Una Panorámica del Problema de Inversión de Prioridades en Sistemas Operativos de Tiempo Real

Claudio Aciti

INTIA/INCA - Depto de Computación y Sistemas - Facultad de Ciencias Exactas

Universidad Nacional del Centro de la Provincia de Buenos Aires (7000) - Tandil - Argentina

caciti@exa.unicen.edu.ar

Resumen

En últimos años, el interés por los sistemas de tiempo real se ha renovado debido al enorme crecimiento de los sistemas embebidos. Al usar sistemas de tiempo real es de vital importancia conocer el comportamiento temporal de sus servicios, y qué medidas adopta en situaciones que puedan degradar su rendimiento. Esta información facilita el análisis y la evaluación del sistema operativo en su fase de desarrollo, lo que permite detectar algunos posibles puntos débiles en el diseño. En un entorno de multiprogramación con prioridades, la competencia por el acceso a los recursos puede crear anomalías, y en consecuencia estropear la planificación temporal. Estas anomalías son conocidas como los interbloqueos y la inversión de prioridades. Esta última no tiene solución pero si puede ser minimizada. En este proyecto se pretende analizar el problema de la inversión de prioridades y los mecanismos para abordarla. También se hace un estudio de los protocolos disponibles en varios de los sistemas operativos de tiempo real más importantes, para minimizar esta problemática.

1. Introducción

En la actualidad, los sistemas embebidos han logrado un gran auge gracias a sus diferentes campos de aplicaciones y sus bajos costos comparados con sistemas tradicionales [8, 18]. Es muy común el uso cotidiano de sistemas embebidos, ya sea en electrónica de consumo (lavarropas, heladeras, microondas, relojes, consolas de juegos, control remoto, cámaras de video, fax, CD, DVD, GPS, televisión digital), en sistemas de comunicación (sistemas de telefonía, contestadores, celulares, beepers, PDAs, routers), en automóviles (inyección electrónica, frenos, elevadores de vidrios, control de asientos, instrumentación, seguridad), en la industria (instrumentación, monitoreo, control, robótica, control de tráfico, manejo de códigos de barras, ascensores), en medicina (monitores cardíacos, renales y de apnea, marcapasos, máquina de diálisis), entre otros [9]. Estas aplicaciones necesitan claramente ser controladas por sistemas de tiempo real, por lo que el interés por estos se ha visto renovado al igual que sus problemáticas.

Aunque el término tiempo real se utiliza con excesiva frecuencia para un gran número de aplicaciones que tienen respuesta rápida, la correcta definición de este tipo de sistemas se puede enunciar como sistemas informáticos en los que la respuesta de la aplicación ante estímulos externos, no solo debe ser lógicamente correcto sino que debe realizarse dentro de un plazo de tiempo establecido [17]. Dado que el sistema operativo es el elemento que más restringe la capacidad de reacción de un sistema de tiempo real, resulta de vital importancia conocer el comportamiento temporal de sus servicios. Esta información facilita el análisis y la evaluación del sistema operativo en su fase de desarrollo, lo que permite detectar puntos débiles en el diseño [2, 17].

Entre las medidas que caracterizan temporalmente a los sistemas operativos de tiempo real se puede nombrar al cambio de contexto, a la latencia de interrupción, a la sincronización de procesos, a la comunicación

entre procesos, etc. Cuando se diseña un sistema con desalojo, en el que se comparten recursos, es necesario prestar atención al problema de la inversión de prioridades. Desconocer la forma en que el sistema operativo resuelve esta anomalía puede llevar a situaciones desastrosas, como es el famoso caso ocurrido en Marte [16].

A continuación, se hace una descripción del problema. En la 3^{ra} sección se presentan los mecanismos para evitar este problema. En la 4^{ta} sección se hace un estudio de como algunos de los sistemas operativos de tiempo real principales tratan este tema.

2. Descripción del Problema

Se define un sistema operativo de tiempo real a aquel que, no solo debe ser correcto lógicamente, sino que también debe hacerlo en un tiempo limitado [17]. En estos sistemas, a las tareas se les asignan prioridades según su importancia. Se dice que la programación en tiempo real está, generalmente, basada en un ranking de prioridades y en una regla simple que dice que nada debería retrasar la ejecución de una tarea de prioridad máxima.

La sincronización de tareas con prioridades es muy dificultoso ya que pueden entrar en conflicto y porque demasiados niveles de prioridades pueden sabotear la perfomance. Cuando un sistema asigna prioridades a partir de un rango muy grande, va a ser necesario que el sistema tenga un alto costo de computación. A raíz de esto, una cuestión que suele surgir es, si se tiene en claro la diferencia entre las tareas como para que estas tengan prioridades tipo $94, 95, \ldots, 100$ [21, 22, 23].

Los recursos, a los que acceden las tareas, pueden ser dispositivos de Hardware externos (sensores, actuadores, cámaras, etc), elementos de computadora (memorias, discos rigidos, disqueteras, etc) y elementos de software (mutex, colas, semáforos, etc). Cada recurso tiene n_k unidades, y cada unidad se usa de forma no interrumpible y exclusiva, aunque si pueden detenerse para hacer un cambio de contexto con otra tarea. Los recursos que pueden ser accedidos por más de una tarea a la vez, se modelan como varias unidades, para lograr que el acceso sea exclusivo.

Cuando una acción quiere usar una unidad de un recurso, primero debe bloquearlo para usarlo de forma exclusiva y no interrumpible. Al finalizar, debe desbloquearlo para dejarlo liberado. El segmento de un proceso que bloquea una unidad y luego la desbloquea se llama región crítica (Figura 1). Si la petición de un bloqueo falla, la acción solicitante se bloquea y espera hasta que el recurso que necesita sea liberado.

```
while(1)
    sección de ingreso
        sección crítica
    sección de egreso
    sección restante
do
```

Figura 1: Estructura General de un Proceso

Una región crítica se representa de la siguiente forma $[R, n; e]$, donde R es el recurso, n la cantidad de unidades, y e las unidades de tiempo de ejecución que necesita. Una vez que se accede a una región crítica no se la puede liberar hasta que no termina de usarse. A partir de esta última afirmación es donde surge el problema de inversiñon de prioridades en los sistemas de tiempo real.

Los sistemas de tiempo real duros deben completar una tarea crítica dentro de una lapso preestablecido. Para cumplirlo, las tareas deben indicarle al planificador el plazo de tiempo en que deben terminar o realizar la E/S. El planificador solo aceptará aquellas tareas a las que puede garantizar que terminarán en tiempo y forma, y rechazará a las tareas con las que no puede cumplir. Para que el planificador pueda dar semejantes garantías debe conocer con exactitud cuanto tarda la ejecución de cada tipo de función del sistema operativo.

Los sistemas de tiempo real blando son menos restrictivos. Requieren que los procesos críticos tengan mayor prioridad que otros que no lo son. Aunque agregarle funcionalidad de tiempo real a sistemas de tiempo

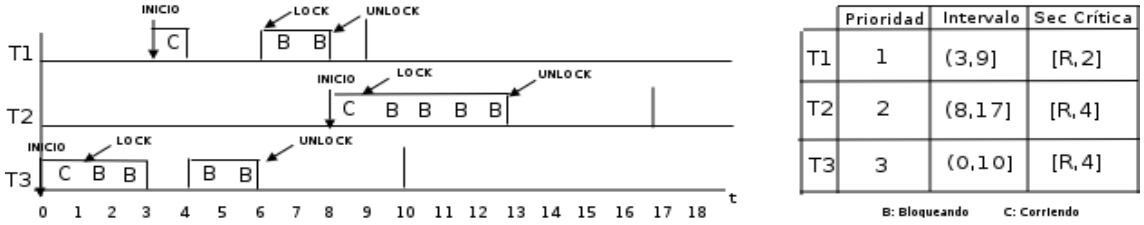
compartido puede hacer que la asignación de recursos no sea equitativa y causar retardos mas largos, o incluso inanición, en algunos procesos. La implementación de una funcionalidad de tiempo real blando requiere de un diseño cuidadoso de planificación y aspectos relacionados del sistema operativo. En primer lugar, los procesos deben tener una prioridad asignada (cuanto mas crítico sea el proceso mayor prioridad debe tener). La prioridad de los procesos de tiempo real no deberán degradarse. En segundo lugar, la latencia de despacho debe ser pequeña, ya que cuanto menor sea, mayor es la rapidez con que un proceso de tiempo real empieza a ejecutarse una vez que esté listo.

Es facil conseguir que se cumpla la primera propiedad, ya que solo se necesita garantizar que los procesos de tiempo real no degraden su prioridad. La segunda propiedad es más complicado de garantizar, ya que muchos sistemas operativos están obligados a esperar a que se complete una llamada al sistema o bien a que ocurra un bloque por E/S, antes de efectuar una conmutación de contexto.En un entorno de multiprogramación, varios procesos pueden competir por un número finito de recursos. Cuando un proceso necesita leer/escribir datos de un recurso que está siendo accedido por un recurso de prioridad más baja, debe esperar a que este termine de usarlo y lo libere. Esta situación se llama inversión de prioridades [17].

La competencia por los recursos puede crear anomalías, y en consecuencia estropea la planificación temporal. Estas anomalías son los interbloqueos y la inversión de prioridades.

2.1. Inversión de Prioridades

Se define Inversión de Prioridades a la situación en donde una acción, que necesita acceder a un recurso, debe esperar porque está siendo usado por una acción de menor prioridad. Este problema no tiene solución pero si puede minimizarse aplicando diferentes mecanismos.



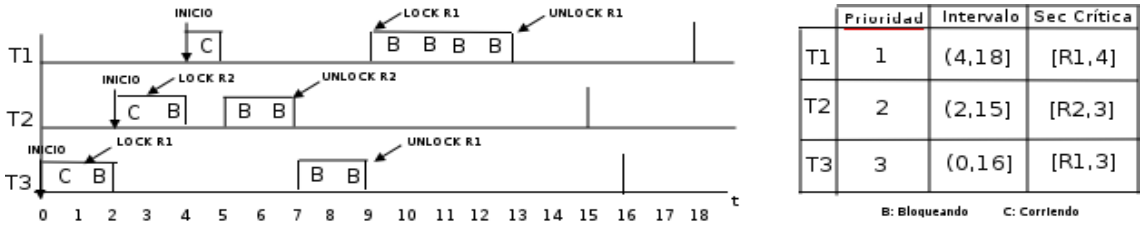
	Prioridad	Intervalo	Sec Crítica
T1	1	(3,9]	[R,2]
T2	2	(8,17]	[R,4]
T3	3	(0,10]	[R,4]

B: Bloqueando C: Corriendo

Figura 2: Inversión de prioridades

En la Figura 2, en $t = 3$ se inicia $T1$ y al tener mayor prioridad desaloja a $T3$. En $t = 4$, la tarea $T1$ quiere entrar en su sección crítica pero no puede bloquear al recurso R porque ya lo tiene bloqueado $T3$. Entonces $T1$, debe esperar a que $T3$ termine y libere el recurso R.

Esta anomalía puede tambien darse de forma transitiva. O sea que una acción que espera por un recurso, está siendo demorada por otra acción de menor prioridad que está usando otro recurso.



	Prioridad	Intervalo	Sec Crítica
T1	1	(4,18]	[R1,4]
T2	2	(2,15]	[R2,3]
T3	3	(0,16]	[R1,3]

B: Bloqueando C: Corriendo

Figura 3: Inversión de prioridades transitiva

En la Figura 3, en $t = 2$ da inicio la tarea $T2$ y desaloja a la tarea $T3$. En $t = 3$, $T2$ ingresa a su sección crítica y bloquea al recurso $R2$. En $t = 4$, la tarea de máxima prioridad $T1$, da inicio y toma el control del

cpu, pero cuando en $t = 5$ quiere entrar en su sección crítica para acceder a $R1$, debe esperar porque lo tiene bloqueado $T3$ desde $t = 1$. Es por esto qué, primero $T1$ debe esperar a que $T2$, que está usando el recurso $R2$, termine, así $T3$ puede retomar su ejecución y terminar de usar el recurso $R1$. Una vez terminado este camino, recién ahí puede $T1$ ingresar a su sección crítica.

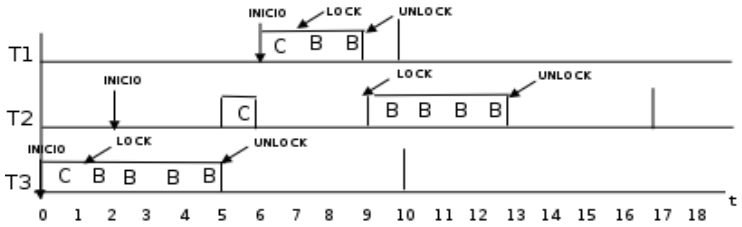
A continuación se presentan los mecanismos usados para controlar estas anomalías.

3. Mecanismos de control de la inversión de prioridades

Los mecanismos de control no pueden evitar el problema de la inversión de prioridades, pero si pueden minimizarlo. El propósito de estos es que, una tarea no retrase demasiado su ejecución por tener esperar a otra con menor prioridad. Antes de repasar los mecanismos de minimización, cabe recordar que una vez que una tarea ingresó en su sección crítica debe terminar antes de liberar el recurso sino se produciría una inconsistencia. También vale tener en cuenta que los cambios de contexto solo producen retrasos, por lo que es conveniente no abusar de ellos.

3.1. Secciones críticas no interrumpibles

Cuando una acción consigue un recurso, pasa a planificarse con la máxima prioridad. Este mecanismo evita totalmente los cambios de contexto y los interbloqueos, pero no controla la inversión de prioridades. En general tiene un rendimiento bajo, aunque es muy buen protocolo en situaciones donde las secciones críticias son muy cortas y si hay acciones que tienen muchas interacciones entre ellas.



	Prioridad	Intervalo	Sec Crítica
T1	1	(6,10]	[R,2]
T2	2	(5,17]	[R,4]
T3	3	(0,10]	[R,4]

B: Bloqueando C: Corriendo

Figura 4: Protocolo de secciones críticas no interrumpibles

En la Figura 4, se puede ver en $t = 2$ como la tarea $T2$ no toma el control porque la tarea $T1$ está dentro de su sección crítica y es ininterrumpible. De esta forma, se evita el cambio de contexto con la tarea $T2$. Se puede observar en $t = 6$, que $T1$ desaloja a $T2$ porque esta última todavía no ingresó en su sección crítica.

3.2. Protocolo de Herencia de Prioridad

Este protocolo ajusta la prioridad dinamicamente de la acción que está usando un recurso, y este es requerido por otra acción de mayor prioridad. Se suponen dos procesos P_x y P_y, donde $\pi(P_x)$ es la prioridad de P_x y $\pi(P_y)$ es la prioridad de (P_y) y $\pi(P_x) < \pi(P_y)$. Si P_x está bloqueando al recurso R mientras lo usa, y la acción P_y intenta bloquearlo, entonces P_x hereda la prioridad de P_y mientras esté dentro de su sección crítica. De esta forma se evita que P_x sea desalojado [3].

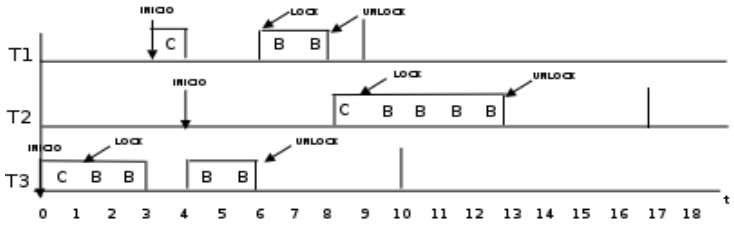
En la Figura 5, en $t = 3$ la tarea $T1$ inicia y en $t = 4$ intenta bloquear al recurso R pero no logra porque está bloqueado por la tarea $T3$. En ese instante, $T3$ hereda dinámicamente su prioridad por la de $T1$. Con este cambio, se puede observar que en $t = 4$, la tarea $T2$ no tiene suficiente prioridad para tomar el control.

Este protocolo, si bien es simple, genera un retraso en la finalización de las tareas. Los pasos que debe seguir son:

1. Bloquear la tarea de mayor prioridad;

2. Pasar a la prioridad menor;
3. Reiniciar la tarea de menor prioridad;
4. Realizar los cálculos en la sección crítica de la tarea de menor prioridad;
5. Desbloquear la tarea de mayor prioridad cuando el recurso este listo;
6. Reiniciar la tarea de mayor prioridad.

Como máximo se puede bloquear una vez por región crítica y una vez por recurso.



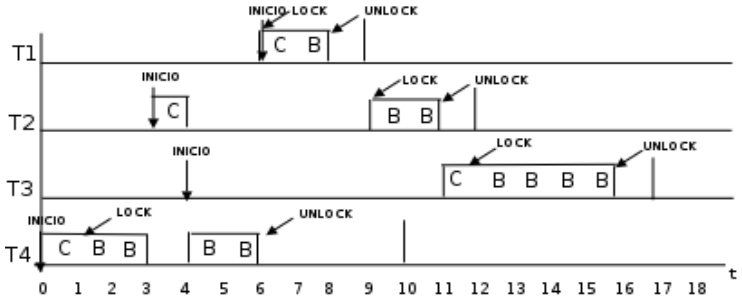
	Prioridad	Intervalo	Sec Crítica
T1	1	(3,9]	[R,2]
T2	2	(4,17]	[R,4]
T3	3	(0,10]	[R,4]

B: Bloqueando C: Corriendo

Figura 5: Protocolo de Herencia de Prioridad

3.3. Protocolo de Techo de Prioridad

Este protocolo designa como techo de prioridad de un recurso a la máxima prioridad de las tareas que lo usan. El protocolo consiste en que la prioridad de una tarea sea dinámica y que su valor sea el máximo entre su prioridad básica y las prioridades de las tareas a las que bloquea. Entonces, una tarea solo puede usar un recurso si su prioridad dinámica es mayor que el techo de todos los recursos en uso por otras tareas [3].



	Prioridad	Intervalo	Sec Crítica
	1	(6,9]	[R3,1]
T2	2	(3,12]	[R1,2]
T3	3	(4,17]	[R2,4]
T3	4	(0,10]	[R1,4]

B: Bloqueando C: Corriendo

Figura 6: Protocolo Techo de Prioridad

En la Figura 6, se ve que en $t = 3$ la tarea $T2$ da inicio a su ejecución, un instante despues, quiere bloquear al recurso $R1$ pero este ya está bloqueado por la tarea $T4$. Entonces, $T4$ cambia su prioridad dinámicamente por la prioridad de la tarea de mayor prioridad que la bloquea ($T2$ en este caso). En $t = 4$, la tarea $T3$ no puede ni siquiera iniciar porque su prioridad es menor que la actual que tiene $T4$, pero en $t = 6$, el control lo toma $T1$, ya que su prioridad es la mayor y quiere usar el recurso disponible $R3$.

Con este mecanismo, cada tarea se puede bloquear una vez como máximo en cada ciclo. Además, evita los interbloqueos.

3.4. Protocolo de Techo de Prioridad Inmediato

Es un derivado del Protocolo de Techo de Prioridad. En este protocolo, la tarea que accede a un recurso hereda inmeditamente el techo de prioridad del recurso. Este protocolo es más facil de implementar y es más

eficiente (hay menos cambios de contexto). Si se tiene en cuenta el ejemplo presentado en la Figura 6, en $t = 3$, la tarea $T2$ no hubiese siquiera iniciado y se hubiese evitado el cambio de contexto.

4. Prioridades en los Sistemas Operativos de Tiempo Real

Las prioridades en los sistemas operativos de tiempo real varían en la cantidad disponible, en los algoritmos de asignación y en los protocolos para evitar la inversión de prioridad. A continuación se estudian los sistemas operativos de tiempo real más usados: FreeRTOS, MaRTE OS, QNX, RTAI/Linux, RTLinux, VxWorks y WindowsCE.

4.1. FreeRTOS

En FreeRTOS, es un mini kernel de tiempo real. Se usa el protocolo de herencia de prioridad para evitar el problema de la inversión de prioridades [6].

El valor de la prioridad más alta se define, en el archivo `FreeRTOSConfig.h`, asignandole un valor a la constante `MAX_PRIORITIES`.

Los valores de las prioridades van desde 0 hasta MAX_PRIORITIES - 1. La prioridad más baja es la 0, y la más alta es MAX_PRIORITIES (por defecto, las tareas tienen valor 0).

Las funciones de para el manejo de prioridades son (task.h):

- `unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask )`: Retorna la prioridad de una tarea.
- `void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority )`: Setea la prioridad de una tarea.

y las macros

- `taskENTER_CRITICAL`: Indica el inicio de una región crítica de código.
- `taskEXIT_CRITICAL`: Indica el final de una región crítica de código.

4.2. MarteOS

MaRTE OS, con un core basado en ADA, es desarrollado por el Grupo de Computadoras y Tiempo Real de la Universidad de Cantabria. Tiene como mecanismo para manejar la inversión de prioridades a la Herencia de Prioridad y el techo de Prioridad [7].

Si se desea no tratar la inversión de prioridad entonces se setea la marcro `PTHREAD_PRIO_NONE`. En cambio, para usar el protocolo techo de prioriedad se debe setear la macro `PTHREAD_PRIO_PROTECT`, y si se quiere usar el protocolo herencia de prioridad se debe usar la macro `PTHREAD_PRIO_INHERIT`.

Los niveles de prioridades son determinados al momento de compilar el sistema operativo. Los valores de las prioridades se cambairn en el archivo de configuración `configuration_parameters.ads`.

4.3. QNX

QNX, es un micro-kernel de tiempo real de muy alta performance [12]. QNX soporta procesos y threads con 64 niveles de prioridad (anteriormente usaba 32 niveles) y utiliza un esquema de planificación basado en prioridades dinámicas [11]. La prioridad por defecto de un proceso es la que hereda de su padre; normalmente a un proceso iniciado desde el shell se le asigna la prioridad 10.

Este sistema operativo designa a la prioridad 0 como la mínima prioridad y a 63 como la máxima.

Las funciones de para el manejo de prioridades son:

- `getprio()`: Retorna la prioridad de una tarea.

- `setprio()`: Setea la prioridad de una tarea.

El mecansimo que usa para resolver el problema de inversión es la herencia de prioridad.

4.4. RTAI/Linux

En RTAI, es un parche en el kernel Linux que le agrega funcionalidad de tiempo Real. Este sistema es desarrollado por el Politecnico di Milano - Dipartimento di Ingeneria Aerospaziale (DIAPM). El mecanismo para manejar la inversión de prioridades es la Herencia de Prioridad [13, 14].

Los valores de las prioridades son definidos en el archivo `base\include\rtai_sched.h` como:

- `RT_SCHED_HIGHEST_PRIORITY 0`
- `RT_SCHED_LOWEST_PRIORITY 0x3fffffff`
- `RT_SCHED_LINUX_PRIORITY 0x7fffffff`

donde se puede ver que las prioridades de menor valor son las de mayor prioridad.

Las funciones de para el manejo de prioridades son:

- `int rt_get_prio (RT_TASK *task)`: Retorna la prioridad base nativa de la tarea actual.
- `int rt_get_inher_prio (RT_TASK *task)`: Retorna la prioridad base que ha heredado de otras tareas.
- `int rt_change_prio(RT_TASK *task,int priority)`: Cambia la prioridad de una tarea.

4.5. RTLinux

En RTLinux, las prioridades de las tareas son estáticas y son manejadas con un algoritmo FIFO o un Round Robin [15]. En principio, el kernel de RTLinux no soporta ningún mecanismo para evitar la inversión de prioridades. 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 [21, 23]. Aunque existe una extensión que dá la posibilidad de usar el mecanismo de herencia de prioridad y el techo de prioridad.

Las funciones de para el manejo de prioridades en `rtl sched.c` son:

- `pthread_mutexattr_getprioceiling` : Retorna el valor del techo de prioridad.
- `pthread_mutexattr_setprioceiling`: Setea el valor del techo de prioridad.
- `sched_get_priority_max`: Retorna el valor de la prioridad máxima.
- `sched_get_priority_min`: Retorna el valor de la prioridad mínima.

Los valores de las prioridades para las tareas de tiempo real van desde el 0 al 100, siendo 0 la mayor prioridad. El resto de las tareas tienen prioridades dentro del rango de 101 a 140.

4.6. VxWorks

En VxWorks, de la empresa WindRiverSystem, el mecanismo para manejar la inversión de prioridades es la Herencia de Prioridad [19].

El scheduling de prioridades maneja 256 valores. Siendo 0 la prioridad máxima y 255 la prioridad mínima.

Las funciones de para el manejo de prioridades en `rtl sched.c` son:

- `taskPrioritySet( )`: Setea la prioridad de la tarea.
- `taskPriorityGet( )`: Retorna la prioridad de la tarea.
- `sched_get_priority_max( )`: Retorna la máxima prioridad.
- `sched_get_priority_min( )`: Retorna la mínima prioridad.

4.7. WindowsCE

WindowsCE es un sistema preemptive y multitasking. Soporta corriendo 32 procesos simultaneamente. Un proceso consiste en 1 o más threads, y uno de ellos es designado como primario en cada proceso. WindowsCE soporta 256 niveles de prioridades, siendo 0 la prioridad más alta y 255 la más baja. Las prioridades consideradas más altas son las que pertenecen al rango entre 0 y 247. Las 8 restantes, están reservadas por prevención, por si es necesario degradar la performance del sistema [20].

Las funciones de para el manejo de prioridades son:

- `CeGetThreadPriority`: Retorna la prioridad base nativa de la tarea actual.
- `CeSetThreadPriority`: Setea la prioridad de una tarea.

WindowsCe garantiza el manejo de la inversión de prioridad, solo en un nivel de profundidad, para el caso en que una Tarea $T1$ de mayor prioridad espera por la tarea $T2$, quien a su vez está esperando por $T3$. En este caso, $T2$ hereda la prioridad de $T1$. Para las tareas $T2$ y $T3$, se planifican basados en un algoritmo de planificación normal, en donde la inversión de prioridad está permitida.

5. Conclusiones

La sincronización de tareas con prioridades es un tema sin resolución. La mejor estrategia para reducir los conflictos es reducir la competición por los recursos compartidos. Entre los mecanismos para minimizar el problema de inversión, el protocolo techo de prioridad es el que mejor performance tiene aunque no está implementado casi en ningún sistema operativo (en MaRTEOS y una extensión de RTLinux). El mecanismo herencia de prioridad, si bien no elimina la inversión de prioridades (ni los interbloqueos), es la forma más común que tienen los sistemas operativos para afrontar la inversión de prioridades.

Referencias

[1] M. Aldea Rivas, M. Gonzalez Harbour. *MaRTE OS: An Ada Kernel for Real-Time Embedded Applications*. Proc. of the 6th International Conference on Reliable Software Technologies (Ada-Europe 2001). Belgium. 2001

[2] M. Alonso, M. Aldea, M. González Harbour. *Caracterización temporal de los servicios del sistema operativo de tiempo real MaRTE OS*. Simposio de Sistemas de Tiempo Real. pp: 53-60. ISBN: 84-9732-448-X. 2005

[3] G. Buttazzo. *Why Real-Time Computing?*. Proccedings of the 2006 ANIPLA. International Congress on Methodologies for Emerging Techonologies in Automation. 2006

[4] G. Buttazzo. *Rate Monotonic vs EDF: Judgement Day*. Embedded Systems, Lecture Notes in Computer Science(2855) pp:67-83. 2003

[5] A. Crespo, A. Alonso. *Una Panorámica de los sistemas de tiempo real*. Revista Iberoamericana de Automática e Informática Industrial, pp:7–18, vol 3, num 2, 2006.

[6] *FreeRTOS*.
`http://www.freertos.org`.

[7] Grupo de Computadoras y Tiempo Real de Cantabria. *MaRTE OS*.
`http://marte.unican.es`

[8] J. A. Jaramillo Villegas, L. M. Perez Perez, H. J. Osorio Ríos. *Linux sobre una FPGA*. Sciencia et technica. Año XIII. Número 37. ISSN 0122-1701. 2007

[9] J. Osio, F. Salguero, J. Rapallini, A. Quijano. *Análisis de Modelos Computacionales para Sistemas Embebidos*. XII Iberchip, IWS06. Costa Rica. 2006

[10] F. Proctor. *Measuring performance in Real-Time Linux*. In Real-Time Linux Worksop. Milan. Italia. 2001

[11] QNX. *QNX Operating System. System Architecture*. QNX Software System Ltd. 1997

[12] QNX Neutrino RTOS.
www.qnx.com

[13] Politecnico di Milano - Dipartimento di Ingeneria Aerospaziale. *RTAI the RealTime Application Interface for Linux from DIAPM*.
https://www.rtai.org.

[14] Politecnico di Milano - Dipartimento di Ingeneria Aerospaziale. *RTAI_User_Manual*. 2006
https://www.rtai.org

[15] FSM Labs. *RTLinux*.
http://www.rtlinux.org.

[16] L. Sha, R. Rajkumar, J. P. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. In IEEE Transactions on Computers, vol. 39, pp:1175-1185. 1990

[17] A. Silverchatz, P. B. Galvin. *Sistemas Operativos*. Prentice Hall. 1999

[18] J. Valvano. *Introducción a los sistemas de microcomputadores*. Thomson Learning Ibero. ISBN 9706863168. 2003

[19] Wind River Corp. *VxWorks Real-Time Operating System*.
http://www.windriver.com/vxworks.

[20] Microsoft. *WindowsCE*.
http://msdn.microsoft.com.

[21] V. Yodaiken. *Temporal inventory and realtime synchronization in RTLinuxPro*. Technical report, FSM-Labs Inc, 2004.
http://www.fsmlabs.com/developers/

[22] V. Yodaiken. *Against Priority Inheritance*. FMSLABS Technical Report. 2002.
http://www.fsmlabs.com/developers/

[23] V. Yodaiken. *FSMLabs Lean POSIX for RTLinux*. Technical report, FSMLabs Inc, 2001.
http://www.fsmlabs.com/developers/