Andreu Carminati

Um Mecanismo de Sincronização para o Kernel do Linux para Aplicações de Tempo Real

Florianópolis - SC, Brasil

5 de Novembro de 2010

Andreu Carminati

Um Mecanismo de Sincronização para o Kernel do Linux para Aplicações de Tempo Real

Monografia apresentada para obtenção do Grau de Bacharel em Ciências da Computação pela Universidade Federal de Santa Catarina.

Orientador:
Rômulo Silva de Oliveira

Co-orientador:
Luís Fernando Friedrich

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis - SC, Brasil

5 de Novembro de 2010

Monografia de Trabalho de Conclusão de Curso sob o título *"Um Mecanismo de Sincronização para o Kernel do Linux para Aplicações de Tempo Real"*, defendida por Andreu Carminati e aprovada em 5 de Novembro de 2010, em Florianópolis, Estado de Santa Catarina, pela banca examinadora constituída pelos professores:

Prof. Dr. Rômulo Silva de Oliveira
Orientador

Prof. Dr. Luís Fernando Friedrich
Co-orientador

Prof. Dr. José Mazzuco Júnior
Universidade Federal de Santa Catarina

Resumo

Em sistemas operacionais modernos (com *multithreading* e/ou *multitasking*), mecanismos de sincronização de processos são essenciais para a manutenção da consistência interna do sistema, dado que estes garantem a coordenação quanto ao compartilhamento de recursos ou estruturas de dados, evitando assim, o conhecido efeito chamado condição de corrida (ou *race conditions*). Este trabalho trata do estudo do funcionamento e implementação dos mecanismos de sincronização no sistema operacional Linux e como estes impactam na execução de tarefas de tempo real. Nestes sistemas (incluindo 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 trabalho trata como o protocolo foi implementado no kernel de tempo real e compara testes feitos sobre o protocolo implementado e o *Priority Inheritance* atualmente em uso no kernel de tempo real. Também faz parte deste trabalho a estudo do *overhead* da implementação proposta.

Palavras chave: Linux, PREEMPT-RT, tempo real, sincronização de Processos.

Abstract

In modern operating systems (with multithreading and/or multitasking), process synchronization mechanisms are essential to maintaining the internal system consistency, since these ensures coordination regarding to resource sharing or data structures, thus avoiding the known effect called race condition. This work deals with the study of the operation and implementation of synchronization mechanisms in the Linux operating system and how these impact on the execution of real time tasks. In these systems (including 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 work 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 work 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. Also a part of this work, an *overhead* study of the proposed implementation.

Keywords: Linux, PREEMPT-RT, real-time, process synchronization.

Dedicatória

Dedico este trabalho a minha família, em especial aos meus pais Evanilde e Elecir e à minha namorada Morgana.

Agradecimentos

Agradeço ao Professor Rômulo, que sempre esteve prontamente disposto a discutir o andamento do trabalho e por suas minuciosas correções dos artigos que serviram como base para esta monografia. Também agradeço ao CNPq pelo suporte financeiro.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução p. 11

1.1 Objetivo Geral . . . . p. 12

1.2 Objetivos Específicos . . . . p. 12

1.3 Motivações . . . . p. 12

1.4 Organização do Trabalho . . . . p. 13

2 Fundamentação sobre o Kernel do Linux p. 14

2.1 Linux Para Tempo Real . . . . p. 14

2.1.1 Motivação Para a Utilização do Linux para Aplicações de Tempo Real p. 14

2.1.2 O Patch PREEMPT-RT . . . . p. 15

2.2 A Exclusão Mútua no Kernel do Linux . . . . p. 17

2.2.1 Mutex no kernel Mainline . . . . p. 17

2.2.2 Mutex no kernel PREEMPT-RT . . . . p. 18

2.3 Conclusões . . . . p. 20

3 O Protocolo Immediate Priority Ceiling p. 21

3.1 Priority Ceiling Clássico . . . . p. 21

3.2 Immediate Priority Ceiling . . . . p. 22

3.3 Conclusões . . . . p. 23

4 Proposta de Implementação do Protocolo Immediate Priority Ceiling p. 24

4.1 Descrição da Implementação . . . . p. 24

4.1.1 API Proposta . . . . p. 25

4.1.2 Otimização por Postergação de Ajuste de Prioridade . . . . p. 26

4.1.3 Mantenabilidade da Implementação . . . . p. 29

4.2 Conclusões . . . . p. 30

5 Avaliação do Protocolo Immediate Priority Ceiling Implementado p. 31

5.1 Definições Preliminares . . . . p. 31

5.2 Análise da Implementação . . . . p. 31

5.2.1 Resultados da Utilização do mutex com PI . . . . p. 33

5.2.2 Resultados da Utilização do mutex com IPC . . . . p. 35

5.2.3 Comparação do PI com o IPC . . . . p. 36

5.3 Overhead da Implementação Proposta . . . . p. 37

5.3.1 Definição do Overhead . . . . p. 37

5.3.2 Descrição do Experimento Utilizado . . . . p. 37

5.3.3 Resultados Obtidos . . . . p. 39

5.3.4 Avaliação Estatística dos Resultados . . . . p. 39

5.4 Conclusões . . . . p. 40

6 Conclusões p. 43

Referências Bibliográficas p. 45

Apêndice A – Artigo p. 47

Apêndice B – Códigos Fonte p. 64

Lista de Figuras

2.1 Inversão de prioridade . . . . p. 18

2.2 Inversão de prioridade resolvida por PI . . . . p. 19

2.3 Inversão de prioridade não resolvida por PI . . . . p. 20

3.1 Possível inversão de prioridade resolvida por IPC . . . . p. 23

4.1 Diagrama de interação entre o protocolo IPC e tarefas . . . . p. 24

5.1 Histograma das latências de ativação (alta prioridade utilizando PI) . . . . . . p. 33

5.2 Gráfico dos tempos de bloqueio (tarefa de alta prioridade utilizando PI) . . . p. 34

5.3 Histograma do tempo de resposta (alta prioridade utilizando PI) . . . . p. 34

5.4 Histograma das latências de ativação (alta prioridade utilizando IPC) . . . . . p. 35

5.5 Gráfico dos tempos de bloqueio (alta prioridade Utilizando IPC) . . . . p. 35

5.6 Histograma do tempo de resposta (alta prioridade utilizando IPC) . . . . p. 36

5.7 Histograma do tempo de resposta da tarefa de alta prioridade . . . . p. 37

Lista de Tabelas

5.1 Configuração do conjunto de tarefas . . . . p. 33

5.2 Ações realizadas pelas tarefas . . . . p. 33

5.3 Tempos médios de resposta e desvio padrão . . . . p. 34

5.4 Resumo dos resultados encontrados . . . . p. 37

5.5 Configuração do conjunto de tarefas do teste de *overhead* . . . . p. 38

5.6 Tempo de CPU da tarefa de medição e estatísticas relacionadas . . . . p. 41

5.7 Dados do teste t-Student . . . . p. 42

1 Introdução

Em sistemas operacionais modernos (com *multithreading* e/ou *multitasking*), mecanismos de sincronização de processos são essenciais para a manutenção da consistência interna do sistema, dado que estes garantem a coordenação quanto ao compartilhamento de recursos ou estruturas de dados, evitando assim, o conhecido efeito chamado condição de corrida ou *race conditions*. Também faz parte do assunto sincronização de processos, a ocorrência e detecção de *deadlocks*[1], mas estes estão fora do escopo deste trabalho.

O tema escolhido trata do estudo do funcionamento e implementação de mecanismos de sincronização no sistema operacional Linux (visto como sistema operacional de tempo real) e como estes impactam na execução de tarefas de tempo real com alta prioridade em cenários de sobrecarga e alta probabilidade de contenção por compartilhamento de recursos.

Sistemas operacionais de tempo real são sistemas com comportamento determinístico e dedicados a executar conjuntos de tarefas com diversos níveis de prioridade e com restrições temporais. Como restrições temporais podemos citar por exemplo, o período de uma tarefa periódica, seu tempo de computação e seu *deadline*[2]. O sistema deve garantir que nenhuma destas restrições venha a ser violada (em sistemas *hard real-time*), ou evitar ao máximo tais violações (em sistemas *soft real-time*). Sincronização neste tipo de sistema também é fundamental para o cumprimento de restrições temporais, e é nisto que este trabalho se foca.

Este trabalho apresenta o panorama geral de sincronização no kernel (e PREEMPT-RT) do Linux e também mostra como a exclusão mútua pode ser melhorada para algumas classes de aplicações de tempo real. Esta melhora será dada com a implementação de um novo mecanismo (ou protocolo) de sincronização dentro do kernel do Linux, o *Immediate Priority Ceiling*. Este

[1] Situação de impasse entre dois ou mais processos ficam bloqueados, em que um aguarda o outro para poder continuar sua execução.

[2] *Deadline* é o marco limite na escala de tempo para a execução de uma tarefa.

trabalho se foca no objetivo secundário dos mecanismos de sincronização em sistemas de tempo real, que é evitar o aparecimento de inversões de prioridade descontroladas.

1.1 Objetivo Geral

O objetivo geral deste trabalho é implementar um mecanismo de sincronização alternativo para o kernel de tempo real do Linux. Também compõe o objetivo geral a comparação da implementação proposta com o mecanismo equivalente já implementado no kernel.

1.2 Objetivos Específicos

Enquadram-se como objetivos específicos do trabalho realizado, a implementação de um mecanismo de sincronização para o kernel de tempo real (PREEMPT-RT) baseado no protocolo *Immediate Pryority Ceiling*. Como foi enunciado nos objetivos gerais, este protocolo será comparado com o protocolo existente, que é o *Priority Inheritance*, em termos de métricas conhecidas como, tempo de resposta, latência de ativação e outros. Também é um objetivo específico uma análise do *overhead* da implementação proposta.

1.3 Motivações

Atualmente, têm-se cada vez mais utilizado o Linux em aplicações de tempo real com diversos focos (aplicações embarcadas e financeiras por exemplo). Isto ocorre devido a todas as vantagens que este sistema moderno de propósito geral pode oferecer, como ambiente multitarefa, escalabilidade, pilha de rede, recursos gráficos, amplo suporte a praticamente todo tipo de *hardware*, estabilidade de código, evolução contínua e constantes atualizações para eliminação de falhas. Outra grande vantagem na utilização do Linux é a possibilidade de se poder estudar, alterar e fazer qualquer tipo de ajuste que se faça necessário para adequá-lo a uma determinada aplicação ou classe de aplicações. Por este motivo, estudar os mecanismos de sincronização é importante tanto para validá-los quanto para melhorá-los tendo em vista a utilização em soluções de tempo real. A escolha deste tema decorre também do fato de sincronização para tempo real ser um assunto bem conhecido na literatura, mas mesmo assim, representar questões em aberto no desenvolvimento da árvore de tempo real do Linux.

1.4 Organização do Trabalho

Este trabalho será organizado da seguinte forma: no capítulo 2 serão apresentadas de forma sucinta as modificações efetuadas no kernel para que este apresente comportamento adequado à tempo real. Este capítulo também descreve os mecanismos de sincronização para o problemas da exclusão mútua presentes no kernel, tendo em vista a versão de tempo real supracitada e o *mainline*[3]. O capítulo 3 explana sobre o protocolo de sincronização para tempo real *Immediate Priority Ceiling* (que será implementado) mas sem antes explicar sobre o seu precursor, o *Priority Ceiling* Clássico. O capítulo 4 apresenta a implementação proposta e uma variação otimizada. O capítulo 5 expõe uma comparação da implementação proposta com a já existente e um estudo do *overhead* da nova implementação. O Capítulo 6 apresenta as conclusões e considerações finais.

[3]Mainline é a versão padrão do kernel do Linux, sem nenhum tipo de *patch* aplicado.

2 *Fundamentação sobre o Kernel do Linux*

Este capítulo apresenta uma visão geral sobre Linux para tempo real e sobre a implementação do patch PREEMPT-RT, utilizado no trabalho. Também é mostrado como se comportam as primitivas de exclusão mútua tanto no kernel *mainline* quanto no PREEMPT-RT em relação ao problema da inversão de prioridades.

2.1 Linux Para Tempo Real

2.1.1 Motivação Para a Utilização do Linux para Aplicações de Tempo Real

Atualmente, existe uma vasta gama de sistemas operacionais de tempo real dedicados, como QNX Neutrino (KRTEN, 2009), VxWorks (RIVER, 2005) e muitos outros. No entanto, estes sistemas são extremamente restritos no que diz respeito aos serviços que podem prover para as tarefas (*Runtime*), o que muitas vezes pode não ser o suficiente para determinadas classes de aplicações. Provavelmente, estes recursos estarão disponíveis mais facilmente em Sistemas de Propósito Geral, entretanto serão providos de forma não determinística (ou sem nenhum tipo de garantia temporal), o que dificulta ou torna arriscada a utilização destes. O ponto principal é que, hoje em dia, está cada vez maior o interesse em se ter características de um SOPG (Sistema Operacional de Propósito Geral) em um SOTR (Sistema Operacional de Tempo Real) dada a vasta gama de serviços que este pode fornecer. Uma opção construtiva seria adicionar recursos de tempo real em SOPG's já existentes e bem conhecidos (API bem difundida, como por exemplo sistemas que implementam o padrão POSIX (IEEE, 1998), tendo assim um *framework* único de desenvolvimento). Esta prática é comum, mas dependendo da complexidade do sistema em questão, torna-se difícil (na grande maioria impossível) a análise formal deste (no sentido de prover algum tipo de garantia temporal). Desta forma, o kernel do Linux entra em foco, por ser um sistema multiplataforma, robusto e com uma extensa comunidade de desenvolvedores

e patrocinadores. Sistemas de tempo real baseados em SOPG são úteis tanto para sistemas embarcados quanto para aplicações corporativas (como por exemplo aplicações bancárias ou de mercado financeiro, que lidam com transações com algum tipo de restrição temporal) mesmo como supracitado, tendo sua validação formal dificultada.

Neste contexto, tanto em indústrias quanto em centros acadêmicos de pesquisa, têm-se utilizado o Linux como plataforma tanto para *soft* quando para *hard* real-time. Utilizando para isto, modificações (*patches*[1]) ou *nanokernels*[2] que permitam execuções de aplicações com garantias temporais (como RTAI (DOZIO; MANTEGAZZA, 2003) e XENOMAI (GERUM,)), ou ao menos, estatisticamente determinísticas (como PREEMPT-RT (MOLNAR, ; ROSTEDT; HART, 2007) que será apresentado na próxima seção).

2.1.2 O Patch PREEMPT-RT

O PREEMPT-RT é um *patch* para o *kernel* do Linux, mantido por Ingo Molnar, que tem como um dos principais objetivos permitir a execução determinística de tarefas com alta prioridade no *kernel* do Linux, através de um *kernel* com baixas latências e totalmente preemptável (com exceção do código de interrupção – dependente de arquitetura). As principais alterações feitas no kernel *mainline* para que isto seja possível são:

- Temporização de alta resolução (GLEIXNER; MOLNAR, 2006): *Timers* baseados em temporização de hardware, e não mais em jiffies[3], o que garante alta precisão devido a granularidades de tempo menores. Na implementação de temporização de alta resolução, existe o conceito de sistema *tickless* (ou NOHZ como é conhecido na documentação do kernel), ou seja, não existe o conceito de *tick* periódico. Neste contexto, a próxima interrupção de *timer* é sempre configurada para o próximo evento de interesse (*timeouts* e *timers*). Também foi feita a separação de temporizadores de baixa resolução (basicamente *timeouts*) que continuam a ser gerenciados como nas versões antigas do kernel, através de calendários [4], dos temporizadores de alta resolução, gerenciados através de árvores

[1]Patches são arquivos que, quando aplicados ao código fonte do Linux, introduzem algum tipo de mudança

[2]Nanokernel é geralmente definido com uma camada de virtualização ou camada de abstração de hardware (*HAL - hardware abstraction layer*)

[3]Jiffie é caracterizado como a contagem dos *ticks* periódicos, ou seja um contador incrementado a uma frêquencia fixa

[4]Calendário é o tipo de estrutura de dados utilizada no kernel, e que possui apenas o tempo de remoção constante (*O(1)*)

red-black (com inserções e remoções em tempo determinístico).

- Conversão de spinlocks em rt-mutexes (ROSTEDT, 2006) (nem todos foram modificados, aqui entram técnicas de compilação que convertem spinlocks em mutexes. Alguns locks são mantidos como spinlocks normais (raw_spinlock_t ou atomic_spinlocks nas versões mais recentes do kernel) como os do sistema de sincronização RCU (read-copy-update) (MCKENNEY et al., 2001), os spinlocks presentes no código de escalonamento, entre outros.

- Protocolo de herança de prioridade: O protocolo se tornou viável com a substituição da antiga API de semáforos por mutexes. Embora o kernel padrão tenha implementado herança de prioridade, esta está restrita ao FUTEXES, que são mutexes para *userspace*, ou seja, o código não é utilizado diretamente nas estruturas do kernel em si. O código que utiliza os FUTEXES pode ser visto na implementação dos mutexes no *pthreads* para Linux por exemplo.

- Threaded Interrupt Service Routines: (é feita de forma transparente ao programador de *device-drivers*). Desta maneira, os tratadores de interrupção passam a ser entidades escalonáveis, o que significa que não preemptarão (não causarão interferência (CARMINATI; OLIVEIRA, 2009) na terminologia de tempo real) tarefas de tempo real com alta prioridade. Para cada tratador, existe uma *Thread* associada.

- Threaded Softirqs[5]: Do mesmo modo que os tratadores de interrupção, as *softirqs* (e por consequência as *tasklets*) passam também a ser executadas em *Threads* de *kernel*, ou seja, também serão entidades escalonáveis do sistema, sendo assim propagarão consideravelmente menos interferência, contribuindo significativamente no determinismo de tarefas de tempo real com alta prioridade. Com o PREEMPT-RT, cada *softirq* possuirá uma *Thread* por processador (também com prioridade de tempo real sob a política SCHED_FIFO) por CPU, e nota se que, por serem Threads, políticas de escalonamento e prioridades poderão ser ajustadas conforme a necessidade.

[5]Softirqs (LOVE, 2005) são mecanismos de postergação de trabalho, que normalmente seria executado em tratadores de interrupção. O sua motivação foi a necessidade de se obter tempos de resposta menores para tais tratadores

- Workqueues com prioridade associada.

Com estas alterações, o kernel do Linux se torna altamente preemptivo e com *paths* de códigos não preemptáveis muito menores em relação ao kernel. Por padrão, o PREEMPT-RT mesmo já sendo um projeto maduro, ainda não vem totalmente inserido no *kernel mainline*.

2.2 A Exclusão Mútua no Kernel do Linux

Nesta seção, será apresentado o mecanismo de sincronização de alto nível (mutex) presente no kernel atual, e em sua variação para tempo real, o PREEMPT-RT. Mutexes são mecanismos possivelmente bloqueantes para o problema da exclusão mútua.

2.2.1 Mutex no kernel Mainline

Os mutexes no kernel *mainline* são implementados de forma tradicional, com bloqueio de tarefas por ordem de chegada. Foram propostos basicamente para substituir os semáforos, visto que estes eram muito frequentemente utilizados para exclusão mútua. Quando se diz que os mutexes foram implementados de forma tradicional, refere-se ao fato de não existir mecanismos que evitem o aparecimento de inversões de prioridades.

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 (PREEMPT-RT Linux).

Situações como a apresentada na figura 2.1 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*).

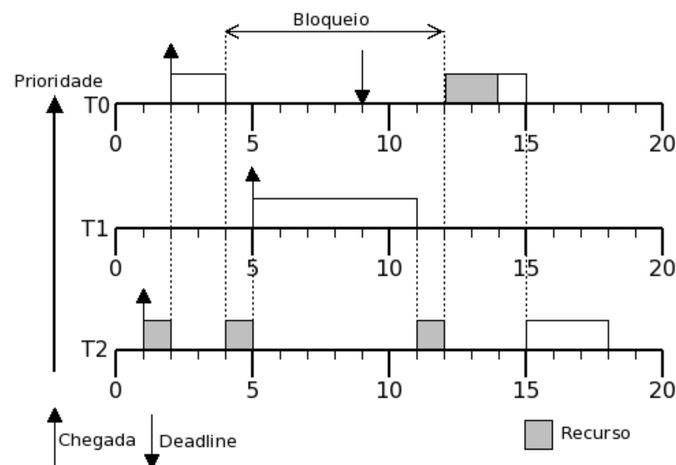

Figura 2.1: Inversão de prioridade

2.2.2 Mutex no kernel PREEMPT-RT

Já no kernel de tempo real PREEMPT-RT existe a implementação de mutexes com *Priority Inheritance* (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 (inversão de prioridade).

O Protocolo Priority Inheritance

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 2.2 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 o recurso compartilhado 1, 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*.

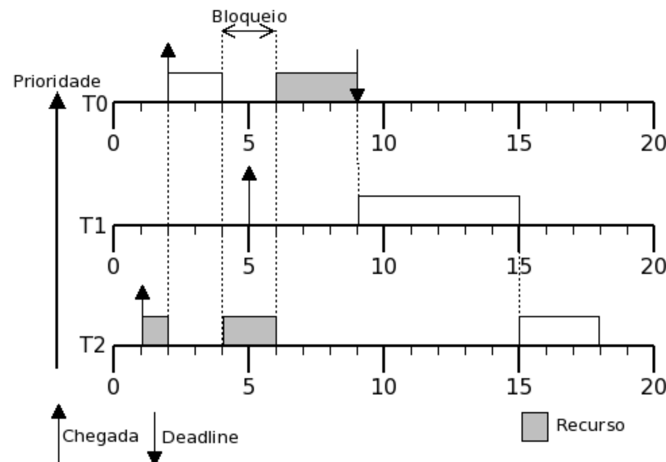

Figura 2.2: Inversão de prioridade resolvida por PI

Alguns Problemas Conhecidos do Protocolo PI

Alguns dos problemas (YODAIKEN, 2003) deste protocolo são os excessivos chaveamentos de contexto e um tempo de espera maior que a maior das seções críticas (SHA; RAJKUMAR; LEHOCZKY, 1990) (para a tarefa de alta prioridade), dependendo do padrão de chegadas do conjunto de tarefas que compartilham determinados recursos. Induzindo assim a formação de cadeias de bloqueios possivelmente longas.

Figura 2.3 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 tarefa de prioridade intermediária) têm as seções críticas dos recursos 1 e 2 aninhadas. Neste exemplo, a tarefa T0 quando bloqueado no recurso 1 em *t=5*, cede a sua prioridade para a tarefa T1, que também bloqueia no recurso 2 em *t=6*. T1 por sua vez, cede sua prioridade para a tarefa T2, que retoma a sua execução e libera o recurso 2, permitindo T1 usar esse recurso e em seguida liberar o recurso 1 para a tarefa T0 em *t=10*. T0 retoma a sua execução, mas perde o *deadline*, que ocorre em *t=13*. Neste exemplo, T0 tarefa foi bloqueado por uma parte do tempo da seção de crítica externa de T1 mais uma parte do tempo da seção crítica do T2. Em um sistema maior o tempo de bloqueio de T0 seria, no pior caso, a soma de muitas seções críticas, muitas delas não associados a T0.

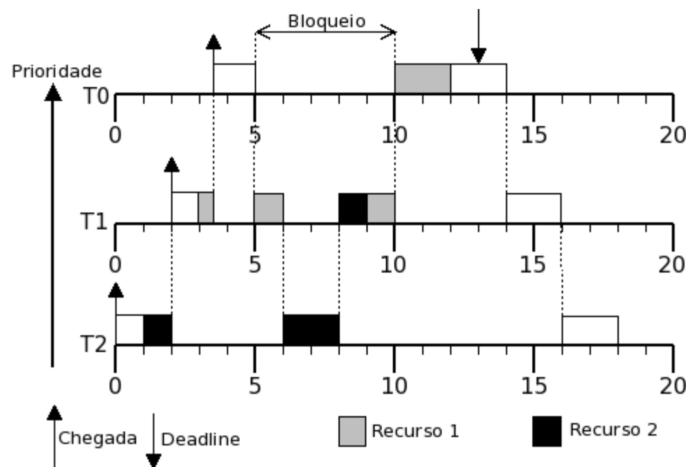

Figura 2.3: Inversão de prioridade não resolvida por PI

2.3 Conclusões

O Linux vêm se destacando cada vez mais como alternativa de sistema operacional para aplicações de tempo real. Existem duas formas de se obter sistemas determinísticos a partir do Linux: através de *nanokernels* ou através melhorias no próprio sistema (escalonamento, sincronização). Este trabalho se baseia no *patch* PREEMPT-RT que se enquadra na segunda forma citada anteriormente. O *patch* PREEMPT-RT introduz uma série de alterações no kernel do Linux para que este se torne altamente preemptivo, com temporização precisa e com controle de inversões de prioridade.

Inversões de prioridade ocorrem naturalmente em sistemas operacionais de propósito geral, como o Linux *mainline*. No kernel de tempo real (PREEMPT-RT/Linux), o controle de inversões de prioridade é feito através do protocolo *Priority Inheritance*. No entanto, este protocolo apresenta alguns problemas, como por exemplo, tempos de bloqueio possivelmente longos e excessivos chaveamentos de contexto.

3 *O Protocolo Immediate Priority Ceiling*

Este capítulo explana sobre o protocolo de sincronização para tempo real *Immediate Priority Ceiling*, mas antes explica um pouco sobre o seu precursor, o *Priority Ceiling* Clássico.

3.1 Priority Ceiling Clássico

É um protocolo para sincronização de tarefas e prevenção de inversões de prioridade descontroladas proposto por (SHA; RAJKUMAR; LEHOCZKY, 1990).

O Protocolo possui as seguintes características:

- Tempo de espera máximo pelo recurso conhecido: O tempo máximo que uma tarefa de alta prioridade deve esperar por um recurso é equivalente a maior das seções criticas de todas as tarefas que utilizam o recurso com prioridade inferior a essa. Isto significa que o protocolo evita a formação de cadeias de bloqueios.
- Prevenção de *deadlocks* em sistemas monoprocessados: é provado que o protocolo previne a ocorrência de *deadlocks* (SHA; RAJKUMAR; LEHOCZKY, 1990).

A cada recurso é associada uma prioridade teto (ou *ceiling*) que é a maior prioridade entre as prioridades das tarefas que acessas o recurso.

A ideia por traz do protocolo é que, uma tarefa T só poderá executar uma seção crítica se sua prioridade for superior a todas as prioridades teto de todos os recursos bloqueados por tarefas que não sejam T. Caso a tarefa T não puder executar a seção crítica, ela cederá sua prioridade para a tarefa que a bloqueia.

Pelo fato de se ter que rastrear todos os recursos bloqueados e todas as solicitações de bloqueios globalmente, este protocolo se torna-se difícil de ser implementado.

3.2 Immediate Priority Ceiling

O Protocolo de sincronização para prioridade fixa *Immediate Priority Ceiling* (IPC) (BURNS; WELLINGS, 2001), é uma variação do *Protocolo Priority Ceiling*. Este protocolo também é conhecido como *Stack Resource Policy* para prioridade fixa ((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; PALENCIA, 2003), o tempo máximo de bloqueio para uma tarefa sob prioridade fixa utilizando recurso compartilhado pelo protocolo IPC é a maior seção crítica do sistema cuja prioridade teto seja mais alta ou igual a prioridade da tarefa em questão e seja utilizada por uma tarefa de prioridade mais baixa que ela. Também pode ser entendido pela pela fórmula 3.1, onde CS_{kj} é a k-ésima seção crítica da tarefa j e $Ceil(CS_{kj})$ é a prioridade teto do recurso associado a CS_{kj}. Apesar de, teoricamente, o B_i ser enunciado como bloqueio, na prática ele aparece associado à latência de ativação da tarefa, pois a tarefa só será ativada quando o recurso de interesse estiver disponível.

$$B_i = max(CS_{kj}) \forall_k, \forall_{(i \neq j)} | (P_i \leq P_j) \bigwedge (Ceil(CS_{kj}) \geq P_i) \tag{3.1}$$

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 3.1 mostra um exemplo semelhante ao mostrado na figura 2.3, 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=3.5*, 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*.

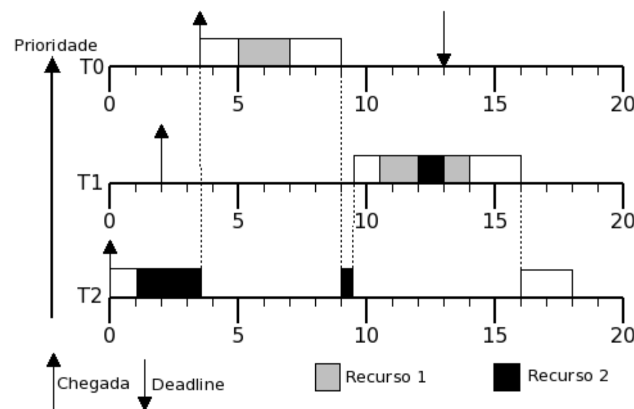

Figura 3.1: Possível inversão de prioridade resolvida por IPC

3.3 Conclusões

O protocolo *Priority Ceiling* contorna o problema da inversão de prioridades descontrolada, limitando o tempo de espera por recursos. Esta limitação ocorre pelo fato do protocolo não permitir a formação de cadeias de bloqueios. Entretanto, a implementação deste protocolo é muito difícil, e por vezes ineficiente. O protocolo *Immediate Priority Ceiling* foi proposto como uma simplificação do *Priotity Ceiling*, onde uma tarefa herda imediatamente a prioridade teto de um recurso adquirido, tornando o protocolo mais passível de implementação.

4 *Proposta de Implementação do Protocolo Immediate Priority Ceiling*

Este capítulo discorre sobre a implementação proposta, incluindo a API, estruturas de dados e comportamento geral da solução.

4.1 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 implementação proposta está baseada na árvore de desenvolvimento kernel-tip, branch rt/head (MOLNAR, 2010). Apesar da implementação dos rt-mutexes considerar sistemas multiprocessados, esta implementação do IPC considera somente monoprocessados.

A implementação foi feita primariamente para uso em *device-drivers* (ou seja, em espaço de *kernel*), como mostra a figura 4.1, 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 immpc_mutex*, e é

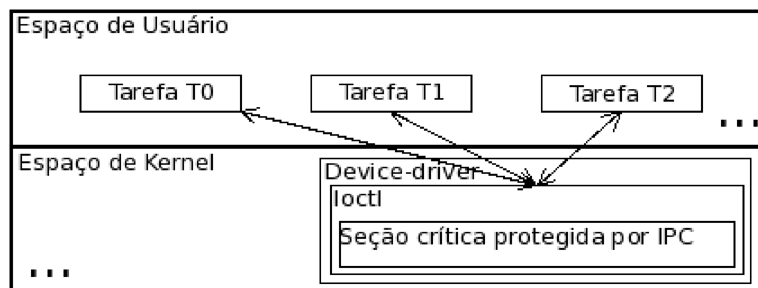

Figura 4.1: Diagrama de interação entre o protocolo IPC e tarefas

apresentado no código 1. Nesta estrutura, *wait_lock* é o *spinlock* que protege o acesso à estrutura, *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 immpc_mutex {
        raw_spinlock_t   wait_lock;
        struct plist_node      on_task_entry;
        struct task_struct *owner;
        int             ceiling;
   ...
};
```

Outra estrutura auxiliar foi definida, *immpc_synchronization_ctx*, apresentada na listagem de código 2. Esta estrutura foi incluído na estrutura task_struct para o controle dos mutex IPC, onde cada mutex adquirido por uma tarefa é adicionada ao *mutex_list* (linha 5) (pelo campo *on_task_entry* na estrutura *immpc_mutex*).

Código 2 Estrutura que representa um contexto de sincronização IPC

```
1 struct immpc_synchronization_ctx{
2 #if defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
3    int        need_prio_change;
4 #endif
5    struct     list_head  mutex_list;
6 };
```

4.1.1 API Proposta

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

- **DEFINE_IMMPC_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 1 a 99 (segundo a especificação de prioridades estáticas para tempo real (IEEE, 1998).

- **void immpc_mutex_init(struct immpc_mutex *lock, const char *name, int prio):** Inicializa um mutex alocado dinamicamente (por exemplo, via slab cache ou kmalloc). Pode ser utilizado para reincializar um mutex alocado estaticamente.

- **void immpc_mutex_lock(struct immpc_mutex *lock):** Função de aquisição do mutex. 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. 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 (*immpc_mutexes*) adquiridos até o momento.

- **void immpc_mutex_unlock(struct immpc_mutex *lock):** Efetua a liberação do recurso e realiza o reajuste da prioridade da tarefa que a chama.

- **void immpc_mutex_set_ceiling(struct immpc_mutex *lock, int newceiling):** Esta função altera o ceiling do mutex especificado. Esta função é o único mecanismo para ajustar o ceiling de um mutex inicializado. A política de ajuste (via *procfs*, no carga do módulo ou *ioctl*, por exemplo) deve ser definido pelo desenvolvedor do *driver*. Como na macro de definição estática, os valore para o *ceiling* devem estar no intervalo de 1 a 99.

4.1.2 Otimização por Postergação de Ajuste de Prioridade

Muitas vezes, a execução dos algoritmos das operações de bloqueio/desbloqueio se torna dispendiosa, especialmente quando essas operações ocorrem muito frequentemente dentro de algum segmento do kernel (em um *loop* rápido, por exemplo). Idealmente, haveria alguma forma de otimizar este processo, e é o que é feito pelo chamado *fastpath*.

Fastpath é a estratégia de adquirir ou liberar um mutex sem a necessidade de bloquear o *spinlock* que protege a estrutura deste. Ele é usado principalmente por motivos de desempenho, e é implementado com o uso de instruções atômicas (instruções conhecido como comparar e trocar - cmpxchg). Para arquiteturas que não possuem essa instrução, o *fastpath* não pode ser usado.

Os *fastpaths* implementados são muito semelhantes aos existentes na implementação original do rt-mutexes (uma pequena quantidade de código foi alterado), e funcionam de forma muito

semelhante. No protocolo PI (rt-mutexes), que é implementado para funcionar em multiprocessadores no Linux/PREEMPT-RT, quando o mutex não puder ser adquirido/liberado pelo *fastpath* porque alguma condição necessária não foi atendida (o mutex não está disponível, ou seja, foi adquirido por uma tarefa que não está em execução ou está sendo executado em outra CPU, por exemplo), é então executado o *slowpath*, que bloqueia a estrutura e retoma o processo de aquisição/liberação deste mutex.

Na implementação do protocolo IPC (immpc_mutex, que é implementado para trabalhar em monoprocessadores no Linux/PREEMPT-RT), o *fastpath*, quando ativado, nunca irá falhar (o mutex está sempre disponível quando solicitado segundo o protocolo IPC). Neste caso, o *slowpath* só será usado se a arquitetura não oferecer suporte *cmpxchg* ou o sistema não for compilado com suporte a *fastpath*.

De fato, um dos problemas do protocolo IPC é o fato do ajuste de prioridades das tarefas que adquirem algum mutex não poder ser efetuado atomicamente. O que foi feito para viabilizar o *fastpath* sob esta restrição foi a implementação de um mecanismo de postergação de ajuste de prioridades (com uma alteração no escalonador de tarefas). Uma tarefa que adquire um mutex têm uma *flag* ativada e passa para um estado definido como "Tarefa com ajuste de prioridade pendente". Esta postergação significa que o ajuste deverá ser efetuado em um momento mais propício, desde que não agora. O momento mais adequado para se efetuar é ajuste é a iminência de uma preempção ou reescalonamento do sistema.

Umas das vantagens da postergação é, como já foi citado, a viabilização de *fastpath*. Outra vantagem decorrente é o fato do ajuste não precisar ser realmente efetuado em caso de seções cíticas pequenas, ou seja, quanto menor a seção crítica, menor a probabilidade de ocorrer um reescalonamento em seu interior (motivado por uma interrupção de *timer* por exemplo).

Mesmo sendo atômica a aquisição do mutex pelo *fastpath*, ainda é necessária a realização posterior de uma operação não atômica, alem da habilitação da *flag* de "Tarefa com ajuste de prioridade pendente": a adição do mutex na lista de mutexes da tarefa, mas estas são operações *lockfree*, pois são sempre efetuadas pela tarefa que acabou de adquirir o mutex.

A listagem de código 3 apresenta um algoritmo simplificado para o *fastpath* de aquisição/-liberação. A parte não atômica é feita por chamadas para as funções track_immpc_mutex

e untrack_immp_mutex. O trabalho não atômico das funções track_immpc_mutex e untrack_immp_mutex é adicionar e remover, respectivamente, o mutex na lista de mutexes da tarefa chamadora, além de ativar a *flag* de ajuste de prioridade pendente. Quando a arquitetura não suportar instruções cmpxchg, a macro immpc_mutex_cmpxchg (linha 2) simplesmente retorna 0, forçando a aquisição/liberação pelo *slowpath*.

Código 3 Algoritmo simplificado para o *fastpath* de aquisição/liberação

```
1  #if defined(__HAVE_ARCH_CMPXCHG)
2  #define immpc_mutex_cmpxchg(l,c,n)\
3          (cmpxchg(&l->owner, c, n) == c)
4  #else
5  #define immpc_mutex_cmpxchg(l,c,n)   (0)
6
7  void immpc_mutex_fastlock(struct immpc_mutex* lock){
8
9      /*falhara se o sistema nao suportar cmpxchg
10     ou n o for compilado para utiliza  o do pastpath*/
11     if(immpc_mutex_cmpxchg(lock->owner, NULL, current)){
12         track_immpc_mutex(lock, current);
13     } else {
14
15         /*processo tradicional*/
16         immpc_mutex_slowlock(lock);
17     }
18 }
19
20 void immpc_mutex_fastlock(struct immpc_mutex* lock){
21
22     /*falhara se o sistema nao suportar cmpxchg
23     ou n o for compilado para utiliza  o do pastpath*/
24     if(immpc_mutex_cmpxchg(lock->owner, current, NULL)){
25         untrack_immpc_mutex(lock, current);
26     } else {
27
28         /*processo tradicional*/
29         immpc_mutex_slowlock(lock);
30     }
31 }
```

4.1.3 Mantenabilidade da Implementação

O *Patch* implementado para o IPC é mantido com o suporte da ferramenta de controle de versões distribuída Git (TORVALDS; HAMANO, 2005). A versão atual apresenta as seguintes estatísticas de alterações (em relação ao *branch* rt/head da árvore de desenvolvimento kernel-tip (MOLNAR, 2010)):

```
include/linux/immpc_mutex.h |  162 +++++++
include/linux/sched.h       |   11 +-
kernel/Makefile             |    2 +
kernel/fork.c               |    5 +
kernel/immpc_mutex.c        | 1020 +++++++++++++++++++++++++++++++++++++
kernel/sched.c              |    5 +
6 files changed, 1202 insertions(+), 3 deletions(-)
create mode 100644 include/linux/immpc_mutex.h
create mode 100644 kernel/immpc_mutex.c
```

O que indica que foram criados dois arquivos:

- include/linux/immpc_mutex.h
- kernel/immpc_mutex.c

E alterados outros quatro:

- include/linux/sched.h
- kernel/Makefile
- kernel/fork.c
- kernel/sched.c

Sendo que o arquivo mais alterado foi o include/linux/sched.h, e mesmo assim, de forma mínima. Desta forma, a implementação fica confinada nos arquivos criados e não nos arquivos previamente existentes do *core* do kernel, alterando somente o estritamente necessário para o funcionamento do mecanismo.

4.2 Conclusões

Este capítulo mostrou a implementação proposta. Também foi mostrado uma otimização dependente de arquitetura, na qual o ajuste de prioridades é efetuado somente no momento de interesse, evitando que este ocorra muito frequentemente para o caso de seções críticas muito pequenas. Por último, foi mostrado que a implementação não necessita de grande esforço para sua manutenção perante novas versões do kernel, dado que esta altera muito pouco do código já existente no *core* do sistema.

5 *Avaliação do Protocolo Immediate Priority Ceiling Implementado*

Neste capítulo, serão apresentados testes comparativos envolvendo o protocolo *Immediate Priority Celing* e *Priority Inheritance* presente no kernel de tempo real do Linux.

5.1 Definições Preliminares

Para para a realização e interpretação dos testes, será utilizada as seguintes definições de tempo real:

Latência de Ativação: Também conhecida como *Release Jitter*. É o tempo entre uma tarefa ser acordada (inserida na fila *ready*) e efetivamente começar a executar.

Tempo de Bloqueio: É o tempo entre a solicitação e a efetiva posse de um recurso mutualmente exclusivo.

Tempo de Resposta: É o intervalo de tempo desde a liberação (ativação) de uma tarefa até a sua conclusão (dentro de um período).

5.2 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* 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 [1] 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 5.1. 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 5.2.

A tabela 5.1 apresenta os intervalos entre ativações expressos com uma pseudo-aleatoriedade, ou seja, com valores distribuídos 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 5.1 são apresentados os tamanhos das seções críticas de cada uma das tarefas. Outra informação mostrada na tabela 5.1 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 escalonamento 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.

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).

[1] Tarefas esporádicas são tarefas com período desconhecido. No entanto possuem um intervalo mínimo entre cada ativação.

Tabela 5.1: 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 5.2: 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)			

5.2.1 Resultados da Utilização do mutex com PI

Com herança de prioridade, a tarefa de alta prioridade apresentou latências de ativação aproximadamente (como pode ser observado no histograma da figura 5.1) 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 5.2, 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.

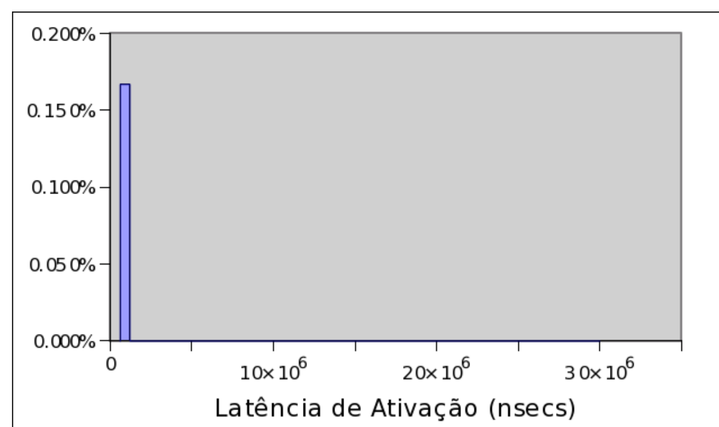

Figura 5.1: Histograma das latências de ativação (alta prioridade utilizando PI)

Em relação ao tempo de resposta (como pode ser observado no histograma da figura 5.3), 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 5.3 o pior caso no tempo de resposta máximo observado como sendo 64.157.591

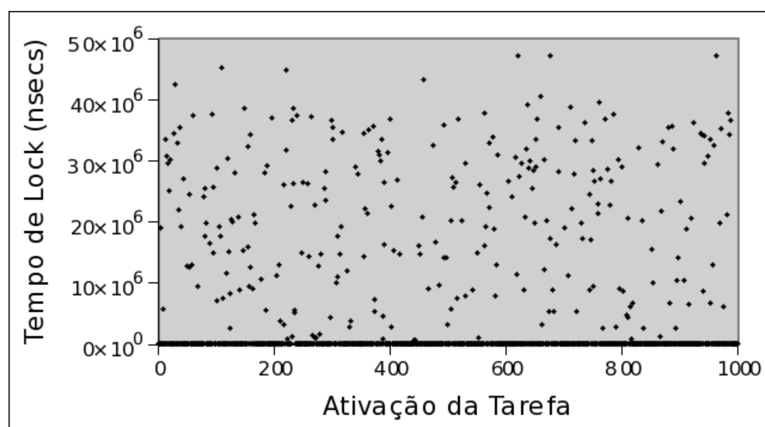

Figura 5.2: Gráfico dos tempos de bloqueio (tarefa de alta prioridade utilizando PI)

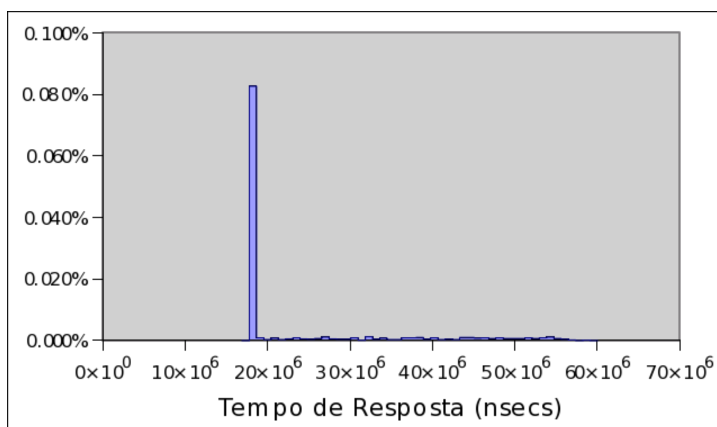

Figura 5.3: Histograma do tempo de resposta (alta prioridade utilizando PI)

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.

Tabela 5.3: Tempos médios de resposta e desvio padrão

Protocolo:	PI	IPC
Tempo médio de resposta :	22.798.549 ns	21.014.311 ns
Desvio padrão:	11.319.355 ns	8.723.159 ns
Máx:	64.157.591 ns	50.811.328 ns

5.2.2 Resultados da Utilização do mutex com IPC

Utilizando IPC, pode-se notar no histograma da figura 5.4 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 5.5, o que é o esperado segundo a definição do protocolo implementado. Já no histograma do tempo de resposta da figura 5.6, aparece uma cauda (valores mais altos, mas com baixas ocorrências) devido à latência de ativação sofrida.

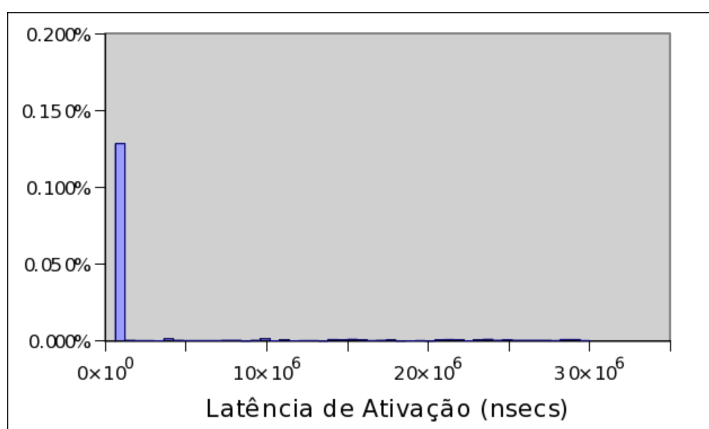

Figura 5.4: Histograma das latências de ativação (alta prioridade utilizando IPC)

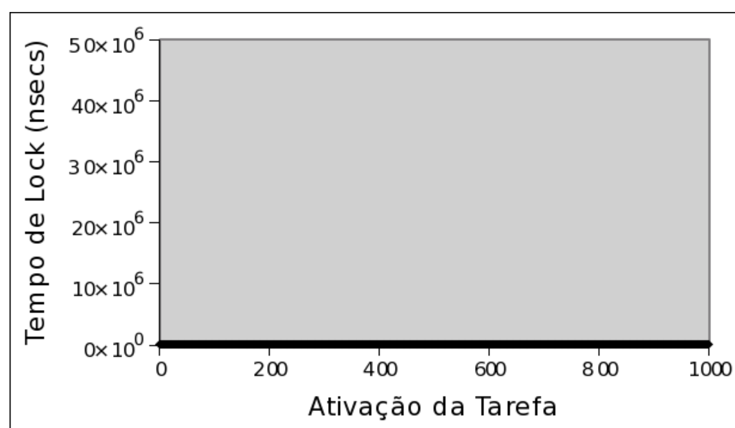

Figura 5.5: Gráfico dos tempos de bloqueio (alta prioridade Utilizando IPC)

Também neste teste, pode-se observar na tabela 5.3 o pior caso no tempo de resposta máximo observado como sendo 50.811.328 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.

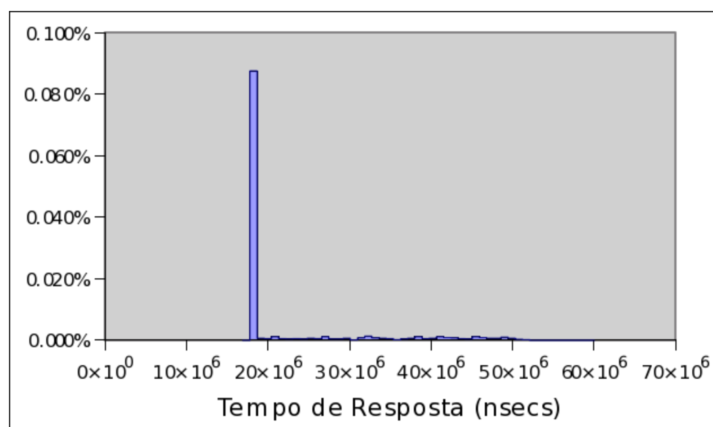

Figura 5.6: Histograma do tempo de resposta (alta prioridade utilizando IPC)

5.2.3 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 5.3, o protocolo IPC apresentou o desvio padrão e o tempo médio de resposta menor que o PI. Outro ponto importante na tabela 5.3 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 5.7 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 distribuíram 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 5.3, 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 5.4 resume qualitativamente os resultados encontrados.

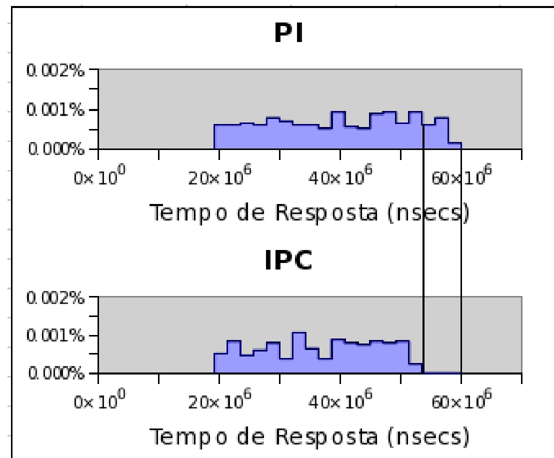

Figura 5.7: Histograma do tempo de resposta da tarefa de alta prioridade

Tabela 5.4: Resumo dos resultados encontrados

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

5.3 Overhead da Implementação Proposta

5.3.1 Definição do Overhead

Definimos como *overhead* qualquer queda na capacidade do sistema em produzir trabalho útil. Desta forma, para este estudo, o *overhead* será considerado como a diminuição do tempo de CPU fornecido ao resto do sistema, dada a presença de um conjunto de tarefas de mais alta prioridade compartilhando recursos protegidos por PI ou IPC.

5.3.2 Descrição do Experimento Utilizado

Para avaliação do protocolo implementado em termos de *overhead* imposto ao sistema, foi utilizado um conjunto de tarefas de teste como especificado na tabela 5.5. Nesta mesma tabela,

são apresentadas as configurações das tarefas.

Para realizar a estimativa do *overhead*, foi criada uma tarefa com prioridade 51 (em política SCHED_FIFO). Esta prioridade fica acima da prioridade padrão dos *Threaded irq handlers* (ROSTEDT; HART, 2007) e das *Softirqs* (LOVE, 2005). Isto foi feito para manter a tarefa de medição fora da interferência dos mecanismos de tratamento de interrupções e postergação de trabalho do Linux. Todo o tempo que sobra de processamento (não utilizado pelas tarefas de teste sincronizadas por IPC ou PI) é então atribuído a tarefa de medição. Tanto a tarefa de medição quanto o conjunto de tarefas sincronizadas por IPC ou PI foram fixadas a uma mesma CPU (CPU 0 de um sistema com 2 núcleos).

A tarefa de medição é iniciada no mesmo instante que as tarefas de tempo real e termina antes que estas terminem. Em cada iteração de teste, a tarefa de medição executa por 17 segundos. Segundo consta na tabela 5.5, a tarefa T0' executa 40 ativações, as demais executam até o término desta.

Tarefa	**T0'**	**T1'**	**T2'**	**T3'**	**T4'**	**T5'**	**T6'**
Prioridade	70	65	64	63	62	61	60
Inervalo de ativação(ms)	rand in [500,1000]	rand in [100,200]	rand in [100,200]	rand in [100,200]	rand in [90,180]	rand in [90,180]	rand in [90,180]
Recurso	R1	R1, R2	R1, R2	R1, R2	R2	R2	R2
Tamanho da sec. crit.	aprox. 17 ms	aprox. 2x17 ms	aprox. 2x17 ms	aprox. 2x17 ms	aprox. 17 ms	aprox. 17 ms	aprox. 17 ms
Número de ativações	40	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep

Tabela 5.5: Configuração do conjunto de tarefas do teste de *overhead*

Para obtenção da estimativa de *overhead*, a tarefa de medição é executada em loop infinito pelo tempo especificado acima (17 segundos). O *overhead* será notado pelo quanto a tarefa de medição conseguiu utilizar a CPU no intervalo pré-definido, levando em conta a execução do conjunto de tarefas sincronizado por IPC ou PI (sempre pareando uma execução de IPC com uma de PI). Os testes foram repetidos para ambos os casos, com e sem *fastpath*.

5.3.3 Resultados Obtidos

Os valores dos tempos de CPU obtidos pela tarefa de medição são apresentados na tabela 5.6, que foi ordenada para facilitar a comparação visual. A tabela 5.6 também apresenta os dados estatísticos básicos das amostras obtidas.

5.3.4 Avaliação Estatística dos Resultados

Descrição do Procedimento de Análise Estatística

Para avaliação dos resultados foi utilizado o teste estatístico de hipóteses para médias com variância desconhecida (teste t-Student). Por hipótese, o *overhead* do PI e do IPC são iguais (para ambos os casos, com e sem *fastpath*), ou seja, as médias de IPC e PI são iguais, então:

1. $H_0 : \mu_{PI} = \mu_{IPC_{NOFP}}$ e, ou
2. $H_{0'} : \mu_{PI} = \mu_{IPC_{FP}}$

Os dados apresentados na Tabela 5.7, foram obtido a partir da tabela 5.6 mais a informação do número de amostras (n=40). Esta tabela fornece os dados necessários para o teste de hipóteses.

Interpretação do Resultado

Para a hipótese H_0, como os dados produziram o valor *t = -12,51*, o qual não pertence à região de aceitação (segundo a distribuição t-Studend), o teste rejeita H_0 com um nível de significância de 0,1%.

Ao nível de significância de 0,1% os dados comprovam uma diferença entre PI e IPC_{NOFP}. Existe a probabilidade inferior a 0,1% de que as diferenças observadas nos dados apresentados sejam provenientes de fatores casuais do sistema.

No entanto, para o caso do IPC_{FP} (com *fastpath*), o valor de t = 1,64 indica uma equivalência entre o *overhead* causada pelo IPC_{FP} e PI. Ou seja, é notável que para arquiteturas que suportam instruções atômicas cmpxchg, o *fastpath* pode realmente fazer a diferença em termos de *overhead*.

No IPC_{NOFP} (sem *fastpath*), há sempre uma necessidade de verificação de prioridade e ajuste. Outro ponto é que se uma tarefa com prioridade abaixo da prioridade teto de um determinado recurso adquire esse recurso, a sua prioridade necessariamente terá que ser ajustada, e isso certamente influenciará o *overhead*. Como os testes mostram, há uma probabilidade razoável das tarefas encontrarem os recursos que precisam disponíveis, então, nem sempre o algoritmo de propagação de prioridades (no caso do PI) será executado, mas sempre haverá ajustes de prioridades (para o caso do IPC_{NO}), exceto para a tarefa que define a prioridade teto do recurso (tarefa de mais alta prioridade entre as tarefas que utilizam o recurso).

No caso do IPC_{FP} (com *fastpath*), só irá ocorrer o ajuste de prioridade, se houver uma iminência de preempção por outra tarefa. Assim, há possibilidades do ajuste de prioridade efetivamente não ocorrer. Outro ponto importante que ajuda a explicar a resultados é que o IPC com fasthpath executa algumas instruções mais do que o *fastpath* do PI, no entanto, o protocolo PI induz a mais chaveamentos de contexto, levando a uma *overhead* muito próximo.

5.4 Conclusões

Como visto nos testes, em termos de tempo de resposta o IPC apresentou melhores resultados. Outro ponto a favor do protocolo IPC aparece quando se observa a diferença no pior caso do tempo de resposta observado nos testes, que no IPC foi cerca de uma seção crítica menor que no PI. Mesmo induzindo a tempos de bloqueio e consequentemente resposta menores, testes realizados mostraram que o *overhead* do IPC_{NOFP} (sem *fastpath*/não otimizado) implementado é maior que o do PI nativo do PREEMPT-RT. Já para o IPC_{FP} (com *fastpath*/otimizado), os testes realizados apresentaram um *overhead* equivalente.

Estat.	IPC_{FP}	PI		IPC_{NOFP}	PI
	407.035.114	406.435.450		401.424.676	405.553.163
	407.285.636	407.765.565		401.820.286	406.458.117
	407.373.521	407.835.019		401.948.422	406.658.942
	407.819.905	407.876.696		402.597.568	406.856.751
	408.377.891	408.048.768		403.542.887	407.004.448
	408.381.506	408.101.843		403.547.748	407.238.676
	408.406.446	408.219.724		403.874.521	407.350.956
	408.417.061	408.220.571		404.566.739	407.427.672
	408.711.409	408.433.649		404.686.516	407.513.400
	408.763.366	408.446.200		404.707.405	407.577.573
	408.801.061	408.549.755		404.757.367	407.596.261
	408.883.156	408.575.046		404.808.759	407.706.637
	408.949.145	408.575.106		404.826.216	407.785.273
	409.148.907	408.606.476		404.827.682	408.013.960
	409.202.266	408.783.361		404.970.791	408.315.982
	409.443.672	408.785.943		404.978.816	408.495.828
	409.518.531	408.790.334		405.086.464	408.602.577
	409.658.142	408.862.429		405.122.964	408.604.205
	409.662.722	408.987.951		405.165.352	408.702.488
	409.679.490	409.156.013		405.169.874	408.789.189
	409.721.677	409.171.710		405.245.988	408.846.701
	409.761.649	409.212.383		405.409.364	408.945.776
	409.772.169	409.230.027		405.441.947	408.956.142
	409.837.278	409.381.642		405.505.815	408.961.259
	409.899.266	409.399.327		405.560.936	408.984.062
	410.158.821	409.409.126		405.626.303	409.024.543
	410.207.755	409.424.264		405.750.834	409.159.619
	410.419.774	409.502.679		405.770.308	409.395.789
	410.477.452	409.626.412		405.784.920	409.459.483
	410.492.265	409.710.682		405.917.678	409.465.209
	410.496.194	409.833.770		405.974.737	409.513.038
	410.546.730	409.866.286		405.994.401	409.593.505
	410.661.307	409.872.273		406.031.717	409.602.110
	410.679.530	409.873.581		406.043.948	409.977.571
	410.715.126	410.066.899		406.253.972	410.018.274
	410.853.474	410.129.241		406.322.758	410.079.834
	410.857.619	410.179.304		406.406.613	410.284.319
	411.127.684	410.226.307		406.618.569	410.312.717
	411.289.886	410.617.801		406.661.586	410.467.174
	431.575.650	412.029.972		406.945.074	410.907.269
Média:	410.076.756	409.095.489		405.042.463	408.605.162
Variância:	13.330.393.417.558	943.973.889.922		1.706.936.654.588	1.535.642.083.490
Mínimo:	408.711.409	408.433.649		404.686.516	407.513.400
Máximo:	431.575.650	412.029.972		406.945.074	410.907.269

Tabela 5.6: Tempo de CPU da tarefa de medição e estatísticas relacionadas

	Sem *fastpath* (IPC_{NOFP})	Com *fastpath* (IPC_{FP})
$Sa^2_{IPC,PI}$	1.621.289.369.039	7.137.183.653.740
n	40	40
α	0,1%	0,1%
$d.f.$	80	80
t	-12,51	1,64

Tabela 5.7: Dados do teste t-Student

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 (via exclusão mútua), 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. A proposta de implementação considera somente sistemas monoprocessados pelo fato de não existir na literatura, protocolos para sistemas multiprocessados aplicáveis ao Linux no qual o tempo de bloqueio possa ser facilmente delimitado. Esta inexistência ocorre pelo fato do kernel do Linux possuir um modelo de processos e sincronização muito mais flexível que os modelos previstos nos protocolos para sistemas multiprocessados.

O Protocolo IPC pode ser mais adequado para aplicações dedicadas, que utitizem 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). 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 e na cache. No protocolo IPC, quando uma tarefa é ativada, ela não sofre mais bloqueios em sua execução, independentemente do número se seções críticas, o que não ocorre com o PI, onde a tarefa pode ser bloqueada a cada tentativa de entrada em uma seção crítica.

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 e sistemas embarcados 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, este foi cerca de uma seção crítica menor que no PI, como pode ser visto também na tabela 5.3. 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 não ocorre com o IPC. Neste, este tempo será sempre de no máximo uma seção crítica.

Embora os tempos de bloqueio e de resposta serem menores menores no IPC, testes mostraram que o *overhead* da versão não-otimizada do IPC implementado é maior do que o PI (rt-mutex) nativo no Linux/PREEMPT-RT. Esse *overhead* é causado provavelmente pela ausência de um *fastpath* na implementação desta versão, pois há um conjunto de operações de bloqueio/-desbloqueio, que não pode ser executado atomicamente como na PI. Estes operações incluem mudanças de prioridade e rastreamento de mutexes adquiridos por tarefas, por exemplo. No entanto, a versão otimizada mostrou um *overhead* equivalente. Esta equivalência surge devido ao fato do IPC executar mais instruções nas primitivas de bloqueio/desbloqueio (no entanto, consideravelmente menos do que na versão não otimizada do IPC), e que são compensadas pelo fato do protocolo PI realizar mais chaveamentos de contexto e ajustes de prioridade em cadeia (para propagação de prioridades no caso de bloqueios).

Referências Bibliográficas

BAKER, T. A stack-based resource allocation policy for realtime processes. In: *IEEE Real-Time Systems Symposium*. [S.l.: s.n.], 1990. v. 270.

BURNS, A.; WELLINGS, A. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. [S.l.]: Addison Wesley, 2001.

CARMINATI, A.; OLIVEIRA, R. de. Interferência das Hard irqs e Softirqs em Tarefas com Prioridade de Tempo Real no Linux. Anais do Workshop de Sistemas Operacionais, 2009.

DOZIO, L.; MANTEGAZZA, P. Linux real time application interface (rtai) in low cost high performance motion control. *Proceedings of the conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*, 2003.

GERUM, P. http://www.xenomai.org - last access 01/21, 2009.

GLEIXNER, T.; MOLNAR, I. *High resolution timers and dynamic ticks design notes*. [S.l.], 2006. Http://www.kernel.org/doc/Documentation/Documentation/timers/highres.txt.

HARBOUR, M.; PALENCIA, J. Response time analysis for tasks scheduled under EDF within fixed priorities. In: CITESEER. *Proceedings of the 24th IEEE International Real-Time Systems Symposium*. [S.l.], 2003. p. 200.

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

KRTEN, R. Getting Started with QNX Neutrino: A Guide for Realtime Programmers. 2009.

LOVE, R. *Linux Kernel Development (Novell Press)*. [S.l.]: Novell Press, 2005.

MCKENNEY, P. et al. Read-copy update. In: CITESEER. *Ottawa Linux Symposium*. [S.l.], 2001.

MOLNAR, I. *PREEMPT-RT*. Http://www.kernel.org/pub/linux/kernel/projects/rt - Last access 01/21, 2009.

MOLNAR, I. *PREEMPT-RT Development Branch*. 2010. Git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git.

RIVER, W. *VxWorks: Reference Manual*. [S.l.]: Wind River Systems.(Wind River, 2005), 2005.

ROSTEDT, S. *RT Mutex Design - Linux Kernel Documentation*. [S.l.], 2006. Http://www.kernel.org/doc/Documentation/rt-mutex-design.txt.

ROSTEDT, S.; HART, D. Internals of the RT Patch. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2007. v. 2007.

SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, v. 39, n. 9, p. 1175–1185, 1990.

TORVALDS, L.; HAMANO, J. *GIT-fast version control system*. 2005.

YODAIKEN, V. Against priority inheritance. *FSMLABS Technical Paper*, Citeseer, 2003. Available at `http://yodaiken.com/papers/inherit.pdf`.

APÊNDICE A – Artigo

Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux

Andreu Carminati[1]

[1] Departamento de Informática e Estatística – Universidade Federal de Santa Catarina

andreu@das.ufsc.br

Abstract. *In general purpose operating 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 compares tests on the protocol implemented and Priority Inheritance, currently used in the real-time kernel.*

1. Introduction

In real-time operating systems such as Linux/PREEMPT-RT [Molnar , Rostedt and Hart 2007], task synchronization mechanisms must ensure both the maintenance of internal consistency in relation to resources or data structures, and determinism in waiting time for these. They should avoid unbounded priority inversions, where a high priority task is blocked indefinitely waiting for a resource that is in possession of a task with lower priority.

In general purpose systems, such as 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 (PI) [Sha et al. 1990].

The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling (IPC) [Lampson and Redell 1980, Sha et al. 1990], for use in drivers dedicated to real-time applications, for example. In this scenario, an embedded Linux supports an specific known application that does not change task priorities after its initialization. It is not the objective of this paper to propose a complete replacement of the existing protocol, as mentioned above, but an alternative for use in some situations. The work in this article only considered uniprocessor systems.

This paper is organized as follows: section 2 presents the current synchronization scenario of the mainline kernel and PREEMPT-RT, section 3 explains about the Immediate Priority Ceiling protocol, section 4 explains how the protocol was implemented in the Linux real-time kernel, section 5 compares tests made upon the protocol implemented

and Priority Inheritance implemented in the real-time kernel and section 6 presents an overhead analysis between IPC and PI.

2. Mutual Exclusion in the Linux Kernel

Since there is no mechanism in the mainline kernel that prevents the appearance of priority inversions, situations like the one shown in Figure 1 can occur very easily, where task T2, activated at $t = 1$, acquires the shared resource. Then, task T0 is activated at $t = 2$ but blocks because the resource is held by T2. T2 resumes execution, and is preempted by T1, which is activated and begins to run from $t = 5$ to $t = 11$. But task T0 misses the deadline at $t = 9$, and the resource required for its completion was only available at $t = 12$ (after the deadline).

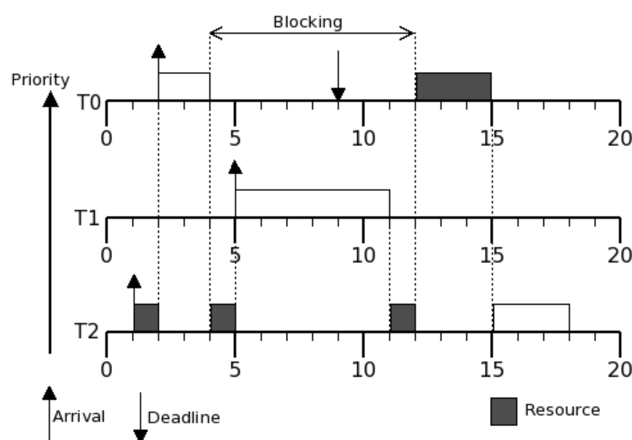

Figure 1. Priority inversion

In the real-time kernel PREEMPT-RT exists the implementation of PI (Priority Inheritance). As mentioned above, it is a mechanism used to accelerate the release of resources in real-time systems, and to avoid the effect of indefinite delay of high priority tasks that can be blocked waiting for resources held by tasks of lower priority.

In the PI protocol, a high priority task that is blocked on some resource, gives its priority to the low priority task (holding that resource), so will release the resource without suffering preemptions by tasks with intermediate priority. This protocol can generate chaining of priority adjustments (a sequence of cascading adjustments) depending on the nesting of critical sections.

Figure 2 presents an example of how the PI protocol can help in the problem of priority inversion. In this example, task T2 is activated at $t = 1$ and acquires a shared resource, at $t = 1$. Task T0 is activated and blocks on the resource held by T2 at $t = 4$. T2 inherits the priority from T0 and prevents T1 from running, when activated at $t = 5$. At $t = 6$, task T2 releases the resource, its priority changes back to its normal priority, and task T0 can conclude without missing its deadline.

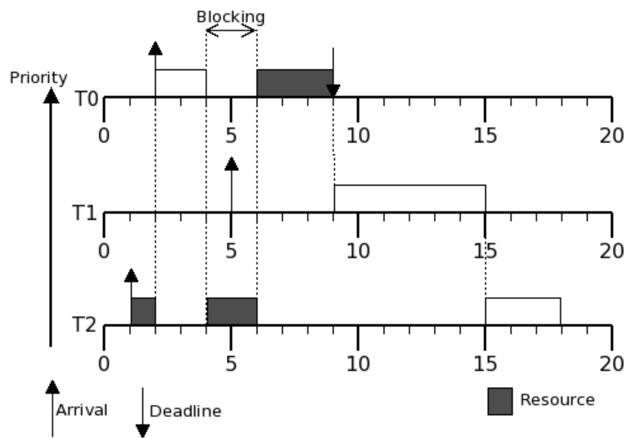

Figure 2. Priority inversion resolved by PI

Some of the problems [Yodaiken 2003] of this protocol are the number of context switches and blocking times larger than the largest of the critical sections [Sha et al. 1990] (for the high priority task), depending on the pattern of arrivals of the task set that shares certain resources.

Figure 3 is an example where protocol PI does not prevent the missing of the deadline of the highest priority task. In this example, there is the nesting of critical sections, where T1 (the intermediate priority) has the critical sections of resources 1 and 2 nested. In this example, task T0, when blocked on resource 1 at $t = 5$, gives his priority to task T1, which also blocks on resource 2 at $t = 6$. T1 in turn gives its priority to task T2, which resumes its execution and releases the resource 2, allowing T1 to use that resource and to release the resource 1 to T0 at $t = 10$. T0 resumes its execution but it misses its deadline, which occurs at $t = 13$. In this example, task T0 was blocked by a part of the time of the external critical section of T1 plus a part of the time of the critical section of T2. In a larger system the blocking time of T0 in the worst case would be the sum of many critical sections, mostly associated with resources not used by T0.

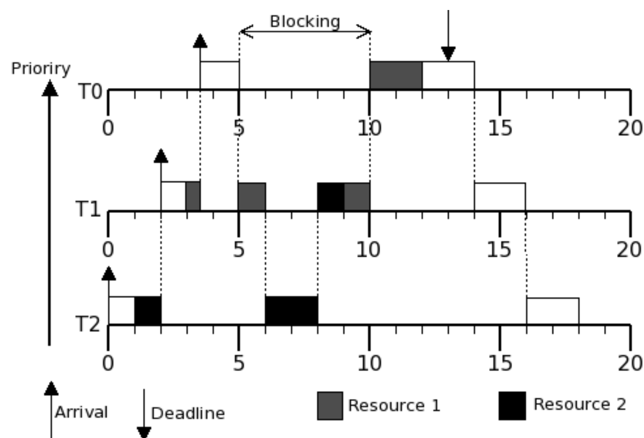

Figure 3. Priority inversion not resolved by PI

3. The Immediate Priority Ceiling Protocol

The Immediate Priority Ceiling (IPC) [Burns and Wellings 2001] synchronization protocol for fixed priority tasks, is a variation of Priority Ceiling Protocol [Sha et al. 1990, Baker 1990] or *Highest Locker Priority*. This protocol is an alternative mechanism for unbounded priority inversion control, and prevention of deadlocks in uniprocessor systems.

In the PI protocol, the priority is associated only to tasks. In IPC, the priority is associated with both tasks and resources. A resource protected by IPC has a priority ceiling, and this priority is the highest priority of all task priorities that access this resource.

According to [Harbour and Palencia 2003], the maximum block time of a task under fixed priority using shared resource protected by IPC protocol is the larger critical section of the system whose priority ceiling is higher than the priority of the task in question and is also used by a lower priority task.

What happens in IPC can be considered as preventive inheritance, where the priority is adjusted immediately when occurs a resource acquisition, and not when the resource becomes necessary to a high priority task, as in PI (you can think of PI as the IPC, but with dynamic adjustment of the ceiling). This preventive priority setting prevents low priority tasks from being preempted by tasks with intermediate priorities, which have priorities higher than low priority tasks and lower than the resource priority ceiling.

Figure 4 shows an example similar to that shown in Figure 3, but this time using IPC. In this example, the high priority task does not miss its deadline, because when task T2 acquires resource 2 at $t = 1$, its priority is raised to the ceiling of the resource (priority of T1), preventing task T1, activated at $t = 2$, from starting its execution. At $t = 3.5$, task T0 is activated and begins its execution. The task is no longer blocked because resource 1 is available. Task T0 does not miss its deadline.

4. Description of the Implementation

The Immediate Priority Ceiling Protocol was implemented based on the code of rt_mutexes already in the patch PREEMPT-RT. The rt_mutexes are mutexes that implement the Priority Inheritance protocol. The kernel version used for implementation was the 2.6.31.6 [Torvalds 2010] with PREEMPT-RT patch rt19. Although rt_mutexes are implemented in PREEMPT-RT for both uniprocessor and multiprocesors, our implementation of IPC considers only the uniprocessor case.

The implementation was made primarily for use in device-drivers (kernel space), as shown in Figure 5, where there is an example of tasks sharing a critical section protected by IPC and accessed through an ioctl system call to a device-driver.

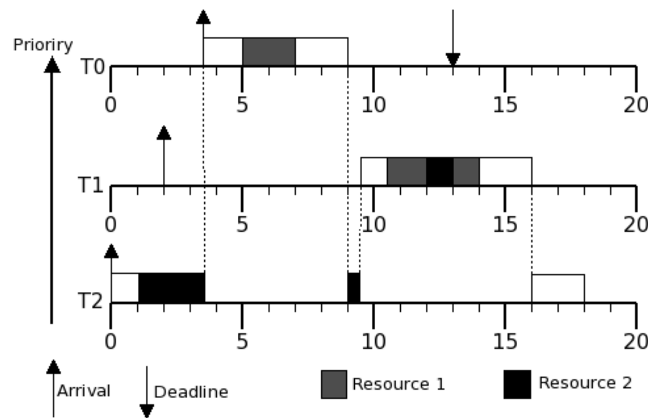

Figure 4. Priority inversion resolved by IPC

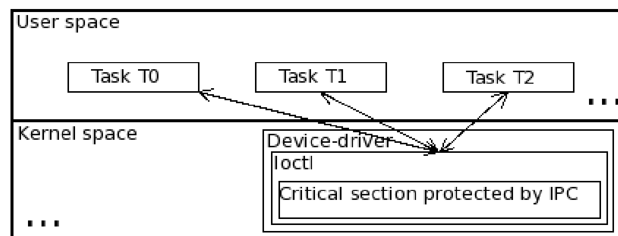

Figure 5. Diagram of interaction between the IPC protocol and tasks

estrutura,

The type that represents the IPC protocol was defined as *struct immpc_mutex*, and it is presented in code 1. In this structure, *wait_lock* is the spinlock that protects the access to the structure, *on_task_entry* serves to manage the locks acquired (and, consequently, control of priorities), *owner* stores a pointer to the task owner of the mutex (or null pointer if the mutex is available) and finally the *ceiling*, which stores the priority ceiling of the mutex.

Code 1 Data structure that represents a IPC mutex

```
struct immpc_mutex {
        atomic_spinlock_t wait_lock;
        struct plist_node on_task_entry;
        struct task_struct *owner;
        int ceiling;
...
};
```

The proposed implementation presents the following API of functions and macros:

- **DEFINE_IMMPC_MUTEX(mutexname, priority):** This macro is provided for the definition of a static IPC mutex, where *mutexname* is the identifier of the mutex and priority is the ceiling of the mutex, or a value in the range of 0 to 99 (according to the specification of static priorities for real-time tasks

on Linux). The current version can only create mutexes with priorities set at compile time, thus, the priority ceiling should be assigned by the designer of the application. This is not a too restrictive assumption when an embedded Linux runs a known application that does not change task priorities after its initialization.

- **void immpc_mutex_lock(struct immpc_mutex * lock):** Mutex acquisition function. In uniprocessor systems this is a nonblocking function because, according to the IPC protocol, if a task requests a resource, it is because this resource is available (the owner is null). In multiprocessor systems this function can generate blocks, because the resource can be in use on other processor (the *owner* field differs from zero). In this article, only the uniprocessor version will be taken into consideration. The main role of this function is to manage the priority of the calling task along with the resource blocking, taking into account all immpc_mutexes acquired so far.

- **void immpc_mutex_unlock(struct immpc _mutex * lock):** Effects the release of the resource and the adjustment of the priority of the calling task. In multiprocessor systems, this function also makes the job of selecting the next task (by the *wait_list*) that will use the resource. What also occurs in multiprocessor systems is the effect "next owner pending" (also present in the original implementation of priority inheritance) also known as steal lock, where the task of highest priority can acquire the mutex again, even though it has been assigned to another task that did not take possession of it.

One major difference between the proposed implementation and the already existent in the PREEMPT-RT is that the latter one enables the following optimizations:

- PI can perform atomic lock: if a task attempts to acquire a mutex that is available, this operation can be performed atomically (operation atomic compare and exchange) by what is known as fast path. But this is only possible for architectures that have this type of atomic operation. Otherwise if the lock is not available and/or the architecture does not have exchange and compare instruction, the lock will not be atomic (slow path).

- PI can perform atomic unlock: when a task releases a mutex which has no tasks waiting, this operation can be performed atomically.

As mentioned earlier, these optimizations are possible only for PI mutexes. In the case of IPC, there will always be a need for verification and a possible adjustment of priority.

5. Implementation Analysis

We developed a device-driver that has the function of providing the critical sections necessary to perform the tests. This device-driver exports a single service as a service

call ioctl (more specifically unlocked_ioctl, because the ioctl is protected by traditional Big Kernel Lock, which certainly would prevent the proper execution of the tests). It multiplexes the calls of the three tasks in their correspondent critical sections. This device-driver provides critical sections to run with both IPC and PI.

In order to carry out tests for the analysis of the implementation it was used a set of sporadic tasks executed in user space. The interval between activations, the resources used and the size of the critical section within the device-driver used by each task are presented in Table 1. All critical sections are executed within the function ioctl, within Linux kernel. A high-level summary of actions performed by each task (in relation to resources used) is presented in Table 2.

Table 1 shows the intervals between activations expressed with a pseudo-randomness, ie, with values uniformly distributed between minimum and maximum values. This randomness was included to improve the distribution of results because, with fixed periods, arrival patterns were being limited to a much more restricted set. Table 1 also presents the sizes of the critical sections of each task. Other information shown in Table 1 is the number of activations performed for each task. For the high priority task, there were 1000 monitored activations (latency, response time, critical section time, lock time, etc). For other tasks there was no restriction on the number of activations.

The high priority task has one of the highest priorities of the system. The other tasks were regarded as medium and low priorities. But they also have absolute high priorities. All tasks have been configured with the scheduling policy SCHED_FIFO, which is one of the policies for real-time [IEEE 1998] available in Linux.

Even with the use of a SMP machine for testing, all tasks were set at only one CPU (CPU 0). The tests were conducted using both IPC and PI for comparison purposes.

Task	**T0/High**	**T1/Med.**	**T2/Low**
Priority	70	65	60
Activation interval	rand in [400,800] ms	rand in [95,190] ms	rand in [85,170] ms
Resource	R1	R1,R2	R2
Critical section size	aprox. 17 ms	aprox. 2x17 ms	aprox. 17 ms

Table 1. Configuration of the set of tasks

Mutex R1 has been configured with priority ceiling 70 (which is the priority of task T0) and R2 has been configured with priority ceiling 65 (which is the priority of task T1).

Task	T0/High	T1/Med.	T2/Low
Action 1	Lock(R1)	Lock(R1)	Lock(R2)
Action 2	Critical Sec.	Critical Sec.	Critical Sec.
Action 3	Unlock(R1)	Lock(R2)	Unlock(R2)
Action 4		Critical Sec.	
Action 5		Unlock(R2)	
Action 6		Unlock(R1)	

Table 2. Actions realized by tasks

5.1. Results of the Use of the PI mutex

With priority inheritance, the high priority task had activation latencies as can be seen in the histogram of Figure 6 appearing in the interval [20000, 30000] nanoseconds (range where the vertical bar is situated in the histogram). Because of finding the resource busy with a certain frequency (as illustrated in Figure 7, waiting time for the resource), the task was obligated to perform volunteer context switch for propagation of its priority along the chain of locks.

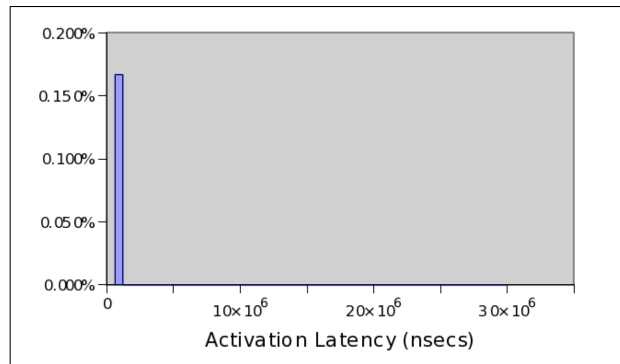

Figure 6. Histogram of activation latencies (high priority task using PI)

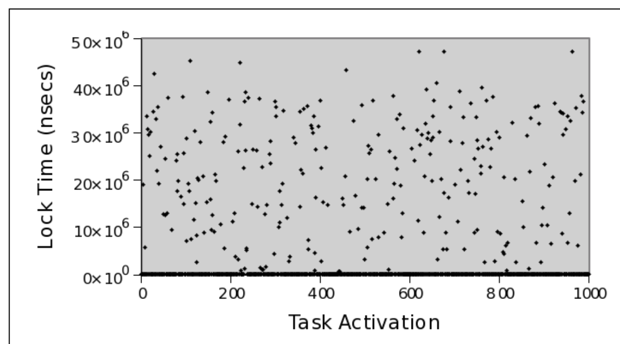

Figure 7. Blocking time (high priority task using PI)

Regarding the response time (as can be seen in the histogram of Figure 8) it was consistent with the blocking time sustained, with a maximum of nearly 3 times the

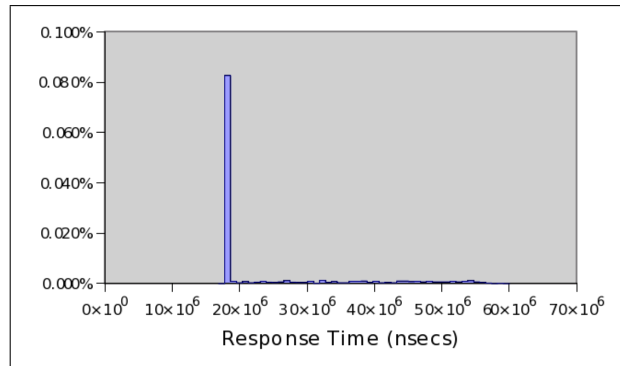

Figure 8. Histogram of response time (high priority task using PI)

size of the critical section, in accordance with the task set. It can be seen in Table 3 the worst-case response time observed is 64,157,591 ns. The theoretical worst-case response time for this test would be, with an appropriate synchronized activation, 68 ms, or 17 ms own critical section of task T0 added to 34 ms of task T1 and 17 ms of task T2. In this test, there is a good approximation of the theoretical limit.

Protocol:	PI	IPC
Average response time:	22,798,549 ns	21,014,311 ns
Std dev:	11,319,355 ns	8,723,159 ns
Max:	64,157,591 ns	50,811,328 ns

Table 3. Average response times and standard deviations

5.2. Results of the Use of the IPC mutex

Using IPC, it can be noted in the histogram of Figure 9 that the task of highest priority presented, with low frequency, varying values of activation latency (seen in the tail of the histogram). Waiting times set by the resource appear in Figure 10, which is expected according to the definition of the protocol implemented. A tail appears in the histogram of response time (Figure 11) due to activation latency (higher values, but with only a few occurrences).

As it can be seen in Table 3, the worst-case response time observed is (maximum) 50,811,328 ns. In this test, the theorical limit is 51 ms, ie, 17 ms own critical section of task T0 added to 34 ms of task T1. Also in this test there is a good approximation of the theorical limit.

5.3. Comparison between PI and IPC

One can observe that, in general, IPC has behavior similar to PI. The differences appear in the lock time where, by definition, in uniprocessor systems, the resource is always

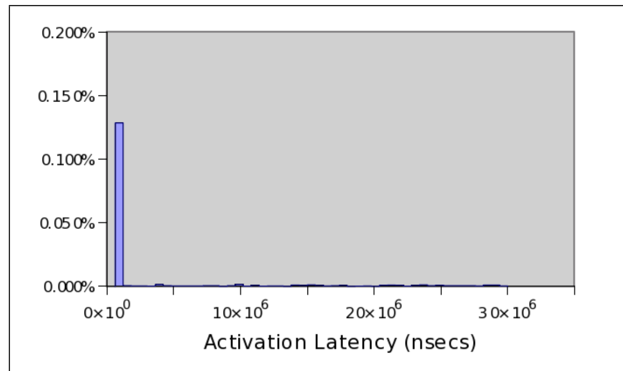

Figure 9. Histogram of activation latencies (high priority task using IPC)

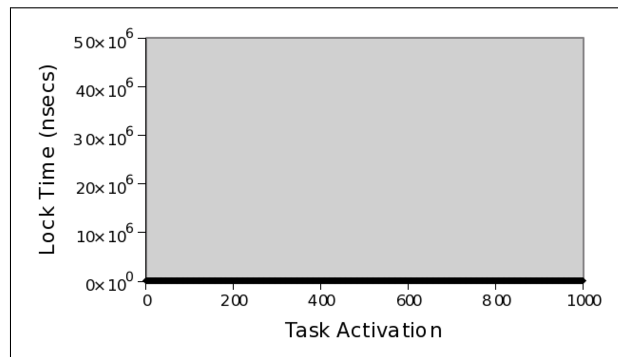

Figure 10. Blocking time (high priority task using IPC)

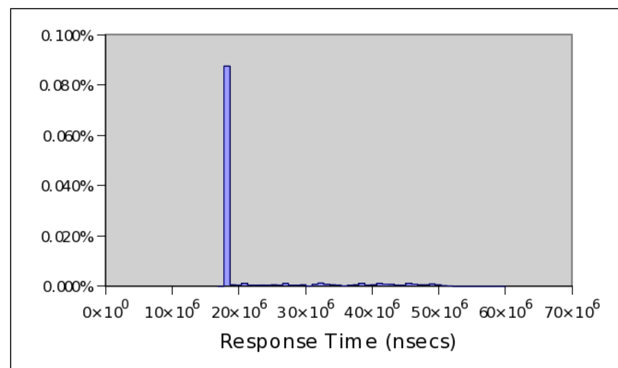

Figure 11. Histogram of response time (high priority task using IPC)

available when using IPC protocol. For the PI protocol, the blocking time will appear with the primitive lock, and this time may be longer than that with IPC. In IPC the blocking time appears before the activation time, and it has a maximum length of a single critical section (in the conditions described above).

According to Table 3, the IPC protocol presented standard deviation and average response time smaller than PI. Another important point in Table 3 is that the worst-case response time observed in the IPC test was almost a critical section smaller than the PI (the size of a critical section is 17 ms, and the difference between the worst case of IPC and PI is around 14 ms).

Figure 12 shows the tail of the response time histograms of IPC and PI combined. In this figure, the response times of the IPC protocol concentrates on lower values. For the PI, these are distributed more uniformly to higher values, indicating an average response time smaller for the IPC protocol. This histogram also indicates in its final portion that the worst case, as it was also observed in Table 3, has a difference of one critical section in favor of the IPC protocol. This difference in the worst case was reported in the figure by two vertical lines, where the distance between them is about the duration of one critical section. Table 4 summarizes the results qualitatively.

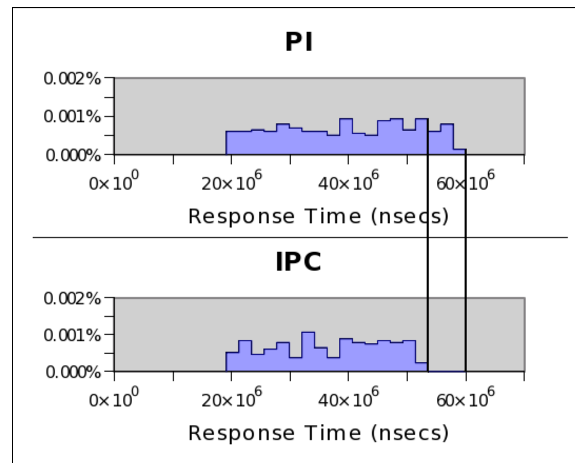

Figure 12. Histogram of the response time of the high priority task

Protocol	**PI**	**IPC**
Activation Latency	Not varied	Varied
Blocking time	Varied	Not varied
Response time	Blocking time dependent	Latency dependent

Table 4. Summary of expected results

6. Implementation Overhead

We define overhead as any decrease in the system's ability to produce useful work. Thus, for this study, the overhead will be considered as the reducing of the CPU time available to the rest of the system, given the presence of a set of higher priority tasks sharing resources protected by PI or IPC.

To evaluate the protocol implemented in terms of overhead imposed on the system, we used a set of test tasks as specified in Table 5. In the same table, it is presented

the tasks configurations, some of which are identical to the tasks used to evaluate the protocol in the previous section. For example, for task T0’, the size of the critical section is equal to the size of the critical section of task T0 of the previous test (represented by "== T0").

To perform an estimative of the overhead, it was created a measuring task with priority 51 (with policy SCHED_FIFO). This priority is above the default priority of threaded irq handlers [Rostedt and Hart 2007] and softirqs [Love 2005] . This was done to keep the measuring task above the interference of the mechanisms of interrupt handling and work postponement of Linux. Every CPU time that remains (not used by the test tasks synchronized by IPC or PI) is then assigned to the measurement task. Both the measurement task and the task set synchronized by IPC or PI were fixed to a single CPU (CPU 0 in a system with 2 cores.)

The measurement task is activated before the activation of real-time tasks and ends after they terminate. In each test iteration, the measurement task runs for 9 seconds, and the higher priority tasks begin 1 second after it starts. As shown in Table 5, task T0’ executes 10 activations, the others will run until the end of this task.

Task	**T0’**	**T1’**	**T2’**	**T3’**	**T4’**	**T5’**	**T6’**
Priority	70	65	64	63	62	61	60
Activation interval	== T0	== T1	== T1	== T1	== T2	== T2	== T2
Resource	R1	R1, R2	R1, R2	R1, R2	R2	R2	R2
Critical section size	== T0	== T1	== T1	== T1	== T2	== T2	== T2
Number of activ.	10	T0’ dep	T0’ dep	T0’ dep	T0’ dep	T0’ dep	T0’ dep

Table 5. Configuration of the set of tasks

To obtain the overhead estimative, the measurement task is executed in an infinite loop incrementing a variable by the time specified above (9 seconds). The overhead will be noticed by how much the measurement task could increment a count, taking into account the execution of the set of tasks synchronized by IPC or PI. The values of the counts made by the measurement task are presented in Table 6, which was ordered to facilitate visual comparison.

Table 7 presents the basic statistical data related to the samples presented in Table 6.

To evaluate the results we used the statistical hypothesis test for averages with unknown variance (Student t-test). By hypothesis, the overhead of PI and IPC are equal,

IPC	PI
187,882,717	188,776,389
188,035,384	189,155,169
188,160,733	189,202,563
188,194,113	189,263,630
188,207,825	189,331,186
188,240,432	189,361,353
188,563,788	189,387,326
188,603,802	189,418,120
188,616,385	189,428,218
188,703,718	189,437,569
188,736,889	189,447,533
188,742,876	189,471,453
188,935,538	189,475,286
188,952,045	189,489,740
188,962,343	189,492,896
188,993,374	189,494,791
189,000,638	189,569,661
189,178,721	189,572,258
189,203,245	189,604,953
189,307,878	189,638,715
189,478,899	189,696,190
189,536,986	189,778,227
189,674,412	189,825,606
189,785,580	189,867,362
189,858,951	190,046,853
189,900,585	190,207,326
190,030,444	190,252,943
190,047,436	190,342,313
190,066,156	190,349,981
190,105,987	190,387,518
190,328,052	190,538,130
190,338,011	190,539,875

Table 6. Counter values of measurement task

Protocol	IPC	PI
Average(μ)	189,136,685.72	189,682,847.91
Var.(S_x^2)	524,003,070,588.27	191,603,683,258.35
Minimum	187,882,717	188,776,389
Maximum	190,338,011	190,539,875

Table 7. Basic statistics of the found values

ie, the average of IPC and PI are equal ($H_0 : \mu_{PI} = \mu_{IPC}$).

The data presented in Table 8, which provides the data necessary for the hypothe-

sis test, was obtained from the data showed in Table 7 plus the information of the number of samples (n = *32*)

$Sa^2_{IPC,PI}$	357,803,376,923.31
$Sa_{IPC,PI}$	598,166.68
n	32
α	0.1%
$d.f.$	60
t	-3.65

Table 8. Student's t-test data

6.1. Analysis of the Results

Because the data produced the value of t = 3.65, which does not belong to the region of acceptance (in t-Student distribution), the test rejects H_0 with a significance level of 0.1 %. At significance level (α) of 0.1%, the collected data indicates a difference between PI and IPC. There is a probability smaller than 0.1% that the differences observed on the presented data are from casual factors of the system only.

These differences are likely due to the fast path of the PI implementation. In IPC, there is always a need of priority verification, and this can not be performed atomically. Another point is that if a task with priority below the priority ceiling of a given resource acquires that resource, its priority has to be changed, and this may influence the overhead. As those tests show, there is a reasonable probability of tasks finding resources available, not always the priorities propagation algorithm (PI) will run. But almost always there will be priority adjustments (IPC), except for the task that defines the priority ceiling of the resource.

7. Conclusions

Task synchronization is fundamental in multitasking and/or multithread systems, specially in real-time systems. In addition to protection against race conditions, these mechanisms must prevent the emergence of uncontrolled priority inversions, which could cause the missing of deadline, leading real-time applications to present incorrect behavior, and possibly harmful consequences (depending on the application). In this context, it was proposed an alternative for some applications to the protocol implemented in the real-time Linux branch.

The IPC protocol may be suitable for dedicated applications that use architectures without instruction compare and exchange because, in this way, the implementation may not use the fast path (via atomic instructions). Another advantage of the IPC is that it generates less context switches than the PI, inducing faster response times due to switching overhead as well as lower failure rates in the TLB.

One of the disadvantages of the IPC for wider use is the need for manual determination of the priority ceiling of IPC mutexes. But this is not a problem for automation and control applications for example. Dedicated device-drivers are fully aware of the priorities of the tasks that access them, justifying the manual setting of the ceiling in this case.

As seen in the tests, the PI protocol may be more appropriate if the latency of activation is important. But if the blocking time is more relevant, IPC may be the best solution. In terms of average response time, the two solutions were similar, but IPC showed lower average response time probably due to the latency of activation being less than the waiting time of the PI. Another point in favor of the IPC protocol appears when we compare the difference in the worst-case response time observed in the tests since the IPC case was about a critical section smaller than in the PI case, as can be seen in Table 3. The PI protocol has a response time that may vary depending of the pattern of resource sharing and sequences of activation, which does not occur with IPC. Its blocking time will always be at most one critical section.

Although blocking/response times are smaller in the IPC, tests show that the overhead of IPC implemented is greater than the native PI in PREEMPT-RT. This overhead is most likely caused by the absence of a fast path in the implementation of IPC. There is a set of operations on lock/unlock that can not be executed atomically as in PI. These operations involve priority changes and tracking mutexes acquired by tasks.

References

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.

Love, R. (2005). *Linux Kernel Development (Novell Press)*. Novell Press.

Molnar, I. Preempt-rt. http://www.kernel.org/pub/linux/kernel/projects/rt - Last access 01/21, 2010.

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*. Available at `http://yodaiken.com/papers/inherit.pdf`.

APÊNDICE B – Códigos Fonte

Listagem de código: immpc_mutex.h

```
1  /*
2   * Alternate RT Mutexes: blocking mutual exclusion locks with Priority Ceiling support
3   *
4   * started by Andreu Carminati, highly based on original PI work from:
5   *
6   *  Copyright (C) 2004-2006 Red Hat, Inc., Ingo Molnar <mingo@redhat.com>
7   *  Copyright (C) 2006, Timesys Corp., Thomas Gleixner <tglx@timesys.com>
8   *
9   * This file contains the public data structure and API definitions.
10  */
11
12 #ifndef __LINUX_immpc_mutex_H
13 #define __LINUX_immpc_mutex_H
14
15 #include <linux/rtmutex.h>
16 #include <linux/plist.h>
17 #include <linux/list.h>
18 #include <linux/spinlock_types.h>
19
20 //#define DEBUG_IMMPC_MUTEX
21 //#define IPC_DEBUG
22 //#define CONFIG_IPC_DEBUG_NO_FP
23 //#define CONFIG_IPC_DYNAMIC_CEILING
24
25
26 #define CONFIG_IMMPC_FASTPATH
27 #define CONFIG_IMMPC_DELAYED_PRIO_ADJ
28
29
30 //#define IPC_DEBUG_CONSISTENCY
31
32 #define DYNAMIC_CEILING 0xFFFF
33
34 struct task_struct;
35
36 /**
37  * The immpc_mutex structure
38  *
39  * @wait_lock: spinlock to protect the structure
40  * @wait_list: pilist head to enqueue waiters in priority order
```

```
41   * @owner:   the mutex owner
42   * @ceiling:    the mutex ceiling
43   */
44  struct immpc_mutex {
45      raw_spinlock_t wait_lock;
46      struct plist_head wait_list;
47      struct list_head      on_task_entry;
48      struct task_struct    *owner;
49      int           ceiling;
50
51  #ifdef CONFIG_DEBUG_RT_MUTEXES
52      int           save_state;
53      const char      *name, *file;
54      //int          line;
55      //void           *magic;
56  #endif
57  };
58
59  struct immpc_mutex_waiter {
60      struct plist_node      task_wait_entry;
61      struct task_struct    *task;
62      struct immpc_mutex    *lock;
63  #ifdef CONFIG_DEBUG_RT_MUTEXES
64      unsigned long      ip;
65      struct pid      *deadlock_task_pid;
66      struct immpc_mutex    *deadlock_lock;
67  #endif
68  };
69
70  struct ipc_synchronization_ctx{
71  //#if defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
72      int need_prio_change;
73  //#endif
74      raw_spinlock_t lock;
75      struct list_head mutex_list;
76      //void (*initialize)(struct ipc_synchronization_ctx*);
77      //void (*callback)(void);
78  };
79
80  extern void __delayed_prio_change(void);
81
82  #if defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
83  #define check_delayed_prio_change() \
84      if(unlikely(current->ipc_ctx.need_prio_change)){ \
85          __delayed_prio_change(); \
86      }
87  #else
88  #define check_delayed_prio_change() while(0);
89  #endif
90
91  #ifdef DEBUG_IMMPC_MUTEX
92
93  #else
94  static inline void immpc_mutex_dbg_msg(const char *msg) {}
```

```
95  #endif
96  //struct rt_mutex_waiter;
97  //struct hrtimer_sleeper;
98
99  //#ifdef CONFIG_DEBUG_RT_MUTEXES
100 // extern int rt_mutex_debug_check_no_locks_freed(const void *from,
101 //                      unsigned long len);
102 // extern void rt_mutex_debug_check_no_locks_held(struct task_struct *task);
103 //#else
104  static inline int immpc_mutex_debug_check_no_locks_freed(const void *from,
105                              unsigned long len)
106  {
107    return 0;
108  }
109
110 #define immpc_mutex_setprio rt_mutex_setprio
111
112 extern void immpc_mutex_debug_task_free(struct task_struct *tsk);
113
114
115 #define __immpc_mutex_INITIALIZER(mutexname, prio) \
116    { .wait_lock = __RAW_SPIN_LOCK_UNLOCKED(mutexname.wait_lock) \
117    , .wait_list = PLIST_HEAD_INIT_RAW(mutexname.wait_list, mutexname.wait_lock) \
118    , .owner = NULL \
119    , .ceiling = (MAX_RT_PRIO-1 - prio)}
120
121 #define DEFINE_immpc_mutex(mutexname, prio) \
122    struct immpc_mutex mutexname = __immpc_mutex_INITIALIZER(mutexname, prio)
123
124 /**
125  * immpc_mutex_is_locked - is the mutex locked
126  * @lock: the mutex to be queried
127  *
128  * Returns 1 if the mutex is locked, 0 if unlocked.
129  */
130 static inline int immpc_mutex_is_locked(struct immpc_mutex *lock)
131 {
132    return lock->owner != NULL;
133 }
134
135 extern void __immpc_mutex_init(struct immpc_mutex *lock, const char *name, int prio); //ok
136 extern void immpc_mutex_destroy(struct immpc_mutex *lock); //ok
137
138 extern void immpc_mutex_lock(struct immpc_mutex *lock); //ok
139 extern int immpc_mutex_lock_interruptible(struct immpc_mutex *lock, //ok
140                    int detect_deadlock); //ok
141 extern int immpc_mutex_lock_killable(struct immpc_mutex *lock, //ok
142                int detect_deadlock); //ok
143 extern int immpc_mutex_timed_lock(struct immpc_mutex *lock,
144                 struct hrtimer_sleeper *timeout,
145                 int detect_deadlock); //ok
146
147 extern int immpc_mutex_trylock(struct immpc_mutex *lock); //ok
148
```

```
149 extern void immpc_mutex_unlock(struct immpc_mutex *lock); //ok
150
151 extern void immpc_mutex_set_ceiling(struct immpc_mutex *lock, int newceiling);
152
153 #ifdef CONFIG_immpc_mutexES
154 # define INIT_immpc_mutexES(tsk)                    \
155     .pi_waiters = PLIST_HEAD_INIT_ATOMIC(tsk.pi_waiters, tsk.pi_lock),     \
156     INIT_immpc_mutex_DEBUG(tsk)
157 #else
158 # define INIT_immpc_mutexES(tsk)
159 #endif
160
161
162 /*INTERNAL DEFINITIONS:*/
163
164 /*
165  * lock->owner state tracking:
166  */
167 #define immpc_mutex_OWNER_PENDING     1UL
168 #define immpc_mutex_HAS_WAITERS   2UL
169 #define immpc_mutex_OWNER_MASKALL     3UL
170
171 #endif
```

Listagem de código: immpc_mutex.c

```
1  /*
2   * IPC-Mutexes: simple blocking mutual exclusion mutexes with Immediate
3   * Priority Ceiling Protocol support
4   *
5   *
6   * started by Andreu Carminati, highly based on original PI work from:
7   *
8   *  Copyright (C) 2004-2006 Red Hat, Inc., Ingo Molnar <mingo@redhat.com>
9   *  Copyright (C) 2005-2006 Timesys Corp., Thomas Gleixner <tglx@timesys.com>
10  *  Copyright (C) 2005 Kihon Technologies Inc., Steven Rostedt
11  *  Copyright (C) 2006 Esben Nielsen
12  *
13  * Adaptive Spinlocks:
14  *  Copyright (C) 2008 Novell, Inc., Gregory Haskins, Sven Dietrich,
15  *                                    and Peter Morreale,
16  * Adaptive Spinlocks simplification:
17  *  Copyright (C) 2008 Red Hat, Inc., Steven Rostedt <srostedt@redhat.com>
18  *
19  *  See Documentation/rt-mutex-design.txt for details.
20  */
21
22 #include <linux/spinlock.h>
23 #include <linux/module.h>
24 #include <linux/sched.h>
25 #include <linux/timer.h>
26 #include <linux/hardirq.h>
27 #include <linux/semaphore.h>
28 #include <linux/plist.h>
```

```
29 #include <linux/irqflags.h>
30
31 #include <linux/immpc_mutex.h>
32
33 #define ENABLE_PRIO_ADJUST 0x1
34 #define DISABLE_PRIO_ADJUST 0X0
35
36
37 static void immpc_mutex_adjust_prio(struct task_struct *);
38 static inline void __immpc_mutex_adjust_prio(struct task_struct *);
39
40 /*
41 * This task needs to postpone its priority adjustment.
42 */
43 static void postergate_prio_adj(struct task_struct* tsk)
44 {
45     tsk->ipc_ctx.need_prio_change = ENABLE_PRIO_ADJUST;
46 }
47
48 /*
49 * Return the mutex owner.
50 */
51 static inline struct task_struct *immpc_mutex_owner(struct immpc_mutex *lock)
52 {
53     return (struct task_struct *)
54         ((unsigned long)lock->owner & ~immpc_mutex_OWNER_MASKALL);
55 }
56
57 #if defined(IPC_DEBUG_CONSISTENCY)
58 /*
59 * Consistency check (the owner must be the task)
60 */
61 static inline void ipc_check_consistency(struct immpc_mutex *lock,
62                         struct task_struct *owner)
63 {
64     if(immpc_mutex_owner(lock) != owner){
65         printk("<1>IPC Expected owner: %p, real owner %p\n",
66                         owner, immpc_mutex_owner(lock));
67         BUG();
68     }
69 }
70 #else
71 static inline void ipc_check_consistency(struct immpc_mutex *lock,
72                             struct task_struct *owner){}
73 #endif
74
75 static inline void clear_immpc_mutex_waiters(struct immpc_mutex *lock)
76 {
77     lock->owner = (struct task_struct *)
78             ((unsigned long)lock->owner & ~immpc_mutex_HAS_WAITERS);
79 }
80
81 /*
82 * Verify if the immpc_mutex has at last, one waiter.
```

```
83   */
84   static inline int immpc_mutex_has_waiters(struct immpc_mutex *lock)
85   {
86      return !plist_head_empty(&lock->wait_list);
87   }
88
89   static void fixup_immpc_mutex_waiters(struct immpc_mutex *lock)
90   {
91      if (!immpc_mutex_has_waiters(lock)){
92         clear_immpc_mutex_waiters(lock);
93      }
94   }
95
96   /*
97   * Task has acquired the lock and must track it.
98   * lock->wait lock must be held (irq_save too).
99   */
100  static inline void task_add_immpc_mutex(struct task_struct *task, struct immpc_mutex *lock)
101  {
102     //plist_node_init(&lock->on_task_entry, lock->ceiling);
103     //plist_add(&lock->on_task_entry, &task->ipc_ctx.mutex_list);
104     //plist_node_init(&lock->on_task_entry, lock->ceiling);
105     list_add(&lock->on_task_entry, &task->ipc_ctx.mutex_list);
106  }
107
108  /*
109  * Task is no more the owner of lock, untrack it.
110  * Must be called with lock->wait_lock taken
111  */
112  static inline void task_del_immpc_mutex(struct task_struct *task,
113                              struct immpc_mutex *lock)
114  {
115     //plist_del(&lock->on_task_entry, &task->ipc_ctx.mutex_list);
116     list_del(&lock->on_task_entry);
117  }
118  /*
119  * for fastlock acquisition.
120  */
121  static inline void __track_immpc_mutex(struct immpc_mutex *lock, struct task_struct *task)
122  {
123     unsigned long flags;
124
125     task_add_immpc_mutex(task, lock);
126
127     #if defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
128     postergate_prio_adj(current);
129     #else
130     immpc_mutex_adjust_prio(current);
131     #endif
132
133     //postergate_prio_adj(task);
134  }
135
136  static inline void track_immpc_mutex(struct immpc_mutex *lock, struct task_struct *task)
```

```
137 {
138     unsigned long flags;
139 
140     //raw_spin_lock_irqsave(&lock->wait_lock, flags);
141     __track_immpc_mutex(lock, task);
142     //raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
143 
144 }
145 /*
146 * for fastunlock acquisition
147 */
148 static inline void __untrack_immpc_mutex(struct immpc_mutex *lock, struct task_struct *task)
149 {
150     unsigned long flags;
151 
152     task_del_immpc_mutex(task, lock);
153     //postergate_prio_adj(task);
154     #if defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
155     postergate_prio_adj(current);
156     #else
157     immpc_mutex_adjust_prio(current);
158     #endif
159 }
160 
161 static inline void untrack_immpc_mutex(struct immpc_mutex *lock, struct task_struct *task)
162 {
163     unsigned long flags;
164 
165     //raw_spin_lock_irqsave(&lock->wait_lock, flags);
166     __untrack_immpc_mutex(lock, task);
167     //raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
168 }
169 
170 /*
171 * Return the highest priority mutex waiter.
172 */
173 static inline struct immpc_mutex_waiter *
174 immpc_mutex_top_waiter(struct immpc_mutex *lock)
175 {
176     struct immpc_mutex_waiter *w;
177 
178     w = plist_first_entry(&lock->wait_list, struct immpc_mutex_waiter,
179                 task_wait_entry);
180     BUG_ON(w->lock != lock);
181 
182     return w;
183 }
184 
185 
186 /*
187 * Verify if the task is owning a immpc_mutex
188 */
189 static inline int task_has_immpc_mutex(struct task_struct *p)
190 {
```

```
191     //return !plist_head_empty(&p->ipc_ctx.mutex_list);
192     return !list_empty(&p->ipc_ctx.mutex_list);
193 }
194
195 /*
196 * Return the bigger mutex ceiling of the task.
197 */
198 static inline struct immpc_mutex *
199 task_top_immpc_mutex(struct task_struct *p)
200 {
201     struct list_head *pos, *head;
202     struct immpc_mutex *mutex1, *mutex2;
203
204     head = &p->ipc_ctx.mutex_list;
205     mutex1 = list_first_entry(head, struct immpc_mutex, on_task_entry);
206
207     list_for_each(pos, head){
208         mutex2 = list_entry(pos, struct immpc_mutex, on_task_entry);
209
210         if(mutex2->ceiling < mutex1->ceiling){
211             mutex1 = mutex2;
212         }
213     }
214
215     //return plist_first_entry(&p->ipc_ctx.mutex_list, struct immpc_mutex,
216     //            on_task_entry);
217
218     return mutex1;
219 }
220
221
222 static inline void init_lists(struct immpc_mutex *lock)
223 {
224     if (unlikely(!lock->wait_list.prio_list.prev)) {
225         plist_head_init_raw(&lock->wait_list, &lock->wait_lock);
226
227     }
228 }
229
230 /*
231 * Verify if exists a owner pending on a mutex
232 */
233 static inline unsigned long immpc_mutex_owner_pending(struct immpc_mutex *lock)
234 {
235     return (unsigned long)lock->owner & immpc_mutex_OWNER_PENDING;
236 }
237
238 /*
239  * We can speed up the acquire/release, if the architecture
240  * supports cmpxchg and if there's no debugging state to be set up
241  */
242 #if defined(__HAVE_ARCH_CMPXCHG) && defined(CONFIG_IMMPC_FASTPATH) && \
243         defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
244 # define immpc_mutex_cmpxchg(l,c,n) (cmpxchg(&l->owner, c, n) == c)
```

```
245 static inline void mark_immpc_mutex_waiters(struct immpc_mutex *lock)
246 {
247     unsigned long owner, *p = (unsigned long *) &lock->owner;
248 
249     do {
250         owner = *p;
251     } while (cmpxchg(p, owner, owner | immpc_mutex_HAS_WAITERS) != owner);
252 }
253 #else
254 # define immpc_mutex_cmpxchg(l,c,n) (0)
255 static inline void mark_immpc_mutex_waiters(struct immpc_mutex *lock)
256 {
257     lock->owner = (struct task_struct *)
258             ((unsigned long)lock->owner | immpc_mutex_HAS_WAITERS);
259 }
260 #endif
261 
262 static void
263 immpc_mutex_set_owner(struct immpc_mutex *lock, struct task_struct *owner,
264             unsigned long mask)
265 {
266     unsigned long val = (unsigned long)owner | mask;
267 
268     if (immpc_mutex_has_waiters(lock)){
269         val |= immpc_mutex_HAS_WAITERS;
270     }
271 
272     lock->owner = (struct task_struct *)val;
273 }
274 
275 
276 /*
277 * The task must block to wait for a ipc mutex.
278 */
279 static int task_blocks_on_immpc_mutex(struct immpc_mutex *lock,
280                 struct immpc_mutex_waiter *waiter,
281                 struct task_struct *task,
282                 int detect_deadlock, unsigned long flags)
283 {
284 
285     raw_spin_lock(&task->ipc_ctx.lock);
286 
287     waiter->task = task;
288     waiter->lock = lock;
289 
290     plist_node_init(&waiter->task_wait_entry, task->prio);
291     plist_add(&waiter->task_wait_entry, &lock->wait_list);
292 
293     raw_spin_unlock(&task->ipc_ctx.lock);
294 
295     return 1; /*TODO: eh isso mesmo???*/
296 }
297 
298 /*
```

```
299   * Calculate task priority from the waiter list priority
300   *
301   * Return task->normal_prio when the waiter list is empty or when
302   * the waiter is not allowed to do priority boosting
303   */
304  static int immpc_mutex_getprio(struct task_struct *task)
305  {
306      //struct immpc_mutex *lock;
307      //unsigned long flags;
308
309      if (likely(!task_has_immpc_mutex(task)))
310          return task->normal_prio;
311  #ifdef    IPC_DEBUG
312      printk("<1>(%d)␣Mutex␣prio:␣%d\n",
313      current->pid, task_top_immpc_mutex(task)->ceiling);
314  #endif
315
316
317      return min(task_top_immpc_mutex(task)->ceiling,
318              task->normal_prio);
319  }
320
321  /*
322   * Adjust the priority of a task.
323   *
324   * This can be both boosting and unboosting. task->pi_lock must be held.
325   */
326  static inline void __immpc_mutex_adjust_prio(struct task_struct *task)
327  {
328      int prio = immpc_mutex_getprio(task);
329
330      if (unlikely(task->prio != prio)){
331          immpc_mutex_setprio(task, prio);
332  #ifdef IPC_DEBUG
333          printk("<1>(%d)␣task->prio:␣%d,␣prio:␣%d\n", current->pid,
334                                  task->prio, prio);
335  #endif
336      }
337  }
338
339
340  /*
341   * Adjust task priority (undo boosting).
342   * (Note: We do this outside of the protection of lock->wait_lock to
343   * allow the lock to be taken while or before we readjust the priority
344   * of task. We do not use the atomic_spin_xx_mutex() variants here as we are
345   * outside of the debug path.)
346   */
347  static void immpc_mutex_adjust_prio(struct task_struct *task)
348  {
349      unsigned long flags;
350
351      raw_spin_lock_irqsave(&task->ipc_ctx.lock, flags);
352      __immpc_mutex_adjust_prio(task);
```

```
353     raw_spin_unlock_irqrestore(&task->ipc_ctx.lock, flags);
354 }
355
356
357 void __delayed_prio_change(void)
358 {
359     struct ipc_synchronization_ctx* ctx;
360     //int saved_state;
361
362     ctx = &current->ipc_ctx;
363 #ifdef IPC_DEBUG
364     printk("<1>==IPC:␣delayed␣prio␣change,␣before:%d\n", current->prio);
365 #endif
366     immpc_mutex_adjust_prio(current);
367 #ifdef IPC_DEBUG
368     printk("<1>==IPC:␣delayed␣prio␣change,␣after:%d\n", current->prio);
369 #endif
370     ctx->need_prio_change = DISABLE_PRIO_ADJUST;
371 }
372
373 /*
374  * Wake up the next waiter on the lock.
375  *
376  * Remove the top waiter from the current tasks waiter list and from
377  * the lock waiter list. Set it as pending owner. Then wake it up.
378  *
379  * Called with lock->wait_lock held.
380  */
381
382 /*
383 TODO: aqui, devemos nos certificar de que existem tarafas esperando antes de acorda-las.
384 */
385 static void wakeup_next_waiter(struct immpc_mutex *lock, int savestate)
386 {
387     struct immpc_mutex_waiter *waiter;
388     struct task_struct *pendowner;
389
390     waiter = immpc_mutex_top_waiter(lock);
391     plist_del(&waiter->task_wait_entry, &lock->wait_list);
392
393
394     pendowner = waiter->task;
395     waiter->task = NULL;
396
397 #ifdef IPC_DEBUG
398     printk("<1>IPC␣pid:␣%d␣kicks␣pid:␣%d\n", current->pid, pendowner->pid);
399 #endif
400     /*
401      * Do the wakeup before the ownership change to give any spinning
402      * waiter grantees a headstart over the other threads that will
403      * trigger once owner changes.
404      */
405     if (!savestate)
406         wake_up_process(pendowner);
```

```
407     else {
408        /*
409         * We can skip the actual (expensive) wakeup if the
410         * waiter is already running, but we have to be careful
411         * of race conditions because they may be about to sleep.
412         *
413         * The waiter-side protocol has the following pattern:
414         * 1: Set state != RUNNING
415         * 2: Conditionally sleep if waiter->task != NULL;
416         *
417         * And the owner-side has the following:
418         * A: Set waiter->task = NULL
419         * B: Conditionally wake if the state != RUNNING
420         *
421         * As long as we ensure 1->2 order, and A->B order, we
422         * will never miss a wakeup.
423         *
424         * Therefore, this barrier ensures that waiter->task = NULL
425         * is visible before we test the pendowner->state.  The
426         * corresponding barrier is in the sleep logic.
427         */
428        smp_mb();
429
430        /* If !RUNNING && !RUNNING_MUTEX */
431        if (pendowner->state & ~TASK_RUNNING_MUTEX)
432           wake_up_process_mutex(pendowner);
433     }
434
435     immpc_mutex_set_owner(lock, pendowner, immpc_mutex_OWNER_PENDING);
436
437  }
438
439  #define STEAL_LATERAL 1 /*steal from tasks with same prio*/
440  #define STEAL_NORMAL  0
441
442  /*
443   * Note that RT tasks are excluded from lateral-steals to prevent the
444   * introduction of an unbounded latency
445   */
446  static inline int lock_is_stealable(struct task_struct *task,
447                      struct task_struct *pendowner, int mode)
448  {
449      if (mode == STEAL_NORMAL || rt_task(task)) {
450          if (task->prio >= pendowner->prio)
451              return 0;
452      } else if (task->prio > pendowner->prio)
453          return 0;
454
455      return 1;
456  }
457
458  /*
459   * Optimization: check if we can steal the lock from the
460   * assigned pending owner [which might not have taken the
```

```
461     * lock yet]:
462     */
463    static inline int try_to_steal_lock(struct immpc_mutex *lock,
464                    struct task_struct *task, int mode)
465    {
466       struct task_struct *pendowner = immpc_mutex_owner(lock);
467
468       if (!immpc_mutex_owner_pending(lock))
469          return 0;
470
471       if (pendowner == task)
472          return 1;
473
474       raw_spin_lock(&pendowner->ipc_ctx.lock);
475       if (!lock_is_stealable(task, pendowner, mode)) {
476          raw_spin_unlock(&pendowner->ipc_ctx.lock);
477          return 0;
478       }
479
480       /*
481        * Check if a waiter is enqueued on the pending owners
482        * pi_waiters list. Remove it and readjust pending owners
483        * priority.
484        */
485       if (likely(!immpc_mutex_has_waiters(lock))) {
486          raw_spin_unlock(&pendowner->ipc_ctx.lock);
487          return 1;
488       }
489       raw_spin_unlock(&pendowner->ipc_ctx.lock);
490
491       return 1;
492    }
493
494    /*
495     * Try to take an rt-mutex
496     *
497     * This fails
498     * - when the lock has a real owner
499     * - when a different pending owner exists and has higher priority than current
500     *
501     * Must be called with lock->wait_lock held.
502     */
503    static int do_try_to_take_immpc_mutex(struct immpc_mutex *lock, int mode)
504    {
505       /*blocks fast path acquisition*/
506       mark_immpc_mutex_waiters(lock);
507
508    // TODO tirar este if, o caso de baixo cobre
509       if(likely(!immpc_mutex_is_locked(lock))){
510             /* We got the lock. */
511    #ifdef   IPC_DEBUG
512          printk("<1>(%d)␣We␣got␣the␣lock␣path␣2\n", current->pid);
513    #endif
514
```

```
515         immpc_mutex_set_owner(lock, current, 0);
516
517         return 1;
518     }
519     if (immpc_mutex_owner(lock) && !try_to_steal_lock(lock, current, mode)){
520         return 0;
521     }
522
523     /* We got the lock. */
524 #ifdef IPC_DEBUG
525     printk("<1>(%d) We got the lock\n", current->pid);
526 #endif
527
528     immpc_mutex_set_owner(lock, current, 0);
529
530     return 1;
531 }
532
533 static inline int try_to_take_immpc_mutex(struct immpc_mutex *lock)
534 {
535     return do_try_to_take_immpc_mutex(lock, STEAL_NORMAL);
536 }
537
538 /*
539  * Remove a waiter from a lock
540  *
541  * Must be called with lock->wait_lock held
542  */
543 static void remove_waiter(struct immpc_mutex *lock,
544             struct immpc_mutex_waiter *waiter,
545             unsigned long flags)
546 {
547     //int first = (waiter == immpc_mutex_top_waiter(lock));
548     //struct task_struct *owner = immpc_mutex_owner(lock);
549
550     raw_spin_lock(&current->ipc_ctx.lock);
551
552     plist_del(&waiter->task_wait_entry, &lock->wait_list);
553     waiter->task = NULL;
554
555     raw_spin_unlock(&current->ipc_ctx.lock);
556 }
557
558 /**
559  * __rt_mutex_slowlock() - Perform the wait-wake-try-to-take loop
560  * @lock:          the rt_mutex to take
561  * @state:         the state the task should block in (TASK_INTERRUPTIBLE
562  *         or TASK_UNINTERRUPTIBLE)
563  * @timeout:         the pre-initialized and started timer, or NULL for none
564  * @waiter:        the pre-initialized rt_mutex_waiter
565  * @detect_deadlock:  passed to task_blocks_on_rt_mutex
566  *
567  * lock->wait_lock must be held by the caller.
568  */
```

```
569 /*
570 TODO
571 */
572 static int __sched
573 __immpc_mutex_lock_path(struct immpc_mutex *lock, int state,
574             struct hrtimer_sleeper *timeout,
575             struct immpc_mutex_waiter *waiter,
576             int detect_deadlock, unsigned long flags)
577 {
578     int ret = 0;
579 
580     for (;;) {
581         /* Try to acquire the lock: */
582         if (try_to_take_immpc_mutex(lock)){
583             break;
584         }
585 
586         /*
587          * TASK_INTERRUPTIBLE checks for signals and
588          * timeout. Ignored otherwise.
589          */
590         if (unlikely(state == TASK_INTERRUPTIBLE)) {
591             /* Signal pending? */
592             if (signal_pending(current))
593                 ret = -EINTR;
594             if (timeout && !timeout->task)
595                 ret = -ETIMEDOUT;
596             if (ret)
597                 break;
598         }
599 
600         /*
601          * waiter->task is NULL the first time we come here and
602          * when we have been woken up by the previous owner
603          * but the lock got stolen by a higher prio task.
604          */
605         if (!waiter->task) {
606             /*TODO: este retono est   certo?*/
607             task_blocks_on_immpc_mutex(lock, waiter, current,
608                             detect_deadlock, flags);
609             /*
610              * If we got woken up by the owner then start loop
611              * all over without going into schedule to try
612              * to get the lock now:
613              */
614             if (unlikely(!waiter->task)) {
615                 /*
616                  * Reset the return value. We might
617                  * have returned with -EDEADLK and the
618                  * owner released the lock while we
619                  * were walking the pi chain.
620                  */
621                 ret = 0;
622                 continue;
```

```
623                 }
624             }
625 
626             raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
627 
628             if (waiter->task){
629 #ifdef  IPC_DEBUG
630                 printk("<1>IPC␣pid␣%d␣sleep\n", current->pid);
631 #endif
632                 schedule();
633 #ifdef  IPC_DEBUG
634                 printk("<1>IPC␣pid␣%d␣wakeup\n", current->pid);
635 #endif
636             }
637             raw_spin_lock_irqsave(&lock->wait_lock, flags);
638 
639             set_current_state(state);
640         }
641 
642         return ret;
643 }
644 
645 
646 static int __sched immpc_mutex_slowlock(struct immpc_mutex *lock, int state,
647                 struct hrtimer_sleeper *timeout,
648                 int detect_deadlock)
649 {
650         int ret = 0;//, saved_lock_depth = -1;
651         struct immpc_mutex_waiter waiter;
652         unsigned long flags;
653 
654         waiter.task = NULL;
655 
656         raw_spin_lock_irqsave(&lock->wait_lock, flags);
657         init_lists(lock);
658 
659         /* Try to acquire the lock again: */
660         if (try_to_take_immpc_mutex(lock)) {
661 #ifdef  IPC_DEBUG
662             printk("<1>(%d)␣Try␣to␣take␣mutex␣success\n", current->pid);
663 #endif
664             __track_immpc_mutex(lock, current);
665 
666             raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
667 
668  #ifdef  IPC_DEBUG
669             printk("<1>(%d)␣Lock␣added␣on␣task␣list\n", current->pid);
670  #endif
671 
672             return 0;
673         }
674 
675         set_current_state(state);
676 
```

```
677     /* Setup the timer, when timeout != NULL */
678     if (unlikely(timeout)) {
679         hrtimer_start_expires(&timeout->timer, HRTIMER_MODE_ABS);
680         if (!hrtimer_active(&timeout->timer))
681             timeout->task = NULL;
682     }
683
684     ret = __immpc_mutex_lock_path(lock, state, timeout, &waiter,
685                 detect_deadlock, flags);
686
687     set_current_state(TASK_RUNNING);
688
689     if (unlikely(waiter.task)){
690         remove_waiter(lock, &waiter, flags);
691     }
692
693     /*
694      * try_to_take_rt_mutex() sets the waiter bit
695      * unconditionally. We might have to fix that up.
696      */
697     fixup_immpc_mutex_waiters(lock);
698
699     /* Remove pending timer: */
700     if (unlikely(timeout))
701         hrtimer_cancel(&timeout->timer);
702
703     if(likely(!ret)){
704         /*aqui temos que adicionar o lock na lista da task e ajustar a prioridade da tarefa*/
705         //task_add_immpc_mutex(current, lock);
706         /*adjust only if we*/
707         //#if defined(CONFIG_IMMPC_FASTPATH)
708         //postergate_prio_adj(current);
709         //#else
710         //immpc_mutex_adjust_prio(current);
711         //#endif
712         //ipc_check_consistency(lock, current);
713         __track_immpc_mutex(lock, current);
714     }
715
716     raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
717
718     return ret;
719 }
720
721 /*
722  * Slow path try-lock function:
723  */
724 static inline int
725 immpc_mutex_slowtrylock(struct immpc_mutex *lock)
726 {
727     unsigned long flags;
728     int ret = 0;
729
730     raw_spin_lock_irqsave(&lock->wait_lock, flags);
```

```
731
732     if (likely(immpc_mutex_owner(lock) != current)) {
733
734         init_lists(lock);
735
736         ret = try_to_take_immpc_mutex(lock);
737         /*
738          * try_to_take_rt_mutex() sets the lock waiters
739          * bit unconditionally. Clean this up.
740          */
741         fixup_immpc_mutex_waiters(lock);
742     }
743
744     raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
745
746     return ret;
747 }
748
749 static void __sched immpc_mutex_slowunlock(struct immpc_mutex* lock)
750 {
751     unsigned long flags;
752
753     raw_spin_lock_irqsave(&lock->wait_lock, flags);
754
755     ipc_check_consistency(lock, current);
756
757     //task_del_immpc_mutex(current, lock);
758     __untrack_immpc_mutex(lock, current);
759
760 /*otimiza  o para uniprocessor*/
761     if (likely(!immpc_mutex_has_waiters(lock))) {
762         lock->owner = NULL;
763     } else {
764 #ifdef IPC_DEBUG
765     printk("<1>(%d)␣Waking␣up␣next␣waiter\n", current->pid);
766 #endif
767          wakeup_next_waiter(lock, 0);
768     }
769
770     //postergate_prio_adj(current);
771
772     raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
773 }
774 /*
775  * debug aware fast / slowpath lock,trylock,unlock
776  *
777  * The atomic acquire/release ops are compiled away, when either the
778  * architecture does not support cmpxchg or when debugging is enabled.
779  */
780 static inline int
781 immpc_mutex_fastlock(struct immpc_mutex *lock, int state,
782             int detect_deadlock,
783             int (*slowfn)(struct immpc_mutex *lock, int state,
784                 struct hrtimer_sleeper *timeout,
```

```
785             int detect_deadlock))
786 {
787    int result = 0;
788    unsigned long flags;
789
790    raw_local_irq_save(flags);
791    if (likely(immpc_mutex_cmpxchg(lock, NULL, current))) {
792       track_immpc_mutex(lock, current);
793  #ifdef IPC_DEBUG
794       printk("<1>(%d)␣Lock␣acquired␣by␣fp\n", current->pid);
795  #endif
796       raw_local_irq_restore(flags);
797    } else {
798       raw_local_irq_restore(flags);
799
800       result = slowfn(lock, state, NULL, detect_deadlock);
801    }
802    ipc_check_consistency(lock, current);
803    return result;
804 }
805
806
807 static inline int
808 immpc_mutex_timed_fastlock(struct immpc_mutex *lock, int state,
809          struct hrtimer_sleeper *timeout, int detect_deadlock,
810          int (*slowfn)(struct immpc_mutex *lock, int state,
811                   struct hrtimer_sleeper *timeout,
812                   int detect_deadlock))
813 {
814    unsigned long flags;
815
816    raw_local_irq_save(flags);
817
818    if (!detect_deadlock && likely(immpc_mutex_cmpxchg(lock, NULL, current))) {
819       track_immpc_mutex(lock, current);
820  #ifdef IPC_DEBUG
821       printk("<1>(%d)␣Lock␣acquired␣by␣fp␣(timed)\n", current->pid);
822  #endif
823       raw_local_irq_restore(flags);
824
825       return 0;
826    } else {
827       raw_local_irq_restore(flags);
828
829       return slowfn(lock, state, timeout, detect_deadlock);
830    }
831 }
832
833 static inline int
834 immpc_mutex_fasttrylock(struct immpc_mutex *lock,
835             int (*slowfn)(struct immpc_mutex *lock))
836 {
837    unsigned long flags;
838
```

```
839     raw_local_irq_save(flags);
840
841     if (likely(immpc_mutex_cmpxchg(lock, NULL, current))) {
842         track_immpc_mutex(lock, current);
843  #ifdef IPC_DEBUG
844         printk("<1>(%d)␣Lock␣acquired␣by␣fp␣(try)\n", current->pid);
845  #endif
846         raw_local_irq_restore(flags);
847         return 1;
848     }
849     raw_local_irq_restore(flags);
850
851     return slowfn(lock);
852 }
853
854 static inline void
855 immpc_mutex_fastunlock(struct immpc_mutex *lock,
856             void (*slowfn)(struct immpc_mutex *lock))
857 {
858     unsigned long flags;
859     ipc_check_consistency(lock, current);
860
861     raw_local_irq_save(flags);
862
863     if (likely(immpc_mutex_cmpxchg(lock, current, NULL))){
864         untrack_immpc_mutex(lock, current);
865 #ifdef IPC_DEBUG
866         printk("<1>(%d)␣Lock␣released␣by␣fp\n", current->pid);
867 #endif
868         raw_local_irq_restore(flags);
869     }else {
870         raw_local_irq_restore(flags);
871
872         slowfn(lock);
873     }
874 }
875
876 /*======================================================================*/
877 /*======================== Public Interface ===========================*/
878 /*======================================================================*/
879
880 /**
881  * __rt_mutex_init - initialize the rt lock
882  *
883  * @lock: the rt lock to be initialized
884  *
885  * Initialize the rt lock to unlocked state.
886  *
887  * Initializing of a locked rt lock is not allowed
888  */
889
890 void __immpc_mutex_init(struct immpc_mutex *lock, const char *name, int prio)
891 {
892     lock->owner = NULL;
```

```
893     raw_spin_lock_init(&lock->wait_lock);
894     plist_head_init_raw(&lock->wait_list, &lock->wait_lock);
895     lock->ceiling = prio;
896 
897 }
898 
899 EXPORT_SYMBOL_GPL(__immpc_mutex_init);
900 
901 /**
902  * rt_mutex_destroy - mark a mutex unusable
903  * @lock: the mutex to be destroyed
904  *
905  * This function marks the mutex uninitialized, and any subsequent
906  * use of the mutex is forbidden. The mutex must not be locked when
907  * this function is called.
908  */
909 void immpc_mutex_destroy(struct immpc_mutex *lock)
910 {
911     WARN_ON(immpc_mutex_is_locked(lock));
912 #ifdef CONFIG_DEBUG_RT_MUTEXES
913     lock->magic = NULL;
914 #endif
915 }
916 
917 EXPORT_SYMBOL_GPL(immpc_mutex_destroy);
918 
919 
920 /**
921  * rt_mutex_lock - lock a rt_mutex
922  *
923  * @lock: the rt_mutex to be locked
924  */
925 void __sched immpc_mutex_lock(struct immpc_mutex *lock)
926 {
927     might_sleep();
928 #ifdef  IPC_DEBUG
929     printk("<1>(%d)␣CS␣Begin\n", current->pid);
930     printk("<1>(%d)␣Lock:␣current␣prio:␣%d\n", current->pid, current->prio);
931     printk("<1>(%d)␣Lock:␣mutex␣prio:␣%d\n", current->pid, lock->ceiling);
932 #endif
933     //rt_mutex_fastlock(lock, TASK_UNINTERRUPTIBLE, 0, rt_mutex_slowlock);
934     //immpc_mutex_lock_path(lock, TASK_UNINTERRUPTIBLE, NULL, 0);
935     immpc_mutex_fastlock(lock, TASK_UNINTERRUPTIBLE, 0, immpc_mutex_slowlock);
936 #ifdef  IPC_DEBUG
937     printk("<1>(%d)␣Lock:␣current␣prio␣(boosted):␣%d\n", current->pid, current->prio);
938 #endif
939 }
940 EXPORT_SYMBOL_GPL(immpc_mutex_lock);
941 
942 
943 /**
944  * rt_mutex_lock_interruptible - lock a rt_mutex interruptible
945  *
946  * @lock:       the rt_mutex to be locked
```

```
947   * @detect_deadlock: deadlock detection on/off
948   *
949   * Returns:
950   *  0        on success
951   * -EINTR    when interrupted by a signal
952   * -EDEADLK when the lock would deadlock (when deadlock detection is on)
953   */
954  int __sched immpc_mutex_lock_interruptible(struct immpc_mutex *lock,
955                          int detect_deadlock)
956  {
957      might_sleep();
958
959      return immpc_mutex_fastlock(lock, TASK_INTERRUPTIBLE,
960                  detect_deadlock, immpc_mutex_slowlock);
961  }
962  EXPORT_SYMBOL_GPL(immpc_mutex_lock_interruptible);
963
964  /**
965   * rt_mutex_unlock - unlock a rt_mutex
966   *
967   * @lock: the rt_mutex to be unlocked
968   */
969  void __sched immpc_mutex_unlock(struct immpc_mutex *lock)
970  {
971  #ifdef   IPC_DEBUG
972      printk("<1>(%d) unLock: current prio: %d\n",current->pid, current->prio);
973      printk("<1>(%d) unLock: mutex prio: %d\n",current->pid, lock->ceiling);
974  #endif
975
976  // immpc_mutex_unlock_path(lock);
977      immpc_mutex_fastunlock(lock, immpc_mutex_slowunlock);
978  #ifdef   IPC_DEBUG
979      printk("<1>(%d) unLock: current prio (boosted): %d\n",current->pid, current->prio);
980      printk("<1>(%d) CS End\n",current->pid);
981  #endif
982  }
983  EXPORT_SYMBOL_GPL(immpc_mutex_unlock);
984
985  /**
986   * immpc_mutex_lock_killable - lock a immpc_mutex killable
987   *
988   * @lock:           the immpc_mutex to be locked
989   * @detect_deadlock: deadlock detection on/off
990   *
991   * Returns:
992   *  0        on success
993   * -EINTR    when interrupted by a signal
994   * -EDEADLK when the lock would deadlock (when deadlock detection is on)
995   */
996  int __sched immpc_mutex_lock_killable(struct immpc_mutex *lock,
997                      int detect_deadlock)
998  {
999      might_sleep();
1000
```

```
1001        return immpc_mutex_fastlock(lock, TASK_KILLABLE,
1002                    detect_deadlock, immpc_mutex_slowlock);
1003    }
1004
1005    EXPORT_SYMBOL_GPL(immpc_mutex_lock_killable);
1006
1007    /**
1008     * immpc_mutex_timed_lock - lock a immpc_mutex interruptible
1009     *         the timeout structure is provided
1010     *         by the caller
1011     *
1012     * @lock:        the immpc_mutex to be locked
1013     * @timeout:        timeout structure or NULL (no timeout)
1014     * @detect_deadlock: deadlock detection on/off
1015     *
1016     * Returns:
1017     *  0         on success
1018     * -EINTR    when interrupted by a signal
1019     * -ETIMEDOUT  when the timeout expired
1020     * -EDEADLK when the lock would deadlock (when deadlock detection is on)
1021     */
1022    int
1023    immpc_mutex_timed_lock(struct immpc_mutex *lock, struct hrtimer_sleeper *timeout,
1024                int detect_deadlock)
1025    {
1026        might_sleep();
1027
1028        return immpc_mutex_timed_fastlock(lock, TASK_INTERRUPTIBLE, timeout,
1029                            detect_deadlock, immpc_mutex_slowlock);
1030    }
1031    EXPORT_SYMBOL_GPL(immpc_mutex_timed_lock);
1032
1033
1034    /**
1035     * immpc_mutex_trylock - try to lock a immpc_mutex
1036     *
1037     * @lock:    the immpc_mutex to be locked
1038     *
1039     * Returns 1 on success and 0 on contention
1040     */
1041    int __sched immpc_mutex_trylock(struct immpc_mutex *lock)
1042    {
1043        return immpc_mutex_fasttrylock(lock, immpc_mutex_slowtrylock);
1044    }
1045    EXPORT_SYMBOL_GPL(immpc_mutex_trylock);
1046
1047    /**
1048     * immpc_mutex_set_ceiling - change the ceiling of a mutex
1049     *
1050     * @lock:    the immpc_mutex to be adjusted
1051     * @newceiling: the new mutex ceiling
1052     *
1053     * Returns 1 on success and 0 on contention
1054     */
```

```
1055 extern void immpc_mutex_set_ceiling(struct immpc_mutex *lock, int newceiling)
1056 {
1057     //unsigned long flags;
1058
1059     //raw_spin_lock_irqsave(&lock->wait_lock, flags);
1060
1061     lock->ceiling = (MAX_RT_PRIO-1 - newceiling);
1062
1063     //raw_spin_unlock_irqrestore(&lock->wait_lock, flags);
1064 }
```