UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Daniel Bristot de Oliveira

CARACTERIZAÇÃO DA EXECUÇÃO DE TAREFAS NO LINUX PREEMPT-RT ATRAVÉS DE UMA FERRAMENTA DE TRACE

Florianópolis

2014

CARACTERIZAÇÃO DA EXECUÇÃO DE TAREFAS NO LINUX PREEMPT-RT ATRAVÉS DE UMA FERRAMENTA DE TRACE

Daniel Bristot de Oliveira

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisistos para a obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

Prof. Rômulo Silva de Oliveira, Dr.
Orientador

Prof. Jomi Fred Hübner, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia de Automação e Sistemas

Banca Examinadora:

Prof. Carlos Barros Montez, Dr.
Presidente

Prof. Rafael Rodrigues Obelheiro, Dr. - UDESC/Joinville

Prof. Rivalino Matias Júnior, Dr. - UFU/Uberlândia

Ao meu pai, Sr. Ângelo de Oliveira

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus e a comunidade qual participo, a Paróquia Santo Antônio em São José. Em especial ao Pe. Hélio da Cunha, pelo exemplo de dedicação à Deus e ao próximo.

A Intelbras, em primeiro lugar como instituição, por flexibilizar a minha jornada de trabalho e permitir meu ingresso no Mestrado. Em segundo lugar como um ambiente de amigos, agradeço a todos que me apoiaram, incentivaram e principalmente me escutaram, tanto nos dias de empolgação quanto nos dias de cansaço. A Intelbras foi fundamental para a parte técnica deste trabalho, por me dar a possibilidade de enfrentar desafios reais, em sistemas de tempo real reais e que hoje estão por ai, facilitando a comunicação de pessoas.

A minha família, principalmente por entender porque eu não estava lá em diversos momentos e por me apoiar sempre. Em especial agradeço a minha irmã Alessandra de Oliveira que sempre me escutou, apoiou, me dando seu exemplo e sua cumplicidade. Ao meu pai, Sr. Ângelo de Oliveira, por ter me feito crescer escutando que o esforço do seu trabalho era para a formação acadêmica de seus filhos e por ter sido sempre um grande profissional, 24x7.

Escolhi a ciência da computação por causa de um sistema operacional, o Linux. Minha graduação foi focada em sistemas operacionais. Minha carreira profissional foi construída com sistemas operacionais: primeiro como administrador de sistemas, por causa do FreeBSD, depois como desenvolvedor de sistemas embarcados, por causa do Linux. Aqui agradeço a comunidade de Sistemas Operacionais *Open Source*, em especial meu amigo Marcelo Araújo, por compartilhar a paixão por sistemas operacionais e por software livre.

Meu último agradecimento vai para o Prof. Rômulo. Primeiro por ter me orientado com muita paciência, frente aos meus recorrentes e teimosos erros na escrita deste trabalho. Mas de maneira mais especial, gostaria de agradecer ao Prof. por todo o conhecimento compartilhado nestes últimos anos, não só aquele conhecimento que se adquire em livros, mas aquele conhecimento que só quem realmente gosta do que faz tem, tanto de sistemas operacionais e sistemas de tempo real, quanto de vida, sempre com muito entusiasmo.

Hard never stopped us before, did it ;-)

Peter Zijlstra, na LKML, sobre a dificuldade de resolver problemas de mutex aninhados: https://lkml.org/lkml/2009/4/16/284.

RESUMO

Sistemas de tempo real são sistemas computacionais que respondem a eventos, os quais requerem um tempo limite máximo de resposta. O não cumprimento do tempo limite de resposta faz com que o sistema perca de maneira parcial ou total o seu valor. O *patch* PREEMPT-RT é o padrão *de facto* para Linux de tempo real, sendo utilizado tanto em pesquisas quanto na indústria. Apesar disto, são frequentes as discussões sobre as diferenças entre o Linux de tempo real e a teoria de sistemas de tempo real. Dentre os pontos de divergência, estão o método de análise e a métrica utilizada para avaliar o sistema. A principal métrica de análise do Linux de tempo real é a latência de escalonamento, principalmente no PREEMPT-RT. Apesar de eficiente, ao ponto de tornar o Linux capaz de atender requisitos temporais de diversas aplicações, este método é simplista se comparado com a teoria de sistemas de tempo real. Na teoria busca-se comprovar analiticamente que um conjunto de tarefas irá cumprir os seus *deadlines*, apesar das interferências e bloqueios que estas podem sofrer durante a sua execução. Este trabalho apresenta a relação entre as abstrações utilizadas no método de análise de tempo de resposta com as funções do *kernel* do Linux, no que diz respeito às funções que afetam temporalmente a execução das tarefas de tempo real. A partir desta relação, uma nova ferramenta de *trace* é apresentada. Esta ferramenta cria uma nova forma de visualizar a execução das tarefas de tempo real, que permite monitorar os eventos que afetam o comportamento temporal das tarefas, utilizando as abstrações do método de análise de tempo de resposta. A partir da utilização da ferramenta proposta foi possível, para as tarefas com um comportamento típico de tarefas de tempo real na teoria, caracterizar a execução das tarefas de tempo real no Linux e desenvolver um conjunto de equações que determinam qual a origem dos tempos de respostas.

Palavras-chave: Linux, Tempo Real, Análise de Tempo de Resposta, *Trace*, Medições.

ABSTRACT

Real-time systems are computational systems that respond to events which require a maximum response time. A failure in the attempt to satisfy the timing requirements makes system to lose partially or entirely their value. The PREEMPT-RT patch is the *de facto* standard for real-time Linux, being used for both industry and research. Despite this, there are frequent discussions about the differences between the real-time Linux and theory of real-time systems. Among the points of contention are the method of analysis and the metrics used to evaluate the system. The main metric for the analysis of the real-time Linux is the scheduling latency, mainly in the PREEMPT-RT. Despite efficient, at the point of being able to meet timing requirements of various real time applications on Linux, this method is simplistic if compared with the theory of real-time systems. The real-time theory tries to analytically prove that a set of tasks will meet their deadlines, despite the interference and locks it can suffer during its execution. This work presents the relationship between the abstractions used in the response time analysis and the functions of Linux kernel, with regard to the functions that affects the timing behavior of the real-time tasks. From this relationship, a new tool for trace is presented. This tool creates a new way to trace the real-time tasks, enabling the monitoring of the events that affect the timing behavior of tasks, using the abstractions used in the response time-analysis method. From the use of the proposed tool was possible, for tasks with a typical behavior of real-time tasks, to characterize the execution of real-time tasks on Linux and develop a set equations that determine the origin of the response time.

Keywords: Real-Time, Linux, Trace, Response Time Analysis, Measurements.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
CPU	*Central Processing Unit*
CTF	*Commom Trace Format*
FULL HD	*Full High Definition*
GCC	*GNU C Compiler*
HD	*High Definition*
IOCTL	*Input/Output Control*
IRQ	*Interrupt Request Line*
KVM	*Kernel-based Virtual Machine*
LTTng	*Linux Trace Toolkit Next Generation*
LTTV	*Linux Trace Toolkit Viewer*
MMU	*Memory Management Unit*
NMI	*Non-Maskable Interrupt*
nop	*No Operation*
NP-difícil	Não Polinomial Difícil
PID	*Process identifier*
RAM	*Random Access Memory*
RCU	*Read-copy update*
RT	*Real-Time*
SRT	Sistemas de tempo real
WCET	*Worst Case Execution Time*

LISTA DE SÍMBOLOS

r_i	*Release time*
a_i	Tempo de chegada
s_i	*Start time*
J_i	Atraso de ativação
C_i	Tempo de computação
d_i	*Deadline* absoluto
D_i	*Deadline* relativo
f_i	*Finish time*
R_i	*Response time*
L_i	*Lateness*
X_i	*Slack time*
B_i	Tempo de bloqueio
W_i	Janela de ocupação
T_i	Período da tarefa
τ_i	Tarefa periódica
di_i	Instante de tempo absoluto em que a interrupção foi desabilitada
ei_i	Instante de tempo absoluto em que a interrupção foi reabilitada
id	Ocorrência de uma interrupção em uma seção de interrupção desabilitada
rji_i	Atraso de ativação de uma interrupção
ep_i	Instante tempo absoluto em que a preempção foi reabilitada
sw_i	Instante de tempo absoluto em que a tarefa foi ativada
pd	Preempção desabilitada
pj_i	Atraso de preempção
se_i	Instante de tempo absoluto em que o escalonamento estava permitido na CPU que o processo foi escalonado
cw_i	Instante de tempo absoluto da troca de contexto
sj_i	Atraso de escalonamento
rjp_i	Atraso de ativação de um processo
bs_{ji}	Instante de tempo absoluto em que a função de exclusão mútua foi chamada, pela j-tésima vez
bf_{ji}	Instante de tempo absoluto em que a função de exclusão mútua retornou, pela j-tésima vez

ct_{ji}	Variável binária, o valor 1 quando a tarefa sofreu contenção ou 0 se a tarefa não sofreu contenção
b_{ji}	Tempo de bloqueio do j-tésimo bloqueio da ativação de uma tarefa
is_i	Instante de tempo absoluto do inicio da execução do tratador de interrupção
if_i	Instante de tempo absoluto do retorno da execução do tratador de interrupção
$iwtnmi_i$	Janela de execução para uma interrupção não mascarável
$iwtmi_i$	Janela de execução para uma interrupção mascarável
iet_i	Tempo de execução do tratador de interrupção
RI_i	Tempo de resposta de um tratador de interrupção
$procint_{ji}$	Determina se a interrupção faz parte do contexto do processo ou não
ip_{ji}	Interferência de uma interrupção em uma *thread*
II_i	Interferência de interrupções
s_i	Tempo absoluto em que a tarefa suspende
RP	Tempo de resposta de uma *thread*
$ctxin_i$	Instante de tempo absoluto da troca de contexto que inicia a execução de uma nova ativação de uma tarefa
$scdr_i$	Instante de tempo absoluto que a função de escalonamento retornou a execução da tarefa
$scin_i$	Janela de execução do escalonamento de entrada
$siet_i$	Tempo de execução do escalonamento de entrada
$scdc_i$	Instante de tempo absoluto que a função de escalonamento iniciou a execução da tarefa
$ctxout_i$	Instante de tempo absoluto da troca que contexto que a removeu a tarefa do processador no estado S
$scout_i$	Janela de execução do escalonamento de saída
$soet_i$	Tempo de execução do escalonamento de saída
$schw_i$	Janela de execução do escalonamento de entrada e saída
set_i	Tempo de execução de escalonamento de entrada e saída
e	Número de janelas de execução que ocorreram dentro da ativação de determinada tarefa
pi_{ei}	Interferência de *threads* em outras *threads*
PI_i	Tempo total de interferência de *threads*

I_i	Interferência
$ischw_i$	Janela de execução do escalonamento de interferência
$iset_i$	Tempo de execução de escalonamento de interferência
WOS_i	Janela de execução do *overhead* de escalonamento
EOS_i	Tempo de execução do *overhead* de escalonamento
pwt_i	Define-se a janela de execução da *thread*
CP_i	Tempo de execução de uma tarefa com contexto de *thread*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de tempo real são sistemas computacionais que respondem a eventos que requerem um tempo limite máximo de resposta. O não cumprimento do tempo limite de resposta faz com que o sistema perca parcial ou totalmente seu valor. Apesar de isto levar a crer que o sistema deva ser o mais rápido possível, esta é uma falsa concepção, os sistemas de tempo real são sistemas com comportamento previsível tanto em relação a sua execução lógica quanto sua execução no tempo.

Os sistemas de tempo real estão presentes em nosso cotidiano em diversas aplicações, desde sistemas críticos, como sistema de controle de voo de uma aeronave ou sistemas de controle de frenagem e tração de carros, nos quais uma falha lógica ou temporal pode causar uma catástrofe com perda de vidas humanas, a sistemas de consumo como *video-games*, sistemas de telecomunicações e servidores de aplicações financeiras, nos quais o atraso no cumprimento de um *deadline* é tolerável, porém, não desejável. A estas duas classes de sistemas, nas quais os danos de uma falha temporal são críticos ou toleráveis é dado o nome de sistemas de tempo real rígido (*hard real-time*) e sistemas de tempo real brando (*soft real-time*), respectivamente.

Apesar da teoria de sistemas de tempo real para sistemas com um único processador estar bastante consolidada e já aplicada na indústria, a teoria de sistemas com mais de um processador é ainda hoje um campo aberto e fonte de diversas pesquisas. Principalmente porque no passado os fabricantes de processadores produziam e avançavam em tecnologia e velocidade de processamento de um único processador, o que mudou nos últimos anos devido a limitações alcançadas no processo de desenvolvimento e fabricação de processadores com frequências muito altas. A indústria partiu para o desenvolvimento de processadores com múltiplas CPUs, alavancando assim as pesquisas sobre sistemas de tempo real nesta classe de processadores (DAVIS; BURNS, 2009). Isto principalmente no que diz respeito ao escalonamento de tarefas e a bloqueios, sendo a primeira uma tarefa tipicamente implementada nos sistemas operacionais.

Aliado ao crescimento das plataformas multi-processadas, nasceu também a necessidade dos sistemas operacionais se adequarem a esta tecnologia. Um dos sistemas operacionais que se adequou rapidamente aos multiprocessadores foi o Linux. O sistema operacional Linux, devido ao suporte a diversas arquiteturas de *hardware* e funcionalidade, vem sendo utilizado em diversas plataformas, desde sistemas embarcados como telefones celulares até grandes servidores como o *cluster* de servidores que controlam a bolsa de valores Nasdaq (CORBET, 2010b). Em ambos exemplos o sistema

operacional Linux precisa atender a requisições com tempos limite de resposta. No exemplo dos celulares na execução de aplicações multimídia e no sistema de controle da bolsa de valores são efetuadas centenas de milhões de operações por dia, com picos de um milhão de compras por segundo, das quais, cada operação deve ter um tempo de resposta abaixo dos $250\mu s$ (CORBET, 2010b).

Além de se tornar uma plataforma de mercado, o Linux também se tornou uma plataforma de pesquisa. Este fato se dá principalmente por ter seu código fonte aberto, por executar em diversas arquiteturas de *hardware* e, em particular, por possuir suporte a arquiteturas de múltiplos processadores bastante estável e testado (BRANDENBUG; ANDERSON, 2009). Apesar disto, existe uma grande lacuna entre a academia e as implementações de tempo real do Linux (GLEIXNER, 2010). Para os pesquisadores, apesar do Linux apresentar características comuns no estudo de tempo real, como escalonador de prioridade fixa e suporte à herança de prioridade, estas funcionalidades representam o estado da arte de 20 anos atrás (BRANDENBUG; ANDERSON, 2009). Por outro lado, para os desenvolvedores do Linux, os cenários artificiais utilizados nas pesquisas, apesar de válidos para pesquisas, dificilmente são mapeados para problemas encontrados no uso do Linux em sistemas de tempo real (GLEIXNER, 2010).

Estas divergências aparecem logo nos primeiros comparativos entre a forma de ver um sistema de tempo real pela academia e pelos usuários e desenvolvedores do Linux. Um dos pontos de diferença é que o *patch* de tempo real do Linux tem como maior objetivo diminuir as latências do sistema, diminuindo assim o tempo de resposta das tarefas de maior prioridade, enquanto na teoria de sistemas de tempo real busca-se analiticamente comprovar que um conjunto de tarefas irá executar antes dos seus *deadlines*, apesar das interferências e bloqueios que estas podem sofrer durante a sua execução.

Esta diferença influencia também em como a execução do sistema é observada e analisada. No Linux, a maneira mais comum de observar a execução do sistema, para análise de tempo real, é a partir do *trace* da latência de execução de funções ou atividades no *kernel*, como o tempo que uma função demorou para executar, ou quanto tempo demorou para o sistema fazer uma troca de contexto. Já na análise de tempo real na academia, o sistema é observado como um gráfico no qual é possível ver como as variáveis analíticas, como bloqueio e interferência, influenciam na execução das tarefas, e se o sistema cumpriu ou não as suas restrições temporais.

Por causa das diferenças entre as suposições feitas nos sistemas de tempo real teóricos e na execução do Linux, não é possível hoje fazer um estudo analítico da execução das tarefas de tempo real do Linux. Porém, acredita-se que o fato de poder acompanhar a execução do Linux, levando em

conta os parâmetros usados na análise teórica de sistemas, pode fazer com que seja mais fácil de entender a execução do Linux em tempo real. Isto tanto pelos acadêmicos, já os familiarizando com os métodos do Linux e o modelo de execução no Linux, quanto para os desenvolvedores, que podem ter uma visão teórica de tempo real da execução do Linux. Assim, estes dois mundos são aproximados.

1.1 OBJETIVOS

O objetivo deste trabalho é relacionar as abstrações utilizadas no método de análise de tempo de resposta com as funções do *kernel* do Linux, no que diz respeito às funções que afetam temporalmente a execução das tarefas de tempo real no Linux. A partir desta relação, deseja-se adaptar uma ferramenta de *trace* existente a fim de criar uma nova forma de *trace* que permita monitorar os eventos que afetam o comportamento temporal das tarefas de tempo real no *kernel* do Linux com o *patch* PREEMPT-RT, em termos de escalonamentos, bloqueios e interferências.

A partir do uso desta ferramenta, deseja-se melhorar o entendimento dos mecanismos internos do *kernel* e como eles impactam o tempo de resposta das tarefas. Com isto acredita-se ser possível definir algumas equações que descrevem algebricamente como o tempo de resposta das tarefas é formado. Entretanto, devido a complexidade e flexibilidade do Linux, não é objetivo deste trabalho adaptar o Linux para o modelo de tempo de resposta, nem definir um novo método de análise de tempo de resposta para o Linux.

Acredita-se que esta nova forma de observar a execução do Linux possa servir de base para trabalhos futuros que desenvolvam métodos de análise de tempo de resposta para o Linux com o *patch* PREEMPT-RT.

1.2 ORGANIZAÇÃO

Para alcançar os objetivos, o trabalho foi dividido em cinco etapas, sendo estas:

- Sistemas de tempo real: apresenta um levantamento teórico sobre tempo real, análise de tempo de resposta de tarefas e Linux e suas variantes de tempo real;
- Pontos de interesse no *kernel* do Linux: apresenta um estudo sobre os mecanismos de exclusão mútua e métodos que interferem na execução temporal do *kernel* do Linux;

- Ferramentas de *trace*: estudo das ferramentas de *trace* existentes no Linux e a sua capacidade de obter dados sobre os pontos de interesse;
- Adaptação de uma ferramenta: a customização de uma ferramenta de *trace* para obter os pontos de interesse presentes no *kernel* do Linux;
- Resultados: apresenta os resultados obtidos com a adaptação da ferramenta para visualizar a interferência temporal na execução do *kernel* do Linux.

Cada uma destas etapas se tornou um capítulo desta dissertação, sendo estas apresentadas nesta ordem nos próximos capítulos.

2 SISTEMAS DE TEMPO REAL

Os sistemas de tempo real são sistemas computacionais que respondem a eventos que possuem um tempo limite máximo de resposta que, caso superado, faz com que o sistema perca parcial ou totalmente seu valor. Como consequência, a corretude desta classe de sistemas não se dá apenas no fato da corretude lógica, mas também com corretude temporal (BUTTAZZO, 2005).

Os sistemas de tempo real são sistemas computacionais ou de comunicação que executam trabalhos, ou *jobs*. O conjunto de trabalhos correlacionados é chamado tarefa, ou *task*. Os *jobs*[1] executam e são escalonados em um recurso, este recurso pode ser um disco, meio de comunicação, processador, etc. (LIU, 2000). Neste trabalho o termo recurso está ligado a processador.

Cada *job* possui um conjunto de parâmetros temporais que são utilizados para a análise de escalonamento. A Figura 1 mostra algumas dessas variáveis, as quais são descritas a seguir (BUTTAZZO, 2005) (LIU, 2000).

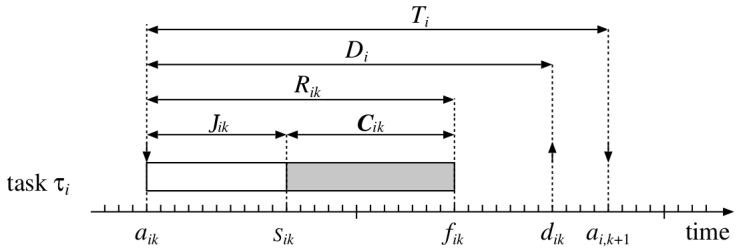

Figura 1 – Parâmetros de tarefa de tempo real

O tempo de chegada a_i é o tempo em que a tarefa torna-se pronta para a execução. Este tempo também é conhecido como *release time*, indicado por r_i.

O *start time* s_i é o tempo em que o *job* inicia a sua execução.

A diferença de tempo entre o tempo de chegada a_i e o *start time* s_i é denominado *release jitter*, denotado pela variável J_i. O *release jitter* ocorre quando um *job* torna-se pronta para a execução, porém, tem seu inicio atrasado. Um exemplo de *release jitter* ocorre quando o timer do sistema sinaliza que um *job* periódico deve iniciar sua execução, porém, a entrega da interrupção do timer é atrasada por o sistema estar com as interrupções desabilitadas.

O tempo de execução C_i ou *execution time* de um *job* é o tempo necessário para um *job* completar sua execução quando este executa sem inter-

[1] Para evitar ambiguidades por erros de tradução, serão utilizados alguns termos em inglês.

rupções e com todos os recursos necessários. O tempo de execução de um *job* pode variar por diversos motivos. Por exemplo, dependendo dos dados de entrada, o *job* poderá executar desvios que irão alterar o tempo de execução, e se o sistema subjacente possui mecanismos como memória *cache* e *pipeline* os tempos de execução podem variar dependendo do estado da memória *cache* e do *branch prediction*. A maioria das análises leva em consideração o pior tempo de execução, geralmente referenciado como *WCET*, da abreviação de *worst case execution time*.

O *deadline* ou *deadline* absoluto, identificado por d_i, de um *job* é o instante de tempo máximo no qual o *job* deve ter terminado a sua execução. Na maioria dos casos é mais natural utilizar o tempo de resposta de um *job* como o requisito temporal a ser cumprido pelo sistema. O tempo de resposta é a diferença de tempo entre a chegada de um *job* e o seu término, onde o tempo máximo de resposta permitido de um *job* é o seu *deadline* relativo, identificado por D_i, assim $D_i = d_i - a_i$.

Finish time f_i é o tempo em que o *job* termina a sua execução e o tempo de resposta ou *response time* R_i é a diferença entre o *finish time* e o tempo de chegada: $R_i = f_i - a_i$.

Lateness L_i, $L_i = f_i - d_i$, representa o atraso de término de um *job* em relação ao seu *deadline*. Logo, se uma tarefa terminar antes do seu *deadline* o seu L_i é negativo e *tardiness* E_i ou *exceeding time* é o tempo que um *job* executa após o seu *deadline* $E_i = max(0, L_i)$.

Laxity ou *slack time* X_i é o tempo máximo que uma tarefa pode ter sua ativação atrasada e mesmo assim terminar antes do seu *deadline*: $X_i = d_i - a_i - C_i$.

Valor ou *value* v_i representa a importância de uma tarefa em relação as outras tarefas do sistema.

As tarefas podem ser classificadas também pela constância com que os *jobs* são ativados, em particular uma tarefa pode ser periódica ou aperiódica. As tarefas periódicas são identificadas por ter a ativação de seus *jobs* como uma sequencia infinita, idêntica, regular e constante, e geralmente são denotadas por τ_i. O tempo de ativação da primeira instância de uma tarefa periódica é denominada fase e denotada por ϕ_i. O tempo da k-ésima ativação de uma tarefa periódica é dada por $\phi_i + (k-1)T_i$ onde T_i é o período da tarefa. Na prática as tarefas periódicas são caracterizadas pelo seu *deadline* D_i (muitas vezes igual ao período T_i) e tempo de computação C_i. As tarefas aperiódicas também consistem de sequencia infinita de chamadas do mesmo *job*, porém sua ativação não é regular e podem acontecer a qualquer momento, inclusive ao mesmo tempo. Um caso específico de tarefas aperiódicas, quando existe um tempo mínimo entre as ativações de dois *jobs*, neste caso as tarefas são chamadas de esporádicas.

Estes parâmetros são utilizados em diversas fórmulas que ajudam a demonstrar, por exemplo, que um sistema é escalonável ou qual o tempo máximo de resposta de uma tarefa em determinado algoritmo de escalonamento.

A definição de *deadline* ajuda a definir o modelo de sistemas de tempo real rígido e brando. Se a perda de um *deadline* causa uma consequência catastrófica no sistema este se encaixa na categoria *hard* e cada instância deve executar com garantia de que irá cumprir o *deadline* mesmo no pior caso. Já uma tarefa é dita *soft* se a perda de um *deadline* prejudica o desempenho do sistema, porém, não põe o sistema em risco.

2.1 ESCALONAMENTO EM TEMPO REAL

Em geral, para definir um problema de escalonamento é necessário especificar três conjuntos: O conjunto de n tarefas $\tau = \tau_1, \tau_2, ..., \tau_n$, um conjunto de m processadores $\rho = \rho_1, \rho_2, ..., \rho_m$ e um conjunto de q recursos $\sigma = \sigma_1, \sigma_2, ..., \sigma_q$. Neste contexto, a análise se dá na capacidade de atribuir tempo dos processadores de ρ e recursos de σ para tarefas em τ, de modo a completar todas as tarefas em τ dentro das restrições propostas (BUTTAZZO, 2005).

Segundo (BUTTAZZO, 2005) os algoritmos de escalonamento podem ser classificados como:

Preemptivo: quando uma tarefa pode ser interrompida a qualquer momento, de acordo com uma política de escalonamento;

Não preemptivo: quando uma tarefa, após iniciar sua execução é executada até o seu termino, as decisões de escalonamento são tomadas apenas no fim de cada execução;

Cooperativo: As tarefas podem sofrer preempção em apenas alguns pontos de sua execução, efetivamente a execução das tarefas consiste de uma série de sessões não preemptivas (BUTTAZZO, 2005);

Estático: são aqueles nos quais as decisões são baseadas em parâmetros fixos, definidos antes das suas ativações;

Dinâmico: quando as decisões de escalonamento são baseadas em parâmetros dinâmicos que podem mudar durante a execução do sistema;

Off-line: quando a execução completa do conjunto de tarefas é determinada antes da execução das tarefas, o escalonamento é gerado desta maneira, armazenado em uma tabela e depois executado por um despachante;

On-line: quando as decisões de escalonamento são tomadas em tempo de execução, por exemplo quando uma tarefa entra no sistema, ou quando uma tarefa termina a sua execução;

Ótimo: Um algoritmo é dito ótimo se este minimiza alguma função de custo do sistema, ou, quando não existe uma função de custo e a única meta é determinar se o escalonamento é possível, neste caso um algoritmo é dito ótimo se ele sempre encontra uma forma possível de escalonar o conjunto de tarefas para o qual exista uma forma de escalona-lo;

Heurístico: Um algoritmo é dito como heurístico se este encontra, utilizando um função heurística, um escalonamento possível, porém, este não garante ser o escalonamento ótimo, mesmo que exista um;

Clarividente: um algoritmo é dito ser clarividente se este consegue prever o futuro, isto é, se este conhece antecipadamente todos os parâmetros de cada tarefa. Apesar deste algoritmo não existir, este é utilizado para comparar o desempenho de algoritmos reais;

Existem diversas abordagens para o escalonamento de tarefas em tempo real. Como exemplo, pode-se citar três: *clock-driven*, *weight round-robin* e *priority driven* (LIU, 2000).

Nos algoritmos *clock-driven* as decisões de escalonamento de quando cada tarefa executa são feitas *a priori*, antes da execução do sistema. Tipicamente, todos os parâmetros das tarefas de tempo real rígido possuem seus parâmetros fixos e conhecidos. O escalonamento dos *jobs* é computado *off-line*, armazenado e executado por um despachante. Como vantagem tem-se a garantia do cumprimento dos *deadlines*, como desvantagem tem-se na perda de flexibilidade do sistema (LIU, 2000).

A abordagem *round-robin* é usada principalmente para sistemas de tempo compartilhado. Nesta abordagem as tarefas prontas para a execução são enfileiradas e para cada tarefa é dada permissão para execução de uma fatia (*slice*) de tempo. Se o *job* não termina a sua execução no *slice* cedido a ele, este sofre preempção e retorna a execução no próximo *slice*. Nesta abordagem cada tarefa irá obter $1/n$ de tempo de CPU, sendo n o número de tarefas ativas no sistema a cada rodada. A abordagem *weight round-robin* é semelhante ao *round-robin*, porém, o *slice* de tempo cedido a cada tarefa não é igual. Cada tarefa recebe um peso *wt* e executa *wt slices* por rodada, o tempo de cada round é a soma dos pesos de cada tarefa vezes o tempo de um *slice*. Com este controle é possível adiantar ou atrasar o término de uma tarefa (LIU, 2000).

Os algoritmos dirigidos por prioridade (*priority driven*) se referem a uma grande classe de algoritmos. Nesta abordagem os *jobs* são organizados em filas de prioridade. A cada ponto de decisão do escalonador este seleciona o *job* de maior prioridade para executar no processador disponível. Desta maneira, os recursos ficam livres apenas quando não há *jobs* nas filas de execução. A esta característica é dado o nome de *work-conserving* (LIU, 2000).

Com relação a priorização das tarefas e *jobs*, os algoritmos de escalonamento podem ser classificados em 3 categorias: prioridade fixa por tarefa, fixa por *job* e dinâmica. Nos algoritmos de prioridade fixa por tarefa cada tarefa possui uma prioridade fixa para todos as ativações de seus *jobs*, como exemplos de algoritmos temos o *Rate Monitonic* e *Deadline Monotonic*. Nos de prioridade fixa por *job* cada *job* de uma tarefa pode ter uma prioridade diferente, porém, após atribuída a prioridade do *job* esta não pode ser alterada. Como exemplo temos o algoritmo EDF (*Early Deadline First*). Por fim, nos algoritmos de prioridade dinâmica cada *job* pode ter sua prioridade alterada a qualquer momento, como exemplo tem-se o algoritmo LLF (*Least Laxity First*) (DAVIS; BURNS, 2009).

Dentro destas características, este trabalho pretende estudar os algoritmos preemptivos, dinâmicos, *on-line* e baseados em prioridade fixa por tarefa.

2.2 ANÁLISE DO TEMPO DE RESPOSTA

Para garantir a corretude temporal na execução de tarefas de tempo real, é necessário saber se dado um conjunto de tarefa e um modelo de execução, todas tarefas executam sem exceder o seu *deadline*. Existem diversos modos analíticos para se obter esta resposta, dependendo do modelo de execução do sistema. O Linux apresenta o modelo de prioridade fixa, com controle de inversão de prioridade. Para este modelo é possível saber se o sistema é escalonável utilizando o método de análise de tempo de resposta.

O tempo máximo de resposta é o tempo que uma tarefa demora para terminar a sua execução no instante de tempo crítico do sistema, considerando todos os bloqueios e interferências que a tarefa pode receber de outras tarefas do sistema. O tempo de resposta de uma tarefa τ_i é representado pela variável R_i, que é obtido com a Equação 2.1.

$$\mathrm{R}_i = C_i + \sum_{j \in hp(i)} I_j \tag{2.1}$$

onde $hp(i)$ é o conjunto de prioridades maiores que i e I é a interferência que a tarefa τ_i pode receber das tarefas τ_j no tempo R_i. A interferência I_j é calculada com a seguinte equação:

$$\mathrm{I}_j = \left\lceil \frac{R_i}{P_j} \right\rceil .C_j \tag{2.2}$$

Onde, o número de execuções da tarefa τ_j no tempo R_i é determinado por $\lceil R_i/P_j \rceil$. Juntando as duas, temos a seguinte equação:

$$\mathrm{R}_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil .C_j \tag{2.3}$$

Um sistema é dito escalonável se, para toda tarefa τ_i, R_i é menor que D_i. Algo a se notar é que o R_i aparece nos dois lados da equação, o que implica na necessidade de utilizar um método iterativo para determinar R_i. Inicialmente o valor de R_i é C_i, então é utilizada a Equação 2.4 de modo iterativo, até que $R_i^{n+1} = R_i^n$ ou $R_i^{n+1} > D_i^n$.

$$\mathrm{R}_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil .C_j \tag{2.4}$$

Este modelo pode ser estendido para aceitar que haja um atraso entre a chegada de uma tarefa e o início de sua execução, a este atraso no início da execução de uma tarefa é dado o nome de *release jitter*, denotado pela variável J [2].

Para poder adicionar o *release jitter* na fórmula é necessário alterar a expressão, adicionando o conceito de período ocupado (*busy period*). Um período ocupado i corresponde a uma janela de tempo W_i onde ocorre a execução contínua de tarefas de prioridade maior ou igual a i. O cálculo do período ocupado é semelhante ao do tempo de resposta, onde se substitui o R_i por W_i, e o J_j é adicionado à equação, resultando na seguinte equação:

$$\mathrm{W}_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil .C_j$$

[2] Em algumas referências J representa as tarefas aperiódicas. Neste texto as tarefas são representadas apenas por τ. J representa o *release jitter*.

(2.5)

A solução para a recorrência de W_i é resolvida de maneira iterativa, análoga à resolução de R_i. Para determinar o tempo de resposta R_i é necessário considerar além do período ocupado W_i da i-ésima tarefa, também o *release jitter* J_i:

$$\mathrm{R}_i = W_i + J_i \tag{2.6}$$

Ainda é possível aprimorar o modelo adicionando a possibilidade de as tarefas possuírem mecanismos de sincronização e exclusão mútua, o que é comum em sistemas multi tarefa. A necessidade de exclusão mútua cria a possibilidade de inversão de prioridade. A inversão de prioridade acontece sempre que uma tarefa de maior prioridade é bloqueada por uma tarefa de menor prioridade que retem algum mecanismo de exclusão mútua. A inversão de prioridade é inerente à necessidade de exclusão mútua, porém, é desejável que o tempo de inversão de prioridade seja de alguma forma limitado. Para que a inversão de prioridade seja limitada foram introduzidos protocolos de exclusão mútua que limitam de maneira determinista o tempo que uma tarefa de maior prioridade é bloqueada por uma de menor prioridade. Como exemplo destes protocolos temos o Protocolo de Herança de Prioridade (*Priority Inheritance Protocol*) e o Protocolo de Prioridade Teto (*Priority Ceiling Protocol*) (BUTTAZZO, 2005).

Estes protocolos possuem métodos analíticos para determinar o maior tempo de bloqueio B_i que uma tarefa τ_i pode sofrer. Com isto é possível determinar o tempo de resposta de uma tarefa levando em consideração os bloqueios recebidos. A Equação 2.7 estende o modelo da Equação 2.5.

$$\mathrm{W}_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil . C_j \tag{2.7}$$

Com isto, é possível determinar o tempo de resposta levando em conta:

- C: O tempo de computação da tarefa;
- P: O período de execução da tarefa;
- B: O tempo de bloqueio que a tarefa pode sofrer;
- $W_{hp(i)}$: A interferência das tarefas de maior prioridade;

- J: O atraso de ativação que uma tarefa pode receber.

2.3 LINUX EM TEMPO REAL

Nos últimos anos o sistema operacional Linux vem se tornando uma plataforma de pesquisa bastante utilizada, principalmente por ter seu código fonte aberto e executar em diversas arquiteturas de *hardware*. Em particular, o Linux vem sendo utilizado como ponto de partida em diversas pesquisas com sistemas de tempo real. Outro fator importante é que o Linux possui suporte a arquiteturas de múltiplos processadores bastante estável e testado e por isto continuará sendo alvo de pesquisas em um futuro próximo (BRANDENBUG; ANDERSON, 2009).

As variantes de tempo real do Linux podem ser caracterizadas em duas classes: A classe *nativa*, como o PREEMPT_RT (PREEMPT_RT, 2013) e o Litmus-RT (LITMUSRT, 2010), em que o *kernel* do Linux é o único *kernel* presente e é responsável por garantir os requisitos de tempo real, e a classe *para-virtualizada*, em que um *microkernel* de tempo real rígido executa em paralelo com o *kernel* do Linux e é responsável por executar as tarefas de tempo real, como exemplo tem-se o RTAI (RTAI, 2013) e o Xenomai (XENOMAI, 2013). Apesar da segunda abordagem ser capaz de atender requisitos de tempo real rígido, as tarefas que executam no *microkernel* não conseguem acessar os serviços do Linux em tempo real. Assim, esta abordagem não dá ao Linux a capacidade de executar em tempo real, apenas de um *microkernel* de tempo real co-existir com o kernel do Linux no mesmo *hardware* (BRANDENBUG; ANDERSON, 2009).

Apesar de incluir diversas pesquisas e implementações clássicas de tempo real, como algoritmos de escalonamento de prioridade fixa e controle de inversão de prioridade em exclusão mutua através do algoritmo de herança de prioridade, é possível classificar o Linux apenas como um sistema de tempo real brando. Isto ocorre pela existência de muitas sessões não preemptivas excessivamente longas (BRANDENBUG; ANDERSON, 2009). O *kernel* não tem noção de tempo no escalonamento de tarefas, os escalonadores utilizam o conceito de fatia de tempo que atribui à tarefa um tempo, atualizado dinamicamente, para execução e o sistema não utiliza parâmetros comuns em tempo real para fazer a distribuição das tarefas nos múltiplos processadores de um sistema multiprocessado (FAGGIOLI et al., 2009).

Além das características do Linux, devido ao indeterminismo adicionado pela arquitetura de *hardware*, como os vários níveis de memória *cache* (BUTTAZZO, 2005) e pela falta de maturidade das ferramentas de análise de tempo para determinar o WCET de um tarefa em sistemas multiproces-

sados, tanto a configuração do sistema como a garantia de que os *deadlines* serão cumpridos sempre é uma tarefa bastante difícil, mesmo em sistema com baixa carga. Por outro lado, os tempos de execução médio das tarefas podem ser estimados de uma maneira imparcial a partir de dados observados (ANDERSON; MILLS, 2010), propiciando assim uma abordagem de tempo real brando, onde busca-se maximizar uma métrica de escalonamento ou limitar o *tardness* das tarefas.

2.4 STR EM SISTEMAS MULTIPROCESSADOS

As pesquisas em sistemas de tempo real iniciaram no fim dos anos 60 e início dos anos 70, com um grande esforço nas décadas de 80 e 90. A maior parte deste esforço foi colocado em pesquisas onde várias tarefas compartilhavam apenas um processador. Atualmente, apesar de ainda haver campos de pesquisa para sistemas monoprocessados, esta tecnologia já se encontra em um estado maduro, tendo gerado diversos livros e sido adotada pela indústria com sucesso (DAVIS; BURNS, 2009).

A pesquisa em sistemas multiprocessados teve início na mesma época que as pesquisas em sistemas monoprocessados. Porém, um artigo de 1969 já relatava que o escalonamento em sistemas multiprocessados adicionava um grande grau de complexidade, e poucos resultados obtidos nas pesquisas em sistemas monoprocessados poderiam ser generalizados diretamente aos sistemas multiprocessados. O simples fato de uma tarefa poder executar em apenas um processador, mesmo havendo outros processadores livres, adiciona muita dificuldade para o escalonamento em sistemas multiprocessados (DAVIS; BURNS, 2009).

Em relação a teoria de sistemas monoprocessados, o escalonamento em sistemas multiprocessados adiciona diversas anomalias. As anomalias em escalonamento ocorrem quando a mudança nos parâmetros do conjunto de tarefas resultam em um efeito contra-intuitivo na escalonabilidade do sistema. Por exemplo, aumentar o período ou diminuir o WCET de tarefas enquanto todos os outros parâmetros são mantidos, no modelo de sistemas de tempo real monoprocessados resulta em uma queda na utilização do processador e é esperado que isto aumente a escalonabilidade do sistema. Porém, em sistemas multiprocessados esta alteração pode fazer com que um sistema escalonável se torne não escalonável em alguns algoritmos. Outra anomalia conhecida é a do instante crítico, onde em sistemas monoprocessados o instante crítico é dado quando todas as tarefas possuem o mesmo *release time*. Porém no escalonamento global com prioridade fixa uma tarefa não necessariamente possui o pior tempo de resposta quando todas as tarefas são liberadas no mesmo

instante (DAVIS; BURNS, 2009).

Na seção anterior foi apresentado o conceito de prioridades de tarefas em escalonamento. Em adição a isto, os algoritmos de escalonamento em sistemas multiprocessados podem ser classificados em relação a alocação e migração de tarefas em processadores. Existem três classes de algoritmos (DAVIS; BURNS, 2009):

Sem migração: As tarefas são atribuídas aos processadores e executam somente neste processador.

Migração em nível de tarefas: Os *jobs* de uma tarefa podem ser executados em diferentes processadores, porém os *jobs* não podem migrar de processador.

Migração em nível de jobs: Os *jobs* podem migrar de processador a qualquer momento, porém os jobs não podem executar paralelamente.

Nos algoritmos onde não é possível migrar as tarefas de um processador, as tarefas são atribuídas a um processador, e as tarefas de cada processador são escalonadas como em um sistema monoprocessado. A esta classe de algoritmos é dado o nome de escalonamento particionado. Nesta classe de sistemas o problema não está em como escalonar as tarefas, mas sim em como atribuir as tarefas a cada processador, sendo este um problema clássico de *bin-packing* que possui complexidade computacional NP-Difícil (DAVIS; BURNS, 2009).

Já os algoritmos onde são permitidas migrações de *jobs* ou tarefas são denominados algoritmos globais, onde tem-se uma única fila de execução e as tarefas são atribuídas aos processadores de maneira dinâmica (DAVIS; BURNS, 2009).

Existe também uma classe de algoritmos onde algumas tarefas possuem sua execução fixada em um único processador, geralmente as tarefas de maior prioridade, a as demais tarefas são escalonadas de maneira global. A esta classe de algoritmos é dado o nome de algoritmos semi-particionados (BASTONI et al., 2011).

A próxima seção descreve algumas características de hardware de sistemas multiprocessados que são relevantes ao escalonamento em tempo real.

2.5 HARDWARE DE SISTEMAS MULTIPROCESSADOS

Com respeito a capacidade de processamento dos seus processadores, os sistemas multiprocessados podem ser classificados em três categorias (DAVIS; BURNS, 2009):

Heterogêneo: Os processadores tem capacidade de processamento diferente. Uma tarefa pode ter tempo de computação diferente, dependendo da

capacidade de cada processador. Também, uma tarefa pode conseguir executar em apenas determinado processador do sistema, devido as suas capacidades.

Homogêneo: Os processadores são idênticos, assim, o tempo de execução de cada tarefa é o mesmo em todos os processadores.

Uniforme: As tarefas podem executar em qualquer processador, porém o tempo de execução de cada tarefa depende da velocidade de cada processador. Um processador com velocidade 2 irá executar uma tarefa na metade do tempo que um processador de velocidade 1 irá executar a mesma tarefa.

Além da capacidade de processamento de processador e a relação entre estes, os sistemas multiprocessados podem ser classificados com respeito a forma com que a memória é compartilhada entre os múltiplos processadores:

Sistemas distribuídos: cada processador possui memória local e compartilha em rede com outros computadores. Onde a memória local possui tempo de acesso diferente da memória compartilhada.

Sistemas de memoria compartilhada e acesso uniforme: todos os processadores possuem acesso direto à memória via um barramento, sendo o tempo de acesso à memória o mesmo em cada processador.

Sistemas de memória compartilhada e acesso não uniforme: Todos os processadores tem acesso à memória via um barramento de memória, porém, o acesso a determinadas áreas de memória pode ter tempo de acesso diferente (BRANDENBURG, 2011).

Dentre as características apresentadas, este trabalho irá se basear em arquiteturas de memória compartilhada de acesso uniforme e de processadores homogêneos.

3 PONTOS DE INTERESSE NO KERNEL DO LINUX

Por ser um sistema operacional capaz de executar em sistemas multiprocessados, o Linux apresenta uma completa API para garantir o sincronismo de operações, protegendo o acesso a dados de operações paralelas, que poderiam causar erros de execução no sistema. Isto é feito garantindo, por exemplo, a exclusividade de acesso a um conjunto de dados, ou não permitindo que uma tarefa seja interrompida por uma outra tarefa, tanto por preempção ou pela ocorrência de uma interrupção. Com o objetivo de apresentar essas interferências no fluxo de execução das tarefas no Linux, este capítulo faz o levantamento das formas de garantir acesso exclusivo, e com isto gerar bloqueios e interferências na execução das tarefas.

3.1 CONTEXTOS DE EXECUÇÃO

Antes de iniciar o estudo dos métodos de exclusão mútua, é preciso entender os contextos de execução do *kernel*, pois estes contextos apresentam restrições no modo de execução das rotinas do *kernel*, refletindo nas premissas e utilização dos métodos de exclusão mútua.

Quando um processo executa uma chamada do sistema ou aciona um tratador de exceção, por exemplo por uma falta na memória virtual, o processo ativa rotinas do *kernel*. Neste ponto o *kernel* está executando em nome do processo, ou, no contexto do processo. Quando o *kernel* está no contexto de um processo, as suas funções podem bloquear, colocando o processo para dormir chamando as rotinas de escalonamento. Quando o *kernel* termina de executar em nome do processo, o controle é retornado ao processo em espaço do usuário. Vale ressaltar que um processo só consegue executar rotinas do *kernel* utilizando duas interfaces: via chamadas do sistema ou via interrupção, não existe outro método para um processo ativar rotinas no *kernel* (LOVE, 2010).

Além do contexto do processo existe o contexto de interrupção. Quando o *kernel* está executando um tratador de interrupção o *kernel* está no contexto de interrupção. O contexto de interrupção não está associado a um processo, não podendo ser escalonado, logo, não é possível bloquear no contexto de interrupção.

Além das *threads* e processos que executam em espaço do usuário, existem *threads* que executam no espaço de memória do *kernel*. Assim como as *threads* em espaço do usuário, as *threads* no espaço do *kernel* são escalonáveis, podendo então utilizar métodos de exclusão mútua que bloqueiam.

Um exemplo de rotina que executa em uma *thread* do *kernel* são as *workqueues*, utilizada no desenvolvimento de *drivers*.

Para diminuir o tempo de execução dos tratadores de interrupção, algumas tarefas do tratador são postergadas, utilizando *softirqs* ou *workqueues*. A decisão de postergar para uma *softirq* ou *workqueue* vem da possibilidade da rotina dormir ou não. Quando não é necessário dormir, utiliza-se uma *softirq*, por exemplo, por meio das *tasklets*. Quando há a necessidade de dormir, utiliza-se as *workqueues*.

Por não dormir, as rotinas no contexto de interrupção executam em um contexto não preemptivo. Um efeito negativo disto é que o sistema acaba por apresentar maiores latências. Para diminuir o número de pontos não preemptivos do sistema, e assim diminuir as latências, no PREEMPT_RT as *softirqs* e os códigos do tratadores de interrupção são executados em *threads* do kernel, assim, mudando o contexto de execução e permitindo que estes códigos sejam interrompidos e durmam.

3.2 MÉTODOS DE EXCLUSÃO MÚTUA DO LINUX

Existem diversos métodos de exclusão mútua no *kernel* do Linux, isto porque existem restrições em como o sistema pode executar em seus diferentes contextos. Por exemplo, não é permitido que um trecho de código suspenda (seu estado no escalonador passe para dormindo) quando executando um tratador de interrupção, assim, exigindo uma abordagem de espera ocupada, em outras partes do sistema é possível que a tarefa suspenda. Em alguns casos, o dado protegido pela exclusão mútua é muito concorrido por leitura, porém, dificilmente para escrita. Por causa destas diversas premissas, foram desenvolvidos diversos métodos, buscando otimizar algumas métricas de desempenho e determinismo. A seguir são descritos os métodos de exclusão mútua do *kernel* do Linux, sua API e detalhes de sua implementação e uso.

3.2.1 Spinlock

Os *spinlocks* são o modo mais comum de exclusão mútua do *kernel*. Em uma sessão protegida por um *spinlock*, somente uma tarefa terá acesso permitido à região crítica. Se uma tarefa tenta adquirir um *spinlock* que não está retido por algum outra tarefa, o *lock* é adquirido. Se uma tarefa tenta adquirir um *spinlock* e este já foi adquirido por outra tarefa, esta ficará em espera ocupada tentando adquirir o *spinlock*, desta forma, consumindo a CPU

até que a outra tarefa libere o *spinlock* e esta possa continuar. Quando uma tarefa tenta adquirir um *spinlock* e fica nesta espera ocupada esta é dita estar em contenção (*contended*) (LOVE, 2010).

Apesar da espera ocupada pelo *lock* consumir tempo de CPU em vão, isto evita um controle mais complexo, como colocar a tarefa em modo de espera, chamar rotinas do escalonador, trocar o contexto para outra tarefa e, quando o *lock* estiver disponível, trocar o contexto para o da tarefa que espera o *spinlock*. Assim, esta característica de espera ocupada do *spinlock* é benéfica quando se tem sessões críticas pequenas (LOVE, 2010). Os *spinlocks* são utilizados principalmente em regiões do *kernel* onde uma tarefa não pode dormir, como nos tratadores de interrupção.

3.2.1.1 API

Um *spinlock* pode ser definido de duas maneiras, de maneira estática:

```
DEFINE_SPINLOCK(mr_lock);
```

ou de maneira dinâmica:

```
spinlock_t mr_lock;
spin_lock_init(&mr_lock);
```

Para usar um *spinlock* para proteger uma sessão *critica*, é preciso adquirir o *spinlock*, executar a sessão critica e liberar o *spinlock*. Para isto, utiliza-se as funções *spin_lock* e *spin_unlock*, respectivamente, como no exemplo abaixo:

```
spin_lock(&mr_lock);
[...] // sessão critica
spin_unlock(&mr_lock);
```

Além das funções de *lock* e *unlock*, existem duas funções auxiliares. A função *spin_trylock()* que tenta adquirir um *lock*, retornando 0 se não conseguiu adquirir, ou outro valor caso tenha conseguido, e a função *spin_is_locked()* que retorna verdadeiro ou falso caso o *lock* esteja ou não sendo utilizado, respectivamente.

3.2.1.2 *Deadlock* com *spinlocks*

Um fato importante é que os *spinlocks* não são recursivos, isto é, uma tarefa que tente adquirir um *spinlock* que já possui irá ficar em espera ocupada para sempre, causando um *deadlock*.

Um outro possível modo de se ter um *deadlock* com *spinlocks* é quando um tratador de interrupção e uma *thread* do *kernel* compartilham um *lock*. Suponha que uma tarefa adquire um *spinlock*, após adquirir, uma interrupção acontece no mesmo processador. Ao tentar adquirir o *spinlock* já retido pela outra tarefa, o tratador ficará executando para sempre, retendo a CPU e nunca deixando a tarefa liberar o *spinlock*. Por isto, em uma sessão critica compartilhada com um tratador de interrupção, as IRQs devem estar desabilitadas neste processador antes de adquirir um *spinlock*. Para facilitar o desenvolvimento, foi adicionada à API dos *spinlocks* dois meios de adquirir os *locks* com o correto tratamento de IRQs. O primeiro modo é utilizando as funções *spin_lock_irq* e *spin_unlock_irq*, como no exemplo a seguir:

```
spin_lock_irq(&mr_lock);
[...] // sessão critica
spin_unlock_irq(&mr_lock);
```

Neste caso, as IRQs serão desabilitadas no processador corrente antes de adquirir o *spinlock*, e habilitadas após liberar o *spinlock*, incondicionalmente. Estas chamadas só devem ser utilizadas em casos onde sabe-se que as interrupções do processador corrente não estão desabilitadas. Caso estejam, ao reabilitar as interrupções, o sistema irá gerar um erro na lógica de habilitar e desabilitar as interrupções. Para evitar problemas, o uso destes métodos é desencorajado, e o uso dos métodos *spin_lock_irqsave* e *spin_unlock_irqrestore* são recomendados, pois estes fazem o correto tratamento de recursão das *flags* de IRQ do processador local. Estes dois métodos recebem mais um parâmetro, um *unsinged long*, que salva e restaura o contexto das *flags* que controlam o estado do processador, onde está contida a informação de habilitar/desabilitar interrupções. O exemplo a seguir demonstra o uso destas funções:

```
unsigned long flags;

spin_lock_irqsave(&mr_lock, flags);
[...] // sessão critica
spin_unlock_irqrestore(&mr_lock, flags);
```

O mesmo pode acontecer com as *softirqs* em uma sessão critica que é compartilhada entre uma *softirq* e uma tarefa executando no *kernel*. Nestes casos, o *spinlock* deve ser adquirido com as *softirqs* desabilitadas. Para este caso também foram criadas as funções auxiliares *spin_lock_bh* e *spin_unlock_bh*, exemplificadas a seguir:

```
spin_lock_bh(&mr_lock);
```

```
[...] // sessão critica
spin_unlock_bh(&mr_lock);
```

3.2.2 *Read/Write spinlocks*

Em alguns casos, os *spinlocks* são utilizados em variáveis que pode-se, claramente, distinguir entre funções que fazem somente a leitura e funções que fazem somente escrita nas variáveis protegidas pelo *spinlock*. Nestes casos, o acesso exclusivo aos dados é necessário quando há uma escrita nos dados, porém, pode haver concorrência quando se tem somente a leitura. Por exemplo, três tarefas podem ler um dado compartilhado de maneira concorrente, não gerando assim contenção, havendo contenção apenas quando uma tarefa espera para escrever, ou as outras tarefas esperam um dado que está sendo escrito.

Para estes casos, foi implementado o *Read/Write spinlock*. Com este *lock* é possível adquirir um *spinlock* para leitura ou para escrita, permitindo acesso concorrente quando há somente leitores e exclusivo quando há um escritor (LOVE, 2010). Como nos *spinlocks*, é possível definir um *rw_lock* de maneira estática:

```
DEFINE_RWLOCK(mr_lock);
```

ou de maneira dinâmica:

```
spinlock_t mr_lock;
rwlock_init(&mr_lock);
```

Para usar o *Read/Write_spinlock* como leitura, usa-se as funções *read_lock* e *read_unlock*, como no exemplo a seguir:

```
read_lock(&mr_lock);
[...] // sessão critica - somente leitura
read_unlock(&mr_lock);
```

Para usar o *Read/Write_spinlock* como escrita, usa-se as funções *write_lock* e *write_unlock*, como no exemplo a seguir:

```
write_lock(&mr_lock);
[...] // sessão critica - somente leitura
write_unlock(&mr_lock);
```

Como nos *spinlocks*, existe um método para adquirir o *lock* sem ficar esperando caso este esteja ocupado:

```
write_trylock(&mr_lock);
read_trylock(&mr_lock);
```

Caso o valor de retorno seja diferente de 0 o *lock* foi adquirido, caso 0, ele já está sendo utilizado.

3.2.2.1 *Deadlock* com *read/write spinlocks*

As mesmas formas de *deadlock* dos *spinlocks* são possíveis nos *read/write_spinlocks*, por isto também existem as chamadas equivalentes para desabilitar as IRQs e as *softirqs*.

Para desabilitar e habilitar as IRQs do processador corrente, incondicionalmente:

```
write_lock_irq(&mr_lock);
read_lock_irq(&mr_lock);

write_unlock_irq(&mr_lock);
read_unlock_irq(&mr_lock);
```

Para desabilitar e habilitar as IRQs do processador corrente, salvando as *flags* de IRQ:

```
write_lock_irqsave(&mr_lock, flags);
read_lock_irqsave(&mr_lock, flags);

write_unlock_irqrestore(&mr_lock, flags);
read_unlock_irqrestore(&mr_lock, flags);
```

Por fim, para desabilitar e habilitar as *softirqs*:

```
write_lock_bh(&mr_lock);
read_lock_bh(&mr_lock);

write_unlock_bh(&mr_lock);
read_unlock_bh(&mr_lock);
```

Além dos casos de *deadlock* já exemplificados com os *spinlocks*, existe um específico do *read/write locks*: não é possível fazer o *upgrade* de *read_lock* para *write_lock*. Por exemplo, o seguinte código irá causar um *deadlock*:

```
read_lock(&mr_lock);
```

```
write_lock(&mr_lock);

write_unlock(&mr_lock);
read_unlock(&mr_lock);
```

O *write_lock* ficará em espera ocupada eternamente, esperando o *read_unlock* que nunca executará.

Um detalhe importante é que o leitores sempre tem precedência sobre os escritores. Enquanto houver um leitor segurando o *lock*, o escritor não pode executar. Como os leitores podem obter o *lock* de maneira concorrente, mesmo que um escritor esteja esperando pelo *lock*, novos leitores poderão adquirir o *lock*, assim, podendo postergar por um período indeterminado a obtenção do *lock* pelo escritor.

3.2.3 Semáforos

No Linux os semáforos, diferente dos *spinlocks*, são *locks* que podem dormir. Quando uma tarefa tenta adquirir um semáforo e este está indisponível, o semáforo coloca a tarefa em uma lista de espera e põe a tarefa para dormir. Quando o semáforo se torna disponível, uma das tarefas na fila de espera é acordada e esta pode adquirir o semáforo.

Por fazer a tarefa dormir enquanto espera pelo *lock*, o semáforo é aconselhado para sessões críticas com longa duração, em que o tempo de execução das operações de suspender e reativar a tarefa sejam menores que a seção crítica. Por esta característica, o semáforo só pode ser utilizado no contexto de processo, isto porque não é permitido dormir no contexto de IRQs. Isto também implica que, quando uma tarefa possui um *spinlock*, esta não pode obter um semáforo, isto porque não é permitido a uma tarefa dormir enquanto retém um *spinlock*. Por outro lado, diferente do *spinlock*, é possível que uma tarefa com um semáforo suspenda.

A escolha entre semáforos e *spinlock* é então dada pelo fato de poder ou não dormir. Caso seja possível dormir, a escolha fica com o tempo em que o *lock* vai ser retido. Se for um curto espaço de tempo, melhor usar *spinlock*, caso o tempo seja maior, melhor usar semáforos (LOVE, 2010).

A implementação de semáforos permite que um número específico de tarefas possam acessar determinado recurso, isto é determinado por um contador. Para implementar exclusão mútua usando um semáforo, basta inicializa-lo com um contador igual a 1. Como a utilização de semáforo com o contador igual a 1 tornou-se um método de exclusão mútua muito utilizado, foi implementado no *kernel* uma forma genérica de utiliza-lo. A esta forma foi dado o nome de *mutex*, descrito na seção 3.2.4.

3.2.3.1 API de semáforos

Os semáforos são definidos pela estrutura *semaphore* e podem ser definidos de maneira dinâmica com a função *sema_init*. Diferente das funções de *spinlock*, o semáforo não possui uma forma de definição estática declarada em sua API. Porém, analisando o código viu-se que é possível declarar um semáforo estaticamente com a função DEFINE_SEMAPHORE, a qual declarar um semáforo com contador em 1, ou ainda, a macro __SEMAPHORE-_INITIALIZER para declarar um semáforo com um contador arbitrário, porém, segundo as convenções do kernel, as funções ou macros iniciadas com __ não devem ser utilizadas ao menos que você tenha certeza do que está fazendo. A seguir são dados exemplos de uso dos três modos de inicialização.

```
int count = 2;
/* dynamic */
struct semaphore mr_sema_1
sema_init(&mr_sema_1, count);

/* static in 1 */
DEFINE_SEMAPHORE(mr_sema_2);

/* static */
__SEMAPHORE_INITIALIZER(mr_sema_3, count);
```

Para adquirir um semáforo pode-se usar três métodos:

- *down*: adquire o semáforo em estado ininterruptível;
- *down_interruptible*: adquire o semáforo em estado interruptível; e
- *down_trylock*: tenta aquirir o semáforo sem bloquear, se conseguir retorna 0, se não, algum valor diferente de 0.

Sobre o modo interruptível, uma tarefa é posta para dormir em estado interruptível ou ininterruptivel. Caso um sinal seja enviado à tarefa em estado interruptível, a tarefa é acordada e o sinal entregue à tarefa. Já em um estado ininterruptível a tarefa não é acordada, atrasando a entrega do sinal até que esta seja liberada. Destas duas, é mais comum o uso da chamada *down_interruptible*. Para liberar o semáforo, utiliza-se somente uma função: *up*. Um exemplo de uso das três funções:

```
/* non interruptible */
down(&mr_sema);
```

```
/* interruptible */
down_interruptible(&mr_sema);

/* non block */
down_trylock(&mr_sema);

up(&mr_sema);
```

Além das três funções *down* documentadas nos livros, uma leitura no *kernel* revelou a função *down_killable*, que é um meio termo entre a *down* e *down_interruptible*. Ela retorna do estado dormindo somente se receber um sinal para terminar sua execução. Isto para evitar o caso onde o usuário envia um sinal para a tarefa terminar sua execução, porém, esta não trata o sinal por estar no estado ininterruptível.

3.2.3.2 Semáforo *Read/Write*

Assim como nos *spinlocks*, os semáforos também tem uma versão para *Read/Write*. A precedência dos leitores sobre os escritores é a mesma do *spinlock*: os leitores tem precedência sobre os escritores. Os semáforos *Read/-Write* não possuem contadores, a regra é a mesma dos *spinlocks*: um escritor em exclusão mútua, ou vários leitores concorrentes.

O semáforo *Read/Write* é definido pela estrutura *rw_semaphore*. Um semáforo *Read/Write* pode ser declarado de maneira estática ou dinâmica, como no exemplo a seguir:

```
/* dynamic */
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);

/* static */
DECLARE_RWSEM(mr_rwsem_2);
```

As funções para adquirir o semáforo para leitura são *down_read* e *down_read_trylock*, bloqueante e não bloqueante, respectivamente. Para escrita existem *down_write* e *down_write_trylock*. Diferente dos *spinlocks*, é possível fazer o *downgrade* de um *rw_semaphore*, isto é feito com a função *downgrade_write*.

```
/* read */
```

```
down_read(&mr_rwsem);
down_read_trylock(&mr_rwsem);
up_read(&mr_rwsem);

/* write */
down_write(&mr_rwsem);
down_write_trylock(&mr_rwsem);
up_write(&mr_rwsem);

downgrade_write(&mr_rwsem);
```

3.2.4 Mutex

Até pouco tempo, o único modo de implementar exclusão mútua sem espera ocupada era com a utilização dos semáforos com o contador em 1. Porém, por dar suporte a diversas operações, como de produtor e consumidor, a implementação dos semáforos possuía um grande custo computacional e também era de difícil de depuração. Buscando uma opção mais simples para exclusão mútua que colocasse as tarefas em espera para dormir, foi implementado o *mutex*. Apesar de ter um comportamento semelhante ao de um semáforo com contador em um, o *mutex* possui uma interface mais simples, melhor desempenho e mais restrições de uso, o que facilita a depuração do sistema (LOVE, 2010).

3.2.4.1 API

Um *mutex* é definido pela estrutura *mutex* e podem ser definidos de forma estática ou dinâmica, com o uso da macro DEFINE_MUTEX(mr_mutex), que definirá estaticamente o *mutex mr_mutex*, ou com a função *mutex_init*, como no exemplo a seguir:

```
struct mutex mr_mutex;
mutex_init(&mr_mutex);
```

Para adquirir um *mutex*, utiliza-se a chamada *mutex_lock*, que tenta adquirir o *mutex*. Caso este não esteja disponível, a tarefa é posta para dormir. Pode-se usar também o *mutex_trylock*, que tenta adquirir o *mutex*, caso consiga retorna 0, caso não consiga retorna imediatamente um valor diferente de zero. Para liberar um *mutex*, utiliza-se a função *mutex_unlock*, como no exemplo a seguir:

```
mutex_lock(&mr_mutex);
/* critical section */
mutex_unlock(&mr_mutex);
```

Por fim, a função *mutex_is_locked* verifica se o *mutex* passado como argumento está livre, retornando 0, ou ocupado, retornando 1.

3.2.4.2 *Mutex* vs Semáforo

A simplicidade e a eficiência dos *mutex* vem de algumas restrições de uso em relação aos semáforos.

Somente uma tarefa por vez pode adquirir um *mutex*, e a mesma tarefa que adquiriu o *mutex* deve liberá-lo. Isto significa que os *mutex* dificilmente serão utilizados pra sincronismos em diferentes contextos de execução, sendo geralmente adquiridos e liberados no mesmo contexto. Não é possível adquirir um *mutex* recursivamente. Uma tarefa não pode sair enquanto retem um *mutex* e os *mutex* não podem ser utilizados nos contextos de IRQs ou *softirqs*, o que é possível com os semáforos, por exemplo quando o produtor é uma IRQ que executa a função não bloqueante *up*.

Com estas restrições, foi possível implementar um método para garantir a segurança do uso dos *mutex*. Por exemplo, quando o *kernel* é configurado com a opção CONFIG_DEBUG_MUTEX, todas as operações com os *mutex* são depuradas e possíveis erros de uso ou infringimento das restrições são alertados.

3.2.5 *RT Mutex*

Os *RT Mutex* estendem a semântica dos *mutex* com o protocolo de herança de prioridade.

Em um *RT Mutex*, quando uma tarefa de baixa prioridade detêm um *mutex* que está bloqueando outra tarefa de alta prioridade, a tarefa de baixa prioridade herda a prioridade da tarefa de alta prioridade. Se a tarefa que herdou a prioridade bloquear em outro *mutex*, este propaga a prioridade para a outra tarefa, até que a tarefa que detém o *mutex* libere o *mutex* que bloqueou a tarefa de alta prioridade. Esta abordagem auxilia a diminuir o tempo de bloqueio de tarefas de alta prioridade, evitando a inversão ilimitada de prioridade (ROSTEDT, 2006).

Quando uma tarefa é bloqueada em um *RT mutex*, esta é adicionada em uma fila de espera. Esta fila é ordenada pela prioridade da tarefa, as tarefas de mesma prioridade obedecem a ordem de chegada.

A implementação do *RT mutex* tem um caminho rápido (*fastpath*) no caso da aquisição de *locks* sem bloqueio e da liberação de um *lock* que não tenha tarefas bloqueadas, ambas as operações não possuem o *overhead* de *locks* internos. Porém, para utilizar este caminho rápido, é necessário que a arquitetura possua a instrução *cmpxchg* (compara e troca) de forma atômica. Caso não tenha, o que é raro, as operações atômicas são garantidas por *spinlocks*.

3.2.5.1 API

Os *RT Mutex* são definidos pela estrutura *mutex*. E, assim como os outros *mutex*, podem ser inicializados de maneira estática como a macro DEFINE_RT_MUTEX ou de forma dinâmica com a função (na verdade é uma macro) *rt_mutex_init*, como no exemplo a seguir.

```
/* dynamic */
struct mutex mr_mutex;
rt_mutex_init(&mr_mutex);

/* static */
DEFINE_RT_MUTEX(mr_mutex2);
```

Existem quatro modos de se adquirir um *RT mutex*. A função *rt_mutex_lock* tenta adquirir o *mutex* de maneira ininterruptível, já função *rt_mutex_lock_interruptible* tenta de maneira interruptível. A função *rt_mutex_lock_killable* só é interruptível por sinais que terminam o processo. A função *rt_mutex_timed_lock* é um modo de adquirir o *mutex* com um limite máximo de tempo de espera, caso contrário, retorna como falha. Por fim, a função *rt_mutex_trylock* que tenta adquirir o *mutex* sem ir para a fila de espera, caso adquira o *mutex*, esta retorna 0, caso não, um valor diferente de 0. Para testar se um *rt_mutex* está liberado existe a função *rt_mutex_is_locked*. Para liberar um *RT mutex* utiliza-se a função *rt_mutex_unlock*. Todas são exemplificadas a seguir:

```
rt_mutex_lock(&mr_mutex);
rt_mutex_lock_interruptible(&mr_mutex, detect_deadlock);
rt_mutex_lock_killable(&mr_mutex, detect_deadlock);
rt_mutex_timed_lock(&mr_mutex, &timeout, detect_deadlock);
rt_mutex_trylock(&mr_mutex);

rt_mutex_is_locked(&mr_mutex);

rt_mutex_unlock(&mr_mutex);
```

3.2.5.2 RT *Mutex* no PREEMPT_RT

No *kernel* do Linux com o *patch* PREEMPT_RT, tanto os *spinlocks* quanto os *mutex* são convertidos em *RT mutex*. Neste ponto surge uma questão, se os *RT mutex* põem as tarefas para dormir, como eles podem substituir os *spinlocks*? No PREEMPT_RT muitas sessões do *kernel* que originalmente estão em um contexto de interrupção, como os tratadores de interrupção e as *softirqs*, são convertidos em tarefas que rodam no espaço de endereçamento do kernel, desta forma, os *spin_locks* utilizados nessa sessões podem ser convertidos em *rt_mutex*. Este mapeamento é feito redefinindo as funções dos *mutex* e *spinlocks* para os *rt_mutex* quando o *kernel* está configurado com a opção CONFIG_PREEMPT_RT_FULL.

Em trechos do *kernel* onde mesmo no PREEMPT_RT não é possível dormir, os *spinlocks* são utilizados, sendo chamados com o prefixo *raw_*, por exemplo, *raw_spin_lock(&mr_lock)*.

3.2.6 RCU

A RCU (*Read-Copy Update*) é um mecanismo de sincronização que permite que leitores e escritores possam acessar dados compartilhados de forma concorrente, como nos *read/write locks*. Porém, a RCU utiliza de algumas características das arquiteturas dos processadores atuais, como a atomicidade das operações de ponteiros alinhados na memória, para conseguir um melhor desempenho, nunca deixando leitores esperando por um escritor (MCKENNEY, 2005). A Figura 2 é uma comparação da RCU com o *Read/-Write lock* (MCKENNEY, 2007).

No exemplo da Figura 2, a RCU permite que as tarefas não bloqueiem durante a execução da seção crítica, diferente da execução utilizando os *Read/Write spinlock*, onde há a contenção em ambos os lados. Devido ao paralelismo, os leitores que adquirem acesso ao dado com as informações de antes destes serem alterado, acabam ler uma versão desatualizada da informação, porém, apesar de desatualizada, estas estão consistentes. Os leitores que adquiriram o dado após a atualização terão o dado já atualizado (em cinza escuro). Como pode-se ver, a RCU aumenta o paralelismo das operações.

Na RCU os leitores nunca bloqueiam. A RCU garante que os dados acessados sempre serão consistentes, isto é, um dado compartilhado nunca será visto em parte atualizado e em parte não. Porém, os dados podem ser ou não o mais atual.

Com a RCU a tarefa dos escritores é dividida em duas partes, a de remoção (*removal*) e recuperação (*reclamation*) dos dados. Quando um es-

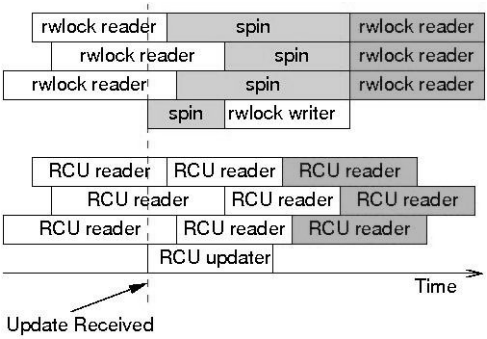

Figura 2 – Comparação entre *Read/Write spinlock* e a RCU (MCKENNEY, 2007)

critor deseja alterar um dado ele primeiro pede a remoção deste, o que ocorre sem bloqueio. Quando um escritor quer atualizar o dado, este pede para o dado ser recuperado (*reclamation*). Esta operação faz o escritor dormir até que todos os leitores que tinham acesso ao dado antigo terminem sua sessão de leitura. Após o dado ser atualizado, o escritor pode fazer o que quiser com o dado que retirou, por exemplo, liberar a memória utilizada por este. Uma forma de entender melhor o uso da RCU é com o seu uso em uma lista encadeada.

A lista encadeada é de dados do tipo da estrutura *foo*, definida como:

```
struct foo {
  struct list_head list;
  int a;
  int b;
  int c;
};
LIST_HEAD(head);
```

A lista *head*, em certo ponto de execução no tempo se apresenta como na Figura 3.

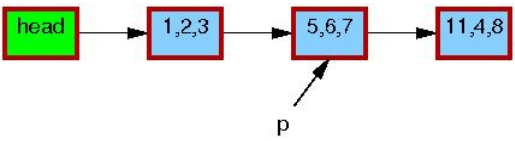

Figura 3 – Exemplo RCU: lista encadeada

Os quadros com as bordas largas significam que alguma tarefa pode estar utilizando a entrada na lista. O dado apontado por *p* é o que pretende-se atualizar com novos valores. Para isto, é necessário alocar uma nova estrutura do mesmo tipo e fazer a copia dos dados e do ponteiro da lista encadeada, e nesta nova estrutura pode ser feita a atualização dos dados, como na Figura 4.

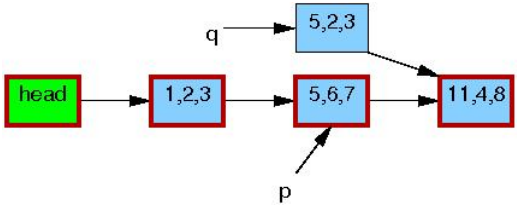

Figura 4 – Exemplo RCU: cópia e atualização dos dados

A partir deste momento, pode-se apontar o dado anterior da lista encadeada para o novo dado, para que novas requisições sejam direcionadas a nova entrada na lista encadeada. Exemplificado na Figura 5.

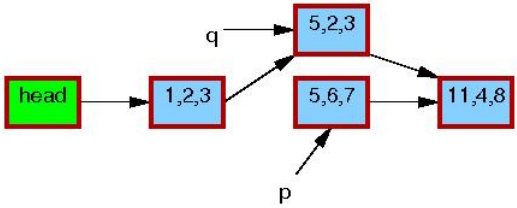

Figura 5 – Exemplo RCU: atualização da lista encadeada

Neste momento as tarefas que iniciaram a leitura antes da atualização terão acesso a versão antiga da estrutura e as que adquiriram após a atualização à nova versão. Porém, apesar das versões serem diferentes, ambas são consistentes. Neste ponto a tarefa que atualizou o dado espera até que a última tarefa que esteja lendo a versão antiga termine a leitura, como na Figura 6:

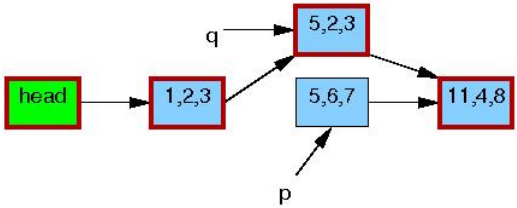

Figura 6 – Exemplo RCU: dado liberado para remoção

Após o dado ser liberado, a tarefa pode fazer o que quiser com este, por exemplo, liberar a memória utiliza, como exemplificado na Figura 7.

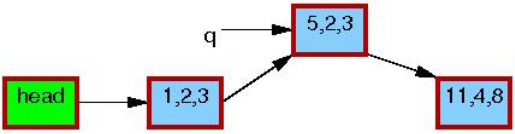

Figura 7 – Exemplo RCU: lista encadeada após a atualização dos dados

Como pode-se observar, apesar de aumentar o número de passos para a execução, em comparação com os *Read/Write mutex* e *spinlocks*, o bloqueio gerado pela RCU é menor. No lado do leitor, não há contenção na RCU, já *Read/Write mutex* ou *spinlock* existe bloqueio, porém, este bloqueio é limitado ao tempo de execução do escritor. Já no lado do escritor, na RCU o escritor deve esperar pelo fim da execução leitores que adquiriram o dado antes da atualização do dado, o que limita o tempo de bloqueio do escritor a no máximo a maior seção critica dos leitores. Já nos *Read/Write spinlocks* e *mutex* o tempo de bloqueio do escritor não pode ser definido, pois mesmo após solicitar o *lock*, novos leitores podem entrar na seção crítica, podendo, no pior caso, postergar indefinidamente o acesso do escritor a seção crítica.

3.2.6.1 API

A API da RCU possui diversas chamadas, porém, pode ser exemplificada usando estas cinco:

- *rcu_read_lock()*;
- *rcu_read_unlock()*;
- *synchronize_rcu()*;
- *call_rcu()*;
- *rcu_assign_pointer()*;
- *rcu_dereference()*.

As funções *void rcu_read_lock(void)* e *void rcu_read_unlock(void)* são utilizadas pelos leitores para sinalizar a entrada e a saída da sessão critica. Não é permitido bloquear entre estas duas chamadas. Porém, em um *kernel* configurado com a opção CONFIG_TREE_PREEMPT_RCU, a tarefa pode

sofrer preempção enquanto executa a leitura de uma sessão critica da RCU. Um detalhe é que estas operações podem ser aninhadas.

A função *void synchronize_rcu(void)* é utilizada pelo escritor para sinalizar que este terminou a escrita e que novas requisições de leitura podem ser feitas no novo dado. Esta função bloqueia o escritor até que todos os leitores que possuam referência ao dado antigo tenham saído da sessão critica. Após todos saírem, a função retorna. A função *synchronize_rcu_expedited()* tem função análoga a *synchronize_rcu*, porém, esta apresenta um o tempo menor de espera pelo leitor, com um *overhead* maior de CPU que a função *synchronize_rcu*. Uma outra forma de fazer isto, de maneira não bloqueante, é usando a função *void call_rcu(struct rcu_head *head, void (*func)(struct rcu_head *head))*, esta função irá registrar uma *callback*, a qual será executada assim que todos os leitores com a referência antiga do dado tenham saído da sessão crítica. Neste caso, a *callback* será responsável por executar as operações depois que todos os leitores tenham terminado de usar o dado antigo.

A macro *typeof(p) rcu_assign_pointer(p, typeof(p) v);* é responsável por atualizar o valor de um ponteiro protegido pela RCU, por isto é utilizada somente no lado do escritor. Por fim, a macro *typeof(p) rcu_dereference(p);* é utilizada para buscar um ponteiro protegido pela RCU, não para remover a sua referência, mas não para sinalizar que ele está sendo usado e proteger este de ser removido durante a execução.

Um exemplo de uso destas funções é dado a seguir, com comentários explicando as suas operações.

```
struct foo {
int a;
char b;
long c;
};
DEFINE_SPINLOCK(foo_mutex);

struct foo *gbl_foo;

void foo_update_a(int new_a)
{
struct foo *new_fp;
struct foo *old_fp;

// Aloca a memoria para o novo dado
new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
```

```
spin_lock(&foo_mutex); // para proteger o ponteiro
old_fp = gbl_foo; // salva a referencia para a
                                  // memoria do dado antigo
*new_fp = *old_fp; // copia os dados para a nova variável.
new_fp->a = new_a; // atualiza a variável a da estrutura.

// atualiza o ponteiro protegido pela RCU.
rcu_assign_pointer(gbl_foo, new_fp);

spin_unlock(&foo_mutex);

// espera as outras tarefas de leitura terminar a execução
synchronize_rcu();
kfree(old_fp); // libera a memoria alocada anteriormente
}


int foo_get_a(void)
{
int retval;

rcu_read_lock(); // Indica o inicio da leitura

// lê o dado do ponteiro protegido
retval = rcu_dereference(gbl_foo)->a;

rcu_read_unlock(); // Indica o fim da leitura
return retval;
}
```

Além das funções básicas exemplificadas aqui, existem outras funções, que tem o mesmo papel, porém, com restrições e áreas de aplicação diferente. A Figura 8 dá uma breve descrição sobre estas (MCKENNEY, 2010).

Attribute	RCU	RCU BH	RCU Sched	SRCU
Purpose	Wait for RCU read-side critical sections	Wait for RCU-bh read-side critical sections & irqs	Wait for RCU-sched read-side critical sections, preempt-disable regions, hardirqs, & NMIs	Wait for SRCU read-side critical sections, allow sleeping readers
Read-side primitives	rcu_read_lock() rcu_read_unlock()	rcu_read_lock_bh() rcu_read_unlock_bh()	rcu_read_lock_sched() rcu_read_unlock_sched() rcu_read_lock_sched_notrace() rcu_read_unlock_sched_notrace() preempt_disable() preempt_enable() (and friends)	srcu_read_lock() srcu_read_unlock()
Update-side primitives (synchronous)	synchronize_rcu() synchronize_net()	synchronize_rcu_bh()	synchronize_sched()	synchronize_srcu()
Update-side primitives (expedited)	synchronize_rcu_expedited()	synchronize_rcu_bh_expedited()	synchronize_sched_expedited()	synchronize_srcu_expedited()
Update-side primitives (asynchronous/callback)	call_rcu()	call_rcu_bh()	call_rcu_sched()	N/A
Update-side primitives (wait for callbacks)	rcu_barrier()	rcu_barrier_bh()	rcu_barrier_sched()	N/A
Read side constraints	No blocking except preemption and "spinlock" acquisition.	No BH enabling	No blocking	No wait for synchronize_srcu()
Read side overhead	Simple instructions (free on !PREEMPT)	BH disable/enable	Preempt disable/enable (free on !PREEMPT)	Simple instructions, preempt disable/enable
Asynchronous update-side overhead (for example, call_rcu())	sub-microsecond	sub-microsecond	sub-microsecond	N/A
Grace-period latency	10s of milliseconds	10s of milliseconds	10s of milliseconds	10s of milliseconds
!PREEMPT_RT default implementation	RCU Sched	RCU BH	RCU Sched	SRCU
PREEMPT_RT default implementation	Preemptible RCU	RCU BH	RCU Sched	SRCU

Figura 8 – RCU API

A coluna RCU foi apresentada, a RCU *Sched* é utilizada em tratadores de NMI, IRQs, e em locais em que a preempção está desabilitada. A *RCU_bh* é utilizada em locais onde pode ocorrer negação de serviço, por exemplo, nas *softirqs* de recepção de pacotes de rede que podem monopolizar uma CPU em um ataque. Por fim, a SRCU é utilizada onde é possível que os leitores suspendam a execução.

3.3 INTERFERÊNCIA NA EXECUÇÃO

Além dos mecanismos de exclusão mútua, existem outras formas de interferir na execução temporal das tarefas. Por exemplo, é possível desabilitar as interrupções do sistema, logo, uma interrupção do relógio pode ter que esperar que as interrupções sejam reabilitadas para poder chamar o tratador de interrupção, causando um atraso na liberação da tarefa. Nas próximas seções serão revisadas as possíveis interferências na execução do sistema, começando pela preempção.

3.3.1 Preempção

Por ser um *kernel* preemptivo, uma *thread* pode ser interrompida a qualquer momento, permitindo que uma *thread* de maior prioridade execute. Porém, existem sessões do *kernel* onde uma rotina não pode ser posta para dormir, por exemplo, para garantir restrições ao modelo de exclusão mútua, como nos *spinlocks*. Por isto foram criados métodos para desabilitar a preempção do sistema, impedindo que uma tarefa seja interrompida por outra tarefa, mesmo que de maior prioridade.

A preempção de um processador pode ser desabilitada com a função *pre-*

empt_disable(), e habilitada com a função *preempt_enable()*. Para cada *preempt_disable* deve-se ter uma chamada *preempt_enable*. Estas chamadas podem ser aninhadas, o número de aninhamentos pode ser recuperado do contador de preempção, a função *preempt_count()* retorna este valor.

A função *preempt_enable*, quando chamada, verifica se o contador de preempção irá ficar em 0, se estiver, a preempção do sistema estará ativa novamente. Como é possível que uma tarefa de maior prioridade esteja pronta para executar, ao reabilitar a preempção, a rotina de escalonamento é chamada. Em casos onde não se deseja verificar se existem tarefas aptas a executar, pode-se utilizar a função *preempt_enable_no_resched()*, que reabilita a preempção sem checar se existe uma nova tarefa de maior prioridade apta a execução.

3.3.2 IRQs

As interrupções são a principal forma interferir na execução temporal do sistema, afinal, o *kernel* do Linux é dirigido a interrupções, sendo elas mascaráveis, não mascaráveis, *traps*, etc. Por exemplo, uma tarefa desabilita as interrupções, no exato momento em que a interrupção é desabilitada chega uma interrupção do timer, esta interrupção irá ficar em espera para ser atendida até que as interrupções do processador sejam reabilitadas, causando assim um atraso na entrega da interrupção do *timer*.

Apesar da interferência na execução, a possibilidade de desabilitar as interrupções é necessária, principalmente para garantir sincronização. Desabilitar as interrupções garante que um tratador de interrupção não irá causar a preempção de uma sessão de código. Ao desabilitar a interrupção a preempção também é desabilitada.

O *kernel* do Linux inclui funções para desabilitar todas as interrupções de um processador ou para desabilitar uma interrupção em todos os processadores, documentadas a seguir.

3.3.2.1 Controlando as interrupções de um processador

Existem duas formas de desabilitar as interrupções do processador corrente: de maneira incondicional ou condicional. A primeira é feita através das funções *local_irq_disable* e *enable*, como a seguir:

```
local_irq_disable()
/* Interrupções desabilitadas */
local_irq_enable()
```

Por ser de maneira incondicional, estas funções não podem ser aninhadas, como no exemplo a seguir:

```
/* Desabilita as interrupções */
local_irq_disable();
/* Operação nao tem efeito */
```

```
local_irq_disable();
/* Habilita as interrupções */
local_irq_enable();
/*
 * Erro! As interrupções são habilitadas!
 * os codigos executados aqui estão desprotegidos
 * a próxima operação não tem efeito.
 */
local_irq_enable();
```

Isto restringe o uso destas funções para locais onde, garantidamente, não haverá aninhamento destes controles de interrupção, por isto o uso destas funções é desencorajado. Para resolver esta restrição, outras duas funções - macros na verdade, estão disponíveis. Esta macros salvam as *flags* do processador para depois serem restauradas. Assim, permitindo o aninhamento das chamadas para desabilitar/habilitar as IRQs. Estas funções são *local_irq_save* e *local_irq_restore*. Exemplificadas a seguir.

```
/* Flags para salvar o estado das irqs */
unsigned long flags;

/* desabilita as interrupções e salva as flags*/
local_irq_save(flags);

/* interrupção desabilitada */

/* habilita as interrupções e restaura as flags*/
local_irq_restore(flags);
```

Assim, é possível desabilitar as interrupções do processador corrente de maneira segura.

3.3.2.2 Controlando uma linha de interrupção

Além de ser possível desabilitar todas as interrupções de um processador, em alguns casos é desejável desabilitar apenas uma IRQ especifica para todo o sistema. Por exemplo, pode ser necessário desabilitar a entrega de uma interrupção antes de manipular o seu estado. O Linux prove quatro funções para esta tarefa.

A primeira é a função *void disable_irq(unsigned int irq)* que desabilita a interrupção passada como argumento em todos os processadores. Caso o tratador de interrupção esteja executando, a função irá bloquear até que o tratador termine. A função *void disable_irq_nosync(unsigned int irq)* também desabilita a interrupção em todos os processadores, porém, sem esperar algum tratador de interrupção que possa estar executando.

A função *void synchronize_irq(unsigned int irq)* irá esperar pelo tratador de interrupção de uma IRQ especifica, antes de retornar.

Por fim a função *void enable_irq(unsigned int irq)* habilita a interrupção.

3.3.3 Migração de threads desabilitadas

Na versão SMP, também se faz necessário desabilitar as migrações de uma *thread*, principalmente nas rotinas de escalonamento. Isto acaba por afetar a escalonabilidade do sistema, por exemplo, impedindo que uma tarefa migre para outro processador. Isto é feito com o auxilio de duas funções: *void migrate_disable(void)* que desabilita a migração da tarefa atual no processador atual, a operação reversa é feita com a função "*void migrate_enable(void)*"

3.4 CONSIDERAÇÕES FINAIS

Atualmente no *kernel* do Linux, por não ter nascido com as premissas de tempo real, a maioria dos mecanismos de exclusão mútua foram desenvolvidos sem o objetivo de ser temporalmente determinista. Por exemplo, no *Read/Write spinlock*, o escritor não possui um tempo limite de espera pelo *lock*. Outro exemplo de indeterminismo vem dos *spinlocks*, pois não é possível garantir que uma tarefa de maior prioridade irá obter o *lock* quando, duas tarefas, de diferentes prioridades, estão em espera ocupada pelo mesmo *lock*, em processadores diferentes.

Por outro lado, tanto a RCU quanto os *RT Mutex*, apresentam comportamento determinista. Na RCU, a única tarefa que fica em estado de espera é o escritor. O tempo máximo de espera pelo escritor é limitado pelo maior tempo de ocupação do lado do leitor.

Os *RT Mutex*, foram criados com o objetivo de ser determinista. A implementação conta com um controle de herança de prioridade para limitar a inversão de prioridades. Porém, por executar este controle, o tempo de execução deste tipo de exclusão mútua é maior que os demais e, por isto, ele é utilizado apenas na versão de tempo real do Linux, o PREEMPT_RT.

Em relação a preempção, apesar de ser preemptivo, a possibilidade de desabilitar a preempção acaba por fazer o *kernel* um sistema misto, preemptivo com pontos de não preempção. Apesar disto causar inversões de prioridade, esta inversão é limitada, no pior caso, a maior seção de preempção do sistema. Como a preempção afeta apenas tarefas que ficam prontas para executar, mas não executam enquanto a tarefa corrente não habilitar a preempção, este caso se assemelha mais a um caso de atraso de ativação que de bloqueio por exclusão mútua.

Outra forma de adicionar atraso na ativação de uma tarefa é desabilitando as interrupções. Ao desabilitar as interrupções de um processador, todas as notificações do *hardware* que chegam via interrupções são atrasadas até as interrupções serem reabilitadas. Um exemplo critico disto é o atraso na entrega de interrupções do *timer*, que disparam tarefas de tempo real periódicas.

Por fim, o fato de desabilitar a migração de uma tarefa pode afetar a decisão de escalonamento do sistema, deixando temporariamente uma tarefa global com as

restrições de uma tarefa fixa em um processador.

Durante este capítulo pode-se observar a grande variedade de mecanismos de sincronismo e exclusão mútua do *kernel* do Linux. Esta grande variedade se dá, tanto por causa das premissas impostas pelos contextos de execução, quanto pela necessidade de se obter um maior desempenho do sistema. Sendo a métrica de desempenho, no caso padrão, uma maior vazão de processamento, e no caso das implementações de tempo real, um maior determinismo.

Existe na literatura outras formas de exclusão mútua, como os *seqlocks*, porém, estes métodos de exclusão mútua acabam por utilizar os métodos citados, por isto não foram citados.

4 FERRAMENTAS DE TRACE

Este capítulo é uma análise das ferramentas de *trace* existentes no Linux, as quais são utilizadas como referência para o *trace* de processos e tempos de resposta, principalmente em sistemas Linux executando tarefas em tempo real. O objetivo desta análise é definir qual das ferramentas atualmente disponíveis mais se adéqua as necessidades deste trabalho. As ferramentas escolhidas para a análise foram: Feather-trace, ferramenta de *trace* desenvolvida pela Universidade de Carolina do Norte para *trace* de processos em seu Linux de tempo real, o Litmus RT; o Ftrace, ferramenta de *trace* oficial do *kernel* do Linux; e o LTTng (*Linux Trace Toolkit - next generation*) ferramenta de *trace* que unifica *trace* no espaço do usuário e do *kernel*, dando acesso as funções de *trace* já existentes do *kernel* do Linux.

Cada uma destas ferramentas possuí uma seção que apresentar as suas principais características e quais mecanismos foram utilizados para implementa-las. Ao final é feita uma comparação entre as ferramentas e a capacidade destas ferramentas de obter as métricas de tempo real relevantes.

4.1 FEATHER-TRACE

O Feather-trace é uma ferramenta de *trace* de eventos desenvolvida para ser utilizada com o Litmus RT (BRANDENBURG, 2012). O Litmus RT é um projeto composto de um *patch* para o *kernel* do Linux, um conjunto de bibliotecas e ferramentas que provêem suporte ao desenvolvimento de escalonadores de tempo real para sistemas multiprocessados no Linux (LITMUSRT, 2012a).

O Feather-trace surgiu como uma ferramenta de *trace* de eventos, tanto para espaço do usuário quanto do *kernel*. Devido à necessidade de medir latências e *overheads* no Litmus RT, principalmente no que diz respeito a decisões de escalonamento e bloqueio, o Feather-trace foi utilizado para fazer medições de tempos pertinentes a sistemas de tempo real. Hoje esta faz parte e é distribuída junto com o Litmus RT. O Feather-trace foi construído com as seguintes características (BRANDENBURG, 2012):

- Baixo *overhead*: É necessário apenas uma instrução para habilitar e desabilitar eventos;
- Seguro para execução em sistemas multiprocessados: Não há necessidade de mecanismos de exclusão mútua;
- Portabilidade: Não requer serviços do sistema operacional, como primitivas de sincronização;
- Independente do contexto de execução: Pode ser executado no contexto de interrupções e segmentos não preemptivos;
- Pode ser utilizado em espaço do usuário.

4.1.1 Componentes

O Feather-trace é composto por dois módulos: Um *tracer* embutido no *kernel* do Litmus RT e um conjunto de ferramentas e bibliotecas em espaço de usuário.

No *kernel* são implementados e utilizados os eventos de *trace*, estes eventos podem ser habilitados e desabilitados em tempo de execução, isto é feito através de um *driver* de caractere do Linux. Este *driver* aceita comandos via IOCTL que habilitam e desabilitam os eventos. Quando este *driver* é lido por uma ferramenta, ele exporta os dados coletados dos eventos habilitados (BRANDENBUG, 2007). Os dados são exportados pelo *driver* em formato binário, para posteriormente serem interpretados e formatados de maneira intuitiva pelas ferramentas no espaço do usuário.

As ferramentas em espaço do usuário são utilizadas para habilitar os eventos de trace no *kernel* e ler e interpretar os dados de *trace* coletados pelo *kernel* (BRANDENBUG, 2007). Existem duas ferramentas que se utilizam da infraestrutura do Feather-trace: o sched_tracer (LITMUSRT, 2012b) e o Unit-Test (MOLLISON, 2011).

O sched_trace utiliza a infra-estrutura do Feather-trace para fazer o *trace* das atividades que envolvem o escalonamento de tarefas, tais como: a ativação de uma tarefa, o início da execução de uma tarefa, trocas de contexto etc. O sched_trace é ativado através do *script* st_trace, que é uma das ferramentas do Feather-trace (LITMUSRT, 2012b). Com o *trace* gerado com o st_trace/sched_trace, a ferramenta Unit-Test é capaz de gerar, de maneira gráfica e intuitiva, uma imagem de como foi executado o escalonamento das tarefas, facilitando a visualização e depuração do sistema (MOLLISON, 2011).

4.1.2 Utilização

Para utilizar o Feather-trace é necessário selecionar um escalonador de tempo real do Litmus RT para executar uma tarefa de tempo real. A ferramenta setsched pode ser utilizada para configurar o escalonador de tempo real do Litmus RT, a Figura 9 exibe as opções existentes no sistema avaliado (LITMUSRT, 2012b).

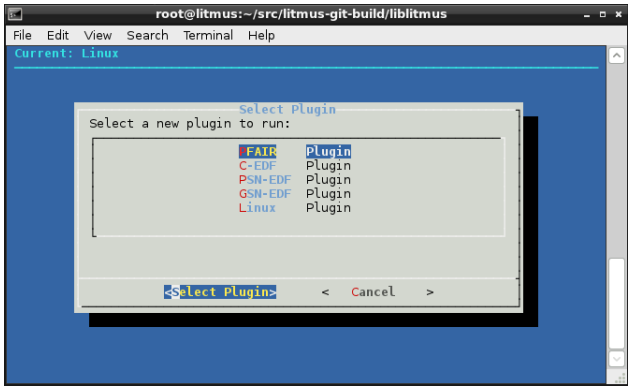

Figura 9 – Opções do setched

Após configurar o escalonador, é necessário executar uma tarefa que utilize a API de tempo real do Litmus RT. O pacote Liblitmus possui o esqueleto de duas aplicações de tempo real: base_task e base_task_mt. Ambas aplicações apresentam apenas o esqueleto de uma aplicação para o Litmus RT. Para conduzir os testes, foi adicionar um código para simular a execução de uma tarefa. Após esta alteração as aplicações foram executadas a fim de se obter os dados de *trace*.

O ft_cat é a ferramenta que controla o Feather-trace. Esta ferramenta possui como argumentos o *driver* do Feather-trace e os eventos de *trace*. O *driver* do Feather-trace é criado de maneira automática em */dev/litmus/ft_trace0* e os possíveis eventos são (LITMUSRT, 2012b):

- SCHED_START, SCHED_END: mede o tempo gasto nas decisões de escalonamento;
- CXS_START, CXS_END: mede o tempo gasto em trocas de contexto;
- SCHED2_START, SCHED2_END: mede o tempo gasto executando as limpezas e gerenciamento de atividades após a troca de contexto, este tempo faz parte do *overhead* de escalonamento, porém, não pode ser adicionado ao intervalo [SCHED_START, SCHED_END] por questões técnicas;
- TICK_START, TICK_END: mede o *overhead* ocorrido no inicio do *quantum* de escalonamento;
- PLUGIN_TICK_START, PLUGIN_TICK_END: como [TICK_START, TICK_END], porém mede somente o tempo gasto pelo *active scheduling plugin*;
- PLUGIN_SCHED_START, PLUGIN_SCHED_END: como [SCHED_START, SCHED_END], porém mede somente o tempo gasto pelo *active scheduling plugin*;
- RELEASE_START, RELEASE_END: mede o tempo gasto para enfileirar um novo *job* lançado na fila de pronto.

Além dos eventos descritos na documentação do Feather-trace, analisando o código fonte do Feather-trace, percebe-se que existem outros eventos disponíveis na configuração padrão do Litmus RT:

- LOCK_START, LOCK_SUSPEND, LOCK_RESUME, LOCK_END: tempos entre iniciar a aquisição de um mutex, suspender a tarefa em um mutex, retornar a execução de um mutex, e terminar a aquisição de um mutex;
- UNLOCK_START, UNLOCK_END: *overhead* para liberar um mutex;
- RELEASE_LATENCY: A latencia de ativação de uma tarefa.

Em um experimento de uso da ferramenta, utilizou-se o seguinte comando:

```
ftcat /dev/litmus/ft_trace0 \
CXS_START CXS_END SCHED_START SCHED_END SCHED2_START SCHED2_END \
LOCK_START LOCK_SUSPEND LOCK_RESUME LOCK_END  RELEASE_LATENCY \
> my_trace
```

O *trace* tem como saída o arquivo my_trace, este arquivo possui um formato binário. Para extrair o *trace* é necessário utilizar uma ferramenta que transforme os dados de binário em um formato textual, existem duas ferramentas que podem fazer isto, o ftdump e o ft2csv. O utilitário ftdump extrai todas as informações do arquivo de trace, como exemplo de execução e saída:

```
ftdump my_trace
struct timestamp:
         size              =  16
         offset(timestamp) =   0
         offset(seq_no)    =   8
         offset(cpu)       =  12
         offset(event)     =  13
CXS_START       seq:17072  timestamp:2843615169444 \
    cpu:2  type:BE       irq:0 irqc:00
CXS_END         seq:17073  timestamp:2843615170950 \
    cpu:2  type:BE       irq:0 irqc:00
RELEASE_LATENCY seq:17074  timestamp:16075  cpu:1  \
    type:RT       irq:1 irqc:04
SCHED_START     seq:17075  timestamp:2843616783393 \
    cpu:1  type:UNKNOWN  irq:0 irqc:00
SCHED_END       seq:17076  timestamp:2843616786090 \
    cpu:1  type:RT       irq:0 irqc:00
CXS_START       seq:17077  timestamp:2843616786296 \
    cpu:1  type:RT       irq:0 irqc:00
CXS_END         seq:17078  timestamp:2843616789102 \
    cpu:1  type:RT       irq:0 irqc:00
```

Já o ft2csv extrai os dados no formato separado por virgula, que é facilmente importado para ferramentas como o Excel ou Matlab, como exemplo de saída:

```
[root@litmus trace]# ft2csv SCHED_START my_trace
[...]
2843616783393, 2843616786090, 2697
2843625475286, 2843625477598, 2312
2843647488112, 2843647490263, 2151
2843656168932, 2843656171662, 2730
2843662701381, 2843662703147, 1766
2843688524711, 2843688530141, 5430
2843866229410, 2843866231995, 2585
2843874908377, 2843874910771, 2394
2843896953177, 2843896955531, 2354
2843905594589, 2843905596856, 2267
2843912131234, 2843912132834, 1600
2843934588545, 2843934590981, 2436
2844115663722, 2844115666128, 2406
[...]
```

A saída acima exibe os tempos de cada chamada SCHED_START e SCHED_END e o delta entre estas.

4.1.3 Como Feather-trace foi desenvolvido

Como descrito nas características gerais, o Feather-trace foi desenvolvido para ser *lock-free*, ser seguro para sistemas multiprocessados e adicionar baixo *overhead* ao sistema. Estas características são fundamentais para o *trace* de tempos em um sistema de tempo real em multiprocessadores, também garantem que o código possa ser utilizado em qualquer contexto do sistema operacional (BRANDENBUG, 2007).

Os eventos de *trace* do Feather-trace são adicionados de forma estática no código. O código de *trace* é adicionado diretamente ao código do *kernel* do Litmus RT antes da compilação. Apesar desta abordagem adicionar maior processamento no sistema operacional, a decisão entre executar ou não um ponto de *trace* é feita de maneira dinâmica com o custo de uma única operação de máquina: um salto incondicional, fazendo assim o tempo de decisão entre executar ou não um ponto de *trace* o menor possível, o de uma única operação (BRANDENBUG, 2007).

4.1.3.1 Gerenciamento de eventos

Os eventos cadastrados no sistema estão registrados em uma tabela global de eventos. Os eventos são definidos pela estrutura *trace_event*, que possui os seguintes atributos:

```
struct trace_event {
        long    id;
        long    count;
        long    start_addr;
        long    end_addr;
};
```

O *id* é um identificador de 32 bits do evento, *count* é um contador que identifica se o evento está habilitado ou não, *start_addr* é o endereço do início das instruções de *trace* e *end_addr* o endereço do fim das instruções de *trace*. As instruções de *trace* são executadas por um código em linguagem de máquina, existe quatro possíveis funções que implementam as funções de *trace*:

- *ft_event(id, callback)*
- *ft_event0(id, callback)*
- *ft_event1(id, callback, param)*
- *ft_event2(id, callback, param, param2)*
- *ft_event3(id, callback, p, p2, p3)*

Todas as chamadas recebem o *id* do evento de *trace* e uma função de *callback* a ser executada pelo evento. A diferença entre as chamadas são os parâmetros passados para a função de *callback*. Abaixo o código, para ilustrar, da função *ft_event1*:

```
#define ft_event1(id, callback, param)      \
__asm__ __volatile__(                       \
    "1: jmp 2f                  \n\t" \
    " movq %0, %%rsi            \n\t" \
    " movq $" #id ", %%rdi      \n\t" \
    " call " #callback "        \n\t" \
    _EVENT_TABLE(id,1b,2f)              \
    "2:                         \n\t" \
: : "r" (param)  : CLOBBER_LIST)
```

O código é bastante simples, este executa um desvio incondicional, que serve para definir se o *trace* deve ou não ser executado, empilhar os argumentos a serem enviados à função de *callback* e chamar a função.

Para habilitar um evento, a função ft_enable_event recebe o ID do evento e percorre a tabela global de eventos do sistema. Caso o evento identificado pelo ID exista no sistema e esteja desabilitado esta função busca o endereço de início da função de *trace*, incrementa o ponteiro em um *byte* e altera o conteúdo deste para 0. O que isto faz na verdade é dizer que o desvio incondicional deve ser de 0 bytes, assim, executando o código do evento. Para desabilitar o processo é semelhante, porém, ao invés de setar em 0 o destino do desvio incondicional, este é setado para o (endereço do fim do evento - endereço de início do evento + 1), assim, pulando o código de evento. Como na arquitetura Intel uma operação em um *byte* na memória, independente do alinhamento, é atômica, não se faz necessário a utilização de mecanismos de exclusão mútua para habilitar ou desabilitar um evento. A Figura 10 exemplifica esta função (BRANDENBUG, 2007).

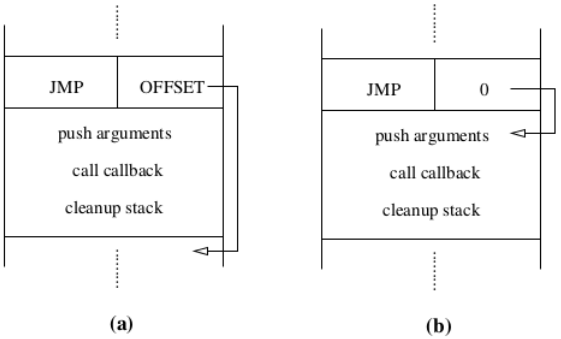

Figura 10 – (a) Habilitando e (b) Desabilitando um evento

4.1.3.2 Os eventos

Os eventos do Feather-trace estão diretamente relacionados às tarefas de escalonamento e exclusão mútua do sistema. Todos os eventos são declarados como macros que chama os eventos ft_event com uma função de *callback* que busca as informações de tempos. A seguir são exibidos os eventos e os locais no código do *kernel* do Litmus RT que estes são chamados.

SCHED_START, SCHED_END: chamadas no início e no fim da função *schedule()*, medindo assim a latência de execução da principal função de escalonamento do sistema;

CXS_START, CXS_END: chamados antes e depois de chamar a função *context_switch*, que faz a troca de contexto no Linux;

SCHED2_START(t),SCHED2_END(t): chamados em funções especificas do Litmus para a arquitetura ARM;

TICK_START(t), TICK_END(t): chamadas no inicio e fim da função *scheduler_tick()* do *kernel* do Linux;

PLUGIN_TICK_START, PLUGIN_TICK_END: semelhante ao trace TICK_START e STOP, porém, na função *litmus_tick()*, que faz parte das funções que gerenciam os escalonadores de tempo real específicos do Litmus RT;

PLUGIN_SCHED_START, PLUGIN_SCHED_END: chamado dentro da função *pick_next_task_litmus()*, que seleciona uma nova tarefa de tempo real do Litmus para ser executada, mede o tempo que a função *litmus_schedule()* demora para executar;

RELEASE_START, RELEASE_END: específico do Litmus RT, mede o tempo de enfileirar uma tarefa na fila de pronto;

LOCK_START, LOCK_END: chamada no inicio e no fim da função *sys_litmus_lock()*, mede o tempo que a função de *lock* do Litmus leva para obter um mutex de tempo real implementado para o Litmus RT. Um detalhe é que isto só vale para os mecanismos de exclusão mútua do Litmus RT e não do kernel do Linux;

UNLOCK_START, UNLOCK_END: mede o tempo para liberar um lock do Litmus RT, como o anterior, não mede os tempos dos *locks* do *kernel* do Linux.

LOCK_SUSPEND, LOCK_RESUME: mede o tempo em que uma tarefa suspende a espera de um *lock*, e retorna para adquirir o *lock*.

RELEASE_LATENCY: mede a latência do *release* de uma tarefa acordada por um *timer*, mas só para as tarefas do Litmus RT.

4.1.3.3 Buffer de log de tempos

Os eventos chamam funções de *callback* que registram o evento em um *buffer* de trace, este *buffer* foi desenvolvido para ser *lock-free* e *smp-safe*. O *buffer* aceita diversos produtores em paralelo, mas só um consumidor. Os produtores são as funções de *callback* e o consumidor do *driver* que

lê as informações e escreve para a aplicação que está lendo deste *driver*. A implementação deste *buffer* e do *driver* são descritas em (BRANDENBUG, 2007).

4.1.3.4 Sched_trace e Unit-Test

O sched_trace e o Unit-Test utilizam a infra-estrutura do Feather-trace para gerar *trace* da execução de tarefas em tempo real, automatizar testes e exibir gráficos de como o sistema escalonou as tarefas. O st_trace é um *script* que habilita eventos do Feather-trace através do utilitário ftcat, este executa um ftcat por CPU via os pseudo *drivers* /dev/litmus/sched_trace[CPU_INDEX], após terminar a execução o st_trace gera um arquivo de trace por CPU. Um exemplo da saída da execução do st_trace:

```
[root@litmus trace]# st_trace
CPU 0: 2365 > st--0.bin [0]
CPU 1: 2366 > st--1.bin [0]
CPU 2: 2367 > st--2.bin [0]
CPU 3: 2368 > st--3.bin [0]
Press Enter to end tracing...

Ending Trace...
Disabling 10 events.
Disabling 10 events.
Disabling 10 events.
Disabling 10 events.
/dev/litmus/sched_trace1: 9888 bytes read.
/dev/litmus/sched_trace0: 7704 bytes read.
/dev/litmus/sched_trace2: 8520 bytes read.
/dev/litmus/sched_trace3: 26808 bytes read.
```

Os arquivos de *trace* podem ser utilizados pelo Unit-Test para gerar saídas, textuais ou gráficas do escalonamento do sistema. Um exemplo de saída gráfica é dado na Figura 11 e a saída textual dos *trace* gerados na execução anterior é:

```
[root@vostro trace]# unit-trace -o *
[...]
Event ID: 8
Job: 2359.120
Type: switch_to
Time: 2209711732636
```

```
Event ID: 9
Job: 2353.123
Type: switch_to
Time: 2209717921320

Event ID: 10
Job: 2359.120
Type: switch_away
Time: 2209721117098
```

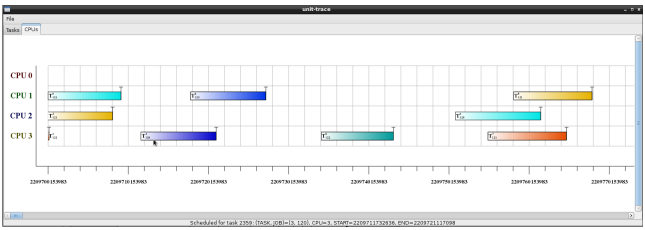

Figura 11 – Visualização dos eventos de escalonamento no unit-trace

4.1.4 Considerações

A forma de habilitar e desabilitar os eventos de *trace* e a independência de mecanismos de exclusão mútua são os pontos fortes do Feather-trace. Porém, os pontos de *trace* existentes atualmente são focados na execução do contexto do Litmus RT e não do sistema como um todo, assim, não é possível atualmente mapear as interferências e os mecanismos de bloqueio existentes no Linux. Para que isto fosse possível, seria necessário adaptar a ferramenta para adicionar novos pontos de *trace*.

4.2 *LINUX FUNCTION TRACER*: FTRACE

O Ftrace é a ferramenta oficial de *trace* e medição de latências do *kernel* do Linux. Uma das características que diferencia o Ftrace das demais ferramentas de *trace* vem do fato do Ftrace estar totalmente embutido no *kernel*, não precisando de ferramentas especias no espaço do usuário. As configurações do Ftrace são feitas através do *debugfs*, que é o sistema de arquivos depuração do *kernel* do Linux e podem ser feitas com os comandos echo e cat. A saída do *trace* é textual e de fácil entendimento (ROSTEDT, 2009). O Ftrace dá suporte a três formas de *trace*:

- Estático utilizando pontos de *trace* no *kernel*: *tracepoints* (ROSTEDT, 2010c);
- Estático na chamada e retorno de funções do *kernel*: *function tracer* (ROSTEDT, 2010a); e
- Dinâmico: *kprobe* (CORBET, 2009).

Apesar dos *tracepoints* e do *function tracer* serem métodos estáticos de depuração do *kernel*, sua ativação ou desativação é feita de maneira dinâmica. Quando não ativos, os pontos de *trace* são convertidos em operações causam pouco *overhead*: uma única operação (CORBET, 2010a). Estas duas características possibilitam que o Ftrace esteja sempre habilitado e sua utilização possa ser feita mesmo em sistemas embarcados, onde a sua utilização também é viável por não haver necessidade de ferramentas especias no espaço do usuário.

4.2.1 Componentes

O Ftrace é composto de:

- Uma interface de configuração: via *debugfs*;
- Um *buffer* para armazenamento de mensagens de depuração; e
- Formas de *trace*: *tracepoints*, *function_trace* e *kprobes*.

Apesar de possibilitar toda a sua utilização com a leitura e escrita de arquivos no *debugfs*, existem duas ferramentas no espaço do usuário que facilitam a utilização do Ftrace: o Trace-cmd e o Kernelshark.

O Trace-cmd é um programa em linha de comando que ativa funções do Ftrace e gera um arquivo de *trace* como saída, podendo também enviar os dados de *trace* para uma outra máquina na rede em tempo de depuração (ROSTEDT, 2010b). A saída do Trace-cmd pode ser vista de maneira textual ou servir de entrada para o Kernelshark.

O Kernelshark é uma ferramenta gráfica que possibilita a visualização e filtragem das informações de uma sessão de *trace*, facilitando assim a análise dos dados de *trace*. Além do *trace*, a ferramenta exibe um gráfico com a ocorrência dos eventos em cada CPU ou de cada tarefa do sistema. Neste gráfico é possível escolher qual o intervalo de tempo o usuário deseja observar, podendo assim visualizar melhor a ocorrência de eventos em um determinado intervalo de tempo (ROSTEDT, 2011). A Figura 12 mostra uma sessão de trace do Ftrace no Kernelshark.

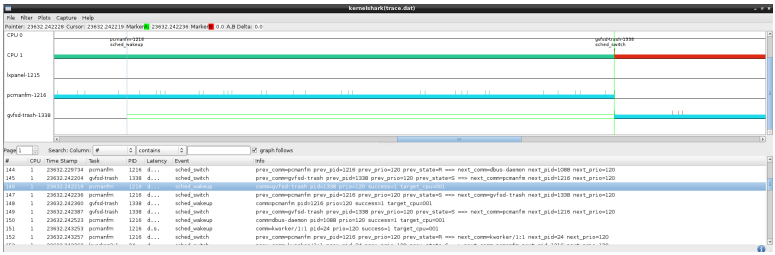

Figura 12 – Interface do Kernelshark

4.2.2 Utilização

Para utilizar o Ftrace é necessário acessar o *debugfs*. Por padrão este diretório é montado dentro do diretório /sys/kernel/debug/ e já vem montado por padrão na maioria dos sistemas para *desktop* e servidores. Caso este não esteja montado, o comando a seguir irá monta-lo no diretório padrão (ROSTEDT, 2009).

```
# mount -t debugfs debugfs /sys/kernel/debug/
```

Dentro do *debugfs* existe o diretório *tracing*, que é a raiz do Ftrace, um exemplo de saída do comando 'ls':

```
[root@prt tracing]# ls
available_events             set_ftrace_filter
available_filter_functions   set_ftrace_notrace
available_tracers            set_ftrace_pid
buffer_size_kb               set_graph_function
buffer_total_size_kb         stack_max_size
current_tracer               stack_trace
dyn_ftrace_total_info        stack_trace_filter
enabled_functions            trace
events                       trace_clock
free_buffer                  trace_marker
function_profile_enabled     trace_options
kprobe_events                trace_pipe
kprobe_profile               trace_stat
latency_hist                 tracing_cpumask
options                      tracing_enabled
per_cpu                      tracing_max_latency
printk_formats               tracing_on
README                       tracing_thresh
saved_cmdlines               uprobe_events
```

```
set_event                    uprobe_profile
```

Estes arquivos são a interface do Ftrace. Neles pode-se configurar e ler a saída dos *traces* executados. Existem três formas de interagir via esta interface: via os *tracers* (algumas vezes chamados de *plugins*) do Ftrace; via *tracepoints* que são os pontos de *trace* estáticos do *kernel* e via *kprobes*, que são eventos dinâmicos do *kernel*. Apesar de ser mais flexível, os eventos dinâmicos do *kprobes* não serão tratados neste trabalho, tanto pelo fato de que as funções e pontos de trace desejados para este trabalho são estáticos, quanto por adicionarem um maior *overhead* no sistema.

Os *plugins* do Ftrace podem ser visualizados lendo o arquivo available_tracers, por exemplo. Os *plugins* habilitados por padrão no Fedora Linux podem ser vistos com o seguinte comando (ROSTEDT, 2010a):

```
[root@ftrace tracing]# cat available_tracers
blk function_graph wakeup_rt wakeup function nop
```

Destes *plugins*, o *function* é a base dos *plugins* do Ftrace, com este *plugin* é possível ver todas as chamadas de função executadas no *kernel*. Como exemplo de sua utilização:

```
[root@ftrace tracing]# echo function > current_tracer
[root@ftrace tracing]# echo 1 > tracing_on
[root@ftrace tracing]# echo 0 > tracing_on
[root@ftrace tracing]# cat trace
# tracer: function
#
#  TASK-PID    CPU#    TIMESTAMP  FUNCTION
#     | |       |          |         |
   bash-1454 [001]  6043.732925: selinux_inode_setattr  \
                                 <-security_inode_setattr
  flush-252  [000]  6043.732925: list_move <-queue_io
   bash-1454 [001]  6043.732925: dentry_has_perm        \
                                 <-selinux_inode_setattr
  flush-252  [000]  6043.732925: list_move <-queue_io
   bash-1454 [001]  6043.732925: inode_has_perm         \
                                 <-dentry_has_perm
  flush-252  [000]  6043.732926: list_move <-queue_io
```

É possível também ajustar filtros nas funções que serão exibidas. Por exemplo, é possível ajustar um filtro para registrar as funções do *kernel* chamadas por um determinado *PID* ou que seu nome pertença a algum padrão de nome de função, como *sched**. No exemplo a seguir o Ftrace é executado com o *plugin function* e irá registrar apenas as funções executadas dentro do contexto do processo da *shell* que está se executando os comandos (ROSTEDT, 2010a).

```
[root@ftrace tracing]# echo $$ > set_ftrace_pid
[root@ftrace tracing]# echo 1 > tracing_on
[root@ftrace tracing]# echo 0 > tracing_on
[root@ftrace tracing]# cat trace
# tracer: function
#
#  TASK-PID    CPU#    TIMESTAMP  FUNCTION
#     | |       |          |         |
   bash-1454  [000]  6883.227214: mutex_lock                    \
                                  <-unmap_mapping_range
   bash-1454  [000]  6883.227214: _cond_resched <-mutex_lock
   bash-1454  [000]  6883.227214: should_resched <-_cond_resched
   bash-1454  [000]  6883.227214: need_resched <-should_resched
   bash-1454  [000]  6883.227215: mutex_unlock                  \
                                  <-unmap_mapping_range
   bash-1454  [000]  6883.227215: truncate_inode_pages          \
                                  <-truncate_pagecache
   bash-1454  [000]  6883.227215: truncate_inode_pages_range    \
                                  <-truncate_inode_pages
```

No exemplo a seguir o *plugin function* é executado com o filtro de quais funções devem ser exibidas, neste exemplo apenas as que possuam o nome iniciado em *sched* (ROSTEDT, 2010a).

```
[root@ftrace tracing]# echo sched* > set_ftrace_filter
[root@ftrace tracing]# echo function > current_tracer
[root@ftrace tracing]# echo 1 > tracing_on
[root@ftrace tracing]# echo 0 > tracing_on
[root@ftrace tracing]# cat trace
# tracer: function
#
#  TASK-PID    CPU#    TIMESTAMP  FUNCTION
#     | |       |          |         |
   bash-1116  [001]  1392.665149: schedule_work                 \
                                  <-tty_flip_buffer_push
   bash-1116  [001]  1392.665156: schedule                      \
                                  <-sysret_careful
   sshd-1048  [001]  1392.665227: schedule_hrtimeout_range      \
                                  <-poll_schedule_timeout
   sshd-1048  [001]  1392.665227: schedule_hrtimeout_range_clock \
                                  <-schedule_hrtimeout_range
   sshd-1048  [001]  1392.665227: schedule                      \
                                  <-schedule_hrtimeout_range_clock
   bash-1116  [001]  1392.665235: schedule_timeout              \
                                  <-n_tty_read
   bash-1116  [001]  1392.665235: schedule <-schedule_timeout
```

Além dos *plugins* é possível habilitar os eventos de *tracepoint* do kernel. Pode-se ver todos os eventos disponíveis em um sistema lendo o arquivo available_events. No exemplo a seguir são exibidos os eventos relacionados a escalonamento do sistema:

```
[root@ftrace tracing]# cat available_events  | grep sched:
sched:sched_kthread_stop
sched:sched_kthread_stop_ret
sched:sched_wakeup
sched:sched_wakeup_new
sched:sched_switch
sched:sched_migrate_task
sched:sched_process_free
sched:sched_process_exit
sched:sched_wait_task
sched:sched_process_wait
sched:sched_process_fork
sched:sched_stat_wait
sched:sched_stat_sleep
sched:sched_stat_iowait
sched:sched_stat_runtime
sched:sched_pi_setprio
```

Como exemplo de uso dos *tracepoints*, foram selecionados todos os eventos de escalonamento do sistema, resultando na seguinte saída:

```
[root@ftrace tracing]# cat available_events  | grep sched: \
                                                 > set_event
[root@ftrace tracing]# cat trace
# tracer: nop
#
#  TASK-PID    CPU#    TIMESTAMP  FUNCTION
#     | |       |          |         |
   sshd-1048  [000]  2591.374843: sched_stat_runtime:            \
                        comm=sshd pid=1048 runtime=72168 [ns]    \
                        vruntime=46326966 [ns]
   sshd-1048  [000]  2591.374845: sched_stat_wait:               \
                        comm=bash pid=1116 delay=81707 [ns]
   sshd-1048  [000]  2591.374846: sched_switch:                  \
                        prev_comm=sshd prev_pid=1048             \
                        prev_prio=120 prev_state=S ==>           \
                        next_comm=bash next_pid=1116             \
                        next_prio=120
   bash-1116  [000]  2591.374854: sched_stat_runtime: comm=bash  \
                        pid=1116 runtime=10911 [ns]              \
```

```
                        vruntime=802223154 [ns]
  bash-1116  [000]  2591.374858: sched_switch: prev_comm=bash   \
                        prev_pid=1116 prev_prio=120             \
                        prev_state=S ==> next_comm=swapper/0     \
                        next_pid=0 next_prio=120
<idle>-0     [001]  2591.404797: sched_migrate_task:           \
                        comm=sshd pid=1048 prio=120 orig_cpu=0  \
                        dest_cpu=1
```

O Ftrace possui muitas possíveis configurações, principalmente no que diz respeito aos filtros, sendo que a documentação do Ftrace na documentação do kernel aborda de maneira bastante extensa as opções do Ftrace.

4.2.3 Como Ftrace foi desenvolvido

O Ftrace um conjunto de ferramentas de *trace* que foram unificadas em uma única interface de configuração. A interface com o usuário é feita através de pseudo arquivos do *debugfs*, esta é uma interface padrão no *kernel* do Linux, onde é possível criar arquivos em memória. Estes arquivos, quando lidos ou escritos, ativam funções do *kernel* que podem acessar dados no *kernel* e trocar informações com o espaço do usuário. Via esta interface é possível utilizar os *tracepoints* e os *plugins* do Ftrace.

Os *tracepoints* e os *plugins* baseados no *function tracer* são métodos estáticos de *trace*, isto é, seus códigos de execução são adicionados ao binário do *kernel* em tempo de compilação, deste modo, estes sempre estão presentes no *kernel*. Para viabilizar a sua utilização, foi necessária a criação de métodos que, quando os *tracers* não estivessem habilitados, causassem pouco ou nenhum *overhead*. Este objetivo foi alcançado nos dois métodos utilizando técnicas de alteração do código de execução do *kernel* em tempo de execução. Apesar destas técnicas modificarem o *kernel* em tempo de execução, ambas fazem de maneira diferente.

Estes dois modos de trace são descritos a seguir, detalhando o seu método de ativação, de trace e interpretação dos dados, começando pelo *function trace*.

4.2.3.1 Function *trace*

O *function tracer* se baseou na forma com que o Gcc e a ferramenta Gprof interagem para fazer *profilling* de aplicações no espaço do usuário. Quando uma aplicação é compilada com a opção -pg do Gcc, em cada cha-

mada de função é adicionado uma chamada para a função *mcount* logo no inicio da função. Esta função *mcount* será chamada com dois argumentos, o endereço da função que chamou *mcount* e o endereço de retorno da função que chamou *mcount*. Por exemplo se a função *foo()* chamou a função *bar()* que chamou *mcount*, o primeiro argumento será o endereço da função *bar()* e o segundo argumento o endereço de retorno da função *bar()* na função *foo()*.

4.2.3.2 Ativando e desativando os *plugins*

Como pode-se supor, a chamada da função *mcount* em cada chamada de função dentro do kernel iria adicionar um *overhead* desnecessário na execução do sistema quando o *trace* não estivesse habilitado, porém, resolveu-se este problema utilizando relocação de código em tempo de execução.

Após a compilação dos fontes, mas antes de *link* dos objetos do kernel, o *script* recordmcount.pl do *kernel* do Linux analisa os códigos objetos e armazena em uma lista o endereço de todas as chamadas da função *mcount* presentes no *kernel*, esta lista é então armazenada na seção de dados do kernel. Durante a inicialização do sistema o *kernel* irá percorrer toda esta lista e alterar o código das operações que chamam a função *mcount* para a operação *nop*, deste modo, quando não habilitado, as chamadas das funções sofrerão um *overhead* mínimo. Além de diminuir o *overhead* esta técnica possibilita que os pontos de *trace* sejam habilitados em funções específicas do *kernel*, isto é que possibilita a filtragem de quais as funções do *kernel* serão exibidas no trace.

Para habilitar o *trace* de funções a operação inversa é feita, a lista é percorrida e as operações *nop* são substituídas para a chamada de função *mcount*, que por sua vez pode chamar uma função de *callback* do *plugin* habilitado do Ftrace. Abaixo o exemplo de um código em linguagem Assembly de uma chamada de função, primeiro com a chamada para *nop*.

```
55 push  \%rbp
48 89 e5  mov  \%rsp,\%rbp
0f 1f 44 00 00 nop (5 bytes)
65 48 8b 04 25 80 c8 mov  \%gs:0xc8800, \%rax
```

E após mudar o código para a chamada *mcount*:

```
55 push  \%rbp
48 89 e5  mov  \%rsp,\%rbp
e8 af 71 00 00 calq <mcout>
65 48 8b 04 25 80 c8 mov  \%gs:0xc8800, \%rax
```

Um detalhe é que esta modificação de código deve ser feita de modo que se garanta que o código alterado não esteja em execução em outra CPU. Atualmente a execução do código que altera o código é feita dentro de uma chamada *stop_machine*. A *stop_machine* é uma função do *kernel* que desabilita a preempção e as interrupções de todas as CPUs, para de executar tarefas nestas CPUs e executa a função passada como parâmetro em uma única CPU, deste modo, simulando um sistema *single-core* e garantindo que o código não está executando nas outras CPUs. Apesar de funcional, isto adiciona uma grande latência no sistema, visto que para habilitar/desabilitar o Ftrace é necessário parar todo sistema.

Quando chamada, a função que implementa um *plugin* do Ftrace irá ser chamada com os argumentos do *mcount* e, a partir disto, esta função irá colher os dados que esta deseja, alocar um espaço no *buffer* do Ftrace e salvar os dados coletados neste *buffer* para serem exibidos quando o *trace* for lido.

4.2.3.3 Tracer

Os *tracers* do Ftrace são definidos pela estrutura *struct tracer* em *kernel/trace/trace.h*. Esta estrutura possui um conjunto de ponteiros para as funções que são responsáveis por realizar as funções do *tracer*, como: inicializar e registrar o *tracer*, habilitar e desabilitar o trace, coletar os dados etc. Por exemplo, a seguir é exibida a estrutura que define do *plugin function* do Ftrace.

```
static struct tracer function_trace __read_mostly =
{
        .name           = "function",
        .init           = function_trace_init,
        .reset          = function_trace_reset,
        .start          = function_trace_start,
        .wait_pipe      = poll_wait_pipe,
        .flags          = &func_flags,
        .set_flag       = func_set_flag,
#ifdef CONFIG_FTRACE_SELFTEST
        .selftest       = trace_selftest_startup_function,
#endif
};
```

Esta estrutura de dados é passada para a função que registra o *trace* no Ftrace. Quando o *function trace* é acionado e ativado, a função *function_trace_init* é chamada. Esta função irá percorrer todos as chamadas *mcount* substituindo o seu código por uma função interna do Ftrace, que tem como

objetivo preparar os dados para chamar a função implementada pelo *function trace*, a função *trace_function*, que irá coletar as informações desejadas e armazenar os dados do trace no *buffer*. A seguir é exibida a função *trace_function*.

```
void
trace_function(struct trace_array *tr,
               unsigned long ip, unsigned long parent_ip,
               unsigned long flags,
               int pc)
{
        struct ftrace_event_call *call = &event_function;
        struct ring_buffer *buffer = tr->buffer;
        struct ring_buffer_event *event;
        struct ftrace_entry *entry;

        /* If we are reading the ring buffer, don't trace */
        if (unlikely(__this_cpu_read(ftrace_cpu_disabled)))
                return;

        event = trace_buffer_lock_reserve(buffer, TRACE_FN,
                                          sizeof(*entry), flags, pc);
        if (!event)
                return;
        entry   = ring_buffer_event_data(event);
        entry->ip                       = ip;
        entry->parent_ip                = parent_ip;

        if (!filter_check_discard(call, entry, buffer, event))
                ring_buffer_unlock_commit(buffer, event);
}
```

Como pode-se observar, esta função recebe um conjunto de argumentos. O primeiro fornece as informações da sessão corrente de *trace* para esta CPU, como por exemplo, o *buffer* para armazenar os dados. O segundo e o terceiro argumentos são os endereços das funções que devem ser armazenados, os dois últimos são os estados das IRQs e do contador de preempção. A função então aloca uma entrada no *buffer* do tipo *TRACE_FN*, os endereços das funções são armazenados nesta entrada e, por fim, os dados são salvos no *buffer*.

4.2.3.4 Tracepoints

Antes da criação dos *tracepoints* havia um problema no *kernel*: o grande número de formas de se fazer trace estático em pontos específicos do *kernel*. Alguns desenvolvedores implementavam suas próprias funções, outros usavam *printk*, outros ainda criavam seus próprios métodos, porém, não havia uma API unificada, que pudesse fazer o *trace* de maneira eficiente, tanto para sua ativação quanto para a leitura dos dados gerados (ROSTEDT, 2010c).

Os objetivos a serem alcançados pelos *tracepoints* eram:

- Poder ser utilizado em qualquer ponto do *kernel*;
- Baixo *overhead*: Na decisão de executar ou não, principalmente quando não executá-lo;
- Armazenar os dados de maneira eficiente; e
- Converter dados crus em informações de maneira intuitiva.

Para facilitar a utilização e adoção dos *tracepoins* um conjunto de macros foram desenvolvidas, estas macros fazem o papel de gerador automático de código, o que facilitou a adoção e utilização em massa dos *tracepoints*. Em sua função mais básica, um *tracepoint* pode ser gerado utilizando a macro *TRACE_EVENT* (ROSTEDT, 2010c):

```
TRACE_EVENT(name, proto, args, struct, assign, print)
```

Os parâmetros passados para a macro são:

- *name*: O nome do *tracepoint* a ser criado;
- *prototipo*: os argumentos do protótipo da função de *trace*;
- *args*: os argumentos da função;
- *struct*: uma estrutura de dados para armazenar os dados passados no *tracepoint*;
- *assign*: como, em um estilo parecido com C, copiar os dados desejados dos dados passados para a função na estrutura definida na função; e
- *print*: o modo de imprimir os dados em uma maneira legível por humanos em ASCII.

Exceto o primeiro argumento, todos os outros são encapsulados em outras macros. Em um exemplo do código que define um *tracepoint* existente no *kernel* temos:

```
TRACE_EVENT(sched_switch,

    TP_PROTO(struct rq *rq, struct task_struct *prev,
           struct task_struct *next),

    TP_ARGS(rq, prev, next),

    TP_STRUCT__entry(
           __array( char,prev_comm,TASK_COMM_LEN )
           __field( pid_t,prev_pid )
           __field( int,prev_prio )
           __field( long,prev_state )
           __array( char,next_comm,TASK_COMM_LEN )
           __field( pid_t,next_pid )
           __field( int,next_prio )
    ),

    TP_fast_assign(
           memcpy(__entry->next_comm, next->comm,
                  TASK_COMM_LEN);
           __entry->prev_pid = prev->pid;
           __entry->prev_prio = prev->prio;
           __entry->prev_state = prev->state;
           memcpy(__entry->prev_comm, prev->comm,
                  TASK_COMM_LEN);
           __entry->next_pid = next->pid;
           __entry->next_prio = next->prio;
    ),

    TP_printk("prev_comm=%s prev_pid=%d prev_prio=%d
               prev_state=%s ==> next_comm=%s
               next_pid=%d next_prio=%d",
           __entry->prev_comm, __entry->prev_pid,
           __entry->prev_prio,
           __entry->prev_state ?
           __print_flags(__entry->prev_state, "|",
                  { 1, "S"} , { 2, "D" }, { 4, "T" },
                  { 8, "t" }, { 16, "Z" }, { 32, "X" },
                  { 64, "x" }, { 128, "W" }) : "R",
           __entry->next_comm, __entry->next_pid,
           __entry->next_prio)
```

```
);
```

O evento demonstrado é executado a cada troca de contexto de *threads* no sistema, o evento é denominado *sched_switch*. A função que este evento implementa recebe três argumentos, o protótipo da função é definido pela macro *TP_PROTO* e o nome dos argumentos da função definidos pela macro *TP_ARGS*. A função implementada pelo *tracepoint* irá possuir uma estrutura definida pela macro *TP_STRUCT__entry* e atribuirá os dados dos argumentos à estrutura conforme definido pela macro *TP_fast_assign*. Estes dados serão armazenados no *buffer* do Ftrace, e quando lidos, serão interpretados pela macro *TP_printk* e impressos na saída o Ftrace.

4.2.3.5 Ativando os tracepoints

A forma de ativação dos *tracepoints* se dá de maneira diferente a do *function tracer*. Dependo do valor de uma variável o código do *trace* é executado ou não. Segundo (CORBET, 2010a), o código que decide se a função de um *tracepoint* irá ou não executar é semelhante ao seguinte exemplo:

```
static inline trace_foo(args)
{
    if (unlikely(trace_foo_enabled))
goto do_trace;
    return;
do_trace:
    /* Actually do tracing stuff */
}
```

Caso o valor da variável *trace_foo_enabled* seja verdadeiro, há um desvio condicional para o código que executa a função de *trace*, caso não, retorna para a função anterior. Vale notar que esta função é *inline* e que o dado é analisado como *unlikely*, sendo esta uma macro no *kernel* do Linux que é convertida em uma sinalização do compilador dizendo que o valor a ser testado muito provavelmente é falso, assim auxiliando o compilador e o processador no uso de *branch prediction*.

Apesar da execução do código estar otimizada, o acesso às variáveis faz com que haja possíveis acessos à memória externa, assim, prejudicando os mecanismos de *cache* do sistema. Por isto, a ativação do *tracepoint* é feita com o auxílio do *Jump Label*. O *Jump Label* é uma macro que em sua definição padrão se parece com desvio condicional da código anterior:

```
#define JUMP_LABEL(key, label)   \
```

```
if (unlikely(*key))          \
        goto label;          \
```

Para habilitar ou não um *tracepoint* com *Jump Label* é necessário chamar as funções *enable_jump_label (void *key)* e *disable_jump_label (void *key)*, estas funções ativam ou desativam o *tracepoint* alterando o valor da chave *key*. No caso padrão, a macro *JUMP_LABEL* se comporta de maneira análoga ao exemplo demonstrado (CORBET, 2010a), porém em algumas arquiteturas, como a Intel, as funções *enable_jump_label ()* e *disable_jump_label ()* atuam como o método utilizado pelo Ftrace: O desvio condicional não é adicionado ao código do *kernel*, somente o desvio condicional. A chave, o *label* e o endereço onde o seu *tracepoint* está no código do *kernel* do Linux são armazenados em uma tabela no sistema. Caso o *tracepoint* seja desabilitado, o endereço do *tracepoint* no código do *kernel* é procurado na tabela de chaves e alterado por operações *nop*. Caso seja habilitado, o código do desvio incondicional é adicionado no endereço do código do *kernel* e o *tracepoint* é habilitado. Deste modo, a opção de executar ou não um *tracepoint* se torna atômica e não há a necessidade de acesso externos à memória para decidir se o *tracepoint* será executado ou não (CORBET, 2010a). No caso de habilitar ou desabilitar o *tracepoint*, são enfrentados os mesmo problemas citados no caso do Ftrace, no que diz respeito ao alterar o código em tempo de execução, garantindo que outras CPUs não estão utilizando o código.

4.2.3.6 *Buffer* de dados

O *buffer* do Ftrace foi criado para suportar sistemas *multicore* com grande eficiência. Para isto, ele foi projetado para não conter áreas protegidas por *mutex* (*lockfree*), e ao invés de ser um *buffer* único, cada CPU possui um *buffer*, gerando assim menor contenção por coerência de *cache*. Diferente do *buffer* do Feather-trace, o *buffer* do Ftrace é fortemente ligado com a API do *kernel* do Linux. Se por um lado isto remove a portabilidade por outro adiciona características nativas do sistema como, por exemplo, aceitar a adição e remoção de CPUs ao sistema em sistemas *multicore* com suporte a *hotplug*, adicionando e removendo os *buffers* do *ftrace* destas CPUS em tempo de execução.

Para obter estas características, foi necessário adicionar restrições ao modelo do *ring-buffer*. O *ring-buffer* do Ftrace pode operar em dois modos: no modo produtor/consumidor ou sobrescrita, sendo que no primeiro, quando o *buffer* está cheio os dados mais novos são descartados, já no modo sobre escrita os dados mais antigos são descartados.

Como mencionado, cada CPU possui um *ring-buffer* formado por pá-

ginas ligadas como uma lista duplamente encadeada. Cada página aceita várias entradas de *trace*, estas podendo ser de tamanho variável. O *buffer* não aceita que vários escritores escrevam ao mesmo tempo na mesma CPU, porém, um escritor pode ser preemptado por outro, estas escritas devem atuar como uma pilha, por exemplo:

```
escritor1 inicia
   <preempted> escritor2 inicia
       <preempted> escritor3 inicia
       escritor3 finaliza
   escritor2 finaliza
escritor1 finaliza
```

Para exemplificar, isto pode ocorrer quando uma tarefa é interrompida por um tratador de interrupção, como as tarefas de escrita no *buffer* são feitas por *callbacks* do Ftrace, estas são desenvolvidas para não dormir, assim, satisfazendo o modelo. O *buffer* aceita múltiplos leitores/consumidores, porém, duas leituras não podem acontecer ao mesmo tempo, um leitor não pode interromper outro leitor e um leitor não pode interromper um escritor. Um leitor pode ler ao mesmo tempo que um escritor, porém, em CPUs diferentes, o escritor pode interromper um leitor.

Para desenvolver este *buffer* sem utilizar mecanismos de exclusão mutua, utilizou-se o próprio endereço da página como marcador para sinalizar se a escrita já terminou ou não. As páginas dos *buffers* são sempre alinhadas em 4 bytes, deste modo, os dois últimos bits do endereço da página do *buffer* serão sempre zeros, logo a operação:

```
address = address & ~3
```

Irá retornar o endereço da página, assumindo que os dois últimos endereços são 0, já a operação:

```
flags = address & 3
```

Irá retornar o estado da próxima página. Estes dois bits são utilizados para manter os seguintes estados:

- HEADER: é a próxima página a se tornar disponível para leitura;
- UPDATE: é a página que está para ser atualizada por uma escrita, e possivelmente o novo *header*.

Quando a escrita em uma página identifica que esta página está para se tornar o *HEADER*, o escritor irá atualizar o seu estado para *UPDATE* e apontará a próxima página como *HEADER*, após isto a página setada como *UPDATE* pode voltar ao seu estado normal. Quando um leitor vai ler a próxima

página do *buffer* este verifica se o ponteiro para a página do *HEADER* possui a *flag HEADER* ou *UPDATE*, caso possua a *UPDATE* o leitor irá ficar em espera ocupada até que a *flag UPDATE* seja alterada e assim poder seguir até o ponteiro que aponta para o *HEADER*.

Todas estas alterações de estados no endereço são feitas em uma operação atômica, por exemplo na arquitetura Intel com a instrução *cmpxchg(A, C, B)* que substitui o valor de A com B caso se A é igual a C.

4.2.3.7 Imprimindo os dados

Para cada *plugin* do Ftrace é definida uma estrutura *trace_event* que possui os dados: *type* que define qual o tipo de dados do *buffer* que estas funções esperam receber e um ponteiro para uma estrutura *trace_event_functions*, que possui o ponteiro para as funções que imprime o dados do *buffer* de determinado *plugin*. Abaixo o exemplo da estrutura que define o tipo de dado e a estrutura para o *plugin function*.

```
static struct trace_event trace_fn_event = {
        .type           = TRACE_FN,
        .funcs          = &trace_fn_funcs,
};
```

O ponteiro para função *funcs* aponta para a estrutura:

```
static struct trace_event_functions trace_fn_funcs = {
        .trace          = trace_fn_trace,
        .raw            = trace_fn_raw,
        .hex            = trace_fn_hex,
        .binary         = trace_fn_bin,
};
```

Que são as funções que imprimem os dados do trace, cada *plugin* pode implementar a saída de dados nos quatro possíveis formatos:

trace: dados formatados como definidos como na macro *TP_printk*;

raw: sem nenhuma formatação;

hex: em hexadecimal;

binary: em binário.

No formato *trace* os dados são formatados e exibidos em um formato intuitivo, por exemplo o endereço da função é traduzido no nome da função.

Porém, este modo aumenta o processamento dos dados feitos no *kernel*. Para diminuir este processamento, outros formatos são disponibilizados. Isto para que o processamento dos dados sejam feitos no espaço do usuário ou até mesmo em um outro computador. Não é obrigatória a implementação de todos os modos.

Em um exemplo de código, as duas funções abaixo implementam o parse dos dados do *buffer* quando estes são do *function tracer*. A função *trace_fn_trace* imprime os dados de maneira padrão, traduzindo os endereços das duas funções que foram passadas como argumento ao *plugin* de *trace* no nome das funções, e adicionando os campos de formatação. Já a função *trace_fn_raw* apenas imprime os endereços no formato hexadecimal. Ambas as funções são úteis, a primeira para análise de maneira facilitada e sem a necessidade de uma ferramenta externa e a segunda economizando processamento o que poderia ser útil para coletar *trace* em um sistema em produção e a formatação dos dados ser feita em um outro computador.

```
static enum print_line_t
trace_fn_trace(struct trace_iterator *iter,
                  int flags,struct trace_event *event)
{
        struct ftrace_entry *field;
        struct trace_seq *s = &iter->seq;

        trace_assign_type(field, iter->ent);

        if (!seq_print_ip_sym(s, field->ip, flags))
                goto partial;

        if ((flags & TRACE_ITER_PRINT_PARENT)
                                && field->parent_ip) {
                if (!trace_seq_printf(s, " <-"))
                        goto partial;
                if (!seq_print_ip_sym(s,
                                        field->parent_ip,
                                        flags))
                        goto partial;
        }
        if (!trace_seq_printf(s, "\n"))
                goto partial;

        return TRACE_TYPE_HANDLED;

 partial:
        return TRACE_TYPE_PARTIAL_LINE;
```

```
}

static enum print_line_t
trace_fn_raw(struct trace_iterator *iter,
                int flags, struct trace_event *event)
{
        struct ftrace_entry *field;

        trace_assign_type(field, iter->ent);

        if (!trace_seq_printf(&iter->seq, "%lx %lx\n",
                              field->ip,
                              field->parent_ip))
                return TRACE_TYPE_PARTIAL_LINE;

        return TRACE_TYPE_HANDLED;
}
```

Os modos de *trace* são habilitados dentro do diretório *options*, escrevendo '1' nos arquivos *bin*, *raw* ou *hex*, como no exemplo a seguir:

```
[root@vostro tracing]# echo 1 > options/raw
```

Deste modo, os traces serão formatados pela função *trace_fn_raw*, e não pela função *trace_fn_trace*, que é a padrão.

4.2.4 Considerações

O Ftrace é a ferramenta padrão de *trace* do Linux e apresenta diversas funcionalidades, como os *plugins* de *trace* ou apresentando uma interface, como o Kprobes. Outra característica vinda do fato de pertencer ao *kernel* é o suporte a diversas arquiteturas de *Hardware*, fácil instalação e constante atualização, com melhorias a cada lançamento do *kernel*.

Por ser implementada em grande parte no *kernel*, a sua utilização via comando simples facilita a utilização e integração com outras ferramentas, porém, adiciona um maior *overhead* nas tarefas do *kernel*, o que pode influenciar, principalmente em parâmetros temporais e na execução do sistema.

As ferramentas auxiliares, como o Kernelshark são de simples utilização e deixam bastante claros os acontecimentos no *kernel*, o que também é um ponto forte do Ftrace.

4.3 LTTNG

O projeto do LTTng busca prover um conjunto de ferramentas de *trace* que auxiliem na depuração de processos no Linux. Principalmente no que diz respeito a problemas de desempenho e depuração de sistemas com múltiplas *threads*. Também é possível fazer o *trace* de sistemas distribuídos (DESNOYERS, 2012).

Para atingir este objetivo o sistema é composto de diversos componentes, como exibe o diagrama de blocos na Figura 13 (DESNOYERS JULIEN DESFOSSEZ, 2012):

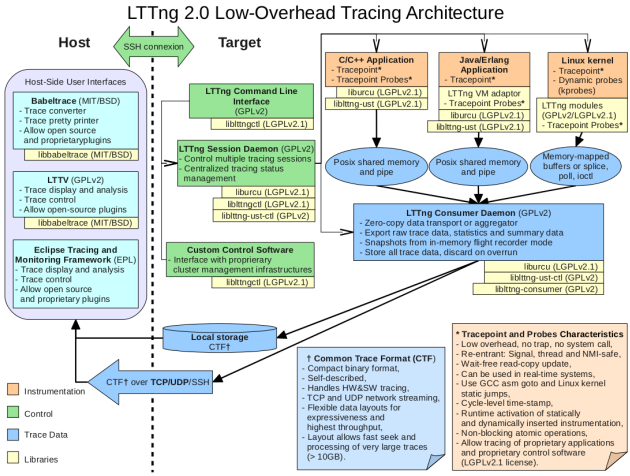

Figura 13 – Arquitetura do LTTng

Como pode-se observar, o sistema pode fazer *trace* tanto de aplicações no espaço do usuário quanto no espaço do *kernel*. Os *traces* são integrados via um *daemon* que coleta os dados e produz um arquivo de *trace* em um formato padronizado, que pode servir de entrada para aplicações de análise tanto em modo texto quando em modo gráfico. No diagrama de blocos pode-se ver que o sistema está sub-dividido em:

- Ferramentas de instrumentação;
- Ferramentas de controle;
- Ferramentas de coleta e análise de dados; e
- Bibliotecas compartilhadas.

Este conjunto de componentes cria uma ferramenta que é capaz de fazer *trace* de todo o ambiente de execução do Linux - espaço do *kernel* e espaço do usuário. A próxima seção descreve estes componentes, ferramentas e bibliotecas, com mais detalhes.

4.3.1 Componentes

As ferramentas de instrumentação são os meios de se adicionar pontos estáticos de trace nas aplicações, sendo o *kernel* do Linux uma destas aplicações. No diagrama exibe-se o suporte a *tracepoints* no *kernel* do Linux, em aplicações em C/C++ e em Java. Além dos *tracepoints*, e não exibido neste diagrama, o LTTng também pode servir de interface para outros mecanismos de *trace* do *kernel* do Linux, como os *plugins* de trace do Ftrace e os contadores de desempenho do Perf.

As ferramentas de controle servem de interface para a configuração das sessões de *trace*, controlando quais os *tracepoints* estão ativos e configurando o *daemon* coleta de dados de *trace* conforme o sistema está configurado.

A ferramenta de coleta dos dados de *trace* é responsável por coletar, armazenar e exportar os dados de uma maneira menos impactante possível, tanto no desempenho quanto nas caraterísticas temporais do sistema. Diferente do Ftrace e do Feather-trace que implementam os *buffers* de *trace* para uma única sessão, a ferramenta de coleta de *trace* do LTTng possibilita a criação de sessões de *trace*, as quais também podem possuir diversos canais, permitindo múltiplas sessões concorrentes de *trace*, possibilitando também que eventos diferentes sejam ativados simultaneamente por diversos usuários, podendo ser estes usuários os usuários normais do sistema e não somente o super-usuário (root).

Os *traces* do LTTng obedecem o padrão CTF. O CTF (*Commom Trace Format*) é um padrão especificado por diversas entidades relacionadas a *trace* de sistemas, principalmente sistemas *multicore*. Entre as empresas envolvidas na criação deste padrão pode-se citar a Siemens, Windriver, Broadcom, Freescale e a Efficios, empresa criada pelo criador do projeto LTTng (EFFICIOS, 2012).

Com este formato padronizado, diversas ferramentas poderão consumir os *traces* do LTTng. Atualmente é possível que estes *traces* sejam analisados após o fim da sessão, porém, há a intenção do projeto de que seja possível a análise dos *traces* em tempo de execução. Atualmente estes dados podem ser observados com o *Babeltrace*, que produz uma saída em texto. Existem também duas interfaces gráficas: primeira delas é o LTTV Viewer,

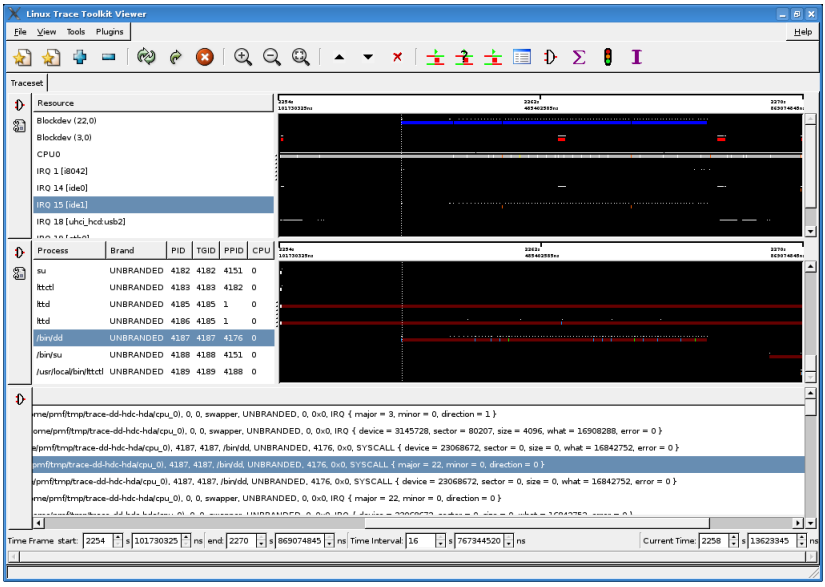

Figura 14 – LTTV

esta interface dá uma visão bastante intuitiva da execução dos pontos de *trace* do sistema, a Figura 14 é um *screenshot* de demonstração desta ferramenta. Esta ferramenta funciona apenas com o formato de *trace* antigo do LTTng, porém o desenvolvimento do suporte ao novo formato está em andamento.

A outra interface é o Eclipse Viewer, esta interface é um *plugin* para o Eclipse que possibilita a navegação no *trace* produzido pelo LTTng, a Figura 15 é um exemplo desta interface.

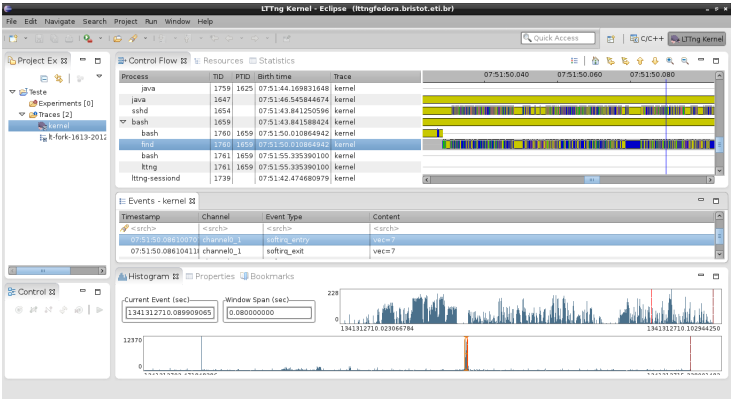

Figura 15 – ECLIPSE LTTng Plugin

Por fim, o LTTng compartilha uma série de bibliotecas entre as ferramentas, estas bibliotecas são:

- *liblttngctl*: implementa os comandos de controle da ferramenta;
- *liburcu*: uma implementação no espaço do usuário da RCU;
- *liblttng-ust-ctl*: implementa os comandos de controle da ferramenta para a *liblttng-usts*;
- *liblttng-ust*: biblioteca que implementa os *tracepoints* para C o espaço do usuário;
- *liblttng-consumer*: biblioteca que implementa a coleta de dados do Consumer *Daemon*; e
- *libbabeltrace*: implementa as principais funções de interpretação dos traces do LTTng.

4.3.2 Utilização

A maneira mais simples de interagir com o LTTng é via a interface de comandos, utilizando o comando lttng. Em um exemplo bastante simples:

```
[root@lttngfedora ~]# lttng create kernel_event
Session kernel_event created.
Traces will be written in                          \
/root/lttng-traces/kernel_event-20120704-060517
[root@lttngfedora ~]# lttng enable-event -a -k
All kernel events are enabled in channel channel0
[root@lttngfedora ~]# lttng start
Tracing started for session kernel_event
[root@lttngfedora ~]# find /usr/ > /dev/null
^C
[root@lttngfedora ~]# lttng stop
Tracing stopped for session kernel_event
[root@lttngfedora ~]# lttng destroy
Session kernel_event destroyed at /root
```

O primeiro comando cria uma nova sessão do LTTng denominada *kernel_event* qual terá seus *traces* armazenados em */root/lttng-traces/kernel_event-20120704-060517*. Um único usuário pode ter diversas sessões, que podem

[0] Basicamente, o que é RCU?: http://lwn.net/Articles/262464/

ser sub-divididas em canais. O segundo comando habilita todos os eventos (parâmetro -a) do *kernel* (parâmetro -k). O terceiro comando inicia a sessão de *trace* e o quarto comando é apenas uma forma de utilizar o sistema para passar por pontos de *trace*. Por fim, o quinto comando para o *trace* em andamento e o sexto destrói a sessão. Todo usuário pode ter uma sessão de *trace*, até de maneira concorrente, porém, somente o usuário *root* e os usuários do grupo *tracing* podem fazer trace de eventos do *kernel*, os demais somente de eventos no espaço do usuário.

Para ver a saída do *trace* do LTTng, pode-se utilizar o Babeltrace, como no exemplo a seguir:

```
[root@lttngfedora ~]# babeltrace                          \
  /root/lttng-traces/kernel_event-20120704-060517/kernel/
[06:05:59.686050289] (+?.??????????) exit_syscall:        \
     { cpu_id = 1 }, { ret = 0 }
[06:05:59.686053139] (+0.000002850) sys_mprotect:         \
     { cpu_id = 1 }, { start = 140434022072320,           \
     len = 135168, prot = 3 }
[06:05:59.686057727] (+0.000004588) exit_syscall:         \
     { cpu_id = 1 }, { ret = 0 }
[06:05:59.686076600] (+0.000018873) sys_poll:             \
     { cpu_id = 1 }, { ufds = 0x7FB966352C30, nfds = 2,   \
     timeout_msecs = -1 }
[06:05:59.686081251] (+0.000004651) exit_syscall:         \
     { cpu_id = 1 }, { ret = 1 }
[06:05:59.686082658] (+0.000001407) sys_recvmsg:          \
     { cpu_id = 1 }, { fd = 17, msg = 0x7FB966351B40,     \
     flags = 256 }
[06:05:59.686087854] (+0.000005196) exit_syscall:         \
     { cpu_id = 1 }, { ret = 4136 }
[06:05:59.686089030] (+0.000001176) sys_poll:             \
     { cpu_id = 1 }, { ufds = 0x7FB966352C30, nfds = 2,   \
     timeout_msecs = -1 }
[06:05:59.686091448] (+0.000002418) exit_syscall:         \
     { cpu_id = 1 }, { ret = 1 }
[06:05:59.686092553] (+0.000001105) sys_recvmsg:          \
     { cpu_id = 1 }, { fd = 17, msg = 0x7FB966351B10,     \
      flags = 0 }
```

Para ver este trace utilizando a interface do Eclipse, é necessário criar um novo projeto do Eclipse. Este projeto deve ser do tipo ”Tracing Project”, como na Figura 16:

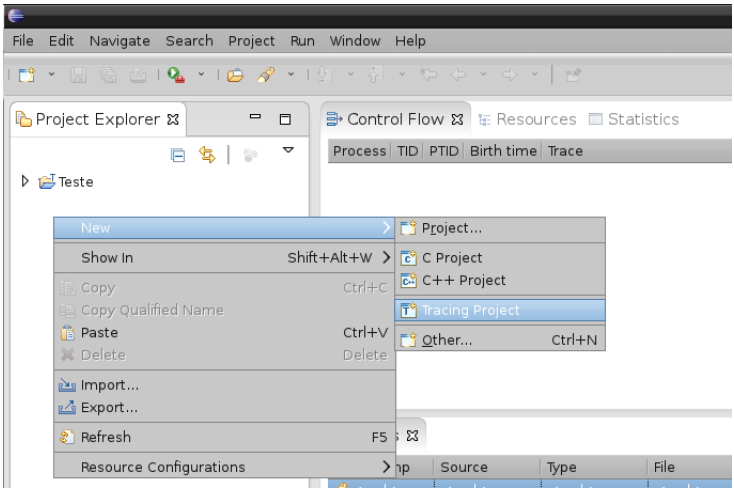

Figura 16 – Criando um projeto no Eclipse

Após isto, o usuário deve-se selecionar a opção para importar um *trace* e abrir o diretório que o evento foi criado, como no exemplo da Figura 17:

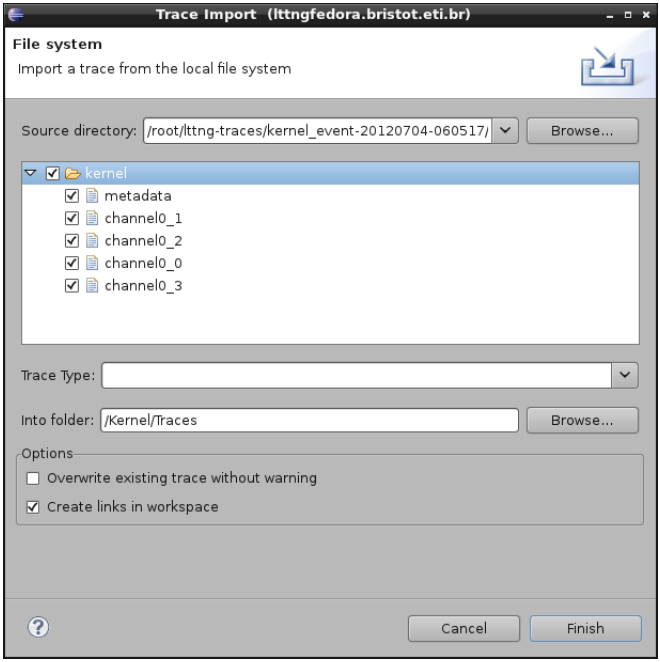

Figura 17 – Importando um projeto

Após isto é necessário informar que o *trace* é um *trace* do kernel, como na Figura 18, e abrir a perspectiva de *trace* no menu *Window*, opção *Open Perspective*.

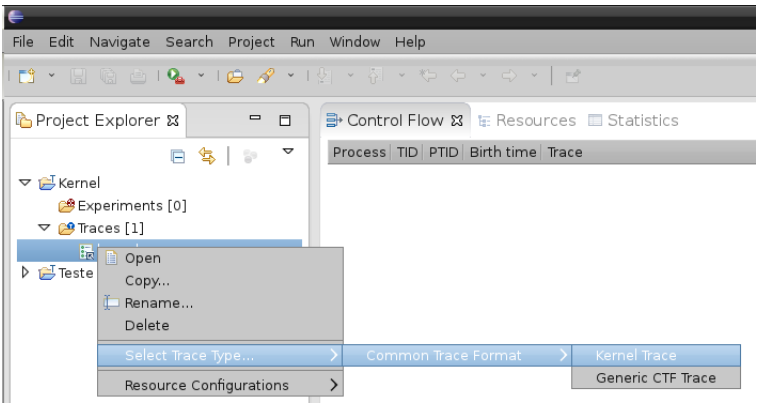

Figura 18 – Selecionando tipo de *trace*

A partir disto é possível navegar no *trace*. A interface do LTTng no Eclipse é bastante completa e intuitiva. Porém, para usar a interface de maneira mais confortável, é necessário um monitor com resolução grande. Em testes utilizando um monitor de 15.6"com resolução HD a tela ficou bastante carregada, para utilizar a ferramenta de maneira confortável foi necessário utilizar um monitor de 22 polegadas com resolução FULL HD. Figura 19 mostra a perspectiva de *trace* do *trace* executado anteriormente.

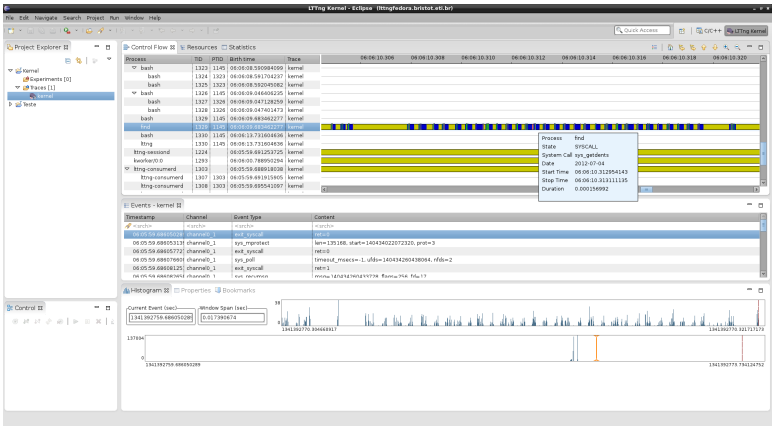

Figura 19 – Explorando o *trace* de exemplo

Apesar de utilizar os *tracepoints* do kernel o LTTng re-escreve os *tracepoints*. Para ver os *tracepoints* disponíveis para o LTTng pode-se utilizar o comando ”lttng list -k”, como no exemplo resumido a seguir:

```
[root@lttngfedora ~]# lttng list -k
Kernel events:
-------------
      timer_init (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      timer_start (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      timer_expire_entry (loglevel: TRACE_EMERG (0)) \
                         (type: tracepoint)
      timer_expire_exit (loglevel: TRACE_EMERG (0)) \
                        (type: tracepoint)
      [...]
```

Também é possível criar uma sessão onde somente um *tracepoint* seja habilitado:

```
[root@lttngfedora ~]# lttng create kernel-sched
Session kernel-tp-timmer created.
Traces will be written in
       /root/lttng-traces/kernel-sched-20120704-073747
[root@lttngfedora ~]# lttng enable-event -k sched_wakeup
kernel event sched_wakeup created in channel channel0
[root@lttngfedora ~]# lttng start
Tracing started for session kernel-sched
[do somethings]
[root@lttngfedora ~]# lttng stop
Tracing stopped for session kernel-sched
[root@lttngfedora ~]# lttng destroy
Session kernel-sched destroyed at /root
```

Além dos eventos do *kernel* é possível gerar pontos de *trace* em aplicações no espaço do usuário. O pacote lttng-ust trás alguns exemplos de *tracepoints* no espaço do usuário, dentro do pacote existe o diretório *tests/hello* com a aplicação *hello.c* de exemplo. Um exemplo de uso de *trace* no espaço do usuário pode ser visto a seguir:

```
[daniel@lttngfedora tests]$ lttng create hellow
Session hellow created.
Traces will be written in
       /home/daniel/lttng-traces/hellow-20120704-080918
[daniel@lttngfedora tests]$ lttng enable-event -a -u
```

```
All UST events are enabled in channel channel0
[daniel@lttngfedora tests]$ lttng start
Tracing started for session hellow
[daniel@lttngfedora tests]$ ./hello/hello
Hello, World!
Tracing...  done.
[daniel@lttngfedora tests]$ lttng stop
Tracing stopped for session hellow
[daniel@lttngfedora tests]$ lttng destroy
Session hellow destroyed at /home/daniel
```

A saída do trace gerado nesta sessão foi:

```
[daniel@lttngfedora tests]$ babeltrace
       /home/daniel/lttng-traces/hellow-20120704-080918
[08:09:41.254591297] (+?.??????????) ust_tests_hello:tptest:      \
  { cpu_id = 1 }, { intfield = 0,  intfield2 = 0x0,               \
  longfield = 0, netintfield = 0, netintfieldhex = 0x0,           \
  arrfield1 = [ [0] = 1, [1] = 2, [2] = 3 ], arrfield2 = "test", \
 _seqfield1_length = 4, seqfield1 = [ [0] = 116, [1] = 101,       \
  [2] = 115, [3] = 116 ], _seqfield2_length = 4,                  \
  seqfield2 = "test", stringfield = "test",                       \
  floatfield = 2222, doublefield = 2 }

[08:09:42.254783405] (+1.000192108) ust_tests_hello:tptest:      \
  { cpu_id = 1 }, { intfield = 1, intfield2 = 0x1, longfield = 1,\
  netintfield = 1, netintfieldhex = 0x1, arrfield1 = [ [0] = 1,  \
  [1] = 2, [2] = 3 ], arrfield2 = "test", _seqfield1_length = 4, \
  seqfield1 = [ [0] = 116, [1] = 101, [2] = 115, [3] = 116 ],    \
  _seqfield2_length = 4, seqfield2 = "test",                     \
  stringfield = "test", floatfield = 2222, doublefield = 2 }

[08:09:43.255149251] (+1.000365846) ust_tests_hello:tptest:      \
  { cpu_id = 1 }, { intfield = 2, intfield2 = 0x2, longfield = 2,\
  netintfield = 2, netintfieldhex = 0x2, arrfield1 = [ [0] = 1,  \
  [1] = 2, [2] = 3 ], arrfield2 = "test", _seqfield1_length = 4, \
  seqfield1 = [ [0] = 116, [1] = 101, [2] = 115, [3] = 116 ],    \
  _seqfield2_length = 4, seqfield2 = "test",                     \
  stringfield = "test", floatfield = 2222, doublefield = 2 }
```

Como pode-se observar, todo este *trace* foi executado com um usuário comum do sistema (daniel), o que é uma vantagem do LTTng.

Além dos *tracepoints*, é possível habilitar outras funções do kernel do Linux, como habilitar que as entradas e saídas de função sejam pontos de

trace, usando a infraestrutura do Ftrace, ou adicionar contadores de hardware como de *cache miss*, utilizando a infraestrutura do Perf.

4.3.3 Como LTTng foi desenvolvido

Como pode-se observar, os métodos de *trace* disponibilizados pelo LTTng são os mesmos métodos disponibilizados pelo Ftrace: *tracepoints* e *trace* de funções. Na verdade os *tracepoints* tem origem no LTTng, já o trace das funções tem origem no Ftrace. A diferença entre o LTTng e o Ftrace não estão nas funções de *trace*, mas em como as sessões de *trace* são organizadas, os dados armazenados e transferidos entre o *kernel* e as ferramentas no espaço do usuário.

4.3.3.1 *Buffers*

Cada sessão do LTTng pode ter vários canais, um canal é uma ponte entre os produtores (quem escreve) e os consumidores (quem lê). O principal objetivo do canal é se comportar como um *buffer* para mover os dados de maneira mais eficiente. O canal consiste de um *buffer* por CPU, isto para garantir o melhor uso da *cache* local e evitar *overheads* oriundos da coerência de cache. Cada *buffer* é composto por vários *sub-buffers* que contém *slots* que são reservados sequencialmente. Um *slot* é uma região do *sub-buffer* que é reservada para a escrita de uma entrada do *trace*. A Figura 20 ajuda a exemplificar o uso dos *slots* e *buffers*.

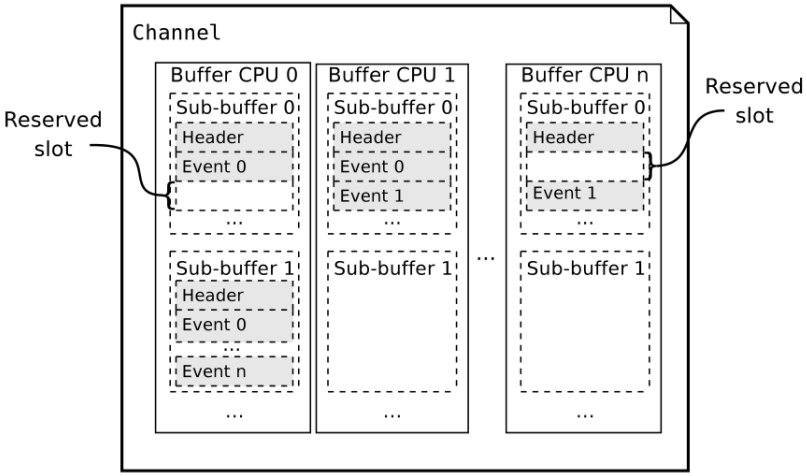

Figura 20 – *Slots*, *sub-buffer* e *buffers* do LTTng

No exemplo da Figura 20, no *buffer* da CPU 0, um *slot* está reservado logo após um *slot* onde o evento 0 já está completamente escrito. Na CPU *n*, o *slot* 0 foi reservado para escrita, mas a escrita ainda não foi confirmada - a escrita é confirmada com a operação de *commit*, porém o dado seguinte já foi escrito e confirmado. Isto pode acontecer quando, logo após uma tarefa reservar o *slot* 0, a tarefa foi interrompida por um tratador de interrupção, que executou, adicionou o *trace*, confirmou e terminou a execução, só então, deixando a tarefa anterior terminar de escrever e confirmar o dado no *slot*. O *sub-buffer* 0 da CPU 1 terminou de alocar todos os *slots* e está pronto para se lido. Para implementar este sistema de *buffer* sem a utilização de mecanismos de exclusão mútua do *kernel*, assim como o Ftrace, o LTTng utiliza os dois últimos bits do endereço do *buffer*, para sinalizar *slot* em uso e *slot* confirmado

4.3.3.2 Coleta de dados

Periodicamente um *timer* do sistema é acionado para verificar se existem *buffers* que podem ser lidos pelo *daemon* que coleta os dados do *kernel*. Se existe dados para serem coletados, a tarefa é acordada.

A tarefa no espaço do usuário - o Consumer mon, se comunica com o *kernel* através do pseudo-arquivo */proc/lttng*. Quando uma sessão é criada o *daemon* lttng-consumerd utiliza este arquivo para criar uma sessão e um

canal. Após isto o Consumer *Daemon* abre uma comunicação com o *kernel* via um *pipe*, utilizando o */proc/lttng* como entrada. Neste *pipe* o Consumer *Daemon* fica esperando dados de maneira assíncrona utilizando a chamada *epoll*. Quando um dado é recebido, o Consumer *Daemon* utiliza IOCTLs para coordenar as ações de copia de dados com o *kernel* e salva os dados utilizando a chamada *splice*, que copia os dados do *buffer* direto para um arquivo ou para a rede, sem ter que copiar os dados do espaço de endereço do kernel para o espaço do usuário, diminuindo assim o *overhead* do sistema.

4.3.3.3 Reescrita dos métodos de *trace*

Apesar do LTTng utilizar os mesmos métodos de *trace* do Ftrace, todos os métodos são reescritos para, ao invés de utilizar os *buffers* e a interface de comunicação do Ftrace, utilizar os *buffers* e a interface de comunicação do LTTng. Por exemplo, para dar suporte ao Ftrace *function tracer*, o LTTng reescreveu a função chamada pelo Ftrace para coletar os dados de *trace*, alterando a forma de armazenamento dos dados coletados. Esta função foi redefinida como:

```
static void
lttng_ftrace_handler(unsigned long ip,
                 unsigned long parent_ip, void **data)
{
        struct lttng_event *event = *data;
        struct lttng_channel *chan = event->chan;
        struct lib_ring_buffer_ctx ctx;
        struct {
                unsigned long ip;
                unsigned long parent_ip;
        } payload;
        int ret;

        if (unlikely(!ACCESS_ONCE(chan->session->active)))
                return;
        if (unlikely(!ACCESS_ONCE(chan->enabled)))
                return;
        if (unlikely(!ACCESS_ONCE(event->enabled)))
                return;

        lib_ring_buffer_ctx_init(&ctx, chan->chan, event,
                                 sizeof(payload),
                                 lttng_alignof(payload), -1);
        ret = chan->ops->event_reserve(&ctx, event->id);
```

```
        if (ret < 0)
                return;
        payload.ip = ip;
        payload.parent_ip = parent_ip;
        lib_ring_buffer_align_ctx(&ctx, lttng_alignof(payload));
        chan->ops->event_write(&ctx, &payload, sizeof(payload));
        chan->ops->event_commit(&ctx);
        return;
}
```

Esta nova função utiliza estruturas de dados do LTTng que definem a sessão e o *payload* dos dados. Primeiro esta função checa se a sessão de *trace* está ativa, se estiver, ela inicializa e reserva um *slot* no *buffer*, ajusta os dados do *payload*, escreve e dá o *commit* dos dados no *buffer* do LTTng.

O mesmo acontece para os *tracepoints*. As definições das macros que definem os *tracepoints* são redefinidas. Isto para, ao invés de usar os métodos de armazenamento dos dados do Ftrace, usar os do LTTng. Isto é feito no *header lttng-modules/probes/lttng-events.h* do LTTng. Por exemplo, a definição de uma classe de eventos é reescrita como:

```
#define DECLARE_EVENT_CLASS(_name, _proto, _args, _tstruct,   \
                        _assign, _print)                      \
static void __event_probe__##_name(void *__data, _proto)      \
{                                                             \
        struct lttng_event *__event = __data;                 \
        struct lttng_channel *__chan = __event->chan;         \
        struct lib_ring_buffer_ctx __ctx;                     \
        size_t __event_len, __event_align;                    \
        size_t __dynamic_len_idx = 0;                         \
        size_t __dynamic_len                                  \
                  [ARRAY_SIZE(__event_fields___##_name)];     \
        struct __event_typemap__##_name __typemap;            \
        int __ret;                                            \
                                                              \
        if (0) {                                              \
                (void) __dynamic_len_idx;                     \
                                /* don't warn if unused */    \
                (void) __typemap;                             \
                                /* don't warn if unused */    \
        }                                                     \
        if (!_TP_SESSION_CHECK(session, __chan->session))     \
                return;                                       \
        if (unlikely(!ACCESS_ONCE(__chan->session->active)))  \
                return;                                       \
        if (unlikely(!ACCESS_ONCE(__chan->enabled)))          \
```

```
        return;                                                 \
    if (unlikely(!ACCESS_ONCE(__event->enabled)))               \
        return;                                                 \
    __event_len = __event_get_size__##_name(__dynamic_len,\
                                            _args);             \
    __event_align = __event_get_align__##_name(_args);          \
    lib_ring_buffer_ctx_init(&__ctx, __chan->chan,              \
                __event,__event_len, __event_align, -1); \
    __ret = __chan->ops->event_reserve(&__ctx,                  \
                __event->id);                                   \
    if (__ret < 0)                                              \
        return;                                                 \
    /* Control code (field ordering) */                         \
    _tstruct                                                    \
    __chan->ops->event_commit(&__ctx);                          \
    return;                                                     \
    /* Copy code, steered by control code */                    \
    _assign                                                     \
}
```

Como pode-se observar, a dinâmica desta função é a mesma da utilizada na função que reescreve o *function tracer*. Esta define as variáveis da sessão e de *payload* do LTTng, verifica se a sessão, o canal e o evento estão ativos e, caso sim, aloca, reserva, formata e dá o *commit* dos dados do *trace*.

Além destas definições, os métodos que copiam os dados dos *tracepoints* são redefinidos. Os *tracepoints* que o LTTng suporta são reescritos. Na versão utilizada neste trabalho, apenas alguns *tracepoints* são suportados. Por exemplo, o *tracepoint irq_handler_exit* é reescrito como:

```
TRACE_EVENT(irq_handler_exit,

    TP_PROTO(int irq, struct irqaction *action,
             int ret),

    TP_ARGS(irq, action, ret),

    TP_STRUCT__entry(
        __field(    int,    irq     )
        __field(    int,    ret     )
    ),

    TP_fast_assign(
        tp_assign(irq, irq)
        tp_assign(ret, ret)
    ),
```

```
        TP_printk("irq=%d ret=%s",
                  __entry->irq, __entry->ret ?
                 "handled" : "unhandled")
)
```

O que muda deste *tracepoint* para o do Ftrace é o modo que os dados são definidos na macro *TP_fast_assign*, que é a forma com que os dados coletados são atribuídos a valores que serão armazenados no *trace*.

Com a redefinição das macros e da forma com que os dados são atribuídos, é necessário que estes *tracepoints* sejam recompilados e que, quando ativos, sejam chamados no lugar dos *tracepoints* do Ftrace. Isto é feito com um módulo do LTTng, que é carregado quando uma sessão é criada.

4.3.4 Considerações

O LTTng é a ferramenta mais completa de *trace*. Possui suporte tanto a aplicações em espaço do usuário quanto no *kernel*, um formato de *trace* padronizado, interfaces gráficas completas e intuitivas e suporte a múltiplas sessões concorrentes.

Porém, por não fazer parte do *kernel* oficial do Linux, a adição de novos pontos de *trace* no LTTng é dificultada, pois é necessário adicionar um *tracepoint* no kernel, reescreve-lo no LTTng e editar as ferramentas que interpretam e exibem as saídas do *trace*.

4.4 COMPARATIVO ENTRE AS FERRAMENTAS

Durante o estudo das ferramentas de *trace* observou-se que, apesar das ferramentas analisadas serem descritas com algumas qualidades em comum, não existia a definição de um conjunto de métricas com as quais se pudesse avaliar e comparar estas ferramentas. Com base nas características que as ferramentas e nas suas funcionalidades, foram levantadas as seguintes métricas qualitativas:

- *Trace Dinâmico*: Capacidade da ferramenta fazer *trace* dinâmico;
- *Trace Estático*: Capacidade da ferramenta fazer *trace* estático;
- *Buffer*: Como é implementado o sistema de armazenamento de informações de trace;

- *API*: Interface de comunicação do sistema de *trace* com outros processos;
- *Ferramentas*: Conjunto de ferramentas para análise dos dados de *trace*;
- *Arquiteturas*: Suporte a arquiteturas de *hardware*;
- *Overhead*: *Overhead* causado pela execução dos pontos de *trace*;
- *Espaço do Usuário*: A capacidade da ferramenta fazer *trace* de aplicações no espaço do usuário;
- *Formato do Trace*: Formato de saída do *trace*;
- *Facilidade de extensão*: Facilidade de adicionar um novo ponto de *trace* no sistema.

Com o levantamento destas métricas qualitativas, foi possível conduzir uma análise comparativa entre as ferramentas de trace.

4.4.1 Análise comparativa das ferramentas de trace

Trace Dinâmico: O Feather-Trace não possui suporte a *trace* dinâmico. O Ftrace e o LTTng ao invés de implementar uma nova versão de *trace* dinâmico, dão interface para a utilização de Kprobes. Kprobes é o método de *trace* dinâmico do *kernel* do Linux.

Trace Estático: O Feather-Trace implementa os pontos de *trace* estático buscando, principalmente, o menor *overhead* na forma de habilitar e, quando não ativos, de não executar os pontos de *trace*. O Ftrace utiliza-se de duas formas de *trace*, o *trace* de funções e de pontos de *trace* no código do *kernel*. Estas duas opções estão disponíveis também para o LTTng, porém, ambas formas foram modificadas para utilizar o sistema de *buffers* do LTTng .

Buffers: Os três sistemas utilizam técnicas semelhantes. O sistema de *buffer* do Feather-Trace possui como objetivo o menor *overhead*, igualmente o Ftrace. Porém, o *buffer* do Ftrace, por ser mais extensivo e estar de acordo com os padrões e APIs do *kernel* do Linux, pode apresentar maior *overhead*, quando comparado ao *buffer* do Feather-Trace. O que ambos possuem de inferior ao LTTng é o fato de não possibilitarem sessões concorrentes de *trace*.

API: A API do Feather-Tracer é única e não integrada com o padrão de desenvolvimento do *kernel* do Linux. Esta API é implementada como um *driver* de caractere que serve de interface de comunicação com as ferramentas, o que não é muito comum nem aceito no *kernel* do Linux. A API do Ftrace é a API padrão do *kernel* do Linux. Esta se utiliza de um subsistema de *debug*

do *kernel*, além disto, todos os dados exportados pelo Ftrace possuem modos de exportação em formatos diversos, como formatados, em hexadecimal, em binário, ou em formato *raw* (crú). No formato *raw* existe um arquivo que descreve o que cada campo significa, determinando o tipo, tamanho, ordem de bit, etc. O LTTng reimplementa esta interface com o objetivo de causar menor *overhead* ao sistema, e implementa também bibliotecas que auxiliam no desenvolvimento de ferramentas que queiram utilizar esta API. Entretanto, esta API é justamente um dos motivos de o LTTng não ser parte integrante do *kernel*: os mantenedores do *kernel* alegam não ser necessário o desenvolvimento de uma nova API de *trace*, pois o kernel já possui uma, a do Ftrace.

Ferramentas: O Feather-Trace possui ferramentas que podem exibir os dados tanto em formato texto quando em formato gráfico, porém, a ferramenta gráfica é bastante simples e de difícil observação dos dados. O Ftrace, com o Trace-cmd e o Kernelshark, possibilita tanto *trace* remoto quanto a visualização dos dados de maneira bastante intuitiva, apresentando opções como filtro de eventos e redimensionamento do campo de observação dos eventos. As grandes vantagens do LTTng são o formato padrão de *trace* e o *plug-in* do Eclipse, que apresenta as informações de uma maneira bastante intuitiva e com a vantagem de integrar *traces* do *kernel* e espaço do usuário em uma única sessão de *trace*.

Arquiteturas: O Feather-Trace possui somente suporte às arquiteturas Intel e ARM. O Ftrace possibilita a extração dos dados utilizando ferramentas como o *cat* e *echo*. Além de executar na grande maioria das arquiteturas que o Linux é suportado, possibilita um uso completo da ferramenta mesmo em sistemas embarcados. Já o LTTng pode ser compilado para diversas arquiteturas, porém, suas ferramentas em espaço do usuário tornam esta tarefa um pouco mais complexa.

Overhead: O Feather-Trace buscou o ótimo no quesito *overhead*. O Ftrace e o LTTng, por utilizarem os mesmos mecanismos de *trace*, possuem características semelhantes. Porém, a interface de comunicação com o espaço do usuário do LTTng é mais eficiente, por utilizar mapeamento de memória ao invés de leitura em arquivos no *debugfs*.

Espaço do usuário: O Feather-Tracer pode ser estendido para ser utilizado no espaço do usuário, porém, este não faz parte da distribuição padrão do Litmus RT. O Ftrace não faz *trace* de aplicações no espaço do usuário, mas possui uma interface que possibilita adicionar marcações ao *trace* a partir de processos. Já o LTTng possibilita que o *trace* do *kernel* e de espaço do usuário sejam feitos de maneira integrada, e ainda possibilita diversas sessões concorrentes de *trace* no espaço do usuário.

Formato do trace: O Feather-Tracer possui um formato binário de *trace* que precisa ser convertido por ferramentas no espaço do usuário. Em

alguns casos, durante os experimentos, estas ferramentas apresentaram defeito por falta de sincronização nos eventos. O Ftrace pode exportar os dados de diversas maneiras, sendo que este pode ser interpretado no *kernel* ou no espaço do usuário. O LTTng precisa de ferramentas no espaço do usuário para formatar os dados. A vantagem do LTTng sobre as outras é que este formato foi padronizado por um consórcio de empresas que definiram um padrão para formato de *trace*, chamado CTF.

Facilidade de extensão: Tanto os *tracepoints* utilizados no Ftrace quanto no Feather-Trace são criados a partir de macros, e podem ser adicionados tanto em código do *kernel* quanto em módulos do *kernel*. Porém, a definição destes pontos no Ftrace é facilitada. Apesar de o LTTng dar os mesmos poderes que o Ftrace, os *tracepoints* devem ser reescritos em um outro formato para suportar o seu sistema de *buffer*. Este novo formato é muito semelhante, mas estes novos *tracepoints* devem ser reescritos e recompilados com os módulos que fazer parte do LTTng.

Após o estudo, as três ferramentas apresentadas foram avaliadas e comparadas utilizando estas métricas. Para facilitar a observação dos dados, estas métricas qualitativas foram transformadas em uma qualificação de 1 a 5, sendo os valores interpretados como: 1: Sem Suporte; 2: Ruim; 3: Satisfatório; 4: Bom e 5: Muito Bom. Estes resultados são exibidos na Tabela 1:

Métrica	**Feather-Trace**	**Ftrace**	**LTTng**
Trace Dinâmico	Sem Suporte	Satisfatório	Satisfatório
Trace Estático	Bom	Muito Bom	Muito Bom
Buffers	Bom	Bom	Muito Bom
API	Ruim	Muito Bom	Bom
Ferramentas	Satisfatório	Bom	Muito Bom
Arquiteturas	Satisfatório	Muito Bom	Satisfatório
Overhead	Muito Bom	Bom	Bom
Espaço do Usuário	Sem Suporte	Ruim	Muito Bom
Formato do trace	Satisfatório	Bom	Muito Bom
Facilidade de extensão	Satisfatório	Muito Bom	Bom

Tabela 1 – Comparativo entre as ferramentas de *trace*.

Foi gerado o gráfico da Figura 21, que apresenta de maneira mais clara a cobertura de cada ferramenta das métricas em que foram avaliadas.

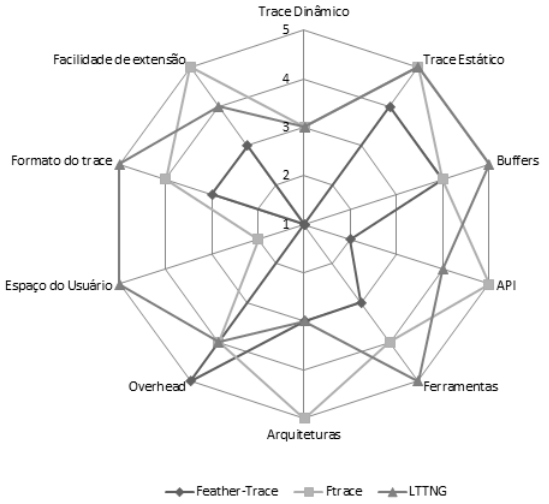

Figura 21 – Comparativo entre as ferramentas

Como se pode observar, na Figura 21, o Feather-Trace foi considerada uma ferramenta boa ou muito boa em um pequeno número de métricas, o que pode-se destacar da ferramenta é justamente o baixo *overhead*. Já o Ftrace e o LTTng se mostram ferramentas mais completas. O Ftrace tem como superioridade a facilidade de extensão e o suporte a arquiteturas, advinda da utilização da API já incorporada ao *kernel* do Linux. Como ponto fraco pode-se destacar a falta de mecanismos para *trace* no espaço do usuário e a falta do suporte a sessões concorrentes de *trace*. Por fim, o LTTng se mostrou superior em pontos como o *trace* no espaço do usuário, o formato padronizado dos *trace* e as ferramentas de análise dos dados, tendo como pontos fracos a API, que é o motivo pelo qual este está incorporado ao *kernel* do Linux, e a facilidade de extensão, também causada pela não integração com a API do *kernel* do Linux.

Com esta análise pode-se concluir que o Feather-Trace é a ferramenta indicada para trace de eventos no *kernel* do Linux quando os eventos são poucos e o *overhead* associado com o trace deve ser o menor possível, como no caso dos testes feitos no Litmus RT. O Ftrace é a ferramenta indicada para *traces* extensivos no *kernel* do Linux, principalmente para análise do fluxo de execução das funções do *kernel* do Linux, utilizando o *function tracer*, ou quando há a necessidade de utilização de *trace* dinâmico. Já o LTTng é a ferramenta indicada para *trace* envolvendo pontos de *trace* no espaço do usuário e a necessidade de execução de *trace* por diversos usuários de forma concorrente.

4.4.2 Escolha da ferramenta

Nenhuma das três ferramentas apresenta de forma clara os pontos de *trace* levantados durante o Capítulo 3. O Feather-Trace apresentou-se como uma ferramenta bastante útil, mas esta muito restrita ao LITMUS RT. A sua forma de *trace*, o *trace* estático, também é implementada nas outras ferramentas, apesar de apresentar maior *overhead*, ainda sim apresentam um bom desempenho. Por isto se descartou a utilização do Feather-Trace.

As outras duas opções, o Ftrace e o LTTng, são capazes de obter as mesmas informações, pois ambas utilizam os mesmos mecanismos de *trace*. As vantagens do LTTng sobre o Ftrace são: *trace* em aplicações no espaço do usuário, múltiplas sessões de *trace* e as ferramentas para análise dos dados. Como levantado no Capítulo 3, os pontos de interesse estão todos no *kernel*, assim, esta vantagem não desqualifica o Ftrace. A necessidade de múltiplas sessões de *trace* também não é um fator determinante para este trabalho. Por fim, como para adicionar pontos de *trace* no LTTng, é necessário antes adicionar os mesmos pontos de *trace* no Ftrace, aliado ao fato de ser mais fácil estender o Ftrace, optou-se por utilizar o Ftrace. Além disto, a utilização do Ftrace não inviabiliza a utilização das ferramentas de análise do LTTng, sendo possível no futuro importar os *traces* para o LTTng e implementar o suporte aos *traces* nas ferramentas de análise do LTTng.

5 ADAPTAÇÕES NA FERRAMENTA DE TRACE

Este capítulo descreve o desenvolvimento das adaptações da ferramenta de *trace*. Foi necessário montar o ambiente de desenvolvimento, definir os pontos de *trace*, fazer o *trace* dos pontos e as alterações no modo que as informações são exibidas. Cada uma destas etapas é descrita nas seções a seguir.

5.1 AMBIENTE DE DESENVOLVIMENTO

O desenvolvimento foi feito em uma máquina virtual, pois cada vez que o *kernel* era recompilado, por causa de uma alteração no código do *tracer*, era necessário reiniciar o sistema, assim, a máquina virtual foi utilizada por dar maior agilidade no processo. Das possíveis implementações de máquinas virtuais, utilizou-se o KVM. A máquina foi criada com 2GB de memória RAM e quatro CPUs. Cada CPU estava vinculada a uma CPU real do computador hospedeiro, um Intel Core i5.

Nesta máquina virtual, foi instalada a distribuição Fedora Linux, na versão 17, para arquitetura Intel 64 bits. Neste sistema optou-se por recompilar o *kernel* do Linux na sua última versão disponível com suporte ao *patch* PREEMPT_RT. Na data do início do desenvolvimento esta versão era a 3.6.7-rt18.

5.2 DEFINIÇÃO DOS PONTOS DE TRACE

A seção 3.1 descreve os contextos de execução do *kernel* do Linux. Cada contexto de execução possui um modo de ser chamado, iniciando a execução de uma nova rotina. O início da execução de uma rotina pode ser considerada a ativação de um *job*, da tarefa que é executada neste contexto. A marcação da ativação do *job* é fundamental para a análise de tempo real.

No contexto do usuário, os processos podem ativar as rotinas do *kernel* de duas formas: via chamadas do sistema ou interrupções. O *hardware* é responsável por ativar rotinas no contexto de interrupção, esta ativação é feita através de algum meio de interrupção, como interrupção de dispositivos, interrupções não mascaráveis, *traps* etc. O contexto das *softirqs* é ativado por rotinas do próprio kernel. No PREEMPT_RT, como as *softirqs* são *threads* do *kernel*, a ativação é feita através das rotinas de escalonamento, acordando o processo das *softirqs*. O mesmo acontece para as demais *threads* do *kernel*.

Disto, pode-se concluir que é necessário fazer *trace* das: formas de interrupção, chamadas do sistema e ativações das rotinas de escalonamento.

Dos métodos de sincronismo e exclusão mutua, é necessário identificar: quando cada método foi chamado, se foi para obter ou liberar um recurso, qual o recurso foi obtido ou liberado, se o recurso foi adquirido na primeira tentativa ou se a rotina sofreu contenção.

Por fim, dos métodos de interferência na execução, é preciso saber quando cada forma de interferência foi ativada e desativada. Podendo a partir disto, avaliar a sua interferência na execução.

5.3 DESENVOLVIMENTO DO *TRACER*

As próximas seções descrevem, passo-a-passo, como se deu o desenvolvimento do novo plugin de *trace* no Ftrace.

5.3.1 Forma de *trace* e IRQs

A primeira tentativa de adaptar o Ftrace foi através dos *tracepoints*. Antes de iniciar o desenvolvimento, viu-se que o Ftrace já possui *tracepoints* para os tratadores de IRQ. Porém, ao ler o código onde estes pontos de trace foram adicionados, viu-se que estes *tracepoints* medem a latência das interrupções de dispositivos, que o tratador de interrupção é a função *do_IRQ*. Porém, existem outras interrupções, como as de faltas de memória geradas pela MMU e de *tick* de relógio, que não passam por este tratador, logo, não são consideradas.

Para fazer o *trace* do início e do fim das interrupções, criou-se dois *tracepoints*: um para sinalizar o início e outro para o fim do tratador de IRQ. Estes *tracepoints* recebem uma *string*, que serve para identificar o ponto de trace. O código a seguir é o código que define o *tracepoint* da entrada de uma IRQ.

```
TRACE_EVENT(irq_named_entry,
        TP_PROTO(char *id),
        TP_ARGS(id),
        TP_STRUCT__entry(
__array( char,  id,   6   )
        ),

        TP_fast_assign(
                memcpy(__entry->id, id, 6);
```

```
        ),

        TP_printk("%s", __entry->id)
);
```

Para chamar este *tracepoint*, utiliza-se a função trace_irq_named_entry (char *name). Para testar o sistema, este *tracepoint* foi adicionado em alguns tratadores de interrupção, como por exemplo, na função *smp_apic_timer_interrupt*. Esta função é responsável por atender as interrupções de timer. Foram adicionados dois *tracepoints*, um logo no início e outro logo no fim da função, deste modo, o delta de tempo entre os dois *tracepoints* é o tempo de execução do tratador de interrupção. A seguir é exibida a função com os *tracepoints*.

```
void __irq_entry
smp_apic_timer_interrupt(struct pt_regs *regs)
{
        struct pt_regs *old_regs = set_irq_regs(regs);

        trace_irq_named_entry("TIMER");

        ack_APIC_irq();

        irq_enter();
        exit_idle();
        local_apic_timer_interrupt();
        irq_exit();

        set_irq_regs(old_regs);

        trace_irq_named_exit("TIMER");
}
```

Apesar de ser funcional, esta abordagem é muito intrusiva no código e dificilmente seria aceita pela comunidade de software livre, pois a chamada dos *tracepoints* acabam por ”sujar”o código da função. Este é um dos motivos de rejeição de vários *patchs* com *tracepoints* enviados para inclusão no *kernel* do Linux.

Pensando em uma outra forma de *trace* para medir o tempo que uma função demorou para executar, observou-se que esta é a característica básica do *trace function_graph* do Ftrace: marcar o início e o fim da execução de uma função, exibindo o tempo que esta demorou para executar, como no exemplo a seguir:

[0]”Os comentários foram removidos”

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 0)               |    sockfd_lookup_light() {
 0)   0.107 us    |      fget_light();
 0)   0.882 us    |    }
```

Neste exemplo, pode-se observar que a função *sockfd_lookup_light* foi executada, sua execução durou 0.882 *us*, na CPU 0. A partir disto, buscou-se utilizar *function_graph* como base para o desenvolvimento de um novo *plugin* de trace do Ftrace. Porém, diferente do *plugin* existente, o novo *plugin* não irá fazer o *trace* de todas as funções, mas somente das funções que apresentam impacto temporal na execução das tarefas.

Como descrito na seção 4.2.3, um *plugin* do Ftrace é formado por duas partes, uma responsável por coletar as informações: funções que são chamadas no início, e possivelmente no fim, de cada função, e armazenam em um *buffer* as informações de *trace*. E funções que são chamadas quando o *trace* é lido: funções que coletam dados do *buffer* e imprimem as informações. A definição destas funções é feita por meio de estruturas de dados que contém ponteiros para as funções que implementam estas rotinas.

Com base no *trace function_graph*, um novo *plugin* foi criado. Denominado *timeflow* e definido no fonte *kernel/trace/trace_timeflow.c*. No início do desenvolvimento deste *plugin*, tentou-se utilizar o máximo de funções já existentes, porém, algumas tiveram que ser alteradas.

As alterações começaram com a definição de novos tipos de dados para o *buffer* do Ftrace. Este tipo de dado, que é uma macro, é utilizado pelo Ftrace para definir que função deve ser chamada para interpretar e imprimir os dados armazenados no *buffer*. Como o *function_graph* insere duas entradas no *trace* para cada função, uma na entrada e outra na saída da execução da função, dois tipos de dados foram criados. Estes novos tipos de dados do Ftrace foram registrados para chamar as funções que imprimem as entradas de *trace* do *plugin* de *trace* do *function_graph*, pois a estrutura de dados com as informações armazenadas no *buffer* é a mesma do *function_graph*.

Também foi necessário definir a estrutura que define um novo *plugin* de trace, como exemplificado na sessão 4.2.3.3. Nesta estrutura foram definias as funções do *tracer*, por exemplo, as funções ativam e desativam o *trace*, que ajustam as *flags* de opções etc. Neste caso, todas as funções foram apontadas para as funções já existentes do *function_graph*. Após estas alterações, o novo *plugin* foi executando, tendo como saída um *plugin* igual ao *function_graph*, a partir disto, buscou-se fazer as alterações desejadas.

O primeiro objetivo foi entender as funções do *tracer* que são cha-

madas, a cada início e fim das funções, para depois fazer as alterações necessárias para fazer o *trace* das funções desejadas, começando pelas funções dos tratadores de IRQs.

Na estrutura *tracer*, que define o *plugin* de *trace*, o ponteiro para função *init* aponta para a função chamada quando um *tracer* é ativado, no caso do *function_grapher*, este ponteiro é inicializado com a função *graph_trace_init*. Na função *graph_trace_init* as funções que são chamadas no início e fim de cada função durante a sessão de *trace* são registradas, como a função *register_ftrace_graph*, sendo a função *trace_graph_entry* chamada na entrada e a função *trace_graph_return* no fim de cada função.

A função *trace_graph_entry* é chamada com um argumento, este argumento contém o endereço da função a ser feito o trace. O endereço da função passa por algumas verificações, se aprovado, uma nova entrada é adicionada no *buffer* de *trace*, fazendo o *trace* da execução desta função. Caso o endereço não seja aprovado, a entrada é descartada.

Uma dessas verificações é se o endereço da função é o endereço de uma das funções configuradas no arquivo *set_ftrace_filter*, que filtra quais funções fazer o *trace*, como descrito na seção 4.2.2. Quando o nome de uma função é passado para este arquivo de configuração do Ftrace, o nome da função é traduzido no endereço de memória desta função, e este endereço é adicionado em um vetor. Quando existe alguma entrada neste vetor, o endereço passado para a função *trace_graph_entry* é comparado com todas a entradas do vetor, caso o endereço esteja no vetor, o *trace* da função é feito, caso não, ignorado.

Então surgiu a ideia de adicionar o endereço das funções desejadas neste vetor, porém, devia-se implementar uma forma mais eficiente de busca neste vetor, pois a busca linear não é um modo eficiente.

Durante os estudos do trace, observou-se que quando o trace *function_graph* vai mostrar o endereço de uma função, e este endereço está entre os valor das *__irqentry_text_start* e *__irqentry_text_end*, este endereço é considerado o endereço de um tratador de IRQ. Investigando, descobriu-se que isto tem ligação com a macro *__irq_entry* adicionada à declaração de todas as IRQs do sistema, como no exemplo da função *smp_apic_timer_interrupt* descrita anteriormente. Esta macro, declarada no *header* do Ftrace, é definida como:

```
#define __irq_entry             \
 __attribute__((__section__(".irqentry.text")))
```

Esta macro é na verdade uma instrução utilizada pelo compilador e pelo *linker*. Esta instrução diz para agrupar as funções, com esta declaração, em uma mesma seção de texto do binário do *kernel*, assim, sabendo o endereço

de início e de fim desta sessão, é possível determinar se uma função é ou não um tratador de IRQ. Para auxiliar neste pesquisa, o *linker* cria as variáveis *__irqentry_text_start* e *__irqentry_text_end*. O valor destas variáveis demarcam o endereço de início e fim da seção do binário do *kernel* onde as funções foram alocadas, bastando testar se o endereço está no intervalo [*__irqentry_text_start*, *__irqentry_text_end*[para definir que esta função é um tratador de IRQ. Isto apresenta algumas vantagens, por exemplo, a pesquisa do vetor, que apresenta uma complexidade $O(n)$, é substituída por uma pesquisa em tempo constante: $O(1)$. Também a adição de um novo tratador de interrupção não exige alterações no *tracer*, basta obedecer a regra de adicionar a macro *__irq_entry* na declaração da função, o que já é um requisito.

A partir disto, criou-se uma nova função de *trace* para o *trace_timeflow*, semelhante a função *trace_graph_entry*, onde foi adicionada uma restrição: só fazer *trace* das funções na seção de texto dos tratadores de IRQ, o que funcionou. A partir deste resultado buscou-se usar a mesma técnicas para as funções de exclusão mutua.

5.3.2 Exclusão mutua

Iniciando pelos *spinlocks*, buscou-se uma solução semelhante a dos tratadores de IRQ. Porém, ao ler a implementação dos *spinlocks*, notou-se que existem diversas implementações do *spinlock*. A escolha de qual definição de *spinlock* usar é feita a partir das configurações de compilação do *kernel*. Feito um levantamento, a Figura 22 demonstra como é determinada, dependendo das configurações de compilação, a implementação de *spinlock* que o *kernel* irá utilizar.

Como demonstrado no diagrama, a definição de *spinlock* depende das seguintes configurações de compilação do *kernel*:

CONFIG_PREEMPT_RT_FULL Define o modo de preempção do *kernel*, caso este seja o modo de tempo real, todos o *spinlocks* são convertidos em *rt_spin_locks*, que acaba por utilizar os mecanismos do *rt_mutex*. Onde não se pode utilizar os *rt_spinlocks*, o *patch* PREEMPT_RT os substitui por *raw_spinlocks*, que são os *spinlocks* tradicionais. Neste trabalho esta opção será utilizada.

CONFIG_SMP Se o sistema não for multiprocessado, os *spinlocks* tornam-se apenas desativação e ativação da preempção. Como este trabalho foca sistemas de tempo real multiprocessados, esta configuração será utilizada.

CONFIG_INLINE_SPIN_LOCK Define se os *spinlocks* vão ser uma ma-

cro ou uma função, um detalhe importante é que os *tracers* baseados

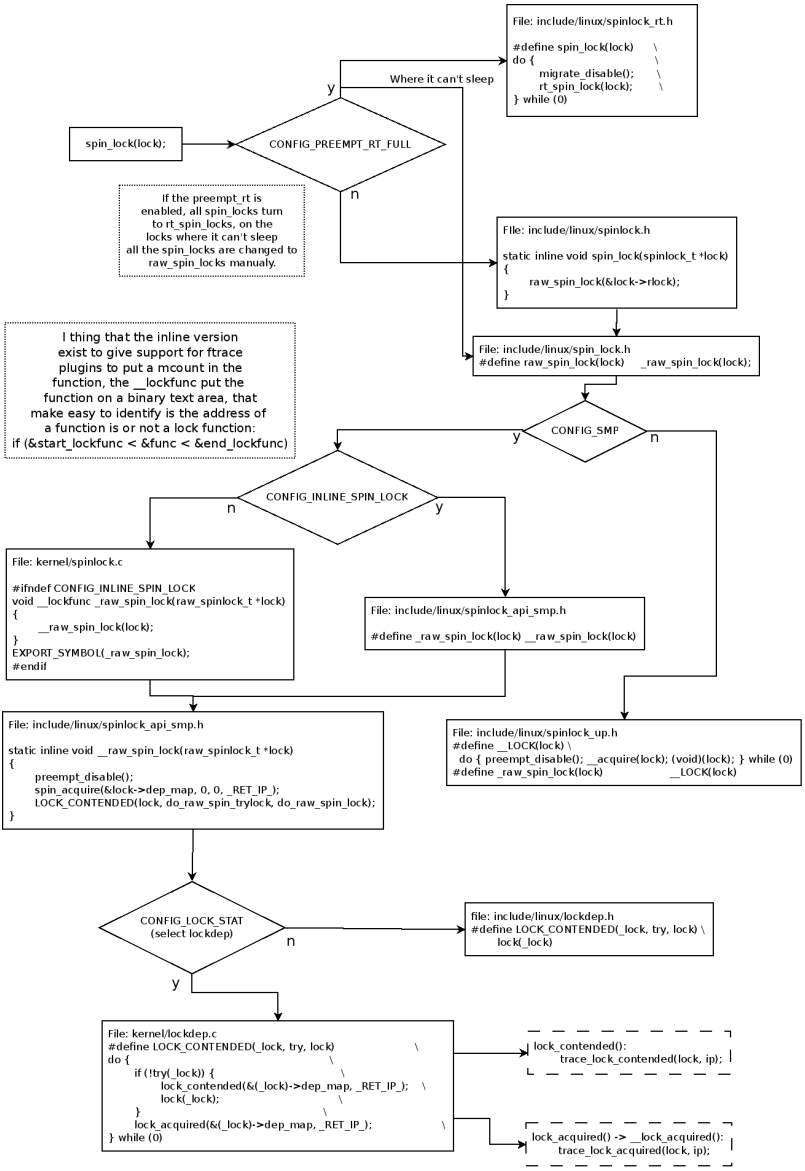

Figura 22 – Definição dos Spinlocks

no Ftrace só fazem *trace* das funções, assim, se os *spinlocks* forem definidos apenas como macros, estes não aparecerão nos *traces* do Ftrace. Por isto optou-se por utilizar esta configuração.

CONFIG_LOCK_STAT Esta opção define se os LOCK_STAT e o LOCK_DEP serão ativos no sistema. Estas são implementações de checagem e *debug* da utilização de *locks*, por exemplo, conferindo se os *locks* são adquiridos e liberados na ordem correta, a fim de evitar *deadlocks*. Esta opção é utilizada, pois adiciona *tracepoints* da aquisição dos *locks*.

Sobre a opção CONFIG_INLINE_SPIN_LOCK, esta possui uma equivalente para cada chamada de aquisição de *spinlock*, como a função *spin__bh*, *spin_lock_irq*, etc. Porém, não existia uma equivalente para a função *unlock*, assim, esta foi implementada. Em todas as declarações *inline* das funções é adicionada a macro *__lockfunc*. Esta macro tem função semelhante ao *__irq_entry*: diz para o compilador e o *linker* agrupar estas funções em uma área de texto do binário do *kernel*, no caso destas funções, as variáveis *__lock_text_start* e *__lock_text_end* contém o endereço de início e fim desta sessão.

Como mencionado, a opção CONFIG_LOCK_STAT adiciona alguns *tracepoints* ao sistema. Estes *tracepoints* dão informações sobre a aquisição e liberação dos *locks* do *kernel*. Com esta opção os seguintes *tracepoints* são adicionados ao sistema:

lock:lock_acquire: Informa que a tarefa quer adquirir o lock;

lock:lock_acquired: Informa que a tarefa adquiriu o lock;

lock:lock_contended: Informa que a tarefa ficou retida em um lock;

lock:lock_release: Informa que a tarefa liberou o lock.

Apesar de adicionar *overhead* as funções de *lock*, o LOCK_DEP foi ativado no sistema por adicionar estes *tracepoints*, não somente nas funções de *spinlock*, mas nas demais funções de exclusão mutua.

Nos demais métodos de exclusão mutua: mutex, semáforos, RCU e RT_MUTEX, foram adicionados, na declarações das funções, a macro *__lockfunc*. Deste modo, para identificar se uma função é uma função de exclusão mútua, basta checar se o endereço da função está entre os valores das variáveis *_lock-_text_start* e *__lock_text_end*.

Por fim, foi adicionada a verificação se a função é ou não uma função de exclusão mútua ao *trace_timeflow*. A partir destas alterações, as chamadas das funções de exclusão mútua começaram a aparecer no *trace*, junto com as funções de IRQ.

Além das funções, foi criada uma função que controla os *tracepoints* do LOCK_DEP, esta função é chamada no início da sessão de trace para ativar os *tracepoints*, e ao final da sessão para desativar os *tracepoints*.

5.3.3 Preempção desabilitada

A primeira ideia sobre o *trace* das operações de controle de preempção, foi colocar um *tracepoint* em cada chamada das funções que habilitam e desabilitam a preempção. Porém, como é possível aninhar as chamadas, o que realmente importa é a primeira chamada que desabilita a preempção, e a última chamada que reabilita a preempção. As demais chamadas, que ocorrem entre as trocas de estados, não importam para a análise temporal do sistema, pois estas acabam por não afetar, temporalmente, a execução do sistema.

Como as funções *preempt_enable* e *disable* são implementadas como macros, estas chamadas não aparecem na execução dos *plugins* do Ftrace. Por isto, utilizou-se *tracepoints*. Durante a leitura e entendimento do código que desabilita e habilita a preempção, observou-se que um *tracer* do Ftrace tinha uma função parecida com o que se desejava implementar. O *tracer preemptoff* mede quanto tempo a preempção ficou desabilitada, exibindo a cadeia de funções que foram chamadas com a preempção desabilitada. A partir disto, procurou-se neste *tracer* o ponto onde se detecta o início e fim de cada seção com preempção desabilitada.

Internamente, as operações de *enable* e *disable* da preempção são controladas pelas funções *add_preempt_count* e *sub_preempt_count*. Nessas funções são chamadas as funções que habilitam e desabilitam o *tracer preemptoff*. Estas funções são chamadas quando o sistema detecta uma mudança no estado da preempção, então, nestes pontos foram adicionados os *tracepoints*, um quando o sistema desabilita a preempção e outro quando o sistema reabilita a preempção. Cada *tracepoint* recebe como argumento o endereço da função que desabilitou ou habilitou a preempção, este endereço é recuperado com a macro _RET_IP_ do GCC. A macro _RET_IP_ diz qual é o endereço de retorno da função, logo, esta é a função que desabilitou/reabilitou a preempção. A seguir é demonstrado um exemplo da saída destes *tracepoints*.

```
sshd-784   [002] d...211 123605.454762: sched_preempt_disable: \
at _raw_spin_lock_irqsave
sshd-784   [002] ....211 123605.454767: sched_preempt_enable: \
at _raw_spin_unlock_irqrestore
```

5.3.4 IRQ desabilitada

De maneira análoga ao controle de preempção, as funções que controlam as IRQs são implementadas como macros, não aparecendo nos *plugins* de *trace* do Ftrace. Também como no controle de preempção, é possível aninhar as chamadas que desabilitam e reabilitam as IRQs, mas o que importa para a análise é o momento em que há a mudança de estado, do sistema com as interrupções habilitadas para desabilitadas e do sistemas com as interrupções desabilitadas para reabilitadas.

A forma de *trace* adotada foi utilizando *tracepoints*. Em toda chamada das funções que desabilitam as IRQs é feito o teste, antes de desativar as IRQs, para ver se as IRQs estão ativas, caso estiverem, o *tracepoint* é chamado. O controle inverso é feito na reabilitação das IRQs. Quando as IRQs são reabilitadas, após a reabilitação, é feito o teste para ver se as IRQs estão habilitadas, caso estiverem, ouve uma mudança no estado, então um *tracepoint* é chamado.

Um exemplo da utilização destes tracepoints é exibido a seguir:

```
ksoftirqd/2-21    [002] d...111 238688.647106: \
     local_irq_disable:  at _raw_spin_lock_irq
ksoftirqd/2-21    [002] d...211 238688.647109: \
     local_irq_enable:  at _raw_spin_unlock_irq
```

Com pode-se ver no exemplo, as interrupções foram desabilitadas na CPU 2 pela função *_raw_spinlock_lock_irq*, e reabilitas na CPU 2 pela função *_raw_spinlock_unlock_irq*, demonstrando o funcionamento descrito na seção 3.2.1.2.

Também foram adicionados dois *tracepoint* que sinalizam quando uma linha de interrupção é desabilitada e reabilitada em todas as CPUs. Como não é possível aninhar estas chamadas, foi necessário apenas adicionar os pontos de trace na função que desabilita e habilita a interrupção.

5.3.5 Migração desabilitada

A migração de processos é desabilitada e habilitada pelas funções *migrate_disable* e *migrate_enable*. Como na preempção, a ideia foi colocar um *tracepoint* nesta função. Porém, também como na preempção, estas chamadas podem ser aninhadas, então tomou-se o mesmo cuidado de fazer o trace apenas quando o sistema muda de estado de habilitado para desabilitado e vice-versa. Um exemplo da saída destes *trace*:

```
bash-949   [001] ....1.. 125250.511951: sched_migrate_enable: \
```

```
comm=bash pid=949 prio=120 cpu=1
bash-949   [001] ....111 125250.511952: sched_migrate_disable: \
comm=bash pid=949 prio=120 cpu=1
```

5.3.6 Funções de escalonamento

Inicialmente pensou-se em ativar alguns *tracepoints* de escalonamento já existentes, sendo os seguintes:

sched:sched_wakeup: Sinaliza quando uma tarefa muda do estado dormindo para pronta para executar;

sched:sched_wakeup_new: Sinaliza quando uma nova tarefa muda do estado dormindo para pronta para executar;

sched:sched_switch: Sinaliza a troca de contexto das tarefas, pode detectar quando uma tarefa mudou o estado de pronta para dormindo;

sched:sched_migrate_task: Sinaliza a migração de uma tarefa de um processador para outro;

sched:sched_pi_setprio: Informa a mudança da prioridade de um processo causada pelo controle de herança de prioridade dos *rt_mutex*;

Porém, durante o estudo sobre as funções de escalonamento, notou-se que as funções centrais do sistema de escalonamento do Linux possuem uma macro antes da definição das funções, esta macro é a *__sched* que define e agrupa todas as funções de decisão de escalonamento em uma área de texto do *kernel*, como visto nas funções de IRQ e de exclusão mútua. As funções de escalonamento são identificadas por estarem entre os valores das variáveis [*__syscallentry_text_start*, *__syscallentry_text_end*[.

Então, além dos *tracepoints*, foi adicionado ao trace as funções de escalonamento do sistema.

5.3.7 Chamadas do sistema

Apesar de existir um *tracepoint* para cada chamada do sistema, para identar as chamadas de função e identificar melhor em que contexto cada função e *tracepoint* aconteceu, decidiu-se fazer o *trace* das chamadas das funções que implementam as chamadas do sistema. Para isto utilizou-se a mesma abordagem dos tratadores de IRQ e mecanismos de exclusão mutua.

que é agrupar as funções das chamadas do sistema em uma única seção de texto do binário do kernel.

O primeiro passo foi definir a macro que define uma seção de texto. Nesta nova seção, as funções que implementam as chamadas do sistema serão agrupadas. Para isto foi necessário adicionar a seguinte macro no *header* do Ftrace:

```
#define __syscall_entry      \
        __attribute__((__section__(".syscallentry.text")))

/* Limits of syscall entrypoints */
extern char __syscallentry_text_start[];
extern char __syscallentry_text_end[];
```

Além da macro, foram criadas as variáveis que conterão os endereços de início e fim da seção de texto do binário do *kernel*. Além da declaração, é preciso informar ao *linker* como criar a seção e preencher as variáveis com os valores de início e fim da seção, para isto é necessário editar o arquivo *include/asm-generic/vmlinux.lds.h* e adicionar as seguintes definições:

```
#ifdef CONFIG_FUNCTION_TIMEFLOW
#define SYSCALLENTRY_TEXT                                       \
                ALIGN_FUNCTION();                               \
                VMLINUX_SYMBOL(__syscallentry_text_start) = .;  \
                *(.syscallentry.text)                           \
                VMLINUX_SYMBOL(__syscallentry_text_end) = .;
#else
#define SYSCALLENTRY_TEXT
#endif
```

Também é necessário informar, no fonte *recordmcount.c* e no *script recordmcount.c* do Ftrace, a existência desta nova seção. Estes fontes são responsáveis por salvar os endereços das funções no fonte do *kernel*, para depois ativar e desativar dinamicamente o *trace* destas funções, como documentado na seção 4.2.3.2.

Após definida, é só adicionar a macro na definição das funções que implementam as chamadas do sistema. As chamadas do sistema são definidas por uma única macro, a SYSCALL_DEFINE(name), então, bastou adicionar a macro *__syscall_entry* na definição. A seguir é exibida a alteração feita, no formato de *patch*, para identificar melhor as alterações.

```
 #define __SYSCALL_DEFINEx(x, name, ...) \
- asmlinkage long \
+ asmlinkage long __syscall_entry     \
  sys##name(__SC_DECL##x(__VA_ARGS__))
```

Para finalizar, foi adicionada uma checagem no *trace*, verificando se o endereço da função estão ou não no intervalo [*__syscallentry_text_start*, *__syscallentry_text_end*[.

5.4 FORMATAÇÃO DOS DADOS

O primeiro passo foi ativar algumas opções que não estão ativas por padrão no *function_graph tracer*, como mostrar o processo e o tempo absoluto na entrada do *trace*. Com estas alterações, a saída do *trace* ficou mais densa, exigindo uma mudança na formatação dos dados, por isto, todas as funções do *trace_timeflow* foram desvinculadas do *trace_graph* e alteradas para mostrar as informações desejadas.

Além dos campos presentes no *function_graph*, adicionou-se o campo prioridade, que diz a prioridade do processo que está em execução, isto é importante para poder comparar as decisões de escalonamento do sistema, por exemplo, para ver em que momento uma tarefa recebeu a prioridade de herança de outra tarefa.

Outra alteração foi quanto a resolução do relógio, no *function_graph* o relógio era truncado para apresentar as informações em microsegundos, no *trace_timeflow* este foi alterado para ter a resolução em nanosegundos.

No *function_graph*, quando uma função que atende IRQ é chamada, é adicionada antes do início da função e depois do fim da função uma linha com uma seta, usando o sinal de '=' como linha, indicando a entrada e uma indicando a saída, como no exemplo abaixo:

```
 3)   ==========> |
 3)                |    smp_apic_timer_interrupt() {
```

No *trace_timeflow* isto foi mantido, e incrementado, quando uma função que implementa uma chamada do sistema é exibida, antes é adicionada antes da execução da função e depois da execução da função uma seta, usando o sinal de '-' como linha, indicando onde iníciou e finalizou a execução de uma chamada do sistema. Em um breve comparativo, o *function_graph* se apresenta como:

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 3)   1.468 us    |          } /* __pollwait */
 3)   2.202 us    |        } /* unix_poll */
 3)   2.633 us    |      } /* sock_poll */
```

```
 3)   0.061 us    |      fput();
 3)   0.148 us    |      fget_light();
 3)               |      sock_poll() {
 3)               |        datagram_poll() {
 3)               |          __pollwait() {
 3)               |            add_wait_queue() {
 3)   0.056 us    |              _raw_spin_lock_irqsave();
 3)   0.061 us    |              _raw_spin_unlock_irqrestore();
 3)   0.980 us    |            }
 3)   1.445 us    |          }
 3)   1.967 us    |        }
 3)   2.458 us    |      }
```

E do *trace_timeflow*[1]:

```
# tracer: timeflow
#
#  TASK-PID    PRIO  CPU      TIME          DURATION                  FUNCTION CALLS
#   |    |      |    |     |      |          |   |                     |   |   |   |
 sendmai-770  [120]  [0] 155865.845398941     ---------->
 sendmai-770  [120]  [0] 155865.845398941                       sys_rt_sigprocmask() {
 sendmai-770  [120]  [0] 155865.845401397                         rt_spin_lock() {
 sendmai-770  [120]  [0] 155865.845403120                           lock_acquire: ffff88006c2ccaf8 &sighand->siglock
 sendmai-770  [120]  [0] 155865.845404163                           irq_disable: at __lock_acquire
 sendmai-770  [120]  [0] 155865.845404995                           irq_enable:  at __lock_acquire
 sendmai-770  [120]  [0] 155865.845406428     4.355 us            } rt_spin_lock ()
 sendmai-770  [120]  [0] 155865.845408287                         rt_spin_unlock() {
 sendmai-770  [120]  [0] 155865.845409104                           lock_release: ffff88006c2ccaf8 &sighand->siglock
 sendmai-770  [120]  [0] 155865.845409916                           irq_disable: at lock_release_holdtime.part.7
 sendmai-770  [120]  [0] 155865.845410698                           irq_enable:  at lock_release_holdtime.part.7
 sendmai-770  [120]  [0] 155865.845411945     2.977 us            } rt_spin_unlock ()
 sendmai-770  [120]  [0] 155865.845413494   + 13.876 us         } sys_rt_sigprocmask ()
 sendmai-770  [120]  [0] 155865.845413494     <----------
```

Interpretando o exemplo do *trace_timeflow*, a primeira e segunda linha de trace mostram que o processo Sendmail, que possui a prioridade normal do Linux, executa a chamada do sistema *rt_sigprocmask*, esta função tenta adquirir o *rt_mutex sighand->siglock*, conseguindo em 4.355us. Dentro da função de *lock* o sistema desabilita e habilita as IRQs, apos 1859 ns, a chamada do sistema libera o *rt_mutex sighand->siglock*, terminando assim a chamada do sistema *sys_rt_sigprocmask* em 13.876 us.

5.5 CONCLUSÃO

Apesar de ser possível implementar um *tracer* semelhante utilizando apenas os *tracepoints*, a utilização de um *plugin* do Ftrace facilitou o desenvolvimento, além de interferir menos no código do sistema. Outro facilitador foi a a descoberta de como agrupar um conjunto de funções em uma área de

[1] O tamanho da fonte para o *trace_timeflow* teve de ser reduzido por causa da grande quantidade de dados que o *trace* deve exibir

texto do binário do *kernel*, auxiliando a identificar, em tempo constante, quais as funções deseja-se fazer o *trace*.

Além da facilidade, o *plugin function_graph* já trazia pronta algumas informações bastante úteis, como por exemplo, o tempo de duração da execução de uma função.

No fim do desenvolvimento, o código do novo *tracer*, o *trace_timeflow*, apesar de semelhante, apresentava diversas alterações do *tracer function_graph*, qual se baseou. Apesar das alterações, o novo *tracer* pode ser utilizado de maneira análoga aos outros *plugins*, como no exemplo a seguir:

```
[root@prt tracing]# cat available_tracers
blk timeflow function_graph wakeup_rt wakeup preemptirqsoff \
                            preemptoff irqsoff function nop
[root@prt tracing]# echo timeflow > current_tracer
[root@prt tracing]# echo 1 > tracing_on
```

No próximo capítulo, este novo *plugin* de *trace* do Ftrace é utilizado, a fim de validar seu funcionamento e mostrar e como é possível, com a análise de sua saída, identificar as interferências e bloqueios que as tarefas sofrem durante a execução no kernel do Linux. Ao final, deseja-se mostrar a relação entre essas interferências e bloqueios com as variáveis utilizadas nos métodos analíticos da teoria de tempo real.

6 CARACTERIZAÇÃO DAS TAREFAS NO LINUX

Este capítulo apresenta o último objetivo deste trabalho, que é avaliar a capacidade da ferramenta de *trace* obter as métricas comuns na teoria de tempo real e caracterizar a execução das tarefas no Linux com o PREEMPT-RT, visando a análise de escalonabilidade de tempo real.

Para cumprir este objetivo, um ambiente experimental foi montado. O ambiente é composto de um sistema multiprocessado, executando o sistema operacional Linux com o *patch* de tempo real PREEMPT_RT, descritos no Capítulo 2. Executando duas tarefas de tempo real, descrita na seção 6.1.2. Neste ambiente, buscou-se identificar na execução do Linux, os valores para as variáveis aleatórias utilizadas na análise de tempo de resposta, descritas nas Seção 2.2. Para isto, serão conduzidas análises dos métodos descritos no Capítulo 3, utilizando a ferramenta de *trace* Ftrace, descrita no Capítulo 4, com as extensões criadas durante este trabalho, descritas no Capítulo 5.

As próximas seções descrevem as etapas do experimento para avaliação da ferramenta e para a caracterização das tarefas no Linux com o PREEMPT_RT.

6.1 CONDIÇÕES DOS EXPERIMENTOS

As seções a seguir descrevem o ambiente de execução, as tarefas e as configurações do ambiente durante a execução dos experimentos.

6.1.1 Equipamento utilizado

Para a execução dos experimentos para a análise do *trace* foi utilizado um servidor Super Micro, com um processador Intel Xeon E5260, de quatro núcleos de 2.4Ghz, oito *threads* de execução e 4GB de memória RAM.

6.1.2 Configuração do ambiente e da tarefa de tempo real

Foram criados dois softwares, a fim simular o comportamento das tarefas de tempo real comum na literatura: uma tarefa periódica que executa como um processo no espaço de usuário no Linux e um módulo que executa no espaço do *kernel* do Linux.

Em cada ativação, a tarefa periódica ativa o módulo no *kernel* através de uma interface de dispositivo de caractere. A ativação é feita através da lei-

tura ou escrita na interface de caractere do módulo. Ao escrever no módulo, a tarefa no espaço do usuário configura um tempo de espera ocupada, a ser executado dentro do *kernel* do Linux durante as operações de leitura do módulo.

Para simular uma seção crítica, a espera ocupada implementada no *kernel* é protegida por uma variável de exclusão mútua. Ao efetuar a leitura, o módulo do *kernel*, executando em nome do processo no espaço do usuário, tenta adquirir a variável de exclusão mútua, onde pode haver contenção. Após adquirir a variável, o módulo executa a espera ocupada pelo tempo determinado na escrita. Ao sair da espera ocupada, o módulo libera a variável de exclusão mútua e retorna para o espaço do usuário.

Em uma ativação sem concorrência a tarefa executa sem bloquear, tanto no espaço do *kernel* quanto no espaço do usuário. Este modo de operação, em que a tarefa suspende somente entre uma ativação e outra e não durante a ativação, dá a tarefa um comportamento comum na literatura para as tarefas de tempo real.

Para simular um ambiente com concorrência, e assim forçar a utilização dos mecanismos de exclusão mútua, foram executadas duas tarefas no espaço do usuário. Estas tarefas acessam o módulo, fazendo leituras concorrentes na interface do módulo, causando contenção na disputa da variável de exclusão mútua compartilhada. O diagrama da Figura 23 descreve a estrutura destas tarefas.

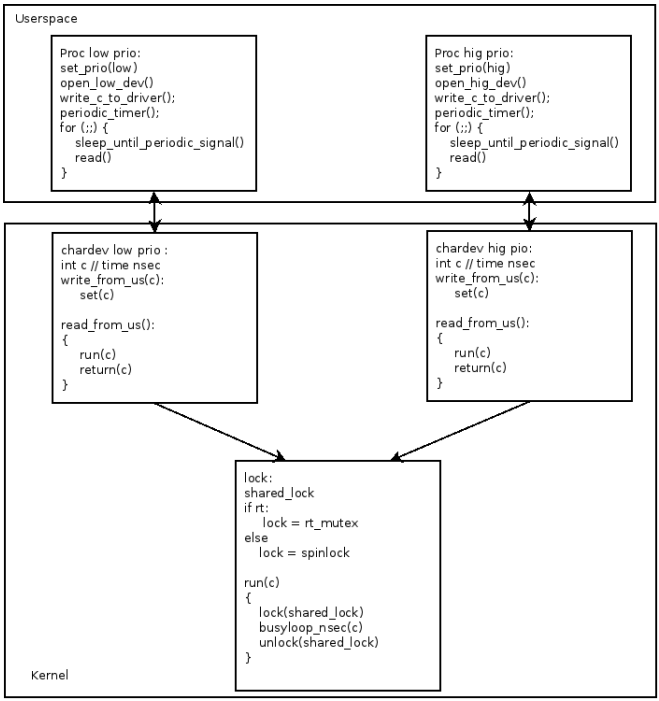

Figura 23 – Aplicação utizada nos experimentos

O mecanismo de exclusão mútua utilizado no módulo pode ser um dos apresentados no Capítulo 3. Pela característica da tarefa, foram conduzidas análises usando *spinlocks* e RT Mutex. Quando necessário o mecanismo utilizado será informado. Nas saídas do trace, a aplicação utilizada nos testes será identificada pelo seu nome: *pi*.

Antes da execução que gerou os dados analisados, foram executados alguns testes no sistema. O primeiro teste foi a execução do experimento com tempos supostos a partir da característica de execução. Após os testes, optou-se por executar as duas tarefas com as maiores prioridades de tempo real do sistema. Porém, ambas com prioridade, período e tempos de execução dentro da sessão critica diferentes. A configuração das tarefas é descrita na Tabela 2.

Identificação	Prioridade	Período	Espera ocupada no *kernel*
pi_{alta}	90 (alta)	250	1 us
pi_{baixa}	70 (baixa)	300	100 us

Tabela 2 – Configuração das tarefas de tempo real.

6.1.3 Tamanho dos buffers

Após executar os primeiros experimentos, notou-se que os dados pareciam inconsistentes, com algumas lacunas de tempo, aparentando um problema na coleta dos dados. Ao analisar o tempo, notou-se que o tamanho original do *buffer* era pequeno demais para os experimentos. De maneira iterativa, foram executados experimentos, onde aumentava-se o tamanho do *buffer* e diminuía-se o tempo do experimento, até achar um tamanho de *buffer* e tempo de experimento em que o tamanho do *trace* era menor que o tamanho do *buffer*, assim, garantindo a consistência dos dados na sessão de *trace*.

Como o *buffer* do Ftrace está no *kernel*, e a memória do *kernel* não pode sofrer paginação para uma memória secundária, por exemplo *swap*, o valor máximo para o tamanho do *buffer* deve ser inferior ao tamanho da memória real do sistema. Como o Ftrace utiliza um *buffer* para cada CPU, o tamanho do *buffer* é definido por CPU. Desta forma, optou-se por utilizar 350MB de *buffer* por CPU, onde em um sistema com 8 CPUs, somam-se 2.8 GB de memória para a sessão de *trace*.

Executando experimentos para este tamanho de *buffer*, viu-se que o tempo de 2 segundos era seguro o bastante para não haver sobrescrita de dados de *trace*. Com este tempo, os *traces* das CPUs com mais ocorrência de eventos, ficavam com cerca de 280 MB. Nestes *traces* era possível ver desde o momento em que o *trace* estava sendo habilitado e nem todos os pontos de *trace* estava ativos, até a execução do fim do *script* que controlava o experimento. O tamanho do *trace* de cada CPU variava de acordo com o número de processos que executavam nesta CPU. Nos experimentos realizados, o tamanho do *trace* global do sistema ficava em torno de 1.2 GB, com cerca de 12 milhões de registros. Este valor é bastante grande, tanto para coleta, quanto para análise. Isto se dá principalmente pela granularidade de eventos de *trace*.

Para evitar interferências do *scripts* de automação do experimento, as linhas de *trace* que ocorrem durante a inicialização e o a finalização do experimento foram removidas.

6.2 CARACTERIZAÇÃO DA EXECUÇÃO DAS TAREFAS NO LINUX

Com o ambiente descrito na Seção 6.1 foram conduzidos alguns experimentos para caracterizar a execução das tarefas no Linux. Com estes experimentos foi possível caracterizar os dois contextos de tarefas do Linux com o PREEMPT_RT, o contexto dos tratadores de interrupção e das *threads*. As próximas seções descrevem a caracterização destas tarefas, iniciando pelos tratadores de interrupção.

6.2.1 Caracterização dos Tratadores de Interrupção no Linux

Uma interrupção pode ocorrer a qualquer momento durante a execução do código. No caso da arquitetura Intel, o tratador de interrupção é chamado no fim da execução da instrução que o processador está executando no momento em que a interrupção foi ativada. O tratador de interrupção pode ser atrasado pelo mascaramento das interrupções, exceto para as interrupções não mascaráveis. Neste ponto, além do controle de interrupção feito pelo sistema operacional, com o uso das funções que desabilitam as interrupções, a arquitetura do processador também define regras de mascaramento das interrupções no processador. Por exemplo, durante a execução de um *page fault* o processador desabilita as interrupções até que o tratador de interrupção retorne com a instrução IRET ou as interrupções sejam reabilitadas pelo sistema operacional. Os pontos nos quais o processador desabilita as interrupções vão além do controle do sistema operacional, por isto não são possíveis de *trace* com as tecnologia atuais. Para cada arquitetura de hardware devem ser estudados e levantados os pontos em que o processador desabilita as interrupções. Levando a arquitetura Intel como base, e de maneira bastante simplificada, considera-se neste trabalho que o processador desabilita as interrupções na chamada de um tratador de interrupções, tanto as mascaráveis quanto as não mascaráveis. Considera-se também que uma interrupção mascarável pode ser interrompida por uma interrupção não mascarável, e também que uma interrupção não mascarável bloqueia as interrupções não mascaráveis.

A seguir é exibido um exemplo da execução de um tratador de interrupção durante os experimentos. Neste exemplo, as entradas de trace que não interferiram na execução temporal do tratador de interrupção foram substituídas por *[...]*, isto para deixar o exemplo mais claro.

```
pi-851    [ 29]  [05]    96.790072126                    _raw_spin_unlock_irqrestore() {
pi-851    [ 29]  [05]    96.790072638                      lock_release: ffff88013900d7b8 &cpu_base->lock
pi-851    [ 29]  [05]    96.790073738                      local_irq_enable:  at _raw_spin_unlock_irqrestore
pi-851    [ 29]  [05]    96.790074617      ==========>
pi-851    [ 29]  [05]    96.790074617                      smp_apic_timer_interrupt() {
```

A primeira coluna exibe o identificador do processo. A segunda coluna a prioridade do processo. Na representação interna do kernel do Linux, o sistema possui 140 prioridades, sendo 0 a maior prioridade e 139 a menor prioridade. A terceira colula exibe o processador em que a tarefa está executando. A quarta coluna exibe o instante de tempo em que a entrada de *trace* aconteceu. A quinta coluna exibe, ou o tempo de execução de uma função ou as sinalizações ——> e =====>, que representam o início e o fim das execuções das chamadas do sistema e das interrupções, respectivamente. Por fim, a partir da sexta coluna é exibida a informação de *trace* do sistema.

Durante a execução da tarefa *pi* um *spinlock* que desabilita as interrup-

ções é chamado dentro do tratamento do RT Mutex. Após liberar o *spinlock* e habilitar as interrupções, o processador inicia a execução do tratador de interrupção do *timer*, que é a função *smp_apic_timer_interrupt*. Neste caso, o tratador de interrupção pode ter ocorrido em qualquer momento durante a seção em que as interrupções locais estavam desabilitadas, desta forma, é possível afirmar que o tratador de interrupção sofreu um atraso de ativação, porém, não é possível determinar o tempo exato do atraso de ativação. Desta forma, é seguro supor o pior caso: que a interrupção ocorreu logo após a tarefa *pi* desabilitar as interrupções.

```
[...]
pi-851    [ 29]  [05]    96.790085103                          _raw_spin_lock_irqsave() {
pi-851    [ 29]  [05]    96.790085750                            lock_acquire: ffffffff82049890 &tk->lock
pi-851    [ 29]  [05]    96.790086461                            lock_acquired: ffffffff82049890 &tk->lock
pi-851    [ 29]  [05]    96.790087185      1.704 us            } _raw_spin_lock_irqsave ()
```

Continuando a execução, durante o tratador de interrupção o código executa a função *_raw_spin_lock_irqsave*, a qual deveria desabilitar as interrupções do processador. Porém, como as interrupções são desabilitadas pelo próprio processador ao chamar um tratador de interrupção, o *trace* identifica que as interrupções já estão desabilitadas, e por isto não imprime a linha informando a troca de estado das interrupções.

```
[...]
pi-851    [ 29]  [05]    96.790126850                            softirq_raise: vec=1 [action=TIMER]
[...]
pi-851    [ 29]  [05]    96.790135125    + 60.143 us          } smp_apic_timer_interrupt ()
pi-851    [ 29]  [05]    96.790135125      <==========
```

Durante a execução, o tratador de interrupção acorda a *softirq* do *timer* e termina a sua execução, retornando o controle ao tarefa anterior.

```
pi-851    [ 29]  [05]    96.790135834                          sched_preempt_enable:  at _raw_spin_unlock_irqrestore
pi-851    [ 29]  [05]    96.790136103    + 63.643 us        } _raw_spin_unlock_irqrestore ()
```

Por fim, ao retornar o controle a tarefa *pi*, a tarefa *pi* continua a execução da função que libera o *spinlock*, reabilitando a preempção e retornando a função que libera o *spinlock*.

De maneira geral, pode-se dizer que um tratador de interrupção pode sofrer o atraso de ativação sempre que as interrupções estão desabilitadas, seja por uma ação do hardware ou do sistema operacional. Assumindo que um tratador de interrupção não mascarável não pode interromper outro tratador de interrupção não mascarável e também descartando problemas de lógica, como divisão por zero que podem gerar uma exceção, o comportamento de um tratador de interrupção não mascarável é caracterizado na Figura 24

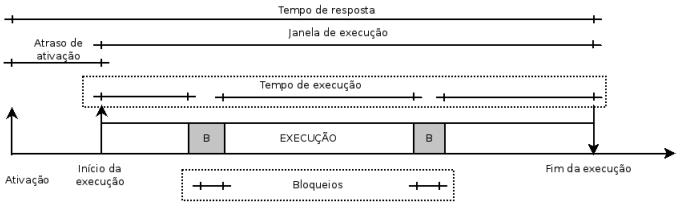

Figura 24 – Caracterização da execução de interrupção não mascarável

Sendo que o atraso de ativação pode acontecer caso o sistema já esteja atendendo outra interrupção não mascarável. Os bloqueios são todos implementados com espera ocupada, como com os *spinlocks*.

Como os tratadores de interrupção mascaráveis podem sofrer interferências dos tratadores de interrupção não mascaráveis, a sua caracterização se difere dos tratadores de interrupção não mascaráveis pela possibilidade de interferência dos tratadores de interrupção não mascaráveis. A caracterização dos tratadores de interrupção mascaráveis é exibida na Figura 25.

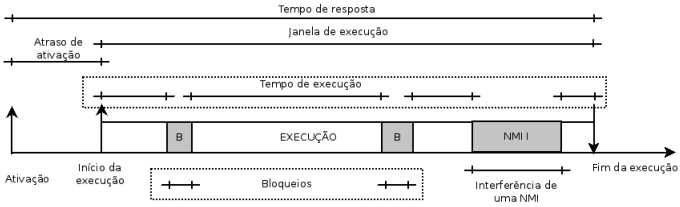

Figura 25 – Caracterização da execução de interrupção mascarável

No caso das interrupções mascaráveis, o atraso de ativação ocorre sempre que o processador estiver com as interrupções mascaráveis mascaradas. O que pode acontecer tanto pelo controle do Linux, quanto pelo controle do processador. Para identificar em que situações o processador desabilita as interrupções mascaráveis é necessário ler a documentação do processador.

6.2.2 Caracterização das *Threads* no Linux

Para caracterizar a execução de uma tarefa de tempo real, com a característica comum na literatura de que uma tarefa de tempo real não suspende voluntariamente durante a sua execução, utilizou-se a tarefa *pi*, descrita na seção 6.1.2. O código em C do *loop* de execução da tarefa *pi* é apresentado a seguir:

```
for(;;) {
```

```
        pause();

        if (read(ti->devfd, buff, 1024) < 0)
                exit(0);
}
```

Dentro do *loop* infinito criado com o *for*, a chamada *pause* suspende a execução da tarefa, esperando pela ativação de um novo *job*. A ativação do novo *job* é dada por um sinal periódico do *timer*, que faz a função pause retornar. Ao retornar a execução, a tarefa executa a operação de leitura do módulo do *kernel*, que ativa a função de espera ocupada protegida por uma variável de exclusão mútua no *kernel*. Após a leitura do módulo, a tarefa finaliza sua ativação suspendendo a execução a espera de um novo sinal que libere um novo *job*.

Em uma demonstração do *trace* da execução de uma ativação da tarefa *pi*, podemos descrever o comportamento de uma tarefa de tempo real. A seguir é dada esta demonstração do *trace* da tarefa *pi*, com comentários descrevendo cada etapa da execução da tarefa de tempo real.

```
 ksoftir-33  [ 98]  [04]    83.910527762                    sched_migrate_task: comm=pi pid=838 prio=9 \
                                                            orig_cpu=4 dest_cpu=0
 ksoftir-33  [ 98]  [04]    83.910529655                    sched_wakeup: comm=pi pid=838 prio=9 success=1 \
                                                            target_cpu=000
   [...]
  <idle>-0   [120]  [00]    83.910546624                  sched_preempt_enable:  at cpu_idle
  <idle>-0   [120]  [00]    83.910547023                  __schedule() {
   [...]
  <idle>-0   [120]  [00]    83.910551519                    sched_switch: prev_comm=swapper/0 prev_pid=0 \
                                                            prev_prio=120 prev_state=R+ ==> next_comm=pi \
                                                            next_pid=838 next_prio=9
```

A *softirq* do *timer* que irá ativar a tarefa *pi*, antes de ativá-la, a migra para a CPU 0, que está em *idle*. Após migrar, ativa a *thread pi*. No código do *kernel*, a tarefa *idle* é implementada com a preempção desabilitada, quando a CPU recebe o evento que uma tarefa foi ativada, esta sai da sua rotina de espera e, ao habilitar a preempção, executa o escalonador, efetuando a troca de contexto, do *idle* para a tarefa *pi*.

```
------------------------------------------
 0)    <idle>-0    =>     pi-838
 ------------------------------------------

    pi-838    [  9]  [00]    83.910555643    + 57.470 us       } schedule ()
    pi-838    [  9]  [00]    83.910555954    + 57.987 us     } sys_pause ()
    pi-838    [  9]  [00]    83.910555954      <----------
    pi-838    [  9]  [00]    83.910557267      ---------->
    pi-838    [  9]  [00]    83.910557267                  do_notify_resume() {
    [...]
    pi-838    [  9]  [00]    83.910578502                    softirq_raise: vec=8 [action=HRTIMER]
    [...]
    pi-838    [  9]  [00]    83.910583716                    sched_wakeup: comm=ksoftirqd/0 pid=3 prio=98 \
                                                             success=1 target_cpu=000
    [...]
    pi-838    [  9]  [00]    83.910605575    + 48.043 us     } do_notify_resume ()
    pi-838    [  9]  [00]    83.910605575      <----------
    pi-838    [  9]  [00]    83.910606705      ---------->
    pi-838    [  9]  [00]    83.910606705                  sys_rt_sigreturn() {
    [...]
    pi-838    [  9]  [00]    83.910610888      4.021 us      } sys_rt_sigreturn ()
    pi-838    [  9]  [00]    83.910610888      <----------
```

Após ser ativada, a rotina do escalonador que havia posto a tarefa para dormir é executada, a fim de devolver o controle para a tarefa, retornando a execução da chamada do *pause*. No retorno da chamada *pause* para o espaço do usuário, são chamadas as funções que fazem o tratamento do sinal do *timer* que ativou a tarefa, no exemplo estas são as funções *do_notify_resume* e *sys_rt_sigreturn*. Apesar de não aparecerem no código em C do programa no espaço do usuário, estas chamadas são executadas pela biblioteca C, na implementação do tratamento do sinal.

```
pi-838    [ 9]  [00]    83.910612437      ---------->
pi-838    [ 9]  [00]    83.910612437                      sys_read() {
pi-838    [ 9]  [00]    83.910613717                        run_higher in: c = 1 inc = 11287
pi-838    [ 9]  [00]    83.910613955                        _raw_spin_lock() {
pi-838    [ 9]  [00]    83.910614175                          sched_preempt_disable: at _raw_spin_lock
pi-838    [ 9]  [00]    83.910614368                          lock_acquire: ffffffffa02d6578 &daniel_lock
pi-838    [ 9]  [00]    83.910614750                          lock_acquired: ffffffffa02d6578 &daniel_lock
pi-838    [ 9]  [00]    83.910615092      0.989 us          } _raw_spin_lock ()
pi-838    [ 9]  [00]    83.910616591                        _raw_spin_unlock() {
pi-838    [ 9]  [00]    83.910616786                          lock_release: ffffffffa02d6578 &daniel_lock
pi-838    [ 9]  [00]    83.910617153                          sched_preempt_enable:  at _raw_spin_unlock
pi-838    [ 9]  [00]    83.910617315      0.571 us          } _raw_spin_unlock ()
pi-838    [ 9]  [00]    83.910617648                        run_higher out: c = 1 inc = 11288
pi-838    [ 9]  [00]    83.910618197      5.609 us        } sys_read ()
pi-838    [ 9]  [00]    83.910618197      <----------
```

A chamada *read* é executada, esta adquire o *spinlock* compartilhado, executa espera ocupada dentro da seção critica, libera o *spinlock* e retorna ao espaço do usuário.

```
pi-838    [ 9]  [00]    83.910618939      ---------->
pi-838    [ 9]  [00]    83.910618939                      sys_pause() {
pi-838    [ 9]  [00]    83.910619150                        schedule() {
[...]
pi-838    [ 9]  [00]    83.910629804                            sched_switch: prev_comm=pi prev_pid=838 \
                                                              prev_prio=9 prev_state=S ==> \
                                                              next_comm=ksoftirqd/0 next_pid=3 \
                                                              next_prio=98
```

Após retornar, a tarefa *pi* chama a função *pause*, suspendendo a execução da *thread*. Após suspender, chama o escalonador, que remove a tarefa *pi* do processador.

Na análise do *trace*, cada ativação da tarefa no contexto de processo foi dividida em partes, as quais auxiliam no entendimento e na modelagem dos tempos da tarefa de tempo real. Estas partes são descritas na Figura 26:

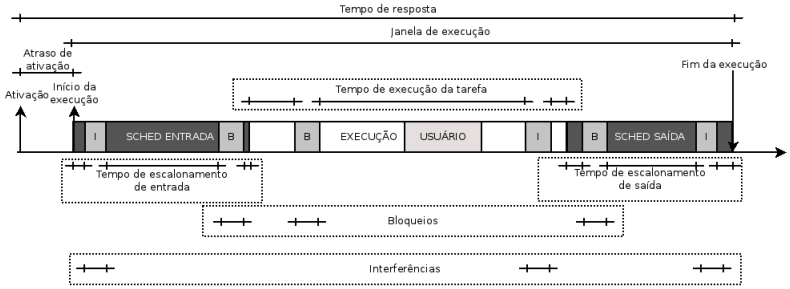

Figura 26 – Caracterização da execução de uma *thread* no Linux

Na Figura 26, como no método de tempo de resposta, temos o atraso de ativação, que é o tempo entre a tarefa ser ativada e a sua execução ser iniciada. O atraso de ativação pode ser afetado pela preempção do processador e pelo tempo de escalonamento de saída da tarefa corrente, ambos no processador em que a tarefa foi ativada.

Após a tarefa iniciar a sua execução, a rotina de escalonamento que havia suspendido a tarefa é executada, até retornar ao fluxo de execução da aplicação. A este tempo de execução, denominamos tempo de escalonamento de entrada. Durante o escalonamento de entrada, a aplicação pode sofrer bloqueios e interrupções.

Quando a tarefa retorna ao seu fluxo de execução, no caso da tarefa de maior prioridade, esta pode sofrer interrupções dos tratadores de interrupções e bloqueios em seções protegidas, além de poder executar tanto no *kernel*, quanto no espaço do usuário. Após terminar a execução e suspender, a tarefa chama uma função que ativa o escalonador, dando início a etapa de escalonamento de saída, qual pode também sofrer interferências e bloqueios.

Além das interferência dos tratadores de interrupção, as tarefas também sofrem interferências de tarefas de alta prioridade no contexto de processo. No *trace* a seguir, tem-se um exemplo de interferência de processo.

```
 ksoftir-21  [ 98]  [2] 443775.167748229                        _raw_spin_unlock_irqrestore() {
 ksoftir-21  [ 98]  [2] 443775.167748549                          lock_release: ffff880094f94f70 &p->pi_lock
 ksoftir-21  [ 98]  [2] 443775.167749228                          local_irq_enable:  at \
                                                                  _raw_spin_unlock_irqrestore
 ksoftir-21  [ 98]  [2] 443775.167749726                          sched_preempt_enable:  at \
                                                                  _raw_spin_unlock_irqrestore
 ksoftir-21  [ 98]  [2] 443775.167750054                          __schedule() {
   [...]
 ksoftir-21  [ 98]  [2] 443775.167761219                            sched_switch: prev_comm=ksoftirqd/2 \
                                                                    prev_pid=21 prev_prio=98 prev_state=R+ \
                                                                    ==> next_comm=pi next_pid=1282 next_prio=29
 ksoftir-21  [ 98]  [2] 443775.167761474                            lock_release: ffff88009d6244d8 &rq->lock
-----------------------------------------
2)   ksoftir-21   =>    pi-1282
-----------------------------------------

   /* tarefa de maior prioridade executa */
   [...]

  pi-1282    [ 29]  [2] 443775.167849947                          sched_switch: prev_comm=pi prev_pid=1282 \
                                                                  prev_prio=29 prev_state=D ==> \
                                                                  next_comm=ksoftirqd/2 next_pid=21 next_prio=29
  pi-1282    [ 29]  [2] 443775.167850257                          lock_release: ffff88009d6244d8 &rq->lock
-----------------------------------------
2)    pi-1282     =>   ksoftir-21
-----------------------------------------

 ksoftir-21  [ 29]  [2] 443775.167851354                          lock_acquire: ffff88009d6244d8 &rq->lock
 ksoftir-21  [ 29]  [2] 443775.167851874                            _raw_spin_unlock_irq() {
 ksoftir-21  [ 29]  [2] 443775.167852294                              lock_release: ffff88009d6244d8 &rq->lock
 ksoftir-21  [ 29]  [2] 443775.167852890                              local_irq_enable:  at _raw_spin_unlock_irq
 ksoftir-21  [ 29]  [2] 443775.167853420      1.180 us              } _raw_spin_unlock_irq ()
 ksoftir-21  [ 29]  [2] 443775.167853897    ! 103.385 us           } __schedule ()
 ksoftir-21  [ 29]  [2] 443775.167854334    ! 105.826 us         } _raw_spin_unlock_irqrestore ()
```

Neste exemplo, a tarefa *ksoftir-21* está executando na CPU 2 com a preempção desabilitada. Após reabilitar a preempção, o sistema chama a

rotina de escalonamento para ceder a CPU à tarefa *pi*, que possui maior prioridade. É possível identificar que a tarefa de baixa prioridade deixa o processador de maneira involuntária ao verificar o estado em que a tarefa sofreu a troca de contexto: o estado R, que significa que a tarefa estava apta a executar. Após a tarefa de alta prioridade executar até bloquear, o que pode ser verificado pelo estado que esta deixa o processador: D, a tarefa *ksoftir-21* retorna a sua execução.

O comportamento exibido pela tarefa é semelhante ao demonstrado na Figura 27, onde a chamada do escalonador é feita dentro janela de execução da tarefa de baixa prioridade, a troca de contexto é executada e, após a tarefa de maior prioridade executar, é feita a troca de contexto para a tarefa de baixa prioridade, que retorna a execução. Ao retornar, a função do escalonamento de entrada é executada, até retornar para o código da aplicação.

Para uma tarefa de baixa prioridade, que pode sofrer interferência de uma tarefa de alta prioridade, o exemplo da Figura 26 também é válido, porém, poderão haver diversas execuções do exemplo da Figura 26 em uma única ativação, como na Figura 27

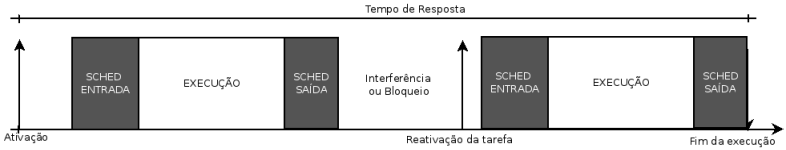

Figura 27 – Caracterização da execução com bloqueio ou interferência

O exemplo da Figura 27, também é válido quando se utiliza métodos de exclusão mútua que suspendem a execução da tarefa, como os semáforos e os mutex. Nestes casos, quando a tarefa suspende esperando por um recurso, haverá duas, ou mais dependendo do número de vezes que a tarefa suspendeu, ocorrências da execução no *trace*. Porém, as execuções farão parte da mesma ativação. Quando uma tarefa de alta prioridade suspende à espera de um recurso ocupado por uma tarefa de baixa prioridade, o tempo que a tarefa ficou suspensa se caracteriza como bloqueio. Já quando uma tarefa de baixa prioridade suspende a espera de um recurso ocupado por uma tarefa de alta prioridade, o tempo que a tarefa ficou suspensa se caracteriza como interferência.

Na próxima seção, são descritas as variáveis utilizadas no modelo de tempo de resposta. Após é descrito como encontrar os eventos que influenciaram nas execução temporal das tarefas de tempo real a partir do *trace* e, a partir destes eventos, como definir os valores para as variáveis comuns na análise de tempo de resposta para as tarefas de tempo real no Linux.

6.3 MÉTRICAS RELEVANTES

Durante as próximas seções, serão descritas as formas de encontrar os pontos de *trace* de interesse, e a partir destes pontos, como definir as variáveis temporais de interesse e o valor destas variáveis, relevantes a ativação de uma tarefa de tempo real no Linux. Vale esclarecer que o objetivo desta seção não é definir o modelo de execução completo do Linux. Também não é objetivo adaptar o Linux para o modelo clássico de tempo de resposta. O objetivo das próximas seções é demonstrar como a ferramenta de *trace* proposta pode auxiliar no entendimento temporal da execução das tarefas de tempo real do Linux e como os valores da execução da aplicação de tempo real de exemplo foram interpretados a fim caracterizar a execução de tarefas no Linux PREEMPT-RT. No futuro estas informações podem servir de base para um modelo analítico de análise de escalonabilidade tempo real.

6.3.1 Variáveis da análise de tempo de resposta

Na seção 2.2, foi apresentado o modelo de análise de tempo de resposta, que é definido utilizando as seguintes variáveis:

- P: Período de execução da tarefa;
- J: Atraso de ativação;
- B: Tempo de bloqueio;
- C: Tempo de computação;
- $W_{hp(i)}$: A interferência da janela de execução de tarefas de maior prioridade.

O valor de P é definido no projeto, dependendo das necessidades temporais de cada tarefa, por isto não será tratado. O valor de $W_{hp(i)}$ é a interferência das janelas de execução das tarefas de maior prioridade, e será tratado aqui como interferência. Deste modo, os valores que se deseja obter a partir da análise do trace são:

- J: O atraso provocados pelos mecanismos que interferem na ativação de uma tarefa;
- B: Identificar os tempos de bloqueio e a sua interferência no tempo de resposta;

- C: O tempo de execução de uma ativação de uma tarefa;
- I: A interferência das tarefas de maior prioridade.

Utilizando estas variáveis como referência, as próximas seções descrevem como obter estas variáveis para o modelo de tarefas de tempo real do Linux com o PREEMPT-RT e como definir os seus valores para uma ativação.

6.3.2 Atraso de ativação

Segundo a definição no modelo de tempo de resposta, descrita na seção 2, o atraso de ativação é o tempo entre a ativação, a_i, da tarefa e o início de sua execução, s_i. O atraso de ativação pode acontecer para os dois contextos de execução, o contexto das interrupções e dos processos, ambos detalhados a seguir.

6.3.2.1 Atraso de ativação das interrupções

A ativação de uma tarefa no contexto das interrupções pode acontecer a qualquer momento, independente da prioridade da tarefa com contexto de processo que esteja executando no sistema. Porém, existe uma forma de atrasar a ativação de um tratador de interrupção: ao desabilitar as interrupções, o que pode ser feito tanto pelo sistema operacional quanto pelo próprio processador.

Quando o sistema está com as interrupções desabilitadas, a entrega das interrupções e o início da execução dos tratadores de interrupção podem ser atrasados. Para identificar este comportamento no *trace* para os casos em que o Linux desabilita as interrupções, basta achar pontos onde uma interrupção é atendida logo após a ativação das interrupções. No trace, isto é identificável com a sinalização
"==========>"logo após o *tracepoint local_irq_enable* ou *enable_irq*. Um exemplo desta ocorrência em uma saída da ferramenta de trace:

```
  ksoftir-3   [ 98]  [0]  2395.451462911                    _raw_spin_lock_irqsave() {
  ksoftir-3   [ 98]  [0]  2395.451463276                      local_irq_disable:  at _raw_spin_lock_irqsave
  ksoftir-3   [ 98]  [0]  2395.451463613                      sched_preempt_disable: at _raw_spin_lock_irqsave
  ksoftir-3   [ 98]  [0]  2395.451463908                      lock_acquire: ffff880091b08730 &p->pi_lock
  ksoftir-3   [ 98]  [0]  2395.451464426                      lock_acquired: ffff880091b08730 &p->pi_lock
  ksoftir-3   [ 98]  [0]  2395.451465031      1.764 us        } _raw_spin_lock_irqsave ()
[...]
  ksoftir-3   [ 98]  [0]  2395.451477263                    sched_wakeup: comm=pi pid=929 prio=29 \
                                                            success=1 target_cpu=002
[...]
  ksoftir-3   [ 98]  [0]  2395.451491728                    _raw_spin_unlock_irqrestore() {
  ksoftir-3   [ 98]  [0]  2395.451492227                      lock_release: ffff880091b08730 &p->pi_lock
  ksoftir-3   [ 98]  [0]  2395.451492926                      local_irq_enable:  at _raw_spin_unlock_irqrestore
  ksoftir-3   [ 98]  [0]  2395.451495651      ==========>
  ksoftir-3   [ 98]  [0]  2395.451495651                      smp_apic_timer_interrupt() {
```

A interrupção, que ativa o tratador de interrupção implementado na função *smp_apic_timer_interrupt*, ocorreu em um momento em que o sistema estava com as interrupções desabilitadas, devido a execução da função *_raw_spin_lock_irqsave*. Não é possível afirmar o exato instante de tempo que a interrupção ocorreu, assim, não é possível definir o tempo exato do atraso de ativação, pois a ativação da interrupção pode ter ocorrido a qualquer momento entre a desativação e a ativação das interrupções. Porém, como para a análise de tempo real o fator de interesse é o pior caso, é seguro assumir que o instante de tempo da ativação da tarefa foi imediatamente após as interrupções serem desabilitadas.

Se definirmos di_i como o instante de tempo absoluto em que a interrupção foi desabilitada, ei_i como o instante de tempo absoluto em que a interrupção foi reabilitada, e *id* como uma variável binária, com valor 0 quando a interrupção ocorre com as interrupções habilitadas e 1 quando a interrupção ocorre com as interrupções desabilitadas, podemos dizer que no pior caso, o atraso de ativação rji_i de uma interrupção é definido pela Equação 6.1.

$$rji_i = \begin{cases} ei_i - di_i & \text{se } id = 1 \\ \text{senão } 0 \end{cases} \tag{6.1}$$

Utilizando os tempos do exemplo de *trace*, tem-se os seguintes valores:

$$\begin{aligned} di_i &= 2395.451463276 \\ ei_i &= 2395.451492926 \\ id &= 1 \\ rji_i &= 2395.451492926 - 2395.451463276 \\ rji_i &= 0.000029650 \\ rji_i &= 29.650us \end{aligned} \tag{6.2}$$

Conforme demonstrado na Equação 6.2, o atraso de ativação do exemplo de *trace* foi de 29.650*us*.

A Equação 6.1 também pode ser utilizada para os casos onde o processador desabilita as interrupções. Mapeando o valor das variáveis di_i e ei_i para os pontos em que o processador desabilitou e reabilitou as interrupções, respectivamente. Para identificar os casos onde o processador desabilita as interrupções sem a intervenção do Linux, é preciso entender o modelo de execução do processador. A partir do modelo de execução do processador é possível definir os pontos em que as ações do processador desabilitam e

reabilitam as interrupções. Por exemplo, ao atender uma interrupção mascarável ou não mascarável, o processador Intel desabilita as interrupções mascaráveis. Caso um tratador de uma interrupção mascarável seja acionado logo após o retorno de um tratador de interrupção mascarável ou não mascarável, é seguro assumir que o di_i para a segunda interrupção é o instante de tempo que o primeiro tratador foi acionado e que o valor de ei_i é o instante de tempo que o primeiro tratador de interrupção retornou.

Os métodos do Linux para desabilitar as interrupções não são válidos para as interrupções não mascaráveis. Isto é, não é possível mascarar as interrupções não mascaráveis pelo sistema operacional, por isto, o sistema operacional não pode atrasar, diretamente, a entrega de uma interrupção não mascarável. Porém, o processador pode atrasar a entrega de uma interrupção não mascarável. Na arquitetura Intel, uma interrupção não mascarável não pode ser atendida se o processador estiver atendendo outra interrupção não mascarável. Logo, uma interrupção não mascarável pode sofrer atraso de ativação de outra interrupção não mascarável. Neste caso, se um tratador de interrupção não mascarável seja chamado logo após o retorno de um outro tratador de interrupção não mascarável, é seguro admitir que para o segundo tratador o valor de di_i é o instante de tempo que o primeiro tratador de interrupção não mascarável foi acionado e que o valor de ei_i é o instante de tempo que o primeiro tratador de interrupção não mascarável retornou.

6.3.2.2 Atraso de ativação para as *threads*

Para as tarefas que são *threads*, a ativação acontece quando o seu estado passa de suspensa para pronta. Porém, após ser ativada, a tarefa tem que ser escalonada, para a partir disto, iniciar a sua execução. O atraso de ativação de uma *thread* é então o atraso de escalonamento da tarefa. O atraso de escalonamento de uma *thread* pode ser afetado de duas formas: pelo controle de preempção na CPU em que a *thread* foi ativada e pela execução da rotina de escalonamento. A forma de computar estes tempos é descritas as seguir.

Para encontrar os pontos de atraso de ativação de preempção para determinada tarefa com contexto de processo, deve-se procurar pontos onde houveram ativações de *threads* de alta prioridade, com o *tracepoint sched_wakeup*, em seções que a preempção estava desabilitada no processador em que a tarefa foi ativada. Os pontos onde a preempção está desabilitada em determinado processador é o tempo entre a ocorrência do *tracepoint sched_preempt_disable* e do *tracepoint sched_preempt_enable*, como no exemplo a seguir:

```
<idle>-0    [120]  [2] 57402.202940259                    sched_preempt_disable: at cpu_idle
   [...]
<idle>-0    [120]  [2] 57402.203030705      ==========>
<idle>-0    [120]  [2] 57402.203030705                    do_IRQ() {
```

```
   [...]
<idle>-0   [120]  [2] 57402.203105352                    sched_wakeup: comm=irq/11-uhci_hcd pid=73 prio=49 \
                                                         success=1 target_cpu=002
   [...]
<idle>-0   [120]  [2] 57402.203152298    ! 121.102 us    } do_IRQ ()
<idle>-0   [120]  [2] 57402.203152298      <==========
   [...]
<idle>-0   [120]  [2] 57402.203188965                    sched_preempt_enable:  at cpu_idle
<idle>-0   [120]  [2] 57402.203190543                    __schedule() {
   [...]
<idle>-0   [120]  [2] 57402.203198124                      sched_switch: prev_comm=swapper/2 prev_pid=0 \
                                                           prev_prio=120 prev_state=R+ ==> \
                                                           next_comm=irq/11-uhci_hcd next_pid=73 \
                                                           next_prio=49

-----------------------------------------
2)    <idle>-0    =>   irq/11--73
-----------------------------------------
```

As linhas com '[...]' foram linhas removidas para deixar o exemplo mais claro. No exemplo, a CPU 2 está executando a tarefa *idle*. A tarefa *idle* executa quando não existem tarefas prontas para executar. Durante a execução da tarefa *idle*, são executadas algumas rotinas de manutenção do sistema e, após isto, o sistema desabilita a preempção e o processador vai para um estado de espera. No *trace* de exemplo, durante o estado de espera, acontece uma interrupção de dispositivo, que é atendida pelo tratador *do_IRQ*. O tratador *do_IRQ* no PREEMPT_RT tem uma única função: ativar a *thread* do *kernel* que atende determinada interrupção. Neste caso, a tarefa *irq/11-uhci_hcd* foi ativada. Após acordar a tarefa, caso o sistema estivesse com a preempção habilitada, o fim do tratador de interrupção iria chamar uma função *schedule*. Porém, como a preempção estava desabilitada, esta chamada foi postergada, para o fim da rotina que tira o processador do estado *idle* reabilitando a preempção.

O caso quando o controle de preempção afeta a ativação de uma tarefa será denominado atraso de preempção. Para calcular o tempo de atraso de preempção de uma *thread*, define-se a variável binária pd, com valor 1 caso a preempção esteja desabilitada no momento em que a tarefa foi acordada, ou, 0 quando a preempção está habilitada no momento em que a tarefa foi acordada. Para o caso onde a preempção está desabilitada, define-se sw_i como o instante de tempo absoluto em que a tarefa foi ativada, ep_i como o instante de tempo absoluto em que a preempção foi reabilitada. A partir destas definições, é possível descrever o atraso de preempção pj_i com a Equação 6.3.

$$pj_i = \begin{cases} ep_i - sw_i & \text{se } pd = 1 \text{ e tarefa de maior prioridade} \\ \text{senão } 0 \end{cases} \tag{6.3}$$

No exemplo de *trace* que demonstra o atraso de preempção, tem-se os seguintes valores:

$$\begin{aligned} sw_i &= 57402.203105352 \\ ep_i &= 57402.203188965 \\ pj_i &= 57402.203188965 - 57402.203105352 \\ pj_i &= 0.000083613 \\ pj_i &= 83.613us \end{aligned} \tag{6.4}$$

Além do atraso da preempção, a tarefa também sofre o atraso de escalonamento. O atraso de escalonamento é o tempo de execução da rotina de escalonamento responsável por remover a tarefa de baixa prioridade, para iniciar a execução da tarefa de alta prioridade.

Para definir o atraso de escalonamento, é preciso definir a variável de escalonamento permitido se_i, como o instante de tempo absoluto em que o escalonamento estava permitido na CPU que o processo foi escalonado. Este pode ser o tempo em que a tarefa de alta prioridade foi ativada, já definida na variável sw_i, quando a preempção do processador estiver habilitada. Ou ainda com o tempo em que a preempção foi reabilitada, já definida na variável ep_i, quando a preempção está desabilitada. Utilizando a variável pd para distinguir os casos, pode-se definir o valor da variável se_i conforme a Equação 6.5.

$$se_i = \begin{cases} sw_i & \text{se } pd = 0 \\ \text{senão } ep_i \end{cases} \tag{6.5}$$

Definindo a variável cw_i como o instante de tempo absoluto da troca de contexto da tarefa de baixa prioridade para a tarefa de alta prioridade, pode-se definir o atraso de escalonamento sj_i com a Equação 6.6.

$$sj_i = cw_i - se_i \tag{6.6}$$

Na Equação 6.8, é calculado o tempo de atraso de escalonamento do exemplo de *trace* demonstrado no início desta seção.

$$\begin{aligned} pd &= 1 \\ se_i &= ep_i \\ ep_i &= 57402.203188965 \\ cw_i &= 57402.203198124 \\ sj_i &= 57402.203198124 - 57402.203188965 \\ sj_i &= 0.000009159 \\ sj_i &= 9.159us \end{aligned} \tag{6.7}$$

A partir da definição do atraso de preempção e do atraso de escalonamento, é possível definir o atraso de ativação de um processo, rjp_i, com a Equação 6.8.

$$rjp_i = sj_i + pj_i \tag{6.8}$$

Na Equação 6.9, é calculado o atraso de ativação do processo no exemplo de *trace* demonstrado no início desta seção.

$$rjp_i = 83.613 + 7.581 = 91.194us \tag{6.9}$$

Com as definições do atraso de ativação de interrupções e das *threads*, pode-se definir, de maneira genérica na Equação 6.10, o valor da variável J_i, cobrindo ambos os casos.

$$J_i = \begin{cases} rjp_i & \text{se for uma } \textit{thread} \\ rji_i & \text{se for uma interrupção} \end{cases} \tag{6.10}$$

6.3.3 Bloqueio

A variável B_i, utilizada na análise de tempo de resposta, é o tempo máximo de bloqueio que uma determinada tarefa pode receber em uma ativação. Cada algoritmo de exclusão mútua, utilizado em sistemas de tempo real, deve prover um método analítico para se calcular o valor de B_i para uma determinada tarefa.

Apesar do Linux implementar métodos de exclusão mútua com características de tempo real, não é possível determinar, analiticamente, o tempo

de bloqueio que uma tarefa pode receber em uma ativação. Outro fato, é que uma tarefa pode bloquear em diversos pontos, com métodos diferentes de exclusão mútua, o que dificulta a análise. Apesar de não ser possível calcular o valor de bloqueio para cada tarefa, com a ferramenta de *trace* proposta é possível visualizar quando e por quanto tempo uma tarefa sofreu bloqueio.

É possível localizar quando uma tarefa sofreu bloqueio procurando pelo *tracepoint lock_contended*. Este *tracepoint* sempre ocorre dentro de uma função que implementa exclusão mútua. Em um exemplo com *spinlock*:

```
ksoftir-21  [ 98]  [2]  2395.481344198                    _raw_spin_lock() {
ksoftir-21  [ 98]  [2]  2395.481344732                      lock_acquire: ffff88009ce244d8 &rq->lock
ksoftir-21  [ 98]  [2]  2395.481345330                      lock_acquired: ffff88009ce244d8 &rq->lock
ksoftir-21  [ 98]  [2]  2395.481345865      1.321 us      } _raw_spin_lock ()
ksoftir-15  [ 98]  [1]  2395.481346021                    _raw_spin_lock() {
ksoftir-15  [ 98]  [1]  2395.481346384                      lock_acquire: ffff88009ce244d8 &rq->lock
ksoftir-15  [ 98]  [1]  2395.481346884                      lock_contended: ffff88009ce244d8 &rq->lock
ksoftir-21  [ 98]  [2]  2395.481355955                    _raw_spin_unlock() {
ksoftir-21  [ 98]  [2]  2395.481356424                      lock_release: ffff88009ce244d8 &rq->lock
ksoftir-21  [ 98]  [2]  2395.481357181      0.897 us      } _raw_spin_unlock ()
ksoftir-15  [ 98]  [1]  2395.481357734                      lock_acquired: ffff88009ce244d8 &rq->lock
ksoftir-15  [ 98]  [1]  2395.481358545    + 12.198 us     } _raw_spin_lock ()
```

Neste exemplo, duas *threads* do *kernel* que executam as *softirqs* disputam o *spinlock rq->lock*. Como é possível que variáveis de exclusão mútua tenham o mesmo nome, é necessário analisar também o endereço da variável, para determinar se as tarefas estão disputando a mesma variável, neste caso o endereço é o mesmo em ambas a *threads*: *ffff88009ce244d8*.

A tarefa *ksoftir-21* adquire o *spinlock*, que estava livre, exibido na primeira execução da função *_raw_spin_lock*. Na segunda execução da função *_raw_spin_lock*, a *thread ksoftir-15* tenta adquirir a mesma variável de exclusão mútua, e então sofre contenção, identificada pelo *tracepoint lock_contended*.

Após a contenção, a *thread ksoftir-21* executa a função *_raw_spin_unlock*, que libera o *lock*, deixando a *thread ksoftir-15* adquirir o *lock*.

Na última linha, no retorno da função *_raw_spin_lock* que sofreu contenção, é exibido o tempo que a função ficou esperando o *spinlock*: 12.198 *us*, em comparação com os 1.321 *us* que a função que não sofreu o bloqueio demorou para adquirir a variável de exclusão mútua.

Durante a execução, uma tarefa pode bloquear em diversos pontos. Desta forma, para cada ativação da tarefa pode-se ter j pontos de bloqueio. Outro fato é que os métodos de exclusão mútua de tempo real, como os RT Mutex, utilizam *spinlocks* em sua implementação, podendo haver bloqueio dentro do bloqueio. Nestes casos, de exclusão mútua aninhada, irá se considerar apenas o método de exclusão mútua que iniciou o encadeamento de chamadas.

Um complicador vem do fato das interrupções poderem acontecer dentro da execução das funções de exclusão mútua, até mesmo quando a tarefa está bloqueada. Nestes casos, o tempo de interrupção deve ser descontado do tempo de bloqueio. Outra forma de interferência vem do fato de alguns

protocolos de exclusão mútua serem preemptivos, logo, as tarefas podem sofrer interferência de outras *threads* durante a aquisição ou contenção de uma variável de exclusão mútua.

Outro complicador são os métodos de exclusão mútua que suspendem o processo. Ao suspender o processo, a função *schedule* é chamada dentro da função que implementa o mecanismo de exclusão mútua, e a tarefa é suspensa com estado ininterruptível. Ao retornar a execução, a tarefa adquire a variável de exclusão mútua e continua a execução.

Para simplificar a análise, os tempos de bloqueio serão medidos pela diferença de tempo entre o instante de tempo absoluto do início e do fim da execução da função que adquire a variável de exclusão mútua. Isto inclui o *overhead* causado pelos controles, como o de inversão de prioridade e escalonamento, que só ocorrem quando há o bloqueio.

Para calcular estes tempos, é necessário definir a variável bs_{ji} como o instante de tempo absoluto em que a função de exclusão mútua foi chamada, pela j-tésima vez na mesma ativação, para adquirir uma variável de exclusão mútua. Define-se também bf_{ji} com o instante tempo absoluto no qual a função de exclusão mútua retornou e a variável binária ct_{ji}, que possui o valor 1 quando a tarefa sofreu contenção ou 0 se a tarefa não sofreu contenção.

Desta forma, o tempo de bloqueio b_{ji} em determinada variável de exclusão mútua pode ser definido como o delta de tempo entre o início e o fim da execução da função que adquire a variável de exclusão mútua, quando há contenção.

Diferente do modelo de tempo de resposta, que assume que uma tarefa ao bloquear cede o processador para que a tarefa de menor prioridade execute, no Linux isto não é sempre verdade. Por exemplo os *spinlocks* que efetuam a espera ocupada, ou com os RT Mutex, que executam operações de controle antes de deixar o processador. Esta diferença faz com que seja possível que ocorram interferências durante o bloqueio. Desta forma, para não incluir o tempo de interferência ao bloqueio, é necessário remover a janela de execução das interferências do tempo de bloqueio. A forma de se definir as interferências serão definidas nas próximas seções, porém, para auxiliar nesta operação, definiu-se uma função auxiliar que retorna o conjunto de interferências que determinada tarefa sofreu dentro de um determinado intervalo de tempo. Esta função auxiliar é denotada por $int_i(start, finish)$.

Pode-se definir o tempo de bloqueio com a Equação 6.11.

$$b_{ji} = \begin{cases} (bf_{ji} - bs_{ji}) - \sum_{\forall x \in int_i(bs_{ji}, bf_{ji})} I_x & \text{se } ct_{ji} = 1; \\ \text{senão } 0. \end{cases} \tag{6.11}$$

Algo a salientar é que, só se considera bloqueio quando a tarefa sofre contenção causada por uma tarefa de igual ou menor prioridade, quando a tarefa sofre contenção por uma tarefa de maior prioridade, isto é considerado interferência. Um complicador é que, em alguns mecanismos de exclusão mútua, a escolha de qual tarefa irá adquirir determinada variável de exclusão mútua, não é baseada na prioridade, o que permite que uma interferência se torne bloqueio durante a execução. Por exemplo, uma tarefa de prioridade média, que sofreu contenção pela tarefa de alta prioridade, está disputando um *spinlock* com uma tarefa de baixa prioridade. No momento em que a tarefa de alta prioridade libera o *spinlock*, por causa da forma indeterminista de atribuir o *spinlock*, a tarefa de baixa prioridade adquire a variável de exclusão mútua, assim, o que era interferência, se torna bloqueio, durante a execução. Este controle pode ser feito analisando a prioridade da tarefa que liberou a variável de exclusão mútua, pelo *tracepoint lock_release* e, caso a variável seja adquirida por outra tarefa durante a contenção, analisando a prioridade do processo que adquiriu a variável de exclusão mútua pelo *tracepoint lock_aquired*.

6.3.3.1 Tempo de bloqueio da ativação de uma tarefa

O tempo de bloqueio B da ativação de uma tarefa, definido na Equação 6.12, é o somatório de todos os j bloqueios que ocorreram durante a ativação.

$$B_i = \sum_{n=1}^{j} b_{ni} \tag{6.12}$$

No Linux com o patch PREEMPT_RT, as tarefas podem executar em dois contextos: O contexto de interrupção e o contexto das *threads*. No *kernel* padrão do Linux as *softirqs*, que são um contexto, executam como *threads* do *kernel*, desta forma, executando no contexto das *threads*. Cada contexto apresenta uma forma de iniciar e finalizar uma ativação. O modo de identificar estas ativações, e suas características temporais, são descritos nas próximas seções.

6.3.4 Contexto das interrupções

No contexto das interrupções, o início e fim de execução do tratador de interrupção são o início e o fim de uma ativação de uma tarefa. Para

facilitar a identificação, cada vez que a função de um tratador de IRQ é executada, o marcador '==========>' é adicionado no *trace*, antes linha que exibe função chamada. Para sinalizar o fim da execução, o marcador '<==========' é adicionado após a função retornar de sua execução. Na linha que exibe o fim da função do tratador de interrupção, é exibido o tempo que o tratador levou para executar. O exemplo a seguir demonstra a execução de um tratador de execução.

```
  ksoftir-3   [ 98]  [0] 443775.151208921      ==========>
  ksoftir-3   [ 98]  [0] 443775.151208921                    smp_apic_timer_interrupt() {
  ksoftir-3   [ 98]  [0] 443775.151210611                      _raw_spin_lock() {
  ksoftir-3   [ 98]  [0] 443775.151210935                        lock_acquire: ffff88009cc0d838 &cpu_base->lock
  ksoftir-3   [ 98]  [0] 443775.151211540                        lock_acquired: ffff88009cc0d838 &cpu_base->lock
  ksoftir-3   [ 98]  [0] 443775.151212342      1.334 us        } _raw_spin_lock ()
[...]
  ksoftir-3   [ 98]  [0] 443775.151248619                      _raw_spin_unlock() {
  ksoftir-3   [ 98]  [0] 443775.151249162                        lock_release: ffff88009cc0d838 &cpu_base->lock
  ksoftir-3   [ 98]  [0] 443775.151249868      0.875 us        } _raw_spin_unlock ()
  ksoftir-3   [ 98]  [0] 443775.151253870    + 44.424 us       } smp_apic_timer_interrupt ()
  ksoftir-3   [ 98]  [0] 443775.151253870      <==========
```

Neste exemplo, a função *smp_apic_timer_interrupt* é executada por uma interrupção do *timer*. O retorno da função mostra o tempo de execução da função: 44.424 *us*. Nota-se que o valor do tempo de execução da função, demonstrado no retorno da função, é menor que o delta entre o instante de tempo do início e o instante de tempo da execução do fim da função, demonstrado na coluna do tempo absoluto. Isto ocorre porque os instantes de tempo utilizados para calcular o tempo de resposta da função não são os instantes de tempo exibido na coluna de tempo absoluto. Os tempos utilizados para calcular o tempo de execução da função são adquiridos durante a execução do *pluguin* de *trace*; Nesta dissertação, os valores utilizados são os valores de tempo absoluto.

Definindo a variável is_i como o instante de tempo absoluto de inicio da execução do tratador de interrupção, e a variável if_i como o instante de tempo absoluto do retorno do tratador de interrupção, pode-se definir a janela de execução para uma interrupção não mascarável, $iwtnmi_i$ como descrito na Equação 6.13.

$$iwtnmi_i = if_i - is_i \tag{6.13}$$

A definição da janela de execução das interrupções mascaráveis se difere da janela de execução das interrupções não mascaráveis pela possibilidade da ocorrência de interrupções dentro da sua janela de execução. Pois é possível que ocorram interrupções não mascaráveis durante a execução de uma interrupção mascarável. Desta forma, a definição da janela de execução para uma interrupção mascarável, $iwtmi_i$ é definida na Equação 6.14.

$$iwtmi_i = if_i - is_i - \sum_{\forall x \in int_i(is_i, if_i)} I_x \tag{6.14}$$

De maneira genérica, podemos definir a janela de execução para uma interrupção iwt_i com a Equação .

$$iwt_i = \begin{cases} iwtnmi_i & \text{se interrupção não mascarável;} \\ iwtmi_i & \text{caso contrário.} \end{cases} \tag{6.15}$$

A Equação 6.17 demonstra exemplo de interrupção exibida no *trace*:

$$\begin{aligned} is_i &= 443775.151208921 \\ if_i &= 443775.151253870 \\ iwtmi_i &= 443775.151253870 - 443775.151208921 - 0 \\ iwt_i &= iwtmi_i = 0.000044949 \\ iwt_i &= 44.949us \end{aligned} \tag{6.16}$$

Da janela de execução, ao extrair o tempo de bloqueio B_i que o tratador de interrupção sofreu, tem-se o tempo de execução do tratador de interrupção, ao qual podemos definir como iet_i na Equação 6.17.

$$iet_i = iwt_i - B_i \tag{6.17}$$

Uma característica importante é que, desde que as interrupções estejam habilitadas, uma interrupção executa independente de qual tarefa possui o processador, desta forma, ignorando as prioridades das tarefas, apresentando o comportamento da tarefa de maior prioridade do sistema. Os tratadores de interrupção executam sem mudar o contexto do processo em execução, por isto, os campos que descrevem o processo, como nome, *pid* e prioridade são do processo que o tratador de interrupção interrompeu.

6.3.4.1 Tempo de resposta de interrupção

O tempo de resposta de uma tarefa é a diferença entre o instante tempo que a tarefa foi ativada e o instante de tempo que a tarefa termina a sua execução. Desta forma, o tempo de resposta de uma tarefa no contexto de

interrupção é definido como a soma do atraso de ativação mais o delta entre a ativação e o retorno do tratador de interrupção. A definição analítica é exibida na Equação 6.18:

$$RI_i = rji_i + (if_i - is_i) \tag{6.18}$$

6.3.4.2 Interferência das interrupção em *threads*

Durante a ativação de uma tarefa com contexto das *threads*, a tarefa pode receber j interferências de tratadores de interrupção. Estes tratadores podem ter duas origens, uma origem externa à aplicação, por exemplo, um *timer* do sistema; ou uma origem interna da aplicação, por exemplo um *page fault*. No primeiro caso, a interrupção é uma interferência no fluxo de execução da tarefa, já no segundo, o tempo pode ser considerado parte do tempo de execução da tarefa. Para distinguir os dois tipos de interrupção, deve-se analisar o motivo da interrupção. Para descrever em forma de equação, atribui-se o valor 1 para a variável binária $procint_{ji}$ quando a interrupção executada faz parte do contexto do processo. Desta forma, a interferência ip da j-tésima interrupção sobre o tempo de resposta da ativação de uma determinada tarefa, é determinada pela Equação 6.19:

$$ip_{ji} = \begin{cases} iwt_{ji} & \text{se } procint_{ji} = 0 \\ 0 & \text{caso contrário.} \end{cases} \tag{6.19}$$

Com a definição acima, podemos contabilizar a interferência das interrupções, II, na ativação de uma tarefa, excluindo as interrupções causadas pelo próprio processo. Desta forma, a interferência de interrupções de um processo é determinada pela Equação 6.20:

$$II_i = \sum_{n=1}^{j} ip_{ni} \tag{6.20}$$

6.3.5 Contexto das *threads*

É difícil definir, de maneira geral, qual o modelo de execução de uma tarefa no Linux no contexto das *threads*. Isto porque o modelo de tarefa não apresenta as mesmas restrições dos modelos de tempo real. Por exemplo, em

sistemas de tempo real, costuma-se ter tarefas com períodos e ativações bem definidas, o que não acontece sempre no Linux. Porém, com a ferramenta de *trace* proposta, é possível observar o comportamento temporal das tarefas, e caso o comportamento da tarefa seja conhecido, é possível fazer as medições de tempo com a ideia de ativação da tarefa.

Para poder definir os tempos de resposta, janela de execução e de computação, se faz necessário entender como é feita a troca de contexto e escalonamento de tarefas, descritos na próxima seção.

6.3.5.1 Troca de contexto e escalonamento

A troca de contexto acontece tanto para processos no *kernel* quanto para processos no espaço do usuário, porém, esta troca é sempre ativada por rotinas dentro das rotinas de escalonamento do *kernel*. Logo, todas a trocas são exibidas pelo ferramenta de *trace*. As trocas de contextos podem ser identificadas pelo *tracepoint sched_switch*, que exibe as seguintes informações:

- *prev_comm*: Nome do processo que está deixando o processador;
- *prev_prio*: Prioridade do processo que está deixando o processador;
- *prev_state*: Estado da tarefa que está deixando o processador;
- *next_comm*: Nome do processo que irá obter o processador; e
- *next_prio*: Prioridade do processo que irá obter o processador.

Deste *tracepoint*, uma informação importante para a análise é a do estado da tarefa, que pode ser um dos seguintes:

- *R*: Pronta para executar (na fila de execução);
- *D*: Dormindo/Bloqueada em estado ininterruptível;
- *S*: Dormindo/Bloqueada em estado interruptível;
- *T*: Parada, por ter recebido um sinal em uma sessão de trace ou debug;
- *X*: Morta (dificilmente irá ocorrer);
- *Z*: Zumbi, uma tarefa que terminou mas não reportou seu estado para a tarefa pai.

Os principais estados são os três primeiros: pronta para executar, dormindo ininterruptível e dormindo interruptível. Quando uma tarefa deixa o processador nos estados S e D, a tarefa deixou o processador de maneira voluntária, isto é, a tarefa terminou a sua execução, ou foi bloqueada a espera de algum recurso, respectivamente. Já quando a tarefa deixa o processador no estado R, esta sofreu uma troca de contexto involuntária, pois esta tinha os recursos necessários para executar, porém, o escalonador decidiu alocar a CPU para outra tarefa.

Quando uma tarefa de tempo real deixa a CPU de forma involuntária, a tarefa irá sofrer uma interferência, pois, deixou a CPU para uma tarefa de igual ou maior prioridade executar. Se levarmos em consideração a premissa que as tarefas de tempo real não suspendem durante a sua execução, quando a tarefa deixa o processador no estado R, esta está sofrendo uma interferência, quando a tarefa deixa o processador de maneira voluntária no estado S, pode-se afirmar que a tarefa terminou a sua ativação. Já quando a tarefa deixa o processador no estado D, a tarefa sofreu alguma forma de bloqueio, por exemplo em um RT Mutex. Apesar de deixar o processador de maneira voluntária, esta ainda está executando dentro da mesma ativação.

Toda troca de contexto é acionada dentro da função *schedule*, ou de uma derivação da chamada *schedule*, como as funções *__schedule* ou *schedule_preempt_disabled*. Quando a função *schedule* é executada, é dentro da sua execução que a troca de contexto é efetuada. Quando a tarefa é reescalonada, a execução da função *schedule* continua, até retornar à execução da tarefa. Uma forma de descrever este comportamento é: a função *schedule* é executada, dentro das sua execução as funções *schedule* executa vão sendo empilhadas em memória, até a troca de contexto tirar a *thread* da CPU. Após retornar a CPU, a pilha de memória continua a mesma, e as funções são desempilhadas até que a função *schedule* retorne ao fluxo de execução da tarefa.

Para exemplificar este fluxo, o *trace* a seguir mostra o comportamento da *thread* do *kernel* que executa as *softirqs*, no momento em que esta deixa o processador para uma tarefa de maior prioridade, sofrendo uma interferência, e depois retorna a sua execução.

```
 ksoftir-21  [ 98]  [2] 443775.167748229                    _raw_spin_unlock_irqrestore() {
 ksoftir-21  [ 98]  [2] 443775.167748549                      lock_release: ffff880094f94f70 &p->pi_lock
 ksoftir-21  [ 98]  [2] 443775.167749228                      local_irq_enable:  at \
                                                              _raw_spin_unlock_irqrestore
 ksoftir-21  [ 98]  [2] 443775.167749726                      sched_preempt_enable:  at \
                                                              _raw_spin_unlock_irqrestore
 ksoftir-21  [ 98]  [2] 443775.167750054                      __schedule() {
 [...]
 ksoftir-21  [ 98]  [2] 443775.167761219                        sched_switch: prev_comm=ksoftirqd/2 \
                                                                prev_pid=21 prev_prio=98 prev_state=R+ \
                                                                ==> next_comm=pi \
                                                                next_pid=1282 next_prio=29


-----------------------------------------
2)   ksoftir-21   =>    pi-1282
-----------------------------------------
 [...]
```

```
 pi-1282     [ 29]  [2] 443775.167849947                          sched_switch: prev_comm=pi prev_pid=1282 \
                                                                  prev_prio=29 prev_state=D ==> \
                                                                  next_comm=ksoftirqd/2 next_pid=21 \
                                                                  next_prio=29

-----------------------------------------
2)    pi-1282     =>   ksoftir-21
-----------------------------------------

 ksoftir-21   [ 29]  [2] 443775.167853897    ! 103.385 us           } __schedule ()
 ksoftir-21   [ 29]  [2] 443775.167854334    ! 105.826 us         } _raw_spin_unlock_irqrestore ()
```

Neste exemplo, ao reabilitar a preempção, a função *__schedule()* é chamada e decide parar a execução da tarefa *ksoftir-21*, mesmo estando apta a executar, para escalonar a *thread* de maior prioridade. Quando a tarefa *ksoftir-21* retorna à CPU, a função *__schedule* executa até retornar ao fluxo de execução da tarefa.

6.3.5.2 Tempo de resposta

O tempo de resposta de uma tarefa é a diferença entre o instante tempo que a tarefa foi ativada e o instante de tempo que a tarefa termina a execução de uma ativação. O instante de tempo que uma tarefa é acordada, no contexto das *threads*, é o instante de tempo absoluto em que o *tracepoint sched_wakeup* sinaliza a ativação da tarefa.

Para poder definir o fim de uma ativação, é necessário saber o comportamento da tarefa. A partir deste momento, restringe-se a analise apresentada neste capítulo para as tarefas que apresentam o comportamento, comum na teoria de sistemas de tempo real, de não suspender-se dentro de uma ativação. Deste modo, pode-se assumir que o fim da execução de uma ativação é dado quando a tarefa deixa o processador de maneira voluntária no estado S.

Com esta premissa, utilizando a variável sw_i utilizada na Equação 6.8, e definindo a variável s_i, com o tempo absoluto em que a tarefa suspende, isto é, que a tarefa deixa o processador no estado 'S', pode-se definir o tempo de resposta *RP* da ativação do processo conforme a Equação 6.21.

$$RP_i = s_i - sw_i \tag{6.21}$$

No método de tempo de resposta, a partir do tempo de computação, e do atraso de ativação, interferências e bloqueios que uma determinada tarefa pode sofrer, é possível determinar o tempo de resposta. Porém, aqui a operação inversa se torna mais prática: A partir da definição do tempo de resposta de uma ativação, removendo o atraso de ativação, bloqueio e interferências, é possível chegar a uma aproximação do tempo de computação da ativação de uma tarefa. As próximas seções levantam uma série de variáveis

que nos auxiliarão a percorrer o caminho inverso do modelo de tempo de resposta.

6.3.5.3 *Overhead* de escalonamento de ativação

As Figuras 26 e 27 demonstram a influência da função de escalonamento na ativação de uma tarefa. No modelo de tempo de resposta, assume-se que não há *overhead* de escalonamento, algumas vezes também é dito que este tempo está contido no tempo de execução da tarefa. Porém, na prática o *overhead* de escalonamento acontece e afeta diretamente o tempo de resposta de uma tarefa. No Linux, o *overhead* de escalonamento está diretamente relacionado com o número interferência de threads que uma determinada tarefa recebe. Como é possível observar o *overhead* de escalonamento nas tarefas, com o objetivo de melhorar a precisão da definição do tempo de execução da ativação de uma tarefa, esta seção define a influência de tempo das rotinas de escalonamento no tempo de resposta da ativação de uma tarefa com contexto de processo.

A chamada de uma das funções de escalonamento, quando causa troca de contexto, executa em duas etapas: para remover o processo e efetuar a troca de contexto, a qual aqui denominamos escalonamento de saída, e para retornar da troca de contexto e retornar a execução da tarefa, a qual denominamos escalonamento de entrada. Cada vez que um processo é escalonado, o seu tempo de execução é influenciado pelo escalonamento de entrada e de saída, nesta ordem.

A variável $ctxin_i$ denota o instante de tempo absoluto da troca de contexto que inicia a execução de uma nova ativação e, $scdr_i$ o instante de tempo absoluto no qual a função de escalonamento retorna à execução da tarefa. A janela de execução do escalonamento de entrada, $scin_i$, é determinada pela Equação 6.22.

$$scin_i = scdr_i - ctxin_i \tag{6.22}$$

Para definirmos o tempo de execução da rotina de escalonamento, é necessário diminuir o tempo de interrupções e bloqueios que ocorreram durante o escalonamento de entrada. Para auxiliar nesta operação, é necessário utilizar a função int_i e definir outra função auxiliar, que retorna o conjunto de bloqueios que ocorreram em um determinado intervalo de tempo na ativação da tarefa, esta função é denotada por $block_i(start, finish)$

Desta forma, podemos dizer que o tempo de execução do escalona-

mento de entrada é determinado pela Equação 6.23.

$$\begin{aligned} s &= ctcin_i \\ f &= scdr_i \\ siet_i &= scin_i - \sum_{\forall x \in int_i(s,f)} I_x - \sum_{\forall x \in block_i(s,f)} B_x \end{aligned} \tag{6.23}$$

Para o calcular o tempo do escalonamento de saída, define-se a variável $scdc_i$ como instante de tempo absoluto em que escalonador foi chamado, e a variável $ctxout_i$ como o instante de tempo absoluto da troca que contexto que removeu a tarefa do processador no estado 'S', que sinaliza o fim da execução. Deste modo, a janela de execução do escalonamento de saída, $scout_i$, é determinada pela Equação 6.24.

$$scout_i = ctxout_i - scdc_i \tag{6.24}$$

Da mesma forma que o escalonamento de entrada, é necessário remover os tempos de bloqueio e interferência para se obter o tempo de execução do escalonamento de saída. A definição do tempo de execução do escalonamento, $soet_i$ de saída é determinada pela Equação 6.25.

$$\begin{aligned} s &= scdc_i \\ f &= ctxout_i \\ soet_i &= scout_i - \sum_{\forall x \in int_i(s,f)} I_x - \sum_{\forall x \in block_i(s,f)} B_x \end{aligned} \tag{6.25}$$

A definição da janela de execução do *overhead* de escalonamento de entrada e saída é determinada pela Equação 6.26.

$$schw_i = scin_i + scout_i \tag{6.26}$$

E o tempo de execução de escalonamento de entrada e saída é determinada pela Equação 6.27.

$$set_i = siet_i + soet_i \tag{6.27}$$

Estas equações representam o comportamento da ativação de uma ta-

refa que não sofreu interferência de um outro processo. A próxima seção descreve o comportamento de uma tarefa quando recebe uma interferência. Isto ajudará a definir os valores da janela do tempo de execução das rotinas de escalonamento quando uma tarefa com contexto de processo sofre interferência de outra tarefa com contexto de processo.

6.3.5.4 Interferência de *threads*

Além das interferência dos tratadores de interrupção, as tarefas também sofrem interferências das *threads* de maior prioridade. No *trace* a seguir, tem-sem um exemplo de interferência de uma *thread*.

```
 ksoftir-21  [ 98]  [2] 443775.167748229                        _raw_spin_unlock_irqrestore() {
 ksoftir-21  [ 98]  [2] 443775.167748549                          lock_release: ffff880094f94f70 &p->pi_lock
 ksoftir-21  [ 98]  [2] 443775.167749228                          local_irq_enable:  at \
                                                                  _raw_spin_unlock_irqrestore
 ksoftir-21  [ 98]  [2] 443775.167749726                          sched_preempt_enable:  at \
                                                                  _raw_spin_unlock_irqrestore
 ksoftir-21  [ 98]  [2] 443775.167750054                          __schedule() {
   [...]
 ksoftir-21  [ 98]  [2] 443775.167761219                            sched_switch: prev_comm=ksoftirqd/2\
                                                                    prev_pid=21 prev_prio=98 prev_state=R+\
                                                                    ==> next_comm=pi next_pid=1282 next_prio=29
 ksoftir-21  [ 98]  [2] 443775.167761474                            lock_release: ffff88009d6244d8 &rq->lock
-----------------------------------------
2)   ksoftir-21   =>    pi-1282
-----------------------------------------

   /* tarefa de maior prioridade executa */
   [...]

  pi-1282    [ 29]  [2] 443775.167849947                          sched_switch: prev_comm=pi prev_pid=1282 \
                                                                  prev_prio=29 prev_state=D ==> \
                                                                  next_comm=ksoftirqd/2 next_pid=21 next_prio=29
  pi-1282    [ 29]  [2] 443775.167850257                          lock_release: ffff88009d6244d8 &rq->lock
-----------------------------------------
2)    pi-1282     =>   ksoftir-21
-----------------------------------------

 ksoftir-21  [ 29]  [2] 443775.167851354                          lock_acquire: ffff88009d6244d8 &rq->lock
 ksoftir-21  [ 29]  [2] 443775.167851874                            _raw_spin_unlock_irq() {
 ksoftir-21  [ 29]  [2] 443775.167852294                              lock_release: ffff88009d6244d8 &rq->lock
 ksoftir-21  [ 29]  [2] 443775.167852890                              local_irq_enable:  at _raw_spin_unlock_irq
 ksoftir-21  [ 29]  [2] 443775.167853420      1.180 us              } _raw_spin_unlock_irq ()
 ksoftir-21  [ 29]  [2] 443775.167853897    ! 103.385 us          } __schedule ()
 ksoftir-21  [ 29]  [2] 443775.167854334    ! 105.826 us        } _raw_spin_unlock_irqrestore ()
```

Neste exemplo, a tarefa *ksoftir-21* está executando na CPU 2 com a preempção desabilitada. Após reabilitar a preempção, o sistema chama a rotina de escalonamento para ceder a CPU ao processo *pi*, que possui maior prioridade. É possível identificar que a tarefa de baixa prioridade deixa o processador de maneira involuntária ao verificar o estado em que a tarefa sofreu a troca de contexto: o estado R, isto é, a tarefa estava apta a executar. Após a tarefa de alta prioridade executa até bloquear, o que pode ser verificado pelo estado que esta deixa o processador: D, e a tarefa *ksoftir-21* retorna a sua execução.

O comportamento exibido pela tarefa é semelhante ao demonstrado na Figura 27, onde a chamada do escalonador é feita dentro janela de execução

da tarefa de baixa prioridade, a troca de contexto é executada e, após a tarefa de maior prioridade executar, é feita a troca de contexto para a tarefa de baixa prioridade, que retorna a execução. Ao retornar, a função de entrada de execução é executada, até retornar para o código da aplicação.

Para melhorar a descrição, se faz necessário a definição da variável *e*, que representa as janelas de execução que ocorreram dentro da ativação de determinada tarefa. Por exemplo, em uma ativação sem interferência de processo, a ativação da tarefa teve apenas uma execução. Em uma ativação em que há uma interferência de processo, a tarefa possuirá duas execuções. O número de execuções é sempre o número de interferências de processo mais um.

O tempo de interferência da tarefa é o tempo entre a troca de contexto de saída da *e*-ésima execução e de entrada da *e*-ésima mais um execução da mesma ativação. Algebricamente, podemos dizer que a interferência de *threads* é pi_{ei} é definida como:

$$pi_{ei} = ctxout_{ei} + ctxin_{(e+1)i} \tag{6.28}$$

Como a ativação de uma tarefa com contexto de *threads* pode ter várias execuções, podemos dizer que o tempo total de interferência de *threads* é dado por:

$$PI_i = \sum_{n=1}^{e} ip_{ni} \tag{6.29}$$

Com esta definição, pode-se definir a interferência I_i, que é a soma das interferências de interrupção e de *threads* que determinada tarefa recebeu em uma ativação, com a seguinte expressão:

$$I_i = II_i + PI_i \tag{6.30}$$

6.3.5.5 *Overhead* de escalonamento de interferência

A diferença entre o *overhead* de escalonamento de ativação, descrito na seção 6.3.5.3, e de interferência, está na forma com que a tarefa deixa o processador. Quando a função de escalonamento resulta em uma troca de contexto no estado 'S', o *overhead* de entrada e saída fazem parte do fim de uma ativação e do início da próxima ativação. Já no *overhead* de escalonamento

de interferência, a chamada do escalonador resulta em uma troca de contexto no estado 'R', e o *overhead* de entrada e saída fazem parte da mesma ativação da tarefa, porém, em janelas de execução diferentes. Além deste modo, existe o modo em que a tarefa inicia a sua execução após deixar a *thread* no estado 'D'. Porém este caso acontece somente dentro de uma função de bloqueio e o seu *overhead* está inserido dentro do tempo de bloqueio.

A definição da janela de execução do escalonamento de interferência, $ischw_i$, e do tempo de execução de escalonamento de interferência, $iset_i$, são as mesmas definições da janela de execução de escalonamento e do tempo de execução de escalonamento, respectivamente. Porém, estas acontecem durante a ativação da *threads*, e não no início e no fim da *threads*.

Porém, diferente do *overhead* de escalonamento, o *overhead* de escalonamento de interferência pode ocorrer diversas vezes durante uma execução, então, considerando a variável e como a e-ésima execução dentro de uma ativação da tarefa, a definição da janela de execução de escalonamento de interferência é dada por:

$$ischw_i = \sum_{n=1}^{e} (iscin_{ni} + iscout_{ni}) \tag{6.31}$$

De forma análoga, o tempo de execução do escalonamento de interferência é dado por:

$$iset_i = \sum_{n=1}^{e} (isiet_{ni} + isoet_{ni}) \tag{6.32}$$

6.3.5.6 Overhead de escalonamento

Com a definição do *overhead* de escalonamento de ativação e de interferência, podemos definir o *overhead* total de escalonamento que a ativação sofreu como a soma destes dois *overheads* de escalonamento. Desta forma, a janela de execução do *overhead* de escalonamento, WOS_i é dada pela Equação 6.33.

$$WOS_i \quad = \quad schw_i + ischw_i \tag{6.33}$$

E o tempo de execução do *overhead* de escalonamento, EOS_i é dado pela Equação 6.34.

$$EOS_i = set_i + iset_i \tag{6.34}$$

6.3.5.7 *Threads* no espaço do *kernel*

As *threads* no espaço do *kernel* podem ser identificadas por ter o nome do processo entre colchetes '[]' na saída do comando *ps aux*. As *threads* no espaço do *kernel* são responsáveis por diversas rotinas do sistema. No PREEMPT_RT, além das *threads* já presentes no *kernel*, os tratadores de interrupção de dispositivos e de *softirq*, que no *kernel* padrão executam no contexto de interrupção, passam a executar em *threads* no contexto de memória do *kernel*, ou como são comumente chamados, de *threads* do *kernel*. Por executarem no *kernel*, todas as informações de sua execução são exibidas no *trace*, o que possibilita a análise completa de sua execução. Os tratadores de interrupção e *softirqs*, por definição, obedecem a restrição de não poderem suspender durante a sua execução, deste modo, se encaixando no modelo mais comum de tarefa de tempo real.

Apesar do código executado nas IRQs e *softirqs* serem o mesmo, com ou sem o PREEMPT_RT, existe uma diferença: os *spinlocks*. Como no PREEMPT_RT os *spinlocks* são convertidos para a versão de tempo real, que pode sofrer interrupção, os tratadores de interrupção de dispositivos e *softirqs* podem ser interrompidos por outras aplicações de maior prioridade, porém, é possível identificar estes casos. O exemplo a seguir é o da execução de uma *softirq*:

```
   pi-847    [  9]  [05]   144.406112853                        sched_switch: prev_comm=pi prev_pid=847 \
                                                                 prev_prio=9 prev_state=S ==> \
                                                                 next_comm=ksoftirqd/5 next_pid=39 \
                                                                 next_prio=98


 ------------------------------------------
 5)     pi-847     =>   ksoftir-39
 ------------------------------------------

 ksoftir-39  [ 98]  [05]   144.406116895    ! 257.828 us     } schedule_preempt_disabled ()
 ksoftir-39  [ 98]  [05]   144.406117710                     do_current_softirqs() {
 [...]
 ksoftir-39  [ 98]  [05]   144.406139080                       sched_migrate_task: comm=pi pid=847 prio=9 \
                                                               orig_cpu=5 dest_cpu=1
 [...]
 ksoftir-39  [ 98]  [05]   144.406141688                       sched_wakeup: comm=pi pid=847 prio=9 \
                                                               success=1 target_cpu=001
 [...]
 ksoftir-39  [ 98]  [05]   144.406184102    + 66.122 us      } do_current_softirqs ()
 [...]
 ksoftir-39  [ 98]  [05]   144.406189534                     schedule_preempt_disabled() {
 [...]
 ksoftir-39  [ 98]  [05]   144.406216584                           sched_switch: prev_comm=ksoftirqd/5 \
                                                                   prev_pid=39 prev_prio=98 prev_state=S \

                                                                   next_pid=0 next_prio=120


 ------------------------------------------
 5)   ksoftir-39   =>    <idle>-0
 ------------------------------------------
```

Neste exemplo, a *softirq* executou por completo sem ser interrompida ou bloqueada. Neste caso, o tempo da janela de execução da tarefa é o tempo entre as trocas de contexto de entrada e de saída, já definidas como $ctxin_i$ e $ctxout_i$ respectivamente. Desta forma, define-se a janela de execução da *thread*, pwt_i, como na Equação 6.35.

$$pwt_i = (ctxout_i - ctxin_i) - II_i \tag{6.35}$$

Nos casos onde ocorrem interferências ou bloqueios de outras *threads*, ocorrem diversas execuções da mesma ativação da tarefa. Nestes casos, existe uma janela de execução para cada execução, e a janela de execução da aplicação é dada pela soma de todas as janelas de execução das execuções, como definido na Equação 6.36.

$$pwt_i = \sum_{n=1}^{e} (ctxout_{ni} - ctxin_{ni}) - II_{ni} \tag{6.36}$$

Desta forma, podemos definir a variável W_i, que pode ter duas possíveis definições, dependendo do contexto de execução da tarefa, se for de interrupção ou de *threads*. Desta forma, utilizando as definições de pwt_i e iwt_i.

$$W_i = \begin{cases} iwt_i & \text{se contexto de interrupção} \\ pwt_i & \text{se contexto de threads} \end{cases} \tag{6.37}$$

6.3.5.8 *Threads* em espaço do usuário

A janela de execução das tarefas em espaço do usuário pode ser medida como no exemplo de *threads* executando no *kernel*, pois toda troca de contexto ocorre dentro do *kernel*, com o mesmo comportamento das tarefas que executam no *kernel*. Porém, uma tarefa em espaço do usuário pode executar de três maneiras diferentes:

- *No espaço do usuário*: executando o código da aplicação, em um contexto de memória próprio;
- *No kernel, em nome do processo*: as rotinas do kernel são chamadas por uma chamada do sistema; ou

- *No kernel, com a ativação de uma interrupção*: por exemplo, uma interrupção por uma falta de memória.

6.3.5.8.1 Tempo de execução no espaço do usuário

No contexto das *threads*, para facilitar a identificação das chamadas do sistema, uma linha com o marcador '———->' é adicionada antes da execução de uma função de chamada do sistema, e o marcador '<———-' é adicionado no retorno da função que da chamada do sistema. Desta forma é possível identificar os pontos de entrada e saída do espaço do usuário.

O tempo de execução no espaço do usuário é o tempo entre o fim e o início de uma nova rotina da *thread* no *kernel*. O exemplo mais comum é o tempo entre o fim e o início de uma chamada do sistema. Pode ocorrer também: entre o fim de uma chamada do sistema e o início de um tratador de interrupção causado pela aplicação, entre o fim de um tratador de interrupção causado pela aplicação e o inicio de um outro tratador de interrupção causado pela aplicação, e o fim do tratador de interrupção causado pela aplicação e o início de uma chamada do sistema.

Este tempo, apesar de não ser utilizado na análise, é importante para definir o tempo de execução da tarefa executando no espaço do usuário. Também serve para definir em que chamada do sistema a aplicação está sofrendo bloqueios internos no *kernel*, ou entre quais chamadas do sistema estão ocorrendo mais faltas de memória, a fim de melhorar o comportamento temporal das tarefas de tempo real.

6.3.5.9 Tempo de execução de uma *thread*

Para estimar o tempo de execução da ativação de uma tarefa no contexto das *threads*, podemos utilizar o tempo de resposta como valor base, e deste remover o atraso de ativação, o tempo de bloqueio, a interferência e o *overhead* execução da ativação do escalonamento. Desta forma, o tempo de execução de uma tarefa com contexto de *thread* é definido como a Equação 6.38.

$$CP_i = R_i - J_i - B_i - I - EOS \tag{6.38}$$

6.3.6 Tempo de execução

Com a definição das variáveis iet_i e CP_i é possível definir, de forma genérica, o tempo de computação de uma tarefa, como na Equação 6.39.

$$C_i = \begin{cases} iet_i & \text{se contexto de interrupção;} \\ CP_i & \text{se contexto de processo;} \end{cases} \tag{6.39}$$

A partir destas definições, serão conduzidas análises para demonstrar a utilização da ferramenta de trace, e a sua capacidade de obter as métricas.

6.4 ANÁLISES DE TRACE

Esta seção demonstra duas análises passo-a-passo de *trace* do ambiente experimental. A primeira utilizando os *spinlocks* como método de exclusão mútua, e a segunda utilizando os RT Mutex. Em ambos os casos, os eventos que definem o comportamento temporal da tarefa são descritos e avaliados utilizando as equações definidas na seção 6.3.1.

6.4.1 Análise com spinlock

Nesta seção são descritos alguns experimentos com *spinlock*, o primeiro utiliza o exemplo de saída de *trace* da seção 6.2.2 e o segundo um exemplo de bloqueio simples.

Para simplificar o entendimento, a saída do *trace* a seguir é a mesma saída de *trace* utilizada na seção 6.2.2.

```
 ksoftir-33  [ 98]  [04]    83.910527762                    sched_migrate_task: comm=pi pid=838 prio=9 \
                                                            orig_cpu=4 dest_cpu=0
 ksoftir-33  [ 98]  [04]    83.910529655                    sched_wakeup: comm=pi pid=838 prio=9 success=1 \
                                                            target_cpu=000
   [...]
  <idle>-0   [120]  [00]    83.910546624                  sched_preempt_enable:  at cpu_idle
  <idle>-0   [120]  [00]    83.910547023                  __schedule() {
   [...]
  <idle>-0   [120]  [00]    83.910551519                    sched_switch: prev_comm=swapper/0 prev_pid=0 \
                                                            prev_prio=120 prev_state=R+ ==> next_comm=pi \
                                                            next_pid=838 next_prio=9

------------------------------------------
 0)    <idle>-0    =>     pi-838
------------------------------------------

    pi-838    [  9]  [00]    83.910555643    + 57.470 us       } schedule ()
    pi-838    [  9]  [00]    83.910555954    + 57.987 us     } sys_pause ()
    pi-838    [  9]  [00]    83.910555954      <----------
    pi-838    [  9]  [00]    83.910557267      ---------->
    pi-838    [  9]  [00]    83.910557267                  do_notify_resume() {
    [...]
    pi-838    [  9]  [00]    83.910578502                    softirq_raise: vec=8 [action=HRTIMER]
    [...]
    pi-838    [  9]  [00]    83.910583716                    sched_wakeup: comm=ksoftirqd/0 pid=3 prio=98 \
```

```
                                                        success=1 target_cpu=000
[...]
pi-838    [ 9]  [00]    83.910605575    + 48.043 us     } do_notify_resume ()
pi-838    [ 9]  [00]    83.910605575      <----------
pi-838    [ 9]  [00]    83.910606705      ---------->
pi-838    [ 9]  [00]    83.910606705                    sys_rt_sigreturn() {
[...]
pi-838    [ 9]  [00]    83.910610888      4.021 us      } sys_rt_sigreturn ()
pi-838    [ 9]  [00]    83.910610888      <----------
pi-838    [ 9]  [00]    83.910612437      ---------->
pi-838    [ 9]  [00]    83.910612437                    sys_read() {
pi-838    [ 9]  [00]    83.910613717                      run_higher in: c = 1 inc = 11287
pi-838    [ 9]  [00]    83.910613955                      _raw_spin_lock() {
pi-838    [ 9]  [00]    83.910614175                        sched_preempt_disable: at _raw_spin_lock
pi-838    [ 9]  [00]    83.910614368                        lock_acquire: ffffffffa02d6578 &daniel_lock
pi-838    [ 9]  [00]    83.910614750                        lock_acquired: ffffffffa02d6578 &daniel_lock
pi-838    [ 9]  [00]    83.910615092      0.989 us        } _raw_spin_lock ()
pi-838    [ 9]  [00]    83.910616591                      _raw_spin_unlock() {
pi-838    [ 9]  [00]    83.910616786                        lock_release: ffffffffa02d6578 &daniel_lock
pi-838    [ 9]  [00]    83.910617153                        sched_preempt_enable:  at _raw_spin_unlock
pi-838    [ 9]  [00]    83.910617315      0.571 us        } _raw_spin_unlock ()
pi-838    [ 9]  [00]    83.910617648                      run_higher out: c = 1 inc = 11288
pi-838    [ 9]  [00]    83.910618197      5.609 us      } sys_read ()
pi-838    [ 9]  [00]    83.910618197      <----------
pi-838    [ 9]  [00]    83.910618939      ---------->
pi-838    [ 9]  [00]    83.910618939                    sys_pause() {
pi-838    [ 9]  [00]    83.910619150                      schedule() {
[...]
pi-838    [ 9]  [00]    83.910629804                          sched_switch: prev_comm=pi prev_pid=838 \
                                                            prev_prio=9 prev_state=S ==> \
                                                            next_comm=ksoftirqd/0 next_pid=3 \
                                                            next_prio=98
```

6.4.1.1 Tempo de resposta

Utilizando a Equação 6.21, obtêm-se o tempo de resposta do exemplo de *trace* da seção 6.2.2 como:

$$\begin{aligned} sw_i &= 83.910529655 \\ s_i &= 83.910629804 \\ RP_i &= 83.910629804 - 83.910529655 = 0.000100149 \\ RP_i &= 100.149us \\ R_i &= RP_i = 100.149us \end{aligned} \tag{6.40}$$

6.4.1.2 Atraso de ativação

Antes de ser reativada, a tarefa é migrada para um processador em *idle*, que devido a característica do *idle*, está com a preempção desabilitada. Deste modo, a tarefa apresenta o atraso de preempção, definido na Equação 6.3. A seguir o tempo do atraso de preempção é calculado:

$$\begin{aligned} pd &= 1 \\ ep_i &= 83.910546624 \\ sw_i &= 83.910529655 \\ pj_i &= 83.910546624 - 83.910529655 = 0.000016969 \\ pj_i &= 16.969us \end{aligned} \tag{6.41}$$

O atraso de escalonamento, definido na Equação 6.5, que depende da variável se_i da Equação 6.6, é calculado da seguinte maneira:

$$\begin{aligned} pd &= 1 \\ se_i &= ep_i = 83.910546624 \\ cw_i &= 83.910551519 \\ sj_i &= 83.910551519 - 83.910546624 = 0.000004895 \\ sj_i &= 4.895us \end{aligned} \tag{6.42}$$

A partir da definição dos valores de pj_i e sj_i, pode-se definir o valor do atraso de ativação, utilizando a Equação 6.8.

$$\begin{aligned} pj_i &= 16.969us \\ sj_i &= 4.895us \\ rpj_i &= 4.895 + 16.969 = 21.864us \\ J_i &= rpj_i = 21.864us \end{aligned} \tag{6.43}$$

6.4.1.3 *Overhead* de escalonamento de ativação

Utilizando a Equação 6.22, pode-se definir a janela de execução do escalonamento de entrada da ativação:

$$\begin{aligned} ctxin_i &= 83.910551519 \\ scdr_i &= 83.910555643 \\ scin_i &= 83.910555643 - 83.910551519 = 0.000004124 \\ scin_i &= 4.124us \end{aligned} \tag{6.44}$$

Com o valor de $scin_i$ é possível estimar o tempo de execução do escalonamento de entrada da ativação. Neste caso, é o mesmo valor da janela de execução, já que não houveram nem bloqueios nem interrupções durante esta execução.

Utilizando a Equação 6.24, é possível determinar a janela de execução do escalonamento de saída da ativação:

$$\begin{aligned} scdc_i &= 83.910619150 \\ ctxout_i &= 83.910629804 \\ scout_i &= 83.910629804 - 83.910619150 = 0.000010654 \\ scout_i &= 10.654us \end{aligned} \quad (6.45)$$

De maneira análoga ao escalonamento de entrada, durante o escalonamento de saída, não houveram nem bloqueios, nem interferências, deste modo, o tempo de execução das rotinas de escalonamento é o mesmo da janela.

Com as definições dos valores das variáveis $scin_i$ e $scout_i$, pode-se utilizar a Equação 6.26, para calcular o valor da janela de execução de escalonamento de ativação, $schw_i$.

$$\begin{aligned} scin_i &= 4.124 \\ scout_i &= 10.654 \\ schw_i &= 4.124 + 10.654 \\ schw_i &= 14.778us \end{aligned} \quad (6.46)$$

Com a definição da variável $schw_i$, sabendo que não houveram interrupções, logo o *overhead* de escalonamento de interrupção é 0. Desta forma, o valor da Equação 6.33 fica:

$$\begin{aligned} schw_i &= 14.778 \\ ischw_i &= 0 \\ WOS_i &= 14.778 + 0 \\ WOS_i &= 14.778us \end{aligned} \quad (6.47)$$

Como não houveram interferências, temos EOS_i como:

$$EOS_i = 14.778 + 0us$$

6.4.1.4 Bloqueio e Interferências

No exemplo, não houveram nem interferências nem bloqueios, durante a execução, desta forma os valores de B e I são iguais a zero.

6.4.1.5 Tempo de execução

Com as variáveis anteriores calculadas, pode-se, com a definição da Equação 6.38, obter o tempo de execução da ativação da tarefa:

$$\begin{aligned} R_i &= 100.149us \\ J_i &= 21.864us \\ I_i &= 0us \\ B_i &= 0us \\ EOS_i &= 14.778us \\ CP_i &= 100.149 - 21.864 - 0 - 0 - 14.778 \\ C_i &= CP_i = 63.507us \end{aligned} \tag{6.48}$$

6.4.1.6 Bloqueio com o spinlock

No exemplo anterior a execução não apresentou bloqueio. O próximo exemplo de *trace* demonstra o comportamento da tarefa *pi* de alta prioridade em uma ativação onde ocorreu o bloqueio na variável de exclusão mútua.

```
pi-837    [ 29]  [07]    83.929505708      ---------->
pi-837    [ 29]  [07]    83.929505708                    sys_read() {
pi-837    [ 29]  [07]    83.929507217                      run_lower in: c = 100 inc = 11627
pi-837    [ 29]  [07]    83.929507474                      _raw_spin_lock() {
pi-837    [ 29]  [07]    83.929507687                        sched_preempt_disable: at _raw_spin_lock
pi-837    [ 29]  [07]    83.929507876                        lock_acquire: ffffffffa02d6578 &daniel_lock
pi-838    [  9]  [00]    83.929507959      ---------->
pi-838    [  9]  [00]    83.929507959                    sys_read() {
pi-837    [ 29]  [07]    83.929508261                       lock_acquired: ffffffffa02d6578 &daniel_lock
pi-837    [ 29]  [07]    83.929508625     1.000 us         } _raw_spin_lock ()
pi-838    [  9]  [00]    83.929509231                      run_higher in: c = 1 inc = 11628
pi-838    [  9]  [00]    83.929509460                      _raw_spin_lock() {
pi-838    [  9]  [00]    83.929509670                        sched_preempt_disable: at _raw_spin_lock
pi-838    [  9]  [00]    83.929509858                        lock_acquire: ffffffffa02d6578 &daniel_lock
pi-838    [  9]  [00]    83.929510243                        lock_contended: ffffffffa02d6578 &daniel_lock
pi-837    [ 29]  [07]    83.929609154                        lock_release: ffffffffa02d6578 &daniel_lock
pi-837    [ 29]  [07]    83.929609649                        sched_preempt_enable:  at _raw_spin_unlock
```

```
pi-837    [ 29]  [07]    83.929609810      0.725 us        } _raw_spin_unlock ()
pi-838    [  9]  [00]    83.929609878                        lock_acquired: ffffffffa02d6578 &daniel_lock
pi-837    [ 29]  [07]    83.929610138                      run_lower out: c = 100 inc = 11629
pi-838    [  9]  [00]    83.929610188    ! 100.577 us      } _raw_spin_lock ()
pi-837    [ 29]  [07]    83.929610805    ! 104.944 us    } sys_read ()
pi-837    [ 29]  [07]    83.929610805      <----------
pi-838    [  9]  [00]    83.929611569                      _raw_spin_unlock() {
pi-838    [  9]  [00]    83.929611759                        lock_release: ffffffffa02d6578 &daniel_lock
pi-838    [  9]  [00]    83.929612115                        sched_preempt_enable:  at _raw_spin_unlock
pi-838    [  9]  [00]    83.929612274      0.558 us        } _raw_spin_unlock ()
pi-838    [  9]  [00]    83.929612619                      run_higher out: c = 1 inc = 11630
pi-838    [  9]  [00]    83.929613148    ! 105.038 us    } sys_read ()
pi-838    [  9]  [00]    83.929613148      <----------
```

Começando pela análise da tarefa *pi* de baixa prioridade, no instante de tempo absoluto 83.929507474, a tarefa inicia a obtenção de um *spinlock*, que a tarefa adquire sem sofrer contenção, retornando a execução no instante de tempo 83.929508625. Desta forma, a resolução da Equação 6.11, fica conforme a Equação 6.51.

$$\begin{aligned} bs_{baixa} &= 83.929507474 \\ bf_{baixa} &= 83.929508625 \\ ct_{baixa} &= 0 \\ \sum_{\forall x \in int_i(bs_{ji}, bf_{ji})} I_x &= 0 \\ b_{baixa} &= 0 \end{aligned} \tag{6.49}$$

Já a tarefa *pi* de alta prioridade, ao iniciar a obtenção de um *spinlock*, no instante de tempo 83.929509460, encontra a seção crítica ocupada e sofre contenção. A função que adquire o *spinlock* retorna apenas no instante de tempo 83.929610188. Como a tarefa que estava na seção crítica era de menor prioridade, esta contenção caracteriza um bloqueio. Desta forma, a resolução da Equação 6.11, fica conforme a Equação 6.51.

$$\begin{aligned} bs_{alta} &= 83.929509460 \\ bf_{alta} &= 83.929610188 \\ ct_{alta} &= 1 \\ \sum_{\forall x \in int_i(bs_{ji}, bf_{ji})} I_x &= 0 \\ b_{alta} &= 83.929610188 - 83.929509460 - 0 = 0.000100728 \\ b_{alta} &= 100.728us \end{aligned} \tag{6.50}$$

Como pode-se ver, a tarefa de alta prioridade recebeu 100.728 us de bloqueio, durante o *trace* de exemplo.

6.4.2 Exemplo com RT Mutex

Com os exemplos do *spinlock*, foi possível fazer uma análise rápida da execução sem contenção, e um exemplo a parte com contenção. O exemplo com RT Mutex a seguir é um exemplo com contenção durante a execução. Para a análise ficar mais clara, o *trace* é analisado por partes, a medida que os eventos das duas tarefas *pi*, a de alta e a de baixa prioridade, acontecem. Para ficar mais clara a computação das variáveis, a descrição e análise dos eventos são feitas após o exemplo de *trace*, e a medida que as variáveis necessárias vão sendo definidas, os valores das variáveis de tempo real serão calculados.

```
ksoftir-3   [ 98]  [00]   168.239595833                sched_migrate_task: comm=pi pid=838 prio=29 \
                                                       orig_cpu=0 dest_cpu=4
ksoftir-3   [ 98]  [00]   168.239599020                sched_wakeup: comm=pi pid=838 prio=29 success=1 \
                                                       target_cpu=004
```

As primeiras linhas do trace sinalizam o evento que acorda a tarefa de baixa prioridade, estes eventos sinalizam a migração da tarefa da CPU 0 para a CPU 4 e o momento em que a tarefa é ativada. Destas linhas pode-se definir a variável sw_{baixa} com o valor 168.239599020.

```
ksoftir-15  [ 98]  [01]   168.239611913                sched_migrate_task: comm=pi pid=839 prio=9 \
                                                       orig_cpu=1 dest_cpu=5
ksoftir-15  [ 98]  [01]   168.239614801                sched_wakeup: comm=pi pid=839 prio=9 success=1 \
                                                       target_cpu=005
```

Logo após vem as mensagens que sinalizam o evento que ativa a tarefa de alta prioridade, estes eventos sinalizam a sua migração da CPU 1 para a CPU 5 e o momento em que a tarefa é acordada. Destas linhas pode-se definir a variável sw_{alta} com o valor 168.239614801.

```
 <idle>-0   [120]  [04]   168.239655846               sched_preempt_enable:  at cpu_idle
 <idle>-0   [120]  [04]   168.239656508               __schedule() {
 <idle>-0   [120]  [04]   168.239662606                 sched_switch: prev_comm=swapper/4 prev_pid=0 \
                                                        prev_prio=120 prev_state=R+ ==> next_comm=pi \
                                                        next_pid=838 next_prio=29
```

Após habilitar a preempção, a CPU 0 sai do *idle* e chama o escalonador, que finaliza a operação de saída do *idle*, trocando o contexto para a tarefa *pi* de baixa prioridade. A partir destes eventos, é possível calcular o atraso de ativação da tarefa de baixa prioridade, utilizando as Equações 6.3, 6.8 e 6.10:

$$
\begin{aligned}
pd &= 1 \\
ep_{baixa} &= 168.239655846 \\
sw_{baixa} &= 168.239599020 \\
pj_{baixa} &= 168.239655846 - 168.239599020 = 0.000056826 \\
pj_{baixa} &= 56.826us \\
pd &= 1 \\
se_{baixa} &= ep_{baixa} = 168.239655846 \\
cw_{baixa} &= 168.239662606 \\
sj_{baixa} &= 168.239662606 - 168.239655846 = 0.000006760 \\
sj_{baixa} &= 6.760us \\
rpj_{baixa} &= 6.760 + 56.826 = 63.586us \\
J_{baixa} &= rpj_{baixa} = 63.586us
\end{aligned}
\tag{6.51}
$$

Desta forma, pode-se definir o atraso de ativação da tarefa *pi* de baixa prioridade: 63.586 us.

```
----------------------------------------
4)    <idle>-0    =>     pi-838
----------------------------------------
   pi-838    [ 29]  [04]   168.239669171    ! 117.713 us       } schedule ()
   pi-838    [ 29]  [04]   168.239669670    ! 118.522 us     } sys_pause ()
   pi-838    [ 29]  [04]   168.239669670      <----------
   pi-838    [ 29]  [04]   168.239671527      ---------->
   pi-838    [ 29]  [04]   168.239671527                   do_notify_resume() {
```

Após adquirir o processador, a tarefa *pi* de baixa prioridade termina de escalonar e inicia a sua execução, retornando ao tratamento dos sinais que ativaram a tarefa. Com estes eventos, utilizando a Equação 6.22, pode-se definir a janela de execução do escalonamento de entrada da ativação da tarefa *pi* de baixa prioridade:

$$
\begin{aligned}
ctxin_{baixa} &= 168.239662606 \\
scdr_{baixa} &= 168.239669171 \\
scin_{baixa} &= 168.239669171 - 168.239662606 = 0.000006565 \\
scin_{baixa} &= 6.565us
\end{aligned}
\tag{6.52}
$$

Como não houveram interferências ou bloqueios, o tempo de execução do escalonamento de entrada é o mesmo da janela de execução.

```
   <idle>-0   [120]  [05]   168.239672930                   sched_preempt_enable:  at cpu_idle
   <idle>-0   [120]  [05]   168.239673531                   __schedule() {
   <idle>-0   [120]  [05]   168.239679375                     sched_switch: prev_comm=swapper/5 prev_pid=0 \
```

```
prev_prio=120 prev_state=R+ ==> next_comm=pi \
next_pid=839 next_prio=9
```

Apresentando o mesmo comportamento da tarefa de baixa prioridade, a CPU 5 reabilita a preempção ao sair do *idle* e chama o escalonador, até que a troca de contexto para a tarefa *pi* de alta prioridade é efetuada. Destes eventos é possível calcular o atraso de ativação da tarefa de alta prioridade utilizando as Equações 6.3, 6.8 e 6.10:

$$\begin{aligned}
pd &= 1 \\
ep_{alta} &= 168.239672930 \\
sw_{alta} &= 168.239614801 \\
pj_{alta} &= 168.239672930 - 168.239614801 = 0.000058129 \\
pj_{alta} &= 58.129us \\
pd &= 1 \\
se_{alta} &= ep_{alta} = 168.239672930 \\
cw_{alta} &= 168.239679375 \\
sj_{alta} &= 168.239679375 - 168.239672930 = 0.000006445 \\
sj_{alta} &= 6.445us \\
rpj_{alta} &= 6.445 + 58.129 = 64.574us \\
J_{alta} &= rpj_{alta} = 64.574us
\end{aligned} \tag{6.53}$$

Desta forma, pode-se definir o atraso de ativação da tarefa *pi* de alta prioridade: 64.574 us, que por sinal, foi um valor bastante próximo dos 63.586 us do atraso de ativação da tarefa de baixa prioridade.

```
-----------------------------------------
 5)    <idle>-0     =>     pi-839
-----------------------------------------
   pi-839    [  9]  [05]   168.239685443    ! 118.150 us       } schedule ()
   pi-839    [  9]  [05]   168.239685924    ! 118.916 us     } sys_pause ()
   pi-839    [  9]  [05]   168.239685924      <----------
```

Após adquirir o processador, a tarefa de alta prioridade termina de escalonar e inicia a sua execução. Com este evento, utilizando a Equação 6.22, pode-se definir a janela de execução do escalonamento de entrada da ativação da tarefa *pi* de alta prioridade:

$$
\begin{aligned}
ctxin_{alta} &= 168.239679375 \\
scdr_{alta} &= 168.239685443 \\
scin_{alta} &= 168.239685443 - 168.239679375 = 0.000006068 \\
scin_{alta} &= 6.068us
\end{aligned}
\tag{6.54}
$$

Como não houveram interferências ou bloqueios, o tempo de execução do escalonamento de entrada é o mesmo da janela de execução.

```
pi-839    [ 9]  [05]   168.239687921      ---------->
pi-839    [ 9]  [05]   168.239687921                    do_notify_resume() {
pi-838    [ 29] [04]   168.239704433                      softirq_raise: vec=8 [action=HRTIMER]
pi-838    [ 29] [04]   168.239712485                      sched_wakeup: comm=ksoftirqd/4 pid=33 prio=98 \
                                                          success=1 target_cpu=004

pi-839    [ 9]  [05]   168.239720677                      softirq_raise: vec=8 [action=HRTIMER]
pi-839    [ 9]  [05]   168.239728697                      sched_wakeup: comm=ksoftirqd/5 pid=39 prio=98 \
                                                          success=1 target_cpu=005
pi-838    [ 29] [04]   168.239746245    + 74.272 us     } do_notify_resume ()
pi-838    [ 29] [04]   168.239746245      <----------
pi-838    [ 29] [04]   168.239747962      ---------->
pi-838    [ 29] [04]   168.239747962                    sys_rt_sigreturn() {
pi-838    [ 29] [04]   168.239759482    + 11.080 us     } sys_rt_sigreturn ()
pi-838    [ 29] [04]   168.239759482      <----------
pi-839    [ 9]  [05]   168.239761861    + 73.498 us     } do_notify_resume ()
pi-839    [ 9]  [05]   168.239761861      <----------
```

Neste ponto, a tarefa de baixa prioridade tratou os sinais e esta pronta para entrar na seção de leitura do módulo.

```
pi-838    [ 29] [04]   168.239763293      ---------->
pi-838    [ 29] [04]   168.239763293                    sys_read() {
pi-839    [ 9]  [05]   168.239763570      ---------->
pi-839    [ 9]  [05]   168.239763570                    sys_rt_sigreturn() {
pi-838    [ 29] [04]   168.239766266                      run_lower in: c = 100 inc = 12925
pi-838    [ 29] [04]   168.239766795                      _mutex_lock() {
pi-838    [ 29] [04]   168.239767367                        lock_acquire: ffffffffa02e05c8 &daniel_lock
pi-838    [ 29] [04]   168.239768241      0.212 us          rt_mutex_lock();
pi-838    [ 29] [04]   168.239769324      2.104 us        } _mutex_lock ()
```

Ao entrar na leitura, a tarefa de baixa prioridade adquire o mutex compartilhado e entra na seção crítica. Enquanto a tarefa de alta prioridade ainda está tratando o sinal. Como não houve contenção na aquisição do mutex compartilhado, o tempo de bloqueio para a tarefa de baixa prioridade continua em 0.

```
pi-839    [ 9]  [05]   168.239774923    + 10.910 us     } sys_rt_sigreturn ()
pi-839    [ 9]  [05]   168.239774923      <----------
pi-839    [ 9]  [05]   168.239778639      ---------->
pi-839    [ 9]  [05]   168.239778639                    sys_read() {
pi-839    [ 9]  [05]   168.239781557                      run_higher in: c = 1 inc = 12926
pi-839    [ 9]  [05]   168.239782119                      _mutex_lock() {
pi-839    [ 9]  [05]   168.239782660                        lock_acquire: ffffffffa02e05c8 &daniel_lock
```

A tarefa de alta prioridade entra na leitura do módulo. Ao tentar adquirir a variável de exclusão mútua a aquisição é negada. Então a tarefa de alta prioridade entra nos mecanismos de mutex de tempo real. Uma das ações é executar o controle de inversão de prioridade, atribuindo a prioridade da

tarefa de maior prioridade para a tarefa de menor prioridade. Neste ponto, pode-se definir o valor de bs_{1-alta} com o instante do início da execução da aquisição do mutex compartilhado, que é 168.239782119.

```
pi-839    [  9]  [05]   168.239783459                    rt_mutex_lock() {
pi-839    [  9]  [05]   168.239784051                      rt_mutex_slowlock() {
pi-839    [  9]  [05]   168.239800358                        sched_pi_setprio: comm=pi pid=838 oldprio=29 \
                                                             newprio=9
pi-838    [ 29]  [04]   168.239820262                  sched_preempt_enable:  at delay_tsc
pi-838    [  9]  [04]   168.239820748                  sched_preempt_disable: at delay_tsc
```

É possível ver nas mensagens de *trace* da tarefa de baixa prioridade o exato momento em que a tarefa de baixa prioridade herda a prioridade da tarefa de alta prioridade.

```
pi-839    [  9]  [05]   168.239827151                      __rt_mutex_slowlock() {
pi-839    [  9]  [05]   168.239830573                        schedule() {
pi-839    [  9]  [05]   168.239857194                            sched_switch: prev_comm=pi prev_pid=839 \
                                                                 prev_prio=9 prev_state=D ==> \
                                                                 next_comm=ksoftirqd/5 next_pid=39 \
                                                                 next_prio=98
```

Após finalizar o tratamento de controle de inversão de prioridade, a tarefa de alta prioridade muda de estado para suspensa de modo ininterruptível, e chama o escalonador, que a remove do processador. Como esta chamada ao escalonador fez a tarefa deixar o processador no estado 'D', este tempo não é computado como janela de escalonamento, e estará incluso no tempo de bloqueio no mutex.

```
pi-838    [  9]  [04]   168.239871594                  _mutex_unlock() {
pi-838    [  9]  [04]   168.239873955                    rt_mutex_unlock() {
pi-838    [  9]  [04]   168.239874567                      rt_mutex_slowunlock() {
pi-838    [  9]  [04]   168.239896697                        sched_migrate_task: comm=pi pid=839 prio=9 \
                                                             orig_cpu=5 dest_cpu=1
pi-838    [  9]  [04]   168.239902059                        sched_wakeup: comm=pi pid=839 prio=9 \
                                                             success=1 target_cpu=001
pi-838    [  9]  [04]   168.239922015                        sched_pi_setprio: comm=pi pid=838 oldprio=9 \
                                                             newprio=29
```

Após terminar a execução dentro da seção critica, a tarefa de baixa prioridade, ainda executando com a prioridade da tarefa de alta prioridade, chama a função que libera o mutex compartilhado. Ao tentar liberar o mutex compartilhado, a função executa o processo que acorda a tarefa de alta prioridade que está esperando pelo mutex. Este processo migrou a tarefa para um processador que estava em *idle*, acordou a tarefa e, por fim, desfez a mudança de prioridade feita pelo protocolo de herança de prioridade.

```
  <idle>-0   [120]  [01]   168.239934383              sched_preempt_enable:  at cpu_idle
  <idle>-0   [120]  [01]   168.239935022              __schedule() {
  <idle>-0   [120]  [01]   168.239941145                sched_switch: prev_comm=swapper/1 prev_pid=0 \
                                                        prev_prio=120 prev_state=R+ ==> next_comm=pi \
                                                        next_pid=839 next_prio=9


-----------------------------------------
 1)    <idle>-0    =>     pi-839
-----------------------------------------
```

Após sair do *idle*, a CPU 1 executa o código que escalona a tarefa de maior prioridade. Algo a chamar atenção, é que a preempção está desabilitada, que sempre ocorre quando o processador está em *idle*, e é interpretada

como atraso de ativação quando a tarefa é acordada pela primeira vez. Acontece aqui também, porém, neste caso, este valor será adicionado ao tempo de bloqueio da tarefa.

```
pi-838    [ 29]  [04]   168.239944742    + 69.838 us           } rt_mutex_slowunlock ()
pi-838    [ 29]  [04]   168.239945001    + 70.702 us         } rt_mutex_unlock ()
pi-838    [ 29]  [04]   168.239945273    + 73.306 us       } _mutex_unlock ()
pi-838    [ 29]  [04]   168.239946022                      run_lower out: c = 100 inc = 12927
```

Após terminar o processo de liberação do mutex, a tarefa de baixa prioridade retorna a sua execução. Deste processo, é possível ver uma diferença entre a teoria e a pratica. Nos tempos de bloqueio de uma tarefa, sempre é considerado um tempo de espera pelo *lock*, mas não um tempo de liberação do *lock*. Isto pode ser simplificado adicionando este tempo ao tempo de execução da tarefa. Neste exemplo, o tempo de liberação do *lock* adicionou 73.306*us* de *overhead* a tarefa de baixa prioridade, o que é um tempo relativamente grande, pois sem bloquear, esta tarefa deveria executar em pouco mais de 100*us*.

```
pi-838    [ 29]  [04]   168.239947245    ! 183.593 us     } sys_read ()
pi-838    [ 29]  [04]   168.239947245      <----------
pi-838    [ 29]  [04]   168.239948905      ---------->
pi-838    [ 29]  [04]   168.239948905                     sys_pause() {
pi-838    [ 29]  [04]   168.239949247                       schedule() {
```

Após liberar o *lock*, a tarefa de baixa prioridade retorna ao espaço do usuário. Ao todo, o tempo no *read* levou 183.593*us*. Boa parte deste tempo veio da liberação do mutex. Neste ponto, a tarefa de baixa prioridade chama o escalonador, no instante de tempo 168.239949247.

```
pi-839    [  9]  [01]   168.239960447      1.872 us             } _raw_spin_unlock ()
pi-839    [  9]  [01]   168.239960919    ! 176.489 us         } rt_mutex_slowlock ()
pi-839    [  9]  [01]   168.239961366    ! 177.530 us       } rt_mutex_lock ()
pi-839    [  9]  [01]   168.239961820    ! 179.275 us     } _mutex_lock ()
```

A tarefa de alta prioridade, após ser escalonada, retorna ao processo de aquisição da variável de exclusão mútua. Após adquirir o mutex, o fluxo de execução retorna ao fluxo da aplicação. Do retorno da função mutex_lock é possível determinar o valor de bf_{1-alta} com o instante de tempo do retorno da função, que é 168.239961820. Com isto, pode-se calcular o tempo do primeiro bloqueio desta ativação da tarefa *pi* de alta prioridade:

$$\begin{aligned}
bs_{1-alta} &= 168.239782119 \\
bf_{1-alta} &= 168.239961820 \\
ct_{1-alta} &= 1 \\
\sum_{\forall x \in int_i(bs_{ji}, bf_{ji})} I_x &= 0 \qquad (6.55) \\
b_{1-alta} &= 168.239961820 - 168.239782119 - 0 = 0.000179701 \\
b_{1-alta} &= 179.701us
\end{aligned}$$

Com a Equação 6.56, temos que o tempo que a tarefa de alta prioridade ficou bloqueada foi de 179.701 us. Em uma breve comparação deste tempo de bloqueio com o calculado na Equação 6.51, que foi de 100.728 us, em uma situação semelhante de disputa da variável de exclusão mútua, pode-se ver que a aquisição do mutex foi superior em 78 us a aquisição do spinlock. Este tempo superior é resultado do controle de inversão de prioridade e do fato da tarefa bloqueada suspender a execução.

```
pi-839    [  9]  [01]   168.239964621                    _mutex_unlock() {
```

Após executar a seção critica, a tarefa de alta prioridade inicia a liberação do *lock*.

```
pi-838    [ 29]  [04]   168.239964646                    sched_switch: prev_comm=pi \
                                                         prev_pid=838 prev_prio=29 \
                                                         prev_state=S  ==> \
                                                         next_comm=ksoftirqd/4 next_pid=33 \
                                                         next_prio=98

-----------------------------------------
 4)     pi-838     =>   ksoftir-33
-----------------------------------------
```

Neste momento, a tarefa de baixa prioridade termina a execução da ativação. A partir destes momento, pode-se computar as demais variáveis que definem o comportamento temporal da tarefa.

Utilizando a Equação 6.21, pode-se definir o tempo de resposta como:

$$\begin{aligned} sw_{baixa} &= 168.239599020 \\ s_{baixa} &= 168.239964646 \\ RP_{baixa} &= 168.239964646 - 168.239599020 = 0.000365626 \\ RP_{baixa} &= 365.626us \\ R_{baixa} &= RP_i = 365.626us \end{aligned} \tag{6.56}$$

O próximo passo é definição da janela de execução de escalonamento de saída da ativação, utilizando a Equação 6.24.

$$\begin{aligned} scdc_{baixa} &= 168.239949247 \\ ctxout_{baixa} &= 168.239964646 \\ scout_{baixa} &= 168.239964646 - 168.239949247 = 0.000015399 \\ scout_{baixa} &= 15.399us \end{aligned} \tag{6.57}$$

Com a definição do $scout_{baixa}$, é possível definir a janela de execução do escalonamento de ativação, utilizando a Equação 6.26.

$$\begin{aligned} scin_{baixa} &= 6.565 \\ scout_{baixa} &= 15.399 \\ schw_{baixa} &= 6.565 + 15.399 \\ schw_{baixa} &= 21.964us \end{aligned} \tag{6.58}$$

Com a definição da janela de escalonamento de ativação, e como a tarefa não sofreu interferências de processos, pode-se definir a janela e o tempo de execução de escalonamento de ativação utilizando a Equação 6.33:

$$\begin{aligned} schw_{baixa} &= 21.964 \\ ischw_{baixa} &= 0 \\ WOS_{baixa} &= 21.964 + 0 \\ WOS_{baixa} &= 21.964 \\ EOS_{baixa} &= 21.964 + 0us \end{aligned} \tag{6.59}$$

Como o sistema não teve bloqueios ou interferências, os valores de B_baixa e I_baixa é zero. Desta forma, pode-se definir o tempo de execução desta ativação da tarefa *pi* de baixa prioridade utilizando a Equação 6.38:

$$\begin{aligned} R_i &= 365.626us \\ J_i &= 63.586us \\ I_i &= 0us \\ B_i &= 0us \\ EOS_i &= 21.964us \\ CP_i &= 365.626 - 63.586 - 0 - 0 - 21.964 \\ C_i &= CP_i = 280.076us \end{aligned} \tag{6.60}$$

Desta forma, tem-se computado os valores das variáveis que definem o comportamento temporal da tarefa de baixa prioridade.

Um valor interessante é o do tempo de execução. Inicialmente, considerando o tempo de espera ocupada na seção critica, podia-se supor que o tempo de execução seria um valor próximo dos 100 us, mas a computação do valor revelou um número quase três vezes maior. Analisando a saída do trace, é possível perceber que os 180us a mais da execução vem da execução de rotinas de *timer*, na execução da chamada do sistema *do_notify_resume*

que somou 74.272 us, na execução da chamada do sistema *sys_rt_sigreturn* que somou 11.080 us, além do tempo de execução da função *_mutex_unlock*, que para liberar a variável de exclusão mútua somou 73.306 us. Desta forma, somam-se 158.658 us de tempo de execução que é inicialmente ignorado.

```
pi-839    [  9]  [01]   168.239965165                        lock_release: ffffffffa02e05c8 &daniel_lock
pi-839    [  9]  [01]   168.239966832      0.127 us          rt_mutex_unlock();
pi-839    [  9]  [01]   168.239967848      2.813 us        } _mutex_unlock ()
pi-839    [  9]  [01]   168.239968944                      run_higher out: c = 1 inc = 12928
pi-839    [  9]  [01]   168.239970422    ! 191.341 us     } sys_read ()
pi-839    [  9]  [01]   168.239970422      <----------
```

Continuando a execução da tarefa de alta prioridade, ao retornar para o espaço do usuário no fim da execução da chamada *read*, tem-se a estimativa de que a tarefa levou 191.341*us* executando a leitura no módulo, tempo muito superior aos 5.609*us*, que a chamada do sistema levou para executar quando não bloqueou, usando *spinlock*.

```
pi-839    [  9]  [01]   168.239972371      ---------->
pi-839    [  9]  [01]   168.239972371                      do_notify_resume() {
pi-839    [  9]  [01]   168.240005839                        softirq_raise: vec=8 [action=HRTIMER]
pi-839    [  9]  [01]   168.240013714                        sched_wakeup: comm=ksoftirqd/1 pid=15 prio=98 \
                                                           success=1 target_cpu=001
pi-839    [  9]  [01]   168.240037487    + 64.696 us     } do_notify_resume ()
pi-839    [  9]  [01]   168.240037487      <----------
pi-839    [  9]  [01]   168.240038795      ---------->
pi-839    [  9]  [01]   168.240038795                      sys_rt_sigreturn() {
pi-839    [  9]  [01]   168.240045205      6.183 us        } sys_rt_sigreturn ()
pi-839    [  9]  [01]   168.240045205      <----------
pi-839    [  9]  [01]   168.240047616      ---------->
pi-839    [  9]  [01]   168.240047616                      sys_pause() {
pi-839    [  9]  [01]   168.240047925                        schedule() {
pi-839    [  9]  [01]   168.240063374                            sched_switch: prev_comm=pi prev_pid=839 \
                                                               prev_prio=9 prev_state=S ==> \
                                                               next_comm=ksoftirqd/1 next_pid=15 \
                                                               next_prio=98
```

Ao retornar para o espaço do usuário, as funções que tratam os sinais são executadas, isto porque a tarefa executou por mais tempo que seu período, e durante o a sua execução, a tarefa recebeu um sinal para acordar e iniciar a execução de uma nova ativação.

Após tratar o sinal a tarefa suspende a execução, isto porque a tarefa não foi concebida para tratar uma ativação sem antes ter suspendido. Neste caso, o uso da ferramenta de *trace* auxilia na descoberta do problema e, principalmente, no que causa o problema. Neste exemplo, o alto tempo para a obtenção do *lock*, em partes pelo *overhead* de obtenção do *lock*, que gerou a perda do *deadline*.

Utilizando a Equação 6.21, pode-se definir o tempo de resposta como:

$$\begin{aligned} sw_{alta} &= 168.239614801 \\ s_{alta} &= 168.240063374 \\ RP_{alta} &= 168.240063374 - 168.239614801 = 0.000448573 \\ RP_{alta} &= 448.573us \\ R_{alta} &= RP_i = 448.573us \end{aligned} \tag{6.61}$$

Com o tempo de resposta superior ao período, é possível observar que a tarefa perdeu o *deadline*, que foi a causa da execução dos sinais após o retorno da função da leitura do módulo. O próximo passo é definição da janela de execução de escalonamento de saída da ativação, utilizando a Equação 6.24.

$$\begin{aligned} scdc_{alta} &= 168.240047925 \\ ctxout_{alta} &= 168.240063374 \\ scout_{alta} &= 168.240063374 - 168.240047925 = 0.000015449 \\ scout_{alta} &= 15.449us \end{aligned} \tag{6.62}$$

Com a definição do $scout_{alta}$, é possível definir a janela de execução do escalonamento de ativação, utilizando a Equação 6.26.

$$\begin{aligned} scin_{alta} &= 6.068 \\ scout_{alta} &= 15.449 \\ schw_{alta} &= 6.068 + 15.449 = 21.517us \\ schw_{alta} &= 21.517us \end{aligned} \tag{6.63}$$

Com a definição da janela de escalonamento de ativação, e como a tarefa não sofreu interferências de processos, pode-se definir a janela de escalonamento utilizando a Equação 6.33:

$$\begin{aligned} schw_{alta} &= 21.517 \\ ischw_{alta} &= 0 \\ WOS_{alta} &= 21.517 + 0 \\ WOS_{alta} &= 21.517us \end{aligned} \tag{6.64}$$

Como não houveram interferências, pode-se definir o EOS_i como:

$$EOS_{alta} = 21.517 + 0us \tag{6.65}$$

Diferente da ativação da tarefa de baixa prioridade, a tarefa de alta sofreu um bloqueio. Utilizando a Equação 6.12, pode-se definir o valor da variável de bloqueio B_{alta} :

$$B_{alta} = 179.701us \tag{6.66}$$

Como o sistema não sofreu interferências, o valor de I_alta é zero. Desta forma, pode-se definir o tempo de execução desta ativação da tarefa *pi* de alta prioridade utilizando a Equação 6.38:

$$\begin{aligned} R_{alta} &= 448.573us \\ J_{alta} &= 64.574us \\ I_{alta} &= 0us \\ B_{alta} &= 179.701us \\ EOS_{alta} &= 21.517us \\ CP_{alta} &= 448.573 - 64.574 - 0 - 179.701 - 21.517 \\ C_{alta} &= CP_{alta} = 182.781us \end{aligned} \tag{6.67}$$

Considerando que o tempo de execução na seção crítica da execução de alta prioridade é de 1 us, e 100 us para a de baixa prioridade, o comportamento da tarefa de alta prioridade foi semelhante ao da tarefa de baixa prioridade, pois como esperado foi cerca de 100us maior. Porém, diferente da tarefa de baixa prioridade que teve se tempo de computação maior que o esperado por ter que liberar o mutex, a tarefa de alta prioridade teve seu tempo de computação afetado por ter de tratar duas vezes os sinais, um que gerou a ativação e outro do deadline perdido.

6.5 INTERPRETAÇÃO DOS DADOS

A partir das definições de como interpretar o trace e como determinar os valores da variáveis, foi executada uma análise completa de uma seção de *trace*, da qual foram extraídos os valores as seguintes variáveis de cada ativação da tarefa:

- Atraso de ativação;
- Bloqueio;
- Interferência;
- Escalonamento de entrada e saída;
- Tempo de execução;

Para facilitar a análise, foi escolhido o *trace* que utiliza os *raw_spinlocks* com método de exclusão mútua e, a análise foi feita apenas para tarefa de maior prioridade, a qual não sofre interferências de outros processos, apenas das interrupções. A coleta destes dados foi feita através de um *script* em *shell* que filtrava e calculava os tempos. Para automatizar a extração dos dados de modelos mais complexos, se faz necessário o desenvolvimento de uma aplicação mais completa, mas não houve tempo suficiente para desenvolver neste trabalho. Um exemplo da saída dos dados do *script* é exibido a seguir:

```
NEW_JOB: 1849
JITTER  = 0.000032249
SCHED_IN  = 0.000005667
SCHED_OUT  = 0.000015681
RESPONSE  = 0.000250563
BLOCK  = 0.000102173
INTERFERENCE  = 0
EXECUTION  = 0.000094793
```

Ao total foram avaliadas 5259 ativações da tarefa, as quais foram removidas de um trace de 2 segundos. Do trace de 2 segundos, foram removidos os primeiros e últimos 300 ms, isto para remover a interferência dos *scripts* que controlavam a execução do experimento. Foram avaliados 1.4 segundos de execução do experimento. A parir da saída deste *script* foram gerados os gráficos das próximas seções.

6.5.1 Tempo de resposta

A Figura 28 exibe estes tempos de resposta das ativações da tarefa.

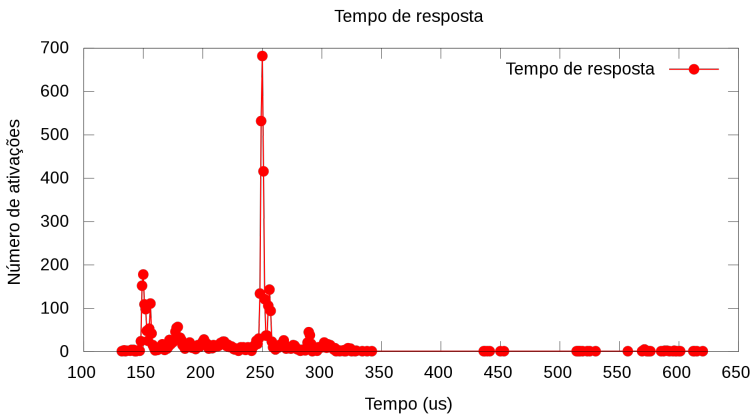

Figura 28 – Tempo de resposta

Dos tempos de resposta, no menor tempo de resposta a tarefa não sofreu nem bloqueio nem interferência, sendo seu tempo de execução a soma dos tempos de atraso de ativação, escalonamento de entrada, execução e escalonamento de saída. Já o maior tempo de resposta foi de 620.147 us, e possuía as seguintes características:

```
NEW_JOB: 1346
JITTER          = 0.000036727
SCHED_IN        = 0.000006006
SCHED_OUT       = 0.000026064
RESPONSE        = 0.000620147
BLOCK       = 0.000384957
INTERFERENCE    = 0
EXECUTION       = 0.000166393
```

O causador do alto tempo de resposta foi o bloqueio. Analisando o *trace* completo, a tarefa ficou bloqueada quatro vezes, em três variáveis de exclusão mútua durante a execução. Duas destas variáveis eram RT *spinlocks*: No RT *spinlock sighand->siglock* a tarefa bloqueou duas vezes, por 94.421 us e 94.243 *us*, no RT *spinlock new_timer->it_lock* por 94.385 *us*; e uma era um *spinlock*, justamente o da função *read* do módulo, que bloqueou a tarefa por 101.908 us. Como a tarefa ficou muito tempo bloqueada, ocorreram duas ativações do *timer* que acorda a tarefa, por isto o tempo de computação da tarefa também foi penalizado.

6.5.2 Atraso de ativação

A Figura 29 exibe a característica dos tempos do atraso de ativação das tarefas.

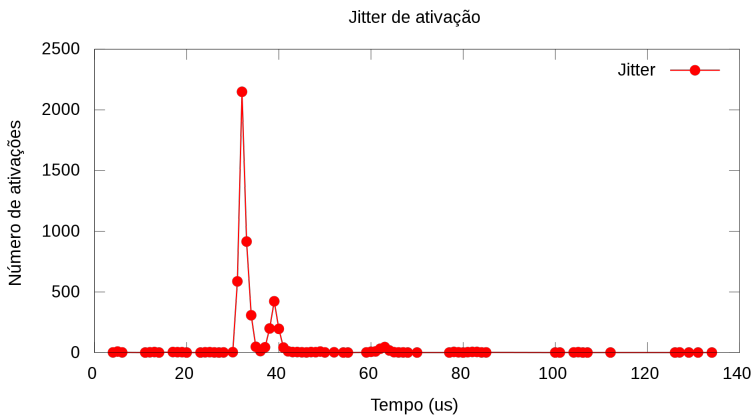

Figura 29 – Atraso de ativação

A maioria dos tempos ficam abaixo dos 50 *us*, com picos em 31, 32 e 33 *us*. Este padrão é causado pelo fato de que, como temos apenas duas tarefas - mais as de manutenção do sistema, em oito processadores, a maioria das ativações são direcionadas aos processadores em *idle*. Apenas em três casos, a tarefa é acordada em um processador que está executando alguma tarefa. Ao contrário do intuitivo, os maiores tempos acontecem justamente quando o processador está saindo do estado de *idle* e executa alguma tarefa de manutenção do sistema. Um dos motivos é exemplificado pelo próprio *trace*, descrito a seguir.

O processador entra em *idle*, e é acordado pelo processador que acordou a tarefa.

```
<idle>-0   [120]  [04]    96.962902431      1.006 us       } rcu_read_unlock ()
<idle>-0   [120]  [04]    97.058018108                     local_irq_enable:  at intel_idle
```

Pelo delta dos dois tempos, o processador ficou em *idle* por cerca de 95 ms: 0.095115677. Ao sair do *idle*, o processador executa algumas funções de manutenção do sistema.

```
<idle>-0   [120]  [04]    97.058020124                     local_irq_disable:  at rcu_idle_exit
<idle>-0   [120]  [04]    97.058020943                     local_irq_enable:  at rcu_idle_exit
<idle>-0   [120]  [04]    97.058021832                     rcu_read_lock() {
<idle>-0   [120]  [04]    97.058022558                       lock_acquire: ffffffff81c36300 read rcu_read_lock
<idle>-0   [120]  [04]    97.058023614      1.253 us       } rcu_read_lock ()
<idle>-0   [120]  [04]    97.058024320                     rcu_read_unlock() {
<idle>-0   [120]  [04]    97.058024929                       lock_release: ffffffff81c36300 rcu_read_lock
<idle>-0   [120]  [04]    97.058026425      1.608 us       } rcu_read_unlock ()
```

Neste ponto, o processador desabilita as interrupções na função *tick_nohz_idle_exit*

```
<idle>-0   [120]  [04]    97.058027054                     local_irq_disable:  at tick_nohz_idle_exit
<idle>-0   [120]  [04]    97.058028973                     _raw_spin_lock() {
<idle>-0   [120]  [04]    97.058029832                       lock_acquire: ffff880138fd4258 &rq->lock
<idle>-0   [120]  [04]    97.058030903                       lock_acquired: ffff880138fd4258 &rq->lock
<idle>-0   [120]  [04]    97.058031951      2.461 us       } _raw_spin_lock ()
<idle>-0   [120]  [04]    97.058032785                     _raw_spin_unlock() {
<idle>-0   [120]  [04]    97.058033366                       lock_release: ffff880138fd4258 &rq->lock
<idle>-0   [120]  [04]    97.058034667      1.370 us       } _raw_spin_unlock ()
<idle>-0   [120]  [04]    97.058035987                     _raw_spin_lock_irqsave() {
<idle>-0   [120]  [04]    97.058036816                       lock_acquire: ffff880138e0d7b8 &cpu_base->lock
<idle>-0   [120]  [04]    97.058037792                       lock_acquired: ffff880138e0d7b8 &cpu_base->lock
<idle>-0   [120]  [04]    97.058038683      2.189 us       } _raw_spin_lock_irqsave ()
<idle>-0   [120]  [04]    97.058039504                     _raw_spin_unlock_irqrestore() {
<idle>-0   [120]  [04]    97.058040141                       lock_release: ffff880138e0d7b8 &cpu_base->lock
<idle>-0   [120]  [04]    97.058041426      1.400 us       } _raw_spin_unlock_irqrestore ()
<idle>-0   [120]  [04]    97.058042584                     _raw_spin_lock_irqsave() {
<idle>-0   [120]  [04]    97.058043331                       lock_acquire: ffff880138e0d7b8 &cpu_base->lock
<idle>-0   [120]  [04]    97.058044232                       lock_acquired: ffff880138e0d7b8 &cpu_base->lock
<idle>-0   [120]  [04]    97.058045093      2.007 us       } _raw_spin_lock_irqsave ()
<idle>-0   [120]  [04]    97.058048056                     _raw_spin_unlock_irqrestore() {
<idle>-0   [120]  [04]    97.058048685                       lock_release: ffff880138e0d7b8 &cpu_base->lock
<idle>-0   [120]  [04]    97.058049858      1.278 us       } _raw_spin_unlock_irqrestore ()
```

E reabilita as interrupções, mais tarde, na mesma função:

```
<idle>-0   [120]  [04]    97.058140963                     local_irq_enable:  at tick_nohz_idle_exit
```

O delta de tempo, entre desabilitar e reabilitar as IRQs, foi de 0.000113909 *s*, ou 113.909 *us*, que é a maior parte do tempo, dos 134.445 *us*, que a tarefa

sofreu de atraso de ativação. Isto ocorreu por causa das rotinas que fazem o controle de tempo no Linux.

Outro motivo comum que afeta o atraso ocorre quando, ao sair do *idle*, o processador recebe uma interrupção, antes de chamar o escalonador.

6.5.3 Escalonamento

Os tempos de execução de escalonamento de entrada e saída são exibidos no gráfico da Figura 30.

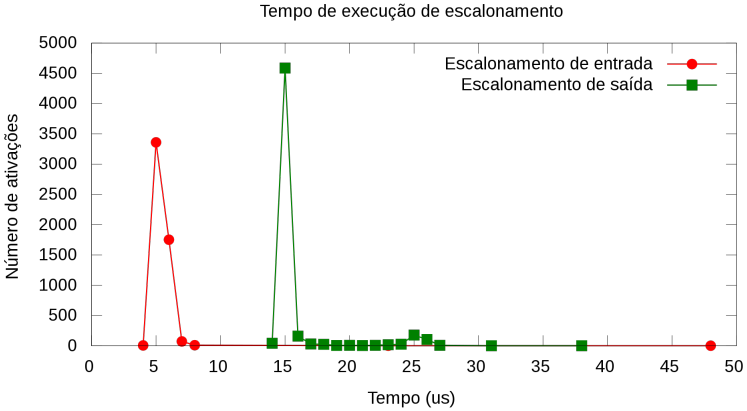

Figura 30 – Escalonamento

Do escalonamento de entrada, apenas dois aconteceram com mais de 10 us, sendo todos os outros abaixo dos 8 us. Apesar de parecer uma saída falsa, observando os traces completos, a explicação para estas duas saídas foram os fluxos de execução, que foram diferentes nestas duas execuções. Analisando o código da função *schedule*, por exemplo, após a troca de contexto, a função *post_scheduler* é chamada, e esta foi uma das possíveis causadoras destes tempos.

No escalonamento de saída, temos um bom exemplo da diferença entre tempo real e desempenho. Apesar do tempo de execução do escalonamento de saída, na maioria dos casos, ser maior que o tempo de execução do escalonamento de entrada, o pior tempo de execução do escalonamento de saída é menor que o pior tempo de execução do escalonamento de entrada.

6.5.4 Tempo de Bloqueio

O gráfico da Figura 31, exibe os tempos de bloqueio que as ativações da tarefa *pi* sofreram em suas ativações.

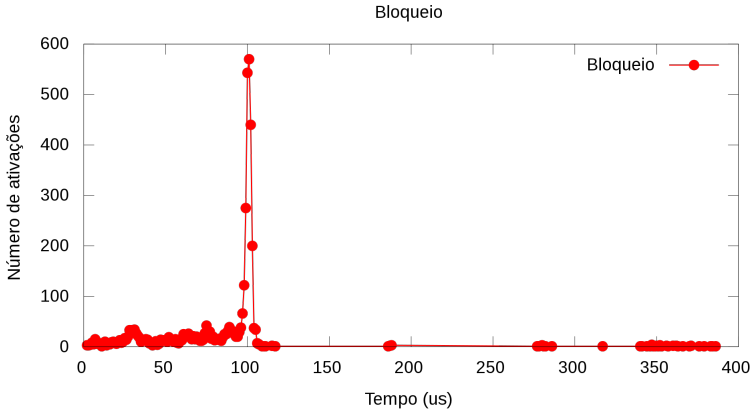

Figura 31 – Bloqueios

A maioria dos bloqueios duraram cerca de 100 us. Muitos destes tempos são os tempos dos bloqueios que acontecem no *lock* compartilhado do módulo, que acontece na leitura do *driver*. Porém, como as duas *threads pi* também acessam as mesmas funções de tempo do sistema, estes acabam compartilhando outros *locks*, como no exemplo de bloqueio demonstrado na seção 6.5.1.

Curiosamente, os *locks sighand->siglock* e *new_timer->it_lock*, que são RT *spinlocks*, causam contenção na casa dos 90 us, o que dá a característica do gráfico, de mostrar diversos eventos entre 0 e pouco mais de 100 us, onde ficam os bloqueios do lock compartilhado e uma contenção de um destes *locks*. Quando há duas contenções os pontos ficam próximos dos 180 us, quando há três contenções ficam próximo aos 260 us e quatro contenções próximo aos 340us. Esta característica acontece, principalmente, porque as duas tarefas acabam por sincronizar por causa das disputas por variáveis de exclusão mútua, assim, fazendo o mesmo fluxo de execução e sofrendo mais bloqueio.

6.5.5 Interferência

A Figura 32 exibe as janelas de execução das interrupções, que causaram as interferências que a tarefa de maior prioridade recebeu.

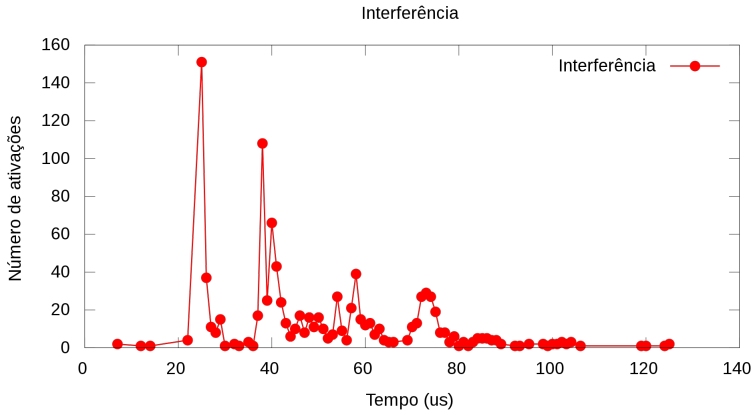

Figura 32 – Interferência

Ao total foram 1080 interrupções, sendo que apenas uma não foi do timer, foi uma nmi, que levou 8 us. Apesar dos picos de execução nos 38 e 25 us, os tempos da janela execução das interrupções do *timer* tendem a ficar maiores por um motivo: ocorrem ao mesmo tempo em diversos processadores, o que causa bloqueio. Por exemplo, o *trace* a seguir é o momento em que as interrupção que causou maior interferência foi chamada:

```
   pi-852    [  9]  [02]    96.881932484      ==========>
   pi-852    [  9]  [02]    96.881932484                    smp_apic_timer_interrupt() {
   pi-851    [ 29]  [01]    96.881932587      ==========>
   pi-851    [ 29]  [01]    96.881932587                    smp_apic_timer_interrupt() {
ksoftir-45   [ 98]  [06]    96.881932689      ==========>
ksoftir-45   [ 98]  [06]    96.881932689                  smp_apic_timer_interrupt() {
ksoftir-39   [ 98]  [05]    96.881932849      ==========>
ksoftir-39   [ 98]  [05]    96.881932849                    smp_apic_timer_interrupt() {
```

Neste exemplo ocorreram quatro interrupções, ao mesmo tempo, em quatro processadores diferentes, quais disputaram o acesso às mesmas seções críticas. Neste caso, o tratador que interrompeu a tarefa de maior prioridade esperou por 44.253 us pelo *raw_spinlock xtime_lock*.

6.5.6 Tempo de execução

O gráfico da figura 33 exibe os tempos de execução das tarefas. O menor tempo de execução foi de 64 us. A maior parte dos tempos de execução,

79.72 % (4193 de 5259), ficou no intervalo entre [93,98] *us*. A diferença entre estes tempos e o menor ocorre porque, na maioria das vezes, a função *do_notify_resume* executa funções de manutenção do tempo no sistema, as quais algumas vezes não acontecem, como no menor tempo de execução.

Já os maiores tempos de execução, que ficam acima dos 110 *us*, são causados quando, por causa dos bloqueios, interferências ou atraso, o sistema perde o *deadline* e tem que tratar os sinais que acordariam a próxima ativação antes de dormir. Isto não aconteceria se o sistema não perdesse o *deadline*.

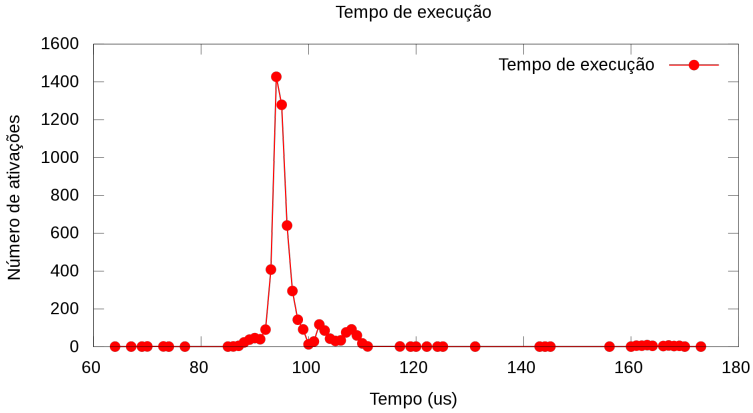

Figura 33 – Tempo de execução

6.5.7 Maiores tempos para as variáveis

A partir da saída da análise do *trace* pode-se identificar, das ativações analisadas, o maior tempo de cada variável.

Variável	Tempo máximo (us)
Atraso de ativação	134
Bloqueio	386
Interferência	125
Execução	173

Tabela 3 – Tempos máximos observados para as variáveis.

6.6 CONCLUSÃO

Este capítulo avaliou a capacidade da ferramenta de fazer o *trace* das funções e mecanismos do *kernel* do Linux que afetam o comportamento temporal das tarefas. Com a nova forma de visualizar a execução do Linux possível mostrar de maneira mais clara o comportamento das tarefas de tempo real executando dentro do *kernel* do Linux.

Partindo das abstrações utilizadas no método de análise de tempo de resposta, foi possível caracterizar a execução das tarefas no Linux com o PREEMPT-RT, mostrando como o tempo de resposta de uma tarefa pode ser afetado pelos atrasos de ativação e bloqueios dentro do *kernel*. A partir da caracterização das tarefas, foi possível definir as equações a serem utilizadas nas medições de *trace* a fim de obter os valores das métricas que compõem o tempo de resposta da ativação de uma tarefa.

Vale destacar que o objetivo destas equações não é adaptar o Linux para o modelo de tempo de resposta, nem definir um novo método de análise de tempo de resposta para o Linux. O objetivo da especificação destas equações foi formalizar a medição dos tempos que influenciam nos tempos de execução das tarefas. Acredita-se que esta formalização das tarefas no Linux possa servir de base para trabalhos futuros que desenvolvam métodos de análise de tempo de resposta.

7 CONSIDERAÇÕES FINAIS

Os sistemas de tempo real são sistemas em que a corretude não depende somente da corretude lógica, mas também da corretude temporal. Os sistemas de tempo real geralmente são caracterizados como sistemas de tempo real rígido (*hard real-time*), nos quais uma falha temporal pode causar uma catástrofe como a perda de vidas humanas, e sistemas de tempo real brando (*soft real-time*), nos quais algumas falhas temporais são toleráveis, porém não desejáveis.

Apesar da teoria de sistemas de tempo real para sistemas com um único processador estar bastante consolidada e já aplicada na indústria, a teoria de sistemas com mais de um processador é ainda hoje um campo aberto e fonte de diversas pesquisas. Aliado ao crescimento das plataformas multiprocessadas, nasceu também a necessidade dos sistemas operacionais se adequarem a esta tecnologia. O sistema operacional Linux foi um dos primeiros sistemas a se adequar aos sistemas multi-processados. Aliado a disponibilidade de seu código fonte, além de ser tornar uma plataforma de mercado para os sistemas multi-processados, o Linux também se tornou uma plataforma de pesquisa, incluindo as pesquisas em sistemas de tempo real.

Dentre as implementações de tempo real do Linux, o PREEMPT-RT se tornou o padrão de fato. O objetivo do PREEMPT-RT é proporcionar maior determinismo à execução do Linux. Para isto, várias modificações foram feitas no *kernel* do Linux a fim de diminuir as latências do sistema, diminuindo assim o tempo de resposta das tarefas de maior prioridade. Apesar desta métrica ter se mostrado eficiente, ela se difere da teoria de sistemas de tempo real. Na teoria de sistemas de tempo real busca-se analiticamente comprovar que um conjunto de tarefas irá executar antes dos seus *deadlines*, apesar das interferências e bloqueios que estas podem sofrer durante a sua execução. Como a forma de análise é a base da teoria de sistemas de tempo real, esta diferença na forma de avaliar o sistema acaba distanciando a teoria e a prática em sistemas de tempo real, principalmente do Linux de tempo real.

Buscando aproximar a teoria de sistemas de tempo real ao Linux, este trabalho buscou relacionar as abstrações utilizadas no método de análise de tempo de resposta com as funções do *kernel* do Linux, no que diz respeito às funções que afetam temporalmente a execução das tarefas de tempo real no Linux. Ao converter diversas rotinas do *kernel* em *threads* que executam no *kernel*, como os tratadores de IRQ de dispositivos, o PREEMPT-RT facilitou a relação entre a abstração de tarefas da teoria com as tarefas no Linux. Outro simplificador é o modo FULL PREEMPTBLE do *kernel*. Neste modo, ao invés do kernel possuir pontos de preempção, o sistema passa a ter pontos de

não preempção. Desta forma, o controle de atraso de ativação para threads é simplificado. O controle de preempção acaba por ser mais fácil de determinar do que determinar os pontos em que o *kernel* permite a preempção. Isto tanto pelo tamanho do código do *kernel* do Linux, quanto pelas suas frequentes alterações, as quais tornam difíceis a especificação de um diagrama que mostre os pontos de preempção do *kernel*, ou a determinação de em que ponto uma tarefa de maior prioridade pode adquirir o processador. Outro simplificador foi a forma com que os bloqueios são gerados. As APIs são bastante simples, facilitando o processo de identificação das funções que podem gerar bloqueios.

Com esta relação entre as variáveis do modelo análitico e as funções do *kernel* do Linux, as ferramentas de *trace* atualmente disponíveis foram avaliadas, a fim de verificar a capacidade destas ferramentas mostrar a execução dos pontos de interesse. Nesta etapa, foram avaliadas três ferramentas: FeatherTrace, Ftrace e o LTTng. Para cada uma destas ferramentas foram apresentados os seus componentes, como utilizar a ferramenta, quais as formas de *trace*, os pontos de trace já existentes, como a ferramenta foi desenvolvida e como estender a ferramenta como novos pontos de *trace*. Do estudo dos pontos de *trace*, viu-se que nenhuma das ferramentas era capaz de fazer o *trace* de todos os pontos que afetam a execução temporal das tarefas de tempo real levantados no capítulo 3. Isto porque a métrica de análise do sistema utilizada por estas ferrametas é justamente a latência de eventos no *kernel*, como de troca de contexto. A falta do pontos de interesse nas ferramentas existentes gerou a necessidade de estender uma das ferramentas estudadas. Da necessidade de decidir qual ferramenta estender, foram elencados um conjunto de métricas para avaliar qualitativamente cada uma destas ferramentas. As três ferramentas foram avaliadas e comparadas utilizando estas métricas. Esta análise comparativa facilitou a decisão de qual ferramenta utilizar: o Ftrace.

Foram testadas duas possibilidades de adaptação do Ftrace, a primeira utilizando apenas tracepoints e a segunda criando um novo *plugin* de *trace*. Optou-se por utilizar uma solução mista com a criação de um novo plugin de *trace*, denominado timeflow, e tracepoints em pontos de controle do sistema. A ativação da ferramenta é feita com a ativação do *plugin* de trace, a qual ativa também os *tracepoints*. Para filtrar as funções que desejava-se fazer o *trace*, optou-se por utilizar uma técnica já utilizada no Ftrace, de agrupar as funções em uma seção do binário do *kernel* e comparar se o endereço da função está nesta seção. Este técnica auxilia tanto em diminuir o *overhead* do *trace*, quanto em facilitar a extensão do *trace*. Também foram feitas alterações na forma com que os dados são exibidos pelo Ftrace, dando maiores informações ao *trace*. Devido a forma com que o Ftrace foi desenvolvido, esta etapa do trabalho acabou por ser mais simples do que o esperado. Na verdade,

a maior parte do desenvolvimento da ferramenta foi utilizado para entender a dinâmica do *kernel*, a fim de entender o seu comportamento, garantindo que a ferramenta estava adquirindo os dados de maneira correta. Durante esta etapa já foi possível perceber que a ferramenta facilitou muito o entendimento da dinâmica do *kernel*. O que antes exigia um entendimento profundo do *kernel*, como explicar o que causou um grande atraso no tempo de resposta de uma tarefa, se tornou uma tarefa simples e intuitiva, bastando conhecer apenas os conceitos básicos de interferências e bloqueios.

Para validar a capacidade da nova ferramenta de melhorar o entendimento do *kernel* do Linux, com foco nos parâmentros que afetam temporalmente as tarefas de tempo real, foi montado um ambiente onde foram conduzidos experimentos de trace com a execução de duas tarefas de tempo real com o comportamento conhecido. A primeira análise feita foi a do próprio *trace*, na qual foi possível determinar quais eventos afetaram temporalmente as tarefas de tempo real no Linux. Com a análise do trace de diversas execuções das tarefas de tempo real no Linux, foi possível caracterizar a execução das *threads* e dos tratadores de IRQ no Linux, desde que respeitadas algumas restrições, com a de não suspender durante a execução. Após esta caracterização, foi possível desenvolver equações com as quais é possível efetuar medições nos tempos de execução de uma tarefa, a fim de obter-se os tempos que compoem o tempo de resposta das tarefas analisadas.

Dos resutlados obtidos nas pesquisas desta dissertação foram publicados dois artigos, sendo um *short paper* publicado no SBESC e um poster aceito no ACM SAC, ambos sobre a relação entre as abstrações utilizadas no modelo de tempo de resposta e as funções do *kernel* do Linux, descritos no Capítulo 3 desta dissertação. Um artigo com a análise comparativa entre as ferramentas de trace disponíveis para o Linux, descritas no Capítulo 4, foi submetido para a Revista IEEE Latinoamericana, este artigo ainda está em revisão. Ainda serão desenvolvidos dois artigos, o primeiro apresentando a ferramenta e a caracterização das tarefas de tempo real no Linux, a ser enviado para uma revista científica. O segundo artigo avaliará a ferramenta utilizada para medir a latência do PREEMPT-RT, o Cyclictest, com a ferramenta de trace desenvolvida neste trabalho, a fim de demosntrar qual a origem da latência medida com o Cyclictest.

Apesar da caracterização e das equações objetivarem apenas a medição das ativações das tarefas, a fim de determinar como é composto o tempo de resposta de uma tarefa após ela ter executado no Linux, acredita-se que o relacionamento das abstrações do método de análise de tempo de resposta com as funções do Linux, a ferramenta de *trace*, a caracterização e as equações de medição apresentadas neste trabalho, possam servir de base para uma futura formalização da execução das tarefas no Linux. Formalização qual, conjunta

com um ambiente controlado, possa servir de base para o surgimento de um novo modelo analítico de tempo de resposta.

Vale ressaltar que um modelo de tempo de resposta para o Linux, invariavelmente, deve levar em consideração a relação de precedência das tarefas do sistema. As tarefas no Linux apresentam uma forte ligação de precedência. Por exemplo, as rotinas que dependem de um *timer* exigem a precedência de uma rotina que ative o *timer*. Estas rotinas podem ser uma interrupção periódica do *timer*, que ativa uma *softirq* que acorda a *thread* que efetivamente depende do *timer*. Desta forma, o atraso de ativação da interrupção e da *softirq*, juntamente com os bloqueios e interferências que estas podem receber, acabam influenciando a ativação da tarefa que efetivamente espera o *timer*. Outro complicador é que, para dar uma maior vazão no sistema, algumas rotinas internas no *kernel* não apresentam um comportamento muito simples de se formalizar ou predizer. Uma destas rotinas é justamente as rotinas do *timer*. Durante a execução de uma tarefa que usa as rotinas do *timer*, o *kernel* pode decidir executar as rotinas que controlam a execução dos relógios de alta precisão, os quais verificam a passagem do tempo. Estas rotinas podem acabar por acionar rotinas que acordem outra tarefa que estava esperando a passagem do tempo. Mesmo que as tarefas não apresentem relação de precedência. Felizmente, estas execuções e peculiaridades do *kernel* que influenciam na execução temporal das tarefas estão mais claras e de fácil entendimento, graças a ferramenta de *trace* apresentada neste trabalho.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDERSON, J.; MILLS, A. A stochastic framework for multiprocessor soft real-time scheduling. In: ALBERS, S. et al. (Ed.). *Scheduling*. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. (Dagstuhl Seminar Proceedings, 10071). ISSN 1862-4405. <http://drops.dagstuhl.de/opus/volltexte/2010/2537>.

BASTONI, A.; BRANDENBUG, B.; ANDERSON, J. Is semi-partitioned scheduling practical? *23rd Euromicro Conference on Real-Time Systems*, 2011.

BRANDENBUG, B. B.; ANDERSON, J. H. Joint opportinities for real-time linux and real-time system research. *Real Time Linux Workshops*, 2009.

BRANDENBUG, J. H. A. B. B. Feather-trace: A light-weight event tracing toolkit. *Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2007.

BRANDENBURG, B. *Feather-Trace: A Light-Weight Event Tracing Toolkit*. Julho 2012. <http://www.cs.unc.edu/ bbb/feathertrace/>. Acessado em 19/07/12.

BRANDENBURG, B. B. *Scheduling And Locking In Multiprocessor Real-Time Operating Systems*. Tese (Doutorado) — University of North Carolina at Chapel Hill, 2011.

BUTTAZZO, G. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 2. ed. [S.l.]: Springer, 2005.

CORBET, J. *Dynamic probes with ftrace*. July 2009. <http://lwn.net/Articles/343766/>. Acessado em 01/05/13.

CORBET, J. *Jump Label*. Outubro 2010. <http://lwn.net/Articles/412072/>. Acessado em 21/07/12.

CORBET, J. *Linux at NASDAQ OMX*. Outubro 2010. <http://lwn.net/Articles/411064/>. Acessado em 22/02/2012.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. *Systems Research*, 2009.

DESNOYERS JULIEN DESFOSSEZ, D. G. M. *LTTng 2.0: Tracing for power users and developers*. Abril 2012. <http://lwn.net/Articles/491510/>. Acessado em 20/08/12.

DESNOYERS, M. *LTTng Project*. Agosto 2012. <http://lttng.org/>. Acessado em 20/08/12.

EFFICIOS. *Common Trace Format - CTF*. Agosto 2012. <http://www.efficios.com/ctf>. Acessado em 20/08/12.

FAGGIOLI, D.; TRIMARCHI, M.; CHECCONI, F. An implementation of the earliest deadline first algorithm in linux. In: SHIN, S. Y.; OSSOWSKI, S. (Ed.). *SAC*. [S.l.]: ACM, 2009. p. 1984–1989. ISBN 978-1-60558-166-8.

GLEIXNER, T. *Realtime Linux: academia v. reality*. July 2010. <http://lwn.net/Articles/471973/>. Acessado em 22/08/12.

LITMUSRT. *LITMUS-RT: Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*. July 2010. <http://www.litmus-rt.org/>. Acessado em 22/08/12.

LITMUSRT. *LITMUS-RT: Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*. 2012. <http://www.litmus-rt.org/>. Acessado em 22/02/2012.

LITMUSRT. *Tracing - litmus*. 2012. <https://wiki.litmus-rt.org/litmus/Tracing>. Acessado em 19/07/12.

LIU, J. W. S. *Real-Time Systems*. [S.l.]: Prentice Hall, 2000.

LOVE, R. *Linux kernel development*. 3. ed. [S.l.]: Addison-Wesley, 2010.

MCKENNEY, P. *What is RCU?* September 2005. <https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/RCU/whatisRCU.txt?id=refs/tags/v3.6.7>. Acessado em 01/05/13.

MCKENNEY, P. *What is RCU? Part 2: Usage*. December 2007. <http://lwn.net/Articles/263130/>. Acessado em 28/04/13.

MCKENNEY, P. *The RCU API table*. December 2010. <http://lwn.net/Articles/419086/>. Acessado em 28/04/13.

MOLLISON, M. *Unit-Trace*. 2011. <http://cs.unc.edu/ mollison/unit-trace/>. Acessado em 19/07/12.

PREEMPT_RT. *Real-Time Linux Wiki*. 2013. <https://rt.wiki.kernel.org/>. Acessado em 22/02/2013.

ROSTEDT, S. *RT-mutex subsystem with PI support*. June 2006. <https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/Documentation/rt-mutex.txt?id=refs/tags/v3.6.7>. Acessado em 01/05/13.

ROSTEDT, S. *Debugging the kernel using Ftrace - part 1*. Dezembro 2009. <http://lwn.net/Articles/365835/>. Acessado em 19/07/12.

ROSTEDT, S. *Secrets of the Ftrace function tracer*. Janeiro 2010. <http://lwn.net/Articles/370423/>. Acessado em 21/07/12.

ROSTEDT, S. *trace-cmd: A front-end for Ftrace*. October 2010. <http://lwn.net/Articles/410200/>. Acessado em 01/05/13.

ROSTEDT, S. *Using the TRACE_EVENT() macro (Part 1)*. March 2010. <http://lwn.net/Articles/379903/>. Acessado em 21/07/12.

ROSTEDT, S. *Using KernelShark to analyze the real-time scheduler*. February 2011. <http://lwn.net/Articles/425583/>. Acessado em 01/05/13.

RTAI. *RTAI - the RealTime Application Interface for Linux from DIAPM*. 2013. <https://www.rtai.org/>. Acessado em 22/02/2013.

XENOMAI. *Xenomai Real-Time Framework for Linux*. 2013. <http://www.xenomai.org/>. Acessado em 22/02/2013.