RT-MINIXv2: Real-Time Process Management and Scheduling

Pablo A. Pessolani
Departamento de Sistemas - Facultad Regional Santa Fe- Universidad Tecnológica Nacional
ppessolani@hotmail.com

***Abstract* — Tanenbaum's MINIX operating system was extended with real-time services to conform RT-MINIXv2. It is a Real-Time Operating System to teach concepts related to real-time kernels but keeps compatibility with MINIX. This article discuss how processes are managed, created and terminated, and finally describes the scheduling algorithm performed by the real-time kernel..**

***Keywords*— Operating Systems, Minix, Process Management, Real-Time Scheduling.**

I. INTRODUCTION

Real-Time Operating Systems (RTOS) instructors can choice among commercial or free licence software to develop their laboratory practice. Commercially available RTOS as RTLinux (http://www.fsmlabs.com/) or QNX (http://www.qnx.com) are too costly and proprietary to be used by academic institutions. Free licence and open source RTOS, as RTAI (http://www.rtai.org/) or the Kansas University real-time variant KURT (http://www.ittc.ku.edu/kurt) have been designed focusing on performance as a key design feature with complex source code readability for grade course students. The first real-time version of MINIX have same drawbacks an it have limited features [1].

A new real-time version of the well known MINIX [2] academic operating system (OS) named RT-MINIXv2 [3] is proposed to teach concepts of real-time programming, in particular those related to real-time kernels.

This article address the Process Management and Scheduling of this version. Also describes an approach to allow that two operating systems, a general purpose operating system and a predictable real-time kernel, co-exist on the same hardware.

The design constrains established for RT-MINIXv2 are:

- *Compatibility with MINIX*: All process that run on MINIX must run on RT-MINIXv2 without modifications and sensible performance impact.
- *Minimal MINIX source code changes*: As MINIX is often used in OS design courses, students have deep knowledge of its source code. Therefore reducing the changes keep the students experience to learn a MINIX based RTOS. Most new code must be added in separeted functions with few changes in the original MINIX code. This also helps for system updates when new versions of MINIX are released.
- *Source Code readability*: As RT-MINIXv2 is focused for academic uses, its source code must be easily understood, perhaps sacrifying performance.

RT-MINIXv2 will let instructors make easily a multiplicity of grade courses assignments, laboratory tests, projects and other academic uses with an open source RTOS. Much of the existing MINIX tools, as text editors, browsers, compilers, linkers, etc. could be used for those practices minimizing the students' apprenticeship.

The aim of the RT-MINIXv2 project is to provide an educational tool for RTOS courses as MINIX does for OS Design and Implementation courses.

The rest of this work is organized as follows. Section II presents an overview of RT-MINIXv2. Section III and Section IV are about MINIX and RT-MINIXv2 process management respectively. Section V, describes the RT-process scheduler. Finally, Section VI presents conclusions.

To simplify the notation, for the following paragraphs all Real-Time related words will be preceded by "RT" prefix and Non Real-Time words will be preceded by "NRT" prefix.

II. OVERVIEW

As Bollella [4] describes, there have three identifiable approaches to marrying RT and NRT technologies, but Yodaiken [5] proposes a new one where a RT-microkernel threats a time-sharing OS as the lower priority task. This approach was adopted for the RT-MINIXv2 kernel. It have the following advantages:

A. By allowing the co-existence of two OS, a clean separation exists between NRT and RT-services
B. The RT-kernel executes in a predictable manner, so it is possible to analyze the conditions under which RT-processes will be guaranteed to be feasible.
C. The time sharing OS can function correctly with few modifications.

RT-MINIXv2 has two execution modes:

- *NRT-mode*: the system runs as MINIX. No RT-processes and interrupt handlers are admited.
- *RT-mode*: The system runs under the RT-kernel control.

To change the execution mode the *rtm_RTstart()* and *rtm_RTstop()* system calls are provided. They are implemented using a special task named *RTMTASK* that is RT-kernel agent that function as glue among processes and the RTOS.

Only NRT-process can be created and terminated under RT-MINIXv2. The RT-kernel does not add new system calls to create RT-processes. On the other hand a NRT-process is converted into a RT-process using the *rtm_setproc()* system call. To terminate a RT-process it must be converted back into a NRT-process (explained in Section IV).

When the system runs in NRT-mode, any process that calls *rtm_setproc()* trying to convert itself into a RT-process, receives an error return code.

It must be considered that on RT-MINIXv2 co-exist two kernels with a shared set of processes but with their own process states and transitions each.

III. BACKGROUND ABOUT MINIX PROCESS MANAGEMENT

As is described by Tanenbaum [2], a process in MINIX have 3 basic states and 4 state transitions. The READY state, the BLOCKED state and the RUNNING state.

When MINIX runs under the RT-kernel control, a fourth process state named REALTIME is added (Fig. 1). A NRT-process is in this state when it has been converted into a RT-process. It will be ignored by the *ready()* MINIX kernel function and ineligible by its scheduler.

The following are the process states transitions under MINIX:

1. *READY to RUNNING*
2. *RUNNING to READY*
3. *RUNNING to BLOCKED*
4. *BLOCKED to READY*

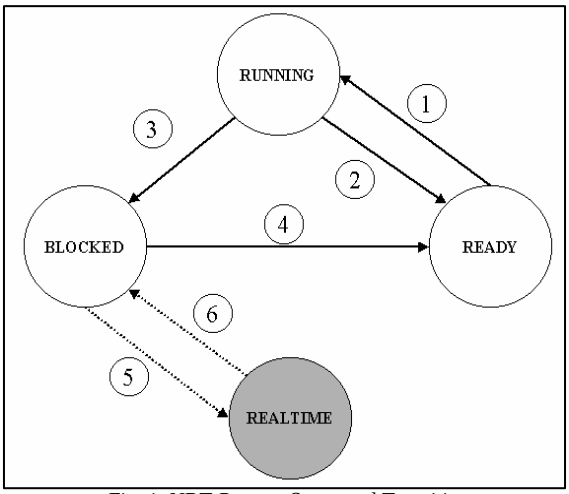

Fig. 1: NRT-Process States and Transitions.

When MINIX is running under the RT-kernel control, two new transitions states are added.

5. *BLOCKED to REALTIME*: The NRT-process has done a *rtm_setproc()* system call to convert it into a RT-process.
6. *REALTIME to BLOCKED*: The RT-process has done a *rtm_setproc()* system call to convert itself into a NRT-process, or when the RT-process calls *exit()* or when it receives a NRT-signal.

IV. RT-MINIXV2 PROCESS MANAGEMENT

A. Real-Time Process Creation

When a NRT-process invokes *rtm_setproc()* system call, it will be in a *BLOCKED* state. Therefore, there is no need to remove that process from any MINIX *READY* queue and could be converted into a RT-process.

Because there are two process schedulers that run on RT-MINIXv2 (the real-time and the MINIX scheduler), some bits in one field in the process descriptor are used to distinguish among RT-processes and NRT-processes. Those bits avoid that a NRT-process could unlock a RT-process and could be inserted into one of the MINIX *READY* queues and be eligible by the NRT-scheduler.

B. RT-Process States and Transitions

The following are the states of a RT-process (Fig. 2):

- *RT-READY*: The RT-process is ready to run and waiting to be selected by the RT-process scheduler.
- *RT-BLOCKED*: The RT-process is suspended (sleeping) because it has done a blocking RT-system call to the RT-kernel.
- *RT-RUNNING*: The RT-process is running under RT-kernel control.
- *STANDARD*: The RT-process has been converted into a NRT-process and must be ignored by the RT-kernel.

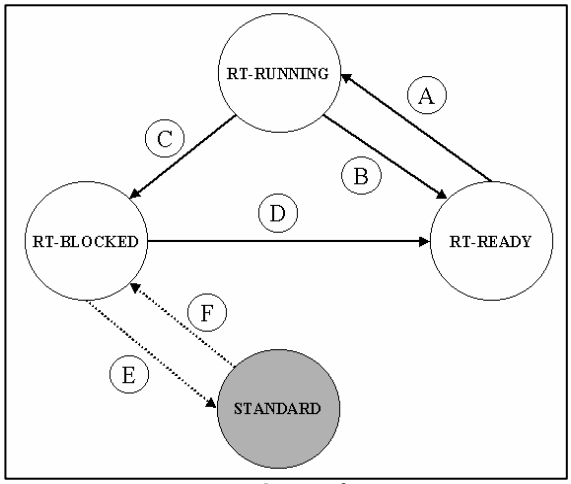

Fig. 2: RT-Process States and Transitions.

The RT-process state transitions are:

A. *RT-READY to RT-RUNNING*: The RT-process has been selected to run by the RT-scheduler.
B. *RT-RUNNING to RT-READY*: The running RT-process has been preempted by other RT-process with higher priority.
C. *RT-RUNNING to RT-BLOCKED*: The RT-process has done a blocking RT-system call.
D. *RT-BLOCKED to RT-READY*: The process has returned from a RT-system call.
E. *RT-BLOCKED to STANDARD*: The RT-process has done a *rtm_setproc()* system call to convert it back into a NRT-process. This transition also occurs when the RT-process calls *exit()* or when a RT-process receives a NRT-signal.
F. *STANDARD to RT-BLOCKED*: The NRT-process has done a *rtm_setproc()* system call to convert it into a RT-process.

The two states called *STANDARD* and *REALTIME* are compound states. *REALTIME* is the set of RT-process states and *STANDARD* is the set of NRT-process states.

C. Process Descriptor Real-Time fields

The MINIX kernel uses a process descriptor table to keep the status information of every process in the system.

Each process descriptor has a field named *p_flags* to indicate the reason why a process is blocked. If *p_flags = 0*, the process can be scheduled by the MINIX process scheduler.

The RT-scheduler recognizes two kinds of processes: periodic and sporadic. A periodic process runs repeatedly, and within a fixed time (period). A sporadic process runs when one event trigger it. There are two bits assigned in

p_flags to distinguish among periodic and sporadic task. The RT-scheduler can only schedule processes with one of these bits set.

New fields were added to the process data structure for RT-process management and statistics collection. Some of these fields are RT-process characterization parameters as:

- *priority*: The efective scheduling priority used by the RT-scheduler to select the next RT-process to run. It is explained in Section V.
- *baseprty*: The base priority used by the Basic Priority Inheritance Protocol (BPIP) to restore the efective priority. It is explained in Section V.
- *period*: The scheduling period of a RT-periodic process. It is specified in RT-timer ticks.
- *limit*: A limit for the number of RT-schedulings.
- *deadline*: The process deadline. It is specified in RT-timer ticks.
- *watchdog*: The PID of a RT-process that provides services to protect the RT-process against deadline expiration. The watchdog can be programmed to perform several actions on the occurrence of a process overrun. When a RT-process does not complete it's work before it's deadline expiration, the RT-kernel could send a *MT_WATCHDOG* message to the watchdog process specified in the descriptor.

D. RT-process Termination

When the system runs in RT-mode, the RT-kernel does not terminate any RT-process, it delegates this task to the MINIX kernel, but before a RT-process can be terminated, it is converted into a NRT-process (Transition F on Fig. 2).

In MINIX, a process could be terminated by one of the following reasons:

- *Normal exit*: Invoking the *exit()* system call by itself (voluntary).
- *Error exit*: The process discover a fatal error and invokes the *exit()* system call (voluntary).
- *Fatal error*: MINIX discovers a process fatal error (often a program bug) and could terminate it sending a signal (unvoluntary).
- *Killed*: Other process send an uncached signal to the process (unvoluntary).

These four reasons show that a process could be terminated by the *exit()* system call or by the *signal()* system call.

1) RT-process Termination using the exit() system call

As RT-processes can not send/receive messages using *send()/receive()* MINIX primitives (except to/from *RTMTASK*), if a RT-process invokes the *exit()* system call it is converted into a NRT-process before the MINIX *exit()* function could be invoked. That conversion is accomplished by the modified *exit()* library function that uses the services of *RTMTASK*.

2) RT-process Termination using the signal() system call

If a NRT-process sends a signal to a RT-process, the target is converted into a NRT-process before it can receive the signal. That conversion is accomplished by the modified *check_sig()* function of the Memory Manager Server (*MM*) using the services of *RTMTASK*.

3) Freeing RT-process resourses and housecleaning

The RT-kernel makes some housecleaning tasks before converting RT-process into a NRT one to keep the system resouces and concistency. Those tasks free unused RT-resources like virtual timers, mailboxes, etc. detailed as following:

- Any RT-process that have a synchronous message waiting in the Mailbox of the terminating RT-process is woken up and the *rtm_send()* system call returns an error code.

- A *MT_SIGNAL* message is sent to the terminating RT-process watchdog.

- A *MT_SIGNAL* message is sent to all process with the watchdog field equals to the terminating RT-process. The watchdog fields are set to HARDWARE.

- All RT-interrupt descriptors watchdog fields equals to the terminating RT-process are set to HARDWARE.

- All asyncronous messages sent by the terminating RT-process to other mailboxes are removed.

The RT-kernel frees the terminating RT-process mailbox and virtual timers.

V. RT-MINIXv2 PROCESS SCHEDULER

A. The RT-Process Scheduler

The process scheduler is the component of the kernel that selects which process to run next. The scheduler can be viewed as the code that divides, using a defined policy, the finite resource of processor time between the runnable processes on a system.

The set of rules used to determine when and how to select a new process to run is called scheduling policy[6]. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time. The policy behind RT-MINIXv2 scheduler is simple:

"A priority scheduled real-time system must ensure that the highest priority runnable process can start to run in a bounded time—and the bound needs to be small". [7]

The RT-scheduler always selects the highest priority runnable process for execution. All unvoluntary context switches are triggered by interrupts. Timer interrupts can cause preemption due to Timer-Driven process activation. If the priority of the activated process is higher than the priority of the currently running process, the execution of current is interrupted and the RT-scheduler is invoked to select another RT-process to run. The RT-scheduler tries to find a ready RT-process with the highest priority. If there are not such process, the MINIX process scheduler is called.

The RT-scheduler uses an optimized process-selection algorithm, based on a set of priority queues and a bitmap [8]. Each bit in the bitmap represents a priority queue. If a bit is set, it means that at least one process is ready in that queue.

Typically, the bitmap is scanned for the highest priority non-empty queue, and the first process in that queue is selected to run.

The RT-scheduler implements fully O(1) scheduling. The algorithm completes in constant-time, regardless of the number of running processes.

RT-MINIXv2 does not use a timeslice for preempting a RT-process. Only a higher priority RT-process can preempt the running process or it must relinquish the CPU by itself.

When a higher priority RT-process enters the *RT-READY* state, the kernel checks whether its priority is higher than the priority of the currently executing RT-process. If it is,

the RT-scheduler is invoked to pick a new RT-process to run (presumably the process that just became runnable).

B. Process Priorities

A common type of scheduling algorithm is priority-based scheduling. Processes with a higher priority will run before those with a lower priority, while processes with the same priority are scheduled round-robin (one after the next, repeating).

1) NRT-Process Priorities

Minix scheduler uses multiple queue-scheduling algorithms for all the processes. They are:

- *TASK_Q*: with the highest priority.
- *SERVER_Q*: with a medium priority.
- *USER_Q*: with the lowest priority.

All process in the same queue have the same priority and the process scheduler selects the next runnable process for execution in FCFS order for tasks and servers and in FIFO order for user level processes.

2) RT-Process Priorities

A RT-processes can have a scheduling priority ranging from 0 to 15; low numeric values correspond to high priorities. A RT-process has both a base priority and an effective priority and is scheduled in accordance with the latter.

The RT-process itself can set its base and effective priority together, but the effective priority may be changed by the Basic Priority Inheritance Protocol [9] implemented in RT-Interprocess Communications (RT-IPC).

The RT-ready queue is implemented as 16 separate queues, one for each priority. Each ready RT-process is appended to the queue corresponding to its effective priority field in FIFO order. The first process descriptor in the highest priority queue will be selected to run.

The *priority* field is also used by the RT-kernel to reduce the interrupt blocking time. Only those interrupt handlers with higher priorities are executed while the current RT-process is running [3].

If all RT-queues are empty, the MINIX scheduler is invoked to select a NRT-process for execution.

C. Priority Queues Management

To manage the RT-ready queues the RT-kernel uses the following data structures:

- *A set of queue descriptors*: It assign one queue descriptor for each priority level.
- *A priority bitmap*: Each bit of this bitmap is allocated for a priority queue descriptor. Initially, all the bits are zero indicating that all queues are empty.

When a RT-process becomes runnable (that is, its state becomes *RT-READY*), the corresponding bit to its priority is set in bitmap, and the process descriptor is appended to the RT-ready queue in accordance with its priority.

Finding the highest priority RT-process on the system is therefore only a matter of finding the first bit set in the priority bitmap. Because the number of priorities is fixed, the time to complete a search is constant and unaffected by the number of running processes on the system.

Each priority queue descriptor have two pointers, one for the head and one for the tail of the queue. The insertions in the queue can be in FIFO or LIFO order. Processes of the same priority will be managed under a FIFO policy. A process that inherits priority by RT-IPC must be inserted and removed from the queues in LIFO order.

Each queue descriptor also contains a field named *inQ* that keeps the current number of runnable RT-processes in the queue.

A set of RT-kernel helper functions let system programmers append, insert or remove process descriptors into/from priority queues.

VI. CONCLUSIONS

RT-MINIXv2 subkernel architecture, source code readability and MINIX compatibility make it suitable for course assignments and Real-Time project developments. Its microkernel have simple process management and efficient scheduling that make it a good choice to conduct RT-experiences as implement Rate Monotonic or Deadline Monotonic algorithms with simple changes.

Time Management, Real-Time Interprocess Communications and Statistics Collections features are being added to RT-MINIXv2, at time of this writing,

ACKNOWLEDGMENTS

The author gratefully acknowledges help received from Telecom Argentina S.A. for sponsoring the author's UNLP Master and Dr. Silvio Gonnet for the revision of this article.

REFERENCES

[1] Gabriel A. Wainer, "Implementing Real-Time services in MINIX", ACM Operating Systems Review, July 1995.

[2] Tanenbaum Andrew S., Woodhull Albert S., "Operating Systems Design and Implementation" 2nd Edition, ISBN: 0-13-638677-6, Prentice-Hall , 1997

[3] Pessolani, Pablo A, "RT-MINIXv2: Architecture and Interrupt Handling", *33th JAIIO, Argentine Symposium on Computing Technology (AST2004)*, 2004.

[4] Gregory Bollella, Kevin Jeffay ,"Support For Real-Time Computing Within General Purpose Operating Systems", 1995

[5] Victor Yodaiken, "The RTLinux Manifesto", *Department of Computer Science New Mexico Institute of Technology.*

[6] Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel Second Edition", O'Reilly, 2003.

[7] Victor Yodaiken, "Against Priority Inheritance", *Fsmlabs Technical Report*, June 25, 2002

[8] "About UNIX and Real-Time Scheduling", http://www.pcengines.ch/schedule.htm., 1989,

[9] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to RealTime Synchronisation", *IEEE Transactions on Computers 39(9)*, pp. 1175-1185 (September 1990).