Inversión de prioridades: prueba de concepto y análisis de soluciones

Raúl Benencia, Luciano Iglesias, Fernando Romero y Fernando G. Tinetti**

Instituto de Investigación en Informática III-LIDI
Facultad de Informática, UNLP
rbenencia@linti.unlp.edu.ar,li@info.unlp.edu.ar,
{fromero,fernando}@lidi.info.unlp.edu.ar,
http://weblidi.info.unlp.edu.ar

Resumen La planificación de tareas es el punto crucial de un sistema de tiempo real. Dicha función es llevada a cabo por el planificador del sistema operativo, diseñado para poder cumplir con las restricciones temporales de dichas tareas, teniendo en cuenta sus valores de prioridad. Al haber recursos compartidos por estas tareas, se produce el efecto llamado inversión de prioridades. En este trabajo se analiza dicho efecto y se evalúan las soluciones implementadas para este problema en el Sistema Operativo de Tiempo Real GNU/Linux con parche RT-PREEMPT.

Keywords: Planificación de tareas de tiempo real, inversión de prioridades, GNU/Linux con parche RT-PREEMPT, Sistemas Operativos de Tiempo Real

1. Introducción

Los sistemas de tiempo real requieren ser correctos lógica y temporalmente. Se deben respetar las restricciones de tiempo en la ejecución de las tareas. De acuerdo a la función que realizan, las tareas pueden requerir plazos estrictos o plazos más relajados. Esto conlleva a la necesidad de una planificación basada en prioridades fijas, de manera de asegurar los límites estrictos. Este tipo de planificación es respecto de uno de los recursos compartidos por las diversas tareas, implicadas en la ejecución, la CPU. De tal manera que si un proceso que implementa una tarea de baja prioridad está usando la CPU y otro proceso con una prioridad mayor requiere su uso, está previsto el desalojo del proceso de menor prioridad para asignársela al de mayor prioridad. El resto de los recursos compartidos por las tareas suele planificarse por demanda (FCFS). Al producirse situaciones de bloqueo de estos recursos, que requieren usarse en forma exclusiva (región crítica), puede pasar que el proceso de mayor prioridad quede relegado por uno de menor prioridad. Esta situación se denomina *inversión de prioridades* [2] y se puede agravar si procesos de prioridad intermedia logran hacerse de la CPU antes de que la tarea de prioridad baja libere el recurso que espera la de

** Investigador CICPBA

alta prioridad. Con lo cual la duración de la inversión de prioridades puede ser ilimitada y no permitir a la tarea de mayor prioridad cumplir con sus restricciones temporales. Los Sistemas Operativos de Tiempo Real suelen contar con mecanismos para atenuar este problema. En rigor, estos mecanismos no evitan la inversión de prioridades, solo la inversión de prioridades ilimitada.

2. Descripción del problema y soluciones existentes

Los Sistemas de Tiempo Real suelen realizar tareas con diferente nivel de urgencia. Si bien todas las tareas pueden tener restricciones temporales para su ejecución, estas restricciones pueden ser estrictas (hard real-time) o menos estrictas (soft real-time) [3]. Esto es manejado por los Sistemas Operativos de Tiempo Real de diferentes maneras. Esto se conoce como planificación de tareas. Uno de los métodos de planificación de tareas se realiza asignando diferentes niveles de prioridad, de tal manera que si se debe incumplir una restricción temporal sea de las tareas con menor nivel de prioridad. A su vez, estas tareas suelen estar implementadas por threads, que se caracterizan por compartir memoria y otros recursos. Esto genera un problema llamado inversión de prioridades, que se describe a continuación: considérese un proceso con prioridad alta, el proceso H, y otro con prioridad baja, el proceso L. Ambos procesos interactúan con el recurso r y, para evitar inconsistencias, deben proteger sus respectivas secciones críticas con un mutex, por ejemplo. Si el proceso H intenta utilizar r mientras L mantiene un lock sobre el mismo, entonces H no podrá continuar su tarea hasta que L no libere el recurso. Hasta aquí se plantea una situación normal. El procedimiento correcto a seguir es asignar el procesador a L para que libere a r lo más pronto posible, para que luego H pueda adquirir el lock sobre el recurso y así continuar su tarea. Sin embargo, el problema de la inversión de prioridades se presenta cuando H espera a que L libere el lock, y mientras L intenta liberar el recurso, el mismo es interrumpido por un proceso de prioridad superior a L pero inferior a H. De esta forma, tanto el proceso L como el proceso H se ven rezagados por el proceso de prioridad media M. En algunos sistemas la inversión de prioridades puede pasar desapercibida puesto que, a pesar de las demoras, las restricciones de tiempo se cumplen y por lo tanto el sistema de tiempo real no falla. Sin embargo, existen numerosas situaciones donde la inversión de prioridades puede causar problemas críticos. Si un proceso de prioridad alta entra en estado de inanición de los recursos que precisa, puede provocar una falla en el sistema que active medidas correctivas, como un watch-dog que reinicie por completo todo el sistema.

2.1. El problema en un caso real: El Rover enviado a Marte

Tal vez el caso real más conocido de inversión de prioridades fue el que ocurrió con el rover que se envió al planeta Marte en la misión Mars Pathfinder [4] [5]. La misión Mars Pathfinder fue catalogada como perfecta a los pocos días de su aterrizaje en la superficie marciana, el 4 de julio del año 1997. Durante

varios días el rover envió cantidades voluminosas de datos, tales como imágenes panorámicas. Sin embargo, luego de pocos días de cumplir con las solicitudes requeridas desde el planeta Tierra, y no mucho después de que el rover comenzara a recolectar datos meteorológicos, el sistema operativo del robot comenzó a reiniciarse continuamente ocasionando severas pérdidas de datos. El problema se encontraba en la administración de las prioridades de VxWorks, el kernel de tiempo real embebido que usaba el Pathfinder. El planificador de VxWorks utilizaba apropiación por prioridades entre los threads. Las tareas en el rover eran ejecutadas como threads. La asignación de prioridades a los mismos se calculaba reflejando la urgencia relativa de dichas tareas. Además, el Pathfinder contenía un bus de información, similar a un área de memoria compartida, utilizado para comunicar información entre los distintos componentes de la nave. El kernel VxWorks proveía una tarea de alta prioridad dedicada a la administración de este bus, cuya función era mover ciertos tipos de datos desde y hacia el mencionado canal. La recolección de datos meteorológicos se realizaba con poca frecuencia en un thread de baja prioridad, y los datos adquiridos se distribuían utilizando el bus de información. Cuando los datos se distribuían por el bus, se adquiría el lock sobre un mutex, se escribía sobre el bus, y se liberaba el lock. Si una interrupción causaba la apropiación de la CPU mientras el thread de baja prioridad mantenía el bloqueo sobre el bus, y el thread de alta prioridad que administraba dicho bus intentaba adquirir el mismo mutex con el objetivo de recibir estos datos, entonces dicho thread se bloqueaba hasta que el thread de baja prioridad liberase el mencionado mutex. Finalmente, el rover también contenía una tarea dedicada a la comunicación que corría sobre un thread con prioridad media. La mayor parte del tiempo esta combinación de threads funcionaba correctamente. Sin embargo, con muy poca frecuencia ocurría que el thread de comunicaciones, de prioridad media, era interrumpido durante un corto intervalo de tiempo cuando el thread dedicado a la administración del bus, de prioridad alta, era bloqueado para esperar por los datos meteorológicos provistos por el thread de prioridad baja. Cuando se daba esta situación, el thread dedicado a las comunicaciones impedía que el thread de baja prioridad pueda ejecutarse para liberar el mutex. Consecuentemente, el thread de administración del bus era efectivamente bloqueado por una tarea de menor prioridad, provocando de esta forma la inversión de prioridades. Luego de un tiempo prudencial, un watch-dog timer del rover notaba el desperfecto y concluía que algo había dejado de funcionar correctamente, provocando un reinicio total del sistema operativo [6].

2.2. Soluciones existentes

Se han propuesto diversas soluciones (protocolos) para el problema de la inversión de prioridades, acá se mencionan las principales [7]:

1. Herencia de prioridad (Priority inheritance)
2. Techo de prioridad (Priority ceiling)
3. Techo de prioridad inmediato (Inmediate priority ceiling)
4. Enmascaramiento de interrupciones

5. Incremento aleatorio
6. Basado en la restauración del recurso (Shadowing)

Herencia de prioridad La solución denominada herencia de prioridad propone eliminar la inversión de prioridades elevando la prioridad de un proceso con un lock en un recurso compartido al máximo de las prioridades de los procesos que estén esperando dicho recurso. La idea de este protocolo consiste en que cuando un proceso bloquea indirectamente a procesos de más alta prioridad, la prioridad original es ignorada y ejecuta la sección crítica correspondiente con la prioridad más alta de los procesos que está bloqueando. Por ejemplo, vuélvanse a considerar los procesos L, M y H de prioridad baja, media y alta respectivamente. Suponer que H está bloqueado esperando a que L libere un lock sobre un recurso compartido. El protocolo de herencia de prioridad, entonces, requiere que L ejecute su sección crítica con la prioridad de H. De esta forma, M no podrá apropiarse del procesador cuando L lo tenga en uso. Por lo tanto M, que es un proceso con más prioridad que L, deberá esperar a que L ejecute su sección crítica, ya que L hereda su prioridad de H. Luego, cuando L termina de ejecutar su sección crítica, su prioridad vuelve a la normalidad y el proceso H es despertado. H, que tiene mayor prioridad que M, se apropia del procesador y se ejecuta hasta terminar. Finalmente, cuando H termina su ejecución prosigue el proceso M, y por último L termina su ejecución. El kernel Linux implementa la herencia de prioridades mediante un sencillo mecanismo que se basa en prohibir la apropiación del procesador mientras se esté ejecutando código del kernel protegido por un spinlock. Las desventajas que presenta este método son [8]:

1. Este método puede causar más cambios de contexto que el de techo de prioridad
2. El anidamiento de secciones críticas protegidas por herencia de prioridad puede producir grandes demoras debido a que cada vez que se cambia una prioridad de una tarea debe ejecutarse el planificador
3. El método falla si se mezclan tareas con y sin herencia de prioridad
4. El peor caso de herencia es peor que otras soluciones al problema
5. Según algunos autores, la mayoría de las implementaciones de herencia de prioridad tienden a complicar el código de las secciones críticas, reduciendo finalmente el desempeño del sistema [9]

Techo de prioridad La solución denominada techo de prioridad es un protocolo que elimina la inversión de prioridades mediante la asignación predefinida de un techo de prioridad a cada recurso. Cuando un proceso adquiere un recurso compartido, la prioridad de dicho proceso se eleva temporalmente al techo de prioridad del mencionado recurso. El techo de prioridad debe ser más alto que la prioridad de todos los procesos que puedan acceder al recurso compartido. De esta forma, cuando un proceso se esté ejecutando con el techo de prioridad de un recurso, el procesador no podrá ser apropiado por otro proceso que quiera acceder al mismo recurso, puesto que todos tendrán menor prioridad [10]. Las desventajas que tiene este método son:

1. Se debe realizar un análisis estático de la aplicación para determinar cuáles serán los techos de prioridad de cada recurso compartido. Para realizar este análisis, todas las tareas que accedan a recursos compartidos deben conocerse de antemano. Esto puede ser difícil, o incluso imposible de determinar en una aplicación compleja [10]
2. La prioridad de los procesos aumenta y disminuye cada vez que se accede a un recurso compartido, aún cuando no haya procesos compitiendo por el mismo
3. Puede dar un bloqueo falso de threads [11]

Esta solución fue propuesta por primera vez en 1980, en uno de los primeros papers que describió el problema de la inversión de prioridades [12].

Techo de prioridad inmediato Es un derivado del Protocolo de Techo de Prioridad. En este protocolo, la tarea que accede a un recurso hereda inmediatamente el techo de prioridad del recurso. Este protocolo es mas fácil de implementar y es más eficiente (hay menos cambios de contexto) [13].

Enmascaramiento de interrupciones El enmascaramiento de interrupciones también se puede utilizar para evitar la inversión de prioridades. En este caso, las interrupciones se enmascaran cuando un proceso entra en una sección crítica, y se vuelven a habilitar cuando sale de la misma. De esta forma, la inversión de prioridades no puede ocurrir puesto que todas las secciones críticas se ejecutan sin ser interrumpidas por procesos de mayor prioridad. Para que este mecanismo funcione correctamente, todas las interrupciones deben estar deshabilitadas. Si sólo algunas se enmascaran, entonces la inversión de prioridades podrá ser re-introducida por el mecanismo de gestión de interrupciones del hardware subyacente. Esta solución al problema de inversión de prioridad suele ser encontrada en sistemas embebidos, debido a su confiabilidad, bajo consumo de recursos y sencillez en su implementación.

Las desventajas de este método son:

1. Requiere que las secciones críticas sean escasas y cortas, puesto que todo el sistema se ve bloqueado mientras un proceso se encuentra en una de ellas.
2. Mientras las interrupciones estén completamente enmascaradas los fallos de página no podrán ser atendidos [12]. Por este motivo, se desaconseja la implementación de esta solución en sistemas de propósito general.

Incremento aleatorio La solución denominada incremento aleatorio propone eliminar la inversión de prioridades mediante el incremento de la prioridad de los procesos de baja prioridad que contengan locks sobre recursos compartidos. La elección del proceso cuya prioridad será incrementada se realiza de forma aleatoria, de ahí el nombre de la técnica. El incremento de la prioridad se mantiene hasta que el lock sea liberado. Esta técnica es utilizada en sistemas operativos Microsoft Windows [14]

Basado en la restauración del recurso Se basa en la técnica de desalojo de la tarea de menor prioridad que toma el recurso al arribar una tarea de mayor prioridad. Hay dos variantes: mantener un log de lo actuado por el de menor prioridad en el recurso y deshacer los cambios de manera inversa o que la tarea de menor prioridad actúe sobre una copia del recurso, que reemplaza al recurso si la tarea de baja prioridad finaliza antes del arribo de la tarea de mayor prioridad. En caso contrario, la tarea de menor prioridad es desalojada, y el recurso aparece inalterado, como si no se hubiera ejecutado la tarea de menor prioridad. Las demoras las sufre la tarea de menor prioridad en este método.

Cabe aclarar que los sistemas operativos de tiempo real, implementan solo algunos de estos métodos:

- *FreeRTOS*: es un mini kernel de tiempo real diseñado para sistemas embebidos, preparado para funcionar en diferentes plataformas de microcontroladores. Implementa el protocolo de herencia de prioridad [15].
- *MarteOS*: es un sistema mínimo de tiempo real, basado en ADA, desarrollado por el Grupo de Computadoras y Tiempo Real de la Universidad de Cantabria. Implementa herencia de prioridad y techo de prioridad [16].
- *QNX*: es un micro-kernel de tiempo real de alto desempeño [17]. Implementa la herencia de prioridad.
- *RTAI*: Es un sistema basado en Linux que le agrega funcionalidad de tiempo Real. Este sistema es desarrollado por el Politecnico di Milano - Dipartimento di Ingeneria Aerospaziale (DIAPM). Implementa Herencia de Prioridad [18].
- *RTLinux*: las prioridades de las tareas son estáticas manejadas con dos variantes de algoritmo: FIFO o Round Robin [19]. Si bien su creador, Victor Yodaiken dice que RTLinux no soporta la herencia de prioridad por la simple razón de que es incompatible con cualquier sistema de tiempo real confiable [20] existe una extension que da la posibilidad de usar el mecanismo de herencia de prioridad y el techo de prioridad.
- *VxWorks*: de la empresa WindRiverSystem, implementa la herencia de prioridad [21].
- Linux con parche RT-preempt: implementa herencia de prioridad y techo de prioridad.

3. GNU/Linux con parche RT-PREEMPT

GNU/Linux con parche RT-PREEMPT [1] es una modificación realizada al kernel Linux que lo convierte prácticamente en apropiativo, con la excepción de algunas pocas regiones de código pequeñas. Esta modificación permite la ejecución de aplicaciones consideradas *hard real-time*.

3.1. Caso de prueba

Como parte del presente trabajo se desarrolló una aplicación para recrear el fenómeno de la inversión de prioridad en GNU/Linux con parche RT-PREEMPT.

Tiene como objetivo evaluar las soluciones de herencia de prioridad y de techo de prioridad implementadas en el kernel. El desarrollo fue realizado en el lenguaje de programación C utilizando semáforos. La aplicación ejecuta tres tareas (*threads*) de las cuales dos precisan hacer uso de una región crítica en común. Dentro de la región crítica los distintos threads hacen simplemente unas instrucciones que consumen CPU. Cada thread tiene asignada una prioridad, baja (L), media (M) y alta (H). H y L, son los threads que requieren acceder a la región crítica compartida. La aplicación se puede ejecutar de tres maneras diferentes según el algoritmo que utiliza para evitar la inversión de prioridades:

- *inversion*: en este modo el programa se ejecuta sin prevenir la inversión de prioridades.
- *inheritance*: en este modo el programa evita la inversión de prioridades utilizando el protocolo de herencia de prioridad.
- *ceiling*: en este modo el programa evita la inversión de prioridades utilizando el protocolo de techo de prioridad.

Sincronización La inversión de prioridades se manifiesta cuando los threads del programa se ejecutan sobre un sistema con un único núcleo. Debido a que este caso de prueba es una simulación, los threads se deben sincronizar para que deliberadamente suceda la inversión de prioridades. Dicha sincronización se lleva a cabo usando tres semáforos a modo de barrera, con el fin de ordenar ejecución y la solicitud de acceso al recurso compartido de los mismos. La sincronización sucede de la siguiente forma, una vez que todos los threads fueron lanzados:

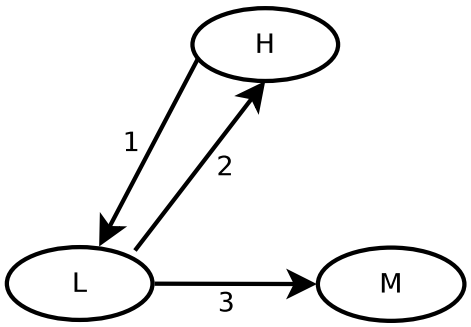

Figura 1. Se encuentran los tres threads representados por cada uno de los elipses y las flechas indican el orden y dirección de los avisos para reproducir el fenómeno de la inversión de prioridad.

1. El thread L espera la señal de que H para comenzar sus acciones. Luego de enviar la señal, H se bloquea esperando a que L bloquee el recurso compartido.

2. Luego de recibir la señal, L procede a bloquear el recurso compartido. Luego de bloquear el recurso compartido, L envía una señal a H informando este último evento.
3. Cuando H recibe la señal de que el recurso ya está bloqueado, procede a enviar una señal a M. Luego de enviar la correspondiente señal, H procede a intentar adquirir el recurso compartido.

4. Resultados

4.1. Tiempo de ejecución de los threads

La aplicación realizada para observar el fenómeno de la inversión de prioridad se ejecutó sobre un kernel Linux 3.2 con el parche PREEMPT RT [1] con un único procesador. La ejecución de *inversion* provocó la ejecución de los threads en el siguiente orden:

- H, hasta la solicitud del recurso compartido.
- M, hasta su finalización.
- L, hasta que liberó el recurso compartido.
- H, desde que se hizo del recurso compartido hasta el final.
- L, hasta su finalización.

Efectivamente se produjo la inversión de prioridades.

La ejecución de *inheritance* y de *ceiling* provocó la ejecución de los threads en el siguiente orden:

- H, hasta la solicitud del recurso compartido.
- L, hasta que liberó el recurso compartido.
- H, desde que se hizo del recurso compartido hasta el final.
- M, hasta su finalización.
- L, hasta su finalización.

En la tabla 1 pueden apreciarse los tiempos de ejecución de cada uno de los threads. Las entradas indicadas en la tabla no consideran el tiempo transcurrido entre que el thread L libera el recurso y termina su ejecución ya que no es de interés para observar el desempeño general del sistema. Como ya se mencionó estos dos mecanismos (herencia y techo de prioridades) no permiten garantizar las restricciones temporales que eventualmente podría tener la tarea de alta prioridad (H), ya que queda atada a la liberación del recurso compartido por parte de la tarea de baja prioridad (L), pero si evitan que tareas de una prioridad intermedia como M, se ejecuten antes que H.

4.2. Sobrecarga del Sistema Operativo

La implementación de cada una de la soluciones sobrecarga al sistema operativo con tareas que debe realizar al momento de adquirir o liberar un recurso.

	Inversion	Inheritance	Ceiling
L	6368	3164	3121
M	3199	9594	9550
H	9559	6400	6340

Cuadro 1. Tiempos de ejecución de cada uno de los threads, expresados en milisegundos.

Por este motivo, no utilizar ninguno de los protocolos que resuelve el problema sería como un caso base sin sobrecarga. La solución de herencia de prioridad en cada solicitud de bloqueo de un recurso compartido, debe analizar si el recurso está o no bloqueado por otro proceso de menor prioridad y de ser así pasarle la prioridad a este para que pueda finalizar lo antes posible. De la misma forma cuando libera el recurso debe analizar si tiene una prioridad heredada para restablecer su prioridad original. Por otro lado, la solución de techo de prioridades debe en cada solicitud de bloqueo determinar la prioridad techo entre todos los procesos que están a la espera del recurso y asignárselo al proceso que tiene el recurso para que este se ejecute inmediatamente. En el momento de la liberación del recurso debe asignar la prioridad techo al proceso de mayor prioridad de entre los que esperan por dicho recurso.

Se realizó en GNU/Linux con parche RT-PREEMPT un bloqueo de un semáforo[1]. Dicho bloqueo se realizó con la seguridad de que el semáforo no estaba bloqueado previamente, con la intención de hacer un primer análisis de la sobrecarga que cada algoritmo que soluciona la inversión de prioridad conlleva. Con respecto a la opción no impedir la inversión de prioridad, el bloqueo del semáforo con herencia de prioridad llevó un 47 % más de tiempo, mientras que usando el techo de prioridad llevó un 1555 % más de tiempo.

5. Conclusiones y líneas futuras

Con respecto a la ejecución de la prueba de concepto, mencionada en 4.1, y como ya se mencionó allí ni la solución por herencia de prioridades ni la solución de techo de prioridades garantizan el determinismo necesario en un sistema *hard real-time* ya que están ligados a la liberación del recurso compartido por parte de la tarea de baja prioridad. La solución de restauración del recurso si puede garantizar el determinismo necesario, penalizando a las tareas de menor prioridad a que tengan que desechar todo su trabajo con el recurso compartido.

Respecto de la sobrecarga del Sistema Operativo (4.2), sólo se analizó la situación con la garantía de que el recurso compartido estaba libre para poder ser bloqueado. El tiempo para el caso de techo de prioridades es mucho más alto que los demás, tal vez debido a la búsqueda inicial de la prioridad techo.

Como lineas futuras de investigación queda la opción de ver cuál es la sobrecarga cuando hay otras tareas en ejecución bloqueando el uso del recurso

[1] pthread_mutex_lock

compartido. También sería de interés analizar la sobrecarga que podría producir una implementación de restauración del recurso.

Referencias

1. RTwiki. http://rt.wiki.kernel.org/index.php/Main_Page.
2. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, September 1990.
3. Alan Burns and Andy Wellings. *Sistemas de tiempo real y lenguajes de programación*. Addison Wesley, Madrid [etc.], 2003.
4. Mars pathfinder. http://mars.jpl.nasa.gov/MPF/.
5. The risks digest volume 19: Issue 54. http://catless.ncl.ac.uk/Risks/19.54.html#subj6.
6. What really happened on mars? – authoritative account. http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html.
7. Tarek Helmy and Syed S. Jafri. Avoidance of priority inversion in real time systems based on resource restoration. *IJCSA*, 3(1):40–50, 2006.
8. Victor Yodaiken. Against priority inheritance, 2002.
9. Priority inheritance in the kernel [LWN.net]. http://lwn.net/Articles/178253/.
10. How to use priority inheritance | embedded. http://embedded.com/design/configurable-systems/4024970/How-to-use-priority-inheritance.
11. J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley. On using priority inheritance in real-time databases. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 210–221, 1991.
12. Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, February 1980.
13. Andreu Carminati, Rômulo Silva de Oliveira, and Luís Fernando Friedrich. Implementation and evaluation of the synchronization protocol immediate priority ceiling in PREEMPT-RT linux. *Journal of Software*, 7(3), March 2012.
14. Priority inversion (windows). http://msdn.microsoft.com/en-us/library/windows/desktop/ms684831(v=vs.85).aspx.
15. FreeRTOS - market leading RTOS (real time operating system) for embedded systems supporting 34 microcontroller architectures. http://www.freertos.org/.
16. MaRTE OS home page. http://marte.unican.es/.
17. QNX operating systems, development tools, and professional services for connected embedded systems. http://www.qnx.com/.
18. RTAI - official website. https://www.rtai.org/.
19. RTLinux. http://es.wikipedia.org/w/index.php?title=RTLinux&oldid=64581733, March 2013. Page Version ID: 64581733.
20. Rtlinuxpro Victor Yodaiken and Victor Yodaiken. Temporal inventory and real-time synchronization in.
21. Wind river. http://www.windriver.com/.