MINIX4RT: Real-Time Semaphores

Pablo A. Pessolani
Departamento de Sistemas - Facultad Regional Santa Fe- Universidad Tecnológica Nacional
Santa Fe – CP 3000 - Argentina
ppessolani@frsf.utn.edu.ar

ABSTRACT

MINIX4RT is an extension of the well-known MINIX Operating System that adds Hard Real-Time services in a new microkernel but keeping backward compatibility with standard MINIX versions.
Semaphores are the primitive synchronization and mutual exclusion mechanism in many operating systems, but MINIX does not provide those facilities. Semaphores were added to MINIX4RT, and since it is a Real-Time Operating System, they must meet some processing requirements such as dequeueing waiting processes in priority order and avoiding the Priority Inversion problem. This article describes the Real-Time Semaphores facilities available on MINIX4RT, its design, implementation, performance tests and their results.
Keywords: Minix, Real-Time, Semaphores, Priority Inheritance, Priority Inversion.

1. INTRODUCTION

Real-Time Operating System (RTOS) services must consume limited and guaranteed amounts of time. That deterministic timing behavior is the key difference against time sharing Operating Systems (OS).
MINIX4RT (previously named RT-MINIXv2) [1, 2] is a Real-Time (RT) version of the well known MINIX 2.0.2 [3] Operating System designed to teach concepts on RT-programming, in particular, those related to RT-kernels; but it can be used as a serious system on resource-limited computers. It is a good tool for experimenting with novel OS policies and mechanisms, and for evaluating the impact of architectural innovations.
Design constraints for MINIX4RT were:

- *MINIX Compatibility:* All processes that run on MINIX must run on MINIX4RT 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. Reducing the source code changes keeps students´ experience to learn a MINIX-based RTOS. Most new codes must be added in separated functions with few changes in the original MINIX code. This restriction also helps with easier system updates for newer MINIX versions.
- *Source Code readability*: As MINIX4RT is focused on academic uses, its source code must be easily understood, perhaps sacrificing performance.

MINIX uses message passing as its central paradigm because it has a Client/Server microkernel based architecture. Messages have fixed sizes and a strict copy to value semantics. In OS without Virtual Memory as MINIX, a message transfer implies a copy of the message from the sender's process address space to the destination's process address space. Since the copy is a time-consuming operation, it reduces the performance of simple synchronization or mutual exclusion primitives. Semaphores have a lower performance cost because they do not need that copy. Furthermore, as every operation in a hard RTOS, MINIX4RT RT-Semaphore primitives need to have deterministic execution and blocking times.

The rest of this work is organized as follows. Section 2 introduces MINIX4RT. Section 3 presents background information about synchronization and mutual exclusion primitives on MINIX. Section 4 presents the proposed RT-Semaphore model. Section 5 provides information about RT-Semaphores basic data structures. Section 6 describes kernel primitives to operate on RT-Semaphores. A performance evaluation is provided in Section 7. Finally, Section 8 presents conclusions and future works.

2. OVERVIEW OF MINIX4RT

MINIX4RT provides the capability of running Real-Time and Non Real-Time (NRT) processes on the same machine [1]. RT-processes are executed when necessary regardless of what MINIX is doing.
The RT-microkernel works by treating the MINIX OS kernel as a task being executed under a small RTOS based on software emulation of interrupt control hardware. In fact, MINIX is like the idle process for the RT-microkernel being executed only when there are no RT-processes to run. When MINIX requests the hardware to disable interrupts, the RT-microkernel intercepts that request, records it, and returns to MINIX. If one of those “disabled” interrupts occurs, the RT-microkernel records its occurrence and returns without executing the MINIX interrupt handler. Later, when MINIX requests the hardware to enable interrupts, the RT-microkernel intercepts that request and executes all previously "disabled" handlers with recorded interrupts. This emulation prevents MINIX from disabling RT-interrupts imposing long latencies to the execution of RT-interrupt service routines and RT-processes.
The major features of MINIX4RT are summarized as follows:

- *Layered Architecture*: MINIX4RT has a layered architecture that helps to change a component without affecting the others [1].
- *Real-Time Sub-kernel*: An RT-microkernel that deals with interrupts, Interprocess Communications (IPC), time management and scheduling is installed below MINIX kernel. The advantages of using a microkernel for RTOS are a better preemptability, a smaller kernel size, and an easier addition/removal of services [1].
- *Timer/Event Driven Interrupt Management*: Device Driver writers can choice between two strategies of RT-Interrupt management [1].
- *Fixed Priority Hardware Interrupt Processing*: A priority can be assigned to each hardware interrupt that let them be serviced in priority order [1].
- *Two Stages Interrupt Handling*: Interrupt can be serviced in two stages. The hardware interrupt handler (inside interrupt time) performs the first part of the needed work and a software Interrupt handler (outside interrupt time) does the remaining work [1].
- *Fixed Priority Real-Time Scheduling*: Each process has an assigned priority. The RT-kernel schedules them in priority order with preemption [2].
- *Periodic and Non-Periodic RT-processing*: A period can be specified for a periodic process; the RT-microkernel schedules it on period expiration [2].

- *Process and Interrupt Handlers Deadline Expiration Watchdogs*: The use of watchdog processes is a common use strategy to deal with malfunctioning RT-processes. When a process does not perform its regular function in a specified time (deadline) another process (watchdog) is signaled to take corrective actions [2].
- *Timer Resolution Management Detached from MINIX Timer:* A Timer interrupt of 50 Hz is emulated for the MINIX kernel even though the hardware Timer interrupt has a higher frequency [4].
- *Software Timers:* There are system facilities named Virtual Timers (VT) used for time-related purposes as alarms, timeouts, periodic processing, etc. One particular feature of MINIX4RT is that it handles software timer actions in priority order [4].
- *Real-Time Interprocess Communications:* MINIX4RT IPC uses unidirectional communication channels called Message Queues that handle messages in priority order and guarantee message delivery in a timely fashion and avoid the Priority Inversion problem [5].
- *Statistics and Real-Time Metrics:* There are several facilities to gather information about the system status and performance.

Only NRT-processes can be created and terminated under MINIX4RT. 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 *mrt_set2rt()* System Call. Therefore a RT-process is managed by the RT-kernel and blocked for the MINIX kernel; and a NRT-process is managed by the MINIX kernel and blocked for the RT-kernel. Before converting a process, several parameters (such as priority, period, watchdog process, etc.) must be passed to the RT-kernel using the *mrt_setpattr()* System Call.

3. Synchronization And Mutual Exclusion On Minix

Rendezvous Message Transfer is the basic mechanism that MINIX uses to communicate, synchronize and make mutual exclusion among Tasks, Servers and Users' processes and to notify hardware interrupt occurrence.
Those primitives are implemented as the following kernel functions[3]:

- *mini_send(caller, destination, msg):* If the destination process is blocked waiting for that message from the *caller*, the message is copied from the *caller's* message buffer pointed by *msg* to the *destination's* message buffer, otherwise the *caller* process is blocked.
- *mini_rec(caller, sender, msg):* If the *sender* process is blocked trying to send a message to the *caller* process, the message is copied from the *sender's* buffer to the buffer pointed by *msg,* and the *sender* process is unblocked, otherwise the *caller* process is blocked.

4. MINIX4RT Semaphore Model

A semaphore is a kernel object that one or more processes can acquire or release for synchronization or mutual exclusion purposes. They constitute the classic method for restricting access to shared resources in a multiprogramming environment. In a RT-environment, semaphore operations need to have deterministic execution and blocking times.
MINIX4RT RT-Semaphores are implemented inside the RT-microkernel and do not use any MINIX IPC primitives because:

- *mini_send()* and *mini_rec()* kernel functions could change the caller's RT-process to a READY state for the MINIX kernel. It would be therefore selected to be executed by its NRT-scheduler ignoring all its RT-execution attributes.
- If an RT-process makes a request to a NRT-process using *mini_send()*, the RT-process must wait for the reply from the NRT-process running at NRT-priority. This behavior could produce/cause an Unbounded Priority Inversion (explained in Section 6).

In the same way, RT-processes are inhibited from making any MINIX System Calls (except *exit()*) due to the use of MINIX IPC primitives. For this reason, MINIX4RT offers two sets of facilities:

- *System Calls*: To be used by NRT-processes to set the RT-environment or to get RT-statistics. These System Calls use MINIX primitives and do not have timing constraints.
- *Kernel Calls*: To be used by RT-processes to provide RT-services. These Kernel Calls do not use MINIX primitives and do have timing constraints.

MINIX4RT Semaphores have the following features:

- Configurable dequeueing policy (Priority order or FIFO order).
- Basic Priority Inheritance Protocol (BPIP) support to avoid Unbounded Priority Inversion [6].
- Statistical counters of ups (also known as signal) and downs (also known as wait) operations on the semaphore.
- Timeout support.

To eliminate the allocation delay, the RT-kernel reserves a memory space (called the System Semaphore Pool) where semaphore objects are stored.

5. RT-Semaphore Data Structures

MINIX4RT defines new data structures to operate with RT-Semaphores. It defines RT-kernel data structures and User-space data structures as described in the following sections.

RT-Semaphore Kernel Data Structure

The RT-microkernel defines an RT-Semaphore Descriptor data structure that has the following fields and data type definition:

```
struct MRT_sem_s {
  int index;   /* semaphore ID            */
  int value;   /* semaphore Value         */
  priority_t  priority;
               /* Ceiling priority        */
  unsigned int flags;
               /* semaphore policy flags  */
  int owner;   /* semaphore owner         */
  long ups;    /*  #  of sem up() calls   */
  long  downs; /*  #  of sem down() calls */
  MRT_proc_t  *carrier;
              /* the process that has     */
              /* locked a mutex semaphore */
  link_t  alloclk; /* Allocated list link */
  link_t  locklk;  /* Locked list link    */
  char  name[MAXPNAME]; /* semaphore name */
  plist_t plist;
              /* Priority List of waiting */
              /* processes                */
  };
typedef struct MRT_sem_s MRT_sem_t;
```

• *index*: Identifies the Semaphore Descriptor into the System Semaphore Pool.
• *value*: The semaphore value that can be set by the *mrt_semalloc()* System Call. It is increased by one for each *mrt_semup()* System Call or *MRT_semup()* RT-Kernel Call. It is decreased by one for each *mrt_semdown()* System Call or *MRT_semdown()* RT-Kernel Call.
• *priority*: The ceiling priority used by the Priority Ceiling Protocol and the Semaphore Inheritance Protocol not implemented in the current version.
• *flags*: RT-Semaphore policy flags. It is an OR of the following bits:
- SEM_PRTYORDER: If it is set, the waiting RT-processes will be woken up in priority order, otherwise they will be woken up in First Come First Served (FCFS) order.

- SEM_MUTEX: If it is set, the RT-Semaphore will be used as a mutex, otherwise it will be a counting RT-semaphore.
- SEM_PRTYINHERIT: If it is set, the RT-kernel applies the Basic Priority Inheritance Protocol to RT-Semaphore operations. This option is valid only if the SEM_PRTYORDER and the SEM_MUTEX bits are set.

• *owner*: The process which makes the *mrt_semalloc()* System Call.

• *ups* and *downs*: Statistical counters of *MRT_semup()* and *MRT_semdown()* RT-kernel calls since the RT-Semaphore allocation.

• *carrier*: The process that has locked the mutex RT-Semaphore.

• *alloclk*: A data structure to build a linked list of allocated RT-Semaphores. It is also used to insert/remove a RT-Semaphore into/from the Free list of the System Semaphore Pool.

• *locklk*: A data structure to build a linked list of RT-Semaphores locked by a RT-process.

• *name*: A name assigned to the RT-Semaphore.

• *plist*: A data structure to build a priority list of waiting RT-processes.

RT-Semaphore Userspace Data Structure

MINIX4RT defines several Userspace Data Structures to operate on RT-Semaphores as described in the following sections.

RT-Semaphore Attributes Data Structure: The fields of RT-Semaphore Attributes data structure have the same meanings of the RT-Semaphore Descriptor data structure. It is used by the *mrt_semalloc()* and the *mrt_semattr()* system calls.

```
struct mrt_semattr_s {
  int value; /* semaphore Value            */
  unsigned int flags;
          /* semaphore policy/status flags */
  priority_t  priority;
          /* Ceiling priority              */
          /* for future uses               */
  char name[MAXPNAME]; /* semaphore name    */
    };
typedef struct mrt_semattr_s mrt_sematr_t;
```

RT-Semaphore Statistics Data Structure: This data structure is used to get RT-Semaphore statistics. It is used by the *mrt_semstat()* system call.

```
struct mrt_semstat_s {
  long ups; /* # of sem up() calls         */
  long downs; /* # of sem down() calls     */
  int     maxinQ;
          /* maximum # of enqueued processes */
    };
typedef struct mrt_semstat_s mrt_semstat_t;
```

• *ups and downs*: Statistical counters of *mrt_semup()* and *mrt_semdown()* system calls since the RT-Semaphore allocation.

• *maxinQ:* The maximum number of waiting RT-processes enqueued into the RT-Semaphore list.

RT-Semaphore Internal Data Structure: This data structure is used to get the internal status of a RT-Semaphore. It is used by the *mrt_semint()* system call.

```
struct mrt_semint_s {
 int      index;  /* semaphore ID      */
 int      owner;  /* semaphore owner     */
 int      inQ;    /* # of process enqueued   */
  };
typedef struct mrt_semint_s mrt_semint_t;
```

• *index*: It identifies the Semaphore Descriptor into the System Semaphore Pool.

• *owner*: The process which makes the *mrt_semalloc()* System Call.

RT-Semaphore Down Data Structure: This data structure is used by the *mrt_semdown()* Kernel Call.

```
struct mrt_down_s {
  int index;          /* semaphore ID      */
  lcounter_t  timeout; /* timeout in ticks  */
    };
typedef struct mrt_down_s mrt_down_t;
```

• *index*: The identification of the RT-Semaphore.

• *timeout*: A timeout in Timer ticks can be specified to wait for the request RT-Semaphore.

RT-Semaphore Waiting RT-Processes Priority List

To manage the waiting RT-Processes on a RT-Semaphore, the RT-kernel uses a Priority List Data Structure (see Figure 1):

On insertion operations, the priority-th bit in the bitmap is set and the Process Descriptor is appended to the Priority List in accordance with its priority field.

Thus, finding the highest priority RT-process in the priority list is just a matter of finding the most significant bit set in the bitmap. Since the number of priorities is fixed, the time to complete a search is constant and unaffected by the number of RT-processes in the Priority List.

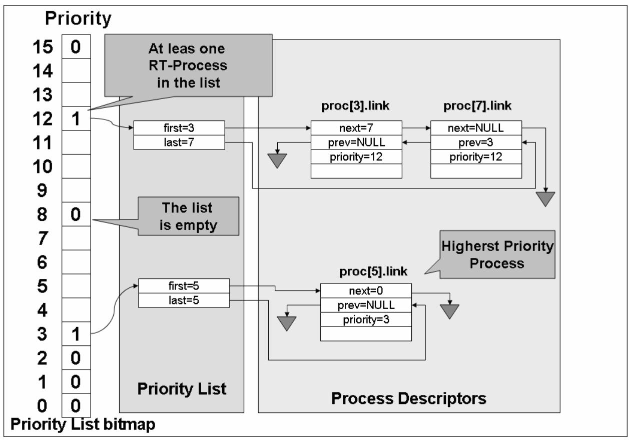

Figure 1: RT-Semaphore Waiting RT-Processes Priority List

6. RT-SEMAPHORES KERNEL CALLS

In many RT-applications, there are resources that must be shared among processes so as to prevent more than one process from using the resource at the same time (mutual exclusion).

The Unbounded Priority Inversion problem is an undesired situation where a higher priority process waits for a semaphore locked by a lower priority process and a medium priority process preempts it delaying the semaphore release and therefore the high priority process execution.

Many mechanisms have been developed to avoid it. Sha, Rajkumar and Lehosky [7] suggest two protocols to avoid the Unbounded Priority Inversion problem. They are the Basic Priority Inheritance Protocol (BPIP) and the Priority Ceiling Protocol (PCP).

Under the BPIP, if a lower priority process blocks a higher priority process, the lower priority process inherits the priority of the higher priority process for the duration of its critical section. The BPIP potentially requires priorities to be modified when processes try to lock a locked semaphore. The process that has locked the requested semaphore may inherit the highest priority among the petitioner's priorities. To achieve the correct behavior and to be in compliance with BPIP, priority inheritance needs to be a transitive operation. Therefore, the RT-kernel must search across the chain of petitioner processes, applying the priority inheritance until it finds the process that has no pending requests. MINIX4RT provides RT-Semaphore primitives that are in compliance with the BPIP, offering a deterministic timing behavior.

***mrt_semdown()* Kernel Call**: The *mrt_semdown()* Kernel Call decreases the semaphore's count by one. If the resulting semaphore value drops below zero, the caller process will be blocked and its descriptor is inserted into the RT-Semaphore Waiting RT-Processes Priority List.
For RT-Semaphores used as mutexes, the process that has locked the RT-semaphore increases its priority to the caller's priority, if it is higher than its owns. If that RT-process is blocked waiting for another RT-semaphore, the Priority Inheritance Protocol is applied to all RT-processes in the chain.
A timeout in Timer ticks can be specified to wait for the RT-Semaphore release. A special value of MRT_NOWAIT can be specified to return without waiting if the RT-semaphore is locked by other RT-process. MRT_FOREVER must be specified as timeout to wait for the RT-Semaphore release. On timeout expiration:

- The RT-process descriptor is removed from RT-Semaphore Waiting RT-Processes Priority List.
- The caller is unblocked returning an E_MRT_TIMEOUT error code.

For RT-Semaphores used as mutexes, the priority of the RT-process that had locked the RT-Semaphore is set to the highest priority waiting process into RT-Semaphore Waiting RT-Processes Priority List or its base priority specified in the *MRT_setpattr()* System Call.

***mrt_semup()* Kernel Call**: If the semaphore value is lower than zero, its absolute value indicates the number of waiting RT-processes blocked trying to down the semaphore. The *mrt_semup()* Kernel Call increases the semaphore's count by one, removes the highest priority process (if the SEM_PRTYORDER bit is set in flags) or the first process into RT-Semaphore Waiting RT-Processes Priority List and unblocks it.
For RT-Semaphores used as mutexes, if the BPIP had raised the caller's priority when it locked the semaphore, its priority is returned to its base priority specified in the *MRT_setpattr()* System Call.

7. PERFORMANCE EVALUATION

This section describes the tests performed on MINIX4RT Semaphores and their results. The RT-Semaphore operations performance was tested with four kinds of system setups/policies (see Table 1), with and without timeout settings and with and without applying BPIP. The tests consist of 10000 rounds of the Producer/Consumer algorithm (two down operations and two up operations per process per round).

Table 1: Setups and Policies of Semaphore Operations Performance Tests

Test Name	With Timeout	Priority List/FIFO	Priority Inheritance
TEST1	No	Priority List	No BPIP
TEST2	Yes	Priority List	No BPIP
TEST3	No	Priority List	BPIP
TEST4	Yes	Priority List	BPIP

The tests were performed under different kinds of loads on the tested system (see Figure 2):

1. *Without Load (NOLoad):* All unneeded processes are killed before the test.
2. *CPU Load (CPULoad):* A NRT-script loads the CPU without any I/O operation.
3. *I/O Disk Load (HDLoad):* A NRT-process access files on the hard disk.
4. *I/O RS232e Load (RSLoad):* A NRT-file transfer over the serial port at 19200 Kbps.

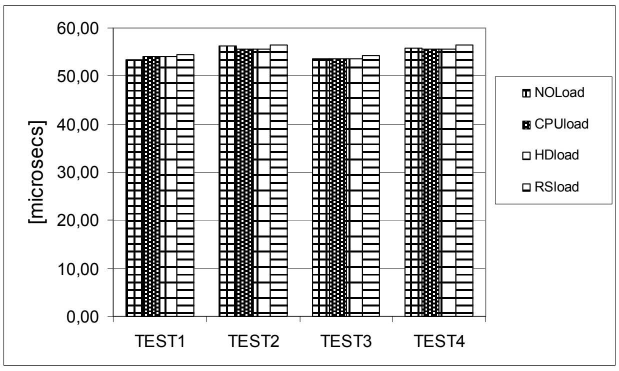

Figure 2: Down-Up pair processing time.

Table 2 presents Down-Up pair processing times.

Table 2: Down-Up Pair Processing Times [µs]

	TEST1	TEST2	TEST3	TEST4
NOLoad	53,42	56,25	53,52	55,75
CPUload	54,02	55,67	53,57	55,67
HDload	54,05	55,65	53,57	55,57
RSload	54,45	56,42	54,32	56,45

All the tests were carried out with the Programmable Interval Timer set up at 1000[Hz]. This fact implies the execution of the Timer Interrupt Service Routine 1000 times per second adding a significant overhead to the measurements, but it presents a more realistic scenario. Other tests performed on MINIX4RT present an average Timer Interrupt Service Time of 32[µs].
The equipment used for those tests was an IBM Model 370C Notebook, Intel® DX4 75 MHz, AT Bus, Memory 8 MB, and MINIX4RT (Kernel 12052007). Even though the equipment is quite old, it allows for performance comparisons against reports of other systems with similar hardware.
Sacha [8] reports QNX signal times about 40-45[µs] on a 486/66 MHz. His results show the same order of magnitude than the tests results on MINIX4RT, considering that they include down time plus up time and the CPU clock difference.

8. CONCLUSIONS AND FUTURE WORKS

MINIX has proved to be a feasible test-bed for OS development and extensions that could be easily added. In a similar way, MINIX4RT has an architecture that can be used as a starting point for adding RT-services. Even though it was designed for an academic environment, it can be optimized for production systems even in embedded systems. MINIX4RT combines Hard Real-Time with the standard MINIX platform, so that time sensitive control algorithms can operate together with background processing without worrying about interference.
MINIX4RT algorithms were developed to minimize priority inversion to meet applications with strict timing constraints. A sample of this is the use of Priority Lists and the use of the Basic Priority Inheritance Protocol.
The RT-microkernel has basic features as Interrupt Management, Process Management, Time Management, RT-IPC and Statistics gathering, making it a good choice to conduct coding experiences in Real-Time Operating Systems courses.
Near future works on MINIX4RT are:

- *Operating System Profiling:* Runtime profiling is a key technique to prove new concepts, debug problems, and optimize performance.
- *Port Real-time to MINIX3:* The current version of MINIX has a more strict compliance with a Client/Server microkernel based Operating System. Those changes bring about the need of rewriting some components of MINIX4RT code to enable running under MINIX3.

9. ACKNOWLEDGEMENTS

The author gratefully acknowledges help received from María Inés Fidalgo for the language revision of this article.

10. REFERENCES

[1] Pessolani, Pablo A, "RT-MINIXv2: Architecture and Interrupt Handling", 5th Argentine Symposium on Computing Technology, 2004.
[2] Pessolani, Pablo A., "RT-MINIXv2: Real-Time Process Management and Scheduling", 6th Argentine Symposium on Computing Technology, 2005.
[3] Tanenbaum Andrew S., Woodhull Albert S., "Sistemas Operativos: Diseño e Implementación" 2da Edición, ISBN 9701701658, Editorial Prentice-Hall , 1999.
[4] Pessolani, Pablo A., "MINIX4RT: Time Management and Timer Facilities", 7th Argentine Symposium on Computing Technology, 2006.
[5] Pessolani, Pablo A., "MINIX4RT: Real-Time Interprocess Communications Facilities", Workshop de Arquitecturas, Redes y Sistemas Operativos, XII Congreso Argentino de Ciencias de la Computación, 2006.
[6] Mark W. Borger, Ragunathan Rajkumar. "Implementing Priority Inheritance Algorithms in an Ada Runtime System", Technical Remailbox . CMU/SEI-89-TR-15. ESD-TR-89-23. Software Engineering Institute Carnegie Mellon University, 1989.
[7] Sha, L., Lehoczky, J.P., and Rajkumar, R. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". Tech. Rept. CMU-CS-87-181, Carnegie Mellon University, Computer Science Department, 1987.
[8] Krzysztof M. Sacha, "Measuring the Real-Time Operating System Performance", Institute of Control and Computation Engineering, Warsaw University of Technology, Poland, 1995.

Received: Jun. 2008. Accepted: Aug. 2008.