Sergio Antonio Rodríguez

UMA PROPOSTA DE MODIFICAÇÕES NO SISTEMA OPERACIONAL LINUX PARA PROCESSAMENTO DIGITAL DE SINAIS EM TEMPO REAL

São Paulo
2011

Sergio Antonio Rodríguez

UMA PROPOSTA DE MODIFICAÇÕES NO SISTEMA OPERACIONAL LINUX PARA PROCESSAMENTO DIGITAL DE SINAIS EM TEMPO REAL

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do Título de Doutor em Engenharia.

São Paulo
2011

Sergio Antonio Rodríguez

UMA PROPOSTA DE MODIFICAÇÕES NO SISTEMA OPERACIONAL LINUX PARA PROCESSAMENTO DIGITAL DE SINAIS EM TEMPO REAL

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do Título de Doutor em Engenharia.

Área de Concentração:
Sistemas Eletrônicos

Orientador: Prof. Dr. Phillip M.S. Burt

São Paulo
2011

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, de junho de 2011.

Assinatura do autor _____

Assinatura do orientador _____

FICHA CATALOGRÁFICA

Rodriguez, Sergio Antonio
Uma proposta de modificações no sistema operacional Linux para processamento digital de sinais em tempo real / S. A. Rodriguez. -- ed.rev. -- São Paulo, 2011.
298 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Telecomunicações e Controle.

1.Linux 2.Processamento digital de sinais 3.Tempo real I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Tele-comunicações e Controle II.t.

RESUMO

Esta tese estuda modificações no sistema operacional Linux para a arquitetura x86 da Intel, com a finalidade de aumentar o desempenho, nessa plataforma, das aplicações de processamento digital de sinais em tempo real. Inicialmente são estabelecidos requisitos para um sistema operacional voltado para o processamento digital de sinais. Os requisitos são estabelecidos com base na estrutura dos programas de processamento digital de sinais em tempo real e nas situações mais comuns nesse tipo de processamento. Um fator chave quando se trata de processamento em tempo real é a latência para colocar o aplicativo em execução. Nesse contexto, o trabalho desenvolve um modelo para a latência no tratamento das interrupções externas no Linux. Usando esse modelo é desenvolvido um método para medir as várias componentes dessa latência, método este baseado na colocação de marcadores de tempo no núcleo do Linux. O método de medida proposto é usado para medir a latência do Linux no tratamento de uma interrupção externa em três condições diferentes. O estudo finaliza propondo, implementando e testando alterações no Linux que visam melhorar o desempenho, desse sistema, em aplicações de processamento digital de sinais em tempo real.

Palavras-chaves: Linux. Processamento digital de sinais. Tempo-real.

ABSTRACT

This work studies modifications in the Linux operating system for the Intel x86 architecture, with the purpose of increasing the performance in applications of digital signal processing in real-time. Initially the requirements for a digital signal processing operating systems are established. These requirements are based on the structure of the programs of digital signal processing in real-time and the most common situations in this type of processing. An important factor in real-time processing is the latency to put the process in execution. In this context, this work develops a latency model for the I/O interrupts handling in Linux. By using this model, a method is developed to measure components of this latency. This measurement method is based on time markers collocated in the Linux kernel. The proposed method is used to measure the Linux latency for an I/O interrupt handling in three different conditions. Finally this research proposes, implements and tests alterations in Linux with the purpose of increasing the performance of this operating system in applications of digital signal processing in real-time.

Key-words: Linux. Digital signal processing. Real-time.

LISTA DE ILUSTRAÇÕES

Figura 1.1 - Modelo de um sistema PDS em tempo real com base em um microcomputador do tipo IBM-PC com hardware PDS dedicado 28
Figura 1.2 - Modelo de um sistema PDS em tempo real com base em um microcomputador do tipo IBM-PC com Linux e placa adaptadora.... 28
Figura 2.1 - Diagrama de blocos de um programa típico de PDS 37
Figura 2.2 - Diagrama de blocos de um programa típico de tempo real 38
Figura 2.3 - Operação em modo *Push-In* 39
Figura 2.4 - Operação em modo *Pull-In* 40
Figura 2.5 - Diagrama de blocos de um sistema para reconhecimento de comandos de voz em aviões de combate 41
Figura 2.6 - Diagrama de blocos de uma estação rádio base desenvolvida com microcomputador do tipo IBM-PC 42
Figura 2.7 - Diagrama de blocos de um rádio receptor desenvolvido com microcomputador do tipo IBM-PC 43
Figura 2.8 - Modelo de um sistema para processamento digital de sinais baseado em um microcomputador do tipo IBM-PC com hardware PDS dedicado 44
Figura 2.9 - Modelo de um sistema para processamento digital de sinais baseado em um microcomputador do tipo IBM-PC com Linux e placa adaptadora 45
Figura 2.10 - Modelo de camadas para o software de processamento digital de sinal 47
Figura 2.11 - Diagrama de blocos de uma placa adaptadora 49
Figura 2.12 - Modelo de operação do sistema operacional PDS 50
Figura 3.1 - Modelo de operação de um SO-PDS 57
Figura 4.1 - Modelo para a latência de um processo PDS 74
Figura 4.2 – Componentes da latência de tratamento de interrupção 75
Figura 4.3 - Bloco de instruções correspondente ao endereço constante nas entradas da tabela IDT para interrupções E/S externas 81

Figura 4.4 - Bloco de instruções que inicia o tratamento de uma interrupção E/S externa 81
Figura 4.5 - Parte principal do código da função `handle_IRQ_event` 84
Figura 4.6 - Componentes da latência de despacho 86
Figura 4.7 - Trecho final do código da rotina `do_irq` 87
Figura 4.8 - Código parcial da rotina `irq_exit` 88
Figura 6.1 - Método para medição da latência no tratamento das interrupções E/S externas no Linux 111
Figura 6.2 - Esquema para medição da latência 113
Figura 6.3 - Diagrama de blocos Aplicativo Gerenciamento 120
Figura 6.4 - Código parcial da Rotina PDS 121
Figura 6.5 - Diagrama de blocos da função `le_dados` 123
Figura 6.6 - Diagrama de blocos da rotina `driver_medida` 125
Figura 6.7 - Código fonte do Marcador TSC 126
Figura 6.8 - CDF da latência de hardware 131
Figura 6.9 - Gráfico (1-CDF) da latência de hardware 132
Figura 6.10 - CDF da latência de entrada no tratador de interrupções 133
Figura 6.11 - Gráfico (1-CDF) da latência de entrada no tratador de interrupções 134
Figura 6.12 - CDF da latência de compartilhamento 135
Figura 6.13 - CDF da latência de saída do tratador de interrupções 136
Figura 6.14 - CDF da latência de pilha de IRQs 137
Figura 6.15 - Gráfico (1-CDF) da latência de pilha de IRQs 138
Figura 6.16 - CDF da latência de d*eferrable functions* 139
Figura 6.17 - Gráfico (1-CDF) da latência de d*eferrable functions* 140
Figura 6.18 - CDF da latência de preempção do núcleo desabilitada 141
Figura 6.19 - Gráfico (1-CDF) da latência de preempção do núcleo desabilitada 142
Figura 6.20 - CDF da latência de escalonamento 143
Figura 6.21 - Gráfico (1-CDF) da latência de escalonamento 144
Figura 6.22 - CDF da latência de troca de contexto 145
Figura 6.23 - Gráfico (1-CDF) da latência de troca de contexto 146
Figura 6.24 - CDF da latência total 147
Figura 6.25 - Gráfico (1-CDF) da latência total 148

Figura 6.26 - Histograma períodos de recuperação para o teste com carga 149
Figura 7.1 - Diagrama de estados do SO-PDS 164
Figura 7.2 - Diagrama de blocos de um programa PDS..... 167
Figura 7.3 - Diagrama de estados para o processo PDS 168
Figura 7.4 - Diagrama de blocos de parte do tratador de interrupções do Linux . 171
Figura 7.5 - Código fonte parcial da rotina `do_irq` 171
Figura 7.6 - Diagrama de blocos da função `do_IRQ_PDS_N`..... 173
Figura 7.7 - Diagrama de blocos da função `do_IRQ_PDS_S` 175
Figura 7.8 - Diagrama de blocos da função `driver_le_amostra` 178
Figura 7.9 - CDF da latência total para o Teste I 180
Figura 7.10 - CDF da latência total para o Teste II 182
Figura A1 - Algumas chamadas ao sistema do padrão POSIX 194
Figura A2 - Estados de um processo em um sistema operacional 196
Figura A3 - Estados de um processo no Linux 197
Figura B.1 - Um exemplo da execução de fluxos de controle do núcleo aninhados 206
Figura B.2 - Diagrama de bloco de um sistema multi-APIC 210
Figura C.1 - Vetores de interrupção no Linux para a arquitetura x86 217
Figura C.2 - Um exemplo de configuração do sinal IRQ e seus dispositivos em um microcomputador compatível com o IBM_PC 218
Figura C.3 - Principais campos do descritor `irq_desc_t` 219
Figura C.4 - Alguns bits do campo `status` e seus respectivos significados 220
Figura C.5 - Lista de manipuladores de interrupções no Linux 221
Figura C.6 - Exemplo da estrutura `irq_chip` para o controlador 8259A 222
Figura C.7 - Principais campos do descritor `irq_action` 223
Figura C.8 - Bits do campo `flags` e seus significados 223
Figura C.9 - Estruturas de dados para o tratamento das interrupções 224
Figura C.10 - Bloco de instruções endereçado pela IDT para interrupções E/S externas 225
Figura C.11 - Bloco de instruções que inicia o tratamento de uma interrupção E/S externa 225
Figura C.12 - Macro `SAVE_ALL` que salva os registradores 226

Figura C.13 - Trecho da rotina `do_IRQ` (1ª Parte) 227
Figura C.14 - Trecho da rotina `do_IRQ` (2ª Parte) 228
Figura C.15 - Definição da rotina `irq_enter` 228
Figura C.16 - Trecho da rotina `do_IRQ` (3ª Parte) 229
Figura C.17 - Trecho da rotina `do_IRQ` (4ª Parte) 230
Figura C.18 - Trecho da rotina `do_IRQ` (5ª Parte) 231
Figura C.19 - Código da rotina `irq_exit` 231
Figura C.20 - Trecho da rotina `__do_IRQ` (1ª Parte) 232
Figura C.21 - Trecho da rotina `__do_IRQ` (2ª Parte) 233
Figura C.22 - Trecho da rotina `__do_IRQ` (3ª Parte) 234
Figura C.23 - Trecho da rotina `__do_IRQ` (4ª Parte) 236
Figura C.24 - Parte principal do código da função `handle_IRQ_event` 237
Figura C.25 - Fluxograma do retorno da interrupção em núcleo sem preempção 238
Figura C.26 - Fluxograma do retorno da interrupção em núcleo com preempção 239
Figura C.27 - Instruções do tratador da interrupção do temporizador da APIC Local 241
Figura D.1 - Exemplo de sistema de multiplexação por divisão de tempo com base em microcomputador compatível com o IBM-PC 242
Figura D.2 - Influência do tempo de processamento na subamostragem 244
Figura F.1 - Instruções para tratar a interrupção do temporizador da APIC Local 248
Figura F.2 - Código da rotina `smp_apic_timer_interrupt` 249
Figura I.1 - CDF aproximada da latência de pilha de IRQs 263
Figura I.2 - CDF da latência de entrada no tratador de interrupções 264
Figura I.3 - CDF da latência de saída do tratador de interrupções 265
Figura I.4 - CDF da latência de d*eferrable functions* 266
Figura I.5 - CDF da latência de preempção do núcleo desabilitada 267
Figura I.6 - CDF da latência de escalonamento 268
Figura I.7 - CDF da latência de troca de contexto 269
Figura K.1 – CDF da latência de hardware 281

Figura K.2 – Gráfico (1-CDF) da latência de hardware 282
Figura K.3 – CDF da latência de entrada no tratador de interrupções 283
Figura K.4 – Gráfico (1-CDF) da latência de entrada no tratador de interrupções 284
Figura K.5 – CDF da latência de compartilhamento 285
Figura K.6 – CDF da latência de saída do tratador de interrupções 286
Figura K.7 – CDF da latência de pilha de IRQs 287
Figura K.8 – Gráfico (1-CDF) da latência de pilha de IRQs 288
Figura K.9 – CDF da latência de d*eferrable functions* 289
Figura K.10 – Gráfico (1-CDF) da latência de d*eferrable functions* 290
Figura K.11 – CDF da latência de preempção do núcleo desabilitada 291
Figura K.12 – Gráfico (1-CDF) da latência de preempção do núcleo desabilitada 292
Figura K.13 – CDF da latência de escalonamento 293
Figura K.14 – Gráfico (1-CDF) da latência de escalonamento 294
Figura K.15 – CDF da latência de troca de contexto 295
Figura K.16 – Gráfico (1-CDF) da latência de troca de contexto 296
Figura K.17 – CDF da latência total 297
Figura K.18 – Gráfico (1-CDF) da latência total 298

LISTA DE TABELAS

Tabela 6.1 - Descrição dos campos da estrutura `real_time_struct` 127
Tabela 6.2 - Estatísticas da latência de hardware 131
Tabela 6.3 - Estatísticas da latência de entrada no tratador de interrupções 133
Tabela 6.4 - Estatísticas da latência de compartilhamento 135
Tabela 6.5 - Estatísticas da latência de saída do tratador de interrupções 136
Tabela 6.6 - Estatísticas da latência de pilha de IRQs 137
Tabela 6.7 - Estatísticas da latência de *deferrable functions* 139
Tabela 6.8 - Estatísticas da latência de preempção do núcleo desabilitada 141
Tabela 6.9 - Estatísticas da latência de escalonamento 143
Tabela 6.10 - Estatísticas da latência de troca de contexto 145
Tabela 6.11 - Estatísticas latência total 147
Tabela 6.12 – Número de Falhas e Períodos de Recuperação em 20.000 medições 148
Tabela 7.1 - Campos da estrutura de dados `pds_rt_st` 163
Tabela 7.2 - Teste I Estatísticas da latência total 180
Tabela 7.3 - Teste II Estatísticas da latência total 182
Tabela I.1 - Estatísticas da latência de entrada no tratador de interrupções 264
Tabela I.2 - Estatísticas da latência de saída do tratador de interrupções 265
Tabela I.3 - Estatísticas da latência de *deferrable functions* 266
Tabela I.4 - Estatísticas da latência de preempção do núcleo desabilitada 267
Tabela I.5 - Estatísticas da latência de escalonamento 268
Tabela I.6 - Estatísticas da latência de troca de contexto 269
Tabela K.1 – Estatísticas da latência de hardware 281
Tabela K.2 – Estatísticas da latência de entrada no tratador de interrupções ... 283
Tabela K.3 – Estatísticas da latência de compartilhamento 285
Tabela K.4 – Estatísticas da latência de saída do tratador de interrupções 286
Tabela K.5 – Estatísticas da latência de pilha de IRQs 287
Tabela K.6 – Estatísticas da latência de *deferrable functions* 289
Tabela K.7 – Estatísticas da latência de preempção do núcleo desabilitada 291
Tabela K.8– Estatísticas da latência de escalonamento 293
Tabela K.9 – Estatísticas da latência de troca de contexto 295

Tabela K.10 – Estatísticas da latência total 297

LISTA DE SIGLAS

ACPI	*Advanced Configuration and Power Interface*
ACPI PMT	*ACPI Power Management Timer*
A/D	Analógica / Digital
API	*Application Programming Interface*
APIC	*Advanced Programmable Interrupt Controller*
APIC Local	*Local Advanced Programmable Interrupt Controller*
BIOS	*Basic Input Output System*
CDF	*Cumulative Distribution Function*
CISC	*Complex Instruction Set Computer*
CMOS	*Complementary Metal-Oxide-Semiconductor*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processor*
E/S	Entrada / Saída
FIFO	*First In-First Out*
HPET	*High Precision Event Timer*
IDT	*Interrupt Descriptor Table*
IPI	I*nterprocessor Interrupts*
IRQ	*Interrupt ReQuest*
IRQa	Interrupção que sinaliza a amostragem
IRQvn	Interrupção cujo vetor é o número n
ISA	*Industry Standard Architecture*
ISR	*Interrupt Service Routine*
MEi	Mudança de Estado i
NPDS	Não é de Processamento Digital de Sinais
PC	*Personal Computer*
PCo	*Program Counter*
PCI	*Peripheral Component Interconnect*
PDS	Processamento Digital de Sinais
PIC	*Programmable Interrupt Controller*
PIT	P*rogrammable Interval Timer*

POSIX	*Portable Operating Systems Interface for Unix*
PMT	*Power Management Timer*
RAM	*Random Access Memory*
RISC	R*educed Instruction Set Computer*
RTC	*Real Time Clock*
SMMP	*Single Memory Multi Processor*
SO	Sistema Operacional
SO-PDS	Sistema Operacional para Processamento Digital de Sinais
SO-TR	Sistema Operacional para Tempo Real
SP	*Stack Pointer*
TR	Tempo Real
TSC	*Time Stamp Counter*
OSADL	*Open Source Automation Development Lab*
x86	Microprocessadores de arquitetura Intel 64 e IA-32

SUMÁRIO

Capítulo 1 – Introdução 25

1.1 Introdução 25
1.2 Etapas do Estudo 26
1.3 Motivação do Estudo 27
1.4 Objetivos do Estudo 29
1.5 Limites do Estudo 30
1.6 Estrutura do Trabalho 31
1.7 Contribuições da Tese e Publicação 34
1.7.1 Contribuições 34
1.7.2 Publicações 35

Capítulo 2 – Contexto e Objetivo do Estudo 36

2.1 Introdução 36
2.2 Conceitos Básicos 36
2.2.1 Estruturas de um Programa PDS e de um Programa Tempo Real 36
2.2.2 Periodicidade da Leitura das Amostras 38
2.2.3 Controle da Freqüência de Amostragem 39
2.3 Contexto do Estudo 41
2.3.1 1º Exemplo 41
2.3.2 2º Exemplo 42
2.3.3 3º Exemplo 43
2.3.4 Especificando o Contexto do Estudo 44
2.3.5 Analisando o Objetivo Principal do Estudo 46
2.4 Placa Adaptadora 49
2.5 Modelo de Operação de um SO-PDS 49
2.5.1 Modelo de Dois Processos 50
2.5.2 Modo de Operação 50
2.6 Resumo 51

Capítulo 3 - Primeiros Requisitos para o SO-PDS 52
3.1 Introdução 52
3.2 Periodicidade da Leitura das Amostras 52
3.3 Controle da Freqüência de Amostragem 52
3.4 Interrupção E/S Conhecida (IRQa) e Não-Compartilhada 54
3.5 O Processo PDS em um Sistema Operacional Multiprocessos 56
3.6 Baixas Latências 59
3.7 O Processo PDS é Único 62
3.8 Estados do SO-PDS e Forma de Operação 63
3.9 Preempção de Processo Seletiva 64
3.10 Objetivo dos Requisitos 65
3.10.1 Escalonador Determinístico 66
3.10.2 Tratamento das Interrupções 67
3.10.3 Temporizadores de Alta Resolução 68
3.10.4 Núcleo com Preempção 68
3.11 Processo de Usuário x Módulo em Modo Supervisor 69
3.12 Requisitos Não-Funcionais 70
3.13 Resumo 72

Capítulo 4 – Modelo de Latência no Linux e Requisitos Adicionais para o SO-PDS 73
4.1 Introdução 73
4.2 Modelo de Latência 73
4.3 Latência de Tratamento de Interrupção 75
4.3.1 Latência da Leitura da Interrupção 75
4.3.2 Latência de Mascaramento 76
4.3.3 Latência da Fila do Controlador Programável de Interrupções 78
4.3.4 Latência de Operação do Microprocessador 80
4.3.5 Latência de Salvamento dos Registradores 80
4.3.6 Latência de Entrada no Tratador de Interrupções 82
4.3.6.1 Análise da Rotina: `do_IRQ` 82
4.3.6.2 Análise da Rotina: `__do_IRQ` 83

4.3.6.3 Análise da Rotina: `handle_IRQ_event` 83
4.3.7 Latência de Compartilhamento 84
4.4 Estabelecendo Limites entre Latências 85
4.5 Latência de Despacho 85
4.5.1 Latência de Saída do Tratador de Interrupções 86
4.5.1.1 Análise da Rotina: `handle_IRQ_event` 86
4.5.1.2 Análise da Rotina: `__do_IRQ` 87
4.5.1.3 Análise da Rotina: `do_IRQ` 87
4.5.2 Latência de Pilha de IRQs 89
4.5.3 Latência de *Deferrable Functions* 90
4.5.4 Latência de Preempção do Núcleo Desabilitada 90
4.5.5 Latência de Escalonamento 92
4.5.6 Latência de Troca de Contexto 92
4.5.7 Observações Finais 92
4.6 Latência de Execução 94
4.6.1 Interrupções E/S 94
4.6.2 *Deferrable Functions* 95
4.6.3 Memória Virtual 95
4.7 Resumo 96

Capítulo 5 – Fontes de Tempo e o Escalonador no Linux 97
5.1 Introdução 97
5.2 Dispositivos de Tempo na Arquitetura x86 97
5.2.1 *Real Time Clock* (RTC) 98
5.2.2 *Programmable Interval Timer* (PIT) 98
5.2.3 *CPU Local Timer* 99
5.2.4 *High Precision Event Timer* (HPET) 100
5.2.5 *Time Stamp Counter* (TSC) 101
5.2.6 *ACPI Power Management Timer* 102
5.3 Utilização do PIT e do *CPU Local Timer* pelo Linux 102
5.4 Conclusões Relacionadas aos Dispositivos de Tempo 103
5.4.1 *CPU Local Timer* Não Está Ativo 103

5.4.2 *CPU Local Timer* Está Ativo – Monoprocessador 104
5.4.3 *CPU Local Timer* Está Ativo – Multiprocessador 104
5.5 Importância dos Tratadores de Interrupção em Sistemas TR 105
5.6 O Escalonador do Linux 106
5.6.1 Classes de Processos 107
5.6.2 Análise 108
5.7 Resumo 109

Capítulo 6 – Medição da Latência no Tratamento das Interrupções E/S Externas no Linux 110
6.1 Introdução 110
6.2 Introdução ao Método de Medição 110
6.2.1 Núcleo Instrumentalizado 111
6.2.2 Rotina PDS 112
6.2.3 Aplicativo de Gerenciamento 113
6.3 Motivação 113
6.4 Análise do Método 114
6.4.1 Erro Sistemático 114
6.4.2 Latência de Hardware e Latência de Software 115
6.4.3 1ª Limitação - Latência de Hardware 115
6.4.4 2ª Limitação - Latência de Software 117
6.4.5 Resumo das Limitações 118
6.4.6 Observações Finais 119
6.5 Método de Medição 119
6.5.1 Aplicativo de Gerenciamento 120
6.5.2 Rotina PDS 121
6.5.3 Chamadas ao Sistema 122
6.5.4 Alteração no Tratador da Interrupção 124
6.5.5 Marcadores 125
6.5.6 Estrutura: `real_time_struct` 126
6.5.7 Colocação dos Marcadores 128
6.6 Metodologia Usada 128

6.6.1 Tipos de Medidas 129
6.6.2 Volume de Dados 130
6.7 Resultados 130
6.7.1 Latência de Hardware 131
6.7.2 Latência de Entrada no Tratador de Interrupções 133
6.7.3 Latência de Compartilhamento 135
6.7.4 Latência de Saída do Tratador de Interrupções 136
6.7.5 Latência de Pilha de IRQs 137
6.7.6 Latência de *Deferrable Functions* 139
6.7.7 Latência de Preempção do Núcleo Desabilitada 141
6.7.8 Latência de Escalonamento 143
6.7.9 Latência de Troca de Contexto 145
6.7.10 Latência Total 147
6.7.11 Subamostragem no Sistema 148
6.8 Conclusão 149
6.9 Resumo 150

Capítulo 7 – Alterações no Linux para Processamento o Digital de Sinais 151
7.1 Introdução 151
7.2 Latências – Revisão 152
72.1 Latência de Tratamento de Interrupção 152
7.2.2 Latência de Despacho 153
7.2.3 Latência de Execução 154
7.3 Tratador da Interrupção IRQa Simplificado - Método 1 155
7.4 Tratamento de Interrupção Postergado - Método 2 156
7.4.1 Medição dos Tempos 156
7.4.2 Lógica de Postergação 157
7.5 Troca Seletiva de Processo - Método 3 157
7.6 Desabilitação da Interrupção pelo Núcleo Postergada - Método 4 159
7.6.1 Medição do Tempo 160
7.6.2 Lógica de Postergação 161

7.6.3 Observações 161
7.7 Bloqueio das *Deferrable Functions* – Método 5 162
7.8 Alterações no Linux – 1ª Parte 162
7.8.1 Estrutura 162
7.8.2 Estados de Operação do SO-PDS 164
7.8.3 Processo PDS 165
7.8.4 Estrutura Programa PDS 166
7.8.5 Estados do Processo PDS 168
7.8.6 Chamadas ao sistema 169
7.9 Descrição das Alterações no Linux – 2ª Parte 170
7.9.1 Introdução 170
7.9.2 Análise da Rotina: `do_irq` 171
7.9.3 Análise da Função: `do_IRQ_PDS_N` 172
7.9.4. Análise da Função: do_IRQ_PDS_S 174
7.10 Descrição das Alterações no Linux – 3ª Parte 177
7.11 Comparação do Linux 2.6.25 Original e Modificado (SO-PDS) 179
7.11.1 Testes I 180
7.11.2 Testes II 181
7.12 Resumo 183

Capítulo 8 – Conclusões e Propostas de Estudos Futuros 184
8.1 Conclusões 184
8.2 Propostas de Estudos Futuros 185

Referências 186

Apêndice A – Uma Visão Geral do Núcleo do Linux 193
A.1 Introdução 193
A.2 O Modelo Processo/Núcleo 193
A.3 Fluxo de Controle do Núcleo 195
A.4 Processos 195
A.5 Threads do Núcleo (*Kernel Threads*) 197

A.6 *Deferrable Functions* 198
A.7 Núcleo Reentrante 198
A.8 Preempção do Núcleo 199
A.9 Módulos 200

Apêndice B – Introdução às Interrupções 202
B.1 Introdução 202
B.2 Definição 202
B.3 Classificação das Interrupções 203
B.4 Aspectos Básicos das Interrupções 204
B.5 Interrupções Aninhadas 205
B.6 Interrupções – Hardware da família x86 207
B.6.1 Sistema Monoprocessador com 8259A 207
B.6.2 Sistema Multiprocessador com APIC 209
B.6.3 Sistema Monoprocessador com APIC 211
B.6.4 A Tabela Descritora das Interrupções 211
B.7 Como o Hardware Trata uma Interrupção 212
B.7.1 Antes de Tratar a Interrupção 212
B.7.2 Depois de Tratar a Interrupção 213

Apêndice C – Interrupções no Linux 214
C.1 Introdução 214
C.2 Interrupções E/S no Linux – Aspectos Gerais 214
C.2.1 Divisão das Ações 214
C.2.2 Compartilhamento do Sinal de Interrupção 215
C.2.3 Funcionamento Básico 216
C.3 Aspectos Específicos da Arquitetura x86 217
C.4 Estruturas do Linux para Tratar as Interrupções E/S Externas 219
C.4.1 Descritor de Interrupção 219
C.4.2 Abstração do Hardware 220
C.4.3 O Manipulador da Interrupção (*Handler*) 221
C.4.4 Objeto para o Controlador Programável de Interrupções 222

C.4.5 Descritor para a Rotina de Serviço de Interrupção 222
C.4.6 Resumo Estruturas de Dados 223
C.4.7 Pilhas Múltiplas Modo Supervisor 224
C.5 Tratamento de Interrupção E/S Externa – 1ª Parte 224
C.5.1 Salvando os Registradores 225
C.5.2 Análise da Linha: `common_interrupt` 225
C.5.3 Análise da Rotina: `do_IRQ()` 227
C.5.4 Análise da Rotina: `__do_IRQ` 232
C.5.5 Análise da Função: `handle_IRQ_event` 237
C.6 Tratamento de Interrupção E/S Externa – 2ª Parte 238
C.6.1 Núcleo Compilado Sem Preempção 238
C.6.2 Núcleo Compilado Com Preempção 239
C.7 Tratamento de Interrupção E/S Locais 240

Apêndice D – Exemplo de Subamostragem 242

Apêndice E - Sistemas de Tempo Real 245
E.1 Introdução 245
E.2 Definição Sistema Tempo Real 245
E.3 Classificação dos Sistemas Tempo Real 245
E.4 Requisitos 246

Apêndice F - A Latência na Interrupção E/S Local 247
F.1 Introdução 247
F.2 Latência de Tratamento de Interrupção 247
F.2.1 Latência de Salvamento dos Registradores 248
F.2.2 Latência de Entrada no Tratador de Interrupções 249
F.2.3 Latência de Despacho 249

Apêndice G - Método de Medição - Aplicativo em Modo Usuário 251
G.1 Introdução 251

G.2 Bibliotecas e Definições 251
G.3 Estrutura 251
G.4 Rotina PDS 252
G.5 Aplicativo de Gerenciamento 254

Apêndice H - Instrumentalização do Núcleo 256
H.1 Introdução 256
H.2 Arquivo: `/linux/include/asm-x86/tempo_real.h` 256
H.2.1 Referências Externas 256
H.2.2 Estrutura: `real_time_struct` 256
H.3 Arquivo: `/linux/arch/x86kernel/tempo_real.c` 257
H.3.1 Bibliotecas 257
H.3.2 Marcador do PIT 257
H.3.3 Marcador TSC 258
H.3.4 Marcador Entrada ao Escalonador 258
H.3.5 Marcador Saída do Escalonador 258
H.3.6 Marcador Troca de Contexto 258
H.3.7 Inicia a Medição 259
H.3.8 Finaliza a Medição 259
H.3.9 Inicializa Estrutura na Carga do Sistema 260
H.3.10 Limpa Estrutura entre Medidas 260
H.3.11 Rotina Leitura de Dados 260
H.3.12 Rotina que Simula o *Driver* 261

Apêndice I - Resultados Testes Pentium IV 262
I.1 Introdução 262
I.2 Limitações Encontradas no Pentium IV 262
I.3 Resultados 263
I.3.1 Latência de Entrada no Tratador de Interrupções 264
I.3.2 Latência de Saída do Tratador de Interrupções 265
I.3.3 Latência de *Deferrable Functions* 266
I.3.4 Latência de Preempção do Núcleo Desabilitada 267

I.3.5 Latência de Escalonamento 268
I.3.6 Latência de Troca de Contexto 269

Apêndice J – Alterações no Núcleo para Implementar o Sistema Operacional PDS 270
J.1 Estrutura: `pds_real_time_struct` 270
J.2 Chamada ao Sistema 189 270
J.3 Rotina: `do_IRQ` 271
J.4 Função: `ver_codigo_irq` 271
J.5 Função: `ver_trata_irq` 272
J.6 Rotina: `mede_inicio_irq` 272
J.7 Rotina: `mede_fim_irq` 273
J.8 Rotina: `mask_ack_irqPDS` 274
J.9 Função: `do_IRQ_PDS_S` 274
J.10 Rotina: `atualiza_prox_irqa` 274
J.11 Rotina: `hlevel_irq_PDS` 275
J.12 Função: `handle_IRQ_evPDS` 275
J.13 Rotina: `driver_pds_isr` 275
J.14 Rotina: `bloqueia_irqs` 276
J.15 Rotina: `desbloqueia_irqs` 276
J.16 Grupo de Rotinas: `schedule_pds e duas de suas funções` 276
J.16.1 Rotina: `schedule_pds` 276
J.16.2 Função: `ver_schedule_pds` 277
J.16.3 Função: `pega_pds_task` 277
J.17 Rotina: `esvazia_fila` 278
J.18 Função: `driver_le_amostra` 278
J.19 Função: `driver_eof` 279
J.20 Função: `ver_pds()` 279

Apêndice K - Resultados 2º Teste no Pentium III 280
K.1 Introdução 280

K.2 Metodologia 280
K.3 Resultados 280
K.3.1 Latência de Hardware 281
K.3.2 Latência de Entrada no Tratador de Interrupções 283
K.3.3 Latência de Compartilhamento 285
K.3.4 Latência de Saída do Tratador de Interrupções 286
K.3.5 Latência de Pilha de IRQs 287
K.3.6 Latência de *Deferrable Functions* 289
K.3.7 Latência de Preempção do Núcleo Desabilitada 291
K.3.8 Latência de Escalonamento 293
K.3.9 Latência de Troca de Contexto 295
K.3.10 Latência Total 297

Capítulo 1 - Introdução

1.1 Introdução

Nos últimos anos houve um aumento da capacidade de processamento dos microcomputadores ou *Personal Computer* (PC), seja pelo aumento da freqüência de operação do microprocessador, seja pelo surgimento dos sistemas com multiprocessadores ou multinúcleos (*multicore*). (1), (2), (3)

Esse aumento da capacidade de processamento faz com que se pense na possibilidade de utilizar microcomputadores em aplicações que atualmente se baseiam em microprocessadores de uso específico como, por exemplo, microprocessadores específicos para o processamento digital do sinal ou *Digital Signal Processor* (DSP). Uma extensão desse raciocínio seria o uso de um microcomputador compatível com o IBM-PC (família x86 da Intel) e com o sistema operacional Linux para o processamento digital de sinais em tempo real.

Atualmente o campo do processamento digital de sinais é bastante extenso, incluindo aplicações que são de tempo real como, por exemplo, aplicativos para reconhecimento da fala (*speech recognition*), mas também incluindo aplicações que não são de tempo real como, por exemplo, aplicativos para análise de sedimentos no fundo do oceano. O foco deste trabalho é em aplicações de processamento digital de sinais em tempo real. (4), (5)

No desenvolvimento de uma aplicação para o processamento digital de sinais em tempo real, um dos fatores chave é o sistema operacional Linux, que, por ser de uso geral, não foi projetado para atender os requisitos de um aplicativo com essa finalidade e, portanto, apresenta certas limitações.

É interessante observar que o Linux também não foi projetado para atender os requisitos dos sistemas de tempo real e, conseqüentemente, apresenta limitações quando utilizado para executar aplicativos dessa natureza. Essas limitações do

Linux têm motivado grande número de estudos com a finalidade de melhorar este sistema operacional em aplicações que tenham restrições de tempo. Com esse propósito foi criada a fundação Real Time Linux Foundation, Inc. e desde 1999 tem-se realizado o Real Time Linux Workshop. (6), (7), (8), (9), (10)

O trabalho aqui desenvolvido se propõe a estudar o sistema operacional Linux 2.6.25 para a arquitetura x86 de 32 bits[1,2] e, a implementar alterações para diminuir as limitações deste sistema operacional na execução de aplicações voltadas para o processamento digital de sinais em tempo real. As alterações propostas possuem vários níveis de complexidade e podem ser implantadas de forma independente, sem prejuízo às principais funcionalidades do Linux. Este sistema modificado será chamado no decorrer do texto de sistema operacional para processamento digital de sinais ou, abreviadamente, de SO-PDS.

1.2 Etapas do Estudo

O projeto de alteração do núcleo do Linux foi dividido em quatro etapas como mostrado a seguir:

1- Primeiramente foram estabelecidos os requisitos funcionais para um SO-PDS genérico com base nas características de um aplicativo para o processamento digital de sinais em tempo real. O fato de um SO-PDS focar na execução de um tipo específico de aplicação permitiu, na 4º Etapa, o desenvolvimento de soluções de software que normalmente não são implementadas em outros tipos de sistemas operacionais como, por exemplo, os de tempo real.

2- Na segunda etapa, com base no funcionamento do microprocessador da família x86 e na análise dos programas fonte do núcleo do Linux, foram identificadas as várias fontes de latência no processamento digital de sinais. Esta análise serviu para:

[1] O estudo realizado limitou-se à arquitetura x86 de 32 bits, o que ficará subentendido como forma de simplificar o texto.
[2] A distribuição do Linux usada durante o estudo foi a Fedora.

a. Identificar possíveis pontos de melhoria do núcleo.
b. Identificar requisitos funcionais adicionais que são específicos para um SO-PDS baseado em Linux.
c. Desenvolver um modelo da latência do Linux no processamento digital de sinais.

3- Na terceira etapa, o núcleo do Linux 2.6.25 foi modificado pela colocação de funções para medição de tempo, em um processo chamado de instrumentalização do núcleo. Um objetivo desta alteração é medir, em diversas condições de carga do sistema, as várias componentes da latência conforme o modelo desenvolvido na 2ª etapa. Um segundo objetivo é o fornecimento de uma métrica para comparar o Linux na sua versão original com a versão modificada para o processamento digital de sinais em tempo real (ver 4ª etapa).

4- Na última etapa, o núcleo do Linux 2.6.25 foi modificado visando que o sistema operacional atenda requisitos funcionais de um sistema operacional para o processamento digital de sinais em tempo real com base no Linux.

1.3 Motivação do Estudo

A figura 1.1 mostra o modelo de um sistema típico de processamento de sinais digitais em tempo real, baseado em um microcomputador compatível com o IBM-PC. Pode-se observar que a lógica do processamento digital de sinais[3] (PDS) fica dividida entre o hardware (Hardware PDS) e o software (Aplicativo PDS), o que faz com que, normalmente, o hardware seja projetado para uma aplicação específica. Este tipo de sistema tem a desvantagem que mudanças na aplicação podem acarretar alterações no software e no hardware.

[3] Ao longo do texto será usada a abreviatura PDS para processamento digital de sinais.

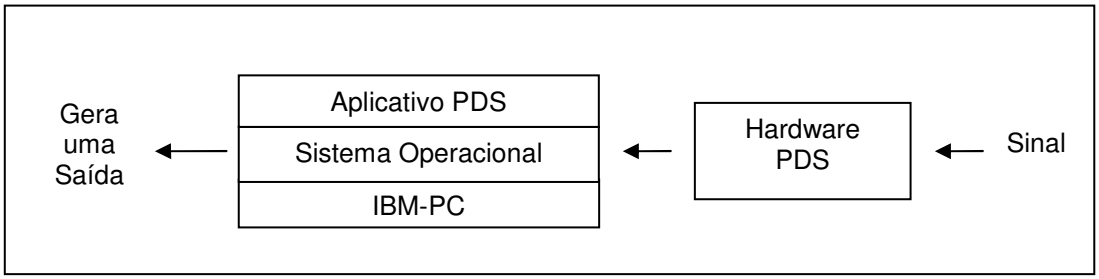

Figura 1.1 – Modelo de um sistema PDS em tempo real com base em um microcomputador do tipo IBM-PC com hardware PDS dedicado

A motivação deste trabalho é o desenvolvimento de sistemas como o mostrado na figura 1.2. Neste tipo de sistema, sinais analógicos ou digitais entram em uma placa adaptadora que tem como principal função digitalizar o sinal, se necessário, e passá-lo ao microcomputador. A placa adaptadora não executa nenhuma função relacionada com o processamento digital de sinais, o qual é realizado pelo aplicativo PDS, sendo executado em um SO-PDS compatível com o Linux. A idéia básica é que toda a lógica PDS esteja no software, o que torna o sistema mais versátil em casos de alteração da aplicação.

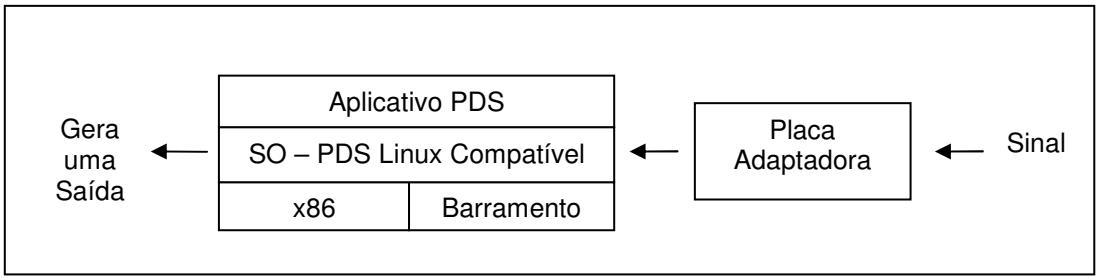

Figura 1.2 – Modelo de um sistema PDS em tempo real com base em um microcomputador do tipo IBM-PC com Linux e placa adaptadora

A motivação de usar o Linux para o desenvolvimento do SO-PDS decorre de ser um sistema operacional de código aberto, permitindo alterações no seu código fonte. Uma vantagem deste sistema operacional é a existência de ferramentas de programação que podem facilitar o desenvolvimento do aplicativo PDS.

A motivação para a utilização da arquitetura x86, justifica-se por ser uma arquitetura muito difundida e bem conhecida atualmente, o que permite o desenvolvimento de sistemas escaláveis de baixo custo. Pode-se ressaltar que como o Linux pode ser portado para outras arquiteturas, uma vez desenvolvido o

SO-PDS compatível com o Linux para o x86 de 32 bits, é possível uma extensão para outra arquitetura aproveitando-se o trabalho aqui desenvolvido.

1.4 Objetivos do Estudo

Os objetivos do estudo são:

1- Especificar os requisitos de um sistema operacional voltado para o processamento digital de sinais em tempo real (SO-PDS) com base no Linux. Na especificação do SO-PDS dois fatores foram considerados:

   a. Eficiência – Neste texto, o conceito de eficiência está relacionado à capacidade de atender as especificações de funcionamento de um aplicativo PDS em tempo real com a menor capacidade de hardware possível. Como apresentado nos próximos capítulos, este conceito está ligado à latência do SO-PDS para tratar os dados gerados pela placa adaptadora (item 1.3).

   b. Facilidade de Uso – Neste texto, o conceito de facilidade de uso está relacionado à capacidade do usuário de poder alternar entre aplicativos PDS em tempo real, sem a necessidade de simulações de escalonamento ou estudos de carga, para verificar se as restrições de tempo são atendidas. A idéia por trás deste conceito é que o próprio SO-PDS monitore o funcionamento do aplicativo PDS em tempo real.

2- Alterar o núcleo do Linux versão 2.6.25 para a arquitetura x86 de 32 bits de forma a atender os requisitos estabelecidos para um SO-PDS baseado no Linux. Sendo que este pode ser considerado o principal objetivo.

É importante ressaltar que este trabalho está limitado ao Linux e, conseqüentemente, a placa adaptadora apresentada no item 1.3 e citada em outros

capítulos do texto não é objeto de estudo. Como explicado no Capítulo 2, item 2.4, o motivo desta decisão é que existem soluções de mercado para essa placa.

1.5 Limites do Estudo

Uma limitação imposta ao trabalho é que soluções baseadas em sistemas do tipo *Single Memory Multi Processor* (SMMP) não são consideradas. A razão para esta limitação decorre de que em sistemas com mais de um processador (ou núcleo), certas características podem influenciar no projeto de um SO-PDS como, por exemplo:

- Carga da CPU[4] – Os sistemas multiprocessadores, de forma geral, possuem mecanismos para distribuir carga entre os vários processadores do sistema. Soluções baseadas nesses mecanismos de distribuição de carga para implementar um SO-PDS não são estudadas.

- CPU dedicada – Um fator a ser considerado é até que ponto não é vantajoso deixar um processador dedicado para executar o aplicativo PDS. De uma forma heurística pode-se dizer que fatores como freqüência máxima do sinal e o tempo para o processamento da amostra - os quais são específicos para uma aplicação PDS - têm grande impacto sobre esse fator.

É importante observar que um sistema operacional voltado para o processamento de sinais em tese não precisa ser multiprocessos, bastaria ter apenas o processo que executa o aplicativo PDS. Neste trabalho soluções dedicadas como essas não são estudadas, sendo assim, neste texto SO-PDS refere-se a um sistema operacional multiprocessos por convenção. O motivo para considerar um SO-PDS como sendo multiprocessos é a versatilidade de um sistema com essa característica, o qual permite, por exemplo, que tarefas, que não sejam de PDS em tempo real, possam ser executadas no mesmo hardware, aproveitando tempo de processamento ocioso.

[4] CPU é abreviatura de *Central Processing Unit* ou Unidade de Processamento Central.

A limitação imposta não invalida o trabalho aqui desenvolvido pelas razões expostas a seguir:

- Como mostrado no Capítulo 3, um dos objetivos do trabalho é diminuir a latência no tratamento da interrupção de hardware[5], usada para sinalizar a chegada de uma amostra do sinal digital a ser processado. Desta forma os resultados obtidos neste trabalho podem ser utilizados em sistemas multiprocessadores nos quais se utilize a interrupção de hardware para sinalizar o início do processamento da amostra. Neste caso, o sistema multiprocessador beneficia-se de uma melhoria projetada para um sistema monoprocessador, a qual, logicamente, não é ótima já que, como comentado, não aproveita recursos decorrentes do sistema ter mais de um processador.

- Atualmente máquinas monoprocessador ainda são utilizadas em várias áreas como, por exemplo, automação industrial onde normalmente aplicativos PDS são necessários. (11), (12), (13)

1.6 Estrutura do Trabalho

O trabalho está assim dividido:

- **Capítulo 2 –** Estabelece o contexto no qual o trabalho está inserido e aprofunda a discussão sobre o objetivo do estudo. O capítulo também trata de forma sucinta a placa adaptadora e faz uma primeira abordagem ao SO-PDS.

- **Capítulo 3** – Neste capítulo, tomando como base algumas características das aplicações PDS, os principais requisitos de um SO-PDS são estabelecidos. Também são analisadas limitações do Linux em aplicações que tenham restrições de tempo. Os requisitos levantados neste capítulo servem como base para as modificações a serem

[5] A noção de interrupção de hardware é apresentada no Apêndice B, item B.2.

propostas com o intuito de melhorar o Linux na execução de aplicações PDS em tempo real, as quais são tratadas no Capítulo 7.

- **Capítulo 4** – Neste capítulo o foco principal é a latência do Linux quando um processo PDS é colocado no estado de em execução. As várias fontes de latência são identificadas de acordo com a sua origem e, como resultado desse estudo, é desenvolvido um modelo de latência para o Linux. Também são estabelecidos requisitos adicionais para um SO-PDS baseado no Linux, os quais surgem do estudo mais detalhado da forma como esse sistema operacional trata as interrupções.

- **Capítulo 5** – Este capítulo descreve as várias fontes de tempo na arquitetura x86, como são usadas pelo Linux e as limitações para o desenvolvimento de um SO-PDS com base nesse sistema operacional. Também trata aspectos do escalonador relacionados com os processos de tempo real no Linux 2.6.25.

- **Capítulo 6** – Neste capítulo é proposto um método baseado em software para medir as latências do núcleo do Linux quando um processo PDS é colocado em execução. O método é usado para avaliar as latências do Linux 2.6.25.

- **Capítulo 7** – Neste capitulo são apresentadas soluções para melhorar o sistema operacional Linux 2.6.25 na execução de aplicações voltadas para o processamento digital de sinais em tempo real.

- **Capítulo 8** – Este capítulo apresenta as conclusões obtidas pelos estudos realizados e sugestões de estudos futuros.

- **Apêndice A** – Contém informações básicas sobre o núcleo do Linux e estabelece alguns dos conceitos usados neste trabalho, como preempção e núcleo reentrante, entre outros.

- **Apêndice B** – Contém uma introdução básica sobre interrupções e descreve como é feito o tratamento de uma interrupção pelo hardware da família x86.

- **Apêndice C** – Descreve o tratamento de uma interrupção pelo Linux 2.6.25. São apresentadas as estruturas usadas, aspectos específicos da arquitetura x86 relacionados com o sistema operacional e uma descrição de como é feito o tratamento de uma interrupção pelo núcleo.

- **Apêndice D** – Este apêndice traz uma introdução ao conceito de subamostragem.

- **Apêndice E** – O apêndice apresenta uma definição para sistema de tempo real e uma classificação para este tipo de sistema. Também trata os tipos de requisitos encontrados para especificar um sistema.

- **Apêndice F** – Este apêndice estabelece um modelo para a latência no tratamento de uma interrupção Entrada/Saída (E/S) local, sendo uma extensão do Capítulo 4.

- **Apêndice G** – Neste apêndice são apresentados os programas fonte que foram usados para as medidas das latências apresentadas no Capítulo 6.

- **Apêndice H** – Neste apêndice são apresentados os programas fonte que implementam a instrumentalização do Linux 2.6.25, conforme descrito no Capítulo 6.

- **Apêndice I** – Este apêndice apresenta alguns resultados para as latências do Pentium IV.

- **Apêndice J** – Neste apêndice são apresentados os programas fonte que modificam o núcleo do Linux 2.6.25 com a finalidade de melhorar esse

sistema operacional na execução de aplicações de processamento digital de sinais em tempo real.

1.7 Contribuições da Tese e Publicação

1.7.1 Contribuições

As principais contribuições desta tese são:

1- Estabelecimento da noção de sistema operacional para processamento digital de sinais através da definição de seus principais requisitos funcionais.

2- Estabelecimento de um novo modelo para a latência no tratamento de interrupções E/S no Linux, mais completo do que o atualmente encontrado na literatura.

3- Desenvolvimento de um método baseado na instrumentalização do Linux 2.6.25, para a medição de várias componentes da latência no tratamento de interrupções E/S externas no Linux.

4- Apresentação de resultados quantitativos para várias componentes da latência no Linux 2.6.25, usando o método de medição proposto.

5- Proposição de cinco métodos para melhorar o Linux 2.6.25 na execução de aplicativos de processamento digital de sinais em tempo real, a saber:

   a. Tratador da Interrupção IRQa Simplificado,
   b. Tratamento de Interrupção Postergado,
   c. Troca Seletiva de Processo,
   d. Desabilitação da Interrupção pelo Núcleo Postergada,
   e. Bloqueio das *Deferrable Functions*.

6- Implementação e teste de quatro dos cinco métodos propostos. Os métodos implementados são:

   a. Tratador da Interrupção IRQa Simplificado,
   b. Tratamento de Interrupção Postergado,
   c. Troca Seletiva de Processo,
   d. Bloqueio das *Deferrable Functions*.

1.7.2 Publicações

O modelo de latência desenvolvido no Capítulo 4 e os resultados obtidos no Capítulo 6 foram usados para escrever, em co-autoria com o Professor Doutor Phillip M.S. Burt, um artigo intitulado "*A Latency Model of Linux 2.6 for Digital Signal Processing in Real Time*", o qual foi aceito para publicação na 12º Real Time Linux Workshop, que é um dos principais simpósios na área do Linux em tempo real. O *link* do simpósio é <www.osadl.org/RTLWS-Submitted-Papers.rtlws12-submitted-papers.0.html>.

Capítulo 2 - Contexto e Objetivo do Estudo

2.1 Introdução

Neste capítulo, inicialmente são apresentados alguns conceitos básicos usados ao longo do texto. A continuação é apresentada o contexto e analisado o objetivo do estudo com mais detalhes do que no capítulo anterior. O capítulo termina exemplificando a placa adaptadora e colocando as noções iniciais sobre o SO-PDS.

2.2 Conceitos Básicos

Neste item são apresentados alguns conceitos básicos, vistos a seguir, que são:

1- Estruturas de um programa PDS e de um programa tempo real.
2- Periodicidade da leitura das amostras.
3- Controle da freqüência de amostragem.

2.2.1 Estruturas de um Programa PDS e de um Programa Tempo Real

Um SO-PDS deve levar em conta a estrutura particular dos programas PDS, como forma de atender com mais eficiência os requisitos desse tipo de programa do que um sistema operacional de uso geral. A estrutura característica de um programa PDS pode ser vista no diagrama de blocos da figura 2.1.

Na figura 2.1 pode-se ver que depois de uma rotina de início para, por exemplo, iniciar as estruturas de dados, o programa entra em um laço. No laço, basicamente são efetuadas quatro operações: ler a amostra[1] de entrada, processar a amostra, gerar o dado de saída e verificar se é o final do processamento. O final do processamento pode ser sinalizado por um dado particular de entrada ou por

[1] Neste texto será usado o termo amostra em uma forma mais geral, ou seja, pode ser um único dado (ex: caractere) ou um conjunto de dados (ex: bloco).

meio de uma sinalização do SO-PDS. Neste último caso, o programa não precisa testar no laço a ocorrência do fim do processamento.

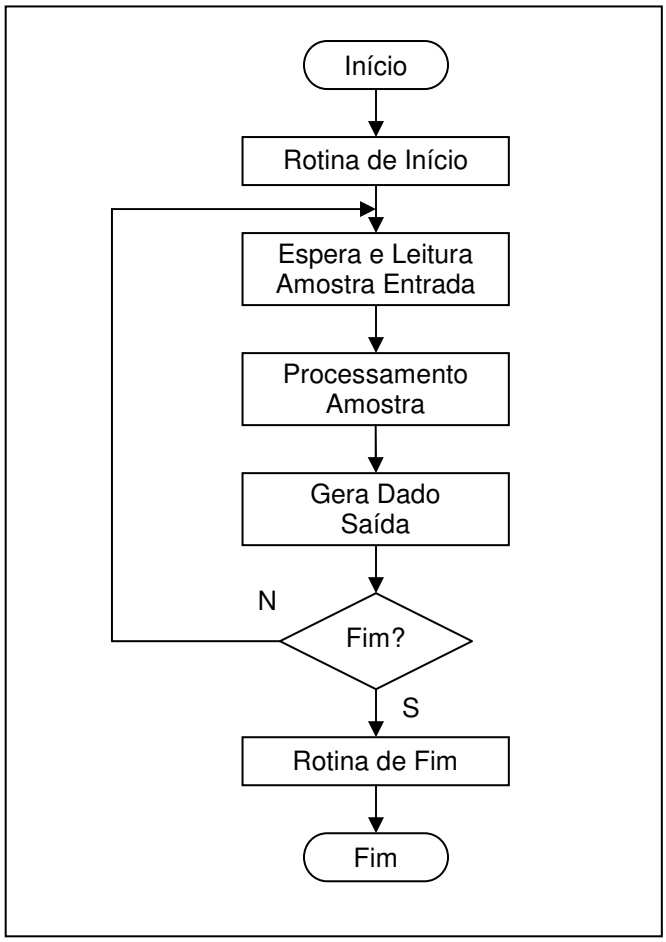

Figura 2.1 – Diagrama de blocos de um programa típico de PDS

No diagrama de blocos da figura 2.2 a seguir, pode-se observar que a estrutura de um programa de tempo real (TR) é semelhante à estrutura de um programa PDS. (14), (15)

Como se pode ver na figura, o programa fica à espera de um sinal e, quando o sinal chega, um processamento é realizado e, na seqüência, uma ação a qual deve ser executada em um tempo máximo que é especificado de acordo com a aplicação. A semelhança nas estruturas dos programas faz com que muitos dos requisitos de um SO-PDS coincidam com os requisitos necessários para um sistema

operacional de tempo real (SO-TR). De fato, será visto no item 2.2.2 que o primeiro pode ser considerado um caso particular do segundo.

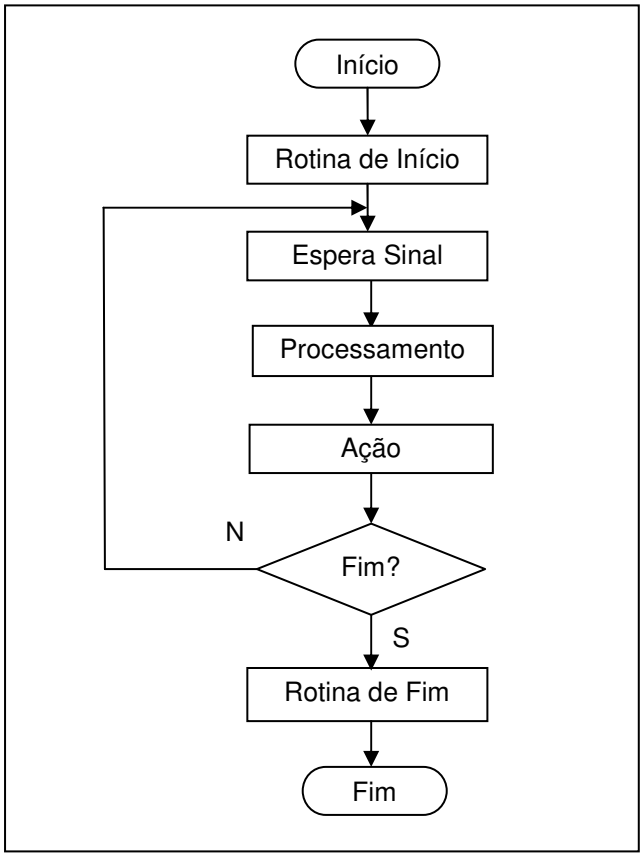

Figura 2.2 – Diagrama de blocos de um programa típico de tempo real

2.2.2 Periodicidade da Leitura das Amostras

Em se tratando de processamento digital de sinais, a freqüência de amostragem tem grande importância, já que, normalmente em uma aplicação PDS, os sinais, sejam analógicos ou digitais, são amostrados com um período fixo de tempo previamente conhecido, período este estabelecido pela freqüência máxima do sinal e pelo Teorema da Amostragem. (16), (17)

Este fato implica que se pode assumir que um programa PDS executa a instrução de leitura com um período fixo (T_A) que é previamente conhecido quando o programa inicia a sua execução. Período este que estabelece um tempo máximo

para o programa: processar a amostra, gerar uma saída e ficar à espera da próxima amostra.

Embora em um programa TR o sinal que inicia o processamento (ver figura 2.2) possa ser periódico, de forma geral as aplicações de tempo real não fazem assunções em relação à periodicidade do sinal e, conseqüentemente, o SO-TR não pode assumir esta restrição. Esta restrição de processar periodicamente o sinal de entrada faz com que o SO-PDS possa ser considerado um caso particular do SO-TR.

2.2.3 Controle da Freqüência de Amostragem

Decorre do visto no item anterior que a leitura da amostra por um programa PDS deve ser controlada por uma fonte de tempo. Em sistemas operacionais isto pode ser feito de duas formas: *Push-In* e *Pull-In*. (18)

No modo *Push-In* a fonte de tempo usada para controlar o período de amostragem está na placa adaptadora. A placa sinaliza ao microprocessador o instante de leitura da amostra e, portanto, em termos de sistema, é a placa adaptadora que controla a amostragem. A figura 2.3 mostra esta forma de funcionamento.

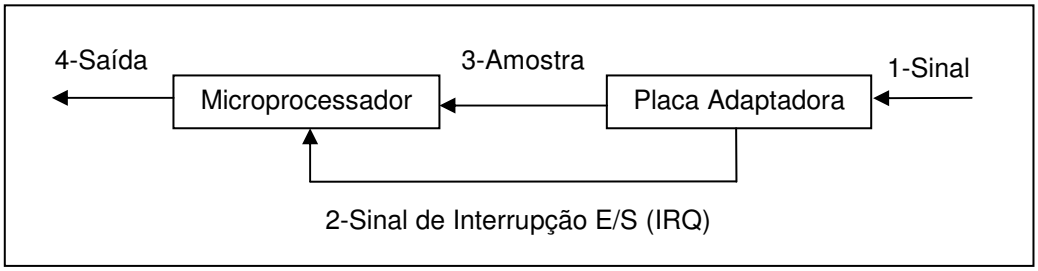

Figura 2.3 - Operação em modo *Push-In*

Com base na figura 2.3 pode-se descrever este modo de operação:

1- A placa adaptadora recebe o sinal a ser processado. Este sinal é amostrado para se obter uma amostra que é transmitida ao microprocessador.

2- A placa adaptadora gera um sinal de interrupção E/S[2] (IRQ) ao microprocessador para sinalizar o instante da amostragem.
3- O microprocessador lê a amostra.
4- A amostra é tratada e um dado de saída é gerado.

No modo *Pull-In* a fonte de tempo usada para controlar o período de amostragem é um mecanismo de tempo interno do microprocessador ou do conjunto de chips (*chip set*) do microcomputador. Neste modo de operação a placa adaptadora não tem participação ativa no controle do período de amostragem.

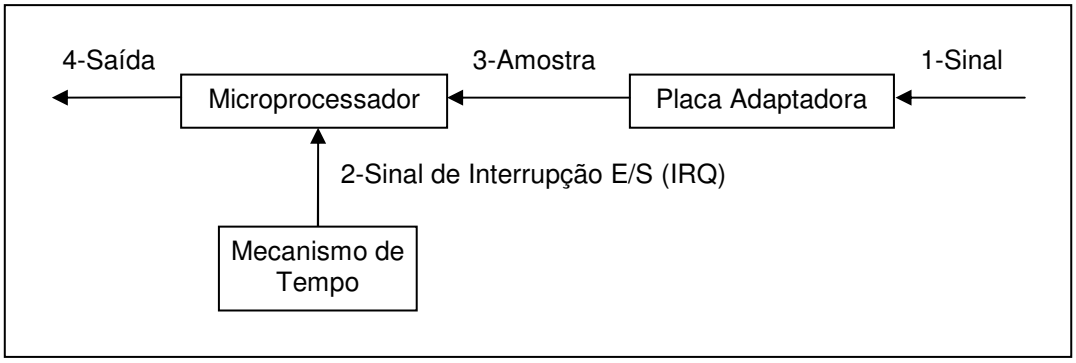

Figura 2.4 - Operação em modo *Pull-In*

Com base na figura 2.4 pode-se descrever este modo de operação:

1- A placa adaptadora recebe o sinal a ser processado. Quando for necessário, a placa digitaliza o sinal que é transmitido ao microprocessador.
2- O mecanismo de tempo que controla a freqüência de amostragem gera um sinal de interrupção E/S (IRQ) ao microprocessador para sinalizar o instante da amostragem.
3- O microprocessador lê a amostra.
4- A amostra é tratada e um dado de saída é gerado.

[2] Neste texto o termo interrupção E/S é usado como sinônimo de interrupção mascarável. Ver Apêndice B, item B.3.

2.3 Contexto do Estudo

Neste item, inicialmente são apresentadas três aplicações envolvendo o processamento digital de sinais, como forma de definir o contexto no qual se encaixa o estudo e, a seguir, apresentado o objetivo.

2.3.1 1º Exemplo

O diagrama de blocos da figura 2.5 representa um sistema para o reconhecimento de comandos de voz em aviões de combate. (4)

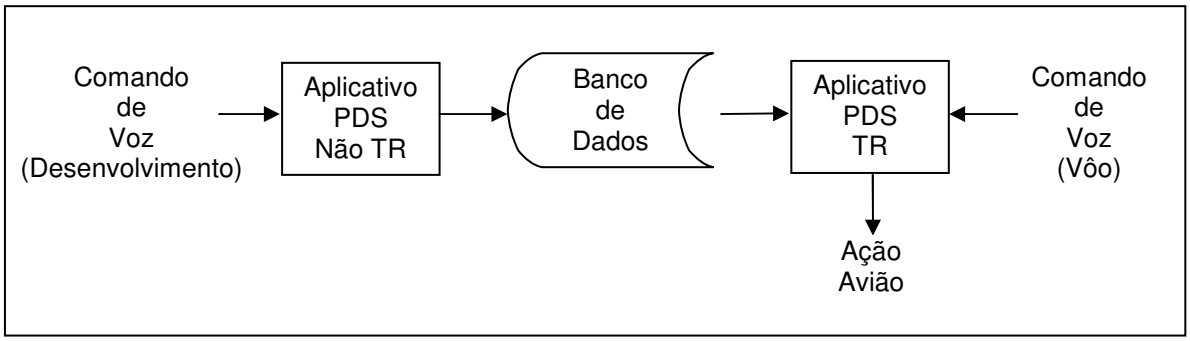

Figura 2.5 – Diagrama de blocos de um sistema para reconhecimento de comandos de voz em aviões de combate

Como mostrado na figura 2.5, o sistema é composto basicamente de dois aplicativos:

1- Um aplicativo PDS que não é de tempo real, o qual tem como finalidade analisar comandos de voz, durante o desenvolvimento do sistema, para a obtenção de um banco de dados de fonemas, a ser usado para, em vôo, identificar o comando.

2- Um segundo aplicativo PDS que é de tempo real o qual, em vôo, tem a função de: a) analisar o comando de voz, b) identificar a ação a ser realizada usando o banco de dados e d) executar a ação.

A finalidade deste primeiro exemplo é distinguir os aplicativos PDS de tempo real daqueles que não são de tempo real. Embora os segundos sejam encontrados em diversas áreas como, por exemplo, geologia e oceanografia, este estudo foca nos primeiros. A seguir são apresentados dois exemplos de aplicações PDS em tempo real, como forma de contextualizar o trabalho.

2.3.2 2º Exemplo

O diagrama de blocos da figura 2.6 representa uma estação rádio-base desenvolvida pela empresa Vanu Inc. com base em microcomputadores IBM-PC rodando Linux. (19), (20)

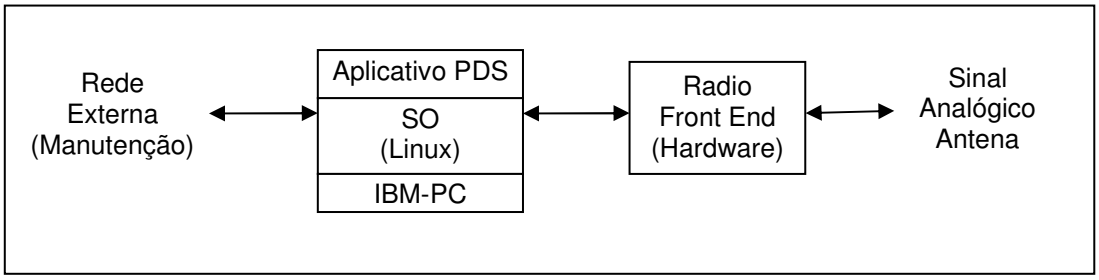

Figura 2.6 – Diagrama de blocos de uma estação rádio base desenvolvida com microcomputador do tipo IBM-PC

Como mostrado na figura, o sistema é composto por dois blocos principais:

1- Um bloco de hardware dedicado que implementa o *Rádio Front End*. As principais funções deste bloco são: conversão de freqüência, filtragem, conversão analógica digital (A/D) e canalização do sinal. Este bloco não é desenvolvido pela Vanu Inc., sendo adquirido de terceiros.

2- O segundo bloco é composto por um servidor Pentium Dual Core (3,4 GHz) com o sistema operacional Linux, no qual é executado o aplicativo PDS desenvolvido pela Vanu Inc. que decodifica o sinal digital.

Uma vantagem deste produto é que todo o aplicativo PDS foi desenvolvido em C++ o que facilita a sua atualização. Na figura 2.6 pode-se observar que o servidor está ligando a uma rede externa, o que permite a manutenção remota do aplicativo PDS.

2.3.3 3º Exemplo

O terceiro exemplo pode ser visto no diagrama de blocos da figura 2.7, o qual representa um receptor de ondas curtas, médias e longas chamado DRM Software Radio, desenvolvido pela empresa VT Merlin Communications. (21)

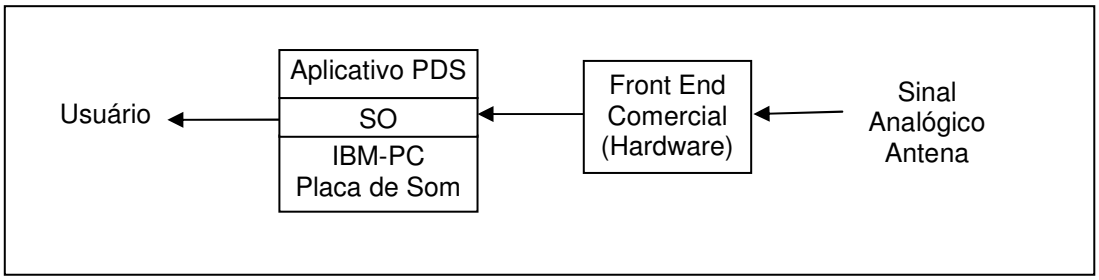

Figura 2.7 – Diagrama de blocos de um rádio receptor desenvolvido com microcomputador do tipo IBM-PC

O sistema da figura 2.7 está dividido em dois blocos principais:

1- Um bloco de hardware que é responsável por fazer a translação de freqüência do sinal analógico da antena para 12 kHz. Este sinal de 12 kHz entra na placa de som do microcomputador compatível com o IBM-PC.

2- Um segundo bloco que é composto pelo microcomputador com o aplicativo PDS (chamado DRM Software Radio) o qual decodifica o sinal de entrada, gera o sinal de áudio e fornece um painel virtual para a interface com o usuário.

2.3.4 Especificando o Contexto do Estudo

As duas aplicações apresentadas nos itens 2.3.2 e 2.3.3 possuem as seguintes características:

1- Envolvem o tratamento digital de sinais.
2- São de tempo real e podem ser classificadas como tempo real suave[3] (*soft real-time*).
3- O processamento digital do sinal está dividido entre um hardware dedicado e um software que está sendo executado em um microcomputador.

Essas aplicações podem ser enquadradas no modelo mostrado na figura 2.8, o qual foi visto de forma superficial no Capítulo 1 (ver figura 1.1).

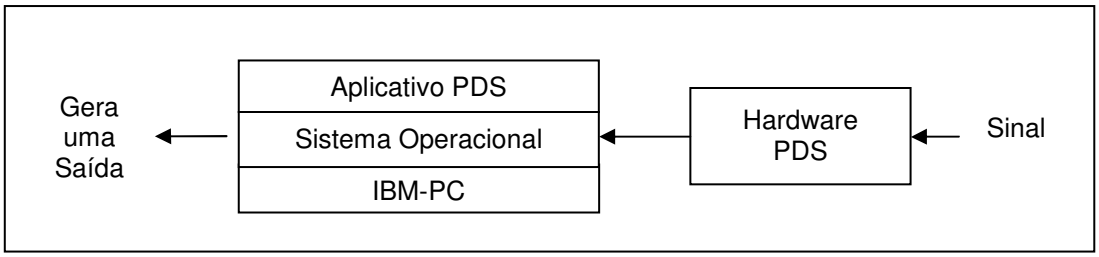

Figura 2.8 - Modelo de um sistema para processamento digital de sinais baseado em um microcomputador do tipo IBM-PC com hardware PDS dedicado

Neste modelo de sistema, sinais analógicos ou digitais entram em um hardware que efetua, em tempo real, todo ou parte do processamento digital de sinais. O sinal digital resultante deste processamento é passado ao microcomputador do tipo IBM-PC, por meio de uma interface padrão (ex: *Peripheral Component Interconnect* - PCI) ou porta (ex: porta serial). Neste ambiente, o aplicativo PDS pode executar a interface com o usuário, executar um segundo processamento ou gerar uma saída. Pode-se observar que a lógica do processamento fica parte no hardware (Hardware PDS) e parte no software

[3] A definição de tempo real suave pode ser encontrada no Apêndice E, item E.3.

(Aplicativo PDS). Neste tipo de sistema, o hardware é normalmente específico para uma aplicação PDS, o que torna o sistema pouco versátil.

Os exemplos aqui ficaram restritos à área de telecomunicações, mas sistemas como o apresentado na figura 2.8 podem, também, ser encontrados em outras áreas como, por exemplo, instrumentação médica e automação industrial. (3), (11), (12), (13)

O desenvolvimento de processadores de uso geral mais rápidos e conseqüentemente de microcomputadores mais rápidos, motiva o desenvolvimento de sistemas nos quais o processamento digital de sinais seja executado totalmente pelo aplicativo PDS, ou seja, por software. Um modelo possível para este tipo de sistema - descrito de forma sucinta no Capítulo 1 (ver figura 1.2) - pode ser visto na figura 2.9.

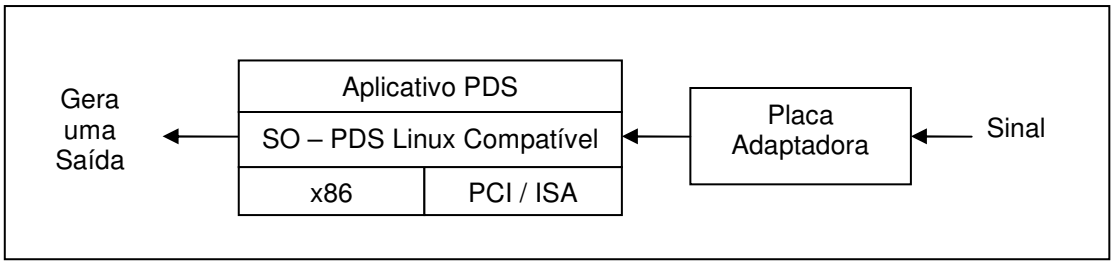

Figura 2.9 - Modelo de um sistema para processamento digital de sinais baseado em um microcomputador do tipo IBM-PC com Linux e placa adaptadora

No sistema da figura 2.9, sinais analógicos ou digitais entram em uma placa adaptadora a qual, se necessário faz a amostragem (tipicamente com uma freqüência de amostragem constate) e a digitalização do sinal. As amostras digitalizadas do sinal são passadas ao sistema de arquitetura x86 com o SO-PDS (compatível com o Linux) por meio da interface padrão PCI ou padrão *Industry Standard Architecture* (ISA). Neste ambiente, o aplicativo - que contém toda a lógica do processamento digital do sinal - processa a amostra de entrada em tempo real, gerando um dado de saída, por exemplo, para o usuário. A idéia básica neste sistema é que toda a lógica PDS esteja no software e não no hardware. Isto faz com

que, dentro de certos limites da placa adaptadora[4], o sistema possa tratar, em tempo real, sinais vindo de diferentes tipos de fontes (ex: um microfone ou um sensor), apenas trocando-se o aplicativo PDS, o que torna o sistema mais versátil do que o apresentado na figura 2.8.

Pode-se resumir o contexto do trabalho como: sistemas que executam processamento digital de sinais em tempo real, do tipo tempo real suave (*soft real-time*), voltados para aplicações corporativas (exemplo do item 2.3.2) e aplicações para o usuário final (exemplo do item 2.3.3).

2.3.5 Analisado o Objetivo Principal do Estudo

O desenvolvimento de um sistema como da figura 2.9 pode ser dividido em dois módulos.

O primeiro módulo é o hardware, o qual é representado pela placa adaptadora. Como já foi comentado, o objetivo é que a placa adaptadora não faça parte do processamento digital. Esta placa possui duas funções básicas: fazer a conversão analógica digital (se necessário) e gerar os sinais para o barramento padrão (ex: PCI). A placa adaptadora não é tratada neste trabalho, mas algumas informações básicas são apresentadas no item 2.4.

O segundo módulo é todo o software envolvido, o qual pode ser dividido em 3 (três) camadas como mostrado na figura 2.10. Na figura tem-se:

1- Chamada de *driver*, a 1ª camada é composta pelo conjunto de rotinas que fazem a interface com a placa adaptadora. Esta camada depende do modelo de placa adaptadora usada.

2- A 2ª camada é composta pelas rotinas do SO-PDS que fazem a interface entre o *driver* e o aplicativo PDS.

[4] A limitação da placa é dada pela impedância de entrada da mesma e pela freqüência máxima do dispositivo de conversão analógico digital.

3- A 3ª camada é programa PDS, o qual depende da aplicação que está sendo executada.

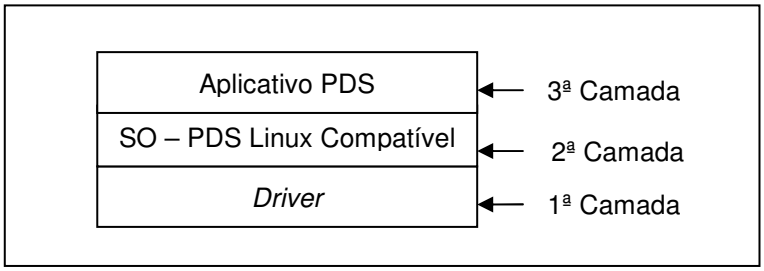

Figura 2.10 – Modelo de camadas para o software de processamento digital de sinal

O estudo deste trabalho foca na 2ª camada que, como já foi comentado no Capítulo 1, é um fator limitante para o tipo de sistema proposto, pois o sistema operacional Linux não foi projetado especificamente para aplicações PDS em tempo real.

Embora idealmente a meta seja o desenvolvimento de sistemas como toda a lógica PDS no software (figura 2.9), pode-se observar que sistemas como o da figura 2.8 podem tirar vantagem de uma 2ª camada mais eficiente para, por exemplo, implementar um hardware PDS com menos funcionalidades.

Pode-se resumir, como foi colocado no Capítulo 1 (item 1.4), o objetivo principal do estudo da seguinte maneira: alterar o núcleo do Linux versão 2.6.25 para a arquitetura x86 de 32 bits de forma a atender os requisitos estabelecidos para um SO-PDS baseado no Linux. No item 2.5 será apresentada uma introdução ao funcionamento de um sistema operacional para o processamento digital de sinais (SO-PDS) e aos temas tratados no texto.

Para terminar este item, os conceitos apresentados no Capítulo 1 (item 1.4), eficiência e facilidade de uso, são explorados com base nos exemplos.

Pode-se constatar que o exemplo do item 2.3.1 é uma aplicação militar, o do item 2.3.2 é uma aplicação corporativa e o exemplo do item 2.3.3 é uma aplicação para o usuário final.

A – Eficiência

Em aplicações corporativas ou militares existe a possibilidade do hardware se adequar ao software, ou seja, se o aplicativo PDS em tempo real requer certa capacidade de processamento para atender a freqüência de amostragem especificada, o hardware pode ser escolhido de acordo.

Por sua vez, em aplicações voltadas para o usuário final é vantajoso, do ponto de vista comercial, que a capacidade necessária de processamento para atender a freqüência de amostragem do aplicativo PDS seja a menor possível. Portanto, é interessante que o SO-PDS seja eficiente na execução de aplicativos PDS em tempo real.

Uma vantagem adicional de um SO-PDS eficiente é que, dada uma plataforma de hardware, maior é a freqüência de amostragem que pode ser tratada e, portanto, maior é o número de aplicações PDS em tempo real que podem ser executadas nessa plataforma.

B - Facilidade de Uso

Em aplicações corporativas ou militares existe a possibilidade de conhecer previamente a carga máxima do sistema. Essas aplicações podem se beneficiar de ferramentas para simulação de escalonamento, as quais verificam se a freqüência de amostragem especificada para a aplicação PDS é atendida na plataforma de hardware escolhida e para a carga máxima do sistema.

Em aplicações voltadas para o usuário final não há um conhecimento prévio da carga máxima, a qual, inclusive, pode variar com o tempo. Esse tipo de aplicação PDS em tempo real dificilmente pode-se valer das ferramentas de simulação – por sua especificidade - para verificar se a freqüência de amostragem é atendida no hardware utilizado.

Para as condições descritas acima, é vantajoso que o SO-PDS tenha um mecanismo, executado em tempo real, para monitorar se a freqüência de amostragem é atendida.

2.4 Placa Adaptadora

Embora este trabalho foque unicamente no sistema operacional, a título de informação a placa adaptadora é abordada de forma sucinta neste item.

Na figura 2.11 pode-se ver o diagrama de blocos de um exemplo de placa adaptadora. Tem-se um primeiro bloco que digitaliza o sinal, que é suposto analógico. O segundo bloco simplesmente gera os sinais necessários para o padrão do barramento, que neste caso é suposto PCI. No exemplo da figura 2.11 pode-se observar que não há lógica PDS incluída na placa.

Para este segundo bloco existem soluções que podem ser adaptadas como, por exemplo, a placa da empresa ASIC Design Services chamada PCILiteDS que fornece uma interface *backend* compatível com a de um microprocessador Intel. (22)

Uma outra opção é a placa de aquisição KPCI-3101 da empresa Keithley Instruments Co. (23)

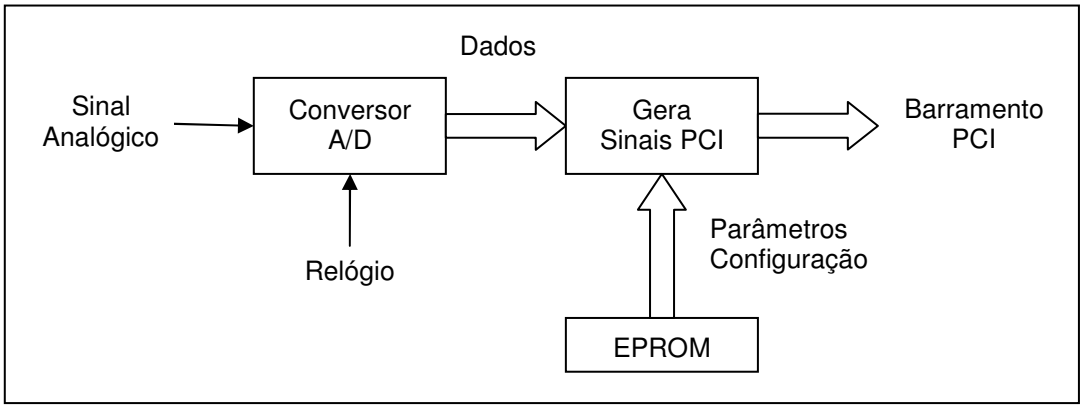

Figura 2.11 – Diagrama de blocos de uma placa adaptadora

2.5 Modelo de Operação de um SO-PDS

Neste item, apresenta-se o modo de operação característico de um sistema operacional para processamento digital de sinais. Também é feita uma introdução

aos temas tratados no decorrer do texto. Com a finalidade de facilitar a explanação, primeiramente um modelo simplificado é estabelecido.

2.5.1 Modelo de Dois Processos

O modelo consiste de um SO-PDS, o qual é multiprocessos por convenção, no qual estão sendo executados apenas dois processos. Os dois processos são definidos como segue:

- Um processo que faz o tratamento do sinal digital, o qual é chamado de processo PDS. Este processo se caracteriza por ler uma amostra de entrada, processar a amostra e gerar um dado de saída. Como no tratamento de sinais digitais normalmente há especificações em relação à freqüência de amostragem, este processo possui requisitos de tempo estritos, ou seja, deve ler uma amostra em períodos de tempo fixos (T_A) que corresponde a uma freqüência de amostragem (f_A).

- Um processo que não tem requisitos de tempo, que é chamado de processo NPDS, o qual é uma abstração que engloba todos os processos, inclusive os processos do sistema ou *daemons*, que estão compartilhando o sistema multiprocessos com o processo PDS.

2.5.2 Modo de Operação

Do ponto de vista de operação, um SO-PDS pode ser exemplificado como na figura 2.12. Na figura tem-se:

1- Em t_0 chega o sinal de amostragem. O SO-PDS suspende o processo NPDS que está de posse da CPU e passa a posse da CPU ao processo PDS.

2- De posse da CPU, o processo PDS lê a amostra da placa adaptadora e a trata em um tempo de processamento de amostra T_{PA}, onde por requisito de sistema tem-se $T_A > T_{PA}$.

3- Em t_1 o processo PDS libera a posse da CPU e fica bloqueado à espera do próximo sinal de amostragem. No intervalo de tempo restante (T_R) até o próximo sinal de amostragem em t_2, o SO-PDS passa a posse da CPU para o processo NPDS maximizando assim o uso do sistema.

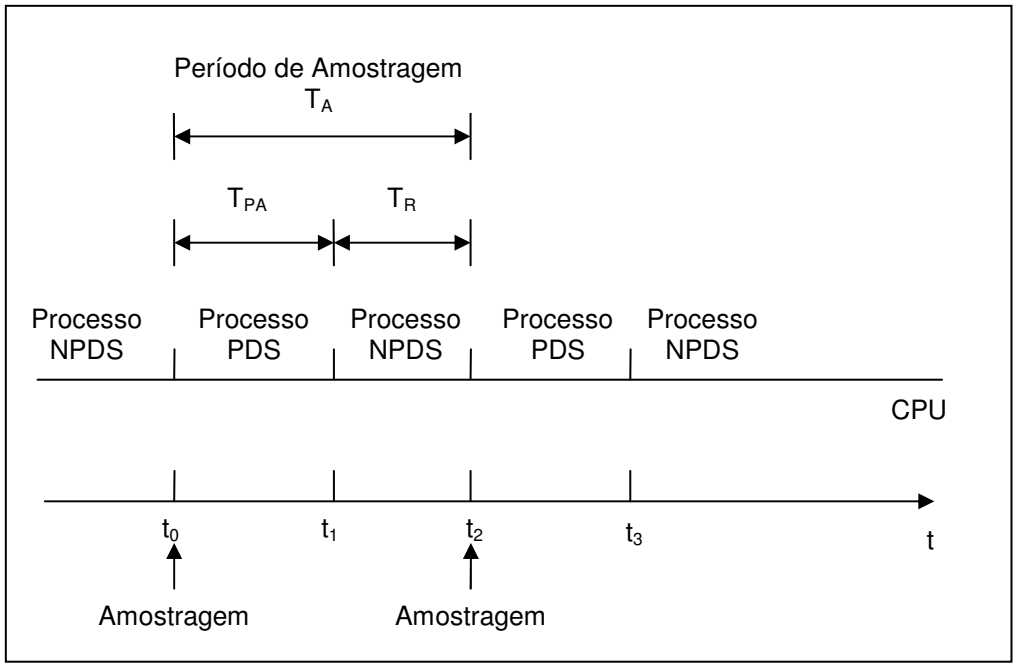

Figura 2.12 – Modelo de operação do sistema operacional PDS

Nesta primeira abordagem, para simplificar o modelo, os tempos de troca de processo podem ser considerados como estando inclusos em T_{PA} e T_R. Este trabalho estuda aspectos relacionados ao sistema operacional, que já podem ser levantados com base neste modelo simplificado:

- Como o SO-PDS controla e sinaliza o período de amostragem.
- As latências envolvidas no SO-PDS, como por exemplo, o tempo necessário para a troca de processo.

2.6 Resumo

Neste capítulo foi caracterizado o contexto do estudo e, também, foram apresentados os principais temas a serem discutidos no decorrer do texto: controle da sinalização do período de amostragem e latência na troca de processo.

Capítulo 3 - Primeiros Requisitos para o SO-PDS

3.1 Introdução

Neste capítulo são estabelecidas as simplificações e os requisitos de um SO-PDS. As simplificações são estabelecidas com base na estrutura característica de um programa PDS e nas situações mais comuns encontradas em processamento digital de sinais. Os requisitos são estabelecidos a partir de certos recursos e facilidades que um sistema SO-PDS deve oferecer.

3.2 Periodicidade da Leitura das Amostras

Como foi visto no Capítulo 2, item 2.2.2, em se tratando de processamento digital de sinais, pode-se assumir que um programa PDS executa a instrução de leitura da amostra com um período fixo (T_A), previamente conhecido quando o programa inicia a sua execução. O período pode ser estabelecido em tempo de codificação ou em tempo de execução como um dado informado pelo usuário. Adicionalmente, também se pode assumir que o período T_A é informado ao SO-PDS. Essas duas assunções sugerem o requisito estabelecido a seguir:

1º Requisito

O programa PDS executa a leitura da amostra com um período fixo (T_A) que é informado ao SO-PDS.

3.3 Controle da Freqüência de Amostragem

No Capítulo 2, item 2.2.3, o método para controle do período de amostragem *Pull-In* foi apresentado o qual, como foi visto, requer um mecanismo de tempo do sistema. Essa necessidade de uma fonte de tempo para controlar o período de amostragem, levanta no Linux três limitações:

- Os temporizadores por software (*timers*) do Linux 2.6.25 são desenvolvidos com base em *deferrable functions*. No Linux há um intervalo de tempo entre a execução deste tipo de função e sua ativação, portanto o núcleo não tem como garantir que um temporizador por software seja executado imediatamente após a expiração do tempo. Por esta razão esse tipo de temporizador não é apropriado para aplicações PDS e aplicações TR. (24)

- Além do problema da latência, até o Linux 2.6.16, esses temporizadores por software não possuíam a resolução necessária para aplicações PDS e aplicações TR, o que acarretou o desenvolvimento de temporizadores de alta resolução (*high resolution timers*). Ver item 3.10.3.

- O Linux não configura os mecanismos de tempo da arquitetura x86 com resolução suficiente para uma aplicação PDS. Por exemplo, a IRQ0[1] pode ser configurada com uma freqüência máxima de 1.535,626770 Hz. (24)

Estas limitações por serem bem conhecidas são tratadas na literatura. (24), (25), (26)

As duas formas de controlar a freqüência de amostragem analisadas no Capítulo 2 (item 2.2.3) levantam um requisito: é desejável que um SO-PDS seja compatível com ambas as formas de controle.

2º Requisito

a) O SO-PDS deve suportar as duas formas de controle da freqüência de amostragem: *Push-In* e *Pull-In*.
b) A forma de controle é informada ao SO-PDS pelo programa PDS.
c) No caso da forma de controle *Pull-In*, o SO-PDS deve fornecer um mecanismo de tempo para controle do período de amostragem.

[1] A IRQ0 também conhecida como interrupção do relógio é tratada, com mais detalhes, no Capítulo 5, item 5.2.2.

3.4 Interrupção E/S Conhecida (IRQa) e Não-Compartilhada

Neste item são analisadas como as duas formas de controle da freqüência, apresentadas no Capítulo 2, item 2.2.3, podem sinalizar o instante de amostragem. Na seqüência é apresentado o 3º requisito.

No método *Push-In* a placa adaptadora, por ser um dispositivo externo, sinaliza o instante da amostragem ao microprocessador por meio de uma interrupção E/S.

No método *Pull-In*, como é o sistema operacional que controla a freqüência de amostragem, existem duas possibilidades para implementar esse controle: temporizadores de software (*timers*) ou temporizadores de hardware (relógios ou contadores da arquitetura).

Os temporizadores de software de um sistema operacional podem usar, além da interrupção E/S, outras formas de sinalização para indicar o instante de amostragem, como, por exemplo, uma exceção. No entanto, neste trabalho não são considerados os temporizadores de software como forma de controlar o instante de amostragem, pelos motivos expostos a seguir:

1- Foi mostrado que esse tipo de temporizador, por ser desenvolvido com base em *deferrable functions*, não é apropriado para controlar processos PDS (ver item 3.3).

2- É vantajoso, do ponto de desenvolvimento, que o SO-PDS trate as duas formas de controle da freqüência de amostragem da mesma forma. Logo é conveniente que o método *Pull-In* também sinalize o instante de amostragem por uma interrupção E/S.

Portanto, neste estudo, assume-se que o método *Pull-In* usa um mecanismo de tempo da arquitetura (relógios ou contadores) que sinaliza a CPU por meio de uma interrupção E/S. (24)

Do exposto, assume-se que em um SO-PDS a sinalização do instante de amostragem é feita através de uma interrupção E/S a qual, neste texto, é chamada de IRQa.

A continuação é imposta uma restrição adicional: a IRQa é previamente conhecida e não-compartilhada.

A primeira parte desta restrição pode ser entendida da seguinte forma: assume-se que o número da IRQ usada para sinalizar o instante de amostragem é informado ao SO-PDS, seja em tempo de codificação do programa PDS, seja em tempo de execução

Agora é analisada a segunda parte. O hardware atual permite que um sinal de interrupção (IRQ) seja compartilhado por mais de um dispositivo externo. (24), (27), (28) Uma das formas do Linux 2.6.25 atender este recurso de hardware é o mecanismo de alocação dinâmica de interrupções. (24) Pensando em termos de SO-PDS, não é interessante que a IRQa seja compartilhada, já que isto implica que a interrupção tenha que ser tratada até que se possa identificar se é ou não um sinal de amostragem. No método *Pull-In* são usados mecanismos de tempo da arquitetura que normalmente não compartilham interrupções. (24) No método *Push-In* é necessário que a placa adaptadora tenha um sinal de interrupção não-compartilhado. Na prática, esta restrição para a placa adaptadora não tem maior impacto no ambiente x86 com Linux, pois:

- No desenvolvimento de aplicações típicas de PDS tem-se algum domínio sobre o hardware da máquina que pode ser configurado pelo *Basic Input Output System* (BIOS). (27), (28)

- Em máquinas compatíveis com o IBM-PC algumas interrupções são alocadas de forma estática. (24), (27), (28)

- Mesmo no caso em que a interrupção é alocada de forma dinâmica pelo Linux, o *driver* do dispositivo especifica o número da interrupção e se a interrupção E/S pode ser ou não compartilhada. (24)

É importante ressaltar que esta restrição não implica que apenas um sinal é amostrado, já que a placa adaptadora pode ter mais de uma entrada e, neste trabalho, o termo amostra também se refere a um bloco de dados. A restrição imposta implica sim, que os vários sinais chegando à placa adaptadora são amostrados de forma sincronizada com o mesmo período.

A identificação da IRQa e o seu não compartilhamento abre a possibilidade ao sistema operacional de implementar um mecanismo de prioridade no tratador de interrupções. Dar diferentes prioridades às interrupções é um mecanismo também usado em sistemas operacionais de tempo real.

3º Requisito

a) O instante de amostragem é sinalizado por uma interrupção de E/S denominada de IRQa.
b) A IRQa não é compartilhada por outros dispositivo de hardware.
c) O programa PDS informa a IRQa (número da interrupção) ao SO-PDS.

3.5 O Processo PDS em um Sistema Operacional Multiprocessos

Neste item o modelo de operação de um SO-PDS é visto com mais detalhes do que quando foi apresentado no Capítulo 2, item 2.5. Novamente será usado o modelo simplificado visto no Capítulo 2, item 2.5.1, onde se supõe que existem apenas dois processos:

- *Processo PDS* - Processo que executa o programa PDS, portanto, faz a leitura das amostras com um período de tempo fixo (T_A). Este processo por ter restrição de tempo é considerado um processo de tempo real.

- *Processo NPDS* – É um processo que não tem requisitos de tempo. Este processo é uma simplificação que engloba todos os processos, inclusive os processos do sistema (*daemons*), que estão compartilhando a CPU com o processo PDS.

A figura 3.1 mostra de forma gráfica o modelo de operação. Como pode ser visto na figura, em t_0 chega o sinal de amostragem representado por IRQa. Neste instante assume-se que o processo NPDS deve estar no estado em execução[2] e o processo PDS deve estar no estado bloqueado.

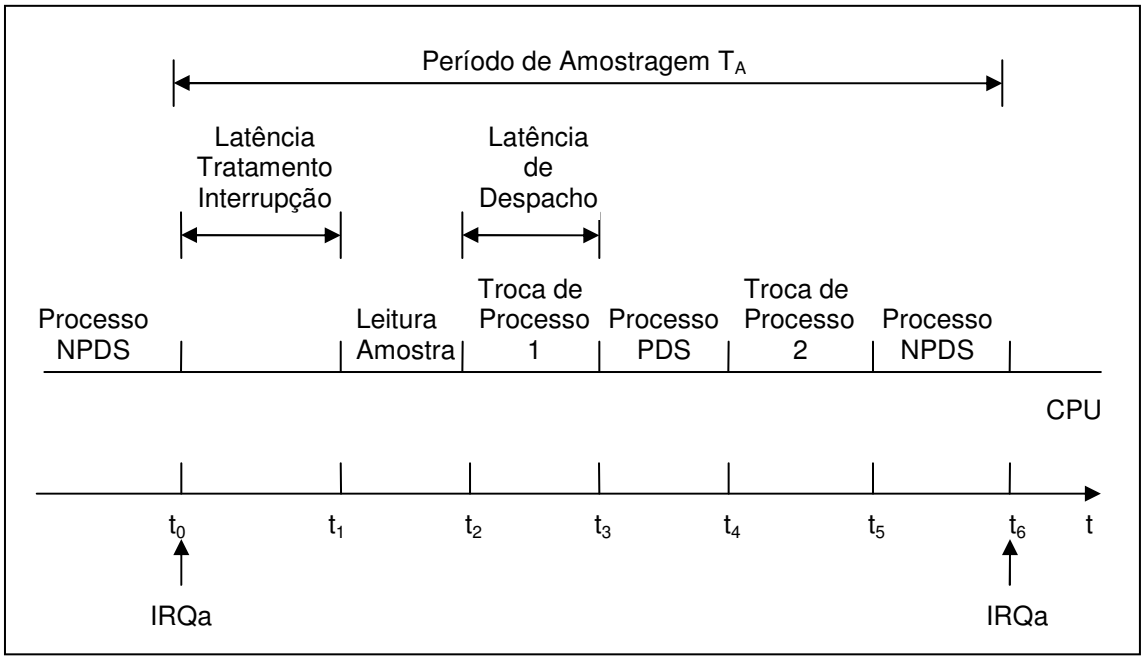

Figura 3.1 – Modelo de operação de um SO-PDS

Entre os instantes t_0 e t_1, o núcleo executa algumas instruções antes de acessar o dispositivo externo, o que acarreta um tempo de latência chamado de latência de tratamento de interrupção.

Entre os instantes t_1 e t_2, o sistema operacional efetua a leitura da amostra do dispositivo externo[3] (ex: placa adaptadora) e coloca o processo PDS no estado pronto.

O passo seguinte é colocar o processo PDS no estado em execução. Para isso, o sistema operacional deve executar o escalonador e efetuar a troca de

[2] Neste texto é usada a nomenclatura para os estados do processo estabelecida em (18), os quais estão descritos no Apêndice A, item A.4.

[3] No caso da placa adaptadora suportar *Direct Memory Access* (DMA) o núcleo não precisa ler os dados de uma porta de E/S. Uma explicação sobre DMA pode ser encontrada na referência (14).

processo, o que acarreta um tempo de latência (entre t_2 e t_3) denominado de latência de despacho (*Dispatch Latency*).

No instante t_3, o processo PDS começa o tratamento da amostra que termina em t_4. Como o processo PDS deve esperar o novo sinal de amostragem (em t_6), o sistema operacional coloca o processo PDS no estado bloqueado e executa a troca de processo para passar o processo NPDS ao estado de em execução (intervalo de tempo entre t_4 e t_5).

O modelo proposto assumiu duas condições que, a seguir, são analisadas.

Foi assumido que o processo NPDS estava no estado em execução e o processo PDS no estado bloqueado quando da chegada do sinal de amostragem. Se esta hipótese não for cumprida, o sistema está subamostrando[4], ou seja, está amostrando o sinal de entrada com uma freqüência menor do que a especificada para a aplicação PDS, o que pode ser explicado como segue.

Se o processo PDS estiver no estado em execução pode-se concluir que o processamento de uma amostra anterior não terminou quando da chegada do sinal de amostragem. Se o processo PDS estiver no estado pronto pode-se concluir que o sinal de amostragem chegou no intervalo entre t_2 e t_3 (ver na figura 3.1, Troca de Processo 1), antes que fosse concluída a troca de processo e, portanto, antes que fosse iniciado o processamento da amostra anterior. Em ambos os casos, um tempo de processamento maior que o período de amostragem é necessário para tratar uma amostra, o que exige uma diminuição da freqüência de amostragem. Normalmente para uma aplicação PDS, essa diminuição da freqüência de amostragem implica em falha do sistema.

É possível utilizar simuladores de escalonamento para determinar se a freqüência de amostragem do processo PDS pode ser atendida em uma determinada plataforma e para uma determinada carga máxima de processamento. (29), (30), (31), (32). No entanto, como foi explicado no Capítulo 2, item 2.3.5, neste

[4] O tema subamostragem é tratado no Apêndice D.

estudo, essas ferramentas de simulação não são consideradas, já que um dos focos do SO-PDS são as aplicações PDS em tempo real para o usuário final. Nesse tipo de aplicativo, a utilização desse tipo de ferramental, por não ser voltado a usuários finais, não é muito viável na prática.

Portanto, um requisito desejável é que o SO-PDS consiga detectar que o processo PDS está subamostrando.

4º Requisito

O SO-PDS deve detectar subamostragem.

No modelo proposto também foi assumida implicitamente a restrição que o programa PDS é executado principalmente como um processo em Modo Usuário[5,6]. Este tipo de processo é chamado neste texto de processo de usuário.

5º Requisito

O programa PDS é executado como um processo de usuário.

A restrição acima leva à seguinte questão: com o recurso dos módulos[7] no Linux, não seria mais conveniente fazer o programa PDS como parte do núcleo do sistema operacional, eliminando a latência de despacho? Esta questão será tratada no item 3.11.

3.6 Baixas Latências

É importante que um SO-PDS tenha valores baixos para a latência de tratamento de interrupção e para a latência de despacho, se comparados ao período de amostragem. Estes mesmos requisitos também são importantes em um SO-TR, o que levou ao surgimento de um grande número de estudos com o objetivo de reduzir

[5] O conceito de Modo Usuário é apresentado no Apêndice A, item A.2.
[6] Observa-se que o processo pode eventualmente executar chamadas ao sistema e, conseqüentemente, passar ao Modo Supervisor.
[7] O conceito de módulo no Linux é apresentado no Apêndice A, item A.9.

ambas as latências. (7), (33), (34), (35), (36), (37), (38), (39), (40), (41), (42), (43), (44), (45)

Uma técnica para obter latência de tratamento de interrupção baixa é estabelecer um mecanismo de prioridade nas interrupções. Em alguns sistemas operacionais de tempo real uma interrupção de maior prioridade é atendida mesmo que uma de menor já esteja sendo tratada. (36) Em uma versão de SO-TR baseado no Linux foram usadas *threads* do núcleo[8] para estabelecer prioridades no tratamento das interrupções. (41)

Normalmente a latência de despacho é dividida em duas partes. A primeira parte chamada de latência de escalonamento é o tempo gasto pelo algoritmo de escalonamento para selecionar um processo. A segunda parte é o tempo gasto pelo sistema operacional para realizar a troca das estruturas dos processos, operação esta chamada de troca de contexto ou troca de processo na literatura. (25), (46) De forma geral se assume que a troca de contexto é feita de forma ótima, sendo assim, as pesquisas se concentram em aprimorar os algoritmos de escalonamento. (8), (42), (43), (44), (45)

Outro fator que influi na latência de despacho é se o núcleo pode, ou não, sofrer preempção[9]. (24), (41), (47) Nos núcleos sem preempção, um processo em Modo Supervisor[10] não perde a posse da CPU, ou seja, a rotina do núcleo que o processo está executando é sempre finalizada. Em núcleo com preempção esta limitação não existe, ou seja, o processo pode perder a posse da CPU mesmo que esteja executando uma rotina do núcleo. Portanto, em termos de SO-PDS, um núcleo sem preempção faz com que o processo PDS deva esperar que o processo NPDS volte a Modo Usuário antes de assumir a posse da CPU, o que implica em uma latência de despacho maior do que em um núcleo com preempção.

Com o intuito de obter baixa latência de despacho, o núcleo do Linux a partir da versão 2.6.1 pode de modo geral sofrer preempção quando está executando

[8] O conceito de *thread* do núcleo (*kernel thread*) é apresentado no Apêndice A, item A.5.
[9] O conceito de preempção do núcleo é apresentado no Apêndice A, item A.8.
[10] O conceito de Modo Supervisor é apresentado no Apêndice A, item A.2.

várias funções. No entanto, até a versão 2.6.25 há trechos do núcleo que quando em execução bloqueiam a preempção. Isto é possível através de um mecanismo do núcleo, que permite habilitar e desabilitar a preempção do núcleo. Existem rotinas do núcleo, como o escalonador, que desabilitam a preempção ao iniciar uma atualização nas estruturas de dados do núcleo e, que habilitam a preempção ao terminar a atualização. Ver referências (24), (48).

Portanto, do exposto, pode-se estabelecer o seguinte requisito:

6º Requisito

O SO-PDS deve apresentar baixas latências de tratamento de interrupção e despacho se comparadas com o período de amostragem.

Para finalizar este item, o 6º Requisito é analisado com relação a um tema encontrado na literatura de tempo real: o conceito de que tempo real não é rapidez. (14), (49)

Esse conceito pode ser explicado da seguinte forma: se uma determinada aplicação PDS em tempo real tem a sua freqüência de amostragem atendida para uma latência ΔLat e uma determinada carga máxima do sistema, não há vantagem, em termos de tempo real, em diminuir a latência ΔLat, já que essa aplicação específica, em não havendo mudança na carga máxima do sistema, não se beneficiaria dessa diminuição. Esse raciocínio levanta a seguinte questão: qual a finalidade de buscar latências mínimas em um SO-PDS? Esta pergunta é respondida da seguinte forma:

1- As baixas latências tornam o SO-PDS eficiente na execução do aplicativo PDS. A eficiência em um SO-PDS é uma característica desejável, como foi visto no Capítulo 1, item 1.4 e no Capítulo 2, item 2.3.5.

2- Quanto menor latência em um SO-PDS, maior é a freqüência de amostragem do aplicativo PDS que o sistema operacional pode

executar. Isso faz com que mais aplicativos PDS possam ser executados em um mesmo hardware, o que é uma vantagem do ponto de vista do usuário final.

3- Partindo da hipótese que a latência seja uma variável aleatória, em função de variações na carga do sistema, tem-se que quanto menor a média da latência, menor é a probabilidade de subamostragem de um processo PDS em tempo real.

` Desta forma, procurou-se justificar o 6º Requisito em relação ao conceito de que tempo real não é rapidez.

3.7 O Processo PDS é Único

Como a idéia básica é melhorar o desempenho do processamento digital de sinais no ambiente x86 com Linux, pode-se supor que só exista um processo PDS no sistema.

A justificativa desta restrição é que a existência de dois processos PDS faria com que em certos instantes de amostragem, múltiplos de ambos os períodos de amostragem, houvesse uma condição de corrida pela CPU, aumentando a latência para iniciar a execução de um deles. Como se parte da hipótese que não há controle sobre a carga máxima do sistema e nem simulações de escalonamento (ver item 3.5), esse aumento da latência pode fazer com que um dos processos PDS subamostre.

É importante ressaltar que, na prática, esta restrição atende boa parte das aplicações em processamento digital de sinais em tempo real. Em aplicações corporativas normalmente tem-se um aplicativo PDS em tempo real sendo executado no microcomputador, como no exemplo visto no Capítulo 2, item 2.3.2. Mesmo em aplicações voltadas para o usuário final não é comum o cenário de duas aplicações PDS sendo executadas simultaneamente.

Do ponto de vista de desenvolvimento, não há grande dificuldades para garantir que o processo PDS seja único, o que pode ser feito, por exemplo, com um mecanismo de trava (*lock*) ao acesso de uma estrutura de dados.

Esta restrição traz duas vantagens:

1- Estabelece um mecanismo de prioridade[11] dos processos no SO-PDS.
2- Simplifica o algoritmo de escalonamento.

Essa restrição pode ser colocada como mostrado a seguir:

7º Requisito

a) Um SO-PDS deve estabelecer a diferenciação entre o processo que executa o programa PDS (processo PDS) e os outros que não têm restrição de tempo (processos NPDS).
b) Um SO-PDS deve garantir que o processo PDS seja único.

3.8 Estados do SO-PDS e Forma de Operação

Neste item, são definidos dois estados de operação para o SO-PDS:

- Normal – Neste estado não um há processo PDS no sistema e, portanto, o funcionamento do SO-PDS se aproxima ao de um sistema operacional de uso geral.

- PDS – Neste estado há um processo PDS no sistema, que pode estar no estado em execução ou no estado bloqueado.

Esta diferenciação no estado do sistema operacional traz algumas vantagens no desenvolvimento do núcleo de um SO-PDS quando, por exemplo, são

[11] A prioridade do processo é usada pelo escalonador para selecionar o processo que será colocado no estado de em execução. A prioridade do processo no Linux é tratada no Capítulo 5, item 5.6.

usadas técnicas adaptativas[12]. Como este requisito é uma decisão de projeto, já que visa facilitar o desenvolvimento, pode ser considerado um requisito desejável, mas não obrigatório.

A forma de operação mais direta é que o processo solicite ao SO-PDS a permissão para passar ao status de processo PDS. Isto pode ser feito na Rotina de Início (ver Capítulo 2, figura 2.1), onde o programa, por exemplo, através de uma chamada ao sistema faz a solicitação. Informações como período de amostragem e número da interrupção a ser usada para o controle da amostragem podem ser passadas como parâmetros da chamada ao sistema. Se SO-PDS já está no estado PDS a requisição é rejeitada (haja vista que só pode haver um processo PDS), caso contrário o sistema operacional aceita a requisição e passa do estado Normal para PDS. Quando o processo PDS é finalizado, seja de forma normal ou pelo operador do sistema, o sistema operacional muda o estado de PDS para Normal.

8º Requisito (Não Obrigatório)

Um SO-PDS tem pelo menos dois estados de operação:

a) Normal quando não há processo PDS no sistema.
b) PDS quando há um processo PDS no sistema, sendo que pode estar no estado em execução ou no estado bloqueado.

3.9 Preempção de Processo Seletiva

Em sistemas multiprocessos, geralmente um processo possui um intervalo de tempo máximo para ficar no estado em execução, ou seja, de posse da CPU. Esgotado esse intervalo de tempo, o núcleo através do escalonador retira o processo do estado de em execução e passa a posse da CPU a outro processo. (18), (24)

[12] Técnica adaptativa é uma técnica que durante uma primeira fase, chamada de treinamento, estima parâmetros do sistema operacional como forma de melhorar o desempenho durante uma segunda fase, chamada de operação. Exemplos de uso desta técnica serão vistos no Capítulo 7.

É um requisito que o processo PDS não tenha restrição de tempo para ficar no estado de em execução, ou seja, o processo PDS só perde a posse da CPU quando ele a libera ao finalizar o processamento da amostra e, conseqüentemente, passa ao estado de bloqueado, haja vista que fica à espera do próximo instante de amostragem.

9º Requisito

Em um SO-PDS, o processo PDS não tem restrição de tempo para ficar no estado de em execução.

3.10 Objetivo dos Requisitos

Neste estudo, procura-se desenvolver um SO-PDS com base no Linux que não se afaste muito do núcleo do Linux original. Considerando esta diretiva de projeto, este item tem dois objetivos principais: fazer uma introdução às soluções desenvolvidas para o Linux atender os requisitos de tempo real e, mostrar, de forma introdutória, como as restrições impostas para o SO-PDS permitem o uso de mecanismos que não são normalmente usados em sistemas operacionais de tempo real.

Uma abordagem normalmente usada para desenvolver um sistema operacional de tempo real com base no Linux é o mecanismo de dois núcleos: um núcleo de tempo real e o núcleo do Linux. Nesta arquitetura, o núcleo de tempo real basicamente: 1) controla as interrupções, 2) executa as aplicações de tempo real e 3) executa o núcleo do Linux como um processo de baixa prioridade. Como o núcleo de tempo real situa-se entre o hardware e o núcleo do Linux, esta abordagem é semelhante ao Linux sendo executado em uma máquina virtual. Exemplos desse tipo de implementação são os sistemas operacionais RTLinux e o RTAI. Esta arquitetura tem a desvantagem que os programas de tempo real são executados pelo núcleo de tempo real, o qual normalmente não é totalmente compatível com o núcleo do Linux, o que pode dificultar a portabilidade de programas ou sistemas mais complexos. Como esses sistemas se afastam da linha principal do Linux (*main*

line), neste trabalho esta arquitetura não foi considerada para projetar o SO-PDS. (50), (51), (52)

Em relação a outras soluções atualmente desenvolvidas com base no Linux, este item foca apenas nos trabalhos realizados pelo laboratório *Open Source Automation Development Lab* (OSADL). (53) Esta escolha deve-se ao fato de que as soluções desenvolvidas por este laboratório são próximas à linha principal[13] do Linux mantida por Linus Torvalds, o que fez com que algumas dessas soluções fossem, com o tempo, adotadas na linha principal. (48) O desenvolvimento da OSADL para o Linux em tempo real foca, principalmente, nas seguintes áreas:

- Escalonador determinístico.
- Tratamento das interrupções.
- Temporizadores de alta resolução.
- Núcleo com preempção.

3.10.1 Escalonador Determinístico

O escalonador determinístico é um algoritmo que seleciona o processo a ser executado pela CPU em um mesmo número de instruções independentemente do número de processos no sistema. (24)

O Linux 2.6.25 utiliza um escalonador determinístico que além de separar os processos de tempo real daqueles que não têm restrição de tempo, permite dar prioridades diferentes aos processos de tempo real. (24), (48) O escalonador será tratado no Capítulo 5, item 5.6.

É importante observar que um escalonador determinístico não implica necessariamente em baixa latência, haja vista que esse tipo de algoritmo requer a atualização de estruturas de dados.

[13] Neste texto será usado o termo linha principal para o núcleo do Linux mantido por Linus Torvalds, também chamado de *mainline* ou *vanilla*.

Neste trabalho a idéia é usar a unicidade do processo PDS (7º Requisito) para simplificar o algoritmo do escalonamento, uma vez que do ponto de vista teórico o escalonamento do processo PDS se reduz a verificar se o mesmo está no estado pronto, obtendo-se assim uma latência de escalonamento baixa.

3.10.2 Tratamento das Interrupções

Em relação ao tratamento das interrupções[14] em sistemas de tempo real, dois são os objetivos principais: manter as interrupções desabilitadas no microprocessador o menor tempo possível e, como já foi colocado, estabelecer prioridades às interrupções no sistema operacional.

No Linux o primeiro objetivo é alcançado permitido que as ações necessárias para o tratamento de uma interrupção sejam divididas em três classes: 1) críticas, 2) não-críticas e 3) não-críticas e postergáveis. Nesta classificação apenas as instruções críticas são realizadas com as interrupções desabilitadas.

O segundo objetivo visa permitir que um tratador de interrupções sofra preempção quando uma interrupção de maior prioridade é sinalizada pelo microprocessador. Este recurso não está implementado no Linux 2.6.25. (24), (48) A solução pesquisada pela OSADL se baseia na execução do tratador de interrupções como uma *thread* do núcleo. (48)

Neste trabalho, este problema é abordado usando os seguintes requisitos:

1- Periodicidade da leitura das amostras com período T_A conhecido do SO-PDS (1º Requisito).

2- A sinalização de amostragem é feita por uma interrupção E/S (IRQa) não-compartilhada e conhecida do SO-PDS (3º Requisito).

[14] Uma introdução às interrupções é apresentada no Apêndice B e o Apêndice C trata das interrupções no Linux.

Como é mostrado no Capítulo 7, esses dois requisitos são usados para desenvolver um tratador de interrupções com uma latência de tratamento de interrupção baixa para a IRQa.

3.10.3 Temporizadores de Alta Resolução

No Linux a implementação de temporizadores de alta resolução consiste no desenvolvimento de uma *Application Programming Interface* (API) estabelecida pela norma *Portable Operating Systems Interface for Unix* (POSIX) 1003.1b Seção 14 (Relógios e Temporizadores). (54)

No Linux esta API foi incluída a partir da versão 2.6.16, o que permite uma resolução maior para o temporizadores que a resolução dada pela IRQ0, a qual para a arquitetura x86 é de aproximadamente 1 ms. (55)

No Linux esses temporizadores são implementados usando-se *deferrable functions*, o que implica nas limitações de precisão já tratadas no item 3.3. Como colocado no item 3.4, neste trabalho assume-se a restrição que o controle do período de amostragem é realizado por hardware.

3.10.4 Núcleo com Preempção

O desenvolvimento de um núcleo com preempção é feito basicamente diminuindo as regiões críticas do núcleo, ou seja, partes do núcleo que devem ser executadas de forma atômica. No Linux os trabalhos focaram na diminuição do tempo com as interrupções desabilitadas e em mudanças nos mecanismo de sincronização. (24), (41), (48)

Embora avanços tenham sido realizados, o núcleo do Linux 2.6.25 ainda possui trechos de código que não podem sofrer preempção quando em execução. (24) Neste trabalho esta limitação é contornada fazendo restrições no aplicativo PDS, como é mostrado no Capítulo 7.

3.11 Processo de Usuário x Módulo em Modo Supervisor

A seguir são apresentadas as principais razões para executar o programa PDS como um processo de usuário e não como um módulo em Modo Supervisor com base nas referências (24), (56).

- O módulo é executado como um fluxo de controle do núcleo[15] (*kernel control path*) e, portanto, não pode executar funções que levem a um bloqueio do processo à custa do qual o módulo está sendo executado. (24), (56)

- Em certos casos, erros de programação em módulos podem levar ao travamento (*freeze*) de todo o sistema. (24)

- No Linux, o tamanho da pilha do núcleo pode ser configurado por parâmetro de compilação em um mínimo de 4096 bytes. Este tamanho pode ser pequeno para conter toda a cadeia das funções executadas no contexto do núcleo e das funções necessárias para o tratamento da amostra. (24)

- O núcleo pode efetuar aritmética de ponto flutuante de forma limitada. Na arquitetura x86 o núcleo deve salvar o estado da unidade de ponto flutuante (*floating point unit*) na pilha do núcleo. Como no Linux a pilha do núcleo forma com a pilha do processo uma única estrutura, o núcleo deve garantir que não haja preempção do processo corrente até que o uso da unidade de ponto flutuante tenha terminado. (24), (56)

- O módulo é lincado apenas no núcleo e, portanto, só as funções exportadas pelo núcleo podem ser usadas. Não há bibliotecas para serem lincadas. Isto implica que módulo teria de incluir além do

[15] O conceito de fluxo de controle do núcleo (*kernel control path*) é apresentado no Apêndice A, item A.3.

programa todas as funções necessárias para o tratamento da amostra. (24), (56)

- Ao usar as estruturas de dados e as funções do núcleo, o módulo é compilado para uma versão específica do núcleo já que essas interfaces podem mudar entre versões do mesmo. Existem algumas técnicas para desenvolver módulos para múltiplas versões de núcleo. (56)

- A compilação de um módulo é bem mais complexa do que a de um programa de usuário. São necessárias versões compatíveis de compiladores, utilitário dos módulos e outros utilitários, além de uma árvore do núcleo. Tentar compilar o núcleo e seus módulos com a versão errada dos utilitários pode ocasionar problemas. É importante notar que mesmo versões mais novas podem ocasionalmente levar a erros assim como versões mais antigas. (56)

- Quando o módulo termina, os recursos usados não são liberados pelo sistema operacional, devem ser liberados via programação no próprio módulo, caso contrário os recursos ficam presos até a próxima carga do sistema. (56)

- As ferramentas para depurar um módulo são mais limitadas do que as ferramentas para depurar programas que são executados como processos de usuário. (56)

Ao escolher que o programa PDS seja executado como um processo de usuário, está-se trocando tempo de latência por facilidade no desenvolvimento e segurança na execução.

3.12 Requisitos Não-Funcionais

No decorrer do capítulo foram apresentados requisitos funcionais[16] para um SO-PDS, neste item são tratados, de forma sucinta, os requisitos não-funcionais. A

[16] Os conceitos de requisitos funcionais e não-funcionais são apresentados no Apêndice E, item E.4.

referência (14) apresenta os requisitos não-funcionais divididos em 5 (cinco) categorias: interfaces externas, desempenho, banco de dados lógico, restrições de design e atributos do software. Neste item são tratados apenas parâmetros para medida de desempenho de um SO-PDS.

A primeira pergunta que se apresenta é: que grandezas podem ser usadas como medida de desempenho para um SO-PDS? Pelas características do sistema estudado, pode-se pensar em usar a latência e o número de falhas por unidade de tempo como formas de medir o desempenho do sistema.

A latência normalmente é caracterizada por dois parâmetros: a média e a variância. De forma geral pode-se assumir que um sistema com média e variância menores para a latência tem melhor desempenho do que um outro com valores maiores para ambos os parâmetros. Como será visto no Capítulo 6, uma limitação desses parâmetros é que eles variam em função da carga do sistema operacional e, portanto, caracterizam o desempenho do SO-PDS para uma condição particular.

Como foi visto no item 3.5, para um SO-PDS a subamostragem representa uma falha no sistema e, portanto, o número de vezes que o sistema subamostra durante um intervalo de tempo pode ser considerando como um parâmetro para medição do desempenho[17]. A ocorrência de subamostragem depende da variação na latência e no tempo de processamento da amostra (ver Apêndice D). Isto implica que este parâmetro varia em função da carga do sistema operacional e da aplicação PDS executada e, portanto, também caracteriza o desempenho do SO-PDS para uma condição particular, o que representa uma limitação.

Do exposto neste item, pode-se observar que o estabelecimento de requisitos não-funcionais de desempenho para um SO-PDS vai depender de condições específicas de carga do sistema operacional e da aplicação PDS executada.

[17] O tempo para recuperação da falha, ou seja, o tempo em que o sistema fica subamostrando também é um parâmetro que pode ser considerado, porém não é tratado no texto.

3.13 Resumo

Neste capítulo os principais requisitos para um sistema operacional PDS foram estabelecidos e também foram levantadas algumas limitações do Linux 2.6.25 para aplicações de tempo real. O próximo capítulo será dedicado a um estudo mais detalhado das latências do Linux 2.6.25 na arquitetura x86, as quais foram tratadas brevemente no item 3.5.

Capítulo 4 - Modelo de Latência no Linux e Requisitos Adicionais para o SO-PDS

4.1 Introdução

No capítulo anterior foi apresentado o problema da latência ao passar um processo PDS do estado bloqueado para o estado em execução. Neste capítulo, este mesmo problema é analisado mais detalhadamente. São identificadas e analisadas todas as latências que ocorrem desde que uma interrupção de amostragem chega ao sistema até que termina o tratamento da amostra de entrada. A partir deste estudo requisitos adicionais são estabelecidos para SO-PDS.

O microprocessador da família x86 possui dois tipos de interrupção[1]: mascarável e não mascarável. A interrupção não mascarável é acionada através do pino `NMI` do microprocessador e não é usada pelos periféricos dos microcomputadores do tipo PC, já que estes sinalizam o microprocessador através das interrupções mascaráveis. Portanto, o estudo deste capítulo está focado nas interrupções mascaráveis, também chamadas de interrupções E/S.

O material aqui desenvolvido parte da hipótese que o instante da amostragem é sinalizado por uma interrupção E/S externa. O caso em que o instante da amostragem é sinalizado por uma interrupção E/S local é descrito no Apêndice F.

4.2 Modelo de Latência

Como foi visto no Capítulo 2 e no Capítulo 3, o processo PDS realiza as seguintes operações a cada período de amostragem: lê uma amostra de entrada, trata a amostra e gera um dado de saída. Decorre desta característica do processo que se pode dividir a latência total em duas partes: uma latência de leitura e uma latência de execução.

[1] Ver no Apêndice B, item B.3, a classificação completa usada neste texto para as interrupções.

A latência de leitura é o tempo decorrido entre a chegada do sinal de amostragem e a primeira instrução a ser executada pelo processo PDS para tratar a amostra, descontando-se o tempo de leitura da amostra do dispositivo externo. No Capítulo 3, item 3.5, a latência de leitura foi dividida em duas partes: uma latência de tratamento de interrupção e uma latência de despacho. A latência de tratamento de interrupção é o tempo entre a chegada do sinal de interrupção (IRQa) e a primeira instrução da rotina que efetua a leitura da amostra do dispositivo externo. Por sua vez a latência de despacho é o tempo decorrido entre a instrução que coloca o processo PDS no estado de pronto e a primeira instrução do processo PDS que realiza o tratamento da amostra.

A latência de execução mede o tempo perdido pelo processo PDS com funções do núcleo que não estão relacionadas ao tratamento da amostra como, por exemplo, o tratamento de interrupções. A latência de execução pode ser definida com sendo a diferença entre o tempo gasto pelo processo PDS para tratar a amostra e o tempo que gastaria se o tratamento da mesma fosse atômico (sem interrupções).

A figura 4.1 mostra um modelo de latência mais completo do que o apresentado no Capítulo 3.

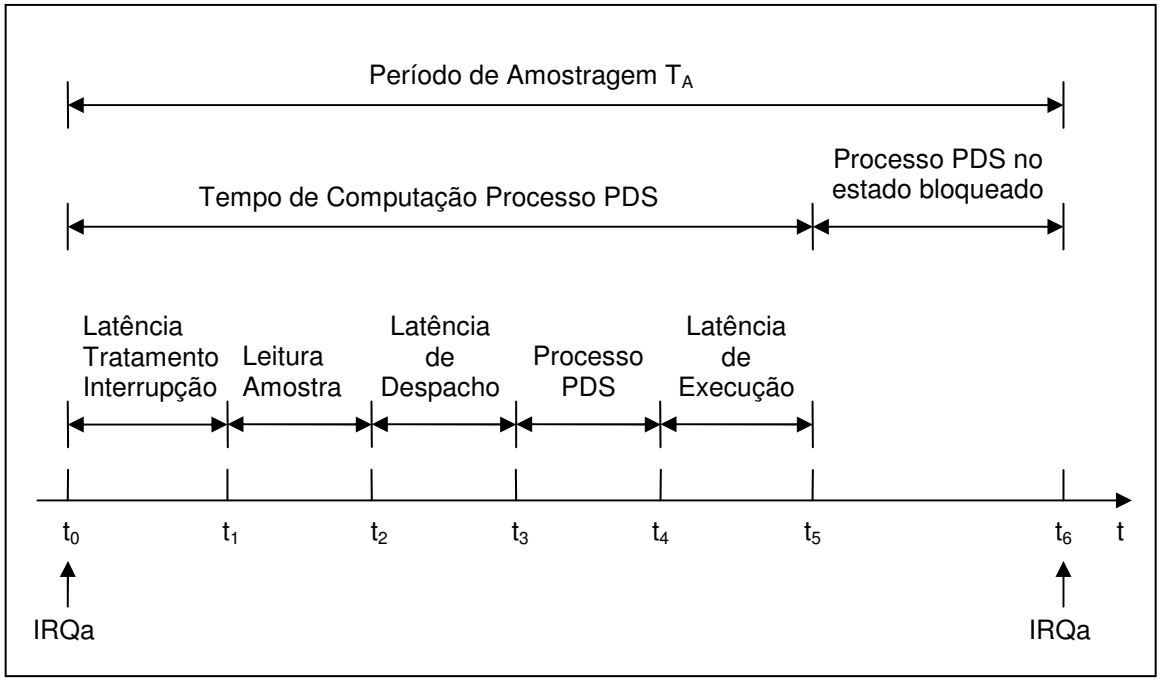

Figura 4.1 – Modelo para a latência de um processo PDS

Ressalta-se a simplificação na figura 4.1, já que a latência de execução em realidade é uma somatória de tempo perdido durante a execução do processo PDS em um período.

4.3 Latência de Tratamento de Interrupção

Neste item são estudadas todas as fontes que contribuem para formar a latência de tratamento de interrupção, onde a interrupção a ser tratada é por hipótese uma interrupção E/S externa. A figura 4.2 mostra as componentes da latência de tratamento de interrupção, as quais são definidas no decorrer deste item.

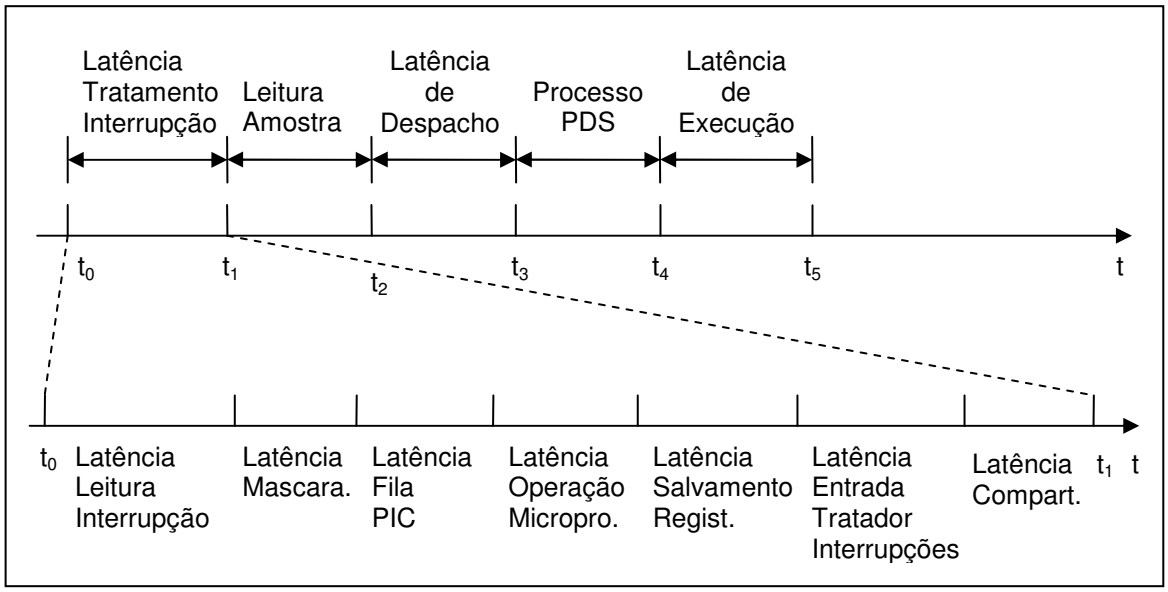

Figura 4.2 – Componentes da latência de tratamento de interrupção

4.3.1 Latência da Leitura da Interrupção

De forma geral os microprocessadores e em particular os da família x86 verificam a existência de interrupções quando uma instrução é executada de forma completa. Isto implica que há um intervalo de tempo entre a sinalização da interrupção e o microprocessador detectar a interrupção. Neste estudo, este intervalo de tempo é definido como sendo a latência de leitura da interrupção.

Decorre da definição que a latência de leitura de interrupção vai depender do tempo de execução da instrução que estava sendo processada quando a interrupção foi sinalizada.

A família x86 possui uma arquitetura do tipo *Complex Instruction Set Computer* (CISC) que tem, como uma de suas características, instruções complexas com tempos diferentes de execução. Em se tratando de microprocessadores de uso geral, existe uma segunda arquitetura chamada de arquitetura R*educed Instruction Set Computer* (RISC) que se caracteriza por possuir instruções simples que são executadas em um único ciclo de relógio. Isto faz com a segunda arquitetura seja mais adequada para sistemas de tempo real rígido[2] (*hard real-time*). (14)

Além da variação com a instrução em si, nos microprocessadores da família x86 o tempo de execução de uma instrução também depende dos valores dos dados processados, o que dificulta a previsibilidade da execução que é um fator importante em sistemas de tempo real rígido. (57), (58)

Estes dois fatores de variação no tempo de execução fazem com que a latência de leitura da interrupção na família x86 tenha uma variância maior se comparada à obtida em microprocessadores DSP específicos e à obtida em microprocessadores de uso geral RISC.

4.3.2 Latência de Mascaramento

Nos microprocessadores da família x86, uma interrupção mascarável é tratada ou não dependendo do valor do bit `IF` do registrador `EFLAGS`. Se `IF`=1 a interrupção é tratada e se `IF`=0 a interrupção não é tratada.

A latência de mascaramento é definida como sendo o intervalo de tempo durante o qual se posterga o tratamento de uma interrupção devido ao microprocessador estar com o bit `IF`=0, ou seja, ignorando todas as interrupções mascaráveis que chegam ao sistema.

[2] A noção de tempo real rígido (*hard real-time*) é apresentada no Apêndice E, item E.3.

O bit IF é zerado de duas formas: pelo microprocessador ou pelo núcleo do sistema operacional. Ambos os casos são analisados neste item.

Quando o microprocessador trata uma interrupção E/S, por exemplo, uma tecla é apertada ou a placa de rede sinaliza, coloca os registradores EFLAGS, CS e EIP na pilha; coloca o bit IF=0 e aciona a rotina que irá tratar a interrupção. Isto é feito pelo microprocessador para impedir que duas interrupções E/S sejam tratadas simultaneamente. Caso o núcleo do Linux constate que o tratador da interrupção pode funcionar com as interrupções habilitadas coloca o bit IF=1, habilitando as interrupções. Caso contrário, ao término da rotina, o comando IRET retorna os valores originais salvos na pilha para os registradores, o que faz o bit IF=1 (habilitando as interrupções). Portanto quando o núcleo está tratando uma interrupção E/S existe um intervalo de tempo no qual o microprocessador está ignorando as interrupções que chegam ao sistema. Se a interrupção IRQa chegar durante esse intervalo de tempo, há uma latência para que a mesma seja tratada. Esta latência é menor quando as rotinas de tratamento das interrupções podem ser interrompidas (executadas com o bit IF=1) do que quando as mesmas não podem ser interrompidas (executadas com o bit IF = 0).

O microprocessador possui duas instruções para manipular de forma direta o bit IF, a instrução STI que faz IF = 1 e a instrução CLI que faz IF = 0[3]. O microprocessador tem um mecanismo de proteção que permite determinar quais são as formas de operação que podem manipular o bit IF. O Linux configura esse mecanismo de tal forma que apenas processos em Modo Supervisor possam alterar o bit IF, ou seja, somente as rotinas do núcleo habilitam ou desabilitam as interrupções. O Linux fornece funções para alterar o bit IF como, por exemplo, local_irq_enable() e local_irq_disable(). O núcleo desabilita as interrupções quando precisa executar um conjunto de instruções sem interrupção[4], como por exemplo, para configurar mecanismos de sincronização; sendo que as habilita ao término desse processamento. Se uma interrupção IRQa chegar durante

[3] Existem outras instruções que podem ser usadas para manipular o bit IF. Ver referência (59).
[4] Observa-se que em sistemas multiprocessadores, desabilitar as interrupções de uma CPU não garante a manipulação atômica de uma estrutura.

o intervalo em que as interrupções estão desabilitadas pelo núcleo, há uma latência para que a mesma seja tratada.

Em resumo, tem-se uma latência de mascaramento que decorre das interrupções estarem desabilitadas. Duas são as causas:

- O microprocessador desabilita as interrupções automaticamente quando uma interrupção é tratada.

- O núcleo desabilita as interrupções através de funções específicas e isso pode ocorrer quando, por exemplo:
  - uma exceção é tratada[5],
  - uma *deferrable function* está sendo executada.

Decorre da explanação o seguinte requisito:

10º Requisito

Em um SO-PDS, o núcleo deve manter as interrupções desabilitadas pelo menor intervalo de tempo possível.

Esta restrição limita-se ao núcleo, já que o microprocessador continua automaticamente desabilitando as interrupções quando uma é sinalizada.

4.3.3 Latência da Fila do Controlador Programável de Interrupções

Os dispositivos de hardware externos requisitam interrupções através do sinal de requisição de interrupção ou *Interrupt ReQuest* (IRQ). Todos os sinais de IRQ são ligados aos pinos de entrada de um circuito chamado Controlador Programável de Interrupções ou PIC (*Programmable Interrupt Controller*) que monitora as vias, esperando que uma seja ativada. Quando uma via é ativada, o PIC realiza as seguintes operações:

[5] Uma chamada ao sistema é considerada neste item como um caso particular de uma exceção.

1- Coloca o vetor de interrupção na porta de E/S.
2- Sinaliza o microprocessador pelo pino `INTR`.
3- Espera a sinalização do microprocessador (*ack*).
4- Libera o pino `INTR`.

Da descrição realizada, observa-se que decorre um intervalo de tempo entre a ativação de um sinal e a liberação do pino `INTR` (passo 4), a qual habilita o PIC para tratar outro sinal de interrupção. Se interrupções chegarem durante esse intervalo de tempo, o controlador as coloca em um estado pendente para tratá-las posteriormente, quando o pino `INTR` for liberado. O controlador estabelece uma prioridade para o tratamento, ou seja, dentre as interrupções que estiverem pendentes, aquela com maior prioridade é atendida primeiro.

A latência da fila do controlador programável de interrupções é o tempo em que uma interrupção fica pendente para ser tratada na fila do controlador, tempo este que depende da prioridade da interrupção. Esta definição sugere o requisito a seguir:

11º Requisito

Em um SO-PDS, a IRQa deve ser a interrupção de maior prioridade no controlador programável de interrupções.

No controlador 8259A a forma default para estabelecer a prioridade é pelo número do pino, ou seja, a IRQ0 tem prioridade sobre a IRQ1. No entanto este controlador permite, via programação, alterar a forma de como essa prioridade é estabelecida. (60)

O controlador IO-APIC (820993AA) dispõe de uma tabela de registradores de E/S mapeada em memória, permitindo que a prioridade dos sinais das interrupções não esteja relacionada com o número do pino físico. A tabela possui um registrador para cada sinal de interrupção e nesse registrador pode-se, via programação, determinar o número do vetor, que estabelece a prioridade. (24), (61)

4.3.4 Latência de Operação do Microprocessador

Quando o microprocessador recebe uma interrupção deve fazer algumas operações antes de passar o controle para o tratador de interrupções. No caso específico dos microprocessadores da família x86, as principais operações realizadas pelo microprocessador são[6]:

1- O processador salva o estado dos registradores `EFLAG`, `CS` e `EIP` na pilha.
2- Carrega os registradores `CS` e `EIP` com valores que definem o endereço lógico da primeira instrução da rotina que trata a interrupção IRQn.

O tempo gasto pelo microprocessador para executar estas operações gera uma latência que, neste texto, é chamada latência de operação do microprocessador, a qual é inerente ao hardware. Do descrito, pode-se constatar que os microprocessadores da família x86 não salvam automaticamente todos os registradores de uso comum.

4.3.5 Latência de Salvamento dos Registradores

Como os microprocessadores da família x86 não salvam automaticamente todos os registradores de uso comum (item 4.3.4), esta operação fica a cargo do sistema operacional e deve ser executada antes de tratar a interrupção.

O Linux implementa esse requisito da seguinte forma: todas as interrupções E/S externas são direcionadas para o mesmo endereço. Isso é feito colocando um único endereço em todas as entradas da *Interrupt Descriptor Table*[7] (IDT) que sejam de interrupções E/S externas. O bloco de instruções relacionado a esse endereço pode ser visto na figura 4.3.

---

[6] Uma explicação mais completa sobre como o hardware trata uma interrupção pode ser encontrada no Apêndice B, item B.6 e item B.7.

[7] Informações sobre a Tabela Descritora das Interrupções podem ser encontradas no Apêndice B, item B.6.4.

```
001    pushl $n-256
002    jmp common_interrupt
```

Figura 4.3 – Bloco de instruções correspondente ao endereço constante nas entradas da tabela IDT para interrupções E/S externas
Extraído da referência (24)

O sistema operacional salva o código da interrupção na pilha (linha 001) e salta para um bloco comum que trata todas as interrupções E/S externas (linha 002).

A seguir, na figura 4.4 tem-se as instruções realizadas pelo sistema operacional após o salto para a linha `common_interrupt`.

```
001    common_interrupt:
002            SAVE_ALL
003            TRACE_IRQS_OFF
004            movl %esp,%eax
005            call do_IRQ
006            jmp ret_from_intr
007            CFI_ENDPROC
```

Figura 4.4 – Bloco de instruções que inicia o tratamento de uma interrupção E/S externa
Extraído de `linux/arch/x86/kernel/entry_32.s`

Na linha 002 pode-se ver a macro `SAVE_ALL` que salva os registradores de uso comum e, na linha 003 a macro `TRACE_IRQS_OFF` que monitora o estado das interrupções, ou seja, se estão habilitadas ou desabilitadas. A linha 004 salva o endereço da pilha corrente no registrador `eax` e, a linha 005 inicia o tratamento da interrupção.

A execução do código da figura 4.3 e das linhas 002 a 004 da figura 4.4 implica em uma latência que, neste estudo, é chamada de latência de salvamento de registradores. Neste trabalho procura-se dividir a latência pela sua origem. Esta latência deve-se a que:

- O microprocessador não salva todos os registradores de uso comum.
- O sistema operacional é um sistema de uso geral.

No item a seguir 4.3.6, é apresentada uma latência que decorre unicamente do fato do Linux ser um sistema de uso geral.

4.3.6 Latência de Entrada no Tratador de Interrupções

Neste item são descritas de forma seqüencial todas as operações realizadas a partir da chamada da rotina `do_IRQ` até que a rotina de serviço de interrupção ou *Interrupt Service Routine*[8] (ISR) seja executada.

Como pode ser visto na figura 4.4 (linha 005), inicialmente o Linux trata todas as interrupções da mesma forma através da rotina `do_IRQ`. Esta rotina executa algumas operações que não estão relacionadas ao tratamento da interrupção em si, mas que decorrem de ser o Linux um sistema operacional de uso geral. Como este item visa identificar as latências, essas operações serão descritas de forma sucinta. O Apêndice C e a referência (24) tratam do tema com mais detalhes.

4.3.6.1 Análise da Rotina: `do_IRQ`[9]

Operações realizadas pela rotina `do_IRQ`:

1- Verificar a validade do número da interrupção. Se o número não for válido aciona a macro `BUG`.
2- Salvar novamente os registradores de uso comum em uma estrutura.
3- Executar a macro `irq_enter()` que incrementa um contador para controlar o número de interrupções aninhadas.
4- Se o tamanho da pilha do núcleo é de 4 KB e o sistema estava em Modo Usuário quando ocorreu a interrupção, o sistema troca para a chamada pilha IRQ (*stack IRQ*).
5- Chamar a função que trata a interrupção determinada. O ponteiro desta função está no descritor da interrupção, podendo mudar em função do

[8] Informações sobre a rotina de serviço de interrupção podem ser encontradas no Apêndice C, item C.2.
[9] Observa-se que `do_irq` e `__do_irq` são rotinas diferentes do núcleo do Linux 2.6.25.

hardware. A função padrão para as interrupções E/S externas na arquitetura x86 é a `__do_IRQ`, que recebe como parâmetro de entrada o número da IRQ. Ver Apêndice C, (24).

4.3.6.2 Análise da Rotina: `__do_IRQ`[9]

Operações realizadas pela rotina `__do_IRQ()`:

1- Atualizar as estatísticas de interrupção por CPU.
2- Executar um travamento (*lock*) no descritor da interrupção.
3- Executar o método `ack` do objeto `chip` que consta no descritor da interrupção. Este método sinaliza o controlador programável de interrupções e seu funcionamento depende do modelo de controlador instalado no hardware.
4- Inicializar alguns campos para controle da interrupção que constam no descritor.
5- Verificar se a interrupção não foi bloqueada pelo núcleo[10]. No caso da interrupção ter sido bloqueada pelo sistema operacional, a interrupção é ignorada.
6- Verificar se a interrupção tem alguma ISR cadastrada no descritor. Se não tiver a interrupção é ignorada.
7- Executar um destravamento (*unlock*) no descritor da interrupção.
8- Chamar a função `handle_IRQ_event`.

4.3.6.3 Análise da Rotina: `handle_IRQ_event`

Na figura 4.5 tem-se parte do código fonte da função `handle_IRQ_event`.

Esta função basicamente realiza as seguintes operações:

1- Se a interrupção pode ser tratada com as interrupções habilitadas, as habilita (linha 002).

[10] O Linux possui um mecanismo em software para detectar e isolar interrupções espúrias geradas pelo hardware.

2- Executar todas as ISR cadastradas no descritor (linhas 003 a 012).

```
001   // Verifica se pode habilitar as interrupções
002   if (!(action->flags & IRQF_DISABLED))
                      local_irq_enable_in_hardirq();

003    do {
004       // Executa a Rotina de Serviço de Interrupção (ISR)
005       ret = action->handler(irq, action->dev_id);

006       // Se a IRQ foi tratada atualiza o status
007       if (ret == IRQ_HANDLED) status |= action->flags;

008       // Carrega o valor de retorno
009       retval |= ret;

010       // Busca a próxima Rotina de Serviço de Interrupção
011        action = action->next;

012      } while (action);

013   // Verifica se dispositivo é gerador de entropia
014   if (status & IRQF_SAMPLE_RANDOM)
                      add_interrupt_randomness(irq);

015    // Desabilita as interrupções
016    local_irq_disable();
017    return retval;
```

Figura 4.5 – Parte principal do código da função `handle_IRQ_event`
Extraído de `linux/kernel/irq/handle.c`

A latência de entrada no tratador de interrupções é definida como sendo o tempo de execução decorrido desde a primeira instrução da rotina `do_IRQ` (item 4.3.6.1) até a instrução que habilita ou não as interrupções na função `handle_IRQ_event` (figura 4.5, linha 002).

4.3.7 Latência de Compartilhamento

Na figura 4.5, linha 005, pode-se ver a chamada à ISR que efetivamente vai tratar a interrupção. Como o Linux permite o compartilhamento de interrupções, deve acionar todas ISRs cadastradas para tratar uma determinada interrupção e por isso a necessidade do laço (linha 003, linha 011 e linha 012) que acaba gerando a latência de compartilhamento.

Observa-se que se a IRQa fosse compartilhada, só após o final do laço seria possível afirmar se a interrupção foi gerada pelo sinal de amostragem ou por outro dispositivo; já se a IRQa não for compartilhada essa informação é obtida apenas pelo número da interrupção que é conhecido ao iniciar a rotina `do_IRQ`. Este fato justifica parte do 3º Requisito visto no Capítulo 3, item 3.4.

Para uma IRQa tem-se que, ao terminar o laço:

- O dado foi lido da placa adaptadora e transferido para o espaço de memória do núcleo se necessário.
- O processo PDS teve seu estado trocado de bloqueado para pronto e, portanto, deve ser colocado no estado em execução.
- O sistema operacional foi avisado que há uma solicitação de troca de processo. Isto é feito colocando o bit `TIF_NEED_RESCHED` igual a 1 (um) no campo `flags` do processo corrente.

4.4 Estabelecendo Limites entre Latências

Com base no item 4.3.7 pode-se estabelecer que:

- O fim do laço (linha 012 da figura 4.5) é o término da latência de tratamento de interrupção.
- A primeira instrução após o fim do laço (linha 014 da figura 4.5) é o início da latência de despacho, a qual será vista no item 4.5.

4.5 Latência de Despacho

Como no item 4.3, a latência de despacho será divida em várias componentes. A figura 4.6 mostra as componentes da latência de despacho, as quais são definidas no decorrer deste item.

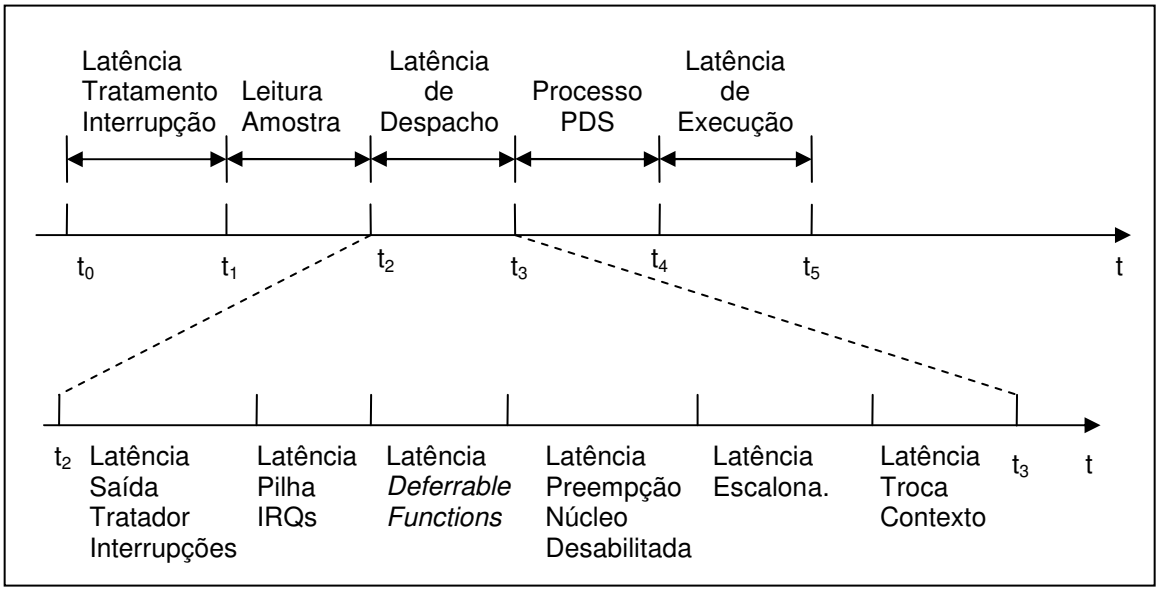

Figura 4.6 - Componentes da latência de despacho

4.5.1 Latência de Saída do Tratador de Interrupções

A latência de saída do tratador de interrupções é o tempo gasto com as operações realizadas depois que a ISR foi executada e, que decorrem do Linux ser um sistema operacional de uso geral.

4.5.1.1 Análise da Rotina: `handle_IRQ_event`

Na figura 4.5 tem-se as seguintes operações:

1- Se o dispositivo pode ser uma fonte de eventos aleatórios, o mesmo é usado para o gerador de números aleatórios do núcleo (linha 014).
2- Desabilitar as interrupções (linha 016).
3- Retornar à rotina `__do_IRQ`.

Do analisado no passo 1, tem-se:

12º Requisito

A IRQa não deve ser fonte de evento aleatório.

4.5.1.2 Análise da Rotina: `__do_IRQ`

1- Executar a função `note_interrupt` que analisa o tratamento da interrupção. Se a interrupção não foi tratada por nenhuma ISR, o núcleo incrementa um contador que faz parte do mecanismo para detectar se o hardware está gerando interrupções espúrias. Caso o núcleo conclua que a interrupção é espúria, altera o campo `status` do descritor da interrupção para `IRQ_DISABLED` e passa a ser ignorada (ver item 4.3.6.2, passo 5).
2- Executar um travamento (*lock*) no descritor da interrupção.
3- Atualizar alguns campos para controle da interrupção que constam no descritor.
4- Executar o método `end` do objeto `chip` que consta no descritor da interrupção. Este método sinaliza o controlador programável de interrupções e, seu funcionamento depende do modelo de controlador instalado no hardware.
5- Executar um destravamento (*unlock*) no descritor da interrupção.
6- Retornar à rotina `do_IRQ`.

4.5.1.3 Análise da Rotina: `do_IRQ`

Na figura 4.7 tem-se as instruções finais da rotina `do_IRQ`:

```
001    // Rotina __do_irq, supondo caso padrão (ver item 4.3.6.1)
002    desc->handle_irq(irq, desc);

003    irq_exit();
004    set_irq_regs(old_regs);
005    return 1;
```

Figura 4.7 – Trecho final do código da rotina `do_irq`
Extraído de `linux/arch/x86/kernel/irq_32.c`

No código da figura 4.7 pode-se ver:

1- A linha 003 executa a rotina `irq_exit`.
2- A linha 004 recupera os valores antigos dos registradores.

Na figura 4.8 tem-se parte do código fonte da rotina `irq_exit`.

```
001    /*
002     Exit an interrupt context. Process softirqs if needed and
003     possible
004    */

005    void irq_exit(void)
006    {
007            account_system_vtime(current);
008            trace_hardirq_exit();
009            sub_preempt_count(IRQ_EXIT_OFFSET);

010            if (!in_interrupt() && local_softirq_pending())
                                          invoke_softirq();

               /* Linhas relacionadas com o parâmetro de compilação
                CONFIG_NO_HZ foram excluídas */

011            preempt_enable_no_resched();
012    }
```

Figura 4.8 – Código parcial da rotina `irq_exit`
Extraído de `linux/kernel/softirq.c`

No código da figura 4.8 três instruções merecem ser observadas:

1- Na linha 009, o Linux decrementa o contador de interrupções aninhadas.
2- Na linha 010, se não há mais interrupções aninhadas e há *deferrable functions* pendentes, o Linux inicia a execução das mesmas.
3- Quando o Linux executa uma *deferrable function* bloqueia a preempção do núcleo e, conseqüentemente, ao terminar tem que desbloqueá-las como é visto na linha 011.

Portanto, após o tratamento da última interrupção aninhada, o Linux executa as *deferrable functions*, se houver, o que gera uma latência chamada latência de *deferrable functions*. Esta definição sugere o requisito abaixo:

13º Requisito

Durante o tratamento da IRQa as *deferrable functions* não devem ser executadas.

4.5.2 Latência de Pilha de IRQs

Foi visto no item 4.3.6.3 que o tratador de interrupções pode habilitar as interrupções. Desta forma é possível que quando a IRQa chegue à CPU, o sistema esteja executando o tratador de uma interrupção IRQn (com n≠a), que obrigatoriamente habilitou as interrupções.

O Linux 2.6.25 não permite que haja a preempção do núcleo quando uma interrupção está sendo tratada. Sendo assim, o processo PDS só pode ser passado ao estado em execução quando todas as interrupções pendentes tenham sido tratadas. Como as interrupções aninhadas formam uma pilha, define-se como latência de pilha de IRQs[11] o tempo necessário para terminar o tratamento de todas as interrupções na pilha, ou seja, para esvaziar a pilha de interrupções. Esta definição sugere o requisito a seguir:

14º Requisito

Um SO-PDS deve permitir a preempção do núcleo quando uma interrupção é tratada.

A explanação feita também motiva o seguinte requisito:

15º Requisito

Em um SO-PDS, a IRQa deve ser tratada com as interrupções desabilitadas.

O requisito acima evita que o tratamento da IRQa seja retardado por interrupções que estejam chegando ao sistema e, como pode ser visto na figura 4.5, este requisito não apresenta grande complexidade para ser implementado no Linux.

É interessante observar que caso o 14º requisito não possa ser implementado, por exemplo, devido à complexidade envolvida e, seja obrigatório esvaziar a pilha de IRQs, antes de colocar o processo PDS no estado em execução,

[11] Não confundir com pilha IRQ que foi vista no item 4.3.6.1.

pode ser estabelecido um requisito adicional como forma de mitigar a latência de pilha de IRQs.

Quando termina de tratar a IRQa, o Linux recupera o contexto (ou seja os valores do registradores quando da chegada da IRQa) e continua tratando a IRQn com as interrupções habilitadas até que o tratador as desabilite (linha 016 da figura 4.5). Estendendo o raciocínio, todas as interrupções da pilha serão executadas com as interrupções habilitadas por certo intervalo de tempo. Este fato levanta mais um requisito:

16º Requisito[12]

Após o tratamento da IRQa as interrupções devem ficar desabilitadas até que o processo PDS esteja no estado em execução.

Embora o requisito acima só tenha função no caso em que o núcleo não sofre preempção quando está tratando uma interrupção, o mesmo será mantido como uma forma de diminuir a latência de pilha de IRQs.

4.5.3 Latência de *Deferrable Functions*

Define-se como latência de *deferrable functions* o tempo decorrido na execução das *deferrable functions*, como explicado no item 4.5.1.3.

4.5.4 Latência de Preempção do Núcleo Desabilitada

No item 4.5.2 foi estudado o caso em que estava sendo tratada uma interrupção quando da chegada da IRQa à CPU. Neste item será estudado o caso em que o núcleo está tratando uma exceção, sendo que a chamada ao sistema será considerada um caso particular de exceção.

Antes disso, um esclarecimento. O Linux 2.6.25 habilita a preempção do núcleo através do parâmetro de compilação `CONFIG_PREEMPT`. Se o parâmetro está

[12] Este requisito é ampliado no item 4.6.1.

definido, o núcleo pode sofrer preempção; se não está definido, não pode sofrer preempção. Neste item será assumido que o núcleo pode sofrer preempção.

Mesmo no Linux 2.6.25 compilado com preempção, existem situações nas quais a preempção do núcleo é desabilitada (ex: execução de *deferrable functions*), em uma forma semelhante à que é feita com as interrupções. Isto faz com que depois de tratar uma interrupção, o sistema operacional tenha que verificar se a preempção do núcleo está habilitada ou desabilitada no caso de haver a necessidade de uma troca de processo.

Ao final do tratamento da IRQa, o núcleo foi sinalizado da necessidade de uma troca de processo e, portanto, precisa verificar o estado da preempção do núcleo para efetuar ou não a troca de processo. Nesse caso tem-se:

- Se a preempção do núcleo estiver desabilitada, o núcleo retoma o tratamento da exceção até que termine ou até que a preempção seja habilitada. Normalmente quando o sistema operacional habilita a preempção do núcleo, verifica se há necessidade de troca de processo e, em havendo, chama o escalonador. Quando o núcleo finaliza o tratamento de uma exceção faz uma verificação semelhante.

- Se a preempção do núcleo estiver habilitada, o núcleo chama o escalonador.

No caso em que o núcleo não foi compilado com preempção tem-se que o escalonador só será chamado depois que o tratamento da exceção termine.

Da descrição acima fica claro que independente da forma de compilação pode ocorrer uma latência em decorrência da preempção do núcleo estar desabilitada. Este fato motiva o requisito a seguir:

17º Requisito

Em um SO-PDS a preempção do núcleo não deve ser desabilitada.

4.5.5 Latência de Escalonamento

Como foi visto no item 4.3.7, a ISR sinalizou ao núcleo a necessidade da troca do processo, haja vista que o processo PDS passou ao estado pronto. O núcleo deve acionar o escalonador para verificar qual é o processo que assumirá a posse da CPU. A latência de escalonamento é o tempo gasto pelo escalonador para selecionar o processo PDS.

É importante destacar que esta latência não mede apenas o tempo para executar o algoritmo de escalonamento, mas sim o tempo total necessário para que o processo PDS seja selecionado. No Linux esta latência vai depender da existência, ou não, de processos com prioridade maior ou igual à do processo PDS no sistema[13].

Em um SO-PDS, embora esta latência possa ser diminuída, não pode ser eliminada, haja vista que o núcleo precisa determinar qual é o processo PDS e verificar o seu estado.

4.5.6 Latência de Troca de Contexto

Uma vez selecionado o processo PDS, para ser colocado em execução, o sistema operacional deve efetuar a troca dos contextos, a qual origina a chamada latência de troca de contexto.

4.5.7 Observações Finais

1ª Observação:

Se a IRQa chega à CPU quando um processo está sendo executado em Modo Usuário, ao término do tratamento da IRQa o núcleo chama o escalonador. Neste caso a latência de pilha de IRQs e de preempção do núcleo desabilitada tem valor igual a zero.

[13] O mecanismo de prioridade no escalonador do Linux é tratado no Capítulo 5, item 5.6.

2ª Observação:

A referência (24) mostra que no Linux:

1- O tratador de uma exceção pode ser interrompido por um tratador de interrupção ou por outro tratador de exceção. Decorre deste fato que pode haver exceções aninhadas. O número máximo de exceções aninhadas é igual a dois, a saber: uma chamada ao sistema e uma exceção de falta de página.

2- O tratador de uma interrupção só pode ser interrompido por outro tratador de interrupção.

Com essas duas informações, os raciocínios desenvolvidos nos itens 4.5.2 e 4.5.4 podem ser estendidos para casos mais complexos de interrupções aninhadas em exceções.

3ª Observação

É importante observar que o estudo realizado neste item 4.5 foi feito com a hipótese de que haverá uma troca de processo após o tratamento da interrupção, haja vista que é o caso na sinalização do instante de amostragem. Modificações devem ser feitas para o caso de não haver troca de processo, que estão abaixo a título de tornar o texto completo.

No caso de não haver troca de processo:

1- As latências de preempção do núcleo desabilitada, escalonamento e troca de contexto não se aplicam.

2- Define-se uma latência nova que é a Latência de Recuperação de Registradores que é o tempo necessário para recuperar os valores dos registradores que foram armazenados na pilha (item 4.3.5).

4.6 Latência de Execução

A latência de execução pode ser definida como a diferença entre o tempo efetivamente gasto no processamento da amostra de entrada e o tempo que seria gasto se o processamento fosse efetuado de forma atômica. Os fatores que podem retardar o processamento são:

- Interrupções E/S
- *Deferrable Functions*
- Memória Virtual

4.6.1 Interrupções E/S

O processo PDS pode ser retardado pela sinalização de interrupções pelos dispositivos de hardware, portanto, do ponto de vista de otimizar o tempo de computação é interessante colocar a restrição a seguir:

18º Requisito

O processo PDS deve ser executado com as interrupções E/S desabilitadas.

Pode-se agrupar o 16º Requisito (ver item 4.5.2) e o 18º Requisito em um único requisito mais completo:

19º Requisito

Desde a sinalização da IRQa até o que o processo PDS passa ao estado bloqueado, as interrupções devem estar desabilitadas.

Como desabilitar as interrupções é um tema tratado no Capítulo 7.

4.6.2 ***Deferrable Functions***

O Linux executa as *deferrable functions* à custa do processo corrente e por isso também recebem o nome de *soft irqs*. Foi visto que as *deferrable functions* são acionadas após uma interrupção ser tratada, mas ocorre que este não é o único momento em que podem ser chamadas. Essas funções também são acionadas ao final do tratamento de uma exceção. A referência (24) explica que embora existam certos pontos padrão de chamada, podem existir outros em função da versão e do distribuidor do Linux.

Portanto não se pode assumir que desabilitando as interrupções, está-se desabilitando as *deferrable functions*. Existe a possibilidade do processo PDS executar uma chamada ao sistema que ao ser tratada aciona as *deferrable functions*.

Decorre do exposto acima o requisito:

20º Requisito

Desde a chegada da IRQa até o que o processo PDS passa ao estado bloqueado, as *deferrable functions* não devem ser executadas.

Como atender este requisito é um tópico tratado no Capítulo 7.

4.6.3 **Memória Virtual**

Nos microcomputadores modernos os processos possuem uma memória virtual. Dependendo da utilização da memória *Random Access Memory* (RAM) pelo sistema, existe a possibilidade de que algumas páginas do espaço de memória do processo PDS estejam no disco rígido. Desta forma ao chegar o sinal de amostragem, existe a possibilidade que o processo PDS não tenha parte do programa ou dos dados carregados na memória RAM e ocorra uma (ou mais de uma) Falta de Página (*Page Fault)* durante o processamento da amostra. Neste caso o sistema operacional deve carregar a página que está no disco rígido, o que

aumenta de forma significativa o tempo para tratar a amostra. Esta falta de página pode ocorrer por três fatores:

- No Linux a carga do programa é sob demanda. Sendo assim, dependendo do tamanho do programa pode ocorrer que o processo PDS não esteja com todas as instruções em memória.

- O Linux remove da RAM páginas de memória pouco usadas para atender a requisição de memória de outros processos.

- O comando `malloc` é executado sob demanda e, portanto, quando um programa requisita memória de forma dinâmica, a mesma só é fisicamente alocada quando for usada.

O Linux permite a proteção das páginas de memória de um processo. As páginas protegidas não são removidas da memória RAM para o disco pela rotina de substituição de páginas. Decorre do exposto e considerando que o tamanho de um programa PDS normalmente é pequeno, se comparado à memória RAM de um microcomputador, pode-se estabelecer o requisito a seguir:

21º Requisito

a) Um SO-PDS deve carregar o programa PDS totalmente e não sob demanda.
b) Um SO-PDS deve proteger o espaço de memória do processo PDS de forma a impedir que páginas sejam removidas da memória RAM para o disco rígido.
c) Um programa PDS deve evitar a alocação dinâmica de memória.

4.7 Resumo

Neste capítulo as fontes de latência que influem no tempo de computação de um processo PDS foram analisadas para o Linux na arquitetura x86. No tocante ao hardware, o estudo focou nas interrupções E/S externas por serem as geradas pelos dispositivos mais comuns. É importante observar que a análise feita aqui para um processo PDS, pode ser aplicada sem alterações para um processo TR.

Capítulo 5 - Dispositivos de Tempo e o Escalonador no Linux

5.1 Introdução

No Capítulo 2, item 2.2.3, foi mostrado que, dependendo da forma como é controlada a amostragem, o SO-PDS deve dispor de um relógio ou de um temporizador exclusivo para controlar o período de amostragem. No Capítulo 3, item 3.4, foi estabelecido que essa fonte de tempo deve sinalizar o instante da amostragem através de uma interrupção E/S. Um dos objetivos deste capítulo é analisar os dispositivos de tempo da arquitetura x86 que atendem este requisito e como o Linux os utiliza e configura.

Ao final do capítulo, aspectos do escalonador são analisados, com o objetivo de determinar formas de adaptar o Linux para atender o requisito que o processo PDS seja único (Capítulo 3, item 3.7) e o requisito de preempção de processo seletiva (Capítulo 3, item 3.9).

5.2 Dispositivos de Tempo na Arquitetura x86

Os microcomputadores desenvolvidos com base nos microprocessadores da família x86 fornecem relógios e temporizadores para uso do sistema operacional. Os dispositivos de tempo que atendem o requisito de sinalizar o microprocessador por meio de uma interrupção E/S são:

- *Real Time Clock* (RTC)
- *Programmable Interval Timer* (PIT)
- *CPU Local Timer*
- *High Precision Event Timer* (HPET)

Existem outros dois mecanismos de tempo que não sinalizam ao microprocessador, a saber:

- *Time Stamp Counter* (TSC)
- *Advanced Configuration and Power Interface* (ACPI) *Power Management Timer*

5.2.1 ***Real Time Clock*** **(RTC)**

Os microcomputadores compatíveis com o IBM-PC incluem um relógio chamado *Real Time Clock* (RTC) que é independente da CPU e de outros circuitos. O RTC continua funcionando mesmo quando a máquina é desligada, já que é alimentado por uma bateria.

O RTC pode sinalizar de forma periódica com uma freqüência que varia de 2 Hz a 8.192 Hz por meio da IRQ8. Também pode ser programado para sinalizar a IRQ8 em um determinado horário.

O núcleo do sistema operacional pode acessar o RTC por meio das portas E/S `0x70` e `0x71`. O Linux usa o RTC para obter as informações de data e hora do sistema, sendo que o administrador pode ler e gravar o RTC executando o programa do sistema `clock`. O Linux também dá suporte para que os processos programem o RTC usando o dispositivo de arquivo `/dev/rtc`.

5.2.2 ***Programmable Interval Timer*** **(PIT)**

Os microcomputadores compatíveis com o IBM-PC também incluem um dispositivo de tempo chamado *Programmable Interval Timer* (PIT).

O funcionamento deste circuito de tempo é equivalente ao de um temporizador regressivo periódico, que a cada período de tempo sinaliza uma interrupção E/S ao microprocessador. O núcleo programa a freqüência da interrupção em múltiplos da freqüência do oscilador interno do PIT e usa essa interrupção, chamada de interrupção do relógio, para contar o número de períodos de tempo decorridos desde a carga do sistema.

O PIT é normalmente acessado pelas portas E/S `0x40` a `0x43` e pode, por exemplo, ser implementado com um integrado CMOS 8254. Neste caso a freqüência do oscilador interno é 1.193.182 Hz e, o contador programável é de 16 bits. Nas máquinas compatíveis com o IBM-PC, o PIT usa a IRQ0 para sinalizar a interrupção do relógio.

Durante a carga do sistema o Linux programa o PIT usando a constante `HZ`, a qual determina de forma aproximada o número de interrupções por segundo. No núcleo do Linux 2.6.25 esta constante é definida por default com o valor 1.000, o que equivale a uma interrupção de relógio a cada 1 ms aproximadamente. No entanto este valor pode variar dependendo da opção de compilação e da distribuição do Linux, já que basta alterar o valor da constante `HZ` e compilar o núcleo. O Linux estabelece uma faixa de valores válidos para `HZ`, que tem valor mínimo 12 e máximo 1.535. Caso o hardware tenha um sinal externo de sincronismo, o Linux pode alterar o contador do PIT para ajustar a freqüência da interrupção IRQ0.

Em uma máquina com um PIT de freqüência de oscilação igual a 1.193.182 Hz, tem-se que a IRQ0 pode ter uma freqüência mínima de 12,0000 Hz e uma freqüência máxima de 1.535,63 Hz.

5.2.3 ***CPU Local Timer***

A partir do Pentium III, a Intel introduziu um controlador programável de interrupções chamado *Advanced Programmable Interrupt Controller* (APIC) Local. O APIC Local disponibiliza ao núcleo um temporizador chamado *CPU Local Timer,* cujo funcionamento é semelhante ao PIT (item 5.2.2). As principais diferenças são:

- O contador programável possui 32 bits.

- Em sistemas SMMP, o APIC Local envia interrupções apenas para seu processador enquanto o PIT gera a interrupção de relógio de forma global, ou seja, pode ser tratada por qualquer CPU.

- Como o APIC Local não tem um oscilador interno exclusivo para o *CPU Local Timer*, a freqüência de operação é baseada no relógio do barramento[1] (*bus*). O contador pode ser programado para decrementar a cada 1, 2, 4, 6, 16, 32, 64 ou 128 períodos do sinal do relógio.

O Linux pode ser configurado pelo parâmetro `CONFIG_X86_LOCAL_APIC` para usar ou não o APIC Local[2]. No caso de estar configurado para usar o APIC Local, durante a carga do sistema o Linux tenta programar o *CPU Local Timer* para usar o vetor de interrupção 239 (`0xef`). Em não tendo sucesso, zera a variável do sistema `using_apic_timer` indicando que o *CPU Local Timer* não está ativo e, desta forma, o sistema operacional passa a usar apenas o PIT.

5.2.4 ***High Precision Event Timer*** **(HPET)**

O *High Precision Event Timer* (HPET) é um novo dispositivo de tempo, desenvolvido em conjunto pela Intel e Microsoft, que disponibiliza certo número de temporizadores por hardware que podem ser acessados pelo núcleo do sistema operacional. Apesar do HPET não ser comum em máquinas de usuário final, o Linux 2.6.25 oferece suporte ao mesmo.

A especificação para este componente prevê de 1 até 8 contadores independentes de 32 bits ou 64 bits. O contador deve ser crescente e controlado por seu próprio sinal de relógio, cuja freqüência deve ser de no mínimo 100 MHz. (62)

A cada contador podem ser associados até 32 temporizadores. Cada temporizador é composto de um comparador (*comparator*) e um registrador de conferência (*match register*). O comparador é um circuito que compara o valor do contador com o do registrador de conferência e, gera um sinal de interrupção se os valores são iguais. Alguns temporizadores podem ser configurados para sinalizarem de forma periódica. A especificação recomenda um mínimo de 3 temporizadores, sendo que o número máximo é de 256 (8 contadores com 32 temporizadores). (62)

---

[1] Em máquinas mais antigas é usado o sinal de relógio do barramento interno do APIC.
[2] O APIC Local é tratado com mais detalhes no Apêndice B, item B.6.2.

O chip do HPET pode ser programado pelo núcleo do sistema operacional através de registradores mapeados em memória. O mapeamento é estabelecido pelo BIOS que reporta ao núcleo a posição inicial em memória. (62)

O Linux 2.6.25, durante a carga do sistema operacional, verifica se o HPET está presente. Quando presente no hardware, o núcleo do Linux 2.6.25 utiliza os dois primeiros temporizadores, que são inicializados durante a carga. O temporizador 0 é usado para gerar um sinal de interrupção periódico e o temporizador 1 é usado para emular o RTC (item 5.2.1). O HPET pode ser programado por meio de um *driver*, o qual fornece duas APIs, uma para ser usada em programas executados em Modo Usuário e outra para programas executados em Modo Supervisor.

Na arquitetura x86 com Linux, este dispositivo vem suprir a falta de temporizadores de hardware que sinalizam por meio de uma interrupção e que possuam a resolução necessária a aplicativos PDS, como é analisado no item 5.3 e no item 5.4. Observa-se que pelas características explicadas, o HPET atende todos os requisitos para o controle da freqüência de amostragem por um SO-PDS quando em modo *Pull-In* (Capítulo 3, item 3.3 e item 3.4).

Apesar da importância do HPET, principalmente para o desenvolvimento de aplicativos PDS e TR, este dispositivo não será estudado neste trabalho porque as máquinas disponíveis para os testes não tinham esse dispositivo. Por esse motivo, as análises realizadas no item 5.3 e no item 5.4 partem da hipótese que o HPET não está presente no hardware.

5.2.5 *Time Stamp Counter* (TSC)

Todo microprocessador x86 inclui um pino de entrada chamado `CLK` que recebe o sinal de relógio do oscilador externo. A partir do Pentium, os microprocessadores da família x86 possuem um contador que é incrementado a cada sinal do relógio. O contador é acessível por meio do registrador de 64 bits chamado *Time Stamp Counter* (TSC) que pode ser lido usando a instrução de montagem `rdtsc`. Quando o núcleo usa este registrador para medir intervalos de

tempo, deve levar em conta a freqüência do oscilador externo cuja aferição é feita durante a carga do sistema operacional.

5.2.6 ***ACPI Power Management Timer***

O *ACPI Power Management Timer* (ACPI PMT) é outro dispositivo de tempo incluído em praticamente todas as placas mãe baseadas em ACPI. A sua freqüência de relógio é fixada aproximadamente em 3,58 MHz. Este dispositivo é apenas um contador que é acrescido a cada sinal do relógio. Para ler este contador, o núcleo do sistema operacional acessa uma porta E/S determinada pelo BIOS, quando da inicialização do sistema.

Uma vantagem do ACPI PMT sobre o TSC é que o primeiro mantém a freqüência de operação fixa, enquanto a freqüência do segundo pode variar, já que está relacionada à freqüência de operação do oscilador externo do microprocessador. Esta vantagem pode ser observada em máquinas nas quais o BIOS pode dinamicamente diminuir a tensão de alimentação do processador para economizar energia da bateria, alterando assim a freqüência de operação do TSC.

5.3 Utilização do PIT e do *CPU Local Timer* pelo Linux

O Linux utiliza o PIT (IRQ0) e o *CPU Local Timer* (IRQv239[3]) para atualizar as suas estruturas de tempo. Como foi comentado, durante a carga do sistema operacional o Linux verifica se o *CPU Local Timer* está ou não disponível para uso. Essa disponibilidade determina a forma de funcionamento do núcleo:

- **Se o *CPU Local Timer* não está ativo.** O núcleo utiliza a IRQ0 para efetuar todas as atualizações nas estruturas de tempo.

- **Se o *CPU Local Timer* está ativo**. O núcleo utiliza a IRQv239 para atualizar as estruturas de tempo ligadas à CPU específica que gerou a

[3] Neste texto será usada a notação IRQv239 para indicar a interrupção cujo vetor é igual a 239. Desta forma, usa-se IRQi para representar o número do sinal da interrupção e IRQvi para indicar o vetor da interrupção. Ver Apêndice C, item C.3 para uma explicação sobre a necessidade desta diferenciação.

interrupção como, por exemplo, o tempo de execução de um processo. O núcleo utiliza a IRQ0 para atualizar as estruturas de tempo gerais do sistema como, por exemplo, temporizadores de aplicativos.

5.4 Conclusões Relacionadas aos Dispositivos de Tempo

Lembrando da necessidade de uma fonte de tempo no caso do SO-PDS funcionar no modo *Pull-In* (Capítulo 3, item 3.3) e considerando o analisado nos itens anteriores, algumas conclusões já podem ser obtidas.

5.4.1 *CPU Local Timer* Não Está Ativo

No caso do *CPU Local Timer* não estar ativo tem-se:

- O sistema operacional não dispõe de um mecanismo de tempo para usar exclusivamente no controle da freqüência de amostragem.

- A freqüência de amostragem só pode ser um submúltiplo inteiro da freqüência da IRQ0 a qual, como foi visto, é normalmente 1.000 Hz (aproximadamente). Embora a freqüência de amostragem dependa do tamanho do bloco de dados[4], de forma geral, uma freqüência de 1.000 Hz pode ser considerada baixa para os aplicativos PDS mais comuns.

- Como o instante de amostragem é sinalizado pela IRQ0, o tratador de interrupções deverá acionar duas rotinas de serviço de interrupção. A primeira ISR é a `timer_interrupt` que atualiza as estruturas de tempo globais e locais do Linux. A segunda ISR é a que lê a amostra do dispositivo externo. O uso da IRQ0 para executar essas duas rotinas acarreta um aumento da latência de despacho. É interessante observar que esta nova latência não pode ser considerada como sendo a latência de compartilhamento (Capítulo 4, item 4.3.7), haja vista que as causas que as geram não são as mesmas.

[4] Lembra-se que neste texto, o termo amostra é usado de uma forma mais geral, ou seja, pode ser um único dado (ex: caractere) ou um conjunto de dados (ex: bloco).

5.4.2 *CPU Local Timer* Está Ativo – Monoprocessador

No caso do *CPU Local Timer* estar ativo e o sistema ser monoprocessador, este dispositivo pode ser usado para controlar a freqüência de amostragem, adaptando o mecanismo de checagem do dispositivo (explicado no item 5.2.3) da forma mostrada a seguir.

Durante a carga do sistema uma vez verificado que o *CPU Local Timer* está presente e ativo, pode-se zerar a variável `using_apic_timer`. Como a variável está zerada, o núcleo atualizará todas as estruturas de tempo por meio da IRQ0 e a IRQv239 ficará livre para controlar a amostragem. Esta alteração requer alguns cuidados:

1- A entrada da tabela descritora das interrupções[5] (IDT) para a IRQv239 deve ser configurada adequadamente, ou seja, com o endereço da rotina que irá ler a amostra do dispositivo externo.

2- O *CPU Local Timer* só deve ser configurado e liberado para sinalizar a IRQv239 quando o aplicativo PDS estiver no sistema, caso contrário serão geradas interrupções espúrias[6].

5.4.3 *CPU Local Timer* Está Ativo – Multiprocessador

Embora o foco deste trabalho não sejam sistemas SMMP, neste item são feitas algumas observações, para esse tipo de hardware, em relação à disponibilidade dos dispositivos de tempo.

Em sistemas SMMP o Linux usa a IRQ0 para atualizar as estruturas de tempo globais e a IRQv239 para atualizar as estruturas de tempo locais. Portanto, não há fontes de tempo livres para serem usadas no controle da freqüência de amostragem.

---

[5] A tabela descritora das interrupções (IDT) é vista no Apêndice B, item B.6.4.

[6] Uma forma alternativa seria configurar o *CPU Local Timer* com uma freqüência baixa durante a carga do sistema e usar o mecanismo de bloqueio de interrupções espúrias do Linux (Capítulo 4, item 4.5.1.2).

Uma solução para esse problema, seria usar a IRQ0 ou a IRQv239 para o controle da freqüência de amostragem, de forma semelhante à explicada no item 5.4.1 (executando duas ISRs com a interrupção) e, portanto, com as mesmas limitações levantadas naquele item. Neste caso, por ser uma interrupção externa, a utilização da IRQ0 traz duas vantagens, se comparada à IRQv239:

- Por ser uma interrupção global, qualquer processador pode atendê-la.

- Do ponto de vista de desenvolvimento, a introdução de uma segunda ISR para ser acionada pela IRQ0 é relativamente simples, usando os recursos para compartilhar interrupções do Linux. Ver Apêndice C, item C.2.2.

Uma segunda solução seria liberar o APIC Local de um dos processadores para o controle da freqüência, de forma semelhante à explicada no item 5.4.2. Esta solução tem a desvantagem que só o processador escolhido para liberar o temporizador executa o processo PDS. Também introduz o problema da atualização das estruturas de tempo locais desse processador, haja vista que não se pode usar diretamente IRQ0 para essa finalidade, a qual por ser uma interrupção global, pode ser tratada por qualquer processador.

5.5 Importância dos Tratadores de Interrupção em Sistemas TR

Em sistemas de tempo real é importante que o tempo de computação dos tratadores de interrupção seja o menor possível de forma a minimizar a latência do sistema. Este fator se torna mais crítico no caso dos tratadores de interrupção que não podem ser executados com as interrupções habilitadas.

No Linux 2.6.25, as rotinas que tratam a IRQ0 (`timer_interrupt`) e a IRQv239 (`smp_local_timer_interrupt`) têm duas características em comum:

1- São executadas de forma periódica, haja vista que tratam duas interrupções sinalizadas por temporizadores, que são configurados pelo núcleo para funcionarem de forma periódica (item 5.2.2 e item 5.2.3).

2- Como essas rotinas atualizam estruturas para controle de tempo do Linux, são executadas com as interrupções desabilitadas. (24)

Pode-se concluir que para essa versão do Linux, a soma dos tempos de computação dessas duas rotinas estabelece um limite mínimo para o período de amostragem no caso de aplicativos PDS, assim como para o tempo de resposta em sistemas de tempo real.

Embora a solução ideal em um SO-PDS seja que todo o núcleo possa sofrer preempção, como as ISRs dos relógios basicamente atualizam estruturas de tempo do núcleo, transformar essas rotinas em reentrantes[7], para que possam ser executadas com as interrupções habilitadas, pode acarretar em uma lógica de sincronização complexa. Como uma forma de contornar essa complexidade, pode-se adotar uma solução não ótima através do requisito abaixo:

22º Requisito

Em um SO-PDS as atualizações das estruturas de tempo devem ser feitas de forma eficiente, com a finalidade de obter tempos baixos de computação se comparados ao período de amostragem.

No Linux, uma linha de estudo para atender esse requisito seria a simplificação dessas rotinas, abrindo mão de alguns recursos, como por exemplo, as estatísticas de uso do sistema, para possibilitar períodos menores de amostragem.

5.6 O Escalonador do Linux

O Linux divide os processos existentes no sistema de duas formas: por classes de escalonamento e por prioridade.

[7] A noção de rotina reentrante é apresentada no Apêndice A, item A.7.

As classes de escalonamento definem como o escalonador tratará os processos. Três são as classes: `SCHED_FIFO`, `SCHED_RR` e `SCHED_NORMAL`. As duas primeiras classes são usadas para processos de tempo real e a última para processos convencionais de tempo compartilhado.

Os processos de tempo real têm uma prioridade de tempo real que varia de 1 (maior prioridade) a 99 (menor prioridade). O escalonador usa esta prioridade para escolher qual dos processos de tempo real no estado pronto passará a ser executado.

Para todos os processos o Linux estabelece uma prioridade estática que varia de 100 (maior prioridade) a 139 (menor prioridade) e que pode ser alterada pelo usuário. Esta prioridade é usada para determinar o tamanho da janela de tempo de processamento, de tal forma que maior prioridade implica em maior janela de tempo.

Para os processos convencionais, o Linux também estabelece uma prioridade dinâmica que o escalonador atualiza de acordo com o perfil de processamento. Esta prioridade dinâmica é usada pelo escalonador para selecionar o processo convencional que passará ao estado em execução quando é feita a troca de processo.

5.6.1 Classes de Processos

Neste item será visto como o Linux trata as diferentes classes.

`SCHED_FIFO` – Nesta classe o escalonador preserva a posição do processo na fila de prontos quando o coloca em execução. Como o processo não tem alterada sua posição, o mesmo só sofre preempção quando outro processo de tempo real de maior prioridade estiver no estado pronto. Se não for o caso, o processo detém a posse da CPU sem restrição de tempo até que a libere, por exemplo, para esperar um recurso.

SCHED_RR – Nesta classe o escalonador passa o processo para o final da fila de prontos quando o coloca em execução. Esta classe de processo sofre preempção em duas situações. A primeira é quando um processo de tempo real de maior prioridade estiver no estado pronto. A segunda é quando ao término de sua janela de tempo um processo de tempo real de igual prioridade estiver no estado pronto.

SCHED_NORMAL – Processos normais de tempo compartilhado.

5.6.2 Análise

Neste item é analisado o comportamento do escalonador com base no modelo de dois processos estabelecido no Capítulo 2, item 2.5.

Como o processo PDS é um processo de tempo real e o de maior prioridade, no Linux seria um processo da classe `SCHED_FIFO` e prioridade de tempo real igual a 1 (um). Pelo exposto no item 5.6.1 pode-se concluir que o processo PDS só perderá a posse da CPU quando terminar o tratamento da amostra e passar ao estado bloqueado para esperar o próximo instante de amostragem. Portanto o Linux na sua versão 2.6.25 atende o requisito de preempção de processo seletiva (ver Capítulo 3, item 3.9).

O Linux não atende o requisito que o processo PDS seja único (ver Capítulo 3, item 3.7), haja vista que pode existir mais de um processo da classe `SCHED_FIFO` e prioridade de tempo real igual a 1 (um). Partindo da hipótese que o processo PDS e o processo NPDS sejam dessa mesma classe e prioridade, esta não unicidade implica na seguinte limitação:

1- Como foi visto, o processo PDS passa ao estado bloqueado para esperar o próximo instante de amostragem e, portanto, deixa a fila de prontos.

2- Na hipótese do processo NPDS tomar a posse da CPU no intervalo de tempo decorrido desde o bloqueio até a chegada do sinal de amostragem, o processo PDS mesmo no estado pronto só poderá retomar a posse da CPU quando o processo NPDS a liberar.

Pode-se ver que essa forma de operação aumenta a latência de preempção e que o requisito do processo PDS ser único é uma forma de se garantir a maior prioridade para este processo.

5.7 Resumo

Neste capitulo foram abordados dois temas importantes para o objetivo de aperfeiçoar o Linux para o processamento digital de sinais. Primeiramente foram estudadas as fontes de tempo nos microcomputadores compatíveis ao IBM-PC e como o Linux as utiliza. Constatou-se que só em um caso particular existe um mecanismo de tempo que possa ser usado exclusivamente para controlar a freqüência de amostragem. Também foi visto o escalonador e se constatou uma limitação no processamento em tempo real que decorre do Linux permitir mais de um processo com prioridade máxima.

Capítulo 6 - Medição da Latência no Tratamento das Interrupções E/S Externas no Linux

6.1 Introdução

No Capítulo 4 foi apresentado um modelo para a latência no tratamento das interrupções E/S. Neste capítulo é apresentado um método que mede a latência total e algumas de suas componentes no tratamento da interrupção IRQ0. O método proposto também permite estimar algumas das componentes definidas pelo modelo do Capítulo 4 para outras interrupções E/S externas.

O método mede a latência diretamente no código do núcleo do Linux, o qual é modificado colocando-se funções para medir o tempo de execução dos trechos a serem estudados. Esta forma de medição baseada na modificação do código é conhecida na literatura como instrumentalização do núcleo. Embora esta idéia não seja nova, o método aqui desenvolvido se propõe a fazer as medições com base em um modelo mais completo do que o normalmente usado.

O capítulo termina apresentando e analisando os dados obtidos com a aplicação do método em três cenários diferentes de carga e configuração do Linux.

6.2 Introdução ao Método de Medição

Neste item é apresentada uma introdução ao método de medição e às suas concepções básicas.

A figura 6.1 mostra um esquema simplificado do método de medição, onde se pode ver que o projeto está dividido em três módulos que são explicados a seguir.

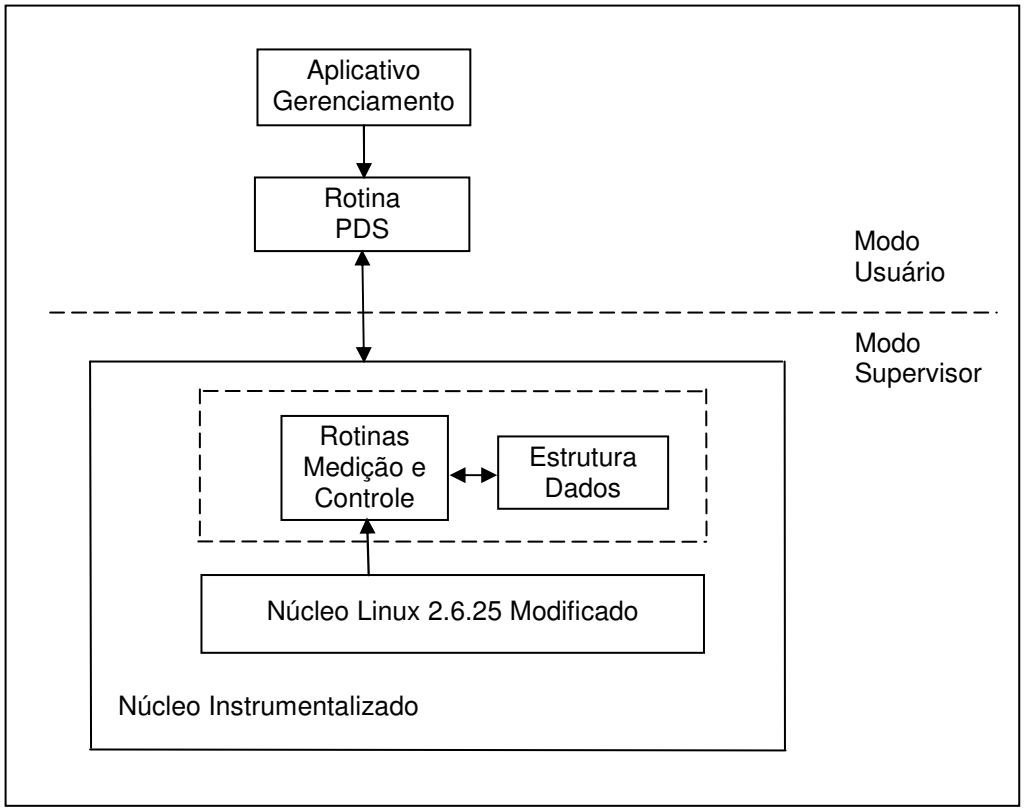

Figura 6.1 – Método para medição da latência no tratamento das interrupções E/S externas no Linux

6.2.1 Núcleo Instrumentalizado

O primeiro módulo consiste nas alterações introduzidas no núcleo para a sua instrumentalização. Essas alterações podem ser divididas em duas partes.

A primeira parte é representada na figura 6.1 pela caixa “Núcleo Linux 2.6.25 Modificado”. Consiste na alteração dos trechos do núcleo original, a serem estudados, através da colocação de chamadas a funções de medição de tempo que, no texto, serão chamadas de marcadores. Esses marcadores lêem mecanismos de tempo do sistema, marcando os instantes em que um trecho de interesse é iniciado e finalizado, de forma que por diferença pode-se obter a latência.

A segunda parte é representada na figura 6.1 pela área pontilhada. É composta por um conjunto de rotinas e de uma estrutura de dados, que foram incorporados ao núcleo original. A estrutura de dados tem a função de armazenar

os valores dos marcadores e as informações para controle da medição. O conjunto de rotinas é composto pelos marcadores, por rotinas para controle e por rotinas que fazem a interface com a Rotina PDS.

É importante esclarecer que os marcadores só lêem os tempos para uma determinada interrupção que é informada ao núcleo pela Rotina PDS. Por analogia à nomenclatura usada nos capítulos anteriores, a interrupção monitorada é chamada de IRQa.

6.2.2 Rotina PDS

O segundo módulo consiste de uma rotina com a estrutura de um programa PDS (ver Capítulo 2, item 2.2.1), a qual é executada como um processo de usuário (ver Capítulo 3, item 3.5).

Esta rotina primeiramente informa ao núcleo o número da IRQa a ser monitorada. Na continuação, entra em um laço com um número fixo de repetições, onde são realizadas as seguintes operações:

1- Passa ao estado bloqueado, ficando à espera da interrupção de amostragem IRQa[1].
2- Lê os valores medidos no tratamento da IRQa, os quais estão gravados na estrutura de dados no espaço de memória do núcleo.
3- Grava os valores medidos em uma matriz no espaço de memória do processo.

Finalizando o laço, a Rotina PDS grava o conjunto de dados em um arquivo texto no disco rígido.

[1] Ao detectar a IRQa, o tratador de interrupções coloca o processo no estado pronto e sinaliza o escalonador da necessidade da troca de processo.

6.2.3 Aplicativo de Gerenciamento

Este módulo é composto por um programa cuja função é gerenciar o processo de medição. A idéia básica deste programa é permitir que intervalos com medição sejam intercalados de intervalos sem medição, como mostrado na figura 6.2. O objetivo deste procedimento é equilibrar o número total de dados com o tempo total de monitoração, como mostrado no item 6.5.1.

As principais funções deste programa são:

1- Controlar a duração do intervalo sem medição.
2- Chamar a Rotina PDS passando o número de repetições, ou seja, o número de medições a serem realizadas. Número este que determina a duração aproximada do intervalo com medição.
3- Controlar o número total de conjuntos de medidas, o que determina aproximadamente o tempo total de monitoração.

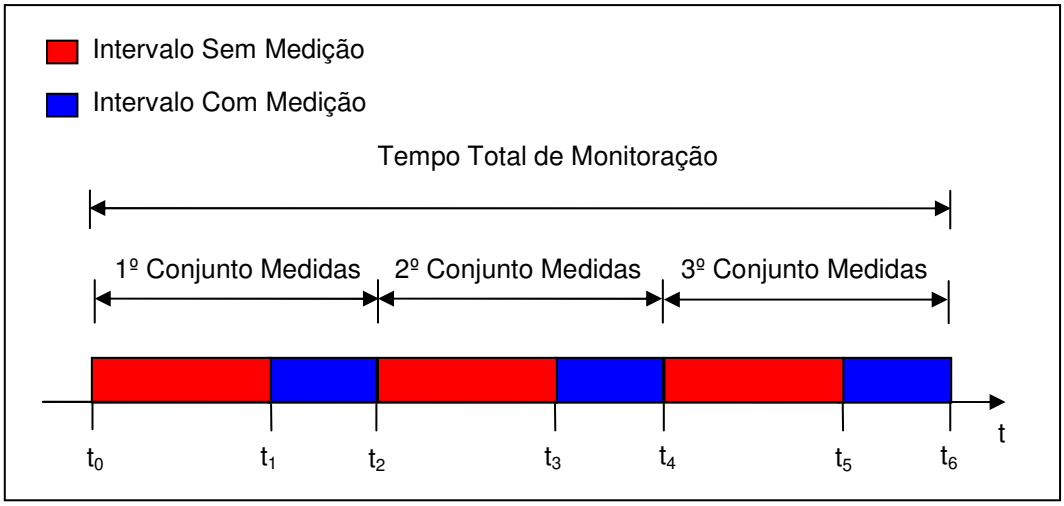

Figura 6.2 – Esquema para medição da latência

6.3 Motivação

Embora existam dispositivos de hardware específicos para a medição da latência em sistemas, a motivação para o desenvolvimento proposto neste capítulo será justificada a seguir.

Alguns trabalhos anteriormente realizados sobre a latência do Linux (25), (46) usaram métodos de medições que podem ser chamados indiretos, haja vista que são baseados em *benchmarks* do sistema operacional. Outros trabalhos já realizados usaram meios diretos de medição (63), (64), mas não se basearam em um modelo de latência tão completo como o proposto neste estudo.

Como a medição é feita de forma a simular um processo PDS, os valores obtidos com o método proposto servem como referência para avaliar as modificações que são introduzidas no Linux para o processamento digital de sinais no Capítulo 7.

A medição direta das várias componentes da latência serve como base para um método adaptativo que procura mitigar uma dessas componentes em um SO-PDS. Uma das razões da latência de mascaramento (Capítulo 4, item 4.3.2) decorre de que trechos do tratador de interrupções são executados com as interrupções desabilitadas, latência esta que se agrava nos casos em que a rotina de serviço da interrupção não é reentrante. Em um SO-PDS pode-se ter uma estimativa do instante de chegada da IRQa, haja vista que o período de amostragem é conhecido. Se o SO-PDS também tiver estimativas do tempo de execução de todas as interrupções, o sistema pode avaliar se uma interrupção IRQn (com $n \neq a$) pode ou não ser tratada completamente antes da chegada IRQa, diminuindo assim a latência de mascaramento. Este tema é analisado no Capítulo 7.

6.4 Análise do Método

Neste item é feita uma análise do método e são estabelecidas as suas limitações.

6.4.1 Erro Sistemático

Uma possível crítica ao método é que a introdução dos marcadores de tempo implica em modificações no núcleo. Essas modificações, por sua vez, causam erros sistemáticos em decorrência do tempo de execução dos marcadores. Esses erros, porém podem ser minimizados medindo o tempo de execução dos

marcadores e fazendo as devidas compensações. Também é interessante observar que normalmente o tempo de processamento dos marcadores é bem menor do que os trechos que eles medem e, portanto, o erro sistemático não tem um valor alto.

6.4.2 Latência de Hardware e Latência de Software

Neste item são feitas duas definições que visam facilitar a descrição no restante do texto. As definições são uma conseqüência natural da forma como foi idealizado o método.

Decorre do explicado no Capítulo 4, item 4.3.5, que o primeiro marcador só pode ser colocado no tratador de interrupções após os registradores de uso comum serem salvos. Isto faz com que as latências causadas principalmente pelo hardware só possam ser medidas somadas. Desta forma, define-se a latência de hardware como sendo a soma das seguintes latências: leitura da interrupção, mascaramento, fila do PIC, operação do microprocessador e salvamento dos registradores (ver Capítulo 4, item 4.3).

Por analogia pode-se definir a latência de software como sendo aquela causada principalmente pela camada de software, ou seja, pelo núcleo do sistema operacional. Define-se latência de software como sendo a soma das seguintes latências: entrada no tratador de interrupções, compartilhamento, saída do tratador de interrupções, pilha de IRQs, *deferrable functions*, preempção do núcleo desabilitada, escalonamento e troca de contexto (ver Capítulo 4, item 4.3.6 e item 4.5). Ressalta-se que as componentes da latência de software são medidas individualmente pelo método proposto.

Das definições acima se tem que a latência total é soma da latência de hardware e da latência de software.

6.4.3 1ª Limitação - Latência de Hardware

A medição da latência de hardware requer um sincronismo entre o dispositivo externo que gerou a interrupção e o núcleo. Para medir esta latência

deve-se saber o instante em que o dispositivo sinalizou a IRQa e o instante em que o primeiro marcador de tempo no tratador de interrupções foi executado. Isto implica em que o relógio do dispositivo externo deve ser sincronizado com um dos relógios do sistema ou o dispositivo externo deve permitir acesso ao seu relógio ou temporizador.

À primeira vista existe a possibilidade de usar a IRQ0 ou a IRQv239 para medir a latência de hardware, pois ambas são sinalizadas por mecanismos de tempo da arquitetura x86, como mostrado no Capítulo 5 (item 5.3). No entanto deve-se descartar o uso da IRQv239 que, por não ser uma interrupção sinalizada por um dispositivo externo, é tratada de forma diferente pelo Linux como mostrado no Apêndice F. Conseqüentemente o método aqui proposto monitora a IRQ0 para medir a latência de hardware.

A interrupção IRQ0 é sinalizada quando o contador regressivo do PIT zera (Capítulo 5, item 5.2.2) e, assim, basta ao primeiro marcador ler esse contador para obter a latência de hardware.

Portanto, devido às limitações da arquitetura x86, o método só mede a latência de hardware para a IRQ0 e, conseqüentemente, só mede a latência total para esta interrupção.

Uma vantagem no uso da IRQ0 é que, sendo periódica, permite medições regulares da latência. A principal desvantagem é que as medições feitas com esta interrupção têm latência da fila do PIC nula, haja vista que esta é a interrupção de maior prioridade.

Em microcomputadores com APIC Local, a limitação acima pode ser contornada medindo também a latência de hardware com a IRQv239, a qual é sinalizada pelo *CPU Local Timer* (Capítulo 5, item 5.2.3). A lógica de programação é semelhante à usada para IRQ0, já que a IRQv239 é sinalizada quando zera o contador, o qual pode ser lido pelo núcleo. Como é o número do vetor que estabelece a prioridade, se a IRQ0 tem a maior prioridade, a IRQv239 tem uma das prioridades mais baixa. Portanto, em média, a diferença entre a latência de

hardware para a IRQv239 e para a IRQ0 é uma boa estimativa para a latência da fila do PIC. Esta forma de estimar a latência da fila do PIC não é estudada neste texto.

6.4.4 2ª Limitação - Latência de Software

Neste item é analisada a medição da latência de software, ou de outra forma, a medição das diversas componentes que formam esta latência.

Como o método consiste basicamente na colocação de marcadores em trechos do núcleo que tratam as interrupções E/S externas, pode-se concluir que não há, como no item anterior, uma restrição em relação à interrupção a ser monitorada para medir a latência de software e suas componentes.

Com base no apresentado no Capítulo 4, a seguir é descrita a dinâmica para medir a latência de uma interrupção que seja tratada com as interrupções habilitadas. Partindo da hipótese que a IRQa possa ser tratada com as interrupções habilitadas, tem-se o seguinte exemplo:

1- Em t_0 a IRQa começa a ser tratada com as interrupções desabilitadas.
2- Em t_1 o tratador de interrupções habilita as interrupções e, a seguir, a rotina de serviço da interrupção (ISR) que trata a IRQa é acionada. Ver Capítulo 4, item 4.3.6.3.
3- Por hipótese, em t_2 a execução da ISR da IRQa é interrompida pela chegada ao sistema de uma interrupção IRQn que começa a ser tratada. Neste caso, trechos do tratador de interrupções que contém os marcadores são indevidamente executados antes que a medição da IRQa seja finalizada.
4- Em t_3 o tratamento da IRQn finaliza e a ISR da IRQa é retomada.

Pelo modelo estabelecido no Capítulo 4, item 4.5.2 o tempo gasto para o tratamento da IRQn ($t_3 - t_2$) deve ser computado na latência de pilha de IRQs[2].

[2] A latência de pilha de IRQs é o tempo necessário para esvaziar a pilha de interrupções aninhadas. Como, por hipótese, a IRQa é tratada com as interrupções habilitadas, nesta caso, na pilha pode haver interrupções acima e abaixo da IRQa para serem tratadas.

Este exemplo simples mostra que a implementação do método demanda uma lógica de controle para, por exemplo, evitar que marcadores sejam acionados de forma errada e para contabilizar o tempo gasto com interrupções sinalizadas após o início da medição da IRQa.

Pode-se mostrar que a lógica de controle fica simplificada impondo-se a condição que a interrupção a ser medida não é tratada com as interrupções habilitadas. Na implementação desenvolvida neste trabalho esta restrição foi assumida pelas razões expostas a seguir:

1- O objetivo principal é medir a latência em um processo PDS e, como foi visto, a IRQa é tratada com as interrupções desabilitadas por requisito do SO-PDS. Ver Capítulo 4, item 4.5.2.
2- A latência total só pode ser medida para a IRQ0, que é tratada com as interrupções desabilitadas, como foi visto no Capítulo 5, item 5.5.
3- A lógica simples de controle implica em menor carga computacional na execução dos marcadores.

6.4.5 Resumo das Limitações

Em resumo tem-se:

1- O método proposto só mede a latência total e a latência de hardware para a IRQ0 e, portanto, é necessário que a IRQ0 esteja ativa no sistema.

2- O algoritmo desenvolvido mede a latência de software para interrupções E/S externas que não sejam tratadas com as interrupções habilitadas.

É importante observar que primeira limitação é inerente ao método, por ser este baseado em software. Já a segunda limitação decorre da forma como foi implementado o método, ou seja, com um algoritmo de controle simples.

6.4.6 Observações Finais

Dada uma IRQa qualquer, as componentes da latência de software podem ser classificadas em dois tipos:

1- **Latências que não dependem da interrupção -** São as latências de trechos do núcleo que são executados da mesma forma por todas as interrupções. Exemplos deste tipo de latência: entrada no tratador de interrupções e *deferrable functions*.

2- **Latências que dependem da interrupção** – São as latências de trechos do núcleo que são executados de forma diferente dependendo da interrupção. Duas são as latências que podem ser encaixadas neste tipo: compartilhamento e pilha de IRQs. A primeira depende do compartilhamento ou não do sinal de interrupção (Capítulo 4, item 4.3.7) e a segunda depende, como foi mostrado, de habilitar ou não as interrupções durante o tratamento.

Com base nessa classificação tem-se:

1- Para as latências do primeiro tipo os valores obtidos para uma IRQa em particular são uma boa estimativa para qualquer interrupção E/S externa do sistema.

2- Da forma como foi desenvolvido o método, as latências do primeiro tipo são medidas corretamente mesmo usando-se uma interrupção IRQa que seja tratada com as interrupções habilitadas.

6.5 Método de Medição

Neste item, o método de medição da latência no tratamento das interrupções E/S externas é apresentado em maiores detalhes. Diferentemente do que foi feito no item 6.2, aqui a análise é de cima para baixo (*top-down*), ou seja, começando com as funcionalidades de nível mais alto.

6.5.1 Aplicativo de Gerenciamento

A figura 6.3 mostra o diagrama de blocos do Aplicativo de Gerenciamento. Pode-se ver que o programa basicamente: 1) estabelece os dados para monitoração conforme explicado no item 6.2.3, 2) controla o tempo sem medição usando um temporizador do Linux, 3) chama a Rotina PDS e 4) controla o número de conjuntos de medições.

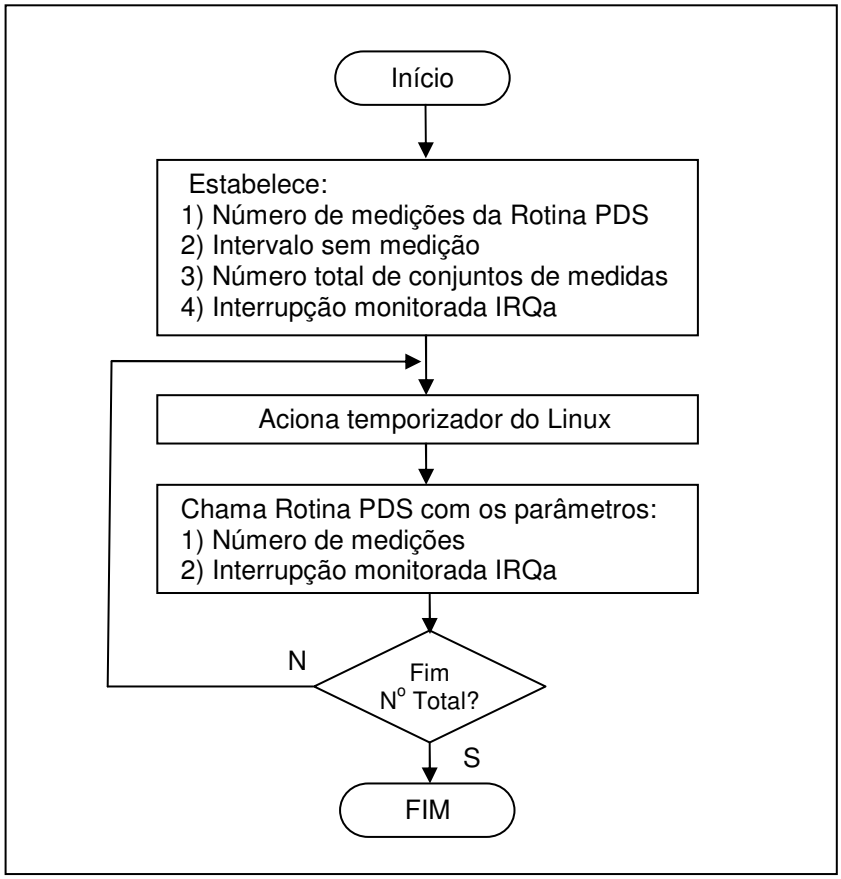

Figura 6.3 – Diagrama de blocos Aplicativo Gerenciamento

Foram desenvolvidas duas versões do Aplicativo de Gerenciamento. A primeira usa o escalonamento padrão do Linux e a segunda usa os recursos de escalonamento de tempo real do Linux 2.6.25 (ver Capítulo 5, item 5.6). O código fonte da segunda versão se encontra no Apêndice G, item G.5.

Como foi visto no item 6.4.3, o método monitora a IRQ0, que no Linux 2.6.25 é configurada com um período de aproximadamente 1 ms (ver Capítulo 5, item 5.2.2). De acordo com as referências (46) e (63), o tempo de monitoração

normalmente recomendado para um sistema operacional é da ordem de horas, o que implicaria em 3.600.000 medições para o monitoramento mínimo de uma hora. O desenvolvimento do Aplicativo de Gerenciamento visa equilibrar esse volume de dados com o tempo total de medição, como mostrado na figura 6.2 (item 6.2.3).

6.5.2 Rotina PDS

A figura 6.4 mostra parte do código da Rotina PDS. Foram excluídas, entre outras coisas, as instruções que manipulam os valores medidos pelos marcadores (ver entre as linhas 015 e 016). No final do laço foram excluídas as instruções que gravam o arquivo com os dados (ver entre as linhas 021 e 022). O código fonte completo desta rotina encontra-se no Apêndice G, item G.4.

```
001    int rotina_pds(int irq, int tot_a)
002    {
003       int ok, i, conta, lim_inf;
004       struct real_time_struct dados;

          // Inicia Medição
005       ok=syscall(188,1,irq);
006       if (ok!=0) {
007           printf("Dispositivo Não aberto \n");
008           return 1;
009       };

          // Laço de Leitura
010       for (conta=0;conta<tot_a;conta++) {
011           ok=syscall(189, (char*) &dados);

               // Verifica leitura
012           if (ok!=0)  {
013                   printf("Erro na Leitura = %d \n", ok);
014                   return 1;
015           }

               // *** Excluídas linhas que tratam os dados ***

016       };    // fim loop

          // Finaliza Medição
017       ok=syscall(188,0,irq);
018       if (ok!=0) {
019               printf("Dispositivo Não Fechado \n");
020                   return 1;
021       };
          // *** Excluídas linhas que gravam o arquivo texto ***
022       return 0;
023   }
```

Figura 6.4 – Código parcial da Rotina PDS

Analisando o código da figura pode-se ver que:

1- A linha 005 inicia as medições através da chamada ao sistema 188, desenvolvida para o método de medida. Esta chamada ao sistema libera o núcleo para realizar as medições e é descrita em mais detalhes no próximo item. Pode-se ver que por meio do 3º parâmetro (`irq`) é informado ao núcleo o número da interrupção a ser monitorada.

2- A linha 010 inicia um laço de leitura com o número de repetições determinado pelo Aplicativo de Gerenciamento (`tot_a`).

3- A linha 011 efetua uma leitura através da chamada ao sistema 189, também desenvolvida para o método de medida. Como pode ser visto, por meio desta chamada, os valores obtidos pelos marcadores são passados para o espaço de memória do processo através de uma estrutura do tipo `real_time_struct`. Esta chamada ao sistema é vista com mais detalhes no próximo item.

6.5.3 Chamadas ao Sistema

Como foi descrito no item anterior, a Rotina PDS utiliza duas chamadas ao sistema que foram especificamente desenvolvidas para o método. A documentação usada para o desenvolvimento das chamadas ao sistema no Linux encontra-se na referência (65).

A chamada ao sistema de número 188 funciona de forma semelhante ao `open` e `close` de um arquivo. As funções desta chamada ao sistema são: 1) informar ao núcleo o código da interrupção a ser monitorada, 2) determinar ao núcleo que inicie o processo de medição e 3) determinar ao núcleo que finalize o processo de medição. Essas funções são realizadas atualizando campos de uma estrutura de dados no espaço de memória do núcleo, a qual será vista no item 6.5.6.

A chamada ao sistema de número 189 realiza a transferência dos dados entre os espaços de memória e, basicamente, executa a função `le_dados` cujo diagrama de blocos é apresentado na figura 6.5.

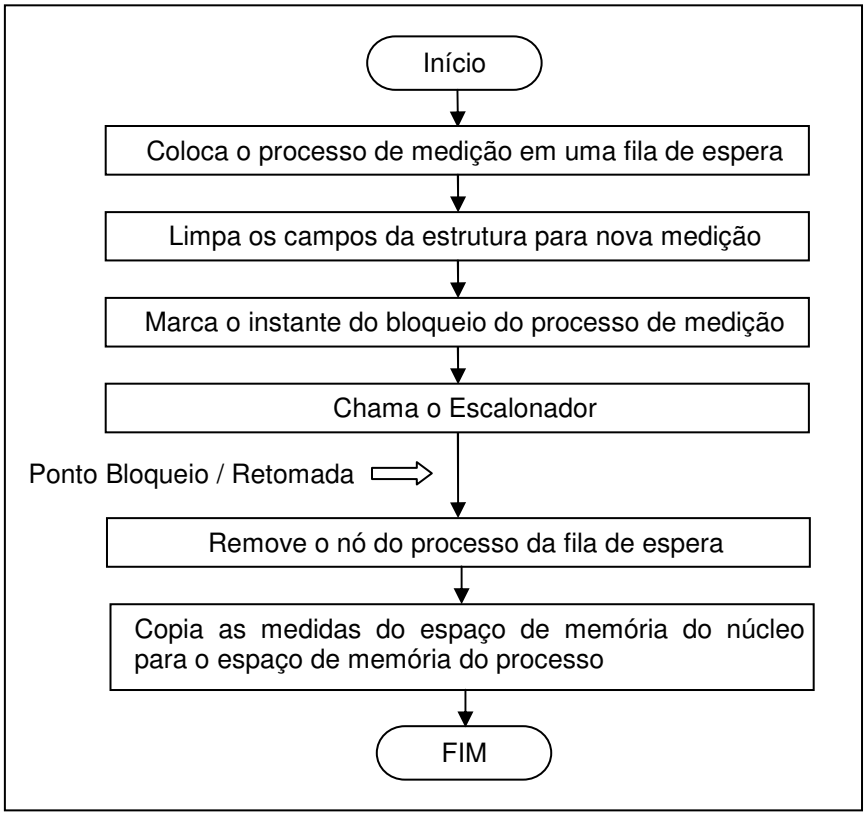

Figura 6.5 – Diagrama de blocos da função `le_dados`

Como pode ser visto no diagrama de blocos, esta função inicialmente bloqueia o processo de medição, ou seja, o processo que executa o Aplicativo de Gerenciamento. O processo fica à espera da IRQa e, retomada a execução, copia os dados do espaço de memória do núcleo para o espaço de memória do processo. Como é mostrado no próximo item, o tratador de interrupções, ao detectar a IRQa, coloca o processo no estado pronto e, posteriormente, o escalonador o passa ao estado em execução. O código fonte da função `le_dados` pode ser encontrada no Apêndice H, item H.3.11.

Juntando o descrito no item 6.5.2 com este, pode-se ver que a Rotina PDS, quando monitorando uma interrupção periódica como, por exemplo, a IRQ0, tem as características de um programa PDS, ou seja:

- Há um laço de leitura.
- O processo fica bloqueado à espera da interrupção periódica IRQ0.
- Há passagem de dados do espaço de memória do núcleo para o espaço de memória do processo, o que simula a leitura da placa adaptadora.

6.5.4 Alteração no Tratador da Interrupção

Como foi visto no item anterior, o processo de medição fica bloqueado à espera da IRQa quando a Rotina PDS é executada. Normalmente em um sistema operacional, cabe à rotina de serviço de interrupção (ISR) passar o processo para o estado pronto e sinalizar o núcleo da necessidade de chamar o escalonador. Para implementar essas funcionalidades, foi desenvolvida uma rotina, chamada `driver_medida`, a qual é acionada pelo tratador de interrupções após a rotina de serviço de interrupção da IRQa ter sido executada. A figura 6.6 mostra o diagrama de blocos dessa rotina, cujo código fonte pode ser visto no Apêndice H, item H.3.12.

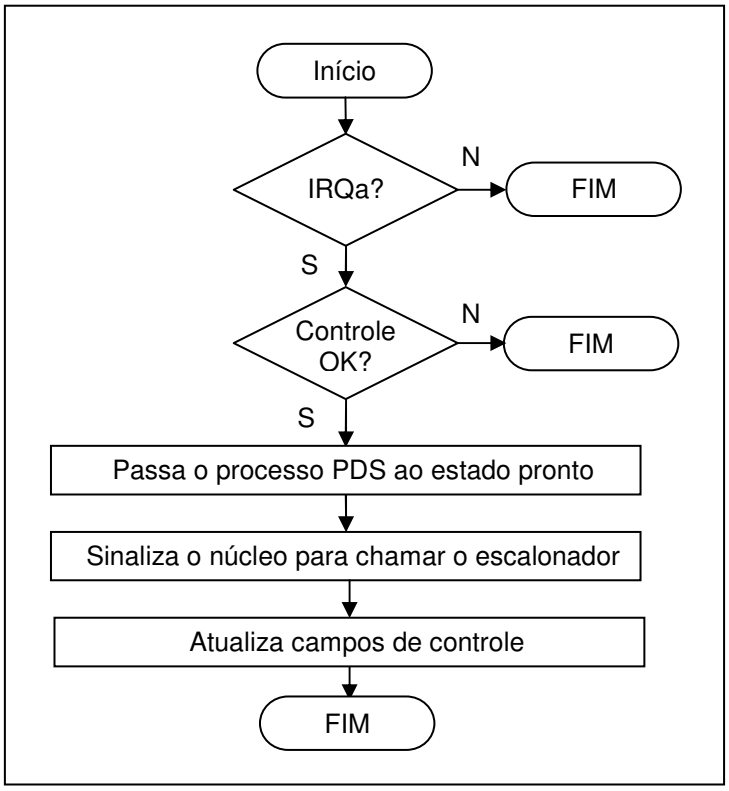

Figura 6.6 – Diagrama de blocos da rotina `driver_medida`

No diagrama de blocos da figura 6.6 pode-se ver que a rotina: 1) confere se a interrupção sendo tratada é a IRQa, 2) verifica alguns campos de controle adicionais, 3) muda o estado do processo de bloqueado para pronto, 4) sinaliza ao núcleo que é necessário chamar o escalonador e 5) atualiza campos de controle.

Finalizado o tratamento da interrupção IRQa, o núcleo chama o escalonador, que tem a função de mudar o estado do processo de medição de pronto para em execução.

6.5.5 Marcadores

Foram desenvolvidos vários tipos de marcadores, que são utilizados conforme o trecho do núcleo a ser medido. A seguir são apresentados os diferentes tipos de marcadores e os itens no Apêndice H onde podem ser vistos os respectivos códigos fontes:

- Marcador do PIT – Utilizado para medir a latência de hardware (item H.3.2).
- Marcador TSC – Utilizado em vários trechos do núcleo (item H.3.3).
- Marcador Entrada ao Escalonador – Utilizado para medir o instante de entrada no escalonador (item H.3.4).
- Marcador Saída do Escalonador – Utilizado para medir o instante de saída do escalonador (item H.3.5).
- Marcador Troca de Contexto – Utilizado para medir o instante aproximado em que se inicia a troca de contexto (item H.3.6).

O Marcador do PIT, como o próprio nome diz, utiliza o *Programmable Interval Timer*[3] (PIT) para medir o instante de tempo. Todos os outros tipos de marcadores utilizam o *Time Stamp Counter*[4] (TSC) para medir o instante de tempo e a diferença entre eles está na lógica de controle.

[3] O *Programmable Interval Timer* (PIT) é tratado no Capítulo 5, item 5.2.2.
[4] O *Time Stamp Counter* (TSC) é tratado no Capítulo 5, item 5.2.5.

A título de exemplo, a figura 6.7 mostra o código do Marcador TSC que é o mais utilizado.

```
001    void mede_tsc(int etapa, int codigo) {

           // Controle
002        if (!rt_st.medida_on) return;
003        if (codigo!=rt_st.irq_cod) return;
004        if rs_st.flags_medida[etapa] return;

           // Medida
005        rt_st.flags_medida[etapa]=true;
006        rt_st.medida[etapa]=get_cycles();
007        return;
}
```

Figura 6.7 – Código fonte do Marcador TSC

Na figura 6.7 pode-se ver:

1- Os parâmetros de entrada são os números da etapa e da interrupção que está sendo monitorada. O número da etapa permite usar a mesma função em vários pontos do tratador de interrupções.
2- Um bloco (linhas 002 a 005) que implementa a lógica de controle.
3- A leitura do registrador `tsc` usado a função `get_cycles()` (linha 006).

6.5.6 **Estrutura**: `real_time_struct`

No Linux 2.6.25 foi incluída uma estrutura chamada `real_time_struct` que tem como finalidade principal armazenar, no espaço de memória do núcleo, os valores de tempo obtidos pelos marcadores.

A estrutura também possui alguns campos para controle e para armazenar informações sobre o hardware. Explicados a seguir, os principais campos podem ser vistos na tabela 6.1 e a estrutura completa, no Apêndice H, item H.2.2.

Tabela 6.1 – Descrição dos campos da estrutura `real_time_struct`

Campo	Descrição
`medida_on`	Indica se o núcleo está ou não em processo de medição
`irq_cod`	Número da IRQ que está sendo monitorada ou IRQa
`flag_medida_pit`	Indica se o PIT já foi medido
`medida_pit`	Medida efetuada pelo marcador que usa o PIT
`flags_medida[20]`	Indica se uma medida foi ou não efetuada
`medida[20]`	Medidas efetuadas pelos marcadores que usam o TSC
`irq_aninhadas`	Número de interrupções IRQa tratadas durante a medida
`irq_handle`	Informa se o processo foi para o estado pronto

O campo `medida_on` serve para habilitar e desabilitar a medição.

O campo `irq_cod` serve para determinar a interrupção que está sendo analisada. Embora o método proposto meça a latência total apenas para a IRQ0, ele pode ser usado com outras interrupções E/S externas.

O marcador que lê o PIT armazena o valor medido no campo `medida_pit` e usa o campo `flag_media_pit` para controlar a atualização da medida.

Os marcadores que lêem o registrador TSC são numerados e armazenam os valores medidos no vetor `medida` e usam o vetor `flags_medida` para controle da atualização.

O campo `irq_aninhadas` é usado para determinar se a latência da IRQ0 foi medida completamente antes da chegada de outra IRQ0. Pensando em termos de SO-PDS, o campo sinaliza que o sistema está subamostrando[5].

O campo `flag_medida_pit`, o vetor `flags_medida` e o campo `irq_handle` são necessários já que, em havendo interrupções aninhadas ou subamostragem, alguns trechos do tratador de interrupções são executados mais de uma vez. O campo `flag_medida_pit` e o vetor `flags_medida` servem para evitar que um marcador faça uma medida incorreta. O campo `irq_handle` tem a

[5] O campo `irq_aninhadas` diferente de zero é uma condição suficiente para a subamostragem, mas não é uma condição necessária porque a atualização deste campo não leva em consideração o tempo de processamento da amostra.

função de controlar a mudança de estado do processo no tratador de interrupções (ver item 6.5.4).

Durante a carga do sistema operacional a estrutura é carregada com os valores iniciais. A rotina que efetua a inicialização é chamada na rotina `start_kernel` imediatamente após a rotina `early_init_irq_lock_class`. O campo `medida_on` é carregado de forma que os marcadores não efetuem nenhuma medição até serem acionados pela Rotina PDS (item 6.5.2). O código fonte desta rotina, chamada `boot_medida()`, pode ser visto no Apêndice H, item H.3.9.

6.5.7 Colocação dos Marcadores

Os marcadores foram colocados no núcleo do Linux 2.6.25 de forma a medir as seguintes componentes da latência, conforme o estabelecido pelo modelo: 1) hardware, 2) entrada no tratador de interrupções, 3) compartilhamento, 4) saída do tratador de interrupções, 5) pilha de IRQs, 6) *deferrable functions*, 7) preempção do núcleo desabilitada, 8) escalonamento e 9) troca de contexto.

6.6 Metodologia Usada

Foram usadas duas máquinas:

1- Pentium III, relógio de freqüência 1GHz e 1GB de memória RAM.
2- Pentium IV, relógio de freqüência 2GHz e 2GB de memória RAM.

Na primeira máquina foram feitos dois testes usando a IRQ0, mas com metodologias diferentes. Os dados obtidos para o primeiro teste e a metodologia usada são apresentados neste capítulo e para o segundo teste no Apêndice K. Na segunda máquina foi efetuado um teste com a IRQ15 e não foram obtidas todas as componentes da latência. Os resultados para essa máquina estão no Apêndice I.

A freqüência usada para a IRQ0 foi de 1.000 Hz (padrão do Linux 2.6.25), portanto, neste caso simula-se um Aplicativo PDS que lê e trata a amostra com um período de amostragem de 1 ms.

6.6.1 Tipos de Medidas

Em todos os testes as medidas foram obtidas de três formas diferentes como é mostrado a seguir com as respectivas nomenclaturas:

1- **"Padrão - Sem Carga"** – Nesta medição foi usado o núcleo na configuração padrão para os parâmetros de compilação[6] e para os programas carregados em segundo plano (*daemons*). O Linux foi carregado no modo linha de comando (*shell*) e o Aplicativo de Gerenciamento foi executado.

2- **"Padrão - Com Carga"** – Nesta medição foi usado o núcleo na configuração padrão para os parâmetros de compilação e para os programas carregados em segundo plano (*daemons*). O Linux foi carregado em modo gráfico e o Aplicativo de Gerenciamento foi executado simultaneamente com programas de carga.

3- **"Preempção – Com Carga"** – Nesta medição foram usados os recursos disponíveis no Linux 2.5.25 para tempo real. O núcleo foi configurado para ter preempção do núcleo, mas os programas carregados no segundo plano (*daemons*) foram mantidos na configuração padrão. O Linux foi carregado em modo gráfico e o Aplicativo de Gerenciamento foi executado em tempo real[7] (Classe `SCHED_FIFO`) com prioridade máxima, simultaneamente com programas de carga.

Neste capítulo, foram dois os programas usados para carga do sistema:

- Um programa que fica em laço infinito calculando a inversa de uma matriz. Uma vez obtida a matriz inversa, há um intervalo de tempo, controlado por um temporizador, antes de iniciar uma nova repetição.

[6] Na sua configuração padrão de parâmetros de compilação, o Linux não tem a preempção do núcleo.
[7] Os recursos de escalonamento de tempo real do Linux são tratados no Capítulo 5, item 5.6.

- Um programa que fica em laço infinito gravando, lendo e excluindo um arquivo texto. Entre as três operações há um intervalo de tempo controlado por um temporizador.

O objetivo de utilizar diversos temporizadores no programas de carga é ter várias *deferrable functions* no sistema.

6.6.2 Volume de Dados

Em todas as medições realizadas, a configuração do Aplicativo de Gerenciamento foi efetuada da seguinte forma:

- Conjuntos de dados = 200
- Amostras por conjunto = 100
- Intervalo entre conjuntos = 120 s

Desta forma foram obtidas 20.000 medidas em um intervalo de 6 horas e 40 minutos aproximadamente.

6.7 Resultados

A seguir são apresentados os gráficos da distribuição acumulada de freqüência ou CDF (*Cumulative Distribution Function*) e algumas estatísticas para os resultados obtidos. Para algumas componentes também é apresentado o gráfico di-log de 1-CDF, com o objetivo de avaliar o comportamento da CDF para os valores próximos a 1 (um)[8]. Em alguns subitens foi usada a análise de variância para testar a igualdade das médias, técnica tratada na referência (66).

Para cada componente medida é apresentada uma pequena análise dos dados obtidos, mas ressalta-se que uma análise detalhada dos resultados obtidos extrapola o escopo deste trabalho.

[8] Observa-se que o ponto (V_{MAX} , 1) da CDF no gráfico di-log de 1-CDF corresponde a (log V_{MAX} , -∞). Portanto o gráfico di-log de 1-CDF é truncado no segundo maior valor obtido.

6.7.1 Latência de Hardware

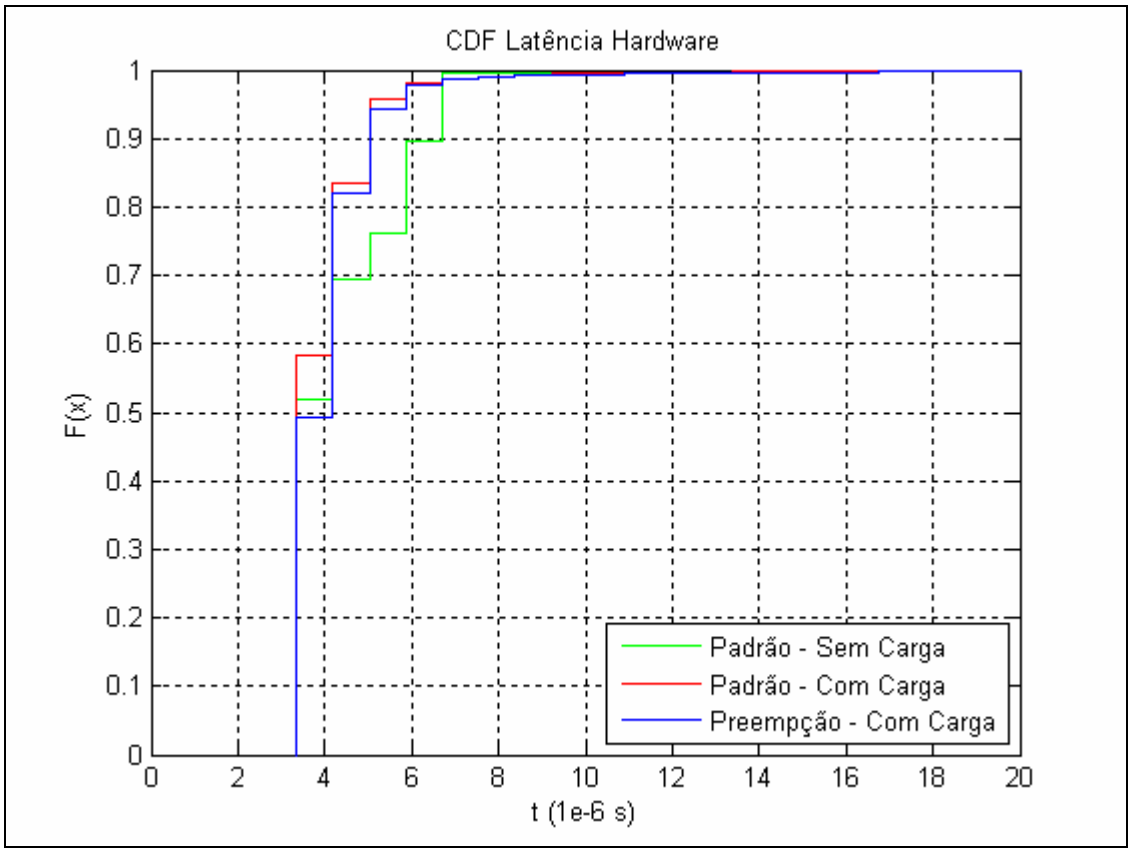

Figura 6.8 – CDF da latência de hardware
Obs.: Gráfico da CDF truncado

Tabela 6.2 – Estatísticas da latência de hardware

	Padrão Sem Carga (µs)	Padrão Com Carga (µs)	Preempção Com Carga (µs)
Média	4,38	3,99	4,11
Máximo	52,0	181,0	137,4
Mínimo	3,35	3,35	3,35
Mediana	3,35	3,35	4,19

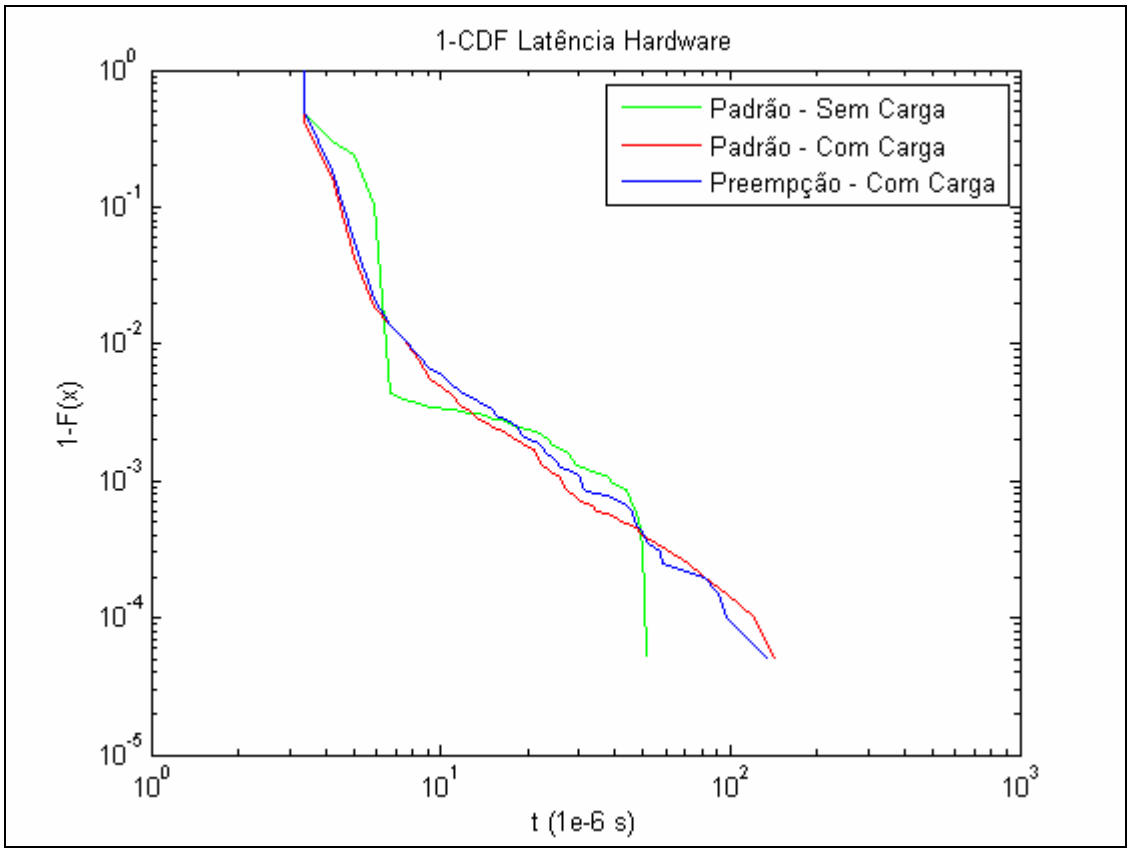

Figura 6.9 – Gráfico (1-CDF) da latência de hardware

Análise:

Esta latência é medida com o PIT que tem uma freqüência de operação de 1.193.182 Hz e, portanto, uma resolução de 0,8381 μs (Capítulo 5, item 5.2.2). Pode-se assumir a precisão da medida como sendo metade da resolução. (67) Portanto, tem-se que a precisão dos valores medidos é de 0,42 μs.

Esperar-se-ia que o sistema com carga apresentasse uma latência de hardware média maior do que sem carga. No entanto, decorre do colocado acima que os valores médios, dentro da precisão utilizada, podem ser considerados iguais.

Uma conclusão do exposto é que o programa intensivo em E/S não carregou suficientemente o sistema, considerando-se a precisão da medida e a capacidade de processamento.

Foi realizado um teste usando a placa de rede – ver Apêndice K – com a finalidade de aumentar o número de interrupções E/S sinalizadas no sistema.

6.7.2 Latência de Entrada no Tratador de Interrupções

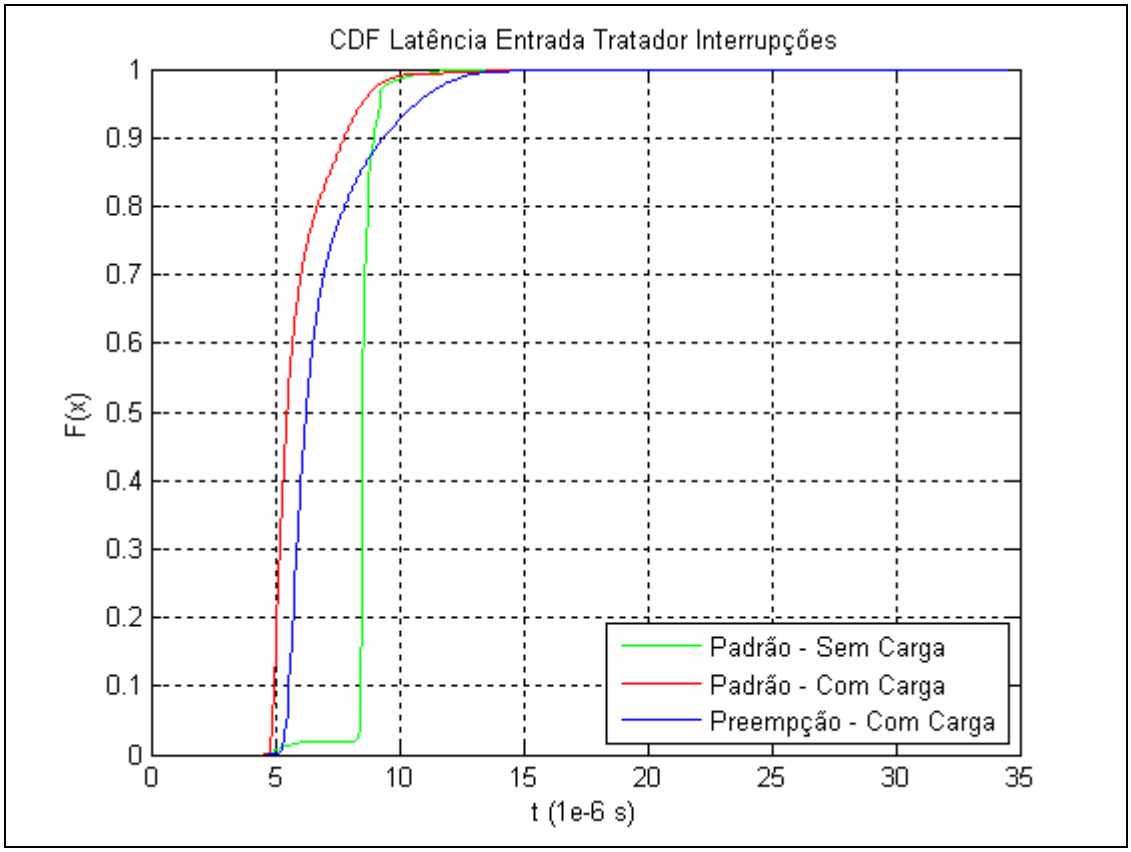

Figura 6.10 – CDF da latência de entrada no tratador de interrupções

Tabela 6.3 – Estatísticas da latência de entrada no tratador de interrupções

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	8,59	5,96	6,89
Máximo	13,47	19,34	34,59
Mínimo	4,50	4,52	4,64
Mediana	8,53	5,48	6,25

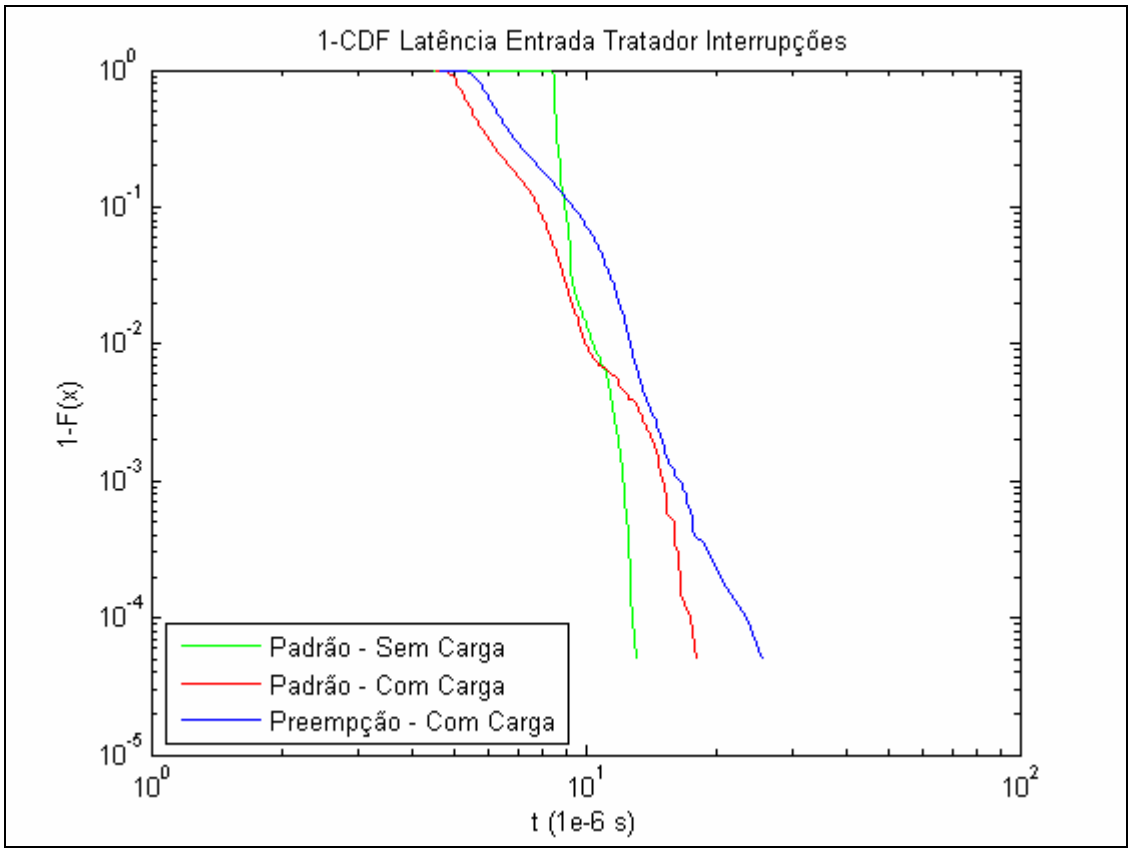

Figura 6.11 – Gráfico (1-CDF) da latência de entrada no tratador de interrupções

Análise:

Inicialmente, salienta-se que uma explicação detalhada dos dados obtidos requer um estudo mais completo, o que foge ao escopo deste trabalho.

Seria de se esperar uma latência média menor para o sistema sem carga, o que não é o observado nos dados obtidos. Uma tese para explicar esta diferença é que no teste sem carga, o sistema está na maioria das vezes (~ 98%) em Modo Usuário quando a IRQa é sinalizada e, nestes casos, o núcleo cria a pilha de IRQ, o que demanda tempo de processamento (Capítulo 4, item 4.3.6.1).

Os maiores valores para as estatísticas foram obtidos no teste “Preempção – Com Carga”. Isto pode ser explicado como segue: por suportar sistemas multiprocessadores, o Linux 2.6.25 utiliza, ao atualizar as estruturas do núcleo, mecanismos de sincronização, que dependem do núcleo ter ou não preempção, mesmo com as interrupções desabilitadas. No Linux, o núcleo com preempção tem uma sobrecarga de processamento se comparado ao núcleo sem preempção. (24)

6.7.3 Latência de Compartilhamento

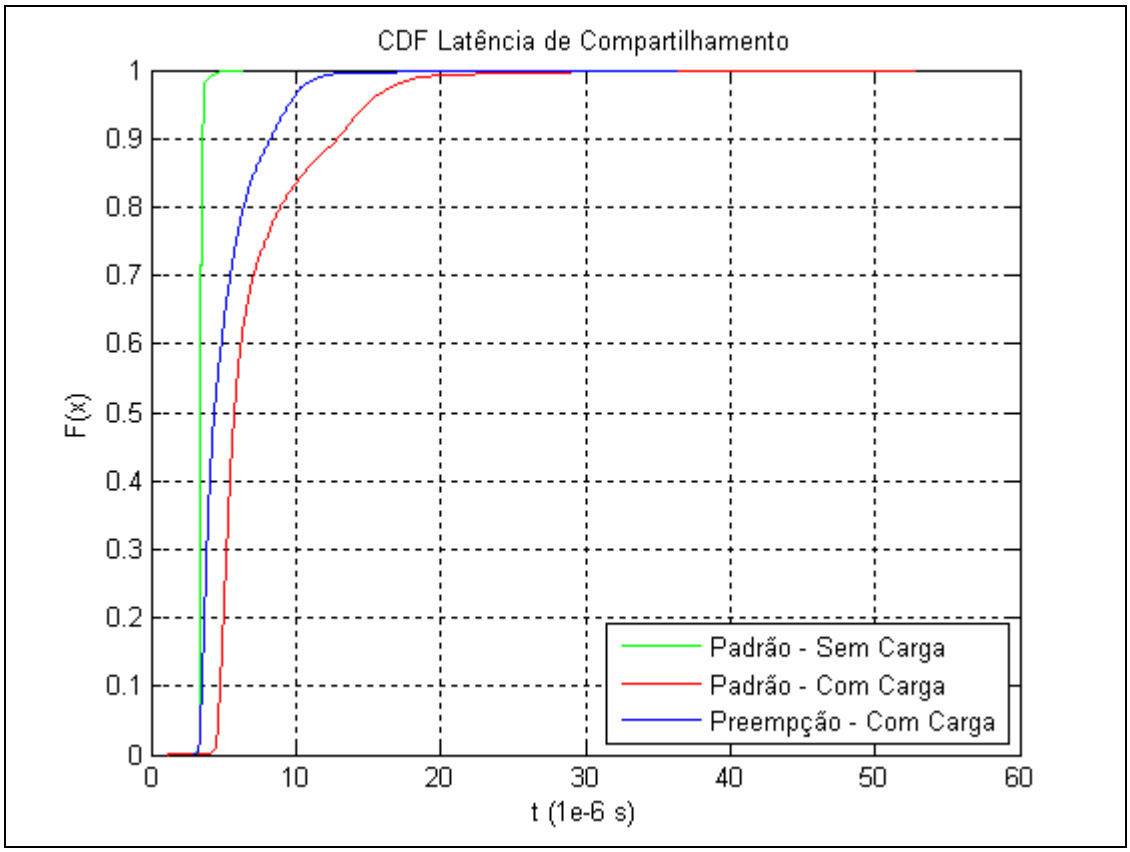

Figura 6.12 – CDF da latência de compartilhamento

Tabela 6.4 – Estatísticas da latência de compartilhamento

	Padrão Sem Carga (µs)	Padrão Com Carga (µs)	Preempção Com Carga (µs)
Média	3,46	7,29	5,19
Máximo	6,40	52,9	36,4
Mínimo	1,10	1,14	2,97
Mediana	3,43	5,75	4,37

Análise:

Como a interrupção IRQ0 não é compartilhada seriam de se esperar valores próximos para os três testes[9]. Uma explicação para a menor média no teste sem carga seria a maior probabilidade do descritor da IRQ0 e de outros dados usados no tratamento desta interrupção estarem em memória cache. O trecho do núcleo medido basicamente percorre uma lista ligada, essa simplicidade do código faz com o maior processamento no núcleo com preempção não tenha influência.

[9] A análise de variância rejeita a igualdade das três médias ao nível de 5% de significância.

6.7.4 Latência de Saída do Tratador de Interrupções

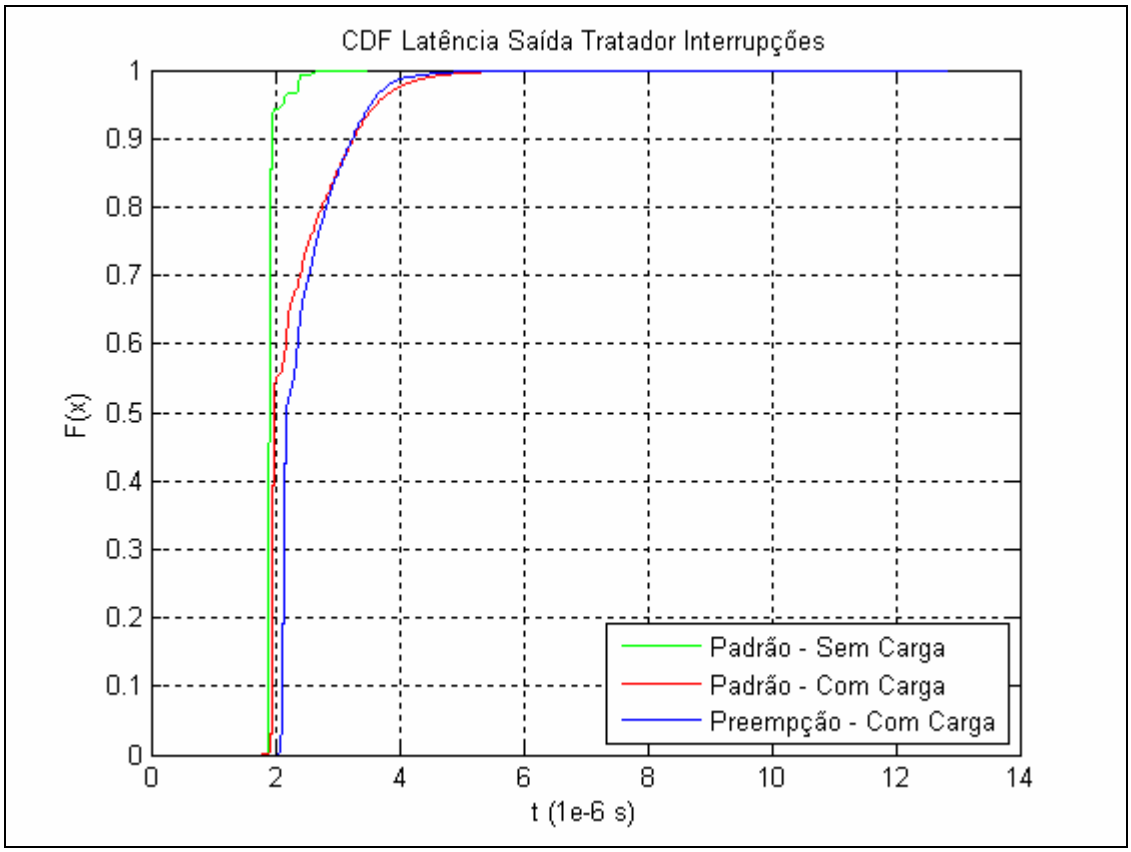

Figura 6.13 – CDF da latência de saída do tratador de interrupções

Tabela 6.5 – Estatísticas da latência de saída do tratador de interrupções

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	1,94	2,35	2,48
Máximo	3,48	8,10	12,84
Mínimo	1,83	1,79	2,02
Mediana	1,91	1,98	2,18

Análise:

Uma explicação para o menor valor médio no teste sem carga seria a coincidência de cache (*cache hit*), como colocado no item anterior.

Os maiores valores para as estatísticas no teste “Preempção – Com Carga” podem ser explicados pelo maior processamento no núcleo com preempção, já que, neste trecho medido, várias rotinas do núcleo são acionadas. (24)

6.7.5 Latência de Pilha de IRQs

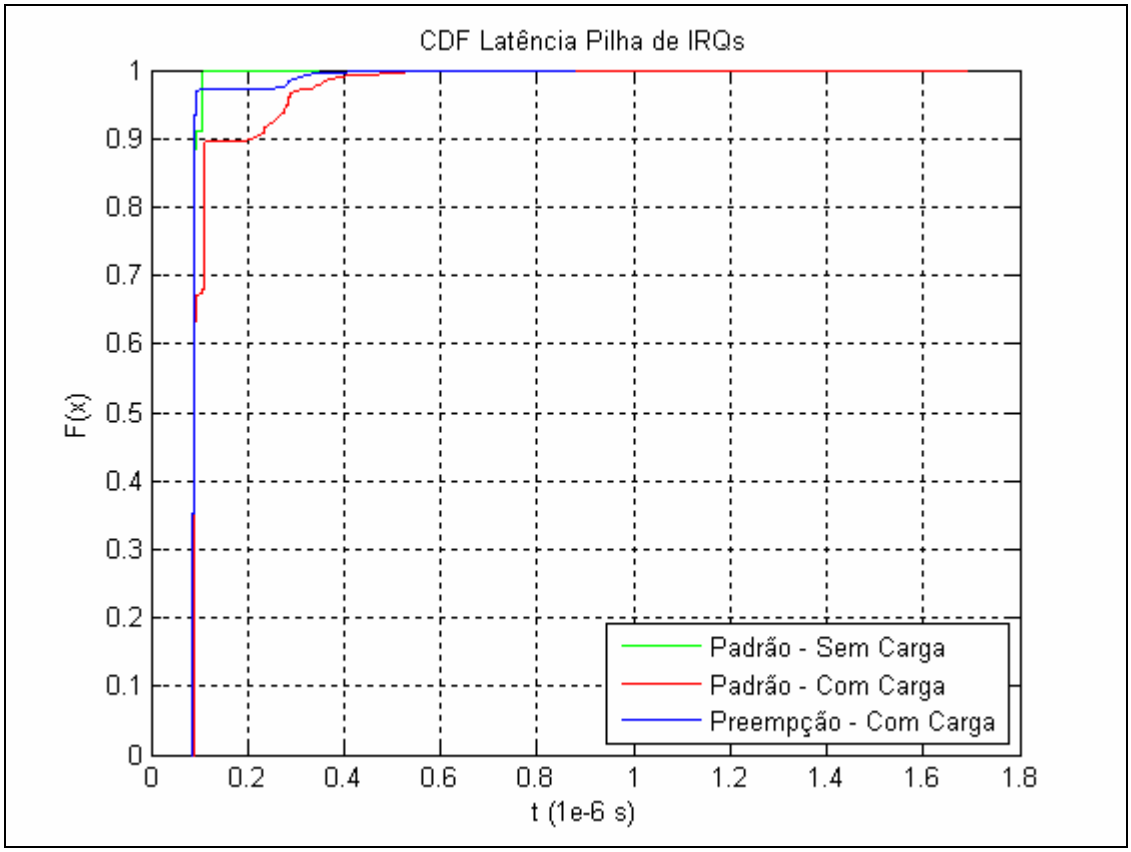

Figura 6.14 – CDF da latência de pilha de IRQs

Tabela 6.6 – Estatísticas da latência de pilha de IRQs

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	0,0926	0,118	0,0956
Máximo	0,349	1,70	0,879
Mínimo	0,0912	0,0912	0,0855
Mediana	0,0912	0,0912	0,0912

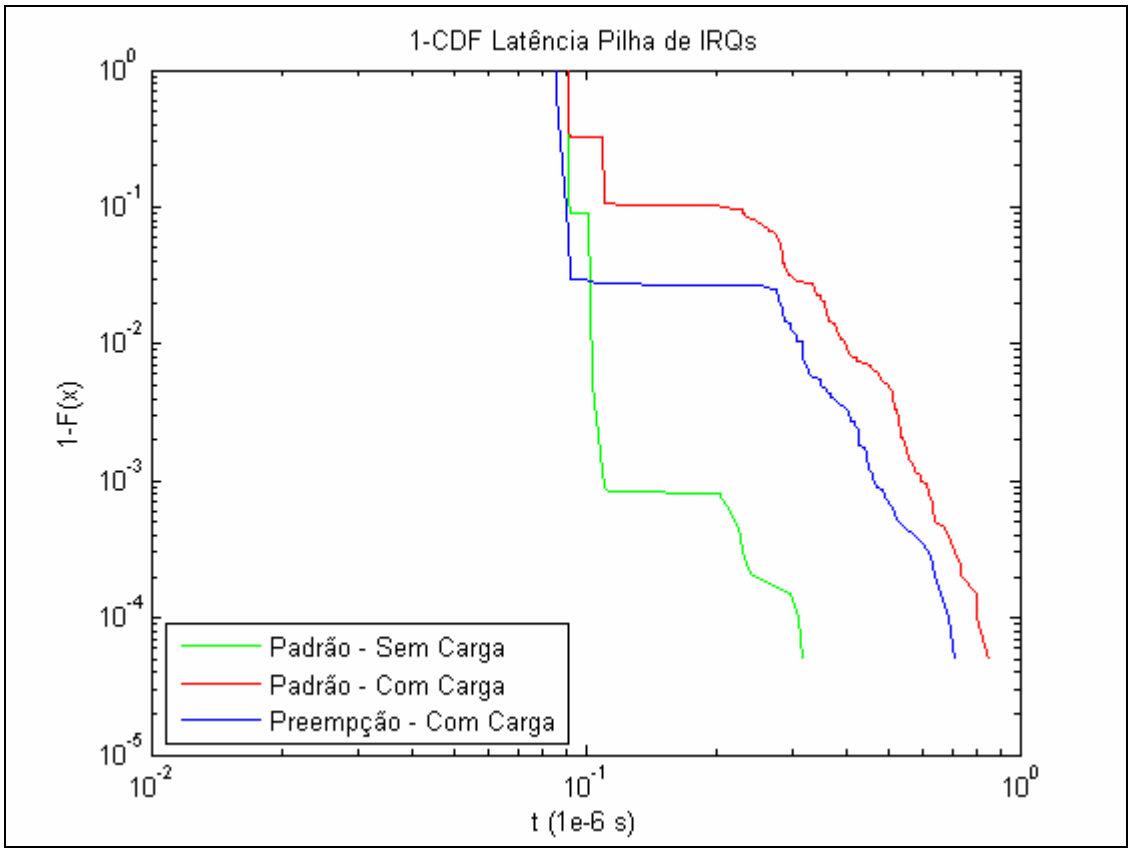

Figura 6.15 – Gráfico (1-CDF) da latência de pilha de IRQs

Análise:

Analisando o código fonte do núcleo, pode-se ver que, mesmo sem o aninhamento de interrupções, esta latência não é nula, já que são executadas instruções para verificar o estado da pilha. No sistema usado, constatou-se que a IRQ14, sinalizada pelo disco rígido, era tratada com as interrupções habilitadas e, portanto, poderiam ocorrer aninhamentos de interrupções durante os testes.

No caso em que há uma única interrupção IRQn na pilha quando a IRQa é sinalizada, esta latência mede no mínimo o tempo de saída do tratador de interrupções dessa IRQn. Comparando os dados com os do item 6.7.4, pode-se ver que os valores obtidos neste item são bem menores.

Esses valores menores mostram que não houve aninhamento de interrupções e confirmam a suposição de que o programa intensivo em E/S não carregou suficientemente o sistema (ver item 6.7.1 e Apêndice K). Nesse cenário a coincidência de cache explicaria o menor valor médio para o teste sem carga.

6.7.6 Latência de *Deferrable Functions*

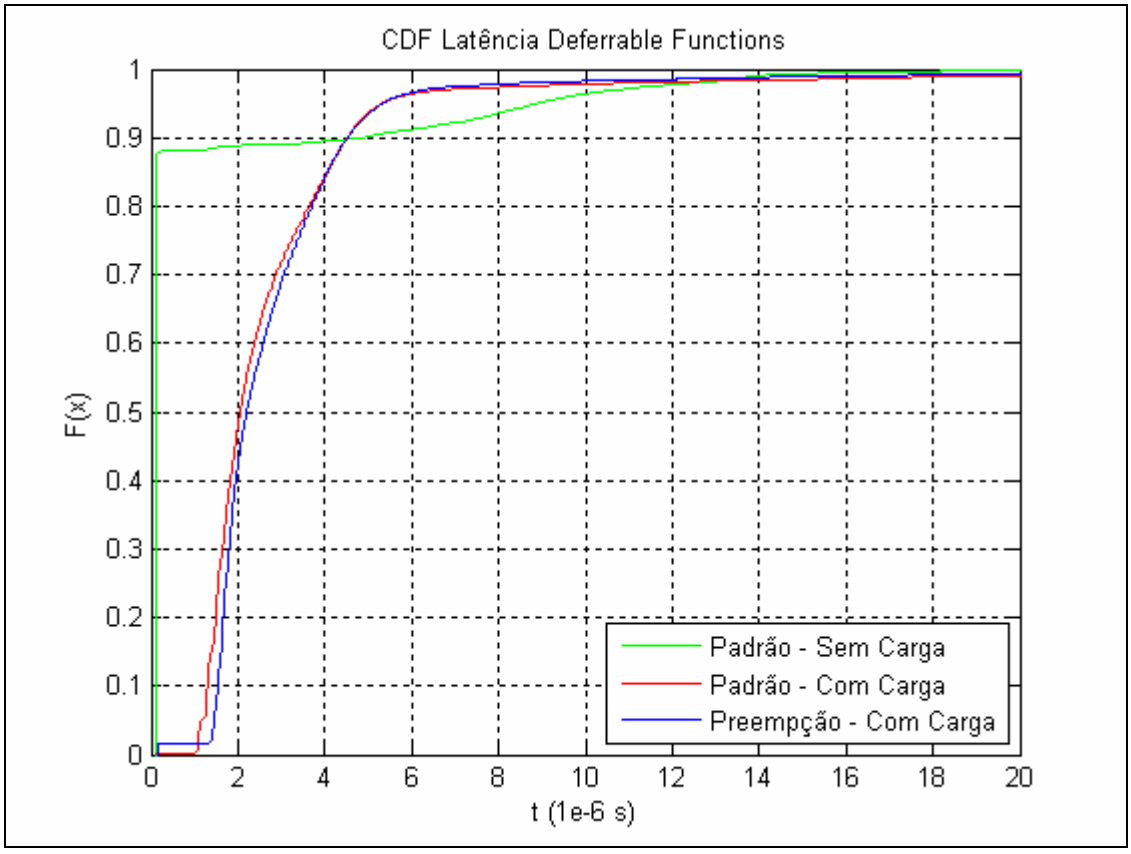

Figura 6.16 – CDF da latência de d*eferrable functions*
Obs.: Gráfico da CDF truncado

Tabela 6.7 – Estatísticas da latência de *deferrable functions*

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	1,14	2,89	2,92
Máximo	54,5	582,1	143,2
Mínimo	0,115	0,113	0,138
Mediana	0,123	2,05	2,20

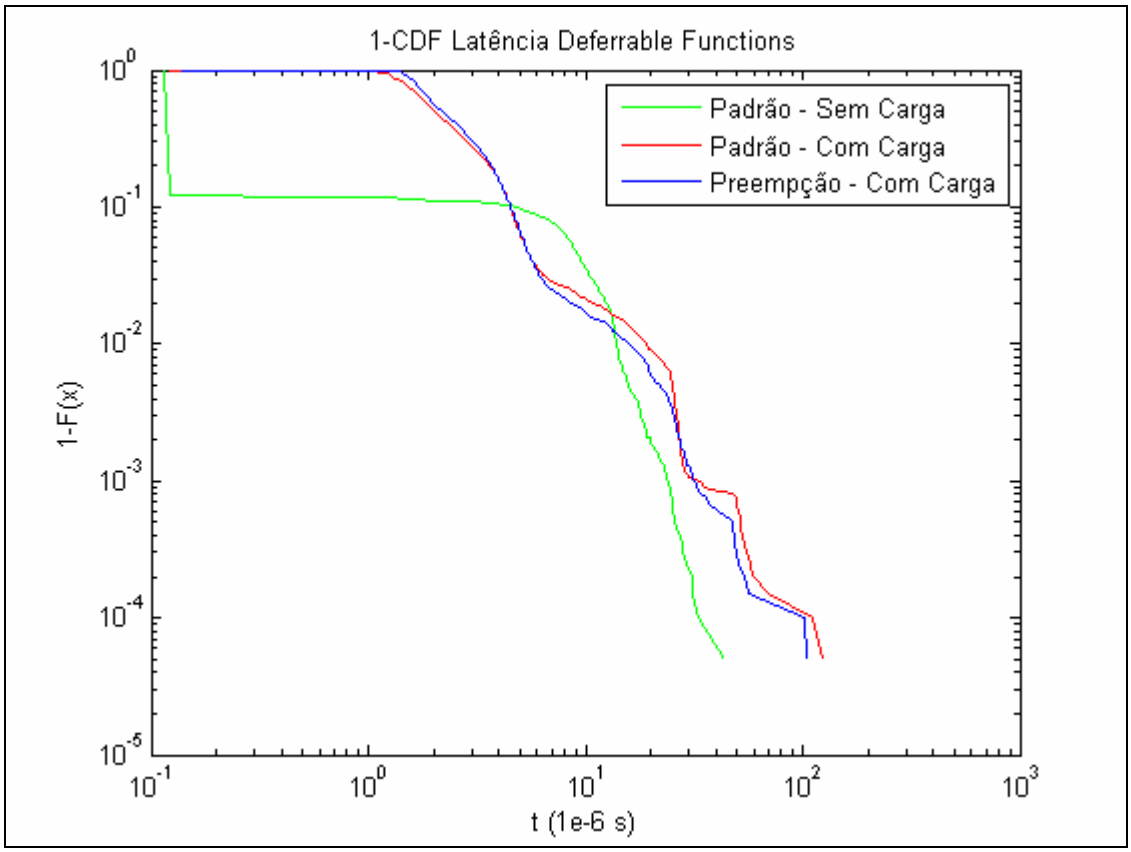

Figura 6.17 – Gráfico (1-CDF) da latência de d*eferrable functions*

Análise:

Pode-se observar que, mesmo no teste sem carga, *deferrable functions* são executadas no sistema.

Nos testes com carga, valores próximos[10] são esperados, pois nesses testes a principal fonte de *deferrable functions* são os temporizadores dos programas de carga (ver item 6.6.1).

É interessante observar que em um SO-PDS está latência é nula por conta do 13º Requisito, colocado no Capítulo 4, item 4.5.1.3.

[10] A análise de variância rejeita a igualdade das duas médias ao nível de 5% de significância.

6.7.7 Latência de Preempção do Núcleo Desabilitada

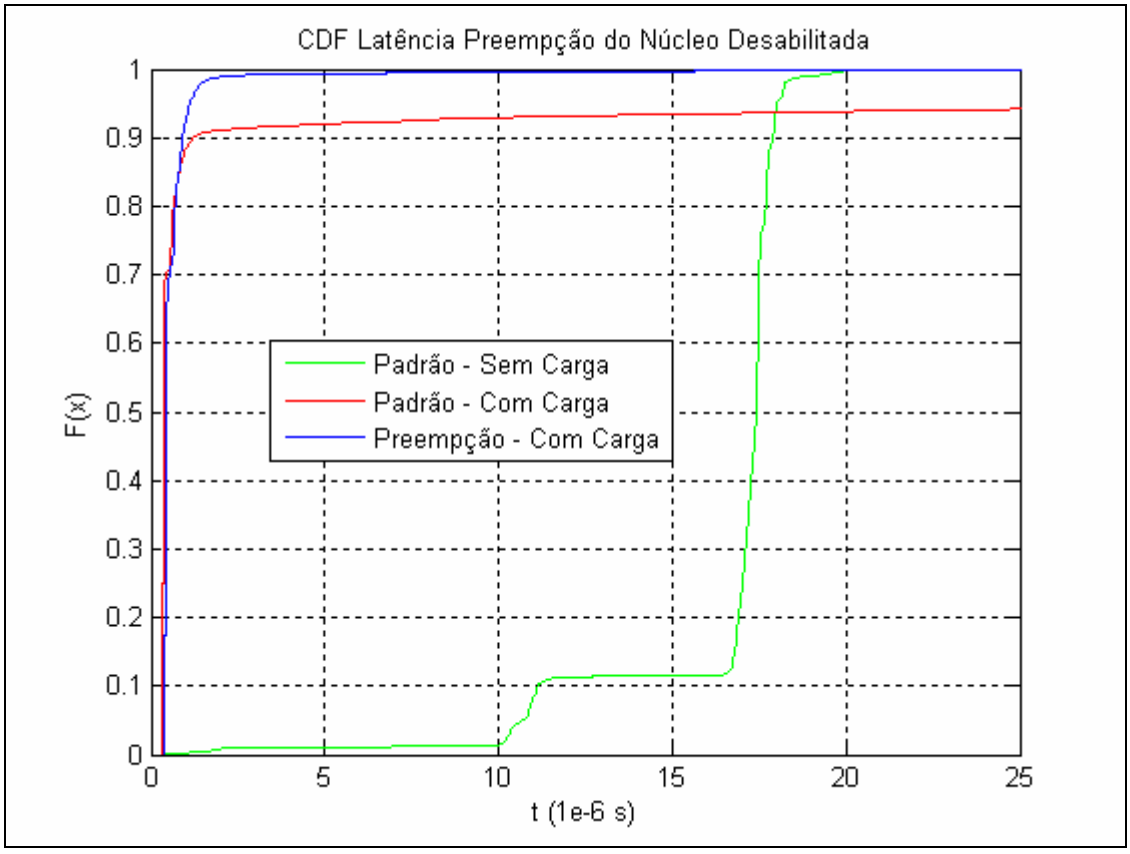

Figura 6.18 – CDF da latência de preempção do núcleo desabilitada
Obs.: Gráfico da CDF truncado

Tabela 6.8 – Estatísticas da latência de preempção do núcleo desabilitada

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	16,65	31,13	0,769
Máximo	664,7	1,42e+3	345,5
Mínimo	0,314	0,241	0,332
Mediana	17,42	0,369	0,433

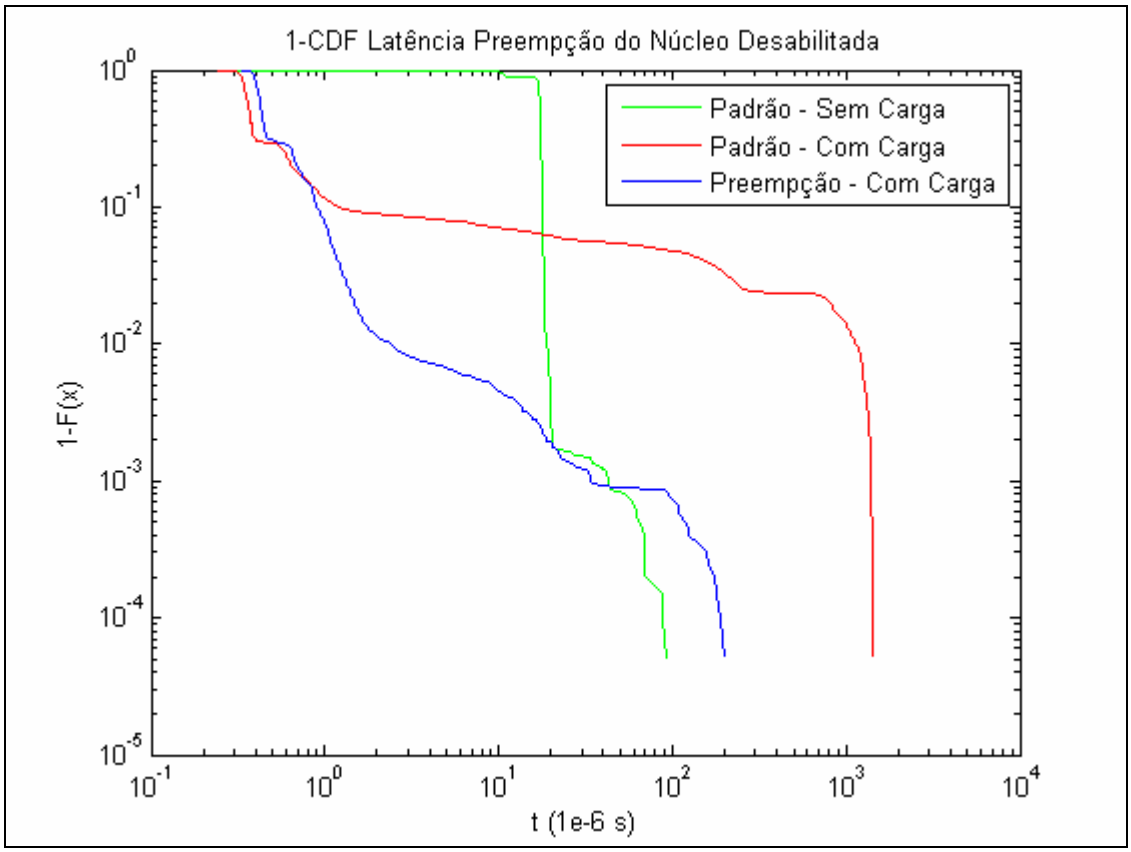

Figura 6.19 – Gráfico (1-CDF) da latência de preempção do núcleo desabilitada

Análise:

Nos testes com carga, a sobreposição inicial dos dois gráficos é explicada pelo fato de que, se IRQa for sinalizada quando o processo em execução está em Modo Usuário, a latência é baixa e independe do núcleo ter ou não preempção.

Como esperado, o núcleo com preempção tem o menor valor médio. Além disso, o teste “Padrão – Sem Carga” tem uma latência média ~ 21,6 vezes maior do que o teste “Preempção – Com carga”. Os dados obtidos justificam numericamente:

1- O 17º Requisito que faz com que esta latência seja nula em um SO-PDS (Capítulo 4, item 4.5.4).

2- Os esforços dos desenvolvedores do Linux em tempo real para fazer com que o núcleo seja totalmente com preempção (Capítulo 3, item 3.10.4).

Uma explicação para o comportamento da CDF para o teste sem carga extrapola os objetivos propostos para este texto.

6.7.8 Latência de Escalonamento

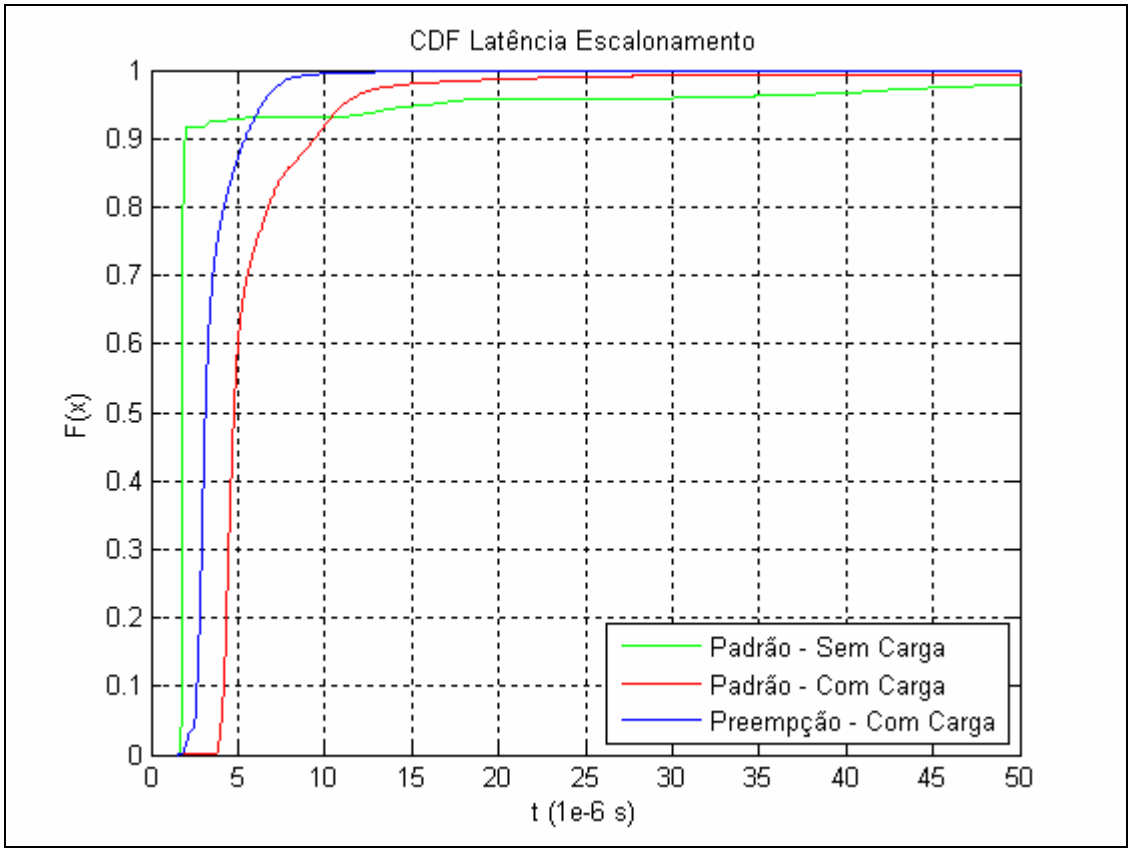

Figura 6.20 – CDF da latência de escalonamento
Obs.: Gráfico da CDF truncado

Tabela 6.9 – Estatísticas da latência de escalonamento

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	6,75	7,85	3,66
Máximo	531,0	1,08e+3	123,4
Mínimo	1,54	1,91	1,58
Mediana	1,81	4,77	3,12

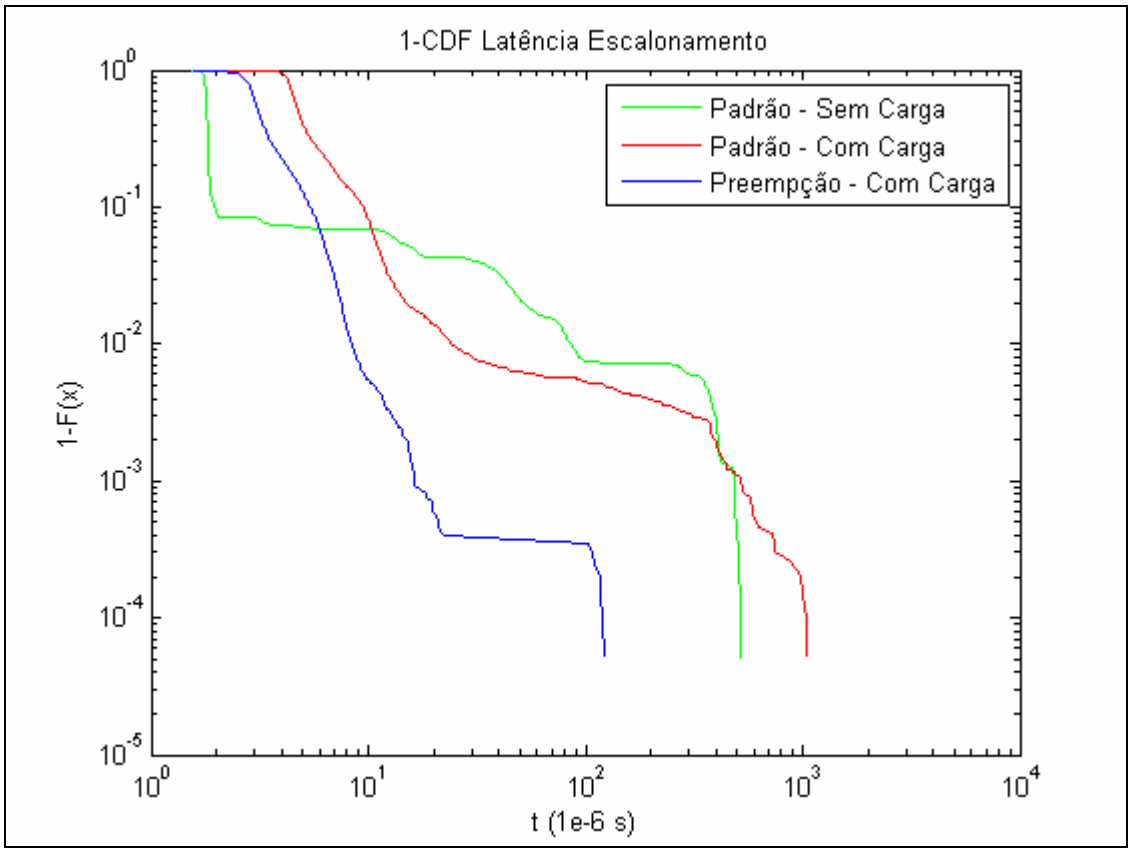

Figura 6.21 – Gráfico (1-CDF) da latência de escalonamento

Análise:

A menor latência média foi obtida no teste “Preempção – Com Carga”. Isto se justifica porque, nesse teste, o processo de medição está em modo tempo real (Classe `SCHED_FIFO`) e tem a maior prioridade no sistema.

Como esperado, a maior latência média foi obtida no teste “Padrão – Com Carga”. Isto se justifica porque, nesse teste, o processo de medição está no modo e prioridade padrão do Linux e, também, porque o sistema está com carga.

Os valores maiores de latência no teste “Padrão – Sem Carga” são explicados pela existência de programas carregados em segundo plano (*daemons*) sendo executados com prioridade maior ou igual à prioridade do processo de medição.

6.7.9 Latência de Troca de Contexto

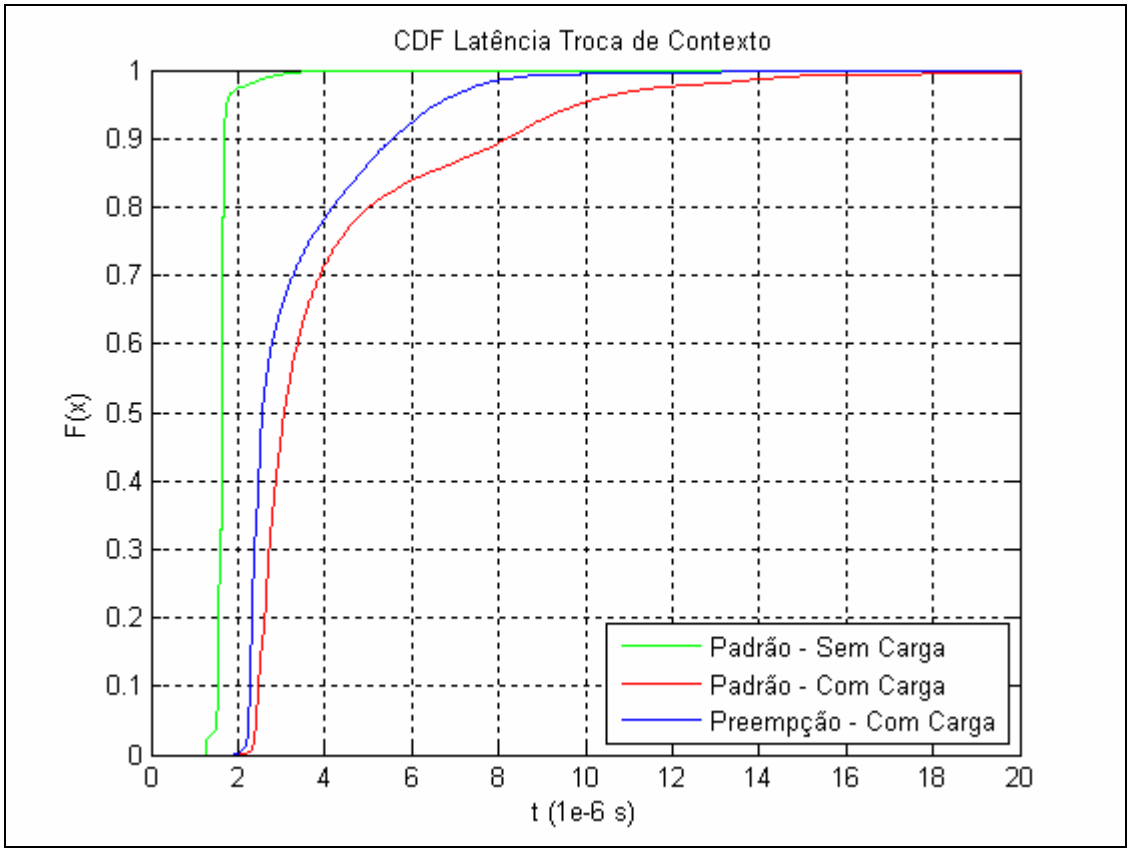

Figura 6.22 – CDF da latência de troca de contexto
Obs.: Gráfico da CDF truncado

Tabela 6.10 – Estatísticas da latência de troca de contexto

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	1,70	4,26	3,33
Máximo	157,25	170,6	154,8
Mínimo	1,21	1,94	1,92
Mediana	1,63	3,06	2,56

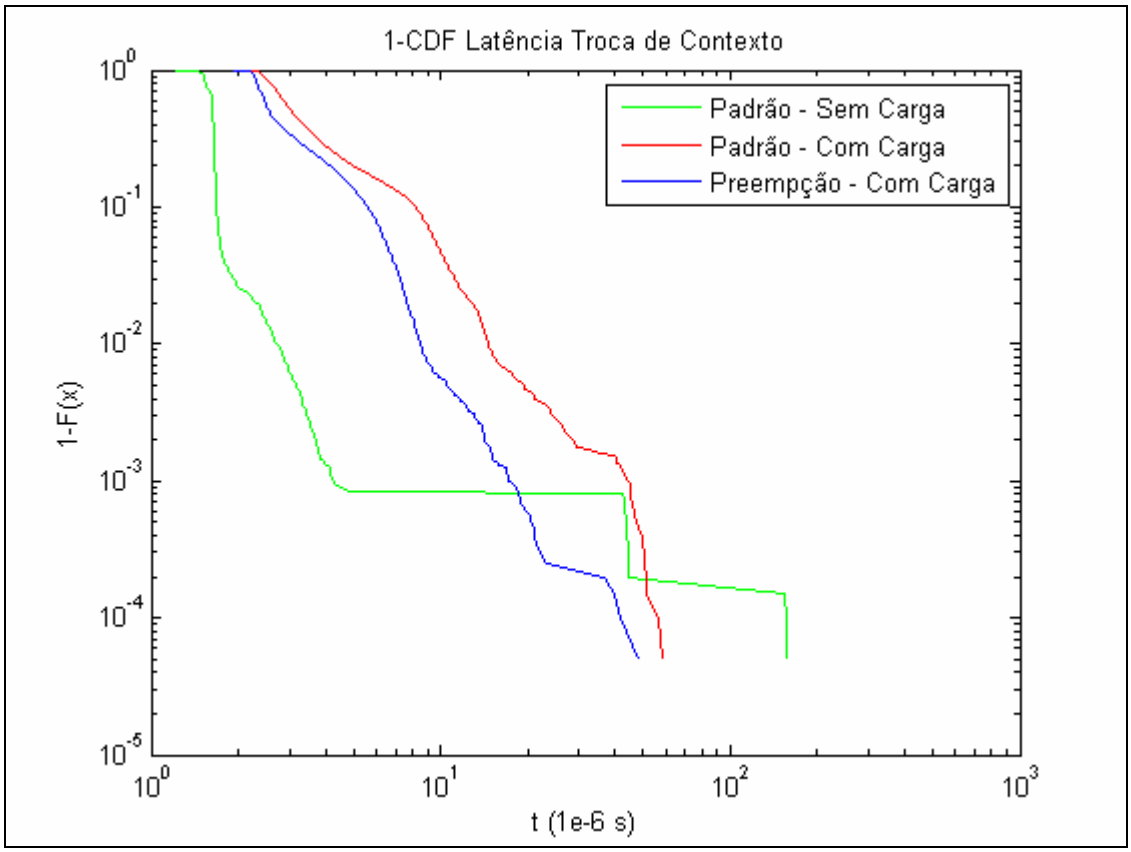

Figura 6.23 – Gráfico (1-CDF) da latência de troca de contexto

Análise:

O valor médio mais baixo no teste “Padrão – Sem Carga” pode ser explicado pelo fato de que esta operação atualiza estruturas do núcleo e, conseqüentemente, esta latência pode ser influenciada pela coincidência de cache.

Os valores altos obtidos nos três testes se justificam porque a parte final da troca de contexto é executada com as interrupções habilitadas. Estando as interrupções habilitadas, pode-se ter uma latência devido a:

1- Interrupções sendo tratadas durante a troca de contexto.
2- *Deferrable functions* sendo executadas durante a troca de contexto.
3- Processos de maior prioridade sendo executados, que foram colocados no estado pronto por uma interrupção tratada durante a troca de contexto[11] (item 1).

[11] Depois de efetuar a troca de contexto e antes de finalizar, o escalonador verifica se foi requisitada alguma troca de processo e, se for o caso, a executa.
Lembra-se que, só após o término da rotina do escalonador, o processo de medição é efetivamente retomado e o marcador do fim desta latência é executado.

6.7.10 Latência Total

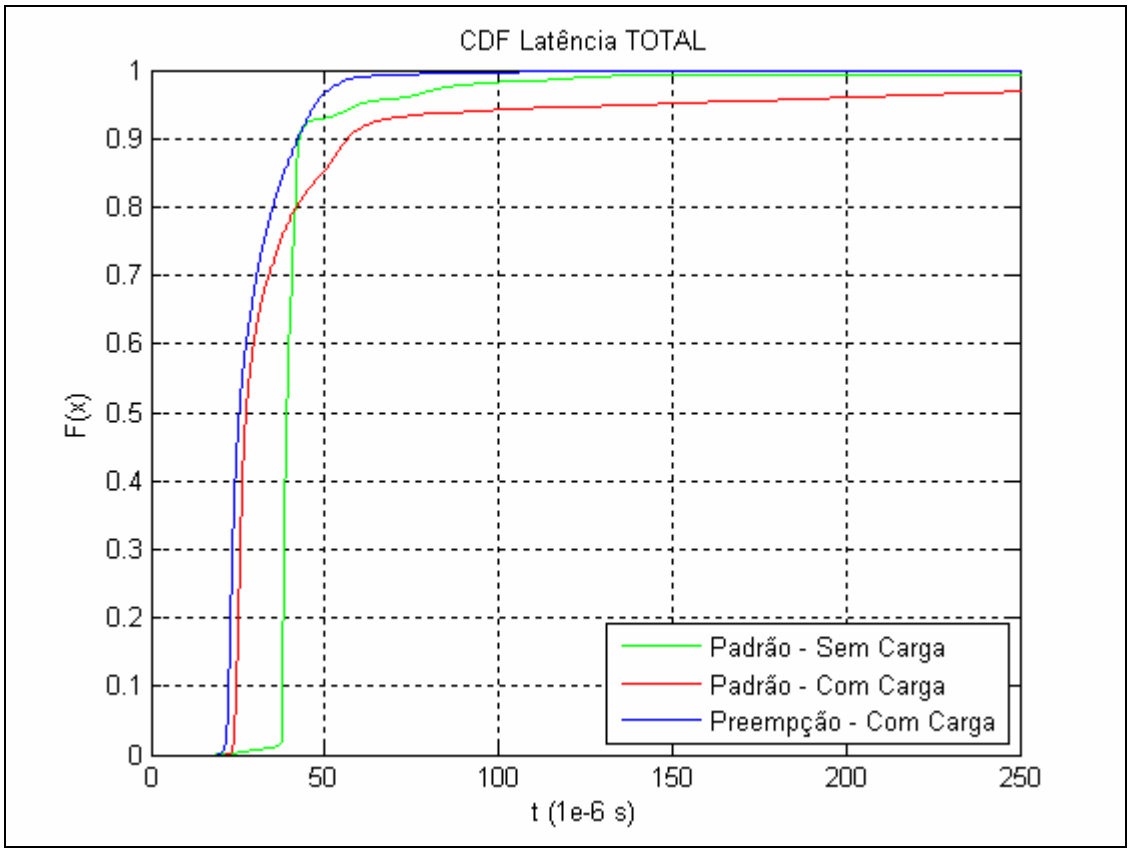

Figura 6.24 – CDF da latência total
Obs.: Gráfico da CDF truncado

Tabela 6.11 – Estatísticas da latência total

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	44,6	61,60	29,60
Máximo	691,9	1,42e+3	536,7
Mínimo	18,90	21,64	19,09
Mediana	39,12	27,33	25,38

Observa-se que a soma das médias das componentes é aproximadamente igual à latência total média, indicando que os dados obtidos são consistentes.

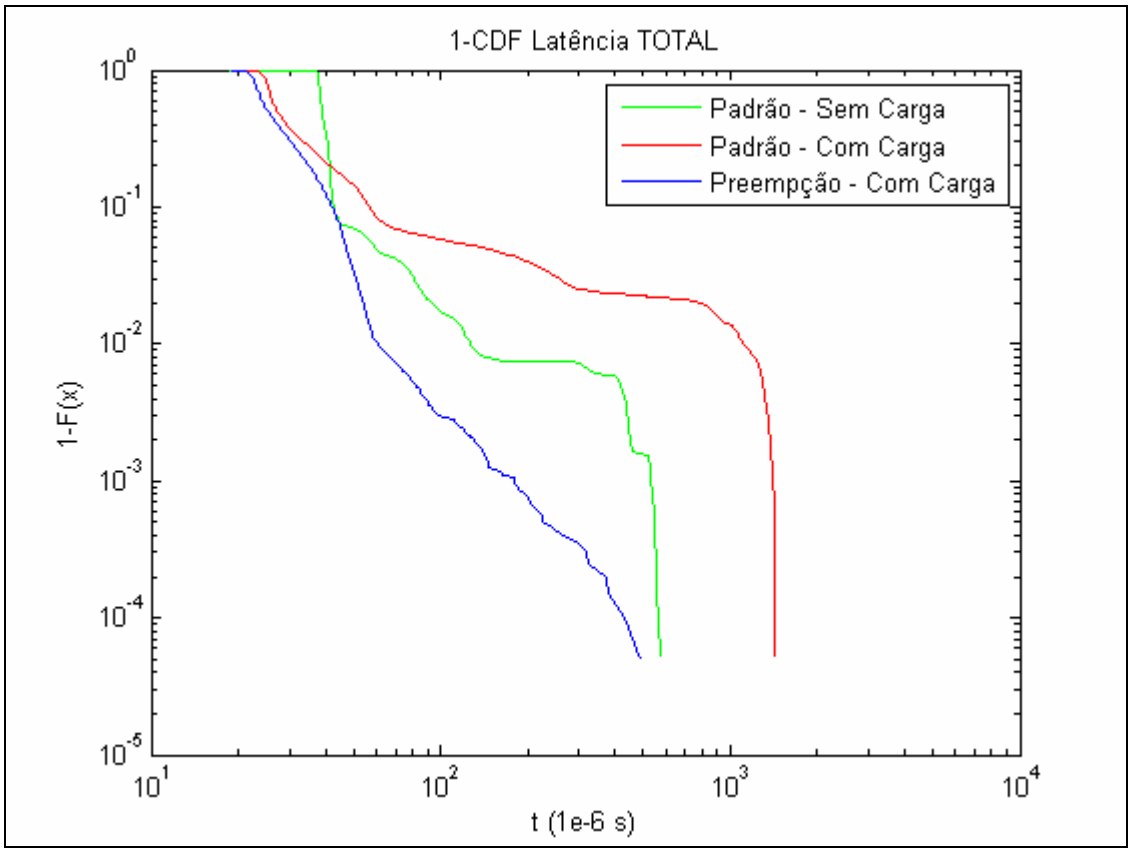

Figura 6.25 – Gráfico (1-CDF) da latência total

6.7.11 Subamostragem no Sistema

Como foi mostrado no item 6.5.6 foi desenvolvido um mecanismo que permite estimar um limite mínimo para o número de vezes que o sistema passa a subamostrar – definido como uma falha na tabela 6.12 - e o número de períodos que fica nesse estado, ou seja, o tempo de recuperação. Como foi esclarecido, esse mecanismo não leva em consideração interrupções sinalizadas durante o tratamento da amostra. Os valores obtidos com esse mecanismo encontram-se na tabela 6.12.

Tabela 6.12 – Número de Falhas e Períodos de Recuperação em 20.000 medições

	Padrão Sem Carga	**Padrão Com Carga**	**Preempção Com Carga**
Número de Falhas	6	989	6
Períodos Recuperação	1	1 a 28	1

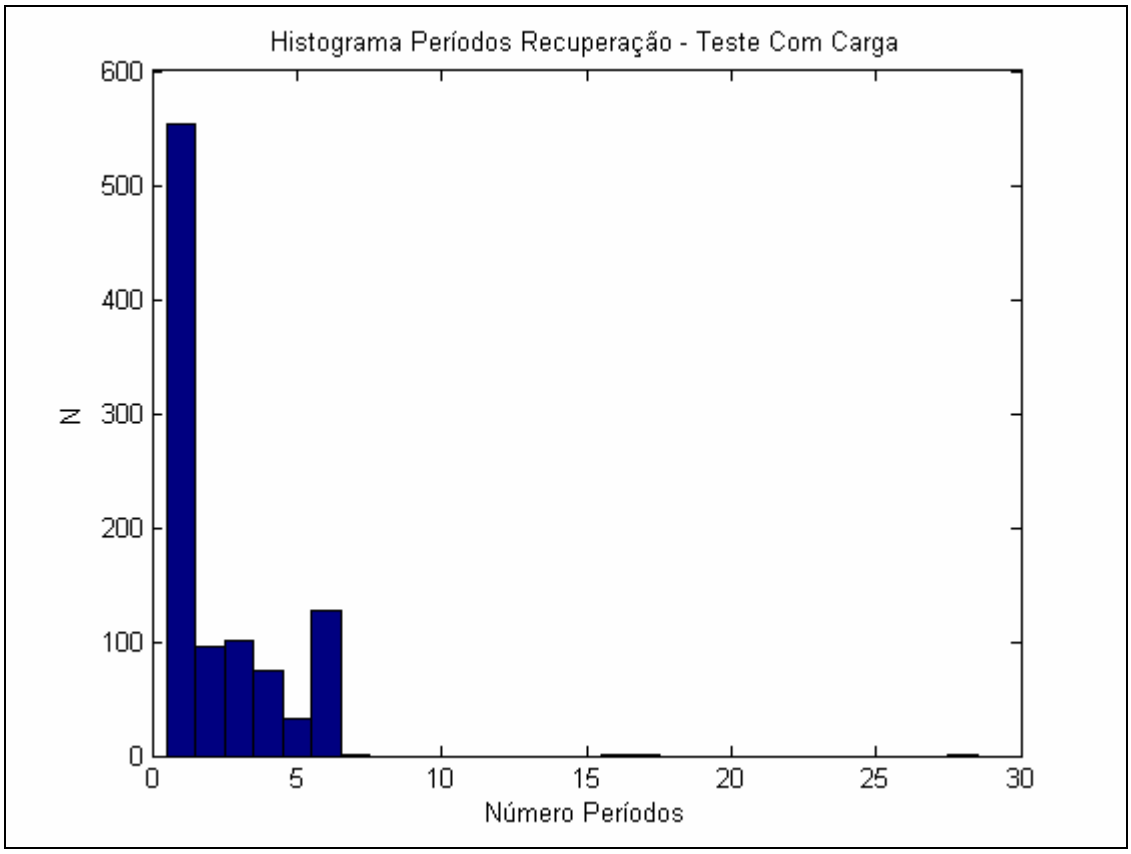

Figura 6.26 – Histograma períodos de recuperação para o teste com carga

6.8 Conclusão

Embora o foco principal deste trabalho não seja estudar o comportamento das várias componentes da latência, duas conclusões podem ser obtidas com os testes realizados:

1- Foi confirmada a importância da preempção do núcleo para processos em tempo real.

2- Mesmo usando todos os recursos à disposição no Linux 2.6.25, na máquina estudada, tem-se subamostragem para uma freqüência de 1.000 Hz, a qual pode ser considerada, de forma geral, baixa para aplicações de processamento digital de sinais.

6.9 Resumo

Neste capítulo foi apresentado um método para medir a latência total no tratamento da IRQ0 e estimar algumas componentes da latência de software. Utilizando o método desenvolvido foram efetuadas medidas para a IRQ0 no Linux 2.6.25 em três cenários diferentes.

Capítulo 7 - Alterações no Linux para o Processamento Digital de Sinais

7.1 Introdução

Neste capítulo são apresentadas modificações no Linux 2.6.25 com base nas análises realizadas no Capítulo 4 e no Capítulo 5, as quais têm como objetivo melhorar o desempenho deste sistema operacional em aplicações de processamento digital de sinais em tempo real.

As modificações apresentadas não tratam da operação do sistema em modo *Pull-In*, ou seja, quando SO-PDS controla a freqüência de amostragem. Como foi visto no Capítulo 5, o Linux utiliza todas as fontes de tempo da arquitetura x86 e, portanto, não há uma fonte de tempo livre para controlar exclusivamente o período de amostragem. Desta forma a pesquisa de uma solução para esse tipo de operação passa pela utilização do relógio HPET.

Pelo colocado acima, o estudo neste capítulo é voltado para o caso em que a placa adaptadora controla a freqüência de amostragem e, portanto, está focado na diminuição da latência no tratamento da interrupção que sinaliza a amostragem.

As modificações podem ser agrupadas em cinco grupos, grupos estes que serão chamados de métodos, haja vista que cada grupo tem uma funcionalidade específica, como é mostrado a seguir:

1- **Método 1**: Tratador da Interrupção IRQa Simplificado (item 7.3).
2- **Método 2**: Tratamento de Interrupção Postergado (item 7.4).
3- **Método 3**: Troca Seletiva de Processo (item 7.5).
4- **Método 4**: Desabilitação da Interrupção pelo Núcleo Postergada (item .7.6).
5- **Método 5**: Bloqueio das D*eferrable Functions* (item 7.7).

O capítulo inicialmente faz uma revisão da latência no tratamento das interrupções E/S externas, a fim de mostrar como as alterações propostas buscam mitigar cada uma de suas componentes. Na continuação, os fundamentos de cada método são explicados. O capítulo finaliza com uma explicação de como foram implementados os métodos e com uma comparação entre a latência total medida no sistema operacional alterado e a medida no Linux 2.6.25.

7.2 Latências - Revisão

Neste item é apresentado um resumo das latências vistas no Capítulo 4, com a finalidade de separar aquelas que são mitigadas pelos métodos propostos, daquelas que não são.

7.2.1 Latência de Tratamento de Interrupção

Neste item são revistas as várias componentes que formam a latência de tratamento de interrupção. A ordem das componentes é a mesma estabelecida no Capítulo 4.

1- **Leitura da interrupção (item 4.3.1)** - Decorre da arquitetura dos microprocessadores da família x86 e, portanto, não tem como ser mitigada.

2- **Mascaramento (item 4.3.2)** - Decorre das interrupções serem desabilitadas pelo microprocessador ou pelo sistema operacional. A componente da latência de mascaramento causada pelo microprocessador é tratada pelo Método 2. A componente da latência de mascaramento causada pelo núcleo é tratada pelo Método 4.

3- **Fila do controlador programável de interrupções (item 4.3.3)** – Decorre do mecanismo de prioridade do controlador de interrupções. É desejável que a interrupção que controla a amostragem (IRQa) tenha a maior prioridade, ou seja, que tenha o vetor de interrupção de menor número possível. Isto implica em modificar os vetores de interrupção, os

quais são estabelecidos na carga do sistema pelo BIOS. Por envolver aspectos ligados ao hardware, que extrapolam os objetivos deste texto, esta latência não é tratada por nenhum dos métodos propostos.

4- **Operação do microprocessador (item 4.3.4)** - Decorre unicamente do microprocessador e, portanto, não tem como ser mitigada.

5- **Salvamento dos registradores (item 4.3.5)** – A maior parte desta latência é gerada pelo salvamento dos registradores e pelas operações de controle do sistema operacional. O Método 1 mitiga esta componente diminuindo o número de instruções para controle da interrupção.

6- **Entrada no tratador de interrupções (item 4.3.6)** – Esta latência é originada pelas camadas de software do tratador de interrupções, sendo mitigada pelo Método 1.

7- **Compartilhamento (item 4.3.7)** – Originada pelo compartilhamento de interrupções, é mitigada pelo requisito que a interrupção que sinaliza a amostragem não é compartilhada. O Método 1 leva este requisito em consideração.

7.2.2 Latência de Despacho

Neste item são revistas as várias componentes que formam a latência de despacho. A ordem de apresentação das componentes é a mesma estabelecida no Capítulo 4.

1- **Saída do tratador de interrupções (item 4.5.1)** - Por ter origem semelhante à latência de entrada no tratador de interrupções, esta latência é mitigada pelo Método 1.

2- **Pilha de IRQs (item 4.5.2) -** Ocorre quando há interrupções aninhadas e decorre de não haver preempção do núcleo enquanto uma interrupção estiver sendo tratada. Esta latência é eliminada pelo Método 3.

3- ***Deferrable functions*** **(item 4.5.3)** - Decorre da execução dessas funções após o tratamento da interrupção. Esta latência é eliminada pelo Método 1.

4- **Preempção do núcleo desabilitada (item 4.5.4) -** Decorre da preempção do núcleo estar desabilitada. Esta latência é mitigada pelo Método 3.

5- **Escalonamento (item 4.5.5)** – É diminuída pelo desenvolvimento de uma nova rotina de escalonamento, a qual faz parte do Método 3.

6- **Troca de Contexto (item 4.5.6)** – No Linux, parte da rotina que efetua a troca de contexto é executada com as interrupções habilitadas. Esta latência é mitigada pelo Método 3, que executa toda a troca de contexto com as interrupções desabilitadas.

7.2.3 Latência de Execução

Como foi visto no Capítulo 4, item 4.6, a latência de execução é o tempo de CPU gasto, durante a execução do processo PDS, com operações que não estão relacionadas ao mesmo. Três são as fontes desta latência: interrupções E/S, *deferrable functions* e memória virtual.

A latência gerada pela memória virtual está relacionada à forma como o programa é carregado na memória. Por não ser um tema ligado ao tratamento das interrupções não é tratado neste texto.

O programa PDS é executado com as interrupções desabilitadas como parte da implementação do Método 3 e, portanto, a componente da latência gerada pelas interrupções E/S é eliminada por esse método.

A componente da latência gerada pelas *deferrable functions* é eliminada pelo Método 5.

7.3 Tratador da Interrupção IRQa Simplificado - Método 1

A idéia básica deste método é que a interrupção de amostragem (IRQa) seja tratada por um tratador de interrupções mais simples do que o tratador original do Linux, o qual possui algumas instruções que decorrem do Linux possuir as características a seguir:

1- Permitir que uma rotina de serviço da interrupção (ISR) seja executada com as interrupções habilitadas.
2- Tratar sistemas SMMP.
3- Implementar mecanismos de controle das interrupções, que se justificam principalmente em um sistema operacional voltado para servidores.
4- Executar funções do núcleo após o tratamento das interrupções (ex: *deferrable functions*).

O método proposto está baseado nas seguintes premissas:

1- A IRQa é tratada com as interrupções desabilitadas.
2- A IRQa não é compartilhada.
3- O sistema é monoprocessador.

Com bases nas premissas acima, foi desenvolvida uma nova rotina para tratar exclusivamente a interrupção IRQa, a qual não executa as *deferrable functions* ao final do tratamento da interrupção. Desta forma essas funções são executadas após o tratamento de todas as interrupções, exceto a IRQa.

Além da nova rotina, o método leva em consideração o parâmetro de compilação `CONFIG_TRACE_IRQFLAGS_SUPPORT`, que não é definido. Parte da latência de salvamento dos registradores (Capítulo 4, item 4.3.5) é gerada pela execução da macro `TRACE_IRQS_OFF`, que atualiza o estado das interrupções para controle do Linux. Esse controle é desabilitado através do parâmetro de compilação `CONFIG_TRACE_IRQFLAGS_SUPPORT`, trocando-se um controle maior por uma latência menor.

7.4 Tratamento de Interrupção Postergado - Método 2

O Capítulo 3 mostrou que o sinal de amostragem é periódico e estabeleceu que o período é informado ao SO-PDS. Portanto, a partir do instante da chegada de uma IRQa, o sistema operacional pode estimar o instante de chegada da próxima.

No Capítulo 6 foi desenvolvido um método que, através da introdução de marcadores de tempo no núcleo, estima a latência de software no tratamento da interrupção IRQ0. Este método pode ser modificado para medir a latência de software no tratamento de todas as interrupções E/S externas do sistema.

Os dois fatos acima podem ser combinados para desenvolver um método adaptativo, que diminui a componente da latência de mascaramento originada pelo tratamento das interrupções. Explicado a seguir, o método é dividido em duas partes: medição dos tempos e lógica de postergação.

7.4.1 Medição dos Tempos

A idéia consiste em colocar marcadores de tempo no tratador de interrupções, com o objetivo de medir a latência de software de forma semelhante à apresentada no Capítulo 6. Como os marcadores são colocados apenas no tratador de interrupções, é preciso levar em consideração certas limitações na medição:

1- Existem interrupções que podem chamar o escalonador, por exemplo, a interrupção sinalizada pelo disco rígido. Neste caso, a medida deve ser ajustada para incluir o tempo gasto com a troca de processo. O método desenvolvido neste texto não implementa este ajuste. Uma solução possível seria somar ao tempo medido um valor médio para o tempo de troca de processo, sempre que o tratador de interrupções solicitar a execução do escalonador. Esse valor médio pode ser atualizado periodicamente por um *daemon*.

2- Como foi visto no Capítulo 4, item 4.3.6.3, podem existir interrupções que são tratadas com as interrupções habilitadas. Para esse tipo de

interrupção, o método desenvolvido implementa um mecanismo para incluir no tempo total o tempo gasto com o tratamento de outras interrupções, sinalizadas durante o tratamento da interrupção que está sendo medida.

Nesta primeira versão do método, considera-se apenas o último valor medido como estimativa do tempo total para o tratamento da interrupção.

7.4.2 Lógica de Postergação

A lógica de postergação é trivial, como mostrado a seguir:

1- O sistema operacional tem uma estimativa do instante t_a de sinalização da próxima IRQa.
2- Em t_0 chega uma interrupção IRQn com $n \neq a$.
3- Ao entrar no tratador de interrupções, o sistema operacional obtém o tempo de processamento da última IRQn e estima o instante t_f no qual o tratamento da interrupção termina.
4- Se $t_a > t_f$ o tratamento da interrupção IRQn é executado, caso contrário, a interrupção é armazenada em uma fila para execução posterior.

Como não há sincronismo entre o núcleo e a interrupção que sinaliza a amostragem, a primeira estimativa de t_a é um valor inicial independente, o que ocasiona um transiente.

7.5 Troca Seletiva de Processo - Método 3

O método Troca Seletiva de Processo se propõe a eliminar as latências de pilha de IRQs e de preempção do núcleo desabilitada, como explicado a seguir, haja vista que estas duas latências têm a mesma origem: quando um processo em Modo Supervisor executa certas rotinas do núcleo não pode sofrer preempção, ou seja, uma troca de processo.

Essa limitação do Linux pode ser explicada como segue:

1- O processo A está em Modo Supervisor, executando uma rotina do núcleo com as interrupções habilitadas. Um exemplo possível é a execução da rotina de serviço de interrupção (ISR) de uma interrupção IRQn. Como por hipótese as interrupções estão habilitadas, a ISR permite o aninhamento de interrupções.

2- Caso o Linux permitisse a preempção do núcleo e supondo que em t_0 uma interrupção IRQa fosse sinalizada, ocorreria uma troca de processo e, o processo PDS passaria a ser executado. Conseqüentemente seria interrompida a rotina do núcleo que o processo A estava executando, que no exemplo proposto é a ISR da IRQn.

3- No Linux, a rotina do núcleo interrompida só é retomada quando o processo A volte ao estado de em execução. Como o Linux não discrimina um processo em particular não há como saber quando processo A vai ser executado novamente. Conseqüentemente, supondo que o Linux permitisse a preempção do núcleo, existiria a possibilidade que estruturas do núcleo manipuladas pela rotina fossem corrompidas. No exemplo proposto, a ISR só seria retomada quando o escalonador selecionasse novamente o processo A, postergando o tratamento da IRQn por um tempo indeterminado.

Do exposto surge a idéia básica do método que possibilita a preempção do núcleo sem incorrer no problema acima, a qual consiste em retomar o processo que estava sendo executado no instante de sinalização da IRQa, após o processo PDS passar ao estado bloqueado. Desta forma, a execução da rotina do núcleo seria retomada logo após o processo PDS tenha finalizado o tratamento da amostra.

Esta solução não tem restrições adicionais no caso de uma pilha de interrupções aninhadas que se iniciou com o processo em Modo Usuário. Neste caso, no Linux original, o núcleo não sofreria preempção para que todas as interrupções da pilha fossem atendidas rapidamente.

A solução tem restrições no caso em que IRQa é sinalizada quando o processo em Modo Supervisor está, por exemplo, atualizando uma estrutura do núcleo com a preempção desabilitada. Neste caso, no Linux original, o núcleo não sofreria preempção para não deixar inacabada a atualização da estrutura. Desta forma, ao usar o método, existe a possibilidade de que o processo PDS atualize erradamente as mesmas estruturas, que estavam sendo manipuladas quando a IRQa foi sinalizada. Como o programa PDS é basicamente intensivo em CPU em Modo Usuário, pode-se levantar a hipótese de que é baixa probabilidade desse tipo de erro ocorrer. Como o Linux fornece um mecanismo para diferenciar ambos os casos, é possível só aplicar o método para o primeiro.

O método proposto consiste basicamente no desenvolvimento de um novo escalonador que:

1- **Quando a IRQa é sinalizada**: a) salva o identificador do processo NPDS que estava em execução, b) seleciona o processo PDS e, c) passa o processo PDS ao estado de em execução.

2- **Quando o processo PDS é bloqueado**: a) recupera o identificador do processo NPDS e, b) passa o processo NPDS ao estado de em execução.

Por ser mais simples que o escalonador original do Linux, este novo escalonador tem a vantagem adicional de diminuir a latência de escalonamento.

7.6 Desabilitação da Interrupção pelo Núcleo Postergada - Método 4

A idéia aqui proposta baseia-se em aspectos teóricos e não foi implementada, ficando como sugestão para futuras pesquisas.

O método é um método adaptativo baseado no recurso do Linux chamado perfil do código do núcleo (*profiling the kernel code*). Este método visa diminuir a componente da latência de mascaramento gerada quando o núcleo desabilita as interrupções por meio do campo `IF`.

O método baseia-se na hipótese de que, na maior parte das vezes, o núcleo desabilita e habilita as interrupções pelas rotinas `local_irq_disable()` e `local_irq_enable()` respectivamente. As exceções ficam por conta dos trechos do núcleo desenvolvidos em linguagem de montagem que manipulam o campo `IF` por meio de instruções especificas.

De forma semelhante ao item 7.5, a idéia básica é estimar se o intervalo de tempo em que as interrupções ficariam desabilitadas é maior ou menor ao intervalo de tempo para a chegada da IRQa. Se o primeiro intervalo é menor, as interrupções são desabilitadas, se é maior, as interrupções não são desabilitadas. Como no item anterior, o método é dividido em duas partes: medição dos tempos e lógica de postergação.

7.6.1 Medição do Tempo

A parte da medição é desenvolvida incluindo novas funcionalidades às rotinas já existentes, como é descrito a seguir:

1- Rotina `local_irq_disable()`:

   a. Usando a pilha do processo se obtém o endereço da instrução que fez a chamada à rotina.
   b. Usando um marcador de tempo determina-se o instante em que a interrupção é desabilitada.
   c. Em uma estrutura de dados na memória grava-se o endereço da instrução e a medida de tempo.

2- Rotina `local_irq_enable()`:

   a. Usando um marcador de tempo determina-se o instante em que a interrupção é habilitada.
   b. Na estrutura, procura-se o endereço da instrução que chamou a rotina `local_irq_disable()` e que foi salvo no passo 1.c.

c. Calcula-se o intervalo de tempo com as interrupções desabilitadas para o bloco de instruções que inicia no endereço obtido pela rotina `local_irq_disable()`.

7.6.2 Lógica de Postergação

A lógica de postergação é implementada na rotina `local_irq_disable()` e basicamente:

1- Obtém-se a estimativa do instante t_a de chegada da próxima IRQa (Ver item 7.4).
2- Usando a pilha do processo, obtém-se o endereço da instrução que fez a chamada à rotina `local_irq_disable()`.
3- Com o endereço desta instrução efetua-se uma busca na estrutura de dados e, com os dados armazenados, estima-se o instante de tempo t_f em que o sistema habilitará as interrupções.
4- Se $t_a > t_f$ as interrupções são desabilitadas. Caso contrário o núcleo entra em um laço de espera da IRQa.

7.6.3 Observações

1- O Linux procura minimizar o tempo com as interrupções desabilitadas. Portanto, existe a possibilidade que o intervalo de tempo máximo com as interrupções desabilitadas seja menor que o intervalo de tempo necessário para a execução da lógica de postergação, o que inviabilizaria o método.

2- A heurística de um intervalo de tempo máximo baixo com as interrupções desabilitadas é o que justifica o laço de espera no caso de $t_f \geq t_a$. Se não há tempo suficiente para executar o trecho de código com as interrupções desabilitadas antes da chegada da IRQa, assume-se a hipótese de que não há tempo suficiente para bloquear o processo, o que implicaria em uma chamada ao escalonador e em uma troca de contexto.

3- Uma limitação do método é nos casos em que o trecho de núcleo executado não chama diretamente a rotina `local_irq_disable()`, como por exemplo, quando o núcleo chama a rotina de um mecanismo de sincronização (ex: semáforo) que, por sua vez, chama a rotina `local_irq_disable()`.

7.7 Bloqueio das *Deferrable Functions* – Método 5

Este método bloqueia a execução das *deferrable functions* quando o processo em execução é o processo PDS. Bastante simples, o método consiste basicamente de:

1- Uma função que verifica se o processo em execução é o processo PDS. No Apêndice J, item J.20, pode ser visto o código fonte da função desenvolvida, denominada `ver_pds()`.

2- Uma alteração na função `local_softirq_pending()`.

7.8 Alterações no Linux – 1ª Parte

A descrição das alterações realizadas no Linux para implementar o Método 1, Método 2 e Método 3 está dividida em três partes. Este primeiro item fornece uma visão geral do funcionamento do sistema operacional alterado, o qual será chamado, no restante do capítulo, abreviadamente de SO-PDS.

7.8.1 Estrutura

No núcleo do Linux 2.6.25 foi incluída uma estrutura do tipo `pds_real_time_struct` e que foi chamada `pds_rt_st`. Os campos dessa estrutura podem ser vistos na tabela 7.1 e são explicados a seguir. A definição da estrutura encontra-se no Apêndice J, item J.1.

Observa-se que no controle de tempo foi usado o registrador `tsc`, portanto, a unidade de tempo usada é o ciclo de relógio da CPU. Na estrutura, os campos que armazenam informações de tempo são de 64 bits.

Tabela 7.1 – Campos da estrutura de dados `pds_rt_st`

Campo	Função
`modo`	Controla o estado de operação do SO-PDS
`irq_cod`	Número da interrupção que sinaliza a amostragem (IRQa)
`periodo_amost`	Período de amostragem
`monitorada[50]`	Controla a medição do tempo de tratamento da IRQi
`medida[50][3]`	Armazena os tempos do tratamento da interrupção IRQi
`overflow_tsc`	Informa que registrador `tsc` zerou
`prox_irq_amost`	Instante estimado da próxima IRQa
`fila_conta`	Tamanho da fila das interrupções postergadas
`fila_irqs[50]`	Fila das interrupções postergadas
`pds_estado`	Controla o estado do processo PDS
`*pds`	Ponteiro que identifica o processo PDS
`*npds`	Ponteiro que identifica processo NPDS

A - Campos de uso geral

O campo `modo` controla o estado de operação no qual o SO-PDS se encontra. Os estados de operação do SO-PDS são tratados no item 7.8.2.

O campo `irq_cod` contém o número da IRQa e o campo `periodo_amost` armazena o período de amostragem.

B - Campos exclusivos para o Método 2 (ver item 7.4)

O vetor `monitorada` serve para controlar a medição do tempo de tratamento da interrupção IRQi e, evitar uma medida errônea quando há o aninhamento de interrupções. A matriz `medida` contém as seguintes informações relacionadas ao tratamento da IRQi: instante inicial, instante final e intervalo de tempo decorrido. Embora o hardware do PC suporte, nas máquinas com APIC E/S[1], no máximo 24 interrupções E/S externas, o SO-PDS suporta até 50.

[1] O APIC E/S é tratado no Apêndice B, item B.6.2.

O campo `prox_irq_amost` armazena o instante estimado de chegada da próxima interrupção IRQa. O registrador `tsc` é um contador ascendente, podendo zerar. Neste caso, é preciso uma sinalização para evitar comparações erradas, o que é feito pelo campo `overflow_tsc`.

O campo `fila_conta` e o vetor `fila_irqs` implementam uma fila do tipo *First In – First Out* (FIFO) para armazenar as interrupções postergadas.

C - Campos exclusivos para o Método 3 (ver item 7.5)

O campo `pds_estado` serve para monitorar o estado do processo PDS. Os estados do processo PDS são explicados no item 7.8.5.

Os campos `pds` e `npds` são ponteiros para o descritor do processo PDS e do processo NPDS respectivamente.

7.8.2 Estados de Operação do SO-PDS

Foram definidos três estados de operação para o SO-PDS. O diagrama de estados é apresentado na figura 7.1 e os estados são descritos a seguir:

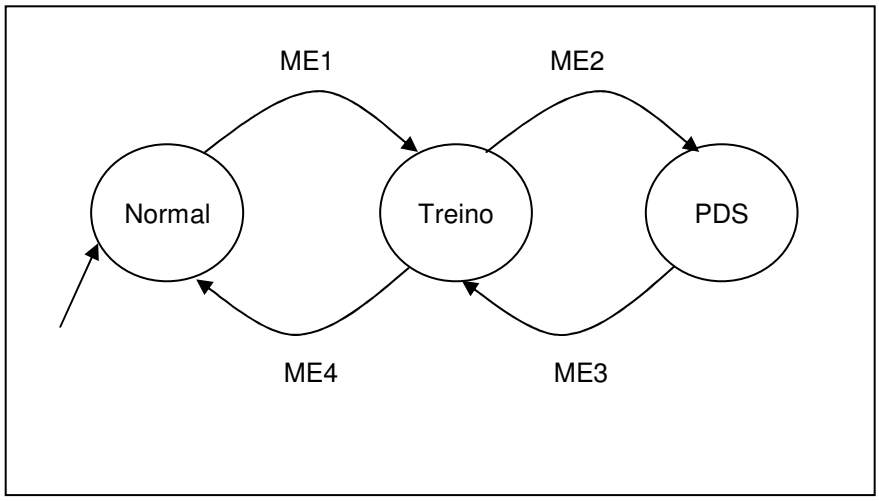

Figura 7.1 – Diagrama de estados do SO-PDS

1- **Normal** – Neste estado, a operação do sistema é semelhante à do Linux 2.6.25, ou seja, nenhuma operação relacionada ao SO-PDS é executada. É o estado no qual é carregado o sistema operacional.

2- **Treino** – Neste estado, o SO-PDS mede os tempos para o tratamento das interrupções do sistema. Esta fase de treinamento é necessária para o Método 2 (item 7.4).

3- **PDS** – Neste estado, o processo PDS já foi estabelecido. Basicamente, o processo PDS pode estar bloqueado, esperando a amostra, ou em execução, tratando a amostra. O item 7.8.3 apresenta mais detalhes sobre este estado.

As mudanças de estado (ME1, ME2, ME3 e ME4) são feitas executando a chamada ao sistema de número 188, descrita no item 7.8.6, com os parâmetros adequados. Para mudar o estado, a chamada ao sistema basicamente altera o campo `modo` da estrutura `pds_rt_st`.

7.8.3 Processo PDS

Para passar ao status de processo PDS, um processo qualquer deve executar a chamada ao sistema 188, solicitando a mudança de estado de “Treino” para “PDS”, indicada por ME2 na figura 7.1. Ao solicitar esta mudança de estado, o processo deve informar, como parâmetros da chamada ao sistema, o número da IRQa e o período de amostragem (T_A).

Essa mudança de estado só é realizada se o SO-PDS estiver no estado “Treino”, caso contrário a operação não é executada e o processo não adquire o status de processo PDS.

Ao fazer a mudança de estado, o núcleo salva o número da IRQa e o período de amostragem (T_A) na estrutura `pds_rt_st`. O núcleo também salva no campo `pds` um ponteiro para o descritor do processo que solicitou a mudança de estado e, conseqüentemente, passou ao status de processo PDS.

Para o processo PDS deixar o status PDS, basta efetuar a mudança do estado “PDS” para o estado “Treino”.

É interessante observar que este mecanismo garante o requisito de unicidade do processo PDS (ver Capítulo 3, item 3.7).

7.8.4 Estrutura Programa PDS

Na figura 7.2 a seguir pode ser visto o diagrama de blocos simplificado[2] de um exemplo de programa PDS, o qual implementa todas as mudanças de estados definidas para o SO-PDS.

Na figura 7.2 tem-se:

1- O estado do SO-PDS é mudado de “Normal” para “Treino” e, na continuação, o processo espera um intervalo de tempo para que as medidas para o Método 2 sejam realizadas. Ver o item 7.4.

2- O estado do SO-PDS é mudado de “Treino” para “PDS” e, conseqüentemente, o processo passa ao status de processo PDS. Ver o item 7.8.3.

3- O processo PDS fica bloqueado na operação de leitura da amostra até a sinalização da IRQa. A operação de leitura é realizada pela chamada ao sistema de número 189, que é explicada no item 7.8.6.

4- A execução do processo PDS é retomada e a amostra é tratada.

5- Ao final do processamento digital do sinal, informa-se ao SO-PDS que não há mais amostras para serem lidas. Está operação[3] também é

[2] O digrama de blocos apresentado não leva em consideração o caso das trocas de estado não serem realizadas.
[3] Esta operação faz parte do Método 3, que é explicado no item 7.5.

realizada pela chamada ao sistema de número 189 (ver passo 3). A informação é usada pelo núcleo para:

a. Retomar o processo NPDS.
b. Como o processo PDS não mais ficará bloqueado à espera da IRQa, o núcleo deverá usar o escalonador do Linux para colocá-lo novamente no estado em execução.

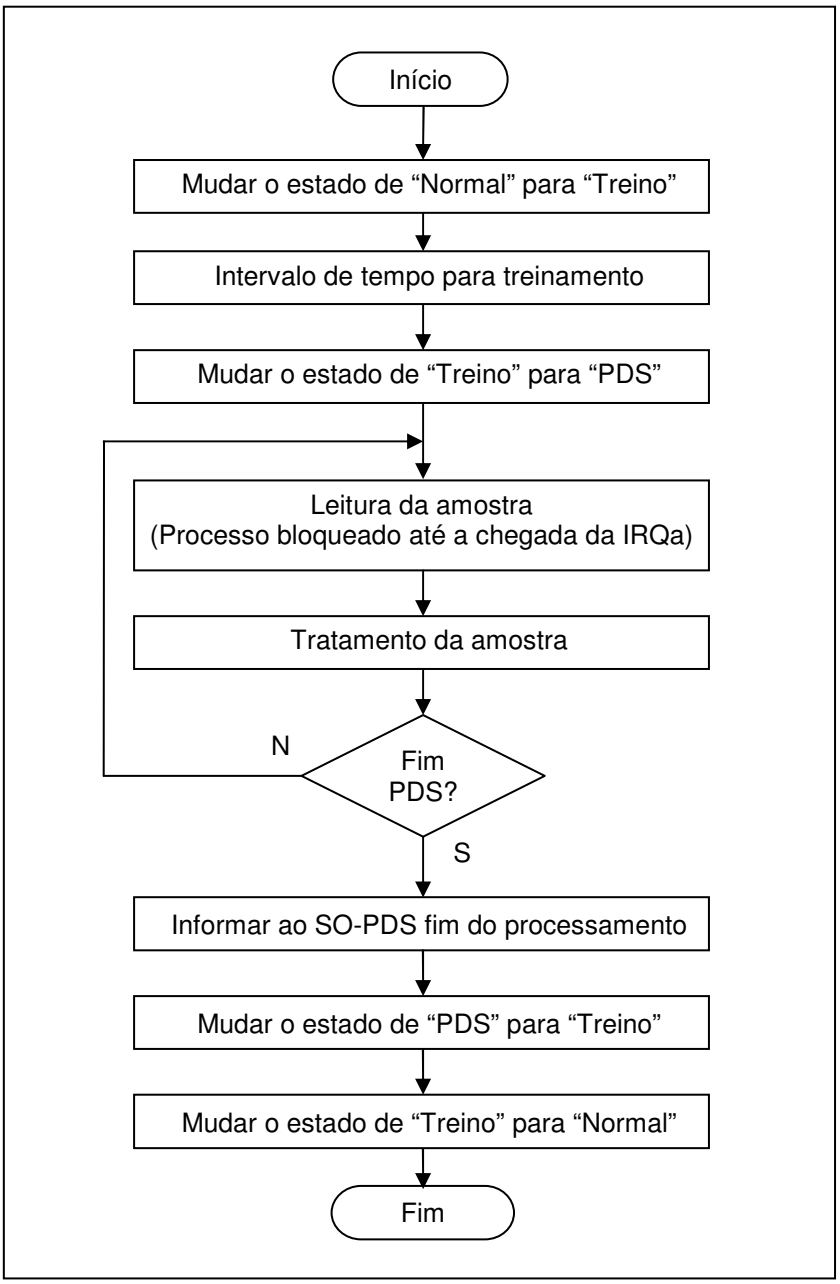

Figura 7.2 – Diagrama de blocos de um programa PDS

7.8.5 Estados do Processo PDS

Foi desenvolvido um controle para o estado do processo PDS, que difere um pouco do controle usado pelo Linux, descrito no Apêndice A, item A.4. O controle é implementado atualizando-se o campo `pds_estado` na estrutura `pds_rt_st`. O diagrama de estados pode ser visto na figura 7.3 e os estados[4] são definidos na continuação.

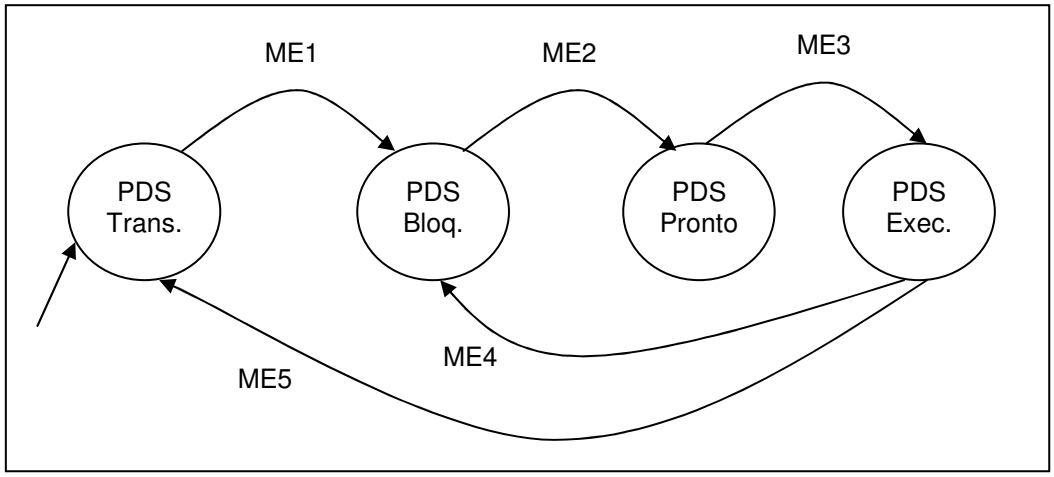

Figura 7.3 – Diagrama de estados para o processo PDS

1- **PDS Bloqueado** – O processo PDS está bloqueado aguardando a sinalização da IRQa.
2- **PDS Pronto** – A IRQa já foi tratada, mas o processo PDS ainda não detém a posse da CPU.
3- **PDS Execução** – O processo PDS está tratando a amostra.
4- **PDS Transição** – O processo PDS está neste estado em duas situações:
   a. O processo adquiriu o status de processo PDS, mas ainda não executou, pela primeira vez, a operação de leitura da amostra.
   b. O processo já terminou o processamento digital do sinal, mas ainda não deixou o status de processo PDS.

[4] Devido à semelhança entre os estados definidos e os estados comuns a um sistema operacional, foi usada a sigla PDS para diferenciá-los.

A seguir uma descrição de como são realizadas as mudanças de estado:

1- **ME1** – A chamada ao sistema que efetua a leitura da amostra coloca o processo PDS no estado “PDS Bloqueado”, antes de passá-lo para o estado bloqueado no Linux.
2- **ME2** – A rotina de serviço da interrupção (ISR) da IRQa coloca o processo PDS no estado “PDS Pronto”.
3- **ME3** – Quando a chamada ao sistema que efetua a leitura da amostra é retomada, o processo passa para o estado “PDS Execução”.
4- **ME4** – Esta mudança de estado decorre do laço que executa novamente a chamada ao sistema de leitura da amostra (ver passo 1).
5- **ME5** – Esta mudança de estado é efetuada quando se informa ao SO-PDS que foi finalizado o processamento digital do sinal.

Este mecanismo foi implementado para manter a consistência do SO-PDS em duas situações:

1- Quando há a ocorrência de subamostragem.
2- Chegada de uma IRQa quando o processo PDS está no estado “PDS Transição”.

7.8.6 Chamadas ao sistema

As duas chamadas ao sistema desenvolvidas para o método de medição (ver Capítulo 6, item 6.5.3) foram alteradas para implementar o SO-PDS.

A - Chamada ao Sistema 188

Como mencionado no item 7.8.2, a chamada ao sistema de número 188 é usada para mudar o estado do SO-PDS. Esta chamada tem três parâmetros: `ope`, `num_irq` e `periodo_a`. O primeiro parâmetro `ope` indica qual a mudança de estado a ser feita. Os outros dois parâmetros (`num_irq` e `periodo_a`) servem para informar ao sistema operacional, respectivamente, o número da IRQa e o período de amostragem. Esses dois parâmetros só são utilizados quando o SO-PDS passa do estado “Treino” para o estado “PDS”.

B - Chamada ao Sistema 189

Como foi mencionado no item 7.8.4, a chamada ao sistema de número 189 executa duas funções diferentes no SO-PDS:

1- A primeira função, chamada `driver_le_amostra`, basicamente bloqueia o processo PDS e, quando a execução é retomada, transfere a amostra do espaço de memória do núcleo para o espaço de memória do processo PDS. Pela sua importância esta função é analisada com mais detalhes no item 7.10.

2- A segunda função, chamada `driver_eof()`, basicamente passa o processo PDS para o estado "PDS Transição" e retoma a execução do processo NPDS.

Esta chamada ao sistema tem dois parâmetros: `ope` e `buff`. O primeiro parâmetro é usado para indicar qual a função será executada. O segundo parâmetro é um ponteiro para um buffer no espaço de memória do processo PDS, o qual só é usado pela função `driver_le_amostra`. Esta chamada ao sistema pode ser vista no Apêndice J, item J.2.

7.9 Descrição das Alterações no Linux – 2ª Parte

Neste item são analisadas as alterações realizadas no tratador de interrupções do Linux, que, como visto, também incluem um novo escalonador.

7.9.1 Introdução

Como uma introdução à análise, na figura 7.4 é apresentado o diagrama de blocos da parte do tratador de interrupções que foi o foco das alterações. Estas mesmas instruções foram vistas no Capítulo 4, item 4.3.5.

A finalidade deste diagrama de blocos é mostrar que as alterações efetuadas no tratador de interrupções começam pela rotina `do_irq`, que foi totalmente refeita.

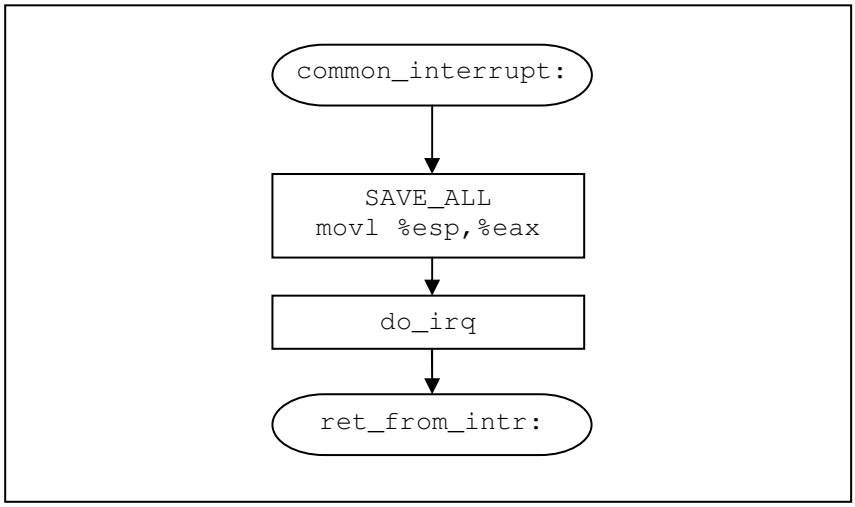

Figura 7.4 – Diagrama de blocos de parte do tratador de interrupções do Linux

7.9.2 Análise da Rotina: `do_irq`

O código da figura 7.5 mostra a parte principal da nova rotina `do_irq`. O código fonte completo pode ser encontrado no Apêndice J, item J.3.

```
001    /* Separa o tratamento das IRQs */
002    if (ver_codigo_irq(irq)) {
003           ax_int=do_IRQ_PDS_S(regs);
004           return ax_int;
005    }

006    else {
007           ax_int=do_IRQ_PDS_N(regs);
008           return ax_int;
009    }
```

Figura 7.5 – Código fonte parcial da rotina `do_irq`

Como se pode ver na figura, o tratamento da interrupção é direcionado para duas funções diferentes, dependendo do valor lógico retornado pela função `ver_codigo_irq`, que tem como parâmetro de entrada o código da interrupção `irq`. O código fonte completo da função pode ser encontrado no Apêndice J, item

J.4. A função retorna o valor `True` se as três condições a seguir são satisfeitas simultaneamente:

1- O SO-PDS estiver no estado “PDS”.
2- O processo PDS não estiver no estado “PDS Transição”.
3- A interrupção `irq` for a interrupção IRQa.

A idéia básica é dividir o tratamento das interrupções em dois tratadores:

1- O primeiro tratador `do_IRQ_PDS_S` trata a interrupção IRQa quando há um processo PDS sendo executado no sistema. Este tratador será descrito no item 7.9.4.

2- O segundo tratador `do_IRQ_PDS_N` é o tratador usado em todas as outras situações. Este tratador será visto no item 7.9.3.

7.9.3 Análise da Função: `do_IRQ_PDS_N`

A figura 7.6 na continuação mostra o diagrama de blocos da função `do_IRQ_PDS_N`. Na figura tem-se:

1- A função `ver_trata_irq` verifica se a interrupção `irq` vai ser tratada ou postergada. Se a função retorna `True`, a interrupção é tratada, se retorna `False`, o tratamento da interrupção é postergado. A função pode ser encontrada no Apêndice J, item J.5. A lógica de postergação pode ser explicada como segue:

   a. A interrupção só é postergada se o SO-PDS estiver no estado “PDS”.
   b. Se a fila de interrupções postergadas tiver pelo menos um elemento, todas as interrupções subseqüentes são postergadas. Isto é feito para manter o tratamento das interrupções postergadas na mesma ordem em que elas foram sinalizadas.

c. A interrupção é postergada se o instante estimado de finalização do tratamento da interrupção é posterior ao instante estimado para sinalização da próxima IRQa.

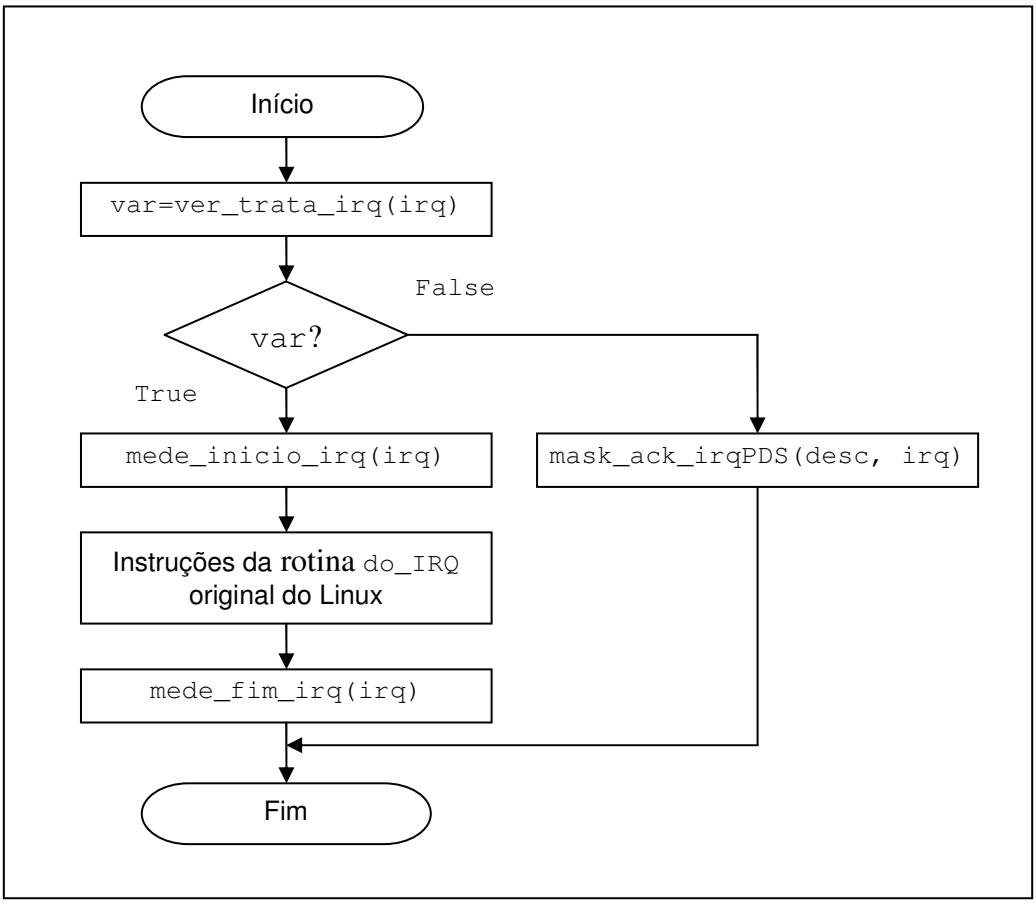

Figura 7.6 – Diagrama de blocos da função `do_IRQ_PDS_N`

A – Ramo Esquerdo

2- A rotina `mede_inicio_irq` mede o instante inicial em que o tratamento da interrupção `irq` foi iniciado e, também, implementa um controle para evitar medições erradas em caso de aninhamento de interrupções. Esta função pode ser vista no Apêndice J, item J.6.

3- A continuação a função `do_IRQ_PDS_N` é composta pelas mesmas instruções da rotina `do_IRQ` original do Linux, as quais efetuam o tratamento da interrupção.

4- A rotina `mede_fim_irq`, cujo código fonte pode ser visto no Apêndice J, item J.7, realiza basicamente as seguintes operações:

   a. Verifica o controle de aninhamento (ver passo 2).
   b. Mede o instante em que o tratamento da interrupção finalizou.
   c. Verifica se o registrador `tsc` zerou.
   d. Calcula o tempo decorrido no tratamento da interrupção.

B – Ramo Direito

5- A rotina `mask_ack_irqPDS`, cujo código fonte pode ser encontrado no Apêndice J, item J.8, é executada. Para esta rotina tem-se:

   a. O parâmetro `desc` é um ponteiro para o descritor da interrupção. Este descritor contém as informações necessárias para o tratamento da interrupção. Ver Apêndice C, item C.4.
   b. A rotina sinaliza a interrupção (*ack*) ao PIC e mascara a interrupção no próprio PIC.

7.9.4 Análise da Função: do_IRQ_PDS_S

A figura 7.7 mostra um diagrama de blocos da função `do_IRQ_PDS_S`, a qual só é usada para tratar a IRQa quando o SO-PDS está em modo PDS. O diagrama de blocos da figura mostra apenas as principais rotinas e funções executadas pela função, cujo código fonte completo pode ser visto no Apêndice J, item J.9. No diagrama de blocos tem-se:

1- A rotina `atualiza_prox_irqa()` calcula o instante estimado de sinalização da próxima IRQa. A rotina verifica se o registrador `tsc` zerou e marca um campo com esta informação. Isto é feito para que não haja erros nas comparações de tempos realizadas pelo SO-PDS. O código fonte completo desta rotina encontra-se no Apêndice J, item J.10.

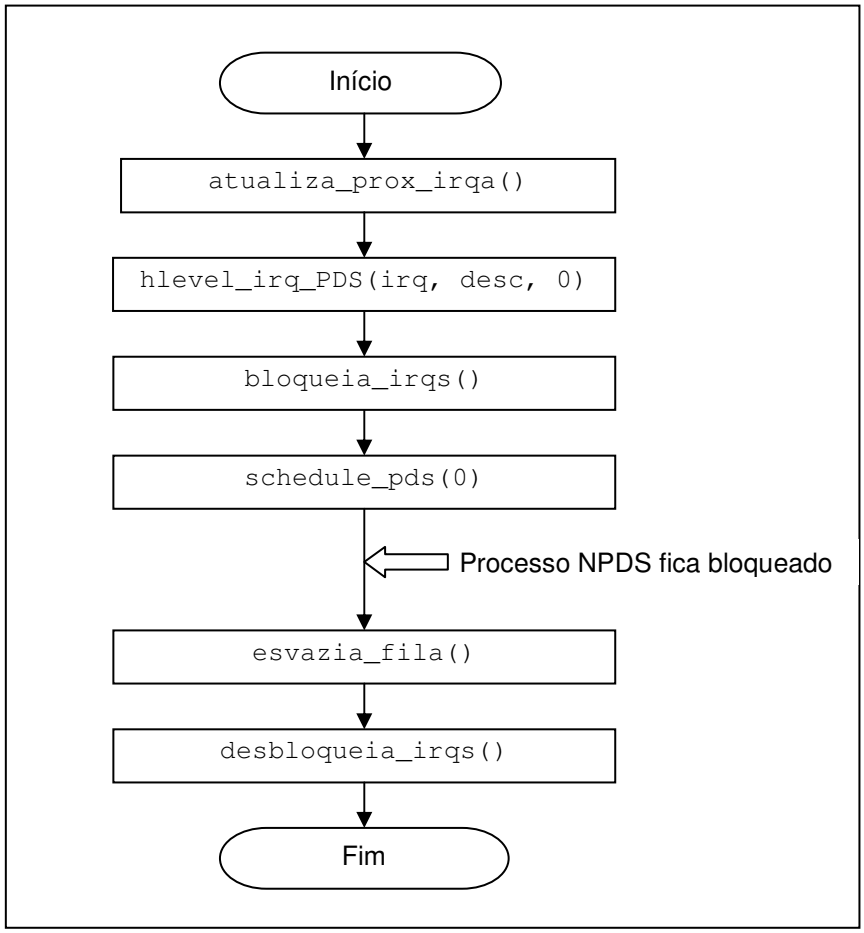

Figura 7.7 – Diagrama de blocos da função `do_IRQ_PDS_S`

2- A rotina `hlevel_irq_PDS` é um tratador simplificado de interrupções (ver item 7.3), na qual algumas operações de controle das interrupções do Linux foram eliminadas.

O código fonte desta rotina encontra-se no Apêndice J, item J.11. No Apêndice J, item J.12, tem-se o código fonte da função `handle_IRQ_evPDS`, o qual é necessário para o entendimento, já que esta função é chamada pela `hlevel_irq_PDS` para efetuar parte importante do tratamento das interrupções.

No diagrama de blocos, pode ser visto que a rotina `hlevel_irq_PDS` possui três parâmetros de entrada. O dois primeiros são respectivamente: o número da interrupção e um ponteiro para o descritor da interrupção. Esta função trata a IRQa e as interrupções que foram

postergadas, sendo que o terceiro parâmetro `operacao` determina qual operação deve ser executada, já que essas duas operações diferem em alguns comandos. O parâmetro `operacao` assume os valores: 0-tratamento IRQa e 1-tratamento interrupções postergadas.

Como a rotina `hlevel_irq_PDS` trata a IRQa, ela é a encarregada de desbloquear o processo PDS, de forma semelhante à vista no Capítulo 6, item 6.5.4. Isto é feito executando-se a rotina `driver_pds_isr`, cujo código fonte pode ser visto no Apêndice J, item J.13.

3- A rotina `bloqueia_irqs()` bloqueia todas as interrupções diretamente no controlador programável de interrupções. Isto é feito usando o método `mask` do objeto `chip`. Ver Apêndice C, item C.4. O código fonte desta rotina encontra-se no Apêndice J, item J.14 e o código fonte da rotina `desbloqueia_irqs()` no Apêndice J, item J.15.

4- A rotina `schedule_pds` é uma versão simplificada do escalonador do Linux. Este escalonador simplificado tem duas formas de operação dependendo do parâmetro de entrada `ope` que pode assumir dois valores: 0 (zero) ou 1(um). Neste item é explicada a operação quando `ope` = 0, que é o valor do parâmetro de entrada no diagrama de blocos da figura 7.7.

As principais operações efetuadas pela rotina são:

a. Verificar se houve subamostragem. Em havendo subamostragem a troca de processo não é feita.
b. Recuperar da estrutura `pds_rt_st` o ponteiro para o descritor do processo PDS.
c. Armazenar na estrutura `pds_rt_st` um ponteiro para o descritor do processo corrente (NPDS), para que este possa ser retomado ao término do tratamento da amostra.
d. Efetuar a troca de contexto.

Os códigos fonte desta rotina e de duas de suas funções, especialmente desenvolvidas para o SO-PDS, podem ser encontrados no Apêndice J, item J.16.

5- A rotina `esvazia_fila()` percorre a fila de interrupções postergadas e trata as interrupções armazenadas usando a rotina `hlevel_irq_PDS`, vista no passo 2. O código fonte desta rotina encontra-se no Apêndice J, item J.17.

7.10 Descrição das Alterações no Linux – 3ª Parte

Neste item são tratadas as funções `driver_le_amostra` e `driver_eof()` que implementam a chamada ao sistema 189 e que já foram vistas de forma superficial no item 7.8.6. Como a segunda é uma simplificação da primeira, somente a primeira será analisada.

O diagrama de blocos da função `driver_le_amostra` pode ser visto na figura 7.8 e o código fonte no Apêndice J, item J.18. O código fonte da função `driver_eof()` encontra-se no Apêndice J, item J.19.

Como pode ser visto na figura 7.8, a função `driver_le_amostra` basicamente:

1- Bloqueia o processo PDS que fica à espera da IRQa.

2- Ao reiniciar a execução, após o tratamento da IRQa, faz a transferência da amostra do espaço de memória do núcleo para o espaço de memória do processo PDS.

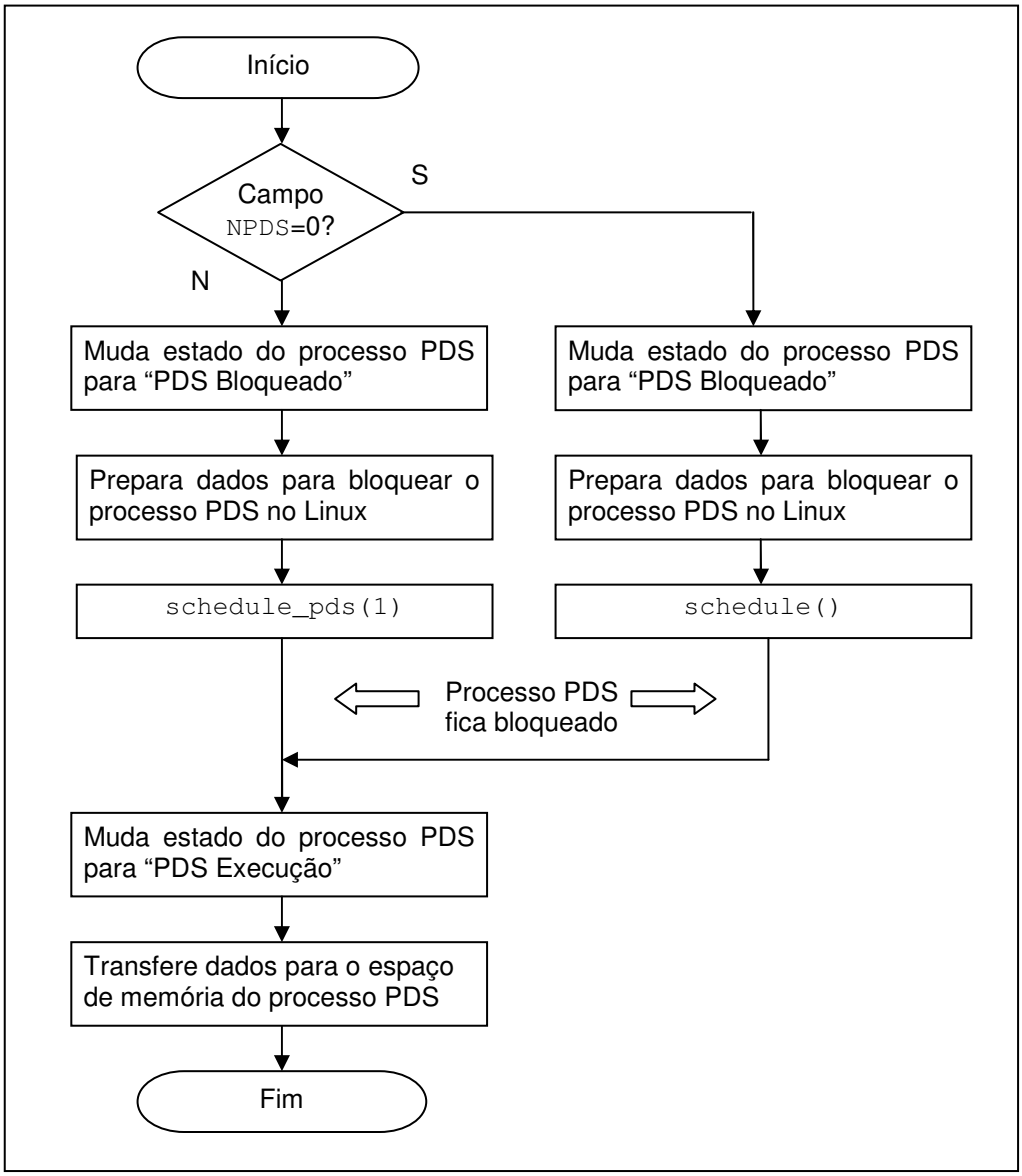

Figura 7.8 – Diagrama de blocos da função `driver_le_amostra`

A existência dos dois ramos no diagrama de blocos da figura 7.8 é justificada como segue:

1- A primeira vez que processo PDS executa a função de leitura, o campo `NPDS` tem valor igual a 0 (zero), que não é um valor válido para um ponteiro. Isto decorre pelo fato que ainda nenhuma IRQa foi tratada e, portanto, não há um processo NPDS para ser retomado após o tratamento da amostra. Conseqüentemente deve-se chamar o escalonador do Linux `schedule()`. Pelo explicado, o ramo da direita na figura 7.8 só é executado uma única vez.

2- Nas leituras subseqüentes, o processo PDS é colocado no estado pronto pelo tratador da IRQa, que é executado às custas de um processo NPDS.

Após o tratamento da IRQA, um ponteiro para o descritor do processo NPDS é armazenado no campo `NPDS` da estrutura e, na continuação, o processo PDS é colocado em execução para que a amostra seja tratada (ver item 7.9.4).

Após o tratamento da amostra, quando o processo PDS é bloqueado pelo comando de leitura, o processo NPDS deve ser retomado, o que é feito pela rotina `schedule_pds(1)`.

7.11 Comparação do Linux 2.6.25 Original e Modificado (SO-PDS)

O sistema operacional modificado (SO-PDS) foi testado, como mostrado a seguir, de duas formas:

1- **Testes I** – Neste teste foram implementados no Linux 2.6.25 os seguintes métodos: Método 1, Método 2 e Método 5. Este primeiro sistema operacional modificado, chamado de SO-PDS.vA, possui algumas diferenças se comparado ao descrito nos itens anteriores que decorrem da não implementação do Método 3.

2- **Testes II** – Neste teste foram implementados no Linux 2.6.25 os seguintes métodos: Método 1, Método 2, Método 3 e Método 5. Este segundo sistema operacional modificado, descrito nos itens anteriores, foi chamado de SO-PDS.vB.

O aplicativo de medição da latência, chamado de Aplicativo de Gerenciamento no Capítulo 6, item 6.5.1, foi adaptado para funcionar em ambos os sistemas modificados. Basicamente o programa foi simplificado para medir só a latência total. Esta simplificação visou obter um arquivo executável menor do que o original para contornar as restrições da memória virtual (ver Capítulo 4, item 4.6.3).

7.11.1 Testes I

O SO-PDS.vA foi configurado para ter preempção do núcleo e os *daemons* padrão. O Aplicativo de Gerenciamento foi executado: 1) em modo gráfico, 2) em escalonamento de tempo real (Classe SCHED_FIFO) com prioridade máxima e 3) com carga. A metodologia e a máquina usadas foram as mesmas do Capítulo 6, item 6.6 e item 6.7. Os resultados obtidos para este teste, junto com os dados do Capítulo 6, item 6.7.10 para comparação, estão a seguir.

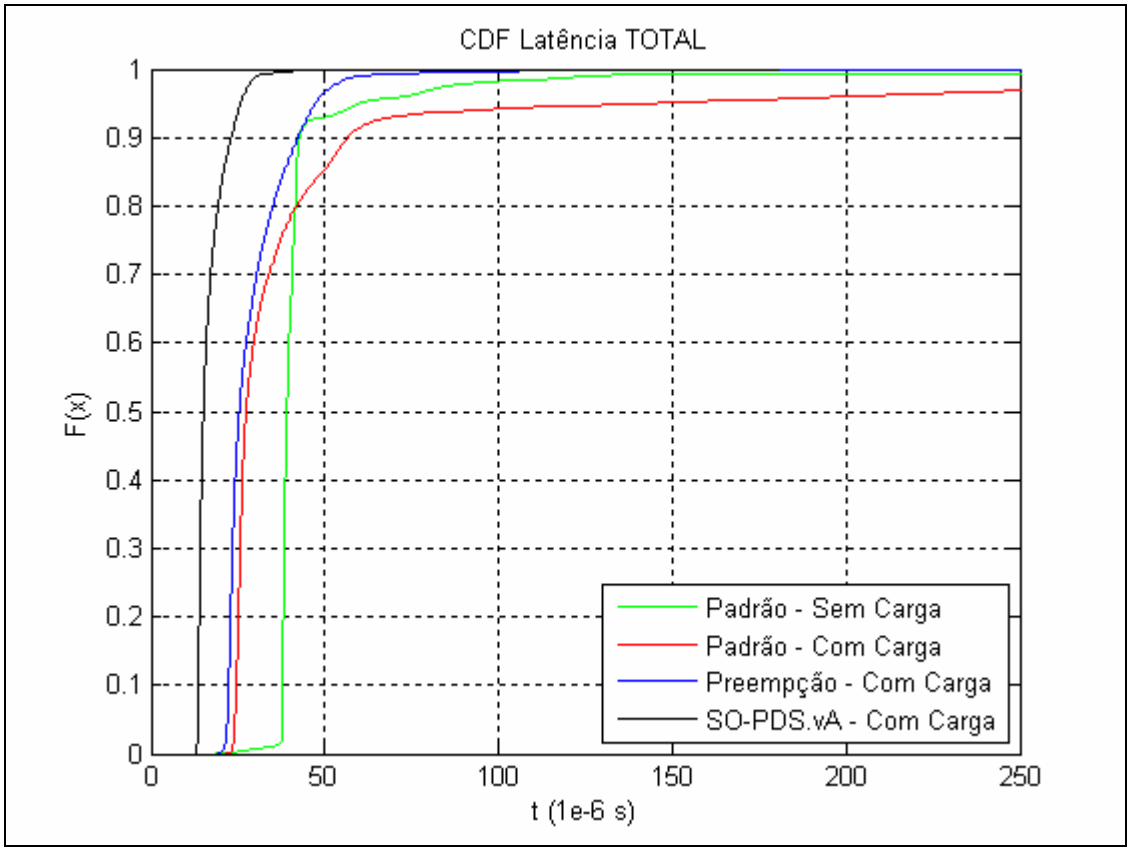

Figura 7.9 – CDF da latência total para o Teste I
Obs.: Há gráficos da CDF truncados.

Tabela 7.2 – Teste I Estatísticas da latência total

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**	**SO-PDS.vA Com Carga (µs)**
Média	44,6	61,60	29,60	16,83
Máximo	691,9	1,42e+3	536,7	181,0
Mínimo	18,90	21,64	19,09	12,58
Mediana	39,12	27,33	25,38	15,07

Análise:

Comparando a latência média obtida no teste “Preempção – Com Carga” com a obtida no “SO-PDS.vA – Com Carga” pode-se ver uma melhoria de 43,14 %. Esta melhoria pode ser explicada como segue:

1- O SO-PDS.vA utiliza os recursos de preempção e escalonamento de tempo real disponíveis no Linux 2.6.25 original.
2- O SO-PDS.vA elimina para a IRQa a sobrecarga de processamento do núcleo com preempção (ver Capítulo 6, item 6.7.2 e item 6.7.4).
3- O SO-PDS.vA não executa as *deferrable functions* ao final do tratamento da IRQa

7.11.2 Testes II

Durante os testes de depuração realizados no SO-PDS.vB observou-se:

1- Com o ambiente gráfico carregado, o sistema operacional mostrou-se instável, ou seja, apresentava travamentos intermitentes quando os programas de depuração eram executados.

2- Quando carregado em linha de comando, o sistema operacional apresentou alguns travamentos esporádicos.

Decorre desses problemas encontrados que a comparação entre o Linux 2.6.25 e o SO-PDS.vB só foi possível usando o tipo de medida chamado de sem carga no Capítulo 6, item 6.6.1. Nesse tipo de medida, o sistema operacional é carregado no modo linha de comando (*shell*) e o Aplicativo de Gerenciamento é executado.

Usando os parâmetros estabelecidos para o Aplicativo de Gerenciamento no Capítulo 6, item 6.6.2, foram obtidas 20.0000 medidas para o SO-PDS.vB. É importante observar que, mesmo sem carga, não se conseguiu obter de forma ininterrupta as 20.000 medidas, o que equivale a um tempo de processamento de 6 horas e 40 minutos aproximadamente.

Embora as condições experimentais não sejam rigorosamente as mesmas e o SO-PDS.vB ainda não esteja completamente estável, a seguir, as latências totais do SO-PDS.vB e do Linux 2.6.25 são comparadas. Os dados para a latência do Linux 2.6.25 são aqueles apresentados no Capítulo 6, item 6.7.10 com o nome “Padrão - Sem Carga”. Os resultados para ambos os sistemas operacionais são apresentados a seguir.

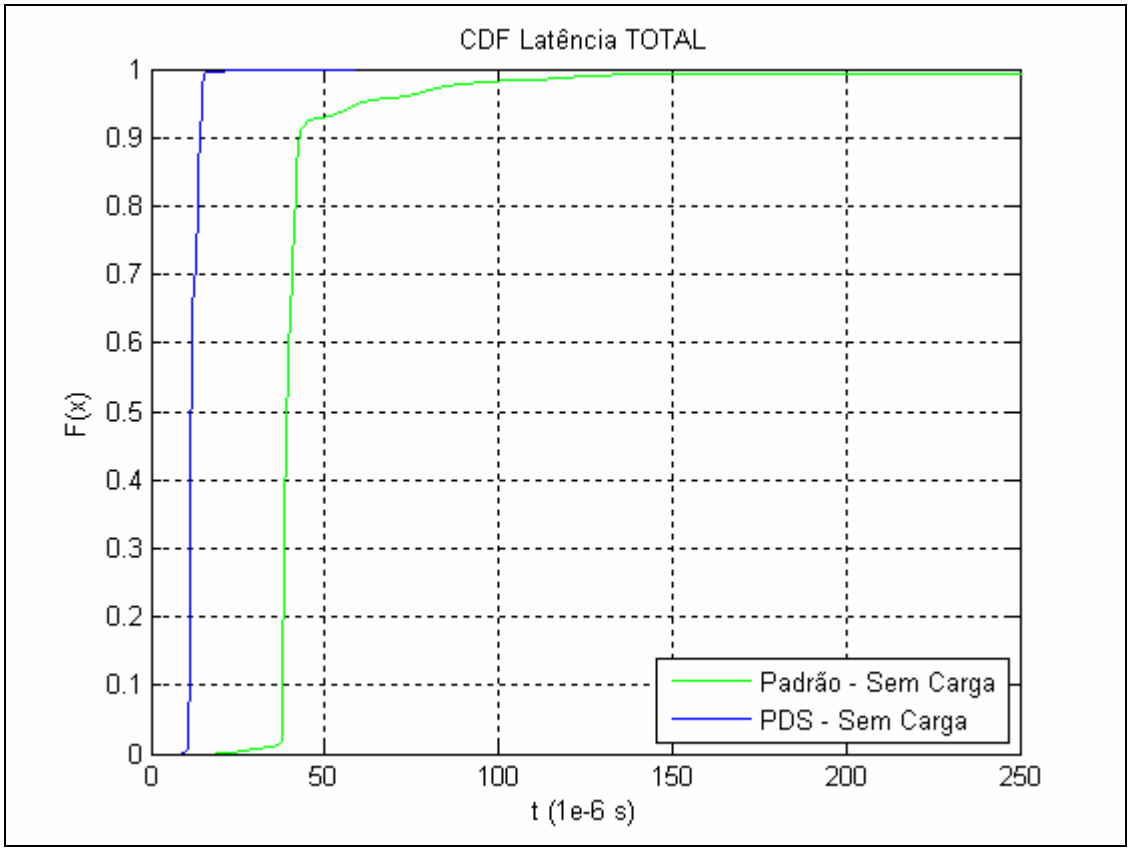

Figura 7.10 – CDF da latência total para o Teste II
Obs.: Apenas o gráfico “Padrão - Sem Carga” foi truncado

Tabela 7.3 – Teste II Estatísticas da latência total

	Padrão Sem Carga (µs)	**SO-PDS.vB Sem Carga (µs)**
Média	44,6	12,15
Máximo	691,9	59,24
Mínimo	18,90	8,762
Mediana	39,12	11,51

Analise dos Problemas:

1- Nos testes realizados sem carga foi constatado que o principal ponto de travamento é no final do processamento, quando se deve chavear do escalonador PDS para o escalonador Linux.

2- Nos testes realizados com carga existe a possibilidade que os travamentos decorram do programa Aplicativo de Gerenciamento não estar completamente em memória RAM durante a execução, em razão da memória virtual.

7.12 Resumo

Neste capítulo foram apresentados cinco métodos para melhorar o desempenho do Linux 2.6.25 em aplicações de processamento digital. Quatro desses métodos foram implementados para desenvolver um SO-PDS compatível com o Linux. O capítulo termina com uma comparação entre as latências totais do sistema modificado e do Linux 2.6.25.

Capítulo 8 – Conclusões e Propostas de Estudos Futuros

8.1 Conclusões

As conclusões obtidas pelo estudo realizado são:

1- É tecnicamente viável o desenvolvimento de um sistema operacional específico para o processamento digital de sinais em tempo real (SO-PDS). Mostrou-se que o desenvolvimento de um SO-PDS aproveitaria a estrutura particular dos aplicativos PDS para obter um melhor desempenho, se comparado aos sistemas operacionais de uso geral, na execução desse tipo de aplicativo em tempo real.

2- Demonstrou-se a limitação da arquitetura x86 e do Linux para o desenvolvimento do modo de operação *Pull-In*. Esta limitação decorre da forma de como o Linux configura os dois temporizadores programáveis à disposição do núcleo e, também, por ser este um número mínimo de temporizadores para aplicações PDS em tempo real. Esta não é uma conclusão exclusiva deste trabalho, haja vista que a Intel e Microsoft desenvolveram, para essa arquitetura, o *High Precision Event Timer* (HPET), que disponibiliza ao sistema operacional no mínimo 3 e no máximo 256 temporizadores programáveis.

3- Demonstrou-se quantitativamente a importância da preempção do núcleo em sistemas operacionais de tempo real.

4- Para o hardware testado, mesmo usando todos os recursos de tempo real do Linux 2.6.25, constatou-se uma freqüência de amostragem máxima da ordem de 1.000 Hz a qual pode ser considerada, de uma forma geral, baixa para aplicações de processamento digital de sinais em tempo real.

5- Mostrou-se que, usando soluções de software baseadas na estrutura de um programa PDS, o núcleo do Linux 2.6.25 pode ser modificado para se obter um melhor desempenho na execução de aplicativos PDS em tempo real.

8.2 Propostas de Estudo Futuro

1- Uma primeira proposta de pesquisa futura é tornar o núcleo modificado pelo método Troca Seletiva de Processo estável, haja vista que a implementação deste método causou travamentos do sistema operacional, principalmente, quando executado com os programas de carga (ver Capítulo 7, item 7.11.2).

2- Uma segunda proposta para pesquisa é a implementação do método chamado Desabilitação da Interrupção pelo Núcleo Postergada descrito no Capítulo 7, item 7.6.

3- Como foi mostrado no Capítulo 4, item 4.6.3, o Linux tem limitações para a execução do programa PDS em função de usar memória virtual. O Linux realiza a carga do programa sob demanda, faz alocação dinâmica de memória e remove as páginas menos usadas dos processos da memória RAM. Essas formas de gerenciamento da memória RAM do Linux não são apropriadas para um processo PDS como foi estabelecido pelo 21º Requisito. Uma outra proposta é a pesquisa de métodos para contornar essas limitações quando o programa PDS está sendo executado.

4- Foi visto no Capítulo 2, item 2.2.3 que existem duas formas de controlar a freqüência de amostragem: *Push-In* e *Pull-In*. Neste trabalho, devido às limitações do hardware disponível, não foi estudada a implementação do método *Pull-In* com base no *High Precision Event Timer* (HPET). Desta forma um tema para pesquisa é o desenvolvimento do controle da forma de amostragem *Pull-In* no Linux com base no HPET.

****

REFERÊNCIAS

1- HENNESSY, J. L.; PATTERSON D.A. **Arquitetura de Computadores**: Uma Abordagem Quantitativa. Rio de Janeiro: Campus, 2003. 827 p.

2- ANCEAU, F. Past, Present and Future of Microprocessors. **Design of Systems on a Chip**: Design and Test. Dordrecht: Springer, 2006. 236 p.

3- TOMPKINS, W.J. Evolution of microcomputer-based medical instrumentation. In: ANNUAL INTERNATIONAL CONFERENCE OF THE IEEE, ENGINEERING IN MEDICINE AND BIOLOGY SOCIETY, EMBC 2009, 2009, Minneapolis.

4- ENGLUND, C. **Speech recognition in the JAS 39 Gripen aircraft - adaptation to speech at different G-loads**. 2004. 57 p. Thesis (Master) - Royal Institute of Technology, Department of Speech, Music and Hearing, Estocolmo, 2004.

5- BARBU, M.; KAMINSKY, E.J.; TRAHAN, R.E. Fractional Fourier Transform for Sonar Signal Processing. In: PROCEEDINGS OF MTS/IEEE, 2005, v.2, p. 1630-1635, Washington D.C.

6- REALTIME LINUX FOUNDATION. A non-profit corporation for the real-time Linux community. Disponível em: <www.realtimelinuxfoudation.org>. Acesso em: 09 nov. 2010.

7- DIETRICH, S.; WALKER, D. The Evolution of Real-Time Linux. In: PROCEEDINGS OF 7TH REAL-TIME LINUX WORKSHOP, 2005, Lille, France.

8- LIN, K.; WANG, Y. The Design and Implementation of Real-Time Schedulers in RED-Linux. In: PROCEEDINGS OF THE IEEE, v.91, N° 7, p. 114-130, 2003.

9- YODAIKEN, V.; BARABANOV, M. A Real-Time Linux. In: PROCEEDINGS OF LINUX APPLICATIONS DEVELOPMENT AND DEPLOYMENT CONFERENCE (USELINUX), 1997, Anahein, USA.

10- SCHIESSER, G.; MCGUIRE N. Soft Real-Time Linux Overview. In: PROCEEDINGS OF 7TH REAL-TIME LINUX WORKSHOP, 2005, Lille, France.

11- THE LOCKFORMER COMPANY. Lockformer – Rollforming, Plasma Cutting, Notching, and other Equipment for the HVAC Industry. Disponível em: <www.lockformer.com> – Acesso em: 09 nov. 2010.

12- PALLI, G. BIAGIOTTI, L., MELCHIOR, C. An Open Source Distributed Platform for the Control of the PUMA 560 Manipulator. In: PROCEEDINGS OF 9TH REAL-TIME LINUX WORKSHOP, 2007, Linz, Austria.

13- THE LOCKFORNER COMPANY. **The Vulcan 1000D**. Lisle, Illinois, [2008]. 10 p.

14- LAPLANTE, P. A. **Real-Time Systems**: Design and Analysis. Piscataway, NJ: IEEE Press, 2004. 504 p.

15- SOMMERVILLE, I. **Software Engineering**. Essex, England: Pearson Education Limited, 2007. 840 p.

16- OPPENHEIM, A.; WILLSKY, A.; NAWAB, S. H. **Signals & Systems**. Upper Saddle River, NJ: Pearson Education, 1996. 957 p.

17- RIBEIRO, M. P.; BARRADAS, O. **Telecomunicações Sistemas Analógico-Digitais**. Rio de Janeiro: Livros Técnicos e Científicos, 1980. 1.174 p.

18- TANENBAUM A, S. **Sistemas Operacionais Modernos**. São Paulo: Pearson Prentice Hall, 2003.695 p.

19- KUMAGAI, J. Radio Revolutionaries. A Massachusetts company reinvents the cellular base station. **IEEE Spectrum**, jan. 2007.

20- VANU INC. Vanu, Inc. provides innovative wireless infrastructure solutions to enable operators to access new markets and new revenue streams. Disponível em: <www.vanu.com>. Acesso em: 09 nov. 2010.

21- VT MERLIN COMMUNICATIONS. **DRM Software Radio – User Manual Version 1.3**. Southampton, [2005], 31 p.

22- ASIC DESIGN SERVICES. **PCILite Datasheet**. Midrand, [2004], 58 p.

23- KEITHLEY INSTRUMENTS. Keithley Instruments Inc. A greater measure of confidence. Disponível em: <www.keithley.com>. Acesso em: 26 jan. 2011.

24- BOVET P, D.; CESATI, M. **Understanding the Linux Kernel**. Sebastopol, CA: O'Reilly Media Inc., 2005. 923 p.

25- ABENI, L.; GOEL, A.; KRASIC, C.; SNOW, J.; WALPOLE, J. A Measurement-Based Analysis of the Real-Time Performance of Linux. In: PROCEEDINGS OF THE 8TH IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM (RTAS), 2002.

26- SAVAGE, S.; TOKUDA, H. RT-match timers: Exporting time to the user. In: PROCEEDINGS OF USENIX 3RD MACH SYMPOSIUM, 1993.

27- NORTON, P. **Peter Norton's Complete Guide to PC Upgrades**. Indianapolis, Indiana: Sams Publishing, 1999. 751 p.

28- TORRES, G. **Hardware Curso Completo**. Rio de Janeiro: Axcel Books do Brasil Editora Ltda, 1999. 1.147 p.

29- CRUZ, G. M.; LIMA, G. Simulador de Escalonamento para Sistemas de Tempo Real. Disponível em: <http://www.lasid.ufba.br/publicacoes/artigos/giselia.pdf>. Acesso em: 15 Jun. 2011

30- AUDSLEY, N. C.; BURNS, A.; RICHARDSON, M. F.; WELLINGS, A. J. Stress: a simulator for hard real-time systems. **Software: Practice and Experience**, Vol. 24, p. 543-564, Jun. 1994.

31- KRAMP, T.; ADRIAN, M.; KOSTER, R. An open framework for real-time scheduling simulation. In: PROCEEDINGS OF 15 IPDPS WORKSHOPS ON PARALLEL AND DISTRIBUTED PROCESSING, 2000.

32- STORCH, M. F. A framework for the simulation of complex real-time systems. In: PROCEEDINGS OF IEEE REAL-TIME TECHNOLOGY AND APPLICATION SYMPOSIUM, Jun. 2006.

33- LIU, M.; SHAO, Z.; WANG, M.; WEI, H.; WANG, T. Implementing Hybrid Operating Systems with Two-Level Hardware Interrupts. In: PROCEEDINGS OF 28TH IEEE INTERNATIONAL REAL-TIME SYSTEMS SYMPOSIUM (RTSS 2007), p. 244-253, 2007.

34- STODOLSKI, D.; CHEN, J.B.; BERSHAD, B.N. Fast Interrupt Priority Management in Operating System Kernels. In: USENIX MICROKERNELS AND OTHER KERNEL ARCHITECTURES SYMPOSIUM, p. 105-110, 1993.

35- LEYVA-DEL-FOYO, L.E.; MEJIA-ALVAREZ, P.; NIZ, D. Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware. In: PROCEEDINGS OF 12TH IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM, p. 14-23, 2006.

36- INTERVALZERO INC. **Hard Real-Time with IntervalZero RTX on the Windows Platform**. Disponível em: <www.intevalzero.com>. Acesso em: 19 nov. 2010.

37- STRAUMANN, T. Open Source Real Time Operating Systems Overview. In: PROCEEDINGS OF 8TH INTERNATIONAL CONFERENCE ON ACCELERATOR & LARGE EXPERIMENTAL PHYSICS CONTROL SYSTEMS, p. 235-237, 2001, San Jose, California.

38- RALPH, M. Deferred Interrupt Processing Improves System Response. **Micro Digital Inc**., 2005.

39- KLIEDERMACHER, D. Minimizing Interrupt Response Time. **Information Quarterly**, v. 4, N° 1, p. 52-54, 2005.

40- SHI, H.; CAI, M.; DONG, J. Interrupt Synchronization Lock for Real-time Operating Systems. In: PROCEEDINGS OF 6TH IEEE INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY (CIT'06), 2006, Seoul, Korea.

41- HEURSCH, A.C.; GRAMBOW, D.; HORSTKOTTE, H. R. Steps Towards a Fully Preemptable Linux Kernel. In: PROCEEDINGS OF 27TH IEEE WORKSHOP ON REAL-TIME PROGRAMMING (WRTP'03), 2003, Lagow, Poland.

42- WANG, Y.; LIN, K. Implementing a General Real-Time Scheduling Framework. In: PROCEEDINGS OF IEEE REAL-TIME SYSTEMS SYMPOSIUM, p. 246-255, 1999, Phoenix, AZ.

43- KANG, D.; LEE, W.; PARK, C. Dynamic Kernel Thread Scheduling for Real-Time Linux. In: PROCEEDINGS OF 8TH REAL-TIME LINUX WORKSHOP, 2006, Lanzhou, R. P. China.

44- DIAZ, A.; RIPOLL, I.; CRESPO, A.; BALBASTRE, P. A New Application-Defined Scheduling Implementation in RT Linux. In: PROCEEDINGS OF 6TH REAL-TIME LINUX WORKSHOP, 2004, Singapore.

45- MARCHAND, A.; SILLY-CHETTO, M. QoS and Aperiodic Tasks Scheduling for Real-Time Linux Applications. In: PROCEEDINGS OF 6TH REAL-TIME LINUX WORKSHOP, 2004, Singapore.

46- CLARK, W. Linux Scheduler Latency. **Red Hat, Inc.**, 2002.

47- MERCER, C. W.; TOKUDA, H. Preemptibility in Real-Time Operating Systems. In: PROCEEDINGS OF 13TH IEEE REAL-TIME SYSTEMS SYMPOSIUM, 1992.

48- REALTIME-LINUX PROJECTS - OSADL. Disponível em: <www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html> Acesso em: 26 jan. 2011.

49- STANKOVIC, J.A. Misconceptions about real-time computing: a serious for next-generation systems. **IEEE Computer**, vol 21, p. 10-19, oct. 1988.

50- REAL TIME LINUX FOUDATION, INC. Disponível em: <www.realtimelinuxfoundation.org> Acesso em: 01 Jun. 2011.

51- RTAI – REAL TIME APPLICATION INTERFACE OFFICIAL WEBSITE. Disponível em: <www.rtai.org> Acesso em: 01 Jun. 2011

52- GERUM P. Xenomai – Implementing a RTOS emulation framework on GNU/Linux. Disponível em: <http://www.xenomai.org/documentation>. Acesso em: 01 Jun. 2011.

53- OSADL - OPEN SOURCE AUTOMATION DEVELOPMENT LAB. Disponível em: <www.osadl.org>. Acesso em: 19 nov. 2010.

54- HIGH RESOLUTION TIMERS PROJECT. This is the home page for the high resolution timers project. Disponível em: <high-res-timers.sourceforge.net>. Acesso em: 26 jan. 2011.

55- COBERT. The high-resolution timer API. Disponível em: <lwn.net/artcles/167897>. Acesso em: 19 nov. 2010.

56- CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. **Linux Device Drivers**. Sebastopol, CA: O'Reilly Media Inc, 2005. 616 p.

57- BLALOCK G. Microprocessors Outperform DSPs 2:1. In: MICROPROCESSOR REPORT THE INSIDERS' GUIDE TO MICROPROCESSOR HARDWARE, v. 10, N° 17, dez. 1996.

58- BERKELEY DESIGN TECHNOLOGY, INC. A source of analysis, advice, and engineering for embedded processing technology and applications. Disponível em: <www.bdti.com>. Acesso em: 26 jan. 2011

59- INTEL CO. **Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1**. Denver, [2006]. 642 p.

60- INTEL CO. **8259A Programmable Interrupt Controller (8259A / 8269A-2)**. Denver, [1988]. 24 p.

61- INTEL CO. **82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)**. Denver, [1996]. 20 p.

62- INTEL CO. **IA-PC HPET (High Precision Event Timers) Specification**. Denver, [2004]. 33 p

63- REGINER P.; LIMA G.; BARRETO L. Avaliação do Determinismo Temporal no Tratamento de Interrupções em Plataformas de Tempo Real Linux. In: Anais do XXVIII Congresso do SBC – WSO Workshop de Sistemas Operacionais, Belém do Pará, PA, Jul. 2008.

64- RODRIGUEZ S. A.; BURT P. M. S. A Latency Model of Linux 2.6 for Digital Signal Processing in Real Time. In: PROCEEDINGS OF 12TH REAL-TIME LINUX WORKSHOP, Nairobi, Kenya, oct. 2010.

65- SILBERSCHATZ, A.; GALUIN, P.B.; GAGNE, G. **Operating System Concepts 7th Edition**. Danvers, MA: John Wiley & Sons Inc, 2005. 921p.

66- COSTA NETO, P. L. O. **Estatística**. São Paulo: Edgard Blücher, 1977. 264 p.

67- HELENE, O. A. M.; VANIN, V. R. **Tratamento Estatístico de Dados em Física Experimental**. São Paulo: Edgard Blücher, 1981. 105 p.

68- DEITEL, H. M.; DEITEL P. J.; CHOFFNESS, D.R. **Sistema Operacionais**. São Paulo: Pearson Prentice Hall, 2005. 760 p.

69- INTEL CO. **Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture**. Denver, [2006], 468 p.

70- INTEL CO. **Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2**. Denver, [2006], 624 p.

71- STALLINGS W. **Arquitetura e Organização de Computadores**. 5ª Edição. São Paulo: Pearson Prentice Hall, 2002.

****

Apêndice A - Uma Visão Geral do Núcleo do Linux

A.1 Introdução

Este apêndice apresenta uma visão geral do núcleo do Linux. O núcleo de um sistema operacional fornece um ambiente de execução para os programas. O núcleo deve fornecer uma série de serviços e suas correspondentes interfaces que são usadas pelas aplicações, haja vista que estas não interagem diretamente com os recursos de hardware. Este texto tomou como base as referências (18), (24), (56), (68).

A.2 O Modelo Processo/Núcleo

Uma CPU pode funcionar em vários modos de operação. Estes modos de operação são usados para estabelecer diferentes níveis de privilégios aos programas em execução no sistema. Por exemplo, os microprocessadores da família x86 têm quatro modos de operação, no entanto, o Linux nesta plataforma só usa dois: o Modo Usuário e o Modo Supervisor[1].

Um programa em execução tem associado a ele um processo no sistema operacional. Quando o processo está em Modo Usuário, não pode acessar diretamente as estruturas ou rotinas do núcleo. Quando o processo passa ao Modo Supervisor, esta restrição não é mais aplicada. Cada modelo de CPU fornece instruções específicas para alternar entre os vários modos de execução. Normalmente um programa é executado em Modo Usuário e vai para o Modo Supervisor apenas quando requisita um serviço fornecido pelo núcleo. Quando termina de atender a requisição do programa, o núcleo o coloca de volta ao Modo Usuário.

[1] O Modo Supervisor é também conhecido na literatura como Modo Núcleo (*Kernel Mode*) e Modo Protegido.

O núcleo em si não é um processo, mas um gerenciador de processos. Este modelo processo/núcleo assume que, para requisitar um serviço do núcleo, o processo usa uma forma específica de programação denominada chamada ao sistema. Uma chamada ao sistema possui um grupo de parâmetros que identifica o serviço requisitado. A chamada ao sistema é a responsável por executar a instrução para trocar de Modo Usuário para Modo Supervisor. A figura A.1 mostra alguns exemplos de chamadas ao sistema que fazem parte do padrão POSIX, o qual é atendido pelo Linux.

Chamada	**Descrição**
`pid=fork()`	Criar um processo filho idêntico ao processo pai. Onde: o `pid` - número do processo
`s=stat(nome, &buf)`	Obter a informação do estado do arquivo. Onde: o `nome` - nome do arquivo o `&buff` - endereço do buffer
`s=chdir(dirnome)`	Alterar o diretório de trabalho. Onde: o `dirnome` - nome do diretório
`s=rmdir(nome)`	Remover um diretório vazio. Onde: o `nome` - nome do diretório
`s=kill(pid, sinal)`	Enviar um sinal a um processo: o `pid` - número do processo o `sinal` - código do sinal
Obs.: A variável `s` é o código de retorno que volta o valor -1 se ocorrer um erro.	

Figura A.1 – Algumas chamadas ao sistema do padrão POSIX
Adaptado da referência (18)

Além dos processos de usuários, o Linux inclui processos privilegiados chamados threads do núcleo (*kernel threads*) que serão vistos no item A.5.

Uma rotina do núcleo pode ser chamada das seguintes formas:

- Um processo executa uma chamada ao sistema.

- A CPU que executa o processo sinaliza uma exceção, que é uma condição incomum como, por exemplo, uma instrução inválida. O núcleo trata a exceção à custa do processo que a causou.

- Um dispositivo periférico sinaliza uma interrupção à CPU para notificá-la de algum evento como, por exemplo: início de uma operação de E/S, mudança de estado ou término de uma operação de E/S. Cada sinal de interrupção é tratado pelo núcleo por meio de uma rotina chamada tratador de interrupções.

- Como uma thread do núcleo é executada em Modo Supervisor, a mesma é considerada parte do núcleo.

A.3 Fluxo de Controle do Núcleo

Um conceito encontrado na literatura sobre sistemas operacionais é o de fluxo de controle do núcleo (*kernel control path*), o qual pode ser definido como segue.

Quando o núcleo está atendendo uma interrupção, uma exceção ou uma chamada ao sistema, executa um conjunto de instruções do núcleo (ex: tratador de interrupções). A execução deste conjunto de instruções é chamada de fluxo de controle do núcleo.

A.4 Processos

Para poder gerenciar os diversos processos do sistema, o núcleo cria para cada uma deles uma estrutura, chamada de descritor de processo, na qual as informações sobre o processo são armazenadas. Por exemplo, quando o núcleo pára a execução de um processo, salva o conteúdo dos vários registradores do microprocessador no descritor de processo.

Entre as informações armazenadas tem-se:

- O contador de programa (PCo - *program counter*) e os registradores do ponteiro da pilha (SP - *stack pointer*).
- Os registradores de uso geral.
- Os registradores de ponto flutuante.
- Os registradores de controle do processo, que contêm informações sobre o estado da CPU.
- Os registradores de gerenciamento de memória, que mantém o controle da memória RAM usada pelo processo.

Quando o núcleo retoma a execução de um processo, usa os campos do descritor do processo para carregar os registradores. Como o endereço da próxima instrução a ser executada foi salvo no descritor de processo, o processo pode ser re-iniciado.

De forma geral, em um sistema operacional, um processo pode estar em vários estados. O número de estados e as nomenclaturas usadas para designá-los podem variar com a literatura consultada. Neste texto será usado um número reduzido de estados cujas nomenclaturas, baseada na referência (18), se encontram, a seguir, na figura A.2.

Estado	**Descrição**
Pronto	O processo está esperando para entrar em execução
Bloqueado	O processo está suspenso até que alguma condição se torne verdadeira
Execução	O processo está de posse da CPU

Figura A.2 – Estados de um processo em um sistema operacional

A seguir serão vistos os estados de um processo no Linux, que por ser um sistema operacional real, utiliza um número maior de estados do que os colocados na figura A.2.

Quando um processo não está em execução na CPU, ele está esperando por algum evento. O Linux distingue os vários estados de espera, através de filas de

descritores de processos. Cada fila, que pode estar vazia, corresponde ao conjunto de processos esperando por algum evento específico. Os estados de um processo no Linux podem ser vistos na figura A.3.

Estado	Descrição
TASK_RUNNING	O processo está de posse da CPU ou esperando para entrar em execução.
TASK_INTERRUPTIBLE	O processo está suspenso até que alguma condição se torne verdadeira.
TASK_UNINTERRUPTIBLE	O processo está suspenso até que alguma condição se torne verdadeira. Exceto que o recebimento de um sinal não altera o estado do processo. Este estado raramente é usado.
TASK_STOPPED	A execução do processo foi parada.
TASK_TRACED	A execução do processo foi parada pelo depurador.
EXIT_ZOMBIE	A execução do processo foi terminada, mas o processo pai ainda não sinalizou para encerrar o processo.
EXIT_DEAD	O estado final, o processo está sendo removido do sistema.

Figura A.3 – Estados de um processo no Linux

A.5 Threads do Núcleo (*Kernel Threads*)

O Linux delega algumas tarefas críticas (ex: conexões de rede e remoção de páginas não usadas) a processos que rodam de forma intermitente e em segundo plano (*background*), haja vista que, desta forma, o usuário final tem uma melhor reposta. Como a maioria dessas tarefas é executada somente no Modo Supervisor, os sistemas operacionais modernos as delegam a processos chamados de threads do núcleo. No Linux, as threads do núcleo têm as seguintes características:

- Rodam somente no Modo Supervisor enquanto os processos alternam entre Modo Supervisor e Modo Usuário.
- Como rodam no Modo Supervisor usam o espaço de endereçamento do núcleo.
- Não interagem com o usuário e não requerem terminais.
- São normalmente criadas durante a carga do sistema e permanecem ativas no sistema até que o mesmo seja desligado.

A.6 ***Deferrable Functions***

Deferrable functions ou *softirq* é um mecanismo do Linux que permite o armazenamento de rotinas do núcleo em uma fila, para serem executadas a posterior. O núcleo periodicamente verifica na fila se há rotinas ativas a serem executadas. No núcleo do Linux esses pontos de verificação variam com a versão e com a arquitetura. (24)

O Linux usa esse mecanismo para, por exemplo, verificar o término dos temporizadores de software. (24). Outro uso comum é para o tratamento das interrupções como explicado a seguir.

De forma geral, o tratamento de uma interrupção pode ser dividido em tarefas críticas e não críticas. As tarefas críticas normalmente devem ser executadas rapidamente e com as interrupções desabilitadas, enquanto as tarefas não críticas podem ser executadas em um segundo momento e com as interrupções habilitadas. Se todas as tarefas (críticas e não críticas) fossem executadas sequencialmente o sistema ficaria com as interrupções desabilitadas por um tempo maior do que o necessário. Uma forma de contornar este problema é executar as tarefas não críticas por meio de uma *deferrable function*.

Para mais detalhes de como o Linux implementa este mecanismo ver a referência (24).

A.7 **Núcleo Reentrante**

Partes do núcleo do Linux são reentrantes. Isto significa que vários processos podem estar sendo executados em Modo Supervisor simultaneamente. É claro que em sistemas com um único processador, apenas um processo está sendo executado e vários estão suspensos em Modo Supervisor aguardando a CPU ou o término de alguma operação de E/S.

Uma maneira de implementar a reentrância do núcleo é desenvolvendo funções ou rotinas que manipulem apenas variáveis locais e não estruturas de

dados globais, de forma que, instâncias dessas funções possam ser executadas simultaneamente. Sempre lembrando que em sistemas com um único processador, apenas uma instância está sendo executada e várias estão suspensas. As funções e as rotinas com esta característica são chamadas de reentrantes.

Sistemas operacionais de tempo real são geralmente desenvolvidos usando funções reentrantes. Em sistemas operacionais de uso geral, o núcleo reentrante pode ser obtido associando funções não reentrantes com mecanismo de bloqueio para assegurar que apenas um processo execute uma função não reentrante.

Se uma interrupção ocorre, um sistema operacional reentrante pode suspender o processo corrente mesmo que ele esteja em Modo Supervisor. Essa capacidade é muito útil para que os dispositivos externos sejam atendidos de forma rápida, o que explica importância do núcleo reentrante em aplicações de tempo real.

A.8 Preempção do Núcleo

Um primeiro conceito para um núcleo com preempção seria: é o núcleo que permite a troca de processo quando o processo a ser substituído está executando alguma função do núcleo, ou seja, está em Modo Supervisor. No entanto devido à complexidade do Linux, este conceito deve ser ampliado, como explicado a seguir.

Em núcleos com ou sem preempção, um processo em Modo Supervisor pode voluntariamente liberar a CPU, ocasionando uma troca de processo. Esta troca é chamada de **troca de processo planejada**.

O que diferencia um núcleo com preempção de um núcleo sem preempção, é a forma de como um processo em Modo Supervisor reage a eventos assíncronos (ex: uma interrupção) que induzam a uma troca de processo (ex: o tratador de interrupção liberou para execução um processo de maior prioridade). Esta troca é chamada de **troca de processo forçada**. Em um núcleo sem preempção, o processo corrente em Modo Supervisor só pode sofrer uma troca de processo forçada quando voltar ao Modo Usuário. Já em um núcleo com preempção, o processo corrente em Modo Supervisor pode sofrer uma troca de processo forçada.

A principal motivação para ter um núcleo com preempção é reduzir a latência de despacho quando se está executando um processo em Modo Supervisor. A latência de despacho pode ser definida como: o intervalo de tempo entre o instante em que um processo passa para o estado de pronto e o instante em que passa para o estado de em execução. Nos núcleos com preempção, caso um processo de maior prioridade tenha sido colocado no estado pronto por algum evento (ex: interrupção), o fluxo de controle do núcleo em execução é interrompido e, efetua-se a troca do processo. Nos núcleos sem preempção, após o tratamento do evento, o fluxo de controle do núcleo é retomado e, somente, efetua-se a troca quando o processo passa ao Modo Usuário. Do descrito, fica claro a vantagem do núcleo com preempção, por ter menor latência de despacho, em sistemas operacionais que tratam com processos de tempo real.

No Linux 2.6.25 pode-se configurar o núcleo com ou sem preempção dependendo do parâmetro de compilação `CONFIG_PREEMPT`. Isto é feito porque a preempção do núcleo implica em maior carga de processamento.

É importante observar que, apesar do Linux 2.6.1 ter preempção em partes do núcleo, até a versão 2.6.25 o núcleo não é todo com preempção, ou seja, quando o núcleo está executando certas rotinas não pode sofrer preempção. Os casos em que isso ocorre são:

- O núcleo está executando uma rotina de serviço da interrupção.
- O núcleo está executando uma *deferrable function*.
- A preempção do núcleo foi desativada como, por exemplo, durante a execução das travas rotativas (*spin locks*).

A.9 Módulos

Uma dos recursos do Linux é a capacidade de estender, quando em execução, o conjunto de serviços oferecidos pelo núcleo. Isto significa que se podem adicionar funcionalidades ao núcleo, assim como remover funcionalidades, enquanto o sistema está carregado e sendo executado. (56)

Cada pedaço de código que pode ser adicionado ao núcleo em execução é chamado de módulo. O módulo é um código objeto, ou seja, um código que não está lincado a um código executável completo. Um módulo pode ser lincado ao núcleo usando-se o programa `insmod` e removido usando-se o programa `rmmod`. (56)

Os módulos normalmente são divididos em classes, embora não seja uma divisão rígida e possa haver mais de um critério de classificação. Dentre os critérios para a classificação, um critério normalmente usado tem como base o dispositivo que o módulo trata. O Linux divide os dispositivos em três tipos fundamentais e como normalmente um módulo trata apenas um desses tipos, essa mesma divisão é usada para os módulos. (56)

A classificação por tipo de dispositivo é mostrada a seguir:

- Módulo para Caractere (*char module*) - são módulos que tratam dispositivos que efetuam operações de E/S através de um fluxo de bytes (*stream of bytes*). Estes dispositivos são conhecidos como dispositivos de caracteres (*character devices*).

- Módulo para Blocos (*block module*) – são módulos que tratam dispositivos que efetuam operações de E/S através de blocos. Estes dispositivos são conhecidos como dispositivos de bloco (*block devices*).

- Módulo de Rede (*network module*) – são módulos que tratam os dispositivos que fazem as interfaces de rede (*network interfaces*) e, portanto, processam pacotes de dados trocados com a rede.

****

Apêndice B - Introdução às Interrupções

B.1 Introdução

Este apêndice traz material básico sobre interrupções na arquitetura x86. O tratamento das exceções não será abordado. Maiores informações sobre interrupções e exceções podem ser obtidas nas referências (18), (24) e (68); que foram usadas como base para este texto. Informações adicionais sobre o hardware da arquitetura x86 podem ser encontradas nas referências (27), (28), (59), (60), (61), (69) e (70).

B.2 Definição

Uma interrupção é normalmente definida como um evento que altera a seqüência das instruções executadas por um microprocessador. Esses eventos correspondem a sinais elétricos gerados por circuitos de hardware internos ou externos à CPU.

Normalmente as interrupções são divididas em síncronas e assíncronas:

- Interrupções síncronas são geradas pela unidade de controle[1] da CPU durante a execução de uma instrução. São denominadas de síncronas, pois a unidade de controle somente as lança após terminar a execução da instrução.

- Interrupções assíncronas são geradas por dispositivos de hardware em instantes de tempo arbitrários em relação ao relógio da CPU.

O manual da Intel (69) chama as interrupções síncronas de exceções e as assíncronas de interrupções. Estas serão as denominações usadas neste texto.

[1] Informações sobre a arquitetura básica dos microprocessadores podem ser encontradas na referência (71).

Interrupções são normalmente geradas por temporizadores internos ou dispositivos de E/S (Entrada/Saída).

Exceções são normalmente geradas por erros de programação (ex: divisão por zero) ou alguma condição incomum no fluxo do processo (ex: falta de página ou requisição de um serviço do núcleo).

B.3 Classificação das Interrupções

A documentação da Intel (59) classifica as interrupções como mostrado abaixo:

- Interrupções mascaráveis - todas as requisições de interrupção (*Interrupt ReQuest*) ou IRQ lançadas por um dispositivo de E/S geram interrupções que podem ser mascaradas. Este tipo de interrupção pode estar em dois estados: mascarada ou não mascarada. Uma interrupção mascarada é ignorada pela unidade de controle da CPU enquanto permanecer nesse estado. Decorre deste fato que uma interrupção mascarada é também chamada de interrupção desabilitada[2].

- Interrupções não mascaráveis - apenas uns poucos eventos (ex: falhas de hardware) geram interrupções que não podem ser mascaradas, ou seja, interrupções que sempre são atendidas pela unidade de controle da CPU.

As interrupções mascaráveis também são chamadas na literatura de interrupções E/S[3] e podem ser divididas em dois grupos:

- Externas - são interrupções geradas por dispositivos externos ao microprocessador.

[2] Neste texto será usado o termo interrupção desabilitada por ser mais comum na literatura de sistemas operacionais.
[3] Neste texto será usado o termo interrupção E/S por ser mais comum na literatura de sistemas operacionais.

- Locais - são interrupções geradas por dispositivos que fazem parte do conjunto do microprocessador. No caso específico da arquitetura x86, são interrupções geradas pelo controlador programável de interrupções avançado local (*Advanced Programmable Interrupt Controller*) ou APIC Local. O controlador programável de interrupção será tratado no item B.6.2.

Esta divisão se justifica, pois o Linux trata as interrupções E/S locais e as interrupções E/S externas de forma diferente, como pode ser visto no Apêndice C ou na referência (24).

Cada interrupção é identificada por um número que varia de 0 a 255. A Intel chama esse número de vetor. Os vetores das interrupções não mascaráveis são fixos, enquanto os vetores das interrupções mascaráveis (interrupções E/S) podem ser configurados dependendo do hardware (ver item B.6).

B.4 Aspectos Básicos das Interrupções

Quando um sinal de interrupção é levantado, a CPU deve interromper o processo corrente e trocar para uma nova rotina. Isto é feito basicamente em dois passos:

1- Alguns dados são colocados na pilha do núcleo como, por exemplo, o contador de programa (`EIP`) e o registrador `CS`[4].
2- O contador de programa é carregado com o endereço da rotina que trata o tipo de interrupção que gerou o sinal.

O código que trata uma interrupção não é executado como um processo, mas como um fluxo de controle do núcleo que é executado à custa do tempo de computação do processo corrente. Isto é feito porque um fluxo de controle do núcleo requer menos tempo de processamento para ser carregado e descarregado.

[4] O registrador `CS` é o registrador de segmento de código (*code segment*) que aponta para o segmento de memória que contém as instruções do programa.

O tratamento de interrupções é uma das tarefas mais sensíveis realizadas pelo núcleo do sistema operacional. Isto se deve aos seguintes motivos:

- O objetivo do núcleo é tratar a interrupção da forma mais rápida possível. Para atender a este requisito, as ações executadas pelo núcleo, para tratar uma interrupção, são divididas em dois grupos: as ações críticas que são realizadas rapidamente e as ações não críticas que são realizadas em um segundo momento.

- Como as interrupções chegam a qualquer momento, pode existir o caso em que uma interrupção seja sinalizada enquanto outra interrupção, anteriormente sinalizada, estiver sendo tratada. Para que a interrupção sinalizada possa ser tratada pelo núcleo, o tratador de interrupções deve ser codificado para permitir a execução de forma aninhada. Além disso, quando o último fluxo de controle do núcleo terminar, o núcleo deve ser capaz de retomar a execução do processo corrente ou trocar de processo se for necessário. O aninhamento de interrupções é tratado no item B.5.

É importante observar que nem sempre o núcleo aceita tratar uma nova interrupção E/S enquanto estiver tratando uma interrupção previamente sinalizada. Também existem regiões críticas no código do núcleo que são executadas com as interrupções desabilitadas.

B.5 Interrupções Aninhadas

Neste item é analisado o conceito de interrupções aninhadas em um sistema operacional.

Cada interrupção tratada acarreta a execução de um fluxo de controle do núcleo à custa do tempo de computação do processo corrente. As primeiras

instruções deste fluxo de controle do núcleo salvam os valores dos registradores da CPU na pilha do núcleo[5] (*kernel stack*) e as últimas instruções os restauram.

Quando esse fluxo de controle do núcleo pode ser arbitrariamente aninhado, o tratador de interrupção pode ser interrompido por outra interrupção, gerando assim fluxos de controle aninhados como mostrado na figura B.1. Na figura, pode-se ver que a última instrução de um fluxo de controle do núcleo nem sempre retoma a execução do processo em Modo Usuário.

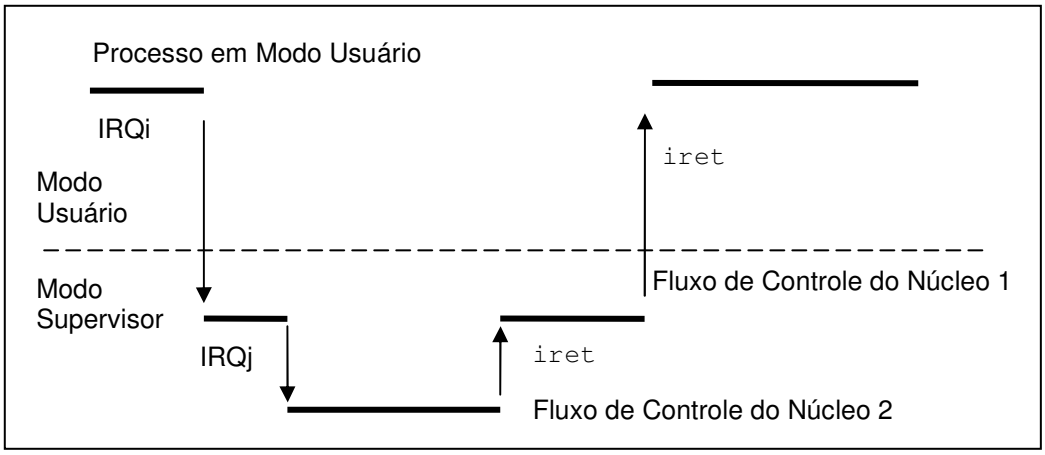

Figura B.1 - Um exemplo da execução de fluxos de controle do núcleo aninhados

No Linux, o preço a pagar, para permitir esse aninhamento é que não pode ser efetuada uma troca de processo até que todos os fluxos de controle do núcleo tenham terminado. Isto ocorre, pois as informações necessárias para retomar um fluxo de controle do núcleo estão armazenadas na pilha do núcleo a qual é rigidamente ligada ao processo corrente.

Uma interrupção pode interromper o tratamento de uma exceção ou de uma outra interrupção. Já uma exceção só pode interromper o tratamento de outra exceção, sendo que, no máximo só duas exceções podem ser aninhadas[6].

[5] A família x86 permite determinar uma pilha para cada forma de operação do microprocessador.
[6] Ver a referência (24) para uma explicação mais detalhada.

O Linux permite aninhar interrupções com a finalidade de liberar mais rapidamente o controlador programável de interrupções e, conseqüentemente, os controladores de dispositivos. No entanto duas observações são importantes:

1- O Linux só permite o aninhamento da interrupção *n* se a rotina (ou conjunto de rotinas[7]) que trata a interrupção *n* assim o permitir, ou seja, a rotina que trata uma interrupção determina se vai ou não ser interrompida pela sinalização de outra interrupção. Esta informação fica armazenada em uma estrutura chamada descritor de interrupção[8] (Ver Apêndice C, item C.4).

2- Mesmo com o aninhamento de interrupções, o Linux realiza um conjunto de instruções com as interrupções desabilitadas durante o tratamento de uma interrupção.

B.6 Interrupções – Hardware da família x86

Neste item são vistos aspectos do hardware da família x86 relacionados às interrupções. Basicamente são analisados os dois principais controladores programáveis de interrupções desta arquitetura.

B.6.1 Sistema Monoprocessador com 8259A

Cada dispositivo de hardware normalmente requisita interrupções E/S através de um único sinal de saída denominado sinal de requisição de interrupção ou IRQ (*Interrupt ReQuest*)[9].

Todos os sinais de IRQ são ligados aos pinos de entrada de um circuito chamado Controlador Programável de Interrupções (*Programmable Interrupt Controller*) ou PIC que realiza as seguintes operações:

---

[7] O Linux permite o compartilhamento de interrupções E/S externas e, portanto, pode existir mais de uma rotina para tratar uma interrupção (ver Apêndice C, item C.2.2).

[8] O Linux não armazena o endereço das rotinas que tratam as interrupções E/S externas na IDT (ver item B.6.4 e Apêndice C, item C.5).

[9] Dispositivos mais sofisticados podem usar vários sinais de IRQ como, por exemplo, as placas PCI que podem usar até 4 sinais de IRQ.

1- Monitora as vias de IRQ, verificando se algum sinal foi ativado. Se uma ou mais vias são ativadas, a que tiver o pino com menor número é selecionada.

2- Quando o sinal é ativado, o PIC executa as seguintes operações:

   2.1 Converte o sinal recebido no vetor correspondente.
   2.2 Coloca o vetor na porta de E/S do controlador de interrupções para que a CPU o possa ler através do barramento de dados.
   2.3 Ativa um sinal no pino `INTR` do microprocessador gerando a interrupção.
   2.4 Espera até que a CPU escreva em uma da portas de E/S do PIC, respondendo o sinal de interrupção. Quando isso ocorre o pino `INTR` é liberado.

As vias de IRQ são sequencialmente numeradas iniciando em 0 (zero), portanto, a primeira via de IRQ é usualmente chamada de IRQ0. O padrão da Intel é que o vetor associado com a IRQn é n+32.

Uma via qualquer de IRQ pode ser ativada e desativada programando o PIC. Uma interrupção desativada no PIC não é perdida, o PIC a envia à CPU assim que a via é ativada novamente. Esta propriedade do PIC é usada pelos tratadores de interrupção para processar de forma seriada as IRQs de mesmo tipo.

Ativar e desativar as IRQs no PIC é diferente de, respectivamente, não mascarar e mascarar as interrupções. Quando o bit `IF` do registrador `eflag` é igual a zero, as interrupções mascaráveis sinalizadas pelo PIC são ignoradas temporariamente pelo microprocessador. Por ser mais comum, neste texto usa-se habilitar e desabilitar IRQs para se referenciar ao não mascaramento / mascaramento de uma interrupção.

Nas versões mais antigas dos microcomputadores de padrão IBM-PC, o PIC é implementado associando dois 8259A em cascata. Cada 8259A trata até oito IRQs

diferentes. Como o pino INT do PIC escravo é conectado ao pino IRQ2 do PIC mestre, o número de sinais de IRQ está limitado a 15 (quinze). Com este arranjo, a via de saída do mestre pode ser ligada de forma direta ao pino INTR da CPU.

B.6.2 Sistema Multiprocessador com APIC

Em sistemas com dois ou mais processadores, a saída do PIC não pode ser ligada de forma direta ao pino INTR do processador. Além desta limitação de hardware, para explorar de forma eficiente o paralelismo da arquitetura *Single Memory Multi Processor* (SMMP) deve ser possível direcionar as interrupções para qualquer CPU do sistema.

Por essas razões, a Intel introduziu no Pentium III um novo componente denominado de Controlador Programável de Interrupções Avançado (*Advanced Programmable Interrupt Controller*) Local ou APIC Local. Este circuito é uma versão mais avançada do Controlador Programável de Interrupções 8259A. Todos os atuais microprocessadores 80x86 incluem um APIC Local.

Cada APIC Local possui: um registrador de 32 bits, um relógio interno, um dispositivo temporizador interno e duas vias adicionais de IRQ (LINT0 e LINT1) reservadas para interrupções internas do APIC Local. Neste texto, as interrupções internas geradas pelo APIC Local são chamadas de interrupções E/S locais (ver item B.3).

Todos os APICs Locais são conectados a um APIC E/S externo, formando um sistema múltiplo, como pode ser visto de forma esquemática na figura B.2.

Na figura, um barramento de comunicação interliga os APIC Locais com o APIC E/S. As vias de IRQ dos dispositivos externos são ligadas ao APIC E/S, o qual funciona como um roteador com relação aos APICs Locais. Neste texto, as interrupções geradas pelos dispositivos externos são chamadas de interrupções E/S externas.

O APIC E/S consiste de: um conjunto de 24 vias de IRQ, uma Tabela de Redirecionamento de Interrupções (*Interrupt Redirection Table*) com 24 entradas, registradores programáveis e um mecanismo para enviar e receber mensagens entre os APICs Locais pelo barramento de comunicação.

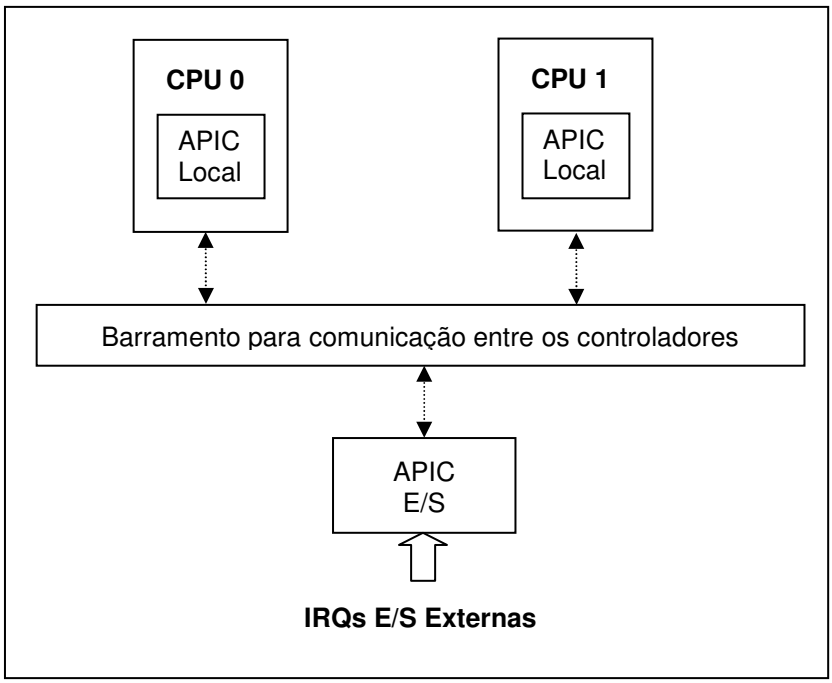

Figura B.2 – Diagrama de bloco de um sistema multi-APIC

Diferentemente do 8259A, a prioridade da interrupção não está relacionada com o número do pino. Cada entrada da Tabela de Redirecionamento pode ser individualmente programada para indicar o vetor de interrupção, sua prioridade, o processador de destino e como o processador é selecionado.

Um processador pode ser selecionado de duas formas:

- Distribuição Estática - o sinal de IRQ é direcionado para o APIC Local constante na respectiva entrada da Tabela de Redirecionamento. A interrupção pode ser entregue para uma CPU, para um subconjunto ou para todas as CPUs.

- Distribuição Dinâmica - o sinal de IRQ é direcionado para o APIC Local do processador que está executando o processo de menor prioridade. No caso de dois ou mais processadores possuírem a mesma prioridade é usada uma técnica chamada arbitragem para selecionar um dos processadores.

B.6.3 Sistema Monoprocessador com APIC

Atualmente sistemas monoprocessador incluem um circuito APIC E/S que pode ser configurado de duas formas diferentes:

- Emulando um controlador 8259A conectado à CPU - O APIC Local é desabilitado. As duas vias locais `LINT0` e `LINT1`, que geram o sinal de IRQ, são configuradas como os pinos `INTR` e `NMI` respectivamente.

- Como um APIC E/S - O APIC Local é habilitado e todas as interrupções são recebidas através do APIC E/S.

B.6.4 A Tabela Descritora das Interrupções

Uma tabela do sistema operacional chamada Tabela Descritora das Interrupções (*Interrupt Descriptor Table*) ou IDT associa cada vetor de interrupção com o correspondente tratador de interrupção. A IDT deve ser corretamente inicializada pelo núcleo antes que o sistema operacional habilite as interrupções.

O registrador `idtr` da CPU permite que a IDT seja colocada em qualquer posição da memória. O registrador especifica tanto o endereço base como o tamanho da tabela. Da mesma forma que a IDT, o núcleo deve incicializar corretamente o registrador antes de habilitar as interrupções.

B.7 Como o Hardware Trata uma Interrupção

Neste item é descrito, de forma breve, como a unidade de controle do microprocessador trata uma interrupção.

B.7.1 Antes de Tratar a Interrupção

Ao terminar a execução de uma instrução, o par de registradores `cs` e `eip` contém o endereço lógico da próxima instrução a ser executada. Antes de tratar essa instrução, a unidade de controle verifica se uma interrupção ou uma exceção ocorreu durante a execução da instrução que foi finalizada.

Se uma interrupção foi sinalizada, a unidade de controle realiza as seguintes operações:

1- Determina o vetor *i* ($0 \leq i \leq 255$) associado à interrupção.
2- Lê a entrada *i* da IDT que é referenciada pelo registrador `idtr`.
3- Determina o endereço base do segmento de memória que contém o tratador da interrupção.
4- Verifica se o tratador da interrupção tem um nível de privilégio menor do que o processo que está de posse do processador[10]. Se o nível de privilégio do tratador for menor, a exceção Proteção Geral (*general-protection exception* ou `#GP`) é sinalizada, pois o tratador da interrupção não pode ter um nível de privilégio menor do que o processo que está sendo executado[11].
5- Verifica se uma troca de nível de privilégios está sendo efetuada. No caso de haver uma troca de nível, a unidade de controle passa a usar a pilha associada ao novo nível de privilégio.
6- Salva os valores dos registradores `eflags`, `cs` e `eip` na pilha.

[10] A família x86 reserva dois bits para designar o nível de prioridade do modo de operação. O valor 0 (zero) corresponde ao maior nível de privilégio que é normalmente usado pelo núcleo. O valor 3 (três) corresponde ao menor nível de privilégios. Obs.: O texto se refere ao nível em si e não ao valor dos bits associados a cada nível.

[11] Maiores informações sobre as formas de operação e níveis de privilégios na família x86 podem ser encontradas na referência (69).

7- Carrega `cs` e `eip` com os valores correspondentes encontrados na entrada *i* da IDT. Estes valores definem o endereço lógico da primeira instrução do tratador da interrupção. Este passo equivale a um salto para o tratador.

B.7.2 Depois de Tratar a Interrupção

Depois que a interrupção é tratada, o tratador da interrupção deve devolver a posse da CPU ao processo corrente que foi interrompido. Isto é feito através da instrução `iret` que faz com que a unidade de controle realize as seguintes operações:

1- Carrega os registradores `cs`, `eip` e `eflags` com os valores salvos na pilha.
2- Verifica se o tratador da interrupção tem o mesmo nível de privilégio do que o processo corrente. Se não é necessária uma troca de nível de privilégio, não há mais nenhuma operação a ser executada, caso contrário continua no passo 3.
3- Efetua a troca da pilha para a pilha associada ao nível de privilégio do processo corrente.
4- Examina os registradores de segmentos `ds`, `es`, `fs` e `gs`. Se algum deles estiver referenciando um segmento com um nível de privilégio maior do que o do processo corrente, o conteúdo dos registradores é limpo. Isso é feito para impedir que programas maliciosos acessem segmentos de memória de maior nível de privilégio.

****

Anexo C - Interrupções no Linux

C.1 Introdução

Este apêndice apresenta como o Linux trata as interrupções E/S. O material deste anexo está baseado na referência (24) e nos programas fontes do Linux 2.6.25. As interrupções E/S neste trabalho foram divididas em dois grupos:

1- Interrupções E/S Externas - são interrupções geradas por dispositivos externos de entrada e saída.

2- Interrupções E/S Locais - são interrupções geradas pelo APIC Local e podem ser divididas em dois subgrupos:

- Internas – são interrupções geradas pelo temporizador e por sensores como, por exemplo, o sensor de temperatura.
- Inter-processadores (*interprocessor interrupts*) ou IPIs - são interrupções geradas por mensagens entre os processadores.

Como este texto não foca em sistemas SMMP, a forma como o Linux trata as IPIs não é tratada neste anexo.

C.2 Interrupções E/S no Linux – Aspectos Gerais

Alguns aspectos básicos das interrupções E/S no Linux são apresentados neste item.

C.2.1 Divisão das Ações

Em relação às interrupções E/S é preciso considerar que nem todas as ações necessárias para tratar uma interrupção têm o mesmo nível de urgência. Operações que requeiram muito processamento e que não sejam urgentes não

devem ser executadas no tratador de interrupções, haja vista que enquanto o tratador de interrupções está sendo executado, geralmente, as interrupções são temporariamente ignoradas.

Outro aspecto importante é que o processo, à custa do qual o tratador de interrupções está sendo executado, deve estar sempre no estado em execução, ou seja, o manipulador de interrupções não pode executar processamentos que bloqueiam o processo corrente como, por exemplo, acesso a disco.

Para atender os requisitos acima, o Linux divide as ações para tratar uma interrupção em três classes:

1- Críticas – são executadas no tratador de interrupções de forma imediata com as interrupções E/S desabilitadas.
2- Não Críticas – são executadas pelo tratador de interrupções com as interrupções E/S habilitadas.
3- Não Críticas e Postergáveis – são executadas por meio das funções chamadas *deferrable functions*.

C.2.2 Compartilhamento do Sinal de Interrupção

Atualmente nos microcomputadores existem dispositivos que podem compartilhar o sinal de interrupção e dispositivos que não compartilham o sinal de interrupção. Um exemplo de dispositivo que compartilha o sinal de interrupção são os dispositivos que usam o barramento padrão PCI. Um exemplo de dispositivo que não compartilha o sinal de interrupção são os dispositivos mais antigos que usam o barramento padrão ISA.

O exposto acima traz duas implicações. A primeira é que o vetor de interrupção nem sempre é suficiente para identificar o dispositivo. A segunda é que o tratador de interrupções deve ser flexível o suficiente para atender ambos os tipos de dispositivos.

O Linux atende o compartilhamento do sinal de interrupção, implementando o tratador de interrupções em duas camadas. A primeira camada identifica a interrupção IRQn através do vetor de interrupção e acessa uma estrutura do núcleo que contém as informações relativas à interrupção. A estrutura possui uma lista ligada com os endereços de todas as rotinas que podem tratar a interrupção. A segunda camada executa todas essas rotinas, chamadas de rotinas de serviço de interrupção (*Interrupt Service Routine*) ou ISR. Cada rotina verifica se o dispositivo que ela atende sinalizou, em sendo o caso, trata a interrupção, caso contrário ignora a interrupção.

Outro recurso disponível no Linux é a alocação dinâmica de IRQ, ou seja, o sinal de interrupção é associado ao dispositivo no último momento possível. Desta forma, um mesmo vetor de IRQ pode ser usado por diferentes dispositivos desde que não seja de forma simultânea. É importante observar que em microcomputadores compatíveis com o IBM-PC nem todos os dispositivos suportam alocação dinâmica de interrupção.

C.2.3 Funcionamento Básico

Independente do tipo de dispositivo que sinalizou a interrupção, todos os tratadores de interrupção E/S executam as mesmas quatro operações básicas:

1- Salva o valor do vetor da interrupção e o conteúdo dos registradores na pilha do núcleo.
2- Envia um sinal de reconhecimento (*ack*) ao PIC.
3- Executa todas as ISRs associadas à interrupção.
4- Termina o tratamento da interrupção e volta para o endereço de retorno.

As operações realizadas pelo núcleo do sistema operacional após o termino do tratador da interrupção dependem de vários fatores como, por exemplo, o modo de execução do processo corrente, quando da sinalização da IRQ, e se a interrupção solicitou, ou não, a chamada ao escalonador (ver item C.6).

C.3 Aspectos Específicos da Arquitetura x86

A figura C.1 mostra como os vetores de interrupção são alocados na arquitetura x86. Por exemplo, as interrupções E/S externas podem ser alocadas a qualquer vetor no intervalo de 32 a 238, no entanto, pode-se ver na tabela que o Linux usa o vetor 128 para implementar a chamada ao sistema.

Intervalo Vetor	Uso
0-19	Interrupções não mascaráveis e exceções
20-31	Reservados pela Intel
32-127	Interrupções E/S externas
128	Exceção programada para chamada ao sistema
129-238	Interrupções E/S externas
239	Interrupção do temporizador do APIC Local
240	Interrupção do sensor térmico do APIC Local
241-250	Reservados pelo Linux para uso futuro
251-253	Interrupção entre processadores
254	Interrupção de erro do APIC Local
255	Interrupção de sinal espúrio do APIC Local

Figura C.1 - Vetores de interrupção no Linux para a arquitetura x86

A arquitetura compatível com o IBM-PC requer que alguns dispositivos sejam estaticamente conectados a vias específicas de IRQ. Em particular:

1- O temporizador tem que ser conectado ao sinal IRQ0.
2- O escravo do PIC 8259A deve ser conectado ao sinal IRQ2.
3- O co-processador matemático deve ser conectado ao sinal IRQ13[1].

Geralmente um dispositivo de hardware só pode ser conectado a um número limitado de sinais de IRQs. São três as maneiras de selecionar um sinal de IRQ para um dispositivo que seja configurável:

1- Por ajuste de *jumpers* na placa do dispositivo.
2- Por intermédio de um programa utilitário que configura o dispositivo.

[1] Apesar dos processadores 80x86 não mais usarem este dispositivo, o Linux dá suporte ao mesmo.

3- Por um protocolo de hardware executado na carga do sistema. Os dispositivos de hardware (ou periféricos) declaram quais IRQs podem usar e o valor final é determinada de forma a reduzir o conflito. Uma vez o processo terminado, o dispositivo pode determinar qual a IRQ que lhe foi assinalada usando funções que leia algumas portas E/S do mesmo.

É importante observar que o núcleo deve descobrir o número correspondente de cada dispositivo de E/S antes de habilitar as interrupções. Isto é feito na carga do sistema durante a inicialização do dispositivo. A figura C.2 mostra um arranjo possível de IRQs e dispositivos em uma máquina compatível com o IBM-PC.

IRQ	Vetor IRQ	Dispositivo
0	32	Temporizador ou Sinal de Relógio
1	33	Teclado
2	34	PIC Escravo
3	35	Segunda Porta Serial
4	36	Primeira Porta Serial
6	38	Unidade de Disquete
8	40	Relógio da Placa Mãe
10	42	Interface de Rede
11	43	Porta USB, Placa de Som
12	44	Mouse PS/2
13	45	Co-processador Matemático
14	46	Controlador de Disco EIDE – 1ª cadeia
15	47	Controlador de Disco EIDE – 2ª cadeia

Figura C.2 - Um exemplo de configuração do sinal IRQ e seus dispositivos em um microcomputador compatível com o IBM_PC

A figura C.2 mostra que a toda interrupção E/S externas associa-se dois números diferentes: 1) o número do sinal da interrupção e 2) o número do vetor. Por sua vez, a figura C.1 e a figura 2 mostram que a toda interrupção E/S local associa-se um único número: o número do vetor. Para evitar erros de interpretação estabelece-se a seguinte notação:

- IRQn – Interrupção cujo sinal é o número n.
- IRQvn – Interrupção cujo vetor é o número n.

Pela nomenclatura estabelecida tem-se para o teclado: IRQ1 e IRQv33. Já para o temporizador do APIC Local tem-se apenas IRQv239, pois esta interrupção e uma interrupção E/S local.

C.4 Estruturas do Linux para Tratar as Interrupções E/S Externas

Neste item são descritas as principais estruturas usadas pelo Linux para o tratamento de uma interrupção E/S externa. O Linux usa essas estruturas para atender ao compartilhamento de IRQs e para tratar todas as interrupções E/S externas de uma maneira uniforme[2].

C.4.1 Descritor de Interrupção

O Linux estabelece, para cada vetor de interrupção, um descritor que contém todas as informações para o tratamento da interrupção. Os descritores estão agrupados em um vetor. O descritor é chamado `irq_desc_t` e a matriz `irq_desc`. A figura C.3 mostra os principais campos do descritor, a estrutura completa pode ser encontrada em `linux/include/linux/irq.h`.

Campo	Descrição
`handler`	Ponteiro para a rotina que manipula a interrupção (ver item C.4.3)
`irq_chip`	Ponteiro para o objeto que trata o controlador programável de interrupções (ver item C.4.4)
`action`	Ponteiro para o primeiro nó da lista ligada que contém informações sobre as ISRs associadas à interrupção (ver item C.4.5)
`status`	Status da interrupção
`depth`	Tem valor zero se a interrupção está desbloqueada e positivo se está bloqueada
`irq_count`	Contador do número de interrupções
`irqs_unhandled`	Contador de interrupções não tratadas
`lock`	Trava rotativa (*Spin Lock*[3]) para serializar o acesso ao descritor da IRQ e ao PIC

Figura C.3 – Principais campos do descritor `irq_desc_t`

[2] A IDT é tratada no Apêndice B, item B 6.4 e mais informações podem ser encontradas em (24).
[3] Informações sobre a trava rotativa (*Spin Lock*) podem ser encontradas na referência (24).

Uma interrupção é dita não tratada se, como o próprio nome diz, não pôde ser tratada pelo núcleo. Isto ocorre por duas razões: não há ISR cadastrada para o vetor da interrupção ou nenhuma das ISRs cadastradas, associadas ao vetor, reporta ao sistema operacional o tratamento da interrupção. O núcleo verifica o número de interrupções não tratadas como forma de detectar e bloquear interrupções espúrias geradas pelo mau funcionamento do hardware. Como um sinal de interrupção pode ser compartilhado por mais de um dispositivo, o núcleo não pode desabilitar uma IRQ à primeira interrupção espúria. O descritor da interrupção possui dois campos (`irq_count` e `irqs_unhandled`) para estimar o percentual de interrupções espúrias. Quando este percentual ultrapassa um limite pré-estabelecido, o núcleo bloqueia a interrupção. O bloqueio é efetuado marcando o bit `IRQ_DISABLED` do campo `status` do descritor.

O campo `status` possui uma série de bits que sinalizam os vários estados de uma interrupção, os mais relevantes são mostrados na figura C.4.

Campo	Descrição
`IRQ_PROGRESS`	O tratador da interrupção está sendo executado
`IRQ_DISABLED`	O sinal de interrupção está bloqueado
`IRQ_PENDING`	Uma IRQ ocorreu, essa ocorrência foi sinalizada ao PIC, mas o núcleo ainda não a tratou
`IRQ_REPLAY`	O sinal de interrupção foi desabilitado, mas existe uma IRQ que ainda não foi tratada

Figura C.4 – Alguns bits do campo `status` e seus respectivos significados

C.4.2 Abstração do Hardware

Alguns campos do descritor estão diretamente relacionados à forma como o Linux implementa a abstração do hardware para tratar uma interrupção. Sendo um sistema operacional multi-plataforma, o Linux precisa abstrair dois aspectos das interrupções ligados ao hardware: o tipo da interrupção e o controlador de interrupções.

Foram identificados quatro tipos de interrupção para os sistemas monoprocessador: nível (*level*), borda (*edge*), simples (*simple*) e rápida[4]. Com o aparecimento dos sistemas multipocessador foi introduzido um quinto tipo: por CPU (*per CPU*). A diferença entre os tipos reside no momento em que o controlador programável de interrupções deve ser acionado e com qual instrução. Para não ter um tratador de interrupções com vários desvios e para otimizar o fluxo das instruções, foram desenvolvidas rotinas específicas para cada tipo de interrupção. O endereço da rotina, que trata uma interrupção específica, é armazenado no campo `handler` do descritor. Ver item C.4.3.

Para tratar os diferentes modelos de controladores de interrupção, o Linux usa um enfoque orientado a objeto, sendo que o endereço do objeto que manipula o controlador é armazenado no campo `irq_chip` do descritor. Ver item C.4.4.

C.4.3 O Manipulador da Interrupção (*Handler*)

Como foi visto no item anterior (C.4.2), o fluxo das operações a serem executadas no controlador programável de interrupções depende do tipo de interrupção, sendo que o Linux tem uma rotina específica para cada um dos cincos tipos. A seguir na figura C.5 tem-se o tipo de interrupção, a rotina e o arquivo do núcleo no qual pode ser encontrada. Quando o *driver* ou o núcleo configura o descritor da interrupção deve selecionar uma das seis rotinas para cadastrar no campo `handler`.

Tipo	**Rotina / Arquivo**
Nível	`handle_level_irq – linux/kernel/irq/chip.c`
Borda	`handle_edge_irq – linux/kernel/irq/chip.c`
Simples	`handle_simple_irq – linux/kernel/irq/chip.c`
Rápida	`handle_fasteoi_irq – linux/kernel/irq/chip.c`
Por CPU	`handle_percpu_irq – linux/kernel/irq/chip.c`
Padrão x86	`__do_IRQ()-linux/kernel/irq/handle.c`

Figura C.5 – Lista de manipuladores de interrupções no Linux

[4] A classificação das interrupções em tipos foi realizada originalmente por Russell King.

Na figura pode-se observar uma sexta rotina (__do_IRQ) que é o super-manipulador (*super-handler*) do Linux, o qual pode tratar todos os tipos de interrupções normalmente encontrados na arquitetura x86. Este é o manipulador padrão usado pelo núcleo do Linux nesta arquitetura. O parâmetro de compilação CONFIG_GENERIC_HARDIRQS_NO__DO_IRQ determina se o super-manipulador será ou não usado pelo núcleo.

C.4.4 Objeto para o Controlador Programável de Interrupções

O Linux deve dar suporte a vários controladores de interrupção como, por exemplo, o 8259A[5] e o APIC Local[5]. Para poder tratar os controladores de uma forma uniforme, o Linux usa um enfoque orientado a objeto, ou seja, usa uma estrutura chamada irq_chip (também chamada hw_interrupt_type) que contém os endereços das rotinas, as quais executam as operações suportadas pelo controlador de interrupções instalado na máquina.

A vantagem deste enfoque é que os *drivers* não precisam saber qual o tipo de controlador está instalado no sistema. A figura C.6 mostra um exemplo da estrutura irq_chip configurada para o 8259A, cuja definição pode ser encontrada em linux/include/linux/irq.h.

```
001    static struct irq_chip i8259A_chip = {
002            .name           = "XT-PIC",
003            .mask           = disable_8259A_irq,
004            .unmask= enable_8259A_irq,
005            .mask_ack       = mask_and_ack_8259A,
006    };
```

Figura C.6 – Exemplo da estrutura irq_chip para o controlador 8259A
Extraído de linux/arch/x86/kernel/i8259_32.c

C.4.5 Descritor para a Rotina de Serviço de Interrupção

Como já foi visto, no descritor da interrupção existe o ponteiro action (ver figura C.3) que aponta para o primeiro nó de uma lista ligada, a qual contém todas

[5] Ver o Anexo B, item B.6 para informações sobre estes controladores programáveis de interrupção.

as rotinas para tratar os dispositivos que sinalizam essa interrupção. A cada interrupção, o núcleo percorre a lista e executa todas as rotinas de serviço de interrupção. Cada nó da lista é um descritor `irq_action` o qual contém as informações necessárias para executar a ISR. Os principais campos do descritor podem ser vistos na figura C.7. A descrição desta estrutura por ser encontrada em `linux/include/linux/interrupt.h`.

Campo	Descrição
`handler`	Ponteiro para a ISR de um dispositivo de E/S
`flags`	Campo que descreve a relação entre o sinal de IRQ e o dispositivo de E/S (ver figura C.8)
`name`	Nome do dispositivo de E/S
`dev_id`	Identifica o dispositivo de E/S (para uso do dispositivo de E/S)
`next`	Ponteiro para o próximo nó da lista ligada
`irq`	Sinal de IRQ
`dir`	Ponteiro para um descritor no diretório `/proc/irq/n` associado com a IRQn.

Figura C.7 – Principais campos do descritor `irq_action`

Campo	Descrição
`SA_INTERRUPT`	O tratador da interrupção é executado com as interrupções desabilitadas.
`SA_SHIRQ`	O dispositivo permite o compartilhamento do sinal de interrupção com outros dispositivos.
`SA_SAMPLE_RANDOM`	O dispositivo de E/S pode ser fonte de eventos aleatórios para o gerador de números aleatórios do núcleo.

Figura C.8 - Bits do campo `flags` e seus significados

C.4.6 Resumo Estruturas de Dados

Na continuação a figura C.9 mostra esquematicamente como se relacionam as estruturas de dados do Linux para o tratamento das interrupções.

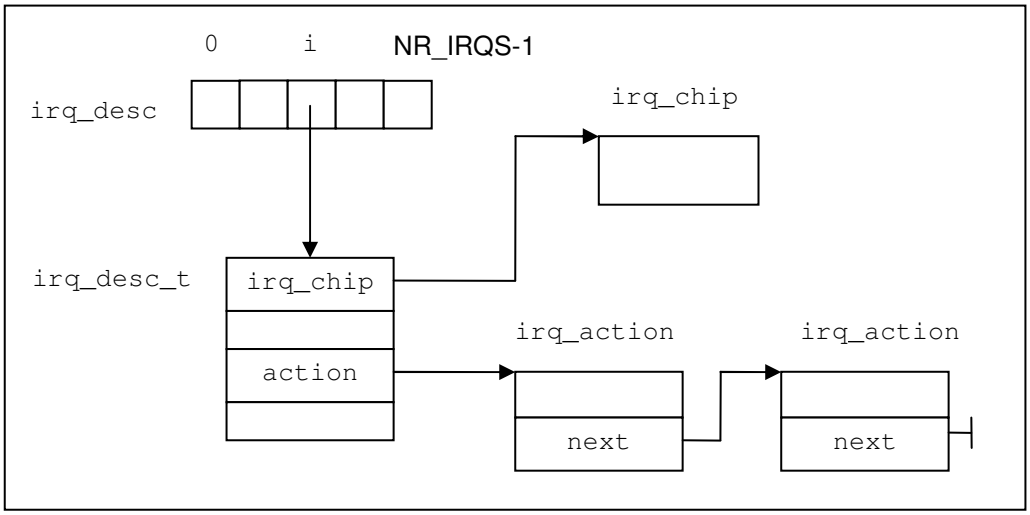

Figura C.9 - Estruturas de dados para o tratamento das interrupções

C.4.7 Pilhas Múltiplas Modo Supervisor

No Linux, o descritor do processo (`thread_info`) é ligado à pilha do núcleo em uma estrutura de dados (`thread_union`) que ocupa uma, ou duas páginas, de memória dependendo da forma como o sistema operacional foi compilado.

Se o tamanho da estrutura `thread_union` é de 8 KB (duas páginas), a pilha do núcleo do processo corrente é usada para todo tipo de fluxo de controle do núcleo: exceções, interrupções e *deferrable functions*.

Se o tamanho da estrutura é de 4 KB (uma página), o núcleo faz o uso de três pilhas diferentes dependendo do tipo de fluxo de controle do núcleo. Quando o núcleo trata uma exceção usa a pilha Exceção, quando trata uma interrupção usa a pilha IRQ e quando executa uma *deferrable function* usa a pilha Soft IRQ. Cada uma dessas pilhas está contida em uma página de memória. Em sistemas SMMP, todos os processadores possuem suas próprias pilhas.

C.5 Tratamento de Interrupção E/S Externa – 1ª Parte

A análise de como o Linux trata as interrupções E/S externas será dividida em duas partes. A primeira parte, neste item C.5, foca o tratamento da interrupção

em si e a segunda parte, no item C.6, analisa as operações efetuadas pelo núcleo após a interrupção ser tratada.

C.5.1 Salvando os Registradores

Quando a CPU recebe uma interrupção E/S, processa a instrução do endereço armazenado na entrada na IDT correspondente ao vetor da IRQ. No Linux, para as interrupções E/S externas, as entradas da IDT apontam para um mesmo endereço. Isto se justifica, pois no tratamento de uma interrupção a primeira tarefa é salvar o conteúdo dos registradores, a qual deve ser feita por um código em linguagem de montagem (*assembly language*). Desta forma, o Linux usa um único trecho para tratar o início comum a todas as interrupções E/S externas. O endereço que consta na IDT aponta para o bloco de instruções da figura C.10.

```
001     pushl $n-256
002     jmp common_interrupt
```

Figura C.10 – Bloco de instruções endereçado pela IDT para interrupções E/S externas Extraído da referência (24)

O código da figura salva na pilha o vetor da interrupção menos 256. O núcleo representa as IRQs através de números negativos, porque reserva os números positivos para identificar as chamadas ao sistema. Na linha 002 há um salto para a linha `common_interrupt`.

C.5.2 Análise da Linha: `common_interrupt`

A figura C.11 mostra o código que inicia na linha `common_interrupt`.

```
001     common_interrupt:
002             SAVE_ALL
003             TRACE_IRQS_OFF
004             movl %esp,%eax
005             call do_IRQ
006             jmp ret_from_intr
007             CFI_ENDPROC
```

Figura C.11 – Bloco de instruções que inicia o tratamento de uma interrupção E/S externa. Em `linux/arch/x86/kernel/entry_32.s`

1- SAVE_ALL (Linha 002)

Na figura C.11, linha 002, pode-se ver que é executada a macro `SAVE_ALL`. A figura C.12 a seguir mostra o código fonte desta macro. Nesta figura pode-se ver que os registradores são salvos de forma alinhada na pilha, exceto aqueles que já foram salvos automaticamente pela unidade de controle do microprocessador. A macro finaliza carregando o seletor do segmento de dados de usuário em `ds` e `es` (instruções 028 a 030).

```
001    #define SAVE_ALL
002            cld;
003            pushl %es;
004            CFI_ADJUST_CFA_OFFSET 4;
005            pushl %ds;
006            CFI_ADJUST_CFA_OFFSET 4;
007            pushl %eax;
008            CFI_ADJUST_CFA_OFFSET 4;
009            CFI_REL_OFFSET eax, 0;
010            pushl %ebp;
011            CFI_ADJUST_CFA_OFFSET 4;
012            CFI_REL_OFFSET ebp, 0;
013            pushl %edi;
014            CFI_ADJUST_CFA_OFFSET 4;
015            CFI_REL_OFFSET edi, 0;
016            pushl %esi;
017            CFI_ADJUST_CFA_OFFSET 4;
018            CFI_REL_OFFSET esi, 0;
019            pushl %edx;
020            CFI_ADJUST_CFA_OFFSET 4;
021            CFI_REL_OFFSET edx, 0;
022            pushl %ecx;
023            CFI_ADJUST_CFA_OFFSET 4;
024            CFI_REL_OFFSET ecx, 0;
025            pushl %ebx;
026            CFI_ADJUST_CFA_OFFSET 4;
027            CFI_REL_OFFSET ebx, 0;
028            movl $(__USER_DS), %edx;
029            movl %edx, %ds;
030            movl %edx, %es;
```

Figura C.12 – Macro `SAVE_ALL` que salva os registradores
Extraído de `linux/arch/x86/kernel/entry_32.s`

2– TRACE_IRQS_OFF (Linha 003)

Se a constante de compilação do núcleo `CONFIG_TRACE_IRQFLAGS` está definida, é executada a macro `TRACE_IRQS_OFF`. Esta macro junto com a macro `TRACE_IRQS_ON` fazem parte de um mecanismo para monitorar o estado das interrupções. Para maiores informações ver os arquivos:

- linux/include/asm-x86/irqflags.h
- linux/kernel/lockdep.c
- linux/Documentation/irqflags-tracing.txt

3- Outras Instruções (Linhas 003 a 006)

Na figura C.11, na linha 004 o endereço da pilha corrente é salvo no registrador `eax` e, na linha 005 a rotina `do_IRQ` é chamada. A linha 006 faz um desvio incondicional para a linha `ret_from_intr`, onde se inicia o que é chamado neste texto de retorno da interrupção. As operações executadas a partir desta linha serão vistas no item C.6. A rotina `do_IRQ` é analisada a seguir no item C.5.3.

C.5.3 Análise da Rotina: `do_IRQ()`

A rotina `do_IRQ()` será analisada por partes. O código encontra-se dividido em várias figuras, mas a numeração é seqüencial para toda a rotina. O código fonte da rotina pode ser encontrado em `linux/arch/x86/kernel/irq_32.c`.

1 - Definição de Variáveis

A rotina tem como parâmetro de entrada o endereço de uma estrutura que representa o conjunto dos registradores. Como o endereço da pilha foi salvo no registrador `eax` e os registradores foram salvos na pilha, a variável `regs` contém os valores de todos os registradores no instante da interrupção.

```
001    fastcall unsigned int do_IRQ(struct pt_regs *regs)
002    {
003     struct pt_regs *old_regs;
004     int irq = ~regs->orig_eax;
005     struct irq_desc *desc = irq_desc + irq;

006     #ifdef CONFIG_4KSTACKS
007        union irq_ctx *curctx, *irqctx;
008        u32 *isp;
009     #endif
```

Figura C.13 – Trecho da rotina `do_IRQ` (1ª Parte)

Na figura C.13 tem-se:

1- Na linha 004 é recuperado o vetor da interrupção.
2- Na linha 005 o descritor da interrupção é carregado em `desc`.
3- Nas linhas 006 a 009 é definida uma estrutura para a pilha IRQ (ver item C.4.7).

2 - Checando a Interrupção e Salvando os Registradores

```
010     if (unlikely((unsigned)irq >= NR_IRQS)) {
011     printk(KERN_EMERG "%s: cannot handle IRQ

        %d\n",__FUNCTION__, irq);
012             BUG();
013     }

014     old_regs = set_irq_regs(regs);
015     irq_enter();
```

Figura C.14 – Trecho da rotina `do_IRQ` (2ª Parte)

Na figura C.14 pode-se ver:

1- O bloco de linhas 010 a 013 verifica a validade do número da interrupção.
2- Na linha 014 os valores iniciais dos registradores são salvos na variável `old_regs`.
3- Na linha 015 é chamada a rotina `irq_enter()`.

2.1 – Análise da Rotina: irq_enter()

```
001     #define irq_enter()          \
002             do {                          \
003                     account_system_vtime(current);      \
004                     add_preempt_count(HARDIRQ_OFFSET);  \
005                     trace_hardirq_enter();              \
006             } while (0)
```

Figura C.15 – Definição da rotina `irq_enter`
Extraído de `linux/include/linux/hardirq.h`

Analisando o código da figura C.15:

1- A função da linha 003 não executa nenhuma operação na arquitetura x86 como pode ser verificado em:
   - `linux/include/linux/hardirq.h`

2- Na linha 004 a função `add_preempt_count` incrementa um contador do número de tratadores de interrupção aninhados. O contador é armazenado no campo `preempt_count` da estrutura `thread_info` do processo corrente. Ver os arquivos:
   - `linux/include/linux/preempt.h`
   - `linux/kernel/sched.c`

3- Na linha 005 dependendo do parâmetro de compilação `CONFIG_TRACE_IRQFLAGS`, o campo `hardirq_context` do descritor do processo corrente é incrementado de um. Ver o arquivo:
   - `linux /include/linux/irqflags.h`

3 – Verificando o Tamanho da Pilha

```
016    #ifdef CONFIG_DEBUG_STACKOVERFLOW
017     {
018      long esp;
019     __asm__ __volatile__("andl %%esp,%0" : "=r" (esp) : "0"
        (THREAD_SIZE - 1));
020     if (unlikely(esp < (sizeof(struct thread_info) +
        STACK_WARN))) {
021     printk("do_IRQ: stack overflow: %ld\n", esp - zeof(struct
        thread_info));
022      dump_stack();
023      }
024     }
025    #endif
```

Figura C.16 – Trecho da rotina `do_IRQ` (3ª Parte)

Na figura C.16 tem-se:

1- Dependendo do parâmetro `CONFIG_DEBUG_STACKOVERFLOW`, se o espaço livre da pilha for menor que 1KB envia uma mensagem à console.

4 – Trocando para a Pilha IRQ e Chamando o Manipulador

```
026     #ifdef CONFIG_4KSTACKS

027     curctx = (union irq_ctx *) current_thread_info();
028     irqctx = hardirq_ctx[smp_processor_id()];


029     if (curctx != irqctx) {
030     int arg1, arg2, ebx;

031             isp = (u32*) ((char*)irqctx + sizeof(*irqctx));
032             irqctx->tinfo.task = curctx->tinfo.task;
033             irqctx->tinfo.previous_esp = current_stack_pointer;

034             irqctx->tinfo.preempt_count =
                        (irqctx->tinfo.preempt_count & ~SOFTIRQ_MASK) |
                        (curctx->tinfo.preempt_count & SOFTIRQ_MASK);

035             asm volatile(
036                             "       xchgl   %%ebx,%%esp      \n"
037                             "       call    *%%edi           \n"
038                             "       movl    %%ebx,%%esp      \n"
039                             : "=a" (arg1), "=d" (arg2), "=b" (ebx)
040                             :  "0" (irq),   "1" (desc),  "2" (isp),
041                             "D" (desc->handle_irq)
042                             : "memory", "cc"
043                     );
044             }
045      else
046     #endif
047             desc->handle_irq(irq, desc);
```

Figura C.17 – Trecho da rotina `do_IRQ` (4ª Parte)

Na figura C.17 tem-se:

1- O bloco das linhas 027 a 045 só é compilado se o Linux está configurado com um tamanho de pilha do núcleo de 4 KB (`CONFIG_4KSTACKS`). Se for o caso, este bloco tem a função de copiar os dados da pilha do núcleo para a pilha IRQ (ver item C.4.7). Se a pilha do núcleo for configurada em 8 KB, só é compilada a linha 047 que chama o manipulador da interrupção (ver item C.4.3).

2- A linha 027 chama a função `current_thread_info()` que retorna o endereço da pilha do núcleo e a linha 028 retorna o endereço da pilha IRQ.

3- A linha 029 compara os endereços obtidos no passo anterior. Se os dois endereços são iguais, o núcleo já está usando a pilha IRQ e há um desvio para a linha 047 que chama o manipulador da interrupção. Isto ocorre, por exemplo, quando há interrupções aninhadas. Se os dois endereços não são iguais, o bloco das linhas 030 a 044 efetua a troca da pilha e chama o manipulador da interrupção.

5 – Instruções Finais

```
048     irq_exit();
049     set_irq_regs(old_regs);
050     return 1;
```

Figura C.18 – Trecho da rotina `do_IRQ` (5ª Parte)

Na figura C.18 acima estão as linhas finais da rotina `do_IRQ`. Pode-se ver:

1- A linha 048 chama a função `irq_exit()`.
2- A linha 049 recupera os valores iniciais dos registradores.

5.1 – Análise da Rotina: `irq_exit()`

```
001     void irq_exit(void)
002     {
003       account_system_vtime(current);
004       trace_hardirq_exit();
005       sub_preempt_count(IRQ_EXIT_OFFSET);

006       if (!in_interrupt() && local_softirq_pending())
             invoke_softirq();
007       preempt_enable_no_resched();

008     }
```

Figura C.19 – Código da rotina `irq_exit`
Extraído de `linux/kernel/softirq.c`

No código da figura C.19 tem-se:

1- A função da linha 003 não executa nenhuma operação na arquitetura x86 como pode ser verificado em:
   - `linux/include/linux/hardirq.h`

2- Na linha 005 o Linux decrementa o contador de interrupções aninhadas.
3- Na linha 006 se não há mais interrupções aninhadas e há *deferrable functions* pendentes, o Linux inicia a execução das mesmas.
4- Quando o Linux executa uma *deferrable function* bloqueia a preempção do núcleo e, consequentemente, ao terminar tem que desbloqueá-las (linha 007).

C.5.4 Análise da Rotina: __do_IRQ

De forma análoga ao item anterior, o super-manipulador de interrupção `__do_irq` será analisado por partes. O código completo encontra-se em `linux/kernel/irq/handle.c`.

1 – 1ª Parte

```
001    fastcall unsigned int __do_IRQ(unsigned int irq)

002    {
003      struct irq_desc *desc = irq_desc + irq;
004      struct irqaction *action;
005      unsigned int status;

006      kstat_this_cpu.irqs[irq]++;
007      if (CHECK_IRQ_PER_CPU(desc->status)) {
008           irqreturn_t action_ret;
009           if (desc->chip->ack) desc->chip->ack(irq);
010           action_ret = handle_IRQ_event(irq, desc->action);
011           desc->chip->end(irq);
012           return 1;
013     }
```

Figura C.20 – Trecho da rotina `__do_IRQ` (1ª Parte)

No código da figura C.20 tem-se:

1- A linha 006 atualiza o número de interrupções tratadas por CPU.

2- O bloco de instruções 007 a 013 trata a interrupção no caso de ser um sistema SMMP e estar configurado por parâmetro de compilação para especificar interrupções por processador (`CONFIG_IRQ_PER_CPU`). Se configurado desta forma, em um sistema SMMP este bloco de instruções é executado pela CPU que está encarregada de atender a interrupção.

2 – 2ª Parte

```
014    spin_lock(&desc->lock);
015    if (desc->chip->ack) desc->chip->ack(irq);

016   status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
017   status |= IRQ_PENDING;

018   action = NULL;
019   if (likely(!(status & (IRQ_DISABLED | IRQ_INPROGRESS)))) {
020                 action = desc->action;
021                 status &= ~IRQ_PENDING;
022                 status |= IRQ_INPROGRESS;
023    }
024   desc->status = status;

025    if (unlikely(!action)) goto out;
```

Figura C.21 – Trecho da rotina `__do_IRQ` (2ª Parte)

Operações efetuadas no código da figura C.21:

1- A linha 014 executa um travamento (*lock*) no descritor da interrupção.

2- A linha 015 chama o método `ack` do objeto `chip` que consta no descritor da interrupção. Como foi visto no item C.4.4, o funcionamento do método depende do modelo de controlador de interrupções instalado no hardware.

3- A linha 016 limpa os bits `IRQ_REPLAY` e `IRQ_WAITING` que fazem parte do mecanismo do Linux para controlar o tratamento de uma interrupção.

4- A linha 017 marca o bit `IRQ_PENDING` indicando que o controlador já foi sinalizado, mas que a interrupção ainda não foi tratada.

5- O bloco das linhas 019 a 023 é executado se a interrupção não foi bloqueada pelo núcleo e se a interrupção não está sendo tratada por outra CPU. A linha 020 carrega na variável `action` o endereço do primeiro nó da lista ligada que contém as ISRs da interrupção. Nas linhas 021 e 022 o campo `status` é atualizado, marcando-se o bit `IRQ_INPROGRESS` para indicar que o tratamento da interrupção está em andamento.

6- Na linha 025 pode-se ver o desvio condicional para a linha `out` se a variável `action` está nula (*null*), o que ocorre quando: 1) a interrupção está bloqueada, 2) o tratamento da interrupção já está em andamento em outra CPU e 3) não há ISR cadastrada para essa interrupção.

C – 3ª Parte

```
026     for (;;) {

027     irqreturn_t action_ret;
028     spin_unlock(&desc->lock);

029     action_ret = handle_IRQ_event(irq, action);

030     if (!noirqdebug) note_interrupt(irq, desc, action_ret);
031     spin_lock(&desc->lock);

032     if (likely(!(desc->status & IRQ_PENDING))) break;
033     desc->status &= ~IRQ_PENDING;
034     }

035     desc->status &= ~IRQ_INPROGRESS;
```

Figura C.22 – Trecho da rotina `__do_IRQ` (3ª Parte)

Na figura C.22 tem-se:

1- A linha 028 libera o descritor de interrupção. Isto é feito para que em um sistema SMMP uma interrupção de mesmo vetor possa ser atendida, haja vista que o controlador de interrupções já foi sinalizado.

2- A linha 029 chama a função `handle_IRQ_event` que basicamente percorre a lista ligada e executa todas as ISRs (ver item C.4.5).

3- A linha 030 chama a função `note_interrupt` que analisa se a interrupção foi, ou não, tratada com sucesso por alguma ISR. Faz parte do mecanismo de bloqueio de interrupções espúrias implementado pelo Linux.

4- A linha 031 executa um travamento (*lock*) no descritor da interrupção.

5- Na linha 032, se o bit `IRQ_PENDING` está limpo o programa sai do laço e desvia para a linha 035.

6- Na linha 035 o bit `IRQ_INPROGRESS` é limpo, sinalizando que a interrupção foi tratada.

O comando `for` (linha 026) assim como as linhas 032 e 033 são explicados pela existência de um mecanismo de delegação do tratamento de interrupções no Linux em sistemas SMMP. Este mecanismo pode ser explicado com o exemplo abaixo:

1- Chega uma primeira IRQn ao sistema que é direcionada à CPUx.
2- A CPUx sinaliza o controlador, liberando-o para enviar mais interrupções (linha 015).
3- Como não há outra CPU tratando uma IRQn anterior, a CPUx limpa o bit `IRQ_PENDING` (linha 21), marca o bit `IRQ_PROGRESS` (linha 22) e inicia a chamada às ISRs da interrupção (linha 29).

4- Enquanto a CPUx executa as operações do item anterior, chega ao sistema outra interrupção IRQn que é direcionada à CPUy.
5- A CPUy ao executar a linha 017 marca o bit `IRQ_PENDING`. Na linha 019 a CPUy encontra o bit `IRQ_PROGRESS` marcado, ou seja, uma IRQn anterior já está sendo tratada por outra CPU. A CPUy não trata interrupção (desvia para `out` na linha 025) e através do bit `IRQ_PENDING` sinaliza à CPUx que deve tratar outra IRQn.
6- A CPUx depois de tratar a interrupção verifica que a interrupção está com o bit `IRQ_PENDING` marcado (linha 032), ou seja, outro processador lhe delegou o tratamento de outra IRQn. A CPUx limpa o bit (linha 033) e executa outra iteração do laço.

Embora esta lógica simples funcione na maioria dos casos, em sistemas SMMP existem situações na qual ela falha por isso o Linux usa também o estado `IRQ_REPLAY`. O mecanismo ligado a este bit não será explicado, detalhes sobre como é usado podem ser encontrados na referência (24).

D – 4ª Parte

```
036    out:
037       desc->chip->end(irq);
038       spin_unlock(&desc->lock);
039       return 1;
```

Figura C.23 – Trecho da rotina `__do_IRQ` (4ª Parte)

Da figura C.23 tem-se:

1- Na linha 037 o método `end` é executado. As operações efetuadas pelo método dependem do controlador de interrupções.

2- Na linha 038 o descritor de interrupção é liberado.

C.5.5 Análise da Função: handle_IRQ_event

Na figura C.24 tem-se a parte principal do código da função `handle_IRQ_event` comentado. Esta função basicamente realiza as seguintes operações:

1- Na linha 002, se a interrupção pode ser tratada com as interrupções habilitadas as habilita.

2- Executa todas as ISR cadastradas no descritor (linhas 003 a 012).

```
001   // Verifica se pode habilitar as interrupções
002   if (!(action->flags & IRQF_DISABLED))
                      local_irq_enable_in_hardirq();

003    do {
004       // Executa a Rotina de Serviço de Interrupção (ISR)
005       ret = action->handler(irq, action->dev_id);

006       // Se a IRQ foi tratada atualiza o status
007       if (ret == IRQ_HANDLED) status |= action->flags;

008       // Carrega o valor de retorno
009       retval |= ret;

010       // Busca a próxima Rotina de Serviço de Interrupção
011        action = action->next;

012      } while (action);

013   // Verifica de dispositivo é gerador de entropia
014   if (status & IRQF_SAMPLE_RANDOM)
                      add_interrupt_randomness(irq);

015    // Desabilita as interrupções
016    local_irq_disable();

017    return retval;
```

Figura C.24 – Parte principal do código da função `handle_IRQ_event`
Em `linux/kernel/irq/handle.c`

C.6 Tratamento de Interrupção E/S Externa – 2ª Parte

Neste item é tratado o que foi chamado no texto de retorno da interrupção que consiste nas operações realizadas a partir da linha `ret_from_intr` (ver figura C.11). Como as instruções estão em linguagem de montagem, apenas serão apresentados dois fluxogramas simplificados que foram adaptados da referência (24), a qual trata do tema com mais detalhes. O bloco de instruções encontra-se em `linux/arch/x86/kernel/entry_32.s`.

C.6.1 Núcleo Compilado Sem Preempção

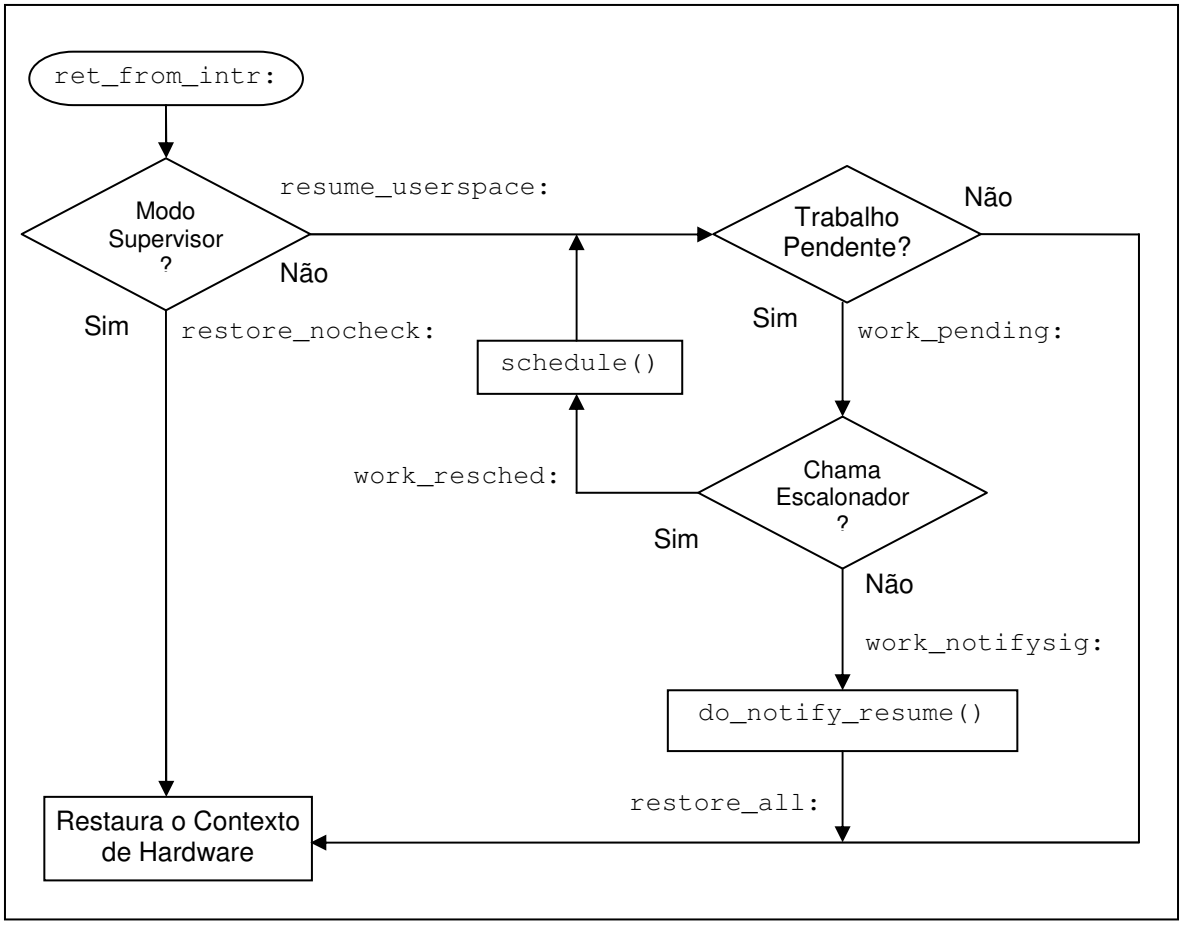

Figura C.25 – Fluxograma do retorno da interrupção em núcleo sem preempção
Adaptado da referência (24)

Analisando o fluxograma da figura C.25:

1- Se o processo estava em Modo Supervisor quando a interrupção foi sinalizada, a execução do fluxo de controle do núcleo é retomada após o tratamento da interrupção.
2- Como o núcleo é sem preempção, só há troca de processo, após o tratamento da interrupção, se o processo estava em Modo Usuário no momento em que a interrupção foi sinalizada.
3- O núcleo verifica se há alguma operação pendente que pode, por exemplo, decorrer do tratamento da interrupção. Exemplo dessas operações: chamada ao escalonador e sinalização a um processo.

C.6.2 Núcleo Compilado Com Preempção

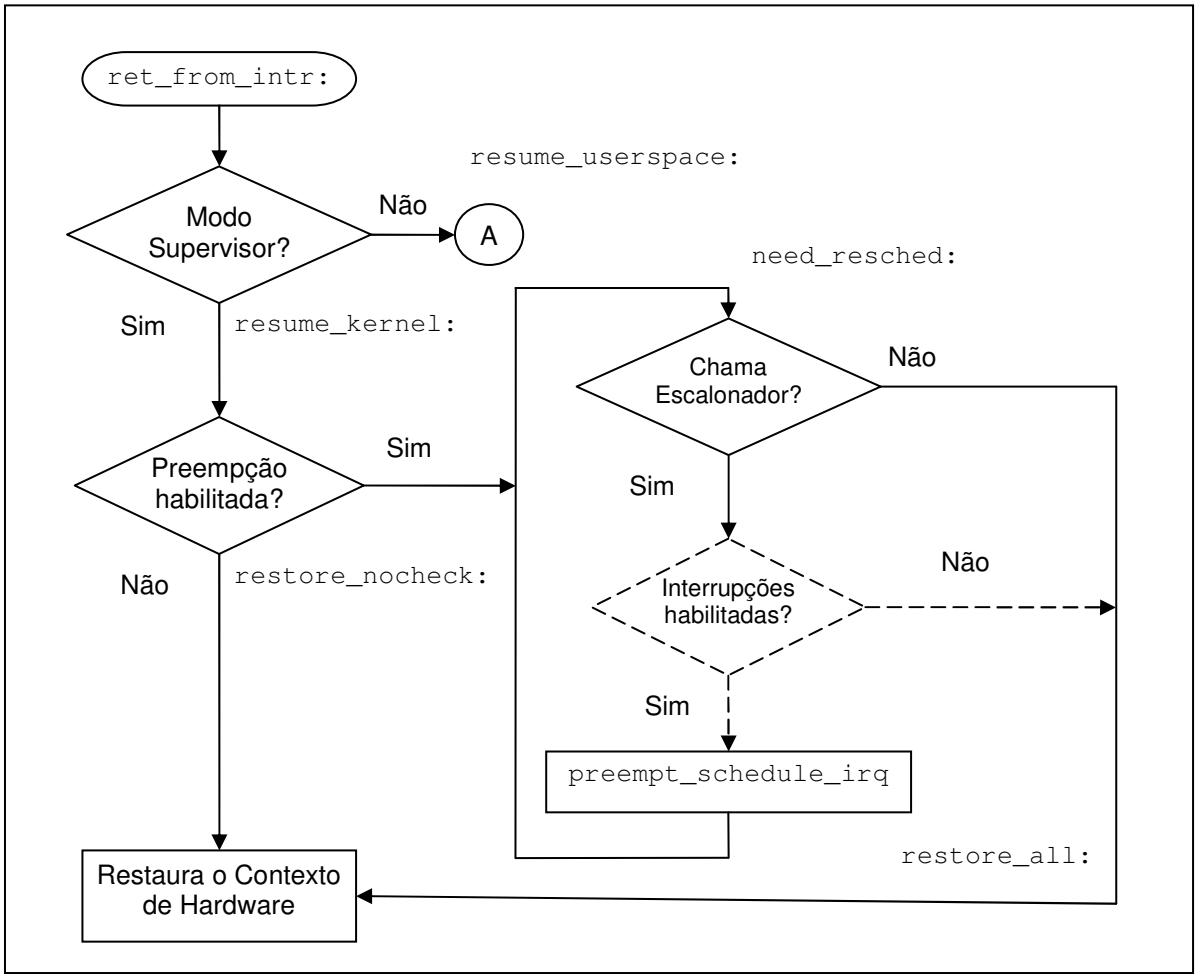

Figura C.26 – Fluxograma do retorno da interrupção em núcleo com preempção
Adaptado da referência (24)

Analisando o fluxo da figura C.26

1- O Linux trata o retorno do tratamento de uma interrupção e de uma exceção com o mesmo bloco de instruções. Isto faz com algumas instruções só tenham função no retorno de uma exceção. É o caso, por exemplo, na figura C.26 da parte do fluxo que está tracejada a qual só é executada no retorno de uma exceção.
2- Se o processo corrente estava em Modo Usuário na chegada da interrupção, as instruções são as mesmas do item anterior (item C.6.1) haja vista que há um desvio para a linha `resume_userspace`.
3- Pode-se observar que não há troca de processo quando a preempção do núcleo está desativada.

C.7 Tratamento de Interrupção E/S Locais

Neste item é analisado como o Linux trata as interrupções E/S locais. O foco deste item é nas interrupções que são geradas pelos sensores e pelo temporizador do APIC Local que podem ser classificadas como interrupções E/S locais, por serem internas ao processador, se considerarmos um sistema SMMP.

Como foi visto no item C.5.1, todas as entradas da IDT das interrupções E/S externas têm o mesmo endereço, o que faz com todas as interrupções desse tipo sejam inicialmente tratadas pelo mesmo bloco de instruções. Como as interrupções E/S locais são específicas da arquitetura x86, o Linux configura a entrada da IDT com o endereço da rotina específica que irá tratar a interrupção.

O código para tratar a interrupção gerada pelo temporizador do APIC Local (IRQv239) pode ser visto na figura C.27. Este é um bloco padrão para as interrupções E/S locais, sendo que para as outras interrupções desse tipo apenas muda a rotina que é chamada na linha 007.

```
001     RING0_INT_FRAME;
002     pushl $~(nr);
003     CFI_ADJUST_CFA_OFFSET 4;
004     SAVE_ALL;
005     TRACE_IRQS_OFF
006     movl %esp,%eax;
007     call smp_apic_timer_interrupt;
008     jmp ret_from_intr;
009     CFI ENDPROC
```

Figura C.27– Instruções do tratador da interrupção do temporizador da APIC Local
Extraído de `linux/arch/x86/kernel/entry_32.s`

Pode-se ver a semelhança entre o código da figura e aqueles tratados nos itens C.5.1 (figura C.10) e C.5.2 (figura C.11). A principal diferença consiste que nas interrupções E/S externas é chamada a rotina `do_IRQ` enquanto neste tipo de interrupção é chamada a rotina de serviço de interrupção que irá tratar o dispositivo (linha 007).

****

Apêndice D - Exemplo de Subamostragem

Neste apêndice, através de um exemplo, o conceito de subamostragem é estabelecido. Na figura D.1 pode-se ver um exemplo de um sistema hipotético de comunicação digital com multiplexação por divisão de tempo, baseado em um microcomputador compatível com o IBM-PC. Maiores informações sobre multiplexação por divisão de tempo podem ser encontradas na referência (17).

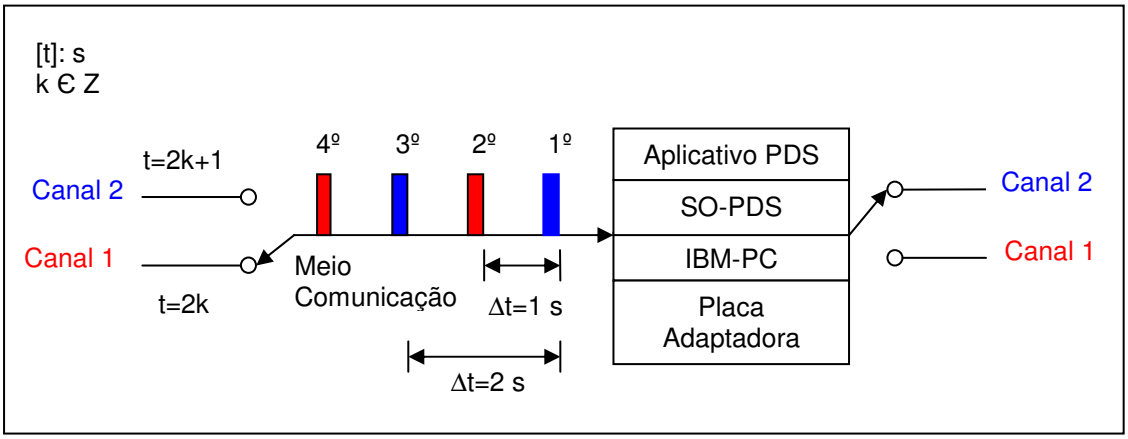

Figura D.1 – Exemplo de sistema de multiplexação por divisão de tempo com base em microcomputador compatível com o IBM-PC

O funcionamento do sistema da figura pode ser descrito como segue:

1- Nos instantes t=2k amostras do canal 1 (vermelho) são colocadas no meio de comunicação e, nos instante t=2k+1 são colocadas amostras do canal 2 (azul).

2- As amostras chegam ao sistema hipotético desenvolvido com base um microcomputador compatível com o IBM-PC. Neste tipo de sistema é o aplicativo PDS que faz todo o processamento, ou seja, lê a amostra do meio de comunicação, determina o canal e envia o dado para a saída.

Na figura, pode-se observar que o sistema tem teoricamente um intervalo de tempo Δt=1 s para tratar a 1ª amostra (azul) e gerar a saída, antes que a 2ª amostra (vermelha) chegue. Na prática uma pequena variação é permitida.

Também se pode observar que se o sistema, devido à carga do sistema, demorar um intervalo de tempo $\Delta t=2$ s para tratar a 1ª amostra (azul) e gerar a saída, a 2º amostra (vermelha) é perdida, haja vista que a 3º amostra (azul) chegou ao sistema.

Quando o sistema começa a perder amostras como descrito acima, diz-se que o sistema está subamostrando. No exemplo descrito o sistema deve amostrar o sinal de entrada com um freqüência de 1 Hz, ou seja, a cada 1s. Ao perder a 2º amostra, o sistema passou a ler a entrada a cada 2s, ou seja, a freqüência de amostragem passou a ser de 0,5 Hz.

A conseqüência do sistema estar subamostrando depende especificamente da aplicação PDS que está sendo executada. No exemplo proposto podem ocorrer hipoteticamente duas situações dependendo da aplicação:

1- Se a aplicação PDS detectar a subamostragem, a 3ª amostra é direcionada para o Canal 2 e, como a 2ª amostra foi perdida, apenas o Canal 1 é afetado, não recebendo a informação.

2- Se a aplicação PDS não detectar a subamostragem, a 3ª amostra é direcionada erradamente para o Canal 1 e, neste caso, ambos os canais tem seu funcionamento afetado.

É interessante observar que probabilidade de ocorrer subamostragem depende da latência para iniciar o processamento da amostra e do tempo de processamento da amostra. A figura D.2 a seguir exemplifica a dependência da subamostragem com o tempo de processamento. Como a idéia e a analisar a influência da latência, no exemplo da figura, estabelece-se duas hipóteses: 1) o tempo de processamento é determinístico e 2) a latência não é determinística.

A figura mostra um período de amostragem para dois sistemas distintos, onde se tem:

1- No instante IRQa sinaliza o instante de amostrar o sinal de entrada, ou seja, o instante de ler a amostra de entrada.
2- Ambos os sistemas tem uma latência L não determinística para iniciar o tratamento da amostra.
3- O Sistema 1 demora um intervalo de tempo ΔP1 para tratar a amostra, o Sistema 2 demora ΔP2, com ΔP2 > ΔP1.

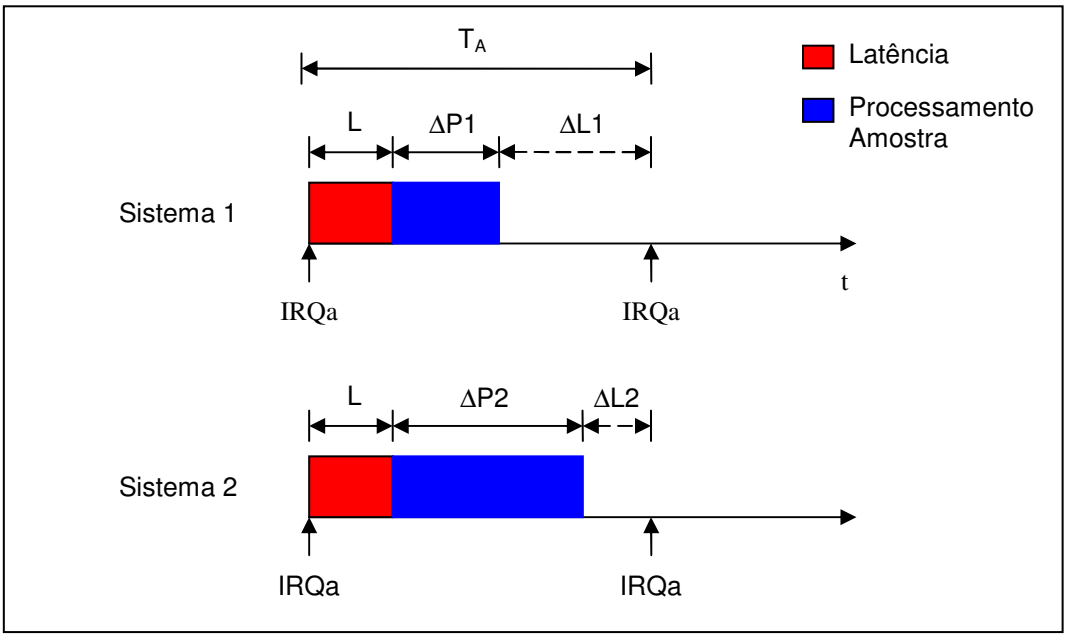

Figura D.2 – Influência do tempo de processamento na subamostragem

Com base nas hipóteses estabelecidas, pode-se observar que no Sistema 2 é necessária uma variação (ΔL2) menor do que a variação (ΔL1) necessária no Sistema 1, para que a soma da latência e do tempo de processamento ultrapasse o período de amostragem (T_A) e, conseqüentemente, o sistema passe a subamostrar.

Pode-se afirmar que se a latência variar da mesma forma em ambos os sistemas, a probabilidade do Sistema 2 subamostrar é maior ou igual à do Sistema 1, para um mesmo intervalo de tempo. Este exemplo mostra que a subamostragem em um sistema depende do tempo de processamento da amostra, sendo que, um raciocínio semelhante pode ser usado para mostrar a dependência com a latência para iniciar o processamento.

****

Apêndice E - Sistemas de Tempo Real

E.1 Introdução

Este apêndice apresenta uma definição para sistema de tempo real e uma classificação para este tipo de sistema. Também apresenta os tipos de requisitos encontrados para especificar um sistema. Este texto tomou como base a referência (14).

E.2 Definição Sistema Tempo Real

Um sistema de tempo real é um sistema que deve satisfazer requisitos estritos de tempo de resposta ou há o risco de conseqüências severas, incluído a falha total do sistema.

Falha do sistema é quando o sistema não pode satisfazer um, ou mais de um, dos requisitos estipulados na especificação formal do sistema.

E.3 Classificação dos Sistemas Tempo Real

Os sistemas de tempo-real são classificados na literatura como segue:

- Sistema de tempo real suave (*soft real-time*) é um sistema no qual o desempenho é degradado, mas não é danificado pelo não atendimento dos requisitos estritos de tempo. Ex: sistemas para tocar arquivos de músicas.

- Sistema de tempo real rígido (*hard real-time*) é um sistema no qual o não atendimento de um único limite de tempo pode levar a uma falha completa ou catastrófica do sistema. Ex: sistema de aviônicos para controle de mísseis.

- Sistema de tempo real firme (*firm real-time*) é um sistema no qual um número mínimo de não atendimentos aos limites de tempo não leva à falha total do sistema, mas um número maior que esse mínimo pode levar a uma falha completa ou catastrófica do sistema: Ex: sistema para controle de robôs.

E.4 Requisitos

Engenharia de requisitos é uma subdisciplina da Engenharia de Software que trata da determinação dos objetivos, funções, limitações de um sistema de software e, também, da representação desses aspectos para modelagem e análise.

Apesar de haver várias formas de taxonomia para os requisitos, a maioria dos autores divide os requisitos em funcionais e não funcionais.

- Requisitos funcionais são as descrições de todas as entradas do sistema e da seqüência de operação associada a cada conjunto de entradas. Os requisitos funcionais também incluem as especificações das saídas para situações normais e anormais de cada entrada possível. Essencialmente requisitos funcionais descrevem completa e especificamente o comportamento do sistema.

- Requisitos não funcionais são especificações do sistema que não estão relacionados à funcionalidade, mas à qualidade do mesmo. Alguns exemplos desses requisitos:

    - Especificações sobre as interfaces externas.
    - Especificações sobre o desempenho do sistema.
    - Especificações sobre o banco de dados lógico.
    - Especificações sobre os requisitos de design como, por exemplo, obediência a padrões.
    - Especificações sobre confiabilidade, segurança, portabilidade e manutenção.

****

Apêndice F - A Latência na Interrupção E/S Local

F.1 Introdução

A Intel introduziu no Pentium III um componente chamado Controlador Programável de Interrupções Avançado Local (*Local Advanced Programmable Interrupt Controller*) ou APIC Local. Este componente dispõe ao sistema operacional um temporizador (*timer*) que sinaliza o microprocessador através de uma interrupção E/S, de forma semelhante aos dispositivos externos.

O microprocessador trata as interrupções[1] sinalizadas pelo APIC Local da mesma forma que trata as interrupções dos dispositivos externos, o que não ocorre com o Linux. Este fato faz com que haja diferenças em termos de latência entre o tratamento de uma interrupção E/S local e uma interrupção E/S externa, o qual é o tema tratado no Capítulo 4.

Neste apêndice, o modelo de latência estabelecido no Capítulo 4 é adaptado para a interrupção E/S local. O estudo se justifica porque no modo de operação *Pull-In* (ver Capítulo 2, item 2.2.3), uma fonte de tempo da arquitetura é necessária para o controle da freqüência de amostragem, sendo que neste caso o temporizador do APIC Local seria uma possibilidade, como analisado no Capítulo 5, item 5.4.

O modelo de latência estabelecido no Capítulo 4 define a latência de execução (ver item 4.6) a qual não está associado à interrupção e, sim, à execução do processo PDS, por isso não será tratada neste apêndice.

F.2 Latência de Tratamento de Interrupção

As latências originadas exclusivamente pelo hardware não se alteram, ou seja:

[1] O APIC Local possui outros recursos que geram interrupções. Ver Apêndice C, item C.3.

1- Latência da Leitura da Interrupção (Capítulo 4, item 4.3.1).
2- Latência de Mascaramento (Capítulo 4, item 4.3.2).
3- Latência da Fila do Controlador Programável de Interrupções (Capítulo 4, item 4.3.3).
4- Latência de Operação do Microprocessador (Capítulo 4, item 4.3.4).

F.2.1 Latência de Salvamento dos Registradores

No Capítulo 4 (item 4.3.5) é mostrado que todas as entradas da IDT das interrupções E/S externas têm o mesmo endereço, o que faz com todas as interrupções desse tipo sejam inicialmente tratadas pelo mesmo bloco de instruções. Já para as interrupções E/S locais, o Linux configura a entrada da IDT com o endereço da rotina específica que irá tratar a interrupção.

O código para tratar a interrupção gerada pelo temporizador do APIC Local pode ser visto na figura F.1. O código da figura é endereçado pela entrada 239 da IDT.

```
001     RING0_INT_FRAME;
002     pushl $~(nr);
003     CFI_ADJUST_CFA_OFFSET 4;
004     SAVE_ALL;
005     TRACE_IRQS_OFF
006     movl %esp,%eax;
007     call smp_apic_timer_interrupt;
008     jmp ret_from_intr;
009     CFI_ENDPROC
```

Figura F.1 – Instruções para tratar a interrupção do temporizador da APIC Local
Extraído de linux/arch/x86/kernel/entry_32.s

Comparando os códigos apresentados no Capítulo 4, item 4.3.5 (figuras 4.3 e 4.4) com o código da figura F.1, constata-se semelhanças entre eles o que justifica o conceito de latência de salvamento dos registradores para este tipo de interrupção. Do ponto de vista teórico, esta latência de salvamento dos registradores difere da definida para as interrupções E/S externas, haja vista que os blocos de instruções possuem algumas diferenças. Observa-se na linha 007 a chamada para a rotina específica que realiza o tratamento propriamente dito da interrupção.

F.2.2 Latência de Entrada no Tratador de Interrupções

O código da rotina `smp_apic_timer_interrupt` pode ser visto na figura F.2.

```
001    void smp_apic_timer_interrupt(struct pt_regs *regs)

002    {
003      struct pt_regs *old_regs = set_irq_regs(regs);
004      int cpu = smp_processor_id();

005      per_cpu(irq_stat, cpu).apic_timer_irqs++;


006     /* NOTE! We'd better ACK the irq immediately,
007        because timer handling can be slow. */
008      ack_APIC_irq();

009      irq_enter();
010      smp_local_timer_interrupt();
011      irq_exit();
012      set_irq_regs(old_regs);
013    }
```

Figura F.2 – Código da rotina `smp_apic_timer_interrupt`
Extraído de linux/arch/x86/kernel/apic.c

Pode-se ver que a latência de entrada no tratador de interrupções também existe, embora, se reduza às instruções das linhas 003, 004, 005, 008 e 009[2]. Como na linha 010 tem-se a ISR que trata da interrupção, não há latência de compartilhamento (ver Capítulo 4, item 4.3.7). As instruções das linhas 011 e 012 por serem executadas após a ISR, são consideradas como fazendo parte da latência de despacho.

F.2.3 Latência de Despacho

De forma resumida tem-se:

1- Na figura F.2 pode-se ver que a latência de saída do tratador de interrupções continua existindo, mas limitada às instruções das linhas 011 e 012.

[2] Do ponto de vista estritamente teórico a execução da linha 008 não pode ser considerada uma latência já que é uma operação necessária ao tratamento da interrupção.

2- Comparando os códigos das figuras 4.4 (item 4.3.5) e 4.7 (item 4.5.1.3) no Capítulo 4 com os das figuras F.1 e F.2 respectivamente, pode-se concluir que todas as outras latências não se alteram, a saber:

- latência de pilha de IRQs (Capítulo 4, item 4.5.2),
- latência de d*eferrable functions* (Capítulo 4, item 4.5.3),
- latência de preempção do núcleo desabilitada (Capítulo 4, item 4.5.4),
- latência de escalonamento (Capítulo 4, item 4.5.5),
- latência de troca de contexto (Capítulo 4, item 4.5.6).

****

Apêndice G - Método de Medição - Aplicativo em Modo Usuário

G.1 Introdução

Neste item são apresentadas todas as rotinas que compõem o aplicativo que trata as medidas em Modo Usuário. As linhas abaixo compõem um arquivo único chamado `Aplicativo_Medicao.c` e, estão separadas em itens para facilitar o entendimento. O Aplicativo de Gerenciamento descrito no texto (Capítulo 6, item 6.5.1) corresponde à rotina `main` do item G.5

G.2 Bibliotecas e Definições

```
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <stdint.h>
#include <sys/time.h>
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <signal.h>
#include <sched.h>

#define read_tsc(x) __asm__ __volatile__ (".byte 0x0f, 0x31":"=A" (*x));
```

G.3 Estrutura

```
struct real_time_struct {
      char              medida_on;                  // Medição Funcionando
      int               irq_cod;                    // Código IRQ
      char              flag_medida_pit;            // Flag PIT
      unsigned int      medida_pit;                 // Medida PIT
      char              flags_medida[20];           // Flag TSC
      uint64_t          medida[20];                 // Medidas TSC

      int               irq_aninhadas;              // IRQ
      char              irq_handle;                 // IRQa tratada
      char              scheduler;                  // Scheduler liberado

      unsigned long     tsc_khz;                    // Frequencia do TSC
      unsigned long     pit_ctr;                    // PIT-Clock Tick Rate
      unsigned long     pit_hz;                     // PIT-Kernel Rate
      unsigned long     pit_latch;                  // PIT-Kernel Inic.
      unsigned long     processo_tr;                // Processo TR
};
```

G.4 Rotina PDS

```
int rotina_pds(int irq, int tot_a)
{
      int ok, i, conta, lim_inf;
      struct real_time_struct dados;

      uint64_t result[100][25];
      uint64_t temp_0, temp_1;

      FILE *arq;

      // Inicializa variáveis
      lim_inf=5;                    // Amostras a desprezar

      for (conta=0; conta<100; conta++) {
            for (i=0; i<25; i++) result[conta][i]=0;
      }

      // Inicia Medição
      ok=syscall(188,1,irq);
      if (ok!=0) {
            printf("Dispositivo Não aberto \n");
            return 1;
      };

      // Laço - Ignora primeiras amostras
      for (conta=0; conta<lim_inf; conta++) {
            ok=syscall(189, (char*) &dados);
            if (ok!=0)  {
                  printf("Erro na Leitura = %d \n", ok);
                  return 1;
            }
      }

      // Laço de Leitura
      for (conta=0;conta<tot_a;conta++) {

            ok=syscall(189, (char*) &dados);

            // Le 1a Marca
            read_tsc(&temp_0);

            // Verifica leitura
            if (ok!=0)  {
                  printf("Erro na Leitura = %d \n", ok);
                  return 1;
            }

            // Marcador 0 - (Processo Bloqueado)
            result[conta][0]= dados.medida[0];

            // Marcador 1 - Início tratamento IRQa
            result[conta][1]= dados.medida[1];       // TSC
            result[conta][20] = dados.medida_pit;    // PIT

            // Marcador 2 - Fim Camada Entrada / Início Laço
            result[conta][2]= dados.medida[2];

            // Marcador 3 - Início ISR
            result[conta][3]= dados.medida[3];
```

```
      // Marcador 4 - Fim ISR
      result[conta][4]= dados.medida[4];

      // Marcador 5 - Fim Laço / Início Camada Saída
      result[conta][5]= dados.medida[5];

      // Marcador 6 - Fim Camada Saída / Início Pilha IRQs
      result[conta][6]= dados.medida[6];

      // Marcador 7 - Fim Pilha IRQs / Início Deferrable Functions
      result[conta][7]= dados.medida[7];

      // Marcador 8 - Fim Deferrable Functions / Início Preempcao
      result[conta][8]= dados.medida[8];

      // Marcador 9  - Início Escalonador
      result[conta][9]= dados.medida[9];

      // Marcador 10 -  Fim Escalonador
      result[conta][10]= dados.medida[10];

      // Marcador 11 - Início Troca de Contexto
      result[conta][11]= dados.medida[11];

      // Marcador 12 - Fim Troca de Contexto
      result[conta][12]= dados.medida[12];

      // Marcador 13 - Início tratamento da amostra
      result[conta][13]= temp_0;

      // Grava Dados Adicionais
      result[conta][15]= dados.tsc_khz;
      result[conta][16]= dados.pit_ctr;
      result[conta][17]= dados.pit_hz;
      result[conta][18]= dados.pit_latch;
      result[conta][19]= dados.irq_aninhadas;

      // Contador
      result[conta][21]=conta;

      // Marcador 14 - Fim tratamento da amostra
      read_tsc(&temp_1);
      result[conta][14] = temp_1;

      printf("Medida = %d \n", conta);
};    // fim loop


// Finaliza Medição
ok=syscall(188,0,irq);
if (ok!=0) {
      printf("Dispositivo Não Fechado \n");
      return 1;
};

// Gravação Arquivo
arq=fopen("result_latências.csv", "a");

for (conta=0; conta<tot_a; conta++) {
      for (i=0;i<25;i++) {
```

```
                fprintf(arq, "%llu^", result[conta][i]);
            }
            fprintf(arq, "\n");
      }
      fclose(arq);

      return 0;
}
```

G.5 Aplicativo de Gerenciamento

```
void alarme(int codigo) {
      int i;
      i=0;
}


int main(int argc, char *argv[]) {

      int cont, total, interv, sinal, ok, irqa, total_a;
      int politica_padrao;

      struct sched_param param_padrao;
      struct sched_param param_treal;
      struct sigaction sinal_H;


      // Dados Iniciais Escalonador
      politica_padrao=sched_getscheduler(0);
      ok=sched_getparam(0, &param_padrao);
      if (ok==-1) return;
      param_treal.sched_priority = sched_get_priority_max(SCHED_FIFO);

      // Dados Iniciais Alarme
      sinal_H.sa_handler=&alarme;
      sigaction(SIGALRM, &sinal_H, NULL);
      sinal=0;

      // Dados Medição
      interv=120;                  // Intervalo entre as Janelas (segundos)
      total=200;                   // Total Conjunto de Dados
      total_a=100;                 // Total de Amostras por Conjunto
      irqa=0;                      // Interrupção Monitorada (IRQa)

      for (cont=0;cont<total;cont++) {

            // Alarme
            printf("Sistema em Pausa i= %d \n", cont);
            alarm(interv);
            sinal=pause();
            printf("Sistema Rodando i= %d \n", cont);

            // Processo em Escalonamento TR
            ok=sched_setscheduler(0, SCHED_FIFO, &param_treal);
            if (ok==-1) break;

            // Medição
            sinal=rotina_pds(irqa, total_a);
            if (sinal==1) break;

            // Processo em Escalonamento Normal
```

```
            ok=sched_setscheduler(0, politica_padrao, &param_padrao);
            if (ok==-1) break;
      }
      printf("*** FINAL *** \n");
}
```

****

Apêndice H - Instrumentalização do Núcleo

H.1 Introdução

A instrumentalização do núcleo consiste basicamente de 2 (dois) arquivos que foram incluídos ao núcleo Linux 2.6.25 e de alterações em rotinas já existentes no núcleo.

H.2 Arquivo: `/linux/include/asm-x86/tempo_real.h`

O arquivo `tempo_real.h` inclui as referências externas às funções e define a estrutura que é usada no espaço de memória do núcleo.

H.2.1 Referências Externas

```
#include <asm/tsc.h>
#include <linux/clocksource.h>


extern void boot_medida(void);

extern void mede_pit(int codigo);
extern void mede_tsc(int etapa, int codigo);

extern void mede_sched_in(int etapa);
extern void mede_sched_out(int etapa);

extern void mede_tro_cntx(int etapa, unsigned long pro_tr);

extern int inic_medida(int num_irq);
extern int fim_medida(void);

extern int le_dados(char *end_buffer);
extern void driver_medida(int codigo);
```

H.2.2 Estrutura: `real_time_struct`

```
struct real_time_struct {
      bool  medida_on;                        // Medindo Sim/Não
      int   irq_cod;                          // IRQ monitorada

      bool              flag_medida_pit;      // Flag Medida PIT
      unsigned int      medida_pit;           // Medida PIT

      bool              flags_medida[20];     // Flag Medidas TSC
      cycles_t          medida[20];           // Medidas TSC

      int               irq_aninhadas;        // No IRQa Subamostragem
```

```
      bool              irq_handle;             // IRQa tratada

      bool              scheduler;              // Liberado Escalonador

      unsigned long     tsc_khz;                // Frequencia do TSC
      unsigned long     pit_ctr;                // PIT - Clock Tick Rate
      unsigned long     pit_hz;                 // PIT - Kernel Rate
      unsigned long     pit_latch;              // PIT - Kernel Contador

      unsigned long     processo_tr;            // Processo TR
};
```

H.3 Arquivo: `/linux/arch/x86kernel/tempo_real.c`

Este arquivo contém as rotinas que, na prática, instrumentalizam o núcleo.

H.3.1 Bibliotecas

```
#include <asm/tempo_real.h>

#include <linux/module.h>                 // asmlinkage
#include <asm/linkage.h>                  // idem

#include <linux/kernel.h>                 // printk
#include <asm/tsc.h>                      // TSC do Kernel
#include <linux/wait.h>                   // filas de espera
#include <linux/linkage.h>                // chamada do sistema
#include <linux/sched.h>                  // escalonador e estados
#include <asm/uaccess_32.h>               // transferemcia memória
#include <asm/ptrace.h>                   // estrutura regs na IRQ
#include <linux/thread_info.h>            // set_need_resched
#include <asm/i8253.h>                    // PIT
#include <asm/io.h>                       // IO do PIT
#include <asm/current.h>                  // current

static struct real_time_struct rt_st;     // Estrutura

static DECLARE_WAIT_QUEUE_HEAD(real_time_wait);      // Cria fila de espera
```

H.3.2 Marcador do PIT

```
void mede_pit(int codigo) {
      unsigned int count;

      // Controle
      if (!rt_st.medida_on) return;
      if (codigo!=rt_st.irq_cod) return;
      rt_st.irq_aninhadas=rt_st.irq_aninhadas+1;
      if (rt_st.flag_medida_pit) return;

      // Medida
      count=0;
      outb_pit(0x00, PIT_MODE);     /* latch the count ASAP */
      count = inb_pit(PIT_CH0);     /* read the latched count */
      count |= inb_pit(PIT_CH0) << 8;
```

```
	rt_st.medida_pit= count;
	rt_st.flag_medida_pit=true;
	return;
}
```

H.3.3 Marcador TSC

```
void mede_tsc(int etapa, int codigo) {

	// Controle
	if (!rt_st.medida_on) return;
	if (codigo!=rt_st.irq_cod) return;
	if (rt_st.flags_medida[etapa]) return;

	// Medida
	rt_st.medida[etapa]=get_cycles();
	rt_st.flags_medida[etapa]=true;
	return;
}
```

H.3.4 Marcador Entrada ao Escalonador

```
void mede_sched_in(int etapa) {

	// Controle
	if (!rt_st.medida_on) return;
	if (!rt_st.scheduler) return;
	if (rt_st.flags_medida[etapa]) return;

	// Medida
	rt_st.medida[etapa]=get_cycles();
	rt_st.flags_medida[etapa]=true;
	return;
}
```

H.3.5 Marcador Saída do Escalonador

```
void mede_sched_out(int etapa) {

	// Controle
	if (!rt_st.medida_on) return;
	if (!rt_st.scheduler) return;
	if (rt_st.flags_medida[etapa]) return;

	// Medida
	rt_st.medida[etapa]=get_cycles();
	rt_st.flags_medida[etapa]=true;
	return;
}
```

H.3.6 Marcador Troca de Contexto

```
void mede_tro_cntx(int etapa, unsigned long pro_tr) {

	// Controle
	if (!rt_st.medida_on) return;
```

```
	if (!rt_st.scheduler) return;
	if (rt_st.flags_medida[etapa]) return;
	if (pro_tr!=rt_st.processo_tr) return;

	// Medida
	rt_st.medida[etapa]=get_cycles();
	rt_st.flags_medida[etapa]=true;
	return;
}
```

H.3.7 Inicia a Medição

```
int inic_medida(int num_irq) {
	int i;

	// Conciste IRQ
	if (num_irq>255) return -1;
	if (num_irq<0) return -1;

	// Coloca valores padrão na estrutura
	rt_st.irq_cod=num_irq;
	rt_st.flag_medida_pit=false;
	rt_st.medida_pit=0;
	for (i=0; i<20; i++) {
		rt_st.flags_medida[i]=false;
		rt_st.medida[i]=0;
	};
	rt_st.irq_aninhadas=-1;
	rt_st.irq_handle=false;
	rt_st.scheduler=false;
	rt_st.tsc_khz=tsc_khz;
	rt_st.processo_tr=current->pid;
	return 0;
}
```

H.3.8 Finaliza a Medição

```
int fim_medida(void) {
	int i;

	rt_st.medida_on=false;

	// Coloca valores padrão na estrutura
	rt_st.irq_cod=-1;
	rt_st.flag_medida_pit=false;
	rt_st.medida_pit=0;
	for (i=0; i<20; i++) {
		rt_st.flags_medida[i]=false;
		rt_st.medida[i]=0;
	};
	rt_st.irq_aninhadas=-1;
	rt_st.irq_handle=false;
	rt_st.scheduler=false;
	rt_st.tsc_khz=0;
	rt_st.processo_tr=0;
	return 0;
}
```

H.3.9 Inicializa Estrutura na Carga do Sistema

```
void boot_medida(void) {
      int i;

      // Coloca valores padrão na estrutura
      printk(KERN_ALERT "Estrutura Medicao - BOOT OK \n");

      rt_st.medida_on=false;
      rt_st.irq_cod=-1;
      rt_st.flag_medida_pit=false;
      rt_st.medida_pit=0;
      for (i=0; i<20; i++) {
            rt_st.flags_medida[i]=false;
            rt_st.medida[i]=0;
      };
      rt_st.irq_aninhadas=0;
      rt_st.irq_handle=false;
      rt_st.scheduler=false;
      rt_st.tsc_khz=0;
      rt_st.pit_ctr = CLOCK_TICK_RATE;
      rt_st.pit_hz = HZ;
      rt_st.pit_latch = LATCH;
      rt_st.processo_tr=0;
}
```

H.3.10 Limpa Estrutura entre Medidas

```
void limpa_est_leitura(void) {
      int i;

      rt_st.flag_medida_pit=false;
      rt_st.medida_pit=0;
      for (i=0; i<20; i++) {
            rt_st.flags_medida[i]=false;
            rt_st.medida[i]=0;
      };
      rt_st.irq_aninhadas=-1;
      rt_st.irq_handle=false;
      rt_st.scheduler=false;
      // Libera a Medicao quando tudo esta pronto
      rt_st.medida_on=true;
}
```

H.3.11 Rotina Leitura de Dados

```
int le_dados(char *end_buffer) {

      wait_queue_t      wait;

      init_waitqueue_entry(&wait, current);     // Estrutura do nó
      current->state=TASK_UNINTERRUPTIBLE;      // Muda o estado
      add_wait_queue(&real_time_wait, &wait);   // Inclui processo fila

      limpa_est_leitura();
      mede_tsc(0,rt_st.irq_cod);                // Marca início de espera

      schedule();
```

```
      // Rotina volta a rodar
      remove_wait_queue(&real_time_wait, &wait);  // Exclui processo fila
      rt_st.medida[12]=get_cycles();              // Marca retomada


      // Passa endereço da estrutura para a chamada do sistema
      if (copy_to_user(end_buffer, (char*) &rt_st, sizeof(rt_st)))
                        return -EFAULT;
      return 0;
}
```

H.3.12 Rotina que Simula o *Driver*

```
void driver_medida(int codigo) {

      // Controle 1
      if (!rt_st.medida_on) return;
      if (codigo!=rt_st.irq_cod) return;
      if (rt_st.irq_handle) return;

      // Tira processo da fila de espera
      wake_up(&real_time_wait);
      set_need_resched();

      // Coontrole 2
      rt_st.irq_handle=true;
      rt_st.scheduler=true;
      return;
}
```

****

Apêndice I - Resultados Testes Pentium IV

I.1 Introdução

Neste apêndice são apresentados os resultados obtidos para algumas componentes da latência de software, no tratamento de uma interrupção E/S externa, para o Pentium IV. Também é apresentada uma breve descrição das limitações encontradas para a utilização do método nesse hardware.

I.2 Limitações Encontradas no Pentium IV

Ao testar o sistema operacional instrumentalizado no Pentium IV constatou-se que durante o processo de carga a IRQ0 era ativada e, posteriormente, desativada. Portanto, com o sistema operacional carregado a IRQ0 não estava mais ativa, o que inviabilizou a utilização do método para medir a latência total (ver Capítulo 6, item 6.4.3).

Observou-se que o sistema sinalizava seguidamente, mas não de forma periódica, a IRQ15 usada pelo Controlador de Disco EIDE – 2ª cadeia que, nesta máquina em particular, era o disco rígido. Optou-se então por monitorar essa interrupção para realizar os testes para a latência de software e suas componentes.

Durante testes realizados constatou-se que a IRQ15 era tratada com as interrupções habilitadas como pode ser observado nos gráficos da figura I.1. Apesar dos gráficos apresentarem erros experimentais, em decorrência do algoritmo usado para medição (ver Capítulo 6, item 6.4.4), demonstram o aninhamento de interrupções durantes as medições realizados.

Pelo exposto, no Pentium IV só algumas componentes da latência de software puderam ser medidas de forma correta (ver Capítulo 6, item 6.4.6).

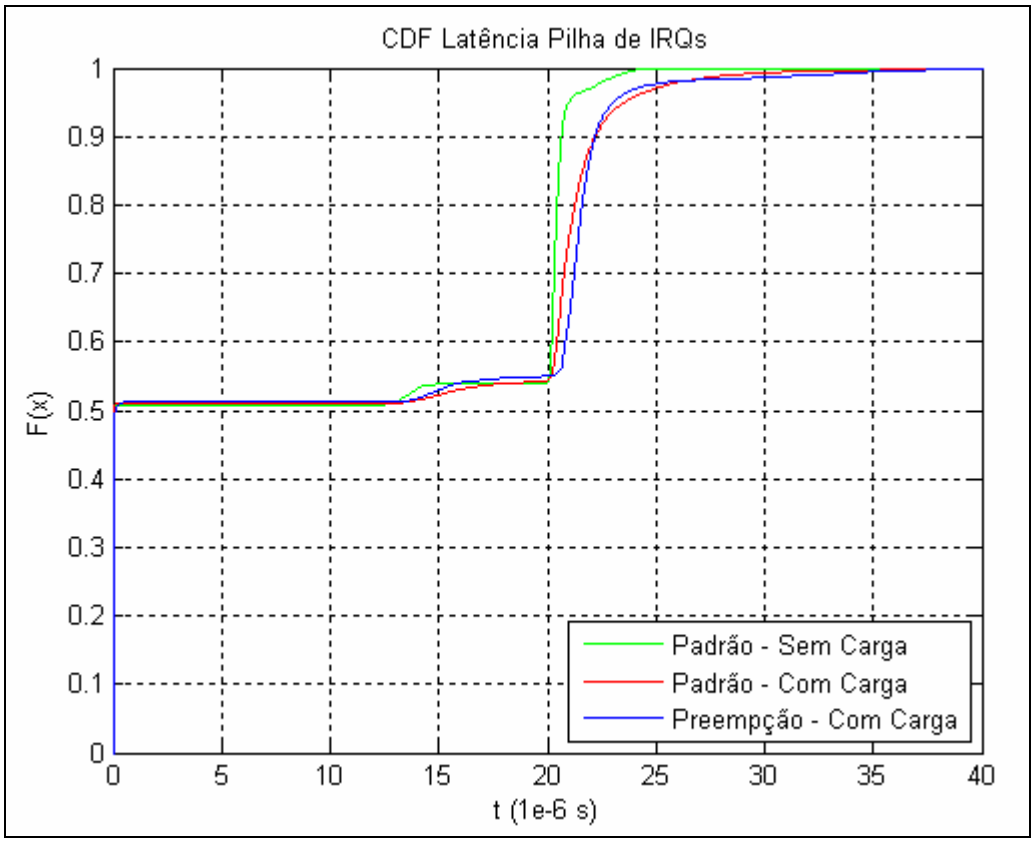

Figura I.1 – CDF aproximada da latência de pilha de IRQs
Valores com erros experimentais

I.3 Resultados

A configuração da máquina, a metodologia utilizada e os programas de cargas estão descritos no Capítulo 6, item 6.6. Os resultados apresentados são os daquelas componentes da latência de software que não são afetadas pelo fato da interrupção monitorada ser tratada com as interrupções habilitadas, como colocado no Capítulo 6, item 6.4.6.

Os valores obtidos são qualitativamente semelhantes ao obtidos no Capítulo 6, por isso não é apresentada nenhuma análise.

I.3.1 Latência de Entrada no Tratador de Interrupções

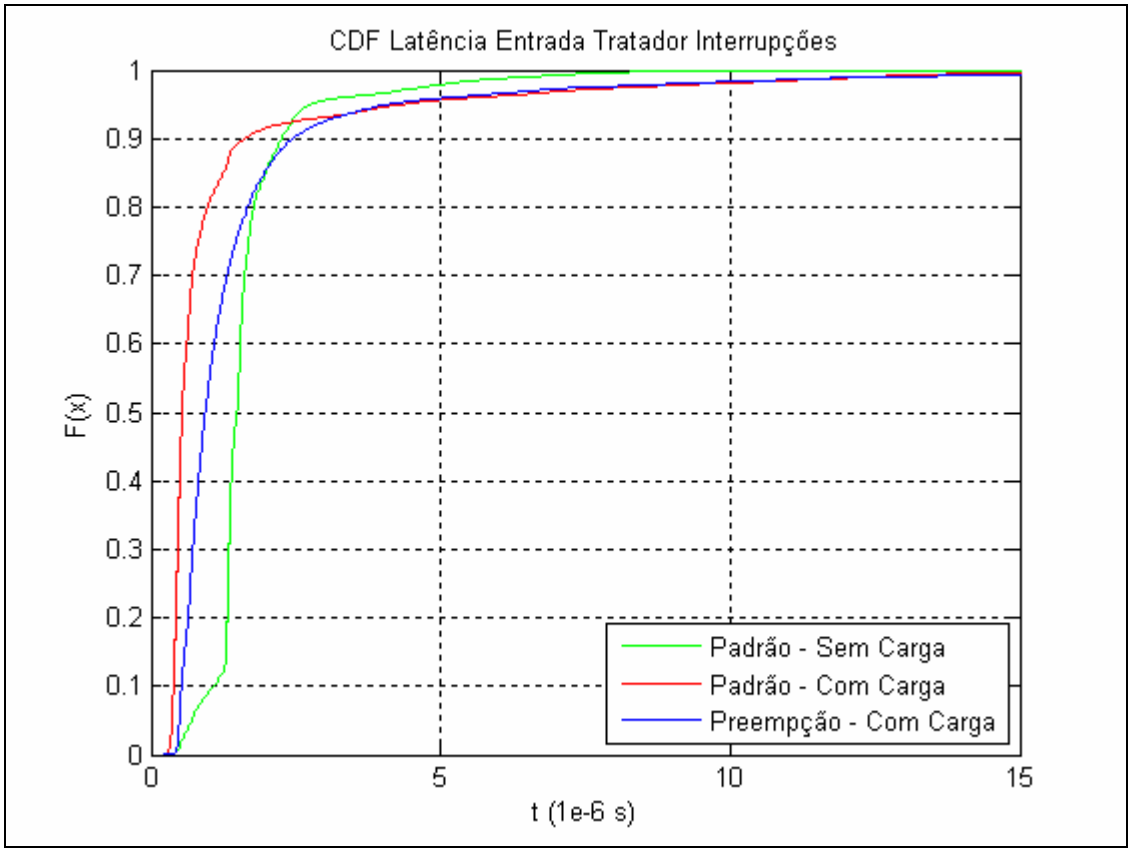

Figura I.2 – CDF da latência de entrada no tratador de interrupções
Obs.: Gráfico da CDF truncado

Tabela I.1 – Estatísticas da latência de entrada no tratador de interrupções

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	1,67	1,15	1,52
Máximo	18,2	42,8	37,5
Mínimo	0,301	0,265	0,214
Mediana	1,48	0,533	0,938

I.3.2 Latência de Saída do Tratador de Interrupções

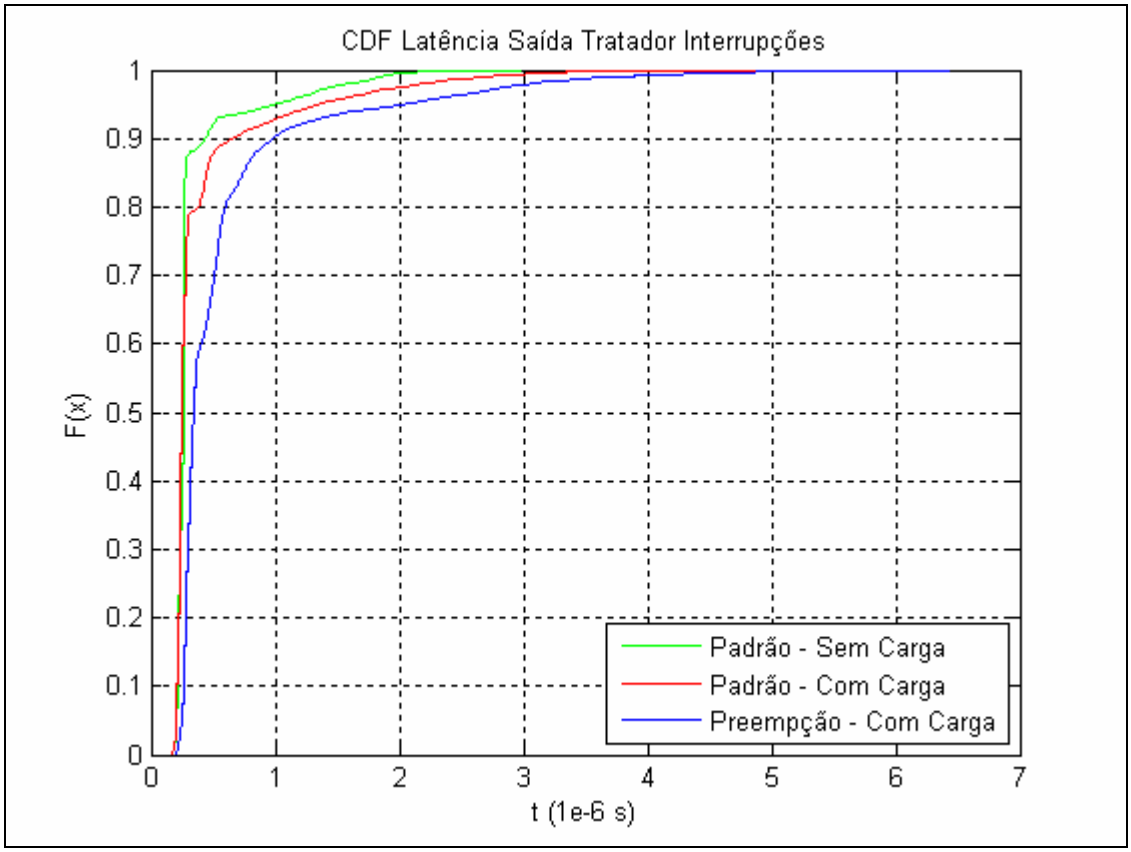

Figura I.3 – CDF da latência de saída do tratador de interrupções

Tabela I.2 – Estatísticas da latência de saída do tratador de interrupções

	Padrão Sem Carga (µs)	Padrão Com Carga (µs)	Preempção Com Carga (µs)
Média	0,333	0,398	0,581
Máximo	2,99	5,37	6,44
Mínimo	0,152	0,146	0,184
Mediana	0,261	0,248	0,345

I.3.3 Latência de *Deferrable Functions*

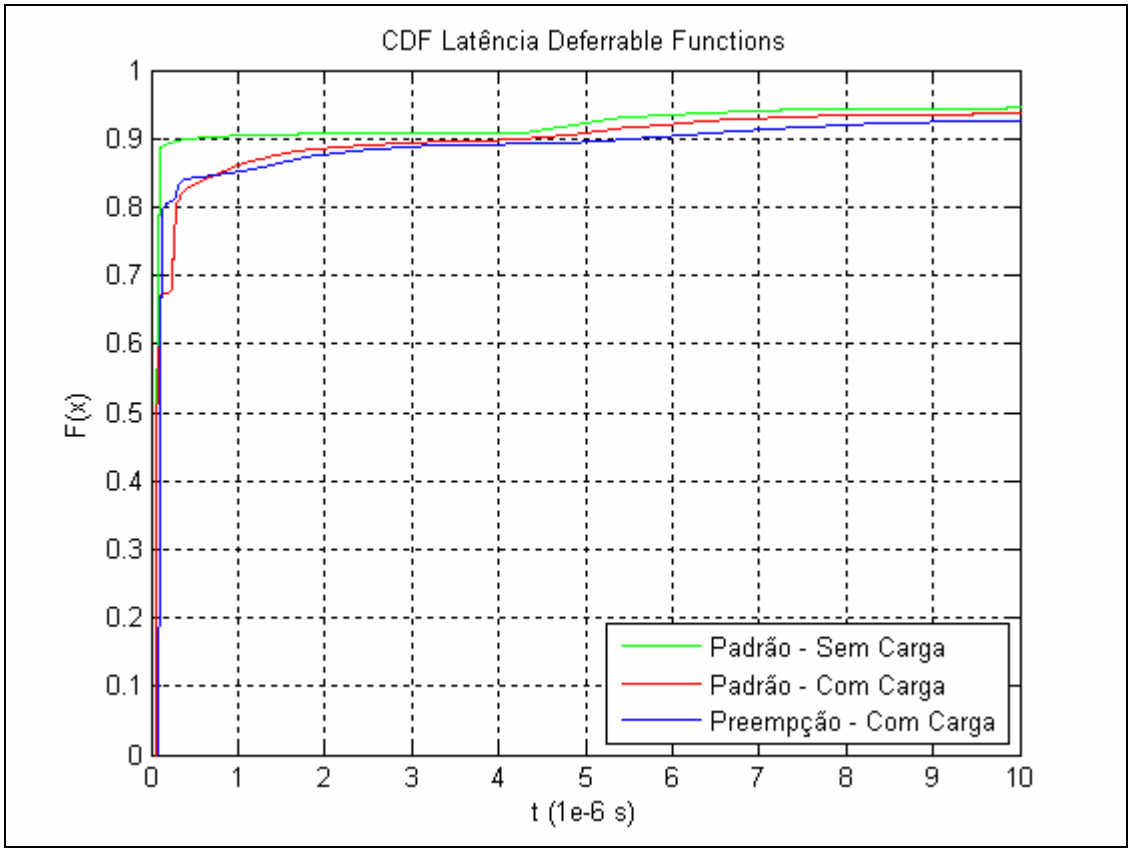

Figura I.4 – CDF da latência de d*eferrable functions*
Obs.: Gráfico da CDF truncado

Tabela I.3 – Estatísticas da latência de *deferrable functions*

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	3,36	4,15	4,62
Máximo	70,3	266,8	170,0
Mínimo	0,0559	0,0559	0,0619
Mediana	0,0679	0,0679	0,110

I.3.4 Latência de Preempção do Núcleo Desabilitada

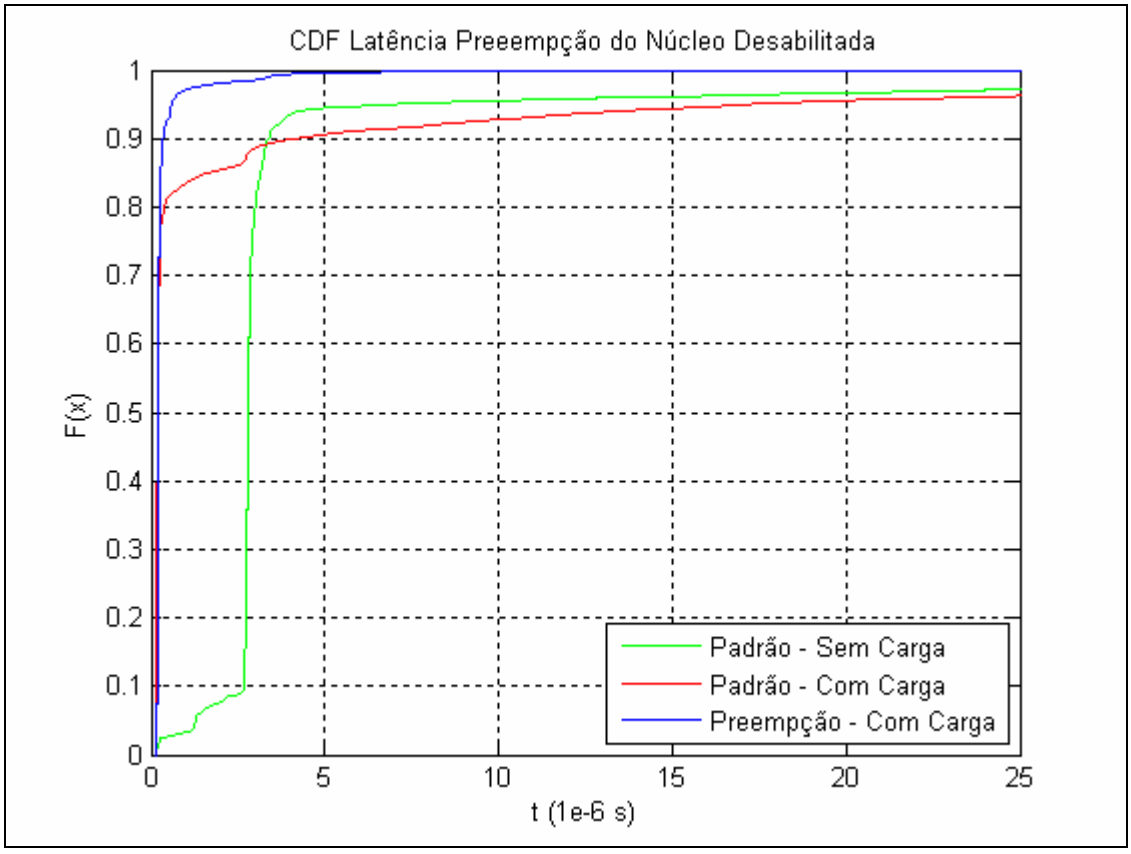

Figura I.5 – CDF da latência de preempção do núcleo desabilitada
Obs.: Gráfico da CDF truncado

Tabela I.4 – Estatísticas da latência de preempção do núcleo desabilitada

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	4,49	5,08	0,329
Máximo	253,9	497,7	99,5
Mínimo	0,128	0,114	0,130
Mediana	2,79	0,188	0,198

I.3.5 Latência de Escalonamento

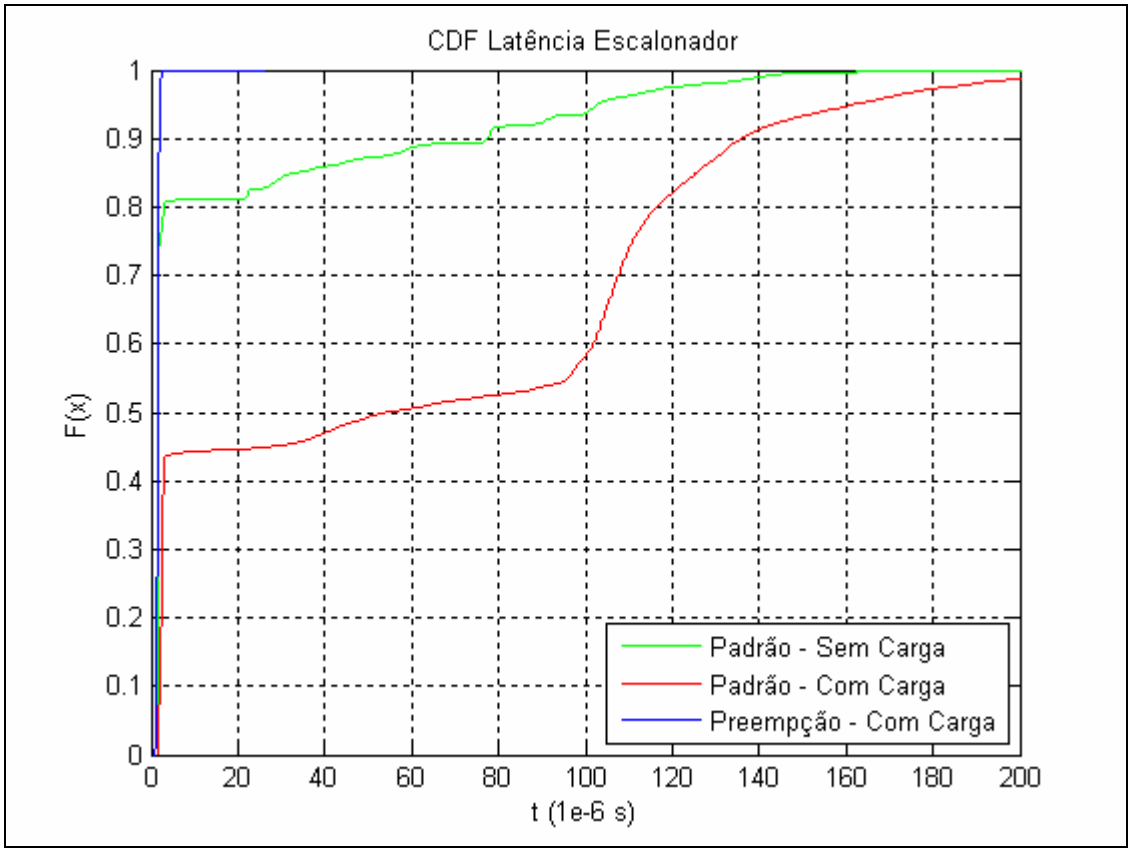

Figura I.6 – CDF da latência de escalonamento
Obs.: Gráfico da CDF truncado

Tabela I.5 – Estatísticas da latência de escalonamento

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	16,0	63,7	1,55
Máximo	457,9	488,5	26,5
Mínimo	1,35	1,37	0,671
Mediana	1,57	54,4	1,58

I.3.6 Latência de Troca de Contexto

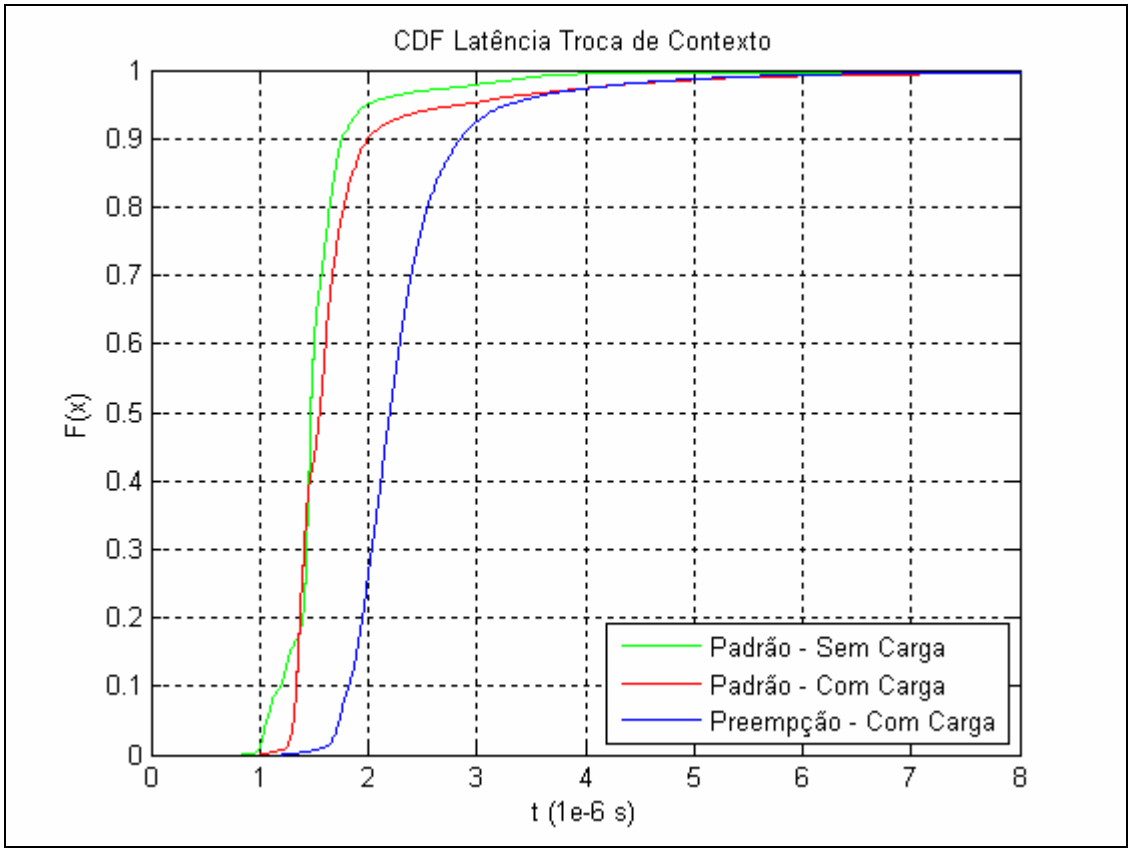

Figura I.7 – CDF da latência de troca de contexto
Obs.: Gráfico da CDF truncado

Tabela I.6 – Estatísticas da latência de troca de contexto

	Padrão Sem Carga (µs)	Padrão Com Carga (µs)	Preempção Com Carga (µs)
Média	1,57	1,81	2,39
Máximo	244,8	240,5	79,6
Mínimo	0,814	1,00	1,21
Mediana	1,47	1,55	2,20

****

Apêndice J – Alterações no Núcleo para Implementar o Sistema Operacional PDS

J.1 Estrutura: `pds_real_time_struct`

```
struct pds_real_time_struct {

	int		modo;			// Modo 0-Normal 1-Treino 2-PDS

	int		monitorada[50];		// IRQ esta sendo monitorada
	cycles_t	medida[50][3];// Valores das medidas

	int		irq_cod;		// Codigo da IRQa


	bool		overflow_tsc; // TSC Zerou
	cycles_t	prox_irq_amost;		// Próxima IRQa


	int		fila_conta;		// Tamanho fila IRQs
	int		fila_irqs[50];// Fila IRQs

	cycles_t	periodo_amost;// Periodo de amostragem


	int		pds_estado;		// 0-Bloq. 1-Pronto 2-Exec. 9-
Trans.
	struct task_struct *pds;		// Processo PDS
	struct task_struct *npds;	// Processo NPDS
};
```

J.2 Chamada ao Sistema 189

```
asmlinkage int sys_medelat2(int ope, char __user *buff) {
	int ok;


	switch (ope) {

		/* ********** */
		/* Medicao TR */
		/* ********** */
		case 0:

			// Rotina que trata a chamada ao sistema
			ok=le_dados(buff);
			return ok;


		/* ************* */
		/* Driver SO-PDS */
		/* ************* */
		case 1:
			ok=driver_le_amostra(buff);
			return ok;
```

```
                case 2:
                        ok=driver_eof();
                        return ok;
        }
        return 0;
}
```

J.3 Rotina: `do_IRQ`

```
unsigned int do_IRQ(struct pt_regs *regs)
{
        /* high bit used in ret_from_ code */
        unsigned int ax_int;
        int irq = ~regs->orig_ax;


        /* *** Verifica IRQ valida *** */
        if (unlikely((unsigned)irq >= NR_IRQS)) {
                printk(KERN_EMERG "%s: cannot handle IRQ %d\n",
                                        __FUNCTION__, irq);
                BUG();
        }


        /* *** Separa o tratamento das IRQs *** */
        if (ver_codigo_irq(irq)) {
                ax_int=do_IRQ_PDS_S(regs);
                return ax_int;
        }

        else {
                ax_int=do_IRQ_PDS_N(regs);
                return ax_int;
        }
}
```

J.4 Função: `ver_codigo_irq`

```
bool ver_codigo_irq(int codigo) {

        // SO-PDS --> Normal
        if (pds_rt_st.modo==0) return false;

        // SO-PDS --> Treino
        if (pds_rt_st.modo==1) return false;


        // Processo PDS --> PDS Transição
        if (pds_rt_st.pds_estado==9) return false;

        // Código não é IRQa
        if (codigo!= pds_rt_st.irq_cod) return false;

        return true;
}
```

J.5 Função: `ver_trata_irq`

```
bool ver_trata_irq(int codigo) {
       bool ok;
       cycles_t hora_atu;
       cycles_t hora_est;

       // SO-PDS --> Normal
       if (pds_rt_st.modo==0) return true;

       // SO-PDS --> Treino
       if (pds_rt_st.modo==1) return true;

       // Fila não está vazia --> Coloca IRQ na fila para manter ordem
       if (pds_rt_st.fila_conta>-1) {
              ok=fila_irq_in(codigo);

              // Overflow na fila --> Tratar IRQ
              if (!ok) return true;

              return false;
       }

       // Não há dados de treinamento --> Tratar IRQ
       if (pds_rt_st.medida[codigo][2]==0) return true;

       // Registrador TSC zerou. Não pode haver comparação --> Tratar IRQ
       if (pds_rt_st.overflow_tsc) return true;

       // Estimativa fim do tratamento da interrupção
       hora_atu=get_cycles();
       hora_est=hora_atu + pds_rt_st.medida[codigo][2];

      // Overflow, não pode haver comparação --> Tratar IRQ
       if (hora_atu>hora_est) return true;

       // Há tempo suficiente --> Tratar a IRQ
       if (pds_rt_st.prox_irq_amost>hora_est) return true;

       // Não há tempo suficiente --> Armazena IRQ na fila
       ok=fila_irq_in(codigo);

       // Overflow na fila --> TratarIRQ
       if (!ok) return true;

       // Nao Tratar IRQ
       return false;
}
```

J.6 Rotina: `mede_inicio_irq`

```
void mede_inicio_irq(int codigo) {

       // Verifica código para evitar erros no núcleo
       if (codigo<0) return;
       if (codigo>49) return;

       // SO-PDS --> Normal
       if (pds_rt_st.modo==0) return;
```

```
        // Coloca IRQ na Pilha de Aninhamento
        pds_rt_st.monitorada[codigo]++;

        // Pilha Aninhamento tem mais de um elemento --> IRQ aninhada
        if (pds_rt_st.monitorada[codigo]!=1) return;

        // Efetua Medida
        pds_rt_st.medida[codigo][0]=get_cycles();
}
```

J.7 Rotina: `mede_fim_irq`

```
void mede_fim_irq(int codigo) {

        // Verifica código para  evitar erros no núcleo
        if (codigo<0) return;
        if (codigo>49) return;

        // SO-PDS --> Normal
        if (pds_rt_st.modo==0) return;

        // Pilha Vazia --> Leitura nao comecou. Evita erros de transiente.
        if (pds_rt_st.monitorada[codigo]==0) return;

        // Tira IRQ da Pilha de Aninhamento
        pds_rt_st.monitorada[codigo]--;

        // Verifica erro na Pilha de Aninhamento
        if (pds_rt_st.monitorada[codigo]<0) {

                // Algum erro --> Concerta
                pds_rt_st.monitorada[codigo]=0;

                pds_rt_st.medida[codigo][0]=0;
                pds_rt_st.medida[codigo][1]=0;
                pds_rt_st.medida[codigo][2]=0;

                return;
        }

        // Pilha Aninamento não está vazia --> IRQ aninhada
        if (pds_rt_st.monitorada[codigo]!=0) return;

        //Efetua Medida
        pds_rt_st.medida[codigo][1]=get_cycles();

        // Calcula tempo IRQ - Verifica se TSC zerou
        if (pds_rt_st.medida[codigo][1]>pds_rt_st.medida[codigo][0]) {

                // TSC não zerou
             pds_rt_st.medida[codigo][2]=pds_rt_st.medida[codigo][1] -
             pds_rt_st.medida[codigo][0];
        }

        else {
                // TSC zerou --> Perde os dados
                pds_rt_st.medida[codigo][0]=0;
                pds_rt_st.medida[codigo][1]=0;
                pds_rt_st.medida[codigo][2]=0;
        }
```

```
}
```

J.8 Rotina: mask_ack_irqPDS

```
void mask_ack_irqPDS(struct irq_desc *desc, int irq)
{
	if (desc->chip->mask_ack)
		desc->chip->mask_ack(irq);
	else {
		desc->chip->mask(irq);
		desc->chip->ack(irq);
	}
}
```

J.9 Função: do_IRQ_PDS_S

```
unsigned int do_IRQ_PDS_S(struct pt_regs *regs)
{
	struct pt_regs *old_regs;
	/* high bit used in ret_from_ code */
	int irq = ~regs->orig_ax;
	struct irq_desc *desc = irq_desc + irq;

	// Salva Registradores
	old_regs = set_irq_regs(regs);

	// Atualiza proximo instante amostragem
	atualiza_prox_irqa();

	// Trata IRQa
	hlevel_irq_PDS(irq, desc, 0);

	// Recupera Registradores
	set_irq_regs(old_regs);

	// Bloqueia IRQS
	bloqueia_irqs();

	// Chama Escalonador
	schedule_pds(0);

	// Bloqueia IRQS
	desbloqueia_irqs();

	return 1;
}
```

J.10 Rotina: atualiza_prox_irqa

```
void atualiza_prox_irqa(void) {
	cycles_t hora_atual;
	cycles_t hora_proxima;

	hora_atual=get_cycles();
	hora_proxima=hora_atual + pds_rt_st.periodo_amost;

	if (hora_proxima>hora_atual) pds_rt_st.overflow_tsc=false;
```

```
        else pds_rt_st.overflow_tsc=true;

        pds_rt_st.prox_irq_amost=hora_proxima;
}
```

J.11 Rotina: `hlevel_irq_PDS`

```
void hlevel_irq_PDS(unsigned int irq, struct irq_desc *desc, unsigned int
operacao)
{

        struct irqaction *action;
        irqreturn_t action_ret;

        // Tratamento IRQa --> Mask e Ack IRQa
        if (operacao==0) mask_ack_irqPDS(desc, irq);

        action = desc->action;
        if (unlikely(!action || (desc->status & IRQ_DISABLED))) return;

        action_ret = handle_IRQ_evPDS(irq, action);

        if (operacao==0) {
                if (!(desc->status & IRQ_DISABLED) && (desc->chip->unmask)
                        desc->chip->unmask(irq);
        }
}
```

J.12 Função: `handle_IRQ_evPDS`

```
irqreturn_t handle_IRQ_evPDS(unsigned int irq, struct irqaction *action)
{
        irqreturn_t ret, retval = IRQ_NONE;
        unsigned int status = 0;

        ret = action->handler(irq, action->dev_id);
        if (ret == IRQ_HANDLED)      status |= action->flags;
        retval |= ret;

        // Alteracao PDS - Simula acao da ISR da Placa
        if (ret == IRQ_HANDLED) driver_pds_isr(irq);

        return retval;
}
```

J.13 Rotina: `driver_pds_isr`

```
void driver_pds_isr(int codigo) {

        // Codigo nao confere
        if (codigo!= pds_rt_st.irq_cod) return;

        // Processo PDS nao está bloqueado --> Subamostragem
        if (pds_rt_st.pds_estado==1) return;
        if (pds_rt_st.pds_estado==2) return;

        // Tira processo da fila de espera
```

```
        wake_up(&pds_tr_wait);
        pds_rt_st.pds_estado=1;

        return;
}
```

J.14 Rotina: `bloqueia_irqs`

```
void bloqueia_irqs(void){
        int irq;
        struct irq_desc *desc;

        for (irq=0; irq<NR_IRQS; irq++) {
                desc=irq_desc+irq;
                desc->chip->mask(irq);
        }
}
```

J.15 Rotina: `desbloqueia_irqs`

```
void desbloqueia_irqs(void){
        int irq;
        struct irq_desc *desc;

        for (irq=0; irq<NR_IRQS; irq++) {
                desc=irq_desc+irq;
                desc->chip->unmask(irq);
        }
}
```

J.16 Grupo de Rotinas: `schedule_pds e duas de suas funções`

J.16.1 Rotina: `schedule_pds`

```
void schedule_pds(int ope)
{
        struct task_struct *prev, *next;
        struct rq *rq;
        int cpu;

        // Verifica subamostragem. Ver item I.16.2
        if (!ver_schedule_pds(ope)) return;

        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
        prev = rq->curr;

        spin_lock(&rq->lock);

        // Recupera o processo da estrutura. Ver item I.16.3
        next=pega_pds_task(ope, prev);

        if (next==prev) {
                spin_unlock(&rq->lock);
                return;
        }
```

```
        rq->curr = next;
        context_switch(rq, prev, next); /* unlocks the rq */
}
```

J.16.2 Função: `ver_schedule_pds`

```
bool ver_schedule_pds(int codigo) {

        /* *** (0) NPDS --> PDS *** */
        if (codigo==0) {

                // Não tem processo PDS --> Estado Transição Inicial
                if (pds_rt_st.pds==0) return false;


                //  Verifica Estado --> Subamostragem
                if (pds_rt_st.pds_estado==0) return false; // Bloqueado
                if (pds_rt_st.pds_estado==2) return false; // Execucao

                // Estado Transição Final
                if (pds_rt_st.pds_estado==9) {
                        pds_rt_st.pds=0;
                        pds_rt_st.npds=0;
                        return false;
                }

                // Troca de processo OK
                return true;
        }


        /* *** (1) PDS --> NPDS *** */

        // Nao tem NPDS --> Condição de ERRO (BUG)
        if (pds_rt_st.npds==0) return false;

        // Troca de processo OK
        return true;
}
```

J.16.3 Função: `pega_pds_task`

```
struct task_struct *pega_pds_task(int codigo, struct task_struct *processo)
{

        /* *** (0) NPDS --> PDS *** */
        if (codigo==0) {
                pds_rt_st.npds=processo;
                return pds_rt_st.pds;
        }

        /* *** (1) PDS --> NPDS *** */
        return pds_rt_st.npds;
}
```

J.17 Rotina: esvazia_fila

```
void esvazia_fila(void)
{
	int i;
	int irq_pds;
	unsigned int irq;

	struct irq_desc *desc;

	for (i=0; i<50; i++) {

		// Interrupcao Retorno
		irq_pds=fila_irq_out();
		if (irq_pds==-1) break;

		irq = (unsigned int) irq_pds;

		desc=irq_desc + irq;
		hlevel_irq_PDS(irq, desc, 1);
	}
}
```

J.18 Função: driver_le_amostra

```
int driver_le_amostra(char *end_buffer) {
	wait_queue_t  wait;

	if (pds_rt_st.npds==0) {

		// Nao tem processo NPDS --> Escalonador Normal
		pds_rt_st.pds=current;
		pds_rt_st.pds_estado=0;			// PDS --> Bloqueado


		// Coloca o processo bloqueado no Linux
		init_waitqueue_entry(&wait, current);
		current->state=TASK_INTERRUPTIBLE;
		add_wait_queue(&pds_tr_wait, &wait);


		// Chama escalonador do Linux
		schedule();
	}
	else {

		// Tem Processo NPDS --> Escalonador PDS
		pds_rt_st.pds=current;
		pds_rt_st.pds_estado=0;			// PDS --> Bloqueado


		// Coloca o processo bloqueado no Linux
	     init_waitqueue_entry(&wait, current);
		current->state=TASK_INTERRUPTIBLE;
		add_wait_queue(&pds_tr_wait, &wait);

		// Chama escalonador PDS - Retoma processo NPDS
		schedule_pds(1);
	}
```

```
        pds_rt_st.pds_estado=2;             // PDS --> Em Execucao
        remove_wait_queue(&pds_tr_wait, &wait);


        // Transfere os dados
       if (copy_to_user(end_buffer, (char*) &rt_st, sizeof(rt_st)))
                          return -EFAULT;

        return 0;
}
```

J.19 Função: `driver_eof`

```
int driver_eof(void) {
        wait_queue_t  wait;

        if (pds_rt_st.npds==0) {

                // Nao tem processo NPDS
                pds_rt_st.pds=0;
                pds_rt_st.pds_estado=9;
        }
        else {

                // Tem Processo NPDS --> Escalonador PDS
                pds_rt_st.pds_estado=9;


                // Coloca o processo em espera
                init_waitqueue_entry(&wait, current);
                current->state=TASK_INTERRUPTIBLE;
                add_wait_queue(&pds_tr_wait, &wait);

                schedule_pds(1);

                remove_wait_queue(&pds_tr_wait, &wait);
        }
        return 0;
}
```

J.20 Função: `ver_pds()`

```
bool ver_pds() {
        if (current==pds_rt_st.pds) return true
        return false
}
```

****

Apêndice K - Resultados 2º Teste no Pentium III

K.1 Introdução

Neste apêndice são apresentados os resultados obtidos para a latência da IRQ0 em um Pentium III, no qual foi usada a placa de rede para carregar o sistema com interrupções E/S externas.

K.2 Metodologia

A metodologia empregada foi semelhante à utilizada no Capítulo 6 (item 6.6). Foram feitas as três medidas descritas no Capítulo 6, item 6.6.1: “Padrão – Sem Carga”, “Padrão Com Carga” e “Preempção – Com Carga”. O volume de dados foi mesmo estabelecido no Capítulo 6, item 6.6.2.

A diferença entre o teste deste apêndice e o do Capítulo 6 está nos programas usados para a carga do sistema. Além dos dois programas descritos no Capítulo 6, item 6.6.1 (cálculo da inversa de uma matriz e leitura/gravação de um arquivo), foi executado um programa que ficava em laço fazendo o *download* de um arquivo de um servidor FTP público na Internet. Portanto o teste descrito neste apêndice tem uma carga maior de processamento em comparação ao do Capítulo 6.

K.3 Resultados

Os resultados são apresentados de forma semelhante à do Capítulo 6, item 6.7. Como os valores obtidos são qualitativamente semelhantes ao obtidos no Capítulo 6, em cada item é apresentada uma pequena comparação com os resultados mostrados naquele capítulo.

K.3.1 Latência de Hardware

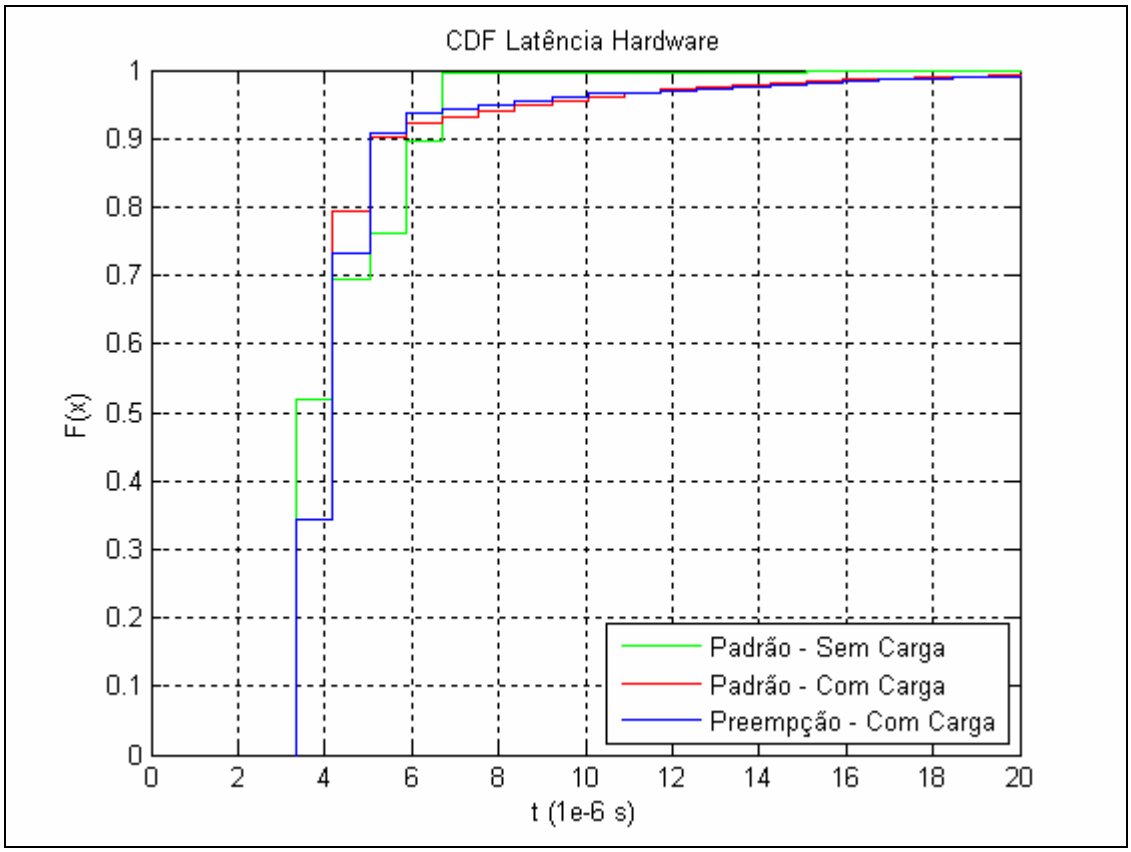

Figura K.1 – CDF da latência de hardware
Obs.: Gráfico da CDF truncado

Tabela K.1 – Estatísticas da latência de hardware

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	4,38	4,83	4,78
Máximo	52,0	269,0	253,1
Mínimo	3,35	3,35	3,35
Mediana	3,35	4,20	4,19

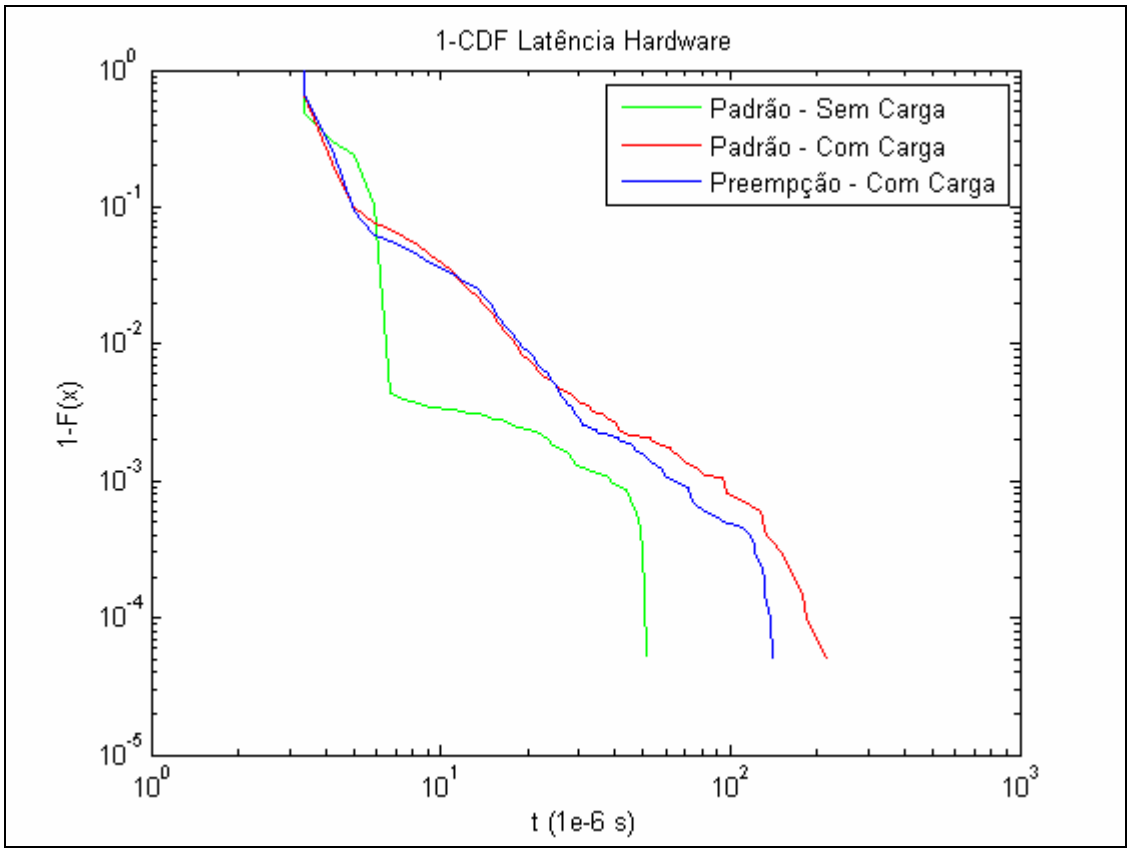

Figura K.2 – Gráfico (1-CDF) da latência de hardware

Análise:

Comparando os valores obtidos neste item com os obtidos no Capítulo 6 (item 6.7.1) pode-se ver que há um aumento da latência nas medidas com carga. Na medida “Padrão Com Carga” o aumento é de ~21% e na “Preempção Com Carga” é de ~16%.

Os resultados obtidos estão de acordo com o esperado já que a finalidade de usar a placa de rede nesta metodologia é obter um número maior de interrupções E/S externas do que na metodologia descrita no Capítulo 6 e, assim, obter maiores valores para esta latência.

K.3.2 Latência de Entrada no Tratador de Interrupções

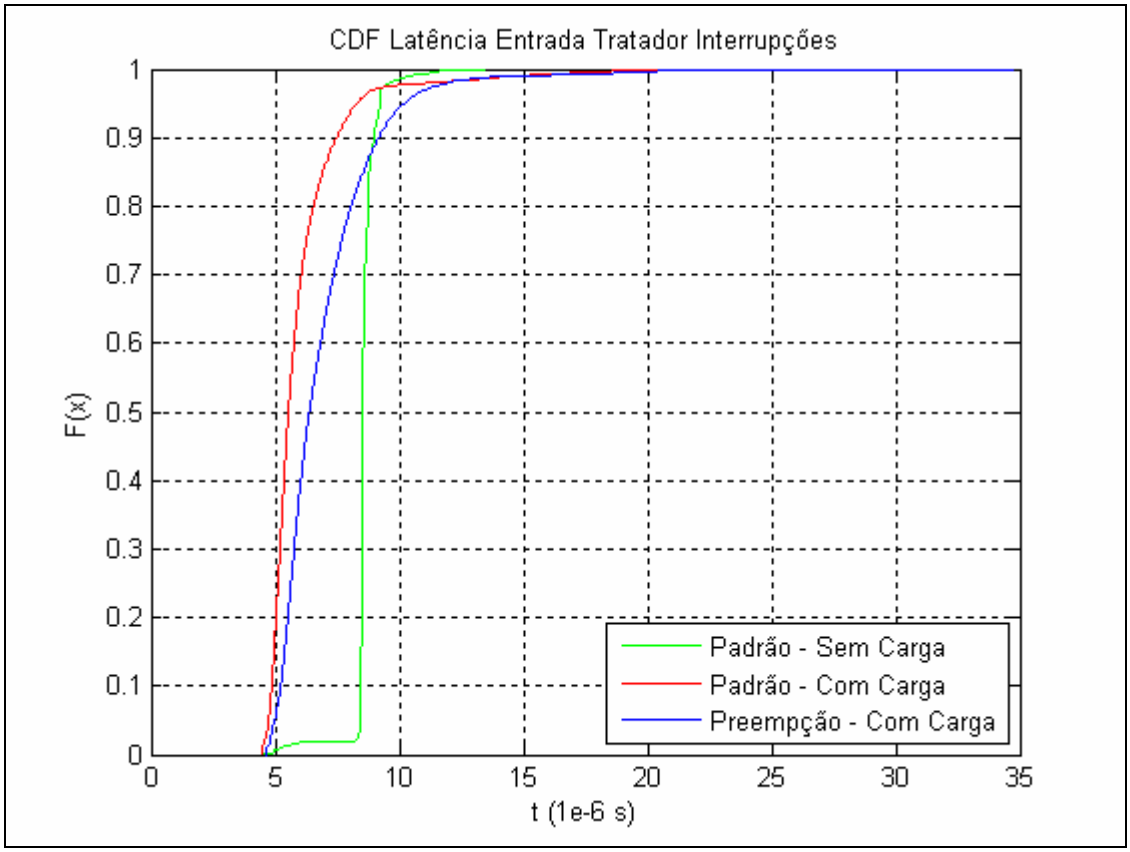

Figura K.3 – CDF da latência de entrada no tratador de interrupções

Tabela K.2 – Estatísticas da latência de entrada no tratador de interrupções

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	8,59	6,00	6,92
Máximo	13,47	27,9	34,8
Mínimo	4,50	4,42	4,60
Mediana	8,53	5,53	6,37

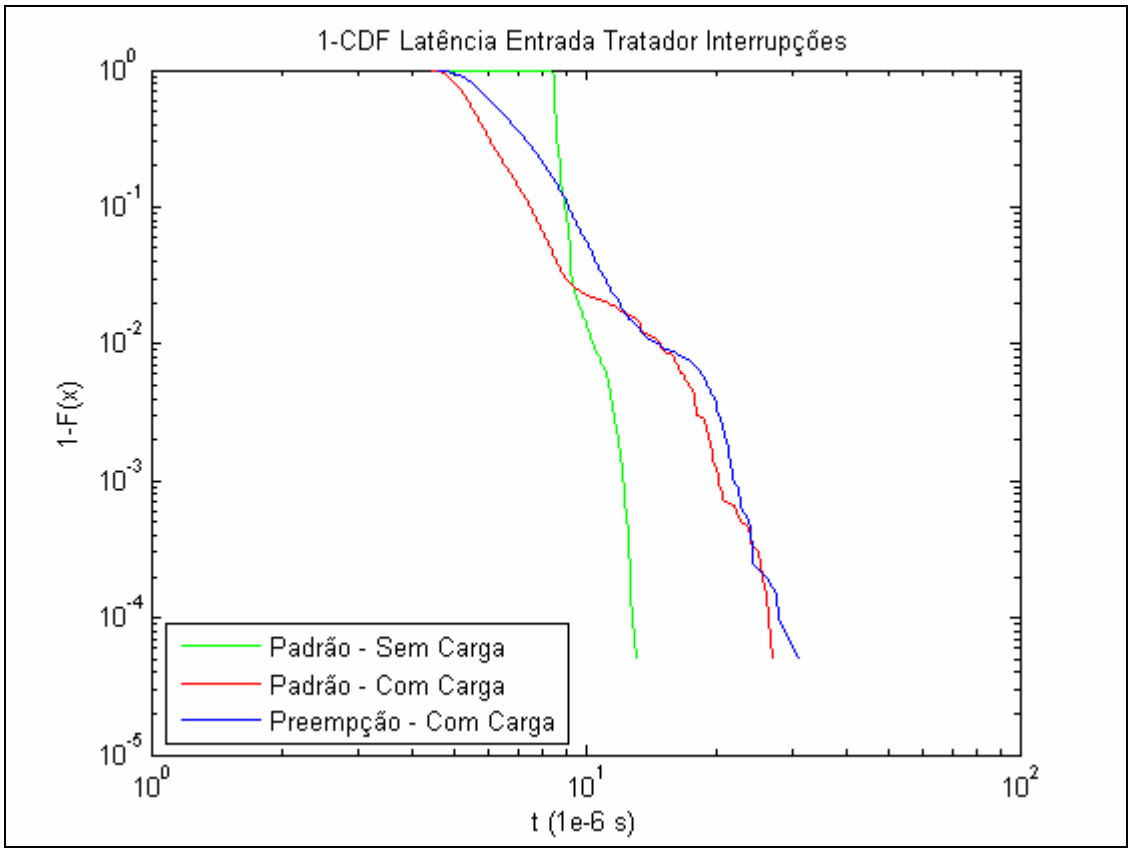

Figura K.4 – Gráfico (1-CDF) da latência de entrada no tratador de interrupções

Analise:

Observa-se que o valor obtido na medida “Padrão Com Carga” é ~0,7% maior do que o obtido no Capítulo 6 (item 6.7.2) e, por sua vez, o valor obtido na medida “Preempção Com Carga” é ~0,4% maior.

Esses resultados estão de acordo com o esperado, já que a influência da maior carga nesta latência se reflete em um número menor de coincidência de cache (*cache hit*).

K.3.3 Latência de Compartilhamento

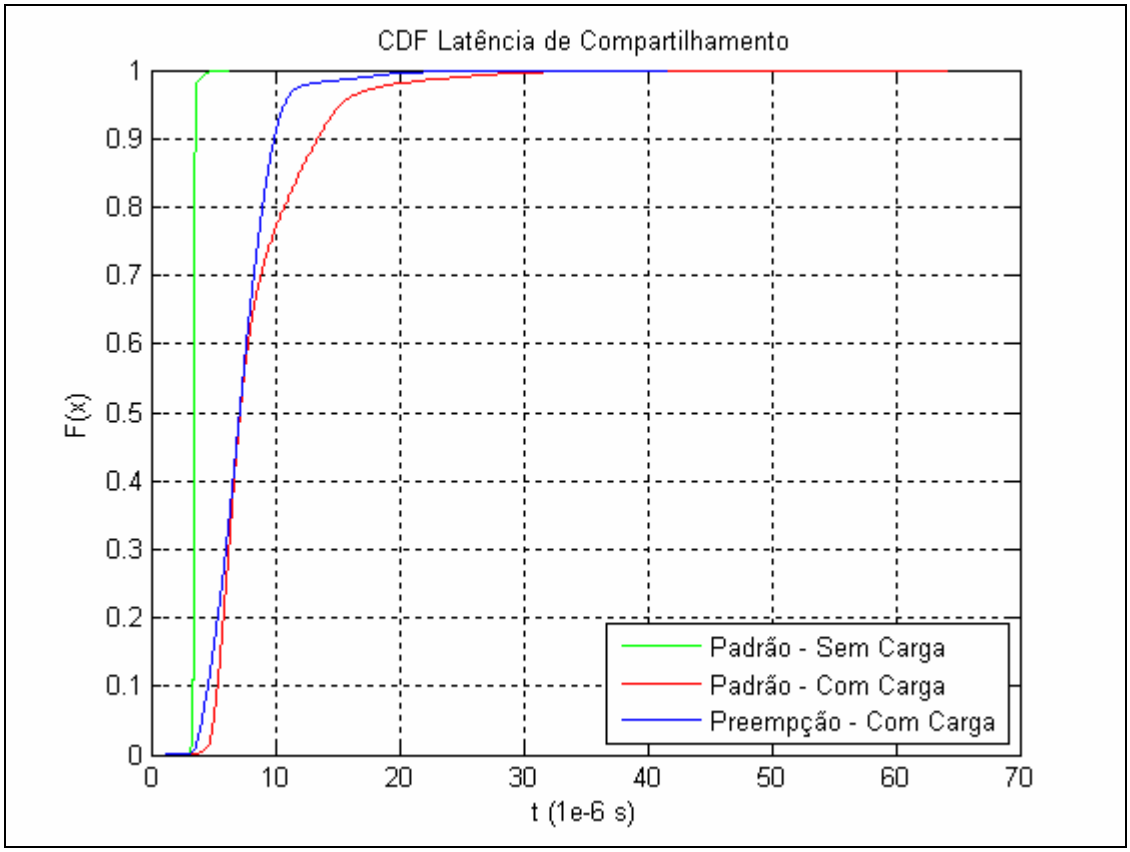

Figura K.5 – CDF da latência de compartilhamento

Tabela K.3 – Estatísticas da latência de compartilhamento

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	3,46	8,50	7,36
Máximo	6,40	64,2	41,6
Mínimo	1,10	1,13	1,10
Mediana	3,43	7,18	7,13

Análise:

A análise dos dados obtidos neste item é semelhante à do item K.3.2. Com um aumento de carga, a probabilidade das estruturas usadas para o tratamento das interrupções estarem na memória cache é menor (ver Capítulo 6, item 6.7.3) e, conseqüentemente, maior a latência de compartilhamento.

K.3.4 Latência de Saída do Tratador de Interrupções

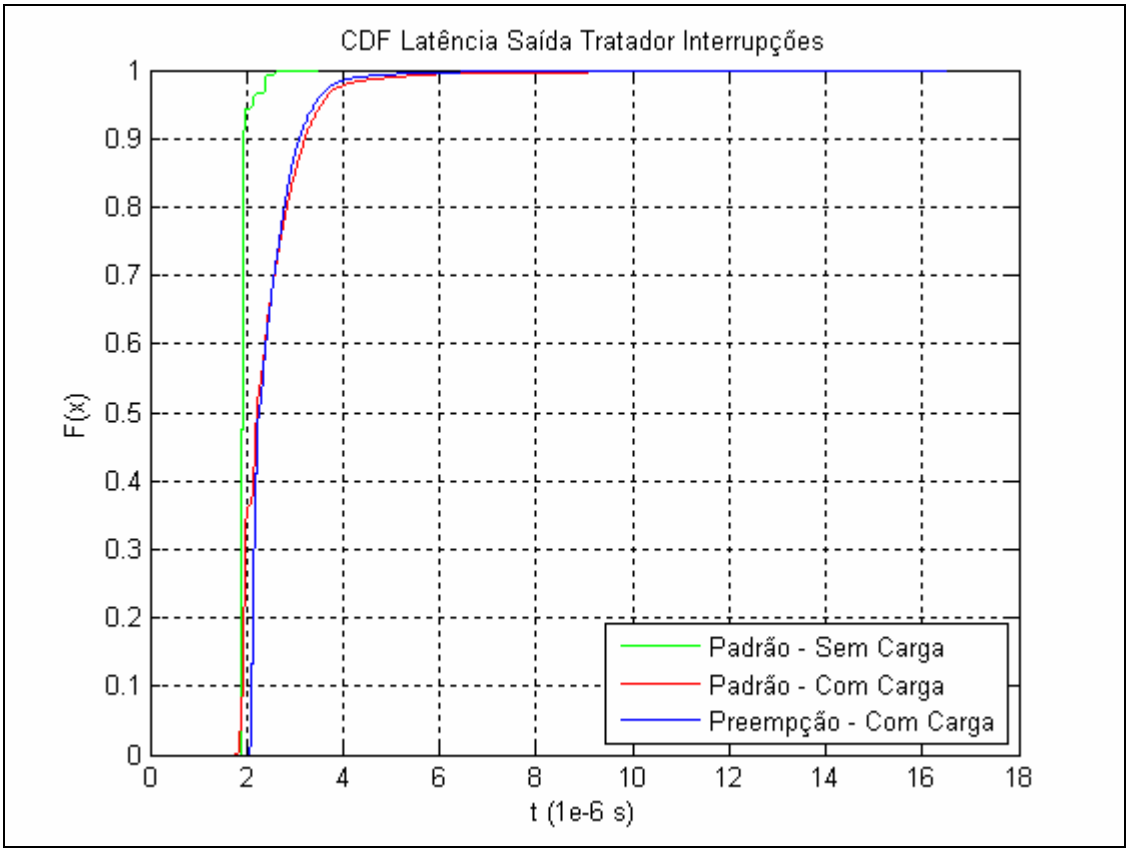

Figura K.6 – CDF da latência de saída do tratador de interrupções

Tabela K.4 – Estatísticas da latência de saída do tratador de interrupções

	Padrão Sem Carga (µs)	Padrão Com Carga (µs)	Preempção Com Carga (µs)
Média	1,94	2,45	2,50
Máximo	3,48	16,0	16,55
Mínimo	1,83	1,77	2,02
Mediana	1,91	2,21	2,26

Análise:

A interpretação dos dados obtidos neste item é semelhante à do item K.3.2.

K.3.5 Latência de Pilha de IRQs

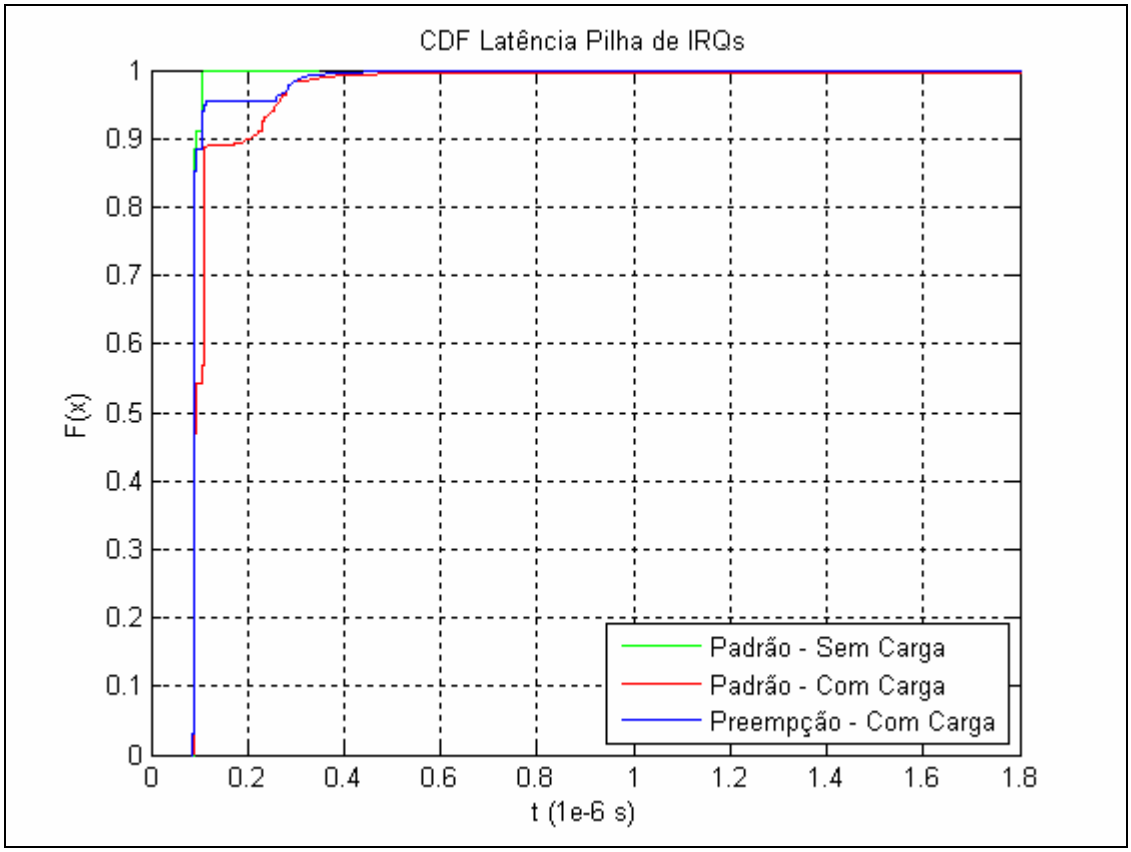

Figura K.7 – CDF da latência de pilha de IRQs
Obs.: Gráfico da CDF truncado

Tabela K.5 – Estatísticas da latência de pilha de IRQs

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	0,0926	2,94	0,133
Máximo	0,349	1,01e+3	166,5
Mínimo	0,0912	0,0841	0,0855
Mediana	0,0912	0,0926	0,0912

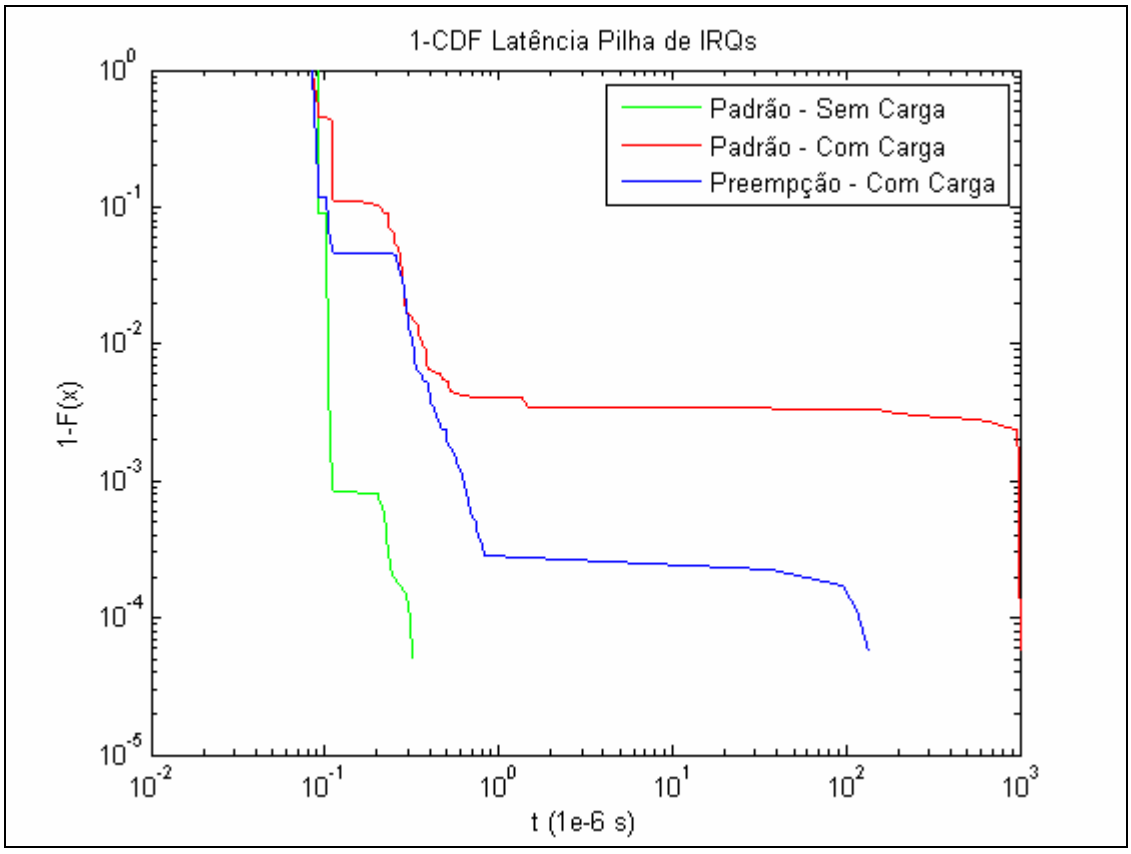

Figura K.8 – Gráfico (1-CDF) da latência de pilha de IRQs

Análise:

Como no item K.3.1, neste item era esperado um aumento da latência nas medidas com carga devido ao aumento do número de interrupções E/S externas, em comparação à metodologia usada no Capítulo 6 (ver item 6.7.5).

Havendo maior número de interrupções E/S externas sendo sinalizadas no sistema, maior é a probabilidade de aninhamento de interrupções e, portanto, maior a latência de Pilha de IRQs.

Observa-se que os dados obtidos estão de acordo com o esperado (ver tabelas 6.6 e K.5).

K.3.6 Latência de *Deferrable Functions*

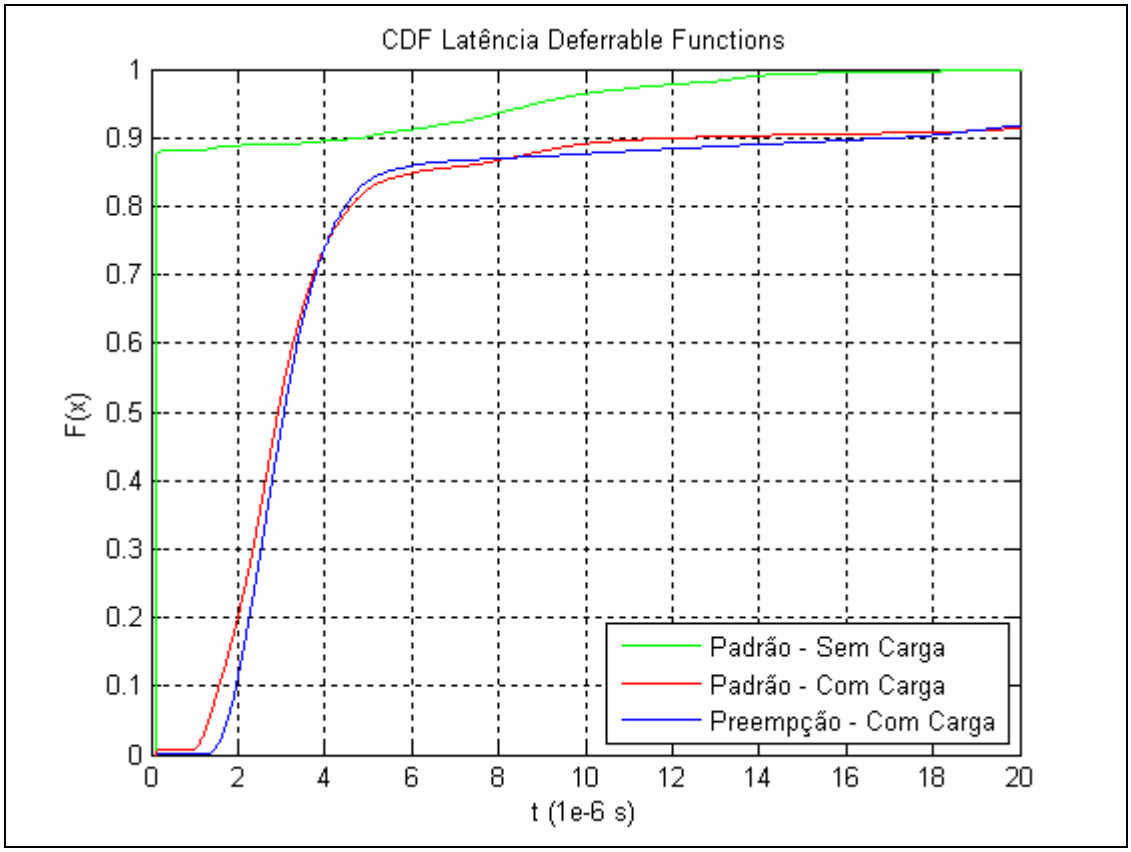

Figura K.9 – CDF da latência de d*eferrable functions*
Obs.: Gráfico da CDF truncado

Tabela K.6 – Estatísticas da latência de *deferrable functions*

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	1,14	13,4	13,9
Máximo	54,5	1,15e+3	1,02e+3
Mínimo	0,115	0,123	0,138
Mediana	0,123	2,92	3,07

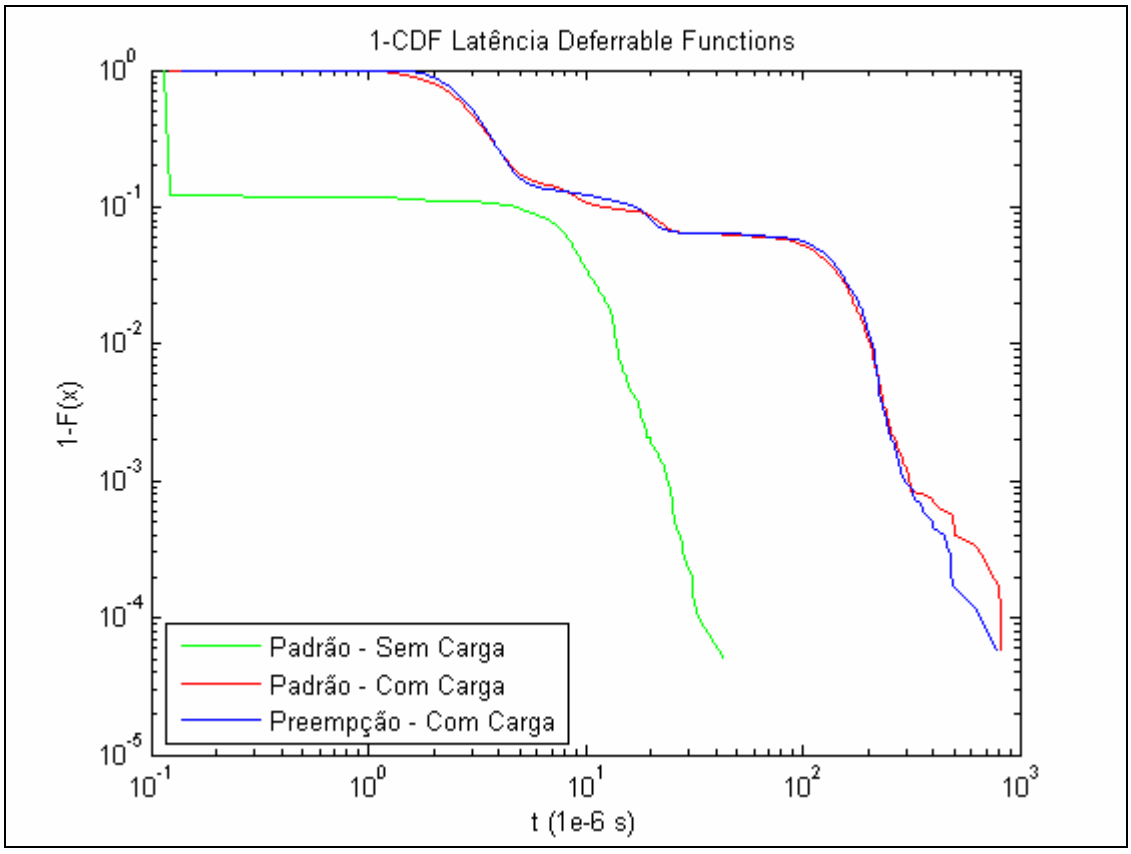

Figura K.10 – Gráfico (1-CDF) da latência de d*eferrable functions*

Análise:

Observa-se que os dados obtidos estão de acordo em comparação aos obtidos no Capítulo 6, item 6.7.6.

Na metodologia usada neste teste há um aumento de *deferrable functions* devido ao processo que executa o *download* do arquivo pela rede e, conseqüentemente, era de se esperar uma maior latência de *deferrable functions* nas medidas com carga em comparação à obtida no Capítulo 6 (item 6.7.6), como foi observada.

K.3.7 Latência de Preempção do Núcleo Desabilitada

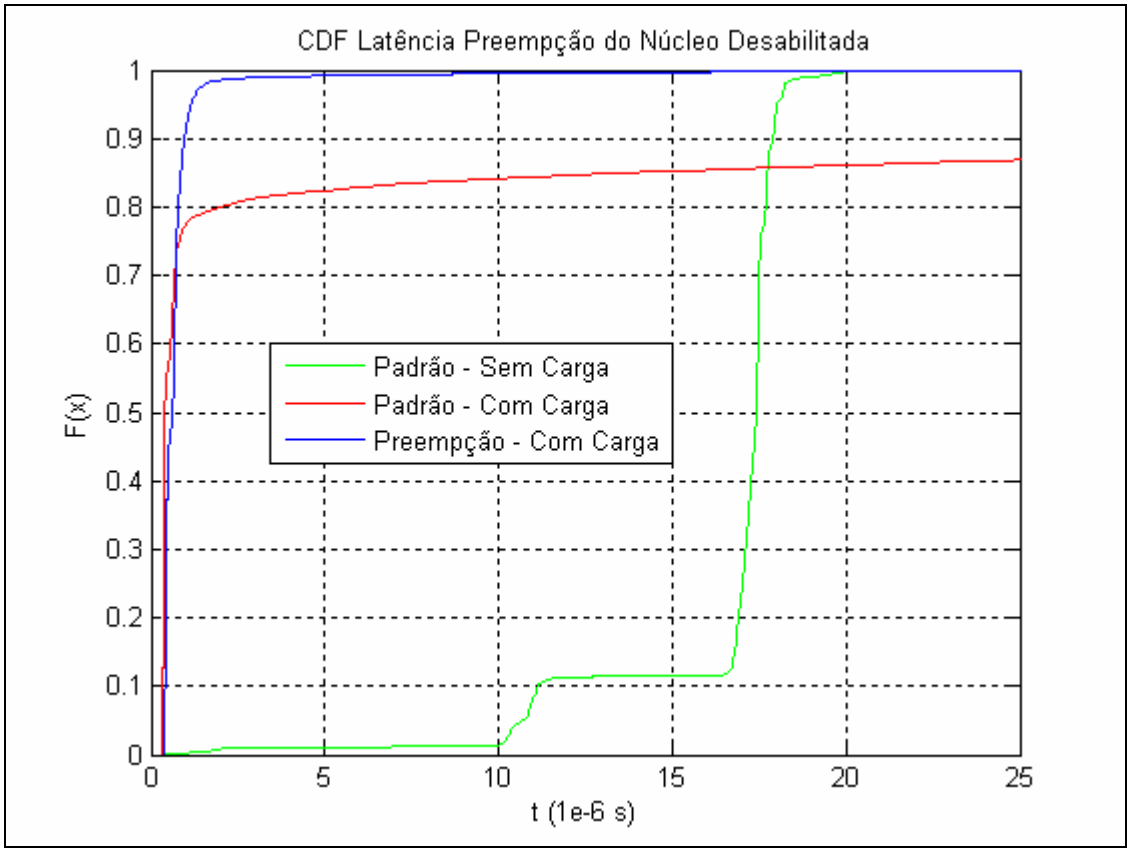

Figura K.11 – CDF da latência de preempção do núcleo desabilitada
Obs.: Gráfico da CDF truncado

Tabela K.7 – Estatísticas da latência de preempção do núcleo desabilitada

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	16,65	25,63	0,815
Máximo	664,7	1,42e+3	243,1
Mínimo	0,314	0,265	0,331
Mediana	17,42	0,396	0,617

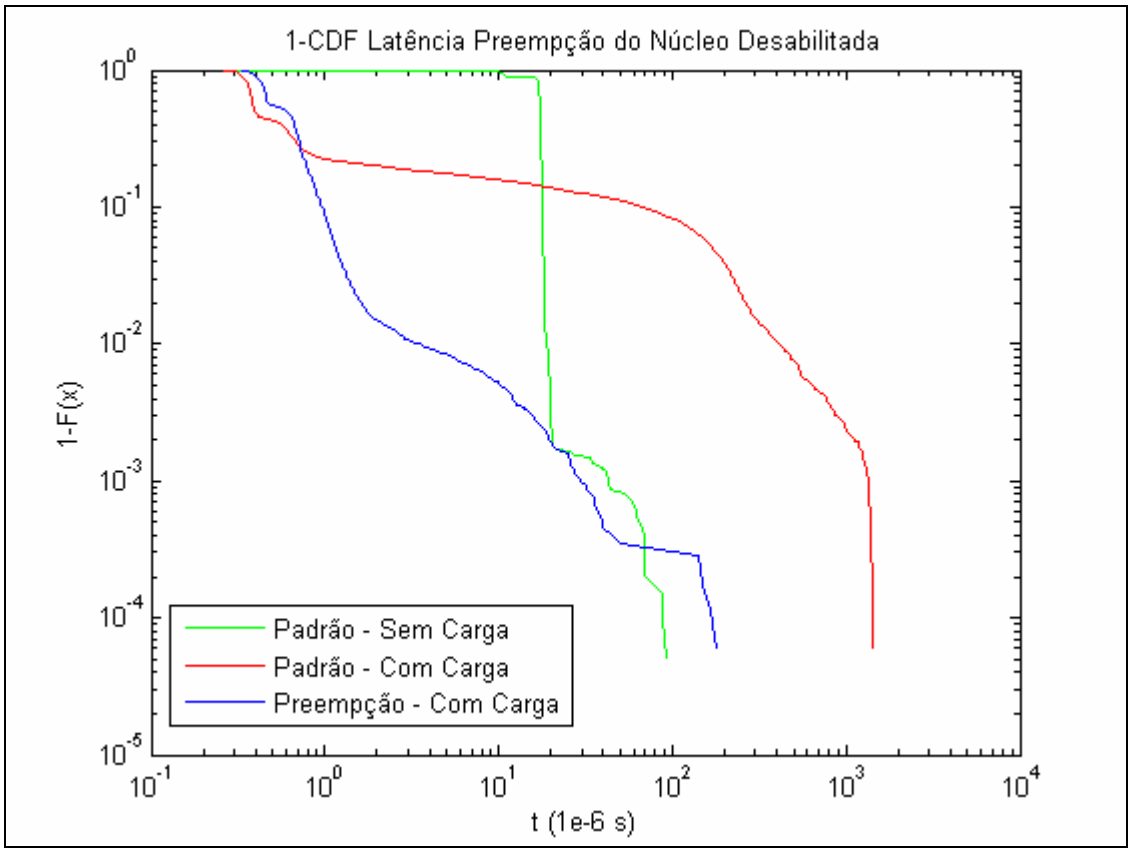

Figura K.12 – Gráfico (1-CDF) da latência de preempção do núcleo desabilitada

Análise:

Aumentando a carga do sistema é de se esperar um aumento desta latência se comparada à obtida no Capítulo 6 (item 6.7.7) nas medidas com carga.

Na medida “Preempção Com Carga” pode-se ver que o aumento é da ordem de ~6%, que se deve ao fato do Linux 2.6.25 ter trechos do núcleo que não sofrem preempção.

Na medida “Padrão Com Carga”, embora não se reflita na média devido a valores extremos, o aumento pode ser observado comparando os gráficos das figuras 6.18 e K.12 ou comparando a mediana que teve um aumento de ~7%.

.

K.3.8 Latência de Escalonamento

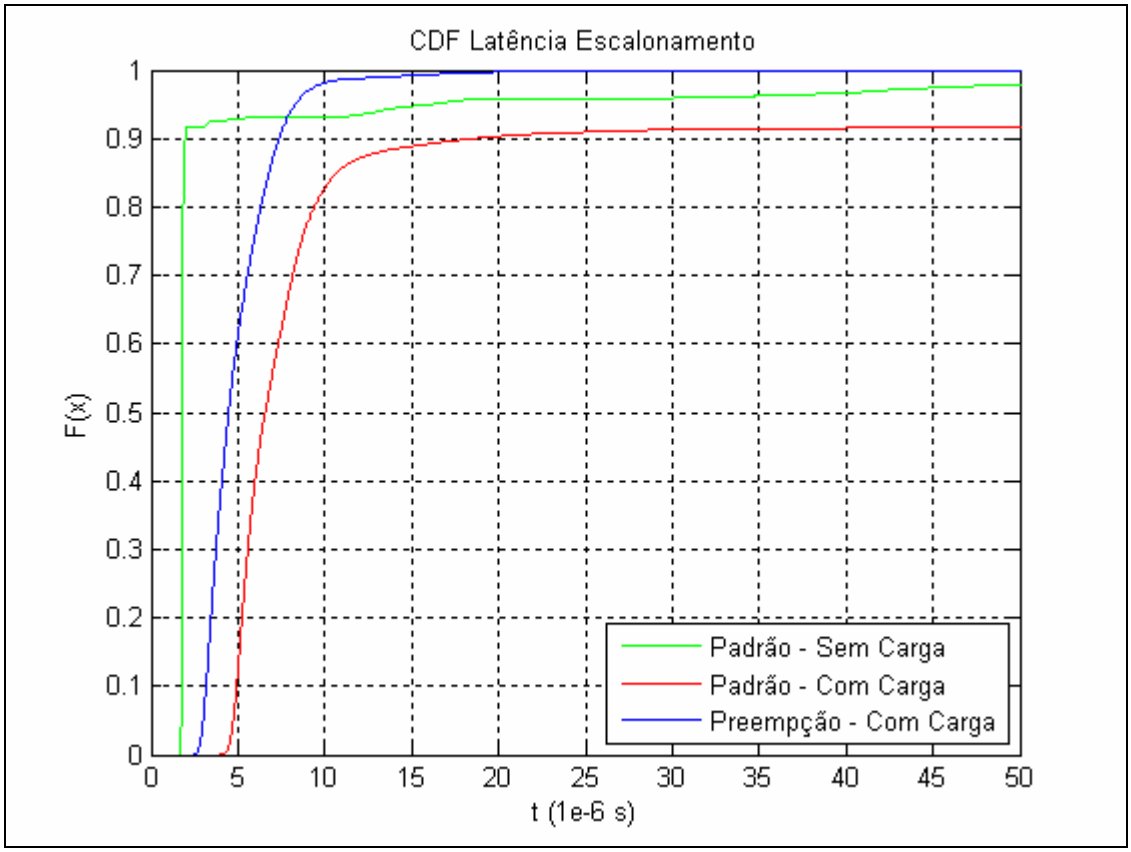

Figura K.13 – CDF da latência de escalonamento
Obs.: Gráfico da CDF truncado

Tabela K.8– Estatísticas da latência de escalonamento

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	6,75	40,93	5,09
Máximo	531,0	1,42e+3	439,4
Mínimo	1,54	3,98	2,43
Mediana	1,81	6,58	4,46

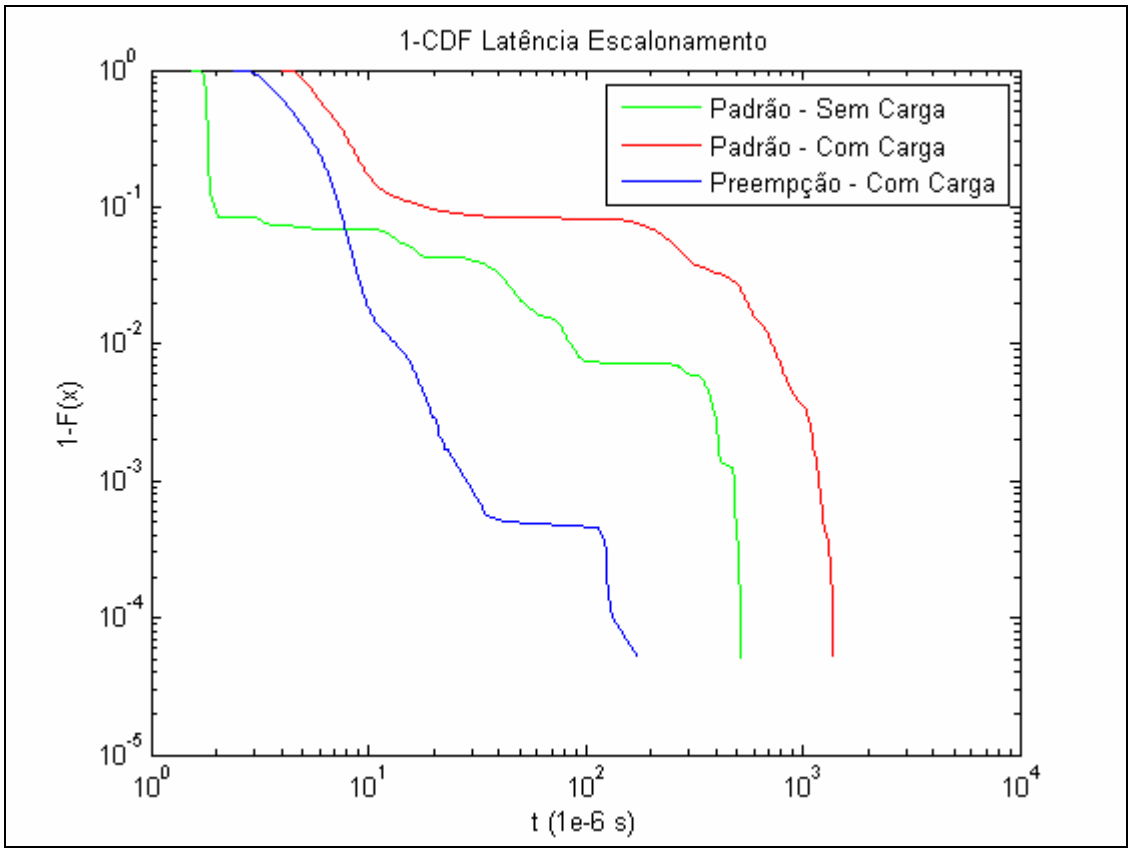

Figura K.14 – Gráfico (1-CDF) da latência de escalonamento

Análise:

Ao haver mais um processo em execução no sistema era de se esperar um aumento na latência de escalonamento se comparada à obtida no Capítulo 6 (item 6.7.8) nas medidas com carga.

Para a medida “Padrão Com Carga” o aumento foi de ~421% e para a medida “Preempção Com Carga” foi de ~39%. Esse aumento menor na segunda era esperado, já que nessa medida são usados os requisitos de tempo real do Linux que dão maior prioridade ao processo do Aplicativo de Gerenciamento (ver Capítulo 6, item 6.6.1).

K.9 Latência de Troca de Contexto

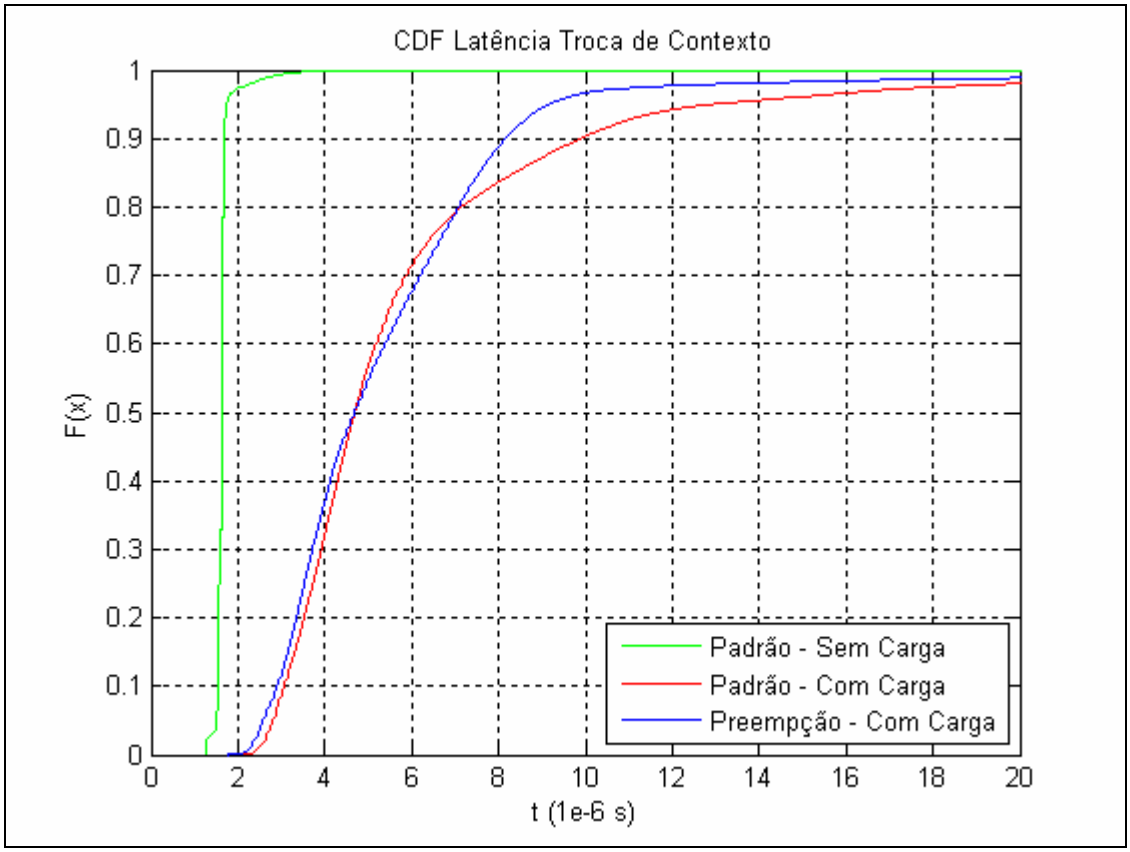

Figura K.15 – CDF da latência de troca de contexto
Obs.: Gráfico da CDF truncado

Tabela K.9 – Estatísticas da latência de troca de contexto

	Padrão Sem Carga (µs)	Padrão Com Carga (µs)	Preempção Com Carga (µs)
Média	1,70	7,45	6,51
Máximo	157,25	523,5	407,4
Mínimo	1,21	2,10	1,78
Mediana	1,63	4,69	4,67

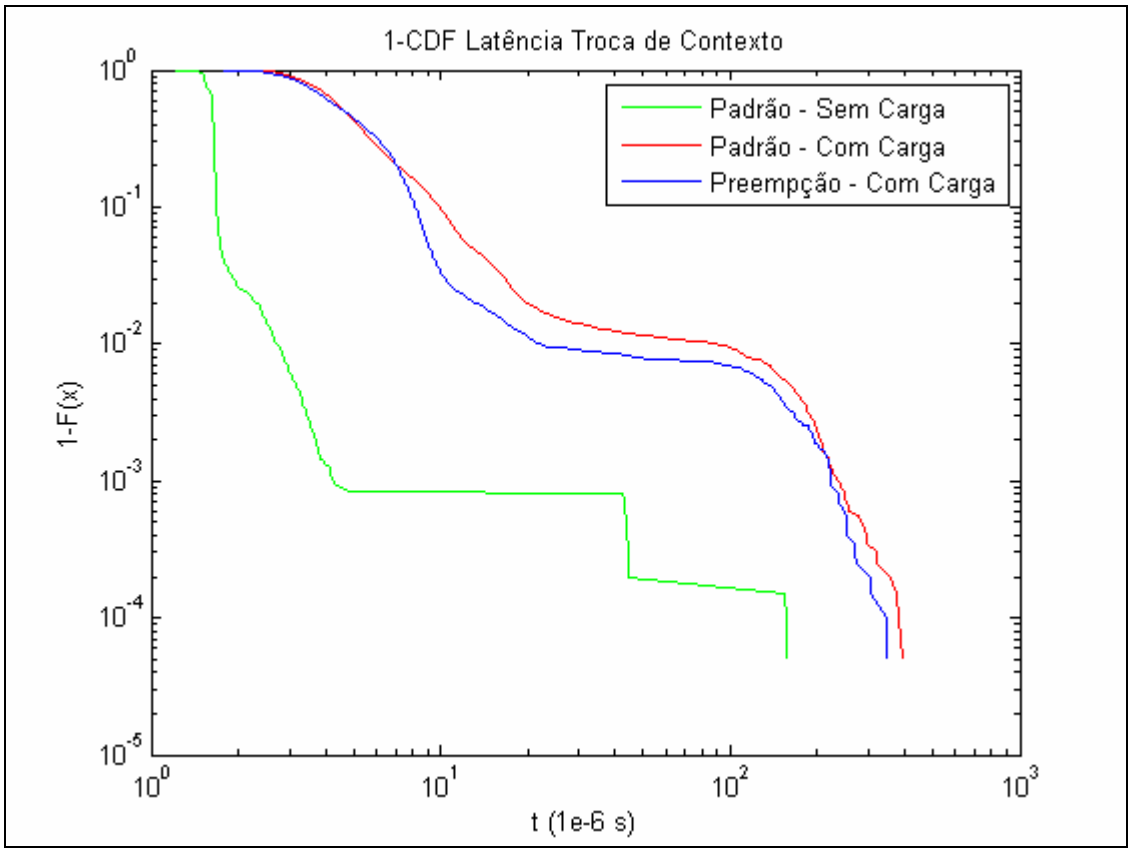

Figura K.16 – Gráfico (1-CDF) da latência de troca de contexto

Análise:

É importante observar que a latência de troca de contexto depende do número de interrupções E/S externas sendo sinalizadas e das *deferrable functions* (ver Capítulo 6, item 6.7.9).

Como na metodologia usada neste apêndice há um aumento desses dois fatores (ver item K.3.1 e item K.3.6), era de se esperar um aumento desta latência se comparada à obtida no Capítulo 6 (item 6.7.9), o que foi observado.

K.3.10 Latência Total

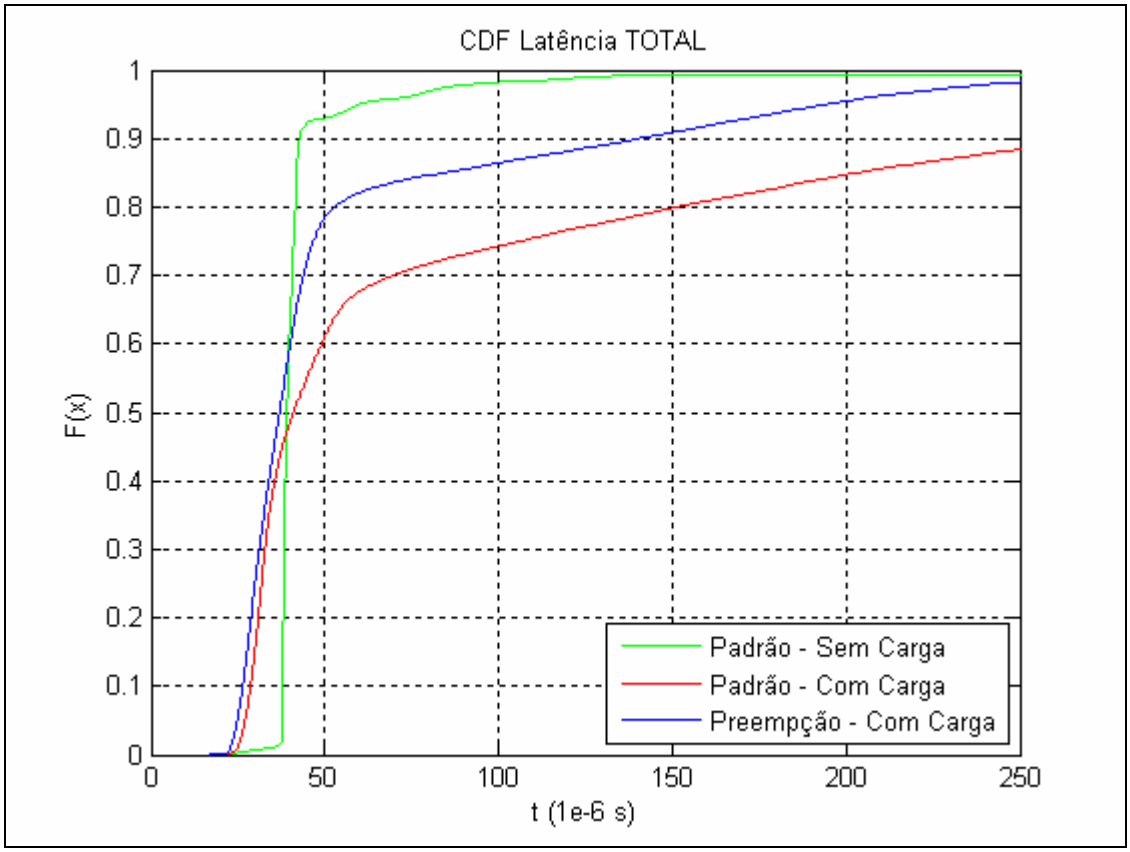

Figura K.17 – CDF da latência total
Obs.: Gráfico da CDF truncado

Tabela K.10 – Estatísticas da latência total

	Padrão Sem Carga (µs)	**Padrão Com Carga (µs)**	**Preempção Com Carga (µs)**
Média	44,6	111,9	47,90
Máximo	691,9	1,42e+3	1,39e+3
Mínimo	18,90	19,76	16,83
Mediana	39,12	40,96	37,0

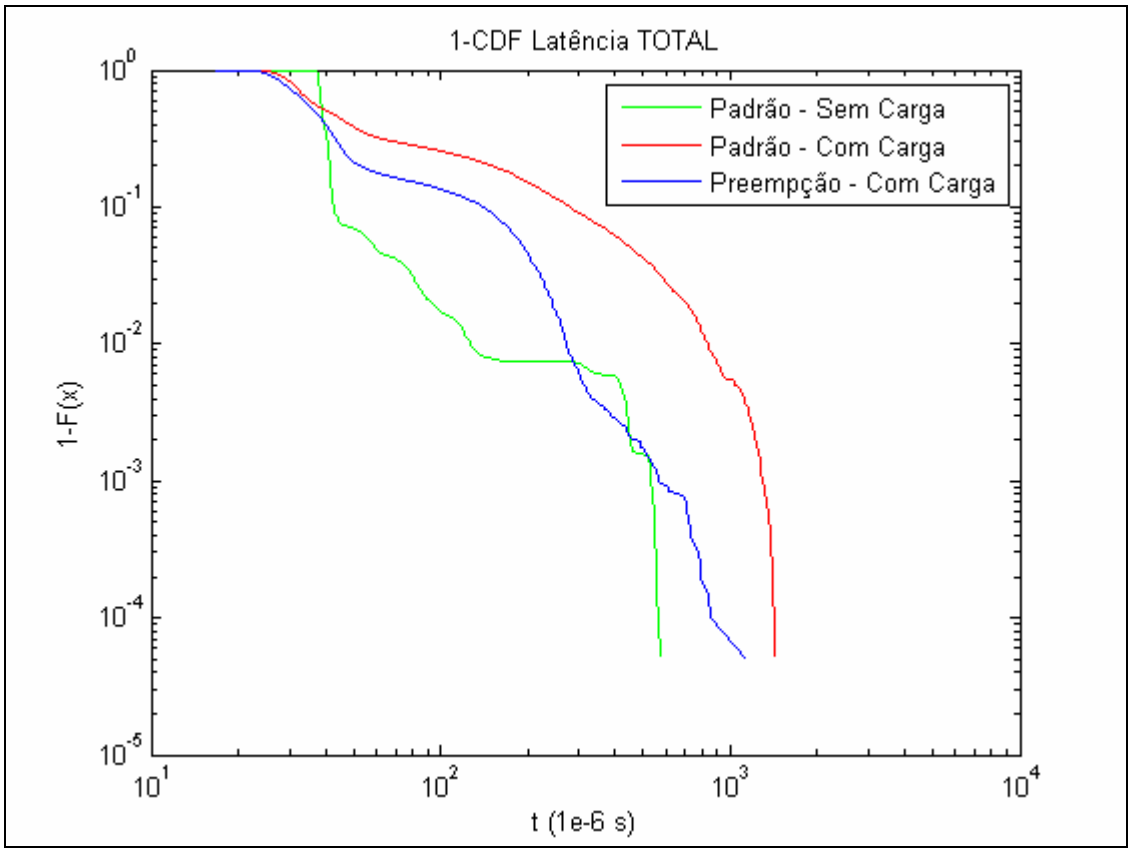

Figura K.18 – Gráfico (1-CDF) da latência total

Análise:

1- A soma das médias das componentes é aproximadamente igual à latência total média, indicando que os dados obtidos são consistentes.

2- Há um aumento da latência total com carga em relação aos valores obtidos no Capítulo 6 (item 6.7.10), como esperado.

****