Implementação e Avaliação do Protocolo de Sincronização *Immediate Priority Ceiling* no Kernel do Linux

Andreu Carminati[1], **Rômulo Silva de Oliveira**[1], **Luís Fernando Friedrich**[2]

[1]Departamento de Automação e Sistemas – Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040–900 – Florianópolis – SC – Brasil

[2]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
88049–900 – Florianópolis – SC – Brasil

{andreu,romulo}@das.ufsc.br, {fernando}@inf.ufsc.br

Abstract. *In general purpose systems, such as the mainline Linux, priority inversions occur frequently and are not considered harmful, nor are avoided as in real-time systems. In the current version of the kernel PREEMPT-RT, the protocol that implements the priority inversion control is the Priority Inheritance. The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling, for use in drivers dedicated to real-time applications, for example. This article explains how the protocol was implemented in the real-time kernel and compare tests on the protocol implemented and Priority Inheritance, currently in use in the real-time kernel.*

Resumo. *Em sistemas de propósito geral, como o mainline Linux, inversões de prioridades ocorrem frequentemente e não são consideradas nocivas, e nem são evitadas como em sistemas de tempo real. Na versão atual do kernel PREEMPT-RT o protocolo que implementa o controle de inversões é o Priority Inheritance. O objetivo deste trabalho é propor a implementação de um protocolo alternativo, o Immediate Priority Ceiling, para uso em drivers dedicados a aplicações de tempo real, por exemplo. Este artigo explica como o protocolo foi implementado no kernel de tempo real e compara testes feitos sobre o protocolo implementado e o Herança de Prioridade atualmente em uso no kernel de tempo real.*

1. Introdução

Em sistemas [1] de tempo real, como Linux/*PREEMPT-RT* [Molnar , Rostedt and Hart 2007], mecanismos de sincronização de tarefas devem garantir tanto a manutenção da consistência interna do sistema em relação a recursos ou estruturas de dados, quanto determinismo no tempo de espera por estes. Eles devem evitar inversões de prioridades descontroladas, onde uma tarefa de alta prioridade fica bloqueada indefinidamente a espera de um recurso que está em posse de uma tarefa de mais baixa prioridade.

Em sistemas de propósito geral, como o *mainline* Linux, inversões de prioridades ocorrem frequentemente e não são consideradas nocivas, e nem são evitadas como em sistemas de tempo real. Na versão atual do kernel *PREEMPT-RT* o protocolo que implementa o controle de inversões é o *Priority Inheritance* (PI) [Sha et al. 1990].

O objetivo deste trabalho é propor a implementação de um protocolo alternativo, mais especificamente, o *Immediate Priority Ceiling* (IPC) [Lampson and Redell 1980, Sha et al. 1990], para uso em *drivers* dedicados a aplicações de tempo real, por exemplo. Não é objetivo deste trabalho propor uma substituição do protocolo existente, como dito anteriormente, mas uma alternativa para uso em algumas situações.

Este trabalho está organizado da seguinte forma: a seção 2 aborda o panorama atual da sincronização no kernel *mainline* e o *PREEMPT-RT*, a seção 3 explana sobre protocolo *Immediate Priority Ceiling*, a seção 4 explica como o protocolo foi implementado no kernel de tempo real do Linux e a seção 5 compara testes feitos sobre o protocolo implementado e o Herança de Prioridade implementado no kernel de tempo real.

2. A Exclusão Mútua no Kernel do Linux

Como no kernel *mainline* não existem mecanismos que evitem o aparecimento de inversões de prioridades, situações como a apresentada na figura 1 podem ocorrer muito facilmente, onde a tarefa T2, ativada em *t=1*, adquire o recurso compartilhado. Em seguida, a tarefa T0 é ativada em *t=2* mas bloqueia no recurso detido por T2. T2 retoma sua execução, quanto é preemptada por T1, que é ativada e começa a executar de *t=5* a *t=11*. Mas a tarefa T0 perde o *deadline* em *t=9*, e o recurso necessário para o seu término só foi disponibilizado em *t=12* (após o *deadline*).

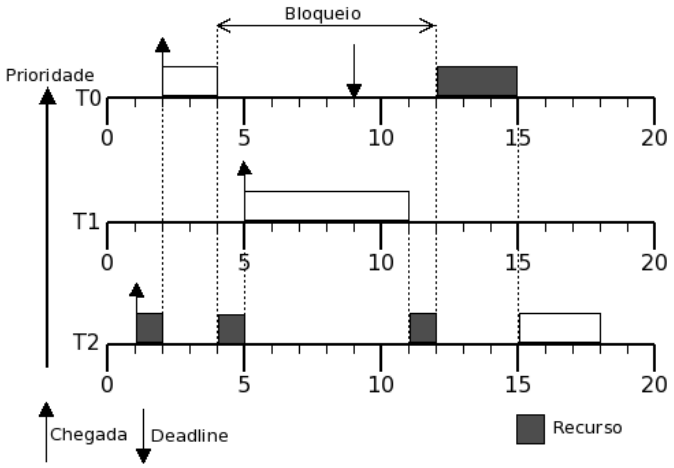

Figura 1. Inversão de prioridade

Já no kernel de tempo real *PREEMPT-RT* existe a implementação do PI. Como dito anteriormente, é um mecanismo utilizado para acelerar a liberação de recursos em sistemas de tempo real, bem como evitar o efeito de atraso indefinido de tarefas de alta prioridade que podem estar bloqueadas a espera de recursos detidos por tarefas de mais baixa prioridade.

No protocolo PI, uma tarefa de alta prioridade que é bloqueada em algum recurso, cede sua prioridade para a tarefa de mais baixa prioridade (que detêm este recurso), para que esta libere o recurso sem sofrer preempções por tarefas de prioridade intermediária. Este protocolo pode gerar encadeamento de ajustes de prioridade (uma sequência de ajustes em cascata) dependendo dos aninhamentos de seções críticas.

A figura 2 apresenta um exemplo de como o protocolo PI pode ajudar no problema da inversão de prioridade. Neste exemplo, a tarefa T2 é ativada em *t=1* e adquire um

recurso compartilhado, em *t=1*. A tarefa T0 é ativada e bloqueia no recurso detido por T2 em *t=4*. T2 herda a prioridade de T0 e impede que T1 execute, quando ativada em *t=5*. Em *t=6* a tarefa T2 libera o recurso, sua prioridade volta ao normal, e a tarefa T0 pode concluir sem perda de *deadline*. Alguns dos problemas [Yodaiken 2003] deste protocolo

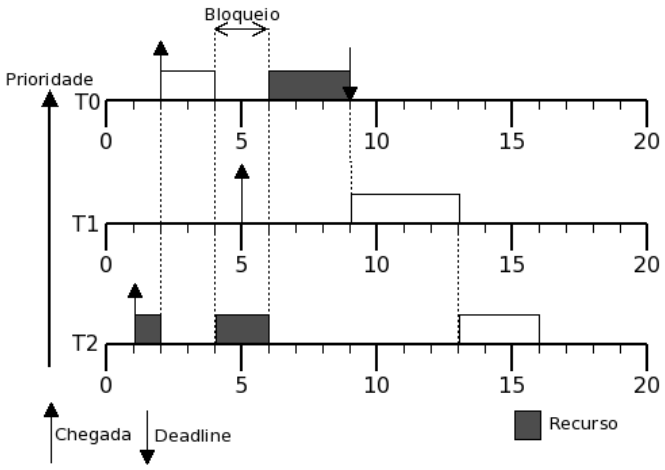

Figura 2. Inversão de prioridade resolvida por PI

são os excessivos chaveamentos de contexto e um tempo de espera maior que a maior das seções críticas [Sha et al. 1990] (para a tarefa de alta prioridade), dependento do padrão de chegadas do conjunto de tarefas que compartilham determinados recursos. Induzindo assim a formação de cadeias de bloqueios possivelmente longas.

A figura 3 é um exemplo onde o protocolo PI não impede a perda do *deadline* da tarefa de mais alta prioridade. Neste exemplo, ocorre o aninhamento de seções críticas, onde T1 (a de média prioridade) possui as seções críticas dos recursos 1 e 2 aninhadas. Ainda neste exemplo, a tarefa T0, quando bloqueada no recurso 1 em *t=3*, cede sua prioridade para a tarefa T1, que também bloqueia no recurso 2 em *t=3*. T1 por sua vez cede sua prioridade para a tarefa T2, que retoma sua execução e libera o recurso 2 permitindo que T2 também utilize este último e libere o recurso 1 para T0 em *t=7*. T0 retoma sua execução mas não consegue executar sem perda de *deadline*, que ocorre em *t=11*.

3. O Protocolo *Immediate Priority Ceiling*

O Protocolo de 3 sincronização para prioridade fixa *Immediate Priority Ceiling* (IPC) [Burns and Wellings 2001], é uma variação do *Protocolo Priority Ceiling* [Sha et al. 1990, Baker 1990] ou *Highest Priority Locker*. Trata-se de um mecanismo alternativo para prevenção de inversões de prioridade descontroladas, bem como prevenção de *deadlocks* em sistemas monoprocessados.

No protocolo PI, a prioridade está associada somente às tarefas. No IPC a prioridade está associada tanto às tarefas quanto aos recursos. Um recurso protegido por IPC possui uma prioridade de teto (ou *ceiling*), e esta prioridade representa a maior das prioridades entre todas as prioridades das tarefas que acessam este recurso.

Segundo [Harbour and Palencia 2003], o tempo máximo de bloqueio para uma tarefa sob prioridade fixa utilizando recurso compartilhado pelo protocolo IPC é a maior

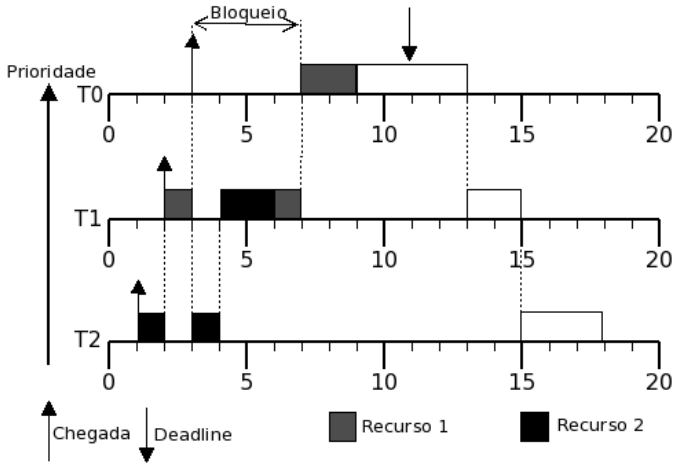

Figura 3. Inversão de prioridade não resolvida por PI

seção crítica do sistema cuja prioridade teto seja mais alta que a prioridade da tarefa em questão e seja utilizada por uma tarefa de prioridade mais baixa que ela.

O que ocorre no IPC pode ser considerado como uma herança preventiva, onde a prioridade é ajustada imediatamente quando ocorre a aquisição do recurso, e não quando o recurso se tornar necessário a uma tarefa de alta prioridade, como ocorre no PI (pode-se pensar no PI como sendo o IPC, mas com o ajuste dinâmico de teto). Este ajuste preventivo evita que tarefas de baixa prioridade sejam preemptadas por tarefas com prioridades intermediárias, as quais possuem prioridade superior a de baixa e inferior a prioridade teto do recurso.

A figura 4 mostra um exemplo semelhante ao mostrado na figura 3, mas desta vez utilizando IPC. Neste exemplo, a tarefa de alta prioridade não perde o *deadline*, pois quando a tarefa T2 adquire o recurso 2 em *t=1*, sua prioridade é elevada à prioridade teto deste (prioridade de T1), impedindo que a tarefa T1, ativada em *t=2* inicie sua execução. Em *t=3*, a tarefa T0 é ativada e inicia sua execução. A tarefa não mais é bloqueada, pois o recurso 2 está livre. A tarefa T0 não perde o *deadline*.

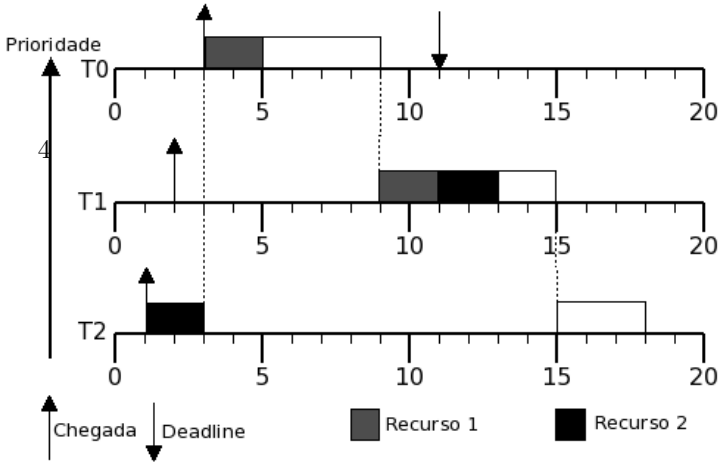

Figura 4. Inversão de prioridade resolvida por IPC

4. Descrição da Implementação

O Protocolo *Immediate Priority Ceiling* foi implementado tendo como base o código dos *rt_mutexes* existente no *PREEMPT-RT*. Os rt_mutexes são mutexes que implementam o protocolo de herança de prioridade (*Priority Inheritance*). A versão do kernel utilizada para implementação foi a 2.6.31.6 [Torvalds 2010] com o *patch PREEMPT-RT* rt19.

A implementação foi feita primariamente para uso em *device-drivers* (ou seja, em espaço de *kernel*), como mostra a figura 5, onde existe um exemplo de tarefas compartilhando uma seção crítica protegida por IPC e acessada através de uma *system call ioctl* em um *driver*. O tipo de dado que representa o protocolo IPC foi definido como *struct*

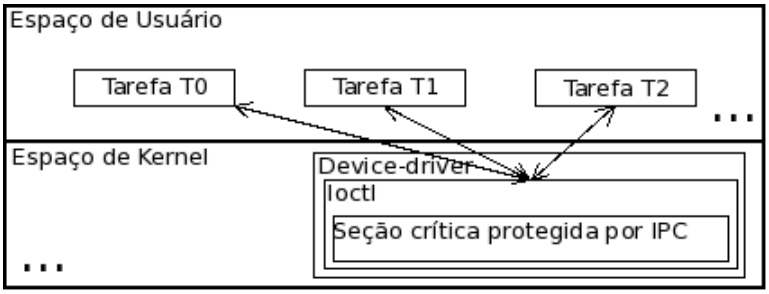

Figura 5. Diagrama de interação entre o protocolo IPC e tarefas

ipc_mutex, e é apresentado no código 1. Nesta estrutura, *wait_lock* é o *spinlock* que protege o acesso à estrutura, *wait_list* é a lista ordenada por prioridades que armazena pedidos de *lock* pendentes, *on_task_entry* serve para as tarefas fazerem o rastreamento dos *locks* adquiridos (e, por consequência, controle de prioridades), *owner* armazena um ponteiro para a tarefa dona do *mutex* (ou ponteiro nulo, caso a *mutex* esteja livre) e finalmente *ceiling*, que armazena a prioridade teto do *mutex*.

Código 1 Estrutura de dados que representa um mutex IPC

```
struct ipc_mutex {
        atomic_spinlock_t wait_lock;
        struct plist_head wait_list;
        struct plist_node   on_task_entry;
        struct task_struct *owner;
        int  ceiling;
...
};          5
```

A implementação proposta apresenta a seguinte API de funções e macros:

- **DEFINE_IPC_MUTEX(mutexname, prio):** Macro fornecida para definição estática de um mutex IPC, onde *mutexname* é o identificador do mutex e *prio* é a prioridade teto (ou *ceiling*) do mutex, ou seja, um valor no intervalo de 0 a 99 (segundo a especificação de prioridades estáticas para tempo real no Linux). Na versão atual, só podem ser criados mutexes com prioridades definidas em tempo de compilação, desta forma, a prioridade teto deve ser atribuída pelo projetista da aplicação.

- **void ipc_mutex_lock(struct ipc_mutex *lock):** Função de aquisição do mutex. Em sistemas monoprocessados esta função não é bloqueante pois, de acordo com o protocolo IPC, se uma tarefa chegar a solicitar o uso de um recurso é porque este está disponível (campo *owner* nulo). Já em sistemas multiprocessados esta função pode gerar bloqueios, pois o recurso pode estar sendo utilizado em outro processador (campo *owner* diferente de nulo). Mas, mesmo assim, pelo tempo máximo da maior seção crítica, levando em consideração todas as tarefas de mais baixa prioridade. Neste artigo, somente a versão monoprocessada será levada em consideração. O principal papel desta função é gerenciar a prioridade da tarefa que a chama juntamente com o bloqueio do recurso, levando em consideração todos os mutexes (*ipc_mutexes*) adquiridos até o momento.

- **void ipc_mutex_unlock(struct ipc_mutex *lock):** Efetua a liberação do recurso e o reajuste da prioridade da tarefa que o chama. Em sistemas multiprocessados, esta função também efetua o trabalho de selecionar a próxima tarefa (via campo *wait_list*) que utilizará o recurso. O que também ocorre em sistemas multiprocessados é o efeito "próximo dono de recurso pendente"(também presente na implementação original da herança de prioridade) também conhecido como roubo de bloqueio, onde a tarefa de mais alta prioridade pode adquirir o recurso novamente, mesmo este tendo sido atribuído a outra tarefa que ainda não tomou posse deste.

Uma diferença marcante entre implementação proposta e aquela já existente no *PREEMPT-RT* está no fato da última viabilizar as seguintes otimizações:

- PI pode realizar *lock* atômico: quanto uma tarefa tenta adquirir um recurso que está livre, esta operação pode ser executada atomicamente (operação *atomic compare and exchange*) pelo que é conhecido como *fast path*. Mas isto apenas é possível para arquiteturas que possuam este tipo de operação atômica. Caso contrário (o *lock* está ocupado e/ou a arquitetura não possui instrução *exchange and compare*) o lock sempre será não atômico (*slow path*).

- PI pode realizar *unlock* atômico: quando uma tarefa libera um recurso o qual não possui tarefas em espera, esta operação também pode ser executada atomicamente.

Como foi dito anteriormente, estas otimizações são possíveis apenas para os mutexes com PI. No caso do IPC, sempre haverá a necessidade de uma verificação e um possível ajuste de prioridade.

5. Análise da Implementação

Como recurso compartilhado entre as tarefas, foi desenvolvido um *device-driver* que tem a função de prover as seções críticas necessárias à execução dos testes. Este *device-driver* exporta um único serviço, que é uma chamada de serviço *ioctl* (mais especificamente *unlocked_ioctl*, pois o *ioctl* tradicional é protegido pelo antigo *Big Kernel Lock*, que com certeza impediria a execução correta dos testes), que multiplexa as chamadas das três

tarefas em suas seções criticas correspondentes. Este *device-driver* fornece as seções críticas tanto para execução com IPC quanto para PI.

Para realização de testes que permitam a análise da implementação, foi utilizado um conjunto de tarefas esporádicas executadas em espaço de usuário. O intervalo entre ativações, os recursos utilizados, bem como o tamanho da seção crítica dentro do *device-driver* usada por cada tarefa são apresentados na tabela 1. Todas as seções críticas são executadas dentro da função *ioctl*, no espaço de kernel do Linux. Um resumo em alto nível das ações executadas em cada tarefa (em relação aos recursos utilizados) é apresentado na tabela 2.

A tabela 1 apresenta os intervalos entre ativações expressos com uma pseudo-aleatoriedade, ou seja, com valores distribuidos uniformemente entre valores mínimos e máximos. Esta aleatoriedade foi inserida nos testes para melhorar a distribuição dos resultados, pois com períodos fixos, os padrões de chegadas estariam sendo limitados a um conjunto muito mais restrito. Ainda na tabela 1 são apresentados os tamanhos das seções críticas de cada uma das tarefas. Outra informação mostrada na tabela 1 é o número de ativações efetuadas por cada tarefa. Para a tarefa de alta prioridade, houveram 1000 ativações monitoradas (latências, tempo de resposta, tempo de seção crítica, tempo de bloqueio, etc), e para as outras não houve restrição de ativações.

A tarefa de alta prioridade possui uma das prioridades mais altas do sistema. As outras tarefas foram consideradas como média e baixa prioridade em comparação com a de alta, pois também possuem prioridades elevadas. Todas as tarefas foram configuradas com a política de escalonameto SCHED_FIFO, que é uma das políticas para tempo real [IEEE 1998] presentes no Linux.

Mesmo com a utilização de uma máquina multiprocessada para testes, as tarefas foram fixadas em apenas uma CPU (CPU 0). Os testes foram realizados tanto com IPC quanto PI para fins comparativos.

Tabela 1. Configuração do conjunto de tarefas

Tarefa	Prio.	Intervalo ativ.	Recursos	Tam. seção crit.	N. ativações
T1	70	aleat[400,800]ms	R1	aprox. 17ms	1000
T2	65	aleat[95,190]ms	R1,R2	aprox. 2x17ms	sem limite
T3	60	aleat[85,170]ms	R2	aprox. 17ms	sem limite

Tabela 2. Ações realizadas pelas tarefas

Tarefa	Ação 1	Ação 2	Ação 3	Ação 4	Ação 5	Ação 6
Alta	L(R1)	Seção Crít.	U(R1)			
Média	L(R1)	Seção Crít.	L(R2)	Seção Crít.	U(R2)	U(R1)
Baixa	L(R2)	Seção Crít.	U(R2)			

O mutex R1 foi configurado com prioridade teto 70 (que é a prioridade da tarefa T0) e o R2 foi configurado com a prioridade 65 (que é a prioridade da tarefa T1)

5.1. Resultados da Utilização do mutex com PI

Com herança de prioridade, a tarefa de alta prioridade apresentou latências de ativação aproxidamente (como pode ser observado no histograma da figura 6) constantes no intervalo [20000, 30000] nanosegundos. Entretanto a tarefa, pelo fato de encontrar o recurso ocupado com certa frequência (como mostra o gráfico da figura 7, onde a tarefa apresenta tempos de espera pelo recurso (*lock*) bem variados), é obrigada a executar chaveamentos de contexto voluntários para propagação de prioridade ao longo da cadeia de bloqueios.

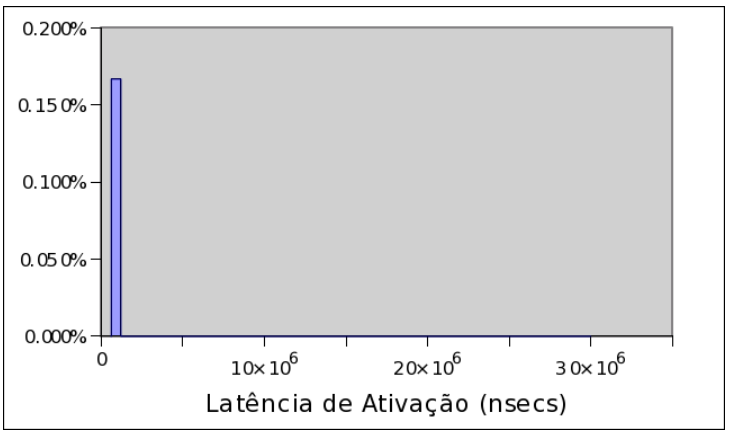

Figura 6. Histograma das latências de ativação (alta prioridade utilizando PI)

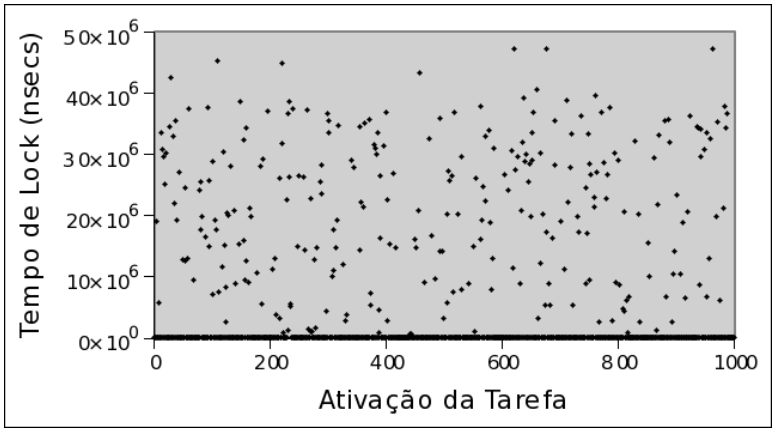

Figura 7. Gráfico dos tempos de bloqueio (tarefa de alta prioridade utilizando PI)

Em relação ao tempo de resposta (como pode ser observado no histograma da figura 8), este foi coerente com os tempos de bloqueios sofridos, com valor máximo com quase 3 vezes o tamanho da seção crítica, de acordo com o conjunto de tarefas de teste. Ainda pode-se observar na tabela 3 o pior caso no tempo de resposta máximo observado como sendo 64157591 ns. O pior caso teórico no tempo de resposta para este teste seria, com as devidas ativações sincronizadas, 68 ms, ou seja, 17 ms seção crítica de própria tarefa T0 adicionado de 34 ms da tarefa T1, mais 17 ms da tarefa T2. Neste teste, houve uma boa aproximação do limite teórico.

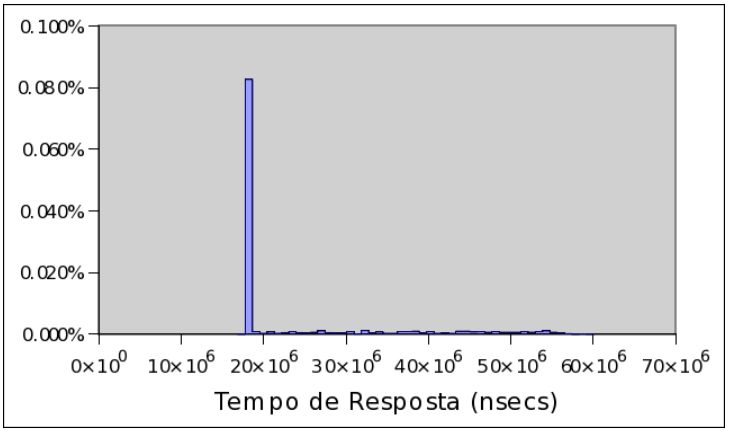

Figura 8. Histograma do tempo de resposta (alta prioridade utilizando PI)

Tabela 3. Tempos médios de resposta e desvio padrão

Protocolo:	PI	IPC
Tempo médio de resposta :	22798549 ns	21014311 ns
Desvio padrão:	11319355 ns	8723159 ns
Máx:	64157591 ns	50811328 ns

5.2. Resultados da Utilização do mutex com IPC

Utilizando IPC, pode-se notar no histograma da figura 9 que a tarefa de alta prioridade apresentou, com baixa frequência, valores variados de latência (aparente na cauda deste histograma) de ativação. Os tempos de espera pelo recurso aparecem constantes na figura 10, o que é o esperado segundo a definição do protocolo implementado. Já no histograma do tempo de resposta da figura 11, aparece uma cauda (valores mais altos, mas com baixas ocorrências) devido à latência de ativação sofrida.

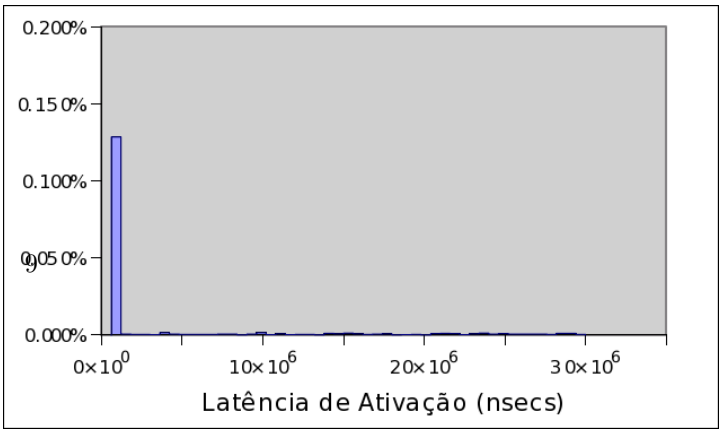

Figura 9. Histograma da latências de ativação (alta prioridade utilizando IPC)

Também neste teste, pode-se observar na tabela 3 o pior caso no tempo de resposta máximo observado como sendo 50811328 ns. Neste teste, o limite teórico seria 51 ms, ou seja, 17 ms seção crítica de própria tarefa T0 adicionado de 34 ms da tarefa T1. Também neste teste houve uma boa aproximação do limite teórico.

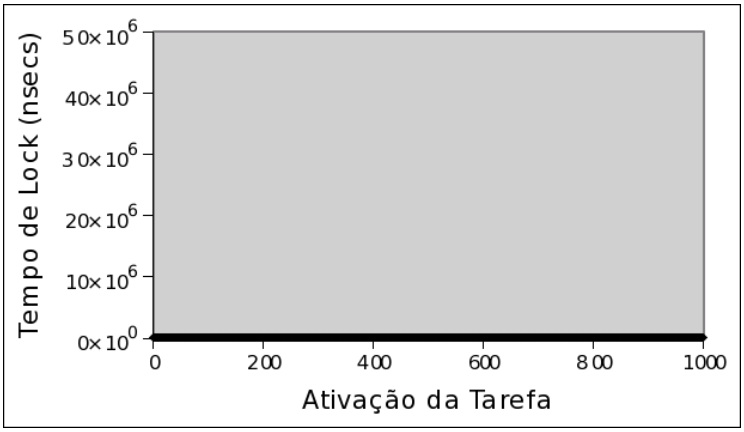

Figura 10. Gráfico dos tempos de bloqueio (alta prioridade Utilizando IPC)

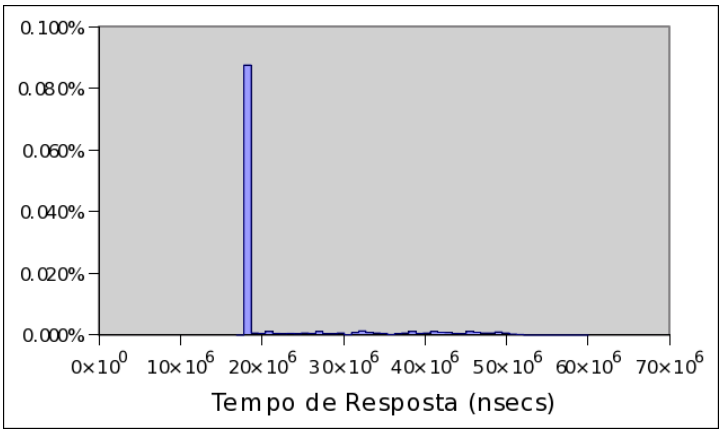

Figura 11. Histograma do tempo de resposta (alta prioridade utilizando IPC)

5.3. Comparação do PI com o IPC

O que se pode observar é que, de forma geral, protocolo IPC possui comportamento semelhante ao PI.

As diferenças aparecem no tempo de *lock* onde, por definição, em sistemas monoprocessados, o recurso estará sempre disponível quando requisitado segundo protocolo IPC. Em relação ao protocolo PI, o tempo de bloqueio aparecerá no tempo da primitiva de *lock*, e este tempo poderá ser mais extenso que no IPC, pois neste o tempo aparecerá antes da ativação, e terá tamanho máximo de uma seção crítica (nas condições apresentadas anteriormente).

Segundo a tabela 3, o protocolo IPC apresentou o desvio padrão e o tempo médio de resposta menor que o PI. Outro ponto importante na tabela 3 é que o pior caso no tempo de resposta observado nos testes do protocolo IPC foi quase uma seção crítica menor que o do PI (o tamanho de uma seção crítica é de 17 ms, e a diferença do pior caso entre IPC e PI está em torno de 14 ms)

A figura 12 apresenta o histograma da parte não constante (relativa às ativações da tarefa onde houve bloqueio). Nesta figura, os tempos de resposta do protocolo IPC se concentraram em valores menores, e no PI, estes se distribuiram mais uniformemente até

valores maiores, indicando como citado anteriormente, tempo médio de resposta menor para o protocolo IPC. Este histograma também indica em sua porção final que o pior caso, como também foi observado na tabela 3, apresenta uma diferença de uma seção crítica no tempo de resposta contando a favor do protocolo IPC. Esta diferença no pior caso foi assinalada na figura com duas linhas verticais, onde a distância entre estas representa aproximadamente uma seção crítica. A tabela 4 resume qualitativamente os resultados encontrados.

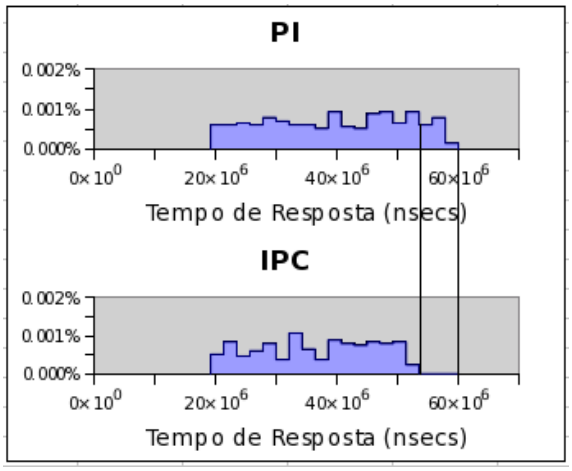

Figura 12. Histograma do tempo de resposta da tarefa de alta prioridade

Tabela 4. Resumo dos resultados esperados

Protocolo	PI	IPC
Latência de Ativação	Não variada	Variada
Tempo de Bloqueio	Variado	Não Variado
Tempo de Resposta	Dependente do Tempo de Bloqueio	Dependente da Latência

6. Conclusões

Sincronização de tarefas é algo fundamental em sistemas multitarefa e/ou *multithread*, ainda mais em sistemas de tempo real. Além de proteção contra condições de corrida, estes mecanismos devem evitar o aparecimento de inversões de prioridades descontroladas, que poderiam causar perdas de *deadline*, induzindo aplicações de tempo real a apresentarem comportamento falho e possivelmente nocivo (dependendo da aplicação). Neste contexto, foi proposto uma alternativa (para certas aplicações) ao protocolo presente na árvore de tempo real do Linux.

O Protocolo IPC pode ser mais adequado para aplicações dedicadas, que ulitizem arquiteturas sem instrução *compare and exchange*, pois desta forma, a implementação do kernel de tempo real não poderá fazer uso do caminho rápido (via instruções atômicas), equivalendo assim, o *overhead* do código executado na implementação de PI e IPC (mas não o *overhead* por bloqueios e latências). Outra vantagem do IPC é que este efetua menos chaveamentos de contexto que o PI, induzindo tempos de resposta menores, devido ao próprio *overhead* do chaveamento e também a menores taxas de falhas na TLB.

Uma das desvantagens do IPC para utilização mais geral é a necessidade de determinação manual da prioridade teto do *mutex* IPC. Mas isto não é um problema para aplicações no contexto de controle e automação por exemplo, onde um *device-driver* dedicado tem pleno conhecimento das prioridades das tarefas que o acessam, sendo justificável a definição manual do teto para este caso.

Como visto nos testes, se a questão de latência de ativação for relevante, o protocolo PI pode ser mais adequado, mas se o tempo de *lock* deve ser constante, IPC pode ser a melhor solução. Em termos de tempo de resposta, as duas soluções apresentaram comportamento semelhante, mas o IPC mesmo assim apresentou tempo médio de resposta menor, provavelmente devido a latência de ativação ser menor que o tempo de espera do PI. Outro ponto a favor do protocolo IPC aparece quando se observa a diferença no pior caso do tempo de resposta observado nos testes, pois no IPC foi cerca de uma seção crítica menor que no PI, como pode ser visto também na tabela 3. O protocolo PI possui um tempo de resposta que pode variar em dependência do padrão de compartilhamento de recursos e sequências de ativação, o que nao ocorre com o IPC. Neste, este tempo será sempre de no máximo uma seção crítica.

Como trabalho futuro, pretende-se implementar uma versão com teto adaptativo, ou seja, poder ser definido automaticamente em *run-time*. Também pretende-se ampliar o estudo do protocolo IPC para sistemas multiprocessados.

Referências

Baker, T. (1990). A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, volume 270.

Burns, A. and Wellings, A. (2001). *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley.

Harbour, M. and Palencia, J. (2003). Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 200. Citeseer.

IEEE, C. S., editor (1998). *POSIX.13. IEEE Std. 1003.13-1998. Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers.

Lampson, B. and Redell, D. (1980). Experience with processes and monitors in Mesa.

Molnar, I. Preempt-rt. http://www.kernel.org/pub/linux/kernel/projects/rt - Last access 01/21, 2009. [12]

Rostedt, S. and Hart, D. (2007). Internals of the RT Patch. In *Proceedings of the Linux Symposium*, volume 2007.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185.

Torvalds, L. (2010). "Linux Kernel Version 2.6.31.6". http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.31.6.tar.bz2 - Last access 03/21, 2010.

Yodaiken, V. (2003). Against priority inheritance. *FSMLABS Technical Paper*. disponível em `http://yodaiken.com/papers/inherit.pdf`.