Inversión de prioridades: análisis de desempeño

Santiago Alessandri, Luciano Iglesias, Fernando Romero y Fernando G. Tinetti**

Instituto de Investigación en Informática III-LIDI
Facultad de Informática, UNLP
san.lt.ss@gmail.com,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. Esta función es llevada a cabo por el planificador del sistema operativo, diseñado para poder cumplir con las restricciones temporales de las tareas a realizar, teniendo en cuenta sus valores de prioridad. Cuando hay recursos compartidos por estas tareas, se puede producir el efecto llamado inversión de prioridades. En este trabajo se analiza este efecto y se evalúan las soluciones implementadas para este problema en el Sistema Operativo de Tiempo Real GNU/Linux con parche RT-PREEMPT [1]. Además, se evalúa otro mecanismo, conocido como restauración de recursos, que no está implementado en GNU/Linux.

Keywords: Planificación de tareas de tiempo real, inversión de prioridades, Restauración de recursos

1. Introducción

En los sistemas de tiempo real hay tiempos esperables debidos al procesamiento de las instrucciones que implementan las diferentes tareas, junto con las necesarias para las funciones del Sistema Operativo. Además aparecen demoras no previstas que afectan el funcionamiento de estos sistemas en tiempo real [2]:

1. El bloqueo debido a inversión de prioridades.
2. La demora del proceso debido a la concurrencia.
3. Secciones de código no interrumpible.

En el presente trabajo se describe el problema de inversión de prioridades, se analizan los métodos existentes para mitigar las demoras que surgen. Entre estos métodos se encuentran:

1. Herencia de prioridades
2. Techo de prioridades
3. Restauración de recursos

** Investigador CICPBA

Un sistema de tiempo real debe realizar funciones de diferente grado de criticidad. Por ello es conveniente que estas funciones estén a cargo de diferentes tareas. Las mismas tendrán distintas prioridades de tal manera que si no pueden cumplirse los requerimientos temporales de todas las tareas, se dejen de cumplir las que implementan funciones de menor grado de criticidad. Una manera de lograr esto es usar prioridades fijas de ejecución para cada tarea, a ser tenidas en cuenta por el planificador. También es común que entre dichas tareas se comparta un recurso, por ejemplo un buffer, cuyo acceso por las tareas que lo comparten debe ser sincronizado, empleando una protección sobre el recurso, por ejemplo un semáforo. De esta manera, mientras una tarea accede a la región crítica, las demás tareas que tengan que acceder quedan bloqueadas en su intento por acceder a la región crítica. Si las tareas que quedan demoradas tienen una prioridad mayor que la que está dentro de la región crítica, nos encontramos con que se produce una inversión de prioridades. Si el sistema operativo usa un planificador apropiativo, esta situación se ve agravada si la tarea que está en la región crítica, es desalojada del uso de CPU por una tarea de prioridad intermedia. Si se prolonga la situación de no acceso a la CPU por el proceso de baja prioridad que está en la sección crítica, la demora por la inversión de prioridades puede llegar a ser indeterminada. Esta situación debe ser evaluada en el caso del test de factibilidad de planificación, que verifica que se cumplan los plazos de todas las tareas [3] [4].

El tiempo de respuesta máximo de una tarea de alta prioridad cuando una tarea de menor prioridad esta lista para ser ejecutada está dada por la Ec. 1.

$$R = C + B + I \tag{1}$$

Donde C es el peor caso de tiempo de ejecución de la tarea, B es el tiempo de bloqueo e I es la máxima interferencia que sufre la tarea en su ejecución. El tiempo de bloqueo B será la suma de los tiempos máximos de ejecución de cada uno de los tiempos de ejecución de aquellas tareas que comparten el mismo recurso.

Las tareas se implementan en GNU/Linux a través de threads por los que nos referiremos a ellas de esta forma.

2. Mecanismos para mitigar la inversión

El bloqueo en cualquiera de los mecanismos, comienza en el momento en que el thread quiere acceder a la región crítica haciendo uso de la primitiva *lock*. Con lo cual el tiempo de lock es el tiempo de bloqueo que sufren los threads antes de poder hacer uso del recurso compartido. En las siguientes secciones se describe los métodos de mitigación del problema de inversión de prioridades que intentan reducir éste tiempo de bloqueo.

2.1. Techo y herencia de prioridades

La solución denominada herencia de prioridad propone disminuir el tiempo de bloqueo debido a la inversión de prioridades elevando la prioridad del thread

con un lock en un recurso compartido al máximo de las prioridades de los threads que estén esperando el recurso. De esta manera la demora se reduce al tiempo de ejecución de la sección crítica del thread de baja prioridad. La idea de este protocolo consiste en que cuando un thread bloquea indirectamente a threads de más alta prioridad, la prioridad original es ignorada y ejecuta la sección crítica correspondiente con la prioridad más alta de los threads que está bloqueando [3] [5].

La solución denominada techo de prioridad es un método que disminuye el tiempo de bloque debido a la inversión de prioridades mediante la asignación predefinida de un techo de prioridad a cada recurso, de manera similar a la solución de herencia. Cuando un thread adquiere un recurso compartido, la prioridad de este thread se eleva temporalmente al techo de prioridad del mencionado recurso. El techo de prioridad debe ser más alto que la prioridad de todos los threads que puedan acceder al recurso compartido. De esta forma, cuando un thread se está ejecutando con el techo de prioridad de un recurso, el procesador no podrá ser apropiado por otro thread que quiera acceder al mismo recurso, puesto que todos tendrán menor prioridad [5].

2.2. Copia y restauración

La idea fundamental de esta solución reside en la generación de copias del recurso compartido manteniendo el estado de quién es realmente el dueño del recurso (thread de mayor prioridad). De esta forma se logra que procesos de mayor prioridad no incurran en esperas innecesarias evitando así la inversión de prioridades. Se tomó como base la propuesta planteada por Tarek Helmy y Syed S. Jafri en [6]. Allí se plantea un sistema para evitar la inversión de prioridades basado en la copia y restauración de recursos. Cuando un thread quiere acceder a un recurso, si el thread es considerado de alta prioridad se le da el recurso real para que trabaje, mientras que si el thread es considerado de baja prioridad lo que se le da para que trabaje es una copia del recurso.

En caso que el recurso esté siendo usado por un thread de baja prioridad y un thread de alta prioridad lo requiera, se le indica al thread de baja prioridad que le ha sido revocado el uso del recurso. El thread de baja prioridad verifica sistemáticamente si el recurso que posee le fue revocado y en caso de que esto haya pasado, se restaura el recurso y se elimina el thread de baja prioridad, permitiéndole al thread de alta prioridad acceder al mismo.

Para implementar este mecanismo los autores introducen una capa intermedia entre los threads y el programa principal. Esta capa es la encargada de realizar el desalojo y la finalización de los threads de baja prioridad cuyo recurso ha sido revocado. Además, los threads deben ser modificados para que periódicamente realicen el chequeo del recurso asignado y auto determinen si deben ser finalizados [7] [8] [9].

3. Casos de prueba

Utilizando GNU/Linux con parche RT-PREEMPT [1] se llevaron a cabo una serie de pruebas para probar en situaciones de carga los distintos mecanismos para mitigar la inversión de prioridades: herencia de prioridades, techo de prioridades y restauración de recursos.

Se realizó la ejecución de cuatro casos diferentes (no hacer nada para evitar la inversión de prioridad, herencia de prioridades, techo de prioridades y restauración de recursos). Para cada uno de estos cuatro casos se realizaron 1.000 ejecuciones y se calcularon los estadísticos básicos (media aritmética y desvío estándar), tanto del tiempo de *lock* como de *unlock* del thread de máxima prioridad.

3.1. Desarrollo para la experimentación de restauración de recursos

La técnica aquí presentada sigue los principios fundamentales del trabajo [6]. Plantea cambios que harán posible su implementación como una biblioteca en lenguaje C que sea utilizable en GNU/Linux con parche RT-PREEMPT.

Se describe el comportamiento al intentar acceder al recurso (lock) y al dejar de usar el recurso (unlock).

Cuando un thread quiere acceder al recurso, si el mismo está libre o está siendo utilizado por un recurso de menor prioridad, se genera una copia del recurso y se le indica al thread que trabaje con esta copia como si fuera el recurso original. En caso que el recurso está siendo utilizado por un thread de mayor o igual prioridad que el thread que hace el requerimiento, este se lo bloquea manteniendo el orden por prioridad en la cola de espera por el recurso.

Cuando un thread deja de usar el recurso debe liberarlo. Este proceso consiste en verificar si el thread tiene el acceso exclusivo al recurso o el mismo le fue entregado a otro de mayor prioridad. Si el thread es el actual poseedor o dueño del recurso entonces se actualiza el recurso real utilizando la copia que posea el thread, en caso contrario la copia es descartada. Se le indica al thread si la liberación del recurso fue válida o no. Además, en caso de que la liberación haya sido exitosa, se despierta al thread con mayor prioridad que esté esperando por el recurso si es que los hay.

La biblioteca consta de una estructura de datos que representa al recurso y contiene toda la información para la administración del mismo. Además se proveen las funciones necesarias para la administración del recurso.

3.2. Primitivas de restauración de recursos

La implementación provee cuatro primitivas para la administración del recurso.

1. Inicialización del recurso.

El método dado para realizar la inicialización del recurso toma como parámetros un puntero a la estructura del recuso que quiere ser inicializada, el puntero al recurso real a administrar y el puntero a la función de que debe usarse para generar las copias del recurso.
La inicialización consiste en asignación de los valores correctos a la estructura de recurso. Se fija como recurso original y actual el valor indicado por el puntero al recurso real, se guarda el puntero de la función de copia, se inicializa la lista de colas de esperas y se indica que el recurso no tiene dueño.
Luego de la inicialización, los threads pueden utilizar las funciones para el acceso al recurso.

2. Bloqueo del recurso (lock).
   Esta función recibe un puntero al recurso al cual se quiere acceder y cuando se concede el acceso se devuelve el puntero a una copia.
   Esta función es bloqueante ya que si el recurso está siendo utilizado por un thread de mayor prioridad, el thread será puesto en una cola de espera para acceder al recurso más adelante.
   El bloqueo del recurso consiste en:
   a) Si el recurso está siendo usado por un thread de mayor o igual prioridad que él, se coloca al thread en la cola de espera de su prioridad.
   b) Cuando el recurso está siendo usado por un thread de menor prioridad o está libre: se genera una copia del recurso actual, se asigna al thread llamador como dueño del thread y se le devuelve el puntero a la copia para que el thread lo use.
3. Liberación del recurso (unlock).
   Cuando un thread deja de usar un recurso debe proceder a liberarlo. Esta función recibe como parámetro el puntero al recurso a ser liberado y la copia del recurso utilizado por el thread. La función retorna un valor indicando si la liberación fue exitosa o no.
   También se verifica que el proceso que está liberando el recurso sea el actual dueño del mismo. En caso de que esto sea así, se reemplaza el recurso real dentro del recurso por la copia que le había sido dada al thread. Luego, se despierta al siguiente thread que esté esperando por el recurso y se le indica al proceso llamador que el recurso fue liberado de forma exitosa. En caso contrario, se descarta la copia y se le indica al proceso que la liberación no fue exitosa, por lo que deberá volver a adquirir el recurso.
4. Destrucción del recurso.
   Está operación deshace la estructura de control utilizada para la administración del recurso. La función recibe como parámetro el puntero al recurso a liberar. El valor de retorno de la función es un puntero al recurso real.

3.3. Caso con inversión, herencia y techo de prioridades

Se ejecutó en cada uno de los tres casos un programa con 20 threads que se repitió mil veces. Dichos threads tenían las prioridades asignadas de la siguiente

forma: uno de ellos con una prioridad mayor a todos los demás, 18 con una prioridad intermedia y el restante con una prioridad menor a todos los demás. Durante la ejecución del programa se forzó la inversión mediante mecanismos de sincronización para evaluar las diferentes técnicas para mitigar. El thread de menor prioridad le avisó al de mayor prioridad que podía ejecutarse una vez que este había hecho el lock. Luego, el thread de mayor prioridad le avisó a todos los intermedios que podían ejecutarse y realizó a continuación el lock.

3.4. Caso de copia y restauración de recursos

Se ejecutó un programa con 20 threads que de manera ordenada y de menor a mayor prioridad van solicitando el acceso a un recurso compartido. Este programa también se repitió su ejecución mil veces como en los tres casos anteriores. Cada uno después de hacer el lock le avisa al de siguiente prioridad que ya puede hacerlo. La implementación de las primitivas de lock y unlock se llevan a cabo mediante un biblioteca en espacio de usuario como se explicó anteriormente.

4. Resultados

4.1. Demoras de las primitivas lock y unlock para thread de mayor prioridad

La aplicación realizada para observar el fenómeno de la inversión de prioridad se ejecutó sobre un kernel Linux 3.2 con el parche RT-PREEMPT [1] en un dual-core con hyper threading. La ejecución en más de un core revela diferencias con la ejecución en un solo core, debido a la posibilidad que se agrega al poder tener threads de alta y baja prioridad corriendo simultáneamente (aunque esto no es objeto de estudio de este trabajo). Los mecanismos para mitigar la inversión de prioridades fueron diseñados originalmente para los sistemas monocore.

En el cuadro 1 pueden apreciarse los tiempos de promedio de las primitivas lock y unlock del thread de mayor prioridad de cada uno de los casos. De igual manera en el cuadro 2 pueden apreciarse los desvíos estándar de las primitivas lock y unlock del thread de mayor prioridad.

	Inversión	Herencia	Techo	Restauración de recursos
lock	41.430.000.000	4.126.000.000	4.119.000.000	2.621
unlock	1.518	3.126	2.110	7.260

Cuadro 1. Media de los tiempos de las primitivas lock y unlock para el thread de mayor prioridad, expresados en nanosegundos.

	Inversión	Herencia	Techo	Restauración de recursos
lock	526.835.662 (1,27 %)	24.830.764 (0,6 %)	22.327.248 (0,54 %)	1583,243 (60,4 %)
unlock	1060,712 (69,87 %)	499,656 (15,98 %)	295,6587 (14,01 %)	2208,688 (39,42 %)

Cuadro 2. Desvío estándar de los tiempos de las primitivas lock y unlock para el thread de mayor prioridad, expresados en nanosegundos. Entre paréntesis aparece el porcentaje que representa el desvío respecto de la media.

5. Conclusiones y líneas futuras

Podemos observar que en el caso de que se produce la inversión y no se aplica ningún método para mitigarla, el tiempo requerido para realizar el lock es un orden de magnitud más grande en promedio que los casos de techo y herencia de prioridad. Esta cuestión es coherente con el hecho de que la espera del lock requiere la liberación del recurso compartido por parte de todos los demás threads de prioridad intermedia.

Para los casos de techo y herencia de prioridad, se puede observar que la media de ambos es similar ya que las tareas a realizar por el planificador son equivalentes. Por otro lado el tiempo de unlock es comparable por las mismas razones. El desvío estándar está por debajo del uno por ciento, cosa que muestra que los tiempos de interferencia así como el mecanismo para mitigar la inversión no son significativos en el resultado. Hay que tener en cuenta que en el tiempo de lock está incluido el tiempo de ejecución que el thread de baja prioridad requiere para terminar la ejecución de su sección crítica.

Para el caso de restauración de recursos, se ve claramente que hay una ventaja clara en el tiempo de bloqueo. Este tiempo es insignificante en relación a lo que es techo y herencia, esto es debido a que no existe demora en el acceso al recurso compartido por parte del thread de mayor prioridad ya que los de menor prioridad trabajan sobre una copia diferente. Hay que tener en cuenta que este mecanismo tiene una dependencia del tamaño que tenga el recurso compartido ya que este debe ser copiado durante la ejecución del lock y restaurado durante la ejecución del unlock. Para las pruebas realizadas el recurso compartido era un espacio de memoria muy pequeño. El desvío estándar es proporcionalmente mucho más grande que en los casos anteriores debido a que los valores de las primitivas lock y unlock son muy pequeños. La única demora que tienen es la provocada por la generación de la copia y actualización del estado del recurso. Este es el punto donde se podrá producir una inversión de prioridades, mientras se tiene el lock del recurso global. Para mitigar esto se reduce la sección crítica al mínimo y se usa herencia de prioridades en este lock.

La técnica aquí propuesta tiene ciertas desventajas entre las cuales tenemos:

Solo admite recursos que se puedan copiar: al necesitar realizar copias de los recursos, el tipo de recursos que pueden utilizarse con esta técnica se ve limitado. Esto se debe a que no todos los recursos admiten copias de sí mismos, es necesario que exista una función que pueda generar una copia del recurso para poder aplicar esta técnica. Archivos y sockets son ejemplos de recurso que no podrán utilizarse con esta técnica.

El acceso al recurso implica el copiado del mismo. El tiempo que tarda en realizarse dicha copia puede ser intolerable. Además, debe analizarse el tamaño que pueden ocupar las copias de los recursos.

Otra desventaja de esta técnica es que no es transparente al programador. Los programadores deben proveer las funciones de copiado para los distintos recursos.

Una limitación de esta técnica es que no permite el bloqueo anidado de recursos de forma transparente sino que queda en manos del programador lograr que el comportamiento en un anidamiento de recursos sea semánticamente correcto.

Por último, también hay que tener en cuenta que pueden existir salidas en los threads (de baja prioridad) de la sección crítica fallidas (unlock) que implican realizar cómputo nuevamente.

Como líneas futuras de investigación se propone analizar el protocolo de restauración de recursos discriminando entre bloqueos de lectura y bloqueos de lectura/escritura para así de esta forma evitar situaciones de copia innecesarias [2].

Referencias

1. RTwiki. http://rt.wiki.kernel.org/index.php/Main_Page.
2. J. Triplett J. Walpole D. Guniguntala, P. E. McKenney. *The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux.* 2008.
3. Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX.* Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.
4. John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 1998.
5. Romero Fernando Tinetti Fernando G. Benencia Raúl, Iglesias Luciano. *Inversión de prioridades: prueba de concepto y análisis de soluciones.* 2013.
6. Tarek Helmy and Syed S. Jafri. Avoidance of priority inversion in real time systems based on resource restoration. *IJCSA*, 3(1):40–50, 2006.
7. Troy D. Hanson. A hash table for c structures. http://troydhanson.github.io/uthash/index.html, 2006.
8. David R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
9. Peter C. Dibble. *Real-Time Programming with the Java Platform.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.