF process to run */
        edf_rp = edf_run_head;

        /* edf_rp contains only one process so
         * edf_rp is not linked to other processes
         */
        edf_rp->p_rt_link = NIL_PROC;

        /* substitute the current schedulable process with the previous scheduled
            process
         * in the EDF run queue.
         */
        edf_run_head = prev_p;

        /* add the new schedulable EDF process to the scheduling queue */
        lock_enqueue(edf_rp);

        #if DEBUG_EDF_SCHED == 1
        kprintf("edf_sched(): edf_rp = %s %d\n", edf_rp->p_name,
            edf_rp->p_endpoint);
        #endif
    }
}
```

```c
/*===========================================================================*
 *              show_rt_data                                                 *
 *===========================================================================*/
PUBLIC void show_rt_data(void)
{
  struct proc **xpp;

  /* Dump scheduling info for Real-time Schedulers.
   * Currently Rate-Monotonic and Earliest Deadline First is supported.
   */

  if (rt_sched == SCHED_EDF) {

      /* Current RT scheduler is Earliest Deadline First.
       * Display processes in edf_rp, run queue, block list and wait queue.
       */

      kprintf("- edf_rp: ");
      if (edf_rp != NIL_PROC) {
          kprintf("[%d:%d:%d]", edf_rp->p_endpoint, edf_rp->p_rt_nextperiod,
              edf_rp->p_rt_ticksleft);
      }

      kprintf(" RQ: ");
      xpp = &edf_run_head;
      while (*xpp != NIL_PROC) {
          kprintf("[%d:%d] ", (*xpp)->p_endpoint, (*xpp)->p_rt_nextperiod);
          xpp = &(*xpp)->p_rt_link;
      }

      kprintf(" BQ: ");
      xpp = &edf_block_head;
      while (*xpp != NIL_PROC) {
          kprintf("[%d:%d] ", (*xpp)->p_endpoint, (*xpp)->p_rt_nextperiod);
          xpp = &(*xpp)->p_rt_link;
      }

      kprintf(" WQ: ");
      xpp = &edf_wait_head;
      while (*xpp != NIL_PROC) {
          kprintf("[%d:%d] ", (*xpp)->p_endpoint, (*xpp)->p_rt_nextperiod);
          xpp = &(*xpp)->p_rt_link;
      }

      kprintf("\n");
  } else if (rt_sched == SCHED_RM) {

      /* Current RT scheduler is Rate-Monotonic.
       * Display processes in the Rate-Monotonic queue.
       */

      kprintf("RM Q: ");

      xpp = &rdy_head[RT_Q];
      while (*xpp != NIL_PROC) {
          kprintf("[%d:%d:%s] ", (*xpp)->p_rt_priority, (*xpp)->p_endpoint,
              (*xpp)->p_name);
          xpp = &(*xpp)->p_nextready;
      }

      kprintf("\n");
```

```
    }
}
```

Listing 28: /usr/src/kernel/proto.h

```
/* Function prototypes. */

#ifndef PROTO_H
#define PROTO_H

/* Struct declarations. */
struct proc;
struct timer;

/* clock.c */
_PROTOTYPE( void clock_task, (void)                                      );
_PROTOTYPE( void clock_stop, (void)                                      );
_PROTOTYPE( clock_t get_uptime, (void)                                   );
_PROTOTYPE( unsigned long read_clock, (void)                             );
_PROTOTYPE( void set_timer, (struct timer *tp, clock_t t, tmr_func_t f) );
_PROTOTYPE( void reset_timer, (struct timer *tp)                         );

/* main.c */
_PROTOTYPE( void main, (void)                                            );
_PROTOTYPE( void prepare_shutdown, (int how)                             );

/* utility.c */
_PROTOTYPE( int kprintf, (const char *fmt, ...)                          );
_PROTOTYPE( void panic, (_CONST char *s, int n)                          );

/* proc.c */
_PROTOTYPE( int sys_call, (int call_nr, int src_dst,
                                        message *m_ptr, long bit_map)    );
_PROTOTYPE( int lock_notify, (int src, int dst)                          );
_PROTOTYPE( int lock_send, (int dst, message *m_ptr)                     );
_PROTOTYPE( void lock_enqueue, (struct proc *rp)                         );
_PROTOTYPE( void lock_dequeue, (struct proc *rp)                         );
_PROTOTYPE( void balance_queues, (struct timer *tp)                      );
#if DEBUG_ENABLE_IPC_WARNINGS
_PROTOTYPE( int isokendpt_f, (char *file, int line, int e, int *p, int f));
#define isokendpt_d(e, p, f) isokendpt_f(__FILE__, __LINE__, (e), (p), (f))
#else
_PROTOTYPE( int isokendpt_f, (int e, int *p, int f)                      );
#define isokendpt_d(e, p, f) isokendpt_f((e), (p), (f))
#endif

_PROTOTYPE( void edf_sched, (void) );
_PROTOTYPE( void show_rt_data, (void) );

/* start.c */
_PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mds,
                                U16_t parmoff, U16_t parmsize)           );

/* system.c */
_PROTOTYPE( int get_priv, (register struct proc *rc, int proc_type)      );
_PROTOTYPE( void send_sig, (int proc_nr, int sig_nr)                     );
_PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr)                    );
_PROTOTYPE( void sys_task, (void)                                        );
_PROTOTYPE( void get_randomness, (int source)                            );
_PROTOTYPE( int virtual_copy, (struct vir_addr *src, struct vir_addr *dst,
                                vir_bytes bytes)                         );
#define numap_local(proc_nr, vir_addr, bytes) \
        umap_local(proc_addr(proc_nr), D, (vir_addr), (bytes))
_PROTOTYPE( phys_bytes umap_local, (struct proc *rp, int seg,
                vir_bytes vir_addr, vir_bytes bytes)                     );
```

```
_PROTOTYPE( phys_bytes umap_remote, (struct proc *rp, int seg,
                vir_bytes vir_addr, vir_bytes bytes)                   );
_PROTOTYPE( phys_bytes umap_bios, (struct proc *rp, vir_bytes vir_addr,
                vir_bytes bytes)                                       );
_PROTOTYPE( void clear_endpoint, (struct proc *rc)                     );

#if (CHIP == INTEL)

/* exception.c */
_PROTOTYPE( void exception, (unsigned vec_nr)                          );

/* i8259.c */
_PROTOTYPE( void intr_init, (int mine)                                 );
_PROTOTYPE( void intr_handle, (irq_hook_t *hook)                       );
_PROTOTYPE( void put_irq_handler, (irq_hook_t *hook, int irq,
                                irq_handler_t handler)  );
_PROTOTYPE( void rm_irq_handler, (irq_hook_t *hook)                    );

/* klib*.s */
_PROTOTYPE( void int86, (void)                                         );
_PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
                phys_clicks dst_clicks, vir_bytes dst_offset)          );
_PROTOTYPE( void enable_irq, (irq_hook_t *hook)                        );
_PROTOTYPE( int disable_irq, (irq_hook_t *hook)                        );
_PROTOTYPE( u16_t mem_rdw, (U16_t segm, vir_bytes offset)              );
_PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
                phys_bytes count)                                      );
_PROTOTYPE( void phys_memset, (phys_bytes source, unsigned long pattern,
                phys_bytes count)                                      );
_PROTOTYPE( void phys_insb, (U16_t port, phys_bytes buf, size_t count)  );
_PROTOTYPE( void phys_insw, (U16_t port, phys_bytes buf, size_t count)  );
_PROTOTYPE( void phys_outsb, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void phys_outsw, (U16_t port, phys_bytes buf, size_t count) );
_PROTOTYPE( void reset, (void)                                         );
_PROTOTYPE( void level0, (void (*func)(void))                          );
_PROTOTYPE( void monitor, (void)                                       );
_PROTOTYPE( void read_tsc, (unsigned long *high, unsigned long *low)    );
_PROTOTYPE( unsigned long read_cr0, (void)                             );
_PROTOTYPE( void write_cr0, (unsigned long value)                      );
_PROTOTYPE( void write_cr3, (unsigned long value)                      );
_PROTOTYPE( unsigned long read_cpu_flags, (void)                       );

/* mpx*.s */
_PROTOTYPE( void idle_task, (void)                                     );
_PROTOTYPE( void restart, (void)                                       );

/* The following are never called from C (pure asm procs). */

/* Exception handlers (real or protected mode), in numerical order. */
void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
void                              _PROTOTYPE( double_fault, (void) );
void                              _PROTOTYPE( copr_seg_overrun, (void) );
void                              _PROTOTYPE( inval_tss, (void) );
void                              _PROTOTYPE( segment_not_present, (void) );
void                              _PROTOTYPE( stack_exception, (void) );
void                              _PROTOTYPE( general_protection, (void) );
void                              _PROTOTYPE( page_fault, (void) );
void                              _PROTOTYPE( copr_error, (void) );

/* Hardware interrupt handlers. */
_PROTOTYPE( void hwint00, (void) );
_PROTOTYPE( void hwint01, (void) );
_PROTOTYPE( void hwint02, (void) );
_PROTOTYPE( void hwint03, (void) );
_PROTOTYPE( void hwint04, (void) );
_PROTOTYPE( void hwint05, (void) );
_PROTOTYPE( void hwint06, (void) );
_PROTOTYPE( void hwint07, (void) );
_PROTOTYPE( void hwint08, (void) );
_PROTOTYPE( void hwint09, (void) );
_PROTOTYPE( void hwint10, (void) );
_PROTOTYPE( void hwint11, (void) );
_PROTOTYPE( void hwint12, (void) );
_PROTOTYPE( void hwint13, (void) );
_PROTOTYPE( void hwint14, (void) );
_PROTOTYPE( void hwint15, (void) );

/* Software interrupt handlers, in numerical order. */
_PROTOTYPE( void trp, (void) );
_PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
_PROTOTYPE( void level0_call, (void) );

/* protect.c */
_PROTOTYPE( void prot_init, (void)                                    );
_PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
                vir_bytes size, int privilege)                        );
_PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
                vir_bytes size, int privilege)                        );
_PROTOTYPE( phys_bytes seg2phys, (U16_t seg)                          );
_PROTOTYPE( void phys2seg, (u16_t *seg, vir_bytes *off, phys_bytes phys));
_PROTOTYPE( void enable_iop, (struct proc *pp)                        );
_PROTOTYPE( void alloc_segments, (struct proc *rp)                    );

/* system/do_vm.c */
_PROTOTYPE( void vm_map_default, (struct proc *pp)                    );

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* M68000 specific prototypes go here. */
#endif /* (CHIP == M68000) */

#endif /* PROTO_H */
```

Listing 29: /usr/src/kernel/table.c

```
/* The object file of "table.c" contains most kernel data. Variables that
 * are declared in the *.h files appear with EXTERN in front of them, as in
 *
 *    EXTERN int x;
 *
 * Normally EXTERN is defined as extern, so when they are included in another
 * file, no storage is allocated.  If EXTERN were not present, but just say,
 *
 *    int x;
 *
 * then including this file in several source files would cause 'x' to be
 * declared several times.  While some linkers accept this, others do not,
 * so they are declared extern when included normally.  However, it must be
 * declared for real somewhere.  That is done here, by redefining EXTERN as
 * the null string, so that inclusion of all *.h files in table.c actually
 * generates storage for them.
 *
 * Various variables could not be declared EXTERN, but are declared PUBLIC
 * or PRIVATE. The reason for this is that extern variables cannot have a
 * default initialization. If such variables are shared, they must also be
 * declared in one of the *.h files without the initialization.  Examples
 * include 'boot_image' (this file) and 'idt' and 'gdt' (protect.c).
 *
 * Changes:
 *    Aug 02, 2005   set privileges and minimal boot image (Jorrit N. Herder)
 *    Oct 17, 2004   updated above and tasktab comments  (Jorrit N. Herder)
 *    May 01, 2004   changed struct for system image  (Jorrit N. Herder)
 */
#define _TABLE

#include "kernel.h"
#include "proc.h"
#include "ipc.h"
#include <minix/com.h>
#include <ibm/int86.h>

/* Define stack sizes for the kernel tasks included in the system image. */
#define NO_STACK        0
#define SMALL_STACK     (128 * sizeof(char *))
#define IDL_S   SMALL_STACK     /* 3 intr, 3 temps, 4 db for Intel */
#define HRD_S   NO_STACK        /* dummy task, uses kernel stack */
#define TSK_S   SMALL_STACK     /* system and clock task */

/* Stack space for all the task stacks.  Declared as (char *) to align it. */
#define TOT_STACK_SPACE (IDL_S + HRD_S + (2 * TSK_S))
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];

/* Define flags for the various process types. */
#define IDL_F   (SYS_PROC | PREEMPTIBLE | BILLABLE)     /* idle task */
#define TSK_F   (SYS_PROC)                              /* kernel tasks */
#define SRV_F   (SYS_PROC | PREEMPTIBLE)                /* system services */
#define USR_F   (BILLABLE | PREEMPTIBLE)                /* user processes */

/* Define system call traps for the various process types. These call masks
 * determine what system call traps a process is allowed to make.
 */
#define TSK_T   (1 << RECEIVE)                  /* clock and system */
#define SRV_T   (~0)                            /* system services */
#define USR_T   ((1 << SENDREC) | (1 << ECHO))  /* user processes */
```

```
/* Send masks determine to whom processes can send messages or notifications.
 * The values here are used for the processes in the boot image. We rely on
 * the initialization code in main() to match the s_nr_to_id() mapping for the
 * processes in the boot image, so that the send mask that is defined here
 * can be directly copied onto map[0] of the actual send mask. Privilege
 * structure 0 is shared by user processes.
 */
#define s(n)            (1 << s_nr_to_id(n))
#define SRV_M   (~0)
#define SYS_M   (~0)
#define USR_M (s(PM_PROC_NR) | s(FS_PROC_NR) | s(RS_PROC_NR) | s(SYSTEM)) |
    s(SS_PROC_NR)
#define DRV_M (USR_M | s(SYSTEM) | s(CLOCK) | s(DS_PROC_NR) | s(LOG_PROC_NR) |
    s(TTY_PROC_NR))

/* Define kernel calls that processes are allowed to make. This is not looking
 * very nice, but we need to define the access rights on a per call basis.
 * Note that the reincarnation server has all bits on, because it should
 * be allowed to distribute rights to services that it starts.
 */
#define c(n)    (1 << ((n)-KERNEL_CALL))
#define RS_C    ~0
#define DS_C    ~0
#define PM_C    ~(c(SYS_DEVIO) | c(SYS_SDEVIO) | c(SYS_VDEVIO) | c(SYS_IRQCTL) |
    c(SYS_INT86))
#define FS_C    (c(SYS_KILL) | c(SYS_VIRCOPY) | c(SYS_VIRVCOPY) | c(SYS_UMAP) |
    c(SYS_GETINFO) | c(SYS_EXIT) | c(SYS_TIMES) | c(SYS_SETALARM))
#define DRV_C (FS_C | c(SYS_SEGCTL) | c(SYS_IRQCTL) | c(SYS_INT86) | c(SYS_DEVIO)
    | c(SYS_SDEVIO) | c(SYS_VDEVIO))
#define TTY_C (DRV_C | c(SYS_ABORT) | c(SYS_VM_MAP) | c(SYS_IOPENABLE))
#define MEM_C   (DRV_C | c(SYS_PHYSCOPY) | c(SYS_PHYSVCOPY) | c(SYS_VM_MAP) | \
        c(SYS_IOPENABLE))
#define SS_C    c(SYS_GETINFO) /* semaphore server only needs SYS_GETINFO */

/* The system image table lists all programs that are part of the boot image.
 * The order of the entries here MUST agree with the order of the programs
 * in the boot image and all kernel tasks must come first.
 *
 * Each entry provides the process number, flags, quantum size, scheduling
 * queue, allowed traps, ipc mask, and a name for the process table. The
 * initial program counter and stack size is also provided for kernel tasks.
 *
 * Note: the quantum size must be positive in all cases!
 */
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,     0, "idle"  },
 { CLOCK,clock_task, TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0,     0, "clock" },
 { SYSTEM, sys_task, TSK_F,  8, TASK_Q, TSK_S, TSK_T,     0,     0, "system"},
 { HARDWARE,      0, TSK_F,  8, TASK_Q, HRD_S,     0,     0,     0, "kernel"},
 { PM_PROC_NR,    0, SRV_F, 32,      3, 0,     SRV_T, SRV_M,  PM_C, "pm"    },
 { FS_PROC_NR,    0, SRV_F, 32,      4, 0,     SRV_T, SRV_M,  FS_C, "fs"    },
 { RS_PROC_NR,    0, SRV_F,  4,      3, 0,     SRV_T, SYS_M,  RS_C, "rs"    },
 { DS_PROC_NR,    0, SRV_F,  4,      3, 0,     SRV_T, SYS_M,  DS_C, "ds"    },
 { TTY_PROC_NR,   0, SRV_F,  4,      1, 0,     SRV_T, SYS_M, TTY_C, "tty"   },
 { MEM_PROC_NR,   0, SRV_F,  4,      2, 0,     SRV_T, SYS_M, MEM_C, "mem"   },
 { LOG_PROC_NR,   0, SRV_F,  4,      2, 0,     SRV_T, SYS_M, DRV_C, "log"   },
 { SS_PROC_NR,    0, SRV_F,  4,      3, 0,     SRV_T, SYS_M, SS_C,  "ss"    },
 { INIT_PROC_NR,  0, USR_F,  8, USER_Q, 0,     USR_T, USR_M,     0, "init"  },
};

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

```
 * the first chunk of the ipc mask has enough bits to accommodate the processes
 * in the image.
 * If a problem is detected, the size of the 'dummy' array will be negative,
 * causing a compile time error. Note that no space is actually allocated
 * because 'dummy' is declared extern.
 */
extern int dummy[(NR_BOOT_PROCS==sizeof(image)/
        sizeof(struct boot_image))?1:-1];
extern int dummy[(BITCHUNK_BITS > NR_BOOT_PROCS - 1) ? 1 : -1];
```

Listing 30: /usr/src/kernel/system.c

```
/* This task handles the interface between the kernel and user-level servers.
 * System services can be accessed by doing a system call. System calls are
 * transformed into request messages, which are handled by this task. By
 * convention, a sys_call() is transformed in a SYS_CALL request message that
 * is handled in a function named do_call().
 *
 * A private call vector is used to map all system calls to the functions that
 * handle them. The actual handler functions are contained in separate files
 * to keep this file clean. The call vector is used in the system task's main
 * loop to handle all incoming requests.
 *
 * In addition to the main sys_task() entry point, which starts the main loop,
 * there are several other minor entry points:
 *   get_priv:          assign privilege structure to user or system process
 *   send_sig:          send a signal directly to a system process
 *   cause_sig:         take action to cause a signal to occur via PM
 *   umap_local:        map virtual address in LOCAL_SEG to physical
 *   umap_remote:       map virtual address in REMOTE_SEG to physical
 *   umap_bios:         map virtual address in BIOS_SEG to physical
 *   virtual_copy:      copy bytes from one virtual address to another
 *   get_randomness:    accumulate randomness in a buffer
 *   clear_endpoint:    remove a process' ability to send and receive messages
 *
 * Changes:
 *   Jun 14, 2009   added kernel calls for RT to initialize (Bianco Zandbergen)
 *   Aug 04, 2005   check if system call is allowed  (Jorrit N. Herder)
 *   Jul 20, 2005   send signal to services with message  (Jorrit N. Herder)
 *   Jan 15, 2005   new, generalized virtual copy function  (Jorrit N. Herder)
 *   Oct 10, 2004   dispatch system calls from call vector  (Jorrit N. Herder)
 *   Sep 30, 2004   source code documentation updated  (Jorrit N. Herder)
 */

#include "debug.h"
#include "kernel.h"
#include "system.h"
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <minix/endpoint.h>
#if (CHIP == INTEL)
#include <ibm/memory.h>
#include "protect.h"
#endif

/* Declaration of the call vector that defines the mapping of system calls
 * to handler functions. The vector is initialized in sys_init() with map(),
 * which makes sure the system call numbers are ok. No space is allocated,
 * because the dummy is declared extern. If an illegal call is given, the
 * array size will be negative and this won't compile.
 */
PUBLIC int (*call_vec[NR_SYS_CALLS])(message *m_ptr);

#define map(call_nr, handler) \
    {extern int dummy[NR_SYS_CALLS>(unsigned)(call_nr-KERNEL_CALL) ? 1:-1];} \
    call_vec[(call_nr-KERNEL_CALL)] = (handler)

FORWARD _PROTOTYPE( void initialize, (void));

/*===========================================================================*
```

```
 *                               sys_task                                      *
 *===========================================================================*/
PUBLIC void sys_task()
{
/* Main entry point of sys_task.  Get the message and dispatch on type. */
  static message m;
  register int result;
  register struct proc *caller_ptr;
  unsigned int call_nr;
  int s;

  /* Initialize the system task. */
  initialize();

  while (TRUE) {
      /* Get work. Block and wait until a request message arrives. */
      receive(ANY, &m);
      call_nr = (unsigned) m.m_type - KERNEL_CALL;
      who_e = m.m_source;
      okendpt(who_e, &who_p);
      caller_ptr = proc_addr(who_p);

      /* See if the caller made a valid request and try to handle it. */
      if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("SYSTEM: request %d from %d denied.\n", call_nr,m.m_source);
#endif
          result = ECALLDENIED;                 /* illegal message type */
      } else if (call_nr >= NR_SYS_CALLS) {             /* check call number */
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("SYSTEM: illegal request %d from %d.\n", call_nr,m.m_source);
#endif
          result = EBADREQUEST;                 /* illegal message type */
      }
      else {
          result = (*call_vec[call_nr])(&m);    /* handle the system call */
      }

      /* Send a reply, unless inhibited by a handler function. Use the kernel
       * function lock_send() to prevent a system call trap. The destination
       * is known to be blocked waiting for a message.
       */
      if (result != EDONTREPLY) {
          m.m_type = result;                    /* report status of call */
          if (OK != (s=lock_send(m.m_source, &m))) {
              kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
          }
      }
  }
}

/*===========================================================================*
 *                               initialize                                    *
 *===========================================================================*/
PRIVATE void initialize(void)
{
  register struct priv *sp;
  int i;

  /* Initialize IRQ handler hooks. Mark all hooks available. */
  for (i=0; i<NR_IRQ_HOOKS; i++) {
      irq_hooks[i].proc_nr_e = NONE;
  }

  /* Initialize all alarm timers for all processes. */
  for (sp=BEG_PRIV_ADDR; sp < END_PRIV_ADDR; sp++) {
    tmr_inittimer(&(sp->s_alarm_timer));
  }

  /* Initialize the call vector to a safe default handler. Some system calls
   * may be disabled or nonexistant. Then explicitely map known calls to their
   * handler functions. This is done with a macro that gives a compile error
   * if an illegal call number is used. The ordering is not important here.
   */
  for (i=0; i<NR_SYS_CALLS; i++) {
      call_vec[i] = do_unused;
  }

  /* Process management. */
  map(SYS_FORK, do_fork);             /* a process forked a new process */
  map(SYS_EXEC, do_exec);             /* update process after execute */
  map(SYS_EXIT, do_exit);             /* clean up after process exit */
  map(SYS_NICE, do_nice);             /* set scheduling priority */
  map(SYS_PRIVCTL, do_privctl);       /* system privileges control */
  map(SYS_TRACE, do_trace);           /* request a trace operation */

  /* Signal handling. */
  map(SYS_KILL, do_kill);             /* cause a process to be signaled */
  map(SYS_GETKSIG, do_getksig);       /* PM checks for pending signals */
  map(SYS_ENDKSIG, do_endksig);       /* PM finished processing signal */
  map(SYS_SIGSEND, do_sigsend);       /* start POSIX-style signal */
  map(SYS_SIGRETURN, do_sigreturn);   /* return from POSIX-style signal */

  /* Device I/O. */
  map(SYS_IRQCTL, do_irqctl);         /* interrupt control operations */
  map(SYS_DEVIO, do_devio);           /* inb, inw, inl, outb, outw, outl */
  map(SYS_SDEVIO, do_sdevio);         /* phys_insb, _insw, _outsb, _outsw */
  map(SYS_VDEVIO, do_vdevio);         /* vector with devio requests */
  map(SYS_INT86, do_int86);           /* real-mode BIOS calls */

  /* Memory management. */
  map(SYS_NEWMAP, do_newmap);         /* set up a process memory map */
  map(SYS_SEGCTL, do_segctl);         /* add segment and get selector */
  map(SYS_MEMSET, do_memset);         /* write char to memory area */
  map(SYS_VM_SETBUF, do_vm_setbuf);   /* PM passes buffer for page tables */
  map(SYS_VM_MAP, do_vm_map);         /* Map/unmap physical (device) memory */

  /* Copying. */
  map(SYS_UMAP, do_umap);             /* map virtual to physical address */
  map(SYS_VIRCOPY, do_vircopy);       /* use pure virtual addressing */
  map(SYS_PHYSCOPY, do_physcopy);     /* use physical addressing */
  map(SYS_VIRVCOPY, do_virvcopy);     /* vector with copy requests */
  map(SYS_PHYSVCOPY, do_physvcopy);   /* vector with copy requests */

  /* Clock functionality. */
  map(SYS_TIMES, do_times);           /* get uptime and process times */
  map(SYS_SETALARM, do_setalarm);     /* schedule a synchronous alarm */

  /* System control. */
  map(SYS_ABORT, do_abort);           /* abort MINIX */
  map(SYS_GETINFO, do_getinfo);       /* request system information */
  map(SYS_IOPENABLE, do_iopenable);   /* Enable I/O */

  /* Real-time */
```

```
  map(SYS_RT_SET_SCHED, do_rt_set_sched);   /* set real-time scheduler */
  map(SYS_RT_SET, do_rt_set);               /* make process real-time */
  map(SYS_RT_SHOW_DATA, do_rt_show_data);   /* showing scheduler debug info */
  map(SYS_RT_NEXTPERIOD, do_rt_nextperiod); /* wait till next period for EDF
      processes */
  map(SYS_RT_SCHED_BRIDGE, do_rt_set_sched_bridge); /* set the value of
      rt_sched_bridge */
  map(SYS_KLOG_SET, do_klog_set); /* set the kernel log state */
  map(SYS_KLOG_COPY, do_klog_copy); /* copy kernel log to user application */
}

/*===========================================================================*
 *                               get_priv                                    *
 *===========================================================================*/
PUBLIC int get_priv(rc, proc_type)
register struct proc *rc;              /* new (child) process pointer */
int proc_type;                         /* system or user process flag */
{
/* Get a privilege structure. All user processes share the same privilege
 * structure. System processes get their own privilege structure.
 */
  register struct priv *sp;                    /* privilege structure */

  if (proc_type == SYS_PROC) {                 /* find a new slot */
      for (sp = BEG_PRIV_ADDR; sp < END_PRIV_ADDR; ++sp)
          if (sp->s_proc_nr == NONE && sp->s_id != USER_PRIV_ID) break;
      if (sp->s_proc_nr != NONE) return(ENOSPC);
      rc->p_priv = sp;                         /* assign new slot */
      rc->p_priv->s_proc_nr = proc_nr(rc);     /* set association */
      rc->p_priv->s_flags = SYS_PROC;          /* mark as privileged */
  } else {
      rc->p_priv = &priv[USER_PRIV_ID];        /* use shared slot */
      rc->p_priv->s_proc_nr = INIT_PROC_NR;    /* set association */
      rc->p_priv->s_flags = 0;                 /* no initial flags */
  }
  return(OK);
}

/*===========================================================================*
 *                               get_randomness                              *
 *===========================================================================*/
PUBLIC void get_randomness(source)
int source;
{
/* On machines with the RDTSC (cycle counter read instruction - pentium
 * and up), use that for high-resolution raw entropy gathering. Otherwise,
 * use the realtime clock (tick resolution).
 *
 * Unfortunately this test is run-time - we don't want to bother with
 * compiling different kernels for different machines.
 *
 * On machines without RDTSC, we use read_clock().
 */
  int r_next;
  unsigned long tsc_high, tsc_low;

  source %= RANDOM_SOURCES;
  r_next= krandom.bin[source].r_next;
  if (machine.processor > 486) {
      read_tsc(&tsc_high, &tsc_low);
      krandom.bin[source].r_buf[r_next] = tsc_low;
  } else {
      krandom.bin[source].r_buf[r_next] = read_clock();
  }
  if (krandom.bin[source].r_size < RANDOM_ELEMENTS) {
        krandom.bin[source].r_size ++;
  }
  krandom.bin[source].r_next = (r_next + 1 ) % RANDOM_ELEMENTS;
}

/*===========================================================================*
 *                               send_sig                                    *
 *===========================================================================*/
PUBLIC void send_sig(int proc_nr, int sig_nr)
{
/* Notify a system process about a signal. This is straightforward. Simply
 * set the signal that is to be delivered in the pending signals map and
 * send a notification with source SYSTEM.
 *
 * Process number is verified to avoid writing in random places, but we
 * don't kprintf() or panic() because that causes send_sig() invocations.
 */
  register struct proc *rp;
  static int n;

  if(!isokprocn(proc_nr) || isemptyn(proc_nr))
        return;

  rp = proc_addr(proc_nr);
  sigaddset(&priv(rp)->s_sig_pending, sig_nr);
  lock_notify(SYSTEM, rp->p_endpoint);
}

/*===========================================================================*
 *                               cause_sig                                   *
 *===========================================================================*/
PUBLIC void cause_sig(proc_nr, sig_nr)
int proc_nr;                    /* process to be signalled */
int sig_nr;                     /* signal to be sent, 1 to _NSIG */
{
/* A system process wants to send a signal to a process.  Examples are:
 *  - HARDWARE wanting to cause a SIGSEGV after a CPU exception
 *  - TTY wanting to cause SIGINT upon getting a DEL
 *  - FS wanting to cause SIGPIPE for a broken pipe
 * Signals are handled by sending a message to PM.  This function handles the
 * signals and makes sure the PM gets them by sending a notification. The
 * process being signaled is blocked while PM has not finished all signals
 * for it.
 * Race conditions between calls to this function and the system calls that
 * process pending kernel signals cannot exist. Signal related functions are
 * only called when a user process causes a CPU exception and from the kernel
 * process level, which runs to completion.
 */
  register struct proc *rp;

  /* Check if the signal is already pending. Process it otherwise. */
  rp = proc_addr(proc_nr);
  if (! sigismember(&rp->p_pending, sig_nr)) {
      sigaddset(&rp->p_pending, sig_nr);
      if (! (rp->p_rts_flags & SIGNALED)) {             /* other pending */
          if (rp->p_rts_flags == 0) lock_dequeue(rp);   /* make not ready */
          rp->p_rts_flags |= SIGNALED | SIG_PENDING;    /* update flags */
          send_sig(PM_PROC_NR, SIGKSIG);
      }
  }
}

/*===========================================================================*
 *                              umap_bios                                    *
 *===========================================================================*/
PUBLIC phys_bytes umap_bios(rp, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
vir_bytes vir_addr;             /* virtual address in BIOS segment */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address at the BIOS. Note: currently, BIOS
 * address zero (the first BIOS interrupt vector) is not considered, as an
 * error here, but since the physical address will be zero as well, the
 * calling function will think an error occurred. This is not a problem,
 * since no one uses the first BIOS interrupt vector.
 */

  /* Check all acceptable ranges. */
  if (vir_addr >= BIOS_MEM_BEGIN && vir_addr + bytes <= BIOS_MEM_END)
        return (phys_bytes) vir_addr;
  else if (vir_addr >= BASE_MEM_TOP && vir_addr + bytes <= UPPER_MEM_END)
        return (phys_bytes) vir_addr;

#if DEAD_CODE   /* brutal fix, if the above is too restrictive */
  if (vir_addr >= BIOS_MEM_BEGIN && vir_addr + bytes <= UPPER_MEM_END)
        return (phys_bytes) vir_addr;
#endif

  kprintf("Warning, error in umap_bios, virtual address 0x%x\n", vir_addr);
  return 0;
}

/*===========================================================================*
 *                              umap_local                                   *
 *===========================================================================*/
PUBLIC phys_bytes umap_local(rp, seg, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
int seg;                        /* T, D, or S segment */
vir_bytes vir_addr;             /* virtual address in bytes within the seg */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */
  vir_clicks vc;                /* the virtual address in clicks */
  phys_bytes pa;                /* intermediate variables as phys_bytes */
#if (CHIP == INTEL)
  phys_bytes seg_base;
#endif

  /* If 'seg' is D it could really be S and vice versa.  T really means T.
   * If the virtual address falls in the gap,  it causes a problem. On the
   * 8088 it is probably a legal stack reference, since "stackfaults" are
   * not detected by the hardware.  On 8088s, the gap is called S and
   * accepted, but on other machines it is called D and rejected.
   * The Atari ST behaves like the 8088 in this respect.
   */

  if (bytes <= 0) return( (phys_bytes) 0);
  if (vir_addr + bytes <= vir_addr) return 0;   /* overflow */
  vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;   /* last click of data */

#if (CHIP == INTEL) || (CHIP == M68000)
```

```
  if (seg != T)
        seg = (vc < rp->p_memmap[D].mem_vir + rp->p_memmap[D].mem_len ? D : S);
#else
  if (seg != T)
        seg = (vc < rp->p_memmap[S].mem_vir ? D : S);
#endif

  if ((vir_addr>>CLICK_SHIFT) >= rp->p_memmap[seg].mem_vir +
        rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );

  if (vc >= rp->p_memmap[seg].mem_vir +
        rp->p_memmap[seg].mem_len) return( (phys_bytes) 0 );

#if (CHIP == INTEL)
  seg_base = (phys_bytes) rp->p_memmap[seg].mem_phys;
  seg_base = seg_base << CLICK_SHIFT;   /* segment origin in bytes */
#endif
  pa = (phys_bytes) vir_addr;
#if (CHIP != M68000)
  pa -= rp->p_memmap[seg].mem_vir << CLICK_SHIFT;
  return(seg_base + pa);
#endif
#if (CHIP == M68000)
  pa -= (phys_bytes)rp->p_memmap[seg].mem_vir << CLICK_SHIFT;
  pa += (phys_bytes)rp->p_memmap[seg].mem_phys << CLICK_SHIFT;
  return(pa);
#endif
}

/*===========================================================================*
 *                              umap_remote                                  *
 *===========================================================================*/
PUBLIC phys_bytes umap_remote(rp, seg, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
int seg;                        /* index of remote segment */
vir_bytes vir_addr;             /* virtual address in bytes within the seg */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */
  struct far_mem *fm;

  if (bytes <= 0) return( (phys_bytes) 0);
  if (seg < 0 || seg >= NR_REMOTE_SEGS) return( (phys_bytes) 0);

  fm = &rp->p_priv->s_farmem[seg];
  if (! fm->in_use) return( (phys_bytes) 0);
  if (vir_addr + bytes > fm->mem_len) return( (phys_bytes) 0);

  return(fm->mem_phys + (phys_bytes) vir_addr);
}

/*===========================================================================*
 *                              virtual_copy                                 *
 *===========================================================================*/
PUBLIC int virtual_copy(src_addr, dst_addr, bytes)
struct vir_addr *src_addr;      /* source virtual address */
struct vir_addr *dst_addr;      /* destination virtual address */
vir_bytes bytes;                /* # of bytes to copy  */
{
/* Copy bytes from virtual address src_addr to virtual address dst_addr.
 * Virtual addresses can be in ABS, LOCAL_SEG, REMOTE_SEG, or BIOS_SEG.
 */
```

```
  struct vir_addr *vir_addr[2]; /* virtual source and destination address */
  phys_bytes phys_addr[2];      /* absolute source and destination */
  int seg_index;
  int i;

  /* Check copy count. */
  if (bytes <= 0) return(EDOM);

  /* Do some more checks and map virtual addresses to physical addresses. */
  vir_addr[_SRC_] = src_addr;
  vir_addr[_DST_] = dst_addr;
  for (i=_SRC_; i<=_DST_; i++) {
        int proc_nr, type;
        struct proc *p;

        type = vir_addr[i]->segment & SEGMENT_TYPE;
        if(type != PHYS_SEG && isokendpt(vir_addr[i]->proc_nr_e, &proc_nr))
           p = proc_addr(proc_nr);
        else
           p = NULL;

      /* Get physical address. */
      switch(type) {
      case LOCAL_SEG:
          if(!p) return EDEADSRCDST;
          seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
          phys_addr[i] = umap_local(p, seg_index, vir_addr[i]->offset, bytes);
          break;
      case REMOTE_SEG:
          if(!p) return EDEADSRCDST;
          seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
          phys_addr[i] = umap_remote(p, seg_index, vir_addr[i]->offset, bytes);
          break;
      case BIOS_SEG:
          if(!p) return EDEADSRCDST;
          phys_addr[i] = umap_bios(p, vir_addr[i]->offset, bytes );
          break;
      case PHYS_SEG:
          phys_addr[i] = vir_addr[i]->offset;
          break;
      default:
          return(EINVAL);
      }

      /* Check if mapping succeeded. */
      if (phys_addr[i] <= 0 && vir_addr[i]->segment != PHYS_SEG)
          return(EFAULT);
  }

  /* Now copy bytes between physical addresseses. */
  phys_copy(phys_addr[_SRC_], phys_addr[_DST_], (phys_bytes) bytes);
  return(OK);
}


/*===========================================================================*
 *                              clear_endpoint                               *
 *===========================================================================*/
PUBLIC void clear_endpoint(rc)
register struct proc *rc;              /* slot of process to clean up */
{
  register struct proc *rp;            /* iterate over process table */
  register struct proc **xpp;             /* iterate over caller queue */
  int i;
  int sys_id;

  if(isemptyp(rc)) panic("clear_proc: empty process", proc_nr(rc));

  /* Make sure that the exiting process is no longer scheduled. */
  if (rc->p_rts_flags == 0) lock_dequeue(rc);
  rc->p_rts_flags |= NO_ENDPOINT;

  /* If the process happens to be queued trying to send a
   * message, then it must be removed from the message queues.
   */
  if (rc->p_rts_flags & SENDING) {
      int target_proc;

      okendpt(rc->p_sendto_e, &target_proc);
      xpp = &proc_addr(target_proc)->p_caller_q; /* destination's queue */
      while (*xpp != NIL_PROC) {                 /* check entire queue */
          if (*xpp == rc) {                      /* process is on the queue */
              *xpp = (*xpp)->p_q_link;           /* replace by next process */
#if DEBUG_ENABLE_IPC_WARNINGS
              kprintf("Proc %d removed from queue at %d\n",
                  proc_nr(rc), rc->p_sendto_e);
#endif
              break;                             /* can only be queued once */
          }
          xpp = &(*xpp)->p_q_link;               /* proceed to next queued */
      }
      rc->p_rts_flags &= ~SENDING;
  }
  rc->p_rts_flags &= ~RECEIVING;

  /* Likewise, if another process was sending or receive a message to or from
   * the exiting process, it must be alerted that process no longer is alive.
   * Check all processes.
   */
  for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
      if(isemptyp(rp))
        continue;

      /* Unset pending notification bits. */
      unset_sys_bit(priv(rp)->s_notify_pending, priv(rc)->s_id);

      /* Check if process is receiving from exiting process. */
      if ((rp->p_rts_flags & RECEIVING) && rp->p_getfrom_e == rc->p_endpoint) {
          rp->p_reg.retreg = ESRCDIED;           /* report source died */
          rp->p_rts_flags &= ~RECEIVING;         /* no longer receiving */
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("Proc %d receive dead src %d\n", proc_nr(rp), proc_nr(rc));
#endif
          if (rp->p_rts_flags == 0) lock_enqueue(rp);/* let process run again */
      }
      if ((rp->p_rts_flags & SENDING) && rp->p_sendto_e == rc->p_endpoint) {
          rp->p_reg.retreg = EDSTDIED;           /* report destination died */
          rp->p_rts_flags &= ~SENDING;           /* no longer sending */
#if DEBUG_ENABLE_IPC_WARNINGS
          kprintf("Proc %d send dead dst %d\n", proc_nr(rp), proc_nr(rc));
#endif
          if (rp->p_rts_flags == 0) lock_enqueue(rp);/* let process run again */
      }
  }
```

```
}
```

Listing 31: /usr/src/kernel/system.h

```
/* Function prototypes for the system library.  The prototypes in this file
 * are undefined to do_unused if the kernel call is not enabled in config.h.
 * The implementation is contained in src/kernel/system/.
 *
 * The system library allows to access system services by doing a kernel call.
 * System calls are transformed into request messages to the SYS task that is
 * responsible for handling the call. By convention, sys_call() is transformed
 * into a message with type SYS_CALL that is handled in a function do_call().
 *
 * Changes:
 *   Jul 30, 2005   created SYS_INT86 to support BIOS driver  (Philip Homburg)
 *   Jul 13, 2005   created SYS_PRIVCTL to manage services  (Jorrit N. Herder)
 *   Jul 09, 2005   updated SYS_KILL to signal services  (Jorrit N. Herder)
 *   Jun 21, 2005   created SYS_NICE for nice(2) kernel call  (Ben J. Gras)
 *   Jun 21, 2005   created SYS_MEMSET to speed up exec(2)  (Ben J. Gras)
 *   Apr 12, 2005   updated SYS_VCOPY for virtual_copy()  (Jorrit N. Herder)
 *   Jan 20, 2005   updated SYS_COPY for virtual_copy()  (Jorrit N. Herder)
 *   Oct 24, 2004   created SYS_GETKSIG to support PM  (Jorrit N. Herder)
 *   Oct 10, 2004   created handler for unused calls  (Jorrit N. Herder)
 *   Sep 09, 2004   updated SYS_EXIT to let services exit  (Jorrit N. Herder)
 *   Aug 25, 2004   rewrote SYS_SETALARM to clean up code  (Jorrit N. Herder)
 *   Jul 13, 2004   created SYS_SEGCTL to support drivers  (Jorrit N. Herder)
 *   May 24, 2004   created SYS_SDEVIO to support drivers  (Jorrit N. Herder)
 *   May 24, 2004   created SYS_GETINFO to retrieve info  (Jorrit N. Herder)
 *   Apr 18, 2004   created SYS_VDEVIO to support drivers  (Jorrit N. Herder)
 *   Feb 24, 2004   created SYS_IRQCTL to support drivers  (Jorrit N. Herder)
 *   Feb 02, 2004   created SYS_DEVIO to support drivers  (Jorrit N. Herder)
 */

#ifndef SYSTEM_H
#define SYSTEM_H

/* Common includes for the system library. */
#include "debug.h"
#include "kernel.h"
#include "proto.h"
#include "proc.h"

/* Default handler for unused kernel calls. */
_PROTOTYPE( int do_unused, (message *m_ptr) );

_PROTOTYPE( int do_exec, (message *m_ptr) );
#if ! USE_EXEC
#define do_exec do_unused
#endif

_PROTOTYPE( int do_fork, (message *m_ptr) );
#if ! USE_FORK
#define do_fork do_unused
#endif

_PROTOTYPE( int do_newmap, (message *m_ptr) );
#if ! USE_NEWMAP
#define do_newmap do_unused
#endif

_PROTOTYPE( int do_exit, (message *m_ptr) );
#if ! USE_EXIT
#define do_exit do_unused
#endif
```

```
_PROTOTYPE( int do_trace, (message *m_ptr) );
#if ! USE_TRACE
#define do_trace do_unused
#endif

_PROTOTYPE( int do_nice, (message *m_ptr) );
#if ! USE_NICE
#define do_nice do_unused
#endif

_PROTOTYPE( int do_copy, (message *m_ptr) );
#define do_vircopy      do_copy
#define do_physcopy     do_copy
#if ! (USE_VIRCOPY || USE_PHYSCOPY)
#define do_copy do_unused
#endif

_PROTOTYPE( int do_vcopy, (message *m_ptr) );
#define do_virvcopy     do_vcopy
#define do_physvcopy    do_vcopy
#if ! (USE_VIRVCOPY || USE_PHYSVCOPY)
#define do_vcopy do_unused
#endif

_PROTOTYPE( int do_umap, (message *m_ptr) );
#if ! USE_UMAP
#define do_umap do_unused
#endif

_PROTOTYPE( int do_memset, (message *m_ptr) );
#if ! USE_MEMSET
#define do_memset do_unused
#endif

_PROTOTYPE( int do_vm_setbuf, (message *m_ptr) );
_PROTOTYPE( int do_vm_map, (message *m_ptr) );

_PROTOTYPE( int do_abort, (message *m_ptr) );
#if ! USE_ABORT
#define do_abort do_unused
#endif

_PROTOTYPE( int do_getinfo, (message *m_ptr) );
#if ! USE_GETINFO
#define do_getinfo do_unused
#endif

_PROTOTYPE( int do_privctl, (message *m_ptr) );
#if ! USE_PRIVCTL
#define do_privctl do_unused
#endif

_PROTOTYPE( int do_segctl, (message *m_ptr) );
#if ! USE_SEGCTL
#define do_segctl do_unused
#endif

_PROTOTYPE( int do_irqctl, (message *m_ptr) );
#if ! USE_IRQCTL
#define do_irqctl do_unused
#endif
```

```
_PROTOTYPE( int do_devio, (message *m_ptr) );
#if ! USE_DEVIO
#define do_devio do_unused
#endif

_PROTOTYPE( int do_vdevio, (message *m_ptr) );
#if ! USE_VDEVIO
#define do_vdevio do_unused
#endif

_PROTOTYPE( int do_int86, (message *m_ptr) );

_PROTOTYPE( int do_sdevio, (message *m_ptr) );
#if ! USE_SDEVIO
#define do_sdevio do_unused
#endif

_PROTOTYPE( int do_kill, (message *m_ptr) );
#if ! USE_KILL
#define do_kill do_unused
#endif

_PROTOTYPE( int do_getksig, (message *m_ptr) );
#if ! USE_GETKSIG
#define do_getksig do_unused
#endif

_PROTOTYPE( int do_endksig, (message *m_ptr) );
#if ! USE_ENDKSIG
#define do_endksig do_unused
#endif

_PROTOTYPE( int do_sigsend, (message *m_ptr) );
#if ! USE_SIGSEND
#define do_sigsend do_unused
#endif

_PROTOTYPE( int do_sigreturn, (message *m_ptr) );
#if ! USE_SIGRETURN
#define do_sigreturn do_unused
#endif

_PROTOTYPE( int do_times, (message *m_ptr) );
#if ! USE_TIMES
#define do_times do_unused
#endif

_PROTOTYPE( int do_setalarm, (message *m_ptr) );
#if ! USE_SETALARM
#define do_setalarm do_unused
#endif

_PROTOTYPE( int do_iopenable, (message *m_ptr) );

/* prototypes for real-time and kernel logging */

_PROTOTYPE( int do_rt_set_sched, (message *m_ptr) );
#if ! USE_RT_SET_SCHED
#define do_rt_set_sched do_unused
#endif
```

```
_PROTOTYPE( int do_rt_set, (message *m_ptr) );
#if ! USE_RT_SET
#define do_rt_set do_unused
#endif

_PROTOTYPE( int do_rt_show_data, (message *m_ptr) );
#if ! USE_RT_SHOW_DATA
#define do_rt_show_data do_unused
#endif

_PROTOTYPE( int do_rt_nextperiod, (message *m_ptr) );
#if ! USE_RT_NEXTPERIOD
#define do_rt_nextperiod do_unused
#endif

_PROTOTYPE( int do_rt_set_sched_bridge, (message *m_ptr) );
#if ! USE_RT_SET_SCHED_BRIGE
#define do_rt_set_sched_bridge do_unused
#endif

_PROTOTYPE( int do_klog_set, (message *m_ptr) );
#if ! USE_KLOG_SET
#define do_klog_set do_unused
#endif

_PROTOTYPE( int do_klog_copy, (message *m_ptr) );
#if ! USE_KLOG_COPY
#define do_klog_copy do_unused
#endif

#endif  /* SYSTEM_H */
```

Listing 32: /usr/src/kernel/system/do_exit.c

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_EXIT
 *
 * The parameters for this kernel call are:
 *    m1_i1:    PR_ENDPT                (slot number of exiting process)
 */

#include "../system.h"

#include <minix/endpoint.h>
#include <minix/rt.h>

#if USE_EXIT

FORWARD _PROTOTYPE( void clear_proc, (register struct proc *rc));

/*===========================================================================*
 *                              do_exit                                      *
 *===========================================================================*/
PUBLIC int do_exit(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_exit. A user process has exited or a system process requests
 * to exit. Only the PM can request other process slots to be cleared.
 * The routine to clean up a process table slot cancels outstanding timers,
 * possibly removes the process from the message queues, and resets certain
 * process table fields to the default values.
 */
  int exit_e;

  /* Determine what process exited. User processes are handled here. */
  if (PM_PROC_NR == who_p) {
      if (m_ptr->PR_ENDPT != SELF) {            /* PM tries to exit self */
          if(!isokendpt(m_ptr->PR_ENDPT, &exit_e)) /* get exiting process */
             return EINVAL;
          clear_proc(proc_addr(exit_e));        /* exit a user process */
          return(OK);                           /* report back to PM */
      }
  }

  /* The PM or some other system process requested to be exited. */
  clear_proc(proc_addr(who_p));
  return(EDONTREPLY);
}

/*===========================================================================*
 *                              clear_proc                                   *
 *===========================================================================*/
PRIVATE void clear_proc(rc)
register struct proc *rc;               /* slot of process to clean up */
{
  register struct proc *rp;             /* iterate over process table */
  register struct proc **xpp;           /* iterate over caller queue */
  int i;
  int sys_id;
  char saved_rts_flags;

  /* Don't clear if already cleared. */
  if(isemptyp(rc)) return;
```

```
/* Remove the process' ability to send and receive messages */
clear_endpoint(rc);

/* Turn off any alarm timers at the clock. */
reset_timer(&priv(rc)->s_alarm_timer);

/* Make sure that the exiting process is no longer scheduled. */
if (rc->p_rts_flags == 0) lock_dequeue(rc);

/* Check the table with IRQ hooks to see if hooks should be released. */
for (i=0; i < NR_IRQ_HOOKS; i++) {
    int proc;
    if (rc->p_endpoint == irq_hooks[i].proc_nr_e) {
      rm_irq_handler(&irq_hooks[i]);  /* remove interrupt handler */
      irq_hooks[i].proc_nr_e = NONE;  /* mark hook as free */
    }
}

/* Release the process table slot. If this is a system process, also
 * release its privilege structure.  Further cleanup is not needed at
 * this point. All important fields are reinitialized when the
 * slots are assigned to another, new process.
 */
saved_rts_flags = rc->p_rts_flags;
rc->p_rts_flags = SLOT_FREE;
if (priv(rc)->s_flags & SYS_PROC) priv(rc)->s_proc_nr = NONE;

/* Clean up virtual memory */
if (rc->p_misc_flags & MF_VM)
      vm_map_default(rc);

/* additional clean-up for real-time processes */
if (is_rtp(rc)) {

    #if 0
    kprintf("Exit from RT process %d %s\n", rc->endpoint, rc->p_name);
    #endif

    /* set process slot to non-RT */
    rc->p_rt = 0;

    /* additional clean-up for EDF processes */
    if (rt_sched == SCHED_EDF) {
        /* remove process from one of the EDF data structures */

        /* check edf_rp */
        if (edf_rp != NIL_PROC && edf_rp == rc) {
            edf_rp = NIL_PROC;

            /* This process was last running.
             * Check if other EDF process can run now.
             */
            edf_sched();
        } else {

            /* check run queue */
            xpp = &edf_run_head;
            while (*xpp != NIL_PROC && *xpp != rc) {
                xpp = &(*xpp)->p_rt_link;
            }

            /* *xpp only equals NIL_PROC if
```

```
                 * the queue is empty or the process is not found.
                 */
                if (*xpp != NIL_PROC) {
                    /* Remove process from queue.
                     * The pointer pointing to the exited process
                     * will point to the process linked (pointed)
                     * by the exited process.
                     */
                    *xpp = (*xpp)->p_rt_link;
                }

                /* check  block list */
                xpp = &edf_block_head;
                while (*xpp != NIL_PROC && *xpp != rc) {
                    xpp = &(*xpp)->p_rt_link;
                }

                if (*xpp != NIL_PROC) {
                    *xpp = (*xpp)->p_rt_link;
                }

                /* check  wait queue */
                xpp = &edf_wait_head;
                while (*xpp != NIL_PROC && *xpp != rc) {
                    xpp = &(*xpp)->p_rt_link;
                }

                if (*xpp != NIL_PROC) {
                    *xpp = (*xpp)->p_rt_link;
                }
            }
        }
    }
}

#endif /* USE_EXIT */
```

Listing 33: /usr/src/kernel/system/do_klog_copy.c

```c
/* The kernel call implemented in this file:
 *   m_type:    SYS_KLOG_COPY
 *
 * The parameters for this kernel call are:
 *    m1_i1:    KLOG_ENDPT    Endpoint of user program
 *    m1_p1:    KLOG_PTR      (virtual) pointer to kernel log in user program
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/klog.h>

/*===========================================================================*
 *                   do_klog_copy                                            *
 *===========================================================================*/
PUBLIC int do_klog_copy(message *m_ptr)
{
  /* Copy the kernel log from the kernel to a user program. */

  size_t length; /* We store the kernel log size in bytes here */
  phys_bytes src_phys; /* We store the physical address of the log in the kernel
      here */
  phys_bytes dst_phys; /* We store the physical address of the log in the user
      program here */
  int proc_nr; /* process number of user program */

  /* calculate the size of the kernel log which is
   * the size of a struct klog_entry multiplied by the
   * number of entries in the log. KLOG_SIZE is defined in
   * <minix/klog.h>. This is the number of bytes we are gonna copy.
   */
  length = sizeof(struct klog_entry) * KLOG_SIZE;

  /* We can't use virtual memory addresses to copy from kernel to
   * user program. We will have to translate both the source and destination
   * to physical addresses.
   */

  /* Translate virtual address of the log in the kernel
   * to the physical address.
   */
  src_phys = vir2phys(&klog);

  /* Check if endpoint of user program is valid and get the process number */
  if (! isokendpt(m_ptr->KLOG_ENDPT, &proc_nr)) {
      return (EFAULT);
  }

  /* Translate the virtual address of the log in the user program
   * to the physical address.
   */
  dst_phys = numap_local(proc_nr, (vir_bytes) m_ptr->KLOG_PTR, length);

  /* translating went wrong if this is true */
  if (src_phys == 0 || dst_phys == 0) {
      kprintf("systask: do_klog_copy() src_phys or dst_phys == 0\n");
      return (EFAULT);
  }
  /* finally copy the log to the user program */
  phys_copy(src_phys, dst_phys, length);

  return (OK);
}
```

Listing 34: /usr/src/kernel/system/do_klog_set.c

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_KLOG_SET
 *
 * The parameters for this kernel call are:
 *    m1_i1:    KLOG_STATE       Kernel logger state
 *    m1_i2     KLOG_TYPE        Kernel log type
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/klog.h>

/*===========================================================================*
 *                  do_klog_set                                              *
 *===========================================================================*/
PUBLIC int do_klog_set(message *m_ptr)
{

  /* check if the type specified is valid */
  if (m_ptr->KLOG_TYPE != KLOG_CONTEXTSWITCH &&
      m_ptr->KLOG_TYPE != KLOG_CLOCKINT) {

      return EINVAL;
  }

  /* Set the kernel logger state.
   * If set to 1 it will log untill the buffer is full.
   * If the buffer is full the state will be set back to 0.
   */
  klog_state = m_ptr->KLOG_STATE;

  /* Set the kernel log type
   * If set to KLOG_CONTEXTSWITCH it will log every process
   * that will run next after a context switch.
   * If set to KLOG_CLOCKINT it will log every process
   * that was running when the clock interrupt happened.
   */
  klog_type = m_ptr->KLOG_TYPE;

  return (OK);
}
```

Listing 35: /usr/src/kernel/system/do_nice.c

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_NICE
 *
 * The parameters for this kernel call are:
 *    m1_i1:    PR_ENDPT        process number to change priority
 *    m1_i2:    PR_PRIORITY     the new priority
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>

#if USE_NICE

/*===========================================================================*
 *                                do_nice                                    *
 *===========================================================================*/
PUBLIC int do_nice(message *m_ptr)
{
/* Change process priority or stop the process. */
  int proc_nr, pri, new_q ;
  register struct proc *rp;

  /* Extract the message parameters and do sanity checking. */
  if(!isokendpt(m_ptr->PR_ENDPT, &proc_nr)) return EINVAL;
  if (iskerneln(proc_nr)) return(EPERM);

  pri = m_ptr->PR_PRIORITY;
  rp = proc_addr(proc_nr);

  if (is_rtp(rp)) return (EPERM); /* don't allow nice for RT processes */

  if (pri == PRIO_STOP) {

      /* Take process off the scheduling queues. */
      lock_dequeue(rp);
      rp->p_rts_flags |= NO_PRIORITY;
      return(OK);
  }
  else if (pri >= PRIO_MIN && pri <= PRIO_MAX) {

      /* The value passed in is currently between PRIO_MIN and PRIO_MAX.
       * We have to scale this between MIN_USER_Q and MAX_USER_Q to match
       * the kernel's scheduling queues.
       */
      new_q = MAX_USER_Q + (pri-PRIO_MIN) * (MIN_USER_Q-MAX_USER_Q+1) /
          (PRIO_MAX-PRIO_MIN+1);
      if (new_q < MAX_USER_Q) new_q = MAX_USER_Q;       /* shouldn't happen */
      if (new_q > MIN_USER_Q) new_q = MIN_USER_Q;       /* shouldn't happen */

      if (new_q == RT_Q && !is_rtp(rp)) return (EINVAL); /* don't allow other
          processes in the RT queue */

      /* Make sure the process is not running while changing its priority.
       * Put the process back in its new queue if it is runnable.
       */
      lock_dequeue(rp);
      rp->p_max_priority = rp->p_priority = new_q;
      if (! rp->p_rts_flags) lock_enqueue(rp);
```

```
        return(OK);
  }
  return(EINVAL);
}

#endif /* USE_NICE */
```

Listing 36: /usr/src/kernel/system/do_rt_nextperiod.c

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_RT_NEXTPERIOD
 *
 * The parameters for this kernel call are:
 *    m7_i1:    RT_ENDPT  Endpoint of process that wants to give up calc time
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/rt.h>


/*===========================================================================*
 *                    do_rt_nextperiod                                       *
 *===========================================================================*/
PUBLIC int do_rt_nextperiod(message *m_ptr)
{
  /* A process scheduled using Earliest Deadline First can give up
   * remaining calculation time using this kernel call.
   * A process that invoked this kernel call is on the block list waiting
   * for a reply from PM. We will have to remove it from the block list
   * and put it on the wait queue waiting for the next period.
   */
  struct proc * rp;
  struct proc **xpp;
  int proc_nr;

  /* if rt_sched is not SCHED_EDF this kernel call
   * is not allowed!
   */
  if (rt_sched != SCHED_EDF) {
      return (EPERM);
  }

  /* check if endpoint is valid and convert endpoint to
   *  process number stored in proc_nr
   */
  if (! isokendpt(m_ptr->RT_ENDPT, &proc_nr)) {
      return (EINVAL);
  }

  /* get pointer to process from proc_nr */
  rp = proc_addr(proc_nr);

  /* check if process is real-time */
  if (! is_rtp(rp) ) {
      return (EPERM);
  }

  if (rp->p_rts_flags == 0) {
      /* p_rts_flags cannot be 0 because
       * the process is waiting for a reply from
       * PM
       */
      return (EPERM);
  }

  /* remove process from the block list.
   * We first have to find the process in the list.
```

```
     */
    xpp = &edf_block_head;
    while (*xpp != NIL_PROC && *xpp != rp) {
        xpp = &(*xpp)->p_rt_link;
    }

    /* *xpp == NIL_PROC if the list was empty or process not found.
     * This may not happen normally.
     */
    if (*xpp != NIL_PROC) {
        /* Remove the process from the list.
         * We set the pointer that points to the process we remove
         * to the process that the process we remove is linked to.
         */
        *xpp = (*xpp)->p_rt_link;
    }

    /* Add process to the wait queue.
     * This quueue is sorted on next period start.
     * We will have to find the right place in the queue first.
     */
    xpp = &edf_wait_head;
    while (*xpp != NIL_PROC && (*xpp)->p_rt_nextperiod <= rp->p_rt_nextperiod) {
        xpp = &(*xpp)->p_rt_link;
    }

    /* Add process to the queue */
    rp->p_rt_link = *xpp;
    *xpp = rp;

    /* Make sure that no other code will schedule this process
     * until the new period starts by setting the NEXTPERIOD bit.
     * If p_rts_flags is not 0, a process is not runnable.
     */
    rp->p_rts_flags |= NEXTPERIOD;

    #if 0
    kprintf("%d %s gave up %d/%d ticks\n", rp->p_endpoint, rp->p_name,
        rp->p_rt_ticksleft, rp->p_rt_calctime);
    #endif

    return (OK);
}
```

Listing 37: /usr/src/kernel/system/do_rt_set.c

```
/* The kernel call implemented in this file:
 *   m_type:    SYS_RT_SET
 *
 * The parameters for this kernel call are:
 *    m7_i1     RT_ENDPT        endpoint of process that wants to be RT
 *    m7_i2     RT_SCHED        Scheduler type
 *    m7_i3     RM_PRIO         rate-monotonic static priority (used by RM only)
 *    m7_i3     EDF_PERIOD      period of process (used by EDF only)
 *    m7_i4     EDF_CALCTIME    calculation time per period (used by EDF only)
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/rt.h>

FORWARD _PROTOTYPE( int do_rt_set_rm, (message *m_ptr, struct proc *rp) );
FORWARD _PROTOTYPE( int do_rt_set_edf, (message *m_ptr, struct proc *rp) );

/*===========================================================================*
 *                  do_rt_set                                                *
 *===========================================================================*/
PUBLIC int do_rt_set(message *m_ptr)
{
  /* Transform a normal user process into a real-time process */

  struct proc *rp; /* pointer to process that wants to be real-time */
  int proc_nr; /* process number of process that wants to be real-time */

  /* if scheduler is undefined we cannot
   * make processes real-time
   */
  if (rt_sched == SCHED_UNDEFINED) {
      return (EPERM);
  }

  /* if rt_sched is not equal to the
   * scheduler defined in the message
   * we cannot make this process real-time
   */
  if (rt_sched != m_ptr->RT_SCHED) {
      return (EINVAL);
  }

  /* check if endpoint is valid and convert endpoint
   * to process number stored in proc_nr
   */
  if (! isokendpt(m_ptr->RT_ENDPT, &proc_nr)) {
      return (EINVAL);
  }

  /* get pointer to process from process number */
  rp = proc_addr(proc_nr);

  /* a process that is already real-time may
   * not call this function.
   */
  if (is_rtp(rp)) {
      return (EPERM);
  }
  /* Dispatch to the right function.
   * Each scheduler type has its own function.
   */
  switch (rt_sched) {
      case SCHED_RM:
          return do_rt_set_rm(m_ptr, rp);
      case SCHED_EDF:
          return do_rt_set_edf(m_ptr, rp);
      default:
          return (EINVAL);
  }

  /* should not be reached */
  return (EPERM);
}


/*===========================================================================*
 *                   do_rt_set_rm                                            *
 *===========================================================================*/
PRIVATE int do_rt_set_rm(message *m_ptr, struct proc *rp)
{
  struct proc *xp;

  /* if rm_prio_policy equals PRIO_UNIQUE the priority of a
   * process scheduled with RM should be unique. We will have to
   * loop through the process table to check if there is no other process
   * with this priority.
   */
  if (rm_prio_policy == PRIO_UNIQUE) {
      for (xp = BEG_PROC_ADDR; xp < END_PROC_ADDR; ++xp) {
          if (xp->p_rts_flags != SLOT_FREE &&
              is_rtp(xp) && xp->p_rt_priority == m_ptr->RM_PRIO) {
              return (EINVAL);
          }
      }
  }

  if ( rp->p_rts_flags == 0) {
      /* Should not happen normally.
       * A process is runnable if p_rts_flags is zero.
       * The process requesting to be real-time should be
       * blocked waiting for a reply from PM
       * and thus p_rts_flags should not be zero.
       * Still we remove the process from the scheduling queue.
       */
       lock_dequeue(rp);
  }

  /* Now we can change the process structure.
   * First make sure this process will be recognized
   * as a real-time process.
   */
   rp->p_rt = 1;

  /* Set the current and maximum priority to the
   * real-time queue.
   */
  rp->p_max_priority = rp->p_priority = RT_Q;

  /* set the static priority */
  rp->p_rt_priority = m_ptr->RM_PRIO;

  if ( rp->p_rts_flags == 0) {
      /* Should not happen normally.
       * See above. Add process to the scheduling queue.
       */
      lock_enqueue(rp);
  }

  return (OK);
}

/*===========================================================================*
 *                      do_rt_set_edf                                        *
 *===========================================================================*/
PRIVATE int do_rt_set_edf(message *m_ptr, struct proc * rp)
{
  /* check if period and calculation time
   * parameters are valid
   */
  if (m_ptr->EDF_PERIOD <= 0 || m_ptr->EDF_CALCTIME <= 0) {
      return (EINVAL);
  }

  /* check if calctime is not larger than period */
  if (m_ptr->EDF_CALCTIME > m_ptr->EDF_PERIOD) {
      return (EINVAL);
  }

  if ( rp->p_rts_flags == 0) {
      /* Should not happen normally.
       * A process is runnable if p_rts_flags is zero.
       * The process requesting to be real-time should be
       * blocked waiting for a reply from PM
       * and thus p_rts_flags should not be zero.
       * In the case of EDF this is fatal.
       */
       kprintf("systask: do_rt_set (edf): process should not be runnable\n");
       return (EGENERIC);
  }

  /* Now we can change the process structure.
   * First make sure this process will be recognized
   * as a real-time process.
   */
   rp->p_rt = 1;

   /* set the period and calculation time */
   rp->p_rt_period = m_ptr->EDF_PERIOD;
   rp->p_rt_calctime = m_ptr->EDF_CALCTIME;

   /* The first period starts immediately,
    * so set the number of ticks left in this period
    * to the requested calculation time.
    */
   rp->p_rt_ticksleft = rp->p_rt_calctime;

   /* ticks left till next period equals the specified period */
   rp->p_rt_nextperiod = rp->p_rt_period;

   /* use the EDF scheduling queue */
   rp->p_priority = rp->p_max_priority = RT_Q;
```

```
    /* The process should be blocked waiting for a reply
     * from PM. So we add the process to the (front of) block list.
     */
    rp->p_rt_link = edf_block_head;
    edf_block_head = rp;

    /* we did not modify edf_rp or the edf run queue so we don't have
     * to call edf_sched()
     */

    return (OK);
}
```

Listing 38: /usr/src/kernel/system/do_rt_set_sched.c

```c
/* The kernel call implemented in this file:
 *   m_type:    SYS_RT_SET_SCHED
 *
 * The parameters for this kernel call are:
 *    m7_i2:    RT_SCHED    Scheduler type
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/rt.h>


/*===========================================================================*
 *                  do_rt_set_sched
                  *
 *===========================================================================*/
PUBLIC int do_rt_set_sched(message *m_ptr)
{
  /* Set the real-time scheduler.
   * Currently supported schedulers are
   * Rate-Monotonic (SCHED_RM) and
   * Earliest Deadline First (SCHED_EDF).
   */

  /* Check if allowed to change the RT scheduler.
   * It is only allowed to change the scheduler if
   * rt_sched is SCHED_UNDEFINED or rt_sched is set
   * to a scheduler but there are no real-time processes
   * running.
   */
  if (rt_sched != SCHED_UNDEFINED) {
      register struct proc * xp;

      /* search the process table for RT processes */
      for (xp = BEG_PROC_ADDR; xp < END_PROC_ADDR; ++xp) {
          if (is_rtp(xp)) {
              return (EPERM); /* found RT process */
          }
      }
  }

  /* We are allowed to set the scheduler.
   * Check if the scheduler type is valid
   * and set rt_sched to the new scheduler type
   */
  switch (m_ptr->RT_SCHED) {
      case SCHED_UNDEFINED:
          rt_sched = m_ptr->RT_SCHED;
          break;
      case SCHED_RM:
          if (m_ptr->RM_PRIO_POLICY != PRIO_UNIQUE &&
              m_ptr->RM_PRIO_POLICY != PRIO_NOT_UNIQUE) {
              return (EINVAL); /* invalid policy parameter */
          }

          rt_sched = m_ptr->RT_SCHED;
          rm_prio_policy = m_ptr->RM_PRIO_POLICY;
      case SCHED_EDF:
          rt_sched = m_ptr->RT_SCHED;
          break;
      default:
          return (EINVAL); /* invalid scheduler */
          break;
  }

  return (OK);
}
```

Listing 39: /usr/src/kernel/system/do_rt_set_sched_bridge.c

```
/* The kernel call implemented in this file:
 *   m_type:     SYS_RT_SCHED_BRIDGE
 *
 * The parameters for this kernel call are:
 *    m1_i1:     RT_SCHED_BRIDGE     Scheduler bridge state
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/rt.h>


/*===========================================================================*
 *                  do_rt_set_sched                                          *
 *===========================================================================*/
PUBLIC int do_rt_set_sched_bridge(message *m_ptr)
{
  /* Set the scheduling bridge state.
   * If set to 0 high priority processes that are lowered in priority
   * are not allowed to get a lower priority than RT processes.
   * If set to 1 they can. Before Minix shutdowns it will set it to 1 because
   * otherwise the system cannot shutdown properly.
   */
  rt_sched_bridge = m_ptr->RT_SCHED_BRIDGE;

  return (OK);
}
```

Listing 40: /usr/src/kernel/system/do_rt_show_data.c

```c
/* The kernel call implemented in this file:
 *   m_type:    SYS_RT_SHOW_DATA
 *
 * The parameters for this kernel call are:
 *    none
 */

#include "../system.h"
#include <minix/type.h>
#include <sys/resource.h>
#include <minix/rt.h>


/*===========================================================================*
 *                  do_rt_show_data                                          *
 *===========================================================================*/
PUBLIC int do_rt_show_data(message *m_ptr)
{
  /* dumpe scheduling info if a real-time scheduler is set */
  if (rt_sched != SCHED_UNDEFINED) {
      show_rt_data();
  } else {
      kprintf("No Real-time scheduler set!\n");
  }

  return (OK);
}
```

Listing 41: /usr/src/servers/pm/forkexit.c

```c
/* This file deals with creating processes (via FORK) and deleting them (via
 * EXIT/WAIT).  When a process forks, a new slot in the 'mproc' table is
 * allocated for it, and a copy of the parent's core image is made for the
 * child.  Then the kernel and file system are informed.  A process is removed
 * from the 'mproc' table when two events have occurred: (1) it has exited or
 * been killed by a signal, and (2) the parent has done a WAIT.  If the process
 * exits first, it continues to occupy a slot until the parent does a WAIT.
 *
 * The entry points into this file are:
 *   do_fork:    perform the FORK system call
 *   do_pm_exit: perform the EXIT system call (by calling pm_exit())
 *   pm_exit:    actually do the exiting
 *   do_wait:    perform the WAITPID or WAIT system call
 */

#include "pm.h"
#include <sys/wait.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <sys/resource.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"
#include <lib.h>
#include <minix/sem.h>

#define LAST_FEW            2   /* last few slots reserved for superuser */

FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );

/*===========================================================================*
 *                              do_fork                                      *
 *===========================================================================*/
PUBLIC int do_fork()
{
/* The process pointed to by 'mp' has forked.  Create a child process. */
  register struct mproc *rmp;   /* pointer to parent */
  register struct mproc *rmc;   /* pointer to child */
  int child_nr, s;
  phys_clicks prog_clicks, child_base;
  phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */
  pid_t new_pid;
  static int next_child;
  int n = 0, r;

 /* If tables might fill up during FORK, don't even start since recovery half
  * way through is such a nuisance.
  */
  rmp = mp;
  if ((procs_in_use == NR_PROCS) ||
                (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0))
  {
        printf("PM: warning, process table is full!\n");
        return(EAGAIN);
  }

  /* Determine how much memory to allocate.  Only the data and stack need to
   * be copied, because the text segment is either shared or of zero length.
   */
  prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
  prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
  if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(ENOMEM);

  /* Create a copy of the parent's core image for the child. */
  child_abs = (phys_bytes) child_base << CLICK_SHIFT;
  parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
  s = sys_abscopy(parent_abs, child_abs, prog_bytes);
  if (s < 0) panic(__FILE__,"do_fork can't copy", s);

  /* Find a slot in 'mproc' for the child process.  A slot must exist. */
  do {
        next_child = (next_child+1) % NR_PROCS;
        n++;
  } while((mproc[next_child].mp_flags & IN_USE) && n <= NR_PROCS);
  if(n > NR_PROCS)
        panic(__FILE__,"do_fork can't find child slot", NO_NUM);
  if(next_child < 0 || next_child >= NR_PROCS
 || (mproc[next_child].mp_flags & IN_USE))
        panic(__FILE__,"do_fork finds wrong child slot", next_child);

  rmc = &mproc[next_child];
  /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
  child_nr = (int)(rmc - mproc);        /* slot number of the child */
  procs_in_use++;
  *rmc = *rmp;                  /* copy parent's process slot to child's */
  rmc->mp_parent = who_p;                       /* record child's parent */
  /* inherit only these flags */
  rmc->mp_flags &= (IN_USE|SEPARATE|PRIV_PROC|DONT_SWAP);
  rmc->mp_child_utime = 0;              /* reset administration */
  rmc->mp_child_stime = 0;              /* reset administration */

  /* A separate I&D child keeps the parents text segment.  The data and stack
   * segments must refer to the new copy.
   */
  if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
  rmc->mp_seg[D].mem_phys = child_base;
  rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
                        (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  rmc->mp_exitstatus = 0;
  rmc->mp_sigstatus = 0;

  /* Find a free pid for the child and put it in the table. */
  new_pid = get_free_pid();
  rmc->mp_pid = new_pid;        /* assign pid to child */

  /* Tell kernel and file system about the (now successful) FORK. */
  if((r=sys_fork(who_e, child_nr, &rmc->mp_endpoint)) != OK) {
        panic(__FILE__,"do_fork can't sys_fork", r);
  }
  tell_fs(FORK, who_e, rmc->mp_endpoint, rmc->mp_pid);

  /* Report child's memory map to kernel. */
  if((r=sys_newmap(rmc->mp_endpoint, rmc->mp_seg)) != OK) {
        panic(__FILE__,"do_fork can't sys_newmap", r);
  }

  /* Reply to child to wake it up. */
  setreply(child_nr, 0);                /* only parent gets details */
  rmp->mp_reply.endpt = rmc->mp_endpoint;       /* child's process number */

  return(new_pid);                      /* child's pid */
```

```
}

/*===========================================================================*
 *                              do_pm_exit                                   *
 *===========================================================================*/
PUBLIC int do_pm_exit()
{
/* Perform the exit(status) system call. The real work is done by pm_exit(),
 * which is also called when a process is killed by a signal.
 */
  pm_exit(mp, m_in.status);
  return(SUSPEND);              /* can't communicate from beyond the grave */
}

/*===========================================================================*
 *                              pm_exit                                      *
 *===========================================================================*/
PUBLIC void pm_exit(rmp, exit_status)
register struct mproc *rmp;     /* pointer to the process to be terminated */
int exit_status;                /* the process' exit status (for parent) */
{
/* A process is done.  Release most of the process' possessions.  If its
 * parent is waiting, release the rest, else keep the process slot and
 * become a zombie.
 */
  register int proc_nr, proc_nr_e;
  int parent_waiting, right_child, r;
  pid_t pidarg, procgrp;
  struct mproc *p_mp;
  clock_t t[5];
  message m; /* used for call to semaphore server */

  proc_nr = (int) (rmp - mproc);        /* get process slot number */
  proc_nr_e = rmp->mp_endpoint;

  /* Remember a session leader's process group. */
  procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;

  /* If the exited process has a timer pending, kill it. */
  if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr_e, (unsigned) 0);

  /* Do accounting: fetch usage times and accumulate at parent. */
  if((r=sys_times(proc_nr_e, t)) != OK)
        panic(__FILE__,"pm_exit: sys_times failed", r);

  p_mp = &mproc[rmp->mp_parent];                        /* process' parent */
  p_mp->mp_child_utime += t[0] + rmp->mp_child_utime;   /* add user time */
  p_mp->mp_child_stime += t[1] + rmp->mp_child_stime;   /* add system time */

  /* Tell the kernel the process is no longer runnable to prevent it from
   * being scheduled in between the following steps. Then tell FS that it
   * the process has exited and finally, clean up the process at the kernel.
   * This order is important so that FS can tell drivers to cancel requests
   * such as copying to/ from the exiting process, before it is gone.
   */
  sys_nice(proc_nr_e, PRIO_STOP);       /* stop the process */
  if(proc_nr_e != FS_PROC_NR)           /* if it is not FS that is exiting.. */
        tell_fs(EXIT, proc_nr_e, 0, 0);         /* tell FS to free the slot */
  else
        printf("PM: FS died\n");
  if((r=sys_exit(proc_nr_e)) != OK)     /* destroy the process */
        panic(__FILE__,"pm_exit: sys_exit failed", r);
  /* inform semaphore server that a process has exited */
  m.SEM_F_ENDPT = proc_nr_e;
  m.m_type = SEM_PROCEXIT;
  send(SS,&m);

  /* Pending reply messages for the dead process cannot be delivered. */
  rmp->mp_flags &= ~REPLY;

  /* Release the memory occupied by the child. */
  if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
        /* No other process shares the text segment, so free it. */
        free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
  }
  /* Free the data and stack segments. */
  free_mem(rmp->mp_seg[D].mem_phys,
      rmp->mp_seg[S].mem_vir
        + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

  /* The process slot can only be freed if the parent has done a WAIT. */
  rmp->mp_exitstatus = (char) exit_status;

  pidarg = p_mp->mp_wpid;                /* who's being waited for? */
  parent_waiting = p_mp->mp_flags & WAITING;
  right_child =                          /* child meets one of the 3 tests? */
        (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp);

  if (parent_waiting && right_child) {
        cleanup(rmp);                    /* tell parent and release child slot */
  } else {
        rmp->mp_flags = IN_USE|ZOMBIE;   /* parent not waiting, zombify child */
        sig_proc(p_mp, SIGCHLD);         /* send parent a "child died" signal */
  }

  /* If the process has children, disinherit them.  INIT is the new parent. */
  for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
        if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
                /* 'rmp' now points to a child to be disinherited. */
                rmp->mp_parent = INIT_PROC_NR;
                parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
                if (parent_waiting && (rmp->mp_flags & ZOMBIE)) cleanup(rmp);
        }
  }

  /* Send a hangup to the process' process group if it was a session leader. */
  if (procgrp != 0) check_sig(-procgrp, SIGHUP);
}

/*===========================================================================*
 *                              do_waitpid                                   *
 *===========================================================================*/
PUBLIC int do_waitpid()
{
/* A process wants to wait for a child to terminate. If a child is already
 * waiting, go clean it up and let this WAIT call terminate.  Otherwise,
 * really wait.
 * A process calling WAIT never gets a reply in the usual way at the end
 * of the main loop (unless WNOHANG is set or no qualifying child exists).
 * If a child has already exited, the routine cleanup() sends the reply
 * to awaken the caller.
 * Both WAIT and WAITPID are handled by this code.
 */
```

```
  register struct mproc *rp;
  int pidarg, options, children;

  /* Set internal variables, depending on whether this is WAIT or WAITPID. */
  pidarg  = (call_nr == WAIT ? -1 : m_in.pid);      /* 1st param of waitpid */
  options = (call_nr == WAIT ?  0 : m_in.sig_nr);   /* 3rd param of waitpid */
  if (pidarg == 0) pidarg = -mp->mp_procgrp;    /* pidarg < 0 ==> proc grp */

  /* Is there a child waiting to be collected? At this point, pidarg != 0:
   *    pidarg  >  0 means pidarg is pid of a specific process to wait for
   *    pidarg == -1 means wait for any child
   *    pidarg  < -1 means wait for any child whose process group = -pidarg
   */
  children = 0;
  for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
        if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who_p) {
                /* The value of pidarg determines which children qualify. */
                if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
                if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;

                children++;             /* this child is acceptable */
                if (rp->mp_flags & ZOMBIE) {
                        /* This child meets the pid test and has exited. */
                        cleanup(rp);    /* this child has already exited */
                        return(SUSPEND);
                }
                if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
                        /* This child meets the pid test and is being traced.*/
                        mp->mp_reply.reply_res2 = 0177|(rp->mp_sigstatus << 8);
                        rp->mp_sigstatus = 0;
                        return(rp->mp_pid);
                }
        }
  }

  /* No qualifying child has exited.  Wait for one, unless none exists. */
  if (children > 0) {
        /* At least 1 child meets the pid test exists, but has not exited. */
        if (options & WNOHANG) return(0);    /* parent does not want to wait */
        mp->mp_flags |= WAITING;             /* parent wants to wait */
        mp->mp_wpid = (pid_t) pidarg;        /* save pid for later */
        return(SUSPEND);                     /* do not reply, let it wait */
  } else {
        /* No child even meets the pid test.  Return error immediately. */
        return(ECHILD);                      /* no - parent has no children */
  }
}

/*===========================================================================*
 *                              cleanup                                      *
 *===========================================================================*/
PRIVATE void cleanup(child)
register struct mproc *child;   /* tells which process is exiting */
{
/* Finish off the exit of a process.  The process has exited or been killed
 * by a signal, and its parent is waiting.
 */
  struct mproc *parent = &mproc[child->mp_parent];
  int exitstatus;

  /* Wake up the parent by sending the reply message. */
  exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
  parent->mp_reply.reply_res2 = exitstatus;
  setreply(child->mp_parent, child->mp_pid);
  parent->mp_flags &= ~WAITING;         /* parent no longer waiting */

  /* Release the process table entry and reinitialize some field. */
  child->mp_pid = 0;
  child->mp_flags = 0;
  child->mp_child_utime = 0;
  child->mp_child_stime = 0;
  procs_in_use--;
}
```

Listing 42: /usr/src/servers/pm/klog.c

```
/* System calls for kernel logging.     Author: Bianco Zandbergen
 *                                                           19 May 2009
 * The entry points into this file are:
 *   do_klog_set: set the kernel log state
 *   do_klog_copy: copy the kernel log to a user process
 */

#include "pm.h"
#include <minix/klog.h>

/*===========================================================================*
 *                              do_klog_set
                                      *
 *===========================================================================*/
PUBLIC int do_klog_set(void)
{
  /* Set the kernel log state.
   * Invoke the system library function to do the kernel call.
   */
  return(sys_klog_set(m_in.KLOG_STATE, m_in.KLOG_TYPE));
}

/*===========================================================================*
 *                              do_klog_copy
                                      *
 *===========================================================================*/
PUBLIC int do_klog_copy(void)
{
  /* Copy the kernel log from the kernel to the user process.
   * Invoke the system library function to do the kernel call.
   * m_in.m_source is the endpoint of the user process.
   * m_in.KLOG_PTR is the pointer to the kernel log data structure
   * of the user process.
   */
  return(sys_klog_copy(m_in.m_source, m_in.KLOG_PTR));
}
```

Listing 43: /usr/src/servers/pm/proto.h

```
/* Function prototypes. */

struct mproc;
struct stat;
struct mem_map;
struct memory;

#include <timers.h>

/* alloc.c */
_PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks)                   );
_PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks)         );
_PROTOTYPE( void mem_init, (struct memory *chunks, phys_clicks *free)     );
#if ENABLE_SWAP
_PROTOTYPE( int swap_on, (char *file, u32_t offset, u32_t size) );
_PROTOTYPE( int swap_off, (void)                                          );
_PROTOTYPE( void swap_in, (void)                                          );
_PROTOTYPE( void swap_inqueue, (struct mproc *rmp)                        );
#else /* !SWAP */
#define swap_in()                       ((void)0)
#define swap_inqueue(rmp)               ((void)0)
#endif /* !SWAP */
_PROTOTYPE(int mem_holes_copy, (struct hole *, size_t *, u32_t *)         );

/* break.c */
_PROTOTYPE( int adjust, (struct mproc *rmp,
                        vir_clicks data_clicks, vir_bytes sp)             );
_PROTOTYPE( int do_brk, (void)                                            );
_PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
                        vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) );

/* devio.c */
_PROTOTYPE( int do_dev_io, (void) );
_PROTOTYPE( int do_dev_io, (void) );

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (void)                                           );

/* exec.c */
_PROTOTYPE( int do_exec, (void)                                           );
_PROTOTYPE( void rw_seg, (int rw, int fd, int proc, int seg,
                                                phys_bytes seg_bytes)     );
_PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
                        Dev_t dev, time_t ctime)                          );

/* forkexit.c */
_PROTOTYPE( int do_fork, (void)                                           );
_PROTOTYPE( int do_pm_exit, (void)                                        );
_PROTOTYPE( int do_waitpid, (void)                                        );
_PROTOTYPE( void pm_exit, (struct mproc *rmp, int exit_status)            );

/* getset.c */
_PROTOTYPE( int do_getset, (void)                                         );

/* main.c */
_PROTOTYPE( int main, (void)                                              );

/* misc.c */
_PROTOTYPE( int do_reboot, (void)                                         );
_PROTOTYPE( int do_procstat, (void)                                       );
_PROTOTYPE( int do_getsysinfo, (void)                              );
_PROTOTYPE( int do_getprocnr, (void)                               );
_PROTOTYPE( int do_svrctl, (void)                                  );
_PROTOTYPE( int do_allocmem, (void)                                );
_PROTOTYPE( int do_freemem, (void)                                 );
_PROTOTYPE( int do_getsetpriority, (void)                                  );
_PROTOTYPE( ssize_t _read_pm, (int _fd, void *_buf, size_t _n, int s, int e));
_PROTOTYPE( ssize_t _write_pm, (int _fd, void *_buf, size_t _n, int s, int e));


#if (MACHINE == MACINTOSH)
_PROTOTYPE( phys_clicks start_click, (void)                        );
#endif

_PROTOTYPE( void setreply, (int proc_nr, int result)               );

/* signal.c */
_PROTOTYPE( int do_alarm, (void)                                   );
_PROTOTYPE( int do_kill, (void)                                    );
_PROTOTYPE( int ksig_pending, (void)                               );
_PROTOTYPE( int do_pause, (void)                                   );
_PROTOTYPE( int set_alarm, (int proc_nr, int sec)                  );
_PROTOTYPE( int check_sig, (pid_t proc_id, int signo)              );
_PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr)         );
_PROTOTYPE( int do_sigaction, (void)                               );
_PROTOTYPE( int do_sigpending, (void)                              );
_PROTOTYPE( int do_sigprocmask, (void)                             );
_PROTOTYPE( int do_sigreturn, (void)                               );
_PROTOTYPE( int do_sigsuspend, (void)                              );
_PROTOTYPE( void check_pending, (struct mproc *rmp)                );

/* time.c */
_PROTOTYPE( int do_stime, (void)                                   );
_PROTOTYPE( int do_time, (void)                                    );
_PROTOTYPE( int do_times, (void)                                   );
_PROTOTYPE( int do_gettimeofday, (void)                            );

/* timers.c */
_PROTOTYPE( void pm_set_timer, (timer_t *tp, int delta,
        tmr_func_t watchdog, int arg));
_PROTOTYPE( void pm_expire_timers, (clock_t now));
_PROTOTYPE( void pm_cancel_timer, (timer_t *tp));

/* trace.c */
_PROTOTYPE( int do_trace, (void)                                   );
_PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr)        );

/* utility.c */
_PROTOTYPE( pid_t get_free_pid, (void)                             );
_PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
_PROTOTYPE( int no_sys, (void)                                     );
_PROTOTYPE( void panic, (char *who, char *mess, int num)           );
_PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3)       );
_PROTOTYPE( int get_stack_ptr, (int proc_nr, vir_bytes *sp)        );
_PROTOTYPE( int get_mem_map, (int proc_nr, struct mem_map *mem_map)     );
_PROTOTYPE( char *find_param, (const char *key));
_PROTOTYPE( int proc_from_pid, (pid_t p));
_PROTOTYPE( int pm_isokendpt, (int ep, int *proc));

/* rt.c */
_PROTOTYPE( int do_rt_set_sched, (void) );
_PROTOTYPE( int do_rt_set, (void) );
_PROTOTYPE( int do_rt_nextperiod, (void) );
_PROTOTYPE( int do_rt_set_sched_bridge, (void) );

/* klog.c */
_PROTOTYPE( int do_klog_set, (void) );
_PROTOTYPE( int do_klog_copy, (void) );
```

Listing 44: /usr/src/servers/pm/rt.c

```
/* System calls for real-time.  Author: Bianco Zandbergen
 *                                                                19 May 2009
 * The entry points into this file are:
 *   do_rt_set_sched: set the real-time scheduler type
 *   do_rt_set: transform a normal process to real-time process
 *   do_rt_nextperiod: give up remaining calculation time for EDF processes
 *   do_rt_set_sched_bridge: set RT scheduling bridge state
 */

#include "pm.h"
#include <minix/rt.h>

/*===========================================================================*
 *                              do_rt_set_sched
                                      *
 *===========================================================================*/
PUBLIC int do_rt_set_sched(void)
{
  /* Set the Real-time scheduler to use.
   * This setting is system wide and all real-time processes should use the same
   * RT scheduler.
   */
  if (m_in.RT_SCHED == SCHED_EDF) {
      /* Received a request to set the RT scheduler to Earliest Deadline First.
       * Invoke the system library function to do the kernel call and
       * return the result.
       * The second parameter is only used by RM, so we set it to 0.
       */
      return (sys_rt_set_sched(SCHED_EDF, 0));
  } else if (m_in.RT_SCHED == SCHED_RM) {
      /* Received a request to set the RT scheduler to Rate-Monotonic.
       * Invoke the system library function that will do the kernel call and
       * return the result.
       * The second parameter defines wether a RM priority should be unique or
         not.
       */
      return (sys_rt_set_sched(SCHED_RM, m_in.RM_PRIO_POLICY));
  } else {
      /* unknown scheduler type */
      return (EINVAL);
  }
}

/*===========================================================================*
 *                              do_rt_set
                                      *
 *===========================================================================*/
PUBLIC int do_rt_set(void)
{
  /* Transform a non real-time process to real-time.
   * The scheduler type should equal to the current RT scheduler set by
   * do_rt_set_sched(), checks for this are done in the kernel.
   */
  switch (m_in.RT_SCHED) {
      case SCHED_RM:
          /* Process wants to be real-time and use Rate-Monotonic scheduling.
           * Invoke the system library function to do the kernel call and return
           * the result. m_in.m_source is the endpoint of the process that called
           * the system call. Rate-Monotonic processes will have a RM priority
             besides
```

```
            * the Minix scheduling priority. All RM processes share the same Minix
               priority.
            * The priority among RM processes is defined by the RM priority.
            * Last parameter is not used so we set it to 0.
            */
           return (sys_rt_set(m_in.m_source, SCHED_RM, m_in.RM_PRIO, 0));
       case SCHED_EDF:
           /* Process wants to be real-time and use Earliest Deadline First
              scheduling.
            * Invoke the system library function to do the kernel call and return
               the
            * result. An EDF process is a periodic process, so we specify the
               period in
            * system ticks. Within each period the process reserves calculation
               time that
            * is also specified in ticks. If an EDF process does not get the
               calculation time
            * it reserved before a new period starts it has missed a deadline.
            */
           return (sys_rt_set(m_in.m_source, SCHED_EDF, m_in.EDF_PERIOD,
              m_in.EDF_CALCTIME));
       default:
           /* unknown scheduler type */
           return (EINVAL);
  }
}

/*===========================================================================*
 *                              do_rt_nextperiod
                                      *
 *===========================================================================*/
PUBLIC int do_rt_nextperiod(void)
{
  /* Give up remaining calculation time and wait till next period start.
   * This system call may only be used by processes that are scheduled
   * using Earliest Deadline First.
   * Invoke the system library function to do the kernel call and return the
      result.
   * m_in.m_source is the endpoint of the requestion process.
   */
  return (sys_rt_nextperiod(m_in.m_source));
}

/*===========================================================================*
 *                              do_rt_set_sched_bridge
                                      *
 *===========================================================================*/
PUBLIC int do_rt_set_sched_bridge(void)
{
  /* Set the real-time scheduling bridge state.
   * m_in.RT_SCHED_BRIDGE is the state that the requesting process wants it
   * to set to. Invokes the system library function to do the kernel call
   * and return the result. Process priorities can be defined in three parts:
   * highest priority are system processes, second highest priority is for
      real-time
   * processes and lowest priorities are for normal user processes. If state is
      set to 0
   * system processes are not allowed to get a lower priority than real-time
      processes.
   * If the state is set to 1 they can. A system process can get a lower priority
      if it uses the
   * full quantum. System starts up with state set to 0. Before shutdown the
       state will be set by shutdown
   * to 1 otherwise the system will not proper shutdown.
   * This is the only reason for the existance of this functionality.
   */
  return (sys_rt_sched_bridge(m_in.RT_SCHED_BRIDGE));
}
```

Listing 45: /usr/src/servers/pm/table.c

```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "pm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/* Miscellaneous */
char core_name[] = "core";      /* file name where core images are produced */

_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_pm_exit,     /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        no_sys,         /*  3 = read    */
        no_sys,         /*  4 = write   */
        no_sys,         /*  5 = open    */
        no_sys,         /*  6 = close   */
        do_waitpid,     /*  7 = wait    */
        no_sys,         /*  8 = creat   */
        no_sys,         /*  9 = link    */
        no_sys,         /* 10 = unlink  */
        do_waitpid,     /* 11 = waitpid */
        no_sys,         /* 12 = chdir   */
        do_time,        /* 13 = time    */
        no_sys,         /* 14 = mknod   */
        no_sys,         /* 15 = chmod   */
        no_sys,         /* 16 = chown   */
        do_brk,         /* 17 = break   */
        no_sys,         /* 18 = stat    */
        no_sys,         /* 19 = lseek   */
        do_getset,      /* 20 = getpid  */
        no_sys,         /* 21 = mount   */
        no_sys,         /* 22 = umount  */
        do_getset,      /* 23 = setuid  */
        do_getset,      /* 24 = getuid  */
        do_stime,       /* 25 = stime   */
        do_trace,       /* 26 = ptrace  */
        do_alarm,       /* 27 = alarm   */
        no_sys,         /* 28 = fstat   */
        do_pause,       /* 29 = pause   */
        no_sys,         /* 30 = utime   */
        no_sys,         /* 31 = (stty)  */
        no_sys,         /* 32 = (gtty)  */
        no_sys,         /* 33 = access  */
        no_sys,         /* 34 = (nice)  */
        no_sys,         /* 35 = (ftime) */
        no_sys,         /* 36 = sync    */
        do_kill,        /* 37 = kill    */
        no_sys,         /* 38 = rename  */
        no_sys,         /* 39 = mkdir   */
        no_sys,         /* 40 = rmdir   */
        no_sys,         /* 41 = dup     */
        no_sys,         /* 42 = pipe    */
        do_times,       /* 43 = times   */
        no_sys,          /* 44 = (prof)  */
        do_rt_set_sched,/* 45 = rt_set_sched */
        do_getset,       /* 46 = setgid  */
        do_getset,       /* 47 = getgid  */
        no_sys,          /* 48 = (signal)*/
        do_klog_set,/* 49 = klog_set */
        do_klog_copy,/* 50 = klog_copy */
        no_sys,          /* 51 = (acct)  */
        no_sys,          /* 52 = (phys)  */
        no_sys,          /* 53 = (lock)  */
        no_sys,          /* 54 = ioctl   */
        no_sys,          /* 55 = fcntl   */
        no_sys,          /* 56 = (mpx)   */
        do_rt_set,       /* 57 = rt_set  */
        do_rt_nextperiod, /* 58 = rt_nextperiod */
        do_exec,         /* 59 = execve  */
        no_sys,          /* 60 = umask   */
        no_sys,          /* 61 = chroot  */
        do_getset,       /* 62 = setsid  */
        do_getset,       /* 63 = getpgrp */
        do_rt_set_sched_bridge, /* 64 = rt_set_sched_bridge */
        no_sys,          /* 65 = UNPAUSE */
        no_sys,          /* 66 = unused  */
        no_sys,          /* 67 = REVIVE  */
        no_sys,          /* 68 = TASK_REPLY  */
        no_sys,          /* 69 = unused  */
        no_sys,          /* 70 = unused  */
        do_sigaction,    /* 71 = sigaction   */
        do_sigsuspend,   /* 72 = sigsuspend  */
        do_sigpending,   /* 73 = sigpending  */
        do_sigprocmask,  /* 74 = sigprocmask */
        do_sigreturn,    /* 75 = sigreturn   */
        do_reboot,       /* 76 = reboot  */
        do_svrctl,       /* 77 = svrctl  */
        do_procstat,     /* 78 = procstat */
        do_getsysinfo,   /* 79 = getsysinfo */
        do_getprocnr,    /* 80 = getprocnr */
        no_sys,          /* 81 = unused */
        no_sys,          /* 82 = fstatfs */
        do_allocmem,     /* 83 = memalloc */
        do_freemem,      /* 84 = memfree */
        no_sys,          /* 85 = select */
        no_sys,          /* 86 = fchdir */
        no_sys,          /* 87 = fsync */
        do_getsetpriority,       /* 88 = getpriority */
        do_getsetpriority,       /* 89 = setpriority */
        do_time,         /* 90 = gettimeofday */
        do_getset,       /* 91 = seteuid */
        do_getset,       /* 92 = setegid */
        no_sys,          /* 93 = truncate */
        no_sys,          /* 94 = ftruncate */
};
/* This should not fail with "array size is negative": */
extern int dummy[sizeof(call_vec) == NCALLS * sizeof(call_vec[0]) ? 1 : -1];
```

Listing 46: /usr/src/servers/is/dmp.c

```
/* This file contains information dump procedures. During the initialization
 * of the Information Service 'known' function keys are registered at the TTY
 * server in order to receive a notification if one is pressed. Here, the
 * corresponding dump procedure is called.
 *
 * The entry points into this file are
 *   handle_fkey:       handle a function key pressed notification
 */

#include "inc.h"

/* Define hooks for the debugging dumps. This table maps function keys
 * onto a specific dump and provides a description for it.
 */
#define NHOOKS 19

struct hook_entry {
        int key;
        void (*function)(void);
        char *name;
} hooks[NHOOKS] = {
        { F1,   proctab_dmp, "Kernel process table" },
        { F2,   memmap_dmp, "Process memory maps" },
        { F3,   image_dmp, "System image" },
        { F4,   privileges_dmp, "Process privileges" },
        { F5,   monparams_dmp, "Boot monitor parameters" },
        { F6,   irqtab_dmp, "IRQ hooks and policies" },
        { F7,   kmessages_dmp, "Kernel messages" },
        { F9,   sched_dmp, "Scheduling queues" },
        { F10,  kenv_dmp, "Kernel parameters" },
        { F11,  timing_dmp, "Timing details (if enabled)" },
        { SF1,  mproc_dmp, "Process manager process table" },
        { SF2,  sigaction_dmp, "Signals" },
        { SF3,  fproc_dmp, "Filesystem process table" },
        { SF4,  dtab_dmp, "Device/Driver mapping" },
        { SF5,  mapping_dmp, "Print key mappings" },
        { SF6,  rproc_dmp, "Reincarnation server process table" },
        { SF7,  holes_dmp, "Memory free list" },
        { SF8,  data_store_dmp, "Data store contents" },
    { SF9,  rt_sched_dmp, "RT Scheduler dump" },
};

/*===========================================================================*
 *                              handle_fkey                                  *
 *===========================================================================*/
#define pressed(k) ((F1<=(k)&&(k)<=F12 && bit_isset(m->FKEY_FKEYS,((k)-F1+1)))\
        || (SF1<=(k) && (k)<=SF12 && bit_isset(m->FKEY_SFKEYS, ((k)-SF1+1))))
PUBLIC int do_fkey_pressed(m)
message *m;                                     /* notification message */
{
  int s, h;

  /* The notification message does not convey any information, other
   * than that some function keys have been pressed. Ask TTY for details.
   */
  m->m_type = FKEY_CONTROL;
  m->FKEY_REQUEST = FKEY_EVENTS;
  if (OK != (s=sendrec(TTY_PROC_NR, m)))
      report("IS", "warning, sendrec to TTY failed", s);
  /* Now check which keys were pressed: F1-F12, SF1-SF12. */
  for(h=0; h < NHOOKS; h++)
      if(pressed(hooks[h].key))
          hooks[h].function();

  /* Don't send a reply message. */
  return(EDONTREPLY);
}

/*===========================================================================*
 *                              key_name                                     *
 *===========================================================================*/
PRIVATE char *key_name(int key)
{
        static char name[15];

        if(key >= F1 && key <= F12)
                sprintf(name, " F%d", key - F1 + 1);
        else if(key >= SF1 && key <= SF12)
                sprintf(name, "Shift+F%d", key - SF1 + 1);
        else
                sprintf(name, "?");
        return name;
}


/*===========================================================================*
 *                              mapping_dmp                                  *
 *===========================================================================*/
PUBLIC void mapping_dmp(void)
{
  int h;

  printf("Function key mappings for debug dumps in IS server.\n");
  printf("        Key   Description\n");
  printf("-------------------------------------");
  printf("------------------------------------\n");

  for(h=0; h < NHOOKS; h++)
      printf(" %10s.  %s\n", key_name(hooks[h].key), hooks[h].name);
  printf("\n");
}
```

Listing 47: /usr/src/servers/is/dmp_kernel.c

```
/* Debugging dump procedures for the kernel. */

#include "inc.h"
#include <timers.h>
#include <ibm/interrupt.h>
#include <minix/endpoint.h>
#include "../../kernel/const.h"
#include "../../kernel/config.h"
#include "../../kernel/debug.h"
#include "../../kernel/type.h"
#include "../../kernel/proc.h"
#include "../../kernel/ipc.h"

#define click_to_round_k(n) \
        ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))

/* Declare some local dump procedures. */
FORWARD _PROTOTYPE( char *proc_name, (int proc_nr)              );
FORWARD _PROTOTYPE( char *s_traps_str, (int flags)              );
FORWARD _PROTOTYPE( char *s_flags_str, (int flags)              );
FORWARD _PROTOTYPE( char *p_rts_flags_str, (int flags)          );

/* Some global data that is shared among several dumping procedures.
 * Note that the process table copy has the same name as in the kernel
 * so that most macros and definitions from proc.h also apply here.
 */
PUBLIC struct proc proc[NR_TASKS + NR_PROCS];
PUBLIC struct priv priv[NR_SYS_PROCS];
PUBLIC struct boot_image image[NR_BOOT_PROCS];

/*===========================================================================*
 *                              timing_dmp                                   *
 *===========================================================================*/
PUBLIC void timing_dmp()
{
#if ! DEBUG_TIME_LOCKS
  printf("Enable the DEBUG_TIME_LOCKS definition in src/kernel/config.h\n");
#else
  static struct lock_timingdata timingdata[TIMING_CATEGORIES];
  int r, c, f, skipped = 0, printed = 0, maxlines = 23, x = 0;
  static int offsetlines = 0;

  if ((r = sys_getlocktimings(&timingdata[0])) != OK) {
      report("IS","warning: couldn't get copy of lock timings", r);
      return;
  }

  for(c = 0; c < TIMING_CATEGORIES; c++) {
        int b;
        if (!timingdata[c].lock_timings_range[0] || !timingdata[c].binsize)
                continue;
        x = printf("%-*s: misses %lu, resets %lu, measurements %lu: ",
        TIMING_NAME, timingdata[c].names,
                timingdata[c].misses,
                timingdata[c].resets,
                timingdata[c].measurements);
        for(b = 0; b < TIMING_POINTS; b++) {
                int w;
                if (!timingdata[c].lock_timings[b])
                        continue;
```

```
                 x += (w = printf(" %5d: %5d", timingdata[c].lock_timings_range[0]
                     +
                         b*timingdata[c].binsize,
                         timingdata[c].lock_timings[b]));
                 if (x + w >= 80) { printf("\n"); x = 0; }
         }
         if (x > 0) printf("\n");
   }
#endif
}

/*===========================================================================*
 *                              kmessages_dmp                                *
 *===========================================================================*/
PUBLIC void kmessages_dmp()
{
  struct kmessages kmess;               /* get copy of kernel messages */
  char print_buf[KMESS_BUF_SIZE+1];     /* this one is used to print */
  int start;                            /* calculate start of messages */
  int r;

  /* Try to get a copy of the kernel messages. */
  if ((r = sys_getkmessages(&kmess)) != OK) {
      report("IS","warning: couldn't get copy of kmessages", r);
      return;
  }

  /* Try to print the kernel messages. First determine start and copy the
   * buffer into a print-buffer. This is done because the messages in the
   * copy may wrap (the kernel buffer is circular).
   */
  start = ((kmess.km_next + KMESS_BUF_SIZE) - kmess.km_size) % KMESS_BUF_SIZE;
  r = 0;
  while (kmess.km_size > 0) {
        print_buf[r] = kmess.km_buf[(start+r) % KMESS_BUF_SIZE];
        r ++;
        kmess.km_size --;
  }
  print_buf[r] = 0;             /* make sure it terminates */
  printf("Dump of all messages generated by the kernel.\n\n");
  printf("%s", print_buf);              /* print the messages */
}

/*===========================================================================*
 *                              monparams_dmp                                *
 *===========================================================================*/
PUBLIC void monparams_dmp()
{
  char val[1024];
  char *e;
  int r;

  /* Try to get a copy of the boot monitor parameters. */
  if ((r = sys_getmonparams(val, sizeof(val))) != OK) {
      report("IS","warning: couldn't get copy of monitor params", r);
      return;
  }

  /* Append new lines to the result. */
  e = val;
  do {
        e += strlen(e);
        *e++ = '\n';
  } while (*e != 0);

  /* Finally, print the result. */
  printf("Dump of kernel environment strings set by boot monitor.\n");
  printf("\n%s\n", val);
}

/*===========================================================================*
 *                              irqtab_dmp                                   *
 *===========================================================================*/
PUBLIC void irqtab_dmp()
{
  int i,r;
  struct irq_hook irq_hooks[NR_IRQ_HOOKS];
  int irq_actids[NR_IRQ_VECTORS];
  struct irq_hook *e;   /* irq tab entry */
  char *irq[] = {
        "clock",        /* 00 */
        "keyboard",     /* 01 */
        "cascade",      /* 02 */
        "rs232",        /* 03 */
        "rs232",        /* 04 */
        "NIC(eth)",     /* 05 */
        "floppy",       /* 06 */
        "printer",      /* 07 */
        "",     /* 08 */
        "",     /* 09 */
        "",     /* 10 */
        "",     /* 11 */
        "",     /* 12 */
        "",     /* 13 */
        "at_wini_0",    /* 14 */
        "at_wini_1",    /* 15 */
  };

  if ((r = sys_getirqhooks(irq_hooks)) != OK) {
      report("IS","warning: couldn't get copy of irq hooks", r);
      return;
  }
  if ((r = sys_getirqactids(irq_actids)) != OK) {
      report("IS","warning: couldn't get copy of irq mask", r);
      return;
  }

#if 0
  printf("irq_actids:");
  for (i= 0; i<NR_IRQ_VECTORS; i++)
        printf(" [%d] = 0x%08x", i, irq_actids[i]);
  printf("\n");
#endif

  printf("IRQ policies dump shows use of kernel's IRQ hooks.\n");
  printf("-h.id- -proc.nr- -IRQ vector (nr.)- -policy- -notify id-\n");
  for (i=0; i<NR_IRQ_HOOKS; i++) {
        e = &irq_hooks[i];
        printf("%3d", i);
        if (e->proc_nr_e==NONE) {
            printf("    <unused>\n");
            continue;
        }
        printf("%10d  ", e->proc_nr_e);
        printf("     %9.9s (%02d) ", irq[e->irq], e->irq);
        printf("  %s", (e->policy & IRQ_REENABLE) ? "reenable" : "    -    ");
        printf("   %d", e->notify_id);
        if (irq_actids[e->irq] & (1 << i))
                printf("masked");
        printf("\n");
  }
  printf("\n");
}

/*===========================================================================*
 *                              image_dmp                                    *
 *===========================================================================*/
PUBLIC void image_dmp()
{
  int m, i,j,r;
  struct boot_image *ip;
  static char ipc_to[BITCHUNK_BITS*2];

  if ((r = sys_getimage(image)) != OK) {
      report("IS","warning: couldn't get copy of image table", r);
      return;
  }
  printf("Image table dump showing all processes included in system image.\n");
  printf("---name-- -nr- -flags- -traps- -sq- ----pc- -stack-
      -ipc_to[0]--------\n");
  for (m=0; m<NR_BOOT_PROCS; m++) {
      ip = &image[m];
        for (i=j=0; i < BITCHUNK_BITS; i++, j++) {
            ipc_to[j] = (ip->ipc_to & (1<<i)) ? '1' : '0';
            if (i % 8 == 7) ipc_to[++j] = ' ';
        }
        ipc_to[j] = '\0';
      printf("%8s %4d   %s   %s  %3d %7lu %7lu   %s\n",
          ip->proc_name, ip->proc_nr,
               s_flags_str(ip->flags), s_traps_str(ip->trap_mask),
        ip->priority, (long)ip->initial_pc, ip->stksize, ipc_to);
  }
  printf("\n");
}

/*===========================================================================*
 *                              sched_dmp                                    *
 *===========================================================================*/
PUBLIC void sched_dmp()
{
  struct proc *rdy_head[NR_SCHED_QUEUES];
  struct kinfo kinfo;
  register struct proc *rp;
  vir_bytes ptr_diff;
  int r;

  /* First obtain a scheduling information. */
  if ((r = sys_getschedinfo(proc, rdy_head)) != OK) {
      report("IS","warning: couldn't get copy of process table", r);
      return;
  }
  /* Then obtain kernel addresses to correct pointer information. */
  if ((r = sys_getkinfo(&kinfo)) != OK) {
      report("IS","warning: couldn't get kernel addresses", r);
      return;
  }
  /* Update all pointers. Nasty pointer algorithmic ... */
  ptr_diff = (vir_bytes) proc - (vir_bytes) kinfo.proc_addr;
  for (r=0;r<NR_SCHED_QUEUES; r++)
      if (rdy_head[r] != NIL_PROC)
          rdy_head[r] =
              (struct proc *)((vir_bytes) rdy_head[r] + ptr_diff);
  for (rp=BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
      if (rp->p_nextready != NIL_PROC)
          rp->p_nextready =
               (struct proc *)((vir_bytes) rp->p_nextready + ptr_diff);

  /* Now show scheduling queues. */
  printf("Dumping scheduling queues.\n");

  for (r=0;r<NR_SCHED_QUEUES; r++) {
      rp = rdy_head[r];
      if (!rp) continue;
      printf("%2d: ", r);
      while (rp != NIL_PROC) {
          printf("%3d ", rp->p_nr);
          rp = rp->p_nextready;
      }
      printf("\n");
  }
  printf("\n");
}

/*===========================================================================*
 *                              kenv_dmp                                     *
 *===========================================================================*/
PUBLIC void kenv_dmp()
{
    struct kinfo kinfo;
    struct machine machine;
    int r;
    if ((r = sys_getkinfo(&kinfo)) != OK) {
        report("IS","warning: couldn't get copy of kernel info struct", r);
        return;
    }
    if ((r = sys_getmachine(&machine)) != OK) {
        report("IS","warning: couldn't get copy of kernel machine struct", r);
        return;
    }

    printf("Dump of kinfo and machine structures.\n\n");
    printf("Machine structure:\n");
    printf("- pc_at:      %3d\n", machine.pc_at);
    printf("- ps_mca:     %3d\n", machine.ps_mca);
    printf("- processor:  %3d\n", machine.processor);
    printf("- protected:  %3d\n", machine.prot);
    printf("- vdu_ega:    %3d\n", machine.vdu_ega);
    printf("- vdu_vga:    %3d\n\n", machine.vdu_vga);
    printf("Kernel info structure:\n");
    printf("- code_base:  %5u\n", kinfo.code_base);
    printf("- code_size:  %5u\n", kinfo.code_size);
    printf("- data_base:  %5u\n", kinfo.data_base);
    printf("- data_size:  %5u\n", kinfo.data_size);
    printf("- proc_addr:  %5u\n", kinfo.proc_addr);
    printf("- kmem_base:  %5u\n", kinfo.kmem_base);
    printf("- kmem_size:  %5u\n", kinfo.kmem_size);
    printf("- bootdev_base:  %5u\n", kinfo.bootdev_base);
    printf("- bootdev_size:  %5u\n", kinfo.bootdev_size);
    printf("- ramdev_base:   %5u\n", kinfo.ramdev_base);
    printf("- ramdev_size:   %5u\n", kinfo.ramdev_size);
    printf("- params_base:   %5u\n", kinfo.params_base);
    printf("- params_size:   %5u\n", kinfo.params_size);
    printf("- nr_procs:     %3u\n", kinfo.nr_procs);
    printf("- nr_tasks:     %3u\n", kinfo.nr_tasks);
    printf("- release:      %.6s\n", kinfo.release);
    printf("- version:      %.6s\n", kinfo.version);
#if DEBUG_LOCK_CHECK
    printf("- relocking:    %d\n", kinfo.relocking);
#endif
    printf("\n");
}

PRIVATE char *s_flags_str(int flags)
{
        static char str[10];
        str[0] = (flags & PREEMPTIBLE) ? 'P' : '-';
        str[1] = '-';
        str[2] = (flags & BILLABLE)    ? 'B' : '-';
        str[3] = (flags & SYS_PROC)    ? 'S' : '-';
        str[4] = '-';
        str[5] = '\0';

        return str;
}

PRIVATE char *s_traps_str(int flags)
{
        static char str[10];
        str[0] = (flags & (1 << ECHO)) ? 'E' : '-';
        str[1] = (flags & (1 << SEND))  ? 'S' : '-';
        str[2] = (flags & (1 << RECEIVE))  ? 'R' : '-';
        str[3] = (flags & (1 << SENDREC))  ? 'B' : '-';
        str[4] = (flags & (1 << NOTIFY)) ? 'N' : '-';
        str[5] = '\0';

        return str;
}

/*===========================================================================*
 *                              privileges_dmp                               *
 *===========================================================================*/
PUBLIC void privileges_dmp()
{
  register struct proc *rp;
  static struct proc *oldrp = BEG_PROC_ADDR;
  register struct priv *sp;
  static char ipc_to[NR_SYS_PROCS + 1 + NR_SYS_PROCS/8];
  int r, i,j, n = 0;

  /* First obtain a fresh copy of the current process and system table. */
  if ((r = sys_getprivtab(priv)) != OK) {
      report("IS","warning: couldn't get copy of system privileges table", r);
      return;
  }
  if ((r = sys_getproctab(proc)) != OK) {
      report("IS","warning: couldn't get copy of process table", r);
      return;
  }
  printf("\n--nr-id-name---- -flags- -traps- -ipc_to mask------------------------
      \n");

  for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (isemptyp(rp)) continue;
        if (++n > 23) break;
        if (proc_nr(rp) == IDLE)        printf("(%2d) ", proc_nr(rp));
        else if (proc_nr(rp) < 0)       printf("[%2d] ", proc_nr(rp));
        else                            printf(" %2d  ", proc_nr(rp));
        r = -1;
        for (sp = &priv[0]; sp < &priv[NR_SYS_PROCS]; sp++)
            if (sp->s_proc_nr == rp->p_nr) { r ++; break; }
        if (r == -1 && ! (rp->p_rts_flags & SLOT_FREE)) {
            sp = &priv[USER_PRIV_ID];
        }
        printf("(%02u) %-7.7s %s   %s  ",
               sp->s_id, rp->p_name,
               s_flags_str(sp->s_flags), s_traps_str(sp->s_trap_mask)
        );
        for (i=j=0; i < NR_SYS_PROCS; i++, j++) {
            ipc_to[j] = get_sys_bit(sp->s_ipc_to, i) ? '1' : '0';
            if (i % 8 == 7) ipc_to[++j] = ' ';
        }
        ipc_to[j] = '\0';

        printf(" %s \n", ipc_to);
  }
  if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
  oldrp = rp;

}

/*===========================================================================*
 *                              sendmask_dmp                                 *
 *===========================================================================*/
PUBLIC void sendmask_dmp()
{
  register struct proc *rp;
  static struct proc *oldrp = BEG_PROC_ADDR;
  int r, i,j, n = 0;

  /* First obtain a fresh copy of the current process table. */
  if ((r = sys_getproctab(proc)) != OK) {
      report("IS","warning: couldn't get copy of process table", r);
      return;
  }

  printf("\n\n");
  printf("Sendmask dump for process table. User processes (*) don't have [].");
  printf("\n");
  printf("The rows of bits indicate to which processes each process may send.");
  printf("\n\n");

#if DEAD_CODE
  printf("              ");
  for (j=proc_nr(BEG_PROC_ADDR); j< INIT_PROC_NR+1; j++) {
     printf("%3d", j);
  }
  printf("  *\n");

  for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (isemptyp(rp)) continue;
        if (++n > 20) break;

        printf("%8s ", rp->p_name);
        if (proc_nr(rp) == IDLE)        printf("(%2d) ", proc_nr(rp));
        else if (proc_nr(rp) < 0)       printf("[%2d] ", proc_nr(rp));
        else                            printf(" %2d  ", proc_nr(rp));

        for (j=proc_nr(BEG_PROC_ADDR); j<INIT_PROC_NR+2; j++) {
            if (isallowed(rp->p_sendmask, j))   printf(" 1 ");
            else                                printf(" 0 ");
        }
        printf("\n");
  }
  if (rp == END_PROC_ADDR) { printf("\n"); rp = BEG_PROC_ADDR; }
  else printf("--more--\r");
  oldrp = rp;
#endif
}

PRIVATE char *p_rts_flags_str(int flags)
{
        static char str[10];
        str[0] = (flags & NO_MAP) ? 'M' : '-';
        str[1] = (flags & SENDING)  ? 'S' : '-';
        str[2] = (flags & RECEIVING)    ? 'R' : '-';
        str[3] = (flags & SIGNALED)    ? 'I' : '-';
        str[4] = (flags & SIG_PENDING)    ? 'P' : '-';
        str[5] = (flags & P_STOP)    ? 'T' : '-';
        str[6] = '\0';

        return str;
}

/*===========================================================================*
 *                              proctab_dmp                                  *
 *===========================================================================*/
#if (CHIP == INTEL)
PUBLIC void proctab_dmp()
{
/* Proc table dump */

  register struct proc *rp;
  static struct proc *oldrp = BEG_PROC_ADDR;
  int r, n = 0;
  phys_clicks text, data, size;

  /* First obtain a fresh copy of the current process table. */
  if ((r = sys_getproctab(proc)) != OK) {
      report("IS","warning: couldn't get copy of process table", r);
      return;
  }

  printf("\n-nr-----gen---endpoint--name--- -prior-quant- -user---sys----size-rts
      flags-\n");

  for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (isemptyp(rp)) continue;
        if (++n > 23) break;
        text = rp->p_memmap[T].mem_phys;
        data = rp->p_memmap[D].mem_phys;
        size = rp->p_memmap[T].mem_len
                + ((rp->p_memmap[S].mem_phys + rp->p_memmap[S].mem_len) - data);
        if (proc_nr(rp) == IDLE)        printf("(%2d) ", proc_nr(rp));
        else if (proc_nr(rp) < 0)       printf("[%2d] ", proc_nr(rp));
        else                            printf(" %2d  ", proc_nr(rp));
        printf(" %5d %10d ", _ENDPOINT_G(rp->p_endpoint), rp->p_endpoint);
        printf(" %-8.8s %02u/%02u %02d/%02u %6lu%6lu %6uK %s",
               rp->p_name,
               rp->p_priority, rp->p_max_priority,
               rp->p_ticks_left, rp->p_quantum_size,
               rp->p_user_time, rp->p_sys_time,
               click_to_round_k(size),
               p_rts_flags_str(rp->p_rts_flags));
        if (rp->p_rts_flags & (SENDING|RECEIVING)) {
                printf(" %-7.7s", proc_name(_ENDPOINT_P(rp->p_getfrom_e)));
        }
        printf("\n");
  }
  if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
  oldrp = rp;
}
#endif                          /* (CHIP == INTEL) */

/*===========================================================================*
 *                              memmap_dmp                                   *
 *===========================================================================*/
PUBLIC void memmap_dmp()
{
  register struct proc *rp;
  static struct proc *oldrp = proc;
  int r, n = 0;
  phys_clicks size;

  /* First obtain a fresh copy of the current process table. */
  if ((r = sys_getproctab(proc)) != OK) {
      report("IS","warning: couldn't get copy of process table", r);
      return;
  }

  printf("\n-nr/name--- --pc-- --sp-- -----text----- -----data-----
      ----stack----- --size-\n");
  for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (isemptyp(rp)) continue;
        if (++n > 23) break;
        size = rp->p_memmap[T].mem_len
                + ((rp->p_memmap[S].mem_phys + rp->p_memmap[S].mem_len)
                                                - rp->p_memmap[D].mem_phys);
        printf("%3d %-7.7s%7lx%7lx %4x %4x %4x %4x %4x %4x %4x %4x %4x %5uK\n",
               proc_nr(rp),
               rp->p_name,
               (unsigned long) rp->p_reg.pc,
               (unsigned long) rp->p_reg.sp,
               rp->p_memmap[T].mem_vir, rp->p_memmap[T].mem_phys,
                   rp->p_memmap[T].mem_len,
               rp->p_memmap[D].mem_vir, rp->p_memmap[D].mem_phys,
                   rp->p_memmap[D].mem_len,
               rp->p_memmap[S].mem_vir, rp->p_memmap[S].mem_phys,
                   rp->p_memmap[S].mem_len,
               click_to_round_k(size));
  }
  if (rp == END_PROC_ADDR) rp = proc;
  else printf("--more--\r");
  oldrp = rp;
}
```

```
/*===========================================================================*
 *                              proc_name                                    *
 *===========================================================================*/
PRIVATE char *proc_name(proc_nr)
int proc_nr;
{
  if (proc_nr == ANY) return "ANY";
  return cproc_addr(proc_nr)->p_name;
}

/*===========================================================================*
 *              rt_sched_dmp                                                 *
 *===========================================================================*/
PUBLIC void rt_sched_dmp(void)
{
  /* Binded to Shift+F9. Dumps Real-time scheduler info.
   * Invokes system library function to do the kernel call.
   * This kernel call will dump the info on the screen.
   */
  sys_rt_show_data();
}
```

Listing 48: /usr/src/servers/is/main.c

```
/* System Information Service.
 * This service handles the various debugging dumps, such as the process
 * table, so that these no longer directly touch kernel memory. Instead, the
 * system task is asked to copy some table in local memory.
 *
 * Created:
 *   Apr 29, 2004       by Jorrit N. Herder
 */

#include "inc.h"

/* Set debugging level to 0, 1, or 2 to see no, some, all debug output. */
#define DEBUG_LEVEL     1
#define DPRINTF         if (DEBUG_LEVEL > 0) printf

/* Allocate space for the global variables. */
message m_in;           /* the input message itself */
message m_out;          /* the output message used for reply */
int who_e;              /* caller's proc number */
int callnr;             /* system call number */

extern int errno;       /* error number set by system library */

/* Declare some local functions. */
FORWARD _PROTOTYPE(void init_server, (int argc, char **argv)            );
FORWARD _PROTOTYPE(void sig_handler, (void)                             );
FORWARD _PROTOTYPE(void exit_server, (void)                             );
FORWARD _PROTOTYPE(void get_work, (void)                                );
FORWARD _PROTOTYPE(void reply, (int whom, int result)                   );

/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC int main(int argc, char **argv)
{
/* This is the main routine of this service. The main loop consists of
 * three major activities: getting new work, processing the work, and
 * sending the reply. The loop never terminates, unless a panic occurs.
 */
  int result;
  sigset_t sigset;

  /* Initialize the server, then go to work. */
  init_server(argc, argv);

  /* Main loop - get work and do it, forever. */
  while (TRUE) {

      /* Wait for incoming message, sets 'callnr' and 'who'. */
      get_work();

      switch (callnr) {
      case SYS_SIG:
          printf("got SYS_SIG message\n");
          sigset = m_in.NOTIFY_ARG;
          for ( result=0; result< _NSIG; result++) {
              if (sigismember(&sigset, result))
                  printf("signal %d found\n", result);
          }
          continue;
      case PROC_EVENT:
          sig_handler();
          continue;
      case FKEY_PRESSED:
          result = do_fkey_pressed(&m_in);
          break;
      case DEV_PING:
          notify(m_in.m_source);
          continue;
      default:
          report("IS","warning, got illegal request from:", m_in.m_source);
          result = EINVAL;
      }

      /* Finally send reply message, unless disabled. */
      if (result != EDONTREPLY) {
          reply(who_e, result);
      }
  }
  return(OK);                           /* shouldn't come here */
}

/*===========================================================================*
 *                               init_server                                 *
 *===========================================================================*/
PRIVATE void init_server(int argc, char **argv)
{
/* Initialize the information service. */
  int fkeys, sfkeys;
  int i, s;
  struct sigaction sigact;

  /* Install signal handler. Ask PM to transform signal into message. */
  sigact.sa_handler = SIG_MESS;
  sigact.sa_mask = ~0;                  /* block all other signals */
  sigact.sa_flags = 0;                  /* default behaviour */
  if (sigaction(SIGTERM, &sigact, NULL) < 0)
      report("IS","warning, sigaction() failed", errno);

  /* Set key mappings. IS takes all of F1-F12 and Shift+F1-F9. */
  fkeys = sfkeys = 0;
  for (i=1; i<=12; i++) bit_set(fkeys, i);
  for (i=1; i<= 9; i++) bit_set(sfkeys, i);
  if ((s=fkey_map(&fkeys, &sfkeys)) != OK)
      report("IS", "warning, fkey_map failed:", s);
}

/*===========================================================================*
 *                               sig_handler                                 *
 *===========================================================================*/
PRIVATE void sig_handler()
{
  sigset_t sigset;
  int sig;

  /* Try to obtain signal set from PM. */
  if (getsigset(&sigset) != 0) return;

  /* Check for known signals. */
  if (sigismember(&sigset, SIGTERM)) {
      exit_server();
  }
```

```
}

/*===========================================================================*
 *                              exit_server                                  *
 *===========================================================================*/
PRIVATE void exit_server()
{
/* Shut down the information service. */
  int fkeys, sfkeys;
  int i,s;

  /* Release the function key mappings requested in init_server().
   * IS took all of F1-F12 and Shift+F1-F6.
   */
  fkeys = sfkeys = 0;
  for (i=1; i<=12; i++) bit_set(fkeys, i);
  for (i=1; i<= 7; i++) bit_set(sfkeys, i);
  if ((s=fkey_unmap(&fkeys, &sfkeys)) != OK)
      report("IS", "warning, unfkey_map failed:", s);

  /* Done. Now exit. */
  exit(0);
}

/*===========================================================================*
 *                              get_work                                     *
 *===========================================================================*/
PRIVATE void get_work()
{
    int status = 0;
    status = receive(ANY, &m_in);   /* this blocks until message arrives */
    if (OK != status)
        panic("IS","failed to receive message!", status);
    who_e = m_in.m_source;        /* message arrived! set sender */
    callnr = m_in.m_type;       /* set function call number */
}

/*===========================================================================*
 *                              reply                                        *
 *===========================================================================*/
PRIVATE void reply(who, result)
int who;                                /* destination */
int result;                             /* report result to replyee */
{
    int send_status;
    m_out.m_type = result;              /* build reply message */
    send_status = send(who, &m_out);    /* send the message */
    if (OK != send_status)
        panic("IS", "unable to send reply!", send_status);
}
```

Listing 49: /usr/src/servers/is/proto.h

```
/* Function prototypes. */

/* main.c */
_PROTOTYPE( int  main, (int argc, char **argv)                               );

/* dmp.c */
_PROTOTYPE( int do_fkey_pressed, (message *m)                                );
_PROTOTYPE( void mapping_dmp, (void)                                         );

/* dmp_kernel.c */
_PROTOTYPE( void proctab_dmp, (void)                                         );
_PROTOTYPE( void memmap_dmp, (void)                                          );
_PROTOTYPE( void privileges_dmp, (void)                                      );
_PROTOTYPE( void sendmask_dmp, (void)                                        );
_PROTOTYPE( void image_dmp, (void)                                           );
_PROTOTYPE( void irqtab_dmp, (void)                                          );
_PROTOTYPE( void kmessages_dmp, (void)                                       );
_PROTOTYPE( void sched_dmp, (void)                                           );
_PROTOTYPE( void monparams_dmp, (void)                                       );
_PROTOTYPE( void kenv_dmp, (void)                                            );
_PROTOTYPE( void timing_dmp, (void)                                          );
_PROTOTYPE( void rt_sched_dmp, (void)                 );

/* dmp_pm.c */
_PROTOTYPE( void mproc_dmp, (void)                                           );
_PROTOTYPE( void sigaction_dmp, (void)                                       );
_PROTOTYPE( void holes_dmp, (void)                                           );

/* dmp_fs.c */
_PROTOTYPE( void dtab_dmp, (void)                                            );
_PROTOTYPE( void fproc_dmp, (void)                                           );

/* dmp_rs.c */
_PROTOTYPE( void rproc_dmp, (void)                                           );

/* dmp_ds.c */
_PROTOTYPE( void data_store_dmp, (void)                                      );
```

Listing 50: /usr/src/servers/ss/proto.h

```
/* Function prototypes for semaphore server */

/* ss.c */
_PROTOTYPE(void init, (void) );
_PROTOTYPE(int sem_create, (int type, int value, int size));
_PROTOTYPE(int sem_take, (int sem_handler, int block));
_PROTOTYPE(int sem_give, (int sem_handler));
_PROTOTYPE(int sem_flush, (int sem_handler));
_PROTOTYPE(int sem_delete, (int sem_handler));
_PROTOTYPE(int procexit, (int endpoint));
_PROTOTYPE(int sem_value,(int sem_handler));
```

Listing 51: /usr/src/servers/ss/ss.h

```
/* This is the master header for SS.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1     /* tell headers to include POSIX stuff */
#define _MINIX             1     /* tell headers to include MINIX stuff */
#define _SYSTEM            1     /* tell headers that this is the system */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>        /* MUST be first */
#include <ansi.h>                /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>

#include <fcntl.h>
#include <unistd.h>
#include <minix/syslib.h>
#include <minix/sysutil.h>

#include <limits.h>
#include <errno.h>

#include <minix/sem.h>
#include "proto.h"

#ifndef OK
#define OK 0
#endif

/* number of semaphore supported */
#define NR_SEMS 128

/* semaphore structure */
struct sem {
        int sem_type;
        int value;
        int size;
        int used;
        int owner;
        int user;
        struct sem_proc * listp;

};

/* used to save processes that are blocked
 * on a semaphore
 */
struct sem_proc {
        int endpoint;
        int prio;
        struct sem_proc * next;
};

/* semaphore table */
struct sem semtable[NR_SEMS];

/* keep track of sems in use, expirimental */
int used;
```

Listing 52: /usr/src/servers/ss/ss.c

```
/* This file contains the implementation of the Semaphore Server.
 * It contains the following functions:
 *   main: The main program
 *   init: initialize the server
 *   sem_create: create a new semaphore
 *   sem_take: take a semaphore
 *   sem_give: give a semaphore free
 *   sem_delete: delete a semaphore
 *   sem_flush: unblock all blocked processes
 *   sem_value: get value of semaphore
 *   procexit: used by pm to tell the server that a process has exited
 *
 * Changes:
 *   June 14, 2009  initial release (Bianco Zandbergen)
 */
#include <stdio.h>
#include <lib.h>
#include <unistd.h>
#include <stdlib.h>
#include <timers.h>
#include <minix/syslib.h>
#include "../../kernel/proc.h"
#include "ss.h"

/* user processes use virtual sem handlers
 * SS internal uses real sem handlers. We do this because
 * real sem handlers are the same as the array indices in semtable.
 * These start at 0 and an uninitialized sem handler also has the value of
 * 0. To prevend the use of uninitialized sem handlers we use virtual sem handlers
 * outside the SS.
 * VIR_SEM_HANDLER converts a real handler to a virtual handler.
 * REAL_SEM_HANDLER converts a virtual handler to a real handler.
 */
#define VIR_SEM_HANDLER(real) (real+1)
#define REAL_SEM_HANDLER(vir) (vir-1)

/* checks is a sem handler is valid, real sem handler is used as arg */
#define VALID_SEM(X) ((X < NR_SEMS) && (semtable[X].used == 1))

message m;

int main(void) {

  int caller;

  /* reset values */
  init();

  while(1) {
      receive(ANY, &m); /* get work */
      caller = m.m_source;

      /* Invoke the right function depending on the message type.
       * Depending on the function and return value it will send a reply
       * back to the caller process.
       */
      switch(m.m_type) {
          case SEM_CREATE:
              m.m_type = sem_create(m.SEM_F_TYPE, m.SEM_F_VALUE,  m.SEM_F_SIZE);
              if(send(caller, &m) != OK) printf("ss: Sending back failed!\n");
```

```
                break;
            case SEM_TAKE:
                m.m_type = sem_take(REAL_SEM_HANDLER(m.SEM_F_HANDLER),
                    m.SEM_F_BLOCK);
                if (m.m_type == OK || m.m_type == SEM_INVALID_HANDLER || m.m_type
                    == SEM_INVALID_BLOCK ||
                        (m.SEM_F_BLOCK == NO_WAIT && m.m_type == SEM_IN_USE)) {
                    /* We only send back if the sem was available or the sem
                        handler was invalid
                     * or the block arg was invalid or the sem was not
                     * available AND the process does not want to wait.
                     */
                    if(send(caller, &m) != OK) printf("ss: Sending back failed!\n");
                }
                break;
            case SEM_GIVE:
                m.m_type = sem_give(REAL_SEM_HANDLER(m.SEM_F_HANDLER));
                if(send(caller, &m) != OK) printf("ss: Sending back failed! \n");
                break;
            case SEM_FLUSH:
                m.m_type = sem_flush(REAL_SEM_HANDLER(m.SEM_F_HANDLER));
                if(send(caller, &m) != OK) printf("ss: Sending back failed! \n");
                break;
            case SEM_DELETE:
                m.m_type = sem_delete(REAL_SEM_HANDLER(m.SEM_F_HANDLER));
                if(send(caller, &m) != OK) printf("ss: Sending back failed! \n");
                break;
            case SEM_PROCEXIT:
                #ifdef SEM_DEBUG
                printf("ss: process exited! endpt: %d\n",m.SEM_F_ENDPT);
                #endif
                procexit(m.SEM_F_ENDPT);
                /* we don't have to send back! */
                break;
            case SEM_VALUE:
                m.m_type = sem_value(REAL_SEM_HANDLER(m.SEM_F_HANDLER));
                if(send(caller, &m) != OK) printf("ss: Sending back failed! \n");
                break;
            default:
                m.m_type = (EINVAL);
                if (send(caller, &m) != OK) printf("ss: Sending back failed! \n");
        }
    }
}

/*****************************************
 *       init                           *
 *****************************************/
void init(void) {

    int i;
    used = 0;

    for(i=0; i<NR_SEMS; i++) {
        semtable[i].used = 0;
        semtable[i].listp = NULL;
    }
}

/*****************************************
 *       sem_create                     *
 *****************************************/
```

```c
int sem_create(int type, int value, int size)
{
  /* Create a new semaphore */

  int i;

  #ifdef SEM_DEBUG
  printf("sem_create type: %d, value: %d, size: %d\n",type,value,size);
  #endif

  /* check if type is valid
   * semtypes: 1=mutex,2=binary,3=counting
   */
  if (type > 3 || type < 1) return SEM_INVALID_TYPE;

  /* All semaphores in use! */
  if (used >= NR_SEMS) return SEM_NO_AVAILABLE;

  /* type specific checks */
  if (type == SEM_BINARY) {

      /* value must be 0 or 1 */
      if (value != 0 && value != 1) {
          return SEM_INVALID_VALUE;
      }
  } else if (type == SEM_COUNTING) {

      /* value cannot be larger than size, value cannot be negative */
      if (value > size || value < 0) {
          return SEM_INVALID_VALUE;
      } else if (size < 1) {
          /* size must be larger than 1 */
          return SEM_INVALID_SIZE;
      }
  }

  i = 0;

  /* find unused sem */
  while(semtable[i].used != 0 && i < NR_SEMS) i++;
  if (i == NR_SEMS) return SEM_NO_AVAILABLE;

  semtable[i].used = 1;
  semtable[i].sem_type = type;
  semtable[i].listp = NULL; /* no blocked processes in queue */
  semtable[i].owner = m.m_source; /* endpoint nr of the caller process */
  used++;

  /* setting value and size */
  switch (type) {
      case SEM_MUTEX:
          semtable[i].value = 1;
          break;
      case SEM_BINARY:
          semtable[i].value = value;
          break;
      case SEM_COUNTING:
          semtable[i].value = value;
          semtable[i].size = size;
          break;
  }
  #if 0
  printf("ss: found unused sem: real %d vir: %d\n", i, VIR_SEM_HANDLER(i));
  #endif

  /* we return a virtual sem handler to the caller process */
  m.SEM_F_HANDLER = VIR_SEM_HANDLER(i);

  return OK;
}

/****************************************
 *      sem_take                        *
 ****************************************/
int sem_take(int sem_handler, int block)
{
  /* Take a semaphore */

  struct sem_proc * sproc;
  struct sem_proc ** xpp;
  struct proc caller_proc;
  int result;

  /* check if sem handler is valid */
  if (!VALID_SEM(sem_handler)) return SEM_INVALID_HANDLER;

  /* get process structure from caller */
  if (result = sys_getproc(&caller_proc, m.m_source) != OK) {
      printf("ss: sys_getproc failed result: %d\n", result);
  }

  switch (semtable[sem_handler].sem_type) {
      case SEM_MUTEX:
      case SEM_BINARY:
          if (semtable[sem_handler].value == 1) {
              /* sem is available */
              semtable[sem_handler].value = 0;
              semtable[sem_handler].user = m.m_source;
              return OK;
          } else {
              /* sem is not available */

              if (block == NO_WAIT) {
                  /* caller doesn't want to wait */
                  return SEM_IN_USE;

              } else if (block == WAIT_FOREVER) {

                  /* caller does want to wait.
                   * Add caller to the list of waiting processes on the sem.
                   */

                  /* create new sem_proc struct */
                  sproc = malloc(sizeof(struct sem_proc));

                  /* set the priority of the sem_proc struct to the
                   * minix scheduling priority.
                   */
                  sproc->prio = caller_proc.p_priority;

                  /* set the caller endpoint */
                  sproc->endpoint = m.m_source;
```

```
                /* find the right place in the list.
                 * The list is ordered on minix scheduling priority.
                 */
                xpp = &semtable[sem_handler].listp;
                while (*xpp != NULL && (*xpp)->prio <= sproc->prio) {
                    xpp = &(*xpp)->next;
                }

                /* add new sem_proc to the list */
                sproc->next = *xpp;
                *xpp = sproc;
            } else {
                /* invalid block argument specified */
                return SEM_INVALID_BLOCK;
            }
        }
        break;
    case SEM_COUNTING:
        if(semtable[sem_handler].value > 0) {
            /* sem is available */
            semtable[sem_handler].value--;
            return OK;
        } else if (semtable[sem_handler].value == 0) {
            /* sem is not available */

            if (block == NO_WAIT) {
                /* caller does not want to wait */
                return SEM_IN_USE;

            } else if (block = WAIT_FOREVER) {

                /* caller does want to wait.
                 * Add caller to the list of waiting processes on the sem.
                 */

                /* create new sem proc_struct */
                sproc = malloc(sizeof(struct sem_proc));

                /* set the priority of the sem_proc struct to the
                 * minix scheduling priority.
                 */
                sproc->prio = caller_proc.p_priority;

                /* set the caller endpoint */
                sproc->endpoint = m.m_source;

                /* find the right place in the list.
                 * The list is ordered on minix scheduling priority */
                xpp = &semtable[sem_handler].listp;
                while (*xpp != NULL && (*xpp)->prio <= sproc->prio) {
                    xpp = &(*xpp)->next;
                }

                /* add new sem_proc to the list */
                sproc->next = *xpp;
                *xpp = sproc;
            } else {
                /* invalid block argument specified */
                return SEM_INVALID_BLOCK;
            }
        }
        break;
```

```
  }

  /* will be reached if sem is not available and caller should block */
  return -1;
}

/****************************************
 *      sem_give                        *
 ****************************************/
int sem_give(int sem_handler)
{
  /* Give a semaphore free */

  struct sem_proc * sproc;
  message m_out;
  int s;

  /* check if handler is valid */
  if (! VALID_SEM(sem_handler)) return SEM_INVALID_HANDLER;

  switch(semtable[sem_handler].sem_type) {
      case SEM_MUTEX:

          /* a semaphore must be taken in order to give is free.
           * We do this check again at the SEM_BINARY case.
           * It is done here because this check must be done BEFORE
           * the test below. (A sem that is not taken does has have a valid user)
           */
          if (semtable[sem_handler].value > 0) {
              return SEM_ALREADY_FREE;
          }

          /* mutex has one extra condition compared to binary:
           * a give may only be done if the same process
           * took the semaphore.
           */
          if (semtable[sem_handler].user != m.m_source) {
              return SEM_INVALID_MUTEX_GIVE;
          }
          /* do not break! */
      case SEM_BINARY:
          /* a semaphore must be taken in order to give is free */
          if (semtable[sem_handler].value > 0) {
              return SEM_ALREADY_FREE;
          }

          /* give sem free */
          semtable[sem_handler].value = 1;

          /* check for processes in the wait queue blocked
           * on this semaphore. We will have to unblock the next process
           * in the queue. We do this in a while loop because we might need to
           * try multiple times. send() may fail because a process in the queue
           * is not excistant. It may be killed for example. if send() does not
           * fail the code is only executed once.
           */
          while (semtable[sem_handler].listp != NULL) {

              /* we save the head of the block queue to sproc before removing it
                  */
              sproc = semtable[sem_handler].listp;
```

```
            /* remove the head of the queue (sproc) */
            semtable[sem_handler].listp = sproc->next;

            /* try to unblock the process */
            if(s=send(sproc->endpoint, &m) != OK) {

                /* This process is not valid.
                 * Might be killed.
                 * Release memory.
                 */
                free(sproc);
                printf("ss: Sending unblock failed!\n");
            } else {

                /* sem is taken again */
                semtable[sem_handler].value = 0;

                /* save endpoint of process that took the sem */
                semtable[sem_handler].user = sproc->endpoint;

                /* release memory */
                free(sproc);

                /* unblock successful so stop looping */
                break;
            }
        }
        return OK;
        break;
    case SEM_COUNTING:

        /* if value equals size the sem is 'full' and we can't give it free */
        if (semtable[sem_handler].value == semtable[sem_handler].size) {
            return SEM_ALREADY_FREE;
        }

        /* give sem free */
        semtable[sem_handler].value++;

        /* check for processes in the wait queue blocked
         * on this semaphore. We will have to unblock the next process
         * in the queue. We do this in a while loop because we might need to
         * try multiple times. send() may fail because a process in the queue
         * is not excistant. It may be killed for example. if send() does not
         * fail the code is only executed once.
         */
        while (semtable[sem_handler].listp != NULL) {

            /* we save the head of the block queue to sproc before removing it
                */
            sproc = semtable[sem_handler].listp;

            /* remove the head of the block queue */
            semtable[sem_handler].listp = sproc->next;

            /* try to unblock the process */
            if(s=send(sproc->endpoint, &m) != OK) {

                /* This process is not valid.
                 * Might be killed.
                 * Release memory.
                 */
```

```
                    free(sproc);
                    printf("ss: Sending unblock failed!\n");
                } else {

                    /* decrease sem (taking it) */
                    semtable[sem_handler].value--;

                    /* save endpoint of process that took sem */
                    semtable[sem_handler].user = sproc->endpoint;

                    /* release memory */
                    free(sproc);

                    /* unblock successful so stop looping */
                    break;
                }
            }
            return OK;
            break;
        default:
            /* invalid semaphore type */
            return SEM_INVALID_TYPE;
  }

  /* never reached */
  return -1;
}

/****************************************
 *       sem_flush                      *
 ****************************************/
int sem_flush(int handler)
{
  /* unblocks all processes blocked
   * on a binary semaphore.
   * Can be used to synchronize processes.
   */
  struct sem_proc * sproc;
  message m_out;
  m_out.m_type = OK;

  /* check if sem handler is valid */
  if (! VALID_SEM(handler)) return SEM_INVALID_HANDLER;

  /* we only allow flush on binary semaphores */
  if (semtable[handler].sem_type != SEM_BINARY) return SEM_INVALID_TYPE;

  /* unblock all processes on the block queue */
  while (semtable[handler].listp != NULL) {
      sproc = semtable[handler].listp; /* get next process from the queue */

      /* send a message to the process to unblock it */
      if (send(sproc->endpoint, &m_out) != OK) {
          printf("ss: Sending unblock failed!\n");
      }

      semtable[handler].listp = sproc->next; /* remove head from the queue */
      free(sproc); /* release memory */
  }

  return OK;
}
```

```
/*****************************************
 *      sem_delete                       *
 *****************************************/
int sem_delete(int handler)
{
  /* Delete a semaphore. */

  /* check if sem handler is valid */
  if (! VALID_SEM(handler)) return SEM_INVALID_HANDLER;

  switch (semtable[handler].sem_type) {

      case SEM_MUTEX:
      case SEM_BINARY:
          /* A bin/mutex semaphore can only be deleted if it is not taken */
          if (semtable[handler].value == 1) {
              semtable[handler].used = 0; /* release slot */
              used--;
          } else {
              return SEM_IN_USE;
          }
          break;
      case SEM_COUNTING:
          /* A counting semaphore can only be deleted if it is 'full' */
          if (semtable[handler].value == semtable[handler].size) {
              semtable[handler].used = 0; /* release slot */
              used--;
          } else {
              return SEM_IN_USE;
          }
          break;
  }

  return OK;
}

/*****************************************
 *      sem_procexit                     *
 *****************************************/
int procexit(int endpoint)
{
  /* Used by the process manager to inform that a process has exited.
   * Deletes all semaphores with the exited process as owner.
   */
  int no_sems;
  int current_sem;

  current_sem = 0;
  no_sems = 0; /* expirimental */

  /* loop through the semaphore table */
  while (/*no_sems < used &&*/ current_sem < NR_SEMS) {
      /* check if semaphore is in use */
      if (semtable[current_sem].used == 1) {
          no_sems++; /* expirimental, not used */

          /* check if the exited process is the owner.
           * If so release the semaphore slot.
           */
          if (semtable[current_sem].owner == endpoint) {
              semtable[current_sem].used = 0;
              used--;
              printf("ss: deleted sem %d owned by %d\n",current_sem, endpoint);

          }
      }
      current_sem++;
  }

  return OK;
}

/****************************************
 *      sem_value                       *
 ****************************************/
int sem_value(sem_t handler)
{
  /* Returns the value of a semaphore to a user process.
   * May be used for debugging purposes.
   */

  /* check if handler is valid */
  if (! VALID_SEM(handler)) return SEM_INVALID_HANDLER;

  /* save the value to the message the main loop sends back */
  m.SEM_F_VALUE = semtable[handler].value;

  return OK;
}
```

Listing 53: /usr/src/lib/rtminix3/rt_nextperiod.c

```
/* The library function implemented in this file:
 *   rt_nextperiod
 *
 * Function description:
 *   This function gives processes using Earliest
 *   Deadline First scheduling a way to give up
 *   remaining calculation time. (This function may
 *   only be used by processes using EDF scheduling!)
 *
 * The parameters for this library function are:
 *   none
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/rt.h>

int rt_nextperiod(void)
{
  message m; /* message we use for the system call */

  /* do the system call and return the result */
  return (_syscall(MM, RT_NEXTPERIOD, &m));
}
```

Listing 54: /usr/src/lib/rtminix3/rt_set_edf.c

```c
/* The library function implemented in this file:
 *   rt_set_edf
 *
 * Function description:
 *   This function transforms a normal user process into a
 *   real-time process scheduled using Earliest Deadline First.
 *   The operation will only succeed if the RT scheduler is already
 *   set to EDF using the rt_set_sched_edf() library call.
 *
 * The parameters for this library function are:
 *   int period          The period of the process in ticks
 *   int calctime        The reserved calculation time within
 *                       each period in ticks
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/rt.h>

int rt_set_edf(int period, int calctime)
{
  message m; /* message we use for system call */

  m.RT_SCHED = SCHED_EDF; /* set scheduler type */
  m.EDF_PERIOD = period; /* set period in ticks */
  m.EDF_CALCTIME = calctime; /* set calculation time within each period in ticks
      */

  /* do the system call and return the result */
  return (_syscall(MM, RT_SET, &m));
}
```

Listing 55: /usr/src/lib/rtminix3/rt_set_rm.c

```
/* The library function implemented in this file:
 *   rt_set_rm
 *
 * Function description:
 *   This function transforms a normal user process into a
 *   real-time process scheduled using Rate-Monotonic.
 *   The operation will only succeed if the RT scheduler is already
 *   set to RM using the rt_set_sched_rm() library call.
 *
 * The parameters for this library function are:
 *   int period        The fixed priority of the RM process
 *                     compared to other RM processes.
 *                     This can be any positive or negative number.
 *                     If the priority policy is PRIO_UNIQUE,
 *                     this priority may not be in use by an other
 *                     RM process.
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/rt.h>

int rt_set_rm(int prio)
{
  message m; /* message we use for the system call */

  m.RT_SCHED = SCHED_RM; /* set the scheduler type */
  m.RM_PRIO = prio; /* set the Rate-Monotonic priority */

  /* do the system call and return the result */
  return (_syscall(MM, RT_SET, &m));
}
```

Listing 56: /usr/src/lib/rtminix3/rt_set_sched_bridge.c

```
/* The library function implemented in this file:
 *   rt_set_sched_bridge
 *
 * Function description:
 *   Set the rt_sched_bridge value in the kernel.
 *   Depending on this value a system process may or may not
 *   have a lower priority than real-time processes. A system process
 *   can be lowered in priority due to using its full quantum.
 *   This function is in general only used when shutting down.
 *   At shutdown system processes must be able to get a lower priority than
 *   RT processes otherwise the system won't shutdown properly and crashes.
 *
 * The parameters for this library function are:
 *   int state              The state of the scheduling bridge.
 *                          if 0 system process cannot have a lower priority
 *                          than RT processes. if 1 a system process can.
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/rt.h>

int rt_set_sched_bridge(int state)
{
  message m; /* message we use for the system call */

  m.RT_SCHED_BRIDGE = state; /* set the state */

  /* do the system call and return the result */
  return (_syscall(MM, RT_SET_SCHED_BRIDGE, &m));
}
```

Listing 57: /usr/src/lib/rtminix3/rt_set_sched_edf.c

```c
/* The library function implemented in this file:
 *   rt_set_sched_edf
 *
 * Function description:
 *   This function sets the real-time scheduler to Earliest Deadline First.
 *
 * The parameters for this library function are:
 *   none
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/rt.h>

int rt_set_sched_edf(void)
{
  message m; /* message we use for the system call */

  m.RT_SCHED = SCHED_EDF; /* Set the scheduler parameter to EDF */

  /* do the system call and return the result */
  return (_syscall(MM, RT_SET_SCHED, &m));
}
```

Listing 58: /usr/src/lib/rtminix3/rt_set_sched_rm.c

```
/* The library function implemented in this file:
 *   rt_set_sched_rm
 *
 * Function description:
 *   This function sets the real-time scheduler to Rate-Monotonic.
 *
 * The parameters for this library function are:
 *   int prio_policy         the priority of a RM process can be
 *                           enforced to be unique (PRIO_UNIQUE) or
 *                           not (PRIO_NOT_UNIQUE)
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/rt.h>

int rt_set_sched_rm(int prio_policy)
{
  message m; /* message we use for the system call */

  m.RT_SCHED = SCHED_RM; /* set the scheduler parameter to RM */
  m.RM_PRIO_POLICY = prio_policy; /* set the priority policy parameter */

  /* do the system call and return the result */
  return (_syscall(MM, RT_SET_SCHED, &m));
}
```

Listing 59: /usr/src/lib/klog/klog_copy.c

```
/* The library function implemented in this file:
 *   klog_copy
 *
 * Function description:
 *   This function copies the kernel log from the kernel
 *   to the user program.
 *
 * The parameters for this library function are:
 *   data: a pointer to an array of struct klog_entry with the size of KLOG_SIZE
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/klog.h>

int klog_copy(void *data)
{
  message m; /* message we use for the system call */

  m.KLOG_PTR = data; /* set the pointer to the kernel log in the user program */

  /* do the system call and return the result */
  return (_syscall(MM, KLOG_COPY, &m));
}
```

Listing 60: /usr/src/lib/klog/klog_set.c

```c
/* The library function implemented in this file:
 *   klog_set
 *
 * Function description:
 *   This function sets the kernel logger state to 1.
 *   This means that the kernel will start with logging
 *   untill the buffer is full.
 *
 * The parameters for this library function are:
 *   type        The type of event to log.
 *               KLOG_CONTEXTSWITCH to log the next running
 *               process on every context switch.
 *               KLOG_CLOCKINT to log the current running process
 *               on every clock interrupt.
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is saved in errno.
 */
#include <lib.h>
#include <minix/klog.h>

int klog_set(int type)
{
  message m; /* message we use for the system call */
  m.KLOG_STATE = 1; /* set the state to 1 */
  m.KLOG_TYPE = type; /* what to log? */

  /* do the system call and return the result */
  return (_syscall(MM, KLOG_SET, &m));
}
```

Listing 61: /usr/src/lib/sem/sem_b_create.c

```c
/* The library function implemented in this file:
 *   sem_b_create
 *
 * Function description:
 *   This function creates a new binary semaphore.
 *
 * The parameters for this library function are:
 *   handler:        pointer to handler we want to create
 *   value:          initial value (can be 0 or 1)
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_b_create(sem_t *handler, int value)
{
  message m; /* message we use for the system call */
  int result; /* we will have to save the result of the system call */

  m.SEM_F_TYPE = SEM_BINARY; /* we want to create a binary sem */
  m.SEM_F_VALUE = value; /* the initial value */

  /* do the system call and save the result */
  result = _syscall(SS, SEM_CREATE, &m);

  /* Set the semaphore handler in the user program.
   * The handler is only valid of the result was OK.
   */
  *handler = m.SEM_F_HANDLER;

  return (result);
}
```

Listing 62: /usr/src/lib/sem/sem_c_create.c

```c
/* The library function implemented in this file:
 *   sem_c_create
 *
 * Function description:
 *   This function creates a new counting semaphore.
 *
 * The parameters for this library function are:
 *   handler:        pointer to handler we want to create
 *   value:          initial value (can't be lager than size)
 *   size:           size of the counting semaphore
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_c_create(sem_t *handler, int value, int size)
{
  message m; /* message we use for the system call */
  int result; /* we will have to save the result of the system call */

  m.SEM_F_TYPE = SEM_COUNTING; /* we want to create a counting sem */
  m.SEM_F_VALUE = value; /* set the value in the message */
  m.SEM_F_SIZE = size; /* set the size in the message */

  /* do the system call and save the result */
  result = _syscall(SS, SEM_CREATE, &m);

  /* Set the semaphore handler in the user program.
   * The handler is only valid if the result was OK.
   */
  *handler = m.SEM_F_HANDLER;

  return (result);
}
```

Listing 63: /usr/src/lib/sem/sem_m_create.c

```c
/* The library function implemented in this file:
 *   sem_m_create
 *
 * Function description:
 *   This function creates a new mutex semaphore.
 *
 * The parameters for this library function are:
 *   handler:       pointer to handler we want to create
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_m_create(sem_t *handler)
{
    message m; /* message we use for the system call */
    int result; /* we will have to save the result of the system call */

    m.SEM_F_TYPE = SEM_MUTEX; /* we want to create a mutex */

    /* do the system call and save the result */
    result = _syscall(SS, SEM_CREATE, &m);

    /* Set the semaphore handler in the user program.
     * The handler is only valid if the result was OK.
     */
    *handler = m.SEM_F_HANDLER;

    return (result);
}
```

Listing 64: /usr/src/lib/sem/sem_delete.c

```
/* The library function implemented in this file:
 *   sem_delete
 *
 * Function description:
 *   This function deletes a semaphore.
 *
 * The parameters for this library function are:
 *   handler:       the semaphore handler
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_delete(sem_t handler)
{
  message m; /* message we use for the system call */

  m.SEM_F_HANDLER = handler; /* Set the semaphore handler */

  /* do the system call and return the result */
  return (_syscall(SS, SEM_DELETE, &m));

}
```

Listing 65: /usr/src/lib/sem/sem_flush.c

```
/* The library function implemented in this file:
 *   sem_flush
 *
 * Function description:
 *   This function unblocks all processes that are
 *   blocked on a binary semaphore. It can be used
 *   to synchronize processes.
 *
 * The parameters for this library function are:
 *   handler:       the semaphore handler
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_flush(sem_t handler)
{
  message m; /* message we use for the system call */

  m.SEM_F_HANDLER = handler; /* Set the semaphore handler */

  /* do the system call and return the result */
  return (_syscall(SS, SEM_FLUSH, &m));
}
```

Listing 66: /usr/src/lib/sem/sem_give.c

```
/* The library function implemented in this file:
 *   sem_give
 *
 * Function description:
 *   This function gives a semaphore free
 *
 * The parameters for this library function are:
 *   handler:       the semaphore handler
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_give(sem_t handler)
{
  message m; /* message used for the system call */

  m.SEM_F_HANDLER = handler; /* Set the semaphore handler */

  /* do the system call and return the result */
  return (_syscall(SS, SEM_GIVE, &m));

}
```

Listing 67: /usr/src/lib/sem/sem_take.c

```c
/* The library function implemented in this file:
 *   sem_take
 *
 * Function description:
 *   Take a semaphore.
 *
 * The parameters for this library function are:
 *   handler:       the semaphore handler
 *   block:         if block is WAIT_FOREVER and the sem
 *                  is not available the process will block
 *                  until it is available. If block is NO_WAIT
 *                  the process will not be blocked. Instead if
 *                  a semaphore is not available SEM_IN_USE will
 *                  be returned.
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK an error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_take(sem_t handler, int block)
{
  message m; /* message we use for the system call */

  m.SEM_F_HANDLER = handler; /* set the semaphore handler */
  m.SEM_F_BLOCK = block; /* set the block type */

  /* Do the system call and return the result.
   * If the semaphore is not available and block is WAIT_FOREVER
   * it will return when it is unblocked.
   */
  return (_syscall(SS, SEM_TAKE, &m));
}
```

Listing 68: /usr/src/lib/sem/sem_value.c

```c
/* The library function implemented in this file:
 *   sem_value
 *
 * Function description:
 *   Returns the value of a semaphore
 *
 * The parameters for this library function are:
 *   handler:        the semaphore handler
 *   value:          pointer to store the returned value
 *
 * Return value:
 *   Returns OK (0) if the operation succeeded.
 *   If the return value is not OK the error code
 *   is returned. Check <minix/sem.h> for error codes.
 */
#include <lib.h>
#include <minix/sem.h>

int sem_value(sem_t handler, int *value)
{
  message m; /* message we use for the system call */
  int result; /* we have to save the result of the system call */

  m.SEM_F_HANDLER = handler; /* set the semaphore handler */

  /* do the system call and save the result */
  result = _syscall(SS, SEM_VALUE, &m);

  /* Set the value in the user program */
  *value = m.SEM_F_VALUE;

  return result;
}
```

Listing 69: /usr/src/lib/syslib/sys_klog_copy.c

```c
/* The system library function implemented in this file:
 *   sys_klog_copy
 *
 * Function description:
 *   This function does the kernel call to
 *   copy the kernel log from the kernel
 *   to a user program.
 *
 * The parameters for this library function are:
 *   endpoint:      endpoint of user program
 *   data_ptr:      (virtual) pointer to kernel log in user program
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/klog.h>

int sys_klog_copy(int endpoint, void *data_ptr)
{
  message m; /* message we use for the kernel call */

  m.KLOG_ENDPT = endpoint; /* endpoint of user program */
  m.KLOG_PTR = data_ptr; /* pointer to kernel log in user program */

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_KLOG_COPY, &m));
}
```

Listing 70: /usr/src/lib/syslib/sys_klog_set.c

```
/* The system library function implemented in this file:
 *   sys_klog_set
 *
 * Function description:
 *   This function does the kernel call to set the
 *   kernel logger state. If state is set to 1
 *   the kernel will log until the buffer is full.
 *
 * The parameters for this library function are:
 *   state:     kernel logger state
 *   type:      what to log?
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/klog.h>

int sys_klog_set(int state, int type)
{
  message m; /* message we use for the kernel call */

  m.KLOG_STATE = state; /* set the kernel logger state */
  m.KLOG_TYPE = type; /* what to log? */

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_KLOG_SET, &m));
}
```

Listing 71: /usr/src/lib/syslib/sys_rt_nextperiod.c

```c
/* The system library function implemented in this file:
 *   sys_rt_nextperiod
 *
 * Function description:
 *   This function does the kernel call for processes
 *   scheduled by Earliest Deadline First that wants to
 *   give up remaining calculation time in a period.
 *   The process will wait till the next period start
 *   after this call.
 *
 * The parameters for this library function are:
 *   endpoint:      endpoint of process that wants to give up calculation time
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/rt.h>

int sys_rt_nextperiod(int endpoint)
{
  message m; /* message we use for the kernel call */

  m.RT_ENDPT = endpoint; /* set the endpoint */

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_RT_NEXTPERIOD, &m));
}
```

Listing 72: /usr/src/lib/syslib/sys_rt_set.c

```
/* The system library function implemented in this file:
 *   sys_rt_set
 *
 * Function description:
 *   This function will do the kernel call to transform a normal
 *   user process into a real-time process.
 *
 * The parameters for this library function are:
 *   endpoint:      endpoint of process that wants to be real-time
 *   sched:         scheduler type
 *   param1:        if RM: RM prio if EDF: period
 *   param2:        if EDF: calculation time
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/rt.h>

#define PARAM_RM_PRIO          param1
#define PARAM_EDF_PERIOD       param1
#define PARAM_EDF_CALCTIME     param2

int sys_rt_set(int endpoint, int sched, int param1, int param2)
{
  message m; /* message we use for the kernel call */

  m.RT_ENDPT = endpoint; /* set the endpoint of the process that wants to be RT */
  m.RT_SCHED = sched; /* set the scheduler */

  if (sched == SCHED_RM) {

      /* Rate-Monotonic specific parameters */
      m.RM_PRIO = PARAM_RM_PRIO; /* set priority policy */
  } else if (sched == SCHED_EDF) {

      /* Earliest Deadline First specific parameters */
      m.EDF_PERIOD = PARAM_EDF_PERIOD; /* set the period in ticks */
      m.EDF_CALCTIME = PARAM_EDF_CALCTIME; /* set the calculation time per period
          in ticks */
  } else {
      /* unknown scheduler type */
      return (EINVAL);
  }

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_RT_SET, &m));
}
```

Listing 73: /usr/src/lib/syslib/sys_rt_set_sched.c

```
/* The system library function implemented in this file:
 *   sys_rt_set_sched
 *
 * Function description:
 *   This function sets the real-time scheduler.
 *
 * The parameters for this library function are:
 *   type:       scheduler type
 *   policy:     Rate-Monotonic priority policy (used by RM only)
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/rt.h>

int sys_rt_set_sched(int type, int policy)
{
  message m; /* message we use for the kernel call */

  m.RT_SCHED = type; /* set the scheduler type */

  /* rate-monotonic specific parameter */
  if (type == SCHED_RM) {
      m.RM_PRIO_POLICY = policy; /* set the prio policy */
  }

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_RT_SET_SCHED, &m));
}
```

Listing 74: /usr/src/lib/syslib/sys_rt_sched_bridge.c

```
/* The system library function implemented in this file:
 *   sys_rt_sched_bridge
 *
 * Function description:
 *   This function does the kernel call to set the scheduler
 *   bridge state. Depending on this state system processes
 *   are allowed or not allowed to get a lower priority than
 *   RT processes.
 *
 * The parameters for this library function are:
 *   state:     scheduling bridge state
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/rt.h>

int sys_rt_sched_bridge(int state)
{
  message m; /* message we use for the kernel call */

  m.RT_SCHED_BRIDGE = state; /* set the bridge state */

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_RT_SCHED_BRIDGE, &m));
}
```

Listing 75: /usr/src/lib/syslib/sys_rt_show_data.c

```c
/* The system library function implemented in this file:
 *   sys_rt_show_data
 *
 * Function description:
 *   This function does a kernel call to dump
 *   real-time scheduler info.
 *
 * The parameters for this library function are:
 *   none
 *
 * Return value:
 *   Returns 0 if the operation succeeded.
 *   If the return value is not 0 the error code
 *   is returned.
 */
#include "syslib.h"
#include <minix/rt.h>

int sys_rt_show_data(void)
{
  message m; /* message we use for the kernel call */

  /* do the kernel call and return the result */
  return (_taskcall(SYSTASK, SYS_RT_SHOW_DATA, &m));
}
```

Listing 76: /usr/src/commands/reboot/halt.c

```
/* halt / reboot - halt or reboot system (depends on name)

   halt   - calling reboot() with RBT_HALT
   reboot - calling reboot() with RBT_REBOOT

   author: Edvard Tuinder   v892231@si.hhs.NL

   This program calls the library function reboot(2) which performs
   the system-call do_reboot.

 */

#define _POSIX_SOURCE   1
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <minix/rt.h>

void write_log _ARGS(( void ));
void usage _ARGS(( void ));
int main _ARGS(( int argc, char *argv[] ));

char *prog;
char *reboot_code = "delay; boot";

void
usage()
{
  fprintf(stderr, "Usage: %s [-hrRf] [-x reboot-code]\n", prog);
  exit(1);
}

int
main(argc,argv)
int argc;
char **argv;
{
  int flag = -1;                /* default action unknown */
  int fast = 0;                 /* fast halt/reboot, don't bother being nice. */
  int i;
  struct stat dummy;
  char *monitor_code = "";
  pid_t pid;

  if ((prog = strrchr(argv[0],'/')) == NULL) prog = argv[0]; else prog++;

  if (strcmp(prog, "halt") == 0) flag = RBT_HALT;
  if (strcmp(prog, "reboot") == 0) flag = RBT_REBOOT;

  i = 1;
  while (i < argc && argv[i][0] == '-') {
    char *opt = argv[i++] + 1;
    if (*opt == '-' && opt[1] == 0) break;      /* -- */

    while (*opt != 0) switch (*opt++) {
      case 'h': flag = RBT_HALT;        break;
      case 'r': flag = RBT_REBOOT;      break;
      case 'R': flag = RBT_RESET;       break;
      case 'f': fast = 1; break;
      case 'x':
        flag = RBT_MONITOR;
        if (*opt == 0) {
          if (i == argc) usage();
          opt = argv[i++];
        }
        monitor_code = opt;
        opt = "";
        break;
      default:
        usage();
    }
  }

  if (i != argc) usage();

  if (flag == -1) {
    fprintf(stderr, "Don't know what to do when named '%s'\n", prog);
    exit(1);
  }

  if (flag == RBT_REBOOT) {
        flag = RBT_MONITOR;             /* set monitor code for reboot */
        monitor_code = reboot_code;
  }

  if (stat("/usr/bin", &dummy) < 0) {
    /* It seems that /usr isn't present, let's assume "-f." */
    fast = 1;
  }

  signal(SIGHUP, SIG_IGN);
  signal(SIGTERM, SIG_IGN);

  /* Skip this part for fast shut down. */
  if (! fast) {
    /* Run the shutdown scripts. */
    switch ((pid = fork())) {
      case -1:
        fprintf(stderr, "%s: can't fork(): %s\n", prog, strerror(errno));
        exit(1);
      case 0:
        execl("/bin/sh", "sh", "/etc/rc", "down", (char *) NULL);
        fprintf(stderr, "%s: can't execute: /bin/sh: %s\n",
          prog, strerror(errno));
        exit(1);
      default:
        while (waitpid(pid, NULL, 0) != pid) {}
    }
  }

  /* Tell init to stop spawning getty's. */
  kill(1, SIGTERM);

  /* Build scheduling bridge.
```

```
     * This allows system processes to get a lower priority than
     * real-time processes. Otherwise system will not shutdown properly.
     */
    if (rt_set_sched_bridge(1) != 0) {
        printf("Setting up scheduling bridge failed!\n");
    }

    /* Give everybody a chance to die peacefully. */
    printf("Sending SIGTERM to all processes ...\n");
    kill(-1, SIGTERM);
    sleep(1);

    write_log();

    sync();

    reboot(flag, monitor_code, strlen(monitor_code));
    fprintf(stderr, "%s: reboot(): %s\n", strerror(errno));
    return 1;
}
```

Listing 77: /usr/src/commands/simple/edftop.c

```c
/* Author: Ben Gras <beng@few.vu.nl>  17 march 2006 */

/* Modified top to display information for edf processes
 * called edftop, by Bianco Zandbergen (20 june 2009)
 */
#define _MINIX 1
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <pwd.h>
#include <curses.h>
#include <timers.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>
#include <termcap.h>
#include <termios.h>
#include <time.h>
#include <string.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>

#include <sys/ioc_tty.h>
#include <sys/times.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>

#include <minix/ipc.h>
#include <minix/config.h>
#include <minix/type.h>
#include <minix/const.h>

#include <math.h>

#include "../../servers/pm/mproc.h"
#include "../../kernel/const.h"
#include "../../kernel/proc.h"

#define  TC_BUFFER  1024        /* Size of termcap(3) buffer    */
#define  TC_STRINGS  200        /* Enough room for cm,cl,so,se  */

char *Tclr_all;

int print_memory(struct pm_mem_info *pmi)
{
        int h;
        int largest_bytes = 0, total_bytes = 0;
        for(h = 0; h < _NR_HOLES; h++) {
                if(pmi->pmi_holes[h].h_base && pmi->pmi_holes[h].h_len) {
                        int bytes;
                        bytes = pmi->pmi_holes[h].h_len << CLICK_SHIFT;
                        if(bytes > largest_bytes) largest_bytes = bytes;
                        total_bytes += bytes;
                }
        }

    printf("Mem: %dK Free, %dK Contiguous Free\n",
        total_bytes/1024, largest_bytes/1024);

    return 1;
}

int print_load(double *loads, int nloads)
{
    int i;
    printf("load averages: ");
    for(i = 0; i < nloads; i++)
        printf("%s %.2f", (i > 0) ? "," : "", loads[i]);
    printf("\n");
    return 1;
}

#define PROCS (NR_PROCS+NR_TASKS)

int print_proc_summary(struct proc *proc)
{
    int p, alive, running, sleeping;

    alive = running = sleeping = 0;

    for(p = 0; p < PROCS; p++) {
        if(p - NR_TASKS == IDLE)
            continue;
        if(proc[p].p_rts_flags & SLOT_FREE)
            continue;
        alive++;
        if(proc[p].p_rts_flags & ~SLOT_FREE)
            sleeping++;
        else
            running++;
    }
    printf("%d processes: %d running, %d sleeping\n",
        alive, running, sleeping);
    return 1;
}

static struct tp {
    struct proc *p;
    int ticks;
} tick_procs[PROCS];

int cmp_ticks(const void *v1, const void *v2)
{
    struct tp *p1 = (struct tp *) v1, *p2 = (struct tp *) v2;
    if(p1->ticks < p2->ticks)
        return 1;
    if(p1->ticks > p2->ticks)
        return -1;
    if(p1->p->p_nr < p2->p->p_nr)
        return -1;
    if(p1->p->p_nr > p2->p->p_nr)
        return 1;
    return 0;
}

void print_procs(int maxlines,
    struct proc *proc1, struct proc *proc2, int dt,
    struct mproc *mproc)
{
```

```
int p, nprocs, tot=0;
int idleticks = 0, kernelticks = 0, systemticks = 0;

if(dt < 1) return;

for(p = nprocs = 0; p < PROCS; p++) {
    if(proc2[p].p_rts_flags & SLOT_FREE)
        continue;
    tick_procs[nprocs].p = proc2 + p;
    if(proc1[p].p_endpoint == proc2[p].p_endpoint) {
        tick_procs[nprocs].ticks =
            proc2[p].p_user_time-proc1[p].p_user_time;
    } else {
        tick_procs[nprocs].ticks =
            proc2[p].p_user_time;
    }
    if(p-NR_TASKS == IDLE) {
        idleticks = tick_procs[nprocs].ticks;
        continue;
    }

    /* Kernel task time, not counting IDLE */
    if(proc2[p].p_nr < 0)
        kernelticks += tick_procs[nprocs].ticks;
    else if(mproc[proc2[p].p_nr].mp_procgrp == 0)
        systemticks += tick_procs[nprocs].ticks;
    nprocs++;
}

qsort(tick_procs, nprocs, sizeof(tick_procs[0]), cmp_ticks);

printf("CPU states: %6.2f%% user, %6.2f%% system, %6.2f%% kernel, %6.2f%%
    idle",
    100.0*(dt-systemticks-kernelticks-idleticks)/dt,
    100.0*systemticks/dt,
    100.0*kernelticks/dt,
    100.0*idleticks/dt);
printf("\n\n");
maxlines -= 2;

printf("PNR   ENDPT  ST PERIOD CTIME  PRD#  TLP TLNP     USED RESERVED  %%U
    MD COMMAND\n");
maxlines--;

/* prints EDF process list */
for(p=0; p < PROCS; p++) {
    if (proc2[p].p_rts_flags & SLOT_FREE) continue;

    if (proc2[p].p_rt != 1) continue;

    if (maxlines-- <= 0) break;

    /* print state */
    printf("%3d", proc2[p].p_nr);

    /* print endpoint */
    printf(" %7d", proc2[p].p_endpoint);

    /* print state */
    if (proc2[p].p_rts_flags == 0) {
        printf(" RUN");
    } else if (proc2[p].p_rts_flags & NEXTPERIOD) {
            printf(" WNP");
        } else if (proc2[p].p_rts_flags != 0) {
            printf(" BLK");
        } else {
            printf("       ");
        }

        /* print period in ticks */
        printf(" %6d", proc2[p].p_rt_period);

        /* print calculation time per period in ticks */
        printf(" %5d", proc2[p].p_rt_calctime);

        /* print current period number (unfinished period) */
        printf(" %5u", proc2[p].p_rt_periodnr);

        /* print calculation time left in current period */
        printf(" %4d", proc2[p].p_rt_ticksleft);

        /* print ticks till next period start (deadline) */
        printf(" %4d", proc2[p].p_rt_nextperiod);

        /* print total used ticks and total reserved ticks */
        printf(" %8u %8u", proc2[p].p_rt_totalused, proc2[p].p_rt_totalreserved);

        /* print percentage of ticks used */
        if (proc2[p].p_rt_totalused == 0 || proc2[p].p_rt_totalreserved == 0) {
            printf("  ??"); /* prevent divide by zero */
        } else {
            printf(" %3.0f", (float)((float)(proc2[p].p_rt_totalused /
                (float)proc2[p].p_rt_totalreserved) * (float)100));
        }

        /* print missed deadlines */
        printf(" %2u", proc2[p].p_rt_missed_dl);

        /* print process name */
        printf(" %s\n", proc2[p].p_name);
    }
}

void showtop(int r)
{
#define NLOADS 3
    double loads[NLOADS];
    int nloads, i, p, lines = 0;
    static struct proc prev_proc[PROCS], proc[PROCS];
    struct winsize winsize;
        static struct pm_mem_info pmi;
    static int prev_uptime, uptime;
    static struct mproc mproc[NR_PROCS];
    struct tms tms;

    uptime = times(&tms);

    if(ioctl(STDIN_FILENO, TIOCGWINSZ, &winsize) != 0) {
        perror("TIOCGWINSZ");
        fprintf(stderr, "TIOCGWINSZ failed\n");
        exit(1);
    }

        if(getsysinfo(PM_PROC_NR, SI_MEM_ALLOC, &pmi) < 0) {
```

```
        fprintf(stderr, "getsysinfo() for SI_MEM_ALLOC failed.\n");
        exit(1);;
    }

    if(getsysinfo(PM_PROC_NR, SI_KPROC_TAB, proc) < 0) {
        fprintf(stderr, "getsysinfo() for SI_KPROC_TAB failed.\n");
        exit(1);
    }

    if(getsysinfo(PM_PROC_NR, SI_PROC_TAB, mproc) < 0) {
        fprintf(stderr, "getsysinfo() for SI_PROC_TAB failed.\n");
        exit(1);
    }

    if((nloads = getloadavg(loads, NLOADS)) != NLOADS) {
        fprintf(stderr, "getloadavg() failed - %d loads\n", nloads);
        exit(1);
    }


    printf("%s", Tclr_all);

    lines += print_load(loads, NLOADS);
    lines += print_proc_summary(proc);
    lines += print_memory(&pmi);

    if(winsize.ws_row > 0) r = winsize.ws_row;

    print_procs(r - lines - 2, prev_proc,
        proc, uptime-prev_uptime, mproc);

    memcpy(prev_proc, proc, sizeof(prev_proc));
    prev_uptime = uptime;
}

void init(int *rows)
{
    char  *term;
    static char   buffer[TC_BUFFER], strings[TC_STRINGS];
    char *s = strings, *v;

    *rows = 0;

    if(!(term = getenv("TERM"))) {
        fprintf(stderr, "No TERM set\n");
        exit(1);
    }

    if ( tgetent( buffer, term ) != 1 ) {
        fprintf(stderr, "tgetent failed for term %s\n", term);
        exit(1);
    }

    if ( (Tclr_all = tgetstr( "cl", &s )) == NULL )
        Tclr_all = "\f";

    if((v = tgetstr ("li", &s)) != NULL)
        sscanf(v, "%d", rows);
    if(*rows < 1) *rows = 24;
    if(!initscr()) {
        fprintf(stderr, "initscr() failed\n");
        exit(1);
    }
    cbreak();
    nl();
}

void sigwinch(int sig) { }

int main(int argc, char *argv[])
{
    int r, c, s = 0, orig;

    init(&r);

    while((c=getopt(argc, argv, "s:")) != EOF) {
        switch(c) {
            case 's':
                s = atoi(optarg);
                break;
            default:
                fprintf(stderr,
                    "Usage: %s [-s<secdelay>]\n", argv[0]);
                return 1;
        }
    }

    if(s < 1)
        s = 2;

    /* Catch window size changes so display is updated properly right away. */
    signal(SIGWINCH, sigwinch);

    while(1) {
        fd_set fds;
        int ns;
        struct timeval tv;
        showtop(r);
        tv.tv_sec = s;
        tv.tv_usec = 0;

        FD_ZERO(&fds);
        FD_SET(STDIN_FILENO, &fds);
        if((ns=select(STDIN_FILENO+1, &fds, NULL, NULL, &tv)) < 0
            && errno != EINTR) {
            perror("select");
            sleep(1);
        }

        if(ns > 0 && FD_ISSET(STDIN_FILENO, &fds)) {
            char c;
            if(read(STDIN_FILENO, &c, 1) == 1) {
                switch(c) {
                    case 'q':
                        return 0;
                        break;
                }
            }
        }
    }

    return 0;
}
```

Listing 78: /usr/src/commands/simple/klog_copy.c

```c
#include <stdio.h>
#include <minix/klog.h>

struct klog_entry klog[KLOG_SIZE];

int main(void)
{
    int i;

    if (klog_copy(&klog) != 0) {
        printf("Copying kernel log failed!\n");
        return -1;
    }

    for (i=0;i<KLOG_SIZE;i++) {
        printf("%d %d %s\n", klog[i].klog_time, klog[i].klog_endpoint,
            klog[i].klog_name);
    }

    return 0;

}
```

Listing 79: /usr/src/commands/simple/klog_set_ci.c

```c
#include <stdio.h>
#include <minix/klog.h>

int main(void)
{
    if (klog_set(KLOG_CLOCKINT) != 0) {
        printf("Enabling kernel log (on clock interrupt) failed!\n");
        return -1;
    }

    return 0;
}
```

Listing 80: /usr/src/commands/simple/klog_set_cs.c

```c
#include <stdio.h>
#include <minix/klog.h>

int main(void)
{
    if (klog_set(KLOG_CONTEXTSWITCH) != 0) {
        printf("Enabling kernel log (on context switch) failed!\n");
        return -1;
    }

    return 0;
}
```

Listing 81: /usr/src/commands/simple/rt_set_sched_edf.c

```
#include <stdio.h>
#include <minix/rt.h>

int main(void)
{
    if (rt_set_sched_edf() == 0) {
        printf("RT scheduler set to EDF\n");
        return 0;
    } else {
        printf("Setting RT scheduler to EDF failed!\n");
        return 1;
    }
}
```

Listing 82: /usr/src/commands/simple/rt_set_sched_rm_pnu.c

```c
#include <stdio.h>
#include <minix/rt.h>

int main(void)
{
    if (rt_set_sched_rm(PRIO_NOT_UNIQUE) == 0) {
        printf("RT scheduler set to RM prio not unique\n");
        return 0;
    } else {
        printf("Setting RT scheduler to RM prio not unique failed!\n");
        return 1;
    }
}
```

Listing 83: /usr/src/commands/simple/rt_set_sched_rm_pu.c

```c
#include <stdio.h>
#include <minix/rt.h>

int main(void)
{
    if (rt_set_sched_rm(PRIO_UNIQUE) == 0) {
        printf("RT scheduler set to RM prio unique\n");
        return 0;
    } else {
        printf("Setting RT scheduler to RM prio unique failed!\n");
        return 1;
    }
}
```